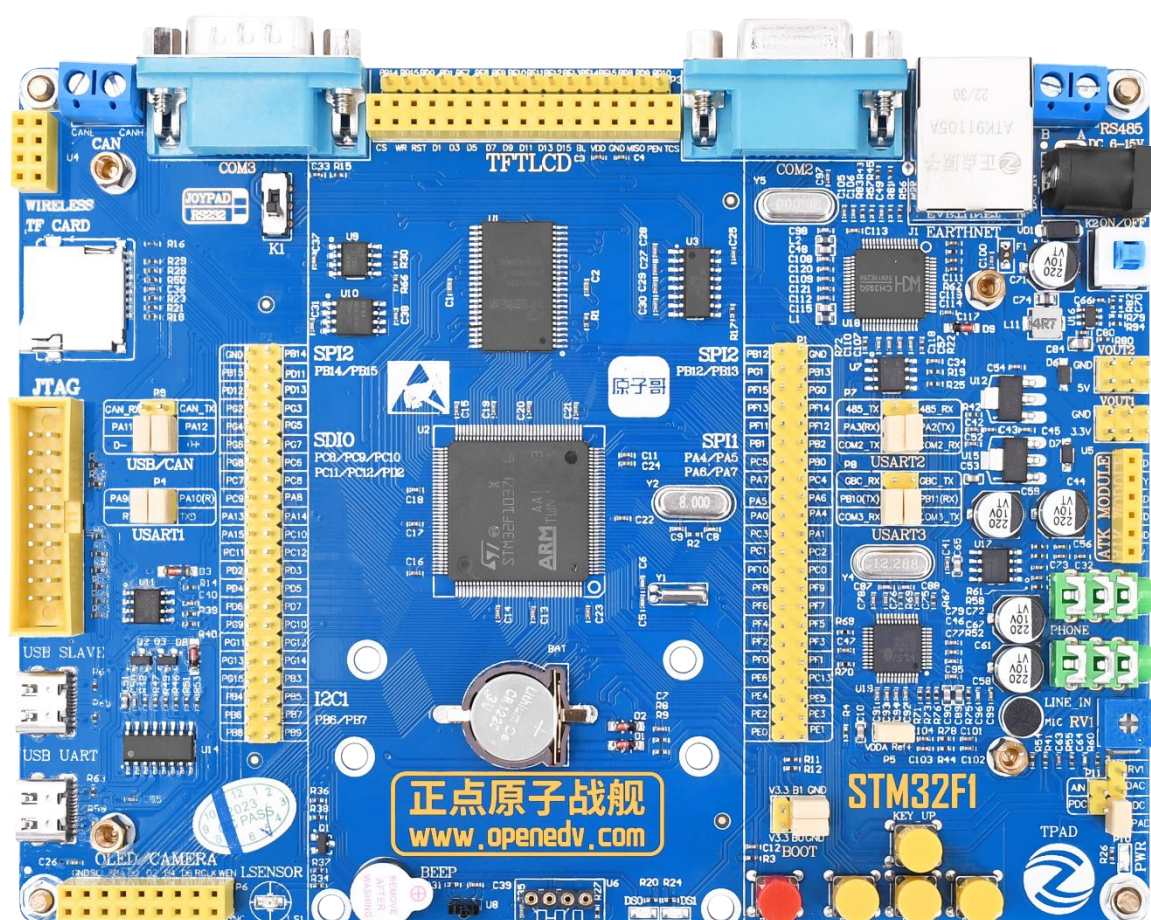


STM32F103 开发指南

V1.3

-正点原子战舰 STM32F103 开发板教程



修订历史:

| 版本 | 日期 | 修改内容 |
|------|------------|---|
| V1.0 | 2022/11/23 | 第一次发布 |
| V1.1 | 2022/12/24 | 添加第六十六章 综合测试实验 |
| V1.2 | 2023/02/14 | 修改 UC-OS II 的三个实验例程中主函数代码及其描述并新增 uc-os2_demo.c、uc-os2_demo.h 文件内容及其描述 |
| V1.3 | 2023/06/27 | 修改提高篇实验内容 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |



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

电话: 020-38271790 传真: 020-36773971

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



扫码下载“原子哥”APP



扫码关注正点原子公众号

| | |
|-------------------------------|----|
| 内容简介 | 1 |
| 第一篇 基础篇 | 2 |
| 第一章 本书学习方法 | 3 |
| 1.1 本书学习顺序 | 3 |
| 1.2 本书参考资料 | 3 |
| 1.3 本书编写规范 | 3 |
| 1.4 本书代码规范 | 4 |
| 1.5 例程资源说明 | 4 |
| 1.6 学习资料查找 | 5 |
| 1.7 给初学者的建议 | 7 |
| 第二章 STM32 简介 | 9 |
| 2.1 初识 STM32 | 9 |
| 2.2 STM32F103 资源简介 | 10 |
| 2.3 STM32F103 设计选型 | 10 |
| 2.3.1 STM32 系列 | 10 |
| 2.3.2 STM32 命名 | 11 |
| 2.3.3 STM32 选型 | 12 |
| 2.3.4 STM32 设计 | 12 |
| 第三章 开发环境搭建 | 16 |
| 3.1 常用开发工具简介 | 16 |
| 3.2 MDK 安装 | 16 |
| 3.3 仿真器驱动安装 | 17 |
| 3.4 CH340 USB 虚拟串口驱动安装 | 17 |
| 第四章 STM32 初体验 | 19 |
| 4.1 使用 MDK5 编译例程 | 19 |
| 4.2 使用串口下载程序 | 22 |
| 4.3 使用 DAP 下载与调试程序 | 25 |
| 4.3.1 使用 DAP 下载程序 | 26 |
| 4.3.2 使用 DAP 仿真调试程序 | 28 |
| 4.3.3 仿真调试注意事项 | 32 |
| 4.4 MDK5 使用技巧 | 33 |
| 4.4.1 文本美化 | 33 |
| 4.4.2 语法检测&代码提示 | 35 |
| 4.4.3 代码编辑技巧 | 37 |
| 4.4.4 其他小技巧 | 40 |
| 第五章 STM32 基础知识入门 | 43 |
| 5.1 C 语言基础知识复习 | 43 |
| 5.1.1 位操作 | 43 |
| 5.1.2 define 宏定义 | 44 |
| 5.1.3 ifdef 条件编译 | 44 |
| 5.1.4 extern 外部申明 | 44 |
| 5.1.5 typedef 类型别名 | 45 |
| 5.1.6 结构体 | 45 |
| 5.1.7 指针 | 46 |
| 5.2 寄存器基础知识 | 47 |
| 5.3 STM32F103 系统架构 | 48 |
| 5.3.1 Cortex M3 内核 & 芯片 | 48 |
| 5.3.2 STM32 系统架构 | 48 |
| 5.3.3 存储器映射 | 50 |

| | |
|--------------------------------|-----|
| 5.3.4 寄存器映射..... | 53 |
| 第六章 新建寄存器版本 MDK 工程 | 57 |
| 6.1 新建寄存器版本 MDK 工程..... | 57 |
| 6.1.1 新建工程文件夹..... | 57 |
| 6.1.2 新建一个工程框架..... | 59 |
| 6.1.3 添加文件 | 61 |
| 6.1.4 魔术棒设置..... | 65 |
| 6.1.5 添加 main.c, 并编写代码..... | 70 |
| 6.2 下载验证..... | 71 |
| 第七章 认识 HAL 库 | 73 |
| 7.1 初识 STM32 HAL 库..... | 73 |
| 7.1.1 CMSIS 标准..... | 73 |
| 7.1.2 HAL 库简介 | 74 |
| 7.1.3 HAL 库能做什么 | 76 |
| 7.2 HAL 库驱动包 | 76 |
| 7.2.1 如何获取 HAL 库固件包 | 77 |
| 7.2.2 STM32Cube 固件包分析 | 78 |
| 7.2.3 CMSIS 文件夹关键文件..... | 79 |
| 7.2.4 stdint.h 简介..... | 81 |
| 7.3 HAL 库框架结构 | 82 |
| 7.3.1 HAL 库文件夹结构 | 82 |
| 7.3.2 HAL 库文件介绍 | 82 |
| 7.4 如何使用 HAL 库..... | 86 |
| 7.4.1 学会用 HAL 库组织开发工具链 | 86 |
| 7.4.2 HAL 库的用户配置文件 | 87 |
| 7.4.3 stm32f1xx_hal.c 文件..... | 89 |
| 7.4.4 HAL 库中断处理 | 93 |
| 7.4.5 正点原子对 HAL 库用法的个性化修改..... | 94 |
| 7.5 HAL 库使用注意事项..... | 94 |
| 第八章 新建 HAL 版本 MDK 工程 | 95 |
| 8.1 新建 HAL 库版本 MDK 工程 | 95 |
| 8.1.1 新建工程文件夹..... | 95 |
| 8.1.2 新建一个工程框架..... | 99 |
| 8.1.3 添加文件 | 101 |
| 8.1.4 魔术棒设置..... | 107 |
| 8.1.5 添加 main.c, 并编写代码..... | 111 |
| 8.2 下载验证..... | 113 |
| 第九章 STM32 启动过程分析 | 114 |
| 9.1 启动模式..... | 114 |
| 9.2 启动文件分析..... | 115 |
| 9.2.1 启动文件中的一些指令..... | 115 |
| 9.2.2 启动文件代码讲解..... | 117 |
| 9.2.3 系统启动流程..... | 121 |
| 9.3 MAP 文件分析..... | 123 |
| 9.3.1 MDK 编译生成文件简介 | 123 |
| 9.3.2 map 文件分析..... | 124 |
| 第十章 STM32CUBEMX 简介 | 128 |
| 10.1 STM32CUBEMX 的作用..... | 128 |
| 10.2 安装 STM32CUBEMX | 129 |
| 10.2.1 安装 JAVA 环境 | 129 |

| | |
|-----------------------------------|-----|
| 10.2.2 安装 STM32CubeMX..... | 129 |
| 10.3 使用 STM32CUBEMX 新建工程 | 132 |
| 10.3.1 打开 STM32CubeMX..... | 132 |
| 10.3.2 下载和关联的 STM32Cube 固件包 | 133 |
| 10.3.3 新建工程 | 136 |
| 10.4 STM32CUBEMX 新建工程使用建议 | 148 |
| 第十一章 STM32 时钟配置 | 150 |
| 11.1 认识时钟树 | 150 |
| 11.1.1 时钟源 | 151 |
| 11.1.2 锁相环 PLL | 152 |
| 11.1.3 系统时钟 SYSCLK | 153 |
| 11.1.4 时钟信号输出 MCO | 155 |
| 11.2 如何修改主频..... | 155 |
| 11.2.1 STM32F1 时钟系统配置 | 155 |
| 11.2.2 STM32F1 时钟使能和配置 | 159 |
| 第十二章 SYSTEM 文件夹介绍..... | 161 |
| 12.1 DELEY 文件夹代码介绍 | 161 |
| 12.1.1 操作系统支持宏定义及相关函数..... | 162 |
| 12.1.2 delay_init 函数..... | 164 |
| 12.1.3 delay_us 函数 | 165 |
| 12.1.4 delay_ms 函数 | 166 |
| 12.1.5 HAL 库延时函数 HAL_Delay..... | 167 |
| 12.2 SYS 文件夹代码介绍 | 168 |
| 12.3 USART 文件夹代码介绍 | 168 |
| 12.3.1 printf 函数支持..... | 168 |
| 第二篇 入门篇 | 171 |
| 第十三章 跑马灯实验..... | 172 |
| 13.1 STM32F1 GPIO 简介 | 172 |
| 13.1.1 GPIO 功能模式..... | 172 |
| 13.1.2 GPIO 基本结构分析 | 172 |
| 13.1.3 GPIO 寄存器介绍 | 178 |
| 13.2 硬件设计 | 181 |
| 1. 例程功能 | 181 |
| 2. 硬件资源 | 181 |
| 3. 原理图 | 181 |
| 13.3 程序设计 | 182 |
| 13.3.1 GPIO 的 HAL 库驱动分析 | 182 |
| 13.3.2 程序流程图..... | 183 |
| 13.3.3 程序解析 | 184 |
| 13.4 下载验证..... | 186 |
| 第十四章 蜂鸣器实验..... | 187 |
| 14.1 蜂鸣器简介 | 187 |
| 14.2 硬件设计 | 187 |
| 1. 例程功能 | 187 |
| 2. 硬件资源 | 187 |
| 3. 原理图 | 188 |
| 14.3 程序设计 | 188 |
| 14.3.1 程序流程图..... | 188 |
| 14.3.2 程序解析 | 189 |
| 14.4 下载验证..... | 190 |

| | |
|----------------------------------|-----|
| 第十五章 按键输入实验..... | 191 |
| 15.1 按键与输入数据寄存器简介 | 191 |
| 15.1.1 独立按键简介..... | 191 |
| 15.1.2 GPIO 端口输入数据寄存器（IDR） | 191 |
| 15.2 硬件设计 | 192 |
| 1. 例程功能 | 192 |
| 2. 硬件资源 | 192 |
| 3. 原理图 | 192 |
| 15.3 程序设计 | 193 |
| 15.3.1 HAL_GPIO_ReadPin 函数..... | 193 |
| 15.3.2 程序流程图..... | 193 |
| 15.3.3 程序解析 | 194 |
| 15.4 下载验证..... | 197 |
| 第十六章 外部中断实验..... | 198 |
| 16.1 NVIC 和 EXTI 简介..... | 198 |
| 16.1.1 NVIC 简介..... | 198 |
| 16.1.2 EXTI 简介 | 201 |
| 16.2 硬件设计 | 203 |
| 1. 例程功能 | 203 |
| 2. 硬件资源 | 203 |
| 3. 原理图 | 204 |
| 16.3 程序设计 | 204 |
| 16.3.1 EXTI 的 HAL 库驱动 | 204 |
| 16.3.2 程序流程图..... | 205 |
| 16.3.3 程序解析 | 206 |
| 16.4 下载验证..... | 209 |
| 第十七章 串口通信实验..... | 210 |
| 17.1 串口简介 | 210 |
| 17.1.1 数据通信的基础概念..... | 210 |
| 17.1.2 串口通信协议简介..... | 212 |
| 17.1.3 STM32F1 的串口简介 | 213 |
| 17.1.4 GPIO 引脚复用功能 | 217 |
| 17.2 硬件设计 | 218 |
| 1. 例程功能 | 218 |
| 2. 硬件资源 | 218 |
| 3. 原理图 | 218 |
| 17.3 程序设计 | 219 |
| 17.3.1 USART 的 HAL 库驱动 | 219 |
| 17.3.2 程序流程图..... | 222 |
| 17.3.3 程序解析 | 222 |
| 17.4 下载验证..... | 227 |
| 第十八章 独立看门狗（IWDG）实验..... | 228 |
| 18.1 IWDG 简介..... | 228 |
| 18.1.1 IWDG 框图..... | 228 |
| 18.1.2 IWDG 寄存器..... | 228 |
| 18.2 硬件设计 | 230 |
| 1. 例程功能 | 230 |
| 2. 硬件资源 | 230 |
| 3. 原理图 | 230 |
| 18.3 程序设计 | 230 |

| | |
|--------------------------------------|-----|
| 18.3.1 IWDG 的 HAL 库驱动 | 230 |
| 18.3.2 程序流程图 | 231 |
| 18.3.3 程序解析 | 232 |
| 18.4 下载验证 | 233 |
| 第十九章 窗口门狗 (WWDG) 实验 | 234 |
| 19.1 WWDG 简介 | 234 |
| 19.1.1 WWDG 框图 | 234 |
| 19.1.2 WWDG 寄存器 | 235 |
| 19.2 硬件设计 | 237 |
| 1. 例程功能 | 237 |
| 2. 硬件资源 | 237 |
| 3. 原理图 | 237 |
| 19.3 程序设计 | 237 |
| 19.3.1 WWDG 的 HAL 库驱动 | 237 |
| 19.3.2 程序流程图 | 238 |
| 19.3.3 程序解析 | 239 |
| 19.4 下载验证 | 240 |
| 第二十章 基本定时器实验 | 241 |
| 20.1 基本定时器简介 | 241 |
| 20.1.1 基本定时器框图 | 241 |
| 20.1.2 TIM6/TIM7 寄存器 | 243 |
| 20.1.3 基本定时器中断应用 | 244 |
| 20.2 硬件设计 | 245 |
| 1. 例程功能 | 245 |
| 2. 硬件资源 | 245 |
| 3. 原理图 | 245 |
| 20.3 程序设计 | 245 |
| 20.3.1 定时器的 HAL 库驱动 | 245 |
| 20.3.2 程序流程图 | 247 |
| 20.3.3 程序解析 | 247 |
| 20.4 下载验证 | 249 |
| 第二十一章 通用定时器实验 | 250 |
| 21.1 通用定时器简介 | 250 |
| 21.2 通用定时器中断实验 | 257 |
| 21.2.1 TIM2/TIM3/TIM4/TIM5 寄存器 | 257 |
| 21.2.2 硬件设计 | 260 |
| 21.2.3 程序设计 | 260 |
| 21.2.4 下载验证 | 262 |
| 21.3 通用定时器 PWM 输出实验 | 263 |
| 21.3.1 TIM2/TIM3/TIM4/TIM5 寄存器 | 264 |
| 21.3.2 硬件设计 | 265 |
| 21.3.3 程序设计 | 265 |
| 21.3.4 下载验证 | 271 |
| 21.4 通用定时器输入捕获实验 | 272 |
| 21.4.1 TIM2/TIM3/TIM4/TIM5 寄存器 | 272 |
| 21.4.2 硬件设计 | 274 |
| 21.4.3 程序设计 | 275 |
| 21.4.4 下载验证 | 281 |
| 21.5 通用定时器脉冲计数实验 | 282 |
| 21.5.1 TIM2/TIM3/TIM4/TIM5 寄存器 | 282 |

| | |
|--------------------------------|-----|
| 21.5.2 硬件设计 | 285 |
| 21.5.3 程序设计 | 285 |
| 21.5.4 下载验证 | 290 |
| 第二十二章 高级定时器实验 | 291 |
| 22.1 高级定时器简介 | 291 |
| 22.2 高级定时器输出指定个数 PWM 实验 | 292 |
| 22.2.1 TIM1/TIM8 寄存器 | 292 |
| 22.2.2 硬件设计 | 295 |
| 22.2.3 程序设计 | 295 |
| 22.2.4 下载验证 | 300 |
| 22.3 高级定时器输出比较模式实验 | 301 |
| 22.3.1 TIM1/TIM8 寄存器 | 301 |
| 22.3.2 硬件设计 | 303 |
| 22.3.3 程序设计 | 304 |
| 22.3.4 下载验证 | 308 |
| 22.4 高级定时器互补输出带死区控制实验 | 309 |
| 22.4.1 TIM1/TIM8 寄存器 | 311 |
| 22.4.2 硬件设计 | 313 |
| 22.4.3 程序设计 | 313 |
| 22.4.4 下载验证 | 318 |
| 22.5 高级定时器 PWM 输入模式实验 | 321 |
| 22.5.1 TIM1/TIM8 寄存器 | 322 |
| 22.5.2 硬件设计 | 324 |
| 22.5.3 程序设计 | 324 |
| 22.5.4 下载验证 | 331 |
| 第二十三章 电容触摸按键实验 | 332 |
| 23.1 电容触摸按键简介 | 332 |
| 23.2 硬件设计 | 333 |
| 1. 例程功能 | 333 |
| 2. 硬件资源 | 333 |
| 3. 原理图 | 333 |
| 23.3 程序设计 | 334 |
| 23.3.1 程序流程图 | 334 |
| 23.3.2 程序解析 | 334 |
| 23.4 下载验证 | 338 |
| 第二十四章 OLED 显示实验 | 339 |
| 24.1 OLED 简介 | 339 |
| 24.1.1 硬件驱动接口模式 | 340 |
| 24.1.2 OLED 显存 | 342 |
| 24.2 硬件设计 | 344 |
| 1. 例程功能 | 344 |
| 2. 硬件资源 | 344 |
| 3. 原理图 | 345 |
| 24.3 程序设计 | 346 |
| 24.3.1 程序流程图 | 346 |
| 24.3.2 程序解析 | 346 |
| 24.4 下载验证 | 354 |
| 第二十五章 TFT-LCD (MCU 屏) 实验 | 356 |
| 25.1 TFT-LCD 简介 | 356 |
| 25.1.1 TFT-LCD 简介 | 356 |

| | |
|-----------------------------------|-----|
| 25.1.2 液晶显示控制器..... | 358 |
| 25.1.3 FSMC 简介..... | 361 |
| 25.1.4 FSMC 关联寄存器简介..... | 366 |
| 25.2 硬件设计..... | 370 |
| 1. 例程功能..... | 370 |
| 2. 硬件资源..... | 370 |
| 3. 原理图..... | 370 |
| 25.3 程序设计..... | 371 |
| 25.3.1 FSMC 和 SRAM 的 HAL 库驱动..... | 371 |
| 25.3.2 程序流程图..... | 374 |
| 25.3.2 程序解析..... | 374 |
| 25.4 下载验证..... | 385 |
| 第二十六章 USART 调试组件实验..... | 386 |
| 26.1 USART 调试组件简介..... | 386 |
| 26.2 硬件设计..... | 387 |
| 1. 例程功能..... | 387 |
| 2. 硬件资源..... | 387 |
| 26.3 程序设计..... | 387 |
| 26.3.1 程序流程图..... | 387 |
| 26.3.2 程序解析..... | 388 |
| 26.4 下载验证..... | 391 |
| 第二十七章 RTC 实时时钟实验..... | 395 |
| 27.1 RTC 时钟简介..... | 395 |
| 27.1.1 RTC 框图..... | 395 |
| 27.1.2 RTC 寄存器..... | 396 |
| 27.2 硬件设计..... | 400 |
| 1. 例程功能..... | 400 |
| 2. 硬件资源..... | 400 |
| 3. 原理图..... | 401 |
| 27.3 程序设计..... | 401 |
| 27.3.1 RTC 的 HAL 库驱动..... | 401 |
| 27.3.2 程序流程图..... | 402 |
| 27.3.3 程序解析..... | 403 |
| 27.4 下载验证..... | 409 |
| 第二十八章 低功耗实验..... | 411 |
| 28.1 电源控制 (PWR) 简介..... | 411 |
| 28.1.1 电源系统..... | 411 |
| 28.1.2 电源监控..... | 412 |
| 28.1.3 电源管理..... | 413 |
| 28.2 PVD 电压监控实验..... | 416 |
| 28.2.1 PWR 寄存器..... | 416 |
| 28.2.2 硬件设计..... | 418 |
| 28.2.3 程序设计..... | 418 |
| 28.2.4 下载验证..... | 421 |
| 28.3 睡眠模式实验..... | 422 |
| 28.3.1 EXTI 寄存器..... | 422 |
| 28.3.2 硬件设计..... | 423 |
| 28.3.3 程序设计..... | 423 |
| 28.3.4 下载验证..... | 427 |
| 28.4 停止模式实验..... | 428 |

| | |
|-----------------------------------|-----|
| 28.4.1 PWR 寄存器 | 428 |
| 28.4.2 硬件设计 | 429 |
| 28.4.3 程序设计 | 429 |
| 28.4.4 下载验证 | 431 |
| 28.5 待机模式实验 | 432 |
| 28.5.1 PWR 寄存器 | 432 |
| 28.5.2 硬件设计 | 432 |
| 28.5.3 程序设计 | 433 |
| 28.5.4 下载验证 | 435 |
| 第二十九章 DMA 实验 | 436 |
| 29.1 DMA 简介 | 436 |
| 29.1.1 DMA 框图 | 436 |
| 29.1.2 DMA 寄存器 | 438 |
| 29.2 硬件设计 | 441 |
| 1. 例程功能 | 441 |
| 2. 硬件资源 | 441 |
| 3. 原理图 | 441 |
| 29.3 程序设计 | 441 |
| 29.3.1 DMA 的 HAL 库驱动 | 441 |
| 29.3.2 程序流程图 | 444 |
| 29.3.3 程序解析 | 444 |
| 29.4 下载验证 | 447 |
| 第三十章 ADC 实验 | 449 |
| 30.1 ADC 简介 | 449 |
| 30.2 单通道 ADC 采集实验 | 455 |
| 30.2.1 ADC 寄存器 | 456 |
| 30.2.2 硬件设计 | 460 |
| 30.2.3 程序设计 | 461 |
| 30.2.4 下载验证 | 467 |
| 30.3 单通道 ADC 采集（DMA 读取）实验 | 469 |
| 30.3.1 ADC & DMA 寄存器 | 469 |
| 30.3.2 硬件设计 | 470 |
| 30.3.3 程序设计 | 471 |
| 30.3.4 下载验证 | 477 |
| 30.4 多通道 ADC 采集（DMA 读取）实验 | 478 |
| 30.4.1 ADC 寄存器 | 478 |
| 30.4.2 硬件设计 | 478 |
| 30.4.3 程序设计 | 479 |
| 30.4.4 下载验证 | 484 |
| 30.5 单通道 ADC 过采样（16 位分辨率）实验 | 486 |
| 30.5.1 ADC 寄存器 | 486 |
| 30.5.2 硬件设计 | 486 |
| 30.5.3 程序设计 | 487 |
| 30.5.4 下载验证 | 489 |
| 第三十一章 内部温度传感器实验 | 490 |
| 31.1 内部温度传感器简介 | 490 |
| 31.2 硬件设计 | 490 |
| 1. 例程功能 | 490 |
| 2. 硬件资源 | 490 |
| 3. 原理图 | 490 |

| | |
|----------------------------|-----|
| 31.3 程序设计 | 491 |
| 31.3.1 ADC 的 HAL 库驱动 | 491 |
| 31.3.2 程序流程图 | 491 |
| 31.3.3 程序解析 | 492 |
| 31.4 下载验证 | 493 |
| 第三十二章 光敏传感器实验 | 494 |
| 32.1 光敏传感器简介 | 494 |
| 32.2 硬件设计 | 494 |
| 1. 例程功能 | 494 |
| 2. 硬件资源 | 494 |
| 3. 原理图 | 495 |
| 32.3 程序设计 | 495 |
| 32.3.1 ADC 的 HAL 库驱动 | 495 |
| 32.3.2 程序流程图 | 496 |
| 32.3.3 程序解析 | 496 |
| 32.4 下载验证 | 497 |
| 第三十三章 DAC 实验 | 498 |
| 33.1 DAC 简介 | 498 |
| 33.2 DAC 输出实验 | 500 |
| 33.2.1 DAC 寄存器 | 500 |
| 33.2.2 硬件设计 | 502 |
| 33.2.3 程序设计 | 503 |
| 33.2.4 下载验证 | 508 |
| 33.3 DAC 输出三角波实验 | 509 |
| 33.3.1 DAC 寄存器 | 509 |
| 33.3.2 硬件设计 | 509 |
| 33.3.3 程序设计 | 509 |
| 33.3.4 下载验证 | 512 |
| 33.4 DAC 输出正弦波实验 | 514 |
| 33.4.1 DAC 寄存器 | 514 |
| 33.4.2 硬件设计 | 514 |
| 33.4.3 程序设计 | 514 |
| 33.4.4 下载验证 | 521 |
| 第三十四章 PWM DAC 实验 | 523 |
| 34.1 PWM DAC 技术的实现原理 | 523 |
| 34.2 硬件设计 | 525 |
| 1. 例程功能 | 525 |
| 2. 硬件资源 | 525 |
| 3. 原理图 | 525 |
| 34.3 程序设计 | 526 |
| 34.3.1 程序流程图 | 526 |
| 34.3.2 程序解析 | 526 |
| 34.4 下载验证 | 529 |
| 第三十五章 IIC 实验 | 530 |
| 35.1 IIC 及 24C02 介绍 | 530 |
| 35.1.1 IIC 简介 | 530 |
| 35.1.2 24C02 简介 | 532 |
| 35.2 硬件设计 | 534 |
| 1. 例程功能 | 534 |
| 2. 硬件资源 | 534 |

| | |
|-------------------------------|-----|
| 3. 原理图 | 534 |
| 35.3 程序设计 | 534 |
| 使用 IIC 传输数据的配置步骤 | 534 |
| 35.3.1 程序流程图 | 535 |
| 35.3.2 程序解析 | 535 |
| 35.4 下载验证 | 545 |
| 第三十六章 SPI 实验 | 547 |
| 36.1 SPI 及 NOR FLASH 介绍 | 547 |
| 36.1.1 SPI 介绍 | 547 |
| 36.1.2 NOR FLASH 简介 | 551 |
| 36.2 硬件设计 | 554 |
| 1. 例程功能 | 554 |
| 2. 硬件资源 | 554 |
| 3. 原理图 | 554 |
| 36.3 程序设计 | 555 |
| 36.3.1 SPI 的 HAL 库驱动 | 555 |
| 36.3.2 程序流程图 | 557 |
| 36.3.3 程序解析 | 557 |
| 36.4 下载验证 | 566 |
| 第三十七章 485 实验 | 567 |
| 37.1 485 简介 | 567 |
| 37.2 硬件设计 | 568 |
| 1. 例程功能 | 568 |
| 2. 硬件资源 | 568 |
| 3. 原理图 | 568 |
| 37.3 程序设计 | 569 |
| 37.3.1 RS485 的 HAL 库驱动 | 569 |
| 37.3.2 程序流程图 | 570 |
| 37.3.3 程序解析 | 570 |
| 37.4 下载验证 | 574 |
| 第三十八章 CAN 通讯实验 | 576 |
| 38.1 CAN 总线简介 | 576 |
| 38.1.1 CAN 简介 | 576 |
| 38.1.2 CAN 协议 | 578 |
| 38.1.3 CAN 寄存器 | 587 |
| 38.2 硬件设计 | 591 |
| 1. 例程功能 | 591 |
| 2. 硬件资源 | 591 |
| 3. 原理图 | 591 |
| 38.3 程序设计 | 592 |
| 38.3.1 CAN 的 HAL 库驱动 | 592 |
| CAN 的初始化配置步骤 | 595 |
| 38.3.2 程序流程图 | 596 |
| 38.3.3 程序解析 | 596 |
| 38.4 下载验证 | 601 |
| 第三十九章 触摸屏实验 | 603 |
| 39.1 触摸屏简介 | 603 |
| 39.1.1 电阻式触摸屏 | 603 |
| 39.1.2 电容式触摸屏 | 605 |
| 39.1.3 触摸控制原理 | 607 |

| | |
|-------------------------------|-----|
| 39.2 硬件设计 | 608 |
| 1. 例程功能 | 608 |
| 2. 硬件资源 | 608 |
| 3. 原理图 | 609 |
| 39.3 程序设计 | 609 |
| 39.3.1 HAL 库驱动 | 609 |
| 39.3.2 程序流程图 | 610 |
| 39.3.3 程序解析 | 610 |
| 39.4 下载验证 | 625 |
| 第四十章 红外遥控实验 | 626 |
| 40.1 红外遥控简介 | 626 |
| 40.1.1 红外遥控技术介绍 | 626 |
| 40.1.2 红外器件特性 | 626 |
| 40.1.3 红外编解码协议介绍 | 626 |
| 40.2 硬件设计 | 628 |
| 1. 例程功能 | 628 |
| 2. 硬件资源 | 628 |
| 3. 原理图 | 628 |
| 40.3 程序设计 | 629 |
| 红外遥控配置步骤 | 629 |
| 40.3.1 程序流程图 | 629 |
| 40.3.2 程序解析 | 630 |
| 40.4 下载验证 | 635 |
| 第四十一章 游戏手柄实验 | 636 |
| 41.1 游戏手柄简介 | 636 |
| 41.2 硬件设计 | 637 |
| 1. 例程功能 | 637 |
| 2. 硬件资源 | 637 |
| 41.3 程序设计 | 638 |
| 41.3.1 程序流程图 | 638 |
| 41.3.2 程序解析 | 639 |
| 41.4 下载验证 | 641 |
| 第四十二章 DS18B20 数字温度传感器实验 | 642 |
| 42.1 DS18B20 及工作时序简介 | 642 |
| 42.1.1 DS18B20 简介 | 642 |
| 42.1.2 DS18B20 工作时序简介 | 642 |
| 42.2 硬件设计 | 644 |
| 1. 例程功能 | 644 |
| 2. 硬件资源 | 644 |
| 3. 原理图 | 644 |
| 42.3 程序设计 | 645 |
| DS18B20 配置步骤 | 645 |
| 42.3.1 程序流程图 | 645 |
| 42.3.2 程序解析 | 646 |
| 42.4 下载验证 | 651 |
| 第四十三章 DHT11 数字温湿度传感器 | 652 |
| 43.1 DHT11 及工作时序简介 | 652 |
| 43.1.1 DHT11 简介 | 652 |
| 43.1.2 DHT11 工作时序简介 | 652 |
| 43.2 硬件设计 | 654 |

| | |
|----------------------------------|-----|
| 1. 例程功能 | 654 |
| 2. 硬件资源 | 654 |
| 3. 原理图 | 654 |
| 43.3 程序设计 | 654 |
| DHT11 配置步骤 | 655 |
| 43.3.1 程序流程图 | 655 |
| 43.3.2 程序解析 | 655 |
| 43.4 下载验证 | 659 |
| 第四十四章 无线通信实验 | 661 |
| 44.1 NRF24L01 无线模块介绍 | 661 |
| 44.1.1 NRF24L01 简介 | 661 |
| 44.1.2 NRF24L01 工作模式介绍 | 662 |
| 44.1.3 NRF24L01 寄存器 | 663 |
| 44.2 硬件设计 | 665 |
| 1. 例程功能 | 665 |
| 2. 硬件资源 | 665 |
| 3. 原理图 | 666 |
| 44.3 程序设计 | 666 |
| NRF24L01 配置步骤 | 666 |
| 44.3.1 程序流程图 | 667 |
| 44.3.2 程序解析 | 667 |
| 44.4 下载验证 | 675 |
| 第四十五章 FLASH 模拟 EEPROM 实验 | 677 |
| 45.1 STM32 FLASH 简介 | 677 |
| 45.1.1 闪存的读取 | 678 |
| 45.1.2 闪存的编程和擦除 | 678 |
| 45.1.3 FLASH 寄存器 | 680 |
| 45.2 硬件设计 | 681 |
| 1. 例程功能 | 681 |
| 2. 硬件资源 | 682 |
| 45.3 程序设计 | 682 |
| 45.3.1 FLASH 的 HAL 库驱动 | 682 |
| 45.3.2 程序流程图 | 684 |
| 45.3.3 程序解析 | 684 |
| 45.4 下载验证 | 688 |
| 第四十六章 摄像头实验 | 690 |
| 46.1 OV7725 模块简介 | 690 |
| 46.1.1 正点原子 OV7725 模块 | 690 |
| 46.1.2 串行摄像头控制总线 (SCCB) 简介 | 692 |
| 46.1.3 输出时序说明 | 693 |
| 46.1.4 图像数据存储和读取说明 | 695 |
| 46.2 硬件设计 | 697 |
| 1. 例程功能 | 697 |
| 2. 硬件资源 | 697 |
| 46.3 程序设计 | 697 |
| OV7725 模块驱动步骤 | 697 |
| 46.3.1 程序流程图 | 698 |
| 46.3.2 程序解析 | 698 |
| 46.4 下载验证 | 706 |
| 第四十七章 SRAM 实验 | 708 |

| | |
|--------------------------------|-----|
| 47.1 存储器简介 | 708 |
| 47.2 SRAM 方案简介 | 709 |
| 47.3 硬件设计 | 711 |
| 1. 例程功能 | 711 |
| 2. 硬件资源 | 711 |
| 3. 原理图 | 711 |
| 47.4 程序设计 | 712 |
| 使用 SRAM 的配置步骤 | 712 |
| 47.4.1 程序流程图 | 713 |
| 47.4.2 程序解析 | 713 |
| 47.5 下载验证 | 717 |
| 第三篇 提高篇 | 719 |
| 第四十八章 内存管理实验 | 720 |
| 48.1 内存管理简介 | 720 |
| 48.2 硬件设计 | 721 |
| 1. 例程功能 | 721 |
| 2. 硬件资源 | 721 |
| 48.3 程序设计 | 722 |
| 48.3.1 程序流程图 | 722 |
| 48.3.2 程序解析 | 722 |
| 48.4 下载验证 | 727 |
| 第四十九章 SD 卡实验 | 730 |
| 49.1 SD 卡简介 | 730 |
| 49.1.1 SD 物理结构 | 730 |
| 49.1.2 命令和响应 | 732 |
| 49.1.3 SD 卡的操作模式 | 733 |
| 49.2 SDIO 接口简介 | 736 |
| 49.2.1 SDIO 主要功能及框图 | 736 |
| 49.2.2 SDIO 的时钟 | 738 |
| 49.2.3 SDIO 的命令与响应 | 738 |
| 49.2.4 SDIO 相关寄存器介绍 | 740 |
| 49.2.5 SDIO 模式下的 SD 卡初始化 | 745 |
| 49.3 硬件设计 | 747 |
| 1. 例程功能 | 747 |
| 2. 硬件资源 | 747 |
| 3. 原理图 | 748 |
| 49.4 程序设计 | 748 |
| 49.4.1 SD 卡的 HAL 库驱动 | 748 |
| 49.4.2 程序流程图 | 751 |
| 49.4.3 程序解析 | 751 |
| 49.5 下载验证 | 755 |
| 第五十章 FATFS 实验 | 757 |
| 50.1 FATFS 简介 | 757 |
| 50.2 硬件设计 | 761 |
| 1. 例程功能 | 761 |
| 2. 硬件资源 | 761 |
| 50.3 程序设计 | 761 |
| 50.3.1 程序流程图 | 762 |
| 50.3.2 程序解析 | 763 |
| 50.4 下载验证 | 771 |

| | |
|----------------------------------|-----|
| 第五十一章 汉字显示实验..... | 772 |
| 51.1 汉字显示原理简介..... | 772 |
| 51.1.1 字符编码介绍..... | 772 |
| 51.1.2 汉字字库简介..... | 773 |
| 51.1.3 汉字显示原理..... | 775 |
| 51.1.4 ffontcode.c 优化（补充说明）..... | 776 |
| 51.2 硬件设计..... | 779 |
| 1. 例程功能..... | 779 |
| 2. 硬件资源..... | 780 |
| 51.3 程序设计..... | 780 |
| 51.3.1 程序流程图..... | 780 |
| 51.3.2 程序解析..... | 781 |
| 51.4 下载验证..... | 789 |
| 第五十二章 图片显示实验..... | 790 |
| 52.1 图片格式介绍..... | 790 |
| 52.1.1 BMP 编码简介..... | 790 |
| 52.1.2 JPEG 编码简介..... | 790 |
| 52.1.2 GIF 编码简介..... | 791 |
| 52.2 硬件设计..... | 791 |
| 1. 例程功能..... | 791 |
| 2. 硬件资源..... | 792 |
| 52.3 程序设计..... | 792 |
| 52.3.1 程序流程图..... | 792 |
| 52.3.2 程序解析..... | 793 |
| 52.4 下载验证..... | 799 |
| 第五十三章 照相机实验..... | 802 |
| 53.1 BMP 编码简介..... | 802 |
| 53.2 硬件设计..... | 804 |
| 1. 例程功能..... | 804 |
| 2. 硬件资源..... | 804 |
| 53.3 程序设计..... | 805 |
| 53.3.1 程序流程图..... | 805 |
| 53.3.2 程序解析..... | 805 |
| 53.4 下载验证..... | 810 |
| 第五十四章 音乐播放器实验..... | 811 |
| 54.1 VS1053 简介..... | 811 |
| 54.2 硬件设计..... | 815 |
| 1. 例程功能..... | 815 |
| 2. 硬件资源..... | 815 |
| 54.3 程序设计..... | 817 |
| 54.3.2 程序流程图..... | 817 |
| 54.3.2 程序解析..... | 818 |
| 54.4 下载验证..... | 823 |
| 第五十五章 录音机实验..... | 824 |
| 55.1 WAV 格式简介..... | 824 |
| 1. 激活 PCM 录音..... | 826 |
| 2. 读取 PCM 数据..... | 827 |
| 54.2 硬件设计..... | 828 |
| 1. 例程功能..... | 828 |
| 2. 硬件资源..... | 828 |

| | |
|-------------------------------|-----|
| 55.3 程序设计 | 828 |
| 55.3.1 程序流程图 | 828 |
| 55.3.2 程序解析 | 829 |
| 55.4 下载验证 | 834 |
| 第五十六章 DSP 测试实验 | 836 |
| 56.1 DSP 简介与环境搭建 | 836 |
| 56.1.1 STM32F103 DSP 简介 | 836 |
| 56.1.2 DSP 库运行环境搭建 | 837 |
| 56.2 硬件设计 | 839 |
| 1. 例程功能 | 839 |
| 2. 硬件资源 | 839 |
| 56.3 程序设计 | 839 |
| 56.3.1 DSP BasicMath 测试 | 839 |
| 56.3.2 DSP FFT 测试 | 842 |
| 56.4 下载验证 | 845 |
| 第五十七章 手写识别实验 | 847 |
| 57.1 手写识别简介 | 847 |
| 57.2 硬件设计 | 848 |
| 1. 例程功能 | 848 |
| 2. 硬件资源 | 848 |
| 57.3 程序设计 | 849 |
| 57.3.1 程序流程图 | 849 |
| 57.3.2 程序解析 | 849 |
| 57.4 下载验证 | 854 |
| 第五十八章 T9 拼音输入法实验 | 856 |
| 58.1 拼音输入法简介 | 856 |
| 58.2 硬件设计 | 856 |
| 1. 例程功能 | 856 |
| 2. 硬件资源 | 857 |
| 58.3 程序设计 | 857 |
| 58.3.1 程序流程图 | 857 |
| 58.3.2 程序解析 | 858 |
| 58.4 下载验证 | 863 |
| 第五十九章 串口 IAP 实验 | 865 |
| 59.1 IAP 简介 | 865 |
| 59.2 硬件设计 | 871 |
| 1. 例程功能 | 871 |
| 2. 硬件资源 | 871 |
| 59.3 程序设计 | 872 |
| 59.3.1 程序流程图 | 872 |
| 59.3.2 程序解析 | 872 |
| 59.4 下载验证 | 876 |
| 第六十章 USB 读卡器实验 | 878 |
| 60.1 USB 简介 | 878 |
| 60.1.1 USB 简介 | 878 |
| 60.1.2 STM32F1 的 USB 特性 | 880 |
| 60.2 硬件设计 | 882 |
| 1. 例程功能 | 882 |
| 2. 硬件资源 | 882 |
| 60.3 程序设计 | 882 |

| | |
|---|-----|
| 60.3.1 程序流程图..... | 886 |
| 60.3.2 usbd_stroage 驱动 | 887 |
| 60.3.3 usbd_conf 驱动 | 891 |
| 60.3.4. main.c 代码..... | 892 |
| 60.5 下载验证..... | 896 |
| 第六十一章 USB 虚拟串口实验..... | 897 |
| 61.1 USB 虚拟串口简介 | 897 |
| 61.2 硬件设计 | 897 |
| 1. 例程功能 | 897 |
| 2. 硬件资源 | 897 |
| 61.3 程序设计 | 898 |
| 61.3.1 程序流程图..... | 900 |
| 61.3.2 usbd_cdc_interface 驱动..... | 901 |
| 61.4 下载验证..... | 905 |
| 第六十二章 网络通讯实验..... | 908 |
| 第六十三章 UCOSII 实验 1-任务调度..... | 909 |
| 63.1 嵌入式实时操作系统介绍..... | 909 |
| 63.1.1 裸机系统和多任务系统的区别..... | 909 |
| 63.1.2 UCOSII 介绍 | 910 |
| 63.1.3 任务定义 | 911 |
| 63.1.4 任务调度 | 912 |
| 63.2 硬件设计 | 912 |
| 1. 例程功能 | 912 |
| 2. 硬件资源 | 912 |
| 3. 原理图 | 912 |
| 63.3 程序设计 | 913 |
| 63.3.1 UCOSII 驱动函数 | 913 |
| 63.3.2 程序流程图..... | 915 |
| 63.3.3 程序解析 | 915 |
| 63.4 下载验证..... | 921 |
| 第六十四章 UCOSII 实验 2-信号量和邮箱..... | 922 |
| 64.1 UCOSII 信号量和邮箱简介 | 922 |
| 64.2 硬件设计 | 923 |
| 1. 例程功能 | 923 |
| 2. 硬件资源 | 924 |
| 3. 原理图 | 924 |
| 64.3 程序设计 | 924 |
| 64.3.1 信号量函数..... | 924 |
| 64.3.2 程序流程图..... | 928 |
| 64.3.3 程序解析 | 928 |
| 64.4 下载验证..... | 937 |
| 第六十五章 UCOSII 实验 3-消息队列、信号量集和软件定时器 | 938 |
| 65.1 UCOSII 消息队列、信号量集和软件定时器简介..... | 938 |
| 65.1.1 消息队列 | 938 |
| 65.1.2 信号量集 | 939 |
| 65.1.3 软件定时器..... | 941 |
| 65.2 硬件设计 | 943 |
| 1. 例程功能 | 943 |
| 2. 硬件资源 | 943 |
| 3. 原理图 | 943 |

| | |
|---------------------------|------|
| 65.3 程序设计 | 944 |
| 65.3.1 UCOSII 程序流程图 | 944 |
| 65.3.2 程序流程图 | 947 |
| 65.3.3 程序解析 | 949 |
| 65.4 下载验证 | 961 |
| 第六十六章 综合测试实验 | 962 |
| 66.1 综合测试实验简介 | 962 |
| 66.2 综合测试实验详解 | 962 |
| 66.2.1 电子图书 | 966 |
| 66.2.2 数码相框 | 967 |
| 66.2.3 音乐播放 | 969 |
| 66.2.4 TOM 猫 | 971 |
| 66.2.5 时钟 | 972 |
| 66.2.6 系统设置 | 973 |
| 66.2.7 FC 游戏机 | 983 |
| 66.2.8 记事本 | 987 |
| 66.2.9 运行器 | 989 |
| 66.2.10 手写画笔 | 990 |
| 66.2.11 照相机 | 993 |
| 66.2.12 录音机 | 996 |
| 66.2.13 USB 连接 | 998 |
| 66.2.14 网络通信 | 999 |
| 66.2.15 无线传书 | 1002 |
| 66.2.16 计算器 | 1004 |
| 66.2.17 拨号 | 1006 |
| 66.2.18 应用中心 | 1008 |
| 66.2.19 电压表 | 1010 |

内容简介

本书将由浅入深，带领大家学习 STM32F103 的各个功能，为您开启 STM32F103 的学习之旅。

本书总共分为三篇：

- 1, 基础篇，主要介绍 STM32F103 的基础入门知识，包括 STM32F103 介绍、软件安装、新建工程、时钟系统介绍、SYSTEM 文件夹介绍等。必须好好学习并掌握；
- 2, 入门篇，主要介绍 STM32F103 常用外设的使用，包括 GPIO、定时器、AD/DA、DMA、触摸液晶屏等。必须好好学习并掌握；
- 3, 提高篇，主要介绍 STM32F103 较难一点外设的使用及一些高级例程，包括：SDIO、USB、内存管理、文件系统、图片解码、OS 入门等。这些例程难度相对高一点，大家一定要沉下心来，戒骄戒躁，循序渐进的学习。对于一些较难的知识点，我们只需要知道怎么用就可以了，并不需要详细研究具体实现过程，比如文件系统、图片解码、USB 协议栈、OS 实现原理等；

本书为正点原子战舰 STM32F103 开发板的配套教程，在开发板配套的光盘里面，有详细原理图以及所有实例的完整代码，这些代码都有详细的注释，所有源码都经过我们严格测试，不会有任何警告和错误，另外，源码有我们生成好的 hex 文件，大家只需要通过仿真器下载到开发板即可看到实验现象，亲自体验实验过程。

本书并没有详细介绍战舰 STM32F103 的硬件，硬件部分内容我们单独编写了一个文档，请参考：《战舰 V4 硬件参考手册.pdf》进行学习。

本书不仅非常适合广大学生和电子爱好者学习 STM32F103，其大量的实验以及详细的解说，也是公司产品开发的不二参考。

第一篇 基础篇

万事开头难，如果打好了基础，后面学习就事半功倍了！本篇将详细介绍 STM32F103 学习的基础知识，包括：环境搭建、STM32 入门知识、新建工程、HAL 库介绍、启动过程分析、时钟系统、SYSTEM 文件夹介绍等部分。学好了这些基础知识，在后面的例程学习部分，将会有非常大的帮助，能极大的提高大家的学习效率。

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

如果您已经学过 STM32 了，本篇内容则可以挑选着学习。

本篇将分为如下章节：

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

第一章 本书学习方法

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

本章将分为如下几个小节：

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

1.1 本书学习顺序

为了让大家更好的学习和使用本书，我们做了以下几点考虑：

- 1，坚持循序渐进的思路编写，从基础到入门，从简单到复杂。
- 2，将知识进行分类介绍，简化学习过程，包括：基础篇、入门篇、提高篇。
- 3，将硬件介绍独立成一个文档《战舰 V4 硬件参考手册.pdf》，本书着重介绍软件知识。

因此，大家在学习本书的时候，我们建议：先通读一遍《战舰 V4 硬件参考手册.pdf》，对开发板的硬件资源有个大概了解，然后从基础篇开始，再到入门篇，最后学习提高篇，循序渐进，逐一攻克。

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

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

1.2 本书参考资料

本书的主要参考资料有以下两份文档：

《STM32F10xxx 参考手册_V10（中文版）.pdf》

《Cortex-M3 权威指南》中文版（宋岩 译）

前者是 ST 官方针对 STM32 的一份通用参考资料，重点介绍 STM32 内部资源及使用，寄存器描述等，内容翔实，但是没有实例，也没有对 Cortex-M3 构架进行多少介绍，读者只能根据自己对书本的理解来编写相关代码。

后者是专门介绍 Cortex-M3 构架的书，有简短的实例，但没有专门针对 STM32 的介绍。所以，在学习 STM32 的时候，必须结合这份资料来看。

另外，由于 STM32F103 的中文版是 V10 版本的，而最新的 STM32F103 英文版参考资料已经是 V20 的了，所以大家在遇到一些有问题/矛盾的地方，可以参考最新的英文版参考手册：

《STM32 英文参考手册》V20

这几份参考文档都在我们提供的光盘资料里面可以找到，路径：A 盘→8，STM32 参考资料 文件夹下可以找到。

1.3 本书编写规范

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

基础篇，共 12 章，主要是一些基础知识介绍，包括开发环境搭建、新建工程、HAL 库介绍、时钟树介绍、SYSTEM 文件夹介绍等，这些章节在结构上没有共性，但是互相有关联，有

一个集成的关系在里面，即：必须先学了前面的知识，才好学习后面的知识点。

入门篇和提高篇，共五十四章，详细介绍了 STM32F103 每一个外设的使用方法 & 驱动代码，并且还介绍了一些非常实用的程序代码（纯软件例程），如：内存管理、文件系统读写、拼音输入法、手写识别、图片解码、IAP 等。这部分内容占了本书的绝大部分篇幅，而且这些章节在结构上都比较有共性，一般分为 4 个部分，如下：

- 1，外设功能介绍
- 2，硬件设计
- 3，程序设计
- 4，下载验证

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

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

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

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

1.4 本书代码规范

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

总结几个规范的关键点：

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

1.5 例程资源说明

战舰 V4 TM32F103 开发板提供的标准例程多达 69 个，提供：寄存器和 HAL 库两个版本的代码（本手册以 HAL 库版本例程做为介绍，我们不再提供寄存器版本文档教程，寄存器版本例程仅供大家参考学习）。我们提供的这些例程，基本都是原创，拥有非常详细的注释，代码风格统一、循序渐进，非常适合初学者入门。

战舰 V4 开发板的例程列表如表 1.5.1 所示：

| 编号 | 实验名字 | 编号 | 实验名字 |
|-----|-----------|----|------------|
| 1 | 跑马灯实验 | 22 | PWM DAC 实验 |
| 2 | 蜂鸣器实验 | 23 | IIC 实验 |
| 3 | 按键输入实验 | 24 | SPI 实验 |
| 4 | 外部中断实验 | 25 | 485 实验 |
| 5 | 串口通信实验 | 26 | CAN 实验 |
| 6 | 独立看门狗实验 | 27 | 触摸屏实验 |
| 7 | 窗口看门狗实验 | 28 | 红外遥控器实验 |
| 8-1 | 基本定时器中断实验 | 29 | 游戏手柄实验 |

| | | | |
|------|-------------------|------|-----------------------------|
| 9-1 | 通用定时器中断实验 | 30 | DS18B20 数字温度传感器实验 |
| 9-2 | 通用定时器 PWM 输出实验 | 31 | DHT11 数字温湿度传感器实验 |
| 9-3 | 通用定时器输入捕获实验 | 32 | 无线通信实验 |
| 9-4 | 通用定时器脉冲计数实验 | 33 | FLASH 模拟 EEPROM 实验 |
| 10-1 | 高级定时器输出指定个数 | 34 | 摄像头实验 |
| 10-2 | 高级定时器输出比较模式实 | 35 | 外部 SRAM 实验 |
| 10-3 | 高级定时器互补输出带死区 | 36 | 内存管理实验 |
| 10-4 | 高级定时器 PWM 输入模式 | 37 | SD 卡实验 |
| 11 | 电容触摸按键实验 | 38 | FATFS 实验 |
| 12 | OLED 实验 | 39 | 汉字显示实验 |
| 13 | TFTLCD (MCU 屏) 实验 | 40 | 图片显示实验 |
| 14 | USMART 调试实验 | 41 | 照相机实验 |
| 15 | RTC 实验 | 42 | 音乐播放器实验 |
| 16-1 | PVD 电压监控实验 | 43 | 录音机实验 |
| 16-2 | 睡眠模式实验 | 44-1 | DSP BasicMath 测试实验 |
| 16-3 | 停止模式实验 | 44-2 | DSP FFT 测试实验 |
| 16-4 | 待机模式实验 | 45 | 手写识别实验 |
| 17 | DMA 实验 | 46 | T9 拼音输入法实验 |
| 18-1 | 单通道 ADC 采集实验 | 47 | 串口 IAP 实验 |
| 18-2 | 单通道 ADC 采集 (DMA 读 | 48 | USB 读卡器(Slave)实验 |
| 18-3 | 多通道 ADC 采集 (DMA 读 | 49 | USB 虚拟串口(Slave)实验 |
| 18-4 | 单通道 ADC 过采样 (16 位 | 50 | 网络通信实验 |
| 19 | 内部温度传感器实验 | 51 | UCOSII 实验 1-任务调度 |
| 20 | 光敏传感器实验 | 52 | UCOSII 实验 2-信号量和邮箱 |
| 21-1 | DAC 输出实验 | 53 | UCOSII 实验 3-消息队列、信号量集和软件定时器 |
| 21-2 | DAC 输出三角波实验 | 54 | 综合测试实验 |
| 21-3 | DAC 输出正弦波实验 | | |

表 1.5.1 战舰 V4 开发板例程表

从上表可以看出，正点原子战舰 V4 开发板的例程基本上涵盖了 STM32F103ZET6 芯片的所有内部资源，并且外扩展了很多有价值的例程，比如：FLASH 模拟 EEPROM 实验、USMART 调试实验、UCOSII 实验、内存管理实验、IAP 实验、拼音输入法实验、手写识别实验、综合实验等。

而且从上表可以看出，例程安排是循序渐进的，首先从最基础的跑马灯开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握。所以，正点原子战舰 V4 STM32F103 开发板是非常适合初学者的。当然，对于想深入了解 STM32F103 内部资源的朋友，正点原子战舰开发板也绝对是一个不错的选择。

1.6 学习资料查找

学习资料包括三个方面：

1，ST 官方的学习资料

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

www.stmcu.org.cn 是 ST 中文社区，里面的资料全部由 ST 中国区的人负责更新和整理，包

含了所有 ST 公司的 MCU 资料，比如：STM32F1 最新的芯片文档（参考手册、数据手册、勘误手册、编程手册等）、软件资源（固件库、配置工具、PC 软件等）、硬件资源（各种官方评估板）等，如图 1.6.1 所示：



图 1.6.1 STM32F1 相关资料（stm32cu.org）

www.st.com 是 ST 官网，ST 最新最全的资料，一般都是放在该网站，ST 中文社区的资料，一般都是从 ST 官网搬过来的，所以如果你想找最新的 STM32 官方资料，应该在 ST 官网找。对于初学者，一般从 ST 中文社区获取 ST 官方资料就可以了。ST 官网的 STM32F103 资料页面如图 1.6.2 所示（注意：默认是英文语言，需要在网页右上角设置成中文）：



图 1.6.2 STM32F103 官网资料（st.com）

- ① STM32F103 的硬件相关资源，在：产品→微控制器→STM32 ARM Cortex 32 位微控制器→STM32 主流 MCU→STM32F1 系列→STM32F103→资源 路径下面可以找到。
- ② STM32F103 的软件相关资源，在：工具与软件→嵌入式软件→微控制器软件→STM32 微控制器软件 路径下面可以找到。

2. 正点原子的学习资料

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



图 1.6.3 正点原子文档中心

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

3, 正点原子论坛

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

在学习过程中，我们难免会遇到一些问题，有任何问题，大家都可以先去开源电子网搜索一下，基本上你能遇到的问题，我们论坛都有人问过了，所以可以很方便的找到一些参考解决方法。如果实在找不到，你也可以在论坛提问，每天原子哥都会在上面给大家做解答。

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

1.7 给初学者的建议

对于学习 STM32，这里我给大家提以下三点建议：

1, 准备一款合适的开发板（强烈建议配仿真器）

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

另外，开发板在精不在多，学好一款，基本上就够用了。

2, 两本参考资料，即《STM32F10xxx 参考手册_V10（中文版）.pdf》和《Cortex-M3 权威指南》。

这两个手册在本章 1.2 节有过介绍，对于我们学习 STM32 和了解 Cortex M3 内核非常有帮助，是学习 STM32 的必备资料，因此初学者尤其要多看这两个手册。

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

3, 戒骄戒躁，勤思敏行。

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

我就曾经遇到一个问题半个月都没解决的情况，但是这半个月我尝试并掌握了很多解决问题的方法，最终解决问题的时候，其实带来的收获远远大于问题本身。所以不要遇到问题就认

怂，就想问别人，问老师，先尝试自己解决一下，比如花个十天半个月去解决一个问题，我相信你也会有很多收获。

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

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

第二章 STM32 简介

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

本章将分为如下几个小节:

2.1 初识 STM32

2.2 STM32F103 资源简介

2.3 STM32F103 设计选型

2.1 初识 STM32

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

战舰开发板使用的 STM32F103ZET6 芯片如图 2.1.1 所示:

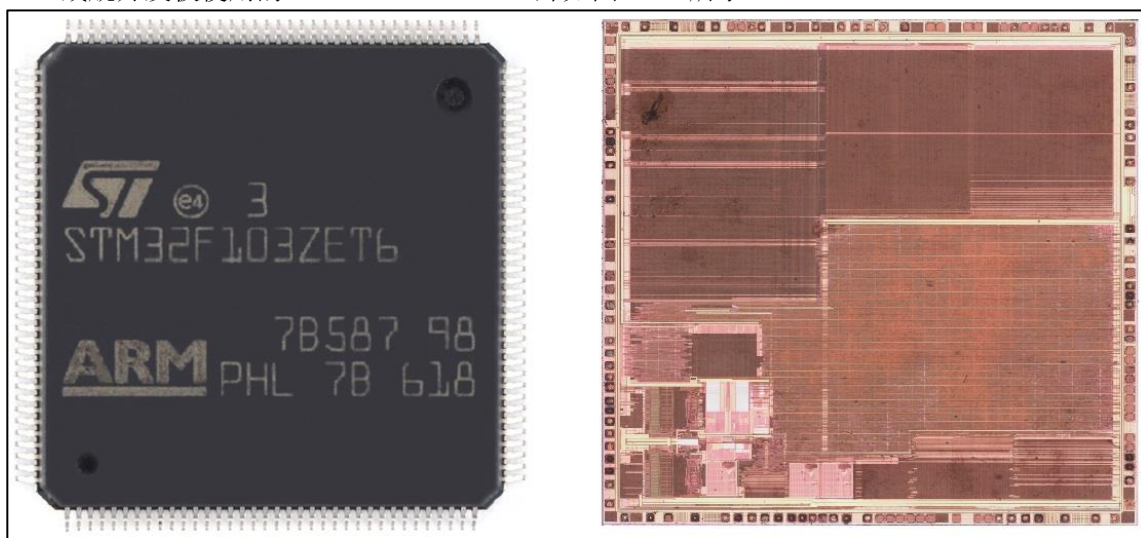


图 2.1.1 STM32F103ZET6 芯片 (LQFP144 脚)

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

STM32 的优异性体现在如下几个方面:

- 1, 超低的价格。8 位机的价格, 32 位机的性能, 是 STM32 最大的优势。
- 2, 超多的外设。STM32 拥有包括: FMC、TIMER、SPI、IIC、USB、CAN、IIS、SDIO、ADC、DAC、RTC、DMA 等众多外设及功能, 具有极高的集成度。
- 3, 丰富的型号。STM32 仅 M3 内核就拥有 F100、F101、F102、F103、F105、F107、F207、F217 等 8 个系列上百种型号, 具有 QFN、LQFP、BGA 等封装可供选择。同时 STM32 还推出了 STM32L 和 STM32W 等超低功耗和无线应用型的 M3 芯片, 另外, ST 还推出了 STM32F4/F7/H7 等更高性能的芯片。
- 4, 优异的实时性能。150 个中断, 16 级可编程优先级, 并且所有引脚都可以作中断输入。
- 5, 杰出的功耗控制。STM32 各个外设都有自己的独立时钟开关, 可以通过关闭相应外设的时钟来降低功耗。
- 6, 极低的开发成本。通过串口即可下载程序, 而且相应的仿真器也很便宜, 支持 JTAG&SWD 调试接口, 最少仅 2 个 IO 口即可实现仿真调试, 极大的降低了开发成本。

再来看一个 STM32 与 51 的性能对比如表 2.1.1 所示:

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

表 2.1.1 STM32 VS 51

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

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

2.2 STM32F103 资源简介

下面来看看 STM32F103ZET6 具体的内部资源如表 2.2.1 所示:

| STM32F103ZET6 资源 | | | | | |
|------------------|-----------|----------|----|--------|---|
| 内核 | Cortex M3 | 通用定时器 | 8 | USART | 5 |
| 主频 | 72Mhz | 高级定时器 | 2 | CAN | 1 |
| FLASH | 512KB | 12 位 ADC | 3 | SDIO | 1 |
| SRAM | 64KB | ADC 通道数 | 18 | FSMC | 1 |
| 封装 | LQFP144 | 12 位 DAC | 2 | DMA | 2 |
| IO 数量 | 112 | SPI | 3 | RTC | 1 |
| 工作电压 | 3.3V | IIC | 2 | USB 从机 | 1 |

表 2.2.1 STM32F103ZET6 内部资源表

由表可知,STM32 内部资源还是非常丰富的,本书将针对这些资源进行详细的使用介绍,并提供丰富的例程,供大家参考学习,相信经过本书的学习,您会对 STM32F103 有一个全面的了解和掌握。

关于 STM32F103 内部资源的详细介绍,请大家参考光盘→A 盘→7,硬件资料→2,芯片资料→STM32F103ZET6.pdf,该文档即 STM32F103 的数据手册,里面有 STM32F103 详细的资源说明和相关性能参数。

2.3 STM32F103 设计选型

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

2.3.1 STM32 系列

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

| 大类 | 系列 | 内核 | 特性 |
|---------|----|------------|-----------|
| 主流级 MCU | G0 | Cortex M0+ | 全新入门级 MCU |
| | G4 | Cortex M4 | 模数混合型 MCU |
| | F0 | Cortex M0 | 入门级 MCU |
| | F1 | Cortex M3 | 基础型 MCU |
| | F3 | Cortex M4 | 混合信号 MCU |
| 高性能 MCU | F2 | Cortex M4 | 高性能 MCU |
| | F4 | Cortex M4 | 高性能 MCU |
| | F7 | Cortex M7 | 高性能 MCU |

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

表 2.3.1.1 STM32 系列分类及说明

可以看到，STM32 主要分两大块，MCU 和 MPU，MCU 就是我们常见的 STM32 微控制器，不能跑 Linux，而 MPU 则是 ST 在 19 年才推出的微处理器，可以跑 Linux。本书重点介绍 MCU 产品，战舰使用的 STM32F103ZET6 属于主流 MCU 分类里面的基础型 F1 系列。

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

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

由于 STM32 系列有很好的兼容性，我们只要能够熟练掌握其中一任何一款 MCU，就可以很方便的学会并使用其他系列的 MCU。比如学好了 STM32F103，再去学 F4/F7/H7 就比较容易学会，由于 STM32F103 系列最早推向市场，资料和教程都是最多的，在市场上的使用也是最为广泛，所以对于没有接触过 STM32 的初学者来说，我们强烈建议先学习 STM32F103，再去学习其他的 STM32 系列。

2.3.2 STM32 命名

STM32 的命名规则如图 2.3.2.1 所示：



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

如图所示，STM32 的产品名字里面包含了：家族、类别、特定功能、引脚数、闪存容量、

封装、温度范围等重要信息，这些信息可以帮助我们识别和区分 STM32 不同芯片。

战舰开发板使用的 STM32F103ZET6，从命名就可以知道如表 2.3.2.1 所示信息：

| 描述 | 型号 | 说明 |
|------|-------|------------------------|
| 家族 | STM32 | ST 公司 32 位 MCU |
| 产品类别 | F | 基础型 |
| 特定功能 | 103 | STM32 基础型 |
| 引脚数 | Z | 144 脚 |
| 闪存容量 | E | 512KB FLASH |
| 封装 | T | QFP 封装 |
| 温度范围 | 6 | -40 ~ +85℃ 工作温度范围（工业级） |

表 2.3.2.1 STM32F103ZET6 型号说明

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

2.3.3 STM32 选型

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

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

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

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

2.3.4 STM32 设计

这里我们简单给大家介绍一下 STM32 的原理图设计，上一节我们通过选型原则可以确定项目所需的 STM32 具体型号，但是在选择型号以后，需要先设计原理图，然后再画 PCB、打样、焊接、调试等步骤。这里我们重点介绍如何设计 STM32F103 的原理图。

任何 MCU 部分的原理图设计，其实都遵循：**最小系统 + IO 分配** 的设计原则。在开始设计原理图之前，我们要通读一遍对 STM32F103 原理图设计非常有用的手册：STM32F103 的数据手册，可以说，不看这个数据手册，我们就无法设计 STM32F103 原理图。

1. 数据手册

在设计 STM32F103 原理图的时候，我们需要用到一个非常重要的文档：STM32F103 数据手册，里面对 STM32F103 的 IO 定义和说明有非常详细的描述，是我们设计原理图的基础。战舰开发板所使用的 STM32F103ZET6 芯片数据手册，存放在：A 盘→7，硬件资料→2，芯片资料→STM32F103ZET6.pdf / STM32F103ZET6(中文版).pdf，接下来我们简单介绍一下如何使用该文档。

STM32F103ZET6.pdf 是最新的英文版（V13）STM32 数据手册

STM32F103ZET6(中文版).pdf 是中文版（V5）STM32 数据手册

大家可以根据自己的喜欢来选择合适的版本进行阅读，内容上基本大同小异，从准确性全

面性的角度来说，看 V13 英文版是最好的，从简单，易懂来说，看 V5 中文版也是可以的。

STM32F103ZET6.pdf 数据手册是针对大容量系列（FLASH 容量在 256KB~512KB 之间），主要包括 8 个章节，如表 2.3.4.1 所示：

| 章节 | 概要说明 |
|-------|---|
| 介绍 | 简单说明数据手册作用：介绍大容量增强型 F103xC/D/E 产品的订购信息和机械特性 |
| 规格说明 | 简单介绍 STM32F103 内部所有资源及外设特点 |
| 引脚定义 | 介绍不同封装的引脚分布、引脚定义等，含引脚特性、复用功能、脚位等 |
| 存储器映像 | 介绍 STM32F103 整个 4GB 存储空间和外设的地址映射关系 |
| 电气特性 | 介绍 STM32F103 的详细电气特性，包括工作电压、电流、温度、各外设资源的电气性能等 |
| 封装特性 | 介绍了 STM32F103 不同封装的封装机械数据（脚距、长短等）、热特性等 |
| 订货代码 | 和 2.3.2 节内容类似，介绍 STM32 具体型号所代表的意义，方便选型订货 |
| 版本历史 | 介绍数据手册不同版本之间的差异和修订内容 |

表 2.3.4.1 STM32F103 数据手册各章节内容概要

整个 STM32F103 数据手册，对我们开发学习 STM32 来说都比较重要，因此建议大家简单的通读一遍这个文档，以加深印象。对于原理图设计，最重要的莫过于引脚定义这一章节了，只有知道了 STM32 的引脚定义，才能开始设计原理图。

STM32F103ZET6 引脚分布如图 2.3.4.1 所示：

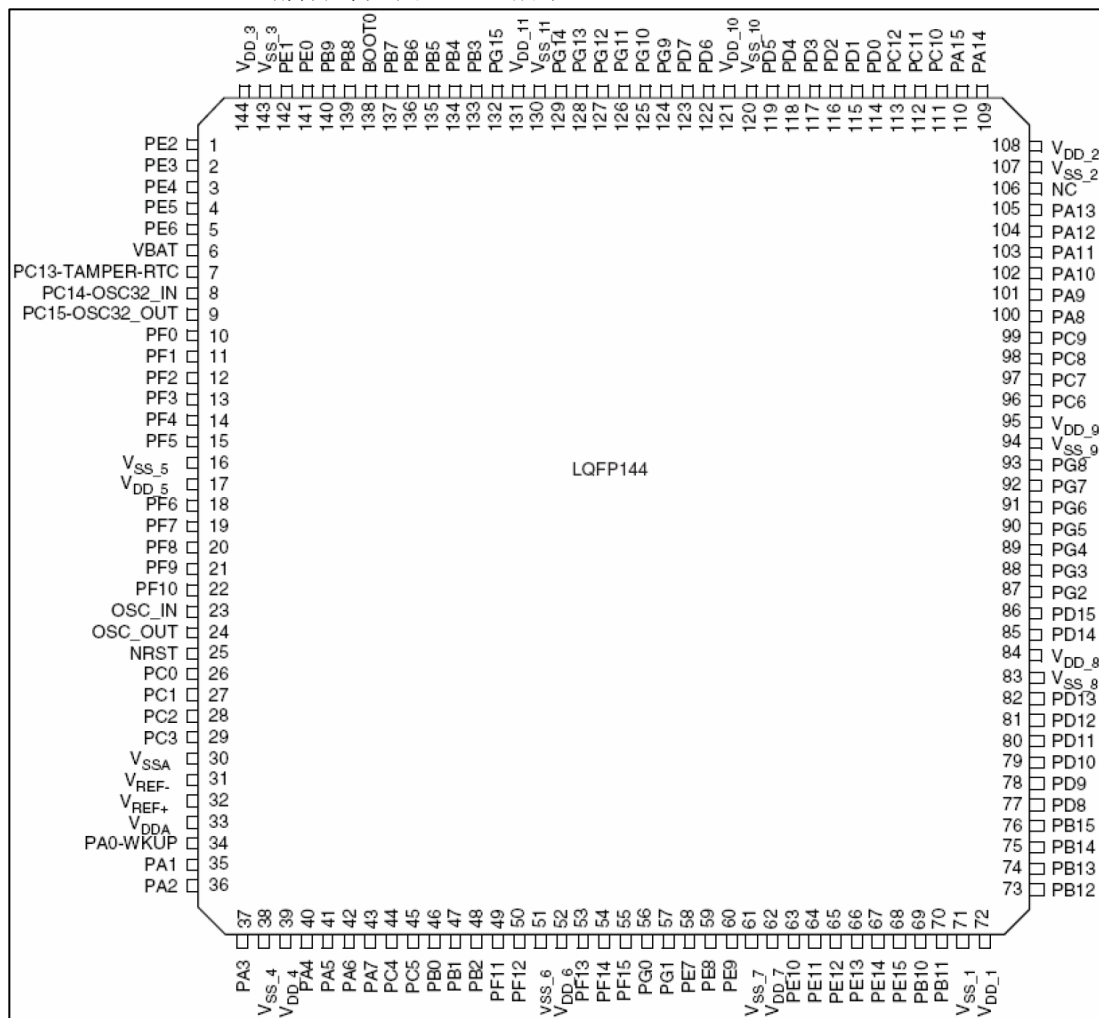


图 2.3.4.1 STM32F103ZET6 芯片引脚分布（摘自 STM32F103 数据手册）

STM32F103ZET6 引脚定义如表 2.3.4.2 所示：

| 脚位 ① | | | | | | ② | ③ | ④ | ⑤ | ⑥ 可选的复用功能 | |
|--------|--------|---------|--------|---------|---------|--------------------|-----------|----------|------------------------|--------------------------------|------------|
| BGA144 | BGA100 | WLCSP64 | LQFP64 | LQFP100 | LQFP144 | 管脚名称 | 类型 (1) | I/O电平(2) | 主功能(3) (复位后) | 默认复用功能 | 重定义功能 |
| E10 | D8 | D8 | 5 | 81 | 114 | PD0 | I/O | FT | OSC_IN ⁽⁸⁾ | FSMC_D2 ⁽⁹⁾ | CAN_RX |
| D10 | E8 | D7 | 6 | 82 | 115 | PD1 | I/O | FT | OSC_OUT ⁽⁸⁾ | FSMC_D3 ⁽⁹⁾ | CAN_TX |
| E9 | B7 | A3 | 54 | 83 | 116 | PD2 | I/O | FT | PD2 | TIM3_ETR USART5_RX/SDIO_CMD | |
| D9 | C7 | - | - | 84 | 117 | PD3 | I/O | FT | PD3 | FSMC_CLK | USART2_CTS |
| C9 | D7 | - | - | 85 | 118 | PD4 | I/O | FT | PD4 | FSMC_NOE | USART2_RTS |
| B9 | B6 | - | - | 86 | 119 | PD5 | I/O | FT | PD5 | FSMC_NWE | USART2_TX |
| E7 | - | - | - | - | 120 | V _{SS_10} | S | | V _{SS_10} | | |
| F7 | - | - | - | - | 121 | V _{DD_10} | S | | V _{DD_10} | | |

表 2.3.4.2 STM32F103ZET6 芯片引脚定义（部分）（摘自 STM32F103 数据手册）
引脚定义表的具体说明如表 2.3.4.3 所示：

| 序号 | 名称 | 说明 |
|----|--------------|--|
| ① | 脚位 | 对应芯片的引脚，LQFP 使用纯数字表示，BGA 使用字母+数字表示 这里列出了 6 种封装的脚位描述，根据实际型号选择合适的封装查阅 |
| ② | 管脚名称 | 即对应引脚的名字，PD0~5 表示 GPIO 引脚，VSS_10/VDD_10 表示第 10 组电源引脚，其他类似 |
| ③ | 类型 | I/O ：表示输入/输出引脚 I ：表示输入引脚 S ：表示电源引脚 |
| ④ | IO 电平 | FT ：表示 5V 兼容的引脚（可以接 5V/3.3V） 空 ：表示 5V 不兼容引脚（仅可以接 3.3V） |
| ⑤ | 主功能 (复位后) | 复位后，该引脚的默认功能 |
| ⑥ | 可选的 复用功能 | 默认复用功能：是指开启复用功能后，该引脚默认的复用功能 重定义功能：是指可以通过重映射的复用功能，需设置重映射寄存器 |

表 2.3.4.3 引脚定义说明

了解了引脚分布和引脚定义以后，我们就可以开始设计 STM32F103 的原理图了。

2. 最小系统

最小系统就是保证 MCU 正常运行的最低要求，一般是指 MCU 的供电、复位、晶振、BOOT 等部分。STM32F103 的最小系统需求如表 2.3.4.4 所示：

| 类型 | 引脚名称 | 说明 |
|----|---------------------|---|
| 电源 | VDD/VSS | 电源正（VDD）/负（VSS）引脚，给 STM32 供电 |
| | VDDA/VSSA | 模拟部分电源正/负引脚，给 STM32 内部模拟部分供电 |
| | VREF+/VREF- | 参考电压正/负引脚，给 STM32 内部 ADC/DAC 提供参考电压 100 脚及以上的 STM32F103 型号才有这两个脚 |
| | VBAT | RTC&后备区域供电引脚，给 RTC 和后备区域供电。一般 VBAT 接电池，用于断电维持 RTC 工作，如不需要，直接将 VBAT 接 VDD 即可 |
| 复位 | NRST | 复位引脚，用于复位 STM32，低电平复位 |
| 启动 | BOOT0/BOOT1 | 启动选择引脚，一般这两个脚各接一个下拉电阻即可 其他启动配置说明详见后续分析 |
| 晶振 | OSC_IN / OSC_OUT | 外部 HSE 晶振引脚，用于给 STM32 提供高精度系统时钟 如果使用内部 HSI 能满足使用需求，这两个脚可以不接晶振 |

| | | |
|----|----------------------|--|
| | OSC32_IN / OSC32_OUT | 外部 LSE 晶振引脚，用于给 STM32 内部 RTC 提供时钟 如果使用内部 LSI 能满足使用需求，这两个脚可以不接晶振 |
| 调试 | SWCLK/SWDIO | SWD 调试引脚，用于调试 STM32 程序，同时 STM32 还支持 JTAG 调试，不过我们不推荐使用！因为 SWD 省 IO！ |

表 2.3.4.4 STM32F103 最小系统需求

完成以上引脚的设计以后，STM32F103 的最小系统就完成了，关于这些引脚的实际原理图，大家可以参考我们战舰开发板的原理图。接下来就可以开始进行 IO 分配了。

3. IO 分配

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

| 分配 | 外设 | 说明 |
|----------|---------------|---|
| 特定 外设 | IIC | IIC 一般用到 2 根线：IIC_SCL 和 IIC_SDA（ST 叫 I2C） 数据手册有 I2C_SCL、I2C_SDA 复用功能的 GPIO 都可选用 |
| | SPI | SPI 用到 4 根线：SPI_CS/MOSI/MISO/SCK 一般 SPI_CS 我们使用通用 GPIO 即可，方便挂多个 SPI 器件 数据手册有 SPI_MOSI/MISO/SCK 复用功能的 GPIO 都可选用 |
| | TIM | 根据需要可选：TIM_CH1/2/3/4/ETR/1N/2N/3N/BKIN 等 数据手册有 TIM_CH1/2/3/4/ETR/1N/2N/3N/BKIN 复用功能的 GPIO 都可选用 |
| | USART UART | USART 有 USART_TX/RX/CTS/RTS/CK 信号 UART 仅有 UART_TX/RX 两个信号 一般用到 2 根线：U(S)ART_TX 和 U(S)ART_RX 数据手册有 U(S)ART_TX/RX 复用功能的 GPIO 都可选用 |
| | USB | USB 用到 2 根线：USB_DP 和 USB_DM 数据手册有 USB_DP、USB_DM 复用功能的 GPIO 都可选用 |
| | CAN | CAN 用到 2 根线：CAN_RX 和 CAN_TX 数据手册有 USB_DP、USB_DM 复用功能的 GPIO 都可选用 |
| | ADC | ADC 根据需要可选：ADC_IN0 ~ ADC_IN15 数据手册有 ADC_IN0 ~ ADC_IN15 复用功能的 GPIO 都可选用 |
| | DAC | DAC 根据需要可选：DAC_OUT1 / DAC_OUT2 DAC 固定为：DAC_OUT1 使用 PA4、DAC_OUT2 使用 PA5 |
| | SDIO | SDIO 一般用到 6 根线：SDIO_D0/1/2/3/SCK/CMD 数据手册有 SDIO_D0/1/2/3/SCK/CMD 复用功能的 GPIO 都可选用 |
| | FSMC | 根据需要可选：FSMC_D0~15/A0~25/ NBL0~1/NE1~4/NCE2~3/ NOE/NWE/NWAIT/CLK 等 数据手册有 FSMC_D0~15/A0~25/ NBL0~1/NE1~4/NCE2~3/NOE/ NWE/NWAIT/CLK 复用功能的 GPIO 都可选用 |
| 通用 | GPIO | 在完成特定外设的 IO 分配以后，就可以进行 GPIO 分配了 比如将按键、LED、蜂鸣器等仅需要高低电平读取/输出的外设连接到空闲的普通 GPIO 即可 |
| 微调 | IO | 微调主要包括两部分： 1，当 IO 不够用的时候，通用 GPIO 和特定外设可能要共用 IO 口 2，为了方便布线，可能要调整某些 IO 口的位置 这两点，得根据实际情况进行调整设置，做到：尽可能多的可以同时使用所有功能，尽可能方便布线。 |

表 2.3.4.5 IO 分配

经过以上几个步骤，我们就可以完成 STM32F103 的原理图设计了。

第三章 开发环境搭建

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

本章将分为如下几个小节：

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

3.1 常用开发工具简介

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

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

表 3.1.1 常用开发工具

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

3.2 MDK 安装

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

MDK5 的安装分为两步：1，安装 MDK5；2，安装器件支持包。

MDK 软件下载地址：<https://www.keil.com/download/product>，目前最新版本是 MDK5.36。

器件支持包下载地址：<https://www.keil.com/dd2/pack>，STM32F1 支持包最新版本是 2.3.0。

MDK5.36 和 2.3.0 的 STM32F1 器件支持包我们都已经放在光盘 A 盘了，具体路径为：A 盘 → 6，软件资料 → 1，软件 → MDK5，如图 3.2.1 所示：

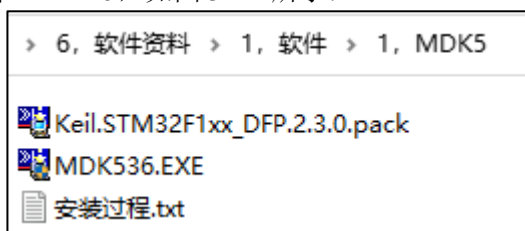


图 3.2.1 MDK5 软件及 STM32F1 器件支持包

MDK5 的安装比较简单，具体安装步骤请参考图 3.2.1 的 [安装过程.txt](#) 进行安装即可，需要提醒一下大家，在选择安装路径的时候，强烈建议大家将 Pack 的路径和 Core 的路径放在一个位置，比如我们安装在 D 盘（都安装在：D:\MDK5.36 路径下），如图 3.2.2 所示：

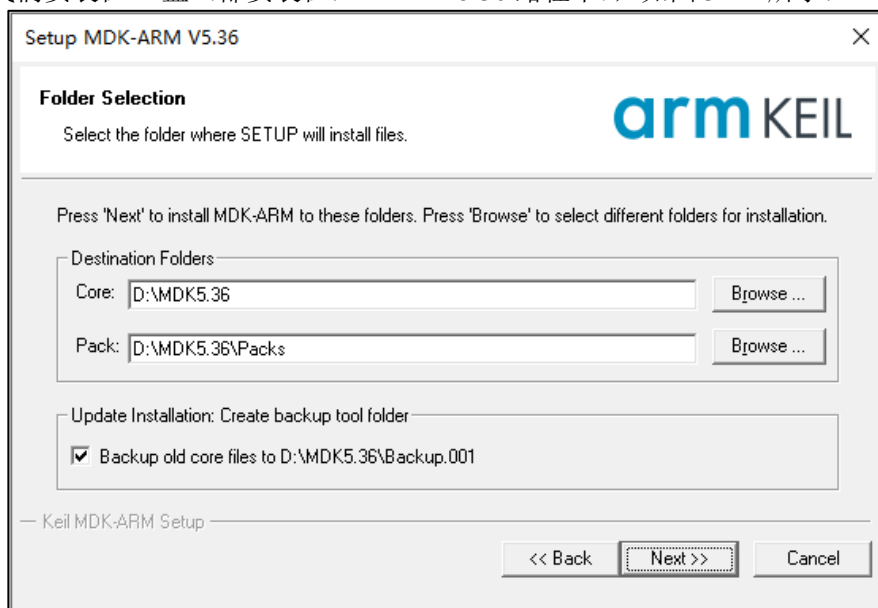


图 3.2.2 设置 Core 和 Pack 安装路径

安装完成后，在我们电脑桌面会显示 MDK5 图标，如图 3.2.3 所示：

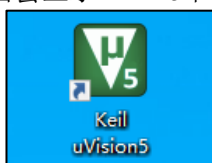


图 3.2.3 桌面显示 MDK5 图标

3.3 仿真器驱动安装

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

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

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

3.4 CH340 USB 虚拟串口驱动安装

安装 CH340 USB 虚拟串口驱动，以便我们使用电脑通过 USB 和 STM32 进行串口通信。开发板使用的 USB 虚拟串口芯片是 CH340C，其驱动我们已经放在开发板 A 盘 → 6，软件资料 → 1，软件 → CH340 驱动(USB 串口驱动)_XP_WIN7 共用 文件夹里面，如图 3.4.1 所示：



图 3.4.1 CH340 驱动

双击 SETUP.EXE 进行安装，安装完成后，如图 3.4.2 所示：

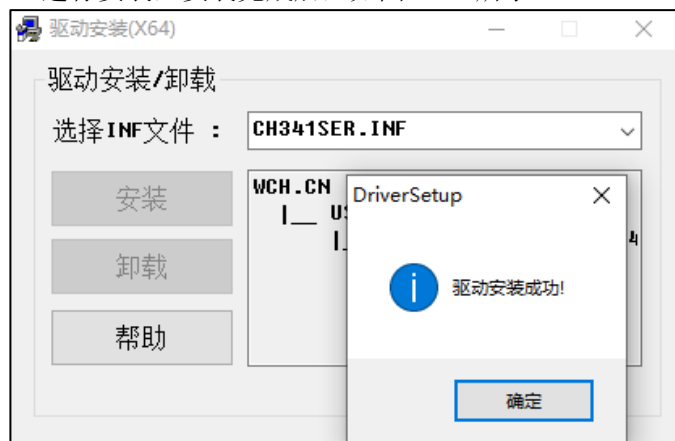


图 3.4.2 CH340 驱动安装成功

在驱动安装成功之后，将开发板的 USB_UART 接口通过 USB 连接到电脑，此时电脑就会自动给其安装驱动了。在安装完成之后，可以在电脑的设备管理器里面找到 USB 串口（如果找不到，则重启下电脑），如图 3.4.3 所示：

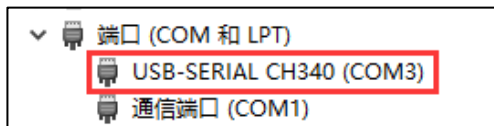


图 3.4.3 设备管理器显示 CH340 USB 虚拟串口

在图 3.4.3 中可以看到，我们的 USB 虚拟串口被识别为 COM3，这里需要注意的是：不同电脑可能不一样，你的可能是 COM4、COM5 等，但是 USB-SERIAL CH340，这个一定是一样的。如果没找到 USB 串口，则有可能是你安装有误，或者系统不兼容。

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

第四章 STM32 初体验

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

本章将分为如下几个小节：

- 4.1 使用 MDK5 编译例程
- 4.2 使用串口下载程序
- 4.3 使用 DAP 下载与调试程序
- 4.4 MDK5 使用技巧

4.1 使用 MDK5 编译例程

我们在编写完代码以后，需要对代码进行编译，编译成功以后，才能下载到开发板进行验证、调试等操作。战舰开发板标准例程源码路径：A 盘→4，程序源码，如图 4.1.1 所示：



图 4.1.1 战舰开发板例程源码

这 4 个压缩包说明如表 4.1.1：

| 压缩包 | 说明 |
|-------------------|---|
| 标准例程-寄存器版本 | 寄存器版本标准例程，无独立教程，仅供进阶参考 |
| 标准例程-HAL 库版本 | HAL 库版本标准例程，搭配 STM32F1 开发指南.pdf 学习 |
| 扩展例程 | 包含 ucOS、FreeRTOS、emWIN、LittlevGL、lwIP 例程 搭配对应的开发指南学习 |
| STM32 启动文件 | STM32F103 启动文件，用于新建工程 |
| ATKNCr(数字字母手写识别库) | 正点原子提供的数字字母手写识别库 |

表 4.1.1 源码压缩包说明

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

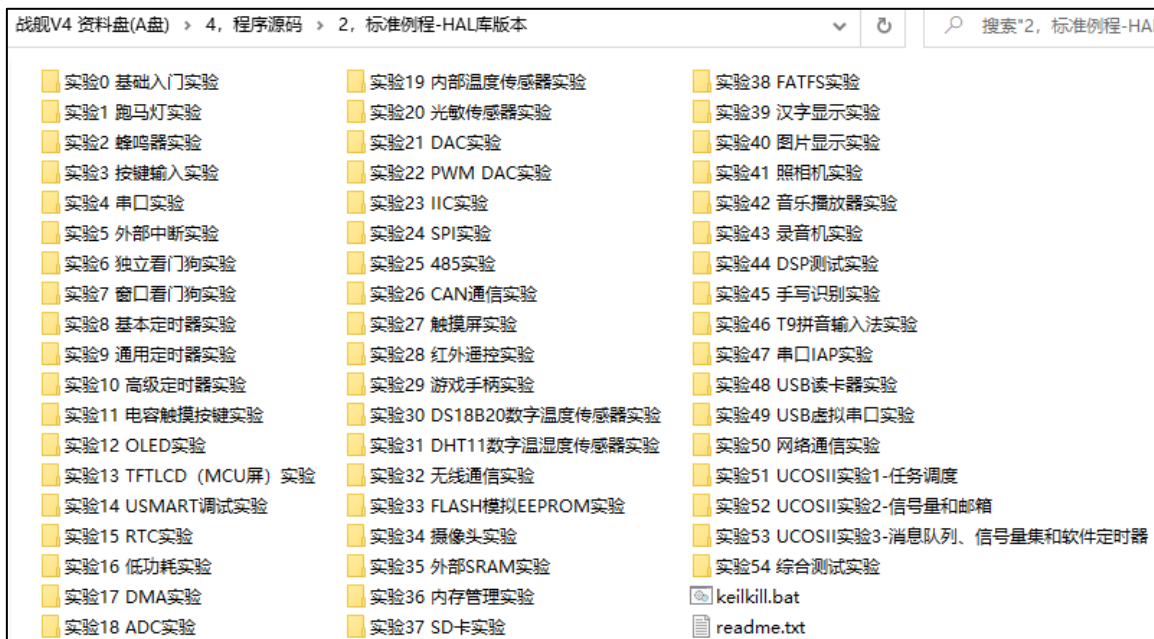


图 4.1.2 标准例程-HAL 库版本

战舰 V4 总共 54 个实验的标准例程，部分实验有多个例程，比如：通用定时器实验、高级定时器实验、ADC 实验等，所以我们的总例程数达到了 69 个。这里我们没有把实验 0 统计进来，因为实验 0 主要是新建工程等 MDK 使用基础教学例程。

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

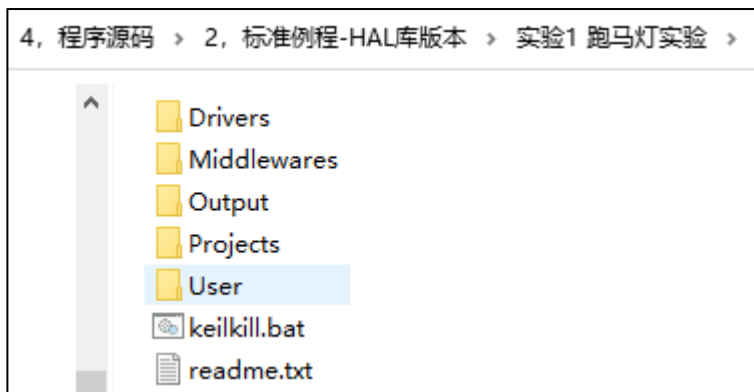


图 4.1.3 跑马灯例程工程目录结构

工程目录下有 5 个文件夹，我们将在后续“新建 HAL 库版本 MDK 工程”这一章给大家详细介绍其作用（其实看文件夹名字的字面意思基本就知道是做啥用的了），这里我们不多说。例程 MDK 工程文件的路径为：Projects→MDK-ARM→atk_f103.uvprojx，如图 4.1.4 所示：

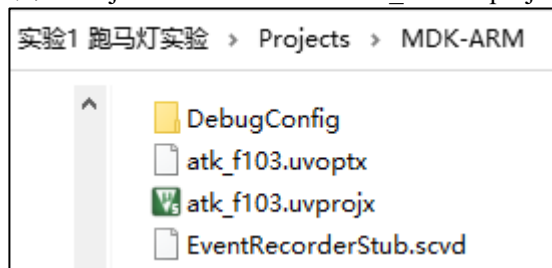


图 4.1.4 跑马灯实验例程 MDK 工程

注意：一定要先安装 MDK5（详见第三章），否则无法打开该工程文件！

双击 atk_f103.uvprojx 打开该工程文件，进入 MDK IDE 界面，如图 4.1.5 所示：

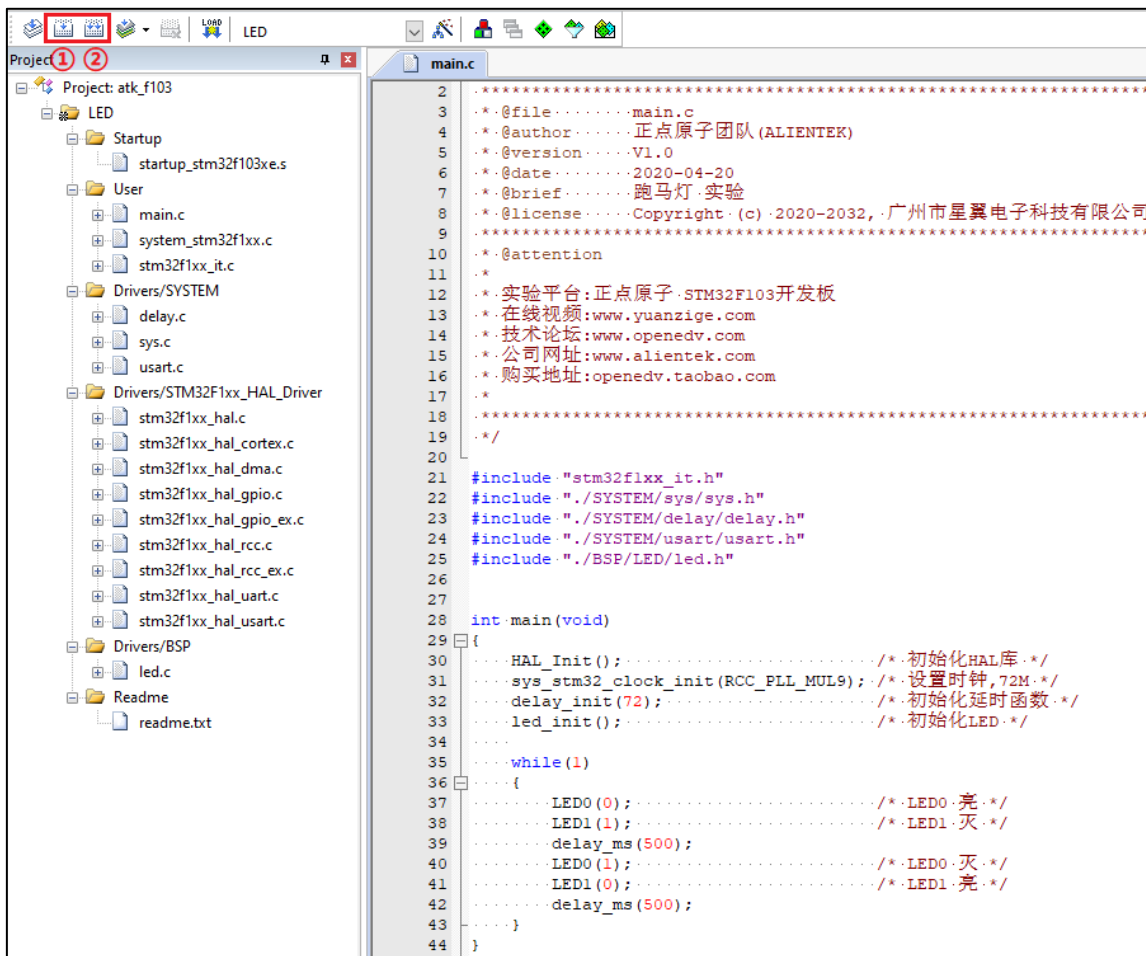


图 4.1.5 MDK 打开跑马灯实验例程

- ① 是编译按钮，表示编译当前工程项目文件，如果之前已经编译过了，则只会编译有改动的文件。所以一般第一次会比较耗时间，后续因为只编译改动文件，从而大大缩短了编译时间。该按钮可以通过 F7 快捷键进行操作。
- ② 是重新编译当前工程所有文件按钮，工程代码较多时全部重新编译会耗费比较多的时间，建议少用。

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

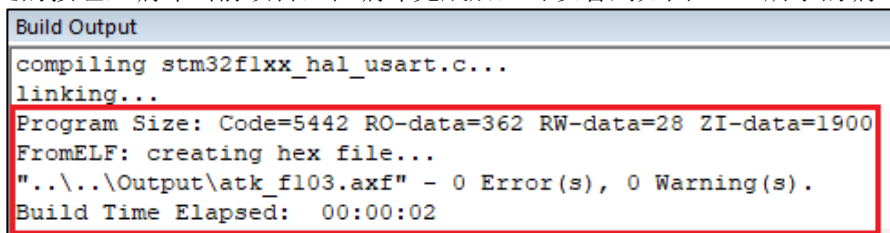


图 4.1.6 编译提示信息

图中：

Code：表示代码大小，占用 5442 字节。

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

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

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

因此图 4.1.6 的提示信息表示：代码总大小（Program Size）为：FLASH 占用 5832 字节（Code + RO + RW），SRAM 占用 1928 字节（RW + ZI）；成功创建了 Hex 文件（可执行文件，放在 Output 目录下）；编译 0 错误，0 警告；编译耗时 2 秒钟。

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

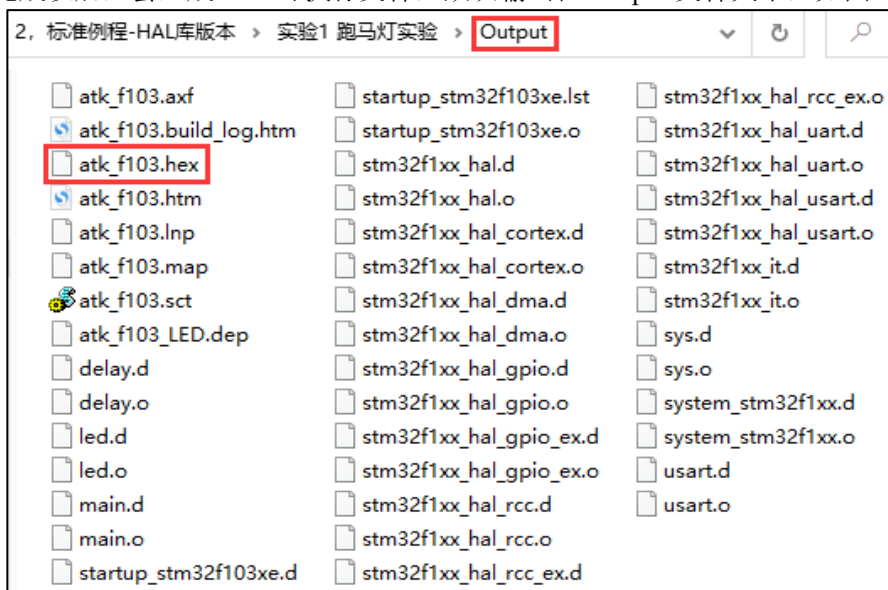


图 4.1.7 Hex 可执行文件

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

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

4.2 使用串口下载程序

STM32 常见的下载方式有三种：

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

表 4.2.1 STM32 常见下载方式对比

经过以上对比，因此我们推荐使用 **SWD 下载**，强烈推荐大家购买一个仿真器（如 ST LINK、CMSIS DAP 等），可以极大的方便学习和开发。不推荐使用 **串口下载**（速度慢、无法仿真和调试）和 **JTAG 下载**（占用 IO 多）。

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

| BOOT0 | BOOT1 | 启动模式 | 说明 |
|-------|-------|--------------------------------|---|
| 0 | X | 用户闪存存储器启动 地址：0X0800 0000 | 用户闪存存储器启动，也就是 FLASH 启动 正常运行我们自己的程序，就应该用此设置 |
| 1 | 0 | 系统存储器启动 地址：0X1FFF F000 | 系统存储器启动，用于串口下载程序 |
| 1 | 1 | SRAM 启动（不常用） 地址：0X2000 0000 | SRAM 启动，一般没用 |

表 4.2.2 STM32 启动模式

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

接下来，我们将向大家介绍，如何利用串口给 STM32F103（以下简称 STM32）下载代码。

STM32 通过串口 1 实现程序下载，战舰开发板通过自带的 USB 转串口来实现串口下载。看起来像是 USB 下载（只需一根 USB 线，并不需要串口线）的，实际上，是通过 USB 转成串口，然后再下载的。

下面，我们就一步步教大家如何在实验平台上利用 USB 串口来下载代码。

首先确保开发板接线正确，并成功上电，默认的出厂设置如图 4.2.1 所示：

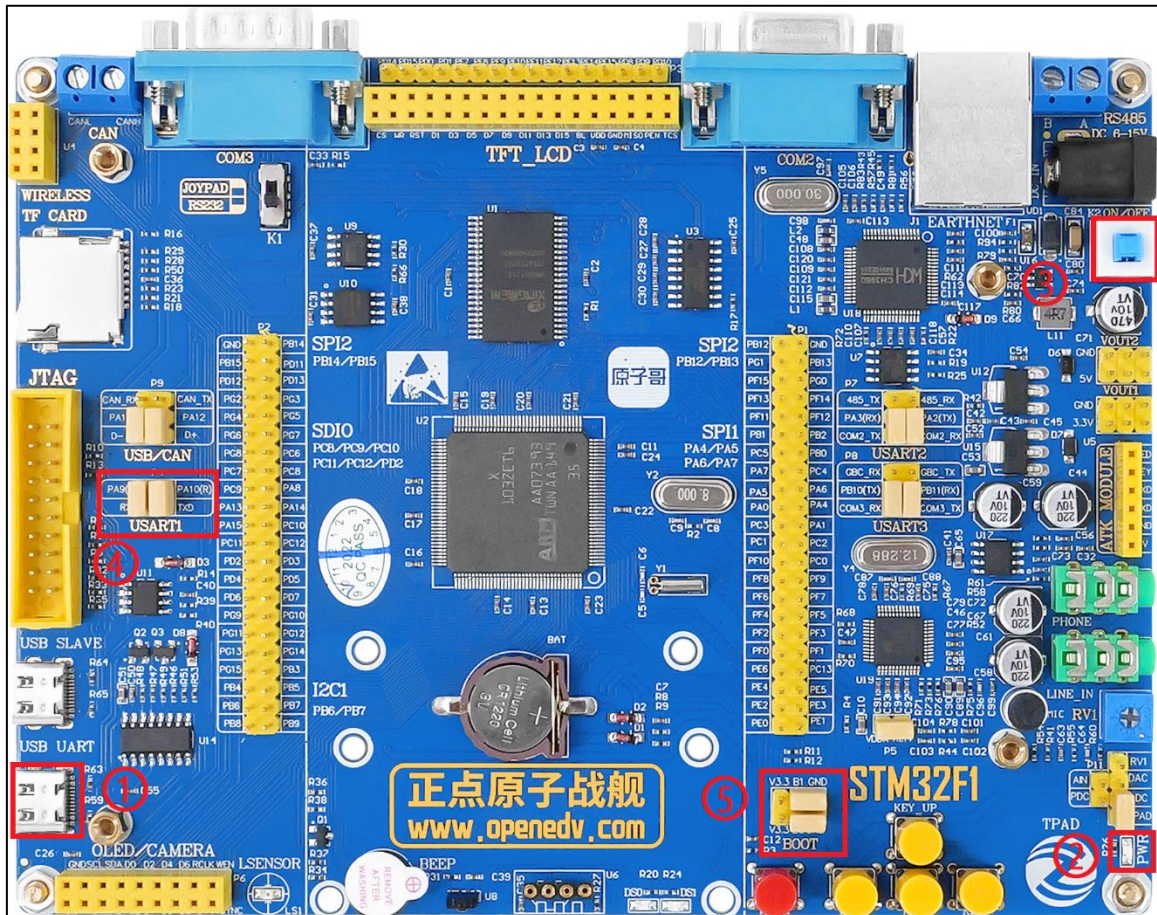


图 4.2.1 开发板设置

- ①处的 USB_UART 通过 USB 线连接电脑，实现 USB 转串口，同时支持给开发板供电。
- 确保电源灯②亮起（蓝色），如果不亮检查供电和电源开关③是否按下？
- 确保 P4 端子的 RXD 和 PA9（STM32 的 TXD），TXD 和 PA10(STM32 的 RXD)通过跳线帽连接起来（标号④），这样我们就把 CH340C 和 STM32 的串口 1 连接上了。
- ⑤处的 BOOT 设置为 BOOT0（简称 B0）和 BOOT1（简称 B1）都接 GND（一键下载电路自动控制，待会介绍）。

由表 4.2.1 可知，使用串口下载 STM32 程序需要修改 BOOT 的设置，四个步骤如下：

1. 把 B0 接 V3.3
2. 保持 B1 接 GND
3. 按一下复位按键
4. 使用上位机软件下载代码

通过这几个步骤，我们就可以通过串口下载代码了，下载完成之后，如果没有设置从 0X08000000 开始运行，则代码不会立即运行，此时，你还需要把 B0 接回 GND，然后再按一次

复位，才会开始运行你刚刚下载的代码。所以整个过程，你得调动 2 次跳线帽，还得按 2 次复位，比较繁琐。而我们的一键下载电路，则利用串口的 DTR 和 RTS 信号，分别控制 STM32 的复位和 B0，配合正点原子团队研发的上位机软件 ATK-XISP，设置：**DTR 的低电平复位，RTS 高电平进 BootLoader**，这样，B0 和 STM32 的复位，完全可以由下载软件自动控制，从而实现一键下载。一键下载的原理图级别解释详见：战舰 V4 硬件参考手册.pdf

串口程序下载需要用到：ATK-XISP 这个上位机软件。这个软件可以实现对 STM32F1 到 STM32H7 等系列芯片的串口编程。ATK-XISP 软件在光盘路径：A 盘→6，软件资料→1，软件→STM32 串口下载软件（ATK-XISP），如图 4.2.2 所示：



图 4.2.2 ATK-XISP 串口编程软件

双击打开 ATK-XISP，进行如图 4.2.3 所示的设置：



图 4.2.3 ATK-XISP 设置

- ① 搜索串口选择 CH340 虚拟的串口（我的是 COM23，不同电脑可能不同，需要根据实际情况选择对应的串口，详见 3.4 节），然后设置波特率为 460800，保证最大速度下载。
- ② 选择 4.1 节编译生成的 hex 文件（在 Output 文件夹）
- ③ 勾选校验和编程后执行两项，可以保证下载代码的正确性，并下载完后自动运行，省去按复位的麻烦。注意：千万不要勾选：使用 RamIsp 和连续烧录模式！否则下载失败！
- ④ 选择 **DTR 的低电平复位，RTS 高电平进 BootLoader**（别选错！），以匹配一键下载电路，实现一键下载代码，省去设置 BOOT0、按复位的麻烦。

设置好之后，我们就可以通过按**开始编程（P）**这个按钮，一键下载代码到 STM32 上，下载成功后如图 4.2.4 所示：



图 4.2.4 下载完成

图 4.2.4 中，我们圈出了 ATK-XISP 对一键下载电路的控制过程，先是获取芯片信息，然后对全片擦除，还显示了文件的首地址以及文件大小等。

另外，下载成功后，会有“共写入 xxxxB，进度 100%，耗时 xxxx 毫秒”的提示，并且从 0X80000000 处开始运行了。此时表示代码下载完成，并已经成功运行了，我们查看开发板就可以看到 DS0（红灯）、DS1（绿灯）开始交替闪烁了，如图 4.2.5 所示：



图 4.2.5 开发板运行代码

至此，STM32 使用串口下载程序完成。其他实验例程，大家可以用同样的方法进行下载。

4.3 使用 DAP 下载与调试程序

上一节我们使用串口给 STM32 下载程序，但是串口下载并不能仿真调试代码，只能下载后观看运行结果，所以在调试代码 bug 的时候，最好还是用仿真器进行下载调试，本节我们将给大家介绍如何使用仿真器给 STM32 下载代码，并调试代码。

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

DAP 与开发板连接如图 4.3.1 所示：

- DAP 通过 USB 线连接电脑，且仿真器①处的蓝灯常亮。
- DAP 通过 20P 灰排线连接开发板，如下图③处所示。
- 确保开发板已经正常供电(可经由图中的⑤⑥给开发板供电，但我们建议使用⑤处的电源插座+开发板适配的 DC 直流电源，以避免部分需要用到大电流的例程运行不正常)，供电后确保⑦处的开关是按下的状态，这里蓝色电源灯亮起表示开发板已经正常供电。
- B0, B1 都接 GND，如下图④处所示。

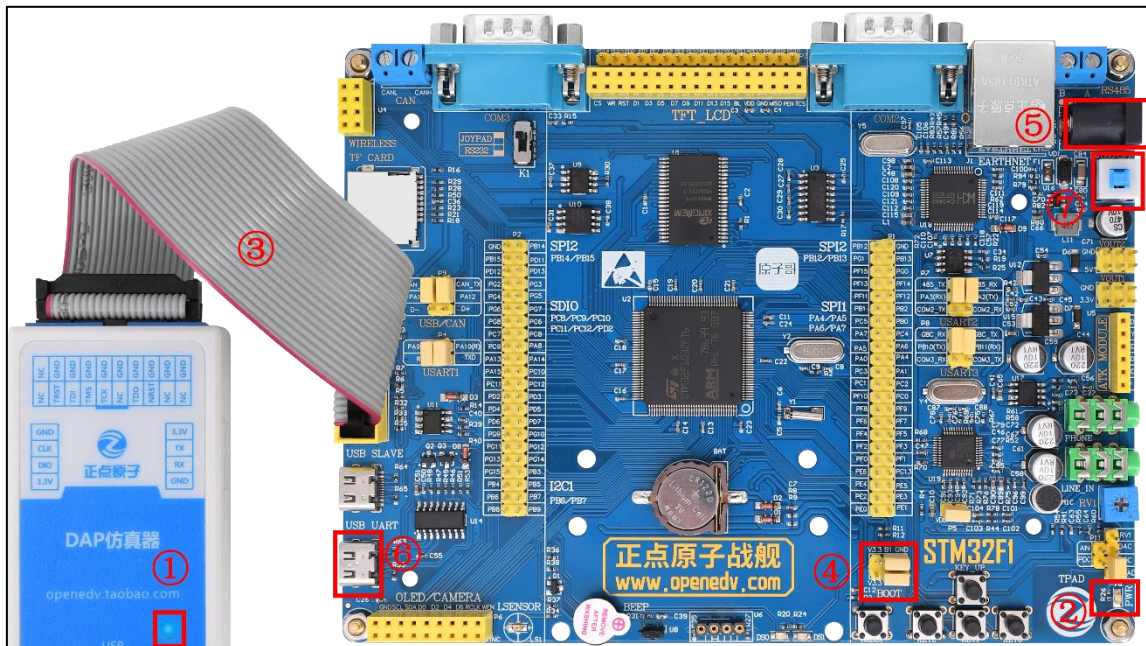


图 4.3.1 DAP 与开发板连接

4.3.1 使用 DAP 下载程序

在 4.1 节的跑马灯例程 MDK IDE 界面下，点击 按钮，打开 Options for Target 选项卡，在 Debug 栏选择仿真工具为 use: CMSIS-DAP Debugger，如图 4.3.1.1 所示：

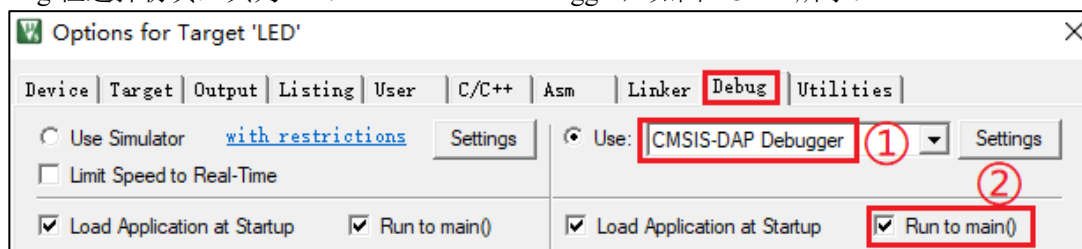


图 4.3.1.1 Debug 选项卡设置

- ① 选择使用 CMSIS-DAP Debugger 仿真器仿真调试代码。如果你使用的是其他仿真器，比如 STLINK、JLINK 等，请在这里选择对应的仿真器型号。
- ② 该选项选中后，只要点击仿真就会直接运行到 main 函数，如果没选择这个选项，则会先执行 startup_stm32f103xe.s 文件的 Reset_Handler，再跳到 main 函数。

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

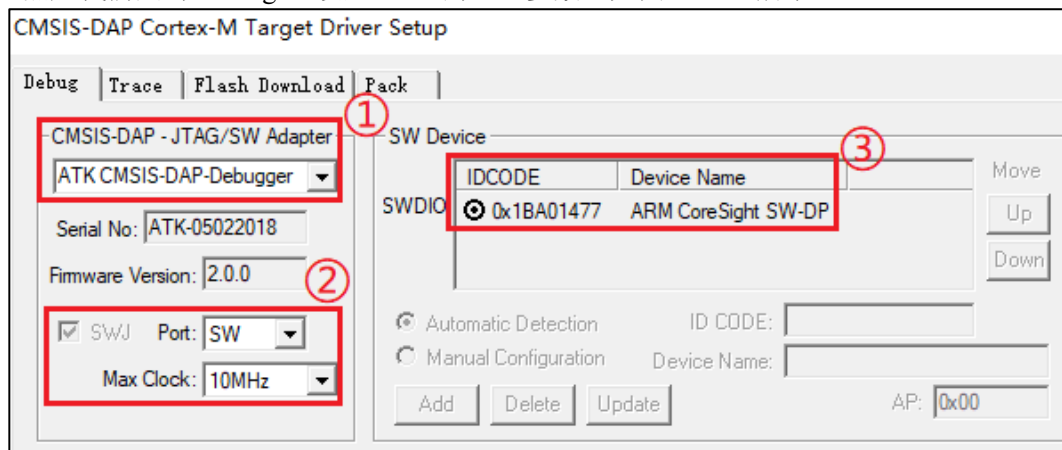


图 4.3.1.2 DAP 仿真器参数设置

- ① 表示 MDK 找到了 ATK CMSIS-DAP 仿真器，如果这里显示为空，则表示没有仿真器被找到，请检查你的电脑是否接了仿真器？并安装了对应的驱动？
- ② 设置接口方式，这里选择 SW（比 JTAG 省 IO），通信速度设置为 10Mhz（实际上大概只有 4M 的速度，MDK 会自动匹配）。
- ③ 表示 MDK 通过仿真器的 SW 接口找到了目标芯片，ID 为：0x1BA01477。如果这里显示：No target connected，则表示没找到任何器件，请检查仿真器和开发板连接是否正常？开发板是否供电了？

其他部分使用默认设置，设置完成以后单击“确定”按钮，完成此部分设置，接下来我们还需要在 Utilities 选项卡里面设置下载时的目标编程器，如图 4.3.1.3 所示：

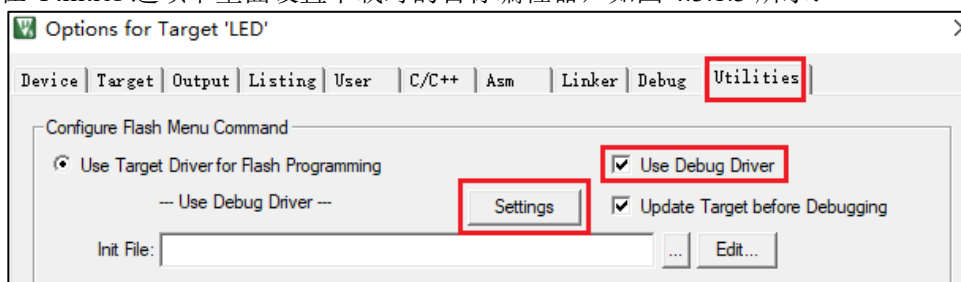


图 4.3.1.3 FLASH 编程器选择

上图中，我们直接勾选 Use Debug Driver，即和调试一样，选择 DAP 来给目标器件的 FLASH 编程，然后点击 Settings，进入 FLASH 算法设置，设置如图 4.3.1.4 所示：

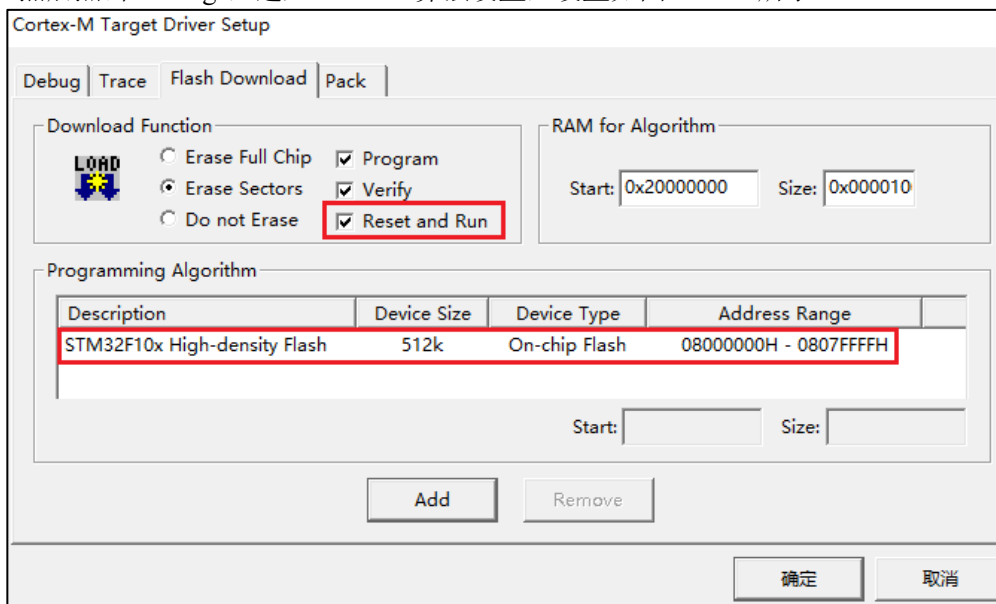



图 4.3.1.4 FLASH 算法设置

这里 MDK5 会根据我们新建工程时选择的目标器件，自动设置 flash 算法。我们使用的是 STM32F103ZET6，FLASH 容量为 512K 字节，所以 Programming Algorithm 里面默认会有 512K 型号的 STM32F10x High-density Flash 算法。另外，如果这里没有 flash 算法，大家可以点击 Add 按钮，自行添加即可。最后，选中 Reset and Run 选项，以实现在编程后自动运行，其他默认设置即可。


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

```
Build Output
Load "..\\..\\Output\\atk_f103.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 11:20:01
```

图 4.3.1.5 仿真器下载代码

下载完后，就可以看到 DS0 和 DS1 交替闪烁了，说明代码下载成功。

4.3.2 使用 DAP 仿真调试程序

在正常编译完例程以后（0 错误，0 警告），点击：（开始/停止仿真按钮），开始仿真（如果开发板的代码没被更新过，则会先更新代码（即下载代码），再仿真），如图 4.3.2.1 所示：

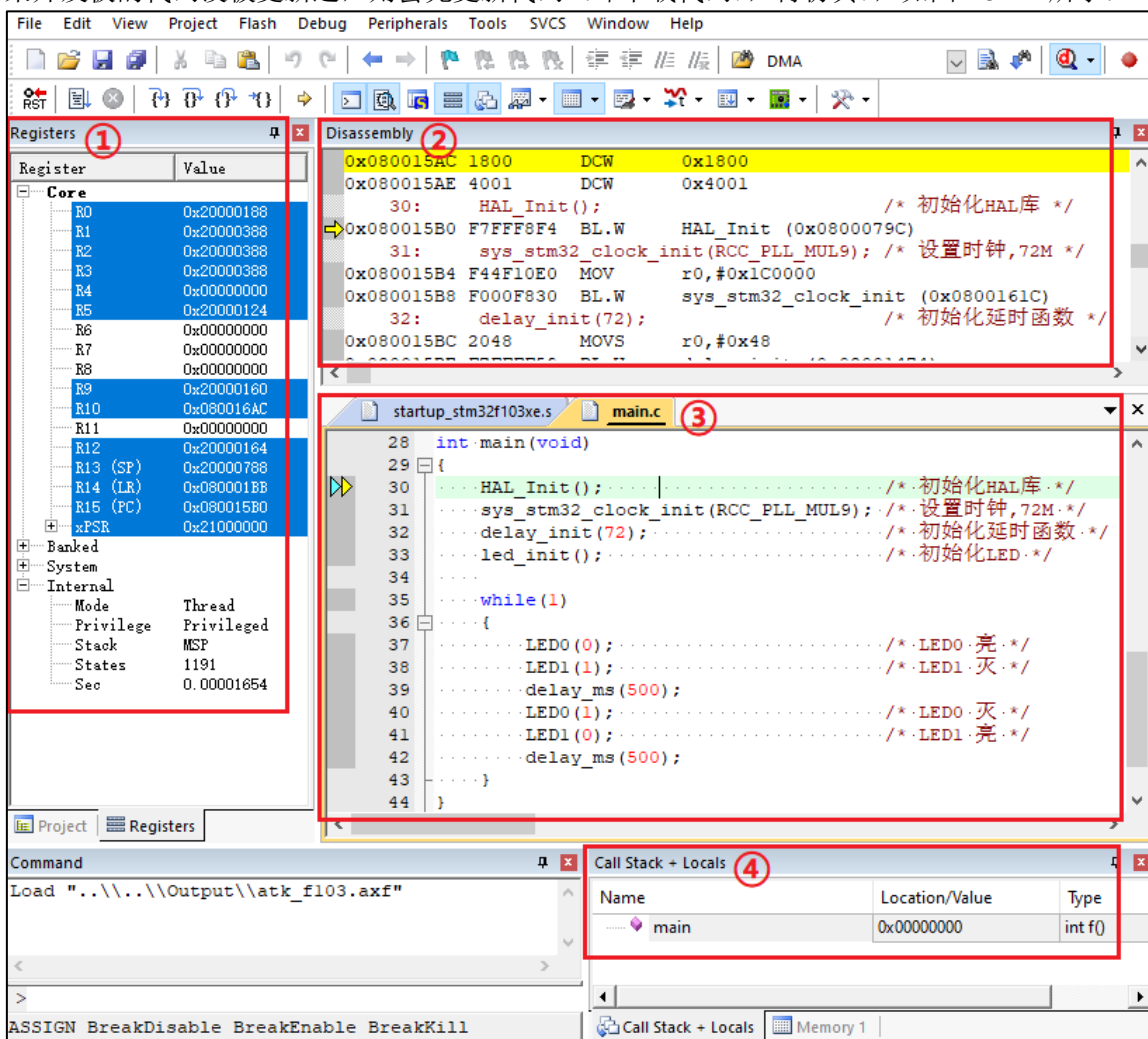


图 4.3.2.1 开始仿真

- ① Register: 寄存器窗口，显示了 Cortex M3 内核寄存器 R0~R15 的值，还显示了内部的线程模式（处理器模式、线程模式）及特权级别（用户级、特权级），并且还显示了当前程序的运行时间（Sec），该选项卡一般用于查看程序运行时间，或者比较高级的 bug 查找（涉及到分析 R0~R14 数据是否异常了）。
- ② Disassembly: 反汇编窗口，将 C 语言代码和汇编对比显示（指令存放地址，指令代码，指令，具体操作），方便从汇编级别查看程序运行状态，同样也属于比较高级别的 bug 查找。

- ③ 代码窗口，在左侧有黄绿色三角形，黄色的三角形表示将要执行的代码，绿色的三角形表示当前光标所在代码（C 代码 或 当前汇编行代码对应的 C 代码）。一般情况下，这两个三角形是同步的，只有在点击光标查看代码的时候，才可能不同步。
- ④ Call Stack + Locals：调用关系&局部变量窗口，通过该窗口可以查看函数调用关系，以及函数的局部变量，在仿真调试的时候，是非常有用的。

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

图 4.3.2.1 中，还有一个很重要的工具条：Debug 工具条，其内容和作用如图 4.3.2.2 所示：

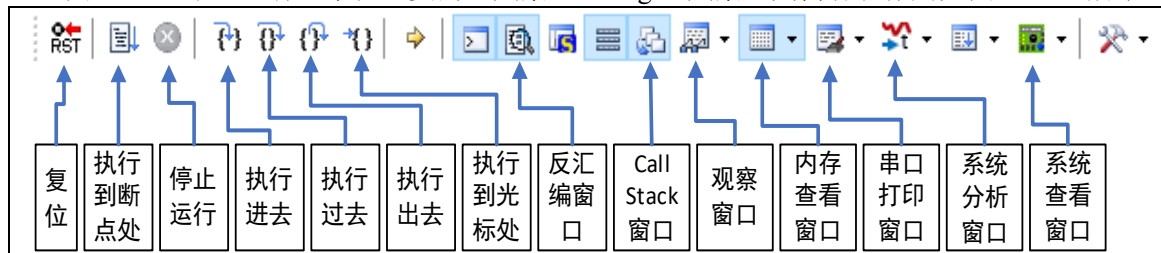


图 4.3.2.2 Debug 工具条

复位：其功能等同于硬件上按复位按钮。相当于实现了一次硬复位。按下该按钮之后，代码会重新从头开始执行。

执行到断点处：该按钮用来快速执行到断点处，有时候你并不需要观看每步是怎么执行的，而是想快速的执行到程序的某个地方看结果，这个按钮就可以实现这样的功能，前提是你查看的地方设置了断点。

停止运行：此按钮在程序一直执行的时候会变为有效，通过按该按钮，就可以使程序停下来，进入到单步调试状态。

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

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

执行出去：该按钮是在进入了函数单步调试的时候，有时候可能不必再执行该函数的剩余部分了，通过该按钮就可以一步执行完该函数的余部分，并跳出函数，回到函数被调用的位置。

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

反汇编窗口：通过该按钮，就可以查看汇编代码，这对分析程序很有用。

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

观察窗口：MDK5 提供 2 个观察窗口（下拉选择），该按钮按下，会弹出一个显示变量的窗口，输入你所想要观察的变量/表达式，即可查看其值，是很常用的一个调试窗口。

内存查看窗口：MDK5 提供 4 个内存查看窗口（下拉选择），该按钮按下，会弹出一个内存查看窗口，可以在里面输入你要查看的内存地址，然后观察这一片内存的变化情况。是很常用的一个调试窗口。

串口打印窗口：MDK5 提供 4 个串口打印窗口（下拉选择），该按钮按下，会弹出一个类似串口调试助手界面的窗口，用来显示从串口打印出来的内容。

系统分析窗口：该图标下面有 6 个选项（下拉选择），我们一般用第一个，也就是逻辑分析窗口(Logic Analyzer)，点击即可调出该窗口，通过 SETUP 按钮新建一些 IO 口，就可以观察这些 IO 口的电平变化情况，以多种形式显示出来，比较直观。

系统查看窗口：该按钮可以提供各种外设寄存器的查看窗口（通过下拉选择），选择对应外设，即可调出该外设的相关寄存器表，并显示这些寄存器的值，方便查看设置的是否正确。

Debug 工具条上的其他几个按钮用的比较少，我们这里就不介绍了。以上介绍的是比较常用的，当然也不是每次都用得着这么多，具体看你程序调试的时候有没有必要观看这些东西，

来决定要不要看。

我们在图 4.3.2.1 的基础上：关闭反汇编窗口（Disassembly）、添加观察窗口 1（Watch1）。然后调节一下窗口位置，然后将全局变量：g_fac_us（在 delay.c 里面定义）加入 Watch1 窗口（方法：双击 Enter expression 添加/直接拖动变量 t 到 Watch1 窗口即可），如图 4.3.2.3 所示：

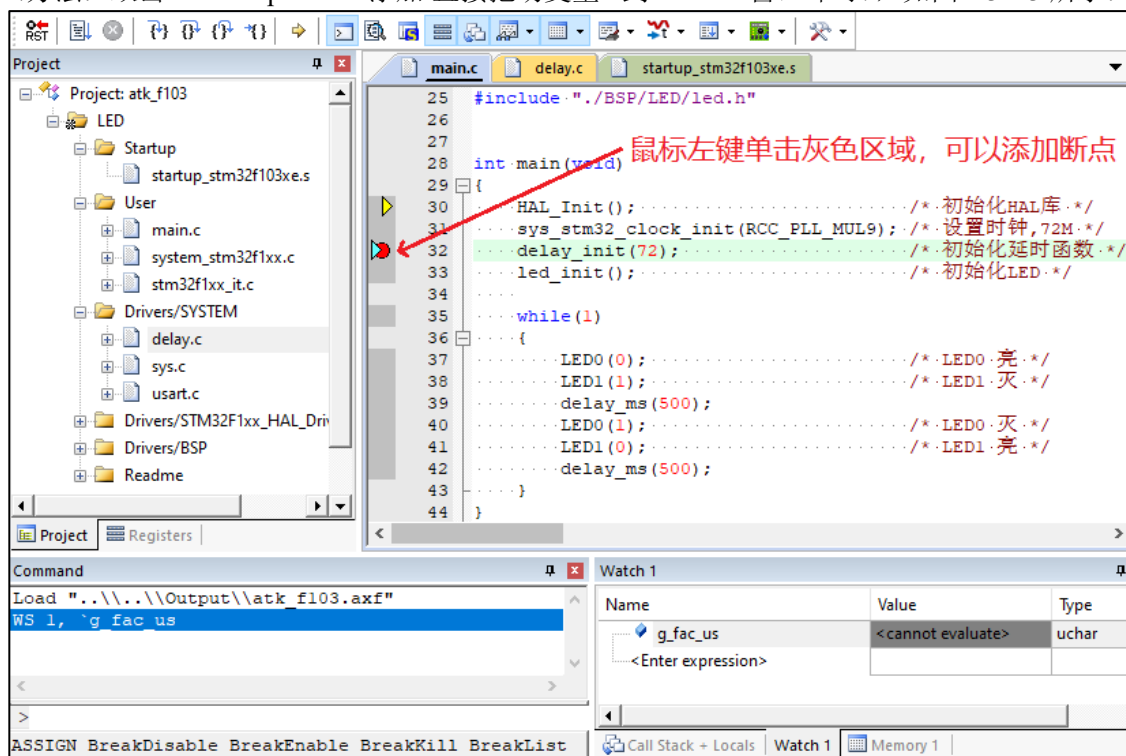


图 4.3.2.3 开始仿真

此时可以看到 Watch1 窗口的 g_fac_us 的值提示：无法计算，我们把鼠标光标放在第 32 行左侧的灰色区域，然后按下鼠标左键，即可放置一个断点（红色的实心点，也可以通过鼠标右键弹出菜单来加入），这样就在 delay_init 函数处放置一个断点，然后点击：[F5]，执行到该断点处，然后再点击：[F7]，执行进入 delay_init 函数，如图 4.3.2.4 所示：

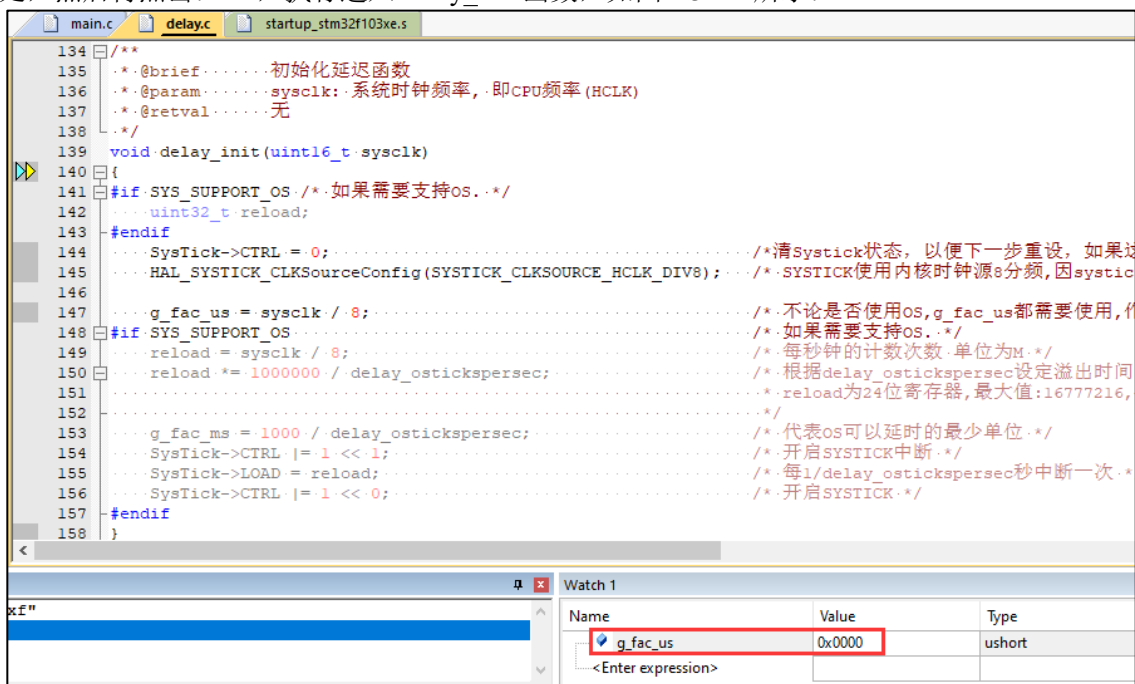



图 4.3.2.4 执行进入 delay_init 函数

此时，可以看到 `g_fac_us` 的值已经显示出来了，默认是 0（全局变量如果没有赋初值，一般默认都是 0），然后继续单击：，单步运行代码到 `g_fac_us = sysclk / 8;` 这一行，再把鼠标光标放在 `sysclk` 上停留一会，就可以看到 MDK 自动提示 `sysclk`（`delay_init` 的输入参数）的值为：0X0048，即 72，如图 4.3.2.5 所示：

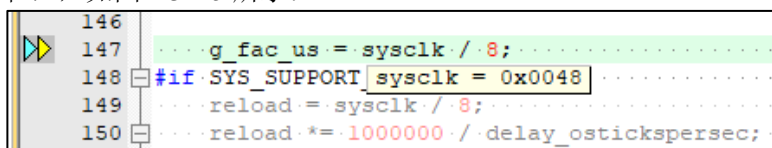


图 4.3.2.5 单步运行到 `g_fac_us` 赋值

然后再单步执行过去这一行，就可以在 Watch1 窗口中看到 `g_fac_us` 的值变成 9 了，如图 4.3.2.6 所示：


| Watch 1 | | |
|---|--------|--------|
| Name | Value | Type |
|  <code>g_fac_us</code> | 0x0009 | ushort |
| <Enter expression> | | |

图 4.3.2.6 Watch1 窗口观看 `g_fac_us` 的值

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

然后，我们再回到 `main.c`，在第一个 `LED0(0)` 处放置一个断点，运行到断点处，如图 4.3.2.7 所示：

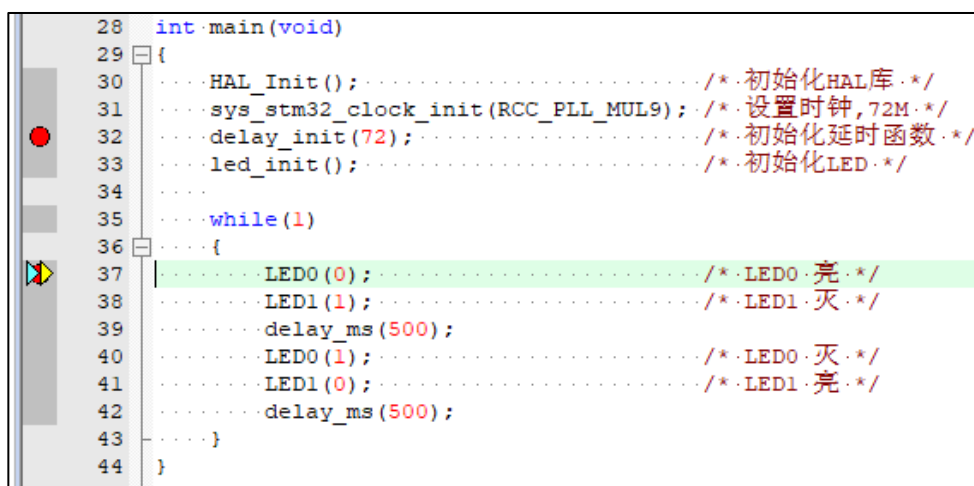




图 4.3.2.7 运行到 `LED0(0)` 代码处

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

最后，我们在 `delay.c` 文件的 `delay_us` 函数，第二行处设置一个断点，然后运行到断点处，如图 4.3.2.8 所示：

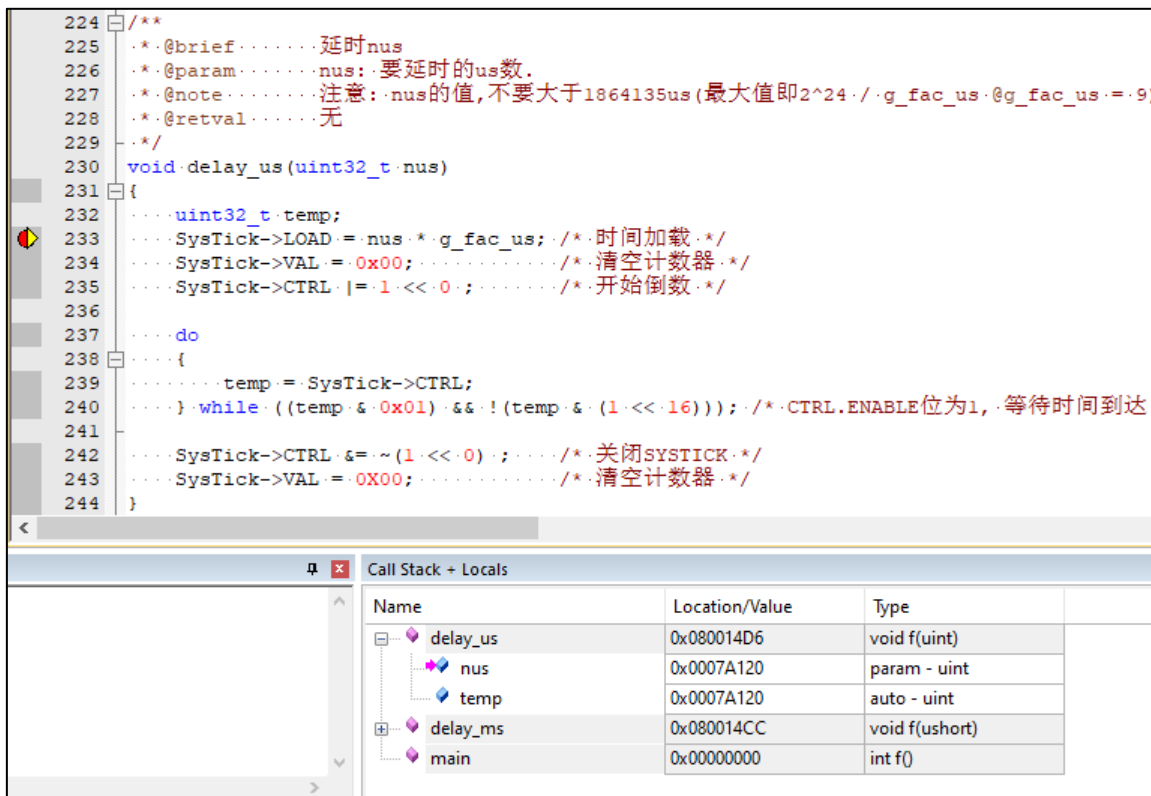


图 4.3.2.8 查看函数调用关系及局部变量

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

关于 DAP 的仿真调试，我们暂时就讲这么多。

4.3.3 仿真调试注意事项

1. 由于 MDK5.23 以后对中文支持不是很好，具体现象是：在仿真的时候，当有断点未清除时点击结束仿真，会出现如图 4.3.3.1 所示的报错：

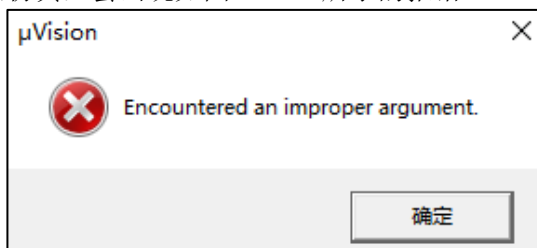


图 4.3.3.1 仿真结束时报错！

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

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

又得重设的麻烦。

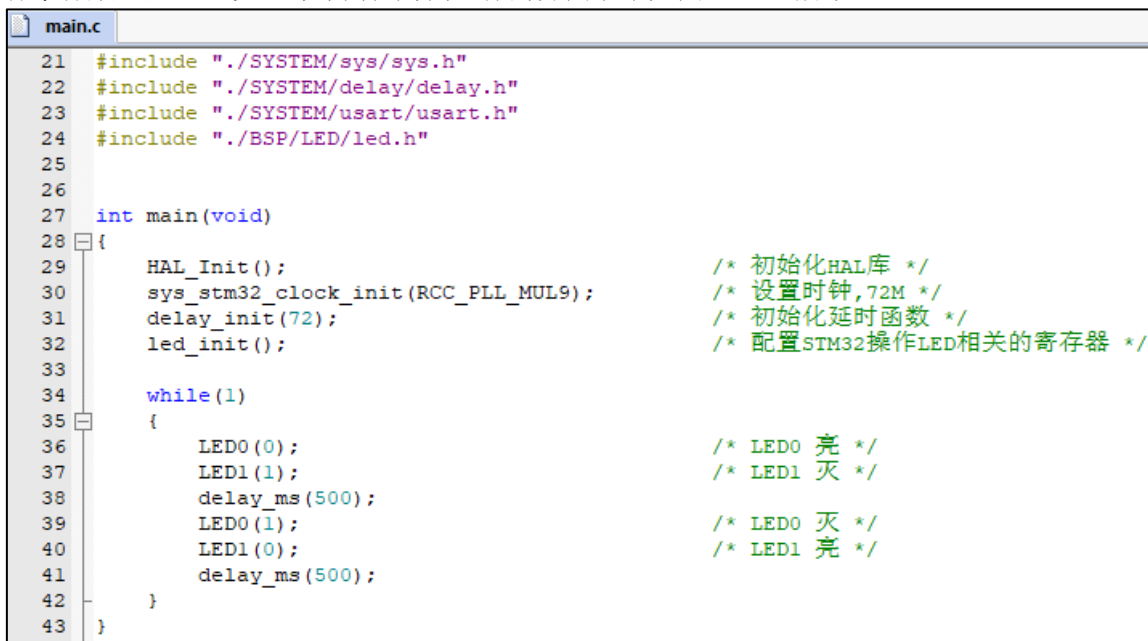
- 2, 关于 STM32 软件仿真, 老版本的教程, 我们给大家介绍过如何使用 MDK 进行 STM32 软件仿真, 由于其限制较多 (只支持部分 F1 型号), 而且仿真器越来越便宜, 硬件仿真更符合实际调试需求, 调试效果更好。所以后续我们只介绍硬件仿真, 不再推荐大家使用软件仿真了!
- 3, 仿真调试找 bug 是一个软件工程师必备的基本技能。MDK 提供了很多工具和窗口来辅助我们找问题, 只要多使用, 多练习, 肯定就可以把仿真调试学好。这对我们后续的独立开发项目, 非常有帮助。因此极力推荐大家多练习使用仿真器查找代码 bug, 学会这个基本技能。
- 4, 调试代码不要浅尝辄止, 要想尽办法找问题, 具体的思路: 先根据代码运行的实际现象分析问题, 确定最可能出问题的地方, 然后在相应的位置放置断点, 查看变量, 查看寄存器, 分析运行状态和预期结果是否一致? 从而找到问题原因, 解决 bug。特别提醒: 一定不要浅尝辄止, 很多朋友只跟踪到最上一级函数, 就说死机了, 不会跟踪进去找问题! 所以一定要一层层进入各种函数, 越是底层 (甚至汇编级别), 越好找到问题原因。

4.4 MDK5 使用技巧

本节, 我们将向大家介绍 MDK5 软件的一些使用技巧, 这些技巧在代码编辑和编写方面会非常有用, 希望大家好好掌握, 最好实际操作一下, 加深印象。


4.4.1 文本美化

文本美化, 主要是设置一些关键字、注释、数字等的颜色和字体。如果你刚装 MDK, 没进行字体颜色配置, 以跑马灯例程为例, 你的界面效果如图 4.4.1.1 所示:



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

图 4.4.1.1 MDK 默认配色效果

上图是 MDK 默认的设置, 可以看到其中的关键字和注释等字体的颜色不是很漂亮, 而 MDK 提供了我们自定义字体颜色的功能。我们可以在工具条上点击  (配置对话框) 弹出如图 4.4.1.2 所示界面:

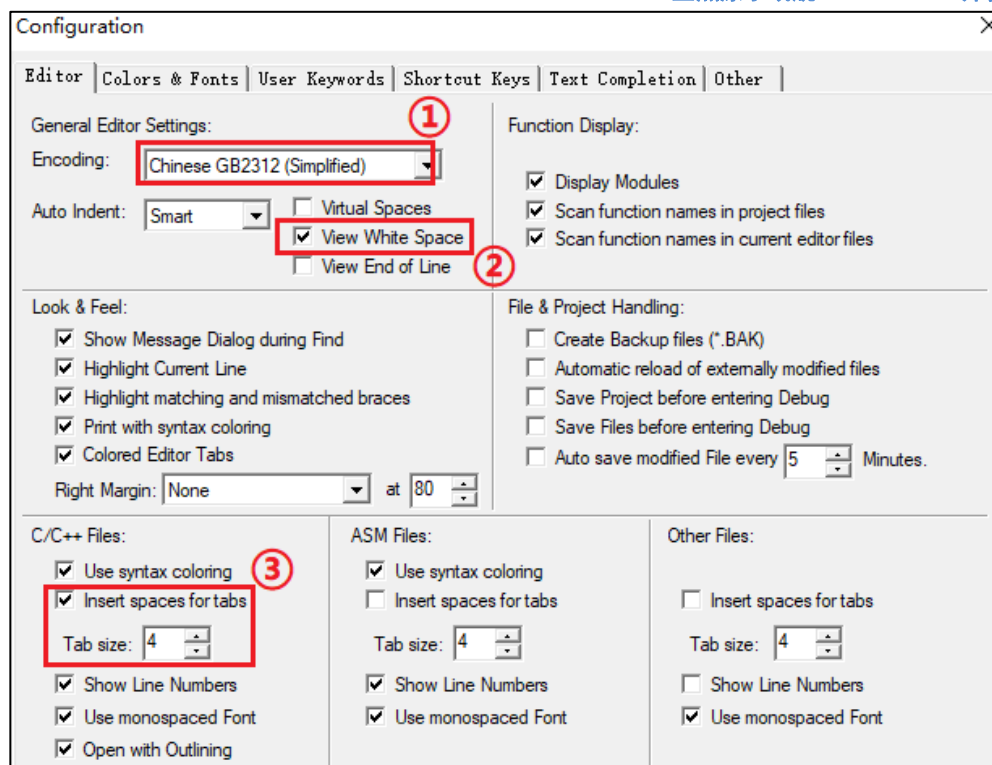


图 4.4.1.2 置对话框

- ① 设置代码编辑器字体使用：Chinese GB2312(Simplified)，以更好的支持中文。
- ② 设置编辑器的空格可见：View White Space，所有空格使用“.”替代，TAB 使用“→”替代，这样可以方便我们对代码进行对齐操作。同时，我们推荐所有的对齐都用空格来替代，这样在不同软件之间查看源代码，就不会引起由于 TAB 键大小不一样导致代码不对齐的问题，方便使用不同软件查看和编辑代码。
- ③ 设置 C/C++ 文件，TAB 键的大小为 4 个字符，且字符使用空格替代（Insert spaces for tabs）。这样我们在使用 TAB 键进行代码对齐操作的时候，都会用空格替代，保证不同软件使用代码都可以对齐。

然后，选择：Colors & Fonts 选项卡，在该选项卡内，我们就可以设置自己的代码的字体和颜色了。由于我们使用的是 C 语言，故在 Window 下面选择：C/C++ Editor files 在右边就可以看到相应的元素了。如图 4.4.1.3 示：

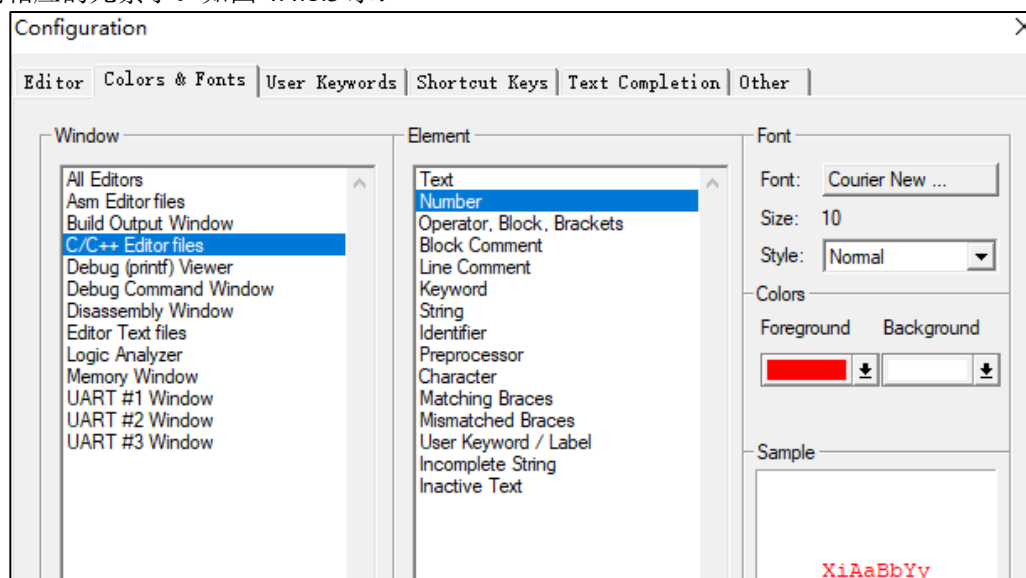


图 4.4.1.3 Colors & Fonts 选项卡

然后点击各个元素（Element）修改为你喜欢的颜色（注意双击，且有时候可能需要设置多次才生效，MDK 的 bug），当然也可以在 Font 栏设置你字体的类型，以及字体的大小等。

然后，点击 User Keywords 选项卡，设置用户定义关键字，以便用户自定义关键字也显示对应的颜色（对应图 4.4.1.3 中的 User Keyword/Lable 颜色）。在 User Keywords 选项卡对话框下面输入你自己定义的关键字，如图 4.4.1.4 示：

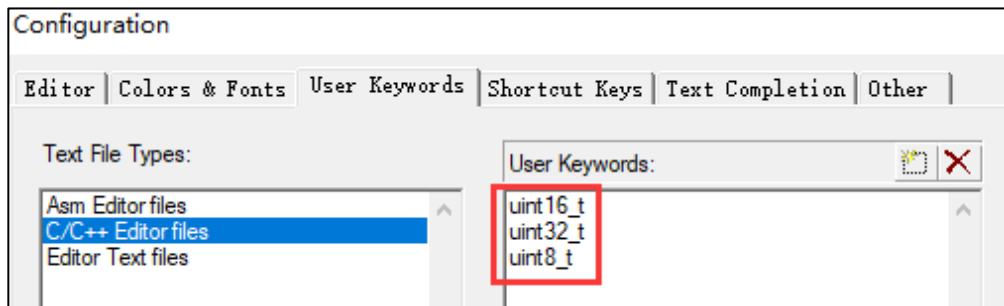


图 4.4.1.4 用户自定义关键字

这里我们设置了 uint8_t、uint16_t 和 uint32_t 等三个用户自定义关键字，相当于 unsigned char、unsigned short 和 unsigned int。如果你还有其他自定义关键字，在这里添加即可。设置成之后，点击 OK，就可以在主界面看到你所修改后的结果，例如我修改后的代码显示效果如图 4.4.1.5 示：

```

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

```


图 4.4.1.5 设置完后显示效果

这就比开始的效果好看一些了。字体大小，则可以直接按住：ctrl+鼠标滚轮，进行放大或者缩小，或者也可以在刚刚的配置界面设置字体大小。

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

其实在这个编辑配置对话框里，还可以对其他很多功能进行设置，比如动态语法检测等，我们将在 4.4.2 节介绍。

4.4.2 语法检测&代码提示

MDK4.70 以上的版本，新增了代码提示与动态语法检测功能，使得 MDK 的编辑器越来越好用了，这里我们简单说一下如何设置，同样，点击 ，打开配置对话框，选择 Text Completion 选项卡，如图 4.4.2.1 所示：

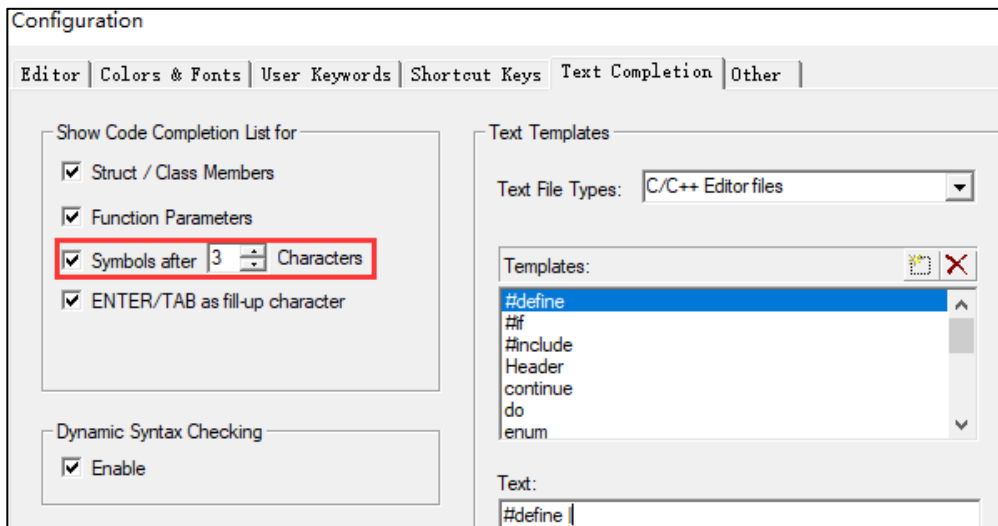


图 4.4.2.1 Text Completion 选项卡设置

Strut / Class Members, 用于开启结构体/类成员提示功能。

Function Parameters, 用于开启函数参数提示功能。

Symbols after xx characters, 用于开启代码提示功能, 即在输入多少个字符以后, 提示匹配的内容(比如函数名字、结构体名字、变量名字等), 这里默认设置 3 个字符以后, 就开始提示。如图 4.4.2.2 所示:

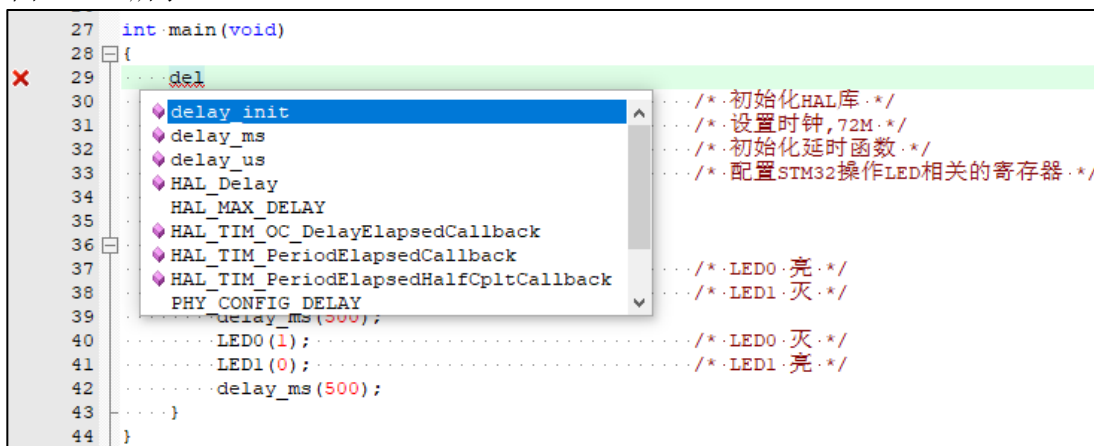




图 4.4.2.2 代码提示

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

Dynamic Syntax Checking, 则用于开启动态语法检测, 比如编写的代码存在语法错误的时候, 会在对应行前面出现  图标, 如出现警告, 则会出现  图标, 将鼠标光标放图标上面, 则会提示产生的错误/警告的原因, 如图 4.4.2.3 所示:

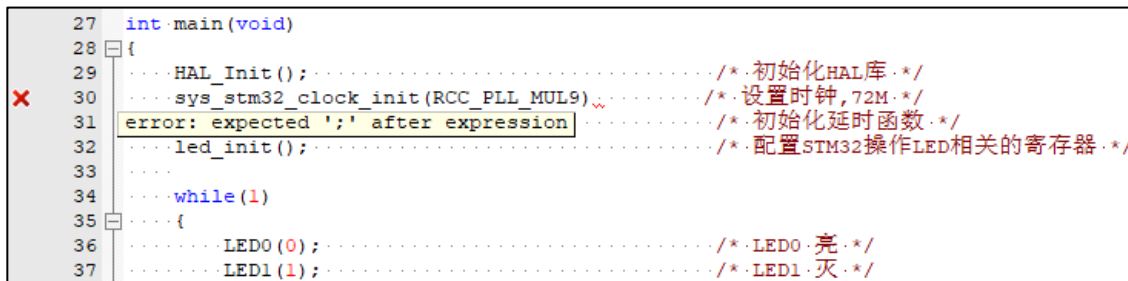


图 4.4.2.3 语法动态检测功能

这几个功能, 对我们编写代码很有帮助, 可以加快代码编写速度, 并且及时发现各种问题。不过这里要提醒大家, 语法动态检测这个功能, 有的时候会误报(比如 sys.c 里面, 就有误报), 大家可以不用理会, 只要能编译通过(0 错误, 0 警告), 这样的语法误报, 一般直接忽略即可。

4.4.3 代码编辑技巧

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

1. TAB 键的妙用

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

假设我们前面的串口 1 中断回调函数如图 4.4.3.1 所示：

```
149 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
150 {
151     if (huart->Instance == USART_1) ...../*如果是串口1*/
152     {
153         if ((g_usart_rx_sta & 0x8000) == 0) ...../*接收未完成*/
154         {
155             if (g_usart_rx_sta & 0x4000) ...../*接收到了0x0d*/
156             {
157                 if (aRxBuffer[0] != 0x0a)
158                 {
159                     g_usart_rx_sta = 0; ...../*接收错误,重新开始*/
160                 }
161             }
162             else
163             {
164                 g_usart_rx_sta |= 0x8000; ...../*接收完成了*/
165             }
166         }
167         else ...../*还没收到0x0d*/
168         {
169             if (aRxBuffer[0] == 0x0d)
170             {
171                 g_usart_rx_sta |= 0x4000;
172             }
173             else
174             {
175                 g_usart_rx_buf[g_usart_rx_sta & 0X3FFF] = aRxBuffer[0];
176                 g_usart_rx_sta++;
177                 if (g_usart_rx_sta > (USART_REC_LEN - 1))
178                 {
179                     g_usart_rx_sta = 0; ...../*接收数据错误,重新开始接收.....*/
180                 }
181             }
182         }
183     }
```

图 4.4.3.1 头大的代码

上图的代码很不规范，这还只是短短的 30 来行代码，如果你的代码有几千行，全部是这个样子，不头大才怪。这时我们就可以通过 TAB 键的妙用来快速修改为比较规范的代码格式。

选中一块然后按 TAB 键，你可以看到整块代码都跟着右移了一定距离，如图 4.4.3.2 所示：

```

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

```

图 4.4.3.2 代码整体偏移

接下来我们就是要多选几次，然后多按几次 TAB 键就可以达到迅速使代码规范化的目的，最终效果如图 4.4.3.3 所示

```

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

```

图 4.4.3.3 修改后的代码

图 4.4.3.3 中的代码相对于图 4.4.3.1 中的要好看多了，经过这样的整理之后，整个代码一下就变得有条理多了，看起来很舒服。

2. 快速定位函数/变量被定义的地方

上一节，我们介绍了 TAB 键的功能，接下来我们介绍一下如何快速查看一个函数或者变量所定义的地方。

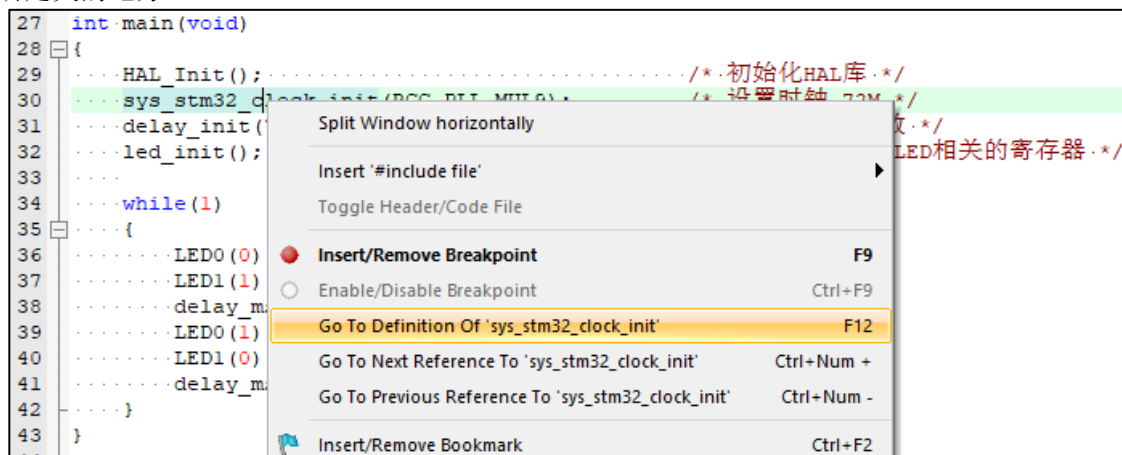


图 4.4.3.4 快速定位

大家在调试代码或编写代码的时候，一定想看看某个函数是在那个地方定义的，具体里面的内容是怎么样的，也可能想看看某个变量或数组是在哪个地方定义的等。尤其在调试代码或者看别人代码的时候，如果编译器没有快速定位的功能的时候，你只能慢慢的自己找，代码量比较少还好，如果代码量一大，那就郁闷了，有时候要花很久的时间来找这个函数到底在哪里。幸好 MDK 提供了这样的快速定位的功能。只要你把光标放到这个函数/变量（xxx）的上面（xxx 为你想要查看的函数或变量的名字），然后右键，可以看到如图 4.4.3.4 所示的菜单栏。

在图 4.4.3.4 中，我们找到 Go to Definition Of ‘sys_stm32_clock_init’ 这个地方，然后单击左键就可以快速跳到 sys_stm32_clock_init 函数的定义处（注意要先在 Options for Target 的 Output 选项卡里面勾选 Browse Information 选项，再编译，再定位，否则无法定位！）。如图 4.4.3.5 所示：

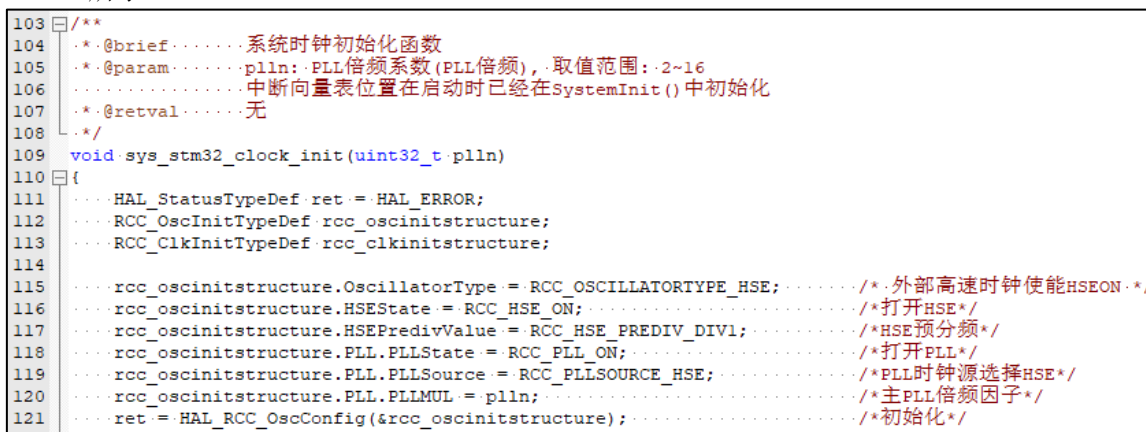



图 4.4.3.5 定位结果

对于变量，我们也可以按这样的操作快速来定位这个变量被定义的地方，大大缩短了你查找代码的时间。

很多时候，我们利用 Go to Definition 看完函数/变量的定义后，又想返回之前的代码继续看，此时我们可以通过 IDE 上的  按钮（Back to previous position）快速的返回之前的位置，这个按钮非常好用！

3. 快速注释与快速消注释

接下来，我们介绍一下快速注释与快速消注释的方法。在调试代码的时候，你可能会想注释某一片的代码，来看看执行的情况，MDK 提供了这样的快速注释/消注释块代码的功能。也

是通过右键实现的。这个操作比较简单，就是先选中你要注释的代码区，然后右键，选择 Advanced→Comment Selection 就可以了。

以 led_init 函数为例，比如我要注释掉下图中所选中区域的代码，如图 4.4.3.6 所示：

```

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

```

图 4.4.3.6 选中要注释的区域

我们只要在选中了之后，选择右键，再选择 Advanced→Comment Selection 就可以把这段代码注释掉了。执行这个操作以后的结果如图 4.4.3.7 所示：

```

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

```

图 4.4.3.7 注释完毕

这样就快速的注释掉了一片代码，而在某些时候，我们又希望这段注释的代码能快速的取消注释，MDK 也提供了这个功能。与注释类似，先选中被注释掉的地方，然后通过右键→Advanced，不过这里选择的是 Uncomment Selection。

4.4.4 其他小技巧

除了前面介绍的几个比较常用的技巧，这里还介绍几个其他的小技巧，希望能让你的代码编写如虎添翼。

第一个是快速打开头文件。在将光标放到要打开的引用头文件上，然后右键选择 Open Document “XXX”，就可以快速打开这个文件了（XXX 是你打开的头文件名字）。如图 4.4.4.1 所示：

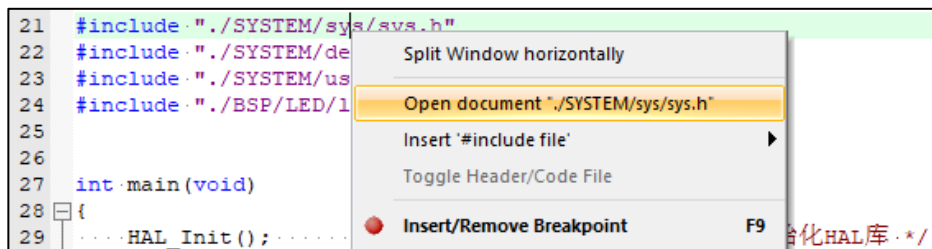


图 4.4.4.1 快速打开头文件

第二个小技巧是查找替换功能。这个和 WORD 等很多文档操作的替换功能是差不多的，在 MDK 里面查找替换的快捷键是“CTRL+H”，只要你按下该按钮就会调出如图 4.4.4.2 所示界面：

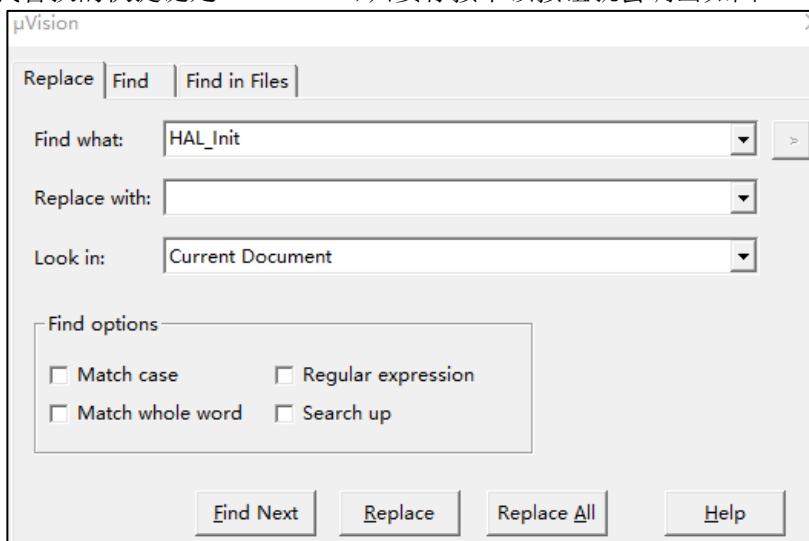


图 4.4.4.2 替换文本

这个替换的功能在有的时候是很有用的，它的用法与其他编辑工具或编译器的差不多，相信各位都不陌生了，这里就不啰嗦了。

第三个小技巧是跨文件查找功能，先双击你要找的函数/变量名（这里以系统时钟初始化函数：sys_stm32_clock_init 为例），然后再点击 IDE 上面的 ，弹出如图 4.4.4.3 所示对话框：

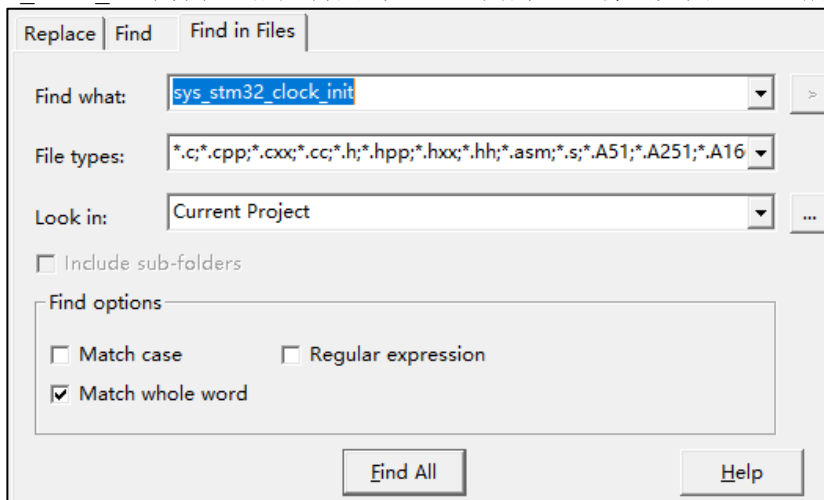


图 4.4.4.3 跨文件查找

点击 Find All，MDK 就会帮你找出所有含有 Stm32_Clock_Init 字段的文件并列其所在位置，如图 4.4.4.4 所示：

```
Find in Files
Searching for 'sys_stm32_clock_init'...
J:\Source Code\F103ZE\实验1 跑马灯实验\User\main.c(31) :      sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟,72M */
J:\Source Code\F103ZE\实验1 跑马灯实验\Drivers\SYSTEM\sys\sys.h(45) : void sys_stm32_clock_init(uint32_t pllcn);
J:\Source Code\F103ZE\实验1 跑马灯实验\Drivers\SYSTEM\sys\sys.c(109) : void sys_stm32_clock_init(uint32_t pllcn)
Lines matched: 3      Files matched: 3      Total files searched: 79
```

图 4.4.4.4 查找结果

该方法可以很方便的查找各种函数/变量，而且可以限定搜索范围（比如只查找.c 文件和.h 文件等），是非常实用的一个技巧。

第五章 STM32 基础知识入门

本章，我们着重介绍 STM32 的一些基础知识，让大家对 STM32 开发有一个初步的了解，为后面 STM32 的学习做铺垫，方便后面的学习。本章内容大家第一次看的时候可以只了解一个大概，后面需要用到这方面的知识的时候再回过头来仔细看看。

本章将分为如下几个小节：

- 5.1 C 语言基础知识复习
- 5.2 寄存器基础知识
- 5.3 STM32F103 系统架构
- 5.4 存储器映射
- 5.5 寄存器映射

5.1 C 语言基础知识复习

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

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

5.1.1 位操作

C 语言位操作相信学过 C 语言的人都不陌生了,简而言之,就是对基本类型变量可以在位级别进行操作。这节的内容很多朋友都应该很熟练了，我这里也就点到为止，不深入探讨。下面我们先讲解几种位操作符，然后讲解位操作使用技巧。C 语言支持如下 6 种位操作：

| 运算符 | 含义 | 运算符 | 含义 |
|-----|------|-----|------|
| & | 按位与 | ~ | 按位取反 |
| | 按位或 | << | 左移 |
| ^ | 按位异或 | >> | 右移 |

表 5.1.1.1 六种位操作

这些与或非，取反，异或，右移，左移这些到底怎么回事，这里我们就不多做详细，相信大家学 C 语言的时候都学习过了。如果不懂的话，可以百度一下，非常多的知识讲解这些操作符。下面我们想着重讲解位操作在单片机开发中的一些实用技巧。

1，在不改变其他位的值的情况下，对某几个位进行设置。

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

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

然后再与需要设置的值进行|或运算：

```
GPIOA->CRL |= 0x00000040; /* 设置 bit6 的值为 1，不改变其他位的值 */
```

2，移位操作提高代码的可读性。

移位操作在单片机开发中非常重要，下面是 delay_init 函数的一行代码：

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

这个操作就是将 CTRL 寄存器的第 1 位（从 0 开始算起）设置为 1，为什么要通过左移而不是直接设置一个固定的值呢？其实这是为了提高代码的可读性以及可重用性。这行代码可以很直观明了的知道，是将第 1 位设置为 1。如果写成：

```
SysTick->CTRL |= 0x0002;
```

这个虽然也能实现同样的效果，但是可读性稍差，而且修改也比较麻烦。

3，~按位取反操作使用技巧

按位取反在设置寄存器的时候经常被使用，常用于清除某一个/某几个位。下面是 `delay_us` 函数的一行代码：

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

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

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

可见前者的可读性，及可维护性都要比后者好很多。

4. ^按位异或操作使用技巧

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

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

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

5.1.2 define 宏定义

`define` 是 C 语言中的预处理命令，它用于宏定义，可以提高源代码的可读性，为编程提供方便。常见的格式：

```
#define 标识符 字符串
```

“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。例如：

```
#define HSE_VALUE 8000000U
```

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

至于 `define` 宏定义的其他一些知识，比如宏定义带参数这里我们就不多讲解。

5.1.3 ifdef 条件编译

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

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

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

```
#ifdef
    程序段 1
#endif
```

条件编译在 MDK 里面是用得很多，在 `stm32f10x.h` 这个头文件中经常会看到这样的语句：

```
#if !defined (STM32F1)
#define STM32F1
#endif
```

如果没有定义 `STM32F1` 这个宏，则定义 `STM32F1` 宏。条件编译也是 c 语言的基础知识，这里也就点到为止吧。

5.1.4 extern 外部申明

C 语言中 `extern` 可以置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。这里面要注意，对于 `extern` 申明变量可以多次，但定义只有一次。在我们的代码中你会看到这样的语句：

```
extern uint16_t g_usart_rx_sta;
```

这个语句是申明 `g_usart_rx_sta` 变量在其他文件中已经定义了，在这里要使用到。所以，你肯定可以找到在某个地方有变量定义的语句：

```
uint16_t g_usart_rx_sta;
```

`extern` 的使用比较简单，但是也会经常用到，需要掌握。

5.1.5 typedef 类型别名

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

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

定义了一个结构体 GPIO，这样我们定义结构体变量的方式为：

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

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

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

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

```
GPIO_TypeDef gpiox;
```

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

5.1.6 结构体

经常很多用户提到，他们对结构体使用不是很熟悉，但是 MDK 中太多地方使用结构体以及结构体指针，这让他们一下子摸不着头脑，学习 STM32 的积极性大大降低，其实结构体并不是那么复杂，这里我们稍微提一下结构体的一些知识，还有一些知识我们会在下面的“寄存器映射”中讲到一些。

声明结构体类型：

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

例如：

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

在结构体声明的时候可以定义变量，也可以申明之后定义，方法是：

```
struct 结构体名字 结构体变量列表 ;
```

例如：

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

结构体成员变量的引用方法是：

结构体变量名字.成员名

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

例如：

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

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

```
usart3->BaudRate;
```

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

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

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

这种方式是有效的同时在一定场合是可取的。但是试想，如果有一天，我们希望往这个函数里面再传入一个/几个参数，那么势必我们需要修改这个函数的定义，重新加入新的入口参数，随着开发不断的增多，那么是不是我们就要不断的修改函数的定义呢？这是不是给我们开发带来很多的麻烦？那又怎样解决这种情况呢？

我们使用结构体参数，就可以在不改变入口参数的情况下，只需要改变结构体的成员变量，就可以达到改变入口参数的目的。

结构体就是将多个变量组合为一个有机的整体，上面的函数，usartx,BaudRate,Parity, Mode等这些参数，他们对于串口而言，是一个有机整体，都是来设置串口参数的，所以我们可以将他们通过定义一个结构体来组合在一个。MDK 中是这样定义的：

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

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

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

这样，任何时候，我们只需要修改结构体成员变量，往结构体中间加入新的成员变量，而不需要修改函数定义就可以达到修改入口参数同样的目的了。这样的好处是不用修改任何函数定义就可以达到增加变量的目的。

理解了结构体在这个例子中间的作用吗？在以后的开发过程中，如果你的变量定义过多，如果某几个变量是用来描述某一个对象，你可以考虑将这些变量定义在结构体中，这样也许可以提高你的代码的可读性。

使用结构体组合参数，可以提高代码的可读性，不会觉得变量定义混乱。当然结构体的作用就远远不止这个了，同时，MDK 中用结构体来定义外设也不仅仅只是这个作用，这里我们只是举一个例子，通过最常用的场景，让大家理解结构体的一个作用而已。后面一节我们还会讲解结构体的一些其他知识。

5.1.7 指针

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

申明指针我们一般以 p 开头，如：

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

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

```
int main(void)
{
    uint8_t temp = 0x88;          /* 定义变量 temp */
    uint8_t *p_num = &temp;      /* 定义指针 p_num, 指向 temp 的地址 */
}
```

```
HAL_Init(); /* 初始化 HAL 库 */
sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72M */
delay_init(72); /* 延时初始化 */
usart_init(115200); /* 初始化串口 */
printf("temp:0X%X\r\n", temp); /* 打印 temp 的值 */
printf("*p_num: 0X %X\r\n", *p_num); /* 打印 *p_num 的值 */
printf("p_num: 0X %X\r\n", (uint32_t)p_num); /* 打印 p_num 的值 */
printf("&p_num: 0X %X\r\n", (uint32_t)&p_num); /* 打印 &p_num 的值 */
while (1);
}
```

此代码的输出结果为:

```
ATK
XCOM
temp:0X88
*p_num:0X88
p_num:0X20000774
&p_num:0X20000770
```

图 5.1.7.1 输出结果

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

*p_num: 取 p_num 指向的地址所存储的值, 即 temp 的值。

&p_num: 取 p_num 指针的地址, 即指针自身的地址。

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

5.2 寄存器基础知识

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

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

从大方向来区分, STM32 寄存器分为两类, 如表 5.2.1 所示:

| 大类 | 小类 | 说明 |
|-------|-------------|---|
| 内核寄存器 | 内核相关寄存器 | 包含 R0~R15、xPSR、特殊功能寄存器等 |
| | 中断控制寄存器 | 包含 NVIC 和 SCB 相关寄存器, NVIC 有: ISER、ICER、ISPR、IP 等; SCB 有: VTOR、AIRCRCR、SCR 等 |
| | SysTick 寄存器 | 包含 CTRL、LOAD、VAL 和 CALIB 四个寄存器 |
| | 内存保护寄存器 | 可选功能, STM32F103 没有 |
| | 调试系统寄存器 | ETM、ITM、DWT、IPIU 等相关寄存器 |
| 外设寄存器 | | 包含 GPIO、UART、IIC、SPI、TIM、DMA、ADC、DAC、RTC、I/WWDG、PWR、CAN、USB 等各种外设寄存器 |

表 5.2.1 STM32 寄存器分类

其中, 内核寄存器, 我们一般只需要关心中断控制寄存器和 SysTick 寄存器即可, 其他三大类, 我们一般很少直接接触。而外设寄存器, 则是学到哪个外设, 就了解哪个外设相关寄存器即可, 所以整体来说, 我们需要关心的寄存器并不是很多, 而且很多都是有共性的, 比如 STM32F103ZET6 有 8 个定时器, 我们只需要学习了其中一个的相关寄存器, 其他 7 个基本都是一样。

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

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

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

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

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

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

5.3 STM32F103 系统架构

STM32F103 是 ST 公司基于 ARM 授权 Cortex M3 内核而设计的一款芯片，而 Cortex M 内核使用的是 ARM v7-M 架构，是为了替代老旧的单片机而量身定做的一个内核，具有低成本、低功耗、实时性好、中断响应快、处理效率高等特点。

5.3.1 Cortex M3 内核 & 芯片

ARM 公司提供内核（如 Cortex M3，简称 CM3，下同）授权，完整的 MCU 还需要很多其他组件。芯片公司（ST、NXP、TI、GD、华大等）在得到 CM3 内核授权后，就可以把 CM3 内核用在自己的硅片设计中，添加：存储器，外设，I/O 以及其它功能块。不同厂家设计出的单片机会有不同的配置，包括存储器容量、类型、外设等都各具特色，因此才会有市面上各种不同应用的 ARM 芯片。Cortex M3 内核和芯片的关系如图 5.3.1.1 所示：

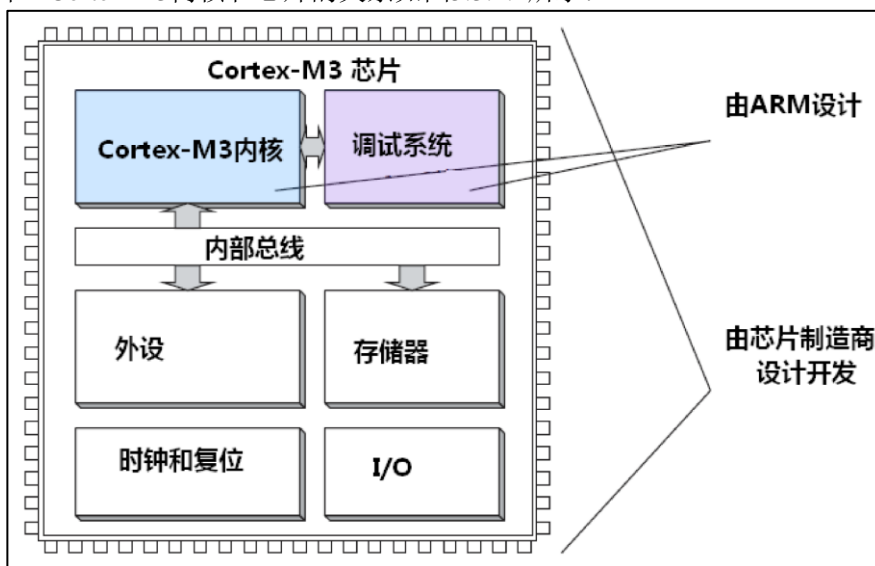


图 5.3.1.1 Cortex M3 内核 & 芯片关系

可以看到，ARM 公司提供 CM3 内核和调试系统，其他的（外设（IIC、SPI、UART、TIM 等）、存储器（SRAM、FLASH 等）、I/O 等）由芯片制造商设计开发。这里 ST 公司就是 STM32F103 芯片的制造商。

5.3.2 STM32 系统架构

STM32F103ZET6 内部系统结构如图 5.3.2.1：

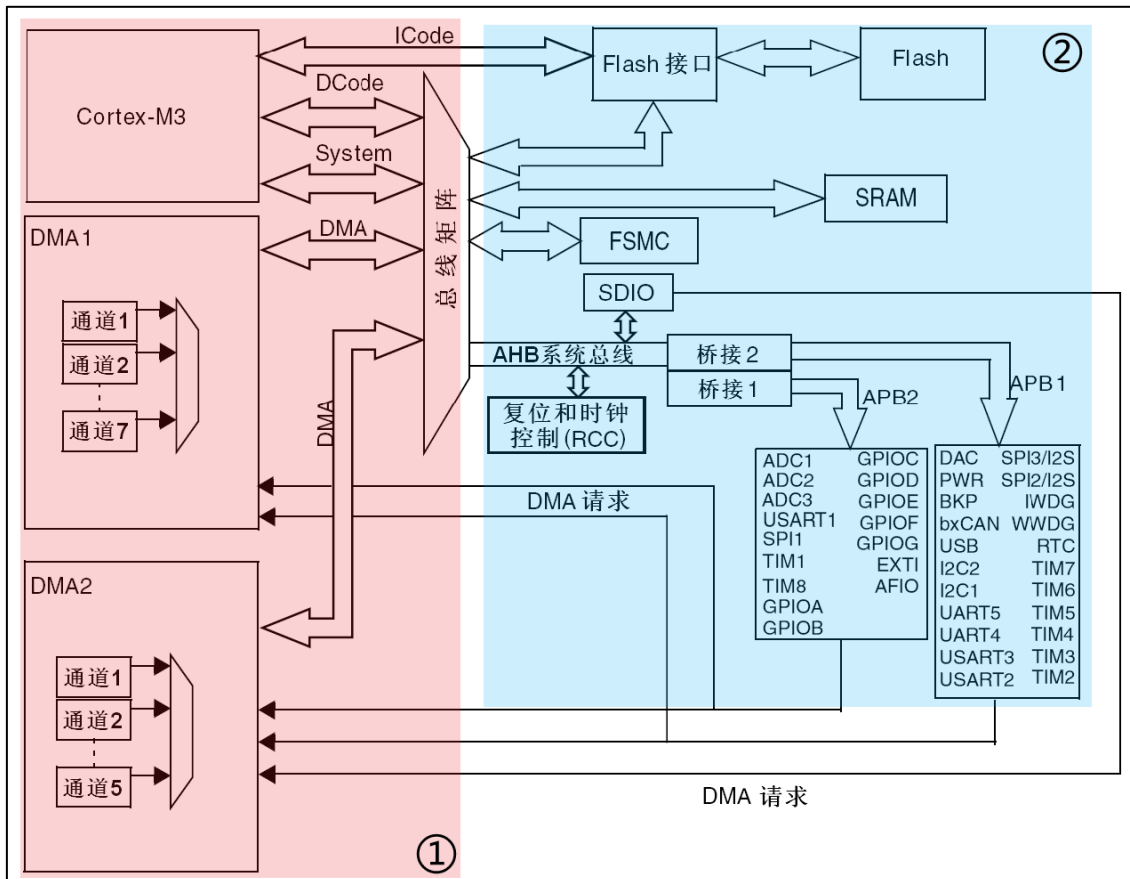


图5.3.2.1 STM32F103系统结构

STM32F103的系统主要由：**四个驱动单元**（可以主动发起通信，图中①区域）和**四个被动单元**（只能被驱动工作，图中②区域）组成，如表5.3.2.1所示：

| 驱动单元 | 被动单元 |
|-------------------------------|-----------------------------|
| Cortex M3 内核 DCode 总线 (D-Bus) | 内部 FLASH |
| Cortex M3 内核 系统总线 (S-Bus) | 内部 SRAM |
| 通用 DMA1 | FSMC |
| 通用 DMA2 | AHB 到 APB 的桥, 所连接的所有 APB 设备 |

表5.3.2.1 STM32F103单元结构

这里的驱动/被动单元都是指连接了总线矩阵的部分，未连接总线矩阵的部分，则不算作驱动/被驱动单元。接下来我们介绍一下这些单元。

1. I Code 总线 (I - Bus)

这是Cortex M3内核的指令总线，连接闪存指令接口（如：FLASH），用于获取指令。由于该总线功能单一，并没有直接连接到总线矩阵，因此被排除在驱动单元之外。

2. D Code 总线 (D - Bus)

这是Cortex M3内核的数据总线，连接闪存存储器数据接口（如：SRAM、FLASH等），用于各种数据访问，如常量、变量等。

3. 系统总线 (S-Bus)

这是Cortex M3内核的系统总线，连接所有外设（如：GPIO、SPI、IIC、TIM等），用于控制各种外设工作，如配置各种外设相关寄存器等。

4. DMA 总线

DMA是直接存储访问控制器，可以实现数据的自动搬运，整个过程不需要CPU处理。如可以实现DMA传输内存数据到DAC，输出任意波形，传输过程不需要CPU参与，可以大大节省CPU支，从而更高效的处理事务。STM32F103ZET6内部有2个DMA控制器，可以实现内存到外设、外设到内存、内存到内存的数据传输。

5. 内部 FLASH

内部FLASH即单片机的硬盘，用于代码/数据存储，CPU通过ICode总线经FLASH接口访问内部FLASH，FLASH最高访问速度是24Mhz，因此以72M速度访问时，需要插入2个时钟周期延迟。

6. 内部 SRAM

内部SRAM即单片机的内存，用于数据存储，直接挂载在总线矩阵上面，CPU通过DCode总线实现0等待延时访问SRAM，最快总线频率可达72Mhz，从而保证高效高速的访问内存。

7. FSMC

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

8. AHB/APB 桥

AHB总线连接总线矩阵，同时通过2个APB桥连接APB1和APB2，AHB总线速度最大为72Mhz，APB2总线速度最大也是72Mhz，但是APB1总线速度最大只能是36Mhz。这三个总线上面挂载了STM32内部绝大部分外设。

9. 总线矩阵

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

5.3.3 存储器映射

STM32是一个32位单片机，他可以很方便的访问4GB以内的存储空间（ $2^{32}=4GB$ ），因此Cortex M3内核将图5.3.2.1中的所有结构，包括：FLASH、SRAM、外设及相关寄存器等全部组织在同一个4GB的线性地址空间内，我们可以通过C语言来访问这些地址空间，从而操作相关外设（读/写）。数据字节以小端格式（小端模式）存放在存储器中，数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中。

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

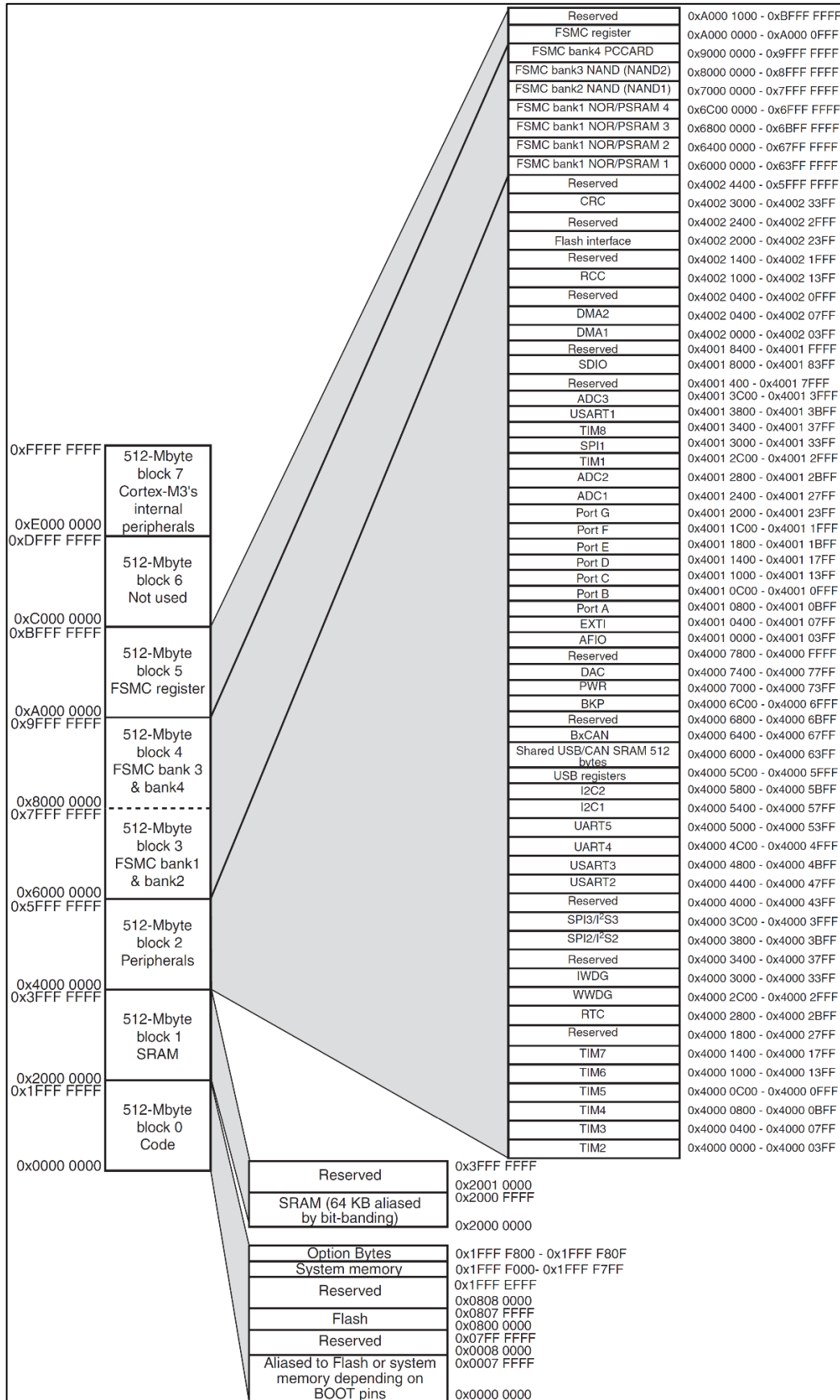


图5.3.3.1 STM32F103存储器映射

存储块功能介绍

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

| 存储块 | 功能 | 地址范围 |
|---------|----------------|----------------------------------|
| Block 0 | Code | 0X0000 0000 ~ 0x1FFF FFFF（512MB） |
| Block 1 | SRAM | 0X2000 0000 ~ 0x3FFF FFFF（512MB） |
| Block 2 | 外设 | 0X4000 0000 ~ 0x5FFF FFFF（512MB） |
| Block 3 | FSMC Bank1&2 | 0X6000 0000 ~ 0x7FFF FFFF（512MB） |
| Block 4 | FSMC Bank3&4 | 0X8000 0000 ~ 0x9FFF FFFF（512MB） |
| Block 5 | FSMC 寄存器 | 0XA000 0000 ~ 0xBFFF FFFF（512MB） |
| Block 6 | 没用到 | 0XC000 0000 ~ 0xDFFF FFFF（512MB） |
| Block 7 | Cortex M3 内部外设 | 0XE000 0000 ~ 0xFFFF FFFF（512MB） |

表5.3.3.1 STM32 存储块功能及地址范围

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

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

表5.3.3.2 STM32 存储块0的功能划分

可以看到，我们用户FLASH大小是512KB，这是对于我们使用的STM32F103ZET6来说，如果是其他型号，可能FLASH会更小，当然，如果ST喜欢，也是可以随时推出更大容量的STM32F103单片机的，因为这里保留了一大块地址空间。还有，STM32的出厂固化BootLoader非常精简，整个BootLoder只占了2KB FLASH空间。

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

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

表5.3.3.3 STM32 存储块1的功能划分

这个块仅使用了64KB大小（仅大容量STM32F103型号才有这么多SRAM，比如STM32F103ZET6等），用于SRAM访问，同时也有大量保留地址用于扩展。

第三个块是Block 2，用于外设访问，STM32内部大部分的外设都是放在这个块里面的，该存储块里面包括了AHB、APB1和APB2三个总线相关的外设，其中AHB和APB2是高速总线（72Mhz max），APB1是低速总线（36M max）。其功能划分如表5.3.3.4所示：

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

表5.3.3.4 STM32 存储块2的功能划分

同样可以看到，各个总线之间，都有预留地址空间，方便后续扩展。关于STM32各个外设具体挂在哪个总线上面，大家可以参考前面的 STM32F103系统结构图 和 STM32F103存储器映射图 进行查找对应。

5.3.4 寄存器映射

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

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

当然STM32内部的寄存器有非常多，远远不止8个开关这么简单，但是原理是差不多的，每个寄存器的每一个位，一般都有特定的作用，涉及到寄存器描述，大家可以参考《STM32F10XXX参考手册（中文版）_V10.pdf》相关章节的寄存器描述部分，有详细的描述。

1. 寄存器描述解读

我们以GPIO的ODR寄存器为例，其参考手册的描述如图5.3.4.1所示：

| 8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E) ① | | | | | | | | | | | | | | | |
|---------------------------------------|-------|--|-------|-------|-------|------|------|------|------|------|------|------|------|------|------|
| 地址偏移：0Ch ② | | | | | | | | | | | | | | | |
| 复位值：0x0000 0000 | | | | | | | | | | | | | | | |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| ④ 位31:16 | | 保留，始终读为0。 | | | | | | | | | | | | | |
| ④ 位15:0 | | ODRy[15:0]：端口输出数据(y = 0...15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注：对GPIOx_BSRR(x = A..E)，可以分别地对各个ODR位进行独立的设置/清除。 | | | | | | | | | | | | | |

图5.3.4.1 端口输出数据寄存器描述

① 寄存器名字

每个寄存器都有一个对应的名字，以简单表达其作用，并方便记忆，这里GPIOx_ODR表示寄存器英文名，x可以从A~E，说明有5个这样的寄存器（每个端口有一个，事实上最新的STM32F103型号，可能还有F,G等端口，IO数量更多）。

② 寄存器偏移量及复位值

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

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

③ 寄存器位表

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

④ 位功能描述

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

其他寄存器描述，参照以上方法解读接口。

2. 寄存器映射举例

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

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

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

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

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

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

当然，为了简单，我们只举了一个简单实例，实际上大量寄存器的映射，使用结构体是最方便的方式，stm32f103xe.h 里面使用结构体方式对 STM32F103 的寄存器做了详细映射，等下我们再介绍。

3. 寄存器地址计算

STM32F103大部分外设寄存器地址都是在存储块2上面的，见图5.3.3.1。具体某个寄存器地址，由三个参数决定：1、总线基地址（BUS_BASE_ADDR）；2，外设基于总线基地址的偏移量（PERIPH_OFFSET）；3，寄存器相对外设基地址的偏移量（REG_OFFSET）。可以表示为：

寄存器地址 = BUS_BASE_ADDR + PERIPH_OFFSET + REG_OFFSET

总线基地址（BUS_BASE_ADDR），STM32F103内部有三个总线（APB1、APB2和AHB），对应的总线基地址如表5.3.4.1所示：

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

表5.3.4.1 总线基地址

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

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

外设基于总线基地址的偏移量（PERIPH_OFFSET），这个不同外设偏移量不一样，我们可以在STM32F103存储器映射图（图5.3.3.1）里面找到具体的偏移量，以GPIO为例，其偏移量如表5.3.4.2所示：

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

表5.3.4.2 GPIO外设基地址及相对总线偏移量

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

知道了外设基地址，再在参考手册里面找到具体某个寄存器相对外设基地址的偏移量就可以知道该寄存器的实际地址了，以GPIOB的相关寄存器为例，如表5.3.4.3所示：

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

表 5.3.4.3 GPIOB 寄存器相对外设基地址的偏移量

上表的偏移量，就是寄存器基于外设基地址的偏移量（REG_OFFSET）。

因此，我们根据前面的公式，很容易可以计算出GPIOB_ODR的地址：

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

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

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

4. stm32f103xe.h 寄存器映射说明

STM32F103所有寄存器映射都在stm32f103xe.h里面完成，包括各种基地址定义、结构体定义、外设寄存器映射、寄存器位定义（占了绝大部分）等，整个文件有1W多行，非常庞大。我们不需要对该文件进行全面分析，因为很多内容都是相似的，我们只需要知道寄存器是如何被映射的，就可以了，至于寄存器位定义这些内容，知道是怎么回事就可以了。

我们还是以GPIO为例进行说明，看看stm32f103xe.h是如何对GPIO的寄存器进行映射的，通过对GPIO寄存器映射，了解stm32f103xe.h的映射规则。

stm32f103xe.h文件主要包含五个部分内容，如表5.3.4.4所示：

| 文件 | 主要组成部分 | 说明 |
|---------------|--------------|---|
| stm32f103xe.h | 中断编号定义 | 定义 IRQn_Type 枚举类型，包含 STM32F103 内部所有中断编号（中断号），方便后续编写代码 |
| | 外设寄存器结构体类型定义 | 以外设为基本单位，使用结构体类型定义对每个外设的所有寄存器进行封装，方便后面的寄存器映射 |
| | 寄存器映射 | 1，定义总线地址和外设基地址 2，使用外设结构体类型定义将外设基地址强制转换成结构体指针，完成寄存器映射 |
| | 寄存器位定义 | 定义外设寄存器每个功能位的位置及掩码 |
| | 外设判定 | 判断某个外设是否合法（即是否存在该外设） |

表5.3.4.4 stm32f103xe.h文件主要组成部分

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

- 1， 外设寄存器结构体类型定义
- 2， 外设基地址定义
- 3， 寄存器映射（通过将外设基地址强制转换为外设结构体类型指针即可）

以GPIO为例，其寄存器结构体类型定义如下：

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

```
__IO uint32_t LCKR; /* GPIO_LCKR 寄存器，相对外设基地址偏移量：0x18 */
} GPIO_TypeDef;
```

GPIO外设基地址定义如下：

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

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

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

GPIO外设寄存器映射定义如下：

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

以上三部分代码，就完成了STM32F103内部GPIOA~GPIOG的寄存器映射，其原理其实是比较简单的，包括两个核心知识点：1，结构体地址自增；2，地址强制转换；

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

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

这样我们就使用结构体方式完成了对GPIO寄存器的映射，其他外设的寄存器映射也都是这个方法，这里就不一一介绍了。关于stm32f103xe.h的寄存器映射，我们就介绍到这里。

第六章 新建寄存器版本 MDK 工程

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

本章将分为如下两个小节：

6.1 新建寄存器版本 MDK 工程

6.2 下载验证

6.1 新建寄存器版本 MDK 工程

本节我们将教大家如何新建一个 STM32F103 的 MDK5 工程。为了方便大家参考，我们将本节最终新建好的工程模板存放在 A 盘：4、程序源码\2，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-2，新建工程实验-寄存器版本，如遇新建工程问题，请打开该实验对比。

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

在新建工程之前，首先我们要做如下准备：

1，STM32Cube 官方固件包：我们使用的固件包版本是 STM32Cube_FW_F1_V1.8.3，固件包路径：**A 盘→8，STM32 参考资料→1，STM32CubeF1 固件包。**

1，开发环境搭建：参考本书第三章相关内容。

6.1.1 新建工程文件夹

新建工程文件夹分为 2 个步骤：1，新建工程文件夹；2，拷贝工程相关文件。

1. 新建工程文件夹

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

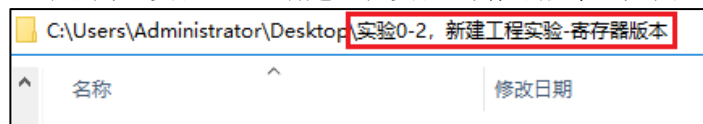


图 6.1.1.1 新建工程根目录文件夹

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

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

表 6.1.1.1 工程根目录新建文件夹及其作用

新建完成以后，最后得到我们的工程根目录文件夹如图 6.1.1.2 所示。

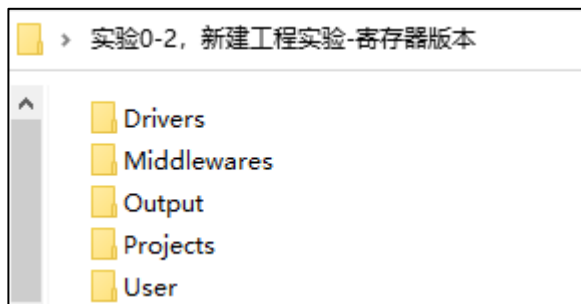


图 6.1.1.2 工程根目录文件夹

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

2. 拷贝工程相关文件

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

Drivers 文件夹

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

| 文件夹名称 | 作用 |
|--------|---|
| BSP | 存放开发板板级支持包驱动代码，如各种外设驱动 |
| CMSIS | 存放 CMSIS 底层代码，如启动文件（.s 文件）、stm32f1xx.h 等 |
| SYSTEM | 存放正点原子系统级核心驱动代码，如 sys.c、delay.c 和 usart.c 等 |

表 6.1.1.2 Drivers 包含文件夹

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

CMSIS 文件夹，用于存放 CMSIS 底层代码（ARM 和 ST 提供），如：启动文件（.s 文件）、stm32f1xx.h 等各种头文件。该文件夹我们可以直接从 STM32CubeF1 固件包（路径：A 盘→8，STM32 参考资料→1，STM32CubeF1 固件包）里面拷贝，不过由于固件包里面的 CMSIS 兼容了太多芯片，导致非常大（100 多 MB），因此我们根据实际情况，对其进行了大幅精简，精简后的 CMSIS 文件夹大小为 1MB 左右。精简后的 CMSIS 文件夹大家可以在：A 盘→4，程序源码→1，标准例程-寄存器版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

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

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

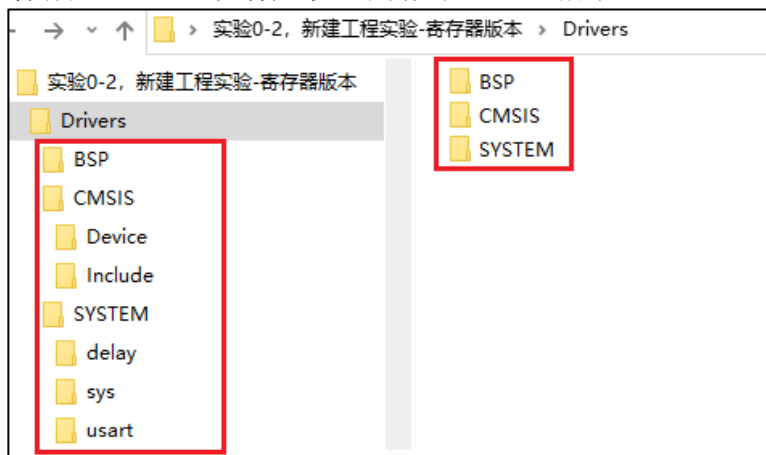


图 6.1.1.3 工程根目录文件夹

Middlewares 文件夹

该文件夹用于存放正点原子和其他第三方提供的中间层代码（组件/Lib 等），如：USMART、

MALLOC、TEXT、FATFS、USB、LWIP、各种 OS、各种 GUI 等等。本章我们暂时用不到该文件夹，不过可以先建好备用，后面的实验将会陆续添加各种文件。

Output 文件夹

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

Projects 文件夹

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

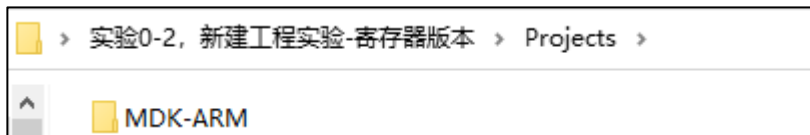


图 6.1.1.4 在 Projects 文件夹下新建 MDK-ARM 文件夹

User 文件夹

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

6.1.2 新建一个工程框架

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

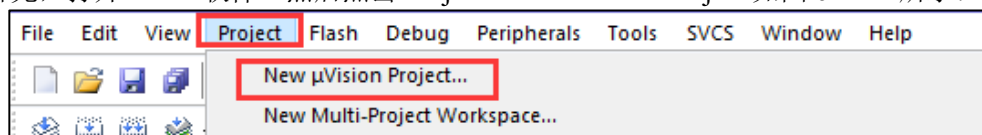


图 6.1.2.1 新建 MDK 工程

然后弹出工程命名和保存的操作窗口，我们将工程文件保存路径设置在上一节新建的工程文件夹内，具体路径为：桌面→实验 0-2，新建工程实验-寄存器版本→Projects→MDK-ARM，工程名字我们取：atk_f103，最后点击保存即可。具体操作窗口如图 6.1.2.2 所示：

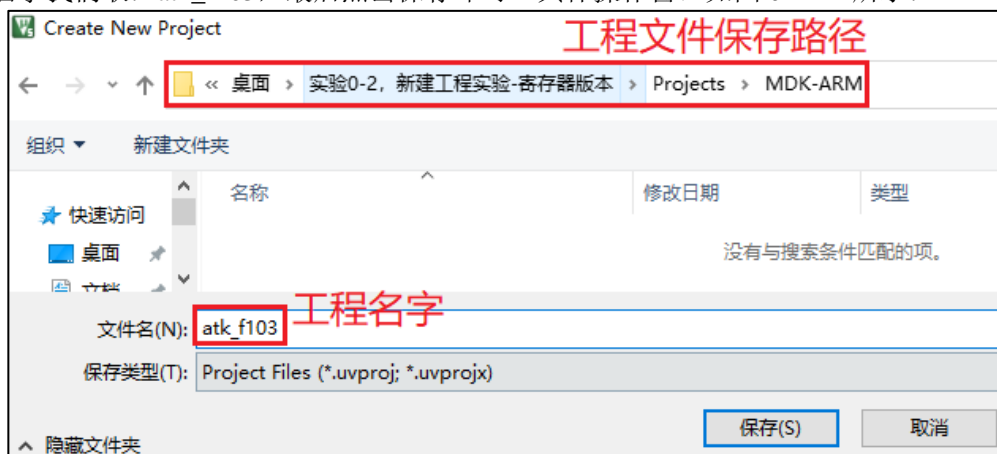


图 6.1.2.2 保存工程界面

之后，弹出器件选择对话框，如图 6.1.2.3 所示。因为战舰开发板所使用的 STM32 型号为 STM32F103ZET6，所以我们选择：STMicroelectronics→STM32F1 Series→STM32F103→STM32F103ZE（如果使用的是其他系列的芯片，选择相应的型号就可以了，特别注意：**一定要安装对应的器件 pack** 才会显示这些内容哦！！如果没得选择，请关闭 MDK，然后安装 A 盘：6，软件资料\1，软件\MDK5\Keil.STM32F1xx_DFP.2.3.0.pack 这个安装包后重试）。

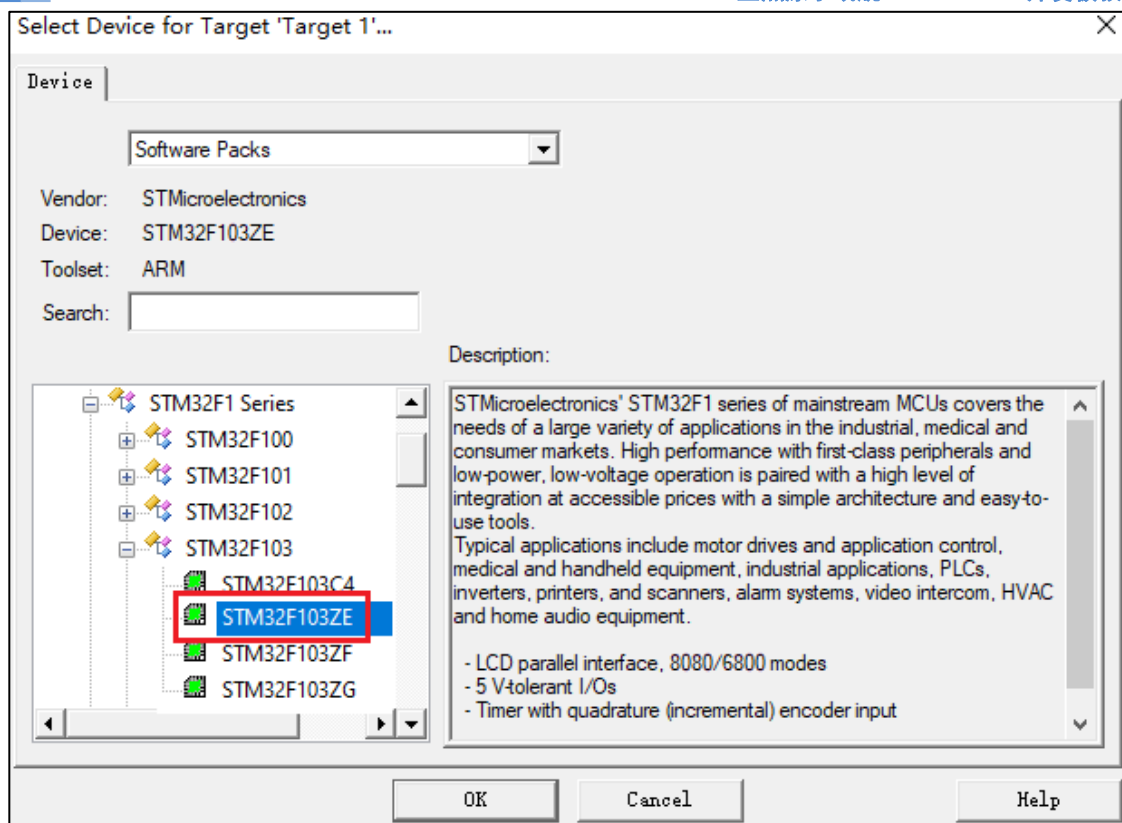


图 6.1.2.3 器件选择界面

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

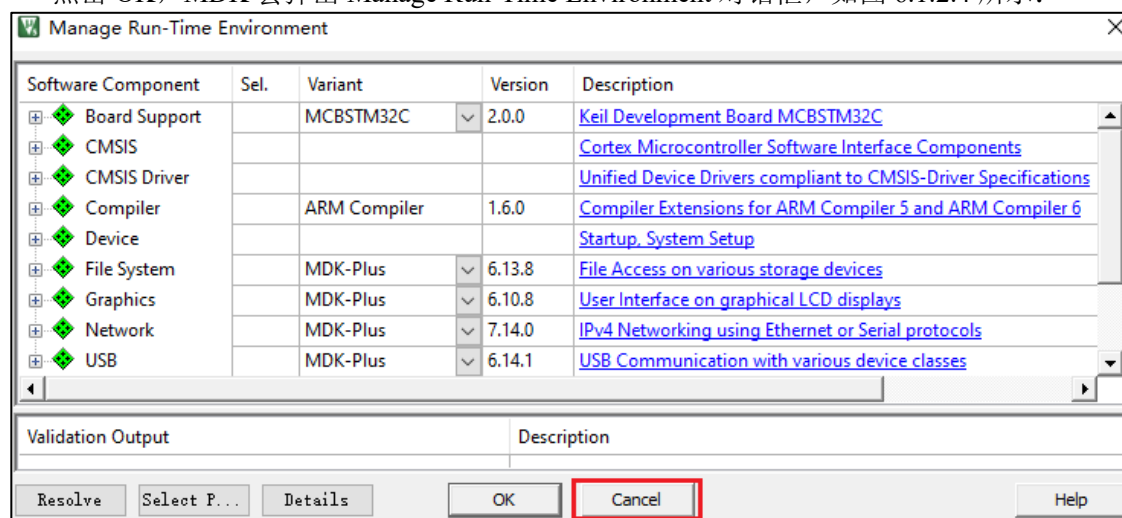


图 6.1.2.4 Manage Run-Time Environment 界面

这是 MDK5 新增的一个功能，在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，不过这里我们不做介绍。所以在图 6.1.2.4 所示界面，我们直接点击 Cancel，即可，得到如图 6.1.2.5 所示界面：

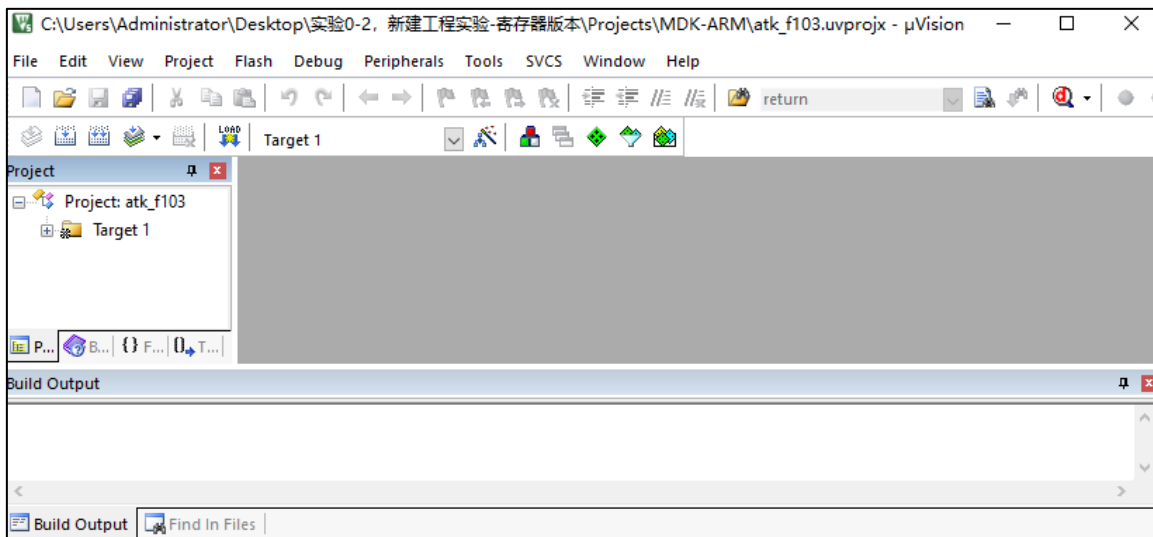


图 6.1.2.5 工程初步建立

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



图 6.1.2.6 MDK 新建工程时自动创建的文件夹

这三个文件夹的作用如表 6.1.2.1 所示：

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

表 6.1.2.1 三个文件夹及其作用

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

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

6.1.3 添加文件

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

1. 设置工程名和分组名

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

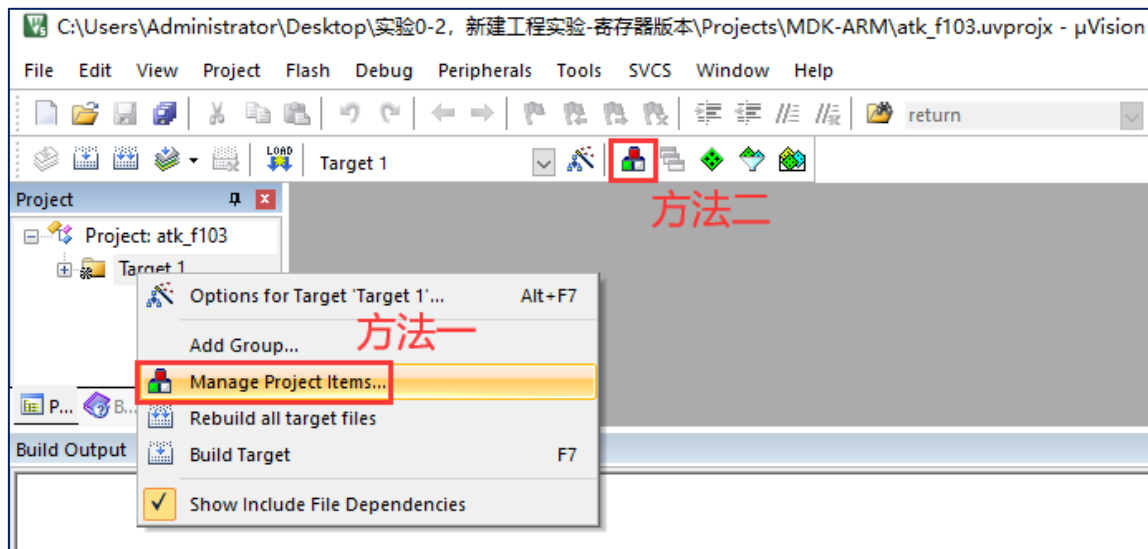


图 6.1.3.1 进入工程管理界面

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

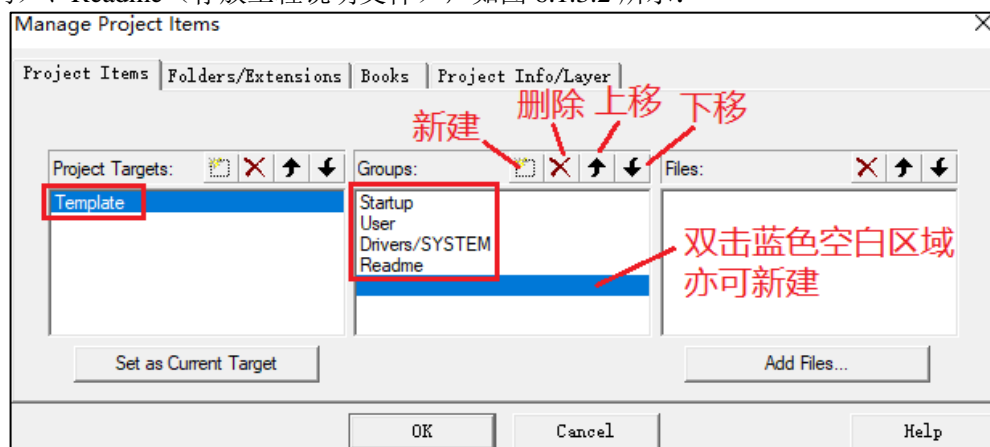


图 6.1.3.2 设置工程名和分组名

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

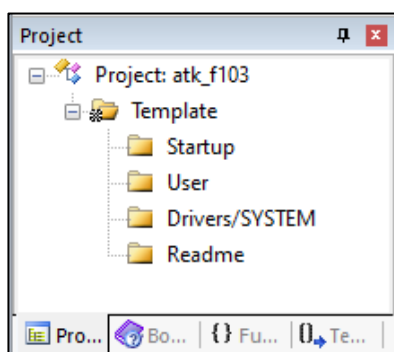


图 6.1.3.3 设置成功

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

注意：为了让工程结构清晰，我们会尽量让 MDK 的工程分组和我们前面新建的工程文件

夹对应起来，由于 MDK 分组不支持多级目录，因此我们将路径也带入分组命名里面，以便区分。如：User 分组对应 User 文件夹里面的源码，Drivers/SYSTEM 分组，对应 Drivers/SYSTEM 文件夹里面的源码，Drivers/BSP 分组对应 Drivers/BSP 文件夹里面的源码等。

2. 添加启动文件

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

该文件由 ST 官方提供，对于 STM32F103 来说有 4 个启动文件可选，如表 6.1.3.1 所示：

| 启动文件 | 对应 FLASH 容量 | 说明 |
|-----------------------|--------------------|-----------------------|
| startup_stm32f103x6.s | Flash≤32KB | 用于小容量 F103 系列芯片的启动文件 |
| startup_stm32f103xb.s | 64KB≤Flash≤128KB | 用于中容量 F103 系列芯片的启动文件 |
| startup_stm32f103xe.s | 256KB≤Flash≤512KB | 用于大容量 F103 系列芯片的启动文件 |
| startup_stm32f103xg.s | 768KB≤Flash≤1024KB | 用于超大容量 F103 系列芯片的启动文件 |

表 6.1.3.1 STM32F103 系列启动文件

启动文件存放在 STM32CubeF1 软件包的：Drivers→CMSIS→Device→ST→STM32F1xx→Source→Templates→arm 文件夹下。因为我们开发板使用的是 STM32F103ZET6，对应的启动文件为：startup_stm32f103xe.s，为了节省空间，在精简版 CMSIS 文件夹里面我们把其他启动文件都删了。而且，为了更好的匹配寄存器版本代码，我们对 startup_stm32f103xe.s 做了 2 处修改：

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

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

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

```
Reset_Handler    PROC
                    EXPORT Reset_Handler            [WEAK]
                    IMPORT __main
                    ;寄存器版本代码，因为没有用到 SystemInit 函数，所以注释掉以下代码为防止报错！
                    ;HAL 库版本代码，建议加上这里（提供 SystemInit 函数），以初始化 stm32 时钟等。
                    ;IMPORT SystemInit
                    ;LDR     R0, =SystemInit
                    ;BLX     R0

                    LDR     R0, =__main
                    BX      R0
                    ENDP
```

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

这了我们使用方法 1 添加（路径：实验 0-2，新建工程实验-寄存器版本\Drivers\CMSIS\Device\ST\STM32F1xx\Source\Templates\arm），如图 6.1.3.4 所示：

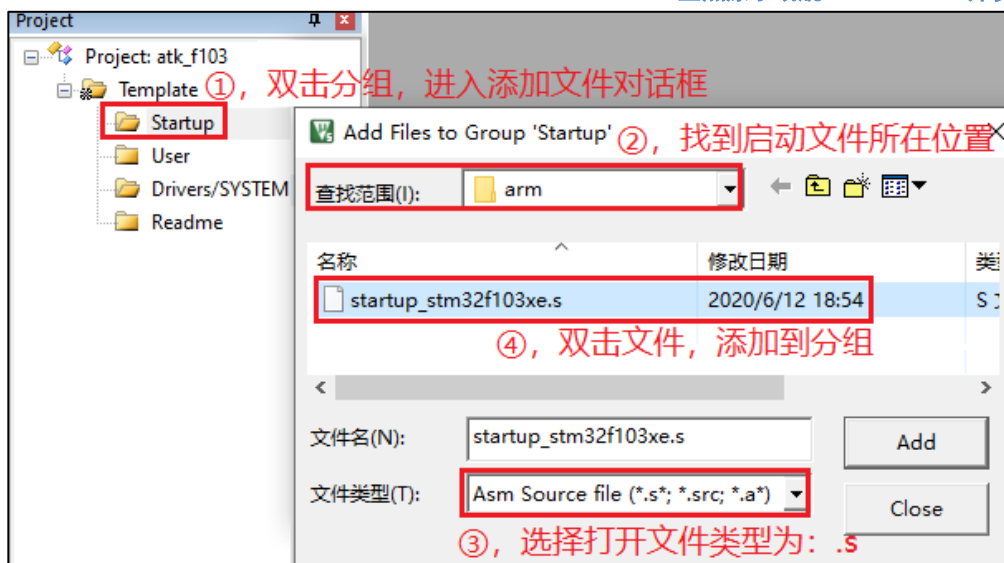


图 6.1.3.4 双击分组添加启动文件 (startup_stm32f103xe.s)

上图中, 我们也可以点击 Add 按钮进行文件添加。添加完后, 点击 Close, 完成启动文件添加, 得到工程分组如图 6.1.3.5 所示:

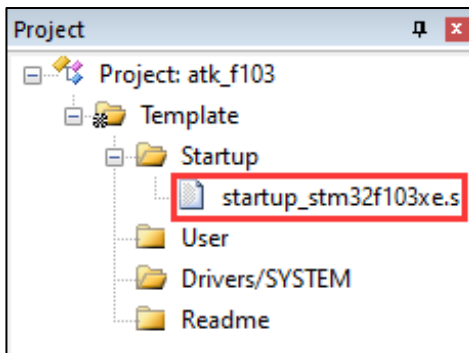



图 6.1.3.5 启动文件添加成功

3. 添加 SYSTEM 源码

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

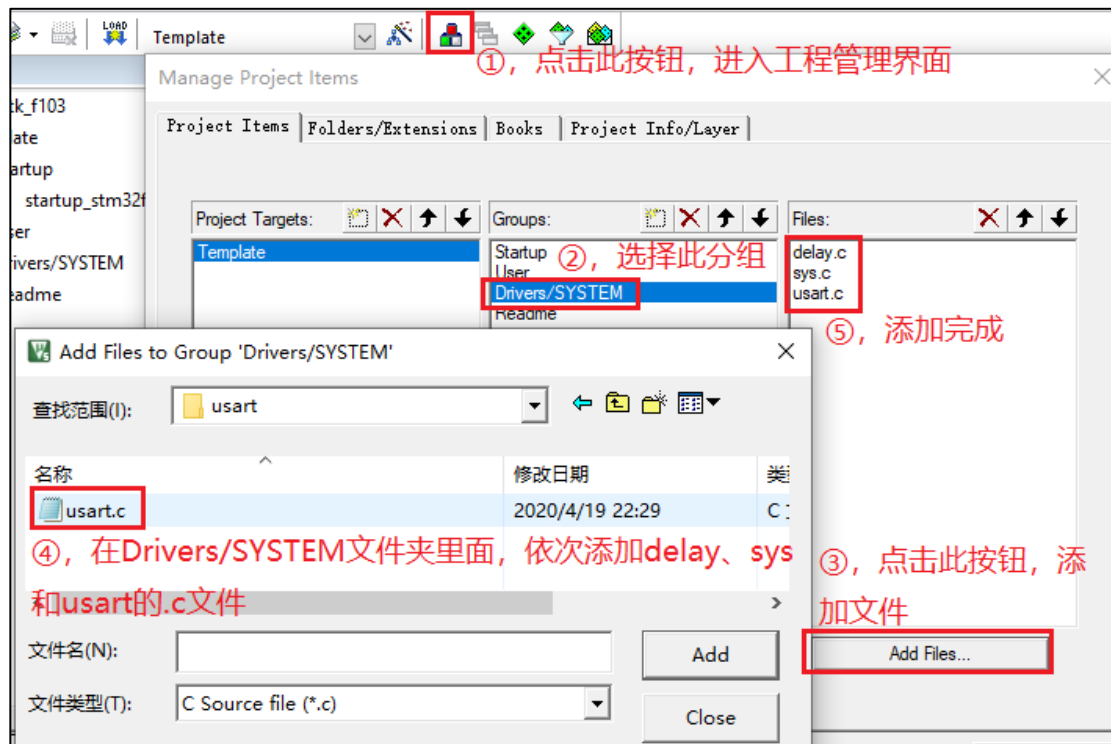


图 6.1.3.6 添加 SYSTEM 源码

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

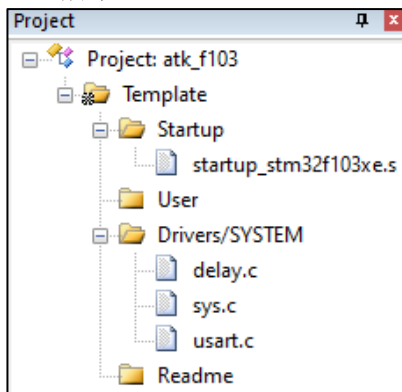


图 6.1.3.7 SYSTEM 源码添加完成

6.1.4 魔术棒设置

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

1. 设置 Target 选项卡

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

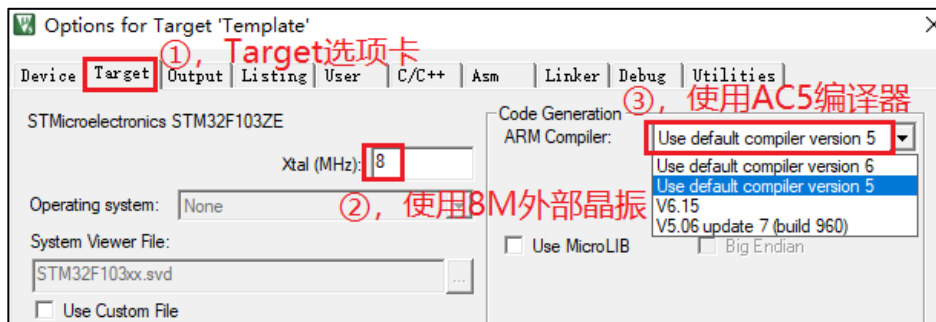


图 6.1.4.1 Target 选项卡设置

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

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

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

表 6.1.4.1 AC5&AC6 简单对比

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

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

表 6.1.4.2 AC5&AC6 设置差异

2. 设置 Output 选项卡

在魔术棒→Output 选项卡里面，进行如图 6.1.4.2 所示设置：

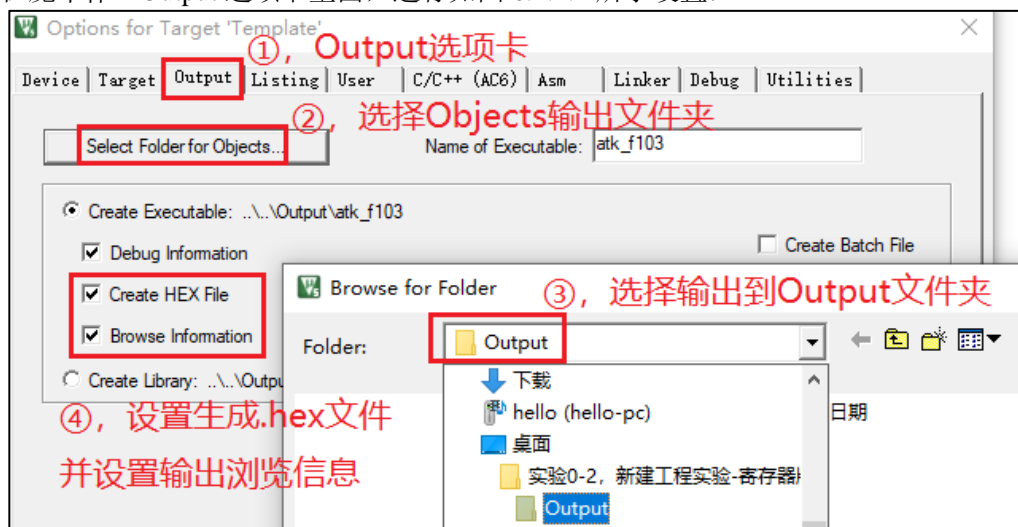


图 6.1.4.2 设置 Output 选项卡

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

3. 设置 Listing 选项卡

在魔术棒→Listing 选项卡里面，进行如图 6.1.4.3 所示设置：

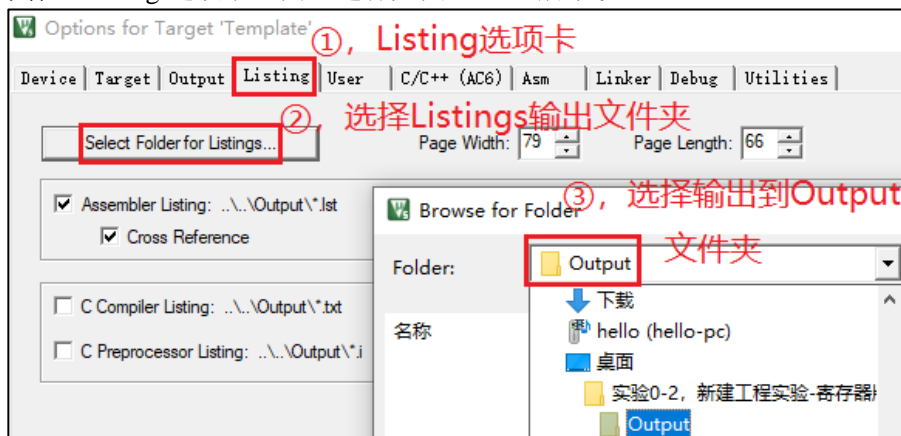


图 6.1.4.3 设置 Listing 选项卡

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

4. 设置 C/C++选项卡

在魔术棒→C/C++选项卡里面，进行如图 6.1.4.4 所示设置：

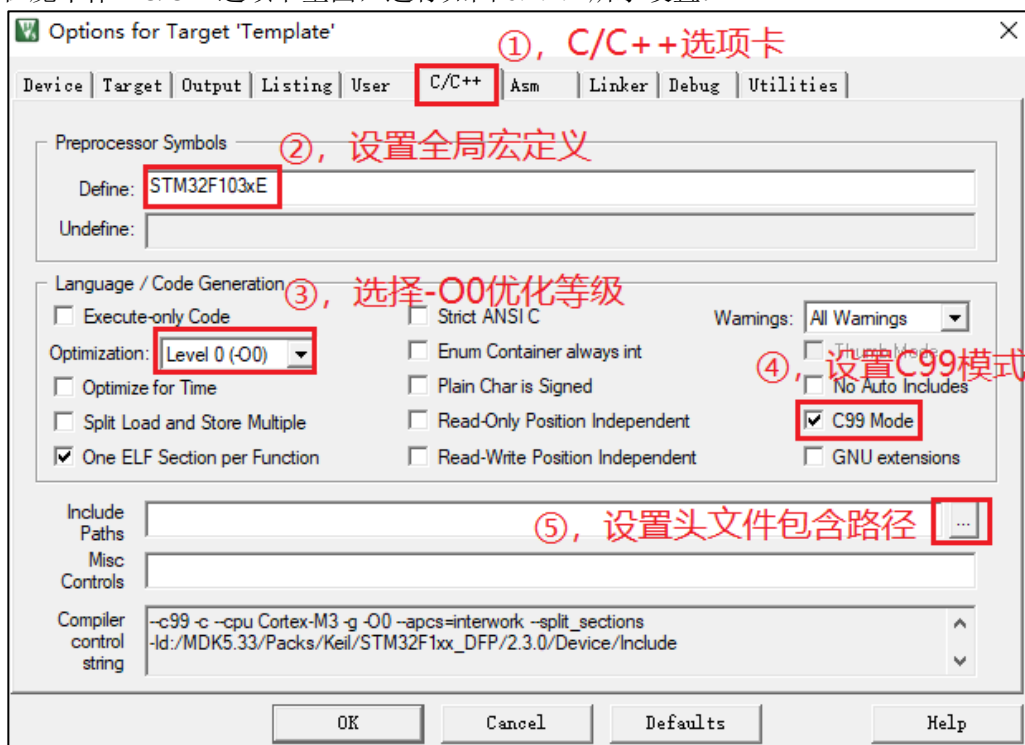


图 6.1.4.4 设置 C/C++选项卡

在②处设置了全局宏定义：STM32F103xE，用于定义所用 STM32 型号，在 stm32f1xx.h 里面会用到该宏定义。

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

在④处勾选 C99 模式，即使用 C99 C 语言标准。

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

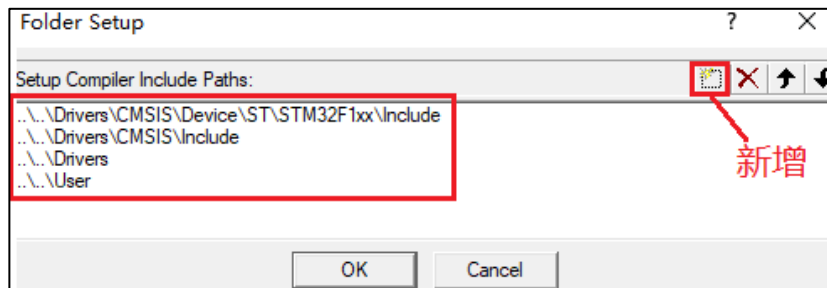


图 6.1.4.5 设置头文件包含路径

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

关于相对路径，这里大家记住 3 点：

- 1，默认路径就是指 MDK 工程所在的路径，即.uvprojx 文件所在路径（文件夹）
- 2，“./”表示当前目录（相对当前路径，也可以写做“.”）
- 3，“../”表示当前目录的上一层目录（也可以写做“..”）

举例来说，上图中：..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include，前面两个“..”，表示 Drivers 文件夹在当前 MDK 工程所在文件夹（MDK-ARM）的上 2 级目录下，具体解释如图 6.1.4.6 所示：

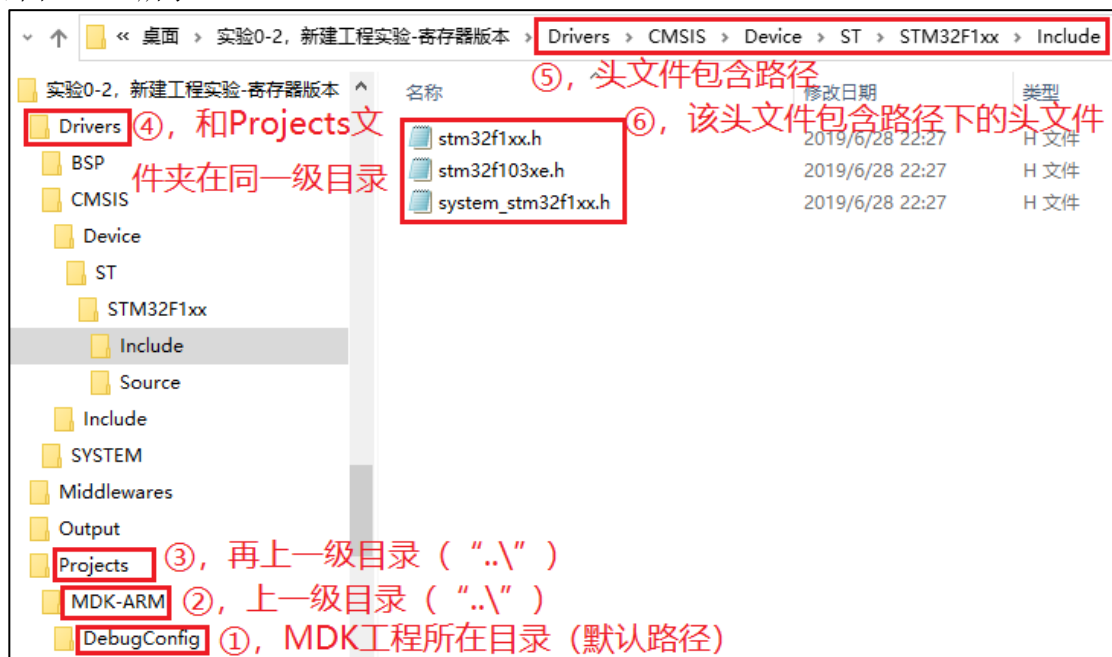


图 6.1.4.6 ..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include 的解释

上图表示根据头文件包含路径：..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include，编译器可以找到⑥处所包含的这些头文件，即代码里面可以直接 include 这些头文件使用。

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

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

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

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

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

5. 设置 Debug 选项卡

在魔术棒→Debug 选项卡里面，进行如图 6.1.4.7 所示设置：

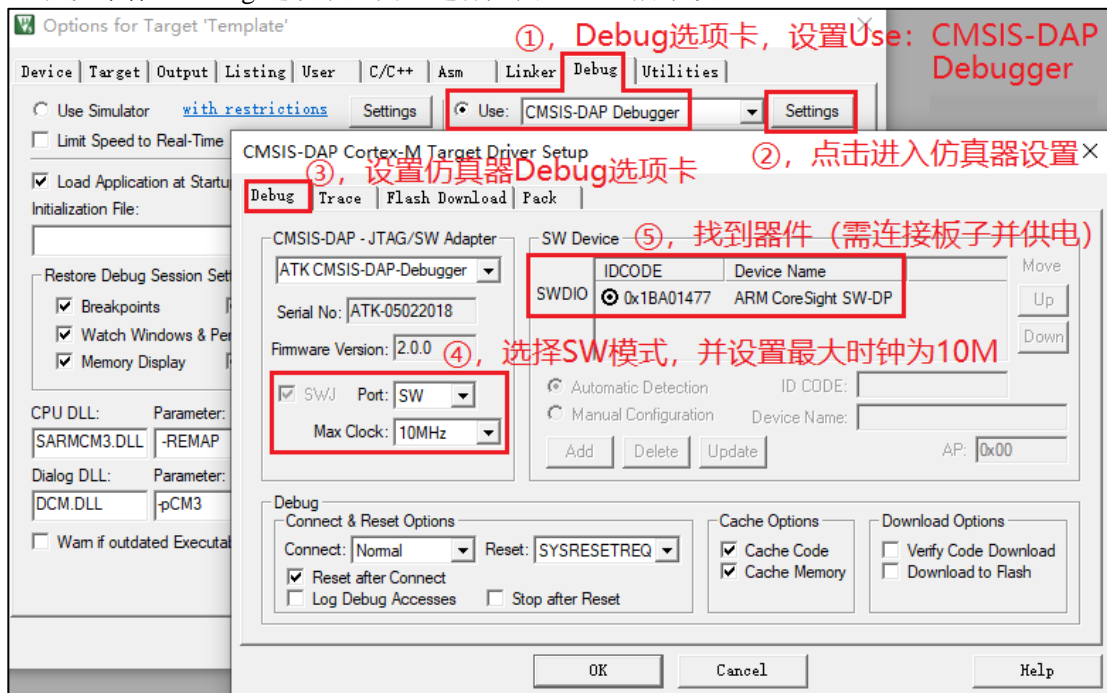


图 6.1.4.7 Debug 选项卡设置

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

6. 设置 Utilities 选项卡

在魔术棒→Debug 选项卡里面，进行如图 6.1.4.8 所示设置：

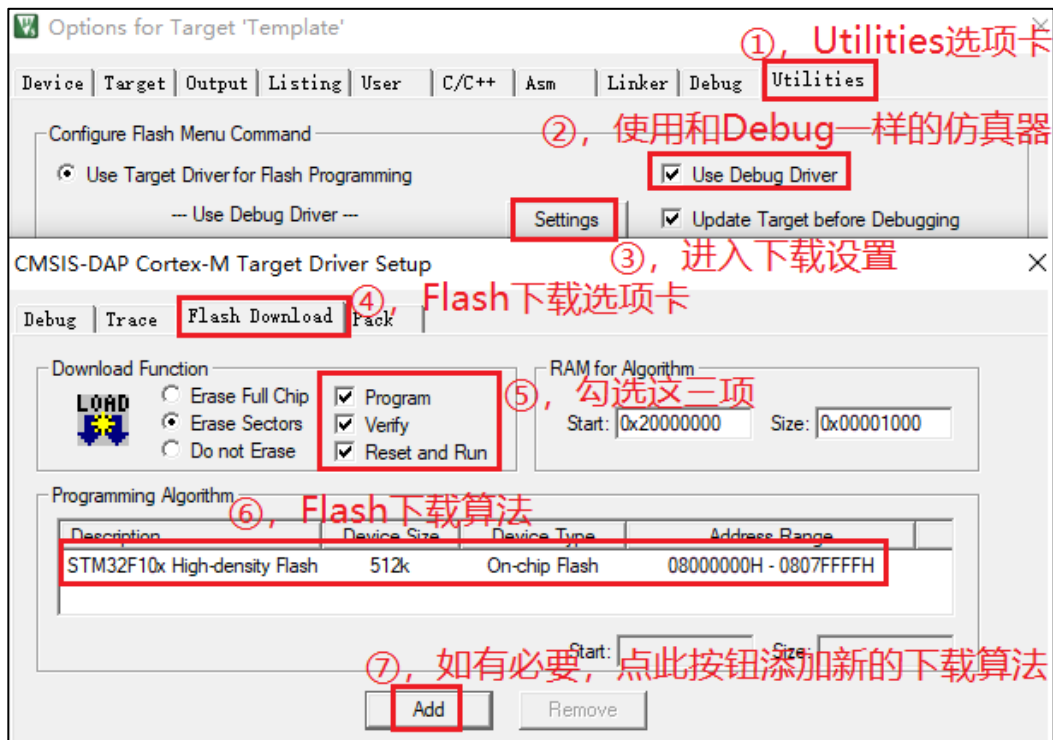



图 6.1.4.8 Utilities 选项卡设置

图中⑥处下载算法, 是 MDK 默认添加的, 针对 STM32F10x 大容量系列产品 (FLASH 容量在 256KB~512KB 之间)。一般我们用这个即可。如果⑥处没有下载算法, 则点击⑦处按钮, 执行添加一下下载算法即可 (名字和⑥处的算法名字一样)。

6.1.5 添加 main.c, 并编写代码

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

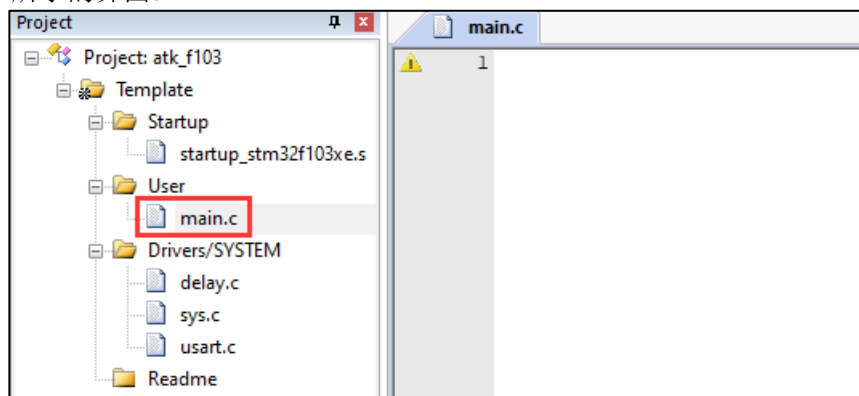


图 6.1.5.1 在 User 分组下加入 main.c 文件

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

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

int main(void)
{
    uint8_t t = 0;
    sys_stm32_clock_init(9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
}
```

```

    usart_init(72, 115200);      /* 串口初始化 */

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

```

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

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

输入完代码，如图 6.1.5.2 所示

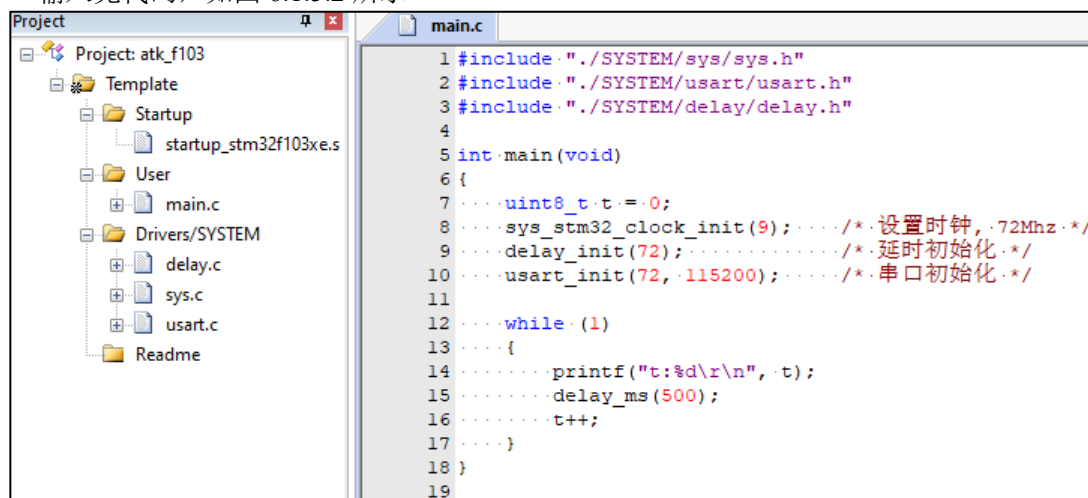


图 6.1.5.2 main.c 代码编写完成

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

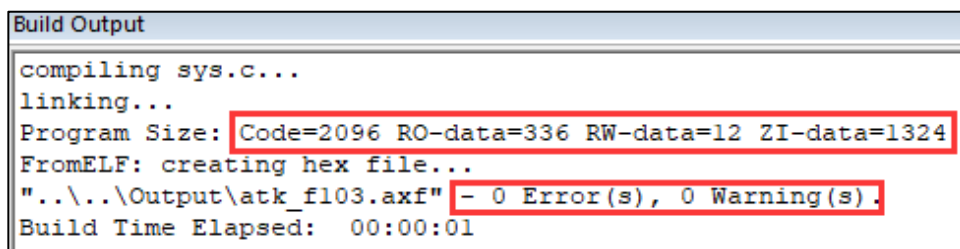


图 6.1.5.3 编译结果

编译结果提示：代码总大小 (Program Size) 为：FLASH 占用 2444 字节 (Code+RO+RW)，SRAM 占用 1336 字节 (RW+ZI)；并成功创建了 Hex 文件 (可执行文件，放在 Output 目录下)；编译 0 错误，0 警告。


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

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

至此，新建寄存器版本 MDK 工程完成。

6.2 下载验证

有两种方法可以给 STM32F103 芯片下载代码：1，使用串口下载；2，使用仿真器下载。这

两种下载方法，我们在本书的 4.2 和 4.3 节给大家有做过详细介绍。这里我们以仿真器下载为例，在 MDK 主界面，点击：（下载按钮，也可以按键盘快捷键：F8），就可以将代码下载到开发板，如图 6.2.1 所示：

```
Build Output
Load "..\..\..\Output\atk_f103.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 12:55:25
```

图 6.2.1 下载成功

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

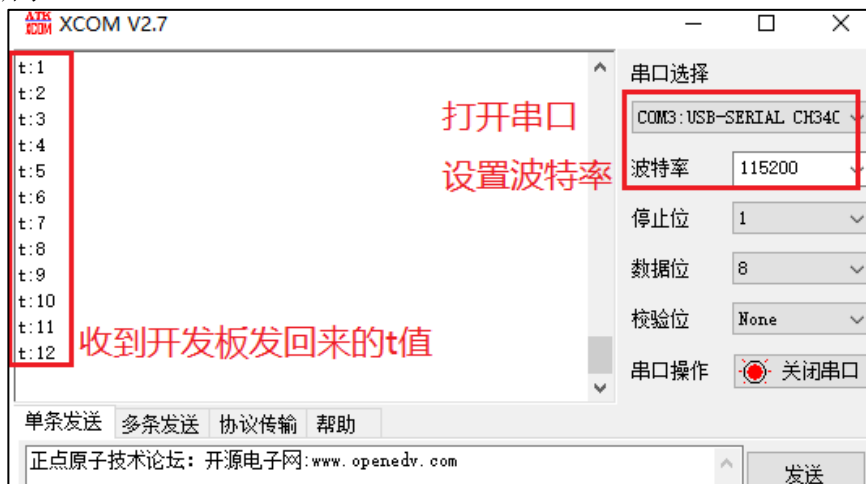


图 6.2.2 收到开发板发回来的数据

说明我们的程序运行正常，下载验证无误。

HAL，英文全称 Hardware Abstraction Layer，即硬件抽象层。HAL 库是 ST 公司提供的外设驱动代码的驱动库，用户只需要调用库的 API 函数，便可间接配置寄存器。我们要写程序控制 STM32 芯片，其实最终就是控制它的寄存器，使之工作在我们需要的模式下，HAL 库将大部分寄存器的操作封装成了函数，我们只需要学习和掌握 HAL 库函数的结构和用法，就能方便地驱动 STM32 工作，以节省开发时间。

7.5 HAL 库使用注意事项

The diagram illustrates the CMSIS architecture stack, showing the flow of data and control from the application code down to the hardware and debug logic.

Application code (top layer) connects to the **μVision® debugger** and the **CMSIS-Pack**.

The **CMSIS-Pack** is divided into four main components:

- CMSIS-RTOS** (light blue)
- CMSIS-DSP** (light blue)
- Standard middleware** (dark blue)
- Device-specific middleware** (dark blue)

Below these components are the **CMSIS-Drivers** (light blue) and the **Peripheral HAL** (green).

All these components interface with the **CMSIS-CORE** (light blue bar).

The **CMSIS-CORE** interfaces with the **Arm Cortex® processor**, **Communication peripherals**, and **Specialized peripherals**, which are part of the **Microcontroller device**.

For debugging, the **CMSIS-CORE** interfaces with the **CMSIS-SVD** and **CMSIS-DAP**, which connect to the **CoreSight™ debug logic**.

结合 STM32F1 的芯片来说, 其 CMSIS 应用程序的简单结构框图, 不包括实时操作系统和

中间设备等组件，其结构如图 7.1.1.2 所示。

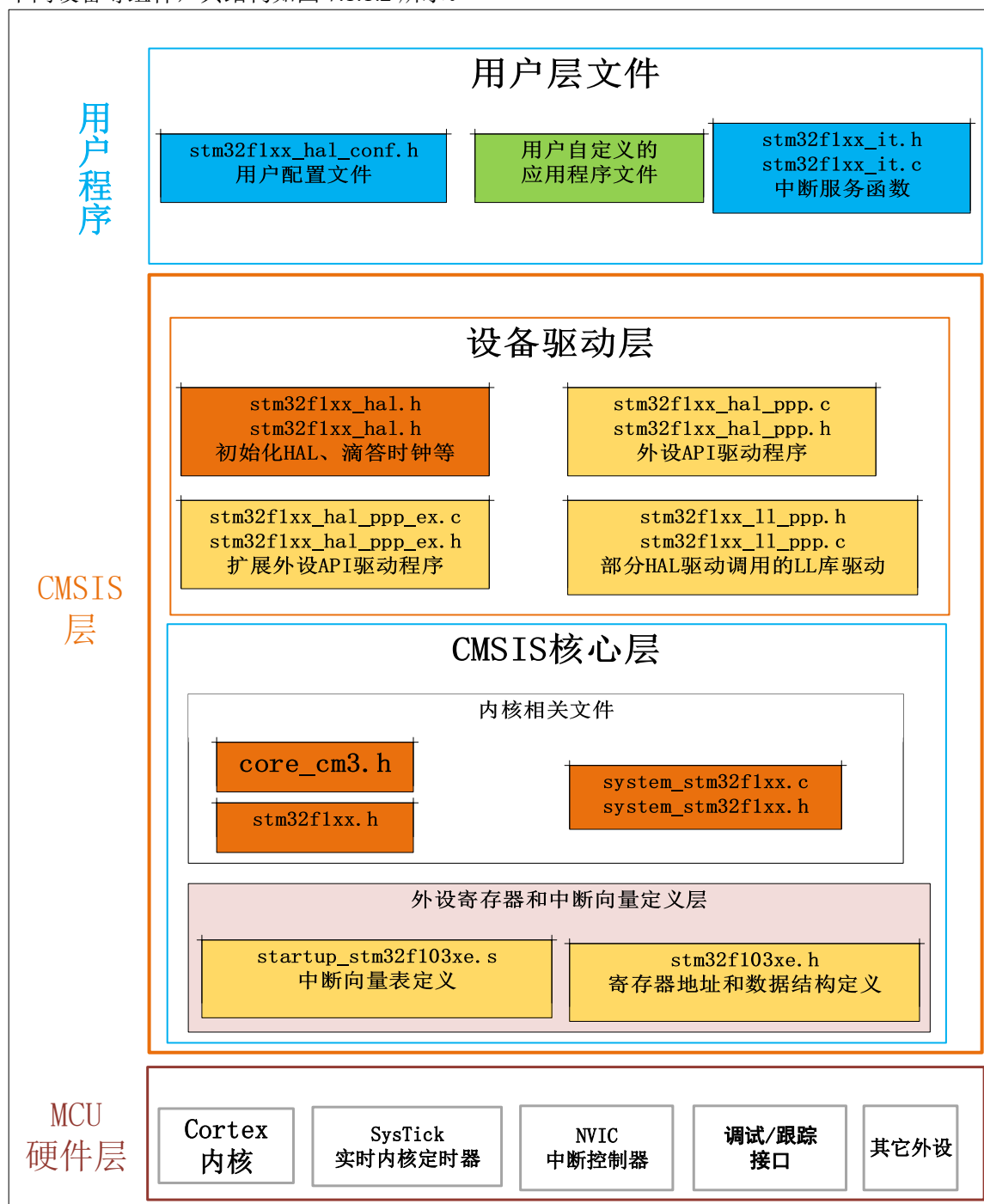


图 7.1.1.2 CMSIS 分级下的 stm32f1 的文件分布

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

7.1.2 HAL 库简介

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

1. 标准外设库 (Standard Peripheral Libraries)

标准外设库 (Standard Peripheral Library) 是对 STM32 芯片的一个完整的封装, 包括所有标准器件外设的器件驱动器, 是 ST 最早推出的针对 STM 系列主控的库函数。标准库的设计的初衷是减少用户的程序编写时间, 进而降低开发成本。几乎全部使用 C 语言实现并严格按照 “Strict ANSI-C”、MISRA-C 2004 等多个 C 语言标准编写。但标准外设库仍然接近于寄存器操作, 主要就是将一些基本的寄存器操作封装成了 C 函数。开发者仍需要关注所使用的外设是在哪个总线之上, 具体寄存器的配置等底层信息。



图 7.1.2.1 ST 的标准库函数家族

ST 为各系列提供的标准外设库稍微有些区别。例如, STM32F1x 的库和 STM32F3x 的库在文件结构上就有些不同, 此外, 在内部的实现上也稍微有些区别, 这个在具体使用 (移植) 时, 需要注意一下! 但是, 不同系列之间的差别并不是很大, 而且在设计上是相同的。STM32 的标准外设库涵盖以下 3 个抽象级别:

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

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

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

2. HAL 库

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

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

HAL 库是从 ST 公司从自身芯片的整个生产生态出发, 为了方便维护而作的一次整合, 以改变标准外设库带来各系列芯片操作函数结构差异大、分化大、不利于跨系列移植的情况。相比标准外设库, STM32CubeHAL 库表现出更高的抽象整合水平, HAL 库的 API 集中关注各外设的公共函数功能, 这样便于定义一套通用的用户友好的 API 函数接口, 从而可以轻松实现从一个 STM32 产品移植到另一个不同的 STM32 系列产品。但由于封闭函数为了适应最大的兼容性, HAL 库的一些代码实际上的执行效率要远低于寄存器操作。但即便如此, HAL 库仍是 ST 未来主推的库。

3. LL 库:

LL 库 (Low Layer) 目前与 HAL 库捆绑发布, 它设计为比 HAL 库更接近于硬件底层的操作, 代码更轻量级, 代码执行效率更高的库函数组件, 可以完全独立于 HAL 库来使用, 但 LL 库不匹配复杂的外设, 如 USB 等。所以 LL 库并不是每个外设都有对应的完整驱动配置程序。

使用 LL 库需要对芯片的功能有一定的认知和了解,它可以:

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

对于 HAL 库和 LL 库的关系,如图 7.1.2.2 CubeF1 的软件框架所示,可以看出它们设计为彼此独立的分支,但又同属于 HAL 库体系。

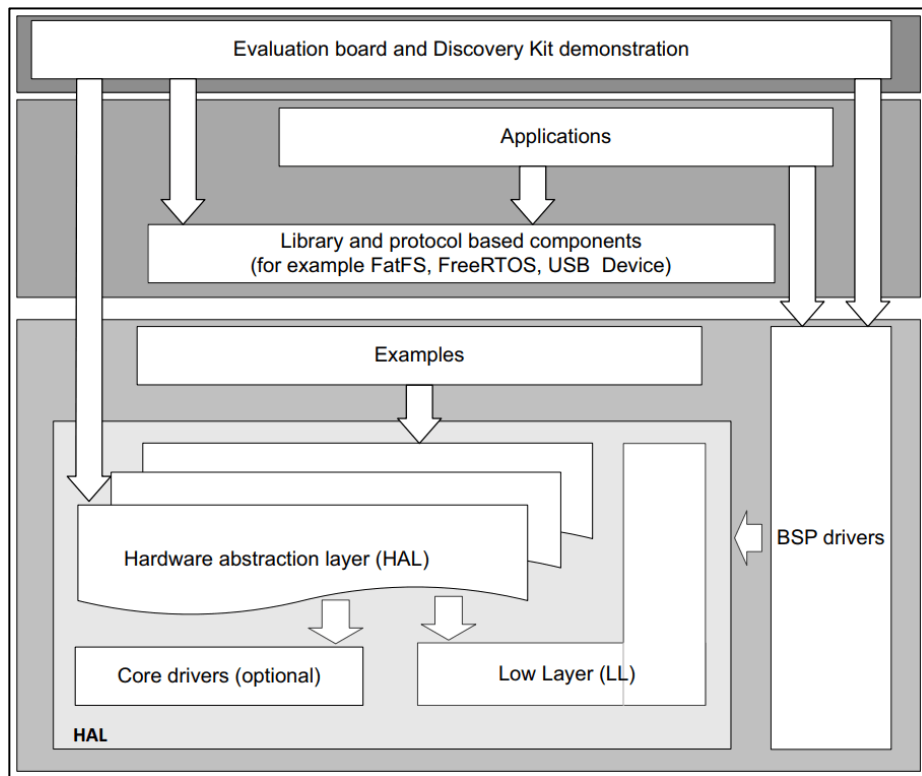


图 7.1.2.2 CubeF1 的软件框架

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

7.1.3 HAL 库能做什么

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

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

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

7.2 HAL 库驱动包

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

7.2.1 如何获取 HAL 库固件包

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

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

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

ST 提供了多种获取固件包的方法。本节只介绍从 ST 官方网站上直接获取固件库的方法。网页登陆：www.st.com，在打开的页面中依次选择：

“Tools & Software→Ecosystem →STM32Cube →新页面→Prodcut selector”，如图 7.2.1.1：

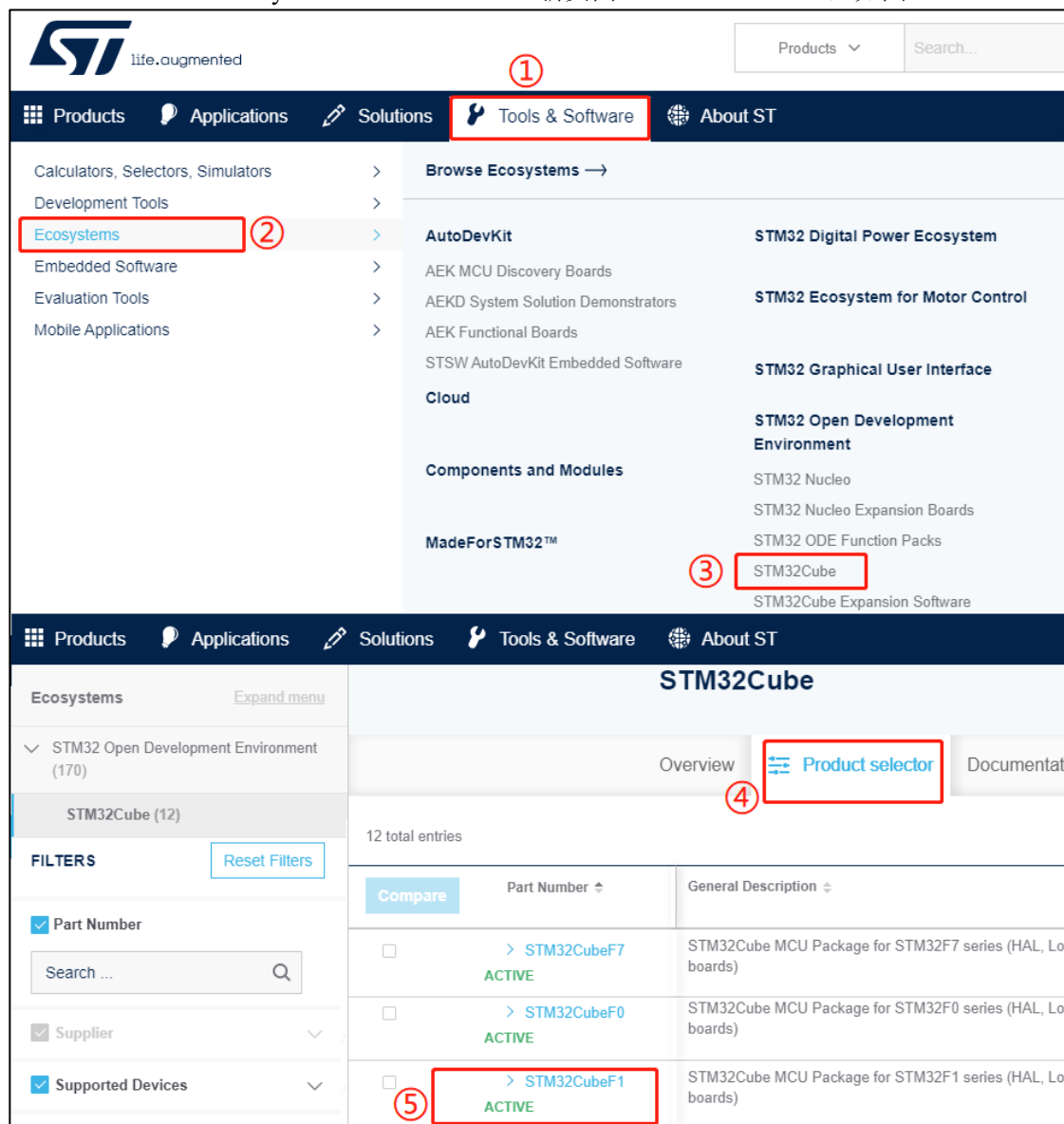


图 7.2.1.1 找到 STM32CubeF1 的固件下载位置

在展开的页面中选择我们需要和固件，这展开“STM32CubeF1”即可看到我们需要的 F1 的安装包，按下图操作，在新的窗口中拉到底部，选择适合自己的下载方式，注册帐号即可获取相应的驱动包。



图 7.2.1.2 下载 STM32CubeF1 固件包

STM32Cube 固件包，我们已经给大家下载好并且放到 **A 盘→8, STM32 参考资料→1, STM32F1xx 固件库**，当前固件包版本是:STM32Cube_FW_F1_V1.8.3(注意这个压缩包是 1.8.3, 但压缩包里的文件夹命名是 1.8.0, 后面讲到文件夹路径时我们按 ST 给的 1.8.0 来说明)。因为现在是 STM32F103 的学习，所以我们准备好的固件包是 F1 的。大家要根据自己学习的芯片，下载对应的固件包。如果需要最新的 HAL 库固件，大家可按照上述方法到官网重新获取即可。

7.2.2 STM32Cube 固件包分析

STM32Cube 固件包完全兼容 STM32CubeMX。对于图形配置工具 STM32CubeMX 入门使用，由于需要 STM32F1 基础才能入门使用，所以我们安排在后面第十章给大家讲解。本小节，我们主要讲解 STM32Cube 固件包的结构。

解压缩后的 STM32CubeF1 固件包的目录结构，如图 7.2.2.1 所示。

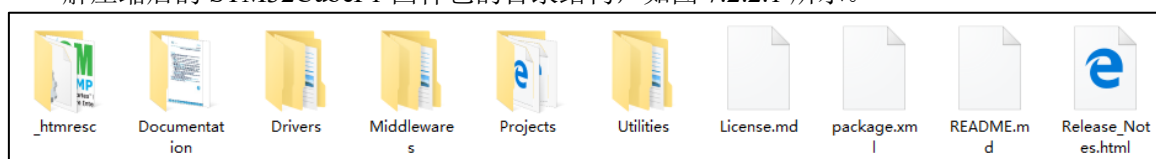


图 7.2.2.1 STM32CubeF1 固件包的目录结构

下面对 STM32CubeF1 固件包进行简要介绍。对于 Documentation 文件夹，里面是一个 STM32CubeF1 英文说明文档，是 ST 官方指导如何使用 HAL 库的指引。接下来我们通过几个表格依次来介绍一下 STM32CubeF1 中几个关键的文件夹。

(1) Drivers 文件夹

Drivers 文件夹包含 BSP, CMSIS 和 STM32F1xx_HAL_Driver 三个子文件夹。三个子文件夹具体说明请参考下表 7.2.2.1:

| | | |
|-------------|-----------|---|
| Drivers 文件夹 | BSP 文件夹 | 也叫板级支持包，用于适配 ST 官方对应的开发板的硬件驱动程序，每一种开发板对应一个文件夹。例如触摸屏，LCD，SRAM 以及 EEPROM 等板载硬件资源等驱动。这些文件针只匹配特定的开发板使用，不同开发板可能不能直接使用。 |
| | CMSIS 文件夹 | 顾名思义就是符合 CMSIS 标准的软件抽象层组件相关文件。文件夹内部文件比较多。主要包括 DSP 库(DSP_LIB 文件夹), Cortex-M 内核及其设备文件 (Include 文件夹)，微控制器专用头文件/启 |

| | | |
|--|--------------------------|--|
| | | 动代码/专用系统文件等(Device 文件夹)。在新建工程的时候,会使用到这个文件夹内部很多文件。 |
| | STM32F1xx_HAL_Driver 文件夹 | 这个文件夹非常重要,它包含了所有的 STM32F1xx 系列 HAL 库头文件和源文件。它的作用是屏蔽了复杂的硬件寄存器操作,统一了外设的接口函数。该文件夹包含 Src 和 Inc 两个子文件夹,其中 Src 子文件夹存放的是.c 源文件,Inc 子文件夹存放的是与之对应的.h 头文件。每个.c 源文件对应一个.h 头文件。源文件名称基本遵循 stm32H7xx_hal_ppp.c 定义格式,头文件名称基本遵循 stm32f1xx_hal_ppp.h 定义格式。比如 gpio 相关的 API 的声明和定义在文件 stm32H7xx_hal_gpio.h 和 stm32H7xx_hal_gpio.c 中。该文件夹的文件在我们新建工程章节都会使用到,我们后面会做详细介绍。 |

表 7.2.2.1 Drivers 文件夹介绍

(2) Middlewares 文件夹

该文件夹下面有 ST 和 Third_Party 2 个子文件夹。ST 文件夹下面存放的是 STM32 相关的一些文件,包括 STemWin 和 USB 库等。Third_Party 文件夹是第三方中间件,这些中间价都是非常成熟的开源解决方案。具体说明请见下表 7.2.2.2:

| | | | |
|-----------------|------------------|------------------------------|-----------------------------|
| Middlewares 文件夹 | ST 子文件夹 | STemWin 文件夹 | STemWin 工具包。Segger 提供。 |
| | | STM32_USB_Device_Library 文件夹 | USB 从机设备支持包。 |
| | | STM32_USB_Host_Library 文件夹 | USB 主机设备支持包。 |
| | Third_Party 子文件夹 | FatFs 文件夹 | FAT 文件系统支持包。采用的 FATFS 文件系统。 |
| | | FreeRTOS 文件夹 | FreeRTOS 实时系统支持包。 |
| | | LibJPEG 文件夹 | 基于 C 语言的 JPEG 图形解码支持包。 |
| | | LwIP 文件夹 | LwIP 网络通信协议支持包。 |

表 7.2.2.2 Middlewares 文件夹介绍

(3) Projects 文件夹

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

(4) Utilities 文件夹

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

(5) 其它几个文件

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

License.md: 用于声明软件版权信息的文件。

package.xml: 描述固件包版本信息的文件。

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

7.2.3 CMSIS 文件夹关键文件

上一节中我们对 STM32cube 固件包的主要目录结构做了分析。这一小节在上一小节的基础上,我们来分析一下 CMSIS 文件夹:由命名可知,该文件夹和 7.1.1 一小节中提到的 CMSIS 标

准是一致的，CMSIS 为软件包的内容制定了标准，包括文件目录的命名和内容构成，5.7.0 版本 CMSIS 规定软件包目录如表 7.2.3.1 所示：

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

表 7.2.3.1 CMSIS v5.7.0 的文件夹规范

知道了 CMSIS 规定的组件及其文件目录的大概内容后，我们再来看看 ST 提供的 CMSIS 文件夹，如上节提到的，它的位置是“STM32Cube_FW_F1_V1.8.0\Drivers\CMSIS”。打开文件夹内容如图 7.2.3.1 所示，可以发现它的目录结构完全按照 CMSIS 标准执行，仅仅是作了部分删减。

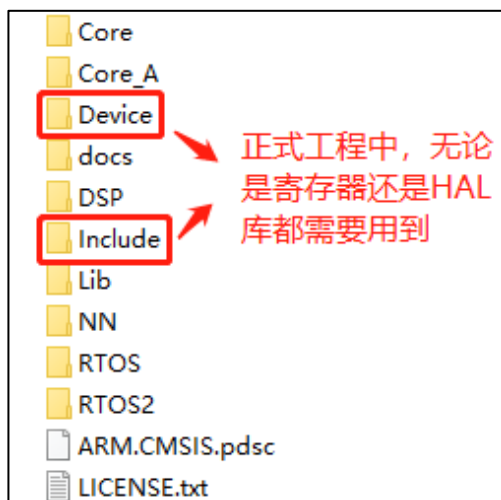


图 7.2.3.1 STM32CubeF1 固件包的 CMSIS 文件夹

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

(1) Device 文件夹

Device 文件夹关键文件介绍如下表 7.2.3.2 所示：

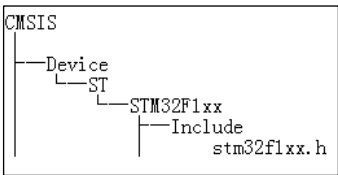
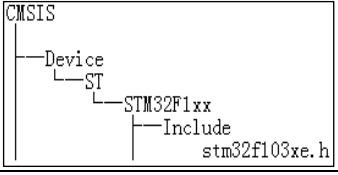
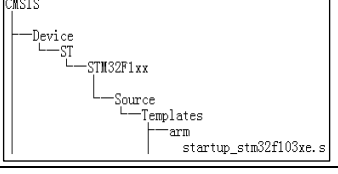
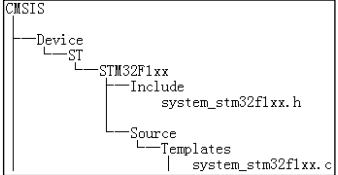
| 文件 | 描述 |
|---|--|
| stm32f1xx.h  | 包含了很多条件定义和常用的枚举变量类型，与宏定义配合，选择性包含某一特定的 STM32F1 系列芯片的头文件。这个文件使我们在使用 STM32F1 系列的不同型号芯片时，不需要每次都修改工程头文件，只需要修改宏定义并增加特定型号芯片的头文件即可快速选择使用不同类型的 F1 芯片。 |
| stm32f103xg.h  | STM32F1 大容量系列芯片通用的片上外设访问层头文件。Include 目录下有多个这样具体到型号的名字类似的头文件，我们具体使用哪个文件时需要根据实际使用的型号来确定。这个文件的主要作用是定义声明寄存器以及封装内存操作，以结构体和宏定义标识符的形式。 |
| startup_stm32f103xe.s  | STM32F103 系列芯片的启动文件，每个系列都有与之对应的启动文件。启动文件的作用主要是进行堆栈的初始化，中断向量表以及中断函数定义等。启动文件有一个很重要的作用就是系统复位后引导进入 main 函数。后面会细讲。 |
| system_stm32f1xx.h system_stm32f1xx.c  | 主要是声明和定义系统初始化函数 SystemInit 以及系统时钟更新函数 SystemCoreClockUpdate。SystemInit 函数的作用是进行时钟系统的一些初始化操作以及中断向量表偏移地址设置，但它并没有设置具体的时钟值，这是与标准库的最大区别，在使用标准库的时候，SystemInit 函数会帮我们配置好系统时钟配置相关的各个寄存器。在启动文件 startup_stm32f103xe.s 中会设置系统复位后，直接调用 SystemInit 函数进行系统初始化。SystemCoreClockUpdate 函数是在系统时钟配置进行修改后，调用这个函数来更新 SystemCoreClock 的值，变量 SystemCoreClock 是一个全局变量，开放这个变量可以方便我们在用户代码中直接使用这个变量来进行一些时钟运算。 |

表 7.2.3.2 Device 文件夹关键文件介绍

表 7.2.3.2 列出的文件都是正式工程中必须的文件。固件包的 CMSIS 文件包括了所有 STM32F1 芯片型号的文件，而我们只用到 STM32F103 系列，所以只针对我们用到的系列文件来讲。

(2) Include 文件夹

Include 文件夹存放了符合 CMSIS 标准的 Cortex-M 内核头文件。想要深入学习内核的朋友可以配合内核相关的手册去学习。对于 STM32F1 的工程，我们只要把我们需要的添加到工程即可，需要的头文件有：cmsis_armcc.h、cmsis_armclang.h、cmsis_compiler.h、cmsis_version.h、core_cm3.h 和 mpu_armv7.h。这几个头文件，对比起来，我们会比较多接触的是 core_cm3.h。

core_cm3.h 是内核底层的文件，由 ARM 公司提供，包含一些 AMR 内核指令，如软件复位，开关中断等功能。今后在需要的例程再去讲解其程序，现在要提到的是它包含了一个重要的头文件 stdint.h。

7.2.4 stdint.h 简介

stdint.h 是从 c99 中引进的一个标准 C 库的文件。在 2000 年 3 月，ANSI 采纳了 C99 标准。ANSI C 被几乎所有广泛使用的编译器（如：MDK、IAR）支持。多数 C 代码是在 ANSI C 基础上写的。任何仅使用标准 C 并且不和硬件相关的代码，在任意平台上用遵循 ANSI C 标准

的编译器下能编译成功。就是说这套标准不依赖硬件，独立于任何硬件，可以跨平台。

stdint.h 可以在 MDK 安装目录下找到，如 MDK5 安装在 C 盘时，可以在路径：

C:\Keil_v5\ARM\ARMCC\include 找到。stdint.h 的作用就是提供了类型定义，其部分类型定义代码如下：

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

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

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

7.3 HAL 库框架结构

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

7.3.1 HAL 库文件夹结构

HAL 库头文件和源文件在 STM32Cube 固件包的 STM32F1xx_HAL_Driver 文件夹中，打开该文件夹，如图 7.3.1.1 所示。

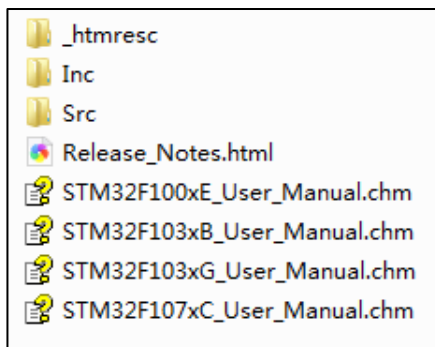


图 7.3.1.1 STM32F1xx_HAL_Driver 文件夹目录结构

STM32F1xx_HAL_Driver 文件夹下的 Src（Source 的简写）文件夹存放是所有外设的驱动程序源码，Inc（Include 的简写）文件夹存放的是对应源码的头文件。Release_Notes.html 是 HAL 库的版本更新信息。最后三个是库的用户手册，方便我们查阅对应库函数的使用。

打开 Src 和 Inc 文件夹，大家会发现基本都是 stm32f1xx_hal_ 和 stm32f1xx_ll_ 开头的.c 和.h 文件。刚学 HAL 库的朋友可能会说，stm32f1xx_hal_ 开头的是 HAL 库，我能理解。那 stm32f1xx_ll_ 开头的文件又是什么？这就告诉大家，stm32f1xx_ll_ 开头的文件是 LL 库。

7.3.2 HAL 库文件介绍

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

| 文件 | 描述 |
|--|---|
| stm32f1xx_hal.c stm32f1xx_hal.h | 初始化 HAL 库，（比如 HAL_Init, HAL_DeInit, HAL_Delay 等），主要实现 HAL 库的初始化、系统滴答，HAL 库延时函数、IO 重映射和 DBGMCU 功能。 |
| stm32f1xx_hal_conf.h HAL 库中本身没有这个文件，可以自行定义，也可以直 | HAL 的用户配置文件，stm32f1xx_hal.h 引用了这个文件，用来对 HAL 库进行裁剪。由于 Hal 库的很多配置都是通过预编译的条件宏来决定是否使用这一 HAL 库的功 |

| | |
|--|---|
| 接使用《Inc》文件夹下 stm32f1xx_hal_conf_template.h 的内容作为参考模版 | 能，这也是当前的主流库如 LWIP/FreeRTOS 等的做法， 无需修改库函数的源码，通过使能/不使能一些宏来实现 库函数的裁剪。 |
| stm32f1xx_hal_def.h | 通用 HAL 库资源定义，包含 HAL 的通用数据类型定义， 声明、枚举，结构体和宏定义。如 HAL 函数操作结果返 回值类型 HAL_StatusTypeDef 就是在这个文件中定义的。 |
| stm32f1xx_hal_cortex.h stm32f1xx_hal_cortex.c | 它是一些 Cortex 内核通用函数声明和定义，例如中断优先 级 NVIC 配置，MPU，系统软复位以及 SysTick 配置等， 与前面 core_cm3.h 的功能类似。 |
| stm32f1xx_hal_ppp.c stm32f1xx_hal_ppp.h | 外设驱动函数。对于所有的 STM32 该驱动名称都相同， ppp 代表一类外设，包含该外设的操作 API 函数。例如： 当 ppp 为 adc 时，这个函数就是 stm32f1xx_hal_adc.c/h， 可以分别在 Src/Inc 目录下找到。 |
| stm32f1xx_hal_ppp_ex.c stm32f1xx_hal_ppp_ex.h | 外设特殊功能的 API 文件，作为标准外设驱动的功能补充 和扩展。针对部分型号才有的特殊外设作功能扩展，或外 设的实现功能与标准方式完全不同的情况下作重新初始化 的备用接口。ppp 的含义同标准外设驱动 |
| stm32f1xx_ll_ppp.c stm32f1xx_ll_ppp.h | LL 库文件，在一些复杂外设中实现底层功能，在部分 stm32f1xx_hal_ppp.c 中被调用 |

表 7.3.2.1 HAL 库关键文件介绍

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

不止文件命名有一定规则，stm32f1xx_hal_ppp(c/h)中的函数和变量命名也严格按照命名规则，如表 7.3.2.2 所示的命名规则在大部分情况下都是正确的：

| 文件名 | stm32f1xx_hal_ppp(c/h) | stm32f1xx_hal_ppp_ex(c/h) |
|----------|--|--|
| 函数名 | HAL_PPP_Function HAL_PPP_FeatureFunction_MODE | HAL_PPPEX_Function HAL_PPPEX_FeatureFunction_MODE |
| 外设句柄 | PPP_HandleTypeDef | 无 |
| 初始化参数结构体 | PPP_InitTypeDef | PPP_InitTypeDef |
| 枚举类型 | HAL_PPP_StructnameTypeDef | 无 |

表 7.3.2.2 HAL 库函数、变量命名规则

对于 HAL 的 API 函数，常见的有以下几种：

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

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

➤ 配置和初始化用的结构体：

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

➤ 特殊处理的结构体

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

➤ 外设句柄结构体

HAL 驱动的重要参数，可以同时定义多个句柄结构以支持多外设多模式。HAL 驱动的操作结果也可以通过这个句柄获得。有些 HAL 驱动的头文件中还定义了一些跟这个句柄相关的一些外设操作。如用外设结构体句柄与 HAL 定义的一些宏操作配合，即可实现一些常用的寄存器位操作。比较常见的 HAL 库寄存器操作如表 7.3.2.3 所示：

| 宏定义结构 | 用途 |
|---|-----------|
| HAL_PPP_ENABLE_IT(HANDLE , INTERRUPT) | 使能外设中断 |
| HAL_PPP_DISABLE_IT(HANDLE , INTERRUPT) | 禁用外设中断 |
| HAL_PPP_GET_IT(HANDLE , INTERRUPT) | 获取外设某一中断源 |
| HAL_PPP_CLEAR_IT(HANDLE , INTERRUPT) | 清除外设中断 |
| HAL_PPP_GET_FLAG(HANDLE , FLAG) | 获取外设的状态标记 |
| HAL_PPP_CLEAR_FLAG(HANDLE , FLAG) | 清除外设的状态标记 |
| HAL_PPP_ENABLE(HANDLE) | 使能某一外设 |
| HAL_PPP_DISABLE(HANDLE) | 禁用某一外设 |
| HAL_PPP_XXXX(HANDLE , PARAM) | 针对外设的特殊操作 |
| HAL_PPP_GET_IT_SOURCE(HANDLE , INTERRUPT) | 检查外设的中断源 |

表 7.3.2.3 HAL 库驱动部分与外设句柄相关的宏

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

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

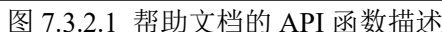
最后要分享的是 HAL 库的回调函数，这部分允许用户重定义，并在其中实现用户自定义的功能，也是我们使用 HAL 库的最常用的接口之一：

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

表 7.3.2.4 HAL 库驱动中常用的回调函数接口

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

ST 官方给我们提供了快速查找 API 函数的帮助文档。我们如果解压了 stm32cubef1 的固件包后，在路径 “STM32Cube_FW_F1_V1.8.3\Drivers\STM32F1xx_HAL_Driver” 下可以找到几个 chm 格式的文档，根据我们开发板主控芯片 STM32F103ZE 我们没有找到直接可用的，但可以查看型号接近的：STM32F103xG_User_Manual.chm(因为 G 系列比 E 系列引脚功能更多，只是查看 API 函数不响应使用的)。双击打开后，可以看到左边目录下有四个主题，我们来查看 Modules。以外设 GPIO 为例，讲一下怎么使用这个文档。点击 GPIO 外设的主题下的 IO operation functions / functions 看看里面的 API 函数接口描述，如图 7.3.2.1 所示。



下面举个例子，比如我们要让 PB4 输出高电平。先看函数功能，HAL_GPIO_WritePin 函数就是我们的 GPIO 口输出设置函数，如图 7.3.2.2 所示。



第三个形参是 GPIO PinState PinState, 看形参描述: 该参数可以是枚举里的两个数, 一个

是 GPIO_PIN_RESET：表示该位清零，另一个是 GPIO_PIN_SET：表示设置该位，即置 1，我们要输出 1，所以要置 1 该位，那么我们第三个形参就是 GPIO_PIN_SET。

最后看函数返回值：None，没有返回值。

所以最后得出我们要调用的函数是：

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

帮助文档的使用就讲到这。

本节只针对 HAL 库作了一个简单的介绍，想要了解更多知识，ST 提供了关于 HAL 库 LL 库的更详细的说明文档，我们已经把它放到光盘资料 A 盘《8，STM32 参考资料》中了，大家可以自行查阅《Description of STM32F1 HAL and low-layer drivers.pdf》获取所需知识。

7.4 如何使用 HAL 库

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

7.4.1 学会用 HAL 库组织开发工具链

需要按照芯片使用手册建议的步骤去配置芯片。HAL 库驱动提供了芯片的驱动接口，但我们需要强调一个概念是使用 HAL 库的开发是对芯片功能的开发，而不是开发这个库，也不是有这个库能就直接开发。如果我们对芯片的功能不作了解的话，仍然不知道按照怎样的步骤和寻找哪些可用的接口去实现想要实现的功能。ST 提供芯片使用手册《STM32F1xx 参考手册.pdf》告诉我们使用某一外设功能时如何具体地去操作每一个用到的寄存器的细节，后面我们的例程讲解过程也会结合这个手册来分析配置过程。

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

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

| 类别 | 文件名 | 描述 | 是否必须 |
|--------|------------------------|-----------------------------|------|
| 用户程序文件 | main.c | 存放 main 函数，不一定要在这个文件 | 否 |
| | main.h | 包含头文件、声明等作用，已删除 | 否 |
| | stm32f1xx_it.c | 用户中断服务函数存放文件，不一定放到这个文件，可删除 | 否 |
| | stm32f1xx_it.h | | 否 |
| | stm32f1xx_hal_conf.h | 用户配置文件 | 是 |
| | stm32f1xx_hal_msp.c | 回调函数存放文件，已删除 | 否 |
| 设备驱动层 | stm32f1xx_hal.c | HAL 库的初始化、系统滴答，HAL 库延时函数等功能 | 是 |
| | stm32f1xx_hal.h | | 是 |
| | stm32hxx_hal_def.h | 通用 HAL 库资源定义 | 是 |
| | stm32f1xx_hal_ppp.c | 外设的操作 API 函数文件 | 是 |
| | stm32f1xx_hal_ppp.h | | 是 |
| | stm32f1xx_hal_ppp_ex.c | 拓展外设特性的 API 函数文件 | 是 |
| | stm32f1xx_hal_ppp_ex.h | | 是 |
| | stm32f1xx_LL_ppp.c | LL 库文件，在一些复杂外设中实现底层功能 | 是 |
| | stm32f1xx_LL_ppp.h | | 是 |
| | stm32f1xx.h | STM32F1 系列的顶层头文件 | 是 |
| | stm32f103xe.h | STM32F103E 系列片上外设头文件 | 是 |
| | system_stm32f1xx.c | 主要存放系统初始化函数 SystemInit | 是 |
| | system_stm32f1xx.h | | 是 |

| | | | |
|-----------|---------------------|-------------------------|---|
| CMSIS 核心层 | startup_stm32f1xx.s | 启动文件，运行到 main 函数前的准备 | 是 |
| | core_cm3.h | 内核寄存器定义，如 SysTick、SCB 等 | 是 |
| | cmsis_armcc.h | 内核头文件，一般都不需要去了解 | 是 |
| | cmsis_armclang.h | | |
| | cmsis_compiler.h | | |
| | cmsis_version.h | | |
| | mpu_armv7.h | | |

表 7.4.1.1 基于 CMSIS 应用程序文件描述

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

7.4.2 HAL 库的用户配置文件

stm32f1xx_hal_conf.h 用于裁剪 HAL 库和定义一些变量，官方没有直接提供这个文件，但在 STM32Cube_FW_F1_V1.8.0\Drivers\STM32F1xx_HAL_Driver\Inc 这个路径下提供了一个模版文件《stm32f1xx_hal_conf_template.h》，我们可以直接复制这个文件重命名为 stm32f1xx_hal_conf.h，做一些简单的修改即可，也可以从官方的例程中直接复制过来。我们开发板使用的芯片是 STM32F103 的 E 系列，所以也可以从下面的路径获取这个配置文件：STM32Cube_FW_F1_V1.8.0\Projects\STM3210E_EVAL\Templates\Inc。

(1) 配置外部高速晶振的频率。HSE_VALUE 这个参数表示我们的外部高速晶振的频率。这个参数请务必根据我们板子外部焊接的晶振频率来修改，源码在 78 行开始，官方默认是 25M。正点原子 STM32F103 开发板外部高速晶振的频率是 8MHZ。我们没有在代码的其它地方定义过 HSE_VALUE 这个值，所以编译器最终会引用这里的值 8MHz 作为外部调整晶振的频率值。

注意事项：使用官方的开发板需要定义 USE_STM3210C_EVAL 这个宏，我们没有在代码的其它位置中或者编译器的预编译选项中定义过这个宏。

```
#if !defined (HSE_VALUE)
#if defined(USE_STM3210C_EVAL)
#define HSE_VALUE 25000000U /*!< Value of the External oscillator in Hz */
#else
#define HSE_VALUE 8000000U /*!< Value of the External oscillator in Hz */
#endif
#endif /* HSE_VALUE */
```

还可以把上面的代码直接精简为一行，效果是一样的：

```
#define HSE_VALUE 8000000U /*!< Value of the External oscillator in Hz */
```

(2) 还有一个参数就是外部低速晶振频率，这个用于 RTC 时钟，这个官方默认是 32.768KHZ，我们开发板的低速晶振也是这个频率，所以不用修改，源码在 111 行。

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

(3) 用户配置文件可以用来选择使能何种外设，源码配置在 31 行到 71 行，代码如下。

```
/* ##### Module Selection ##### */
/**
 * @brief This is the list of modules to be used in the HAL driver
 */
#define HAL_MODULE_ENABLED
#define HAL_ADC_MODULE_ENABLED
#define HAL_CEC_MODULE_ENABLED
#define HAL_COMP_MODULE_ENABLED
#define HAL_CORTEX_MODULE_ENABLED
...中间省略...
#define HAL_USART_MODULE_ENABLED
#define HAL_WWDG_MODULE_ENABLED
#define HAL_MMC_MODULE_ENABLED
```

比如不使用 GPIO 的功能，把源码在 49 行这个宏注释掉即可，具体如下。

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

结合同样在《stm32f1xx_hal_conf.h》中的 246 行的代码：

```
#ifndef HAL_GPIO_MODULE_ENABLED
```



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

这是一个条件编译符，与`#endif`配合使用。这里要表达的意思是，只要工程中定义了`HAL_GPIO_MODULE_ENABLED`这个宏，就会包含`stm32f1xx_hal_gpio.h`这个头文件到我们的工程，同时`stm32f1xx_hal_gpio.c`中的`#ifdef`到`#endif`之间的程序（116行到579行）就会参与编译，否则不编译。所以只要我们屏蔽了`stm32f1xx_hal_conf.h`文件49行的宏，GPIO的驱动代码就不被编译。也就起到选择使能何种外设的功能，其他外设同理。使用时定义，否则不定义。这样就可以在不修改源码的前提下方便地裁剪HAL库代码的体积了。

注意第一个宏定义：

```
#define HAL_MODULE_ENABLED
```

它决定了《`stm32f1xx_hal.c`》中的第47~587行的代码是否能使用，也是根据条件编译来实现的。其中包含`HAL_Init()`、`HAL_Delay()`、`HAL_GetTick()`这些其它驱动函数可能需要引用的函数，所以这个宏也是必须要定义的。

官方的示范例程，就是通过屏蔽外设的宏的方法来选择使能何种外设。表现上就是编译时间会变短，因为屏蔽了不使用的HAL库驱动，编译时间自然就短了。正点原子的例程选择另外一种方法，就是工程中只保留需要的`stm32f1xx_hal_ppp.c`，不需要的不添加到工程里，由于找不到源文件且没有引用这些文件，同样编译器不会去编译这些代码。

关于配置文件我们暂时只讲这些，具体其它需要修改的地方，我们在例程讲解中再去说明。

（4）大家看到`STM32F1xx_hal_conf.h`文件的127行。

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

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

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

（5）断言这个功能我们在使程中不使用。断言这个功能用来判断HAL函数的形参是否有效，并在参数错误时启用这个断言功能，告诉开发者代码错误的位置，断言功能由用户自己决定。这个功能的使能开关代码是一个宏，在源码的160行，默认是关闭的，代码如下。

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

通过宏`USE_FULL_ASSERT`来选择功能，在源码375行到389，代码如下。

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

也是通过条件编译符来选择对应的功能。当用户自己需要使用断言功能，怎么做呢？首先需要定义宏`USE_FULL_ASSERT`来使能断言功能，即把源码的160行的注释去掉即可。然后看到源码423行的`assert_failed()`这个函数。根据这个宏定义，我们还需要去定义这个函数的功能，可以按以下格式自定义这个函数，出错后使代码停留在这里：

```
#ifndef USE_FULL_ASSERT
```

```
/**
```



```

* @brief      当编译提示出错的时候此函数用来报告错误的文件和所在行
* @param      file: 指向源文件
*             line: 指向在文件中的行数
* @retval     无
*/
void assert_failed(uint8_t* file, uint32_t line)
{
    while (1)
    {
    }
}
#endif

```

可以看到这个函数里面没有实现如何功能，就是一个什么不做的死循环，具体功能请根据自己的需求去实现。`file` 是指向源文件的指针，`line` 是指向源文件的行数。`__FILE__` 是表示源文件名，`__LINE__` 是表示在源文件中的行数。比如我们可以实现打印出这个错误的两个信息等等，但前提是你已经学会了如何使芯片输出打印信息这些功能。

总的来说断言功能就是，在 HAL 库中，如果定义了 `USE_FULL_ASSERT` 这个宏，那么所有的 HAL 库函数将会检查函数的形参是否正确。如果错误将会调用 `assert_failed()` 这个函数，程序就会停留在这里，用户可以定位到出错的函数。这个功能实际上是在芯片上运行的时候的增加错误提示信息的功能，属于调试功能的一部分，实际我们的编译器就可以帮助定位到参数错误的问题并提示信息。在 F103 的工程中我们不使用这个功能。

7.4.3 stm32f1xx_hal.c 文件

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

1. HAL_Init()函数

源码在 142 行到 167 行，简化函数如下（下面的代码只针对 F1 的 HAL 固件 1.8.3 版本，其它版本可能有差异）：

```

HAL_StatusTypeDef HAL_Init(void)
{
    /* 配置 Flash 的预取控制器 */
    #if (PREFETCH_ENABLE != 0)
    #if defined(STM32F101x6) || defined(STM32F101xB) || defined(STM32F101xE) || de-
defined(STM32F101xG) || \ defined(STM32F102x6) || defined(STM32F102xB) || \
    defined(STM32F103x6) || defined(STM32F103xB) || defined(STM32F103xE) || de-
defined(STM32F103xG) || \ defined(STM32F105xC) || defined(STM32F107xC)

    /* Prefetch buffer is not available on value line devices */
    __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
    #endif
    #endif /* 使能 Flash 的预取控制器 */

    /* 配置中断优先级顺序 */
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

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

    /* 初始化其它底层硬件（如果必要） */
    HAL_MspInit();

    /* 返回函数状态 */
    return HAL_OK;
}

```

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

- 1) 使能 Flash 的预取缓冲器（根据闪存编程手册，打开预取指令可以提高对 I-Code 总线

的访问效率，且 AHB 时钟的预分频系数不为 1 时，必须打开预取缓冲器）

2) 设置 NVIC 优先级分组为 4。

3) 配置滴答定时器每 1ms 产生一个中断。

在这个阶段，系统时钟还没有配置好，因此系统还是默认使用内部高速时钟源 HSI 在跑程序。对于 F1 来说，HSI 的主频是 8MHZ。所以如果用户不配置系统时钟的话，那么系统将会使用 HSI 作为系统时钟源。

4) 调用 HAL_MspInit 函数初始化底层硬件，HAL_MspInit 函数在 STM32F1xx_hal.c 文件里面做了弱定义。关于弱定义这个概念，后面会有讲解，现在不理解没关系。正点原子的 HAL 库例程是没有使用到这个函数去初始化底层硬件，而是单独调用需要用到的硬件初始化函数。用户可以根据自己的需求选择是否重新定义该函数来初始化自己的硬件。

注意事项：为了方便和兼容性，正点原子的 HAL 库例程中的中断优先级分组设置为分组 2，即把源码的 HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4)改为如下代码：

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);
```

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

2. HAL_DeInit ()函数

源码在 175 行到 194 行，简化后函数如下：

```
HAL_StatusTypeDef HAL_DeInit(void)
{
    /* 重置所有外设 */
    __HAL_RCC_APB1_FORCE_RESET();
    __HAL_RCC_APB1_RELEASE_RESET();

    __HAL_RCC_APB2_FORCE_RESET();
    __HAL_RCC_APB2_RELEASE_RESET();

    /* 对底层硬件初始化 */
    HAL_MspDeInit();

    /* 返回函数状态 */
    return HAL_OK;
}
```

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

1) 复位了 APB1、APB2 的时钟。

2) 调用 HAL_MspDeInit 函数，对底层硬件初始化进行复位。HAL_MspDeInit 也在 STM32F1xx_hal.c 文件里面做了弱定义，并且与 HAL_MspInit 函数是一对存在。HAL_MspInit 函数负责对底层硬件初始化，HAL_MspDeInit 函数则是对底层硬件初始化进行复位。这两个函数都是需要用户根据自己的需求去实现功能，也可以不使用。

3. HAL_InitTick ()函数

源码在 234 行到 255 行，简化函数如下：

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

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

```

    }
    else
    {
        return HAL_ERROR;
    }

    /* 返回函数状态 */
    return HAL_OK;
}

```

该函数用于初始化滴答定时器的时钟基准，主要功能如下：

1) 配置滴答定时器 1ms 产生一次中断。

2) 配置滴答定时器的中断优先级。

3) 该函数是 `__weak` 修饰的函数，如果在关联工程中的其它地方没有定义 `__weak` 后面的函数，这里是 `HAL_InitTick()`，则使用此处的定义，否则就使用其它地方定义好的函数功能。我们可以通过重定义这个函数来选择其它的时钟源（如定时器）作为 HAL 库函数的时基或者通过重定义不开启 SysTick 的功能和中断等。

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

4. 滴答定时器相关的函数

源码在 293 行到 406 行，相关函数如下：

```

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

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

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

/* 设置滴答定时器中断频率 */
HAL_StatusTypeDef HAL_SetTickFreq(HAL_TickFreqTypeDef Freq)
{
    HAL_StatusTypeDef status = HAL_OK;
    HAL_TickFreqTypeDef prevTickFreq;

    assert_param(IS_TICKFREQ(Freq));

    if (uwTickFreq != Freq)
    {
        /* 备份滴答定时器中断频率 */
        prevTickFreq = uwTickFreq;

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

        /* 应用新的滴答定时器中断频率 */
    }
}

```

```

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

return status;
}

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

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

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

    while ((HAL_GetTick() - tickstart) < wait)
    {
    }
}

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

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

```

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

5. HAL 库版本相关的函数

源码在 422 行到 484 行，相关函数声明在 stm32f1xx_hal.h 中，详看如下：

```

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

```

这些函数了解一下就好了，用得不多。

6. 调试功能相关函数

源码在 490 行到 587 行，函数声明如下：

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

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

7.4.4 HAL 库中断处理

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

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

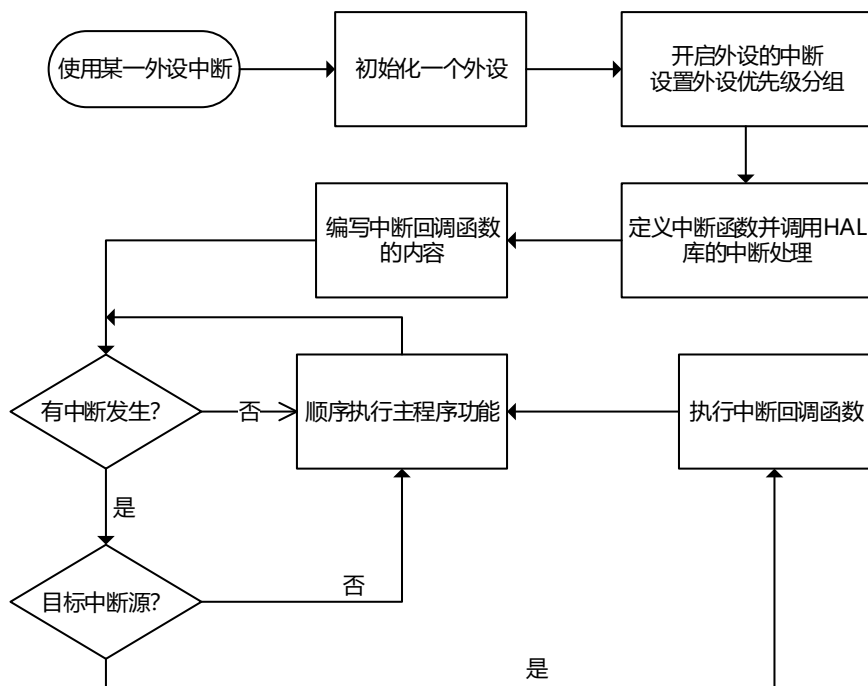


图 7.4.4.1 HAL 驱动中断处理流程

结合以前的 HAL 库文件介绍章节，以上的流程大概就是：设置外设的控制句柄结构体 PPP_HandleType 和初始化 PPP_InitType 结构体的参数，然后调用 HAL 库对应这个驱动的初始化 HAL_PPP_Init()，由于这个 API 中有针对外设初始化细节的接口 Hal_PPP_Mspinit()，我们需要重新实现这个函数并完成外设时钟、IO 等细节差异的设置，完成各细节处理后，使用 HAL_NVIC_SetPriority()、HAL_NVIC_EnableIRQ()来使能我们的外设中断；

定义中断处理函数 PPP_IRQHandler，并在中断函数中调用 HAL_ppp_function_IRQHandler()来判断和处理中断标记；HAL 库中断处理完成后，根据对应中的调用我们需要自定义的中断回调接口 HAL_PPP_ProcessCpltCallback();如串口接收函数 HAL_UART_RxCpltCallback()，我们在这个函数中实现我们对串口接收数据想做的处理；

中断响应处理完成后，stm32 芯片继续顺序执行我们定义的主程序功能，按照以上处理的标准流程完成了一次中断响应。

7.4.5 正点原子对 HAL 库用法的个性化修改

前面介绍了 ST 官方建议的 HAL 库的使用方法，这部分我们结合我们实际例程的编写，列出例程中对 HAL 库使用上的一些与官方推荐用法的差异，读者们请结合自己的使用习惯，辩证地去看待我们这种修改方式：

- 1，每个例程的 BSP 下都有一个 HAL 库，并且代码中对文件的引用尽量使用相对路径，保证每个复制完整的例程，在其他路径下也能编译通过；这种做法增加了 HAL 库全部例程的体积；

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

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

- 4，我们编写的初始化函数或者芯片操作时序上的延时，我们使用到了 `delay_ms()`、`delay_us()` 等函数进行初始化，使用的是 `Systick` 作的精准延时，而 HAL 库默认也使用 `Systick` 作延时处理，为解决这种冲突和兼容我们大部分的驱动代码，我们在例程中使用 `delay.c` 中的延时函数取代 `Hal_Delay()`；取消原来 HAL 库的 `Systick` 延时设置。

7.5 HAL 库使用注意事项

本小节根据经验跟大家讲述一些关于 HAL 库使用的注意事项，供读者参考。

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

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

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

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

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

- 6，我们建议如无致命 BUG 出现，尽量使用开发学习时已经测试稳定的 HAL 库来继续进行开发，不必要频繁更新 HAL 库，因为更新 HAL 库后可能会导致原来的程序不能正常运行，有些函数操作的结构方式变化了等等，这里只是笔者的建议，大家自己的实际情况权衡。

第八章 新建 HAL 版本 MDK 工程

在前面的章节我们介绍了 STM32F1xx 官方固件包的一些知识，本章我们将重点讲解新建 HAL 库版本的 MDK 工程的详细步骤。我们把本章新建好的工程放在光盘里，路径：**4，程序源码\2，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-3，新建工程实验-HAL 库版本**，大家在学习新建工程过程中遇到一些问题，可以直接打开这个工程，然后对比学习。

本章将分为如下两个小节：

8.1 新建 HAL 库版本 MDK 工程

8.2 下载验证

8.1 新建 HAL 库版本 MDK 工程

本节我们将教大家如何新建一个 STM32F103 的 MDK5 工程。为了方便大家参考，我们将本节最终新建好的工程模板存放在“A 盘：4、程序源码\2，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-3，新建工程实验-HAL 库版本”，如遇新建工程问题，请打开该实验对比。

在新建工程之前，首先我们要做如下准备：

- 1) STM32Cube 官方固件包：我们使用的固件包版本是 STM32Cube_FW_F1_V1.8.3，固件包路径：**A 盘→8，STM32 参考资料→1，STM32CubeF1 固件包**。
- 2) 开发环境搭建：参考本书第三章相关内容。

8.1.1 新建工程文件夹

新建工程文件夹分为 2 个步骤：1，新建工程文件夹；2，拷贝工程相关文件。

1. 新建工程文件夹

首先我们要在电脑某个路径下新建一个文件作为工程的根目录文件，后续的工程文件都将在这个文件夹里建立，我们把这个文件夹重命名为“实验 0-3，新建工程实验-HAL 库版本”。如图 8.1.1.1 所示。



图 8.1.1.1 新建工程根目录文件夹

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

| 名称 | 作用 |
|-------------|--|
| Drivers | 存放与硬件相关的驱动层文件 |
| Middlewares | 存放正点原子提供的中间层组件文件和第三方中间层文件 |
| Output | 存放工程编译输出文件 |
| Projects | 存放 MDK 工程文件 |
| User | 存放 HAL 库用户配置文件、main.c、stm32f1xx_it.c、以及我们自己编写的其它应用程序 |

表 8.1.1.1 工程根文件目录下新建的文件夹

新建完成以后，最后得到我们的工程根目录文件夹如图 8.1.1.2 所示。

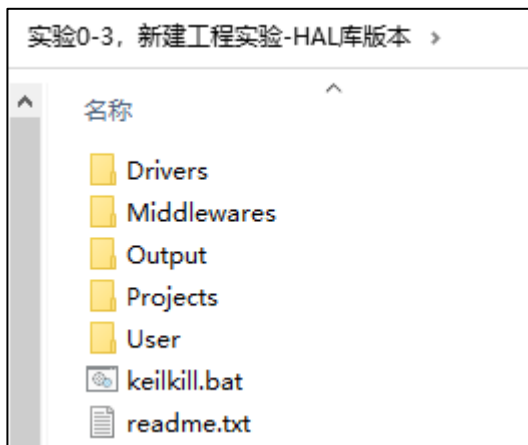


图 8.1.1.2 工程根文件目录

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

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

2. 拷贝工程相关文件

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

Drivers 文件夹

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

| 文件夹名称 | 作用 |
|----------------------|---|
| BSP | 存放开发板板级支持包驱动代码，如各种外设驱动 |
| CMSIS | 存放 CMSIS 底层代码，如启动文件（.s 文件）、stm32f1xx.h 等 |
| STM32F1xx_HAL_Driver | 存放 ST 提供的 F1xx HAL 库驱动代码 |
| SYSTEM | 存放正点原子系统级核心驱动代码，如 sys.c、delay.c 和 usart.c 等 |

表 8.1.1.2 Drivers 包含文件夹

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

SYSTEM 文件夹，用于存放正点原子提供的系统级核心驱动代码，sys.c/h,usart.c/h,delay.c/h，方便大家快速搭建自己的工程。该文件同样可以从“**A 盘→4，程序源码→2，标准例程-HAL 库版本**”文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

CMSIS 文件夹，用于存放 CMSIS 底层代码（ARM 和 ST 提供），如：启动文件（.s 文件）、stm32f1xx.h 等各种头文件。该文件夹我们可以直接从 STM32CubeF1 固件包（路径：A 盘→8，STM32 参考资料→1，STM32CubeF1 固件包）里面拷贝，CMSIS 的文件夹路径在“STM32CubeF1 固件包→Drivers”。由于这个文件夹原来设计是用于匹配全部 F1 系列的芯片的，导致非常大，部分文件对我们的例程来说不会使用到，而且浪费磁盘的存储空间，所以我们会对这个文件夹进行精简：打开目录“CMSIS\Device\ST\STM32F1xx”，其中的 Include 文件夹里都是芯片的头文件我们只留下如图 8.1.1.3 这三个头文件，其他删除。

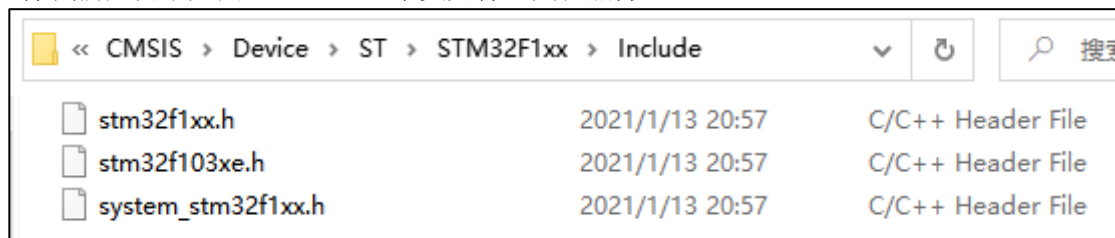


图 8.1.1.3 Include 文件夹留下的文件

Source 文件夹下的 Templates 文件夹留下如图 8.1.1.4 的内容。

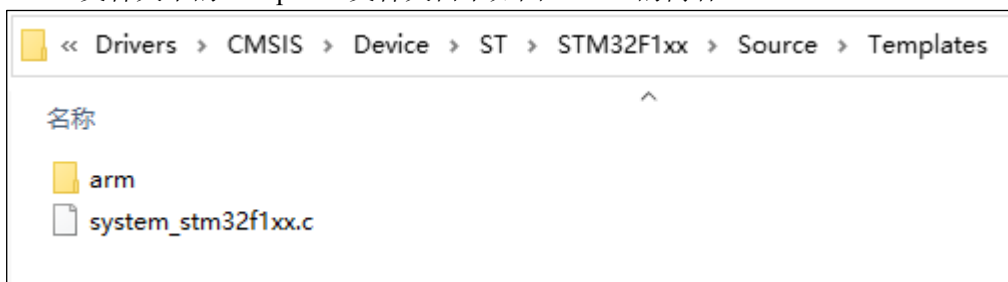


图 8.1.1.4 Templates 文件夹留下的文件

arm 文件夹存放的是启动文件，我们只需要 startup_stm32f103xe.s，其他全部删除。如图 8.1.1.5 所示。

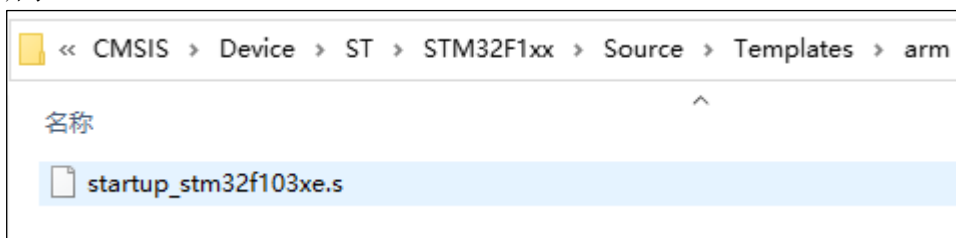


图 8.1.1.5 删除多余启动文件后的目录文件情况

最后就是 CMSIS 文件夹下的 Include 文件夹，里面都是内核的头文件，我们只需要如图 8.1.1.6 的内容。

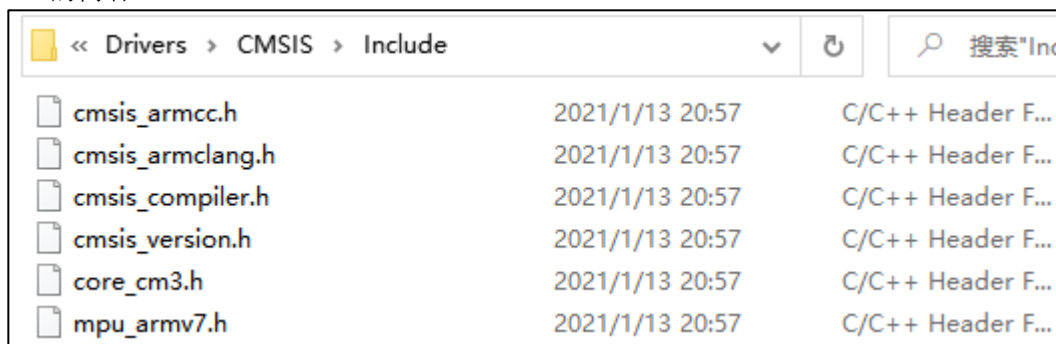


图 8.1.1.6 CMSIS/Include 文件夹留下的文件

到这里 CMSIS 文件夹就处理完成了。精简后的 CMSIS 文件夹大家也可以在“A 盘→4，程序源码→2，标准例程-HAL 版本”文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

STM32F1xx_HAL_Driver 文件夹，用于存放 ST 提供的 F1xx HAL 库驱动代码。该文件夹我们可以直接从 STM32CubeF1 固件包里面拷贝。直接拷贝“STM32CubeF1 固件包→Drivers”路径下的“STM32F1xx_HAL_Driver”文件夹到我们工程的 Drivers 下。该文件夹目录最终如图 8.1.1.7 的内容。

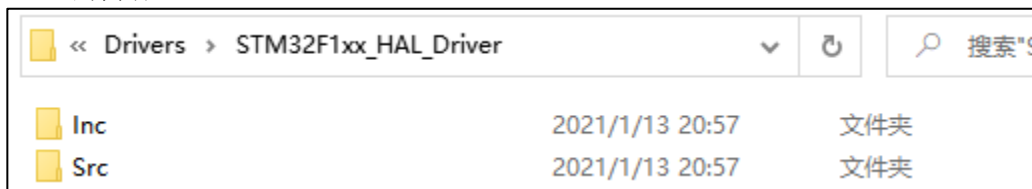


图 8.1.1.7 STM32F1xx HAL_Driver 文件夹留下的文件

到这里，我们就完成了把官方固件包中必要的驱动文件添加到我们工程文件中。最终我们新建的 Drivers 文件夹目录下的文件构成如图 8.1.1.8 所示。



图 8.1.1.8 工程根目录下的 Drivers 文件夹

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

Middlewares 文件夹

Middlewares 文件夹用于存放正点原子提供的中间层组件文件和第三方中间层文件，比如：USMART、MALLOC、TEXT、FATFS、USB、LWIP、各种 OS、各种 GUI 等等。我们新建工程实验暂时用不到，留空就行，后面的实验将会陆续添加各种文件。

Output 文件夹

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

Projects 文件夹

Projects 文件夹用于存放 MDK 工程，因为我们的工程是基于 ARM，所以我们在 Projects 文件夹里面新建一个命名为 MDK-ARM 的文件夹，用于存放 MDK 的工程文件，如图 8.1.1.9 所示。



图 8.1.1.9 在 Projects 文件夹下新建 MDK-ARM 文件夹

User 文件夹

User 文件夹用于存放 HAL 库用户配置文件、main.c、中断处理文件，以及配置文件 stm32f1xx_hal_conf.h。

我们首先从官方固件包里面直接拷贝官方的模板工程下的 HAL 库用户配置文件和中断处理文件到我们的 User 文件夹里。官方的模板工程路径：[STM32Cube_FW_F1_V1.8.0\Projects\STM3210E_EVAL\Templates](#)，打开 Template_Project 文件夹，如图 8.1.1.10 所示。

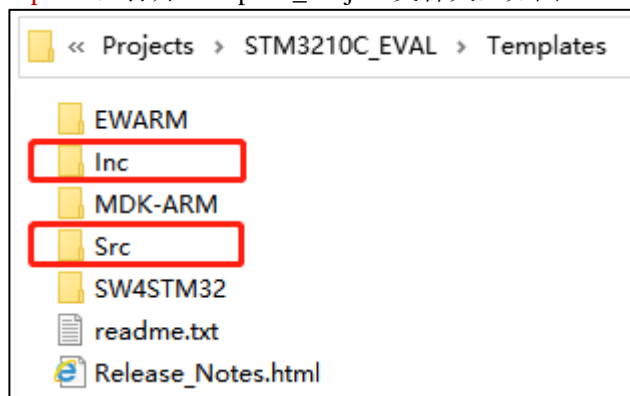


图 8.1.1.10 官方模板工程根目录

我们需要的文件就在 Inc 和 Src 文件夹里面，在这两个文件夹里面找到：stm32f1xx_it.c、stm32f1xx_it.h、stm32f1xx_hal_conf.h 这三个文件，并且拷贝到我们的 User 文件夹下。

main.c 文件我们也是放在 User 文件夹下，后面在 MDK 里面教大家新建.c 文件并保存。User 文件夹最终构成图如图 8.1.1.11 所示。

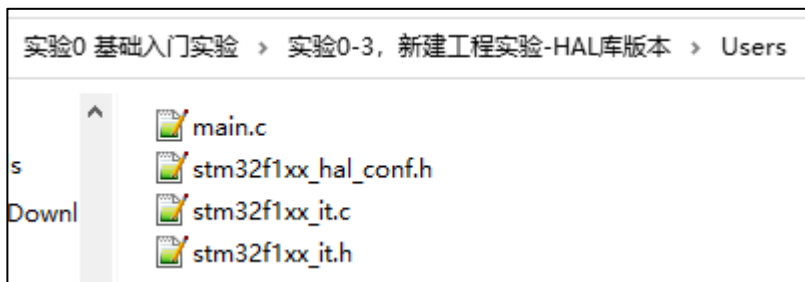


图 8.1.1.11 User 文件夹最终构成图

8.1.2 新建一个工程框架

打开 Keil uVision5，点击菜单 Project -> New Uvision Project，如图 8.1.2.1 所示。

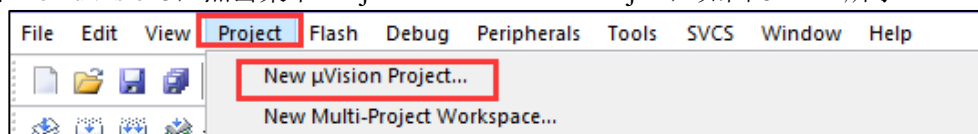


图 8.1.2.1 新建工程

然后弹出工程命名和保存的操作窗口，工程文件保存路径为：实验 0-2，新建工程实验-HAL 库版本\Projects\MDK-ARM，工程名字我们取：atk_f103，最后点击保存即可。具体操作窗口如图 8.1.2.2 所示。



图 8.1.2.2 工程命名和保存

接下来会弹出一个选择 Device 的界面，就是选择我们的芯片设备型号，大家根据自己使用的芯片型号依次选择即可。STM32F103 战舰开发板的芯片型号是：STM32F103ZET6，所以我们选择：STMicroelectronics→STM32F1 Series→STM32F103→STM32F103ZE（如果使用的是其他芯片，选择相应的型号就可以了），如图 8.1.2.3 所示。

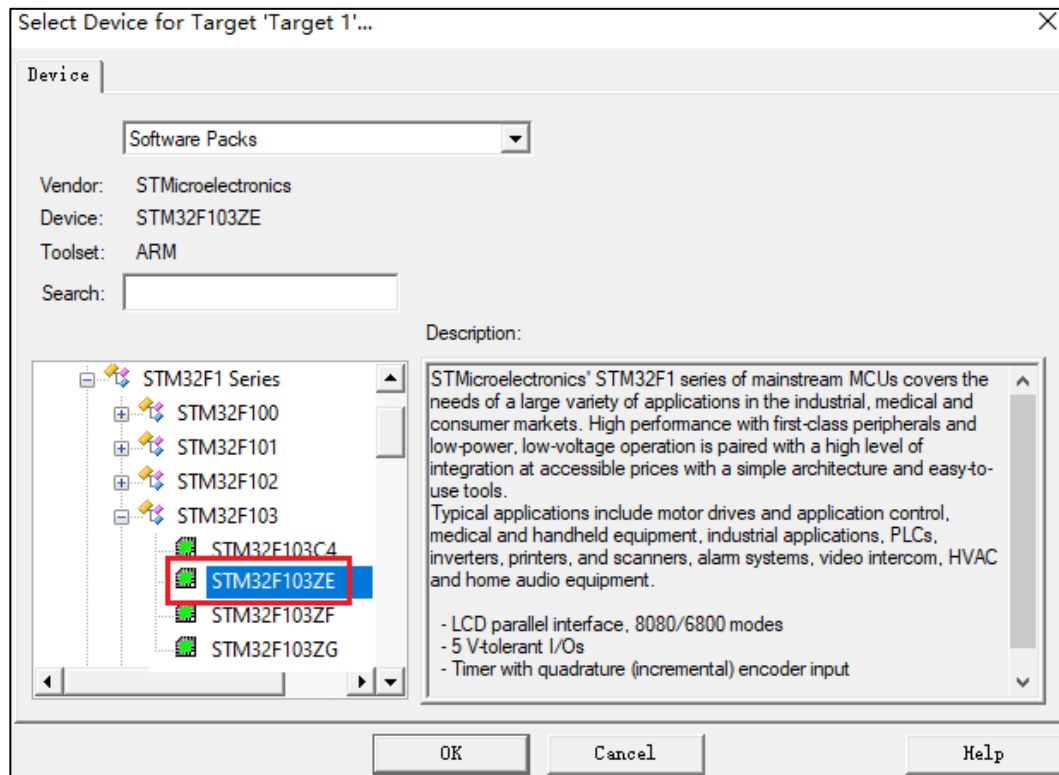


图 8.1.2.3 选择芯片型号

特别注意：一定要安装对应的器件支持包（即 **pack** 包）才会显示这些内容哦，如果没得选择，请关闭 MDK，然后安装光盘：6，软件资料\1，软件\MDK5\Keil.STM32F1xx_DFP.2.3.0 这个安装包后重试。

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

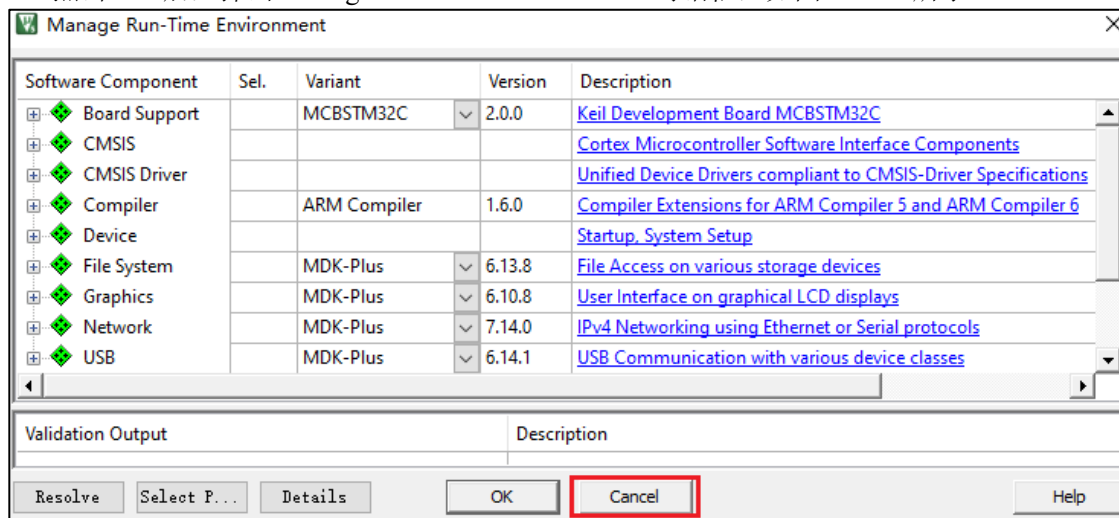


图 8.1.2.4 Manage Run-Time Environment 界面

在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，不过这里我们不需要。我们直接点击 Cancel 即可。这样就得到了我们的初步工程，如图 8.1.2.5 所示。

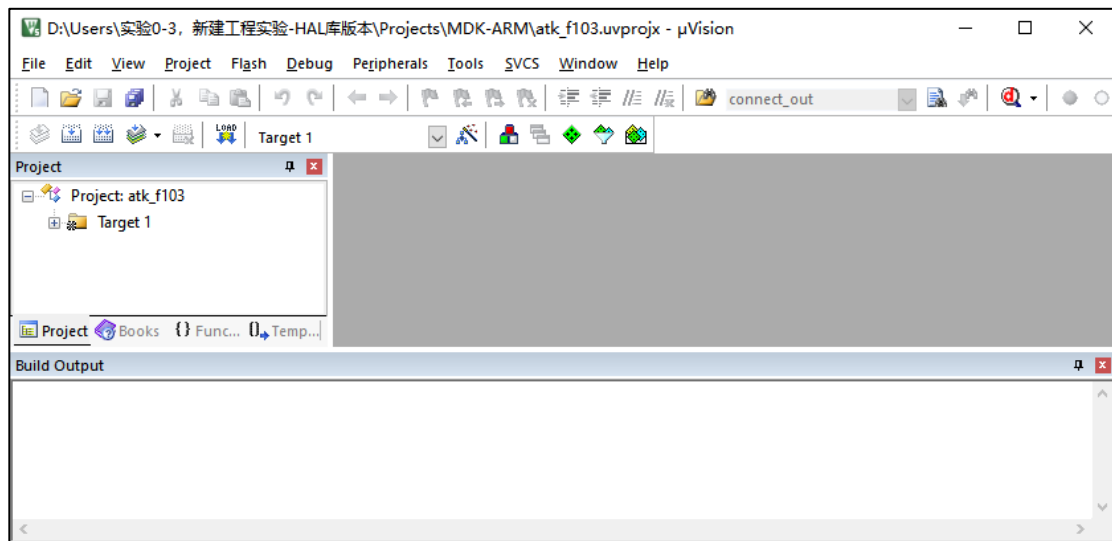


图 8.1.2.5 初步工程

这只是一个工程的框架，我们还需要把自己需要用到的文件添加到工程里面。虽然前面，我们在工程文件夹里放了很多文件，但是它们并没有关联到工程里面。

我们看看初步工程建立好后，MDK-ARM 文件夹的内容，如图 8.1.2.6 所示。

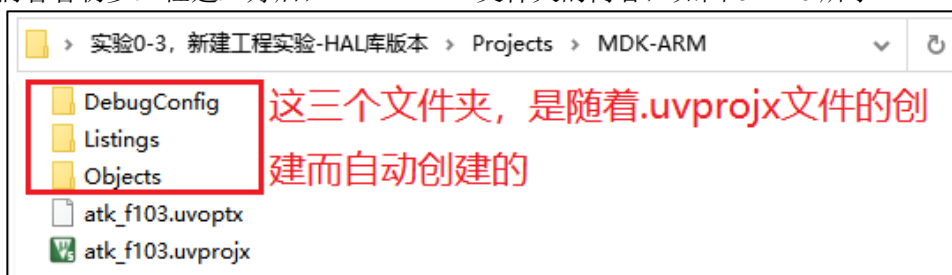


图 8.1.2.6 工程文件夹 MDK-ARM 目录

这里我们说明一下，atk_f103.uvprojx 是工程文件，非常关键，不能轻易删除，MDK5.31 生成的工程文件是以.uvprojx 为后缀。DebugConfig, Listings 和 Objects 三个文件夹是 MDK 自动生成的文件夹。其中 DebugConfig 文件夹用于存储一些调试配置文件，Listings 和 Objects 文件夹用来存储 MDK 编译过程的一些中间文件。这里，我们把 Listings 和 Objects 文件夹删除，我们后面会把编译中间文件存放到 Output 文件夹。当然，我们不删除这两个文件夹也没有关系，只是我们不用它而已。

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

8.1.3 添加文件

本节将分 5 个步骤：1，设置工程名和分组；2，添加启动文件；3，添加 User 源码；4，添加 SYSTEM 源码；5，添加 STM32F1xx_HAL_Driver 源码。

1. 设置工程名和分组名

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

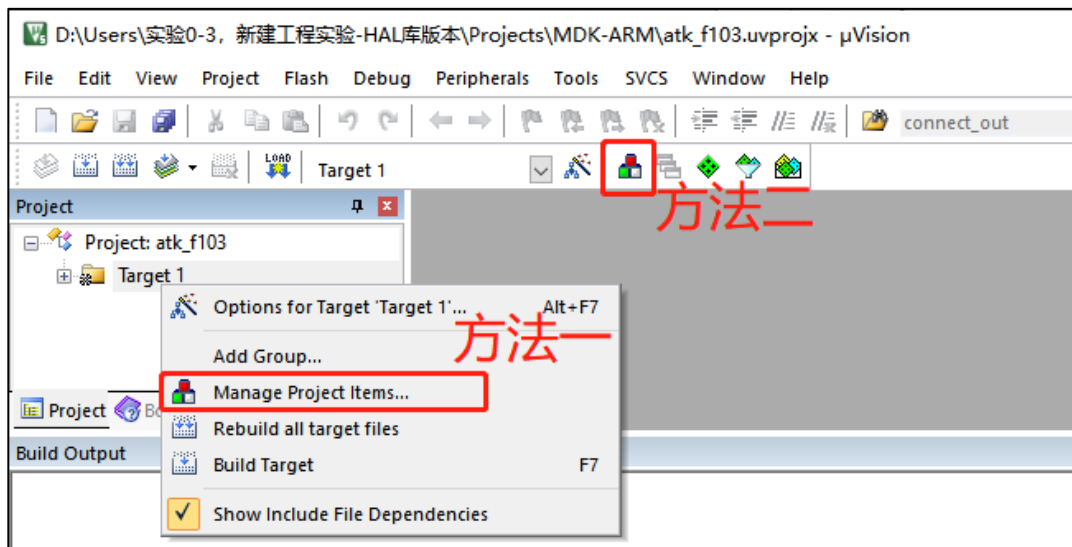


图 8.1.3.1 点击 Management Project Itmes

在工程管理界面，我们可以执行设置工程名字（Project Targets）、分组名字（Groups）以及添加每个分组的文件（Files）等操作。我们设置工程名字为：Template，并设置五个分组：Startup（存放启动文件）、User（存放 main.c 等用户代码）、Drivers/SYSTEM（存放系统级驱动代码）、Drivers/STM32F1xx_HAL_Driver（存放 ST 提供的 HAL 库驱动代码）、Readme（存放工程说明文件），如图 8.1.3.2 所示：

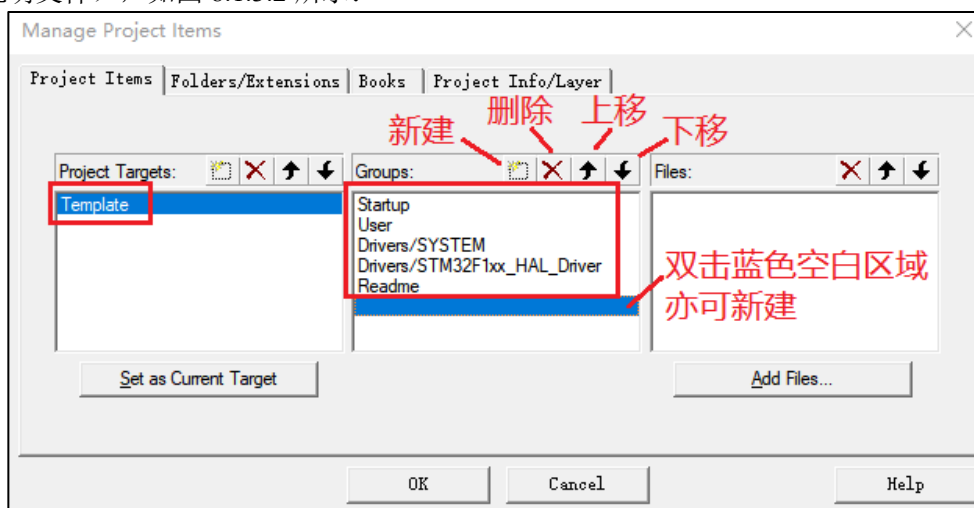


图 8.1.3.2 设置工程名和分组名

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

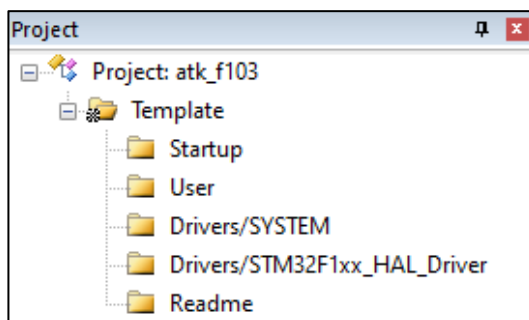


图 8.1.3.3 设置成功

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

注意：为了让工程结构清晰，我们会尽量让 MDK 的工程分组和我们前面新建的工程文件夹对应起来，由于 MDK 分组不支持多级目录，因此我们将路径也带入分组命名里面，以便区分。如：User 分组对应 User 文件夹里面的源码，Drivers/SYSTEM 分组，对应 Drivers/SYSTEM 文件夹里面的源码，Drivers/STM32F1xx_HAL_Driver 分组对应 Drivers/STM32F1xx_HAL_Driver 文件夹里面的源码等。

2. 添加启动文件

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

该文件由 ST 官方提供，对于 STM32F103 来说有 4 个启动文件可选，如表 8.1.3.1 所示：

| 启动文件 | 对应 FLASH 容量 | 说明 |
|-----------------------|--------------------|-----------------------|
| startup_stm32f103x6.s | Flash≤32KB | 用于小容量 F103 系列芯片的启动文件 |
| startup_stm32f103xb.s | 64KB≤Flash≤128KB | 用于中容量 F103 系列芯片的启动文件 |
| startup_stm32f103xe.s | 256KB≤Flash≤512KB | 用于大容量 F103 系列芯片的启动文件 |
| startup_stm32f103xg.s | 768KB≤Flash≤1024KB | 用于超大容量 F103 系列芯片的启动文件 |

表 8.1.3.1 STM32F103 系列启动文件

启动文件存放的位置在前面也有所说明，因为我们开发板使用的是 STM32F103ZET6，对应的启动文件为：startup_stm32f103xe.s。

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

这里我们使用方法 1 添加（路径：实验 0-3，新建工程实验-HAL 库版本\Drivers\CMSIS\Device\ST\STM32F1xx\Source\Templates\arm），如图 8.1.3.4 所示：

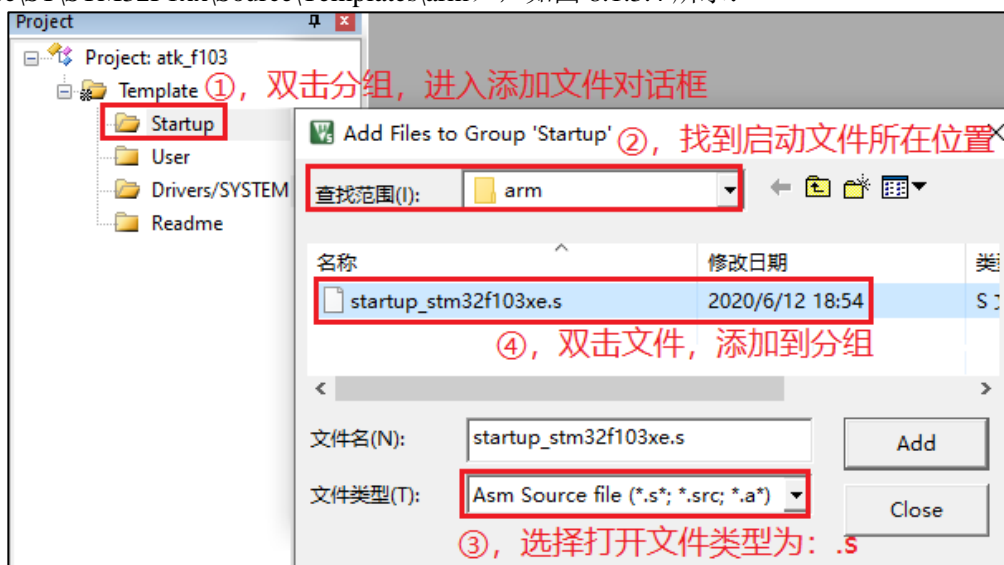


图 8.1.3.4 双击分组添加启动文件（startup_stm32f103xe.s）

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

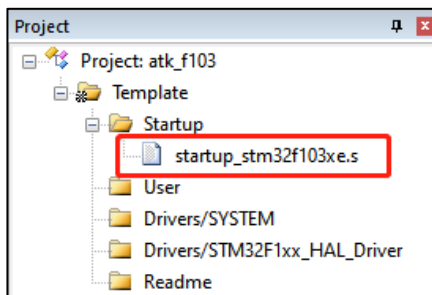


图 8.1.3.5 启动文件添加成功

3. 添加 User 源码

这里我们在工程管理界面（方法 2）进行 User 源码添加。点击：按钮，进入工程管理界面，选中 User 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32f1xx_it.c 和 system_stm32f1xx.c 到该分组下，如图 8.1.3.6 所示：

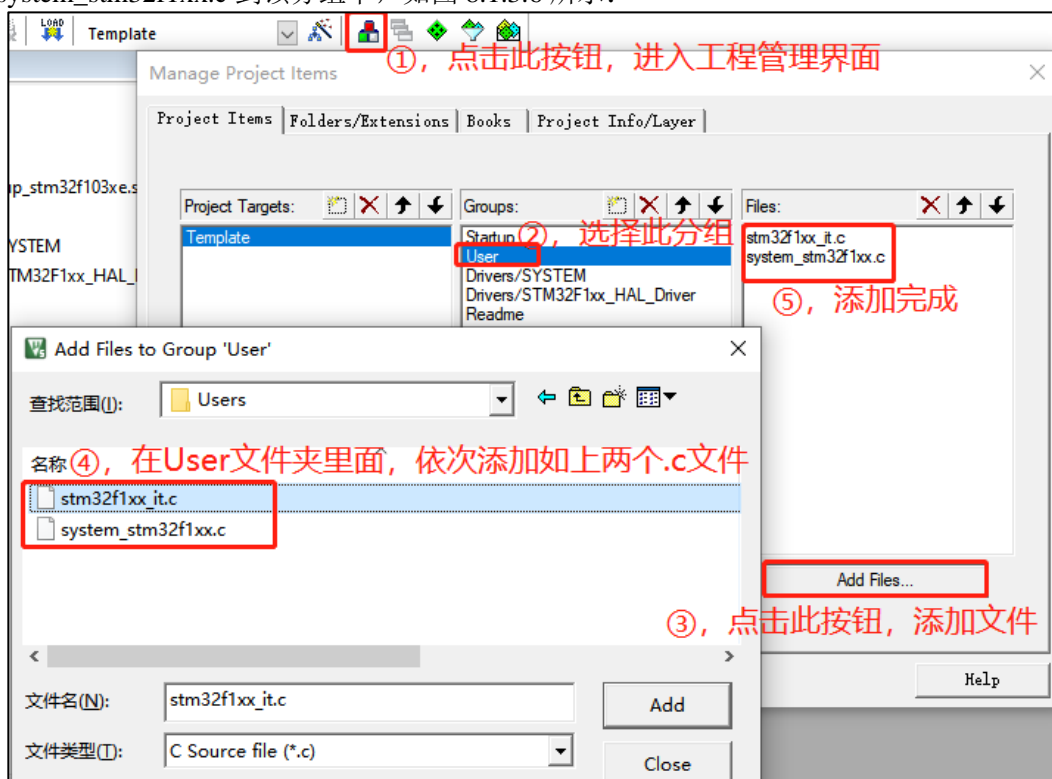


图 8.1.3.6 添加 User 源码

注意：这些源码都是在第 8.1.1 小节的第一步拷贝过来的，如果之前没拷贝，是找不到这些源码的。添加完成后，如图 8.1.3.7 所示：

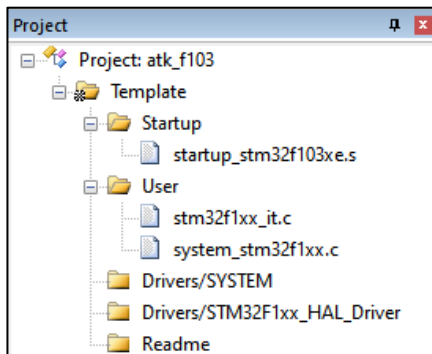



图 8.1.3.7 User 源码添加完成

4. 添加 SYSTEM 源码

同样的，我们也是在工程管理界面进行 SYSTEM 源码添加。点击：按钮，进入工程管理界面，选中 Drivers/SYSTEM 分组，然后点击：Add Files，进入文件添加对话框，依次添加 delay.c、sys.c 和 usart.c 到该分组下，如图 8.1.3.8 所示：

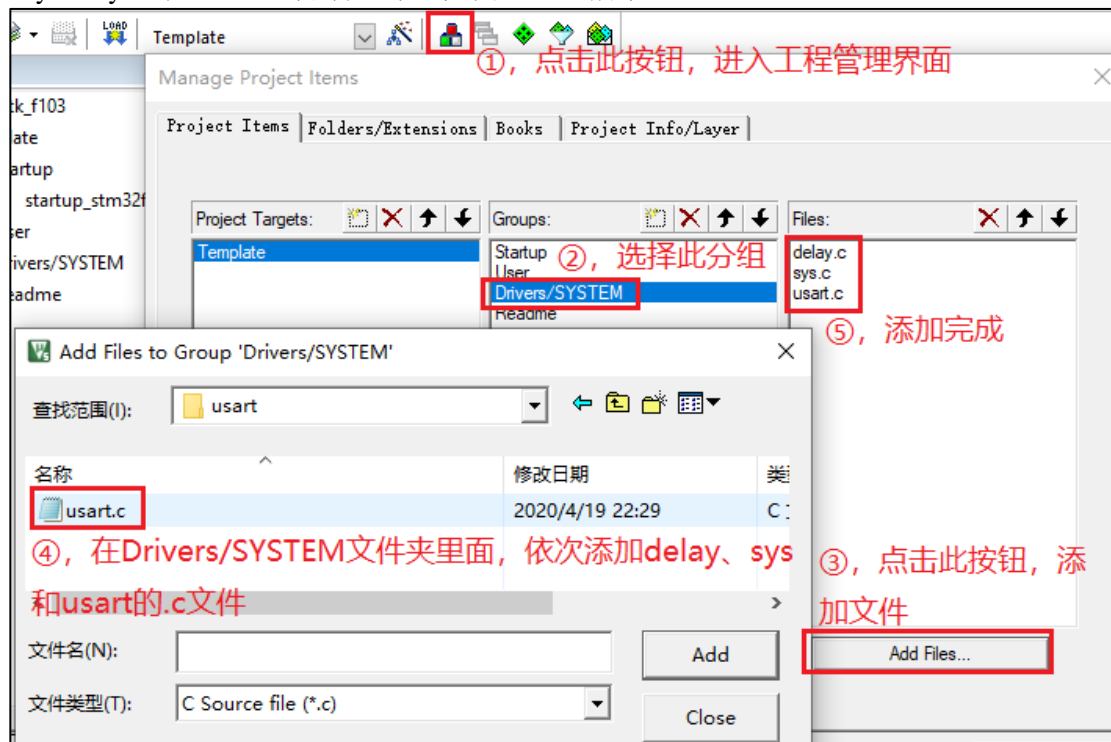


图 8.1.3.8 添加 SYSTEM 源码

添加完成后，如图 8.1.3.9 所示：

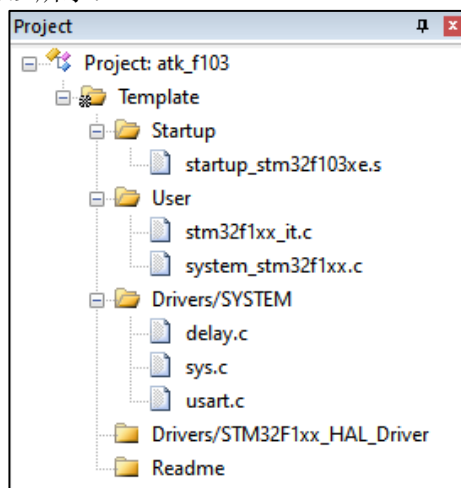



图 8.1.3.9 SYSTEM 源码添加完成

5. 添加 STM32F1xx_HAL_Driver 源码

接下来我们往 Drivers/STM32F1xx_HAL_Driver 分组里添加文件，这里的操作跟前面添加 SYSTEM 源码一样的。点击：按钮，进入工程管理界面，选中 Drivers/STM32F1xx_HAL_Driver 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32f1xx_hal.c、stm32f1xx_hal_cortex.c、stm32f1xx_hal_dma.c、stm32f1xx_hal_gpio.c、stm32f1xx_hal_gpio_ex.c、stm32f1xx_hal_rcc.c、stm32f1xx_hal_rcc_ex.c、stm32f1xx_hal_uart.c 和 stm32f1xx_hal_usart.c 到

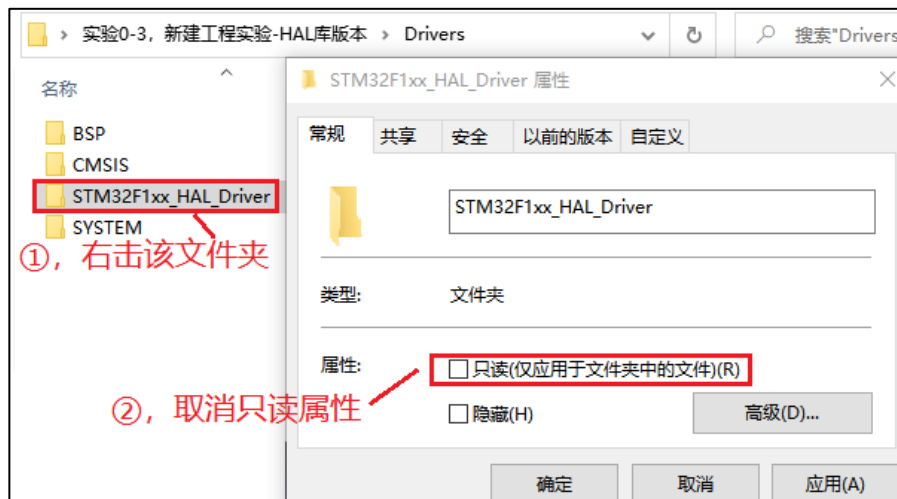



图 8.1.3.12 取消工程文件夹的只读权限

8.1.4 魔术棒设置

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

1. 设置 Target 选项卡

在魔术棒→Target 选项卡里面，我们进行如图 8.1.4.1 所示设置：

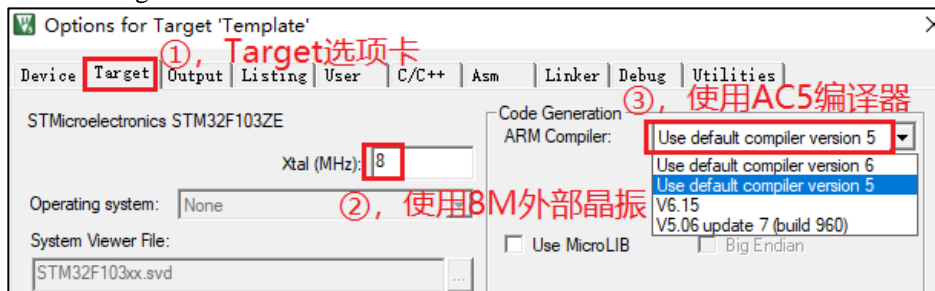


图 8.1.4.1 Target 选项卡设置

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

这里我们说明一下 AC5 和 AC6 编译的差异，如表 8.1.4.1 所示：

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

表 8.1.4.1 AC5&AC6 简单对比

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

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

表 8.1.4.2 AC5&AC6 设置差异

2. 设置 Output 选项卡

在魔术棒→Output 选项卡里面，进行如图 8.1.4.2 所示设置：

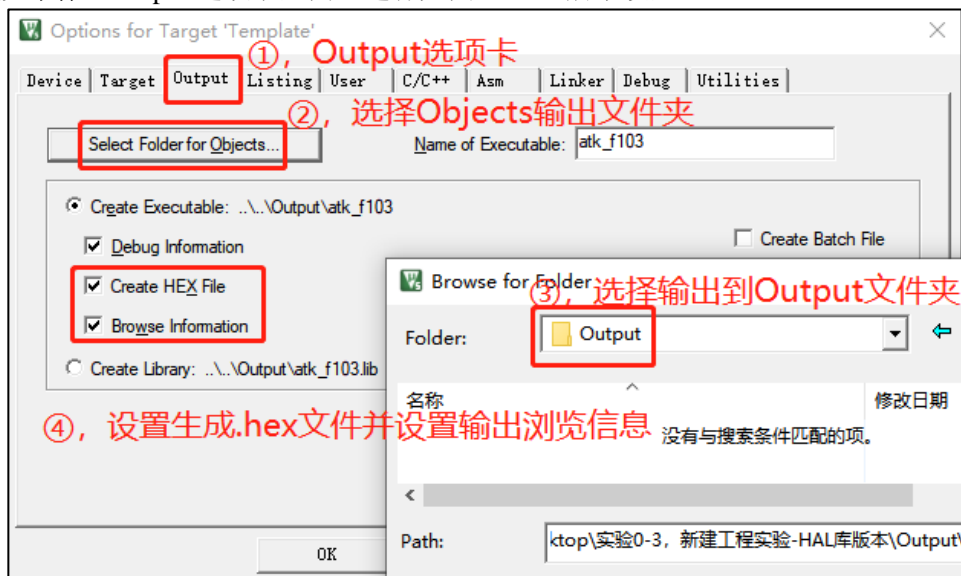


图 8.1.4.2 设置 Output 选项卡

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

3. 设置 Listing 选项卡

在魔术棒→Listing 选项卡里面，进行如图 8.1.4.3 所示设置：

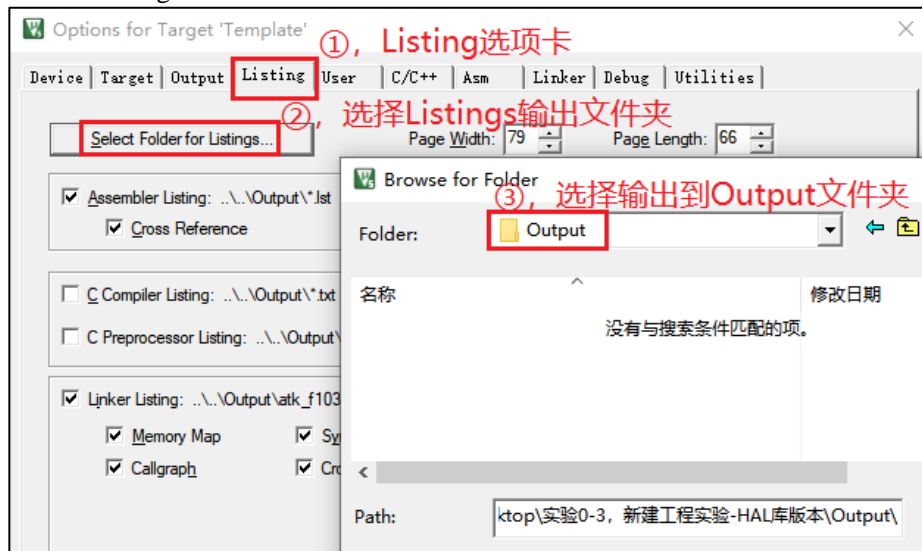


图 8.1.4.3 设置 Listing 选项卡

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

4. 设置 C/C++ 选项卡

在魔术棒→C/C++ 选项卡里面，进行如图 8.1.4.4 所示设置：

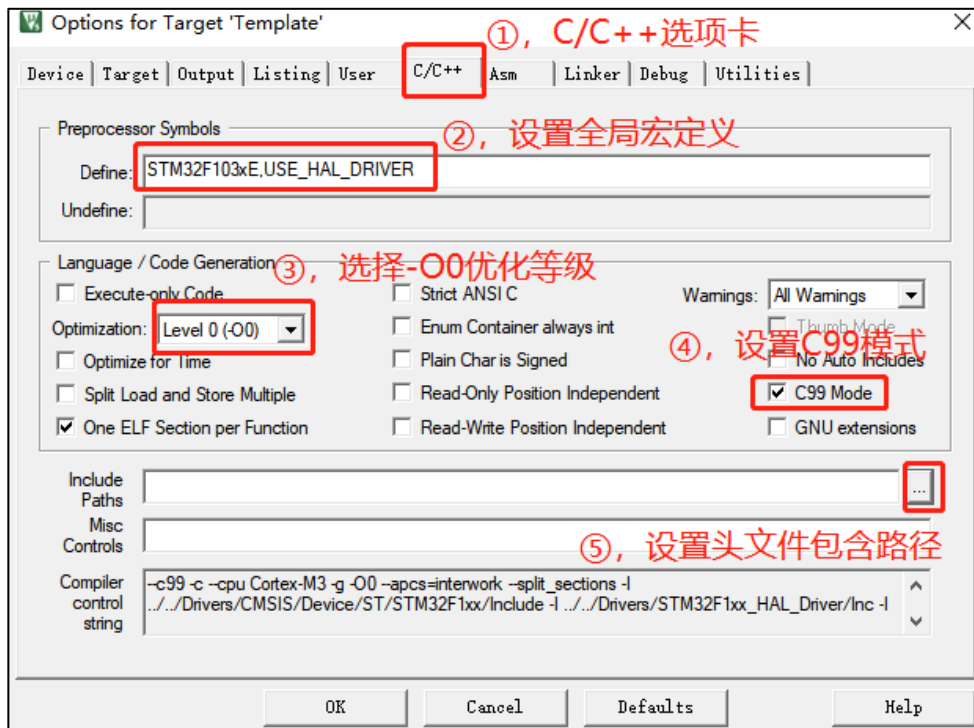


图 8.1.4.4 设置 C/C++选项卡

在②处设置了全局宏定义：USE_HAL_DRIVER 和 STM32F103xE，他们之间是用英文逗号隔开的。添加全局宏定义标识符，在工程中任何地方都可见，在 stm32f1xx.h 里面会用到该宏定义。

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

在④处勾选 C99 模式，即使用 C99 C 语言标准。

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

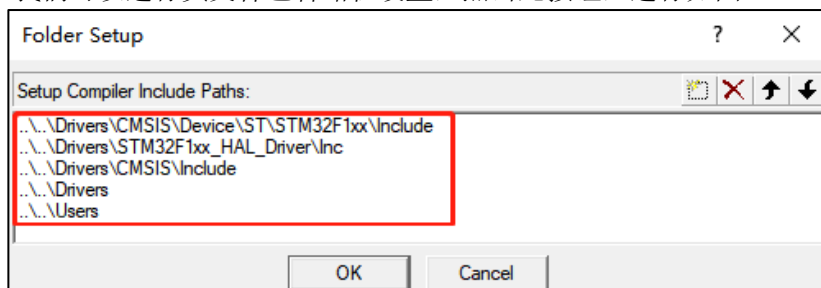


图 8.1.4.5 设置头文件包含路径

上图中我们设置了 5 个头文件包含路径，其中 4 个在 Drivers 文件夹下，一个在 User 文件夹下。为避免频繁设置头文件包含路径，正点原子最新源码的 include 全部使用相对路径，也就是我们只需要在头文件包含路径里面指定一个文件夹，那么该文件夹下的其他文件夹里面的源码，如果全部是使用相对路径，则无需再设置头文件包含路径了，直接在 include 里面就指明了头文件所在。

关于相对路径，这里大家记住 3 点：

- 1，默认路径就是指 MDK 工程所在的路径，即.uvprojx 文件所在路径（文件夹）
- 2，“./”表示当前目录（相对当前路径，也可以写做“.”）
- 3，“../”表示当前目录的上一层目录（也可以写做“..”）

举例来说，上图中：..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include，前面两个“..”，表示 Drivers 文件夹在当前 MDK 工程所在文件夹（MDK-ARM）的上 2 级目录下，具体解释如图 8.1.4.6 所示：

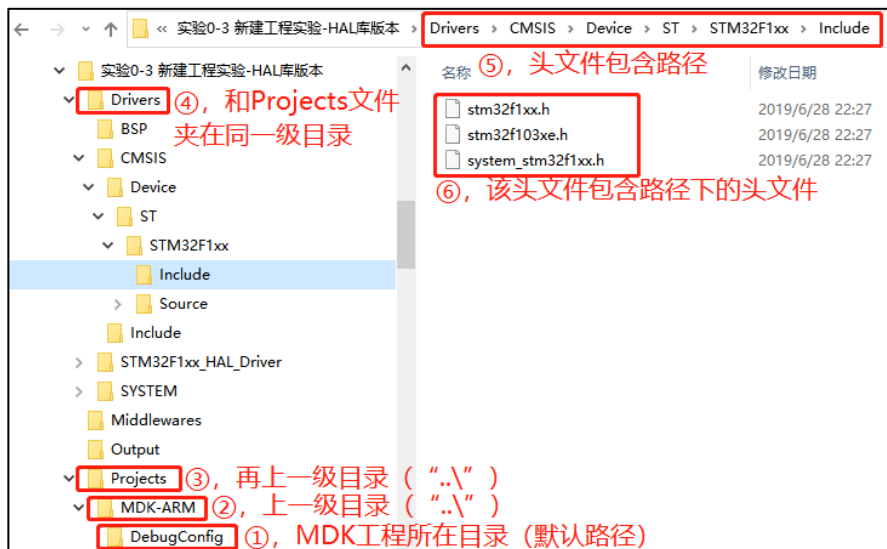


图 8.1.4.6 ..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include 的解释

上图表示根据头文件包含路径：..\..\Drivers\CMSIS\Device\ST\STM32F1xx\Include，编译器可以找到⑥处所包含的这些头文件，即代码里面可以直接 include 这些头文件使用。

再举个例子，在完成如图 8.1.4.5 所示的头文件包含路径设置以后，我们在代码里面编写：
#include "../SYSTEM/sys/sys.h"

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

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

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

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

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

5. 设置 Debug 选项卡

在魔术棒→Debug 选项卡里面，进行如图 8.1.4.7 所示设置：

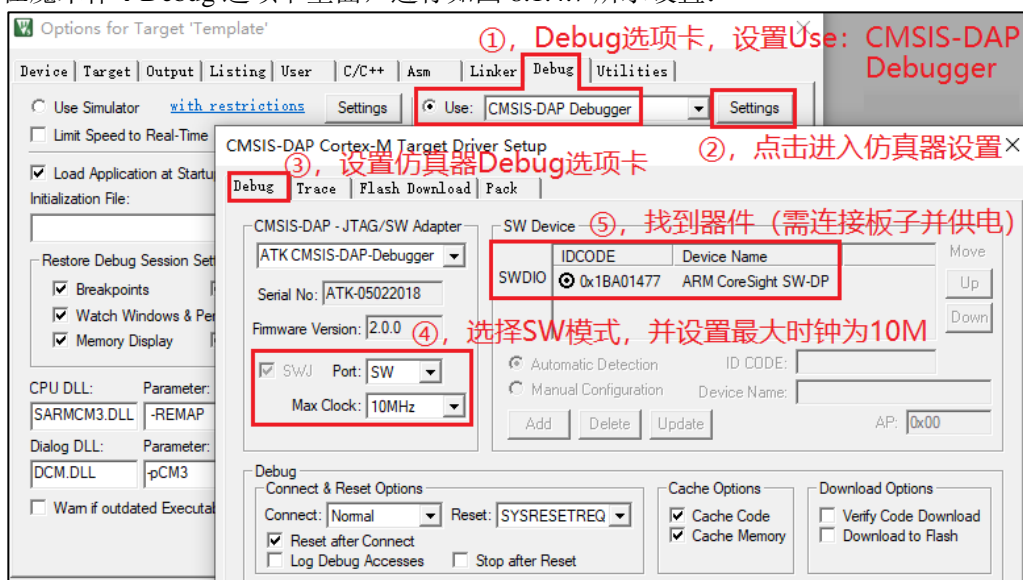


图 8.1.4.7 Debug 选项卡设置

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

6. 设置 Utilities 选项卡

在魔术棒→Utilities 选项卡里面，进行如图 6.1.4.8 所示设置：

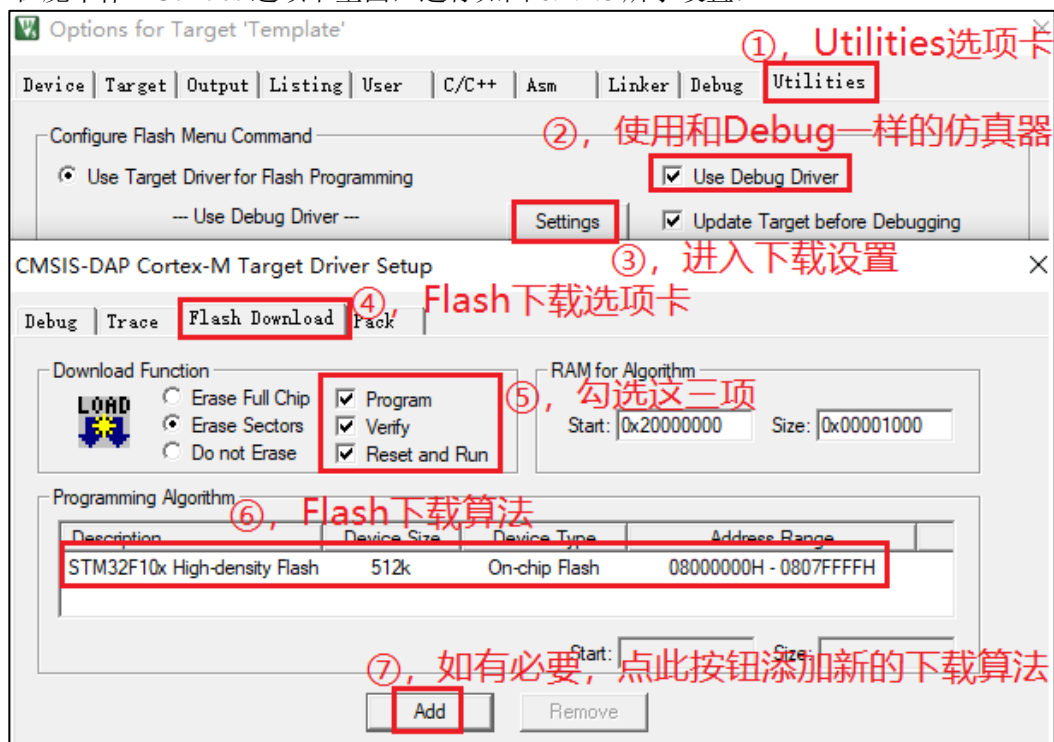


图 8.1.4.8 Utilities 选项卡设置

图中⑥处下载算法，是 MDK 默认添加的，针对 STM32F10x 大容量系列产品（FLASH 容量在 256KB~512KB 之间）。一般我们用这个即可。如果⑥处没有下载算法，则点击⑦处按钮，执行添加一下下载算法即可（名字和⑥处的算法名字一样）。

8.1.5 添加 main.c，并编写代码

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

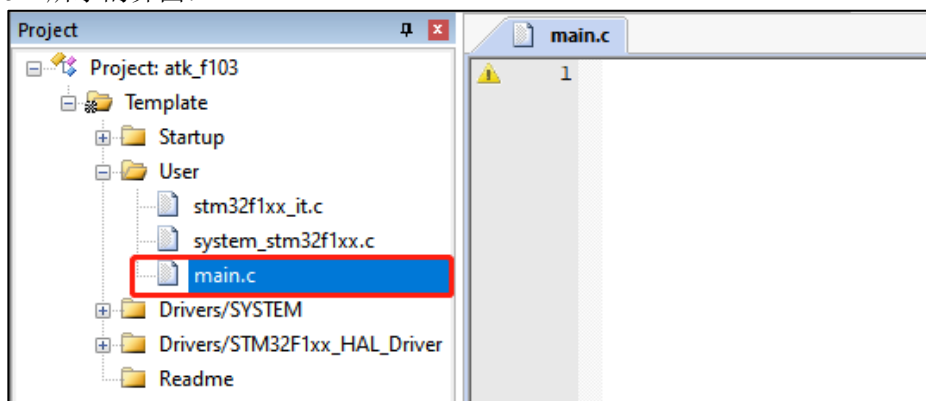


图 8.1.5.1 在 User 分组下加入 main.c 文件

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

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

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

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    led_init(); /* LED 初始化 */
    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET); /* PB5 置 1 */
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_RESET); /* PE5 置 0 */
        delay_ms(500);
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET); /* PB5 置 1 */
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_RESET); /* PE5 置 0 */
        delay_ms(500);
    }
}

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


    __HAL_RCC_GPIOB_CLK_ENABLE(); /* IO 口 PB 时钟使能 */
    __HAL_RCC_GPIOE_CLK_ENABLE(); /* IO 口 PE 时钟使能 */

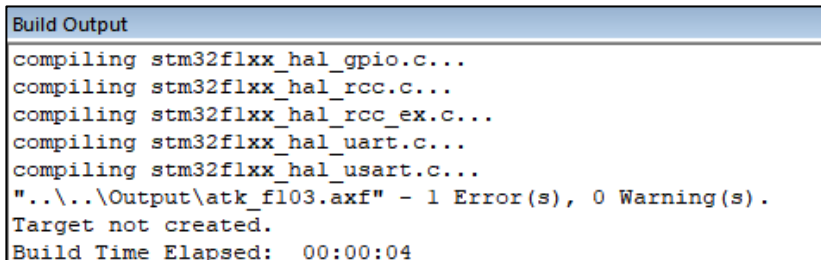
    gpio_initstruct.Pin = GPIO_PIN_5; /* LED0 引脚 */
    gpio_initstruct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_initstruct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_initstruct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(GPIOB, &gpio_initstruct); /* 初始化 LED0 引脚 */

    gpio_initstruct.Pin = GPIO_PIN_5; /* LED1 引脚 */
    HAL_GPIO_Init(GPIOE, &gpio_initstruct); /* 初始化 LED1 引脚 */
}
```

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

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

编写完 main.c 以后, 我们点击:  (Rebuild) 按钮, 编译整个工程, 编译结果如图 8.1.5.2。



```
Build Output
compiling stm32flxx_hal_gpio.c...
compiling stm32flxx_hal_rcc.c...
compiling stm32flxx_hal_rcc_ex.c...
compiling stm32flxx_hal_uart.c...
compiling stm32flxx_hal_usart.c...
"...\\Output\\atk_fl03.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:04
```

图 8.1.5.2 编译结果

我们在 Build Output 下找到第一个错误, 双击这个错误信息定位到错误发生的代码位置。

如图这个错误说找不到 main.h, 因为我们也不需要用到 main.h, 双击这个错误会弹出下面的 STM32F1xx_it.c 文件对应包含 main.h 的语句。我们只需要把它删除, 然后重新编译。

再次编译, 发现还有一处警告, 这里是 HAL_IncTick 函数没有声明, 如图 8.1.5.3 所示。

```
Build Output
Rebuild target 'Template'
assembling startup_stm32f103xe.s...
compiling stm32f1xx_it.c...
..\..\Users\stm32f1xx_it.c(141): warning: #223-D: function "HAL_IncTick" declared implicitly
    HAL_IncTick();
..\..\Users\stm32f1xx_it.c: 1 warning, 0 errors
```

图 8.1.5.3 编译警告

因为这个函数是在 STM32F1xx_hal.c 定义了, 并且在 STM32F1xx_hal.h 声明了。我们把 STM32F1xx_hal.h 包含进来即可。这里还有一个原因是整个工程没有包含 STM32F1xx_hal.h 的语句, 我们需要用到它, 所以在这里把它包含进来。官方的 main.h 是有包含这个头文件的。我们不用 main.h 文件, 我们在 STM32F1xx_it.c 文件刚才删除包含 main.h 的语句的位置, 编写包含 STM32F1xx_hal.h 语句, 如图 8.1.5.3 所示。

```
main.c  stm32f1xx_it.c
22  /* Includes .....
23  #include "stm32f1xx_it.h"
24  #include "stm32f1xx_hal.h"
```

图 8.1.5.3 包含 STM32F1xx_hal.h 头文件到工程

再进行编译就会发现 0 错误 0 警告, 结果如图 8.1.5.4 所示:

```
Build Output
compiling stm32f1xx_hal_usart.c...
linking...
Program Size: Code=5390 RO-data=362 RW-data=28 ZI-data=1900
FromELF: creating hex file...
"..\..\Output\atk_f103.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:05
```


图 8.1.5.4 编译结果

编译结果提示: 代码总大小 (Program Size) 为: FLASH 占用 5780 字节 (Code+RO+RW), SRAM 占用 1928 字节 (RW+ZI); 并成功创建了 Hex 文件 (可执行文件, 放在 Output 目录下)。

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

至此, 新建 HAL 库版本 MDK 工程完成。

8.2 下载验证

有两种方法可以给 STM32F103 芯片下载代码: 1, 使用串口下载; 2, 使用仿真器下载。这两种下载方法, 我们在本书的 4.2 和 4.3 节给大家有做过详细介绍。这里我们以仿真器下载为例, 在 MDK 主界面, 点击:  (下载按钮, 也可以按键盘快捷键: F8), 就可以将代码下载到开发板, 如图 8.2.1 所示:

```
Build Output
Load "..\..\Output\atk_f103.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 19:11:46
```

图 8.2.1 下载成功

上图提示: Application running..., 则表示代码下载成功, 且开始运行。可以看到 LED0 和 LED1 交叉闪烁。

第九章 STM32 启动过程分析

本章给大家分析 STM32F1 的启动过程，这里的启动过程是指从 STM32 芯片上电复位执行的第一条指令开始，到执行用户编写的 main 函数这之间的过程。我们编写程序，基本都是用 C 语言编写，并且以 main 函数作为程序的入口。但是事实上，main 函数并非最先执行的，在此之前需要做一些准备工作，准备工作通过启动文件的程序来完成。理解 STM32 启动过程，对今后的学习和分析 STM32 程序有很大的帮助。

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

本章将分为如下几个小节：

9.1 启动模式

9.2 启动文件分析

9.3 map 文件分析

9.1 启动模式

我们知道的复位方式有三种：上电复位，硬件复位和软件复位。当产生复位，并且离开复位状态后，CM3 内核做的第一件事就是读取下列两个 32 位整数的值：

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

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

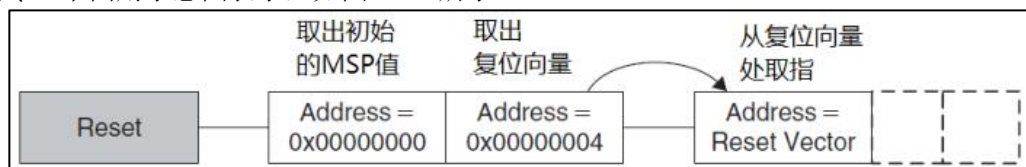


图 9.1.1 复位序列

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

对于 STM32F1 的启动模式（也称自举模式），我们看表 9.1.1 进行分析。

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

表 9.1.1 启动模式选择表

注：启动引脚的电平：0：低电平；1：高电平；x:任意电平，即高低电平均可

由表 9.1.1 可以看到，STM32F1 根据 BOOT 引脚的电平选择启动模式，这两个 BOOT 引脚根据外部施加的电平来决定芯片的启动地址。（0 和 1 的准确电平范围可以查看 F103 系列数据手册 I/O 特性表，但我们最好是设置成 Gnd 和 VDD 的电平值）

(1) 内部 FLASH 启动方式

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

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

(2) 内部 SRAM 启动方式

类似于内部 Flash，当芯片上电后采样到 BOOT0 和 BOOT1 引脚均为高电平时，地址 0x00000000 和 0x00000004 被映射到内部 SRAM 的首地址 0x20000000 和 0x20000004，内核从 SRAM 空间获取内容进行自举。在实际应用中，由启动文件 startup_stm32f103xe.s 决定了 0x00000000 和 0x00000004 地址存储什么内容，链接时，由分散加载文件(sct)决定这些内容的绝对地址，即分配到内部 FLASH 还是内部 SRAM。

(3) 系统存储器启动方式

当芯片上电后采样到 BOOT0=1, BOOT1=0 的组合时，内核将从系统存储器的 0x1FFF0000 及 0x1FFF0004 获取 MSP 及 PC 值进行自举。系统存储器是一段特殊的空间，用户不能访问，ST 公司在芯片出厂前就在系统存储器中固化了一段代码。因而使用系统存储器启动方式时，内核会执行该代码，该代码运行时，会为 ISP(In System Program)提供支持，在 STM32F1 上最常见的是检测 USART1 传输过来的信息，并根据这些信息更新自己内部 FLASH 的内容，达到升级产品应用程序的目的，因此这种启动方式也称为 ISP 启动方式。

9.2 启动文件分析

STM32 启动文件由 ST 官方提供，在官方的 STM32Cube 固件包里，对于 STM32F103 系列芯片的启动文件，我们选用的是 startup_STM32F103xe.s 这个文件。启动文件用汇编编写，是系统上电复位后第一个执行的程序。

启动文件主要做了以下工作：

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

9.2.1 启动文件中的一些指令

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

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

表 9.2.1.1 启动文件的汇编指令

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

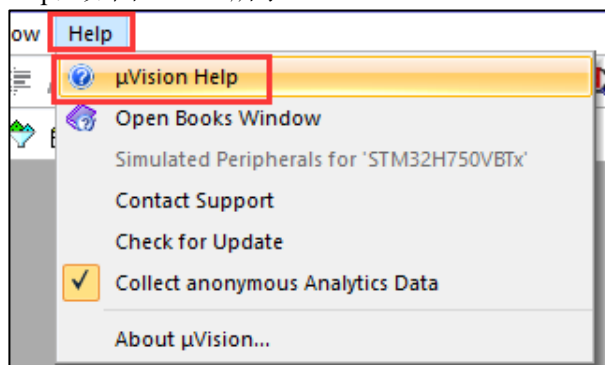


图 9.2.1.1 打开索引搜索工具的方法

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

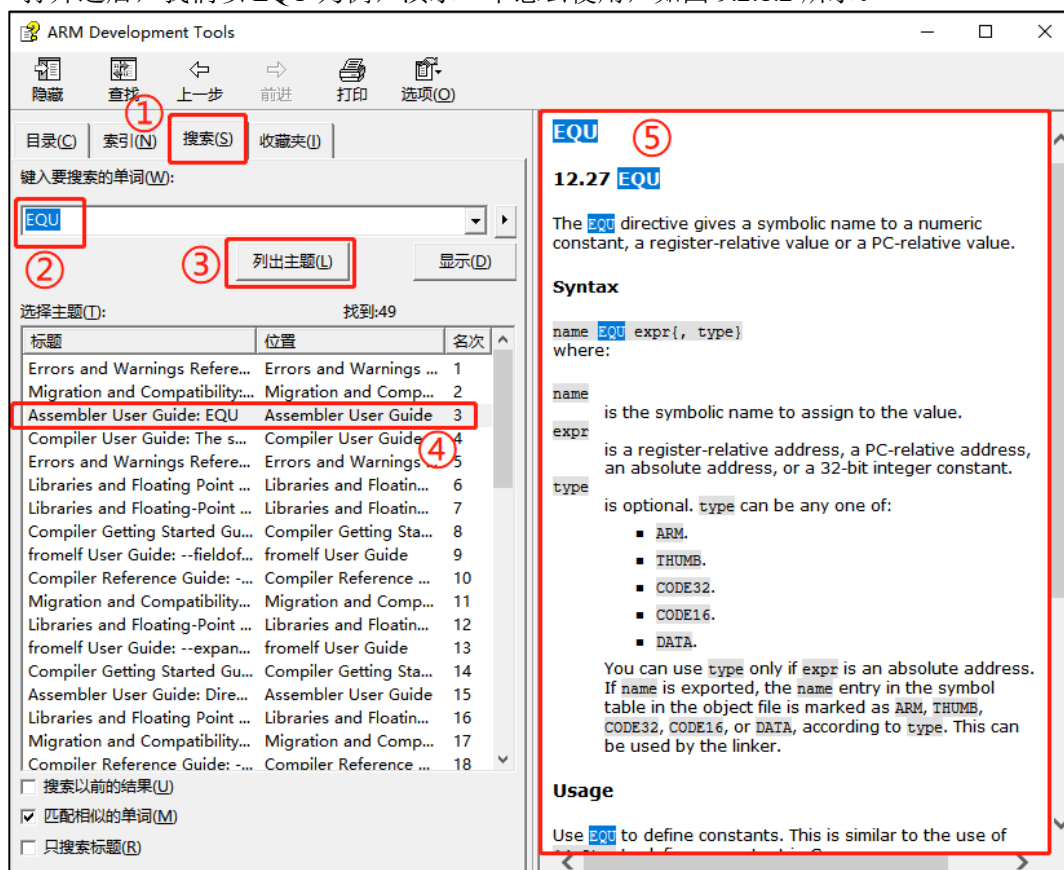


图 9.2.1.2 搜索 EQU 汇编指令

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

9.2.2 启动文件代码讲解

(1) 栈空间的开辟

栈空间的开辟，源码如图 9.2.2.1 所示：

```

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

图 9.2.2.1 栈空间的开辟

源码含义：开辟一段大小为 0x0000 0400（1KB）的栈空间，段名为 STACK，NOINIT 表示不初始化；READWRITE 表示可读可写；ALIGN=3，表示按照 2^3 对齐，即 8 字节对齐。AREA 汇编一个新的代码段或者数据段。

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

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

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

(2) 堆空间的开辟

堆空间的开辟，源码如图 9.2.2.2 所示：

```

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

图 9.2.2.2 堆空间的开辟

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

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

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

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

```

50                PRESERVE8
51                THUMB
    
```

图 9.2.2.3 PRESERVE8 和 THUMB 指令

PRESERVE8：指示编译器按照 8 字节对齐。

THUMB：指示编译器之后的指令为 THUMB 指令。

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

(3) 中断向量表定义（简称：向量表）

为中断向量表定义一个数据段，如图 9.2.2.4 所示：

```

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

图 9.2.2.4 为中断向量表定义一个数据段

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

STM32F103 的中断向量表定义代码，如图 9.2.2.5 所示。

```

60  __Vectors ..... DCD ..... initial_sp ..... ; Top of Stack
61  ..... DCD ..... Reset_Handler ..... ; Reset_Handler
62  ..... DCD ..... NMI_Handler ..... ; NMI_Handler
63  ..... DCD ..... HardFault_Handler ..... ; Hard Fault_Handler
64  ..... DCD ..... MemManage_Handler ..... ; MPU Fault_Handler
65  ..... DCD ..... BusFault_Handler ..... ; Bus Fault_Handler
66  ..... DCD ..... UsageFault_Handler ..... ; Usage Fault_Handler
67  ..... DCD ..... 0 ..... ; Reserved
68  ..... DCD ..... 0 ..... ; Reserved
69  ..... DCD ..... 0 ..... ; Reserved
70  ..... DCD ..... 0 ..... ; Reserved
71  ..... DCD ..... SVC_Handler ..... ; SVC_Handler
72  ..... DCD ..... DebugMon_Handler ..... ; Debug Monitor_Handler
73  ..... DCD ..... 0 ..... ; Reserved
74  ..... DCD ..... PendSV_Handler ..... ; PendSV_Handler
75  ..... DCD ..... SysTick_Handler ..... ; SysTick_Handler
76
77  ..... ; External Interrupts (外部中断)
78  ..... DCD ..... WWDG_IRQHandler ..... ; Window Watchdog
79  ..... DCD ..... PVD_IRQHandler ..... ; PVD through EXTI Line detect
80  ..... DCD ..... TAMPER_IRQHandler ..... ; Tamper
81  ..... DCD ..... RTC_IRQHandler ..... ; RTC
82  ..... DCD ..... FLASH_IRQHandler ..... ; Flash
83  ..... DCD ..... RCC_IRQHandler ..... ; RCC
84  ..... ;...限于篇幅这里省略部分中断向量表...
85  ..... DCD ..... DMA2_Channel3_IRQHandler ..... ; DMA2_Channel3
86  ..... DCD ..... DMA2_Channel4_5_IRQHandler ..... ; DMA2_Channel4 & Channel5
87  __Vectors_End
88
89  __Vectors_Size EQU __Vectors_End - __Vectors
90
    
```

图 9.2.2.5 中断向量表定义代码

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

DCD：分配一个或者多个以字为单位的内存，以四字节对齐，并要求初始化这些内存。中断向量表被放置在代码段的最前面。例如：当我们的程序在 FLASH 运行时，那么向量表的起始地址是：0x0800 0000。结合图 9.2.2.5 可以知道，地址 0x0800 0000 存放的是栈顶地址。DCD 以四字节对齐分配内存，也就是下个地址是 0x0800 0004，存放的是 Reset_Handler 中断函数入口地址。

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

STM32F103 的中断向量表可以在《STM32F10xxx 参考手册_V10（中文版）.pdf》的第 9 章的 9.1.2 小节找到，与中断向量表定义代码是对应的。

(4) 复位程序

接下来是定义只读代码段，如图 9.2.2.6 所示：

```

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

图 9.2.2.6 定义只读代码段

定义一个段名为.text，只读的代码段，在 CODE 区。

复位子程序代码，如图 9.2.2.7 所示：


```

144 ; Reset handler
145 Reset_Handler...PROC
146 .....EXPORT...Reset_Handler.....[WEAK]
147 .....IMPORT...__main
148 .....IMPORT...SystemInit
149 .....LDR...R0,=SystemInit
150 .....BLX...R0
151 .....LDR...R0,=__main
152 .....BX...R0
153 .....ENDP

```

图 9.2.2.7 复位子程序代码

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

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

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

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

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

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

LDR 表示从存储器中加载字到一个存储器中。

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

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

(5) 中断服务程序

```

155 ; Dummy Exception Handlers (infinite loops which can be modified)
156
157 NMI_Handler...PROC
158 .....EXPORT...NMI_Handler.....[WEAK]
159 .....B.....
160 .....ENDP
161 HardFault_Handler\
162 .....PROC
163 .....EXPORT...HardFault_Handler.....[WEAK]
164 .....B.....
165 .....ENDP
166 ;...省略部分内核中断...
167 SysTick_Handler PROC
168 .....EXPORT...SysTick_Handler.....[WEAK]
169 .....B.....
170 .....ENDP
171
172 Default_Handler PROC
173
174 .....EXPORT...WWDG_IRQHandler.....[WEAK]
175 .....EXPORT...PVD_IRQHandler.....[WEAK]
176 .....EXPORT...TAMPER_IRQHandler.....[WEAK]
177 ;省略部分外部声明，这时是声明为weak函数
178 .....EXPORT...DMA2_Channel4_5_IRQHandler.[WEAK]
179
180 WWDG_IRQHandler
181 PVD_IRQHandler
182 TAMPER_IRQHandler
183 ;省略部分中断向量声明
184 DMA2_Channel4_5_IRQHandler
185 .....B.....
186
187 .....ENDP

```

图 9.2.2.8 中断服务程序

接下来就是中断服务程序了，如图 9.2.2.8 所示。

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

这些中断函数分为系统异常中断和外部中断，外部中断根据不同芯片有所变化。B 指令是跳转到一个标号，这里跳转到一个 ‘.’，表示无限循环。

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

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

这里的系统异常中断部分是内核的，外部中断部分是外设的。

(6) 用户堆栈初始化

ALIGN 指令，如图 9.2.2.9 所示：



图 9.2.2.9 ALIGN 指令

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

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

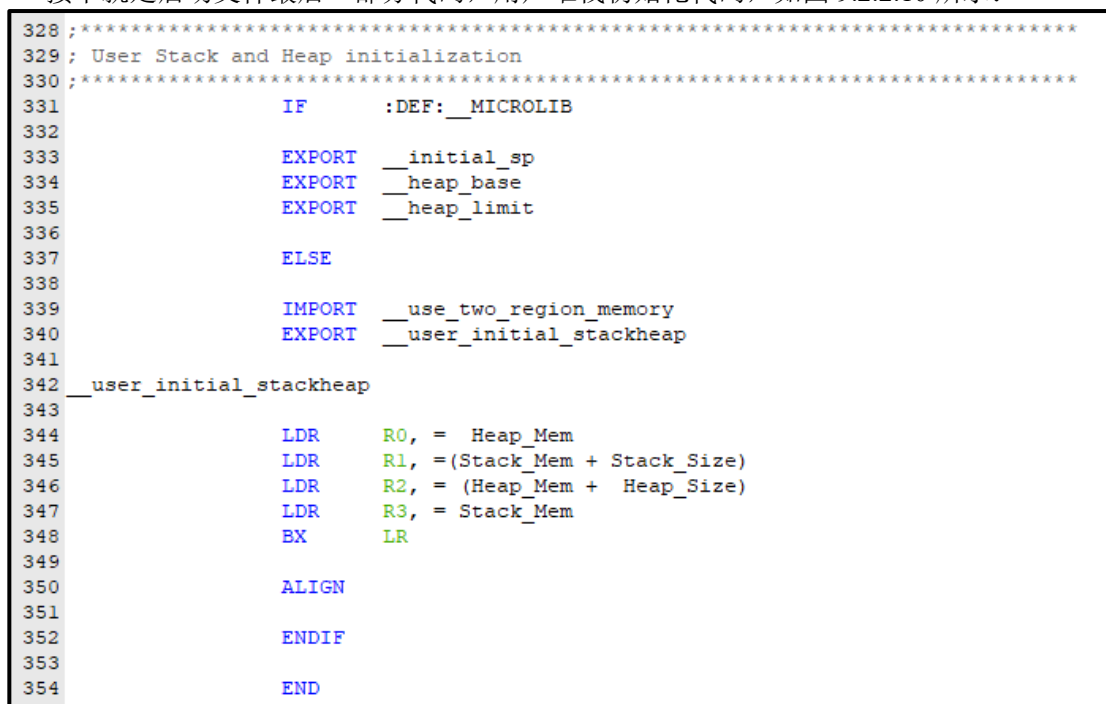


图 9.2.2.10 用户堆栈初始化代码

IF, ELSE, ENDIF 是汇编的条件分支语句。

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

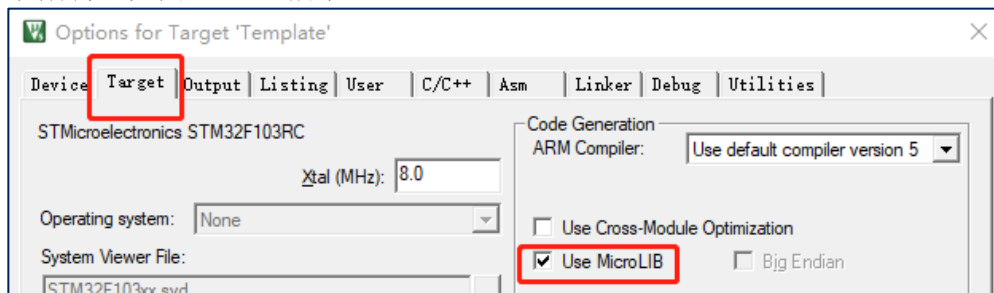


图 9.2.2.11 __MICROLIB 定义方法

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

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

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

IMPORT 声明 __use_two_region_memory 标号来自外部文件。

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

340 行标号 __user_initial_stackheap, 表示用户堆栈初始化程序入口。

接下来进行堆栈空间初始化, 堆是从低到高生长, 栈是从高到低生长, 是两个互相独立的数据段, 并且不能交叉使用。

344 行保存堆起始地址。345 行保存栈大小。346 行保存堆大小。347 行保存栈顶指针。348 行跳转到 LR 标号给出的地址, 不用返回。354 行 END 表示到达文件的末尾, 文件结束。

Use MicroLIB

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

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

9.2.3 系统启动流程

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

当产生复位, 并且离开复位状态后, CM3 内核做的第一件事就是读取下列两个 32 位整数的值:

- (1) 从地址 0x0800 0000 处取出堆栈指针 MSP 的初始值, 该值就是栈顶地址。
- (2) 从地址 0x0800 0004 处取出程序计数器指针 PC 的初始值, 该值指向中断服务程序 Reset_Handler。下面用示意图表示, 如图 9.2.3.1 所示。



图 9.2.3.1 复位序列

我们看看 STM32F103 开发板 HAL 库例程的实验 1 跑马灯实验中, 取出的 MSP 和 PC 的值是多少, 方法如图 9.2.3.2 所示。

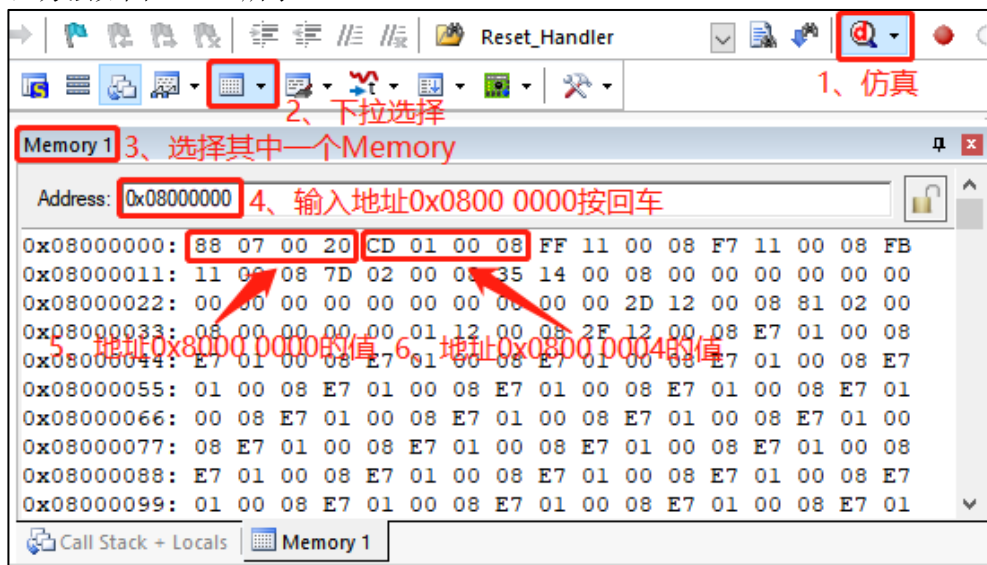


图 9.2.3.2 取出的 MPS 和 PC 的值

由图 9.2.3.2 可以知道地址 0x0800 0000 的值是 0x2000 0788，地址 0x0800 0004 的值是 0x0800 01CD，即堆栈指针 SP=0x2000 0788，程序计数器指针 PC = 0x0800 01CD（即复位中断服务程序 Reset_Handler 的入口地址）。因为 CM3 内核是小端模式，所以倒着读。

请注意，这与传统的 ARM 架构不同——其实也和绝大多数的其它单片机不同。传统的 ARM 架构总是从 0 地址开始执行第一条指令。它们的 0 地址处总是一条跳转指令。而在 CM3 内核中，0 地址处提供 MSP 的初始值，然后就是向量表（向量表在以后还可以被移至其它位置）。向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令，就是 Reset_Handler 这个函数。下面继续以 MINI 开发板 HAL 库例程实验 1 跑马灯实验为例，代码从地址 0x0800 0000 开始被执行，讲解一下系统启动，初始化堆栈、MSP 和 PC 后的内存情况。

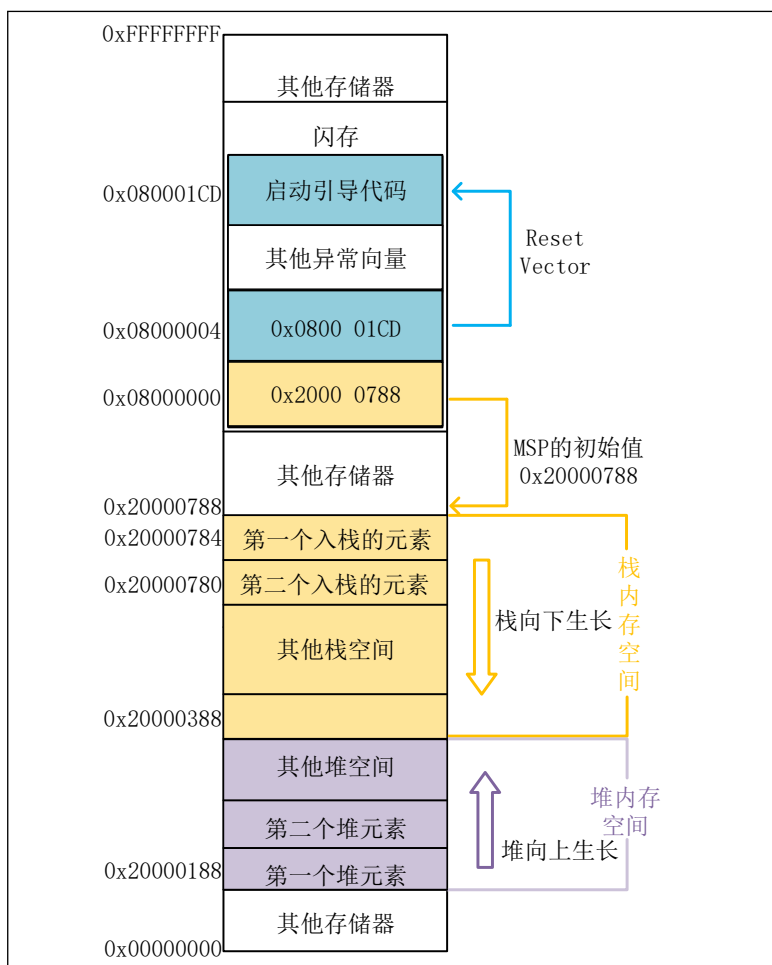


图 9.2.3.3 初始化堆栈、MSP 和 PC 后的内存情况

因为 CM3 使用的是向下生长的满栈，所以 MSP 的初始值必须是堆栈内存的末地址加 1。

举例来说，如果你的栈区域在 0x2000 0388 - 0x2000 0787（1KB 大小）之间，那么 MSP 的初始值就必须是 0x2000 0788。

向量表跟随在 MSP 的初始值之后——也就是第 2 个表目。

R15 是程序计数器，在汇编代码中，可以使用名字“PC”来访问它。ARM 规定：PC 最低两位并不表示真实地址，最低位 LSB 用于表示是 ARM 指令（0）还是 Thumb 指令（1），因为 CM3 主要执行 Thumb 指令，所以这些指令的最低位都是 1（都是奇数）。因为 CM3 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。比如说：

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

如果向 PC 写数据，就会引起一次程序的分支（但是不更新 LR 寄存器）。CM3 中的指令至少是半字对齐的，所以 PC 的 LSB 总是读回 0。然而，在分支时，无论是直接写 PC 的值还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即 LSB=1），表明是在 Thumb 状态下执

行。倘若写了 0，则视为转入 ARM 模式，CM3 将产生一个 fault 异常。

正因为上述原因，图 9.2.3.3 中使用 0x0800 01CD 来表达地址 0x0800 01CC。当 0x0800 01CD 处的指令得到执行后，就正式开始了程序的执行（即去到 C 的世界）。所以在此之前初始化 MSP 是必需的，因为可能第 1 条指令还没执行就会被 NMI 或是其它 fault 打断。MSP 初始化好后就已经为它们的服务例程准备好了堆栈。

STM32 启动文件分析就给大家介绍到这里，更多内容请看 [《STM32 启动文件浅析》](#)。

9.3 map 文件分析

9.3.1 MDK 编译生成文件简介

MDK 编译工程，会生成一些中间文件（如.o、.axf、.map 等），最终生成 hex 文件，以便下载到 MCU 上面执行，以 STM32F103 开发板 HAL 库例程的实验 1 跑马灯实验为例（其他开发板类似），编译过程产生的所有文件，都存放在 Output 文件夹下，如图 9.3.1.1 所示：

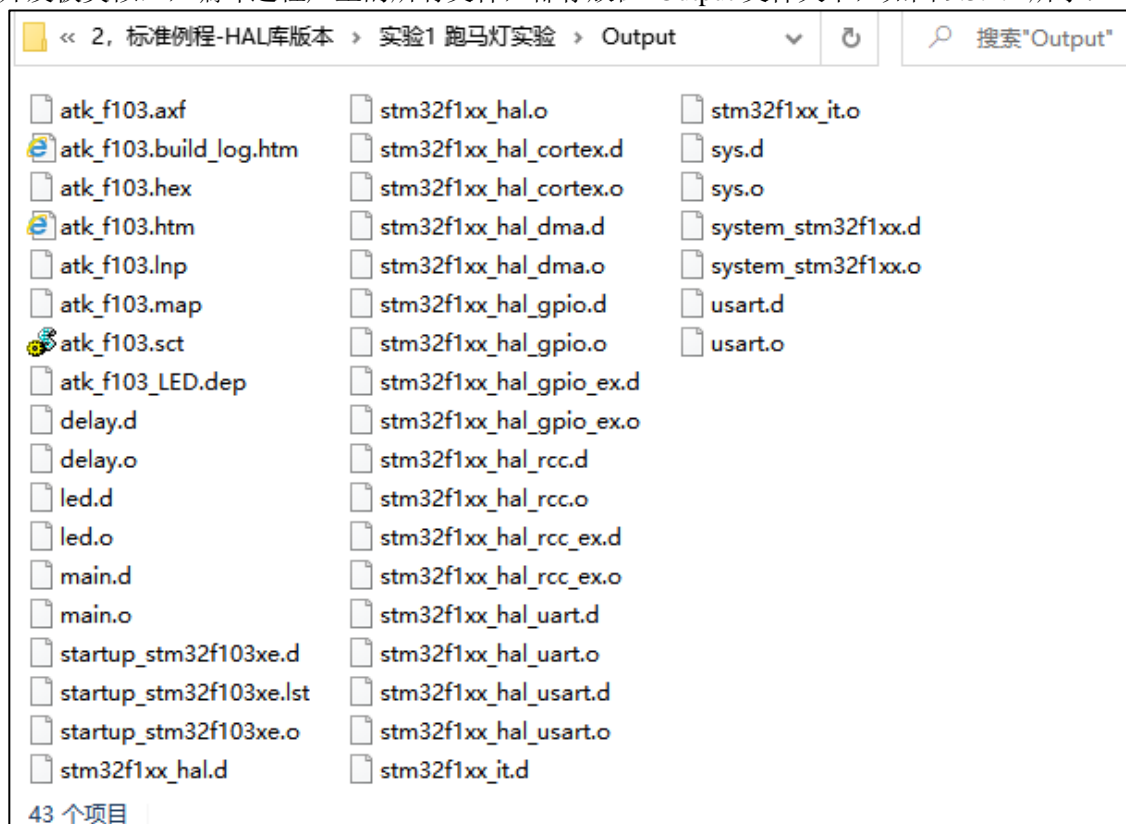


图 9.3.1.1 MDK 编译过程生成的文件

这里总共生成了 43 个文件，共 11 个类型，分别是：.axf、.crf、.d、.dep、.hex、.lnp、.lst、.o、.htm、bulild_log.htm 和.map。43 个文件（勾选 Browse informatio-n 时为 59 个）看着不是很多，但是随着工程的增大，这些文件也会越来越多，大项目编译一次，可以生成几百甚至上千个这种文件，不过文件类型基本就是上面这些。

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

| 文件类型 | 说明 |
|------|---|
| .o | 可重定向 1 对象文件，每个源文件（.c/.s 等）编译都会生成一个.o 文件 |
| .axf | 由 ARMCC 编译生产的可执行对象文件，不可重定向 2（绝对地址）多个.o 文件链接生成.axf 文件，我们在仿真的时候，需要用到该文件 |
| .hex | Intel Hex 格式文件，可用于下载到 MCU，.hex 文件由.axf 文件转换而来 |
| .crf | 交叉引用文件，包含浏览信息（定义、标识符、引用） |

| | |
|----------------|--|
| .d | 由 ARMCC/GCC 编译生产的依赖文件（.o 文件所对应的依赖文件） 每个.o 文件，都有一个对应的.d 文件 |
| .dep | 整个工程的依赖文件 |
| .lnp | MDK 生成的链接输入文件，用于命令输入 |
| .lst | C 语言或汇编编译器生成的列表文件 |
| .htm | 链接生成的列表文件 |
| .build_log.htm | 最近一次编译工程时的日志记录文件 |
| .map | 连接器生成的列表文件/MAP 文件， 该文件对我们非常有用 |

表 9.3.1.1 常见的中间文件类型说明

注 1，可重定向是指该文件包含数据/代码，但是并没有指定地址，它的地址可由后续链接的时候进行指定。

注 2，不可重定向是指该文件所包含的数据/代码都已经指定地址了，不能再改变。

9.3.2 map 文件分析

.map 文件是编译器链接时生成的一个文件，它主要包含了交叉链接信息。通过.map 文件，我们可以知道整个工程的函数调用关系、FLASH 和 RAM 占用情况及其详细汇总信息，能具体到单个源文件（.c/.s）的占用情况，根据这些信息，我们可以对代码进行优化。.map 文件可以分为以下 5 个组成部分：

- 1， 程序段交叉引用关系（Section Cross References）
- 2， 删除映像未使用的程序段（Removing Unused input sections from the image）
- 3， 映像符号表（Image Symbol Table）
- 4， 映像内存分布图（Memory Map of the image）
- 5， 映像组件大小（Image component sizes）

9.3.2.1 map 文件的 MDK 设置

要生成 map 文件，我们需要在 MDK 的魔术棒→Listing 选项卡里面，进行相关设置，如图 9.3.2.1.1 所示：

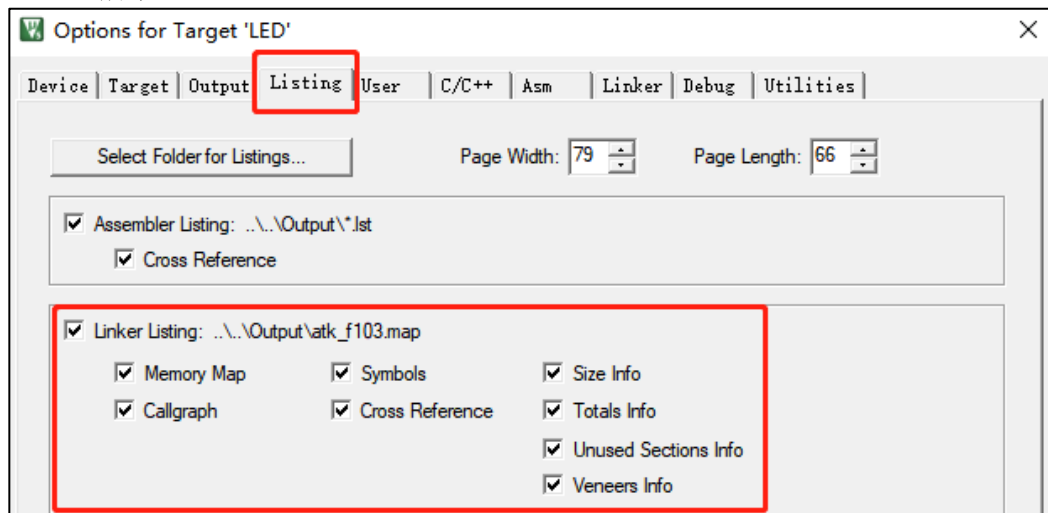


图 9.3.2.1.1 .map 文件生成设置

图 9.3.2.1.1 中红框框出的部分就是我们z需要设置的，默认情况下，MDK 这部分设置就是全勾选的，如果我们想取消掉一些信息的输出，则取消相关勾选即可（一般不建议）。

如图 9.3.2.1.1 设置好 MDK 以后，我全编译当前工程，当编译完成后（无错误），就会生成.map 文件。在 MDK 里面打开.map 文件的方法如图 9.3.2.1.2 所示：

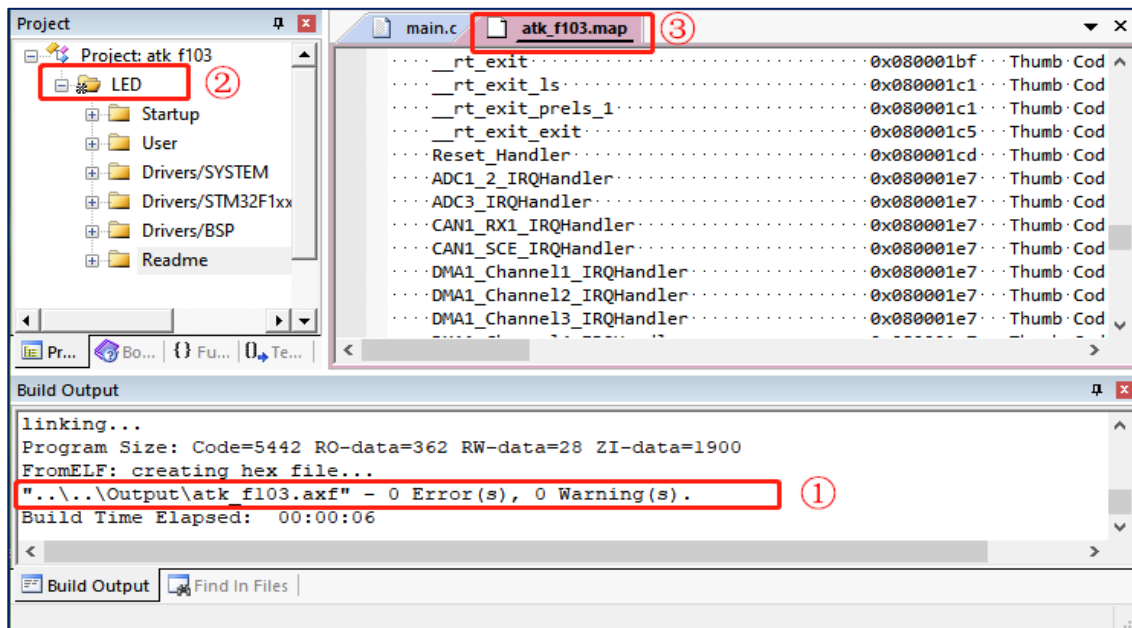


图 9.3.2.1.2 打开.map 文件

- 1, 先确保工程编译成功（无错误）。
- 2, 双击 LED, 打开.map 文件。
- 3, map 文件打开成功。

9.3.2.2 map 文件的基础概念

为了更好的分析 map 文件，我们先对需要用到的一些基础概念进行一个简单介绍，相关概念如下：

- Section: 描述映像文件的代码或数据块，我们简称程序段
- RO: Read Only 的缩写，包括只读数据（RO data）和代码（RO code）两部分内容，占用 FLASH 空间
- RW: Read Write 的缩写，包含可读写数据（RW data，有初值，且不为 0），占用 FLASH（存储初值）和 RAM（读写操作）
- ZI: Zero initialized 的缩写，包含初始化为 0 的数据（ZI data），占用 RAM 空间。
- .text: 相当于 RO code
- .constdata: 相当于 RO data
- .bss: 相当于 ZI data
- .data: 相当于 RW data

9.3.2.3 map 文件的组成部分说明

我们前面说 map 文件分为 5 个部分组成，下面以 STM32F103 开发板 HAL 库例程的实验 1 跑马灯实验为例，简要讲解一下。

1. 程序段交叉引用关系 (S S section Cross References)

这部分内容描述了各个文件（.c/.s 等）之间函数（程序段）的调用关系，举个例子如图 9.3.2.3.1 所示：

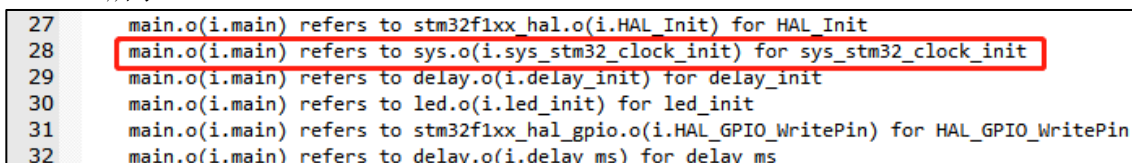


图 9.3.2.3.1 程序段交叉引用关系图

上图中，框出部分：main.o(i.main) refers to sys.o(i.sys_stm32_clock_init) for sys_stm32_clock_init 表示：main.c 文件中的 main 函数，调用了 sys.c 中的 sys_stm32_clock_init 函数。其中：i.main 表示 main 函数的入口地址，同理 i.sys_stm32_clock_init 表示 sys_stm32_clock_init 函数的入口地址。

2. 删除映像未使用的程序段 (Removing Unused input sections from the image)

这部分内容描述了工程中由于未被调用而被删除的冗余程序段（函数/数据），如图 9.3.2.3.2 所示：

```

364 Removing usart.o(.rev16_text), (4 bytes).
365 Removing usart.o(.revsh_text), (4 bytes).
366 Removing usart.o(.rrx_text), (6 bytes).
367 Removing usart.o(i.HAL_UART_MspInit), (180 bytes).
368 Removing usart.o(i._ttywrch), (4 bytes).
369 Removing usart.o(i.fputc), (28 bytes).
370 Removing usart.o(i.usart_init), (56 bytes).

557 216 unused section(s) (total 15556 bytes) removed from the image.
558
559 =====
    
```

图 9.3.2.3.2 删除未用到的程序段

上图中，列出了所有被移除的程序段，比如 usart.c 里面的 usart_init 函数就被移除了，因为该例程没用到 usart_init 函数。

另外，在最后还有一个统计信息：216 unused section(s) (total 15556bytes) removed from the image.表示总共移除了 216 个程序段（函数/数据），大小为 15556 字节。即给我们的 MCU 节省了 15556 字节的程序空间。

为了更好的节省空间，我们一般在 MDK→魔术棒→C/C++选项卡里面勾选：One ELF Section per Function，如图 9.3.2.3.3 所示：

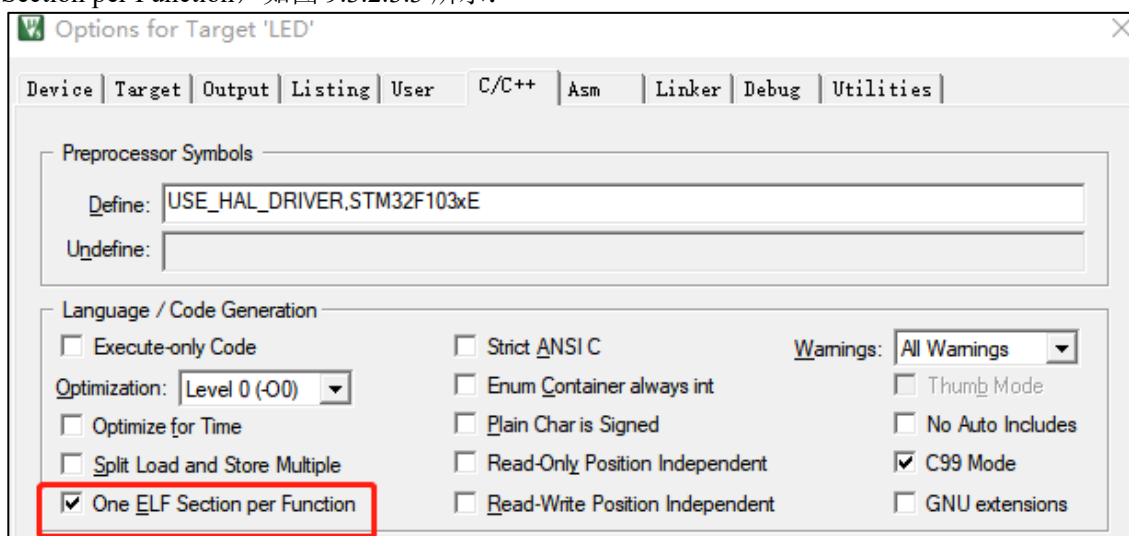


图 9.3.2.3.3 MDK 勾选 One ELF Section per Function

3. 映像符号表 (Image Symbol Table)

映像符号表 (Image Symbol Table) 描述了被引用的各个符号（程序段/数据）在存储器中的存储地址、类型、大小等信息。映像符号表分为两类：本地符号 (Local Symbols) 和全局符号 (Global Symbols)。

本地符号 (Local Symbols) 记录了用 static 声明的全局变量地址和大小，c 文件中函数的地址和用 static 声明的函数代码大小，汇编文件中的标号地址（作用域：限本文件）。

全局符号 (Global Symbols) 记录了全局变量的地址和大小，C 文件中函数的地址及其代码大小，汇编文件中的标号地址（作用域：全工程）。

4. 映像内存分布图 (Memory Map of the image)

映像文件分为加载域 (Load Region) 和运行域 (Execution Region)。一个加载域必须有

至少一个运行域（可以有多个运行域），而一个程序又可以有多个加载域。加载域为映像程序的实际存储区域，而运行域则是 MCU 上电后的运行状态。加载域和运行域的简化关系（这里仅表示一个加载域的情况）图，如图 9.3.2.3.4 所示：

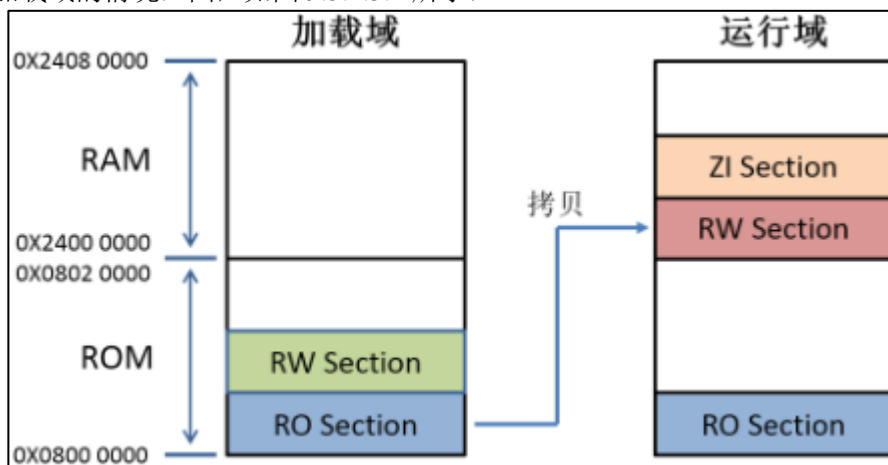


图 9.3.2.3.4 加载域运行域关系

由图可知，RW 区也是存放在 ROM（FLASH）里面的，在执行 main 函数之前，RW（有初值且不为 0 的变量）数据会被拷贝到 RAM 区，同时还会在 RAM 里面创建 ZI 区（初始化为 0 的变量）。

5. 映像组件大小（Image component sizes）

映像组件大小（Image component sizes）给出了整个映像所有代码（.o）占用空间的汇总信息。这部分是程序实际功能可执行代码的存储空间。

由于篇幅较长，更多内容请大家查阅《STM32 MAP 文件浅析》文档的内容。

第十章 STM32CubeMX 简介

STM32CubeMX 是由 ST 公司开发的图形化代码自动生成工具,能够快速生成初始化代码,如配置 GPIO, 时钟树, 中间件等,使用户专注于业务代码的开发。现在 ST 主推 HAL 库代码,经典的标准外设库已经停止维护了,新产品也只提供 HAL 库的代码,因此,我们学习 HAL 库是更加有优势的,由于 HAL 库具有低耦合、通用、抽象了硬件层,使得开发者无需太过关注硬件驱动的实现,使得开发更加的简单快速,更容易维护,因此被越来越多的产品所使用。

本章将分为如下几个小节:

10.1 STM32CubeMX 的作用

10.2 安装 STM32CubeMX

10.3 使用 STM32CubeMX 新建工程

10.4 STM32CubeMX 新建工程使用建议

10.1 STM32CubeMX 的作用

STM32CubeMX 具有如下特性:

- ① 直观的选择 MCU 型号,可指定系列、封装、外设数量等条件;
- ② 微控制器图形化配置;
- ③ 自动处理引脚冲突;
- ④ 动态设置时钟树,生成系统时钟配置代码;
- ⑤ 可以动态设置外围和中间件模式和初始化;
- ⑥ 功耗预测;
- ⑦ C 代码工程生成器覆盖了 STM32 微控制器初始化编译软件,如 IAR, KEIL, GCC;
- ⑧ 可以独立使用或者作为 Eclipse 插件使用;
- ⑨ 可作为 ST 的固件包、芯片手册等的下载引擎;

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

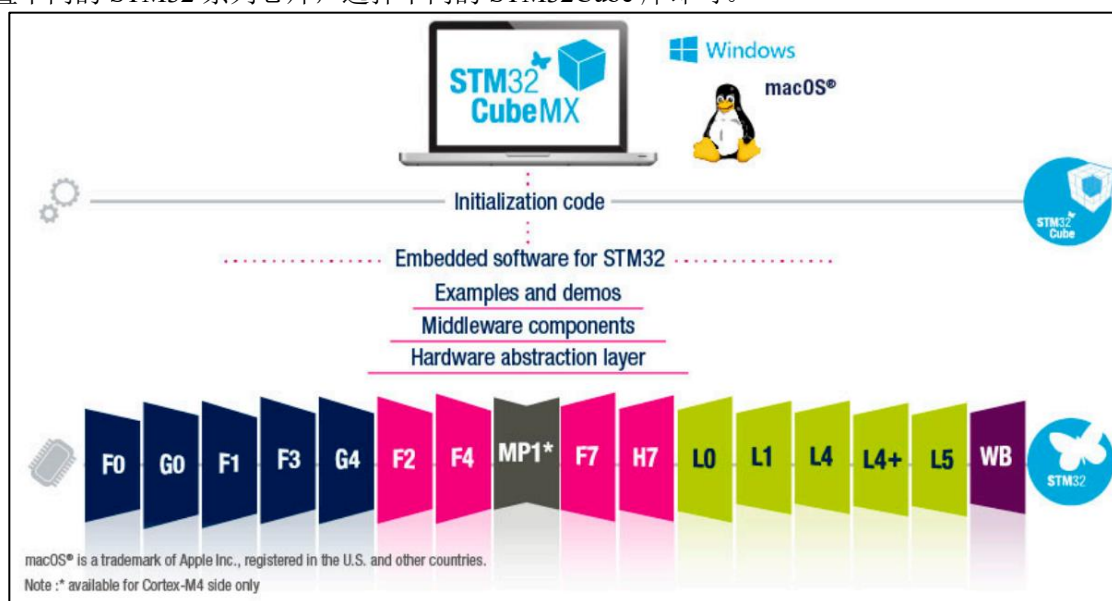


图 10.1.1 STM32CubeMX 和 STM32Cube 库的关系

当然，自动生成的驱动代码我们不去仔细专研其原理的话，对学习的提升很有限，而且在出现 BUG 的时候难以快速定位解决，因此我们也要了解其背后的原理。

10.2 安装 STM32CubeMX

STM32CubeMX 运行环境搭建包含两个部分。首先是 Java 运行环境安装，其次是 STM32CubeMX 软件安装。

10.2.1 安装 JAVA 环境

对于 Java 运行环境，大家可以到 Java 官网 www.java.com 下载最新的 Java 软件，也可以直接从我们光盘复制安装包，目录为：**A 盘→6，软件资料→1，软件→3、STM32CubeMX→Java 安装包**，Java 安装包文件下有 x64 和 x86 两个文件夹，分别是 64 位和 32 位的电脑的安装包，大家根据自己电脑的位数选择即可。比如 64 位电脑选择 x64 文件夹的 jre-8u301-windows-x64.exe 安装包，并根据提示安装即可。安装完成之后提示界面如下图 10.2.1.1 所示。



图 10.2.1.1 Java 安装成功提示界面

安装完 Java 运行环境之后，为了检测是否正常安装，我们可以打开 Windows 的命令输入框，输入：`java -version` 命令，如果显示 Java 版本信息，则安装成功。提示信息如下图 10.2.1.2：



图 10.2.1.2 查看 Java 版本

10.2.2 安装 STM32CubeMX

在安装了 Java 运行环境之后，接下来我们安装 STM32CubeMX 图形化工具。该软件可以直接从光盘复制，目录为：**A 盘→6，软件资料→1，软件→STM32CubeMX**，也可以直接从 ST 官方下载，下载地址为：<https://www.st.com/en/development-tools/stm32cubemx.html>。

接下来我们直接双击 **SetupSTM32CubeMX-6.3.0.exe**，安装步骤如下。

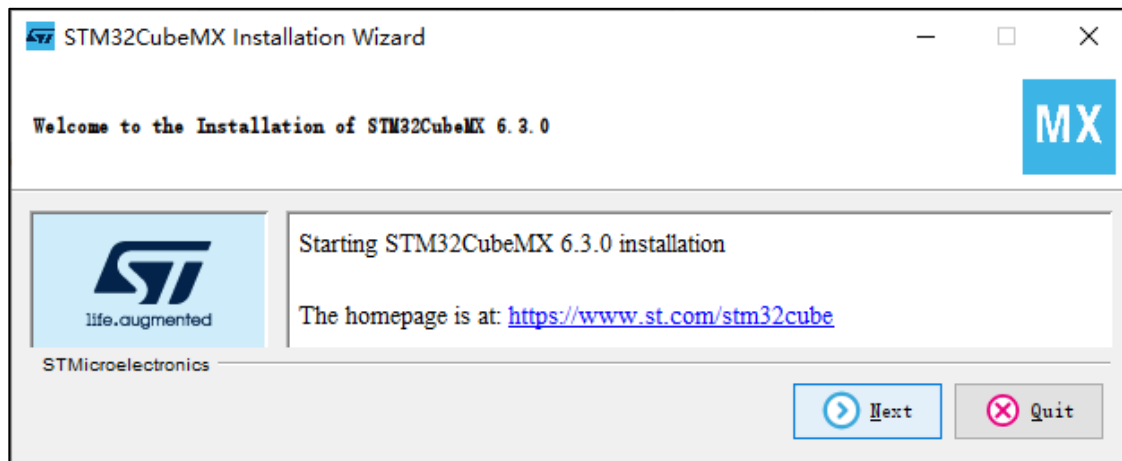


图 10.2.2.1 启动安装

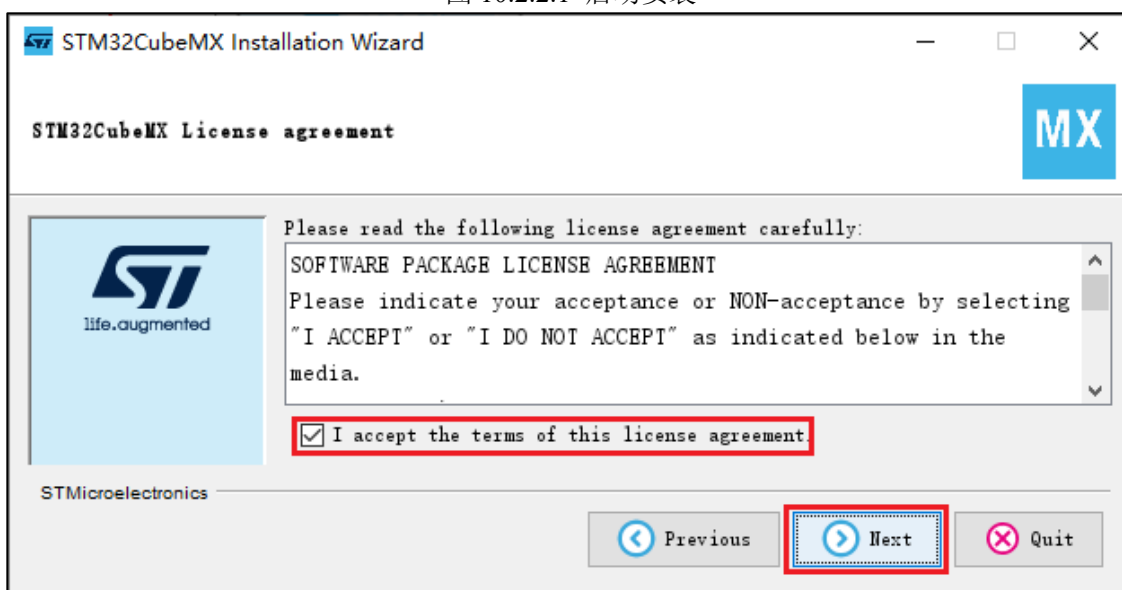


图 10.2.2.2 接受本许可协议

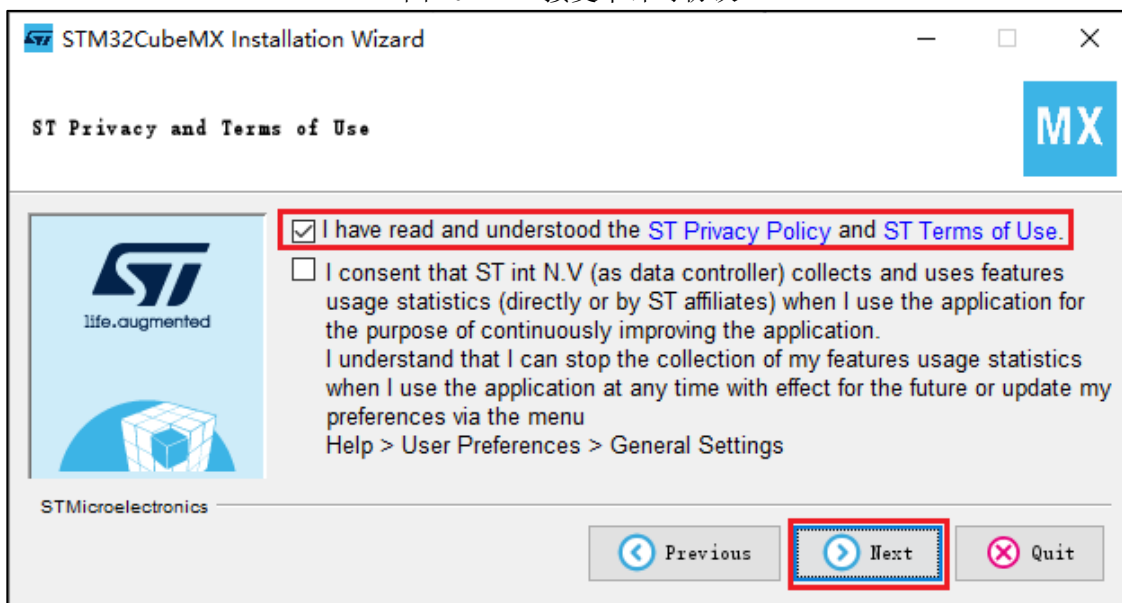


图 10.2.2.3 勾选第一项即可

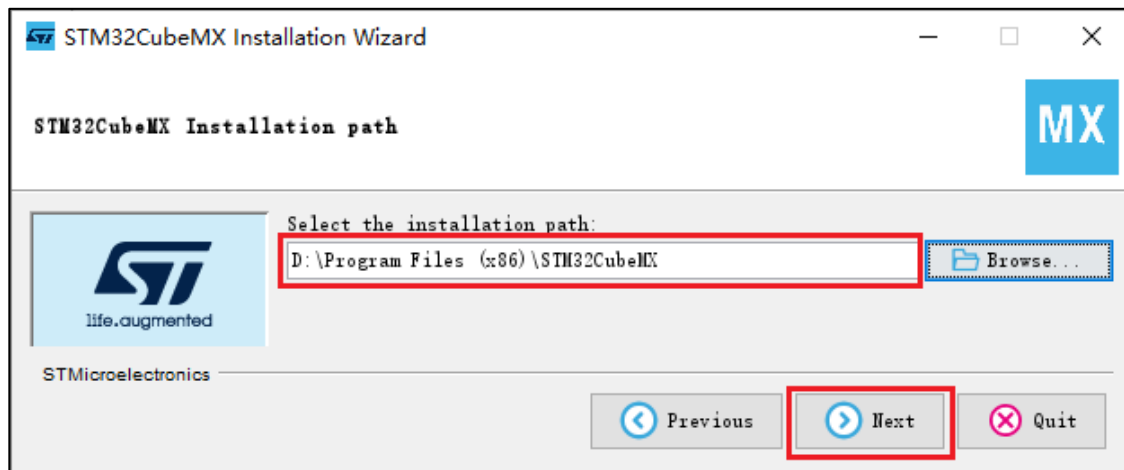


图 10.2.2.4 指定安装路径

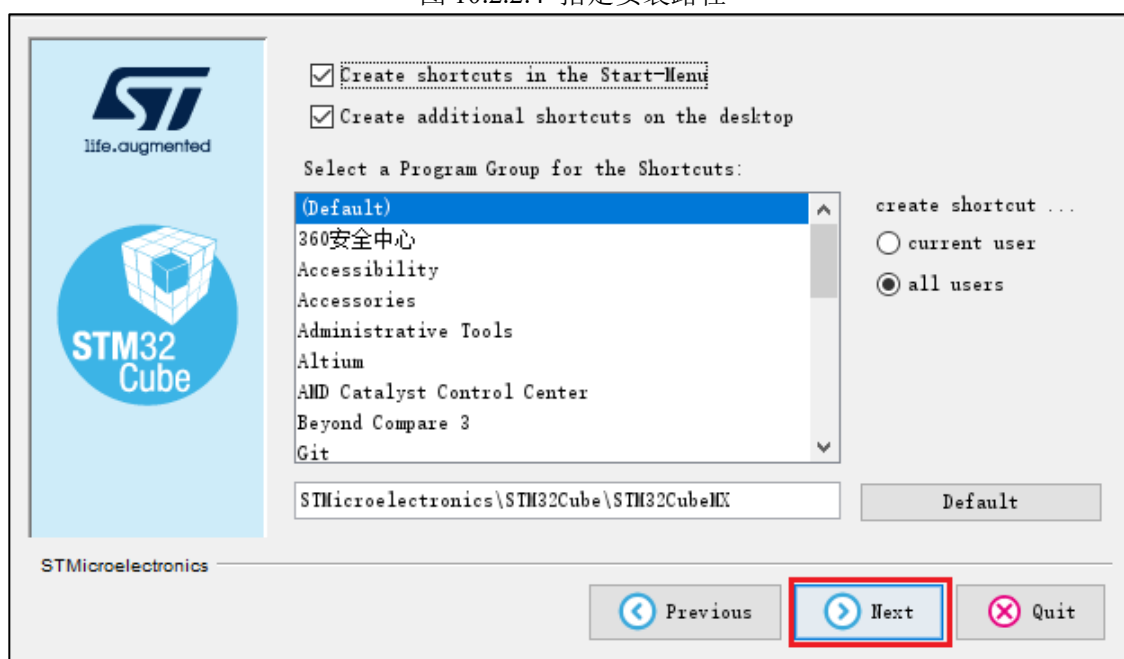


图 10.2.2.5 创建快捷方式

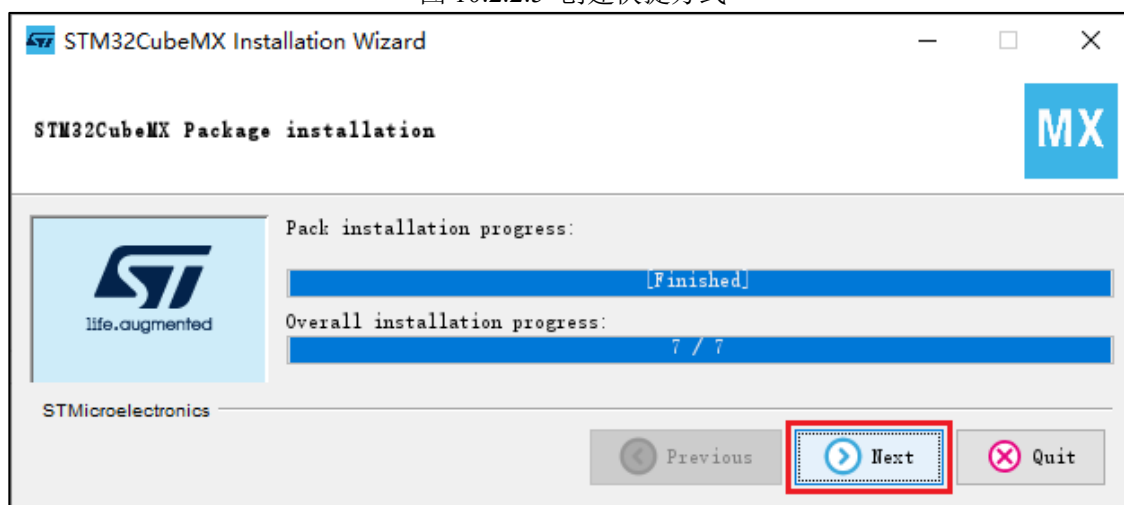


图 10.2.2.5 安装进度提示

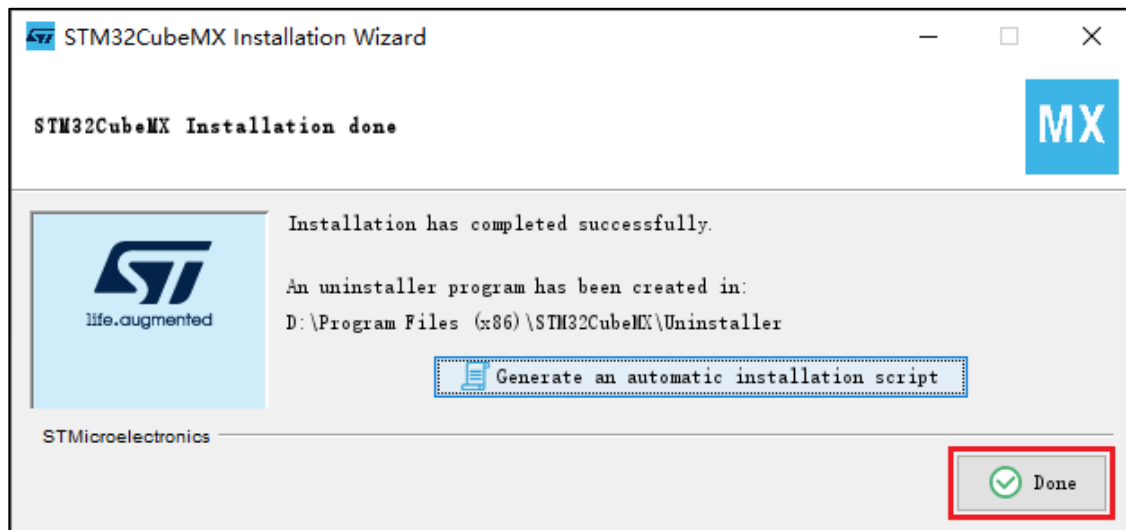


图 10.2.2.6 完成安装

10.3 使用 STM32CubeMX 新建工程

CubeMX 建立的工程结构和本书介绍的代码风格有所差异，限于篇幅，本部分只介绍如何使用 CubeMX 生成 MDK 工程的方法。

10.3.1 打开 STM32CubeMX

双击如图 10.3.1.1 所示的 STM32CubeMX 桌面快捷方式图标，打开后 CubeMX 主界面如图 10.3.1.2 所示。



图 10.3.1.1 CubeMX 快捷方式

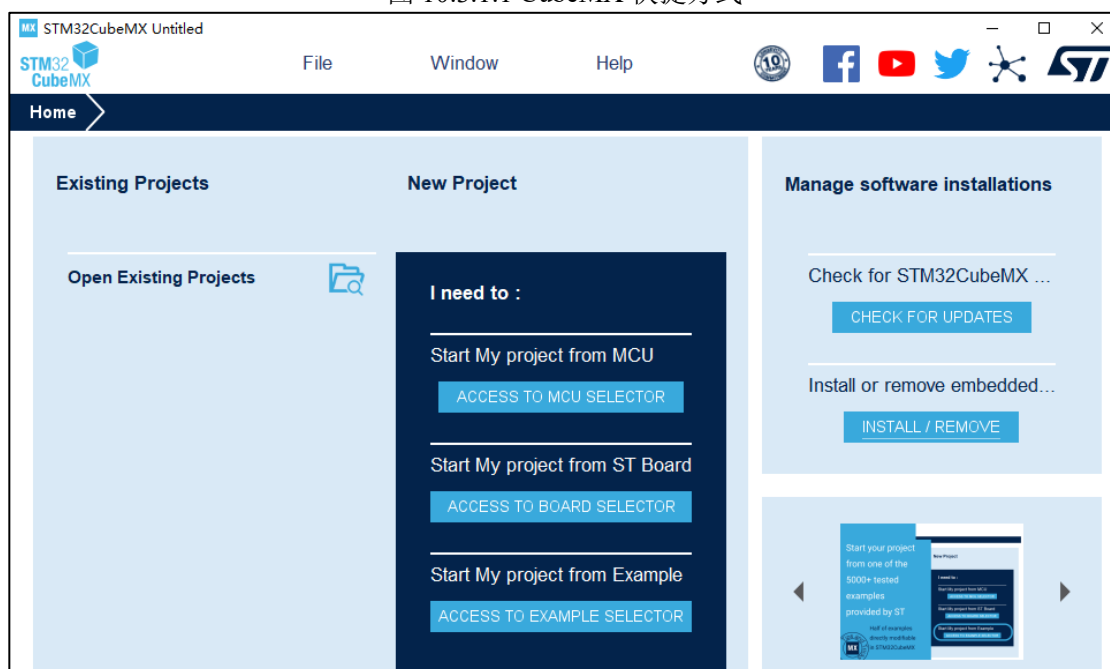


图 10.3.1.2 CubeMX 主界面

10.3.2 下载和关联的 STM32Cube 固件包

新建工程前，我们需要先安装关联与 STM32 主芯片对应的 STM32Cube 固件包，点击 Help->Manage embedded software packages，如图 10.3.2.1 所示。

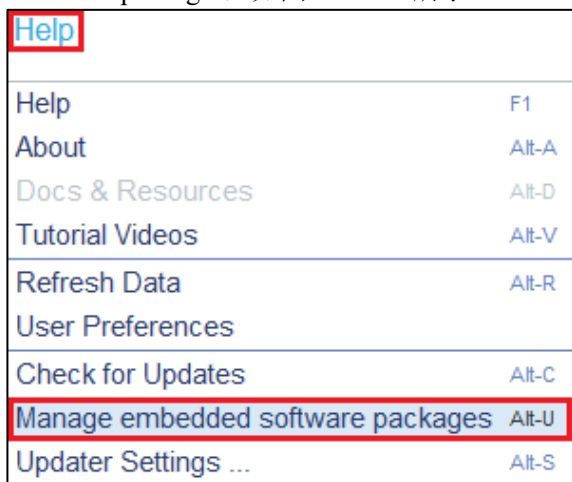


图 10.3.2.1 管理固件包

在弹出的软件包管理界面中，我们可以选择安装驱动包的方式，有以下两种方法：

方式一：从网络下载安装，按照图 10.3.2.2 的步骤，在该窗口找到 STM32F1 列表选项，因为我们的教程源码使用的固件包是 1.8.3 版本的，所以我们勾选 1.8.3 版本，等待安装完成即可。

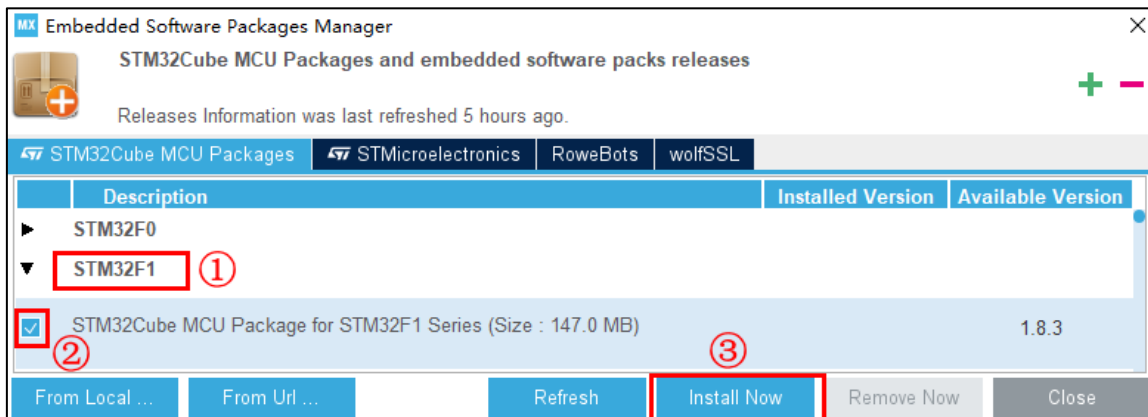


图 10.3.2.2 下载和关联 STM32Cube 固件包

方法二：不通过网络，直接点击从本地导入。由于直接使用上面的安装包管理器的“From Local”选项导入压缩包有时候会直接报错，比如可以直接导入“stm32cube_fw_f1_v180.zip”固件包，但直接导入“stm32cube_fw_f1_v183.zip”的安装包时 CubeMX 软件会报错，所以我们采用以下方法来处理：

- ① 选择 CubeMX 的菜单，Help->Updater Settings，或者在 CubeMX 的活动窗口处于前台运行状态时使用它推荐的快捷键“Alt+S”，可以在弹出“更新设置”界面下，找到“Repository Folder”，即 CubeMX 的资源仓库，默认是安装在 C 盘的一个文件夹下，但由于 CubeMX 会把固件库解压到这个文件夹下，在我们使用的 STM32 型号多了或者经过一段时间的更新使用后，这个文件夹常常会变得非常大（数十 Gb）。所以我们这里把默认的路径设置为“D:\STM32Cube\Repository”，但使用自定义路径时 CubeMX 关联的路径一定不能出现中文出现，大家根据自己的需要决定是否设置。操作方法如图 10.3.2.3 和图 10.3.2.4 所示。

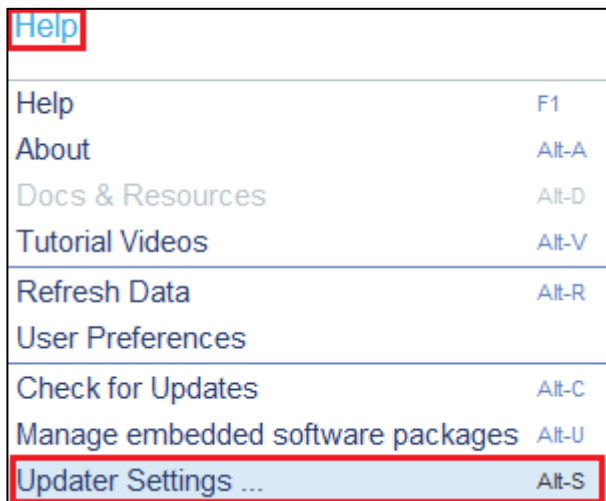


图 10.3.2.3 设置更新选项

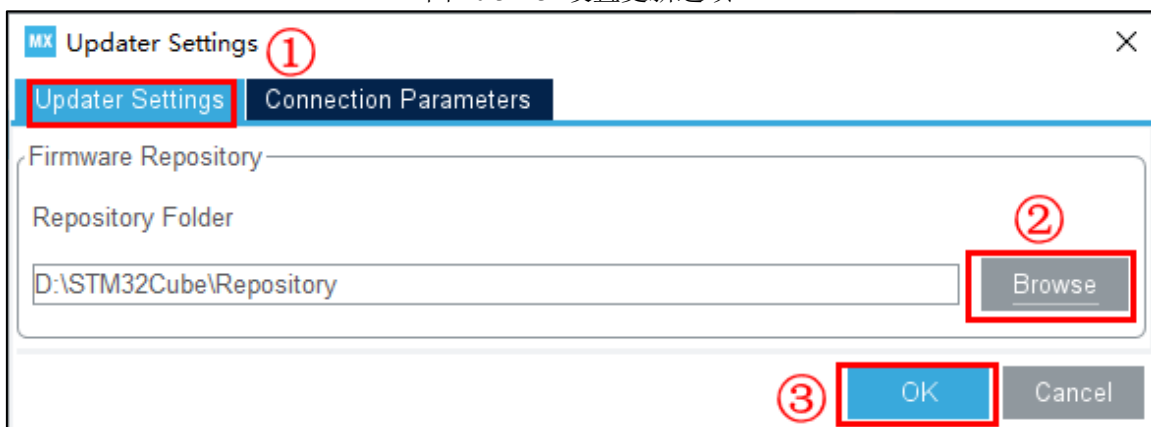


图 10.3.2.4 修改 CubeMX 的固件库路径（非必须）

- ② 我们复制光盘提供的固件包，把它放到上一步我们设置或者找到的 CubeMX 的仓库文件夹。光盘资料路径为：**A 盘→8, STM32 参考资料→1, STM32CubeF1 固件包**，我们这里要使用的是 CubeF1 固件的 1.8.3 版本。由于 CubeF1 的 1.8.3 版本是 1.8.0 版本的补充包，所以需要把“stm32cube_fw_fl_v180.zip”和“stm32cube_fw_fl_v183.zip”两个固件包都复制到对应的路径下。复制后目录 CubeMX 的仓库目录的状况如图 10.3.2.4 所示。



图 10.3.2.4 复制固件包到 CubeMX 对应的仓库路径

- ③ 先解压“stm32cube_fw_fl_v180.zip”到当前仓库路径的根路径，注意这里解压缩后的第一级目录就是压缩包下的目录，如图 10.3.2.5 所示。

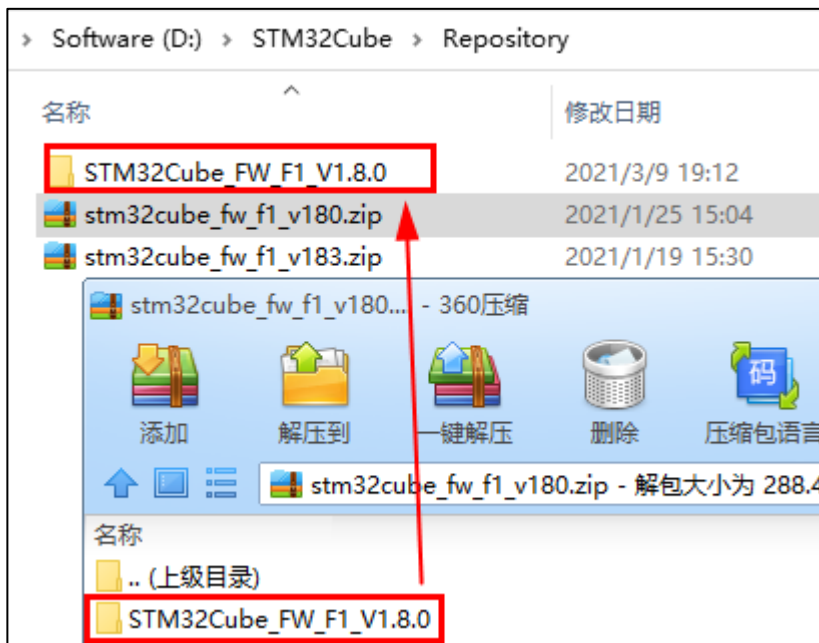


图 10.3.2.5 先解压 “stm32cube_fw_f1_v180.zip”

- ④ 接着解压 “stm32cube_fw_f1_v183.zip” 到当前仓库路径的根路径，这里打开压缩包可以发现 CubeF1 的 1.8.3 版本固件压缩包下的文件夹与 1.8.0 的完全相同，我们解压缩并替换文件为 1.8.3 中的文件。解压后路径下仍只有一个 “STM32Cube_FW_F1_V1.8.0” 文件夹。

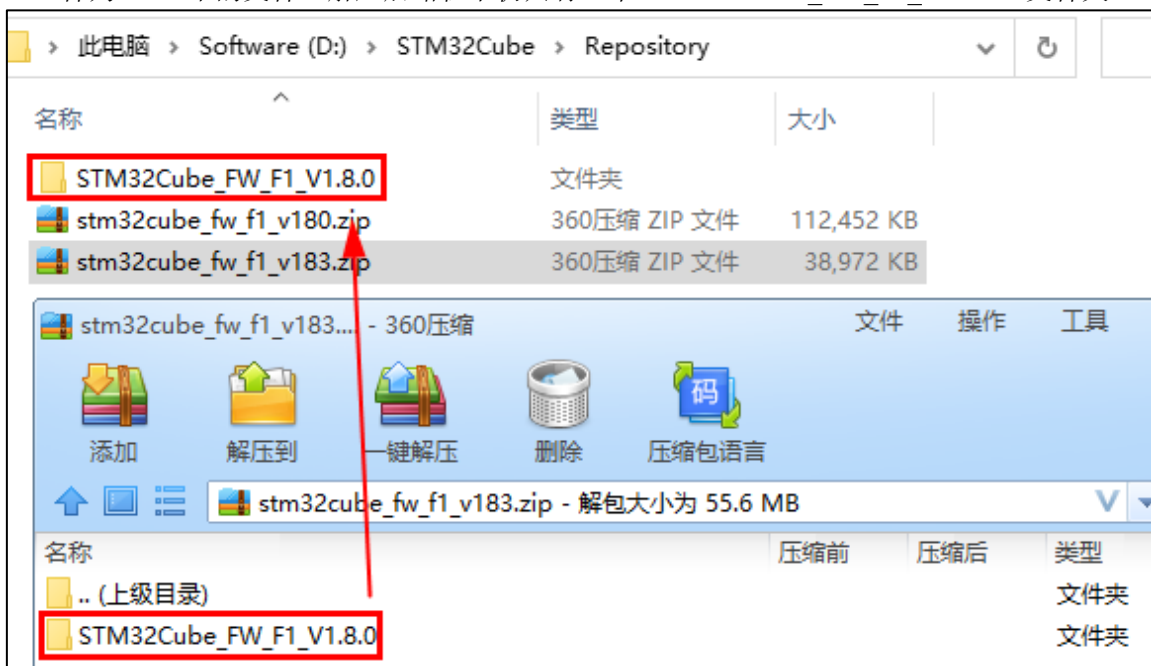


图 10.3.2.6 解压 “stm32cube_fw_f1_v183.zip”，并替换原有文件

- ⑤ 关闭 CubeMX 软件并重启。重启后通过 CubeMX 的安装包管理器，可以发现 CubeMX 已经检测到安装好了对应的驱动库。至此离线安装步骤结束。

| | Description | Installed Version | Available Version |
|-------------------------------------|--|-------------------|-------------------|
| ▼ | STM32F1 | | |
| <input checked="" type="checkbox"/> | STM32Cube MCU Package for STM32F1 Series | 1.8.3 | 1.8.3 |
| <input type="checkbox"/> | STM32Cube MCU Package for STM32F1 Series (Size : 147.0 MB) | | 1.8.2 |

图 10.3.2.7 成功离线安装 CubeF1 的 1.8.3 版本固件库

10.3.3 新建工程

通过上一步安装固件库后，我们就可以使用 STM32CubeMX 配置工程，步骤如下：

1. 工程初步建立
2. HSE 和 LSE 时钟源设置
3. 时钟系统（时钟树）配置
4. GPIO 功能引脚配置
5. 配置 Debug 选项
6. 生成工程源码
7. 用户程序

接下来我们将按照上述步骤，和大家一起使用 STM32CubeMX 工具生成一个 MDK 工程，如果之前还没有安装过 MDK 的，请先按教程关于 MDK 安装和介绍部分的内容先安装好软件。

1 工程初步建立

方法一：依次点击“File->New Project”即可建新工程。如果之前打开过的话，左侧最近打开的过程一列会有打开的工程列表，直接点击这些工程也可以打开。

方法二：直接点击 ACCESS TO MCU SELECTOR。

具体操作如图 10.3.3.1 所示。

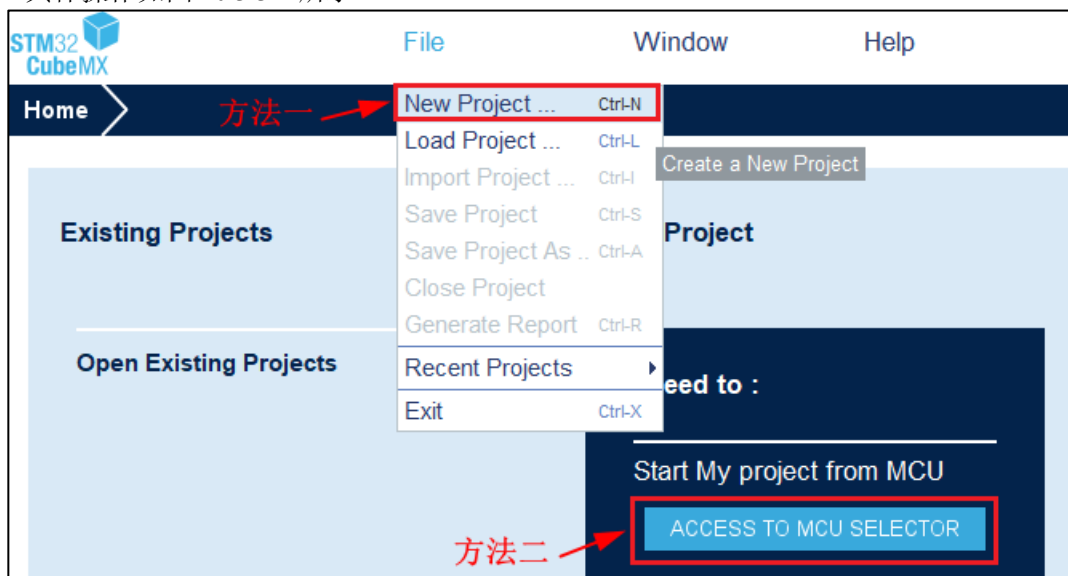


图 10.3.3.1 新建工程

点击新建工程后，可能会弹出如图 10.3.3.2 的窗口，提示需要联网下载一些文件，可能等待时间比较长，可以直接选择取消即可。我们可以通过关闭自动更新设置来禁止弹出这个窗口。

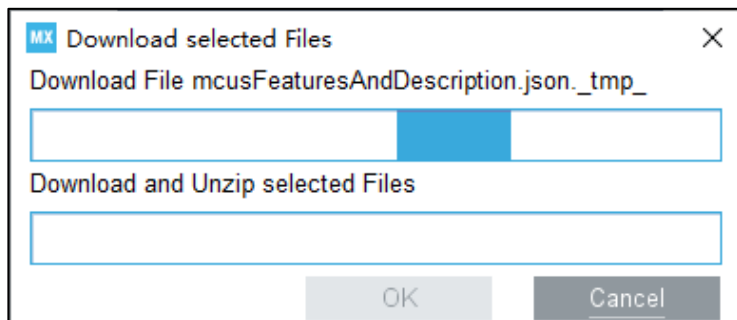


图 10.3.3.2 启动时联网更新检测
之后都可以进入芯片选型界面，如图 10.3.3.3 所示。

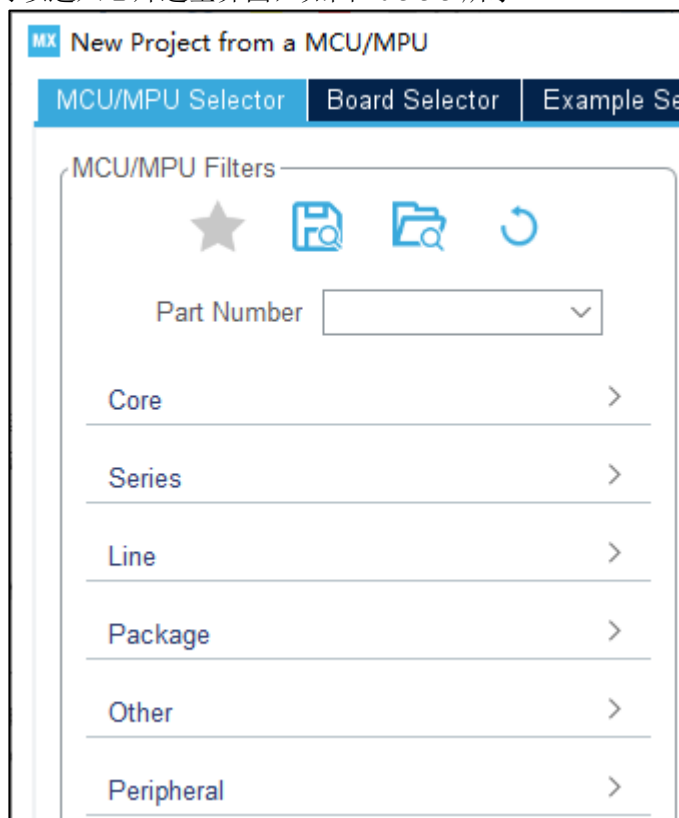


图 10.3.3.3 芯片选型界面
选择具体的芯片型号，如图 10.3.3.4 所示。



图 10.3.3.4 选择具体的芯片型号

鼠标双击选择的芯片型号后，弹出主设计界面，如图 10.3.3.5 所示。

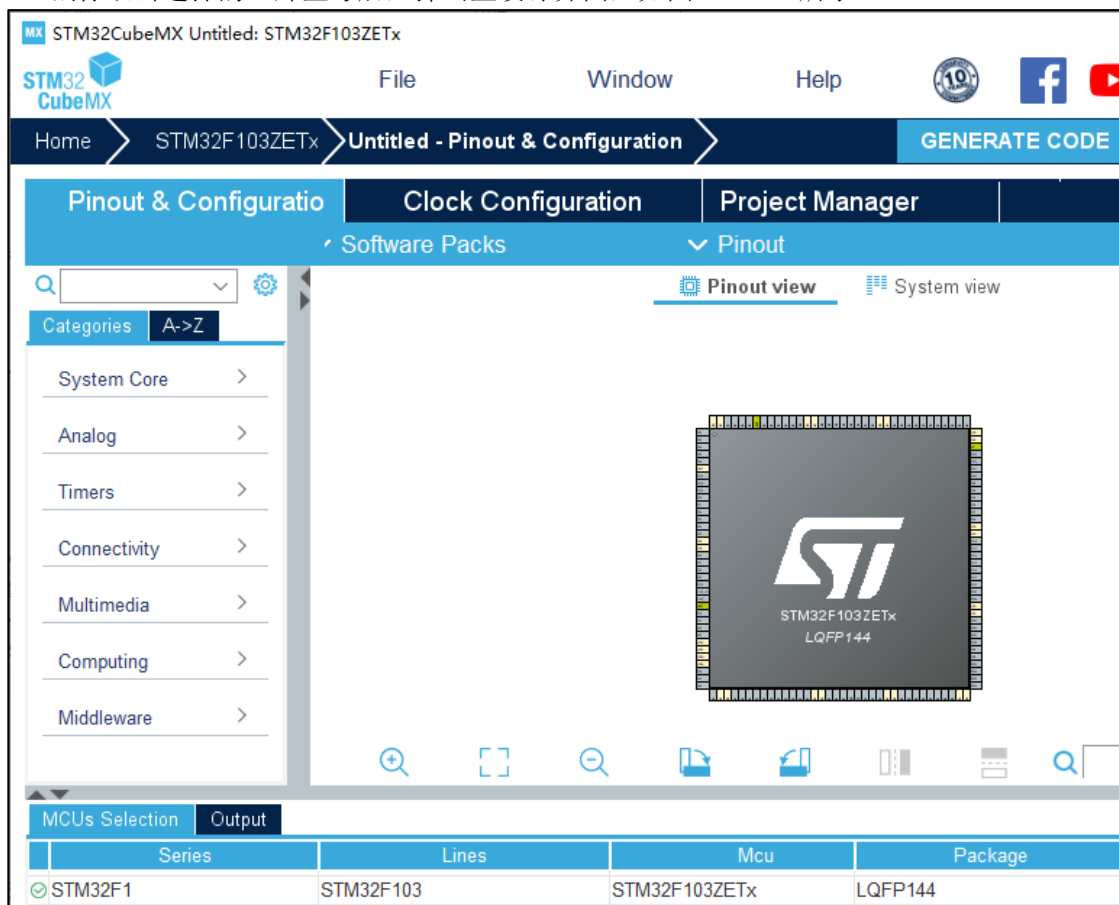


图 10.3.3.5 主设计界面

2 HSE 和 LSE 时钟源设置

进入工程主设计界面后，首先设置时钟源 HSE 和 LSE。如图 10.3.3.6 所示。

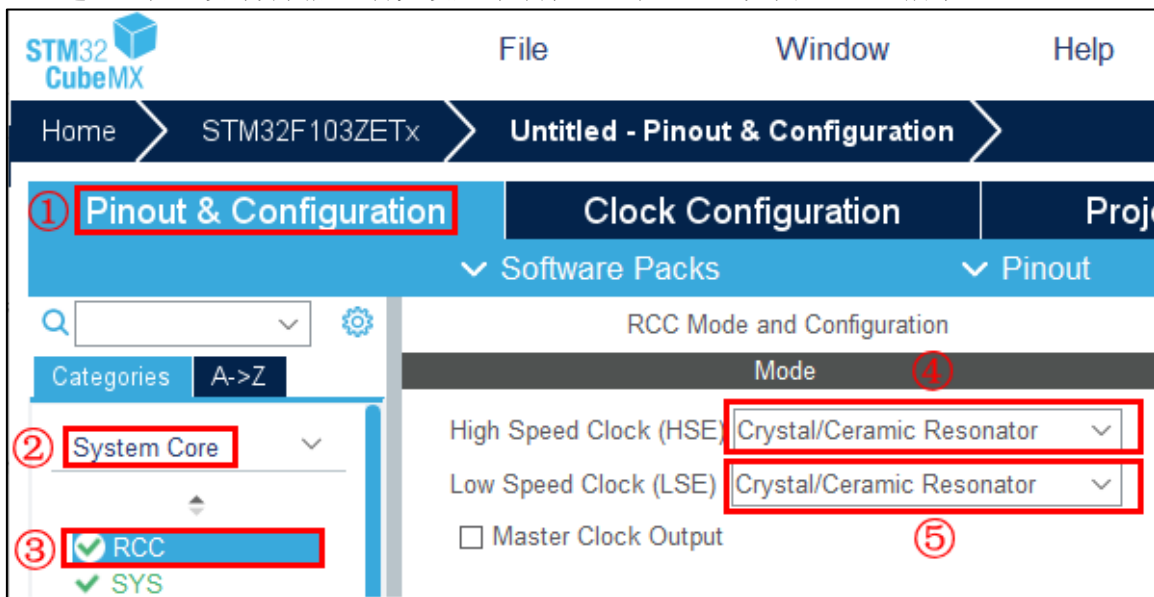


图 10.3.3.6 设置时钟源 HSE 和 LSE

图 10.3.3.6 中的标号④和⑤，我们都选择了 Crystal/Ceramic Resonator，表示外部晶振作为它们的时钟源。我们开发板的外部高速晶振和外部低速晶振分别是：8MHZ 和 32.768KHZ，所

以 HSE 时钟频率就是 8MHZ，LSE 时钟频率就是 32.768KHZ。

选项 Master Clock Output 1 用来选择是否使能 MCO1 引脚时钟输出。

3 时钟系统（时钟树）配置

点击 Clock Configuration 选项卡即可进入时钟系统配置栏，如下图 10.3.3.7 所示：

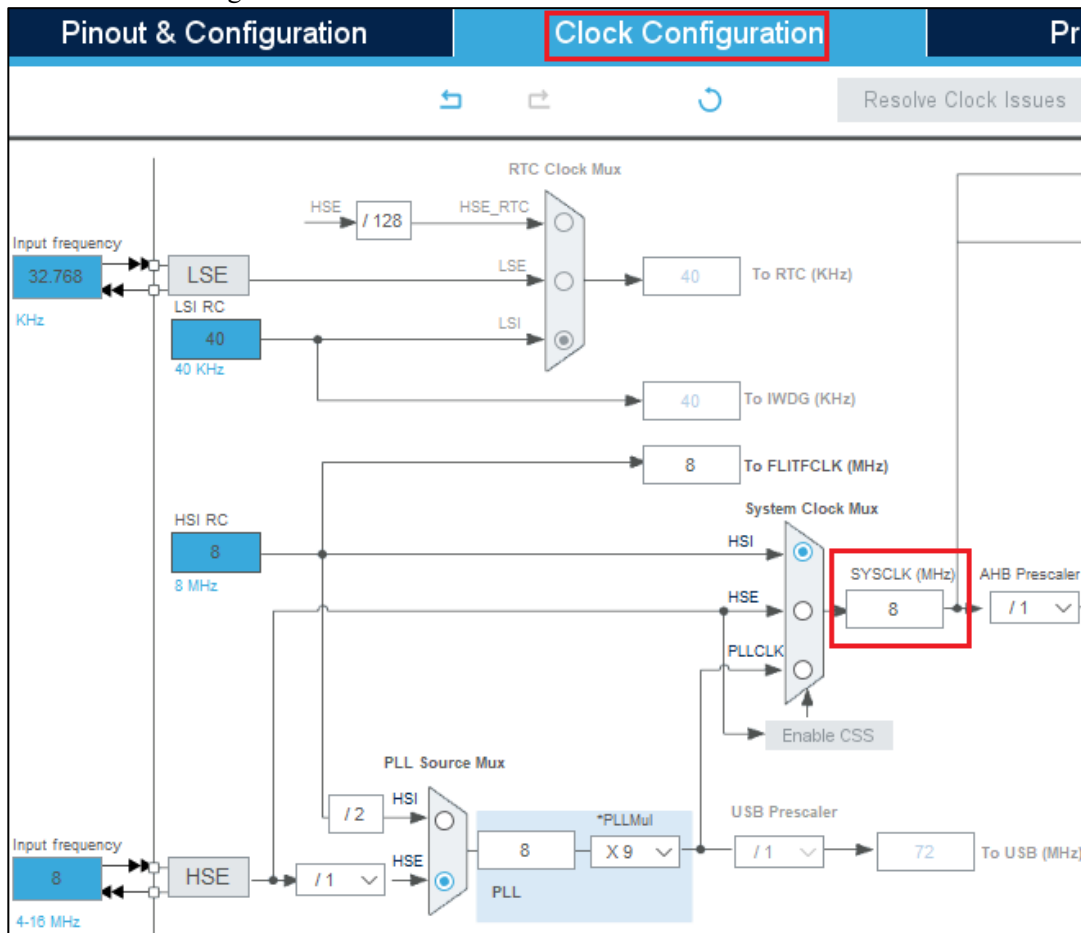


图 10.3.3.7 时钟系统配置栏

进入 Clock Configuration 配置栏之后可以看到，界面展现一个完整的 STM32F1 时钟系统框图。从这个时钟树配置图可以看出，配置的主要是外部晶振大小，分频系数，倍频系数以及选择器。在我们配置的工程中，时钟值会动态更新，如果某个时钟值在配置过程中超过允许值，那么相应的选项框会红色提示。

这里，我们将配置一个以 HSE 为时钟源，配置 PLL 相关参数，然后系统时钟选择 PLLCLK 为时钟源，最终配置系统时钟为 72MHz 的过程。同时，还配置了 AHB，APB1，APB 和 SysTick 的相关分频系数。由于图片比较大，我们把主要的配置部分分两部分来讲解，第一部分是配置系统时钟，第二部分是配置 SYSTICK、AHB、APB1 和 APB2 的分频系数。首先我们来看看第一部分配置如下图 10.3.3.8 所示：

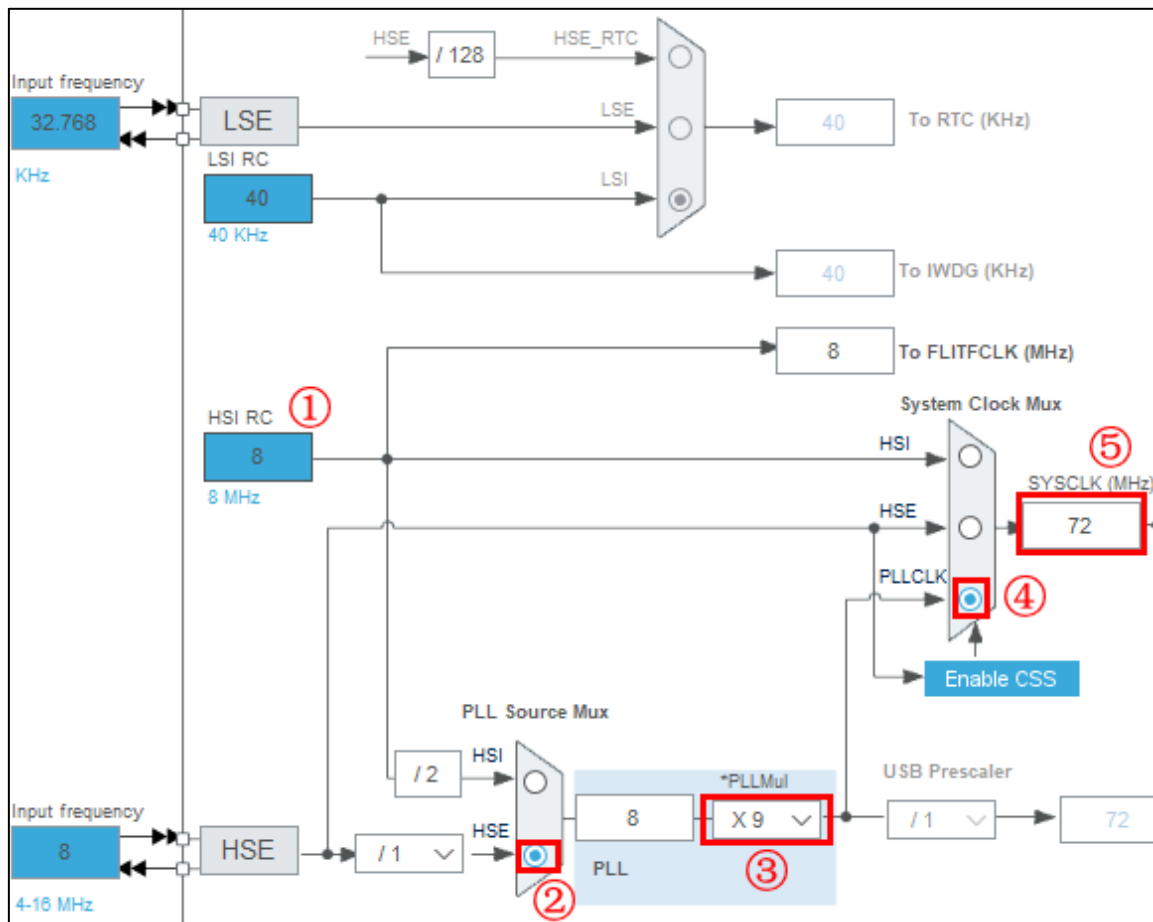


图 10.3.3.8 系统时钟配置图

我们把系统时钟配置分为七个步骤，分别用标号 1~5 表示，详细过程为：

- ① 时钟源参数设置：我们选择 HSE 为时钟源，所以我们要根据硬件实际的高速晶振频率（这里我们是 8MHz）填写。
- ② 时钟源选择：我们配置选择器选择 HSE 即可。
- ③ PLL 倍频系数 PLLMUL 配置。倍频系数 PLLMUL 我们设置为 9。
- ④ 系统时钟时钟源选择：PLL, HSI 还是 HSE。我们选择 PLL，选择器选择 PLLCLK 即可。
- ⑤ 经过上面配置以后此时 SYSCLK=72MHz。

经过上面的 5 个步骤，就配置好 STM32F1 的系统时钟为 72MHz。接下来我们还需要配置 AHB、APB1、APB2 和 SysTick 的分频系数，为 STM32 的片上外设或 M3 内核设置对应的工作时钟，为后续使用这些硬件功能做好准备。配置如下图 10.3.3.9 所示：

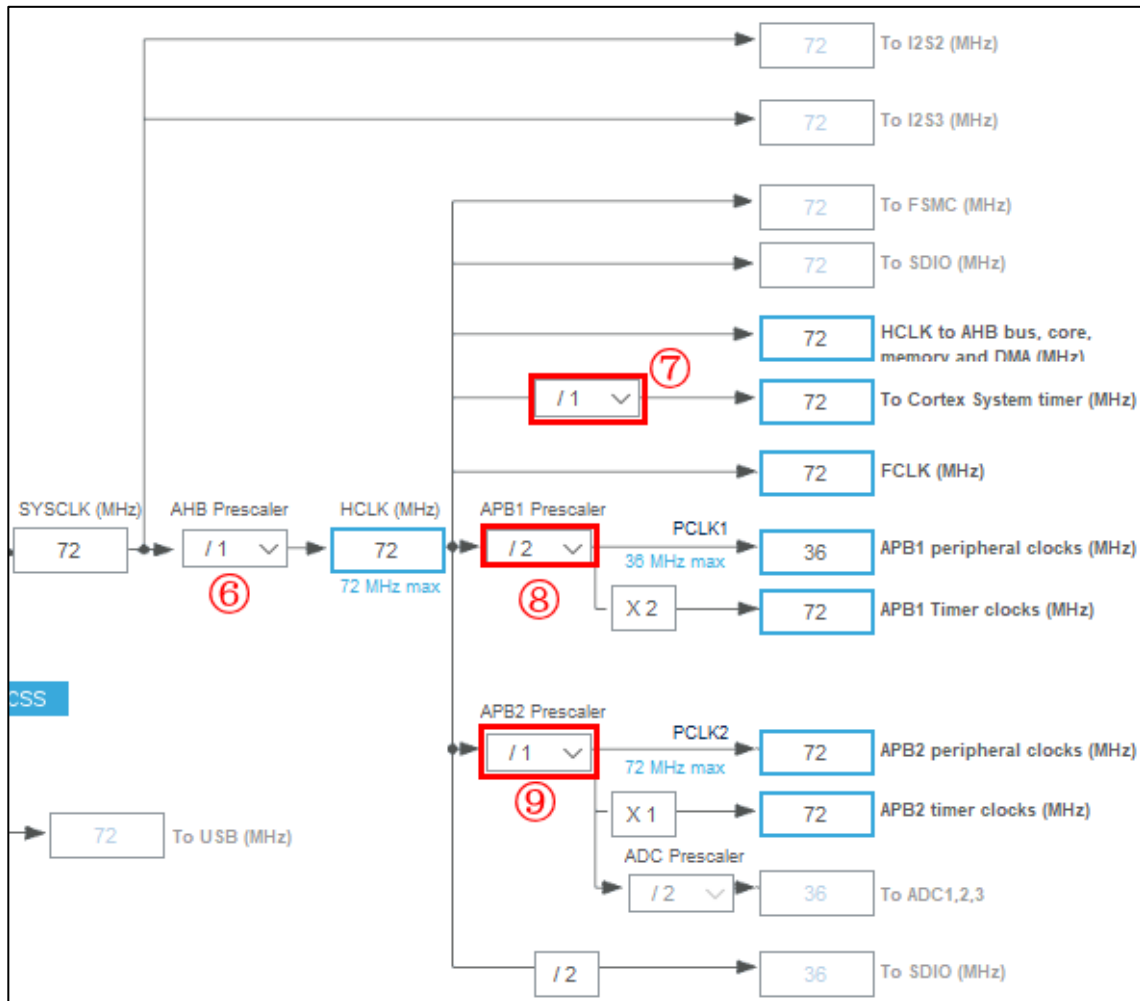


图 10.3.3.9 AHB、APB1、APB2、APB3 和 APB4 总线时钟配置

AHB、APB1 和 APB2 总线时钟以及 SysTick 时钟的来源源于系统时钟 SYSCLK。其中 AHB 总线时钟 HCLK 由 SYSCLK 经过 AHB 预分频器之后得到,如果我们要设置 HCLK 为 72MHz(最大为 72Mhz),那么我们需要配置图中标号⑥的地方为 1 即可。得到 HCLK 之后,接下来我们将在图标号⑦~⑨处同样的方法依次配置 SysTick、APB1 和 APB 分频系数分别为 1、2 和 1。注意!systick 固定为 72MHz,配置完成之后,那么 HCLK=72MHZ, SysTick=72MHz, PCLK1=36MHz, PCLK2=72MHz, 这和之前例程配置的时钟是主频一样的。

以上方法是手动计算的方法,是为了帮助我们更好地去认识 STM32 时钟的配置方法,当然 CubeMX 也提供了更简单的方法:在图 10.3.3.9 的“HCLK(MHz)”位置,实际上是可以编辑的。我们直接输入我们要的主频,这里是 72Mhz,按回车键, CubeMX 会帮我们提供一种设置主频和其它时钟的建议,选择是后会由软件自动配置好,当然只有启用外部的晶振后才能配置到 72Mhz 的时钟,这里大家自己尝试一下就清楚了,我们不展开讲述了。

4 GPIO 功能引脚配置

本小节,我们讲解怎么使用 STM32CubeMX 工具配置 STM32F1 的 GPIO 口。STM32F103 战舰开发板的 PB5 和 PE5 引脚各连接一个 LED 灯,我们来学习配置这两个 IO 口的相关参数。这里我们回到 STM32CubeMX 的 Pinout&Configuration 选项,在搜索栏输入 PB5 后回车,可以在引脚图中显示位置,如下图 10.3.3.10 所示:

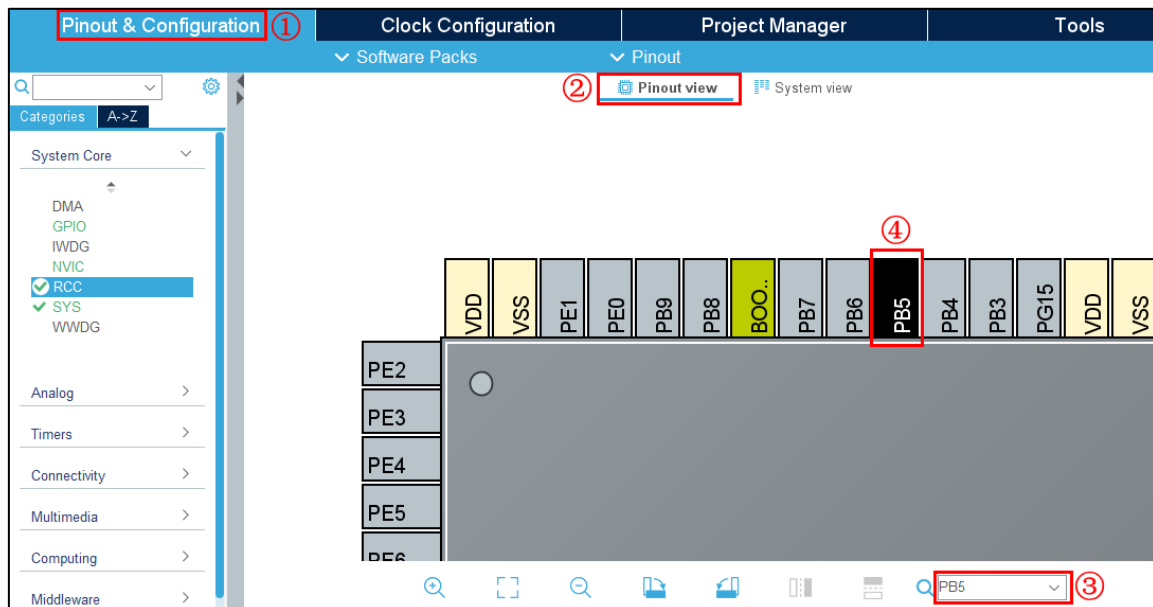


图 10.3.3.10 搜索引脚位置

接下来，我们在图 10.3.3.11 引脚图中点击 PB5，在弹出的下拉菜单中，选择 IO 口的功能为 GPIO_Output。操作方法如下图 10.3.3.11 所示：

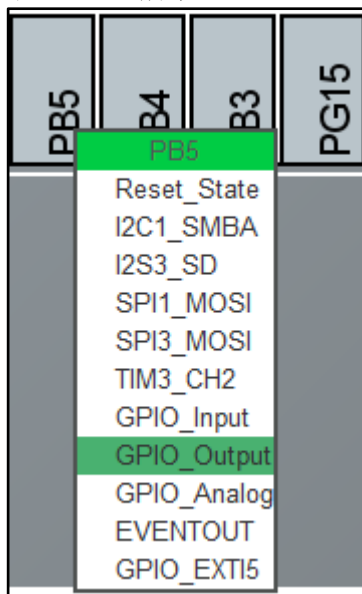


图 10.3.3.11 配置 GPIO 模式

同样的方法，我们配置 PE5 选择功能为 GPIO_Oput 即可。设置好即可看到引脚从灰色变成绿色，标识该管脚已经启用。这里我们需要说明一下，如果我们要配置 IO 口为外部中断引脚或者其他复用功能，我们选择相应的选项即可。配置完 IO 口功能之后，还要配置 IO 口的速度，上下拉等参数。这些参数我们通过 System Core 下的 GPIO 选项进行配置，如图 10.3.3.12 所示。

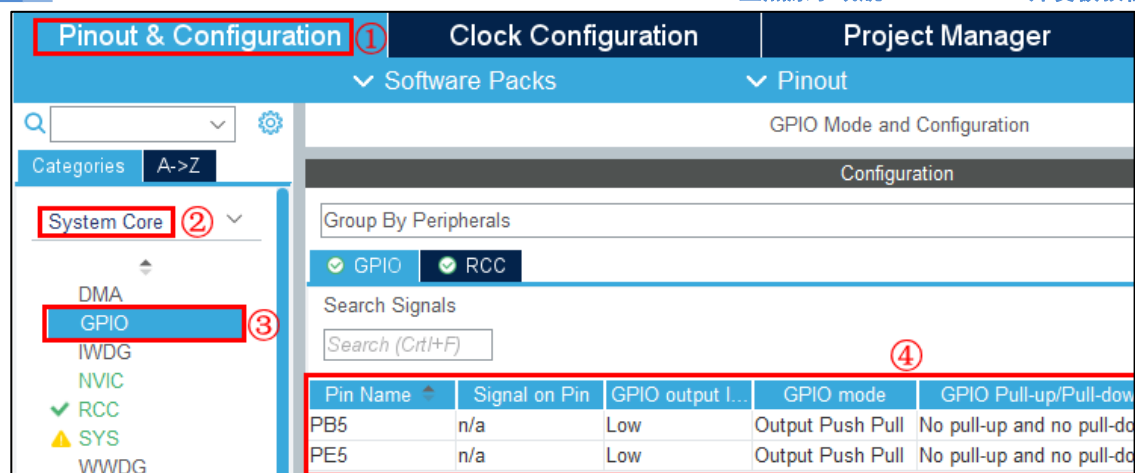


图 10.3.3.12 GPIO 选项

我们先配置 PB5，PE5 和 PB5 配置方法一样的。点击图 10.3.3.12 的④号框里面的 PB5，配置如图 10.3.3.13 所示。

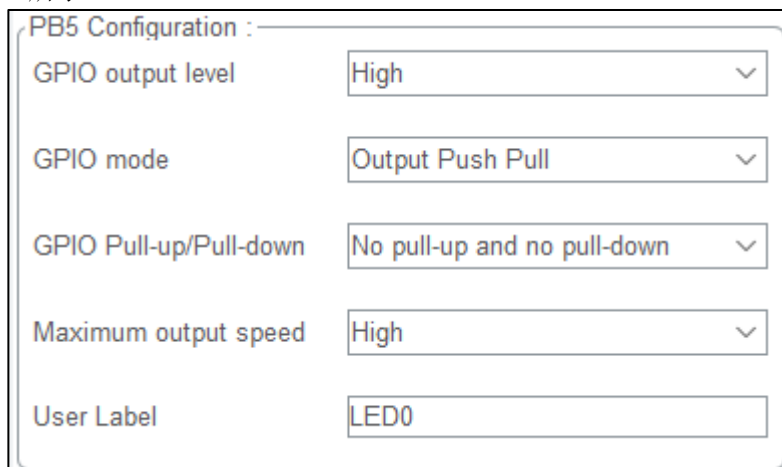


图 10.3.3.13 配置 GPIO 口详细参数

GPIO output level 是 IO 的初始值，由于 LED 一端接 VCC，另一端接 GPIO，故要点亮 LED 灯时，使 GPIO 输出低电平即可。为了一开始让 LED 灯熄灭，我们设置初始值输出高电平。

GPIO mode 我们已经在视图中配置为推挽输出了，这里不需要修改。

GPIO Pull-up/Pull-down 默认是无上下拉，我们这里用默认配置。

Maximum output speed 输出速度配置，默认是低速，我们设置为高速。

User Label 用户符号，我们可以给 PB5 起一个别的名字 LED0。

PE5 也是按照这样的方法配置即可。

5 配置 Debug 选项

由于 CubeMX 默认把 Debug 选项关闭了，这样会给我们带来麻烦：用 CubeMX 生成的工程编译下载一次后，后续再次下载就会提示错误，因此我们要把 Debug 选项打开。这里有多种选择，我们设置成图 10.3.3.14 所示的情况即可。

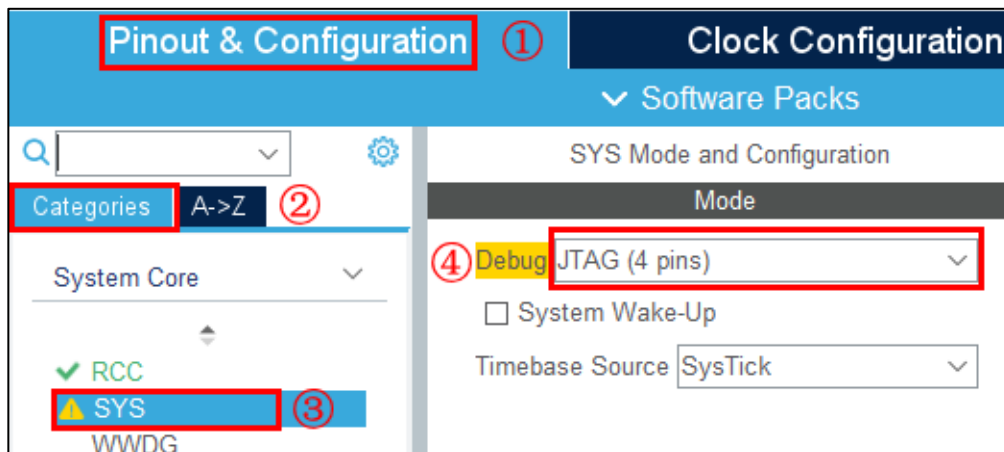


图 10.3.3.14 打开 Debug 选项

如果已经不小心关闭了 Debug 选项，那么下次下载的时候按住复位键，等到工程提示的时候松开复位键即可，因为 STM32 的芯片默认复位上电时的 Debug 引脚功能是开启的。

6 生成工程源码

接下来我们学习怎么设置生成一个工程，如图 10.3.3.15 所示。选择 Project Manager-> Project 选项用来配置工程的选项，我们了解一下里面的信息。

Project Name: 工程名称，填入工程名称（半角，不能有中文字符）

Project Location: 工程保存路径，点击 Browse 选择保存的位置（半角，不能有中文字符）

Toolchain Folder Location: 工具链文件夹位置，默认即可。

Application Structure: 应用的结构，选择 Basic（基础），不勾选 Do not generate the main(), 因为我们要其生成 main 函数。

Toolchain/IDE: 工具链/集成开发环境，我们使用 Keil，因此选择 MDK-ARM，Min Version 选择 V5.27，这里根据 CubeMX 的版本可能会有差异，我们默认使用 V5 以上的版本即可。

Linker Settings 链接器设置：

Minimum Heap Size 最小堆大小，默认（大工程需按需调整）。

Minimum Stack Size 最小栈大小，默认（大工程需按需调整）。

MCU and Firmware Package 是 MCU 及固件包设置：

MCU Reference: 目标 MCU 系列名称。

Firmware Package Name and Version: 固件包名称及版本。

勾选 Use Default Firmware Location，文本框里面的路径就是固件包的存储地址，我们使用默认地址即可。（这里因为我有两个版本的固件包，所以它默认使用最新的，这个关系不大，就用新的）。这样工程生成的设置就设置好了，如图 10.3.3.15 所示。

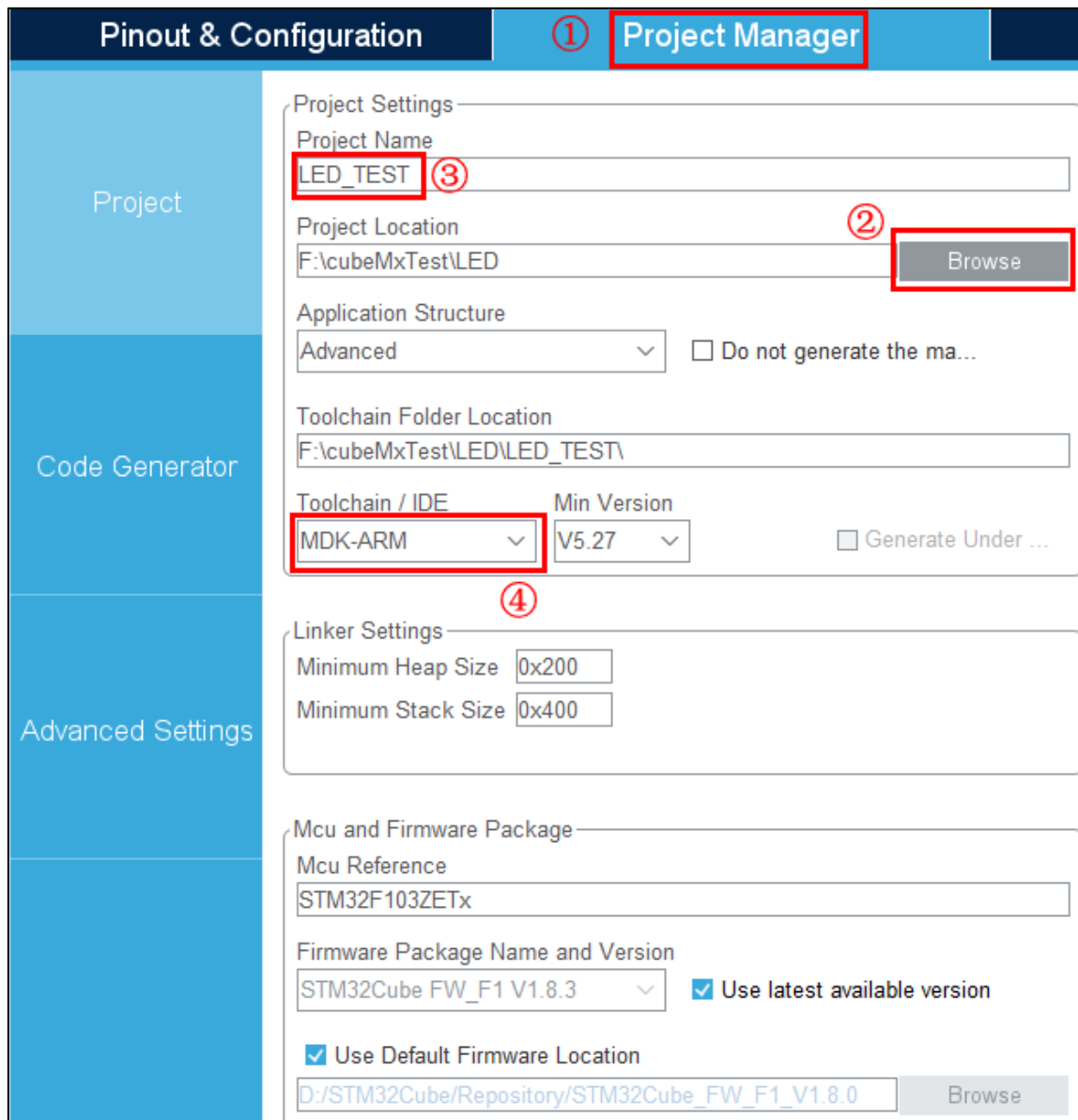


图 10.3.3.15 工程配置

打开 Project Manager-> Code Generator 选项，Generated files 生成文件选项，勾选 Generate peripheral initialization as a pair of '.c/.h' files per peripheral，勾选这个选项的话将会将每个外设单独分开成一组.c、.h 文件，使得代码结构更加的清晰，如图 10.3.3.16 所示。

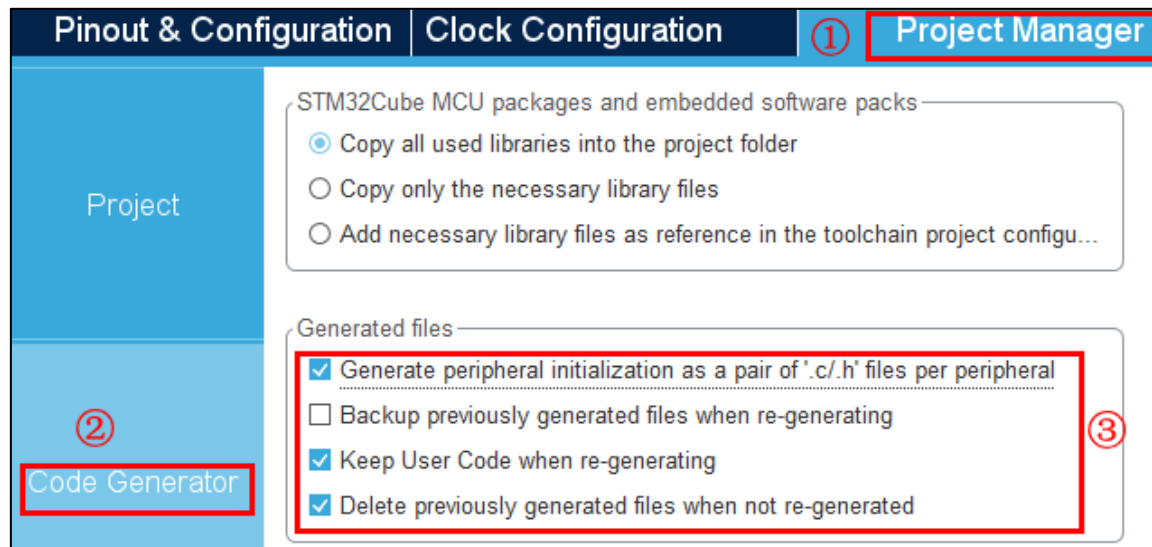


图 10.3.3.16 代码生成器设置

由于 CubeMX 默认勾选了复制所有的库，即工程中不使用到的代码也会复制进来，为了节省 CubeMX 生成工程的空间，我们勾选生成工程时只复制用到的库（这一步是可选操作，大家根据自己的实际选择），如图图 10.3.3.17 所示：

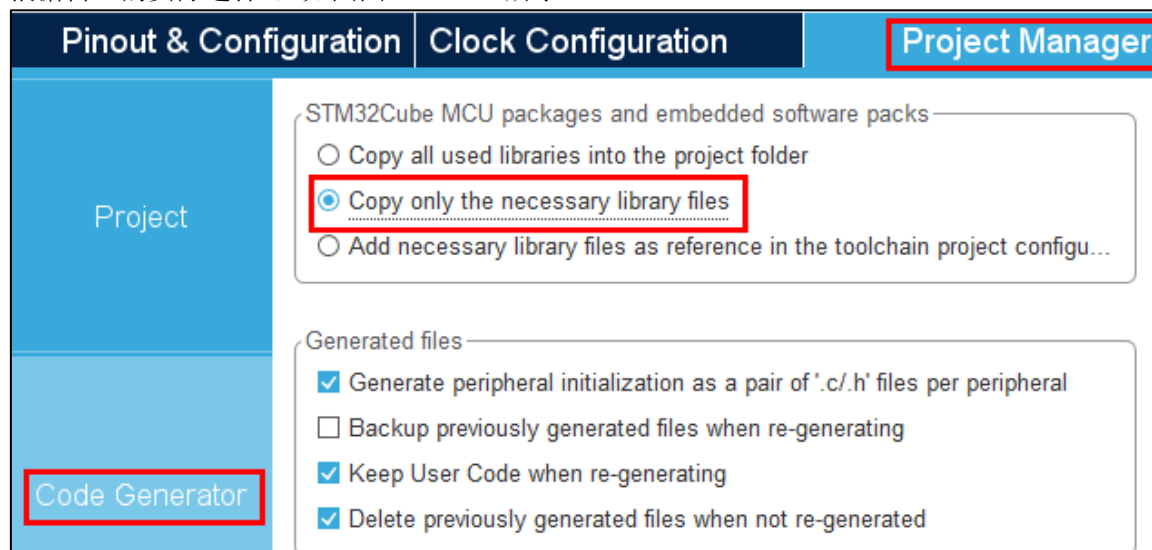


图 10.3.3.17 设置只复制与工程相关的库驱动以减小工程大小
至此工程最基础配置就已经完成，点击蓝色按钮(SENERATE CODE)就可以生成工程。

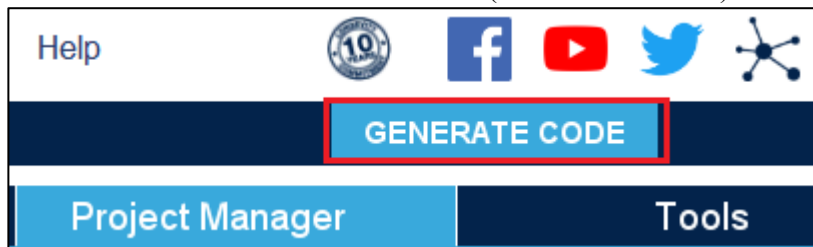


图 10.3.3.18 生成工程

如果我们的 CubeMX 工程放置配置路径中没有中文。生成代码后会弹出类似图 10.3.3.19 的提示窗口，点击 Open Project 就打开 MDK 工程（如果是中文路径则会报错，这里暂时不用管，我们先往下继续操作）。

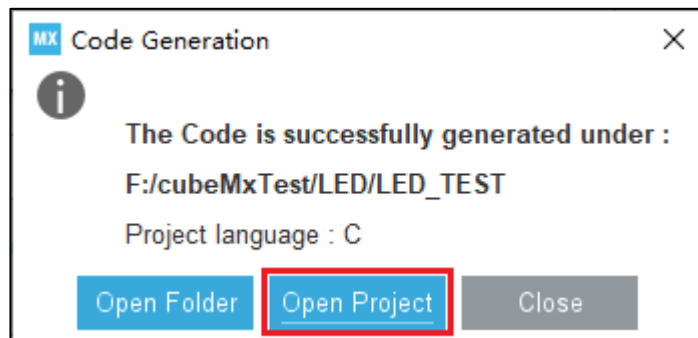


图 10.3.3.19 打开工程

完整的 STM32F1 工程就已经生成完成。生成后的工程目录结构如下图 10.3.3.20 所示：

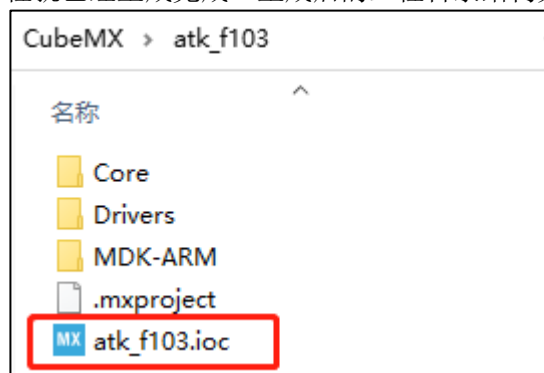


图 10.3.3.20 STM32CubeMX 生成的工程目录结构

Drivers 文件夹存放的是 HAL 库文件和 CMSIS 相关文件。

Inc 文件夹存放的是工程必须的部分头文件。

MDK-ARM 下面存放的是 MDK 工程文件。

Src 文件夹下面存放的是工程必须的部分源文件。

Template.ioc 是 STM32CubeMX 工程文件，双击该文件就会在 STM32CubeMX 中打开。

7 用户程序

在编写用户程序之前，首先我们打开生成的工程模板进行编译，因为我们在之前步骤生成的 CubeMX 工程为 LED_TEST.ioc，故生成的 MDK 工程位置是 .MDK-ARM\LED_TEST.uvprojx，如果大家配置的 CubeMX 的工程名和路径名不含中文或中文字符，按上述步骤生成的工程就可以直接编译通过了。

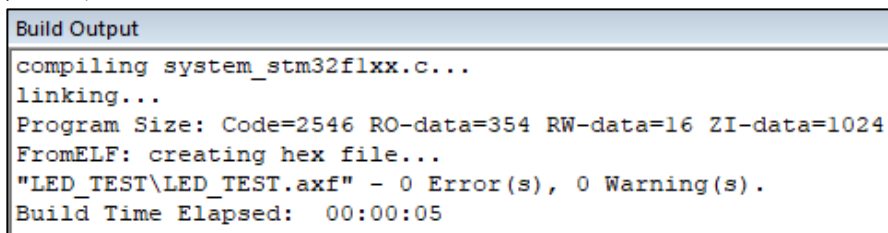


图 10.3.3.21 STM32CubeMX 生成的 MDK 工程编译通过

接下来我们中生成的工程模板的 main.c 文件中找到 main 函数，这里我们删掉了源码注释，关键源码如下：

```
int main(void)
{
    HAL_Init();
    SystemClock_config();
    MX_GPIO_Init();
    /* USER CODE BEGIN WHILE */
    while (1)
    {
```

```
/* USER CODE END WHILE */
}
}
```

大家需要注意，STM32CubeMX 生成的 main.c 文件中，有很多地方有“/* USER CODE BEGIN X*/”和“/* USER CODE END X*/”格式的注释，我们在这些注释的 BEGIN 和 END 之间编写代码，那么重新生成工程之后，这些代码会保留而不会被覆盖。

我们编写一个跑马灯的用户程序，程序具体如下：

```
/**
 * @brief The application entry point.
 * @retval int
 */
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    /* USER CODE BEGIN WHILE */
    while (1)
    {
        HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);
        HAL_Delay(500);
        HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);
        HAL_Delay(500);
    }
    /* USER CODE END WHILE */
}
```

编写好程序后，编译没有任何警告和错误。可以直接下载程序到开发板中，使用 DAP 下载，请注意设置 MDK 的下载选项，如果不清楚设置的读者可以回看本书第四章的相关知识。下载后，可以看到 LED0 和 LED1 同时按 500ms 的频率亮灭，效果与其它版本的新建工程相同。

本小节使用 STM32CubeMX 新建的工程模板在我们光盘目录：“4，程序源码\2，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-4，新建工程实验-CubeMX 版本”中有存放，大家在编写用户代码过程中可以参考该工程的 main.c 文件。

10.4 STM32CubeMX 新建工程使用建议

① 使用 CubeMX 的环境搭建工程，工程文件夹路径、文件名不要带任何中文及中文字符，否则会遇到各种报错；

② 本书以新建工程-HAL 库版本为基准来展开，不对 CubeMX 的使用过多讲解。使用 CubeMX 可以帮助我们快速搭建工程，使用户专注于应用开发，但 STM32 的开发与硬件密切相关，对 STM32 开发来说，抛开底层只专注做应用并不实际，毕竟无法使用一套通用设计来满足不同用户的需求；

③ 关于新建 CubeMX 的工程路径中有中文的情况的解决：

如果我们配置的 CubeMX 工程路径里面有中文可能会报以下的错误（再次强调，CubeMX 的关联路径、文件名不要有中文出现）：

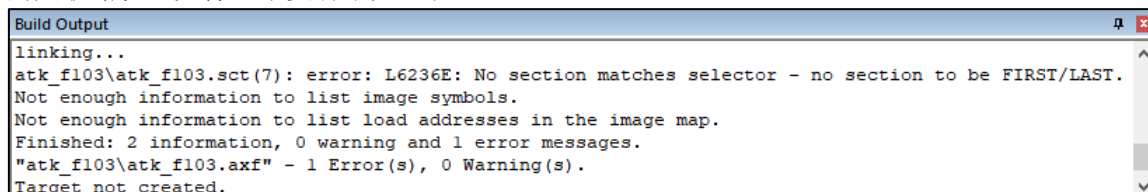


图 10.4.1 直接编译报错

造成错误的原因是 CubeMX 对中文的支持不友好，且生成 MDK 工程默认通过工程中的 CMSIS 那个绿色的控件选择启动文件而不是直接添加启动文件（startup_xxx.s）到我们的工程中，而中文路径时就会找不到，有两个解决办法：

1、用 CubeMX 生成的工程不要放置在包含中文路径的文件夹下；

2、添加启动文件到我们的工程中，我们新建一个 application/MDK-ARM 分组，把 startup_stm32f103xe.s 添加到这个分组，如图 10.3.3.22 所示：

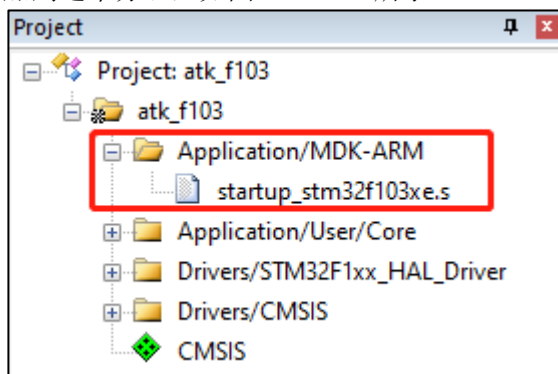


图 10.4.2 STM32CubeMX 生成的工程目录结构

④ 关于配置的文件 CubeMX 工程(.ioc 后缀)名字有中文的情况，我们建议重新新建工程或者把生成的工程文件重命名为英文。因为带中文的 CubeMX 工程生成的 MDK 的 Output 目录有中文，MDK 也会报错，尽管可以重新设置 MDK 工程的 Output 目录和添加③所描述步骤的启动文件，使本次编译通过，但下次重新用 CubeMX 生成工程时，仍旧需要重复修改配置。

第十一章 STM32 时钟配置

MCU 都是基于时序控制的一个系统。这一讲将结合《STM32F10xxx 参考手册_V10（中文版）.pdf》和《Cortex-M3 权威指南》的知识，对 STM32F1 的整体架构作一个简单的介绍，帮助大家更全面、系统地认识 STM32F1 系统的主控结构。了解时钟系统在整个 STM32 系统的贯穿和驱动作用，学会设置 STM32 的系统时钟。

本章将分为如下几个小节：

11.1 认识时钟树

11.2 如何修改主频

11.1 认识时钟树

数字电路的知识告诉我们：任意复杂的电路控制系统都可以经由门电路组成的组合电路实现。回顾《第五章 STM32 基础知识入门》的知识点，我们知道 STM32 内部也是由多种多样的电路模块组合在一起实现的。当一个电路越复杂，在达到正确的输出结果前，它可能因为延时会有一些短暂的中间状态，而这些中间状态有时会导致输出结果会有一个短暂的错误，这叫做电路中的“毛刺现象”，如果电路需要运行得足够快，那么这些错误状态会被其它电路作为输入采样，最终形成一系列的系統错误。为了解决这个问题，在单片机系统中，设计时以时序电路控制替代纯粹的組合电路，在每一级输出结果前对各个信号进行采样，从而使得电路中某些信号即使出现延时也可以保证各个信号的同步，可以避免电路中发生的“毛刺现象”，达到精确控制输出的效果。

由于时序电路的重要性，因此在 MCU 设计时就设计了专门用于控制时序的电路，在芯片设计中称为时钟树设计。由此设计出来的时钟，可以精确控制我们的单片机系统，这也是我们这节要展开分析的时钟分析。为什么是时钟树而不是时钟呢？一个 MCU 越复杂，时钟系统也会相应地变得复杂，如 STM32F1 的时钟系统比较复杂，不像简单的 51 单片机一个系统时钟就可以解决一切。对于 STM32F1 系列的芯片，正常工作的主频可以达到 72Mhz，但并不是所有外设都需要系统时钟这么高的频率，比如看门狗以及 RTC 只需要几十 kHz 的时钟即可。同一个电路，时钟越快功耗越大，同时抗电磁干扰能力也会越弱，所以对于较为复杂的 MCU 一般都是采取多时钟源的方法来解决这些问题。

STM32 本身非常复杂，外设非常的多，为了保持低功耗工作，STM32 的主控默认不开启这些外设功能。用户可以根据自己的需要决定 STM32 芯片要使用的功能，这个功能开关在 STM32 主控中也就是各个外设的时钟。

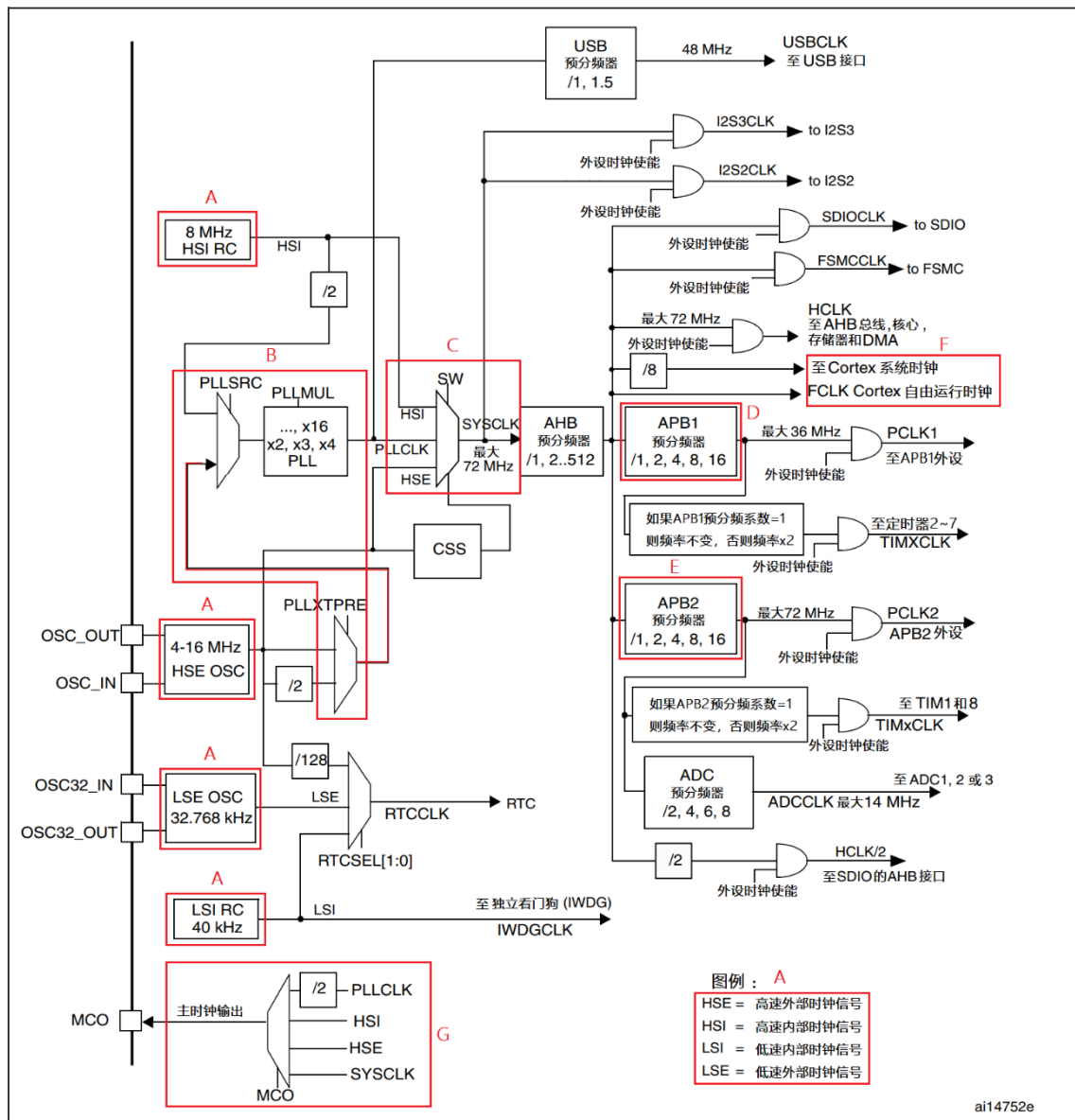


图 11.1.1 STM32F1 时钟系统图

如图 11.1.1 为一个简化的 STM32F1 时钟系统。图中已经把我们主要关注几处标注出来。**A** 部分表示其它电路需要的输入源时钟信号；**B** 为一个特殊的振荡电路“PLL”，由几个部分构成；**C** 为我们重点需要关注的 MCU 内的主时钟“SYSCLK”；AHB 预分频器将 SYSCLK 分频或不分频后分发给其它外设进行处理，包括到 **F** 部分的 Cortex-M 内核系统的时钟。**D**、**E** 部分分别为定时器等外设的时钟源 APB1/APB2。**G** 是 STM32 的时钟输出功能，其它部分等我们学习到再详细探讨。接下来我们来详细了解这些部分的功能。

11.1.1 时钟源

对于 STM32F1，输入时钟源（Input Clock）主要包括 HSI，HSE，LSI，LSE。其中，从时钟频率来分可以分为高速时钟源和低速时钟源，其中 HSI、HSE 高速时钟，LSI 和 LSE 是低速时钟。从来源可分为外部时钟源和内部时钟源，外部时钟源就是从外部通过接晶振的方式获取时钟源，其中 HSE 和 LSE 是外部时钟源；其他是内部时钟源，芯片上电即可产生，不需要借助外部电路。下面我们看看 STM32 的时钟源。

(1) 2 个外部时钟源：

- 高速外部振荡器 HSE (High Speed External Clock signal)

外接石英/陶瓷谐振器，频率为 4MHz~16MHz。本开发板使用的是 8MHz。

- **低速外部振荡器 LSE (Low Speed External Clock signal)**

外接 32.768kHz 石英晶体，主要作用于 RTC 的时钟源。

(2) 2 个内部时钟源：

- **高速内部振荡器 HSI (High Speed Internal Clock signal)**

由内部 RC 振荡器产生，频率为 8MHz。

- **低速内部振荡器 LSI (Low Speed Internal Clock signal)**

由内部 RC 振荡器产生，频率为 40kHz，可作为独立看门狗的时钟源。

芯片上电时默认由内部的 HSI 时钟启动，如果用户进行了硬件和软件的配置，芯片才会根据用户配置调试尝试切换到对应的外部时钟源，所以同时了解这几个时钟源信号还是很有必要的。如何设置时钟的方法我们会在后文提到。

11.1.2 锁相环 PLL

锁相环是自动控制系统中常用的一个反馈电路，在 STM32 主控中，锁相环的作用主要有两个部分：输入时钟净化和倍频。前者是利用锁相环电路的反馈机制实现，后者我们用于使芯片在更高且频率稳定的时钟下工作。

在 STM32 中，锁相环的输出也可以作为芯片系统的时钟源。根据图 11.1.1 的时钟结构，使用锁相环时只需要进行三个部分的配置。为了方便查看，截取了使用 PLL 作为系统时钟源的配置部分，如图 11.1.2.1 所示。

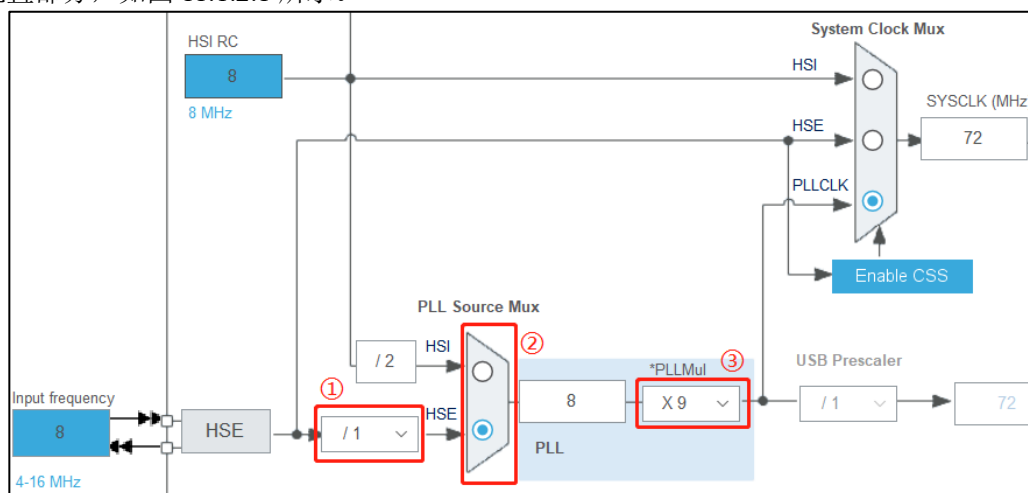


图 11.1.2.1 PLL 时钟配置图

图 11.1.2.1 借用了在 CubeMX 下用锁相环配置 72MHz 时钟的一个示例：

- ◆ **PLLXTPRE: HSE 分频器作为 PLL 输入 (HSE divider for PLL entry)**

即图 11.1.2.1 在标注为①的地方，它专门用于 HSE，ST 设计它有两种方式，并把它的功能放在 RCC_CFGR 寄存器中，我们引用如图 11.1.2.2。

| | |
|-----|--|
| 位17 | PLLXTPRE: HSE分频器作为PLL输入 (HSE divider for PLL entry) 由软件置'1'或清'0'来分频HSE后作为PLL输入时钟。只能在关闭PLL时才能写入此位。 0: HSE不分频 1: HSE 2分频 |
|-----|--|

图 11.1.2.2 PLLXTPRE 设置选项值

从 F103 参考手册可知它的值有两个：一是 2 分频，另一种是 1 分频（不分频）。经过 HSE 分频器处理后的输出振荡时钟信号比直接输入的时钟信号更稳定。

- ◆ **PLLSRC: PLL 输入时钟源 (PLL entry clock source)**

图中②表示的是 PLL 时钟源的选择器，同样的，参考 F103 参考手册：

| | |
|-----|---|
| 位16 | PLLSRC: PLL输入时钟源 (PLL entry clock source) 由软件置'1'或清'0'来选择PLL输入时钟源。只能在关闭PLL时才能写入此位。 0: HSI振荡器时钟经2分频后作为PLL输入时钟 1: HSE时钟作为PLL输入时钟。 |
|-----|---|

图 11.1.2.3 PLLSRC 锁相环时钟源选择

它有两种可选择的输入源：设计为 HSI 的二分频时钟，另一个是 A 处的 PLLXTPRE 处理后的 HSE 信号。

◆ PLLMUL: PLL 倍频系数 (PLL multiplication factor)

图中③所表示的配置锁相环倍频系数，同样地可以查到在 STM32F1 系列中，ST 设置它的有效倍频范围为 2~16 倍。

结合图 11.1.2.1，要实现 72MHz 的主频率，我们通过选择 HSE 不分频作为 PLL 输入的时钟信号，即输入 8Mhz，通过标号③选择倍频因子，可选择 2-16 倍频，我们选择 9 倍频，这样可以得到时钟信号为 $8 \times 9 = 72\text{MHz}$ 。

11.1.3 系统时钟 SYSCLK

STM32 的系统时钟 SYSCLK 为整个芯片提供了时序信号。我们已经大致知道 STM32 主控是时序电路链接起来的。对于相同的稳定运行的电路，时钟频率越高，指令的执行速度越快，单位时间能处理的功能越多。STM32 的系统时钟是可配置的，在 STM32F1 系列中，它可以为 HSI、PLLCLK、HSE 中的一个，通过 CFGR 的位 SW[1:0]设置。

讲解 PLL 作为系统时钟时，根据我们开发板的资源，可以把主频通过 PLL 设置为 72MHz。仍使用 PLL 作为系统时钟源，如果使用 HSI/2，那么可以得到最高主频 $8\text{MHz}/2 \times 16 = 64\text{MHz}$ 。从上面的图 11.2.1 时钟树图可知，AHB、APB1、APB2、内核时钟等时钟通过系统时钟分频得到。根据得到的这个系统时钟，下面我们结合外设来看一看各个外设时钟源。

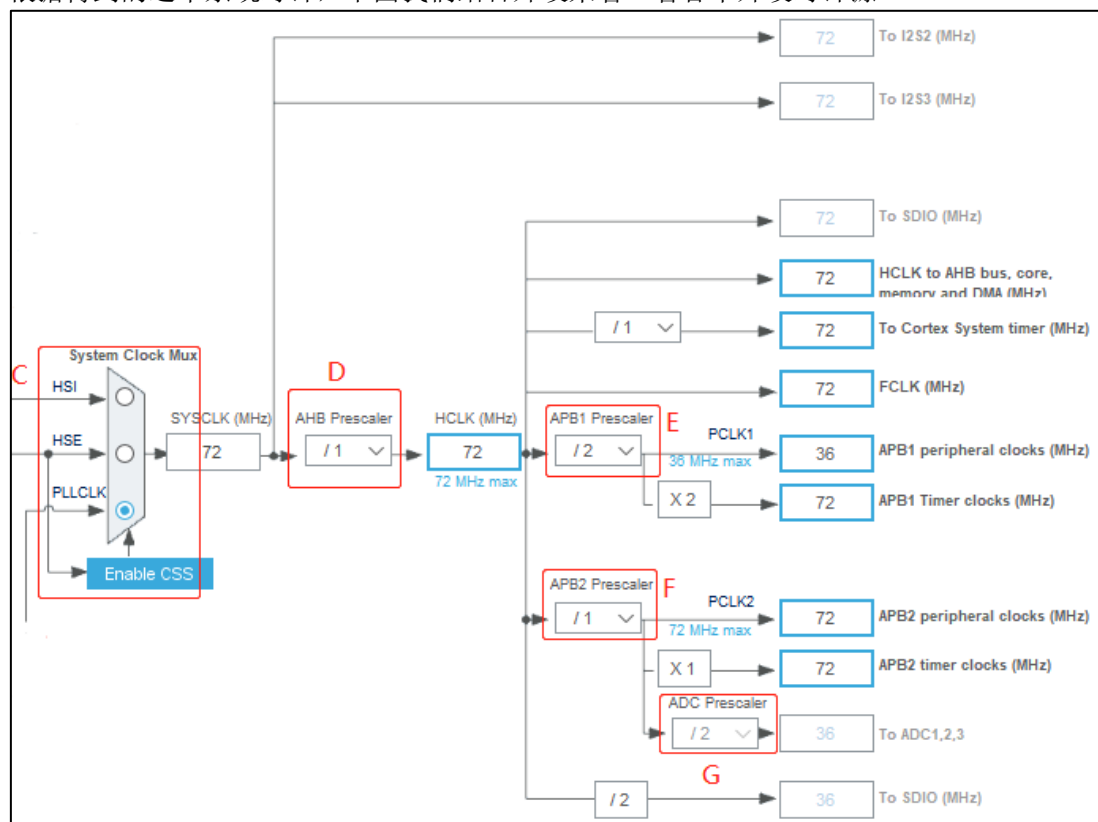


图 11.2.3.1 STM32F103 系统时钟生成图

大家看图 11.2.3.1 STM32F103 系统时钟，标号 C 为系统时钟输入选择，可选时钟信号有外部高速时钟 HSE(8M)、内部高速时钟 HSI(8M)和经过倍频的 PLL CLK(72M)，选择 PLL CLK 作

为系统时钟，此时系统时钟的频率为 72MHz。系统时钟来到**标号 D** 的 AHB 预分频器，其中可选择的分频系数为 1, 2, 4, 8, 16, 32, 64, 128, 256，我们选择不分频，所以 AHB 总线时钟达到最大的 72MHz。

下面介绍一下由 AHB 总线时钟得到的时钟：

APB1 总线时钟，由 HCLK 经过**标号 E** 的低速 APB1 预分频器得到，分频因子可以选择 1, 2, 4, 8, 16，这里我们选择的是 2 分频，所以 APB1 总线时钟为 36M。由于 APB1 是低速总线时钟，所以 APB1 总线最高频率为 36MHz，片上低速的外设就挂载在该总线上，例如有看门狗定时器、定时器 2/3/4/5/6/7、RTC 时钟、USART2/3/4/5、SPI2(I2S2)与 SPI3(I2S3)、I2C1 与 I2C2、CAN、USB 设备和 2 个 DAC。

APB2 总线时钟，由 HCLK 经过**标号 F** 的高速 APB2 预分频器得到，分频因子可以选择 1, 2, 4, 8, 16，这里我们选择的是 1 即不分频，所以 APB2 总线时钟频率为 72M。与 APB2 高速总线链接的外设有外部中断与唤醒控制、7 个通用目的输入/输出(PA、PB、PC、PD、PE、PF 和 PG)、定时器 1、定时器 8、SPI1、USART1、3 个 ADC 和内部温度传感器。其中**标号 G** 是 ADC 的预分频器在后面 ADC 实验中详细说明。

此外，AHB 总线时钟直接作为 SDIO、FSMC、AHB 总线、Cortex 内核、存储器和 DMA 的 HCLK 时钟，并作为 Cortex 内核自由运行时钟 FCLK。

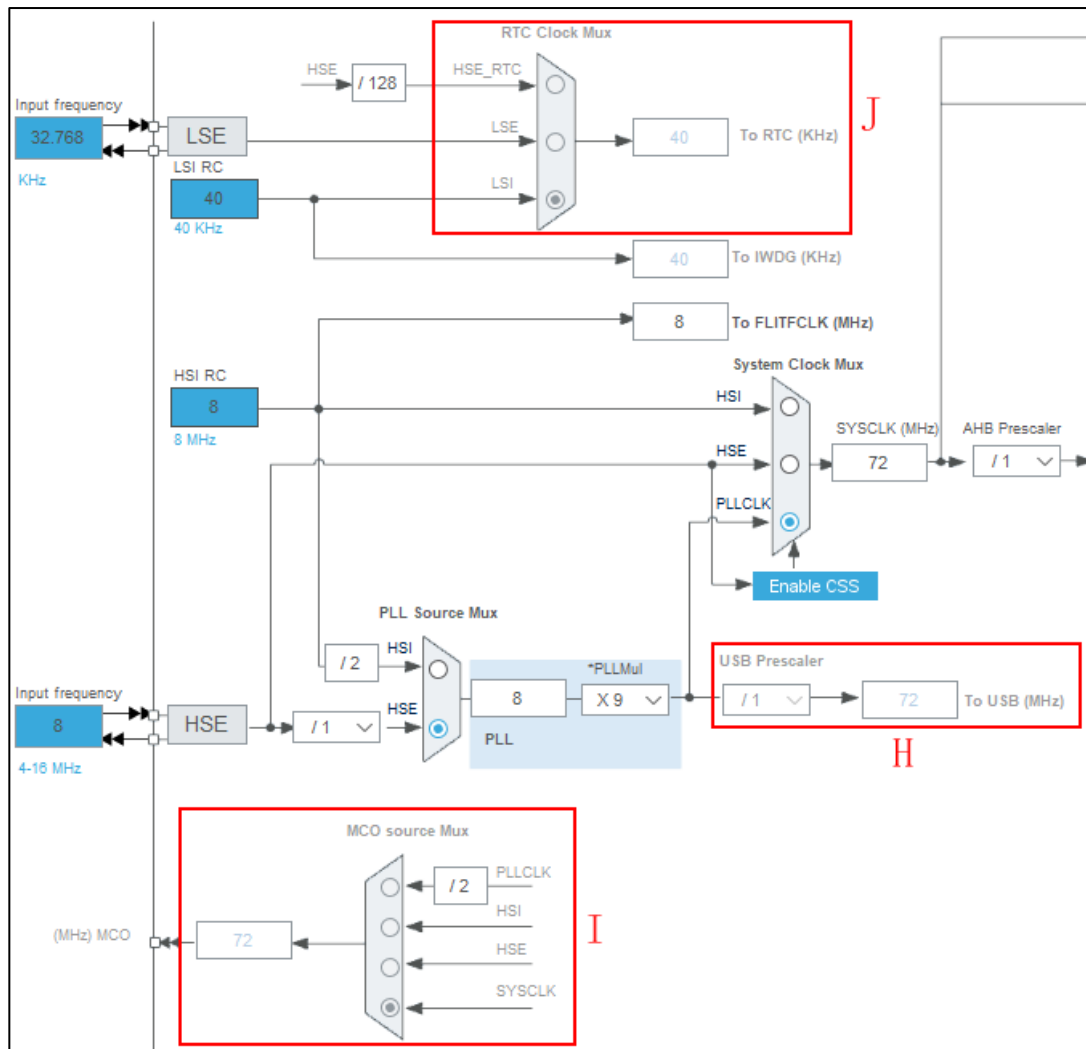


图 11.2.3.2 USB、RTC、MCO 相关时钟

标号 H 是 USBCLK，是一个通用串行接口时钟，时钟来源于 PLLCLK。STM32F103 内置全速功能的 USB 外设，其串行接口引擎需要一个频率为 48MHz 的时钟源。该时钟源只能从 PLL 输出端获取，可以选择为 1.5 分频或者 1 分频，也就是，当需要使用 USB 模块时，PLL 必须使能，并且时钟频率配置为 48MHz 或 72MHz。

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

标号 J 是 RTC 定时器，其时钟源为 HSE/128、LSE 或 LSI。

11.1.4 时钟信号输出 MCO

STM32 允许通过设置，通过 MCO 引脚输出一个稳定的时钟信号。在图 11.1.1 中标注为“G”的部分。以下四个时钟信号可被选作 MCO 时钟：

- SYSCCLK
- HSI
- HSE
- 除 2 的 PLL 时钟

时钟的选择由时钟配置寄存器(RCC_CFGR)中的 MCO[2:0]位控制。

我们可以通过 MCO 引脚来输出时钟信号，测试输出时钟的频率，或作为其它需要时钟信号的外部电路的时钟。

11.2 如何修改主频

STM32F103 默认的情况下（比如：串口 IAP 时或者是未初始化时钟时），使用的是内部 8M 的 HSI 作为时钟源，所以不需要外部晶振也可以下载和运行代码的。

下面我们来讲解如何让 STM32F103 芯片在 72MHZ 的频率下工作，72MHZ 是官方推荐使用的最高的稳定时钟频率。而正点原子的 STM32F103 战舰开发板的外部高速晶振的频率就是 8MHz，我们就是在这个晶振频率的基础上，通过各种倍频和分频得到 72MHZ 的系统工作频率。

11.2.1 STM32F1 时钟系统配置

下面我们将分几步给大家讲解 STM32F1 时钟系统配置过程，这部分内容很重要，请大家认真阅读。

第 1 步：配置 HSE_VALUE

讲解 STM32F1xx_hal_conf.h 文件的时候，我们知道需要宏定义 HSE_VALUE 匹配我们实际硬件的高速晶振频率(这里是 8MHZ)，代码中通过使用宏定义的方式来选择 HSE_VALUE 的值是 25M 或者 8M，这里我们不去定义 USE_STM3210C_EVAL 这个宏或者全局变量即可，选择定义 HSE_VALUE 的值为 8M。代码如下：

```
#if !defined (HSE_VALUE)
#if defined(USE_STM3210C_EVAL)
#define HSE_VALUE 25000000U /*!< Value of the External oscillator in Hz */
#else
#define HSE_VALUE 8000000U /*!< Value of the External oscillator in Hz */
#endif
#endif
#endif /* HSE_VALUE */
```

第 2 步：调用 SystemInit 函数

我们介绍启动文件的时候就知道，在系统启动之后，程序会先执行 SystemInit 函数，进行系统一些初始化配置。启动代码调用 SystemInit 函数如下：

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

LDR R0, =SystemInit
BLX R0
```

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

下面我们来看看 `system_stm32f1xx.c` 文件下定义的 `SystemInit` 程序，源码在 176 行到 188 行，简化函数如下。

```
void SystemInit (void)
{
#if defined(STM32F100xE) || defined(STM32F101xE) || defined(STM32F101xG) || de-
fined(STM32F103xE) || defined(STM32F103xG)
#ifdef DATA_IN_ExtSRAM
    SystemInit_ExtMemCtl();
#endif /* 配置扩展 SRAM */
#endif

    /* 配置中断向量表 */
#if defined(USER_VECT_TAB_ADDRESS)
    SCB->VTOR = VECT_TAB_BASE_ADDRESS | VECT_TAB_OFFSET; /* Vector Table Reloca-
tion in Internal SRAM. */
#endif /* USER_VECT_TAB_ADDRESS */
}
```

从上面代码可以看出，`SystemInit` 主要做了如下两个方面工作：

- 1) 外部存储器配置
- 2) 中断向量表地址配置

然而我们的代码中实际并没有定义 `DATA_IN_ExtSRAM` 和 `USER_VECT_TAB_ADDRESS` 这两个宏，实际上 `SystemInit` 对于正点原子的例程并没有起作用，但我们保留了这个接口。从而避免了去修改启动文件。另外，是可以把一些重要的初始化放到 `SystemInit` 这里，在 `main` 函数运行前就把重要的一些初始化配置好（如 ST 这里是在运行 `main` 函数前先把外部的 SRAM 初始化），这个我们一般用不到，直接到 `main` 函数中处理即可，但也有厂商（如 RT-Thread）就采取了这样的做法，使得 `main` 函数更加简单，但对于初学者，我们暂时不建议这种用法。

HAL 库的 `SystemInit` 函数并没有任何时钟相关配置，所以后续的初始化步骤，我们还必须编写自己的时钟配置函数。

第 3 步：在 main 函数里调用用户编写的时钟设置函数

我们打开 HAL 库例程实验 1 跑马灯实验，看看我们在工程目录 `Drivers\SYSTEM` 分组下面定义的 `sys.c` 文件中的时钟设置函数 `sys_stm32_clock_init` 的内容：

```
/**
 * @brief      系统时钟初始化函数
 * @param      pll_n: PLL 倍频系数(PLL 倍频)，取值范围：2~16
 *              中断向量表位置在启动时已经在 SystemInit() 中初始化
 * @retval     无
 */
void sys_stm32_clock_init(uint32_t pll_n)
{
    HAL_StatusTypeDef ret = HAL_ERROR;
    RCC_OscInitTypeDef rcc_osc_init = {0};
    RCC_ClkInitTypeDef rcc_clk_init = {0};

    rcc_osc_init.OscillatorType = RCC_OSCILLATORTYPE_HSE; /* 选择要配置 HSE */
    rcc_osc_init.HSEState = RCC_HSE_ON; /* 打开 HSE */
    rcc_osc_init.HSEPredivValue = RCC_HSE_PREDIV_DIV1; /* HSE 预分频系数 */
    rcc_osc_init.PLL.PLLState = RCC_PLL_ON; /* 打开 PLL */
    rcc_osc_init.PLL.PLLSource = RCC_PLLSOURCE_HSE; /* PLL 时钟源选择 HSE */
    rcc_osc_init.PLL.PLLMUL = pll_n; /* PLL 倍频系数 */
    ret = HAL_RCC_OscConfig(&rcc_osc_init); /* 初始化 */

    if (ret != HAL_OK)
    {

```

```

        while (1); /* 时钟初始化失败后，程序将可能无法正常执行，可以在这里加入自己的处理 */
    }

    /* 选中 PLL 作为系统时钟源并且配置 HCLK, PCLK1 和 PCLK2 */
    rcc_clk_init.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK
                              | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
    rcc_clk_init.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK; /* 设置系统时钟来自 PLL */
    rcc_clk_init.AHBCLKDivider = RCC_SYSCLK_DIV1;        /* AHB 分频系数为 1 */
    rcc_clk_init.APB1CLKDivider = RCC_HCLK_DIV2;         /* APB1 分频系数为 2 */
    rcc_clk_init.APB2CLKDivider = RCC_HCLK_DIV1;         /* APB2 分频系数为 1 */
    /* 同时设置 FLASH 延时周期为 2WS，也就是 3 个 CPU 周期。 */
    ret = HAL_RCC_ClockConfig(&rcc_clk_init, FLASH_LATENCY_2);

    if (ret != HAL_OK)
    {
        while (1); /* 时钟初始化失败后，程序将可能无法正常执行，可以在这里加入自己的处理 */
    }
}

```

函数 `sys_stm32_clock_init` 就是用户的时钟系统配置函数，除了配置 PLL 相关参数确定 SYSCLK 值之外，还配置了 AHB、APB1 和 APB2 的分频系数，也就是确定了 HCLK, PCLK1 和 PCLK2 的时钟值。

我们首先来看看使用 HAL 库配置 STM32F1 时钟系统的一般步骤：

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

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

下面我们详细讲解这个 2 个步骤。

步骤 1：配置时钟源相关参数，使能并选择 HSE 作为 PLL 时钟源，配置 PLL1，我们调用的函数为 `HAL_RCC_OscConfig()`，该函数在 HAL 库头文件 `STM32F1xx_hal_rcc.h` 中声明，在文件 `STM32F1xx_hal_rcc.c` 中定义。首先我们来看看该函数声明：

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

该函数只有一个形参，就是结构体 `RCC_OscInitTypeDef` 类型指针。接下来我们看看结构体 `RCC_OscInitTypeDef` 的定义：

```

typedef struct
{
    uint32_t OscillatorType;        /* 需要选择配置的振荡器类型 */
    uint32_t HSEState;              /* HSE 状态 */
    uint32_t HSEPredivValue;        /* HSE 预分频值 */
    uint32_t LSEState;              /* LSE 状态 */
    uint32_t HSISState;             /* HIS 状态 */
    uint32_t HSICalibrationValue;   /* HIS 校准值 */
    uint32_t LSISState;             /* LSI 状态 */
    RCC_PLLInitTypeDef PLL;         /* PLL 配置 */
}RCC_OscInitTypeDef;

```

该结构体前面几个参数主要是用来选择配置的振荡器类型。比如我们要开启 HSE，那么我们会设置 `OscillatorType` 的值为 `RCC_OSCILLATORTYPE_HSE`，然后设置 `HSEState` 的值为 `RCC_HSE_ON` 开启 HSE。对于其他时钟源：HIS、LSI、LSE，配置方法类似。

`RCC_OscInitTypeDef` 这个结构体还有一个很重要的成员变量是 `PLL`，它是结构体 `RCC_PLLInitTypeDef` 类型。它的作用是配置 PLL 相关参数，我们来看看它的定义：

```

typedef struct
{
    uint32_t PLLState;              /* PLL 状态 */
    uint32_t PLLSource;             /* PLL 时钟源 */
    uint32_t PLLMUL;                /* PLL 倍频系数 M */
}RCC_PLLInitTypeDef;

```

从 `RCC_PLLInitTypeDef` 结构体的定义很容易看出该结构体主要用来设置 PLL 时钟源以及相关分频倍频参数。这个结构体的定义的相关内容请结合时钟树中红色框的内容一起理解。

接下来我们看看我们的时钟初始化函数 `sys_stm32_clock_init` 中的配置内容：

```
/* 使能 HSE，并选择 HSE 作为 PLL 时钟源，配置 PLLMUL */
RCC_OscInitTypeDef rcc_osc_init = {0};
rcc_osc_init.OscillatorType = RCC_OSCILLATORTYPE_HSE; /* 使能 HSE */
rcc_osc_init.HSEState = RCC_HSE_ON; /* 打开 HSE */
rcc_osc_init.HSEPredivValue = RCC_HSE_PREDIV_DIV1; /* HSE 预分频 */
rcc_osc_init.PLL.PLLState = RCC_PLL_ON; /* 打开 PLL */
rcc_osc_init.PLL.PLLSource = RCC_PLLSOURCE_HSE; /* PLL 时钟源为 HSE */
rcc_osc_init.PLL.PLLMUL = pll_n; /* 主 PLL 倍频因子 */
ret = HAL_RCC_OscConfig(&rcc_osc_init); /* 初始化 */
```

通过函数的该段程序，我们开启了 HSE 时钟源，同时选择 PLL 时钟源为 HSE，然后把 `sys_stm32_clock_init` 的形参直接设置作为 PLL 的参数 M 的值，这样就达到了设置 PLL 时钟源相关参数的目的。

设置好 PLL 时钟源参数之后，也就是确定了 PLL 的时钟频率，然后到我们的步骤 2。

步骤 2：配置系统时钟源，以及 SYSCLK、AHB、APB1 和 APB2 相关参数，用函数 `HAL_RCC_ClockConfig()`，声明如下：

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

该函数有两个形参，第一个形参 `RCC_ClkInitStruct` 是结构体 `RCC_ClkInitTypeDef` 类型指针变量，用于设置 SYSCLK 时钟源以及 SYSCLK、AHB、APB1 和 APB2 的分频系数。第二个形参 `FLatency` 用于设置 FLASH 延迟。

`RCC_ClkInitTypeDef` 结构体类型定义比较简单，我们来看看其定义：

```
typedef struct
{
    uint32_t ClockType; /* 要配置的时钟 */
    uint32_t SYSCLKSource; /* 系统时钟源 */
    uint32_t AHBCLKDivider; /* AHB 分频系数 */
    uint32_t APB1CLKDivider; /* APB1 分频系数 */
    uint32_t APB2CLKDivider; /* APB2 分频系数 */
}RCC_ClkInitTypeDef;
```

我们在 `sys_stm32_clock_init` 函数中的实际应用配置内容如下：

```
/****** 具体配置******/
/*选中 PLL 作为系统时钟源并且配置 HCLK, PCLK1 和 PCLK2*/
/*设置系统时钟源为 PLL*/
/*AHB 分频系数为 1*/
/*APB1 分频系数为 2*/
/*APB2 分频系数为 1*/
/*同时设置 FLASH 延时周期为 2WS，也就是 3 个 CPU 周期。*/
/*******/
rcc_clk_init.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                           RCC_CLOCKTYPE_HCLK |
                           RCC_CLOCKTYPE_PCLK1 |
                           RCC_CLOCKTYPE_PCLK2);
rcc_clk_init.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
rcc_clk_init.AHBCLKDivider = RCC_SYSCLK_DIV1;
rcc_clk_init.APB1CLKDivider = RCC_HCLK_DIV2;
rcc_clk_init.APB2CLKDivider = RCC_HCLK_DIV1;
ret = HAL_RCC_ClockConfig(&rcc_clk_init, FLASH_LATENCY_2);
```

`sys_stm32_clock_init` 函数中的 `RCC_ClkInitTypeDef` 结构体配置内容：

第一个参数 `ClockType` 配置表示我们要配置的是 SYSCLK、HCLK、PCLK1 和 PCLK 四个时钟。

第二个参数 `SYSCLKSource` 配置选择系统时钟源为 PLL。

第三个参数 `AHBCLKDivider` 配置 AHB 分频系数为 1。

第四个参数 `APB1CLKDivider` 配置 APB1 分频系数为 2。

第五个参数 `APB2CLKDivider` 配置 APB2 分频系数为 1。

根据我们在 `mian` 函数中调用 `sys_stm32_clock_init(RCC_PLL_MUL9)` 时设置的形参数值，我们可以计算出，PLL 时钟为 $PLLCLK = HSE * 9 = 8MHz * 9 = 72MHz$ 。

同时我们选择系统时钟源为 PLL，所以系统时钟 $SYSCCLK = 72MHz$ 。AHB 分频系数为 1，故频率为 $HCLK = SYSCCLK / 1 = 72MHz$ 。APB1 分频系数为 2，故其频率为 $PCLK1 = HCLK / 2 = 36MHz$ 。APB2 分频系数为 1，故其频率为 $PCLK2 = HCLK / 1 = 72MHz$ 。我们总结一下通过调用函数 `sys_stm32_clock_init(RCC_PLL_MUL9)` 之后的关键时钟频率值：

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

最后我们来看看函数 `HAL_RCC_ClockConfig` 第二个入口参数 `FLatency` 的含义，为了使 FLASH 读写正确（因为 72Mhz 的时钟比 Flash 的操作速度 24Mhz 要快得多，操作速度不匹配容易导致 Flash 操作失败），所以需要设置延时时间。对于 STM32F1 系列，FLASH 延迟配置参数值是通过下表 11.2.1.1 来确定的，具体可以参考《STM32F10xxx 闪存编程参考手册》3 寄存器说明/3.1 闪存访问控制寄存器。

LATENCY: 时延

这些位表示 SYSCCLK(系统时钟)周期与闪存访问时间的比例

000: 零等待状态，当 $0 < SYSCCLK \leq 24MHz$

001: 一个等待状态，当 $24MHz < SYSCCLK \leq 48MHz$

010: 两个等待状态，当 $48MHz < SYSCCLK \leq 72MHz$

表 11.2.1.1 FLASH 推荐的等待状态和编程延迟数

从上可以看出，我们设置值为 `FLASH_LATENCY_2`，也就是 2WS，也就是 3 个 CPU 周期，为什么呢？因为经过上面的配置之后，系统时钟频率达到了最高的 72MHz，对应的就是两个等待状态，所以选择 `FLASH_LATENCY_2`。

时钟系统配置相关知识就给大家讲解到这里。

11.2.2 STM32F1 时钟使能和配置

上一节我们讲解了时钟系统配置步骤。在配置好时钟系统之后，如果我们要使用某些外设，例如 GPIO，ADC 等，我们还要使能这些外设时钟。这里大家必须注意，如果在使用外设之前没有使能外设时钟，这个外设是不可能正常运行的。STM32 的外设时钟使能是在 RCC 相关寄存器中配置的。因为 RCC 相关寄存器非常多，有兴趣的同学可以直接打开《STM32F10xxx 参考手册_V10（中文版）.pdf》6.3 小节查看所有 RCC 相关寄存器的配置。接下来我们来讲解通过 STM32F1 的 HAL 库使能外设时钟的方法。

在 STM32F1 的 HAL 库中，外设时钟使能操作都是在 RCC 相关固件库文件头文件 `STM32F1xx_hal_rcc.h` 定义的。大家打开 `STM32F1xx_hal_rcc.h` 头文件可以看到文件中除了少数几个函数声明之外大部分都是宏定义标识符。外设时钟使能在 HAL 库中都是通过宏定义标识符来实现的。首先，我们来看看 GPIOA 的外设时钟使能宏定义标识符：

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

这段代码主要是定义了一个宏定义标识符 `__HAL_RCC_GPIOA_CLK_ENABLE()`，它的核心操作是通过下面这行代码实现的：

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

这行代码的作用是，设置寄存器 `RCC->APB2ENR` 的相关位为 1，至于哪个位，是由宏定义标识符 `RCC_APB2ENR_IOPAEN` 的值决定的，而它的值为：

```
#define RCC_APB2ENR_IOPAEN_Pos (0U)
#define RCC_APB2ENR_IOPAEN_Msk (0x1UL << RCC_APB2ENR_IOPAEN_Pos)
#define RCC_APB2ENR_IOPAEN (RCC_APB2ENR_IOPAEN_Msk)
```


上面三行代码很容易计算出来 `RCC_APB2ENR_IOPAEN=(0x00000001<<2)`，因此上面代码的作用是设置寄存器 `RCC->APB2ENR` 寄存器的位 2 为 1。我们可以从 STM32F1 的参考手册中搜索 `APB2ENR` 寄存器定义，位 2 的作用是用来使用 `GPIOA` 时钟。`APB2ENR` 寄存器的位 2 描述如下：

位 0 IOPAEN：IO 端 A 时钟使能(I/O port A clock enable)

由软件置 ‘1’ 或清 ‘0’

0：IO 端口 A 时钟关闭

1：IO 端口 A 时钟开启

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

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

对于其他外设，同样都是在 `STM32F1xx_hal_rcc.h` 头文件中定义，大家只需要找到相关宏定义标识符即可，这里我们列出几个常用使能外设时钟的宏定义标识符使用方法：

```
__HAL_RCC_DMA1_CLK_ENABLE(); /* 使能 DMA1 时钟 */
```

```
__HAL_RCC_USART2_CLK_ENABLE(); /* 使能串口 2 时钟 */
```

```
__HAL_RCC_TIM1_CLK_ENABLE(); /* 使能 TIM1 时钟 */
```

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

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

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

```
__HAL_RCC_DMA1_CLK_DISABLE(); /* 禁止 DMA1 时钟 */
```

```
__HAL_RCC_USART2_CLK_DISABLE(); /* 禁止串口 2 时钟 */
```

```
__HAL_RCC_TIM1_CLK_DISABLE(); /* 禁止 TIM1 时钟 */
```

关于 STM32F1 的外设时钟使能和禁止方法我们就给大家讲解到这里。

第十二章 SYSTEM 文件夹介绍

SYSTEM 文件夹里面的代码由正点原子提供，是 STM32F1xx 系列的底层核心驱动函数，可以用在 STM32F1xx 系列的各个型号上面，方便大家快速构建自己的工程。本章，我们将向大家介绍这些代码的由来及其功能，也希望大家可以灵活使用 SYSTEM 文件夹提供的函数，来快速构建工程，并实际应用到自己的项目中去。

SYSTEM 文件夹下包含了 delay、sys、usart 等三个文件夹。分别包含了 delay.c、sys.c、usart.c 及其头文件。这 3 个 c 文件提供了系统时钟设置、延时和串口 1 调试功能，任何一款 STM32F1 都具备这几个基本外设，所以可以快速地将这些设置应用到任意一款 STM32F1 产品上，通过这些驱动文件实现快速移植和辅助开发的效果。

本章将分为如下几个小节：

12.1 delay 文件夹代码介绍

12.2 sys 文件夹代码介绍

12.3 usart 文件夹代码介绍

12.1 delay 文件夹代码介绍

delay 文件夹内包含了 delay.c 和 delay.h 两个文件，这两个文件用来实现系统的延时功能，其中包含 7 个函数：

```
void delay_osschedlock(void);
void delay_osschedunlock(void);
void delay_ostimedly(uint32_t ticks);
void SysTick_Handler(void);
void delay_init(uint16_t sysclk);
void delay_us(uint32_t nus);
void delay_ms(uint16_t nms);
```

前面 4 个函数，仅在支持操作系统（OS）的时候，需要用到，而后面 3 个函数，则不论是否支持 OS 都需要用到。

在介绍这些函数之前，我们先了解一下 delay 延时的编程思想：CM3 内核处理器，内部包含了一个 SysTick 定时器，SysTick 是一个 24 位的向下递减的计数定时器，当计数值减到 0 时，将从 RELOAD 寄存器中自动重装载定时初值，开始新一轮计数。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在《STM32F10xxx 参考手册_V10(中文版).pdf》里面介绍的很简单，其详细介绍，请参阅《Cortex-M3 权威指南》第 133 页。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

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

这里，我们以 UCOSII 为例，介绍如何实现操作系统和我们的 delay 函数共用 SysTick 定时器。首先，我们简单介绍下 UCOSII 的时钟：ucos 运行需要一个系统时钟节拍（类似“心跳”），而这个节拍是固定的（由 OS_TICKS_PER_SEC 宏定义设置），比如要求 5ms 一次（即可设置：OS_TICKS_PER_SEC=200），在 STM32 上面，一般是由 SysTick 来提供这个节拍，也就是 SysTick 要设置为 5ms 中断一次，为 ucos 提供时钟节拍，而且这个时钟一般是不能被打断的（否则就不准了）。

因为在 ucos 下 systick 不能再被随意更改，如果我们还想利用 systick 来做 delay_us 或者 delay_ms 的延时，就必须想点办法了，这里我们利用的是**时钟摘取法**。以 delay_us 为例，比如 delay_us(50)，在刚进入 delay_us 的时候先计算好这段延时需要等待的 systick 计数次数，这里为 50*72（假设系统时钟为 72Mhz，在经过 8 分频之后，systick 的频率等于 1/8 系统时钟频率，那么 systick 每增加 1，就是 1/9us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*9，一旦检测到变化达到或者超过这个值，就说明延时 50us 时间到了。这样，我们只是抓取 SysTick 计数器的变化，并不需要修改 SysTick 的任何状态，完全不影响 SysTick 作为 UCOS 时钟节拍的功能，这就是实现 delay 和操作系统共用 SysTick 定时器的原理。

下面我们开始介绍这几个函数。

12.1.1 操作系统支持宏定义及相关函数

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

```
/*
 * 当 delay_us/delay_ms 需要支持 OS 的时候需要三个与 OS 相关的宏定义和函数来支持
 * 首先是 3 个宏定义:
 *     delay_osrunning      : 用于表示 OS 当前是否正在运行, 以决定是否可以使用相关函数
 *     delay_ostickspersec  : 用于表示 OS 设定的时钟节拍, delay_init 将根据这个参数来初始化、
 *                           systick
 *     delay_osintnesting   : 用于表示 OS 中断嵌套级别, 因为中断里面不可以调度, delay_ms 使用
 *                           该参数来决定如何运行
 *
 * 然后是 3 个函数:
 *     delay_osschedlock    : 用于锁定 OS 任务调度, 禁止调度
 *     delay_osschedunlock  : 用于解锁 OS 任务调度, 重新开启调度
 *     delay_ostimedly      : 用于 OS 延时, 可以引起任务调度.
 *
 * 本例程仅作 UCOSII 和 UCOSIII 的支持, 其他 OS, 请自行参考着移植
 */

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

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

/**
 * @brief      us 级延时, 关闭任务调度 (防止打断 us 级延迟)
 * @param      无
 * @retval     无
 */
static void delay_osschedlock(void)
{
#ifdef CPU_CFG_CRITICAL_METHOD /* 使用 UCOSIII */
    OS_ERR err;
    OSSchedLock(&err); /* UCOSIII 的方式, 禁止调度, 防止打断 us 延时 */
#else /* 否则 UCOSII */
    OSSchedLock(); /* UCOSII 的方式, 禁止调度, 防止打断 us 延时 */
#endif
}

/**
 * @brief      us 级延时, 恢复任务调度
 * @param      无
 * @retval     无
 */
static void delay_osschedunlock(void)
{
#ifdef CPU_CFG_CRITICAL_METHOD /* 使用 UCOSIII */

```

```

    OS_ERR err;
    OSSchedUnlock(&err);          /* UCOSIII 的方式,恢复调度 */
#else                             /* 否则 UCOSII */
    OSSchedUnlock();              /* UCOSII 的方式,恢复调度 */
#endif
}

/**
 * @brief      us 级延时,恢复任务调度
 * @param      ticks: 延时的节拍数
 * @retval     无
 */
static void delay_ostimedly(uint32_t ticks)
{
#ifdef CPU_CFG_CRITICAL_METHOD
    OS_ERR err;
    OSTimeDly(ticks, OS_OPT_TIME_PERIODIC, &err); /* UCOSIII 延时采用周期模式 */
#else
    OSTimeDly(ticks); /* UCOSII 延时 */
#endif
}

/**
 * @brief      systick 中断服务函数,使用 OS 时用到
 * @param      ticks: 延时的节拍数
 * @retval     无
 */
void SysTick_Handler(void)
{
    if (delay_osrunning == 1) /* OS 开始跑了,才执行正常的调度处理 */
    {
        OSIntEnter();          /* 进入中断 */
        OSTimeTick();          /* 调用 ucos 的时钟服务程序 */
        OSIntExit();           /* 触发任务切换软中断 */
    }
    HAL_IncTick();
}
#endif

```

以上代码,仅支持 UCOSII 和 UCOSIII,不过,对于其他 OS 的支持,也只需要对以上代码进行简单修改即可实现。

支持 OS 需要用到的三个宏定义(以 UCOSII 为例)即:

```

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

```

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

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

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

支持 OS 需要用到的 4 个函数,即:

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

函数: delay_osschedunlock, 同样用于 delay_us 延时,作用是在延时结束后恢复 OS 的调度,继续正常的 OS 任务调度。

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

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

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

12.1.2 delay_init 函数

该函数用来初始化 2 个重要参数：g_fac_us 以及 g_fac_ms；同时把 SysTick 的时钟源选择为外部时钟，如果需要在支持操作系统（OS），只需要在 sys.h 里面，设置 SYS_SUPPORT_OS 宏的值为 1 即可，然后，该函数会根据 delay_ostickspersec 宏的设置，来配置 SysTick 的中断时间，并开启 SysTick 中断。具体代码如下：

```
/**
 * @brief      初始化延迟函数
 * @param      sysclk: 系统时钟频率，即 CPU 频率(HCLK)
 * @retval     无
 */
void delay_init(uint16_t sysclk)
{
    #if SYS_SUPPORT_OS      /* 如果需要在支持 OS. */
        uint32_t reload;
    #endif
    SysTick->CTRL = 0; /*清 SysTick 状态，以便下一步重设，如果这里开了中断会关闭其中断*/
    /* SYSTICK 使用内核时钟源 8 分频,因 systick 的计数器最大值只有 2^24 */
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK_DIV8);
    g_fac_us = sysclk / 8; /* 不论是否使用 OS,g_fac_us 都需要使用,作为 1us 的基础时基 */
    #if SYS_SUPPORT_OS      /* 如果需要在支持 OS. */
        reload = sysclk / 8; /* 每秒的计数次数 单位为 M */
        reload *= 1000000/delay_ostickspersec; /* 根据 delay_ostickspersec 设定溢出时间*/
        g_fac_ms = 1000 / delay_ostickspersec; /* 代表 OS 可以延时的最少单位 */
        SysTick->CTRL |= 1 << 1; /* 开启 SYSTICK 中断 */
        SysTick->LOAD = reload; /* 每 1/delay_ostickspersec 秒中断一次 */
        SysTick->CTRL |= 1 << 0; /* 开启 SYSTICK */
    #endif
}
```

可以看到，delay_init 函数使用了条件编译，来选择不同的初始化过程，如果不使用 OS 的时候，只是设置一下 SysTick 的时钟源以及确定 g_fac_us 值。而如果使用 OS 的时候，则会进行一些不同的配置，这里的条件编译是根据 SYS_SUPPORT_OS 这个宏来确定的，该宏在 sys.h 里面定义。

SysTick 是 MDK 定义了一个结构体（在 core_m3.h 里面），里面包含 CTRL、LOAD、VAL、CALIB 等 4 个寄存器，

SysTick->CTRL（地址：0xE000_E010）的各位定义如图 12.1.2.1 所示：

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

图 12.1.2.1 SysTick->CTRL 寄存器各位定义

SysTick->LOAD（地址：0xE000_E014）的定义如图 12.1.2.2 所示，这里要注意这个的最大可重装值只有 24 位：

| 位段 | 名称 | 类型 | 复位值 | 描述 |
|------|--------|-----|-----|----------------|
| 23:0 | RELOAD | R/W | 0 | 当倒数至零时，将被重装载的值 |

图 12.1.2.2 SysTick->LOAD 寄存器各位定义

SysTick->VAL（地址：0xE000_E018）的定义如图 12.1.2.3 所示：

| 位段 | 名称 | 类型 | 复位值 | 描述 |
|------|---------|------|-----|--|
| 23:0 | CURRENT | R/Wc | 0 | 读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志 |

图 12.1.2.3 SysTick->VAL 寄存器各位定义

SysTick->CALIB（地址：0xE000_E01C）不常用，在这里我们也用不到，故不介绍了。

HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK_DIV8);这句代码把 SysTick 的时钟设置为内核时钟的 1/8，这里需要注意的是：SysTick 的时钟源自 HCLK，假设置系统时钟为 72MHZ，经过分频器 8 分频后，那么 SysTick 的时钟即为 9Mhz，也就是 SysTick 的计数器 VAL 每减 1，就代表时间过了 1/9us。

在不使用 OS 的时候：fac_us，为 us 延时的基数，也就是延时 1us，Systick 定时器需要走过的时钟周期数。

当使用 OS 的时候，fac_us 还是 us 延时的基数，不过这个值不会被写到 SysTick->LOAD 寄存器来实现延时，而是通过时钟摘取的办法实现的（前面已经介绍了）。而 g_fac_ms 则代表 ucos 自带的延时函数所能实现的最小延时时间（如 delay_ostickspersec=200，那么 g_fac_ms 就是 5ms）。

12.1.3 delay_us 函数

该函数用来延时指定的 us，其参数 nus 为要延时的微秒数。该函数有使用 OS 和不使用 OS 两个版本，这里我们首先介绍不使用 OS 的时候，实现函数如下：

```
/**
 * @brief      延时 nus
 * @param      nus: 要延时的 us 数.
 * @note       注意：nus 的值,不要大于 1864135us(最大值即 2^24/g_fac_us @g_fac_us = 9)
 * @retval     无
 */
void delay_us(uint32_t nus)
{
    uint32_t temp;
    SysTick->LOAD = nus * g_fac_us; /* 时间加载 */
    SysTick->VAL = 0x00;           /* 清空计数器 */
    SysTick->CTRL |= 1 << 0;      /* 开始倒数 */
    do
    {
        temp = SysTick->CTRL;
    } while ((temp & 0x01) &&
              !(temp & (1 << 16))); /* CTRL.ENABLE 位必须为 1，并等待时间到达 */
    SysTick->CTRL &= ~(1 << 0);   /* 关闭 SYSTICK */
    SysTick->VAL = 0x00;           /* 清空计数器 */
}
```

不使用 OS 的 delay_us 函数，首先结合需要延时的时间 nus 与 us 延时的基数，得到 SysTick 计数次数，赋值给 SysTick 重装载数值寄存器。对计数器进行清空操作 SysTick->VAL=0x00。然后对 SysTick->CTRL 寄存器的位 0 赋 1 操作，即使能 SysTick 定时器。当重装载寄存器的值递减到 0 的时候，清除 SysTick->CTRL 寄存器 COUNTFLAG 标志这个特性作为判断的条件，等待计数值变为零，即计时时间到了。

对于使用 OS 的时候，delay_us 的实现函数是使用的时钟摘取法，只不过使用 delay_osschedlock 和 delay_osschedunlock 两个函数，用于调度上锁和解锁，这是为了防止 OS 在 delay_us 的时候打断延时，可能导致的延时不准，所以我们利用这两个函数来实现免打断，从而保证延时精度。

再来看看使用 OS 的时候，delay_us 的实现函数如下：

```
/**
 * @brief      延时 nus
 * @param      nus: 要延时的 us 数.
```

```

* @note      nus 取值范围: 0 ~ 477218588(最大值即 2^32 / g_fac_us @g_fac_us = 9)
* @retval    无
*/
void delay_us(uint32_t nus)
{
    uint32_t ticks;
    uint32_t told, tnow, tcnt = 0;
    uint32_t reload;
    reload = SysTick->LOAD;      /* LOAD 的值 */
    ticks = nus * g_fac_us;      /* 需要的节拍数 */
    delay_osschedlock();        /* 阻止 OS 调度, 防止打断 us 延时 */
    told = SysTick->VAL;         /* 刚进入时的计数器值 */
    while (1)
    {
        tnow = SysTick->VAL;

        if (tnow != told)
        {
            if (tnow < told)
            {
                tcnt += told - tnow; /* 这里注意一下 SYSTICK 是一个递减的计数器就可以了 */
            }
            else
            {
                tcnt += reload - tnow + told;
            }
            told = tnow;

            if (tcnt >= ticks) break; /* 时间超过/等于要延迟的时间, 则退出. */
        }
    };
    delay_osschedunlock();      /* 恢复 OS 调度 */
}

```

这里就正是利用了我们前面提到的时钟摘取法, ticks 是延时 nus 需要等待的 SysTick 计数次数 (也就是延时时间), told 用于记录最近一次的 SysTick->VAL 值, 然后 tnow 则是当前的 SysTick->VAL 值, 通过他们的对比累加, 实现 SysTick 计数次数的统计, 统计值存放在 tcnt 里面, 然后通过对比 tcnt 和 ticks, 来判断延时是否到达, 以达到不修改 SysTick 实现 nus 延时的效果, 从而可以和 OS 共用一个 SysTick。

上面的 delay_osschedlock 和 delay_osschedunlock 是 OS 提供的两个函数, 用于调度上锁和解锁, 这里为了防止 OS 在 delay_us 的时候打断延时, 可能导致的延时不准, 所以我们利用这两个函数来实现免打断, 从而保证延时精度!

12.1.4 delay_ms 函数

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

```

/**
* @brief      延时 nms
* @param      nms: 要延时的 ms 数 (0 < nms <= 65535)
* @retval    无
*/
void delay_ms(uint16_t nms)
{
    /* 这里用 1000, 是考虑到可能有超频应用, 如 128Mhz 时, delay_us 最大只能延时 1048576us 左右 */
    uint32_t repeat = nms / 1000;
    uint32_t remain = nms % 1000;
    while (repeat)
    {
        delay_us(1000 * 1000); /* 利用 delay_us 实现 1000ms 延时 */
        repeat--;
    }
}

```

```
if (remain)
{
    delay_us(remain * 1000);    /* 利用 delay_us, 把尾数延时(remain ms)给做了 */
}
}
```

该函数其实就是多次调用 `delay_us` 函数, 来实现毫秒级延时的。我们做了一些处理, 使得调用 `delay_us` 函数的次数减少, 这样时间会更加精准。再来看看使用 OS 的时候, `delay_ms` 的实现函数如下:

```
/**
 * @brief      延时 nms
 * @param      nms: 要延时的 ms 数 (0< nms <= 65535)
 * @retval     无
 */
void delay_ms(uint16_t nms)
{
    /* 如果 os 已经在跑了, 并且不是在中断里面 (中断里面不能任务调度) */
    if (delay_osrunning && delay_osintnesting == 0)
    {
        if (nms >= g_fac_ms)          /* 延时的时间大于 OS 的最少时间周期 */
        {
            delay_ostimedly(nms / g_fac_ms);    /* OS 延时 */
        }
        nms %= g_fac_ms;              /* OS 已经无法提供这么小的延时了, 采用普通方式延时 */
    }
    delay_us((uint32_t)(nms * 1000));    /* 普通方式延时 */
}
```

该函数中, `delay_osrunning` 是 OS 正在运行的标志, `delay_osintnesting` 则是 OS 中断嵌套次数, 必须 `delay_osrunning` 为真, 且 `delay_osintnesting` 为 0 的时候, 才可以调用 OS 自带的延时函数进行延时 (可以进行任务调度), `delay_ostimedly` 函数就是利用 OS 自带的延时函数, 实现任务级延时的, 其参数代表延时的时钟节拍数 (假设 `delay_ostickspersec=200`, 那么 `delay_ostimedly(1)`, 就代表延时 5ms)。

当 OS 还未运行的时候, 我们的 `delay_ms` 就是直接由 `delay_us` 实现的, OS 下的 `delay_us` 可以实现很长的延时 (达到 53 秒) 而不溢出! 所以放心的使用 `delay_us` 来实现 `delay_ms`, 不过由于 `delay_us` 的时候, 任务调度被上锁了, 所以还是建议不要用 `delay_us` 来延时很长的时间, 否则影响整个系统的性能。

当 OS 运行的时候, 我们的 `delay_ms` 函数将先判断延时时长是否大于等于 1 个 OS 时钟节拍 (`g_fac_ms`), 当大于这个值的时候, 我们就通过调用 OS 的延时函数来实现 (此时任务可以调度), 不足 1 个时钟节拍的时候, 直接调用 `delay_us` 函数实现 (此时任务无法调度)。

12.1.5 HAL 库延时函数 HAL_Delay

前面我们在 7.4.2 章节介绍 `STM32F1xx_hal.c` 文件时, 已经讲解过 `Systick` 实现延时相关函数。实际上, HAL 库提供的延时函数, 只能实现简单的毫秒级别延时, 没有实现 us 级别延时。我们看看 HAL 库的 `HAL_Delay` 函数原定义:

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

HAL 库实现延时功能非常简单, 首先定义了一个 32 位全局变量 `uwTick`, 在 `Systick` 中断

服务函数 SysTick_Handler 中通过调用 HAL_IncTick 实现 uwTick 值不断增加，也就是每隔 1ms 增加 uwTickFreq，而 uwTickFreq 默认是 1。而 HAL_Delay 函数在进入函数之后先记录当前 uwTick 的值，然后不断在循环中读取 uwTick 当前值，进行减运算，得出的就是延时的毫秒数，整个逻辑非常简单也非常清晰。

但是，HAL 库的延时函数在中断服务函数中使用 HAL_Delay 会引起混乱(虽然一般禁止在中断中使用延时函数)，因为它通过中断方式实现，而 SysTick 的中断优先级是最低的，所以在中断中运行 HAL_Delay 会导致延时出现严重误差。所以教程中推荐大家使用正点原子提供的延时函数库，但我们在第七章介绍 HAL 库时，也提到过，不使用操作系统(OS)的情况下，我们禁用了 SysTick 中断，会导致部分 HAL 库函数无法超时退出，读者需要特别留意。

HAL 库的 ms 级别的延时函数 `__weak void HAL_Delay(uint32_t Delay)`；它是弱定义函数，所以用户可以自己重新定义该函数。例如：我们在 `delay.c` 文件可以这样重新定义该函数：

```
/**
 * @brief HAL 库延时函数重定义
 * @param Delay 要延时的毫秒数
 * @retval None
 */
void HAL_Delay(uint32_t Delay)
{
    delay_ms(Delay);
}
```

12.2 sys 文件夹代码介绍

sys 文件夹内包含了 `sys.c` 和 `sys.h` 两个文件，在 `sys.c` 主要实现下面的几个函数，以及一些汇编函数。

```
/* 函数声明 */
void sys_nvic_set_vector_table(uint32_t baseaddr, uint32_t offset);
void sys_standby(void);
void sys_soft_reset(void);
uint8_t sys_stm32_clock_init(uint32_t plln);

/* 汇编函数 */
void sys_wfi_set(void);
void sys_intx_disable(void);
void sys_intx_enable(void);
void sys_msr_msp(uint32_t addr);
```

下面讲一下函数的功能，`sys_nvic_set_vector_table` 函数主要是设置中断向量表偏移地址，`sys_standby` 函数用于进入待机模式，`sys_soft_reset` 函数用于系统软复位，`sys_stm32_clock_init` 函数是系统时钟初始化函数，在 11.2.1 小节 STM32F1 时钟系统配置章节已经有说明了，大家可以复习这部分知识点。

在 `sys.h` 文件中只是对于 `sys.c` 的函数进行声明。

12.3 usart 文件夹代码介绍

该文件夹下面有 `usart.c` 和 `usart.h` 两个文件。在我们的工程使用串口 1 和串口调试助手来实现调试功能，可以把单片机的信息通过串口助手显示到电脑屏幕。串口相关知识，**我们将在第十七章讲解串口实验的时候给大家详细讲解**。本节只给大家讲解 `printf` 函数支持相关的知识。

12.3.1 printf 函数支持

在我们学习 C 语言时，可以通过 `printf` 函数把需要的参数显示到屏幕上，可以做一些简单的调试信息，但对于单片机来说，如果想实现类似的功能来用 `printf` 辅助调试的话，是否有办法呢？有，这就是这一节要讲的内容。

标准库下的 `printf` 为调试属性的函数，如果直接使用，会使单片机进入半主机模式 (semihosting)，这是一种调试模式，直接下载代码后出现程序无法运行，但是在连接调试器进行 Debug 时程序反而能正常工作的情况。半主机是 ARM 目标的一种机制，用于将输入/输出请

求从应用程序代码通信到运行调试器的主机。例如，此机制可用于允许 C 库中的函数（如 printf() 和 scanf()）使用主机的屏幕和键盘，而不是在目标系统上设置屏幕和键盘。这很有用，因为开发硬件通常不具有最终系统的所有输入和输出设备，如屏幕、键盘等。半主机是通过一组定义好的软件指令（如 SVC）SVC 指令（以前称为 SWI 指令）来实现的，这些指令通过程序控制生成异常。应用程序调用相应的半主机调用，然后调试代理处理该异常。调试代理（这里的调试代理是仿真器）提供与主机之间的必需通信。也就是说使用半主机模式必须使用仿真器调试。

如果想在独立环境下运行调试功能的函数，我们这里是 printf，printf 对字符 ch 处理后写入文件 f，最后使用 fputc 将文件 f 输出到显示设备。对于 PC 端的设备，fputc 通过复杂的源码，最终把字符显示到屏幕上。那我们需要做的，就是把 printf 调用的 fputc 函数重新实现，重定向 fputc 的输出，同时避免进入半主机模式。

要避免半主机模式，现在主要有两种方式：一是使用 MicroLib，即微库；另一种方法是确保 ARM 应用程序中没有链接 MicroLib 的半主机相关函数，我们要取消 ARM 的半主机工作模式，这可以通过代码实现。

先说微库，ARM 的 C 微库 MicroLib 是为嵌入式设备开发的一套类似于标准 C 接口函数的精简代码库，用于替代默认 C 库，是专门针对专业嵌入式应用开发而设计的，特别适合那些对存储空间有特别要求的嵌入式应用程序，这些程序一般不在操作系统下运行。使用微库编写程序要注意其与默认 C 库之间存在的一些差异，如 main() 函数不能声明带参数，也无须返回；不支持 stdio，除了无缓冲的 stdin、stdout 和 stderr；微库不支持操作系统函数；微库不支持可选的单或两区存储模式；微库只提供分离的堆和栈两区存储模式等等，它裁减了很多函数，而且还有很多东西不支持。如果原来用标准库可以跑，选择 MicroLib 后却突然不行了，是很常见的。与标准的 C 库不一样，微库重新实现了 printf，使用微库的情况下就不会进入半主机模式了。Keil 下使用微库的方法很简单，在“Target”下勾选“Use MicroLib”即可。

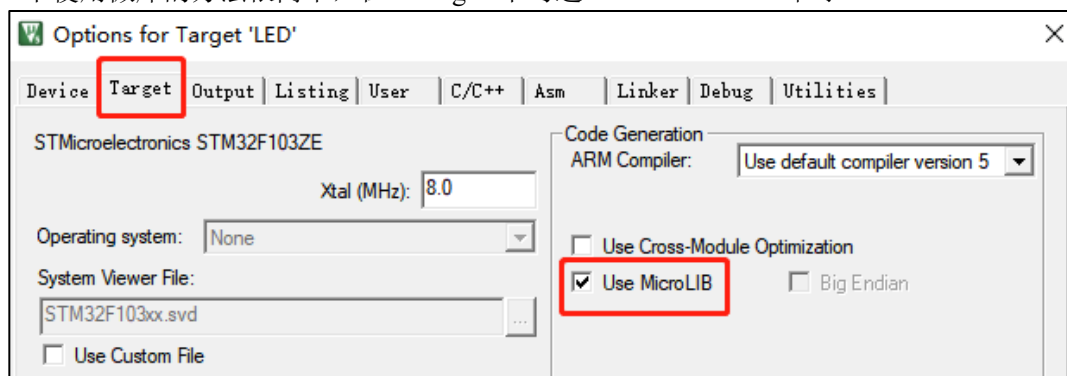


图 12.3.1.1 LED 工程下使用微库的方法

在 Keil5 中，不管是否使用半主机模式，使用 printf, scanf, fopen, fread 等都需要自己填充底层函数，以 printf 为例，需要补充定义 fputc，启用微库后，在我们初始化和使能串口 1 之后，我们只需要重新实现 fputc 的功能即可将每个传给 fputc 函数的字符 ch 重定向到串口 1，如果这时接上串口调试助手的话，可以看到串口的数据。实现的代码如下：

```
#define USART_UX          USART1
/* 重定义 fputc 函数，printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while ((USART_UX->SR & 0X40) == 0); /* 等待上一个字符发送完成 */
    USART_UX->DR = (uint8_t)ch;          /* 将要发送的字符 ch 写入到 DR 寄存器 */
    return ch;
}
```

上面说到了微库的一些限制，使用时注意某些函数与标准库的区别就不会影响到我们代码的正常功能。如果不想使用微库，那就要用到我们提到的第二种方法：取消 ARM 的半主机工作模式；只需在代码中添加不使用半主机的声明即可，对于 AC5 和 AC6 编译器版本，声明半主机的语法不同，为了同时兼容这两种语法，我们在利用编译器自带的宏 __ARMCC_VERSION 判定编译器版本，并根据版本不同选择不同的语法声明不使用半主机模式，具体代码如下：

```
#if (__ARMCC_VERSION >= 6010050) /* 使用 AC6 编译器时 */
```



```
__asm(".global __use_no_semihosting\n\t"); /* 声明不使用半主机模式 */
__asm(".global __ARM_use_no_argv \n\t"); /* AC6 下需要声明 main 函数为无参数格式，否则部分例程可能出现半主机模式 */

#else
/* 使用 AC5 编译器时，要在这里定义 __FILE 和 不使用半主机模式 */
#pragma import(__use_no_semihosting)
/* 解决 HAL 库使用时，某些情况可能报错的 bug */
struct __FILE
{
    int handle;
};
#endif
```

使用的上面的代码，Keil 的编译器就不会把标准库的这部分函数链接到我们的代码里。

如果用到原来半主机模式下的调试函数，需要重新实现它的一些依赖函数接口，对于 printf 函数需要实现的接口，我们的代码中将它们实现如下：

```
/* 不使用半主机模式，至少需要重定义 _ttywrch\ _sys_exit\ _sys_command_string 函数，以同时兼容 AC6 和 AC5 模式 */
int _ttywrch(int ch)
{
    ch = ch;
    return ch;
}
/* 定义 _sys_exit() 以避免使用半主机模式 */
void _sys_exit(int x)
{
    x = x;
}
char *_sys_command_string(char *cmd, int len)
{
    return NULL;
}
```

fputc 的重定向和之前一样，重定向到串口 1 即可，如果硬件资源允许，读者有特殊需求，也可以重定向到 LCD 或者其它串口。

```
/* 重定义 fputc 函数，printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while ((USART_UX->SR & 0x40) == 0); /* 等待上一个字符发送完成 */
    USART_UX->DR = (uint8_t)ch; /* 将要发送的字符 ch 写入到 DR 寄存器 */
    return ch;
}
```

第二篇 入门篇

功夫不负有心人，相信学习至此你已经掌握了基础篇介绍的知识。我们希望通过前面的章节你已经掌握了 STM32 开发的工具和方法。本篇我们将和大家一起来学习 STM32 的一些基础外设，这些外设实际项目中经常会用到，希望大家认真学习和掌握，以便将来更好、更快的完成实际项目开发。

本篇将采取一章一实例的方式，介绍 STM32 常用外设的使用，通过本篇的学习，我们将带领大家进入 STM32 的精彩世界。

第十三章 跑马灯实验

本章将通过一个经典的跑马灯程序，带大家开启 STM32F103 之旅。通过本章的学习，我们将了解到 STM32F103 的 IO 口作为输出使用的方法。我们将通过代码控制开发板上的 LED 灯：LED0、LED1 交替闪烁，实现类似跑马灯的效果。

本章分为如下 4 个小节：

13.1 STM32F1 GPIO 简介

13.2 硬件设计

13.3 程序设计

13.4 下载验证

13.1 STM32F1 GPIO 简介

GPIO 是控制或者采集外部器件的信息的外设，即负责输入输出。它按组分配，每组 16 个 IO 口，组数视芯片而定。STM32F103ZET6 芯片是 144 脚的芯片，具有 GPIOA、GPIOB、GPIOC、GPIOD、GPIOE、GPIOF 和 GPIOG 七组 GPIO 口，共有 112 个 IO 口可供我们编程使用。这里重点说一下 STM32F103 的 IO 电平兼容性问题，STM32F103 的绝大部分 IO 口，都兼容 5V，至于到底哪些是兼容 5V 的，请看 STM32F103xE 的数据手册（注意是数据手册，不是中文参考手册），见表 5 大容量 STM32F103xx 引脚定义，凡是有 FT 标志的，都是兼容 5V 电平的 IO 口，可以直接接 5V 的外设（注意：如果引脚设置的是模拟输入模式，则不能接 5V！），凡是不带 FT 标志的，就建议大家不要接 5V 了，可能烧坏 MCU。

13.1.1 GPIO 功能模式

GPIO 有八种工作模式，分别是：

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

13.1.2 GPIO 基本结构分析

我们知道了 GPIO 有八种工作模式，具体这些模式是怎么实现的？下面我们通过 GPIO 的基本结构图来分别进行详细分析，先看看总的框图，如图 13.1.2.1 所示。

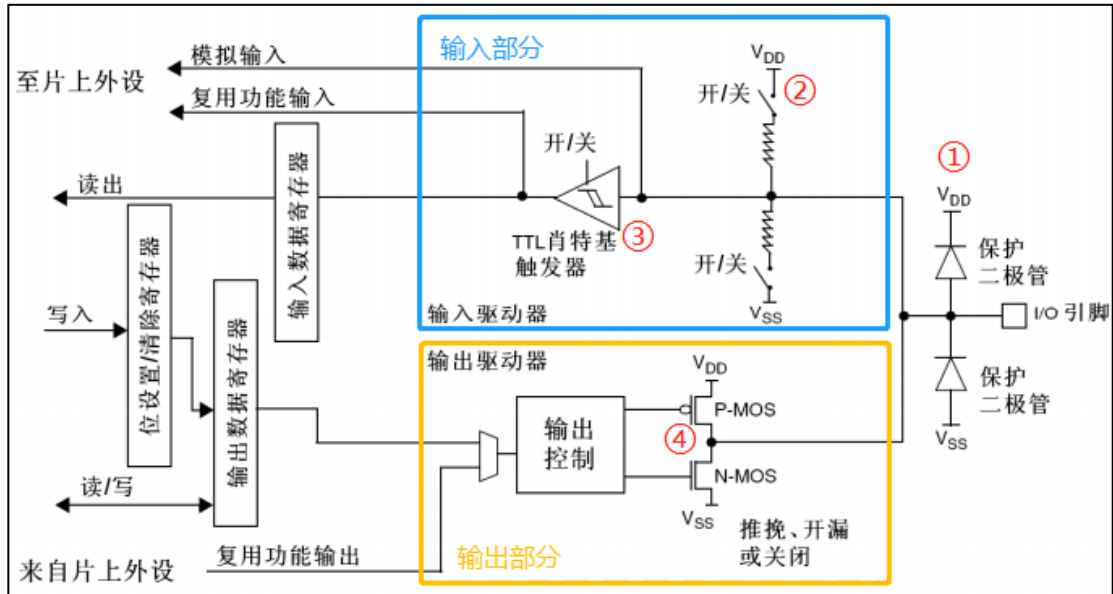


图 13.1.2.1 GPIO 的基本结构图

如上图所示，可以看到右边只有 I/O 引脚，这个 I/O 引脚就是我们可以看到的芯片实物的引脚，其他部分都是 GPIO 的内部结构。

① 保护二极管

保护二极管共有两个，用于保护引脚外部过高或过低的电压输入。当引脚输入电压高于 V_{DD} 时，上面的二极管导通，当引脚输入电压低于 V_{SS} 时，下面的二极管导通，从而使输入芯片内部的电压处于比较稳定的值。虽然有二极管的保护，但这样的保护却很有限，大电压大电流的接入很容易烧坏芯片。所以在实际的设计中我们要考虑设计引脚的保护电路。

② 上拉、下拉电阻

它们阻值大概在 30~50K 欧之间，可以通过上、下两个对应的开关控制，这两个开关由寄存器控制。当引脚外部的器件没有干扰引脚的电压时，即没有外部的上、下拉电压，引脚的电平由引脚内部上、下拉决定，开启内部上拉电阻工作，引脚电平为高，开启内部下拉电阻工作，则引脚电平为低。同样，如果内部上、下拉电阻都不开启，这种情况就是我们所说的浮空模式。浮空模式下，引脚的电平是不可确定的。引脚的电平可以由外部的上、下拉电平决定。需要注意的是，STM32 的内部上拉是一种“弱上拉”，这样的上拉电流很弱，如果有要求大电流还是得外部上拉。

③ 施密特触发器

对于标准施密特触发器，当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变，也就是说输出由高电平翻转为低电平，或是由低电平翻转为高电平对应的阈值电压是不同的。只有当输入电压发生足够的变化时，输出才会变化，因此将这种元件命名为触发器。这种双阈值动作被称为迟滞现象，表明施密特触发器有记忆性。从本质来说，施密特触发器是一种双稳态多谐振荡器。

施密特触发器可作为波形整形电路，能将模拟信号波形整形为数字电路能够处理的方波波形，而且由于施密特触发器具有滞回特性，所以可用于抗干扰，以及在闭回路正回授/负回授配置中用于实现多谐振荡器。

下面看看比较器跟施密特触发器的作用的比较，就清楚的知道施密特触发器对外部输入信号具有一定抗干扰能力，如图 13.1.2.2 所示。

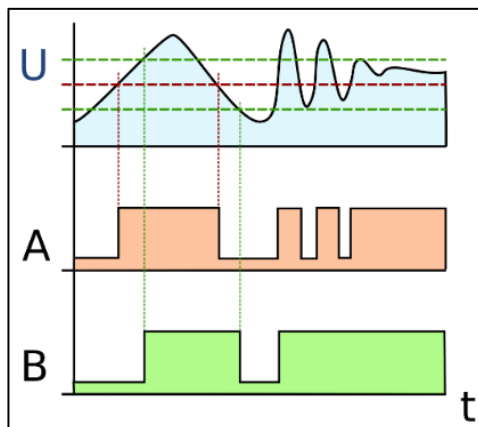


图 13.1.2.2 比较器的 (A) 和施密特触发器 (B) 作用比较

④ P-MOS 管和 N-MOS 管

这个结构控制 GPIO 的开漏输出和推挽输出两种模式。开漏输出：输出端相当于三极管的集电极，要得到高电平状态需要上拉电阻才行。推挽输出：这两只对称的 MOS 管每次只有一只导通，所以导通损耗小、效率高。输出既可以向负载灌电流，也可以从负载拉电流。推挽式输出既能提高电路的负载能力，又能提高开关速度。

上面我们对 GPIO 的基本结构图中的关键器件做了介绍，下面分别介绍 GPIO 八种工作模式对应结构图的工作情况。

1、输入浮空

输入浮空模式：上拉/下拉电阻为断开状态，施密特触发器打开，输出被禁止。输入浮空模式下，IO 口的电平完全是由外部电路决定。如果 IO 引脚没有连接其他的设备，那么检测其输入电平是不确定的。该模式可以用于按键检测等场景。

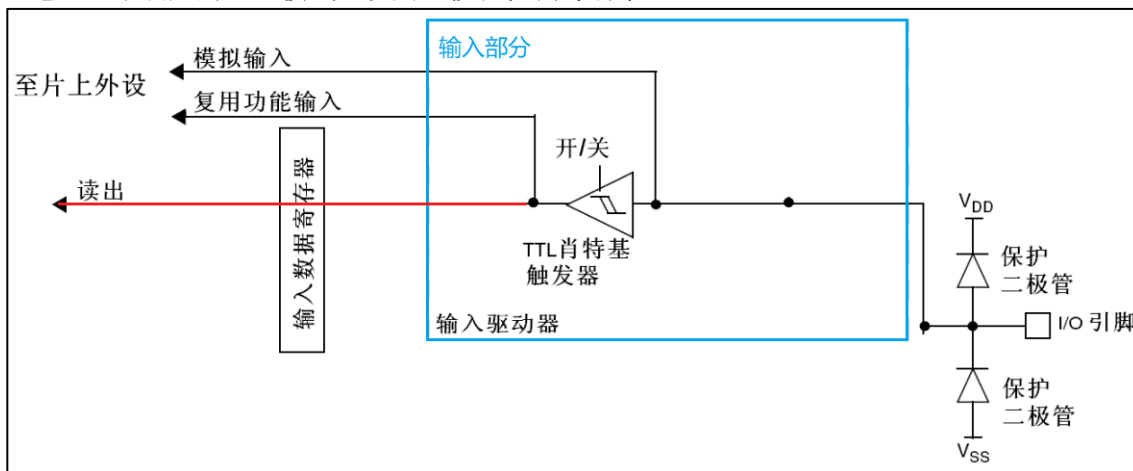


图 13.1.2.3 输入浮空模式

2、输入上拉

输入上拉模式：上拉电阻导通，施密特触发器打开，输出被禁止。在需要外部上拉电阻的时候，可以使用内部上拉电阻，这样可以节省一个外部电阻，但是内部上拉电阻的阻值较大，所以只是“弱上拉”，不适合做电流型驱动。

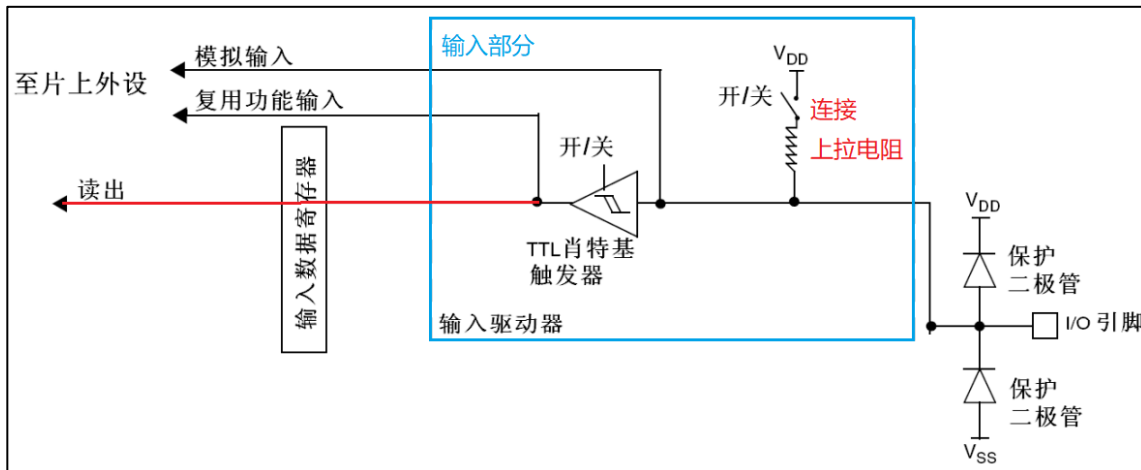


图 13.1.2.4 输入上拉模式

3、输入下拉

输入下拉模式：下拉电阻导通，施密特触发器打开，输出被禁止。在需要外部下拉电阻的时候，可以使用内部下拉电阻，这样可以节省一个外部电阻，但是内部下拉电阻的阻值较大，所以不适合做电流型驱动。

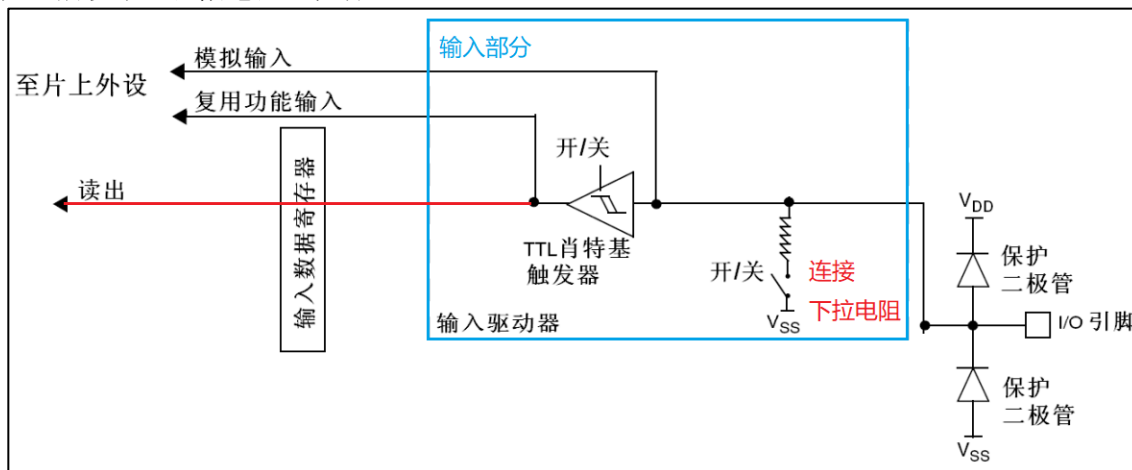


图 13.1.2.5 输入下拉模式

4、模拟功能

模拟功能：上下拉电阻断开，施密特触发器关闭，双 MOS 管也关闭。其他外设可以通过模拟通道输入输出。该模式下需要用到芯片内部的模拟电路单元单元，用于 ADC、DAC、MCO 这类操作模拟信号的外设。

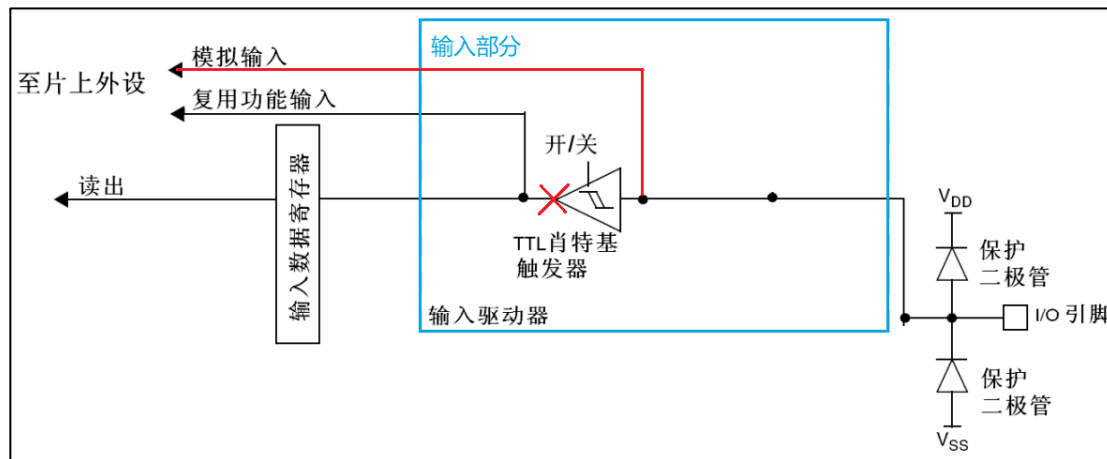


图 13.1.2.6 模拟功能

5、开漏输出

开漏输出模式：STM32 的开漏输出模式是数字电路输出的一种，从结果上看它只能输出低电平 V_{SS} 或者高阻态，常用于 IIC 通讯（IIC_SDA）或其它需要进行电平转换的场景。根据《STM32F10xxx 参考手册_V10（中文版）.pdf》第 108 页关于“GPIO 输出配置”的描述，我们可以知道开漏模式下，IO 是这样工作的：

- P-MOS 被“输出控制”控制在截止状态，因此 IO 的状态取决于 N-MOS 的导通状况；
- 只有 N-MOS 还受控制于输出寄存器，“输出控制”对输入信号进行了逻辑非的操作；
- 施密特触发器是工作的，即可以输入，且上下拉电阻都断开了，可以看成浮空输入；

根据参考手册的描述，同时为了方便大家理解，我们在“输出控制”部分做了等效处理，如图 13.1.2.7 所示。图 13.1.2.7 中写入输出数据寄存器①的值怎么对应到 IO 引脚的输出状态②是我们最关心的。

根据参考手册的描述：开漏输出模式下 P-MOS 一直在截止状态，即不导通，所以 P-MOS 管的栅极相当于一直接 V_{DD} 。如果输出数据寄存器①的值为 0，那么 IO 引脚的输出状态②为低电平，这是我们需要的控制逻辑，怎么做到的呢？是这样的，输出数据寄存器的逻辑 0 经过“输出控制”的取反操作后，输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS 管就会导通，使得 IO 引脚连接到 V_{SS} ，即输出低电平。如果输出数据寄存器的值为 1，经过“输出控制”的取反操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止。又因为 P-MOS 管是一直截止的，使得 IO 引脚呈现高阻态，即不输出低电平，也不输出高电平。因此要 IO 引脚输出高电平就必须接上拉电阻。又由于 F1 系列的开漏输出模式下，内部的上下拉电阻不可用，所以只能通过接芯片外部上拉电阻的方式，实现开漏输出模式下输出高电平。如果芯片外部不接上拉电阻，那么开漏输出模式下，IO 无法输出高电平。

在开漏输出模式下，施密特触发器是工作的，所以 IO 口引脚的电平状态会被采集到输入数据寄存器中，如果对输入数据寄存器进行读访问可以得到 IO 口的状态。也就是说开漏输出模式下，我们可以读取 IO 引脚状态。

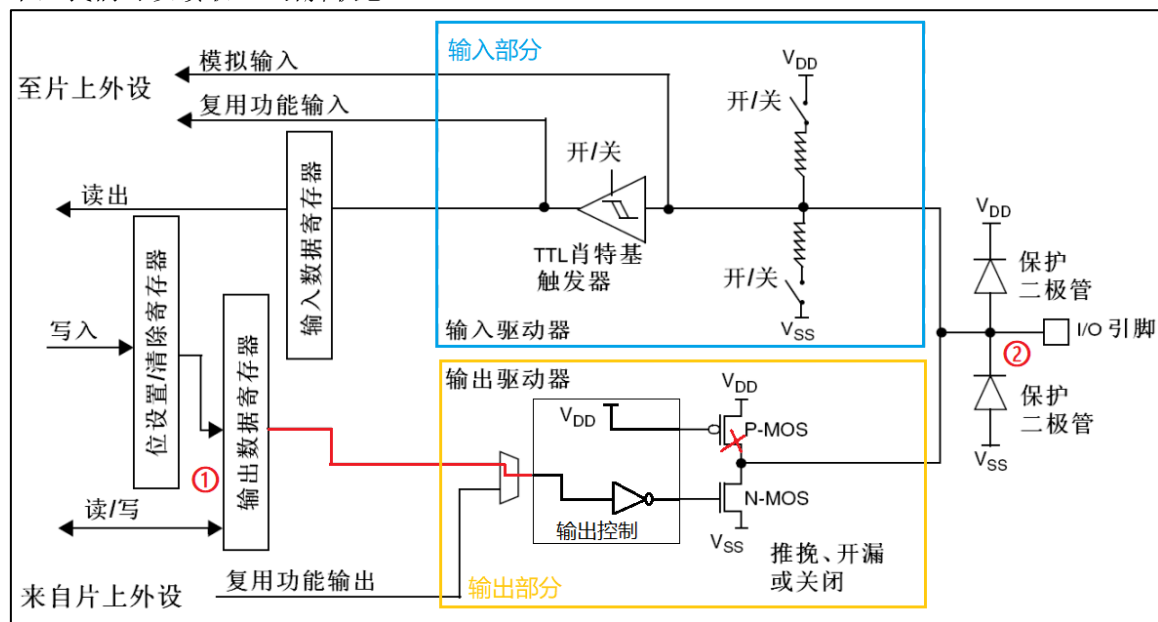


图 13.1.2.7 开漏输出模式

6、推挽输出

推挽输出模式：STM32 的推挽输出模式，从结果上看它会输出低电平 V_{SS} 或者高电平 V_{DD} 。推挽输出跟开漏输出不同的是，推挽输出模式 P-MOS 管和 N-MOS 管都用上。同样地，我们根据参考手册推挽模式的输出描述，可以得到等效原理图，如图 13.1.2.8 所示。根据手册描述可以把“输出控制”简单地等效为一个非门。

如果输出数据寄存器①的值为 0，经过“输出控制”取反操作后，输出逻辑 1 到 P-MOS 管的栅极，这时 P-MOS 管就会截止，同时也会输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS 管就会导通，使得 IO 引脚接到 V_{SS} ，即输出低电平。

如果输出数据寄存器的值为 1，经过“输出控制”取反操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止，同时也会输出逻辑 0 到 P-MOS 管的栅极，这时 P-MOS 管就会导通，使得 IO 引脚接到 V_{DD} ，即输出高电平。

由上述可知，推挽输出模式下，P-MOS 管和 N-MOS 管同一时间只能有一个管是导通的。当 IO 引脚在做高低电平切换时，两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都有较大的提高。

另外在推挽输出模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态。

由于推挽输出模式下输出高电平时，是直接连接 V_{DD} ，所以驱动能力较强，可以做电流型驱动，驱动电流最大可达 25mA，但是芯片的总电流有限，所以并不建议这样用，最好还是使用芯片外部的电源。

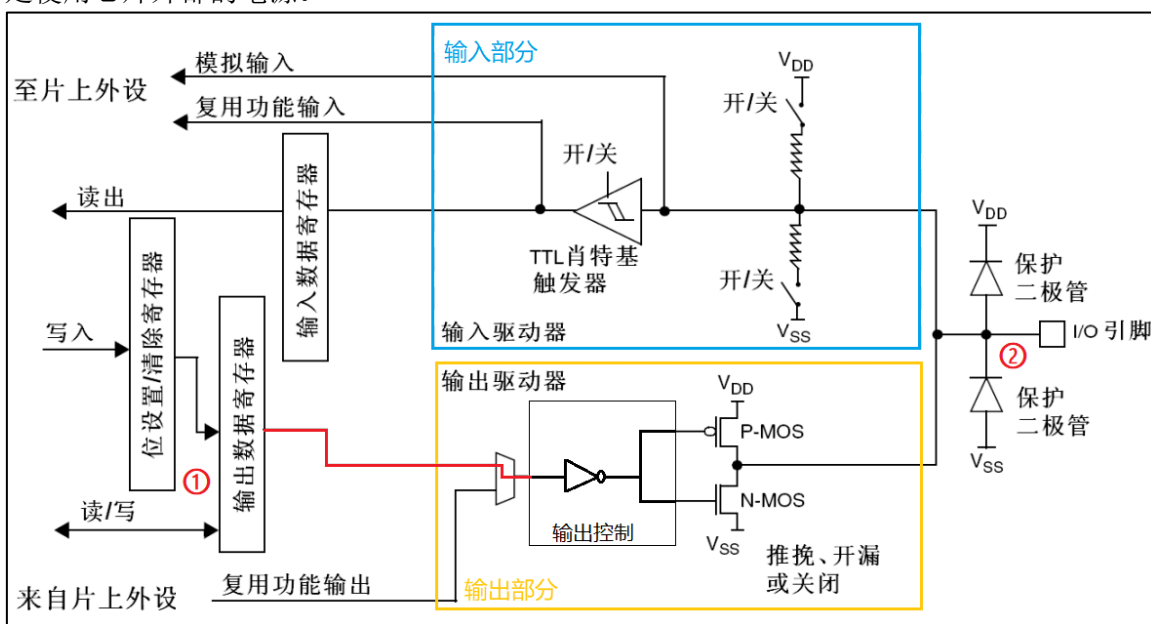


图 13.1.2.8 推挽输出模式

7、开漏式复用功能

开漏式复用功能：一个 IO 口可以是通用的 IO 口功能，还可以是其他外设的特殊功能引脚，这就是 IO 口的复用功能。一个 IO 口可以是多个外设的功能引脚，我们需要选择作为其中一个外设的功能引脚。当选择复用功能时，引脚的状态是由对应的外设控制，而不是输出数据寄存器。除了复用功能外，其他的结构分析请参考开漏输出模式。

另外在开漏式复用功能模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态，同时外设可以读取 IO 口的信息。

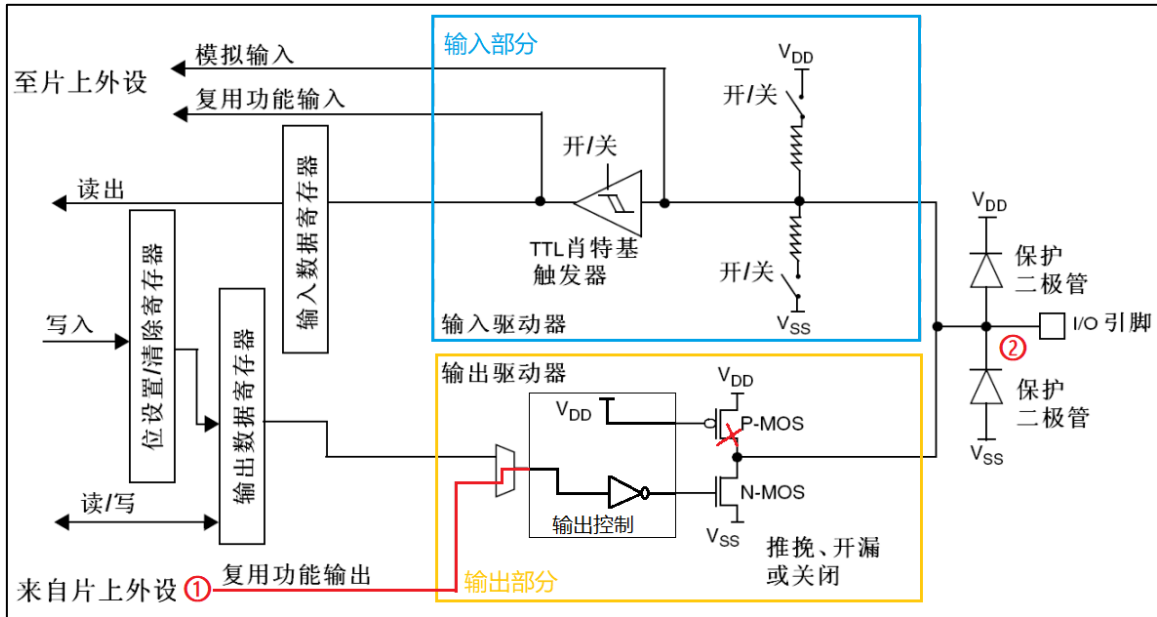


图 13.1.2.9 开漏式复用功能

8、推挽式复用功能

推挽式复用功能：复用功能介绍请查看开漏式复用功能，结构分析请参考推挽输出模式，这里不再赘述。

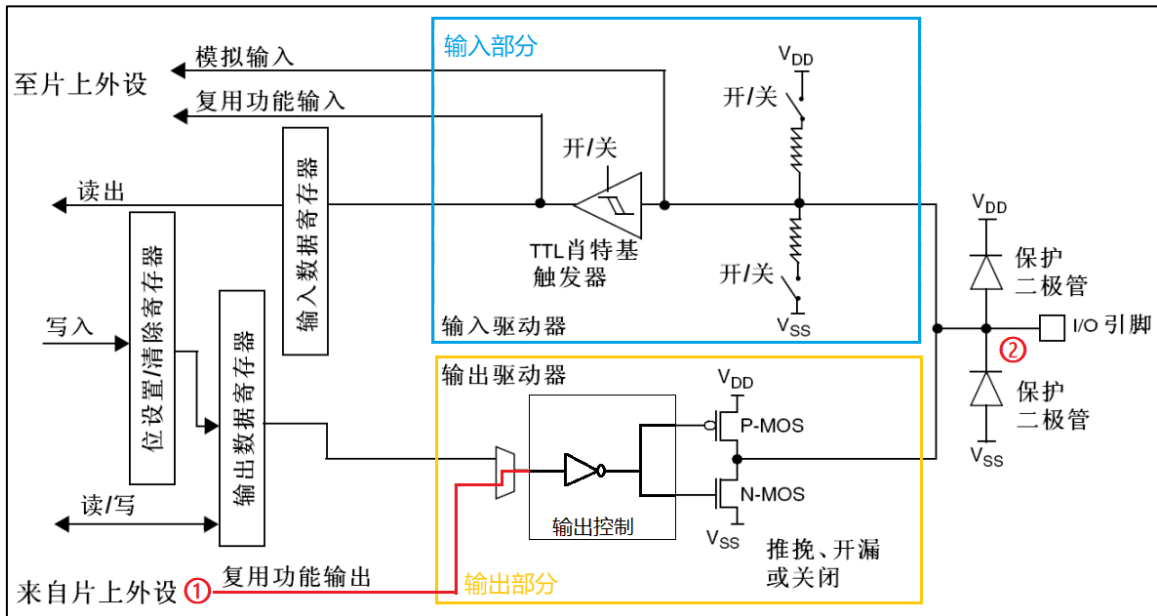


图 13.1.2.10 推挽式复用功能

13.1.3 GPIO 寄存器介绍

STM32F1 每组（这里是 A~D）通用 GPIO 口有 7 个 32 位寄存器控制，包括：

- 2 个 32 位端口配置寄存器（CRL 和 CRH）
- 2 个 32 位端口数据寄存器（IDR 和 ODR）
- 1 个 32 位端口置位/复位寄存器（BSRR）
- 1 个 16 位端口复位寄存器（BRR）
- 1 个 32 位端口锁定寄存器（LCKR）

下面我们将带大家理解本章用到的寄存器，没有介绍到的寄存器后面用到会继续介绍。这里主要是带大家学会怎么理解这些寄存器的方法，其他寄存器理解方法是一样的。因为寄存器

太多不可能一个个列出来讲，以后基本就是只会把重要的寄存器拿出来讲述，希望大家尽快培养自己学会看手册的能力。下面先看 GPIO 的 2 个 32 位配置寄存器：

● 端口配置寄存器（GPIO_x_CRL 和 GPIO_x_CRH）

这两个寄存器都是 GPIO 口配置寄存器，不过 CRL 控制端口的低八位，CRH 控制端口的高八位。寄存器的作用是控制 GPIO 口的工作模式和工作速度，寄存器描述如图 13.1.3.1 和图 13.1.3.2 所示。

| | | | | | | | | | | | | | | | |
|-----------|----|---|----|------------|----|----|----|-----------|----|----|----|------------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF7[1:0] | | | | MODE7[1:0] | | | | CNF6[1:0] | | | | MODE5[1:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF3[1:0] | | | | MODE3[1:0] | | | | CNF2[1:0] | | | | MODE1[1:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位31:30 | | CNFy[1:0]: 端口x配置位(y = 0...7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| 位29:28 | | MODEy[1:0]: 端口x的模式位(y = 0...7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式，最大速度10MHz 10: 输出模式，最大速度2MHz 11: 输出模式，最大速度50MHz | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

图 13.1.3.1 GPIO_x_CRL 寄存器描述

| | | | | | | | | | | | | | | | |
|------------|----|--|----|-------------|----|----|----|------------|----|----|----|-------------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| CNF15[1:0] | | | | MODE15[1:0] | | | | CNF14[1:0] | | | | MODE13[1:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CNF11[1:0] | | | | MODE11[1:0] | | | | CNF9[1:0] | | | | MODE9[1:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位31:30 | | CNFy[1:0]: 端口x配置位(y = 8...15) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00): 00: 模拟输入模式 01: 浮空输入模式(复位后的状态) 10: 上拉/下拉输入模式 11: 保留 在输出模式(MODE[1:0]>00): 00: 通用推挽输出模式 01: 通用开漏输出模式 10: 复用功能推挽输出模式 11: 复用功能开漏输出模式 | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| 位9:28 | | MODEy[1:0]: 端口x的模式位(y = 8...15) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00: 输入模式(复位后的状态) 01: 输出模式，最大速度10MHz 10: 输出模式，最大速度2MHz 11: 输出模式，最大速度50MHz | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

图 13.1.3.2 GPIO_x_CRH 寄存器描述

每组 GPIO 下有 16 个 IO 口，一个寄存器共 32 位，每 4 个位控制 1 个 IO，所以才需要两个寄存器完成。我们看看这个寄存器的复位值，然后用复位值举例说明一下这样的配置值代表

什么意思。比如 GPIOA_CRL 的复位值是 0x44444444，4 位为一个单位都是 0100，以寄存器低四位说明一下，首先位 1: 0 为 00 即是设置为 PA0 为输入模式，位 3: 2 为 01 即设置为浮空输入模式。所以假如 GPIOA_CRL 的值是 0x44444444，那么 PA0~PA7 都是设置为输入模式，而且是浮空输入模式。

上面这 2 个配置寄存器就是用来配置 GPIO 的相关工作模式和工作速度，它们通过不同的配置组合方法，就决定我们所说的 8 种工作模式。下面，我们来列表阐述，如表 13.1.3.1 所示。

| 配置模式 | | CNF1 | CNF0 | MODE1 | MODE0 | PxODR寄存器 | | |
|--------|----------------|------|------|------------------------|-------|----------|--|-----|
| 通用输出 | 推挽(Push-Pull) | 0 | 0 | 01 10 11 见表18 | | 0 或 1 | | |
| | 开漏(Open-Drain) | | 1 | | | 0 或 1 | | |
| 复用功能输出 | 推挽(Push-Pull) | 1 | 0 | | | | | 不使用 |
| | 开漏(Open-Drain) | | 1 | | | | | 不使用 |
| 输入 | 模拟输入 | 0 | 0 | 00 | | | | 不使用 |
| | 浮空输入 | | 1 | | | | | 不使用 |
| | 下拉输入 | 1 | 0 | | | 0 | | |
| | 上拉输入 | | | | | 1 | | |

表 13.1.3.1 配置寄存器 4 个组合下的 8 种工作模式

因为本章需要 GPIO 作为输出口使用，所以我们再来看看端口输出数据寄存器。

● 端口输出数据寄存器 (ODR)

该寄存器用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.3 所示。

| | | | | | | | | | | | | | | | |
|-------|-------|---|-------|-------|-------|------|------|------|------|------|------|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位15:0 | | ODRy[15:0]: 端口输出数据(y = 0...15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注: 对GPIOx_BSRR(x = A...E), 可以分别地对各个ODR位进行独立的设置/清除。 | | | | | | | | | | | | | |

图 13.1.3.3 GPIOx_ODR 寄存器描述

该寄存器低 16 位有效，分别对应每一组 GPIO 的 16 个引脚。当 CPU 写访问该寄存器，如果对应的某位写 0(ODRy=0)，则表示设置该 IO 口输出的是低电平，如果写 1(ODRy=1)，则表示设置该 IO 口输出的是高电平，y=0~15。

此外，除了 ODR 寄存器，还有一个寄存器也是用于控制 GPIO 输出的，它就是 BSRR 寄存器。

● 端口置位/复位寄存器 (BSRR)

该寄存器也用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.4 所示。

| | | | | | | | | | | | | | | | |
|--------|------|---|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |
| 位31:16 | | BRy : 清除端口x的位y (y = 0...15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 清除对应的ODRy位为0 注: 如果同时设置了BSy和BRy的对应位, BSy位起作用。 | | | | | | | | | | | | | |
| 位15:0 | | BSy : 设置端口x的位y (y = 0...15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0: 对对应的ODRy位不产生影响 1: 设置对应的ODRy位为1 | | | | | | | | | | | | | |

图 13.1.3.4 GPIOx_BSRR 寄存器描述

为什么有了 ODR 寄存器, 还要这个 BSRR 寄存器呢? 我们先看看 BSRR 的寄存器描述, 首先 BSRR 是只写权限, 而 ODR 是可读可写权限。BSRR 寄存器 32 位有效, 对于低 16 位 (0-15), 我们往相应的位写 1(BSy=1), 那么对应的 IO 口会输出高电平, 往相应的位写 0(BSy=0), 对 IO 口没有任何影响, 高 16 位 (16-31) 作用刚好相反, 对相应的位写 1(BRy=1)会输出低电平, 写 0(BRy=0)没有任何影响, y=0~15。

也就是说, 对于 BSRR 寄存器, 你写 0 的话, 对 IO 口电平是没有任何影响的。我们要设置某个 IO 口电平, 只需要相关位设置为 1 即可。而 ODR 寄存器, 我们要设置某个 IO 口电平, 我们首先需要读出来 ODR 寄存器的值, 然后对整个 ODR 寄存器重新赋值来达到设置某个或者某些 IO 口的目的, 而 BSRR 寄存器直接设置即可, 这在多任务实时操作系统中作用很大。BSRR 寄存器还有一个好处, 就是 BSRR 寄存器改变引脚状态的时候, 不会被中断打断, 而 ODR 寄存器有被中断打断的风险。

13.2 硬件设计

1. 例程功能

LED 灯: DS0 和 DS1 每过 500ms 一次交替闪烁, 实现类似跑马灯的效果。

2. 硬件资源

1) LED 灯

DS0 - PB5

DS1 - PE5

3. 原理图

本章用到的硬件用到 LED 灯: DS0 和 DS1。电路在开发板上已经连接好了, 所以在硬件上不需要动任何东西, 直接下载代码就可以测试使用。其连接原理图如图 13.2.1 所示:

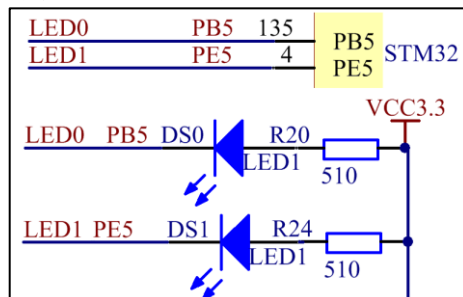


图 13.2.1 LED 与 STM32F103 连接原理图

13.3 程序设计

了解了 GPIO 的结构原理和寄存器，还有我们的实验功能，下面开始设计程序。

13.3.1 GPIO 的 HAL 库驱动分析

HAL 库中关于 GPIO 的驱动程序在 STM32F1xx_hal_gpio.c 文件以及其对应的头文件。

1. HAL_GPIO_Init 函数

要使用一个外设我们首先要对它进行初始化，所以我们先看外设 GPIO 的初始化函数。其声明如下：

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

● 函数描述：

用于配置 GPIO 功能模式，还可以设置 EXTI 功能。

● 函数形参：

形参 1 是端口号，可以有以下的选择：

```
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG ((GPIO_TypeDef *) GPIOG_BASE)
```

这是库里面的选择项，实际上我们的芯片只能从 GPIOA~GPIOE，因为我们只有 5 组 IO 口。

形参 2 是 GPIO_InitTypeDef 类型的结构体变量，其定义如下：

```
typedef struct
{
    uint32_t Pin;           /* 引脚号 */
    uint32_t Mode;          /* 模式设置 */
    uint32_t Pull;          /* 上拉下拉设置 */
    uint32_t Speed;         /* 速度设置 */
} GPIO_InitTypeDef;
```

该结构体很重要，下面对每个成员介绍一下。

成员 Pin 表示引脚号，范围：GPIO_PIN_0 到 GPIO_PIN_15，另外还有 GPIO_PIN_All 和 GPIO_PIN_MASK 可选。

成员 Mode 是 GPIO 的模式选择，有以下选择项：

```
#define GPIO_MODE_INPUT      (0x00000000U) /* 输入模式 */
#define GPIO_MODE_OUTPUT_PP (0x00000001U) /* 推挽输出 */
#define GPIO_MODE_OUTPUT_OD (0x00000011U) /* 开漏输出 */
#define GPIO_MODE_AF_PP     (0x00000002U) /* 推挽式复用 */
#define GPIO_MODE_AF_OD     (0x00000012U) /* 开漏式复用 */
#define GPIO_MODE_AF_INPUT  GPIO_MODE_INPUT

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

#define GPIO_MODE_IT_RISING  (0x11110000U) /* 外部中断, 上升沿触发检测 */
#define GPIO_MODE_IT_FALLING (0x11210000U) /* 外部中断, 下降沿触发检测 */
/* 外部中断, 上升和下降双沿触发检测 */
#define GPIO_MODE_IT_RISING_FALLING (0x11310000U)

#define GPIO_MODE_EVT_RISING (0x11120000U) /* 外部事件, 上升沿触发检测 */
#define GPIO_MODE_EVT_FALLING (0x11220000U) /* 外部事件, 下降沿触发检测 */
/* 外部事件, 上升和下降双沿触发检测 */
#define GPIO_MODE_EVT_RISING_FALLING (0x11320000U)
```

成员 Pull 用于配置上下拉电阻，有以下选择项：

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

成员 Speed 用于配置 GPIO 的速度，有以下选择项：

```
#define GPIO_SPEED_FREQ_LOW      (0x00000002U) /* 低速 */
#define GPIO_SPEED_FREQ_MEDIUM   (0x00000001U) /* 中速 */
#define GPIO_SPEED_FREQ_HIGH     (0x00000003U) /* 高速 */
```

- 函数返回值：

无

- 注意事项：

HAL 库的 EXTI 外部中断的设置功能整合到此函数里面，而不是单独独立一个文件。这个我们到外部中断实验再细讲。

2. HAL_GPIO_WritePin 函数

HAL_GPIO_WritePin 函数是 GPIO 口的写引脚函数。其声明如下：

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

- 函数描述：

用于设置引脚输出高电平或者低电平，通过 BSRR 寄存器复位或者置位操作。

- 函数形参：

形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

形参 3 是要设置输出的状态，是枚举型有两个选择：GPIO_PIN_SET 表示高电平，GPIO_PIN_RESET 表示低电平。

- 函数返回值：

无

3. HAL_GPIO_TogglePin 函数

HAL_GPIO_TogglePin 函数是 GPIO 口的电平翻转函数。其声明如下：

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

- 函数描述：

用于设置引脚的电平翻转，也是通过 BSRR 寄存器复位或者置位操作。

- 函数形参：

形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

- 函数返回值：

无

本实验我们用到上面三个函数，其他的 API 函数后面用到再进行讲解。

GPIO 输出配置步骤

1) 使能对应 GPIO 时钟

STM32 在使用任何外设之前，我们都要先使能其时钟（下同）。本实验用到 PB5 和 PE5 两个 IO 口，因此需要先使能 GPIOB 和 GPIOE 的时钟，代码如下：

```
HAL_RCC_GPIOB_CLK_ENABLE();
HAL_RCC_GPIOE_CLK_ENABLE();
```

2) 设置对应 GPIO 工作模式（推挽输出）

本实验 GPIO 使用推挽输出模式，控制 LED 亮灭，通过函数 HAL_GPIO_Init 设置实现。

3) 控制 GPIO 引脚输出高低电平

在配置好 GPIO 工作模式后，我们就可以通过 HAL_GPIO_WritePin 函数控制 GPIO 引脚输出高低电平，从而控制 LED 的亮灭了。

13.3.2 程序流程图

程序流程图能帮助我们更好的理解一个工程的功能和实现的过程，对学习和设计工程有很好的主导作用。本实验的程序流程图如下：

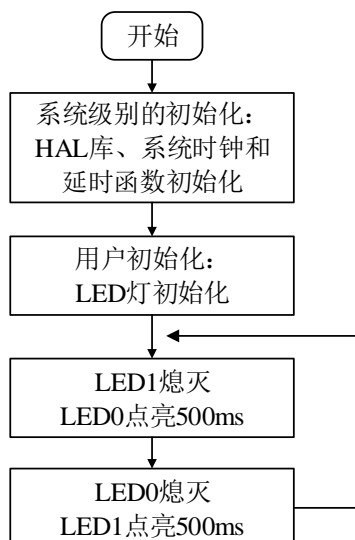


图 13.3.2.1 跑马灯实验程序流程图

13.3.3 程序解析

1. led 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。LED 驱动源码包括两个文件：led.c 和 led.h（正点原子编写的外设驱动基本都是包含一个.c 文件和一个.h 文件，下同）。

下面我们先解析 led.h 的程序，我们把它分两部分功能进行讲解。

● LED 灯引脚宏定义

由硬件设计小节，我们知道 LED 灯在硬件上分别连接到 PB5 和 PE5，再结合 HAL 库，我们做了下面的引脚定义。

```

/* LED0 引脚定义 */
#define LED0_GPIO_PORT      GPIOB
#define LED0_GPIO_PIN      GPIO_PIN_5
#define LED0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)
/* LED1 引脚定义 */
#define LED1_GPIO_PORT      GPIOE
#define LED1_GPIO_PIN      GPIO_PIN_5
#define LED1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
  
```

这样的好处是进一步隔离底层函数操作，移植更加方便，函数命名更亲近实际的开发板。比如：当我们看到 LED0_GPIO_PORT 这个宏定义，我们就知道这是灯 LED0 的端口号；看到 LED0_GPIO_PIN 这个宏定义，就知道这是灯 LED0 的引脚号；看到 LED0_GPIO_CLK_ENABLE 这个宏定义，就知道这是灯 LED0 的时钟使能函数。大家后面学习时间长了就会慢慢熟悉这样的命名方式。

特别注意：这里的时钟使能函数宏定义，使用了 do{ }while(0)结构，是为了避免在某些使用场景出错的问题（下同），详见《嵌入式单片机 C 代码规范与风格》第六章第 2 点。

__HAL_RCC_GPIOx_CLK_ENABLE 函数是 HAL 库的 IO 口时钟使能函数，x=A 到 G。

● LED 灯操作函数宏定义

为了后续对 LED 灯进行便捷的操作，我们为 LED 灯操作函数做了下面的定义。

```

/* LED 端口操作定义 */
#define LED0(x) do{ x ? \
    HAL_GPIO_WritePin(LED0_GPIO_PORT,LED0_GPIO_PIN, GPIO_PIN_SET) : \
    HAL_GPIO_WritePin(LED0_GPIO_PORT,LED0_GPIO_PIN, GPIO_PIN_RESET);\
}while(0) /* LED0 翻转 */
#define LED1(x) do{ x ? \
    HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_SET) : \
    HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_RESET);\
}
  
```

```

        }while(0)          /* LED1 翻转 */
/* LED 电平翻转定义 */
#define LED0_TOGGGLE()    do{ HAL_GPIO_TogglePin(LED0_GPIO_PORT,
                           LED0_GPIO_PIN); }while(0) /* LED0 = !LED0 */
#define LED1_TOGGGLE()    do{ HAL_GPIO_TogglePin(LED1_GPIO_PORT,
                           LED1_GPIO_PIN); }while(0) /* LED1 = !LED1 */

```

LED0 和 LED1 这两个宏定义，分别是控制 LED0 和 LED1 的亮灭。例如：对于宏定义标识符 LED0(x)，它的值是通过条件运算符来确定：

当 x=0 时，宏定义的值为 HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_RESET)，也就是设置 LED0_GPIO_PORT(PB5)输出低电平；

当 x!=0 时，宏定义的值为 HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_SET)，也就是设置 LED0_GPIO_PORT(PB5)输出高电平。

根据前述定义，如果要设置 LED0 输出低电平，那么调用宏定义 LED0(0)即可，如果要设置 LED0 输出高电平，调用宏定义 LED0(1)即可。宏定义 LED1(x)同理。

LED0_TOGGGLE 和 LED1_TOGGGLE 这三个宏定义，分别是控制 LED0 和 LED1 的翻转。这里利用 HAL_GPIO_TogglePin 函数实现 IO 口输出电平翻转操作。

下面我们再解析 led.c 的程序，这里只有一个函数 led_init，这是 LED 灯的初始化函数，其定义如下：

```

/**
 * @brief      初始化 LED 相关 IO 口，并使能时钟
 * @param      无
 * @retval     无
 */
void led_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    LED0_GPIO_CLK_ENABLE();          /* LED0 时钟使能 */
    LED1_GPIO_CLK_ENABLE();          /* LED1 时钟使能 */
    gpio_init_struct.Pin = LED0_GPIO_PIN;          /* LED0 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;    /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;           /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct); /* 初始化 LED0 引脚 */

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

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

```

对 LED 灯的两个引脚都设置为中速上拉的推挽输出。最后关闭 LED 灯的输出，防止没有操作就亮了。

2. main.c 代码

在 main.c 里面编写如下代码：

```

#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/usart/usart.h"
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LED/led.h"

int main(void)
{
    HAL_Init();          /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);       /* 延时初始化 */
    led_init();           /* 初始化 LED */
}

```

```
while (1)
{
    LED0(0);                /* LED0 灭 */
    LED1(1);                /* LED1 亮 */
    delay_ms(500);
    LED1(1);                /* LED0 灭 */
    LED0(0);                /* LED1 亮 */
    delay_ms(500);
}
```

首先是调用系统级别的初始化:初始化 HAL 库、系统时钟和延时函数。接下来,调用 led_init 来初始化 LED 灯。最后在无限循环里面实现 LED0 和 LED1 间隔 500ms 交替闪烁一次。

13.4 下载验证

我们先来看看编译结果,如图 13.4.1 所示。

```
Build Output
compiling stm32flxx_hal_rcc.c...
compiling stm32flxx_hal_uart.c...
compiling stm32flxx_hal_usart.c...
compiling led.c...
linking...
Program Size: Code=5442 RO-data=362 RW-data=28 ZI-data=1900
FromELF: creating hex file...
"..\\..\\Output\\atk_fl03.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:02
```

图 13.4.1 编译结果

可以看到没有 0 错误, 0 警告。从编译信息可以看出, 我们的代码占用 FLASH 大小为: 5804 字节 (5442+362+28), 所用的 SRAM 大小为: 1928 个字节 (28+1900)。这里我们解释一下, 编译结果里面的几个数据的意义:

Code: 表示程序所占用 FLASH 的大小 (FLASH)。

RO-data: 即 Read Only-data, 表示程序定义的常量 (FLASH)。

RW-data: 即 Read Write-data, 表示已被初始化的变量 (FLASH + RAM)

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

有了这个就可以知道你当前使用的 flash 和 ram 大小了, 所以, 一定要注意的是程序的大小不是 .hex 文件的大小, 而是编译后的 Code 和 RO-data 之和。接下来, 大家就可以下载验证了。这里我们使用 DAP 仿真器 (也可以使用其他调试器) 下载。

下载完之后, 运行结果如图 13.4.2 所示, 可以看到 LED 灯的 LED0 和 LED1 交替亮。



图 13.4.2 程序运行结果

至此, 我们的跑马灯实验的学习就结束了, 本章介绍了 STM32F103 的 IO 口的使用及注意事项, 是后面学习的基础, 希望大家好好理解。

第十四章 蜂鸣器实验

上一章，我们介绍了 STM32F1 的 IO 口作为输出的使用。本章，我们将通过另外一个例子继续讲述 STM32F1 的 IO 口作为输出的使用，不同的是本章讲的不是用 IO 口直接驱动器件，而是通过三极管间接驱动。我们将利用一个 IO 口来控制板载的有源蜂鸣器。

本章分为如下几个小节：

- 14.1 蜂鸣器简介
- 14.2 硬件设计
- 14.3 程序设计
- 14.4 下载验证

14.1 蜂鸣器简介

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

STM32F103 战舰开发板板载的蜂鸣器是电磁式的有源蜂鸣器，如图 14.1.1 所示：



图 14.1.1 有源蜂鸣器

这里的有源不是指电源的“源”，而是指有没有自带震荡电路，有源蜂鸣器自带了震荡电路，一通电就会发声；无源蜂鸣器则没有自带震荡电路，必须外部提供 2~5KHz 左右的方波驱动，才能发声。

上一章，我们利用 STM32 的 IO 口直接驱动 LED 灯，本章的蜂鸣器，我们能否直接用 STM32 的 IO 口驱动呢？让我们来分析一下：STM32F1 的单个 IO 最大可以提供 25mA 电流（来自数据手册），而蜂鸣器的驱动电流是 30mA 左右，两者十分相近，但是全盘考虑，STM32F1 整个芯片的电流，最大也就 150mA，如果用 IO 口直接驱动蜂鸣器，其他地方用电就得省着点了，所以我们不用 STM32F1 的 IO 直接驱动蜂鸣器，而是通过三极管扩流后再驱动蜂鸣器，这样 STM32F1 的 IO 只需要提供不到 1mA 的电流就足够了。

IO 口使用虽然简单，但是和外部电路的匹配设计，还是要十分讲究的，考虑越多，设计就越可靠，可能出现的问题也就越少。

14.2 硬件设计

1. 例程功能

蜂鸣器每隔 300ms 响或者停一次。LED0 每隔 300ms 亮或者灭一次。LED0 亮的时候蜂鸣器不叫，而 LED0 熄灭的时候，蜂鸣器叫。

2. 硬件资源

- 1) LED 灯
LED - PB5
- 2) 蜂鸣器
BEEP - PB8

3. 原理图

蜂鸣器在硬件上是直接连接好了的，不需要经过任何设置，直接编写代码就可以了。蜂鸣器的驱动信号连接在 STM32F1 的 PB8 上。如图 14.2.1 所示：

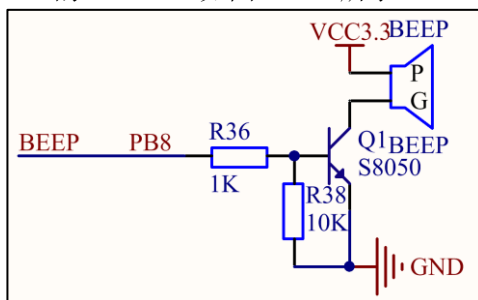


图 14.2.1 蜂鸣器与 STM32F1 连接原理图

我们用一个 NPN 三极管（S8050）来驱动蜂鸣器，驱动信号通过 R36 和 R38 间的电压获得，芯片上电时默认电平为低电平，故上电时蜂鸣器不会直接响起。当 PB8 输出高电平的时候，蜂鸣器将发声，当 PB8 输出低电平的时候，蜂鸣器停止发声。

14.3 程序设计

本实验我们只是用到 GPIO 外设输出功能，关于 HAL 库的 GPIO 的 API 函数请看跑马灯实验的介绍。下面我们直接分析本实验的程序流程图。

14.3.1 程序流程图

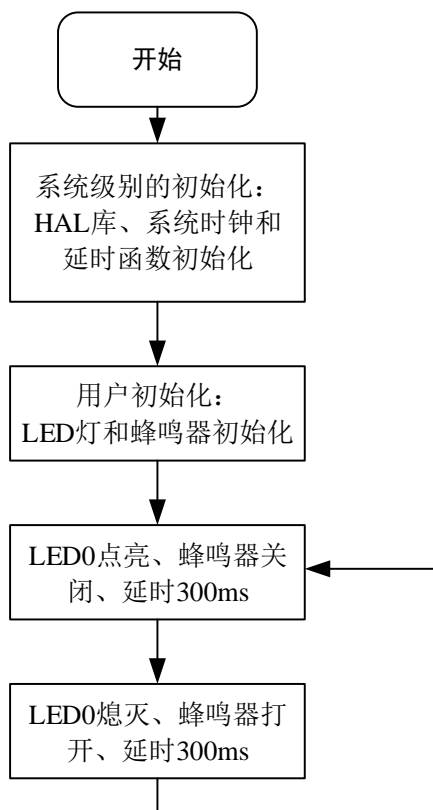


图 14.3.1.1 蜂鸣器实验程序流程图

14.3.2 程序解析

1. 蜂鸣器驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。蜂鸣器（BEEP）驱动源码包括两个文件：beep.c 和 beep.h。

下面我们先解析 beep.h 的程序，我们把它分两部分功能进行讲解。

● 蜂鸣器引脚定义

由硬件设计小节，我们知道驱动蜂鸣器的三极管在硬件上连接到 PB8，类似跑马灯实验，我们做了下面的引脚定义。

```
/* 引脚 定义 */
#define BEEP_GPIO_PORT      GPIOB
#define BEEP_GPIO_PIN       GPIO_PIN_8
/* PB 口时钟使能 */
#define BEEP_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)
```

● 蜂鸣器操作函数定义

为了后续对蜂鸣器进行便捷的操作，我们为蜂鸣器操作函数做了下面的定义。

```
/* 蜂鸣器控制 */
#define BEEP(x)              do{ x ? \
                              HAL_GPIO_WritePin(BEEP_GPIO_PORT,BEEP_GPIO_PIN,GPIO_PIN_SET):\
                              HAL_GPIO_WritePin(BEEP_GPIO_PORT, BEEP_GPIO_PIN,GPIO_PIN_RESET);\
                              }while(0)
/* BEEP 状态翻转 */
#define BEEP_TOGGLE()       do{HAL_GPIO_TogglePin(BEEP_GPIO_PORT,BEEP_GPIO_PIN);\
                              }while(0)
```

BEEP(x)这个宏定义就是控制蜂鸣器的打开和关闭的。例如：如果要打开蜂鸣器，那么调用宏定义 BEEP(1)即可，如果要关闭蜂鸣器，调用宏定义 BEEP(0)即可。

BEEP_TOGGLE()是控制蜂鸣器进行翻转的。这里也利用 HAL_GPIO_TogglePin 函数实现 IO 口输出电平取反操作。

下面我们再解析 beep.c 的程序，这里只有一个函数 beep_init，这是蜂鸣器的初始化函数，其定义如下：

```
/**
 * @brief      初始化 BEEP 相关 IO 口，并使能时钟
 * @param      无
 * @retval     无
 */
void beep_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    BEEP_GPIO_CLK_ENABLE(); /* BEEP 时钟使能 */

    gpio_init_struct.Pin = BEEP_GPIO_PIN; /* 蜂鸣器引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(BEEP_GPIO_PORT, &gpio_init_struct); /* 初始化蜂鸣器引脚 */

    BEEP(0); /* 关闭蜂鸣器 */
}
```

对蜂鸣器的控制引脚模式设置为高速上拉的推挽输出。最后关闭蜂鸣器，防止没有操作就响了。

2. main.c 代码

在 main.c 里面编写如下代码：

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

```
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LED/led.h"
#include "../BSP/BEEP/beep.h"

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72M */
    delay_init(72); /* 初始化延时函数 */
    led_init(); /* 初始化 LED */
    beep_init(); /* 初始化蜂鸣器 */

    while (1)
    {
        LED0(0);
        BEEP(0);
        delay_ms(300);
        LED0(1);
        BEEP(1);
        delay_ms(300);
    }
}
```

首先初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 led 灯，调用 beep_init 函数初始化蜂鸣器。最后在无限循环里面实现 LED0 和蜂鸣器间隔 300ms 交替闪烁和打开关闭一次。

14.4 下载验证

下载完之后，可以看到 LED0 亮的时候蜂鸣器不叫，而 LED0 熄灭的时候，蜂鸣器叫（因为他们的有效信号相反）。间隔为 0.3 秒左右，符合预期设计。

至此，本章的学习就结束了。通过本章，我们进一步学习 IO 作为输出的使用方法，同时巩固了前面知识的学习。希望大家在开发板上实际验证一下，从而加深印象。

第十五章 按键输入实验

上一章，我们介绍了 STM32F1 的 IO 口作为输出的使用。本章，我们将向大家介绍如何使用 STM32F1 的 IO 口作为输入。我们将利用板载的 3 个按键，来控制板载的两个 LED 灯亮灭。通过本章的学习，我们将了解到 STM32F1 的 IO 口作为输入的使用方法。

本章分为如下几个小节

15.1 按键与输入数据寄存器

15.2 硬件设计

15.3 程序设计

15.4 下载验证

15.1 按键与输入数据寄存器简介

15.1.1 独立按键简介

几乎每个开发板都会板载有独立按键，因为按键用处很多。常态下，独立按键是断开的，按下的时候才闭合。每个独立按键会单独占用一个 IO 口，通过 IO 口的高低电平判断按键的状态。但是按键在闭合和断开的时候，都存在抖动现象，即按键在闭合时不会马上就稳定的连接，断开时也不会马上断开。这是机械触点，无法避免。独立按键抖动波形图如下：

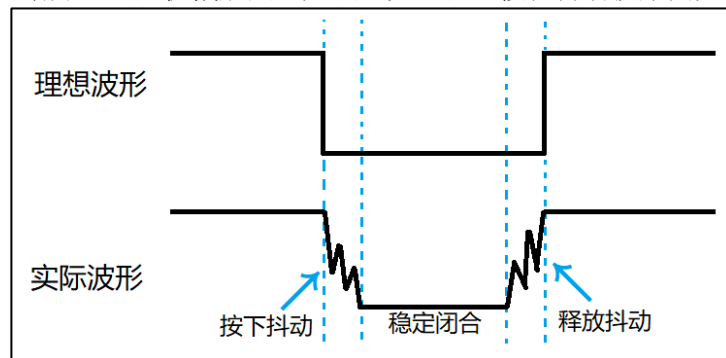


图 15.1.1.1 独立按键抖动波形图

图中的按下抖动和释放抖动的时间一般为 5~10ms，如果在抖动阶段采样，其不稳定状态可能出现一次按键动作被认为是多次按下的情况。为了避免抖动可能带来的误操作，我们要做的措施就是给按键消抖（即采样稳定闭合阶段）。消抖方法分为硬件消抖和软件消抖，我们常用软件的方法消抖。

软件消抖：方法很多，我们例程中使用最简单的延时消抖。检测到按键按下后，一般进行 10ms 延时，用于跳过抖动的时间段，如果消抖效果不好可以调整这个 10ms 延时，因为不同类型的按键抖动时间可能有偏差。待延时过后再检测按键状态，如果没有按下，那我们就判断这是抖动或者干扰造成的；如果还是按下，那么我们就认为这是按键真的按下了。对按键释放的判断同理。

硬件消抖：利用 RC 电路的电容充放电特性来对抖动产生的电压毛刺进行平滑出来，从而实现消抖，但是成本会更高一点，本着能省则省的原则，我们推荐使用软件消抖即可。

15.1.2 GPIO 端口输入数据寄存器（IDR）

本实验我们将会用到 GPIO 端口输入数据寄存器，下面来介绍一下。

该寄存器用于存储 GPIOx 的输入状态，它连接到施密特触发器上，IO 口外部的电平信号经过触发器后，模拟信号就被转化成 0 和 1 这样的数字信号，并存储到该寄存器中。寄存器描述如图 15.1.2.1 所示。

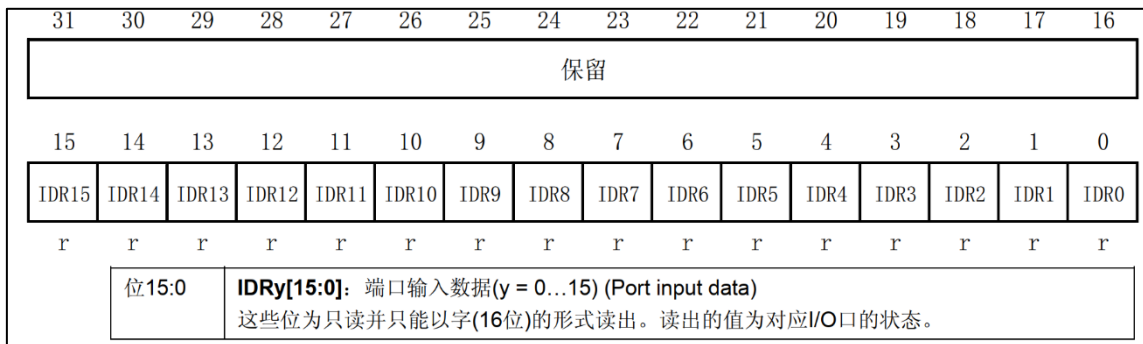


图 15.1.2.1 GPIOx IDR 寄存器描述

该寄存器低 16 位有效，分别对应每一组 GPIO 的 16 个引脚。当 CPU 访问该寄存器，如果对应的某位为 0(IDRy=0)，则说明该 IO 口输入的是低电平，如果是 1(IDRy=1)，则表示输入的是高电平，y=0~15。

15.2 硬件设计

1. 例程功能

通过开发板上的三个独立按键控制 LED 灯：KEY_UP 控制蜂鸣器翻转，KEY1 控制 LED1 翻转，KEY2 控制 LED0 翻转，KEY0 控制 LED0/LED1 同时翻转。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 独立按键

KEY0 - PE4

KEY1 - PE3

KEY2 - PE2

KEY_UP - PA0

3. 原理图

独立按键硬件部分的原理图，如图 15.2.1 所示：

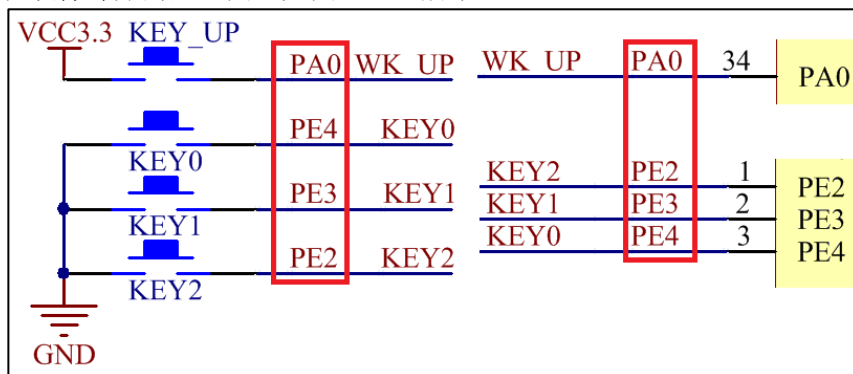


图 15.2.1 独立按键与 STM32F1 连接原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 是低电平有效的，而 KEY_UP 则是高电平有效的，并且外部都没有上下拉电阻，所以需要在 STM32F103 内部设置上下拉，来确定设置空闲电平状态。

15.3 程序设计

15.3.1 HAL_GPIO_ReadPin 函数

HAL_GPIO_ReadPin 函数是 GPIO 口的读引脚函数。其声明如下：

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

- **函数描述：**
用于读取 GPIO 引脚状态，通过 IDR 寄存器读取。
- **函数形参：**
形参 1 是端口号，可以选择范围：GPIOA~GPIOG。
形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。
- **函数返回值：**
引脚状态值 0 或者 1

GPIO 输入配置步骤

1) 使能对应 GPIO 时钟

本实验用到 PA0 和 PE2/3/4 等四个 IO 口，因此需要先使能 GPIOA 和 GPIOE 的时钟，代码如下：

```
HAL_RCC_GPIOA_CLK_ENABLE();  
HAL_RCC_GPIOE_CLK_ENABLE();
```

2) 设置对应 GPIO 工作模式（上拉/下拉输入）

本实验 GPIO 使用输入模式（带上拉/下拉），从而可以读取 IO 口的状态，实现按键检测，GPIO 模式通过函数 HAL_GPIO_Init 设置实现。

3) 读取 GPIO 引脚高低电平

在配置好 GPIO 工作模式后，我们就可以通过 HAL_GPIO_ReadPin 函数读取 GPIO 引脚的高低电平，从而实现按键检测了。

15.3.2 程序流程图

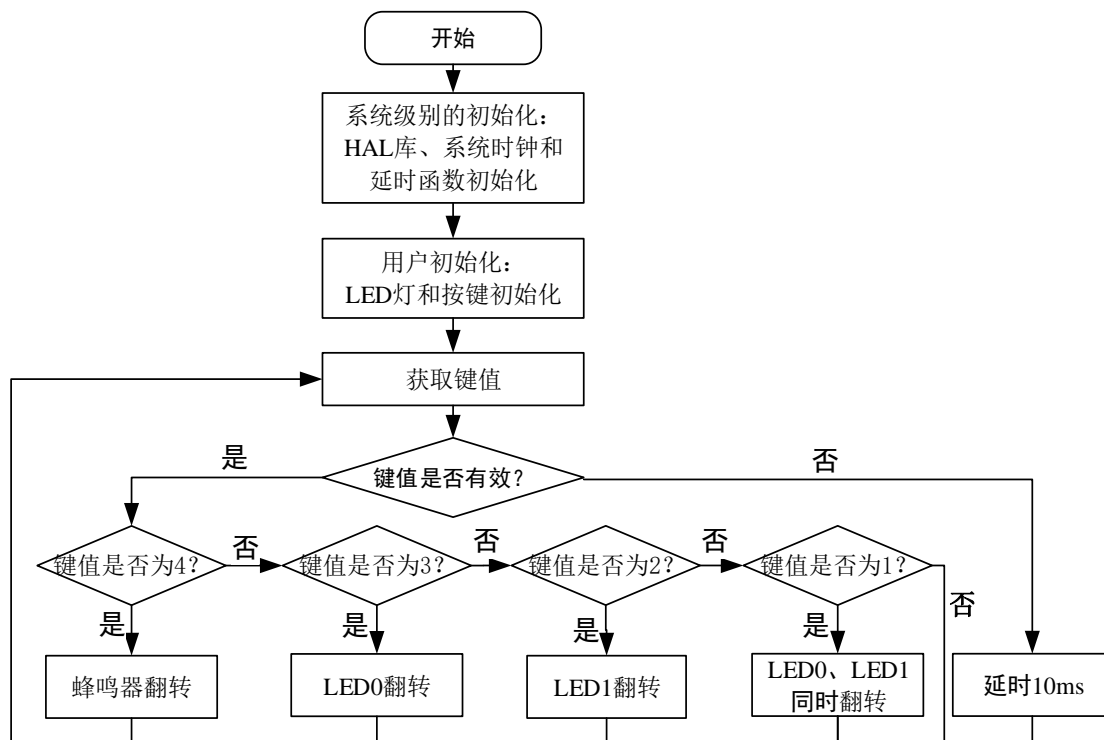


图 15.3.2.1 按键输入实验程序流程图

15.3.3 程序解析

1. 按键驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。按键（KEY）驱动源码包括两个文件：key.c 和 key.h。

下面我们先解析 key.h 的程序，我们把它分两部分功能进行讲解。

● 按键引脚定义

由硬件设计小节，我们知道 KEY0、KEY1、KEY2 和 KEY_UP 分别来连接到 PE4、PE3、PE2 和 PA0 上，我们做了下面的引脚定义。

```
/* 引脚 定义 */
#define KEY0_GPIO_PORT      GPIOE
#define KEY0_GPIO_PIN      GPIO_PIN_4
/* PE 口时钟使能 */
#define KEY0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define KEY1_GPIO_PORT      GPIOE
#define KEY1_GPIO_PIN      GPIO_PIN_3
/* PE 口时钟使能 */
#define KEY1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define KEY2_GPIO_PORT      GPIOE
#define KEY2_GPIO_PIN      GPIO_PIN_2
/* PE 口时钟使能 */
#define KEY2_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define WKUP_GPIO_PORT      GPIOA
#define WKUP_GPIO_PIN      GPIO_PIN_0
/* PA 口时钟使能 */
#define WKUP_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)
```

● 按键操作函数定义

为了后续对按键进行便捷的操作，我们为按键操作函数做了下面的定义。

```
#define KEY0 HAL_GPIO_ReadPin(KEY0_GPIO_PORT, KEY0_GPIO_PIN) /* 读取 KEY0 引脚 */
#define KEY1 HAL_GPIO_ReadPin(KEY1_GPIO_PORT, KEY1_GPIO_PIN) /* 读取 KEY1 引脚 */
#define KEY2 HAL_GPIO_ReadPin(KEY2_GPIO_PORT, KEY2_GPIO_PIN) /* 读取 KEY2 引脚 */
#define WK_UP HAL_GPIO_ReadPin(WKUP_GPIO_PORT, WKUP_GPIO_PIN) /* 读取 WKUP 引脚 */

#define KEY0_PRES 1 /* KEY0 按下 */
#define KEY1_PRES 2 /* KEY1 按下 */
#define KEY2_PRES 3 /* KEY2 按下 */
#define WKUP_PRES 4 /* KEY_UP 按下(即 WK_UP) */
```

KEY0、KEY1、KEY2 和 WK_UP 分别是读取对应按键状态的宏定义。用 HAL_GPIO_ReadPin 函数实现，该函数的返回值就是 IO 口的状态，返回值是枚举类型，取值 0 或者 1。

KEY0_PRES、KEY1_PRES、KEY2_PRES 和 WKUP_PRES 则是按键对应的四个键值宏定义标识符。

下面我们再解析 key.c 的程序，这里有两个函数，先看按键初始化函数，其定义如下：

```
/**
 * @brief      按键初始化函数
 * @param      无
 * @retval     无
 */
void key_init(void)
{
    GPIO_InitTypeDef gpio_init_struct; /* GPIO 配置参数存储变量 */
    KEY0_GPIO_CLK_ENABLE(); /* KEY0 时钟使能 */
    KEY1_GPIO_CLK_ENABLE(); /* KEY1 时钟使能 */
    KEY2_GPIO_CLK_ENABLE(); /* KEY2 时钟使能 */
    WKUP_GPIO_CLK_ENABLE(); /* WKUP 时钟使能 */
}
```

```

gpio_init_struct.Pin = KEY0_GPIO_PIN;          /* KEY0 引脚 */
gpio_init_struct.Mode = GPIO_MODE_INPUT;        /* 输入 */
gpio_init_struct.Pull = GPIO_PULLUP;            /* 上拉 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;  /* 高速 */
HAL_GPIO_Init(KEY0_GPIO_PORT, &gpio_init_struct); /* KEY0 引脚模式设置 */

gpio_init_struct.Pin = KEY1_GPIO_PIN;          /* KEY1 引脚 */
gpio_init_struct.Mode = GPIO_MODE_INPUT;        /* 输入 */
gpio_init_struct.Pull = GPIO_PULLUP;            /* 上拉 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;  /* 高速 */
HAL_GPIO_Init(KEY1_GPIO_PORT, &gpio_init_struct); /* KEY1 引脚模式设置 */

gpio_init_struct.Pin = KEY2_GPIO_PIN;          /* KEY2 引脚 */
gpio_init_struct.Mode = GPIO_MODE_INPUT;        /* 输入 */
gpio_init_struct.Pull = GPIO_PULLUP;            /* 上拉 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;  /* 高速 */
HAL_GPIO_Init(KEY2_GPIO_PORT, &gpio_init_struct); /* KEY2 引脚模式设置 */

gpio_init_struct.Pin = WKUP_GPIO_PIN;          /* WKUP 引脚 */
gpio_init_struct.Mode = GPIO_MODE_INPUT;        /* 输入 */
gpio_init_struct.Pull = GPIO_PULLDOWN;          /* 下拉 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;  /* 高速 */
HAL_GPIO_Init(WKUP_GPIO_PORT, &gpio_init_struct); /* WKUP 引脚模式设置 */
}

```

这里需要注意的是：KEY0 和 KEY1 是低电平有效的（即一端接地），所以我们要设置为内部上拉，而 KEY_UP 是高电平有效的（即一端接电源），所以我们要设置为内部下拉。

另一个函数是按键扫描函数，其定义如下：

```

/**
 * @brief      按键扫描函数
 * @note       该函数有响应优先级(同时按下多个按键)：WK_UP > KEY2 > KEY1 > KEY0!!
 * @param      mode:0 / 1, 具体含义如下：
 * @arg        0, 不支持连续按(当按键按下不放时，只有第一次调用会返回键值，
 *              必须松开以后，再次按下才会返回其他键值)
 * @arg        1, 支持连续按(当按键按下不放时，每次调用该函数都会返回键值)
 * @retval     键值，定义如下：
 *              KEY0_PRES, 1, KEY0 按下
 *              KEY1_PRES, 2, KEY1 按下
 *              KEY2_PRES, 3, KEY2 按下
 *              WKUP_PRES, 4, WKUP 按下
 */
uint8_t key_scan(uint8_t mode)
{
    static uint8_t key_up = 1; /* 按键按松开标志 */
    uint8_t keyval = 0;
    if (mode) key_up = 1;      /* 支持连续按 */

    if (key_up && (KEY0 == 0 || KEY1 == 0 || KEY2 == 0 || WK_UP == 1))
    { /* 按键松开标志为 1，且有任意一个按键按下了 */
        delay_ms(10);          /* 去抖动 */
        key_up = 0;
        if (KEY0 == 0) keyval = KEY0_PRES;
        if (KEY1 == 0) keyval = KEY1_PRES;
        if (KEY2 == 0) keyval = KEY2_PRES;
        if (WK_UP == 1) keyval = WKUP_PRES;
    }
    else if (KEY0 == 1 && KEY1 == 1 && KEY2 == 1 && WK_UP == 0)
    { /* 没有任何按键按下，标记按键松开 */
        key_up = 1;
    }
}

```

```
return keyval;          /* 返回键值 */
}
```

key_scan 函数用于扫描这 4 个 IO 口是否有按键按下。key_scan 函数，支持两种扫描方式，通过 mode 参数来设置。

当 mode 为 0 的时候，key_scan 函数将不支持连续按，扫描某个按键，该按键按下之后必须要松开，才能第二次触发，否则不会再响应这个按键，这样的好处就是可以防止按一次多次触发，而坏处就是在需要长按的时候比较不合适。

当 mode 为 1 的时候，key_scan 函数将支持连续按，如果某个按键一直按下，则会一直返回这个按键的键值，这样可以方便的实现长按检测。

有了 mode 这个参数，大家就可以根据自己的需要，选择不同的方式。这里要提醒大家，因为该函数里面有 static 变量，所以该函数不是一个可重入函数，在有 OS 的情况下，这个大家要留意下。可以看到该函数的消抖延时是 10ms。同时还有一点要注意的是，该函数的按键扫描是有优先级的，最优先的是 KEY_UP，第二优先的是 KEY0，最后是按键 KEY2。该函数有返回值，如果有按键按下，则返回非 0 值，如果没有或者按键不正确，则返回 0。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;

    HAL_Init();          /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);      /* 延时初始化 */
    led_init();          /* 初始化 LED */
    beep_init();         /* 初始化蜂鸣器 */
    key_init();          /* 初始化按键 */
    LED0(0);             /* 先点亮 LED0 */

    while(1)
    {
        key = key_scan(0); /* 得到键值 */
        if (key)
        {
            switch (key)
            {
                case WKUP_PRES: /* 控制蜂鸣器 */
                    BEEP_TOGGLE(); /* BEEP 状态取反 */
                    break;
                case KEY2_PRES: /* 控制 LED0 (RED) 翻转 */
                    LED0_TOGGLE(); /* LED0 状态取反 */
                    break;
                case KEY1_PRES: /* 控制 LED1 (GREEN) 翻转 */
                    LED1_TOGGLE(); /* LED1 状态取反 */
                    break;
                case KEY0_PRES: /* 同时控制 LED0，LED1 翻转 */
                    LED0_TOGGLE(); /* LED0 状态取反 */
                    LED1_TOGGLE(); /* LED1 状态取反 */
                    break;
            }
        }
        else
        {
            delay_ms(10);
        }
    }
}
```

首先是调用系统级别的初始化:初始化 HAL 库、系统时钟和延时函数。接下来,调用 `led_init` 来初始化 LED 灯,调用 `key_init` 函数初始化按键。最后在无限循环里面扫描获取键值,接着用键值判断哪个按键按下,如果有按键按下则翻转相应的灯,如果没有按键按下则延时 10ms。

15.4 下载验证

在下载好程序后,我们可以按 KEY0、KEY1、KEY2 和 KEY_UP 来看看 LED 灯的变化,是否和我们预期的结果一致?

至此,我们的本章的学习就结束了。本章学习了 STM32F103 的 IO 作为输入的使用方法,在前面的 GPIO 输出的基础上又学习了一种 GPIO 使用模式,大家可以回顾前面跑马灯实验介绍的 GPIO 的八种模式类型巩固 GPIO 的知识。

第十六章 外部中断实验

在前面几章的学习中，我们掌握了 STM32F1 的 IO 口最基本的操作。本章我们将介绍如何把 STM32F1 的 IO 口作为外部中断输入来使用，在本章中，我们将以中断的方式，实现我们在第十五章所实现的功能。

本章分为如下几个小节：

16.1 STM32F1 NVIC 和外部中断简介

16.2 硬件设计

16.3 程序设计

16.4 下载验证

16.1 NVIC 和 EXTI 简介

16.1.1 NVIC 简介

什么是 NVIC？NVIC 即嵌套向量中断控制器，全称 Nested vectored interrupt controller。它是内核的器件，所以它的更多描述可以看内核有关的资料《Cortex-M3 权威指南》。M3 内核都是支持 256 个中断，其中包含了 16 个系统中断和 240 个外部中断，并且具有 256 级的可编程中断设置。然而芯片厂商一般不会对内核的这些资源全部用完，如 STM32F103ZET6 的系统中断有 10 个，外部中断有 60 个。下面我们看看系统中断部分：

| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|----|-----|-------|--------------------|----------------------------------|-----------------------------|
| | - | - | - | 保留 | 0x0000_0000 |
| | -3 | 固定 | Reset | 复位 | 0x0000_0004 |
| | -2 | 固定 | NMI | 不可屏蔽中断 RCC时钟安全系统(CSS)联接到NMI向量 | 0x0000_0008 |
| | -1 | 固定 | 硬件失效(HardFault) | 所有类型的失效 | 0x0000_000C |
| | 0 | 可设置 | 存储管理(MemManage) | 存储器管理 | 0x0000_0010 |
| | 1 | 可设置 | 总线错误(BusFault) | 预取指失败，存储器访问失败 | 0x0000_0014 |
| | 2 | 可设置 | 错误应用(UsageFault) | 未定义的指令或非法状态 | 0x0000_0018 |
| | - | - | - | 保留 | 0x0000_001C ~0x0000_002B |
| | 3 | 可设置 | SVCall | 通过SWI指令的系统服务调用 | 0x0000_002C |
| | 4 | 可设置 | 调试监控(DebugMonitor) | 调试监控器 | 0x0000_0030 |
| | - | - | - | 保留 | 0x0000_0034 |
| | 5 | 可设置 | PendSV | 可挂起的系统服务 | 0x0000_0038 |
| | 6 | 可设置 | SysTick | 系统嘀嗒定时器 | 0x0000_003C |

表 16.1.1.1 中断向量表-系统中断部分

关于 60 个外部中断部分在《STM32F10xxx 参考手册_V10（中文版）.pdf》的 9.1.2 小节有详细的列表，这里就不列出来了。STM32F103 的中断向量表在 STM32F103xx.h 文件中被定义。

16.1.1.1 NVIC 寄存器

NVIC 相关的寄存器定义了可以在 core_cm3.h 文件找到。我们直接通过程序的定义来分析 NVIC 相关的寄存器，其定义如下：

```
typedef struct
{
```



```

__IOM uint32_t ISER[8U]; /* 中断使能寄存器 */
uint32_t RESERVED0[24U];

__IOM uint32_t ICER[8U]; /* 中断清除使能寄存器 */
uint32_t RESERVED1[24U];

__IOM uint32_t ISPR[8U]; /* 中断使能挂起寄存器 */
uint32_t RESERVED2[24U];

__IOM uint32_t ICPR[8U]; /* 中断解挂寄存器 */
uint32_t RESERVED3[24U];

__IOM uint32_t IABR[8U]; /* 中断有效位寄存器 */
uint32_t RESERVED4[56U];

__IOM uint8_t IP[240U]; /* 中断优先级寄存器 (8Bit 位宽) */
uint32_t RESERVED5[644U];

__IOM uint32_t STIR; /* 软件触发中断寄存器 */
} NVIC_Type;

```

STM32F103 的中断在这些寄存器的控制下有序的执行的。只有了解这些中断寄存器，才能方便的使用 STM32F103 的中断。下面重点介绍这几个寄存器：

ISER[8]: ISER 全称是：Interrupt Set Enable Registers，这是一个中断使能寄存器组。上面说了 CM3 内核支持 256 个中断，这里用 8 个 32 位寄存器来控制，每个位控制一个中断。但是 STM32F103 的可屏蔽中断最多只有 60 个，所以对我们来说，有用的就是两个(ISER[0]和 ISER[1])，总共可以表示 64 个中断。而 STM32F103 只用了其中的 60 个。ISER[0]的 bit0~31 分别对应中断 0~31；ISER[1]的 bit0~27 对应中断 32~59，这样总共 60 个中断就可以分别对应上了。你要使能某个中断，必须设置相应的 ISER 位为 1，使该中断被使能(这里仅仅是使能，还要配合中断分组、屏蔽、IO 口映射等设置才算是一个完整的中断设置)。具体每一位对应哪个中断，请参考 stm32f103xe.h 里面的第 69 行。

ICER[8]: 全称是：Interrupt Clear Enable Registers，是一个中断除能寄存器组。该寄存器组与 ISER 的作用恰好相反，是用来清除某个中断的使能的。其对应位的功能，也和 ICER 一样。这里要专门设置一个 ICER 来清除中断位，而不是向 ISER 写 0 来清除，是因为 NVIC 的这些寄存器都是写 1 有效的，写 0 是无效的。具体为什么是这样子，可以查看《Cortex-M3 权威指南》第 125 页，NVIC 章节。

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

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

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

IP [240]: 全称是：Interrupt Priority Registers，是一个中断优先级控制的寄存器组。这个寄存器组相当重要！STM32F103 的中断分组与这个寄存器组密切相关。IP 寄存器组由 240 个 8bit 的寄存器组成，每个可屏蔽中断占用 8bit，这样总共可以表示 240 个可屏蔽中断。而 STM32F103 只用到了其中的 60 个。IP[59]~IP[0]分别对应中断 59~0。而每个可屏蔽中断占用的 8bit 并没有全部使用，而是只用了高 4 位。这 4 位，又分为抢占优先级和子优先级。抢占优先级在前，子优先级在后。而这两个优先级各占几个位又要根据 SCB->AIRCRR 中的中断分组设置来决定。关于中断优先级控制的寄存器组我们下面详细讲解。

16.1.1.2 中断优先级

STM32 中的中断优先级可以分为：抢占式优先级和响应优先级，响应优先级也称子优先级，每个中断源都需要被指定这两种优先级。抢占式优先级和响应优先级的区别：

抢占优先级：抢占优先级高的中断可以打断正在执行的抢占优先级低的中断。

响应优先级：抢占优先级相同，响应优先级高的中断不能打断响应优先级低的中断。

还有一种情况就是当两个或者多个中断的抢占式优先级和响应优先级相同时，那么就遵循自然优先级，看中断向量表的中断排序，数值越小，优先级越高。

在 NVIC 中由寄存器 NVIC_IPR0~NVIC_IPR59 共 60 个寄存器控制中断优先级，每个寄存器的 8 位，所以就有了 240 个宽度为 8bit 的中断优先级控制寄存器，原则上每个外部中断可配置的优先级为 0~255，数值越小，优先级越高。但是实际上 M3 芯片为了精简设计，只使用了高四位[7:4]，低四位取零，这样以至于最多只有 16 级中断嵌套，即 $2^4=16$ 。

对于 NVCI 的中断优先级分组：STM32F103 将中断分为 5 个组，组 0~4。该分组的设置是由 SCB->AIRCRR 寄存器的 bit10~8 来定义的。具体的分配关系如表 16.1.1.2.1 所示：

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

表 16.1.1.2.1 AIRCRR 中断分组设置表

通过表 16.1.1.2.1，我们就可以清楚的看到组 0~4 对应的配置关系，例如优先级分组设置为 3，那么此时所有的 60 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

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

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

16.1.1.3 NVIC 相关函数

ST 公司把 core_cm3.h 文件的 NVIC 相关函数封装到 STM32F1xx_hal_cortex.c 文件中，下面列出我们较为常用的函数进行，想了解更多其他的函数请自行查阅。

1. HAL_NVIC_SetPriorityGrouping 函数

HAL_NVIC_SetPriorityGrouping 是设置中断优先级分组函数。其声明如下：

```
void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

- **函数描述：**
用于设置中断优先级分组。
- **函数形参：**
形参 1 是中断优先级分组号，可以选择范围：NVIC_PRIORITYGROUP_0 到 NVIC_PRIORITYGROUP_4（共 5 组）。
- **函数返回值：**
无
- **注意事项：**
这个函数在一个工程里基本只调用一次，而且是在程序 HAL 库初始化函数里面已经被调用，后续就不会再调用了。因为当后续调用设置成不同的中断优先级分组时，有可能造成前面设置好的抢占优先级和响应优先级不匹配。

2. HAL_NVIC_SetPriority 函数

HAL_NVIC_SetPriority 是设置中断优先级函数。其声明如下：

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

- **函数描述：**
用于设置中断的抢占优先级和响应优先级（子优先级）。
- **函数形参：**
形参 1 是中断号，可以选择范围：IRQn_Type 定义的枚举类型，定义在 stm32f103xe.h。
形参 2 是抢占优先级，可以选择范围：0 到 15。

形参 3 是响应优先级，可以选择范围：0 到 15。

- 函数返回值：
无

3. HAL_NVIC_EnableIRQ 函数

HAL_NVIC_EnableIRQ 是中断使能函数。其声明如下：

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

- 函数描述：
用于使能中断。
- 函数形参：
形参 **IRQn** 是中断号，可以选择范围：IRQn_Type 定义的枚举类型，定义在 stm32f103xe.h。
- 函数返回值：
无

4. HAL_NVIC_DisableIRQ 函数

HAL_NVIC_DisableIRQ 是中断失能函数。其声明如下：

```
void HAL_NVIC_disableIRQ(IRQn_Type IRQn);
```

- 函数描述：
用于中断失能。
- 函数形参：
形参 **IRQn** 是中断号，可以选择范围：IRQn_Type 定义的枚举类型，定义在 stm32f103xe.h。
- 函数返回值：
无

5. HAL_NVIC_SystemReset 函数

HAL_NVIC_SystemReset 是系统复位函数。其声明如下：

```
void HAL_NVIC_SystemReset(void);
```

- 函数描述：
用于软件复位系统。
- 函数形参：
无形参
- 函数返回值：
无

其他的 NVIC 函数用得较少，我们就不一一列出来了。NVIC 的介绍就到这，下面介绍外部中断。

16.1.2 EXTI 简介

EXTI 即是外部中断和事件控制器，它是由 20 个产生事件/中断请求的边沿检测器组成。每一条输入线都可以独立地配置输入类型（脉冲或挂起）和对应的触发事件（上升沿或下降沿或者双边沿都触发）。每个输入线都可以独立地被屏蔽。挂起寄存器保持着状态线的中断请求。

EXTI 的功能框图是最直接把有关 EXTI 的知识点连接起来的图，掌握了该图的来龙去脉，就会对 EXTI 有了一个整体熟悉，编程时候可以得心应手。EXTI 的功能框图如图 16.1.2.1。

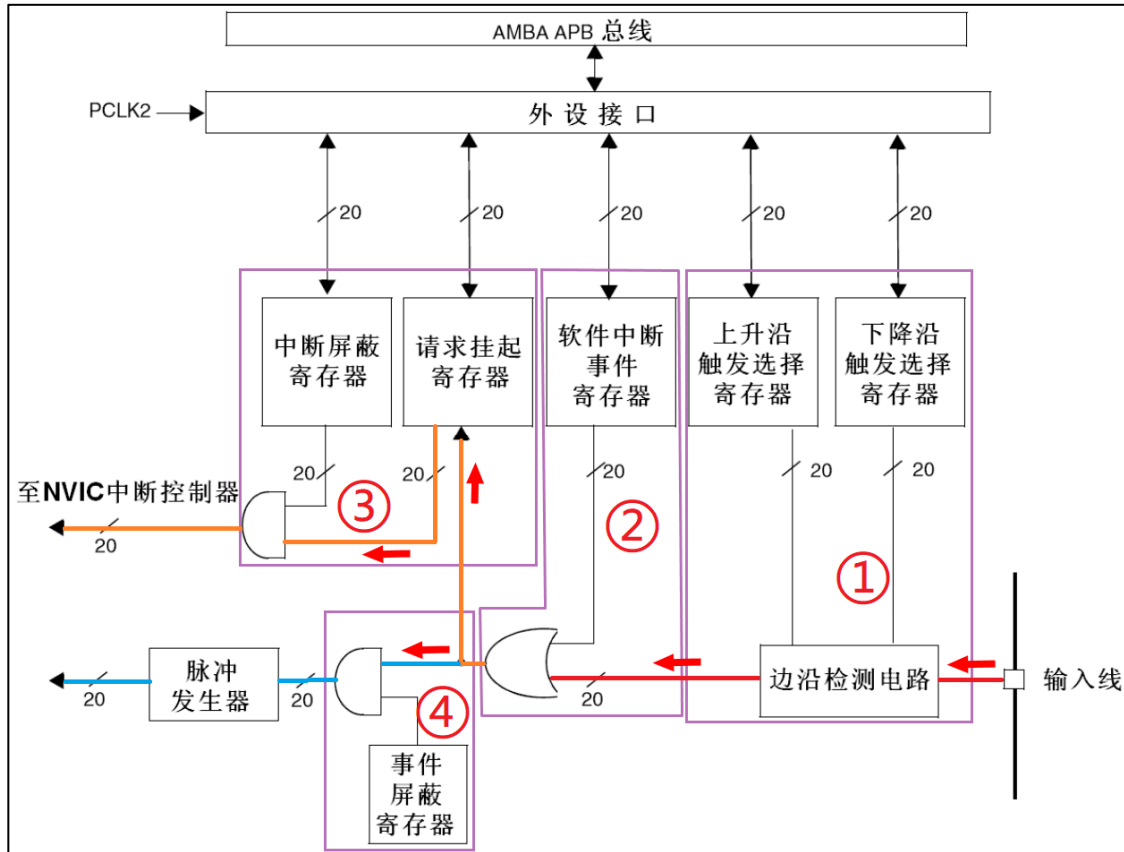


图 16.1.2.1 EXTI 功能框图

从 EXTI 功能框图可以看到有两条主线，一条是由输入线到 NVIC 中断控制器，一条是由输入线到脉冲发生器。这就恰恰是 EXTI 的两大功能，产生中断与产生事件，两者从硬件上就存在不同。

下面让我们看一下 EXTI 功能框图的产生中断的线路，最终信号是流入 NVIC 控制器中。输入线是线路的信息输入端，它可以通过配置寄存器设置为任何一个 GPIO 口，或者是一些外设的事件。输入线一般都是存在电平变化的信号。

标号①是一个边沿检测电路，包括边沿检测电路，上升沿触发选择寄存器(EXTI_RTSR)和下降沿触发选择寄存器(EXTI_FTSR)。边沿检测电路以输入线作为信号输入端，如果检测到有边沿跳变就输出有效信号‘1’，就输出有效信号‘1’到标号②部分电路，否则输出无效信号‘0’。边沿跳变的标准在于开始的时候对于上升沿触发选择寄存器或下降沿触发选择寄存器对应位的设置，对应位的设置可以参照一下表 16.1.2.1。

标号②是一个或门电路，它的两个信号输入端分别是软件中断事件寄存器(EXTI_SWIER)和边沿检测电路的输入信号。或门电路只要输入端有信号‘1’，就会输出‘1’，所以就会输出‘1’到标号③电路和标号④电路。通过对软件中断事件寄存器的读写操作就可以启动中断/事件线，即相当于输出有效信号‘1’到或门电路输入端。

标号③是一个与门电路，它的两个信号输入端分别是中断屏蔽寄存器(EXTI_IMR)和标号②电路输出信号。与门电路要求输入都为‘1’才输出‘1’，这样子的情况下，如果中断屏蔽寄存器(EXTI_IMR)设置为 0 时，不管从标号②电路输出的信号特性如何，最终标号③电路输出的信号都是 0；假如中断屏蔽寄存器(EXTI_IMR)设置为 1 时，最终标号③电路输出的信号才由标号②电路输出信号决定，这样子就可以简单控制 EXTI_IMR 来实现中断的目的。标号④电路输出‘1’就会把请求挂起寄存器(EXTI_PR)对应位置 1。

最后，请求挂起寄存器(EXTI_PR)的内容就输出到 NVIC 内，实现系统中断事件的控制。

接下来我们看看 EXTI 功能框图的产生事件的线路。

产生事件线路是从标号 2 之后与中断线路有所不用，之前的线路都是共用的。标号④是一个与门，输入端来自标号 2 电路以及来自于事件屏蔽寄存器(EXTI_EMR)。如果 EXTI_EMR 寄

寄存器设置为 0，那不管标号 2 电路输出的信号是 ‘0’ 还是 ‘1’，最终标号 4 输出的是 ‘0’；如果 EXTI_EMR 寄存器设置为 1，最终标号④电路输出信号就由标号③电路输出的信号决定，这样子就可以简单的控制 EXTI_EMR 来实现是否产生事件的目的。

标号④电路输出有效信号 1 就会使脉冲发生器电路产生一个脉冲，而无效信号就不会使其产生脉冲信号。脉冲信号产生可以给其他外设电路使用，例如定时器，模拟数字转换器等，这样的脉冲信号一般用来触发 TIM 或者 ADC 开始转换。

产生中断线路目的使把输入信号输入到 NVIC，进一步运行中断服务函数，实现功能。而产生事件线路目的是传输一个脉冲信号给其他外设使用，属于硬件级功能。

EXTI 支持 19 个外部中断/事件请求，这些都是信息输入端，也就是上面提及到了输入线，具体如下：

EXTI 线 0~15：对应外部 IO 口的输入中断

EXTI 线 16：连接到 PVD 输出

EXTI 线 17：连接到 RTC 闹钟事件

EXTI 线 18：连接到 USB 唤醒事件

EXTI 线 19：连接到以太网唤醒事件

从上面可以看出，STM32F1 供给 IO 口使用的中断线只有 16 个，但是 STM32F1 的 IO 口却远远不止 16 个，所以 STM32 把 GPIO 管脚 GPIOx.0~GPIOx.15(x=A,B,C,D,E,F,G)分别对应中断线 0~15。这样子每个中断线对应了最多 9 个 IO 口，以线 0 为例：它对应了 GPIOA.0、GPIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0 和 GPIOG.0。而中断线每次只能连接到 1 个 IO 口上，这样就需要通过配置决定对应的中断线配置到哪个 GPIO 上了。

GPIO 和中断线映射关系是在寄存器 AFIO_EXTICR1 ~ AFIO_EXTICR4 中配置的。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|----|----|----|---|----|----|----|---------------|----|----|----|------------|----|----|----|
| EXTI3[3:0] | | | | EXTI2[3:0] | | | | EXTI1[3:0] | | | | EXTI0[3:0] | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位15:0 | | | | EXTIx[3:0]: EXTIx配置(x = 0 ... 3) (EXTI x configuration) 这些位可由软件读写，用于选择EXTIx外部中断的输入源。参看9.2.5节。 | | | | | | | | | | | |
| | | | | 0000: PA[x]引脚 | | | | 0100: PE[x]引脚 | | | | | | | |
| | | | | 0001: PB[x]引脚 | | | | 0101: PF[x]引脚 | | | | | | | |
| | | | | 0010: PC[x]引脚 | | | | 0110: PG[x]引脚 | | | | | | | |
| | | | | 0011: PD[x]引脚 | | | | | | | | | | | |

图 16.1.2.2 AFIO_EXTICR1 寄存器

AFIO_EXTICR1 寄存器配置 EXTI0 到 EXTI3 线，包含的外部中断的引脚包括 PAx 到 PGx，x=0 到 3。AFIO_EXTICR2 寄存器配置 EXTI4 到 EXTI7 线，包含的外部中断的引脚包括 PAx 到 PGx，x=4 到 7，AFIO_EXTICR2 寄存器请打开参考手册查看（这里没有截图出来）。AFIO_EXTICR3 和 AFIO_EXTICR4 以此类推。

另外要注意的是，我们配置 AFIO 相关寄存器前，还需要打开 AFIO 时钟。

16.2 硬件设计

1. 例程功能

通过外部中断的方式让开发板上的三个独立按键控制 LED 灯：KEY_UP 控制 LED0 翻转，KEY1 控制 LED1 翻转，KEY0 控制 LED2 翻转。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 独立按键

KEY0 - PE4
KEY1 - PE3
KEY2 - PE2
KEY_UP - PA0

3. 原理图

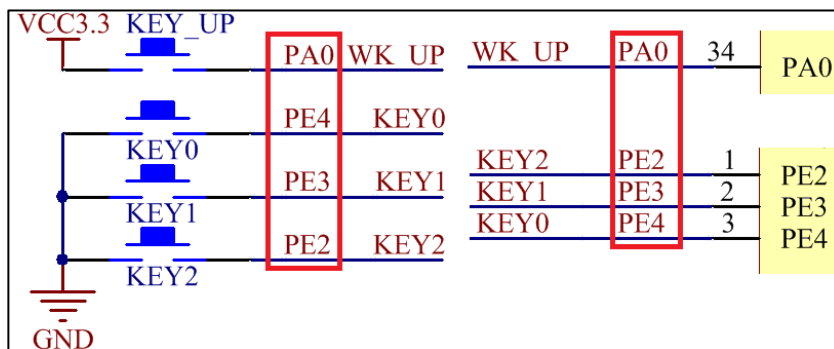


图 16.2.1 独立按键与 STM32F1 连接原理图

独立按键硬件部分的连接原理的如图 16.2.1。这里需要注意的是：KEY0、KEY1 和 KEY2 设计为采样到按键另一端的低电平为有效，而 KEY_UP 则需要采样到高电平才为按键有效，并且外部都没有上下拉电阻，所以需要在 STM32F103 内部设置上下拉以设置空闲电平。

16.3 程序设计

16.3.1 EXTI 的 HAL 库驱动

前面讲解 HAL_GPIO_Init 函数的时候有提到过：HAL 库的 EXTI 外部中断的设置功能整合到 HAL_GPIO_Init 函数里面，而不是单独独立一个文件。所以我们的外部中断的初始化函数也是用 HAL_GPIO_Init 函数。

既然是要用到外部中断，所以我们的 GPIO 的模式要从下面的三个模式中选中一个：

```
#define GPIO_MODE_IT_RISING          (0x11110000U) /* 外部中断，上升沿触发检测 */
#define GPIO_MODE_IT_FALLING        (0x11210000U) /* 外部中断，下降沿触发检测 */
/* 外部中断，上升和下降双沿触发检测 */
#define GPIO_MODE_IT_RISING_FALLING (0x11310000U)
```

KEY0、KEY1 和 KEY2 是低电平有效的，程序设计为按键按下触发中断，所以我们要选择下降沿触发检测，而 KEY_UP 是高电平有效的，那么就应该选择上升沿触发检测。

EXTI 外部中断配置步骤

1) 使能对应 GPIO 口时钟

本实验用到的 GPIO 和按键输入实验是一样的，因此 GPIO 时钟使能也是一样的，请参考上一章代码。

2) 设置 GPIO 工作模式，触发条件，开启 AFIO 时钟，设置 IO 口与中断线的映射关系

这些步骤 HAL 库全部封装在 HAL_GPIO_Init 函数里面，我们只需要设置好对应的参数，再调用 HAL_GPIO_Init 函数即可完成配置。

3) 配置中断优先级（NVIC），并使能中断。

配置好 GPIO 模式以后，我们需要设置中断优先级和使能中断，中断优先级我们使用 HAL_NVIC_SetPriority 函数设置，中断使能我们使用 HAL_NVIC_EnableIRQ 函数设置。

4) 编写中断服务函数。

每开启一个中断，就必须编写其对应的中断服务函数，否则将导致死机（CPU 将找不到中断服务函数）。中断服务函数接口厂家已经在 startup_stm32f103xe.s 中做好了，STM32F1 的 IO 口外部中断函数只有 7 个，分别为：

```
void EXTI0_IRQHandler();
```



```
void EXTI1_IRQHandler();
void EXTI2_IRQHandler();
void EXTI3_IRQHandler();
void EXTI4_IRQHandler();
void EXTI9_5_IRQHandler();
void EXTI15_10_IRQHandler();
```

中断线 0-4, 每个中断线对应一个中断函数, 中断线 5-9 共用中断函数 EXTI9_5_IRQHandler, 中断线 10-15 共用中断函数 EXTI15_10_IRQHandler。一般情况下, 我们可以把中断控制逻辑直接编写在中断服务函数中, 但是 HAL 库把中断处理过程进行了简单封装, 请看步骤 5 讲解。

5) 编写中断处理回调函数 HAL_GPIO_EXTI_Callback

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

我们先看一下 HAL_GPIO_EXTI_IRQHandler 函数定义:

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00U)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin); /* 清中断标志位 */
        HAL_GPIO_EXTI_Callback(GPIO_Pin); /* 外部中断回调函数 */
    }
}
```

该函数实现的作用非常简单, 通过入口参数 GPIO_Pin 判断中断来自哪个 IO 口, 然后清除相应的中断标志位, 最后调用回调函数 HAL_GPIO_EXTI_Callback() 实现控制逻辑。在所有的中断服务函数中直接调用外部中断共用处理函数 HAL_GPIO_EXTI_IRQHandler, 然后在回调函数 HAL_GPIO_EXTI_Callback 中通过判断中断是来自哪个 IO 口编写相应的中断服务控制逻辑。

因此我们可以在 HAL_GPIO_EXTI_Callback 里面实现控制逻辑编写, 详见本实验源码。

16.3.2 程序流程图

下面看看本实验的程序流程图:

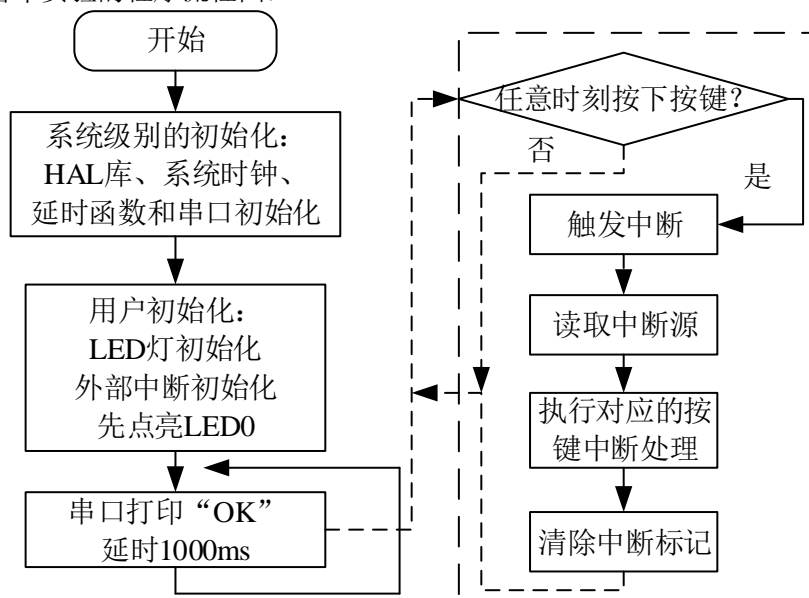


图 16.3.2.1 外部中断实验程序流程图

主程序初始外设, 在按键初始化时初始化按键的采样边缘。

16.3.3 程序解析

1. 外部中断驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。外部中断(EXTI)驱动源码包括两个文件：exti.c 和 exti.h。

下面我们先解析 exti.h 的程序。

● 外部中断引脚定义

由硬件设计小节，我们知道 KEY0、KEY1、KEY2 和 KEY_UP 分别来连接到 PE4、PE3、PE2 和 PA0 上，我们做了下面的引脚定义。

```
/* 引脚 和 中断编号 & 中断服务函数 定义 */
#define KEY0_INT_GPIO_PORT      GPIOE
#define KEY0_INT_GPIO_PIN      GPIO_PIN_4
/* PE 口时钟使能 */
#define KEY0_INT_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY0_INT_IRQn          EXTI4_IRQn
#define KEY0_INT_IRQHandler     EXTI4_IRQHandler

#define KEY1_INT_GPIO_PORT      GPIOE
#define KEY1_INT_GPIO_PIN      GPIO_PIN_3
/* PE 口时钟使能 */
#define KEY1_INT_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY1_INT_IRQn          EXTI3_IRQn
#define KEY1_INT_IRQHandler     EXTI3_IRQHandler

#define KEY2_INT_GPIO_PORT      GPIOE
#define KEY2_INT_GPIO_PIN      GPIO_PIN_2
/* PE 口时钟使能 */
#define KEY2_INT_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY2_INT_IRQn          EXTI2_IRQn
#define KEY2_INT_IRQHandler     EXTI2_IRQHandler

#define WKUP_INT_GPIO_PORT      GPIOA
#define WKUP_INT_GPIO_PIN      GPIO_PIN_0
/* PA 口时钟使能 */
#define WKUP_INT_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)
#define WKUP_INT_IRQn          EXTI0_IRQn
#define WKUP_INT_IRQHandler     EXTI0_IRQHandler
```

KEY0、KEY1、KEY2 和 WK_UP 分别连接 PE4、PE3、PE2 和 PA0，即对应了 EXTI4、EXTI3、EXTI2 和 EXTI0 这三条外部中断线。这里需要注意的是 EXTI0 到 EXTI4 都是有单独的中断向量，EXTI5 到 EXTI9 是公用 EXTI9_5_IRQn，EXTI10 到 EXTI15 是公用 EXTI15_10_IRQn。

下面我们再解析 exti.c 的程序，先看外部中断初始化函数，其定义如下：

```
/**
 * @brief      外部中断初始化程序
 * @param      无
 * @retval     无
 */
void extix_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    key_init();
    gpio_init_struct.Pin = KEY0_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
    gpio_init_struct.Pull = GPIO_PULLUP;
    /* KEY0 配置为下降沿触发中断 */
    HAL_GPIO_Init(KEY0_INT_GPIO_PORT, &gpio_init_struct);
    gpio_init_struct.Pin = KEY1_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
```

```

gpio_init_struct.Pull = GPIO_PULLUP;
/* KEY1 配置为下降沿触发中断 */
HAL_GPIO_Init(KEY1_INT_GPIO_PORT, &gpio_init_struct);
gpio_init_struct.Pin = KEY2_INT_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
gpio_init_struct.Pull = GPIO_PULLUP;
/* KEY2 配置为下降沿触发中断 */
HAL_GPIO_Init(KEY2_INT_GPIO_PORT, &gpio_init_struct);
gpio_init_struct.Pin = WKUP_INT_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_IT_RISING;          /* 上升沿触发 */
gpio_init_struct.Pull = GPIO_PULLDOWN;
/* WKUP 配置为下降沿触发中断 */
HAL_GPIO_Init(WKUP_GPIO_PORT, &gpio_init_struct);
HAL_NVIC_SetPriority(KEY0_INT_IRQn, 0, 2);             /* 抢占 0, 子优先级 2 */
HAL_NVIC_EnableIRQ(KEY0_INT_IRQn);                   /* 使能中断线 1 */

HAL_NVIC_SetPriority(KEY1_INT_IRQn, 1, 2);             /* 抢占 1, 子优先级 2 */
HAL_NVIC_EnableIRQ(KEY1_INT_IRQn);                   /* 使能中断线 15 */

HAL_NVIC_SetPriority(KEY2_INT_IRQn, 2, 2);             /* 抢占 2, 子优先级 2 */
HAL_NVIC_EnableIRQ(KEY2_INT_IRQn);                   /* 使能中断线 15 */

HAL_NVIC_SetPriority(WKUP_INT_IRQn, 3, 2);            /* 抢占 3, 子优先级 2 */
HAL_NVIC_EnableIRQ(WKUP_INT_IRQn);                   /* 使能中断线 0 */
}

```

外部中断初始化函数主要做了两件事情，先是调用 IO 口初始化函数 HAL_GPIO_Init 来初始化 IO 口，然后设置中断优先级并使能中断线。

4 个外部中断服务函数，用于产生中断事件时进行处理，其定义如下：

```

/**
 * @brief      KEY0 外部中断服务程序
 * @param      无
 * @retval     无
 */
void KEY0_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY0 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(KEY0_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(KEY0_INT_GPIO_PIN);
}

/**
 * @brief      KEY1 外部中断服务程序
 * @param      无
 * @retval     无
 */
void KEY1_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY1 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(KEY1_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(KEY1_INT_GPIO_PIN);
}

/**
 * @brief      KEY2 外部中断服务程序
 * @param      无
 * @retval     无
 */
void KEY2_INT_IRQHandler(void)
{

```

```

/* 调用中断处理公用函数 清除 KEY2 所在中断线 的中断标志位 */
HAL_GPIO_EXTI_IRQHandler(KEY2_INT_GPIO_PIN);
/* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
__HAL_GPIO_EXTI_CLEAR_IT(KEY2_INT_GPIO_PIN);
}

/**
 * @brief      WK_UP 外部中断服务程序
 * @param      无
 * @retval     无
 */
void WKUP_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY_UP 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(WKUP_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(WKUP_INT_GPIO_PIN);
}

```

所有的外部中断服务函数里都只调用了同样一个函数 HAL_GPIO_EXTI_IRQHandler，该函数是外部中断共用入口函数，函数内部会进行中断标志位清零，并且调用中断处理共用回调函数 HAL_GPIO_EXTI_Callback。但是它们的形参不同，我们的回调函数也是根据形参去判断是哪个 IO 口的外部中断线被触发。

外部中断回调函数，其定义如下：

```

/**
 * @brief      中断服务程序中需要做的事情
 *              在 HAL 库中所有的外部中断服务函数都会调用此函数
 * @param      GPIO_Pin: 中断引脚号
 * @retval     无
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    delay_ms(20); /* 消抖 */
    switch(GPIO_Pin)
    {
        case KEY0_INT_GPIO_PIN:
            if (KEY0 == 0)
            {
                LED0_TOGGLE(); /* LED0 状态取反 */
                LED1_TOGGLE(); /* LED1 状态取反 */
            }
            break;
        case KEY1_INT_GPIO_PIN:
            if (KEY1 == 0)
            {
                LED0_TOGGLE(); /* LED0 状态取反 */
            }
            break;
        case KEY2_INT_GPIO_PIN:
            if (KEY2 == 0)
            {
                LED1_TOGGLE(); /* LED1 状态取反 */
            }
            break;
        case WKUP_INT_GPIO_PIN:
            if (WK_UP == 1)
            {
                BEEP_TOGGLE(); /* 蜂鸣器状态取反 */
            }
            break;
    }
}

```

我们在前面中断函数的处理过程中都调用了 HAL_GPIO_EXTI_IRQHandler() 这个接口，它

主要帮我们进行了寄存器操作，清除了中断事件，清除完中断源后，调用中断回调函数 HAL_GPIO_EXTI_Callback，这个接口是一个 __weak 的接口，我们通过重新实现这个函数来实现真正的外部中断控制逻辑。在该函数内部，通过判断 IO 引脚号来确定中断是来自哪个 IO 口，也就是哪个中断线，然后编写相应的控制逻辑。所以在该函数内部，我们通过 switch 语句判断 IO 口来源，例如是来自 GPIO_PIN_0，那么一定是来自 PA0，因为中断线一次只能连接一个 IO 口，而三个 IO 口中引脚号为 0 的 IO 口只有 PA0，所以中断线 0 一定是连接 PA0，也就是外部中断由 PA0 触发。其他的引脚号的逻辑类似。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);            /* 延时初始化 */
    usart_init(115200);        /* 串口初始化为 115200 */
    led_init();                /* 初始化 LED */
    beep_init();               /* 初始化蜂鸣器 */
    extix_init();              /* 初始化外部中断输入 */
    LED0(0);                  /* 先点亮 LED0 */

    while (1)
    {
        printf("OK\r\n");
        delay_ms(1000);
    }
}
```

首先是调用系统级别的初始化：初始化 HAL 库、系统时钟和延时函数。接下来，调用串口初始化函数，调用 led_init 来初始化 LED 灯，调用 extix_init 函数初始化外部中断，点亮红灯。最后在无限循环里面执行打印“OK”后 1000ms 的重复动作。逻辑控制代码都在中断回调函数中完成。

16.4 下载验证

在下载好程序后，我们可以按 KEY0、KEY1、KEY2 和 KEY_UP 来看看 LED 灯以及蜂鸣器的变化，是否和我们预期的结果一致？

至此，我们的本章的学习就结束了。本章学习了 STM32F103 外部中断的使用方法。

第十七章 串口通信实验

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

- 17.1 串口简介
- 17.2 硬件设计
- 17.3 程序设计
- 17.4 下载验证

17.1 串口简介

学习串口前，我们先来了解一下数据通信的一些基础概念。

17.1.1 数据通信的基础概念

在单片机的应用中，数据通信是必不可少的一部分，比如：单片机和上位机、单片机和外围器件之间，它们都有数据通信的需求。由于设备之间的电气特性、传输速率、可靠性要求各不相同，于是就有了各种通信类型、通信协议，我们最常有：USART、IIC、SPI、CAN、USB 等。下面，我们先来学习数据通信的一些基础概念。

1. 数据通信方式

按数据通信方式分类，可分为串行通信和并行通信两种。串行和并行的对比如下图所示：

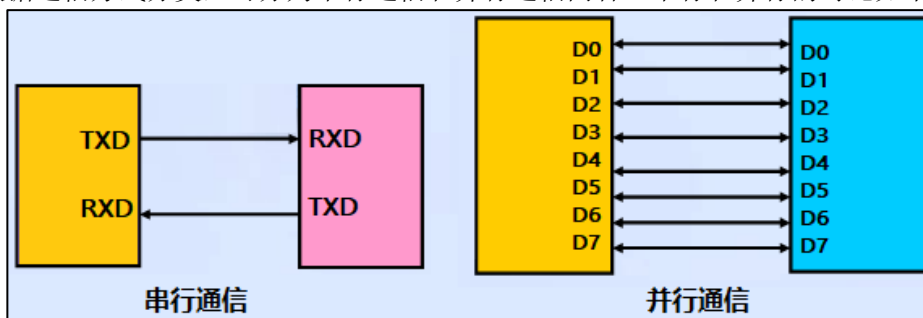


图 17.1.1.1 数据传输方式

串行通信的基本特征是数据逐位顺序依次传输，优点是传输线少、布线成本低、灵活度高等优点，一般用于近距离人机交互，特殊处理后也可以用于远距离，缺点就是传输速率低。

而并行通信是数据各位可以通过多条线同时传输，优点是传输速率高，缺点就是布线成本高，抗干扰能力差因而适用于短距离、高速率的通信。

2. 数据传输方向

根据数据传输方向，通信又可分为全双工、半双工和单工通信。全双工、半双工和单工通信的比较如下图所示：

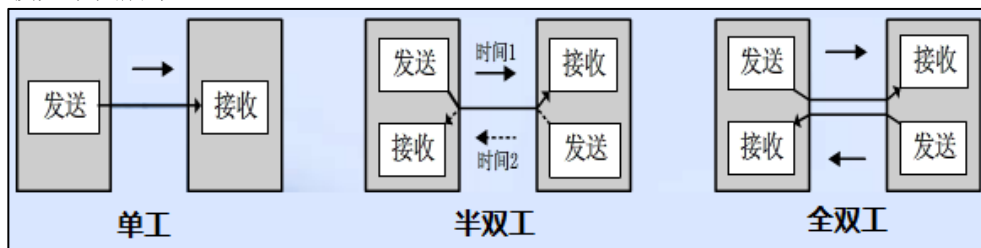


图 17.1.1.2 数据传输方式

单工是指数据传输仅能沿一个方向，不能实现反方向传输，如校园广播。

半双工是指数据传输可以沿着两个方向，但是需要分时进行，如对讲机。

全双工是指数据可以同时进行双向传输，日常的打电话属于这种情形。

这里注意全双工和半双工通信的区别：半双工通信是共用一条线路实现双向通信，而全双工是利用两条线路，一条用于发送数据，另一条用于接收数据。

3. 数据同步方式

根据数据同步方式，通信又可分为同步通信和异步通信。同步通信和异步通信比较如下图所示：

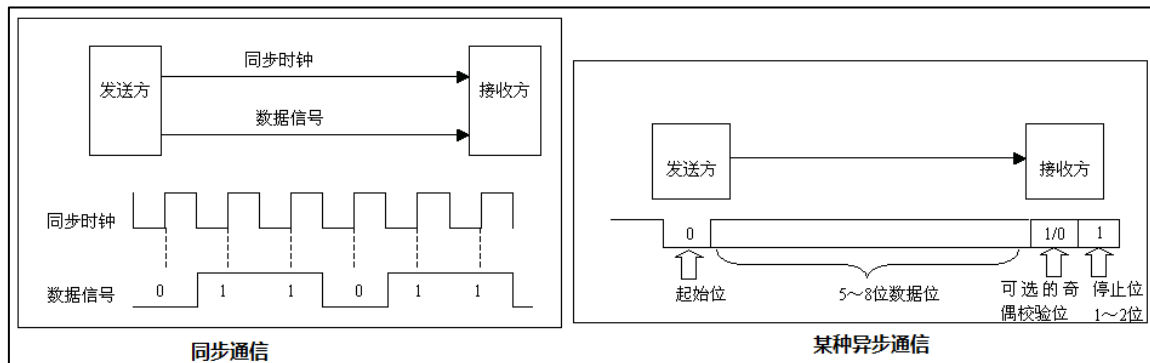


图 17.1.1.3 数据同步方式

同步通信要求通信双方共用同一时钟信号，在总线上保持统一的时序和周期完成信息传输。优点：可以实现高速率、大容量的数据传输，以及点对多点传输。缺点：要求发送时钟和接收时钟保持严格同步，收发双方时钟允许的误差较小，同时硬件复杂。

异步通信不需要时钟信号，而是在数据信号中加入开始位和停止位等一些同步信号，以便使接收端能够正确地将每一个字符接收下来，某些通信中还需要双方约定传输速率。优点：没有时钟信号硬件简单，双方时钟可允许一定误差。缺点：通信速率较低，只适用点对点传输。

4. 通信速率

在数字通信系统中，通信速率（传输速率）指数据在信道中传输的速度，它分为两种：传信率和传码率。

传信率：每秒钟传输的信息量，即每秒钟传输的二进制位数，单位为 bit/s（即比特每秒），因而又称为**比特率**。

传码率：每秒钟传输的码元个数，单位为 Baud（即波特每秒），因而又称为**波特率**。

比特率和**波特率**这两个概念又常常被人们混淆。比特率很好理解，我们来看看波特率，波特率被传输的是码元，码元是信号被调制后的概念，每个码元都可以表示一定 bit 的数据信息量。举个例子，在 TTL 电平标准的通信中，用 0V 表示逻辑 0，5V 表示逻辑 1，这时候这个码元就可以表示两种状态。如果电平信号 0V、2V、4V 和 6V 分别表示二进制数 00、01、10、11，这时候每一个码元就可以表示四种状态。

由上述可以看出，码元携带一定的比特信息，所以比特率和波特率也是有一定的关系的。

比特率和波特率的关系可以用以下式子表示：

$$\text{比特率} = \text{波特率} * \log_2 M$$

其中 M 表示码元承载的信息量。我们也可以理解 M 为码元的进制数。

举个例子：波特率为 100 Baud，即每秒传输 100 个码元，如果码元采用十六进制编码（即 M=2，代入上述式子），那么这时候的比特率就是 400 bit/s。如果码元采用二进制编码（即 M=2，代入上述式子），那么这时候的比特率就是 100 bit/s。

可以看出采用二进制的时候，波特率和比特率数值上相等。但是这里要注意，它们的相等只是数值相等，其意义上不同，看波特率和波特率单位就知道。由于我们的所用的数字系统都是二进制的，所以有部分人久而久之就直接把波特率和比特率混淆了。

17.1.2 串口通信协议简介

串口通信是一种设备间常用的串行通信方式，串口按位（bit）发送和接收字节。尽管比特字节（byte）的串行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。**串口通信协议**是指规定了数据包的内容，内容包含了起始位、主体数据、校验位及停止位，双方需要约定一致的数据包格式才能正常收发数据的有关规范。在串口通信中，常用的协议包括 RS-232、RS-422 和 RS-485 等。

随着科技的发展，RS-232 在工业上还有广泛的使用，但是在商业技术上，已经慢慢的使用 USB 转串口取代了 RS-232 串口。我们只需要在电路中添加一个 USB 转串口芯片，就可以实现 USB 通信协议和标准 UART 串行通信协议的转换，而我们开发板上的 USB 转串口芯片是 CH340C 这个芯片。关于 USB 转串口芯片的原理图请看 17.2 小节。

下面我们来学习串口通信协议，这里主要学习串口通信的协议层。

串口通信的数据包由发送设备的 TXD 接口传输到接收设备的 RXD 接口。在串口通信的协议层中，规定了数据包的内容，它由起始位、主体数据、校验位以及停止位组成，通讯双方的数据包格式要约定一致才能正常收发数据，其组成如图 17.1.2.1 所示。

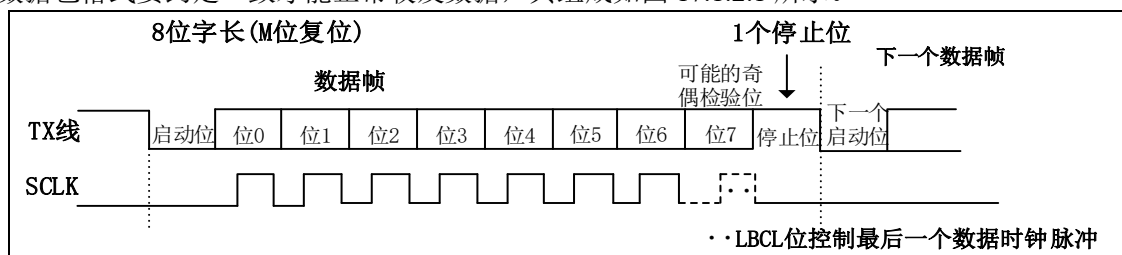


图 17.1.2.1 串口通信协议数据帧格式

串口通信协议数据包组成可以分为波特率和数据帧格式两部分。

1. 波特率

本章主要讲解的是串口异步通信，异步通信是不需要时钟信号的，但是这里需要我们约定好两个设备的波特率。波特率表示每秒钟传送的码元符号的个数，所以它决定了数据帧里面每一个位的时间长度。两个要通信的设备的波特率一定要设置相同，我们常见的波特率是 4800、9600、115200 等。

2. 数据帧格式

数据帧格式需要我们提前约定好，串口通信的数据帧包括起始位、停止位、有效数据位以及校验位。

● 起始位和停止位

串口通信的一个数据帧是从起始位开始，直到停止位。数据帧中的起始位是由一个逻辑 0 的数据位表示，而数据帧的停止位可以是 0.5、1、1.5 或 2 个逻辑 1 的数据位表示，只要双方约定一致即可。

● 有效数据位

数据帧的起始位之后，就接着是数据位，也称有效数据位，这就是我们真正需要的数据，有效数据位通常会被约定为 5、6、7 或者 8 个位长。有效数据位是低位（LSB）在前，高位（MSB）在后。

● 校验位

校验位可以认为是一个特殊的数据位。校验位一般用来判断接收的数据位有无错误，检验方法有：奇检验、偶检验、0 检验、1 检验以及无检验。下面分别介绍一下：

奇校验是指有效数据为和校验位中“1”的个数为奇数，比如一个 8 位长的有效数据为：10101001，总共有 4 个“1”，为达到奇校验效果，校验位设置为“1”，最后传输的数据是 8 位的有效数据加上 1 位的校验位总共 9 位。

偶校验与奇校验要求刚好相反，要求帧数据和校验位中“1”的个数为偶数，比如数据帧：

11001010，此时数据帧“1”的个数为4个，所以偶校验位为“0”。

0 校验是指不管有效数据中的内容是什么，校验位总为“0”，**1 校验**是校验位总为“1”。

无校验是指数据帧中不包含校验位。

我们一般是使用无校验的情况。

17.1.3 STM32F1 的串口简介

STM32F103 的串口资源相当丰富，功能也相当强劲。STM32F103ZET6 最多可提供 5 路串口，有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范、具有 DMA 等。

STM32F1 的串口分为两种：USART（即通用同步异步收发器）和 UART（即通用异步收发器）。UART 是在 USART 基础上裁剪掉了同步通信功能，只剩下异步通信功能。简单区分同步和异步就是看通信时需不需要对外提供时钟输出，我们平时用串口通信基本都是异步通信。

STM32F1 有 3 个 USART 和 2 个 UART，其中 USART1 的时钟源来自于 APB2 时钟，其最大频率为 72MHz，其他 4 个串口的时钟源可以来自于 APB1 时钟，其最大频率为 36MHz。

STM32 的串口输出的是 TTL 电平信号，如果需要 RS-232 标准的信号可使用 MAX3232 芯片进行转换，而本实验我们是通过 USB 转串口芯片 CH340C 来与电脑的上位机进行通信。

17.1.3.1 USART 框图

下面先来学习如图 17.1.3.1.1 所示的 USART 框图，通过 USART 框图引出 USART 的相关知识，从而有了一个很好的整体掌握，对之后的编程也会有一个清晰的思路。

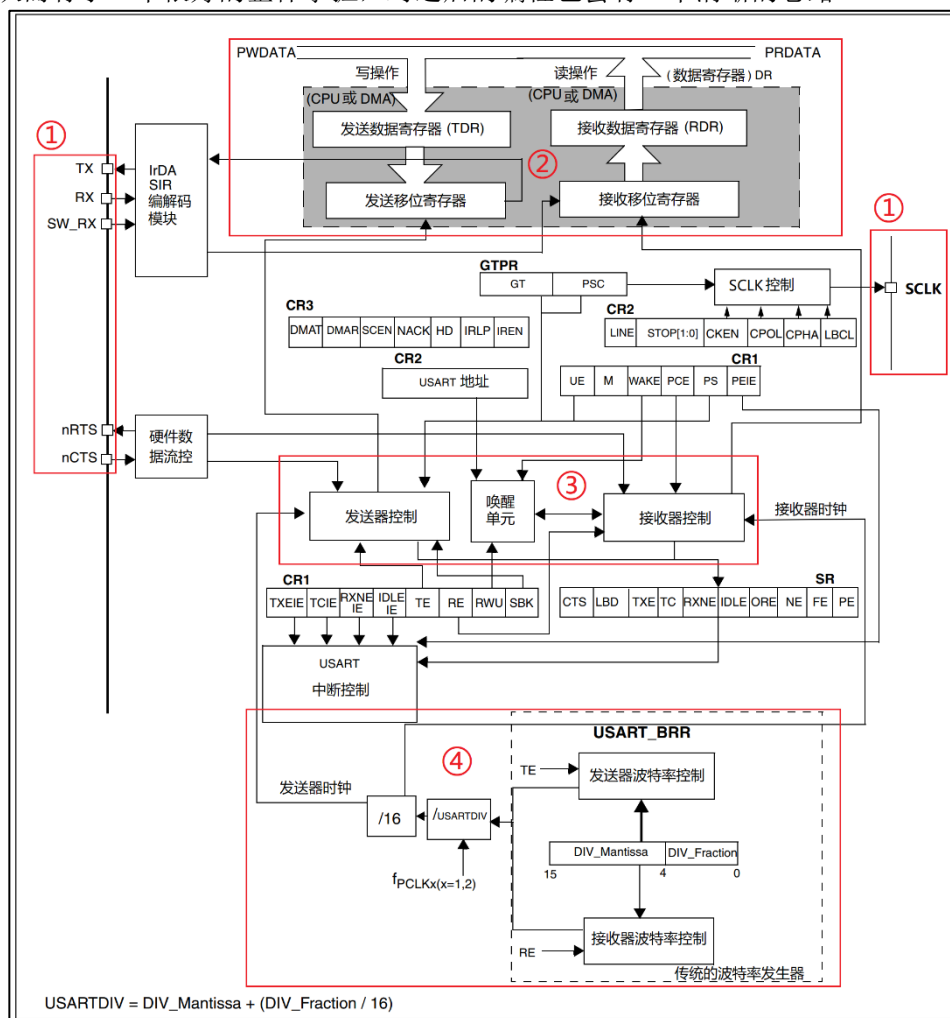


图 17.1.3.1.1 USART 框图

为了方便大家理解，我们把整个框图分成几个部分来介绍。

① USART 信号引脚

TX: 发送数据输出引脚

RX: 接收数据输入引脚

SCLK: 发送器时钟输出, 适用于同步传输

SW_RX: 数据接收引脚, 属于内部引脚, 用于智能卡模式

IrDA_RDI: IrDA 模式下的数据输入

IrDA_TDO: IrDA 模式下的数据输出

nRTS: 发送请求, 若是低电平, 表示 USART 准备好接收数据

nCTS: 清除发送, 若是高电平, 在当前数据传输结束时阻断下一次的数据发送

② 数据寄存器

USART_DR 包含了已发送或接收到的数据。由于它本身就是两个寄存器组成的, 一个专门给发送用的 (TDR), 一个专门给接收用的 (RDR), 该寄存器具备读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。当进行数据发送操作时, 往 USART_DR 中写入数据会自动存储在 TDR 内; 当进行读取操作时, 向 USART_DR 读取数据会自动提去 RDR 数据。

USART 数据寄存器 (USART_DR) 低 9 位数据有效, 其他数据位保留。USART_DR 的第 9 位数据是否有效跟 USART_CR1 的 M 位设置有关, 当 M 位为 0 表示 8 位数据字长; 当 M 位为 1 时表示 9 位数据字长, 一般使用 8 位数据字长。

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

③ 控制器

USART 有专门控制发送的发送器, 控制接收的接收器, 还有唤醒单元、中断控制等等, 具体在后面讲解 USART 寄存器的时候细讲。

④ 时钟与波特率

这部分的主要功能就是为 USART 提供时钟以及配置波特率。

波特率, 即每秒钟传输的码元个数, 在二进制系统中 (串口的数据帧就是二进制的形式), 波特率与波特率的数值相等, 所以我们今后在把串口波特率理解为每秒钟传输的二进制位数。

波特率通过以下公式得出:

$$baud = \frac{fck}{16 * USARTDIV}$$

fck 是给串口的时钟 (USART2\3\4\5 的时钟源为 PCLK1, USART1 的时钟源为 PCLK2), USARTDIV 是一个无符号的定点数, 存放在波特率寄存器 (USART_BRR) 的低 16 位, DIV_Mantissa[11:0] 存放的是 USARTDIV 的整数部分, DIV_Fractionp[3:0] 存放的是 USARTDIV 的小数部分。

下面举个例子说明:

当串口 1 设置需要得到 115200 的波特率, fck = 72MHz, 那么可得:

$$115200 = \frac{72000000}{16 * USARTDIV}$$

得到 USARTDIV = 39.0625, 分离 USARTDIV 的整数部分与小数部分, 整数部分为 39, 即 0x27, 那么 DIV_Mantissa = 0x27; 小数部分为 0.0625, 转化为十六进制即 0.0625*16=1, 所以 DIV_Fractionp = 0x1, USART_BRR 寄存器应该赋值为 0x271, 成功设置波特率为 115200。

值得注意 **USARTDIV 是允许有余数的**, 我们用四舍五入进行取整, 这样会导致波特率会有所偏差, 而这样的小误差是可以被允许的。

17.1.3.2 USART 寄存器

使用 STM32F103 的 USART 的配置步骤在《STM32F10xxx 参考手册_V10 (中文版).pdf》中有列出, 这里我们引用手册中的配置步骤:

1. 通过在 USART_CR1 寄存器上置位 UE 位来激活 USART。
2. 编程 USART_CR1 的 M 位来定义字长。

3. 在 USART_CR2 中编程停止位的位数。
4. 如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位(DMAT)。按多缓冲器通信中的描述配置 DMA 寄存器。
5. 利用 USART_BRR 寄存器选择要求的波特率。
6. 设置 USART_CR1 中的 TE 位，发送一个空闲帧作为第一次数据发送。
7. 把要发送的数据写进 USART_DR 寄存器(此动作清除 TXE 位)。在只有一个缓冲器的情况下，对每个待发送的数据重复步骤 7。
8. 在 USART_DR 寄存器中写入最后一个数据字后，要等待 TC=1，它表示最后一个数据帧的传输结束。当需要关闭 USART 或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传输。

我们按照上面的步骤配置就可以使用 STM32F103 的串口了：只要你开启了串口时钟，并设置相应 IO 口的模式，然后配置一下波特率，数据位长度，奇偶校验位等信息，就可以使用了。总结地来说，我们要学会配置 USART 对应的寄存器就可以使用串口功能了。下面，我们就简单介绍下这几个与串口基本配置直接相关的寄存器。

(1) 串口时钟使能

串口作为 STM32F103 的一个外设，其时钟由外设时钟使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。

注意：除了串口 1 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR 寄存器，而 APB2（72M）的频率一般是 APB1（36M）的一倍。

(2) 串口复位

一般系统刚开始配置外设的时候，都会先执行复位该外设的操作，可以使外设的对应寄存器恢复到默认值，方便我们进行配置。串口 1 的复位就是通过配置 APB2RSTR 寄存器的第 14 位来实现的。APB2RSTR 寄存器的描述如图 17.1.3.2.1 所示：

| | | | | | | | | | | | | | | | |
|----------|----------|---------|----------|----------|----------|----|----------|----------|-----------|-----------|------------|------------|----------|----------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | DACRST | PWR RST | BKP RST | CAN2 RST | CAN1 RST | 保留 | I2C2 RST | I2C1 RST | UART5R ST | UART4R ST | USART3 RST | USART2 RST | 保留 | | |
| | | 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 |
| SPI3 RST | SPI2 RST | 保留 | WWDG RST | 保留 | 保留 | 保留 | 保留 | 保留 | TIM7 RST | TIM6 RST | TIM5 RST | TIM4 RST | TIM3 RST | TIM2 RST | |
| rW | rW | | rW | | | | | | rW | rW | rW | rW | rW | rW | rW |

图 17.1.3.2.1 RCC_APB2RSTR 寄存器

(3) 串口波特率设置

每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以达到配置不同波特率的目的。在前面的 USART 框图部分描述过，为了让大家更好了解波特率寄存器，下面截取 USART_BRR 寄存器图，如图 17.1.3.2.2 所示：

| | | | | | | | | | | | | | | | |
|--------------------|----|--|----|----|----|----|----|----|----|----|----|-------------------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DIV_Mantissa[11:0] | | | | | | | | | | | | DIV_Fraction[3:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位15:4 | | DIV_Mantissa[11:0]: USARTDIV的整数部分 这12位定义了USART分频器除法因子(USARTDIV)的整数部分。 | | | | | | | | | | | | | |
| 位3:0 | | DIV_Fraction[3:0]: USARTDIV的小数部分 这4位定义了USART分频器除法因子(USARTDIV)的小数部分。 | | | | | | | | | | | | | |

图 17.1.3.2.2 USART_BRR 寄存器

(4) 串口控制

STM32F103 每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3 个寄存器来设置的。USART_CR1 寄存器的描述如图 17.1.3.2.3 所示：

| | | | | | | | | | | | | | | | |
|-----|----|----|------|-----|----|------|-------|------|--------|--------|----|----|-----|-----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | UE | M | WAKE | PCE | PS | PEIE | TXEIE | TCIE | RXNEIE | IDLEIE | TE | RE | RWU | SBK | |
| res | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

图 17.1.3.2.3 USART_CR1 寄存器

该寄存器的高 18 位没有用到，低 14 位用于串口的功能设置。我们在这里只介绍需要用到的一些位，其他位可以参考《STM32F10xxx 参考手册_V10 (中文版).pdf》。UE 为串口使能位，通过该位置 1，使能串口。M 为字长，当该位为 0 的时候设置串口为 8 个字长外加 n 个停止位，停止位的个数 (n) 是根据 USART_CR2 的[13:12]位设置来决定的，默认为 0。PCE 为校验使能位，设置为 0，即禁止校验，否则使能校验。PS 为校验位选择，设置为 0 为偶校验，否则奇校验。TXIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_SR 中的 TXE 位为 1 时，将产生串口中断。TCIE 为发送完成中断使能位，设置该位为 1，当 USART_SR 中的 TC 位为 1 时，将产生串口中断。RXNEIE 为接收缓冲区非空中断使能，设置该位为 1，当 USART_SR 中的 ORE 或者 RXNE 位为 1 时，将产生串口中断。TE 为发送使能位，设置为 1，将开启串口的发送功能。RE 为接收使能位，用法同 TE。

(5) 数据发送与接收

STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到数据的时候，也是存在该寄存器内。在前面的 USART 框图已经对 USART_DR 有详细的介绍，大家可以自行查阅。下面看一下寄存器的各位描述如图 17.1.3.2.4。

| | | | | | | | | | | | | | | | |
|------|----|---|----|----|----|----|----|---------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | DR[8:0] | | | | | | | |
| | | | | | | | | rw | rw | rw | rw | rw | rw | rw | rw |
| 位8:0 | | DR[8:0]: 数据值 (Data value) 包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。TDR寄存器提供了内部总线和输出移位寄存器之间的并行接口(参见图248)。RDR寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位(USART_CR1中PCE位被置位)进行发送时，写到MSB的值(根据数据的长度不同，MSB是第7位或者第8位)会被后来的校验位该取代。 当使能校验位进行接收时，读到的MSB位是接收到的校验位。 | | | | | | | | | | | | | |

图 17.1.3.2.4 USART_DR 寄存器

(6) 串口状态

串口状态通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如图 17.1.3.2.5 所示：

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|-------|-------|-----|-------|-------|------|-----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | CTS | LBD | TXE | TC | RXNE | IDLE | ORE | NE | FE | PE |
| | | | | | | rc w0 | rc w0 | r | rc w0 | rc w0 | r | r | r | r | r |

图 17.1.3.2.5 USART_SR 寄存器

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。

RXNE (读数据寄存器非空)，当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TC (发送完成)，当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如

果设置了这个位的中断，则会产生中断。该位也有两种清零方式：

- 1) 读 USART_SR，写 USART_DR。
- 2) 直接向该位写 0

通过以上一些寄存器的操作再加上 IO 口的配置，我们就可以达到串口最基本的配置了，关于串口更详细的介绍，请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》关于通用同步异步收发器这一章的相关知识。

17.1.4 GPIO 引脚复用功能

我们知道芯片有许多外设，而引脚的资源是很有限的，为了解决这个问题，方法就是引脚复用，这样使得引脚除了作为普通的 IO 口之外，还会与一些外设关联起来，作为第二功能使用，而且一个引脚不单单只有一种复用功能，而是拥有多个第二功能，但是一次只允许一个外设的复用功能，以确保共用同一个 IO 引脚的外设之间不会产生冲突。

下面我们把之前没讲解的复用功能寄存器 AFIO 讲解一下。

AFIO 寄存器的作用就是复用功能 I/O 和调试配置的，STM32F103ZET6 共有 6 个 AFIO 的寄存器，事件控制寄存器 AFIO_EVCR、复用重映射和调试 I/O 配置寄存器 AFIO_MAPR、外部中断配置寄存器 AFIO_EXTICR1、外部中断配置寄存器 AFIO_EXTICR2、外部中断配置寄存器 AFIO_EXTICR3 和外部中断配置寄存器 AFIO_EXTICR4。

在对这些寄存器进行读写操作前，应先打开 AFIO 时钟，该时钟在 RCC_APB2ENR 寄存器上的位 0 上配置，在位 0 上置 0 表示辅助功能 IO 时钟关闭；在位 0 上置 1 表示辅助功能 IO 时钟开启。

事件控制寄存器 AFIO_EVCR，用得比较少这里不作过多介绍。

AFIO_EXTICRx 寄存器，在中断章节再来详细讲解，本章节不涉及这些寄存器的配置。

复用重映射和调试 I/O 配置寄存器 AFIO_MAPR 寄存器描述，如图 17.1.4.1 所示。

| | | | | | | | | | | | | | | | |
|-----------------------------|--------------------|---|----------------|---------------------|--------------|---------------------|----|---------------------|----|-----------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | SWJ_CFG[2:0] | | | 保留 | | | ADC2_E TRGREG _REMAP | ADC2_E TRGINJ _REMAP | ADC1_E TRGREG _REMAP | ADC1_E TRGINJ _REMAP | TIM5CH 4_I REM AP |
| W W W | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PDO1_ REMAP | CAN_REMAP [1:0] | | TIM4_ REMAP | TIM3_REMAP [1:0] | | TIM2_REMAP [1:0] | | TIM1_REMAP [1:0] | | USART3_REMAP [1:0] | | USART2_ _REMAP | USART1_ _REMAP | I2C1_ REMAP | SPI1_ REMAP |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位2 | | USART1_REMAP: USART1的重映像 (USART1 remapping) 该位可由软件置'1'或置'0'，控制USART1的TX和RX复用功能在GPIO端口的映像。 0: 没有重映像(TX/PA9, RX/PA10); 1: 重映像(TX/PB6, RX/PB7)。 | | | | | | | | | | | | | |

图 17.1.4.1 AFIO_MAPR 寄存器

在对 AFIO_MAPR 寄存器某些位进行写入实现引脚的重新映射，这时候，复用功能不再映射到它们原始分配上。例如 AFIO_MAPR 寄存器位 2 是对 USART1 的重映射，置 0: 没有重映像(TX/PA9, RX/PA10); 置 1: 重映像(TX/PB6, RX/PB7)。默认情况下，PA9 和 PA10 是作为串口 1 的引脚使用，假如 PA9 和 PA10 被用作其他地方，但还是需要用到串口 1，那么就可以在 AFIO_MAPR 的位 2 置 1，把串口 1 的引脚重映射到 PB6 和 PB7。这个串口初始化的过程，就有点变化，需要初始化 AFIO 时钟，和对 AFIO_MAPR 的第 2 位进行置 1 操作，其他与普通串口配置没有区别。

HAL 库关于端口复用相关的代码在 **STM32F1xx_hal_gpio_ex.h** 文件中可以找到，USART1 重映射操作代码如下：

```
/**
 * @brief Enable the remapping of USART1 alternate function TX and RX.
 * @note ENABLE: Remap (TX/PB6, RX/PB7)
 * @retval None
 */
```

```
#define __HAL_AFIO_REMAP_USART1_ENABLE() \
    AFIO_REMAP_ENABLE(AFIO_MAPR_USART1_REMAP)

/**
 * @brief Disable the remapping of USART1 alternate function TX and RX.
 * @note DISABLE: No remap (TX/PA9, RX/PA10)
 * @retval None
 */
#define __HAL_AFIO_REMAP_USART1_DISABLE() \
    AFIO_REMAP_DISABLE(AFIO_MAPR_USART1_REMAP)
```

以上是使用重映射的串口 1 的介绍。

17.2 硬件设计

1. 例程功能

LED0 闪烁，提示程序在运行。STM32 通过串口 1 和上位机对话，STM32 在收到上位机发过来的字符串(以回车换行结束)后，会返回给上位机。同时每隔一定时间，通过串口 1 输出一段信息到电脑。

2. 硬件资源

1) LED 灯

LED0 – PB5

2) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340C 上面)，需要跳线帽连接。

3. 原理图

USB 转串口硬件部分的原理图，如图 17.2.1 所示：

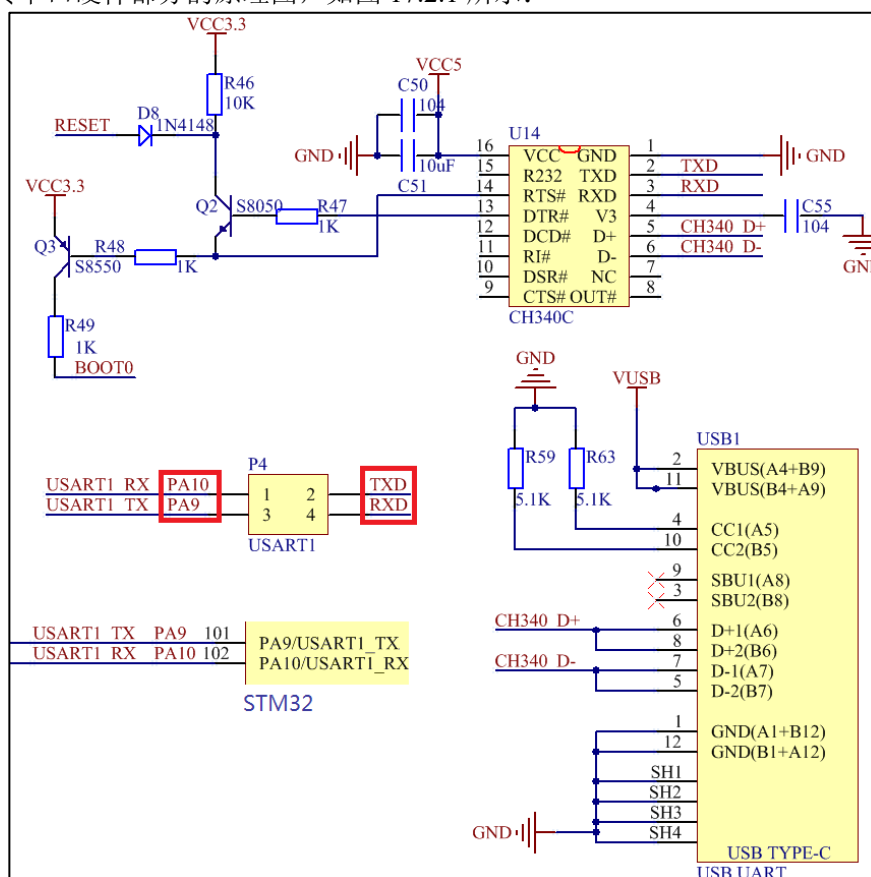


图 17.2.1 USB 转串口原理图

这里需要注意的是：上图中的 P4 的 RXD 和 PA9 用跳线帽连接，以及 TXD 和 PA10 也用跳线帽连接。如图 17.2.2 所示：



图 17.2.2 短路帽连接

17.3 程序设计

17.3.1 USART 的 HAL 库驱动

HAL 库中关于串口的驱动程序比较多，我们主要先来学习本章需要用到的，其余的后续用到再讲解。因为我们现在只是用到异步收发器功能，所以我们现在只需要 STM32F1xx_hal_uart.c 文件（及其头文件）的驱动代码，STM32F1xx_hal_usart.c 是通用同步异步收发器，暂时没有用到，可以暂时不看。用到一个外设第一个函数就应该是其初始化函数。

1. HAL_UART_Init 函数

要使用一个外设首先要对它进行初始化，所以先看串口的初始化函数，其声明如下：

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

- **函数描述：**

用于初始化异步模式的收发器。

- **函数形参：**

形参 1 是串口的句柄，结构体类型是 UART_HandleTypeDef，其定义如下：

```
typedef struct
{
    USART_TypeDef          *Instance;          /* USART 寄存器基地址 */
    UART_InitTypeDef       Init;               /* USART 通信参数 */
    uint8_t                *pTxBuffPtr;        /* 指向 USART 发送缓冲区 */
    uint16_t               TxXferSize;         /* USART 发送数据的大小 */
    __IO uint16_t           TxXferCount;        /* USART 发送数据的个数 */
    uint8_t                *pRxBuffPtr;        /* 指向 USART 接收缓冲区 */
    uint16_t               RxXferSize;         /* USART 接收数据大小 */
    __IO uint16_t           RxXferCount;        /* USART 接收数据的个数 */
    DMA_HandleTypeDef      *hdmatx;            /* USART 发送参数设置 (DMA) */
    DMA_HandleTypeDef      *hdmarx;            /* USART 接收参数设置 (DMA) */
    HAL_LockTypeDef         Lock;              /* 锁定对象 */
    __IO HAL_UART_StateTypeDef gState;          /* USART 发送状态结构体 */
    __IO HAL_UART_StateTypeDef RxState;        /* USART 接收状态结构体 */
    __IO uint32_t           ErrorCode;          /* USART 操作错误信息 */
}UART_HandleTypeDef;
```

- 1) **Instance:** 指向 USART 寄存器基地址。实际上这个基地址 HAL 库已经定义好了，可以选择范围：USART1~USART3、USART4、USART5。
- 2) **Init:** USART 初始化结构体，用于配置通讯参数，如波特率、数据位数、停止位等等。下面我们再详细讲解这个结构体。
- 3) **pTxBuffPtr, TxXferSize, TxXferCount:** 分别是指向发送数据缓冲区的指针，发送数据的大小，发送数据的个数。
- 4) **pRxBuffPtr, RxXferSize, RxXferCount:** 分别是指向接收数据缓冲区的指针，接收数据的大小，接收数据的个数；
- 5) **hdmatx, hdmarx:** 配置串口发送接收数据的 DMA 具体参数。
- 6) **Lock:** 对资源操作增加操作锁保护功能，可选 HAL_UNLOCKED 或者 HAL_LOCKED 两个参数。如果 gState 的值等于 HAL_UART_STATE_RESET，则可认为串口未被初始化，此时，分配锁资源，并且调用 HAL_UART_MspInit 函数来对串口的 GPIO 和时钟进行初始化。
- 7) **gState, RxState:** 分别是 USART 的发送状态、工作状态的 struct 体和 USART 接受状态的 struct 体。

体。HAL_UART_StateTypeDef 是一个枚举类型，列出串口在工作过程中的状态值，有些值只适用于 gState，如 HAL_UART_STATE_BUSY。

- 8) **ErrorCode:** 串口错误操作信息。主要用于存放串口操作的错误信息。

下面，我们来了解 UART_InitTypeDef 这个结构体类型，该结构体用于配置 UART 的各个通信参数，包括波特率，停止位等，具体说明如下：

```
typedef struct
{
    uint32_t BaudRate;          /* 波特率 */
    uint32_t WordLength;        /* 字长 */
    uint32_t StopBits;          /* 停止位 */
    uint32_t Parity;             /* 校验位 */
    uint32_t Mode;               /* UART 模式 */
    uint32_t HwFlowCtl;          /* 硬件流设置 */
    uint32_t OverSampling;       /* 过采样设置 */
}UART_InitTypeDef;
```

- 1) **BaudRate:** 波特率设置。一般设置为 2400、9600、19200、115200。
- 2) **WordLength:** 数据帧字长，可选 8 位或 9 位。这里我们设置为 8 位字长数据格式。
- 3) **StopBits:** 停止位设置，可选 0.5 个、1 个、1.5 个和 2 个停止位，一般我们选择 1 个停止位。
- 4) **Parity:** 奇偶校验控制选择，我们设定为无奇偶校验位。
- 5) **Mode:** UART 模式选择，可以设置为只收模式，只发模式，或者收发模式。这里我们设置为全双工收发模式。
- 6) **HwFlowCtl:** 硬件流控制选择，我们设置为无硬件流控制。
- 7) **OverSampling:** 过采样选择，选择 8 倍过采样或者 16 过采样，一般选择 16 过采样。

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值，有 4 个，分别是 HAL_OK 表示成功，HAL_ERROR 表示错误，HAL_BUSY 表示忙碌，HAL_TIMEOUT 超时。后续遇到该结构体也是一样的。

2. HAL_UART_Receive_IT 函数

HAL_UART_Receive_IT 函数是开启串口接收中断函数。其声明如下：

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                       uint8_t *pData, uint16_t Size);
```

● 函数描述：

用于开启以中断的方式接收指定字节。数据接收在中断处理函数里面实现。

● 函数形参：

形参 1 是 UART_HandleTypeDef 结构体指针类型的串口句柄。

形参 2 是要接收的数据地址。

形参 3 是要接收的数据大小，以字节为单位。

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

3. HAL_UART_IRQHandler 函数

HAL_UART_IRQHandler 函数是 HAL 库中断处理公共函数。其声明如下：

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

● 函数描述：

该函数是 HAL 库中断处理公共函数，在串口中断服务函数中被调用。

● 函数形参：

形参 1 是 UART_HandleTypeDef 结构体指针类型的串口句柄。

● 函数返回值：

无

● 注意事项：

该函数是 HAL 库已经定义好，用户一般不能随意修改。如果用户要在中断中实现自己的逻辑代码，可以直接在函数 HAL_UART_IRQHandler 的前面或者后面添加新代码，也可以直接在 HAL_UART_IRQHandler 调用的各种回调函数里面执行，这些回调都是弱定义的，方便用户直接在其它文件里面重定义。串口回调函数主要有下面几个：

```
__weak void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortTransmitCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortReceiveCpltCallback(UART_HandleTypeDef *huart)
```

本实验我们用到的是接收回调函数 HAL_UART_RxCpltCallback，就是在接收回调函数里面编写我们的接收逻辑代码，具体请参考实验源码。

串口通讯配置步骤

1) 串口参数初始化（波特率、字长、奇偶校验等）

HAL 库通过调用串口初始化函数 HAL_UART_Init 完成对串口参数初始化，详见例程源码

注意：该函数会调用：HAL_UART_MspInit 函数来完成对串口底层的初始化，包括：串口及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2) 使能串口和 GPIO 口时钟

本实验用到 USART1 串口，使用 PA9 和 PA10 作为串口的 TX 和 RX 脚，因此需要先使能 USART1 和 GPIOA 时钟。参考代码如下：

```
_HAL_RCC_USART1_CLK_ENABLE(); /* 使能 USART1 时钟 */
_HAL_RCC_GPIOA_CLK_ENABLE(); /* 使能 GPIOA 时钟 */
```

3) GPIO 模式设置（速度、上下拉、复用功能等）

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

4) 开启串口相关中断，配置串口中断优先级

本实验我们使用串口中断来接收数据。我们使用 HAL_UART_Receive_IT 函数开启串口中断接收，并设置接收 buffer 及其长度。通过 HAL_NVIC_EnableIRQ 函数使能串口中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5) 编写中断服务函数

串口 1 中断服务函数为：USART1_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个串口中断通用处理函数 HAL_UART_IRQHandler，该函数在串口接收完数据后，又会调用回调函数 HAL_UART_RxCpltCallback，用于给用户处理串口接收到的数据。

因此我们需要在 HAL_UART_RxCpltCallback 函数实现数据接收处理，详见本例程源码。

6) 串口数据接收和发送

最后我们可以通过读写 USART_DR 寄存器，完成串口数据的接收和发送，HAL 库也给我们提供了：HAL_UART_Receive 和 HAL_UART_Transmit 两个函数用于串口数据的接收和发送。

大家可以根据实际情况选择使用以上介绍的方式来收发串口数据。

17.3.2 程序流程图

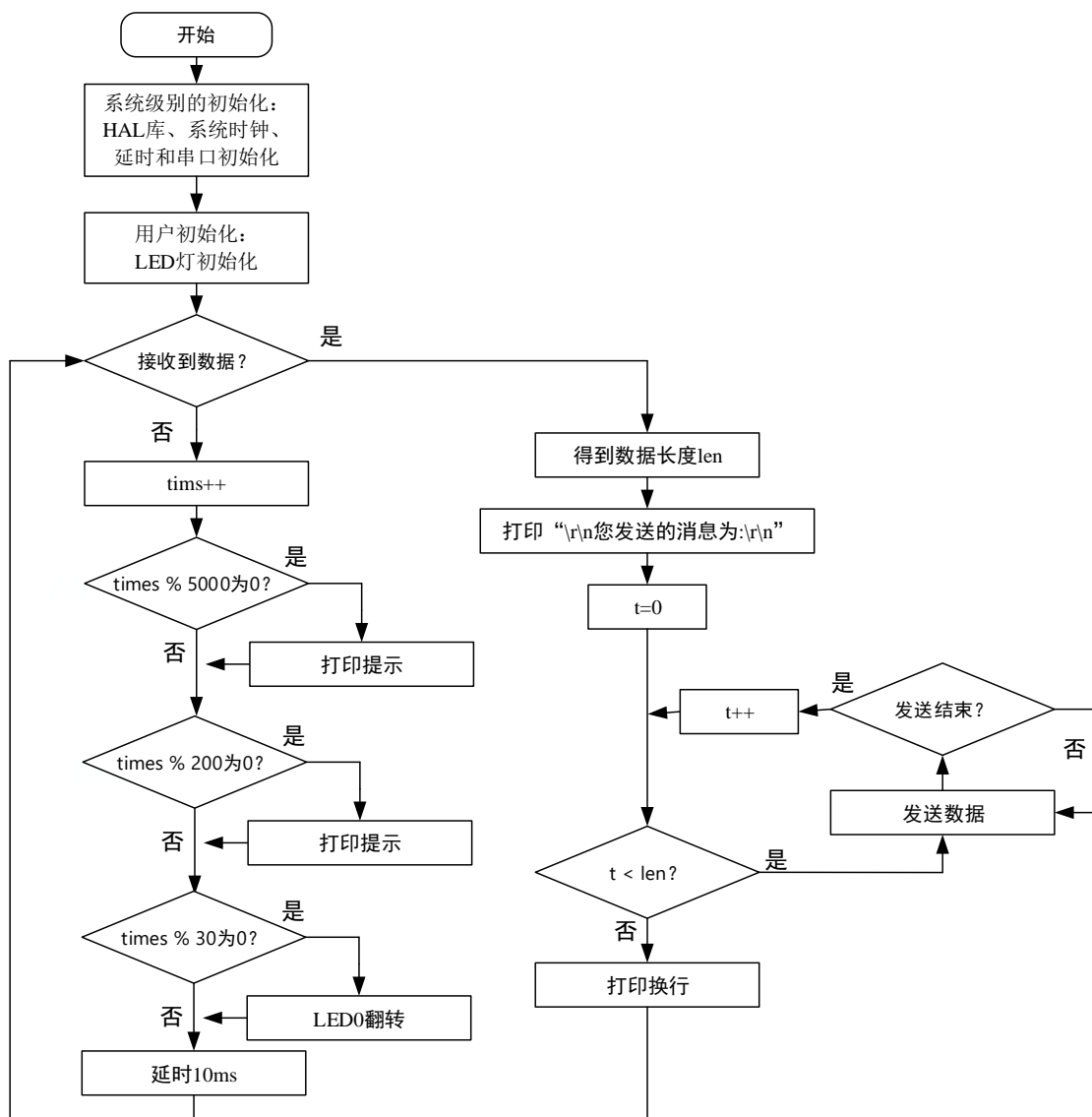


图 17.3.2.1 串口通信实验程序流程图

17.3.3 程序解析

1. 串口1驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。串口1(USART1)驱动源码包括两个文件：usart.c 和 usart.h。

下面我们先解析 usart.h 的程序。

● 串口1引脚定义

由硬件设计小节，我们知道 PA9 和 PA10 分别被复用为串口1的发送和接收引脚，我们做了下面的引脚定义。

```

/* 串口1的GPIO */
#define USART_TX_GPIO_PORT      GPIOA
#define USART_TX_GPIO_PIN      GPIO_PIN_9
/* 发送引脚时钟使能 */
#define USART_TX_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)
#define USART_RX_GPIO_PORT      GPIOA
    
```



```
#define USART_RX_GPIO_PIN      GPIO_PIN_10
/* 接收引脚时钟使能 */
#define USART_RX_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define USART_UX                USART1
#define USART_UX_IRQn           USART1_IRQn
#define USART_UX_IRQHandler     USART1_IRQHandler
/* USART1 时钟使能 */
#define USART_UX_CLK_ENABLE()   do{ __HAL_RCC_USART1_CLK_ENABLE(); }while(0)
```

USART1_IRQn 也就是我们中断向量的 37 号中断，USART1_IRQHandler 是串口 1 的中断服务函数。每个串口都有自己的中断函数，但是我们最终都是通过回调函数去实现逻辑代码，当然我们亦可在中断函数里实现逻辑代码。

另外我们还定义了三个宏，具体如下：

```
#define USART_REC_LEN    200      /* 定义最大接收字节数 200 */
#define USART_EN_RX      1        /* 使能 (1) / 禁止 (0) 串口 1 接收 */
#define RXBUFFERSIZE     1        /* 缓存大小 */
```

可以看到 USART_REC_LEN 表示最大接收字节数，这里定义的是 200 个字节，后续如果有需求要发送更大的数据包，可以改大这个值，这里不改太大，是避免浪费太多内存。USART_EN_RX 则是用于使能串口 1 的接收数据。RXBUFFERSIZE 是缓冲大小。

下面我们再解析 `usart.c` 的程序，先看串口 1 的初始化函数，其定义如下：

```
/**
 * @brief      串口 x 初始化函数
 * @param      baudrate: 波特率，根据自己需要设置波特率值
 * @note       注意：必须设置正确的时钟源，否则串口波特率就会设置异常。
 *             这里的 USART 的时钟源在 sys_stm32_clock_init() 函数中已经设置过了。
 * @retval     无
 */
void usart_init(uint32_t baudrate)
{
    uartx_handle.Instance = USART_UX;                /* USART1 */
    uartx_handle.Init.BaudRate = baudrate;            /* 波特率 */
    uartx_handle.Init.WordLength = UART_WORDLENGTH_8B; /* 字长为 8 位数据格式 */
    uartx_handle.Init.StopBits = UART_STOPBITS_1;     /* 一个停止位 */
    uartx_handle.Init.Parity = UART_PARITY_NONE;      /* 无奇偶校验位 */
    uartx_handle.Init.HwFlowCtl = UART_HWCONTROL_NONE; /* 无硬件流控 */
    uartx_handle.Init.Mode = UART_MODE_TX_RX;         /* 收发模式 */
    HAL_UART_Init(&uartx_handle);                    /* HAL_UART_Init() 会使能 USART1 */

    /* 该函数会开启接收中断：标志位 UART_IT_RXNE，并且设置接收缓冲以及接收缓冲接收最大数据量 */
    HAL_UART_Receive_IT(&uartx_handle, (uint8_t *)aRxbuffer, RXBUFFERSIZE);
}
```

uartx_handle 是结构体 UART_HandleTypeDef 类型的全局变量，UART_HandleTypeDef 结构体成员的含义请回到前面回顾。波特率我们直接赋值给 uartx_handle.Init.BaudRate 这个成员，可以看出很方便。需要注意的是，最后一行代码调用函数 HAL_UART_Receive_IT，作用是开启接收中断，同时设置接收的缓存区以及接收的数据量。

上面的初始化函数只是串口初始化的其中一部分，我们还有一部分初始化需要 HAL_UART_MspInit 函数去完成。HAL_UART_MspInit 是 HAL 库定义的弱定义函数，这里我们做重定义以实现我们的初始化需求。HAL_UART_MspInit 函数在 HAL_UART_Init 函数中会被调用，其定义如下：

```
/**
 * @brief      UART 底层初始化函数
 * @param      huart: UART 句柄类型指针
 * @note       此函数会被 HAL_UART_Init() 调用
 *             完成时钟使能，引脚配置，中断配置
 * @retval     无
 */
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
}
```

```

GPIO_InitTypeDef gpio_init_struct;
if(huart->Instance == USART1)                /* 如果是串口 1, 进行串口 1 MSP 初始化 */
{
    USART_UX_CLK_ENABLE();                    /* USART1 时钟使能 */
    USART_TX_GPIO_CLK_ENABLE();               /* 发送引脚时钟使能 */
    USART_RX_GPIO_CLK_ENABLE();               /* 接收引脚时钟使能 */

    gpio_init_struct.Pin = USART_TX_GPIO_PIN; /* TX 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;  /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;      /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(USART_TX_GPIO_PORT, &gpio_init_struct); /* 初始化发送引脚 */

    gpio_init_struct.Pin = USART_RX_GPIO_PIN; /* RX 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_AF_INPUT;
    HAL_GPIO_Init(USART_RX_GPIO_PORT, &gpio_init_struct); /* 初始化接收引脚 */

#ifdef USART_EN_RX
    HAL_NVIC_EnableIRQ(USART_UX_IRQn);        /* 使能 USART1 中断通道 */
    HAL_NVIC_SetPriority(USART_UX_IRQn, 3, 3); /* 抢占优先级 3, 子优先级 3 */
#endif
}
}

```

怎么来理解这个函数呢？该函数主要实现底层的初始化，事实上这个函数的代码还可以直接放到 `usart_init` 函数里面，但是 HAL 库为了代码的功能分层初始化，定义这个函数方便用户使用。所以我们也按照 HAL 库的这个结构来初始化外设。这个函数首先是调用 `if(huart->Instance == USART1)` 判断是要初始化哪个串口，因为每个串口初始化都会调用 `HAL_UART_MspInit` 这个函数，所以需要判断是哪个串口要初始化才做相应的处理。只能说 HAL 库这样的结构机制有好处，自然也有坏处。

首先就是使能串口以及 PA9 和 PA10 的时钟，PA9 和 PA10 需要用做复用功能，复用功能模式有两个选择：GPIO_MODE_AF_PP 推挽式复用和 GPIO_MODE_AF_OD 开漏式复用，我们选择的是推挽式复用，因为 PA9 是一个发送管脚，所以模式设置为复用推挽输出，而 PA10 是一个接收管脚，所以它的模式设置为浮空输入即可，GPIO_MODE_AF_INPUT 实质就是一个 GPIO_MODE_INPUT 浮空输入模式。选择了推挽式复用。然后就是调用 `HAL_GPIO_Init` 函数进行 IO 口的初始化。

最后因为我们用到串口中断，所以还需要中断相关的配置。`HAL_NVIC_EnableIRQ` 函数使能串口 1 复用通道。`HAL_NVIC_SetPriority` 函数配置串口中断的抢占优先级以及响应优先级。串口初始化由上述两个函数完成。

下面就该讲到串口中断服务函数了，其定义如下：

```

/**
 * @brief      串口 1 中断服务函数
 * @param      无
 * @retval     无
 */
void USART_UX_IRQHandler(void)
{
#ifdef SYS_SUPPORT_OS                /* 使用 OS */
    OSIntEnter();
#endif

    HAL_UART_IRQHandler(&g_uart1_handle); /* 调用 HAL 库中断处理公用函数 */

#ifdef SYS_SUPPORT_OS                /* 使用 OS */
    OSIntExit();
#endif
}

```

在中断服务函数里主要就调用 HAL 库的串口中断公共处理函数 `HAL_UART_IRQHandler()`，然后该函数内部再调用相关的中断回调函数，我们这里用到的是串口接收完成中断回调函数

HAL_UART_RxCpltCallback 来处理用户的逻辑代码。

下面主要来看看 HAL_UART_RxCpltCallback 函数的定义：

```
/**
 * @brief      UART 数据接收回调接口
 *              数据处理在这里进行
 * @param      huart:串口句柄
 * @retval      无
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance == USART_1)           /* 如果是串口 1 */
    {
        if((g_usart_rx_sta & 0x8000) == 0)     /* 接收未完成 */
        {
            if(g_usart_rx_sta & 0x4000)        /* 接收到了 0x0d (即回车键) */
            {
                if(aRxBuffer[0] != 0x0a)      /* 接收到的不是 0x0a (即不是换行键) */
                {
                    g_usart_rx_sta = 0;        /* 接收错误, 重新开始 */
                }
                else                          /* 接收到的是 0x0a (即换行键) */
                {
                    g_usart_rx_sta |= 0x8000;  /* 接收完成了 */
                }
            }
            else                               /* 还没收到 0x0D (即回车键) */
            {
                if (aRxBuffer[0] == 0x0d)
                {
                    g_usart_rx_sta |= 0x4000;
                }
                else
                {
                    g_usart_rx_buf[g_usart_rx_sta & 0X3FFF] = aRxBuffer[0];
                    g_usart_rx_sta++;
                    if (g_usart_rx_sta > (USART_REC_LEN - 1))
                    {
                        g_usart_rx_sta = 0;     /* 接收数据错误, 重新开始接收 */
                    }
                }
            }
        }
        HAL_UART_Receive_IT(&g_uart1_handle, (uint8_t *)g_rx_buffer,
                            RXBUFFERSIZE);
    }
}
```

因为我们设置了串口句柄成员变量 RxXferSize 为 1，那么每当串口 1 接收到一个字符后触发接收完成中断，便会在中断服务函数中引导执行该回调函数。当串口接受到一个字符后，它会保存在缓存 g_rx_buffer 中，由于我们设置了缓存大小为 1，而且 RxXferSize=1，所以每次接受一个字符，会直接保存到 RxXferSize[0]中，我们直接通过读取 RxXferSize[0]的值就是本次接收到的字符。这里我们设计了一个小小的接收协议：通过这个函数，配合一个数组 g_usart_rx_buf，一个接收状态寄存器 g_usart_rx_sta（此寄存器其实就是一个全局变量，由作者自行添加。由于它起到类似寄存器的功能，这里暂且称之为寄存器）实现对串口数据的接收管理。数组 g_usart_rx_buf 的大小由 USART_REC_LEN 定义，也就是一次接收的数据最大不能超过 USART_REC_LEN 个字节。g_usart_rx_sta 是一个接收状态寄存器其各的定义如表 17.3.3.1 所示：

| g_usart_rx_sta | | |
|----------------|-------------|------------|
| bit15 | bit14 | bit13~0 |
| 接收完成标志 | 接收到 0X0D 标志 | 接收到的有效字节个数 |

表 15.3.3.1 接收状态寄存器位定义表

设计思路如下：

当接收到从电脑发过来的数据，把接收到的数据保存在数组 `g_usart_rx_buf` 中，同时在接收状态寄存器（`g_usart_rx_sta`）中计数接收到的有效数据个数，当收到回车（回车的表示由 2 个字节组成：0X0D 和 0X0A）的第一个字节 0X0D 时，计数器将不再增加，等待 0X0A 的到来，而如果 0X0A 没有来到，则认为这次接收失败，重新开始下一次接收。如果顺利接收到 0X0A，则标记 `g_usart_rx_sta` 的第 15 位，这样完成一次接收，并等待该位被其他程序清除，从而开始下一次的接收，而如果迟迟没有收到 0X0D，那么在接收数据超过 `USART_REC_LEN` 的时候，则会丢弃前面的数据，重新接收。

可以看到，在回调函数后面调用了 `UART_Receive_IT` 函数。该函数在这里的主要作用是重新开启接收完成中断。

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

2. main.c 代码

在 `main.c` 里面编写如下代码：

```
int main(void)
{
    uint8_t len;
    uint16_t times = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟为 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */

    while (1)
    {
        if (g_usart_rx_sta & 0x8000) /* 接收到了数据? */
        {
            len = g_usart_rx_sta & 0x3fff; /* 得到此次接收到的数据长度 */
            printf("\r\n 您发送的消息为:\r\n");

            /*发送接收到的数据*/
            HAL_UART_Transmit(&uartx_handler, (uint8_t*)g_usart_rx_buf, len, 1000);
            /*等待发送结束*/
            while(__HAL_UART_GET_FLAG(&uartx_handler, UART_FLAG_TC) != SET);

            printf("\r\n\r\n"); /* 插入换行 */
            g_usart_rx_sta = 0;
        }
        else
        {
            times++;
            if (times % 5000 == 0)
            {
                printf("\r\n 正点原子 STM32 开发板 串口实验\r\n");
                printf("正点原子@ALIENTEK\r\n\r\n\r\n");
            }

            if (times % 200 == 0) printf("请输入数据,以回车键结束\r\n");

            if (times % 30 == 0) LED0_TOGGLE(); /* 闪烁 LED,提示系统正在运行. */

            delay_ms(10);
        }
    }
}
```

```
}  
}  
}
```

我们主要看无限循环里面的逻辑：首先判断全局变量 `g_usart_rx_sta` 的最高位是否为 1，如果为 1 的话，那么代表前一次数据接收已经完成，接下来就是把我们自定义接收缓冲的数据发送到串口，在上位机显示。这里比较重点的两条语句是：第一条是调用 HAL 串口发送函数 `HAL_UART_Transmit` 来发送一段字符到串口。第二条是我们发送一个字节之后，要检测这个数据是否已经被发送完成了。如果全局变量 `g_usart_rx_sta` 的最高位为 0，则执行一段时间往上位机发送提示字符，以及让 LED0 每隔一段时间翻转，提示系统正在运行。

17.4 下载验证

在下载好程序后，可以看到板子上的 LED0 开始闪烁，说明程序已经在跑了。串口调试助手，我们用 XCOM，该软件在光盘有提供，且无需安装，直接可以运行，但是需要你的电脑安装有 .NET Framework 4.0(WIN 自带了)或以上版本的环境才可以，该软件的详细介绍请看：<http://www.openedv.com/posts/list/22994.htm> 这个帖子。

接着我们打开 XCOM（正点原子的串口调试助手，位于光资料盘(A 盘)→6，软件资料→1，软件→串口调试助手→XCOM），设置串口为开发板的 USB 转串口（CH340 虚拟串口，得根据你自己的电脑选择，我的电脑是 COM15，另外，请注意：波特率是 115200）。因为我们在程序上面设置了必须输入回车，串口才认可接收到的数据，所以必须在发送数据后再发送一个回车符，这里 XCOM 提供的发送方法是通过勾选发送新行实现，只要勾选了这个选项，每次发送数据后，XCOM 都会自动多发一个回车(0X0D+0X0A)。设置好了发送新行，我们再在发送区输入你想要发送的文字，然后单击发送，可以看到如图 15.4.1 所示信息：

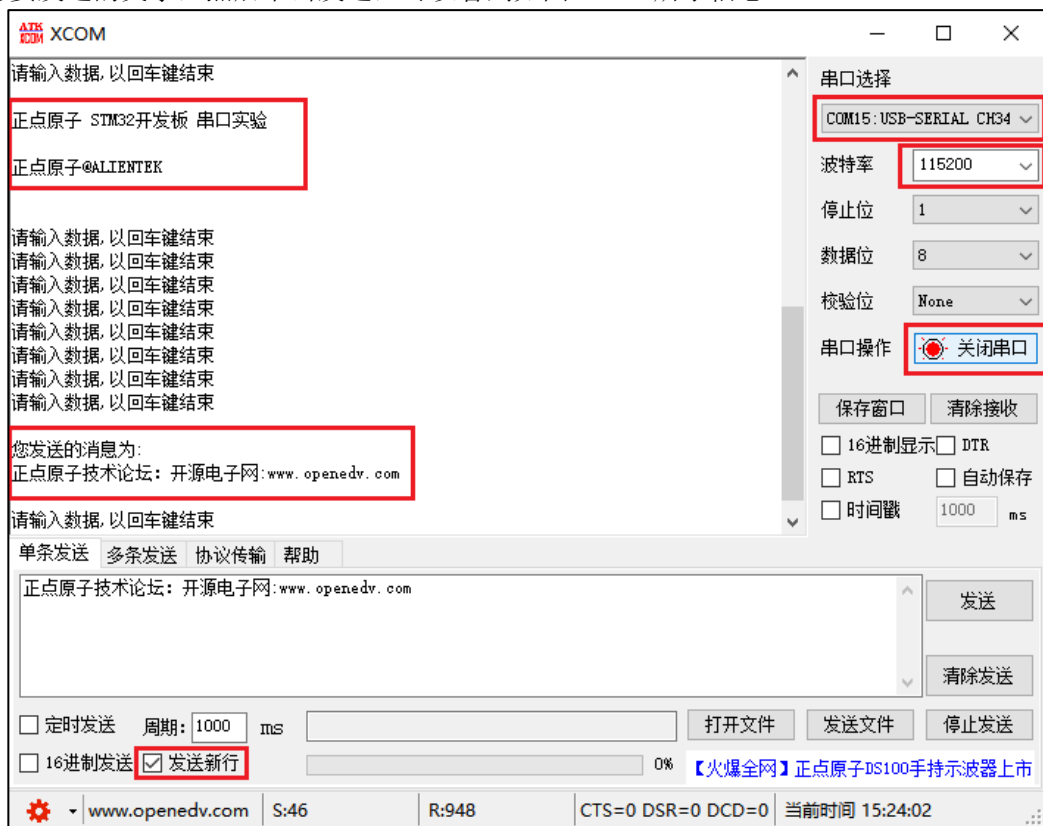


图 15.4.1 串口助手

可以看到，我们发送的消息被发送回来了。大家可以试试，如果不发送回车（取消发送新行），在输入内容之后，直接按发送是什么结果，大家测试一下吧。

第十八章 独立看门狗（IWDG）实验

本章我们学习如何使用 STM32F1 的独立看门狗（以下简称 IWDG）。STM32F1 内部自带了 2 个看门狗：独立看门狗（IWDG）和窗口看门狗（WWDG）。这一章我们只介绍独立看门狗，窗口看门狗将在下一章介绍。在本章中，我们将通过按键 KEY_UP 来喂狗，然后通过 LED0 提示复位状态。

本章分为如下几个小节：

18.1 IWDG 简介

18.2 硬件设计

18.3 程序设计

18.4 下载验证

18.1 IWDG 简介

独立看门狗本质上是一个定时器，这个定时器有一个输出端，可以输出复位信号。该定时器是一个 12 位的递减计数器，当计数器的值减到 0 的时候，就会产生一个复位信号。如果在计数没减到 0 之前，重置计数器的值的话，那么就不会产生复位信号，这个动作我们称为喂狗。看门狗功能由 VDD 电压域供电，在停止模式和待机模式下仍然可以工作。

18.1.1 IWDG 框图

下面先来学习 IWDG 框图，通过学习 IWDG 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

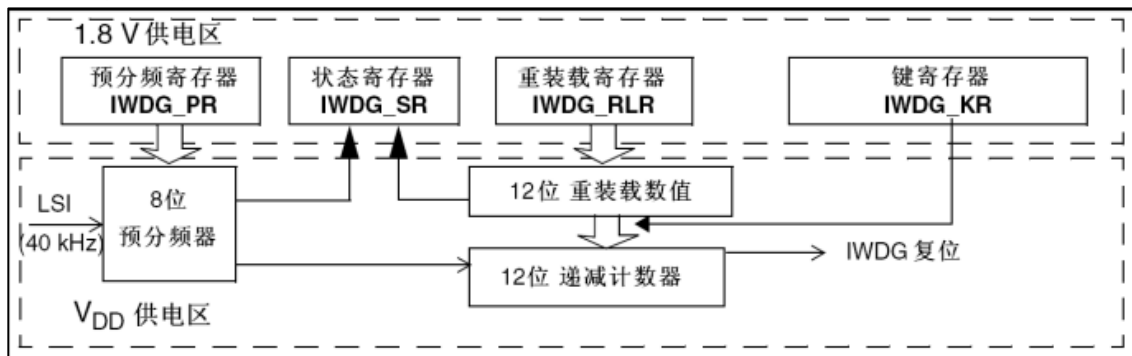


图 18.1.1.1 IWDG 框图

从 IWDG 框图整体认知就是，IWDG 有一个输入（时钟 LSI），经过一个 8 位的可编程预分频器提供时钟给一个 12 位递减计数器，满足条件就会输出一个复位信号。IWDG 内部输入/输出信号如下表：

| 信号名称 | 信号类型 | 说明 |
|---------|------|-------------|
| LSI | 数字信号 | LSI 时钟 |
| IWDG 复位 | 数字信号 | IWDG 复位信号输出 |

表 18.1.1.1 IWDG 内部输入/输出信号

STM32F103 的独立看门狗由内部专门的 40Khz 低速时钟（LSI）驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 40Khz，而是在 30~60Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 40Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

18.1.2 IWDG 寄存器

IWDG 的框图很简单，用到的寄存器也不多。我们主要用到其中 3 个寄存器：

● 键寄存器 (IWDG_KR)

键寄存器可以看作是独立看门狗的控制寄存器，其描述如图 18.1.2.1 所示：

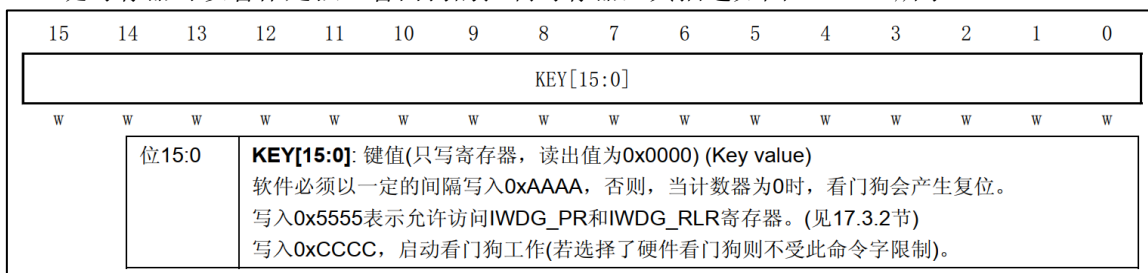


图 18.1.2.1 WDG_KR 寄存器

在键寄存器(IWDG_KR)中写入 0xCCCC，开始启用独立看门狗；此时计数器开始从其复位值 0xFFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号(IWDG_RESET)。无论何时，只要键寄存器 IWDG_KR 中被写入 0xAAAA，IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位。

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

● 预分频寄存器 (IWDG_PR)

预分频寄存器描述如图 18.1.2.2 所示：

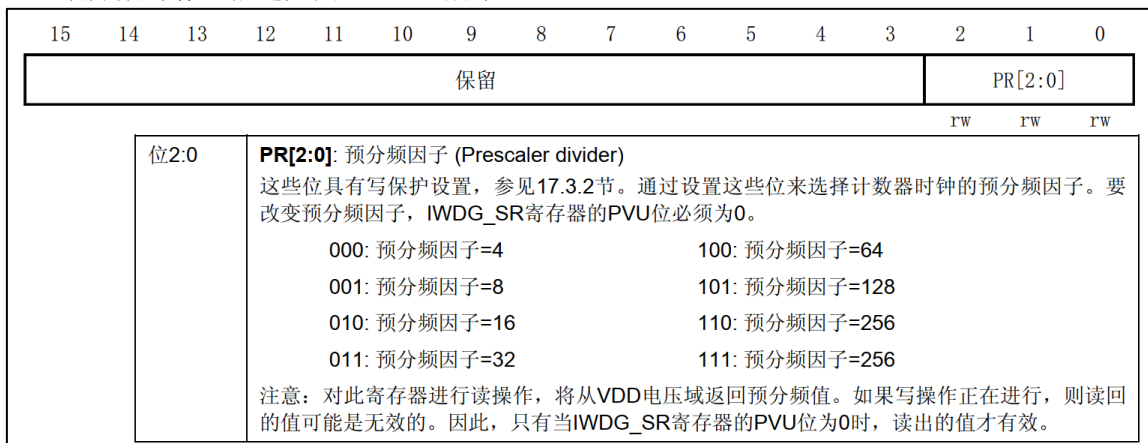


图 18.1.2.2 IWDG_PR 寄存器

该寄存器用来设置看门狗时钟 (LSI) 的分频系数，最低为 4，最高位 256，该寄存器是一个 32 位的寄存器，但是我们只用了最低 3 位，其他都是保留位。

● 重载寄存器 (IWDG_RLR)

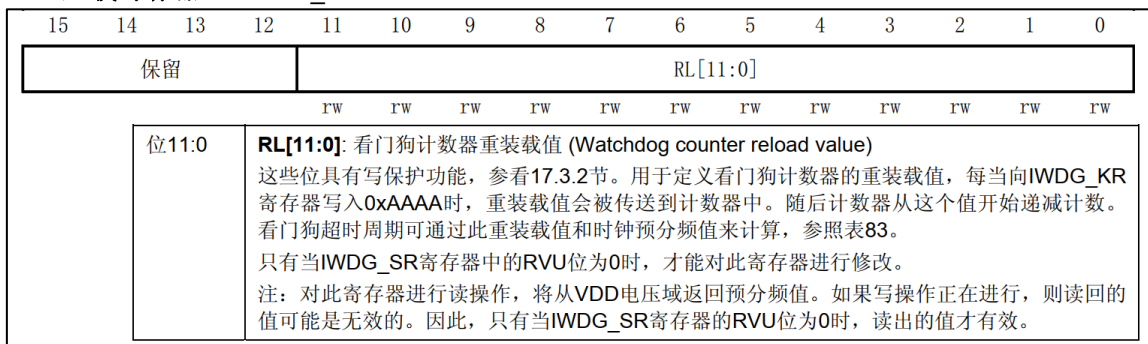


图 18.1.2.3 IWDG_RLR 寄存器

该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器，只有低 12 位是有效的。

18.2 硬件设计

1. 例程功能

在配置看门狗后，LED0 将常亮，如果 KEY_UP 按键按下，就喂狗，只要 KEY_UP 不停的按，看门狗就一直不会产生复位，保持 LED0 的常亮，一旦超过看门狗定溢出时间（Tout）还没按，那么将会导致程序重启，这将导致 LED0 熄灭一次。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
WK_UP - PA0。
- 3) 独立看门狗

3. 原理图

独立看门狗实验的核心是在 STM32F103 内部进行，并不需要外部电路。但是考虑到指示当前状态和喂狗等操作，我们需要 2 个 IO 口，一个用来触发喂狗信号，另外一个用来指示程序是否重启。喂狗我们采用板上的 KEY_UP 键来操作，而程序重启，则是通过 LED0 来指示的。

18.3 程序设计

18.3.1 IWDG 的 HAL 库驱动

IWDG 在 HAL 库中的驱动代码在 stm32f1xx_hal_iwdg.c 文件（及其头文件）中。

1. HAL_IWDG_Init 函数

IWDG 的初始化函数，其声明如下：

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

● 函数描述：

用于初始化 IWDG。

● 函数形参：

形参 1 是 IWDG 句柄，IWDG_HandleTypeDef 结构体类型，其定义如下：

```
typedef struct
{
    IWDG_TypeDef          *Instance; /* IWDG 寄存器基地址 */
    IWDG_InitTypeDef      Init;      /* IWDG 初始化参数 */
} IWDG_HandleTypeDef;
```

1) Instance: 指向 IWDG 寄存器基地址。

2) Init: IWDG 初始化结构体，用于配置计数器的相关参数。

IWDG_InitTypeDef 这个结构体类型定义如下：

```
typedef struct
{
    uint32_t Prescaler; /* 预分频系数 */
    uint32_t Reload;     /* 重装载值 */
} IWDG_InitTypeDef;
```

1) Prescaler: 预分频系数，IWDG_PRESCALER_4 到 IWDG_PRESCALER_256。

2) Reload: 重装载值，范围：0 到 0x0FFF。

3) Window: 窗口值。

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

2. HAL_IWDG_Refresh 函数

HAL_IWDG_Refresh 函数是独立看门狗的喂狗函数。其声明如下：

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

- **函数描述：**
用于把重装载寄存器的值重载到计数器中，喂狗，防止 IWDG 复位。
- **函数形参：**
形参 1 是 IWDG_HandleTypeDef 结构体指针类型的 IWDG 句柄。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。

独立看门狗配置步骤

1) 取消寄存器写保护，设置看门狗预分频系数和重装载值

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

通过该函数设置看门狗的分频系数，和重装载的值。看门狗的喂狗时间（也就是看门狗溢出时间）的计算方式为：

$$T_{out} = ((4 \times 2^{prer}) \times rlr) / 40$$

其中 Tout 为看门狗溢出时间（单位为 ms）。

prer 为看门狗时钟预分频值（IWDG_PR 值），范围为 0~7。

rlr 为看门狗的重装载值（IWDG_RLR 的值）。

比如我们设定 prer 值为 4（4 代表的是 64 分频，HAL 库中可以使用宏定义标识符 IWDG_PRESCALER_64），定时值为 Tout=1 秒，那么就可以得到 $T_{out} = 64 \times rlr / 40KHz = 1s$ ，这样，得到看门狗的溢出时间要为 1s 需要设置 rlr 为 625，只要你在一秒钟之内，有一次写入 0XAAAA 到 IWDG_KR，就不会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 40KHz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

2) 重载计数值喂狗（向 IWDG_KR 写入 0XAAAA）

在 HAL 中重载计数值的函数是 HAL_IWDG_Refresh，该函数的作用是把值 0xAAAA 写入到 IWDG_KR 寄存器，从而触发计数器重载，即实现独立看门狗的喂狗操作。

3) 启动看门狗(向 IWDG_KR 写入 0XCCCC)

HAL 库函数里面启动独立看门狗是通过宏定义标识符来实现的：

```
#define __HAL_IWDG_START(__HANDLE__)  
    WRITE_REG((__HANDLE__)->Instance->KR, IWDG_KEY_ENABLE);
```

我们只需要调用宏定义标识符 __HAL_IWDG_START 即可实现看门狗使能。实际上，当我们调用了看门狗初始化函数 HAL_IWDG_Init 之后，在内部已经调用了该宏启动看门狗。

18.3.2 程序流程图

下面看看本实验的程序流程图：

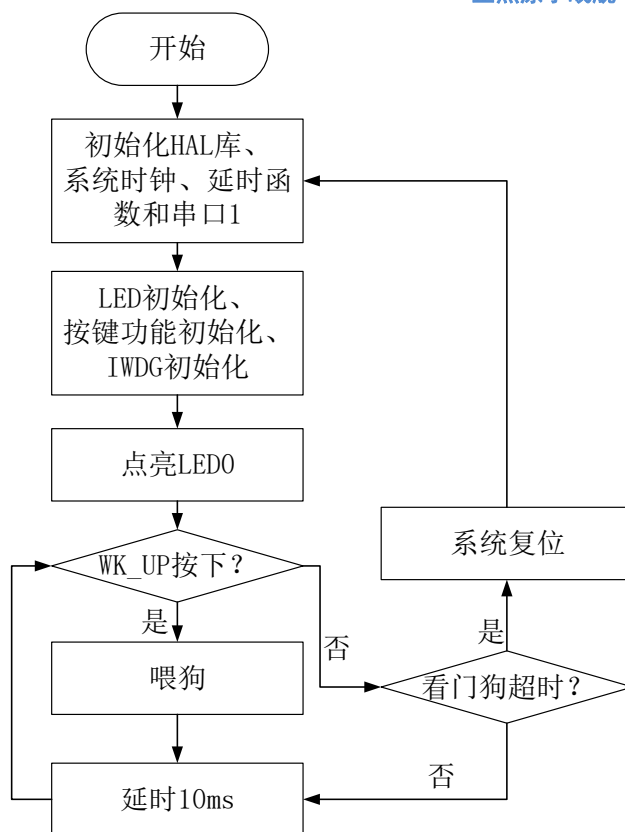


图 18.3.2.1 独立看门狗实验程序流程图

18.3.3 程序解析

1. IWDG 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。独立看门狗（IWDG）驱动源码包括两个文件：**wdg.c** 和 **wdg.h**。

wdg.h 头文件只有函数的声明，就不解释了。下面我们直接解析 **wdg.c** 的程序，先看 IWDG 的初始化函数，其定义如下：

```

/**
 * @brief      初始化独立看门狗
 * @param      prer: IWDG_PRESCALER_4~IWDG_PRESCALER_256,对应 4~256 分频
 * @arg        分频因子 = 4 * 2^prer. 但最大值只能是 256!
 * @param      rlr: 自动重装载值,0~0xFFFF.
 * @note       时间计算(大概):Tout=((4 * 2^prer) * rlr) / 40 (ms).
 * @retval     无
 */
void iwdg_init(uint8_t prer, uint16_t rlr)
{
    g_iwdg_handle.Instance = IWDG1;
    g_iwdg_handle.Init.Prescaler = prer;          /* 设置 IWDG 分频系数 */
    g_iwdg_handle.Init.Reload = rlr;              /* 重装载值 */
    g_iwdg_handle.Init.Window = IWDG_WINDOW_DISABLE; /* 关闭窗口功能 */
    HAL_IWDG_Init(&g_iwdg_handle);
}
    
```

IWDG_Init 是独立看门狗初始化函数，主要设置预分频数和重装载寄存器的值。通过这两个寄存器，就可以大概知道看门狗复位的时间周期为多少了。

```

/**
 * @brief      喂独立看门狗
 * @param      无
    
```

```

* @retval    无
*/
void iwdg_feed(void)
{
    HAL_IWDG_Refresh(&g_iwdg_handle); /* 重装载计数器 */
}

```

iwdg_feed 函数用来喂狗，在该函数内部只需调用 HLA 库函数 HAL_IWDG_Refresh。

2. main.c 代码

在 main.c 里面编写如下代码：

```

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */
    delay_ms(100); /* 延时 100ms 再初始化看门狗, LED0 的变化"可见" */
    iwdg_init(IWDG_PRESCALER_64, 625); /* 预分频数 64, 重载值为 625, 溢出时间约为 1s */
    LED0(0); /* 点亮 LED0 (红灯) */

    while (1)
    {
        if (key_scan(1) == WKUP_PRES) /* 如果 WK_UP 按下, 则喂狗 */
        {
            iwdg_feed(); /* 喂狗 */
        }
        delay_ms(10);
    }
}

```

在 main 函数里，先初始化系统和用户的外设代码，然后先点亮 LED0，在无限循环里开始获取按键的键值，并判断是不是按键 WK_UP 按下，是的话就喂狗，不是则延时 10ms，继续上述操作。当 1 秒钟后都没测到按键 WK_UP 按下，IWDG 就会产生一次复位信号，系统复位，可以看到 LED0 因系统复位熄灭一次，再亮。反之，当按下按键 WK_UP 后，1 秒内再按下按键 WK_UP，就会及时喂狗，结果就是系统不会复位，LED0 也就不会闪烁。

iwdg_init(IWDG_PRESCALER_64, 625);这个语句有必要跟大家说明一下，这里的第一个形参直接使用 HAL 库自定义的 IWDG_PRESCALER_64，即预分频系数为 64，重载值是 625，所以可由公式得到 $T_{out}=64 \times 625 / 40 = 1000ms$ ，即溢出时间就是 1s。只要你在 1 秒钟之内，有一次写 0XAAAA 到 IWDG_KR，就不会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 40Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

18.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，证明系统在不停的复位，否则 LED0 常亮。这时我们试试不停的按 KEY_UP 按键，可以看到 LED0 就常亮了，不会再闪烁。说明我们的实验设计成功了。

第十九章 窗口门狗（WWDG）实验

本章我们学习如何使用 STM32F1 的另外一个看门狗，窗口看门狗（以下简称 WWDG）。我们将使用窗口看门狗的中断功能来喂狗，通过 LED0 和 LED1 提示程序的运行状态。

本章分为如下几个小节：

19.1 WWDG 简介

19.2 硬件设计

19.3 程序设计

19.4 下载验证

19.1 WWDG 简介

窗口看门狗（WWDG）通常被用来监测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。窗口看门狗跟独立看门狗一样，也是一个递减计数器，不同的是它们的复位条件不一样。窗口看门狗产生复位信号有两个条件：

- 1) 当递减计数器的数值从 0x40 减到 0x3F 时（T6 位跳变到 0）。
- 2) 当喂狗的时候如果计数器的值大于 W[6:0] 时，此数值在 WWDG_CFR 寄存器定义。

上述的两个条件详细解释是，当计数器的值减到 0x40 时还不喂狗的话，到下一个计数就会产生复位，这个值称为窗口的下限值，是固定的值，不能改变。这个跟独立看门狗类似，不同的是窗口看门狗的计数器的值在减到某一个数之前喂狗的话也会产生复位，这个值叫窗口的上限，上限值 W[6:0] 由用户设置。窗口看门狗计数器的上限值和下限值就是窗口的含义，喂狗也必须在窗口之内，否则就会复位。

19.1.1 WWDG 框图

下面先来认识 WWDG 的逻辑结构，可以帮助我们快速认识窗口看门狗的工作原理，如图 19.1.1.1 所示。

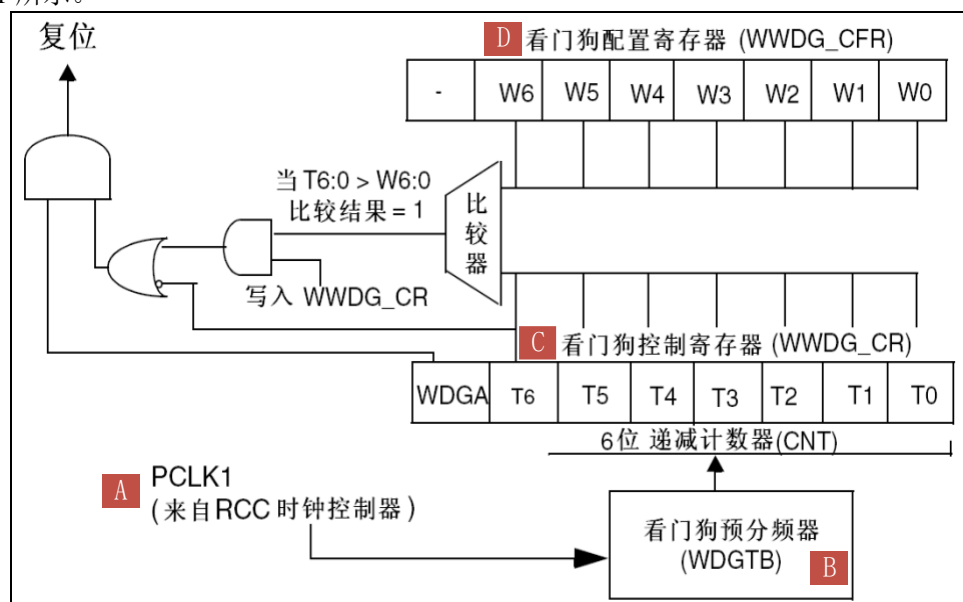


图 19.1.1.1 WWDG 逻辑框图

WWDG 有一个来自 RCC 的 PCLK1 输入时钟，经过一个 4096 的分频器(4096 分频在设计时已经设定死了，图中并没有给出来，但我们可以通过查看寄存器 WWDG_CFR 的 WDTB 位的描述知道)，再经过一个分频系数可选（1、2、4、8）的可编程预分频器提供时钟给一个 7 位递减计数器，这里有两个输出信号。

结合寄存器分析窗口看门狗的上限值和下限值。W[6:0] 是 WWDG_CFR 寄存器的低 7 位，用于与递减计数器 T[6:0] 比较的窗口值，也就是我们说的上限值，由用户设置。0x40 就是下限值，递减计数器达到这个值就会产生复位。T6 位就是 WWDG_CR 寄存器的位 6，即递减计数器 T[6:0] 的最高位。他们的关系可以用图 19.1.1.2 来说明：

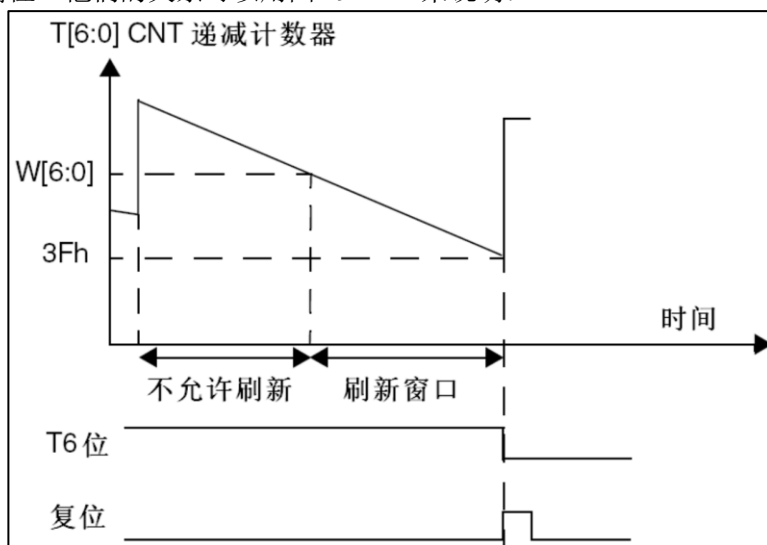


图 19.1.1.2 窗口看门狗工作示意图

图 19.1.1.2 可以看出，递减计数器的值递减过程中，当 $T[6:0] > W[6:0]$ 是不允许刷新 T[6:0] 的值，即不允许喂狗，否则会产生复位。只有在 $W[6:0] < T[6:0] < 0x3F$ 这个时间可以喂狗，这就是喂狗的窗口时间。当 $T[6:0] = 0x3F$ ，即 T6 位为 0 这一刻，也会产生复位。

上限值 W[6:0] 是由用户自己设置，但是一定要确保大于 0x40，否则就不存在上图的窗口了，下限值 0x40 是固定的，不可修改。

知道了窗口看门狗的工作原理，下面学习如何计算窗口看门狗的超时公式：

$$T_{WWDG} = (4096 \times 2^{WDGTB} \times (T[5:0] + 1)) / F_{PCLK1}$$

其中：

T_{WWDG} ：WWDG 超时时间(单位为 ms)

F_{PCLK1} ：APB1 的时钟频率（单位为 KHz）

2^{WDGTB} ：是 WWDG_CFR 寄存器设置的预分频系数值

T[5:0]：窗口看门狗的计数器低 6 位的值

根据以上公式，假设 $F_{PCLK1} = 36\text{Mhz}$ ，那么可以得到最小-最大超时时间表如下表所示：

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

表 19.1.1.2 36M 时钟下窗口看门狗的最小最大超时表

19.1.2 WWDG 寄存器

WWDG 只有 3 个寄存器，具体如下：

- 控制寄存器 (WWDG_CR)

窗口看门狗的控制寄存器描述如图 19.1.2.1 所示：

| | | | | | | | | | | | | | | | |
|------|--|----|----|----|----|----|----|------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | WDGA | T6 | T5 | T4 | T3 | T2 | T1 | T0 |
| | | | | | | | | rS | rW | rW | rW | rW | rW | rW | rW |
| 位7 | WDGA: 激活位 (Activation bit) 此位由软件置'1', 但仅能由硬件在复位后清'0'。当WDGA=1时, 看门狗可以产生复位。 0: 禁止看门狗 1: 启用看门狗 | | | | | | | | | | | | | | |
| 位6:0 | T[6:0]: 7位计数器(MSB至LSB) (7-bit counter) 这些位用来存储看门狗的计数器值。每 (4096×2^{WDGTB}) 个PCLK1周期减1。当计数器值从40h变为3Fh时(T6变成0), 产生看门狗复位。 | | | | | | | | | | | | | | |

图 19.1.2.1 WWDG_CR 寄存器

该寄存器只有低八位有效, 其中 T[6: 0]用来存储看门狗的计数器的值, 随时更新的, 每隔 $(4096 \times 2^{WDGTB[2:0]})$ PCLK 个周期减 1。当该计数器的值从 0x40 变为 0x3F 的时候, 将产生看门狗复位。

WDGA 位则是看门狗的激活位, 该位由软件置 1, 启动看门狗, 并且一定要注意的是该位一旦设置, 就只能在硬件复位后才能清零了。

● 配置寄存器 (WWDG_CFR)

配置寄存器描述如图 19.1.2.2 所示:

| | | | | | | | | | | | | | | | |
|------|--|----|----|----|----|----|----|-----|---------|---------|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | EWI | WDG TB1 | WDG TB0 | W6 | W5 | W4 | W3 | W2 |
| | | | | | | | | rS | rW | rW | rW | rW | rW | rW | rW |
| 位9 | EWI: 提前唤醒中断 (Early wakeup interrupt) 此位若置'1', 则当计数器值达到40h, 即产生中断。 此中断只能由硬件在复位后清除。 | | | | | | | | | | | | | | |
| 位8:7 | WDGTB[1:0]: 时基 (Timer base) 预分频器的时基可以设置如下: 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8 | | | | | | | | | | | | | | |
| 位6:0 | W[6:0]: 7位窗口值 (7-bit window value) 这些位包含了用来与递减计数器进行比较用的窗口值。 | | | | | | | | | | | | | | |

图 19.1.2.2 WWDG_CFR 寄存器

该寄存器中的 EWI 位是提前唤醒中断, 如果该位置 1, 当递减计数器等于 0x40 时产生提前唤醒中断, 我们就可以及时喂狗以避免 WWDG 复位。因此, 我们一般都会用该位来设置中断, 当窗口看门狗的计数器值减到 0X40 的时候, 如果该位设置, 并开启了中断, 则会产生中断, 我们可以在中断里面向 WWDG_CR 重新写入计数器的值, 来达到喂狗的目的。注意这里在进入中断后, 必须在不大于 1 个窗口看门狗计数周期的时间 (在 pclk1 频率为 36M 且 WDTB 为 0 的条件下, 该时间为 113us) 内重新写 WWDG_CR, 否则, 看门狗将产生复位!

● 状态寄存器 (WWDG_SR)

该寄存器用来记录当前是否有提前唤醒的标志。该寄存器仅有位 0 有效, 其他都是保留位。当计数器值达到 0x40 时, 此位由硬件置 1。它必须通过软件写 0 来清除。对此位写 1 无效。即使中断未被使能, 在计数器的值达到 0x40 的时候, 此位也会被置 1。

19.2 硬件设计

1. 例程功能

先点亮 LED0 延时 300ms 后，初始化窗口看门狗，进入死循环，关闭 LED0。然后等待窗口看门狗中断的到来，在中断里面，喂狗，并执行 LED1 的翻转操作。我们将通过 LED0 来指示 STM32F1 是否被复位了，如果被复位了就会点亮 300ms。LED1 用来指示中断喂狗，每次中断喂狗翻转一次。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 窗口看门狗

3. 原理图

窗口看门狗属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 LED0 和 LED1 来指示 STM32F103 的复位情况和窗口看门狗的喂狗情况。

19.3 程序设计

19.3.1 WWDG 的 HAL 库驱动

WWDG 在 HAL 库中的驱动代码在 stm32f1xx_hal_wwdg.c 文件（及其头文件）中。

1. HAL_WWDG_Init 函数

IWDG 的初始化函数，其声明如下：

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

● 函数描述：

用于初始化 WWDG。

● 函数形参：

形参 1 是 WWDG 句柄，WWDG_HandleTypeDef 结构体类型，其定义如下：

```
typedef struct
{
    WWDG_TypeDef          *Instance; /* WWDG 寄存器基地址 */
    WWDG_InitTypeDef      Init;      /* WWDG 初始化参数 */
}WWDG_HandleTypeDef;
```

1) Instance: 指向 WWDG 寄存器基地址。

2) Init: WWDG 初始化结构体，用于配置计数器的相关参数。

WWDG_InitTypeDef 这个结构体类型定义如下：

```
typedef struct
{
    uint32_t Prescaler; /* 预分频系数 */
    uint32_t Window;    /* 窗口值 */
    uint32_t Counter;    /* 计数器值 */
    uint32_t EWIMode;    /* 提前唤醒中断使能 */
} WWDG_InitTypeDef;
```

1) Prescaler: 预分频系数，WWDG_PRESCALER_1\WWDG_PRESCALER_2\

WWDG_PRESCALER_4\WWDG_PRESCALER_8 四个值，分别表示 1\2\4\8 分频。

2) Window: 窗口值，即上限值。

3) Counter: 计数器值，用于保存要设置计数器的值。

4) EWIMode: 提前唤醒中断使能。

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

2. HAL_WWDG_Refresh 函数

HAL_WWDG_Refresh 函数是窗口看门狗的喂狗函数。其声明如下：

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

- **函数描述：**
该函数实际就是往 CR 寄存器重写 Counte 这个预先保存的计数器值。
- **函数形参：**
形参 1 是 WWDG_HandleTypeDef 结构体指针类型的 WWDG 句柄。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。

窗口看门狗配置步骤

1) 使能 WWDG 时钟

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

```
HAL_RCC_WWDG_CLK_ENABLE();
```

2) 设置窗口值，分频数和计数器初始值

在 HAL 库中，这三个值都是通过函数 HAL_WWDG_Init 来设置的，详见本例程源码。

3) 开启 WWDG

通过设置 WWDG_CR 寄存器的 WDGA(bit7)位为 1 来实现开启窗口看门狗，同样是在 HAL_WWDG_Init 函数里面实现。

4) 使能中断通道并配置优先级（如果开启了 WWDG 中断）

WWDG 的中断也是通过：HAL_NVIC_EnableIRQ 函数使能，通过 HAL_NVIC_SetPriority 函数设置优先级。

HAL 库同样为看门狗提供了 MSP 回调函数 HAL_WWDG_MspInit，一般情况下，步骤 1 和步骤 4 的步骤，我们均放在该回调函数中。

5) 编写中断服务函数

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

窗口看门狗中断服务函数为：WWDG_IRQHandler，喂狗函数为：HAL_WWDG_Refresh。

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

HAL 库定义了一个 WWDG 中断处理共用函数 HAL_WWDG_IRQHandler，我们在 WWDG 中断服务函数中会调用该函数。同时该函数会调用回调函数 HAL_WWDG_EarlyWakeupCallback，提前唤醒中断逻辑（喂狗、闪灯）我们写在回调函数 HAL_WWDG_EarlyWakeupCallback 中。

19.3.2 程序流程图

本实验利用窗口看门狗的特性，配置一个合适的窗口时间，并开启了提前唤醒中断，如果程序未在合适的时间喂狗，则会触发窗口看门狗中断。我们在进入中断后需要第一时间喂狗，否则系统将会复位，并用 LED1 的翻转来表示一次未及时喂狗的事件。

下面的流程图表示了 main 函数必要的编程步骤。

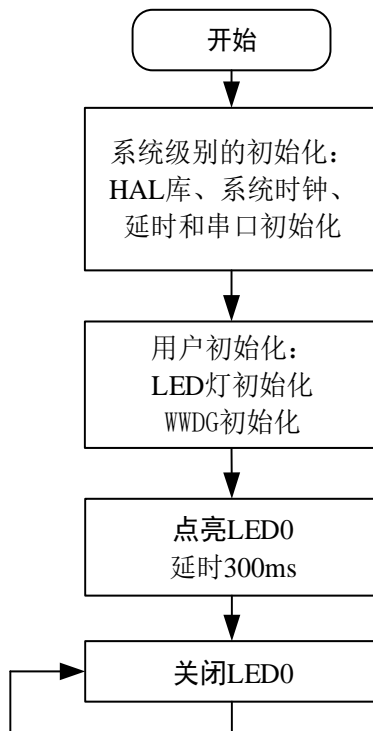


图 19.3.2.1 窗口看门狗实验程序流程图

19.3.3 程序解析

1. WWDG 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。窗口看门狗（WWDG）驱动源码包括两个文件：wdg.c 和 wdg.h。

wdg.h 头文件只有函数的声明，就不解释了。下面我们直接解析 wdg.c 的程序，先看 WWDG 的初始化函数，其定义如下：

```

/**
 * @brief      初始化窗口看门狗
 * @param      tr: T[6:0], 计数器值
 * @param      tw: W[6:0], 窗口值
 * @param      fprer: 分频系数 (WDGTB), 范围:WWDG_PRESCALER_1~WWDG_PRESCALER_8,
 *                  表示 2^WDGTB 分频, Fwwdg=PCLK1/(4096*2^fprer). 一般 PCLK1=36Mhz
 * @retval     无
 */
void wwdg_init(uint8_t tr, uint8_t wr, uint32_t fprer)
{
    wwdg_handler.Instance = WWDG;
    wwdg_handler.Init.Prescaler = fprer;          /* 设置分频系数 */
    wwdg_handler.Init.Window = wr;               /* 设置窗口值 */
    wwdg_handler.Init.Counter = tr;              /* 设置计数器值 */
    wwdg_handler.Init.EWIMode = WWDG_EWI_ENABLE; /* 使能窗口看门狗提前唤醒中断 */
    HAL_WWDG_Init(&wwdg_handler);               /* 初始化 WWDG */
}
    
```

WWDG_Init 是独立看门狗初始化函数，主要设置预分频数、窗口值和计数器的值，以及选择是否使能窗口看门狗提前唤醒中断。

因为用到中断，我们用 HAL_WWDG_MspInit 函数来编写窗口看门狗中断的初始化代码。当然大家也可以 HAL_WWDG_MspInit 函数的代码放到 wwdg_init 函数里面。这个初始化框架就是 HAL 库的特点。

```

void HAL_WWDG_MspInit(WWDG_HandleTypeDef *hwwdg)
{
    
```

```
__HAL_RCC_WWDG1_CLK_ENABLE(); /* 使能窗口看门狗时钟 */

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

HAL_WWDG_MspInit 函数会被 HAL_WWDG_Init 函数调用。该函数使能窗口看门狗的时钟，并设置窗口看门狗中断的抢占优先级为 2，响应优先级为 3。

```
/**
 * @brief      窗口看门狗中断服务程序
 * @param      无
 * @retval     无
 */
void WWDG_IRQHandler(void)
{
    HAL_WWDG_IRQHandler(&g_wwdg_handle);
}
```

WWDG_IRQHandler 函数是窗口看门狗中断服务函数，而这个函数实际上就是调用 HAL 库的中断处理函数 HAL_WWDG_IRQHandler。逻辑程序在下面的这个回调函数中：

```
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg)
{
    HAL_WWDG_Refresh(&g_wwdg_handle); /* 更新窗口看门狗值 */
    LED1_TOGGLE(); /* LED1 闪烁 */
}
```

在回调函数内部调用 HAL_WWDG_Refresh 函数喂狗，并翻转 LED1。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    LED0(0); /* 点亮 LED0 红灯 */
    delay_ms(300); /* 延时 300ms 再初始化看门狗, LED0 的变化"可见" */
    /* 计数器值为 7f, 窗口寄存器为 5f, 分频数为 8 */
    wwdg_init(0X7F, 0X5F, WWDG_PRESCALER_8);
    while (1)
    {
        LED0(1); /* 关闭红灯 */
    }
}
```

在 main 函数里，先初始化系统和用户的外设代码，然后先点亮 LED0，延时 300ms 后，初始化窗口看门狗，进入死循环，关闭 LED0。

调用 wwdg_init(0X7F,0X5F,WWDG_PRESCALER_8)这个语句，就设置计数器值为 7f，窗口寄存器为 5f，分频数为 8，然后可由前面的公式得到窗口上限时间 $T_{wwdg}=4096 \times 8 \times (0x7F-0x5F)/36MHz=29.12ms$ ，窗口下限时间 $T_{wwdg}=4096 \times 8 \times (0x7F-0x3F)/36MHz=58.25ms$ ，即喂狗的窗口区间为 29.12~58.25ms。我们在程序的其它地方没有喂狗，所以程序会在 58.25ms 左右进入死前中断，我们在中断中喂狗一次，并翻转 LED1。

19.4 下载验证

下载代码后，可以看到 LED0 亮了一下就熄灭，紧接着 LED1 开始不停的闪烁。可以接入示波器测试得每秒钟闪烁 17 次左右，说明程序在中断不停的喂狗，和我们预期的一致。

第二十章 基本定时器实验

STM32F103 有众多的定时器，其中包括 2 个基本定时器（TIM6 和 TIM7）、4 个通用定时器（TIM2~TIM5）、2 个高级控制定时器（TIM1 和 TIM8），这些定时器彼此完全独立，不共享任何资源。本章我们学习如何使用 STM32F103 的基本定时器中断。我们将使用 TIM6 的定时器中断来控制 LED1 的翻转，在主函数用 LED0 的翻转来提示程序正在运行。

本章分为如下几个小节：

- 20.1 基本定时器简介
- 20.2 硬件设计
- 20.3 程序设计
- 20.4 下载验证

20.1 基本定时器简介

STM32F103 有两个基本定时器 TIM6 和 TIM7，它们的功能完全相同，资源是完全独立的，可以同时使用。其主要特性如下：16 位自动重载递增计数器，16 位可编程预分频器，预分频系数 1~65536，用于对计数器时钟频率进行分频，还可以触发 DAC 的同步电路，以及生成中断/DMA 请求。

20.1.1 基本定时器框图

下面先来学习基本定时器框图，通过学习基本定时器框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

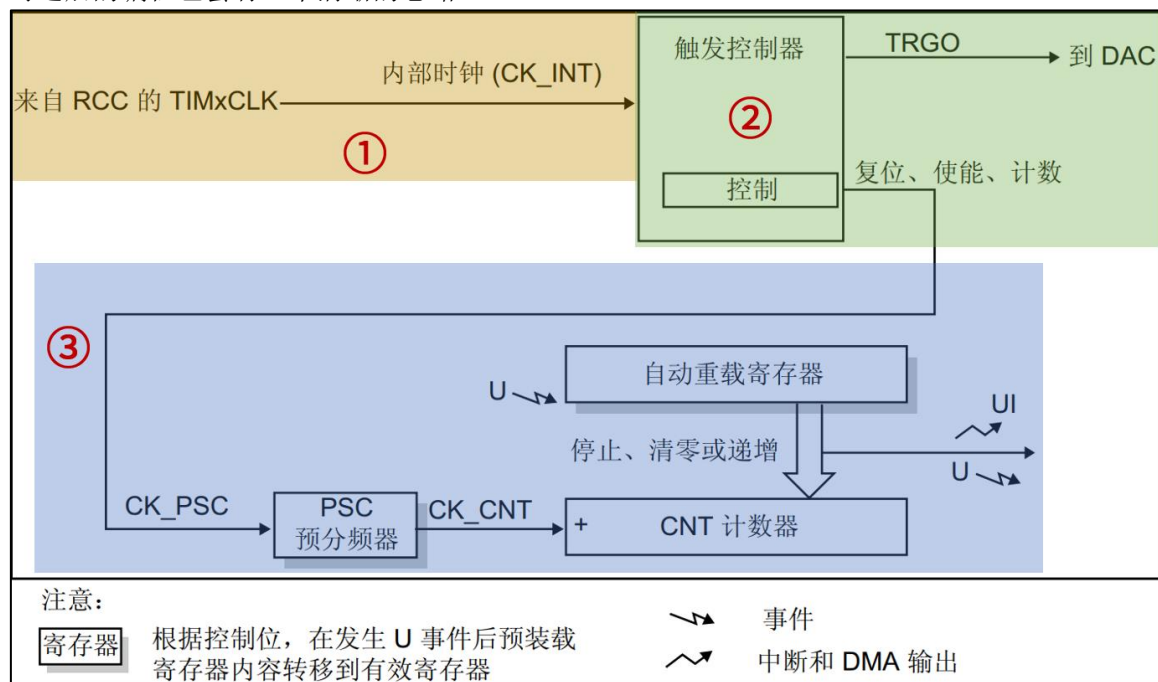


图 20.1.1.1 基本定时器框图

① 时钟源

定时器的核心就是计算器，要实现计数功能，首先要给它一个时钟源。基本定时器时钟挂载在 APB1 总线，所以它的时钟来自于 APB1 总线，但是基本定时器时钟不是直接由 APB1 总线直接提供，而是先经过一个倍频器。当 APB1 的预分频器系数为 1 时，这个倍频器系数为 1，即定时器的时钟频率等于 APB1 总线时钟频率；当 APB1 的预分频器系数 ≥ 2 分频时，这个倍频器系数就为 2，即定时器的时钟频率等于 APB1 总线时钟频率的两倍。我们在

sys_stm32_clock_init 时钟设置函数已经设置 APB1 总线时钟频率为 36M，APB1 总线的预分频器分频系数是 2，所以挂载在 APB1 总线的定时器时钟频率为 72Mhz。

② 控制器

控制器除了控制定时器复位、使能、计数等功能之外，还可以用于触发 DAC 转换。

③ 时基单元

时基单元包括：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)。基本定时器的这三个寄存器都是 16 位有效数字，即可设置值范围是 0~65535。

时基单元中的预分频器 PSC，它有一个输入和一个输出。输入 CK_PSC 来源于控制器部分，实际上就是来自于内部时钟(CK_INT)，即 2 倍的 APB1 总线时钟频率(72MHz)。输出 CK_CNT 是分频后的时钟，它是计数器实际的计数时钟，通过设置预分频器寄存器(TIMx_PSC)的值可以得到不同频率 CK_CNT，计算公式如下：

$$f_{CK_CNT} = f_{CK_PSC} / (PSC[15:0] + 1)$$

上式中，PSC[15:0]是写入预分频器寄存器(TIMx_PSC)的值。

另外：预分频器寄存器(TIMx_PSC)可以在运行过程中修改它的数值，新的预分频数值将在下一个更新事件时起作用。因为更新事件发生时，会把 TIMx_PSC 寄存器值更新到其影子寄存器中，这才会起作用。

什么是影子寄存器？从框图上看，可以看到图 20.1.1.1 中的预分频器 PSC 后面有一个影子，自动重载寄存器也有个影子，这就表示这些寄存器有影子寄存器。影子寄存器是一个实际起作用的寄存器，不可直接访问。举个例子：我们可以把预分频系数写入预分频器寄存器(TIMx_PSC)，但是预分频器寄存器只是起到缓存数据的作用，只有等到更新事件发生时，预分频器寄存器的值才会被自动写入其影子寄存器中，这时才真正起作用。

自动重载寄存器及其影子寄存器的作用和上述同理。不同点在于自动重载寄存器是否具有缓冲作用还受到 ARPE 位的控制，当该位置 0 时，ARR 寄存器不进行缓冲，我们写入新的 ARR 值时，该值会马上被写入 ARR 影子寄存器中，从而直接生效；当该位置 1 时，ARR 寄存器进行缓冲，我们写入新的 ARR 值时，该值不会马上被写入 ARR 影子寄存器中，而是要等到更新事件发生才会被写入 ARR 影子寄存器，这时才生效。预分频器寄存器则没有这样相关的控制位，这就是它们不同点。

值得注意的是，**更新事件的产生有两种情况**，一是由软件产生，将 TIMx_EGR 寄存器的位 UG 置 1，产生更新事件后，硬件会自动将 UG 位清零。二是由硬件产生，满足以下条件即可：计数器的值等于自动重载寄存器影子寄存器的值。下面来讨论一下硬件更新事件。

基本定时器的计数器(CNT)是一个递增的计数器，当寄存器(TIMx_CR1)的 CEN 位置 1，即使能定时器，每来一个 CK_CNT 脉冲，TIMx_CNT 的值就会递增加 1。当 TIMx_CNT 值与 TIMx_ARR 的设定值相等时，TIMx_CNT 的值就会被自动清零并且会生成**更新事件**（如果开启相应的功能，就会产生 DMA 请求、产生中断信号或者触发 DAC 同步电路），然后下一个 CK_CNT 脉冲到来，TIMx_CNT 的值就会递增加 1，如此循环。在此过程中，TIMx_CNT 等于 TIMx_ARR 时，我们称之为定时器溢出，因为是递增计数，故而又称为**定时器上溢**。定时器溢出就伴随着更新事件的发生。

由上述可知，我们只要设置预分频寄存器和自动重载寄存器的值就可以控制定时器更新事件发生的时间。自动重载寄存器(TIMx_ARR)是用于存放一个与计数器作比较的值，当计数器的值等于自动重载寄存器的值时就会生成更新事件，硬件自动置位相关更新事件的标志位，如：更新中断标志位。

下面举个例子来学习如何设置预分频寄存器和自动重载寄存器的值得到我们想要的定时器上溢事件发生的时间周期。比如我们需要一个 500ms 周期的定时器更新中断，一般思路是先设置预分频寄存器，然后才是自动重载寄存器。考虑到我们设置的 CK_INT 为 72MHz，我们把预分频系数设置为 7200，即写入预分频寄存器的值为 7199，那么 $f_{CK_CNT} = 72\text{MHz} / 7200 = 10\text{KHz}$ 。这样就得到计数器的计数频率为 10KHz，即计数器 1 秒钟可以计 10000 个数。我们需要 500ms 的中断周期，所以我们让计数器计数 5000 个数就能满足要求，即需要设置自动重载寄存器的值为 4999，另外还要把定时器更新中断使能位 UIE 置 1，CEN 位也要置 1。

20.1.2 TIM6/TIM7 寄存器

下面介绍 TIM6/TIM7 的几个重要的寄存器，具体如下：

● 控制寄存器 1 (TIMx_CR1)

TIM6/TIM7 的控制寄存器 1 描述如图 20.1.2.1 所示。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|-----|----|--|----|----|----|---|---|------|-----|---|---|---|-----|-----|------|-----|
| 保留 | | | | | | | | ARPE | 保留 | | | | OPM | URS | UDIS | CEN |
| res | | | | | | | | rw | res | | | | rw | rw | rw | rw |
| 位7 | | ARPE: 自动重载预装载使能 (Auto-reload preload enable) 0: TIMx_ARR寄存器没有缓冲 1: TIMx_ARR寄存器具有缓冲 | | | | | | | | | | | | | | |
| 位0 | | CEN: 计数器使能 (Counter enable) 0: 关闭计数器 1: 使能计数器 注: 门控模式只能在软件已经设置了CEN位时有效, 而触发模式可以自动地由硬件设置CEN位。 在单脉冲模式下, 当产生更新事件时CEN被自动清除。 | | | | | | | | | | | | | | |

图 20.1.2.1 TIMx_CR1 寄存器

该寄存器，我们需要注意的是：位 0 (CEN) 用于使能或者禁止计数器，该位置 1 计数器开始工作，置 0 则停止。还有位 7 (ARPE) 用于控制自动重载寄存器 ARR 是否具有缓冲作用，如果 ARPE 位置 1，ARR 起缓冲作用，即只有在更新事件发生时才会把 ARR 的值写入其影子寄存器里；如果 ARPE 位置 0，那么修改自动重载寄存器的值时，该值会马上被写入其影子寄存器中，从而立即生效。

● DMA/中断使能寄存器 (TIMx_DIER)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|--|----|----|----|---|-----|-----|---|---|---|---|---|---|-----|
| 保留 | | | | | | | UDE | 保留 | | | | | | | UIE |
| res | | | | | | | rw | res | | | | | | | rw |
| 位8 | | UDE: 更新DMA请求使能 (Update DMA request enable) 0: 禁止更新DMA请求 1: 使能更新DMA请求 | | | | | | | | | | | | | |
| 位0 | | UIE: 更新中断使能 (Update interrupt enable) 0: 禁止更新中断 1: 使能更新中断 | | | | | | | | | | | | | |

图 20.1.2.2 TIMx_DIER 寄存器

该寄存器位 0 (UIE) 用于使能或者禁止更新中断，因为本实验我们用到中断，所以该位需要置 1。位 8 (UDE) 用于使能或者禁止更新 DMA 请求，我们暂且用不到，置 0 即可。

● 状态寄存器 (TIMx_SR)

TIM6/TIM7 的状态寄存器描述如图 20.1.2.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|---|----|----|----|---|---|---|---|---|---|---|---|---|-------|
| 保留 | | | | | | | | | | | | | | | UIF |
| res | | | | | | | | | | | | | | | rc w0 |
| 位0 | | UIF：更新中断标志 (Update interrupt flag) 硬件在更新中断时设置该位，它由软件清除。 0： 没有产生更新。 1： 产生了更新中断。下述情况下由硬件设置该位： — 计数器产生上溢或下溢并且TIMx_CR1中的UDIS=0； — 如果TIMx_CR1中的URS=0并且UDIS=0，当使用TIMx_EGR寄存器的UG位重新初始化计数器CNT时。 | | | | | | | | | | | | | |

图 20.1.2.3 TIMx_SR 寄存器

该寄存器位 0 (UIF) 是中断更新的标志位，当发生中断时由硬件置 1，然后就会执行中断服务函数，需要软件去清零，所以我们必须在中断服务函数里把该位清零。如果中断到来后，

不把该位清零，那么系统就会一直进入中断服务函数，这显然不是我们想要的。

● 计数器寄存器 (TIMx_CNT)

TIM6/TIM7 的计数器寄存器描述如图 20.1.2.4 所示：

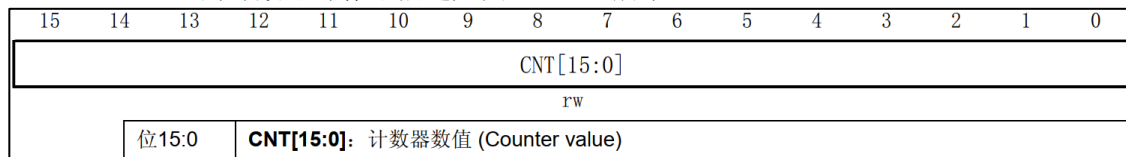


图 20.1.2.4 TIMx_CNT 寄存器

该寄存器位[15:0]就是计数器的实时的计数值。

● 预分频寄存器 (TIMx_PSC)

TIM6/TIM7 的预分频寄存器描述如图 20.1.2.5 所示：

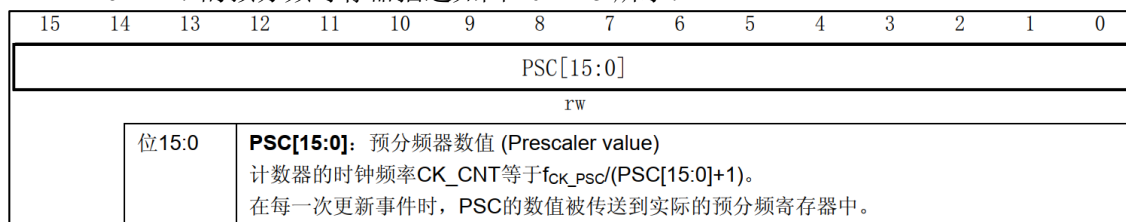


图 20.1.2.5 TIMx_PSC 寄存器

该寄存器是 TIM6/TIM7 的预分频寄存器，比如我们要 7200 分频，就往该寄存器写入 7199。注意这是 16 位的寄存器，写入的数值范围是 0 到 65535，分频系数范围：1 到 65536。

● 自动重载寄存器 (TIMx_ARR)

TIM6/TIM7 的自动重载寄存器描述如图 20.1.2.6 所示：

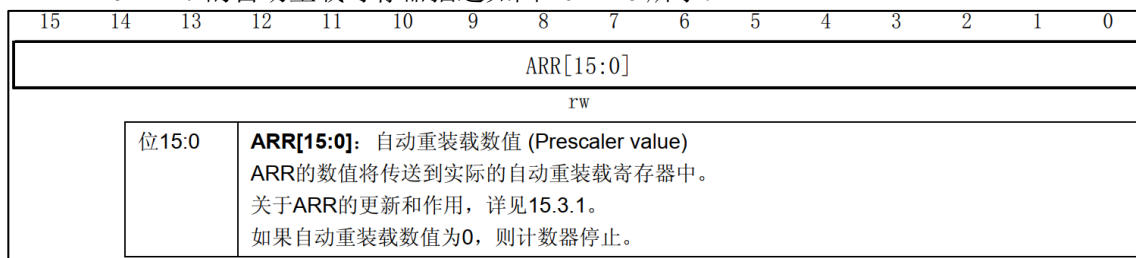


图 20.1.2.6 TIMx_ARR 寄存器

该寄存器可以由 APRE 位设置是否进行缓冲。计数器的值会和 ARR 寄存器影子寄存器进行比较，当两者相等，定时器就会溢出，从而发生更新事件，如果打开更新中断，还会发生更新中断。

20.1.3 基本定时器中断应用

本实验，我们主要配置定时器产生周期性溢出，从而在定时器更新中断中做周期性的操作，如周期性翻转 LED 灯。假设计数器计数模式为递增计数模式，那么实现周期性更新中断原理示意图如下所示：

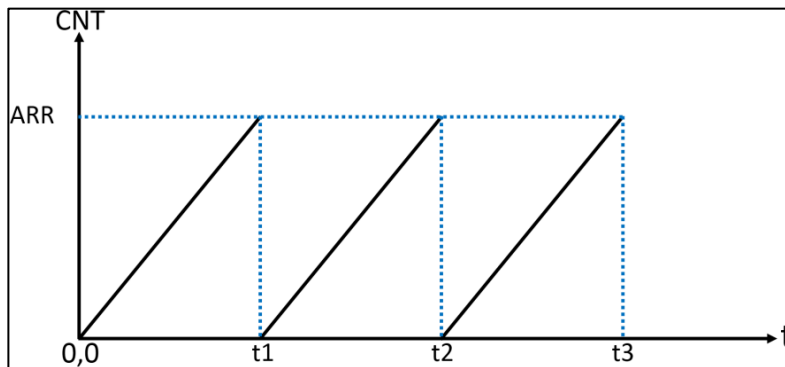


图 20.1.3.1 基本定时器中断示意图

如图 20.1.3.1 所示，CNT 计数器从 0 开始计数，当 CNT 的值和 ARR 相等时（t1），产生一个更新中断，然后 CNT 复位（清零），然后继续递增计数，依次循环。图中的 t1、t2、t3 就是定时器更新中断产生的时刻。

通过修改 ARR 的值，可以改变定时时间。另外，通过修改 PSC 的值，使用不同的计数频率（改变图中 CNT 的斜率），也可以改变定时时间。

20.2 硬件设计

1. 例程功能

LED0 用来指示程序运行，每 200ms 翻转一次。我们在更新中断中，将 LED1 的状态取反。LED1 用于指示定时器发生更新事件的频率，500ms 取反一次。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 定时器 6

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 LED1 来指示 STM32F103 的定时器进入中断的频率。

20.3 程序设计

20.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在 STM32F1xx_hal_tim.c 和 STM32F1xx_hal_tim_ex.c 文件（以及它们的头文件）中。

1. HAL_TIM_Base_Init 函数

定时器的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim);
```

● 函数描述：

用于初始化定时器。

● 函数形参：

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量（亦称定时器句柄），结构体定义如下：

```
typedef struct
{
    TIM_TypeDef                *Instance;      /* 外设寄存器基地址 */
    TIM_Base_InitTypeDef       Init;          /* 定时器初始化结构体 */
    HAL_TIM_ActiveChannel      Channel;        /* 定时器通道 */
    DMA_HandleTypeDef          *hdma[7];      /* DMA 管理结构体 */
    HAL_LockTypeDef            Lock;           /* 锁定资源 */
    __IO HAL_TIM_StateTypeDef  State;          /* 定时器状态 */
}TIM_HandleTypeDef;
```

1) Instance: 指向定时器寄存器基地址。

2) Init: 定时器初始化结构体，用于配置定时器的相关参数。

3) Channel: 定时器的通道选择，基本定时器没有该功能。

4) hdma[7]: 用于配置定时器的 DMA 请求。

5) Lock: ADC 锁资源。

6) State: 定时器工作状态。

我们主要看 TIM_Base_InitTypeDef 这个结构体类型定义：

```
typedef struct
```



```
{
    uint32_t Prescaler;          /* 预分频系数 */
    uint32_t CounterMode;        /* 计数模式 */
    uint32_t Period;             /* 自动重载值 ARR */
    uint32_t ClockDivision;      /* 时钟分频因子 */
    uint32_t RepetitionCounter;   /* 重复计数器 */
    uint32_t AutoReloadPreload;  /* 自动重载预装载使能 */
} TIM_Base_InitTypeDef;
```

- 1) Prescaler: 预分频系数, 即写入预分频寄存器的值, 范围 0 到 65535。
- 2) CounterMode: 计数器计数模式, 这里基本定时器只能向上计数。
- 3) Period: 自动重载值, 即写入自动重载寄存器的值, 范围 0 到 65535。
- 4) ClockDivision: 时钟分频因子, 也就是定时器时钟频率 CK_INT 与数字滤波器所使用的采样时钟之间的分频比, 基本定时器没有此功能。
- 5) RepetitionCounter: 设置重复计数器寄存器的值, 用在高级定时器中。
- 6) AutoReloadPreload: 自动重载预装载使能, 即控制寄存器 1 (TIMx_CR1) 的 ARPE 位。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIM_Base_Start_IT 函数

HAL_TIM_Base_Start_IT 函数是更新定时器中断和使能定时器的函数。其声明如下:

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim);
```

● 函数描述:

该函数调用了 __HAL_TIM_ENABLE_IT 和 __HAL_TIM_ENABLE 两个函数宏定义, 分别是更新定时器中断和使能定时器的宏定义。

● 函数形参:

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量, 即定时器句柄。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

● 注意事项:

下面分别列出单独使能/关闭定时器中断和使能/关闭定时器方法:

```
HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE); /* 使能句柄指定的定时器更新中断 */
HAL_TIM_DISABLE_IT(htim, TIM_IT_UPDATE); /* 关闭句柄指定的定时器更新中断 */
HAL_TIM_ENABLE(htim); /* 使能句柄 htim 指定的定时器 */
HAL_TIM_DISABLE(htim); /* 关闭句柄 htim 指定的定时器 */
```

定时器中断配置步骤

1) 开启定时器时钟

HAL 中定时器使能是通过宏定义标识符来实现对相关寄存器操作的, 方法如下:

```
__HAL_RCC_TIMx_CLK_ENABLE(); /* x=1~8 */
```

2) 初始化定时器参数, 设置自动重载值, 分频系数, 计数方式等

定时器的初始化参数是通过定时器初始化函数 HAL_TIM_Base_Init 实现的。

注意: 该函数会调用: HAL_TIM_Base_MspInit 函数, 我们可以通过后者存放定时器时钟和中断等初始化的代码。

3) 使能定时器更新中断, 开启定时器计数, 配置定时器中断优先级

通过 HAL_TIM_Base_Start_IT 函数使能定时器更新中断和开启定时器计数。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断, 通过 HAL_NVIC_SetPriority 函数设置中断优先级。

4) 编写中断服务函数

定时器中断服务函数为: TIMx_IRQHandler 等, 当发生中断的时候, 程序就会执行中断服务函数。HAL 库提供了一个定时器中断公共处理函数 HAL_TIM_IRQHandler, 该函数又会调用 HAL_TIM_PeriodElapsedCallback 等一些回调函数, 需要用户根据中断类型选择重定义对应的中断回调函数来处理中断程序。

20.3.2 程序流程图

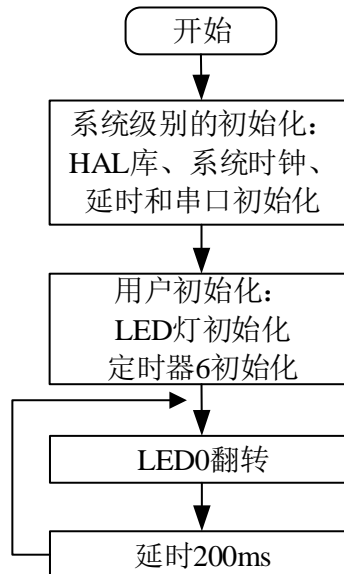


图 20.3.2.1 基本定时器中断实验程序流程图

程序开始先进行一系列初始化，然后在 main 中让 LED0 每过 200ms 翻转一次，用于指示系统代码正在运行。LED1 的翻转将在定时器更新中断里进行，请看程序解析。

20.3.3 程序解析

1. 基本定时器中断驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。基本定时器驱动代码包括两个文件：btim.c 和 btim.h。

首先看 btim.h 头文件的几个宏定义：

```

/* 基本定时器 定义 */
/* TIMX 中断定义
 * 默认是针对 TIM6/TIM7.
 * 注意：通过修改这 4 个宏定义,可以支持 TIM1~TIM8 任意一个定时器.
 */
#define BTIM_TIMX_INT          TIM6
#define BTIM_TIMX_INT_IRQn     TIM6_DAC_IRQn
#define BTIM_TIMX_INT_IRQHandler TIM6_DAC_IRQHandler
/* TIM6 时钟使能 */
#define BTIM_TIMX_INT_CLK_ENABLE() do{ __HAL_RCC_TIM6_CLK_ENABLE(); }while(0)

```

通过修改这 4 个宏定义，可以支持 TIM1~TIM8 任意一个定时器。

下面我们解析 btim.c 的程序，先看定时器的初始化函数，其定义如下：

```

/**
 * @brief      基本定时器 TIMX 定时中断初始化函数
 * @note
 *
 * 基本定时器的时钟来自 APB1,当 PPRE1 ≥ 2 分频的时候
 * 基本定时器的时钟为 APB1 时钟的 2 倍，而 APB1 为 36M，所以定时器时钟 = 72Mhz
 * 定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 * Ft=定时器工作频率,单位:Mhz
 *
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void btim_timx_int_init(uint16_t arr, uint16_t psc)
{

```

```
g_timx_handle.Instance = BTIM_TIMX_INT;          /* 定时器 x */
g_timx_handle.Init.Prescaler = psc;              /* 预分频 */
g_timx_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数器 */
g_timx_handle.Init.Period = arr;                 /* 自动装载值 */
HAL_TIM_Base_Init(&g_timx_handle);

HAL_TIM_Base_Start_IT(&g_timx_handle); /* 使能定时器 x 和定时器 x 更新中断 */
}
```

btim_timx_int_init 函数用来初始化定时器，我们可以通过修改宏定义 BTIM_TIMX_INT 来初始化 TIM1~TIM8 中的任意一个，本章我们是初始化基本定时器 6。该函数的 2 个形参：arr 设置自动重载寄存器(TIMx_ARR)，psc 设置预分频器寄存器(TIMx_PSC)。HAL_TIM_Base_Init 函数初始化定时器后，再调用 HAL_TIM_Base_Start_IT 函数使能定时器和更新定时器中断。

因为我们在 sys_stm32_clock_init 函数里面已经初始化 APB1 的时钟为 HCLK 的 2 分频，所以 APB1 的时钟为 36M，而从 STM32F1 的内部时钟树图得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 的时钟频率将为 APB1 时钟的两倍。因此，TIM6 的时钟为 72M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

$$T_{out} = ((arr+1) * (psc+1)) / T_{clk}$$

其中：

Tout：定时器溢出时间（单位为 s）。

Tclk：定时器的时钟源频率（单位为 MHz）。

arr：自动重装寄存器（TIMx_ARR）的值。

psc：预分频器寄存器（TIMx_PSC）的值

定时器底层驱动初始化函数如下：

```
/**
 * @brief      定时器底层驱动，开启时钟，设置中断优先级
 *             此函数会被 HAL_TIM_Base_Init() 函数调用
 * @param      无
 * @retval     无
 */
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == BTIM_TIMX_INT)
    {
        BTIM_TIMX_INT_CLK_ENABLE();          /* 使能 TIMx 时钟 */
        /* 设置中断优先级，抢占优先级 1，子优先级 3 */
        HAL_NVIC_SetPriority(BTIM_TIMX_INT_IRQn, 1, 3);
        HAL_NVIC_EnableIRQ(BTIM_TIMX_INT_IRQn); /* 开启 TIMx 中断 */
    }
}
```

HAL_TIM_Base_MspInit 函数用于存放 GPIO、NVIC 和时钟相关的代码，这里首先判断定时器的寄存器基地址，满足条件后，首先设置使能定时器的时钟，然后设置定时器中断的抢占优先级为 1，响应优先级为 3，最后开启定时器中断。这里没有用到 IO 引脚，所以不用初始化 GPIO。

接着是定时器中断服务函数的定义，这里用的是宏名，其定义如下：

```
/**
 * @brief      定时器中断服务函数
 * @param      无
 * @retval     无
 */
void BTIM_TIMX_INT_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&timx_handle);
}
```

这个函数实际上调用 HAL 库的定时器中断公共处理函数 HAL_TIM_IRQHandler。HAL 库的中断公共处理函数，会根据中断标志位调用各个中断回调函数，中断要处理的逻辑代码就写到这个回调函数中。比如这里我们使用到的是更新中断，定义的更新中断回调函数如下：

```
/**
 * @brief      定时器更新中断回调函数
 * @param      htim:定时器句柄指针
 * @retval     无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == BTIM_TIMX_INT)
    {
        LED1_TOGGLE();
    }
}
```

更新中断回调函数是所有定时器公用的，所以我们就需要在更新中断回调函数中对定时器寄存器基地址进行判断，只有符合对应定时器发生的更新中断，才能进行相应的处理，从而避免多个定时器同时使用到更新中断，导致更新中断代码的逻辑混乱。这里我们使用定时器 6 的更新中断，所以进入更新中断回调函数后，先判断是不是定时器 6 的寄存器基地址，当然这里使用宏的形式，BTIM_TIMX_INT 的原型就是 TIM6，执行的逻辑代码是翻转 LED1。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    btim_timx_int_init(5000 - 1, 7200 - 1); /* 10Khz 的计数频率, 计数 5K 次为 500ms */
    while (1)
    {
        LED0_TOGGLE();
        delay_ms(200);
    }
}
```

在 main 函数里，先初始化系统和用户的外设代码，然后在 while(1)里每 200ms 翻转一次 LED0。由前面的内容知道，定时器 6 的时钟频率为 72MHZ，而调用 btim_timx_int_init 初始化函数之后，就相当于写入预分频寄存器的值为 7199，写入自动重载寄存器的值为 4999。由公式得：

$$T_{out} = ((4999+1)*(7199+1))/72000000 = 0.5s = 500ms$$

20.4 下载验证

下载代码后，可以看到 LED0 不停闪烁（每 400ms 一个周期），而 LED1 也是不停的闪烁，但是闪烁时间较 LED0 慢（每 1s 一个周期）。

第二十一章 通用定时器实验

本章我们主要来学习通用定时器，通过上一章的学习，我们知道 STM32F103 有 4 个通用定时器（TIM2~TIM5）。这些定时器彼此完全独立，不共享任何资源。本章我们将通过四个实验来学习通用定时器的各个功能，分别是通用定时器中断实验、通用定时器 PWM 输出实验、通用定时器输入捕获实验和通用定时器脉冲计数实验。

本章分为如下几个小节：

- 21.1 通用定时器简介
- 21.2 通用定时器中断实验
- 21.3 通用定时器 PWM 输出实验
- 21.4 通用定时器输入捕获实验
- 21.5 通用定时器脉冲计数实验

21.1 通用定时器简介

STM32F103 的通用定时器有 4 个，为了更好的区别各个定时器的特性，我们列了一个表格，如下所示：

| 定时器类型 | 定时器 | 计数器位数 | 计数模式 | 预分频系数（整数） | 产生 DMA 请求 | 捕获/比较通道 | 互补输出 |
|-------|------------------------------|-------|------------|-----------|-----------|---------|------|
| 基本定时器 | TIM6 TIM7 | 16 | 递增 | 1~65536 | 可以 | 0 | 无 |
| 通用定时器 | TIM2 TIM3 TIM4 TIM5 | 16 | 递增、递减、中央对齐 | 1~65536 | 可以 | 4 | 无 |
| 高级定时器 | TIM1 TIM8 | 16 | 递增、递减、中央对齐 | 1~65536 | 可以 | 4 | 有 |

表 21.1.1 定时器基本特性表

注：该表参考数据手册《STM32F103ZET6.pdf》的 2.3.17 小节（第 19 页）。

由上表知道：该 STM32 芯片的计数器都是 16 位的。通用定时器和高级定时器其实也就是在基本定时器的基础上，添加了一些其他功能，如：输入捕获、输出比较、输出 PWM 和单脉冲模式等。而通用定时器数量较多，其特性也有一些的差异，但是基本原理都一样。

通用定时器框图

下面先来学习通用定时器框图，通过学习通用定时器框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

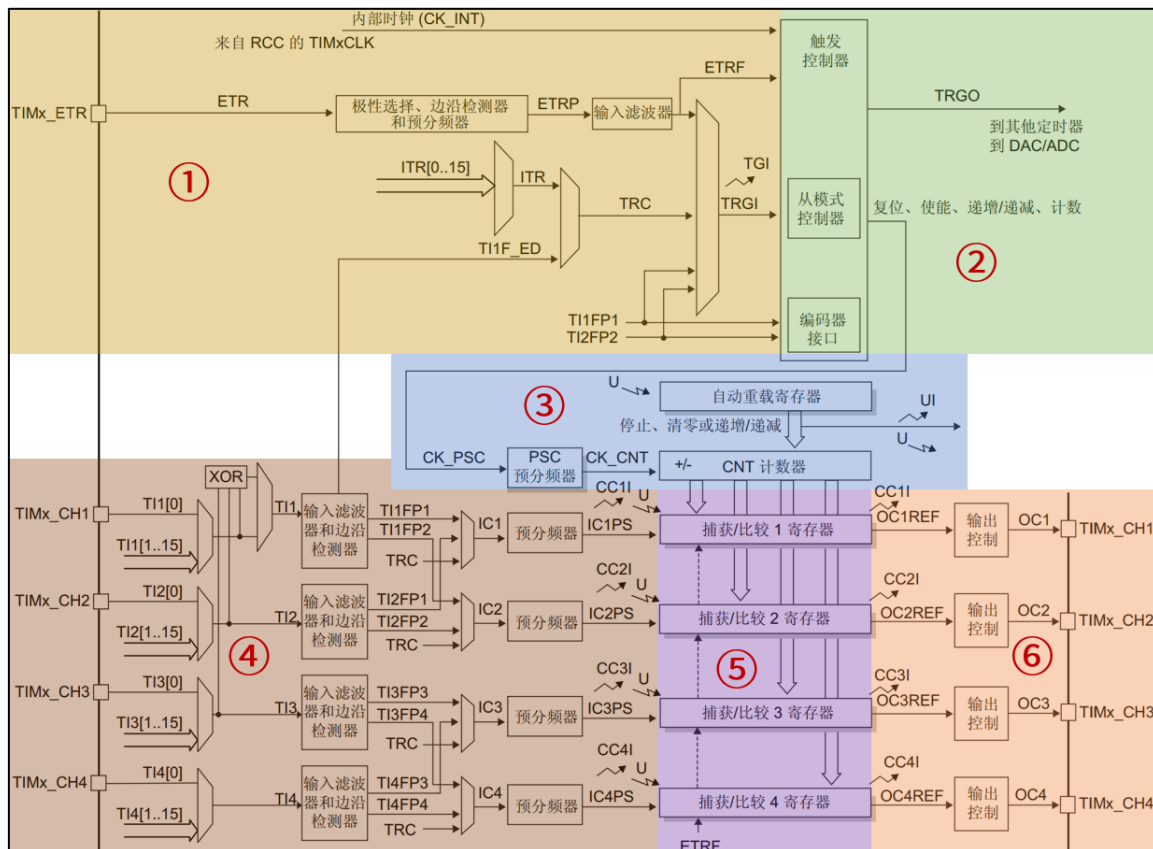


图 21.1.1 通用定时器框图

如上图，通用定时器的框图比基本定时器的框图复杂许多，为了方便介绍，我们将其分成六个部分讲解：

① 时钟源

通用定时器时钟可以选择下面四类时钟源之一：

- 1) 内部时钟(CK_INT)
- 2) 外部时钟模式 1：外部输入引脚(TIMx)，x=1, 2（即只能来自于通道 1 或者通道 2）
- 3) 外部时钟模式 2：外部触发输入(ETR)
- 4) 内部触发输入(ITRx)：使用一个定时器作为另一定时器的预分频器

通用定时器时钟源的设置方法如下表所示：

| 定时器时钟类型 | 设置方法 |
|-----------------------|--|
| 内部时钟(CK_INT) | 设置 TIMx_SMCR 的 SMS=0000 |
| 外部时钟模式 1：外部输入引脚(TIMx) | 设置 TIMx_SMCR 的 SMS=1111 |
| 外部时钟模式 2：外部触发输入(ETR) | 设置 TIMx_SMCR 的 ECE=1 |
| 内部触发输入(ITRx) | 设置可参考《STM32F10xxx 参考手册_V10(中文版).pdf》14.3.15 小节 |

表 21.1.2 通用定时器时钟源设置方法

内部时钟 (CK_INT)

STM32F1 系列的定时器 TIM2/TIM3/TIM4/TIM5/TIM6/TIM7 都是挂载在 APB1 总线上，这些定时器的内部时钟(CK_INT)实际上来自于 APB1 总线提供的时钟。但是这些定时器时钟不是由 APB1 总线直接提供，而是要先经过一个倍频器。在 HAL 库版本例程源码的 sys.c 文件中，系统时钟初始化函数 sys_stm32_clock_init 已经设置 APB1 总线时钟频率为 36MHz，APB1 预分频器的预分频系数为 2，所以这些定时器时钟源频率为 72MHz。因为当 APB1 预分频器的预分频系数 ≥ 2 分频时，挂载在 APB1 总线上的定时器时钟频率是该总线时钟频率的两倍。这个和基本定时器一样，可回顾基本定时器这部分内容。

另外，高级定时器 TIM1 和 TIM8 是挂载在 APB2 总线上的，也随便给大家说一下它们的情况。由图 11.1.1 STM32F1 时钟系统图可以知道，如果 APB2 预分频系数为 1，挂载在该总线的定时器时钟频率不变，否则频率是该总线时钟频率的 2 倍。我们在系统时钟初始化函数 sys_stm32_clock_init 已经设置 APB2 总线时钟频率为 72MHz，APB2 预分频器的预分频系数为 1，所以 TIM1 和 TIM8 时钟源频率为 72MHz。

外部时钟模式 1 (TI1、TI2)

外部时钟模式 1 这类时钟源，顾名思义时钟信号来自芯片外部。时钟源进入定时器的流程如下：外部时钟源信号→IO→TIMx_CH1（或者 TIMx_CH2），这里需要注意的是：外部时钟模式 1 下，时钟源信号只能从 CH1 或者 CH2 输入到定时器，CH3 和 CH4 都是不可以的。从 IO 到 TIMx_CH1（或者 TIMx_CH2），就需要我们配置 IO 的复用功能，才能使 IO 和定时器通道相连通。

时钟源信号来到定时器 CH1 或 CH2 后，需要经过什么“关卡”才能到达计数器作为计数的时钟频率的，下面通过外部时钟模式 1 框图给大家解答。

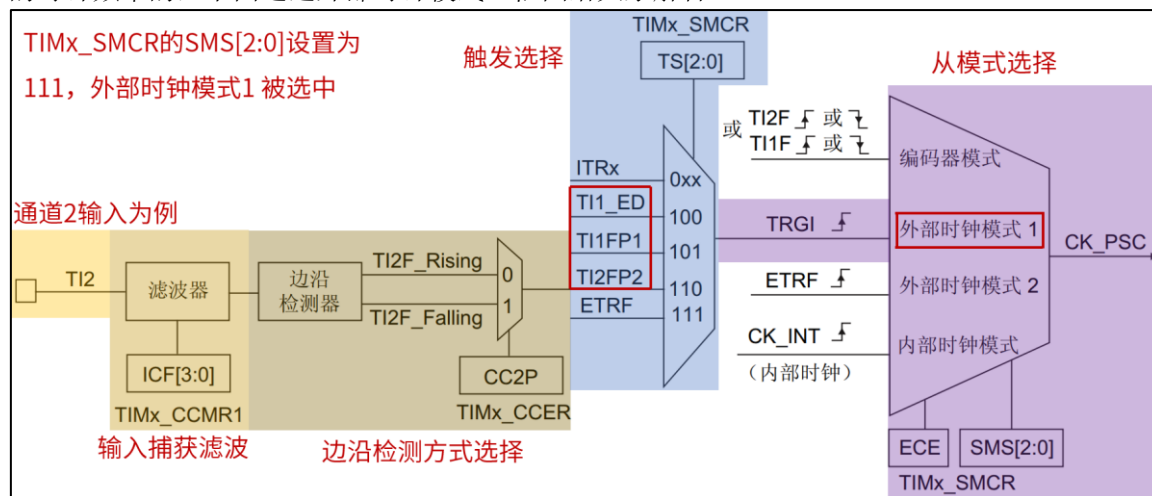


图 21.1.2 外部时钟模式 1 框图

图 21.1.2 中是以 CH2（通道 2）为例的，时钟源信号到达 CH2 后，那么这里我们把这个时钟源信号用 TI2 表示，因为它只是个信号，来到定时器内部，那我们就按定时器内部的信号来命名，所谓入乡随俗。

TI2 首先经过一个滤波器，由 ICF[3:0] 位来设置滤波方式，也可以设置不使用滤波器。

接着经过边沿检测器，由 CC2P 位来设置检测的边沿，可以上升沿或者下降沿检测。

然后经过触发输入选择器，由 TS[4:0] 位来选择 TRGI（触发输入信号）的来源。可以看到图 21.1.2 中框出了 TI1F_ED、TI1FP1 和 TI2FP2 三个触发输入信号（TRGI）。TI1F_ED 表示来自于 CH1，并且没有经过边沿检测器过滤的信号，所以它是 CH1 的双边沿信号，即上升沿或者下降沿都是有效的。TI1FP1 表示来自 CH1 并经过边沿检测器后的信号，可以是上升沿或者下降沿。TI2FP2 表示来自 CH2 并经过边沿检测器后的信号，可以是上升沿或者下降沿。这里以 CH2 为例，那只能选择 TI2FP2。如果是 CH1 为例，那就可以选择 TI1F_ED 或者 TI1FP1。

最后经过从模式选择器，由 ECE 位和 SMS[2:0] 位来选择定时器的时钟源。这里我们介绍的是外部时钟模式 1，所以 ECE 位置 0，SMS[2:0]=111 即可。CK_PSC 需要经过定时器的预分频器分频后，最终就能到达计数器进行计数了。

外部时钟模式 2 (ETR)

外部时钟模式 2，顾名思义时钟信号来自芯片外部。时钟源进入定时器的流程如下：外部时钟源信号→IO→TIMx_ETR。从 IO 到 TIMx_ETR，就需要我们配置 IO 的复用功能，才能使 IO 和定时器相连通。

时钟源信号来到定时器 TIMx_ETR 后，需要经过什么“关卡”才能到达计数器作为计数的时钟频率的，下面通过外部时钟模式 2 框图给大家解答。

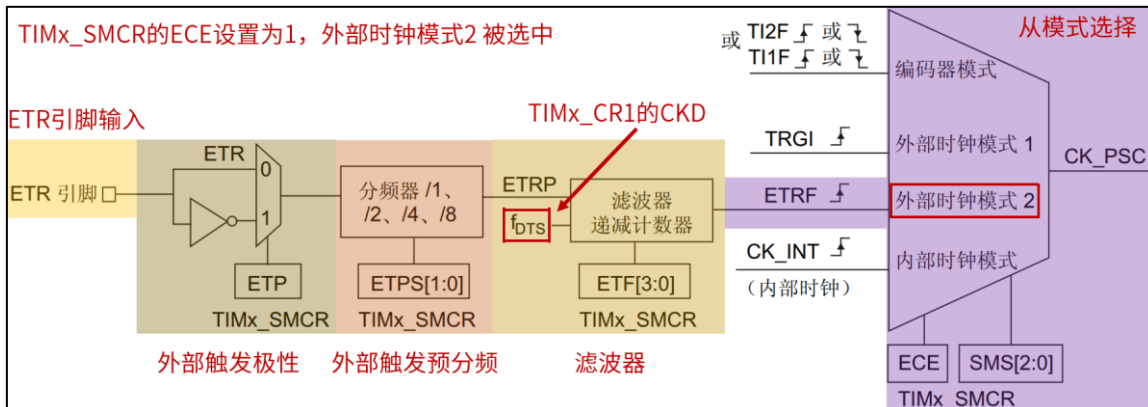


图 21.1.3 外部时钟模式 2 框图

图 21.1.3 中，可以看到在外部时钟模式 2 下，定时器时钟信号首先从 ETR 引脚进来。

接着经过外部触发极性选择器，由 ETP 位来设置上升沿有效还是下降沿有效，选择下降沿有效的话，信号会经过反相器。

然后经过外部触发预分频器，由 ETPS[1:0] 位来设置预分频系数，系数范围：1、2、4、8。

紧接着经过滤波器器，由 ETF[3:0] 位来设置滤波方式，也可以设置不使用滤波器。fDTS 由 TIMx_CR1 寄存器的 CKD 位设置。

最后经过从模式选择器，由 ECE 位和 SMS[2:0] 位来选择定时器的时钟源。这里我们介绍的是外部时钟模式 2，直接把 ECE 位置 1 即可。CK_PSC 需要经过定时器的预分频器分频后，最终就能到达计数器进行计数了。

内部触发输入 (ITRx)

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，即实现定时器的级联。下面以 TIM1 作为 TIM2 的预分频器为例，给大家介绍。

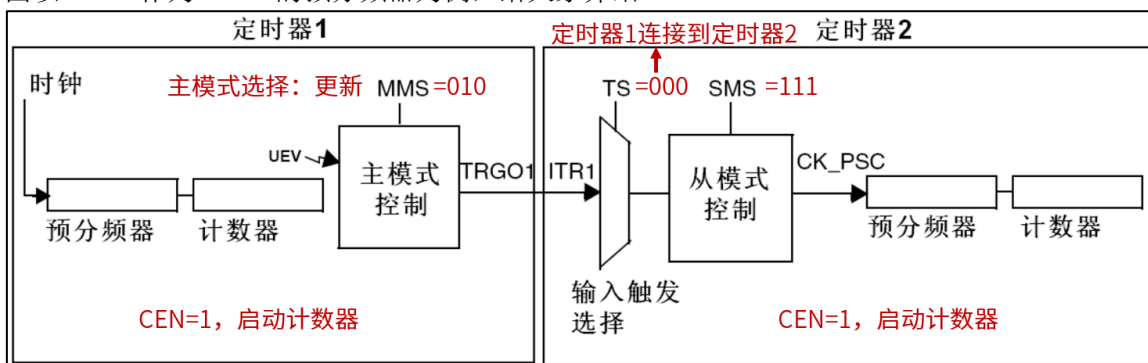


图 21.1.4 TIM1 作为 TIM2 的预分频器框图

上图中，TIM1 作为 TIM2 的预分频器，需要完成的配置步骤如下：

1, TIM1_CR2 寄存器的 MMS[2:0] 位设置为 010，即 TIM1 的主模式选择为更新（选择更新事件作为触发输出（TRGO））。

2, TIM2_SMCR 寄存器的 TS[2:0] 位设置为 000，即使用 ITR1 作为内部触发。TS[2:0] 位用于配置触发选择，除了 ITR1，还有其他的选择，详细描述如下图所示：

| | | |
|------|--|-------------------------|
| 位6:4 | TS[2:0]: 触发选择 (Trigger selection) 这3位选择用于同步计数器的触发输入。 | |
| | 000: 内部触发0(ITR0), TIM1 | 100: TI1的边沿检测器(TI1F_ED) |
| | 001: 内部触发1(ITR1), TIM2 | 101: 滤波后的定时器输入1(TI1FP1) |
| | 010: 内部触发2(ITR2), TIM3 | 110: 滤波后的定时器输入2(TI2FP2) |
| | 011: 内部触发3(ITR3), TIM4 | 111: 外部触发输入(ETRF) |
| | 关于每个定时器中ITRx的细节，参见表78。 | |
| | 注：这些位只能在未用到(如SMS=000)时被改变，以避免在改变时产生错误的边沿检测。 | |

图 21.1.5 触发选择

上图中的触发选择中,我们在讲解外部时钟模式 1 的时候说过 TI1F_ED、TI1FP1 和 TI2FP2,以及外部时钟模式 2 讲的 ETRF,它们都是属于外部的,其余的都是内部触发了。那么这内部触发都代表什么意思呢?大家打开《STM32F10xxx 参考手册_V10(中文版).pdf》的 285 页,就可以找下面这个表。

| 从定时器 | ITR0 (TS = 000) | ITR1 (TS = 001) | ITR2 (TS = 010) | ITR3 (TS = 011) |
|-------------|-----------------|-----------------|-----------------|-----------------|
| TIM2 | TIM1 | TIM8 | TIM3 | TIM4 |
| TIM3 | TIM1 | TIM2 | TIM5 | TIM4 |
| TIM4 | TIM1 | TIM2 | TIM3 | TIM8 |
| TIM5 | TIM2 | TIM3 | TIM4 | TIM8 |

表 21.1.3 TIMx 内部触发连接

在步骤 2 中, TS[2:0]位设置为 000, 使用 ITR1 作为内部触发, 这个 ITR1 什么意思? 由表 21.1.3 可以知道, 当从模式定时器为 TIM2 时, ITR1 表示主模式定时器就是 TIM1。这里只是 TIM2~5 的内部触发连接情况, 其他定时器请查看参考手册的相应章节。

3, TIM2_SMCR 寄存器的 SMS[2:0]位设置为 111, 即从模式控制器选择外部时钟模式 1。

4, TIM1 和 TIM2 的 CEN 位都要置 1, 即启动计数器。

定时器的时钟源这部分内容是非常重要的, 因为这计数器工作的基础。虽然定时器有四类时钟源之多, 但是我们最常用的还是内部时钟。

② 控制器

控制器包括: 从模式控制器、编码器接口和触发控制器 (TRGO)。从模式控制器可以控制计数器复位、启动、递增/递减、计数。编码器接口针对编码器计数。触发控制器用来提供触发信号给别的外设, 比如为其它定时器提供时钟或者为 DAC/ADC 的触发转换提供信号。

③ 时基单元

时基单元包括: 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)。这部分内容和基本定时器基本一样的, 大家可以参考基本定时器的介绍。

不同点是: 通用定时器的计数模式有三种: **递增计数模式**、**递减计数模式**和**中心对齐模式**; TIM2 和 TIM5 的计数器是 32 位的。递增计数模式在讲解基本定时器的时候已经讲过了, 那么对应到递减计数模式就很好理解了。就是来了一个计数脉冲, 计数器就减 1, 直到计数器寄存器的值减到 0, 减到 0 时定时器溢出, 由于是递减计数, 故而称为定时器下溢, 定时器溢出就会伴随着更新事件的发生。然后计数器又从自动重载寄存器影子寄存器的值开始继续递减计数, 如此循环。最后是中心对齐模式, 字面上不太好理解。该模式下, 计数器先从 0 开始递增计数, 直到计数器的值等于自动重载寄存器影子寄存器的值减 1 时, 定时器上溢, 同时生成更新事件, 然后从自动重载寄存器影子寄存器的值开始递减计算, 直到计数值等于 1 时, 定时器下溢, 同时生成更新事件, 然后又从 0 开始递增计数, 依此循环。每次定时器上溢或下溢都会生成更新事件。计数器的计数模式的设置请参考 TIMx_CR1 寄存器的位 CMS 和位 DIR。

下面通过一张图给大家展示定时器工作在不同计数模式下, 更新事件发生的情况。

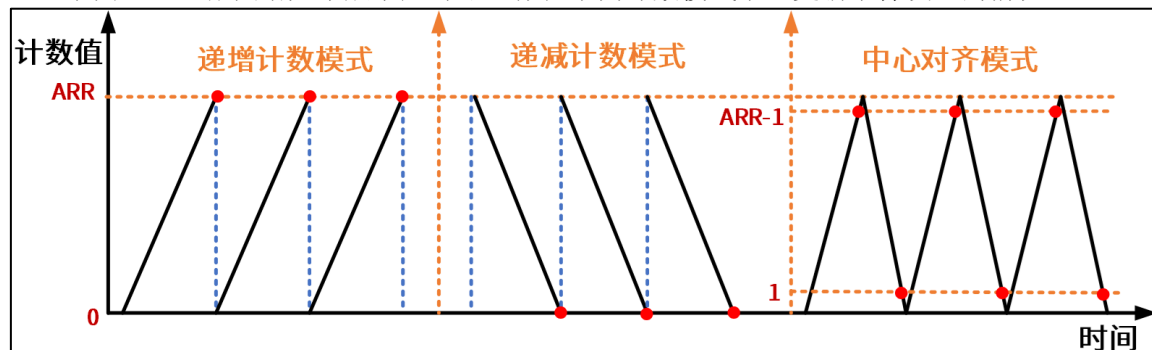


图 21.1.6 更新事件发生条件

上图中, 纵轴表示计数器的计数值, 横轴表示时间, ARR 表示自动重载寄存器的值, 小红点就是更新事件发生的时间点。举个例子, 递增计数模式下, 当计数值等于 ARR 时, 计数器的

值被复位为 0，定时器溢出，并伴随着更新事件的发生，后面继续递增计数。递减计数模式和中心对齐模式请参考前面的描述。

上表的描述属于硬件更新事件发生条件，我们还可以通过 UG 位产生软件更新事件。

关于影子寄存器和定时器溢出时间计算公式等内容可以参考基本定时器的相关内容。

④ 输入捕获

图 21.1.1 中的第④部分是输入捕获，一般应用是要和第⑤部分一起完成测量功能。TIMx_CH1~TIMx_CH4 表示定时器的 4 个通道，这 4 个通道都是可以独立工作的。IO 端口通过复用功能与这些通道相连。配置好 IO 端口的复用功能后，将需要测量的信号输入到相应的 IO 端口，输入捕获部分可以对输入的信号的上升沿，下降沿或者双边沿进行捕获，常见的测量有：测量输入信号的脉冲宽度、测量 PWM 输入信号的频率和占空比等。后续有相应的实验。

下面简单说一下测量高电平脉冲宽度的工作原理，方便大家的理解：一般先要设置输入捕获的边沿检测极性，如：我们设置上升沿检测，那么当检测到上升沿时，定时器会把计数器 CNT 的值锁存到相应的捕获/比较寄存器 TIMx_CCRy 里，y=1~4。然后我们再设置边沿检测为下降沿检测，当检测到下降沿时，定时器会把计数器 CNT 的值再次锁存到相应的捕获/比较寄存器 TIMx_CCRy 里。最后，我们将前后两次锁存的 CNT 的值相减，就可以算出高电平脉冲期间内计数器的计数个数，再根据定时器的计数频率就可以计算出这个高电平脉冲的时间。如果要测量的高电平脉宽时间长度超过定时器的溢出时间周期，就会发生溢出，这时候我们还需要做定时器溢出的额外处理。低电平脉冲捕获同理。

上面的描述是第④部分输入捕获整体上的一个应用情况，下面我们来看第④部分的细节。当需要测量的信号进入通道后，需要经过哪些“关卡”？我们用图 21.1.7 给大家讲解。

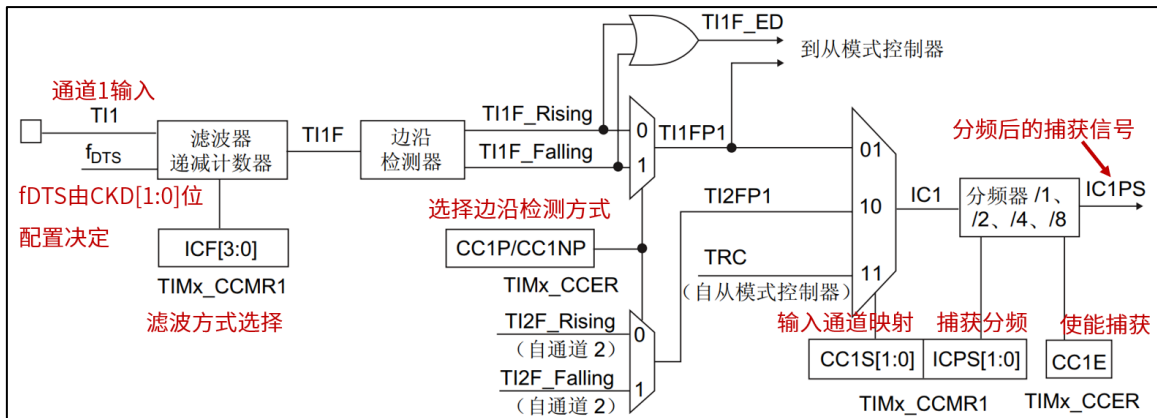


图 21.1.7 通道 1 输入阶段

图 21.1.7 是图 21.1.1 第④部分通道 1 的“放大版”，这里是以通道 1 输入捕获为例进行介绍，其他通道同理。

待测量信号到达 TIMx_CH1 后，那么这里我们把这个待测量信号用 TI1 表示，原因在讲解外部时钟模式 1 的时候说过，所谓“入乡随俗”。

TI1 首先经过一个滤波器，由 ICF[3:0] 位来设置滤波方式，也可以设置不使用滤波器。fDTS 由 TIMx_CR1 寄存器的 CKD 位设置。

接着经过边沿检测器，由 CC1P 位来设置检测的边沿，可以上升沿或者下降沿检测。CC1NP 是配置互补通道的边沿检测的，在高级定时器才有，通用定时器没有。

然后经过输入捕获映射选择器，由 CC1S[1:0] 位来选择把 IC1 映射到 TI1、TI2 还是 TRC。这里我们的待测量信号从通道 1 进来，所以选择 IC1 映射到 TI1 上即可。

紧接着经过输入捕获 1 预分频器，由 ICPS[1:0] 位来设置预分频系数，范围：1、2、4、8。

最后需要把 CC1E 位置 1，使能输入捕获，IC1PS 就是分频后的捕获信号。这个信号将会到达图 21.1.1 的第⑤部分。

下面我们接着看图 21.1.1 的第⑤部分的“放大版”，如下图所示：

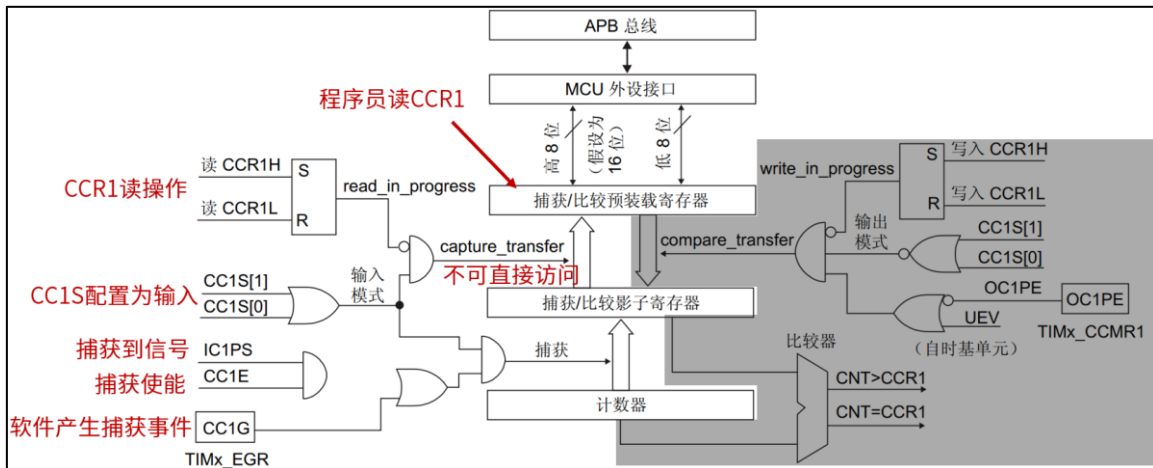


图 21.1.8 捕获/比较通道 1 主电路 (输入捕获功能部分)

图 21.1.8 中，灰色阴影部分是输出比较功能部分，讲到第⑥部分输出比较的时候再介绍。左边没有阴影部分就是输入捕获功能部分了。

首先看到捕获/比较预装载寄存器,我们以通道 1 为例,那么它就是 CCR1 寄存器,通道 2、通道 3、通道 4 就分别对应 CCR2、CCR3、CCR4。在图 21.1.1 中就可以看到 CCR1~4 是有影子寄存器的,所以这里就可以看到图 21.1.8 中有捕获/比较影子寄存器,该寄存器不可直接访问。

图 21.1.8 左下角的 CC1G 位可以产生软件捕获事件，那么硬件捕获事件如何产生的？这里我们还是以通道 1 输入为例，CC1S[1:0] = 01，即 IC1 映射到 TI1 上；CC1E 位置 1，使能输入捕获；比如不滤波、不分频，ICF[3:0] = 00，ICPS[1:0] = 00；比如检测上升沿，CC1P 位置 0；接着就是等待测量信号的上升沿到来。当上升沿到来时，IC1PS 信号就会触发输入捕获事件发生，计数器的值就会被锁存到捕获/比较影子寄存器里。当 CCR1 寄存器没有被进行读操作的时候，捕获/比较影子寄存器里的值就会锁存到 CCR1 寄存器中，那么程序员就可以读取 CCR1 寄存器，得到计数器的计数值。检测下降沿同理。

⑤ 输入捕获和输出比较公用部分

该部分需要结合第④部分或者第⑥部分共同完成相应功能。

⑥ 输出比较

图 21.1.1 中的第⑥部分是输出比较，一般应用是要和第⑤部分一起完成定时器输出功能。TIMx_CH1~TIMx_CH4 表示定时器的 4 个通道，这 4 个通道都是可以独立工作的。IO 端口通过复用功能与这些通道相连。

下面我们按照输出信号产生过程顺序给大家介绍定时器如何实现输出功能的？首先看到第⑤部分的“放大版”图，如下图所示：

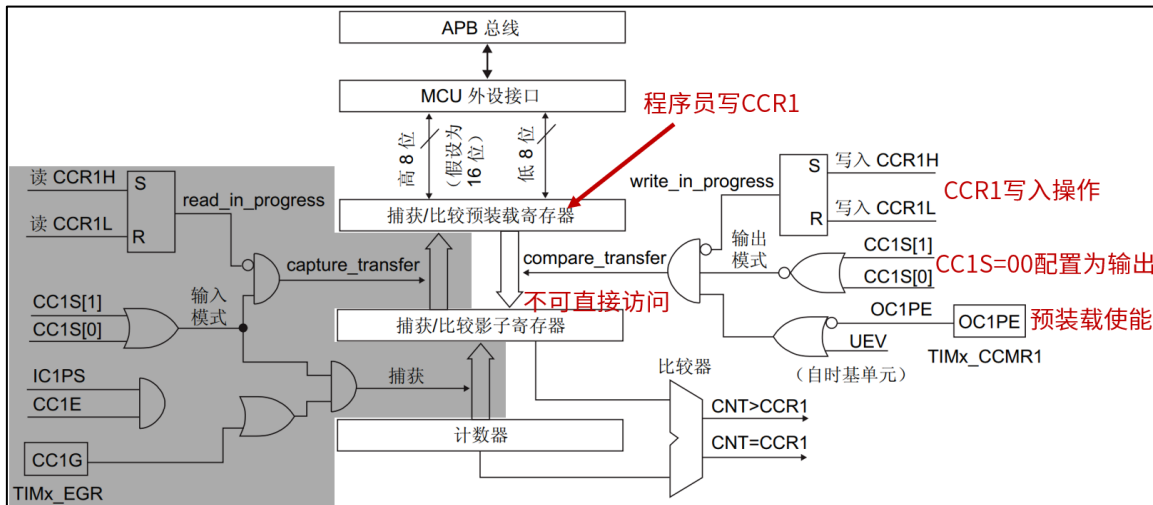


图 21.1.9 捕获/比较通道 1 主电路 (输出比较功能部分)

图 21.1.9 中，灰色阴影部分是输入捕获功能部分，前面已经讲过。这里我们看到右边没有阴影部分就是输出比较功能部分了。下面以通道 1 输出比较功能为例给大家介绍定时器如何实现输出功能的。

首先程序员写 CCR1 寄存器，即写入比较值。这个比较值需要转移到对应的捕获/比较影子寄存器后才会真正生效。什么条件下才能转移？图 21.1.9 中可以看到 compare_transfer 旁边的与门，需要满足三个条件：CCR1 不在写入操作期间、CCIS[1:0]=0 配置为输出、OC1PE 位置 0（或者 OC1PE 位置 1，并且需要发生更新事件，这个更新事件可以软件产生或者硬件产生）。

当 CCR1 寄存器的值转移到其影子寄存器后，新的值就会和计数器的值进行比较，它们的比较结果将会通过第⑥部分影响定时器的输出。

下面来看看第⑥部分通道 1 的“放大版”，如下图所示：

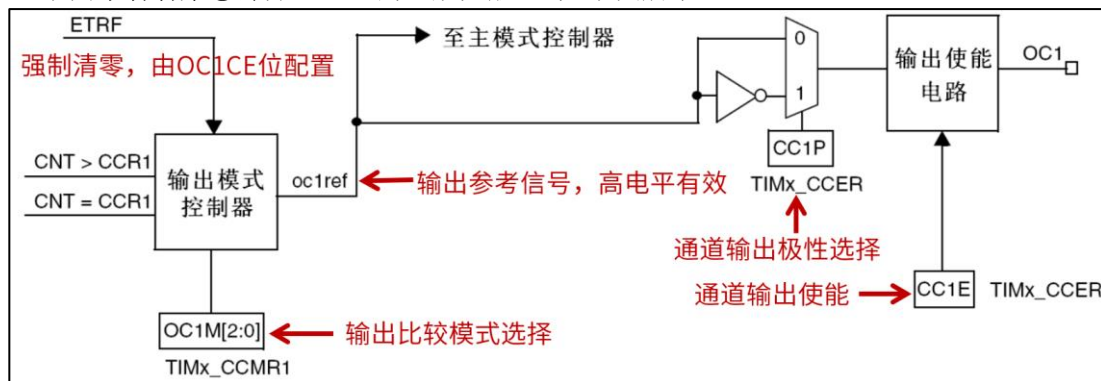


图 21.1.10 通道 1 输出阶段

上图中，可以看到输出模式控制器，由 OC1M[2:0] 位配置输出比较模式，该位的描述请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》相关定时器章节的 TIMx_CCMR1 寄存器。F1 系列有 8 种输出比较模式之多，后面用到再来介绍。

oc1ref 是输出参考信号，高电平有效，为高电平时称之为有效电平，为低电平时称之为无效电平。它的高低电平受到三个方面的影响：OC1M[3:0] 位配置的输出比较模式、第⑤部分比较器的比较结果、还有就是 OC1CE 位配置的 ETRF 信号。ETRF 信号可以将 Oc1ref 电平强制清零，该信号来自 IO 外部。

一般来说，当计数器的值和捕获/比较寄存器的值相等时，输出参考信号 oc1ref 的极性就会根据我们选择的输出比较模式而改变。如果开启了比较中断，还会发生比较中断。

CC1P 位用于选择通道输出极性。

CC1E 位置 1 使能通道输出。

OC1 信号就会从 TIMx_CH1 输出到 IO 端口，再到 IO 外部。

下面分别通过四个实验来学习通用定时器的功能。

21.2 通用定时器中断实验

本小节我们来学习使用通用定时器中断，以定时器 3 中断为例，首先来了解相关的寄存器。

21.2.1 TIM2/TIM3/TIM4/TIM5 寄存器

下面介绍 TIM2/TIM3/TIM4/TIM5 的几个与定时器中断相关且重要的寄存器，相关内容可以参考《STM32F10xxx 参考手册_V10（中文版）.pdf》定时器的相关章节。

● 控制寄存器 1 (TIMx_CR1)

TIM2/TIM3/TIM4/TIM5 的控制寄存器 1 描述如图 21.2.1.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|--|----|----|----|----------|----|------|----------|----|-----|-----|-----|------|-----|
| 保留 | | | | | | CKD[1:0] | | ARPE | CMS[1:0] | | DIR | OPM | URS | UDIS | CEN |
| rW | | | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位7 | | ARPE: 自动重载预装载允许位 (Auto-reload preload enable) 0: TIMx_ARR寄存器没有缓冲; 1: TIMx_ARR寄存器被装入缓冲器。 | | | | | | | | | | | | | |
| 位6:5 | | CMS[1:0]: 选择中央对齐模式 (Center-aligned mode selection) 00: 边沿对齐模式。计数器依据方向位(DIR)向上或向下计数。 01: 中央对齐模式1。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 只在计数器向下计数时被设置。 10: 中央对齐模式2。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 只在计数器向上计数时被设置。 11: 中央对齐模式3。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位, 在计数器向上和向下计数时均被设置。 注: 在计数器开启时(CEN=1), 不允许从边沿对齐模式转换到中央对齐模式。 | | | | | | | | | | | | | |
| 位4 | | DIR: 方向 (Direction) 0: 计数器向上计数; 1: 计数器向下计数。 注: 当计数器配置为中央对齐模式或编码器模式时, 该位为只读。 | | | | | | | | | | | | | |
| 位0 | | CEN: 使能计数器 0: 禁止计数器; 1: 使能计数器。 注: 在软件设置了CEN位后, 外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 在单脉冲模式下, 当发生更新事件时, CEN被自动清除。 | | | | | | | | | | | | | |

图 21.2.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的一些位, 其中: 位 7 (APRE) 用于控制自动重载寄存器是否进行缓冲, 如果 ARPE 位置 1, ARR 起缓冲作用, 即只有在更新事件发生时才会把 ARR 的值写入其影子寄存器里; 如果 ARPE 位置 0, 那么修改自动重载寄存器的值时, 该值会马上被写入其影子寄存器中, 从而立即生效。

CMS[1:0]位, 用于设置边沿对齐模式还是中心对齐模式, 本实验我们使用边沿对齐模式, 所以设置为 00 即可。

DIR 位, 用于控制定时器的计数方向, 我们使用递增计数模式, 所以设置 DIR 位为 0。

CEN 位, 用于使能计数器的工作, 必须要设置该位为 1, 计数器才会开始计数。

● 从模式控制寄存器 (TIMx_SMCR)

TIM2/TIM3/TIM4/TIM5 的从模式控制寄存器描述如图 21.2.1.2 所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|-----|--|----|----------|----|----|-----|---------|----|----|----|----------|----|----|----|
| ETP | ECE | ETPS[1:0] | | ETF[3:0] | | | MSM | TS[2:0] | | 保留 | | SMS[2:0] | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位2:0 | | SMS[2:0]: 从模式选择 (Slave mode selection) 当选择了外部信号, 触发信号(TRGI)的有效边沿与选中的外部输入极性相关(见输入控制寄存器和控制寄存器的说明) 000: 关闭从模式 – 如果CEN=1, 则预分频器直接由内部时钟驱动。 001: 编码器模式1 – 根据TI1FP1的电平, 计数器在TI2FP2的边沿向上/下计数。 010: 编码器模式2 – 根据TI2FP2的电平, 计数器在TI1FP1的边沿向上/下计数。 011: 编码器模式3 – 根据另一个信号的输入电平, 计数器在TI1FP1和TI2FP2的边沿向上/下计数。 100: 复位模式 – 选中的触发输入(TRGI)的上升沿重新初始化计数器, 并且产生一个更新寄存器的信号。 101: 门控模式 – 当触发输入(TRGI)为高时, 计数器的时钟开启。一旦触发输入变为低, 则计数器停止(但不复位)。计数器的启动和停止都是受控的。 110: 触发模式 – 计数器在触发输入TRGI的上升沿启动(但不复位), 只有计数器的启动是受控的。 111: 外部时钟模式1 – 选中的触发输入(TRGI)的上升沿驱动计数器。 注: 如果TI1F_EN被选为触发输入(TS=100)时, 不要使用门控模式。这是因为, TI1F_ED在每次TI1F变化时输出一个脉冲, 然而门控模式是要检查触发输入的电平。 | | | | | | | | | | | | | |

图 21.2.1.2 TIMx_SMCR 寄存器

该寄存器的 SMS[2:0]位，用于从模式选择，其实就是选择计数器输入时钟的来源。比如通用定时器中断实验我们设置 SMS[2:0]=000，禁止从模式，这样 PSC 预分频器的时钟就直接来自内部时钟（CK_INT），按照我们例程 sys_stm32_clock_init 函数的配置，频率为 72Mhz（APB1 总线时钟频率的 2 倍）。

● DMA/中断使能寄存器（TIMx_DIER）

TIM2/TIM3/TIM4/TIM5 的 DMA/中断使能寄存器描述如图 21.2.1.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|-----|---|-------|-------|-------|-------|-----|----|-----|----|-------|-------|-------|-------|-----|
| 保留 | TDE | 保留 | CC4DE | CC3DE | CC2DE | CC1DE | UDE | 保留 | TIE | 保留 | CC4IE | CC3IE | CC2IE | CC1IE | UIE |
| 位0 | | UIE：允许更新中断 (Update interrupt enable) 0：禁止更新中断； 1：允许更新中断。 | | | | | | | | | | | | | |

图 21.2.1.3 TIMx_DIER 寄存器

该寄存器用于使能/失能触发 DMA 请求、捕获/比较中断以及更新中断。本实验只用到更新中断，所以把位 0（UIE）置 1 即可。

● 状态寄存器（TIMx_SR）

TIM2/TIM3/TIM4/TIM5 的状态寄存器描述如图 21.2.1.4 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|--|-------|-------|-------|----|---|-----|-------|-------|-------|-------|-------|-------|
| 保留 | | | CC4OF | CC3OF | CC2OF | CC1OF | 保留 | | TIF | 保留 | CC4IF | CC3IF | CC2IF | CC1IF | UIF |
| | | | rc w0 | rc w0 | rc w0 | rc w0 | | | | rc w0 | | rc w0 | rc w0 | rc w0 | rc w0 |
| 位0 | | | UIF：更新中断标记 (Update interrupt flag) | | | | | | | | | | | | |
| | | | 当产生更新事件时该位由硬件置'1'。它由软件清'0'。 | | | | | | | | | | | | |
| | | | 0：无更新事件产生； | | | | | | | | | | | | |
| | | | 1：更新中断等待响应。当寄存器被更新时该位由硬件置'1'： | | | | | | | | | | | | |
| | | | <div><div>— 若TIMx_CR1寄存器的UDIS=0、URS=0，当TIMx_EGR寄存器的UG=1时产生更新事件(软件对计数器CNT重新初始化)；</div><div>— 若TIMx_CR1寄存器的UDIS=0、URS=0，当计数器CNT被触发事件重初始化时产生更新事件。(参考同步控制寄存器的说明)</div></div> | | | | | | | | | | | | |

图 21.2.1.4 TIMx_SR 寄存器

该寄存器都是一些中断标志位，比如更新中断标志位、捕获/比较中断标志位等。在通用定时器中断实验我们用到更新中断标志位，当定时器更新中断到来后，位 0（UIF）会由硬件置 1，我们需要在中断服务函数里面把该位清零。

● 计数寄存器（TIMx_CNT）

TIM2/TIM3/TIM4/TIM5 的计数器寄存器描述如图 21.2.1.5 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|----|--|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CNT[15:0] | | | | | | | | | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位15:0 | | CNT[15:0]：计数器的值 (Counter value) | | | | | | | | | | | | | |

图 21.2.1.5 TIMx_CNT 寄存器

TIM2/TIM3/TIM4/TIM5 的计数寄存器都是 16 位有效的，计数模式可以是递增计数模式、递减计数模式和中心对齐计数模式，计数值范围 0~65535。可以直接写该寄存器设置计数的初始值，也可以读取该寄存器获取计数器值。

● 预分频寄存器（TIMx_PSC）

TIM2/TIM3/TIM4/TIM5 的预分频器寄存器描述如图 21.2.1.6 所示。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|----|--|----|----|----|----|----|----|----|----|----|----|----|----|----|
| PSC[15:0] | | | | | | | | | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位15:0 | | PSC[15:0]：预分频器的值 (Prescaler value) 计数器的时钟频率CK_CNT等于f _{CK_PSC} /(PSC[15:0]+1)。 PSC包含了当更新事件产生时装入当前预分频器寄存器的值。 | | | | | | | | | | | | | |

图 21.2.1.6 TIMx_PSC 寄存器

定时器的预分频寄存器都是 16 位的，即写入该寄存器的数值范围是 0 到 65535，表示 1 到 65536 分频。比如我们要 7200 分频，就往该寄存器写入 7199。

● 自动重载寄存器（TIMx_ARR）

TIM2/TIM3/TIM4/TIM5 的自动重载寄存器描述如图 21.2.1.7 所示。

| | | | | | | | | | | | | | | | |
|-----------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ARR[15:0] | | | | | | | | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位15:0 | | ARR[15:0]: 自动重载的值 (Auto reload value) ARR包含了将要传送至实际的自动重载寄存器的数值。 详细参考14.3.1节：有关ARR的更新和动作。 当自动重载的值为空时，计数器不工作。 | | | | | | | | | | | | | |

图 21.2.1.7 TIMx_ARR 寄存器

自动重载寄存器是低 16 位有效。该寄存器可以由 APRE 位设置是否进行缓冲。计数器的值会和自动重载寄存器影子寄存器进行比较，当两者相等，定时器就会溢出，从而发生更新事件，如果打开了更新中断，还会发生更新中断。

21.2.2 硬件设计

1. 例程功能

LED0 用来指示程序正在运行，200ms 翻转一次。LED1 在定时器中断中翻转，500ms 进入中断一次。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 定时器 3

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 LED1 来指示 STM32F103 的定时器进入中断的频率，LED0 则指示程序的运行状态。

21.2.3 程序设计

本实验的相关 HAL 库驱动以及实验配置步骤请参考基本定时器相关内容，基本一样。不同点是基本定时器只能是递增计数模式，通用定时器可以递增计数模式、递减计数模式和中心对齐模式。

21.2.3.1 程序流程图

下面看看本实验的程序流程图，main 函数中并没有对 LED1 的操作，我们把对 LED1 的操作放到定时器的中断中进行处理：

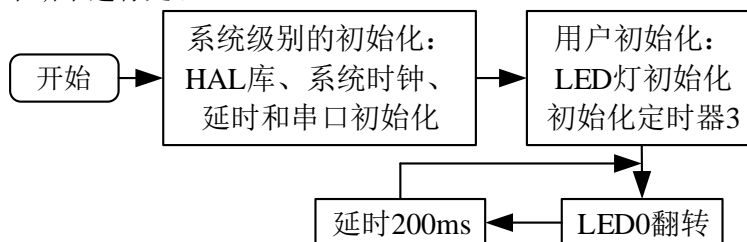


图 21.2.3.1.1 通用定时器中断实验程序流程图

21.2.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。通用定时器驱动源码包括两个文件：`gtim.c` 和 `gtim.h`。

首先看 `gtim.h` 头文件的几个宏定义：

```
/* TIMX 中断定义
 * 默认是针对 TIM2~TIM5
 * 注意：通过修改这 4 个宏定义，可以支持 TIM1~TIM8 任意一个定时器。
 */

#define GTIM_TIMX_INT          TIM3
#define GTIM_TIMX_INT_IRQn     TIM3_IRQn
#define GTIM_TIMX_INT_IRQHandler TIM3_IRQHandler
/* TIM3 时钟使能 */
#define GTIM_TIMX_INT_CLK_ENABLE() do{ __HAL_RCC_TIM3_CLK_ENABLE(); }while(0)
```

通过修改这 4 个宏定义，可以支持 TIM1~TIM8 任意一个定时器。

下面再来看一下 `gtim.c` 文件的代码，主要包括两个函数，先来看看通用定时器的初始化函数，其定义如下：

```
/**
 * @brief      通用定时器 TIMx 定时中断初始化函数
 * @note
 *
 *      通用定时器的时钟来自 APB1，当 PPRE1 ≥ 2 分频的时候
 *      通用定时器的时钟为 APB1 时钟的 2 倍，而 APB1 为 36M，所以定时器时钟 = 72Mhz
 *      定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft=定时器工作频率，单位:Mhz
 *
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void gtim_timx_int_init(uint16_t arr, uint16_t psc)
{
    GTIM_TIMX_INT_CLK_ENABLE(); /* 使能 TIMx 时钟 */

    g_timx_handle.Instance = GTIM_TIMX_INT; /* 通用定时器 x */
    g_timx_handle.Init.Prescaler = psc; /* 预分频系数 */
    g_timx_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数模式 */
    g_timx_handle.Init.Period = arr; /* 自动装载值 */
    HAL_TIM_Base_Init(&g_timx_handle);

    /* 设置中断优先级，抢占优先级 1，子优先级 3 */
    HAL_NVIC_SetPriority(GTIM_TIMX_INT_IRQn, 1, 3);
    HAL_NVIC_EnableIRQ(GTIM_TIMX_INT_IRQn); /* 开启 ITMx 中断 */

    HAL_TIM_Base_Start_IT(&g_timx_handle); /* 使能定时器 x 和定时器 x 更新中断 */
}
```

这里配置的参数和基本定时器中断实验的是一样的，只是这里没有使用到 HAL 库的 `HAL_TIM_Base_MspInit` 函数来存放 NVIC 和使能时钟的代码，而是全部存放到 `gtim_timx_int_init` 函数里。在一个项目中，用到多个定时器时，建议大家使用这种方式来处理代码，这样方便代码的管理。

下面再来看看定时器中断服务函数，其定义如下：

```
/**
 * @brief      定时器中断服务函数
 * @param      无
 * @retval     无
 */
void GTIM_TIMX_INT_IRQHandler(void)
{
    /* 以下代码没有使用定时器 HAL 库共用处理函数来处理，而是直接通过判断中断标志位的方式 */
}
```

```
if (__HAL_TIM_GET_FLAG(&g_timx_handle, TIM_FLAG_UPDATE) != RESET)
{
    LED1_TOGGLE();
    /* 清除定时器溢出中断标志位 */
    __HAL_TIM_CLEAR_IT(&g_timx_handle, TIM_IT_UPDATE);
}
}
```

可以看到，这里我们没有使用 HAL 库的定时器公共处理函数来处理中断部分的代码，而是通过自行判断中断标志位的方式来处理。只不过获取标志位的方式还是使用 HAL 库的函数宏 `__HAL_TIM_GET_FLAG()`，大家也可以直接使用寄存器的方式来操作。

通过 `__HAL_TIM_GET_FLAG()` 获取中断标志位并判断是否发生了中断，然后处理中断程序，最后通过 `__HAL_TIM_CLEAR_IT()` 将中断标志位清零，这样就完成了一次对中断的处理。这样的方式来处理中断，也是大家学习 HAL 库需要掌握的。在一个项目中，用到多个定时器相关中断时，建议大家使用这种方式来处理代码，这样方便代码的管理。

21.2.4 下载验证

下载代码后，可以看到 LED0 不停闪烁（每 400ms 一个周期），而 LED1 也是不停的闪烁，但是闪烁时间较 LED0 慢（每 1s 一个周期）。

21.3 通用定时器 PWM 输出实验

本小节我们来学习使用通用定时器的 PWM 输出模式。

脉冲宽度调制(PWM), 是英文 “Pulse Width Modulation” 的缩写, 简称脉宽调制, 是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。我们可以让定时器产生 PWM, 在计数器频率固定时, PWM 频率或者周期由自动重载寄存器 (TIMx_ARR) 的值决定, 其占空比由捕获/比较寄存器 (TIMx_CCRx) 的值决定。PWM 产生原理示意图如下图所示:

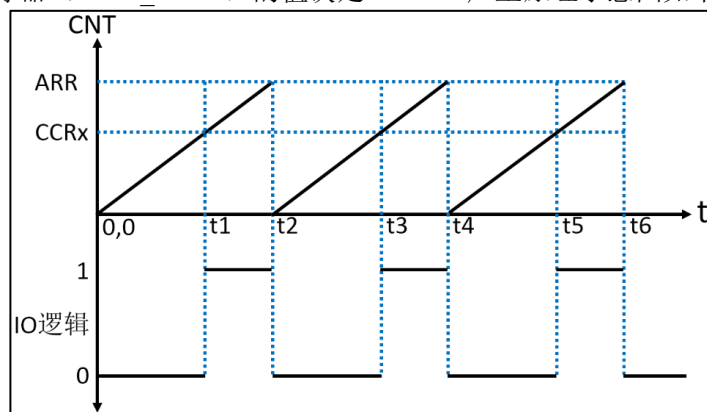


图 21.3.1 PWM 生成示意图

上图中, 定时器工作在递增计数模式, 纵轴是计数器的计数值 CNT, 横轴表示时。当 $CNT < CCRx$ 时, IO 输出低电平 (逻辑 0); 当 $CNT \geq CCRx$ 时, IO 输出高电平 (逻辑 1); 当 $CNT = ARR$ 时, 定时器溢出, CNT 的值被清零, 然后继续递增, 依次循环。在这个循环中, 改变 CCRx 的值, 就可以改变 PWM 的占空比, 改变 ARR 的值, 就可以改变 PWM 的频率, 这就是 PWM 输出的原理。

定时器产生 PWM 的方式有许多种, 下面我们以前沿对齐模式 (即递增计数模式/递减计数模式) 为例, PWM 模式 1 或者 PWM 模式 2 产生 PWM 的示意图, 如下图所示:

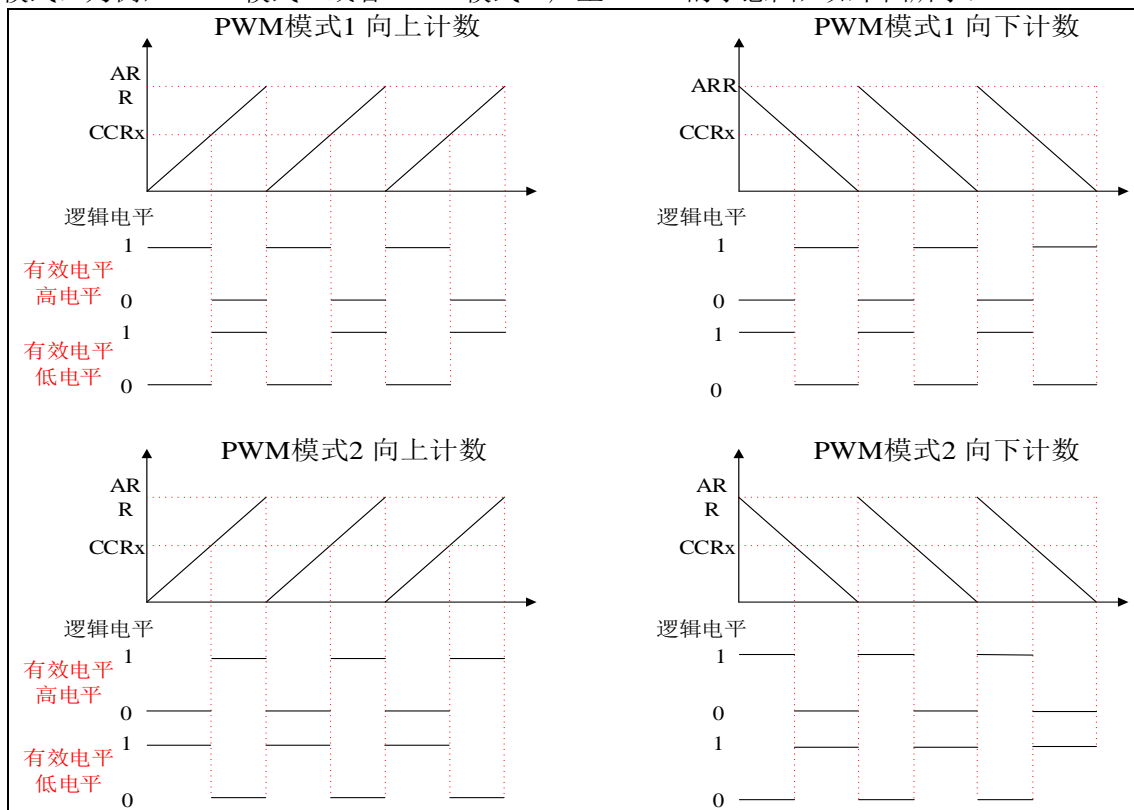


图 21.3.2 产生 PWM 示意图

STM32F103 的定时器除了 TIM6 和 TIM7，其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出！本实验我们以使用 TIM3 的 CH2 产生一路 PWM 输出为例进行学习。

21.3.1 TIM2/TIM3/TIM4/TIM5 寄存器

要使 STM32F103 的通用定时器 TIMx 产生 PWM 输出，除了上一小节介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM。这三个寄存器分别是：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）。接下来我们简单介绍一下这三个寄存器。

● 捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）

TIM2/TIM3/TIM4/TIM5 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 21.3.1.1 所示：

| | | | | | | | | | | | | | | | | |
|---|----|-----------|----|-------------|-------|-----------|-----------|----|-------------|-----------|----|----|-------|-------|-----------|--|
| 通道可用于输入(捕获模式)或输出(比较模式)，通道的方向由相应的CCxS定义。该寄存器其它位的作用在输入和输出模式下不同。OCxx描述了通道在输出模式下的功能，ICxx描述了通道在输出模式下的功能。因此必须注意，同一个位在输出模式和输入模式下的功能是不同的。 | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| OC2CE | | OC2M[2:0] | | | OC2PE | OC2FE | CC2S[1:0] | | OC1CE | OC1M[2:0] | | | OC1PE | OC1FE | CC1S[1:0] | |
| IC2F[3:0] | | | | IC2PSC[1:0] | | IC1F[3:0] | | | IC1PSC[1:0] | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | |

图 21.3.1.1 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在只用到输出比较，输入捕获后面的实验再讲解。关于该寄存器的详细说明，请参考《STM32F10xxx 参考手册_V10(中文版).pdf》第 288 页，14.4.7 节。比如我们要让 TIM3 的 CH2 输出 PWM 波为例进行介绍，该寄存器的模式设置位 OC2M[2:0]就是对应着通道 2 的模式设置，此部分由 3 位组成。总共可以配置成 8 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110 或者 111，分别对应 PWM 模式 1 和 PWM 模式 2。这两种 PWM 模式的区分就是输出有效电平的极性相反。位 OC2PE 控制输出比较通道 2 的预装载使能，实际就是控制 CCR2 寄存器是否进行缓冲。因为 CCR2 寄存器也是有影子寄存器的，影子寄存器才是真正起作用的寄存器。CC2S[1:0]用于设置通道 2 的方向（输入/输出）默认设置为 0，就是设置通道作为输出使用。

● 捕获/比较使能寄存器（TIMx_CCER）

TIM2/TIM3/TIM4/TIM5 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 21.3.1.2 所示：

| | | | | | | | | | | | | | | | |
|----|----|------|------|----|----|------|------|----|---|------|------|----|---|------|------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | CC4P | CC4E | 保留 | | CC3P | CC3E | 保留 | | CC2P | CC2E | 保留 | | CC1P | CC1E |
| RW | | RW | RW | RW | | RW | RW | RW | | RW | RW | RW | | RW | RW |

图 21.3.1.2 TIMx_CCER 寄存器

该寄存器比较简单，要让 TIM3 的 CH2 输出 PWM 波，这里我们要使能 CC2E 位，该位是通道 2 输入/输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1。CC2P 位是设置通道 2 的输出极性，我们默认设置 0。

● 捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）

捕获/比较寄存器（TIMx_CCR1/2/3/4），该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 2，所以来看看 TIMx_CCR2 寄存器，描述如图 21.3.1.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CCR2[15:0] | | | | | | | | | | | | | | | |
| IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW | IW |
| 位15:0 | | CCR2[15:0]: 捕获/比较2的值 (Capture/Compare 2 value) 若CC2通道配置为输出: CCR2包含了装入当前捕获/比较2寄存器的值(预装载值)。 如果在TIMx_CCMR2寄存器(OC2PE位)中未选择预装载特性, 写入的数值会被立即传输至当前寄存器中。否则只有当更新事件发生时, 此预装载值才传输至当前捕获/比较2寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较, 并在OC2端口上产生输出信号。 若CC2通道配置为输入: CCR2包含了由上一次输入捕获2事件(IC2)传输的计数器值。 | | | | | | | | | | | | | |

图 21.3.1.3 TIMx_CCR2 寄存器

在输出模式下, 捕获/比较寄存器影子寄存器的值与 CNT 的值比较, 根据比较结果产生相应动作, 利用这点, 我们通过修改这个寄存器的值, 就可以控制 PWM 的占空比了。

21.3.2 硬件设计

1. 例程功能

使用 TIM3 通道 2 (由 PB5 复用) 输出 PWM, PB5 引脚连接了 LED0, 从而实现 PWM 输出控制 LED0 亮度。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 定时器 3 输出通道 2 (由 PB5 复用)

3. 原理图

定时器属于 STM32F103 的内部资源, 只需要软件设置好即可正常工作。我们通过 LED0 来间接指示定时器的 PWM 输出情况。

21.3.3 程序设计

21.3.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面介绍基本定时器已经介绍了部分, 这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_PWM_Init 函数

定时器的 PWM 输出模式初始化函数, 其声明如下:

```
HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef *htim);
```

- **函数描述:**
用于初始化定时器的 PWM 输出模式。
- **函数形参:**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量, 基本定时器的时候已经介绍。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。
- **注意事项:**

该函数实现的功能以及使用方法和 HAL_TIM_Base_Init 类似, 作用都是初始化定时器的 ARR 和 PSC 等参数。为什么 HAL 库要提供这个函数而不直接让我们使用 HAL_TIM_Base_Init 函数呢? 这是因为 HAL 库为定时器的针对 PWM 输出定义了单独的 MSP 回调函数 HAL_TIM_PWM_MspInit, 所以当我们调用 HAL_TIM_PWM_Init 进行 PWM 初始化之后, 该函

数内部会调用 MSP 回调函数 HAL_TIM_PWM_MspInit。当我们使用 HAL_TIM_Base_Init 初始化定时器参数的时候，它内部调用的回调函数是 HAL_TIM_Base_MspInit，这里大家注意区分。

2. HAL_TIM_PWM_ConfigChannel 函数

定时器的 PWM 通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_PWM_ConfigChannel(TIM_HandleTypeDef *htim,
                                             TIM_OC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**

该函数用于设置定时器的 PWM 通道。

- **函数形参：**

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。

形参 2 是 TIM_OC_InitTypeDef 结构体类型指针变量，用于配置定时器的输出比较参数。

重点了解一下 TIM_OC_InitTypeDef 结构体指针类型，其定义如下：

```
typedef struct
{
    uint32_t OCMODE;          /* 输出比较模式选择，寄存器的时候说过了，共 8 种模式 */
    uint32_t Pulse;           /* 设置比较值 */
    uint32_t OCPolarity;      /* 设置输出比较极性 */
    uint32_t OCNPolarity;     /* 设置互补输出比较极性 */
    uint32_t OCFAstMode;      /* 使能或失能输出比较快速模式 */
    uint32_t OCIdleState;     /* 选择空闲状态下的非工作状态（OC1 输出） */
    uint32_t OCNIdleState;    /* 设置空闲状态下的非工作状态（OC1N 输出） */
} TIM_OC_InitTypeDef;
```

我们重点关注前三个结构体成员。成员变量 OCMODE 用来设置模式，这里我们设置为 PWM 模式 1。成员变量 Pulse 用来设置捕获比较值。成员变量 TIM_OC_Polarity 用来设置输出极性。其他成员 TIM_OutputNState，TIM_OCNPolarity，TIM_OCIdleState 和 TIM_OCNIdleState 后面用到再介绍。

形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。这里我们使用的是定时器 3 的通道 2，所以取值为 TIM_CHANNEL_2 即可。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_PWM_Start 函数

定时器的 PWM 输出启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

- **函数描述：**

用于使能通道输出和启动计数器，即启动 PWM 输出。

- **函数形参：**

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

对于单独使能定时器的方法，在上一章定时器实验我们已经讲解。实际上，HAL 库也同样提供了单独使能定时器的输出通道函数，函数为：

```
void TIM_CCxChannelCmd(TIM_TypeDef *TIMx, uint32_t Channel,
                       uint32_t ChannelState);
```

HAL_TIM_PWM_Start 函数内部也调用了该函数。

4. HAL_TIM_ConfigClockSource 函数

配置定时器时钟源函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_ConfigClockSource(TIM_HandleTypeDef *htim,
                                             TIM_ClockConfigTypeDef *sClockSourceConfig);
```

- **函数描述：**

用于配置定时器时钟源。

- **函数形参：**

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

形参 2 是 TIM_ClockConfigTypeDef 结构体类型指针变量，用于配置定时器时钟源参数。

TIM_ClockConfigTypeDef 定义如下：

```
typedef struct
{
    uint32_t ClockSource;      /* 时钟源 */
    uint32_t ClockPolarity;    /* 时钟极性 */
    uint32_t ClockPrescaler;   /* 定时器预分频器 */
    uint32_t ClockFilter;      /* 时钟过滤器 */
} TIM_ClockConfigTypeDef;
```

- 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

- 注意事项：

该函数主要配置 TIMx_SMCR 寄存器。默认情况下，定时器的时钟源是内部时钟。本实验就是使用内部时钟的，所以我们不用对时钟源就行初始化，默认即可。这里只是让大家知道有这个函数可以设定定时器的时钟源。比如用 HAL_TIM_ConfigClockSource 初始化选择内部时钟，方法如下：

```
TIM_HandleTypeDef timx_handle;      /* 定时器 x 句柄 */
TIM_ClockConfigTypeDef sClockSourceConfig = {0};
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL; /* 选择内部时钟 */
HAL_TIM_ConfigClockSource(&timx_handle, &sClockSourceConfig);
```

后面的定时器初始化凡是用到内部时钟我们都没有去初始化，系统默认即可。

定时器 PWM 输出模式配置步骤

1) 开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 3 通道 2，对应 IO 是 PB5，它们的时钟开启方法如下：

```
_HAL_RCC_TIM3_CLK_ENABLE(); /* 使能定时器 3 */
_HAL_RCC_GPIOB_CLK_ENABLE(); /* 开启 GPIOB 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的 PWM 输出功能时，通过 HAL_TIM_PWM_Init 函数初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3) 设置 TIMx_CHy 的 PWM 模式，输出比较极性，比较值等参数

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

4) 使能 TIMx，使能 TIMx 的 CHy 输出

在 HAL 库中，通过调用 HAL_TIM_PWM_Start 函数来使能 TIMx 的某个通道输出 PWM。

5) 修改 TIM3_CCR2 来控制占空比

在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而我们通过修改比较值来控制 PWM 的输出占空比。HAL 库中提供一个修改占空比的宏定义：

```
_HAL_TIM_SET_COMPARE ( __HANDLE__, __CHANNEL__, __COMPARE__ )
```

__HANDLE__ 是 TIM_HandleTypeDef 结构体类型指针变量，__CHANNEL__ 对应 PWM 的输出通道，__COMPARE__ 则是要写到捕获/比较寄存器（TIMx_CCR1/2/3/4）的值。实际上该宏定义最终还是往对应的捕获/比较寄存器写入比较值来控制 PWM 波的占空比，如下解析：

比如我们要修改定时器 3 通道 2 的输出比较值（控制占空比），寄存器操作方法：

```
TIM3->CCR2 = ledrpwmval; /* ledrpwmval 是比较值，并且动态变化的，
                           所以我们要周期性调用这条语句，已达到及时修改 PWM 的占空比 */
```

__HAL_TIM_SET_COMPARE 这个宏定义函数最终也是调用这个寄存器操作的，所以说我们使用 HAL 库的函数其实就是间接操作寄存器的。

21.3.3.2 程序流程图

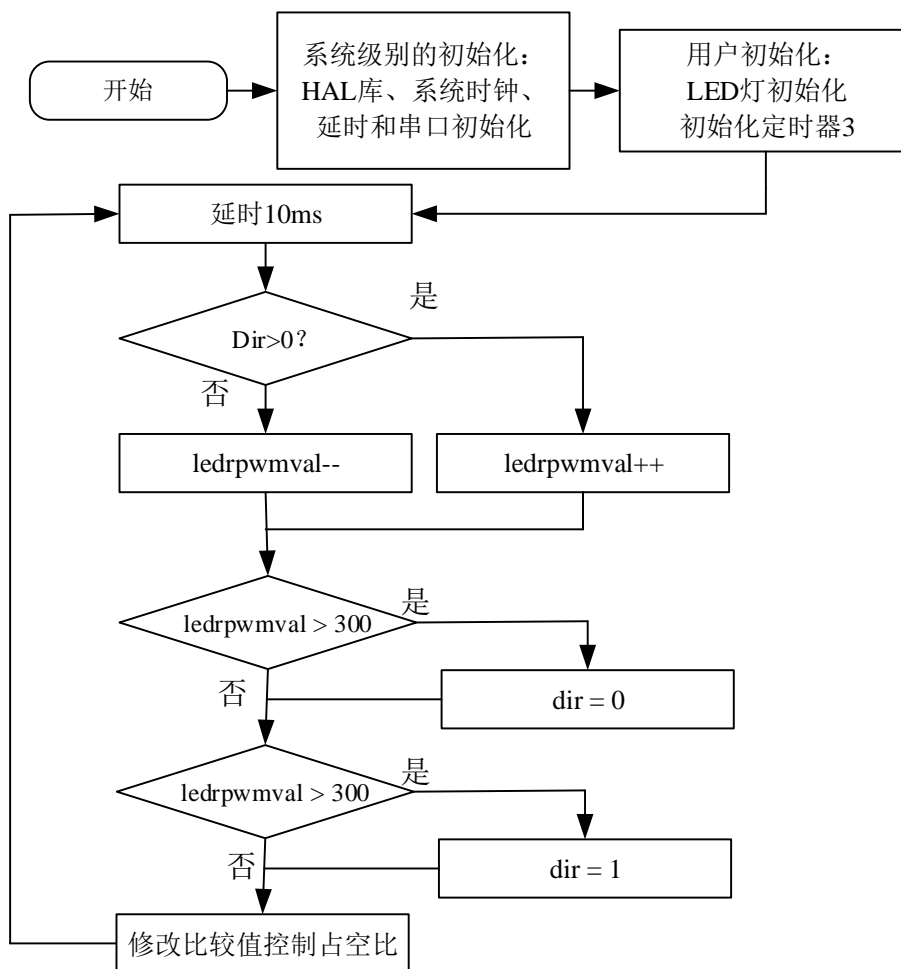


图 21.3.3.2.1 通用定时器 PWM 输出实验程序流程图

21.3.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。通用定时器驱动源码包括两个文件：`gtim.c` 和 `gtim.h`。

首先看 `gtim.h` 头文件的几个宏定义：

```

/* TIMX PWM 输出定义
 * 这里输出的 PWM 控制 LED0 (RED) 的亮度
 * 默认是针对 TIM2~TIM5
 * 注意：通过修改这几个宏定义,可以支持 TIM1~TIM8 任意一个定时器,任意一个 IO 口输出 PWM
 */
#define GTIM_TIMX_PWM_CHY_GPIO_PORT      GPIOB
#define GTIM_TIMX_PWM_CHY_GPIO_PIN      GPIO_PIN_5
#define GTIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE() do{__HAL_RCC_GPIOB_CLK_ENABLE();\
}while(0) /* PB 口时钟使能 */

/* TIMX REMAP 设置
 * 因为我们 LED0 接在 PB5 上,必须通过开启 TIM3 的部分重映射功能,才能将 TIM3_CH2 输出到 PB5 上
 * 因此,必须实现 GTIM_TIMX_PWM_CHY_GPIO_REMAP
 * 对那些使用默认设置的定时器 PWM 输出脚,不用设置重映射,是不需要该函数的!
 */
#define GTIM_TIMX_PWM_CHY_GPIO_REMAP() do{__HAL_RCC_AFIO_CLK_ENABLE();\
__HAL_AFIO_REMAP_TIM3_PARTIAL();\

```



```

}while(0)

#define GTIM_TIMX_PWM TIM3
#define GTIM_TIMX_PWM_CHY TIM_CHANNEL_2 /* 通道 Y, 1<= Y <=4 */
#define GTIM_TIMX_PWM_CHY_CCRX TIM3->CCR2 /* 通道 Y 的输出比较寄存器 */
#define GTIM_TIMX_PWM_CHY_CLK_ENABLE() do{ __HAL_RCC_TIM3_CLK_ENABLE();\
}while(0) /* TIM3 时钟使能 */

```

可以把上面的宏定义分成三部分，第一部分是定时器 3 输出通道 2 对应的 IO 口的宏定义。第二部分是定时器 3 的部分重映射功能的宏定义，第三部分则是定时器 3 输出通道 2 的相应宏定义。

下面看 `gtim.c` 的程序，首先是通用定时器 PWM 输出初始化函数。

```

/**
 * @brief      通用定时器 TIMX 通道 Y PWM 输出 初始化函数（使用 PWM 模式 1）
 * @note
 *
 *      通用定时器的时钟来自 APB1,当 D2PPRE1≥2 分频的时候
 *      通用定时器的时钟为 APB1 时钟的 2 倍，而 APB1 为 36M，所以定时器时钟 = 72Mhz
 *      定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft=定时器工作频率,单位:Mhz
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void gtim_timx_pwm_chy_init(uint16_t arr,uint16_t psc)
{
    g_timx_pwm_chy_handle.Instance = GTIM_TIMX_PWM; /* 定时器 x */
    g_timx_pwm_chy_handle.Init.Prescaler = psc; /* 定时器分频 */
    g_timx_pwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数模式 */
    g_timx_pwm_chy_handle.Init.Period = arr; /* 自动重装值 */
    HAL_TIM_PWM_Init(&g_timx_pwm_chy_handle); /* 初始化 PWM */

    g_timx_oc_pwm_chy_handle.OCMode = TIM_OCMODE_PWM1; /* 模式选择 PWM1 */
    /* 设置比较值,此值用来确定占空比，默认比较值为自动重装值的一半,即占空比为 50% */
    g_timx_oc_pwm_chy_handle.Pulse = arr/2;
    g_timx_oc_pwm_chy_handle.OCpolarity = TIM_OCPOLARITY_LOW; /* 输出比较极性为低 */
    HAL_TIM_PWM_ConfigChannel(&g_timx_pwm_chy_handle, &g_timx_oc_pwm_chy_handle,
                             GTIM_TIMX_PWM_CHY); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_Start(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY); /* 开启 PWM 通道 */
}

```

`HAL_TIM_PWM_Init` 初始化 TIM3 并设置 TIM3 的 ARR 和 PSC 等参数，其次通过调用函数 `HAL_TIM_PWM_ConfigChannel` 设置 TIM3_CH2 的 PWM 模式以及比较值等参数，最后通过调用函数 `HAL_TIM_PWM_Start` 来使能 TIM3 以及使能 PWM 通道 TIM3_CH2 输出。

本实验我们使用 PWM 的 MSP 初始化回调函数 `HAL_TIM_PWM_MspInit` 来存放时钟、GPIO 的初始化代码，其定义如下：

```

/**
 * @brief      定时器底层驱动，时钟使能，引脚配置
 *              此函数会被 HAL_TIM_PWM_Init() 调用
 * @param      htim:定时器句柄
 * @retval     无
 */
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == GTIM_TIMX_PWM)
    {
        GPIO_InitTypeDef gpio_init_struct;
        GTIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE(); /* 开启通道 y 的 GPIO 时钟 */
        GTIM_TIMX_PWM_CHY_CLK_ENABLE();

        gpio_init_struct.Pin = GTIM_TIMX_PWM_CHY_GPIO_PIN; /* 通道 y 的 GPIO 口 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* 复用推挽输出 */
    }
}

```



```

gpio_init_struct.Pull = GPIO_PULLUP;          /* 上拉 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
HAL_GPIO_Init(GTIM_TIMX_PWM_CHY_GPIO_PORT, &gpio_init_struct);
GTIM_TIMX_PWM_CHY_GPIO_REMAP();              /* IO 口 REMAP 设置, 设置重映射 */
}
}

```

该函数首先判断定时器寄存器基地址, 符合条件后, 开启对应的 GPIO 时钟和定时器时钟, 并且初始化 GPIO。上面是使用 HAL 库标准的做法, 我们亦可把 HAL_TIM_PWM_MspInit 函数里面的代码直接放到 gtim_timx_pwm_chy_init 函数里。这样做的好处是当一个项目中用到多个定时器时, 代码的移植性、可读性好, 方便管理。

在 main.c 里面编写如下代码:

```

int main(void)
{
    uint16_t ledpwmval = 0;
    uint8_t dir = 1;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    /* 72M/72=1M 的计数频率, 自动重载为 500, 那么 PWM 频率为 1M/500=2kHz */
    gtim_timx_pwm_chy_init(500 - 1, 72 - 1);

    while (1)
    {
        delay_ms(10);

        if (dir) ledpwmval++; /* dir==1 ledpwmval 递增 */
        else ledpwmval--; /* dir==0 ledpwmval 递减 */

        if (ledpwmval > 300) dir = 0; /* ledpwmval 到达 300 后, 方向为递减 */
        if (ledpwmval == 0) dir = 1; /* ledpwmval 递减到 0 后, 方向改为递增 */

        /* 修改比较值控制占空比 */
        __HAL_TIM_SET_COMPARE(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY,
                               ledpwmval);
    }
}

```

本小节开头我们就说 PWM 波频率由自动重载寄存器 (TIMx_ARR) 的值决定, 其占空比则由捕获/比较寄存器 (TIMx_CCRx) 的值决定。下面结合实际看看具体怎么计算:

定时器 3 的时钟源频率为 2 倍 APB1 总线时钟频率, 即频率为 72MHz, 而调用 gtim_timx_pwm_chy_init 初始化函数之后, 就相当于写入预分频寄存器的值为 71, 写入自动重载寄存器的值为 499。基本定时器讲的定时器溢出公式由公式得:

$$T_{out} = ((arr+1)*(psc+1))/T_{clk} = ((499+1)*(71+1))/72000000 = 0.0005s$$

再由频率是周期的倒数关系得到 PWM 的频率为 2000Hz。

占空比怎么计算的呢? 结合图 21.3.1, 我们分两种情况分析, 输出比较极性为低和输出比较极性为高, 它们的情况正好相反。因为在 main 函数中的比较值是动态变化的, 不利于我们计算占空比, 我们假设比较值固定为 200, 在本实验中可以调用如下语句得到。

```
__HAL_TIM_SET_COMPARE(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY, 200);
```

因为 LED0 是低电平有效, 所以我们在 gtim_timx_pwm_chy_init 函数中设置了输出比较极性为低, 那么当比较值固定为 200 时, 占空比 = $((arr+1) - CCR1) / (arr+1) = (500-200)/500 = 60\%$ 。其中 arr 是写入自动重载寄存器 (TIMx_ARR) 的值, CCR2 就是写入捕获/比较寄存器 2 (TIMx_CCR2) 的值。这里我们还需要提醒一下, 占空比是指在一个周期内, 高电平时间相对于总时间所占的比例。

另外一种情况: 设置了输出比较极性为高, 那么当比较值固定为 200 时, 占空比 = $CCR2 / (arr+1) = 200/500 = 40\%$ 。可以看到输出比较极性为低和输出比较极性为高的占空比正好反过来。

在这里, 我们也用了 DS100 示波器进行验证, 效果图如图 21.3.3.1 所示:

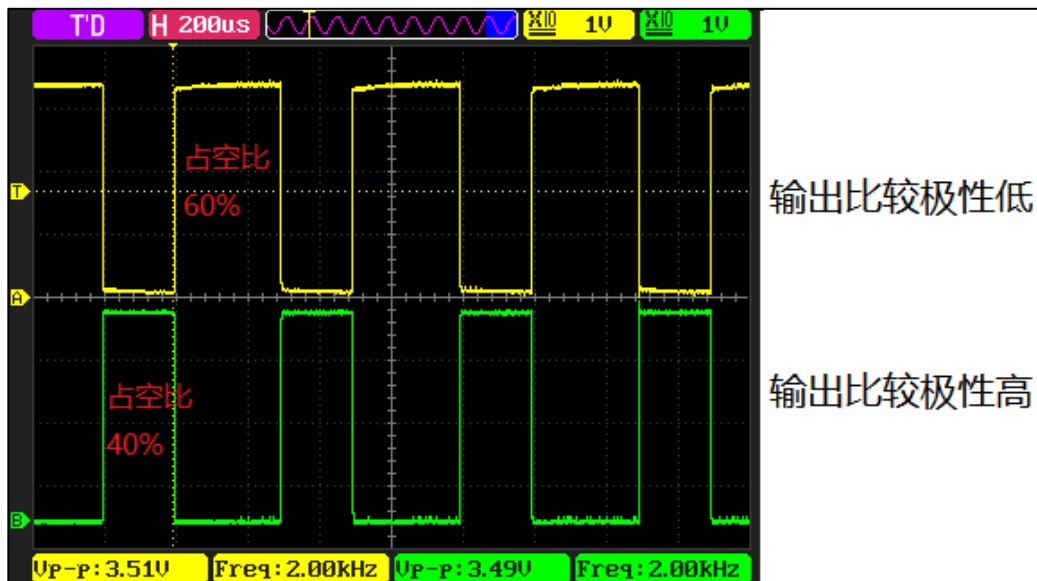


图 21.3.3.3.1 验证效果图

这里把输出比较极性低和输出比较极性高的 PWM 波形都显示出来了。本实验默认设置 PWM 模式 1、输出比较极性低，当 CCR2 寄存器的值设置为 200 时，对应的 PWM 波形如上图黄色的波形图。如果把输出比较极性设置为高，对应的波形图就是绿色的波形图了。

大家感兴趣也可以自行用示波器进行验证。

21.3.4 下载验证

下载代码后，此时定时器 3 通道 2 输出 PWM 信号到 PB5 口。可以看到 LED0 不停的由暗变到亮，然后又从亮变到暗。

21.4 通用定时器输入捕获实验

本小节我们来学习使用通用定时器的输入捕获模式。

输入捕获模式可以用来测量脉冲宽度或者测量频率。我们以测量脉宽为例，用一个简图来说明输入捕获脉宽测量原理，如图 21.4.1 所示：

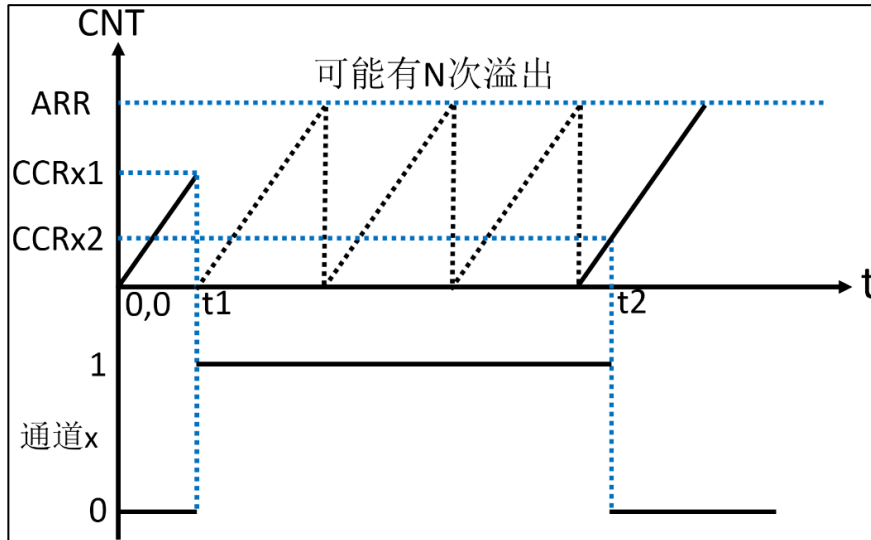


图 21.4.1 输入捕获脉宽测量原理

图 21.4.1 中， $t1$ 到 $t2$ 的时间段，就是我们需要测量的高电平时间。测量方法如下：假如定时器工作在递增计数模式，首先设置定时器通道 x 为上升沿捕获，这样在 $t1$ 时刻上升沿到来时，就会发生捕获事件。这里我们还会打开捕获中断，所以捕获事件发生就意味着捕获中断也会发生。在捕获中断里将计数器值清零，并设置通道 x 为下降沿捕获，这样 $t2$ 时刻下降沿到来时，就会发生捕获事件和捕获中断。捕获事件发生时，计数器的值会被锁存到捕获/比较寄存器中（比如通道 1 对应的是 $CCR1$ 寄存器）。那么在捕获中断里，我们读取捕获/比较寄存器就可以获取到高电平脉冲时间内，计数器计数的个数，从而可以算出高电平脉冲的时间。这里是假设定时器没有溢出为前提的。

实际上， $t1$ 到 $t2$ 时间段，定时器可能会产生 N 次溢出，这就需要对定时器溢出做相应的处理，防止高电平太长，导致测量出错。在 $t1$ 到 $t2$ 时间段，假设定时器溢出 N 次，那么高电平脉冲时间内，计数器计数的个数计算方法为： $N * (ARR + 1) + CCRx2$ ， $CCRx2$ 表示 $t2$ 时间点，捕获/比较寄存器的值。经过计算得到高电平脉宽时间内计数器计数个数后，用这个个数乘以计数器的计数周期，就可得到高电平持续的时间。就是输入捕获测量高电平脉宽时间的整个过程。

STM32F103 的定时器除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。输入捕获，简单的说就是通过检测 $TIMx_CHy$ 上的边沿信号，在边沿信号发生跳变（比如上升沿/下降沿）时，会发生捕获事件，将当前定时器的值（ $TIMx_CNT$ ）锁存到对应通道的捕获/比较寄存器（ $TIMx_CCRx$ ）里，完成一次捕获。同时还可以配置捕获事件发生时是否触发捕获中断/DMA。另外还要考虑测量的过程中是否可能发生定时器溢出，如果可能溢出，还要做溢出处理。

21.4.1 TIM2/TIM3/TIM4/TIM5 寄存器

通用定时器输入捕获实验需要用到的寄存器有： $TIMx_ARR$ 、 $TIMx_PSC$ 、 $TIMx_CCMR1$ 、 $TIMx_CCER$ 、 $TIMx_DIER$ 、 $TIMx_CR1$ 、 $TIMx_CCR1$ 这些寄存器在前面的章节都有提到，在这里只需针对性的介绍。

● 捕获/比较模式寄存器 1/2（ $TIMx_CCMR1/2$ ）

该寄存器我们在 PWM 输出实验时讲解了它作为输出功能的配置，现在重点学习输入捕获模式的配置。因为本实验我们用到定时器 5 通道 1 输入，所以我们要看 $TIMx_CCMR1$ 寄存器，其描述如图 21.4.1.1 所示：

图 21.4.1.1 TIMx CCMR1 寄存器

TIM2/TIM3/TIM4/TIM5 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx CCER 寄存器描述如图 21.4.1.2 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|--|------|----|----|------|------|----|---|------|------|----|---|------|------|
| 保留 | | CC4P | CC4E | 保留 | | CC3P | CC3E | 保留 | | CC2P | CC2E | 保留 | | CC1P | CC1E |
| rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | rW |
| 位1 | | CC1P : 输入/捕获1输出极性 (Capture/Compare 1 output polarity) CC1通道配置为输出 : 0: OC1高电平有效 | | | | | | | | | | | | | |

图 21.4.1.2 TIMx_CCER 寄存器

我们要用到这个寄存器的最低2位,CC1E和CC1P位。要使能输入捕获,必须设置CC1E=1,而CC1P则根据自己的需要来配置。我们这里是保留默认设置值0,即高电平触发捕获。

接下来我们再看看DMA/中断使能寄存器:TIMx_DIER,该寄存器的各位描述见图21.2.1.3(在21.2.1小节)。本小节,我们需要用到中断来处理捕获数据,所以必须开启通道1的捕获比较中断,即CC1IE设置为1。同时我们还需要在定时器溢出中断中累计定时器溢出的次数,所以还需要使能定时器的更新中断,即UIE置1。

控制寄存器:TIMx_CR1,我们只用到了它的最低位,也就是用来使能定时器的。

最后再来看看捕获/比较寄存器1:TIMx_CCR1,该寄存器用来存储发生捕获事件时,TIMx_CNT的值,我们从TIMx_CCR1就可以读出通道1捕获事件发生时刻的TIMx_CNT值,通过两次捕获(一次上升沿捕获,一次下降沿捕获)的差值,就可以计算出高电平脉冲的宽度(注意,对于高电平脉宽太长的情况,还要计算定时器溢出的次数)。

21.4.2 硬件设计

1. 例程功能

1、使用TIM5_CH1来做输入捕获,捕获PA0上的高电平脉宽,并将脉宽时间通过串口打印出来,然后通过按WK_UP按键,模拟输入高电平,例程中能测试的最长高电平脉宽时间为:4194303 us。

2、LED0闪烁指示程序运行。

2. 硬件资源

1) LED灯:

LED0—PB5

2) 独立按键:

WK_UP-PA0

3) 定时器5,使用TIM5通道1,将PA0复用为TIM5_CH1。

3. 原理图

定时器属于STM32F103的内部资源,只需要软件设置好即可正常工作。我们借助WK_UP做输入脉冲源并通过串口上位机来监测定时器输入捕获的情况。

21.4.3 程序设计

21.4.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_IC_Init 函数

定时器的输入捕获模式初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**
用于初始化定时器的输入捕获模式。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，介绍基本定时器的时候已经介绍。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**

与 PWM 输出实验一样，当使用定时器做输入捕获功能时，在 HAL 库中并不使用定时器初始化函数 HAL_TIM_Base_Init 来实现，而是使用输入捕获特定的定时器初始化函数 HAL_TIM_IC_Init。该函数内部还会调用输入捕获初始化回调函数 HAL_TIM_IC_MspInit 来初始化输入通道对应的 GPIO（复用），以及输入捕获相关的配置。

2. HAL_TIM_IC_ConfigChannel 函数

定时器的输入捕获通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim,
                                           TIM_IC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**
该函数用于设置定时器的输入捕获通道。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
形参 2 是 TIM_IC_InitTypeDef 结构体类型指针变量，用于配置定时器的输入捕获参数。
重点了解一下 TIM_IC_InitTypeDef 结构体指针类型，其定义如下：

```
typedef struct
{
    uint32_t ICPolarity; /* 输入捕获触发方式选择，比如上升、下降和双边沿捕获 */
    uint32_t ICSelection; /* 输入捕获选择，用于设置映射关系 */
    uint32_t ICPrescaler; /* 输入捕获分频系数 */
    uint32_t ICFilter; /* 输入捕获滤波器设置 */
} TIM_IC_InitTypeDef;
```

该结构体成员我们现在介绍一下。成员变量 ICPolarity 用来设置输入信号的有效捕获极性，取值范围为：TIM_ICPOLARITY_RISING（上升沿捕获），TIM_ICPOLARITY_FALLING（下降沿捕获）和 TIM_ICPOLARITY_BOTHEDGE（双边沿捕获）。成员变量 ICSelection 用来设置映射关系，我们配置 IC1 直接映射在 TI1 上，选择 TIM_ICSELECTION_DIRECTTI（另外还有两个输入通道 TIM_ICSELECTION_INDIRECTTI 和 TIM_ICSELECTION_TRC）。成员变量 ICPrescaler 用来设置输入捕获分频系数，可以设置为 TIM_ICPSC_DIV1（不分频），TIM_ICPSC_DIV2（2 分频），TIM_ICPSC_DIV4（4 分频）以及 TIM_ICPSC_DIV8（8 分频），本实验需要设置为不分频，所以选值为 TIM_ICPSC_DIV1。成员变量 ICFilter 用来设置滤波器长度，这里我们不使用滤波器，所以设置为 0。

形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_IC_Start_IT 函数

启动定时器输入捕获模式函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_Start_IT(TIM_HandleTypeDef *htim,
```

```
uint32_t Channel);
```

- **函数描述:**
用于启动定时器的输入捕获模式，且开启输入捕获中断。
- **函数形参:**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。
形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。
- **注意事项:**
如果我们不需要开启输入捕获中断，只是开启输入捕获功能，HAL 库函数为：

```
HAL_StatusTypeDef HAL_TIM_IC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

定时器输入捕获模式配置步骤

1) 开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 5 通道 1，对应 IO 是 PA0，它们的时钟开启方法如下：

```
HAL_RCC_TIM5_CLK_ENABLE();           /* 使能定时器 5 */
HAL_RCC_GPIOA_CLK_ENABLE();          /* 开启 GPIOA 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输入捕获功能时，我们调用的是 HAL_TIM_IC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3) 设置 TIMx_CHy 的输入捕获模式，开启输入捕获

在 HAL 库中，定时器的输入捕获模式是通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器某个通道为输入捕获通道，包括映射关系，输入滤波和输入分频等。

4) 使能定时器更新中断，开启捕获功能以及捕获中断，配置定时器中断优先级

通过 __HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start_IT 函数使能定时器并开启捕获功能以及捕获中断。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

因为我们要捕获的是高电平信号的脉宽，所以，第一次捕获是上升沿，第二次捕获时下降沿，必须在捕获上升沿之后，设置捕获边沿为下降沿，同时，如果脉宽比较长，那么定时器就会溢出，对溢出必须做处理，否则结果就不准了。

5) 编写中断服务函数

定时器 5 中断服务函数为：TIM5_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个定时器中断通用处理函数 HAL_TIM_IRQHandler，该函数会调用一些定时器相关的回调函数，用于给用户处理定时器中断到了之后，需要处理的程序。本实验我们除了用到更新（溢出）中断回调函数 HAL_TIM_PeriodElapsedCallback 之外，还要用到捕获中断回调函数 HAL_TIM_IC_CaptureCallback。详见本实验例程源码。

21.4.3.2 程序流程图

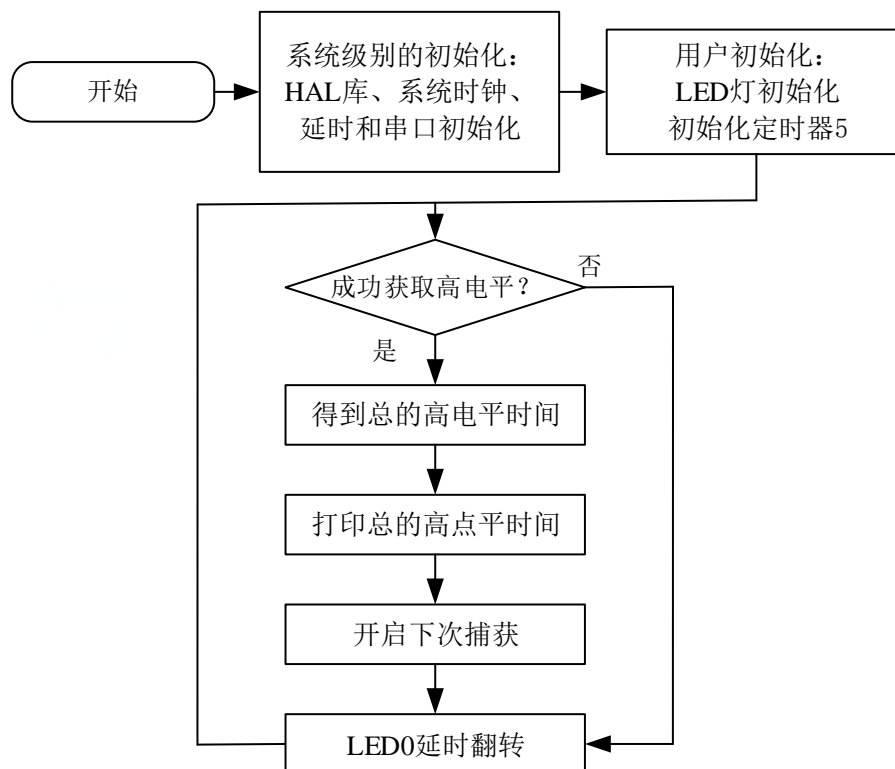


图 21.4.3.2.1 通用定时器输入捕获实验程序流程图

21.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。通用定时器驱动源码包括两个文件：`gtim.c` 和 `gtim.h`。

首先看 `gtim.h` 头文件的几个宏定义：

```

/* TIMX 输入捕获定义
 * 这里的输入捕获使用定时器 TIM5_CH1, 捕获 WK_UP 按键的输入
 * 默认是针对 TIM2~TIM5.
 * 注意：通过修改这几个宏定义, 可以支持 TIM1~TIM8 任意一个定时器, 任意一个 IO 口做输入捕获
 * 特别要注意: 默认用的 PA0, 设置的是下拉输入! 如果改其他 IO, 对应的上下拉方式也得改!
 */
#define GTIM_TIMX_CAP_CHY_GPIO_PORT      GPIOA
#define GTIM_TIMX_CAP_CHY_GPIO_PIN      GPIO_PIN_0
#define GTIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE();\
}while(0) /* PA 口时钟使能 */

#define GTIM_TIMX_CAP      TIM5
#define GTIM_TIMX_CAP_IRQn TIM5_IRQn
#define GTIM_TIMX_CAP_IRQHandler TIM5_IRQHandler
#define GTIM_TIMX_CAP_CHY TIM_CHANNEL_1
/* 通道 Y, 1<= Y <=4 */
#define GTIM_TIMX_CAP_CHY_CCRX      TIM5->CCR1
/* 通道 Y 的输出比较寄存器 */
#define GTIM_TIMX_CAP_CHY_CLK_ENABLE() do{ __HAL_RCC_TIM5_CLK_ENABLE();\
}while(0) /* TIM5 时钟使能 */
    
```

可以把上面的宏定义分成两部分，第一部分是定时器 5 输入通道 1 对应的 IO 口的宏定义，第二部分则是定时器 5 输入通道 1 的相应宏定义。

下面看 `gtim.c` 的程序，首先是通用定时器输入捕获初始化函数。

```

/**
 * @brief      通用定时器 TIMX 通道 Y 输入捕获 初始化函数
    
```

```

* @note
*      通用定时器的时钟来自 APB1, 当 PPRE1 ≥ 2 分频的时候
*      通用定时器的时钟为 APB1 时钟的 2 倍, 而 APB1 为 36M, 所以定时器时钟 = 72Mhz
*      定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
*      Ft=定时器工作频率, 单位:Mhz
*
* @param      arr: 自动重装值
* @param      psc: 时钟预分频数
* @retval      无
*/
void gtim_timx_cap_chy_init(uint16_t arr, uint16_t psc)
{
    TIM_IC_InitTypeDef timx_ic_cap_chy = {0};

    g_timx_cap_chy_handle.Instance = GTIM_TIMX_CAP;           /* 定时器 5 */
    g_timx_cap_chy_handle.Init.Prescaler = psc;               /* 定时器分频 */
    g_timx_cap_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数模式 */
    g_timx_cap_chy_handle.Init.Period = arr;                  /* 自动重装载值 */
    HAL_TIM_IC_Init(&g_timx_cap_chy_handle);

    timx_ic_cap_chy.ICPolarity = TIM_ICPOLARITY_RISING;       /* 上升沿捕获 */
    timx_ic_cap_chy.ICSelection = TIM_ICSELECTION_DIRECTTI;   /* 映射到 TI1 上 */
    timx_ic_cap_chy.ICPrescaler = TIM_ICPSC_DIV1;             /* 配置输入分频, 不分频 */
    timx_ic_cap_chy.ICFilter = 0;                             /* 配置输入滤波器, 不滤波 */
    HAL_TIM_IC_ConfigChannel(&g_timx_cap_chy_handle, &timx_ic_cap_chy,
                             GTIM_TIMX_CAP_CHY); /* 配置 TIM5 通道 1 */

    HAL_TIM_ENABLE_IT(&g_timx_cap_chy_handle, TIM_IT_UPDATE); /* 使能更新中断 */
    /* 使能通道输入以及使能捕获中断 */
    HAL_TIM_IC_Start_IT(&g_timx_cap_chy_handle, GTIM_TIMX_CAP_CHY);
}

```

HAL_TIM_IC_Init 初始化定时器的基础工作参数, 如: ARR 和 PSC 等, 第二部分则是调用 HAL_TIM_IC_ConfigChannel 函数配置输入捕获通道映射关系, 滤波和分频等。最后是使能更新中断和使能通道输入以及定时器捕获中断。通道对应的 IO、时钟开启和 NVIC 的初始化都在 HAL_TIM_IC_MspInit 函数里编写, 其定义如下:

```

/**
* @brief      通用定时器输入捕获初始化接口
*              HAL 库调用的接口, 用于配置不同的输入捕获
* @param      htim: 定时器句柄
* @note      此函数会被 HAL_TIM_IC_Init() 调用
* @retval      无
*/
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == GTIM_TIMX_CAP) /* 输入通道捕获 */
    {
        GPIO_InitTypeDef gpio_init_struct;
        GTIM_TIMX_CAP_CHY_CLK_ENABLE(); /* 使能 TIMx 时钟 */
        GTIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE(); /* 开启捕获 IO 的时钟 */

        gpio_init_struct.Pin = GTIM_TIMX_CAP_CHY_GPIO_PIN; /* 输入捕获的 GPIO 口 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLDOWN; /* 下拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
        HAL_GPIO_Init(GTIM_TIMX_CAP_CHY_GPIO_PORT, &gpio_init_struct);

        HAL_NVIC_SetPriority(GTIM_TIMX_CAP_IRQn, 1, 3); /* 抢占 1, 子优先级 3 */
        HAL_NVIC_EnableIRQ(GTIM_TIMX_CAP_IRQn); /* 开启 ITMx 中断 */
    }
}

```

该函数调用 HAL_GPIO_Init 函数初始化定时器输入通道对应的 IO，并且开启 GPIO 时钟，使能定时器。其中要注意 IO 口复用功能的选择一定要选对了。最后配置中断抢占优先级和响应优先级，以及打开定时器中断。

通过上面的两个函数输入捕获的初始化就完成了，下面先来介绍两个变量。

```
/* 输入捕获状态(g_timxchy_cap_sta)
 * [7] :0,没有成功的捕获;1,成功捕获到一次.
 * [6] :0,还没捕获到高电平;1,已经捕获到高电平了.
 * [5:0]:捕获高电平后溢出的次数,最多溢出 63 次,所以最长捕获值= 63*65536+65535 = 4194303
 * 注意:为了通用,我们默认 ARR 和 CCRy 都是 16 位寄存器,对于 32 位的定时器(如:TIM5),
 * 也只按 16 位使用
 * 按 1us 的计数频率,最长溢出时间为:4194303 us, 约 4.19 秒
 * (说明一下:正常 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
 */
uint8_t g_timxchy_cap_sta = 0; /* 输入捕获状态 */
uint16_t g_timxchy_cap_val = 0; /* 输入捕获值 */
```

这两个变量用于辅助实现高电平捕获。其中 g_timxchy_cap_sta，是用来记录捕获状态，（这个变量，我们把它当成一个寄存器那样来使用）。对其各位赋予状态含义，描述如下表所示：

| g_timxchy_cap_sta | | |
|-------------------|----------|----------------|
| bit7 | bit6 | bit5~0 |
| 捕获完成标志 | 捕获到高电平标志 | 捕获高电平后定时器溢出的次数 |

表 21.4.3.3.1 g_timxchy_cap_sta 各位描述

变量 g_timxchy_cap_sta 的位[5:0]是用于记录捕获高电平时器溢出次数，总共 6 位，所以最多可以记录溢出的次数为 2 的 6 次方减一次，即 63 次。

变量 g_timxchy_cap_val，则用来记录捕获到下降沿的时候，TIM5_CNT 寄存器的值。

下面开始看中断服务函数的逻辑程序，HAL_TIM_IRQHandler 函数会调用下面两个回调函数，我们的逻辑代码就是放在回调函数里，函数定义如下：

```
/**
 * @brief      定时器输入捕获中断处理回调函数
 * @param      htim:定时器句柄指针
 * @note       该函数在 HAL_TIM_IRQHandler 中会被调用
 * @retval     无
 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if ((g_timxchy_cap_sta & 0X80) == 0) /* 还没成功捕获 */
    {
        if (g_timxchy_cap_sta & 0X40) /* 捕获到一个下降沿 */
        {
            g_timxchy_cap_sta |= 0X80; /* 标记成功捕获到一次高电平脉宽 */
            g_timxchy_cap_val = HAL_TIM_ReadCapturedValue(&g_timx_cap_chy_handler,
                GTIM_TIMX_CAP_CHY); /* 获取当前的捕获值 */
            TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handler,
                GTIM_TIMX_CAP_CHY); /* 一定要先清除原来的设置 */
            TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handler, GTIM_TIMX_CAP_CHY,
                TIM_ICPOLARITY_RISING); /* 配置 TIM5 通道 1 上升沿捕获 */
        }
        else /* 还未开始,第一次捕获上升沿 */
        {
            g_timxchy_cap_sta = 0; /* 清空 */
            g_timxchy_cap_val = 0;
            g_timxchy_cap_sta |= 0X40; /* 标记捕获到了上升沿 */
            __HAL_TIM_DISABLE(&g_timx_cap_chy_handler); /* 关闭定时器 5 */
            __HAL_TIM_SET_COUNTER(&g_timx_cap_chy_handler,0); /* 计数器清零 */
            TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handler,
                GTIM_TIMX_CAP_CHY); /* 一定要先清除原来的设置!! */
            TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handler, GTIM_TIMX_CAP_CHY,
```



```

        TIM_ICPOLARITY_FALLING); /* 定时器 5 通道 1 设置为下降沿捕获 */
    __HAL_TIM_ENABLE(&g_timx_cap_chy_handler); /* 使能定时器 5 */
}
}

/**
 * @brief      定时器更新中断回调函数
 * @param      htim:定时器句柄指针
 * @note       此函数会被定时器中断函数共同调用的
 * @retval     无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == GTIM_TIMX_CAP)
    {
        if ((g_timxchy_cap_sta & 0X80) == 0) /* 还没成功捕获 */
        {
            if (g_timxchy_cap_sta & 0X40) /* 已经捕获到高电平了 */
            {
                if ((g_timxchy_cap_sta & 0X3F) == 0X3F) /* 高电平太长了 */
                {
                    TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handle,
                                                GTIM_TIMX_CAP_CHY); /* 一定要先清除原来的设置 */
                    /* 配置 TIM5 通道 1 上升沿捕获 */
                    TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handle,
                                                GTIM_TIMX_CAP_CHY, TIM_ICPOLARITY_RISING);
                    g_timxchy_cap_sta |= 0X80; /* 标记成功捕获了一次 */
                    g_timxchy_cap_val = 0XFFFF;
                }
                else /* 累计定时器溢出次数 */
                {
                    g_timxchy_cap_sta++;
                }
            }
        }
    }
}
}

```

现在我们来介绍一下，捕获高电平脉宽的思路：首先，设置 TIM5_CH1 捕获上升沿，然后等待上升沿中断到来，当捕获到上升沿中断，此时如果 g_timxchy_cap_sta 的第 6 位为 0，则表示还没有捕获到新的上升沿，就先把 g_timxchy_cap_sta、g_timxchy_cap_val 和 TIM5_CNT 寄存器等清零，然后再设置 g_timxchy_cap_sta 的第 6 位为 1，标记捕获到高电平，最后设置为下降沿捕获，等待下降沿到来。如果等待下降沿到来期间，定时器发生了溢出，就用 g_timxchy_cap_sta 变量对溢出次数进行计数，当最大溢出次数来到的时候，就强制标记捕获完成，并配置定时器通道上升沿捕获。当下降沿到来的时候，先设置 g_timxchy_cap_sta 的第 7 位为 1，标记成功捕获一次高电平，然后读取此时的定时器值到 g_timxchy_cap_val 里面，最后设置为上升沿捕获，回到初始状态。

这样，我们就完成一次高电平捕获了，只要 g_timxchy_cap_sta 的第 7 位一直为 1，那么就不会进行第二次捕获，我们在 main 函数处理完捕获数据后，将 g_timxchy_cap_sta 置零，就可以开启第二次捕获。

在 main.c 里面编写如下代码：

```

int main(void)
{
    uint32_t temp = 0;
    uint8_t t = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
}

```

```

led_init(); /* 初始化 LED */
key_init(); /* 初始化按键 */
gtim_timx_cap_chy_init(0xFFFF, 72 - 1); /* 以 1Mhz 的频率计数 捕获 */
while (1)
{
    if (g_timxchy_cap_sta & 0X80) /* 成功捕获到了一次高电平 */
    {
        temp = g_timxchy_cap_sta & 0X3F;
        temp *= 65536; /* 溢出时间总和 */
        temp += g_timxchy_cap_val; /* 得到总的高电平时间 */
        printf("HIGH:%d us\r\n", temp); /* 打印总的高点平时间 */
        g_timxchy_cap_sta = 0; /* 开启下一次捕获 */
    }
    t++;
    if (t > 20) /* 200ms 进入一次 */
    {
        t = 0;
        LED0_TOGGLE(); /* LED0 闪烁 ,提示程序运行 */
    }
    delay_ms(10);
}
}

```

先看 `gtim_timx_cap_chy_init(0xFFFF, 72 - 1)` 这个语句，这两个形参分别设置自动重载寄存器的值为 65535，以及预分频器寄存器的值为 71。定时器 5 是 16 位的计数器，这里设置为最大值 65535。预分频系数，我们设置为 72 分频，定时器 5 的时钟频率是 2 倍的 APB1 总线时钟频率，即 72MHz，可以得到计数器的计数频率是 1MHz，即 1us 计数一次，所以我们的捕获时间精度是 1us。这里可以知道定时器的溢出时间是 65536us。

While (1) 无限循环通过判断 `g_timxchy_cap_sta` 的第 7 位，来获知有没有成功捕获到一次高电平，如果成功捕获，先计算总的高电平时间，再通过串口传输到电脑。

21.4.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口端口，我这边是 COM15，然后按 KEY_UP 按键，可以看到串口打印的高电平持续时间，如图 21.4.4.1 所示：



图 21.4.4.1 打印捕获到的高电平时间

21.5 通用定时器脉冲计数实验

前面我们介绍了通用定时器的四类时钟源，本小节我们来学习使用通用定时器的外部时钟模式 1 这类时钟源。

前面的三个通用定时器实验的时钟源都是来自内部时钟 (CK_INT)，本实验我们将使用外部时钟模式 1：外部输入引脚 (TIx)作为定时器的时钟源。关于这个外部输入引脚(TIx)，我们使用 WK_UP 按键按下产生的高电平脉冲作为定时器的计数器时钟，每按下一次按键产生一次高电平脉冲，计数器加一。

下面通过框图给大家展示本实验用到定时器内部哪些资源，如下图所示：

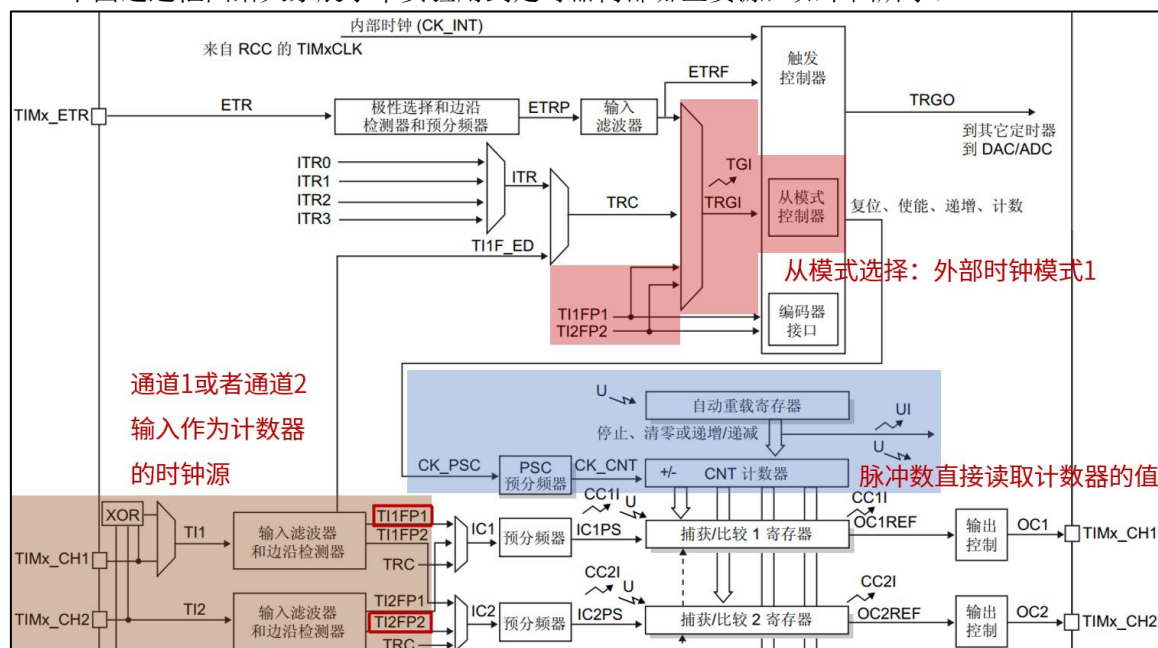


图 21.5.1 脉冲计数实验原理

前面介绍过，外部时钟模式 1 的外部输入引脚只能是通道 1 或者通道 2 对应的 IO，通道 3 或者通道 4 是不可以的。以通道 1 输入为例，外部时钟源信号通过通道 1 输入后，接下来我们用 TI1 表示该信号。TI1 分别要经过滤波器、边沿检测器后，来到 TI1FP1，被触发输入选择器选择为触发源，接着来到从模式控制器。从模式选择为外部时钟模式 1，这时候外部时钟源信号就会到达时基单元的预分频器，后面就是经过分频后就作为计数器的计数时钟了。这个过程描述，大家亦可参考前面介绍外部时钟模式 1 的描述。因为前面已经介绍过，这里只是简单讲一下。

如果大家想时钟源信号的上升沿和下降沿，计数器都计数，可以选择 TIF_ED 作为触发输入选择器的触发源。

假设计数器工作在递增计数模式，那么每来一个选择的边沿，计数器就加一。最后，外部时钟源信号的边沿计数个数会保存计数器寄存器中，我们只需要直接读取 CNT 的值即可。这里是没有考虑定时器溢出的情况，如果定时器溢出还需要对溢出进行处理。比如开启更新中断，定时器溢出后，在更新中断里，对溢出次数进行记录，然后用溢出次数乘以溢出一次计数的个数，再加上 CNT 现在的值，就可以得到总的计数个数了。在例程中，我们也是这样处理的。

下面开始讲解本实验用到的寄存器配置情况。

21.5.1 TIM2/TIM3/TIM4/TIM5 寄存器

通用定时器脉冲计数实验需要用到的寄存器有：TIMx_ARR、TIMx_PSC、TIMx_CCMR1、TIMx_CCER、TIMx_DIER、TIMx_CR1、TIMx_EGR 这些寄存器在前面的章节都有提到，在这里只需针对性的介绍。

● 捕获/比较模式寄存器 1/2 (TIMx_CCMR1/2)

该寄存器我们在 PWM 输出实验时讲解了他作为输出功能的配置，在输入捕获实验学习了

输入捕获模式的配置，本小节我们的外部信号也同样要作为输入信号给定时器作为时钟源，所以我们要看输入捕获模式定时器对应功能。WK_UP 按键（PA0）对应着定时器 2 的通道 1，这个可以在数据手册《STM32F103ZET6（中文版）.pdf》21 页找到。接下来我们开始配置 TIMx_CCMR1 寄存器，其描述如图 21.5.1.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | | | | | | | | | |
|---|--|----|----|-------------|-------|-----------|----|-----------|-----------|----|-------------|-------|-------|-----------|----|---------------------------------|---|--|---|--|--|--|--|---|--|---|--|---|--|---|--|
| OC2CE | OC2M[2:0] | | | OC2PE | OC2FE | CC2S[1:0] | | OC1CE | OC1M[2:0] | | | OC1PE | OC1FE | CC1S[1:0] | | | | | | | | | | | | | | | | | |
| IC2F[3:0] | | | | IC2PSC[1:0] | | | | IC1F[3:0] | | | IC1PSC[1:0] | | | | | | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | | | | | | | | | | | | | | | | |
| <div><div>位7:4</div><div>IC1F[3:0]: 输入捕获1滤波器 (Input capture 1 filter) 这几位定义了TI1输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，它记录到N个事件后会产生一个输出的跳变： <table><tr><td>0000: 无滤波器，以f_{DTS}采样</td><td>1000: 采样频率f_{SAMPLING}=f_{DTS}/8, N=6</td></tr><tr><td>0001: 采样频率f_{SAMPLING}=f_{CK_INT}, N=2</td><td>1001: 采样频率f_{SAMPLING}=f_{DTS}/8, N=8</td></tr><tr><td>0010: 采样频率f_{SAMPLING}=f_{CK_INT}, N=4</td><td>1010: 采样频率f_{SAMPLING}=f_{DTS}/16, N=5</td></tr><tr><td>0011: 采样频率f_{SAMPLING}=f_{CK_INT}, N=8</td><td>1011: 采样频率f_{SAMPLING}=f_{DTS}/16, N=6</td></tr><tr><td>0100: 采样频率f_{SAMPLING}=f_{DTS}/2, N=6</td><td>1100: 采样频率f_{SAMPLING}=f_{DTS}/16, N=8</td></tr><tr><td>0101: 采样频率f_{SAMPLING}=f_{DTS}/2, N=8</td><td>1101: 采样频率f_{SAMPLING}=f_{DTS}/32, N=5</td></tr><tr><td>0110: 采样频率f_{SAMPLING}=f_{DTS}/4, N=6</td><td>1110: 采样频率f_{SAMPLING}=f_{DTS}/32, N=6</td></tr><tr><td>0111: 采样频率f_{SAMPLING}=f_{DTS}/4, N=8</td><td>1111: 采样频率f_{SAMPLING}=f_{DTS}/32, N=8</td></tr></table>注：在现在的芯片版本中，当ICx F[3:0]=1、2或3时，公式中的f_{DTS}由CK_INT替代。</div></div> | | | | | | | | | | | | | | | | 0000: 无滤波器，以f _{DTS} 采样 | 1000: 采样频率f _{SAMPLING} =f _{DTS} /8, N=6 | 0001: 采样频率f _{SAMPLING} =f _{CK_INT} , N=2 | 1001: 采样频率f _{SAMPLING} =f _{DTS} /8, N=8 | 0010: 采样频率f _{SAMPLING} =f _{CK_INT} , N=4 | 1010: 采样频率f _{SAMPLING} =f _{DTS} /16, N=5 | 0011: 采样频率f _{SAMPLING} =f _{CK_INT} , N=8 | 1011: 采样频率f _{SAMPLING} =f _{DTS} /16, N=6 | 0100: 采样频率f _{SAMPLING} =f _{DTS} /2, N=6 | 1100: 采样频率f _{SAMPLING} =f _{DTS} /16, N=8 | 0101: 采样频率f _{SAMPLING} =f _{DTS} /2, N=8 | 1101: 采样频率f _{SAMPLING} =f _{DTS} /32, N=5 | 0110: 采样频率f _{SAMPLING} =f _{DTS} /4, N=6 | 1110: 采样频率f _{SAMPLING} =f _{DTS} /32, N=6 | 0111: 采样频率f _{SAMPLING} =f _{DTS} /4, N=8 | 1111: 采样频率f _{SAMPLING} =f _{DTS} /32, N=8 |
| 0000: 无滤波器，以f _{DTS} 采样 | 1000: 采样频率f _{SAMPLING} =f _{DTS} /8, N=6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0001: 采样频率f _{SAMPLING} =f _{CK_INT} , N=2 | 1001: 采样频率f _{SAMPLING} =f _{DTS} /8, N=8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0010: 采样频率f _{SAMPLING} =f _{CK_INT} , N=4 | 1010: 采样频率f _{SAMPLING} =f _{DTS} /16, N=5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0011: 采样频率f _{SAMPLING} =f _{CK_INT} , N=8 | 1011: 采样频率f _{SAMPLING} =f _{DTS} /16, N=6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0100: 采样频率f _{SAMPLING} =f _{DTS} /2, N=6 | 1100: 采样频率f _{SAMPLING} =f _{DTS} /16, N=8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0101: 采样频率f _{SAMPLING} =f _{DTS} /2, N=8 | 1101: 采样频率f _{SAMPLING} =f _{DTS} /32, N=5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0110: 采样频率f _{SAMPLING} =f _{DTS} /4, N=6 | 1110: 采样频率f _{SAMPLING} =f _{DTS} /32, N=6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0111: 采样频率f _{SAMPLING} =f _{DTS} /4, N=8 | 1111: 采样频率f _{SAMPLING} =f _{DTS} /32, N=8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <div><div>位3:2</div><div>IC1PSC[1:0]: 输入/捕获1预分频器 (Input capture 1 prescaler) 这2位定义了CC1输入(IC1)的预分频系数。 一旦CC1E='0'(TIMx_CCER寄存器中)，则预分频器复位。 00: 无预分频器，捕获输入口上检测到的每一个边沿都触发一次捕获； 01: 每2个事件触发一次捕获； 10: 每4个事件触发一次捕获； 11: 每8个事件触发一次捕获。</div></div> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <div><div>位1:0</div><div>CC1S[1:0]: 捕获/比较1选择 (Capture/Compare 1 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00: CC1通道被配置为输出； 01: CC1通道被配置为输入，IC1映射在TI1上； 10: CC1通道被配置为输入，IC1映射在TI2上； 11: CC1通道被配置为输入，IC1映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 注：CC1S仅在通道关闭时(TIMx_CCER寄存器的CC1E='0')才是可写的。</div></div> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 21.5.1.1 TIMx_CCMR1 寄存器

用到定时器 2 的通道 1，所以要配置 TIM2_CCMR1 寄存器的位[7: 0]，其中 CC1S[1:0]，这两个位用于 CCR1 的通道配置，这里我们设置 IC1S[1:0]=01，也就是配置 IC1 映射在 TI1 上，即 CCR1 对应 TIMx_CH1。

输入捕获 1 预分频器 IC1PSC[1:0]，我们是 1 次高电平脉冲就触发 1 次计数，所以不用分频选择 00 即可。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度，关于滤波长度的介绍请看上一个实验。这里，我们不做滤波处理，所以设置 IC1F[3:0]=0000，只要采集到上升沿，就触发捕获。

● 捕获/比较使能寄存器 (TIMx_CCER)

TIM2/TIM3/TIM4/TIM5 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 21.5.1.2 所示：

| | | | | | | | | | | | | | | | |
|----|------|--|----|------|------|----|------|------|----|------|------|----|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | CC4P | CC4E | 保留 | CC3P | CC3E | 保留 | CC2P | CC2E | 保留 | CC1P | CC1E | | | | |
| rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | |
| 位1 | | CC1P : 输入/捕获1输出极性 (Capture/Compare 1 output polarity) CC1通道配置为输出 : 0: OC1高电平有效 1: OC1低电平有效 CC1通道配置为输入 : 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0: 不反相: 捕获发生在IC1的上升沿; 当用作外部触发器时, IC1不反相。 1: 反相: 捕获发生在IC1的下降沿; 当用作外部触发器时, IC1反相。 | | | | | | | | | | | | | |
| 位0 | | CC1E : 输入/捕获1输出使能 (Capture/Compare 1 output enable) CC1通道配置为输出 : 0: 关闭— OC1禁止输出。 1: 开启— OC1信号输出到对应的输出引脚。 CC1通道配置为输入 : 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0: 捕获禁止; 1: 捕获使能。 | | | | | | | | | | | | | |

图 21.5.1.2 TIMx_CCER 寄存器

我们要用到这个寄存器的最低2位,CC1E和CC1P位。要使能输入捕获,必须设置CC1E=1,而CC1P则根据自己的需要来配置。我们这里是保留默认设置值0,即高电平触发捕获。

● 从模式控制寄存器 (TIMx_SMCR)

TIM2/TIM3/TIM4/TIM5 的从模式控制寄存器,该寄存器用于配置从模式,以及定时器的触发源相关的设置。TIMx_SMCR 寄存器描述如图 21.5.1.3 所示。

| | | | | | | | | | | | | | | | | | | | | | | | |
|------------------------|-------------------------|-----------|----|---|----|----|----|-----|---------|----|----|----------|----|----|----|------------------------|-------------------------|------------------------|-------------------------|------------------------|-------------------------|------------------------|-------------------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | |
| ETP | ECE | ETPS[1:0] | | ETF[3:0] | | | | MSM | TS[2:0] | | 保留 | SMS[2:0] | | | | | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | | | | | | | | |
| | | 位6:4 | | <p>TS[2:0]: 触发选择 (Trigger selection)</p> <p>这3位选择用于同步计数器的触发输入。</p> <table><tr><td>000: 内部触发0(ITR0), TIM1</td><td>100: TI1的边沿检测器(TI1F_ED)</td></tr><tr><td>001: 内部触发1(ITR1), TIM2</td><td>101: 滤波后的定时器输入1(TI1FP1)</td></tr><tr><td>010: 内部触发2(ITR2), TIM3</td><td>110: 滤波后的定时器输入2(TI2FP2)</td></tr><tr><td>011: 内部触发3(ITR3), TIM4</td><td>111: 外部触发输入(ETRF)</td></tr></table> <p>关于每个定时器中ITRx的细节, 参见表78。</p> <p>注: 这些位只能在未用到(如SMS=000)时被改变, 以避免在改变时产生错误的边沿检测。</p> | | | | | | | | | | | | 000: 内部触发0(ITR0), TIM1 | 100: TI1的边沿检测器(TI1F_ED) | 001: 内部触发1(ITR1), TIM2 | 101: 滤波后的定时器输入1(TI1FP1) | 010: 内部触发2(ITR2), TIM3 | 110: 滤波后的定时器输入2(TI2FP2) | 011: 内部触发3(ITR3), TIM4 | 111: 外部触发输入(ETRF) |
| 000: 内部触发0(ITR0), TIM1 | 100: TI1的边沿检测器(TI1F_ED) | | | | | | | | | | | | | | | | | | | | | | |
| 001: 内部触发1(ITR1), TIM2 | 101: 滤波后的定时器输入1(TI1FP1) | | | | | | | | | | | | | | | | | | | | | | |
| 010: 内部触发2(ITR2), TIM3 | 110: 滤波后的定时器输入2(TI2FP2) | | | | | | | | | | | | | | | | | | | | | | |
| 011: 内部触发3(ITR3), TIM4 | 111: 外部触发输入(ETRF) | | | | | | | | | | | | | | | | | | | | | | |
| | | 位2:0 | | <p>SMS[2:0]: 从模式选择 (Slave mode selection)</p> <p>当选择了外部信号, 触发信号(TRGI)的有效边沿与选中的外部输入极性相关(见输入控制寄存器和控制寄存器的说明)</p> <p>000: 关闭从模式 – 如果CEN=1, 则预分频器直接由内部时钟驱动。</p> <p>001: 编码器模式1 – 根据TI1FP1的电平, 计数器在TI2FP2的边沿向上/下计数。</p> <p>010: 编码器模式2 – 根据TI2FP2的电平, 计数器在TI1FP1的边沿向上/下计数。</p> <p>011: 编码器模式3 – 根据另一个信号的输入电平, 计数器在TI1FP1和TI2FP2的边沿向上/下计数。</p> <p>100: 复位模式 – 选中的触发输入(TRGI)的上升沿重新初始化计数器, 并且产生一个更新寄存器的信号。</p> <p>101: 门控模式 – 当触发输入(TRGI)为高时, 计数器的时钟开启。一旦触发输入变为低, 则计数器停止(但不复位)。计数器的启动和停止都是受控的。</p> <p>110: 触发模式 – 计数器在触发输入TRGI的上升沿启动(但不复位), 只有计数器的启动是受控的。</p> <p>111: 外部时钟模式1 – 选中的触发输入(TRGI)的上升沿驱动计数器。</p> <p>注: 如果TI1F_EN被选为触发输入(TS=100)时, 不要使用门控模式。这是因为, TI1F_ED在每次TI1F变化时输出一个脉冲, 然而门控模式是要检查触发输入的电平。</p> | | | | | | | | | | | | | | | | | | | |

图 21.5.1.3 TIMx_SMCR 寄存器

因为我们要让外部引脚脉冲信号作为定时器的时钟源,所以位[2:0]我们设置的值是111,即外部时钟模式1。位[6:4]是触发选择设置,TIMx_CH1对应TI1FP1,TIMx_CH2则对应TI2FP2,我们是定时器通道1,所以需要配置的值为101。ETF[3:0]和ETPS[1:0]分别是外部触发滤波器

和外部触发预分频器，我们没有用到。

接下来我们再看看 DMA/中断使能寄存器：TIMx_DIER，该寄存器的各位描述见图 21.2.1.3（在 21.2.1 小节）。本实验，我们需要用到定时器更新中断，在中断服务函数中累加定时器溢出的次数，所以需要使能定时器的更新中断，即 UIE 置 1。

控制寄存器 1：TIMx_CR1，我们只用到了它的最低位，也就是用来使能定时器的。

21.5.2 硬件设计

1. 例程功能

使用 TIM2_CH1 做输入捕获，我们将捕获 PA0 上的高电平脉宽，并将脉宽进行计数，通过串口打印出来。大家可以通过按 WK_UP 按键，输入高电平脉冲，通过按 KEY0 重设当前计数。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键：
KEY0 - PE4
WK_UP - PA0
- 3) 定时器 2，使用 TIM2 通道 1，PA0 复用为 TIM2_CH1。

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们借助 WK_UP 做输入脉冲源，捕获 PA0 上的高电平脉宽，然后对脉宽进行计数并通过串口上位机打印出来。还可以通过按 KEY0 重设当前计数。

21.5.3 程序设计

21.5.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们针对定时器从模式介绍 HAL_TIM_SlaveConfigSynchro 函数。

1. HAL_TIM_SlaveConfigSynchro 函数

该函数定义如下：

```
HAL_StatusTypeDef HAL_TIM_SlaveConfigSynchro(TIM_HandleTypeDef *htim,
                                              TIM_SlaveConfigTypeDef *sSlaveConfig);
```

- **函数描述：**
该函数用于配置定时器的从模式。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
形参 2 是 TIM_SlaveConfigTypeDef 结构体类型指针变量，用于配置定时器的从模式。
重点了解一下 TIM_SlaveConfigTypeDef 结构体指针类型，其定义如下：

```
typedef struct
{
    uint32_t SlaveMode;           /* 从模式选择 */
    uint32_t InputTrigger;        /* 输入触发源选择 */
    uint32_t TriggerPolarity;     /* 输入触发极性 */
    uint32_t TriggerPrescaler;    /* 输入触发预分频 */
    uint32_t TriggerFilter;       /* 输入滤波器设置 */
} TIM_SlaveConfigTypeDef;
```

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

定时器从模式脉冲计数配置步骤

1) 开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 2 通道 1，对应 IO 是 PA0，它们的时钟开启方法如下：

```
__HAL_RCC_TIM2_CLK_ENABLE();           /* 使能定时器 2 */
__HAL_RCC_GPIOA_CLK_ENABLE();          /* 开启 GPIOA 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输入捕获功能时，我们调用的是 HAL_TIM_IC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能和优先级设置等代码。

3) 选择从模式：外部触发模式 1

TIMx_SMCR 寄存器控制着定时器的从模式，包括模式选择、触发源选择、极性和输入预分频等。这里我们需要设置外部时钟模式 1，定时器输入 1 (TI1FP1)，并且不使用滤波器（提高响应精度），也不使用分频。HAL 库是通过 HAL_TIM_SlaveConfigSynchro 函数来初始化定时器从模式配置参数的。

4) 设置定时器为输入捕获模式，开启输入捕获

在 HAL 库中，定时器的输入捕获模式是通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器某个通道为输入捕获通道，包括映射关系，输入滤波和输入分频等。

5) 使能定时器更新中断，开启捕获功能，配置定时器中断优先级

通过 __HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start 函数使能定时器并开启捕获功能。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

6) 编写中断服务函数

定时器中断服务函数为：TIMx_IRQHandler 等，当发生中断的时候，程序就会执行中断服务函数。HAL 库提供了一个定时器中断公共处理函数 HAL_TIM_IRQHandler，该函数会根据中断类型调用相关的中断回调函数。用户根据自己的需要重定义这些中断回调函数来处理中断程序。本实验我们不使用 HAL 库的中断回调机制，而是把中断程序写在定时器中断服务函数里。详见本实验例程源码。

21.5.3.2 程序流程图

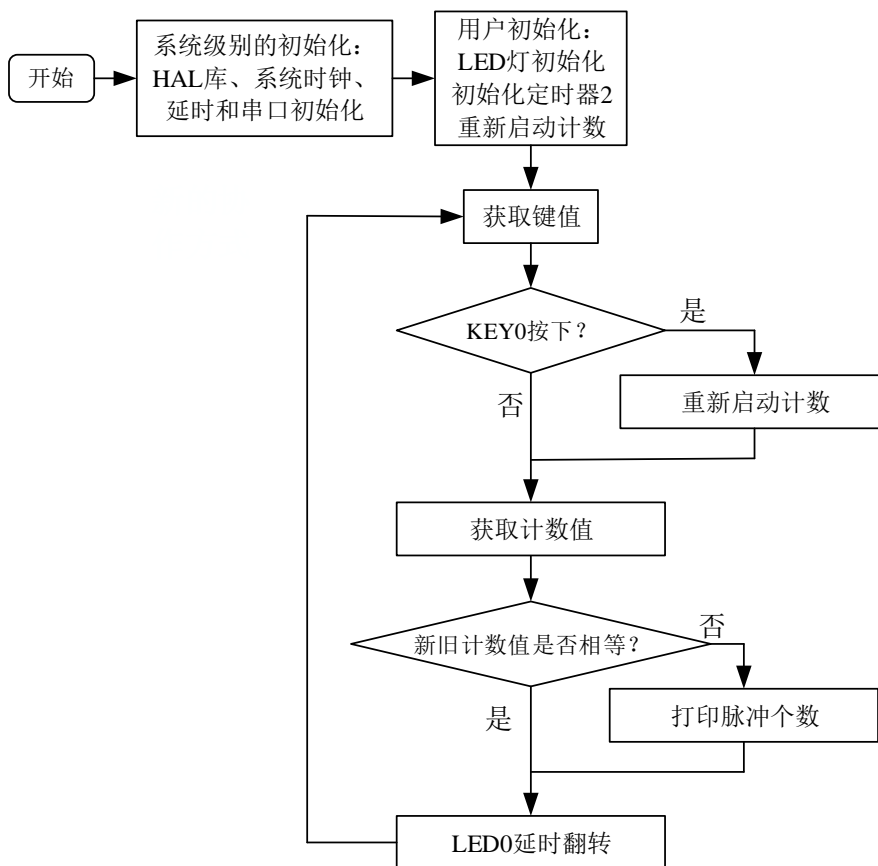


图 21.5.3.2.1 通用定时器脉冲计数实验程序流程图

21.5.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。通用定时器驱动源码包括两个文件：`gtim.c` 和 `gtim.h`。

首先看 `gtim.h` 头文件的几个宏定义：

```

/* TIMX 输入计数定义
* 这里的输入计数使用定时器 TIM2_CH1,捕获 WK_UP 按键的输入
* 默认是针对 TIM2~TIM5, 只有 CH1 和 CH2 通道可以用做输入计数, CH3/CH4 不支持!
* 注意: 通过修改这几个宏定义,可以支持 TIM1~TIM8 任意一个定时器,CH1/CH2 对应 IO 口做输入计数
* 特别要注意:默认用的 PA0,设置的是下拉输入!如果改其他 IO,对应的上下拉方式也得改!
*/
#define GTIM_TIMX_CNT_CHY_GPIO_PORT          GPIOA
#define GTIM_TIMX_CNT_CHY_GPIO_PIN          GPIO_PIN_0
#define GTIM_TIMX_CNT_CHY_GPIO_CLK_ENABLE() \
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0) /* PA 口时钟使能 */

#define GTIM_TIMX_CNT          TIM2
#define GTIM_TIMX_CNT_IRQn     TIM2_IRQn
#define GTIM_TIMX_CNT_IRQHandler TIM2_IRQHandler
#define GTIM_TIMX_CNT_CHY      TIM_CHANNEL_1 /* 通道 Y, 1<= Y <=2 */
#define GTIM_TIMX_CNT_CHY_CLK_ENABLE() \
do{ __HAL_RCC_TIM2_CLK_ENABLE(); }while(0) /* TIM2 时钟使能 */
    
```

可以把上面的宏定义分成两部分，第一部分是定时器 2 输入通道 1 对应的 IO 口的宏定义，第二部分则是定时器 2 输入通道 1 的相应宏定义。需要注意的点是：只有 CH1 和 CH2 通道可以用做输入计数，CH3/CH4 不支持！

下面看 `gtim.c` 的程序，首先是通用定时器脉冲计数初始化函数，其定义如下：

```

/**
 * @brief      通用定时器 TIMx 通道 Y 脉冲计数 初始化函数
 * @note
 *
 *      本函数选择通用定时器的时钟选择：外部时钟源模式 1 (SMS[2:0] = 111)
 *      CNT 的计数时钟源就来自 TIMx_CH1/CH2，可以实现外部脉冲计数 (脉冲接入 CH1/CH2)
 *
 *      时钟分频数 = psc，一般设置为 0，表示每一个时钟都会计数一次，以提高精度。
 *      通过读取 CNT 和溢出次数，经过简单计算，可以得到当前的计数值，从而实现脉冲计数
 * @param      arr：自动重装值
 * @retval     无
 */
void gtim_timx_cnt_chy_init(uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;
    TIM_SlaveConfigTypeDef tim_slave_config = {0};

    GTIM_TIMX_CNT_CHY_CLK_ENABLE();           /* 使能 TIMx 时钟 */
    GTIM_TIMX_CNT_CHY_GPIO_CLK_ENABLE();      /* 开启 GPIOA 时钟 */

    g_timx_cnt_chy_handle.Instance = GTIM_TIMX_CNT;           /* 定时器 x */
    g_timx_cnt_chy_handle.Init.Prescaler = psc;               /* 定时器分频 */
    g_timx_cnt_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数模式 */
    g_timx_cnt_chy_handle.Init.Period = 65535;                /* 自动重装载值 */
    HAL_TIM_IC_Init(&g_timx_cnt_chy_handle);

    gpio_init_struct.Pin = GTIM_TIMX_CNT_CHY_GPIO_PIN;       /* 输入捕获的 GPIO 口 */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;                  /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLDOWN;                    /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;            /* 高速 */
    HAL_GPIO_Init(GTIM_TIMX_CNT_CHY_GPIO_PORT, &gpio_init_struct);

    /* 从模式：外部触发模式 1 */
    tim_slave_config.SlaveMode = TIM_SLAVEMODE_EXTERNAL1; /* 从模式：外部触发模式 1 */
    tim_slave_config.InputTrigger = TIM_TS_TI1FP1;         /* TI1FP1 作为触发输入源 */
    tim_slave_config.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING; /* 上升沿 */
    tim_slave_config.TriggerPrescaler = TIM_TRIGGERPRESCALER_DIV1; /* 不分频 */
    tim_slave_config.TriggerFilter = 0x0;                  /* 滤波：本例中不需要任何滤波 */
    HAL_TIM_SlaveConfigSynchrono(&g_timx_cnt_chy_handle, &tim_slave_config);

    /* 设置中断优先级，抢占优先级 1，子优先级 3 */
    HAL_NVIC_SetPriority(GTIM_TIMX_CNT_IRQn, 1, 3);
    HAL_NVIC_EnableIRQ(GTIM_TIMX_CNT_IRQn);

    __HAL_TIM_ENABLE_IT(&g_timx_cnt_chy_handle, TIM_IT_UPDATE); /* 使能更新中断 */
    HAL_TIM_IC_Start(&g_timx_cnt_chy_handle, GTIM_TIMX_CNT_CHY); /* 使能通道输入 */
}

```

gtim_timx_cnt_chy_init 函数包含了输入通道对应 IO 的初始代码、NVIC、使能时钟、定时器基础工作参数和从模式配置的所有代码。下面来看看该函数的代码内容。

第一部分使能定时器和 GPIO 的时钟。

第二部分调用 HAL_TIM_IC_Init 函数初始化定时器的基础工作参数，如：ARR 和 PSC 等。

第三部分是定时器输入通道对应的 IO 的初始化。

第四部分调用 HAL_TIM_SlaveConfigSynchronization 函数配置从模式选择、输入捕获通道映射关系、捕获边沿和滤波等。

第五部分是 NVIC 的初始化，配置抢占优先级、响应优先级和开启 NVIC 定时器中断。

最后是使能更新中断和使能通道输入。

我们在通用定时器输入捕获实验使用了 HAL_TIM_IC_MspInit 函数，为了方便代码的管理和移植性等，这里就不再使用这个函数了。当一个项目使用到多个定时器时，也同样建议大家不要使用 HAL 库的回调机制，真的不方便。至于前面为什么要用 HAL_TIM_IC_MspInit 函数，

只是为了让大家知道 HAL 库回调机制的使用方法。大家可以根据自己的情况权衡利弊，决定是否使用 HAL 库的回调机制。

下面先看中断服务函数，在基本定时器中断实验中，我们知道中断逻辑程序的逻辑代码是放在更新中断回调函数里面的，这是 HAL 库回调机制标准的做法。因为我们在通用定时器输入捕获实验中使用过 HAL_TIM_PeriodElapsedCallback 更新中断回调函数，所以本实验我们不使用 HAL 库这套回调机制，而是直接将中断处理写在定时器中断服务函数中，定时器中断服务函数定义如下：

```
/**
 * @brief      通用定时器 TIMX 脉冲计数 更新中断服务函数
 * @param      无
 * @retval     无
 */
void GTIM_TIMX_CNT_IRQHandler(void)
{
    /* 以下代码没有使用定时器 HAL 库共用处理函数来处理，而是直接通过判断中断标志位的方式 */
    if(__HAL_TIM_GET_FLAG(&g_timx_cnt_chy_handle, TIM_FLAG_UPDATE) != RESET)
    {
        g_timxchy_cnt_ofcnt++;          /* 累计溢出次数 */
    }
    __HAL_TIM_CLEAR_IT(&g_timx_cnt_chy_handle, TIM_IT_UPDATE);
}
```

在函数中，使用 __HAL_TIM_GET_FLAG 函数宏获取更新更新中断标志位，然后判断是否发生更新中断，如果发生了更新中断，表示脉冲计数的个数等于 ARR 寄存器的值，那么我们让 g_timxchy_cnt_ofcnt 变量++，累计定时器溢出次数。最后调用 __HAL_TIM_CLEAR_IT 函数宏清除更新中断标志位。这样就完成一次对更新中断的处理。

再来介绍两个自定义的功能函数，第一个是获取当前计数值函数，其定义如下：

```
/**
 * @brief      通用定时器 TIMX 通道 Y 获取当前计数值
 * @param      无
 * @retval     当前计数值
 */
uint32_t gtim_timx_cnt_chy_get_count(void)
{
    uint32_t count = 0;
    count = g_timxchy_cnt_ofcnt * 65536; /* 计算溢出次数对应的计数值 */
    count += __HAL_TIM_GET_COUNTER(&g_timx_cnt_chy_handler); /* 加上当前 CNT 的值 */
    return count;
}
```

该函数先是计算定时器溢出次数对应的计数个数，因为定时器每溢出一次的计数个数是 65536，所以这里用 g_timxchy_cnt_ofcnt 乘以 65536，就可以得到溢出计数的个数，前提是 ARR 寄存器的值得设置为 65535。然后再加上定时器计数器当前的值，计数器当前的值通过调用 __HAL_TIM_GET_COUNTER 函数宏可以获取。函数返回值是脉冲计数的总个数。

第二个自定义功能函数是重启计数器函数，其定义如下：

```
/**
 * @brief      通用定时器 TIMX 通道 Y 重启计数器
 * @param      无
 * @retval     当前计数值
 */
void gtim_timx_cnt_chy_restart(void)
{
    __HAL_TIM_DISABLE(&g_timx_cnt_chy_handler);          /* 关闭定时器 TIMX */
    g_timxchy_cnt_ofcnt = 0;                                /* 累加器清零 */
    __HAL_TIM_SET_COUNTER(&g_timx_cnt_chy_handler, 0);    /* 计数器清零 */
    __HAL_TIM_ENABLE(&g_timx_cnt_chy_handler);            /* 使能定时器 TIMX */
}
```


该函数先关闭定时器，然后做清零操作，包括：记录溢出次数全局变量 `g_timxchy_cnt_ofcnt` 和定时器计数器的值，最后使能定时器重新计数。通用定时器脉冲计数实验的整体驱动和逻辑程序还算比较容易理解。

下面看看 `main.c` 里面编写的代码：

```
int main(void)
{
    uint32_t curcnt = 0;
    uint32_t oldcnt = 0;
    uint8_t key = 0;
    uint8_t t = 0;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */
    gtim_timx_cnt_chy_init(0); /* 定时器计数初始化，不分频 */
    gtim_timx_cnt_chy_restart(); /* 重启计数 */
    while (1)
    {
        key = key_scan(0); /* 扫描按键 */
        if (key == KEY0_PRES) /* KEY0 按键按下，重启计数 */
        {
            gtim_timx_cnt_chy_restart(); /* 重新启动计数 */
        }
        curcnt = gtim_timx_cnt_chy_get_count(); /* 获取计数值 */
        if (oldcnt != curcnt)
        {
            oldcnt = curcnt;
            printf("CNT:%d\r\n", oldcnt); /* 打印脉冲个数 */
        }
        t++;
        if (t > 20) /* 200ms 进入一次 */
        {
            t = 0;
            LED0_TOGGLE(); /* LED0 闪烁，提示程序运行 */
        }
        delay_ms(10);
    }
}
```

调用定时器初始化函数 `gtim_timx_cnt_chy_init(0)`，形参是 0，表示设置预分频器寄存器的值为 0，即表示不分频。如果形参设置为 1，就是 2 分频，这种情况，要按按键 `WK_UP` 两次才会计数一次，大家不妨试试。该函数内部已经设置自动重载寄存器的值为 65535，所以在不分频的情况下，定时器发生一次更新中断，表示脉冲计数了 65536 次。

21.5.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，我们再打开串口调试助手，然后每按 `KEY_UP` 按键一次，就可以看到串口打印的高电平脉冲次数。如果按 `KEY0` 按键，就会重设当前计数，从 0 开始计数，如图 21.5.4.1 所示：

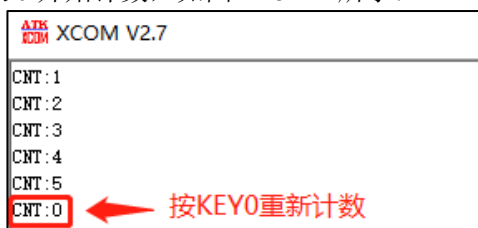


图 21.5.4.1 打印高电平脉冲次数

第二十二章 高级定时器实验

本章我们主要来学习高级定时器，STM32F103 有 2 个高级定时器（TIM1 和 TIM8）。我们将通过四个实验来学习高级定时器的各个功能，分别是高级定时器输出指定个数 PWM 实验、高级定时器输出比较模式实验、高级定时器互补输出带死区控制实验和高级定时器 PWM 输入模式实验。

本章分为如下几个小节：

22.1 高级定时器简介

22.2 高级定时器输出指定个数 PWM 实验

22.3 高级定时器输出比较模式实验

22.4 高级定时器互补输出带死区控制实验

22.5 高级定时器 PWM 输入模式实验

22.1 高级定时器简介

高级定时器的框图和通用定时器框图很类似，只是添加了其它的一些功能，如：重复计数器、带死区控制的互补输出通道、断路输入等。这些功能在高级定时器框图的位置如下：

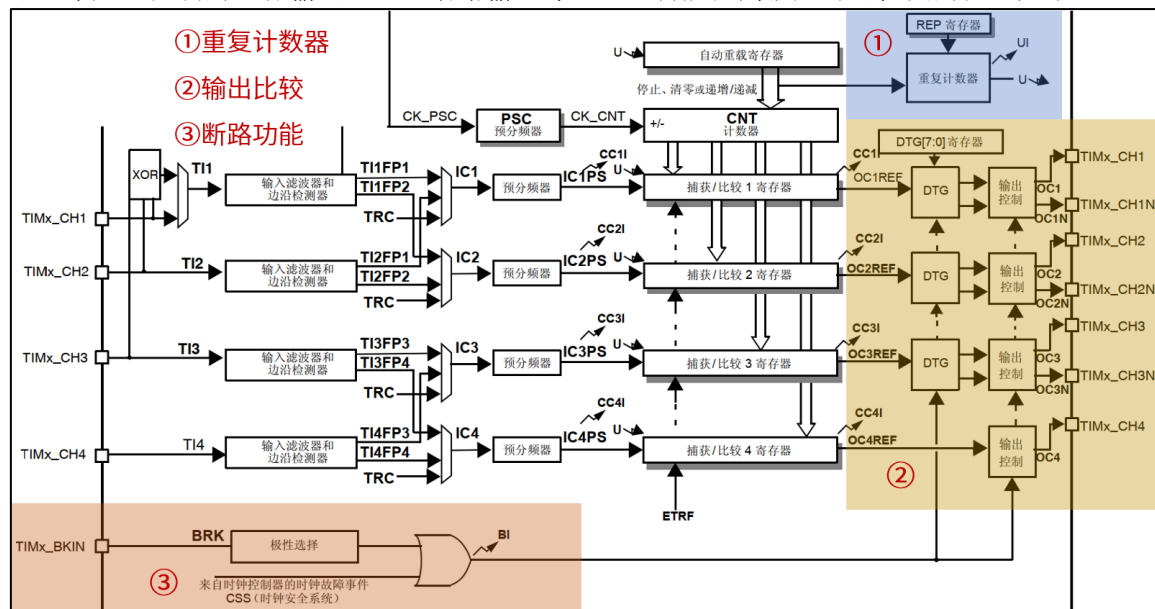


图 22.1.1 高级定时器框图

上图中，框出来三个部分，这是和通用定时器不同的地方，下面来分别介绍它们。

① 重复计数器

在 F1 系列中，高级定时器 TIM1 和 TIM8 都有重复计数器。下面来介绍一下重复计数器有什么作用？在学习基本定时器和通用定时器的时候，我们知道定时器发生上溢或者下溢时，会直接生成更新事件。但是有重复计数器的定时器并不完全是这样的，定时器每次发生上溢或下溢时，重复计数器的值会减一，当重复计数器的值为 0 时，再发生一次上溢或者下溢才会生成定时器更新事件。如果我们设置重复计数器寄存器 RCR 的值为 N，那么更新事件将在定时器发生 N+1 次上溢或下溢时发生。

这里需要注意的是重复计数器寄存器是具有影子寄存器的，所以 RCR 寄存器只是起缓冲的作用。RCR 寄存器的值会在更新事件发生时，被转移至其影子寄存器中，从而真正生效。

重复计数器的特性，在控制生成 PWM 信号时很有用，后面会有相应的实验。

② 输出比较

高级定时器输出比较部分和通用定时器相比，多了带死区控制的互补输出功能。图 22.1.1 第②部分的 TIMx_CH1N、TIMx_CH2N 和 TIMx_CH3N 分别是定时器通道 1、通道 2 和通道 3 的互补输出通道，通道 4 是没有互补输出通道的。DTG 是死区发生器，死区时间由 DTG[7:0] 位来配置。如果不使用互补通道和死区时间控制，那么高级定时器 TIM1 和 TIM8 和通用定时器的输出比较部分使用方法基本一样，只是要注意 MOE 位得置 1 定时器才能输出。

如果使用互补通道，那么就有一定的区别了，具体我们在高级定时器互补输出带死区控制实验小节再来介绍。

③ 断路功能

断路功能也称刹车功能，一般用于电机控制的刹车。F1 系列有一个断路通道，断路源可以是刹车输入引脚 (TIMx_BKIN)，也可以是一个时钟失败事件。时钟失败事件由复位时钟控制器中的时钟安全系统产生。系统复位后，断路功能默认被禁止，MOE 位为低。

使能断路功能的方法：将 TIMx_BDTR 的位 BKE 置 1。断路输入引脚 TIMx_BKIN 的输入有效电平可通过 TIMx_BDTR 寄存器的位 BKP 设置。

使能刹车功能后：由 TIMx_BDTR 的 MOE、OSSI、OSSR 位，TIMx_CR2 的 OISx、OISxN 位，TIMx_CCER 的 CCxE、CCxNE 位控制 OCx 和 OCxN 输出状态。无论何时，OCx 和 OCxN 输出都不能同时处在有效电平。

当发生断路输入后，会怎么样？

1，MOE 位被异步地清零，OCx 和 OCxN 为无效、空闲或复位状态(由 OSSI 位选择)。

2，OCx 和 OCxN 的状态：由相关控制位状态决定，当使用互补输出时：根据情况自动控制输出电平，参考《STM32F10xxx 参考手册_V10 (中文版).pdf》手册第 245 页的表 75 带刹车功能的互补通道 OcX 和 OcXN 的控制位。

3，BIF 位置 1，如果使能了 BIE 位，还会产生刹车中断；如果使能了 TDE 位，会产生 DMA 请求。

4，如果 AOE 位置 1，在下一个更新事件 UEV 时，MOE 位被自动置 1。

高级定时器框图部分就简单介绍到这里，下面通过实际的实验来学习高级定时器。

22.2 高级定时器输出指定个数 PWM 实验

要实现定时器输出指定个数 PWM，只需要掌握下面几点内容：

第一，如果大家还不清楚定时器是如何输出 PWM 的，请回顾通用定时器 PWM 输出实验的内容，这部分的知识是一样的。但是需要注意的是：我们需要把 MOE 位置 1，这样高级定时器的通道才能输出。

第二，要清楚重复计数器特性，设置重复计数器寄存器 RCR 的值为 N，那么更新事件将在定时器发生 N+1 次上溢或下溢时发生。换句话说来说就是，想要指定输出 N 个 PWM，只需要把 N-1 写入 RCR 寄存器。因为在边沿对齐模式下，定时器溢出周期对应着 PWM 周期，我们只要在更新事件发生时，停止输出 PWM 就行。

第三，为了保证定时器输出指定个数的 PWM 后，定时器马上停止继续输出，我们使能更新中断，并在定时器中断里关闭计数器。

原理部分我们就讲到这里，下面直接开始寄存器的介绍。

22.2.1 TIM1/TIM8 寄存器

下面介绍 TIM1/TIM8 这些高级定时器中使用到的几个重要的寄存器，其他更多关于定时器的资料可以参考《STM32F10xxx 参考手册_V10 (中文版).pdf》的第 13 章。

● 控制寄存器 1 (TIMx_CR1)

TIM1/TIM8 的控制寄存器 1 描述如图 22.2.1.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|--|----|----|----|----------|----|------|----------|----|-----|-----|-----|------|-----|
| 保留 | | | | | | CKD[1:0] | | ARPE | CMS[1:0] | | DIR | OPM | URS | UDIS | CEN |
| | | | | | | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位7 | | ARPE: 自动重载预装载允许位 (Auto-reload preload enable) 0: TIMx_ARR寄存器没有缓冲; 1: TIMx_ARR寄存器被装入缓冲器。 | | | | | | | | | | | | | |
| 位4 | | DIR: 方向 (Direction) 0: 计数器向上计数; 1: 计数器向下计数。 注: 当计数器配置为中央对齐模式或编码器模式时, 该位为只读。 | | | | | | | | | | | | | |
| 位0 | | CEN: 使能计数器 (Counter enable) 0: 禁止计数器; 1: 使能计数器。 注: 在软件设置了CEN位后, 外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 | | | | | | | | | | | | | |

图 22.2.1.1 TIMx_CR1 寄存器

上图中我们只列出了本章需要用的一些位, 其中: 位 7 (ARPE) 用于控制自动重载寄存器是否具有缓冲作用, 在基本定时器的时候已经讲过, 请回顾。在本实验中我们把该位要置 1, 这样就算改变 ARR 寄存器的值, 该值也不会马上生效, 而是等待之前设置的 PWM 完整输出后 (发生更新事件) 才生效。位 4 (DIR) 用于配置计数器的计数方向, 这里我们默认置 0。位 0 (CEN), 用于使能计数器的工作, 必须要设置该位为 1, 才可以开始计数。

● 捕获/比较模式寄存器 1/2 (TIMx_CCMR1/2)

TIM1/TIM8 的捕获/比较模式寄存器 (TIMx_CCMR1/2), 该寄存器一般有 2 个: TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 CH2, 而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 22.2.1.2 所示:

| 15 | | 14 | | 13 | | 12 | | 11 | | 10 | | 9 | | 8 | | 7 | | 6 | | 5 | | 4 | | 3 | | 2 | | 1 | | 0 | |
|-----------|--|-----------|--|-------------|--|-------|--|-----------|--|-----------|--|----|--|-------------|--|-----------|--|----|--|-------|--|-------|--|-----------|--|----|--|----|--|----|--|
| OC2CE | | OC2M[2:0] | | | | OC2PE | | OC2FE | | CC2S[1:0] | | | | OC1CE | | OC1M[2:0] | | | | OC1PE | | OC1FE | | CC1S[1:0] | | | | | | | |
| IC2F[3:0] | | | | IC2PSC[1:0] | | | | IC1F[3:0] | | | | | | IC1PSC[1:0] | | | | | | | | | | | | | | | | | |
| rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | |

图 22.2.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下, 功能不一样, 我们前面已经说过。比如我们要让 TIM1 的 CH1 输出 PWM 波为例, 该寄存器的模式设置位 OC1M[2:0]就是对应着通道 1 的模式设置, 此部分由 3 位组成, 总共可以配置成 8 种模式, 我们使用的是 PWM 模式, 所以这 3 位必须设置为 110 或者 111, 分别对应 PWM 模式 1 和 PWM 模式 2。这两种 PWM 模式的区别就是输出有效电平的极性相反, 这里我们设置为 PWM 模式 1。位 3 OC1PE 是输出比较通道 1 的预装使能, 该位需要置 1, 另外 CC1S[1:0]用于设置通道 1 的方向 (输入/输出) 默认设置为 0, 就是设置通道作为输出使用。

● 捕获/比较使能寄存器 (TIMx_CCER)

TIM1/TIM8 的捕获/比较使能寄存器, 该寄存器控制着各个输入输出通道的开关。TIMx_CCER 寄存器描述如图 22.2.1.3 所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|------|------|-------|-------|------|------|-------|-------|------|------|-------|-------|------|------|
| 保留 | | CC4P | CC4E | CC3NP | CC3NE | CC3P | CC3E | CC2NP | CC2NE | CC2P | CC2E | CC1NP | CC1NE | CC1P | CC1E |
| | | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |

图 22.2.1.3 TIMx_CCER 寄存器

该寄存器比较简单, 要让 TIM1 的 CH1 输出 PWM 波, 这里我们要使能 CC1E 位, 该位是通道 1 输入/输出使能位, 要想 PWM 从 IO 口输出, 这个位必须设置为 1。CC1P 位是设置通道 1 的输出极性, 我们设置 0, 即 OC1 高电平有效。

● 事件产生寄存器 (TIMx_EGR)

TIM1/TIM8 的事件产生寄存器, 该寄存器作用是让用户用软件方式产生各类事件。TIMx_EGR 寄存器描述如图 22.2.1.4 所示:

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|----|----|------|------|------|------|------|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | BG | TG | COMG | CC4G | CC3G | CC2G | CC1G | UG |
| | | | | | | | | W | W | W | W | W | W | W | W |

图 22.2.1.4 TIMx_EGR 寄存器

UG 位是更新事件的控制位，作用和定时器溢出时产生的更新事件一样，区别是这里是通过软件产生的，而定时器溢出是硬件自己完成的。只有开启了更新中断，这两种方式都可以产生更新中断。本实验用到该位去产生软件更新器事件，在需要的时候把 UG 位置 1 即可，会由硬件自动清零。

● 重复计数器寄存器 (TIMx_RCR)

重复计数器寄存器用于设置重复计数器值，因为它具有影子寄存器，所以它本身只是起缓冲作用。当更新事件发生时，该寄存器的值会转移到其影子寄存器中，从而真正起作用。TIMx_RCR 寄存器描述如图 22.2.1.5 所示：

| | | | | | | | | | | | | | | | |
|------|----|--|----|----|----|---|---|----------|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | REP[7:0] | | | | | | | |
| | | | | | | | | RW | RW | RW | RW | RW | RW | RW | RW |
| 位7:0 | | REP[7:0]: 重复计数器的值 (Repetition counter value) 开启了预装载功能后，这些位允许用户设置比较寄存器的更新速率(即周期性地从预装载寄存器传输到当前寄存器)；如果允许产生更新中断，则会同时影响产生更新中断的速率。 每次向下计数器REP_CNT达到0，会产生一个更新事件并且计数器REP_CNT重新从REP值开始计数。由于REP_CNT只有在周期更新事件U_RC发生时才重载REP值，因此对TIMx_RCR寄存器写入的新值只在下次周期更新事件发生时才起作用。 这意味着在PWM模式中，(REP+1)对应着： <ul style="list-style-type: none"> 在边沿对齐模式下，PWM周期的数目； 在中心对称模式下，PWM半周期的数目； | | | | | | | | | | | | | |

图 22.2.1.5 TIMx_RCR 寄存器

该寄存器的 REP[7:0]位是低 8 位有效，即最大值 255。因为这个寄存器只是起缓冲作用，如果大家对该寄存器写入值后，想要立即生效，可以通过对 UG 位写 1，产生软件更新事件。

● 捕获/比较寄存器 1/2/3/4 (TIMx_CCR1/2/3/4)

捕获/比较寄存器 (TIMx_CCR1/2/3/4)，该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器的描述，如图 22.2.1.6 所示：

| | | | | | | | | | | | | | | | |
|------------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CCR1[15:0] | | | | | | | | | | | | | | | |
| 位15:0 | | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| | | CCR1[15:0]: 捕获/比较通道1的值 (Capture/Compare 1 value) 若CC1通道配置为输出： CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载功能，写入的数值会立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 若CC1通道配置为输入： CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。 | | | | | | | | | | | | | |

图 22.2.1.6 TIMx_CCR1 寄存器

在输出模式下，捕获/比较寄存器影子寄存器的值与 CNT 的值比较，根据比较结果产生相应动作，利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的占空比了。

● 断路和死区寄存器 (TIMx_BDTR)

高级定时器 TIM1/8 的通道用作输出时，还必须配置断路和死区寄存器 (TIMx_BDTR) 的位 MOE，该寄存器各位描述如图 22.3.1.7 所示：

| | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|------|------|-----------|----------|----|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MOE | AOE | BKP | BKE | OSSR | OSSI | LOCK[1:0] | DTG[7:0] | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |

注释：根据锁定设置，AOE、BKP、BKE、OSSI、OSSR和DTG[7:0]位均可被写保护，有必要在第一次写入TIMx_BDTR寄存器时对它们进行配置。

| | |
|-----|---|
| 位15 | <p>MOE: 主输出使能 (Main output enable)</p> <p>一旦刹车输入有效，该位被硬件异步清'0'。根据AOE位的设置值，该位可以由软件清'0'或被自动置1。它仅对配置为输出的通道有效。</p> <p>0: 禁止OC和OCN输出或强制为空闲状态；</p> <p>1: 如果设置了相应的使能位(TIMx_CCER寄存器的CCxE、CCxNE位)，则开启OC和OCN输出。</p> <p>有关OC/OCN使能的细节，参见13.4.9节，TIM1和TIM8捕获/比较使能寄存器(TIMx_CCER)。</p> |
|-----|---|

图 22.3.1.7 TIMx_BDTR 寄存器

本实验，我们只需要关注该寄存器的位 15 (MOE)，要想高级定时器的 PWM 正常输出，则必须设置 MOE 位为 1，否则不会有输出。

22.2.2 硬件设计

1. 例程功能

通过 TIM8_CH1 (由 PC6 复用) 输出 PWM，然后为了指示 PWM 的输出情况，我们用杜邦线将 PC6 和 PE5 引脚的排针连接起来，从而实现 PWM 输出控制 LED1 (硬件已连接在 PPE5 引脚上) 的亮灭。注意的点是：PE5 要设置成浮空输入，避免引脚冲突，我们在 main 函数中设置好了，请看源码。上电默认输出 5 个 PWM 波，连接好杜邦线后可以看见 LED1 亮灭五次。之后按一下按键 KEY0，就会输出 5 个 PWM 波控制 LED1 亮灭五次。LED0 闪烁提示系统正在运行。

2. 硬件资源

- 1) LED 灯：
 - LED0 – PB5
 - LED1 – PE5
- 2) 独立按键：
 - KEY0 – PE4
- 3) 定时器 8，使用 TIM8 通道 1，由 PC6 复用。用杜邦线将 PC6 和 PE5 引脚连接起来。

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 LED1 来指示 STM32F103 的定时器的 PWM 输出情况，所以需要一根杜邦线连接 PC6 和 PE5，同时还用按键 KEY0 进行控制。

22.2.3 程序设计

本实验用到的 HAL 库函数介绍请回顾通用定时器 PWM 输出实验。下面介绍一下定时器输出指定个数 PWM 的配置步骤。

定时器输出指定个数 PWM 配置步骤

1) 开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1，对应 IO 是 PC6，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE(); /* 使能定时器 8 */
__HAL_RCC_GPIOC_CLK_ENABLE(); /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用 HAL_TIM_PWM_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3) 设置定时器为 PWM 模式，输出比较极性，比较值等参数

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

本实验我们设置 TIM8 的通道 1 为 PWM1 模式，使用杜邦线把 PC6 与 PE5 进行连接，因为我们的 LED1（连接 PE5）是低电平亮，而我们希望输出最后一个 PWM 波的时候，LED1 就灭，所以我们设置输出比较极性为高。捕获/比较寄存器的值（即比较值）设置为自动重装载值的一半，即 PWM 占空比为 50%。

4) 使能定时器更新中断，开启定时器并输出 PWM，配置定时器中断优先级

通过 HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_PWM_Start 函数使能定时器并开启输出 PWM。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5) 编写中断服务函数

定时器中断服务函数为：TIMx_IRQHandler 等，当发生中断的时候，程序就会执行中断服务函数。HAL 库提供了一个定时器中断公共处理函数 HAL_TIM_IRQHandler，该函数会根据中断类型调用相关的中断回调函数。用户根据自己的需要重定义这些中断回调函数来处理中断程序。本实验我们不使用 HAL 库的中断回调机制，而是把中断程序写在定时器中断服务函数里。详见本章例程源码。

22.2.3.1 程序流程图

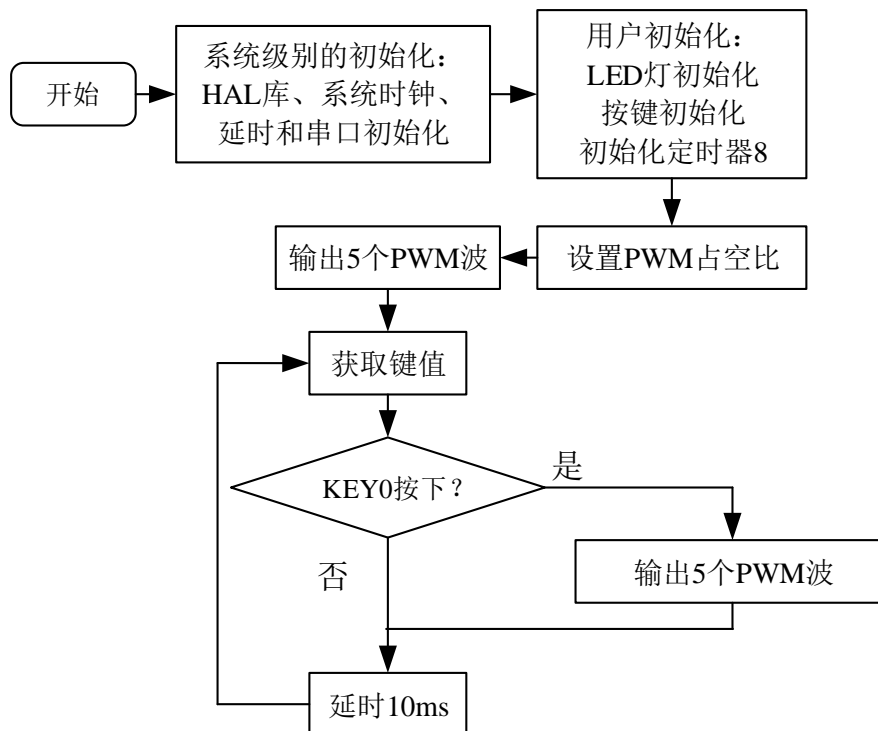


图 22.2.3.2.1 高级定时器输出指定个数 PWM 实验程序流程图

22.2.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。本章节的四个实验源码都是存放在 atim.c 和 atim.h 中，源码中也有明确的注释。

首先看 atim.h 头文件的几个宏定义：

```
/* TIMX 输出指定个数 PWM 定义
 * 这里输出的 PWM 通过 PC6 (TIM8_CH1) 输出，我们用杜邦线连接 PC6 和 PE5，然后在程序里面将 PE5 设
 * 置成浮空输入就可以 看到 TIM8_CH1 控制 LED1 (GREEN) 的亮灭，亮灭一次表示一个 PWM 波
 * 默认使用的是 TIM8_CH1.
 * 注意：通过修改这几个宏定义，可以支持 TIM1/TIM8 定时器，任意一个 IO 口输出指定个数的 PWM
 */
#define ATIM_TIMX_NPWM_CHY_GPIO_PORT      GPIOC
#define ATIM_TIMX_NPWM_CHY_GPIO_PIN        GPIO_PIN_6
#define ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE() do{__HAL_RCC_GPIOC_CLK_ENABLE();\
}while(0) /* PC 口时钟使能 */
#define ATIM_TIMX_NPWM                      TIM8
#define ATIM_TIMX_NPWM_IRQn                 TIM8_UP_IRQn
#define ATIM_TIMX_NPWM_IRQHandler           TIM8_UP_IRQHandler
#define ATIM_TIMX_NPWM_CHY                  TIM_CHANNEL_1 /* 通道 Y, 1<= Y <=4 */
#define ATIM_TIMX_NPWM_CHY_CCRX             TIM8->CCR1 /* 通道 Y 的输出比较寄存器 */
#define ATIM_TIMX_NPWM_CHY_CLK_ENABLE()     do{__HAL_RCC_TIM8_CLK_ENABLE();\
}while(0)
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输入通道 1 对应的 IO 口的宏定义，第二部分则是定时器 8 输入通道 1 的相应宏定义。

下面看 atim.c 的程序，首先是输出指定个数 PWM 初始化函数，其定义如下：

```
/**
 * @brief      高级定时器 TIMX 通道 Y 输出指定个数 PWM 初始化函数
 * @note
 *
 * 高级定时器的时钟来自 APB2，而 PCLK2 = 72Mhz，我们设置 PPRE2 不分频，因此
 * 高级定时器时钟 = 72Mhz
 * 定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 * Ft=定时器工作频率，单位:Mhz
 * @param      arr: 自动重装值
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void atim_timx_npwm_chy_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;
    TIM_OC_InitTypeDef timx_oc_npwm_chy; /* 定时器输出 */
    ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE(); /* TIMX 通道 IO 口时钟使能 */
    ATIM_TIMX_NPWM_CHY_CLK_ENABLE(); /* TIMX 时钟使能 */

    g_timx_npwm_chy_handle.Instance = ATIM_TIMX_NPWM; /* 定时器 x */
    g_timx_npwm_chy_handle.Init.Prescaler = psc; /* 定时器分频 */
    g_timx_npwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_npwm_chy_handle.Init.Period = arr; /* 自动重装值 */
    g_timx_npwm_chy_handle.Init.AutoReloadPreload =
        TIM_AUTORELOAD_PRELOAD_ENABLE; /* 使能 TIMx_ARR 进行缓冲 */
    g_timx_npwm_chy_handle.Init.RepetitionCounter = 0; /* 重复计数器初始值 */
    HAL_TIM_PWM_Init(&g_timx_npwm_chy_handle); /* 初始化 PWM */

    gpio_init_struct.Pin = ATIM_TIMX_NPWM_CHY_GPIO_PIN; /* 通道 y 的 GPIO 口 */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(ATIM_TIMX_NPWM_CHY_GPIO_PORT, &gpio_init_struct);
```

```
timx_oc_npwm_chy.OCMode = TIM_OCMode_PWM1; /* 模式选择 PWM 1 */
timx_oc_npwm_chy.Pulse = arr / 2; /* 设置比较值,此值用来确定占空比 */
timx_oc_npwm_chy.OCpolarity = TIM_OCPolarity_HIGH; /* 输出比较极性为高 */
HAL_TIM_PWM_ConfigChannel(&g_timx_npwm_chy_handle, &timx_oc_npwm_chy,
                          ATIM_TIMX_NPWM_CHY); /* 配置 TIMx 通道 y */

/* 设置中断优先级,抢占优先级 1,子优先级 3 */
HAL_NVIC_SetPriority(ATIM_TIMX_NPWM_IRQn, 1, 3);
HAL_NVIC_EnableIRQ(ATIM_TIMX_NPWM_IRQn); /* 开启 ITMx 中断 */

__HAL_TIM_ENABLE_IT(&g_timx_npwm_chy_handle, TIM_IT_UPDATE); /* 允许更新中断 */
HAL_TIM_PWM_Start(&g_timx_npwm_chy_handle, ATIM_TIMX_NPWM_CHY); /* 使能输出 */
}
```

atim_timx_npwm_chy_init 函数包含了输出通道对应 IO 的初始代码、NVIC、使能时钟、定时器基础工作参数和输出模式配置的所有代码。下面来看看该函数的代码内容。

第一部分使能定时器和 GPIO 的时钟。

第二部分调用 HAL_TIM_PWM_Init 函数初始化定时器基础工作参数,如:ARR 和 PSC 等。

第三部分是定时器输出通道对应的 IO 的初始化。

第四部分调用 HAL_TIM_PWM_ConfigChannel 设置 PWM 模式以及比较值等参数。

第五部分是 NVIC 的初始化,配置抢占优先级、响应优先级和开启 NVIC 定时器中断。

最后是使能更新中断和使能通道输出。

为了方便代码的管理和移植性等,这里就没有使用 HAL_TIM_PWM_MspInit 这个函数来存放使能时钟、GPIO、NVIC 相关的代码,而是全部存放在 gtim_timx_npwm_chy_init 函数中。

下面我们看设置 PWM 个数的函数,其定义如下:

```
/* g_npwm_remain 表示当前还剩下多少个脉冲要发送
 * 每次最多发送 256 个脉冲
 */
static uint32_t g_npwm_remain = 0;

/**
 * @brief      高级定时器 TIMx NPWM 设置 PWM 个数
 * @param      rcr: PWM 的个数, 1~2^32 次方个
 * @retval      无
 */
void atim_timx_npwm_chy_set(uint32_t npwm)
{
    if (npwm == 0) return;
    g_npwm_remain = npwm; /* 保存脉冲个数 */
    /* 产生一次更新事件,在中断里面处理脉冲输出 */
    HAL_TIM_GenerateEvent(&g_timx_npwm_chy_handle, TIM_EVENTSOURCE_UPDATE);
    __HAL_TIM_ENABLE(&g_timx_npwm_chy_handle); /* 使能定时器 TIMx */
}
```

我们要输出多少个周期的 PWM 就用这个函数来设置。该函数作用是把我们的设置输出的 PWM 个数的值赋值给静态全局变量 g_npwm_remain,该变量会在更新中断服务函数回调函数中发挥作用。最后对 TIMx_EGR 寄存器 UG 位写 1,产生一次更新事件,并使能定时器。

下面来介绍定时器中断服务函数,其定义如下:

```
/**
 * @brief      定时器中断服务函数
 * @param      无
 * @retval      无
 */
void ATIM_TIMX_NPWM_IRQHandler(void)
{
    uint16_t npwm = 0;
    /* 以下代码没有使用定时器 HAL 库共用处理函数来处理,而是直接通过判断中断标志位的方式 */
    if(__HAL_TIM_GET_FLAG(&g_timx_npwm_chy_handle, TIM_FLAG_UPDATE) != RESET)
    {
        if (g_npwm_remain >= 256) /* 还有大于 256 个脉冲需要发送 */
        {

```

```

        g_npwm_remain=g_npwm_remain - 256;
        npwm = 256;
    }
    else if (g_npwm_remain % 256) /* 还有位数（不到 256）个脉冲要发送 */
    {
        npwm = g_npwm_remain % 256;
        g_npwm_remain = 0;          /* 没有脉冲了 */
    }
    if (npwm)                      /* 有脉冲要发送 */
    {
        ATIM_TIMX_NPWM->RCR = npwm - 1; /* 设置 RCR 值为 npwm-1, 即 npwm 个脉冲 */
        HAL_TIM_GenerateEvent(&g_timx_npwm_chy_handle,
                               TIM_EVENTSOURCE_UPDATE); /* 产生一次更新事件,以更新 RCR 寄存器 */
        __HAL_TIM_ENABLE(&g_timx_npwm_chy_handle); /* 使能定时器 TIMX */
    }
    else
    {
        /* 关闭定时器 TIMX,使用 __HAL_TIM_DISABLE 需要失能通道输出,所以不用 */
        ATIM_TIMX_NPWM->CR1 &= ~(1 << 0);
    }
    /* 清除定时器更新中断标志位 */
    __HAL_TIM_CLEAR_IT(&g_timx_npwm_chy_handle, TIM_IT_UPDATE);
}
}

```

这里我们没有使用 HAL 库的中断回调机制,而是想寄存器操作一样,直接通过判断中断标志位处理中断。通过 `__HAL_TIM_GET_FLAG` 函数宏判断是否发生更新中断,然后进行更新中断的代码处理,最后通过 `__HAL_TIM_CLEAR_IT` 函数宏清除更新中断标志位。

因为重复计数器寄存器 (TIM8_RCR)是 8 位有效的,所以在定时器中断服务函数中首先对全局变量 `g_npwm_remain` (即我们要输出的 PWM 个数)进行判断,是否大于 256,如果大于 256,那就得分次写入重复计数器寄存器。写入重复计数寄存器后,需要产生软件更新事件把 RCR 寄存器的值更新到 RCR 影子寄存器中,最后一定不要忘记清除定时器更新中断标志位。

在 main 函数里面编写如下代码:

```

int main(void)
{
    uint8_t key = 0;
    uint8_t t = 0;
    GPIO_InitTypeDef gpio_init_struct;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */

    /* 将 LED1 引脚设置为输入模式,避免和 PC6 冲突 */
    gpio_init_struct.Pin = LED1_GPIO_PIN; /* LED1 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_INPUT; /* 设置输入状态 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct); /* 初始化 LED1 引脚 */

    atim_timx_npwm_chy_init(5000 - 1, 7200 - 1); /* 10Khz 的计数频率,2hz 的 PWM 频率 */
    /* 设置 PWM 占空比,50%,这样可以控制每一个 PWM 周期,LED1 (BLUE) 有一半时间是亮的,
    一半时间是灭的,LED1 亮灭一次,表示一个 PWM 波 */
    ATIM_TIMX_NPWM_CHY_CCRX = 2500;
    atim_timx_npwm_chy_set(5); /* 输出 5 个 PWM 波(控制 LED1)闪烁 5 次 */

    while (1)
    {

```



```

key = key_scan(0);
if (key == KEY0_PRES) /* KEY0 按下 */
{
    gtim_timx_npwm_chy_set(5); /* 输出 5 个 PWM 波 (控制 LED1 闪烁 5 次) */
}
t++;
delay_ms(10);
if (t > 50) /* 控制 LED1 闪烁, 提示程序运行状态 */
{
    t = 0;
    LED0_TOGGLE();
}
}
}

```

先看 `gtim_timx_npwm_chy_init(5000 - 1, 7200 - 1)`; 这个语句, 这两个形参分别设置自动重载寄存器的值为 4999, 以及预分频器寄存器的值为 7199。按照 `sys_stm32_clock_init` 函数的配置, 定时器 8 的时钟频率等于 APB2 总线时钟频率, 即 72MHz, 可以得到计数器的计数频率是 10KHz。自动重载寄存器的值决定的是 PWM 周期或频率 (请回顾 21.3 小节的内容), 计数器计 5000 个数所用的时间是 PWM 的周期。在边沿对齐模式下, 定时器的溢出周期等于 PWM 的周期。根据定时器溢出时间计算公式, 可得:

$$T_{out} = ((arr+1)*(psc+1))/T_{clk} = ((4999+1)*(7199+1))/72000000 = 0.5s$$

再由频率是周期的倒数关系得到 PWM 的频率为 2Hz。

占空比则由捕获/比较寄存器 (TIMx_CCRx) 的值决定, 这里就是由 TIM8_CCR1 寄存器决定。初始化定时器 8 时我们设置通道输出比较极性为高, `GTIM_TIMX_NPWM_CHY_CCRX = 2500`, 就设置了占空比为 50%。因为我们的 LED 灯是低电平点亮, 所以正占空比期间 LED 灯熄灭, 负占空比期间 LED 灯亮。

22.2.4 下载验证

首先用杜邦线连接好 PE5 和 PC6 引脚的排针。下载代码后, 可以看到 LED1 亮灭五次, 然后我们每按一下按键 KEY0, LED1 都会亮灭五次。

下面我们使用正点原子 DS100 手持数字示波器, 把 PC6 引脚的波形截获, 具体如下:

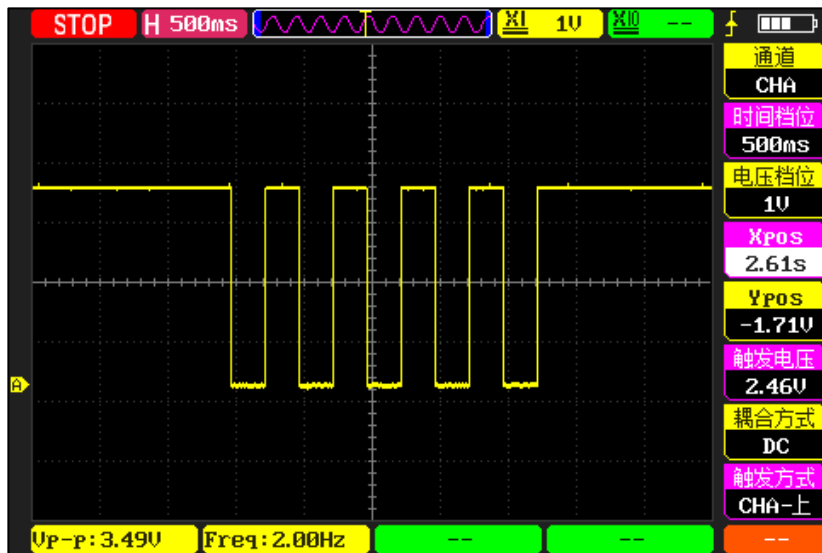


图 22.2.4 PC6 引脚波形图

由 LED 的原理图可以知道, PC6 引脚输出低电平 LED1 亮、输出高电平 LED1 灭。图 22.2.4 中, 从左往右看, 可以知道, LED0 一开始是熄灭的, 然后经过 5 次亮灭, 最后就是一直保持熄灭的状态。PWM 频率是 2Hz, 占空比 50%, 请大家自行测量。

22.3 高级定时器输出比较模式实验

本小节我们来学习使用高级定时器输出比较模式下翻转功能，通过定时器 4 个通道分别输出 4 个 50% 占空比、不同相位的 PWM。

输出比较模式下翻转功能作用是：当计数器的值等于捕获/比较寄存器影子寄存器的值时，OC1REF 发生翻转，进而控制通道输出（OCx）翻转。通过翻转功能实现输出 PWM 的具体原理如下：PWM 频率由自动重载寄存器（TIMx_ARR）的值决定，在这个过程中，只要自动重载寄存器的值不变，那么 PWM 占空比就固定为 50%。我们可以通过捕获/比较寄存器（TIMx_CCRx）的值改变 PWM 的相位。生成 PWM 的原理如图 22.3.1 所示：

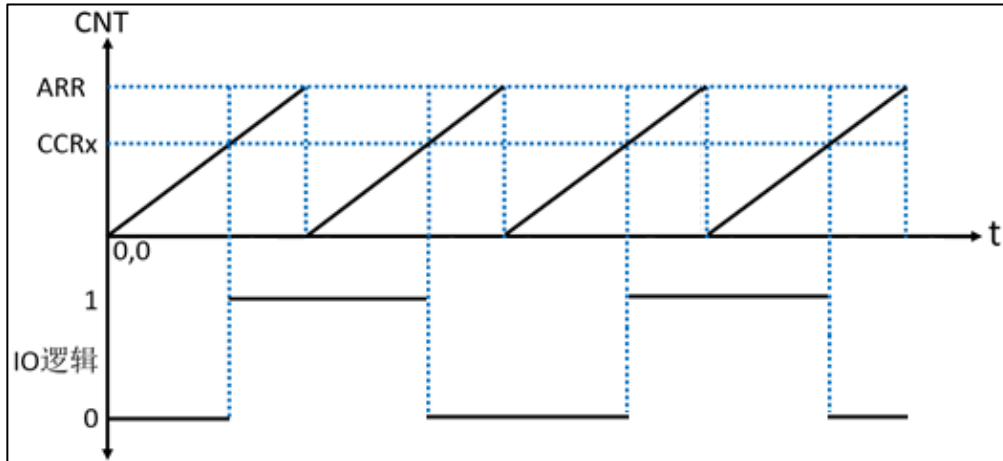


图 22.3.1 翻转功能输出 PWM 原理示意图

本实验就是根据图 22.3.1 的原理来设计的，具体实验是：我们设置固定的 ARR 值为 999，那么 PWM 占空比固定为 50%，通过改变 4 个通道的捕获/比较寄存器（TIMx_CCRx）的值使得每个通道输出的 PWM 的相位都不一样，注意捕获/比较寄存器的值设置范围是：0 ~ ARR。比如：TIMx_CCR1=250-1，TIMx_CCR2=500-1，TIMx_CCR3=750-1，TIMx_CCR4=1000-1，那么可以得到通道 1~通道 4 输出的 PWM 的相位分别是：25%、50%、75%、100%。翻转功能输出的 PWM 周期，这里用 T 表示，其计算公式如下：

$$T = 2 * (arr + 1) * ((psc + 1) / Tclk)$$

其中：

T：翻转功能输出的 PWM 周期（单位为 s）。

Tclk：定时器的时钟源频率（单位为 MHz）。

arr：自动重装寄存器（TIMx_ARR）的值。

psc：预分频器寄存器（TIMx_PSC）的值。

22.3.1 TIM1/TIM8 寄存器

高级定时器输出比较模式除了用到定时器的时基单元：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）之外。主要还用到以下这些寄存器：

- 控制寄存器 1（TIMx_CR1）

TIM1/TIM8 的控制寄存器 1 描述如图 22.3.1.1 所示。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|--|----|----|----|----------|----|------|----------|----|-----|-----|-----|------|-----|
| 保留 | | | | | | CKD[1:0] | | ARPE | CMS[1:0] | | DIR | OPM | URS | UDIS | CEN |
| | | | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位7 | | ARPE: 自动重载预装载允许位 (Auto-reload preload enable) 0: TIMx_ARR寄存器没有缓冲; 1: TIMx_ARR寄存器被装入缓冲器。 | | | | | | | | | | | | | |
| 位4 | | DIR: 方向 (Direction) 0: 计数器向上计数; 1: 计数器向下计数。 注: 当计数器配置为中央对齐模式或编码器模式时, 该位为只读。 | | | | | | | | | | | | | |
| 位0 | | CEN: 使能计数器 (Counter enable) 0: 禁止计数器; 1: 使能计数器。 注: 在软件设置了CEN位后, 外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 | | | | | | | | | | | | | |

图 22.3.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的一些位, 其中: 位 7 (APRE) 用于控制自动重载寄存器是否具有缓冲作用, 在基本定时器的時候已经讲过, 请回顾。本实验中, 我们把该位置 1。

位 4 (DIR) 用于配置计数器的计数方向, 本实验默认置 0 即可。

位 CEN 位, 用于使能计数器的工作, 必须要设置该位为 1, 才可以开始计数。

其它位保持复位值即可。

● 捕获/比较模式寄存器 1/2 (TIMx_CCMR1/2)

TIM1/TIM8 的捕获/比较模式寄存器 (TIMx_CCMR1/2), 该寄存器一般有 2 个: TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 CH2, 而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 22.3.1.2 所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|-----------|----|-----------|----|-------------|-------|-----------|-----------|----|-------------|-----------|----|----|-------|-------|-----------|--|
| OC2CE | | OC2M[2:0] | | | OC2PE | OC2FE | CC2S[1:0] | | OC1CE | OC1M[2:0] | | | OC1PE | OC1FE | CC1S[1:0] | |
| IC2F[3:0] | | | | IC2PSC[1:0] | | IC1F[3:0] | | | IC1PSC[1:0] | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | |

图 22.3.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下, 功能不一样, 我们现在用到输出比较模式。关于该寄存器的详细说明, 请参考《STM32F10xxx 参考手册_V10 (中文版).pdf》第 240 页, 13.4.7 节。

本实验我们用到了定时器 8 输出比较的 4 个通道, 所以我们需要配置 TIM1_CCMR1 和 TIM1_CCMR2。以 TIM1_CCMR1 寄存器为例, 模式设置位 OC1M[2:0]就是对应着通道 1 的模式设置, 此部分由 3 位组成, 总共可以配置成 8 种模式, 我们使用的是翻转功能, 所以这 3 位必须设置为 011。通道 2 也是如此, 将位 OC2M[2:0] 设置为 011。通道 3 和通道 4 就要设置 TIM1_CCMR2 寄存器的位 OC3M[2:0]和位 OC4M[2:0]。除此之外, 我们还要设置输出比较的预装载使能位, 如通道 1 对应输出比较的预装载使能位 OC1PE 置 1, 其他通道也要把相应位置 1。

● 捕获/比较使能寄存器 (TIMx_CCER)

TIM1/TIM8 的捕获/比较使能寄存器, 该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 22.3.1.3 所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|------|------|-------|-------|------|------|-------|-------|------|------|-------|-------|------|------|
| 保留 | | CC4P | CC4E | CC3NP | CC3NE | CC3P | CC3E | CC2NP | CC2NE | CC2P | CC2E | CC1NP | CC1NE | CC1P | CC1E |
| | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

图 22.3.1.3 TIMx_CCER 寄存器

该寄存器比较简单, 要让 TIM8 的 4 个通道都输出, 我们需要把对应的捕获/比较 1 输出使能位置 1。通道 1 到通道 4 的使能位分别是: CC1E、CC2E、CC3E、CC4E, 我们把这 4 个位置 1, 使能通道输出。

● 捕获/比较寄存器 1/2/3/4 (TIMx_CCR1/2/3/4)

捕获/比较寄存器 (TIMx_CCR1/2/3/4), 该寄存器总共有 4 个, 对应 4 个通道 CH1~CH4。

本实验 4 个通道都要使用到，以通道 1 对应的 TIMx_CCR1 寄存器为例，其描述如下图所示：

| | | | | | | | | | | | | | | | |
|------------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CCR1[15:0] | | | | | | | | | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位15:0 | | CCR1[15:0]: 捕获/比较通道1的值 (Capture/Compare 1 value) 若CC1通道配置为输出: CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载功能，写入的数值会立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 若CC1通道配置为输入: CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。 | | | | | | | | | | | | | |

图 22.3.1.4 TIMx_CCR1 寄存器

这里，我们通过改变 TIMx_CCR1/2/3/4 寄存器的值来改变 4 个通道输出的 PWM 的相位。

● TIM1/TIM8 断路和死区寄存器 (TIMx_BDTR)

本实验用的是高级定时器，我们还需要配置：断路和死区寄存器 (TIMx_BDTR)，该寄存器各位描述如图 22.3.1.5 所示。

| | | | | | | | | | | | | | | | |
|--|-----|--|-----|------|------|-----------|----|----------|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MOE | AOE | BKP | BKE | OSSR | OSSI | LOCK[1:0] | | DTG[7:0] | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 注释：根据锁定设置，AOE、BKP、BKE、OSSI、OSSR和DTG[7:0]位均可被写保护，有必要在第一次写入TIMx_BDTR寄存器时对它们进行配置。 | | | | | | | | | | | | | | | |
| 位15 | | MOE: 主输出使能 (Main output enable) 一旦刹车输入有效，该位被硬件异步清'0'。根据AOE位的设置值，该位可以由软件清'0'或被自动置1。它仅对配置为输出的通道有效。 0: 禁止OC和OCN输出或强制为空闲状态； 1: 如果设置了相应的使能位(TIMx_CCER寄存器的CCxE、CCxNE位)，则开启OC和OCN输出。 有关OC/OCN使能的细节，参见13.4.9节，TIM1和TIM8捕获/比较使能寄存器(TIMx_CCER)。 | | | | | | | | | | | | | |

图 22.3.1.5 TIMx_BDTR 寄存器

该寄存器，我们只需要关注位 15 (MOE)，要想高级定时器的通道正常输出，则必须设置 MOE 位为 1，否则不会有输出。

22.3.2 硬件设计

1. 例程功能

使用输出比较模式的翻转功能，通过定时器 8 的 4 路通道输出占空比固定为 50%、相位分别是 25%、50%、75%和 100%的 PWM。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) PC6 复用为 TIM8_CH1
PC7 复用为 TIM8_CH2
PC8 复用为 TIM8_CH3
PC9 复用为 TIM8_CH4

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们需要通过示波器观察 PC6、PC7、PC8 和 PC9 引脚 PWM 输出的情况。

22.3.3 程序设计

22.3.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，请回顾，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_OC_Init 函数

定时器的输出比较模式初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**
用于初始化定时器的输出比较模式。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIM_OC_ConfigChannel 函数

定时器的输出比较通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_ConfigChannel(TIM_HandleTypeDef *htim,
TIM_OC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**
该函数用于初始化定时器的输出比较通道。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
形参 2 是 TIM_OC_InitTypeDef 结构体类型指针变量，用于配置定时器的输出比较参数。
在通用定时器 PWM 输出实验已经介绍过 TIM_OC_InitTypeDef 结构体指针类型。
形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_OC_Start 函数

定时器的输出比较启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

- **函数描述：**
用于启动定时器的输出比较模式。
- **函数形参：**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。
形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
HAL 库也同样提供了单独使能定时器的输出通道函数，函数为：

```
void TIM_CCxChannelCmd(TIM_TypeDef *TIMx, uint32_t Channel,
uint32_t ChannelState);
```

HAL_TIM_OC_Start 函数内部也调用了该函数。

定时器输出比较模式配置步骤

1) 开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1、2、3、4，对应 IO 是 PC6/PC7/PC8/PC9，它们的时钟开启方法如下：

```
HAL_RCC_TIM8_CLK_ENABLE(); /* 使能定时器 8 */
HAL_RCC_GPIOC_CLK_ENABLE(); /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx, 设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的输出比较模式时, 我们调用的是 HAL_TIM_OC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意: 该函数会调用 HAL_TIM_OC_MspInit 函数, 我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3) 设置定时器为输出比较模式, 输出比较极性, 输出比较值、翻转功能等参数。

在 HAL 库中, 通过 HAL_TIM_OC_ConfigChannel 函数来设置定时器为输出比较模式, 根据需求设置输出比较的极性, 设置输出比较值、翻转功能等。

最后我们通过 __HAL_TIM_ENABLE_OCxPRELOAD 函数使能通道的预装载。

4) 开启定时器并输出 PWM

通过 HAL_TIM_OC_Start 函数使能定时器并开启输出。

22.3.3.2 程序流程图

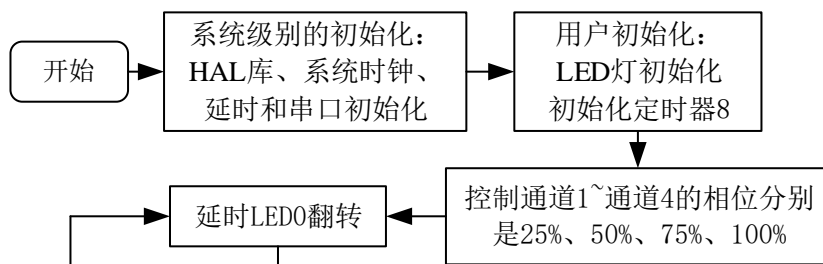


图 22.3.3.2.1 高级定时器输出比较模式实验程序流程图

22.3.3.3 程序解析

这里我们只讲解核心代码, 详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件: atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义:

```

/* TIMX 输出比较模式 定义
 * 这里通过 TIM8 的输出比较模式,控制 PC6,PC7,PC8,PC9 输出 4 路 PWM, 占空比 50%, 并且每一路 PWM
 * 之间的相位差为 25%, 修改 CCRx 可以修改相位. 默认是针对 TIM8
 * 注意: 通过修改这些宏定义,可以支持 TIM1/TIM8 任意一个定时器,任意一个 IO 口使用输出比较模式,
 * 输出 PWM
 */
#define ATIM_TIMX_COMP_CH1_GPIO_PORT      GPIOC
#define ATIM_TIMX_COMP_CH1_GPIO_PIN      GPIO_PIN_6
#define ATIM_TIMX_COMP_CH1_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0) /* PC 口时钟使能 */

#define ATIM_TIMX_COMP_CH2_GPIO_PORT      GPIOC
#define ATIM_TIMX_COMP_CH2_GPIO_PIN      GPIO_PIN_7
#define ATIM_TIMX_COMP_CH2_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0) /* PC 口时钟使能 */

#define ATIM_TIMX_COMP_CH3_GPIO_PORT      GPIOC
#define ATIM_TIMX_COMP_CH3_GPIO_PIN      GPIO_PIN_8
#define ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0) /* PC 口时钟使能 */

#define ATIM_TIMX_COMP_CH4_GPIO_PORT      GPIOC
#define ATIM_TIMX_COMP_CH4_GPIO_PIN      GPIO_PIN_9
#define ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0) /* PC 口时钟使能 */

#define ATIM_TIMX_COMP                    TIM8
#define ATIM_TIMX_COMP_CH1_CCRX          ATIM_TIMX_COMP->CCR1 /* 通道 1 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CH2_CCRX          ATIM_TIMX_COMP->CCR2 /* 通道 2 的输出比较寄存器 */
  
```

```
#define ATIM_TIMX_COMP_CH3_CCRX ATIM_TIMX_COMP->CCR3 /* 通道 3 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CH4_CCRX ATIM_TIMX_COMP->CCR4 /* 通道 4 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CLK_ENABLE()
do{ __HAL_RCC_TIM8_CLK_ENABLE();}while(0) /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 1 输出通道 1~通道 4 对应的 IO 口的宏定义。第二部分则是定时器 8 的相应宏定义。

下面看到 `atim.c` 文件的程序，首先是高级定时器输出比较模式初始化函数，其定义如下：

```
/**
 * @brief      高级定时器 TIMX 输出比较模式 初始化函数（使用输出比较模式）
 * @note
 *
 *      配置高级定时器 TIMX 4 路输出比较模式 PWM 输出, 实现 50% 占空比, 不同相位控制
 *      注意, 本例程输出比较模式, 每 2 个计数周期才能完成一个 PWM 输出, 因此输出频率减半
 *      另外, 我们还可以开启中断在中断里面修改 CCRx, 从而实现不同频率/不同相位的控制
 *      但是我们不推荐这么使用, 因为这可能导致非常频繁的中断, 从而占用大量 CPU 资源
 *
 *      高级定时器的时钟来自 APB2, 而 PCLK2 = 72Mhz, 我们设置 PPRE2 不分频, 因此
 *      高级定时器时钟 = 72Mhz
 *      定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft=定时器工作频率, 单位:Mhz
 *
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void atim_timx_comp_pwm_init(uint16_t arr, uint16_t psc)
{
    TIM_OC_InitTypeDef timx_oc_comp_pwm = {0};

    g_timx_comp_pwm_handle.Instance = ATIM_TIMX_COMP; /* 定时器 x */
    g_timx_comp_pwm_handle.Init.Prescaler = psc; /* 定时器分频 */
    g_timx_comp_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_comp_pwm_handle.Init.Period = arr; /* 自动重装载值 */
    g_timx_comp_pwm_handle.Init.AutoReloadPreload =
        TIM_AUTORELOAD_PRELOAD_ENABLE; /* 使能影子寄存器 TIMx_ARR */
    HAL_TIM_OC_Init(&g_timx_comp_pwm_handle); /* 输出比较模式初始化 */

    timx_oc_comp_pwm.OCMode = TIM_OCMode_TOGGLE; /* 比较输出模式翻转功能 */
    timx_oc_comp_pwm.Pulse = 250 - 1; /* 设置输出比较寄存器的值 */
    timx_oc_comp_pwm.OCpolarity = TIM_OCPOLARITY_HIGH; /* 输出比较极性为高 */
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &timx_oc_comp_pwm,
        TIM_CHANNEL_1); /* 初始化定时器的输出比较通道 1 */

    /* CCR1 寄存器预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_1);

    tim_oc_handle.Pulse = 500;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &tim_oc_handle,
        TIM_CHANNEL_2); /* 初始化定时器的输出比较通道 2 */

    /* CCR2 寄存器预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_2);

    tim_oc_handle.Pulse = 750;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &tim_oc_handle,
        TIM_CHANNEL_3); /* 初始化定时器的输出比较通道 3 */

    /* CCR3 寄存器预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_3);
    tim_oc_handle.Pulse = 1000;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &tim_oc_handle,
        TIM_CHANNEL_4); /* 初始化定时器的输出比较通道 4 */

    /* CCR4 寄存器预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_4);
}
```

```
HAL_TIM_OC_Start(&g_timx_comp_pwm_handle, TIM_CHANNEL_1);
HAL_TIM_OC_Start(&g_timx_comp_pwm_handle, TIM_CHANNEL_2);
HAL_TIM_OC_Start(&g_timx_comp_pwm_handle, TIM_CHANNEL_3);
HAL_TIM_OC_Start(&g_timx_comp_pwm_handle, TIM_CHANNEL_4);
}
```

在 `atim_timx_comp_pwm_init` 函数中，首先调用 `HAL_TIM_OC_Init` 函数初始化定时器的 ARR 和 PSC 等参数。然后通过调用函数 `HAL_TIM_OC_ConfigChannel` 设置通道 1~通道 4 的工作参数，包括：输出比较模式功能、输出比较寄存器的值，输出极性等。接着调用 `__HAL_TIM_ENABLE_OCxPRELOAD` 函数宏使能 CCR1/2/3/4 寄存器的预装载。最后通过调用函数 `HAL_TIM_OC_Start` 来使能 TIM1 通道 1~通道 4 输出。

`HAL_TIM_OC_Init` 函数会调用 `HAL_TIM_OC_MspInit` 回调函数，我们把使能定时器和通道对应的 IO 时钟、IO 初始化的代码存放到该函数里，其定义如下：

```
/**
 * @brief      定时器底层驱动，时钟使能，引脚配置
 *             此函数会被 HAL_TIM_OC_Init() 调用
 * @param      htim:定时器句柄
 * @retval     无
 */
void HAL_TIM_OC_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_COMP)
    {
        GPIO_InitTypeDef gpio_init_struct;

        ATIM_TIMX_COMP_CLK_ENABLE();

        ATIM_TIMX_COMP_CH1_GPIO_CLK_ENABLE();
        ATIM_TIMX_COMP_CH2_GPIO_CLK_ENABLE();
        ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE();
        ATIM_TIMX_COMP_CH4_GPIO_CLK_ENABLE();

        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH1_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;
        gpio_init_struct.Pull = GPIO_NOPULL;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH1_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH2_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH2_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH3_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH3_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH4_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH4_GPIO_PORT, &gpio_init_struct);
    }
}
```

该函数主要是使能定时器和通道对应的 IO 时钟，初始化 IO 口。

在 `main.c` 里面编写如下代码：

```
int main(void)
{
    uint8_t t = 0;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    atim_timx_comp_pwm_init(1000 - 1, 72 - 1); /* 1Mhz 的计数频率 1Khz 的周期。 */
    ATIM_TIMX_COMP_CH1_CCRX = 250 - 1; /* 通道 1 相位 25% */
    ATIM_TIMX_COMP_CH2_CCRX = 500 - 1; /* 通道 2 相位 50% */
    ATIM_TIMX_COMP_CH3_CCRX = 750 - 1; /* 通道 3 相位 75% */
}
```

```
ATIM_TIMX_COMP_CH4_CCRX = 999 - 1; /* 通道 4 相位 99% */
while (1)
{
    delay_ms(10);
    t++;
    if (t >= 20)
    {
        LED0_TOGGLE(); /* LED0 (RED) 闪烁 */
        t = 0;
    }
}
```

本小节开头我们讲解了输出比较模式翻转功能如何产生 PWM 波，下面结合程序一起计算出 PWM 波的周期，频率等参数。

定时器 8 时钟源的时钟频率等于 APB2 总线时钟频率，即 72MHz，而调用 `atim_timx_comp_pwm_init(1000 - 1, 72 - 1)` 初始化函数之后，就相当于写入预分频寄存器的值为 71，写入自动重载寄存器的值为 999。将这些参数代入本小节介绍的翻转功能输出的 PWM 周期计算公式，可得：

$$T = 2 * (arr + 1) * ((psc + 1) / Tclk) = 2 * (999 + 1) * ((71 + 1) / 72000000) = 0.002s$$

由上述式子得到 PWM 周期为 2ms，频率为 500Hz。ARR 值为固定为 1000，所以占空比则固定为 50%。定时器 8 通道 1~通道 4 输出的 PWM 波的相位分别是：25%、50%、75%、100%。

22.3.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了。我们需要借助示波器观察 PC6、PC7、PC8 和 PC9 引脚 PWM 输出的情况，如下图所示：



图 22.3.4.1 相位为 25%、50%、75%、100% 的 PWM 波

图 22.3.4.1 中，由上到下分别是引脚 PE9、PE11、PE13 和 PE14 输出的 PWM，即分别对应的是 TIM8_CH1、TIM8_CH2、TIM8_CH3 和 TIM8_CH4 输出的相位为 25%、50%、75% 和 100% 的 PWM。大家可以把其中一个通道的捕获/比较寄存器的值设置为 0，那么就可以得到 PWM 初相位的波形，即相位为 0%。

22.4 高级定时器互补输出带死区控制实验

本小节我们来学习使用高级定时器的互补输出带死区控制功能。对于刚接触这个知识的朋友可能会问：什么是互补输出？还带死区控制？What？下面给大家简单说一下。

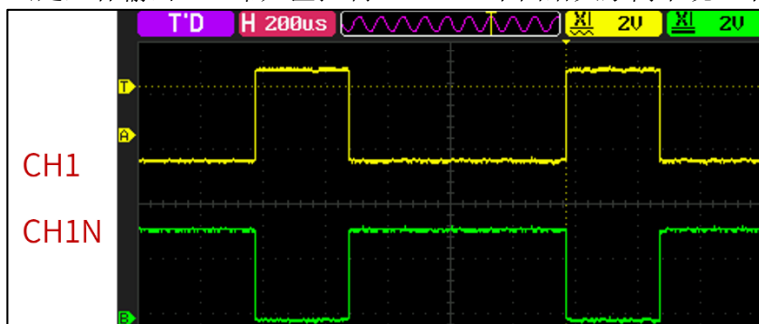


图 22.4.1 互补输出

上图中，CH1 输出黄色的 PWM，它的互补通道 CH1N 输出绿色的 PWM。通过对比，可以知道这两个 PWM 刚好是反过来的，CH1 的 PWM 为高电平期间，CH1N 的 PWM 则是低电平，反之亦然，这就是互补输出。

下面来看一下什么是带死区控制的互补输出？

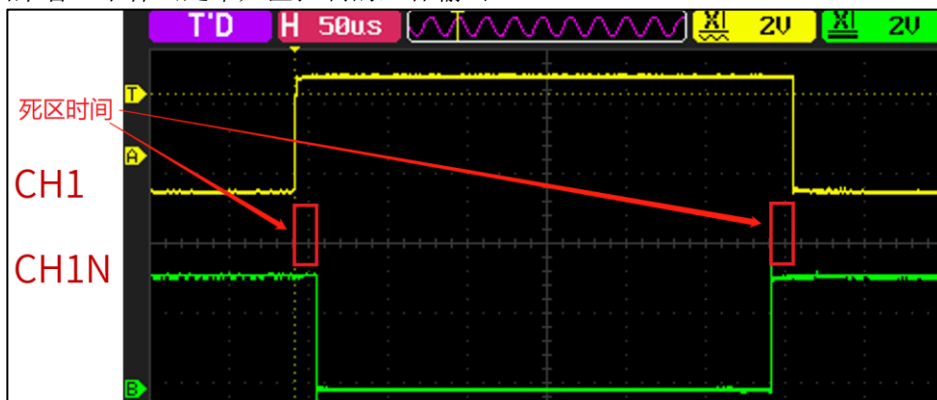


图 22.4.2 带死区控制的互补输出

上图中，CH1 输出的 PWM 和 CH1N 输出的 PWM 在高低电平转换间，插入了一段时间才实现互补输出。这段时间称为死区时间，可以通过 DTG[7:0]位配置控制死区时间的长度，后面会详细讲解如何配置死区时间。上图中，箭头指出的两段死区时间的长度是一样的，因为都是由同一个死区发生器产生。

理解了互补输出和带死区控制的互补输出，下面来看一下带死区控制的互补输出有什么用？带死区控制的互补输出经常被用于控制电机的 H 桥中，下面给大家画了一个 H 桥的简图：

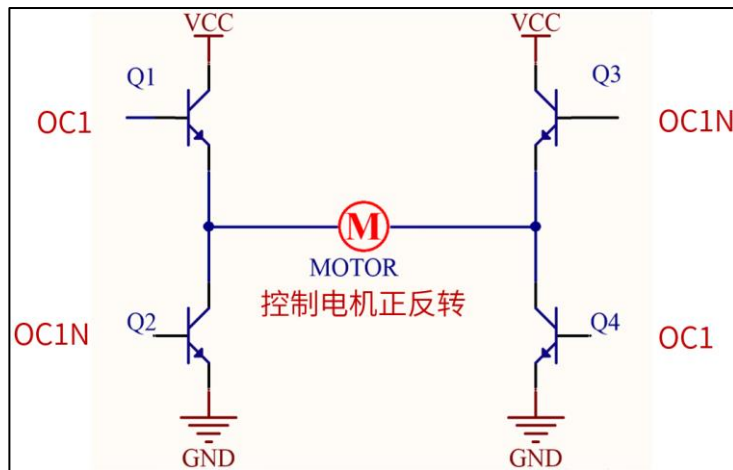


图 22.4.3 H 桥简图

图 22.4.3 是 H 桥的简图，实际控制电机正反转的 H 桥会根据复杂些，而且更多的是使用 MOS 管，这里只是为了解释带死区控制的互补输出在 H 桥中的控制逻辑原理，大家理解原理就行。上图的 H 桥搭建全部使用的是 NPN，并且导通逻辑都是基极为高电平时导通。如果 Q1 和 Q4 三极管导通，那么电机的电流方向是从左到右（假设电机正转）；如果 Q2 和 Q3 三极管导通，那么电机的电流方向是从右到左（假设电机反转）。上述就是 H 桥控制电机正反转的逻辑原理。但是同一侧的三极管是不可以同时导通的，否则会短路，比如：Q1 和 Q2 同时导通或者 Q3 和 Q4 同时导通，这都是不可取的。

下面大家想一下图 22.4.1 的 OC1（CH1）和 OC1N（CH1N）输出的 PWM 输入到图 22.4.3 的 H 桥中，会怎样？按理来说应该是 OC1N 输出高电平的时候，OC1 输出就是低电平，刚好 Q2 和 Q3 导通，电机的电流方向是从右到左（假设电机反转）；反之，OC1 输出高电平的时候，OC1N 输出就是低电平，刚好 Q1 和 Q4 导通，电机的电流方向是从左到右（假设电机正转），这似乎已经完美解决电机正反转问题了。实际上，元器件是有延迟特性的，比如：控制信号从 OC1 传导至电机，是要经过一定的时间的，复杂的 H 桥电路更是如此。由于元器件特性，就会导致直接使用互补输出信号驱动 H 桥时存在短路现象。为了避免这种情况，于是就有了带死区控制的互补输出来驱动 H 桥电路。如图 22.4.2 的死区时间就是为了解决元器件延迟特性的。用户必须根据与输出相连接的器件及其特性（电平转换器的固有延迟、开关器件产生的延迟）来调整死区时间。

死区时间计算

下面来看一下定时器的死区时间是怎么计算并设置的？死区时间是由 TIMx_CR1 寄存器的 CKD[1:0]位和 TIMx_BDTR 寄存器的 DTG[7:0]位来设置，如下图所示：

TIMx_CR1

CKD[1:0]: 时钟分频 (Clock division)

此位域指示定时器时钟 (CK_INT) 频率与死区发生器以及数字滤波器 (ETR、Tlx) 所使用的死区及采样时钟 (t_{DTS}) 之间的分频比，

00: $t_{DTS}=t_{CK_INT}$
 01: $t_{DTS}=2*t_{CK_INT}$
 10: $t_{DTS}=4*t_{CK_INT}$
 11: 保留，不要设置成此值

TIMx_BDTR

DTG[7:0]: 配置死区发生器 (Dead-time generator setup)

此位域定义插入到互补输出之间的死区持续时间。DT 与该持续时间相对应。

DTG[7:5]=0xx => $DT=DTG[7:0]*t_{dtg}$ ，其中 $t_{dtg}=t_{DTS}$ 。
 DTG[7:5]=10x => $DT=(64+DTG[5:0])*t_{dtg}$ ，其中 $t_{dtg}=2*t_{DTS}$ 。
 DTG[7:5]=110 => $DT=(32+DTG[4:0])*t_{dtg}$ ，其中 $t_{dtg}=8*t_{DTS}$ 。
 DTG[7:5]=111 => $DT=(32+DTG[4:0])*t_{dtg}$ ，其中 $t_{dtg}=16*t_{DTS}$ 。

示例：如果 $T_{DTS}=125ns$ (8MHz)，则可能的死区值为：
 0 到 15875 ns（步长为 125 ns）
 16 us 到 31750 ns（步长为 250 ns）
 32 us 到 63us（步长为 1 us）
 64 us 到 126 us（步长为 2 us）

图 22.4.4 CKD[1:0]和 DTG[7:0]位

死区时间计算分三步走：

第一步：通过 CKD[1:0]位确定 t_{DTS} 。根据 CKD[1:0]位的描述，可以得到下面的式子：

$$t_{DTS} = \frac{2^{CKD[1:0]}}{T_{clk}}$$

其中：

CKD[1:0]: CKD[1:0]位设置的值。

T_{clk} : 定时器的时钟源频率（单位为 MHz）。

假设定时器时钟源频率是 72MHz，我们设置 CKD[1:0]位的值为 2，代入上面的式子可得：

$$t_{DTS} = \frac{2^{CKD[1:0]}}{T_{clk}} = \frac{2^2}{72000000} = \frac{4}{72000000} = 55.56ns$$

通过上式可得 t_{DTS} 约等于 55.56ns，本实验例程中我们也是这样设置的。

第二步：根据 DTG[7:5]选择计算公式。

第三步：代入选择的公式计算。

下面给大家举个例子，假设定时器时钟源频率是 72MHz，我们设置 CKD[1:0]位的值为 2，DTG[7:0]位的值为 250。从上面的例子知道 CKD[1:0]位的值为 2，得到的 $t_{DTS}=55.56ns$ 。下面来看一下 DTG[7:0]位的值为 250，应该选择 DTG[7:0]位描述中哪条公式？250 的二进制数为 11111010，即 DTG[7:5]为 111，所以选择第四条公式： $DT=(32+DTG[4:0]) * t_{dtg}$ ，其中 $t_{dtg} = 16 * t_{DTS}$ 。可以看到手册上的式子符号大小写乱乱的，这里大小写不敏感。由手册的公式可以得到 $DT = (32+DTG[4:0]) * 16 * t_{DTS} = (32+26) * 16 * 55.56ns = 51559.68ns = 51.56\mu s$ ，即死区时间为 51.56us。死区时间计算方法就给大家介绍到这里。

关于互补输出和死区插入的更多内容请看《STM32F10xxx 参考手册_V10（中文版）.pdf》手册的 13.3.11 小节，下面我们介绍相关的寄存器。

22.4.1 TIM1/TIM8 寄存器

高级定时器互补输出带死区控制除了用到定时器的时基单元：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) 之外。主要还用到以下这些寄存器：

● 控制寄存器 1 (TIMx_CR1)

TIM1/TIM8 的控制寄存器 1 描述如图 22.4.1.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----------|------|----------|-----|-----|-----|------|-----|----|---|
| 保留 | | | | | | CKD[1:0] | ARPE | CMS[1:0] | DIR | OPM | URS | UDIS | CEN | | |
| rW | | | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | |

图 22.4.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的一些位，其中：位 7 (APRE) 用于控制自动重载寄存器是否进行缓冲，在基本定时器的时候已经讲过，请回顾。本实验中，我们把该位置 1。

CKD[1:0]位指示定时器时钟 (CK_INT) 频率与死区发生器以及数字滤波器 (ETR、Tlx) 所使用的死区及采样时钟 (t_{DTS}) 之间的分频比。我们设置 CKD[1:0]位为 10，结合高级定时器时钟源频率等于 APB2 总线时钟频率，即 72MHz，可以得到 $t_{DTS}=55.56ns$ 。

CEN 位，用于使能计数器的工作，必须要设置该位为 1，才可以开始计数。

● 捕获/比较模式寄存器 1/2 (TIMx_CCMR1/2)

TIM1/TIM8 的捕获/比较模式寄存器 (TIMx_CCMR1/2)，该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 22.4.1.2 所示：

| 15 | | 14 | | 13 | | 12 | | 11 | | 10 | | 9 | | 8 | | 7 | | 6 | | 5 | | 4 | | 3 | | 2 | | 1 | | 0 | |
|-----------|--|-----------|--|----|--|-------------|--|-------|--|-----------|--|----|--|-----------|--|-----------|--|-------------|--|-------|--|-------|--|-----------|--|----|--|----|--|---|--|
| OC2CE | | OC2M[2:0] | | | | OC2PE | | OC2FE | | CC2S[1:0] | | | | OC1CE | | OC1M[2:0] | | | | OC1PE | | OC1FE | | CC1S[1:0] | | | | | | | |
| IC2F[3:0] | | | | | | IC2PSC[1:0] | | | | | | | | IC1F[3:0] | | | | IC1PSC[1:0] | | | | | | | | | | | | | |
| rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | rW | | | |

图 22.4.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在用到输出比较模式。关于该寄存器的详细说明，请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》第 240 页。

本实验我们用到了定时器 1 输出比较的通道 1，所以我们需要配置 TIM1_CCMR1 模式设置位 OC1M[2:0]，我们使用的是 PWM 模式 1，所以这 3 位必须设置为 110。

● 捕获/比较使能寄存器 (TIMx_CCER)

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 22.4.1.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|------|------|-------|-------|------|------|-------|-------|------|------|-------|-------|------|------|----|
| 保留 | CC4P | CC4E | CC3NP | CC3NE | CC3P | CC3E | CC2NP | CC2NE | CC2P | CC2E | CC1NP | CC1NE | CC1P | CC1E | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

图 22.4.1.3 TIMx_CCER 寄存器

该寄存器比较简单，要让 TIM1 的通道 1 输出，我们需要把对应的捕获/比较 1 输出使能位 CC1E 置 1。因为本实验中，我们需要实现互补输出，所以还需要把 CC1NE 位置 1，使能互补通道输出。CC1P 和 CC1NP 分别是通道 1 输出和通道 1 互补输出的极性设置位。这里我们把 CC1P 和 CC1NP 位都置 1，即输出极性为低，就可以得到互补的 PWM。

● 捕获/比较寄存器 1/2/3/4 (TIMx_CCR1/2/3/4)

捕获/比较寄存器 (TIMx_CCR1/2/3/4)，该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器描述如图 22.4.1.4 所示：

| | | | | | | | | | | | | | | | |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CCR1[15:0] | | | | | | | | | | | | | | | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |

图 22.4.1.4 TIMx_CCR1 寄存器

该寄存器 16 位有效位，本实验中可以通过改变该寄存器的值来改变 PWM 的占空比。

● 断路和死区寄存器 (TIMx_BDTR)

TIM1/TIM8 断路和死区寄存器，该寄存器各位描述如图 22.4.1.5 所示：

| | | | | | | | | | | | | | | | |
|-----|----|-----|----|-----|----|-----|----|------|----|------|----|-----------|----|----------|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MOE | | AOE | | BKP | | BKE | | OSSR | | OSSI | | LOCK[1:0] | | DTG[7:0] | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |

注释：根据锁定设置，AOE、BKP、BKE、OSSI、OSSR 和 DTG[7:0] 位均可被写保护，有必要在第一次写入 TIMx_BDTR 寄存器时对它们进行配置。

| | |
|-------|--|
| 位 14 | AOE : 自动输出使能 (Automatic output enable) 0: MOE 只能被软件置 '1'; 1: MOE 能被软件置 '1' 或在下一个更新事件被自动置 '1' (如果刹车输入无效)。 注：一旦 LOCK 级别 (TIMx_BDTR 寄存器中的 LOCK 位) 设为 '1'，则该位不能被修改。 |
| 位 13 | BKP : 刹车输入极性 (Break polarity) 0: 刹车输入低电平有效; 1: 刹车输入高电平有效。 注：一旦 LOCK 级别 (TIMx_BDTR 寄存器中的 LOCK 位) 设为 '1'，则该位不能被修改。 注：任何对该位的写操作都需要一个 APB 时钟的延迟以后才能起作用。 |
| 位 12 | BKE : 刹车功能使能 (Break enable) 0: 禁止刹车输入 (BRK 及 CCS 时钟失效事件); 1: 开启刹车输入 (BRK 及 CCS 时钟失效事件)。 注：当设置了 LOCK 级别 1 时 (TIMx_BDTR 寄存器中的 LOCK 位)，该位不能被修改。 注：任何对该位的写操作都需要一个 APB 时钟的延迟以后才能起作用。 |
| 位 7:0 | UTG[7:0] : 死区发生器设置 (Dead-time generator setup) 这些位定义了插入互补输出之间的死区持续时间。假设 DT 表示其持续时间： $DTG[7:5]=0xx \Rightarrow DT=DTG[7:0] \times T_{dtg}, T_{dtg} = T_{DTS};$ $DTG[7:5]=10x \Rightarrow DT=(64+DTG[5:0]) \times T_{dtg}, T_{dtg} = 2 \times T_{DTS};$ $DTG[7:5]=110 \Rightarrow DT=(32+DTG[4:0]) \times T_{dtg}, T_{dtg} = 8 \times T_{DTS};$ $DTG[7:5]=111 \Rightarrow DT=(32+DTG[4:0]) \times T_{dtg}, T_{dtg} = 16 \times T_{DTS};$ 例：若 $T_{DTS} = 125ns(8MHZ)$ ，可能的死区时间为： 0 到 15875ns，若步长为 125ns; 16us 到 31750ns，若步长为 250ns; 32us 到 63us，若步长为 1us; 64us 到 126us，若步长为 2us; 注：一旦 LOCK 级别 (TIMx_BDTR 寄存器中的 LOCK 位) 设为 1、2 或 3，则不能修改这些位。 |

图 22.4.1.5 TIMx_BDTR 寄存器

该寄存器控制定时器的断路和死区控制的功能。我们先看断路控制，用到断路输入功能（断路输入引脚为 PE15），位 BKE 置 1 即可。

位 BKP 选择断路输入信号有效电平。本实验中，我们选择高电平有效，即 BKP 置 1。

位 AOE 是自动输出使能位，如果使能 AOE 位，那么在我们输入刹车信号后再断开了刹车信号，互补的 PWM 会自动恢复输出，如果失能 AOE 位，那么在输入刹车信号后再断开了刹车

信号，互补的 PWM 就不会恢复输出，而是一直保持刹车信号输入时的状态。为了方便观察，我们使能该位，即置 1。

位 MOE 是使能主输出，想要高级定时器的通道正常输出，则必须设置 MOE 位为 1。

最后是 DTG[7:0]位，用于设置死区时间，前面已经教过大家怎么设置了。这里以我们例程的设置为例，CKD[1:0] 设置为 10，定时器时钟源频率是 72MHz，所以 $t_{DTS} = 55.56ns$ 。

本例程的 DTG[7:0]位的值设置为十进制 100，即二进制数 0110 0100。DTG[7:5]=011，符合第一条式子： $DT = DTG[7:0] * t_{dtg}$ ，其中 $t_{dtg} = t_{DTS}$ 。DT 是死区时间，可以得到 $DT = 100 * 55.56ns = 5.56us$ 。到后面下载验证小节，我们通过示波器验证一下这个死区时间计算的理论和实际值是否一样。

22.4.2 硬件设计

1. 例程功能

1, 利用 TIM1_CH1(PE9)输出 70%占空比的 PWM 波, 它的互补输出通道(PE8)则是输出 30%占空比的 PWM 波。

2, 刹车功能, 当给刹车输入引脚(PE15)输入高电平时, 进行刹车, 即 PE8 和 PE9 停止输出 PWM 波。

3, LED0 闪烁指示程序运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) 定时器 1

TIM1 正常输出通道 PE9

TIM1 互补输出通道 PE8

TIM1 刹车输入 PE15

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们需要通过示波器观察 PE8 和 PE9 引脚 PWM 输出的情况。还可以通过给 PE15 引脚接入高电平进行刹车。

22.4.3 程序设计

22.4.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIMEx_ConfigBreakDeadTime 函数

定时器的断路和死区时间配置初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIMEx_ConfigBreakDeadTime(TIM_HandleTypeDef *htim,
                                                    TIM_BreakDeadTimeConfigTypeDef *sBreakDeadTimeConfig);
```

● 函数描述：

用于初始化定时器的断路（即刹车）和死区时间。

● 函数形参：

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。

形参 2 是 TIM_BreakDeadTimeConfigTypeDef 结构体类型指针变量，用于配置断路和死区参数，其定义如下：

```
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;          /* 寄存器锁定配置 */
uint32_t DeadTime;          /* 死区时间设置 */
uint32_t BreakState;        /* 断路（即刹车）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 AutomaticOutput;   /* 自动恢复输出使能控制 */
} TIM_BreakDeadTimeConfigTypeDef;
```

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIMEx_PWMN_Start 函数

定时器的互补输出启动函数。其声明如下:

```
HAL_StatusTypeDef HAL_TIMEx_PWMN_Start(TIM_HandleTypeDef *htim,
                                       uint32_t Channel);
```

● 函数描述:

该函数用于启动定时器的互补输出。

● 函数形参:

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。

形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

定时器互补输出带死区控制配置步骤

1) 开启 TIMx 和通道输出以及刹车输入的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 1 通道 1，对应 IO 是 PE9，互补输出通道引脚是 PE8，刹车输入引脚是 PE15，它们的时钟开启方法如下:

```
HAL_RCC_TIM1_CLK_ENABLE();          /* 使能定时器 1 */
HAL_RCC_GPIOE_CLK_ENABLE();         /* 开启 GPIOE 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

这里我们要使用定时器的 PWM 模式功能，所以调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。注意:本实验要使用该函数配置 TIMx_CR1 寄存器的 CKD[1:0] 位，从而确定 tDTS，方便后续设置死区时间。

注意:该函数会调用:HAL_TIM_PWM_MspInit 函数，但是为不跟前面的实验共用该回调函数，提高独立性，我们就直接在 atim_timx_cplm_pwm_init 函数中，使能定时器时钟和 GPIO 时钟，初始化通道对应 IO 引脚等。

3) 设置定时器为 PWM 模式，输出比较极性，互补输出极性等参数

通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式，根据需求设置 OCy 输出极性和 OCyN 互补输出极性等。

4) 设置死区参数

通过 HAL_TIMEx_ConfigBreakDeadTime 函数来设置死区参数，比如:设置死区时间、运行模式的关闭输出状态、空闲模式的关闭输出状态、刹车输入有效信号极性和是否允许刹车后自动恢复输出等。

5) 启动 OCy 输出以及 OCyN 互补输出

通过 HAL_TIM_PWM_Start 函数启动 OCy 输出，通过 HAL_TIMEx_PWMN_Start 函数启动启动 OCyN 互补输出。

22.4.3.2 程序流程图

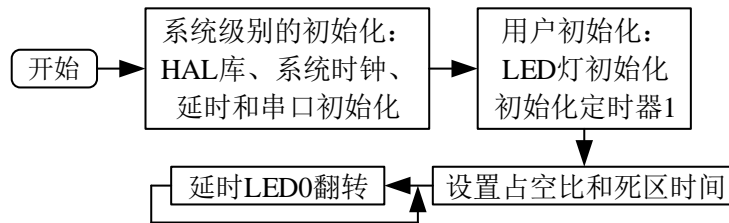


图 22.4.3.2.1 高级定时器互补输出带死区控制实验

22.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```

/*****
/* TIMX 互补输出模式 定义
* 这里设置互补输出相关硬件配置，CHY 即正常输出，CHYN 即互补输出
* 修改 CCRx 可以修改占空比。
* 默认是针对 TIM1
* 注意：通过修改这些宏定义，可以支持 TIM1/TIM8 定时器，任意一个 IO 口输出互补 PWM(前提是必须有互补输出功能)
*/

/* 输出通道引脚 */
#define ATIM_TIMX_CPLM_CHY_GPIO_PORT      GPIOE
#define ATIM_TIMX_CPLM_CHY_GPIO_PIN      GPIO_PIN_9
#define ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0) /* PE 口时钟使能 */

/* 互补输出通道引脚 */
#define ATIM_TIMX_CPLM_CHYN_GPIO_PORT      GPIOE
#define ATIM_TIMX_CPLM_CHYN_GPIO_PIN      GPIO_PIN_8
#define ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0) /* PE 口时钟使能 */

/* 刹车输入引脚 */
#define ATIM_TIMX_CPLM_BKIN_GPIO_PORT      GPIOE
#define ATIM_TIMX_CPLM_BKIN_GPIO_PIN      GPIO_PIN_15
#define ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0) /* PE 口时钟使能 */

/* TIMX REMAP 设置
* 因为 PE8/PE9/PE15，默认并不是 TIM1 的复用功能脚，必须开启完全重映射，才可以将：
TIM1_CH1->PE9; TIM1_CH1N->PE8; TIM1_BKIN->PE15;
* 这样，PE8/PE9/PE15，才能用作 TIM1 的 CH1N/CH1/BKIN 功能。
* 所以必须实现 ATIM_TIMX_CPLM_CHYN_GPIO_REMAP，通过 sys_gpio_remap_set 函数设置重映射
* 如果我们使用默认的复用功能输出，则不用设置重映射，是可以不需要该函数的！根据具体需要来实现。
*/
#define ATIM_TIMX_CPLM_CHYN_GPIO_REMAP() do{ __HAL_RCC_AFIO_CLK_ENABLE();\
__HAL_AFIO_REMAP_TIM1_ENABLE();\
}while(0)

/* 互补输出使用的定时器 */
#define ATIM_TIMX_CPLM      TIM1
#define ATIM_TIMX_CPLM_CHY      TIM_CHANNEL_1
#define ATIM_TIMX_CPLM_CHY_CCR1      ATIM_TIMX_CPLM->CCR1
#define ATIM_TIMX_CPLM_CLK_ENABLE()
do{ __HAL_RCC_TIM1_CLK_ENABLE(); }while(0) /* TIM1 时钟使能 */
*****/

```

可以把上面的宏定义分成两部分，第一部分包括是定时器 1 输出、互补输出和刹车输入通道对应的 IO 口的宏定义，第二部分则是定时器 1 的相应宏定义。注意：因为 PE8/PE9/PE15，默认并不是 TIM1 的复用功能脚，必须开启完全重映射，具体请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》第 123 页，AFIO_MAPR 寄存器的描述。

下面来看 atim.c 文件的程序，首先是高级定时器互补输出初始化函数，其定义如下：

```
/**
 * @brief      高级定时器 TIMX 互补输出 初始化函数（使用 PWM 模式 1）
 * @note
 *
 *      配置高级定时器 TIMX 互补输出，一路 OCy 一路 OCyN，并且可以设置死区时间
 *
 *      高级定时器的时钟来自 APB2，而 PCLK2 = 72Mhz，我们设置 PPRE2 不分频，因此
 *      高级定时器时钟 = 72Mhz
 *      定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft=定时器工作频率,单位:Mhz
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval      无
 */

void atim_timx_cplm_pwm_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct = {0};
    TIM_OC_InitTypeDef tim_oc_cplm_pwm = {0};

    ATIM_TIMX_CPLM_CLK_ENABLE();          /* TIMx 时钟使能 */
    ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE(); /* 通道 x 对应 IO 口时钟使能 */
    ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE(); /* 通道 x 互补通道对应 IO 口时钟使能 */
    ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE(); /* 通道 x 刹车输入对应 IO 口时钟使能 */

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHY_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(ATIM_TIMX_CPLM_CHY_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHYN_GPIO_PIN;
    HAL_GPIO_Init(ATIM_TIMX_CPLM_CHYN_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_BKIN_GPIO_PIN;
    HAL_GPIO_Init(ATIM_TIMX_CPLM_BKIN_GPIO_PORT, &gpio_init_struct);

    ATIM_TIMX_CPLM_CHYN_GPIO_REMAP();      /* 重映射定时器 IO */

    g_timx_cplm_pwm_handle.Instance = ATIM_TIMX_CPLM; /* 定时器 x */
    g_timx_cplm_pwm_handle.Init.Prescaler = psc;      /* 定时器预分频系数 */
    g_timx_cplm_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_cplm_pwm_handle.Init.Period = arr;        /* 自动重装值 */
    /* CKD[1:0] = 10, tDTS = 4 * tCK_INT = Ft / 4 = 18Mhz */
    g_timx_cplm_pwm_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV4;
    g_timx_cplm_pwm_handle.Init.AutoReloadPreload =
        TIM_AUTORELOAD_PRELOAD_ENABLE; /* 使能影子寄存器 TIMx_ARR */
    HAL_TIM_PWM_Init(&g_timx_cplm_pwm_handle);

    tim_oc_cplm_pwm.OCMode = TIM_OCMode_PWM1;        /* PWM 模式 1 */
    tim_oc_cplm_pwm.OCpolarity = TIM_OCPOLARITY_LOW; /* OCy 低电平有效 */
    tim_oc_cplm_pwm.OCNPolarity = TIM_OCNPOLARITY_LOW; /* OCyN 低电平有效 */
    tim_oc_cplm_pwm.OCIdleState = TIM_OCIDLESTATE_SET; /* 当 MOE=0, OCx=1 */
    tim_oc_cplm_pwm.OCNIdleState = TIM_OCNIDLESTATE_SET; /* 当 MOE=0, OCxN=1 */
    HAL_TIM_PWM_ConfigChannel(&g_timx_cplm_pwm_handle, &tim_oc_cplm_pwm,
        ATIM_TIMX_CPLM_CHY);
    /* 设置死区参数，开启死区中断 */
}
```

```

/* 运行模式的关闭输出状态 */
g_sbreak_dead_time_config.OffStateRunMode = TIM_OSSR_DISABLE;
/* 空闲模式的关闭输出状态 */
g_sbreak_dead_time_config.OffStateIDLEMode = TIM_OSSI_DISABLE;
g_sbreak_dead_time_config.LockLevel = TIM_LOCKLEVEL_OFF; /* 不用寄存器锁功能 */
g_sbreak_dead_time_config.BreakState = TIM_BREAK_ENABLE; /* 使能刹车输入 */
/* 刹车输入有效信号极性为高 */
g_sbreak_dead_time_config.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
/* 使能 AOE 位, 允许刹车结束后自动恢复输出 */
g_sbreak_dead_time_configAutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
HAL_TIMEx_ConfigBreakDeadTime(&g_timx_cplm_pwm_handle,
                               &g_sbreak_dead_time_config);

/* 使能 OCy 输出 */
HAL_TIM_PWM_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
/* 使能 OCyN 输出 */
HAL_TIMEx_PWMN_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
}

```

在 `atim_timx_cplm_pwm_init` 函数中, 没有使用 HAL 库的 MSP 回调, 而是把相关的初始化都写到该函数里面。

第一部分, 使能定时器和相关通道对应的 GPIO 时钟, 以及初始化相关 IO 引脚。

第二部分, 通过 `HAL_TIM_PWM_Init` 函数初始化定时器的 ARR 和 PSC 等参数。

第三部分, 通过 `HAL_TIM_PWM_ConfigChannel` 函数设置 PWM 模式 1、输出极性, 以及输出空闲状态等。

第四部分, 通过 `HAL_TIMEx_ConfigBreakDeadTime` 函数配置断路功能。

最后一定记得要调用 `HAL_TIM_PWM_Start` 函数和 `HAL_TIMEx_PWMN_Start` 函数启动通道输出和互补通道输出。

为了方便, 我们还定义了设置输出比较值和死区时间的函数, 其定义如下:

```

/**
 * @brief      定时器 TIMX 设置输出比较值 & 死区时间
 * @param      ccr: 输出比较值
 * @param      dtg: 死区时间
 * @arg        dtg[7:5]=0xx 时, 死区时间 = dtg[7:0] * tDTS
 * @arg        dtg[7:5]=10x 时, 死区时间 = (64 + dtg[6:0]) * 2 * tDTS
 * @arg        dtg[7:5]=110 时, 死区时间 = (32 + dtg[5:0]) * 8 * tDTS
 * @arg        dtg[7:5]=111 时, 死区时间 = (32 + dtg[5:0]) * 16 * tDTS
 * @note       tDTS = 1 / (Ft / CKD[1:0]) = 1 / 18M = 55.56ns
 * @retval     无
 */
void atim_timx_cplm_pwm_set(uint16_t ccr, uint8_t dtg)
{
    g_sbreak_dead_time_config.DeadTime = dtg; /* 死区时间设置 */
    HAL_TIMEx_ConfigBreakDeadTime(&g_timx_cplm_pwm_handle,
                                   &g_sbreak_dead_time_config); /* 重设死区时间 */
    __HAL_TIM_MOE_ENABLE(&g_timx_cplm_pwm_handle); /* MOE=1, 使能主输出 */
    ATIM_TIMX_CPLM_CHY_CCRy = ccr; /* 设置比较寄存器 */
}

```

通过重新调用 `HAL_TIMEx_ConfigBreakDeadTime` 函数设置死区时间, 注意这里的 `g_sbreak_dead_time_config` 是全局结构体变量, 在 `atim_timx_cplm_pwm_init` 函数已经初始化其他结构体成员了, 这里只是对 `DeadTime` 成员 (死区时间) 配置。死区时间的计算方法前面已经讲解过, 这里只要把要设置的 DTG[7:0] 值, 通过 `dtg` 形参赋值给 `DeadTime` 结构体成员就行。另外一个形参是 `ccr`, 用于设置捕获/比较寄存器的值, 即控制 PWM 的占空比。

在 `main.c` 里面编写如下代码:

```

int main(void)
{
    uint8_t t = 0;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
}

```

```

usart_init(115200);          /* 串口初始化为 115200 */
led_init();                  /* 初始化 LED */
atim_timx_cplm_pwm_init(1000 - 1, 72 - 1); /* 1Mhz 的计数频率 1Khz 的周期. */
atim_timx_cplm_pwm_set(300, 100);          /* 占空比:7:3,死区时间 100 * tDTS */

while (1)
{
    delay_ms(10);
    t++;
    if (t >= 20)
    {
        LED0_TOGGLE();          /* LED0 (RED) 闪烁 */
        t = 0;
    }
}

```

先看 `atim_timx_cplm_pwm_init(1000 - 1, 72 - 1)` 这个语句，这两个形参分别设置自动重载寄存器的值为 999，以及定时器预分频器寄存器的值为 71。先看预分频系数，我们设置为 72 分频，定时器 1 的时钟源频率等于 APB2 总线时钟频率，即 72MHz，可以得到计数器的计数频率是 1MHz，即每 1us 计数一次。再到自动重载寄存器的值为 999 决定的是 PWM 的频率（周期），可以得到 PWM 的周期为 $(999+1)*1us = 1000us = 1ms$ 。边沿对齐模式下，使用 PWM 模式 1 或者 PWM 模式 2，得到的 PWM 周期是定时器溢出时间。这里的 1ms，也可以直接通过定时器溢出时间计算公式 $Tout = ((arr+1)*(psc+1))/Tclk$ 得到。

调用 `atim_timx_cplm_pwm_set(300, 100)` 这个语句，相当于设置捕获/比较寄存器的值为 300，DTG[7:0]的值为 100。通过计算可以得到 PWM 的占空比为 70%，死区时间为 5.56us。根据 PWM 生成原理分析，再结合图 21.3.2 产生 PWM 示意图，以及我们在 `atim_timx_cplm_pwm_init` 函数配置 PWM 模式 1、OCy 输出极性为低，占空比的计算很简单，可以由 $(1000-300)/1000$ 得到。关于死区时间的计算方法，前面已经讲解过，这里以 DTG[7:0]的值为 100 为例，再来讲解一遍计算过程。由前面讲解的内容知道，我们例程配置 CKD[1:0]位的值为 2，可以得到 $tDTS = 55.56ns$ 。基于这个前提，通过改变 DTG[7:0]的值，可以得到不同的死区时间。这里我们配置 DTG[7:0]的值为 100，即二进制数 0110 0100，符合第一种情况 $dtg[7:5]=0xx$ 时，死区时间 $DT = DTG[7:0] * tDTS$ 。可以得到死区时间 $DT = 100 * 55.56 ns = 5.56us$ 。

下面我们下载到开发板子验证一下。

22.4.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了。我们需要借助示波器观察 PE9 正常输出和 PE8 互补输出 PWM 的情况，示波器显示截图如图 22.4.4.1 所示：

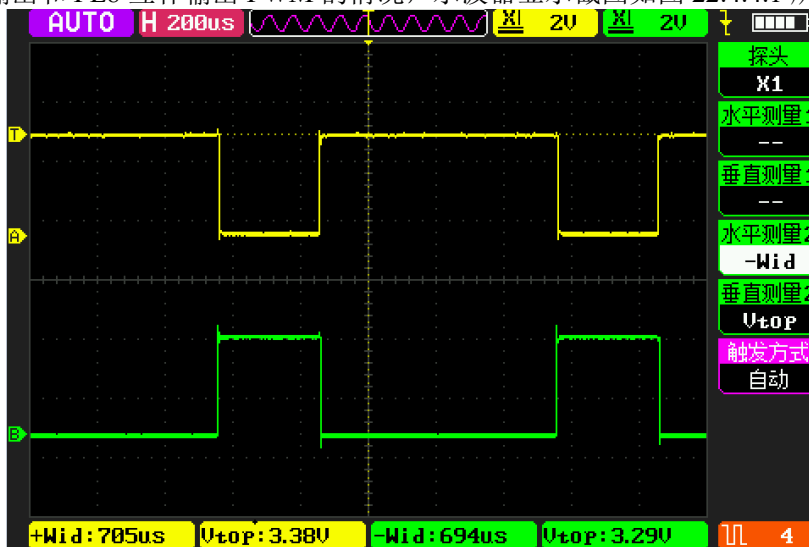


图 22.4.4.1 PE8 正常输出和 PE9 互补输出 PWM 的情况

图 22.4.4.1 中的由上到下分别是 PE9 输出 70% 占空比的 PWM 波和 PE8 互补输出 30% 占空比的 PWM 波。互补输出的 PWM 波的正脉宽减去正常的 PWM 的负脉宽的值除以 2 就是死区时间，也可以是正常的 PWM 的正脉宽减去互补输出的 PWM 波的负脉宽的值除以 2。我们使用第一种方法得到：死区时间 = (705 - 694) / 2 us = 5.5us。与我们理论到的的值 5.56us 基本一样，这样的误差是正常的。

要是不相信，我们再举个例子，我们把调用的函数改为 `atim_timx_cplm_pwm_set(300, 250)`，即配置 DTG[7:0] 的值为 250，这个例子的计算过程在本实验前面死区时间计算的内容讲过，这里就不再赘述。经过计算得到死区时间 DT = 51.56us。修改好后，示波器显示截图如下图所示：

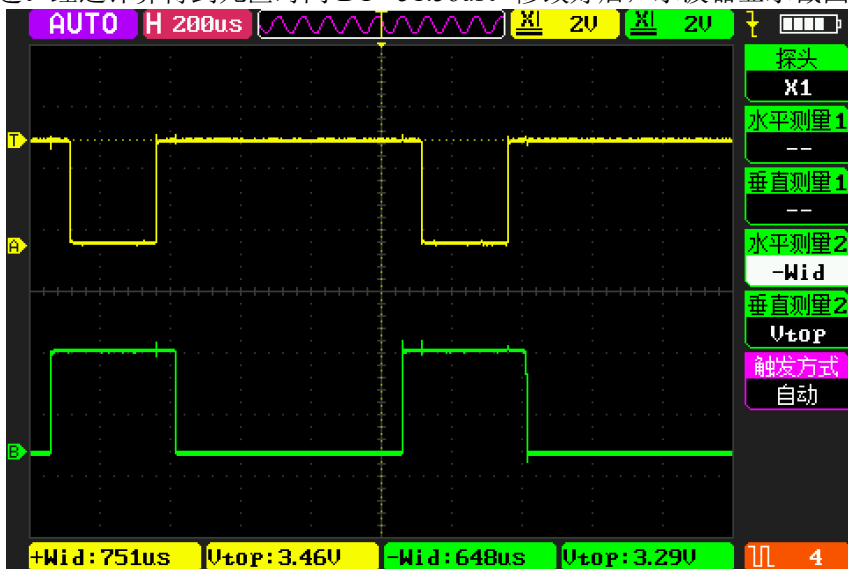


图 22.4.4.2 修改程序后 PE9 正常输出和 PE8 互补输出 PWM 的情况

由图 22.4.4.2 可得到，死区时间 = (751 - 648) / 2 us = 51.5us。与我们理论到的的值 51.56us 也是差不多的，误差在正常范围。由此证明我们的死区时间设置是没有问题。

刹车功能验证：当给刹车输入引脚（PE15）接入高电平（这里直接用杜邦线连接 PE15 到 3.3V）时，就会进行刹车，即 PE9 和 PE8 停止输出 PWM 波，如图 22.4.4.3 所示：

刹车功能验证：当给刹车输入引脚（PE15）接入高电平（这里直接用杜邦线把 PE15 连接到 3.3V）时，就会进行刹车，MOE 位被硬件清零。由《STM32F10xxx 参考手册_V10（中文版）.pdf》第 245 页 表 75 可以知道刹车信号输入后，如果存在时钟，经过死区后 OCx=OISx 且 OCxN=OISxN。在 `atim_timx_cplm_pwm_init` 函数中，我们设置当 MOE=0 时，OCx=1、OCxN=1，即 PE9 和 PE8 都是输出高电平。下面通过示波器来验证一下，如图 22.4.4.3 所示：

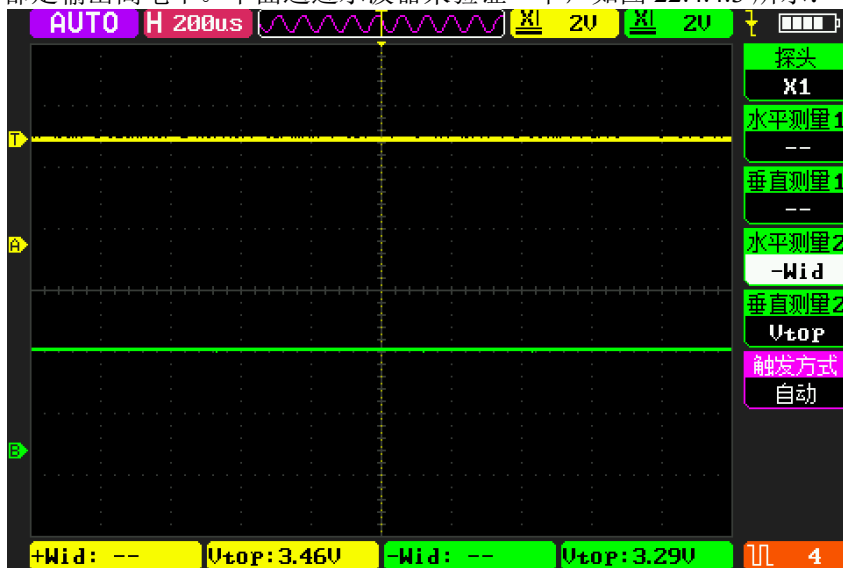


图 22.4.4.3 刹车后的输出情况

从上图可以看到 PE9 和 PE8 输出的都是高电平，符合我们预期的设置。

另外因为我们使能了 AOE 位（即把该位置 1），如果刹车输入为无效极性时，MOE 位在发生下一个更新事件时自动置 1，恢复运行模式（即继续输出 PWM）。因此当停止给 PE15 接入低电平（拔掉之前连接的杜邦线），PWM 会自动恢复输出。

22.5 高级定时器 PWM 输入模式实验

本小节我们来学习使用高级定时器 PWM 输入模式，此模式是输入捕获模式的一个特例。PWM 输入模式经常被应用于测量 PWM 脉宽和频率。PWM 输入模式在《STM32F10xxx 参考手册_V10（中文版）.pdf》手册 216 页有详细的文字描述。下面我们结合这些文字，配合高级定时器框图给大家介绍 PWM 输入的工作原理。

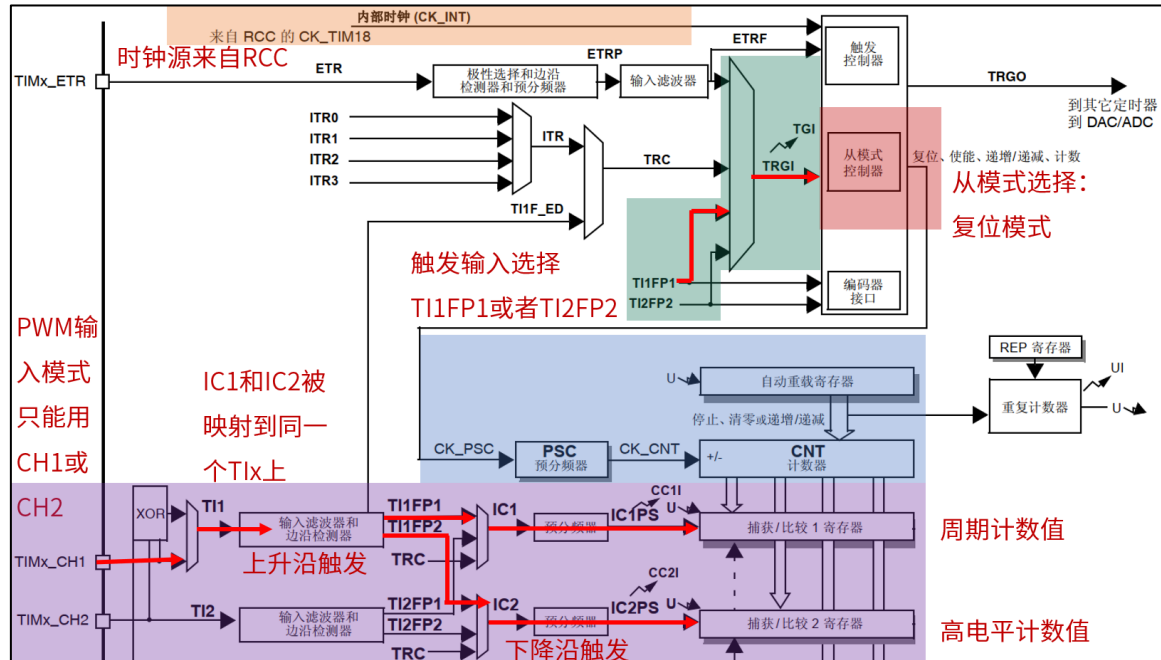


图 22.5.1 PWM 输入模式工作原理示意图

第一，确定定时器时钟源。本实验中我们使用内部时钟（CK_INT），F1 系列高级定时器挂载在 APB2 总线上，按照 sys_stm32_clock_init 函数的配置，定时器时钟频率等于 APB2 总线时钟频率，即 72MHz。计数器的计数频率确定了测量的精度。

第二，确定 PWM 输入的通道。PWM 输入模式下测量 PWM，PWM 信号输入只能从通道 1（CH1）或者通道 2（CH2）输入。

第三，确定 IC1 和 IC2 的捕获边沿。这里以通道 1（CH1）输入 PWM 为例，一般我们习惯设置 IC1 捕获边沿为上升沿捕获，IC2 捕获边沿为下降沿捕获。

第四，选择触发输入信号（TRGI）。这里也是以通道 1（CH1）输入 PWM 为例，那么我们就应该选择 TI1FP1 为触发输入信号。如果是通道 2（CH2）输入 PWM，那就选择 TI2FP2 为触发输入信号。可以看到这里并没有对应通道 3（CH3）或者通道 4（CH4）的触发输入信号，所以我们只选择通道 1 或者通道 2 作为 PWM 输入的通道。

第五，从模式选择：复位模式。复位模式的作用是：在出现所选触发输入（TRGI）上升沿时，重新初始化计数器并生成一个寄存器更新事件。

第六，读取一个 PWM 周期内计数器的计数个数，以及高电平期间的计数个数，再结合计数器的计数周期（即计一个数的时间），最终通过计算得到输入的 PWM 周期和占空比等参数。以通道 1（CH1）输入 PWM，设置 IC1 捕获边沿为上升沿捕获，IC2 捕获边沿为下降沿捕获为例，那么 CCR1 寄存器的值+1 就是 PWM 周期内计数器的计数个数，CCR2 寄存器的值+1 就是 PWM 高电平期间计数器的计数个数。通过这两个值就可以计算出 PWM 的周期或者占空比等参数。

再举个例子，以通道 1（CH1）输入 PWM，设置 IC1 捕获边沿为下降沿捕获，IC2 捕获边沿为上升沿捕获为例，那么 CCR1 寄存器的值+1 依然是 PWM 周期内计数器的计数个数，但是 CCR2 寄存器的值+1 就是 PWM 低电平期间计数器的计数个数。通过这两个得到的参数依然可以计算出 PWM 的其它参数。这个大家了解一下就可以了，一般我们使用第六介绍例子。

通过上面的描述,如果大家还不理解,下面我们结合 PWM 输入模式时序来分析一下。PWM 输入模式时序图如图 22.5.2 所示:

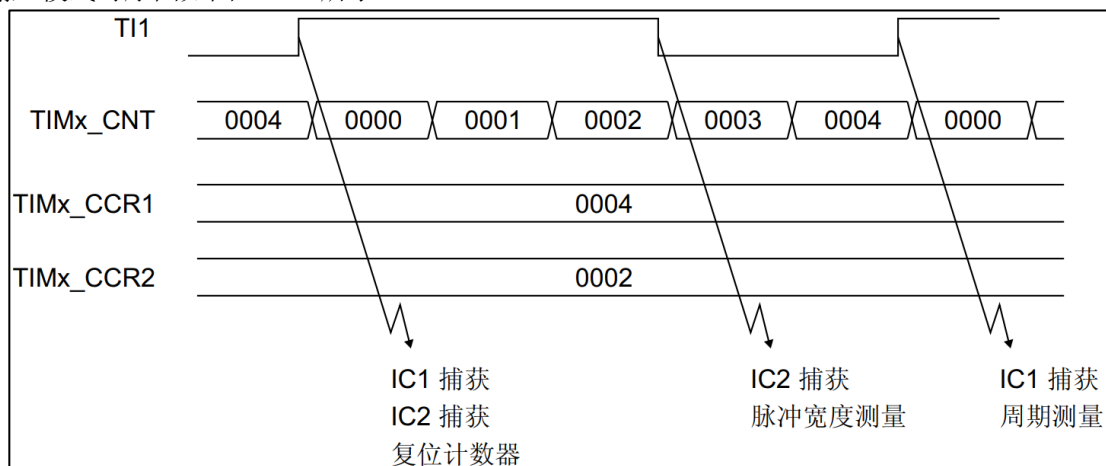


图 22.5.2 PWM 输入模式时序图

图 22.5.2 是以通道 1 (CH1) 输入 PWM, 设置 IC1 捕获边沿为上升沿捕获, IC2 捕获边沿为下降沿捕获为例的 PWM 输入模式时序图。

从时序图可以看出,计数器的计数模式是递增计数模式。从左边开始看,当 TI1 来了上升沿时,计数器的值被复位为 0 (原因是从模式选择为复位模式),IC1 和 IC2 都发生捕获事件。然后计数器的值计数到 2 的时候,IC2 发生了下降沿捕获,捕获事件会导致这时候的计数器的值被锁存到 CCR2 寄存器中,该值+1 就是高电平期间计数器的计数个数。最后计数器的值计数到 4 的时候,IC1 发生了上升沿捕获,捕获事件会导致这时候的计数器的值被锁存到 CCR1 寄存器中,该值+1 就是 PWM 周期内计数器的计数个数。

假设计数器的计数频率是 72MHz,那我们就可以计算出这个 PWM 的周期、频率和占空比等参数了。下面就以这个为例给大家计算一下。由计数器的计数频率为 72MHz,可以得到计数器计一个数的时间是 13.8ns (即测量的精度是 13.8ns)。知道了测量精度,再来计算 PWM 的周期, PWM 周期 $= (4+1) * (1/72000000) = 69.4ns$,那么 PWM 的频率就是 14.4MHz。占空比 $= (2+1)/(4+1) = 3/5$ (即占空比为 60%)。

22.5.1 TIM1/TIM8 寄存器

高级定时器 PWM 输入模式实验除了用到定时器的时基单元: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) 之外。主要还用到以下这些寄存器:

● 从模式控制寄存器 (TIMx_SMCR)

TIM1/TIM8 的从模式控制寄存器描述如图 22.5.1.1 所示:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|-----|--|----|----|----------|----|----|-----|----|---------|----|----|----|----------|----|
| ETP | ECE | ETPS[1:0] | | | ETF[3:0] | | | MSM | | TS[2:0] | | 保留 | | SMS[2:0] | |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | | RW | RW | RW |
| 位6:4 | | TS[2:0]: 触发选择 (Trigger selection) 这3位选择用于同步计数器的触发输入。 001: 内部触发1(ITR1) 101: 滤波后的定时器输入1(TI1FP1) 010: 内部触发2(ITR2) 110: 滤波后的定时器输入2(TI2FP2) | | | | | | | | | | | | | |
| 位2:0 | | SMS[2:0]: 从模式选择 (Slave mode selection) 当选择了外部信号,触发信号(TRGI)的有效边沿与选中的外部输入极性相关(见输入控制寄存器和控制寄存器的说明) 100: 复位模式 - 选中的触发输入(TRGI)的上升沿重新初始化计数器,并且产生一个更新寄存器的信号。 | | | | | | | | | | | | | |

图 22.5.1.1 TIMx_SMCR 寄存器

该寄存器的 SMS[2:0]位,用于从模式选择。比如在本实验中我们需要用到复位模式,所以设置 SMS[2:0]=100。TS[2:0]位是触发选择,我们设置为滤波后的定时器输入 1 (TI1FP1),即

TS[2:0]为 101。

● 捕获/比较模式寄存器 1/2 (TIMx_CCMR1/2)

TIM1/TIM8 的捕获/比较模式寄存器 (TIMx_CCMR1/2)，该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 22.5.1.2 所示：

| 15 | | 14 | | 13 | | 12 | | 11 | | 10 | | 9 | | 8 | | 7 | | 6 | | 5 | | 4 | | 3 | | 2 | | 1 | | 0 | |
|-----------|--|-----------|--|----|--|-------------|--|-------|--|-----------|--|----|--|-----------|--|-----------|--|----|--|-------------|--|-------|--|-----------|--|----|--|----|--|----|--|
| OC2CE | | OC2M[2:0] | | | | OC2PE | | OC2FE | | CC2S[1:0] | | | | OC1CE | | OC1M[2:0] | | | | OC1PE | | OC1FE | | CC1S[1:0] | | | | | | | |
| IC2F[3:0] | | | | | | IC2PSC[1:0] | | | | | | | | IC1F[3:0] | | | | | | IC1PSC[1:0] | | | | | | | | | | | |
| rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | | rw | |

图 22.5.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在用到输入捕获模式。关于该寄存器的详细说明，请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》第 240 页，13.4.7 节。

本实验我们通过定时器 1 通道 1 输入 PWM 信号，所以 IC1 和 IC2 都映射到 TI1 上。配置 CC1S[1:0]=01、CC2S[1:0]=10，其他位不用设置，默认为 0 即可。

● 捕获/比较使能寄存器 (TIMx_CCER)

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 22.5.1.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|------|------|-------|-------|------|------|-------|-------|------|------|-------|-------|------|------|----|
| 保留 | CC4P | CC4E | CC3NP | CC3NE | CC3P | CC3E | CC2NP | CC2NE | CC2P | CC2E | CC1NP | CC1NE | CC1P | CC1E | |
| | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

图 22.5.1.3 TIMx_CCER 寄存器

IC1 捕获上升沿，所以 CC1P 位置 0，即捕获发生在 IC1 的上升沿。IC2 捕获下降沿，所以 CC2P 位置 1，即捕获发生在 IC1 的下降沿。设置好捕获边沿后，还需要使能这两个通道捕获，即 CC1E 和 CC2E 位置 1。

● 捕获/比较寄存器 1/2/3/4 (TIMx_CCR1/2/3/4)

捕获/比较寄存器 (TIMx_CCR1/2/3/4)，该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器描述如图 22.5.1.4 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CCR1[15:0] | | | | | | | | | | | | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

图 22.5.1.4 TIMx_CCR1 寄存器

本实验中，CCR1 寄存器用于获取 PWM 周期内计数器的计数个数。CCR2 寄存器用于获取 PWM 高电平期间计数器的计数个数。

● DMA/中断使能寄存器 (TIMx_DIER)

DMA/中断使能寄存器描述如图 22.5.1.5 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|-----|--|-------|-------|-------|-------|-----|-----|-----|-------|-------|-------|-------|-------|-----|
| 保留 | TDE | COMDE | CC4DE | CC3DE | CC2DE | CC1DE | UDE | BIE | TIE | COMIE | CC4IE | CC3IE | CC2IE | CC1IE | UIE |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位1 | | CC1IE: 允许捕获/比较1中断 (Capture/Compare 1 interrupt enable) 0: 禁止捕获/比较1中断; 1: 允许捕获/比较1中断。 | | | | | | | | | | | | | |
| 位0 | | UIE: 允许更新中断 (Update interrupt enable) 0: 禁止更新中断; 1: 允许更新中断。 | | | | | | | | | | | | | |

图 22.5.1.5 TIMx_DIER 寄存器

该寄存器位 0 (UIE) 用于使能或者禁止更新中断，因为本实验我们用到更新中断，所以该位需要置 1。位 1 (CC1IE) 用于使能或者禁止捕获/比较 1 中断，我们用到捕获中断，所以该位需要置 1。

22.5.2 硬件设计

1. 例程功能

首先通过 TIM3_CH2(PB5)输出 PWM 波。然后把 PB5 输出的 PWM 波用杜邦线接入 PC6 (定时器 8 通道 1)，最后通过串口打印 PWM 波的脉宽和频率等信息。通过 LED1 闪烁来提示程序正在运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) 定时器 3 通道 2 (PB5) 输出 PWM 波

定时器 8 通道 1 (PC6) 输入 PWM 波

3. 原理图

定时器属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们把 PB5 引脚输出的 PWM 波用杜邦线接入 PC6 引脚，然后通过电脑串口上位机软件观察打印出来的信息。

22.5.3 程序设计

定时器 PWM 输入模式实验用到的 HAL 库中的驱动代码在前面实验都有介绍过了。我们在程序解析再详细讲解应用到的函数，下面介绍一下高级定时器 PWM 输入模式的配置步骤。

高级定时器 PWM 输入模式配置步骤

1) 开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1，对应 IO 是 PC6，它们的时钟开启方法如下：

```
HAL_RCC_TIM8_CLK_ENABLE(); /* 使能定时器 8 */
HAL_RCC_GPIOC_CLK_ENABLE(); /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的输入捕获功能时，我们调用的是 HAL_TIM_IC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数，但是为不跟前面的实验共用该回调函数，提高独立性，我们就直接在 atim_timx_pwm_in_chy_init 函数中，使能定时器时钟和 GPIO 时钟，初始化通道对应 IO 引脚等。

3) 从模式配置，IT1 触发更新

通过 HAL_TIM_SlaveConfigSynchronization 函数，配置从模式：复位模式、定时器输入触发源、边缘检测、是否滤波等。

4) 设置 IC1 捕获相关参数

通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器捕获通道 1 的工作方式，包括边缘检测极性、映射关系，输入滤波和输入分频等。

5) 设置 IC2 捕获相关参数

通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器捕获通道 2 的工作方式，包括边缘检测极性、映射关系，输入滤波和输入分频等。

6) 使能定时器更新中断，开启捕获功能，配置定时器中断优先级

通过 __HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start_IT 函数使能定时器并开启通道 1 或者通道 2 的捕获功能，使能捕获中断。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

7) 编写中断服务函数

TIM1 和 TIM8 有独立的输入捕获中断服务函数，分别是 TIM1_CC_IRQHandler 和 TIM8_CC_IRQHandler，其他定时器则没有，所以如果是 TIM1 和 TIM8 可以直接使用输入捕获中断服务函数来处理输入捕获中断。在使用 TIM1 的时候，如果要考虑定时器 1 溢出，可以重定义更新中断服务函数 TIM1_UP_IRQHandler。如果使用 HAL 库的中断回调机制，可以在相关中断服务函数中直接调用定时器中断公共处理函数 HAL_TIM_IRQHandler，然后我们直接重定义相关的中断回调函数来编写中断程序即可。本实验为了兼容性，我们自定义一个中断处理函数 atim_timx_pwm_in_chy_process，里面包含了捕获中断和更新中断的处理，具体看源码。

22.5.3.1 程序流程图

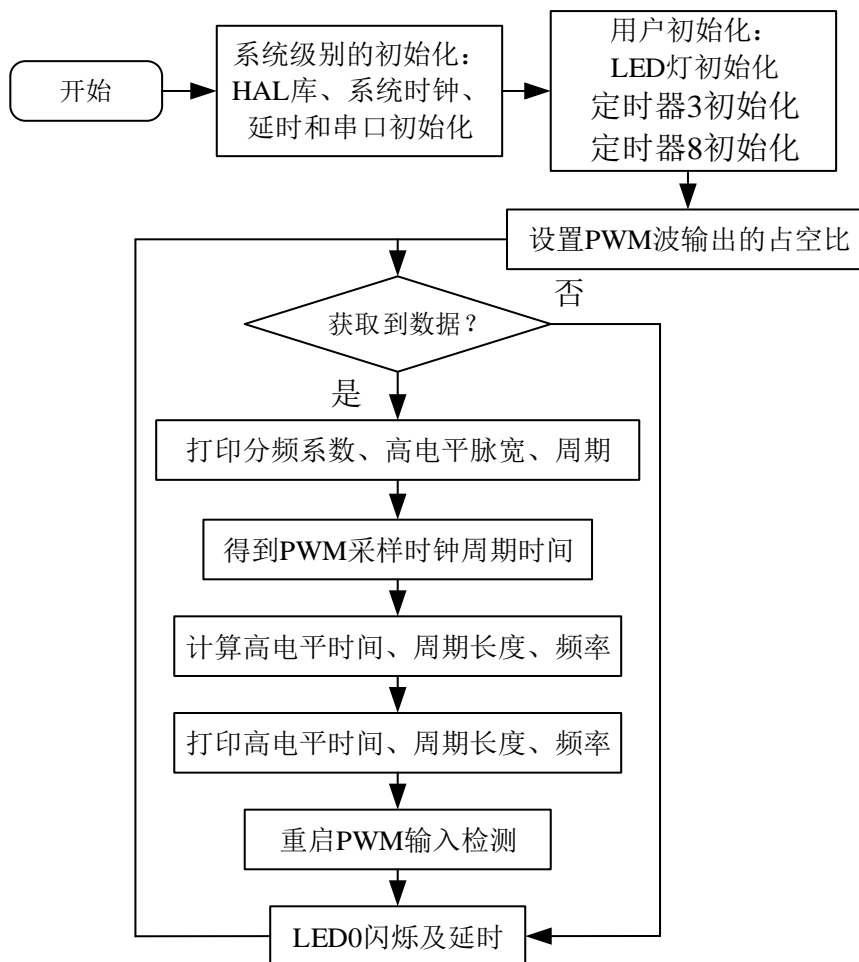


图 22.5.3.1.1 高级定时器 PWM 输入模式实验程序流程图

22.5.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```

/* TIMX PWM输入模式 定义
 * 这里的输入捕获使用定时器 TIM8_CH1
 * 默认是针对 TIM1/TIM8 等高级定时器
 * 注意：通过修改这几个宏定义，可以支持 TIM1~TIM8 任意一个定时器的通道 1/通道 2
 */
#define ATIM_TIMX_PWMIN_CHY_GPIO_PORT      GPIOC
#define ATIM_TIMX_PWMIN_CHY_GPIO_PIN      GPIO_PIN_6
  
```

```
#define ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE() \
do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0) /* PC 口时钟使能 */

#define ATIM_TIMX_PWMIN TIM8
#define ATIM_TIMX_PWMIN_IRQn TIM8_UP_IRQn
#define ATIM_TIMX_PWMIN_IRQHandler TIM8_UP_IRQHandler
#define ATIM_TIMX_PWMIN_CHY TIM_CHANNEL_1 /* 通道 Y, 1<= Y <=2 */
#define ATIM_TIMX_PWMIN_CHY_CLK_ENABLE()
do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0) /* TIM8 时钟使能 */

/* TIM1 / TIM8 有独立的捕获中断服务函数,需要单独定义,对于 TIM2~5 等,则不需要以下定义 */
#define ATIM_TIMX_PWMIN_CC_IRQn TIM8_CC_IRQn
#define ATIM_TIMX_PWMIN_CC_IRQHandler TIM8_CC_IRQHandler
```

可以把上面的宏定义分成三部分,第一部分包括是定时器 8 通道 1 对应的 IO 口的宏定义,第二部分则是定时器 8 的相应宏定义,另外针对 TIM1/ TIM8 有独立的捕获中断服务函数,需要单独定义。

下面看 `atim.c` 的程序,首先是高级定时器 PWM 输入模式初始化函数,其定义如下:

```
/**
 * @brief      定时器 TIMX 通道 Y PWM 输入模式 初始化函数
 * @note
 *
 *      高级定时器的时钟来自 APB2, 而 PCLK2 = 72Mhz, 我们设置 PPRE2 不分频, 因此
 *      高级定时器时钟 = 72Mhz
 *      定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft=定时器工作频率,单位:Mhz
 *
 *      本函数初始化的时候: 使用 psc=0, arr 固定为 65535. 得到采样时钟频率为 72Mhz,
 *      精度约 13.8ns
 *
 * @param      无
 * @retval     无
 */
void atim_timx_pwmin_chy_init(void)
{
    GPIO_InitTypeDef gpio_init_struct = {0};
    TIM_SlaveConfigTypeDef slave_config = {0};
    TIM_IC_InitTypeDef tim_ic_pwmin_chy = {0};

    ATIM_TIMX_PWMIN_CHY_CLK_ENABLE();
    ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE();
    __HAL_RCC_AFIO_CLK_ENABLE();
    AFIO_REMAP_PARTIAL(AFIO_EVCR_PORT_PC, AFIO_EVCR_PIN_PX6); /* 复用 TIM8_CH1/PC6 */

    gpio_init_struct.Pin = ATIM_TIMX_PWMIN_CHY_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(ATIM_TIMX_PWMIN_CHY_GPIO_PORT, &gpio_init_struct);

    g_timx_pwmin_chy_handle.Instance = ATIM_TIMX_PWMIN; /* 定时器 8 */
    g_timx_pwmin_chy_handle.Init.Prescaler = 0; /* 定时器预分频系数 */
    g_timx_pwmin_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_pwmin_chy_handle.Init.Period = 65535; /* 自动重载值 */
    HAL_TIM_IC_Init(&g_timx_pwmin_chy_handle);

    /* 从模式配置, IT1 触发更新 */
    slave_config.SlaveMode = TIM_SLAVEMODE_RESET; /* 从模式: 复位模式 */
    slave_config.InputTrigger = TIM_TS_TI1FP1; /* 定时器输入触发源: TI1FP1 */
    slave_config.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING; /* 上升沿检测 */
    slave_config.TriggerFilter = 0; /* 不滤波 */
    HAL_TIM_SlaveConfigSynchro(&g_timx_pwmin_chy_handle, &slave_config);
```

```

/* IC1 捕获: 上升沿触发 TI1FP1 */
tim_ic_pwm_min_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING; /* 上升沿检测 */
tim_ic_pwm_min_chy.ICSelection = TIM_ICSELECTION_DIRECTTI; /* IC1 映射到 TI1 上 */
tim_ic_pwm_min_chy.ICPrescaler = TIM_ICPSC_DIV1; /* 不分频 */
tim_ic_pwm_min_chy.ICFilter = 0; /* 不滤波 */
HAL_TIM_IC_ConfigChannel(&g_timx_pwm_min_chy_handle, &tim_ic_pwm_min_chy,
                        TIM_CHANNEL_1);

/* IC2 捕获: 上升沿触发 TI1FP2 */
tim_ic_pwm_min_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING; /* 下降沿检测 */
tim_ic_pwm_min_chy.ICSelection = TIM_ICSELECTION_INDIRECTTI; /* IC2 映射到 TI1 上 */
HAL_TIM_IC_ConfigChannel(&g_timx_pwm_min_chy_handle, &tim_ic_pwm_min_chy,
                        TIM_CHANNEL_2);

/* 设置中断优先级, 抢占优先级 1, 子优先级 3 */
HAL_NVIC_SetPriority(ATIM_TIMX_PWM_MIN_IRQn, 1, 3);
HAL_NVIC_EnableIRQ(ATIM_TIMX_PWM_MIN_IRQn); /* 开启 TIMx 中断 */

/* TIM1/TIM8 有独立的输入捕获中断服务函数 */
if (ATIM_TIMX_PWM_MIN == TIM1 || ATIM_TIMX_PWM_MIN == TIM8)
{
    /* 设置中断优先级, 抢占优先级 1, 子优先级 3 */
    HAL_NVIC_SetPriority(ATIM_TIMX_PWM_MIN_CC_IRQn, 1, 3);
    HAL_NVIC_EnableIRQ(ATIM_TIMX_PWM_MIN_CC_IRQn); /* 开启 TIMx 中断 */
}

__HAL_TIM_ENABLE_IT(&g_timx_pwm_min_chy_handle, TIM_IT_UPDATE);
HAL_TIM_IC_Start_IT(&g_timx_pwm_min_chy_handle, TIM_CHANNEL_1);
HAL_TIM_IC_Start_IT(&g_timx_pwm_min_chy_handle, TIM_CHANNEL_2);
}

```

在 `atim_timx_pwm_min_chy_init` 函数中, 没有使用 HAL 库的 MSP 回调, 而是把相关的初始化都写到该函数里面。

第一部分, 使能定时器和相关通道对应的 GPIO 时钟, 以及初始化相关 IO 引脚。

第二部分, 通过 `HAL_TIM_IC_Init` 函数初始化定时器的 ARR 和 PSC 等参数。

第三部分, 通过 `HAL_TIM_SlaveConfigSynchronization` 函数配置从模式, 复位模式等。

第四部分, 通过 `HAL_TIM_IC_ConfigChannel` 函数分别配置 IC1 和 IC2。

第五部分, 配置 NVIC, 如使能定时器中断, 配置抢占优先级和响应优先级。

最后, 通过调用 `HAL_TIM_IC_Start_IT` 函数和 `__HAL_TIM_ENABLE_IT` 函数宏使能捕获中断和更新中断, 并且使能定时器。

为了方便, 我们定义了重新启动捕获函数, 其定义如下:

```

/**
 * @brief      定时器 TIMX PWM 输入模式 重新启动捕获
 * @param      无
 * @retval     无
 */
void atim_timx_pwm_min_chy_restart(void)
{
    sys_intx_disable(); /* 关闭中断 */

    g_timxchy_pwm_min_sta = 0; /* 清零状态, 重新开始检测 */
    g_timxchy_pwm_min_psc = 0; /* 分频系数清零 */

    /* 以最大的计数频率采集, 以得到最好的精度 */
    __HAL_TIM_SET_PRESCALER(&g_timx_pwm_min_chy_handle, 0);
    __HAL_TIM_SET_COUNTER(&g_timx_pwm_min_chy_handle, 0); /* 计数器清零 */

    __HAL_TIM_ENABLE_IT(&g_timx_pwm_min_chy_handle, TIM_IT_CC1); /* 使能捕获中断 */
    __HAL_TIM_ENABLE_IT(&g_timx_pwm_min_chy_handle, TIM_IT_UPDATE); /* 使能更新中断 */
    __HAL_TIM_ENABLE(&g_timx_pwm_min_chy_handle); /* 使能定时器 TIMX */
}

```

```

    ATIM_TIMX_PWMIN->SR = 0;          /* 清除所有中断标志位 */

    sys_intx_enable();                /* 打开中断 */
}

```

该函数首先关闭所有中断，然后把一些状态标志位清零、设置定时器预分频系数、计数器值、使能相关中断、以及清除相关中断标志位，最后才允许被中断。

最后要介绍的是中断服务函数，在定时器1的输入捕获中断服务函数TIM1_CC_IRQHandler和更新中断服务函数TIM1_UP_IRQHandler里面都是直接调用atim_timx_pwmmin_chy_process函数。输入捕获中断服务函数和更新中断服务函数都是用到宏定义的，这三个函数定义如下：

```

/**
 * @brief      定时器 TIMX 更新/溢出 中断服务函数
 * @note      TIM1/TIM8 的这个函数仅用于更新/溢出中断服务, 捕获在另外一个函数!
 *            其他普通定时器则更新/溢出/捕获, 都在这个函数里面处理!
 * @param      无
 * @retval     无
 */
void ATIM_TIMX_PWMIN_IRQHandler(void)
{
    atim_timx_pwmmin_chy_process();
}

/**
 * @brief      定时器 TIMX 输入捕获 中断服务函数
 * @note      仅 TIM1/TIM8 有这个函数, 其他普通定时器没有这个中断服务函数!
 * @param      无
 * @retval     无
 */
void ATIM_TIMX_PWMIN_CC_IRQHandler(void)
{
    atim_timx_pwmmin_chy_process();
}

/**
 * @brief      定时器 TIMX 通道 Y PWM 输入模式 中断处理函数
 * @note      因为 TIM1/TIM8 等有多个中断服务函数, 而 TIM2~5/TIM12/TIM15 等普通定时器只有 1 个中断服务
 *            函数, 为了更好的兼容, 我们对中断处理统一放到 atim_timx_pwmmin_chy_process 函数里面进行处理
 * @param      无
 * @retval     无
 */
static void atim_timx_pwmmin_chy_process(void)
{
    static uint8_t sflag = 0;          /* 启动 PWMIN 输入检测标志 */

    if (g_timxchy_pwmmin_sta)
    {
        g_timxchy_pwmmin_psc = 0;
        ATIM_TIMX_PWMIN->SR = 0;        /* 清除所有中断标志位 */
        __HAL_TIM_SET_COUNTER(&g_timx_pwmmin_chy_handle, 0);    /* 计数器清零 */
        return ;
    }

    /* 如果发生了更新中断 */
    if (__HAL_TIM_GET_FLAG(&g_timx_pwmmin_chy_handle, TIM_FLAG_UPDATE))
    {
        /* 清除更新中断标记 */
        __HAL_TIM_CLEAR_FLAG(&g_timx_pwmmin_chy_handle, TIM_FLAG_UPDATE);
        /* 没有发生周期捕获中断, 且捕获未完成 */
        if (__HAL_TIM_GET_FLAG(&g_timx_pwmmin_chy_handle, TIM_FLAG_CC1) == 0)
        {

```

```

    sflag = 0;
    if (g_timxchy_pwmmin_psc == 0) /* 从 0 到 1 */
    {
        g_timxchy_pwmmin_psc++;
    }
    else
    {
        if (g_timxchy_pwmmin_psc == 65535) /* 已经最大了,可能是无输入状态 */
        {
            g_timxchy_pwmmin_psc = 0; /* 重新恢复不分频 */
        }
        else if (g_timxchy_pwmmin_psc > 32767) /* 不能倍增了 */
        {
            g_timxchy_pwmmin_psc = 65535; /* 直接等于最大分频系数 */
        }
        else
        {
            g_timxchy_pwmmin_psc += g_timxchy_pwmmin_psc; /* 倍增 */
        }
    }

    __HAL_TIM_SET_PRESCALER(&g_timx_pwmmin_chy_handle,
                            g_timxchy_pwmmin_psc); /* 设置定时器预分频系数 */
    __HAL_TIM_SET_COUNTER(&g_timx_pwmmin_chy_handle, 0); /* 计数器清零 */
    ATIM_TIMX_PWMIN->SR = 0; /* 清除所有中断标志位 */
    return ;
}

if (sflag == 0) /* 第一次采集到捕获中断 */
{
    /* 检测到了第一次周期捕获中断 */
    if (__HAL_TIM_GET_FLAG(&g_timx_pwmmin_chy_handle, TIM_FLAG_CC1))
    {
        sflag = 1; /* 标记第一次周期已经捕获,第二次周期捕获可以开始了 */
    }
    ATIM_TIMX_PWMIN->SR = 0; /* 清除所有中断标志位 */
    return ; /* 完成此次操作 */
}

if (g_timxchy_pwmmin_sta == 0) /* 还没有成功捕获 */
{
    /* 检测到了周期捕获中断 */
    if (__HAL_TIM_GET_FLAG(&g_timx_pwmmin_chy_handle, TIM_FLAG_CC1))
    {
        g_timxchy_pwmmin_hval = HAL_TIM_ReadCapturedValue(
            &g_timx_pwmmin_chy_handle, TIM_CHANNEL_2) + 1; /* 高电平脉宽捕获值 */
        g_timxchy_pwmmin_cval = HAL_TIM_ReadCapturedValue(
            &g_timx_pwmmin_chy_handle, TIM_CHANNEL_1) + 1; /* 周期捕获值 */

        /* 高电平脉宽必定小于周期长度 */
        if (g_timxchy_pwmmin_hval < g_timxchy_pwmmin_cval)
        {
            g_timxchy_pwmmin_sta = 1; /* 标记捕获成功 */

            g_timxchy_pwmmin_psc = ATIM_TIMX_PWMIN->PSC; /* 获取 PWM 输入分频系数 */

            if (g_timxchy_pwmmin_psc == 0) /* 分频系数为 0 的时候,修正读取数据 */
            {
                g_timxchy_pwmmin_hval++; /* 修正系数为 1, 加 1 */
                g_timxchy_pwmmin_cval++; /* 修正系数为 1, 加 1 */
            }
        }
    }
}

```



```

sflag = 0;
/* 每次捕获 PWM 输入成功后, 停止捕获, 避免频繁中断影响系统正常代码运行 */
ATIM_TIMX_PWMIN->CR1 &= ~(1 << 0); /* 关闭定时器 TIMX */
/* 关闭通道 1 捕获中断 */
__HAL_TIM_DISABLE_IT(&g_timx_pwmmin_chy_handle, TIM_IT_CC1);
/* 关闭通道 2 捕获中断 */
__HAL_TIM_DISABLE_IT(&g_timx_pwmmin_chy_handle, TIM_IT_CC2);
/* 关闭更新中断 */
__HAL_TIM_DISABLE_IT(&g_timx_pwmmin_chy_handle, TIM_IT_UPDATE);
ATIM_TIMX_PWMIN->SR = 0; /* 清除所有中断标志位 */
}
else
{
    atim_timx_pwmmin_chy_restart();
}
}
}
ATIM_TIMX_PWMIN->SR = 0; /* 清除所有中断标志位 */
}

```

atim_timx_pwmmin_chy_process 函数包含了捕获中断程序和更新中断程序的处理。如果发生了更新中断（即定时器溢出），证明超出定时器量程，这里会加大预分频系数，以得到更大的量程。量程变大了，那么测量的精度就会降低，所谓鱼和熊掌不可兼得。代码中的“if(sflag == 0) /* 第一次采集到捕获中断 */”这个程序段，表示第一次采集到捕获中断。这时候相当于第一次捕获到上升沿，我们只是把 sflag 标志位置 1，然后清除所有中断标志位，等待下次的捕获中断发生。如果再次发生捕获中断，就会来到“if(g_timxchy_pwmmin_sta == 0) /* 还没有成功捕获 */”程序段。通过 HAL_TIM_ReadCapturedValue 函数获取 CCR1 和 CCR2 寄存器的值，把这个获取到的寄存器值+1 才是对应的计数器计数个数。如果预分频系数为 0 的时候，还要把这两个寄存器的值再+1，这样计算的结果更准确。其它的代码细节请大家自行查看源码，有详细的注释。

注释代码：实验 10-1 高级定时器输出指定个数 PWM 实验使用到 TIM8_UP_IRQHandler 中断服务函数，本实验同样使用到该函数，编译会报错，这里的做法是屏蔽实验 10-1 的相关代码。具体请看 atim.c 文件源码。atim.c 文件的程序就介绍到这。

下面介绍一下待测试的 PWM 怎么得到。因为在实验 9-2 通用定时器 PWM 输出实验我们已经编写了 PWM 波输出的程序，所以这里直接使用通用定时器的 PWM 输出实验的代码进行初始化，从而让 TIM3_CH2(PB5)输出 PWM 波。然后我们用杜邦线把 PB5 和 PC6 连接起来。这样 PB5 输出的 PWM 就可以输入到 PC6（定时器 8 通道 1）进行测量。

在 main.c 里面编写如下代码：

```

int main(void)
{
    uint8_t t = 0;
    double ht, ct, f, tpsec;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    gtim_timx_pwm_chy_init(10 - 1, 72 - 1); /* 1Mhz 的计数频率, 100Khz PWM */
    atim_timx_pwmmin_chy_init(); /* 初始化 PWM 输入捕获 */

    GTIM_TIMX_PWM_CHY_CCRX = 2; /* 低电平宽度 2, 高电平宽度 8 */
    while (1)
    {
        delay_ms(10);
        t++;
        if (t >= 20) /* 每 200ms 输出一次结果, 并闪烁 LED0, 提示程序运行 */
        {
            if (g_timxchy_pwmmin_sta) /* 捕获了一次数据 */
            {

```

```
printf("\r\n"); /* 输出空,另起一行 */
printf("PWM PSC :%d\r\n", g_timxchy_pwmmin_psc); /* 打印分频系数 */
printf("PWM Hight:%d\r\n", g_timxchy_pwmmin_hval); /* 打印高电平脉宽 */
printf("PWM Cycle:%d\r\n", g_timxchy_pwmmin_cval); /* 打印周期 */
/* 得到 PWM 采样时钟周期时间 */
tpsc = ((double)g_timxchy_pwmmin_psc + 1)/72;
ht = g_timxchy_pwmmin_hval * tpsc; /* 计算高电平时间 */
ct = g_timxchy_pwmmin_cval * tpsc; /* 计算周期长度 */
f = (1 / ct) * 1000000; /* 计算频率 */
printf("PWM Hight time:%.3fus\r\n", ht); /* 打印高电平脉宽长度 */
printf("PWM Cycle time:%.3fus\r\n", ct); /* 打印周期时间长度 */
printf("PWM Frequency :%.3fHz\r\n", f); /* 打印频率 */
atim_timx_pwmmin_chy_restart(); /* 重启 PWM 输入检测 */
}
LED0_TOGGLE(); /* LED0 (RED) 闪烁 */
t = 0;
}
}
```

先看 `gtim_timx_pwm_chy_init(10 - 1, 72 - 1)` 这个语句, 这两个形参分别设置自动重载寄存器的值为 9, 以及定时器预分频寄存器的值为 71。先看预分频系数, 我们设置为 72 分频, 定时器 1 的时钟频率等于 APB2 总线时钟频率, 即 72MHz, 可以得到计数器的计数频率是 1MHz, 即 1us 计数一次。再到自动重载寄存器的值为 9 决定的是 PWM 波的频率 (周期), 可以得到 PWM 的周期为 $10 \times 1\mu s = 10\mu s$ 。然后通过 `GTIM_TIMX_PWM_CHY_CCRX = 2` 这个语句设置占空比, 低电平宽度 2, 总的周期宽度是 10, 所以高电平宽度 8。即产生的 PWM 波周期为 10us, 频率为 100KHz, 占空比为 80%。下载验证的时候验证一下捕获到的与输出的是否一致。

`atim_timx_pwmmin_chy_init` 这个语句, 就初始化 PWM 输入捕获。然后在无限循环中每 200ms 判断是否 `g_timxchy_pwmmin_sta` 标志变量, 是否捕获到数据, 捕获到就打印和计数相关信息。

下面我们下载到开发板验证一下。

22.5.4 下载验证

下载代码后, 可以看到 LED0 在闪烁, 说明程序已经正常在跑了, 我们再打开串口调试助手, 选择对应的串口端口。然后用杜邦线把 PB5 引脚连接到 PC6 引脚, 就可以看到串口助手不断打印 PWM 波的信息, 如图 22.5.4.1 所示:

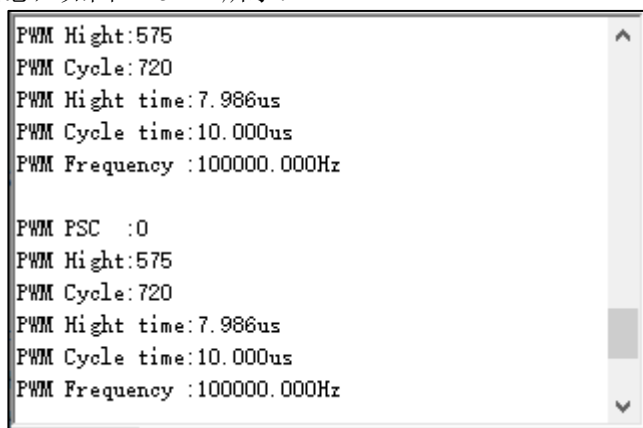


图 22.5.4.1 打印高电平脉冲次数

可以看到打印出来的 PWM 波信息为: 周期是 10us, 频率是 100KHz, 占空比是 80%, 和我们的预想结果一样。

大家可以通过 `gtim_timx_pwm_chy_init` 函数的形参设置其他参数的 PWM 波, 以及 `GTIM_TIMX_PWM_CHY_CCRX` 设置占空比。这里的测试的 PWM 波有一定的范围, 不是全范围的 PWM 都可以进行准确的测试, 大家可以进行验证。

第二十三章 电容触摸按键实验

上一章，我们介绍了 STM32F1 的输入捕获功能及其使用。这一章，我们将向大家介绍如何通过输入捕获功能，来做一个电容触摸按键。在本章中，我们将用 TIM5 的通道 2（PA1）来做输入捕获，并实现一个简单的电容触摸按键，通过该按键控制 LED1 的亮灭。

从本章分为如下几个部分：

- 23.1 电容触摸按键简介
- 23.2 硬件设计
- 23.3 软件设计
- 23.4 下载验证

23.1 电容触摸按键简介

前面我们学习过了机械按键，这节我们将介绍另一种人机交互设备：电容触摸按键。

电容式触摸按键已经广泛应用在家用电器、消费电子市场，其主要优势有：无机械装置，使用寿命长；非接触式感应，面板不需要开孔；产品更加美观简洁；防水可以做到很好。

战舰 F103 上的触摸按键 TPAD 其实就是一小块覆铜区域，其形状就是正点原子的 logo，如图 23.1.1 所示。



图 23.1.1 电容按键 TPAD 外观

与机械按键不同，这里我们使用的是检测电容充放电时间的方法来判断是否有触摸，图 23.1.2 中的 A、B 分别表示有无人体按下时电容的充放电曲线。其中 R 是外接的电容充电电阻，Cs 是没有触摸按下时 TPAD 与 PCB 之间的杂散电容。而 Cx 则是有手指按下时，手指与 TPAD 之间形成的电容。图中的开关是电容放电开关（实际使用时，由 STM32F1 的 IO 代替）。

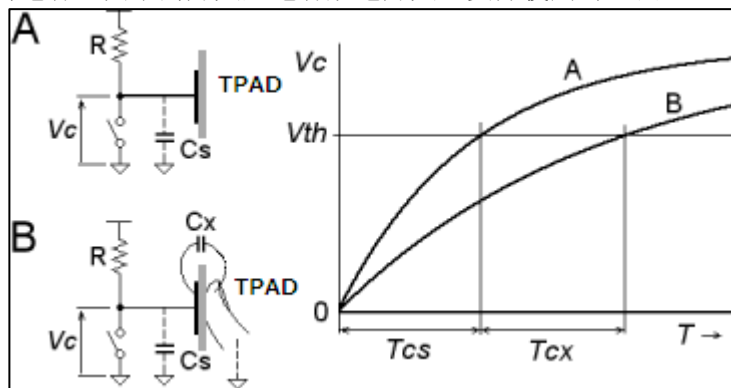


图 23.1.2 电容触摸按键原理

先用开关将 Cs（或 Cs+Cx）上的电放尽，然后断开开关，让 R 给 Cs（或 Cs+Cx）充电，当没有手指触摸的时候，Cs 的充电曲线如图中的 A 曲线。而当有手指触摸的时候，手指和 TPAD 之间引入了新的电容 Cx，此时 Cs+Cx 的充电曲线如图中的 B 曲线。从上图可以看出，A、B 两种情况下，Vc 达到 Vth 的时间分别为 Tcs 和 Tcs+Tcx。

其中，除了 Cs 和 Cx 我们需要计算，其他都是已知的，根据电容充放电公式：

$$V_c = V_0 \times (1 - e^{(-\frac{t}{RC})})$$

其中 Vc 为电容电压，V0 为充电电压，R 为充电电阻，C 为电容容值，e 为自然底数，t 为

充电时间。根据这个公式，我们就可以计算出 C_s 和 C_x 。利用这个公式，我们还可以把战舰开发板作为一个简单的电容计，直接可以测电容容量了，有兴趣的朋友可以捣鼓下。

在本章中，其实我们只要能够区分 T_{cs} 和 $T_{cs}+T_{cx}$ ，就已经可以实现触摸检测了，当充电时间在 T_{cs} 附近，就可以认为没有触摸，而当充电时间大于 $T_{cs}+T_x$ 时，就认为有触摸按下（ T_x 为检测阈值）。

本章，我们使用 PA1(TIM5_CH2)来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA1 为推挽输出，将电容 C_s （或 C_s+C_x ）放电，然后配置 PA1 为浮空输入，利用外部上拉电阻给电容 $C_s(C_s+C_x)$ 充电，同时开启 TIM5_CH2 的输入捕获，检测上升沿，当检测到上升沿的时候，就认为电容充电完成了，完成一次捕获检测。

在 MCU 每次复位重启的时候，我们执行一次捕获检测（可以认为没触摸），记录此时的值，记为 `g_tpad_default_val`，作为判断的依据。在后续的捕获检测，我们就通过与 `g_tpad_default_val` 的对比，来判断是不是有触摸发生。

关于输入捕获的配置，在前面已经有详细介绍了，这里我们就不再介绍。至此，电容触摸按键的原理介绍完毕。

23.2 硬件设计

1. 例程功能

LED0 用来指示程序运行，150ms 变换一次状态，即约 300ms 一次闪烁。不断扫描按键的状态，如果判定了电容触摸按键按下，我们就把 LED1 的状态翻转一次。

2. 硬件资源

1) LED 灯:

LED0 - PB5

LED1 - PE5

2) 定时器 5，使用 TIM5 通道 2，将 PA1 复用为 TIM5_CH1

3) GPIO: PA1，用于控制触摸按键 TPAD

3. 原理图

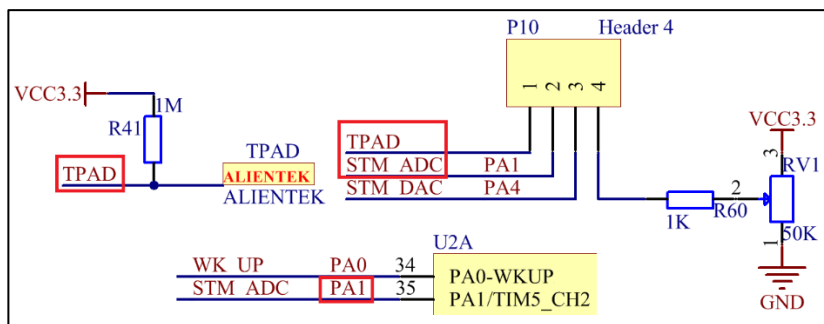


图 23.2.3.1 电容触摸按键 TPAD 连接原理图

由于设计时 PA1 不直接连接到电容触摸按键，而是引到了插针上，我们需要通过跳线帽把 P10 上标为 ADC 的引脚与标号为“TPAD”的标号连接到一起，如图 23.2.3.2 所示。

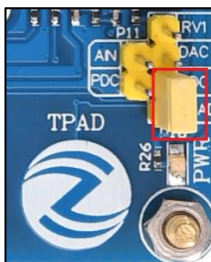


图 23.2.3.2 用跳线帽连接电容按键 TPAD 和 PA1

23.3 程序设计

我们在基本定时器一节已经学习过定时器的输入捕获功能，这里我们可以类似地，用定时器 5 来实现 TPAD 引脚上的电平状态进行捕获的功能。

本实验用到的 HAL 库驱动请回顾基本定时器实验的介绍。下面直接从程序流程图开始介绍。

23.3.1 程序流程图

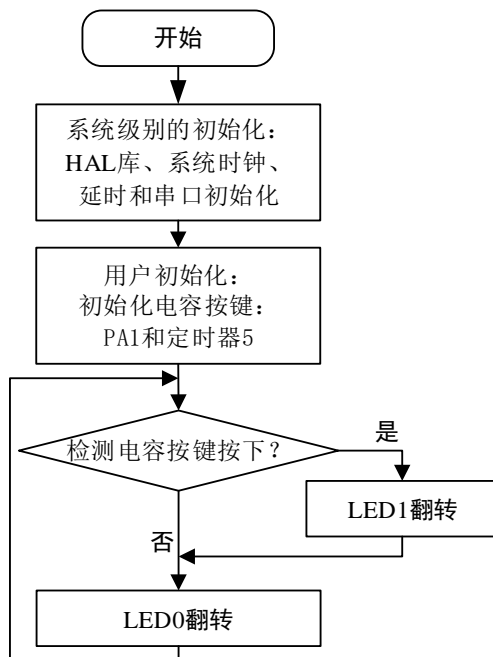


图 23.3.1.1 电容触摸按键实验程序流程图

23.3.2 程序解析

1. TPAD 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，TPAD 的驱动主要包括两个文件：tpad.c 和 tpad.h。

首先看 tpad.h 头文件的几个宏定义：

```

/* TPAD 引脚 及 定时器 定义 */
#define TPAD_GPIO_PORT      GPIOA
#define TPAD_GPIO_PIN      GPIO_PIN_1
#define TPAD_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define TPAD_TIMX_CAP      TIM5
#define TPAD_TIMX_CAP_CHY  TIM_CHANNEL_2 /* 通道 Y, 1<= Y <=4 */
#define TPAD_TIMX_CAP_CHY_CCRX TIM5->CCR2 /* 通道 Y 的捕获/比较寄存器 */
#define TPAD_TIMX_CAP_CHY_CLK_ENABLE() \
    do{ __HAL_RCC_TIM5_CLK_ENABLE(); }while(0) /* TIM5 时钟使能 */
  
```

PA1 是定时器 5 的 PWM 通道 2，如果我们使用其它定时器和它们对应的捕获通道的其它 IO，我们只需要修改上面的宏即可。

基于前面描述的触摸按键的原理：上电时检测 TPAD 上的电容的充放电时间，并以此为基础，每次需要重新检测 TPAD 时，通过比较充放电的时长来检测当前是否有按下。所以我们需要用定时器的输入捕获功能来监测 TPAD 充电时间。

首先，编写设置 TPAD 电容触摸按键的定时器输入捕获功能函数 tpad_timx_cap_init()，代码如下：


```

/**
 * @brief      触摸按键输入捕获设置
 * @param      arr      : 自动重装值
 * @param      psc      : 时钟预分频数
 * @retval     无
 */
static void tpad_timx_cap_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;
    TIM_IC_InitTypeDef timx_ic_cap_chy;
    TPAD_GPIO_CLK_ENABLE(); /* TPAD 引脚 时钟使能 */
    TPAD_TIMX_CAP_CHY_CLK_ENABLE(); /* 定时器 时钟使能 */

    gpio_init_struct.Pin = TPAD_GPIO_PIN; /* 输入捕获的 GPIO 口 */
    gpio_init_struct.Mode = GPIO_MODE_INPUT; /* 输入 */
    gpio_init_struct.Pull = GPIO_PULLDOWN; /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(TPAD_GPIO_PORT, &gpio_init_struct); /* TPAD 引脚下拉输入 */

    g_timx_cap_chy_handle.Instance = TPAD_TIMX_CAP; /* 定时器 5 */
    g_timx_cap_chy_handle.Init.Prescaler = psc; /* 定时器分频 */
    g_timx_cap_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 向上计数模式 */
    g_timx_cap_chy_handle.Init.Period = arr; /* 自动重装值 */
    g_timx_cap_chy_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1; /* 不分频 */
    HAL_TIM_IC_Init(&g_timx_cap_chy_handle);
    timx_ic_cap_chy.ICPolarity = TIM_ICPOLARITY_RISING; /* 上升沿捕获 */
    timx_ic_cap_chy.ICSelection = TIM_ICSELECTION_DIRECTTI; /* 映射 TIM1 */
    timx_ic_cap_chy.ICPrescaler = TIM_ICPSC_DIV1; /* 配置输入不分频 */
    timx_ic_cap_chy.ICFilter = 0; /* 配置输入滤波器，不滤波 */
    HAL_TIM_IC_ConfigChannel(&g_timx_cap_chy,
        &timx_ic_cap_chy, TPAD_TIMX_CAP_CHY); /* 配置 TIM5 通道 2 */
    HAL_TIM_IC_Start(&g_timx_cap_chy_handle, TPAD_TIMX_CAP_CHY); /* 使能输入捕获 */
}

```

这和我们《实验 9-3 通用定时器输入捕获实验》的代码基本一样，不同的是实验 9-3 代码中调用函数是 HAL_TIM_IC_Start_IT，使能输入捕获通道的同时开启了输入捕获中断，而该函数最后调用的是 HAL_TIM_IC_Start，只是开启了输入捕获通道并没有开启输入捕获中断。

接下来，我们通过控制变量法去检测 TPAD 电容触摸按键按下和没有按下的情况。每次先给 TPAD 放电（STM32 输出低电平）相同时间，然后释放，监测 VCC 每次给 TPAD 的充电时间，由此可以得到一个充电时间。这里会涉及到两个函数 tpad_reset、tpad_get_val，代码如下：

```

/**
 * @brief      复位 TPAD
 * @note      我们将 TPAD 按键看做是一个电容，当手指按下/不按下时容值有变化
 *            该函数将 GPIO 设置成推挽输出，然后输出 0，进行放电，然后再设置
 *            GPIO 为浮空输入，等待外部大电阻慢慢充电
 * @param      无
 * @retval     无
 */
static void tpad_reset(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    gpio_init_struct.Pin = TPAD_GPIO_PIN; /* 输入捕获的 GPIO 口 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(TPAD_GPIO_PORT, &gpio_init_struct);
    /* TPAD 引脚输出 0，放电 */
    HAL_GPIO_WritePin(TPAD_GPIO_PORT, TPAD_GPIO_PIN, GPIO_PIN_RESET);
    delay_ms(5);
    g_timx_cap_chy_handle.Instance->SR = 0; /* 清除标记 */
}

```

```

g_timx_cap_chy_handle.Instance->CNT = 0;          /* 归零 */

gpio_init_struct.Pin = TPAD_GPIO_PIN;            /* 输入捕获的 GPIO 口 */
gpio_init_struct.Mode = GPIO_MODE_INPUT;          /* 输入 */
gpio_init_struct.Pull = GPIO_NOPULL;              /* 浮空 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
HAL_GPIO_Init(TPAD_GPIO_PORT, &gpio_init_struct); /* TPAD 引脚浮空输入 */
}

/**
 * @brief      得到定时器捕获值
 * @note      如果超时，则直接返回定时器的计数值
 *            我们定义超时时间为：TPAD_ARR_MAX_VAL - 500
 * @param      无
 * @retval     捕获值/计数值（超时的情况下返回）
 */
static uint16_t tpad_get_val(void)
{
    uint32_t flag = (TPAD_TIMX_CAP_CHY== TIM_CHANNEL_1)?TIM_FLAG_CC1:\
                    (TPAD_TIMX_CAP_CHY== TIM_CHANNEL_2)?TIM_FLAG_CC2:\
                    (TPAD_TIMX_CAP_CHY== TIM_CHANNEL_3)?TIM_FLAG_CC3:TIM_FLAG_CC4;

    tpad_reset();
    while (__HAL_TIM_GET_FLAG(&g_timx_cap_chy_handle, flag) == RESET)
    { /* 等待通道 CHY 捕获上升沿 */
        if (g_timx_cap_chy_handler.Instance->CNT > TPAD_ARR_MAX_VAL - 500)
        {
            return g_timx_cap_chy_handle.Instance->CNT; /* 超时了,直接返回 CNT 的值 */
        }
    }

    return TPAD_TIMX_CAP_CHY_CCRX; /* 返回捕获/比较值 */
}

```

tpad_reset 函数，该函数顾名思义就是进行一次复位操作。先设置 PA1 输出低电平，电容放电，同时清除中断标志位并且计数器值清零，然后配置 PA1 为复用功能浮空输入，利用外部上拉电阻给电容 C 充电，同时开启 TIM5_CH2 的输入捕获。

而实现充电时间的获取主要是通过调用 **tpad_get_val** 函数。函数 **tpad_get_val** 得到定时器的一次捕获值，函数内部先调用 **tpad_reset**，复位 TPAD 电容触摸按键，让其处于被充电状态，后面捕获到上升沿就把捕获到的值（或溢出值）作为返回值返回。

得到充电时间后，接下来我们要做的就是获取没有按下 TPAD 时的充电时间，并把它作为基准来确认后续有无按下操作，我们定义全局变量 **g_tpad_default_val** 来保存这个值，通过多次平均的滤波算法来减小误差，编写的初始化函数 **tpad_init** 代码如下。

```

/**
 * @brief      初始化触摸按键
 * @param      psc : 分频系数(值越小，越灵敏，最小值为：1)
 * @retval     0，初始化成功；1，初始化失败；
 */
uint8_t tpad_init(uint16_t psc)
{
    uint16_t buf[10];
    uint16_t temp;
    uint8_t j, i;
    tpad_timx_cap_init(TPAD_ARR_MAX_VAL, psc - 1); /* 以 72/(psc-1)Mhz 的频率计数 */

    for (i = 0; i < 10; i++) /* 连续读取 10 次 */
    {
        buf[i] = tpad_get_val();
        delay_ms(10);
    }
}

```

```

for (i = 0; i < 9; i++)          /* 排序 */
{
    for (j = i + 1; j < 10; j++)
    {
        if (buf[i] > buf[j])      /* 升序排列 */
        {
            temp = buf[i];
            buf[i] = buf[j];
            buf[j] = temp;
        }
    }
}

temp = 0;

for (i = 2; i < 8; i++)          /* 取中间的 6 个数据进行平均 */
{
    temp += buf[i];
}

g_tpad_default_val = temp / 6;
printf("g_tpad_default_val:%d\r\n", g_tpad_default_val);

if (g_tpad_default_val > TPAD_ARR_MAX_VAL / 2)
{
    return 1;    /* 初始化遇到超过 TPAD_ARR_MAX_VAL/2 的数值,不正常! */
}

return 0;
}

```

得到无按下时 TPAD 电容按键的充电初始值后，我们需编写一个按键扫描函数，以便在需要监控 TPAD 的地方调用，代码如下：

```

/**
 * @brief      扫描触摸按键
 * @param      mode      : 扫描模式
 * @arg        0, 不支持连续触发 (按下一次必须松开才能按下一次);
 * @arg        1, 支持连续触发 (可以一直按下)
 * @retval     0, 没有按下; 1, 有按下;
 */
uint8_t tpad_scan(uint8_t mode)
{
    static uint8_t keyen = 0;    /* 0, 可以开始检测; >0, 还不能开始检测; */
    uint8_t res = 0;
    uint8_t sample = 3;          /* 默认采样次数为 3 次 */
    uint16_t rval;

    if (mode)
    {
        sample = 6;    /* 支持连接的时候, 设置采样次数为 6 次 */
        keyen = 0;    /* 支持长按, 每次调用该函数都可以检测 */
    }
    rval = tpad_get_maxval(sample);
    if (rval > (g_tpad_default_val + TPAD_GATE_VAL))
    {
        /* 大于 tpad_default_val+TPAD_GATE_VAL, 有效 */
        if (keyen == 0)
        {
            res = 1;    /* keyen==0, 有效 */
        }

        //printf("r:%d\r\n", rval);    /* 输出计数值, 调试的时候才用到 */
        keyen = 3;    /* 至少要再过 3 次之后才能按键有效 */
    }
    if (keyen) keyen--;
}

```

```
return res;
}
```

函数 `tpad_scan` 用于扫描 TPAD 是否有触摸，该函数的参数 `mode`，用于设置是否支持连续触发。返回值如果是 0，说明没有触摸，如果是 1，则说明有触摸。该函数同样包含了一个静态变量，用于检测控制，类似按键实验中的 `key_scan` 函数。所以该函数同样是不可重入的。在函数中，我们通过连续读取 3 次（不支持连续按的时候）TPAD 的值，取最大值和 `g_tpad_default_val` 作比较，如果他们的差值大于 `TPAD_GATE_VAL` 阈值（自定义的阈值）说明有触摸，如果小于则说明无触摸。上述取最大值的操作会依靠 `tpad_get_maxval` 函数，通过 `n` 次调用 `tpad_get_val` 采集捕获值，然后进行比较后获取 `n` 次采样值中的最大值，函数比较简单就不列出来了。

2. main.c 代码

在 `main.c` 里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    tpad_init(6); /* 初始化触摸按键 */

    while (1)
    {
        if (tpad_scan(0)) /* 成功捕获到了一次上升沿 (此函数执行时间至少 15ms) */
        {
            LED1_TOGGLE(); /* LED1 取反 */
        }

        t++;
        if (t == 15)
        {
            t = 0;
            LED0_TOGGLE(); /* LED0 取反 */
        }
        delay_ms(10);
    }
}
```

`main` 函数比较简单，`tpad_init(6)` 函数执行之后，就开始触摸按键的扫描。当有触摸时，对 LED1 状态取反，而 LED0 则有规律的间隔取反，提示程序正在运行。

这里要提醒一下大家，不要把 `usart_init(115200)` 去掉，因为在 `tpad_init` 函数里面，我们有用到 `printf`，如果你去掉了 `usart_init`，就会导致 `printf` 无法执行，从而死机。

至此，我们的软件设计就完成了。

23.4 下载验证

下载代码后，可以看到 LED0 不停闪烁（每 300ms 闪烁一次），用手指按下电容按键时，LED1 的状态发生改变（亮灭交替一次）。这里记得 TPAD 引脚和 PA1 都是连接到开发板上的排针上的，开始测试前需要连接好，否则测试就不准了，如果下载代码前没有连接好，请连接后复位重新测试即可。

第二十四章 OLED 显示实验

本章我们来学习使用 OLED 液晶显示屏，在开发板上我们预留了 OLED 模块接口，需要准备一个 OLED 显示模块。下面我们一起来点亮 OLED，并实现 ASCII 字符的显示。

本章分为如下几个小节：

24.1 OLED 简介

24.2 硬件设计

24.3 程序设计

24.4 下载验证

24.1 OLED 简介

OLED，即有机发光二极管（Organic Light-Emitting Diode），又称为有机电激光显示（Organic Electroluminescence Display，OLED）。OLED 可按发光材料分为两种：小分子 OLED 和高分子 OLED（也可称为 PLED）。OLED 是一种利用多层有机薄膜结构产生电致发光的器件，它很容易制作，而且只需要低的驱动电压，OLED 由于同时具备自发光（不需背光源）、对比度高、厚度薄、视角广、反应速度快、功耗低、柔性好等优异特性，目前主要用于显示领域，OLED 在节能照明领域的开发也成为全球趋势。

本章我们将介绍 ALINETEK 的 OLED 显示模块及其使用方法，该模块有以下特点：

- 1) 模块有单色和双色两种可选，单色为纯蓝色，而双色则为黄蓝双色（分区域的双色，前 16 行为黄色，后 48 行为蓝色，且黄蓝色之间有一行不显示的间隔区。）。
- 2) 尺寸小，显示尺寸为 0.96 寸，而模块的尺寸仅为 27mm*26mm 大小。
- 3) 高分辨率，该模块的分辨率为 128*64。
- 4) 多种接口方式，该模块提供了总共 4 种接口包括：6800、8080 两种并行接口方式、4 线 SPI 接口方式以及 IIC 接口方式（只需要 2 根线就可以控制 OLED 了！）。
- 5) 不需要高压，直接接 3.3V 就可以工作了。

这里要提醒大家的是，该模块不和 5.0V 接口兼容，所以请大家在使用的时候一定要小心，别直接接到 5V 的系统上去，否则可能烧坏模块。以下 4 种模式通过模块的 BS1 和 BS2 设置，BS1 和 BS2 的设置与模块接口模式的关系如表 24.1.1 所示：

| 接口方式 | 4 线 SPI | IIC | 8 位 6800 | 8 位 8080 |
|------|---------|-----|----------|----------|
| BS1 | 0 | 1 | 0 | 1 |
| BS2 | 0 | 0 | 1 | 1 |

表 24.1.1 OLED 模块接口方式设置表

表 24.1.1 中：“1”代表接 VCC，而“0”代表接 GND。该模块的外观图如图 24.1.1 所示：

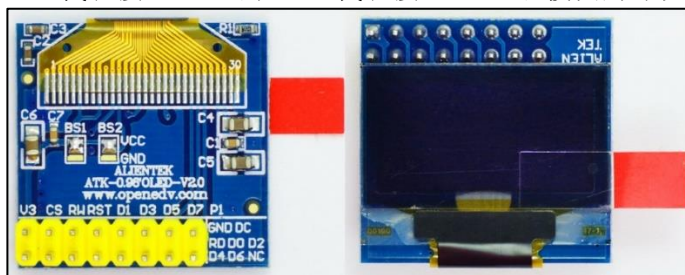


图 24.1.1 正点原子 OLED 模块外观图

正点原子 OLED 模块默认设置是：BS1 和 BS2 接 VCC，即使用 8080 并口方式，如果你想设置为其他模式，则需要在 OLED 的背面，用烙铁修改 BS1 和 BS2 的设置。模块的原理图如图 24.1.2 所示：

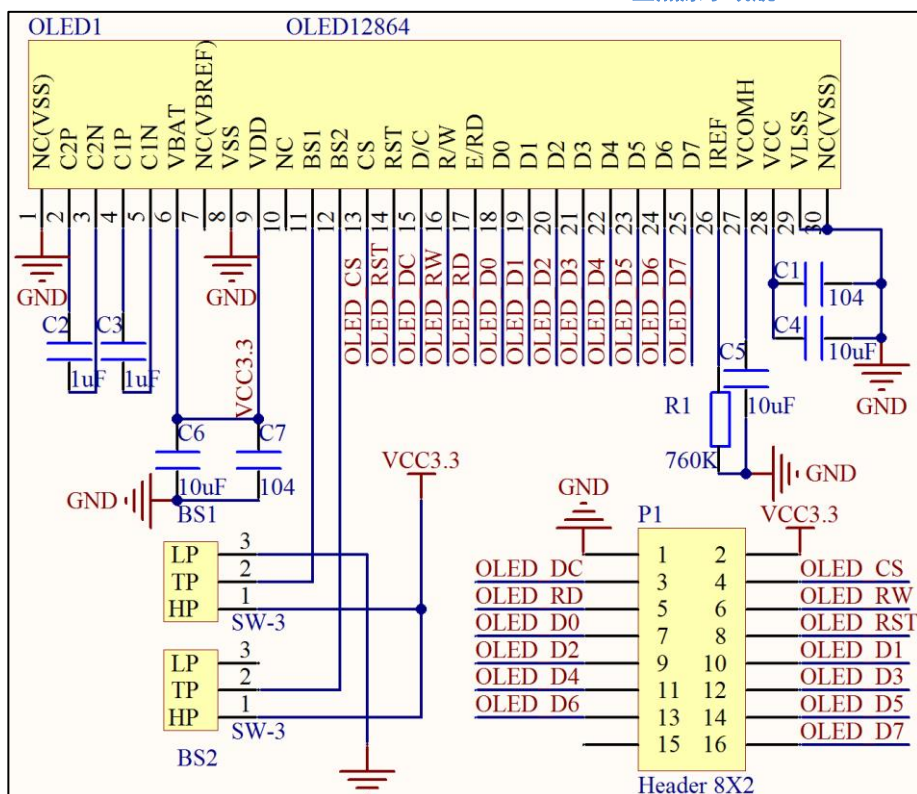


图 24.1.2 正点原子 OLED 模块原理图

该模块采用 8*2 的 2.54 排针与外部连接，总共有 16 个管脚，在 16 条线中，我们只用了 15 条，有一个是悬空的。15 条线中，电源和地线占了 2 条，还剩下 13 条信号线。在不同模式下，我们需要的信号线数量是不同的，在 8080 模式下，需要全部 13 条，而在 IIC 模式下，仅需要 2 条线就够了！这其中有一条是共同的，那就是复位线 RST (RES)，RST 上的低电平，将导致 OLED 复位，在每次初始化之前，都应该复位一下 OLED 模块。

正点原子 OLED 模块的控制器是 SSD1306，本章，我们将学习如何通过 STM32F103 来控制该模块显示字符和数字，本章的实例代码将可以支持两种方式与 OLED 模块连接，一种是 8080 的并口方式，另外一种为 4 线 SPI 方式。实际使用过程我们也通常只选用其中的一种来实现硬件上的连接，我们会分别介绍这两种模式，读者可以选择性阅读。

24.1.1 硬件驱动接口模式

1. 8080 并口模式

首先我们介绍一下模块的 8080 并行接口，8080 并行接口的发明者是 INTEL，该总线也被广泛应用于各类液晶显示器，正点原子 OLED 模块也提供了这种接口，使得 MCU 可以快速的访问 OLED。正点原子 OLED 模块的 8080 接口方式需要如下一些信号线：

CS: OLED 片选信号。

WR: 向 OLED 写入数据。

RD: 从 OLED 读取数据。

D[7: 0]: 8 位双向数据线。

RST(RES): 硬复位 OLED。

DC: 命令/数据标志 (0, 读写命令; 1, 读写数据)。

模块的 8080 并口写的过程为：先根据要写入的数据的类型，设置 DC 为高（数据）/低（命令），设置 WR 起始电平为高，然后拉低片选，选中 SSD1306，接着我们在整个读时序上保持 RD 为高电平，然后拉低 WR 的电平准备写入数据，向数据线（D[7: 0]）上输入要写的信息；拉高 WR，这样得到一个 WR 的上升沿，在这个上升沿，使数据写入到 SSD1306 里面；

SSD1306 的 8080 并口写时序图如图 24.1.1.1 所示：

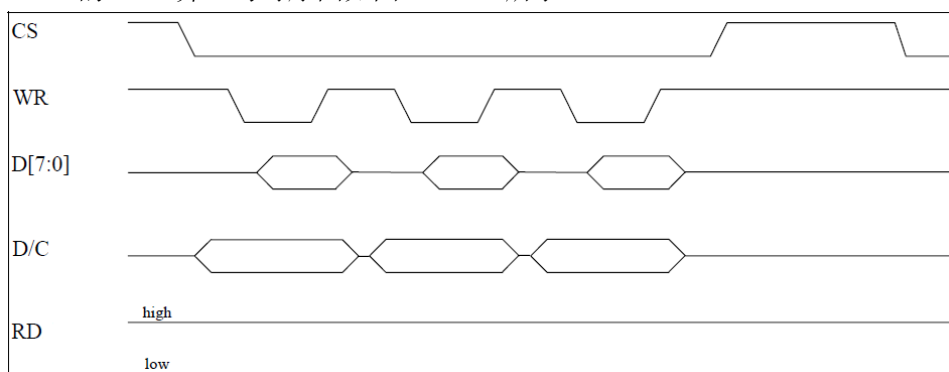


图 24.1.1.1 8080 并口写时序图

模块的 8080 并口读的过程为：先根据要写入的数据的类型，设置 DC 为高（数据）/低（命令），设置 RD 起始电平为高，然后拉低片选 CS 信号，选中 SSD1306，接着我们在整个读时序上保持 WR 为高电平，然后类似写时序，同样的，在 RD 的上升沿，使数据锁存到数据线（D[7:0]）上；SSD1306 的 8080 并口读时序图如图 24.1.1.2 所示：

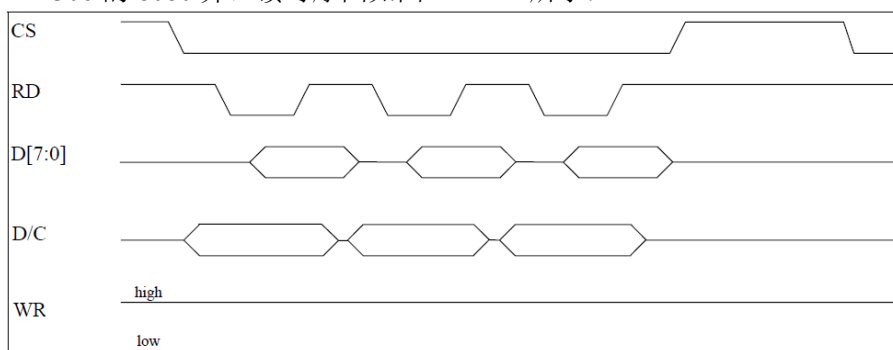


图 24.1.1.2 8080 并口读时序图

SSD1306 的 8080 接口方式下，控制脚的信号状态所对应的功能如表 24.1.1.1：

| 功能 | RD | WR | CS | DC |
|-----|----|----|----|----|
| 写命令 | H | ↑ | L | L |
| 读状态 | ↑ | H | L | L |
| 写数据 | H | ↑ | L | H |
| 读数据 | ↑ | H | L | H |

表 24.1.1.1 控制脚信号状态功能表

在 8080 方式下读数据操作的时候，我们有时候（例如读显存的时候）需要一个假读命（Dummy Read），以使得微控制器的操作频率和显存的操作频率相匹配。在读取真正的数据之前，由一个的假读的过程。这里的假读，其实就是第一个读到的字节丢弃不要，从第二个开始，才是我们真正要读的数据。

一个典型的读显存的时序图，如图 24.1.1.3 所示：

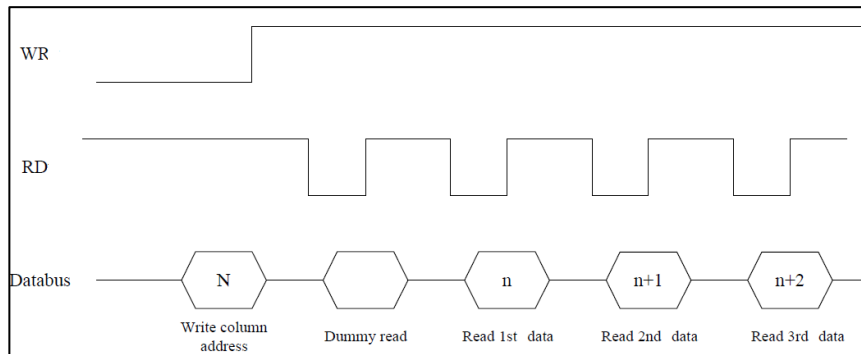


图 24.1.1.3 读显存时序图

可以看到，在发送了列地址之后，开始读数据，第一个是 Dummy Read，也就是假读，我们从第二个开始，才算是真正有效的数据。并行接口模式就介绍到这里。

2. SPI 模式

我们的代码同时兼容 SPI 方式的驱动，如果你使用的是这种驱动方式，则应该把代码中的宏 OLED_MODE 设置为 0，但对于硬件，则需要查看 PCB 背面的电阻设置以确定当前使用的是否为 SPI 模式：

```
#define OLED_MODE 0 /* 0: 4 线串行模式 */
```

我们接下来介绍一下 4 线串行（SPI）方式，4 线串口模式使用的信号线有如下几条：

CS: OLED 片选信号。

RST(RES): 硬复位 OLED。

DC: 命令/数据标志（0，读写命令；1，读写数据）。

SCLK: 串行时钟线。在 4 线串行模式下，D0 信号线作为串行时钟线 SCLK。

SDIN: 串行数据线。在 4 线串行模式下，D1 信号线作为串行数据线 SDIN。

模块的 D2 需要悬空，其他引脚可以接到 GND。在 4 线串行模式下，只能往模块写数据而不能读数据。

在 4 线 SPI 模式下，每个数据长度均为 8 位，在 SCLK 的上升沿，数据从 SDIN 移入到 SSD1306，并且是高位在前的。DC 线还是用作命令/数据的标志线。在 4 线 SPI 模式下，写操作的时序如图 24.1.1.4 所示：

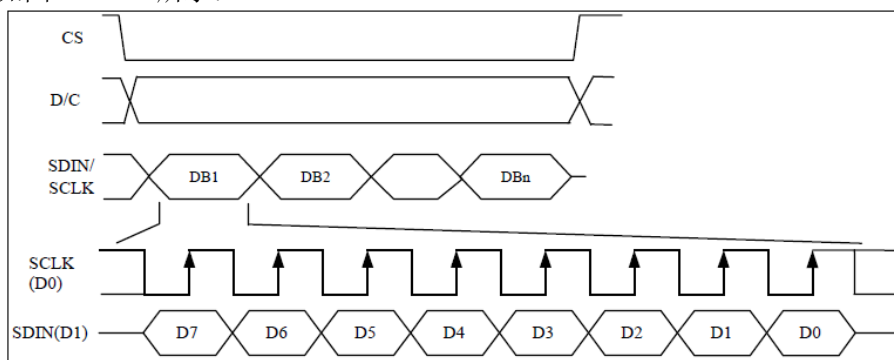


图 24.1.1.4 4 线 SPI 写操作时序图

4 线串行模式就为大家介绍到这里。其他还有几种模式，可以参考 SSD1306 的数据手册的介绍，我们把资料放到“开发板资料 A 盘->7, 硬件资料\3, 液晶资料\OLED 资料\SSD1306-Revision 1.1(Charge Pump).pdf”，如果要使用这些方式，请大家参考该手册并自行实现相应的功能代码。

24.1.2 OLED 显存

接下来，我们介绍一下模块的显存，SSD1306 的显存总共为 128*64bit 大小，SSD1306 将这些显存分为了 8 页，不使用显存对应的行列的重映射，其对应关系如表 24.1.2.1 所示：

| 列 (COM0~63) | 行 (COL0~127) | | | | | | |
|----------------|--------------|------|------|-------|--------|--------|--------|
| | SEG0 | SEG1 | SEG2 | | SEG125 | SEG126 | SEG127 |
| | PAGE0 | | | | | | |
| | PAGE1 | | | | | | |
| | PAGE2 | | | | | | |
| | PAGE3 | | | | | | |
| | PAGE4 | | | | | | |
| | PAGE5 | | | | | | |
| | PAGE6 | | | | | | |
| | PAGE7 | | | | | | |

表 24.1.2.1 SSD1306 显存与屏幕对应关系表

可以看出，SSD1306 的每页包含了 128 个字节，总共 8 页，这样刚好是 128*64 的点阵大小。当 GRAM 的写入模式为页模式时，需要设置低字节起始的列地址（0x00~0x0F）和高字

节的起始列地址（0x10~0x1F），芯片手册中给出了写入 GRAM 与显示的对应关系，写入列地址在写完一字节后自动按列增长，如图 24.1.2.2 所示：

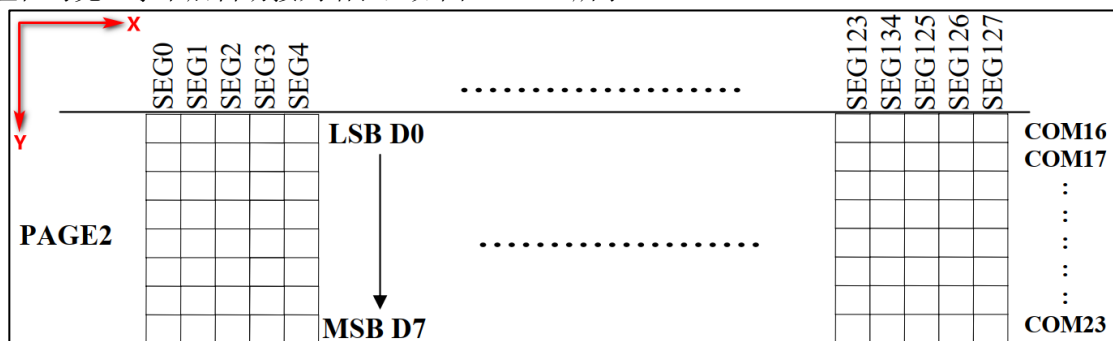


图 24.1.2.2 SSD1306 页 2 显存写入字节与屏幕坐标的关系

因为每次写入都是按字节写入的，这就存在一个问题，如果我们使用只写方式操作模块，那么，每次要写 8 个点，这样，我们在画点的时候，就必须把要设置的点所在的字节的每个位都搞清楚当前的状态（0/1？），否则写入的数据就会覆盖掉之前的状态，结果就是有些不需要显示的点，显示出来了，或者该显示的没有显示了。这个问题在能读的模式下，我们可以先读出来要写入的那个字节，得到当前状况，在修改了要改写的位之后再写进 GRAM，这样就不会影响到之前的状况了。但是这样需要能读 GRAM，对于 4 线 SPI 模式/IIC 模式，模块是不支持读的，而且读→改→写的方式速度也比较慢。

所以我们采用的办法是在 STM32F103 的内部建立一个虚拟的 OLED 的 GRAM（共 128*8=1024 个字节），每次修改时，只修改 STM32F103 上的 GRAM（实际上就是 SRAM），在修改完成后一次性把 STM32F103 上的 GRAM 写入到 OLED 的 GRAM。当然这个方法也有坏处，一个对于那些 SRAM 很小的单片机（比如 51 系列）不太友好，另一个是每次都写入全屏，屏幕刷新率会变低。

SSD1306 的命令比较多，这里我们仅介绍几个比较常用的命令，如表 24.1.2.3 所示：

| 序号 | 指令 | 各位描述 | | | | | | | | 命令 | 说明 |
|----|--------|------|----|----|----|----|----|----|----|----------|----------------------------|
| | HEX | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | |
| 0 | 81 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 设置对比度 | A的值越大屏幕越亮，A的范围从0X00~0XFF |
| | A[7:0] | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | | |
| 1 | AE/AF | 1 | 0 | 1 | 0 | 1 | 1 | 1 | X0 | 设置显示开关 | X0=0, 关闭显示； X0=1, 开启显示； |
| 2 | 8D | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 电荷泵设置 | A2=0, 关闭电荷泵 A2=1, 开启电荷泵 |
| | A[7:0] | * | * | 0 | 1 | 0 | A2 | 0 | 0 | | |
| 3 | B0~B7 | 1 | 0 | 1 | 1 | 0 | X2 | X1 | X0 | 设置页地址 | X[2:0]=0~7对应页0~7 |
| 4 | 00~0F | 0 | 0 | 0 | 0 | X3 | X2 | X1 | X0 | 设置列地址低四位 | 设置8位起始列地址的低四位 |
| 5 | 10~1F | 0 | 0 | 0 | 0 | X3 | X2 | X1 | X0 | 设置列地址高四位 | 设置8位起始列地址的高四位 |

表 24.1.2.3 SSD1306 常用命令表

第 0 个命令为 0X81，用于设置对比度的，这个命令包含了两个字节，第一个 0X81 为命令，随后发送的一个字节为要设置的对比度的值。这个值设置得越大屏幕就越亮。

第 1 个命令为 0XAE/0XAF。0XAE 为关闭显示命令；0XAF 为开启显示命令。

第 2 个命令为 0X8D，该指令也包含 2 个字节，第一个为命令字，第二个为设置值，第二个字节的 BIT2 表示电荷泵的开关状态，该位为 1，则开启电荷泵，为 0 则关闭。在模块初始化的时候，这个必须要开启，否则是看不到屏幕显示的。

第 3 个命令为 0XB0~B7，该命令用于设置页地址，其低三位的值对应着 GRAM 的页地址。

第 4 个指令为 0X00~0X0F，该指令用于设置显示时的起始列地址低四位。

第 6 个指令为 0X10~0X1F，该指令用于设置显示时的起始列地址高四位。

其他命令，我们就不在这里一一介绍了，大家可以参考 SSD1306 datasheet 的第 28 页。从

这页开始，对 SSD1306 的指令有详细的介绍。

最后，我们再来介绍一下 OLED 模块的初始化过程，SSD1306 典型初始化框图如图 24.1.2.2：

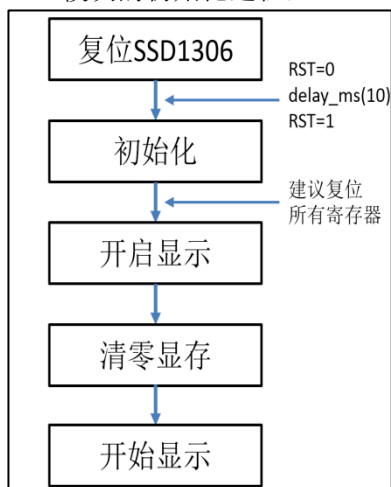


图 24.1.2.2 SSD1306 初始化框图

驱动 IC 的初始化代码，我们直接使用厂家推荐的设置就可以了，只要对细节部分进行一些修改，使其满足我们自己的要求即可，其他不需要变动。

OLED 的介绍就到此为止，我们重点向大家介绍了正点原子 OLED 模块的相关知识，接下来我们将使用这个模块来显示字符和数字。通过以上介绍，我们可以得出 OLED 显示需要的相关设置步骤如下：

1) 设置 STM32F103 与 OLED 模块相连接的 IO。

这一步，先将我们与 OLED 模块相连的 IO 口设置为输出，具体使用哪些 IO 口，这里需要根据连接电路以及 OLED 模块所设置的通讯模式来确定。这些将在硬件设计部分向大家介绍。

2) 初始化 OLED 模块。

其实这里就是上面的初始化框图的内容，通过对 OLED 相关寄存器的初始化，来启动 OLED 的显示。为后续显示字符和数字做准备。

3) 通过函数将字符和数字显示到 OLED 模块上。

这里就是通过我们设计的程序，将要显示的字符送到 OLED 模块就可以了，这些函数将在软件设计部分向大家介绍。

通过以上三步，我们就可以使用正点原子 OLED 模块来显示字符和数字了，在后面我们还将给大家介绍显示汉字的方法。这一部分就先介绍到这里。

24.2 硬件设计

1. 例程功能

使用 8080 并口模式驱动或者使用 4 线 SPI 串口模式，驱动 OLED 模块，不停的显示 ASCII 码和码值。LED0 闪烁，提示程序运行。由于我们还没有学到 SPI 模式驱动，我们这时仅先介绍 8080 模式，感兴趣的读者也可以先进行测试，也可以等我们学习完 SPI 的知识再回过头来学习这个模块的 SPI 驱动的用法。

2. 硬件资源

1) LED 灯

DS0: LED0 - PB5

2) 正点原子 0.96 寸 OLED 模块，在硬件上，OLED 与开发板的 IO 口对应关系如下：

OLED_CS 对应 OV_WRST，即：PD6；

OLED_RS 对应 OV_SCL，即：PD3，高电平为命令，低电平为数据；

OLED_WR 对应 OV_RRST，即：PG14；

OLED_RD 对应 DCMI_SDA，即：PG13；

OLED_RST 对应 DCMI_RESET，即：PG15；
 OLED_D[7:0]对应 DCMI_D[7:0]，即：PC[7:0]

3. 原理图

OLED 模块的原理图在前面已有详细说明了，这里我们介绍 OLED 模块与我们开发板的连接，开发板上有一个 OLED/CAMERA 的接口（P6 接口）可以和正点原子 OLED 模块直接对插（**靠左插!**），连接如图 24.2.1 所示：



图 24.2.1 OLED 模块与开发板连接示意图

开发板上的这些 IO 与 OLED 模块的 IO 的对应关系，我们通过对比原理图，就可以得到以下表格的对应关系：

| 开发板 IO | OLED 模块 IO | STM32 芯片对应 GPIO |
|---------|------------|-----------------|
| VCC3.3 | VCC3.3 | |
| OV_WRST | OLED_CS | PD6 |
| OV_RRST | OLED_RW | PG14 |
| OV_OE | OLED_RST | PG15 |
| OV_D1 | OLED_D1 | PC1 |
| OV_D3 | OLED_D3 | PC3 |
| OV_D5 | OLED_D5 | PC5 |
| OV_D7 | OLED_D7 | PC7 |
| GND | GND | |
| OV_SCL | OLED_DC | PD3 |
| OV_SDA | OLED_RD | PG13 |
| OV_D0 | OLED_D0 | PC0 |
| OV_D2 | OLED_D2 | PC2 |
| OV_D4 | OLED_D4 | PC4 |
| OV_D6 | OLED_D6 | PC6 |
| OV_RCLK | 悬空 | 悬空 |

表 24.2.1 OLED 模块与开发板连接示意图

这些线的连接，开发板的内部已经连接好了，我们只需要将 OLED 模块插上去就好了，注意，这里的 OLED_D[7:0] 我们接到连续的 GPIOC[7:0]上一一对应，所以只需要读一次 PC 就可以得到对应的显示数据，后续会介绍。实物连接如图 24.2.2 所示：



图 24.2.2 OLED 模块与开发板连接实物图

24.3 程序设计

OLED 的驱动程序我们主要讲解 8080 并口的方式，用我们之前学过的 GPIO 的读写的知识就可以完成对 OLED 的控制了。

24.3.1 程序流程图

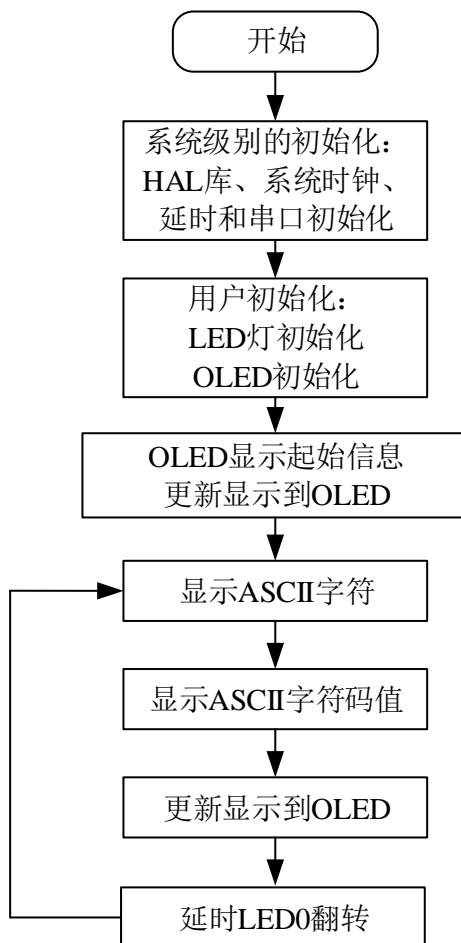


图 24.3.1.1 OLED 实验流程图

24.3.2 程序解析

1. OLED 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，OLED 的驱动主要包括三个文件：oled.c、oled.h 和 oledfont.h。oledfont.h 头文件存放的是 ASCII 字符集，oled.h 存放的是引脚接口宏定义和函数声明等，oled.c 则是驱动代码。

首先看 oledfont.h 头文件的 ASCII 字符集内容：

```

/* 常用 ASCII 表
 * 偏移量 32
 * ASCII 字符集: !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
 * PC2LCD2002 取模方式设置: 阴码+逐列式+顺向+C51 格式
 * 总共: 3 个字符集 (12*12、16*16 和 24*24)，用户可以自行新增其他分辨率的字符集。
 * 每个字符所占用的字节数为: (size/8+((size%8)?1:0))*(size/2), 其中 size: 是字库生成时的点
 * 阵大小 (12/16/24...)
 */
/* 12*12 ASCII 字符集点阵 */
    
```

```
const unsigned char oled_asc2_1206[95][12]={ ...这里省略字符集库... };
/* 16*16 ASCII 字符集点阵 */
const unsigned char oled_asc2_1608[95][16]={ ...这里省略字符集库... };
/* 24*24 ASCII 字符集点阵 */
const unsigned char oled_asc2_2412[95][36]={ ...这里省略字符集库... };
```

该头文件中包含三个大小不同的 ASCII 字符集点阵，其中包括：12*12 ASCII 字符集点阵、16*16 ASCII 字符集点阵、24*24 ASCII 字符集点阵。每个字符集点阵都包含 95 个常用的 ASCII 字符集，从空格符开始（即 ASCII 码表编号的 32~127 对应的字符），分别为：!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~。

上面的 ASCII 字符集，我们可以使用正点原子团队开发的一个款很好的字符提取软件来制作获取。字符提取软件为：ATK_XFONT，该软件可以提供各种字符，包括汉字（字体和大小都可以自己设置）阵提取，且取模方式可以设置好几种，常用的取模方式，该软件都支持。该软件还支持图形模式，也就是用户可以自己定义图片的大小，然后画图，根据所画的图形再生成点阵数据，这功能在制作图标或图片的时候很有用。

该软件的界面如图 24.3.2.1 所示：

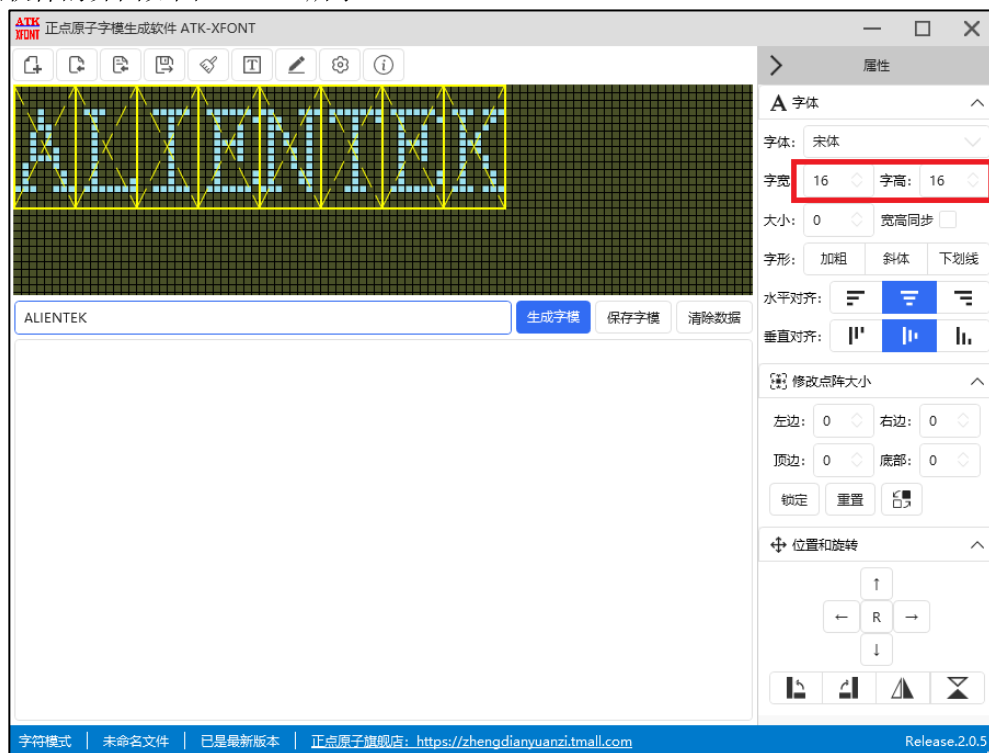


图 24.3.2.1 ATK_XFONT 软件界面

然后我们选择设置，在设置里面设置取模方式如图 24.3.2.2 所示：

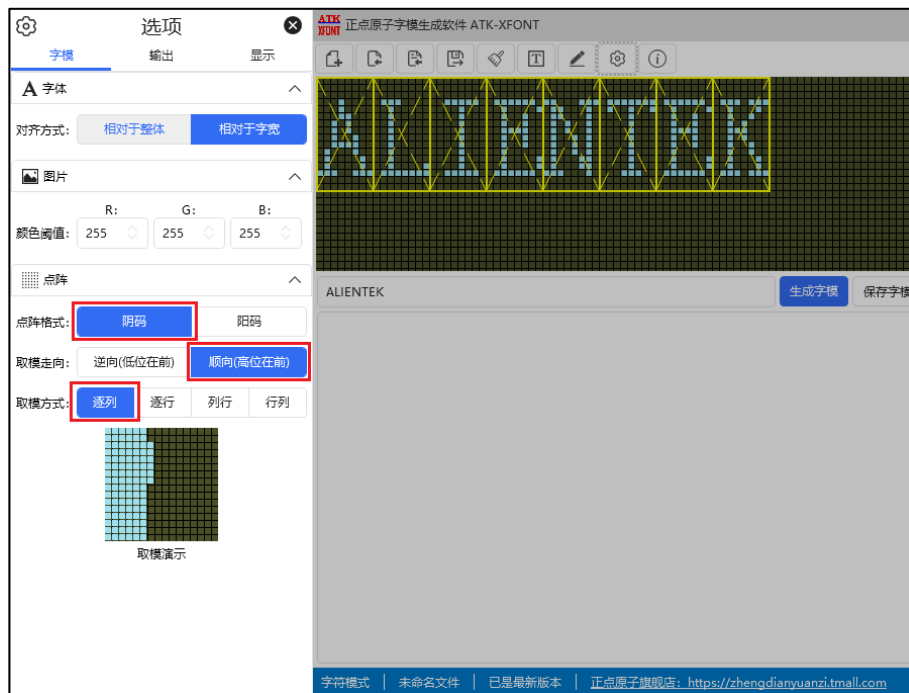


图 24.3.2.2 设置取模方式

上图设置的取模方式，在右上角的取模说明里面有，即：从第一列开始向下每取 8 个点作为一个字节，如果最后不足 8 个点就补满 8 位。取模顺序是从高到低，即第一个点作为最高位。如*-----取为 10000000。其实就是按如图 24.3.2.3 所示的这种方式：

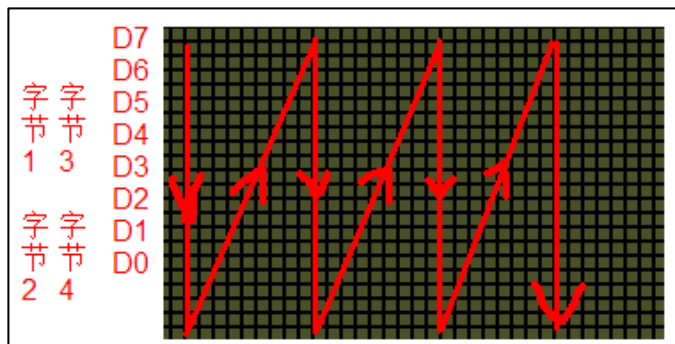


图 24.3.2.3 取模方式图解

从上到下，从左到右，高位在前。我们按这样的取模方式，然后把 ASCII 字符集按 12*6 大小、16*8 和 24*12 大小取模出来（对应汉字大小为 12*12、16*16 和 24*24，字符的只有汉字的一半大！），每个 12*6 的字符占用 12 个字节，每个 16*8 的字符占用 16 个字节，每个 24*12 的字符占用 36 个字节。

oled.c 和 oled.h 文件的代码可以帮助显示我们制作好的字符集。我们还是先看 oled.h 文件的宏定义，首先是 OLED 模式设置宏定义：

```
/* OLED 模式设置
 * 0: 4 线串行模式    (模块的 BS1, BS2 均接 GND)
 * 1: 并行 8080 模式  (模块的 BS1, BS2 均接 VCC)
 */
#define OLED_MODE 1 /* 默认使用 8080 并口模式 */
```

软件中通过宏定义 OLED_MODE 来决定使用 4 线串行模式（0）还是并行 8080 模式（1），由于我们的硬件默认使用 8080 并口模式，所以程序中默认设置这个宏为 1。

关于 OLED 8080 并口模式和 SPI 模式的引脚定义就不列出来了，请看源码。

还有两个关于向 OLED 写入选择命令或者数据的宏定义，后面讲的 oled_wr_byte 函数用到。

```
/* 命令/数据 定义 */
#define OLED_CMD 0 /* 写命令 */
```

```
#define OLED_DATA 1 /* 写数据 */
```

最后就是 oled.c 文件的驱动源码介绍。先是 OLED(SSD1306)的初始化函数，以 8080 并口方式为例，其定义如下：

```
/**
 * @brief      初始化 OLED(SSD1306)
 * @param      无
 * @retval     无
 */
void oled_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    HAL_RCC_GPIOC_CLK_ENABLE(); /* 使能 PORTC 时钟 */
    HAL_RCC_GPIOD_CLK_ENABLE(); /* 使能 PORTD 时钟 */
    HAL_RCC_GPIOG_CLK_ENABLE(); /* 使能 PORTG 时钟 */

    /* PC0 ~ 7 设置 */
    gpio_init_struct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3|
        GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(GPIOC, &gpio_init_struct); /* PC0 ~ 7 设置 */

    gpio_init_struct.Pin = GPIO_PIN_3|GPIO_PIN_6; /* PD3, PD6 设置 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(GPIOD, &gpio_init_struct); /* PD3, PD6 设置 */

    gpio_init_struct.Pin = GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM; /* 中速 */
    HAL_GPIO_Init(GPIOG, &gpio_init_struct); /* WR/RD/RST 引脚模式设置 */

    OLED_WR(1);
    OLED_RD(1);
    OLED_CS(1);
    OLED_RS(1);

    OLED_RST(0);
    delay_ms(100);
    OLED_RST(1);

    oled_wr_byte(0xAE, OLED_CMD); /* 关闭显示 */
    oled_wr_byte(0xD5, OLED_CMD); /* 设置时钟分频因子,震荡频率 */
    oled_wr_byte(80, OLED_CMD); /* [3:0],分频因子;[7:4],震荡频率 */
    oled_wr_byte(0xA8, OLED_CMD); /* 设置驱动路数 */
    oled_wr_byte(0X3F, OLED_CMD); /* 默认 0X3F(1/64) */
    oled_wr_byte(0xD3, OLED_CMD); /* 设置显示偏移 */
    oled_wr_byte(0X00, OLED_CMD); /* 默认为 0 */

    oled_wr_byte(0x40, OLED_CMD); /* 设置显示开始行 [5:0],行数. */

    oled_wr_byte(0x8D, OLED_CMD); /* 电荷泵设置 */
    oled_wr_byte(0x14, OLED_CMD); /* bit2, 开启/关闭 */
    oled_wr_byte(0x20, OLED_CMD); /* 设置内存地址模式 */
    /* [1:0],00,列地址模式;01,行地址模式;10,页地址模式;默认 10; */
    oled_wr_byte(0x02, OLED_CMD);
    oled_wr_byte(0xA1, OLED_CMD); /* 段重定义设置,bit0:0,0->0;1,0->127; */
}
```



```

/* 设置 COM 扫描方向;bit3:0,普通模式;1,重定义模式 COM[N-1]->COM0;N:驱动路数 */
oled_wr_byte(0xC8, OLED_CMD);
oled_wr_byte(0xDA, OLED_CMD); /* 设置 COM 硬件引脚配置 */
oled_wr_byte(0x12, OLED_CMD); /* [5:4]配置 */

oled_wr_byte(0x81, OLED_CMD); /* 对比度设置 */
oled_wr_byte(0xEF, OLED_CMD); /* 1~255;默认 0X7F (亮度设置,越大越亮) */
oled_wr_byte(0xD9, OLED_CMD); /* 设置预充电周期 */
oled_wr_byte(0xf1, OLED_CMD); /* [3:0],PHASE 1;[7:4],PHASE 2; */
oled_wr_byte(0xDB, OLED_CMD); /* 设置 VCOMH 电压倍率 */
/* [6:4]000,0.65*vcc;001,0.77*vcc;011,0.83*vcc; */
oled_wr_byte(0x30, OLED_CMD);
oled_wr_byte(0xA4, OLED_CMD); /* 全局显示开启;bit0:1,开启;0,关闭;(白屏/黑屏) */
oled_wr_byte(0xA6, OLED_CMD); /* 设置显示方式;bit0:1,反相显示;0,正常显示 */
oled_wr_byte(0xAF, OLED_CMD); /* 开启显示 */
oled_clear();
}

```

该函数的结构比较简单，开始是对 GPIO 口的初始化，这里我们用了宏定义 OLED_MODE 来决定要设置的 IO 口，后面的就是一些初始化序列了，我们按照厂家提供的资料来做就可以。值得注意一点的是，因为 OLED 是无背光的，在初始化之后，我们把显存都清空了，所以我们在屏幕上是看不到任何内容的，就像没通电一样，不要以为这就是初始化失败，要写入数据模块才会显示的。

接着，要介绍的是 oled_refresh_gram 更新显存到 OLED 函数，该函数的作用是把我们在程序中定义的二维数组 g_oled_gram 的值一次性刷新到 OLED 的显存 GRAM 中。我们在 oled.c 文件开头定义了如下一个二维数组：

```

/*
 * OLED 的显存
 * 每个字节表示 8 个像素，128,表示有 128 列，8 表示有 64 行，高位表示高行数。
 * 比如:g_oled_gram[0][0],包含了第一列,第 1~8 行的数据. g_oled_gram[0][0].0,即表示坐标
 * (0,0)
 * 类似的: g_oled_gram[1][0].1,表示坐标(1,1), g_oled_gram[10][1].2,表示坐标(10,10),
 * 存放格式如下(高位表示高行数).
 * [0]0 1 2 3 ... 127
 * [1]0 1 2 3 ... 127
 * [2]0 1 2 3 ... 127
 * [3]0 1 2 3 ... 127
 * [4]0 1 2 3 ... 127
 * [5]0 1 2 3 ... 127
 * [6]0 1 2 3 ... 127
 * [7]0 1 2 3 ... 127
 */
static uint8_t g_oled_gram[128][8];

```

该数组值与 OLED 显存 GRAM 值一一对应。在操作的时候我们只需要先修改该数组的值，然后再通过调用 oled_refresh_gram 函数把数组的值一次性刷新到 OLED 的 GRAM 上即可。oled_refresh_gram 函数定义如下：

```

/**
 * @brief      更新显存到 OLED
 * @param      无
 * @retval     无
 */
void oled_refresh_gram(void)
{
    uint8_t i, n;

    for (i = 0; i < 8; i++)
    {
        oled_wr_byte(0xb0 + i, OLED_CMD); /* 设置页地址(0~7) */
        oled_wr_byte(0x00, OLED_CMD); /* 设置显示位置-列低地址 */
        oled_wr_byte(0x10, OLED_CMD); /* 设置显示位置-列高地址 */
    }
}

```

```

        for (n = 0; n < 128; n++)
        {
            oled_wr_byte(g_oled_gram[n][i], OLED_DATA);
        }
    }
}

```

oled_refresh_gram 函数先设置页地址，然后写入列地址（也就是纵坐标），然后从 0 开始写入 128 个字节，写满该页，最后循环把 8 页的内容都写入，就实现了整个从 STM32 显存到 OLED 显存的拷贝。

oled_refresh_gram 函数还调用了 oled_wr_byte 这个函数，也就是我们接着要介绍的函数：该函数和硬件相关，8080 并口模式下该函数定义如下：

```

/**
 * @brief      向 OLED 写入一个字节
 * @param      data: 要输出的数据
 * @param      cmd: 数据/命令标志 0, 表示命令; 1, 表示数据;
 * @retval     无
 */
static void oled_wr_byte(uint8_t data, uint8_t cmd)
{
    oled_data_out(data);
    OLED_RS(cmd);
    OLED_CS(0);
    OLED_WR(0);
    OLED_WR(1);
    OLED_CS(1);
    OLED_RS(1);
}

```

8080 并口模式下的 oled_wr_byte 函数还调用 oled_data_out 函数，其定义如下：

```

/**
 * @brief      通过拼凑的方法向 OLED 输出一个 8 位数据
 * @param      data: 要输出的数据
 * @retval     无
 */
static void oled_data_out(uint8_t data)
{
    GPIOC->ODR = (GPIOC->ODR & 0XFF00) | (data & 0X00FF);
}

```

oled_data_out 函数的处理方法，就是我们前面说的，因为 OLED 的 D0~D7 对于战舰开发板来说正好顺序接到了 GPIOC[0:7]，所以我们只要向 GPIOC 的低位输出一数据即可实现并口数据输出。

如果我们使用 SPI 模式，则操作模块时会调用的函数接口按以下的软件 SPI 方式实现：

```

/**
 * @brief      向 OLED 写入一个字节
 * @param      data: 要输出的数据
 * @param      cmd: 数据/命令标志 0, 表示命令; 1, 表示数据;
 * @retval     无
 */
static void oled_wr_byte(uint8_t data, uint8_t cmd)
{
    uint8_t i;
    OLED_RS(cmd);    /* 写命令 */
    OLED_CS(0);

    for (i = 0; i < 8; i++)
    {
        OLED_SCLK(0);
        if (data & 0x80)
        {
            OLED_SDIN(1);
        }
        else

```

```

    {
        OLED_SDIN(0);
    }
    OLED_SCLK(1);
    data <<= 1;
}
OLED_CS(1);
OLED_RS(1);
}

```

两种模式下的 oled_wr_byte 函数形参是一样的。第一个形参 data 就是要写的内容。第二个形参 cmd 是通过选用 OLED_CMD 和 OLED_DATA 两个宏定义的其中一个，控制选择写命令还是写数据。两种模式下的 oled_wr_byte 函数的时序操作就是根据上面我们对 8080 接口以及 4 线 SPI 接口的时序来编写的。

g_oled_gram[128][8]二维数组中的 128 代表列数 (x 坐标)，而 8 代表的是页，每页又包含 8 行，总共 64 行 (y 坐标)，从高到低对应行数从小到大，如表 24.3.2.1 所示：

| 页数 | <div>x坐标 y坐标</div> | 0 | 1 | 2 | 3 | | 127 |
|-------|------------------------|------------|------------|------------|------------|-------|--------------|
| PAGE0 | 0 | G[0][0].b0 | G[1][0].b0 | G[2][0].b0 | G[3][0].b0 | | G[127][0].b0 |
| | 1 | G[0][0].b1 | G[1][0].b1 | G[2][0].b1 | G[3][0].b1 | | G[127][0].b1 |
| | 2 | G[0][0].b2 | G[1][0].b2 | G[2][0].b2 | G[3][0].b2 | | G[127][0].b2 |
| | 3 | G[0][0].b3 | G[1][0].b3 | G[2][0].b3 | G[3][0].b3 | | G[127][0].b3 |
| | 4 | G[0][0].b4 | G[1][0].b4 | G[2][0].b4 | G[3][0].b4 | | G[127][0].b4 |
| | 5 | G[0][0].b5 | G[1][0].b5 | G[2][0].b5 | G[3][0].b5 | | G[127][0].b5 |
| | 6 | G[0][0].b6 | G[1][0].b6 | G[2][0].b6 | G[3][0].b6 | | G[127][0].b6 |
| | 7 | G[0][0].b7 | G[1][0].b7 | G[2][0].b7 | G[3][0].b7 | | G[127][0].b7 |
| PAGE1 | 8 | G[0][1].b0 | G[1][1].b0 | G[2][1].b0 | G[3][1].b0 | | G[127][1].b0 |
| | 9 | G[0][1].b1 | G[1][1].b1 | G[2][1].b1 | G[3][1].b1 | | G[127][1].b1 |
| | | | | | | | |
| PAGE7 | 61 | G[0][7].b5 | G[1][7].b5 | G[2][7].b5 | G[3][7].b5 | | G[127][7].b5 |
| | 62 | G[0][7].b6 | G[1][7].b6 | G[2][7].b6 | G[3][7].b6 | | G[127][7].b6 |
| | 63 | G[0][7].b7 | G[1][7].b7 | G[2][7].b7 | G[3][7].b7 | | G[127][7].b7 |

表 24.3.2.1 OLED_GRAM 和 OLED 屏坐标对应关系

上表中 G 代表 OLED_GRAM，G[0][0]就表示 OLED_GRAM [0][0]。比如，我们要在 x=3，y=9 这个点写入 1，则可以用这个句子实现：

```
OLED_GRAM[3][1] |= 1<<1;
```

一个通用的在点 (x, y) 置 1 表达式为：

```
OLED_GRAM[x][y/8] |= 1<<(y%8);
```

其中 x 的范围为：0~127；y 的范围为：0~63。

因此，我们可以得出接下来介绍的这个比较重要的函数：OLED 画点函数，其定义如下：

```

/**
 * @brief      OLED 画点
 * @param      x   : 0~127
 * @param      y   : 0~63
 * @param      dot: 1 填充 0,清空
 * @retval     无
 */
void oled_draw_point(uint8_t x, uint8_t y, uint8_t dot)
{
    uint8_t pos, bx, temp = 0;

    if (x > 127 || y > 63) return; /* 超出范围了 */

    pos = y / 8; /* 计算 GRAM 里面的 y 坐标所在的字节，每个字节可以存储 8 个行坐标 */

    bx = y % 8; /* 取余数，方便计算 y 在对应字节里面的位置，及行 (y) 位置 */
    temp = 1 << bx; /* 高位表示高行号，得到 y 对应的 bit 位置，将该 bit 先置 1 */

```

```

if (dot)          /* 画实心点 */
{
    g_oled_gram[x][pos] |= temp;
}
else              /* 画空点,即不显示 */
{
    g_oled_gram[x][pos] &= ~temp;
}
}

```

该函数有 3 个形参，前两个是横纵坐标，第三个 t 为要写入 1 还是 0。该函数实现了我们在 OLED 模块上任意位置画点的功能。

前面我们知道取模方式是：从上到下，从左到右，高位在前。下面根据取模的方式来编写显示字符 oled_show_char 函数，其定义如下：

```

/**
 * @brief      在指定位置显示一个字符,包括部分字符
 * @param      x    : 0~127
 * @param      y    : 0~63
 * @param      size: 选择字体 12/16/24
 * @param      mode: 0,反白显示;1,正常显示
 * @retval     无
 */
void oled_show_char(uint8_t x,uint8_t y,uint8_t chr,uint8_t size,uint8_t mode)
{
    uint8_t temp, t, t1;
    uint8_t y0 = y;
    uint8_t *pfont = 0;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    uint8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size / 2);
    chr = chr - ' '; /* 得到偏移后的值,因为字库是从空格开始存储的,第一个字符是空格 */

    if (size == 12) /* 调用 1206 字体 */
    {
        pfont = (uint8_t *)oled_asc2_1206[chr];
    }
    else if (size == 16) /* 调用 1608 字体 */
    {
        pfont = (uint8_t *)oled_asc2_1608[chr];
    }
    else if (size == 24) /* 调用 2412 字体 */
    {
        pfont = (uint8_t *)oled_asc2_2412[chr];
    }
    else /* 没有的字库 */
    {
        return;
    }

    for (t = 0; t < csize; t++)
    {
        temp = pfont[t];

        for (t1 = 0; t1 < 8; t1++)
        {
            if (temp & 0x80)oled_draw_point(x, y, mode);
            else oled_draw_point(x, y, !mode);

            temp <<= 1;
            y++;

            if ((y - y0) == size)
            {
                y = y0;
                x++;
            }
        }
    }
}

```

```

        break;
    }
}
}
}

```

该函数为字符以及字符串显示的核心部分，函数中 `chr = chr - ' '`；这句是要得到在字符点阵数据里面的实际地址，因为我们的取模是从空格键开始的，例如 `oled_asc2_1206[0][0]`，代表的是空格符开始的点阵码。在接下来的代码，我们也是按照从上到下（先 `y++`），从左到右（再 `x++`）的取模方式来编写的，先得到最高位，然后判断是写 1 还是 0，画点；接着读第二位，如此循环，直到一个字符的点阵全部取完为止。这其中涉及到列地址和行地址的自增，根据取模方式来理解，就不难了。

`oled.c` 的内容比较多，其他的函数请大家自行理解，下面开始 `main.c` 文件的介绍。

2. main.c 代码

在 `main.c` 里面编写如下代码：

```

int main(void)
{
    uint8_t t = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    oled_init(); /* 初始化 OLED */
    oled_show_string(0, 0, "ALIENTEK", 24);
    oled_show_string(0, 24, "0.96' OLED TEST", 16);
    oled_show_string(0, 52, "ASCII:", 12);
    oled_show_string(64, 52, "CODE:", 12);
    oled_refresh_gram(); /* 更新显示到 OLED */

    t = ' ';
    while (1)
    {
        oled_show_char(36, 52, t, 12, 1); /* 显示 ASCII 字符 */
        oled_show_num(94, 52, t, 3, 12); /* 显示 ASCII 字符的码值 */
        oled_refresh_gram(); /* 更新显示到 OLED */
        t++;

        if (t > '~')
        {
            t = ' ';
        }

        delay_ms(500);
        LED0_TOGGLE(); /* LED0 闪烁 */
    }
}

```

`main.c` 主要功能就是在 OLED 上显示一些实验信息字符，然后开始从空格键开始不停的循环显示 ASCII 字符集，并显示该字符的 ASCII 值。最后 LED0 闪烁提示程序正在运行。

24.4 下载验证

下载代码后，LED0 不停的闪烁，提示程序已经在运行了。同时 OLED 模块显示 ASCII 字符集等信息，如图 24.4.1 所示：



图 24.4.1 OLED 显示效果

OLED 显示了三种尺寸的字符：24*12（ALIENTEK）、16*8（0.96' OLED TEST）和 12*6（剩下的内容）。说明我们的实验是成功的，实现了三种不同尺寸 ASCII 字符的显示，在最后一行不停的显示 ASCII 字符以及其码值。

通过这一章的学习，我们学会了正点原子 OLED 模块的使用，在调试代码的时候，又多了一种显示信息的途径，在以后的程序编写中，大家可以好好利用。

第二十五章 TFT-LCD (MCU 屏) 实验

前面我们介绍了 OLED 模块及其显示,但是该模块只能显示单色/双色,不能显示彩色,而且尺寸也较小。本章我们将介绍正点原子的 TFT-LCD 模块 (MCU 屏),该模块采用 TFT-LCD 面板,可以显示 16 位色的真彩图片。在本章中,我们将使用开发板底板上的 TFTLCD 接口 (仅支持 MCU 屏,本章仅介绍 MCU 屏的使用),来点亮 TFT-LCD,并实现 ASCII 字符和彩色的显示等功能,并在串口打印 LCD 控制器 ID,同时在 LCD 上面显示。

本章分为如下几个小节:

25.1 TFTLCD 和 FSMC 简介

25.2 硬件设计

25.3 程序设计

25.4 下载验证

25.1 TFT-LCD 简介

本章我们将通过 STM32F103 的 FSMC 外设来控制 TFT-LCD 的显示,这样我们就可以用 STM32 输出一些信息到显示屏上了。

25.1.1 TFT-LCD 简介

液晶显示器,即 Liquid Crystal Display,利用了液晶导电后透光性可变的特性,配合显示器光源、彩色滤光片和电压控制等工艺,最终可以在液晶阵列上显示彩色的图像。目前液晶显示技术以 TN、STN、TFT 三种技术为主,TFT-LCD 即采用了 TFT (Thin Film Transistor) 技术的液晶显示器,也叫薄膜晶体管液晶显示器。

TFT-LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同的是,它在液晶显示屏的每一个像素上都设置有一个薄膜晶体管 (TFT),可有效地克服非选通时的串扰,使显示液晶屏的静态特性与扫描线数无关,因此大大提高了图像质量。TFT 式显示器具有很多优点:高响应度,高亮度,高对比度等等。TFT 式屏幕的显示效果非常出色,广泛应用于手机屏幕、笔记本电脑和台式机显示器上。

由于液晶本身不会发光,加上液晶本身的特性等原因,使得液晶屏的成像角受限,我们从屏幕的一侧可能无法看清液晶的显示内容。液晶显示器的成像角的大小也是评估一个液晶显示器优劣的指标,目前,规格较好的液晶显示器成像角一般在 $120^{\circ}\sim 160^{\circ}$ 之间。

正点原子 TFT-LCD 模块(MCU 屏)有如下特点:

- 1, 2.8' /3.5' /4.3' /7' 等 4 种大小的屏幕可选。
- 2, 320×240 的分辨率 (3.5' 分辨率为: 320*480, 4.3' 和 7' 分辨率为: 800*480)。
- 3, 16 位真彩显示。
- 4, 自带触摸屏,可以用来作为控制输入。

本章,我们以正点原子 2.8 寸 (此处的寸是代表英寸,下同) 的 TFT-LCD 模块为例介绍, (其他尺寸的 LCD 可参考具体的 LCD 型号的资料,也比较类似),该模块支持 65K 色显示,显示分辨率为 320×240,接口为 16 位的 8080 并口,自带触摸功能。

该模块的外观如图 25.1.1.1 所示:



图 25.1.1.1 正点原子 2.8 寸 TFTLCD 外观图

模块原理图如图 25.1.1.2 所示：

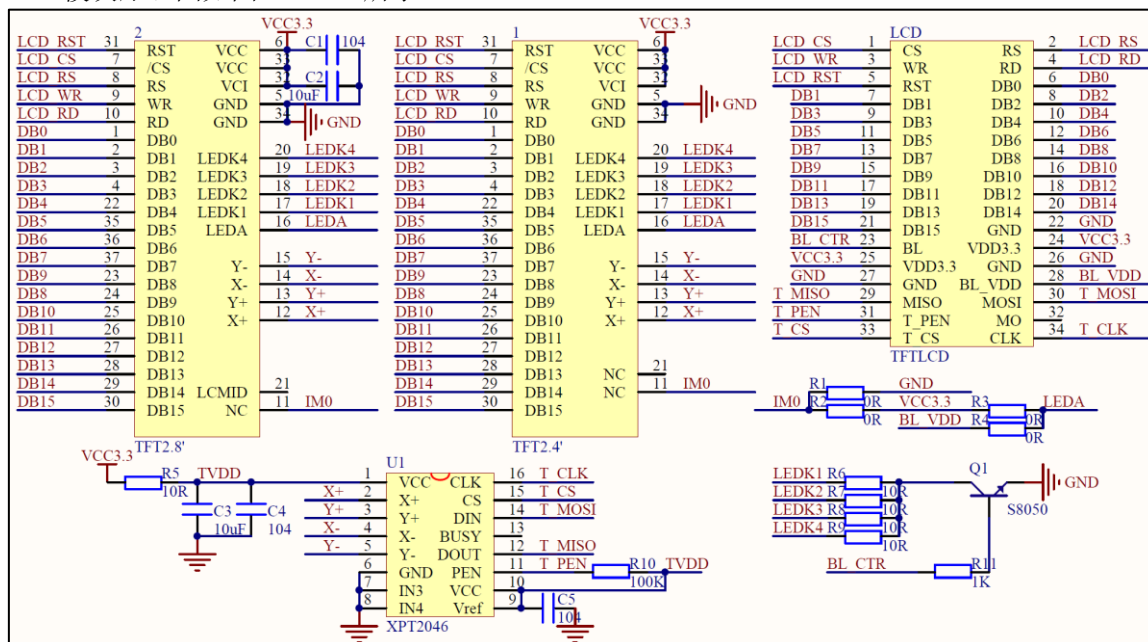


图 25.1.1.2 正点原子 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，即图中 TFT_LCD 部分。从图 25.1.1.2 可以看出，正点原子 TFTLCD 模块采用 16 位的并方式与外部连接。图 25.1.1.2 还列出了触摸控制的接口，但触摸控制是在显示的基础上叠加的一个控制功能，不配置也不会对显示造成影响，我们放到以后的章节再介绍触摸的用法。该模块与显示功能的相关的信号线如表 25.1.1.1：

| 名称 | 功能 |
|----------|-------------------------|
| CS | TFT-LCD 片选信号。 |
| WR | 向 TFT-LCD 写入数据。 |
| RD | 从 TFT-LCD 读取数据。 |
| DB[15:0] | 16 位双向数据线。 |
| RST | 硬复位 TFT-LCD。 |
| RS | 命令/数据标志（0，读写命令；1，读写数据）。 |
| BL | 背光控制。 |

表 25.1.1.1 TFT-LCD 接口信号线

上述的接口线实际是对应到液晶显示控制器上的，这个芯片位于液晶屏的下方，所以我们从外观图上看不到。控制 LCD 显示的过程，就是按其显示驱动芯片的时序，把色彩和位置信息正确地写入对应的寄存器。

25.1.2 液晶显示控制器

正点原子提供 2.8/3.5/4.3/7 寸等 4 种不同尺寸和分辨率的 TFTLCD 模块，其驱动芯片为：ILI9341/ST7789/NT35310/NT35510/SSD1963 等(具体的型号,大家可以通过下载本章实验代码,通过串口或者 LCD 显示查看),这里我们仅以 ILI9341 控制器为例进行介绍,其他的控制基本都类似,我们就不详细阐述了。

ILI9341 液晶控制器自带显存,可配置支持 8/9/16/18 位的总线中的一种,可以通过 3/4 线串行协议或 8080 并口驱动。正点原子的 TFT-LCD 模块上的电路配置为 8080 并口方式,其显存总大小为 172800 (240*320*18/8),即 18 位模式 (26 万色) 下的显存量。在 16 位模式下,ILI9341 采用 RGB565 格式存储颜色数据,此时 ILI9341 的 18 位显存与 MCU 的 16 位数据线以及 RGB565 的对应关系如图 25.1.2.1 所示:

| 9341 显存 | B17 | B16 | B15 | B14 | B13 | B12 | B11 | B10 | B9 | B8 | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|-------------|------|------|------|------|------|-----|------|------|------|------|------|------|------|------|------|------|------|----|
| RGB565 GRAM | R[4] | R[3] | R[2] | R[1] | R[0] | NC | G[5] | G[4] | G[3] | G[2] | G[1] | G[0] | B[4] | B[3] | B[2] | B[1] | B[0] | NC |
| 9341 数据线 | DB15 | DB14 | DB13 | DB12 | DB11 | | DB10 | DB9 | DB8 | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | |
| MCU 数据线 | D15 | D14 | D13 | D12 | D11 | | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |

图 25.1.2.1 16 位数据与显存对应关系图

从图中可以看出,ILI9341 在 16 位模式下面,18 位显存的 B0 和 B12 并没有用到,对外的数据线使用 DB0-DB15 连接 MCU 的 D0-D15 实现 16 位颜色的传输 (使用 8080 MCU 16bit I 型接口,详见 9341 数据手册 7.1.1 节)。

这样 MCU 的 16 位数据,最低 5 位代表蓝色,中间 6 位为绿色,最高 5 位为红色。数值越大,表示该颜色越深。另外,特别注意 ILI9341 所有的指令都是 8 位的 (高 8 位无效),且参数除了读写 GRAM 的时候是 16 位,其他操作参数,都是 8 位的。

知道了屏幕的显色信息后,我们如何驱动它呢? OLED 的章节我们已经描述过 8080 方式操作的时序,我们通过《ILI9341_DS.pdf》来加深一下在 8080 并口方式下如何操作这个芯片。

以写周期为例,8080 方式下的操作时序如图 25.1.2.2 所示。

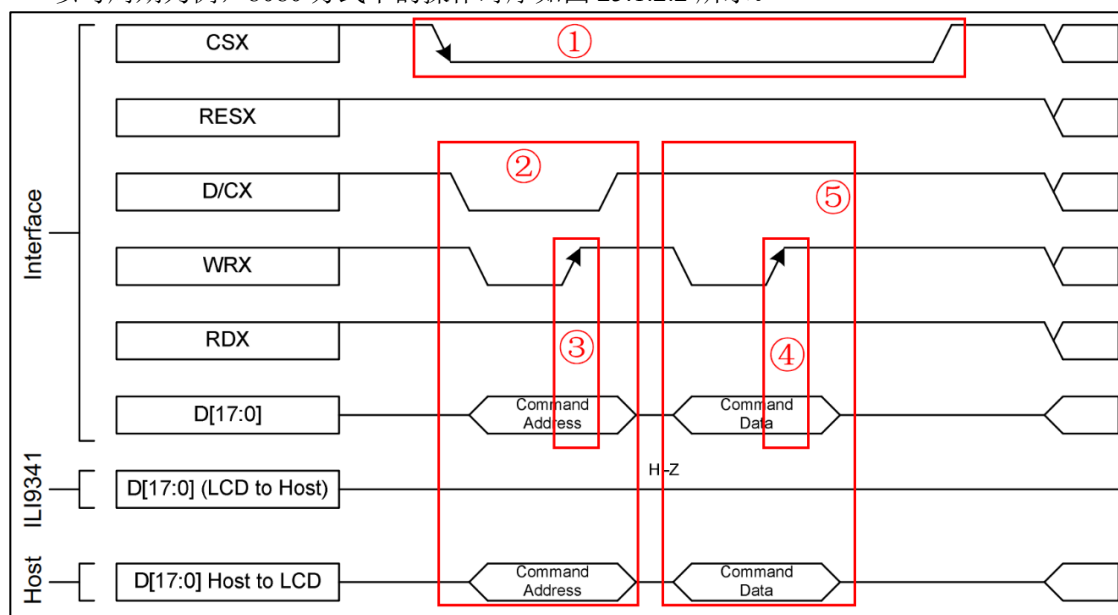


图 25.1.2.2 8080 方式下对液晶控制器的写操作

上图中的各个控制线与我们在表 25.1.1.1 提到的命名有些许差异,因为我们在原理图时往往为了方便自己记忆会对命名进行微调,为了方便读者对照,我们把图 25.1.2.2 中列出的引脚与我们的 TFTLCD 模块的对应关系再列出,如表 25.1.2.1 所示。

| TFTLCD 名称 | ILI9341 | 功能 |
|-----------|---------|-----------------|
| CS | CSX | TFT-LCD 片选信号。 |
| WR | WRX | 向 TFT-LCD 写入数据。 |
| RD | RDX | 从 TFT-LCD 读取数据。 |
| D[15:0] | D[15:0] | 16 位双向数据线。 |

| | | |
|----|------|-------------------------|
| RS | D/CX | 命令/数据标志（0，读写命令；1，读写数据）。 |
|----|------|-------------------------|

表 25.1.2.1 TFT-LCD 引脚与液晶控制器的对应关系

这下我们再分析一下图 25.1.2.2 所示的写操作的时序，控制液晶的主机，在整个写周期内需要控制片选 CSX 拉低（标注为①），之后对其它的控制线的电平才有效。在标号②表示的这个写命令周期中，D/CX 被位低（参考 ILI9341 的引脚定义），同时把命令码通过数据线 D[17:0]（我们实际只用了 16 个引脚）按位编码。注意到③处，需要数据线在入电平拉高后再操持一段时间以便数据被正确采样。

图 25.1.2.2 中⑤表示写数据操作，与前面描述的写命令操作只有 D/CX 的操作不同，读者们可以尝试自己分析一下。更多的关于 ILI9341 的读写操作时序则参考《ILI9341_DS.pdf》。

通过前述的时序分析，我们知道了对于 ILI9341 来说，控制命令有命令码、数据码之分，接下来，我们介绍一下 ILI9341 的几个重要命令。因为 ILI9341 的命令很多，我们这里就不全部介绍了，有兴趣的大家可以找到 ILI9341 的 datasheet 看看。里面对这些命令有详细的介绍。我们将介绍：0xD3，0x36，0x2A，0x2B，0x2C，0x2E 等 6 条指令。

指令 0xD3，是读 ID4 指令，用于读取 LCD 控制器的 ID，该指令如表 25.1.2.2 所示：

| 顺序 | 控制 | | | 各位描述 | | | | | | | | | HEX |
|------|----|----|----|--------|----|----|----|----|----|----|----|----|-----|
| | RS | RD | WR | D15~D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 指令 | 0 | 1 | ↑ | XX | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | D3H |
| 参数 1 | 1 | ↑ | 1 | XX | X | X | X | X | X | X | X | X | X |
| 参数 2 | 1 | ↑ | 1 | XX | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00H |
| 参数 3 | 1 | ↑ | 1 | XX | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 93H |
| 参数 4 | 1 | ↑ | 1 | XX | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 41H |

表 25.1.2.2 0xD3 指令描述

从上表可以看出，0xD3 指令后面跟了 4 个参数，最后 2 个参数，读出来是 0x93 和 0x41，刚好是我们控制器 ILI9341 的数字部分，从而，通过该指令，即可判别所用的 LCD 驱动器是什么型号，这样，我们的代码，就可以根据控制器的型号去执行对应驱动 IC 的初始化代码，从而兼容不同驱动 IC 的屏，使得一个代码支持多款 LCD。

接下来看指令：0x36，这是存储访问控制指令，可以控制 ILI9341 存储器的读写方向，简单的说，就是在连续写 GRAM 的时候，可以控制 GRAM 指针的增长方向，从而控制显示方式（读 GRAM 也是一样）。该指令如表 25.1.2.3 所示：

| 顺序 | 控制 | | | 各位描述 | | | | | | | | | HEX |
|----|----|----|----|--------|----|----|----|----|-----|----|----|----|-----|
| | RS | RD | WR | D15~D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 指令 | 0 | 1 | ↑ | XX | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 36H |
| 参数 | 1 | 1 | ↑ | XX | MY | MX | MV | ML | BGR | MH | 0 | 0 | 0 |

表 25.1.2.3 0x36 指令描述

从上表可以看出，0x36 指令后面，紧跟一个参数，这里主要关注：MY、MX、MV 这三个位，通过这三个位的设置，我们可以控制整个 ILI9341 的全部扫描方向，如表 25.1.2.4 所示：

| 控制位 | | | 效果 |
|-----|----|----|-----------|
| MY | MX | MV | |
| 0 | 0 | 0 | 从左到右,从上到下 |
| 1 | 0 | 0 | 从左到右,从下到上 |
| 0 | 1 | 0 | 从右到左,从上到下 |
| 1 | 1 | 0 | 从右到左,从下到上 |
| 0 | 0 | 1 | 从上到下,从左到右 |
| 0 | 1 | 1 | 从上到下,从右到左 |
| 1 | 0 | 1 | 从下到上,从左到右 |
| 1 | 1 | 1 | 从下到上,从右到左 |

表 25.1.2.4 MY、MX、MV 设置与 LCD 扫描方向关系表

这样，我们在利用 ILI9341 显示内容的时候，就有很大的灵活性了，比如显示 BMP 图片，BMP 解码数据，就是从图片的左下角开始，慢慢显示到右上角，如果设置 LCD 扫描方向为从

左到右，从下到上，那么我们只需要设置一次坐标，然后就不停的往 LCD 填充颜色数据即可，这样可以大大提高显示速度。

实验中，我们默认使用从左到右，从上到下的扫描方式。

接下来看指令：0x2A，这是列地址设置指令，在从左到右，从上到下的扫描方式（默认）下面，该指令用于设置横坐标（x 坐标），该指令如表 25.1.2.5 所示：

| 顺序 | 控制 | | | 各位描述 | | | | | | | | | HEX |
|------|----|----|----|--------|------|------|------|------|------|------|-----|-----|-----|
| | RS | RD | WR | D15~D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 指令 | 0 | 1 | ↑ | XX | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 2AH |
| 参数 1 | 1 | 1 | ↑ | XX | SC15 | SC14 | SC13 | SC12 | SC11 | SC10 | SC9 | SC8 | SC |
| 参数 2 | 1 | 1 | ↑ | XX | SC7 | SC6 | SC5 | SC4 | SC3 | SC2 | SC1 | SC0 | |
| 参数 3 | 1 | 1 | ↑ | XX | EC15 | EC14 | EC13 | EC12 | EC11 | EC10 | EC9 | EC8 | EC |
| 参数 4 | 1 | 1 | ↑ | XX | EC7 | EC6 | EC5 | EC4 | EC3 | EC2 | EC1 | EC0 | |

表 25.1.2.5 0x2A 指令描述

在默认扫描方式时，该指令用于设置 x 坐标，该指令带有 4 个参数，实际上是 2 个坐标值：SC 和 EC，即列地址的起始值和结束值，SC 必须小于等于 EC，且 $0 \leq SC/EC \leq 239$ 。一般在设置 x 坐标的时候，我们只需要带 2 个参数即可，也就是设置 SC 即可，因为如果 EC 没有变化，我们只需要设置一次即可（在初始化 ILI9341 的时候设置），从而提高速度。

与 0x2A 指令类似，指令：0x2B，是页地址设置指令，在从左到右，从上到下的扫描方式（默认）下面，该指令用于设置纵坐标（y 坐标）。该指令如表 25.1.2.6 所示：

| 顺序 | 控制 | | | 各位描述 | | | | | | | | | HEX |
|------|----|----|----|--------|------|------|------|------|------|------|-----|-----|-----|
| | RS | RD | WR | D15~D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 指令 | 0 | 1 | ↑ | XX | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 2BH |
| 参数 1 | 1 | 1 | ↑ | XX | SP15 | SP14 | SP13 | SP12 | SP11 | SP10 | SP9 | SP8 | SP |
| 参数 2 | 1 | 1 | ↑ | XX | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | |
| 参数 3 | 1 | 1 | ↑ | XX | EP15 | EP14 | EP13 | EP12 | EP11 | EP10 | EP9 | EP8 | EP |
| 参数 4 | 1 | 1 | ↑ | XX | EP7 | EP6 | EP5 | EP4 | EP3 | EP2 | EP1 | EP0 | |

表 25.1.2.6 0x2B 指令描述

在默认扫描方式时，该指令用于设置 y 坐标，该指令带有 4 个参数，实际上是 2 个坐标值：SP 和 EP，即页地址的起始值和结束值，SP 必须小于等于 EP，且 $0 \leq SP/EP \leq 319$ 。一般在设置 y 坐标的时候，我们只需要带 2 个参数即可，也就是设置 SP 即可，因为如果 EP 没有变化，我们只需要设置一次即可（在初始化 ILI9341 的时候设置），从而提高速度。

接下来看指令：0x2C，该指令是写 GRAM 指令，在发送该指令之后，我们便可以往 LCD 的 GRAM 里面写入颜色数据了，该指令支持连续写，指令描述如表 25.1.2.7 所示。

| 顺序 | 控制 | | | 各位描述 | | | | | | | | | HEX |
|-------|----|----|----|-----------|----|----|----|----|----|----|----|----|-----|
| | RS | RD | WR | D15~D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 指令 | 0 | 1 | ↑ | XX | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 2CH |
| 参数 1 | 1 | 1 | ↑ | D1[15: 0] | | | | | | | | | XX |
| | 1 | 1 | ↑ | D2[15: 0] | | | | | | | | | XX |
| 参数 n | 1 | 1 | ↑ | Dn[15: 0] | | | | | | | | | XX |

表 25.1.2.7 0x2C 指令描述

由表 25.1.2.7 可知，在收到指令 0x2C 之后，数据有效位宽变为 16 位，我们可以连续写入 LCD GRAM 值，而 GRAM 的地址将根据 MY/MX/MV 设置的扫描方向进行自增。例如：假设设置的是从左到右，从上到下的扫描方式，那么设置好起始坐标（通过 SC，SP 设置）后，每写入一个颜色值，GRAM 地址将会自动自增 1（SC++），如果碰到 EC，则回到 SC，同时 SP++，一直到坐标：EC，EP 结束，期间无需再次设置的坐标，从而大大提高写入速度。

最后，来看看指令：0x2E，该指令是读 GRAM 指令，用于读取 ILI9341 的显存（GRAM），该指令在 ILI9341 的数据手册上面的描述是有误的，真实的输出情况如表 25.1.2.8 所示：

| 顺序 | 控制 | | | 各位描述 D[15:0] | | | | | | | | | HEX |
|----|----|----|----|--------------|------|---|---|---|---|---|---|---|-----|
| | RS | RD | WR | 15~11 | 10~8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | | | | | | | | | | | | | |
|------|---|---|---|---------|----|---------|---------|---|---|---|----|------|------|-------|
| 指令 | 0 | 1 | ↑ | XX | | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 2EH |
| 参数 1 | 1 | ↑ | 1 | XX | | | | | | | | | | dummy |
| 参数 2 | 1 | ↑ | 1 | R1[4:0] | XX | G1[5:0] | | | | | | XX | R1G1 | |
| 参数 3 | 1 | ↑ | 1 | B1[4:0] | XX | R2[4:0] | | | | | XX | B1R2 | | |
| 参数 4 | 1 | ↑ | 1 | G2[5:0] | | XX | B2[4:0] | | | | | XX | G2B2 | |
| 参数 5 | 1 | ↑ | 1 | R3[4:0] | XX | G3[5:0] | | | | | | XX | R3G3 | |
| 参数 N | 1 | ↑ | 1 | 按以上规律输出 | | | | | | | | | | |

表 25.1.2.8 0X2E 指令描述

该指令用于读取 GRAM，如表 25.1.2.7 所示，ILI9341 在收到该指令后，第一次输出的是 dummy 数据，也就是无效的数据，第二次开始，读取到的才是有效的 GRAM 数据（从坐标：SC，SP 开始），输出规律为：每个颜色分量占 8 个位，一次输出 2 个颜色分量。比如：第一次输出是 R1G1，随后的规律为：B1R2→G2B2→R3G3→B3R4→G4B4→R5G5...以此类推。如果我们只需要读取一个点的颜色值，那么只需要接收到参数 3 即可，如果要连续读取（利用 GRAM 地址自增，方法同上），那么就按照上述规律去接收颜色数据。

以上，就是操作 ILI9341 常用的几个指令，通过这几个指令，我们便可以很好的控制 ILI9341 显示我们所要显示的内容了。

25.1.3 FSMC 简介

ILI9341 的 8080 通讯接口时序可以由 STM32 使用 GPIO 接口进行模拟，但这样效率太低，STM32 提供了一种更高效的控制方法——使用 FSMC 接口实现 8080 时序，但 FSMC 是 STM32 片上外设的一种，并非所有的 STM32 都拥有这种硬件接口，使用何种方式驱动需要在芯片选型时就确定好。我们的开发板支持 FSMC 接口，下面我们来了解一下这个接口的功能。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接，FSMC 接口可以通过地址信号，快速找到存储器对应存储块上的数据。STM32F1 的 FSMC 接口支持包括 SRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。F1 系列的大容量型号，且引脚数目在 100 脚及以上的 STM32F103 芯片都带有 FSMC 接口，正点原子战舰 STM32F103 的主芯片为 STM32F103ZET6，是带有 FSMC 接口的。

FSMC 接口的结构如图 25.1.3.1 所示：

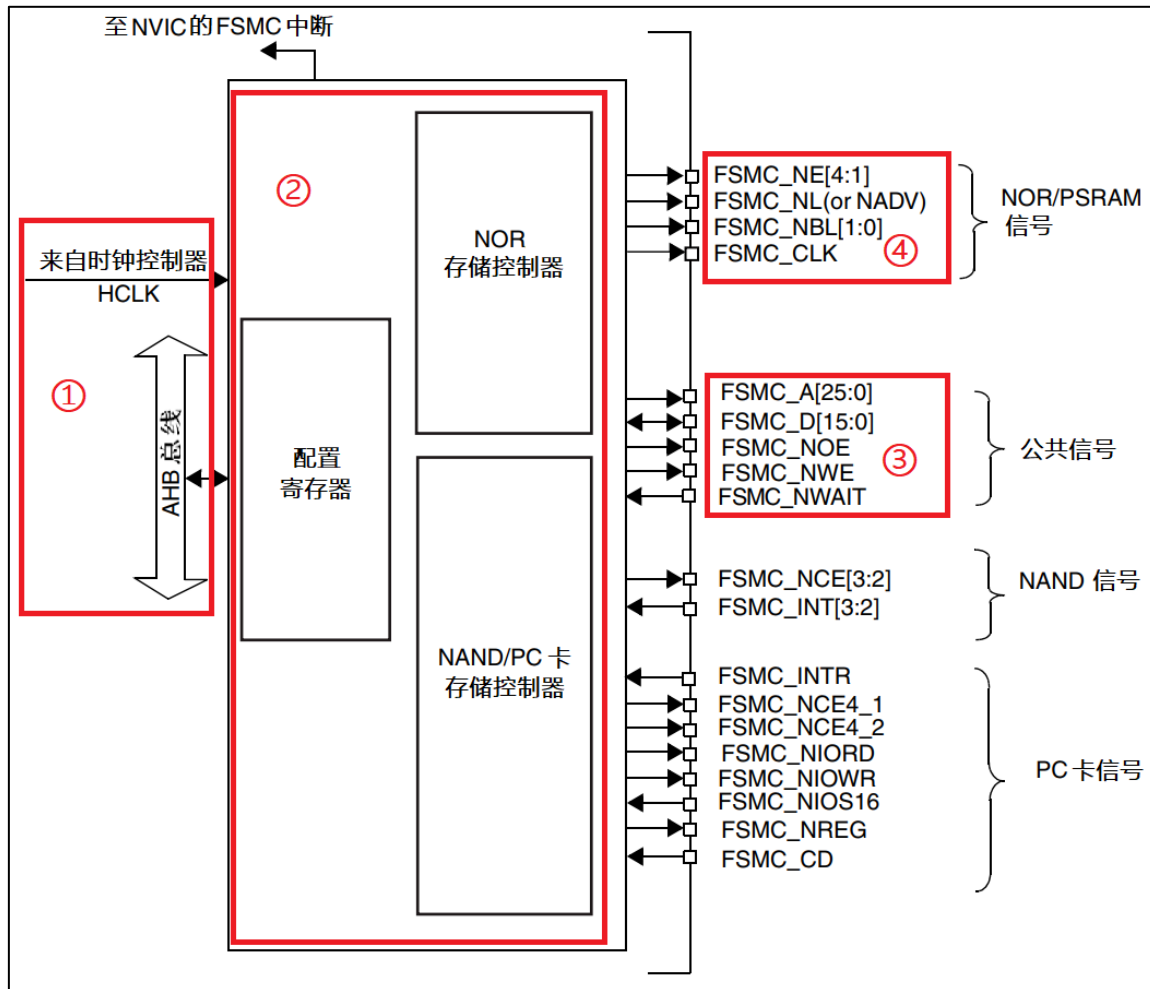


图 25.1.3.1 FSMC 框图

从图 25.1.3.1 我们可以看出，STM32 的 FSMC 可以驱动 NOR/PSRAM、NAND、PC 卡这 3 类设备，他们具有不同的 CS 以区分不同的设备。本部分我们要用到的是 NOR/PSRAM 的功能。①为 FSMC 的总线和时钟源，②为 STM32 内部的 FSMC 控制单元，③是连接硬件的引脚，这里的“公共信号”表示不论我们驱动前面提到的 3 种设备中的哪种，这些 IO 是共享的，所以如果需要用到多种功能的情况，程序上还要考虑分时复用。④是 NOR/PSRAM 会使用到的信号控制线，③和④这些信号比较重要，它们的功能如表 25.1.3.1：

| FSMC 信号名称 | 信号方向 | 功能 |
|--------------|-------|--------------------------------------|
| FSMC_NE[x] | 输出 | STM32F1 有四个片选引脚，x = 1...4，每个对应不同的内存块 |
| FSMC_CLK | 输出 | 时钟(同步突发模式使用) |
| FSMC_A[25:0] | 输出 | 地址总线 |
| FSMC_D[15:0] | 输入/输出 | 双向数据总线 |
| FSMC_NOE | 输出 | 输出使能 |
| FSMC_NWE | 输出 | 写使能 |
| FSMC_NWAIT | 输入 | NOR 闪存要求 FSMC 等待的信号 |
| FSMC_NADV | 输出 | 地址、数据线复用时作锁存信号 |

表 25.1.3.1 FSMC 信号线的功能

在数电的课程中有介绍过存储器的知识，它是可以存储数据的器件。复杂的存储器为了存储更多的数据，常常通过地址线来管理数据存储的位置，这样只要先找到需要读写数据的位置，然后对数据进行读写的操作。由于存储器的这种数据和地址对应关系，采用 FSMC 这种专门硬件接口就能加快对存储器的数据访问。

STM32F1 的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，FSMC 的外部设备地址映像如图 25.1.3.2 所示：

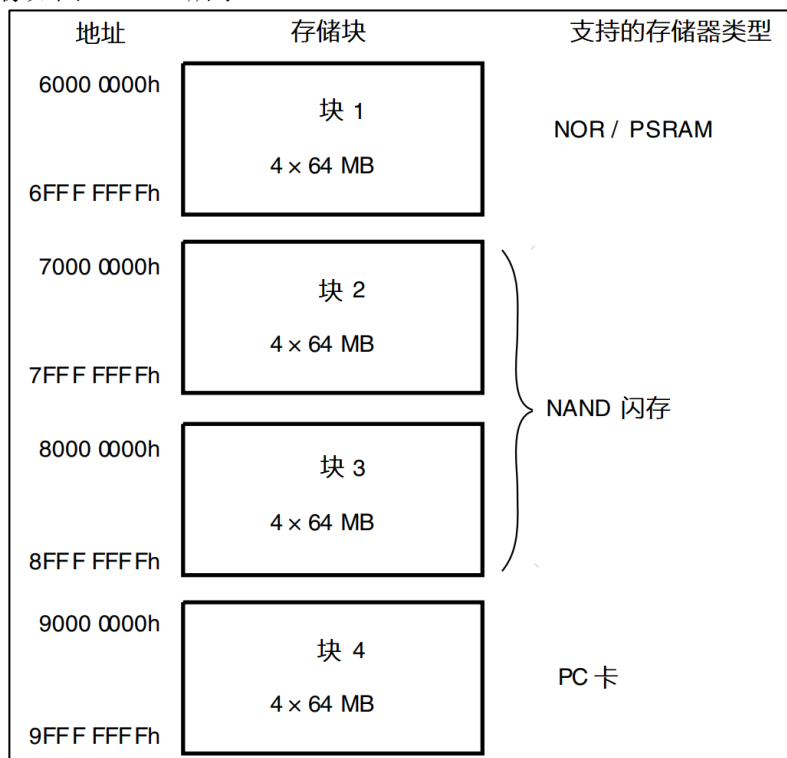


图25.1.3.2 FSMC存储块地址映像

从上图可以看出，FSMC 总共管理 1GB 空间，拥有 4 个存储块（Bank），FSMC 各 Bank 配置寄存器如表 25.1.3.2：

| 内部控制器 | 存储块 | 管理的地址范围 | 支持的设备类型 | 配置寄存器 |
|---------------------------|-------|-------------------------|------------|---|
| NOR FLASH 控制器 | Bank1 | 0X6000,0000~0X6FFF,FFFF | SRAM/ROM | FSMC_BCR1/2/3/4 |
| | | | NOR FLASH | FSMC_BTR1/2/2/3 |
| | | | PSRAM | FSMC_BWTR1/2/3/4 |
| NAND FLASH PC CARD 控制器 | Bank2 | 0X7000,0000~0X7FFF,FFFF | NAND FLASH | FSMC_PCR2/3/4 |
| | Bank3 | 0X8000,0000~0X8FFF,FFFF | | FSMC_SR2/3/4 |
| | Bank4 | 0X9000,0000~0X9FFF,FFFF | PC Card | FSMC_PMEM2/3/4 FSMC_PATT2/3/4 FSMC_PIO4 |

表25.1.3.2 FSMC各Bank配置寄存器表

本章，我们用到的是块 1，所以在本章我们仅讨论块 1 的相关配置，其他块的配置，请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》第 19 章（324 页）的相关介绍。

STM32F1 的 FSMC 存储块 1（Bank1）被分为 4 个区，每个区管理 64M 字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1 的 256M 字节空间可以通过 28 根地址线（HADDR[27:0]）寻址后访问。这里 HADDR 是内部 AHB 地址总线，其中 HADDR[25:0]来自外部存储器地址 FSMC_A[25:0]，而 HADDR[26:27]对 4 个区进行寻址。如表 25.1.3.3 所示：

| Bank1 所选区 | 片选信号 | 地址范围 | HADDR | |
|-----------|----------|-----------------------|---------|--------------|
| | | | [27:26] | [25:0] |
| 第 1 区 | FSMC_NE1 | 0X6000,0000~63FF,FFFF | 00 | FSMC_A[25:0] |
| 第 2 区 | FSMC_NE2 | 0X6400,0000~67FF,FFFF | 01 | |
| 第 3 区 | FSMC_NE3 | 0X6800,0000~6BFF,FFFF | 10 | |
| 第 4 区 | FSMC_NE4 | 0X6C00,0000~6FFF,FFFF | 11 | |

表 25.1.3.3 Bank1 存储区选择表

表 25.1.3.3 中，我们要特别注意 HADDR[25:0]的对应关系：

当 Bank1 接的是 16 位宽度存储器的时候：HADDR[25:1]→FSMC_A[24:0]。

当 Bank1 接的是 8 位宽度存储器的时候：HADDR[25:0] →FSMC_A[25:0]。

不论外部接 8 位/16 位宽设备，FSMC_A[0]永远接在外部设备地址 A[0]。这里，TFTLCD 使用的是 16 位数据宽度，所以 HADDR[0]并没有用到，只有 HADDR[25:1]是有效的，对应关系变为：HADDR[25:1]→FSMC_A[24:0]，相当于右移了一位，具体来说，比如地址：0x7E，对应二进制是：01111110，此时 FSMC_A6 是 0 而不是 1，因为要右移一位，这里请特别注意。

另外，HADDR[27:26]的设置，是不需要我们干预的，比如：当你选择使用 Bank1 的第三个区，即使用 FSMC_NE3 来连接外部设备的时候，即对应了 HADDR[27:26]=10，我们要做的就是配置对应第 3 区的寄存器组，来适应外部设备即可。对于 NOR FLASH 控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1~4，对应 4 个区）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数，拓宽了可选用的外部存储器的速度范围。FSMC 的 NORFLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

1，HCLK与FSMC_CLK的分频系数(CLKDIV)，可以为2~16分频；

2，同步突发访问中获得第1个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 NOR Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如表 25.1.3.4 所列：

| 时序模型 | | 简单描述 | 时间参数 |
|------|---------|---------------------------|----------------------|
| 异步 | Model1 | SRAM/CRAM时序 | DATAST、ADDSET |
| | ModeA | SRAM/CRAM OE选通型时序 | DATAST、ADDSET |
| | Mode2/B | NOR FLASH时序 | DATAST、ADDSET |
| | ModeC | NOR FLASH OE选通型时序 | DATAST、ADDSET |
| | ModeD | 延长地址保持时间的异步时序 | DATAST、ADDSET、ADDHLK |
| 同步突发 | | 根据同步时钟FSMC_CLK读取多个顺序单元的数据 | CLKDIV、DATLAT |

表25.1.3.4 NOR FLASH控制器支持的时序模型

在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读/写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

模式A支持独立的读写时序控制。这个对我们驱动TFTLCD来说非常有用，因为TFTLCD在读的时候，一般比较慢，而在写的时候可以比较快，如果读写用一样的时序，那么只能以读的时序为基准，从而导致写的速度变慢，或者在读数据的时候，重新配置FSMC的延时，在读操作完成的时候，再配置回写的时序，这样虽然也不会降低写的速度，但是频繁配置，比较麻烦。而如果有独立的读写时序控制，那么我们只要初始化的时候配置好，之后就不用再配置，既可以满足速度要求，又不需要频繁改配置。模式A的写操作及读操作时序分别如图25.1.3.3和图25.1.3.4所示：

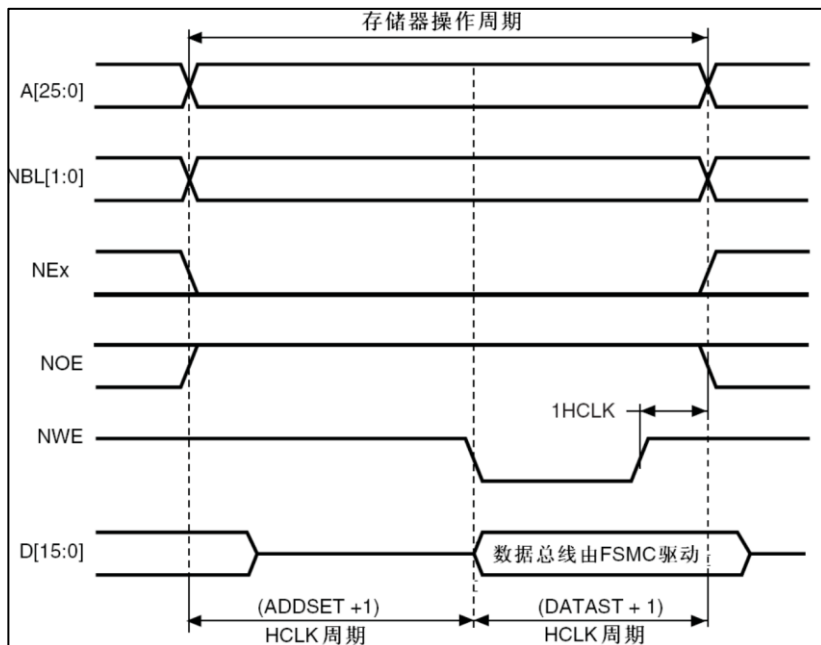


图25.1.3.3 模式A写操作时序

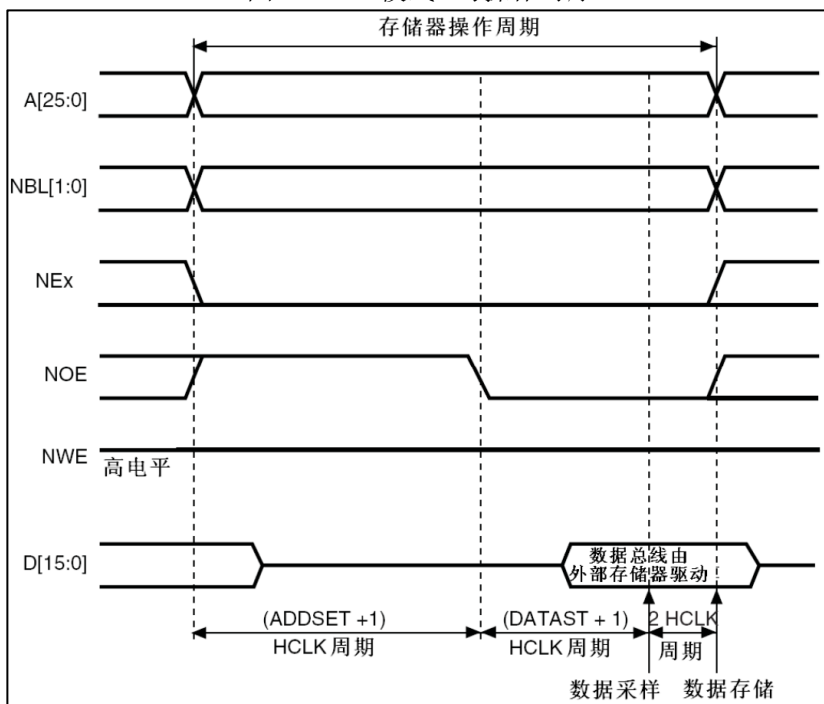


图25.1.3.4 模式A读操作时序图

图 25.1.3.3 和图 25.1.3.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的。

以图 25.1.3.3 所示的写操作时序为例，该图表示一个存储器操作周期由地址建立周期 (ADDSET)、数据建立周期 (DATAST) 组成。在地址建立周期中，数据建立周期期间 NWE 信号拉低发出写信号，接着 FSMC 把数据通过数据线传输到存储器中。注意：NEW 拉高后的那 1 个 HCLK 是必要的，以保证数据线上的信号被准确采样。

读操作模式时序类似，区别是它的一个存储器操作周期由地址建立周期 (ADDSET) 和数据建立周期 (DATAST) 以及 2 个 HCLK 周期组成，且在数据建立周期期间地址线发出要访问的地址，数据掩码信号线指示出要读取地址的高、低字节部分，片选信号使能存储器芯片；地址建立周期结束后读使能信号线发出读使能信号，接着存储器通过数据信号线把目标数据传输给 FSMC，FSMC 把它交给内核。

当 FSMC 外设被配置成正常工作，并且外部接了 PSRAM，若向 0x60000000 地址写入数据如 0xABCD，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。FSMC 会控制片选信号 NE1 输出低电平，相应的 PSRAM 芯片被片选激活，然后使用地址线 A[25:0] 输出 0x60000000，在 NWE 写使能信号线上发出低电平的写使能信号，而要写入的数据信号 0xABCD 则从数据线 D[15:0] 输出，然后数据就被保存到 PSRAM 中了。

到这里大家发现没有，之前讲的液晶控制器的 8080 并口模式与 FSMC 接口很像，区别是 FSMC 通过地址访问设备数据，并且可以自动控制相应电平，而 8080 方式则是直接控制，且没有地址线。对比图 25.1.2.2 和图 25.1.3.3，不难发现它们的相似点，我们概括如表 25.1.3.5：

| FSMC(NOR/PSRAM) | 功能 | 8080 信号线 | 功能 |
|-----------------|------|----------|---------|
| FSMC_NEx | 片选信号 | CSX | 片选信号 |
| FSMC_NWR | 写使能 | WRX | 写使能 |
| FSMC_NOE | 读使能 | RDX | 读使能 |
| FSMC_D[15:0] | 数据信号 | D[17:0] | 数据信号 |
| FSMC_A[25:0] | 地址信号 | D/CX | 数据/命令选择 |

表 25.1.3.5 FSMC (NOR/PSRAM) 方式和 8080 并口对比

如果能用某种方式把 FSMC 的地址线和 8080 方式下的等效起来，那不就可以直接用 FSMC 等效 8080 方式操作 LCD 屏的显存了？FSMC 利用地址线访问数据，并自动设置地址线和相关控制信号线的电平，如果我们对命令操作和数据操作采用不同的地址来访问，同时使得操作数据时，地址线上的一个引脚的电平为高，操作命令时，同一个引脚的电平为低的话，就可以完美解决这个问题了！

战舰 STM32 开发板把 TFT-LCD 就是用的 FSMC_NE4 做片选，把 RS 连接在 A10 上面的。我们来分析一下要实现上面的通过地址自动切换命令和数据的实现方式。

首先 NOR/PSRAM 储块地址范围：0x6000 0000 ~ 0x6FFF FFFF，基地址是 0x6000 0000，每个存储块是 64MB，那么这时候我们访问 LCD 的地址应该是第 4 个存储块，编号从 1 开始，访问 LCD 的起始地址就是 $0x6000\ 0000 + (0x400\ 0000 * (x - 1)) = 0x6C00\ 0000$ ，即从 0x6C00 0000 起的 64MB 内存地址都可以去访问 LCD。

FSMC_A10 对应地址值： $2^{10} * 2 = 0x800$ （16 位模式时，参考表 25.1.3.3 及之后对 HADDR 和 FSMC 地址线对应关系的描述：HADDR[25:1]→FSMC_A[24:0]，所以这里计算时还需要乘 2）；则写命令时的地址为： $0x6C00\ 0000 + 2^{10} * 2 = 0x6C00\ 0800$ 。写数据的地址就是使 FSMC_A10 为 0 的其它任意地址。（大家不要被地址访问的思路带进去了，以为接下来就是用 FSMC 的地址偏移来操作显存了，实际显存的操作还是归 MCU 屏管理。我们使能了 FSMC 功能后，就可以直接在我们设置的地址读写数据。实际上我们只用到了两个固定的地址：一个地址把 FSMC_A10 位置 1，另一个把该位置 0，但要保证这两个地址在各个 BANK 的管理范围内）。

STM32F1 的 FSMC 支持 8/16 位数据宽度，我们这里用到的 LCD 是 16 位宽度的，所以在设置的时候，选择 16 位宽就 OK 了。向这两个地址写的 16 进制数据会被直接送到数据线上，根据地址自动解析为命令或者数据，通过这样一个过程，我们就完成了用 FSMC 模拟 8080 并口的操作，最终完成对液晶控制器的控制。

25.1.4 FSMC 关联寄存器简介

接下来我们讲解一下 Bank1 的几个控制寄存器。

首先，我们介绍 SRAM/NOR 闪存片选控制寄存器：FSMC_BCRx (x=1~4)，该寄存器各位描述如图 25.1.4.1 所示：

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------|----|---|----|----|----|----|----|----|----|----|----|----------|-----|----|----|----|--------|--------|------|---------|---------|---------|---------|----|--------|------|----|------|----|-------|-------|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| 保留 | | | | | | | | | | | | CBURSTRW | 保留 | | | | EXTMOD | WAITEN | WREN | WAITCFG | WRAPMOD | WAITPOL | BURSTEN | 保留 | FACCEN | MWID | | MTYP | | MUXEN | MBKEN | | |
| res | | | | | | | | | | | | rw | res | | | | rw | rw | rw | rw | rw | rw | rw | rw | res | rw | rw | | rw | | rw | rw | rw |
| 位14 | | EXTMOD: 扩展模式使能 (Extended mode enable) 该位允许FSMC使用FSMC_BWTR寄存器，即允许读和写使用不同的时序。 0：不使用FSMC_BWTR寄存器，这是复位后的默认状态。 1：FSMC使用FSMC_BWTR寄存器。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位12 | | WREN: 写使能位 (Write enable bit) 该位指示FSMC是否允许/禁止对存储器的写操作。 0：禁止FSMC对存储器的写操作，否则产生一个AHB错误。 1：允许FSMC对存储器的写操作；这是复位后的默认状态。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位5:4 | | MWID: 存储器数据总线宽度 (Memory databus width) 定义外部存储器总线的宽度，适用于所有类型的存储器。 00：8位， 01：16位(复位后的默认状态)， 10：保留，不能用 11：保留，不能用 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位3:2 | | MTYP: 存储器类型 (Memory type) 定义外部存储器的类型： 00：SRAM、ROM(存储器块2...4在复位后的默认值) 01：PSRAM(Cellular RAM: CRAM) 10：NOR闪存(存储器块1在复位后的默认值) 11：保留 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位0 | | MBKEN: 存储器块使能位 (Memory bank enable bit) 开启对应的存储器块。复位后存储器块1是开启的，其它所有存储器块为禁用。访问一个禁用的存储器块将在AHB总线上产生一个错误。 0：禁用对应的存储器块。 1：启用对应的存储器块。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图25.1.4.1 FSMC_BCRx寄存器各位描述

该寄存器我们在本章用到的设置有: EXTMOD、WREN、MWID、MTYP 和 MBKEN 这几个设置, 我们将逐个介绍。

EXTMOD: 扩展模式使能位, 也就是是否允许读写不同的时序, 很明显, 我们本章需要读写不同的时序, 故该位需要设置为 1。

WREN: 写使能位。我们需要向 TFTLCD 写数据, 故该位必须设置为 1。

MWID[1:0]: 存储器数据总线宽度。我们的 TFTLCD 是 16 位数据线, 所以设置该值为 01。

MTYP[1:0]: 存储器类型。前面提到, 我们把 TFTLCD 当成 SRAM 用, 所以需要设置该值为 00。

MBKEN: 存储块使能位。这个容易理解, 我们需要用到该存储块控制 TFTLCD, 当然要使能这个存储块了。

接下来, 我们看看 SRAM/NOR 闪存片选时序寄存器: FSMC_BTRx (x=1~4), 该寄存器各位描述如图 25.1.4.2 所示:

图25.1.4.2 FSMC BTRx寄存器各位描述

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----|----|--------|----|--|----|----|----|--------|----|----|----|-----|----|--------|----|----|----|----|----|--------|----|---|---|--------|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | ACCMOD | | DATLAT | | | | CLKDIV | | | | 保留 | | DATAST | | | | | | ADDHLD | | | | ADDSET | | | | | | | |
| res | | rw | | rw | | | | rw | | | | res | | rw | | | | | | rw | | | | rw | | | | | | | |
| | | 位29:28 | | ACCMOD: 访问模式 (Access mode) 定义异步访问模式。这2位只在FSMC_BCRx寄存器的EXTMOD位为1时起作用。 00: 访问模式A 01: 访问模式B 10: 访问模式C 11: 访问模式D | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | 位15:8 | | DATAST: 数据保持时间 (Data-phase duration) 这些位定义数据的保持时间(见图162至图174)，适用于SRAM、ROM和异步总线复用模式的NOR闪存操作。 0000 0000: 保留 0000 0001: DATAST保持时间=2个HCLK时钟周期 0000 0010: DATAST保持时间=3个HCLK时钟周期 1111 1111: DATAST保持时间=256个HCLK时钟周期(这是复位后的默认数值)。 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | 位3:0 | | ADDSET: 地址建立时间 (Address setup phase duration) 这些位以HCLK周期数定义地址的建立时间(见图162至图174)，适用于SRAM、ROM和异步总线复用模式的NOR闪存操作。 0000: ADDSET建立时间=1个HCLK时钟周期 1111: ADDSET建立时间=16个HCLK时钟周期(这是复位后的默认数值)。 注：在同步NOR闪存操作中，这个参数不起作用，地址建立时间始终是1个闪存时钟周期。 | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图25.1.4.3 FSMC_BWTRx寄存器各位描述

该寄存器在本章用作写操作时序控制寄存器，需要用到的设置同样是：ACCMOD、DATAST 和 ADDSET 这三个设置。这三个设置的方法同 FSMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FSMC_BTRx 一模一样，同样是选择模式 A，另外 DATAST 和 ADDSET 则对应低电平和高电平持续时间，对 ILI9341 来说，这两个时间只需要 15ns 就够了，比读操作快得多。所以我们这里设置 DATAST 为 1，即 2 个 HCLK 周期，时间约为 28ns。然后 ADDSET（也存在性能问题）设置为 0，即 1 个 HCLK 周期，实际 WR 高电平时间就可以满足。

至此，我们对 STM32F1 的 FSMC 介绍就差不多了，通过以上两个小节的了解，我们可以开始写 LCD 的驱动代码了。不过，这里还要给大家做下科普，在 MDK 的寄存器定义里面，并没有定义 FSMC_BCRx、FSMC_BTRx、FSMC_BWTRx 等这个单独的寄存器，而是将他们进行了一些组合。

FSMC_BCRx 和 FSMC_BTRx，组合成 BTCR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FSMC_BCR1，

BTCR[1]对应 FSMC_BTR1

BTCR[2]对应 FSMC_BCR2

BTCR[3]对应 FSMC_BTR2

BTCR[4]对应 FSMC_BCR3

BTCR[5]对应 FSMC_BTR3

BTCR[6]对应 FSMC_BCR4

BTCR[7]对应 FSMC_BTR4

FSMC_BWTRx 则组合成 BWTR[7]，他们的对应关系如下：

BWTR[0]对应 FSMC_BWTR1，

BWTR[2]对应 FSMC_BWTR2，

BWTR[4]对应 FSMC_BWTR3，

BWTR[6]对应 FSMC_BWTR4，

BWTR[1]、BWTR[3]和 BWTR[5]保留，没有用到。

通过上面的讲解，通过对 FSMC 相关的寄存器的描述，大家对 FSMC 的原理有了一个初步

的认识，如果还不熟悉的朋友，请一定要搜索网络资料理解 FSMC 的原理。只有理解了原理，编程时才会得心应手。

25.2 硬件设计

1. 例程功能

使用开发板的 MCU 屏接口连接正点原子 TFTLCD 模块(仅限 MCU 屏模块),实现 TFTLCD 模块的显示。通过把 LCD 模块插入底板上的 TFTLCD 模块接口，按下复位之后，就可以看到 LCD 模块不停的显示一些信息并不断切换底色。同时该实验会显示 LCD 驱动器的 ID，并且会在串口打印（按复位一次，打印一次）。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1) LED 灯

LED0 – PB5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

TFTLCD 模块的电路见图 25.1.1.2，而开发板的 LCD 接口和正点原子 TFTLCD 模块直接可以对插，开发板上的 LCD 接口如图 25.2.1 所示：



图 25.2.1 TFTLCD 模块与开发板对接的 LCD 接口示意图

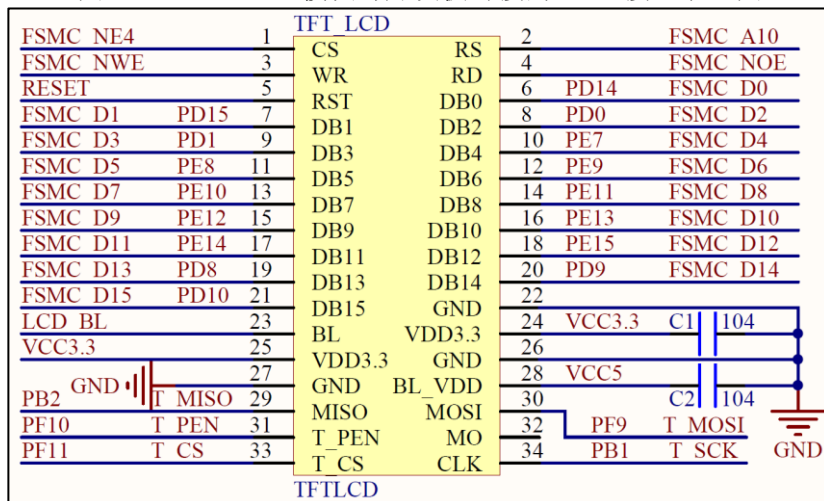


图 25.2.2 TFTLCD 模块与开发板的连接原理图

在硬件上，TFTLCD 模块与开发板的 IO 口对应关系如下：

LCD_BL(背光控制)对应 PB0；

LCD_CS 对应 PG12 即 FSMC_NE4；

LCD_RS 对应 PG0 即 FSMC_A10；

LCD_WR 对应 PD5 即 FSMC_NWE；

LCD_RD 对应 PD4 即 FSMC_NOE；

LCD_D[15:0]则直接连接在 FSMC_D15~FSMC_D0；

这些线的连接，开发板的内部已经连接好了，我们只需要将 TFTLCD 模块插上去就好了。

需要说明的是，开发板上设计的 TFT-LCD 模块插座，已经把模块模块的 RST 信号线直接接到我们开发板的复位脚上，所以不需要软件控制，这样可以省下来一个 IO 口。另外我们还需要一个背光控制线来控制 LCD 的背光灯，因为 LCD 不会自发光，没有背光灯的情况下我们是看不到 LCD 上显示的内容的。所以，我们总共需要的 IO 口数目为 22 个。

25.3 程序设计

25.3.1 FSMC 和 SRAM 的 HAL 库驱动

SRAM 和 FMC 在 HAL 库中的驱动代码在 `stm32f1xx_ll_fsmc.c/stm32f1xx_hal_sram.c` 以及 `stm32f1xx_ll_fsmc.h/stm32f1xx_hal_sram.h` 中。

1. HAL_SRAM_Init 函数

SRAM 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_SRAM_Init(SRAM_HandleTypeDef *hsram,
                                FSMC_NORSRAM_TimingTypeDef *Timing, FSMC_NORSRAM_TimingTypeDef *ExtTiming);
```

● 函数描述：

用于初始化 SRAM，注意这个函数不限制一定是 SRAM，只要时序类似，均可使用。前面说过，这里我们把 LCD 当作 SRAM 使用，因为他们时序类似。

● 函数形参：

形参 1 是 `SRAM_HandleTypeDef` 结构体类型指针变量，其定义如下：

```
typedef struct
{
    FSMC_NORSRAM_TypeDef *Instance;          /* 寄存器基地址 */
    FSMC_NORSRAM_EXTENDED_TypeDef *Extended; /* 扩展模式寄存器基地址 */
    FSMC_NORSRAM_InitTypeDef Init;           /* SRAM 初始化结构体 */
    HAL_LockTypeDef Lock;                    /* SRAM 锁对象结构体 */
    __IO HAL_SRAM_StateTypeDef State;        /* SRAM 设备访问状态 */
    DMA_HandleTypeDef *hdma;                /* DMA 结构体 */
} SRAM_HandleTypeDef;
```

1) `Instance`：指向 FSMC 寄存器基地址。我们直接写 `FSMC_NORSRAM_DEVICE` 即可，因为 HAL 库定义好了宏定义 `FSMC_NORSRAM_DEVICE`，也就是如果是 SRAM 设备，直接填写这个宏定义标识符即可。

2) `Extended`：指向 FSMC 扩展模式寄存器基地址，因为我们要配置的读写时序是不一样的。前面讲的 `FSMC_BCRx` 寄存器的 `EXTMOD` 位，我们会配置为 1 允许读写不同的时序，所以还要指定写操作时序寄存器地址，也就是通过参数 `Extended` 来指定的，这里设置为 `FSMC_NORSRAM_EXTENDED_DEVICE`。

3) `Init`：用于对 FSMC 的初始化配置，这个比较重要，后面再来讲解。

4) `Lock`：用于配置锁状态。

5) `State`：SRAM 设备访问状态。

6) `hdma`：在使用 DMA 时候才使用，这里就先不讲解了。

成员变量 `Init` 是 `FSMC_NORSRAM_InitTypeDef` 结构体指针类型，该变量才是真正用来设置 SRAM 控制接口参数的。下面详细了解这个结构体定义：

```
typedef struct
{
    uint32_t NSBank;          /* 存储区块号 */
    uint32_t DataAddressMux;  /* 地址/数据复用使能 */
    uint32_t MemoryType;      /* 存储器类型 */
    uint32_t MemoryDataWidth; /* 存储器数据宽度 */
    uint32_t BurstAccessMode; /* 突发模式配置 */
    uint32_t WaitSignalPolarity; /* 设置等待信号的极性 */
    uint32_t WrapMode;        /* 突发下存储器传输使能 */
    uint32_t WaitSignalActive; /* 等待状态之前或等待状态期间 */
    uint32_t WriteOperation;  /* 存储器写使能 */
}
```

```
uint32_t WaitSignal;          /* 使能或者禁止通过等待信号来插入等待状态 */
uint32_t ExtendedMode;       /* 使能或者禁止使能扩展模式 */
uint32_t AsynchronousWait;   /* 用于异步传输期间, 使能或者禁止等待信号 */
uint32_t WriteBurst;         /* 用于使能或者禁止异步的写突发操作 */
uint32_t PageSize;           /* 设置页大小 */
}FSMC_NORSRAM_InitTypeDef;
```

NSBank 用来指定使用到的存储块区号, 我们硬件设计时使用的存储块区号 4, 所以选择值为 FSMC_NORSRAM_BANK4。

DataAddressMux 用来设置是否使能地址/数据复用, 该变量仅对 NOR/PSRAM 有效, 所以这里我们选择不使能地址/数据复用值 FSMC_DATA_ADDRESS_MUX_DISABLE 即可。

MemoryType 用来设置存储器类型, 这里我们把 LCD 当 SRAM 使用, 所以设置为 FSMC_MEMORY_TYPE_SRAM 即可。

MemoryDataWidth 用来设置存储器数据总线宽度, 可选 8 位还是 16 位, 这里我们选择 16 位数据宽度 FSMC_NORSRAM_MEM_BUS_WIDTH_16。

WriteOperation 用来设置存储器写使能, 也就是是否允许写入。毫无疑问我们会进行存储器写操作, 所以这里设置为 FSMC_WRITE_OPERATION_ENABLE。

ExtendedMode 用来设置是否使能扩展模式, 也就是是否允许读写使用不同时序, 前面讲解过本实验读写采用不同时序, 所以设置值为使能值 FSMC_EXTENDED_MODE_ENABLE。

其他参数 WriteBurst, BurstAccessMode, WaitSignalPolarity, WaitSignalActive, WaitSignal, AsynchronousWait 等是用于在突发访问和异步时序情况下, 这里我们不做过多讲解。

形参 2 Timing 和 **形参 3 ExtTiming** 都是 FSMC_NORSRAM_TimingTypeDef 结构体类型指针变量, 其定义如下:

```
typedef struct
{
    uint32_t AddressSetupTime;          /* 地址建立时间 */
    uint32_t AddressHoldTime;           /* 地址保持时间 */
    uint32_t DataSetupTime;             /* 数据建立时间 */
    uint32_t BusTurnAroundDuration;     /* 总线周转阶段的持续时间 */
    uint32_t CLKDivision;               /* CLK 时钟输出信号的周期 */
    uint32_t DataLatency;               /* 同步突发 NOR FLASH 的数据延迟 */
    uint32_t AccessMode;                /* 异步模式配置 */
}FSMC_NORSRAM_TimingTypeDef;
```

对于本实验, 读速度比写速度慢得多, 因此读写时序不一样, 所以对于 Timing 和 ExtTiming 要设置了不同的值, 其中 Timing 设置写时序参数, ExtTiming 设置读时序参数。

下面解析一下结构体的成员变量:

AddressSetupTime 用来设置地址建立时间, 可以理解为 RD/WR 的高电平时间。

AddressHoldTime 用来设置地址保持时间, 模式 A 并没有用到。

DataSetupTime 用来设置数据建立时间, 可以理解为 RD/WR 的低电平时间。

BusTurnAroundDuration 用来配置总线周转阶段的持续时间, NOR FLASH 用到。

CLKDivision 用来配置 CLK 时钟输出信号的周期, 以 HCLK 周期数表示。若控制异步存储器, 该参数无效。

DataLatency 用来设置同步突发 NOR FLASH 的数据延迟。若控制异步存储器, 该参数无效。

AccessMode 用来设置异步模式, HAL 库允许其取值范围为 FSMC_ACCESS_MODE_A、FSMC_ACCESS_MODE_B、FSMC_ACCESS_MODE_C 和 FSMC_ACCESS_MODE_D, 这里我们用是异步模式 A, 所以取值为 FSMC_ACCESS_MODE_A。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

● 注意事项:

和其他外设一样, HAL 库也提供了 SRAM 的初始化 MSP 回调函数, 函数声明如下:

```
void HAL_SRAM_MspInit(SRAM_HandleTypeDef *hsram);
```

2. FSMC_NORSRAM_Extended_Timing_Init 函数

FSMC_NORSRAM_Extended_Timing_Init 函数是初始化扩展时序模式函数。其声明如下:

```
HAL_StatusTypeDef FSMC_NORSRAM_Extended_Timing_Init(
    FSMC_NORSRAM_EXTENDED_TypeDef *Device, FSMC_NORSRAM_TimingTypeDef *Timing,
    uint32_t Bank, uint32_t ExtendedMode);
```

- **函数描述：**
该函数用于初始化扩展时序模式。
- **函数形参：**
形参 1 是 FSMC_NORSRAM_EXTENDED_TypeDef 结构体类型指针变量，扩展模式寄存器基地址选择。
形参 2 是 FSMC_NORSRAM_TimingTypeDef 结构体类型指针变量，可以是读或者写时序结构体。
形参 3 是储存区块号。
形参 4 是使能或者禁止扩展模式。
- **函数返回值：**
HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
该函数我们用于重新配置写或者读时序。

FSMC 驱动 LCD 显示配置步骤

1) 使能 FSMC 和相关 GPIO 时钟，并设置好 GPIO 工作模式

我们通过 FSMC 控制 LCD，所以先需要使能 FSMC 以及相关 GPIO 口的时钟，并设置好 GPIO 的工作模式。

2) 设置 FSMC 参数

这里我们需要设置 FSMC 的相关访问参数（数据位宽、访问时序、工作模式等），以匹配液晶驱动 IC，这里我们通过 HAL_SRAM_Init 函数完成 FSMC 参数配置，详见本例程源码。

3) 初始化 LCD

由于我们例程兼容了很多种液晶驱动 IC，所以先要读取对应 IC 的驱动型号，然后根据不同的 IC 型号来调用不同的初始化函数，完成对 LCD 的初始化。

注意：这些初始化函数里面的代码，都是由 LCD 厂家提供，一般不需要改动，也不需要深究，我们直接照抄即可。

4) 实现 LCD 画点&读点函数

在初始化 LCD 完成以后，我们就可以控制 LCD 显示了，而最核心的一个函数，就是画点和读点函数，只要实现这两个函数，后续的各种 LCD 操作函数，都可以基于这两个函数实现。

5) 实现其他 LCD 操作函数

在完成画点和读点两个最基础的 LCD 操作函数以后，我们就可以基于这两个函数实现各种 LCD 操作函数了，比如画线、画矩形、显示字符、显示字符串、显示数字等，如果不够用还可以根据自己需要来添加。详见本例程源码。

25.3.2 程序流程图

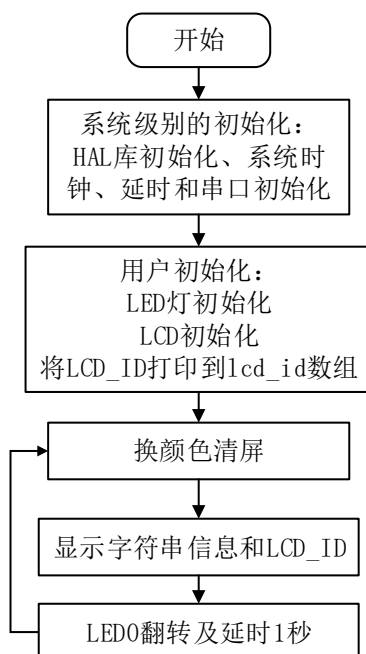


图 25.3.2.1 TFTLCD (MCU 屏) 实验程序流程图

25.3.2 程序解析

1. LCD 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。液晶（LCD）驱动源码包括四个文件：lcd.c、lcd.h、lcd_ex.c 和 lcdfont.h。

lcd.c 和 lcd.h 文件是驱动函数和引脚接口宏定义以及函数声明等。lcd_ex.c 存放各个 LCD 驱动 IC 的寄存器初始化部分代码，是 lcd.c 文件的补充文件，起到简化 lcd.c 文件的作用。lcdfont.h 头文件存放了 4 种字体大小不一样的 ASCII 字符集（12*12、16*16、24*24 和 32*32）。这个跟 oledfont.h 头文件一样的，只是这里多了 32*32 的 ASCII 字符集，制作方法请回顾 OLED 实验。

下面我们还是先介绍 lcd.h 文件，首先是 LCD 的引脚定义：

```

/* LCD RST/WR/RD/BL/CS/RS 引脚 定义
 * LCD_D0~D15, 由于引脚太多, 就不在这里定义了, 直接在 lcd_init 里面修改. 所以在移植的时候, 除了
 * 改这 6 个 IO 口, 还得改 LCD_Init 里面的 D0~D15 所在的 IO 口.
 */

/* RESET 和系统复位脚共用 所以这里不用定义 RESET 引脚 */

#define LCD_WR_GPIO_PORT      GPIOD
#define LCD_WR_GPIO_PIN      GPIO_PIN_5
#define LCD_WR_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

#define LCD_RD_GPIO_PORT      GPIOD
#define LCD_RD_GPIO_PIN      GPIO_PIN_4
#define LCD_RD_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

#define LCD_BL_GPIO_PORT      GPIOB
#define LCD_BL_GPIO_PIN      GPIO_PIN_0
#define LCD_BL_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

/* LCD_CS (需要根据 LCD_FSMC_NEX 设置正确的 IO 口) 和 LCD_RS (需要根据 LCD_FSMC_AX 设置正确的
  IO 口) 引脚 定义 */
#define LCD_CS_GPIO_PORT      GPIOG
#define LCD_CS_GPIO_PIN      GPIO_PIN_12

```



```
#define LCD_CS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIO_CLK_ENABLE();}while(0)

#define LCD_RS_GPIO_PORT          GPIOG
#define LCD_RS_GPIO_PIN           GPIO_PIN_0
#define LCD_RS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0)
```

第一部分的宏定义是 LCD WR/RD/BL/CS/RS/DATA 引脚定义,需要注意的是:LCD 的 RST 引脚和系统复位脚连接在一起,所以不用单独使用一个 IO 口(节省一个 IO 口)。而 DATA 引脚直接用的是 FSMC_D[x]引脚,具体可以查看前面的描述。

下面介绍我们在 lcd.h 里面定义的一个重要的结构体:

```
/* LCD 重要参数集 */
typedef struct
{
    uint16_t width;      /* LCD 宽度 */
    uint16_t height;     /* LCD 高度 */
    uint16_t id;         /* LCD ID */
    uint8_t dir;         /* 横屏还是竖屏控制: 0, 竖屏; 1, 横屏。 */
    uint16_t wramcmd;     /* 开始写 gram 指令 */
    uint16_t setxcmd;     /* 设置 x 坐标指令 */
    uint16_t setycmd;     /* 设置 y 坐标指令 */
} _lcd_dev;
extern _lcd_dev lcddev; /* 管理 LCD 重要参数 */
/* LCD 的画笔颜色和背景色 */
extern uint32_t g_point_color; /* 默认红色 */
extern uint32_t g_back_color; /* 背景颜色,默认为白色 */
```

该结构体用于保存一些 LCD 重要参数信息,比如 LCD 的长宽、LCD ID(驱动 IC 型号)、LCD 横竖屏状态等,这个结构体虽然占用了十几个字节的内存,但是却可以让我们的驱动函数支持不同尺寸的 LCD,同时可以实现 LCD 横竖屏切换等重要功能,所以还是利大于弊的。最后声明 _lcd_dev 结构体类型变量 lcddev, lcddev 在 lcd.c 中定义。

紧接着就是 g_point_color 和 g_back_color 变量的声明,它们也是在 lcd.c 中被定义。g_point_color 变量用于保存 LCD 的画笔颜色, g_back_color 则是保存 LCD 的背景色。

下面是 LCD 背光控制 IO 口的宏定义:

```
/* LCD 背光控制 */
#define LCD_BL(x) do{ x ? \
    HAL_GPIO_WritePin(LCD_BL_GPIO_PORT, LCD_BL_GPIO_PIN, GPIO_PIN_SET) : \
    HAL_GPIO_WritePin(LCD_BL_GPIO_PORT, LCD_BL_GPIO_PIN, GPIO_PIN_RESET); \
}while(0)
```

本实验,我们用到 FSMC 驱动 LCD,通过前面的介绍,我们知道 TFTLCD 的 RS 接在 FSMC 的 A10 上面, CS 接在 FSMC_NE4 上,并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区,我们定义如下 LCD 操作结构体(在 lcd.h 里面定义):

```
/* LCD 地址结构体 */
typedef struct
{
    volatile uint16_t LCD_REG;
    volatile uint16_t LCD_RAM;
} LCD_TypeDef;

/* LCD_BASE 的详细解算方法:
* 我们一般使用 FSMC 的块 1 (BANK1) 来驱动 TFTLCD 液晶屏 (MCU 屏), 块 1 地址范围总大小为 256MB,
均分成 4 块:
* 存储块 1 (FSMC_NE1) 地址范围: 0X6000 0000 ~ 0X63FF FFFF
* 存储块 2 (FSMC_NE2) 地址范围: 0X6400 0000 ~ 0X67FF FFFF
* 存储块 3 (FSMC_NE3) 地址范围: 0X6800 0000 ~ 0X6BFF FFFF
* 存储块 4 (FSMC_NE4) 地址范围: 0X6C00 0000 ~ 0X6FFF FFFF
*
* 我们需要根据硬件连接方式选择合适的片选 (连接 LCD_CS) 和地址线 (连接 LCD_RS)
* 战舰 F103 开发板使用 FSMC_NE4 连接 LCD_CS, FSMC_A10 连接 LCD_RS, 16 位数据线, 计算方法如下:
* 首先 FSMC_NE4 的基地址为: 0X6C00 0000; NEx 的基址为 (x=1/2/3/4): 0X6000 0000 +
(0X400 0000 * (x - 1))
```

```

* FSMC_A10 对应地址值:  $2^{10} * 2 = 0X800$ ; FSMC_Ay 对应的地址为 ( $y = 0 \sim 25$ ):  $2^y * 2$ 
*
* LCD->LCD_REG, 对应 LCD_RS = 0 (LCD 寄存器); LCD->LCD_RAM, 对应 LCD_RS = 1 (LCD 数据)
* 则 LCD->LCD_RAM 的地址为:  $0X6C00\ 0000 + 2^{10} * 2 = 0X6C00\ 0800$ 
* LCD->LCD_REG 的地址可以为 LCD->LCD_RAM 之外的任意地址。
* 由于我们使用结构体管理 LCD_REG 和 LCD_RAM (REG 在前, RAM 在后, 均为 16 位数据宽度)
* 因此 结构体的基地址 (LCD_BASE) = LCD_RAM - 2 =  $0X6C00\ 0800 - 2$ 
*
* 更加通用的计算公式为 ((片选脚 FSMC_NEx)  $x=1/2/3/4$ , (RS 接地址线 FSMC_Ay)  $y=0\sim25$ ):
* LCD_BASE =  $(0X6000\ 0000 + (0X400\ 0000 * (x - 1))) | (2^y * 2 - 2)$ 
* 等效于 (使用移位操作)
* LCD_BASE =  $(0X6000\ 0000 + (0X400\ 0000 * (x - 1))) | ((1 << y) * 2 - 2)$ 
*/
#define LCD_BASE (uint32_t)((0X60000000 + (0X4000000 * (LCD_FSMC_NEX - 1))) |
                           (((1 << LCD_FSMC_AX) * 2) - 2))
#define LCD ((LCD_TypeDef *) LCD_BASE)

```

其中 LCD_BASE, 必须根据我们外部电路的连接来确定, 我们使用 BANK1 的存储块 4 的寻址范围为 0X6C000000~6FFFFFFF, 我们需要在这个地址范围内找到两个地址, 实现对 RS 位 (FSMC_A10 位) 的 0 和 1 的控制。这两个地址的取值方法, 我们在前面的 25.1.3 的末尾已经详细说明了。为了方便控制和节省内存, 我们使这两个地址变成相邻的两个 16 进制指针, 这样就可以用前面定义的 LCD_TypeDef 来管理这两个地址了。

根据我们的算法和定义, 我们将这个地址强制转换为 LCD_TypeDef 结构体地址, 那么可以得到 LCD->LCD_REG 的地址就是 0X6C00 07FE, 对应 A10 的状态为 0 (即 RS=0), 而 LCD->LCD_RAM 的地址就是 0X6C00 0800 (结构体地址自增), 对应 A10 的状态为 1 (即 RS=1)。

所以, 有了这个定义, 当我们要往 LCD 写命令/数据的时候, 可以这样写:

```

LCD->LCD_REG = CMD; /* 写命令 */
LCD->LCD_RAM = DATA; /* 写数据 */

```

而读的时候反过来操作就可以了, 如下所示:

```

CMD = LCD->LCD_REG; /* 读 LCD 寄存器 */
DATA = LCD->LCD_RAM; /* 读 LCD 数据 */

```

这其中, CS、WR、RD 和 IO 口方向都是由 FSMC 硬件自动控制, 不需要我们手动设置了。

最后是一些其他的宏定义, 包括 LCD 扫描方向和颜色, 以及 SSD1963 相关配置参数。

下面开始对 lcd.c 文件介绍, 先看 LCD 初始化函数, 其定义如下:

```

/**
 * @brief      初始化 LCD
 * @note      该初始化函数可以初始化各种型号的 LCD (详见本.c 文件最前面的描述)
 *
 * @param      无
 * @retval     无
 */
void lcd_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    FSMC_NORSRAM_TimingTypeDef fsmc_read_handle;
    FSMC_NORSRAM_TimingTypeDef fsmc_write_handle;

    LCD_CS_GPIO_CLK_ENABLE(); /* LCD_CS 脚时钟使能 */
    LCD_WR_GPIO_CLK_ENABLE(); /* LCD_WR 脚时钟使能 */
    LCD_RD_GPIO_CLK_ENABLE(); /* LCD_RD 脚时钟使能 */
    LCD_RS_GPIO_CLK_ENABLE(); /* LCD_RS 脚时钟使能 */
    LCD_BL_GPIO_CLK_ENABLE(); /* LCD_BL 脚时钟使能 */

    gpio_init_struct.Pin = LCD_CS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* 推挽复用 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(LCD_CS_GPIO_PORT, &gpio_init_struct); /* 初始化 LCD_CS 引脚 */

```

```

gpio_init_struct.Pin = LCD_WR_GPIO_PIN;
HAL_GPIO_Init(LCD_WR_GPIO_PORT, &gpio_init_struct);    /* 初始化 LCD_WR 引脚 */

gpio_init_struct.Pin = LCD_RD_GPIO_PIN;
HAL_GPIO_Init(LCD_RD_GPIO_PORT, &gpio_init_struct);    /* 初始化 LCD_RD 引脚 */

gpio_init_struct.Pin = LCD_RS_GPIO_PIN;
HAL_GPIO_Init(LCD_RS_GPIO_PORT, &gpio_init_struct);    /* 初始化 LCD_RS 引脚 */

gpio_init_struct.Pin = LCD_BL_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(LCD_BL_GPIO_PORT, &gpio_init_struct);    /* 初始化 LCD_BL 引脚 */

g_sram_handle.Instance = FSMC_NORSRAM_DEVICE;
g_sram_handle.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;

g_sram_handle.Init.NSBank = FSMC_NORSRAM_BANK4;        /* 使用 NE4 */
/*地址/数据线不复用*/
g_sram_handle.Init.DataAddressMux = FSMC_DATA_ADDRESS_MUX_DISABLE;
/*16 位数据宽度*/
g_sram_handle.Init.MemoryDataWidth = FSMC_NORSRAM_MEM_BUS_WIDTH_16;
/*是否使能突发访问,仅对同步突发存储器有效,此处未用到*/
g_sram_handle.Init.BurstAccessMode = FSMC_BURST_ACCESS_MODE_DISABLE;
/*等待信号的极性,仅在突发模式访问下有用*/
g_sram_handle.Init.WaitSignalPolarity = FSMC_WAIT_SIGNAL_POLARITY_LOW;
/* 存储器是在等待周期之前的一个时钟周期还是等待周期期间使能 NWAIT */
g_sram_handle.Init.WaitSignalActive = FSMC_WAIT_TIMING_BEFORE_WS;
/* 存储器写使能 */
g_sram_handle.Init.WriteOperation = FSMC_WRITE_OPERATION_ENABLE;
/* 等待使能位,此处未用到 */
g_sram_handle.Init.WaitSignal = FSMC_WAIT_SIGNAL_DISABLE;
/* 读写使用不同的时序 */
g_sram_handle.Init.ExtendedMode = FSMC_EXTENDED_MODE_ENABLE;
/* 是否使能同步传输模式下的等待信号,此处未用到 */
g_sram_handle.Init.AsynchronousWait = FSMC_ASYNCHRONOUS_WAIT_DISABLE;
g_sram_handle.Init.WriteBurst = FSMC_WRITE_BURST_DISABLE; /* 禁止突发写 */

/*FSMC 读时序控制寄存器*/
/* 地址建立时间(ADDSET)为 1 个 HCLK 1/72M=13.9ns */
fsmc_read_handle.AddressSetupTime = 0;
fsmc_read_handle.AddressHoldTime = 0;
/* 数据保存时间(DATAST)为 16 个 HCLK 13.9*16=222.4ns */
fsmc_read_handle.DataSetupTime = 15; /* 部分液晶驱动 IC 读数据时,速度不能太快 */
fsmc_read_handle.AccessMode = FSMC_ACCESS_MODE_A; /* 模式 A */

/*FSMC 写时序控制寄存器*/
/* 地址建立时间(ADDSET)为 1 个 HCLK 13.9ns */
fsmc_write_handle.AddressSetupTime = 0;
fsmc_write_handle.AddressHoldTime = 0;
/* 数据保存时间(DATAST)为 2 个 HCLK 13.9*2= 27.8ns */
fsmc_write_handle.DataSetupTime = 1;
fsmc_write_handle.AccessMode = FSMC_ACCESS_MODE_A; /* 模式 A */
HAL_SRAM_Init(&g_sram_handle, &fsmc_read_handle, &fsmc_write_handle);
delay_ms(50);

/* 尝试 9341 ID 的读取 */
lcd_wr_regno(0XD3);
lcddev.id = lcd_rd_data(); /* dummy read */
lcddev.id = lcd_rd_data(); /* 读到 0x00 */
lcddev.id = lcd_rd_data(); /* 读取 93 */
lcddev.id <= 8;
lcddev.id |= lcd_rd_data(); /* 读取 41 */

```

```

if (lcddev.id != 0X9341)                /* 不是 9341 , 尝试看看是不是 ST7789 */
{
    lcd_wr_regno(0X04);
    lcddev.id = lcd_rd_data();          /* dummy read */
    lcddev.id = lcd_rd_data();          /* 读到 0X85 */
    lcddev.id = lcd_rd_data();          /* 读取 0X85 */
    lcddev.id <= 8;
    lcddev.id |= lcd_rd_data();          /* 读取 0X52 */

    if (lcddev.id == 0X8552)             /* 将 8552 的 ID 转换成 7789 */
    {
        lcddev.id = 0x7789;
    }

    if (lcddev.id != 0x7789)            /* 也不是 ST7789, 尝试是不是 NT35310 */
    {
        lcd_wr_regno(0XD4);
        lcddev.id = lcd_rd_data();      /* dummy read */
        lcddev.id = lcd_rd_data();      /* 读回 0x01 */
        lcddev.id = lcd_rd_data();      /* 读回 0x53 */
        lcddev.id <= 8;
        lcddev.id |= lcd_rd_data();     /* 这里读回 0x10 */

        if (lcddev.id != 0x5310)        /* 也不是 NT35310, 尝试看看是不是 ST7796 */
        {
            lcd_wr_regno(0XD3);
            lcddev.id = lcd_rd_data();    /* dummy read */
            lcddev.id = lcd_rd_data();    /* 读到 0X00 */
            lcddev.id = lcd_rd_data();    /* 读取 0X77 */
            lcddev.id <= 8;
            lcddev.id |= lcd_rd_data();    /* 读取 0X96 */

            if (lcddev.id != 0x7796)     /* 也不是 ST7796, 尝试看看是不是 NT35510 */
            {
                /* 发送密钥 (厂家提供) */
                lcd_write_reg(0xF000, 0x0055);
                lcd_write_reg(0xF001, 0x00AA);
                lcd_write_reg(0xF002, 0x0052);
                lcd_write_reg(0xF003, 0x0008);
                lcd_write_reg(0xF004, 0x0001);

                lcd_wr_regno(0xC500);      /* 读取 ID 低八位 */
                lcddev.id = lcd_rd_data(); /* 读回 0x55 */
                lcddev.id <= 8;

                lcd_wr_regno(0xC501);      /* 读取 ID 高八位 */
                lcddev.id |= lcd_rd_data(); /* 读回 0x10 */

                /* 等待 5ms, 因为 0XC501 指令对 1963 来说就是软件复位指令,
                 * 等待 5ms 让 1963 复位完成再操作 */
                delay_ms(5);

                if (lcddev.id != 0x5510) /* 也不是 NT5510, 尝试看看是不是 ILI9806 */
                {
                    lcd_wr_regno(0XD3);
                    lcddev.id = lcd_rd_data(); /* dummy read */
                    lcddev.id = lcd_rd_data(); /* 读回 0X00 */
                    lcddev.id = lcd_rd_data(); /* 读回 0X98 */
                    lcddev.id <= 8;
                    lcddev.id |= lcd_rd_data(); /* 读回 0X06 */
                }
            }
        }
    }
}

```

```

        if (lcddev.id != 0x9806) /* 也不是 ILI9806, 尝试是不是 SSD1963 */
        {
            lcd_wr_regno(0xA1);
            lcddev.id = lcd_rd_data();
            lcddev.id = lcd_rd_data(); /* 读回 0x57 */
            lcddev.id <<= 8;
            lcddev.id |= lcd_rd_data(); /* 读回 0x61 */

            /* SSD1963 读回的 ID 是 5761H, 为方便区分, 我们强制设置为 1963 */
            if (lcddev.id == 0x5761) lcddev.id = 0x1963;
        }
    }
}

/* 特别注意, 如果在 main 函数里面屏蔽串口 1 初始化, 则会卡死在 printf
 * 里面 (卡死在 f_putc 函数), 所以, 必须初始化串口 1, 或者屏蔽掉下面
 * 这行 printf 语句 !!!!!!!
 */
printf("LCD ID:%x\r\n", lcddev.id); /* 打印 LCD ID */

if (lcddev.id == 0x7789)
{
    lcd_ex_st7789_reginit(); /* 执行 ST7789 初始化 */
}
else if (lcddev.id == 0x9341)
{
    lcd_ex_ili9341_reginit(); /* 执行 ILI9341 初始化 */
}
else if (lcddev.id == 0x5310)
{
    lcd_ex_nt35310_reginit(); /* 执行 NT35310 初始化 */
}
else if (lcddev.id == 0x7796)
{
    lcd_ex_st7796_reginit(); /* 执行 ST7796 初始化 */
}
else if (lcddev.id == 0x5510)
{
    lcd_ex_nt35510_reginit(); /* 执行 NT35510 初始化 */
}
else if (lcddev.id == 0x9806)
{
    lcd_ex_ili9806_reginit(); /* 执行 ILI9806 初始化 */
}
else if (lcddev.id == 0x1963)
{
    lcd_ex_ssd1963_reginit(); /* 执行 SSD1963 初始化 */
    lcd_ssd_backlight_set(100); /* 背光设置为最亮 */
}

lcd_display_dir(0); /* 默认为竖屏 */
LCD_BL(1); /* 点亮背光 */
lcd_clear(WHITE);
}

```

该函数先对 FSMC 相关 IO 进行初始化, 然后使用 HAL_SRAM_Init 函数初始化 FSMC 控制器, 同时我们使用 HAL_SRAM_MspInit 回调函数来初始化相应的 IO 口, 最后读取 LCD 控制器的型号, 根据控制 IC 的型号执行不同的初始化代码, 这样提高了整个程序的通用性。为了简化 lcd.c 的初始化程序, 不同控制 IC 的芯片对应的初始化程序 (如: lcd_ex_st7789_reginit()、

lcd_ex_ili9341_reginit()等) 我们放在 lcd_ex.c 文件中, 这些初始化代码完成对 LCD 寄存器的初始化, 由 LCD 厂家提供, 一般是不需要做任何修改的, 我们直接调用就可以了。

下面是 6 个简单, 但是很重要的函数:

```
/**
 * @brief      LCD 写数据
 * @param      data: 要写入的数据
 * @retval     无
 */
void lcd_wr_data(volatile uint16_t data)
{
    data = data;          /* 使用-O2 优化的时候, 必须插入的延时 */
    LCD->LCD_RAM = data;
}

/**
 * @brief      LCD 写寄存器编号/地址函数
 * @param      regno: 寄存器编号/地址
 * @retval     无
 */
void lcd_wr_regno(volatile uint16_t regno)
{
    regno = regno;        /* 使用-O2 优化的时候, 必须插入的延时 */
    LCD->LCD_REG = regno; /* 写入要写的寄存器序号 */
}

/**
 * @brief      LCD 写寄存器
 * @param      regno: 寄存器编号/地址
 * @param      data: 要写入的数据
 * @retval     无
 */
void lcd_write_reg(uint16_t regno, uint16_t data)
{
    LCD->LCD_REG = regno; /* 写入要写的寄存器序号 */
    LCD->LCD_RAM = data;  /* 写入数据 */
}

/**
 * @brief      LCD 延时函数, 仅用于部分在 mdk -O1 时间优化时需要设置的地方
 * @param      t: 延时的数值
 * @retval     无
 */
static void lcd_opt_delay(uint32_t i)
{
    while (i--); /* 使用 AC6 时空循环可能被优化, 可使用 while(1) __asm volatile(""); */
}

/**
 * @brief      LCD 读数据
 * @param      无
 * @retval     读取到的数据
 */
static uint16_t lcd_rd_data(void)
{
    volatile uint16_t ram; /* 防止被优化 */
    lcd_opt_delay(2);
    ram = LCD->LCD_RAM;
    return ram;
}

/**
 * @brief      准备写 GRAM

```

```

* @param      无
* @retval     无
*/
void lcd_write_ram_prepare(void)
{
    LCD->LCD_REG = lcddev.wramcmd;
}

```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 6 个函数实现起来都非常简单，我们就不多说，注意，上面有几个函数，我们添加了一些对 MDK - O2 优化的支持，去掉的话，在 -O2 优化的时候会出问题。这些函数实现功能见函数前面的备注，通过这几个简单函数的组合，我们就可以对 LCD 进行各种操作了。

下面要介绍的函数是坐标设置函数，该函数代码如下：

```

/**
 * @brief      设置光标位置(对 RGB 屏无效)
 * @param      x,y: 坐标
 * @retval     无
 */
void lcd_set_cursor(uint16_t x, uint16_t y)
{
    if (lcddev.id == 0X1963)
    {
        if (lcddev.dir == 0) /* 竖屏模式，x 坐标需要变换 */
        {
            x = lcddev.width - 1 - x;
            lcd_wr_regno(lcddev.setxcmd);
            lcd_wr_data(0);
            lcd_wr_data(0);
            lcd_wr_data(x >> 8);
            lcd_wr_data(x & 0XFF);
        }
        else /* 横屏模式 */
        {
            lcd_wr_regno(lcddev.setxcmd);
            lcd_wr_data(x >> 8);
            lcd_wr_data(x & 0XFF);
            lcd_wr_data((lcddev.width - 1) >> 8);
            lcd_wr_data((lcddev.width - 1) & 0XFF);
        }

        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
        lcd_wr_data(y & 0XFF);
        lcd_wr_data((lcddev.height - 1) >> 8);
        lcd_wr_data((lcddev.height - 1) & 0XFF);
    }
    else if (lcddev.id == 0X5510)
    {
        lcd_wr_regno(lcddev.setxcmd);
        lcd_wr_data(x >> 8);
        lcd_wr_regno(lcddev.setxcmd + 1);
        lcd_wr_data(x & 0XFF);
        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
        lcd_wr_regno(lcddev.setycmd + 1);
        lcd_wr_data(y & 0XFF);
    }
    else /* 9341/5310/7789/7796/9806 等 设置坐标 */
    {
        lcd_wr_regno(lcddev.setxcmd);
        lcd_wr_data(x >> 8);
        lcd_wr_data(x & 0XFF);
        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
    }
}

```

```

        lcd_wr_data(y & 0xFF);
    }
}

```

该函数实现将 LCD 的当前操作点设置到指定坐标(x,y)。因为 9341/5310/1963/5510 等的设置有些不太一样，所以进行了区别对待。

接下来介绍画点函数，其定义如下：

```

/**
 * @brief      画点
 * @param      x,y: 坐标
 * @param      color: 点的颜色(32 位颜色,方便兼容 LTDC)
 * @retval     无
 */
void lcd_draw_point(uint16_t x, uint16_t y, uint32_t color)
{
    lcd_set_cursor(x, y);          /* 设置光标位置 */
    lcd_write_ram_prepare();       /* 开始写入 GRAM */
    LCD->LCD_RAM = color;
}

```

该函数实现比较简单，就是先设置坐标，然后往坐标写颜色。lcd_draw_point 函数虽然简单，但是至关重要，其他几乎所有上层函数，都是通过调用这个函数实现的。

下面介绍读点函数，用于读取 LCD 的 GRAM，这里说明一下，为什么 OLED 模块没做读 GRAM 的函数，而这里做了。因为 OLED 模块是单色的，所需要全部 GRAM 也就 1K 个字节，而 TFTLCD 模块为彩色的，点数也比 OLED 模块多很多，以 16 位色计算，一款 320×240 的液晶，需要 320×240×2 个字节来存储颜色值，也就是也需要 150K 字节，这对任何一款单片机来说，都不是一个小数目了。而且我们在图形叠加的时候，可以先读回原来的值，然后写入新的值，在完成叠加后，我们又恢复原来的值。这样在做一些简单菜单的时候，是很有用的。这里我们读取 TFTLCD 模块数据的函数为 LCD_ReadPoint，该函数直接返回读到的 GRAM 值。该函数使用之前要先设置读取的 GRAM 地址，通过 lcd_set_cursor 函数来实现。lcd_read_point 的代码如下：

```

/**
 * @brief      读取个某点的颜色值
 * @param      x,y:坐标
 * @retval     此点的颜色(32 位颜色,方便兼容 LTDC)
 */
uint32_t lcd_read_point(uint16_t x, uint16_t y)
{
    uint16_t r = 0, g = 0, b = 0;

    if (x >= lcddev.width || y >= lcddev.height) return 0; /* 超过了范围,直接返回 */

    lcd_set_cursor(x, y);          /* 设置坐标 */

    if (lcddev.id == 0x5510)
    {
        lcd_wr_regno(0x2E00); /* 5510 发送读 GRAM 指令 */
    }
    else
    {
        lcd_wr_regno(0x2E); /* 9341/5310/1963/7789/7796/9806 等发送读 GRAM 指令 */
    }

    r = lcd_rd_data();             /* 假读(dummy read) */

    if (lcddev.id == 0x1963) return r; /* 1963 直接读就可以 */

    r = lcd_rd_data();             /* 实际坐标颜色 */

    /* ILI9341/NT35310/NT35510/ST7789/ILI9806 要分 2 次读出 */
    b = lcd_rd_data();
}

```

```

/* 对于 9341/5310/5510/7789/9806, 第一次读取的是 RG 的值,R 在前,G 在后,各占 8 位 */
g = r & 0xFF;
g <<= 8;
/* 9341/5310/5510/7789/9806 需要公式转换一下 */
return (((r >> 11) << 11) | ((g >> 10) << 5) | (b >> 11));
}

```

在 `lcd_read_point` 函数中, 因为我们的代码不止支持一种 LCD 驱动器, 所以, 我们根据不同的 LCD 驱动器 (`lcddev.id`) 型号, 执行不同的操作, 以实现各个驱动器兼容, 提高函数的通用性。

第十个要介绍的是字符显示函数 `lcd_show_char`, 该函数同前面 OLED 模块的字符显示函数差不多, 但是这里的字符显示函数多了 1 个功能, 就是以叠加方式显示, 或者以非叠加方式显示。叠加方式显示多用于在显示的图片上再显示字符。非叠加方式一般用于普通的显示。该函数实现代码如下:

```

/**
 * @brief      在指定位置显示一个字符
 * @param      x,y : 坐标
 * @param      chr : 要显示的字符:" "---->"~"
 * @param      size : 字体大小 12/16/24/32
 * @param      mode : 叠加方式(1); 非叠加方式(0);
 * @retval     无
 */
void lcd_show_char(uint16_t x, uint16_t y, char chr, uint8_t size,
                  uint8_t mode, uint16_t color)
{
    uint8_t temp, t1, t;
    uint16_t y0 = y;
    uint8_t csize = 0;
    uint8_t *pfont = 0;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size / 2);
    /* 得到偏移后的值 (ASCII 字库是从空格开始取模, 所以-' '就是对应字符的字库) */
    chr = chr - ' ';

    switch (size)
    {
        case 12:
            pfont = (uint8_t *)asc2_1206[chr]; /* 调用 1206 字体 */
            break;

        case 16:
            pfont = (uint8_t *)asc2_1608[chr]; /* 调用 1608 字体 */
            break;

        case 24:
            pfont = (uint8_t *)asc2_2412[chr]; /* 调用 2412 字体 */
            break;

        case 32:
            pfont = (uint8_t *)asc2_3216[chr]; /* 调用 3216 字体 */
            break;

        default:
            return ;
    }

    for (t = 0; t < csize; t++)
    {
        temp = pfont[t]; /* 获取字符的点阵数据 */

        for (t1 = 0; t1 < 8; t1++) /* 一个字节 8 个点 */
        {
            if (temp & 0x80) /* 有效点, 需要显示 */

```

```

    {
        lcd_draw_point(x, y, color);        /* 画点出来,要显示这个点 */
    }
    else if (mode == 0) /* 无效点,不显示 */
    {
        /* 画背景色,相当于这个点不显示(注意背景色由全局变量控制) */
        lcd_draw_point(x, y, g_back_color);
    }

    temp <= 1; /* 移位,以便获取下一个位的状态 */
    y++;

    if (y >= lcddev.height) return;        /* 超区域了 */

    if ((y - y0) == size) /* 显示完一列了? */
    {
        y = y0; /* y 坐标复位 */
        x++;    /* x 坐标递增 */

        if (x >= lcddev.width) return;    /* x 坐标超区域了 */

        break;
    }
}
}
}

```

在 lcd_show_char 函数里面,我们用到了四个字符集点阵数据数组 asc2_1206、asc2_1608、asc2_2412 和 asc2_3216。

lcd.c 的函数比较多,其他的函数请大家自行查看源码,都有详细的注释。

2. main.c 代码

在 main.c 里面编写如下代码:

```

int main(void)
{
    uint8_t x = 0;
    uint8_t lcd_id[12];

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    g_point_color = RED;
    sprintf((char *)lcd_id, "LCD ID:%04X", lcddev.id); /* 将 id 打印到 lcd_id 数组 */

    while (1)
    {
        switch (x)
        {
            case 0: lcd_clear(WHITE); break;
            case 1: lcd_clear(BLACK); break;
            case 2: lcd_clear(BLUE); break;
            case 3: lcd_clear(RED); break;
            case 4: lcd_clear(MAGENTA); break;
            case 5: lcd_clear(GREEN); break;
            case 6: lcd_clear(CYAN); break;
            case 7: lcd_clear(YELLOW); break;
            case 8: lcd_clear(BRRED); break;
            case 9: lcd_clear(GRAY); break;
            case 10: lcd_clear(LGRAY); break;
        }
    }
}

```



```
case 11: lcd_clear(BROWN); break;
}

lcd_show_string(10, 40, 240, 32, 32, "STM32", RED);
lcd_show_string(10, 80, 240, 24, 24, "TFTLCD TEST", RED);
lcd_show_string(10, 110, 240, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(10, 130, 240, 16, 16, (char *)lcd_id, RED); /*显示 LCD_ID*/

x++;
if (x == 12)
    x = 0;

LED0_TOGGLE(); /* 红灯闪烁 */
delay_ms(1000);
}
}
```

main 函数功能主要是显示一些固定的字符，字体大小包括 32*16、24*12、16*8 和 12*6 四种，同时显示 LCD 驱动 IC 的型号，然后不停的切换背景颜色，每 1s 切换一次。而 LED0 也会不停的闪烁，指示程序已经在运行了。其中我们用到一个 sprintf 的函数，该函数用法同 printf，只是 sprintf 把打印内容输出到指定的内存区间上，最终在死循环中通过 lcd_show_string 函数进行屏幕显示，sprintf 的详细用法，请百度学习。

特别注意：

usart_init 函数，不能去掉，因为在 lcd_init 函数里面调用了 printf，所以一旦去掉这个初始化，就会死机！实际上，只要你的代码有用到 printf，就必须初始化串口，否则都会死机，停在 usart.c 里面的 fputc 函数出不来。

25.4 下载验证

下载代码后，LED0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块的显示背景色不停切换，如图 25.4.1 所示：

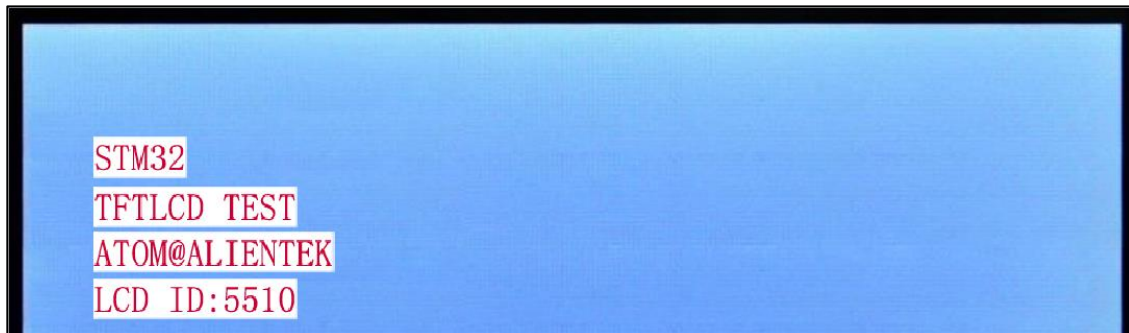


图 25.4.1 TFTLCD 显示效果图

此外，为了让大家能直观的了解 LCD 屏的扫描方式，我们额外编写了两个 main.c 文件（main1.c 和 main2.c，放到 User 文件夹中），方便大家编译下载，观察现象。

使用方法：关闭工程后，先把原实验中的 main.c 改成其他名字，然后把 main1.c 重命名为 main.c，双击 keilkill.bat 清理编译的中间文件，最后打开工程重新编译下载，就可以观察实验现象。观察了 main1.c，可以再观察 main2.c，main2.c 文件的操作方法类似。这两个 main.c 文件的程序非常简单，这里就不讲解，具体请看源码。

第二十六章 USART 调试组件实验

本章，我们将向大家介绍一个十分重要的辅助调试工具：USART 调试组件。该组件由正点原子开发提供，功能类似 linux 的 shell（RTT 的 finsh 也属于此类）。USART 最主要的功能就是通过串口调用单片机里面的函数，并执行，对我们调试代码是很有帮助的。

本章分为如下几个小节：

- 26.1 USART 调试组件简介
- 26.2 硬件设计
- 26.3 程序设计
- 26.4 下载验证

26.1 USART 调试组件简介

USART 是由正点原子开发的一个灵巧的串口调试交互组件，通过它你可以通过串口助手调用程序里面的任何函数，并执行。因此，你可以随意更改函数的输入参数（支持数字（10/16 进制，支持负数）、字符串、函数入口地址等作为参数），单个函数最多支持 10 个输入参数，并支持函数返回值显示，目前最新版本为 V3.5。

USART 的特点如下：

- 1， 可以调用绝大部分用户直接编写的函数。
- 2， 资源占用极少（最少情况：FLASH:4K；SRAM:72B）。
- 3， 支持参数类型多（数字（包含 10/16 进制，支持负数）、字符串、函数指针等）。
- 4， 支持函数返回值显示。
- 5， 支持参数及返回值格式设置。
- 6， 支持函数执行时间计算（V3.1 及以后的版本新特性）。
- 7， 使用方便。

有了 USART，你可以轻易的修改函数参数、查看函数运行结果，从而快速解决问题。比如你调试一个摄像头模块，需要修改其中的几个参数来得到最佳的效果，普通的做法：写函数→修改参数→下载→看结果→不满意→修改参数→下载→看结果→不满意....不停的循环，直到满意为止。这样做很麻烦不说，单片机也是有寿命的啊，老这样不停的刷，很折寿的。而利用 USART，则只需要在串口调试助手里面输入函数及参数，然后直接串口发送给单片机，就执行了一次参数调整，不满意的话，你在串口调试助手修改参数在发送就可以了，直到你满意为止。这样，修改参数十分方便，不需要编译、不需要下载、不会让单片机折寿。

USART 支持的参数类型基本满足任何调试了，支持的类型有：10 或者 16 进制数字、字符串指针（如果该参数是用作参数返回的话，可能会有问题！）、函数指针等。因此绝大部分函数，可以直接被 USART 调用，对于不能直接调用的，你只需要重写一个函数，把影响调用的参数去掉即可，这个重写后的函数，即可以被 USART 调用了。

USART 的实现流程简单概括就是：第一步，添加需要调用的函数（在 usmart_config.c 里面的 usmart_nametab 数组里面添加）；第二步，初始化串口；第三步，初始化 USART（通过 usmart_init 函数实现）；第四步，轮询 usmart_scan 函数，处理串口数据。

接下来我们看下 USART 的组成，USART 组件总共包含 8 个文件，如图 26.1.1.所示：

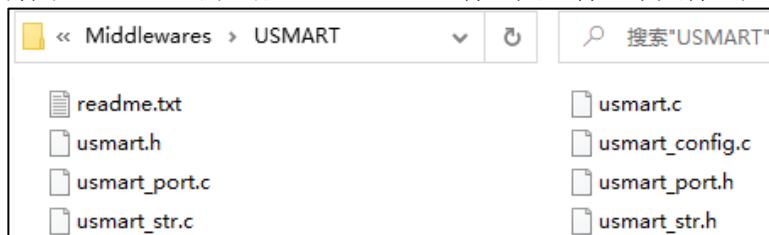


图 26.1.1 USART 组件代码

USART 每个文件的作用如表 26.1.1 所示：

| USMART 文件 | 说明 |
|-----------------|-------------------------------------|
| usmart.c | USMART 核心文件，用于处理命令以及对外交互 |
| usmart.h | USMART 核心文件头文件，定义结构体，宏定义、函数声明等 |
| usmart_str.c | USMART 字符串处理文件，用于字符串转换、参数获取等 |
| usmart_str.h | USMART 字符串处理头文件，用于函数声明 |
| usmart_port.c | USMART 移植文件，用于 USMART 移植 |
| usmart_port.h | USMART 移植头文件，定义用户配置参数、宏定义、函数声明等 |
| usmart_config.c | USMART 函数管理文件，用于添加用户需要 USMART 管理的函数 |
| readme.txt | USMART 介绍文件，用于说明 USMART 版本和功能 |

表 26.1.1 USMART 文件介绍

经过以上简单介绍，我们对 USMART 有了个大概了解，接下来我们将在下一小节给大家介绍 USMART 组件的移植和使用。

26.2 硬件设计

1. 例程功能

本实验通过 usmart 调用单片机里面的函数，实现对 LCD 显示和 LED 以及延时的控制。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
 - LED0 - PB5
 - LED1 - PE5
- 1) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 2) 定时器 4
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

26.3 程序设计

26.3.1 程序流程图

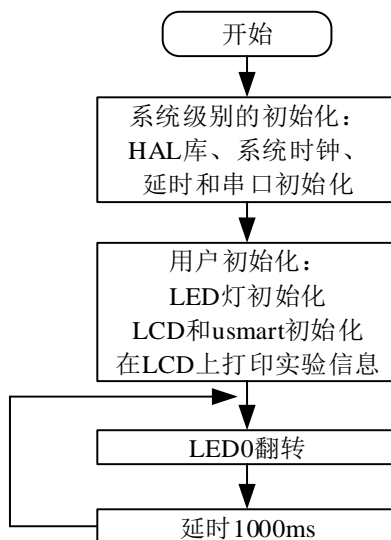


图 26.3.1.1 USMART 调试组件实验程序流程图

26.3.2 程序解析

1. USART 驱动代码

要使用 USART，我们先得进行代码移植，USART 的移植非常简单，我们只需要修改 usmart_port.c 里面的 5 个函数即可完成移植。

第一个函数，USART 输入数据流获取函数，该函数的实现代码如下：

```
/**
 * @brief      获取输入数据流 (字符串)
 * @note       USART 通过解析该函数返回的字符串以获取函数名及参数等信息
 * @param      无
 * @retval     0, 没有接收到数据
 *             其他, 数据流首地址 (不能是 0)
 */
char *usmart_get_input_string(void)
{
    uint8_t len;
    char *pbuf = 0;

    if (g_usart_rx_sta & 0x8000) /* 串口接收完成? */
    {
        len = g_usart_rx_sta & 0x3fff; /* 得到此次接收到的数据长度 */
        g_usart_rx_buf[len] = '\0'; /* 在末尾加入结束符. */
        pbuf = (char*)g_usart_rx_buf;
        g_usart_rx_sta = 0; /* 开启下一次接收 */
    }

    return pbuf;
}
```

该函数通过 SYSTEM 文件夹默认的串口接收来实现输入数据流获取。SYSTEM 文件夹里面的串口接收函数，最大可以一次接收 200 字节，用于从串口接收函数名和参数等。大家如果在其他平台移植，请参考 SYSTEM 文件夹串口接收的实现方式进行移植（详细接收原理请参考：串口通信实验）。

第二个是 usmart_timx_init 函数，该函数的实现代码如下：

```
/**
 * @brief      定时器初始化函数
 * @param      arr:自动重载值
 *             psc:定时器分频系数
 * @retval     无
 */
void usmart_timx_init(uint16_t arr, uint16_t psc)
{
    USART_TIMX_CLK_ENABLE();

    g_timx_usmart_handle.Instance = USART_TIMX; /* 通用定时器 4 */
    g_timx_usmart_handle.Init.Prescaler = psc; /* 分频系数 */
    g_timx_usmart_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 向上计数器 */
    g_timx_usmart_handle.Init.Period = arr; /* 自动装载值 */
    g_timx_usmart_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init(&g_timx_usmart_handle);
    HAL_TIM_Base_Start_IT(&g_timx_usmart_handle); /* 使能定时器和定时器中断 */
    HAL_NVIC_SetPriority(USART_TIMX_IRQn, 3, 3); /* 抢占优先级 3, 子优先级 3 */
    HAL_NVIC_EnableIRQ(USART_TIMX_IRQn); /* 开启 TIM 中断 */
}
```

该函数只有在：USART_ENTIMX_SCAN 值为 1 时，才需要实现，用于定时器初始化，利用定时器完成对 usmart_scan 函数的周期性调用，并实现函数运行时间计时（runtime）功能。该函数里面的：USART_TIMX_CLK_ENABLE、USART_TIMX 和 USART_TIMX_IRQn

等定时器相关宏定义我们在 usmart_prot.h 里面定义，方便大家修改。本实验我们用的 TIM4。

注意：usmart_timx_init 函数是在 usmart.c 里面，被 usmart_init 调用，usmart_init 源码如下：

```
/**
 * @brief      初始化 USMART
 * @param      tclk: 定时器的频率(单位:Mhz)
 * @retval     无
 */
void usmart_init(uint16_t tclk)
{
    #if USMART_ENTIMX_SCAN == 1
        usmart_timx_init(1000, tclk * 100 - 1);
    #endif
    usmart_dev.sptype = 1; /* 十六进制显示参数 */
}
```

该函数有一个参数 tclk，就是用于定时器初始化。注意：这里的 tclk 是指所选 TIM 的时钟频率，而非系统主频！这里我们用的 TIM4，对于 STM32F103 来说，TIM4 的时钟源来自 2 倍的 APB1，频率为：72Mhz。

另外，USMART_ENTIMX_SCAN 是在 usmart_port.h 里面定义的一个是否使能定时器中断扫描的宏定义。如果为 1，就初始化定时器中断，并在中断里面调用 usmart_scan 函数。如果为 0，那么需要用户需要自行间隔一定时间（100ms 左右为宜）调用一次 usmart_scan 函数，以实现串口数据处理。**注意：如果要使用函数执行时间统计功能（runtime 1），则必须设置 USMART_ENTIMX_SCAN 为 1。另外，为了让统计时间精确到 0.1ms，定时器的计数时钟频率必须设置为 10Khz，否则时间就不是 0.1ms 了。**

第三和第四个函数仅用于服务 USMART 的函数执行时间统计功能（串口指令：runtime 1），分别是：usmart_timx_reset_time 和 usmart_timx_get_time，这两个函数代码如下：

```
/**
 * @brief      复位 runtime
 * @note       需要根据所移植到的 MCU 的定时器参数进行修改
 * @param      无
 * @retval     无
 */
void usmart_timx_reset_time(void)
{
    __HAL_TIM_CLEAR_FLAG(&g_timx_usmart_handle, TIM_FLAG_UPDATE); /* 清中断标志 */
    __HAL_TIM_SET_AUTORELOAD(&g_timx_usmart_handle, 0xFFFF); /* 重载值设置最大 */
    __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0); /* 清定时器 CNT */
    usmart_dev.runtime = 0;
}

/**
 * @brief      获得 runtime 时间
 * @note       需要根据所移植到的 MCU 的定时器参数进行修改
 * @param      无
 * @retval     执行时间,单位:0.1ms,最大延时时间为定时器 CNT 值的 2 倍*0.1ms
 */
uint32_t usmart_timx_get_time(void)
{
    /* 在运行期间,产生了定时器溢出 */
    if (__HAL_TIM_GET_FLAG(&g_timx_usmart_handle, TIM_FLAG_UPDATE) == SET)
    {
        usmart_dev.runtime += 0xFFFF;
    }
    usmart_dev.runtime += __HAL_TIM_GET_COUNTER(&g_timx_usmart_handle);
    return usmart_dev.runtime; /* 返回计数值 */
}
```

usmart_timx_reset_time 函数在每次 USMART 调用函数之前执行，清除定时器的计数器，然后在函数执行完之后，调用 usmart_timx_get_time 获取计数器值，从而得到整个函数的运行时间。由于 usmart 调用的函数，都是在中断里面执行的，所以我们不太方便再用定时器的中断功

能来实现定时器溢出统计，因此，USMART 的函数执行时间统计功能，最多可以统计定时器溢出 1 次的时间，对 STM32F103 的 TIM4 来说，该定时器是 16 位的，最大计数是 65535，而由于我们定时器设置的是 0.1ms 一个计时周期（10Khz），所以最长计时时间是：

$$65535 * 2 * 0.1\text{ms} = 13.1 \text{ 秒}$$

也就是说，如果函数执行时间超过 13.1 秒，那么计时将不准确。

最后一个是 USMART_TIMX_IRQHandler 函数，该函数的实现代码如下：

```
/**
 * @brief      USMART 定时器中断服务函数
 * @param      无
 * @retval     无
 */
void USMART_TIMX_IRQHandler(void)
{
    /* 溢出中断 */
    if(__HAL_TIM_GET_IT_SOURCE(&g_timx_usmart_handle,TIM_IT_UPDATE)==SET)
    {
        usmart_dev.scan(); /* usmart 扫描 */
        __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0); /* 清定时器 CNT */
        __HAL_TIM_SET_AUTORELOAD(&g_timx_usmart_handle, 100); /* 恢复原来的设置 */
    }

    __HAL_TIM_CLEAR_IT(&g_timx_usmart_handle, TIM_IT_UPDATE); /* 清除中断标志位 */
}
```

该函数是定时器 TIMX 的中断服务函数，也是一个宏定义函数，同样是在 usmart_port.h 里面定义，方便大家修改。该函数主要用于周期性调用 usmart 扫描函数（实际函数：usmart_scan），完成对输入数据流的处理。同时，清除定时器的 CNT 值，并设置自动重装载值。

完成这几个函数的移植，就可以使用 USMART 了。不过，需要注意的是，usmart 同外部的互交，一般是通过 usmart_dev 结构体实现，所以 usmart_init 和 usmart_scan 的调用分别是通过：usmart_dev.init 和 usmart_dev.scan 实现的。

外我们还需要在 usmart_config.c 文件里面添加想要被 USMART 调用的函数。打开 usmart_config.c 文件，如图 26.3.2.1 所示：

```
/**
 * 用户配置区
 * 这下面要包含所用到的函数所声明的头文件(用户自己添加)
 */
#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LCD/lcd.h" /* 函数所在头文件添加区 */

extern void led_set(uint8_t sta);
extern void test_fun(void(*ledset)(uint8_t), uint8_t sta);

/* 函数名列表初始化(用户自己添加)
 * 用户直接在这里输入要执行的函数名及其查找串
 */
struct _m_usmart_nametab usmart_nametab[] =
{
    #if USMART_USE_WRFUNS == 1 /* 如果使能了读写操作 */
        (void *)read_addr, "uint32_t read_addr(uint32_t addr)",
        (void *)write_addr, "void write_addr(uint32_t addr,uint32_t val)",
    #endif
        (void *)delay_ms, "void delay_ms(uint16_t nms)",
        (void *)delay_us, "void delay_us(uint32_t nus)", /* 用户函数添加区 */
}
```

图 26.3.2.1 添加需要被 USMART 调用的函数

这里的添加函数很简单，只要把函数所在头文件添加进来，并把函数名按上图所示的方式增加即可，默认我们添加了两个函数：delay_ms 和 delay_us。另外，read_addr 和 write_addr 属于 usmart 自带的函数，用于读写指定地址的数据，通过配置 USMART_USE_WRFUNS 宏定义，可以使能或者禁止这两个函数。

这里我们根据自己的需要按上图的格式添加其他函数，usmart_config.c 文件中我们已经添

加了 LCD 的相关函数，大家可以查看本实验的 usmart_config.c 文件，并可在串口助手中调用。具体的调用方法 26.4 小节有具体讲解。

2. main.c 代码

在 main 函数之前，我们添加了 led_set 和 test_fun 两个函数，代码如下：

```
/* LED 状态设置函数 */
void led_set(uint8_t sta)
{
    LED1(sta);
}

/* 函数参数调用测试函数 */
void test_fun(void(*ledset)(uint8_t), uint8_t sta)
{
    ledset(sta);
}
```

led_set 函数，用于设置 LED1 的状态，而第二个函数 test_fun 则是测试 USMART 对函数参数的支持的，test_fun 的第一个参数是函数，在 USMART 里面也是可以调用的。

main 函数代码如下：

```
int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USMART TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while(1)
    {
        LED0_TOGGLE(); /* LED0 (RED) 闪烁 */
        delay_ms(500);
    }
}
```

此部分代码功能如下：经过一系列初始化，显示使用信息后，就是在无限循环中 LED0 翻转延时，并等待串口数据。

26.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。同时，屏幕上显示了一些字符（就是主函数里面要显示的字符）。

我们打开串口调试助手 XCOM，选择正确的串口号→多条发送→勾选发送新行（即发送回车键）选项，然后发送 list 指令，即可打印所有 usmart 可调用函数。如下图所示：

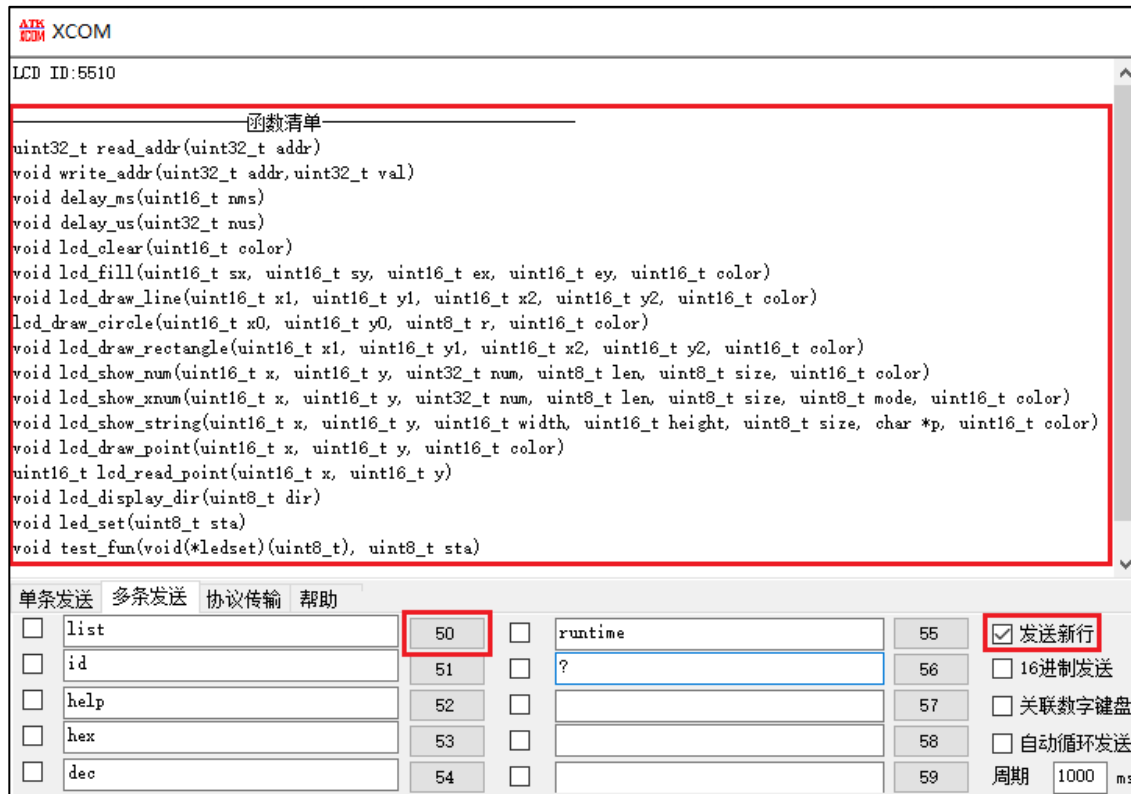


图 26.4.1 驱动串口调试助手

上图中 list、id、help、hex、dec、? 和 runtime 都属于 usmart 自带的系统命令，点击后方的数字按钮，即可发送对应的指令。下面我们简单介绍下这几个命令：

list，该命令用于打印所有 usmart 可调用函数。发送该命令后，串口将受到所有能被 usmart 调用得到函数，如图 26.4.1 所示。

id，该指令用于获取各个函数的入口地址。比如前面写的 test_fun 函数，就有一个函数参数，我们需要先通过 id 指令，获取 led_set 函数的 id（即入口地址），然后将这个 id 作为函数参数，传递给 test_fun。

help（或者“?”也可以），发送该指令后，串口将打印 usmart 使用的帮助信息。

hex 和 dec，这两个指令可以带参数，也可以不带参数。当不带参数的时候，hex 和 dec 分别用于设置串口显示数据格式为 16 进制/10 进制。当带参数的时候，hex 和 dec 就执行进制转换，比如输入：hex 1234，串口将打印：HEX:0X4D2，也就是将 1234 转换为 16 进制打印出来。又比如输入：dec 0X1234，串口将打印：DEC:4660，就是将 0X1234 转换为 10 进制打印出来。

runtime 指令，用于函数执行时间统计功能的开启和关闭，发送：runtime 1，可以开启函数执行时间统计功能；发送：runtime 0，可以关闭函数执行时间统计功能。函数执行时间统计功能，默认是关闭的。

大家可以亲自体验下这几个系统指令，不过要注意，所有的指令都是大小写敏感的，不要写错哦。

接下来，我们将介绍如何调用 list 所打印的这些函数，先来看一个简单的 delay_ms 的调用，我们分别输入 delay_ms(1000)和 delay_ms(0x3E8)，如图 26.4.2 所示：

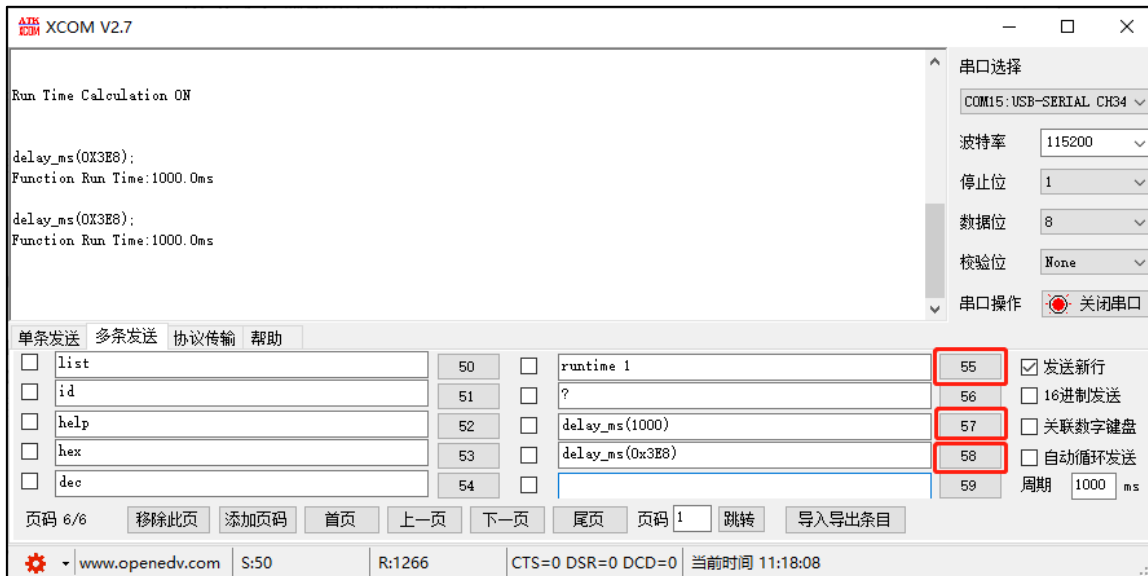


图 26.4.2 串口调用 delay_ms 函数

从上图可以看出, delay_ms(1000)和 delay_ms(0x3E8)的调用结果是一样的, 都是延时 1000ms, 因为 usmart 默认设置的是 hex 显示, 所以看到串口打印的参数都是 16 进制格式的, 大家可以通过发送 dec 指令切换为十进制显示。另外, 由于 USMART 对调用函数的参数大小写不敏感, 所以参数写成: 0X3E8 或者 0x3e8 都是正确的。另外, 发送: runtime 1, 开启运行时间统计功能, 从测试结果看, USMART 的函数运行时间统计功能, 是相当准确的。

我们再看另外一个函数, lcd_show_string 函数, 该函数用于显示字符串, 我们通过串口输入: lcd_show_string(30,200,200,16,16," This is a test for usmart!!", 0xF800), 如图 26.4.3 所示:

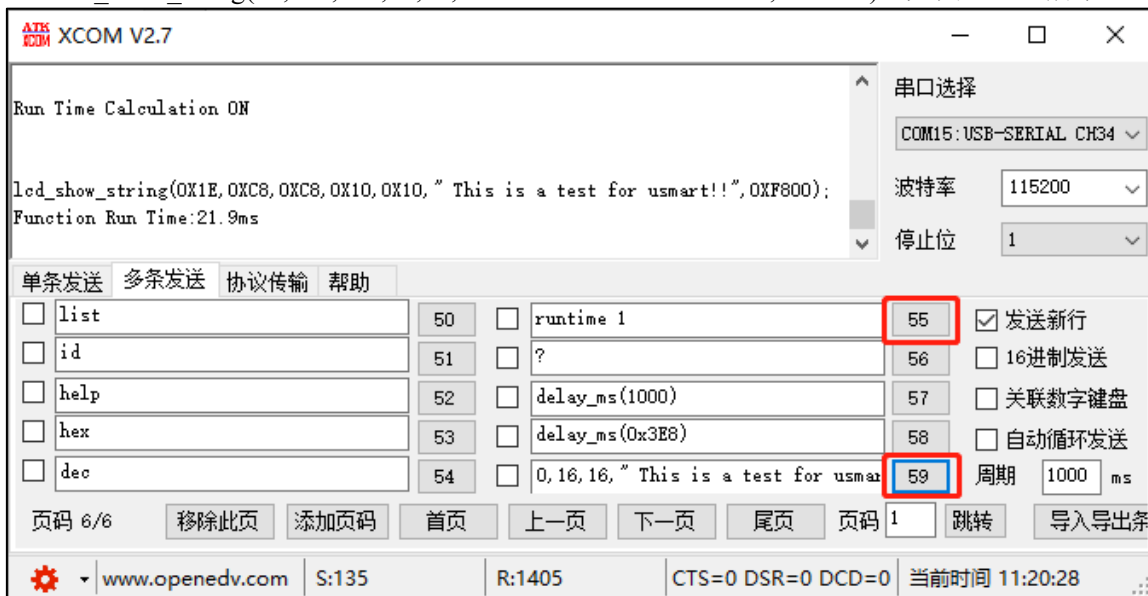


图 26.4.3 串口调用 lcd_show_string 函数

该函数用于在指定区域, 显示指定字符串, 发送给开发板后, 我们可以看到 LCD 在我们指定的地方显示了: This is a test for usmart!! 这个字符串。

其他函数的调用, 也都是一样的方法, 这里我们就不多介绍了, 最后说一下带有函数参数的函数的调用。我们将 led_set 函数作为 test_fun 的参数, 通过在 test_fun 里面调用 led_set 函数, 实现对 LED1 的控制。前面说过, 我们要调用带有函数参数的函数, 就必须先得到函数参数的入口地址(id), 通过输入 id 指令, 我们可以得到 led_set 的函数入口地址是: 0X0800321D, 所以, 我们在串口输入: test_fun(0X0800321D,0), 就可以控制 LED1 亮了。如图 26.4.4 所示:

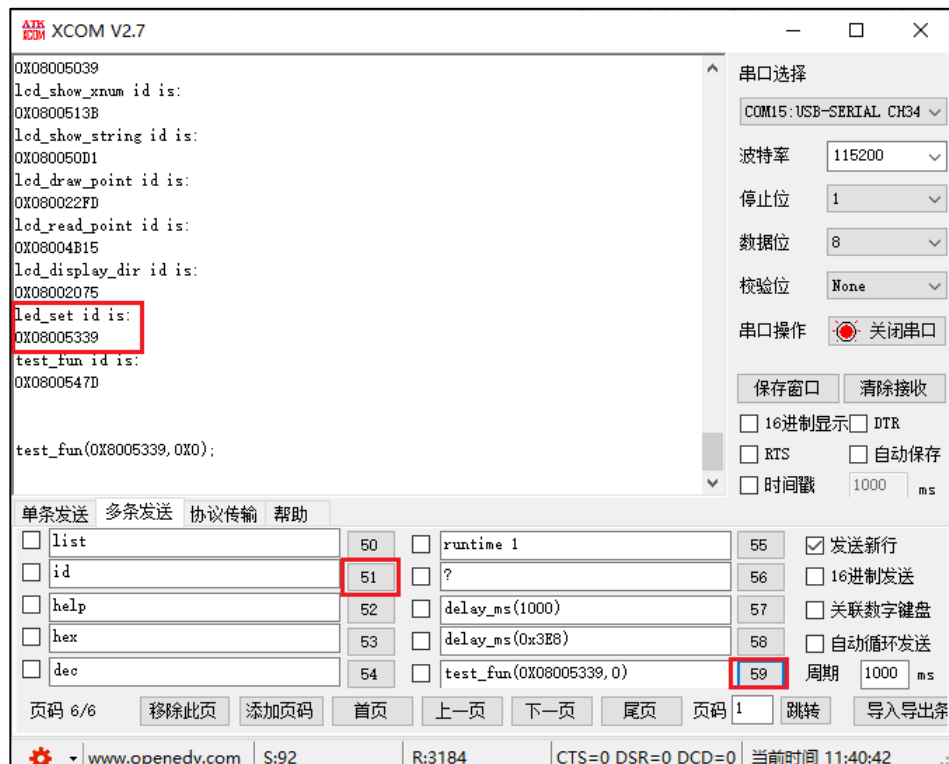


图 26.4.4 串口调用 test_fun 函数

在开发板上，我们可以看到，收到串口发送的 test_fun(0X0800321D,0)后，开发板的 LED1 亮了，然后大家可以通过发送 test_fun(0X0800321D,1)，来关闭 LED1。说明我们成功的通过 test_fun 函数调用 led_set，实现了对 LED1 的控制。也就验证了 USART 对函数参数的支持。

USART 调试组件的使用，就为大家介绍到这里。USART 是一个非常不错的调试组件，希望大家能学会使用，可以达到事半功倍的效果。

第二十七章 RTC 实时时钟实验

本章，我们将介绍 STM32F103 的内部实时时钟（RTC）。我们将使用 LCD 模块来显示日期和时间，实现一个简单的实时时钟，并可以设置闹铃，另外还将介绍 BKP 的使用。

本章分为如下几个小节：

27.1 RTC 时钟简介

27.2 硬件设计

27.3 程序设计

27.4 下载验证

27.1 RTC 时钟简介

STM32F103 的实时时钟（RTC）是一个独立的定时器。STM32 的 RTC 模块拥有一组连续计数的计数器，在相对应的软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统的当前时间和日期。

RTC 模块和时钟配置系统（RCC_BDCR 寄存器）是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变，只要后备区域供电正常，那么 RTC 将可以一直运行。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域（BKP）写保护。

27.1.1 RTC 框图

下面先来学习 RTC 框图，通过学习 RTC 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。RTC 的框图，如图 27.1.1 所示：

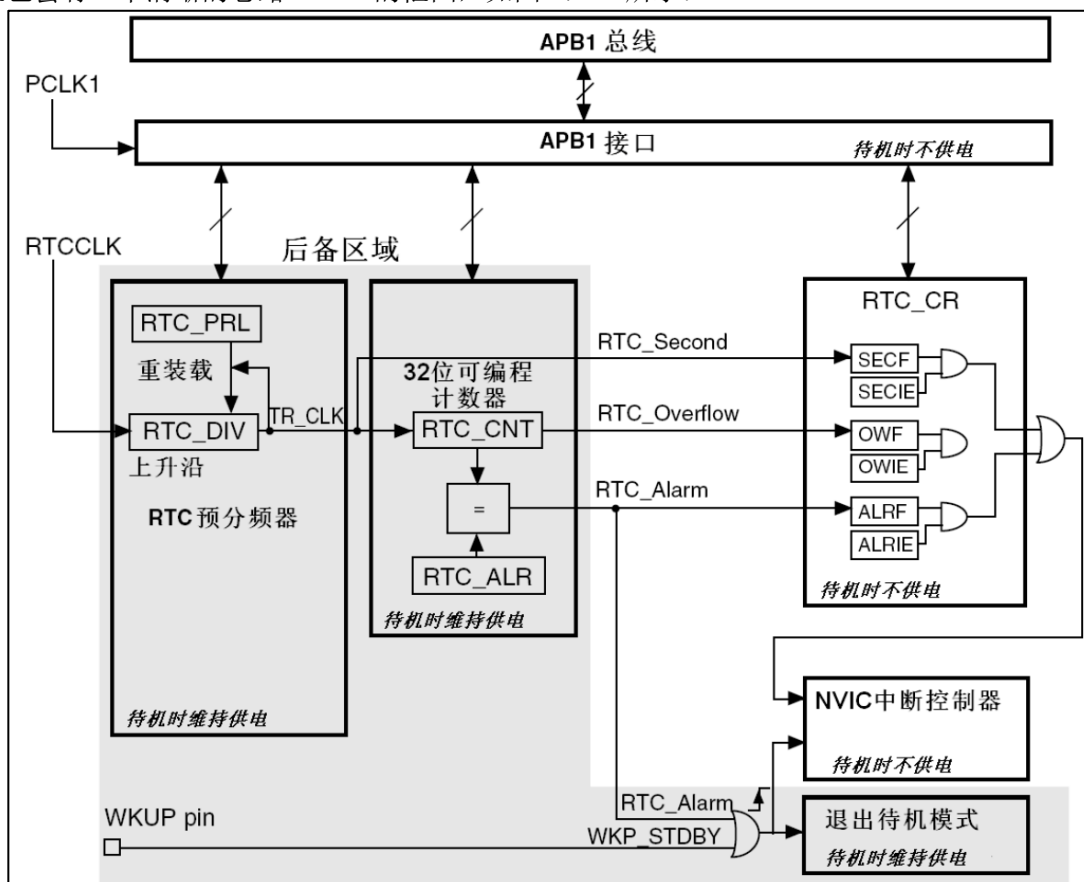


图 27.1.1 RTC 框图

我们在讲解 RTC 架构之前，说明一下框图中浅灰色的部分，他们是属于备份域的，在 VDD 掉电时可在 VBAT 的驱动下继续工作，这部分包括 RTC 的分频器，计数器以及闹钟控制器。在寄存器部分才展开解释一下备用域。下面把 RTC 框图分成以下 2 个部分讲解：

① **APB1 接口**：用来和 APB1 总线相连。通过 APB1 总线可以访问 RTC 相关的寄存器，对其进行读写操作。

② **RTC 核心**：由一组可编程计数器组成，主要分成两个模块。第一个模块是 RTC 的预分频模块，它可编程产生 1 秒的 RTC 时间基准 TR_CLK。RTC 的预分频模块包括了一个 20 位的可编程分频器(RTC 预分频器)。如果在 RTC_CR 寄存器中设置相对应的允许位，则在每个 TR_CLK 周期中 RTC 产生一个中断(秒中断)。第二个模块是一个 32 位的可编程计数器，可被初始化为当前的系统时间，一个 32 位的时钟计数器，按秒钟计算，可以记录 4294967296 秒，约合 136 年左右，作为一般应用足够了。

RTC 还有一个闹钟寄存器 RTC_ALR，用于产生闹钟。系统时间按 TR_CLK 周期累加并与存储在 RTC_ALR 寄存器中的可编程时间相比较，如果 RTC_CR 控制寄存器中设置了相应允许位，比较匹配时将产生一个闹钟中断。

由于备份域的存在，所以 RTC 内核可以完全独立于 RTC APB1 接口。而软件是通过 APB1 接口访问 RTC 的预分频值、计数器值和闹钟值的。但是相关可读寄存器只在 RTC APB1 时钟进行重新同步的 RTC 时钟的上升沿被更新，RTC 标志也是如此。这就意味着，如果 APB1 接口刚刚被开启之后，在第一次的内部寄存器更新之前，从 APB1 上读取的 RTC 寄存器值可能被破坏了（通常读到 0）。因此，若在读取 RTC 寄存器曾经被禁止的 RTC APB1 接口，软件首先必须等待 RTC_CRL 寄存器的 RSF 位（寄存器同步标志位，bit3）被硬件置 1。

27.1.2 RTC 寄存器

接下来，我们介绍本实验我们要用到的 RTC 寄存器。

● RTC 控制寄存器（RTC_CRH/CRL）

RTC 控制寄存器共有两个控制寄存器 RTC_CRH 和 RTC_CRL，两个都是 16 位的。

RTC 控制寄存器高位 **RTC_CRH**，描述如图 27.1.2.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|---|----|----|----|---|---|---|---|---|---|---|------|-------|-------|
| 保留 | | | | | | | | | | | | | OWIE | ALRIE | SECIE |
| | | | | | | | | | | | | | r/w | r/w | r/w |
| 位2 | | OWIE ：允许溢出中断位 (Overflow interrupt enable) 0：屏蔽(不允许)溢出中断 1：允许溢出中断 | | | | | | | | | | | | | |
| 位1 | | ALRIE ：允许闹钟中断 (Alarm interrupt enable) 0：屏蔽(不允许)闹钟中断 1：允许闹钟中断 | | | | | | | | | | | | | |
| 位0 | | SECIE ：允许秒中断 (Second interrupt enable) 0：屏蔽(不允许)秒中断 1：允许秒中断 | | | | | | | | | | | | | |

图 27.1.2.1 RTC_CRH 寄存器

该寄存器是 RTC 控制寄存器高位，本章将用到秒钟中断，所以在该寄存器必须设置最低位为 1，以允许秒钟中断。

RTC 控制寄存器低位 **RTC_CRL**，描述如图 27.1.2.2 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--------------|----|---|----|----|----|---|---|---|---|-------|-----|-------|-------|-------|-------|
| 保留 | | | | | | | | | | RTOFF | CNF | RSF | OWF | ALRF | SECF |
| | | | | | | | | | | r | rw | rc w0 | rc w0 | rc w0 | rc w0 |
| 保留, 被硬件强制为0。 | | | | | | | | | | | | | | | |
| 位15:6 | | | | | | | | | | | | | | | |
| 位5 | | RTOFF: RTC操作关闭 (RTC operation OFF) RTC模块利用这位来指示对其寄存器进行的最后一次操作的状态, 指示操作是否完成。若此位为'0', 则表示无法对任何的RTC寄存器进行写操作。此位为只读位。 0: 上一次对RTC寄存器的写操作仍在进行; 1: 上一次对RTC寄存器的写操作已经完成。 | | | | | | | | | | | | | |
| 位4 | | CNF: 配置标志 (Configuration flag) 此位必须由软件置'1'以进入配置模式, 从而允许向RTC_CNT、RTC_ALR或RTC_PRL寄存器写入数据。只有当此位在被置'1'并重新由软件清'0'后, 才会执行写操作。 0: 退出配置模式(开始更新RTC寄存器); 1: 进入配置模式。 | | | | | | | | | | | | | |
| 位3 | | RSF: 寄存器同步标志 (Registers synchronized flag) 每当RTC_CNT寄存器和RTC_DIV寄存器由软件更新或清'0'时, 此位由硬件置'1'。在APB1复位后, 或APB1时钟停止后, 此位必须由软件清'0'。要进行任何的读操作之前, 用户程序必须等待这位被硬件置'1', 以确保RTC_CNT、RTC_ALR或RTC_PRL已经被同步。 0: 寄存器尚未被同步; 1: 寄存器已经被同步。 | | | | | | | | | | | | | |
| 位2 | | OWF: 溢出标志 (Overflow flag) 当32位可编程计数器溢出时, 此位由硬件置'1'。如果RTC_CRH寄存器中OWIE=1, 则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0: 无溢出; 1: 32位可编程计数器溢出。 | | | | | | | | | | | | | |
| 位1 | | ALRF: 闹钟标志 (Alarm flag) 当32位可编程计数器达到RTC_ALR寄存器所设置的预定值, 此位由硬件置'1'。如果RTC_CRH寄存器中ALRIE=1, 则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0: 无闹钟; 1: 有闹钟。 | | | | | | | | | | | | | |
| 位0 | | SECF: 秒标志 (Second flag) 当32位可编程预分频器溢出时, 此位由硬件置'1'同时RTC计数器加1。因此, 此标志为分辨率可编程的RTC计数器提供一个周期性的信号(通常为1秒)。如果RTC_CRH寄存器中SECIE=1, 则产生中断。此位只能由软件清除。对此位写'1'是无效的。 0: 秒标志条件不成立; 1: 秒标志条件成立。 | | | | | | | | | | | | | |

图 27.1.2.2 RTC_CRL 寄存器

该寄存器是 RTC 控制寄存器低位, 本章我们用到的是该寄存器的 0, 3~5 这几个位, 第 0 位是秒钟标志位, 我们在进入闹钟中断的时候, 通过判断这位来决定是不是发生了秒钟中断。然后必须通过软件将该位清零 (写零)。第 3 位为寄存器同步标志位, 我们在修改控制寄存器 RTC_CRH/RTC_CRL 之前, 必须先判断该位, 是否已经同步了, 如果没有则需要等待同步, 在没同步的情况下修改 RTC_CRH/RTC_CRL 的值是不行的。第 4 位为配置标志位, 在软件修改 RTC_CNT/RTC_PRL 的值的时候, 必须先软件置位该位, 以允许进入配置模式。第 5 位为 RTC 操作位, 该位由硬件操作, 软件只读。通过该位可以判断上次对 RTC 寄存器的操作是否完成, 如果没有, 我们必须等待上一次操作结束才能开始下一次操作。

● RTC 预分频装载寄存器 (RTC_PRLH/RTC_PRL)

RTC 预分频装载寄存器也是有两个寄存器组成, RTC_PRLH 和 RTC_PRL。这两个寄存器用来配置 RTC 时钟的分频数的, 比如我们使用外部 32.768K 的晶振作为时钟的输入频率, 那么我们要设置这两个寄存器的值为 32767, 得到一秒钟的计数频率。

RTC 预分频装载寄存器高位描述如图 27.1.2.3 所示:

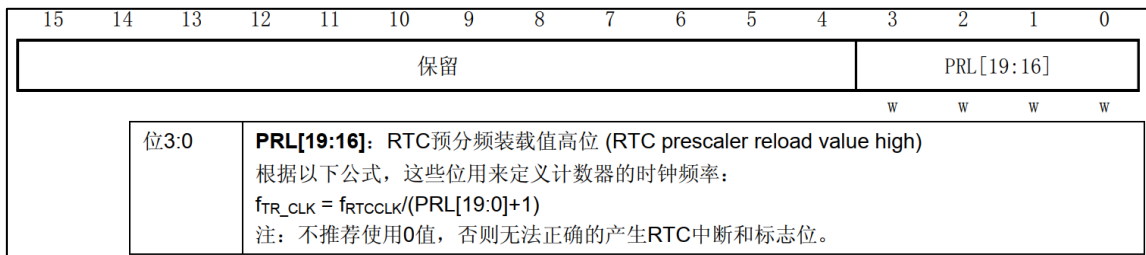


图 27.1.2.3 RTC_PRLH 寄存器

该寄存器是 RTC 预分频装载寄存器高位, 低四位有效用来存放 PRL 的 19~16 位。关于 PRL 的其余位存放在 RTC_PRLH 寄存器。

RTC 预分频装载寄存器低位描述如图 27.1.2.4 所示:

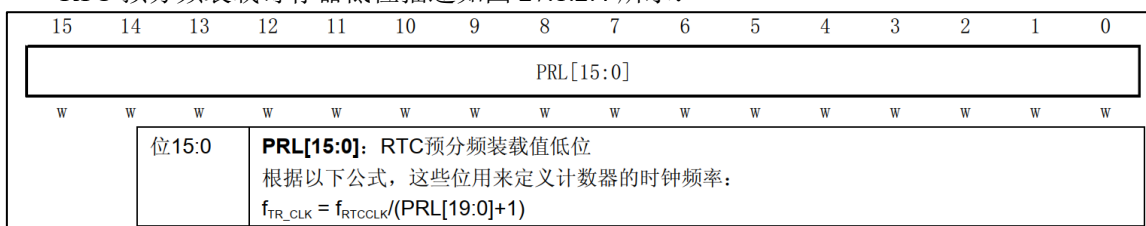


图 27.1.2.4 RTC_PRLH 寄存器

该寄存器是 RTC 预分频装载寄存器低位, 存放 RTC 预分频装载值低位。如果输入时钟是 32.768kHz, 这个预分频寄存器中写入 0x7FFF 可获得周期 1 秒钟的信号。

● RTC 预分频余数寄存器 (RTC_DIVH/RTC_DIVL)

RTC 预分频余数寄存器是用来获得比秒钟更加准确的时钟, 比如可以得到 0.1 秒, 甚至 0.01 秒等, 该寄存器的值自减的, 用于保存还需要多少时钟周期获得一个秒信号。在一次秒钟更新后, 由硬件重新装载。这两个寄存器和 RTC 预分频装载寄存器的各位是一样的, 这里我们就不列出来了。

● RTC 计数器寄存器 (RTC_CNTH/RTC_CNTL)

RTC 计数器寄存器 RTC_CNT, 由 2 个 16 位寄存器组成 RTC_CNTH 和 RTC_CNTL, 总共 32 位, 用来记录秒钟值。注意的是, 修改这两个寄存器的时候要先进入配置模式。RTC_CNT 描述如图 27.1.2.5 所示:

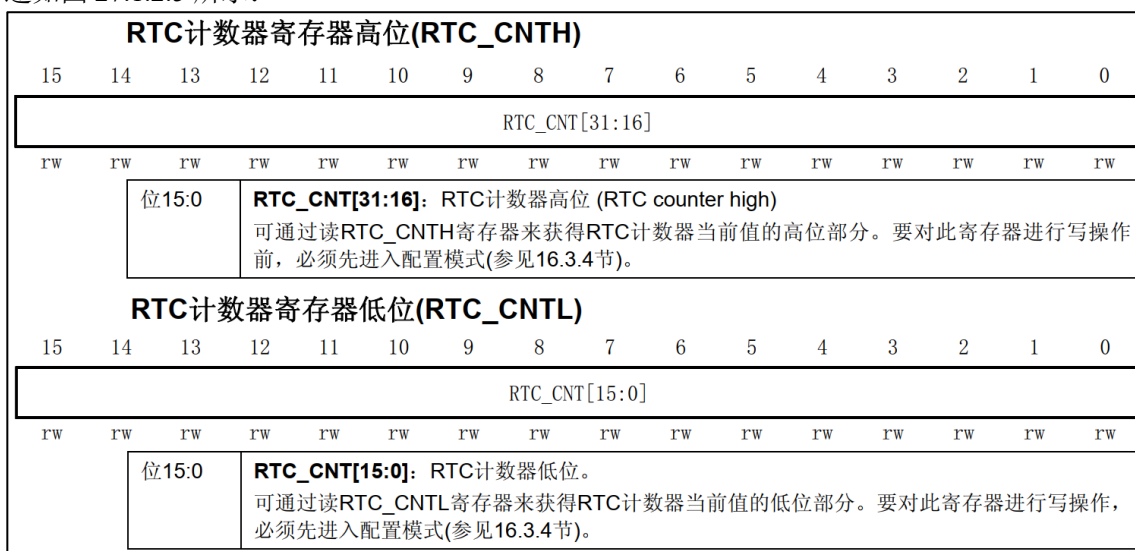


图 27.1.2.5 RTC_CNT 寄存器

● RTC 闹钟寄存器 (RTC_ALRH/RTC_ALRL)

RTC 闹钟寄存器, 该寄存器也是由 2 个 16 位的寄存器组成 RTC_ALRH 和 RTC_ALRL。总共 32 位, 用来标记闹钟产生的时间 (**以秒为单元**)。对于 STM32F1 系列的芯片来说, RTC 外设没有专门的年用日寄存器来分别存放这些信息, 全部日期信息以秒的格式存储在这两个寄存

器中，后面编程时会对时间进行特殊处理。如果 RTC_CNT 的值与 RTC_ALR 的值相等，并使能了中断的话，会产生一个闹钟中断。注意：该寄存器的修改也是要进入配置模式才能进行。

RTC 闹钟寄存器描述如图 27.1.2.6 所示：

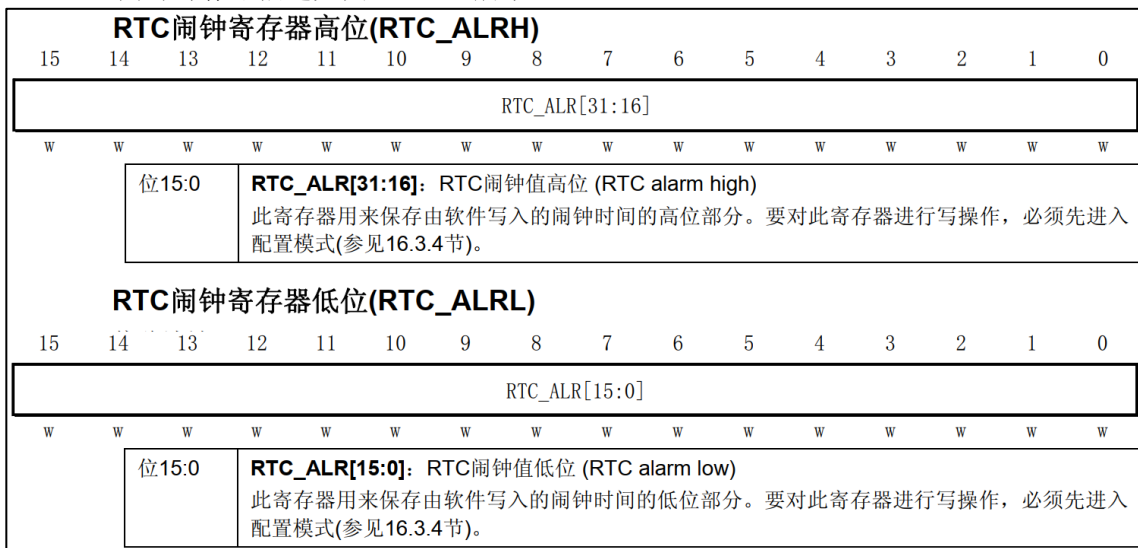


图 27.1.2.5 RTC_ALR 寄存器

● 备份数据寄存器 (BKP_DRx)

备份数据寄存器描述如图 27.1.2.6 所示。

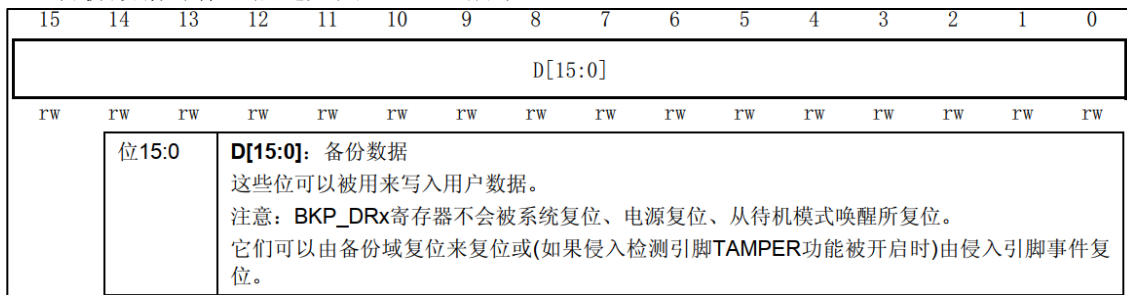


图 27.1.2.6 BKP_DRx 寄存器

该寄存器是一个 16 位寄存器，可以用来存储用户数据，可在 VDD 电源关闭时，通过 VBAT 保持上电状态。备份数据寄存器不会在系统复位、电源复位、从待机模式唤醒时复位。

那么在 MCU 复位后，对 RTC 和备份数据寄存器的写访问就被禁止，需要执行一下操作才可以对 RTC 及备份数据寄存器进行写访问：

- 1) 通过设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位来打开电源和后备接口时钟
- 2) 电源控制寄存器(PWR_CR)的 DBP 位来使能对后备寄存器和 RTC 访问

● 备份区域控制寄存器 (RCC_BDCR)

备份区域控制寄存器描述如图 25.1.2.7 所示。

| | | | | | | | | | | | | | | | |
|-----------|----|---|----|----|----|-------------|----|----|----|----|----|----|------------|------------|---------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | BDRST |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | rw 0 |
| RTC EN | 保留 | | | | | RTCSEL[1:0] | | 保留 | | | | | LSE BYP | LSE RDY | LSEON |
| rw | | | | | | rw | rw | | | | | | rw | r | rw |
| 位16 | | BDRST : 备份域软件复位 (Backup domain software reset) 由软件置'1'或清'0' 0: 复位未激活; 1: 复位整个备份域。 | | | | | | | | | | | | | |
| 位15 | | RTCEN : RTC时钟使能 (RTC clock enable) 由软件置'1'或清'0' 0: RTC时钟关闭; 1: RTC时钟开启。 | | | | | | | | | | | | | |
| 位9:8 | | RTCSEL[1:0] : RTC时钟源选择 (RTC clock source selection) 由软件设置来选择RTC时钟源。一旦RTC时钟源被选定，直到下次后备域被复位，它不能在改变。可通过设置BDRST位来清除。 00: 无时钟; 01: LSE振荡器作为RTC时钟; 10: LSI振荡器作为RTC时钟; 11: HSE振荡器在128分频后作为RTC时钟。 | | | | | | | | | | | | | |
| 位2 | | LSEBYP : 外部低速时钟振荡器旁路 (External low-speed oscillator bypass) 在调试模式下由软件置'1'或清'0'来旁路LSE。只有在外部32kHz振荡器关闭时，才能写入该位 0: LSE时钟未被旁路; 1: LSE时钟被旁路。 | | | | | | | | | | | | | |
| 位1 | | LSERDY : 外部低速LSE就绪 (External low-speed oscillator ready) 由硬件置'1'或清'0'来指示是否外部32kHz振荡器就绪。在LSEON被清零后，该位需要6个外部低速振荡器的周期才被清零。 0: 外部32kHz振荡器未就绪; 1: 外部32kHz振荡器就绪。 | | | | | | | | | | | | | |
| 位0 | | LSEON : 外部低速振荡器使能 (External low-speed oscillator enable) 由软件置'1'或清'0' 0: 外部32kHz振荡器关闭; 1: 外部32kHz振荡器开启。 | | | | | | | | | | | | | |

图 25.1.2.7 RCC_BDCR 寄存器

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的, 所以我们在 RTC 操作之前先要通过这个寄存器选择 RTC 的时钟源, 然后才能开始其他的操作。

27.2 硬件设计

1. 例程功能

本实验通过 LCD 显示 RTC 时间, 并可以通过 usmart 设置 RTC 时间, 从而调节时间, 或设置 RTC 闹钟, 还可以写入或者读取 RTC 后备区域 SRAM。LED0 闪烁, 提示程序运行。

2. 硬件资源

1) LED 灯

LED0 – PE5

LED1 – PB5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) RTC(实时时钟)

4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)

3. 原理图

RTC 属于 STM32F103 内部资源，通过软件设置好就可以了。不过 RTC 不能断电，否则数据就丢失了，我们如果能让时间在断电后还可以继续走，那么必须确保开发板的电池有电。

27.3 程序设计

27.3.1 RTC 的 HAL 库驱动

RTC 在 HAL 库中的驱动代码在 stm32f1xx_hal_rtc.c 文件（及其头文件）中。下面介绍几个重要的 RTC 函数，其他没有介绍的请看源码。

1. HAL_RTC_Init 函数

RTC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

- **函数描述：**

用于初始化 RTC。

- **函数形参：**

形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
    RTC_TypeDef                *Instance;        /* 寄存器基地址 */
    RTC_InitTypeDef            Init;             /* RTC 配置结构体 */
    RTC_DateTypeDef            DateToUpdate;     /* RTC 日期结构体 */
    HAL_LockTypeDef            Lock;             /* RTC 锁定对象 */
    __IO HAL_RTCStateTypeDef   State;           /* RTC 设备访问状态 */
}RTC_HandleTypeDef;
```

1) Instance: 指向 RTC 寄存器基地址。

2) Init: 是真正的 RTC 初始化结构体，其结构体类型 RTC_InitTypeDef 定义如下：

```
typedef struct
{
    uint32_t AsynchPrediv;    /* 异步预分频系数 */
    uint32_t OutPut;          /* 选择连接到 RTC_ALARM 输出的标志 */
}RTC_InitTypeDef;
```

AsynchPrediv 用来设置 RTC 的异步预分频系数，也就是设置两个预分频重载寄存器的相关位，因为异步预分频系数是 19 位，所以最大值为 0x7FFFF，不能超过这个值。

OutPut 用来选择 RTC 输出到 Tamper 引脚的信号，取值为: RTC_OUTPUTSOURCE_NONE（没有输出），RTC_OUTPUTSOURCE_CALIBCLOCK（RTC 时钟经过 64 分频输出到 TAMPER），RTC_OUTPUTSOURCE_ALARM（闹钟脉冲信号输出）和 RTC_OUTPUTSOURCE_SECOND（秒脉冲信号输出）。本实验选择没有输出，不配置即默认值，RTC_OUTPUTSOURCE_NONE。

3) DateToUpdate: 日期结构体。

4) Lock: 用于配置锁状态。

5) State: RTC 设备访问状态。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

注意：实验中没有使用 HAL 库自带的设置 RTC 时间的函数 HAL_RTC_SetTime、设置 RTC 日期的函数 HAL_RTC_SetDate、获取当前 RTC 日期的函数 HAL_RTC_GetTime。原因在于版本的 HAL 库函数不满足我们同时更新年月日时分秒的要求，且在实测中发现写时间会覆盖日期，写日期亦然，所以我们直接通过操作寄存器的方式去编写功能更加全面的函数。

RTC 配置步骤

1) 使能电源时钟，并使能 RTC 及 RTC 后备寄存器写访问。

我们要访问 RTC 和 RTC 备份区域就必须先使能电源时钟，然后使能 RTC 即后备区域访问。电源时钟使能，通过 RCC_APB1ENR 寄存器来设置；RTC 及 RTC 备份寄存器的写访问，

通过 PWR_CR 寄存器的 DBP 位设置。HAL 库设置方法为:

```
_HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 PWR */
_HAL_RCC_BKP_CLK_ENABLE(); /* 使能备份时钟 */
HAL_PWR_EnableBkUpAccess(); /* 取消备份区域写保护 */
```

2) 开启外部低速振荡器 LSE, 选择 RTC 时钟, 并使能。

调用 HAL_RCC_OscConfig 函数配置开启 LSE。

调用 HAL_RCCEx_PeriphCLKConfig 函数选择 RTC 时钟源。

使能 RTC 时钟函数为 _HAL_RCC_RTC_ENABLE。

3) 初始化 RTC, 设置 RTC 的分频, 以及配置 RTC 参数。

在 HAL 中, 通过函数 HAL_RTC_Init 函数配置 RTC 分频系数, 以及 RTC 的工作参数。

注意: 该函数会调用 HAL_RTC_MspInit 函数来完成对 RTC 的底层初始化, 包括: RTC 时钟使能, 时钟源选择等。

4) 设置 RTC 的日期和时间。

根据我们前面的说明, 我们使用操作寄存器的方式重新定义了设置 RTC 日期和时间的函数 rtc_set_time, 使用该函数就可以设置年月日时分秒。

5) 获取 RTC 当前日期和时间。

同样的, 获取 RTC 当前日期和时间的函数 rtc_get_time, 我们也是直接重新定义。该函数不直接返回时间, 而是把时间保存在我们定义的时间结构体里。

通过以上 5 个步骤, 我们就完成了对 RTC 的配置, RTC 即可正常工作, 这些操作不是每次上电都必须执行的, 视情况而定。我们还可以设置闹钟, 这些将在后面介绍。

27.3.2 程序流程图

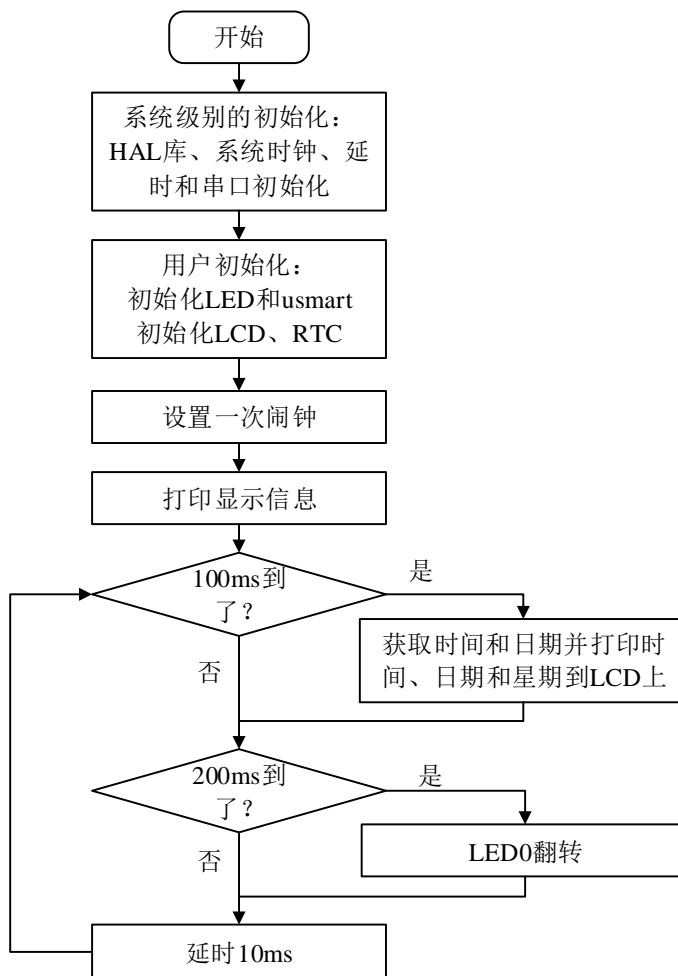


图 27.3.2.1 RTC 实时时钟实验程序流程图

27.3.3 程序解析

1. RTC 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RTC 驱动源码包括两个文件：rtc.c 和 rtc.h。

限于篇幅，rtc.c 中的代码，我们不全部贴出了，只针对几个重要的函数进行介绍。

rtc.h 头文件只有函数的声明，下面我们直接介绍 rtc.c 的程序，先看 RTC 的初始化函数，其定义如下：

```
/**
 * @brief      RTC 初始化
 * @note      默认尝试使用 LSE, 当 LSE 启动失败后, 切换为 LSI.
 *            通过 BKP 寄存器 0 的值, 可以判断 RTC 使用的是 LSE/LSI:
 *            当 BKP0==0X5050 时, 使用的是 LSE
 *            当 BKP0==0X5051 时, 使用的是 LSI
 *            注意: 切换 LSI/LSE 将导致时间/日期丢失, 切换后需重新设置.
 *
 * @param      无
 * @retval     0, 成功
 *            1, 进入初始化模式失败
 */
uint8_t rtc_init(void)
{
    /* 检查是不是第一次配置时钟 */
    uint16_t bkpflag = 0;

    __HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 */
    __HAL_RCC_BKP_CLK_ENABLE(); /* 使能备份时钟 */
    HAL_PWR_EnableBkUpAccess(); /* 取消备份区写保护 */

    bkpflag = rtc_read_bkr(0); /* 读取 BKP0 的值 */

    g_rtc_handle.Instance = RTC;
    /* 时钟周期设置, 理论值: 32767, 这里也可以用 RTC_AUTO_1_SECOND */
    g_rtc_handle.Init.AsynchPrediv = 32767;
    if (HAL_RTC_Init(&g_rtc_handle) != HAL_OK)
    {
        return 1;
    }

    /* 之前未初始化过, 重新配置 */
    if ((bkpflag != 0x5050) && (bkpflag != 0x5051))
    {
        rtc_set_time(2020, 4, 26, 9, 22, 35); /* 设置时间 */
    }

    __HAL_RTC_ALARM_ENABLE_IT(&g_rtc_handle, RTC_IT_SEC); /* 允许秒中断 */

    HAL_NVIC_SetPriority(RTC_IRQn, 0x2, 0); /* 优先级设置 */
    HAL_NVIC_EnableIRQ(RTC_IRQn); /* 使能 RTC 中断通道 */

    rtc_get_time(); /* 更新时间 */
    return 0;
}
```

该函数用来初始化 RTC 配置以及日期和时钟，但是只在第一次的时候设置时间，以后如果重新上电/复位都不会再进行时间设置了（前提是备份电池有电）。在第一次配置的时候，我们是按照上面介绍的 RTC 初始化步骤调用函数 HAL_RTC_Init 来实现的。

我们通过读取 BKP 寄存器 0 的值来判断是否需要设置时间，对 BKP 寄存器 0 的写操作是在 HAL_RTC_MspInit 回调函数中实现，下面会讲。第一次未对 RTC 进行初始化 BKP 寄

寄存器 0 的值非 0x5050 非 0x5051，当进行 RTC 初始化时，BKP 寄存器 0 的值就是 0x5050 或 0x5051，所以以上代码操作确保时间只会设置一次，复位时不会重新设置时间。电池正常供电时，我们设置的时间不会因复位或者断电而丢失。

读取后备寄存器的函数其实还是调用 HAL 库提供的函数接口，写后备寄存器函数同样也是。这两个函数如下：

```
uint32_t HAL_RTCEx_BKUPRead(RTC_HandleTypeDef *hrtc, uint32_t BackupRegister);
void HAL_RTCEx_BKUPWrite(RTC_HandleTypeDef *hrtc, uint32_t BackupRegister,
                          uint32_t Data);
```

这两个函数的使用方法就非常简单，分别用来读和写 BKR 寄存器的值。这里我们只是略微点到为止，详看例程源码。

这里设置时间和日期，是通过 rtc_set_time 函数来实现的，我们之所以不是用 HAL 库自带的设置时间和日期的函数，在前面已经提到了这个原因，就不用多说了。那么 rtc_set_time 是我们直接操作寄存器，同时，它也可以为我们的 USART 所调用，十分方便我们调试时候使用。

接下来，我们用 HAL_RTC_MspInit 函数来编写 RTC 时钟配置等代码，其定义如下：

```
void HAL_RTC_MspInit(RTC_HandleTypeDef *hrtc)
{
    uint16_t retry = 200;

    __HAL_RCC_RTC_ENABLE(); /* RTC 时钟使能 */

    RCC_OscInitTypeDef rcc_oscinitstruct;
    RCC_PeriphCLKInitTypeDef rcc_periphclkinitstruct;

    /* 使用寄存器的方式去检测 LSE 是否可以正常工作 */
    RCC->BDCR |= 1 << 0; /* 开启外部低速振荡器 LSE */

    while (retry && ((RCC->BDCR & 0X02) == 0)) /* 等待 LSE 准备好 */
    {
        retry--;
        delay_ms(5);
    }

    if (retry == 0) /* LSE 起振失败 使用 LSI */
    {
        /* 选择要配置的振荡器 */
        rcc_oscinitstruct.OscillatorType = RCC_OSCILLATORTYPE_LSI;
        rcc_oscinitstruct.LSEState = RCC_LSI_ON; /* LSI 状态: 开启 */
        rcc_oscinitstruct.PLL.PLLState = RCC_PLL_NONE; /* PLL 无配置 */
        HAL_RCC_OscConfig(&rcc_oscinitstruct);

        /* 选择要配置的外设 RTC */
        rcc_periphclkinitstruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
        /* RTC 时钟源选择 LSI */
        rcc_periphclkinitstruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSI;
        HAL_RCCEx_PeriphCLKConfig(&rcc_periphclkinitstruct);
        rtc_write_bkr(0, 0X5051);
    }
    else
    {
        rcc_oscinitstruct.OscillatorType = RCC_OSCILLATORTYPE_LSE;
        rcc_oscinitstruct.LSEState = RCC_LSE_ON; /* LSE 状态: 开启 */
        rcc_oscinitstruct.PLL.PLLState = RCC_PLL_NONE; /* PLL 不配置 */

        rcc_periphclkinitstruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
        rcc_periphclkinitstruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
        HAL_RCCEx_PeriphCLKConfig(&rcc_periphclkinitstruct);
        rtc_write_bkr(0, 0X5050);
    }
}
```

介绍完 RTC 初始化相关函数后，我们来介绍一下 rtc_set_time 函数，代码如下：

```
/**
```



```

* @brief      设置时间，包括年月日时分秒
* @note      以 1970 年 1 月 1 日为基准，往后累加时间
*            合法年份范围为：1970 ~ 2105 年
*            HAL 默认为年份起点为 2000 年
* @param      syear : 年份
* @param      smon  : 月份
* @param      sday  : 日期
* @param      hour  : 小时
* @param      min   : 分钟
* @param      sec   : 秒钟
* @retval     0, 成功; 1, 失败;
*/
uint8_t rtc_set_time(uint16_t syear, uint8_t smon, uint8_t sday, uint8_t hour,
uint8_t min, uint8_t sec)
{
    uint32_t seccount = 0;

    /* 将年月日时分秒转换成总秒钟数 */
    seccount = rtc_date2sec(syear, smon, sday, hour, min, sec);

    __HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 */
    __HAL_RCC_BKP_CLK_ENABLE(); /* 使能备份域时钟 */
    HAL_PWR_EnableBkUpAccess(); /* 取消备份域写保护 */
    /* 上面三步是必须的! */

    RTC->CRL |= 1 << 4; /* 进入配置模式 */

    RTC->CNTL = seccount & 0xffff;
    RTC->CNTH = seccount >> 16;

    RTC->CRL &= ~(1 << 4); /* 退出配置模式 */

    /* 等待 RTC 寄存器操作完成，即等待 RTOFF == 1 */
    while (!__HAL_RTC_ALARM_GET_FLAG(&g_rtc_handle, RTC_FLAG_RTOFF));

    return 0;
}

```

该函数用于设置时间，把我们输入的时间，转换为以 1970 年 1 月 1 日 0 时 0 分 0 秒做起始时间的秒钟信号，后续的计算都以这个时间为基准，由于 STM32 的秒钟计数器可以保存 136 年的秒钟数据，这样我们就可以计时到 2106 年。

接着，我们介绍 `rtc_set_alarma` 函数，该函数用于设置闹钟时间，同 `rtc_set_time` 函数几乎一模一样，主要区别：就是将 `RTC->CNTL` 和 `RTC->CNTH` 换成了 `RTC->ALRL` 和 `RTC->ALRH`，用于设置闹钟时间。RTC 其实是有闹钟中断的，我们这里并没有用到，本实验用到了秒中断，所以在秒中断里顺带处理闹钟中断的事情。具体代码请参考本例程源码。

特别提醒：假如只是使用 HAL 库的 `__HAL_RTC_ALARM_ENABLE_IT` 函数来使能闹钟中断，但是没有设置闹钟相关的 NVIC 和 EXTI，实际上不会产生闹钟中断，只会产生闹钟标志（`RTC->CRL` 的 `ALRL` 置位）。可以通过读取闹钟标志来判断是否发生闹钟事件。

接着，我们介绍一下 `rtc_get_time` 函数，其定义如下：

```

/**
* @brief      得到当前的时间
* @note      该函数不直接返回时间，时间数据保存在 calendar 结构体里面
* @param      无
* @retval     无
*/
void rtc_get_time(void)
{
    static uint16_t daycnt = 0;
    uint32_t seccount = 0;
    uint32_t temp = 0;

```

```
uint16_t temp1 = 0;
/* 平年的月份日期表 */
const uint8_t month_table[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

seccount = RTC->CNTH;      /* 得到计数器中的值(秒钟数) */
seccount <= 16;
seccount += RTC->CNTL;

temp = seccount / 86400;    /* 得到天数(秒钟数对应的) */

if (daycnt != temp)        /* 超过一天了 */
{
    daycnt = temp;
    temp1 = 1970;          /* 从1970年开始 */

    while (temp >= 365)
    {
        if (rtc_is_leap_year(temp1)) /* 是闰年 */
        {
            if (temp >= 366)
            {
                temp -= 366; /* 闰年的秒钟数 */
            }
            else
            {
                break;
            }
        }
        else
        {
            temp -= 365; /* 平年 */
        }

        temp1++;
    }

    calendar.year = temp1; /* 得到年份 */
    temp1 = 0;

    while (temp >= 28)      /* 超过了一个月 */
    {
        /* 当年是不是闰年/2月份 */
        if (rtc_is_leap_year(calendar.year) && temp1 == 1)
        {
            if (temp >= 29)
            {
                temp -= 29; /* 闰年的秒钟数 */
            }
            else
            {
                break;
            }
        }
        else
        {
            if (temp >= month_table[temp1])
            {
                temp -= month_table[temp1]; /* 平年 */
            }
            else
            {
                break;
            }
        }
    }
}
```

```

    temp1++;
}

calendar.month = temp1 + 1; /* 得到月份 */
calendar.date = temp + 1; /* 得到日期 */
}

temp = seccount % 86400; /* 得到秒钟数 */
calendar.hour = temp / 3600; /* 小时 */
calendar.min = (temp % 3600) / 60; /* 分钟 */
calendar.sec = (temp % 3600) % 60; /* 秒钟 */
/* 获取星期 */
calendar.week = rtc_get_week(calendar.year, calendar.month, calendar.date);
}

```

该函数其实就是将存储在秒钟寄存器 RTC->CNTL 和 RTC->CNTH 中的秒钟数据转换为真正的时间和日期。该代码还用到了一个 `calendar` 的结构体，`calendar` 是我们在 `rtc.h` 里面将要定义的一个时间结构体，用来存放时钟的年月日时分秒等信息。因为 STM32 的 RTC 只有秒钟计数器，而年月日，时分秒则需要我们自己软件计算。我们把计算好的值保存在 `calendar` 里面，方便其他函数调用。

接着，我们介绍一下使用多次数最多的函数 `rtc_date2sec`，该函数代码如下：

```

/**
 * @brief      将年月日时分秒转换成秒钟数
 * @note      以 1970 年 1 月 1 日为基准，1970 年 1 月 1 日，0 时 0 分 0 秒，表示第 0 秒钟
 *           最大表示到 2105 年，因为 uint32_t 最大表示 136 年的秒钟数 (不包括闰年)！
 *           本代码参考只 linux mktime 函数，原理说明见此贴：
 *           http://www.openedv.com/thread-63389-1-1.html
 * @param     syear : 年份
 * @param     smon  : 月份
 * @param     sday  : 日期
 * @param     hour  : 小时
 * @param     min   : 分钟
 * @param     sec   : 秒钟
 * @retval    转换后的秒钟数
 */
static long rtc_date2sec(uint16_t syear, uint8_t smon, uint8_t sday,
                        uint8_t hour, uint8_t min, uint8_t sec)
{
    uint32_t Y, M, D, X, T;
    signed char monx = smon; /* 将月份转换成带符号的值，方便后面运算 */

    if (0 >= (monx - 2)) /* 1..12 -> 11,12,1..10 */
    {
        monx += 12; /* Puts Feb last since it has leap day */
        syear -= 1;
    }

    /* 公元元年 1 到现在的闰年数 */
    Y = (syear - 1) * 365 + syear / 4 - syear / 100 + syear / 400;
    M = 367 * monx / 12 - 30 + 59;
    D = sday - 1;
    X = Y + M + D - 719162; /* 减去公元元年到 1970 年的天数 */
    T = ((X * 24 + hour) * 60 + min) * 60 + sec; /* 总秒钟数 */
    return T;
}

```

该函数参考了 linux 的 `mktime` 函数，用于将年月日时分秒转化成秒钟数，进而被其他函数使用，例如 `rtc_set_time` 和 `rtc_set_alarm`，那两个函数的形参是需要使用 `rtc_date2sec` 函数获取秒钟数，进而操作寄存器的方法把总秒数写入特定的寄存器完成相对应的功能。

前面介绍的函数 `rtc_init` 中，存在 RTC 中断使能操作，那么这里必定是有中断服务函数，

接下来看中断服务函数，代码如下：

```
/**
 * @brief      RTC 时钟中断
 * @note      秒钟中断服务函数, 顺带处理闹钟标志
 *           根据 RTC_CRL 寄存器的 SECF 和 ALRF 位区分是哪个中断
 * @param      无
 * @retval     无
 */
void RTC_IRQHandler(void)
{
    if ( __HAL_RTC_ALARM_GET_FLAG(&g_rtc_handle, RTC_FLAG_SEC) != RESET) /*秒中断*/
    {
        rtc_get_time(); /* 更新时间 */
        __HAL_RTC_ALARM_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_SEC); /* 清除秒中断 */
        //printf("sec:%d\r\n", calendar.sec); /* 打印秒钟 */
    }
    /* 顺带处理闹钟标志 */
    if ( __HAL_RTC_ALARM_GET_FLAG(&g_rtc_handle, RTC_FLAG_ALRAF) != RESET)
    {
        __HAL_RTC_ALARM_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_ALRAF); /*清除闹钟标志*/
        printf("Alarm Time:%d-%d-%d %d:%d:%d\n", calendar.year, calendar.month,
            calendar.date, calendar.hour, calendar.min, calendar.sec);
    }
    __HAL_RTC_ALARM_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_OW); /* 清除溢出中断标志 */

    /* 等待 RTC 寄存器操作完成, 即等待 RTOFF == 1 */
    while (!__HAL_RTC_ALARM_GET_FLAG(&g_rtc_handle, RTC_FLAG_RTOFF));
}
```

RTC_IRQHandler 中断服务函数用于 RTC 秒中断的，由于在 rtc_init 中已经配置好了时钟周期为 1 秒，所以每一秒都会跳进 RTC 中断服务函数中。在函数中，判断秒中断是否触发，由于每一次都是秒中断触发，所以可以先更新时间，然后把 printf 的注释去掉看一下效果，是不是每一秒打印一下。接着判断闹钟标志是否置位，这个闹钟标志跟我们的 rtc_set_alarm 函数有关，假设时间到了闹钟设置的时间，就会跳进该秒中断中顺带处理闹钟标志，执行函数体的指令。执行完上述的任务之后，需要在最后清除溢出中断标志。

rtc.c 的其他程序，这里就不再介绍了，请大家直接看源码。

2. main.c 代码

在 main.c 里面编写如下代码：

```
/* 定义字符数组用于显示周 */
char* weekdays[]={"Sunday","Monday","Tuesday","Wednesday",
    "Thursday","Friday","Saturday"};

int main(void)
{
    uint8_t tbuf[40];
    uint8_t t = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    rtc_init(); /* 初始化 RTC */
    rtc_set_alarm(2020, 4, 26, 9, 23, 45); /* 设置一次闹钟 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
```

```

lcd_show_string(30, 70, 200, 16, 16, "RTC TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

while (1)
{
    t++;

    if ((t % 10) == 0)                /* 每 100ms 更新一次显示数据 */
    {
        rtc_get_time();
        sprintf((char *)tbuf, "Time:%02d:%02d:%02d", calendar.hour,
            calendar.min, calendar.sec);
        lcd_show_string(30, 120, 210, 16, 16, (char *)tbuf, RED);
        sprintf((char *)tbuf, "Date:%04d-%02d-%02d", calendar.year,
            calendar.month, calendar.date);
        lcd_show_string(30, 140, 210, 16, 16, (char *)tbuf, RED);
        sprintf((char *)tbuf, "Week:%s", weekdays[calendar.week]);
        lcd_show_string(30, 160, 210, 16, 16, (char *)tbuf, RED);
    }

    if ((t % 20) == 0)
    {
        LED0_TOGGLE();                /* 每 200ms, 翻转一次 LED0 */
    }

    delay_ms(10);
}
}

```

我们在无限循环中每 100ms 读取 RTC 的时间和日期(一次),并显示在 LCD 上面。每 200ms, 翻转一次 LED0。

为方便 RTC 相关函数的调用验证,在 usmart_config.c 里面,修改了 usmart_nametab 如下:

```

/* 函数名列表初始化(用户自己添加)
 * 用户直接在这里输入要执行的函数名及其查找串
 */
struct _m_usmart_nametab usmart_nametab[] =
{
    #if USMART_USE_WRFUNS == 1        /* 如果使能了读写操作 */
        (void *)read_addr, "uint32_t read_addr(uint32_t addr)",
        (void *)write_addr, "void write_addr(uint32_t addr,uint32_t val)",
    #endif
        (void *)delay_ms, "void delay_ms(uint16_t nms)",
        (void *)delay_us, "void delay_us(uint32_t nus)",

        (void *)rtc_read_bkr, "uint16_t rtc_read_bkr(uint32_t bkrx)",
        (void *)rtc_write_bkr, "void rtc_write_bkr(uint32_t bkrx, uint16_t data)",
        (void *)rtc_get_week, "uint8_t rtc_get_week(uint16_t year, uint8_t month,
uint8_t day)",
        (void *)rtc_set_time, "uint8_t rtc_set_time(uint16_t syear, uint8_t smon,
uint8_t sday, uint8_t hour, uint8_t min, uint8_t sec)",
        (void *)rtc_set_alarm, "uint8_t rtc_set_alarm(uint16_t syear, uint8_t smon,
uint8_t sday, uint8_t hour, uint8_t min, uint8_t sec)",
};

```

将 RTC 的一些相关函数加入了 usmart, 这样通过串口就可以直接设置 RTC 时间、闹钟。至此, RTC 的软件设计就完成了, 接下来就让我们来检验一下, 程序是否正确。

27.4 下载验证

将程序下载到开发板后, 可以看到 LED0 不停的闪烁, 提示程序已经在运行了。然后, 可以看到 LCD 开始显示时间, 实际显示效果如图 27.4.1 所示:

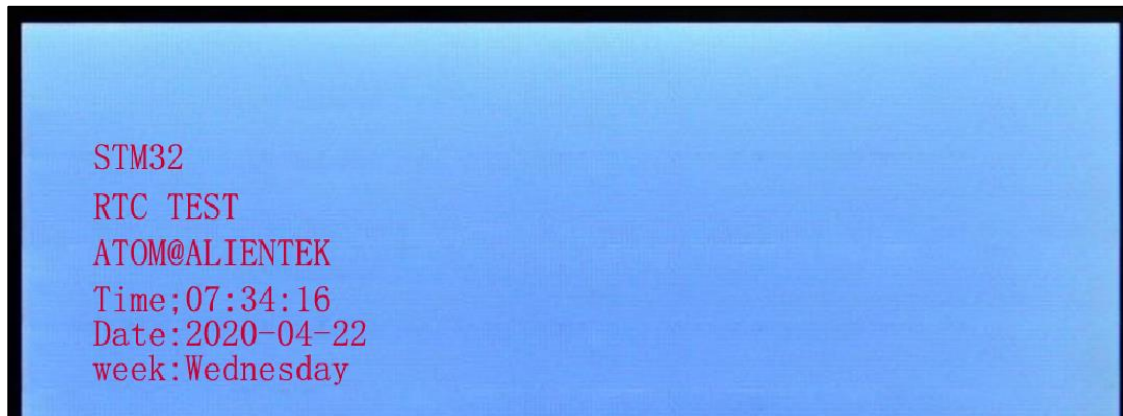


图 27.4.1 RTC 实验测试图

如果时间不正确，可以利用上一章介绍的 usmart 工具，通过串口来设置，并且可以设置闹钟时间等，如图 27.4.2 所示：

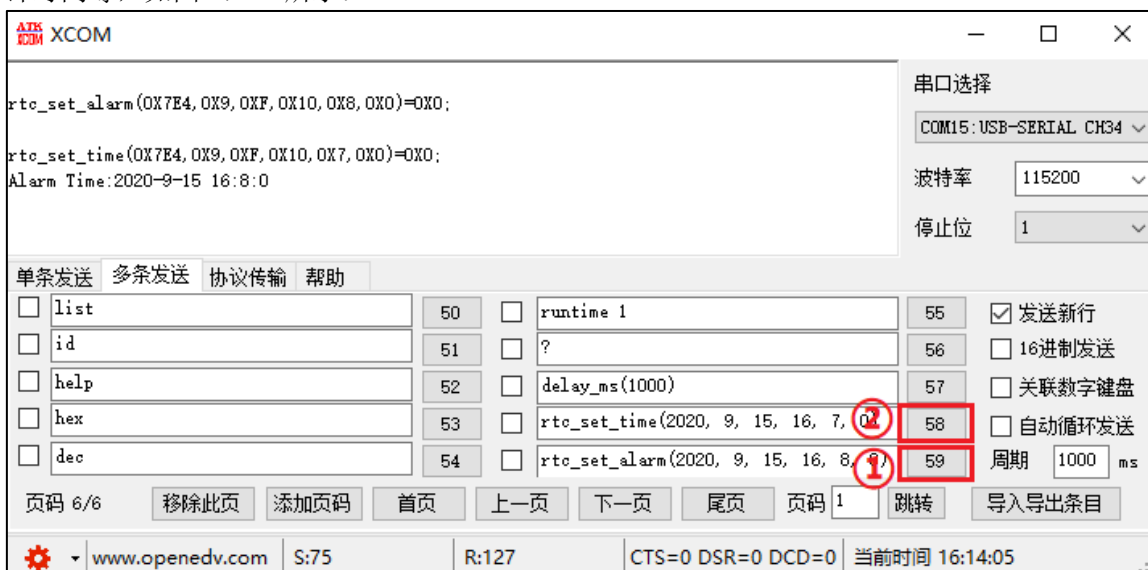


图 27.4.2 通过 USMART 设置时间并测试闹钟

按照图中编号 1、2 顺序，设置闹钟、设置时间。然后等待我们设置的时间到来后，串口打印 Alarm Time:2020-9-15 12:30:45 这个字符串，证明我们的闹钟程序正常运行了！

第二十八章 低功耗实验

本章，我们将介绍 STM32F103 的电源控制（PWR），并实现低功耗模式相关功能。我们将通过四个实验来学习并实现低功耗相关功能，分别是 PVD 电压监控实验、睡眠模式实验、停止模式实验和待机模式实验。

本章分为如下几个小节：

- 28.1 电源控制（PWR）简介
- 28.2 PVD 电压监控实验
- 28.3 睡眠模式实验
- 28.4 停止模式实验
- 28.5 待机模式实验

28.1 电源控制（PWR）简介

电源控制部分（PWR）概述了不同电源域的电架构以及电源配置控制器。PWR 的内容比较多，我们把它们的主要特性概括为以下 3 点：

电源系统：V_{DDA} 供电区域、V_{DD} 供电区域、1.8V 供电区域、后备供电区域。

电源监控：POR/PDR 监控器、PVD 监控器。

电源管理：低功耗模式。

下面将分别对这 3 个特性进行简单介绍。

28.1.1 电源系统

为了方便对电源系统进行管理，设计者把 STM32 的内核和外设等器件跟据功能划分了不同的电源区域，具体如图 28.1.1.1 所示。

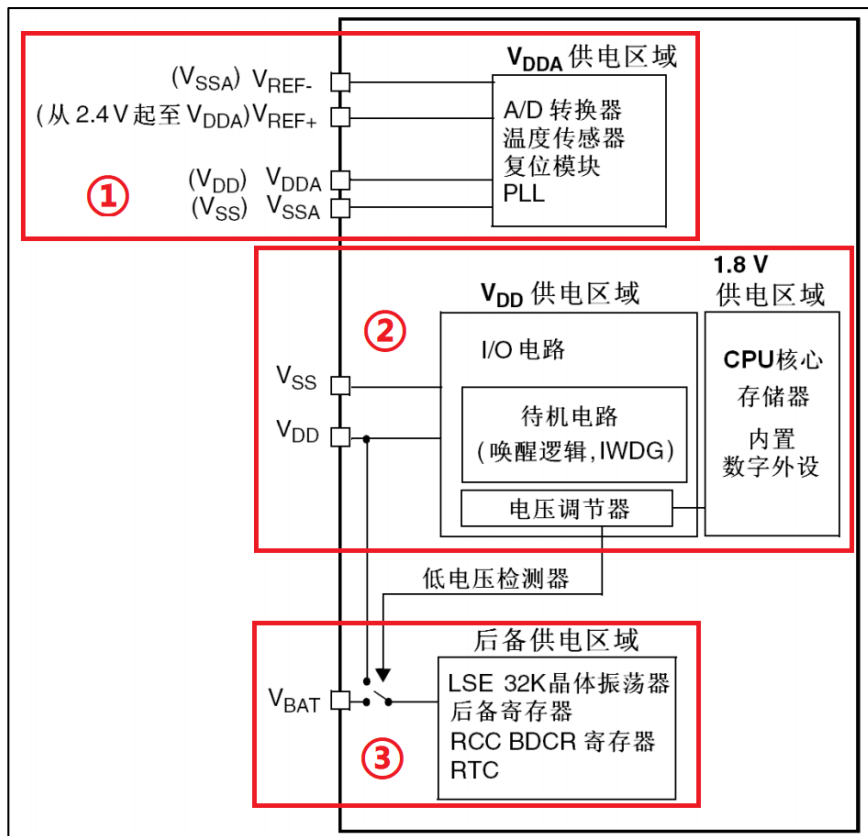


图 28.1.1.1 电源概述框图

在电源概述框图中我们划分了 3 个区域①②③，分别是独立的 A/D 转换器供电和参考电压、电压调节器、电池备份区域。下面分别进行简单介绍：

① 独立的 A/D 转换器供电和参考电压 (V_{DDA} 供电区域)

V_{DDA} 供电区域，主要是 ADC 电源以及参考电压，STM32 的 ADC 模块配备独立的供电方式，使用了 V_{DDA} 引脚作为输入，使用 V_{SSA} 引脚作为独立地连接， V_{REF} 引脚为提供给 ADC 的参考电压。

② 电压调节器 ($V_{DD}/1.8V$ 供电区域)

电压调节器是 STM32 的电源系统中最核心部分，连接 V_{DD} 供电区域和 1.8V 供电区域。 V_{DD} 供电来自于 V_{SS} 和 V_{DD} ，给 I/O 电路以及待机电路供电，电压调节器主要为备份域以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，调节器的输出电压约为 1.8V，因此由调压器供电的区域称为 1.8V 供电区域。

电压调节器根据应用方式不同有三种不同的工作模式。在运行模式下，调节器以正常工作模式为内核、内存和外设提供 1.8V；在停止模式下，调节器以低功耗模式提供 1.8V 电源，以保存寄存器和 SRAM 的内容。在待机模式下，调节器停止供电，除了备用电路和备份域外，寄存器和 SRAM 的内容全部丢失。

③ 电池备份区域 (后备供电区域)

电池备份区域也就是后备供电区域，使用电池或者其他电源连接到 V_{BAT} 脚上，当 V_{DD} 断电时，可以保存备份寄存器的内容和维持 RTC 的功能。同时 V_{BAT} 引脚也为 RTC 和 LSE 振荡器供电，这保证了当主要电源被切断时，RTC 能够继续工作。切换到 V_{BAT} 供电由复位模块中的掉电复位功能控制。

28.1.2 电源监控

电源监控的部分我们主要关注 PVD 监控器，此外还需要知道上电复位(POR)/掉电复位(PDR)。其他部分的内容请大家查看《STM32F10xxx 参考手册_V10 (中文版).pdf》第 4.2 节 (38 页)。

● 上电复位(POR)/掉电复位(PDR)

上电时，当 V_{DD} 低于指定 V_{POR} 阈值时，系统无需外部复位电路便会保持复位模式。一旦 V_{DD} 电源电压高于 V_{POR} 阈值，系统便会退出复位状态，芯片正常工作。掉电时，当 V_{DD} 低于指定 V_{PDR} 阈值时，系统就会保持复位模式。如图 28.1.2.1 所示，RESET 为上电复位信号。

注意：POR 与 PDR 的复位电压阈值是固定的， V_{POR} 阈值 (典型值) 为 1.92V， V_{PDR} 阈值 (典型值) 为 1.88V。

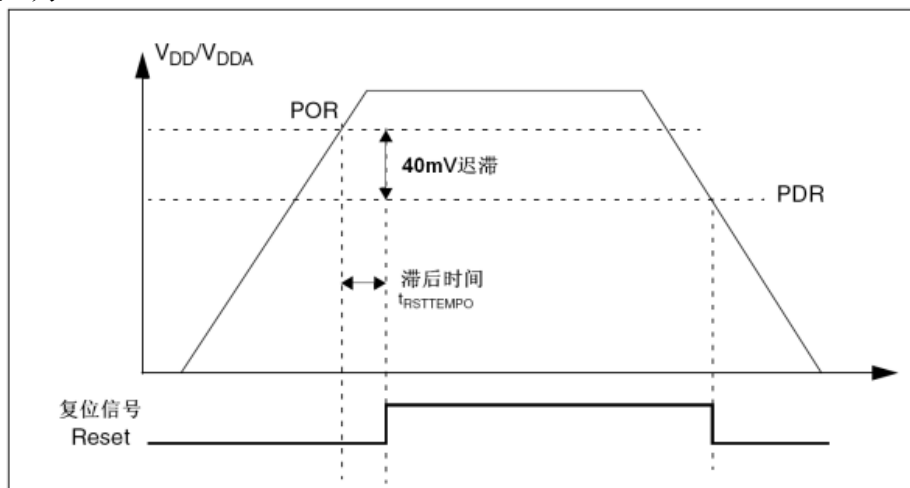


图 28.1.2.1 上电复位/掉电复位波形

● 可编程电压检测器(PVD)

上面介绍的 POR、PDR 功能都是设置电压阈值与外部供电电压 V_{DD} 比较，当 V_{DD} 低于设置的电压阈值时，就会直接进入复位状态，防止电压不足导致的误操作。

下面介绍可编程电压检测器 (PVD)，它可以实时监视 V_{DD} 的电压，方法是将 V_{DD} 与 PWR

控制寄存器 (PWR_CR) 中的 PLS[2:0] 位所选的 V_{PVD} 阈值进行比较。其中 PWR_CSR 寄存器中的 PVDO 位决定了 V_{DD} 是高于 V_{PVD} 还是低于 V_{PVD} ，本实验中配置的是 V_{DD} 低于 V_{PVD} 阈值这个条件。当检测到电压低于 V_{PVD} 阈值时，如果使能 EXTI16 线中断，即使能 PVD 中断，可以产生 PVD 中断，具体取决于 EXTI16 线配置为检测上升还是下降沿，然后在复位前，在中断服务程序中执行紧急关闭系统等任务。PVD 阈值检测波形，如图 28.1.2.2 所示。

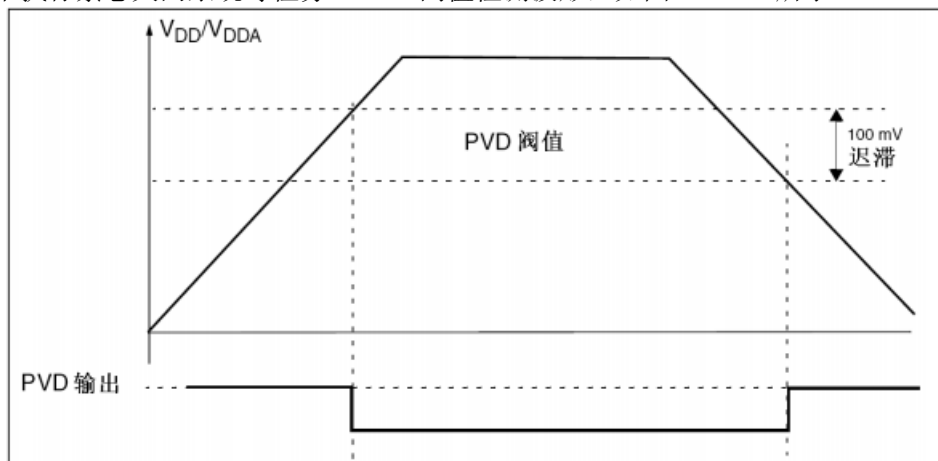


图 28.1.2.2 PVD 检测波形

PVD 阈值有 8 个等级，有上升沿和下降沿的区别，具体由 PWR_CSR 寄存器中的 PVDO 位决定。PVD 阈值等级表具体如表 28.1.2.1 所示。

| Symbol | Parameter | Conditions | Min | Typ | Max |
|-----------|---|-----------------------------|------|------|------|
| V_{PVD} | Programmable voltage detector level selection | PLS[2:0]=000 (rising edge) | 2.1 | 2.18 | 2.26 |
| | | PLS[2:0]=000 (falling edge) | 2 | 2.08 | 2.16 |
| | | PLS[2:0]=001 (rising edge) | 2.19 | 2.28 | 2.37 |
| | | PLS[2:0]=001 (falling edge) | 2.09 | 2.18 | 2.27 |
| | | PLS[2:0]=010 (rising edge) | 2.28 | 2.38 | 2.48 |
| | | PLS[2:0]=010 (falling edge) | 2.18 | 2.28 | 2.38 |
| | | PLS[2:0]=011 (rising edge) | 2.38 | 2.48 | 2.58 |
| | | PLS[2:0]=011 (falling edge) | 2.28 | 2.38 | 2.48 |
| | | PLS[2:0]=100 (rising edge) | 2.47 | 2.58 | 2.69 |
| | | PLS[2:0]=100 (falling edge) | 2.37 | 2.48 | 2.59 |
| | | PLS[2:0]=101 (rising edge) | 2.57 | 2.68 | 2.79 |
| | | PLS[2:0]=101 (falling edge) | 2.47 | 2.58 | 2.69 |
| | | PLS[2:0]=110 (rising edge) | 2.66 | 2.78 | 2.9 |
| | | PLS[2:0]=110 (falling edge) | 2.56 | 2.68 | 2.8 |
| | | PLS[2:0]=111 (rising edge) | 2.76 | 2.88 | 3 |
| | | PLS[2:0]=111 (falling edge) | 2.66 | 2.78 | 2.9 |

表 28.1.2.1 PVD 阈值等级

28.1.3 电源管理

电源管理的部分我们要关注低功耗模式，在 STM32 的正常工作中，具有四种工作模式，运行、睡眠、停止以及待机。在上电复位后，STM32 处于运行状态时，当内核不需要继续运行，就可以选择进入后面的三种模式降低功耗。这三种低功耗模式电源消耗不同、唤醒时间不同和唤醒源不同，我们要根据自身的需要选择合适的低功耗模式。

下面是低功耗模式汇总介绍，如下表所示。

| 模式 | 进入 | 唤醒 | 对1.8V区域时钟的影响 | 对V _{DD} 区域时钟的影响 | 电压调节器 |
|-------------------------------------|---------------------------------------|--|------------------------|--------------------------|----------------------------------|
| 睡眠 (SLEEP-NOW或 SLEEP-ON-EXIT) | WFI | 任一中断 | CPU时钟关, 对其他时钟和ADC时钟无影响 | 无 | 开 |
| | WFE | 唤醒事件 | | | |
| 待机 | PDDS和LPDS位 +SLEEPDEEP位 +WFI或WFE | 任一外部中断(在外部中断寄存器中设置) | 关闭所有1.8V区域的时钟 | HSI 和HSE的振荡器关闭 | 开启或处于低功耗模式(依据电源控制寄存器(PWR_CR)的设置) |
| 待机 | PDDS位 +SLEEPDEEP位 +WFI或WFE | 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 |

表 28.1.3.1 低功耗模式汇总

下面对睡眠模式、停止模式和待机模式，分开介绍。

1、睡眠模式

进入睡眠模式，Cortex_M3 内核停止，所有外设包括 Cortex_M3 核心的外设，如 NVIC、系统时钟 (SysTick) 等仍在运行，有两种进入睡眠模式的模式 WFI 和 WFE。WFI(Wait for interrupt) 和 WFE(Wait for event)是内核指令，会调用一些汇编指令，我们会使用即可，更详细的描述可以查看《CM3 权威指南》。睡眠后唤醒的方式即由等待“中断”唤醒和“事件”唤醒。

| 睡眠模式 | 说明 |
|------|--|
| 进入模式 | 立即睡眠：在执行 WFI 或 WFE 指令时立刻进入睡眠模式 触发条件：内核寄存器 SLEEPDEEP 位置 0，调用 WFI 或 WFE 指令立刻进入睡眠模式 |
| | 退出时睡眠：在退出优先级最低的中断服务程序后才进入睡眠模式 触发条件：内核寄存器 SLEEPONEXIT 位置 1，从中断服务程序返回后立刻睡眠，这种情况下，处理器的所有工作就只是响应中断了，其他时间在睡眠 |
| 退出模式 | 如果执行 WFI 进入睡眠模式，可使用任意中断唤醒 如果执行 WFE 进入睡眠模式，可使用事件唤醒 |
| 唤醒延迟 | 无延时 |

表 28.1.3.2 睡眠模式进入及退出方法

2、停止模式

进入停止模式，所有的时钟都关闭，所有的外设也就停止了工作。但是 V_{DD} 电源是没有关闭的，所以内核的寄存器和内存信息都保留下来，等待重新开启时钟就可以从上次停止的地方继续执行程序。

值得注意的是：当电压调节器处于低功耗模式下，当系统从停止模式退出时，将会有一段额外的启动延时。如果在停止模式期间保持内部调节器开启，则退出启动时间会缩短，但相应的功耗会增加。

| 停止模式 | 说明 |
|------|---|
| 进入模式 | 内核寄存器的 SLEEPDEEP 位置 1，PWR_CR 寄存器的 PDDS 置 0，然后调用 WFI 或 WFE 指令即可进入停止模式 注意：PWR_CR 寄存器 LPDS=0 时，调压器工作在正常模式，LPDS=1 时工作做在低功耗模式 |
| 退出模式 | 如果执行 WFI 进入停止模式： 设置任一外部中断线为中断模式进行中断唤醒 如果执行 WFE 进入停止模式： 设置任一外部中断线为事件模式进行中断唤醒 |
| 唤醒延迟 | HSI RC 唤醒时间 + 电压调节器从低功耗唤醒的时间。 |

表 28.1.3.2 停止模式进入及退出方法

3、待机模式

待机模式可实现最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器，整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。除备份域（RTC 寄存器、RTC 备份寄存器和备份 SRAM）和待机电路中的寄存器外，SRAM 和其他寄存器内容都将丢失。不过如果我们使能了备份区域（备份 SRAM、RTC、LSE），那么待机模式下的功耗，将达到 3.8uA 左右。

那么我们如何进入待机模式呢？其实很简单，只要按表 28.1.3.3 所示的步骤执行就可以了：

| 待机模式 | 说明 |
|------|---|
| 进入模式 | 进入待机模式步骤如下： ①，设置 CM3 系统控制寄存器中的 SLEEPDEEP 位置 1 ②，设置电源控制寄存器（PWR_CR）中 PDDS 位置 1 ③，设置电源控制寄存器（PWR_CR）中的 WUF 位置 0 ④，调用 WFI 或 WFE 指令即可进入待机模式 |
| 退出模式 | 退出条件分别如下： ①，WKUP 引脚的上升沿 ②，RTC 闹钟事件的上升沿 ③，NRST 引脚上外部复位 ④，IWDG 复位 |
| 唤醒延迟 | 复位阶段时电压调节器的启动 |

表 28.1.3.3 待机模式进入及退出方法

28.2 PVD 电压监控实验

本小节我们来学习 PVD 电压监控实验，该部分的知识点内容请回顾 28.1.2 电源监控。我们直接从寄存器介绍开始。

28.2.1 PWR 寄存器

本实验用到 PWR 的部分寄存器，在《STM32F10XXX 参考手册(中文版)》的 4.4 小节可以找到 PWR 寄存器描述。这里我们只介绍 PVD 电压监控实验用到的 PWR 控制寄存器(PWR_CR)，还有就是我们要用到 EXTI16 线中断，所以还要配置 EXTI 相关的寄存器，具体如下：

● PWR 控制寄存器 (PWR_CR)

PWR 控制寄存器描述如图 28.2.1.1 所示：

| | | | | | | | | | | | | | | | |
|-------|----|--|----|----|----|----|----|-----|----------|----|------|------|-------|-------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | DBP | PLS[2:0] | | PVDE | CSBF | CWUF | PDDS | LPDS |
| | | | | | | | | rw | rw | rw | rw | rw | rc_wl | rc_wl | rw |
| 位 7:5 | | PLS[2:0]: PVD电平选择 这些位用于选择电源电压监测器的电压阈值 000: 2.2V 100: 2.6V 001: 2.3V 101: 2.7V 010: 2.4V 110: 2.8V 011: 2.5V 111: 2.9V 注：详细说明参见数据手册中的电气特性部分。 | | | | | | | | | | | | | |
| 位 4 | | PVDE: 电源电压监测器(PVD)使能 0: 禁止PVD 1: 开启PVD | | | | | | | | | | | | | |

图 28.2.1.1 PWR_CR 寄存器（部分）

位[7:5] PLS 用于设置 PVD 检测的电压阈值，即前面我们介绍 PVD 的 8 个等级阈值选择。

位 4 PVDE 位，用于使能或者禁止 PVD 检测，显然我们要使能 PVD 检测，该位置 1。

这个寄存器还有其它的位我们没有列出来，也是跟电源相关的，如待机，掉电等，我们后面的实验再讲解这些功能，这里先跳过。

● EXTI 中断屏蔽寄存器 (EXTI_IMR)

EXTI 中断屏蔽寄存器描述如图 28.2.1.2 所示：

| | | | | | | | | | | | | | | | |
|--------|------|---|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | MR19 | MR18 | MR17 | MR16 |
| | | | | | | | | | | | | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MR15 | MR14 | MR13 | MR12 | MR11 | MR10 | MR9 | MR8 | MR7 | MR6 | MR5 | MR4 | MR3 | MR2 | MR1 | MR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位 19:0 | | MRx: 线x上的中断屏蔽 (Interrupt Mask on line x) 0: 屏蔽来自线x上的中断请求; 1: 开放来自线x上的中断请求。 注：位19只适用于互联型产品，对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.2.1.2 EXTI_IMR 寄存器

我们要使用到 EXTI16 线中断，所以 MR16 位要置 1，即开放来自 EXTI16 线的中断请求。

● EXTI 上升沿触发选择寄存器 (EXTI_RTSR)

EXTI 上升沿触发选择寄存器描述如图 28.2.1.3 所示：

| | | | | | | | | | | | | | | | |
|-------|------|---|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | TR19 | TR18 | TR17 | TR16 |
| | | | | | | | | | | | | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TR15 | TR14 | TR13 | TR12 | TR11 | TR10 | TR9 | TR8 | TR7 | TR6 | TR5 | TR4 | TR3 | TR2 | TR1 | TR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位18:0 | | TRx: 线x上的上升沿触发事件配置位 (Rising trigger event configuration bit of line x) 0: 禁止输入线x上的上升沿触发(中断和事件) 1: 允许输入线x上的上升沿触发(中断和事件) 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.2.1.3 EXTI_RTISR 寄存器

我们要使用到 EXTI16 线中断, 所以 TR16 位要置 1, 即允许 EXTI16 线的上升沿触发。

● EXTI 下降沿触发选择寄存器 (EXTI_FTSR)

EXTI 下降沿触发选择寄存器描述如图 28.2.1.4 所示:

| | | | | | | | | | | | | | | | |
|-------|------|--|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | TR19 | TR18 | TR17 | TR16 |
| | | | | | | | | | | | | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TR15 | TR14 | TR13 | TR12 | TR11 | TR10 | TR9 | TR8 | TR7 | TR6 | TR5 | TR4 | TR3 | TR2 | TR1 | TR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位18:0 | | TRx: 线x上的下降沿触发事件配置位 (Falling trigger event configuration bit of line x) 0: 禁止输入线x上的下降沿触发(中断和事件) 1: 允许输入线x上的下降沿触发(中断和事件) 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.2.1.4 EXTI_FTSR 寄存器

我们要使用到 EXTI16 线中断, 所以 TR16 位要置 1, 即允许 EXTI16 线上的下降沿触发。

● EXTI 挂起寄存器 (EXTI_PR)

EXTI 挂起寄存器描述如图 28.2.1.5 所示:

| | | | | | | | | | | | | | | | |
|-------|-------|--|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | PR19 | PR18 | PR17 | PR16 |
| | | | | | | | | | | | | rc w1 | rc w1 | rc w1 | rc w1 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PR15 | PR14 | PR13 | PR12 | PR11 | PR10 | PR9 | PR8 | PR7 | PR6 | PR5 | PR4 | PR3 | PR2 | PR1 | PR0 |
| rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 | rc w1 |
| 位18:0 | | PRx: 挂起位 (Pending bit) 0: 没有发生触发请求 1: 发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件, 该位被置'1'。在该位中写入'1'可以清除它, 也可以通过改变边沿检测的极性清除。 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.2.1.5 EXTI_PR 寄存器

EXTI 挂起寄存器 EXTI_PR 管理的是 EXTI0 线到 EXTI19 线的中断标志位。在 PVD 中断服务函数里面, 我们记得要对 PR16 位写 1, 来清除 EXTI16 线的中断标志。

28.2.2 硬件设计

1. 例程功能

开发板供电正常的话，LCD 屏会显示"PVD Voltage OK!".当供电电压过低，则会通过 PVD 中断服务函数将 LED1 点亮；当供电电压正常后，会在 PVD 中断服务函数将 LED1 熄灭。LED0 闪烁，提示程序运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) PVD(可编程电压监测器)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PVD 属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 LED0 和 LCD 来指示进入 PVD 中断的情况。

28.2.3 程序设计

28.2.3.1 PWR 的 HAL 库驱动

PWR 在 HAL 库中的驱动代码在 stm32f1xx_hal_pwr.c 文件（及其头文件）中。

1. HAL_PWR_ConfigPVD 函数

PVD 的初始化函数，其声明如下：

```
void HAL_PWR_ConfigPVD (PWR_PVDTypeDef *sConfigPVD);
```

● 函数描述：

用于初始化 PWR。

● 函数形参：

形参 1 是 PWR_PVDTypeDef 结构体类型变量，其定义如下：

```
typedef struct
{
    uint32_t PVDLevel; /* 指定 PVD 检测级别 */
    uint32_t Mode; /* 指定 PVD 的 EXTI 检测模式 */
}PWR_PVDTypeDef;
```

1) PVDLevel: 指向 PVD 检测级别，对应 PWR_CR 寄存器的 PLS 位的设置，取值范围 PWR_PVDLEVEL_0 到 PWR_PVDLEVEL_7，共八个级别。

2) Mode: 指定 PVD 的 EXTI 边沿检测模式。

● 函数返回值：

无

PVD 电压监控配置步骤

1) 配置 PVD，使能 PVD 时钟。

调用 HAL_PWR_ConfigPVD 函数配置 PVD，包括检测电压级别、使用中断线触发方式等。

2) 使能 PVD 检测，配置 PVD/AVD 中断优先级，开启 PVD 中断。

通过 HAL_PWR_EnablePVD 函数使能 PVD 检测。

通过 HAL_NVIC_EnableIRQ 函数使能 PVD 中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

3) 编写中断服务函数。

PVD 中断服务函数为 PVD_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。

HAL 库有专门的 PVD 中断处理函数，我们只需要在 PVD 中断服务函数里面调用 HAL_PWR_PVD_IRQHandler() 函数，然后逻辑代码在 PVD 中断服务回调函数 HAL_PWR_PVDCallback 中编写，详见本例程源码。

28.2.3.2 程序流程图

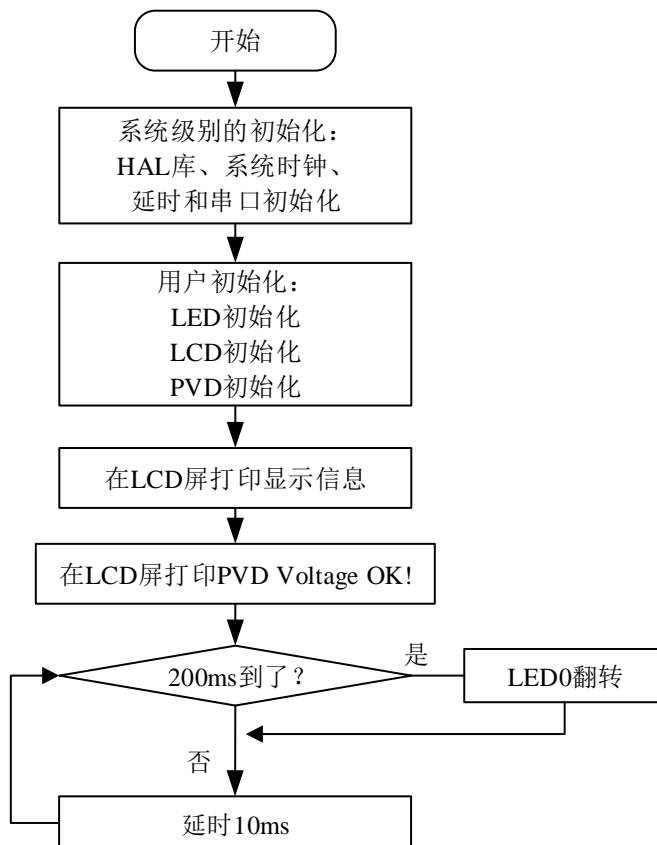


图 28.2.3.2.1 PVD 电压监控实验程序流程图

28.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。该章节有四个实验，每一个实验的代码都是在上一个实验后面追加。pwr.h 头文件只有函数声明，下面直接开始介绍 pwr.c 的程序，首先是 PVD 初始化函数。

```

/**
 * @brief      初始化 PVD 电压监视器
 * @param      pls: 电压等级 (PWR_PVD_detection_level)
 * @arg        PWR_PVDLEVEL_0, 2.2V;
 * @arg        PWR_PVDLEVEL_1, 2.3V;
 * @arg        PWR_PVDLEVEL_2, 2.4V;
 * @arg        PWR_PVDLEVEL_3, 2.5V;
 * @arg        PWR_PVDLEVEL_4, 2.6V;
 * @arg        PWR_PVDLEVEL_5, 2.7V;
 * @arg        PWR_PVDLEVEL_6, 2.8V;
 * @arg        PWR_PVDLEVEL_7, 2.9V;
 * @retval     无
 */
void pwr_pvd_init(uint32_t pls)
{
    PWR_PVDTypeDef pwr_pvd = {0};

    __HAL_RCC_PWR_CLK_ENABLE(); /* 使能 PWR 时钟 */

```

```
pwr_pvd.PVDLevel = pls; /* 检测电压级别 */
/* 使用中断线的上升沿和下降沿双边沿触发 */
pwr_pvd.Mode = PWR_PVD_MODE_IT_RISING_FALLING;
HAL_PWR_ConfigPVD(&pwr_pvd);

HAL_NVIC_SetPriority(PVD_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(PVD_IRQn);
HAL_PWR_EnablePVD(); /* 使能 PVD 检测 */
}
```

这里需要注意的就是 PVD 中断线选择的是上升沿和下降沿双边沿触发，其他的内容前面已经讲过。

下面介绍的是 PVD 中断服务函数及其回调函数，函数定义如下：

```
/**
 * @brief      PVD 中断服务函数
 * @param      无
 * @retval     无
 */
void PVD_IRQHandler(void)
{
    HAL_PWR_PVD_IRQHandler();
}

/**
 * @brief      PVD 中断服务回调函数
 * @param      无
 * @retval     无
 */
void HAL_PWR_PVDCallback(void)
{
    if (__HAL_PWR_GET_FLAG(PWR_FLAG_PVDO)) /* 电压比 PLS 所选电压还低 */
    {
        /* LCD 显示电压低 */
        lcd_show_string(30, 130, 200, 16, 16, "PVD Low Voltage!", RED);
        LED1(0); /* 点亮 LED1, 表明电压低了 */
    }
    else
    {
        /* LCD 显示电压正常 */
        lcd_show_string(30, 130, 200, 16, 16, "PVD Voltage OK! ", BLUE);
        LED1(1); /* 灭掉绿灯 */
    }
}
```

HAL_PWR_PVDCallback 回调函数中首先是判断 V_{DD} 电压是否比 PLS 所选电压还低，是的话，就在 LCD 显示 PVD Low Voltage! 并且点亮 LED1，否则，在 LCD 显示 PVD Voltage OK! 并且关闭 LED1。

在 main 函数里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    pwr_pvd_init(PWR_PVDLEVEL_7); /* PVD 2.9V 检测 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "PVD TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
}
```

```
/* 默认 LCD 显示电压正常 */  
lcd_show_string(30, 130, 200, 16, 16, "PVD Voltage OK! ", BLUE);  
  
while (1)  
{  
    if ((t % 20) == 0)  
    {  
        LED0_TOGGLE();          /* 每 200ms, 翻转一次 LED0 */  
    }  
  
    delay_ms(10);  
    t++;  
}  
}
```

这里我们选择 PVD 的检测电压阈值为 2.9V，其他的代码很好理解，最后下载验证一下。

28.2.4 下载验证

下载代码后，默认 LCD 屏会显示"PVD Voltage OK!"，当供电电压过低，则 LED1 会点亮，并且 LCD 屏会显示 PVD Low Voltage!。当开发板供电正常，LED1 会熄灭，LCD 屏会继续显示"PVD Voltage OK!"。

28.3 睡眠模式实验

本小节我们来学习睡眠模式实验，该部分的知识点内容请回顾 28.1.2.3 电源管理。我们直接从寄存器介绍开始。

28.3.1 EXTI 寄存器

本实验我们用到外部中断来唤醒睡眠模式。进入睡眠模式很简单，直接调用内核指令 WFI（参考 28.1.3 电源管理对这个指令的讲解）即可进入。用外部中断唤醒，就要在进入睡眠模式前，先对外部中断进行配置。例如，使能中断线（EXTI_IMR），使用何种触发模式（EXTI_FTSR/EXTI_RTSR）。当中断触发（即 EXTI_PR 中某位能查询到 1），跳转到中断服务函数里，最后还得手动清除该标记即（EXTI_PR 中对某位进行置 1 处理）。

● EXTI 中断屏蔽寄存器（EXTI_IMR）

EXTI 中断屏蔽寄存器描述如图 28.3.1.1 所示：

| | | | | | | | | | | | | | | | |
|-------|------|---|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | MR19 | MR18 | MR17 | MR16 |
| | | | | | | | | | | | | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MR15 | MR14 | MR13 | MR12 | MR11 | MR10 | MR9 | MR8 | MR7 | MR6 | MR5 | MR4 | MR3 | MR2 | MR1 | MR0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位19:0 | | MRx: 线x上的中断屏蔽 (Interrupt Mask on line x) 0: 屏蔽来自线x上的中断请求; 1: 开放来自线x上的中断请求。 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.3.1.1 EXTI_IMR 寄存器

本实验使用 WK_UP（PA0）唤醒，即 EXTI0 线中断，所以在外部中断服务函数要把 MR0 位置 1。

● EXTI 上升沿触发选择寄存器（EXTI_RTSR）

EXTI 上升沿触发选择寄存器描述如图 28.3.1.2 所示：

| | | | | | | | | | | | | | | | |
|-------|------|---|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | TR19 | TR18 | TR17 | TR16 |
| | | | | | | | | | | | | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TR15 | TR14 | TR13 | TR12 | TR11 | TR10 | TR9 | TR8 | TR7 | TR6 | TR5 | TR4 | TR3 | TR2 | TR1 | TR0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位18:0 | | TRx: 线x上的上升沿触发事件配置位 (Rising trigger event configuration bit of line x) 0: 禁止输入线x上的上升沿触发(中断和事件) 1: 允许输入线x上的上升沿触发(中断和事件) 注: 位19只适用于互联型产品, 对于其它产品为保留位。 | | | | | | | | | | | | | |

图 28.3.1.2 EXTI_RTSR 寄存器

我们要使用到 EXTI0 线中断，所以 TR0 位要置 1，即 EXTI0 使用的是上升沿进行触发。

● EXTI 挂起寄存器（EXTI_PR）

EXTI 挂起寄存器描述如图 28.3.1.3 所示：

| | | | | | | | | | | | | | | | |
|---|------|------|------|------|------|-----|-----|-----|-----|-----|-----|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | PR19 | PR18 | PR17 | PR16 |
| | | | | | | | | | | | | rc | wl | rc | wl |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PR15 | PR14 | PR13 | PR12 | PR11 | PR10 | PR9 | PR8 | PR7 | PR6 | PR5 | PR4 | PR3 | PR2 | PR1 | PR0 |
| rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl | rc | wl |
| <div> <div>位18:0</div> <div> PRx: 挂起位 (Pending bit) 0: 没有发生触发请求 1: 发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件, 该位被置'1'。在该位中写入'1'可以清除它, 也可以通过改变边沿检测的极性清除。 注: 位19只适用于互联型产品, 对于其它产品为保留位。 </div> </div> | | | | | | | | | | | | | | | |

图 28.3.1.3 EXTI_PR 寄存器

在 EXTI0 中断服务函数里面, 需要清除 EXTI0 中断标记, 即对 PR0 位写 1。

28.3.2 硬件设计

1. 例程功能

LED0 闪烁, 表明代码正在运行。按下按键 KEY0 后, LED1 点亮, 提示进入睡眠模式, 此时 LED0 不再闪烁, 说明已经进入睡眠模式。按下按键 WK_UP 后, LED1 熄灭, 提示退出睡眠模式, 此时 LED0 继续闪烁, 说明已经退出睡眠模式。

2. 硬件资源

- 1) LED 灯
LED0 - PB5 LED1 - PE5
- 2) 独立按键
KEY0 - PE4 WK_UP - PA0
- 3) 电源管理(低功耗模式 - 睡眠模式)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F103 的内部资源, 只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入睡眠模式, 再通过 WK_UP 触发 EXTI 中断来唤醒 CPU。LED0 指示程序是否执行, LED1 指示 CPU 是否进入睡眠模式。

28.3.3 程序设计

28.3.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnterSLEEPMode 函数

进入睡眠模式函数, 其声明如下:

```
void HAL_PWR_EnterSLEEPMode (uint32_t Regulator, uint8_t SLEEPEntry);
```

- **函数描述:**
用于设置 CPU 进入睡眠模式。
- **函数形参:**
形参 1 指定稳压器的状态。有两个选择, PWR_MAINREGULATOR_ON 表示稳压器处于正常模式, PWR_LOWPOWERREGULATOR_ON 表示稳压器处于低功耗模式。对应的是 PWR_CR 寄存器的 LPDS 位的设置 (该形参在该函数中没有实用用处)。

形参 2 指定进入睡眠模式的方式。有两个选择，PWR_SLEEPENTRY_WFI 表示使用 WFI 指令，PWR_SLEEPENTRY_WFE 表示使用 WFE 指令。我们选择前者，这两个指令的区别，请参考本教程 28.1.3 电源管理的描述。

- 函数返回值：
无

睡眠模式配置步骤

1) 配置唤醒睡眠模式的方式

这里我们用外部中断的方式唤醒睡眠模式，所以这里需要配置一个外部中断功能，我们用 WK_UP 按键作为中断触发源，接下来就是配置 PA0（连接按键 WK_UP）。

通过 `_HAL_RCC_GPIOA_CLK_ENABLE` 函数使能 GPIOA 的时钟。

通过 `HAL_GPIO_Init` 函数配置 PA0 为上升沿触发检测的外部中断模式，开启下拉电阻等。

通过 `HAL_NVIC_EnableIRQ` 函数使能 EXTI0 中断。

通过 `HAL_NVIC_SetPriority` 函数设置中断优先级。

编写 `EXTI0_IRQHandler` 中断函数，在中断服务函数中调用 `HAL_GPIO_EXTI_IRQHandler` 函数。

最后编写 `HAL_GPIO_EXTI_Callback` 回调函数。由于前面已经介绍过外部中断的配置步骤，这里就介绍到这里，详见本例程源码。

2) 进入 CPU 睡眠模式

通过 `HAL_PWR_EnterSLEEPMode` 函数进入睡眠模式。

3) 通过按下按键触发外部中断唤醒睡眠模式

在本实验中，通过按下 KEY0 按键进入睡眠模式，然后通过按下 WK_UP 按键触发外部中断唤醒睡眠模式。

28.3.3.2 程序流程图

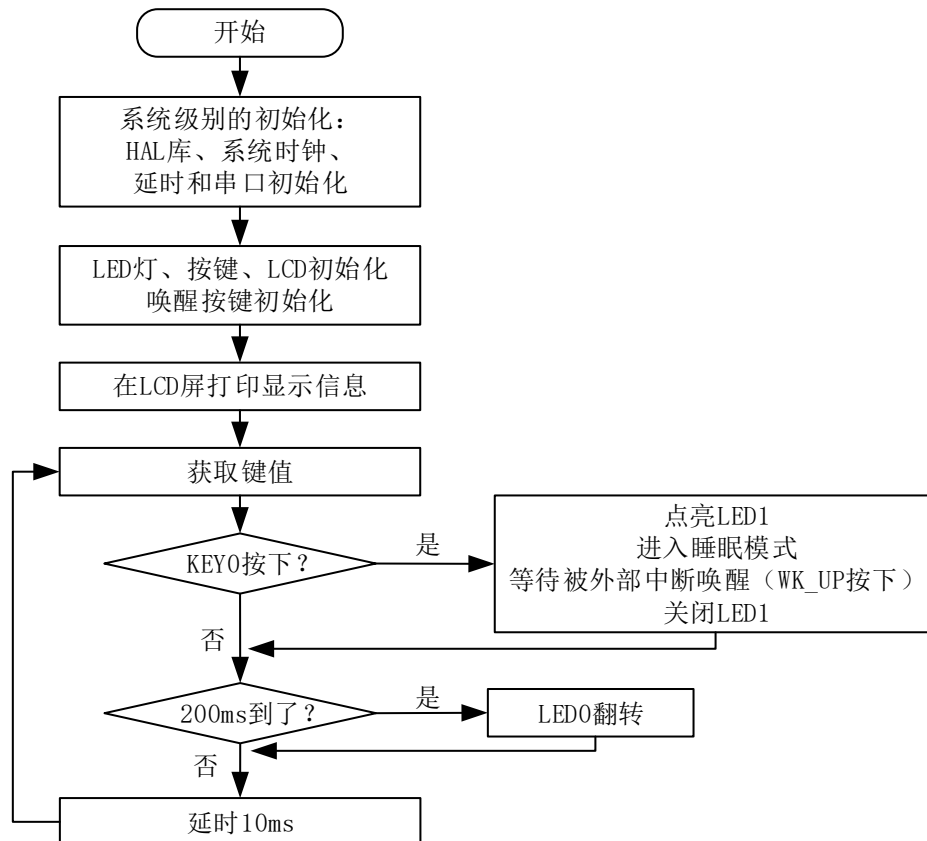


图 28.3.3.2.1 睡眠模式实验程序流程图

28.3.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。睡眠模式实验代码在电压监控实验源码后追加。

首先看本实验在 pwr.h 头文件定义的几个宏定义：

```
/* PWR WKUP 按键 引脚和中断 定义
 * 我们通过 WK_UP 按键唤醒 MCU， 因此必须定义这个按键及其对应的中断服务函数
 */
#define PWR_WKUP_GPIO_PORT      GPIOA
#define PWR_WKUP_GPIO_PIN      GPIO_PIN_0
#define PWR_WKUP_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE();}while(0)

#define PWR_WKUP_INT_IRQn      EXTI0_IRQn
#define PWR_WKUP_INT_IRQHandler EXTI0_IRQHandler
```

这些定义是 WK_UP 按键的相关宏定义，以及其对应的外部中断线 0 的相关定义。

pwr.h 头文件就介绍这部分的程序，下面是 pwr.c 文件，先看低功耗模式下的按键初始化函数，其定义如下：

```
/**
 * @brief      低功耗模式下的按键初始化 (用于唤醒睡眠模式/停止模式)
 * @param      无
 * @retval     无
 */
void pwr_wkup_key_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    PWR_WKUP_GPIO_CLK_ENABLE(); /* WKUP 时钟使能 */

    gpio_init_struct.Pin = PWR_WKUP_GPIO_PIN; /* WKUP 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_IT_RISING; /* 中断, 上升沿 */
    gpio_init_struct.Pull = GPIO_PULLDOWN; /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(PWR_WKUP_GPIO_PORT, &gpio_init_struct); /* WKUP 引脚初始化 */

    HAL_NVIC_SetPriority(PWR_WKUP_INT_IRQn, 2, 2); /* 抢占优先级 2, 子优先级 2 */
    HAL_NVIC_EnableIRQ(PWR_WKUP_INT_IRQn);
}
```

该函数初始化 WK_UP 按键 (PA0)，并设置上升沿触发的外部中断线 0，最后设置中断优先级并使能外部中断线 0。

下面介绍的是进入 CPU 睡眠模式函数，其定义如下：

```
/**
 * @brief      进入 CPU 睡眠模式
 * @param      无
 * @retval     无
 */
void pwr_enter_sleep(void)
{
    /* 进入睡眠模式 */
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
}
```

函数内直接调用 HAL_PWR_EnterSLEEPMode 函数使用 WFI 指令进入睡眠模式。

下面介绍的是 WK_UP 按键外部中断服务函数及其回调函数，函数定义如下：

```
/**
 * @brief      WK_UP 按键 外部中断服务程序
 * @param      无
 * @retval     无
 */
void PWR_WKUP_INT_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(PWR_WKUP_GPIO_PIN);
}
```

```

}

/**
 * @brief      外部中断回调函数
 * @param      GPIO_Pin: 中断线引脚
 * @note       此函数会被 PWR_WKUP_INT_IRQHandler() 调用
 * @retval     无
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == PWR_WKUP_GPIO_PIN)
    {
        /* HAL_GPIO_EXTI_IRQHandler() 函数已经为我们清除了中断标志位，
         * 所以我们进了回调函数可以只关注中断时的控制逻辑，不用再操作寄存器 */
    }
}

```

在 WK_UP 按键外部中断服务函数中我们调用 HAL 库的 HAL_GPIO_EXTI_IRQHandler 函数来处理外部中断。该函数会调用 __HAL_GPIO_EXTI_CLEAR_IT 函数取消屏蔽对应的外部中断线位，这里是 EXTI_IMR 寄存器相应位，还有其他寄存器控制其他外部中断线。我们只是唤醒睡眠模式而已，不需要其他的逻辑程序，所以 HAL_GPIO_EXTI_Callback 回调函数可以什么都不用做，甚至也可以不重新定义这个回调函数（屏蔽该回调函数也可以）。

最后在 main.c 里面编写如下代码：

```

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    pwr_wkup_key_init(); /* 唤醒按键初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "SLEEP TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter SLEEP MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:Exit SLEEP MODE", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            LED1(0); /* 点亮绿灯，提示进入睡眠模式 */
            pwr_enter_sleep(); /* 进入睡眠模式 */
            LED1(1); /* 关闭绿灯，提示退出睡眠模式 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE(); /* 每 200ms，翻转一次 LED0 */
        }

        delay_ms(10);
        t++;
    }
}

```

该部分程序，功能就是按下 KEY0 后，点亮 LED1，进入睡眠模式。然后一直等待外部中断唤醒，当按下按键 WK_UP，就触发外部中断，睡眠模式就被唤醒，然后继续执行后面的程序，关闭 LED1 等。

28.3.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入睡眠模式，此时 LED0 不再闪烁，说明已经进入睡眠模式。按下按键 WK_UP 后，LED1 熄灭，提示退出睡眠模式，此时 LED0 继续闪烁，说明已经退出睡眠模式。

28.4 停止模式实验

本小节我们来学习停止模式实验，该部分的知识点内容请回顾 28.1.2.3 电源管理。我们直接从寄存器介绍开始。

28.4.1 PWR 寄存器

本实验我们用到外部中断来唤醒停止模式。我们用到 WFI 指令进入停止模式，这个后面会讲，进入停止模式后，使用外部中断唤醒。外部中断部分内容参照睡眠模式即可，都是共用同样的配置。

下面主要介绍 PWR_CR 寄存器相关位。

● PWR 控制寄存器（PWR_CR）

PWR 的控制寄存器描述如图 28.4.1.1 所示：

| | | | | | | | | | | | | | | | |
|-----|----|--|----|----|----|----|----|-----|----------|----|------|------|-------|-------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | DBP | PLS[2:0] | | PVDE | CSBF | CWUF | PDDS | LPDS |
| | | | | | | | | rw | rw | rw | rw | rw | rc_wl | rc_wl | rw |
| 位 1 | | PDDS ：掉电深睡眠 与LPDS位协同操作 0：当CPU进入深睡眠时进入停机模式，调压器的状态由LPDS位控制。 1：CPU进入深睡眠时进入待机模式。 | | | | | | | | | | | | | |
| 位 0 | | LPDS ：深睡眠下的低功耗 PDDS=0时，与PDDS位协同操作 0：在停机模式下电压调压器开启 1：在停机模式下电压调压器处于低功耗模式 | | | | | | | | | | | | | |

图 28.4.1.1 PWR_CR 寄存器

通过 PDDS 位选择进入停止模式还是待机模式，停止模式即对 PDDS 位置 0 即可。在停止模式下，电压调节器有两种模式：开启或者低功耗，选择低功耗模式，即 LPDS 置 1。

● 系统控制寄存器（SCB_SCR）

系统控制寄存器描述如图 28.4.1.2 所示：

| 位段 | 名称 | 类型 | 复位值 | 描述 |
|----|-------------|-----|-----|---|
| 4 | SEVONPEND | RW | - | 发生异常悬起时请发送事件，用于在一个新的中断悬起时从 WFE 指令处唤醒。不管这个中断的优先级是否比当前的高，都唤醒。如果没有 WFE 导致睡眠，则下次使用 WFE 时将立即唤醒 |
| 3 | 保留 | - | - | - |
| 2 | SLEEPDEEP | R/W | 0 | 当进入睡眠模式时，使能外部的 SLEEPDEEP 信号，以允许停止系统时钟 |
| 1 | SLEEPONEXIT | R/W | - | 激活“SleepOnExit”功能 |
| 0 | 保留 | - | - | - |

图 28.4.1.2 系统控制寄存器

该寄存器存在于 ARM 内核中，详细描述可查阅《Cortex-M3 权威指南》，在本实验中，我们需要把 SLEEPDEEP 位置 1，这样子后面调用 WFI 命令时，进入的就是停止模式了。在唤醒后，需要清除 SLEEPDEEP 位，进行置 0。

28.4.2 硬件设计

1. 例程功能

LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入停止模式，此时 LED0 不再闪烁，说明已经进入停止模式。按下按键 WK_UP 后，LED1 熄灭，提示退出停止模式，此时 LED0 继续闪烁，说明已经退出停止模式。

2. 硬件资源

- 1) LED 灯
 - LED0 - PB5
 - LED1 - PE5
- 2) 独立按键
 - KEY0 - PE4
 - WK_UP - PA0
- 3) 电源管理(低功耗模式 - 停止模式)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入停止模式，再通过 WK_UP 触发 EXTI 中断来唤醒 CPU。LED0 指示程序是否执行，LED1 指示 CPU 是否进入停止模式。

28.4.3 程序设计

28.4.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnterSTOPMode 函数

进入停止模式函数，其声明如下：

```
void HAL_PWR_EnterSTOPMode (uint32_t Regulator, uint8_t STOPEntry);
```

- **函数描述：**
用于设置 CPU 进入停止模式。
- **函数形参：**
 - 形参 1** 指定稳压器在停止模式下的状态。有两个选择，PWR_MAINREGULATOR_ON 表示稳压器处于正常模式，PWR_LOWPOWERREGULATOR_ON 表示稳压器处于低功耗模式。对应的是 PWR_CR1 寄存器的 LPDS 位的设置。
 - 形参 2** 指定用 WFI 还是 WFE 指令进入停止模式。有两个选择，PWR_STOPENTRY_WFI 表示使用 WFI 指令，PWR_STOPENTRY_WFE 表示使用 WFE 指令。我们选择前者，不了解这两种指令的区别，可以回看 28.1.3 小节的知识。
- **函数返回值：**
无

停止模式配置步骤

1) 配置唤醒停止模式的方式

这里我们用外部中断的方式唤醒停止模式，所以这里需要配置一个外部中断功能，我们用 WK_UP 按键作为中断触发电源，接下来就是配置 PA0（连接按键 WK_UP）。

通过 `_HAL_RCC_GPIOA_CLK_ENABLE` 函数使能 GPIOA 的时钟。

通过 `HAL_GPIO_Init` 函数配置 PA0 为上升沿触发检测的外部中断模式，开启下拉电阻等。

通过 `HAL_NVIC_EnableIRQ` 函数使能 EXTI0 中断。

通过 `HAL_NVIC_SetPriority` 函数设置中断优先级。

编写 EXTI0_IRQHandler 中断函数,在中断服务函数中调用 HAL_GPIO_EXTI_IRQHandler。

最后编写 HAL_GPIO_EXTI_Callback 回调函数。由于前面已经介绍过外部中断的配置步骤,这里就介绍到这里,详见本例程序源码。

2) 进入 CPU 停止模式

通过 HAL_PWR_EnterSTOPMode 函数进入停止模式。

3) 通过按下按键触发外部中断唤醒停止模式

在本实验中,通过按下 KEY0 按键进入停止模式,然后通过按下 WK_UP 按键触发外部中断唤醒停止模式。

28.4.3.2 程序流程图

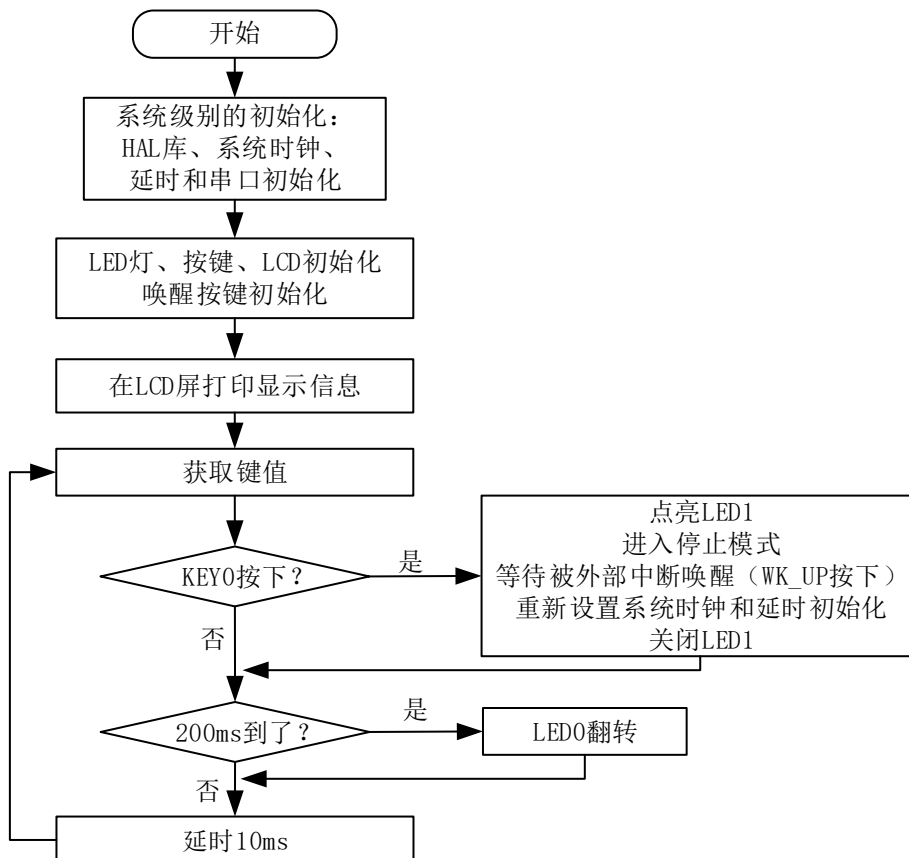


图 28.4.3.2.1 停止模式实验程序流程图

28.4.3.3 程序解析

这里我们只讲解核心代码,详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件: pwr.c 和 pwr.h。停止模式实验代码在睡眠模式实验源码后追加。

首先看 pwr.h 头文件,因为我们还是用到 WK_UP 对应的外部中断线来唤醒停止模式的 CPU, pwr.h 头文件的 WK_UP 按键对应的宏定义我们也是用到的,上个实验已经讲过,这里不再赘述。下面是 pwr.c 文件, WK_UP 按键的相关函数我们还是用上个实验的,我们主要介绍进入停止模式函数,其定义如下:

```

/**
 * @brief      进入停止模式
 * @param      无
 * @retval     无
 */
void pwr_enter_stop(void)
{
    __HAL_RCC_PWR_CLK_ENABLE();

```

```

/* 进入停止模式，设置稳压器为低功耗模式，等待中断唤醒 */
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}

```

该函数因为涉及对电源控制寄存器的操作，所以先调用 __HAL_RCC_PWR_CLK_ENABLE 函数使能 PWR 时钟，然后调用 HAL_PWR_EnterSTOPMode 函数进入停止模式，形参 1 即 PWR_LOWPOWERREGULATOR_ON 设置稳压器为低功耗模式，形参 2 则是选择 WFI 指令。

最后在 main.c 里面编写如下代码：

```

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    pwr_wkup_key_init(); /* 唤醒按键初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "STOP TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter STOP MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:Exit STOP MODE", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            LED1(0); /* 点亮绿灯,提示进入停止模式 */
            pwr_enter_stop(); /* 进入停止模式 */

            /* 从停止模式唤醒，需要重新设置系统时钟，72Mhz */
            sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
            delay_init(72); /* 延时初始化 */
            LED1(1); /* 关闭绿灯,提示退出停止模式 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGGLE(); /* 每 200ms,翻转一次 LED0 */
        }
        delay_ms(10);
        t++;
    }
}

```

该部分程序，功能就是按下 KEY0 后，点亮 LED1，进入停止模式。然后一直等待外部中断唤醒，当按下按键 WK_UP，就触发外部中断，停止模式就被唤醒，然后继续执行后面的程序，重新设置系统时钟 72MHZ 和延时初始化，关闭 LED1 等。

28.4.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入停止模式，此时 LED0 不再闪烁，说明已经进入停止模式。按下按键 WK_UP 后，LED1 熄灭，提示退出停止模式，此时 LED0 继续闪烁，说明已经退出停止模式。

28.5 待机模式实验

本小节我们来学习待机模式实验，该部分的知识点内容请回顾 28.1.2.3 电源管理。我们直接从寄存器介绍开始。

28.5.1 PWR 寄存器

本实验是先对相关的电源控制寄存器配置待机模式的参数，然后通过 WFI 指令进入待机模式，使用 WKUP 引脚的上升沿来唤醒（这是特定的唤醒源）。

下面主要介绍本实验用到的寄存器，系统控制寄存器的部分与上一实验一致，就不介绍了

● PWR 控制寄存器（PWR_CR）

PWR 的控制寄存器描述如图 28.5.1.1 所示：

| | | | | | | | | | | | | | | | |
|-----|----|--|----|----|----|----|----|-----|----------|----|----|------|-------|-------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | DBP | PLS[2:0] | | | PVDE | CSBF | CWUF | PDDS |
| | | | | | | | | rw | rw | rw | rw | rw | rc_wl | rc_wl | rw |
| 位 2 | | CWUF ：清除唤醒位 始终读出为0 0：无功效 1：2个系统时钟周期后清除WUF唤醒位(写) | | | | | | | | | | | | | |
| 位 1 | | PDDS ：掉电深睡眠 与LPDS位协同操作 0：当CPU进入深睡眠时进入停机模式，调压器的状态由LPDS位控制。 1：CPU进入深睡眠时进入待机模式。 | | | | | | | | | | | | | |

图 28.5.1.1 PWR_CR 寄存器

这里我们通过设置 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时，我们需要通过 CWUF 位清除之前的唤醒位。

● 电源控制/状态寄存器（PWR_CSR）

电源控制/状态寄存器描述如图 28.5.1.2 所示：

| | | | | | | | | | | | | | | | |
|-----|----|---|----|----|----|----|------|----|----|----|----|------|-----|-----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | EWUP | 保留 | | | | PVDO | SBF | WUF | |
| | | | | | | | rw | | | | | r | r | r | |
| 位 8 | | EWUP: 使能WKUP引脚 0: WKUP引脚为通用I/O。WKUP引脚上的事件不能将CPU从待机模式唤醒 1: WKUP引脚用于将CPU从待机模式唤醒，WKUP引脚被强置为输入下拉的配置(WKUP引脚上的上升沿将系统从待机模式唤醒) 注：在系统复位时清除这一位。 | | | | | | | | | | | | | |

图 28.5.1.1 PWR_CSR 寄存器

该寄存器我们只关心 EWUP 位，设置 EWUP 为 1，即 WKUP 引脚作为待机模式的唤醒源。

28.5.2 硬件设计

1. 例程功能

LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，进入待机模式，待机模式下大部分引脚处于高阻态，所以说这时候 LED0 会熄灭，TFTLCD 也会熄灭。按下 WK_UP 按键后，退

出待机模式（相当于复位操作），程序重新执行，LED0 继续闪烁，TFTLCD 屏点亮。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
WK_UP - PA0
- 3) 电源管理(低功耗模式 - 待机模式)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F103 的内部资源，只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入待机模式，再通过 WK_UP 上升沿来唤醒 CPU。LED0 指示程序是否执行。

28.5.3 程序设计

28.5.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnableWakeUpPin 函数

使能唤醒引脚函数，其声明如下：

```
void HAL_PWR_EnableWakeUpPin (uint32_t WakeUpPinPolarity);
```

- **函数描述：**
用于使能唤醒引脚。
- **函数形参：**
形参 1 取值范围：PWR_WAKEUP_PIN1。
- **函数返回值：**
无
- **注意事项：**
禁止某个唤醒引脚使用的函数如下：

```
void HAL_PWR_DisableWakeUpPin (uint32_t WakeUpPinPolarity);
```

2. HAL_PWR_EnterSTANDBYMode 函数

进入待机模式函数，其声明如下：

```
void HAL_PWR_EnterSTANDBYMode (void);
```

- **函数描述：**
用于使 CPU 进入待机模式，进入待机模式，首先要设置 SLEEPDEEP 位，接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 上升沿的到来。
- **函数形参：**
无
- **函数返回值：**
无

待机模式配置步骤

1) 进入 CPU 待机模式

在进入待机模式之前我们需要做一些准备：

涉及到操作 PWR 寄存器的内容，所以首先先进行 PWR 时钟的初始化，用 __HAL_RCC_PWR_CLK_ENABLE 函数实现。

通过 HAL_PWR_EnableWakeUpPin 函数使能 WKUP 的唤醒功能。

通过 __HAL_PWR_CLEAR_FLAG 函数清除唤醒标记，详看源码。

通过 HAL_PWR_EnterSTANDBYMode 函数进入待机模式。

2) 通过 WKUP 引脚上升沿触发唤醒睡眠模式

在本实验中，通过按下 KEY0 按键进入待机模式，然后通过按下 WK_UP 按键，使用待机模式中的 WKUP 引脚上升沿的唤醒信号，而不是普通的外部中断，唤醒待机模式。

28.5.3.2 程序流程图

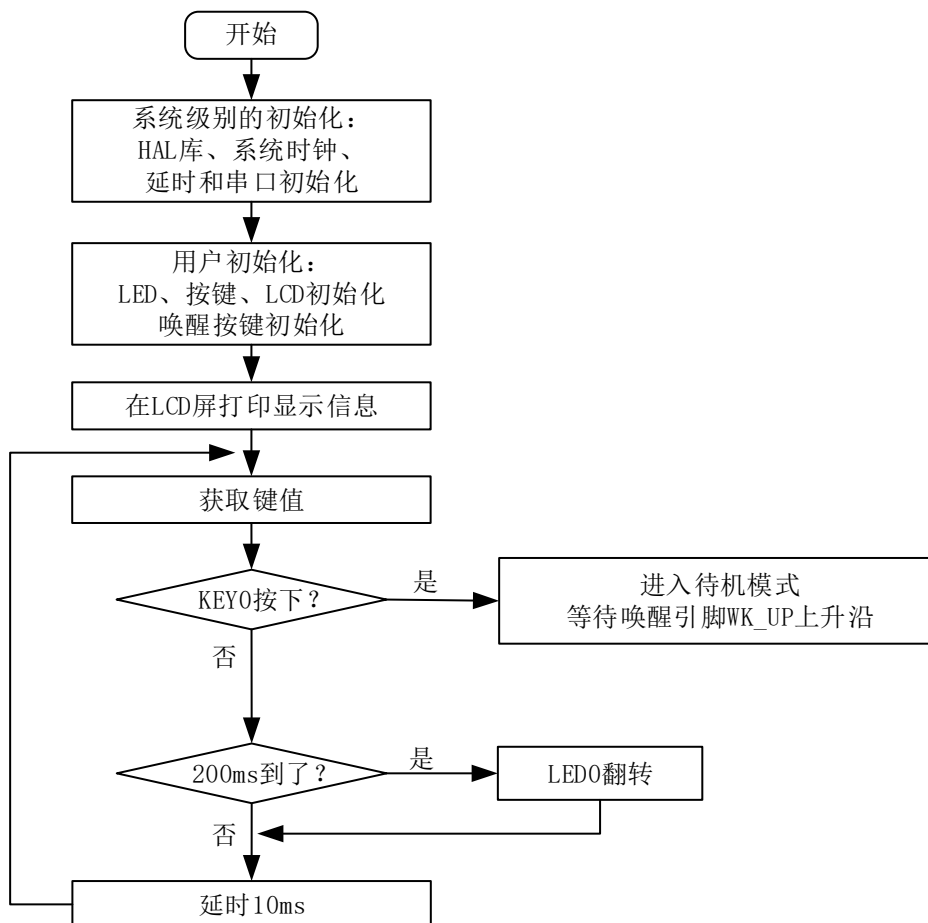


图 28.4.3.2.1 待机模式实验程序流程图

28.5.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。待机模式实验代码在停止模式实验源码后追加。

pwr.h 头文件上的宏定义我们是没有用到的，这里我们使用的是特定唤醒源，与外部中断无关。下面是 pwr.c 文件，我们主要介绍进入待机模式函数，其定义如下：

```

/**
 * @brief      进入待机模式
 * @param      无
 * @retval     无
 */
void pwr_enter_standby(void)
{
    __HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 */

    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); /* 使能 KEY_UP 引脚的唤醒功能 */
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU); /* 需要清此标记，否则将保持唤醒状态 */
    HAL_PWR_EnterSTANDBYMode(); /* 进入待机模式 */
}

```

该函数首先是调用 `__HAL_RCC_PWR_CLK_ENABLE` 来使能 PWR 时钟，然后调用函数 `HAL_PWR_EnableWakeUpPin` 用来设置 WK_UP 引脚作为唤醒源。在进入待机模式前，还得调用 `__HAL_PWR_CLEAR_FLAG` 函数清除一下唤醒标志，要不然会保持唤醒状态。最后调用函数 `HAL_PWR_EnterSTANDBYMode` 进入待机模式。

最后在 main.c 里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    pwr_wkup_key_init(); /* 唤醒按键初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "STANDBY TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter STANDBY MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:Exit STANDBY MODE", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            pwr_enter_standby(); /* 进入待机模式 */
            /* 从待机模式唤醒相当于系统重启(复位), 因此不会执行到这里 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE(); /* 每 200ms, 翻转一次 LED0 */
        }

        delay_ms(10);
        t++;
    }
}
```

该部分程序，经过一系列初始化后，判断到 KEY0 按下就调用 `pwr_enter_standby` 函数进入待机模式，然后等待按下 WK_UP 按键产生 WKUP 上升沿唤醒 CPU。注意待机模式唤醒后，系统会进行复位。

28.5.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，TFTLCD 屏熄灭，此时 LED0 不再闪烁，说明已经进入待机模式。按下按键 WK_UP 后，TFTLCD 屏点亮，LED0 闪烁，说明系统从待机模式中唤醒相当于复位。

第二十九章 DMA 实验

本章,我们将介绍 STM32F103 的 DMA。我们将利用 DMA 来实现串口数据传送,并在 LCD 模块上显示当前的传送进度。

本章分为如下几个小节:

29.1 DMA 简介

29.2 硬件设计

29.3 程序设计

29.4 下载验证

29.1 DMA 简介

DMA, 全称为: Direct Memory Access, 即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输, 也没有中断处理方式那样保留现场和恢复现场的过程, 通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路, 能使 CPU 的效率大为提高。

STM32F103 内部有 2 个 DMA 控制器(DMA2 仅存大容量产品中), DMA1 有 7 个通道。DMA2 有 5 个通道。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁器来协调各个 DMA 请求的优先权。

STM32F103 的 DMA 有以下是一些特性:

- 每个通道都直接连接专用的硬件 DMA 请求, 每个通道都同样支持软件触发。这些功能通过软件来配置。
- 在七个请求间的优先权可以通过软件编程设置(共有四级: 很高、高、中等和低), 假如在相等优先权时由硬件决定(请求 0 优先于请求 1, 依此类推)。
- 独立的源和目标数据区的传输宽度(字节、半字、全字), 模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。
- 支持循环的缓冲器管理。
- 每个通道都有 3 个事件标志(DMA 半传输, DMA 传输完成和 DMA 传输出错), 这 3 个事件标志逻辑或成为一个单独的中断请求。
- 存储器和存储器间的传输。
- 外设和存储器, 存储器和外设的传输。
- 闪存、SRAM、外设的 SRAM、APB1、APB2 和 AHB 外设均可作为访问的源和目标。
- 可编程的数据传输数目: 最大为 65536。

29.1.1 DMA 框图

STM32F103ZET6 有两个 DMA 控制器, DMA1 和 DMA2, 本章, 我们仅针对 DMA1 进行介绍。

下面先来学习 DMA 控制器框图, 通过学习 DMA 控制器框图会有一个很好的整体掌握, 同时对之后的编程也会有一个清晰的思路。STM32F103 的 DMA 控制器框图如图 29.1.1.1 所示:

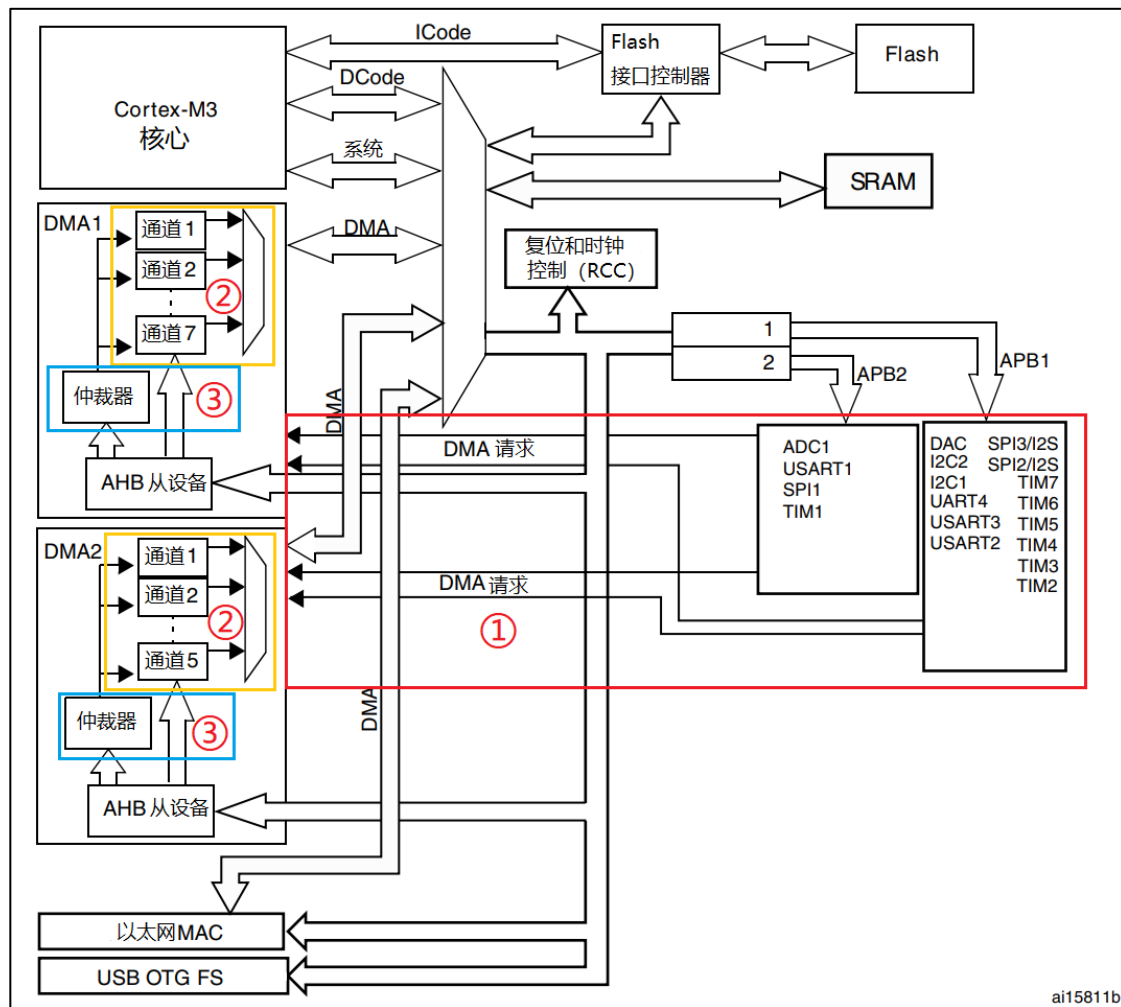


图 29.1.1.1 DMA 控制器框图

图中，我们标记了 3 处位置，起作用分别是：

① DMA 请求

如果外设想要通过 DMA 来传输数据，必须先给 DMA 控制器发送 DMA 请求，DMA 收到请求信号之后，控制器会给外设一个应答信号，当外设应答后且 DMA 控制器收到应答信号之后，就会启动 DMA 的传输，直到传输完毕。

STM32F103 共有 DMA1 和 DMA2 两个控制器，DMA1 有 7 个通道，DMA2 有 5 个通道，不同的 DMA 控制器的通道对应着不同的外设请求，这决定了我们在软件编程上该怎么设置，具体见表 29.1.1.1DMA 请求映像表。

| 外设 | 通道1 | 通道2 | 通道3 | 通道4 | 通道5 | 通道6 | 通道7 |
|----------------------|----------|-----------|---------------------|-----------------------------------|-------------|-----------------------|----------------------|
| ADC1 | ADC1 | | | | | | |
| SPI/I ² S | | SPI1_RX | SPI1_TX | SPI/I2S2_RX | SPI/I2S2_TX | | |
| USART | | USART3_TX | USART3_RX | USART1_TX | USART1_RX | USART2_RX | USART2_TX |
| I ² C | | | | I2C2_TX | I2C2_RX | I2C1_TX | I2C1_RX |
| TIM1 | | TIM1_CH1 | TIM1_CH2 | TIM1_TX4 TIM1_TRIG TIM1_COM | TIM1_UP | TIM1_CH3 | |
| TIM2 | TIM2_CH3 | TIM2_UP | | | TIM2_CH1 | | TIM2_CH2 TIM2_CH4 |
| TIM3 | | TIM3_CH3 | TIM3_CH4 TIM3_UP | | | TIM3_CH1 TIM3_TRIG | |
| TIM4 | TIM4_CH1 | | | TIM4_CH2 | TIM4_CH3 | | TIM4_UP |

表 29.1.1.1 DMA1 请求映像表

| 外设 | 通道1 | 通道2 | 通道3 | 通道4 | 通道5 |
|---------------------|-----------------------|-----------------------------------|--------------------|--------------------|----------|
| ADC3 ⁽¹⁾ | | | | | ADC3 |
| SPI/I2S3 | SPI/I2S3_RX | SPI/I2S3_TX | | | |
| UART4 | | | UART4_RX | | UART4_TX |
| SDIO ⁽¹⁾ | | | | SDIO | |
| TIM5 | TIM5_CH4 TIM5_TRIG | TIM5_CH3 TIM5_UP | | TIM5_CH2 | TIM5_CH1 |
| TIM6/ DAC通道1 | | | TIM6_UP/ DAC通道1 | | |
| TIM7/ DAC通道2 | | | | TIM7_UP/ DAC通道2 | |
| TIM8 ⁽¹⁾ | TIM8_CH3 TIM8_UP | TIM8_CH4 TIM8_TRIG TIM8_COM | TIM8_CH1 | | TIM8_CH2 |

表 29.1.1.2 DMA2 请求映像

② 通道

DMA 具有 12 个独立可编程的通道，其中 DMA1 有 7 个通道，DMA2 有 5 个通道，每个通道对应不同的外设的 DMA 请求。虽然每个通道可以接收多个外设的请求，但是同一时间只能接收一个，不能同时接收多个。

③ 仲裁器

当发生多个 DMA 通道请求时，就意味着有先后响应处理的顺序问题，这个就由仲裁器管理。仲裁器管理 DMA 通道请求分为两个阶段。第一阶段属于软件阶段，可以在 DMA_CCRx 寄存器中设置，有 4 个等级：非常高，高，中和低四个优先级。第二阶段属于硬件阶段，如果两个或以上的 DMA 通道请求设置的优先级一样，则他们优先级取决于通道编号，编号越低优先权越高，比如通道 0 高于通道 1。在大容量产品和互联型产品中，DMA1 控制器拥有高于 DMA2 控制器的优先级。

29.1.2 DMA 寄存器

● DMA 中断状态寄存器（DMA_ISR）

DMA 中断状态寄存器描述如图 29.1.2.1 所示：

| | | | | | | | | | | | | | | | |
|---------------------------|-------|-------|------|--|-------|-------|------|-------|-------|-------|------|-------|-------|-------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | TEIF7 | HTIF7 | TCIF7 | GIF7 | TEIF6 | HTIF6 | TCIF6 | GIF6 | TEIF5 | HTIF5 | TCIF5 | GIF5 |
| | | | | r | r | r | r | r | r | r | r | r | r | r | r |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TEIF4 | HTIF4 | TCIF4 | GIF4 | TEIF3 | HTIF3 | TCIF3 | GIF3 | TEIF2 | HTIF2 | TCIF2 | GIF2 | TEIF1 | HTIF1 | TCIF1 | GIF1 |
| r | | | | r | r | r | r | r | r | r | r | r | r | r | r |
| 位27, 23, 19, 15, 11, 7, 3 | | | | TEIFx : 通道x的传输错误标志(x = 1 ... 7) (Channel x transfer error flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有传输错误(TE); 1: 在通道x发生了传输错误(TE)。 | | | | | | | | | | | |
| 位26, 22, 18, 14, 10, 6, 2 | | | | HTIFx : 通道x的半传输标志(x = 1 ... 7) (Channel x half transfer flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有半传输事件(HT); 1: 在通道x产生了半传输事件(HT)。 | | | | | | | | | | | |
| 位25, 21, 17, 13, 9, 5, 1 | | | | TCIFx : 通道x的传输完成标志(x = 1 ... 7) (Channel x transfer complete flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有传输完成事件(TC); 1: 在通道x产生了传输完成事件(TC)。 | | | | | | | | | | | |
| 位24, 20, 16, 12, 8, 4, 0 | | | | GIFx : 通道x的全局中断标志(x = 1 ... 7) (Channel x global interrupt flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0: 在通道x没有TE、HT或TC事件; 1: 在通道x产生了TE、HT或TC事件。 | | | | | | | | | | | |

图 29.1.2.1 DMA_ISR 寄存器

该寄存器是查询当前 DMA 传输的状态，我们常用的是 TCIFx 位，即通道 DMA 传输完成与否的标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除。

● DMA 中断标志清除寄存器 (DMA IFCR)

DMA 中断标志清除寄存器描述如图 29.1.2.2 所示:

| | | | | | | | | | | | | | | | |
|------------|------------|------------|-----------|---------------------------------|--|------------|-----------|------------|------------|------------|-----------|------------|------------|------------|-----------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | CTEIF 7 | CHTIF 7 | CTCIF 7 | CGIF 7 | CTEIF 6 | CHTIF 6 | CTCIF 6 | CGIF 6 | CTEIF 5 | CHTIF 5 | CTCIF 5 | CGIF 5 |
| | | | | 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 |
| CTEIF 4 | CHTIF 4 | CTCIF 4 | CGIF 4 | CTEIF 3 | CHTIF 3 | CTCIF 3 | CGIF 3 | CTEIF 2 | CHTIF 2 | CTCIF 2 | CGIF 2 | CTEIF 1 | CHTIF 1 | CTCIF 1 | CGIF 1 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| | | | | 位27, 23, 19, 15, 11, 7, 3 | CTEIFx: 清除通道x的传输错误标志(x = 1 ... 7) (Channel x transfer error clear) 这些位由软件设置和清除。 0: 不起作用 | | | | | | | | | | |

图 29.1.2.2 DMA IFCR 寄存器

该寄存器是用来清除 DMA_ISR 的对应位的，通过写 0 清除。在 DMA_ISR 被置位后，我们必须通过向该寄存器对应的位写 0 来清除。

● DMA 通道 x 传输数量寄存器 (DMA CNDTR_x)

DMA 通道 x 传输数量寄存器描述如图 29.1.2.3 所示:

| | | | | | | | | | | | | | | | |
|-----------|-----|--|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| NDT[15:0] | | | | | | | | | | | | | | | |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 位15:0 | | <p>NDT[15:0]: 数据传输数量 (Number of data to transfer)</p> <p>数据传输数量为0至65535。这个寄存器只能在通道不工作(DMA_CCRx的EN=0)时写入。通道开启后该寄存器变为只读，指示剩余的待传输字节数目。寄存器内容在每次DMA传输后递减。</p> <p>数据传输结束后，寄存器的内容或者变为0；或者当该通道配置为自动重新加载模式时，寄存器的内容将被自动重新加载为之前配置时的数值。</p> <p>当寄存器的内容为0时，无论通道是否开启，都不会发生任何数据传输。</p> | | | | | | | | | | | | | |

图 29.1.2.3 DMA CNDTR 寄存器

该寄存器控制着 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值随着传输的进行而减少, 当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成。可以通过这个寄存器的值来获取当前 DMA 传输的进度。

● DMA 通道 x 配置寄存器 (DMA CCR_x)

DMA 通道 x 配置寄存器描述如图 29.1.2.4 所示:

| | | | | | | | | | | | | | | | |
|--------|---|---------|------------|------------|------|------|------|-----|------|------|------|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | MEM2 MEM | PL[1:0] | MSIZE[1:0] | PSIZE[1:0] | MINC | PINC | CIRC | DIR | TEIE | HTIE | TCIE | EN | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位14 | MEM2MEM: 存储器到存储器模式 (Memory to memory mode) 该位由软件设置和清除。 0: 非存储器到存储器模式; 1: 启动存储器到存储器模式。 | | | | | | | | | | | | | | |
| 位13:12 | PL[1:0]: 通道优先级 (Channel priority level) 这些位由软件设置和清除。 00: 低 01: 中 10: 高 11: 最高 | | | | | | | | | | | | | | |
| 位11:10 | MSIZE[1:0]: 存储器数据宽度 (Memory size) 这些位由软件设置和清除。 00: 8位 01: 16位 10: 32位 11: 保留 | | | | | | | | | | | | | | |
| 位9:8 | PSIZE[1:0]: 外设数据宽度 (Peripheral size) 这些位由软件设置和清除。 00: 8位 01: 16位 10: 32位 11: 保留 | | | | | | | | | | | | | | |
| 位7 | MINC: 存储器地址增量模式 (Memory increment mode) 该位由软件设置和清除。 0: 不执行存储器地址增量操作 1: 执行存储器地址增量操作 | | | | | | | | | | | | | | |
| 位6 | PINC: 外设地址增量模式 (Peripheral increment mode) 该位由软件设置和清除。 0: 不执行外设地址增量操作 1: 执行外设地址增量操作 | | | | | | | | | | | | | | |
| 位5 | CIRC: 循环模式 (Circular mode) 该位由软件设置和清除。 0: 不执行循环操作 1: 执行循环操作 | | | | | | | | | | | | | | |
| 位4 | DIR: 数据传输方向 (Data transfer direction) 该位由软件设置和清除。 0: 从外设读 1: 从存储器读 | | | | | | | | | | | | | | |
| 位3 | TEIE: 允许传输错误中断 (Transfer error interrupt enable) 该位由软件设置和清除。 0: 禁止TE中断 1: 允许TE中断 | | | | | | | | | | | | | | |
| 位2 | HTIE: 允许半传输中断 (Half transfer interrupt enable) 该位由软件设置和清除。 0: 禁止HT中断 1: 允许HT中断 | | | | | | | | | | | | | | |
| 位1 | TCIE: 允许传输完成中断 (Transfer complete interrupt enable) 该位由软件设置和清除。 0: 禁止TC中断 1: 允许TC中断 | | | | | | | | | | | | | | |
| 位0 | EN: 通道开启 (Channel enable) 该位由软件设置和清除。 0: 通道不工作 1: 通道开启 | | | | | | | | | | | | | | |

图 29.1.2.4 DMA_CCRx 寄存器

该寄存器控制着 DMA 很多相关信息, 包括数据宽度、外设及存储器宽度、通道优先级、增量模式、传输方向、中断允许、使能等, 所以说 DMA_CCRx 是 DMA 传输的核心控制寄存器。

● **DMA 通道 x 外设地址寄存器 (DMA_CPARx)**

DMA 通道 x 外设地址寄存器描述如图 29.1.2.5 所示:

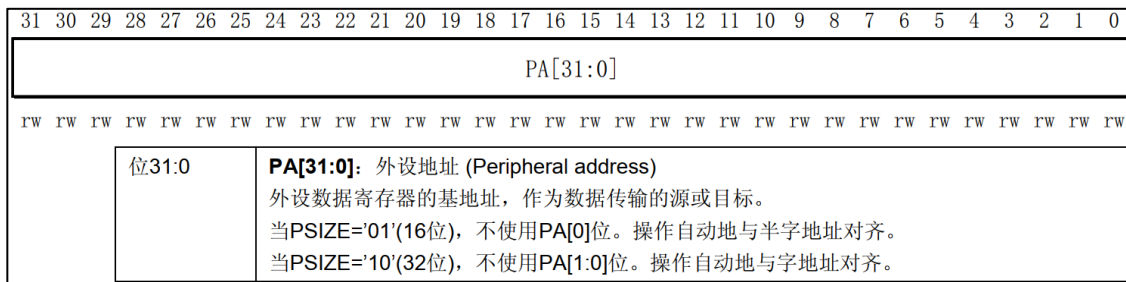


图 29.1.2.5 DMA_CPARx 寄存器

该寄存器是用来存储 STM32 外设的地址，比如我们平常使用串口 1，那么该寄存器必须写入 0x40013804（其实就是&USART1_DR）。其他外设就可以修改成其他对应外设地址就好了。

● DMA 通道 x 存储器地址寄存器 (DMA_CMARx)

DMA 通道 x 存储器地址寄存器用来存放存储器的地址，该寄存器和 DMA_CPARx 差不多，所以就不列出来了。举个应用的例子，在程序中，我们使用到一个 g_sendbuf[5200]数组来做存储器，那么我们在 DMA_CMARx 中写入&g_sendbuf 即可。

29.2 硬件设计

1. 例程功能

每按下按键 KEY0，串口 1 就会以 DMA 方式发送数据，同时在 LCD 上面显示传送进度。打开串口调试助手，可以收到 DMA 发送的内容。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键 KEY0 - PE4
- 3) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

DMA 属于 STM32F103 内部资源，通过软件设置好就可以了。

29.3 程序设计

29.3.1 DMA 的 HAL 库驱动

DMA 在 HAL 库中的驱动代码在 stm32f1xx_hal_dma.c 文件（及其头文件）中。

1. HAL_DMA_Init 函数

DMA 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

- **函数描述：**
用于初始化 DMA1，DMA2。
- **函数形参：**

形参 1 是 DMA_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct __DMA_HandleTypeDef
{
    void *Instance; /* 寄存器基地址 */
    DMA_InitTypeDef Init; /* DAM 通信参数 */
    HAL_LockTypeDef Lock; /* DMA 锁对象 */
    __IO HAL_DMA_StateTypeDef State; /* DMA 传输状态 */
    void *Parent; /* 父对象状态，HAL 库处理的中间变量 */
}
```

```
void (*XferCpltCallback)( struct __DMA_HandleTypeDef *hdma); /*DMA 传输完成回调*/
/* DMA 一半传输完成回调 */
void (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
/* DMA 传输完整的 Memory1 回调 */
void (* XferM1CpltCallback)( struct __DMA_HandleTypeDef * hdma);
/* DMA 传输半完全内存回调 */
void (* XferM1HalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
/*DMA 传输错误回调*/
void (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
/* DMA 传输中止回调 */
void (* XferAbortCallback)( struct __DMA_HandleTypeDef * hdma);
__IO uint32_t          ErrorCode;          /* DMA 存取错误代码 */
DMA_TypeDef            *DmaBaseAddress;    /* DMA 通道基地址 */
uint32_t               ChannelIndex;       /* DMA 通道索引 */
}DMA_HandleTypeDef;
```

这个结构体内容比较多，下面列出几个成员说明一下。

Instance: 是用来设置寄存器基地址，例如要设置的对象是串口 1 的发送，那么就要参考表 29.1.1.1，串口 1 的 DMA 传输需要用到的是 DMA1 的通道 4，即 DMA1_Channel4。

Parent: 是 HAL 库处理中间变量，用来指向 DMA 通道外设句柄。

XferCpltCallback（传输完成回调函数），XferHalfCpltCallback（半传输完成回调函数），XferM1CpltCallback（Memory1 传输完成回调函数）和 XferErrorCallback（传输错误回调函数）是四个函数指针，用来指向回调函数入口地址。

其他成员变量是 HAL 库处理过程状态标识变量，这里就不做过多讲解。

接下来我们重点介绍 Init，它是 DMA_InitTypeDef 结构体类型变量，该结构体定义如下：

```
typedef struct
{
    uint32_t Direction;          /* 传输方向，例如存储器到外设 DMA_MEMORY_TO_PERIPH */
    uint32_t PeriphInc;          /* 外设（非）增量模式，非增量模式 DMA_PINC_DISABLE */
    uint32_t MemInc;             /* 存储器（非）增量模式，增量模式 DMA_MINC_ENABLE */
    uint32_t PeriphDataAlignment; /* 外设数据大小：8/16/32 位 */
    uint32_t MemDataAlignment;   /* 存储器数据大小：8/16/32 位 */
    uint32_t Mode;               /* 模式：循环模式/普通模式 */
    uint32_t Priority;           /* DMA 优先级：低/中/高/非常高 */
}DMA_InitTypeDef;
```

该结构体成员变量非常多，但每个成员变量的配置基本都是 DMA_CCRx 寄存器的相关位。

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

以 DMA 的方式传输串口数据的配置步骤

1) 使能 DMA 时钟。

DMA 的时钟使能是通过 AHB1ENR 寄存器来控制的，这里我们要先使能时钟，才可以配置 DMA 相关寄存器。HAL 库方法为：

```
HAL_RCC_DMA1_CLK_ENABLE(); /* DMA1 时钟使能 */
HAL_RCC_DMA2_CLK_ENABLE(); /* DMA2 时钟使能 */
```

2) 初始化 DMA。

调用 HAL_DMA_Init 函数初始化 DMA 的相关参数，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。例如要使用串口 DMA 发送，所以方式为：

```
HAL_LINKDMA(&g_uart1_handler, hdmatx, g_dma_handle);
```

其中 g_uart1_handler 是串口初始化句柄，我们在 usart.c 中定义过了。g_dma_handle 是 DMA 初始化句柄。hdmatx 是外设句柄结构体的成员变量，在这里实际就是 g_uart1_handler 的成员变量。在 HAL 库中，任何一个可以使用 DMA 的外设，它的初始化结构体句柄都会有一个

DMA_HandleTypeDef 指针类型的成员变量，是 HAL 库用来做相关指向的。hdmatx 就是 DMA_HandleTypeDef 结构体指针类型。

这句话的含义就是把 g_uart1_handler 句柄的成员变量 hdmatx 和 DMA 句柄 g_dma_handle 连接起来，是纯软件处理，没有任何硬件操作。

这里我们就点到为止，如果大家要详细了解 HAL 库指向关系，请查看本实验宏定义标识符 __HAL_LINKDMA 的定义和调用方法，就会很清楚了。

3) 使能串口的 DMA 发送，启动传输。

串口 1 的 DMA 发送实际是串口控制寄存器 CR3 的位 7 来控制的，在 HAL 库中操作该寄存器来使能串口 DMA 发送的函数为 HAL_UART_Transmit_DMA。

这里大家需要注意，调用该函数后会开启相应的 DMA 中断，对于本章实验，我们是通过查询的方法获取数据传输状态，所以并没有做中断相关处理，也没有编写中断服务函数。

HAL 库还提供了对串口的 DMA 发送的停止，暂停，继续等操作函数：

```
HAL_StatusTypeDef HAL_UART_DMAStop(UART_HandleTypeDef *huart);    /* 停止 */
HAL_StatusTypeDef HAL_UART_DMAPause(UART_HandleTypeDef *huart);   /* 暂停 */
HAL_StatusTypeDef HAL_UART_DMAResume(UART_HandleTypeDef *huart);  /* 恢复 */
```

4) 查询 DMA 传输状态。

在 DMA 传输过程中，我们要查询 DMA 传输通道的状态，使用的方法是通过检测 DMA 寄存器的相关位实现：

```
HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TC4);
```

获取当前传输剩余数据量：

```
HAL_DMA_GET_COUNTER(&g_dma_handle);
```

同样，我们也可以设置对应的 DMA 数据流传输的数据量大小，函数为：

```
HAL_DMA_SET_COUNTER(&g_dma_handle, 1000);
```

DMA 相关的库函数我们就讲解到这里，大家可以查看“资料盘/8,STM32 参考资料/1,STM32CubeF1 固件包”路径下的“HAL 库说明手册”详细了解。

5) DMA 中断使用方法。

DMA 中断对于每个通道都有一个中断服务函数，比如 DMA1_Channel4 的中断服务函数为 DMA1_Channel4_IRQHandler。HAL 库提供了通用 DMA 中断处理函数 HAL_DMA_IRQHandler，在该函数内部，会对 DMA 传输状态进行分析，然后调用相应的中断处理回调函数：

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);    /* 发送完成回调函数 */
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart); /* 发送一半回调函数 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);    /* 接收完成回调函数 */
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart); /* 接收一半回调函数 */
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart);     /* 传输出错回调函数 */
```

29.3.2 程序流程图

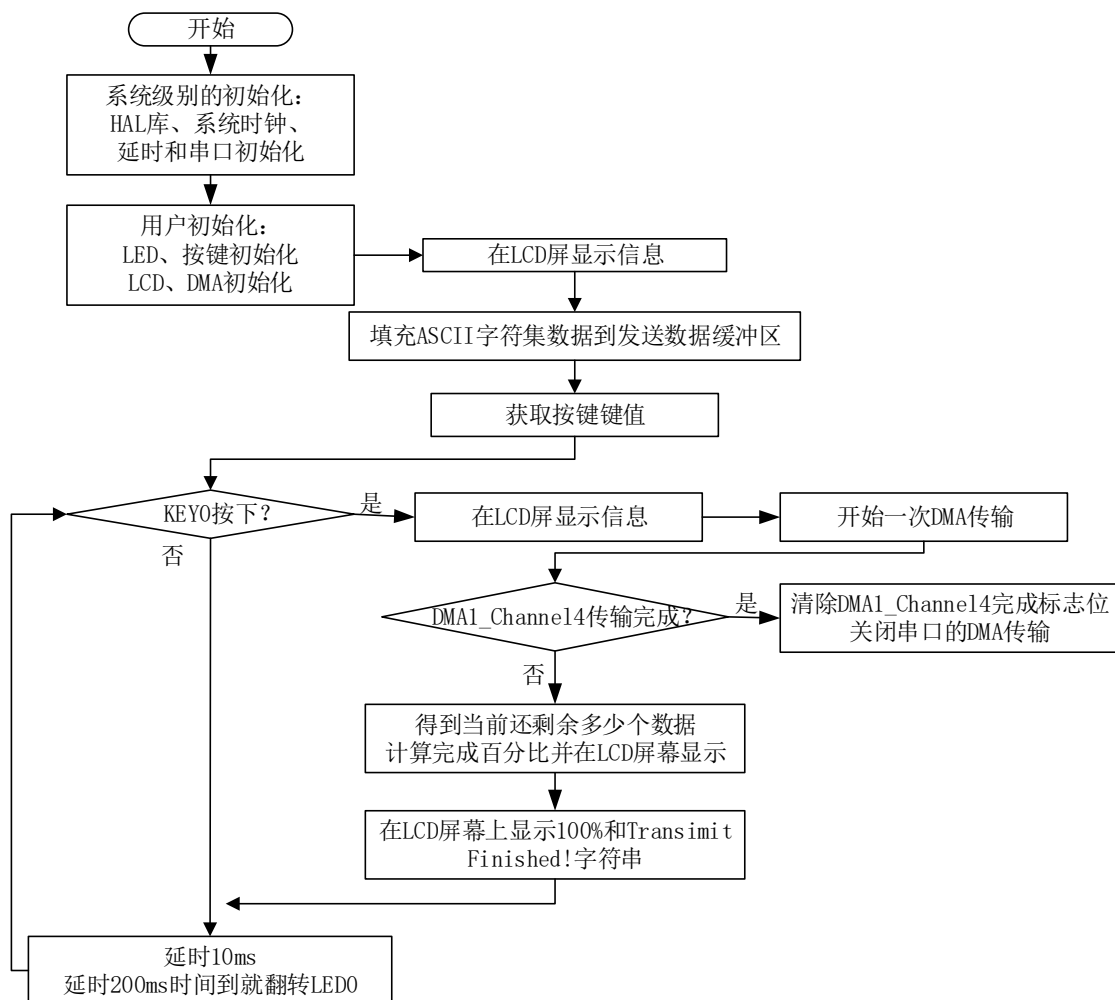


图 30.3.2.1 DMA 实验程序流程图

29.3.3 程序解析

1. DMA 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DMA 驱动源码包括两个文件：dma.c 和 dma.h。

dma.h 头文件只有函数得声明，就不解释了，我们直接介绍 dma.c 的程序。下面是与 DMA 初始化相关的函数，其定义如下：

```

/**
 * @brief      串口 TX DMA 初始化函数
 * @note      这里的传输形式是固定的，这点要根据不同的情况来修改
 *            从存储器 -> 外设模式/8 位数据宽度/存储器增量模式
 *
 * @param      dma_chy      : DMA 的通道，DMA1_Channel1 ~ 7，DMA2_Channel1 ~ 5
 *            某个外设对应哪个 DMA，哪个通道，请参考<<STM32 中文参考手册>> 10.3.7 节
 *            必须设置正确的 DMA 及通道，才能正常使用！
 * @retval     无
 */
void dma_init(DMA_Channel_TypeDef* DMAx_CHx)
{
    /* 大于 DMA1_Channel7，则为 DMA2 的通道了 */
    if ((uint32_t)DMAx_CHx > (uint32_t)DMA1_Channel7)
    {

```

```

    __HAL_RCC_DMA2_CLK_ENABLE();          /* DMA2 时钟使能 */
}
else
{
    __HAL_RCC_DMA1_CLK_ENABLE();          /* DMA1 时钟使能 */
}
/* 将 DMA 与 USART1 联系起来(发送 DMA) */
__HAL_LINKDMA(&g_uart1_handle, hdmatrix, g_dma_handle);

/* Tx DMA 配置 */
g_dma_handle.Instance = DMAx_CHx; /* USART1_TX 的 DMA 通道: DMA1_Channel4 */
g_dma_handle.Init.Direction = DMA_MEMORY_TO_PERIPH; /* 存储器到外设模式 */
g_dma_handle.Init.PeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
g_dma_handle.Init.MemInc = DMA_MINC_ENABLE; /* 存储器增量模式 */
g_dma_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE; /* 外设位宽 */
g_dma_handle.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE; /* 存储器位宽 */
g_dma_handle.Init.Mode = DMA_NORMAL; /* DMA 模式: 正常模式 */
g_dma_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */

HAL_DMA_Init(&g_dma_handle);
}

```

该函数是一个通用的 DMA 配置函数, DMA1/DMA2 的所有通道, 都可以利用该函数配置, 不过有些固定参数可能要适当修改(比如位宽、传输方向等)。该函数在外部只能修改 DMA 数据通道, 更多的其他设置只能在该函数内部修改。对照前面的配置步骤的详细讲解来分析这部分代码即可。

2. main.c 代码

在 main.c 里面编写如下代码:

```

/* 要循环发送的字符串 */
const uint8_t TEXT_TO_SEND[] = {"正点原子 STM32 DMA 串口实验"};
#define SEND_BUF_SIZE (sizeof(TEXT_TO_SEND) + 2) * 200 /* 发送数据长度 */
uint8_t g_sendbuf[SEND_BUF_SIZE]; /* 发送数据缓冲区 */

int main(void)
{
    uint8_t key = 0;
    uint16_t i, k;
    uint16_t len;
    uint8_t mask = 0;
    float pro = 0; /* 进度 */

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    dma_init(DMA1_Channel4); /* 初始化串口 1 TX DMA */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DMA TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Start", RED);

    len = sizeof(TEXT_TO_SEND);
    k = 0;

    for (i = 0; i < SEND_BUF_SIZE; i++) /* 填充 ASCII 字符集数据 */
    {

```



```

if (k >= len)                                /* 入换行符 */
{
    if (mask)
    {
        g_sendbuf[i] = 0x0a;
        k = 0;
    }
    else
    {
        g_sendbuf[i] = 0x0d;
        mask++;
    }
}
else                                          /* 复制 TEXT_TO_SEND 语句 */
{
    mask = 0;
    g_sendbuf[i] = TEXT_TO_SEND[k];
    k++;
}
}

i = 0;

while (1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)                    /* KEY0 按下 */
    {
        printf("\r\nDMA DATA:\r\n");
        lcd_show_string(30, 130, 200, 16, 16, "Start Transimit...", BLUE);
        lcd_show_string(30, 150, 200, 16, 16, "  %", BLUE);    /* 显示百分号 */

        HAL_UART_Transmit_DMA(&g_uart1_handle, g_sendbuf, SEND_BUF_SIZE);
        /* 等待 DMA 传输完成, 此时我们来做另外一些事情, 比如点灯
        * 实际应用中, 传输数据期间, 可以执行另外的任务
        */
        while (1)
        {
            /* 等待 DMA1_Channel4 传输完成 */
            if ( __HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TC4))
            {
                __HAL_DMA_CLEAR_FLAG(&g_dma_handle, DMA_FLAG_TC4);
                HAL_UART_DMAStop(&uartx_handler);    /* 传输完成以后关闭串口 DMA */
                break;
            }

            pro = __HAL_DMA_GET_COUNTER(&g_dma_handle);
            len = SEND_BUF_SIZE;    /* 总长度 */
            pro = 1 - (pro / len);    /* 得到百分比 */
            pro *= 100;    /* 扩大 100 倍 */
            lcd_show_num(30, 150, pro, 3, 16, BLUE);
        }

        lcd_show_num(30, 150, 100, 3, 16, BLUE);    /* 显示 100% */
        /* 提示传送完成 */
        lcd_show_string(30, 130, 200, 16, 16, "Transimit Finished!", BLUE);
    }
    i++;
    delay_ms(10);

    if (i == 20)
    {
        LED0_TOGGLE();    /* LED0 闪烁, 提示系统正在运行 */
    }
}

```

```
i = 0;
}
}
}
```

main 函数的流程大致是：先初始化发送数据缓冲区 g_sendbuf 的值，然后通过 KEY0 开启串口 DMA 发送，在发送过程中，通过 _HAL_DMA_GET_COUNTER(&g_dma_handle) 获取当前还剩余的数据量来计算传输百分比，最后在传输结束之后清除相应标志位，提示已经传输完成。

29.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 29.4.1 所示：



图 29.4.1 DMA 实验测试图

我们打开串口调试助手，然后按 KEY0，可以看到串口显示如图 29.4.2 所示的内容：

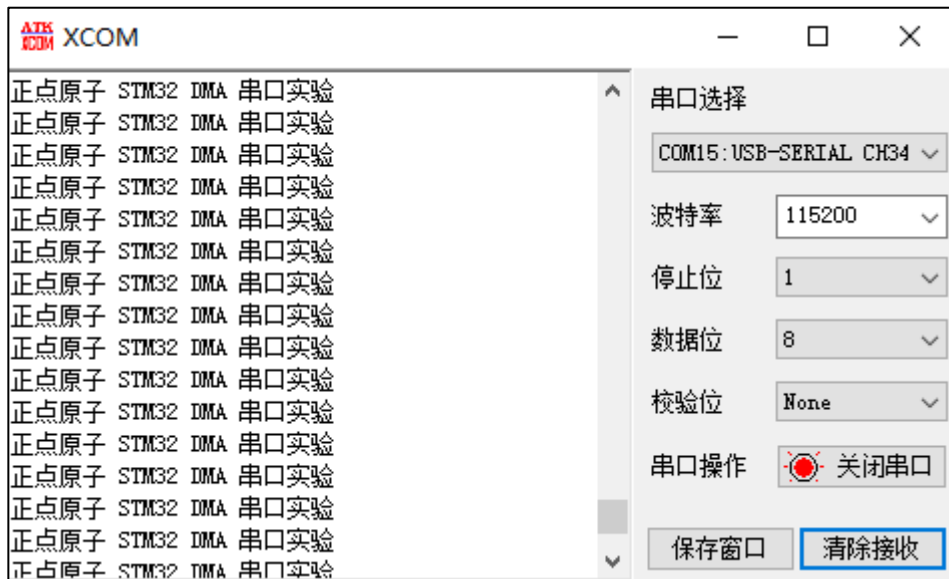


图 29.4.2 串口收到的数据内容

可以看到串口收到了开发板发送过来的数据，同时可以看到 TFTLCD 上显示了进度等信息，如图 29.4.3 所示：

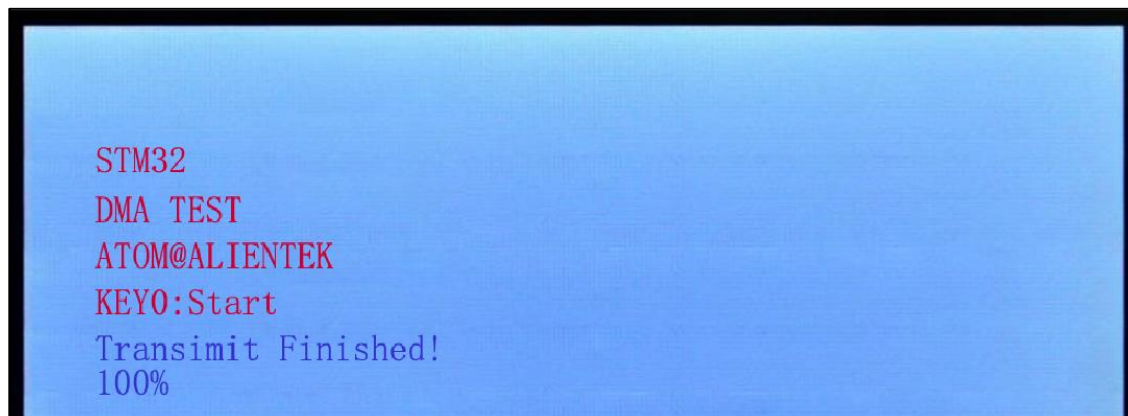


图 29.4.3 DMA 串口数据传输中

至此，我们整个 DMA 实验就结束了，希望大家通过本章的学习，掌握 STM32F103 的 DMA 使用。DMA 是个非常好的功能，它不但能减轻 CPU 负担，还能提高数据传输速度，合理的应用 DMA，往往能让你的程序设计变得简单。

第三十章 ADC 实验

本章，我们将介绍 STM32F103 的 ADC（Analog-to-digital converters，模数转换器）功能。我们通过四个实验来学习 ADC，分别是单通道 ADC 采集实验、单通道 ADC 采集（DMA 读取）实验、多通道 ADC 采集（DMA 读取）实验和单通道 ADC 过采样（16 位分辨率）实验。

本章分为如下几个小节：

- 30.1 ADC 简介
- 30.2 单通道 ADC 采集实验
- 30.3 单通道 ADC 采集（DMA 读取）实验
- 30.4 多通道 ADC 采集（DMA 读取）实验
- 30.5 单通道 ADC 过采样（16 位分辨率）实验

30.1 ADC 简介

ADC 即模拟数字转换器，英文详称 Analog-to-digital converter，可以将外部的模拟信号转换为数字信号。

STM32F103 系列芯片拥有 3 个 ADC（C8T6 只有 2 个），这些 ADC 可以独立使用，其中 ADC1 和 ADC2 还可以组成双重模式（提高采样率）。STM32 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它有 18 个通道，可测量 16 个外部和 2 个内部信号源，其中 ADC3 根据 CPU 引脚的不同其通道数也不同，一般有 8 个外部通道。ADC 中的各个通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以以左对齐或者右对齐存储在 16 位数据寄存器中。

STM32F103 的 ADC 主要特性我们可以总结为以下几条：

- 1、12 位分辨率；
- 2、转换结束、注入转换结束和发生模拟看门狗事件时产生中断
- 3、单次和连续转换模式
- 4、自校准
- 5、带内嵌数据一致性的数据对齐
- 6、采样间隔可以按通道分别编程
- 7、规则转换和注入转换均有外部触发选项
- 8、间断模式
- 9、双重模式（带 2 个或以上 ADC 的器件）
- 10、ADC 转换时间：时钟为 72MHz 为 1.17us
- 11、ADC 供电要求：2.4V 到 3.6V
- 12、ADC 输入范围： $V_{REF-} \leq V_{IN} \leq V_{REF+}$
- 13、规则通道转换期间有 DMA 请求产生

下面来介绍 ADC 的框图：

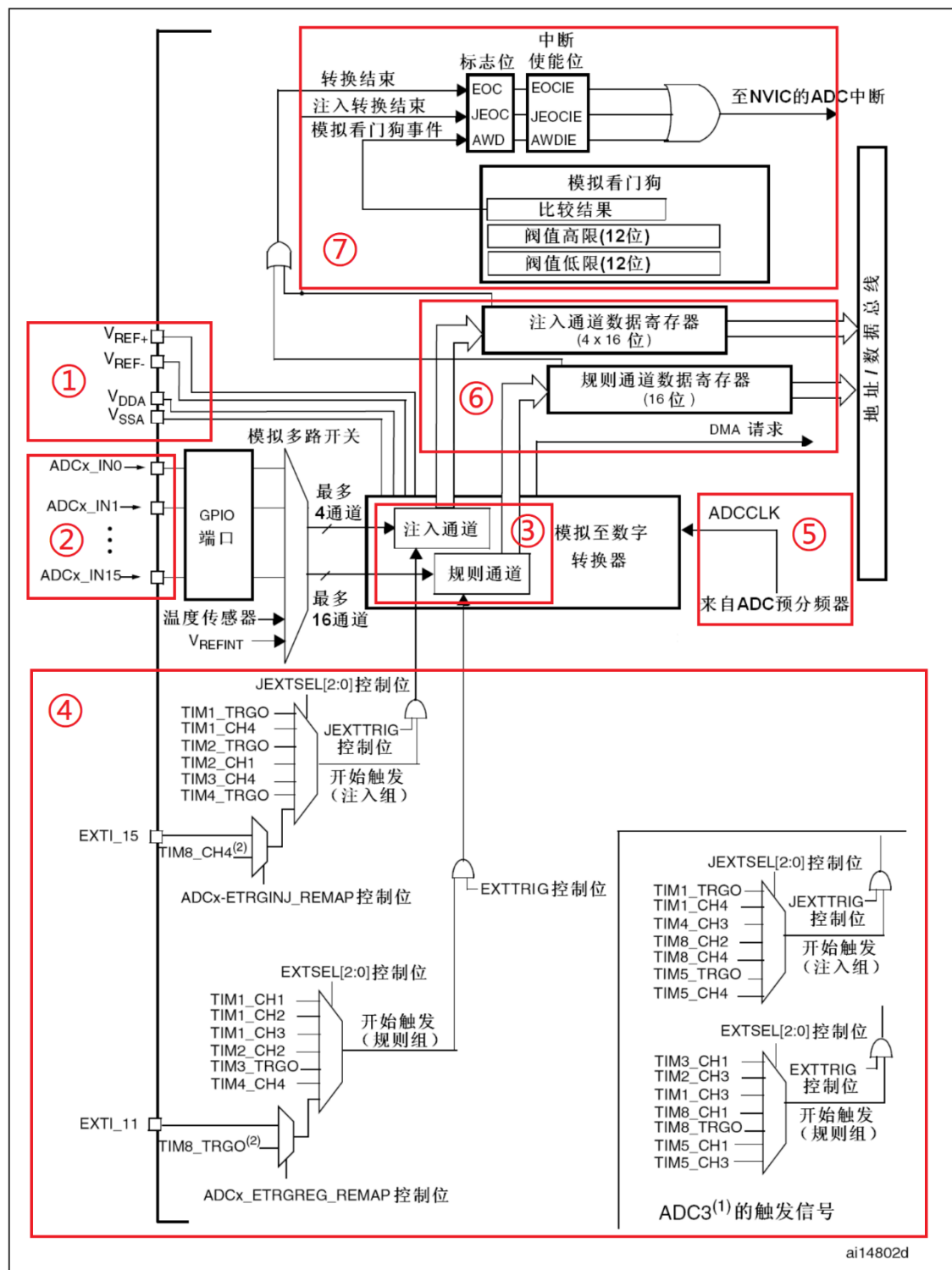


图 30.1.1 ADC 框图

图中，我们按照 ADC 的配置流程标记了七处位置，分别如下：

① 输入电压

在前面 ADC 的主要特性也对输入电压有所提及, ADC 输入范围 $V_{REF-} \leq V_{IN} \leq V_{REF+}$, 最终还是由 V_{REF-} 、 V_{REF+} 、 V_{DDA} 和 V_{SSA} 决定的。下面看一下这几个参数的关系, 如图 30.1.2 所示:

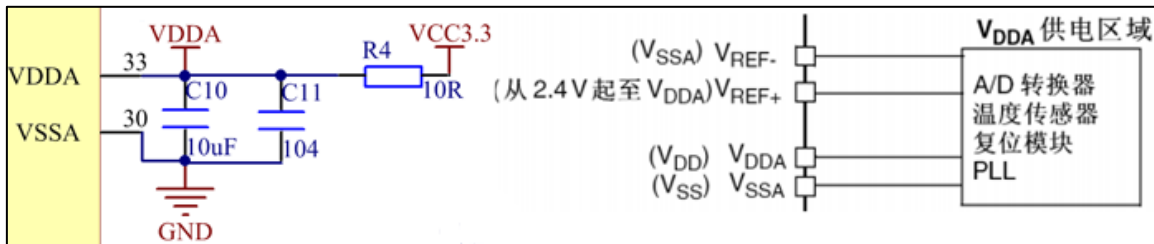


图 30.1.2 参数关系图

从上图可以知道， V_{DDA} 和 V_{REF+} 接 $VCC3.3$ ，而 V_{SSA} 和 V_{REF-} 是接地，所以 ADC 的输入范围即 $0 \sim 3.3V$ 。

② 输入通道

在确定好了 ADC 输入电压后，如何把外部输入的电压输送到 ADC 转换器中呢，在这里引入了 ADC 的输入通道，在前面也提及到了 ADC1 和 ADC2 都有 16 个外部通道和 2 个内部通道；而 ADC3 就有 8 个外部通道。外部通道对应的是上图中的 $ADCx_IN0$ 、 $ADCx_IN1 \dots ADCx_IN15$ 。ADC1 的通道 16 就是内部通道，连接到芯片内部的温度传感器，通道 17 连接到 V_{refint} 。而 ADC2 的通道 16 和 17 连接到内部的 VSS 。ADC3 的通道 9、14、15、16 和 17 连接的是内部的 VSS 。具体的 ADC 通道表见表 30.1.1 所示：

| ADC1 | IO | ADC2 | IO | ADC3 | IO |
|-------|-------------------|-------|----------|-------|----------|
| 通道 0 | PA0 | 通道 0 | PA0 | 通道 0 | PA0 |
| 通道 1 | PA1 | 通道 1 | PA1 | 通道 1 | PA1 |
| 通道 2 | PA2 | 通道 2 | PA2 | 通道 2 | PA2 |
| 通道 3 | PA3 | 通道 3 | PA3 | 通道 3 | PA3 |
| 通道 4 | PA4 | 通道 4 | PA4 | 通道 4 | PF6 |
| 通道 5 | PA5 | 通道 5 | PA5 | 通道 5 | PF7 |
| 通道 6 | PA6 | 通道 6 | PA6 | 通道 6 | PF8 |
| 通道 7 | PA7 | 通道 7 | PA7 | 通道 7 | PF9 |
| 通道 8 | PB0 | 通道 8 | PB0 | 通道 8 | PF10 |
| 通道 9 | PB1 | 通道 9 | PB1 | 通道 9 | 连接内部 VSS |
| 通道 10 | PC0 | 通道 10 | PC0 | 通道 10 | PC0 |
| 通道 11 | PC1 | 通道 11 | PC1 | 通道 11 | PC1 |
| 通道 12 | PC2 | 通道 12 | PC2 | 通道 12 | PC2 |
| 通道 13 | PC3 | 通道 13 | PC3 | 通道 13 | PC3 |
| 通道 14 | PC4 | 通道 14 | PC4 | 通道 14 | 连接内部 VSS |
| 通道 15 | PC5 | 通道 15 | PC5 | 通道 15 | 连接内部 VSS |
| 通道 16 | 连接内部温度传感器 | 通道 16 | 连接内部 VSS | 通道 16 | 连接内部 VSS |
| 通道 17 | 连接内部 V_{refint} | 通道 17 | 连接内部 VSS | 通道 17 | 连接内部 VSS |

表 30.1.1 ADC 通道表

③ 转换顺序

当 ADC 的多个通道以任意顺序进行转换就诞生了成组转换，这里有两种成组转换类型：规则组和注入组。规则组就是图中的规则通道，注入组就是图中的注入通道。为了避免大家对输入通道，以及规则通道和注入通道的理解混淆，后面规则通道以规则组来代称，注入通道以注入组来代称。

规则组最多允许 16 个输入通道进行转换，而注入组最多允许 4 个输入通道进行转换。这里讲解一下规则组和注入组。

规则组（规则通道）

规则组，按字面理解，“规则”就是按照一定的顺序，相当于正常运行的程序，平常用到最多也是规则组。

注入组（注入通道）

注入组，按字面理解，“注入”就是打破原来的状态，相当于中断。当程序执行的时候，中断是可以打断程序的执行。同这个类似，注入组转换可以打断规则组的转换。假如在规则组转换过程中，注入组启动，那么注入组被转换完成之后，规则组才得以继续转换。

为了便于理解，下面看一下规则组和注入组的执行优先级对比图，如图 30.1.3 所示：

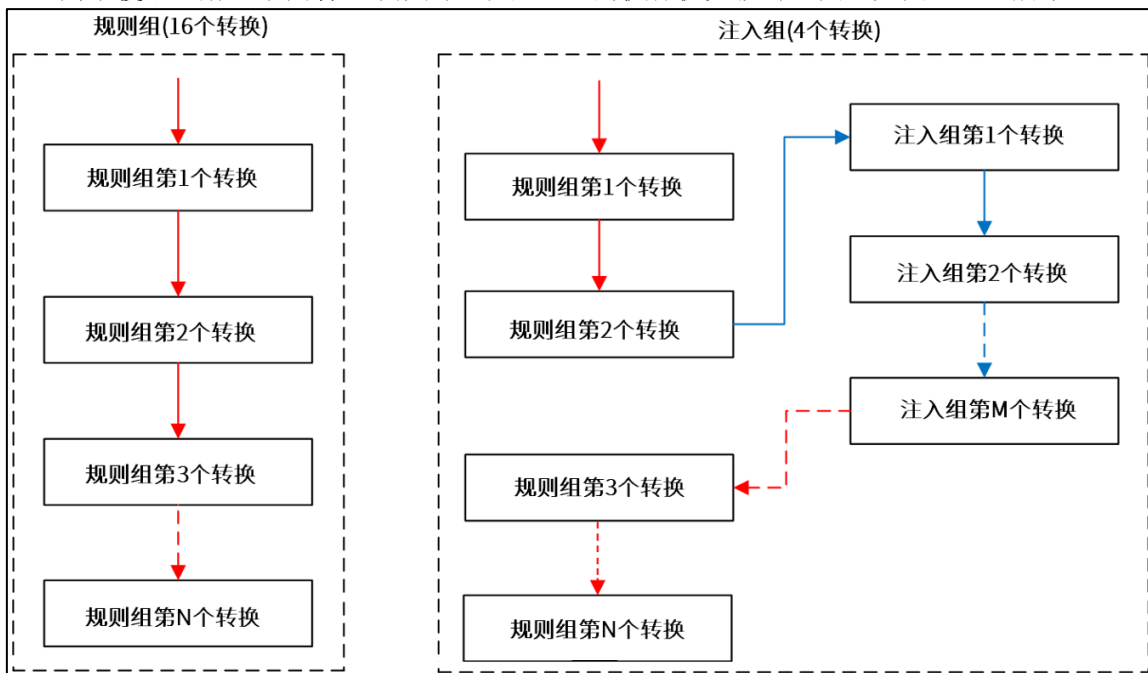


图 30.1.3 规则组和注入组的执行优先级对比图

了解了规则组和注入组的概念后，下面来看看它们的转换顺序，即转换序列。转换序列可以分为规则序列和注入序列。下面分别来介绍它们。

规则序列

规则组最多允许 16 个输入通道进行转换，那么就需要设置通道转换的顺序，即规则序列。规则序列寄存器有 3 个，分别为 SQR1、SQR2 和 SQR3。SQR3 控制规则序列中的第 1 个到第 6 个转换；SQR2 控制规则序列中第 7 个到第 12 个转换；SQR1 控制规则序列中第 13 个到第 16 个转换。规则序列寄存器控制关系汇总如表 30.1.2 所示：

| 规则序列寄存器控制关系汇总 | | | |
|---------------|------------|---------------|---------|
| 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 0~17 |
| | SQ2 [4:0] | 设置第 2 个转换的通道 | 通道 0~17 |
| | SQ3 [4:0] | 设置第 3 个转换的通道 | 通道 0~17 |
| | SQ4 [4:0] | 设置第 4 个转换的通道 | 通道 0~17 |
| | SQ5 [4:0] | 设置第 5 个转换的通道 | 通道 0~17 |
| | SQ6 [4:0] | 设置第 6 个转换的通道 | 通道 0~17 |
| SQR2 | SQ7 [4:0] | 设置第 7 个转换的通道 | 通道 0~17 |
| | SQ8 [4:0] | 设置第 8 个转换的通道 | 通道 0~17 |
| | SQ9 [4:0] | 设置第 9 个转换的通道 | 通道 0~17 |
| | SQ10 [4:0] | 设置第 10 个转换的通道 | 通道 0~17 |
| | SQ11 [4:0] | 设置第 11 个转换的通道 | 通道 0~17 |
| | SQ12 [4:0] | 设置第 12 个转换的通道 | 通道 0~17 |
| SQR1 | SQ13 [4:0] | 设置第 13 个转换的通道 | 通道 0~17 |
| | SQ14 [4:0] | 设置第 14 个转换的通道 | 通道 0~17 |
| | SQ15 [4:0] | 设置第 15 个转换的通道 | 通道 0~17 |
| | SQ16 [4:0] | 设置第 16 个转换的通道 | 通道 0~17 |

| | | | |
|--|----------|---------------|------|
| | SQL[3:0] | 设置规则序列要转换的通道数 | 0~15 |
|--|----------|---------------|------|

表 30.1.2 规则序列寄存器控制关系汇总表

从上表可以知道，当我们想设置 ADC 的某个输入通道在规则序列的第 1 个转换，只需要把相应的输入通道号的值写入 SQR3 寄存器中的 SQ1[4:0]位即可。例如想让输入通道 5 先进行转换，那么就可以把 5 这个数值写入 SQ1[4:0]位。如果还想让输入通道 8 在第 2 个转换，那么就可以把 8 这个数值写入 SQ2[4:0]位。最后还要设置你的这个规则序列的输入通道个数，只需把输入通道个数写入 SQR1 的 SQL[3:0]位。注意：写入 0 到 SQL[3:0]位，表示这个规则序列有 1 个输入通道的意思，而不是 0 个输入通道。

注入序列

注入序列，跟规则序列差不多，决定的是注入组的顺序。注入组最大允许 4 个通道输入，它的注入序列由 JSQR 寄存器配置。注入序列寄存器 JSQR 控制关系如表 30.1.3 所示：

| 注入序列寄存器控制关系汇总 | | | |
|---------------|-----------|---------------|---------|
| 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 0~17 |
| | JSQ2[4:0] | 设置第 2 个转换的通道 | 通道 0~17 |
| | JSQ3[4:0] | 设置第 3 个转换的通道 | 通道 0~17 |
| | JSQ4[4:0] | 设置第 4 个转换的通道 | 通道 0~17 |
| | JL[1:0] | 设置注入序列要转换的通道数 | 0~3 |

表 30.1.3 注入序列寄存器控制关系汇总表

注入序列有多少个输入通道，只需要把输入通道个数写入到 JL[1:0]位，范围是 0~3。注意：写入 0 表示这个注入序列有一个输入通道，而不是 0 个输入通道。这个内容很简单。编程时容易犯错误的是注入序列的转换顺序问题，下面给大家讲解一下。

如果 JL[1:0]位的值小于 3，即设置注入序列要转换的通道个数小于 4，则注入序列的转换顺序是从 JSQx[4:0]（x=4-JL[1:0]）开始。例如：JL[1:0]=10、JSQ4[4:0]=00100、JSQ3[4:0]=00011、JSQ2[4:0]=00111、JSQ1[4:0]=00010，意味着这个注入序列的转换顺序是：7、3、4，而不是 2、7、3。如果 JL[1:0]=00，那么转换顺序是从 JSQ4[4:0]开始。

④ 触发源

在配置好输入通道以及转换顺序后，就可以进行触发转换了。ADC 的触发转换有两种方法：分别是通过 ADON 位或外部事件触发转换。

(1) ADON 位触发转换

当 ADC_CR2 寄存器的 ADON 位为 1 时，再独立给 ADON 位写 1（其它位不能一起改变，这是为了防止误触发），这时会启动转换。这种控制 ADC 启动转换的方式非常简单。

(2) 外部触发转换

另一种方法是通过外部事件触发转换，例如定时器捕获、EXTI 线和软件触发，可以分为规则组外部触发和注入组外部触发。

规则组外部触发使用方法是将 EXTTRIG 位置 1，并且通过 EXTSET[2:0]位选择规则组启动转换的触发源。如果 EXTSET[2:0]位设置为 111，那么可以通过 SWSTART 为启动 ADC 转换，相当于软件触发。

注入组外部触发使用方法是将 JEXTTRIG 位置 1，并且通过 JEXTSET[2:0]位选择注入组启动转换的触发源。如果 JEXTSET[2:0]位设置为 111，那么可以通过 JSWSTART 为启动 ADC 转换，相当于软件触发。

ADC1 和 ADC2 的触发源是一样的，ADC3 的触发源和 ADC1/2 有所不同，这个需要注意。

⑤ 转换时间

(1) ADC 时钟

在学习转换时间之前，我们先来了解 ADC 时钟。从标号⑤框出来部分可以看到 ADC 时钟是要经过 ADC 预分频器的，那么 ADC 的时钟源是什么？ADC 预分频器的分频系数可以设置的范围又是多少？以及 ADC 时钟频率的最大值又是多少？下面将为大家解答。

ADC 的输入时钟是由 PCLK2 经过分频产生的，分频系数是由 RCC_CFGR 寄存器的 ADCPRE[1:0]位设置的，可选择 2/4/8/16 分频。需要注意的是，ADC 的输入时钟频率最大值是 14MHz，如果超过这个值将会导致 ADC 的转换结果准确度下降。

一般我们设置 PCLK2 为 72MHz。为了不超过 ADC 的最大输入时钟频率 14MHz，我们设置 ADC 的预分频器分频系数为 6，就可以得到 ADC 的输入时钟频率为 72MHz/6，即 12MHz。例程中，我们也是如此设置的。

(2) 转换时间

STM32F103 的 ADC 总转换时间的计算公式如下：

$$T_{\text{CONV}} = \text{采样时间} + 12.5 \text{ 个周期}$$

采样时间可通过 ADC_SMPR1 和 ADC_SMPR2 寄存器中的 SMPx[2:0]位设置，x=0~17。ADC_SMPR1 控制的是通道 0~9，ADC_SMPR2 控制的是通道 10~17。每个输入通道都支持通过编程来选择不同的采样时间，采样时间可选的范围如下：

- SMP = 000: 1.5 个 ADC 时钟周期
- SMP = 001: 7.5 个 ADC 时钟周期
- SMP = 010: 13.5 个 ADC 时钟周期
- SMP = 011: 28.5 个 ADC 时钟周期
- SMP = 100: 41.5 个 ADC 时钟周期
- SMP = 101: 55.5 个 ADC 时钟周期
- SMP = 110: 71.5 个 ADC 时钟周期
- SMP = 111: 239.5 个 ADC 时钟周期

可以看出，采样时间最小是 1.5 个时钟周期，设置为这个值，那么我们可以得到最短的转换时间。下面以我们例程的 ADC 时钟配置为例，来给大家计算一下 ADC 的最短转换时间，计算过程如下：

$$T_{\text{CONV}} = 1.5 \text{ 个 ADC 时钟周期} + 12.5 \text{ 个 ADC 时钟周期} = 14 \text{ 个 ADC 时钟周期}$$

例程中，PCLK2 的时钟是 72MHz，经过 ADC 时钟预分频器的 6 分频后，ADC 时钟频率为 12MHz。代入上式可得到：

$$T_{\text{CONV}} = 14 \text{ 个 ADC 时钟周期} = \left(\frac{1}{12000000} \right) * 14 \text{ s} = 1.17\mu\text{s}$$

⑥数据寄存器

ADC 转换完成后的数据输出寄存器。根据转换组的不同，规则组的完成转换的数据输出到 ADC_DR 寄存器，注入组的完成转换的数据输出到 ADC_JDRx 寄存器。假如是使用双重模式，规则组的数据也是存放在 ADC_DR 寄存器。下面给大家简单介绍一下这两个寄存器。

(1) ADC 规则数据寄存器 (ADC_DR)

ADC 规则组数据寄存器 ADC_DR 是一个 32 位的寄存器，独立模式时只使用到该寄存器低 16 位保存 ADC1/2/3 的规则转换数据。在双 ADC 模式下，高 16 位用于保存 ADC2 转换的数据，低 16 位用于保存 ADC1 转换的数据。

因为 ADC 的精度是 12 位的，ADC_DR 寄存器无论高 16 位还是低 16，存放数据的位宽都是 16 位的，所以允许选择数据对齐方式。由 ADC_CR2 寄存器的 ALIGN 位设置数据对齐方式，可选择：右对齐或者左对齐。

细心的朋友可能发现，规则组最多有 16 个输入通道，而 ADC 规则数据寄存器只有一个，如果一个规则组用到好几个通道，数据怎么读取？如果使用多通道转换，那么这些通道的数据也会存放在 DR 里面，按照规则组的顺序，上一个通道转换的数据，会被下一个通道转换的数据覆盖掉，所以当通道转换完成后要及时把数据取走。比较常用的方法是使用 DMA 模式。当规则组的通道转换结束时，就会产生 DMA 请求，这样就可以及时把转换的数据搬运到用户指定的目的地址存放。注意：只有 ADC1 和 ADC3 可以产生 DAM 请求，而由 ADC2 转换的数据可以通过双 ADC 模式，利用 ADC1 的 DMA 功能传输。

(2) ADC 注入数据寄存器 x (ADC_JDRx) (x=1~4)

ADC 注入数据寄存器有 4 个，注入组最多有 4 个输入通道，刚好每个通道都有自己对应的数据寄存器。ADC_JDRx 寄存器是 32 位的，低 16 位有效，高 16 位保留，数据也同样需要选

择对齐方式。也是由 ADC_CR2 寄存器的 ALIGN 位设置数据对齐方式，可选择：右对齐或者左对齐。

⑦中断

ADC 中断可分为三种：规则组转换结束中断、注入组转换结束中断、设置了模拟看门狗状态位中断。它们都有独立的中断使能位，分别由 ADC_CR 寄存器的 EOCIE、JEOCIE、AWDIE 位设置，对应的标志位分别是 EOC、JEOC、AWD。

模拟看门狗中断

模拟看门狗中断发生条件：首先通过 ADC_LTR 和 ADC_HTR 寄存器设置低阈值和高阈值，然后开启了模拟看门狗中断后，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。例如我们设置高阈值是 3.0V，那么模拟电压超过 3.0V 的时候，就会产生模拟看门狗中断，低阈值的情况类似。

DMA 请求

规则组和注入组的转换结束后，除了可以产生中断外，还可以产生 DMA 请求，我们利用 DMA 及时把转换好的数据传输到指定的内存里，防止数据被覆盖。

注意：只有 ADC1 和 ADC3 可以产生 DAM 请求，DMA 相关的知识请回顾 DMA 实验。

⑧单次转换模式和连续转换模式

单次转换模式和连续转换模式在框图中是没有标号，为了更好地学习后续的内容，这里简单给大家讲讲。

(1) 单次转换模式

通过将 ADC_CR2 寄存器的 CONT 位置 0 选择单次转换模式。该模式下，ADC 只执行一次转换，由 ADC_CR2 寄存器的 ADON 位启动（只适用于规则组），也可以通过外部触发启动（适用于规则组或注入组）。

如果规则组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DR 寄存器中、EOC（转换结束）标志位被置 1、如果设置了 EOCIE 位，则产生中断，然后 ADC 停止。

如果注入组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DRJx 寄存器中、JEOC（转换结束）标志位被置 1、如果设置了 JEOCIE 位，则产生中断，然后 ADC 停止。

(2) 连续转换模式

通过将 ADC_CR2 寄存器的 CONT 位置 1 选择连续转换模式。该模式下，ADC 完成上一个通道的转换后会马上自动地启动下一个通道的转换，由 ADC_CR2 寄存器的 ADON 位启动，也可以通过外部触发启动。

如果规则组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DR 寄存器中、EOC（转换结束）标志位被置 1、如果设置了 EOCIE 位，则产生中断。

如果注入组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DRJx 寄存器中、JEOC（转换结束）标志位被置 1、如果设置了 JEOCIE 位，则产生中断。

⑨扫描模式

扫描模式在框图中是没有标号，为了更好地学习后续的内容，这里简单给大家讲讲。

可以通过 ADC_CR1 寄存器的 SCAN 位配置是否使用扫描模式。如果选择扫描模式，ADC 会扫描所有被 ADC_SQRx 寄存器或 ADC_JSQR 选中的所有通道，并对规则组或者注入组的每个通道执行单次转换，然后停止转换。但如果还设置了 CONT 位，即选择连续转换模式，那么转换不会在选择组的最后一个通道上停止，而是再次从选择组的第一个通道继续转换。

如果设置了 DMA 位，在每次 EOC 后，DMA 控制器把规则组通道的转换数据传输到 SRAM 中。而注入通道转换的数据总是存储在 ADC_JDRx 寄存器中。

到这里，我们基本上介绍了 ADC 的大多数基础的知识，其它知识后面用到会继续补充，如果还有不懂的内容，请参考《STM32F10xxx 参考手册_V10（中文版）.pdf》的第 11 章。

30.2 单通道 ADC 采集实验

本实验我们来学习单通道 ADC 采集实验。本实验使用规则组单通道的单次转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。下面先带大家来了解本实验要配置的寄存器。

30.2.1 ADC 寄存器

这里，我们只介绍本实验用到的寄存器的关键位，其它寄存器后续用到会继续介绍。

● ADC 控制寄存器 1 (ADC_CR1)

ADC 控制寄存器 1 描述如图 30.2.1.1 所示：

| | | | | | | | | | | | | | | | |
|--------------|----|---|---------|--------|-------|---------|------|--------|--------|-------|------------|--------------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | AWDEN | JAWDEN | 保留 | | DUALMOD[3:0] | | | |
| | | | | | | | | rW | rW | | | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DISCNUM[2:0] | | | JDISCEN | DISCEN | JAUTO | AWD SGL | SCAN | JEOCIE | AWDIE | EOCIE | AWDCH[4:0] | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位19:16 | | DUALMOD[3:0]: 双模式选择 (Dual mode selection) 软件使用这些位选择操作模式。 0000: 独立模式 0001: 混合的同步规则+注入同步模式 0010: 混合的同步规则+交替触发模式 0011: 混合同步注入+快速交叉模式 0100: 混合同步注入+慢速交叉模式 0101: 注入同步模式 0110: 规则同步模式 0111: 快速交叉模式 1000: 慢速交叉模式 1001: 交替触发模式 注：在ADC2和ADC3中这些位为保留位 在双模式中，改变通道的配置会产生一个重新开始的条件，这将导致同步丢失。建议在进行任何配置改变前关闭双模式。 | | | | | | | | | | | | | |
| 位8 | | SCAN: 扫描模式 (Scan mode) 该位由软件设置和清除，用于开启或关闭扫描模式。在扫描模式中，转换由ADC_SQRx或ADC_JSQRx寄存器选中的通道。 0: 关闭扫描模式； 1: 使用扫描模式。 注：如果分别设置了EOCIE或JEOCIE位，只在最后一个通道转换完毕后才会产生EOC或JEOC中断。 | | | | | | | | | | | | | |

图 30.2.1.1 ADC_CR1 寄存器

SCAN 位用于选择是否使用扫描模式。本实验我们使用单通道采集，所以没必要选择扫描模式，该位置 0 即可。

DUALMOD[3:0]位用来设置 ADC 的操作模式，我们的例程中 ADC 相关的实验都是使用独立模式，所以设置为 0000 即可。

● ADC 控制寄存器 2 (ADC_CR2)

ADC 控制寄存器 2 描述如图 30.2.1.2 所示：

图 30.2.1.2 ADC CR2 寄存器

457

发转换。

● ADC 采样事件寄存器 1 (ADC SMPR1)

ADC 采样事件寄存器 1 描述如图 30.2.1.3 所示:

| | | | | | | | | | | | | | | | |
|-------------|------------|--|----|------------|----|----|------------|------------|--------------|------------|------------|----|------------|------------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | SMP17[2:0] | | | SMP16[2:0] | | | SMP15[2:1] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SMP 15 0 | SMP14[2:0] | | | SMP13[2:0] | | | SMP12[2:0] | | | SMP11[2:0] | | | SMP10[2:0] | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位23:0 | | SMPx[2:0]: 选择通道x的采样时间 (Channel x Sample time selection) | | | | | | | | | | | | | |
| | | 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 | | | | | | | | | | | | | |
| | | 000: 1.5周期 | | | | | | | 100: 41.5周期 | | | | | | |
| | | 001: 7.5周期 | | | | | | | 101: 55.5周期 | | | | | | |
| | | 010: 13.5周期 | | | | | | | 110: 71.5周期 | | | | | | |
| | | 011: 28.5周期 | | | | | | | 111: 239.5周期 | | | | | | |
| | | 注: ADC1的模拟输入通道16和通道17在芯片内部分别连到了温度传感器和V _{REFINT} 。 | | | | | | | | | | | | | |
| | | ADC2的模拟输入通道16和通道17在芯片内部连到了V _{SS} 。 | | | | | | | | | | | | | |
| | | ADC3模拟输入通道14、15、16、17与V _{SS} 相连 | | | | | | | | | | | | | |

图 30.2.1.3 ADC SMPR1 寄存器

● ADC 采样事件寄存器 2 (ADC SMPR2)

ADC 采样事件寄存器 2 描述如图 30.2.1.4 所示:

| | | | | | | | | | | | | | | | |
|-------------|------------|--|----|------------|----|----|------------|------------|----|------------|----|----|------------|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | SMP17[2:0] | | SMP16[2:0] | | | SMP15[2:1] | | |
| | | | | | | | | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SMP 15 0 | SMP14[2:0] | | | SMP13[2:0] | | | SMP12[2:0] | | | SMP11[2:0] | | | SMP10[2:0] | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位23:0 | | SMPx[2:0]: 选择通道x的采样时间 (Channel x Sample time selection) 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000: 1.5周期 | | | | | | | | | | | | | |

图 30.2.1.4 ADC SMPR2 寄存器

ADC 采样时间设置需要由两个寄存器设置，ADC_SMPR1 和 ADC_SMPR2，分别设置通道 10~17 和通道 0~9 的采样时间，每个通道用 3 个位设置。可以看出 ADC 的每个通道的采样时间是支持单独设置的。

一般每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率，看大家怎么衡量选择了。本实验中，我们设置采样时间是 239.5 个周期。结合前面介绍过的转换时间公式：

$$T_{CONV} = \text{采样时间} + 12.5 \text{ 个周期}$$

以及例程中，PCLK2 的时钟是 72MHz，经过 ADC 时钟预分频器的 6 分频后，ADC 时钟频率为 12MHz。代入上式可得到：

$$T_{\text{CONV}} = 239.5 + 12.5 \text{ 个 ADC 时钟周期} = \left(\frac{1}{12000000} \right) * 252 \text{ s} = 21 \mu\text{s}$$

由上式可得, ADC 的转换时间是 21us。

● ADC 规则序列寄存器 1

ADC 规则序列寄存器共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC 规则序列寄存器 1 (ADC_SQR1)，描述如图 30.2.1.5 所示：

| | | | | | | | | | | | | | | | |
|------------|-----------|---|----|----|----|-----------|----|--------|----|----|-----------|-----------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | L[3:0] | | | | SQ16[4:1] | | | |
| | | | | | | | | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SQ16 _0 | SQ15[4:0] | | | | | SQ14[4:0] | | | | | SQ13[4:0] | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位23:20 | | L[3:0]: 规则通道序列长度 (Regular channel sequence length) 这些位由软件定义在规则通道转换序列中的通道数目。 0000: 1个转换 0001: 2个转换 1111: 16个转换 | | | | | | | | | | | | | |
| 位19:15 | | SQ16[4:0]: 规则序列中的第16个转换 (16th conversion in regular sequence) 这些位由软件定义转换序列中的第16个转换通道的编号(0~17)。 | | | | | | | | | | | | | |
| 位14:10 | | SQ15[4:0]: 规则序列中的第15个转换 (15th conversion in regular sequence) | | | | | | | | | | | | | |
| 位9:5 | | SQ14[4:0]: 规则序列中的第14个转换 (14th conversion in regular sequence) | | | | | | | | | | | | | |
| 位4:0 | | SQ13[4:0]: 规则序列中的第13个转换 (13th conversion in regular sequence) | | | | | | | | | | | | | |

图 30.2.1.5 ADC_SQR1 寄存器

L[3:0]用于设置规则组序列的长度，取值范围：0~15，表示规则组的长度是 1~16。本实验只用了 1 个输入通道，所以 L[3:0]位设置为 0000 即可。

SQ13[4:0]~SQ16[4:0]位设置规则组序列的第 13~16 个转换编号，第 1~12 个转换编号的设置请查看 ADC_SQR2 和 ADC_SQR3 寄存器。设置过程非常简单，忘记了请参考前面给大家整理出来的规则序列寄存器控制关系汇总表。

本实验我们使用单通道，ADC1 通道 14，所以规则组序列里只有一个输入通道，我们将 ADC_SQR3 寄存器的 SQ1[4:0]位的值设置为 14 即可。

● ADC 规则数据寄存器 (ADC_DR)

ADC 规则数据寄存器描述如图 30.2.1.6 所示：

| | | | | | | | | | | | | | | | |
|----------------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| ADC2DATA[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DATA[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |
| 位31:16 | | ADC2DATA[15:0]: ADC2转换的数据 (ADC2 data) - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见11.9：双ADC模式 - 在ADC2和ADC3中：不使用这些位。 | | | | | | | | | | | | | |
| 位15:0 | | DATA[15:0]: 规则转换的数据 (Regular data) 这些位为只读，包含了规则通道的转换结果。数据是左对齐或右对齐，如图29和图30所示。 | | | | | | | | | | | | | |

图 30.2.1.6 ADC_DR 寄存器

在规则序列中 AD 转换结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

● ADC 状态寄存器 (ADC_SR)

ADC 状态寄存器描述如图 30.2.1.7 所示：

| | | | | | | | | | | | | | | | |
|----|--|----|----|----|----|----|----|----|----|----|-------|-------|-------|-------|-------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | | | | STRT | JSTRT | JEOC | EOC | AWD |
| | | | | | | | | | | | rc w0 | rc w0 | rc w0 | rc w0 | rc w0 |
| 位4 | STRT : 规则通道开始位 (Regular channel Start flag) 该位由硬件在规则通道转换开始时设置, 由软件清除。 0: 规则通道转换未开始; 1: 规则通道转换已开始。 | | | | | | | | | | | | | | |
| 位3 | JSTRT : 注入通道开始位 (Injected channel Start flag) 该位由硬件在注入通道组转换开始时设置, 由软件清除。 0: 注入通道组转换未开始; 1: 注入通道组转换已开始。 | | | | | | | | | | | | | | |
| 位2 | JEOC : 注入通道转换结束位 (Injected channel end of conversion) 该位由硬件在所有注入通道组转换结束时设置, 由软件清除 0: 转换未完成; 1: 转换完成。 | | | | | | | | | | | | | | |
| 位1 | EOC : 转换结束位 (End of conversion) 该位由硬件在(规则或注入)通道组转换结束时设置, 由软件清除或由读取ADC_DR时清除 0: 转换未完成; 1: 转换完成。 | | | | | | | | | | | | | | |
| 位0 | AWD : 模拟看门狗标志位 (Analog watchdog flag) 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置, 由软件清除 0: 没有发生模拟看门狗事件; 1: 发生模拟看门狗事件。 | | | | | | | | | | | | | | |

图 30.2.1.7 ADC_SR 寄存器

该寄存器保存了 ADC 转换时的各种状态。本实验我们通过 EOC 位的状态来判断 ADC 转换是否完成, 如果查询到 EOC 位被硬件置 1, 就可以从 ADC_DR 寄存器中读取转换结果, 否则需要等待转换完成。

至此, 本实验要用到的 ADC 关键寄存器全部介绍完了, 对于未介绍的部分, 请大家参考《STM32F10xxx 参考手册_V10 (中文版).pdf》第 11 章相关内容。

30.2.2 硬件设计

1. 例程功能

采集 ADC1 通道 1 (PA1) 上面的电压, 并在 LCD 模块上面显示 ADC 规则数据寄存器 12 位的转换值以及将该值换算成电压后的电压值。使用杜邦线将 ADC 和 RV1 排针连接, 使得 PA1 连接到电位器上, 然后将 ADC 采集到的数据和转换后的电压值在 TFTLCD 屏中显示。用户可以通过调节电位器的旋钮改变电压值。LED0 闪烁, 提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) ADC1 :
通道 1 - PA1

3. 原理图

ADC 属于 STM32F103 内部资源，实际上我们只需要软件设置就可以正常工作，另外还需要将待测量的电压源连接到 ADC 通道上，以便 ADC 测量。本实验，我们通过 ADC1 的通道 1（PA1）来采集外部电压值，开发板有一个电位器，可调节的电压范围是：0~3.3V。我们可以通过杜邦线将 PA1 与电位器连接，如下图所示：

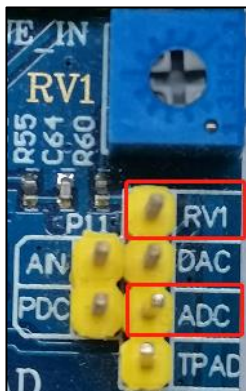


图 30.2.2.1 PA1（对应 ADC 排针）与电位器示意图

使用杜邦线将 ADC 和 RV1 排针连接好后，并下载程序后，就可以用螺丝刀调节电位器变换多种电压值进行测量。

有的朋友可能还想测量其它地方的电压值，我们只需要 1 根杜邦线，一端接到 ADC 排针上，另外一端就接你要测试的电压点。**一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

30.2.3 程序设计

30.2.3.1 ADC 的 HAL 库驱动

ADC 在 HAL 库中的驱动代码在 stm32f1xx_hal_adc.c 和 stm32f1xx_hal_adc_ex.c 文件（及其头文件）中。

1. HAL_ADC_Init 函数

ADC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef *hadc);
```

- **函数描述：**
用于初始化 ADC。
- **函数形参：**

形参 1 是 ADC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
    ADC_TypeDef                *Instance;           /* ADC 寄存器基地址 */
    ADC_InitTypeDef            Init;                /* ADC 参数初始化结构体变量 */
    DMA_HandleTypeDef          *DMA_Handle;        /* DMA 配置结构体 */
    HAL_LockTypeDef            Lock;                /* ADC 锁定对象 */
    __IO uint32_t               State;              /* ADC 工作状态 */
    __IO uint32_t               ErrorCode;          /* ADC 错误代码 */
}ADC_HandleTypeDef;
```

该结构体定义和其他外设比较类似，我们着重看第二个成员变量 Init 含义，它是结构体 ADC_InitTypeDef 类型，结构体 ADC_InitTypeDef 定义为：

```
typedef struct {
    uint32_t DataAlign;           /* 设置数据的对齐方式 */
    uint32_t ScanConvMode;        /* 扫描模式 */
    FunctionalState ContinuousConvMode; /* 开启连续转换模式否则就是单次转换模式 */
    uint32_t NbrOfConversion;     /* 设置转换通道数目 */
};
```

```
FunctionalState DiscontinuousConvMode; /* 是否使用规则通道组间断模式 */
uint32_t NbrOfDiscConversion; /* 配置间断模式的规则通道个数 */
uint32_t ExternalTrigConv; /* ADC 外部触发源选择 */
} ADC_InitTypeDef;
```

- 1) DataAlign: 用于设置数据的对齐方式, 这里可以选择右对齐或者是左对齐, 该参数可选为: ADC_DATAALIGN_RIGHT 和 ADC_DATAALIGN_LEFT。
- 2) ScanConvMode: 配置是否使用扫描。如果是单通道转换使用 ADC_SCAN_DISABLE, 如果是多通道转换使用 ADC_SCAN_ENABLE。
- 3) ContinuousConvMode: 可选参数为 ENABLE 和 DISABLE, 配置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。
- 4) NbrOfConversion: 指定规则组转换通道数目, 范围是: 1~16。
- 5) DiscontinuousConvMode: 配置是否使用规则通道组间断模式, 比如要转换的通道有 1、2、5、7、8、9, 那么第一次触发会进行通道 1 和 2, 下次触发就是转换通道 5 和 7, 这样不连续的转换, 依次类推。此参数只有将 ScanConvMode 使能, 还有 ContinuousConvMode 失能的情况下才有效, 不可同时使能。
- 6) NbrOfDiscConversion: 配置间断模式的通道个数, 禁止规则通道组间断模式后, 此参数忽略。
- 7) ExternalTrigConv: 外部触发方式的选择, 如果使用软件触发, 那么外部触发会关闭。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

2. HAL_ADCEx_Calibration_Start 函数

ADC 的自校准函数, 其声明如下:

```
HAL_StatusTypeDef HAL_ADCEx_Calibration_Start(ADC_HandleTypeDef *hadc);
```

● 函数描述:

首先调用 HAL_ADC_Init 函数配置了相关的功能后, 再调用此函数进行 ADC 自校准功能。

● 函数形参:

ADC_HandleTypeDef 结构体类型指针变量。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

3. HAL_ADC_ConfigChannel 函数

ADC 通道配置函数, 其声明如下:

```
HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef *hadc,
                                         ADC_ChannelConfTypeDef *sConfig);
```

● 函数描述:

调用了 HAL_ADC_Init 函数配置了相关的功能后, 就可以调用此函数配置 ADC 具体通道。

● 函数形参:

形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。

形参 2 是 ADC_ChannelConfTypeDef 结构体类型指针变量, 用于配置 ADC 采样时间, 使用的通道号, 单端或者差分方式的配置等。该结构体定义如下:

```
typedef struct {
    uint32_t Channel; /* ADC 转换通道 */
    uint32_t Rank; /* ADC 转换顺序 */
    uint32_t SamplingTime; /* ADC 采样周期 */
} ADC_ChannelConfTypeDef;
```

- 1) Channel: ADC 转换通道, 范围: 0~19。
- 2) Rank: 在常规转换中的常规组的转换顺序, 可以选择 1~16。
- 3) SamplingTime: ADC 的采样周期, 最大 810.5 个 ADC 时钟周期, 要求尽量大以减少误差。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值。

4. HAL_ADC_Start 函数

ADC 转换启动函数, 其声明如下:

```
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef *hadc);
```


- **函数描述:**
当配置好 ADC 的基础的功能后，就调用此函数启动 ADC。
- **函数形参:**
ADC_HandleTypeDef 结构体类型指针变量。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

5. HAL_ADC_PollForConversion 函数

等待 ADC 规则组转换完成函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef *hadc,
                                             uint32_t Timeout);
```

- **函数描述:**
一般先调用 HAL_ADC_Start 函数启动转换，再调用该函数等待转换完成，然后再调用 HAL_ADC_GetValue 函数来获取当前的转换值。
- **函数形参:**
形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
形参 2 是等待转换的等待时间，单位是毫秒（ms）。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

6. HAL_ADC_GetValue 函数

获取常规组 ADC 转换值函数，其声明如下：

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef *hadc);
```

- **函数描述:**
一般先调用 HAL_ADC_Start 函数启动转换，再调用 HAL_ADC_PollForConversion 函数等待转换完成，然后再调用 HAL_ADC_GetValue 函数来获取当前的转换值。
- **函数形参:**
形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
- **函数返回值:**
当前的转换值，uint32_t 类型数据。

单通道 ADC 采集配置步骤

1) 开启 ADCx 和 ADC 通道对应的 IO 时钟，并配置该 IO 为模拟功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 ADC1 通道 1，对应 IO 是 PA1，它们的时钟开启方法如下：

```
HAL_RCC_ADC1_CLK_ENABLE(); /* 使能 ADC1 时钟 */
HAL_RCC_GPIOA_CLK_ENABLE(); /* 开启 GPIOA 时钟 */
```

2) 初始化 ADCx，配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。

3) 配置 ADC 通道并启动 AD 转换器

在 HAL 库中，通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道，并设置规则序列、采样时间等。

配置好 ADC 通道之后，通过 HAL_ADC_Start 函数启动 AD 转换器。

4) 读取 ADC 值

这里选择查询方式读取，在读取 ADC 值之前需要调用 HAL_ADC_PollForConversion 等待上一次转换结束。然后就可以通过 HAL_ADC_GetValue 来读取 ADC 值。

30.2.3.2 程序流程图

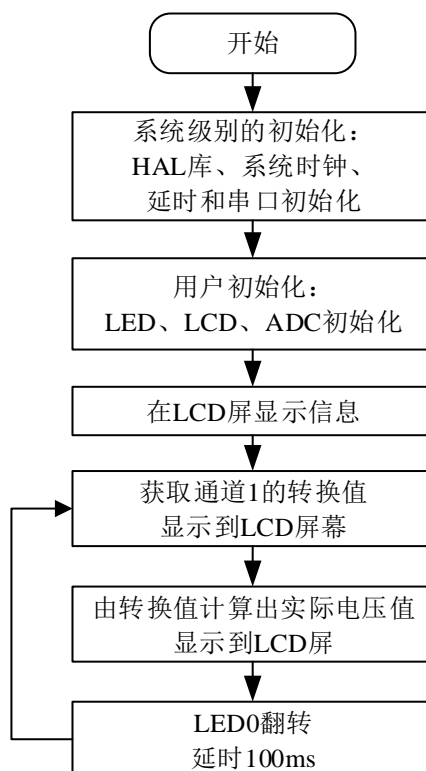


图 30.2.3.2.1 单通道 ADC 采集实验程序流程图

30.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。ADC 驱动源码包括两个文件：adc.c 和 adc.h。本章有四个实验，每一个实验的代码都是在上一个实验后面追加。

adc.h 文件针对 ADC 及通道引脚定义了一些宏定义，具体如下：

```

/* ADC 及引脚 定义 */
#define ADC_ADCX_CHY_GPIO_PORT      GPIOA
#define ADC_ADCX_CHY_GPIO_PIN      GPIO_PIN_1
#define ADC_ADCX_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE();\
                                           }while(0) /* PA 口时钟使能 */

#define ADC_ADCX      ADC1
#define ADC_ADCX_CHY  ADC_CHANNEL_1 /* 通道 y, 0 <= y <= 16 */
/* ADC1 时钟使能 */
#define ADC_ADCX_CHY_CLK_ENABLE() do{ __HAL_RCC_ADC1_CLK_ENABLE();}while(0)
    
```

ADC 的通道与引脚的对应关系在《STM32F103ZET6.pdf》数据手册可以查到，我们这里使用 ADC1 的通道 1，在数据手册中的表格为：

| | | | |
|-----|-----|-----|--|
| PA1 | I/O | PA1 | USART2_RTS ⁽⁷⁾ ADC123_IN1/ TIM5_CH2/TIM2_CH2 ⁽⁷⁾ |
|-----|-----|-----|--|

表 30.2.3.3.1 ADC 通道 1 对应引脚查看表

下面直接开始介绍 adc.c 的程序，首先是 ADC 初始化函数。

```

/**
 * @brief      ADC 初始化函数
 * @note       本函数支持 ADC1/ADC2 任意通道，但是不支持 ADC3
 *             我们使用 12 位精度，ADC 采样时钟=12M，转换时间为：采样周期 + 12.5 个 ADC 周期
 *             设置最大采样周期：239.5，则转换时间 = 252 个 ADC 周期 = 21us
 * @param      无
    
```

```

* @retval    无
*/
void adc_init(void)
{
    g_adc_handle.Instance = ADC_ADCX;                /* 选择哪个 ADC */
    g_adc_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 数据对齐方式：右对齐 */
    g_adc_handle.Init.ScanConvMode = ADC_SCAN_DISABLE; /* 非扫描模式 */
    g_adc_handle.Init.ContinuousConvMode = DISABLE;    /* 关闭连续转换模式 */
    g_adc_handle.Init.NbrOfConversion = 1;            /* 范围是 1~16，这里用到 1 个规则通道 */
    g_adc_handle.Init.DiscontinuousConvMode = DISABLE; /* 禁止规则通道组间断模式 */
    /* 配置间断模式的规则通道个数，禁止规则通道组间断模式后，此参数忽略 */
    g_adc_handle.Init.NbrOfDiscConversion = 0;
    g_adc_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START; /* 软件触发 */
    HAL_ADC_Init(&g_adc_handle);                            /* 初始化 */

    HAL_ADCEx_Calibration_Start(&g_adc_handle);           /* 校准 ADC */
}

```

该函数主要调用了两个 HAL 库函数，HAL_ADC_Init 函数配置了选择哪个 ADC、数据对齐方式、是否使用扫描模式等参数，HAL_ADCEx_Calibration_Start 函数用于校准 ADC。另外 HAL_ADC_Init 函数会调用它的 MSP 回调函数 HAL_ADC_MspInit，该函数用来存放使能 ADC 和通道对应 IO 的时钟和初始化 IO 口等代码，其定义如下：

```

/**
 * @brief      ADC 底层驱动，引脚配置，时钟使能
               此函数会被 HAL_ADC_Init() 调用
 * @param      hadc:ADC 句柄
 * @retval     无
 */
void HAL_ADC_MspInit(ADC_HandleTypeDef *hadc)
{
    if(hadc->Instance == ADC_ADCX)
    {
        GPIO_InitTypeDef gpio_init_struct;
        RCC_PeriphCLKInitTypeDef adc_clk_init = {0};

        ADC_ADCX_CHY_CLK_ENABLE();                /* 使能 ADCx 时钟 */
        ADC_ADCX_CHY_GPIO_CLK_ENABLE();            /* 开启 GPIO 时钟 */

        /* 设置 ADC 时钟 */
        adc_clk_init.PeriphClockSelection = RCC_PERIPHCLK_ADC; /* ADC 外设时钟 */
        /* 分频系数为 6，所以 ADC 的时钟为 72M/6=12MHz */
        adc_clk_init.AdcClockSelection = RCC_ADCCLK2_DIV6;
        HAL_RCCEx_PeriphCLKConfig(&adc_clk_init);    /* 设置 ADC 时钟 */

        /* 设置 AD 采集通道对应 IO 引脚工作模式 */
        gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN; /* ADC 通道 IO 引脚 */
        gpio_init_struct.Mode = GPIO_MODE_ANALOG;      /* 模拟 */
        HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);
    }
}

```

可以看到在 HAL_ADC_MspInit 函数中，我们除了使能 ADC 和通道对应 IO 时钟、初始化 IO 外，还配置了 ADC 的时钟预分频系数。ADC 的时钟源是 PCLK2 (72MHz)，经过 6 分频后，得到 ADC 的输入时钟是 12MHz。

接下来要介绍的函数是 adc_channel_set，其定义如下：

```

/**
 * @brief      设置 ADC 通道采样时间
 * @param      adcx : adc 句柄指针,ADC_HandleTypeDef
 * @param      ch   : 通道号, ADC_CHANNEL_0~ADC_CHANNEL_17
 * @param      stime: 采样时间 0~7, 对应关系为:
 * @arg        ADC_SAMPLETIME_1CYCLE_5, 1.5 个 ADC 时钟周

```

```

        ADC_SAMPLETIME_7CYCLES_5, 7.5 个 ADC 时钟周期
*   @arg      ADC_SAMPLETIME_13CYCLES_5, 13.5 个 ADC 时钟周期      ADC_SAM-
        PLETIME_28CYCLES_5, 28.5 个 ADC 时钟周期
*   @arg      ADC_SAMPLETIME_41CYCLES_5, 41.5 个 ADC 时钟周期      ADC_SAM-
        PLETIME_55CYCLES_5, 55.5 个 ADC 时钟周期
*   @arg      ADC_SAMPLETIME_71CYCLES_5, 71.5 个 ADC 时钟周期      ADC_SAM-
        PLETIME_239CYCLES_5, 239.5 个 ADC 时钟周期
*   @param    rank: 多通道采集时需要设置的采集编号,
        假设你定义 channel1 的 rank=1, channel2 的 rank=2,
        那么对应你在 DMA 缓存空间的变量数组 AdcDMA[0] 就 i 是 channel1 的转换结果,
        AdcDMA[1] 就是通道 2 的转换结果。
        单通道 DMA 设置为 ADC_REGULAR_RANK_1
*   @arg      编号 1~16: ADC_REGULAR_RANK_1~ADC_REGULAR_RANK_16
*   @retval    无
*/
void adc_channel_set(ADC_HandleTypeDef *adc_handle, uint32_t ch, uint32_t rank,
                    uint32_t stime)
{
    ADC_ChannelConfTypeDef adc_ch_conf;

    adc_ch_conf.Channel = ch;                /* 通道 */
    adc_ch_conf.Rank = rank;                 /* 序列 */
    adc_ch_conf.SamplingTime = stime;        /* 采样时间 */
    HAL_ADC_ConfigChannel(adc_handle, &adc_ch_conf); /* 通道配置 */
}

```

该函数主要是通过 HAL_ADC_ConfigChannel 函数选择要配置的 ADC 规则组通道, 并设置通道的序列号和采样时间。

下面要介绍的是获得 ADC 转换后的结果函数, 其定义如下:

```

/**
 * @brief      获得 ADC 转换后的结果
 * @param      ch: 通道值 0~17, 取值范围为: ADC_CHANNEL_0~ADC_CHANNEL_17
 * @retval     无
 */
uint32_t adc_get_result(uint32_t ch)
{
    adc_channel_set(&g_adc_handle, ch, ADC_REGULAR_RANK_1, ADC_SAM-
                    PLETIME_239CYCLES_5); /* 设置通道/序列和采样时间 */
    HAL_ADC_Start(&g_adc_handle);        /* 开启 ADC */
    HAL_ADC_PollForConversion(&g_adc_handle, 10); /* 等待转换结束 */
    /* 返回最近一次 ADC1 规则组的转换结果 */
    return (uint16_t)HAL_ADC_GetValue(&g_adc_handle);
}

```

该函数先是调用我们自己定义的 adc_channel_set 函数选择 ADC 通道、设置转换序列号和采样时间等, 接着调用 HAL_ADC_Start 启动转换, 然后调用 HAL_ADC_PollForConversion 函数等待转换完成, 最后调用 HAL_ADC_GetValue 函数获取转换结果。

接下来要介绍的函数是获取 ADC 某通道多次转换结果平均值函数, 函数定义如下:

```

/**
 * @brief      获取通道 ch 的转换值, 取 times 次, 然后平均
 * @param      ch      : 通道号, 0~17
 * @param      times   : 获取次数
 * @retval     通道 ch 的 times 次转换结果平均值
 */
uint32_t adc_get_result_average(uint32_t ch, uint8_t times)
{
    uint32_t temp_val = 0;
    uint8_t t;

    for (t = 0; t < times; t++) /* 获取 times 次数据 */
    {

```

```

        temp_val += adc_get_result(ch);
        delay_ms(5);
    }

    return temp_val / times;          /* 返回平均值 */
}

```

该函数用于获取 ADC 多次转换结果的平均值，从而提高准确度。

最后在 main 函数里面编写如下代码：

```

int main(void)
{
    uint16_t adcx;
    float temp;
    sys_stm32_clock_init(RCC_PLL_MUL9);    /* 设置时钟，72Mhz */
    delay_init(72);                        /* 延时初始化 */
    usart_init(115200);                    /* 串口初始化为 115200 */
    led_init();                            /* 初始化 LED */
    lcd_init();                            /* 初始化 LCD */
    adc_init();                            /* 初始化 ADC */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH1_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH1_VOL:0.000V", BLUE);

    while (1)
    {
        /* 获取通道 5 的转换值，10 次取平均 */
        adcx = adc_get_result_average(ADC_ADCX_CHY, 10);
        lcd_show_xnum(134, 110, adcx, 5, 16, 0, BLUE); /* 显示 ADCC 采样后的原始值 */

        temp = (float)adcx * (3.3/4096); /* 获取计算后的带小数的实际电压值，比如 3.1111 */
        adcx = temp;                    /* 赋值整数部分给 adcx 变量，因为 adcx 为 u16 整形 */
        /* 显示电压值的整数部分，3.1111 的话，这里就是显示 3 */
        lcd_show_xnum(134, 130, adcx, 1, 16, 0, BLUE);

        temp -= adcx; /* 把已经显示的整数部分去掉，留下小数部分，比如 3.1111-3=0.1111 */
        temp *= 1000; /* 小数部分乘以 1000，例如：0.1111 就转换为 111.1，相当于保留三位小数 */
        /* 显示小数部分（前面转换为了整形显示），这里显示的就是 111. */
        lcd_show_xnum(150, 130, temp, 3, 16, 0x80, BLUE);

        LED0_TOGGLE();
        delay_ms(100);
    }
}

```

此部分代码，我们在 TFTLCD 模块上显示一些提示信息后，将每隔 100ms 读取一次 ADC 通道 1 的转换值，并显示读到的 ADC 值（数字量），以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。ADC 值的显示简单介绍一下：首先在液晶固定位置显示了小数点，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就能在液晶上面显示转换结果的整数和小数部分。

30.2.4 下载验证

下载代码后，可以看到 LCD 显示如图 30.2.4.1 所示：

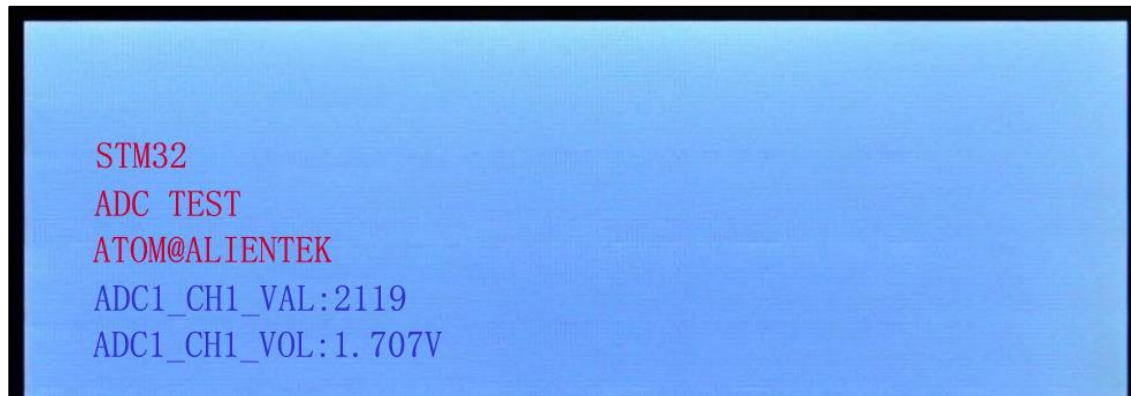


图 30.2.4.1 单通道 ADC 采集实验测试图

上图中，我们使用杜邦线将 ADC 和 RV1 排针连接，使得 PA1 连接到电位器上，测试的是电位器的电压，并可以通过螺丝刀调节电位器改变电压值，范围：0~3.3V。LED0 闪烁，提示程序运行。

大家也可以用杜邦线将 ADC 排针接到其它待测量的电压点，看看测量到的电压值是否准确？**但是要注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

30.3 单通道 ADC 采集（DMA 读取）实验

本实验我们来学习单通道 ADC 采集（DMA 读取）实验。本实验使用规则组单通道的连续转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。由于使用连续转换模式，所以使用 DMA 读取转换结果的方式。下面先带大家来了解本实验要配置的寄存器。

30.3.1 ADC & DMA 寄存器

本实验我们很多的设置和单通道 ADC 采集实验是一样的，所以下面介绍寄存器的时候我们不会继续全部都介绍，而是针对性选择与单通道 ADC 采集实验不同设置的 ADC_CR2 寄存器进行介绍，其他的配置基本一样的。另外因为我们用到 DMA 读取数据，所以还会介绍如何配置相关 DMA 的寄存器。

● ADC 配置寄存器（ADC_CR2）

ADCx 配置寄存器描述如图 30.3.1.1 所示：

| | | | | | | | | | | | | | | | | |
|--------------|--------------|----|----|--|----|----|-----|-------------|-------------|--------------|-------------|-------------|-----|------|------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | |
| 保留 | | | | | | | | TS VREFE | SW START | JSW START | EXT TRIG | EXTSEL[2:0] | | | 保留 | |
| | | | | | | | | rW | rW | rW | rW | rW | rW | rW | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| JEXT TRIG | JEXTSEL[2:0] | | | ALIGN | 保留 | | DMA | 保留 | | | | RST CAL | CAL | CONT | ADON | |
| rW | | rW | | rW | rW | rW | | | | | | rW | | rW | rW | rW |
| | | 位8 | | DMA： 直接存储器访问模式 (Direct memory access mode) 该位由软件设置和清除。详见DMA控制器章节。 0：不使用DMA模式； 1：使用DMA模式。 注：只有ADC1和ADC3能产生DMA请求。 | | | | | | | | | | | | |
| | | 位1 | | CONT： 连续转换 (Continuous conversion) 该位由软件设置和清除。如果设置了此位，则转换将连续进行直到该位被清除。 0：单次转换模式； 1：连续转换模式。 | | | | | | | | | | | | |

图 30.3.1.1 ADC_CR2 寄存器

ADC_CR2 寄存器中我们主要跟前面设置不同的有两个位，分别如下：

DMA 位用于配置使用 DMA 模式，本实验该位置 1。在单通道 ADC 采集实验中，默认设置为 0，即不使用 DMA 模式，规则组转换的结果存储在 ADC_DR 寄存器，然后通过手动读取 ADC_DR 寄存器的方式得到转换结果。本实验我们使用 ADC 的连续转换模式，并通过 DMA 读取转换结果，这样 DMA 就会自动在 ADC_DR 寄存器中读取转换结果。

CONT 位用于设置单次转换模式还是连续转换模式，本实验我们使用连续转换模式，所以 CONT 位置 1 即可。

这里介绍 ADC_CR2 寄存器的这两个位，其它请参考上一个实验的配置。下面介绍 DMA 一些比较重要的寄存器配置。

● DMA 通道 x 外设地址寄存器（DMA_CPARx）（x = 1…7）

DMA 通道 x 外设地址寄存器描述如图 30.3.1.2 所示：

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PA[31:0] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| rW | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位31:0 | | PA[31:0]： 外设地址 (Peripheral address) 外设数据寄存器的基地址，作为数据传输的源或目标。 当PSIZE=01'(16位)，不使用PA[0]位。操作自动地与半字地址对齐。 当PSIZE=10'(32位)，不使用PA[1:0]位。操作自动地与字地址对齐。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 30.3.1.2 DMA_CPARx 寄存器

该寄存器存放的是 DMA 通道 x 外设地址。本实验，我们需要通过 DMA 读取 ADC 转换后存放在 ADC 规则数据寄存器 (ADC_DR) 的结果数据。所以我们需要给 DMA_CPARx 寄存器写入 ADC_DR 寄存器的地址。这样配置后，DMA 就会从 ADC_DR 寄存器的地址读取 ADC 的转换后的数据到某个内存空间。这个内存空间地址需要我们通过 DMA_CMARx 寄存器来设置，比如定义一个变量，把这个变量的地址值写入该寄存器。

注意：DMA_CPARx 寄存器受到写保护，只有 DMA_CCRx 寄存器中的 EN 为“0”时才可以写入，即先要禁止通道开启才可以写入。

● DMA 通道 x 存储器地址寄存器 (DMA_CMARx) (x = 1...7)

DMA 通道 x 存储器地址寄存器描述如图 30.3.1.3 所示：

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MA[31:0] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| rW | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位31:0 | | | | | | | | | | | | | | | | MA[31:0]: 存储器地址 存储器地址作为数据传输的源或目标。 | | | | | | | | | | | | | | | |

图 30.3.1.3 DMA_CMARx 寄存器

该寄存器存放的是 DMA 通道 x 存储器地址。同样的，该寄存器也是受写保护，只有当 DMA_CCRx 的 EN 位为 0 时才可以写入。

● DMA 通道 x 传输数量寄存器 (DMA_CNDTRx) (x = 1...7)

DMA 通道 x 传输数量寄存器描述如图 30.3.1.4 所示：

| | | | | | | | | | | | | | | | |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| NDT[15:0] | | | | | | | | | | | | | | | |
| rW rW rW rW rW rW rW rW rW rW rW rW rW rW rW rW | | | | | | | | | | | | | | | |
| 位15:0 NDT[15:0]: 数据传输数量 (Number of data to transfer) 数据传输数量为0至65535。这个寄存器只能在通道不工作(DMA_CCRx的EN=0)时写入。通道开启后该寄存器变为只读，指示剩余的待传输字节数目。寄存器内容在每次DMA传输后递减。 数据传输结束后，寄存器的内容或者变为0；或者当该通道配置为自动重加载模式时，寄存器的内容将被自动重新加载为之前配置时的数值。 当寄存器的内容为0时，无论通道是否开启，都不会发生任何数据传输。 | | | | | | | | | | | | | | | |

图 30.3.1.4 DMA_CNDTRx

该寄存器控制着 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成。所以可以通过这个寄存器的值来获取当前 DMA 传输的进度。

其它的 DMA 寄存器我们就不一一介绍了，请大家看着寄存器源码对照手册理解，都不难。

30.3.2 硬件设计

1. 例程功能

使用 ADC 采集 (DMA 读取) 通道 1 (PA1) 上面的电压，在 LCD 模块上面显示 ADC 转换值以及换算成电压后的电压值。使用短路帽将 ADC 和 RV1 排针连接，使得 PA1 连接到电位器上，然后将 ADC 采集到的数据和转换后的电压值在 TFTLCD 屏中显示。用户可以通过调节电位器的旋钮改变电压值。LED0 闪烁，提示程序运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) ADC : 通道 1 - PA1
- 5) DMA (DMA1 通道 1)

3. 原理图

ADC 属于 STM32F103 内部资源, 实际上我们只需要软件设置就可以正常工作, 另外还需要将待测量的电压源连接到 ADC 通道上, 以便 ADC 测量。本实验, 我们通过 ADC1 的通道 1 (PA1) 来采集外部电压值, 开发板有一个电位器, 可调节的电压范围是: 0~3.3V。我们可以通过杜邦线将 PA1 与电位器连接, 如下图所示:



图 30.3.2.1 PA1（对应 ADC 排针）与电位器示意图

使用杜邦线将 ADC 和 RV1 排针连接好后, 并下载程序后, 就可以用螺丝刀调节电位器变换多种电压值进行测量。

有的朋友可能还想测量其它地方的电压值, 我们只需要 1 根杜邦线, 一端接到 ADC 排针上, 另外一端就接你要测试的电压点。**一定要保证测试点的电压在 0~3.3V 的电压范围, 否则可能烧坏我们的 ADC, 甚至是整个主控芯片。**

30.3.3 程序设计

30.3.3.1 ADC & DMA 的 HAL 库驱动

单通道 ADC 采集实验已经介绍了一部分 ADC 的 HAL 库 API 函数, 这里要介绍的是 HAL_DMA_Start_IT 和 HAL_ADC_Start_DMA 函数。

1. HAL_DMA_Start_IT 函数

启动 DMA 传输并开启相关中断函数, 其声明如下:

```
HAL_StatusTypeDef HAL_DMA_Start_IT(DMA_HandleTypeDef *hdma,
                                     uint32_t SrcAddress, uint32_t DstAddress, uint32_t DataLength);
```

- **函数描述:**
用于启动 DMA 传输, 并开启相关中断, DMA1 和 DMA2 都是用的这个函数。
- **函数形参:**
形参 1 是 DMA_HandleTypeDef 结构体类型指针变量。
形参 2 是 DMA 传输的源地址。
形参 3 是 DMA 传输的目的地址。
形参 4 是要传输的数据项数目。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

2. HAL_ADC_Start_DMA 函数

启动 ADC (DMA 传输) 方式函数, 其声明如下:

```
HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc,
                                     uint32_t *pData, uint32_t Length);
```

- **函数描述:**
用于启动 ADC（DMA 传输）方式的函数。
- **函数形参:**
形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
形参 2 是 ADC 采样数据传输的目的地址。
形参 3 是要传输的数据项数目。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

单通道 ADC 采集（DMA 读取）配置步骤

1) 开启 ADCx 和 ADC 通道对应的 IO 时钟，并配置该 IO 为模拟功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 ADC1 通道 1，对应 IO 是 PA1，它们的时钟开启方法如下：

```
__HAL_RCC_ADC1_CLK_ENABLE();           /* 使能 ADC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();          /* 开启 GPIOA 时钟 */
```

2) 初始化 ADCx，配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。我们也可以不存放在这个函数里，本实验就没用到这个 MSP 回调函数。

3) 配置 ADC 通道并启动 AD 转换器

在 HAL 库中，通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道，并设置规则序列、采样时间等。

4) 初始化 DMA

通过 HAL_DMA_Init 函数初始化 DMA，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为 __HAL_LINKDMA。

5) 使能 DMA 对应数据流中断，配置 DMA 中断优先级并开启中断，启动 ADC 和 DMA

通过 HAL_ADC_Start_DMA 函数开启 ADC 转换，通过 DMA 传输结果。

通过 HAL_DMA_Start_IT 函数启动 DMA 读取，使能 DMA 中断。

通过 HAL_NVIC_EnableIRQ 函数使能 DMA 数据流中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

6) 编写中断服务函数

DMA 的每个数据流几乎都有一个中断服务函数，比如 DMA1_Channel1 的中断服务函数为 DMA1_Channel1_IRQHandler。简单的做法就是在，对应的中断服务函数里面，通过判断相关的中断标志位的方式，完成中断逻辑代码，最后清楚该中断标志位，本实验的做法就是如此。

还可以通过调用 HAL 库提供的 DMA 中断公用处理函数 HAL_DMA_IRQHandler，然后重新定义相关的中断回调处理函数。

30.3.3.2 程序流程图

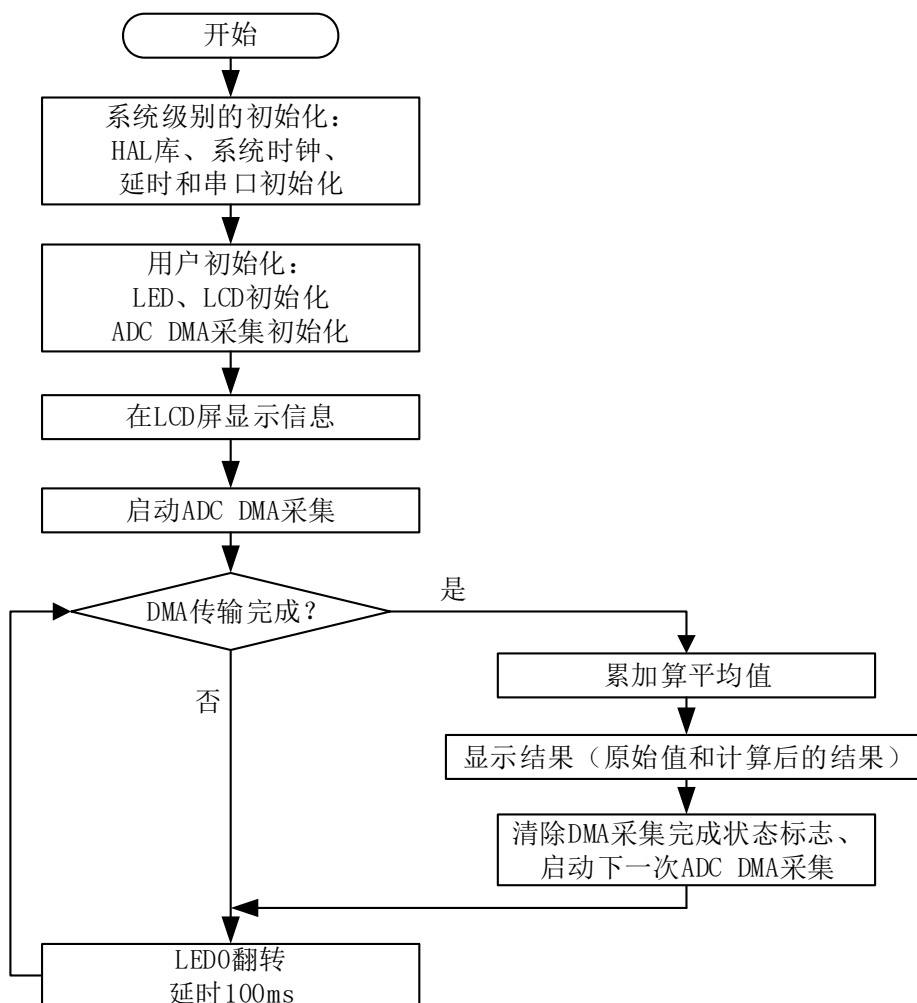


图 30.3.3.2.1 单通道 ADC 采集（DMA 读取）实验程序流程图

30.3.3.3 程序解析

由于本实验用到 DMA，所以在 adc.h 头文件定义了以下一些宏定义：

```

/* ADC 单通道/多通道 DMA 采集 DMA 及通道 定义
 * 注意：ADC1 的 DMA 通道只能是：DMA1_Channel1，因此只要是 ADC1，这里是不能改动的
 *      ADC2 不支持 DMA 采集
 *      ADC3 的 DMA 通道只能是：DMA2_Channel5，因此如果使用 ADC3 则需要修改
 */
#define ADC_ADCX_DMACx          DMA1_Channel1
#define ADC_ADCX_DMACx_IRQn     DMA1_Channel1_IRQn
#define ADC_ADCX_DMACx_IRQHandler DMA1_Channel1_IRQHandler

/*判断 DMA1_Channel1 传输完成标志，是一个假函数形式，不能当函数使用，只能用在 if 等语句里面*/
#define ADC_ADCX_DMACx_IS_TC()  ( DMA1->ISR & (1 << 1) )
/* 清除 DMA1_Channel1 传输完成标志 */
#define ADC_ADCX_DMACx_CLR_TC() do{ DMA1->IFCR |= 1 << 1; }while(0)
    
```

下面给大家介绍 adc.c 文件里面的函数，首先是 ADC DMA 读取初始化函数。

```

/**
 * @brief      ADC DMA 读取 初始化函数
 * @param      mar      : 存储器地址
 * @retval     无
 */
void adc_dma_init(uint32_t mar)
    
```

```

{
    GPIO_InitTypeDef gpio_init_struct;
    RCC_PeriphCLKInitTypeDef adc_clk_init = {0};
    ADC_ChannelConfTypeDef adc_ch_conf = {0};

    ADC_ADCX_CHY_CLK_ENABLE(); /* 使能 ADCx 时钟 */
    ADC_ADCX_CHY_GPIO_CLK_ENABLE(); /* 开启 GPIO 时钟 */

    /* 大于 DMA1_Channel17, 则为 DMA2 的通道了 */
    if ((uint32_t)ADC_ADCX_DMACH > (uint32_t)DMA1_Channel17)
    {
        __HAL_RCC_DMA2_CLK_ENABLE(); /* DMA2 时钟使能 */
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE(); /* DMA1 时钟使能 */
    }

    /* 设置 ADC 时钟 */
    adc_clk_init.PeriphClockSelection = RCC_PERIPHCLK_ADC; /* ADC 外设时钟 */
    /* 分频因子 6 时钟为 72M/6=12MHz */
    adc_clk_init.AdcClockSelection = RCC_ADCCLK2_DIV6;
    HAL_RCCEx_PeriphCLKConfig(&adc_clk_init); /* 设置 ADC 时钟 */

    /* 设置 AD 采集通道对应 IO 引脚工作模式 */
    gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN; /* ADC 通道对应的 IO 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_ANALOG; /* 模拟 */
    HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);

    /* 初始化 DMA */
    g_dma_adc_handle.Instance = ADC_ADCX_DMACH; /* 设置 DMA 通道 */
    g_dma_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY; /* 从外设到存储器模式 */
    g_dma_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
    g_dma_adc_handle.Init.MemInc = DMA_MINC_ENABLE; /* 存储器增量模式 */
    /* 外设数据长度:16 位 */
    g_dma_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_adc_handle.Init.Mode = DMA_NORMAL; /* 外设流控模式 */
    g_dma_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */
    HAL_DMA_Init(&g_dma_adc_handle);

    /* 将 DMA 与 adc 联系起来 */
    __HAL_LINKDMA(&g_adc_dma_handle, DMA_Handle, g_dma_adc_handle);

    g_adc_dma_handle.Instance = ADC_ADCX; /* 选择哪个 ADC */
    g_adc_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 数据对齐方式: 右对齐 */
    g_adc_dma_handle.Init.ScanConvMode = ADC_SCAN_DISABLE; /* 非扫描模式 */
    g_adc_dma_handle.Init.ContinuousConvMode = ENABLE; /* 使能连续转换模式 */
    g_adc_dma_handle.Init.NbrOfConversion = 1; /* 范围是 1~16, 这里用到 1 个规则序列 */
    g_adc_dma_handle.Init.DiscontinuousConvMode = DISABLE; /* 禁止规则组间断模式 */
    /* 配置间断模式的规则通道个数, 禁止规则通道组间断模式后, 此参数忽略 */
    g_adc_dma_handle.Init.NbrOfDiscConversion = 0;
    g_adc_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START; /* 软件触发 */
    HAL_ADC_Init(&g_adc_dma_handle); /* 初始化 */

    HAL_ADCEx_Calibration_Start(&g_adc_dma_handle); /* 校准 ADC */

    /* 配置 ADC 通道 */
    adc_ch_conf.Channel = ADC_ADCX_CHY; /* 通道 */
    adc_ch_conf.Rank = ADC_REGULAR_RANK_1; /* 序列 */
}

```

```

/* 采样时间,设置最大采样周期:239.5 个 ADC 周期 */
adc_ch_conf.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
HAL_ADC_ConfigChannel(&g_adc_dma_handle, &adc_ch_conf); /* 通道配置 */

/* 配置 DMA 数据流请求中断优先级 */
HAL_NVIC_SetPriority(ADC_ADCX_DMACx_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(ADC_ADCX_DMACx_IRQn);
/* 启动 DMA, 并开启中断 */
HAL_DMA_Start_IT(&g_adc_dma_handle, (uint32_t)&ADC1->DR, mar, 0);
HAL_ADC_Start_DMA(&g_adc_dma_handle, &mar, 0); /* 开启 ADC, 通过 DMA 传输结果 */
}

```

adc_dma_init 函数包含了输出通道对应 IO 的初始代码、NVIC、使能时钟、ADC 时钟预分频系数、ADC 工作参数和 ADC 通道配置等代码。下面来看看该函数的代码内容。

第一部分使能 ADC、DMA 和 GPIO 的时钟。

第二部分配置 ADC 时钟预分频系数为 6, 得到 ADC 的输入时钟频率是 12MHz。

第三部分是设置 ADC 采集通道对应 IO 引脚工作模式。

第四部分初始化 DMA, 并通过 __HAL_LINKDMA 宏定义将 DMA 相关的配置关联到 ADC 的句柄中。

第五部分是初始化 ADC, 并校准 ADC。

第六部分是配置 ADC 通道。

第七部分是配置 DMA 数据流请求中断优先级, 并使能该中断。

第八部分是启动 DMA 并开启 DMA 中断, 以及启动 ADC 并通过 DMA 传输转换结果。

为了方便代码的管理和移植性等, 这里就没有使用 HAL_ADC_MspInit 这个函数来存放使能时钟、GPIO、NVIC 相关的代码, 而是全部存放在 adc_dma_init 函数中。

接下来给大家介绍使能一次 ADC DMA 传输函数, 其定义如下:

```

/**
 * @brief      使能一次 ADC DMA 传输
 * @note       该函数用寄存器来操作, 防止用 HAL 库操作对其他参数有修改, 也为了兼容性
 * @param      ndtr: DMA 传输的次数
 * @retval     无
 */
void adc_dma_enable(uint16_t cndtr)
{
    ADC_ADCX->CR2 &= ~(1 << 0); /* 先关闭 ADC */

    ADC_ADCX_DMACx->CCR &= ~(1 << 0); /* 关闭 DMA 传输 */
    while (ADC_ADCX_DMACx->CCR & (1 << 0)); /* 确保 DMA 可以被设置 */
    ADC_ADCX_DMACx->CNDTR = cndtr; /* DMA 传输数据量 */
    ADC_ADCX_DMACx->CCR |= 1 << 0; /* 开启 DMA 传输 */

    ADC_ADCX->CR2 |= 1 << 0; /* 重新启动 ADC */
    ADC_ADCX->CR2 |= 1 << 22; /* 启动规则转换通道 */
}

```

该函数我们使用寄存器来操作, 防止用 HAL 库相关宏操作会对其它参数进行修改, 同时也是为了兼容后面的实验。HAL_DMA_Start_IT 函数已经配置好了 DMA 传输的源地址和目标地址, 本函数只需要调用 ADC_ADCX_DMACx->CNDTR = cndtr; 语句给 DMA_CNDTRx 寄存器写入要传输的数据量, 然后启动 DMA 就可以传输了。

下面介绍的是 ADC DMA 采集中断服务函数, 函数定义如下:

```

/**
 * @brief      ADC DMA 采集中断服务函数
 * @param      无
 * @retval     无
 */
void ADC_ADCX_DMACx_IRQHandler(void)
{
    if (ADC_ADCX_DMACx_IS_TC())
    {

```



```

        g_adc_dma_sta = 1;          /* 标记 DMA 传输完成 */
        ADC_ADCX_DMACx_CLR_TC();    /* 清除 DMA1 通道 1 传输完成中断 */
    }
}

```

在该函数里，通过判断 DMA 传输完成标志位是否是 1，是 1 就给 g_adc_dma_sta 变量赋值为 1，标记 DMA 传输完成，最后清除 DMA 的传输完成标志位。

最后在 main.c 里面编写如下代码：

```

#define ADC_DMA_BUF_SIZE      100      /* ADC DMA 采集 BUF 大小 */
uint16_t g_adc_dma_buf[ADC_DMA_BUF_SIZE]; /* ADC DMA BUF */
extern uint8_t g_adc_dma_sta;          /* DMA 传输状态标志, 0, 未完成; 1, 已完成 */

int main(void)
{
    uint16_t i;
    uint16_t adcx;
    uint32_t sum;
    float temp;

    HAL_Init();          /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72);      /* 延时初始化 */
    usart_init(115200);  /* 串口初始化为 115200 */
    led_init();          /* 初始化 LED */
    lcd_init();          /* 初始化 LCD */
    adc_dma_init((uint32_t)&g_adc_dma_buf); /* 初始化 ADC DMA 采集 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC DMA TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH1_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH1_VOL:0.000V", BLUE);
    adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动 ADC DMA 采集 */

    while (1)
    {
        if (g_adc_dma_sta == 1)
        {
            /* 计算 DMA 采集到的 ADC 数据的平均值 */
            sum = 0;
            for (i = 0; i < ADC_DMA_BUF_SIZE; i++) /* 累加 */
            {
                sum += g_adc_dma_buf[i];
            }
            adcx = sum / ADC_DMA_BUF_SIZE;          /* 取平均值 */
            /* 显示结果 */
            lcd_show_xnum(134, 110, adcx, 4, 16, 0, BLUE); /* 显示 ADCC 采样后的原始值 */
            temp = (float)adcx * (3.3 / 4096); /* 获取计算后的带小数的实际电压值, 比如 3.1111 */
            adcx = temp; /* 赋值整数部分给 adcx 变量, 因为 adcx 为 u16 整形 */
            /* 显示电压值的整数部分, 3.1111 的话, 这里就是显示 3 */
            lcd_show_xnum(134, 130, adcx, 1, 16, 0, BLUE);

            temp -= adcx; /* 把已经显示的整数部分去掉, 留下小数部分, 比如 3.1111-3=0.1111 */
            temp *= 1000; /* 小数部分乘以 1000, 例如: 0.1111 就转换为 111.1, 相当于保留三位小数 */
            /* 显示小数部分 (前面转换为了整形显示), 这里显示的就是 111. */
            lcd_show_xnum(150, 130, temp, 3, 16, 0x80, BLUE);
            g_adc_dma_sta = 0; /* 清除 DMA 采集完成状态标志 */
            adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动下一次 ADC DMA 采集 */
        }
        LED0_TOGGLE();
        delay_ms(100);
    }
}

```

```
}  
}
```

此部分代码，和单通道 ADC 采集实验十分相似，只是这里使能了 DMA 传输数据，DMA 传输的数据存放在 `g_adc_dma_buf` 数组里，这里我们对数组的数据取平均值，减少误差。在 LCD 屏显示结果的处理和单通道 ADC 采集实验一样。首先在液晶固定位置显示了小数点，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就能在液晶上面显示转换结果的整数和小数部分。

30.3.4 下载验证

下载代码后，可以看到 LCD 显示如图 30.3.4.1 所示：



图 30.3.4.1 单通道 ADC 采集（DMA 读取）实验测试图

上图中，我们使用短路帽将 ADC 和 RV1 排针连接，使得 PA1 连接到电位器上，测试电位器的电压，并可以通过螺丝刀调节电位器改变电压值，范围：0~3.3V。LED0 闪烁，提示程序运行。

大家也可以用杜邦线将 ADC 排针接到其它待测量的电压点，看看测量到的电压值是否准确？**但是要注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

30.4 多通道 ADC 采集（DMA 读取）实验

本实验我们来学习多通道 ADC 采集（DMA 读取）实验。本实验使用规则组多通道的连续转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。由于使用连续转换模式，所以使用 DMA 读取转换结果的方式。下面先带大家来了解本实验要配置的寄存器。

30.4.1 ADC 寄存器

本实验我们很多的设置和单通道 ADC 采集（DMA 读取）实验是一样的，所以下面介绍寄存器的时候我们不会继续全部都介绍，而是针对性选择与单通道 ADC 采集（DMA 读取）实验不同设置的 ADC_SQRx 寄存器进行介绍，其他的配置基本一样的。另外我们用到 DMA 读取数据，配置上和单通道 ADC 采集（DMA 读取）实验是一样的。

ADC 规则序列寄存器有四个（ADC_SQR1~ADC_SQR3），具体怎么配置，需要看我们用多少个通道，比如本实验我们使用 6 个通道同时采集 ADC 数据，具体配置如下：

● ADC 规则序列寄存器 1（ADC_SQR1）

ADC 规则序列寄存器 1 描述如图 30.4.1.1 所示：

| | | | | | | | | | | | | | | | |
|------------|-----------|---|----|----|----|-----------|----|--------|----|----|-----------|-----------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | L[3:0] | | | | SQ16[4:1] | | | |
| | | | | | | | | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SQ16 _0 | SQ15[4:0] | | | | | SQ14[4:0] | | | | | SQ13[4:0] | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位23:20 | | L[3:0]: 规则通道序列长度 (Regular channel sequence length) 这些位由软件定义在规则通道转换序列中的通道数目。 0000: 1个转换 0001: 2个转换 1111: 16个转换 | | | | | | | | | | | | | |
| 位19:15 | | SQ16[4:0]: 规则序列中的第16个转换 (16th conversion in regular sequence) 这些位由软件定义转换序列中的第16个转换通道的编号(0~17)。 | | | | | | | | | | | | | |
| 位14:10 | | SQ15[4:0]: 规则序列中的第15个转换 (15th conversion in regular sequence) | | | | | | | | | | | | | |
| 位9:5 | | SQ14[4:0]: 规则序列中的第14个转换 (14th conversion in regular sequence) | | | | | | | | | | | | | |
| 位4:0 | | SQ13[4:0]: 规则序列中的第13个转换 (13th conversion in regular sequence) | | | | | | | | | | | | | |

图 30.4.1.1 ADC_SQR1 寄存器

L[3:0]位用于设置规则序列的长度，取值范围：0~15，表示规则序列长度为 1~16。本实验使用到 6 个通道，所以设置这几个位的值为 5 即可。

SQ13[4:0]~SQ16[4:0]位设置规则组序列的第 13~16 个转换编号，第 1~12 个转换编号的设置请查看 ADC_SQR2 和 ADC_SQR3 寄存器。设置过程非常简单，忘记了请参考前面给大家整理出来的规则序列寄存器控制关系汇总表。

下面我们来看看本实验是怎么设置的：SQ1[4:0]位赋值为 0、SQ2[4:0]位赋值为 1、SQ3[4:0]位赋值为 2、SQ4[4:0]位赋值为 3、SQ5[4:0]位赋值为 4、SQ6[4:0]位赋值为 5，即规则序列 1 到 6 分别对应的输入通道是 0 到 5。SQ1~SQ6 位都是在 ADC_SQR3 寄存器中配置。

30.4.2 硬件设计

1. 例程功能

使用 ADC1 采集（DMA 读取）通道 1\2\3\4\5\6 的电压，在 LCD 模块上面显示对应的 ADC 转换值以及换算成电压后的电压值。可以使用杜邦线连接 PA0\PA1\PA2\PA3\PA4\PA5 到你想要测量的电压源（0~3.3V），然后通过 TFTLCD 显示的电压值。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PE5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) ADC1 : 通道 1 - PA0、通道 2 - PA1、通道 3 - PA2、
通道 4 - PA3、通道 5 - PA4、通道 6 - PA5
- 5) DMA (DMA1 通道 1)

3. 原理图

ADC 和 DMA 属于 STM32F103 内部资源, 实际上我们只需要软件设置就可以正常工作, 另外还需要将待测量的电压源连接到 ADC 通道上, 以便 ADC 测量。本实验, 我们通过 ADC1 的通道 1\2\3\4\5\6 来采集外部电压值, 并通过 DMA 来读取。

30.4.3 程序设计

30.4.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过, 具体调用情况请看程序解析部分。下面介绍多通道 ADC 采集 (DMA 读取) 配置步骤。

多通道 ADC 采集 (DMA 读取) 配置步骤

1) 开启 ADCx 和 ADC 通道对应的 IO 时钟, 并配置该 IO 为模拟功能

首先开启 ADCx 的时钟, 然后配置 GPIO 为模拟模式。本实验我们默认用到 ADC1 通道 0、1、2、3、4、5, 对应 IO 是 PA0、PA1、PA2、PA3、PA4 和 PA5, 它们的时钟开启方法如下:

```
HAL_RCC_ADC1_CLK_ENABLE(); /* 使能 ADC1 时钟 */
HAL_RCC_GPIOA_CLK_ENABLE(); /* 开启 GPIOA 时钟 */
```

2) 初始化 ADCx, 配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意: 该函数会调用: HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。我们也可以不存放在这个函数里, 本实验就没用到这个 MSP 回调函数。

3) 配置 ADC 通道并启动 AD 转换器

在 HAL 库中, 通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道, 并设置规则序列、采样时间等。

配置好 ADC 通道之后, 通过 HAL_ADC_Start 函数启动 AD 转换器。

4) 初始化 DMA

通过 HAL_DMA_Init 函数初始化 DMA, 包括配置通道, 外设地址, 存储器地址, 传输数据量等。

HAL 库为了处理各类外设的 DMA 请求, 在调用相关函数之前, 需要调用一个宏定义标识符, 来连接 DMA 和外设句柄。这个宏定义为 __HAL_LINKDMA。

5) 使能 DMA 对应数据流中断, 配置 DMA 中断优先级, 使能 ADC, 使能并启动 DMA

通过 HAL_ADC_Start_DMA 函数开启 ADC 转换, 通过 DMA 传输结果。

通过 HAL_DMA_Start_IT 函数启动 DMA 读取, 使能 DMA 中断。

通过 HAL_NVIC_EnableIRQ 函数使能 DMA 数据流中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

6) 编写中断服务函数

DMA 的每个数据流几乎都有一个中断服务函数, 比如 DMA1_Channel1 的中断服务函数为 DMA1_Channel1_IRQHandler。简单的做法就是在, 对应的中断服务函数里面, 通过判

断相关的中断标志位的方式，完成中断逻辑代码，最后清楚该中断标志位，本实验的做法就是如此。

还可以通过调用 HAL 库提供的 DMA 中断公用处理函数 HAL_DMA_IRQHandler，然后重新定义相关的中断回调处理函数。

30.4.3.2 程序流程图

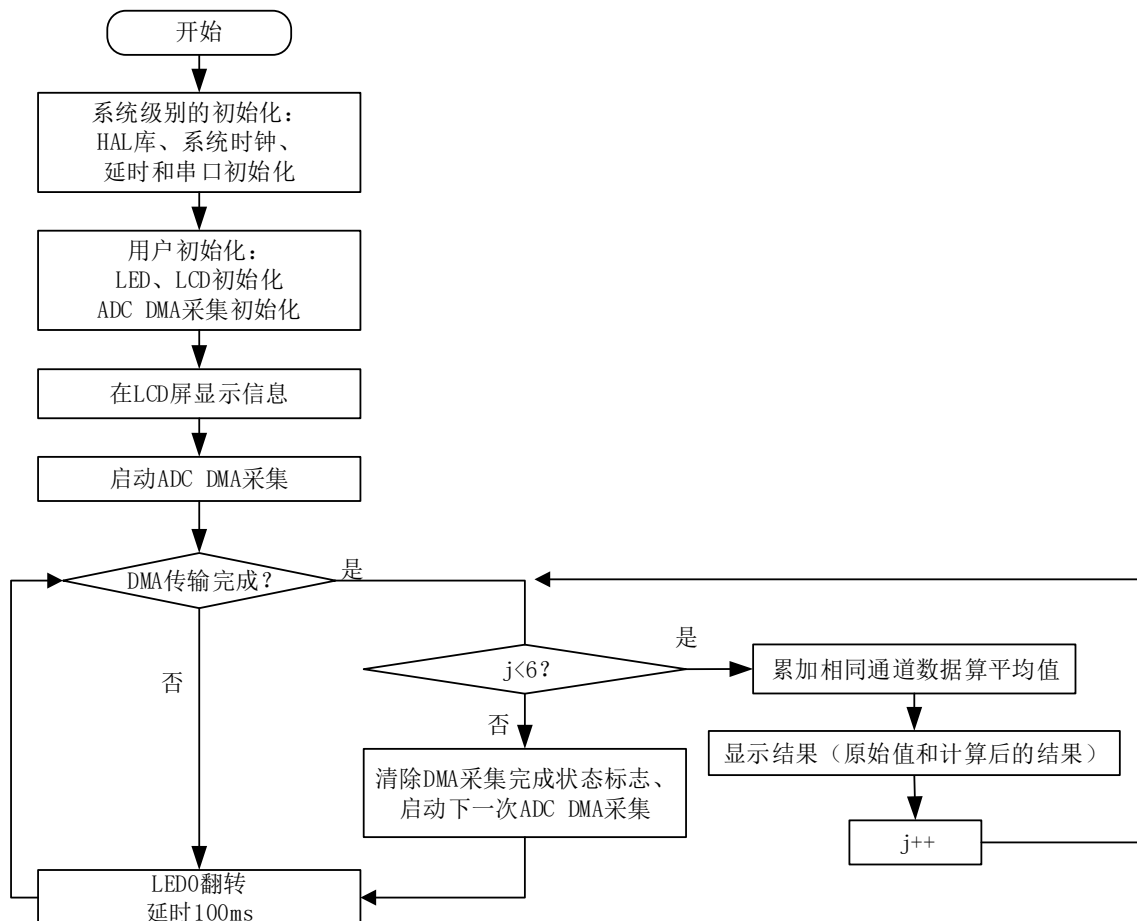


图 30.4.3.2.1 多通道 ADC 采集（DMA 读取）实验程序流程图

30.4.3.3 程序解析

在本实验中 adc.h 头文件只是添加了一些函数声明，下面开始介绍 adc.c 的函数，本实验只增加了一个函数，ADC 的 N 通道(6 通道) DMA 读取初始化函数，其定义如下：

```

/**
 * @brief      ADC N 通道(6 通道) DMA 读取 初始化函数
 * @note      由于本函数用到了 6 个通道，宏定义会比较多内容，
 *            因此，本函数就不采用宏定义方式来修改通道了，
 *            直接在本函数里面修改，这里我们默认使用 PA0~PA5 这 6 个通道。
 *
 *            注意：本函数还是使用 ADC_ADCX (默认=ADC1) ADC_ADCX_DMACH (DMA1_Channel1)
 *            及其相关定义。不要乱修改 adc.h 里面的这两部分内容，必须在理解原理的基础上进行修
 *            改，否则可能导致无法正常使用。
 * @param     mar      : 存储器地址
 * @retval    无
 */
void adc_nch_dma_init(uint32_t mar)
{
    GPIO_InitTypeDef gpio_init_struct;

```

```

RCC_PeriphCLKInitTypeDef adc_clk_init = {0};
ADC_ChannelConfTypeDef adc_ch_conf = {0};

ADC_ADCX_CHY_CLK_ENABLE(); /* 使能 ADCx 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE(); /* 开启 GPIOA 时钟 */

/* 大于 DMA1_Channel17, 则为 DMA2 的通道了 */
if ((uint32_t)ADC_ADCX_DMACH > (uint32_t)DMA1_Channel17)
{
    __HAL_RCC_DMA2_CLK_ENABLE(); /* DMA2 时钟使能 */
}
else
{
    __HAL_RCC_DMA1_CLK_ENABLE(); /* DMA1 时钟使能 */
}

/* 设置 ADC 时钟 */
adc_clk_init.PeriphClockSelection = RCC_PERIPHCLK_ADC; /* ADC 外设时钟 */
/* 分频因子 6 时钟为 72M/6=12MHz */
adc_clk_init.AdcClockSelection = RCC_ADCCLK2_DIV6;
HAL_RCCEx_PeriphCLKConfig(&adc_clk_init); /* 设置 ADC 时钟 */

/*
    设置 ADC1 通道 0~5 对应的 IO 口模拟输入
    AD 采集引脚模式设置, 模拟输入
    PA0 对应 ADC1_IN0
    PA1 对应 ADC1_IN1
    PA2 对应 ADC1_IN2
    PA3 对应 ADC1_IN3
    PA4 对应 ADC1_IN4
    PA5 对应 ADC1_IN5
*/
gpio_init_struct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3
                    |GPIO_PIN_4|GPIO_PIN_5; /* GPIOA0~5 */
gpio_init_struct.Mode = GPIO_MODE_ANALOG; /* 模拟 */
HAL_GPIO_Init(GPIOA, &gpio_init_struct);

/* 初始化 DMA */
g_dma_nch_adc_handle.Instance = ADC_ADCX_DMACH; /* 设置 DMA 通道 */
/* 从外设到存储器模式 */
g_dma_nch_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
g_dma_nch_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
g_dma_nch_adc_handle.Init.MemInc = DMA_MINC_ENABLE; /* 存储器增量模式 */
/* 外设数据长度:16 位 */
g_dma_nch_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
/* 存储器数据长度:16 位 */
g_dma_nch_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
g_dma_nch_adc_handle.Init.Mode = DMA_NORMAL; /* 外设流控模式 */
g_dma_nch_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */
HAL_DMA_Init(&g_dma_nch_adc_handle);

/* 将 DMA 与 adc 联系起来 */
__HAL_LINKDMA(&g_adc_nch_dma_handle, DMA_Handle, g_dma_nch_adc_handle);

/* 初始化 ADC */
g_adc_nch_dma_handle.Instance = ADC_ADCX; /* 选择哪个 ADC */
g_adc_nch_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 数据右对齐 */
g_adc_nch_dma_handle.Init.ScanConvMode = ADC_SCAN_ENABLE; /* 使能扫描模式 */
g_adc_nch_dma_handle.Init.ContinuousConvMode = ENABLE; /* 使能连续转换 */
/* 赋值范围是 1~16, 本实验用到 6 个规则通道序列 */
g_adc_nch_dma_handle.Init.NbrOfConversion = 6;
    
```



```

/* 禁止规则组间断模式 */
g_adc_nch_dma_handle.Init.DiscontinuousConvMode = DISABLE;
/* 配置间断模式的规则通道个数，禁止规则通道间断模式后，此参数忽略 */
g_adc_nch_dma_handle.Init.NbrOfDiscConversion = 0;
g_adc_nch_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START; /* 软件触发 */
HAL_ADC_Init(&g_adc_nch_dma_handle); /* 初始化 */

HAL_ADCEx_Calibration_Start(&g_adc_nch_dma_handle); /* 校准 ADC */

/* 配置 ADC 通道 */
adc_ch_conf.Channel = ADC_CHANNEL_0; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_1; /* 采样序列里的第 1 个 */
/* 采样时间，设置最大采样周期:239.5 个 ADC 周期 */
adc_ch_conf.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

adc_ch_conf.Channel = ADC_CHANNEL_1; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_2; /* 采样序列里的第 2 个 */
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

adc_ch_conf.Channel = ADC_CHANNEL_2; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_3; /* 采样序列里的第 3 个 */
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

adc_ch_conf.Channel = ADC_CHANNEL_3; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_4; /* 采样序列里的第 4 个 */
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

adc_ch_conf.Channel = ADC_CHANNEL_4; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_5; /* 采样序列里的第 5 个 */
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

adc_ch_conf.Channel = ADC_CHANNEL_5; /* 配置使用的 ADC 通道 */
adc_ch_conf.Rank = ADC_REGULAR_RANK_6; /* 采样序列里的第 6 个 */
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf); /* 配置 ADC 通道 */

/* 配置 DMA 数据流请求中断优先级 */
HAL_NVIC_SetPriority(ADC_ADCX_DMACH_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(ADC_ADCX_DMACH_IRQn);

/* 启动 DMA，并开启中断 */
HAL_DMA_Start_IT(&g_dma_nch_adc_handle, (uint32_t)&ADC1->DR, mar, 0);
/* 开启 ADC，通过 DMA 传输结果 */
HAL_ADC_Start_DMA(&g_adc_nch_dma_handle, &mar, 0);
}

```

adc_nch_dma_init 函数包含了输出通道对应 IO 的初始代码、NVIC、使能时钟、ADC 时钟预分频系数、ADC 工作参数和 ADC 通道配置等代码。大部分代码和单通道 ADC 采集(DMA 读取)实验一样，下面来看看该函数的代码内容。

第一部分使能 ADC、DMA 和 GPIO 的时钟。

第二部分配置 ADC 时钟预分频系数为 6，得到 ADC 的输入时钟频率是 12MHz。

第三部分是设置 ADC 采集通道对应 IO 引脚工作模式，这里用到 6 个通道。

第四部分初始化 DMA，并通过 __HAL_LINKDMA 宏定义将 DMA 相关的配置关联到 ADC 的句柄中。

第五部分是初始化 ADC，并校准 ADC。

第六部分是配置 ADC 通道，这里有 6 个通道需要配置。

第七部分是配置 DMA 数据流请求中断优先级，并使能该中断。

第八部分是启动 DMA 并开启 DMA 中断，以及启动 ADC 并通过 DMA 传输转换结果。

为了方便代码的管理和移植性等，这里就没有使用 HAL_ADC_MspInit 这个函数来存放使

能时钟、GPIO、NVIC 相关的代码，而是全部存放在 adc_nch_dma_init 函数中。

最后在 main.c 里面编写如下代码：

```
#define ADC_DMA_BUF_SIZE 50 * 6 /* ADC DMA 采集 BUF 大小，应等于 ADC 通道数的整数倍 */
uint16_t g_adc_dma_buf[ADC_DMA_BUF_SIZE]; /* ADC DMA BUF */
extern uint8_t g_adc_dma_sta; /* DMA 传输状态标志，0,未完成；1,已完成 */

int main(void)
{
    uint16_t i,j;
    uint16_t adcx;
    uint32_t sum;
    float temp;

    sys_stm32_clock_init(9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(72, 115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    adc_nch_dma_init((uint32_t)&g_adc_dma_buf); /* 初始化 ADC DMA 采集 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC 6CH DMA TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    lcd_show_string(30, 110, 200, 12, 12, "ADC1_CH0_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 122, 200, 12, 12, "ADC1_CH0_VOL:0.000V", BLUE);

    lcd_show_string(30, 140, 200, 12, 12, "ADC1_CH1_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 152, 200, 12, 12, "ADC1_CH1_VOL:0.000V", BLUE);

    lcd_show_string(30, 170, 200, 12, 12, "ADC1_CH2_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 182, 200, 12, 12, "ADC1_CH2_VOL:0.000V", BLUE);

    lcd_show_string(30, 200, 200, 12, 12, "ADC1_CH3_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 212, 200, 12, 12, "ADC1_CH3_VOL:0.000V", BLUE);

    lcd_show_string(30, 230, 200, 12, 12, "ADC1_CH4_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 242, 200, 12, 12, "ADC1_CH4_VOL:0.000V", BLUE);

    lcd_show_string(30, 260, 200, 12, 12, "ADC1_CH5_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 272, 200, 12, 12, "ADC1_CH5_VOL:0.000V", BLUE);

    adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动 ADC DMA 采集 */

    while (1)
    {
        if (g_adc_dma_sta == 1)
        {
            /* 循环显示通道 0~通道 5 的结果 */
            for(j = 0; j < 6; j++) /* 遍历 6 个通道 */
            {
                sum = 0; /* 清零 */
                for (i = 0; i < ADC_DMA_BUF_SIZE / 6; i++)
                { /* 每个通道采集了 10 次数据,进行 10 次累加 */
                    sum += g_adc_dma_buf[(6 * i) + j]; /* 相同通道的转换数据累加 */
                }
            }
        }
    }
}
```

```

        adcx = sum / (ADC_DMA_BUF_SIZE / 6); /* 取平均值 */

        /* 显示结果 */
        /* 显示 ADCC 采样后的原始值 */
        lcd_show_xnum(108, 120 + (j * 30), adcx, 4, 12, 0, BLUE);
        /* 获取计算后的带小数的实际电压值, 比如 3.1111 */
        temp = (float)adcx * (3.3 / 4096);
        adcx = temp; /* 赋值整数部分给 adcx 变量, 因为 adcx 为 u16 整形 */
        /* 显示电压值的整数部分, 3.1111 的话, 这里就是显示 3 */
        lcd_show_xnum(108, 122 + (j * 30), adcx, 1, 12, 0, BLUE);

        /* 把已经显示的整数部分去掉, 留下小数部分, 比如 3.1111-3=0.1111 */
        temp -= adcx;
        /* 小数部分乘以 1000, 例如: 0.1111 就转换为 111.1, 相当于保留三位小数。 */
        temp *= 1000;
        /* 显示小数部分 (前面转换为了整形显示), 这里显示的就是 111. */
        lcd_show_xnum(120, 122 + (j * 30), temp, 3, 12, 0x80, BLUE);
    }
    g_adc_dma_sta = 0; /* 清除 DMA 采集完成状态标志 */
    adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动下一次 ADC DMA 采集 */
}
LED0_TOGGLE();
delay_ms(100);
}
}

```

这里使用了 DMA 传输数据, DMA 传输的数据存放在 g_adc_dma_buf 数组里, 该数组的大小是 50 * 6。本实验用到 6 个通道, 每个通道使用 50 个 uint16_t 大小的空间存放 ADC 的结果。输入通道 0 的转换数据存放在 g_adc_dma_buf[0]到 g_adc_dma_buf[49], 输入通道 1 的转换数据存放在 g_adc_dma_buf[50]到 g_adc_dma_buf[99], 后面的以此类推。然后对数组的每个通道的数据取平均值, 减少误差。最后在 LCD 屏上显示 ADC 的转换值和换算成电压后的电压值。

30.4.4 下载验证

下载代码后, LED0 闪烁, 提示程序运行。可以看到 LCD 显示如图 30.4.4.1 所示:

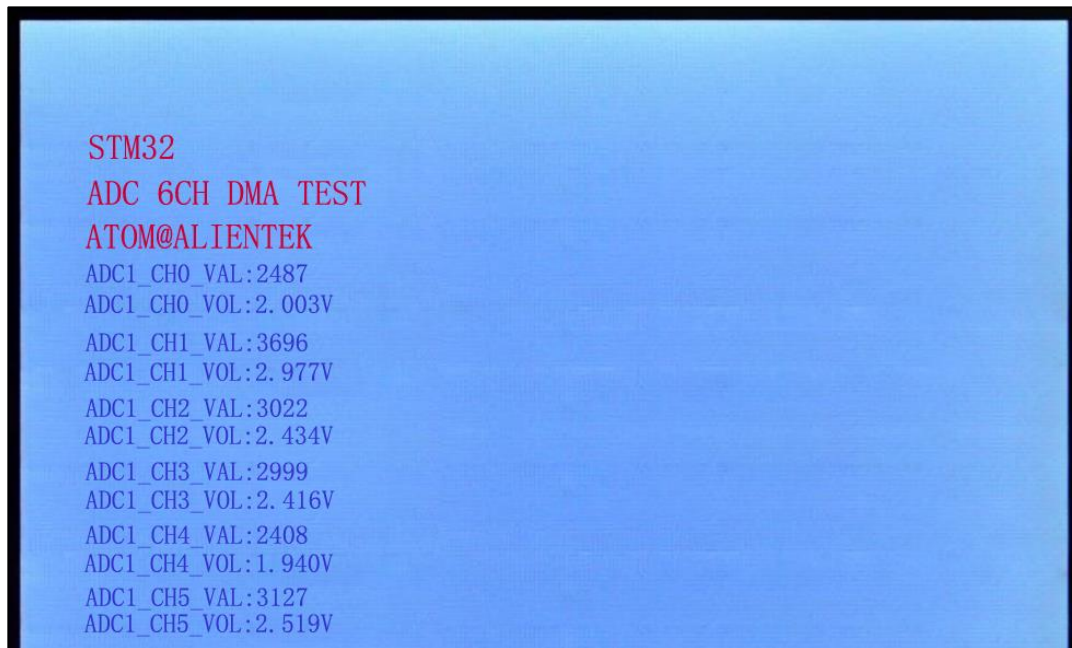


图 30.4.4.1 多通道 ADC 采集 (DMA 读取) 实验测试图

使用 ADC1 采集 (DMA 读取) 通道 0\1\2\3\4\5 的电压, 在 LCD 模块上面显示对应的 ADC 转换值以及换算成电压后的电压值。可以使用杜邦线连接 PA0\PA1\PA2\PA3\PA4\PA5 到你测

量的电压源（0~3.3V）。

这6个通道对应引出来的引脚PA0\PA1\PA2\PA3\PA4\PA5在开发板上的位置，如下图所示：

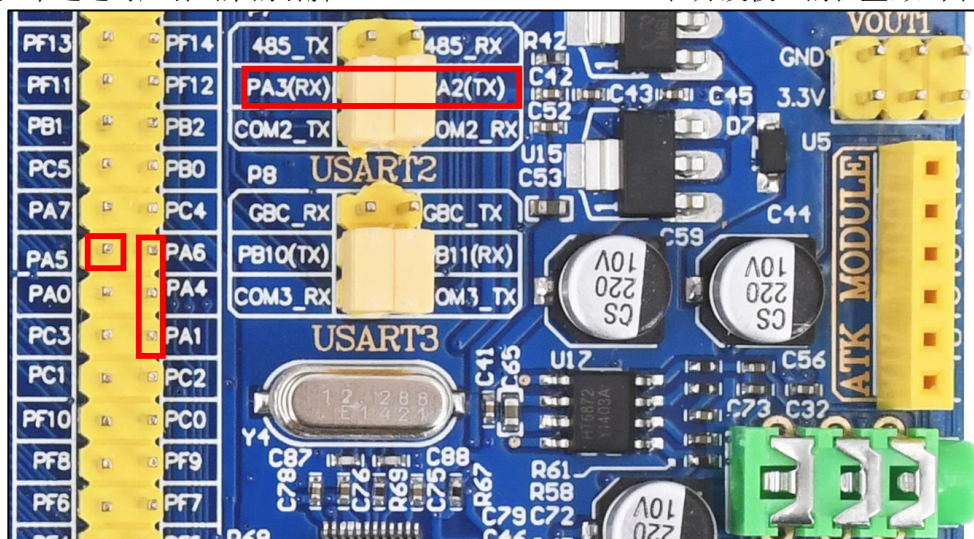


图 30.4.4.2 ADC1 的通道 0\1\2\3\4\5 引脚在开发板位置示意图

这六个通道可以同时测量不同测试点的电压，只需要用杜邦线分别接到不同的电压测试点即可。**注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

30.5 单通道 ADC 过采样（16 位分辨率）实验

本实验我们来学习单通道 ADC 过采样（16 位分辨率）实验。STM32F103 自带的 ADC 分辨率只有 12 位，虽然可以满足一般的应用，但是有些场合可能需要更高的分辨率，怎么办呢？可以使用外部专用的 ADC，或者换一个带更高分辨率 ADC 的主控芯片。这样做往往会增加额外的成本，那么有没有其它办法呢？答案是有的，可以通过引入过采样技术来实现。

ADC 过采样技术，是利用 ADC 多次采集的方式，来提高 ADC 分辨率。下面，简单介绍一下怎么提高 ADC 测量的分辨率？

下面直接给大家介绍一个方程，根据要增加分辨率计算过采样频率方程，方程如下：

$$f_{os} = 4^w \cdot f_s$$

其中，w 是希望增加的分辨率位数， f_s 是初始采样频率要求， f_{os} 是过采样频率。

方程的推导过程比较复杂，这里就不带大家去推导，感兴趣的朋友可以通过下面这个链接自行学习：<https://max.book118.com/html/2018/0506/165038217.shtm>。

由该方程可以知道，采样速度每提高 4 倍，分辨率位数可以提高 1 位。结合 ADC 的实际情况，换个思路来说，分辨率位数每提高 1 位，如果采样频率不变的情况下，那么采样速度就会降低 4 倍。本实验要求得到 16 位分辨率，即需要增加 4 位分辨率，那么采样速度就会降低 256 倍，即需要采集 256 次才能得出 1 次数据，相当于 ADC 速度慢了 256 倍。

理论上只要 ADC 足够快，我们可以无限提高 ADC 精度，但实际上 ADC 并不是无限快的，而且由于 ADC 性能限制，并不是位数无限提高，结果就越好，需要根据自己的实际需求和 ADC 的实际性能来权衡的。

下面来看一下我们怎么实现单通道 ADC 过采样（16 位分辨率）实验的？。

30.5.1 ADC 寄存器

本实验我们很多的设置和单通道 ADC 采集（DMA 读取）实验是一样的，代码实现也是基于该实验实现的，寄存器的介绍请参考前面的 ADC 实验。

30.5.2 硬件设计

1. 例程功能

使用 ADC1 通道 1（PA1），通过软件方式实现 16 位分辨率采集外部电压，并在 LCD 模块上面显示对应的 ADC 转换值以及换算成电压后的电压值。可以使用杜邦线连接 PA1 到你想要测量的电压源（0~3.3V），然后通过 TFTLCD 显示的电压值。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) ADC1 : 通道 1 - PA1

3. 原理图

ADC 属于 STM32F103 内部资源，实际上我们只需要软件设置就可以正常工作，另外还需要将待测量的电压源连接到 ADC 通道上，以便 ADC 测量。本实验，我们通过 ADC1 通道 1（PA1）来采集外部电压值。开发板有一个电位器，可调节的电压范围是：0~3.3V，可以通过断路帽将 PA1 与电位器连接，从而测量电位器的电压。

30.5.3 程序设计

30.5.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。

30.5.3.2 程序流程图

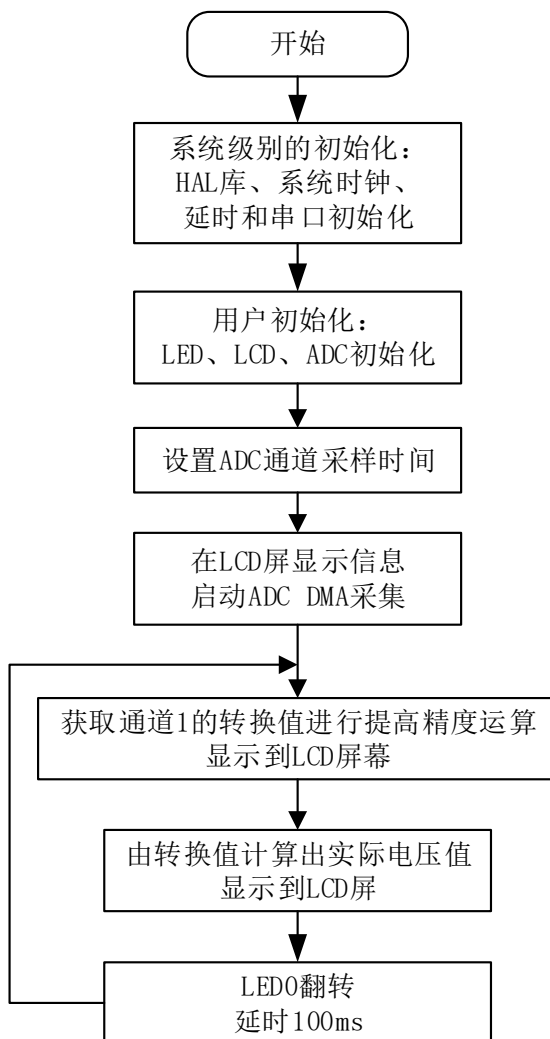


图 30.5.3.2.1 单通道 ADC 过采样（16 位分辨率）实验程序流程图

30.5.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。ADC 驱动源码包括两个文件：adc.c 和 adc.h。本实验沿用前面实验中的函数，并没有改动。

下面介绍一下 main.c 里面编写的代码：

```
/* ADC 过采样次数，这里提高 4bit 分辨率，需要 256 倍采样 */
#define ADC_OVERSAMPLE_TIMES 256
/* ADC DMA 采集 BUF 大小，应等于过采样次数的整数倍 */
#define ADC_DMA_BUF_SIZE ADC_OVERSAMPLE_TIMES * 10

uint16_t g_adc_dma_buf[ADC_DMA_BUF_SIZE]; /* ADC DMA BUF */

extern uint8_t g_adc_dma_sta; /* DMA 传输状态标志，0，未完成；1，已完成 */
extern ADC_HandleTypeDef g_adc_dma_handle; /* ADC（DMA 读取）句柄 */
```



```
int main(void)
{
    uint16_t i;
    uint32_t adcx;
    uint32_t sum;
    float temp;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    adc_dma_init((uint32_t)&g_adc_dma_buf); /* 初始化 ADC DMA 采集 */
    /* 设置 ADCX 对应通道采样时间为 1.5 个时钟周期, 已达到最高的采集速度 */
    adc_channel_set(&g_adc_handle, ADC_ADCX_CHY, ADC_REGULAR_RANK_1,
        ADC_SAMPLETIME_1CYCLE_5);

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC OverSample TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH1_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH1_VOL:0.000V", BLUE);

    adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动 ADC DMA 采集 */

    while (1)
    {
        if (g_adc_dma_sta == 1)
        {
            /* 计算 DMA 采集到的 ADC 数据的平均值 */
            sum = 0;

            for (i = 0; i < ADC_DMA_BUF_SIZE; i++) /* 累加 */
            {
                sum += g_adc_dma_buf[i];
            }

            adcx = sum / (ADC_DMA_BUF_SIZE / ADC_OVERSAMPLE_TIMES); /* 取平均值 */
            /* 除以 2^4 倍, 得到 12+4 位 ADC 精度值, 注意: 提高 N bit 精度, 需要 >> N */
            adcx >>= 4;

            /* 显示 ADCC 采样后的原始值 */
            lcd_show_xnum(134, 110, adcx, 5, 16, 0, BLUE);
            /* 获取计算后的带小数的实际电压值, 比如 3.1111 */
            temp = (float)adcx * (3.3 / 65536);
            adcx = temp; /* 赋值整数部分给 adcx 变量, 因为 adcx 为 u16 整形 */
            /* 显示电压值的整数部分, 3.1111 的话, 这里就是显示 3 */
            lcd_show_xnum(134, 130, adcx, 1, 16, 0, BLUE);

            temp -= adcx; /* 把已经显示的整数部分去掉, 留下小数部分, 比如 3.1111-3=0.1111 */
            temp *= 1000; /* 小数部分乘以 1000, 例如: 0.1111 就转换为 111.1, 相当于保留三位小数 */
            /* 显示小数部分 (前面转换为了整形显示), 这里显示的就是 111. */
            lcd_show_xnum(150, 130, temp, 3, 16, 0x80, BLUE);
            g_adc_dma_sta = 0; /* 清除 DMA 采集完成状态标志 */
            adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动下一次 ADC DMA 采集 */
        }

        LED0_TOGGLE();
        delay_ms(100);
    }
}
```

```
}
```

上面的代码中，ADC_OVERSAMPLE_TIMES 宏定义表示为了提高 4 位分辨率，ADC 需要进行 256 次采样才能得的一次 16 位分辨率的数据。为了减少误差，ADC_DMA_BUF_SIZE 宏定义是 ADC_OVERSAMPLE_TIMES 的 10 倍，为了后期取 16 位转换结果平均值的。g_adc_dma_buf 数组是 uint16_t 类型的，用于存放转换结果。

为了提高 ADC 的采样速度，调用 adc_channel_set 函数将采样时间调整为 1.5 个 ADC 时钟周期，以得到最高的采样速度。

adcx = sum / (ADC_DMA_BUF_SIZE / ADC_OVERSAMPLE_TIMES); 语句可以得到 ADC 采样 10 次的 16 位分辨率转换结果的平均值。adcx >>= 4; 语句对该平均值右移 4 位，这个过程通常被称为抽取。这样就可以得到 16 位有用的数据，该数据的取值范围是 0~65535，这个操作被称为累加和转储。

接下来的代码就是在 LCD 屏显示转换值和换算的电压值，以及让 LED0 闪烁，提示系统正在运行。

30.5.4 下载验证

下载代码后，LED0 闪烁，提示程序运行。可以看到 LCD 显示如图 30.5.4.1 所示：



图 30.5.4.1 单通道 ADC 过采样（16 位分辨率）实验测试图

上图中，我们使用短路帽将 ADC 和 RV1 排针连接，使得 PA1 连接到电位器上，测试电位器的电压，并可以通过螺丝刀调节电位器改变电压值，范围：0~3.3V。LED0 闪烁，提示程序运行。

大家也可以用杜邦线将 ADC 排针接到其它待测量的电压点，看看测量到的电压值是否准确？**但是要注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

第三十一章 内部温度传感器实验

本章，我们将介绍 STM32F103 的内部温度传感器并使用它来读取温度值，然后在 LCD 模块上显示出来。

本章分为如下几个小节：

- 31.1 内部温度传感器简介
- 31.2 硬件设计
- 31.3 程序设计
- 31.4 下载验证

31.1 内部温度传感器简介

STM32 有一个内部温度传感器，可以用来测量 CPU 及周围的温度(内部温度传感器更适用于检测温度的变化，需要测量精确温度的情况下，应使用外置传感器)。对于 STM32F103 来说，该温度传感器在内部和 ADC1_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值。温度传感器模拟输入推荐采样时间是 17.1us。STM32F103 内部温度传感器支持的温度范围为：-40~125 度。精度为±1.5℃左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部温度传感器通道就差不多了。关于 ADC 的设置，我们在上一章已经进行了详细的介绍，这里就不再多说。接下来我们介绍一下和温度传感器设置相关的两个地方。

第一个地方，我们要使用 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CR2 的 AWDEN 位 (bit23) 设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32 的内部温度传感器固定的连接在 ADC1 的通道 16 上，所以，我们在设置好 ADC1 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。计算公式如下：

$$T(^{\circ}\text{C}) = \{ (V25 - V_{\text{sense}}) / \text{Avg_Slope} \} + 25$$

式中：

V25 = Vsense 在 25 度时的数值 (典型值为：1.43)

Avg_Slope = 温度与 Vsense 曲线的平均斜率 (单位：mv/℃ 或 uv/℃) (典型值：4.3mv/℃)。

利用以上公式，我们就可以方便的计算出当前温度传感器的温度了。

31.2 硬件设计

1. 例程功能

通过 ADC 的通道 16 读取 STM32F103 内部温度传感器的电压值，并将其转换为温度值，显示在 TFTLCD 屏上。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) ADC1 通道 16
- 5) 内部温度传感器

3. 原理图

ADC 和内部温度传感器都属于 STM32F103 内部资源，实际上我们只需要软件设置就可以

正常工作，我们需要用到 TFTLCD 模块显示结果。

31.3 程序设计

31.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍读取内部温度传感器 ADC 值的配置步骤。

读取 STM32 内部温度传感器 ADC 值的配置步骤

1) 开启 ADC 时钟

通过 `_HAL_RCC_ADC1_CLK_ENABLE` 函数开启 ADC1 的时钟。

2) 设置 ADC，开启内部温度传感器

调用 `HAL_ADC_Init` 函数来设置 ADC1 时钟分频系数、分辨率、模式、扫描方式等参数。

注意：该函数会调用 `HAL_ADC_MspInit` 回调函数来完成对 ADC 底层的初始化，包括：ADC1 时钟使能、ADC1 时钟源的选择等。

3) 配置 ADC 通道并启动 AD 转换器

调用 `HAL_ADC_ConfigChannel()` 函数配置 ADC1 通道 16，根据需求设置通道、规则序列、采样时间等。然后通过 `HAL_ADC_Start` 函数启动 AD 转换器。

4) 读取 ADC 值，计算温度。

这里选择查询方式读取，在读取 ADC 值之前需要调用 `HAL_ADC_PollForConversion` 等待上一次转换结束。然后就可以通过 `HAL_ADC_GetValue` 来读取 ADC 值。最后根据上面介绍的公式计算出温度传感器的温度值。

31.3.2 程序流程图

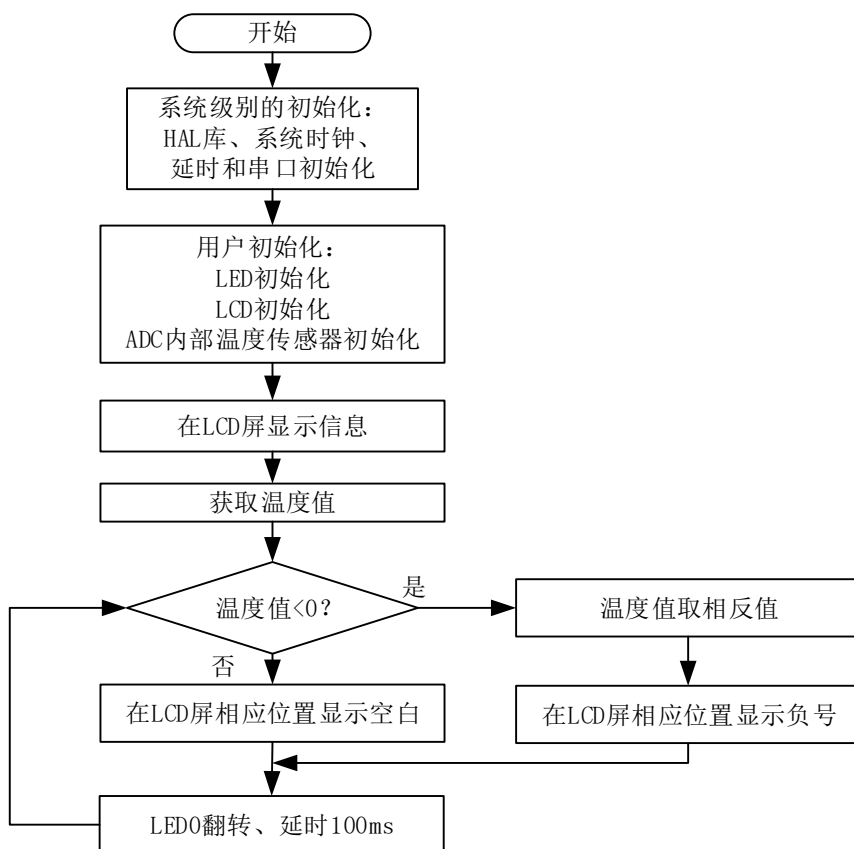


图 31.3.2.1 内部温度传感器实验程序流程图

31.3.3 程序解析

1. adc 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。内部温度传感器驱动源码包括两个文件：adc.c 和 adc.h。本实验和 ADC 章节实验用的都是同一个 adc.c 和 adc.h 代码。

在 adc.h 头文件再加入温度传感器的相关宏定义，该宏定义如下：

```
/* ADC 温度传感器通道 定义 */
#define ADC_TEMPSENSOR_CHX ADC_CHANNEL_16
```

ADC_CHANNEL_16 就是 ADC 通道 16 连接内部温度传感器的通道 16 宏定义。我们在定义为 ADC_TEMPSENSOR_CHX，可以让大家更容易理解这个宏定义的含义。

下面我们直接介绍与内部温度传感器相关的 adc.c 的程序，首先是 ADC 内部温度传感器初始化函数，其定义如下：

```
/**
 * @brief    ADC 内部温度传感器 初始化函数
 * @note     本函数还是使用 adc_init 对 ADC 进行大部分配置,有差异的地方再单独配置
 *           注意：STM32F103 内部温度传感器只连接在 ADC1 的通道 16 上，其他 ADC 无法进行转换.
 *
 * @param    无
 * @retval    无
 */
void adc_temperature_init(void)
{
    adc_init(); /* 先初始化 ADC */
    /* TSVREFE = 1, 启用内部温度传感器和 Vrefint */
    SET_BIT(g_adc_handle.Instance->CR2, ADC_CR2_TSVREFE);
}
```

该函数调用 adc_init 函数配置了 ADC 的基础功能参数，由于前面实验中的 adc_init 实验是对 ADC_CHANNEL_1 进行配置的，而我们对内部温度传感器的初始化步骤与普通 ADC 类似，为了不重复编写代码，我们用位操作函数进行修改 ADC 通道，把 ADC_CR2 的 TSVREFE 位置 1，即 SET_BIT(g_adc_handle.Instance->CR2, ADC_CR2_TSVREFE)；这样子就可以完成对内部温度传感器通道的初始化工作。adc_init 的实现代码，可以回顾以下 ADC 章节内容。

下面讲解一下获取内部温度传感器温度值函数，其定义如下：

```
/**
 * @brief    获取内部温度传感器温度值
 * @param    无
 * @retval    温度值(扩大了 100 倍,单位:℃.)
 */
short adc_get_temperature(void)
{
    uint32_t adcx;
    short result;
    double temperature;

    /* 读取内部温度传感器通道,10 次取平均 */
    adcx = adc_get_result_average(ADC_TEMPSENSOR_CHX, 20);
    temperature = (float)adcx * (3.3 / 4096); /* 转化为电压值 */
    temperature = (1.43 - temperature) / 0.0043 + 25; /* 计算温度 */
    result = temperature * 100; /* 扩大 100 倍. */
    return result;
}
```

该函数先是调用前面 ADC 实验章节写好的 adc_get_result_average 函数取获取通道 ch 的转换值，然后通过温度转换公式，返回温度值。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    short temp;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    adc_temperature_init(); /* 初始化 ADC 内部温度传感器采集 */
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "Temperature TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 120, 200, 16, 16, "TEMPERATE: 00.00C", BLUE);
    while (1)
    {
        temp = adc_get_temperature(); /* 得到温度值 */
        if (temp < 0)
        {
            temp = -temp;
            lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, "-", BLUE); /* 显示负号 */
        }
        else
        {
            lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, " ", BLUE); /* 无符号 */
        }
        lcd_show_xnum(30 + 11 * 8, 120, temp/100, 2, 16, 0, BLUE); /* 显示整数部分 */
        lcd_show_xnum(30 + 14 * 8, 120, temp%100, 2, 16, 0x80, BLUE); /* 显示小数部分 */
        LED0_TOGGLE(); /* LED0 闪烁, 提示程序运行 */
        delay_ms(250);
    }
}
```

该部分的代码逻辑很简单，先是得到温度值，再根据温度值判断正负值，来显示温度符号，再显示整数和小数部分。

31.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 31.4.1 所示：



图 31.4.1 内部温度传感器实验测试图

大家可以看看你的温度值与实际是否相符合（因为芯片会发热，所以一般会比实际温度偏高）？

第三十二章 光敏传感器实验

本章，我们将学习使用 STM32 开发板板载的一个光敏传感器。我们还是要使用到 ADC 采集，通过 ADC 采集电压，获取光敏传感器的电阻变化，从而得出环境光线的变化，并在 TFTLCD 上面显示出来。

本章分为如下几个小节：

32.1 光敏传感器简介

32.2 硬件设计

32.3 程序设计

32.4 下载验证

32.1 光敏传感器简介

光敏传感器是最常见的传感器之一，它的种类繁多，主要有：光电管、光电倍增管、光敏电阻、光敏三极管、太阳能电池、红外线传感器、紫外线传感器、光纤式光电传感器、色彩传感器、CCD 和 CMOS 图像传感器等。光传感器是目前产量最多、应用最广的传感器之一，它在自动控制和非电量电测技术中占有非常重要的地位。

光敏传感器是利用光敏元件将光信号转换为电信号的传感器，它的敏感波长在可见光波长附近，包括红外线波长和紫外线波长。光传感器不只局限于对光的探测，它还可以作为探测元件组成其他传感器，对许多非电量进行检测，只要将这些非电量转换为光信号的变化即可。

STM32F103 战舰开发板板载了一个光敏二极管（光敏电阻），作为光敏传感器，它对光的变化非常敏感。光敏二极管也叫光电二极管。光敏二极管与半导体二极管在结构上是类似的，其管芯是一个具有光敏特征的 PN 结，具有单向导电性，因此工作时需加上反向电压。无光照时，有很小的饱和反向漏电流，即暗电流，此时光敏二极管截止。当受到光照时，饱和反向漏电流大大增加，形成光电流，它随入射光强度的变化而变化。当光线照射 PN 结时，可以使 PN 结中产生电子-空穴对，使少数载流子的密度增加。这些载流子在反向电压下漂移，使反向电流增加。因此可以利用光照强弱来改变电路中的电流。

利用这个电流变化，我们串接一个电阻，就可以转换成电压的变化，从而通过 ADC 读取电压值，判断外部光线的强弱。

本章，我们利用 ADC3 的通道 6（PF8）来读取光敏二极管电压的变化，从而得到环境光线的变化，并将得到的光线强度，显示在 TFTLCD 上面。关于 ADC 的介绍，前面已经有详细介绍了，这里我们就不再细说了。

32.2 硬件设计

1. 例程功能

通过 ADC3 的通道 6（PF8）读取光敏传感器（LS1）的电压值，并转换为 0~100 的光线强度值，显示在 LCD 模块上面。光线越亮，值越大；光线越暗，值越小。大家可以用手指遮挡 LS1 和用手电筒照射 LS1，来查看光强变化。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1) LED 灯

LED : LED0 - PB5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) ADC3 : 通道 6 - PF8

5) 光敏传感器

3. 原理图

我们主要来看看光敏传感器和开发板的连接，如下图所示：

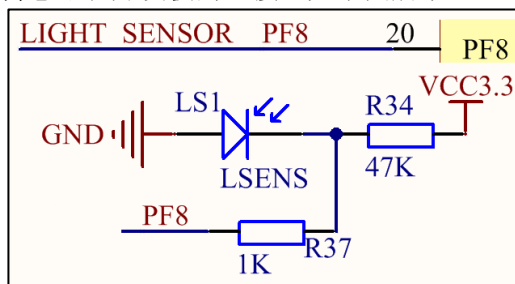


图 32.2.1 光敏传感器与开发板连接示意图

图中，LS1 是光敏二极管，外观看起来像与贴片 LED 类似（战舰位于 OLED 插座旁边，LS1），R34 为其提供反向电压，当环境光线变化时，LS1 两端的电压也会随之改变，通过 ADC3_IN6 通道读取 LIGHT_SENSOR（PF8）上面的电压，即可得到环境光线的强弱。光线越强，电压越低，光线越暗，电压越高。

32.3 程序设计

32.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍读取光敏传感器 ADC 值的配置步骤。

读取光敏传感器 ADC 值配置步骤

1) 开启 ADCx 和 ADC 通道对应的 IO 时钟，并配置该 IO 为模拟功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 ADC3 通道 6，对应 IO 是 PF8，它们的时钟开启方法如下：

```
HAL_RCC_ADC3_CLK_ENABLE(); /* 使能 ADC3 时钟 */
HAL_RCC_GPIOF_CLK_ENABLE(); /* 开启 GPIOF 时钟 */
```

2) 设置 ADC3，开启内部温度传感器

调用 HAL_ADC_Init 函数来设置 ADC3 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来完成对 ADC 底层的初始化，包括：ADC3 时钟使能、ADC3 时钟源的选择等。

3) 配置 ADC 通道并启动 AD 转换器

调用 HAL_ADC_ConfigChannel() 函数配置 ADC3 通道 6，根据需求设置通道、序列、采样时间和校准配置单端输入模式或差分输入模式等。然后通过 HAL_ADC_Start 函数启动 AD 转换器。

4) 读取 ADC 值，转换为光线强度值

这里选择查询方式读取，在读取 ADC 值之前需要调用 HAL_ADC_PollForConversion 等待上一次转换结束。然后就可以通过 HAL_ADC_GetValue 来读取 ADC 值。最后把得到的 ADC 值转换为 0~100 的光线强度值。

32.3.2 程序流程图

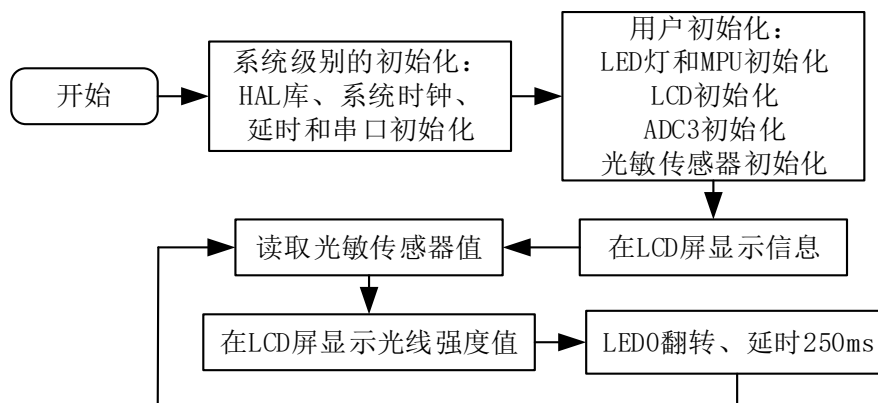


图 32.3.2.1 光敏传感器实验程序流程图

32.3.3 程序解析

1. lsens 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。LSENS 驱动源码包括两个文件：lsens.c 和 lsens.h。本实验还要用到 adc3.c 和 adc3.h 文件的驱动代码。adc3.c\h 文件的代码和单通道 ADC 采集实验的 adc.c\h 文件的代码几乎一样，这里就不再赘述了。

lsens.h 头文件定义了一些宏定义和一些函数的声明，该宏定义如下：

```

/* 光敏传感器对应 ADC3 的输入引脚和通道 定义 */
#define LSENS_ADC3_CHX_GPIO_PORT      GPIOF
#define LSENS_ADC3_CHX_GPIO_PIN        GPIO_PIN_8
#define LSENS_ADC3_CHX_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOF_CLK_ENABLE(); \
                                              }while(0) /* PF 口时钟使能 */

#define LSENS_ADC3_CHX                  ADC_CHANNEL_6 /* 通道 Y, 0 <= Y <= 17 */
    
```

这些宏定义分别是 PF8 及其时钟使能的宏定义，还有 ADC3 通道 6 的宏定义。

下面介绍 lsens.c 的函数，首先是光敏传感器初始化函数，其定义如下：

```

/**
 * @brief      初始化光敏传感器
 * @param      无
 * @retval     无
 */
void lsens_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    LSENS_ADC3_CHX_GPIO_CLK_ENABLE(); /* IO 口时钟使能 */
    /* 设置 AD 采集通道对应 IO 引脚工作模式 */
    gpio_init_struct.Pin = LSENS_ADC3_CHX_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(LSENS_ADC3_CHX_GPIO_PORT, &gpio_init_struct);
    adc3_init(); /* 初始化 ADC */
}
    
```

该函数初始化 PF8 为模拟功能，然后通过 adc3_init 函数初始化 ADC3。

最后是读取光敏传感器值，函数定义如下：

```

/**
 * @brief      读取光敏传感器值
 * @param      无
 * @retval     0~100:0, 最暗;100, 最亮
 */
uint8_t lsens_get_val(void)
{
    uint32_t temp_val = 0;
    
```

```
temp_val = adc3_get_result_average(LSENS_ADC3_CHX, 10); /* 读取平均值 */
temp_val /= 40;
if (temp_val > 100) temp_val = 100;
return (uint8_t)(100 - temp_val);
}
```

lsens_get_val 函数用于获取当前光照强度，该函数通过 adc3_get_result_average 函数得到通道 6 转换的电压值，经过简单量化后，处理成 0~100 的光强值。0 对应最暗，100 对应最亮。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    short adcx;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    lsens_init(); /* 初始化光敏传感器 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "LSENS TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "LSENS_VAL:", BLUE);

    while (1)
    {
        adcx = lsens_get_val();
        lcd_show_xnum(30 + 10 * 8, 110, adcx, 3, 16, 0, BLUE); /* 显示光线强度值 */
        LED0_TOGGLE(); /* LED0 闪烁, 提示程序运行 */
        delay_ms(250);
    }
}
```

该部分的代码逻辑很简单，初始化各个外设之后，进入死循环，通过 lsens_get_val 获取光敏传感器得到的光强值（0~100），并显示在 TFTLCD 上面。

32.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 32.4.1 所示：



图 32.4.1 光敏传感器实验测试图

我们可以通过给 LS1 不同的光照强度，来观察 LSENS_VAL 值的变化，光照越强，该值越大，光照越弱，该值越小，LSENS_VAL 值的范围是 0~100。

第三十三章 DAC 实验

本章，我们将介绍 STM32F103 的 DAC (Digital-to-analog converters, 数模转换器) 功能。我们通过三个实验来学习 DAC，分别是 DAC 输出实验、DAC 输出三角波实验和 DAC 输出正弦波实验。

本章分为如下几个小节：

33.1 DAC 简介

33.2 DAC 输出实验

33.3 DAC 输出三角波实验

33.4 DAC 输出正弦波实验

33.1 DAC 简介

STM32F103 的 DAC 模块 (数字/模拟转换模块) 是 12 位数字输入，电压输出型的 DAC。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 V_{REF+} 以获得更精确的转换结果。

STM32 的 DAC 模块主要特点有：

- ① 2 个 DAC 转换器：每个转换器对应 1 个输出通道
- ② 8 位或者 12 位单调输出
- ③ 12 位模式下数据左对齐或者右对齐
- ④ 同步更新功能
- ⑤ 噪声\三角波形生成
- ⑥ 双 DAC 双通道同时或者分别转换
- ⑦ 每个通道都有 DMA 功能

DAC 通道框图如图 33.1.1 所示：

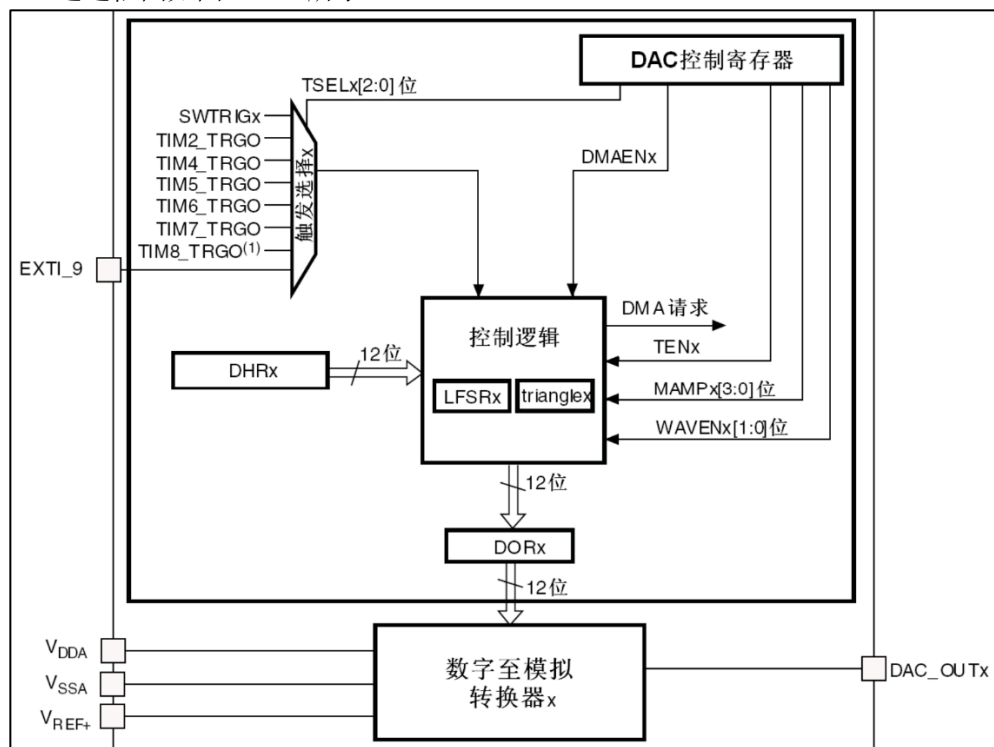


图 33.1.1 DAC 通道框图

图中 VDDA 和 VSSA 为 DAC 模块模拟部分的供电,而 VREF+则是 DAC 模块的参考电压。DAC_OUTx 就是 DAC 的两个输出通道了(对应 PA4 或者 PA5 引脚)。ADC 的这些输入/输出引脚信息如下表所示:

| 引脚名称 | 信号类型 | 说明 |
|----------|-----------|---|
| VREF+ | 正模拟参考电压输入 | DAC 高/正参考电压, $V_{REF+} \leq V_{DDA}$ (3.3V) |
| VDDA | 模拟电源输入 | 模拟电源 |
| VSSA | 模拟电源地输入 | 模拟电源地 |
| DAC_OUTx | 模拟输出信号 | DAC 通道 x 模拟输出, x=1、2 |

表 33.1.1 DAC 输入/输出引脚

从图 33.1.1 可以看出, DAC 输出是受 DORx (x=1/2, 下同) 寄存器直接控制的,但是我们不能直接往 DORx 寄存器写入数据,而是通过 DHRx 间接的传给 DORx 寄存器,实现对 DAC 输出的控制。

前面我们提到,STM32F103 的 DAC 支持 8/12 位模式,8 位模式的时候是固定的右对齐的,而 12 位模式又可以设置左对齐/右对齐。DAC 单通道模式下的数据寄存器对齐方式,总共有 3 种情况,如下图所示:

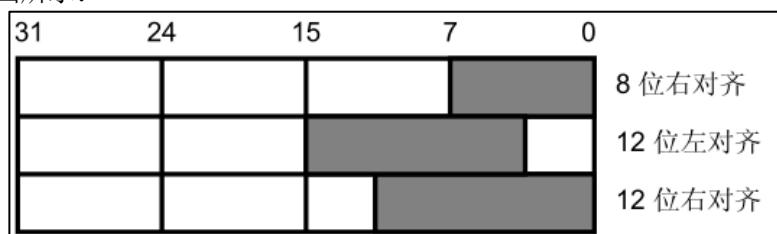


图 33.1.2 DAC 单通道模式下的数据寄存器对齐方式

- ① 8 位数据右对齐: 用户将数据写入 DAC_DHR8Rx[7:0]位 (实际存入 DHRx[11:4]位)。
- ② 12 位数据左对齐: 用户将数据写入 DAC_DHR12Lx[15:4]位 (实际存入 DHRx[11:0]位)。
- ③ 12 位数据右对齐: 用户将数据写入 DAC_DHR12Rx[11:0]位 (实际存入 DHRx[11:0]位)。

我们本章实验中使用的都是单通道模式下的 DAC 通道 1, 采用 12 位右对齐格式, 所以采用第③种情况。另外 DAC 还具有双通道转换功能。

对于 DAC 双通道 (可用时), 也有三种可能的方式, 如下图所示:

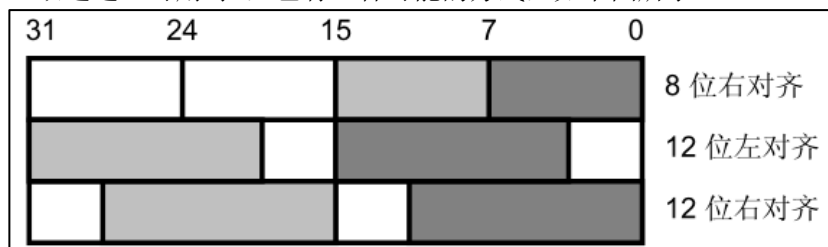


图 33.1.3 DAC 双通道模式下的数据寄存器对齐方式

- ① 8 位数据右对齐: 用户将 DAC 通道 1 的数据写入 DAC_DHR8RD[7:0]位 (实际存入 DHR1[11:4]位), 将 DAC 通道 2 的数据写入 DAC_DHR8RD[15:8]位 (实际存入 DHR2[11:4]位)。
- ② 12 位数据左对齐: 用户将 DAC 通道 1 的数据写入 DAC_DHR12LD[15:4]位 (实际存入 DHR1[11:0]位), 将 DAC 通道 2 的数据写入 DAC_DHR12LD[31:20]位 (实际存入 DHR2[11:0]位)。
- ③ 12 位数据右对齐: 用户将 DAC 通道 1 的数据写入 DAC_DHR12RD[11:0]位 (实际存入 DHR1[11:0]位), 将 DAC 通道 2 的数据写入 DAC_DHR12RD[27:16]位 (实际存入 DHR2[11:0]位)。

DAC 可以通过软件或者硬件触发转换, 通过配置 TENx 控制位来决定。

如果没有选中硬件触发 (寄存器 DAC_CR 的 TENx 位置 0), 存入寄存器 DAC_DHRx 的数据会在 1 个 APB1 时钟周期后自动传至寄存器 DAC_DORx。如果选中硬件触发 (寄存器 DAC_CR 的 TENx 位置 1), 数据传输在触发发生以后 3 个 APB1 时钟周期后完成。一旦数据从 DAC_DHRx

寄存器装入 DAC_DORx 寄存器，在经过时间 t_{SETTLING} 之后，输出即有效，这段时间的长短依电源电压和模拟输出负载的不同会有所变化。我们可以从《STM32F103ZET6.pdf》数据手册查到 t_{SETTLING} 的典型值为 3 μs ，最大是 4 μs ，所以 DAC 的转换速度最快是 333K 左右。

不使用硬件触发（TEN=0），其转换的时间框图如图 33.1.4 所示：

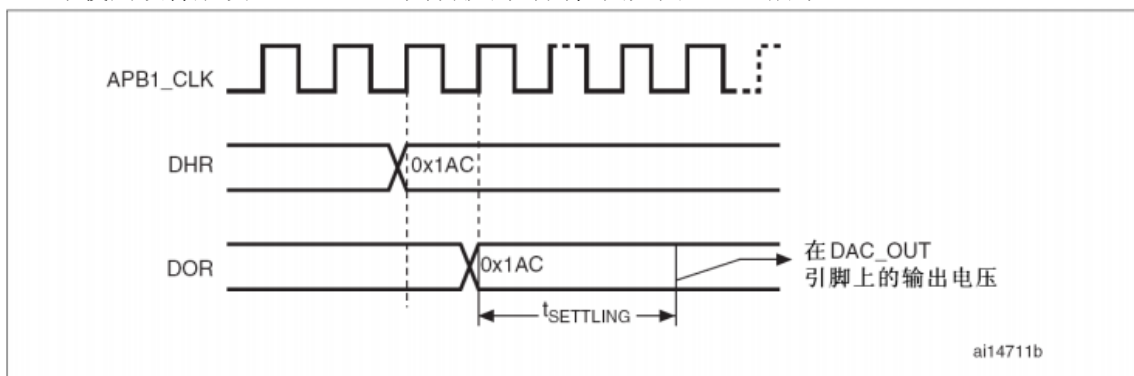


图 33.1.4 TEN=0 时 DAC 模块转换时间框图

当 DAC 的参考电压为 $V_{\text{ref+}}$ 的时候，DAC 的输出电压是线性的从 0~ $V_{\text{ref+}}$ ，12 位模式下 DAC 输出电压与 $V_{\text{ref+}}$ 以及 DORx 的计算公式如下：

$$\text{DACx 输出电压} = V_{\text{ref}} * (\text{DORx} / 4096)$$

如果使用硬件触发（TENx=1），可通过外部事件（定时计数器、外部中断线）触发 DAC 转换。由 TSELx[2:0] 控制位来决定选择 8 个触发事件中的一个来触发转换。触发事件如下表所示：

| 触发源 | 类型 | TSELx[2:0] |
|---|--------------|------------|
| 定时器6 TRGO事件 | 来自片上定时器的内部信号 | 000 |
| 互联型产品为定时器3 TRGO事件 或大容量产品为定时器8 TRGO事件 | | 001 |
| 定时器7 TRGO事件 | | 010 |
| 定时器5 TRGO事件 | | 011 |
| 定时器2 TRGO事件 | | 100 |
| 定时器4 TRGO事件 | | 101 |
| EXTI线路9 | 外部引脚 | 110 |
| SWTRIG(软件触发) | 软件控制位 | 111 |

表 33.1.3 DAC 触发选择

原表见《STM32F10xxx 参考手册_V10（中文版）.pdf》第 185 页表 71。

每个 DAC 通道都有 DMA 功能，两个 DMA 通道分别用于处理两个 DAC 通道的 DMA 请求。如果 DMAENx 位置 1 时，如果发生外部触发（而不是软件触发），就会产生一个 DMA 请求，然后 DAC_DHRx 寄存器的数据被转移到 DAC_NORx 寄存器。

33.2 DAC 输出实验

本实验我们来学习 DAC 输出实验。

33.2.1 DAC 寄存器

下面，我们介绍要实现 DAC 的通道 1 输出，需要用到的一些 DAC 寄存器。

- **DAC 控制寄存器（DAC_CR）**

DAC 控制寄存器描述如图 33.2.1.1 所示：

| | | | | | | | | | | | | | | | |
|------|----|----|--|------------|----|----|----|------------|----|------------|----|----|------|-------|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | DMAEN2 | MAMP2[3:0] | | | | WAVE2[2:0] | | TSEL2[2:0] | | | TEN2 | BOFF2 | EN2 |
| | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | DMAEN1 | MAMP13:0] | | | | WAVE1[2:0] | | TSEL1[2:0] | | | TEN1 | BOFF1 | EN1 |
| | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位7:6 | | | WAVE1[1:0]: DAC通道1噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable) 该2位由软件设置和清除。 00: 关闭波形生成; 10: 使能噪声波形发生器; 1x: 使能三角波发生器。 | | | | | | | | | | | | |
| 位5:3 | | | TSEL1[2:0]: DAC通道1触发选择 (DAC channel1 trigger selection) 该位用于选择DAC通道1的外部触发事件。 000: TIM6 TRGO事件; 001: 对于互联型产品是TIM3 TRGO事件, 对于大容量产品是TIM8 TRGO事件; 010: TIM7 TRGO事件; 011: TIM5 TRGO事件; 100: TIM2 TRGO事件; 101: TIM4 TRGO事件; 110: 外部中断线9; 111: 软件触发。 注意: 该位只能在TEN1= 1(DAC通道1触发使能)时设置。 | | | | | | | | | | | | |
| 位2 | | | TEN1: DAC通道1触发使能 (DAC channel1 trigger enable) 该位由软件设置和清除, 用来使能/关闭DAC通道1的触发。 0: 关闭DAC通道1触发, 写入寄存器DAC_DHRx的数据在1个APB1时钟周期后传入寄存器DAC_DOR1; 1: 使能DAC通道1触发, 写入寄存器DAC_DHRx的数据在3个APB1时钟周期后传入寄存器DAC_DOR1。 注意: 如果选择软件触发, 写入寄存器DAC_DHRx的数据只需要1个APB1时钟周期就可以传入寄存器DAC_DOR1。 | | | | | | | | | | | | |
| 位1 | | | BOFF1: 关闭DAC通道1输出缓存 (DAC channel1 output buffer disable) 该位由软件设置和清除, 用来使能/关闭DAC通道1的输出缓存。 0: 使能DAC通道1输出缓存; 1: 关闭DAC通道1输出缓存。 | | | | | | | | | | | | |
| 位0 | | | EN1: DAC通道1使能 (DAC channel1 enable) 该位由软件设置和清除, 用来使能/失能DAC通道1。 0: 关闭DAC通道1; 1: 使能DAC通道1。 | | | | | | | | | | | | |

图 33.2.1.1 DACx_CR 寄存器

DAC_CR 的低 16 位用于控制通道 1, 高 16 位用于控制通道 2, 下面介绍本实验需要设置的一些位:

EN1 位用于 DAC 通道 1 的使能, 我们需要用到 DAC 通道 1 的输出, 该位必须设置为 1。

BOFF1 位用于 DAC 输出缓存控制, 这里 STM32 的 DAC 输出缓存做的有些不好, 如果使能的话, 虽然输出能力强一点, 但是输出没法到 0, 这是个很严重的问题。所以本章的三个实验我们都不使用输出缓存, 即该位设置为 1。

TEN1 位用于 DAC 通道 1 的触发使能, 我们设置该位为 0, 不使用触发。写入 DHR1 的值会在 1 个 APB1 周期后传送到 DOR1, 然后输出到 PA4 口上。

TSEL1[2:0]位用于选择 DAC 通道 1 的触发方式, 这里我们没有用到外部触发, 所以这几位设置为 0 即可。

WAVE1[1:0]位用于控制 DAC 通道 1 的噪声/波形输出功能, 我们这里没用到波形发生器, 所以默认设置为 00, 不使能噪声/波形输出。

MAMP[3:0]位是屏蔽/幅值选择器, 用来在噪声生成模式下选择屏蔽位, 在三角波生成模式下选择波形的幅值。本实验没有用到波形发生器, 所以设置为 0 即可。

DMAEN1 位用于 DAC 通道 1 的 DMA 使能，本实验没有用到 DMA 功能，所以设置为 0。

● DAC 通道 1 12 位右对齐数据保持寄存器 (DAC_DHR12R1)

DAC 通道 1 的 12 位右对齐数据保持寄存器描述如图 33.2.1.3 所示：

| | | | | | | | | | | | | | | | |
|-------|----|----|----|---|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | DACC1DHR[11:0] | | | | | | | | | | | |
| | | | | rW rW rW rW rW rW rW rW rW rW rW rW | | | | | | | | | | | |
| 位11:0 | | | | DACC1DHR[11:0]: DAC通道1的12位右对齐数据 (DAC channel1 12-bit right-aligned data) 该位由软件写入，表示DAC通道1的12位数据。 | | | | | | | | | | | |

图 33.2.1.3 DAC0_DHR12R1 寄存器

该寄存器用来设置 DAC 输出，通过写入 12 位数据到该寄存器，就可以在 DAC 输出通道 1 (PA4) 得到我们所需要的结果。

33.2.2 硬件设计

1. 例程功能

使用 KEY1/KEY_UP 两个按键，控制 STM32 内部 DAC 的通道 1 输出电压大小，然后通过 ADC2 的通道 14 采集 DAC 输出的电压，在 LCD 模块上面显示 ADC 采集到的电压值以及 DAC 的设定输出电压值等信息。也可以通过 usmart 调用 dac_set_voltage 函数，来直接设置 DAC 输出电压。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯 LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) 独立按键：KEY1 - PE3、WK_UP - PA0
- 5) ADC3：通道 1 - PA1
- 6) DAC1：通道 1 - PA4

3. 原理图

我们来看看原理图上 ADC3 通道 1 (PA1) 和 DAC1 通道 1 (PA4) 引出来的引脚，如下图 所示：

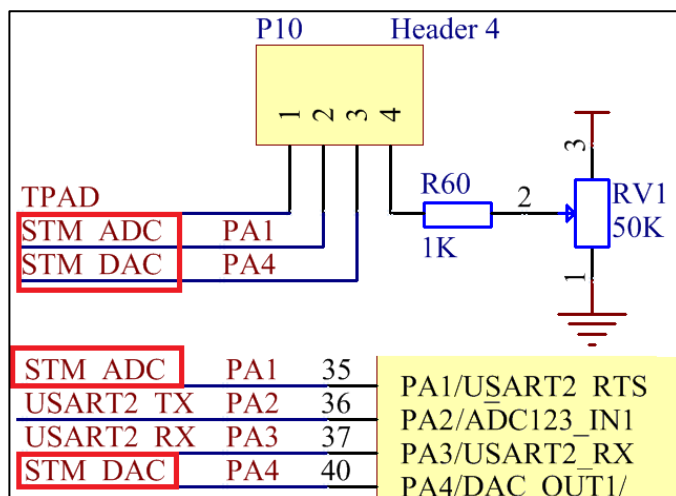


图 33.2.2.1 ADC 和 DAC 在开发板上的连接关系原理图

我们只需要通过跳线帽连接 ADC 和 DAC，就可以使得 ADC3 通道 1（PA1）和 DAC1 通道 1（PA4）连接起来。对应的硬件连接如图 33.2.2.2 所示：

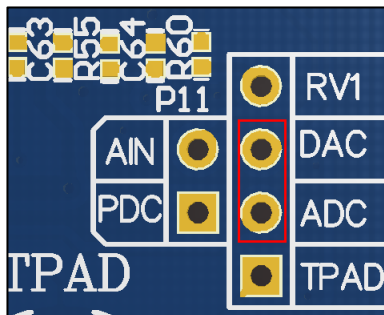


图 33.2.2.2 硬件连接示意图

33.2.3 程序设计

33.2.3.1 DAC 的 HAL 库驱动

DAC 在 HAL 库中的驱动代码在 stm32f1xx_hal_dac.c 和 stm32f1xx_hal_dac_ex.c 文件（及其头文件）中。

1. HAL_DAC_Init 函数

DAC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Init(DAC_HandleTypeDef *hdac);
```

- **函数描述：**

用于初始化 DAC。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
    DAC_TypeDef                *Instance;           /* DAC 寄存器基地址 */
    __IO HAL_DAC_StateTypeDef State;               /* DAC 工作状态 */
    HAL_LockTypeDef            Lock;               /* DAC 锁定对象 */
    DMA_HandleTypeDef          *DMA_Handle1;       /* 通道 1 的 DMA 处理句柄指针 */
    DMA_HandleTypeDef          *DMA_Handle2;       /* 通道 2 的 DMA 处理句柄指针 */
    __IO uint32_t               ErrorCode;          /* DAC 错误代码 */
} DAC_HandleTypeDef;
```

从该结构体看到该函数并没有设置任何 DAC 相关寄存器，即没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC，为后面 HAL 库操作 DAC 做好准备。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

DAC 的 MSP 初始化函数 HAL_DAC_MspInit，该函数声明如下：

```
void HAL_DAC_MspInit(DAC_HandleTypeDef* hdac);
```

2. HAL_DAC_ConfigChannel 函数

DAC 的通道参数初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(DAC_HandleTypeDef *hdac,
                                         DAC_ChannelConfTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**

该函数用来配置 DAC 通道的触发类型以及输出缓冲。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 是 DAC_ChannelConfTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
    uint32_t DAC_Trigger;           /* DAC 触发源的选择 */
    // ... 其他成员 ...
}
```

```
uint32_t DAC_OutputBuffer; /* 启用或者禁用 DAC 通道输出缓冲区 */
} DAC_ChannelConfTypeDef;
```

形参 2 用于选择要配置的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

- 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

3. HAL_DAC_Start 函数

使能启动 DAC 转换通道函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

- 函数描述：

使能启动 DAC 转换通道。

- 函数形参：

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 用于选择要启动的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

- 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

4. HAL_DAC_SetValue 函数

DAC 的通道输出值函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef *hdac, uint32_t Channel,
                                     uint32_t Alignment, uint32_t Data);
```

- 函数描述：

配置 DAC 的通道输出值。

- 函数形参：

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 用于选择要输出的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

形参 3 用于指定数据对齐方式。

形参 4 设置要加载到选定数据保存寄存器中的数据。

- 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

5. HAL_DAC_GetValue 函数

DAC 读取通道输出值函数，其声明如下：

```
uint32_t HAL_DAC_GetValue(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

- 函数描述：

获取所选 DAC 通道的最后一个数据输出值。

- 函数形参：

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 用于选择要读取的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

- 函数返回值：

获取到的输出值。

DAC 输出配置步骤

1) 开启 DACx 和 DAC 通道对应的 IO 时钟，并配置该 IO 为模拟功能

首先开启 DACx 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1，对应 IO 是 PA4，它们的时钟开启方法如下：

```
HAL_RCC_DAC_CLK_ENABLE (); /* 使能 DAC1 时钟 */
HAL_RCC_GPIOA_CLK_ENABLE (); /* 开启 GPIOA 时钟 */
```

2) 初始化 DACx

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC。

注意：该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

3) 配置 DAC 通道并启动 DA 转换器

在 HAL 库中，通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道，根据需求设置触发类型以及输出缓冲。

配置好 DAC 通道之后，通过 HAL_DAC_Start 函数启动 DA 转换器。

4) 设置 DAC 的输出值

通过 HAL_DAC_SetValue 函数设置 DAC 的输出值。

33.2.3.2 程序流程图

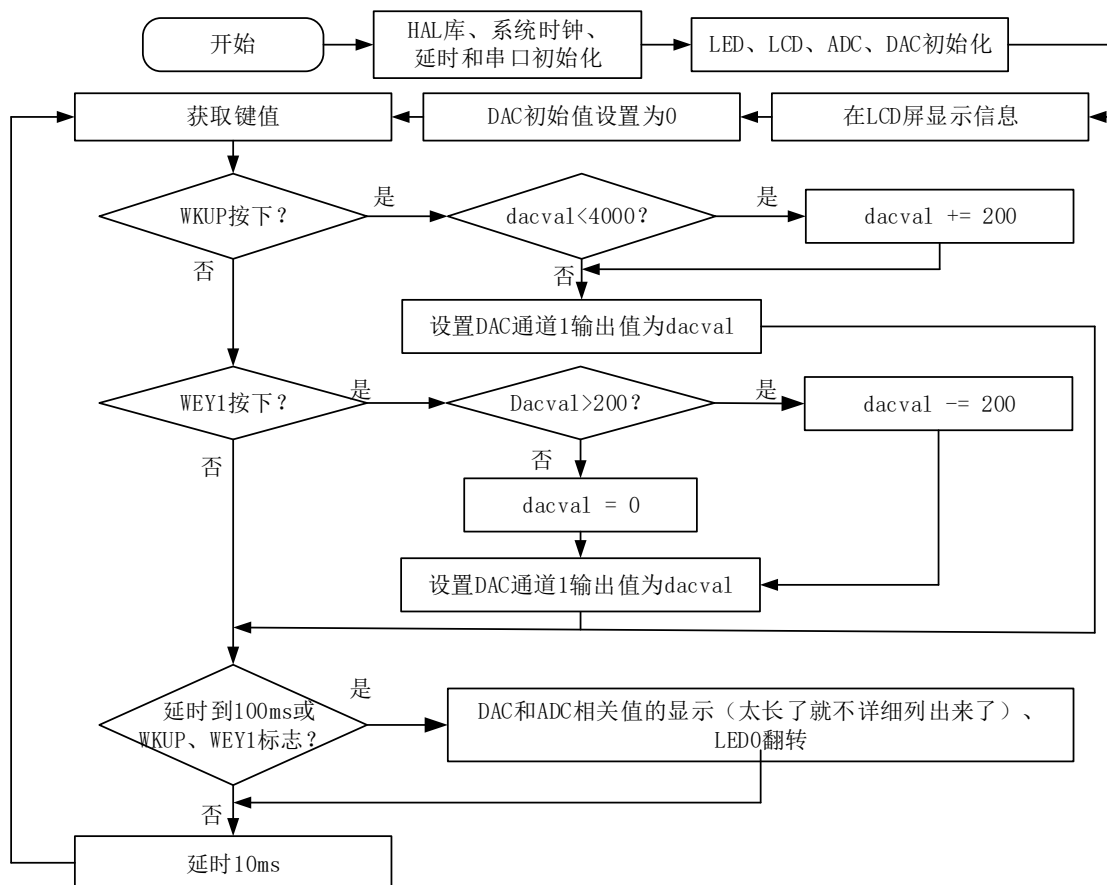


图 33.2.3.2.1 DAC 输出实验程序流程图

33.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。本实验有 3 个实验，每一个实验的代码都是在上一个实验后面追加。

dac.h 文件只有一些声明，下面直接开始介绍 dac.c 的程序，首先是 DAC 初始化函数。

```

/**
 * @brief      DAC 初始化函数
 * @note      本函数支持 DAC1_OUT1/2 通道初始化
 *            DAC 的输入时钟来自 APB1，时钟频率=36Mhz=27.8ns
 *            DAC 在输出 buffer 关闭的时候，输出建立时间：tSETTLING = 4us
 *            因此 DAC 输出的最高速度约为：250Khz，以 10 个点为一个周期，最大能输出 25Khz 左右的波形
 * @param      outx: 要初始化的通道. 1,通道 1; 2,通道 2
 * @retval     无
 */
void dac_init(uint8_t outx)
{
    GPIO_InitTypeDef gpio_init_struct;
    DAC_ChannelConfTypeDef dac_ch_conf;

```



```

__HAL_RCC_DAC_CLK_ENABLE(); /* 使能 DAC1 的时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE(); /* 使能 DAC OUT1/2 的 IO 口时钟 (都在 PA 口, PA4/PA5) */

/* STM32 单片机, 总是 PA4=DAC1_OUT1, PA5=DAC1_OUT2 */
gpio_init_struct.Pin = (outx==1)? GPIO_PIN_4 : GPIO_PIN_5;
gpio_init_struct.Mode = GPIO_MODE_ANALOG;
HAL_GPIO_Init(GPIOA, &gpio_init_struct);

g_dac_handle.Instance = DAC;
HAL_DAC_Init(&g_dac_handle); /* 初始化 DAC */

dac_ch_conf.DAC_Trigger = DAC_TRIGGER_NONE; /* 不使用触发功能 */
dac_ch_conf.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE; /* DAC1 输出缓冲关闭 */

switch(outx)
{
    case 1:
        /* DAC 通道 1 配置 */
        HAL_DAC_ConfigChannel(&g_dac_handle, &dac_ch_conf, DAC_CHANNEL_1);
        HAL_DAC_Start(&g_dac_handle, DAC_CHANNEL_1); /* 开启 DAC 通道 1 */
        break;
    case 2:
        /* DAC 通道 2 配置 */
        HAL_DAC_ConfigChannel(&g_dac_handle, &dac_ch_conf, DAC_CHANNEL_2);
        HAL_DAC_Start(&g_dac_handle, DAC_CHANNEL_2); /* 开启 DAC 通道 1 */
        break;
    default: break;
}
}

```

该函数主要调用 HAL_DAC_Init 和 HAL_DAC_ConfigChannel 函数初始化 DAC, 并调用 HAL_DAC_Start 函数使能 DAC 通道。HAL_DAC_Init 函数会调用 HAL_DAC_MspInit 回调函数, 该函数用于存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。本实验为了让 dac_init 函数支持 DAC 的 OUT1/2 两个通道的初始化, 就没有用到该函数。

下面是设置 DAC 通道 1/2 输出电压函数, 其定义如下:

```

/**
 * @brief      设置通道 1/2 输出电压
 * @param      outx: 1, 通道 1; 2, 通道 2
 * @param      vol : 0~3300, 代表 0~3.3V
 * @retval     无
 */
void dac_set_voltage(uint8_t outx, uint16_t vol)
{
    double temp = vol;
    temp /= 1000;
    temp = temp * 4096 / 3.3;
    if (temp >= 4096) temp = 4095; /* 如果值大于等于 4096, 则取 4095 */

    if (outx == 1) /* 通道 1 */
    {
        /* 12 位右对齐数据格式设置 DAC 值 */
        HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R, temp);
    }
    else /* 通道 2 */
    {
        /* 12 位右对齐数据格式设置 DAC 值 */
        HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_2, DAC_ALIGN_12B_R, temp);
    }
}

```

该函数实际就是将电压值转换为 DAC 输入值, 形参 1 用于设置通道, 形参 2 设置要输出的电压值, 设置的范围: 0~3300, 代表 0~3.3V。

最后在 main 函数里面编写如下代码:

```
int main(void)
```

```

{
    uint16_t adcx;
    float temp;
    uint8_t t = 0;
    uint16_t dacval = 0;
    uint8_t key;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    adc2_init(); /* 初始化 ADC2 */
    dac_init(1); /* 初始化 DAC1_OUT1 通道 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DAC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "WK_UP:+ KEY1:-", RED);

    lcd_show_string(30, 130, 200, 16, 16, "DAC VAL:", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "DAC VOL:0.000V", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "ADC VOL:0.000V", BLUE);

    while (1)
    {
        t++;
        key = key_scan(0); /* 按键扫描 */

        if (key == WKUP_PRES)
        {
            if (dacval < 4000) dacval += 200;
            /* 输出增大 200 */
            HAL_DAC_SetValue(&g_dac1_handler, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dacval);
        }
        else if (key == KEY1_PRES)
        {
            if (dacval > 200) dacval -= 200;
            else dacval = 0;
            /* 输出减少 200 */
            HAL_DAC_SetValue(&g_dac1_handler, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dacval);
        }
        /* WKUP/KEY1 按下了,或者定时时间到了 */
        if (t == 10 || key == KEY1_PRES || key == WKUP_PRES)
        {
            /* 读取前面设置 DAC1_OUT1 的值 */
            adcx = HAL_DAC_GetValue(&g_dac1_handler, DAC_CHANNEL_1);
            lcd_show_xnum(94, 150, adcx, 4, 16, 0, BLUE); /* 显示 DAC 寄存器值 */

            temp = (float)adcx * (3.3 / 4096); /* 得到 DAC 电压值 */
            adcx = temp;
            lcd_show_xnum(94, 170, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */

            temp -= adcx;
            temp *= 1000;
            lcd_show_xnum(110, 170, temp, 3, 16, 0x80, BLUE); /* 显示电压值的小数部分 */

            adcx = adc2_get_result_average(ADC2_CHY, 20); /* 得到 ADC3 通道 1 的转换结果 */
            temp = (float)adcx * (3.3 / 4096); /* 得到 ADC 电压值 (adc 是 16bit 的) */
        }
    }
}

```

```

    adcx = temp;
    lcd_show_xnum(94, 190, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */

    temp -= adcx;
    temp *= 1000;
    lcd_show_xnum(110, 190, temp, 3, 16, 0X80, BLUE); /* 显示电压值的小数部分 */

    LED0_TOGGLE(); /* LED0 闪烁 */
    t = 0;
}
delay_ms(10);
}
}

```

此部分代码，我们通过 KEY_UP（WKUP 按键）和 KEY1（也就是上下键）来实现对 DAC 输出的幅值控制。按下 KEY_UP 增加，按 KEY1 减小。同时在 LCD 上面显示 DHR12R1 寄存器的值、DAC 设置输出电压以及 ADC 采集到的 DAC 输出电压。

33.2.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示如图 33.2.4.1 所示：

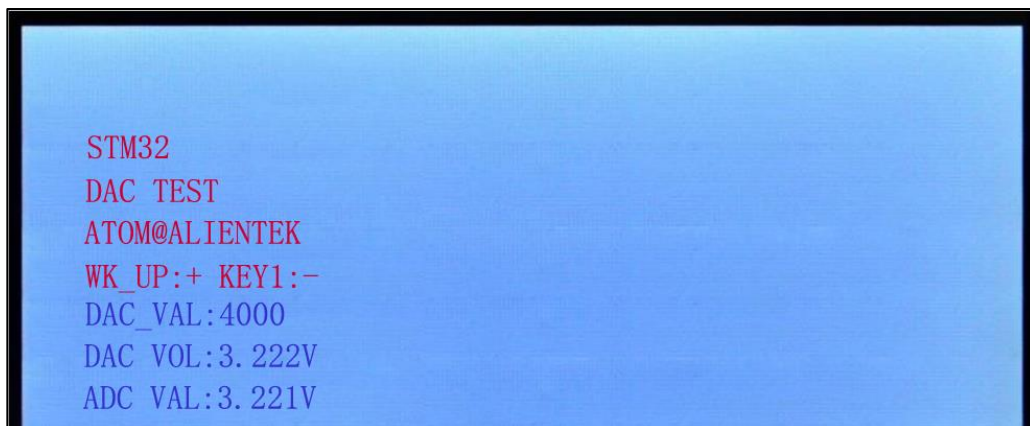


图 33.2.4.1 DAC 输出实验测试图

验证试验前，记得先通过跳线帽连接 ADC 和 DAC 排针，然后我们可以通过按 WK_UP 按键，增加 DAC 输出的电压，这时 ADC 采集到的电压也会增大，通过按 KEY1 减小 DAC 输出的电压，这时 ADC 采集到的电压也会减小。

除此之外，我们还可以通过 usmart 调用 dac_set_voltage 函数，来直接设置 DAC 输出电压，如下图 33.2.4.2 所示：

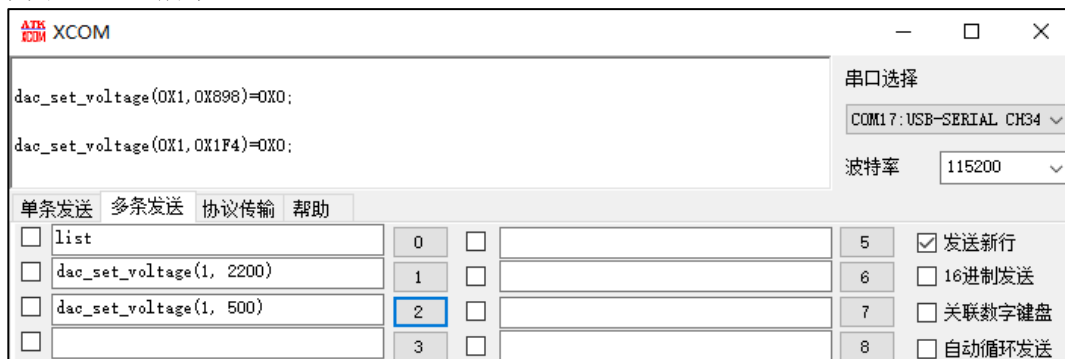


图 33.2.4.2 usmart 测试图

33.3 DAC 输出三角波实验

本实验我们来学习使用如何让 DAC 输出三角波，DAC 初始化部分还是用 DAC 输出实验的，所以做本实验的前提是先学习 DAC 输出实验。

33.3.1 DAC 寄存器

本实验用到的寄存器在 DAC 输出实验都有介绍。

33.3.2 硬件设计

1. 例程功能

使用 DAC 输出三角波，通过 KEY0/KEY1 两个按键，控制 DAC1 的通道 1 输出两种三角波，需要通过示波器接 PA4 进行观察。也可以通过 usmart 调用 dac_triangular_wave 函数，来控制输出哪种三角波。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) 独立按键：KEY0 - PE4、KEY1 - PE3
- 5) DAC1 : 通道 1 - PA4

3. 原理图

我们只需要把示波器的探头接到 DAC1 通道 1 (PA4) 引脚，就可以在示波器上显示 DAC 输出的波形。PA4 对应 P10 的 DAC 排针，如图 33.3.2.1 所示：

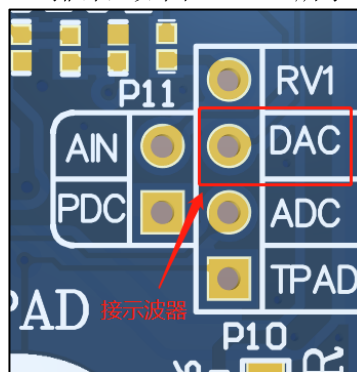


图 33.3.2.1 硬件连接示意图

33.3.3 程序设计

本实验用到的 DAC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍 DAC 输出三角波的配置步骤。

DAC 输出三角波配置步骤

1) 开启 DACx 和 DAC 通道对应的 IO 时钟，并配置该 IO 为模拟功能

首先开启 DACx 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1，对应 IO 是 PA4，它们的时钟开启方法如下：

```
HAL_RCC_DAC_CLK_ENABLE ();          /* 使能 DAC1 时钟 */
HAL_RCC_GPIOA_CLK_ENABLE ();        /* 开启 GPIOA 时钟 */
```

2) 初始化 DACx

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC。

注意：该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

3) 配置 DAC 通道并启动 DA 转换器

在 HAL 库中，通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道，根据需求设置触发类型以及输出缓冲。

配置好 DAC 通道之后，通过 HAL_DAC_Start 函数启动 DA 转换器。

4) 设置 DAC 的输出值

通过 HAL_DAC_SetValue 函数设置 DAC 的输出值。这里我们根据三角波的特性，创建了 dac_triangular_wave 函数用于控制输出三角波。

33.3.3.1 程序流程图

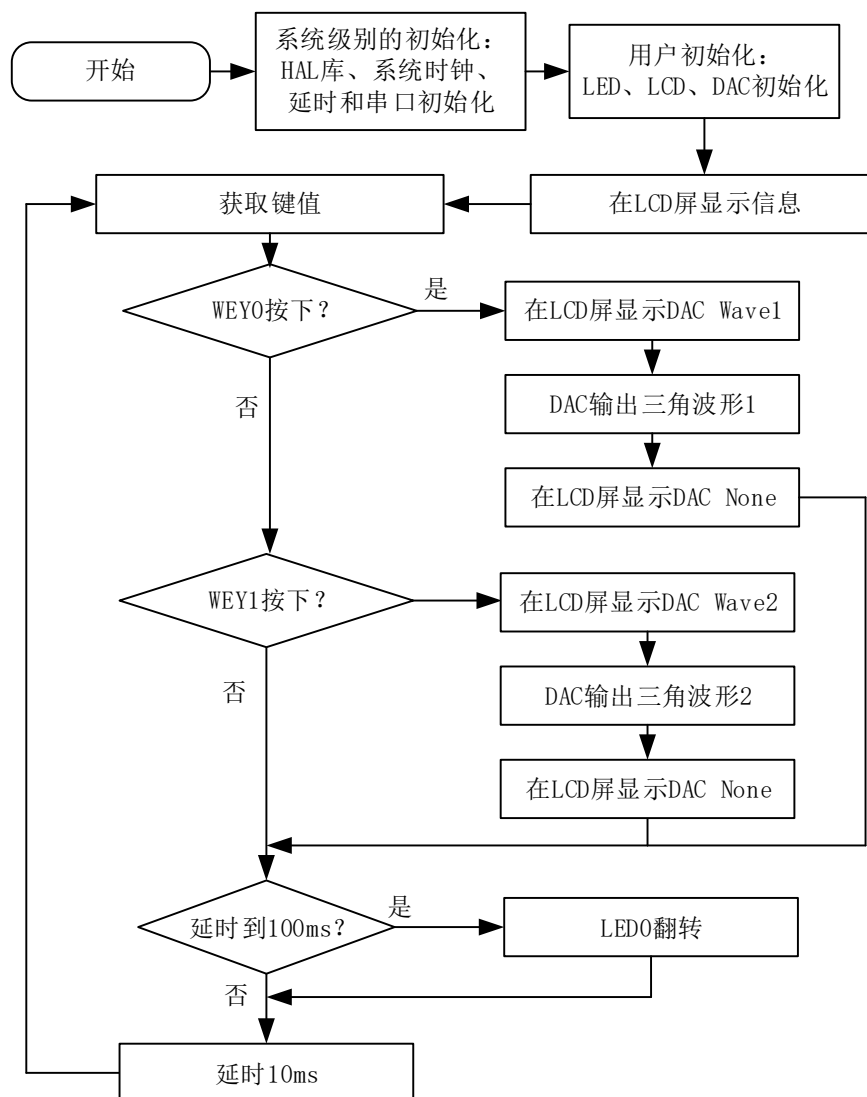


图 33.3.3.1.1 DAC 输出三角波实验程序流程图

33.3.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。

dac.h 文件只有一些声明，下面直接开始介绍 dac.c 的程序，本实验的 DAC 初始化我们还是用到 dac_init 函数，就添加了一个设置 DAC_OUT1 输出三角波函数，其定义如下：

```
/**
 * @brief      设置 DAC_OUT1 输出三角波
 * @note      输出频率  $\approx 1000 / (dt * samples)$  Khz, 不过在 dt 较小的时候, 比如小于 5us
 *            时, 由于 delay_us 本身就不准了(调用函数, 计算等都需要时间, 延时很小的时候, 这些
 *            时间会影响到延时), 频率会偏小.
 *
 * @param      maxval: 最大值(0< maxval<4096), (maxval + 1)必须大于等于 samples/2
 * @param      dt    : 每个采样点的延时时间(单位: us)
 * @param      samples: 采样点的个数, 必须小于等于 (maxval+1)*2, 且 maxval 不能等于 0
 * @param      n      : 输出波形个数, 0~65535
 *
 * @retval     无
 */
void dac_triangular_wave(uint16_t maxval, uint16_t dt, uint16_t samples,
                        uint16_t n)
{
    uint16_t i, j;
    float incval;          /* 递增量 */
    float Curval;          /* 当前值 */
    if((maxval + 1) <= samples) return; /* 数据不合法 */
    incval = (maxval + 1) / (samples / 2); /* 计算递增量 */
    for(j = 0; j < n; j++)
    {
        Curval = 0; /* 先输出 0 */
        HAL_DAC_SetValue(&dac1_handler, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Curval);
        for(i = 0; i < (samples / 2); i++) /* 输出上升沿 */
        {
            Curval += incval; /* 新的输出值 */
            HAL_DAC_SetValue(&dac1_handler, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Curval);
            delay_us(dt);
        }
        for(i = 0; i < (samples / 2); i++) /* 输出下降沿 */
        {
            Curval -= incval; /* 新的输出值 */
            HAL_DAC_SetValue(&dac1_handler, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Curval);
            delay_us(dt);
        }
    }
}
```

该函数用于设置 DAC 通道 1 输出三角波，输出频率 $\approx 1000 / (dt * samples)$ Khz，形参含义在源码已经有详细注释。该函数中，我们使用 HAL_DAC_SetValue 函数来设置 DAC 的输出值，这样得到的三角波在示波器上可以看到。如果有跳动现象（不平稳），是正常的，因为调用函数，计算等都需要时间，这样就会导致输出的波形是不太稳定的。越高性能的 MCU，得到的波形会越稳定。而且用 HAL 库函数操作效率没有直接操作寄存器高，所以可以像寄存器版本实验一样，直接操作 DHR12R1 寄存器，得到的波形会相对稳定些。

由于使用 HAL 库的函数，CPU 花费的时间会更长（因为指令变多了），在时间精度要求比较高的应用，就不适合用 HAL 库函数来操作了，这一点希望大家明白。所以学 STM32 不是说只要会 HAL 库就可以了，对寄存器也是需要有一定的理解，最好是熟悉。这里用 HAL 库操作只是为了演示怎么使用 HAL 库的相关函数。

最后在 main.c 里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
}
```



```

delay_init(72);          /* 延时初始化 */
usart_init(115200);      /* 串口初始化为 115200 */
usmart_dev.init(72);     /* 初始化 USMART */
led_init();              /* 初始化 LED */
lcd_init();              /* 初始化 LCD */
key_init();              /* 初始化按键 */
dac_init(1);             /* 初始化 DAC1_OUT1 通道 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "DAC Triangular WAVE TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "KEY0:Wave1 KEY1:Wave2", RED);
lcd_show_string(30, 130, 200, 16, 16, "DAC None", BLUE); /* 提示无输出 */

while (1)
{
    t++;
    key = key_scan(0);    /* 按键扫描 */
    if (key == KEY0_PRES) /* 高采样率, 约 1Khz 波形 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "DAC Wave1 ", BLUE);
        /* 幅值 4095, 采样点间隔 5us, 200 个采样点, 100 个波形 */
        dac_triangular_wave(4095, 5, 2000, 100);
        lcd_show_string(30, 130, 200, 16, 16, "DAC None ", BLUE);
    }
    else if (key == KEY1_PRES) /* 低采样率, 约 1Khz 波形 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "DAC Wave2 ", BLUE);
        /* 幅值 4095, 采样点间隔 500us, 20 个采样点, 100 个波形 */
        dac_triangular_wave(4095, 500, 20, 100);
        lcd_show_string(30, 130, 200, 16, 16, "DAC None ", BLUE);
    }
    if (t == 10)          /* 定时时间到了 */
    {
        LED0_TOGGLE();    /* LED0 闪烁 */
        t = 0;
    }
    delay_ms(10);
}

```

该部分代码功能是,按下 KEY0 后,DAC 输出三角波 1,按下 KEY1 后,DAC 输出三角波 2,将 `dac_triangular_wave` 的形参代入公式:输出频率 $\approx 1000 / (dt * \text{samples})$ KHz,得到三角波 1 和三角波 2 的频率都是 0.1KHz。

33.3.4 下载验证

下载代码后,可以看到 LED0 不停的闪烁,提示程序已经在运行了。LCD 显示如图 33.3.4.1 所示:

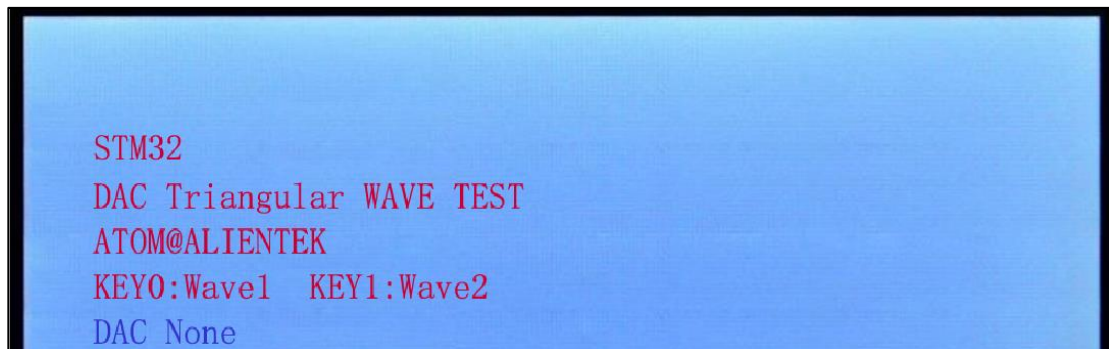


图 33.3.4.1 DAC 输出三角波实验测试图

没有按下任何按键之前，LCD 屏显示 DAC None，当按下 KEY0 后，DAC 输出三角波 1，LCD 屏显示 DAC Wave1，三角波 1 输出完成后 LCD 屏继续显示 DAC None，当按下 KEY1 后，DAC 输出三角波 2，LCD 屏显示 DAC Wave2，三角波 2 输出完成后 LCD 屏继续显示 DAC None。

其中三角波 1 和三角波 2 在示波器的显示情况如下图所示：

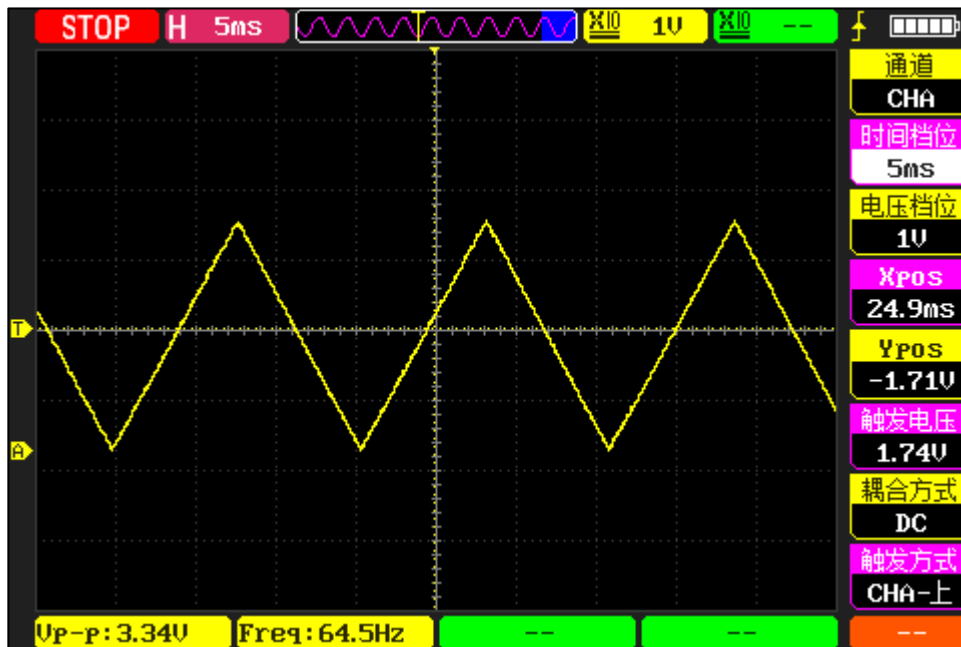


图 33.3.4.2 DAC 输出的三角波 1

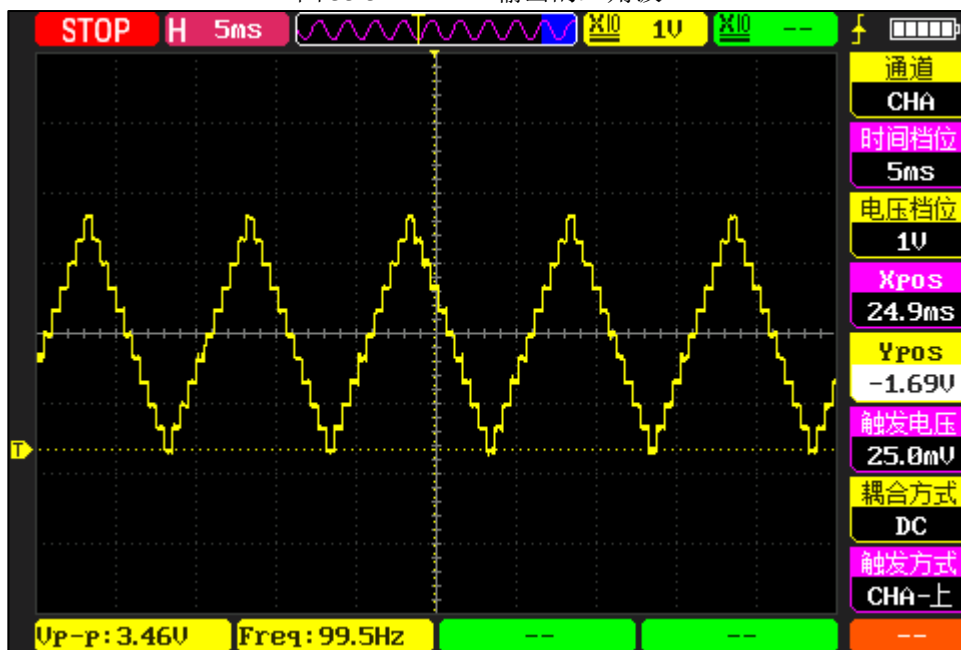


图 33.3.4.3 DAC 输出的三角波 2

由上面两副测试图可以知道，三角波 1 的频率是 64.5Hz，三角波 2 的频率是 99.5Hz。三角波 2 基本接近我们算出来的结果 0.1KHz，三角波 1 有较大误差，在介绍 dac_triangular_wave 函数时也说了原因，加上三角波 1 的采样率比较高，所以误差就会比较大。

33.4 DAC 输出正弦波实验

本实验我们来学习使用如何让 DAC 输出正弦波。实验将用定时器 7 来触发 DAC 进行转换输出正弦波，以 DMA 传输数据的方式。

33.4.1 DAC 寄存器

本实验用到的寄存器在前面的实验都有介绍。

33.4.2 硬件设计

1. 例程功能

使用 DAC 输出正弦波，通过 KEY0/KEY1 两个按键，控制 DAC1 的通道 1 输出两种正弦波，需要通过示波器接 PA4 进行观察。TFTLCD 显示 DAC 转换值、电压值和 ADC 的电压值。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) 独立按键：KEY0 - PE4、KEY1 - PE3
- 5) ADC1：通道 1 - PA1
- 6) DAC1：通道 1 - PA4
- 7) DMA(DMA2_Channel13)
- 8) 定时器 7

3. 原理图

我们只需要把示波器的探头接到 DAC1 通道 1 (PA4) 引脚，就可以在示波器上显示 DAC 输出的波形。PA4 对应 P10 的 DAC 排针，硬件连接如图 33.4.2.1 所示：

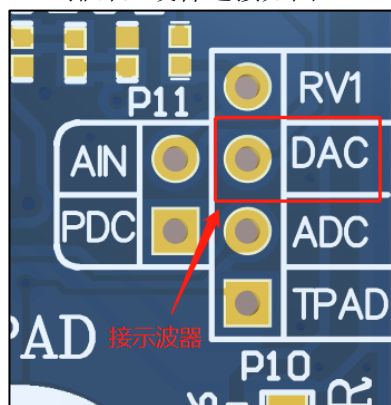


图 33.4.2.1 硬件连接示意图

33.4.3 程序设计

33.4.3.1 DAC 的 HAL 库驱动

本实验用到的 HAL 库 API 函数前面大都介绍过，下面将介绍本实验用到且没有介绍过的。

1. HAL_DAC_Start_DMA 函数

启动 DAC 使用 DMA 方式传输函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Start_DMA(DAC_HandleTypeDef *hdac, uint32_t Channel,
                                     uint32_t *pData, uint32_t Length, uint32_t Alignment);
```

- **函数描述:**
用于启动 DAC 使用 DMA 的方式。
- **函数形参:**
形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。
形参 2 用于选择要启动的通道, 可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。
形参 3 是使用 DAC 输出数据缓冲区的指针。
形参 4 是 DAC 输出数据的长度。
形参 5 是指定 DAC 通道的数据对齐方式, 有: DAC_ALIGN_8B_R (8 位右对齐)、DAC_ALIGN_12B_L (12 位左对齐) 和 DAC_ALIGN_12B_R (12 位右对齐) 三种方式。

- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

2. HAL_DAC_Stop_DMA 函数

停止 DAC 的 DMA 方式函数, 其声明如下:

```
HAL_StatusTypeDef HAL_DAC_Stop_DMA(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

- **函数描述:**
用于停止 DAC 的 DMA 方式。
- **函数形参:**
形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。
形参 2 用于选择要启动的通道, 可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIMEx_MasterConfigSynchronization 函数

配置主模式下的定时器触发输出选择函数, 其声明如下:

```
HAL_StatusTypeDef HAL_TIMEx_MasterConfigSynchronization(
    TIM_HandleTypeDef *htim, TIM_MasterConfigTypeDef *sMasterConfig);
```

- **函数描述:**
用于配置主模式下的定时器触发输出选择。
- **函数形参:**
形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。
形参 2 是 TIM_MasterConfigTypeDef 结构体类型指针变量, 用于配置定时器工作在主/从模式, 以及触发输出(TRGO 和 TRGO2)的选择。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

DAC 输出正弦波配置步骤

1) 开启 DACx 和 DAC 通道对应的 IO 时钟, 并配置该 IO 为模拟功能

首先开启 DACx 的时钟, 然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1, 对应 IO 是 PA4, 它们的时钟开启方法如下:

```
HAL_RCC_DAC_CLK_ENABLE ();          /* 使能 DAC1 时钟 */
HAL_RCC_GPIOA_CLK_ENABLE ();        /* 使能 GPIOA 时钟 */
```

2) 初始化 DACx

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器, 也就是说没有对 DAC 进行任何配置, 它只是 HAL 库提供用来在软件上初始化 DAC。

注意: 该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

3) 配置 DAC 通道

在 HAL 库中, 通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道, 根据需求设置触发类型以及输出缓冲等。

4) 配置 DMA 并关联 DAC

通过 HAL_DMA_Init 函数初始化 DMA，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为 __HAL_LINKDMA。

5) 配置定时器控制触发 DAC

通过 HAL_TIM_Base_Init 函数设置定时器溢出频率。

通过 HAL_TIMEx_MasterConfigSynchronization 函数配置定时器溢出事件用做触发器。

通过 HAL_TIM_Base_Start 函数启动计数。

6) 启动 DAC 转换并以 DMA 方式传输数据

通过 HAL_DAC_Stop_DMA 函数先停止之前的 DMA 传输以及 DAC 输出。

再通过 HAL_DAC_Start_DMA 函数启动 DMA 传输以及 DAC 输出。

33.4.3.2 程序流程图

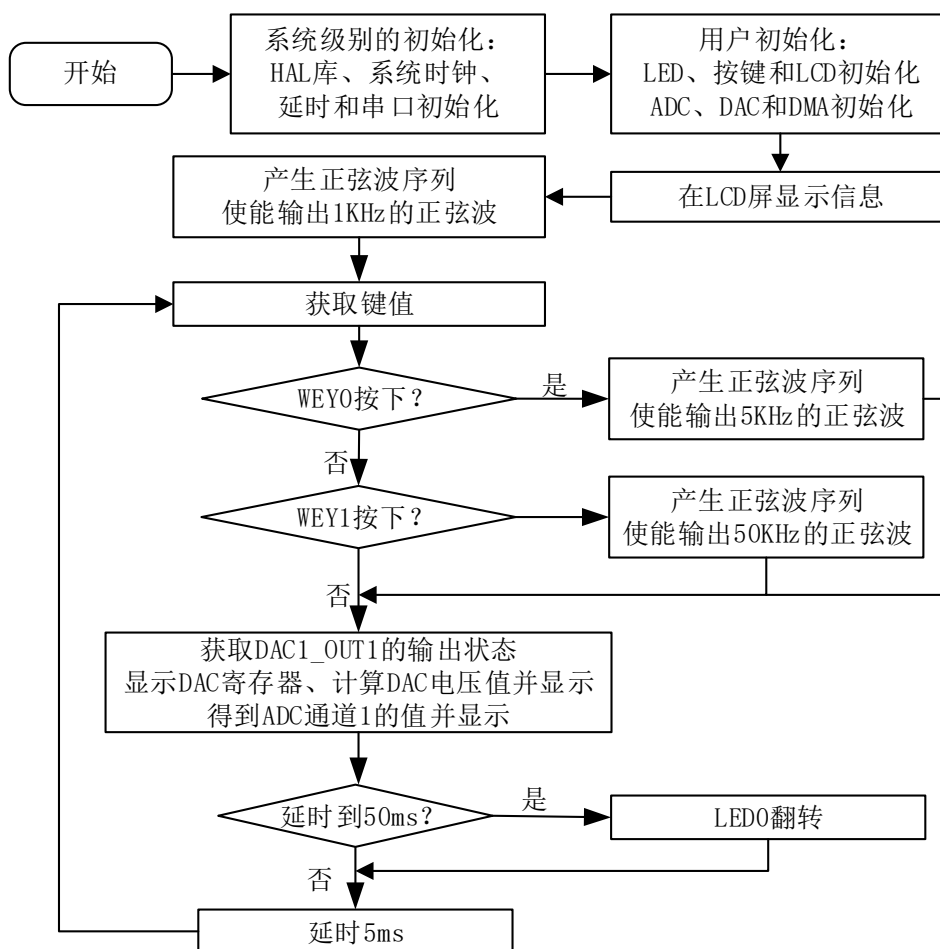


图 33.4.3.2.1 DAC 输出正弦波实验程序流程图

33.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。

dac.h 文件只有一些声明，下面直接开始介绍 dac.c 的程序，本实验的 DAC 以及 DMA 的初始化，我们用到 dac_dma_wave_init 函数，其定义如下：

```

/**
 * @brief      DAC DMA 输出波形初始化函数
 * @note      本函数支持 DAC1_OUT1/2 通道初始化

```

```

*      DAC 的输入时钟来自 APB1, 时钟频率=36Mhz=27.7ns
*      DAC 在输出 buffer 关闭的时候, 输出建立时间:tSETTLING = 4us(F103 数据手册有写)
*      因此 DAC 输出的最高速度约为:300Khz,以 10 个点为一个周期,最大能输出 30Khz 左右的波形
*
* @param      outx: 要初始化的通道. 1,通道 1; 2,通道 2
* @param      par      : 外设地址
* @param      mar      : 存储器地址
* @retval      无
*/
void dac_dma_wave_init(uint8_t outx, uint32_t par, uint32_t mar)
{
    GPIO_InitTypeDef gpio_init_struct;
    DAC_ChannelConfTypeDef dac_ch_conf={0};
    DMA_Channel_TypeDef *dmax_chy;

    if (outx == 1)
    {
        dmax_chy = DMA2_Channel3;      /* OUT1 对应 DMA2_Channel3 */
    }
    else
    {
        dmax_chy = DMA2_Channel4;      /* OUT2 对应 DMA2_Channel4 */
    }

    __HAL_RCC_GPIOA_CLK_ENABLE();      /* DAC 通道引脚端口时钟使能 */
    __HAL_RCC_DAC_CLK_ENABLE();      /* 使能 DAC1 的时钟 */
    __HAL_RCC_DMA2_CLK_ENABLE();      /* DMA2 时钟使能 */

    /* STM32 单片机, 总是 PA4=DAC1_OUT1, PA5=DAC1_OUT2 */
    gpio_init_struct.Pin = (outx==1)? GPIO_PIN_4 : GPIO_PIN_5;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;      /* 模拟 */
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    /* 初始化 DMA */
    g_dma_dac_handle.Instance = dmax_chy;      /* 设置 DMA 通道 */
    g_dma_dac_handle.Init.Direction = DMA_MEMORY_TO_PERIPH; /* 从存储器到外设模式 */
    g_dma_dac_handle.Init.PeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
    g_dma_dac_handle.Init.MemInc = DMA_MINC_ENABLE; /* 存储器增量模式 */
    /* 外设数据长度:16 位 */
    g_dma_dac_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_dac_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_dac_handle.Init.Mode = DMA_CIRCULAR; /* 循环模式 */
    g_dma_dac_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */
    HAL_DMA_Init(&g_dma_dac_handle); /* 初始化 DMA */
    HAL_DMA_Start(&g_dma_dac_handle, mar, par, 0); /* 配置 DMA 传输参数 */

    /* DMA 句柄与 DAC 句柄关联 */
    __HAL_LINKDMA(&g_dac_dma_handle, DMA_Handle1, g_dma_dac_handle);

    /* 初始化 DAC */
    g_dac_dma_handle.Instance = DAC;
    HAL_DAC_Init(&g_dac_dma_handle); /* 初始化 DAC */

    /* 配置 DAC 通道 */
    dac_ch_conf.DAC_Trigger = DAC_TRIGGER_T7_TRGO; /* 使用 TIM7 TRGO 事件触发 */
    dac_ch_conf.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE; /* DAC1 输出缓冲关闭 */

    switch(outx)
    {
        case 1:
            HAL_DAC_ConfigChannel(&g_dac_dma_handle, &dac_ch_conf,

```



```

        DAC_CHANNEL_1); /* DAC 通道 1 配置 */
    break;
case 2:
    HAL_DAC_ConfigChannel(&g_dac_dma_handle, &dac_ch_conf,
        DAC_CHANNEL_2); /* DAC 通道 2 配置 */
    break;
default:break;
}
}

```

该函数用于初始化 DAC 用 DMA 的方式输出正弦波。本函数用到的 API 函数起前面都介绍过，请结合前面介绍过的相关内容来理解源码。这里值得注意的是我们是采用定时器 7 触发 DAC 进行转换输出的。

下面介绍 DAC DMA 使能波形输出函数，其定义如下：

```

/**
 * @brief      DAC DMA 使能波形输出
 * @note      TIM7 的输入时钟频率(f)来自 APB1, f = 36M * 2 = 72Mhz.
 *            DAC 触发频率 ftrgo = f / ((psc + 1) * (arr + 1))
 *            波形频率 = ftrgo / ndtr;
 *
 * @param      outx      : 要初始化的通道. 1,通道 1; 2,通道 2
 * @param      ndtr      : DMA 通道单次传输数据量
 * @param      arr       : TIM7 的自动重装载值
 * @param      psc       : TIM7 的分频系数
 * @retval     无
 */
void dac_dma_wave_enable(uint16_t cndtr, uint16_t arr, uint16_t psc)
{
    TIM_HandleTypeDef tim7_handle= {0};
    TIM_MasterConfigTypeDef tim_master_config= {0};

    __HAL_RCC_TIM7_CLK_ENABLE(); /* TIM7 时钟使能 */

    tim7_handle.Instance = TIM7; /* 选择定时器 7 */
    tim7_handle.Init.Prescaler = psc; /* 预分频 */
    tim7_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数器 */
    tim7_handle.Init.Period = arr; /* 自动装载值 */
    HAL_TIM_Base_Init(&tim7_handle); /* 初始化定时器 7 */

    /* 定时器更新事件用于触发 */
    tim_master_config.MasterOutputTrigger = TIM_TRGO_UPDATE;
    tim_master_config.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    /* 配置定时器 7 的更新事件触发 DAC 转换 */
    HAL_TIMEx_MasterConfigSynchronization(&tim7_handle, &tim_master_config);
    HAL_TIM_Base_Start(&tim7_handle); /* 启动定时器 7 */

    HAL_DAC_Stop_DMA(&g_dac_dma_handle, DAC_CHANNEL_1) /* 先停止之前的传输 */
    HAL_DAC_Start_DMA(&g_dac_dma_handle, DAC_CHANNEL_1,
        (uint32_t *)g_dac_sin_buf, cndtr, DAC_ALIGN_12B_R);
}

```

该函数用于使能波形输出，利用定时器 7 的更新事件来触发 DAC 转换输出。使能定时器 7 的时钟后，调用 HAL_TIMEx_MasterConfigSynchronization 函数配置 TIM7 选择更新事件作为触发输出（TRGO），然后调用 HAL_DAC_Stop_DMA 函数停止 DAC 转换以及 DMA 传输，最后再调用 HAL_DAC_Start_DMA 函数重新配置并启动 DAC 和 DMA。

最后在 main.c 里面编写如下代码：

```

uint16_t g_dac_sin_buf[4096]; /* 发送数据缓冲区 */

/**
 * @brief      产生正弦波序列
 * @note      需保证: maxval > samples/2
 *

```

```

* @param      maxval : 最大值(0 < maxval < 2048)
* @param      samples: 采样点的个数
* @retval      无
*/
void dac_creat_sin_buf(uint16_t maxval, uint16_t samples)
{
    uint8_t i;
    float inc = (2 * 3.1415962) / samples; /* 计算增量 (一个周期 DAC_SIN_BUF 个点) */
    float outdata = 0;

    for (i = 0; i < samples; i++)
    {
        /* 计算以 dots 个点为周期的每个点的值, 放大 maxval 倍, 并偏移到正数区域 */
        outdata = maxval * (1 + sin(inc * i));
        if (outdata > 4095) outdata = 4095; /* 上限限定 */
        //printf("%f\r\n", outdata);
        g_dac_sin_buf[i] = outdata;
    }
}

/**
* @brief      通过 USART 设置正弦波输出参数, 方便修改输出频率.
* @param      arr : TIM7 的自动重装载值
* @param      psc : TIM7 的分频系数
* @retval      无
*/
void dac_dma_sin_set(uint16_t arr, uint16_t psc)
{
    dac_dma_wave_enable(100, arr, psc);
}

int main(void)
{
    uint16_t adcx;
    float temp;
    uint8_t t = 0;
    uint8_t key;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    adc3_init(); /* 初始化 ADC */

    adc3_channel_set(&g_adc3_handle, ADC3_CHY, ADC_CHANNEL_0,
                    ADC_SAMPLETIME_1CYCLE_5);

    /* 初始化 DAC 通道 1 DMA 波形输出 */
    dac_dma_wave_init(1, (uint32_t)&DAC1->DHR12R1, (uint32_t)g_dac_sin_buf);

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DAC DMA Sine WAVE TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:3Khz KEY1:30Khz", RED);

    lcd_show_string(30, 130, 200, 16, 16, "DAC VAL:", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "DAC VOL:0.000V", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "ADC VOL:0.000V", BLUE);

```

```

dac_creat_sin_buf(2048, 100);
/* 100Khz 触发频率, 100 个点, 得到 1Khz 的正弦波 */
dac_dma_wave_enable(100, 10 - 1, 72 - 1);

while (1)
{
    t++;
    key = key_scan(0); /* 按键扫描 */

    if (key == KEY0_PRES) /* 高采样率, 约 1Khz 波形 */
    {
        dac_creat_sin_buf(2048, 100);
        /* 300Khz 触发频率, 100 个点, 得到最高 3KHz 的正弦波. */
        dac_dma_wave_enable(100, 10 - 1, 24 - 1);
    }
    else if (key == KEY1_PRES) /* 低采样率, 约 1Khz 波形 */
    {
        dac_creat_sin_buf(2048, 10);
        /* 300Khz 触发频率, 10 个点, 可以得到最高 30KHz 的正弦波. */
        dac_dma_wave_enable(10, 10 - 1, 24 - 1);
    }

    adcx = DAC1->DHR12R1; /* 获取 DAC1_OUT1 的输出状态 */
    lcd_show_xnum(94, 130, adcx, 4, 16, 0, BLUE); /* 显示 DAC 寄存器值 */

    temp = (float)adcx * (3.3 / 4096); /* 得到 DAC 电压值 */
    adcx = temp;
    lcd_show_xnum(94, 150, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */

    temp -= adcx;
    temp *= 1000;
    lcd_show_xnum(110, 150, temp, 3, 16, 0x80, BLUE); /* 显示电压值的小数部分 */

    adcx = adc3_get_result_average(ADC3_CHY, 10); /* 得到 ADC3 通道 1 的转换结果 */
    temp = (float)adcx * (3.3 / 4096); /* 得到 ADC 电压值 (adc 是 12bit 的) */
    adcx = temp;
    lcd_show_xnum(94, 170, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */

    temp -= adcx;
    temp *= 1000;
    lcd_show_xnum(110, 170, temp, 3, 16, 0x80, BLUE); /* 显示电压值的小数部分 */

    if (t == 40) /* 定时时间到了 */
    {
        LED0_TOGGLE(); /* LED0 闪烁 */
        t = 0;
    }

    delay_ms(5);
}
}

```

adc3_init 函数初始化 ADC3, 用于测量 DAC 通道 1 的电压值。

dac_dma_wave_init 函数初始化 DAC 通道 1, 并指定 DMA 搬运的数据的开始地址和目标地址。dac_creat_sin_buf 函数用于产生正弦波序列, 并保存在 g_dac_sin_buf 数组中, 供给 DAC 转换。在进入 while(1) 循环之前, dac_dma_wave_enable 函数默认配置 DAC 的采样点个数时 100, 并配置定时器 7 的溢出频率为 100KHz。这样就可以输出 1KHz 的正弦波。下面给大家解释一下为什么是输出 1KHz 的正弦波?

定时器 7 的溢出频率为 100KHz, 不记得怎么计算的朋友, 请回顾基本定时器的相关内容, 这里直接把公式列出:

$$T_{out} = ((arr+1) * (psc+1)) / T_{clk}$$

看到 `dac_dma_wave_enable(100, 10 - 1, 72 - 1);` 这个语句，第二个形参是自动重装载值，第三个形参是分频系数，那么代入公式，可得：

$$T_{out} = ((arr+1)*(psc+1))/T_{clk} = ((9+1)*(71+1))/72000000 = 0.00001s$$

得到定时器的更新事件周期是 0.00001 秒，即更新事件频率为 100KHz，也就得到 DAC 输出触发频率为 100KHz。

再结合总一个正弦波共有 100 个采样点，就可以得到正弦波的频率为 $100KHz/100 = 1KHz$ 。

知道了正弦波的频率怎么来的，下面代码中，按下按键 KEY0，得到 3KHz 的正弦波，按下按键 KEY1，得到 30KHz 的正弦波，计算方法都一样的。

`dac_dma_sin_set` 函数可以通过 USART 设置正弦波输出参数，方便修改输出频率。

33.4.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示如图 33.4.4.1 所示：

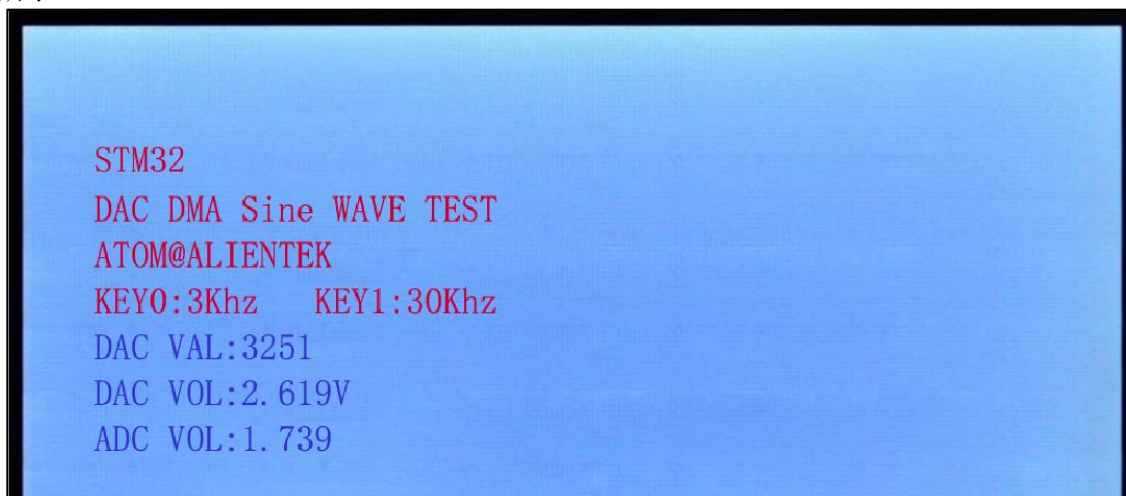


图 33.4.4.1 DAC 输出正弦波实验测试图

用短路帽将 ADC 和 DAC 排针连接后，可以看到 ADC VOL 的值随着 DAC 的输出变化而变化，即 ADC 采集到的值是不停变化的。由于变化太快了，这样看不出采集到值形成什么波形，下面我们借用示波器来进行观察，首先将探头接到 DAC 的排针上。

没有按下任何按键之前，默认输出 1KHz（100 个采样点）的正弦波，如下图所示：

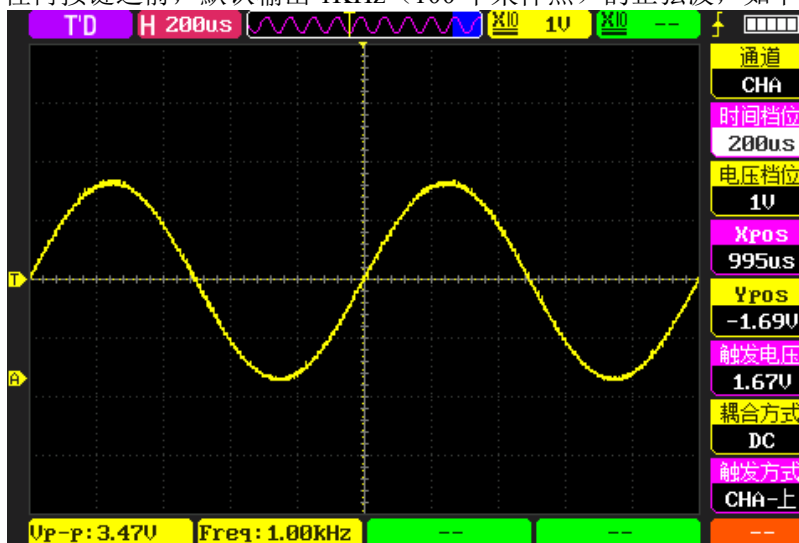


图 33.3.4.2 默认 DAC 输出的的正弦波

当按下 KEY0 后，DAC 输出 3KHz（100 个采样点）的正弦波，如下图所示：

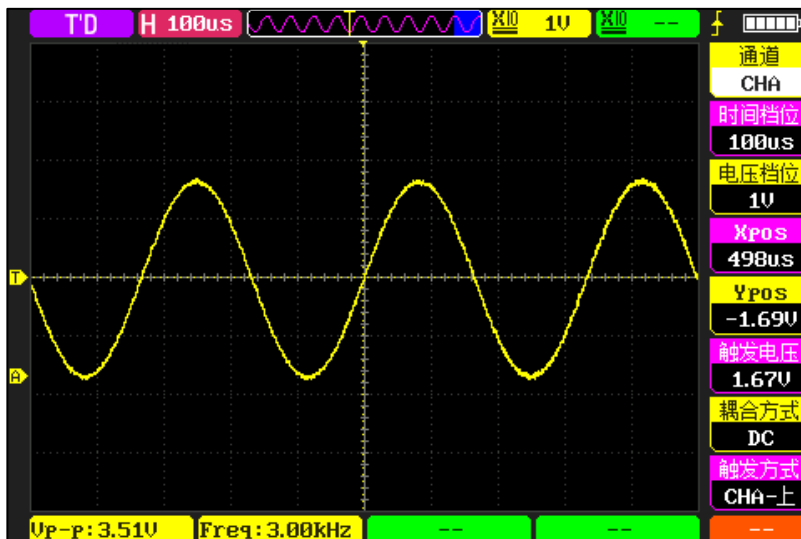


图 33.3.4.3 按下 KEY0, DAC 输出的的正弦波
当按下 KEY1 后, DAC 输出 30KHz (10 个采样点) 的正弦波, 如下图所示:

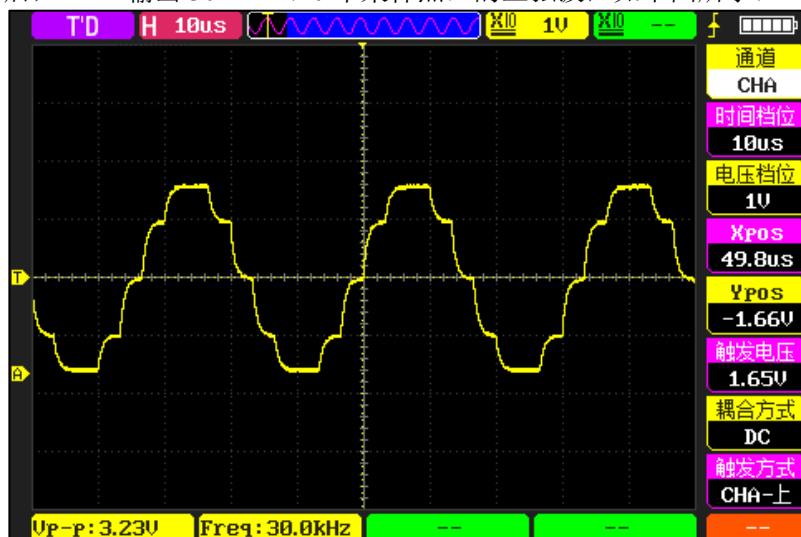


图 33.3.4.4 按下 KEY1, DAC 输出的的正弦波

第三十四章 PWM DAC 实验

前面的章节，我们介绍了 STM32F1 自带 DAC 模块的使用，虽然 STM32F103ZET6 具有内部 DAC，但是也仅仅只有两条 DAC 通道，而 STM32 还有其它的很多型号是没有 DAC 的。通常情况下，采用专用的 D/A 芯片来实现，但是这样就会带来成本的增加。不过 STM32 所有的芯片都有 PWM 输出，并且 PWM 输出通道很多，资源丰富。因此，我们可以使用 PWM + 简单的 RC 滤波来实现 DAC 的输出从而节省成本。

本章我们将向大家介绍如何使用 STM32F1 的 PWM 来设计一个 DAC。我们将使用按键（或 USART）控制 STM32F1 的 PWM 输出，从而控制 PWM DAC 的输出电压，通过 ADC1 的通道 1 采集 PWM DAC 的输出电压，并在 LCD 模块上面显示 ADC 获取到的电压值以及 PWM DAC 的设定输出电压值等信息本章分为如下几个小节：

34.1 PWM DAC 技术的实现原理

34.2 硬件设计

34.3 程序设计

34.4 下载验证

34.1 PWM DAC 技术的实现原理

DAC 工作过程是将源电压按照 8 位、12 位、16 位等分辨率进行分割，其输出的电压是最小精度 LSB（即 $1/2^8$ 、 $1/2^{12}$ 、 $1/2^{16}$ 等）的整数倍，这就是 DAC 输出的电压。

我们来分析一下 PWM 波形的特性。PWM 信号可以被分解为一个直流分量和一个占空比固定，但是平均幅度为零的方波，如图 34.1.1.1 所示。

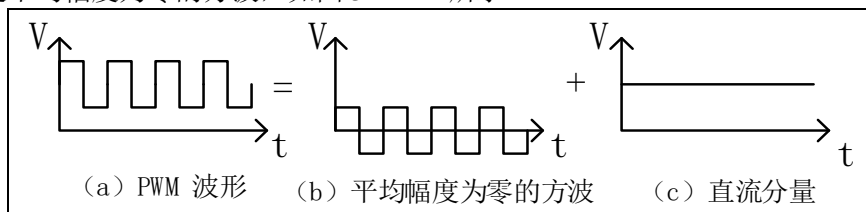


图 34.1.1.1 PWM 波形的等效分解

如果使 PWM 信号的占空比随时间改变，那么其直流分量随之改变，信号滤除交流分量后，将会输出幅度变化的模拟信号。因此通过改变 PWM 信号的占空比，可以产生不同的模拟信号。这种技术称之为 PWM DAC。

PWM 是周期固定，占空比可调的数字信号。在实际电路中，典型的 PWM 波形，如图 34.1.1.2 所示。

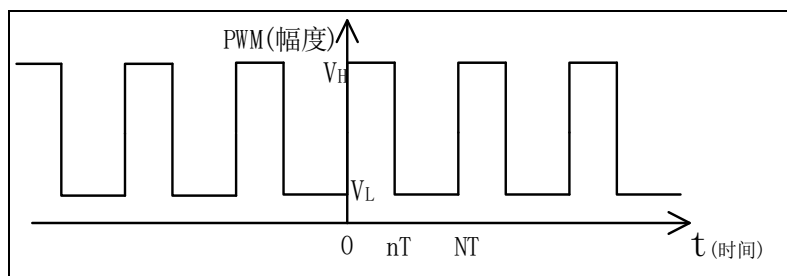


图 34.1.1.2 实际电路典型 PWM 波形图

下面根据高数与信号与系统课程的知识我们作一个简单的推导，感兴趣的同学可以查阅对应的知识，如果不感兴趣，直接跳过推导过程，看最后的结论即可。

我们可以把 PWM 波形用分段函数①表示出来。占空比可以用②的表达式来表示。

$$f(t) = \begin{cases} V_H, & kNT \leq x < kNT + nT \\ V_L, & kNT + nT \leq x \leq kNT + NT \end{cases} \quad ①$$

$$p = \frac{n}{N} \quad ②$$

PWM 是一个周期信号，我们令周期为 NT，由傅里叶变换的知识可知任意周期信号都可按频域展开，我们把分段表达式按频域展开。得到如③的表达式，④⑤⑥为展开系数：

$$f(t) = A_0 + \sum_{k=1}^{\infty} [A_n \cos\left(\frac{2k\pi t}{NT}\right) + B_n \sin\left(\frac{2k\pi t}{NT}\right)] \quad ③$$

$$A_0 = \frac{1}{2NT} \int_{-NT}^{NT} f(t) dt \quad ④$$

$$A_k = \frac{1}{2NT} \int_{-NT}^{NT} f(t) \cos\left(\frac{2k\pi t}{NT}\right) dt \quad ⑤$$

$$B_k = \frac{1}{2NT} \int_{-NT}^{NT} f(t) \sin\left(\frac{2k\pi t}{NT}\right) dt \quad ⑥$$

我们知道 PWM 的幅度为 $V_H - V_L$ ，占空比为 p，代入公式③~⑥求对应的积分，可以求出 f(t) 的展开系数分别如下：

$$A_0 = (V_H - V_L) * p \quad ⑦$$

$$A_k = (V_H - V_L) \times \frac{1}{k\pi} [\sin(k\pi p) - \sin(2k\pi(1 - \frac{p}{2}))] \quad ⑧$$

$$B_k = 0 \quad ⑨$$

根据展开得到的频率参数，由此可得出 PWM 信号及其占空比在时域上的表达式。

$$f(t) = (V_H - V_L) * p + \sum_{k=1}^{\infty} [(V_H - V_L) \times \frac{1}{k\pi} [\sin(k\pi p) - \sin(2k\pi(1 - \frac{p}{2}))]] \times \cos\left(\frac{2k\pi t}{NT}\right) \quad ⑩$$

公式⑩正好验证了图 34.1.1.1 的 PWM 等效原理。由此我们可知 PWM 的输出波形为一个与占空比有关的直流等效信号，同时伴有多个不同频率的信号叠加。如果能把这些频率信号尽可能过滤掉，那么我们通过调整 PWM 的占空比即可方便实现我们需要的 DAC 结果，即 $V_{DAC} = (V_H - V_L) * p$ 。

分辨率也是 DAC 一个重要的参数，它可以表示 DAC 输出的最小精度。存在两个主要误差源影响 PWM 方式 DAC 分辨率。首先，PWM 信号的占空比只能表示有限的分辨率。这是因为 STM32 的 PWM 的占空比是输出比较寄存器 CCRx 与 TIMx_CNT 进行比较的结果，而 CCRx 在 STM32F1 系列中最多能设置为 16 位。那么很显然地，用 PWM 实现的 DAC 分辨率就与 TIMx_CNT 有关，即定时器的时钟频率越高则 CCRx 可以设置的值越多，分辨率相应地越高。但由于定时器最高时钟是 72M，这也会导致分辨率越高，DAC 的速度越慢。

第二个误差源是 PWM 信号中不期望的谐波分量产生的峰峰值。前面 PWM 的频域展开公式⑩说明 PWM 信号需要通过滤波器才能输出一个纹波较小的直流信号，但实际上对于简单设计的滤波器对交流信号的过滤能力是有限的，所以输出信号还会带有一定的交流成份。

根据公式⑧，将 k=1 代入我们可以算出 PWM 的一次谐波幅度：

$$A_1 = (V_H - V_L) \times \frac{1}{\pi} [\sin(\pi p) - \sin(2\pi(1 - \frac{p}{2}))] = \frac{2}{\pi} (V_H - V_L) \sin(\pi p - \pi) \quad ⑪$$

当 $\sin(\pi p - \pi) = 1$ 时滤波器需要达到衰减峰值，可知 PWM 占空比为 50% 时，一次谐波的幅度最大。为了减少这个基波的影响，我们希望滤波器在这个最大幅度下也能把基波的交流影响衰减到 1/2LSB 以下，即后外围滤波器至少需要满足以下条件才能避免 DAC 输出干扰过大：

$$A_{k=1} \times A_{filter} \leq \frac{1}{2} \times \frac{(V_H - V_L)}{2^Y} \quad ⑫$$

根据公式⑪⑫可知 $A_{filter} \leq \frac{\pi}{2^{Y+2}} = 20 \times \log(\frac{\pi}{2^{Y+2}})$ ，当 DAC 为 12 位精度时，代入 Y=12 可知我们设计的滤波器需要衰减 74dB 以上，当为 8 位精度时，衰减需要达到 50dB。

我们知道一阶 RC 电路截止频率计算公式为：

$$f_c = \frac{1}{2\pi RC}$$

把电容等效成一个电阻，对于一阶分压时电压的等效衰减的表达式可以是：

$$V_{out} = V_{IN} \left(\frac{1}{\sqrt{(2\pi f_c R)^2 + 1^2}} \right)$$

根据以上公式就能很好地设计一个满足我们需要的滤波器参数了。为了实现低成本的 RC 电路，我们使用两个一阶 RC 电路串联起来作为滤波器。

STM32F103 的定时器最快的计数频率是 72Mhz，8 位分辨率的时候，PWM 频率为 72M/256=281.25Khz。我们需要把交流信号至少衰减 50dB 左右。

34.2 硬件设计

1. 例程功能

我们将设计一个 8 位的 DAC，使用按键（或 USART）控制 STM32F1 的 PWM 输出（占空比），从而控制 PWM DAC 的输出电压。为了得知 PWM 的输出电压，通过 ADC1 的通道 1 采集 PWM DAC 的输出电压，并在 LCD 模块上面显示 ADC 获取到的电压值以及 PWM DAC 的设定输出电压值等信息。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY_UP - PA0
- 3) PWM 输出通道
TIM1 的通道 1, 对应 IO 是 PA8

3. 原理图

根据前面分析的原理，在硬件设计上就比较简单了。PWM 可以由 STM32 定时器输出，我们只需要在外围增加一个滤波电路即可。我们使用的是 RC 滤波电路，其电路设计如下图所示：

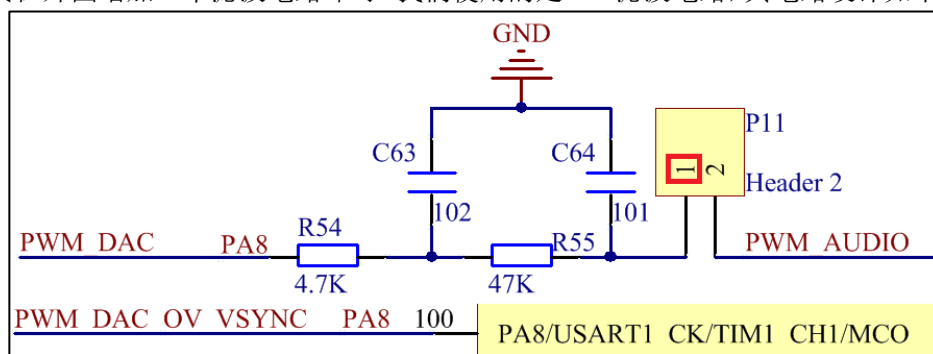


图 25.2.1 PWM DAC 连接原理设计

根据我们的设计,输出 8 位 DAC 时,经过一阶滤波后 DAC 输出的交流信号大概的衰减可以达到 117dB,可见我们设计是符合要求的。

这里有个特别需要注意的地方：因为 PWM_DAC 和 OV_VSYNC 共用了 PA8 引脚，所以在做本实验的时候，不能插摄像头模块或 OLED 模块，否则可能会影响 PWM 转换结果。

如下图所示，本实验需要用短路帽将 PDC 和 ADC 排针连接起来。

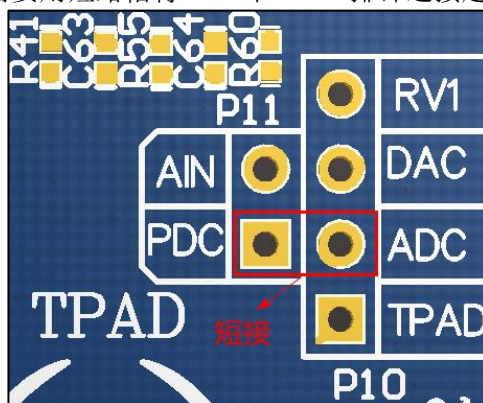


图 25.2.2 PCB 对应 PWM DAC 的位置

34.3 程序设计

34.3.1 程序流程图

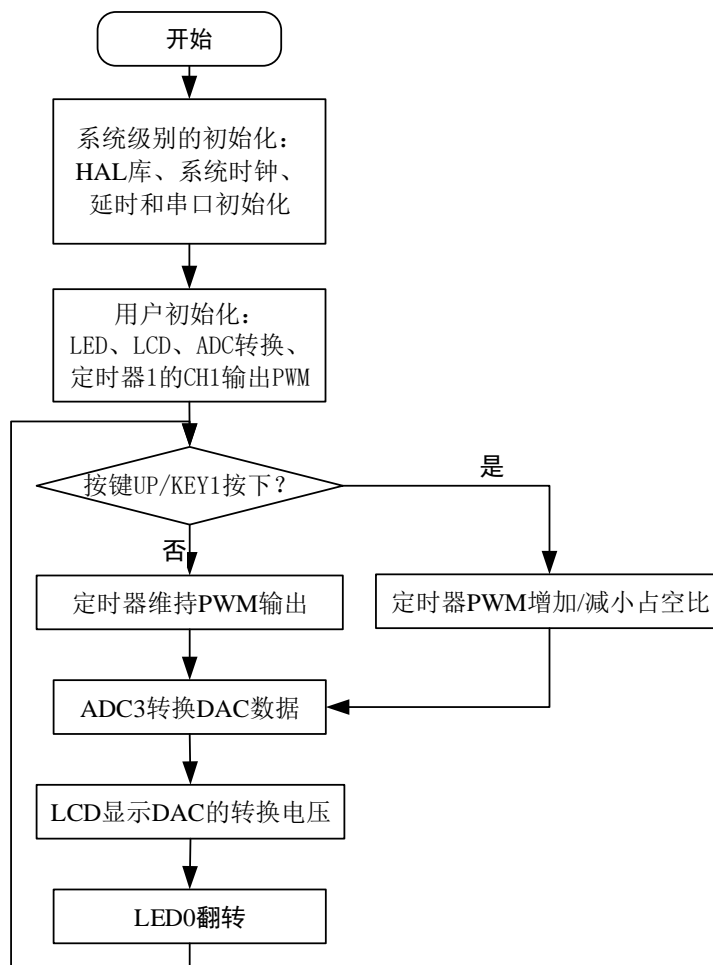


图 34.3.1.1 PWM DAC 实验程序流程图

34.3.2 程序解析

1. PWM DAC 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWM DAC 驱动源码包括两个文件：pwm dac.c 和 pwm dac.h。

在 pwm dac.h 中，定义的宏定义如下：

```
/*
 * PWMDAC 默认是使用 PA8，对应的定时器为 TIM1_CH1，如果你要修改成其他 IO 输出，则相应
 * 的定时器及通道也要进行修改。请根据实际情况进行修改。
 */
#define PWMDAC_GPIO_PORT      GPIOA
#define PWMDAC_GPIO_PIN      GPIO_PIN_8
/* PA 口时钟使能 */
#define PWMDAC_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE();}while(0)

#define PWMDAC_TIMX          TIM1
#define PWMDAC_TIMX_CHY      TIM_CHANNEL_1      /* 通道 Y, 1<= Y <=4 */
#define PWMDAC_TIMX_CCRX      PWMDAC_TIMX->CCR1  /* 通道 Y 的输出比较寄存器 */
/* TIM1 时钟使能 */
#define PWMDAC_TIMX_CLK_ENABLE() do{ __HAL_RCC_TIM1_CLK_ENABLE();}while(0)
```

下面介绍 pwmdac.c 文件的函数，首先是 pwmdac_init 函数，其定义如下：

```
/**
 * @brief      PWM DAC 初始化，实际上就是初始化定时器
 * @note
 *
 *      定时器的时钟来自 APB1 / APB2，当 APB1 / APB2 分频时，定时器频率自动翻倍
 *      所以，一般情况下，我们所有定时器的频率，都是 72Mhz 等于系统时钟频率
 *      定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *      Ft = 定时器工作频率，单位：Mhz
 * @param      arr: 自动重装值。
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void pwmdac_init(uint16_t arr, uint16_t psc)
{
    TIM_OC_InitTypeDef timx_oc_pwmdac = {0};

    PWMDAC_TIMX_CLK_ENABLE(); /* PWM DAC 定时器时钟使能 */

    g_tim1_handle.Instance = TIM1; /* 定时器 1 */
    g_tim1_handle.Init.Prescaler = psc; /* 定时器分频 */
    g_tim1_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数模式 */
    g_tim1_handle.Init.Period = arr; /* 自动重装载值 */
    /* 使能 TIMx_ARR 进行缓冲 */
    g_tim1_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    HAL_TIM_PWM_Init(&g_tim1_handle); /* 初始化 PWM */

    timx_oc_pwmdac.OCMode = TIM_OCMode_PWM1; /* CH1/2 PWM 模式 1 */
    timx_oc_pwmdac.Pulse = 0; /* 设置比较值,此值用来确定占空比 */
    timx_oc_pwmdac.OCpolarity = TIM_OCPOLARITY_HIGH; /* 输出比较极性为高 */
    /* 配置 TIM1 通道 1 */
    HAL_TIM_PWM_ConfigChannel(&g_tim1_handle, &timx_oc_pwmdac, PWMDAC_TIMX_CHY);
    HAL_TIM_PWM_Start(&g_tim1_handle, TIM_CHANNEL_1); /* 开启定时器 1 通道 1 */
}
```

HAL_TIM_PWM_Init 初始化 TIM1 并设置 TIM1 的 ARR 和 PSC 等参数，其次通过调用函数 HAL_TIM_PWM_ConfigChannel 设置定时器通道使用 PWM1 模式以及设置比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM1 以及使能 PWM 通道 TIM1_CH1 输出。

HAL_TIM_PWM_Init 会调用 HAL_TIM_PWM_MspInit 函数，用于存放 GPIO、NVIC 和时钟相关的代码。HAL_TIM_PWM_MspInit 函数定义如下：

```
/**
 * @brief      定时器底层驱动，时钟使能，引脚配置
 * @note
 *
 *      此函数会被 HAL_TIM_PWM_Init() 调用
 * @param      htim: 定时器句柄
 * @retval     无
 */
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
{
    GPIO_InitTypeDef gpio_init_struct;

    if (htim->Instance == TIM1)
    {
        __HAL_RCC_TIM1_CLK_ENABLE(); /* 使能定时器 1 */
        __HAL_AFIO_REMAP_TIM1_PARTIAL(); /* TIM1 通道引脚部分重映射使能 */
        PWMDAC_GPIO_CLK_ENABLE(); /* GPIO 时钟使能 */

        gpio_init_struct.Pin = PWMDAC_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;
        gpio_init_struct.Pull = GPIO_PULLUP;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(PWMDAC_GPIO_PORT, &gpio_init_struct);
    }
}
```

```

    }
}

```

下面介绍 `pwm dac_set_voltage` 函数，该函数先计算得到比较值，然后通过设置比较值来改变占空比，从而控制 PWM DAC 输出的电压值，其定义如下：

```

/**
 * @brief      设置 PWM DAC 输出电压
 * @param      vol : 0~3300, 代表 0~3.3V
 * @retval     无
 */
void pwm dac_set_voltage(uint16_t vol)
{
    float temp = vol;
    temp /= 100;          /* 缩小 100 倍, 得到实际电压值 */
    temp = temp * 256 / 3.3; /* 将电压转换成 PWM 占空比 */
    __HAL_TIM_SET_COMPARE(&g_tim1_handler, PWM DAC_TIMX_CHY, temp); /* 设置新的占空比 */
}

```

最后在 main 函数里面编写如下代码：

```

extern TIM_HandleTypeDef g_tim1_handler;

int main(void)
{
    uint16_t adcx;
    float temp;
    uint8_t t = 0;
    uint8_t key;
    uint16_t pwmval = 0;

    HAL_Init();          /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72);       /* 延时初始化 */
    usart_init(115200);    /* 串口初始化为 115200 */
    usmart_dev.init(72);   /* 初始化 USMART */
    led_init();            /* 初始化 LED */
    lcd_init();            /* 初始化 LCD */
    key_init();            /* 初始化按键 */
    adc3_init();           /* 初始化 ADC */
    pwm dac_init(256 - 1, 0); /* PWM DAC 初始化, Fpwm = 72M/256 = 281.25Khz */
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "PWM DAC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY_UP:+ KEY1:-", RED);
    lcd_show_string(30, 130, 200, 16, 16, "PWM VAL:", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "DAC VOL:0.000V", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "ADC VOL:0.000V", BLUE);
    while (1)
    {
        t++;
        key = key_scan(0); /* 按键扫描 */
        if (key == WKUP_PRES) /* PWM 占空比调高 */
        {
            if (pwmval < 250) /* 范围限定 */
            {
                pwmval += 10;
            }
            /* 输出新的 PWM 占空比 */
            __HAL_TIM_SET_COMPARE(&g_tim1_handler, PWM DAC_TIMX_CHY, pwmval);
        }
        else if (key == KEY1_PRES) /* PWM 占空比调低 */
        {
            if (pwmval > 10) /* 范围限定 */
            {
                pwmval -= 10;
            }
        }
    }
}

```

```

    }
    else
    {
        pwmval = 0;
    }
    /* 输出新的 PWM 占空比 */
    __HAL_TIM_SET_COMPARE(&g_tim1_handler, PWM_DAC_TIMX_CHY, pwmval);
}
if (t == 10 || key == KEY1_PRES || key == WKUP_PRES)
{
    /* WKUP / KEY1 按下了, 或者定时时间到了 */
    /* PWM DAC 定时器输出比较值 */
    adcx = __HAL_TIM_GET_COMPARE(&g_tim1_handler, PWM_DAC_TIMX_CHY);
    lcd_show_xnum(94, 130, adcx, 3, 16, 0, BLUE); /* 显示 CCRX 寄存器值 */
    temp = (float)adcx * (3.3 / 256); /* 得到 DAC 电压值 */
    adcx = temp;
    lcd_show_xnum(94, 150, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */
    temp -= adcx;
    temp *= 1000;
    lcd_show_xnum(110, 150, temp, 3, 16, 0x80, BLUE); /* 电压值的小数部分 */
    adc3 = adc3_get_result_average(ADC3_CHY, 10); /* ADC3 通道 1 的转换结果 */
    temp = (float)adc3 * (3.3 / 4096); /* 得到 ADC 电压值 (adc 是 12bit 的) */
    adc3 = temp;
    lcd_show_xnum(94, 170, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */
    temp -= adc3;
    temp *= 1000;
    lcd_show_xnum(110, 170, temp, 3, 16, 0x80, BLUE); /* 电压值的小数部分 */
    LED0_TOGGLE(); /* LED0 闪烁 */
    t = 0;
}
delay_ms(10);
}
}

```

main 函数初始化了 LED 和 LCD 用于显示效果, 初始化按键和 ADC 用于修改 ADC 的占空比, 辅助显示 ADC。

34.4 下载验证

下载代码后, LED0 不停的闪烁, 提示程序已经在运行了。此时, 可以通过按下 KEY_UP 按键增大输出电压, 按下 KEY1 按键则电压变小。

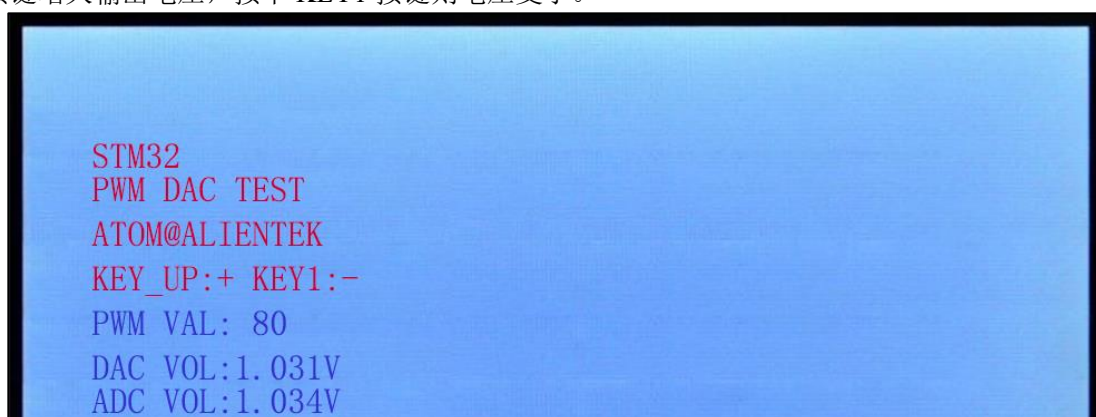


图 34.4.1 TFTLCD 显示效果图

注意: 因为 PWM_DAC 和 OV_VSYNC 共用了 PA8 引脚, 所以在做本例程的时候, 不能插摄像头模块或 OLED 模块, 否则可能会影响 PWM 转换结果!!!

第三十五章 IIC 实验

本章，我们将介绍如何使用 STM32F103 的普通 IO 口模拟 IIC 时序，并实现和 24C02 之间的双向通信，并把结果显示在 TFTLCD 模块上。本章分为如下几个小节：

35.1 IIC 及 24C02 介绍

35.2 硬件设计

35.3 程序设计

35.4 下载验证

35.1 IIC 及 24C02 介绍

35.1.1 IIC 简介

IIC(Inter-Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器以及其外围设备。它是由数据线 SDA 和时钟线 SCL 构成的串行总线，可发送和接收数据，在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送。

IIC 总线有如下特点：

①总线由数据线 SDA 和时钟线 SCL 构成的串行总线，数据线用来传输数据，时钟线用来同步数据收发。

②总线上每一个器件都有一个唯一的地址识别，所以我们只需要知道器件的地址，根据时序就可以实现微控制器与器件之间的通信。

③数据线 SDA 和时钟线 SCL 都是双向线路，都通过一个电流源或上拉电阻连接到正的电压，所以当总线空闲的时候，这两条线路都是高电平。

④总线上数据的传输速率在标准模式下可达 100kbit/s 在快速模式下可达 400kbit/s 在高速模式下可达 3.4Mbit/s。

⑤总线支持设备连接。在使用 IIC 通信总线时，可以有多个具备 IIC 通信能力的设备挂载在上面，同时支持多个主机和多个从机，连接到总线的接口数量只由总线电容 400pF 的限制决定。IIC 总线挂载多个器件的示意图，如下图所示：

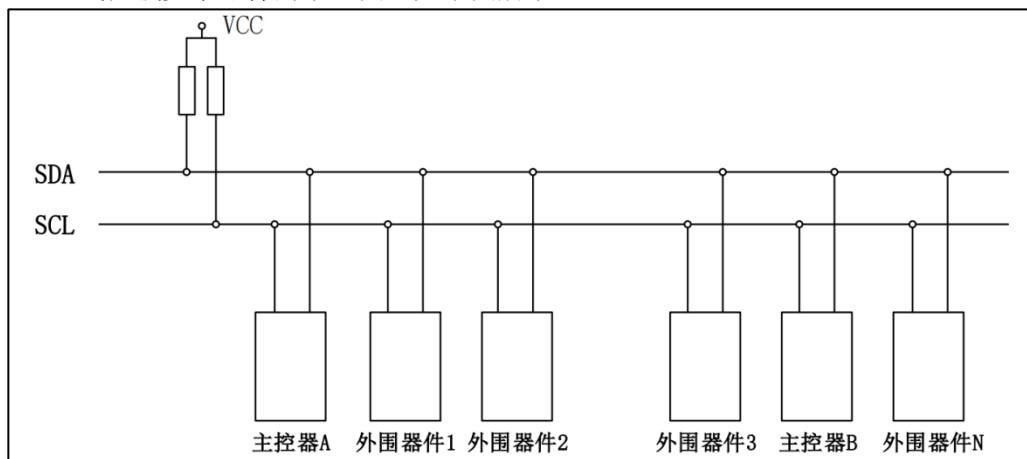


图 35.1.1.1 IIC 总线挂载多个器件

下面来学习 IIC 总线协议，IIC 总线时序图如下所示：

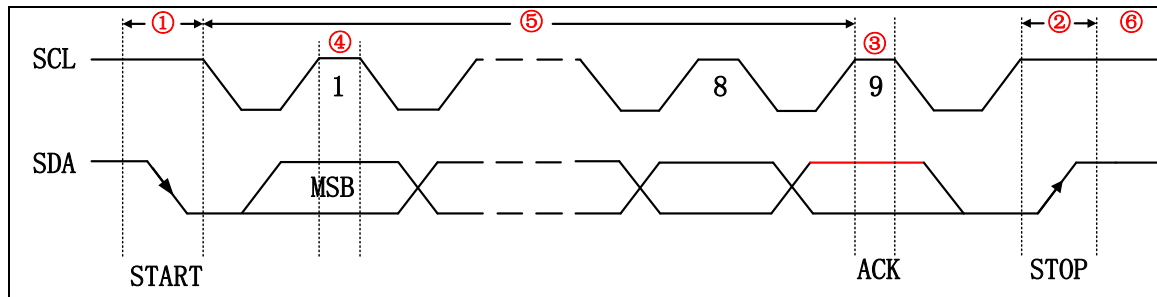


图 35.1.1.2 IIC 总线时序图

为了便于大家更好的了解 IIC 协议，我们从起始信号、停止信号、应答信号、数据有效性、数据传输以及空闲状态等 6 个方面讲解，大家需要对应图 35.1.1.2 的标号来理解。

① 起始信号

当 SCL 为高电平期间，SDA 由高到低的跳变。起始信号是一种电平跳变时序信号，而不是一个电平信号。该信号由主机发出，在起始信号产生后，总线就处于被占用状态，准备数据传输。

② 停止信号

当 SCL 为高电平期间，SDA 由低到高的跳变。停止信号也是一种电平跳变时序信号，而不是一个电平信号。该信号由主机发出，在停止信号发出后，总线就处于空闲状态。

③ 应答信号

发送器每发送一个字节，就在时钟脉冲 9 期间释放数据线，由接收器反馈一个应答信号。应答信号为低电平时，规定为有效应答位（ACK 简称应答位），表示接收器已经成功地接收了该字节；应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功。

观察上图标号③就可以发现，有效应答的要求是从机在第 9 个时钟脉冲之前的低电平期间将 SDA 线拉低，并且确保在该时钟的高电平期间为稳定的低电平。如果接收器是主机，则在它收到最后一个字节后，发送一个 NACK 信号，以通知被控发送器结束数据发送，并释放 SDA 线，以便主机接收器发送一个停止信号。

④ 数据有效性

IIC 总线进行数据传送时，时钟信号为高电平期间，数据线上的数据必须保持稳定，只有在时钟线上的信号为低电平期间，数据线上的高电平或低电平状态才允许变化。数据在 SCL 的上升沿到来之前就需准备好。并在下降沿到来之前必须稳定。

⑤ 数据传输

在 I2C 总线上传送的每一位数据都有一个时钟脉冲相对应（或同步控制），即在 SCL 串行时钟的配合下，在 SDA 上逐位地串行传送每一位数据。数据位的传输是边沿触发。

⑥ 空闲状态

IIC 总线的 SDA 和 SCL 两条信号线同时处于高电平时，规定为总线的空闲状态。此时各个器件的输出级场效应管均处在截止状态，即释放总线，由两条信号线各自的上拉电阻把电平拉高。

了解前面的知识后，下面介绍一下 IIC 的基本的读写通讯过程，包括主机写数据到从机即写操作，主机到从机读取数据即读操作。下面先看一下写操作通讯过程图，见图 35.1.3.2 所示：

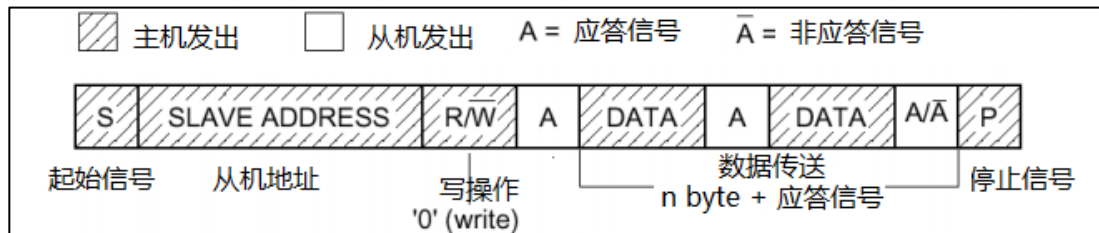


图 35.1.3.2 写操作通讯过程图

主机首先在 IIC 总线上发送起始信号，那么这时总线上的从机都会等待接收由主机发出的数据。主机接着发送从机地址+0(写操作)组成的 8bit 数据，所有从机接收到该 8bit 数据后，自

行检验是否是自己的设备的地址，假如是自己的设备地址，那么从机就会发出应答信号。主机在总线上接收到有应答信号后，才能继续向从机发送数据。注意：IIC 总线上传送的数据信号是广义的，既包括地址信号，又包括真正的数据信号。

接着讲解一下 IIC 总线的读操作过程，先看一下读操作通讯过程图，见图 35.1.3.3 所示。

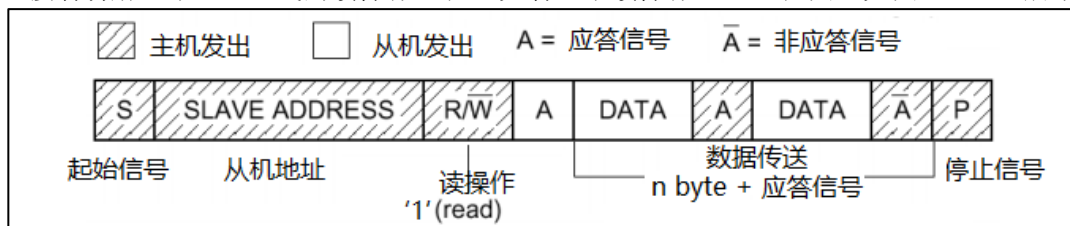


图 35.1.3.3 读操作通讯过程图

主机向从机读取数据的操作，一开始的操作与写操作有点相似，观察两个图也可以发现，都是由主机发出起始信号，接着发送从机地址+1(读操作)组成的 8bit 数据，从机接收到数据验证是否是自身的地址。那么在验证是自己的设备地址后，从机就会发出应答信号，并向主机返回 8bit 数据，发送完之后从机就会等待主机的应答信号。假如主机一直返回应答信号，那么从机可以一直发送数据，也就是图中的 (n byte + 应答信号) 情况，直到主机发出非应答信号，从机才会停止发送数据。

24C02 的数据传输时序是基于 IIC 总线传输时序，下面讲解一下 24C02 的数据传输时序。

35.1.2 24C02 简介

24C02 是一个 2K bit 的串行 EEPROM 存储器，内部含有 256 个字节。在 24C02 里面还有一个 8 字节的页写缓冲器。该设备的通信方式 IIC，通过其 SCL 和 SDA 与其他设备通信，芯片的引脚图如图 35.1.2.1 所示。

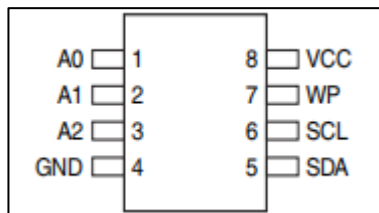


图 35.1.2.1 24C02 引脚图

上图的 WP 引脚是写保护引脚，接高电平只读，接地允许读和写，我们的板子设计是把该引脚接地。每一个设备都有自己的设备地址，24C02 也不例外，但是 24C02 的设备地址是包括不可编程部分和可编程部分，可编程部分是根据上图的硬件引脚 A0、A1 和 A2 所决定。设备地址最后一位用于设置数据的传输方向，即读操作/写操作，0 是写操作，1 是读操作，具体格式如下图 35.1.2.2 所示：

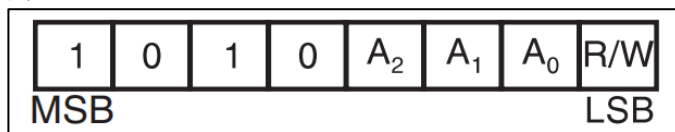


图 35.1.2.2 24C02 设备地址格式图

根据我们的板子设计，A0、A1 和 A2 均接地处理，所以 24C02 设备的读操作地址为：0xA1；写操作地址为：0xA0。

在前面已经说过 IIC 总线的基本读写操作，那么我们就可以基于 IIC 总线的时序的上，理解 24C02 的数据传输时序。

下面把实验中到的数据传输时序讲解一下，分别是对 24C02 的写时序和读时序。24C02 写时序图见图 35.1.4.1 所示。

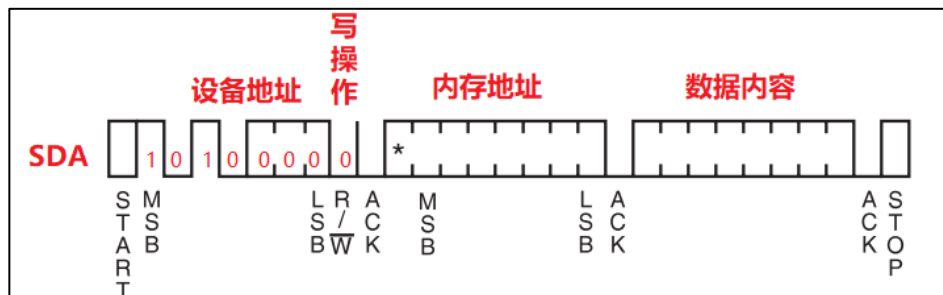


图 35.1.4.1 24C02 写时序图

上图展示的主机向 24C02 写操作时序图，主机在 IIC 总线发送第 1 个字节的数据为 24C02 的设备地址 0xA0，用于寻找总线上找到 24C02，在获得 24C02 的应答信号之后，继续发送第 2 个字节数据，该字节数据是 24C02 的内存地址，再等到 24C02 的应答信号，主机继续发送第 3 字节数据，这里的数据即是写入在第 2 字节内存地址的数据。主机完成写操作后，可以发出停止信号，终止数据传输。

上面的写操作只能单字节写入到 24C02，效率比较低，所以 24C02 有页写入时序，大大提高了写入效率，下面看一下 24C02 页写时序图，图 35.1.4.2 所示。

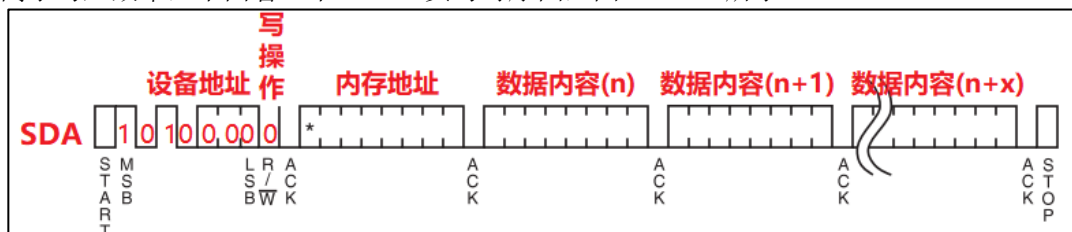


图 35.1.4.2 24C02 页写时序

在单字节写时序时，每次写入数据时都需要先写入设备的内存地址才能实现，在页写时序中，只需要告诉 24C02 第一个内存地址 1，后面数据会按照顺序写入到内存地址 2，内存地址 3 等，大大节省了通信时间，提高了时效性。因为 24C02 每次只能 8bit 数据，所以它的页大小也就是 1 字节。页写时序的操作方式跟上面的单字节写时序差不多，所以不作过多解释了。参考以上说明去理解页写时序。

说完两种写入方式之后，下面看一下图 35.1.2.2 关于 24C02 的读时序。

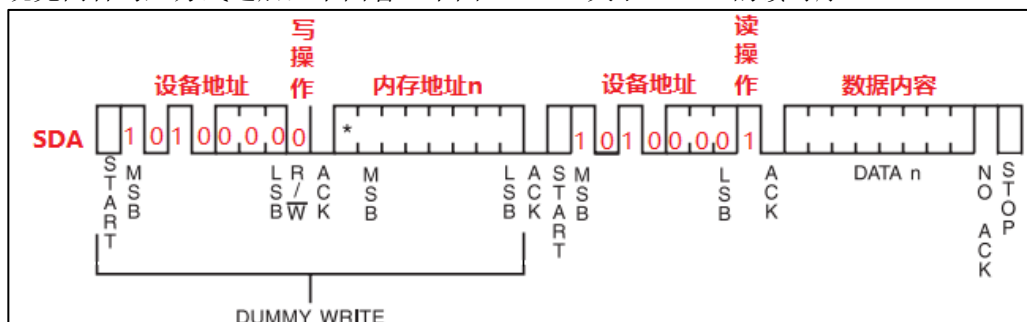


图 35.1.2.2 24C02 读时序图

24C02 读取数据的过程是一个复合的时序，其中包含写时序和读时序。先看第一个通信过程，这里是写时序，起始信号产生后，主机发送 24C02 设备地址 0xA0，获取从机应答信号后，接着发送需要读取的内存地址；在读时序中，起始信号产生后，主机发送 24C02 设备地址 0xA1，获取从机应答信号后，接着从机返回刚刚在写时序中内存地址的数据，以字节为单位传输在总线上，假如主机获取数据后返回的是应答信号，那么从机会一直传输数据，当主机发出的是非应答信号并以停止信号发出为结束，从机就会结束传输。

目前大部分 MCU 都带有 IIC 总线接口，STM32F1 也不例外。但是这里我们不使用 STM32F1 的硬件 IIC 来读写 24C02，而是通过软件模拟。ST 为了规避飞利浦 IIC 专利问题，将 STM32 的硬件 IIC 设计的比较复杂，而且稳定性不怎么好，所以这里我们不推荐使用。有兴趣的读者可以自行研究 STM32F1 的硬件 IIC 的使用。

用软件模拟 IIC，最大的好处就是方便移植，同一个代码兼容所有 MCU，任何一个单片机只要有 IO 口，就可以很快的移植过去，而且不需要特定的 IO 口。而硬件 IIC，则换一款 MCU，基本上就得重新移植，这也是我们推荐使用软件模拟 IIC 的另外一个原因。

35.2 硬件设计

1. 例程功能

每按下 KEY1，MCU 通过 IIC 总线向 24C02 写入数据，通过按下 KEY0 来控制 24C02 读取数据。同时在 LCD 上面显示相关信息。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY1 - PE3
- 3) EEPROM AT24C02
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)(USMART 使用)

3. 原理图

我们主要来看看 24C02 和开发板的连接，如下图所示：

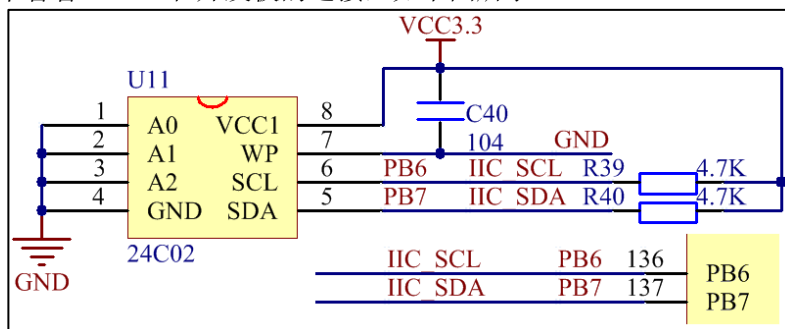


图 35.2.1 IIC 连接原理

24C02 的 SCL 和 SDA 分别连接在 STM32 的 PB6 和 PB7 上。本实验通过软件模拟 IIC 信号建立起与 24C02 的通信，进行数据发送与接收，使用按键 KEY0 和 KEY1 去触发，LCD 屏幕进行显示。

35.3 程序设计

IIC 实验中使用的是软件模拟 IIC，所以用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下使用 IIC 传输数据的配置步骤：

使用 IIC 传输数据的配置步骤

1) 使能 IIC 的 SCL 和 SDA 对应的 GPIO 时钟。

本实验中 IIC 使用的 SCL 和 SDA 分别是 PB6 和 PB7，因此需要先使能 GPIOB 的时钟，代码如下：

```
HAL_RCC_GPIOB_CLK_ENABLE(); /* 使能 GPIOB 时钟 */
```

2) 设置对应 GPIO 工作模式（SCL 推挽输出 SDA 开漏输出）

SDA 线的 GPIO 模式使用开漏输出模式（硬件已接外部上拉电阻，对于 F4 以上板子也可以用内部的上拉电阻），而 SCL 线的 GPIO 模式使用推挽输出模式，通过函数 HAL_GPIO_Init

设置实现。

3) 参考 IIC 总线协议，编写信号函数（起始信号，停止信号，应答信号）

起始信号：SCL 为高电平时，SDA 由高电平向低电平跳变。

停止信号：SCL 为高电平时，SDA 由低电平向高电平跳变。

应答信号：接收到 IC 数据后，向 IC 发出特定的低电平脉冲表示已接收到数据。

4) 编写 IIC 的读写函数

通过参考时序图，在一个时钟周期内发送 1bit 数据或者读取 1bit 数据。读写函数均以一字节数据进行操作。

有了读和写函数，我们就可以对外设进行驱动了。

35.3.1 程序流程图

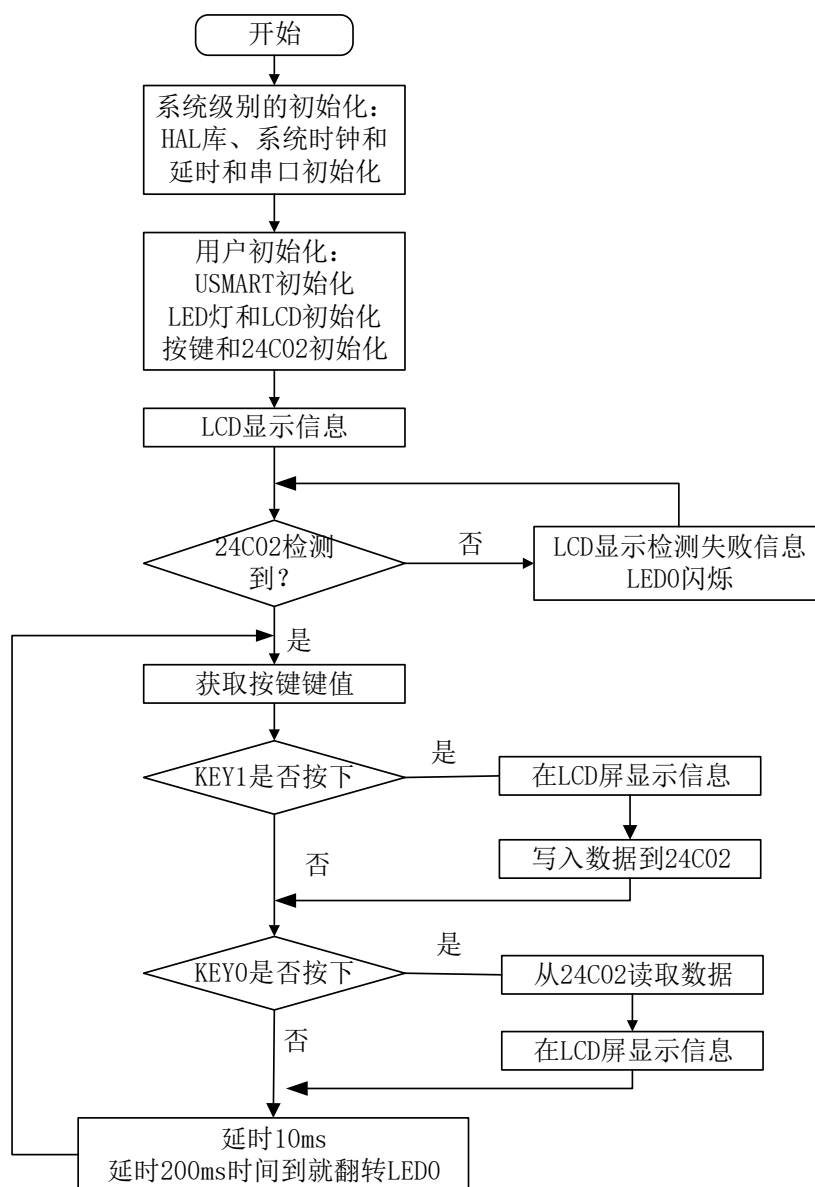


图 35.3.1.1 IIC 实验程序流程图

35.3.2 程序解析

本实验中，我们通过 GPIO 来模拟 IIC，所以不需要在工程分组下添加 HAL 库的 IIC 驱动文件。实验工程中，我们新增了 myiic.c 存放 iic 底层驱动代码，24cxx.c 文件存放 24C02 驱动。

1. IIC 底层驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。IIC 驱动源码包括两个文件：myiic.c 和 myiic.h。

下面我们直接介绍 IIC 相关的程序，首先先介绍 myiic.h 文件，其定义如下：

```
/* 引脚 定义 */
#define IIC_SCL_GPIO_PORT      GPIOB
#define IIC_SCL_GPIO_PIN      GPIO_PIN_6
#define IIC_SCL_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE();}while(0)

#define IIC_SDA_GPIO_PORT      GPIOB
#define IIC_SDA_GPIO_PIN      GPIO_PIN_7
#define IIC_SDA_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE();}while(0)

/* IO 操作 */
#define IIC_SCL(x)              do{ x ? \
                                HAL_GPIO_WritePin(IIC_SCL_GPIO_PORT, IIC_SCL_GPIO_PIN, GPIO_PIN_SET):\
                                HAL_GPIO_WritePin(IIC_SCL_GPIO_PORT, IIC_SCL_GPIO_PIN, GPIO_PIN_RESET);\
                                }while(0) /* SCL */

#define IIC_SDA(x)              do{ x ? \
                                HAL_GPIO_WritePin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN, GPIO_PIN_SET):\
                                HAL_GPIO_WritePin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN, GPIO_PIN_RESET);\
                                }while(0) /* SDA */

/* 读取 SDA */
#define IIC_READ_SDA HAL_GPIO_ReadPin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN)
```

我们通过宏定义标识符的方式去定义 SCL 和 SDA 两个引脚，同时通过宏定义的方式定义了 IIC_SCL() 和 IIC_SDA() 设置这两个管脚可以输出 0 或者 1，主要还是通过 HAL 库的 GPIO 操作函数实现的。另外方便在 iic 操作函数中调用读取 SDA 管脚的数据，这里直接宏定义 IIC_READ_SDA 实现，在后面 iic 模拟信号实现中会频繁调用。

接下来我们看一下 myiic.c 代码中的初始化函数，代码如下：

```
/**
 * @brief      初始化 IIC
 * @param      无
 * @retval     无
 */
void iic_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    IIC_SCL_GPIO_CLK_ENABLE(); /* SCL 引脚时钟使能 */
    IIC_SDA_GPIO_CLK_ENABLE(); /* SDA 引脚时钟使能 */

    gpio_init_struct.Pin = IIC_SCL_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(IIC_SCL_GPIO_PORT, &gpio_init_struct); /* SCL */

    /* SDA 引脚模式设置,开漏输出,上拉,这样就不用再设置 IO 方向了,开漏输出的时候(=1),
       也可以读取外部信号的高低电平 */
    gpio_init_struct.Pin = IIC_SDA_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD; /* 开漏输出 */
    HAL_GPIO_Init(IIC_SDA_GPIO_PORT, &gpio_init_struct); /* SDA */
    iic_stop(); /* 停止总线上所有设备 */
}
```

在 iic_init 函数中主要工作就是对于 GPIO 的初始化，用于 iic 通信，不过这里需要注意的一点是 SDA 线的 GPIO 模式使用开漏模式，同时需要注意：STM32F103 必须要外接上拉电阻！

接下来介绍在上面已经在文字上说明过的 IIC 模拟信号：起始信号、停止信号、应答信号，下面以代码方法实现，大家可以对着图去看代码，有利于理解。

```

/**
 * @brief      IIC 延时函数,用于控制 IIC 读写速度
 * @param      无
 * @retval     无
 */
static void iic_delay(void)
{
    delay_us(2);    /* 2us 的延时, 读写速度在 250Khz 以内 */
}

/**
 * @brief      产生 IIC 起始信号
 * @param      无
 * @retval     无
 */
void iic_start(void)
{
    IIC_SDA(1);
    IIC_SCL(1);
    iic_delay();
    IIC_SDA(0);    /* START 信号: 当 SCL 为高时, SDA 从高变成低, 表示起始信号 */
    iic_delay();
    IIC_SCL(0);    /* 钳住 I2C 总线, 准备发送或接收数据 */
    iic_delay();
}

/**
 * @brief      产生 IIC 停止信号
 * @param      无
 * @retval     无
 */
void iic_stop(void)
{
    IIC_SDA(0);    /* STOP 信号: 当 SCL 为高时, SDA 从低变成高, 表示停止信号 */
    iic_delay();
    IIC_SCL(1);
    iic_delay();
    IIC_SDA(1);    /* 发送 I2C 总线结束信号 */
    iic_delay();
}

```

在这里首先定义一个 `iic_delay` 函数, 目的就是控制 IIC 的读写速度, 通过示波器检测读写速度在 250KHz 内, 所以一秒钟传送 500Kb 数据, 换算一下即一个 bit 位需要 2us, 在这个延时时间内可以让器件获得一个稳定性的数据采集。

为了大家更加清晰了解代码实现的过程, 下面单独把起始信号和停止信号从 iic 总线时序图中抽取出来, 见图 35.2.1.1。

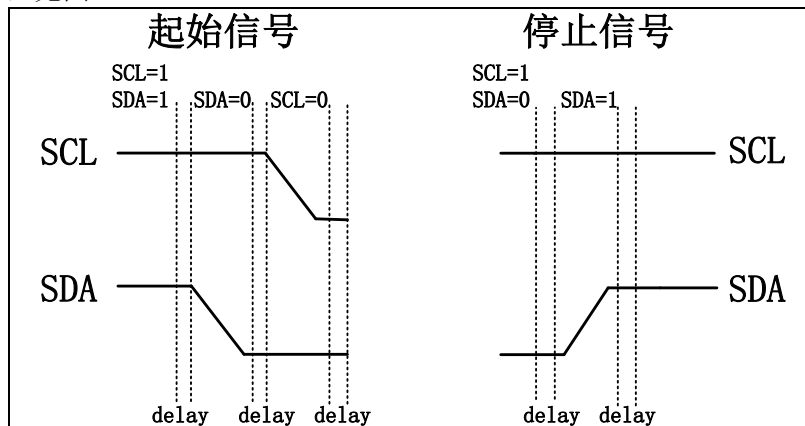


图 35.2.1.1 起始信号与停止信号图

iic_start 函数中，通过调用 myiic.h 中通过宏定义好的可以输出高低电平的 SCL 和 SDA 来模拟 iic 总线中起始信号的发送，在 SCL 时钟线为高电平的时候，SDA 数据线从高电平状态转化到低电平状态，最后拉低时钟线，准备发送或者接收数据。

iic_stop 函数中，也是按着模拟 iic 总线中停止信号的逻辑，在 SCL 时钟线为高电平的时候，SDA 数据线从低电平状态转化到高电平状态。

接下来讲解一下 iic 的发送函数，其定义如下：

```
/**
 * @brief      IIC 发送一个字节
 * @param      data: 要发送的数据
 * @retval     无
 */
void iic_send_byte(uint8_t data)
{
    uint8_t t;

    for (t = 0; t < 8; t++)
    {
        IIC_SDA((data & 0x80) >> 7);    /* 高位先发送 */
        iic_delay();
        IIC_SCL(1);
        iic_delay();
        IIC_SCL(0);
        data <<= 1;    /* 左移 1 位,用于下一次发送 */
    }
    IIC_SDA(1);    /* 发送完成,主机释放 SDA 线 */
}
```

在 iic 的发送函数 iic_send_byte 中，我们把需要发送的数据作为形参，形参大小为 1 个字节。在 iic 总线传输中，一个时钟信号就发送一个 bit，所以该函数需要循环八次，模拟八个时钟信号，才能把形参的 8 个位数据都发送出去。这里使用的是形参 data 和 0x80 与运算的方式，判断其最高位的逻辑值，假如为 1 即需要控制 SDA 输出高电平，否则为 0 控制 SDA 输出低电平。为了更好说明，数据发送的过程，单独拿出数据传输时序图，见图 35.2.1.2。

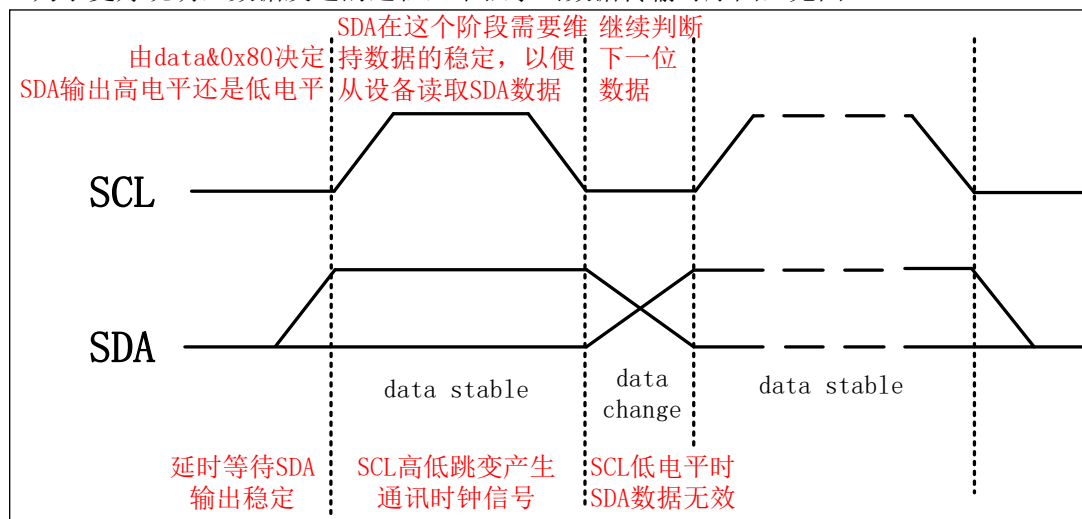


图 35.2.1.2 数据传输时序图

通过上图就可以很清楚了解数据传输时的细节，经过第一步的 SDA 高低电平的确定后，接着需要延时，确保 SDA 输出的电平稳定，在 SCL 保持高电平期间，SDA 线上的数据是有效的，此过程也是需要延时，使得从设备能够采集到有效的电平。然后准备下一位的数据，所以这里需要的是把 data 左移一位，等待下一个时钟的到来，从设备进行读取。把上述的操作重复 8 次就可以把 data 的 8 个位数据发送完毕，循环结束后，把 SDA 线拉高，等待接收从设备发送过来的应答信号。

接着讲解一下 iic 的读取函数 iic_read_byte，它的定义如下：

```
/**
```

```

* @brief      IIC 读取一个字节
* @param      ack:  ack=1 时, 发送 ack; ack=0 时, 发送 nack
* @retval     接收到的数据
*/
uint8_t iic_read_byte(uint8_t ack)
{
    uint8_t i, receive = 0;

    for (i = 0; i < 8; i++) /* 接收 1 个字节数据 */
    {
        receive <<= 1; /* 高位先输出, 所以先收到的数据位要左移 */
        IIC_SCL(1);
        iic_delay();

        if (IIC_READ_SDA)
        {
            receive++;
        }

        IIC_SCL(0);
        iic_delay();
    }

    if (!ack)
    {
        iic_nack(); /* 发送 nACK */
    }
    else
    {
        iic_ack(); /* 发送 ACK */
    }

    return receive;
}

```

`iic_read_byte` 函数具体实现的方式跟 `iic_send_byte` 函数有所不同。首先可以明确的是时钟信号是通过主机发出的, 而且接收到的数据大小为 1 字节, 但是 IIC 传输的单位是 bit, 所以需要执行 8 次循环, 才能把一字节数据接收完整。

具体实现过程: 首先需要有一个变量 `receive` 存放接收到的数据, 在每一次循环开始前都需要对 `receive` 进行左移 1 位操作, 那么 `receive` 的 bit0 位每一次赋值前都是空的, 用来存放最新接收到的数据位, 然后在 SCL 线进行高低电平切换时输出 IIC 时钟, 在 SCL 高电平期间加入延时, 确保有足够的时间能让数据发送并进行处理, 使用宏定义 `IIC_READ_SDA` 就可以判断读取到的高低电平, 假如 SDA 为高电平, 那么 `receive++` 即在 bit0 置 1, 否则不做处理即保持原来的 0 状态。当 SCL 线拉低后, 需要加入延时, 便于从机切换 SDA 线输出数据。在 8 次循环结束后, 我们就获得了 8bit 数据, 把它作为返回值返回, 然而按照时序图, 作为主机就需要发送应答或者非应答信号, 去回复从机。

上面提及到应答信号和非应答信号是在读时序中发生的, 此外在写时序中也存在有一个信号响应, 当发送完 8bit 数据后, 这里是一个等待从机应答信号的操作, 这里我们也定义了, 下面看一下它们的定义:

```

/**
* @brief      等待应答信号到来
* @param      无
* @retval     1, 接收应答失败
*             0, 接收应答成功
*/
uint8_t iic_wait_ack(void)
{
    uint8_t waittime = 0;
    uint8_t rack = 0;

    IIC_SDA(1); /* 主机释放 SDA 线 (此时外部器件可以拉低 SDA 线) */

```

```

iic_delay();
IIC_SCL(1);    /* SCL=1, 此时从机可以返回 ACK */
iic_delay();

while (IIC_READ_SDA)    /* 等待应答 */
{
    waittime++;

    if (waittime > 250)
    {
        iic_stop();
        rack = 1;
        break;
    }
}

IIC_SCL(0);    /* SCL=0, 结束 ACK 检查 */
iic_delay();
return rack;
}

/**
 * @brief      产生 ACK 应答
 * @param      无
 * @retval     无
 */
void iic_ack(void)
{
    IIC_SDA(0);    /* SCL 0 -> 1 时 SDA = 0, 表示应答 */
    iic_delay();
    IIC_SCL(1);    /* 产生一个时钟 */
    iic_delay();
    IIC_SCL(0);
    iic_delay();
    IIC_SDA(1);    /* 主机释放 SDA 线 */
    iic_delay();
}

/**
 * @brief      不产生 ACK 应答
 * @param      无
 * @retval     无
 */
void iic_nack(void)
{
    IIC_SDA(1);    /* SCL 0 -> 1 时 SDA = 1, 表示不应答 */
    iic_delay();
    IIC_SCL(1);    /* 产生一个时钟 */
    iic_delay();
    IIC_SCL(0);
    iic_delay();
}

```

首先先讲解一下 `iic_wait_ack` 函数，该函数主要用在写时序中，当启动起始信号，发送完 8bit 数据到从机时，我们就需要等待以及处理接收从机发送过来的响应信号或者非响应信号，一般就是在 `iic_send_byte` 函数后面调用。

具体实现：首先先释放 SDA，把电平拉高，延时等待从机操作 SDA 线，然后主机拉高时钟线并延时，确保有充足的时间让主机接收到从机发出的 SDA 信号，这里使用的是 `IIC_READ_SDA` 宏定义去读取，根据 IIC 协议，主机读取 SDA 的值为低电平，就表示“应答信号”；读到 SDA 的值为高电平，就表示“非应答信号”。在这个等待读取的过程中加入了超时判断，假如超过这个时间没有接收到数据，那么主机直接发出停止信号，跳出循环，返回等于 1 的变量。在正常等待到应答信号后，主机会把 SCL 时钟线拉低并延时，返回是否接收到应答信

号。

当主机作为接收端时，调用 `iic_read_byte` 函数之后，按照 `iic` 通信协议，需要给从机返回应答或者是非应答信号，这里就是用到了 `iic_ack` 和 `iic_nack` 函数。

具体实现：从上面的说明已经知道了 `SDA` 为低电平即应答信号，高电平即非应答信号，那么还是老规矩，首先根据返回“应答”或者“非应答”两种情况拉低或者拉高 `SDA`，并延时等待 `SDA` 电平稳定，然后主机拉高 `SCL` 线并延时，确保从机能有足够时间去接收 `SDA` 线上的电平信号。然后主机拉低时钟线并延时，完成这一位数据的传送。最后把 `SDA` 拉高，呈高阻态，方便后续通信用到。

2. 24C02 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。24CXX 驱动源码包括两个文件：24cxx.c 和 24cxx.h。

在上一小节已经对 `IIC` 协议中的需要用到的信号都用函数封装好了，那么现在就要定义符合 24C02 时序的函数。为了使代码功能更加健全，所以在 24cxx.h 中宏定义了不同容量大小的 24C 系列型号，具体定义如下：

```
#define AT24C01    127
#define AT24C02    255
#define AT24C04    511
#define AT24C08   1023
#define AT24C16   2047
#define AT24C32   4095
#define AT24C64   8191
#define AT24C128  16383
#define AT24C256  32767
```

```
/* 开发板使用的是 24c02，所以定义 EE_TYPE 为 AT24C02 */
#define EE_TYPE    AT24C02
```

在 24cxx.c 文件中，读/写操作函数对于不同容量大小的 24Cxx 芯片都有相对应的代码块解决处理。下面先看一下 `at24cxx_write_one_byte` 函数，实现在 AT24Cxx 芯片指定地址写入一个数据，代码如下：

```
/**
 * @brief      在 AT24CXX 指定地址写入一个数据
 * @param      addr: 写入数据的目的地址
 * @param      data: 要写入的数据
 * @retval     无
 */
void at24cxx_write_one_byte(uint16_t addr, uint8_t data)
{
    /* 原理说明见:at24cxx_read_one_byte 函数，本函数完全类似 */
    iic_start(); /* 发送起始信号 */

    if (EE_TYPE > AT24C16) /* 24C16 以上的型号，分 2 个字节发送地址 */
    {
        iic_send_byte(0XA0); /* 发送写命令，IIC 规定最低位是 0，表示写入 */
        iic_wait_ack(); /* 每次发送完一个字节，都要等待 ACK */
        iic_send_byte(addr >> 8); /* 发送高字节地址 */
    }
    else
    {
        /* 发送器件 0XA0 + 高位 a8/a9/a10 地址，写数据 */
        iic_send_byte(0XA0 + ((addr >> 8) << 1));
    }

    iic_wait_ack(); /* 每次发送完一个字节，都要等待 ACK */
    iic_send_byte(addr % 256); /* 发送低位地址 */
    iic_wait_ack(); /* 等待 ACK，此时地址发送完成了 */

    /* 因为写数据的时候，不需要进入接收模式了，所以这里不用重新发送起始信号了 */
}
```



```
iic_send_byte(data); /* 发送 1 字节 */
iic_wait_ack(); /* 等待 ACK */
iic_stop(); /* 产生一个停止条件 */
delay_ms(10); /* 注意：EEPROM 写入比较慢, 必须等到 10ms 后再写下一个字节 */
}
```

该函数的操作流程跟前面已经分析过的 24C02 单字节写时序一样, 首先调用 `iic_start` 函数产生起始信号, 然后调用 `iic_send_byte` 函数发送第 1 个字节数据设备地址, 等待 24Cxx 设备返回应答信号; 收到应答信号后, 继续发送第 2 个 1 字节数据内存地址 `addr`; 等待接收应答后, 最后发送第 3 个字节数据写入内存地址的数据 `data`, 24Cxx 设备接收完数据, 返回应答信号, 主机调用 `iic_stop` 函数产生停止信号终止数据传输, 最终需要延时 10ms, 等待 `eeeprom` 写入完毕。

我们的函数兼容 24Cxx 系列多种容量, 就在发送设备地址处做了处理, 这里说一下为什么需要这样子设计。大家请看一下 24Cxx 芯片内存组织表, 见表 35.2.2.1 所示。

| 芯片 | 页数 | 每页字节数 | 总的字节数 | 字节寻址地址线数量 |
|----------|-----|-------|-------|-----------|
| AT24C01A | 16 | 8 | 128 | 7 |
| AT24C02 | 32 | 8 | 256 | 8 |
| AT24C04 | 32 | 16 | 512 | 9 |
| AT24C08A | 64 | 16 | 1024 | 10 |
| AT24C16A | 128 | 16 | 2048 | 11 |
| AT24C32 | 128 | 32 | 4096 | 12 |
| AT24C64A | 256 | 32 | 8192 | 13 |

表 35.2.2.1 24Cxx 芯片内存组织表

主机发送的设备地址和内存地址共同确定了要写入的地方, 这里分析一下 24C16 的使用的是 `iic_send_byte(0XA0+((addr>>8)<<1))` 和 `iic_send_byte(addr % 256)` 确定写入位置, 由于它内存大小一共 2048 字节, 所以只需要定义 11 个寻址地址线, $2048 = 2^{11}$ 。主机下发读写命令的时候带了 3 位, 后面再跟 1 个字节 (8 位) 的地址, 正好 11 位, 就不需要再发后续的地址字节了。

而容量大于 24C16 的芯片, 需要单独发送 2 个字节 (甚至更多) 的地址, 如 24C32, 它的大小为 4096, 需要 12 个寻址地址线支持, $4096 = 2^{12}$ 。24C16 是 2 个字节刚刚好, 而它需要三个字节才能确定写入的位置。24C32 芯片规定设备写地址 0xA0/读地址 0xA1, 后面接着发送 8 位高地址, 最后才发送 8 位低地址。与函数里面的操作是一致的。

接下来看一下 `at24cxx_read_one_byte` 函数, 其定义如下:

```
/**
 * @brief      在 AT24CXX 指定地址读出一个数据
 * @param      readaddr: 开始读数的地址
 * @retval     读到的数据
 */
uint8_t at24cxx_read_one_byte(uint16_t addr)
{
    uint8_t temp = 0;
    iic_start(); /* 发送起始信号 */

    /* 根据不同的 24Cxx 型号, 发送高位地址
     * 1, 24C16 以上的型号, 分 2 个字节发送地址
     * 2, 24C16 及以下的型号, 分 1 个低字节地址 + 占用器件地址的 bit1~bit3 位 用于表示高位地址, 最多 11 位地址
     * 对于 24C01/02, 其器件地址格式 (8bit) 为: 1 0 1 0 A2 A1 A0 R/W
     * 对于 24C04, 其器件地址格式 (8bit) 为: 1 0 1 0 A2 A1 a8 R/W
     * 对于 24C08, 其器件地址格式 (8bit) 为: 1 0 1 0 A2 a9 a8 R/W
     * 对于 24C16, 其器件地址格式 (8bit) 为: 1 0 1 0 a10 a9 a8 R/W
     * R/W : 读/写控制位 0, 表示写; 1, 表示读;
     * A0/A1/A2 : 对应器件的 1, 2, 3 引脚 (只有 24C01/02/04/8 有这些脚)
     * a8/a9/a10: 对应存储整列的高位地址, 11bit 地址最多可以表示 2048 个位置, 可以寻址 24C16 及以内的型号
     */
    if (EE_TYPE > AT24C16) /* 24C16 以上的型号, 分 2 个字节发送地址 */
```

```

{
    iic_send_byte(0XA0);          /* 发送写命令, IIC 规定最低位是 0, 表示写入 */
    iic_wait_ack();              /* 每次发送完一个字节, 都要等待 ACK */
    iic_send_byte(addr >> 8);     /* 发送高字节地址 */
}
else
{ /* 发送器件 0XA0 + 高位 a8/a9/a10 地址, 写数据 */
    iic_send_byte(0XA0 + ((addr >> 8) << 1));
}

iic_wait_ack();                /* 每次发送完一个字节, 都要等待 ACK */
iic_send_byte(addr % 256);      /* 发送低位地址 */
iic_wait_ack();                /* 等待 ACK, 此时地址发送完成了 */

iic_start();                   /* 重新发送起始信号 */
iic_send_byte(0XA1);           /* 进入接收模式, IIC 规定最低位是 0, 表示读取 */
iic_wait_ack();               /* 每次发送完一个字节, 都要等待 ACK */
temp = iic_read_byte(0);       /* 接收一个字节数据 */
iic_stop();                   /* 产生一个停止条件 */
return temp;
}

```

这里的函数的实现跟前面第 35.1.4 小节 24C02 数据传输中的读时序一致, 主机首先调用 iic_start 函数产生起始信号, 然后调用 iic_send_byte 函数发送第 1 个字节数据设备写地址, 使用 iic_wait_ack 函数等待 24Cxx 设备返回应答信号; 收到应答信号后, 继续发送第 2 个 1 字节数据内存地址 addr; 等待接收应答后, 重新调用 iic_start 函数产生起始信号, 这一次的设备方向改变了, 调用 iic_send_byte 函数发送设备读地址, 然后使用 iic_wait_ack 函数去等待设备返回应答信号, 同时使用 iic_read_byte 去读取从从机发出来的数据。由于 iic_read_byte 函数的形参是 0, 所以在获取完 1 个字节的数据后, 主机发送非应答信号, 停止数据传输, 最终调用 iic_stop 函数产生停止信号, 返回从从机 addr 中读取到的数据。

为了方便检测 24Cxx 芯片是否正常工作, 在这里也定义了一个检测函数, 代码如下:

```

/**
 * @brief      检查 AT24CXX 是否正常
 * @note      检测原理: 在器件的末地址写如 0x55, 然后再读取, 如果读取值为 0x55
 *            则表示检测正常. 否则, 则表示检测失败.
 * @param      无
 * @retval     检测结果
 *            0: 检测成功
 *            1: 检测失败
 */
uint8_t at24cxx_check(void)
{
    uint8_t temp;
    uint16_t addr = EE_TYPE;
    temp = at24cxx_read_one_byte(addr); /* 避免每次开机都写 AT24CXX */

    if (temp == 0X55) /* 读取数据正常 */
    {
        return 0;
    }
    else /* 排除第一次初始化的情况 */
    {
        at24cxx_write_one_byte(addr, 0X55); /* 先写入数据 */
        temp = at24cxx_read_one_byte(255); /* 再读取数据 */
        if (temp == 0X55) return 0;
    }
    return 1;
}

```

学到这个地方相信大家, 对于这个操作并不陌生了, 在前面的 RTC 实验也有相似的操作, 可以翻回去看看。这里利用的是 EEPROM 芯片掉电不丢失的特性, 在第一次写入了某个值之

后，再去读一下是否写入成功，这种方式去检测芯片是否正常工作。

此外方便多字节写入和读取，还定义了在规定地址读取指定个数的函数以及在指令地址写入指定个数的函数，代码如下：

```
/**
 * @brief      在 AT24CXX 里面的指定地址开始读出指定个数的数据
 * @param      addr      : 开始读出的地址 对 24c02 为 0~255
 * @param      pbuf      : 数据数组首地址
 * @param      datalen   : 要读出数据的个数
 * @retval     无
 */
void at24cxx_read(uint16_t addr, uint8_t *pbuf, uint16_t datalen)
{
    while (datalen--)
    {
        *pbuf++ = at24cxx_read_one_byte(addr++);
    }
}

/**
 * @brief      在 AT24CXX 里面的指定地址开始写入指定个数的数据
 * @param      addr      : 开始写入的地址 对 24c02 为 0~255
 * @param      pbuf      : 数据数组首地址
 * @param      datalen   : 要写入数据的个数
 * @retval     无
 */
void at24cxx_write(uint16_t addr, uint8_t *pbuf, uint16_t datalen)
{
    while (datalen--)
    {
        at24cxx_write_one_byte(addr, *pbuf);
        addr++;
        pbuf++;
    }
}
```

对于这两个函数都是调用前面的单字节操作函数去实现的，利用 for 循环，连续调用单字节操作函数去实现，这里就不多讲。

3. main.c 代码

在 main.c 里面编写如下代码：

```
const uint8_t g_text_buf[] = {"STM32 IIC TEST"}; /* 要写入到 24c02 的字符串数组 */
#define TEXT_SIZE          sizeof(g_text_buf)    /* TEXT 字符串长度 */

int main(void)
{
    uint8_t key;
    uint16_t i = 0;
    uint8_t datatemp[TEXT_SIZE];

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    at24cxx_init(); /* 初始化 24CXX */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "IIC TEST", RED);
```

```

lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write KEY0:Read", RED);

while (at24cxx_check()) /* 检测不到 24c02 */
{
    lcd_show_string(30, 130, 200, 16, 16, "24C02 Check Failed!", RED);
    delay_ms(500);
    lcd_show_string(30, 130, 200, 16, 16, "Please Check!      ", RED);
    delay_ms(500);
    LED0_TOGGLE(); /* 红灯闪烁 */
}

lcd_show_string(30, 130, 200, 16, 16, "24C02 Ready!", RED);

while (1)
{
    key = key_scan(0);

    if (key == KEY1_PRES) /* KEY1 按下,写入 24C02 */
    {
        lcd_fill(0, 150, 239, 319, WHITE); /* 清除半屏 */
        lcd_show_string(30, 150, 200, 16, 16, "Start Write 24C02....", BLUE);
        at24cxx_write(0, (uint8_t *)g_text_buf, TEXT_SIZE);
        /* 提示传送完成 */
        lcd_show_string(30, 150, 200, 16, 16, "24C02 Write Finished!", BLUE);
    }

    if (key == KEY0_PRES) /* KEY0 按下,读取字符串并显示 */
    {
        lcd_show_string(30, 150, 200, 16, 16, "Start Read 24C02.... ", BLUE);
        at24cxx_read(0, datatemp, TEXT_SIZE);
        /* 提示传送完成 */
        lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is: ", BLUE);
        /* 显示读到的字符串 */
        lcd_show_string(30, 170, 200, 16, 16, (char *)datatemp, BLUE);
    }

    i++;

    if (i == 20)
    {
        LED0_TOGGLE(); /* 红灯闪烁 */
        i = 0;
    }

    delay_ms(10);
}
}

```

main 函数的流程大致是：在 main 函数外部定义要写入 24C02 的字符串数组 g_text_buf。在完成系统级和用户级初始化工作后，检测 24c02 是否存在，然后通过 KEY0 去读取 0 地址存放的数据并把数据显示在 LCD 上；另外还可以通过 KEY1 去 0 地址处写入 g_text_buf 数据并在 LCD 界面中显示传输中，完成后并显示“24C02 Write Finished!”。

35.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了，先按下 KEY1 写入数据，然后再按 KEY0 读取数据，最终 LCD 显示的内容如图 35.4.1 所示：

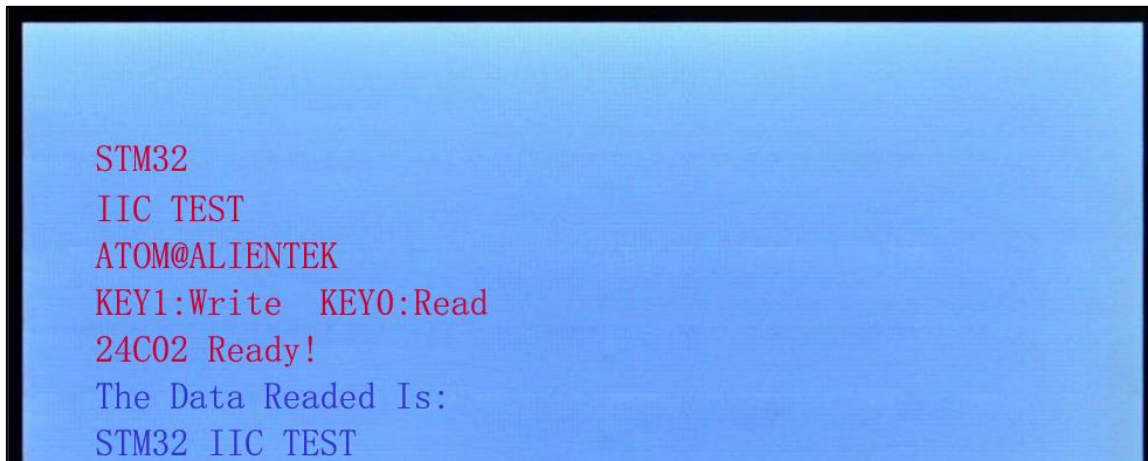


图 35.4.1 IIC 实验程序运行效果图

假如大家需要验证 24C02 的自检函数，可以用跟杜邦线把 PB6 和 PB7 短接，重新上电看看是否能看到报错。

该实验还支持 USMART，在这里我们可以方便测试 24C02 的读写功能，可以操作 24C02 的任意地址，不过在 0~255 这个范围，读写测试图如图 35.4.2 所示。

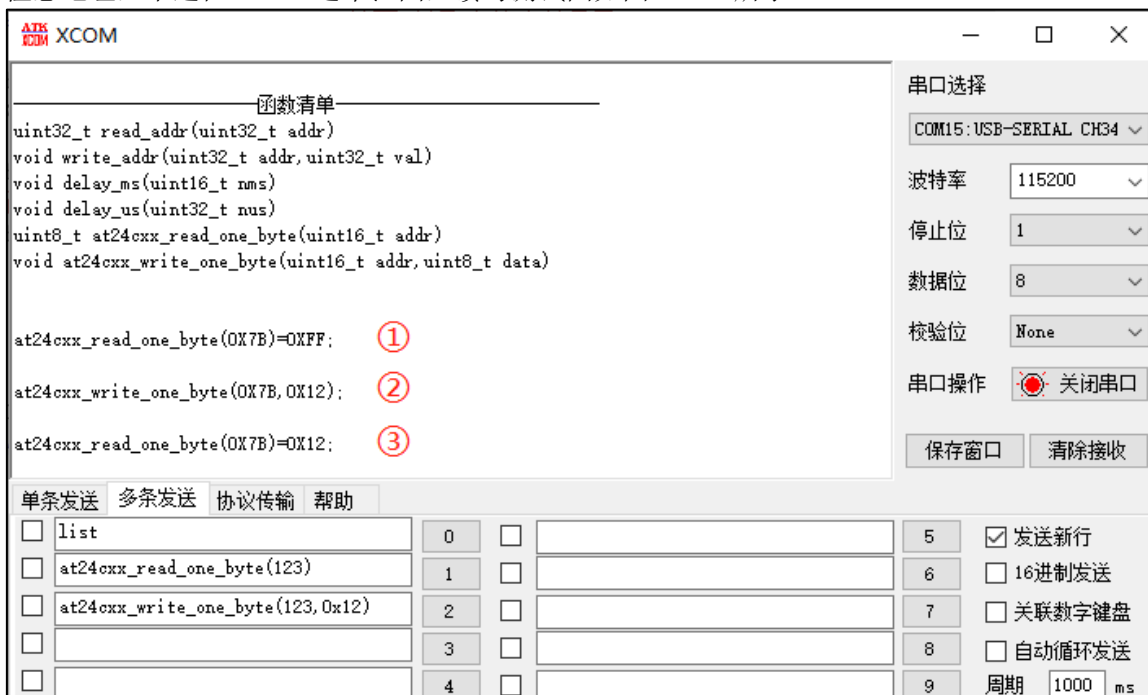


图 35.4.2 24C02 读写测试图

图中，我们首先调用 at24cxx_read_one_byte 函数在 123 地址处读取，获取的数据为 0xFF。通过调用 at24cxx_write_one_byte 函数在 123 地址处写入 0x12 的值，然后继续调用 at24cxx_read_one_byte 函数在 123 地址处读取，获得 0x12 的值，表明实验成功。

至此，我们整个 IIC 实验就结束了，本章内容比较多，需要大家花多点时间去理解，一定要自己去用一下 IIC 通信协议。市面上很多器件都是具有 IIC 通信接口的，可以尝试去驱动它们，这样才能学以致用。

第三十六章 SPI 实验

本章，我们将介绍如何使用 STM32F103 的 SPI 功能，并实现对外部 NOR FLASH 的读写并把结果显示在 TFTLCD 模块上。

本章分为如下几个小节：

36.1 SPI 及 NOR FLASH 芯片介绍

36.2 硬件设计

36.3 程序设计

36.4 下载验证

36.1 SPI 及 NOR FLASH 介绍

36.1.1 SPI 介绍

我们将从结构、时序和寄存器三个部分来介绍 SPI。

36.1.1.1 SPI 框图

SPI 是英语 Serial Peripheral interface 缩写，顾名思义就是串行外围设备接口。SPI 通信协议是 Motorola 公司首先在其 MC68HCXX 系列处理器上定义的。SPI 接口是一种高速的全双工同步的通信总线，已经广泛应用在众多 MCU、存储芯片、AD 转换器和 LCD 之间。大部分 STM32 是有 3 个 SPI 接口，本实验使用的是 SPI2。

我们先看 SPI 的结构框图，了解它的大致功能，如图 36.1.1.1 所示。

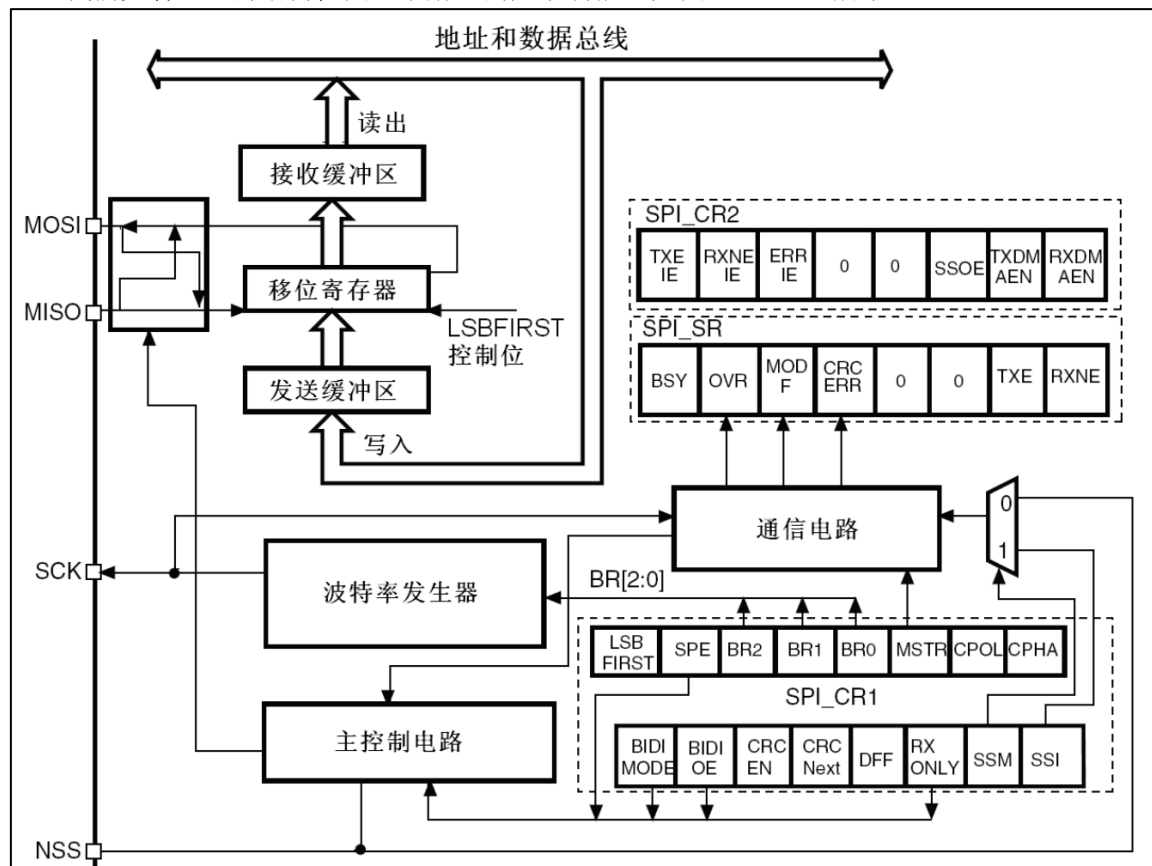


图 36.1.1.1 SPI 框图

围绕框图，我们展开介绍一下 SPI 的引脚信息、工作原理以及传输方式，把 SPI 的 4 种工

作方式放在后面讲解。

SPI 的引脚信息:

MISO (Master In / Slave Out) 主设备数据输入, 从设备数据输出。

MOSI (Master Out / Slave In) 主设备数据输出, 从设备数据输入。

SCLK (Serial Clock) 时钟信号, 由主设备产生。

CS (Chip Select) 从设备片选信号, 由主设备产生。

SPI 的工作原理: 在主机和从机都有一个串行移位寄存器, 主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。串行移位寄存器通过 MOSI 信号线将字节传送给从机, 从机也将自己的串行移位寄存器中的内容通过 MISO 信号线返回给主机。这样, 两个移位寄存器中的内容就被交换。外设的写操作和读操作是同步完成的。如果只是进行写操作, 主机只需忽略接收到的字节。反之, 若主机要读取从机的一个字节, 就必须发送一个空字节引发从机传输。

SPI 的传输方式: SPI 总线具有三种传输方式: 全双工、单工以及半双工传输方式。

全双工通信, 就是在任何时刻, 主机与从机之间都可以同时进行数据的发送和接收。

单工通信, 就是在同一时刻, 只有一个传输的方向, 发送或者是接收。

半双工通信, 就是在同一时刻, 只能为一个方向传输数据。

36.1.1.2 SPI 工作模式

STM32 要与具有 SPI 接口的器件进行通信, 就必须遵循 SPI 的通信协议。每一种通信协议都有各自的读写数据时序, 当然 SPI 也不例外。SPI 通信协议就具备 4 种工作模式, 在讲这 4 种工作模式前, 首先先知道两个单词 CPOL 和 CPHA。

CPOL, 详称 Clock Polarity, 就是时钟极性, 当主从机没有数据传输的时候即空闲状态, SCL 线的电平状态, 假如空闲状态是高电平, CPOL=1; 若空闲状态时低电平, 那么 CPOL=0。

CPHA, 详称 Clock Phase, 就是时钟相位。在这里先科普一下数据传输的常识: 同步通信时, 数据的变化和采样都是在时钟边沿上进行的, 每一个时钟周期都会有上升沿和下降沿两个边沿, 那么数据的变化和采样就分别安排在两个不同的边沿, 由于数据在产生和到它稳定是需要一定的时间, 那么假如我们在第 1 个边沿信号把数据输出了, 从机只能从第 2 个边沿信号去采样这个数据。

CPHA 实质指的是数据的采样时刻, CPHA=0 的情况就表示数据的采样是从第 1 个边沿信号上即奇数边沿, 具体是上升沿还是下降沿的问题, 是由 CPOL 决定的。这里就存在一个问题: 当开始传输第一个 bit 的时候, 第 1 个时钟边沿就采集该数据了, 那数据是什么时候输出来的呢? 那么就有两种情况: 一是 CS 使能的边沿, 二是上一帧数据的最后一个时钟沿。

CPHA = 1 的情况就是表示数据采样是从第 2 个边沿即偶数边沿, 它的边沿极性要注意一点, 不是和上面 CPHA=0 一样的边沿情况。前面的是奇数边沿采样数据, 从 SCL 空闲状态的直接跳变, 空闲状态是高电平, 那么它就是下降沿, 反之就是上升沿。由于 CPHA=1 是偶数边沿采样, 所以需要根据偶数边沿判断, 假如第一个边沿即奇数边沿是下降沿, 那么偶数边沿的边沿极性就是上升沿。不理解的, 可以看一下下面 4 种 SPI 工作模式的图。

由于 CPOL 和 CPHA 都有两种不同状态, 所以 SPI 分成了 4 种模式。我们在开发的时候, 使用比较多的是模式 0 和模式 3。下面请看表 36.1.1.2.1 SPI 工作模式表。

| SPI 工作模式 | CPOL | CPHA | SCL 空闲状态 | 采样边沿 | 采样时刻 |
|----------|------|------|----------|------|------|
| 0 | 0 | 0 | 低电平 | 上升沿 | 奇数边沿 |
| 1 | 0 | 1 | 低电平 | 下降沿 | 偶数边沿 |
| 2 | 1 | 0 | 高电平 | 下降沿 | 奇数边沿 |
| 3 | 1 | 1 | 高电平 | 上升沿 | 偶数边沿 |

表 36.1.1.2.1 SPI 工作模式表

下面分别对 SPI 的 4 种工作模式进行分析:

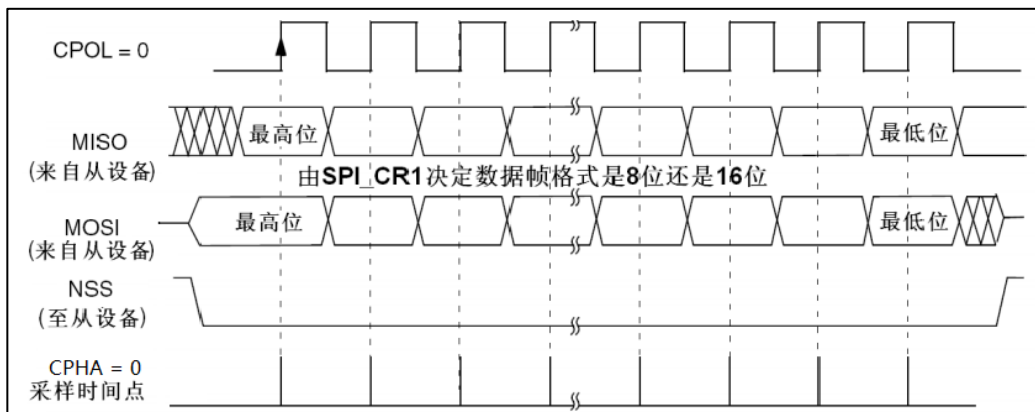


图 36.1.1.2.1 串行时钟的奇数边沿上升沿采样时序图

我们分析一下 $CPOL=0$ 且 $CPHA=0$ 的时序，图 36.1.1.2.1 就是串行时钟的奇数边沿上升沿采样的情况，首先由于配置了 $CPOL=0$ ，可以看到当数据未发送或者发送完毕，SCK 的状态是低电平，再者 $CPHA=0$ 即是奇数边沿采集。所以传输的数据会在奇数边沿上升沿被采集，MOSI 和 MISO 数据的有效信号需要在 SCK 奇数边沿保持稳定且被采样，在非采样时刻，MOSI 和 MISO 的有效信号才发生变化。

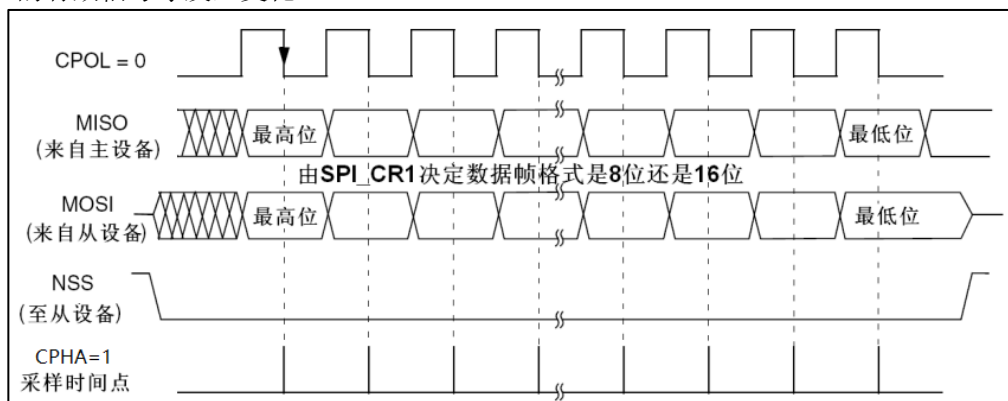


图 36.1.1.2.2 串行时钟的偶数边沿下降沿采样图

现在分析一下 $CPOL=0$ 且 $CPHA=1$ 的时序，图 36.1.1.2.2 是串行时钟的偶数边沿下降沿采样的情况。由于 $CPOL=0$ ，所以 SCK 的空闲状态依然是低电平， $CPHA=1$ 数据就从偶数边沿采样，至于是上升沿还是下降沿，从上图就可以知道，是下降沿。这里有一个误区，空闲状态是低电平的情况下，不是应该上升沿吗，为什么这里是下降沿？首先我们先明确这里是偶数边沿采样，那么看图就很清晰，SCK 低电平空闲状态下，上升沿是在奇数边沿上，下降沿是在偶数边沿上。

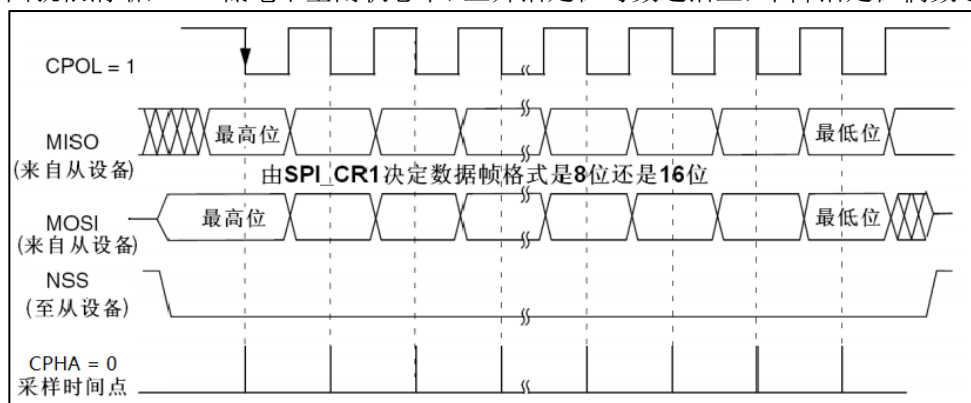


图 36.1.1.2.3 串行时钟的奇数边沿下降沿采样图

图 36.1.1.2.3 这种情况和第一种情况相似，只是这里是 $CPOL=1$ ，即 SCK 空闲状态为高电平，在 $CPHA=0$ ，奇数边沿采样的情况下，数据在奇数边沿下降沿要保持稳定并等待采样。

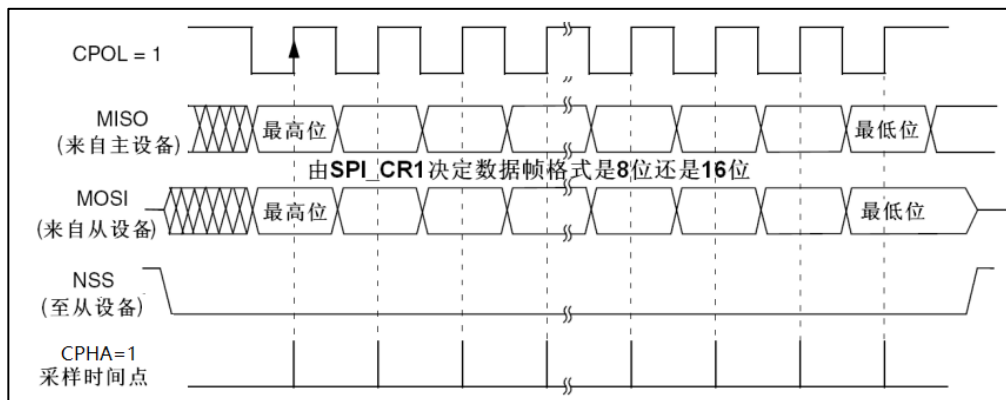


图 36.1.1.2.4 串行时钟的偶数边沿上升沿采样图

图 36.1.1.2.4 是 CPOL=1&&CPHA=1 的情形，可以看到未发送数据和发送数据完毕，SCL 的状态是高电平，奇数边沿的边沿极性是上升沿，偶数边沿的边沿极性是下降沿。因为 CPHA=1，所以数据在偶数边沿上升沿被采样。在奇数边沿的时候 MOSI 和 MISO 会发生变化，在偶数边沿时候是稳定的。

36.1.1.3 SPI 寄存器

在这里我们简单介绍一下本实验用到的寄存器。

● SPI 控制寄存器 1 (SPI_CR1)

SPI 控制寄存器 1 描述如图 36.1.1.3.1 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--------------|------------|---|-----|----|----------------|-----|-----|--------------|-----|---------|----|----|------|------|------|
| BIDI MODE | BIDI OE | CRCE NEXT | CRC | DF | R X ONLY | SSM | SSI | LSB FIRST | SPE | BR[2:0] | | | MSTR | CPOL | CPHA |
| RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| 位11 | | DF : 数据帧格式 (Data frame format) 0: 使用8位数据帧格式进行发送/接收; 1: 使用16位数据帧格式进行发送/接收。 注: 只有当SPI禁止(SPE=0)时, 才能写该位, 否则出错。 注: I ² S模式下不使用。 | | | | | | | | | | | | | |
| 位10 | | RXONLY : 只接收 (Receive only) 该位和BIDIMODE位一起决定在“双线双向”模式下的传输方向。在多个从设备的配置中, 在未访问的从设备上该位被置1, 使得只有被访问的从设备有输出, 从而不会造成数据线上数据冲突。注: I ² S模式下不使用。 0: 全双工(发送和接收); 1: 禁止输出(只接收模式)。 | | | | | | | | | | | | | |
| 位9 | | SSM : 软件从设备管理 (Software slave management) 当SSM被置位时, NSS引脚上的电平由SSI位的值决定。注: I ² S模式下不使用。 0: 禁止软件从设备管理; 1: 启用软件从设备管理。 | | | | | | | | | | | | | |
| 位7 | | LSBFIRST : 帧格式 (Frame format) 0: 先发送MSB; 1: 先发送LSB。 注: 当通信在进行时不能改变该位的值。 注: I ² S模式下不使用。 | | | | | | | | | | | | | |
| 位6 | | SPE : SPI使能 (SPI enable) 0: 禁止SPI设备; 1: 开启SPI设备。 注: I ² S模式下不使用。 注: 当关闭SPI设备时, 请按照第23.3.8节的过程操作。 | | | | | | | | | | | | | |
| 位5:3 | | BR[2:0] : 波特率控制 (Baud rate control) 000: f _{PCLK} /2 001: f _{PCLK} /4 010: f _{PCLK} /8 011: f _{PCLK} /16 100: f _{PCLK} /32 101: f _{PCLK} /64 110: f _{PCLK} /128 111: f _{PCLK} /256 当通信正在进行的时候, 不能修改这些位。 注意: I ² S模式下不使用。 | | | | | | | | | | | | | |
| 位2 | | MSTR : 主设备选择 (Master selection) 0: 配置为从设备; 1: 配置为主设备。 注: 当通信正在进行的时候, 不能修改该位。 注: I ² S模式下不使用。 | | | | | | | | | | | | | |
| 位1 | | CPOL : 时钟极性 (Clock polarity) 0: 空闲状态时, SCK保持低电平; 1: 空闲状态时, SCK保持高电平。 注: 当通信正在进行的时候, 不能修改该位。 注: I ² S模式下不使用。 | | | | | | | | | | | | | |
| 位0 | | CPHA : 时钟相位 (Clock phase) 0: 数据采样从第一个时钟边沿开始; 1: 数据采样从第二个时钟边沿开始。 注: 当通信正在进行的时候, 不能修改该位。 注: I ² S模式下不使用。 | | | | | | | | | | | | | |

图 36.1.1.3.1 SPI_CR1 寄存器 (部分)

该寄存器控制着 SPI 很多相关信息，包括主设备模式选择，传输方向，数据格式，时钟极性、时钟相位和使能等。下面讲解一下本实验配置的位，位 CPHA 置 1，数据采样从第二个时钟边沿开始；位 CPOL 置 1，在空闲状态时，SCK 保持高电平；位 MSTR 置 1，配置为主设备；在位 BR[2:0]置 7，使用 256 分频，速度最低；位 SPE 置 1，开启 SPI 设备；位 LSBFIRST 置 0，MSB 先传输；位 SSI 置 1，禁止软件从设备，即做主机；位 SSM 置 1，软件片选 NSS 控制；位 RXONLY 置 0，传输方式采用的是全双工模式；位 DFF 置 0，使用 8 位数据帧格式。

● SPI 状态寄存器 (SPI_SR)

SPI 状态寄存器描述如图 36.1.1.3.2 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|--|----|----|----|---|---|-----|-----|------|---------|-----|--------|-----|------|
| 保留 | | | | | | | | BSY | OVR | MODF | CRC ERR | UDR | CHSIDE | TXE | RXNE |
| res | | | | | | | | r | r | r | rc w0 | r | r | r | r |
| 位1 | | TXE: 发送缓冲为空 (Transmit buffer empty) 0: 发送缓冲非空; 1: 发送缓冲为空。 | | | | | | | | | | | | | |
| 位0 | | RXNE: 接收缓冲非空 (Receive buffer not empty) 0: 接收缓冲为空; 1: 接收缓冲非空。 | | | | | | | | | | | | | |

图 36.1.1.3.2 SPI_SR 寄存器（部分）

该寄存器是查询当前 SPI 的状态的，我们在实验中用到的是 TXE 位和 RXNE 位，即发送完成和接收完成是否的标记。

● SPI 数据寄存器 (SPI_DR)

SPI 数据寄存器描述如图 36.1.4.3 所示：

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|--|----|----|----|---|---|---|---|---|---|---|---|---|---|
| DR[15:0] | | | | | | | | | | | | | | | |
| rW rW rW rW rW rW rW rW rW rW rW rW rW rW rW rW | | | | | | | | | | | | | | | |
| 位15:0 | | DR[15:0]: 数据寄存器 (Data register) 待发送或者已经收到的数据 数据寄存器对应两个缓冲区：一个用于写(发送缓冲)；另外一个用于读(接收缓冲)。写操作将数据写到发送缓冲区；读操作将返回接收缓冲区里的数据。 对SPI模式的注释： 根据SPI_CR1的DFF位对数据帧格式的选择，数据的发送和接收可以是8位或者16位的。为保证正确的操作，需要在启用SPI之前就确定好数据帧格式。 对于8位的数据，缓冲器是8位的，发送和接收时只会用到SPI_DR[7:0]。在接收时，SPI_DR[15:8]被强制为0。 对于16位的数据，缓冲器是16位的，发送和接收时会用到整个数据寄存器，即SPI_DR[15:0]。 | | | | | | | | | | | | | |

图 36.1.1.3.2 SPI_DR 寄存器

该寄存器是 SPI 数据寄存器，是一个双寄存器，包括了发送缓存和接收缓存。当向该寄存器写数据的时候，SPI 就会自动发送，当收到数据的时候，也是存在该寄存器内。

36.1.2 NOR FLASH 简介

36.1.2.1 FLASH 简介

FLASH 是常见的用于存储数据的半导体器件，它具有容量大、可重复擦写、按“扇区/块”擦除、掉电后数据可继续保存的特性。常见的 FLASH 主要有 NOR FLASH 和 NAND FLASH 两种类型，它们的特性如表 36.1.2.1.1 所示。NOR 和 NAND 是两种数字门电路，可以简单地认为 Flash 内部存储单元使用哪种门作存储单元就是哪类型的 FLASH。U 盘，SSD，eMMC 等为 NAND 型，而 NOR FLASH 则根据设计需要灵活应用于各类 PCB 上，如 BIOS，手机等。

| 特性 | NOR FLASH | NAND FLASH |
|----------|-----------|------------|
| 容量 | 较小 | 很大 |
| 同容量存储器成本 | 较贵 | 较便宜 |

| | | |
|---------|-----------|-------------|
| 擦除单元 | 以“扇区/块”擦除 | 以“扇区/块”擦除 |
| 读写单元 | 可以基于字节读写 | 必须以“块”为单位读写 |
| 读取速度 | 较高 | 较低 |
| 写入速度 | 较低 | 较高 |
| 集成度 | 较低 | 较高 |
| 介质类型 | 随机存储 | 连续存储 |
| 地址线和数据线 | 独立分开 | 共用 |
| 坏块 | 较少 | 较多 |
| 是否支持XIP | 支持 | 不支持 |

表 36.1.2.1.1 NOR FLASH 和 NAND FLASH 特性对比

NOR 与 NAND 在数据**写入前都需要有擦除操作**，但实际上 NOR FLASH 的一个 bit 可以从 1 变成 0，而要从 0 变 1 就要擦除后再写入，NAND Flash 这两种情况都需要擦除。擦除操作的最小单位为“扇区/块”，这意味着有时候即使只写一字节的数据，则这个“扇区/块”上之前的数据都可能会被擦除。

NOR 的地址线和数据线分开，它可以按“字节”读写数据，符合 CPU 的指令译码执行要求，所以假如 NOR 上存储了代码指令，CPU 给 NOR 一个地址，NOR 就能向 CPU 返回一个数据让 CPU 执行，中间不需要额外的处理操作，这体现于表 35.1.2.1.1 中的支持 XIP 特性(eXecute In Place)。因此可以用 NOR FLASH 直接作为嵌入式 MCU 的程序存储空间。

NAND 的数据和地址线共用，只能按“块”来读写数据，假如 NAND 上存储了代码指令，CPU 给 NAND 地址后，它无法直接返回该地址的数据，所以不符合指令译码要求。

若代码存储在 NAND 上，可以把它先加载到 RAM 存储器上，再由 CPU 执行。所以在功能上可以认为 NOR 是一种断电后数据不丢失的 RAM，但它的擦除单位与 RAM 有区别，且读写速度比 RAM 要慢得多。

FLASH 也有对应的缺点，我们在使用过程中需要尽量去规避这些问题：一是 FLASH 的使用寿命，另一个是可能的位反转。

使用寿命体现在：读写上是 FLASH 的擦除次数都是有限的(NOR FLASH 普遍是 10 万次左右)，当它的使用接近寿命的时候，可能会出现写操作失败。由于 NAND 通常是整块擦写，块内有一位失效整个块就会失效，这被称为坏块。使用 NAND Flash 最好通过算法扫描介质找出坏块并标记为不可用，因为坏块上的数据是不准确的。

位反转是数据位写入时为 1，但经过一定时间的环境变化后可能实际变为 0 的情况，反之亦然。位反转的原因很多，可能是器件特性也可能与环境、干扰有关，由于位反转的问题可能存在，所以 FLASH 存储器需要“探测/错误更正(EDC/ECC)”算法来确保数据的正确性。

FLASH 芯片有很多种芯片型号，在我们的 norflash.h 头文件中有定义芯片 ID 的宏定义，对应的就是不同型号的 NOR FLASH 芯片，比如有：W25Q128、BY25Q128、NM25Q128，它们是来自不同的厂商的同种规格的 NOR FLASH 芯片，内存空间都是 128M 字，即 16M 字节。它们的很多参数、操作都是一样的，所以我们的实验都是兼容它们的。

由于这么多的芯片，我们就不一一进行介绍了，就拿其中一款型号进行介绍即可，其他的型号都是类似的。

下面我们以 NM25Q128 为例，认识一下具体的 NOR Flash 的特性。

NM25Q128 是一款大容量 SPI FLASH 产品，其容量为 16M。它将 16M 字节的容量分为 256 个块(Block)，每个块大小为 64K 字节，每个块又分为 16 个扇区(Sector)，每一个扇区 16 页，每页 256 个字节，即每个扇区 4K 个字节。NM25Q128 的最小擦除单位为一个扇区，也就是每次必须擦除 4K 个字节。这样我们需要给 NM25Q128 开辟一个至少 4K 的缓存区，这样对 SRAM 要求比较高，要求芯片必须有 4K 以上 SRAM 才能很好的操作。

NM25Q128 的擦写周期多达 10W 次，具有 20 年的数据保存期限，支持电压为 2.7~3.6V，NM25Q128 支持标准的 SPI，还支持双输出/四输出的 SPI，最大 SPI 时钟可以到 104Mhz（双输出时相当于 208Mhz，四输出时相当于 416Mhz）。

下面我们看一下 NM25Q128 芯片的管脚图，如图 36.1.2.1.2 所示。

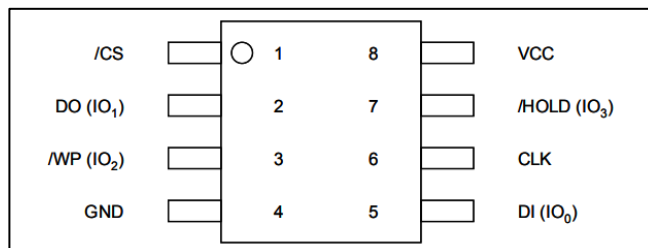


图 36.1.2.1.2 NM25Q128 芯片引脚图

芯片引脚连接如下：CS 即片选信号输入，低电平有效；DO 是 MISO 引脚，在 CLK 管脚的下降沿输出数据；WP 是写保护管脚，高电平可读可写，低电平仅仅可读；DI 是 MOSI 引脚，主机发送的数据、地址和命令从 SI 引脚输入到芯片内部，在 CLK 管脚的上升沿捕获捕获数据；CLK 是串行时钟引脚，为输入输出提供时钟脉冲；HOLD 是保持管脚，低电平有效。

STM32F103 通过 SPI 总线连接到 NM25Q128 对应的引脚即可启动数据传输。

36.1.2.2 NOR FLASH 工作时序

前面对于 NM25Q128 的介绍中也提及存储的体系，NM25Q128 有写入、读取还有擦除的功能，下面就对这三种操作的时序进行分析，在后面通过代码的形式驱动它。

下面先让我们看一下读操作时序，如图 36.1.2.1 所示：

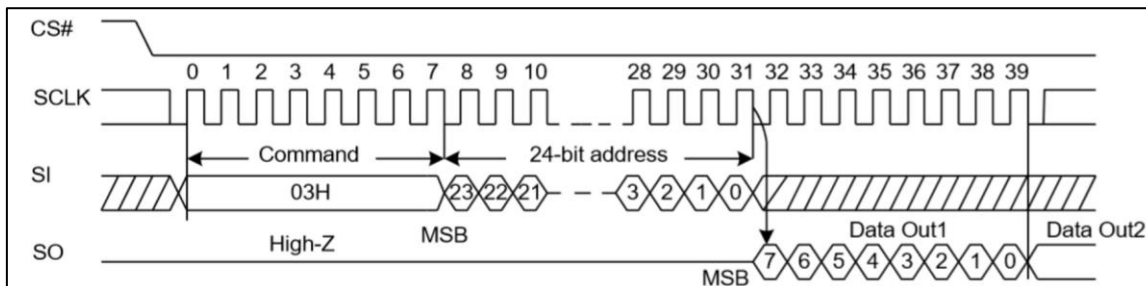


图 36.1.2.2.1 NM25Q128 读操作时序图

从上图可知读数据指令是 03H，可以读出一个字节或者多个字节。发起读操作时，先把 CS 片选管脚拉低，然后通过 MOSI 引脚把 03H 发送芯片，之后再发送要读取的 24 位地址，这些数据在 CLK 上升沿时采样。芯片接收完 24 位地址之后，就会把相对应地址的数据在 CLK 引脚下降沿从 MISO 引脚发送出去。从图中可以看出只要 CLK 一直在工作，那么通过一条读指令就可以把整个芯片存储区的数据读出来。当主机把 CS 引脚拉高，数据传输停止。

接着我们看一下写时序，这里我们先看页写时序，如图 36.1.2.2.2 所示：

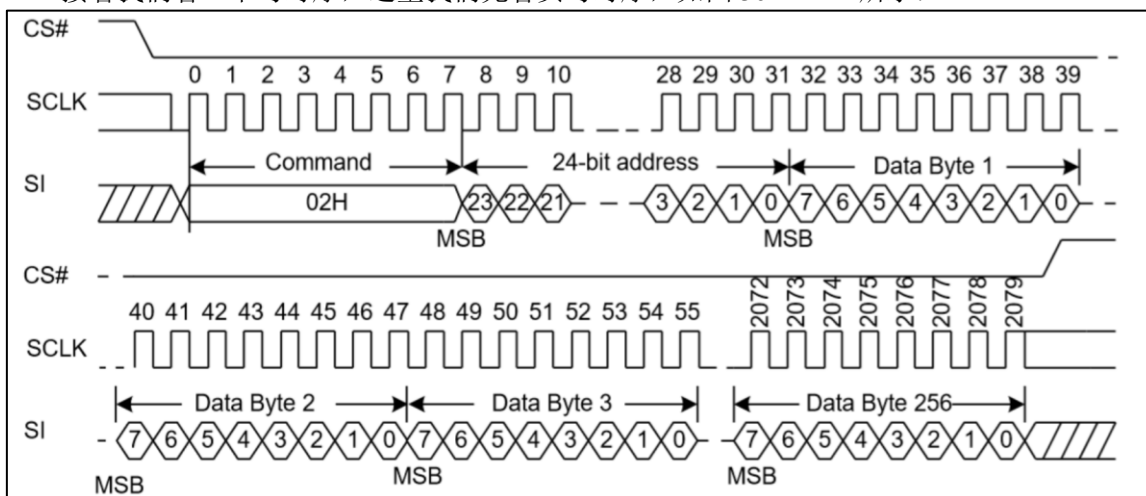


图 36.1.2.2.2 NM25Q128 页写时序

在发送页写指令之前，需要先发送“写使能”指令。然后主机拉低 CS 引脚，然后通过 MOSI 引脚把 02H 发送到芯片，接着发送 24 位地址，最后你就可以发送你需要写的字节数据到芯片。

完成数据写入之后，需要拉高 CS 引脚，停止数据传输。

下面介绍一下扇区擦除时序，如图 36.1.2.2.3 所示：

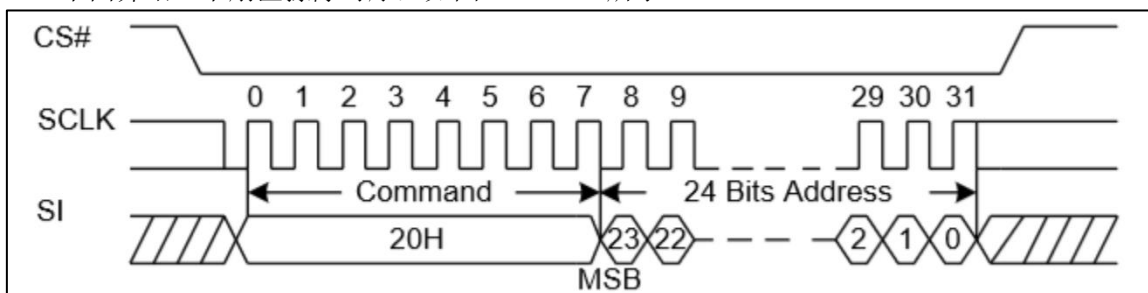


图 36.1.2.2.3 扇区擦除时序图

扇区擦除指的是将一个扇区擦除，通过前面的介绍也知道，NM25Q128 的扇区大小是 4K 字节。擦除扇区后，扇区的位全置 1，即扇区字节为 FFh。同样的，在执行扇区擦除之前，需要先执行写使能指令。这里需要注意的是当前 SPI 总线的状态，假如总线状态是 BUSY，那么这个扇区擦除是无效的，所以在拉低 CS 引脚准备发送数据前，需要先要确定 SPI 总线的状态，这就需要执行读状态寄存器指令，读取状态寄存器的 BUSY 位，需要等待 BUSY 位为 0，才可以执行擦除工作。

接着按时序图分析，主机先拉低 CS 引脚，然后通过 MOSI 引脚发送指令代码 20h 到芯片，然后接着把 24 位扇区地址发送到芯片，然后需要拉高 CS 引脚，通过读取寄存器状态等待扇区擦除操作完成。

此外还有对整个芯片进行擦除的操作，时序比扇区擦除更加简单，不用发送 24bit 地址，只需要发送指令代码 C7h 到芯片即可实现芯片的擦除。

在 NM25Q128 手册中还有许多种方式的读/写/擦除操作，我们这里只分析本实验用到的，其他大家可以参考 NM25Q128 手册。

36.2 硬件设计

1. 例程功能

通过 KEY1 按键来控制 norflash 的写入，通过按键 KEY0 来控制 norflash 的读取。并在 LCD 模块上显示相关信息。我们还可以通过 USMART 控制读取 norflash 的 ID、擦除某个扇区或整片擦除。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 – PB5
- 2) 独立按键
KEY0 – PE4
KEY1 – PE3
- 3) NOR FLASH NM25Q128
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面) (USMART 使用)

3. 原理图

我们主要来看看 norflash 和开发板的连接，如下图所示：

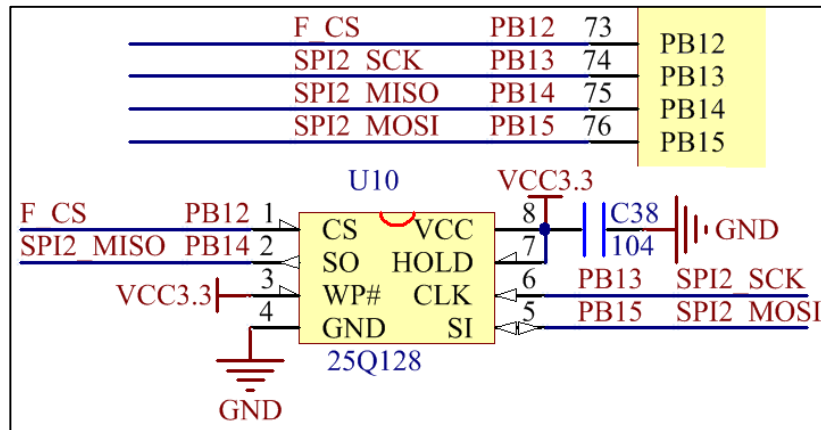


图 36.3.1 NOR FLASH 与开发板的连接原理

通过上图可知，NM25Q128 的 CS、SCK、MISO 和 MOSI 分别连接在 PB12、PB13、PB14 和 PB15 上。本实验还支持多种型号的 SPI FLASH 芯片，比如：BY25Q128/NM25Q128/W25Q128 等等，具体请看 norflash.h 文件的宏定义，在程序上只需要稍微修改一下，后面讲解程序的时候会提到。

36.3 程序设计

36.3.1 SPI 的 HAL 库驱动

SPI 在 HAL 库中的驱动代码在 stm32f1xx_hal_spi.c 文件（及其头文件）中。

1. HAL_SPI_Init 函数

SPI 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

- 函数描述：

用于初始化 SPI。

- 函数形参：

形参 1 是 SPI_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct __SPI_HandleTypeDef
{
    SPI_TypeDef                *Instance;          /* SPI 寄存器基地址 */
    SPI_InitTypeDef            Init;               /* SPI 通信参数 */
    uint8_t                    *pTxBuffPtr;       /* SPI 的发送缓存 */
    uint16_t                    TxXferSize;        /* SPI 的发送数据大小 */
    __IO uint16_t               TxXferCount;       /* SPI 发送端计数器 */
    uint8_t                    *pRxBuffPtr;       /* SPI 的接收缓存 */
    uint16_t                    RxXferSize;        /* SPI 的接收数据大小 */
    __IO uint16_t               RxXferCount;       /* SPI 接收端计数器 */
    void (*RxISR)(struct __SPI_HandleTypeDef *hspi); /* SPI 的接收端中断服务函数 */
    void (*TxISR)(struct __SPI_HandleTypeDef *hspi); /* SPI 的发送端中断服务函数 */
    DMA_HandleTypeDef          *hdmatx;           /* SPI 发送参数设置 (DMA) */
    DMA_HandleTypeDef          *hdmarx;           /* SPI 接收参数设置 (DMA) */
    HAL_LockTypeDef             Lock;              /* SPI 锁对象 */
    __IO HAL_SPI_StateTypeDef   State;             /* SPI 传输状态 */
    __IO uint32_t               ErrorCode;         /* SPI 操作错误代码 */
} SPI_HandleTypeDef;
```

我们这里主要讲解第二个成员变量 Init，它是 SPI_InitTypeDef 结构体类型，该结构体定义如下：

```
typedef struct
{
    uint32_t Mode;                /* 模式：主:SPI_MODE_MASTER 从:SPI_MODE_SLAVE */
    uint32_t Direction;           /* 方向：只接收模式 单线双向通信数据模式 全双工 */
}
```

```
uint32_t DataSize;          /* 数据帧格式： 8 位/16 位 */
uint32_t CLKPolarity;       /* 时钟极性 CPOL 高/低电平 */
uint32_t CLKPhase;         /* 时钟相位 奇/偶数边沿采集 */
uint32_t NSS;              /* SS 信号由硬件（NSS）管脚控制还是软件控制 */
uint32_t BaudRatePrescaler; /* 设置 SPI 波特率预分频值 */
uint32_t FirstBit;         /* 起始位是 MSB 还是 LSB */
uint32_t TIMode;           /* 帧格式 SPI motorola 模式还是 TI 模式 */
uint32_t CRCCalculation;   /* 硬件 CRC 是否使能 */
uint32_t CRCPolynomial;    /* 设置 CRC 多项式 */
} SPI_InitTypeDef;
```

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值。

使用 SPI 传输数据的配置步骤

1) SPI 参数初始化(工作模式、数据时钟极性、时钟相位等)。

HAL 库通过调用 SPI 初始化函数 HAL_SPI_Init 完成对 SPI 参数初始化，详见例程源码。

注意：该函数会调用：HAL_SPI_MspInit 函数来完成对 SPI 底层的初始化，包括：SPI 及 GPIO 时钟使能、GPIO 模式设置等。

2) 使能 SPI 时钟和配置相关引脚的复用功能。

本实验用到 SPI2，使用 PB13、PB14 和 PB15 作为 SPI_SCK、SPI_MISO 和 SPI_MOSI，因此需要先使能 SPI2 和 GPIOB 时钟。参考代码如下：

```
HAL_RCC_SPI2_CLK_ENABLE();
HAL_RCC_GPIOB_CLK_ENABLE();
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

3) 使能 SPI

通过 HAL_SPI_ENABLE 函数使能 SPI，便可进行数据传输。

4) SPI 传输数据

通过 HAL_SPI_Transmit 函数进行发送数据。

通过 HAL_SPI_Receive 函数进行接收数据。

也可以通过 HAL_SPI_TransmitReceive 函数进行发送与接收操作。

5) 设置 SPI 传输速度

SPI 初始化结构体 SPI_InitTypeDef 有一个成员变量是 BaudRatePrescaler，该成员变量用来设置 SPI 的预分频系数，从而决定了 SPI 的传输速度。但是 HAL 库并没有提供单独的 SPI 分频系数修改函数，如果我们需要在程序中偶尔修改速度，那么我们就需要通过设置 SPI_CR1 寄存器来修改，具体实现方法请参考后面软件设计小节相关函数。

36.3.2 程序流程图

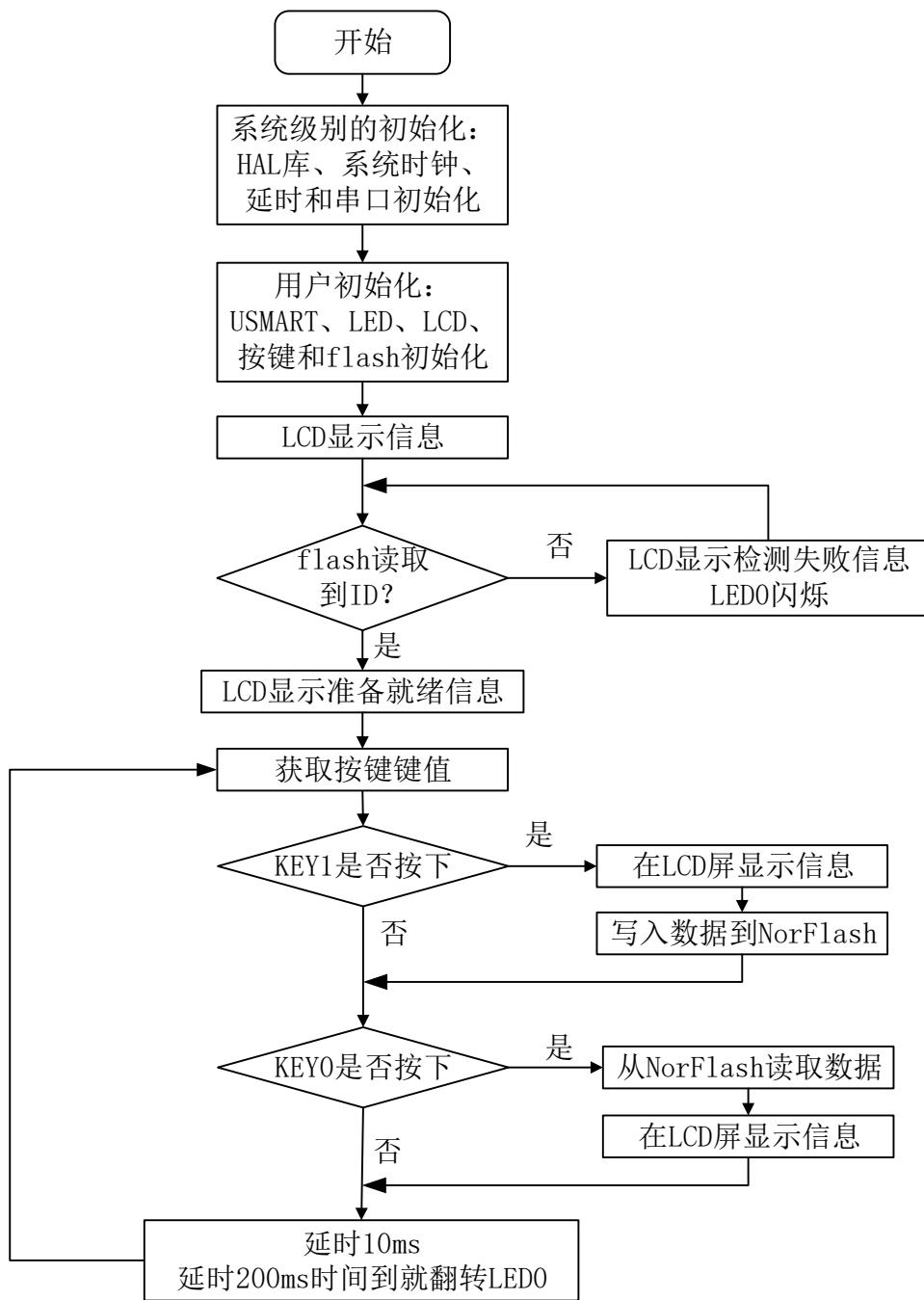


图 36.3.2.1 SPI 实验程序流程图

36.3.3 程序解析

本实验中，我们通过调用 HAL 库的函数去驱动 SPI 进行通信，所以需要在工程分组下添加 stm32f1xx_hal_spi.c 文件。实验工程中，我们新增了 spi.c 存放 spi 底层驱动代码，norflash.c 文件存放 W25Q128/NM25Q128 驱动。

1. SPI 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SPI 驱动源码包括两个文件：spi.c 和 spi.h。

下面我们直接介绍 SPI 相关的程序，首先先介绍 spi.h 文件，其定义如下：

```
/* SPI2 引脚 定义 */
#define SPI2_SCK_GPIO_PORT      GPIOB
#define SPI2_SCK_GPIO_PIN      GPIO_PIN_13
#define SPI2_SCK_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE();}while(0)

#define SPI2_MISO_GPIO_PORT      GPIOB
#define SPI2_MISO_GPIO_PIN      GPIO_PIN_14
#define SPI2_MISO_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define SPI2_MOSI_GPIO_PORT      GPIOB
#define SPI2_MOSI_GPIO_PIN      GPIO_PIN_15
#define SPI2_MOSI_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

/* SPI2 相关定义 */
#define SPI2_SPI                SPI2
#define SPI2_SPI_CLK_ENABLE()   do{ __HAL_RCC_SPI2_CLK_ENABLE();}while(0)
```

我们通过宏定义标识符的方式去定义 SPI 通信用到的三个管脚 SCK、MISO 和 MOSI，同时还宏定义 SPI2 的相关信息。

接下来我们看一下 spi.c 代码中的初始化函数，代码如下：

```
/**
 * @brief      SPI 初始化代码
 * @note      主机模式, 8 位数据, 禁止硬件片选
 * @param      无
 * @retval     无
 */
SPI_HandleTypeDef g_spi2_handler;          /* SPI2 句柄 */

void spi2_init(void)
{
    SPI2_SPI_CLK_ENABLE();                  /* SPI2 时钟使能 */

    g_spi2_handler.Instance = SPI2_SPI;      /* SPI2 */
    g_spi2_handler.Init.Mode = SPI_MODE_MASTER; /* 设置 SPI 工作模式, 设置为主模式 */
    /* 设置 SPI 单向或者双向的数据模式:SPI 设置为双线模式 */
    g_spi2_handler.Init.Direction = SPI_DIRECTION_2LINES;
    /* 设置 SPI 的数据大小:SPI 发送接收 8 位帧结构 */
    g_spi2_handler.Init.DataSize = SPI_DATASIZE_8BIT;
    /* 串行同步时钟的空闲状态为高电平 */
    g_spi2_handler.Init.CLKPolarity = SPI_POLARITY_HIGH;
    /* 串行同步时钟的第二个跳变沿 (上升或下降) 数据被采样 */
    g_spi2_handler.Init.CLKPhase = SPI_PHASE_2EDGE;
    /* NSS 信号由硬件 (NSS 管脚) 还是软件 (使用 SSI 位) 管理:内部 NSS 信号有 SSI 位控制 */
    g_spi2_handler.Init.NSS = SPI_NSS_SOFT;
    /* 定义波特率预分频的值:波特率预分频值为 256 */
    g_spi2_handler.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
    /* 指定数据传输从 MSB 位还是 LSB 位开始:数据传输从 MSB 位开始 */
    g_spi2_handler.Init.FirstBit = SPI_FIRSTBIT_MSB;
    g_spi2_handler.Init.TIMode = SPI_TIMODE_DISABLE; /* 关闭 TI 模式 */
    /* 关闭硬件 CRC 校验 */
    g_spi2_handler.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    g_spi2_handler.Init.CRCPolynomial = 7;          /* CRC 值计算的多项式 */
    HAL_SPI_Init(&g_spi2_handler);                /* 初始化 */

    __HAL_SPI_ENABLE(&g_spi2_handler);            /* 使能 SPI2 */
    /* 启动传输, 实际上就是产生 8 个时钟脉冲, 达到清空 DR 的作用, 非必需 */
    spi2_read_write_byte(0xff);
}
```

在 spi_init 函数中主要工作就是对于 SPI 参数的配置，这里包括工作模式、数据模式、数据大小、时钟极性、时钟相位、波特率预分频值等。关于 SPI 的管脚配置就放在了 HAL_SPI_MspInit 函数里，其代码如下：

```

/**
 * @brief      SPI 底层驱动, 时钟使能, 引脚配置
 * @note      此函数会被 HAL_SPI_Init() 调用
 * @param      hspi: SPI 句柄
 * @retval     无
 */
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)
{
    GPIO_InitTypeDef gpio_init_struct;
    if (hspi->Instance == SPI2_SPI)
    {
        SPI2_SCK_GPIO_CLK_ENABLE(); /* SPI2_SCK 脚时钟使能 */
        SPI2_MISO_GPIO_CLK_ENABLE(); /* SPI2_MISO 脚时钟使能 */
        SPI2_MOSI_GPIO_CLK_ENABLE(); /* SPI2_MOSI 脚时钟使能 */

        gpio_init_struct.Pin = SPI2_SCK_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* SCK 引脚模式设置(复用输出) */
        gpio_init_struct.Pull = GPIO_PULLUP;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(SPI2_SCK_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = SPI2_MISO_GPIO_PIN; /* MISO 引脚模式设置(复用输出) */
        HAL_GPIO_Init(SPI2_MISO_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = SPI2_MOSI_GPIO_PIN; /* MOSI 引脚模式设置(复用输出) */
        HAL_GPIO_Init(SPI2_MOSI_GPIO_PORT, &gpio_init_struct);
    }
}

```

通过以上两个函数的作用就可以完成 SPI 初始化。接下来介绍 SPI 的发送和接收函数，其定义如下：

```

/**
 * @brief      SPI2 读写一个字节数据
 * @param      txdata : 要发送的数据(1 字节)
 * @retval     接收到的数据(1 字节)
 */
uint8_t spi2_read_write_byte(uint8_t txdata)
{
    uint8_t rxdata;
    HAL_SPI_TransmitReceive(&g_spi2_handler, &txdata, &rxdata, 1, 1000);
    return rxdata; /* 返回收到的数据 */
}

```

这里的 spi_read_write_byte 函数直接调用了 HAL 库内置的函数进行接收发送操作。前面已经有介绍了，这里就不展开对 HAL_SPI_TransmitReceive 函数的解析。

由于不同的外设需要的通信速度不一样，所以这里我们定义了一个速度设置函数，通过操作寄存器的方式去实现，其代码如下：

```

/**
 * @brief      SPI2 速度设置函数
 * @note      SPI2 时钟选择来自 APB1, 即 PCLK1, 为 36Mhz
 *            SPI 速度 = PCLK1 / 2^(speed + 1)
 * @param      speed : SPI2 时钟分频系数
 *            取值为 SPI_BAUDRATEPRESCALER_2~SPI_BAUDRATEPRESCALER_256
 * @retval     无
 */
void spi2_set_speed(uint8_t speed)
{
    assert_param(IS_SPI_BAUDRATE_PRESCALER(speed)); /* 判断有效性 */
    __HAL_SPI_DISABLE(&g_spi2_handler); /* 关闭 SPI */
    g_spi2_handler.Instance->CR1 &= 0xFFC7; /* 位 3-5 清零, 用来设置波特率 */
    g_spi2_handler.Instance->CR1 |= speed << 3; /* 设置 SPI 速度 */
    __HAL_SPI_ENABLE(&g_spi2_handler); /* 使能 SPI */
}

```


2. NOR FLASH 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。NOR FLASH 驱动源码包括两个文件：norflash.c 和 norflash.h。

在上一小节已经对 SPI 协议需要用到的东西都封装好了。那么现在就要在 SPI 通信的基础上，通过前面分析的 NM25Q128 的工作时序拟定通信代码。

由于这部分的代码量比较多，这里就不一一贴出来介绍。介绍几个重点，其余的请自行查看源码。首先是 norflash.h 头文件，我们做了一个 FLASH 芯片列表（宏定义），这些宏定义是一些支持的 FLASH 芯片的 ID。接下来是 FLASH 芯片指令表的宏定义，这个请参考 FLASH 芯片手册比对得到，这里就不将代码列出来了。

下面介绍 norflash.c 文件几个重要的函数，首先是 NOR FLASH 初始化函数，其定义如下：

```
/**
 * @brief      初始化 SPI NOR FLASH
 * @param      无
 * @retval     无
 */
void norflash_init(void)
{
    uint8_t temp;

    NORFLASH_CS_GPIO_CLK_ENABLE();          /* NORFLASH CS 脚 时钟使能 */

    /* CS 引脚模式设置(复用输出) */
    GPIO_InitTypeDef gpio_init_struct;
    gpio_init_struct.Pin = NORFLASH_CS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(NORFLASH_CS_GPIO_PORT, &gpio_init_struct);

    NORFLASH_CS(1);                          /* 取消片选 */

    spi2_init();                             /* 初始化 SPI2 */
    spi2_set_speed(SPI_SPEED_2);             /* SPI2 切换到高速状态 18Mhz */

    g_norflash_type = norflash_read_id();    /* 读取 FLASH ID. */

    if (g_norflash_type == W25Q256) /* SPI FLASH 为 W25Q256，必须使能 4 字节地址模式 */
    {
        temp = norflash_read_sr(3); /* 读取状态寄存器 3，判断地址模式 */
        if ((temp & 0X01) == 0)    /* 如果不是 4 字节地址模式，则进入 4 字节地址模式 */
        {
            norflash_write_enable(); /* 写使能 */
            temp |= 1 << 1;          /* ADP=1，上电 4 位地址模式 */
            norflash_write_sr(3, temp); /* 写 SR3 */

            NORFLASH_CS(0);
            spi1_read_write_byte(FLASH_Enable4ByteAddr); /* 使能 4 字节地址指令 */
            NORFLASH_CS(1);
        }
    }

    //printf("ID:%x\r\n", g_norflash_type);
}
```

在初始化函数中，将 SPI 通信协议用到的 CS 引脚配置好，同时根据 FLASH 的通信要求，通过调用 spi2_set_speed 函数把 SPI2 切换到高速状态。然后尝试读取 flash 的 ID，由于 W25Q256 的容量比较大，通信的时候需要 4 个字节，为了函数的兼容性，我们这里做了判断处理。当然，我们使用的 NM25Q128 是 3 字节地址模式的。如果能读到 ID 则说明我们的 SPI 时序能正常操作 FLASH，便可以通过 SPI 接口读写 NOR FLASH 的数据了。

进行其它数据操作时，由于每一次读写操作的时候都需要发送地址，所以这里我们把这个板块封装成函数，函数名是 `norflash_send_address`，实质上就是通过 SPI 的发送接收函数 `spi2_read_write_byte` 实现的，这里就不列出来了，大家可以查看光盘源码。

下面介绍一下 FLASH 读取函数，这里可以根据前面的时序图对照理解，其定义如下：

```
/**
 * @brief      读取 SPI FLASH
 * @note      在指定地址开始读取指定长度的数据
 * @param      pbuf      : 数据存储区
 * @param      addr      : 开始读取的地址(最大 32bit)
 * @param      datalen   : 要读取的字节数(最大 65535)
 * @retval     无
 */
void norflash_read(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint16_t i;

    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_ReadData);          /* 发送读取命令 */
    norflash_send_address(addr);                    /* 发送地址 */

    for(i=0;i<datalen;i++)
    {
        pbuf[i] = spi2_read_write_byte(0xFF);      /* 循环读取 */
    }

    NORFLASH_CS(1);
}
```

该函数用于从 NOR FLASH 的指定位置读出指定长度的数据，由于 NOR FLASH 支持以任意地址（但是不能超过 NOR FLASH 的地址范围）开始读取数据，所以，这个代码相对来说比较简单。首先拉低片选信号，发送读取命令，接着发送 24 位地址之后，程序就可以开始循环读取数据，其地址就会自动增加，读取完数据后，需要拉高片选信号，结束通信。

有读函数，那肯定就有写函数，接下来我们介绍一下 NOR FLASH 写函数，其定义如下：

```
/**
 * @brief      写 SPI FLASH
 * @note      在指定地址开始写入指定长度的数据，该函数带擦除操作！
 *            SPI FLASH 一般是：256 个字节为一个 Page, 4Kbytes 为一个 Sector, 16 个扇区
 *            为 1 个 Block, 擦除的最小单位为 Sector.
 *
 * @param      pbuf      : 数据存储区
 * @param      addr      : 开始写入的地址(最大 32bit)
 * @param      datalen   : 要写入的字节数(最大 65535)
 * @retval     无
 */
uint8_t g_norflash_buf[4096]; /* 扇区缓存 */
void norflash_write(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint32_t secpos;
    uint16_t secoff;
    uint16_t secremain;
    uint16_t i;
    uint8_t *norflash_buf;

    norflash_buf = g_norflash_buf;
    secpos = addr / 4096;          /* 扇区地址 */
    secoff = addr % 4096;          /* 在扇区内的偏移 */
    secremain = 4096 - secoff;     /* 扇区剩余空间大小 */

    if (datalen <= secremain)
    {
        secremain = datalen;      /* 不大于 4096 个字节 */
    }
}
```

```

}

while (1)
{
    norflash_read(norflash_buf, secpos * 4096, 4096); /* 读出整个扇区的内容 */
    for (i = 0; i < secremain; i++) /* 校验数据 */
    {
        if (norflash_buf[secoff + i] != 0xFF)
        {
            break; /* 需要擦除, 直接退出 for 循环 */
        }
    }

    if (i < secremain) /* 需要擦除 */
    {
        norflash_erase_sector(secpos); /* 擦除这个扇区 */
        for (i = 0; i < secremain; i++) /* 复制 */
        {
            norflash_buf[i + secoff] = pbuf[i];
        }
        /* 写入整个扇区 */
        norflash_write_noccheck(norflash_buf, secpos * 4096, 4096);
    }
    else /* 写已经擦除了的, 直接写入扇区剩余区间. */
    {
        norflash_write_noccheck(pbuf, addr, secremain); /* 直接写扇区 */
    }

    if (datalen == secremain)
    {
        break; /* 写入结束了 */
    }
    else /* 写入未结束 */
    {
        secpos++; /* 扇区地址增 1 */
        secoff = 0; /* 偏移位置为 0 */
        pbuf += secremain; /* 指针偏移 */
        addr += secremain; /* 写地址偏移 */
        datalen -= secremain; /* 字节数递减 */
        if (datalen > 4096)
        {
            secremain = 4096; /* 下一个扇区还是写不完 */
        }
        else
        {
            secremain = datalen; /* 下一个扇区可以写完了 */
        }
    }
}
}

```

该函数可以在 NOR FLASH 的任意地址开始写入任意长度（必须不超过 NOR FLASH 的容量）的数据。

我们这里简单介绍一下思路：先获得首地址（addr）所在的扇区，并计算在扇区内的偏移，然后判断要写入的数据长度是否超过本扇区所剩下的长度，如果不超过，再先看看是否要擦除，如果不要，则直接写入数据即可，如果要则读出整个扇区，在偏移处开始写入指定长度的数据，然后擦除这个扇区，再一次性写入。当所需要写入的数据长度超过一个扇区的长度的时候，我们先按照前面的步骤把扇区剩余部分写完，再在新扇区内执行同样的操作，如此循环，直到写入结束。这里我们还定义了一个 g_norflash_buf 的全局变量，用于擦除时缓存扇区内的数据。

简单介绍一下写函数的实质调用，它用到的是通过无检验写 SPI_FLASH 函数实现的，而最终是用到页写函数 norflash_write_page，在前面也对页写时序进行了分析，现在看一下代码：

```
/**
 * @brief      SPI 在一页 (0~65535) 内写入少于 256 个字节的数据
 * @note      在指定地址开始写入最大 256 字节的数据
 * @param      pbuf      : 数据存储区
 * @param      addr      : 开始写入的地址 (最大 32bit)
 * @param      datalen   : 要写入的字节数 (最大 256), 该数不应该超过该页的剩余字节数!!!
 * @retval     无
 */
static void norflash_write_page(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint16_t i;

    norflash_write_enable();                /* 写使能 */

    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_PageProgram); /* 发送写页命令 */
    norflash_send_address(addr);             /* 发送地址 */

    for(i=0;i<datalen;i++)
    {
        spi2_read_write_byte(pbuf[i]);      /* 循环写入 */
    }
    NORFLASH_CS(1);
    norflash_wait_busy();                   /* 等待写入结束 */
}
```

在页写功能的代码中, 先发送写使能命令, 才发送页写命令, 然后发送写入的地址, 再把写入的内容通过一个 for 循环写入, 发送完后拉高片选 CS 引脚结束通信, 等待 flash 内部写入结束。检测 flash 内部的状态可以通过查看 NM25Qxx 状态寄存器 1 的位 0。在这里科普一下 NM25Qxx 的状态寄存器, 可以通过寄存器相关位判断 NM25Qxx 的状态, 下面是 NM25Qxx 状态寄存器表:

| 状态寄存器 | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|---------|----------|------|------|------|------|------|------|------|
| 状态寄存器 1 | SPR | RV | TB | BP2 | BP1 | BP0 | WEL | BUSY |
| 状态寄存器 2 | SUS | CMP | LB3 | LB2 | LB1 | (R) | QE | SRP1 |
| 状态寄存器 3 | HOLD/RST | DRV1 | DRV0 | (R) | (R) | WPS | ADP | ADS |

表 36.3.3.1 NM25Qxx 状态寄存器表

我们也定义了一个函数 `norflash_read_sr`, 去读取 NM25Qxx 状态寄存器的值, 这里就不列出来了, 主要实现的方式也是老套路: 根据传参判断需要获取的是哪个状态寄存器, 然后拉低片选线, 调用 `spi2_read_write_byte` 函数发送该寄存器的命令, 然后通过发送一字节空数据获取读取到的数据, 最后拉高片选线, 函数返回读取到的值。

在 `norflash_write_page` 函数的基础上, 增加了 `norflash_write_noccheck` 函数进行封装解决写入字节可能大于该页剩下的字节数问题, 方便解决写入错误问题, 其代码如下:

```
/**
 * @brief      无检验写 SPI FLASH
 * @note      必须确保所写的地址范围内的数据全部为 0xFF, 否则在非 0xFF 处写入的数据将失败!
 *            具有自动换页功能
 *            在指定地址开始写入指定长度的数据, 但是要确保地址不越界!
 *
 * @param      pbuf      : 数据存储区
 * @param      addr      : 开始写入的地址 (最大 32bit)
 * @param      datalen   : 要写入的字节数 (最大 65535)
 * @retval     无
 */
static void norflash_write_noccheck(uint8_t *pbuf, uint32_t addr,
                                     uint16_t datalen)
{
    uint16_t pageremain;
    pageremain = 256 - addr % 256; /* 单页剩余的字节数 */
```

```

if (datalen <= pageremain)      /* 不大于 256 个字节 */
{
    pageremain = datalen;
}
while (1)
{
    /* 当写入字节比页内剩余地址还少的时候，一次性写完
    * 当写入直接比页内剩余地址还多的时候，先写完整个页内剩余地址，然后根据剩余长度进行不同处理
    */
    norflash_write_page(pbuf, addr, pageremain);

    if (datalen == pageremain)    /* 写入结束了 */
    {
        break;
    }
    else      /* datalen > pageremain */
    {
        pbuf += pageremain; /* pbuf 指针地址偏移,前面已经写了 pageremain 字节 */
        addr += pageremain;  /* 写地址偏移,前面已经写了 pageremain 字节 */
        datalen -= pageremain; /* 写入总长度减去已经写入了的字节数 */

        if (datalen > 256)      /* 剩余数据还大于一页,可以一次写一页 */
        {
            pageremain = 256;    /* 一次可以写入 256 个字节 */
        }
        else      /* 剩余数据小于一页,可以一次写完 */
        {
            pageremain = datalen; /* 不够 256 个字节了 */
        }
    }
}
}

```

上面函数的实现主要是逻辑处理,通过判断传参中的写入字节的长度与单页剩余的字节数,来决定是否是需要在新页写入剩下的字节。这里需要大家自行理解一下。通过调用该函数实现了 `norflash_write` 的功能。

下面简单介绍一下擦除函数 `norflash_erase_sector`,前面工作时序中也有对此描述,现在就看一下代码:

```

/**
 * @brief      擦除一个扇区
 * @note      注意,这里是扇区地址,不是字节地址!!
 *            擦除一个扇区的最少时间:150ms
 *
 * @param      saddr : 扇区地址 根据实际容量设置
 * @retval     无
 */
void norflash_erase_sector(uint32_t saddr)
{
    saddr *= 4096;
    norflash_write_enable();      /* 写使能 */
    norflash_wait_busy();         /* 等待空闲 */
    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_SectorErase); /* 发送写页命令 */
    norflash_send_address(saddr); /* 发送地址 */
    NORFLASH_CS(1);
    norflash_wait_busy();         /* 等待扇区擦除完成 */
}

```

该代码也是老套路,通过发送擦除指令实现擦除功能,要注意的是使用扇区擦除指令前,需要先发送写使能指令,拉低片选线,发送扇区擦除指令之后,发送擦除的扇区地址,实现擦除,最后拉高片选线结束通信。在函数最后通过读取寄存器状态的函数,等待扇区擦除完成。

3. main.c 代码

在 main.c 里面编写如下代码:

```
const uint8_t g_text_buf[] = {"STM32 SPI TEST"}; /* 要写到 FLASH 的字符串数组 */
#define TEXT_SIZE sizeof(g_text_buf) /* TEXT 字符串长度 */

int main(void)
{
    uint8_t key;
    uint16_t i = 0;
    uint8_t datatemp[TEXT_SIZE];
    uint32_t flashsize;
    uint16_t id = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    norflash_init(); /* 初始化 NORFLASH */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "SPI TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write KEY0:Read", RED);

    id = norflash_read_id(); /* 读取 FLASH ID */
    while ((id == 0) || (id == 0xFFFF)) /* 检测不到 FLASH 芯片 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "FLASH Check Failed!", RED);
        delay_ms(500);
        lcd_show_string(30, 130, 200, 16, 16, "Please Check! ", RED);
        delay_ms(500);
        LED0_TOGGLE(); /* LED0 闪烁 */
    }
    lcd_show_string(30, 130, 200, 16, 16, "SPI FLASH Ready!", BLUE);
    flashsize = 16 * 1024 * 1024; /* FLASH 大小为 16M 字节 */

    while (1)
    {
        key = key_scan(0);

        if (key == KEY1_PRES) /* KEY1 按下, 写入 */
        {
            /* 从倒数第 100 个地址处开始, 写入 SIZE 长度的数据 */
            lcd_fill(0, 150, 239, 319, WHITE); /* 清除半屏 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Write FLASH...", BLUE);
            sprintf((char *)datatemp, "%s%d", (char *)g_text_buf, i);
            norflash_write((uint8_t *)datatemp, flashsize - 100, TEXT_SIZE);
            lcd_show_string(30, 150, 200, 16, 16, "FLASH Write Finished!", BLUE);
        }

        if (key == KEY0_PRES) /* KEY0 按下, 读取字符串并显示 */
        {
            /* 从倒数第 100 个地址处开始, 读出 SIZE 个字节 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Read FLASH...", BLUE);
            norflash_read(datatemp, flashsize - 100, TEXT_SIZE);
            lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is: ", BLUE);
            lcd_show_string(30, 170, 200, 16, 16, (char *)datatemp, BLUE);
        }

        i++;
    }
}
```



```
if (i == 20)
{
    LED0_TOGGLE(); /* LED0 闪烁 */
    i = 0;
}
delay_ms(10);
}
```

在 main 函数前面，我们定义了 g_text_buf 数组，用于存放要写入到 FLASH 的字符串。main 函数代码和 IIC 实验那部分代码大同小异，具体流程大致是：在完成系统级和用户级初始化工作后，读取 FLASH 的 ID，然后通过 KEY0 去读取倒数第 100 个地址处开始的数据并把数据显示在 LCD 上；另外还可以通过 KEY1 去倒数第 100 个地址处写入 g_text_buf 数据并在 LCD 界面中显示传输中，完成后并显示“FLASH Write Finished!”。

36.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 36.4.1 所示：

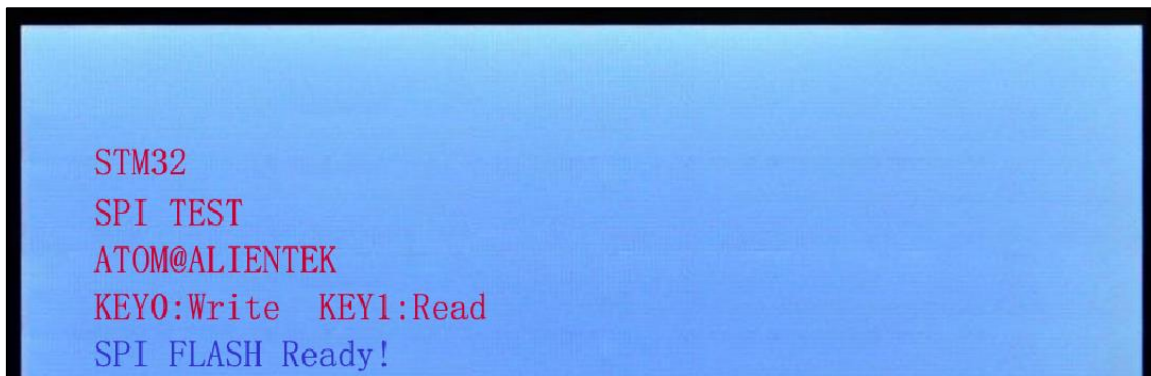


图 36.4.1 SPI 实验程序运行效果图

通过先按下 KEY1 写入数据，然后再按 KEY0 读取数据，得到如图 36.4.2 所示：

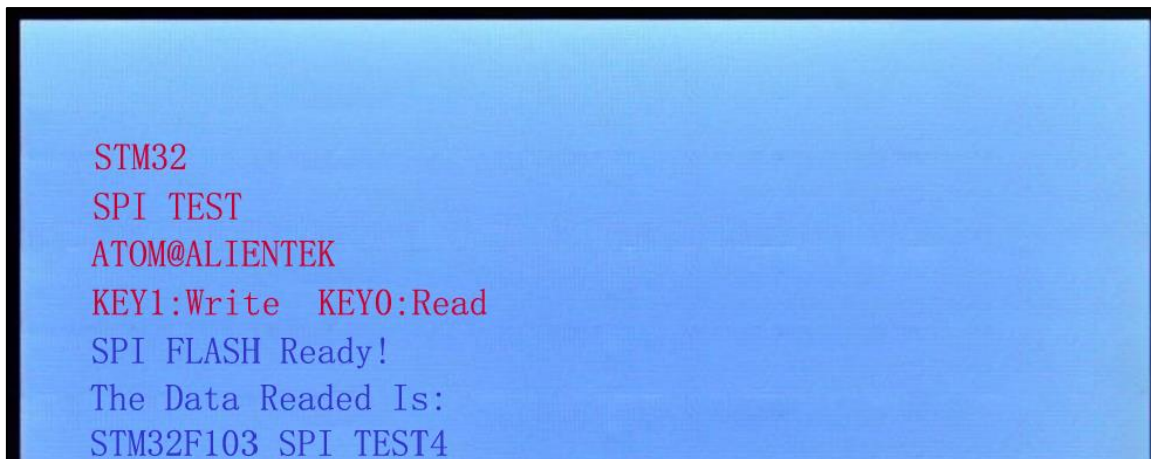


图 36.4.2 操作后的显示效果图

程序在开机的时候会检测 NOR FLASH 是否存在，如果不存在则会在 LCD 模块上显示错误信息，同时 LED0 慢闪。大家可以通过跳线帽把 PB14 和 PB15 短接就可以看到报错了。

该实验还支持 USMART，在这里我们加入了 norflash_read_id 和 norflash_erase_chip 以及 norflash_erase_sector 函数。可以通过 USMART 调用 norflash_read_id 函数去读取 SPI_FLASH 的 ID，也可以调用另外两个擦除函数。需要注意的是假如调用了 norflash_erase_chip 函数将会对整个 SPI_FLASH 进行擦除，一般情况不建议对整个 SPI_FLASH 进行擦除，因为会导致字库和综合例程所需要的系统文件全部丢失。

第三十七章 485 实验

本章我们将向大家介绍如何使用 STM32F1 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32F1 的串口 2 来实现两块开发板之间的 485 通信，并将结果显示在 TFTLCD 模块上。

本章分为如下几个部分：

37.1 485 简介：

37.2 硬件设计

37.3 程序设计

37.4 下载验证

37.1 485 简介

485（一般称作 RS485/EIA-485）隶属于 OSI 模型物理层，是串行通讯的一种。电气特性规定为 2 线，半双工，多点通信的类型。它的电气特性和 RS-232 大不一样。用缆线两端的电压差值来表示传递信号。RS485 仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS485 的特点包括：

- 1，接口电平低，不易损坏芯片。RS485 的电气特性：逻辑“1”以两线间的电压差为+(2~6)V 表示；逻辑“0”以两线间的电压差为-(2~6)V 表示。接口信号电平比 RS232 降低了，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。
- 2，传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200m 时，传输速度可达 100Kbps。
- 3，抗干扰能力强。RS485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。
- 4，传输距离远，支持节点多。RS485 总线最长可以传输 1200m 左右，更远的距离则需要中继传输设备支持但这时（速率 $\leq 100\text{Kbps}$ ）才能稳定传输，一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，比如：线型，总线型网络等，而不能是星型，环型网络。理想情况下 RS485 需要 2 个终端匹配电阻，其阻值要求等于传输电缆的特性阻抗（一般为 120Ω ）。没有特性阻抗的话，当所有的设备都静止或者没有能量的时候就会产生噪声，而且线移需要双端的电压差。没有终接电阻的话，会使得较快速的发送端产生多个数据信号的边缘，导致数据传输出错。485 推荐的一主多从连接方式如图 37.1.1 所示：

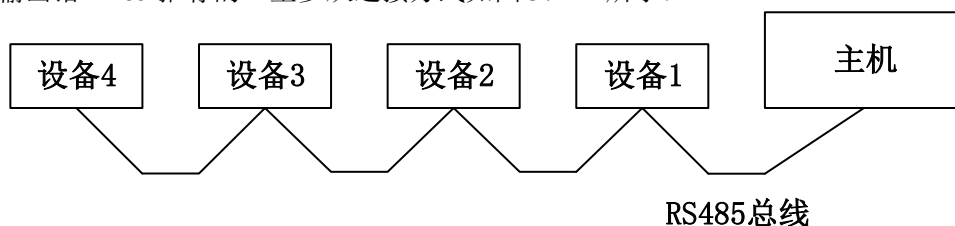


图 37.1.1 RS485 连接

在上面的连接中，如果需要添加匹配电阻，我们一般在总线的起止端加入，也就是主机和设备 4 上面各加一个 120Ω 的匹配电阻。

由于 RS485 具有传输距离远、传输速度快、支持节点多和抗干扰能力更强等特点，所以 RS485 有很广泛的应用。实际多设备时收发器有范围为 -7V 到 +12V 的共模电压，为了稳定传输，也有使用 3 线的布线方式，即在原有的 A、B 两线上多增加一条地线。（4 线制只能实现点对点的全双工通讯方式，这种也叫 RS422，由于布线的难度和通讯局限，相对使用得比较少）。

TP8485E/SP3485 可作为 RS485 的收发器，该芯片支持 3.3V~5.5V 供电，最大传输速度可达 250Kbps，支持多达 256 个节点（单位负载为 1/8 的条件下），并且支持输出短路保护。该芯片

的框图如图 37.1.2 所示：

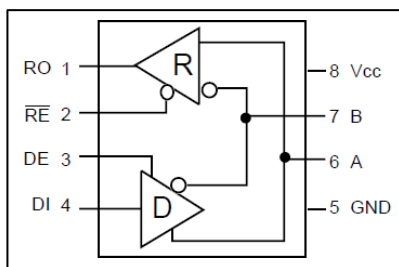


图 37.1.2 TP8485E/SP3485 框图

图中 A、B 总线接口，用于连接 485 总线。RO 是接收输出端，DI 是发送数据收入端，RE 是接收使能信号（低电平有效），DE 是发送使能信号（高电平有效）。

37.2 硬件设计

1. 例程功能

经过前面的学习我们知道实际的 RS485 仍是串行通讯的一种电平传输方式，那么我们实际通讯时可以使用串口进行实际数据的收发处理，使用 485 转换芯片将串口信号转换为 485 的电平信号进行传输，本章，我们只需要配置好串口 2，就可以实现正常的 485 通信了，串口 2 的配置和串口 1 基本类似，只是串口 2 的时钟来自 APB1，最大频率为 36Mhz。

本章将实现这样的功能：通过连接两个战舰 STM32F103 的 RS485 接口，然后由 KEY0 控制发送，当按下一个开发板的 KEY0 的时候，就发送 5 个数据给另外一个开发板，并在两个开发板上分别显示发送的值和接收到的值。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) USART2，用于实际的 485 信号串行通讯。

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) RS485 收发芯片 TP8485/SP3485

5) 开发板两块（485 半双式模式无法自收发，我们需要用两个开发板或者 USB 转 485 调试器+串口助手来帮助我们完成测试，大家根据自己的实际条件选择）

3. 原理图

根据我们需要实现的程序功能，我们设计电路原理如下：

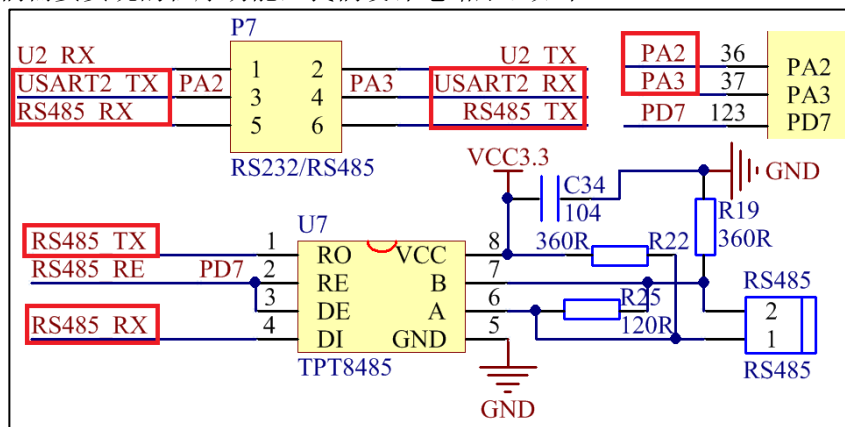


图 37.2.2 RS485 连接原理设计

从上图可以看出：开发板的串口 2 和 TP8485 上的引脚连接到 P7 端上的端子，但不直接相连，所以测试 485 功能时我们需要用跳线帽短接 P7 上的两组排针使之连通。STM32F1 的 PD7

控制 RS485 的收发模式：

当 PD7=0 的时候，为接收模式；当 PD7=1 的时候，为发送模式。

这里需要注意，RS485_RE 信号和 CH395Q_RST 共用 PD7，所以对于我们的战舰开发板来说他们也不可以同时使用，只能分时复用。

另外，图中的 R19 和 R22 是两个偏置电阻，用来保证总线空闲时，A、B 之间的电压差都会大于 200mV（逻辑 1）。从而避免因总线空闲时因 A、B 压差不稳定，可能出现的乱码。

最后，我们用 2 根导线将两个开发板 RS485 端子的 A 和 A，B 和 B 连接起来。这里注意不要接反了（A 接 B），接反了会导致通讯异常！！

37.3 程序设计

37.3.1 RS485 的 HAL 库驱动

由于 485 实际上是串口通讯，我们参照串口实验一节使用类似的 HAL 库驱动即可，在这里分析一下 RS485 配置步骤。

RS485 配置步骤

1) 使能串口和 GPIO 口时钟

本实验用到 USART2 串口，使用 PA2 和 PA3 作为串口的 TX 和 RX 脚，因此需要先使能 USART2 和 GPIOA 时钟。参考代码如下：

```
__HAL_RCC_USART2_CLK_ENABLE(); /* 使能 USART2 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE(); /* 使能 GPIOA 时钟 */
```

2) 串口参数初始化（波特率、字长、奇偶校验等）

HAL 库通过调用串口初始化函数 HAL_UART_Init 完成对串口参数初始化，详见例程源码。

该函数通常会调用：HAL_UART_MspInit 函数来完成对串口底层的初始化，包括：串口及 GPIO 时钟使能、GPIO 模式设置、中断设置等。但是本实验避免与 USART1 冲突，所以把串口底层初始化没有放在 HAL_UART_MspInit 函数里。

3) GPIO 模式设置（速度，上下拉，复用功能等）

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

4) 开启串口相关中断，配置串口中断优先级

本实验我们使用串口中断来接收数据。我们使用 HAL_UART_Receive_IT 函数开启串口中断接收，并设置接收 buffer 及其长度。通过 HAL_NVIC_EnableIRQ 函数使能串口中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5) 编写中断服务函数

串口 2 中断服务函数为：USART2_IRQHandler，当发生中断的时候，程序就会执行中断服务函数，在这里就可以对接收到的数据进行处理，详见本例程源码。

6) 串口数据接收和发送

最后我们可以通过读写 USART_DR 寄存器，完成串口数据的接收和发送，HAL 库也给我们提供了：HAL_UART_Receive 和 HAL_UART_Transmit 两个函数用于串口数据的接收和发送。

大家可以根据实际情况选择使用哪种方式来收发串口数据。

37.3.2 程序流程图

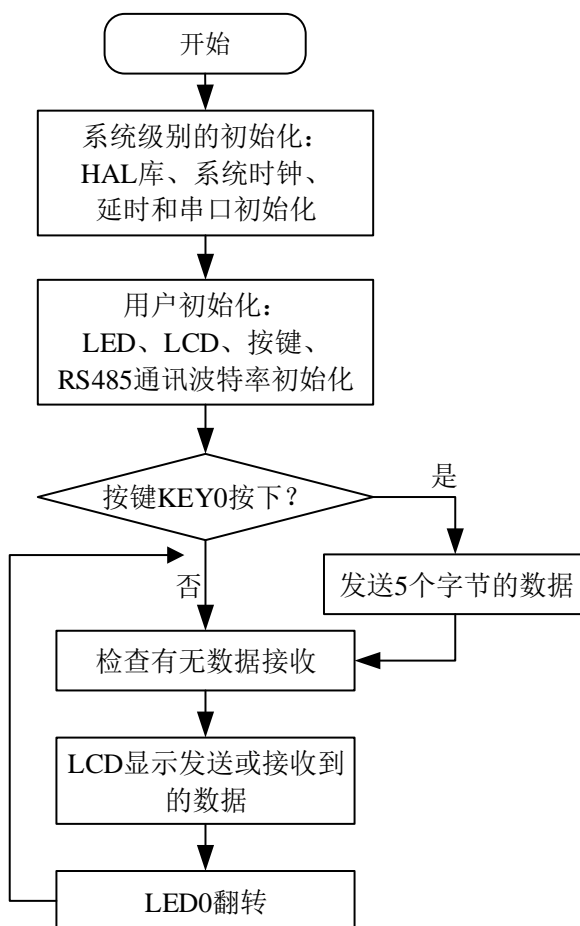


图 37.3.2.1 RS485 实验程序流程图

37.3.3 程序解析

1. RS485 驱动

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RS485 驱动相关源码包括两个文件：rs485.c 和 rs485.h。

为方便修改，我们在 rs485.h 中使用宏定义 485 相关的控制引脚和串口编号，如果需要使用其它的引脚或者串口，修改宏和串口的定义即可，它们在 rs485.h 中定义，它们列出如下：

```

/* RS485 引脚 和 串口 定义 */
#define RS485_RE_GPIO_PORT      GPIOD
#define RS485_RE_GPIO_PIN      GPIO_PIN_7
#define RS485_RE_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE();}while(0)

#define RS485_TX_GPIO_PORT      GPIOA
#define RS485_TX_GPIO_PIN      GPIO_PIN_2
#define RS485_TX_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE();}while(0)

#define RS485_RX_GPIO_PORT      GPIOA
#define RS485_RX_GPIO_PIN      GPIO_PIN_3
#define RS485_RX_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE();}while(0)

#define RS485_UX                USART2
#define RS485_UX_IRQn           USART2_IRQn
#define RS485_UX_IRQHandler     USART2_IRQHandler
#define RS485_UX_CLK_ENABLE()  do{ __HAL_RCC_USART2_CLK_ENABLE();}while(0)
    
```



```

/* 控制 RS485_RE 脚，控制 RS485 发送/接收状态
 * RS485_RE = 0，进入接收模式
 * RS485_RE = 1，进入发送模式
 */
#define RS485_RE(x) do{ x ? \
    HAL_GPIO_WritePin(RS485_RE_GPIO_PORT, RS485_RE_GPIO_PIN, \ GPIO_PIN_SET) : \
    HAL_GPIO_WritePin(RS485_RE_GPIO_PORT, RS485_RE_GPIO_PIN, \ GPIO_PIN_RESET); \
}while(0)

```

1) rs485_init 函数

rs485_init 的配置与串口类似，也需要设置波特率等参数，另外还需要配置收发模式的驱动引脚，我们的程序设计如下：

```

/**
 * @brief      RS485 初始化函数
 * @note      该函数主要是初始化串口
 * @param      baudrate: 波特率，根据自己需要设置波特率值
 * @retval     无
 */
void rs485_init(uint32_t baudrate)
{
    /* IO 及 时钟配置 */
    RS485_RE_GPIO_CLK_ENABLE(); /* 使能 RS485_RE 脚时钟 */
    RS485_TX_GPIO_CLK_ENABLE(); /* 使能 串口 TX 脚 时钟 */
    RS485_RX_GPIO_CLK_ENABLE(); /* 使能 串口 RX 脚 时钟 */
    RS485_UX_CLK_ENABLE(); /* 使能 串口 时钟 */

    GPIO_InitTypeDef gpio_initure;
    gpio_initure.Pin = RS485_TX_GPIO_PIN;
    gpio_initure.Mode = GPIO_MODE_AF_PP;
    gpio_initure.Pull = GPIO_PULLUP;
    gpio_initure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(RS485_TX_GPIO_PORT, &gpio_initure); /* 串口 TX 脚 模式设置 */

    gpio_initure.Pin = RS485_RX_GPIO_PIN;
    gpio_initure.Mode = GPIO_MODE_AF_INPUT;
    HAL_GPIO_Init(RS485_RX_GPIO_PORT, &gpio_initure); /* 串口 RX 脚设置成输入模式 */

    gpio_initure.Pin = RS485_RE_GPIO_PIN;
    gpio_initure.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_initure.Pull = GPIO_PULLUP;
    gpio_initure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(RS485_RE_GPIO_PORT, &gpio_initure); /* RS485_RE 脚 模式设置 */

    /* USART 初始化设置 */
    g_rs485_handler.Instance = RS485_UX; /* 选择 485 对应的串口 */
    g_rs485_handler.Init.BaudRate = baudrate; /* 波特率 */
    g_rs485_handler.Init.WordLength = UART_WORDLENGTH_8B; /* 字长为 8 位数据格式 */
    g_rs485_handler.Init.StopBits = UART_STOPBITS_1; /* 一个停止位 */
    g_rs485_handler.Init.Parity = UART_PARITY_NONE; /* 无奇偶校验位 */
    g_rs485_handler.Init.HwFlowCtl = UART_HWCONTROL_NONE; /* 无硬件流控 */
    g_rs485_handler.Init.Mode = UART_MODE_TX_RX; /* 收发模式 */
    HAL_UART_Init(&g_rs485_handler); /* 使能对应的串口 */

    /* 使能接收中断 */
    __HAL_UART_ENABLE_IT(&g_rs485_handler, UART_IT_RXNE); /* 开启接收中断 */
    HAL_NVIC_EnableIRQ(RS485_UX_IRQn); /* 使能 USART2 中断 */
    HAL_NVIC_SetPriority(RS485_UX_IRQn, 3, 3); /* 抢占优先级 3，子优先级 3 */

    RS485_RE(0); /* 默认为接收模式 */
}

```

可以看到代码基本跟串口的配置一样，只是多了收发控制引脚的配置。

2) 发送函数

发送函数用于输出 485 信号到 485 总线上，我的默认的 485 方式一般空闲时为接收状态，只有发送数据时我们才控制 485 芯片进入发送状态，发送完成后马上回到空闲接收状态，这样可以保证操作过程中 485 的数据丢失最小。我们实现的发送函数如下：

```
/**
 * @brief      RS485 发送 len 个字节
 * @param      buf : 发送区首地址
 * @param      len : 发送字节数 (为了和本代码接收匹配, 这里不要超过 RS485_REC_LEN 个字节)
 * @retval     无
 */
void rs485_send_data(uint8_t *buf, uint8_t len)
{
    RS485_RE(1);    /* 进入发送模式 */
    HAL_UART_Transmit(&g_rs485_handler, buf, len, 1000);    /* 串口 2 发送数据 */
    g_RS485_rx_cnt = 0;
    RS485_RE(0);    /* 进入接收模式 */
}
```

3) 485 接收中断函数

RS485 的接收就与串口中断一样了，不过要注意空闲时要切换回接收状态，否则会收不到数据。我们定义了一个全局的缓冲区 g_RS485_rx_buf 进行接收测试，通过串口中断接收数据，编写的接收代码如下：

```
uint8_t g_RS485_rx_buf[RS485_REC_LEN]; /* 接收缓冲, 最大 RS485_REC_LEN 个字节. */
uint8_t g_RS485_rx_cnt = 0;            /* 接收到的数据长度 */

void RS485_UX_IRQHandler(void)
{
    uint8_t res;

    if ((__HAL_UART_GET_FLAG(&g_rs485_handler, UART_FLAG_RXNE) != RESET))
    {
        /* 接收到数据 */
        HAL_UART_Receive(&g_rs485_handler, &res, 1, 1000);

        if (g_RS485_rx_cnt < RS485_REC_LEN)    /* 缓冲区未满 */
        {
            g_RS485_rx_buf[g_RS485_rx_cnt] = res;    /* 记录接收到的值 */
            g_RS485_rx_cnt++;    /* 接收数据增加 1 */
        }
    }
}
```

4) 485 查询接收数据函数

该函数用于查询 485 总线上接收到的数据，主要实现的逻辑是：一开始进入函数时，先记录下当前接收计数器的值，再来一个延时去判断接收是否结束（即该期间有无接收到数据），假如说接收计数器的值没有改变，就证明接收结束，我们就可以把当前接收缓冲区传递出去。函数实现如下：

```
void rs485_receive_data(uint8_t *buf, uint8_t *len)
{
    uint8_t rxlen = g_RS485_rx_cnt;
    uint8_t i = 0;
    *len = 0;    /* 默认为 0 */
    delay_ms(10);    /* 等待 10ms, 连续超过 10ms 没有接收到一个数据, 则认为接收结束 */

    if (rxlen == g_RS485_rx_cnt && rxlen) /* 接收到了数据, 且接收完成了 */
    {
        for (i = 0; i < rxlen; i++)
        {
            buf[i] = g_RS485_rx_buf[i];
        }

        *len = g_RS485_rx_cnt;    /* 记录本次数据长度 */
    }
}
```

```

    g_RS485_rx_cnt = 0;      /* 清零 */
}
}

```

RS485 的代码就讲到这里，基本是串口的知识，大家不明白的配置可以翻看之前串口章节的知识。

2. main.c 代码

在 main.c 中编写如下代码：

```

int main(void)
{
    uint8_t key;
    uint8_t i = 0, t = 0;
    uint8_t cnt = 0;
    uint8_t rs485buf[5];

    HAL_Init();                /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72);            /* 延时初始化 */
    usart_init(115200);        /* 串口初始化为 115200 */
    usmart_dev.init(72);       /* 初始化 USMART */
    led_init();                /* 初始化 LED */
    lcd_init();                /* 初始化 LCD */
    key_init();                /* 初始化按键 */
    rs485_init(9600);          /* 初始化 RS485 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "RS485 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Send", RED); /* 显示提示信息 */

    lcd_show_string(30, 130, 200, 16, 16, "Count:", RED); /* 显示当前计数值 */
    lcd_show_string(30, 150, 200, 16, 16, "Send Data:", RED); /* 提示发送的数据 */
    lcd_show_string(30, 190, 200, 16, 16, "Receive Data:", RED); /* 提示收到的数据 */

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES) /* KEY0 按下, 发送一次数据 */
        {
            for (i = 0; i < 5; i++)
            {
                rs485buf[i] = cnt + i; /* 填充发送缓冲区 */
                /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 170, rs485buf[i], 3, 16, 0X80, BLUE);
            }
            rs485_send_data(rs485buf, 5); /* 发送 5 个字节 */
        }
        rs485_receive_data(rs485buf, &key);
        if (key) /* 接收到有数据 */
        {
            if (key > 5) key = 5; /* 最大是 5 个数据. */

            for (i = 0; i < key; i++)
            {
                /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 210, rs485buf[i], 3, 16, 0X80, BLUE);
            }
        }

        t++;
        delay_ms(10);
    }
}

```

```

if (t == 20)
{
    LED0_TOGGLE(); /* LED0 闪烁, 提示系统正在运行 */
    t = 0;
    cnt++;
    lcd_show_xnum(30 + 48, 130, cnt, 3, 16, 0X80, BLUE); /* 显示数据 */
}
}
}

```

我们是通过按键控制数据的发送。在此部分代码中, cnt 是一个累加数, 一旦 KEY0 按下, 就以这个数位基准连续发送 5 个数据。当 485 总线收到数据的时候, 就将收到的数据直接显示在 LCD 屏幕上。

37.4 下载验证

在代码编译成功之后, 我们通过下载代码到正点原子战舰 STM32F103 上 (注意要 2 个开发板都下载这个代码), 得到如图 37.4.1 所示:

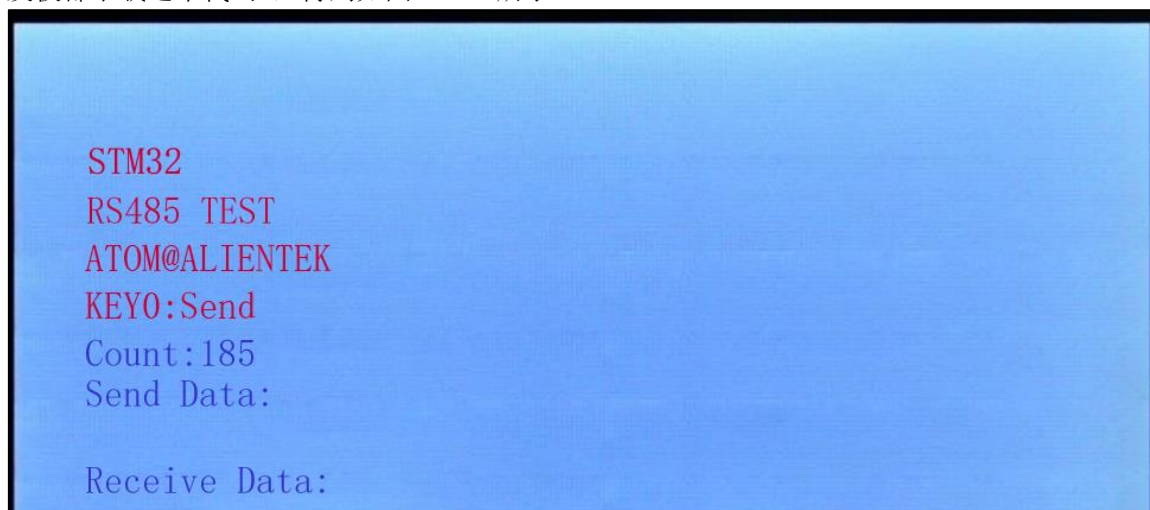


图 37.4.1 程序运行效果图

伴随 DS0 的不停闪烁, 提示程序在运行。此时, 我们按下 KEY0 就可以在另外一个开发板上收到这个开发板发送的数据了。如图 37.4.2 和图 37.4.3 所示:

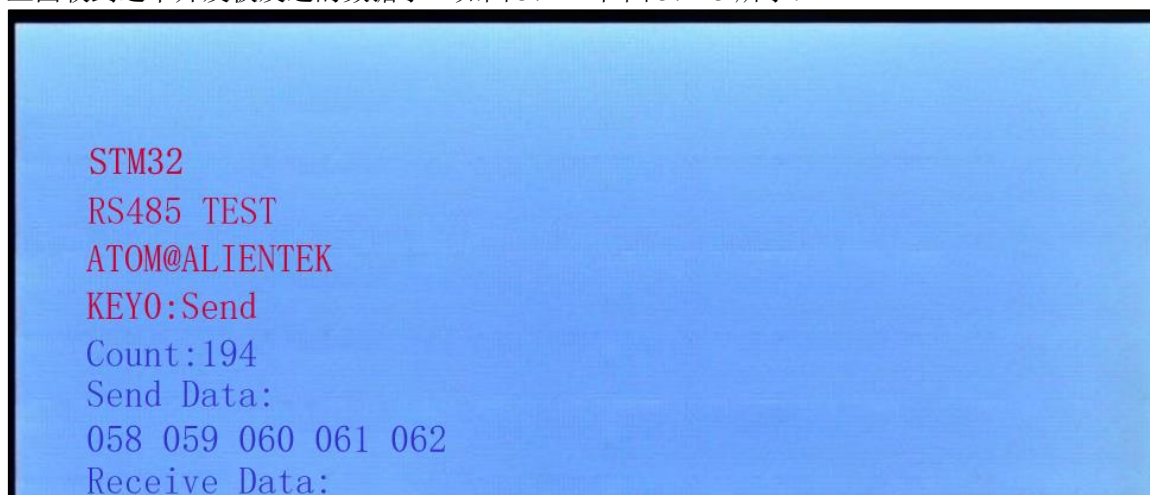


图 37.4.2 发送 RS485 数据的开发板界面

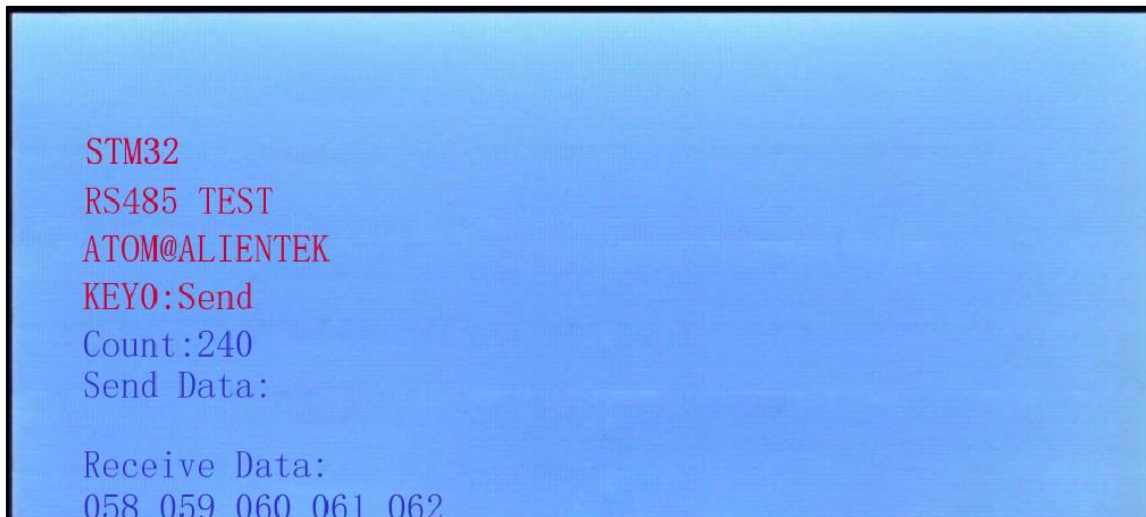


图 37.4.3 接收 RS485 数据的开发板

图 37.4.2 来自开发板 A，发送了 5 个数据，图 37.4.3 来自开发板 B，接收到了来自开发板 A 的 5 个数据。

本章介绍的 485 总线是通过串口控制收发的，我们只需要将 P7 的跳线帽稍作改变(将 PA2/PA3 连接 COM2_RX/COM2_TX)，该实验就变成了一个 RS232 串口通信实验了，通过对接两个开发板的 RS232 接口，即可得到同样的实验现象，不过 RS232 不需要使能脚，有兴趣的读者可以实验一下。

另外，利用 USART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十八章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32 自带的 CAN 控制器来实现 CAN 的收发功能，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 38.1 CAN 简介：
- 38.2 硬件设计
- 38.3 程序设计
- 38.4 下载验证

38.1 CAN 总线简介

38.1.1 CAN 简介

CAN 是 Controller Area Network 的缩写（以下称为 CAN），是 ISO 国际标准化的串行通信协议。在当前的汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，现在在欧洲已是汽车网络的标准协议。

现在，CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。现场总线是当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

CAN 协议具有一下特点：

- ① **多主控制。**在总线空闲时，所有单元都可以发送消息（多主控制），而两个以上的单元同时开始发送消息时，根据标识符（Identifier 以下称为 ID）决定优先级。ID 并不是表示发送的目的地址，而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时，对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜（被判定为优先级最高）的单元可继续发送消息，仲裁失利的单元则立刻停止发送而进行接收工作。
- ② **系统的柔软性。**与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时，连接在总线上的其它单元的软硬件及应用层都不需要改变。
- ③ **通信速度较快，通信距离远。**最高 1Mbps（距离小于 40M），最远可达 10KM（速率低于 5Kbps）。
- ④ **具有错误检测、错误通知和错误恢复功能。**所有单元都可以检测错误（错误检测功能），检测出错误的单元会立即同时通知其他所有单元（错误通知功能），正在发送消息的单元一旦检测出错误，会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止（错误恢复功能）。
- ⑤ **故障封闭功能。**CAN 可以判断出错误的类型是总线上暂时的数据错误（如外部噪声等）还是持续的数据错误（如单元内部故障、驱动器故障、断线等）。由此功能，当总线上发生持续数据错误时，可将引起此故障的单元从总线上隔离出去。
- ⑥ **连接节点多。**CAN 总线是可同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受总线上的时间延迟及电气负载的限制。降低通信速度，可连接的单元数增加；提高通信速度，则可连接的单元数减少。

正是因为 CAN 协议的这些特点，使得 CAN 特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

CAN 协议经过 ISO 标准化后有两个标准：ISO11898 标准（高速 CAN）和 ISO11519-2 标准（低速 CAN）。其中 ISO11898 是针对通信速率为 125Kbps~1Mbps 的高速通信标准，而 ISO11519-2 是针对通信速率为 125Kbps 以下的低速通信标准。

本章，我们使用的是 ISO11898 标准，也就是高速 CAN，其拓扑图如图 38.1.1.1 所示。

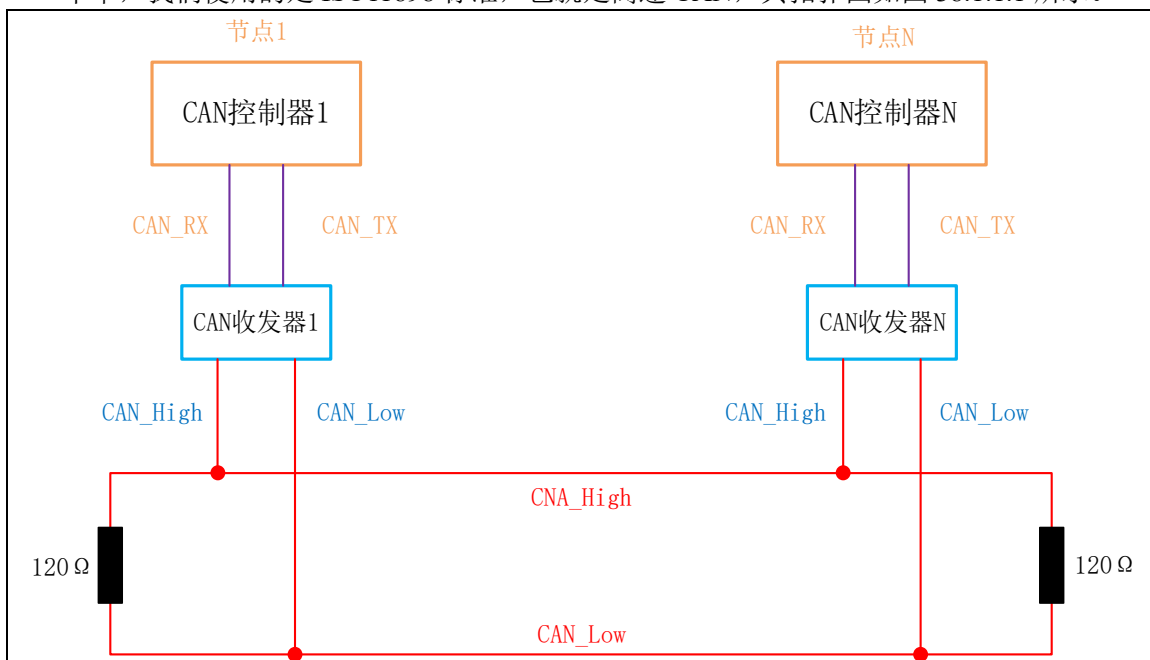


图 38.1.1 高速 CAN 拓扑结构图

从上图可知，高速 CAN 总线呈现的是一个闭环结构，总线是由两根线 CAN_High 和 CAN_Low 组成，且在总线两端各串联了 120Ω 的电阻（用于阻抗匹配，减少回波反射），同时总线上可以挂载多个节点。每个节点都有 CAN 收发器以及 CAN 控制器，CAN 控制器通常是 MCU 的外设，集成在芯片内部；CAN 收发器则需要外加芯片转换电路。

CAN 类似 RS485 也是通过差分信号传输数据。根据 CAN 总线上两根线的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。这是属于物理层特征，ISO11898 物理层特性如图 38.1.1.2 所示：

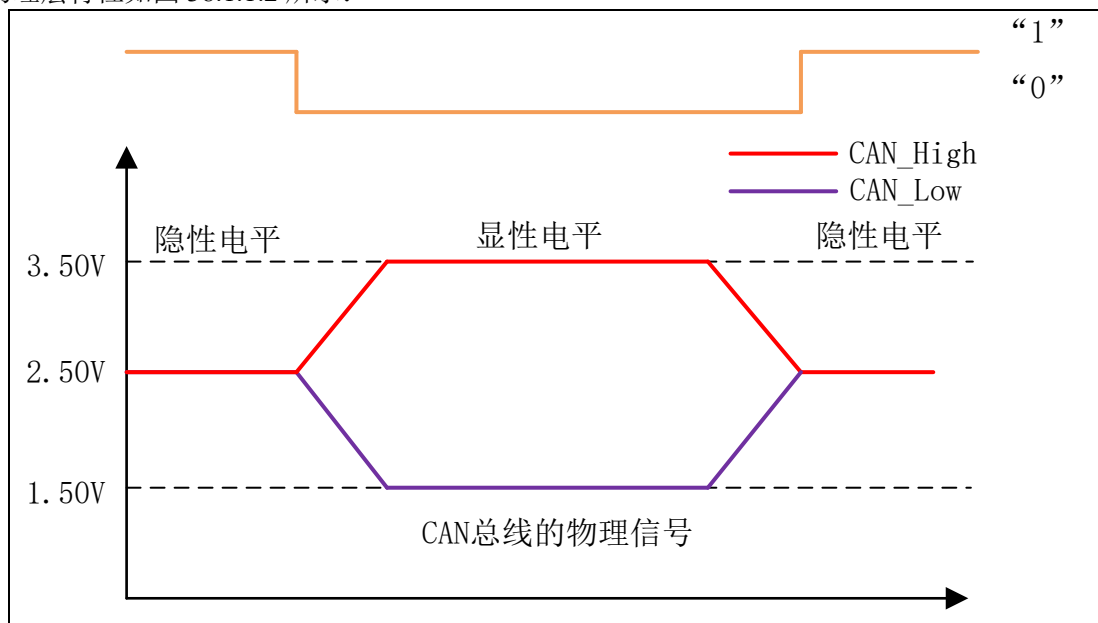


图 38.1.1.2 ISO11898 物理层特性

从该特性可以看出，显性电平对应逻辑 0，CAN_H 和 CAN_L 之差为 2V 左右。而隐性电平对应逻辑 1，CAN_H 和 CAN_L 之差为 0V。在总线上显性电平具有优先权，只要有一个单元输出显性电平，总线上即为显性电平。而隐性电平则具有包容的意味，只有所有的单元都输出隐性电平，总线上才为隐性电平（显性电平比隐性电平更强）。

38.1.2 CAN 协议

CAN 协议是通过以下 5 种类型的帧进行的：

- 数据帧
- 遥控帧
- 错误帧
- 过载帧
- 间隔帧

另外，数据帧和遥控帧有标准格式和扩展格式两种格式。标准格式有 11 个位的标识符(ID)，扩展格式有 29 个位的 ID。各种帧的用途如表 38.1.2.1 所示：

| 帧类型 | 帧用途 |
|-----|--------------------------|
| 数据帧 | 用于发送单元向接收单元传送数据的帧 |
| 遥控帧 | 用于接收单元向具有相同ID的发送单元请求数据的帧 |
| 错误帧 | 用于当检测出错误时向其它单元通知错误的帧 |
| 过载帧 | 用于接收单元通知其尚未做好接收准备的帧 |
| 间隔帧 | 用于将数据帧及遥控帧与前面的帧分离开来的帧 |

表 38.1.2.1 CAN 协议各种帧及其用途

由于篇幅所限，我们这里仅对数据帧进行详细介绍，数据帧一般由 7 个段构成，即：

帧起始。表示数据帧开始的段。

仲裁段。表示该帧优先级的段。

控制段。表示数据的字节数及保留位的段。

数据段。数据的内容，一帧可发送 0~8 个字节的数据。

CRC 段。检查帧的传输错误的段。

ACK 段。表示确认正常接收的段。

帧结束。表示数据帧结束的段。

数据帧的构成如图 38.1.2.1 所示：

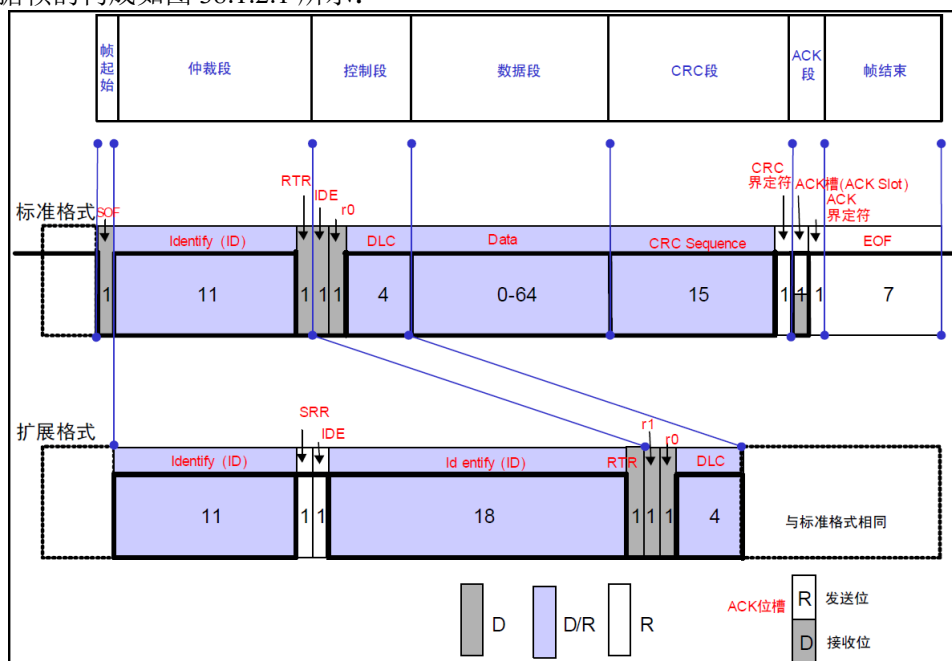


图 38.1.2.1 数据帧的构成

图中 D 表示显性电平，R 表示隐性电平（下同）。

帧起始，这个比较简单，标准帧和扩展帧都是由 1 个位的显性电平表示帧起始。

仲裁段，表示数据优先级的段，标准帧和扩展帧格式在本段有所区别，如图 38.1.2.2 所示：

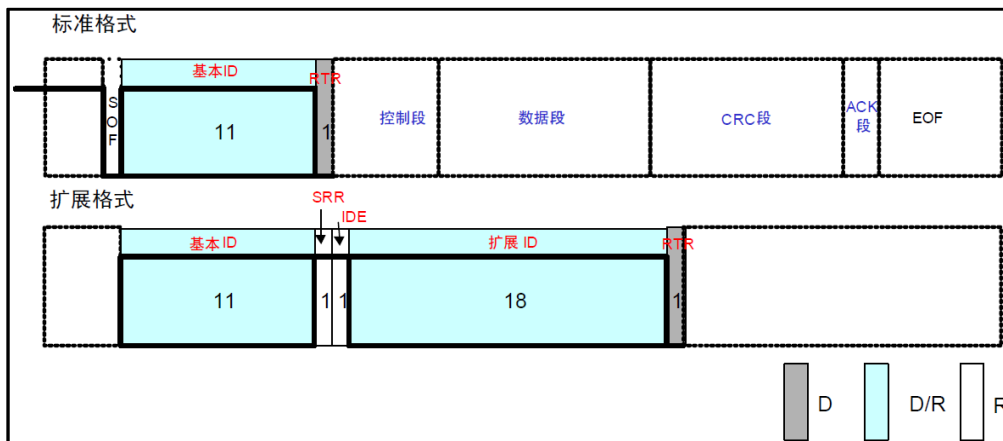


图 38.1.2.2 数据帧仲裁段构成

标准格式的 ID 有 11 个位。禁止高 7 位都为隐性（禁止设定：ID=111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由 ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性（禁止设定：基本 ID=111111XXXX）。

其中 RTR 位用于标识是否是远程帧（0，数据帧；1，远程帧），IDE 位为标识符选择位（0，使用标准标识符；1，使用扩展标识符），SRR 位为代替远程请求位，为隐性位，它代替了标准帧中的 RTR 位。

控制段，由 6 个位构成，表示数据段的字节数。标准帧和扩展帧的控制段稍有不同，如图 38.1.2.3 所示：

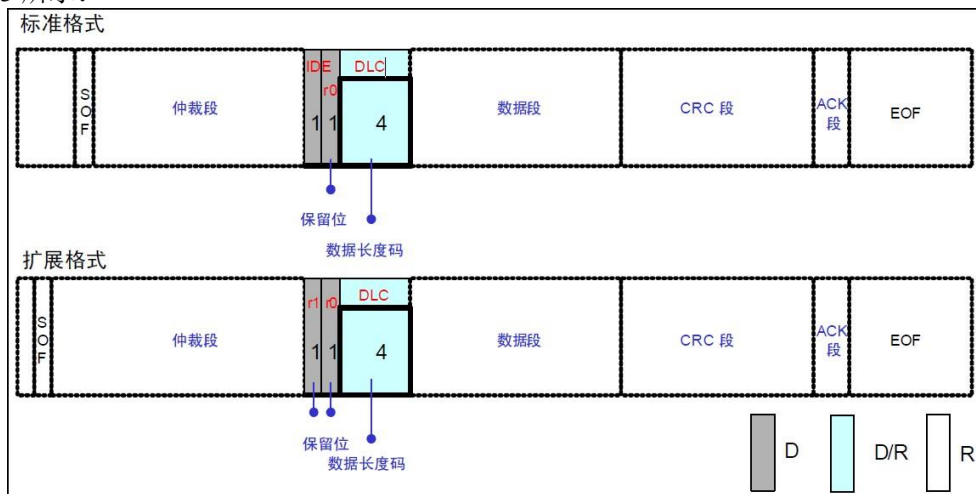


图 38.1.2.3 数据帧控制段构成

上图中，r0 和 r1 为保留位，必须全部以显性电平发送，但是接收端可以接收显性、隐性及任意组合的电平。DLC 段为数据长度表示段，高位在前，DLC 段有效值为 0~8，但是接收方接收到 9~15 的时候并不认为是错误。

数据段，该段可包含 0~8 个字节的数据。从最高位（MSB）开始输出，标准帧和扩展帧在这个段的定义都是一样的。如图图 38.1.2.4 所示：



图 38.1.2.4 数据帧数据段构成

CRC 段，该段用于检查帧传输错误。由 15 个位的 CRC 顺序和 1 个位的 CRC 界定符（用于分隔的位）组成，标准帧和扩展帧在这个段的格式也是相同的。如图 38.1.2.5 所示：

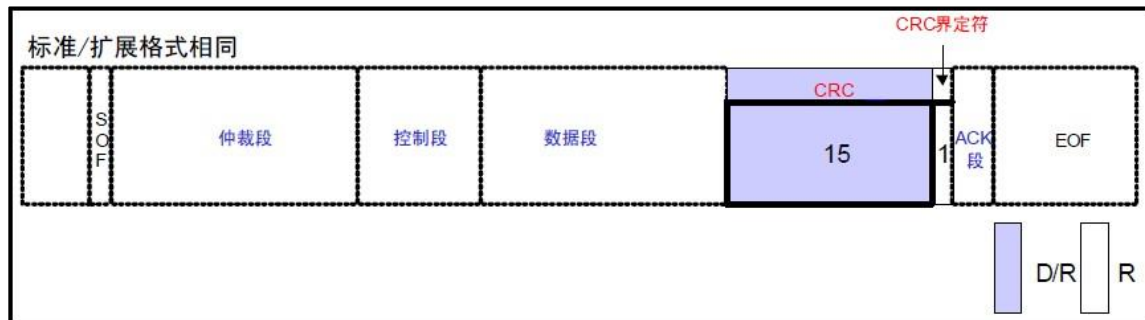


图 38.1.2.5 数据帧 CRC 段构成

此段 CRC 的值计算范围包括：帧起始、仲裁段、控制段、数据段。接收方以同样的算法计算 CRC 值并进行比较，不一致时会通报错误。

ACK 段，此段用来确认是否正常接收。由 ACK 槽(ACKSlot)和 ACK 界定符 2 个位组成。标准帧和扩展帧在这个段的格式也是相同的。如图 38.1.2.6 所示：



图 38.1.2.6 数据帧 ACK 段构成

发送单元的 ACK，发送 2 个位的隐性位，而接收到正确消息的单元在 ACK 槽（ACKSlot）发送显性位，通知发送单元正常接收结束，这个过程叫发送 ACK/返回 ACK。发送 ACK 的是在既不处于总线关闭态也不处于休眠态的所有接收单元中，接收到正常消息的单元（发送单元不发送 ACK）。所谓正常消息是指不含填充错误、格式错误、CRC 错误的消息。

帧结束，这个段也比较简单，标准帧和扩展帧在这个段格式一样，由 7 个位的隐性位组成。至此，数据帧的 7 个段就介绍完了，其他帧的介绍，请大家参考光盘的《CAN 入门书.pdf》相关章节。接下来，我们再来看看 CAN 的位时序。

由发送单元在非同步的情况下发送的每秒钟的位数称为位速率。一个位可分为 4 段。

- 同步段（SS）
- 传播时间段（PTS）
- 相位缓冲段 1（PBS1）
- 相位缓冲段 2（PBS2）

这些段又由可称为 Time Quantum（以下称为 Tq）的最小时间单位构成。

1 位分为 4 个段，每个段又由若干个 Tq 构成，这称为位时序。

1 位由多少个 Tq 构成、每个段又由多少个 Tq 构成等，可以任意设定位时序。通过设定位时序，多个单元可同时采样，也可任意设定采样点。各段的作用和 Tq 数如表 38.1.2.2 所示：

| 段名称 | 段的作用 | Tq 数 | |
|--|---|-------|--------|
| 同步段 (SS: Synchronization Segment) | 多个连接在总线上的单元通过此段实现时序调整，同步进行接收和发送的工作。由隐性电平到显性电平的边沿或由显性电平到隐性电平边沿最好出现在此段中。 | 1Tq | 8~25Tq |
| 传播时间段 (PTS: Propagation Time Segment) | 用于吸收网络上的物理延迟的段。 所谓的网络的物理延迟指发送单元的输出延迟、总线上信号的传播延迟、接收单元的输入延迟。 这个段的时间为以上各延迟时间的和的两倍。 | 1~8Tq | |
| 相位缓冲段 1 (PBS1: Phase Buffer Segment 1) | 当信号边沿不能被包含于 SS 段中时，可在此段进行补偿。 | 1~8Tq | |
| 相位缓冲段 2 (PBS2: Phase Buffer Segment 2) | 由于各单元以各自独立的时钟工作，细微的时钟误差会累积起来，PBS 段可用于吸收此误差。 通过对相位缓冲段加减 SJW 吸收误差。 SJW 加大后允许误差加大，但通信速度下降。 | 2~8Tq | 1~4Tq |
| 再同步补偿宽度 (SJW: reSynchronization Jump Width) | 因时钟频率偏差、传送延迟等，各单元有同步误差。SJW 为补偿此误差的最大值。 | 1~4Tq | |

表 38.1.2.2 一个位各段及其作用

1 个位的构成如图 38.1.2.7 所示：

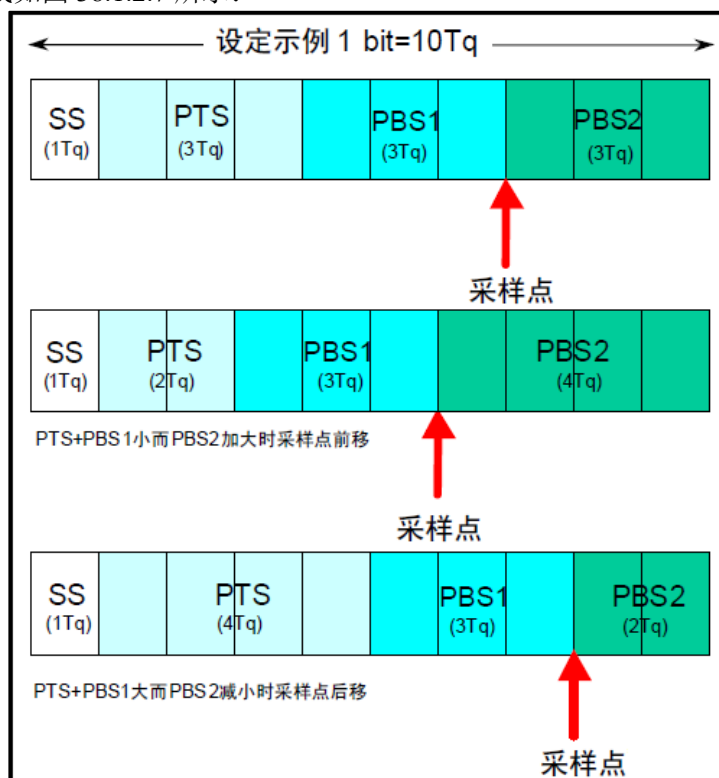


图 38.1.2.7 一个位的构成

上图的采样点，是指读取总线电平，并将读到的电平作为位值的点。位置在 PBS1 结束处。根据这个位时序，我们就可以计算 CAN 通信的波特率了。具体计算方法，我们等下再介绍，前面提到的 CAN 协议具有仲裁功能，下面我们来看看是如何实现的。

在总线空闲态，最先开始发送消息的单元获得发送权。当多个单元同时开始发送时，各发送单元从仲裁段的第一位开始进行仲裁。连续输出显性电平最多的单元可继续发送。实现过程，如图 38.1.2.8 所示：

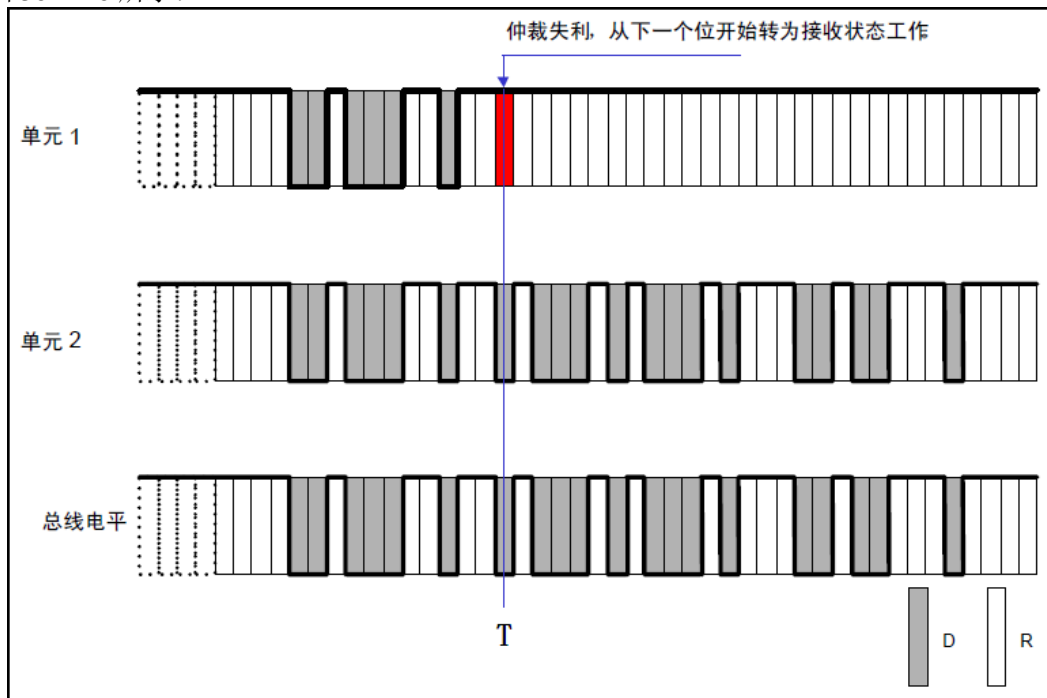


图 38.1.2.8 CAN 总线仲裁过程

上图中，单元 1 和单元 2 同时开始向总线发送数据，开始部分他们的数据格式是一样的，故无法区分优先级，直到 T 时刻，单元 1 输出隐性电平，而单元 2 输出显性电平，此时单元 1 仲裁失利，立刻转入接收状态工作，不再与单元 2 竞争，而单元 2 则顺利获得总线使用权，继续发送自己的数据。这就实现了仲裁，让连续发送显性电平多的单元获得总线使用权。

通过以上介绍，我们对 CAN 总线有了个大概了解（详细介绍参考光盘的：《CAN 入门书.pdf》），接下来我们了解下 STM32F1 的 CAN 控制器。

STM32F1 自带的是 bxCAN，即基本扩展 CAN。它支持 CAN 协议 2.0A 和 2.0B。CAN2.0A 只能处理标准数据帧，扩展帧的内容会识别错误；CAN2.0B Active 可以处理标准数据帧和扩展数据帧；而 CAN2.0B passive 只能处理标准数据帧，扩展帧的内容会忽略。它的设计目标是，以最小的 CPU 负荷来高效处理大量收到的报文。它也支持报文发送的优先级要求（优先级特性可软件配置）。对于安全紧要的应用，bxCAN 提供所有支持时间触发通信模式所需的硬件功能。

STM32F1 的 bxCAN 的主要特点有：

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高达 1Mbps
- 支持时间触发通信
- 具有 3 个发送邮箱
- 具有 3 级深度的 2 个接收 FIFO
- 可变的过滤器组（最多 28 个）

在 STM32 互联型产品中，带有 2 个 CAN 控制器，而我们使用的 STM32F103ZET6 属于增强型，不是互联型，只有 1 个 CAN 控制器。双 CAN 的框图如图 38.1.2.9 所示：

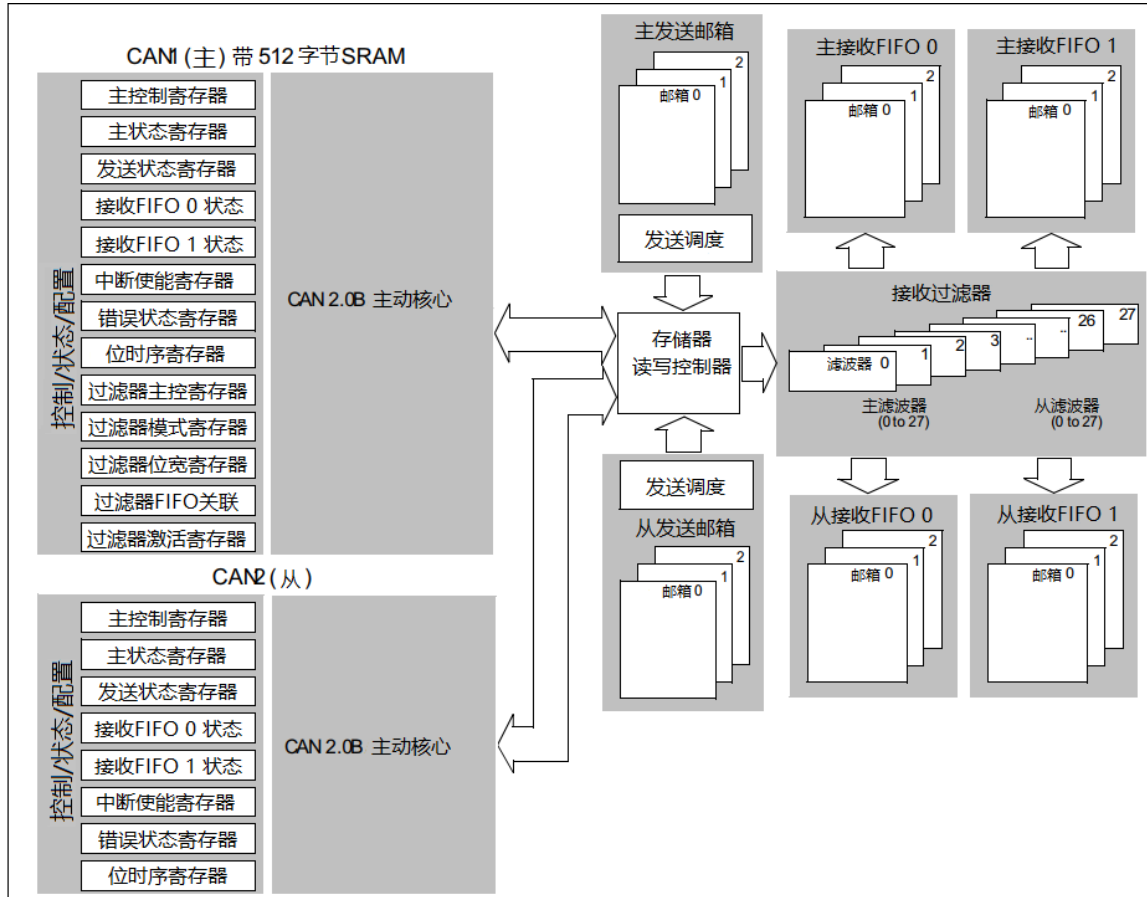


图 38.1.2.9 双 CAN 框图

从图中可以看出两个 CAN 都分别拥有自己的发送邮箱和接收 FIFO，但是他们共用 28 个过滤器。通过 CAN_FMR 寄存器的设置，可以设置过滤器的分配方式。

STM32 的标识符过滤比较复杂，它的存在减少了 CPU 处理 CAN 通信的开销。STM32 的过滤器组最多有 28 个（互联型），但是 STM32F103ZET6 只有 14 个（增强型），每个滤波器组 x 由 2 个 32 为寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32F1 每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

- 1 个 32 位过滤器，包括：STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位
- 2 个 16 位过滤器，包括：STDID[10:0]、IDE、RTR 和 EXTID[17:15]位

此外过滤器可配置为，屏蔽位模式和标识符列表模式。

在屏蔽位模式下，标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照“必须匹配”或“不用关心”处理。

而在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤器标识符相同。

通过 CAN_FMR 寄存器，可以配置过滤器组的位宽和工作模式，如图 38.1.2.10 所示：

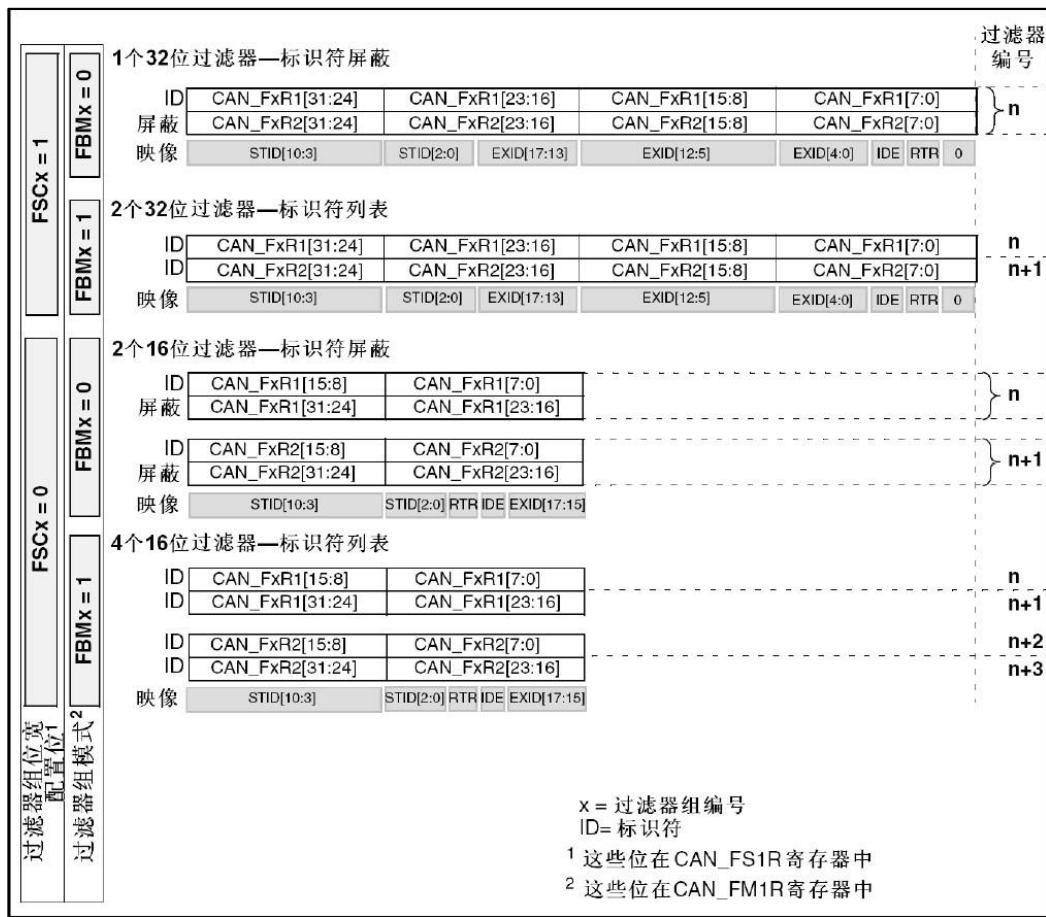


图 38.1.2.10 过滤器组位宽模式设置

为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。

为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。应用程序不用的过滤器组，应该保持在禁用状态。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号，图 38.1.2.10 中的 n)从 0 开始，到某个最大数值—取决于过滤器组的模式和位宽的设置。

举个简单的例子，我们设置过滤器组 0 工作在：1 个 32 为位过滤器-标识符屏蔽模式，然后设置 $CAN_F0R1=0XFFFF0000$ ， $CAN_F0R2=0XFF00FF00$ 。其中存放到 CAN_F0R1 的值就是期望收到的 ID，即我们希望收到的映像（STID+EXTID+IDE+RTR）最好是：0XFFFF0000。而 0XFF00FF00 就是设置我们需要必须关心的 ID，表示收到的映像，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_F0R1 中对应的位一模一样，而另外的 16 个位则不关心，可以一样，也可以不一样，都认为是正确的 ID，即收到的映像必须是 0XFFxx00xx，才算是正确的（ x 表示不关心）。这里需要注意的是标识符选择位 IDE 和帧类型 RTR 需要一致。其情况如下图 38.1.2.11 所示：

| 32位过滤器-标识符屏蔽模式（过滤出一组标识符） | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------------------------------|------------|----|----|----|----|----|----|----|-----------|----|----|----|-------------|----|----|----|------------|----|----|----|----|----|---|---|-----------|---|---|---|-----|-----|---|---|
| bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ID CAN_F0R1 (0xFFFF0000) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 屏蔽 CAN_F0R2 (0xFF00FF00) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 映像 | STID[10:3] | | | | | | | | STID[2:0] | | | | EXID[17:13] | | | | EXID[12:5] | | | | | | | | EXID[4:0] | | | | IDE | RTR | 0 | |
| 过滤出ID | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | X | X | 0 | 0 | 0 |

图 38.1.2.11 过滤器举例图

关于标识符过滤的详细介绍，请参考《STM32F10xxx 参考手册_V10(中文版).pdf》的 22.7.4 节（431 页）。

接下来，我们看看 STM32 的 CAN 发送和接收的流程。

CAN 发送流程

CAN 发送流程为：程序选择 1 个空置的邮箱（TME=1）→设置标识符（ID），数据长度和发送数据→设置 CAN_TxR 的 TXRQ 位为 1，请求发送→邮箱挂号（等待成为最高优先级）→预定发送（等待总线空闲）→发送→邮箱空置。整个流程如图 38.1.2.12 所示：

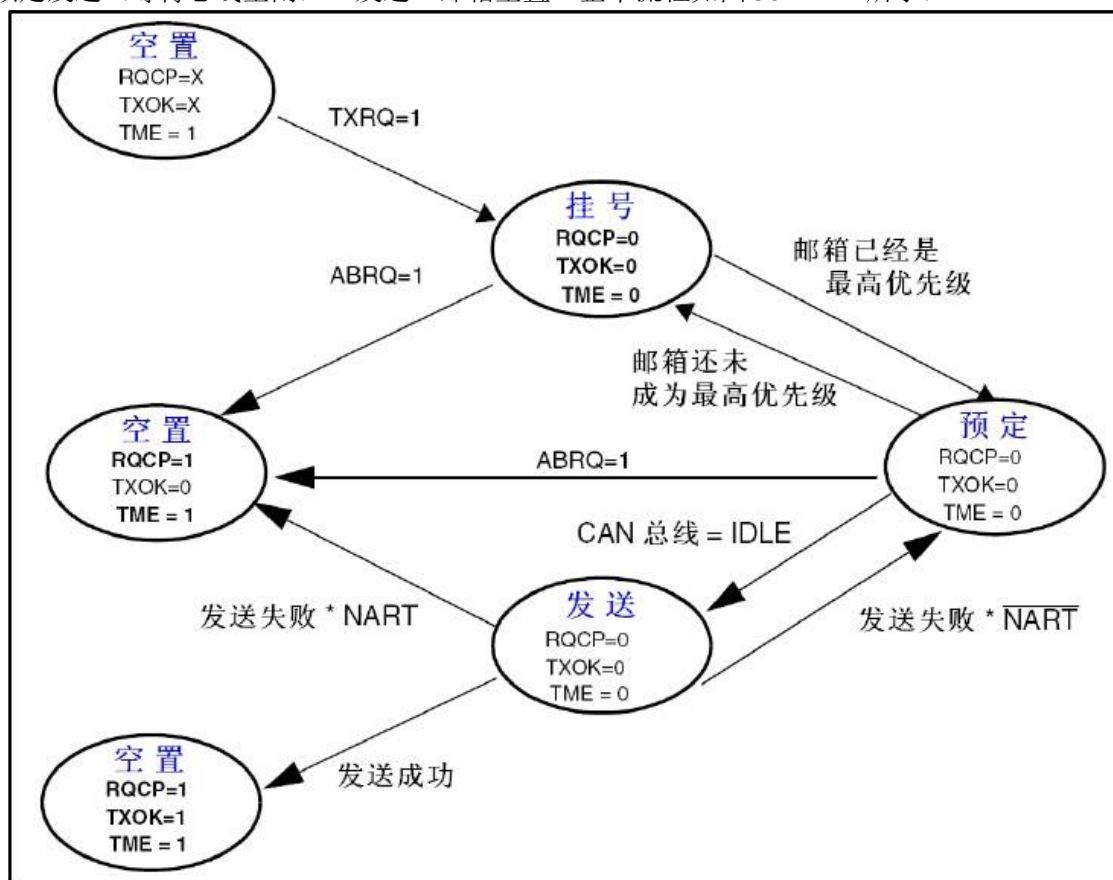


图 38.1.2.12 发送流程图

上图中，还包含了很多其他处理，不强制退出发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

CAN 接收流程

CAN 接收到的有效报文，被存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个过滤器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 38.1.2.13 所示：

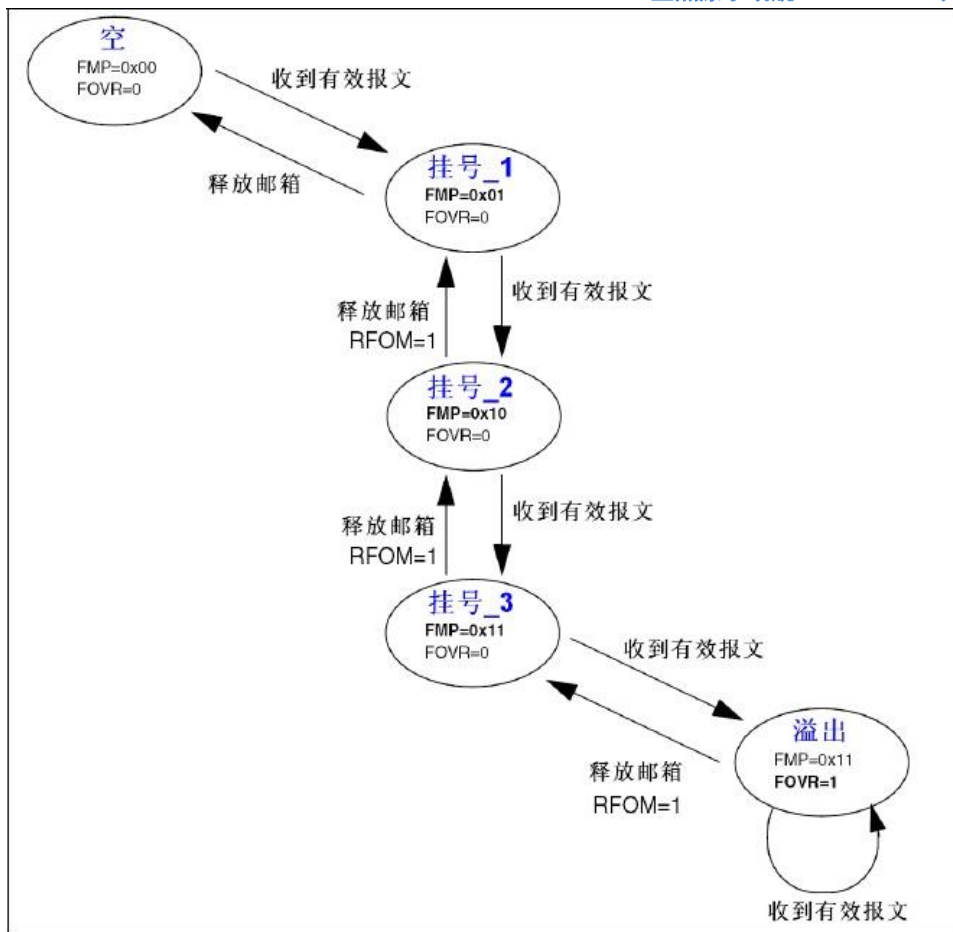


图 38.1.2.13 FIFO 接收报文流程图

FIFO 接收到的报文数，我们可以通过查询 CAN_RFR 的 FMP 寄存器来得到，只要 FMP 不为 0，我们就可以从 FIFO 读出收到的报文。

接下来，我们简单看看 STM32 的 CAN 位时间特性，STM32 的 CAN 位时间特性和之前我们介绍的 CAN 协议中，稍有点区别。STM32 把传播时间段和相位缓冲段 1（STM32 称之为时间段 1）合并了，所以 STM32 的 CAN 一个位只有 3 段：同步段（SYNC_SEG）、时间段 1（BS1）和时间段 2（BS2）。STM32 的 BS1 段可以设置为 1~16 个时间单元，刚好等于我们上面介绍的传播时间段和相位缓冲段 1 之和。STM32 的 CAN 位时序如图 38.1.2.14 所示：

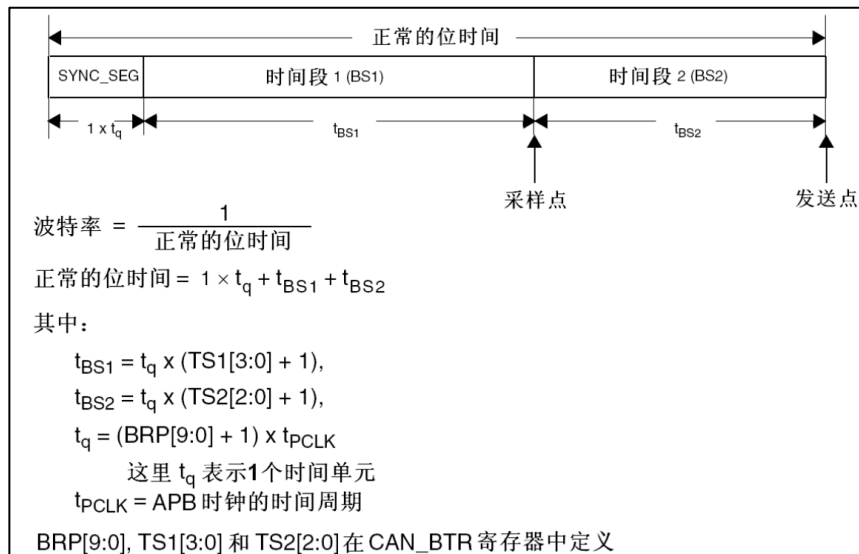


图 38.1.2.14 STM32 CAN 位时序

38.1.3 CAN 寄存器

● CAN 的主控制寄存器 (CAN MCR)

| | | | | | | | | | | | | | | | |
|-------|--|----|----|----|----|----|----|------|------|------|------|------|------|-------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | DBF |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RESET | 保留 | | | | | | | TTCM | ABOM | AWUM | NART | RFLM | TXFP | SLEEP | INRQ |
| rs | res | | | | | | | rw | rw | rw | rw | rw | rw | rw | rw |
| 位0 | INRQ: 初始化请求 (Initialization request) 软件对该位清'0'可使CAN从初始化模式进入正常工作模式：当CAN在接收引脚检测到连续的11个隐性位后，CAN就达到同步，并为接收和发送数据作好准备了。为此，硬件相应地对CAN_MSR寄存器的INAK位清'0'。 软件对该位置1可使CAN从正常工作模式进入初始化模式：一旦当前的CAN活动(发送或接收)结束，CAN就进入初始化模式。相应地，硬件对CAN_MSR寄存器的INAK位置'1'。 | | | | | | | | | | | | | | |

该寄存器的详细描述，请参考《STM32F10xxx 参考手册 V10（中文版）.pdf》的 439 页。

● CAN 位时序寄存器 (CAN BTR)

| | | | | | | | | | | | | | | | |
|--------|------|---|----|----|----|----------|-----|-----|----------|-----|-----|----------|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| SILM | LBKM | 保留 | | | | SJW[1:0] | | 保留 | TS2[2:0] | | | TS1[3:0] | | | |
| r/w | r/w | res | | | | r/w | r/w | res | r/w | r/w | r/w | r/w | r/w | r/w | r/w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | BRP[9:0] | | | | | | | | | |
| res | | | | | | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w |
| 位31 | | SILM: 静默模式(用于调试) (Silent mode (debug)) 0: 正常状态; 1: 静默模式。 | | | | | | | | | | | | | |
| 位30 | | LBKM: 环回模式(用于调试) (Loop back mode (debug)) 0: 禁止环回模式; 1: 允许环回模式。 | | | | | | | | | | | | | |
| 位25:24 | | SJW[1:0]: 重新同步跳跃宽度 (Resynchronization jump width) 为了重新同步，该位域定义了CAN硬件在每位中可以延长或缩短多少个时间单元的上限。 $t_{RJW} = t_{CAN} \times (SJW[1:0] + 1)$ 。 | | | | | | | | | | | | | |
| 位22:20 | | TS2[2:0]: 时间段2 (Time segment 2) 该位域定义了时间段2占用了多少个时间单元 $t_{BS2} = t_{CAN} \times (TS2[2:0] + 1)$ 。 | | | | | | | | | | | | | |
| 位19:16 | | TS1[3:0]: 时间段1 (Time segment 1) 该位域定义了时间段1占用了多少个时间单元 $t_{BS1} = t_{CAN} \times (TS1[3:0] + 1)$ 关于位时间特性的详细信息，请参考22.7.7节位时间特性。 | | | | | | | | | | | | | |
| 位9:0 | | BRP[9:0]: 波特率分频器 (Baud rate prescaler) 该位域定义了时间单元(t_q)的时间长度 $t_q = (BRP[9:0]+1) \times t_{CLK}$ | | | | | | | | | | | | | |

587

CAN 位时序寄存器用于设置分频、Tbs1、Tbs2 以及 Tsjw 等非常重要的参数，直接决定了 CAN 的波特率。

另外该寄存器还可以设置 CAN 的测试模式，STM32 提供了三种测试模式，环回模式、静默模式和环回静默模式。这里我们简单介绍下环回模式。在环回模式下，bxCAN 把发送的报文当作接收的报文并保存(如果可以通过接收过滤器组)在接收 FIFO 的输出邮箱里。也就是环回模式是一个自发自收的模式，如图 38.1.3.3 所示：

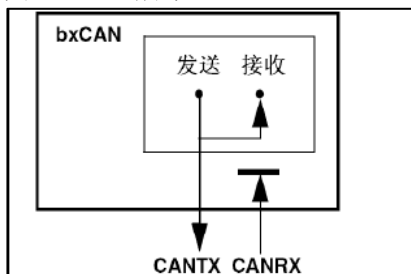


图 38.1.3.3 CAN 环回模式

环回模式可用于自测试。为了避免外部的影响，在环回模式下 CAN 内核忽略确认错误(在数据/远程帧的确认位时刻，不检测是否有显性位)。在环回模式下，bxCAN 在内部把 Tx 输出回馈到 Rx 输入上，而完全忽略 CANRX 引脚的实际状态。发送的报文可以在 CANTX 引脚上检测到。

● CAN 发送邮箱标识符寄存器 (CAN_TlRx)

CAN 发送邮箱标识符寄存器(CAN_TlRx)(x=0~3)，该寄存器各位描述如图 38.1.3.4 所示：

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|------------------------|----|--|----|----|----|----|----|----|----|----|-------------|----|-----|-----|------|
| STID[10:0]/EXID[28:18] | | | | | | | | | | | EXID[17:13] | | | | |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| EXID[12:0] | | | | | | | | | | | | | IDE | RTR | TXRQ |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 位31:21 | | STID[10:0]/EXID[28:18]: 标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据IDE位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。 | | | | | | | | | | | | | |
| 位20:3 | | EXID[17:0]: 扩展标识符 (Extended identifier) 扩展身份标识的低字节。 | | | | | | | | | | | | | |
| 位2 | | IDE: 标识符选择 (Identifier extension) 该位决定发送邮箱中报文使用的标识符类型 0: 使用标准标识符; 1: 使用扩展标识符。 | | | | | | | | | | | | | |
| 位1 | | RTR: 远程发送请求 (Remote transmission request) 0: 数据帧; 1: 远程帧。 | | | | | | | | | | | | | |
| 位0 | | TXRQ: 发送数据请求 (Transmit mailbox request) 由软件对其置'1'，来请求发送邮箱的数据。当数据发送完成，邮箱为空时，硬件对其清'0'。 | | | | | | | | | | | | | |

图 38.1.3.4 寄存器 CAN_TlRx 各位描述

该寄存器主要用来设置标识符(包括扩展标识符)，另外还可以设置帧类型，通过 TXRQ(位 0)值 1，来请求邮箱发送。因为有 3 个发送邮箱，所以寄存器 CAN_TlRx 有 3 个。

● CAN 发送邮箱数据长度和时间戳寄存器(CAN_TDTxR)

CAN 发送邮箱数据长度和时间戳寄存器(CAN_TDTxR)(x=0~2)，该寄存器我们本章仅用来设置数据长度，即最低 4 个位。低 4 位的描述如图 38.1.3.5 所示。

| | | | | | | | | | | | | | | | | |
|------------|----|---|----|----|----|----|-----|-----|----|----|----|----------|----|----|----|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | |
| TIME[15:0] | | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 保留 | | | | | | | TGT | 保留 | | | | DLC[3:0] | | | | |
| res | | | | | | | rw | res | | | | rw | rw | rw | rw | |
| 位3:0 | | DLC[15:0]: 发送数据长度 (Data length code) 该域指定了数据报文的数据长度或者远程帧请求的数据长度。1个报文包含0到8个字节数据，而这由DLC决定。 | | | | | | | | | | | | | | |

图 38.1.3.5 寄存器 CAN_TDTxR 各位描述

● CAN 发送邮箱低字节数据寄存器(CAN_TDLxR)

CAN 发送邮箱低字节数据寄存器(CAN_TDLxR)(x=0~2)，该寄存器各位描述如图 38.1.3.6 所示：

| | | | | | | | | | | | | | | | |
|------------|----|---|----|----|----|----|----|------------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| DATA3[7:0] | | | | | | | | DATA2[7:0] | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| DATA1[7:0] | | | | | | | | DATA0[7:0] | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 位31:24 | | DATA3[7:0] : 数据字节3 (Data byte 3) 报文的数据字节3。 | | | | | | | | | | | | | |
| 位23:16 | | DATA2[7:0] : 数据字节2 (Data byte 2) 报文的数据字节2。 | | | | | | | | | | | | | |
| 位15:8 | | DATA1[7:0] : 数据字节1 (Data byte 1) 报文的数据字节1。 | | | | | | | | | | | | | |
| 位7:0 | | DATA0[7:0] : 数据字节0 (Data byte 0) 报文的数据字节0。 报文包含0到8个字节数据，且从字节0开始。 | | | | | | | | | | | | | |

图 38.1.3.6 寄存器 CAN_TDLxR 各位描述

该寄存器用来存储将要发送的数据，这里只能存储低 4 个字节，另外还有一个寄存器 CAN_TDHxR，该寄存器用来存储高 4 个字节，这样总共就可以存储 8 个字节。CAN_TDHxR 的各位描述同 CAN_TDLxR 类似，我们就不单独介绍了。

● CAN 接收 FIFO 邮箱标识符寄存器(CAN_RIxR)

CAN 接收 FIFO 邮箱标识符寄存器(CAN_RIxR)(x=0/1)，该寄存器各位描述同 CAN_TIxR 寄存器几乎一模一样，只是最低位为保留位，该寄存器用于保存接收到的报文标识符等信息，我们可以通过读该寄存器获取相关信息。

同样的，CAN 接收 FIFO 邮箱数据长度和时间戳寄存器(CAN_RDTxR)、CAN 接收 FIFO 邮箱低字节数据寄存器(CAN_RDLxR)和 CAN 接收 FIFO 邮箱高字节数据寄存器(CAN_RDHxR)分别和发送邮箱的：CAN_TDTxR、CAN_TDLxR 以及 CAN_TDHxR 类似，这里我们就不单独一介绍了。详细介绍，请参考《STM32F10xxx 参考手册_V10(中文版).pdf》22.9.3 节(447 页)。

● CAN 过滤器模式寄存器 (CAN_FMR)

CAN 过滤器模式寄存器 (CAN_FMR)，该寄存器各位描述如图 38.1.3.7 所示：

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | FBM27 | FBM26 | FBM25 | FBM24 | FBM23 | FBM22 | FBM21 | FBM20 | FBM19 | FBM18 | FBM17 | FBM16 |
| | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FBM15 | FBM14 | FBM13 | FBM12 | FBM11 | FBM10 | FBM9 | FBM8 | FBM7 | FBM6 | FBM5 | FBM4 | FBM3 | FBM2 | FBM1 | FBM0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

注：请参考图202：过滤器组位宽设置—寄存器组织。

| | |
|-------|--|
| 位13:0 | FBMx ：过滤器模式 (Filter mode) 过滤器组x的工作模式。 0：过滤器组x的2个32位寄存器工作在标识符屏蔽位模式； 1：过滤器组x的2个32位寄存器工作在标识符列表模式。 注：位27:14只出现在互联型产品中，其它产品为保留位。 |
|-------|--|

图 38.1.3.7 寄存器 CAN_FM1R 各位描述

该寄存器用于设置各过滤器组的工作模式，对 28 个过滤器组的工作模式，都可以通过该寄存器设置，不过该寄存器必须在过滤器处于初始化模式下（CAN_FMR 的 FINIT 位=1），才可以进行设置。对 STM32F103ZET6 来说，只有[13:0]这 14 个位有效。

● CAN 过滤器位宽寄存器(CAN_FS1R)

CAN 过滤器位宽寄存器(CAN_FS1R)，该寄存器各位描述如图 38.1.3.8 所示：

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | FSC27 | FSC26 | FSC25 | FSC24 | FSC23 | FSC22 | FSC21 | FSC20 | FSC19 | FSC18 | FSC17 | FSC16 |
| | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FSC15 | FSC14 | FSC13 | FSC12 | FSC11 | FSC10 | FSC9 | FSC8 | FSC7 | FSC6 | FSC5 | FSC4 | FSC3 | FSC2 | FSC1 | FSC0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

注：请参考图202：过滤器组位宽设置—寄存器组织。

| | |
|-------|--|
| 位13:0 | FSCx ：过滤器位宽设置 (Filter scale configuration) 过滤器组x(13~0)的位宽。 0：过滤器位宽为2个16位； 1：过滤器位宽为单个32位。 注：位27:14只出现在互联型产品中，其它产品为保留位。 |
|-------|--|

图 38.1.3.8 寄存器 CAN_FS1R 各位描述

该寄存器用于设置各过滤器组的位宽，对 28 个过滤器组的位宽设置，都可以通过该寄存器实现。该寄存器也只能在过滤器处于初始化模式下进行设置。对 STM32F103ZET6 来说，同样只有[13:0]这 14 个位有效。

● CAN 过滤器 FIFO 关联寄存器 (CAN_FFA1R)

CAN 过滤器 FIFO 关联寄存器 (CAN_FFA1R)，该寄存器各位描述如图 38.1.3.9 所示：

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | FFA27 | FFA26 | FFA25 | FFA24 | FFA23 | FFA22 | FFA21 | FFA20 | FFA19 | FFA18 | FFA17 | FFA16 |
| | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FFA15 | FFA14 | FFA13 | FFA12 | FFA11 | FFA10 | FFA9 | FFA8 | FFA7 | FFA6 | FFA5 | FFA4 | FFA3 | FFA2 | FFA1 | FFA0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

| | |
|-------|---|
| 位13:0 | FFAx ：过滤器位宽设置 (Filter FIFO assignment for filter x) 报文在通过了某过滤器的过滤后，将被存放到其关联的FIFO中。 0：过滤器被关联到FIFO0； |
|-------|---|

图 38.1.3.9 寄存器 CAN_FFA1R 各位描述

该寄存器设置报文通过过滤器组之后，被存入的 FIFO，如果对应位为 0，则存放到 FIFO0；如果为 1，则存放到 FIFO1。该寄存器也只能在过滤器处于初始化模式下配置。

● CAN 过滤器激活寄存器 (CAN_FA1R)

CAN 过滤器激活寄存器 (CAN_FA1R)，该寄存器各位对应过滤器组和前面的几个寄存器

类似，这里就不列出了，把对应位置 1，即开启对应的过滤器组；置 0 则关闭该过滤器组。

● CAN 的过滤器组 i 的寄存器 x (CAN_FiRx)

CAN 的过滤器组 i 的寄存器 x (CAN_FiRx) (互联产品中 i=0~27, 其它产品中 i=0~13; x=1/2)。该寄存器各位描述如图 38.1.3.10 所示:

| | | | | | | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| FB31 | FB30 | FB29 | FB28 | FB27 | FB26 | FB25 | FB24 | FB23 | FB22 | FB21 | FB20 | FB19 | FB18 | FB17 | FB16 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FB15 | FB14 | FB13 | FB12 | FB11 | FB10 | FB9 | FB8 | FB7 | FB6 | FB5 | FB4 | FB3 | FB2 | FB1 | FB0 |
| rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW |

在所有的配置情况下:

| | |
|-------|--|
| 位31:0 | FB[31:0]: 过滤器位 (Filter bits) 标识符模式 寄存器的每位对应于所期望的标识符的相应位的电平。 0: 期望相应位为显性位; 1: 期望相应位为隐性位。 屏蔽位模式 寄存器的每位指示是否对应的标识符寄存器位一定要与期望的标识符的相应位一致。 0: 不关心, 该位不用于比较; 1: 必须匹配, 到来的标识符位必须与滤波器对应的标识符寄存器位相一致。 |
|-------|--|

图 38.1.3.10 寄存器 CAN_FiRx 各位描述

每个过滤器组的 CAN_FiRx 都由 2 个 32 位寄存器构成, 即: CAN_FiR1 和 CAN_FiR2。根据过滤器位宽和模式的不同设置, 这两个寄存器的功能也不尽相同。关于过滤器的映射, 功能描述和屏蔽寄存器的关联, 请参见前面的图 38.1.2.10 (过滤器组位宽模式设置) 的说明。

关于 CAN 的介绍, 就到此结束了。

38.2 硬件设计

1. 例程功能

通过 KEY_UP 按键 (即 WK_UP 按键) 选择 CAN 的工作模式 (正常模式/环回模式), 然后通过 KEY0 控制数据发送, 接着查询是否有数据接收到, 假如接收到数据, 就将接收到的数据显示在 LCD 模块上。如果是环回模式, 我们不需要 2 个开发板。如果是正常模式, 我们就需要 2 个战舰开发板, 并且将他们的 CAN 接口对接起来, 然后一个开发板发送数据, 另外一个开发板将接收到的数据显示在 LCD 模块上。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY_UP - PA0
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) STM32 自带 CAN 控制器
- 5) CAN 收发芯片 TJA1050/SIT1050T

3. 原理图

STM32 有 CAN 的控制器, 但要实现 CAN 通讯的差分电平, 我们还需要借助外围电路来实现, 根据我们需要实现的程序功能, 我们设计电路原理如下:

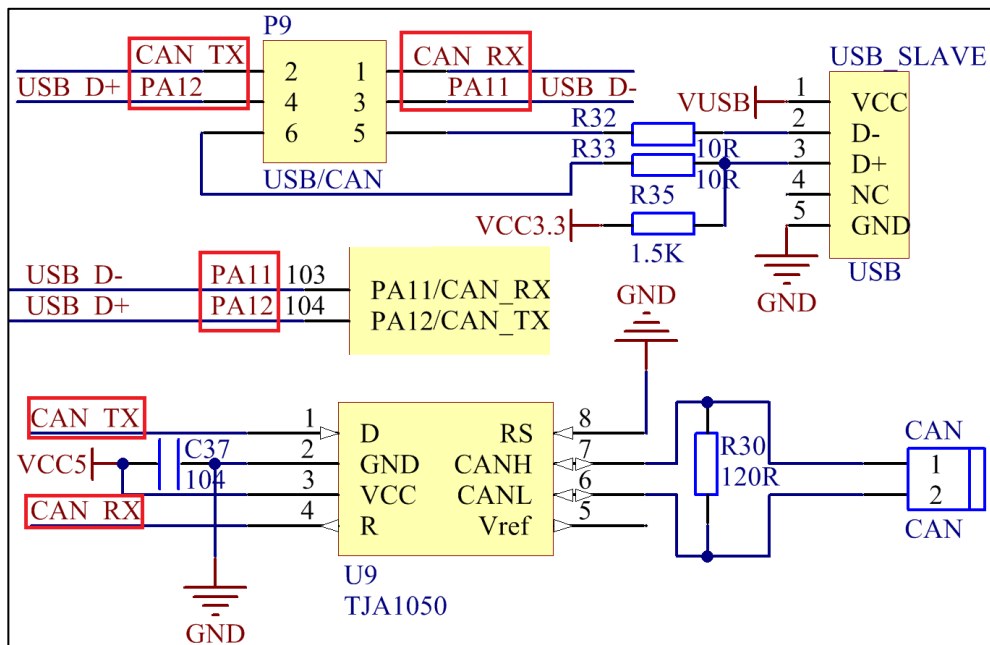


图 38.2.2 CAN 连接原理设计

从上图可以看出:STM32F1 的 CAN 通过 P9 的设置,连接到 TJA1050/SIT1050T 收发芯片,然后通过接线端子 (CAN) 同外部的 CAN 总线连接。图中可以看出,在战舰 STM32F103 开发板上面是带有 120Ω 的终端电阻的,如果我们的开发板不是作为 CAN 的终端的话,需要把这个电阻去掉,以免影响通信。另外,需要注意:CAN 和 USB 共用了 PA11 和 PA12,所以他们不能同时使用。

这里还要注意,我们要设置好开发板上 P9 排针的连接,通过跳线帽将 PA11 和 PA12 分别连接到 CAN_RX 和 CAN_TX 上面,如图 38.2.2 所示。

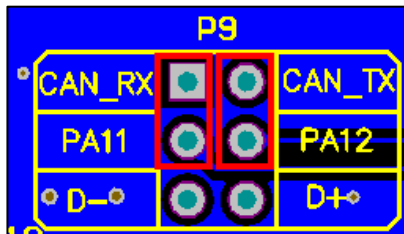


图 38.2.2 CAN 实验需要跳线连接的位置

最后,我们用 2 根导线将两个开发板 CAN 端子的 CAN_L 和 CAN_L, CAN_H 和 CAN_H 连接起来。这里注意不要接反了 (CAN_L 接 CAN_H),接反了会导致通讯异常!!

38.3 程序设计

38.3.1 CAN 的 HAL 库驱动

CAN 在 HAL 库中的驱动代码在 stm32f1xx_hal_can.c 文件 (及其头文件) 中。

1. HAL_CAN_Init 函数

要使用一个外设首先要对它进行初始化,所以先看 CAN 的初始化函数,其声明如下:

```
HAL_StatusTypeDef HAL_CAN_Init(CAN_HandleTypeDef *hcan);
```

● 函数描述:

用于 CAN 控制器的初始化。

● 函数形参:

形参 1 是 CAN 的控制句柄,结构体类型是 CAN_HandleTypeDef,其定义如下:

```
typedef struct __CAN_HandleTypeDef
```

```
{
    CAN_TypeDef                *Instance; /* CAN 控制寄存器基地址 */

```

```

CAN_InitTypeDef      Init;          /* 初始化参数结构体 */
__IO HAL_CAN_StateTypeDef State;    /* CAN 通讯状态 */
__IO uint32_t        ErrorCode;     /* CAN 通讯结果编码 */
} CAN_HandleTypeDef;

```

1) **Instance**: 指向 CAN 寄存器基地址。可以根据 F103 的寄存器的偏移量定义找到各个配置 CAN 的寄存器并对其进行操作。

2) **Init**: can 初始化结构体, 用于配置 CAN 的工作模式等等。它的定义也在 `stm32f1xx_hal_can.h` 中有列出。

```

typedef struct
{
    uint32_t Prescaler;          /* 分频值, 可以配置为 1~1024 间的任意整数 */
    uint32_t Mode;               /* can 操作模式, 有效值参考 CAN_operating_mode 的描述 */
    uint32_t SyncJumpWidth;      /* CAN 硬件的最大超时时间 */
    uint32_t TimeSeg1;           /* CAN_time_quantum_in_bit_segment_1 */
    uint32_t TimeSeg2;           /* CAN_time_quantum_in_bit_segment_2 */
    FunctionalState TimeTriggeredMode; /* 启用或禁用时间触发模式 */
    FunctionalState AutoBusOff;   /* 禁止/使能软件自动断开总线的功能 */
    FunctionalState AutoWakeUp;  /* 禁止/使能 CAN 的自动唤醒功能 */
    FunctionalState AutoRetransmission; /* 禁止/使能 CAN 的自动传输模式 */
    FunctionalState ReceiveFifoLocked; /* 禁止/使能 CAN 的接收 FIFO */
    FunctionalState TransmitFifoPriority; /* 禁止/使能 CAN 的发送 FIFO */
} CAN_InitTypeDef;

```

调用 CAN 的初始化函数时, 我们主要也是对这个结构体赋值, 配置 CAN 的工作模式。

3) **State**: CAN 操作状态, 主要用于 HAL 库中的函数。

4) **ErrorCode**: CAN 错误操作信息。CAN 定义了多个错误返回值, 可以方便查找通讯异常的可能原因, 可以看到 CAN 总线的容错能力要大于串口。

● 函数返回值:

HAL_StatusTypeDef 枚举类型的值, 有 4 个, 分别是 HAL_OK 表示成功, HAL_ERROR 表示错误, HAL_BUSY 表示忙碌, HAL_TIMEOUT 为超时。

调用初始化函数之后, 同样我们需要重定义 HAL_CAN_MspInit 来初始化跟底层硬件相关的配置, 我们后面编写初始化函数时用到。

2. HAL_CAN_ConfigFilter 函数

CAN 的接收过滤器是属于硬件, 可以根据软件的设置, 在接收报文的时候, 可以过滤出符合过滤器配置条件的报文 ID, 大大节省了 CPU 的开销, 过滤器配置函数定义如下:

```

HAL_StatusTypeDef HAL_CAN_ConfigFilter(CAN_HandleTypeDef *hcan,
                                       CAN_FilterTypeDef *sFilterConfig)

```

● 函数描述:

用于配置 CAN 的接收过滤器。

● 函数形参:

形参 1 是 CAN 的控制句柄指针, 初始化函数已经介绍过它的结构了, 这里不重复了。

形参 2 是过滤器的结构体, 这个是根据 STM32 的 CAN 过滤器模式设置的一些配置参数, 它的结构如下:

```

typedef struct
{
    uint32_t FilterIdHigh;        /* 过滤器标识符高位 */
    uint32_t FilterIdLow;        /* 过滤器标识符低位 */
    uint32_t FilterMaskIdHigh;    /* 过滤器掩码号高位 (列表模式下, 也是属于标识符) */
    uint32_t FilterMaskIdLow;    /* 过滤器掩码号低位 (列表模式下, 也是属于标识符) */
    uint32_t FilterFIFOAssignment; /* 与过滤器组管理的 FIFO */
    uint32_t FilterBank;         /* 指定过滤器组, 单 CAN 为 0~13, 双 CAN 可为 0~27 */
    uint32_t FilterMode;         /* 过滤器的模式 标识符屏蔽位模式/标识符列表模式 */
    uint32_t FilterScale;        /* 过滤器的位宽 32 位/16 位 */
    uint32_t FilterActivation;    /* 禁用或者使能过滤器 */
    uint32_t SlaveStartFilterBank; /* 双 CAN 模式下, 规定 CAN 的主从模式的过滤器分配 */
} CAN_FilterTypeDef;

```

我们通过配置过滤器，通过过滤器组的报文，即可从关联的 FIFO 的输出邮箱中获取。

- **函数返回值：**

我们只关注 HAL_OK 的情况。

3. HAL_CAN_Start 函数

使能 CAN 控制器以接入总线进行数据收发处理。

```
HAL_StatusTypeDef HAL_CAN_Start(CAN_HandleTypeDef *hcan)
```

- **函数描述：**

按需要配置完 CAN 总线后，使能 CAN 控制器以接入总线进行数据收发处理。

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

- **函数返回值：**

我们只关注 HAL_OK 的情况。

4. HAL_CAN_ActivateNotification 函数

使能 CAN 的各种中断。

```
HAL_StatusTypeDef HAL_CAN_ActivateNotification(CAN_HandleTypeDef *hcan,
uint32_t ActiveITs)
```

- **函数描述：**

CAN 定义了多种传输中断以满足多种需求，我们只需要在 ActiveITs 中填入相关中断即可，中断源可以在 CAN_IER 寄存器中找到。

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

- **函数返回值：**

我们只关注 HAL_OK 的情况。

5. HAL_CAN_AddTxMessage 函数

发送报文函数。

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan,
CAN_TxHeaderTypeDef *pHeader, uint8_t aData[], uint32_t *pTxMailbox)
```

- **函数描述：**

该函数用于向发送邮箱添加发送报文，并激活发送请求

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

形参 2 是 CAN 发送的结构体，它的结构如下：

```
typedef struct
{
    uint32_t StdId;    /* 标准标识符 11 位 范围:0~0x7FF */
    uint32_t ExtId;    /* 扩展标识符 29 位 范围:0~0x1FFFFFFF */
    uint32_t IDE;      /* 标识符类型 CAN_ID_STD / CAN_ID_EXT */
    uint32_t RTR;      /* 帧类型 CAN_RTR_DATA / CAN_RTR_REMOTE */
    uint32_t DLC;      /* 帧长度 范围:0~8byte */
    FunctionalState TransmitGlobalTime; /* 时间戳是否在开始时捕获 */
} CAN_TxHeaderTypeDef;
```

这里需要注意的是：当标识符选择位 IDE 为 CAN_ID_STD 时，表示本报文是标准帧，使用 StdId 成员存储报文 ID；当它的值为 CAN_ID_EXT 时，表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。其他成员可以对照发送邮箱寄存器相关位进行理解。

形参 3 是报文的内容。

形参 4 是发送邮箱编号，可选三个发送邮箱之一。

6. HAL_CAN_GetRxMessage 函数

接收消息函数。

```
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan, uint32_t
RxFifo, CAN_RxHeaderTypeDef *pHeader, uint8_t aData[])
```

- **函数描述：**

该函数可从接收 FIFO 里面的输出邮箱获取到消息报文。

● 函数形参:

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

形参 2 是接收 FIFO，具体是 FIFO0/1，得看过滤器组关联的 FIFO。

形参 3 是 CAN 接收的结构体，它的结构如下：

```
typedef struct
{
    uint32_t StdId;           /* 标准标识符 11 位 范围:0~0x7FF */
    uint32_t ExtId;          /* 扩展标识符 29 位 范围:0~0x1FFFFFFF */
    uint32_t IDE;             /* 标识符类型 CAN_ID_STD / CAN_ID_EXT */
    uint32_t RTR;             /* 帧类型 CAN_RTR_DATA / CAN_RTR_REMOTE */
    uint32_t DLC;             /* 帧长度 范围:0~8byte */
    uint32_t Timestamp;       /* 在帧接收开始时开始捕获的时间戳 */
    uint32_t FilterMatchIndex; /* 过滤器匹配序号 */
} CAN_RxHeaderTypeDef;
```

在发送结构体中，同样的，也是通过 IDE 位确认该消息报文的标识符类型，该结构体不同于发送结构体还有一个过滤器匹配序号成员，可以查看到此报文是通过哪里过滤器到达接收 FIFO。其他成员可以对照发送邮箱寄存器相关位进行理解。

形参 4 是接收报文的内容。

CAN 的初始化配置步骤

1) CAN 参数初始化（工作模式、波特率等）

HAL 库通过调用 CAN 初始化函数 HAL_CAN_Init 完成对 CAN 参数初始化，详见例程源码。

注意：该函数会调用：HAL_CAN_MspInit 函数来完成对 CAN 底层的初始化，包括：CAN 以及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2) 开启 CAN 和对应管脚时钟，配置 CAN_TX 和 CAN_RX 的复用功能输出

首先开启 CAN 的时钟，然后配置 CAN 相关引脚为复用功能（对应的引脚可查看中文参考手册 P117）。本实验中 CAN_TX 对应的是 PA12，CAN_RX 对应的是 PA11。他们的时钟开启方法如下：

```
HAL_RCC_CAN1_CLK_ENABLE(); /* 使能 CAN1 */
HAL_RCC_GPIOA_CLK_ENABLE(); /* 开启 GPIOA 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

3) 设置过滤器

HAL 库通过调用 HAL_CAN_ConfigFilter 完成 CAN 的过滤器相关参数初始化。

4) CAN 数据接收和发送

通过调用 HAL_CAN_AddTxMessage 函数进行发送消息。

通过调用 HAL_CAN_GetRxMessage 函数进行接收数据。

至此，CAN 就可以开始正常工作了。如果用到中断，就还需要进行中断相关的配置。本实验也提供 CAN 接收中断，详看例程源码，这里就不作介绍了。

38.3.2 程序流程图

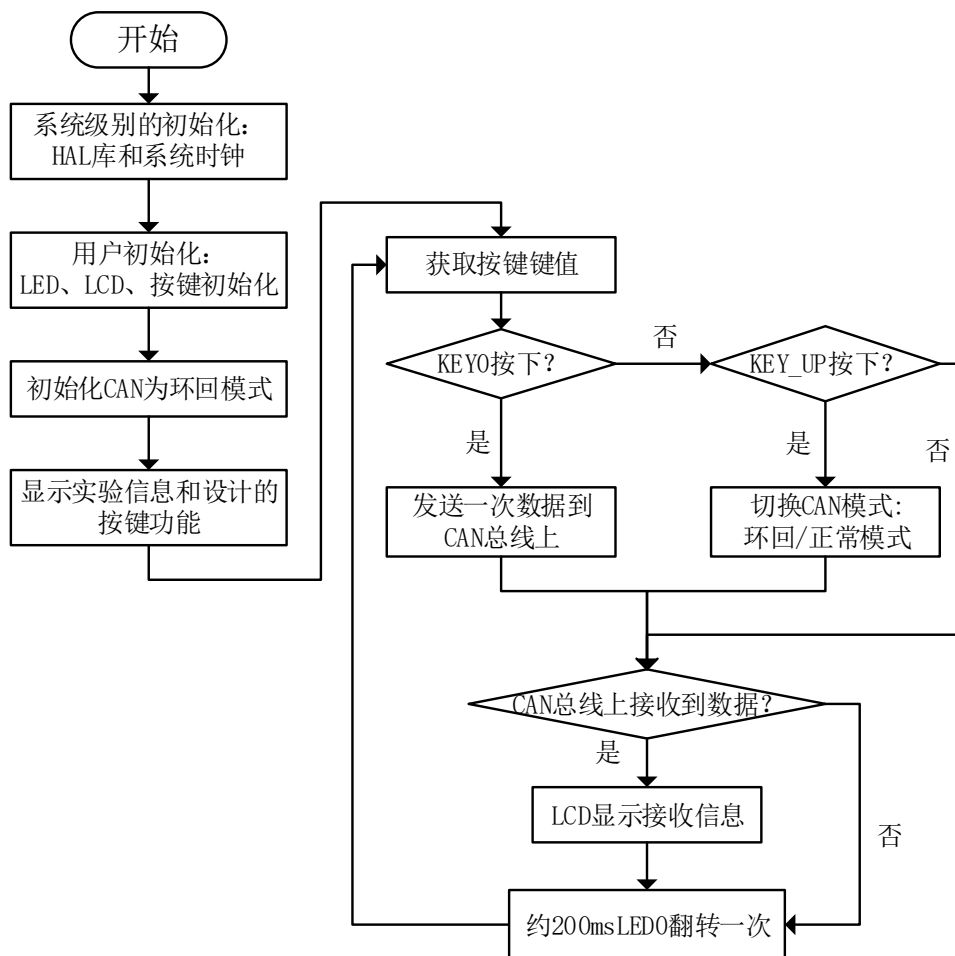


图 38.3.2.1 CAN 通讯实验程序流程图

38.3.3 程序解析

我们要使用 LED、LCD、按键这些功能，直接复制上一个 485 实验的代码，把 rs485 的代码从工程中移除，并在 Drivers/BSP 目录下新建一个《CAN》文件夹，与之前一样，新建 can.c/can.h 文件并把它们加入到工程中。

1. can.c 函数

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。CAN 驱动相关源码包括两个文件：can.c 和 can.h。

我们根据《STM32F10xxx 参考手册_V10（中文版）.pdf》第 22 章的内容，我们利用前面介绍的 HAL 库函数来配置 CAN 的接收时钟及模式等参数，配置过滤器以使能硬件自动过滤功能，最后使能 CAN 以开始 CAN 控制器的工作，编写 CAN 初始化函数。

```

/**
 * @brief      CAN 初始化
 * @param      tsjw      : 重新同步跳跃时间单元.范围: 1~3;
 * @param      tbs2      : 时间段 2 的时间单元.范围: 1~8;
 * @param      tbs1      : 时间段 1 的时间单元.范围: 1~16;
 * @param      brp        : 波特率分频器.范围: 1~1024;
 * @note       以上 4 个参数，在函数内部会减 1，所以，任何一个参数都不能等于 0
 *             CAN 挂在 APB1 上面，其输入时钟频率为 Fpclk1 = PCLK1 = 36Mhz
 *             tq          = brp * tpclk1;

```

```

*          波特率 = Fpclk1 / ((tbs1 + tbs2 + 1) * brp);
*          我们设置 can_init(1, 8, 9, 4, 1), 则 CAN 波特率为:
*          36M / ((8 + 9 + 1) * 4) = 500Kbps
* @param   mode      : CAN_MODE_NORMAL, 正常模式;
*              CAN_MODE_LOOPBACK, 回环模式;
* @retval   0, 初始化成功; 其他, 初始化失败;
*/
uint8_t can_init(uint32_t tsjw, uint32_t tbs2, uint32_t tbs1, uint16_t brp,
                uint32_t mode)
{
    g_canx_handler.Instance = CAN1;
    g_canx_handler.Init.Prescaler = brp; /* 分频系数(Fdiv)为brp+1 */
    g_canx_handler.Init.Mode = mode; /* 模式设置 */
    /* 重新同步跳跃宽度(Tsjw)为tsjw+1个时间单位 CAN_SJW_1TQ~CAN_SJW_4TQ */
    g_canx_handler.Init.SyncJumpWidth = tsjw;
    g_canx_handler.Init.TimeSeg1 = tbs1; /* tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ */
    g_canx_handler.Init.TimeSeg2 = tbs2; /* tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ */
    g_canx_handler.Init.TimeTriggeredMode = DISABLE; /* 非时间触发通信模式 */
    g_canx_handler.Init.AutoBusOff = DISABLE; /* 软件自动离线管理 */
    g_canx_handler.Init.AutoWakeUp = DISABLE; /* 通过软件唤醒睡眠模式 */
    g_canx_handler.Init.AutoRetransmission = ENABLE; /* 禁止报文自动传送 */
    g_canx_handler.Init.ReceiveFifoLocked = DISABLE; /* 报文不锁定,新的覆盖旧的 */
    g_canx_handler.Init.TransmitFifoPriority = DISABLE; /* 优先级由报文标识符决定 */

    if (HAL_CAN_Init(&g_canx_handler) != HAL_OK)
    {
        return 1;
    }

#ifdef CAN_RX0_INT_ENABLE
    /* 使用中断接收, FIFO0 消息挂号中断允许 */
    __HAL_CAN_ENABLE_IT(&g_canx_handler, CAN_IT_RX_FIFO0_MSG_PENDING);
    HAL_NVIC_EnableIRQ(USB_LP_CAN1_RX0_IRQn); /* 使能 CAN 中断 */
    HAL_NVIC_SetPriority(USB_LP_CAN1_RX0_IRQn, 1, 0); /* 抢占优先级 1, 子优先级 0 */
#endif

    CAN_FilterTypeDef sFilterConfig;
    /*配置 CAN 过滤器*/
    sFilterConfig.FilterBank = 0; /* 过滤器 0 */
    sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
    sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
    sFilterConfig.FilterIdHigh = 0x0000; /* 32 位 ID */
    sFilterConfig.FilterIdLow = 0x0000;
    sFilterConfig.FilterMaskIdHigh = 0x0000; /* 32 位 MASK */
    sFilterConfig.FilterMaskIdLow = 0x0000;
    sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0; /* 过滤器 0 关联到 FIFO0 */
    sFilterConfig.FilterActivation = ENABLE; /* 激活滤波器 0 */
    sFilterConfig.SlaveStartFilterBank = 14;

    if (HAL_CAN_ConfigFilter(&g_canx_handler, &sFilterConfig) != HAL_OK)
    {
        /* 过滤器配置 */
        return 2;
    }

    if (HAL_CAN_Start(&g_canx_handler) != HAL_OK)
    {
        /* 启动 CAN 外围设备 */
        return 3;
    }

    return 0;
}

```

调用 HAL_CAN_Init 后会调用 HAL_CAN_MspInit, 我们重定义这个函数, 在函数中初始

化我们用于控制 CAN 的收发引脚:

```
void HAL_CAN_MspInit(CAN_HandleTypeDef *hcan)
{
    if (CAN1 == hcan->Instance)
    {
        CAN_RX_GPIO_CLK_ENABLE(); /* CAN_RX 脚时钟使能 */
        CAN_TX_GPIO_CLK_ENABLE(); /* CAN_TX 脚时钟使能 */
        __HAL_RCC_CAN1_CLK_ENABLE(); /* 使能 CAN1 时钟 */

        GPIO_InitTypeDef gpio_initure;

        gpio_initure.Pin = CAN_TX_GPIO_PIN;
        gpio_initure.Mode = GPIO_MODE_AF_PP;
        gpio_initure.Pull = GPIO_PULLUP;
        gpio_initure.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(CAN_TX_GPIO_PORT, &gpio_initure); /* CAN_TX 设置成复用推挽输出 */

        gpio_initure.Pin = CAN_RX_GPIO_PIN;
        gpio_initure.Mode = GPIO_MODE_AF_INPUT;
        HAL_GPIO_Init(CAN_RX_GPIO_PORT, &gpio_initure); /* CAN_RX 设置成复用输入模式 */
    }
}
```

初始化函数就编写完成了,我们要设置它的工作波特率为 500Kbps,设置工作模式为环回模式,参照各个参数的意义,我们最后用以下的配置来完成初始化设置:

```
can_init(CAN_SJW_1TQ, CAN_BS2_8TQ, CAN_BS1_9TQ, 4, CAN_MODE_LOOPBACK);
```

要与其它的 CAN 节点设备通讯,我们需要编写 CAN 相关的收发函数。首先是发送函数,发送报文是有 ID,所以我们在发送时需要设定 ID,故需要设计一个形参为 ID 号,我们利用 HAL 库的发送函数封装一个更方便我们使用的函数,代码如下:

```
/**
 * @brief      CAN 发送一组数据
 * @note       发送格式固定为: 标准 ID, 数据帧
 * @param      id      : 标准 ID(11 位)
 * @retval     发送状态 0, 成功; 1, 失败;
 */
uint8_t can_send_msg(uint32_t id, uint8_t *msg, uint8_t len)
{
    uint32_t TxMailbox = CAN_TX_MAILBOX0;
    g_canx_txheader.StdId = id; /* 标准标识符 */
    g_canx_txheader.ExtId = id; /* 扩展标识符(29 位) */
    g_canx_txheader.IDE = CAN_ID_STD; /* 使用标准帧 */
    g_canx_txheader.RTR = CAN_RTR_DATA; /* 数据帧 */
    g_canx_txheader.DLC = len;

    if (HAL_CAN_AddTxMessage(&g_canx_handler, &g_canx_txheader,
                             msg, &TxMailbox) != HAL_OK) /* 发送消息 */
    {
        return 1;
    }

    /* 等待发送完成,所有邮箱为空(3 个邮箱) */
    while (HAL_CAN_GetTxMailboxesFreeLevel(&g_canx_handler) != 3);

    return 0;
}
```

在 CAN 初始化时,我们对于过滤器的配置是不过滤任何报文 ID,也就是说可以接收全部报文。但是我们可以编写接收函数时,使用软件的方式过滤报文 ID,通过形参来跟接收到的报文 ID 进行匹配。接收函数代码具体如下:

```
/**
 * @brief      CAN 接收数据查询
 * @note       接收数据格式固定为: 标准 ID, 数据帧
 * @param      id      : 要查询的 标准 ID(11 位)
```

```

* @param      buf      : 数据缓存区
* @retval      接收结果
* @arg        0      , 无数据被接收到;
* @arg        其他, 接收的数据长度
*/
uint8_t can_receive_msg(uint32_t id, uint8_t *buf)
{
    if (HAL_CAN_GetRx FifoFillLevel(&g_canx_handler, CAN_RX_FIFO0) != 1)
    {
        return 0;
    }

    if (HAL_CAN_GetRxMessage(&g_canx_handler, CAN_RX_FIFO0, &g_canx_rxheader,
                             buf) != HAL_OK)
    {
        return 0;
    }

    /* 接收到的 ID 不对 / 不是标准帧 / 不是数据帧 */
    if (g_canx_rxheader.StdId != id || g_canx_rxheader.IDE != CAN_ID_STD ||
        g_canx_rxheader.RTR != CAN_RTR_DATA)
    {
        return 0;
    }
    return g_canx_rxheader.DLC;
}

```

最后，我们可以把 `can_send_msg` 函数加到 USMART 接口中，就可以方便地用串口来调试 CAN 接口了。

2. main.c 代码

在 `main.c` 里面编写如下代码：

```

int main(void)
{
    uint8_t key;
    uint8_t i = 0, t = 0;
    uint8_t cnt = 0;
    uint8_t canbuf[8];
    uint8_t rxlen = 0;
    uint8_t res;
    uint8_t mode = 1; /* CAN 工作模式：0, 正常模式；1, 环回模式 */

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    /* CAN 初始化，环回模式，波特率 500Kbps */
    can_init(CAN_SJW_1TQ, CAN_BS2_8TQ, CAN_BS1_9TQ, 4, CAN_MODE_LOOPBACK);

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "CAN TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "LoopBack Mode", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY0:Send KEK_UP:Mode", RED);
    lcd_show_string(30, 150, 200, 16, 16, "Count:", RED); /* 显示当前计数值 */
    lcd_show_string(30, 170, 200, 16, 16, "Send Data:", RED); /* 提示发送的数据 */
    lcd_show_string(30, 230, 200, 16, 16, "Receive Data:", RED); /* 提示接收的数据 */

    while (1)

```

```

{
    key = key_scan(0);
    if (key == KEY0_PRES) /* KEY0 按下,发送一次数据 */
    {
        for (i = 0; i < 8; i++)
        {
            canbuf[i] = cnt + i; /* 填充发送缓冲区 */
            if (i < 4)
            { /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 190, canbuf[i], 3, 16, 0X80, BLUE);
            }
            else
            { /* 显示数据 */
                lcd_show_xnum(30 + (i-4)*32, 210, canbuf[i], 3, 16, 0X80, BLUE);
            }
        }

        res = can_send_msg(0X12, canbuf, 8); /* ID = 0X12, 发送 8 个字节 */
        if (res)
        { /* 提示发送失败 */
            lcd_show_string(30 + 80, 170, 200, 16, 16, "Failed", BLUE);
        }
        else
        { /* 提示发送成功 */
            lcd_show_string(30 + 80, 170, 200, 16, 16, "OK   ", BLUE);
        }
    }
    else if (key == WKUP_PRES) /* WK_UP 按下, 改变 CAN 的工作模式 */
    {
        mode = !mode;
        if (mode == 0) /* 正常模式, 需要 2 个开发板 */
        { /* CAN 模式初始化, 正常模式, 波特率 500Kbps */
            can_init(CAN_SJW_1TQ, CAN_BS2_8TQ, CAN_BS1_9TQ, 4, CAN_MODE_NORMAL);
            lcd_show_string(30, 110, 200, 16, 16, "Normal Mode ", RED);
        }
        else /* 回环模式, 一个开发板就可以测试了. */
        { /* CAN 模式初始化, 回环模式, 波特率 500Kbps */
            can_init(CAN_SJW_1TQ, CAN_BS2_8TQ, CAN_BS1_9TQ, 4, CAN_MODE_LOOPBACK);
            lcd_show_string(30, 110, 200, 16, 16, "LoopBack Mode", RED);
        }
    }

    rxlen = can_receive_msg(0X12, canbuf); /* CAN ID = 0X12, 接收数据查询 */
    if (rxlen) /* 接收到有数据 */
    {
        lcd_fill(30, 270, 130, 310, WHITE); /* 清除之前的显示 */
        for (i = 0; i < rxlen; i++)
        {
            if (i < 4)
            { /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 250, canbuf[i], 3, 16, 0X80, BLUE);
            }
            else
            { /* 显示数据 */
                lcd_show_xnum(30 + (i-4) * 32, 270, canbuf[i], 3, 16, 0X80, BLUE);
            }
        }
    }

    t++;
    delay_ms(10);
    if (t == 20)
    {
        LED0_TOGGLE(); /* 提示系统正在运行 */
    }
}

```

```

    t = 0;
    cnt++;
    lcd_show_xnum(30 + 48, 150, cnt, 3, 16, 0x80, BLUE); /* 显示数据 */
}
}
}

```

main 函数的执行过程与程序流程图是一致的，这里需要注意：在选择正常模式的情况下，要使两个开发板通信成功，必须保持一致的波特率。其它细节，参考光盘资料中的源码即可。

38.4 下载验证

在代码编译成功之后，我们通过下载代码到开发板上，得到如图 38.4.1 所示：

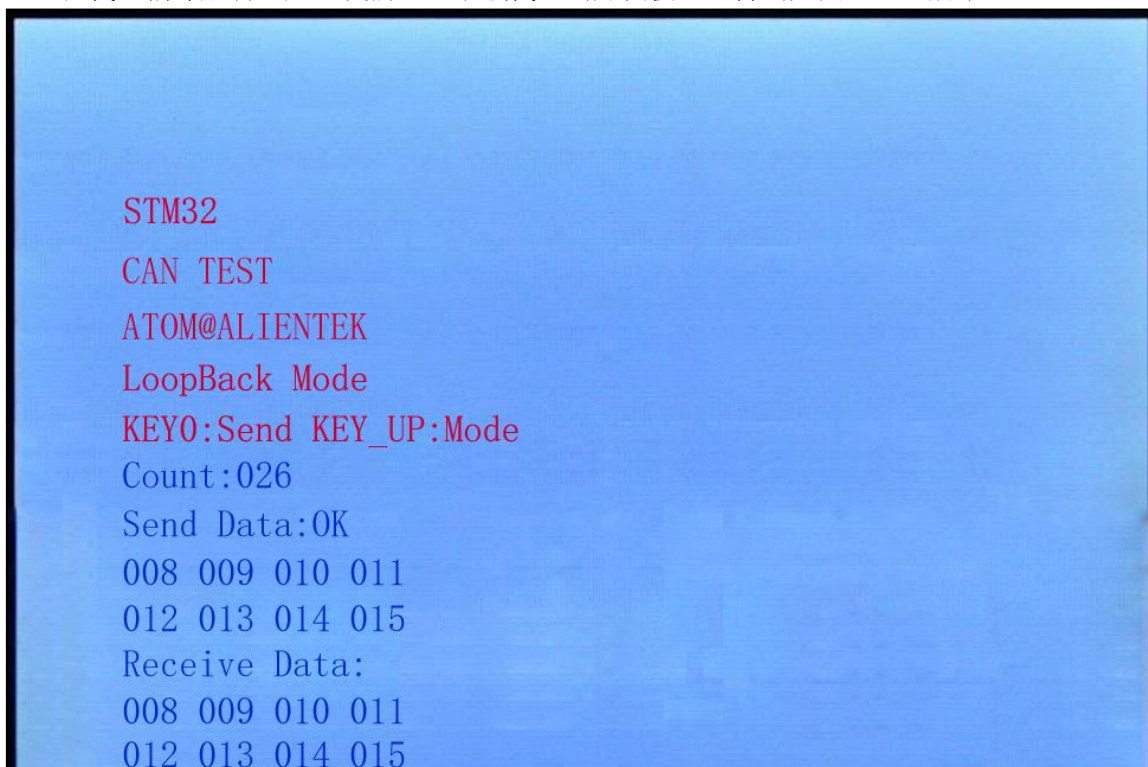


图 38.4.1 程序运行效果图

伴随 LED0 的不停闪烁，提示程序在运行。默认设置的回环模式，按下 KEY0 就可以在 LCD 模块上面看到自发自收的数据（如上图所示），如果我们选择正常模式（KEY_UP 按键切换），就必须连接两个开发板的 CAN 接口，然后就可以互发数据了。如图 38.4.2 和图 38.4.3 所示：


```

STM32
CAN TEST
ATOM@ALIENTEK
Nnormal Mode
KEY0:Send KEY_UP:Mode
Count:208
Send Data:OK
168 169 170 171
172 173 174 175
Receive Data:
    
```

图 38.4.2 CAN 正常模式发送数据

```

STM32
CAN TEST
ATOM@ALIENTEK
Nnormal Mode
KEY0:Send KEY_UP:Mode
Count:110
Send Data:OK

Receive Data:
168 169 170 171
172 173 174 175
    
```

图 38.4.3 CAN 正常模式接收数据

图 38.4.2 来自开发板 A，发送了 8 个数据，图 37.4.3 来自开发板 B，收到了来自开发板 A 的 8 个数据。另外，利用 USMART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十九章 触摸屏实验

本章，我们将介绍如何使用 STM32F1 来驱动触摸屏，正点原子战舰 STM32F103 本身并没有触摸屏控制器，但是它支持触摸屏，可以通过外接带触摸屏的 LCD 模块（比如正点原子 TFTLCD 模块），来实现触摸屏控制。在本章中，我们将向大家介绍 STM32 控制正点原子 TFTLCD 模块（包括电阻触摸与电容触摸），实现触摸屏驱动，最终实现一个手写板的功能。

本章分为如下几个部分：

- 39.1 触摸屏简介
- 39.2 硬件设计
- 39.3 程序设计
- 39.4 下载验证

39.1 触摸屏简介

触摸屏是在显示屏的基础上，在屏幕或屏幕上方分布一层与屏幕大小相近的传感器形成的组合器件。触摸和显示功能由软件控制，可以独立也可以组合实现，用户可以通过侦测传感器的触点再配合相应的软件实现触摸效果。目前最常用的触摸屏有两种：电阻式触摸屏与电容式触摸屏。下面，我们来分别介绍。

39.1.1 电阻式触摸屏

正点原子 2.4/2.8/3.5 寸 TFTLCD 模块自带的触摸屏都属于电阻式触摸屏，下面简单介绍下电阻式触摸屏的原理。

电阻触摸屏的主要部分是一块与显示器表面非常贴合的电阻薄膜屏，这是一种多层的复合薄膜，具体结构如下图 39.1.1.1 所示。

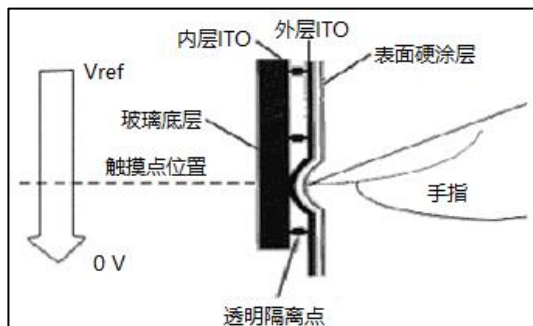


图 39.1.1.1 电阻触摸屏多层结构图

表面硬涂层起保护作用，主要是一层外表面硬化处理、光滑防擦的塑料层。玻璃底层用于支撑上面的结构，主要是玻璃或者塑料平板。透明隔离点用来分离外层 ITO 和内层 ITO。ITO 层是触摸屏关键结构，是涂有铟锡金属氧化物的导电层。还有一个结构上图没有标出，就是 PET 层。PET 层是聚酯薄膜，处于外层 ITO 和表面硬涂层之间，很薄很有弹性，触摸时向下弯曲，使得 ITO 层接触。

当手指触摸屏幕时，两个 ITO 层在触摸点位置就有接触，电阻发生变化，在 X 和 Y 两个方向上产生电信号，然后送到触摸屏控制器，具体情况如下图 39.1.1.2 所示。触摸屏控制器侦测到这一接触并计算出 X 和 Y 方向上的 AD 值，简单来讲，电阻触摸屏将触摸点（X，Y）的物理位置转换为代表 X 坐标和 Y 坐标的电压值。单片机与触摸屏控制器进行通信获取到 AD 值，通过一定比例关系运算，获得 X 和 Y 轴坐标值。

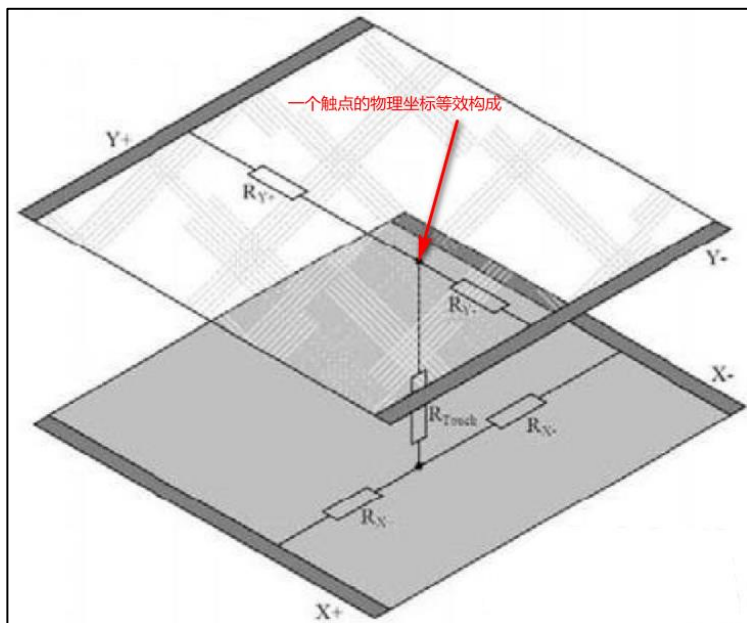


图 39.1.1.2 电阻式触摸屏的触点坐标结构

电阻触摸屏的优点：精度高、价格便宜、抗干扰能力强、稳定性好。

电阻触摸屏的缺点：容易被划伤、透光性不太好、不支持多点触摸。

从以上介绍可知，触摸屏都需要一个 AD 转换器，一般来说是需要一个控制器的。正点原子 TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7543、ADS7846、TSC2046、XPT2046 和 HR2046 等。这几款芯片的驱动基本上是一样的，也就是你只要写出了 XPT2046 的驱动，这个驱动对其他几个芯片也是有效的。而且封装也有一样的，完全 PIN-TO-PIN 兼容。所以在替换起来，很方便。

正点原子 TFTLCD 模块自带的触摸屏控制芯片为 XPT2046 或 HR2046。这里以 XPT2046 作为介绍。XPT2046 是一款 4 导线制触摸屏控制器，使用的是 SPI 通信接口，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换（一次获取 X 位置，一次获取 Y 位置）查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。

XPT2046 的驱动方法也是很简单，主要看懂 XPT2046 通信时序图，如下图 39.1.1.3 所示。

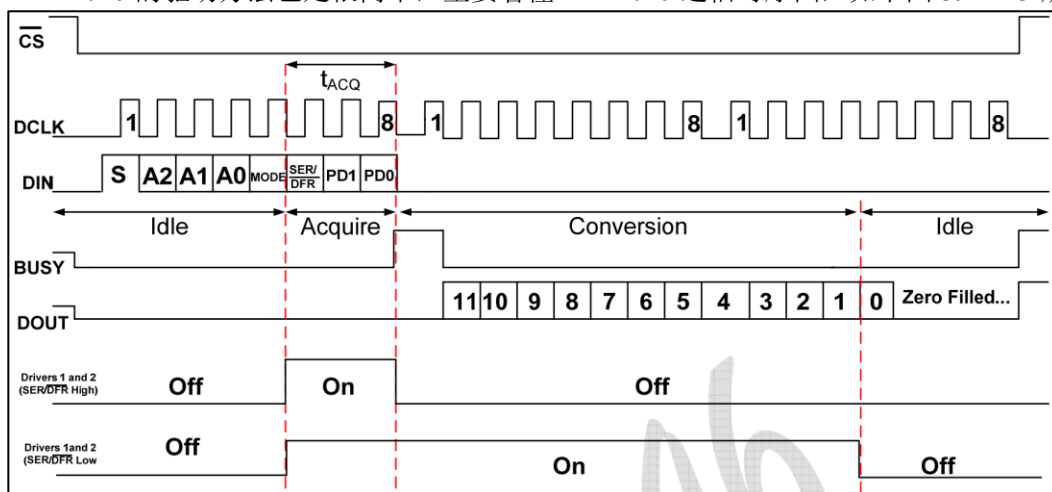


图 39.1.1.3 XPT2046 通信时序图

依照时序图，就可以很好写出这个通信代码，上图具体过程：拉低片选，选中器件→发送命令字→清除 BUSY→读取 16 位数据（高 12 位数据有效即转换的 AD 值）→拉高片选，结束

操作。这里的难点就是需要搞清楚命令字该发送什么？只要搞清楚发送什么数值，就可以获取到 AD 值。命令字的详情如下图 39.1.1.4 所示：

| 位 7 (MSB) | 位 6 | 位 5 | 位 4 | 位 3 | 位 2 | 位 1 | 位 0 (LSB) |
|--------------|-----|-----|-----|------|---------|-----|--------------|
| S | A2 | A1 | A0 | MODE | SER/DFR | PD1 | PD0 |

| 位 | 名称 | 功能描述 |
|-----|---------|--|
| 7 | S | 开始位。为 1 表示一个新的控制字节到来，为 0 则忽略 PIN 引脚上数据 |
| 6-4 | A2-A0 | 通道选择位。参见表 1 和表 2 |
| 3 | MODE | 12 位/8 位转换分辨率选择位。为 1 选择 8 位为转换分辨率，为 0 选择 12 位分辨率 |
| 2 | SER/DFR | 单端输入方式/差分输入方式选择位。为 1 是单端输入方式，为 0 是差分输入方式 |
| 1-0 | PD1-PD0 | 低功耗模式选择位。若为 11，器件总处于供电状态；若为 00，器件在变换之间处于低功耗模式 |

图 39.1.1.4 命令字详情图

位 7，开始位，置 1 即可。位 3，为了提供精度，MODE 位清 0 选择 12 位分辨率。位 2，是进行工作模式选择，为了达到最佳性能，首选差分工作模式即该位清 0 即可。位 1-0 是功耗相关的，直接清 0 即可。而位 6-4 的值要取决于工作模式，在确定了差分功能模式后，通道选择位也确定了，如图 39.1.1.5 所示。

| A2 | A1 | A0 | +REF | -REF | YN | XP | YP | Y-位置 | X-位置 | Z1-位置 | Z2-位置 | 驱动 |
|----|----|----|------|------|-----|-----|-----|------|------|-------|-------|--------|
| 0 | 0 | 1 | YP | YN | | +IN | | 测量 | | | | YP, YN |
| 0 | 1 | 1 | YP | XN | | +IN | | | | 测量 | | YP, XN |
| 1 | 0 | 0 | YP | XN | +IN | | | | | | 测量 | YP, XN |
| 1 | 0 | 1 | XP | XN | | | +IN | | 测量 | | | XP, XN |

图 39.1.1.5 差分模式输入配置图（SER/DFR=0）

从上图，就可以知道：当我们需要检测 Y 轴位置时，A2A1A0 赋值为 001；检测 X 轴位置时，A2A1A0 赋值为 101。结合前面对其他位的赋值，在 X，Y 方向与屏幕相同的情况下，命令字 0xD0 就是读取 X 坐标 AD 值，0x90 就是读取 Y 坐标的 AD 值。假如 X，Y 方向与屏幕相反，0x90 就是读取 X 坐标的 AD 值，而 0xD0 就是读取 Y 坐标的 AD 值。

关于这个芯片其他的功能，也可以参考芯片的 datasheet。

电阻式触摸屏就介绍到这里。

39.1.2 电容式触摸屏

现在几乎所有智能手机，包括平板电脑都是采用电容屏作为触摸屏，电容屏是利用人体感应进行触点检测控制，不需要直接接触或只需要轻微接触，通过检测感应电流来定位触摸坐标。正点原子 4.3/7 寸 TFTLCD 模块自带的触摸屏采用的是电容式触摸屏，下面简单介绍下电容式触摸屏的原理。

电容式触摸屏主要分为两种：

1、表面电容式电容触摸屏。

表面电容式触摸屏技术是利用 ITO(铟锡氧化物，是一种透明的导电材料)导电膜，通过电场感应方式感测屏幕表面的触摸行为进行。但是表面电容式触摸屏有一些局限性，它只能识别一个手指或者一次触摸。

2、投射式电容触摸屏。

投射电容式触摸屏是传感器利用触摸屏电极发射出静电场线。一般用于投射电容传感技术的电容类型有两种：自我电容和交互电容。

自我电容又称绝对电容，是最广为采用的一种方法，自我电容通常是指扫描电极与地构成的电容。在玻璃表面有用 ITO 制成的横向与纵向的扫描电极，这些电极和地之间就构成一个电容的两极。当用手或触摸笔触摸的时候就会并联一个电容到电路中去，从而使在该条扫描线上

的总体的电容量有所改变。在扫描的时候，控制 IC 依次扫描纵向和横向电极，并根据扫描前后的电容变化来确定触摸点坐标位置。笔记本电脑触摸输入板就是采用的这种方式，笔记本电脑的输入板采用 X*Y 的传感电极阵列形成一个传感格子，当手指靠近触摸输入板时，在手指和传感电极之间产生一个小量电荷。采用特定的运算法则处理来自行、列传感器的信号来确定手指的位置。

交互电容又叫做跨越电容，它是在玻璃表面的横向和纵向的 ITO 电极的交叉处形成电容。交互电容的扫描方式就是扫描每个交叉处的电容变化，来判定触摸点的位置。当触摸的时候就会影响到相邻电极的耦合，从而改变交叉处的电容量，交互电容的扫描方法可以侦测到每个交叉点的电容值和触摸后电容变化，因而它需要的扫描时间与自我电容的扫描方式相比要长一些，需要扫描检测 X*Y 根电极。目前智能手机/平板电脑等的触摸屏，都是采用交互电容技术。

正点原子所选择的电容触摸屏，也是采用的是投射式电容屏（交互电容类型），所以后面仅以投射式电容屏作为介绍。

透射式电容触摸屏采用纵横两列电极组成感应矩阵，来感应触摸。以两个交叉的电极矩阵，即：X 轴电极和 Y 轴电极，来检测每一格感应单元的电容变化，如图 39.1.2.1 所示：

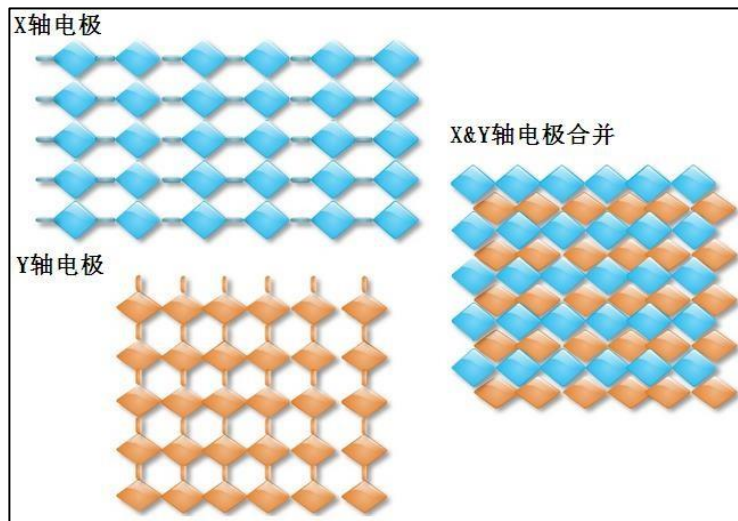


图 39.1.2.1 投射式电容屏电极矩阵示意图

示意图中的电极，实际是透明的，这里是为了方便大家理解。图中，X、Y 轴的透明电极电容屏的精度、分辨率与 X、Y 轴的通道数有关，通道数越多，精度越高。以上就是电容触摸屏的基本原理，接下来看看电容触摸屏的优缺点：

电容触摸屏的优点：手感好、无需校准、支持多点触摸、透光性好。

电容触摸屏的缺点：成本高、精度不高、抗干扰能力差。

这里特别提醒大家电容触摸屏对工作环境的要求是比较高的，在潮湿、多尘、高低温环境下面，都是不适合使用电容屏的。

电容触摸屏一般都需要一个驱动 IC 来检测电容触摸，正点原子的电容触摸屏使用的是 IIC 接口输出触摸数据的触摸芯片。正点原子 7" TFTLCD 模块的电容触摸屏，采用的是 15*10 的驱动结构（10 个感应通道，15 个驱动通道），采用的是 GT911/FT5206 作为驱动 IC。正点原子 4.3" TFTLCD 模块采用的驱动 IC 是：GT9xxx（GT9147/GT917S/GT911/GT1151/GT9271），不同型号感应通道和驱动通道数量都不一样，详看数据手册，但是这些驱动 IC 驱动方式都类似，这里我们以 GT9147 为例给大家做介绍，其他的大家参考着学习即可。

GT9147 与 MCU 通过 4 根线连接：SDA、SCL、RST 和 INT。GT9147 的 IIC 地址，可以是 0X14 或者 0X5D，当复位结束后的 5ms 内，如果 INT 是高电平，则使用 0X14 作为地址，否则使用 0X5D 作为地址，具体的设置过程，请看：GT9147 数据手册.pdf 这个文档。本章我们使用 0X14 作为器件地址（不含最低位，换算成读写命令则是读：0X29，写：0X28），接下来，介绍一下 GT9147 的几个重要的寄存器。

1. 控制命令寄存器（0X8040）

该寄存器可以写入不同值，实现不同的控制，我们一般使用 0 和 2 这两个值，写入 2，即

可软复位 GT9147。在硬复位之后，一般要往该寄存器写 2，实行软复位。然后，写入 0，即可正常读取坐标数据（并且会结束软复位）。

2，配置寄存器组（0X8047~0X8100）

这里共 186 个寄存器，用于配置 GT9147 的各个参数，这些配置一般由厂家提供给我们（一个数组），所以我们只需要将厂家给我们的配置，写入到这些寄存器里面，即可完成 GT9147 的配置。由于 GT9147 可以保存配置信息（可写入内部 FLASH，从而不需要每次上电都更新配置），我们有点注意的地方提醒大家：1，0X8047 寄存器用于指示配置文件版本号，程序写入的版本号，必须大于等于 GT9147 本地保存的版本号，才可以更新配置。2，0X80FF 寄存器用于存储校验和，使得 0X8047~0X80FF 之间所有数据之和为 0。3，0X8100 用于控制是否将配置保存在本地，写 0，则不保存配置，写 1 则保存配置。

3，产品 ID 寄存器（0X8140~0X8143）

这里总共由 4 个寄存器组成，用于保存产品 ID，对于 GT9147，这 4 个寄存器读出来就是：9，1，4，7 四个字符（ASCII 码格式）。因此，我们可以通过这 4 个寄存器的值，来判断驱动 IC 的型号，以便执行不同的初始化。

4，状态寄存器（0X814E）

该寄存器各位描述如表 39.1.2.1 所示：

| 寄存器 | bit7 | bit6 | bit5 | bit4 | bit3 | bit2 | bit1 | bit0 |
|--------|-----------|------|------|------|--------|------|------|------|
| 0X814E | buffer 状态 | 大点 | 接近有效 | 按键 | 有效触点个数 | | | |

表 39.1.2.1 状态寄存器各位描述

这里，我们仅关心最高位和最低 4 位，最高位用于表示 buffer 状态，如果有数据（坐标/按键），buffer 就会是 1，最低 4 位用于表示有效触点的个数，范围是：0~5，0，表示没有触摸，5 表示有 5 点触摸。最后，该寄存器在每次读取后，如果 bit7 有效，则必须写 0，清除这个位，否则不会输出下一次数据！！这个要特别注意！！

5，坐标数据寄存器（共 30 个）

这里共分成 5 组（5 个点），每组 6 个寄存器存储数据，以触点 1 的坐标数据寄存器组为例，如表 39.1.2.2 所示：

| 寄存器 | bit7~0 | 寄存器 | bit7~0 |
|--------|----------------|--------|----------------|
| 0X8150 | 触点 1 x 坐标低 8 位 | 0X8151 | 触点 1 x 坐标高 8 位 |
| 0X8152 | 触点 1 y 坐标低 8 位 | 0X8153 | 触点 1 y 坐标高 8 位 |
| 0X8154 | 触点 1 触摸尺寸低 8 位 | 0X8155 | 触点 1 触摸尺寸高 8 位 |

表 39.1.2.2 触点 1 坐标寄存器组描述

我们一般只用到触点的 x，y 坐标，所以只需要读取 0X8150~0X8153 的数据，组合即可得到触点坐标。其他 4 组分别是：0X8158、0X8160、0X8168 和 0X8170 等开头的 16 个寄存器组成，分别针对触点 2~4 的坐标。同样 GT9147 也支持寄存器地址自增，我们只需要发送寄存器组的首地址，然后连续读取即可，GT9147 会自动地址自增，从而提高读取速度。

GT9147 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：GT9147 编程指南.pdf 这个文档。

GT9147 只需要经过简单的初始化就可以正常使用了，初始化流程：硬复位→延时 10ms→结束硬复位→设置 IIC 地址→延时 100ms→软复位→更新配置（需要时）→结束软复位。此时 GT9147 即可正常使用了。然后，我们不停的查询 0X814E 寄存器，判断是否有有效触点，如果有，则读取坐标数据寄存器，得到触点坐标。特别注意，如果 0X814E 读到的值最高位为 1，就必须对该位写 0，否则无法读到下一次坐标数据。

电容式触摸屏部分，就介绍到这里。

39.1.3 触摸控制原理

前面已经简单地介绍了电阻屏和电容屏的原理，并且知道了不同类型的触摸屏其实是屏幕+触摸传感器组成。那么这里就会有两组相互独立的参数：屏幕坐标和触摸坐标。要实现触摸功能，就是要把触摸点和屏幕坐标对应起来。

我们以 LCD 显示屏为例，我们知道屏幕的扫描方向是可以编程设定的，而触摸点，在触摸传感器安装好后，AD 值的变化方向则是固定的，我们以最常见的屏幕坐标方向：先从左到右，再从上到下扫描为例，此时，屏幕坐标和触点 AD 的坐标有类似的规律：从坐标原点出发，水平方向屏幕坐标增加时，AD 值的 X 方向也增加；屏幕坐标的 Y 方向坐标增加，AD 值的 Y 方向也增加；坐标减少时对应的关系也类似，可以用图 39.1.3.1 的示意图来表示这种关系：

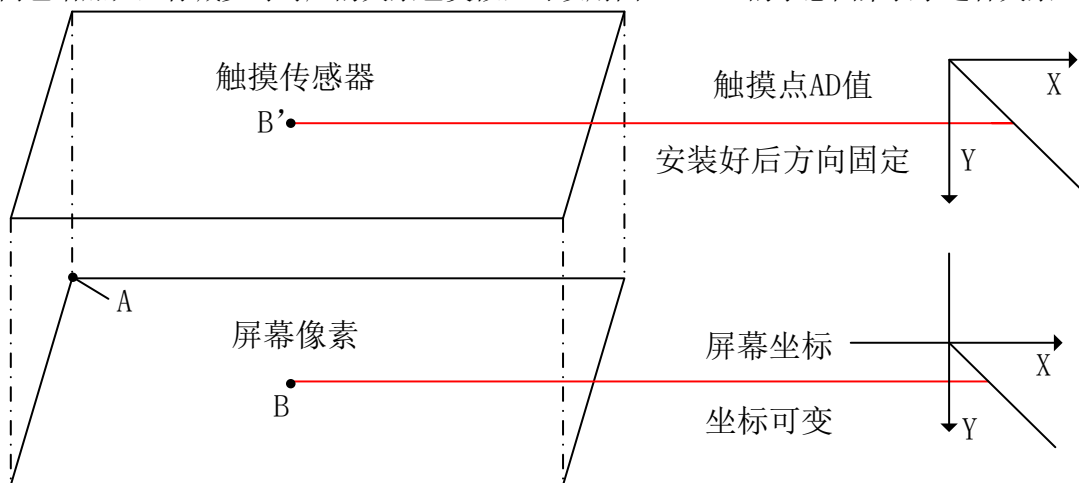


图 39.1.3.1 屏幕坐标和触摸坐标的一种对应关系

这里再来引入两个概念，物理坐标和逻辑坐标。物理坐标指触摸屏上点的实际位置，通常以液晶上点的个数来度量。逻辑坐标指这点被触摸时 A/D 转换后的坐标值。仍以图 39.1.3.1 为例，我们假定液晶最左上角为坐标轴原点 A，在液晶上任取一点 B（实际人手比像素点大得多，一次按下会有多个触点，此处取十字线交叉中心），B 在 X 方向与 A 相距 100 个点，在 Y 方向与 A 距离 200 个点，则这点的物理坐标 B 为（100，200）。如果我们触摸这一点时得到的 X 向 A/D 转换值为 200，Y 向 A/D 转换值为 400，则这点的逻辑坐标 B' 为（200，400）。

需要特别说明的是，正点原子的电容屏的参数已经在出厂时由厂家调好，所以无需进行校准，而且可以直接读到转换后的触点坐标；对于电阻屏，请大家理解并熟记物理坐标和逻辑坐标逻辑上的对应关系，我们后面编程需要用到。

39.2 硬件设计

1. 例程功能

正点原子的触摸屏种类很多，并且设计了规格相对统一的接口。根据屏幕的种类不同，设置了相应的硬件 ID（正点原子自编 ID），可以通过软件判断触摸屏的种类。

本章实验功能简介：开机的时候先初始化 LCD，读取 LCD ID，随后，根据 LCD ID 判断是电阻触摸屏还是电容触摸屏，如果是电阻触摸屏，则先读取 24C02 的数据判断触摸屏是否已经校准过，如果没有校准，则执行校准程序，校准过后再进入电阻触摸屏测试程序，如果已经校准了，就直接进入电阻触摸屏测试程序。

如果是 4.3 寸电容触摸屏，则执行 GT9xxx 的初始化代码；如果是 7 寸电容触摸屏（仅支持新款 7 寸屏，使用 SSD1963+FT5206 方案），则执行 FT5206 的初始化代码，在初始化电容触摸屏完成后，进入电容触摸屏测试程序（电容触摸屏无需校准!!）。

电阻触摸屏测试程序和电容触摸屏测试程序基本一样，只是电容触摸屏支持最多 5 点同时触摸，电阻触摸屏只支持一点触摸，其他一模一样。测试界面的右上角会有一个清空的操作区域（RST），点击这个地方就会将输入全部清除，恢复白板状态。使用电阻触摸屏的时候，可以通过按 KEY0 来实现强制触摸屏校准，只要按下 KEY0 就会进入强制校准程序。

2. 硬件资源

1) LED 灯

LED0 - PB5

- 2) 独立按键
KEY0 - PE4
- 3) EEPROM AT24C02
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 5) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3. 原理图

所有这些资源与 STM32F1 的连接图, 在前面都已经介绍了, 这里我们只针对 TFTLCD 模块与 STM32F1 的连接端口再说明一下, TFTLCD 模块的触摸屏(电阻触摸屏) 总共有 5 根线与 STM32F1 连接, 连接电路图如图 39.2.1 所示:

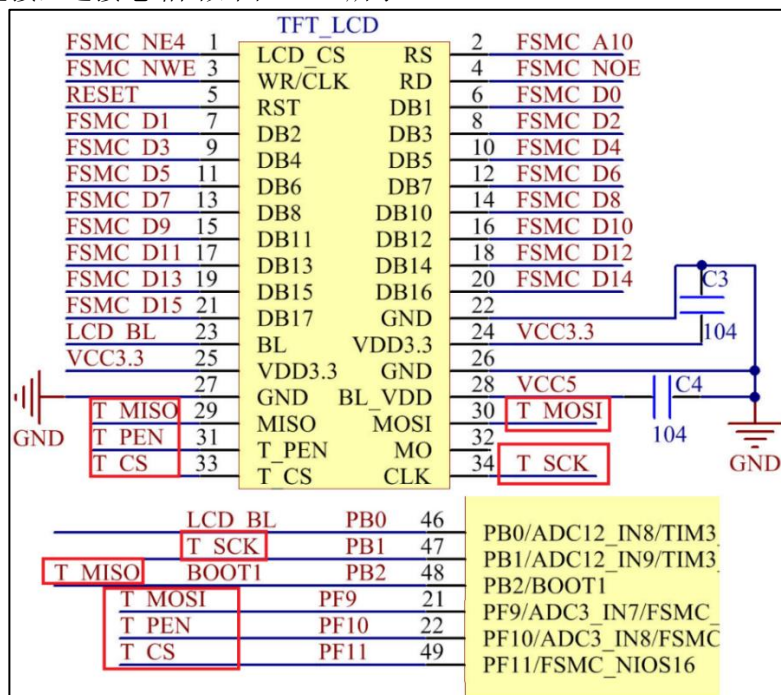


图 39.2.1 触摸屏与 STM32F1 的连接图

从图中可以看出: T_SCK、T_MISO、T_MOSI、T_PEN 和 T_CS 分别连接在 STM32F1 的 PB1、PB2、PF9、PF10 和 PF11 上。

如果是电容式触摸屏, 我们的接口和电阻式触摸屏一样(上图右侧接口), 只是没有用到五根线了, 而是四根线, 分别是: T_PEN(CT_INT)、T_CS(CT_RST)、T_CLK(CT_SCL)和 T_MOSI(CT_SDA)。其中: CT_INT、CT_RST、CT_SCL 和 CT_SDA 分别是 GT9147/FT5206 的: 中断输出信号、复位信号, IIC 的 SCL 和 SDA 信号。我们用查询的方式读取 GT9147/FT5206 的数据, 对于 FT5206 没有用到中断信号(CT_INT), 所以同 STM32F1 的连接, 最少只需要 3 根线即可, 不过 GT9147 等 IC 还需要用到 CT_INT 做 IIC 地址设定, 所以需要 4 根线连接。

39.3 程序设计

39.3.1 HAL 库驱动

触摸芯片我们使用到的是 IIC 和 SPI 的驱动, 这部分的时序分析可以参考之前 IIC/SPI 的章节, 我们直接使用的是软件模拟的方式, 所以只需要使用 HAL 库的驱动的 GPIO 操作部分。

触摸 IC 驱动步骤

1) 初始化通信接口与其 IO (使能时钟、配置 GPIO 工作模式)

触摸 IC 用到的 GPIO 口, 主要是 PB1、PB2、PF9、PF10 和 PF11, 因为都是用软件模拟的方式, 因此在这里我们只需使能 GPIOB 和 GPIOF 时钟即可。参考代码如下:

```
HAL_RCC_GPIOB_CLK_ENABLE(); /* 使能 GPIOB 时钟 */
HAL_RCC_GPIOF_CLK_ENABLE(); /* 使能 GPIOF 时钟 */
```

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

2) 编写通信协议基础读写函数

通过参考时序图，在 IIC 驱动或 SPI 驱动基础上，编写基础读写函数。读写函数均以一字节数据进行操作。

3) 参考触摸 IC 时序图，编写触摸 IC 读写驱动函数

根据触摸 IC 的读写时序进行编写触摸 IC 的读写函数，详见本例程源码。

4) 编写坐标获取函数（电阻触摸屏和电容触摸屏）

查阅数据手册获得命令词(电阻触摸屏)/寄存器(电容触摸屏)，通过读写函数获取坐标数据，详见本例程源码。

39.3.2 程序流程图

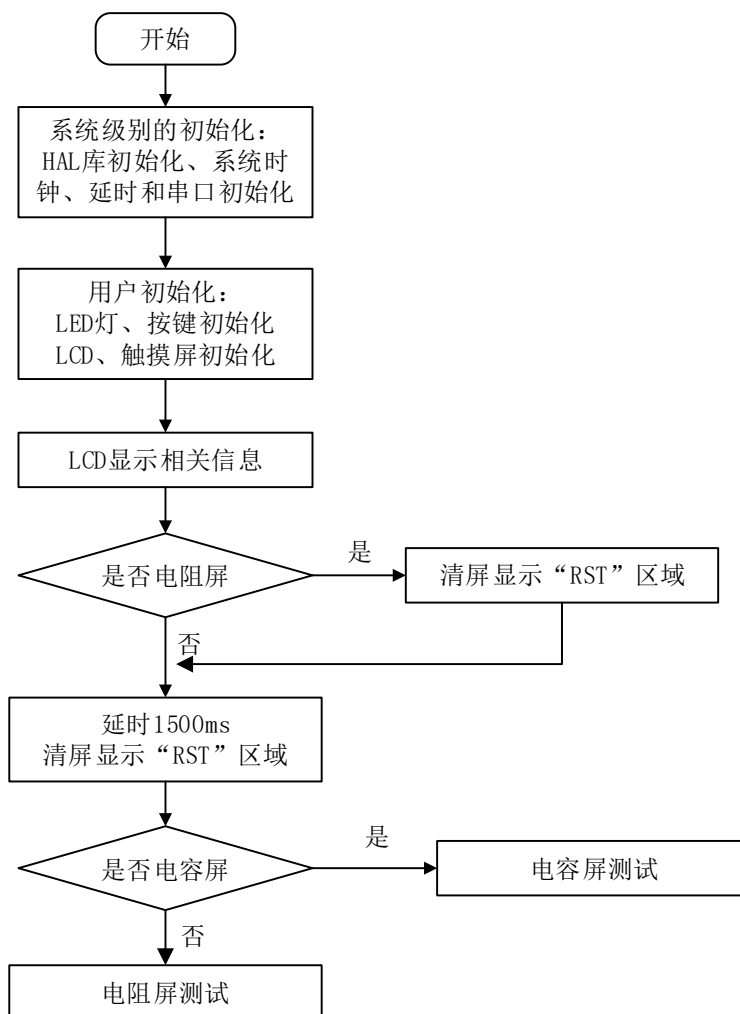


图 39.3.2.1 触摸屏实验流程图

39.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。TOUCH 驱动源码包括如下文件：ctiic.c、ctiic.h、ft5206.c、ft5206.h、gt9xxx.c、gt9xxx.h、touch.c 和 touch.h。

由于正点原子的 TFTLCD 的型号很多，触摸控制这部分驱动代码根据不同屏幕搭载的触摸芯片驱动而有不同，在我们的屏幕上使用的是 LCD ID 来帮助软件上区分。为了解决多种驱动芯片的问题，我们设计了 touch.c/touch.h 这两个文件统一管理各类型的驱动。不同的驱动芯片

类型可以在 touch.c 中集中添加, 并通过 touch.c 中的接口统一调用, 不同的触摸芯片各自编写独立的.c/h 文件, 需要时被 touch.c 调用。电阻触摸屏相关代码也在 touch.c 中实现。

1. 触摸管理驱动代码

因为需要支持的触摸驱动比较多, 为了方便管理和添加新的驱动, 我们用 touch.c 文件来统一管理这些触摸驱动, 然后针对各类触摸芯片编写独立的驱动。为了方便管理触摸, 我们在 touch.h 中定义一个用于管理触摸信息的结构体类型, 具体代码如下:

```
/* 触摸屏控制器 */
typedef struct
{
    uint8_t (*init)(void);          /* 初始化触摸屏控制器 */
    uint8_t (*scan)(uint8_t);       /* 扫描触摸屏.0, 屏幕扫描;1, 物理坐标; */
    void (*adjust)(void);           /* 触摸屏校准 */
    uint16_t x[CT_MAX_TOUCH];       /* 当前坐标 */
    uint16_t y[CT_MAX_TOUCH];       /* 电容屏有最多 10 组坐标, 电阻屏则用 x[0], y[0] 代表: 此次扫描时, 触屏的坐标, 用 x[9], y[9] 存储第一次按下时的坐标 */

    uint16_t sta;                   /* 笔的状态
                                     * b15: 按下 1/ 松开 0;
                                     * b14: 0, 没有按键按下; 1, 有按键按下.
                                     * b13~b10: 保留
                                     * b9~b0: 电容触摸屏按下的点数 (0, 表示未按下, 1 表示按下)
                                     */

    /* 5 点校准触摸屏校准参数 (电容屏不需要校准) */
    float xfac;                     /* 5 点校准法 x 方向比例因子 */
    float yfac;                     /* 5 点校准法 y 方向比例因子 */
    short xc;                       /* 中心 X 坐标物理值 (AD 值) */
    short yc;                       /* 中心 Y 坐标物理值 (AD 值) */

    /* 新增的参数, 当触摸屏的左右上下完全颠倒时需要用到.
     * b0: 0, 竖屏 (适合左右为 X 坐标, 上下为 Y 坐标的 TP)
     *      1, 横屏 (适合左右为 Y 坐标, 上下为 X 坐标的 TP)
     * b1~6: 保留.
     * b7: 0, 电阻屏
     *      1, 电容屏
     */
    uint8_t touchtype;
} _m_tp_dev;

extern _m_tp_dev tp_dev;          /* 触屏控制器在 touch.c 里面定义 */
```

这里我们定义了函数指针, 只要把相对应的触摸芯片的函数指针赋值给它, 就可以通过这个通用接口很方便调用不同芯片的函数接口。正点原子不同的触摸屏区别如下:

1、在使用 4.3 寸屏、10.1 寸屏电容屏时, 使用的是汇顶科技的 GT9xxx 系列触摸屏驱动 IC, 这是一个 IIC 接口的驱动芯片, 我们要编写 gt9xxx 系列芯片的初始化程序, 并编写一个坐标扫描程序, 这里我们先预留这两个接口分别为 gt9xxx_init() 和 gt9xxx_scan(), 在 gt9xxx.c 文件中再专门实现这两个驱动, 标记使用的为电容屏;

2、类似地, 在使用 SSD1963 7 寸屏、7 寸 800*480/1024*600 RGB 屏时, 我们的屏幕搭载的触摸驱动芯片是 ft5206/GT911, FT5206 触摸 IC 预留这两个接口分别为 ft5206_init() 和 ft5206_scan(), 在 ft5206.c 文件中再专门实现这两个驱动, 标记使用的为电容屏; GT911 也是调用 gt9xxx_init() 和 gt9xxx_scan() 接口。

3、当为其它 ID 时, 默认为电阻屏, 而电阻屏默认使用的是 SPI 接口的 XPT2046 芯片。由于电阻屏存在线性误差, 所以在使用前需要进行校准, 这也是为什么在前面的结构体类型中存在关于校准参数的成员。为了避免每次都要进行校准的麻烦, 所以会使用 AT24C02 来存储校准成功后的数据。如何进行校准也会在后面进行讲解。作为电阻屏, 它也有一个扫描坐标函数即

tp_scan()。

(*init)(void)这个结构体函数指针，默认指向 tp_init 的，而在 tp_init 里对触摸屏进行初始化并对(*scan)(uint8_t)函数指针根据触摸芯片类型重新做了指向。在这里简单看一下 touch.c 的触摸屏初始化函数 tp_init，其代码如下：

```
/**
 * @brief      触摸屏初始化
 * @param      无
 * @retval     0, 没有进行校准
 *            1, 进行过校准
 */
uint8_t tp_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    tp_dev.touchtype = 0; /* 默认设置(电阻屏 & 竖屏) */
    tp_dev.touchtype |= lcddev.dir & 0X01; /* 根据 LCD 判定是横屏还是竖屏 */

    if (lcddev.id == 0X5510 || lcddev.id == 0X4342 || lcddev.id == 0X4384 ||
        lcddev.id == 0X1018)
    {
        /* 电容触摸屏, 4.3 寸/10.1 寸屏 */
        gt9xxx_init();
        tp_dev.scan = gt9xxx_scan; /* 扫描函数指向 GT9147 触摸屏扫描 */
        tp_dev.touchtype |= 0X80; /* 电容屏 */
        return 0;
    }
    else if (lcddev.id == 0X1963 || lcddev.id == 0X7084 || lcddev.id == 0X7016)
    {
        /* SSD1963 7 寸屏或者 7 寸 800*480/1024*600 RGB 屏 */
        if (!ft5206_init())
        {
            tp_dev.scan = ft5206_scan; /* 扫描函数指向 FT5206 触摸屏扫描 */
        }
        else
        {
            gt9xxx_init();
            tp_dev.scan = gt9xxx_scan; /* 扫描函数指向 GT9147 触摸屏扫描 */
        }
        tp_dev.touchtype |= 0X80; /* 电容屏 */
        return 0;
    }
    else
    {
        T_PEN_GPIO_CLK_ENABLE(); /* T_PEN 脚时钟使能 */
        T_CS_GPIO_CLK_ENABLE(); /* T_CS 脚时钟使能 */
        T_MISO_GPIO_CLK_ENABLE(); /* T_MISO 脚时钟使能 */
        T_MOSI_GPIO_CLK_ENABLE(); /* T_MOSI 脚时钟使能 */
        T_CLK_GPIO_CLK_ENABLE(); /* T_CLK 脚时钟使能 */

        gpio_init_struct.Pin = T_PEN_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_INPUT; /* 输入 */
        gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
        HAL_GPIO_Init(T_PEN_GPIO_PORT, &gpio_init_struct); /* 初始化 T_PEN 引脚 */

        gpio_init_struct.Pin = T_MISO_GPIO_PIN;
        HAL_GPIO_Init(T_MISO_GPIO_PORT, &gpio_init_struct); /* 初始化 T_MISO 引脚 */

        gpio_init_struct.Pin = T_MOSI_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
        HAL_GPIO_Init(T_MOSI_GPIO_PORT, &gpio_init_struct); /* 初始化 T_MOSI 引脚 */
    }
}
```



```

gpio_init_struct.Pin = T_CLK_GPIO_PIN;
HAL_GPIO_Init(T_CLK_GPIO_PORT, &gpio_init_struct); /* 初始化 T_CLK 引脚 */

gpio_init_struct.Pin = T_CS_GPIO_PIN;
HAL_GPIO_Init(T_CS_GPIO_PORT, &gpio_init_struct); /* 初始化 T_CS 引脚 */

tp_read_xy(&tp_dev.x[0], &tp_dev.y[0]); /* 第一次读取初始化 */
at24cxx_init(); /* 初始化 24CXX */

if (tp_get_adjust_data())
{
    return 0; /* 已经校准 */
}
else /* 未校准? */
{
    lcd_clear(WHITE); /* 清屏 */
    tp_adjust(); /* 屏幕校准 */
    tp_save_adjust_data();
}

tp_get_adjust_data();
}

return 1;
}

```

正点原子的电容屏在出厂时已经由厂家较对好参数了，而电阻屏由于工艺和每个屏的线性有所差异，我们需要先对其进行“校准”，我们在下一点补充说明它的实现。

通过上面的触摸初始化后，我们就可以读取相关的触点信息用于显示编程了，注意到上面还有很多个函数还没实现，比如读取坐标和校准，我们在接下来的代码中将它补充完整。

2. 电阻屏触摸函数

前面我们介绍过了电阻式触摸屏的原理，由于电阻屏的驱动代码都比较类似，我们决定把电阻屏的驱动函数直接添加在 touch.c/touch.h 中实现。

在 touch.c 的初始化函数 tp_init 中，对使用到的 SPI 接口 IO 进行了初始化。接下来介绍一下获取触摸点在屏幕上坐标的算法：先获取逻辑坐标（AD 值），再转换成屏幕坐标。

如何获取逻辑坐标（AD 值），在前面已经分析过了，所以这里我们看一下 tp_read_ad() 函数接口：

```

/**
 * @brief      SPI 读数据
 * @note      从触摸屏 IC 读取 adc 值
 * @param     cmd: 指令
 * @retval    读取到的数据, ADC 值 (12bit)
 */
static uint16_t tp_read_ad(uint8_t cmd)
{
    uint8_t count = 0;
    uint16_t num = 0;
    T_CLK(0); /* 先拉低时钟 */
    T_MOSI(0); /* 拉低数据线 */
    T_CS(0); /* 选中触摸屏 IC */
    tp_write_byte(cmd); /* 发送命令字 */
    delay_us(6); /* ADS7846 的转换时间最长为 6us */
    T_CLK(0);
    delay_us(1);
    T_CLK(1); /* 给 1 个时钟，清除 BUSY */
    delay_us(1);
    T_CLK(0);
}

```



```

for (count = 0; count < 16; count++)    /* 读出 16 位数据,只有高 12 位有效 */
{
    num <<= 1;
    T_CLK(0);        /* 下降沿有效 */
    delay_us(1);
    T_CLK(1);
    if (T_MISO) num++;
}

num >>= 4;            /* 只有高 12 位有效. */
T_CS(1);              /* 释放片选 */
return num;
}

```

这里我们使用的是软件模拟 SPI, 遵照时序编写 SPI 读函数接口。而发送命令字是通过写函数 `tp_write_byte` 来实现, 详看源码。

一次读取的误差会很大, 我们采用平均值滤波的方法, 多次读取数据并丢弃波动最大的最大和最小值, 取余下的平均值。具体可以查看 `tp_read_xoy` 函数内部实现。

```

/* 电阻触摸驱动芯片 数据采集 滤波用参数 */
#define TP_READ_TIMES    5        /* 读取次数 */
#define TP_LOST_VAL      1        /* 丢弃值 */
/**
 * @brief      读取一个坐标值(x 或者 y)
 * @note      连续读取 TP_READ_TIMES 次数据,对这些数据升序排列,
 *            然后去掉最低和最高 TP_LOST_VAL 个数, 取平均值
 *            设置时需满足: TP_READ_TIMES > 2*TP_LOST_VAL 的条件
 * @param      cmd : 指令
 * @arg        0XD0: 读取 X 轴坐标 (@竖屏状态,横屏状态和 Y 对调.)
 * @arg        0X90: 读取 Y 轴坐标 (@竖屏状态,横屏状态和 X 对调.)
 *
 * @retval     读取到的数据 (滤波后的), ADC 值(12bit)
 */
static uint16_t tp_read_xoy(uint8_t cmd)
{
    uint16_t i, j;
    uint16_t buf[TP_READ_TIMES];
    uint16_t sum = 0;
    uint16_t temp;

    for (i = 0; i < TP_READ_TIMES; i++)    /* 先读取 TP_READ_TIMES 次数据 */
    {
        buf[i] = tp_read_ad(cmd);
    }

    for (i = 0; i < TP_READ_TIMES - 1; i++)    /* 对数据进行排序 */
    {
        for (j = i + 1; j < TP_READ_TIMES; j++)
        {
            if (buf[i] > buf[j])                /* 升序排列 */
            {
                temp = buf[i];
                buf[i] = buf[j];
                buf[j] = temp;
            }
        }
    }

    sum = 0;
    for (i = TP_LOST_VAL; i < TP_READ_TIMES - TP_LOST_VAL; i++)
    {
        /* 去掉两端的丢弃值 */
        sum += buf[i]; /* 累加去掉丢弃值以后的数据. */
    }
}

```

```
temp = sum / (TP_READ_TIMES - 2 * TP_LOST_VAL); /* 取平均值 */
return temp;
}
```

有了前述代码，我们就可以通过 `tp_read_xoy(uint8_t cmd)` 接口调取需要的 x 或者 y 坐标的 AD 值了。这里我们加上横屏或者竖屏的处理代码，编写一个可以通过指针一次得到 x 和 y 的两个 AD 值的接口，代码如下：

```
/**
 * @brief      读取 x, y 坐标
 * @param      x,y: 读取到的坐标值
 * @retval     无
 */
static void tp_read_xy(uint16_t *x, uint16_t *y)
{
    uint16_t xval, yval;

    if (tp_dev.touchtype & 0X01) /* X,Y 方向与屏幕相反 */
    {
        xval = tp_read_xoy(0X90); /* 读取 X 轴坐标 AD 值，并进行方向变换 */
        yval = tp_read_xoy(0XD0); /* 读取 Y 轴坐标 AD 值 */
    }
    else /* X,Y 方向与屏幕相同 */
    {
        xval = tp_read_xoy(0XD0); /* 读取 X 轴坐标 AD 值 */
        yval = tp_read_xoy(0X90); /* 读取 Y 轴坐标 AD 值 */
    }

    *x = xval;
    *y = yval;
}
```

为了进一步保证参数的精度，我们连续读两次触摸数据并取平均值作为最后的触摸参数，并对这两次滤波值平均后再传给目标存储区，由于 AD 的精度为 12 位，故该函数读取坐标的值 0~4095，`tp_read_xy2` 的代码如下：

```
/* 连续两次读取 x,y 坐标的数据误差最大允许值 */
#define TP_ERR_RANGE 50 /* 误差范围 */
/**
 * @brief      连续读取 2 次触摸 IC 数据，并滤波
 * @note       连续 2 次读取触摸屏 IC，且这两次的偏差不能超过 ERR_RANGE，满足
 *             条件，则认为读数正确，否则读数错误。该函数能大大提高准确度。
 * @param      x,y: 读取到的坐标值
 * @retval     0，失败；1，成功；
 */
static uint8_t tp_read_xy2(uint16_t *x, uint16_t *y)
{
    uint16_t x1, y1;
    uint16_t x2, y2;

    tp_read_xy(&x1, &y1); /* 读取第一次数据 */
    tp_read_xy(&x2, &y2); /* 读取第二次数据 */

    /* 前后两次采样在+-TP_ERR_RANGE 内 */
    if (((x2 <= x1 && x1 < x2 + TP_ERR_RANGE) || (x1 <= x2 && x2 < x1 + TP_ERR_RANGE)) &&
        ((y2 <= y1 && y1 < y2 + TP_ERR_RANGE) || (y1 <= y2 && y2 < y1 + TP_ERR_RANGE)))
    {
        *x = (x1 + x2) / 2;
        *y = (y1 + y2) / 2;
        return 1;
    }

    return 0;
}
```

根据以上的流程，可以得到电阻屏触摸点的比较精确的 AD 信息。每次触摸屏幕时会对应一组 X、Y 的 AD 值，由于坐标的 AD 值是在 X、Y 方向都是线性的，很容易想到要把触摸信息的 AD 值和屏幕坐标联系起来，这里需要编写一个坐标转换函数，前面在编写初始化接口时讲到的校准函数这时候就派上用场了。

从前面的知识我们就知道触摸屏的 AD 的 X_{AD} 、 Y_{AD} 可以构成一个逻辑平面，LCD 屏的屏幕坐标 X、Y 也是一个逻辑平面，由于存在误差，这两个平面并不重合，校准的作用就是要将逻辑平面映射到物理平面上，即得到触点在液晶屏上的位置坐标。校准算法的中心思想也就是要建立这样一个映射函数现有的校准算法大多是基于线性校准，即首先假定物理平面和逻辑平面之间的误差是线性误差，由旋转和偏移形成。

常用的电阻式触摸屏矫正方法有两点校准法和三点校准法。本文这里介绍的是结合了不同的电阻式触摸屏矫正法的优化算法：五点校正法。其中主要的原理是使用 4 点矫正法的比例运算以及三点矫正法的基准点运算。五点校正法优势在于可以更加精确的计算出 X 和 Y 方向的比例缩放系数，同时提供了中心基准点，对于一些线性电阻系数比较差的电阻式触摸屏有很好的校正功能。校正相关的变量主要有：

- $x[5]$, $y[5]$ 五点定位的物理坐标（LCD 坐标）
- $x1[5]$, $y1[5]$ 五点定位的逻辑坐标（触摸 AD 值）
- KX , KY 横纵方向伸缩系数
- XLC , YLC 中心基点逻辑坐标
- XC , YC 中心基点物理坐标（数值采用 LCD 显示屏的物理长宽分辨率的一半）

$x[5]$, $y[5]$ 五点定位的物理坐标是已知的，其中 4 点分别设置在 LCD 的角落，一点设置在 LCD 正中心，作为基准矫正点，校正关键点和距离布局如图 39.3.3.1 所示。

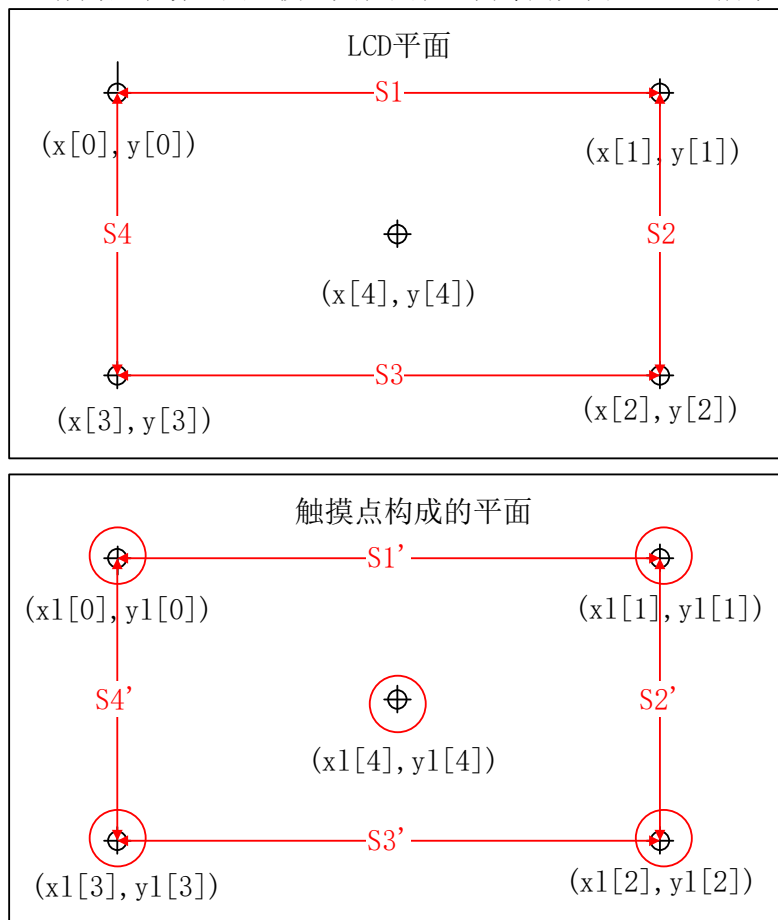


图 39.3.3.1 电阻屏五点校准法的参考点设定


校正步骤如下：

1. 通过先后点击 LCD 的 4 个角落的矫正点，获取 4 个角落的逻辑坐标值。

2. 计算屏幕坐标和四点间距:

$$\begin{aligned} S1 &= x[1] - x[0] \\ S3 &= x[2] - x[3] \\ S2 &= y[2] - y[1] \\ S4 &= y[3] - y[0] \end{aligned}$$

一般取点可以人为的设定 $S1=S3$ 和 $S2=S4$ ，以方便运算。

计算逻辑坐标的四点“间距”，由于实际触点肯定会存在误差，所以触摸点会落在实际设定点的更大范围内，在图 39.3.1 中，设定点为五个点 ，但实际采样时触点有时会落在稍大的外圈范围，图中用红色的圆圈标注了，所以有必要设定一个误差范围：

$$\begin{aligned} S1' &= x1[1] - x1[0] \\ S3' &= x1[2] - x1[3] \\ S2' &= y1[2] - y1[1] \\ S4' &= y1[3] - y1[0] \end{aligned}$$

由于触点的误差，对于逻辑点 $S1'$ 和 $S3'$ 则大概率不会相等，同样的， $S2'$ 和 $S4'$ 也很难取到相等的点，那么为了简化计算，我们强制以 $(S1'+S3')/2$ 的线长作一个矩形一边，以 $(S2'+S4')/2$ 为矩形另一边，这样构建的矩形在误差范围是可以接受的，也方便计算，于是得到 X 和 Y 方向的近似缩放系数：

$$\begin{aligned} KX &= (S1' + S3') / 2 / S1 \\ KY &= (S2' + S4') / 2 / S2 \end{aligned}$$

3. 点击 LCD 正中心，获取中心点的逻辑坐标，作为矫正的基准点。这里也同样的需要限制误差，之后可以得到一个中心点的 AD 值坐标($x1[4], y1[4]$)，这个点的 AD 值我们就作为我们对比的基准点，即 $x1[4]=XLC$ ， $y1[4]=YLC$ ；

4. 完成以上步骤则校正完成。下次点击触摸屏的时候获取的逻辑值 XL 和 YL，便可以按下以公式转换为物理坐标：

$$\begin{aligned} X &= (XL - XLC) / KX + XC \\ Y &= (YL - YLC) / KY + YC \end{aligned}$$

最后一步的转换公式可能不好理解，大家换个角度，如果我们求到的缩放比例是正确的，在取新的触摸的时候，这个触摸点的逻辑坐标和物理坐标的转换，必然与中心点在两方向上的缩放比例相等，用中学数学直线斜率相等的情况，变换便可得到上述公式。

通过上述得到校准参数后，在以后的使用中，我们把所有得到的物理坐标都按照这个关系式来计算，得到的就是触摸点的屏幕坐标。为了省去每次都需要校准的麻烦，我们保存这些参数到 AT24Cxx 的指定扇区地址，这样只要校准一次就可以重复使用这些参数了。

根据上面的原理，我们设计的校准函数 `tp_adjust` 如下：

```
/**
 * @brief      触摸屏校准代码
 * @note      使用五点校准法(具体原理请百度)
 *            本函数得到 x 轴/y 轴比例因子 xfac/yfac 及物理中心坐标值(xc,yc)等 4 个参数
 *            我们规定：物理坐标即 AD 采集到的坐标值，范围是 0~4095.
 *            逻辑坐标即 LCD 屏幕的坐标，范围为 LCD 屏幕的分辨率.
 *
 * @param      无
 * @retval     无
 */
void tp_adjust(void)
{
    uint16_t pxy[5][2];    /* 物理坐标缓存值 */
    uint8_t cnt = 0;
    short s1, s2, s3, s4;  /* 4 个点的坐标差值 */
    double px, py;         /* X,Y 轴物理坐标比例,用于判定是否校准成功 */
    uint16_t outtime = 0;
    cnt = 0;

    lcd_clear(WHITE);      /* 清屏 */
    lcd_show_string(40, 40, 160, 100, 16, TP_REMIND_MSG_TBL, RED); /*显示提示信息*/
    tp_draw_touch_point(20, 20, RED); /* 画点 1 */
}
```

```

tp_dev.sta = 0; /* 消除触发信号 */

while (1) /* 如果连续 10 秒钟没有按下,则自动退出 */
{
    tp_dev.scan(1); /* 扫描物理坐标 */

    if ((tp_dev.sta & 0xc000) == TP_CATH_PRES)
    { /* 按键按下了一次(此时按键松开了) */
        outtime = 0;
        tp_dev.sta &= ~TP_CATH_PRES; /* 标记按键已经被处理过了. */
        pxy[cnt][0] = tp_dev.x[0]; /* 保存 x 物理坐标 */
        pxy[cnt][1] = tp_dev.y[0]; /* 保存 y 物理坐标 */
        cnt++;

        switch (cnt)
        {
            case 1:
                tp_draw_touch_point(20, 20, WHITE); /* 清点 1 */
                tp_draw_touch_point(lcddev.width - 20, 20, RED); /* 画点 2 */
                break;

            case 2:
                tp_draw_touch_point(lcddev.width-20, 20, WHITE); /* 清点 2 */
                tp_draw_touch_point(20, lcddev.height - 20, RED); /* 画点 3 */
                break;

            case 3:
                tp_draw_touch_point(20, lcddev.height-20, WHITE); /* 清点 3 */
                /* 画点 4 */
                tp_draw_touch_point(lcddev.width- 20, lcddev.height - 20, RED);
                break;

            case 4:
                lcd_clear(WHITE); /* 画第五个点了, 直接清屏 */
                /* 画点 5 */
                tp_draw_touch_point(lcddev.width / 2, lcddev.height / 2, RED);
                break;

            case 5: /* 全部 5 个点已经得到 */
                s1=pxy[1][0]-pxy[0][0]; /* 第 2 个点和第 1 个点的 x 轴物理坐标差值(AD 值) */
                s3=pxy[3][0]-pxy[2][0]; /* 第 4 个点和第 3 个点的 x 轴物理坐标差值(AD 值) */
                s2=pxy[3][1]-pxy[1][1]; /* 第 4 个点和第 2 个点的 y 轴物理坐标差值(AD 值) */
                s4=pxy[2][1]-pxy[0][1]; /* 第 3 个点和第 1 个点的 y 轴物理坐标差值(AD 值) */

                px = (double)s1 / s3; /* x 轴比例因子 */
                py = (double)s2 / s4; /* y 轴比例因子 */

                if (px < 0)px = -px; /* 负数改正数 */
                if (py < 0)py = -py; /* 负数改正数 */

                if (px < 0.95 || px > 1.05 || py < 0.95 || py > 1.05 ||
                    abs(s1)>4095||abs(s2)>4095||abs(s3)>4095||abs(s4)>4095||
                    abs(s1)==0 ||abs(s2)==0||abs(s3)==0||abs(s4)==0)
                { /* 比例不合格, 差值大于坐标范围或等于 0, 重绘校准图形 */
                    cnt = 0;
                    /* 清除点 5 */
                    tp_draw_touch_point(lcddev.width/2, lcddev.height/2, WHITE);
                    tp_draw_touch_point(20, 20, RED); /* 重新画点 1 */
                    tp_adjust_info_show(pxy, px, py); /* 显示当前信息,方便找问题 */
                    continue;
                }
            }
        }
    }
}

```

```

        tp_dev.xfac = (float)(s1 + s3) / (2 * (lcddev.width - 40));
        tp_dev.yfac = (float)(s2 + s4) / (2 * (lcddev.height - 40));
        tp_dev.xc = pxy[4][0]; /* X轴,物理中心坐标 */
        tp_dev.yc = pxy[4][1]; /* Y轴,物理中心坐标 */
        lcd_clear(WHITE); /* 清屏 */
        lcd_show_string(35, 110, lcddev.width, lcddev.height, 16,
            "Touch Screen Adjust OK!", BLUE); /* 校准完成 */
        delay_ms(1000);
        tp_save_adjust_data();
        lcd_clear(WHITE); /* 清屏 */
        return; /* 校正完成 */
    }

    }

    delay_ms(10);
    outtime++;
    if (outtime > 1000)
    {
        tp_get_adjust_data();
        break;
    }
}
}

```

注意该函数里面多次使用了 `lcddev.width` 和 `lcddev.height`，用于坐标设置，故在程序调用前需要预先初始化 LCD 得到 LCD 的一些屏幕信息，主要是为了兼容不同尺寸的 LCD（比如 320*240、480*320 和 800*480 的屏都可以兼容）。

有了校准参数后，由于我们需要频繁地进行屏幕坐标和物理坐标的转换，我们为电阻屏增加一个 `tp_scan(uint8_t mode)` 用于转换，为了实际使用上更灵活，我们使这个参数支持物理坐标和屏幕坐标，设计的函数如下：

```

/**
 * @brief      触摸按键扫描
 * @param      mode: 坐标模式
 * @arg        0, 屏幕坐标;
 * @arg        1, 物理坐标 (校准等特殊场合用)
 *
 * @retval      0, 触屏无触摸; 1, 触屏有触摸;
 */
uint8_t tp_scan(uint8_t mode)
{
    if (T_PEN == 0) /* 有按键按下 */
    {
        if (mode) /* 读取物理坐标, 无需转换 */
        {
            tp_read_xy2(&tp_dev.x[0], &tp_dev.y[0]);
        }
        else if (tp_read_xy2(&tp_dev.x[0], &tp_dev.y[0])) /* 读取屏幕坐标, 需要转换 */
        {
            /* 将 X 轴 物理坐标转换成逻辑坐标 (即对应 LCD 屏幕上面的 X 坐标值) */
            tp_dev.x[0] = (signed short)(tp_dev.x[0] - tp_dev.xc)
                / tp_dev.xfac + lcddev.width / 2;
            /* 将 Y 轴 物理坐标转换成逻辑坐标 (即对应 LCD 屏幕上面的 Y 坐标值) */
            tp_dev.y[0] = (signed short)(tp_dev.y[0] - tp_dev.yc)
                / tp_dev.yfac + lcddev.height / 2;
        }

        if ((tp_dev.sta & TP_PRES_DOWN) == 0) /* 之前没有被按下 */
        {
            tp_dev.sta = TP_PRES_DOWN | TP_CATH_PRES; /* 按键按下 */
            tp_dev.x[CT_MAX_TOUCH - 1] = tp_dev.x[0]; /* 记录第一次按下时的坐标 */
            tp_dev.y[CT_MAX_TOUCH - 1] = tp_dev.y[0];
        }
    }
    else

```



```

{
    if (tp_dev.sta & TP_PRES_DOWN)    /* 之前是被按下的 */
    {
        tp_dev.sta &= ~TP_PRES_DOWN; /* 标记按键松开 */
    }
    else    /* 之前就没有被按下 */
    {
        tp_dev.x[CT_MAX_TOUCH - 1] = 0;
        tp_dev.y[CT_MAX_TOUCH - 1] = 0;
        tp_dev.x[0] = 0xffff;
        tp_dev.y[0] = 0xffff;
    }
}

return tp_dev.sta & TP_PRES_DOWN; /* 返回当前的触屏状态 */
}

```

要进行电阻触摸屏的触摸扫描，只要调取 `tp_scan()` 函数，就能灵活地得到触摸坐标。电阻屏的触摸就讲到这里。

3. 电容屏触摸驱动代码

电容触摸芯片使用的是 IIC 接口。IIC 接口部分代码，我们可以参考 `myiic.c` 和 `myiic.h` 的代码，为了使代码独立，我们在“TOUCH”文件夹下也是采用软件模拟 IIC 的方式实现 `ctiic.c` 和 `ctiic.h`，这样 IO 的使用更灵活，这里部分参考 IIC 章节的知识就可以了，这里不重复介绍了。

电容触摸芯片除了 IIC 接口相关引脚 `CT_SCL` 和 `CT_SDA`，还有 `CT_INT` 和 `CT_RST`，接口图如图 39.3.3.2 所示。

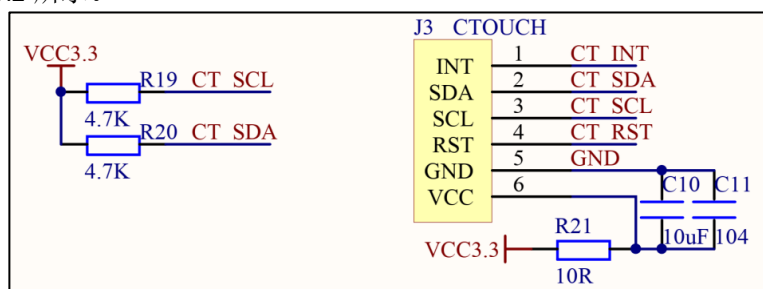


图 39.3.3.2 电容触摸芯片接口图

`gt9xxx_init` 的实现也比较简单，实现 `CT_INT` 和 `CT_RST` 引脚初始化和调用 `ct_iic_init` 函数实现对 `CT_SDA` 和 `CT_SCL` 初始化。由于电容触摸屏在设计时是根据屏幕进行参数设计的，参数已经保存在芯片内部。所以在初始化后，就可以参考手册推荐的 IIC 时序从相对应的坐标数据寄存器中把对应的 XY 坐标数据读出来，再通过数据整理转成 LCD 坐标。

与电阻屏不同的是，我们是通过 IIC 读取状态寄存器的值并非引脚电平。而 `gt9xxx` 系列是支持中断或轮询方式得到触摸状态，本实验使用的是轮询方式：

- 1、按照读时序，先读取寄存器 `0x814E`，若当前 `buffer`（`buffer status` 为 1）数据准备好，则依据有效触点个数到相对应的坐标数据地址处进行坐标数据读取。
- 2、若在 1 中发现 `buffer` 数据（`buffer status` 为 0）未准备好，则等待 1ms 再进行读取。

这样，`gt9xxx_scan()` 函数的实现如下：

```

/* GT9XXX 10 个触摸点(最多) 对应的寄存器表 */
const uint16_t GT9XXX_TPX_TBL[10] =
{
    GT9XXX_TP1_REG, GT9XXX_TP2_REG, GT9XXX_TP3_REG, GT9XXX_TP4_REG, GT9XXX_TP5_REG,
    GT9XXX_TP6_REG, GT9XXX_TP7_REG, GT9XXX_TP8_REG, GT9XXX_TP9_REG, GT9XXX_TP10_REG,
};

/**
 * @brief      扫描触摸屏(采用查询方式)
 * @param      mode : 电容屏未用到次参数，为了兼容电阻屏
 * @retval     当前触屏状态

```

```

* @arg      0, 触屏无触摸;
* @arg      1, 触屏有触摸;
*/
uint8_t gt9xxx_scan(uint8_t mode)
{
    uint8_t buf[4];
    uint8_t i = 0;
    uint8_t res = 0;
    uint16_t temp;
    uint16_t tempsta;
    static uint8_t t = 0; /* 控制查询间隔,从而降低 CPU 占用率 */
    t++;

    if ((t % 10) == 0 || t < 10)
    {
        /* 空闲时,每进入 10 次 CTP_Scan 函数才检测 1 次,从而节省 CPU 使用率 */
        gt9xxx_rd_reg(GT9XXX_GSTID_REG, &mode, 1); /* 读取触摸点的状态 */

        if ((mode & 0X80) && ((mode & 0XF) <= g_gt_tnum))
        {
            i = 0;
            gt9xxx_wr_reg(GT9XXX_GSTID_REG, &i, 1); /* 清标志 */
        }

        if ((mode & 0XF) && ((mode & 0XF) <= g_gt_tnum))
        {
            /* 将点的个数转换为 1 的位数,匹配 tp_dev.sta 定义 */
            temp = 0XFFFF << (mode & 0XF);
            tempsta = tp_dev.sta; /* 保存当前的 tp_dev.sta 值 */
            tp_dev.sta = (~temp) | TP_PRES_DOWN | TP_CATH_PRES;
            tp_dev.x[g_gt_tnum - 1] = tp_dev.x[0]; /* 保存触点 0 的数据 */
            tp_dev.y[g_gt_tnum - 1] = tp_dev.y[0];

            for (i = 0; i < g_gt_tnum; i++)
            {
                if (tp_dev.sta & (1 << i)) /* 触摸有效? */
                {
                    gt9xxx_rd_reg(GT9XXX_TPX_TBL[i], buf, 4); /* 读取 XY 坐标值 */

                    if (lcddev.id == 0X5510) /* 4.3 寸 800*480 MCU 屏 */
                    {
                        if (tp_dev.touchtype & 0X01) /* 横屏 */
                        {
                            tp_dev.y[i] = ((uint16_t)buf[1] << 8) + buf[0];
                            tp_dev.x[i] = 800 - (((uint16_t)buf[3] << 8) + buf[2]);
                        }
                        else
                        {
                            tp_dev.x[i] = ((uint16_t)buf[1] << 8) + buf[0];
                            tp_dev.y[i] = ((uint16_t)buf[3] << 8) + buf[2];
                        }
                    }
                    else /* 其他型号 */
                    {
                        if (tp_dev.touchtype & 0X01) /* 横屏 */
                        {
                            tp_dev.x[i] = (((uint16_t)buf[1] << 8) + buf[0]);
                            tp_dev.y[i] = (((uint16_t)buf[3] << 8) + buf[2]);
                        }
                        else
                        {
                            tp_dev.x[i] = lcddev.width - (((uint16_t)buf[3] << 8) + buf[2]);
                            tp_dev.y[i] = ((uint16_t)buf[1] << 8) + buf[0];
                        }
                    }
                }
            }
        }
    }
}

```

```

        //printf("x[%d]:%d,y[%d]:%d\r\n",i,tp_dev.x[i],i,tp_dev.y[i]);
    }
}

res = 1;

if (tp_dev.x[0] > lcddev.width || tp_dev.y[0] > lcddev.height)
{
    /* 非法数据(坐标超出了) */
    if ((mode & 0XF) > 1) /* 有其他点有数据,则复制第二个触点的数据到第一个触点 */
    {
        tp_dev.x[0] = tp_dev.x[1];
        tp_dev.y[0] = tp_dev.y[1];
        t = 0; /* 触发一次,则会最少连续监测 10 次,从而提高命中率 */
    }
    else /* 非法数据,则忽略此次数据(还原原来的) */
    {
        tp_dev.x[0] = tp_dev.x[g_gt_tnum - 1];
        tp_dev.y[0] = tp_dev.y[g_gt_tnum - 1];
        mode = 0X80;
        tp_dev.sta = tempsta; /* 恢复 tp_dev.sta */
    }
}
else
{
    t = 0; /* 触发一次,则会最少连续监测 10 次,从而提高命中率 */
}
}

if ((mode & 0X8F) == 0X80) /* 无触摸点按下 */
{
    if (tp_dev.sta & TP_PRES_DOWN) /* 之前是被按下的 */
    {
        tp_dev.sta &= ~TP_PRES_DOWN; /* 标记按键松开 */
    }
    else /* 之前就没有被按下 */
    {
        tp_dev.x[0] = 0xffff;
        tp_dev.y[0] = 0xffff;
        tp_dev.sta &= 0XE000; /* 清除点有效标记 */
    }
}

if (t > 240) t = 10; /* 重新从 10 开始计数 */

return res;
}

```

大家可以打开 gt9xxx 芯片对应的编程手册,对照时序,即可理解上述的实现过程,只是程序中为了匹配多种屏幕和横屏显示,添加了一些代码。

电容屏驱动 ft5206.c/ft5206.h 的驱动实现与 gt9xxx 的实现类似,大家参考本例程源码即可。电容屏的触摸实验代码讲解到这里。

4. main 函数和测试代码

在 main.c 里面编程如下代码:

```

void rtp_test(void)
{
    uint8_t key;
    uint8_t i = 0;

    while (1)
    {
        key = key_scan(0);
    }
}

```

```

tp_dev.scan(0);

if (tp_dev.sta & TP_PRES_DOWN) /* 触摸屏被按下 */
{
    if (tp_dev.x[0] < lcddev.width && tp_dev.y[0] < lcddev.height)
    {
        if (tp_dev.x[0] > (lcddev.width - 24) && tp_dev.y[0] < 16)
        {
            load_draw_dialog(); /* 清除 */
        }
        else
        {
            tp_draw_big_point(tp_dev.x[0], tp_dev.y[0], RED); /* 画点 */
        }
    }
}
else
{
    delay_ms(10); /* 没有按键按下时候 */
}

if (key == KEY0_PRES) /* KEY0 按下, 则执行校准程序 */
{
    lcd_clear(WHITE); /* 清屏 */
    tp_adjust(); /* 屏幕校准 */
    tp_save_adjust_data();
    load_draw_dialog();
}

i++;
if (i % 20 == 0) LED0_TOGGLE();
}

/* 10 个触控点的颜色 (电容触摸屏用) */
const uint16_t POINT_COLOR_TBL[10] = {RED, GREEN, BLUE, BROWN, YELLOW, MAGENTA,
CYAN, LIGHTBLUE, BRRED, GRAY};

void ctp_test(void)
{
    uint8_t t = 0;
    uint8_t i = 0;
    uint16_t lastpos[10][2]; /* 最后一次的数据 */
    uint8_t maxp = 5;

    if (lcddev.id == 0X1018) maxp = 10;

    while (1)
    {
        tp_dev.scan(0);

        for (t = 0; t < maxp; t++)
        {
            if ((tp_dev.sta) & (1 << t))
            {
                /* 坐标在屏幕范围内 */
                if (tp_dev.x[t] < lcddev.width && tp_dev.y[t] < lcddev.height)
                {
                    if (lastpos[t][0] == 0XFFFF)
                    {
                        lastpos[t][0] = tp_dev.x[t];
                        lastpos[t][1] = tp_dev.y[t];
                    }

                    lcd_draw_bline(lastpos[t][0], lastpos[t][1], tp_dev.x[t],
                                tp_dev.y[t], 2, POINT_COLOR_TBL[t]); /* 画线 */
                }
            }
        }
    }
}

```

```

        lastpos[t][0] = tp_dev.x[t];
        lastpos[t][1] = tp_dev.y[t];

        if (tp_dev.x[t] > (lcddev.width - 24) && tp_dev.y[t] < 20)
        {
            load_draw_dialog(); /* 清除 */
        }
    }
    else
    {
        lastpos[t][0] = 0xFFFF;
    }
}

delay_ms(5);
i++;

if (i % 20 == 0) LED0_TOGGLE();
}

}

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    tp_dev.init(); /* 触摸屏初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32F103", RED);
    lcd_show_string(30, 70, 200, 16, 16, "TOUCH TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    if (tp_dev.touchtype != 0xFF)
    {
        /* 电阻屏才显示 */
        lcd_show_string(30, 110, 200, 16, 16, "Press KEY0 to Adjust", RED);
    }

    delay_ms(1500);
    load_draw_dialog();

    if (tp_dev.touchtype & 0x80)
    {
        ctp_test(); /* 电容屏测试 */
    }
    else
    {
        rtp_test(); /* 电阻屏测试 */
    }
}

```

上面没有把 main.c 全部代码列出来，只是列出重要函数，这里简单介绍一下这三个函数。

rtp_test，该函数用于电阻触摸屏的测试，该函数代码比较简单，就是扫描按键和触摸屏，如果触摸屏有按下，则在触摸屏上面划线，如果按中“RST”区域，则执行清屏。如果按键 KEY0 按下，则执行触摸屏校准。

ctp_test，该函数用于电容触摸屏的测试，由于我们采用 tp_dev.sta 来标记当前按下的触摸屏点数，所以判断是否有电容触摸屏按下，也就是判断 tp_dev.sta 的最低 5 位，如果有数据，则画线，如果没数据则忽略，且 5 个点画线的颜色各不一样，方便区分。另外，电容触摸屏不需要校准，所以没有校准程序。

main 函数，则比较简单，初始化相关外设，然后根据触摸屏类型，去选择执行 ctp_test 还是 rtp_test。

软件部分就介绍到这里，接下来看看下载验证。

39.4 下载验证

在代码编译成功之后，我们通过下载代码到开发板上，电阻触摸屏测试如图 39.4.1 所示界面：



图 39.4.1 电阻触摸屏测试程序运行效果

图中我们在电阻屏上画了一些内容，右上角的 RST 可以用来清屏，点击该区域，即可清屏重画。另外，按 KEY0 可以进入校准模式，如果发现触摸屏不准，则可以按 KEY0，进入校准，重新校准一下，即可正常使用。

如果是电容触摸屏，测试界面如图 39.4.2 所示：

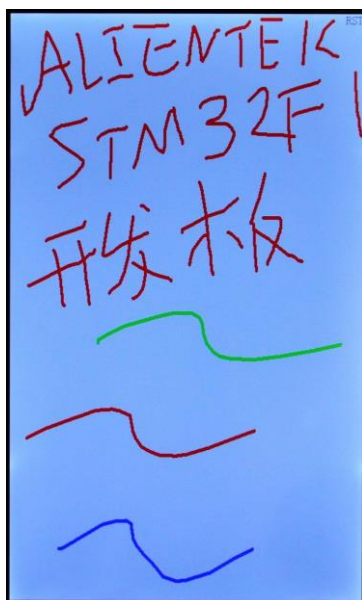


图 39.4.2 电容触摸屏测试界面

图中，同样输入了一些内容。电容屏支持多点触摸，每个点的颜色都不一样，图中的波浪线就是三点触摸画出来的，最多可以 5 点触摸。按右上角的 RST 标志，可以清屏。电容屏无需校准，所以按 KEY0 无效。KEY0 校准仅对电阻屏有效。

第四十章 红外遥控实验

本章，我们将介绍 STM32F103 对红外遥控器的信号解码。STM32 板子上标配的红外接收头和一个小巧的红外遥控器。我们将利用 STM32 的输入捕获功能，解码开发板标配的红外遥控器的编码信号，并将编码后的键值在 LCD 模块中显示出来。

本章分为如下几个小节：

40.1 红外遥控简介

40.2 硬件设计

40.3 程序设计

40.4 下载验证

40.1 红外遥控简介

40.1.1 红外遥控技术介绍

红外遥控是一种无线、非接触控制技术，具有抗干扰能力强，信息传输可靠，功耗低，成本低，易实现等显著优点，被诸多电子设备特别是家用电器广泛采用，并越来越多的应用到计算机系统中。由于红外线遥控不具有像无线电遥控那样穿过障碍物去控制被控对象的能力，所以，在设计红外线遥控器时，不必要像无线电遥控器那样，每套（发射器和接收器）要有不同的遥控频率或编码（否则，就会隔墙控制或干扰邻居的家用电器），所以同类产品的红外线遥控器，可以有相同的遥控频率或编码，而不会出现遥控信号“串门”的情况。这对于大批量生产以及在家用电器上普及红外线遥控提供了极大的方便。由于红外线为不可见光，因此对环境影响很小，再由红外光波动波长远小于无线电波的波长，所以红外线遥控不会影响其他家用电器，也不会影响临近的无线电设备。

40.1.2 红外器件特性

红外遥控的情景中，必定会有一个红外发射端和红外接收端。在本实验中，正点原子的红外遥控器作为红外发射端，红外接收端就是板载的红外接收器，实物图可以查看 40.2.3 小节原理图部分。要使两者通信成功，收/发红外波长与载波频率需一致，在这里波长就是 940nm，载波频率就是 38kHz。

红外发射管也是属于二极管类，红外发射电路通常使用三极管控制红外发射器的导通或者截至，**在导通的时候，红外发射管会发射出红外光，反之，就不会发射出红外光。**虽然我们用肉眼看不到红外光，但是我们借助手机摄像头就能看到红外光。但是红外接收管的特性是**当接收到红外载波信号时，OUT 引脚输出低电平；假如没有接收到红外载波信号时，OUT 引脚输出高电平。**

红外载波信号其实就是由一个个红外载波周期组成。在频率为 38KHz 下，红外载波周期约等于 26.3us ($1s / 38KHz \approx 26.3us$)。在一个红外载波发射周期里，发射红外光时间 8.77us 和不发射红外光 17.53us，发射红外光的占空比一般为 1/3。相对的，整个周期内不发射红外光，就是载波不发射周期。在红外遥控器内已经把载波和不载波信号处理好，我们需要做的就是识别遥控器按键发射出的信号，信号也是遵循某种协议的。

40.1.3 红外编解码协议介绍

红外遥控的编码方式目前广泛使用的是：PWM（脉冲宽度调制）的 NEC 协议和 Philips PPM（脉冲位置调制）的 RC-5 协议的。开发板配套的遥控器使用的是 NEC 协议，其特征如下：

- 1、8 位地址和 8 位指令长度；
- 2、地址和命令 2 次传输（确保可靠性）；
- 3、PWM 脉冲宽度调制，以发射红外载波的占空比代表“0”和“1”；
- 4、载波频率为 38KHz；

5、位时间为 1.125ms 或 2.25ms;

在 NEC 协议中，如何为协议中的数据‘0’或者‘1’？这里分开红外接收器和红外发射器。

红外发射器：发送协议数据‘0’ = 发射载波信号 560us + 不发射载波信号 560us

发送协议数据‘1’ = 发射载波信号 560us + 不发射载波信号 1680us

红外发射器的位定义如下图所示：

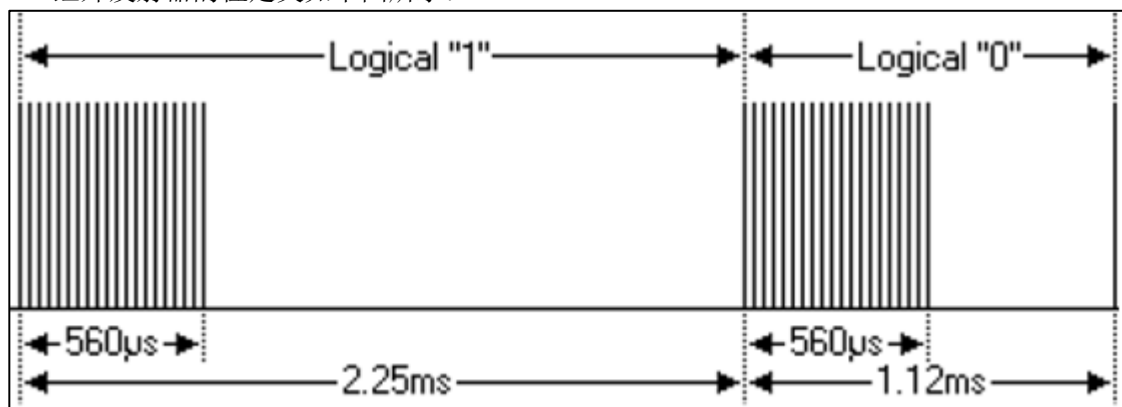


图 40.1.3.1 红外发射器位定义图

红外接收器：接收到协议数据‘0’ = 560us 低电平 + 560us 高电平

接收到协议数据‘1’ = 560us 低电平 + 1680us 高电平

红外接收器的位定义如下图所示：

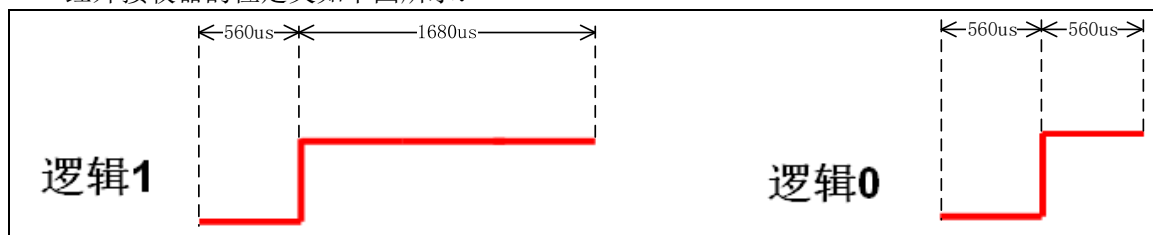


图 40.1.3.2 红外接收器位定义图

NEC 遥控指令的数据格式为：同步码头、地址码、地址反码、控制码、控制反码。同步码由一个 9ms 的低电平和一个 4.5ms 的高电平组成，地址码、地址反码、控制码、控制反码均是 8 位数据格式。按照低位在前，高位在后的顺序发送。采用反码是为了增加传输的可靠性（可用于校验）。

我们遥控器的按键▽按下时，从红外接收头端收到的波形如图 40.1.3.3 所示：

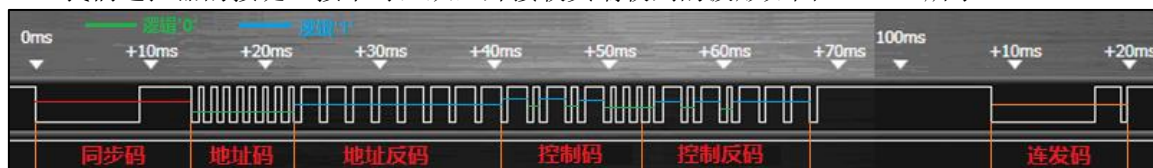


图 40.1.3.3 按键▽所对应的红外波形

从上图中可以看到，其地址码为 0，控制码为 21（正确解码后 00010101）。可以看到在 100ms 之后，我们还收到了几个脉冲，这是 NEC 码规定的连发码(由 9ms 低电平+2.25ms 高电平+0.56ms 低电平+97.94ms 高电平组成)，如果在一帧数据发送完毕之后，按键仍然没有放开，则发射重复码，即连发码可以通过统计连发码的次数来标记按键按下的长短/次数。

第二十一章我们曾经介绍过利用输入捕获来测量高电平的脉宽，本章解码红外遥控信号，刚好可以利用输入捕获的这个功能来实现遥控解码。关于输入捕获的介绍，请参考第二十一章的内容。

40.2 硬件设计

1. 例程功能

本实验开机在 LCD 上显示一些信息之后,即进入等待红外触发,如果接收到正确的红外信号,则解码,并在 LCD 上显示键值和所代表的意义,以及按键次数等信息。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 红外接收头
REMOTE_IN - PB9
- 3) 正点原子红外遥控器
- 4) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 5) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏,16 位 8080 并口驱动)

3. 原理图

红外遥控接收头与 STM32 的连接关系,如下图所示:

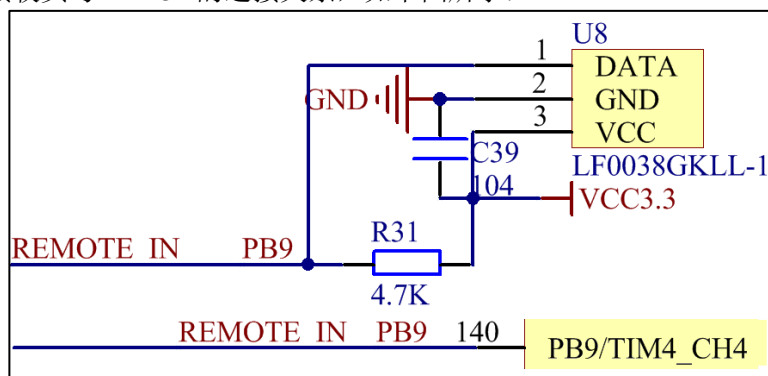


图 40.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头连接在 STM32 的 PB9(TIM4_CH4)上,进行本实验时不需要额外连线。程序将 TIM4_CH4 设计输入捕获,然后将接收到的脉冲信号解码就可以了。

开发板配套的红外遥控器外观如图 40.2.2 所示。



图 40.2.2 红外遥控器

开发板上接收红外遥控器信号的红外管外观如图 40.2.3 所示。使用时需要遥控器有红外管的一端对准开发板上的红外管才能正确收到信号。



图 40.2.3 开发板上的红外接收管位置

40.3 程序设计

由于红外遥控实验采用的是定时器的输入捕获功能，所以这里大家就需要往前面定时器章节输入捕获实验中重温一下输入捕获功能的配置。下面我们也把红外遥控的配置步骤讲解一下。

红外遥控配置步骤

1) 初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

HAL 库通过调用定时器输入捕获初始化函数 HAL_TIM_IC_Init 完成对定时器参数初始化。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数来完成对定时器底层以及其输入通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2) 开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 4 通道 4，对应 IO 是 PB9，它们的时钟开启方法如下：

```
HAL_RCC_TIM4_CLK_ENABLE();          /* 使能定时器 4 */
HAL_RCC_GPIOB_CLK_ENABLE();         /* 开启 GPIOB 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

3) 设置 TIMx_CHy 的输入捕获模式，开启输入捕获

在 HAL 库中，定时器的输入捕获模式是通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器某个通道为输入捕获通道，包括映射关系，输入滤波和输入分频等。

4) 使能定时器更新中断，开启捕获功能以及捕获中断，配置定时器中断优先级

通过 HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start_IT 函数使能定时器并开启捕获功能以及捕获中断。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5) 编写中断服务函数

定时器 4 中断服务函数为：TIM4_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个定时器中断通用处理函数 HAL_TIM_IRQHandler，该函数会调用一些定时器相关的回调函数，用于给用户处理定时器中断到了之后，需要处理的程序。本实验我们除了用到更新（溢出）中断回调函数 HAL_TIM_PeriodElapsedCallback 之外，还要用到捕获中断回调函数 HAL_TIM_IC_CaptureCallback。详见本例程源码。

40.3.1 程序流程图

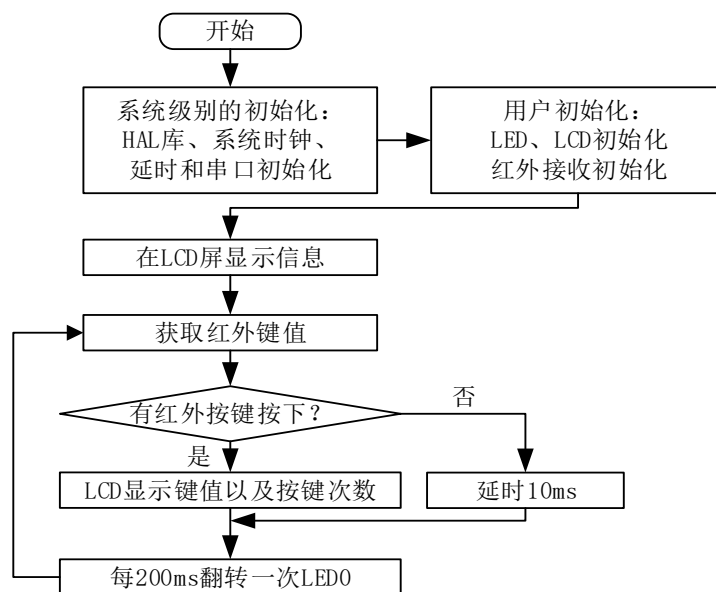


图 40.3.2.1 红外遥控实验程序流程图

40.3.2 程序解析

1. REMOTE 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。REMOTE 驱动源码包括两个文件：remote.c 和 remote.h。remote.h 和前面定时器输入捕获功能的.h 头文件代码相似，这里就不介绍了，详见本例程源码。

下面我们直接介绍 remote.c 的程序，下面是与红外遥控初始化相关的函数，其定义如下：

```
TIM_HandleTypeDef g_tim4_handle; /* 定时器 4 句柄 */
/**
 * @brief      红外遥控初始化
 * @note       设置 IO 以及定时器的输入捕获
 * @param      无
 * @retval     无
 */
void remote_init(void)
{
    TIM_IC_InitTypeDef tim_ic_init_handle;

    g_tim4_handle.Instance = REMOTE_IN_TIMX; /* 通用定时器 4 */
    g_tim4_handle.Init.Prescaler = (72-1); /* 预分频器, 1M 的计数频率, 1us 加 1 */
    g_tim4_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 向上计数器 */
    g_tim4_handle.Init.Period = 10000; /* 自动装载值 */
    g_tim4_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&g_tim4_handle);

    /* 初始化 TIM4 输入捕获参数 */
    tim_ic_init_handle.ICPolarity = TIM_ICPOLARITY_RISING; /* 上升沿捕获 */
    tim_ic_init_handle.ICSelection = TIM_ICSELECTION_DIRECTTI; /* 映射到 TI4 上 */
    tim_ic_init_handle.ICPrescaler = TIM_ICPSC_DIV1; /* 不分频 */
    tim_ic_init_handle.ICFilter = 0x03; /* 8 个定时器时钟周期滤波 */
    HAL_TIM_IC_ConfigChannel(&g_tim4_handle, &tim_ic_init_handle,
                             REMOTE_IN_TIMX_CHY); /* 配置 TIM4 通道 4 */
    HAL_TIM_IC_Start_IT(&g_tim4_handle, REMOTE_IN_TIMX_CHY); /* 开始捕获 TIM 通道 */
    __HAL_TIM_ENABLE_IT(&g_tim4_handle, TIM_IT_UPDATE); /* 使能更新中断 */
}

/**
 * @brief      定时器 4 底层驱动，时钟使能，引脚配置
 * @param      htim: 定时器句柄
 * @note       此函数会被 HAL_TIM_IC_Init() 调用
 * @retval     无
 */
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    if(htim->Instance == REMOTE_IN_TIMX)
    {
        GPIO_InitTypeDef gpio_init_struct;

        REMOTE_IN_GPIO_CLK_ENABLE(); /* 红外接入引脚 GPIO 时钟使能 */
        REMOTE_IN_TIMX_CHY_CLK_ENABLE(); /* 定时器时钟使能 */
        /* 这里用的是 PB9/TIM4_CH4, 参考 AFIO_MAPR 寄存器的设置 */
        __HAL_AFIO_REMAP_TIM4_DISABLE();

        gpio_init_struct.Pin = REMOTE_IN_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_INPUT; /* 复用输入 */
        gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
        HAL_GPIO_Init(REMOTE_IN_GPIO_PORT, &gpio_init_struct); /* 定时器通道引脚 */
    }
}
```



```

/* 设置中断优先级, 抢占优先级 1, 子优先级 3 */
HAL_NVIC_SetPriority(REMOTE_IN_TIMX_IRQn, 1, 3);
HAL_NVIC_EnableIRQ(REMOTE_IN_TIMX_IRQn); /* 开启 ITM4 中断 */
}
}

```

remote_init 函数主要是对红外遥控使用到的定时器 4 和定时器通道 4 进行相关配置, 关于定时器 4 通道 4 的 IO 放在回调函数 HAL_TIM_IC_MspInit 中初始化。

在 remote_init 函数中, 通过调用 HAL_TIM_IC_Init 函数初始化定时器的 ARR 和 PSC 等参数; 通过调用 HAL_TIM_IC_ConfigChannel 函数配置映射关系, 滤波和分频等; 最后调用 HAL_TIM_IC_Start_IT 和 __HAL_TIM_ENABLE_IT 分别使能捕获通道和使能定时器中断。

在 HAL_TIM_IC_MspInit 函数中主要通过 HAL_GPIO_Init 函数对定时器输入通道的 GPIO 口进行配置, 最后还需要设置中断抢占优先级和响应优先级。

通过上面两个函数的配置后, 定时器的输入捕获已经初始化完成, 接下来我们还需要作一些接收处理, 下面先介绍一下三个变量。

```

/* 遥控器接收状态
 * [7] : 收到了引导码标志
 * [6] : 得到了一个按键的所有信息
 * [5] : 保留
 * [4] : 标记上升沿是否已经被捕获
 * [3:0]: 溢出计数器
 */
uint8_t g_remote_sta = 0;
uint32_t g_remote_data = 0; /* 红外接收到的数据 */
uint8_t g_remote_cnt = 0; /* 按键按下的次数 */

```

这三个变量用于辅助实现高电平的捕获。其中 g_remote_sta 是用来记录捕获状态, 这个变量, 我们把它当成一个寄存器来使用。对其各位进行定义, 描述如下表所示:

| g_remote_sta | | | | |
|--------------|------------|------|--------------|--------|
| bit7 | bit6 | bit5 | bit4 | Bit3~0 |
| 收到引导码 | 得到一个按键所有信息 | 保留 | 标记上升沿是否已经被捕获 | 溢出计数器 |

表 40.3.2.1 g_remote_sta 各位描述

变量 g_remote_data 用于存放红外接收到的数据, 而 g_remote_cnt 是存放按键按下的次数。

下面开始看中断服务函数里面的逻辑程序, HAL_TIM_IRQHandler 函数会调用下面两个回调函数, 我们的逻辑代码就是放在回调函数里, 函数的定义如下:

```

/**
 * @brief 定时器输入捕获中断回调函数
 * @param htim: 定时器句柄
 * @retval 无
 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == REMOTE_IN_TIMX)
    {
        uint16_t dval; /* 下降沿时计数器的值 */

        if (RDATA) /* 上升沿捕获 */
        {
            __HAL_TIM_SET_CAPTUREPOLARITY(&g_tim4_handle, REMOTE_IN_TIMX_CHY,
                                           TIM_INPUTCHANNELPOLARITY_FALLING); /* 设置为下降沿捕获 */
            __HAL_TIM_SET_COUNTER(&g_tim4_handle, 0); /* 清空定时器计数器值 */
            g_remote_sta |= 0X10; /* 标记上升沿已经被捕获 */
        }
        else /* 下降沿捕获 */
        {
            /* 读取 CCR4 也可以清 CC4IF 标志位 */
            dval = HAL_TIM_ReadCapturedValue(&g_tim4_handle, REMOTE_IN_TIMX_CHY);
            __HAL_TIM_SET_CAPTUREPOLARITY(&g_tim4_handle, REMOTE_IN_TIMX_CHY,
                                           TIM_INPUTCHANNELPOLARITY_RISING); /* 配置 TIM4 通道 4 上升沿捕获 */
        }
    }
}

```



```

if (g_remote_sta & 0X10)          /* 完成一次高电平捕获 */
{
    if (g_remote_sta & 0X80)      /* 接收到了引导码 */
    {
        if (dval > 300 && dval < 800) /* 560 为标准值,560us */
        {
            g_remote_data >>= 1;    /* 右移一位. */
            g_remote_data &= ~0x80000000; /* 接收到 0 */
        }
        else if (dval > 1400 && dval < 1800) /* 1680 为标准值,1680us */
        {
            g_remote_data >>= 1;    /* 右移一位 */
            g_remote_data |= 0x80000000; /* 接收到 1 */
        }
        else if (dval > 2000 && dval < 3000)
        {
            /* 得到按键键值增加的信息 2250 为标准值 2.25ms */
            g_remote_cnt++;          /* 按键次数增加 1 次 */
            g_remote_sta &= 0XF0;    /* 清空计数器 */
        }
    }
    else if (dval > 4200 && dval < 4700) /* 4500 为标准值 4.5ms */
    {
        g_remote_sta |= 1 << 7;    /* 标记成功接收到了引导码 */
        g_remote_cnt = 0;          /* 清除按键次数计数器 */
    }
}
g_remote_sta &= ~(1<<4);
}
}
}

```

现在我们来介绍一下，捕获高电平脉宽的思路：首先，设置 TIM4_CH4 捕获上升沿，然后等待上升沿中断到来，当捕获到上升沿中断，设置该通道为下降沿捕获，清除 TIM4_CNT 寄存器的值，最后把 g_remote_sta 的位 4 置 1，表示已经捕获到高电平，等待下降沿到来。当下降沿到来的时候，读取此时定时器计数器的值到 dval 中并设置该通道为上升沿捕获，然后判断 dval 的值属于哪个类型（引导码，数据 0，数据 1 或者重发码），相对应就把 g_remote_sta 相关位进行调整。例如，一开始识别为引导码的情况，就需要把 g_remote_sta 第 7 位置 1。当检测到重复码，就把按键次数增量存放在 g_remote_cnt 变量中。

```

/**
 * @brief      定时器更新中断回调函数
 * @param      htim:定时器句柄
 * @retval     无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == REMOTE_IN_TIMX)
    {
        if (g_remote_sta & 0x80)          /* 上次有数据被接收到了 */
        {
            g_remote_sta &= ~0X10;        /* 取消上升沿已经被捕获标记 */

            if ((g_remote_sta & 0X0F) == 0X00)
            {
                g_remote_sta |= 1 << 6;    /* 标记已经完成一次按键的键值信息采集 */
            }

            if ((g_remote_sta & 0X0F) < 14)
            {
                g_remote_sta++;
            }
            else
            {

```

```

        g_remote_sta &= ~(1 << 7);    /* 清空引导标识 */
        g_remote_sta &= 0XF0;        /* 清空计数器 */
    }
}
}

```

定时器更新中断回调函数主要是对标志位进行管理。在函数内通过 `g_remote_sta` 标志的判断，主要思路就是：在接收到引导码的前提下，对 `g_remote_sta` 状态进行判断并在符合条件下进行运算，这里主要就做了两件事：标记完成一次按键信息采集和是否松开按键（即没有接收到数据）。当完成一次按键信息采集时，`g_remote_data` 已经存放了控制反码、控制码、地址反码、地址码。那为啥可以检测是否可以松开按键？是因为接收到重发码的情况下会清空计数器，所以说当我们松开按键接收不到重发码时，溢出中断次数增多最终会导致 `g_remote_sta&0x0f` 值大于 14，进而就可以把引导码，计数器清空，便于下一次的接收。

```

/**
 * @brief      处理红外按键 (类似按键扫描)
 * @param      无
 * @retval     0 , 没有任何按键按下
 *            其他, 按下的按键键值
 */
uint8_t remote_scan(void)
{
    uint8_t sta = 0;
    uint8_t t1, t2;

    if (g_remote_sta & (1 << 6))    /* 得到一个按键的所有信息了 */
    {
        t1 = g_remote_data;        /* 得到地址码 */
        t2 = (g_remote_data >> 8) & 0xff;    /* 得到地址反码 */

        if ((t1 == (uint8_t)~t2) && t1 == REMOTE_ID)
        { /* 检验遥控识别码 (ID) 及地址 */
            t1 = (g_remote_data >> 16) & 0xff;
            t2 = (g_remote_data >> 24) & 0xff;
            if (t1 == (uint8_t)~t2)
            {
                sta = t1;        /* 键值正确 */
            }
        }

        if ((sta == 0) || ((g_remote_sta & 0X80) == 0))
        { /* 按键数据错误/遥控已经没有按下了 */
            g_remote_sta &= ~(1 << 6);    /* 清除接收到有效按键标识 */
            g_remote_cnt = 0;        /* 清除按键次数计数器 */
        }
    }

    return sta;
}

```

`remote_scan` 函数是用来扫描解码结果的，相当于我们的按键扫描，输入捕获解码的红外数据，通过该函数传送给其他程序。

2. main.c 代码

在 `main.c` 里面编写如下代码：

```

int main(void)
{
    uint8_t key;
    uint8_t t = 0;
    char *str = 0;

    HAL_Init();    /* 初始化 HAL 库 */
}

```

```

sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
delay_init(72); /* 延时初始化 */
usart_init(115200); /* 串口初始化为 115200 */
led_init(); /* 初始化 LED */
lcd_init(); /* 初始化 LCD */
remote_init(); /* 红外接收初始化 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "REMOTE TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "KEYVAL:", RED);
lcd_show_string(30, 130, 200, 16, 16, "KEYCNT:", RED);
lcd_show_string(30, 150, 200, 16, 16, "SYMBOL:", RED);

while (1)
{
    key = remote_scan();

    if (key)
    {
        lcd_show_num(86, 110, key, 3, 16, BLUE); /* 显示键值 */
        lcd_show_num(86, 130, g_remote_cnt, 3, 16, BLUE); /* 显示按键次数 */

        switch (key)
        {
            case 0: str = "ERROR"; break;
            case 69: str = "POWER"; break;
            case 70: str = "UP"; break;
            case 64: str = "PLAY"; break;
            case 71: str = "ALIENTEK"; break;
            case 67: str = "RIGHT"; break;
            case 68: str = "LEFT"; break;
            case 7: str = "VOL-"; break;
            case 21: str = "DOWN"; break;
            case 9: str = "VOL+"; break;
            case 22: str = "1"; break;
            case 25: str = "2"; break;
            case 13: str = "3"; break;
            case 12: str = "4"; break;
            case 24: str = "5"; break;
            case 94: str = "6"; break;
            case 8: str = "7"; break;
            case 28: str = "8"; break;
            case 90: str = "9"; break;
            case 66: str = "0"; break;
            case 74: str = "DELETE"; break;
        }

        lcd_fill(86, 150, 116 + 8 * 8, 170 + 16, WHITE); /* 清楚之前的显示 */
        lcd_show_string(86, 150, 200, 16, 16, str, BLUE); /* 显示 SYMBOL */
    }
    else
    {
        delay_ms(10);
    }

    t++;

    if (t == 20)
    {
        t = 0;
        LED0_TOGGLE(); /* LED0 闪烁 */
    }
}
}

```

main 函数代码比较简单,主要是通过 remote_scan 函数获得红外遥控输入的数据(控制码),然后显示在 LCD 上面。正点原子红外遥控器按键对应的控制码图如下图所示。



图 40.3.2.1 红外遥控器按键对应的控制码图（十六进制数）

特别注意:上图中的控制码数值是十六进制的,而我们代码中使用的是十进制的表示方式。此外,正点原子红外遥控器的地址码是 0。

40.4 下载验证

将程序下载到开发板后,可以看到 LED0 不停的闪烁,提示程序已经在运行了。LCD 显示的内容如图 40.4.1 所示:

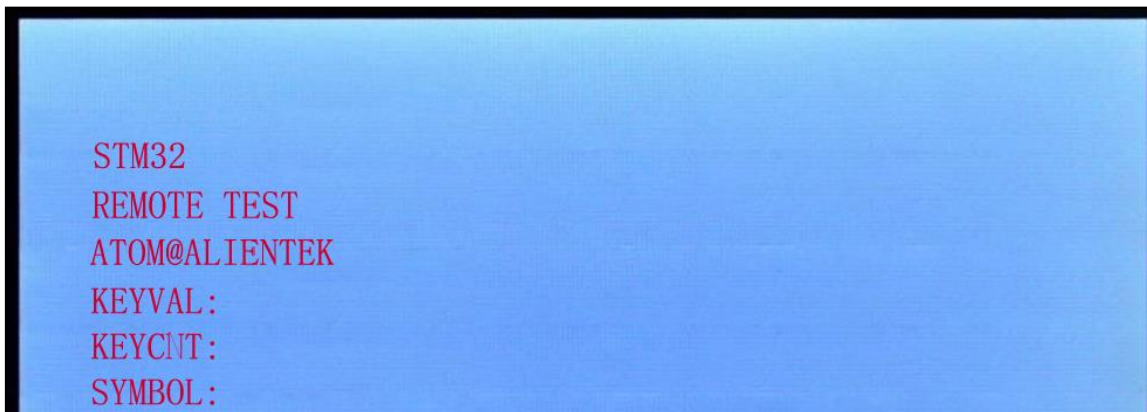


图 40.4.1 程序运行效果图

此时我们通过遥控器按下不同的按键,则可以看到 LCD 上显示了不同按键的键值以及按键次数和对应的遥控器上的符号。如图 40.4.2 所示。

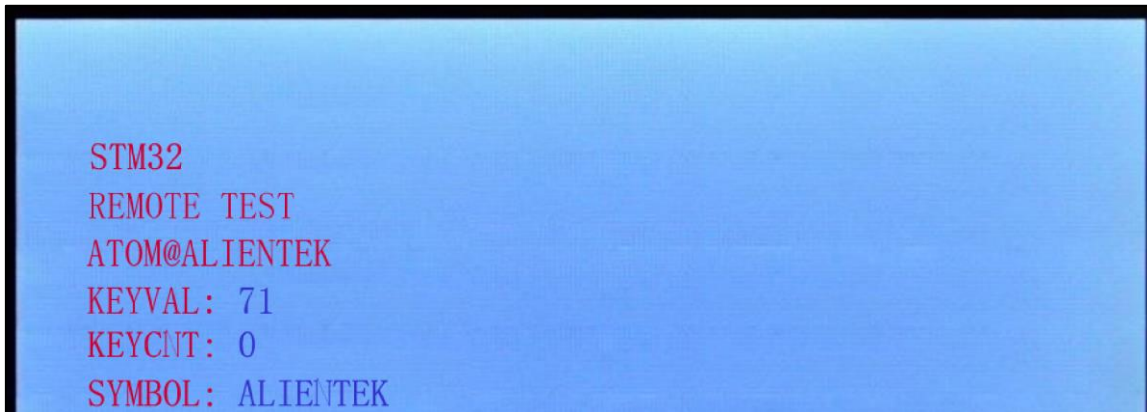


图 40.4.2 解码成功

第四十一章 游戏手柄实验

FC 游戏机（又称：红白机/小霸王游戏机）发行了很多经典的游戏，给不少人的童年留下了无限乐趣。本章，我们将向大家介绍如何通过 STM32 来驱动 FC 游戏机手柄，将 FC 游戏机的手柄作为战舰 STM32 开发板的输入设备（综合实验可以直接通过这个手柄来玩 FC 游戏）。

在本章中，我们将使用 STM32 驱动 FC 手柄，将手柄的按键键值等信息通过 TFT LCD 模块显示出来。本章分为如下几个部分：

- 41.1 游戏手柄简介
- 41.2 硬件设计
- 41.3 软件设计
- 41.4 下载验证

41.1 游戏手柄简介

FC 游戏机曾经是一统天下（现在也还是很多人玩），红极一时，那时任天堂单是 FC 主机的主机的发售收入就超过全美国的电视台的收入总和。本章，我们将使用 STM32 来驱动 FC 手柄，实现手柄控制信号的读取，我们先来了解一下 FC 手柄。

FC 手柄，大致可分为两种：一种手柄插口是 11 针的，一种是 9 针的。但 11 针的现在市面上很少了（因为 11 针手柄是早期 FC 组装兼容机最主要的周边），现在几乎都是 9 针 FC 组装手柄的天下，所以我们本章使用的是 9 针 FC 手柄，该手柄还有一个特点，就是可以直接和 DB9 的串口公头对接！这样与开发板的连接就简单了。FC 手柄的外观如图 41.1.1 所示：



图 41.1.1 FC 手柄外观图

这种手柄一般有 10 个按键（实际是 8 个键值）：上、下、左、右、Start、Select、A、B、A 连发、B 连发。这里的 A 和 A 连发是一个键值，而 B 和 B 连发也是一个键值，只是连发按键当你一直按下的时候，会不停的发送（方便快速按键，比如发炮弹之类的功能）。

FC 手柄的控制电路，由 1 个 8 位并入串出的移位寄存器（CD4021），外加一个时基集成电路（NE555，用于连发）构成。不过现在的手柄，为了节约成本，直接就在 PCB 上做绑定了，所以你拆开手柄，一般是看不到里面有四四方方的 IC，而只有一个黑色的小点，所有电路都集成到这个里面了，但是他们的控制和读取方法还是一样的。

9 针手柄实际上只有 5 根线起作用，不同的文档里命名会有一些差别，分别如下：

VCC = 5V 供电

GND = 地线

LATCH = 锁存信号，由主机发送

CLOCK = 时钟信号，有些文档会叫 PULSE，由主机发送

DATA = 串行数据线 低电平有效。

我们可以把它看为键盘，标准的 FC 手柄的控制读取时序和接线图如图 41.1.2 所示：

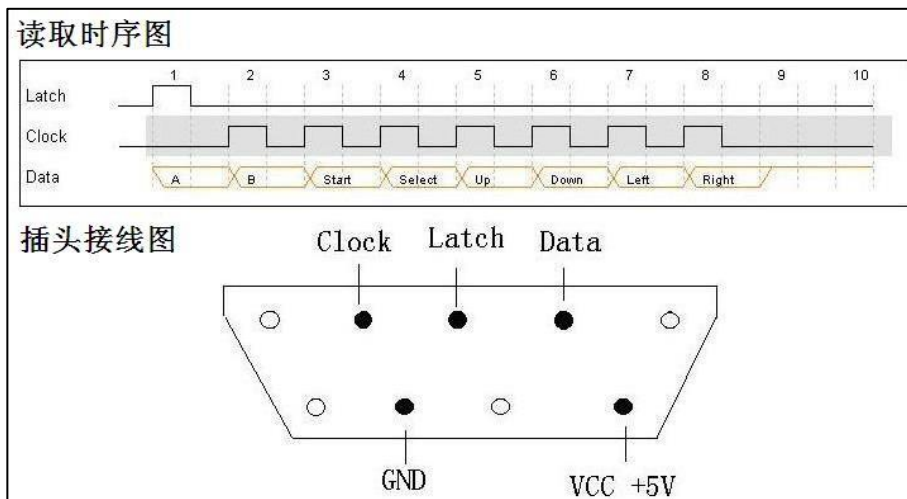


图 41.1.2 FC 手柄读取时序和接线图

从上图可看出，读取手柄按键值的信息十分简单：先 Latch（锁存键值），然后就得到了第一个按键值（A），之后在 Clock 的作用下，依次读取其他按键的键值，总共 8 个按键键值。标准的 NES 手柄高低电平周期 12us，占空比为 50%，且按键按下后 DATA 上的电平为负，不同手柄可能有差异，但时序差不多，我们编程时参考这个时序来实现即可。

有了以上了解，我们就可以通过 STM32 的 IO 来驱动 FC 手柄了。

41.2 硬件设计

1. 例程功能

本实验采用 STM32 的 3 个普通 IO 连接 FC 手柄的 Clock、Data 和 Latch 信号，本章实验功能简介：在主函数不停的查询手柄输入，一旦检测到输入信号，则在 LCD 模块上面显示键值和对应的按键符号。同样我们也是用 LED0 来指示程序正在运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) FC 游戏手柄

Clock (PD3)、Data (PB10) 和 Latch (PB11)

战舰 STM32 开发板板载了一个 FC 手柄接口 (COM3)，其实就是一个 DB9 接公头插座，FC 手柄接口和 COM3 公用一个接口，通过开发板上的 K1 开关来选择，如图 41.2.1 所示。

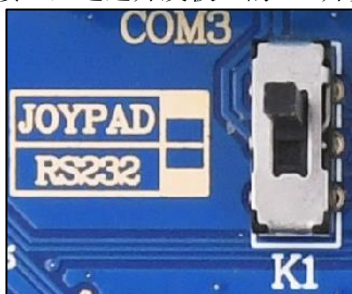


图 41.2.1 COM3 功能选择示意图

当 K1 打到上面 (JOYPAD) 时，COM3 作为 FC 手柄接口，当 K1 打到下面 (RS232) 时，COM3 作为 RS232 串口。COM3 接口与 MCU 的连接原理图如 41.2.2 所示：

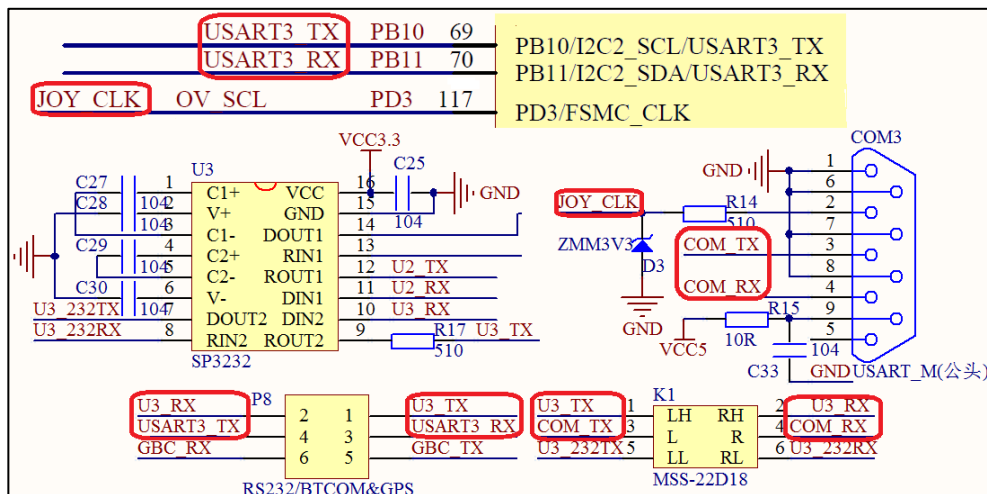


图 41.2.2 FC 手柄接头与 STM32 的连接电路图

图 41.2.2 中，COM3 就是用来连接 FC 手柄，该接头采用标准的 DR9 座，当 K1 开关打到上面的时候，COM_TX 连接 U3_TX、COM_RX 连接 U3_RX，然后通过 P8，连接在 PB11 和 PB10 上面。所以要将 FC 手柄通过 COM3 连接在 STM32 上面，必须 K1 开关打到上面，并且 P8 需要用跳线帽连接 PB10(TX)和 COM3_RX、PB11(RX)和 COM3_TX。

图中的 D3 稳压二极管，是为了防止 COM3 做 RS232 使用时，高压烧坏 MCU 的 IO 口。

本例程使用 FC 手柄（JOYPAD）功能时，COM_TX 是 LAT（Latch）信号，COM_RX 是 DAT（Data）信号，JOY_CLK 是 CLK（Clock）信号，分别连接在 STM32 的 PB11、PB10 和 PD3 上面，这里 JOY_CLK 和 OV_SCL 信号线共用 PD3，所以 FC 手柄和摄像头模块得分时复用 PD3 才可以。

在设置好开发板的连接后(P8跳线帽:PB10(TX)和COM3_RX连接、PB11(RX)和COM3_TX连接, K1 开关打 JOYPAD 位置), 将 FC 手柄插入 COM3 插口即可。

41.3 程序设计

41.3.1 程序流程图

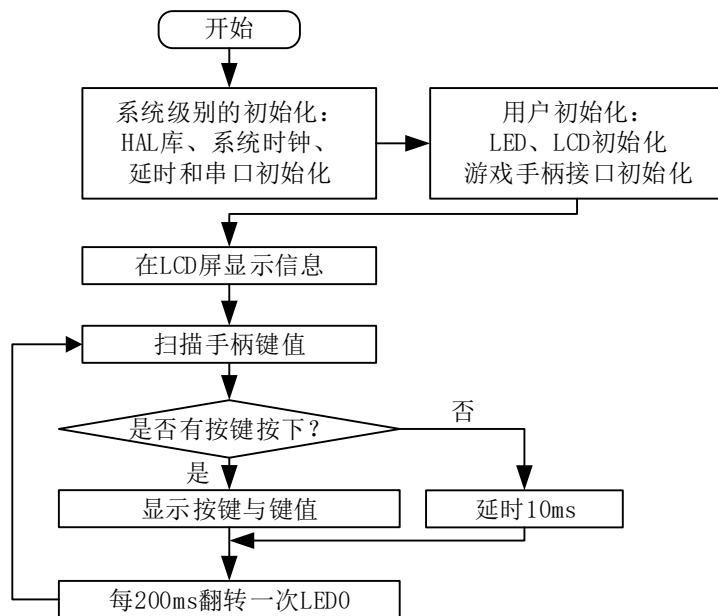


图 41.3.1.1 游戏手柄实验程序流程图

41.3.2 程序解析

1. joypad 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。JOYPAD 驱动源码包括两个文件：joypad.c 和 joypad.h。joypad.h 和前面定时器输入捕获功能的.h 头文件代码相似，这里就不介绍了，详见本例程源码。

下面我们直接介绍 joypad.c 的程序，下面是与红外遥控初始化相关的函数，其定义如下：

```
/**
 * @brief      初始化手柄接口
 * @param      无
 * @retval     无
 */
void joypad_init(void)
{
    JOYPAD_CLK_GPIO_CLK_ENABLE(); /* CLK 所在 IO 时钟初始化 */
    JOYPAD_LAT_GPIO_CLK_ENABLE(); /* LATCH 所在 IO 时钟初始化 */
    JOYPAD_DATA_GPIO_CLK_ENABLE(); /* DATA 所在 IO 时钟初始化 */

    GPIO_InitTypeDef gpio_init_struct;
    gpio_init_struct.Pin = JOYPAD_CLK_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM;
    HAL_GPIO_Init(JOYPAD_CLK_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = JOYPAD_LAT_GPIO_PIN;
    HAL_GPIO_Init(JOYPAD_LAT_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = JOYPAD_DATA_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_INPUT;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM;
    HAL_GPIO_Init(JOYPAD_DATA_GPIO_PORT, &gpio_init_struct);
}

/**
 * @brief      手柄延迟函数
 * @param      t    : 要延时的时间
 * @retval     无
 */
static void joypad_delay(uint16_t t)
{
    while (t--);
}

/**
 * @brief      读取手柄按键值
 * @note      FC 手柄数据输出格式：
 *            每给一个脉冲，输出一位数据，输出顺序：
 *            A -> B -> SELECT -> START -> UP -> DOWN -> LEFT -> RIGHT.
 *            总共 8 位，对于有 C 按钮的手柄，按下 C 其实就等于 A + B 同时按下。
 *            按下是 1，松开是 0。
 * @param      无
 * @retval     按键结果，格式如下：
 *            [7]:右
 *            [6]:左
 *            [5]:下
 *            [4]:上
 *            [3]:Start
 *            [2]:Select
 */
```

```

*           [1]:B
*           [0]:A
*/
uint8_t joypad_read(void)
{
    volatile uint8_t temp = 0;
    uint8_t t;

    JOYPAD_LAT(1);                /* 锁存当前状态 */
    joypad_delay(80);
    JOYPAD_LAT(0);

    for (t = 0; t < 8; t++)        /* 移位输出数据 */
    {
        temp >>= 1;
        if (JOYPAD_DATA == 0)
        {
            temp |= 0x80;          /* LOAD 之后, 就得到第一个数据 */
        }

        JOYPAD_CLK(1);            /* 每给一次脉冲, 收到一个数据 */
        joypad_delay(80);
        JOYPAD_CLK(0);
        joypad_delay(80);
    }

    return temp;
}

```

有了以上函数, 那我们在应用程序中初始化控制手柄的 IO 后, 就可以利用 joypad_read() 返回的键值来设计其它的应用程序了, 如同我们使用开发板上的按键一样简单。

2. main.c 代码

在 main.c 里面编写如下代码。

```

/* 手柄按键符号定义 */
const char *JOYPAD_SYMBOL_TBL[8] = {"Right", "Left", "Down", "Up",
                                     "Start", "Select", "A", "B"};

int main(void)
{
    uint8_t key;
    uint8_t t = 0, i = 0;

    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72);              /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    joypad_init();               /* 游戏手柄初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "JOYPAD TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEYVAL:", RED);
    lcd_show_string(30, 130, 200, 16, 16, "SYMBOL:", RED);

    while (1)
    {
        key = joypad_read();

        if (key)                /* 手柄有按键按下 */
        {

```

```

    lcd_show_num(116, 130, key, 3, 16, BLUE);    /* 显示键值 */

    for (i = 0; i < 8; i++)
    {
        if (key & (0X80 >> i))
        {
            /* 清除之前的显示 */
            lcd_fill(30 + 56, 130, 30 + 56 + 48, 150 + 16, WHITE);
            lcd_show_string(30 + 56, 130, 200, 16, 16,
                           (char *)JOYPAD_SYMBOL_TBL[i], BLUE); /*显示符号*/
        }
    }

    delay_ms(10);
    t++;

    if (t == 20)
    {
        t = 0;
        LED0_TOGGLE(); /* LED0 闪烁 */
    }
}

```

此部分代码也比较简单，初始化 JOYPAD 之后，就一直扫描 FC 手柄 (joypad_read 函数)，然后只要接收到手柄的有效型号，就在 LCD 模块上显示出来。

41.4 下载验证

在代码编译成功之后，我们通过下载代码到正点原子战舰 STM32 开发板上，可以看到 LCD 显示如图 41.4.1 所示的内容：



图 41.4.1 程序运行效果图

此时我们按下 FC 手柄的按键，则可以看到 LCD 上显示了对应按键的键值以及对应的符号。如图 41.4.2 所示：



图 41.4.2 解码游戏手柄数据成功

第四十二章 DS18B20 数字温度传感器实验

本章，我们将介绍 STM32F103 如何读取外部温度传感器的温度，来得到较为准确的环境温度。我们将学习单总线技术，通过它来实现 STM32 和外部温度传感器 DS18B20 的通信，并把从温度传感器得到的温度显示在 LCD 上。

本章分为如下几个小节：

42.1 DS18B20 及工作时序简介

42.2 硬件设计

42.3 程序设计

42.4 下载验证

42.1 DS18B20 及工作时序简介

42.1.1 DS18B20 简介

DS18B20 是由 DALLAS 半导体公司推出的一种“单总线”接口的温度传感器。与传统的热敏电阻等测温元件相比，它是一种新型的体积小、适用电压宽、与微处理器接口简单的数字化温度传感器。单总线结构具有简洁且经济的特点，可使用户轻松地组建传感器网络，从而为测量系统的构建引入全新的概念，测试温度范围为 $-55\sim+125^{\circ}\text{C}$ ，精度为 $\pm 0.5^{\circ}\text{C}$ 。现场温度直接以单总线的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~12 位的数字值读数方式。它的工作电压范围为 3~5.5V，采用多种封装形式，从而使系统设置灵活、方便，设定分辨率以及用户设定的报警温度存储在 EEPROM 中，掉电后依然保存。其内部结构如图 42.1.1.1 所示：

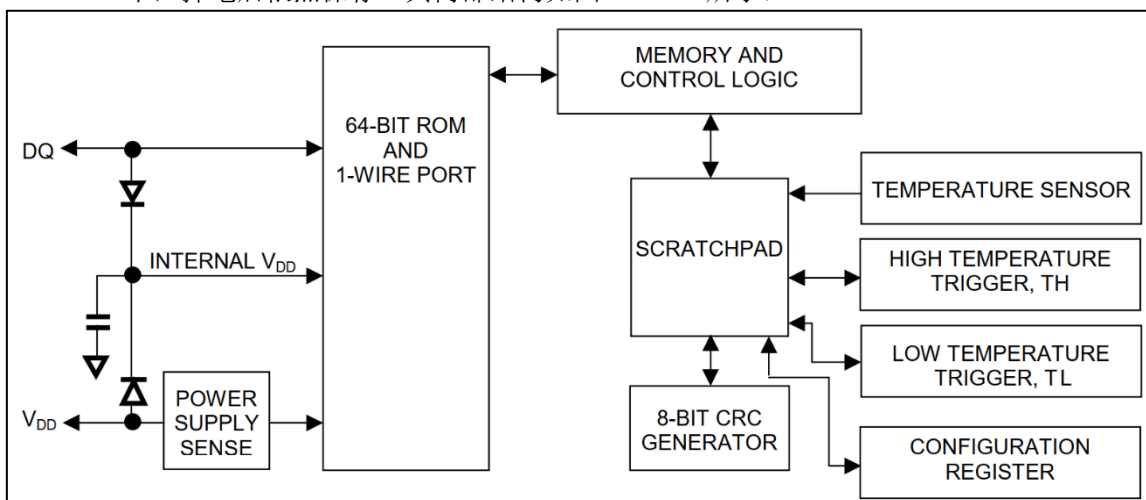


图 42.1.1.1 DS18B20 内部结构图

ROM 中的 64 位序列号是出厂前被设置好的，它可以看作是该 DS18B20 的地址序列码，每个 DS18B20 的 64 位序列号均不相同。64 位 ROM 的排列是：前 8 位是产品家族码，接着 48 位是 DS18B20 的序列号，最后 8 位是前面 56 位的循环冗余校验码($\text{CRC}=\text{X8}+\text{X5}+\text{X4}+1$)。ROM 作用是使每一个 DS18B20 都各不相同，这样设计可以允许一根总线上挂载多个 DS18B20 模块同时工作且不会引起冲突。

42.1.2 DS18B20 工作时序简介

所有单总线器件要求采用严格的信号时序，以保证数据的完整性。DS18B20 共有 6 种信号类型：复位脉冲、应答脉冲、写 0、写 1、读 0 和读 1。所有这些信号，除了应答脉冲以外，都是由主机发出同步信号。并且发送所有的命令和数据都是字节的低位在前。这里我们简单介绍

这几个信号的时序。

1) 复位脉冲和应答脉冲

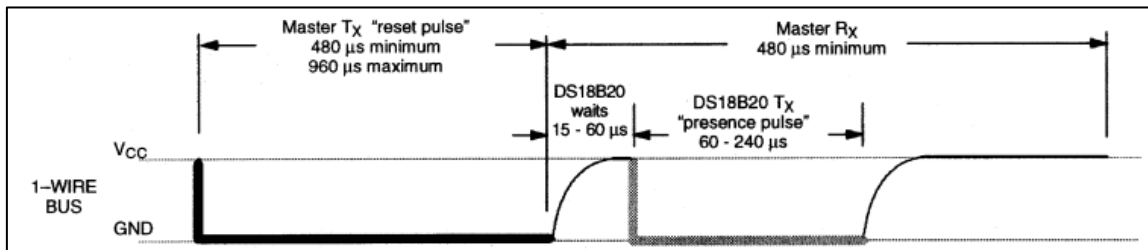


图 42.1.2.1 复位脉冲和应答脉冲时序图

单总线上的所有通信都是以初始化序列开始。主机输出低电平，保持低电平时间至少要在 480us，以产生复位脉冲。接着主机释放总线，4.7K 的上拉电阻将单总线拉高，延时时间要在 15~60us，并进入接收模式（Rx）。接着 DS18B20 拉低总线 60~240us，以产生低电平应答脉冲。

2) 写时序

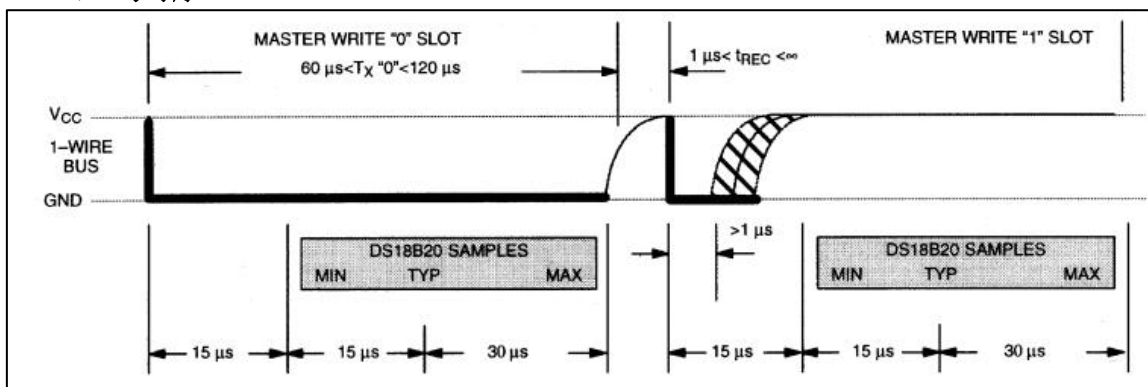


图 42.1.2.2 写时序图

写时序包括写 0 时序和写 1 时序。所有写时序至少需要 60us，且在两次独立的写时序之间至少需要 1us 的恢复时间，两种写时序均起始于主机拉低总线。写 1 时序：主机输出低电平，延时 2us，然后释放总线，延时 60us。写 0 时序：主机输出低电平，延时 60us，然后释放总线延时 2us。

3) 读时序

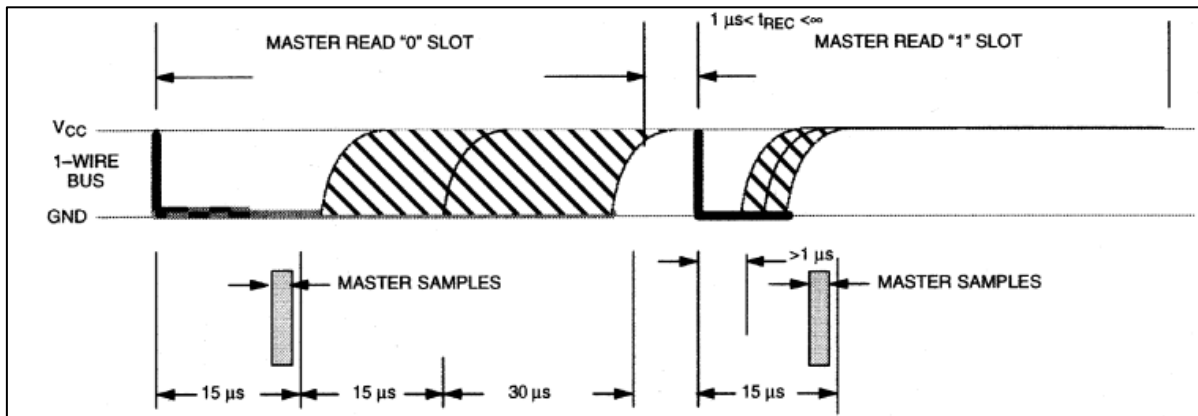


图 42.1.2.3 读时序图

单总线器件仅在主机发出读时序时，才向主机传输数据，所以，在主机发出读数据命令后，必须马上产生读时序，以便从机能够传输数据。所有读时序至少需要 60us，且在 2 次独立的读时序之间至少需要 1us 的恢复时间。每个读时序都由主机发起，至少拉低总线 1us。主机在读时序期间必须释放总线，并且在时序起始后的 15us 之内采样总线状态。典型的读时序过程为：主机输出低电平延时 2us，然后主机转入输入模式延时 12us，然后读取单总线当前的电平，然后延时 50us。

在了解单总线时序之后，我们来看一下 DS18B20 的典型温度读取过程，DS18B20 的典型温度读取过程为：复位→发 SKIPROM (0xCC)→发开始转换命令 (0x44)→延时→复位→发送 SKIPROM 命令 (0xCC)→发送存储器命令 (0xBE)→连续读取两个字节数据（即温度）→结束。

42.2 硬件设计

1. 例程功能

本实验开机的时候先检测是否有 DS18B20 存在，如果没有，则提示错误。只有在检测到 DS18B20 之后才开始读取温度并显示在 LCD 上，如果发现了 DS18B20，则程序每隔 100ms 左右读取一次数据，并把温度显示在 LCD 上。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) DS18B20 温度传感器 - PG11
- 3) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)

3. 原理图

DS18B20 接口与 STM32 的连接关系，如下图所示：

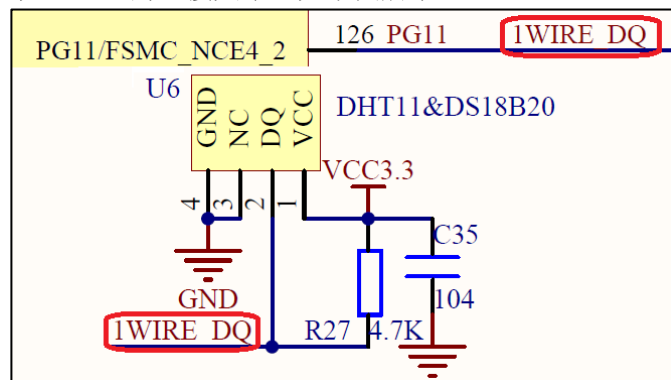


图 42.2.1 DS18B20 连接原理

从上图可以看出，我们使用的是 STM32 的 PG11 来连接 U6 的 DQ 引脚，图中 U6 为 DHT11（数字温湿度传感器）和 DS18B20 共用的一个接口，DHT11 我们将在下一章介绍。

DS18B20 只用到 U6 的 3 个引脚（U6 的 1、2 和 3 脚），将 DS18B20 传感器插入到这个上面就可以通过 STM32 来读取 DS18B20 的温度了。连接示意图如图 42.2.2 所示：

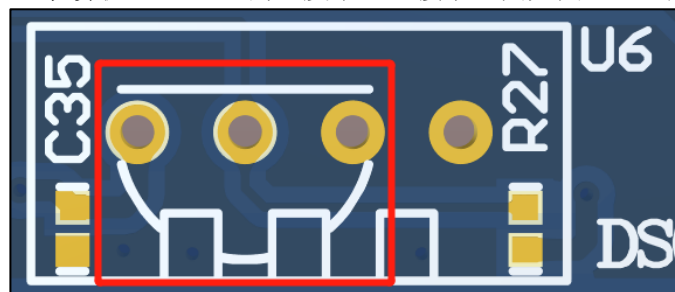


图 42.2.2 DS18B20 连接示意图

从上图可以看出，DS18B20 的平面部分（有字的那面）应该朝内，而曲面部分朝外。然后插入如图所示的三个孔内。

42.3 程序设计

DS18B20 实验中使用的是单总线协议，用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下如何驱动 DS18B20。

DS18B20 配置步骤

1) 使能 DS18B20 数据线对应的 GPIO 时钟。

本实验中 DS18B20 的数据线引脚是 PG11，因此需要先使能 GPIOG 的时钟，代码如下：

```
HAL_RCC_GPIOG_CLK_ENABLE(); /* PG 口时钟使能 */
```

2) 设置对应 GPIO 工作模式（开漏输出）

本实验 GPIO 使用开漏输出模式，通过函数 HAL_GPIO_Init 设置实现。

3) 参考单总线协议，编写信号函数（复位脉冲、应答脉冲、写 0/1、读 0/1）

复位脉冲：主机发出低电平，保持低电平时间至少 480us。

应答脉冲：DS18B20 拉低总线 60~240us，以产生低电平应答信号。

写 1 信号：主机输出低电平，延时 2us，然后释放总线，延时 60us。

写 0 信号：主机输出低电平，延时 60us，然后释放总线，延时 2us。

读 0/1 信号：主机输出低电平延时 2us，然后主机转入输入模式延时 12us，然后读取单总线当前的电平，然后延时 50us。

4) 编写 DS18B20 的读和写函数

基于写 1bit 数据和读 1bit 数据的基础上，编写 DS18B20 写 1 字节和读 1 字节函数。

5) 编写 DS18B20 获取温度函数

参考 DS18B20 典型温度读取过程，编写获取温度函数。

42.3.1 程序流程图

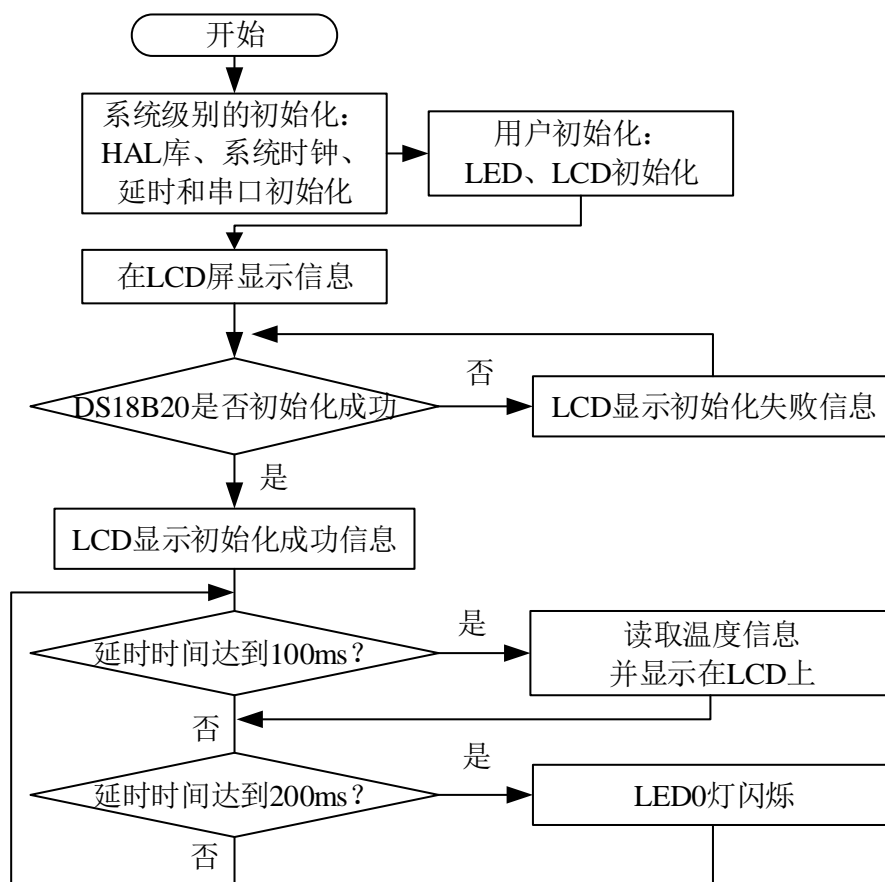


图 42.3.2.1 DS18B20 实验程序流程图

42.3.2 程序解析

1.DS18B20 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。温度传感器驱动源码包括两个文件：ds18b20.c 和 ds18b20.h。

首先我们先看一下 ds18b20 头文件里面的内容，其定义如下：

```
/* DS18B20 引脚 定义 */
#define DS18B20_DQ_GPIO_PORT      GPIOG
#define DS18B20_DQ_GPIO_PIN      GPIO_PIN_11
#define DS18B20_DQ_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOG_CLK_ENABLE();\
                                         }while(0) /* PG 口时钟使能 */

/* IO 操作函数 */
#define DS18B20_DQ_OUT(x) do{ x ? \
    HAL_GPIO_WritePin(DS18B20_DQ_GPIO_PORT,DS18B20_DQ_GPIO_PIN, GPIO_PIN_SET):\
    HAL_GPIO_WritePin(DS18B20_DQ_GPIO_PORT, DS18B20_DQ_GPIO_PIN, GPIO_PIN_RESET);\
    }while(0)

/* 数据端口输出 */
#define DS18B20_DQ_IN      HAL_GPIO_ReadPin(DS18B20_DQ_GPIO_PORT, \
                                              DS18B20_DQ_GPIO_PIN) /* 数据端口输入 */
```

在 ds18b20.h 的操作跟 IIC 实验代码很类似，主要对用到 GPIO 口进行宏定义，以及宏定义 IO 操作函数，方便时序函数调用。

下面我们直接介绍 ds18b20.c 的程序，首先先介绍一下 DS18B20 传感器的初始化函数，其定义如下：

```
/**
 * @brief      初始化 DS18B20 的 IO 口 DQ 同时检测 DS18B20 的存在
 * @param      无
 * @retval     0, 正常
 *            1, 不存在/不正常
 */
uint8_t ds18b20_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    DS18B20_DQ_GPIO_CLK_ENABLE(); /* 开启 DQ 引脚时钟 */

    gpio_init_struct.Pin = DS18B20_DQ_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD; /* 开漏输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(DS18B20_DQ_GPIO_PORT, &gpio_init_struct);
    /* DS18B20_DQ 引脚模式设置,开漏输出,上拉,这样就不用再设置 IO 方向了,开漏输出的时候(=1),也可以读取外部信号的高低电平 */
    ds18b20_reset();
    return ds18b20_check();
}
```

在 ds18b20 的初始化函数中，主要对用到的 GPIO 口进行初始化，同时在函数最后调用复位函数和自检函数，这两个函数在后面会解释到。

下面介绍一下在前面提及的几个信号类型：

```
/**
 * @brief      复位 DS18B20
 * @param      data: 要写入的数据
 * @retval     无
 */
static void ds18b20_reset(void)
{
    DS18B20_DQ_OUT(0); /* 拉低 DQ,复位 */
    delay_us(750); /* 拉低 750us */
    DS18B20_DQ_OUT(1); /* DQ=1,释放复位 */
}
```

```

    delay_us(15);          /* 延迟 15US */
}

/**
 * @brief      等待 DS18B20 的回应
 * @param      无
 * @retval     0, DS18B20 正常
 *            1, DS18B20 异常/不存在
 */
uint8_t ds18b20_check(void)
{
    uint8_t retry = 0;
    uint8_t rval = 0;

    while (DS18B20_DQ_IN && retry < 200)    /* 等待 DQ 变低, 等待 200us */
    {
        retry++;
        delay_us(1);
    }

    if (retry >= 200)
    {
        rval = 1;
    }
    else
    {
        retry = 0;

        while (!DS18B20_DQ_IN && retry < 240)    /* 等待 DQ 变高, 等待 240us */
        {
            retry++;
            delay_us(1);
        }
        if (retry >= 240) rval = 1;
    }
    return rval;
}

```

以上两个函数分别代表着前面所说的复位脉冲与应答信号，大家可以对比前面的时序图进行理解。由于复位脉冲比较简单，所以这里不做展开。现在看一下应答信号函数，函数主要是对于 DS18B20 传感器的回应信号进行检测，对此判断其是否存在。函数的实现也是依据时序图进行逻辑判断，例如当主机发送了复位信号之后，按照时序，DS18B20 会拉低数据线 60~240us，同时主机接收最小时间为 480us，我们就依据这两个硬性条件进行判断，首先需要设置一个时限等待 DS18B20 响应，后面也设置一个时限等待 DS18B20 释放数据线拉高，满足这两个条件即 DS18B20 成功响应。

下面接着看一下写函数：

```

/**
 * @brief      写一个字节到 DS18B20
 * @param      data: 要写入的字节
 * @retval     无
 */
static void ds18b20_write_byte(uint8_t data)
{
    uint8_t j;

    for (j = 1; j <= 8; j++)
    {
        if (data & 0x01)
        {
            DS18B20_DQ_OUT(0);    /* Write 1 */
            delay_us(2);
            DS18B20_DQ_OUT(1);
            delay_us(60);
        }
    }
}

```

```

    }
    else
    {
        DS18B20_DQ_OUT(0);    /* Write 0 */
        delay_us(60);
        DS18B20_DQ_OUT(1);
        delay_us(2);
    }
    data >>= 1;                /* 右移, 获取高一位数据 */
}
}

```

通过形参决定是写 1 还是写 0，按照前面对写时序的分析，我们可以很清晰知道写函数的逻辑处理。

有写函数肯定就有读函数，下面看一下读函数：

```

/**
 * @brief      从 DS18B20 读取一个位
 * @param      无
 * @retval     读取到的位值：0 / 1
 */
static uint8_t ds18b20_read_bit(void)
{
    uint8_t data = 0;
    DS18B20_DQ_OUT(0);
    delay_us(2);
    DS18B20_DQ_OUT(1);
    delay_us(12);
    if (DS18B20_DQ_IN)
    {
        data = 1;
    }
    delay_us(50);
    return data;
}

/**
 * @brief      从 DS18B20 读取一个字节
 * @param      无
 * @retval     读到的数据
 */
static uint8_t ds18b20_read_byte(void)
{
    uint8_t i, b, data = 0;

    for (i = 0; i < 8; i++)
    {
        b = ds18b20_read_bit(); /* DS18B20 先输出低位数据，高位数据后输出 */

        data |= b << i;         /* 填充 data 的每一位 */
    }

    return data;
}

```

在这里 ds18b20_read_bit 函数从 DS18B20 处读取 1 位数据，在前面已经对读时序也进行了详细的分析，所以这里也不展开解释了。

下面介绍读取温度函数，其定义如下：

```

/**
 * @brief      开始温度转换
 * @param      无
 * @retval     无
 */
static void ds18b20_start(void)
{
    ds18b20_reset();
}

```

```

ds18b20_check();
ds18b20_write_byte(0xcc); /* skip rom */
ds18b20_write_byte(0x44); /* convert */
}

/**
 * @brief      从 ds18b20 得到温度值 (精度: 0.1C)
 * @param      无
 * @retval     温度值 (-550~1250)
 * @note       返回的温度值放大了 10 倍.
 *             实际使用的时候,要除以 10 才是实际温度.
 */
short ds18b20_get_temperature(void)
{
    uint8_t flag = 1; /* 默认温度为正数 */
    uint8_t TL, TH;
    short temp;

    ds18b20_start(); /* ds1820 start convert */
    ds18b20_reset();
    ds18b20_check();
    ds18b20_write_byte(0xcc); /* skip rom */
    ds18b20_write_byte(0xbe); /* convert */
    TL = ds18b20_read_byte(); /* LSB */
    TH = ds18b20_read_byte(); /* MSB */

    if (TH > 7)
    { /* 温度为负, 查看 DS18B20 的温度表示法与计算机存储正负数据的原理一致:
       正数补码为寄存器存储的数据自身, 负数补码为寄存器存储值按位取反后+1
       所以我们直接取它实际的负数部分, 但负数的补码为取反后加一, 但考虑到低位可能+1 后
       有进位和代码冗余, 我们这里先暂时没有作+1 的处理, 这里需要留意 */
        TH = ~TH;
        TL = ~TL;
        flag = 0;
    }

    temp = TH; /* 获得高八位 */
    temp <= 8;
    temp += TL; /* 获得低八位 */

    /* 转换成实际温度 */
    if (flag == 0)
    { /* 将温度转换成负温度, 这里的+1 参考前面的说明 */
        temp = (double)(temp+1) * 0.625;
        temp = -temp;
    }
    else
    {
        temp = (double)temp * 0.625;
    }
    return temp;
}

```

在这里简单介绍一下上面用到的 RAM 指令:

跳过 ROM(0xCC), 该指令只适合总线只有一个节点, 它通过允许总线上的主机不提供 64 位 ROM 序列号而直接访问 RAM, 节省了操作时间。

温度转换(0x44), 启动 DS18B20 进行温度转换, 结果存入内部 RAM。

读暂存器(0xBE), 读暂存器 9 个字节内容, 该指令从 RAM 的第一个字节 (字节 0) 开始读取, 直到九个字节 (字节 8, CRC 值) 被读出为止。如果不需要读出所有字节的内容, 那么主机可以在任何时候发出复位信号以中止读操作。

2. main.c 代码

在 main.c 里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DS18B20 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (ds18b20_init()) /* DS18B20 初始化 */
    {
        lcd_show_string(30, 110, 200, 16, 16, "DS18B20 Error", RED);
        delay_ms(200);
        lcd_fill(30, 110, 239, 130 + 16, WHITE);
        delay_ms(200);
    }

    lcd_show_string(30, 110, 200, 16, 16, "DS18B20 OK", RED);
    lcd_show_string(30, 130, 200, 16, 16, "Temp: . C", BLUE);

    while (1)
    {
        if (t % 10 == 0) /* 每 100ms 读取一次 */
        {
            temperature = ds18b20_get_temperature();
            if (temperature < 0)
            {
                lcd_show_char(30 + 40, 130, '-', 16, 0, BLUE); /* 显示负号 */
                temperature = -temperature; /* 转为正数 */
            }
            else
            {
                lcd_show_char(30 + 40, 130, ' ', 16, 0, BLUE); /* 去掉负号 */
            }
            /* 显示正数部分 */
            lcd_show_num(30 + 40 + 8, 130, temperature / 10, 2, 16, BLUE);
            /* 显示小数部分 */
            lcd_show_num(30 + 40 + 32, 130, temperature % 10, 1, 16, BLUE);
        }

        delay_ms(10);
        t++;

        if (t == 20)
        {
            t = 0;
            LED0_TOGGLE(); /* LED0 闪烁 */
        }
    }
}
```

主函数代码比较简单，一系列硬件初始化后，在循环中调用 ds18b20_get_temperature 函数获取温度值，然后显示在 LCD 上。

42.4 下载验证

假定 DS18B20 传感器已经接上去正确的位置，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示当前的温度值的内容如图 42.4.1 所示：



图 42.4.1 程序运行效果图

该程序还可以读取并显示负温度值，具备零下温度条件可以测试一下。

第四十三章 DHT11 数字温湿度传感器

本章，我们将介绍数字温湿度传感器 DHT11 的使用，与前一章的温度传感器相比，该传感器不但能测温度，还能测湿度。我们将学习如何获取 DHT11 传感器的温湿度数据，并把数据显示在 LCD 上。

本章分为如下几个小节：

- 43.1 DHT11 及工作时序简介
- 43.2 硬件设计
- 43.3 程序设计
- 43.4 下载验证

43.1 DHT11 及工作时序简介

43.1.1 DHT11 简介

DHT11 是一款温湿度一体化的数字传感器。该传感器包括一个电容式测湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。通过单片机等微处理器简单的电路连接就能够实时的采集本地湿度和温度。DHT11 与单片机之间能采用简单的单总线进行通信，仅仅需要一个 I/O 口。传感器内部湿度和温度数据 40Bit 的数据一次性传给单片机，数据采用校验和方式进行校验，有效的保证数据传输的准确性。DHT11 功耗很低，5V 电源电压下，工作平均最大电流 0.5mA。

DHT11 的技术参数如下：

- 工作电压范围：3.3V ~ 5.5V
- 工作电流：平均 0.5mA
- 输出：单总线数字信号
- 测量范围：湿度 5 ~ 95%RH，温度 -20 ~ 60°C
- 精度：湿度 $\pm 5\%$ ，温度 $\pm 2^{\circ}\text{C}$
- 分辨率：湿度 1%，温度 0.1°C

DHT11 的管脚排列如图 43.1.1 所示：

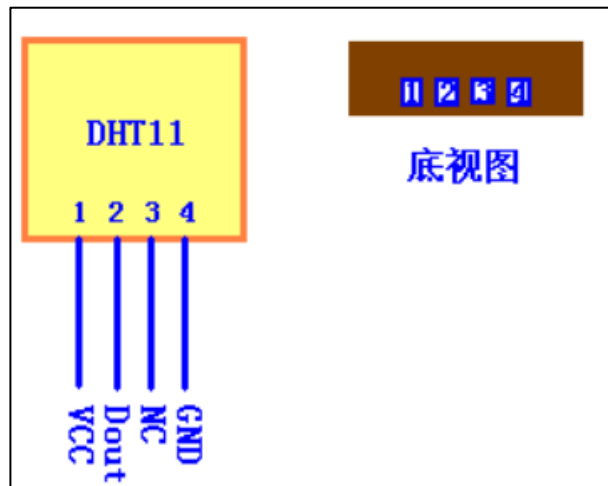


图 43.1.1.1 DHT11 管脚排列图

43.1.2 DHT11 工作时序简介

虽然 DHT11 与 DS18B20 类似，都是单总线访问，但是 DHT11 的访问，相对 DS18B20 来说简单很多。下面我们先来看看 DHT11 的数据结构。

DHT11 数字温湿度传感器采用单总线数据格式。即，单个数据引脚端口完成输入输出双向传输。其数据包由 5byte(40bit)组成。数据分小数部分和整数部分，一次完整的数据传输为 40bit，高位先处。DHT11 的数据格式为：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数部分+8bit 校验和。其中校验和数据为前面四个字节相加。

传感器数据输出的是未编码的二进制数据。数据（湿度、温度、整数、小数）之间应该分开处理。例如，某次从 DHT11 读到的数据如图 43.1.2.1 所示：

| byte4 | byte3 | byte2 | byte1 | byte0 |
|----------|----------|----------|----------|----------|
| 00101101 | 00000000 | 00011100 | 00000000 | 01001001 |
| 整数 | 小数 | 整数 | 小数 | 校验和 |
| 湿度 | | 温度 | | 校验和 |

图 43.1.2.1 某次读取到 DHT11 数据

由以上数据就可得到湿度和温度的值，计算方法：

湿度 = byte4 . byte3 = 45.0(%RH)

温度 = byte2 . byte1 = 28.0(°C)

校验 = byte4 + byte3 + byte2 + byte1 = 73 (= 湿度 + 温度) （校验正确）

可以看出，DHT11 的数据格式十分简单的，DHT11 和 MCU 的一次通信最大为 34ms 左右，建议主机连续读取时间间隔不要小于 2s。

下面，我们介绍一下 DHT11 的传输时序。DHT11 的数据发送流程如图 43.1.2.2 所示：

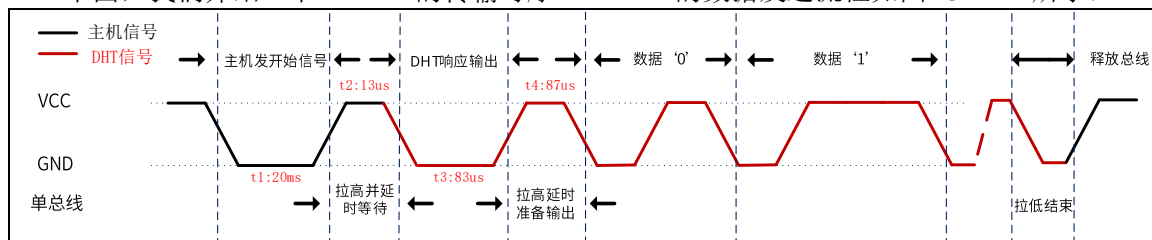


图 43.1.2.2 DHT11 数据发送流程图

首先主机发送开始信号，即：拉低数据线，保持 t1（至少 18ms）时间，然后拉高数据线 t2（10~35us）时间，然后读取 DHT11 的响应，正常的话，DHT11 会拉低数据线，保持 t3（78~88us）时间，作为响应信号，然后 DHT11 拉高数据线，保持 t4（80~92us）时间后，开始输出数据。

DHT11 输出数字 ‘0’ 时序如图 43.1.2.3 所示：

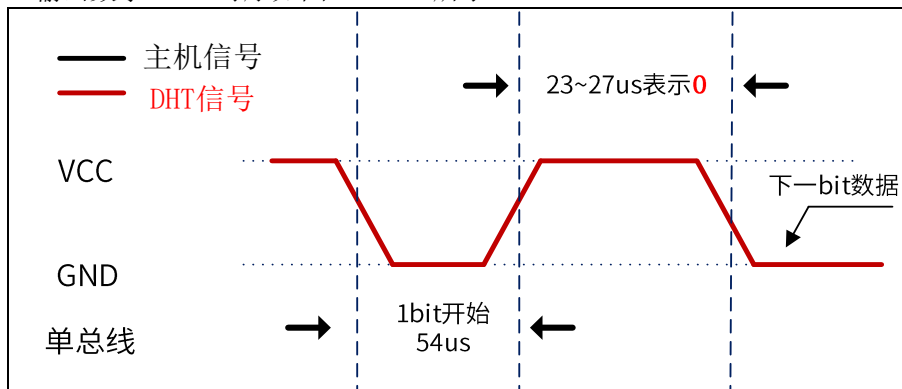


图 43.1.2.3 DHT11 数字 ‘0’ 时序图

DHT11 输出数字 ‘1’ 的时序如图 43.1.2.4 所示：

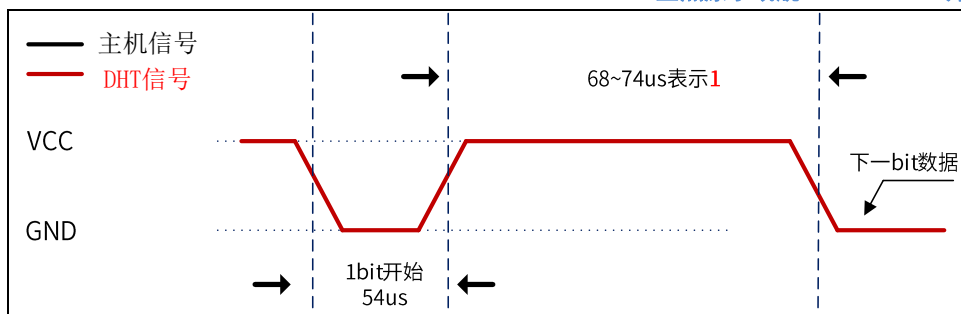


图 43.1.2.4 DHT11 输出数字 ‘1’ 时序图

DHT11 输出数字 ‘0’ 和 ‘1’ 时序，一开始都是 DHT11 拉低数据线 54us，后面拉高数据线保持的时间就不一样，数字 ‘0’ 就是 23~27us，而数字 ‘1’ 就是 68~74us。

通过以上了解，我们就可以通过 STM32F103 来实现对 DHT11 的读取了。DHT11 的介绍就到这里，更详细的介绍，请参考 DHT11 数据手册。

43.2 硬件设计

1. 例程功能

本实验开机的时候先检测是否有 DHT11 存在，如果没有，则提示错误。只有在检测到 DHT11 之后才开始读取温湿度值并显示在 LCD 上，如果发现了 DHT11，则程序每隔 100ms 左右读取一次数据，并把温湿度显示在 LCD 上。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) DHT11 温湿度传感器 - PG11
- 3) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

DHT11 接口与 STM32 的连接关系跟上一章节中 DS18B20 和 STM32 的关系是一样的，使用到的 GPIO 口是 PG11。这里原理图就不列出来了，可以翻看上一章节原理图。

DHT11 和 DS18B20 的接口是共用一个的，不过 DHT11 有 4 条腿，需要把 U6 的 4 个接口都用上，将 DHT11 传感器插入到这个上面就可以通过 STM32F1 来读取温湿度值了。连接示意图如图 43.2.1 所示：

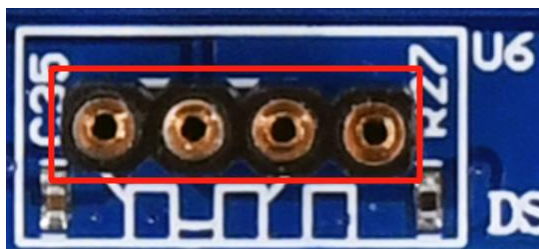


图 43.2.1 DHT11 连接示意图

这里要注意，将 DHT11 贴有字的一面朝内，而有很多孔的一面(网面)朝外，然后插入如图所示的四个孔内就可以了。

43.3 程序设计

DHT11 实验中使用的是单总线协议，用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下如何驱动 DHT11。

DHT11 配置步骤

1) 使能 DHT11 数据线对应的 GPIO 时钟。

本实验中 DHT11 的数据线引脚是 PG11，因此需要先使能 GPIOG 的时钟，代码如下：

```
HAL_RCC_GPIOG_CLK_ENABLE(); /* PG 口时钟使能 */
```

2) 设置对应 GPIO 工作模式（开漏输出）

本实验 GPIO 使用开漏输出模式，通过函数 HAL_GPIO_Init 设置实现。

3) 参考单总线协议，编写信号代码（复位脉冲、应答脉冲、读 0/1）

复位脉冲：拉低数据线，保持至少 18ms 时间，然后拉高数据线 10~35us 时间。

应答脉冲：DHT11 拉低数据线，保持 78~88us 时间。

读 0/1 信号：DHT11 拉低数据线延时 54us，然后拉高数据线延时一定时间，主机通过判断高电平时间得到 0 或者 1。

4) 编写 DHT11 的读函数

基于读 1bit 数据的基础上，编写 DHT11 读 1 字节函数。

5) 编写 DHT11 获取温度函数

参考 DHT11 典型温湿度读取过程，编写获取温湿度函数。

43.3.1 程序流程图

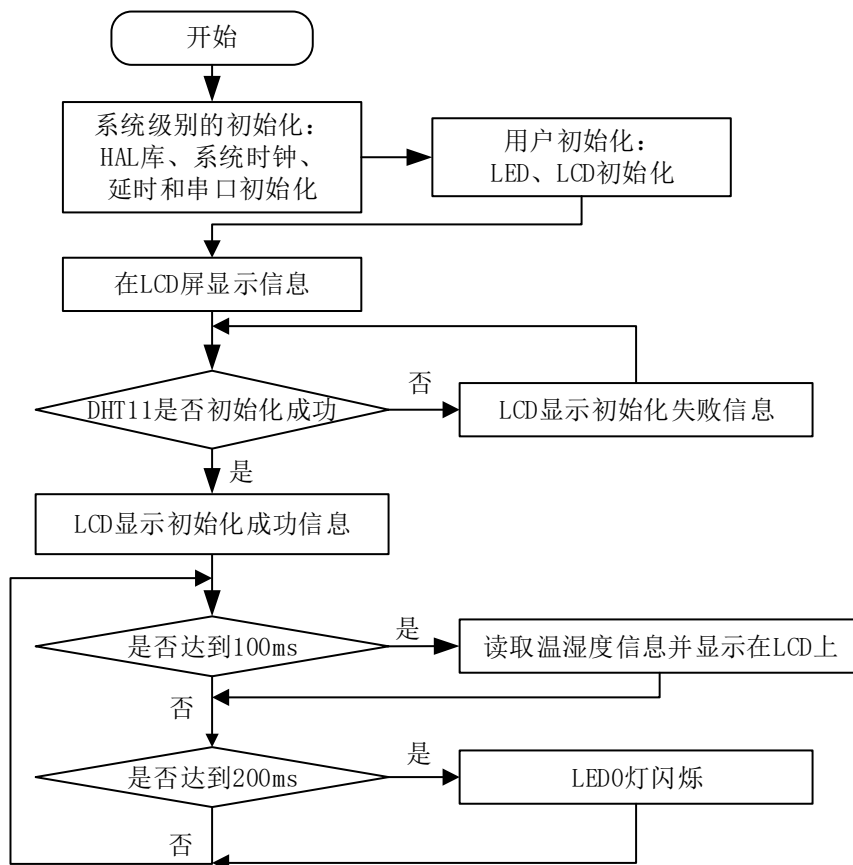


图 43.3.2.1 DHT11 实验程序流程图

43.3.2 程序解析

1.DHT11 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DHT11 驱动源码包括两个文件：dht11.c 和 dht11.h。

首先我们先看一下 dht11 头文件里面的内容，其定义如下：

```
/* DHT11 引脚 定义 */
#define DHT11_DQ_GPIO_PORT      GPIOG
#define DHT11_DQ_GPIO_PIN      GPIO_PIN_11
#define DHT11_DQ_GPIO_CLK_ENABLE()
    do{ __HAL_RCC_GPIOG_CLK_ENABLE(); }while(0)    /* PG 口时钟使能 */

/* IO 操作函数 */
#define DHT11_DQ_OUT(x)    do{ x ? \
    HAL_GPIO_WritePin(DHT11_DQ_GPIO_PORT,DHT11_DQ_GPIO_PIN, GPIO_PIN_SET):\
    HAL_GPIO_WritePin(DHT11_DQ_GPIO_PORT,DHT11_DQ_GPIO_PIN, GPIO_PIN_RESET); \
    }while(0)    /* 数据端口输出 */

/* 数据端口输入 */
#define DHT11_DQ_IN      HAL_GPIO_ReadPin(DHT11_DQ_GPIO_PORT, DHT11_DQ_GPIO_PIN)
```

对 DHT11 的相关引脚以及 IO 操作进行宏定义，方便程序中调用。

下面我们直接介绍 dht11.c 的程序，首先先介绍一下 DHT11 传感器的初始化函数，其定义如下：

```
/**
 * @brief      初始化 DHT11 的 IO 口 DQ 同时检测 DHT11 的存在
 * @param      无
 * @retval     0, 正常
 *             1, 不存在/不正常
 */
uint8_t dht11_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    DHT11_DQ_GPIO_CLK_ENABLE();    /* 开启 DQ 引脚时钟 */

    gpio_init_struct.Pin = DHT11_DQ_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD;    /* 开漏输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;    /* 高速 */
    HAL_GPIO_Init(DHT11_DQ_GPIO_PORT, &gpio_init_struct);    /* 初始化 DQ 引脚 */
    /* DHT11_DQ 引脚模式设置,开漏输出,上拉,这样就不用再设置 IO 方向了,开漏输出的时候(=1),
       也可以读取外部信号的高低电平 */

    dht11_reset();
    return dht11_check();
}
```

在 DHT11 的初始化函数中，主要对用到的 GPIO 口进行初始化，同时在函数最后调用复位函数和自检函数，这两个函数在后面会解释到。

下面介绍的是复位 DHT11 函数和等待 DHT11 的回应函数，它们的定义如下：

```
/**
 * @brief      复位 DHT11
 * @param      data: 要写入的数据
 * @retval     无
 */
static void dht11_reset(void)
{
    DHT11_DQ_OUT(0);    /* 拉低 DQ */
    delay_ms(20);    /* 拉低至少 18ms */
    DHT11_DQ_OUT(1);    /* DQ=1 */
    delay_us(30);    /* 主机拉高 10~35us */
}

/**
 * @brief      等待 DHT11 的回应
 * @param      无
 * @retval     0, DHT11 正常
 *             1, DHT11 异常/不存在
 */
```

```

*/
uint8_t dht11_check(void)
{
    uint8_t retry = 0;
    uint8_t rval = 0;

    while (DHT11_DQ_IN && retry < 100) /* DHT11 会拉低约 83us */
    {
        retry++;
        delay_us(1);
    }
    if (retry >= 100)
    {
        rval = 1;
    }
    else
    {
        retry = 0;
        while (!DHT11_DQ_IN && retry < 100) /* DHT11 拉低后会再次拉高 87us */
        {
            retry++;
            delay_us(1);
        }
        if (retry >= 100) rval = 1;
    }

    return rval;
}

```

以上两个函数分别代表着前面所说的复位脉冲与应答信号，大家可以对比前面的时序图进行理解。那么在上一章 DS18B20 的实验中，也对复位脉冲以及应答信号进行了详细的解释，大家也可以对比理解。

DHT11 与 DS18B20 有所不同，DHT11 是不需要写函数，只需要读函数即可，下面我们看一下读函数：

```

/**
 * @brief      从 DHT11 读取一个位
 * @param      无
 * @retval     读取到的位值：0 / 1
 */
uint8_t dht11_read_bit(void)
{
    uint8_t retry = 0;

    while (DHT11_DQ_IN && retry < 100) /* 等待变为低电平 */
    {
        retry++;
        delay_us(1);
    }

    retry = 0;

    while (!DHT11_DQ_IN && retry < 100) /* 等待变高电平 */
    {
        retry++;
        delay_us(1);
    }

    delay_us(40); /* 等待 40us */

    if (DHT11_DQ_IN) /* 根据引脚状态返回 bit */
    {
        return 1;
    }
    else

```

```

    {
        return 0;
    }
}

/**
 * @brief      从 DHT11 读取一个字节
 * @param      无
 * @retval     读到的数据
 */
static uint8_t dht11_read_byte(void)
{
    uint8_t i, data = 0;

    for (i = 0; i < 8; i++)        /* 循环读取 8 位数据 */
    {
        data <<= 1;                /* 高位数据先输出，先左移一位 */
        data |= dht11_read_bit();  /* 读取 1bit 数据 */
    }

    return data;
}

```

在这里 dht11_read_bit 函数从 DHT11 处读取 1 位数据，大家可以对照前面的读时序图进行分析，读数字 0 和 1 的不同，在于高电平的持续时间，所以这个作为判断的依据。dht11_read_byte 函数就是调用一字节读取函数进行实现。

下面介绍读取温湿度函数，其定义如下：

```

/**
 * @brief      从 DHT11 读取一次数据
 * @param      temp: 温度值 (范围:-20~60°)
 * @param      humi: 湿度值 (范围:5%~95%)
 * @retval     0, 正常.
 *            1, 失败
 */
uint8_t dht11_read_data(uint8_t *temp, uint8_t *humi)
{
    uint8_t buf[5];
    uint8_t i;

    dht11_reset();

    if (dht11_check() == 0)
    {
        for (i = 0; i < 5; i++)    /* 读取 40 位数据 */
        {
            buf[i] = dht11_read_byte();
        }

        if ((buf[0] + buf[1] + buf[2] + buf[3]) == buf[4])
        {
            *humi = buf[0];
            *temp = buf[2];
        }
    }
    else
    {
        return 1;
    }

    return 0;
}

```

读取温湿度函数也是根据时序图进行实现的，在发送复位信号以及应答信号产生后，即可以读取 5Byte 数据进行处理，校验成功即读取数据有效成功。

2. main.c 代码

在 main.c 里面编写如下代码：

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

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32F103", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DHT11 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (dht11_init()) /* DHT11 初始化 */
    {
        lcd_show_string(30, 110, 200, 16, 16, "DHT11 Error", RED);
        delay_ms(200);
        lcd_fill(30, 110, 239, 130 + 16, WHITE);
        delay_ms(200);
    }

    lcd_show_string(30, 110, 200, 16, 16, "DHT11 OK", RED);
    lcd_show_string(30, 130, 200, 16, 16, "Temp: C", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "Humi: %", BLUE);

    while (1)
    {
        if (t % 10 == 0) /* 每 100ms 读取一次 */
        {
            dht11_read_data(&temperature, &humidity); /* 读取温湿度值 */
            lcd_show_num(30 + 40, 130, temperature, 2, 16, BLUE); /* 显示温度 */
            lcd_show_num(30 + 40, 150, humidity, 2, 16, BLUE); /* 显示湿度 */
        }

        delay_ms(10);

        t++;

        if (t == 20)
        {
            t = 0;
            LED0_TOGGLE(); /* LED0 闪烁 */
        }
    }
}
```

主函数代码比较简单，一系列硬件初始化后，如果 DHT11 初始化成功，那么在循环中调用 dht11_get_temperature 函数获取温湿度值，每隔 100ms 读取数据并显示在 LCD 上。

43.4 下载验证

假定 DHT11 传感器已经接上去正确的位置，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示当前的温度值的内容如图 43.4.1 所示：

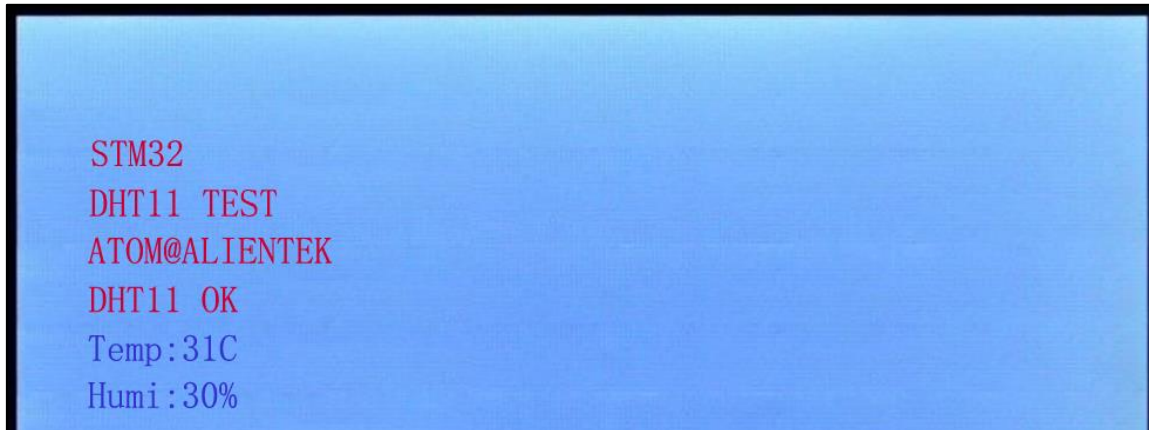


图 43.4.1 程序运行效果图

至此，本章实验结束。大家可以将本章通过 DHT11 读取到的温度值，和前一章的通过 DS18B20 读取到的温度值对比一下，看看哪个更准确？

第四十四章 无线通信实验

本章，我们将介绍如何使用 2.4G 无线模块 NRF24L01 实现无线通信。将使用两块 STM32 开发板，一块用于发送，一块用于接收，从而实现无线数据传输，并把数据显示在 LCD 上。本章分为如下几个小节：

- 44.1 NRF24L01 无线模块介绍
- 44.2 硬件设计
- 44.3 程序设计
- 44.4 下载验证

44.1 NRF24L01 无线模块介绍

44.1.1 NRF24L01 简介

NRF24L01 无线模块，采用的芯片是 NRF24L01+。该芯片是由 NORDIC 公司生产，并且集成 NORDIC 自家的 Enhance ShortBurst 协议，主要特点如下：

- 1) 2.4G 全球开放的 ISM 频段，免许可证使用
- 2) 最高工作速率 2Mbps，高效的 GFSK 调制，抗干扰能力强
- 3) 126 个可选的频道，满足多点通信和调频通信的需要
- 4) 6 个数据通道可支持点对多点的通信地址控制
- 5) 低工作电压（1.9~3.6V）
- 6) 硬件 CRC 和自动处理字头
- 7) 可设置自动应答，确保数据可靠传输

由于高速信号是由芯片内部的射频协议处理后进行无线高速通信，对 MCU 的时钟频率要求不高，只需要对 NRF24L01 某些寄存器进行配置即可。芯片与外部 MCU 是通过 SPI 通信接口进行数据通信，并且最大的 SPI 速度可达 10MHz。

这个芯片是 NRF24L01 的升级版。相比 NRF24L01，升级版支持 250k, 1M, 2M 三种传输速率；支持更多种功率配置，根据不同应用有效节省功耗；稳定性及可靠性更高。

该模块的外形和引脚图如图 44.1.1.1 所示：

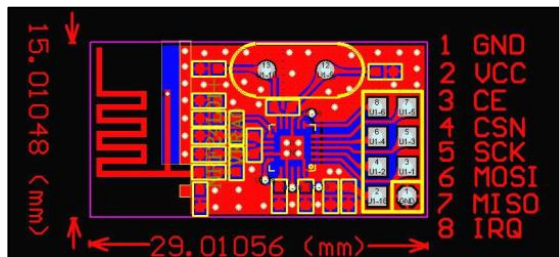


图 44.1.1.1 NRF24L01 无线模块外形和引脚图

模块 VCC 脚的电压范围为 1.9~3.6V，建议不要超过 3.6V，否则可能烧坏模块，一般用 3.3V 电压比较合适。除了 VCC 和 GND 脚，其他引脚都可以和 5V 单片机的 IO 口直连，正是由于其兼容 5V 单片机的 IO，所以使用上具有很大优势。

具体引脚介绍如表 44.1.1.1 所示。

| 模块引脚 | GND | VCC | CE | CSN | SCK | MOSI | MISO | IRQ |
|------|-----|----------|-------|-----|-----|------|------|-----|
| 功能说明 | 地线 | 3.3V 电源线 | 模式控制线 | 片选 | 时钟 | 数据输出 | 数据输入 | 中断 |

表 44.1.1.1 引脚介绍表

引脚部分主要分为电源相关的 VCC 和 GND，SPI 通信接口相关的 CSN/SCK/MOSI/MISO，模式选择相关的 CE，中断相关的 IRQ。CE 引脚会与 CONFIG 寄存器共同控制 NRF24L01 进入某个工作模式。IRQ 引脚会在寄存器的配置下生效，当收到数据、成功发送数据或达到最大重发次数时，IRQ 引脚会变为低电平。

由于在前面的 SPI 实验章节已经对 SPI 通信协议进行了详细的讲解，这里就不做展开。后面代码部分我们再进行分析 NRF24L01 使用到 SPI 的配置。

NRF24L01 的 Enhance ShockBurst™ 模式具体表现在自动应答和重发机制，发送端要求接收端在接收到数据后要有应答信号，便于发送端检测有无数据丢失，一旦有数据丢失，则通过重发功能将丢失的数据恢复，这个过程无需 MCU。Enhance ShockBurst™ 模式可以通过 EN_AA 寄存器进行配置。

接下来看一下 Enhanced ShockBurst™ 模式下 NRF24L01 通信图，如图 44.1.1.2 所示：

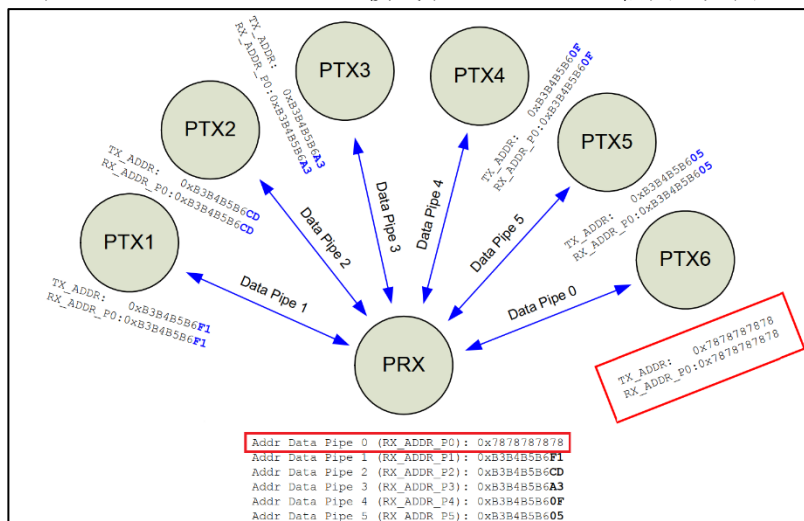


图 44.1.1.2 NRF24L01 通信图

这里我们抽离 PTX6 和 PRX 出来，分析一下通信过程。

PTX6 作为发送端，它就需要设置发送地址，可以看到 TX_ADDR 为 0x78787878，PRX 作为接收端，它使能接收通道 0 并设置接收通道 0 接收地址 0x78787878。通信时，发送端发送数据→接收端接收到数据并记录 TX 地址→接收端以 TX 地址为目的地址发送应答信号→发送端会以通道 0 接收应答信号。

NRF24L01 规定：发送端中的数据通道 0 是用来接收接收端发送的应答信号，所以数据通道 0 的接收地址要与发送地址要相同才能确保收到正确的应答信号，这里十分重要，必须要在相关寄存器中配置正确。

44.1.2 NRF24L01 工作模式介绍

NRF24L01 作为无线通信模块，功耗问题十分重要，有数据发送与空闲状态下能耗肯定是需要调整，所以设计者给芯片设计了多种工作模块，如表 44.1.2.1 所示：

| 24L01 模式 | PWR_UP 位 | PRIM_RX 位 | CE 引脚电平 | FIFO 寄存器状态 |
|----------|----------|-----------|--------------|-----------------|
| 接收模式 | 1 | 1 | 1 | - |
| 发送模式 | 1 | 0 | 1 | 发送所有 TX FIFO 数据 |
| 发送模式 | 1 | 0 | 1(至少 10us)→0 | 发送一级 TX FIFO 数据 |
| 待机模式 II | 1 | 0 | 1 | TX FIFO 为空 |
| 待机模式 I | 1 | - | 0 | 无数据包传输 |
| 掉电模式 | 0 | - | - | - |

表 44.1.2.1 NRF24L01 工作模式表

NRF24L01 工作模式是由 CE 引脚和 CONFIG 寄存器的 PWR_UP 位和 PRIM_RX 位共同控制。CE 引脚在前面也说到是模式控制线，而 PWR_UP 位是上电位，PRIM_RX 位可以理解为配置身份位(TX or RX)。可以看到发送模式有两种，待机模式也有两种，功耗上各不相同，没有标红的发送模式和待机模式 I 是官方推荐使用，更加节能，但是本实验用到的模式就是上表中标红部分，因为标红的模式使用起来更加方便。单看发送模式，使用官方推荐的发送模式，

你要发送三级 TX_FIFO 数据需要产生三个边沿信号（CE 从高电平变为低电平）。而我们使用的发送模式，从 CE 引脚的操作上看，只需要拉高，就可以把所有 TX_FIFO 里的数据发送完成。

NRF24L01 的发送和接收都有三级 FIFO，每一级 FIFO 就有 32 个字节。发送和接收都是对 FIFO 进行操作，并且最大操作的数据量就是一级 FIFO 即 32 字节。发送时，只需要把数据存进 TX_FIFO 并按照发送模式下的操作（参考 NRF24L01 工作模式表中的发送模式）即可让 NRF24L01 启动发射，这个发射过程就包括：无线系统上电，启动内部 16MHz 时钟，无线发送数据打包，高速发送数据。接收时，也是通过读取 RX_FIFO 里的内容。

44.1.3 NRF24L01 寄存器

在这里简单介绍一下本实验用到的 NRF24L01 比较重要的寄存器。

● 配置寄存器（CONFIG）

寄存器地址 0x01，复位值为 0x80，用来配置 NRF24L01 工作状态以及中断相关，描述如图 44.1.2.1 所示：

| 参数 | 位 | 描述 |
|-------------|---|---|
| Reserved | 7 | 保留位 |
| MASK_RX_DR | 6 | 可屏蔽中断 RX_DR（1：IRQ 引脚不显示 RX_DR 中断） （0：RX_DR 中断时，IRQ 输出低电平） |
| MASK_TX_DS | 5 | 可屏蔽中断 TX_DS（1：IRQ 引脚不显示 TX_DS 中断） （0：TX_DS 中断时，IRQ 输出低电平） |
| MASK_MAX_RT | 4 | 可屏蔽中断 MAX_RT（1：IRQ 引脚不显示 MAX_RT 中断） （0：MAX_RT 中断时，IRQ 输出低电平） |
| EN_CRC | 3 | CRC 使能（如果 EN_AA 中任意一位为高则 EN_CRC 强迫为高） |
| CRCO | 2 | CRC 模式（1：16 位 CRC 校验 0：8 位 CRC 校验） |
| PWR_UP | 1 | 上电/掉电模式设置位（1：上电 0：掉电） |
| PRIM_RX | 0 | 接收/发送模式设置位（1：接收模式 0：发送模式） |

图 44.1.2.1 配置寄存器图

需要配置成发送模式，可以把该寄存器赋值为 0x0E，如果配置成接收模式，可以把该寄存器赋值为 0x0F。无论是发送模式还是接收模式，都使能 16 位 CRC 以及使能接收中断、发送中断和最大重发次数中断，这里发送端和接收端配置需要一致。

● 自动应答功能寄存器（EN_AA）

寄存器地址 0x01，复位值为 0x3F，用来设置通道 0~5 的自动应答功能，描述如图 44.1.2.3 所示：

| 参数 | 位 | 描述 |
|----------|-----|---------------|
| Reserved | 7:6 | 保留位 |
| ENAA_P5 | 5 | 数据通道 5，自动应答允许 |
| ENAA_P4 | 4 | 数据通道 4，自动应答允许 |
| ENAA_P3 | 3 | 数据通道 3，自动应答允许 |
| ENAA_P2 | 2 | 数据通道 2，自动应答允许 |
| ENAA_P1 | 1 | 数据通道 1，自动应答允许 |
| ENAA_P0 | 0 | 数据通道 0，自动应答允许 |

图 44.1.2.3 自动应答功能寄存器图

本实验，接收端是以数据通道 0 作为接收通道，并且前面也提及 Enhanced ShockBurst™ 模式的自动应答流程，接收端接收到数据后，需要回复应答信号，通过该寄存器 ENAA_P0 置 1 即可实现。另外，使能自动应答也相当于配置成 Enhanced 模式，所以发送端也需要进行自动应答允许。

● 接收地址允许寄存器（EN_RXADDR）

寄存器地址 0x02，复位值为 0x03，用于使能接收通道 0~5，描述如图 44.1.2.4 所示：

| 参数 | 位 | 描述 |
|----------|-----|-------------|
| Reserved | 7:6 | 保留位 |
| ERX_P5 | 5 | 数据接收通道 5 使能 |
| ERX_P4 | 4 | 数据接收通道 4 使能 |
| ERX_P3 | 3 | 数据接收通道 3 使能 |
| ERX_P2 | 2 | 数据接收通道 2 使能 |
| ERX_P1 | 1 | 数据接收通道 1 使能 |
| ERX_P0 | 0 | 数据接收通道 0 使能 |

图 44.1.2.4 接收地址允许寄存器图

前面也说到接收端使用的是通道 0 进行接收数据，所以 ERX_P0 需要置 1 处理。同样的，发送端也需要使能数据通道 0 来接收应答信号。

● 地址宽度设置寄存器 (SETUP_AW)

寄存器地址 0x03，复位值为 0x03，对接收/发送地址宽度设置位，描述如图 44.1.2.5 所示：

| 参数 | 位 | 描述 |
|----------|-----|--|
| Reserved | 7:2 | 保留位 |
| AW | 1:0 | RX/TX 地址字段宽度 '00' 非法 '01' 3 字节 '10' 4 字节 '11' 5 字节 |

图 44.1.2.5 地址宽度设置寄存器图

本实验中，无论是发送地址还是接收地址都是使用 5 字节，也就是默认设置便是使用 5 字节宽度的地址。

● 自动重发配置寄存器 (SETUP_RETR)

寄存器地址 0x04，复位值为 0x00，对发送端的自动重发数值和延时进行设置，描述如图 44.1.2.6 所示：

| 参数 | 位 | 描述 |
|-----|-----|---|
| ADR | 7:4 | 自动重发延时： 0000~1111→ $86\text{ us} + 250 * (\text{ARD} + 1)\text{ us}$ |
| ARC | 3:0 | 自动重发计数 0000~1111→自动重发次数。0 代表禁止 |

图 44.1.2.6 自动重发配置寄存器图

本实验中，直接对该寄存器写入 0x1A，即自动重发间隔时间为 586us，最大自动重发次数为 10 次。在使能了 MAX_RT 中断时，连续重发 10 次还是发送失败的时候，IRQ 中断引脚就会拉低。

● 射频频率设置寄存器 (RF_CH)

寄存器地址 0x05，复位值为 0x05，对 NRF24L01 的频段进行设置，描述如图 44.1.2.7 所示：

| 参数 | 位 | 描述 |
|----------|-----|------------------------------------|
| Reserved | 7 | 保留位 |
| RF_CH | 6:0 | 0~125，设置 NRF24L01 的射频频率，接收端和发送端需一致 |

图 44.1.2.7 射频频率设置寄存器图

频率计算公式： $2400 + \text{RF_CH}$ (MHz)

本实验中，直接对该寄存器写入 40 即射频频率为 2440MHz。通信双方该寄存器必须配置一样才能通信成功。

● 发射参数设置寄存器 (RF_SETUP)

寄存器地址 0x06，复位值为 0x0E，对 NRF24L01 的发射功率、无线速率进行设置，描述如图 44.1.2.8 所示：

| 参数 | 位 | 描述 |
|-----------|---|---------------|
| CONT_WAVE | 7 | 高电平时，可使载波连续传输 |

| | | |
|------------|-----|--|
| Reserved | 6 | 只允许写 ‘0’ |
| RF_DR_LOW | 5 | 设置射频数据速率 250kbps（结合 RF_DR_HIGH 查看） |
| PLL_LOCK | 4 | PLL_LOCK 允许，仅用于测试模式 |
| RF_DR_HIGH | 3 | 与 RF_DR_LOW 决定传输速率[RF_DR_LOW, RF_DR_HIGH] ‘00’: 1Mbps ‘01’: 2Mbps ‘10’: 250kbps ‘11’: 保留 |
| RF_PWR | 1:0 | 设置射频输出功率 ‘00’: -18dBm ‘01’: -12dBm ‘10’: -6dBm ‘11’: 0dBm |
| Obsolete | 0 | 不用关心 |

图 44.1.2.8 发射参数设置寄存器图

本实验中，直接对该寄存器写入 0x0F 即射频输出功率为 0dBm 增益，传输速率为 2MHz。发送端和接收端该寄存器的配置需一样。功率越小耗电越少，同等条件下，传输距离越小，这里我们设置射频部分功耗为最大，当然大家可以根据实际应用而选择对应的功率配置。

● 状态寄存器 (STATUS)

地址 0x07，复位值为 0x0E，反应 NRF24L01 当前工作状态，描述如图 44.1.2.2 所示：

| 参数 | 位 | 描述 |
|----------|-----|---|
| Reserved | 7 | 保留位 |
| RX_DR | 6 | 接收数据标记，收到数据后置 1。写 1 清除中断 |
| TX_DS | 5 | 数据发送完成标记。工作在自动应答模式，必须收到 ACK 才会置 1。写 1 清除中断。 |
| MAX_RT | 4 | 达到最大重发次数标记。写 1 清除中断（如果 MAX_RX 中断产生则必须清除后系统才能进行通信） |
| RX_P_NO | 3:1 | 接收数据通道： 000~101 数据通道号 110 未使用 111 RX_FIFO 为空 |
| TX_FULL | 0 | TX_FIFO 寄存器满标记（1：满 0：未满） |

图 44.1.2.2 状态寄存器图

该寄存器作为查询作用，作为发送端，发送完数据后，可以查询一下 TX_DS 位状态便知是否成功发送数据，发送数据异常时，也可以通过查询 MAX_RT 位状态获知是否达到最大重发次数。作为接收端，就可以通过查询 RX_OK 位状态获知是否接收到数据。我们查询相关位后都需要将该位置 1 清除中断。

此外，我们还用到设置接收通道 0 地址寄存器 RX_ADDR_P0(0x0A)和发送地址设置寄存器 TX_ADDR(0x10)以及接收通道 0 有效数据看度设置寄存器 RX_PW_P0(0x11)，由于这三个寄存器比较简单，所以这里就不列出来了。

44.2 硬件设计

1. 例程功能

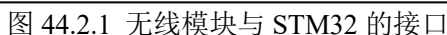
开机的时候先检测 NRF24L01 模块是否存在，在检测到 NRF24L01 模块之后，根据 KEY0 和 KEY1 的设置来决定模块的工作模式。在设定好工作模式之后，就会不停的发送/接收数据，同时在 LCD 上面显示相关信息。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4

- 3) 2.4G 无线模块 NRF24L01 模块
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 5) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 6) SPI2(连接在 PB13/PB14/PB15)

NRF24L01 模块与 STM32 的连接关系, 如下图所示:



由于无线通信实验是双向的，所以至少要有两个模块同时能工作，这里我们使用 2 套开发板来向大家演示。

NRF24L01 配置步驟

注意：该函数会调用：HAL_SPI_MspInit 函数来完成对 SPI 底层的初始化，包括：SPI 及 I/O 时钟使能、GPIO 模式设置等。

```
_HAL_RCC_SPI2_CLK_ENABLE();
_HAL_RCC_GPIOB_CLK_ENABLE();
_HAL_RCC_GPIOG_CLK_ENABLE();
```

通过 HAL_SPI_ENABLE 函数使能 SPI，便可进行数据传输。

也可以通过 HAL_SPI_TransmitReceive 函数进行发送与接收操作。

基于 SPI 的读写函数的基础上，编写 NRF24L01 的读写函数

通过查看寄存器，编写配置 NRF24L01 接收和发送模式的函数。

44.3.1 程序流程图

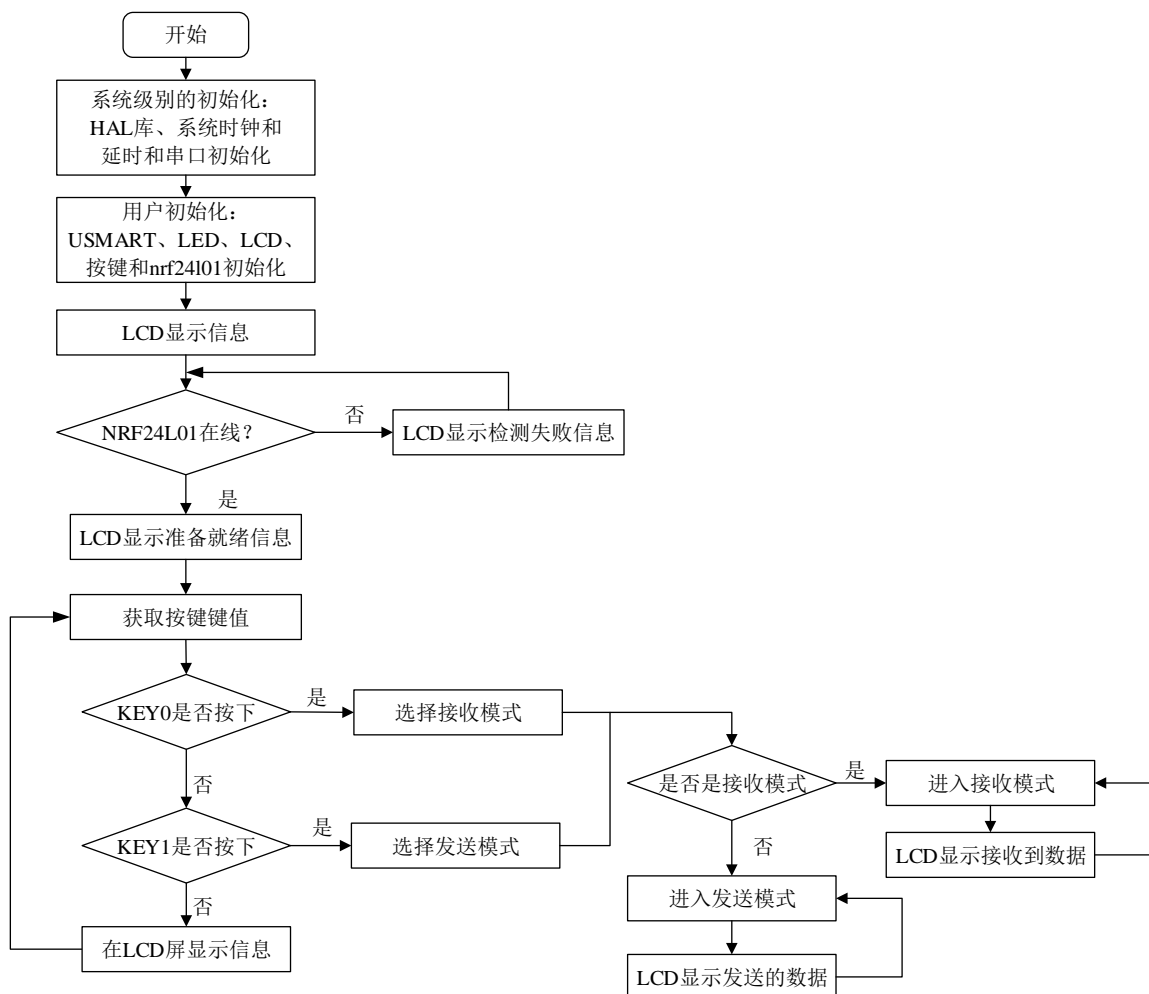


图 44.3.1.1 无线通信实验程序流程图

44.3.2 程序解析

本实验中，使用到的 SPI 配置与 SPI 实验章节差异不大，所以这里不作展开了，大家可以先回顾一下 SPI 实验的内容再来学习。

1. NRF24L01 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。NRF24L01 驱动源码包括两个文件：nrf24l01.c 和 nrf24l01.h。

我们要在 spi.c 文件中封装好的函数的基础上进行调用，实现 nrf24l01 的发送与接收。下面先看一下 nrf24l01.h 文件中定义的信息，其代码如下：

```

/* NRF24L01 操作引脚 定义(不包含 SPI_SCK/MISO/MISO 等三根线) */
#define NRF24L01_CE_GPIO_PORT      GPIOG
#define NRF24L01_CE_GPIO_PIN      GPIO_PIN_8
#define NRF24L01_CE_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0) /* PG 口时钟使能 */

#define NRF24L01_CSN_GPIO_PORT      GPIOG
#define NRF24L01_CSN_GPIO_PIN      GPIO_PIN_7
#define NRF24L01_CSN_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0) /* PE 口时钟使能 */
  
```



```
#define NRF24L01_IRQ_GPIO_PORT      GPIOG
#define NRF24L01_IRQ_GPIO_PIN      GPIO_PIN_6
#define NRF24L01_IRQ_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0) /* PG 口时钟使能 */

/* 24L01 操作线 */
#define NRF24L01_CE(x) do{ x ? \
    HAL_GPIO_WritePin(NRF24L01_CE_GPIO_PORT,NRF24L01_CE_GPIO_PIN,GPIO_PIN_SET):\
    HAL_GPIO_WritePin(NRF24L01_CE_GPIO_PORT,NRF24L01_CE_GPIO_PIN,GPIO_PIN_RESET);\
}while(0) /* 24L01 模式选择信号 */

#define NRF24L01_CSN(x) do{ x ? \
    HAL_GPIO_WritePin(NRF24L01_CSN_GPIO_PORT,NRF24L01_CSN_GPIO_PIN,GPIO_PIN_SET):\
    HAL_GPIO_WritePin(NRF24L01_CSN_GPIO_PORT,\
        NRF24L01_CSN_GPIO_PIN,GPIO_PIN_RESET);\
}while(0) /* 24L01 片选信号 */

#define NRF24L01_IRQ      HAL_GPIO_ReadPin(NRF24L01_IRQ_GPIO_PORT, \
        NRF24L01_IRQ_GPIO_PIN) /* IRQ 主机数据输入 */
```

以上除了有 NRF24L01 的引脚定义及引脚操作函数，此外还有一些 NRF24L01 寄存器操作命令以及其寄存器地址，由于篇幅太大，所以这里就不列出来了，大家可以去看一下工程文件。

下面看一下 NRF24L01 的初始化函数，其定义如下：

```
/**
 * @brief      初始化 24L01 的 IO 口
 * @note      将 SPI2 模式改成 SCK 空闲低电平, 及 SPI 模式 0
 * @param      无
 * @retval     无
 */
void nrf24l01_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    NRF24L01_CE_GPIO_CLK_ENABLE(); /* CE 脚时钟使能 */
    NRF24L01_CSN_GPIO_CLK_ENABLE(); /* CSN 脚时钟使能 */
    NRF24L01_IRQ_GPIO_CLK_ENABLE(); /* IRQ 脚时钟使能 */

    gpio_init_struct.Pin = NRF24L01_CE_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP; /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(NRF24L01_CE_GPIO_PORT, &gpio_init_struct); /* 初始化 CE 引脚 */

    gpio_init_struct.Pin = NRF24L01_CSN_GPIO_PIN;
    HAL_GPIO_Init(NRF24L01_CSN_GPIO_PORT, &gpio_init_struct); /* 初始化 CS 引脚 */

    gpio_init_struct.Pin = NRF24L01_IRQ_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_INPUT; /* 输入 */
    gpio_init_struct.Pull = GPIO_PULLUP; /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(NRF24L01_IRQ_GPIO_PORT, &gpio_init_struct); /* 初始化 CE 引脚 */

    spi2_init(); /* 初始化 SPI2 */
    nrf24l01_spi_init(); /* 针对 NRF 的特点修改 SPI 的设置 */
    NRF24L01_CE(0); /* 使能 24L01 */
    NRF24L01_CSN(1); /* SPI 片选取消 */
}
/**
 * @brief      针对 NRF24L01 修改 SPI2 驱动
 * @param      无
 * @retval     无
 */
```

```
void nrf24l01_spi_init(void)
{
    __HAL_SPI_DISABLE(&g_spi2_handler);    /* 先关闭 SPI2 */
    /* 串行同步时钟的空闲状态为低电平 */
    g_spi2_handler.Init.CLKPolarity = SPI_POLARITY_LOW;
    /* 串行同步时钟的第 1 个跳变沿（上升或下降）数据被采样 */
    g_spi2_handler.Init.CLKPhase = SPI_PHASE_1EDGE;
    HAL_SPI_Init(&g_spi2_handler);
    __HAL_SPI_ENABLE(&g_spi2_handler);    /* 使能 SPI2 */
}
```

在初始化函数中，我们主要对该模块用到的管脚进行配置以及从初始化工作以及需要调用 spi.c 文件中的 spi_init 函数对 SPI 的引脚进行初始化。现在让我们看一下 NRF24L01 的工作时序图见下图 44.3.2.1 所示。

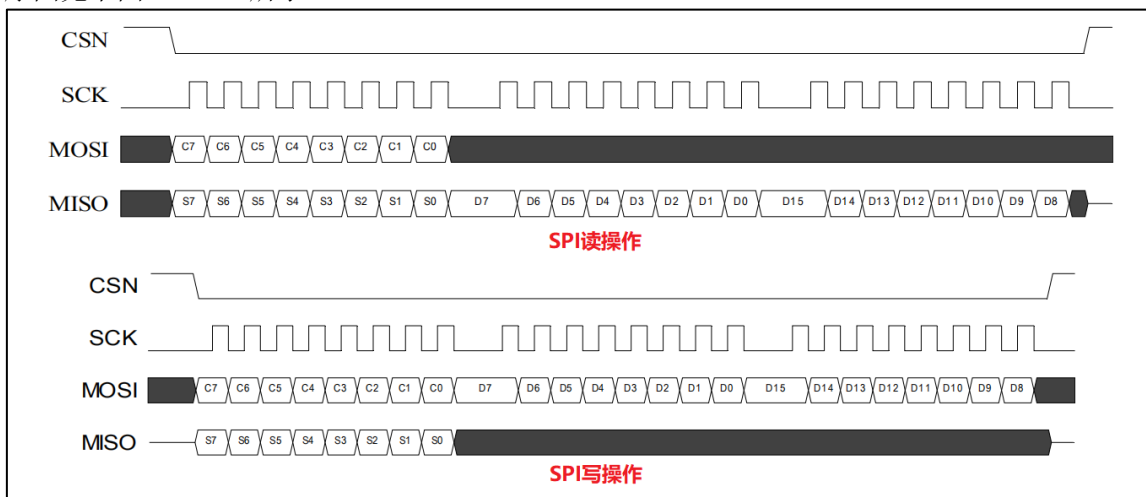


图 44.3.2.1 NRF24L01 的工作时序图

大家可以比对一下前面章节的 SPI 工作时序图，符合工作模式 1 的时序，即在奇数边沿上升沿进行数据的采集。所以我们调用 nrf24l01_spi_init 函数针对 NRF 的特点修改 SPI 的设置。该函数就是将 SPI 的工作模式配置成串行同步时钟空闲状态为低电平，在奇数边沿数据被采集，也就是前面 SPI 实验章节中 SPI 的工作模式 0。现在看看工作时序图的某些标号意义，Cn 代表指令位，Sn 代表状态寄存器位，Dn 代表数据位。

下面介绍一下 NRF24L01 的读写函数，其代码如下：

```
/**
 * @brief      NRF24L01 写寄存器
 * @param      reg    : 寄存器地址
 * @param      value  : 写入寄存器的值
 * @retval     状态寄存器值
 */
static uint8_t nrf24l01_write_reg(uint8_t reg, uint8_t value)
{
    uint8_t status;
    NRF24L01_CSN(0);    /* 使能 SPI 传输 */
    status = spi2_read_write_byte(reg);    /* 发送寄存器号 */
    spi2_read_write_byte(value);    /* 写入寄存器的值 */
    NRF24L01_CSN(1);    /* 禁止 SPI 传输 */
    return status;    /* 返回状态值 */
}

/**
 * @brief      NRF24L01 读寄存器
 * @param      reg    : 寄存器地址
 * @retval     读取到的寄存器值;
 */
static uint8_t nrf24l01_read_reg(uint8_t reg)
```

```

{
    uint8_t reg_val;
    NRF24L01_CSN(0);          /* 使能 SPI 传输 */
    spi2_read_write_byte(reg); /* 发送寄存器号 */
    reg_val = spi2_read_write_byte(0xFF); /* 读取寄存器内容 */
    NRF24L01_CSN(1);          /* 禁止 SPI 传输 */
    return reg_val;            /* 返回状态值 */
}

/**
 * @brief      在指定位置读出指定长度的数据
 * @param      reg    : 寄存器地址
 * @param      pbuf   : 数据指针
 * @param      len    : 数据长度
 * @retval     状态寄存器值
 */
static uint8_t nrf24l01_read_buf(uint8_t reg, uint8_t *pbuf, uint8_t len)
{
    uint8_t status, i;
    NRF24L01_CSN(0);          /* 使能 SPI 传输 */
    status = spi2_read_write_byte(reg); /* 发送寄存器值(位置),并读取状态值 */

    for (i = 0; i < len; i++)
    {
        pbuf[i] = spi2_read_write_byte(0xFF); /* 读出数据 */
    }

    NRF24L01_CSN(1);          /* 关闭 SPI 传输 */
    return status;            /* 返回读到的状态值 */
}

/**
 * @brief      在指定位置写指定长度的数据
 * @param      reg    : 寄存器地址
 * @param      pbuf   : 数据指针
 * @param      len    : 数据长度
 * @retval     状态寄存器值
 */
static uint8_t nrf24l01_write_buf(uint8_t reg, uint8_t *pbuf, uint8_t len)
{
    uint8_t status, i;
    NRF24L01_CSN(0);          /* 使能 SPI 传输 */
    status = spi2_read_write_byte(reg); /* 发送寄存器值(位置),并读取状态值 */

    for (i = 0; i < len; i++)
    {
        spi2_read_write_byte(*pbuf++); /* 写入数据 */
    }

    NRF24L01_CSN(1);          /* 关闭 SPI 传输 */
    return status;            /* 返回读到的状态值 */
}

```

以上是 NRF24L01 的写寄存器函数和读寄存器函数，以及扩展的函数：在指定位置写入指定长度的数据函数和指定位置读取指定长度的数据函数。先讲一下 NRF24L01 读写寄存器函数实现的具体过程：

先拉低片选线→发送寄存器号→发送数据/接收数据→拉高片选线。

这里提及一下 SPI 的相关知识：SPI 是通过移位寄存器进行数据传输，所以发一字节数据就会收到一个字节数据。那么发数据就可以直接发送数据，接收数据只需要发送 0xFF，寄存器会返回要读取的数据。

在指定位置写入指定长度的数据函数和在指定位置读取指定长度的数据函数的实现方式也

是通过调用 SPI 的读写一字节函数实现，这里跟写寄存器和读寄存器函数的实现差不多，这里就不做展开了。不懂的话，也可以回顾一下 SPI 实验章节。

下面看一下这两种模式的初始化过程：

Rx 模式初始化过程：

- 1) 写 Rx 节点的地址
- 2) 使能通道 x 自动应答
- 3) 使能通道 x 接收地址
- 4) 设置通信频率
- 5) 选择通道 x 的有效数据宽度
- 6) 配置发射参数（发射功率、无线速率）
- 7) 配置 NRF24L01 的基本参数以及工作模式

其代码如下：

```
/**
 * @brief      NRF24L01 进入接收模式
 * @note      设置 RX 地址, 写 RX 数据宽度, 选择 RF 频道, 波特率和 LNA_HCURR
 *            当 CE 变高后, 即进入 RX 模式, 并可以接收数据了
 * @param      无
 * @retval     无
 */
void nrf24l01_rx_mode(void)
{
    NRF24L01_CE(0);
    nrf24l01_write_buf(NRF_WRITE_REG + RX_ADDR_P0, (uint8_t *)RX_ADDRESS,
                       RX_ADR_WIDTH); /* 写 RX 节点地址 */

    nrf24l01_write_reg(NRF_WRITE_REG + EN_AA, 0x01); /* 使能通道 0 的自动应答 */
    nrf24l01_write_reg(NRF_WRITE_REG + EN_RXADDR, 0x01); /* 使能通道 0 的接收地址 */
    nrf24l01_write_reg(NRF_WRITE_REG + RF_CH, 40); /* 设置 RF 通信频率 */
    /* 选择通道 0 的有效数据宽度 */
    nrf24l01_write_reg(NRF_WRITE_REG + RX_PW_P0, RX_PLOAD_WIDTH);
    /* 设置 TX 发射参数, 0db 增益, 2Mbps */
    nrf24l01_write_reg(NRF_WRITE_REG + RF_SETUP, 0x0f);
    /* 配置基本工作模式的参数; PWR_UP, EN_CRC, 16BIT_CRC, 接收模式 */
    nrf24l01_write_reg(NRF_WRITE_REG + CONFIG, 0x0f);
    NRF24L01_CE(1); /* CE 为高, 进入接收模式 */
}
```

Tx 模式初始化过程：

- 1) 写 Tx 节点的地址
- 2) 写 Rx 节点的地址，主要为了使能硬件的自动应答
- 3) 使能通道 x 的自动应答
- 4) 使能通道 x 接收地址
- 5) 配置自动重发次数
- 6) 配置通信频率
- 7) 选择通道 x 的有效数据宽度
- 8) 配置发射参数（发射功率、无线速率）
- 9) 配置 NRF24L01 的基本参数以及切换工作模式

其代码如下：

```
/**
 * @brief      NRF24L01 进入发送模式
 * @note      设置 TX 地址, 写 TX 数据宽度, 设置 RX 自动应答地址, 填充 TX 发送数据, 选择 RF 频道,
 *            波特率和 PWR_UP, CRC 使能
 *            当 CE 变高后, 即进入 TX 模式, 并可以发送数据了, CE 为高大于 10us, 则启动发送.
 * @param      无
 * @retval     无
 */
void nrf24l01_tx_mode(void)
```

```

{
    NRF24L01_CE(0);
    nrf24l01_write_buf(NRF_WRITE_REG + TX_ADDR, (uint8_t *)TX_ADDRESS,
                      TX_ADR_WIDTH); /* 写 TX 节点地址 */
    nrf24l01_write_buf(NRF_WRITE_REG + RX_ADDR_P0, (uint8_t *)RX_ADDRESS,
                      RX_ADR_WIDTH); /* 设置 RX 节点地址,主要为了使能 ACK */

    nrf24l01_write_reg(NRF_WRITE_REG + EN_AA, 0x01); /* 使能通道 0 的自动应答 */
    nrf24l01_write_reg(NRF_WRITE_REG + EN_RXADDR, 0x01); /* 使能通道 0 的接收地址 */
    /* 设置自动重发间隔时间:500us + 86us;最大自动重发次数:10 次 */
    nrf24l01_write_reg(NRF_WRITE_REG + SETUP_RETR, 0x1a);
    nrf24l01_write_reg(NRF_WRITE_REG + RF_CH, 40); /* 设置 RF 通道为 40 */
    /* 设置 TX 发射参数,0db 增益,2Mbps,低噪声增益开启 */
    nrf24l01_write_reg(NRF_WRITE_REG + RF_SETUP, 0x0f);
    /* 配置基本工作模式的参数;PWR_UP,EN_CRC,16BIT_CRC,接收模式,开启所有中断 */
    nrf24l01_write_reg(NRF_WRITE_REG + CONFIG, 0x0e);
    NRF24L01_CE(1); /* CE 为高,10us 后启动发送 */
}

```

以上就是两种模式的配置,看过完整代码的,会发现 TX_ADDR 和 RX_ADDR 两个地址是一样的,跟前面说法一致,我们必须保持地址的匹配才能通信成功。以上代码中的发送函数都有一个特点,并不是单纯发送寄存器地址,而是**操作指令+寄存器地址**,这一点需要记得。NRF24L01 的操作指令也有好几个,它是配合寄存器完成特定的操作,其定义如下:

```

/* NRF24L01 寄存器操作命令 */
#define NRF_READ_REG    0x00 /* 读配置寄存器,低 5 位为寄存器地址 */
#define NRF_WRITE_REG   0x20 /* 写配置寄存器,低 5 位为寄存器地址 */
#define RD_RX_PLOAD     0x61 /* 读 RX 有效数据,1~32 字节 */
#define WR_TX_PLOAD     0xA0 /* 写 TX 有效数据,1~32 字节 */
#define FLUSH_TX        0xE1 /* 清除 TX FIFO 寄存器.发射模式下用 */
#define FLUSH_RX        0xE2 /* 清除 RX FIFO 寄存器.接收模式下用 */
#define REUSE_TX_PL     0xE3 /* 重新使用上一包数据,CE 为高,数据包被不断发送. */
#define NOP              0xFF /* 空操作,可以用来读状态寄存器 */

```

经过上面的发送或者接收模式初始化步骤后, NRF24L01 就可以准备启动发送数据或者等待接收数据了。

下面来看一下启动 NRF24L01 发送一次数据的函数,其定义如下:

```

/**
 * @brief      启动 NRF24L01 发送一次数据(数据长度 = TX_PLOAD_WIDTH)
 * @param      ptxbuf : 待发送数据首地址
 * @retval     发送完成状态
 * @arg        0      : 发送成功
 * @arg        1      : 达到最大发送次数,失败
 * @arg        0xFF   : 其他错误
 */
uint8_t nrf24l01_tx_packet(uint8_t *ptxbuf)
{
    uint8_t sta;
    uint8_t rval = 0xFF;

    NRF24L01_CE(0);
    /* 写数据到 TX BUF TX_PLOAD_WIDTH 个字节 */
    nrf24l01_write_buf(WR_TX_PLOAD, ptxbuf, TX_PLOAD_WIDTH);
    NRF24L01_CE(1); /* 启动发送 */

    while (NRF24L01_IRQ != 0); /* 等待发送完成 */

    sta = nrf24l01_read_reg(STATUS); /* 读取状态寄存器的值 */
    nrf24l01_write_reg(NRF_WRITE_REG + STATUS, sta); /* 清除 TX_DS/MAX_RT 中断标志 */

    if (sta & MAX_TX) /* 达到最大重发次数 */
    {

```

```

nrf24l01_write_reg(FLUSH_TX, 0xff); /* 清除 TX FIFO 寄存器 */
rval = 1;
}

if (sta & TX_OK) /* 发送完成 */
{
    rval = 0; /* 标记发送成功 */
}
return rval; /* 返回结果 */
}

```

在这里启动发送数据函数中，具体实现很简单，拉低片选信号→向发送数据寄存器写入数据→拉高片选信号。这里说明一下，在发送完寄存器号后都会返回一个 `status` 值，返回的这个值就是前面介绍的 `STATUS` 寄存器的内容。在这个基础上就可以知道数据是否发送完成以及现在的状态。

然后介绍一下 `NRF24L01` 接收一次数据函数，其定义如下：

```

/**
 * @brief      启动 NRF24L01 接收一次数据 (数据长度 = RX_PLOAD_WIDTH)
 * @param      prxbuf : 接收数据缓冲区首地址
 * @retval     接收完成状态
 * @arg        0 : 接收成功
 * @arg        1 : 失败
 */
uint8_t nrf24l01_rx_packet(uint8_t *prxbuf)
{
    uint8_t sta;
    uint8_t rval = 1;

    sta = nrf24l01_read_reg(STATUS); /* 读取状态寄存器的值 */
    nrf24l01_write_reg(NRF_WRITE_REG + STATUS, sta); /* 清除 TX_DS 或 MAX_RT 中断标志 */

    if (sta & RX_OK) /* 接收到数据 */
    {
        nrf24l01_read_buf(RD_RX_PLOAD, prxbuf, RX_PLOAD_WIDTH); /* 读取数据 */
        nrf24l01_write_reg(FLUSH_RX, 0xff); /* 清除 RX FIFO 寄存器 */
        rval = 0; /* 标记接收完成 */
    }
    return rval; /* 返回结果 */
}

```

在启动接收的过程中，首先需要判断当前 `NRF24L01` 的状态，往后才是真正的读取数据，清除接收寄存器的缓冲，完成数据的接收。这里需要注意的是我们通过 `RX_PLOAD_WIDTH` 和 `TX_PLOAD_WIDTH` 决定了接收和发送的数据宽度，这也决定每次发送和接收的有效字节数。`NRF24L01` 每次最多传输 32 个字节，再多的字节传输则需要多次传输。通信双方的发送和接收数据宽度必须一致才能正常通信。

2. main.c 代码

在 `main.c` 里编写如下代码：

```

int main(void)
{
    uint8_t key, mode;
    uint16_t t = 0;
    uint8_t tmp_buf[33];

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
}

```



```

key_init(); /* 初始化按键 */
nrf24l01_init(); /* 初始化 NRF24L01 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "NRF24L01 TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

while (nrf24l01_check()) /* 检查 NRF24L01 是否在线 */
{
    lcd_show_string(30, 110, 200, 16, 16, "NRF24L01 Error", RED);
    delay_ms(200);
    lcd_fill(30, 110, 239, 130 + 16, WHITE);
    delay_ms(200);
}
lcd_show_string(30, 110, 200, 16, 16, "NRF24L01 OK", RED);

while (1) /* 提醒用户选择模式 */
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        mode = 0; /* 接收模式 */
        break;
    }
    else if (key == KEY1_PRES)
    {
        mode = 1; /* 发送模式 */
        break;
    }

    t++;
    if (t == 100) /* 显示提示信息 */
    {
        lcd_show_string(10, 130, 230, 16, 16, "KEY0:RX_Mode KEY1:TX_Mode", RED);
    }
    if (t == 200) /* 关闭提示信息 */
    {
        lcd_fill(10, 130, 230, 150 + 16, WHITE);
        t = 0;
    }
    delay_ms(5);
}
lcd_fill(10, 130, 240, 166, WHITE); /* 清空上面的显示 */

if (mode == 0) /* RX 模式 */
{
    lcd_show_string(30, 130, 200, 16, 16, "NRF24L01 RX_Mode", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "Received DATA:", BLUE);
    nrf24l01_rx_mode(); /* 进入 RX 模式 */

    while (1)
    {
        if (nrf24l01_rx_packet(tmp_buf) == 0) /* 一旦接收到信息,则显示出来. */
        {
            tmp_buf[32] = 0; /* 加入字符串结束符 */
            lcd_show_string(0, 170, lcddev.width-1, 32, 16, (char *)tmp_buf, BLUE);
        }
        else
            delay_us(100);

        t++;
        if (t == 10000) /* 大约 1s 钟改变一次状态 */
        {

```

```

        t = 0;
        LED0_TOGGLE();
    }
}
else /* TX 模式 */
{
    lcd_show_string(30, 130, 200, 16, 16, "NRF24L01 TX_Mode", BLUE);
    nrf24l01_tx_mode(); /* 进入 TX 模式 */
    mode = ' '; /* 从空格键开始发送 */

    while (1)
    {
        if (nrf24l01_tx_packet(tmp_buf) == 0) /* 发送成功 */
        {
            lcd_show_string(30, 150, 239, 32, 16, "Sended DATA:", BLUE);
            lcd_show_string(0, 170, lcddev.width-1, 32, 16, (char *)tmp_buf, BLUE);
            key = mode;

            for (t = 0; t < 32; t++)
            {
                key++;
                if (key > ('~'))
                    key = ' ';
                tmp_buf[t] = key;
            }

            mode++;
            if (mode > ('~'))
                mode = ' ';
            tmp_buf[32] = 0; /* 加入结束符 */
        }
        else
        {
            lcd_fill(0, 150, lcddev.width, 170 + 16 * 3, WHITE); /* 清空显示 */
            lcd_show_string(30, 150, lcddev.width-1, 32, 16, "Send Failed ", BLUE);
        }

        LED0_TOGGLE();
        delay_ms(200);
    }
}
}

```

程序运行时先通过 `nrf24l01_cheak` 函数检测 NRF24L01 是否存在，如果存在，则让用户选择发送模式还是接收模式，在确定模式之后，设置 NRF24L01 的工作模式，然后执行相对应的数据发送/接收处理。

44.4 下载验证

将程序下载到开发板后，可以看到 LCD 显示的内容如图 44.4.1 所示：

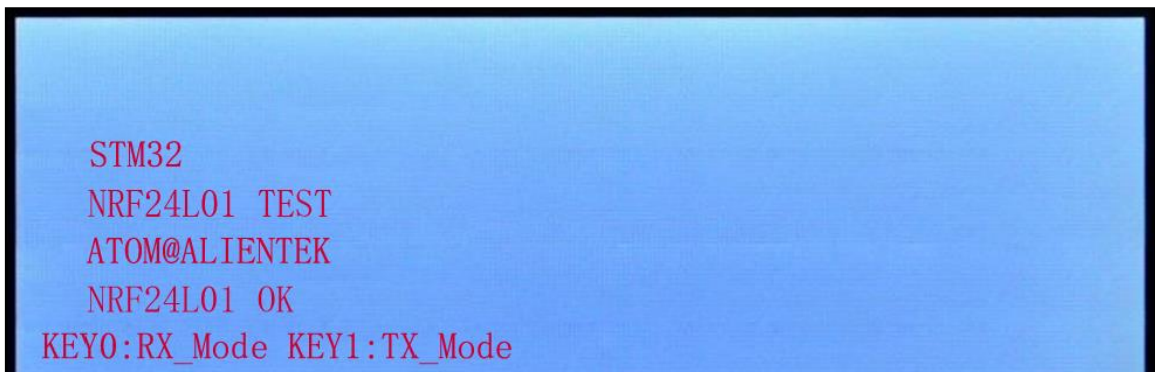


图 44.4.1 选择工作模式图

通过 KEY0 和 KEY1 来选择 NRF24L01 模块所要进入的工作模式，我们两个开发板一个选择发送模式，一个选择接收模式就可以了。设置好的通信界面如图 44.4.2 和图 44.4.3 所示：

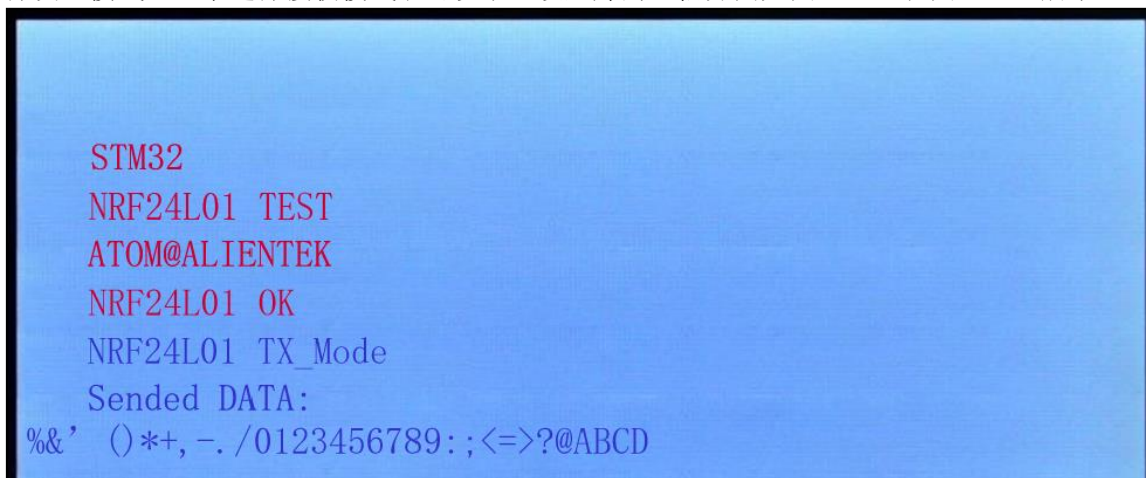


图 44.4.2 开发板 A 发送数据

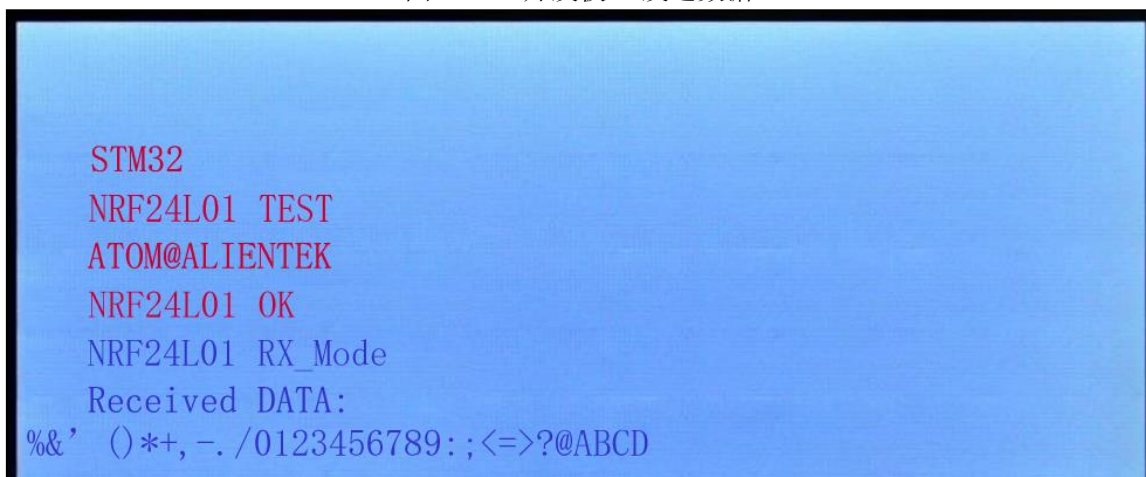


图 44.4.3 开发板 B 接收数据

图 44.4.2 来自于开发板 A，工作在发送模式。图 44.4.3 来自于开发板 B，工作在接收模式，A 发送，B 接收。发送和接收图片的数据不一样，是因为我们拍照的时间不一样导致的。大家看到收发数据是一致的，那就说明实验成功。

第四十五章 FLASH 模拟 EEPROM 实验

STM32 本身没有自带 EEPROM，但是 STM32 具有 IAP（在应用编程）功能，所以我们可以把它的 FLASH 当成 EEPROM 来使用。本章，我们将利用 STM32 内部的 FLASH 来实现第三十六章实验类似的效果，不过这次我们是将数据直接存放在 STM32 内部，而不是存放在 NOR FLASH。

本章分为如下几个小节：

45.1 STM32 FLASH 简介

45.2 硬件设计

45.3 软件设计

45.4 下载验证

45.1 STM32 FLASH 简介

不同型号的 STM32，其 FLASH 容量也有所不同，最小的只有 16K 字节，最大的则达到了 1024K 字节。战舰开发板选择的是 STM32F103ZET6，其 FLASH 容量为 512K 字节，属于大容量产品（另外还有中容量和小容量产品），大容量产品的闪存模块组织如表 45.1.1 所示：

| 块 | 名称 | FLASH 起始地址 | 大小 |
|------------|---------------|---------------------------|-------|
| 主存储器 | 页 0 | 0x0800 0000 – 0x0800 07FF | 2K |
| | 页 1 | 0x0800 0800 – 0x0800 0FFF | 2K |
| | 页 2 | 0x0800 1000 – 0x0800 17FF | 2K |
| | 页 3 | 0x0800 1800 – 0x0800 FFFF | 2K |
| | | | |
| | 页 255 | 0x0800 0800 – 0x0800 0FFF | 2K |
| 信息块 | 启动程序代码 | 0x1FFF F000 – 0x1FFF F7FF | 2K |
| | 用户选择字节 | 0x1FFF F800 – 0x1FFF F80F | 16 |
| 闪存存储器接口寄存器 | FLASH_ACK | 0x4002 2000 – 0x4002 2003 | 4 |
| | FLASH_KEYR | 0x4002 2004 – 0x4002 2007 | 4 |
| | FLASH_OPTKEYR | 0x4002 2008 – 0x4002 200B | 4 |
| | FLASH_SR | 0x4002 200C – 0x4002 200F | 4 |
| | FLASH_CR | 0x4002 2010 – 0x4002 2013 | 4 |
| | FLASH_AR | 0x4002 2014 – 0x4002 2017 | 4 |
| | 保留 | 0x4002 2018 – 0x4002 201B | 4 |
| | FLASH_OBR | 0x4002 201C – 0x4002 201F | 4 |
| | FLASH_WRP | 0x4002 2020 – 0x4002 2023 | 4 |

表 45.1.1 大容量产品闪存模块组织表

STM32 的闪存模块由主存储器、信息块和闪存存储器接口寄存器等 3 部分组成。

主存储器，该部分用来存放代码和数据常数（如 `const` 类型的数据）。对于大容量产品，其被划分为 256 页，每一页 2K 字节（注意：小容量和中容量产品每页只有 1K 字节）。从上表可以看出主存储器的起始地址就是 0x08000000，B0、B1 都接 GND 的时候，就是从 0x08000000 开始运行代码的。

信息块，该部分分为 2 个小部分，其中启动程序代码，用来存储 ST 自带的启动程序，用来串口下载代码，当 B0 接 3V3，B1 接 GND 的时候，运行的就是这部分代码。用户选中字节，则一般用于配置写保护、读保护等功能，本章不作介绍了。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制结构。

对主存储器和信息块的写入由内嵌的闪存编程/擦除控制器（FPEC）管理；编程与擦除的高电压由内部产生。

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正

确地进行。既在进行写或擦除操作时，不能进行代码或数据的读取操作。

45.1.1 闪存的读取

内置闪存模块可以在通用地址空间直接寻址，任何 32 位数据的读操作都能访问闪存模块的内容并得到相应的数据。读接口在闪存端包含一个读控制器，还包含一个 AHB 接口与 CPU 衔接。这个接口的主要工作是产生读内存的控制信号并预取 CPU 要求的指令块，预取指令块仅用于在 I-Code 总线上的取指操作，数据常量是通过 D-Code 总线访问的。这两条总线的访问目标是相同的闪存模块，访问 D-Code 将比预取指令优先级高。

这里要特别留意一个闪存等待时间，因为 CPU 运行速度比 FLASH 快得多，STM32F103 的 FLASH 最快访问速度 $\leq 24\text{Mhz}$ ，如果 CPU 频率超过这个速度，那么必须加入等待时间，比如我们一般使用 72Mhz 的主频，那么 FLASH 等待周期就必须设置为 2，该设置通过 FLASH_ACR 寄存器设置。

例如，我们要从地址 addr，读取一个半字（半字为 16 位，字为 32 位），可以通过如下的语句读取：

```
data = *(vu16*)addr;
```

将 addr 强制转换为 vu16 指针，然后取该指针所指向的地址的值，即得到了 addr 地址的值。类似的，将上面的 vu16 改为 vu8，即可读取指定地址的一个字节。相对 FLASH 读取来说，STM32 FLASH 的写就复杂一点了。下面我们介绍 STM32 闪存的编程和擦除。

45.1.2 闪存的编程和擦除

STM32 的闪存编程是由 FPEC(闪存编程和擦除控制器)模块处理的，这个模块包含 7 个 32 位寄存器，它们分别是：

- FPEC 键寄存器 (FLASH_KEYR)
- 选择字节键寄存器 (FLASH_OPTKEYR)
- 闪存控制寄存器 (FLASH_CR)
- 闪存状态寄存器 (FLASH_SR)
- 闪存地址寄存器 (FLASH_AR)
- 选择字节寄存器 (FLASH_WRPR)

其中 FPEC 键寄存器总共有 3 个键值：

RDPRT 键 = 0X0000 00A5

KEY1 = 0X4567 0123

KEY2 = 0XCDEF 89AB

STM32 复位后，FPEC 模块是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列到 FLASH_KEYR 寄存器可以打开 FPEC 模块（即写入 KEY1 和 KEY2），只有在写保护被解除后，我们才能操作相关寄存器。

STM32 闪存的编程每次必须写入 16 位（不能单纯的写入 8 位数据），当 FLASH_CR 寄存器的 PG 位为 '1' 时，在一个闪存地址写入一个半字将启动一次编程；写入任何非半字的数据，FPEC 都会产生总线错误。在编程过程中（BSY 位为 '1'），任何读写内存的操作都会使 CPU 暂停，直到此次闪存编程结束。

同样，STM32 的 FLASH 在编程的时候，也必须要求其写入地址的 FLASH 是被擦除了的（其值必须是 0xFFFF），否则无法写入，在 FLASH_SR 寄存器的 PGERR 位将得到一个警告。

STM32 的 FLASH 编程过程如图 45.1.2.1 所示：

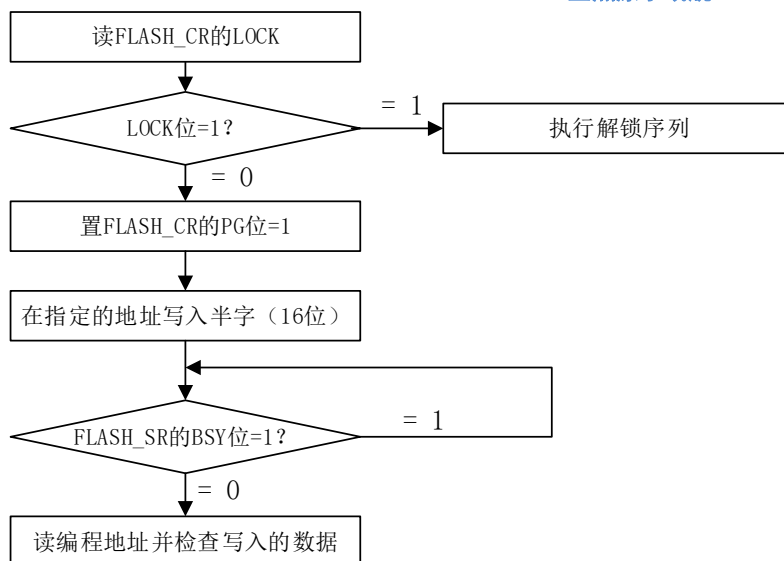


图 45.1.2.1 STM32 闪存编程过程

从上图可以得到闪存的编程顺序如下：

- 1) 检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁
- 2) 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的编程操作
- 3) 设置 FLASH_CR 寄存器的 PG 位为 ‘1’
- 4) 在指定的地址写入要编程的半字
- 5) 等待 BSY 位变为 ‘0’
- 6) 读出写入地址并验证数据

前面提到，我们在 STM32 的 FLASH 编程的时候，要先判断缩写地址是否被擦出了，所以，我们有必要再介绍一下 STM32 的闪存擦除，STM32 的闪存擦除分为两种：页擦除和整片擦除。页擦除过程如图 45.1.2.2 所示：

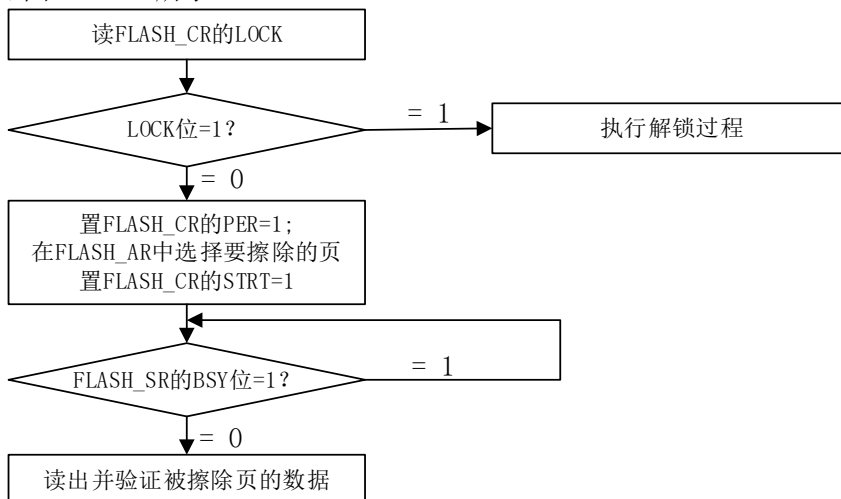


图 45.1.2.2 STM32 闪存页擦除过程

从上图可以看出，STM32 的页擦除顺序为：

- 1) 检查 FLASH_CR 和 LOCK 是否解锁，如果没有则先解锁
- 2) 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的闪存操作
- 3) 设置 FLASH_CR 寄存器的 PER 位为 ‘1’
- 4) 用 FLASH_AR 寄存器选择要擦除的页
- 5) 设置 FLASH_CR 寄存器的 STRT 位为 ‘1’
- 6) 等待 BSY 位变为 ‘0’
- 7) 读出被擦除的页并做验证

本章我们只用到了 STM32 页擦除功能，整片擦除功能我们在这里就不介绍了。

45.1.3 FLASH 寄存器

通过上面的讲解，我们基本对 STM32 闪存的读写执行步骤有所了解。接下来，我们介绍本实验需要用到的一些 FLASH 寄存器。

● FPEC 键寄存器 (FLASH_KEYR)

FPEC 键寄存器描述如图 45.1.3.2 所示：

| | | | | | | | | | | | | | | | |
|--------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| FKEYR[31:16] | | | | | | | | | | | | | | | |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FKEYR[15:0] | | | | | | | | | | | | | | | |
| W | W | W | W | W | W | W | W | W | W | W | W | W | W | W | W |

注：所有这些位是只写的，读出时返回0。

| | |
|-------|---|
| 位31~0 | FKEYR: FPEC键 这些位用于输入FPEC的解锁键。 |
|-------|---|

图 45.1.3.2 FLASH_KEYR 寄存器

该寄存器主要用来解锁 FPEC，必须在该寄存器写入特定的序列(KEY1 和 KEY2)解锁后，才能对 FLASH_CR 寄存器进行写操作。

● FLASH 控制寄存器 (FLASH_CR)

FLASH 控制寄存器描述如图 45.1.3.3 所示：

| | | | | | | | | | | | | | | | | |
|-----|----|---|-------|-----|-------|--------|-----|------|------|-------|-------|----|-----|-----|----|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | |
| 保留 | | | | | | | | | | | | | | | | |
| res | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 保留 | | | EOPIE | 保留 | ERRIE | OPTWRE | 保留 | LOCK | STRT | OPTER | OPTPG | 保留 | MER | PER | PG | |
| res | | rw | | res | rw | | res | | rw | | rw | | res | | rw | |
| 位7 | | LOCK: 锁 只能写'1'。 当该位为'1'时表示FPEC和FLASH_CR被锁住。在检测到正确的解锁序列后，硬件清除此位为'0'。 在一次不成功的解锁操作后，下次系统复位前，该位不能再被改变。 | | | | | | | | | | | | | | |
| 位6 | | STRT: 开始 当该位为'1'时将触发一次擦除操作。该位只可由软件置为'1'并在BSY变为'1'时清为'0'。 | | | | | | | | | | | | | | |
| 位1 | | PER: 页擦除 选择擦除页。 | | | | | | | | | | | | | | |
| 位0 | | PG: 编程 选择编程操作。 | | | | | | | | | | | | | | |

图 45.1.3.3 FLASH_CR 寄存器

该寄存器我们本章只用到了它的 LOCK、STRT、PER 和 PG 等 4 个位。

LOCK 位，该位用于指示 FLASH_CR 寄存器是否被锁住，该位在检测到正确的解锁序列后，硬件将其清零。在一次不成功的解锁操作后，在下次系统复位之前，该位将不再改变。

STRT 位，该位用于开始一次擦除操作。在该位写入 1，将执行一次擦除操作。

PER 位，该位用于选择页擦除操作，在页擦除的时候，需要将该位置 1。

PG 位，该位用于选择编程操作，在往 FLASH 写数据的时候，该位需要置 1。

其他位，我们就不在这里介绍了，请大家参考《STM32F10xxx 闪存编程参考手册》。

● 闪存状态寄存器 (FLASH_SR)

闪存状态寄存器描述如图 45.1.3.4 所示：

| | | | | | | | | | | | | | | | |
|--------------------|----|--|----|----|----|----|----|----|----|-----|--------------|----|-------|----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 保留 | | | | | | | | | | | | | | | |
| res | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | | | | | | | | | | EOP | WRPRT ERR | 保留 | PGERR | 保留 | BSY |
| res | | | | | | | | | | | | | | | |
| rw rw res rw res r | | | | | | | | | | | | | | | |
| 位5 | | EOP: 操作结束 当闪存操作(编程/擦除)完成时, 硬件设置这位为'1', 写入'1'可以清除这位状态。 <i>注: 每次成功的编程或擦除都会设置EOP状态。</i> | | | | | | | | | | | | | |
| 位4 | | WRPRTERR: 写保护错误 试图对写保护的闪存地址编程时, 硬件设置这位为'1', 写入'1'可以清除这位状态。 | | | | | | | | | | | | | |
| 位3 | | 保留。必须保持为清除状态'0' | | | | | | | | | | | | | |
| 位2 | | PGERR: 编程错误 试图对内容不是'0xFFFF'的地址编程时, 硬件设置这位为'1', 写入'1'可以清除这位状态。 <i>注: 进行编程操作之前, 必须先清除FLASH_CR寄存器的STRT位。</i> | | | | | | | | | | | | | |
| 位1 | | 保留。必须保持为清除状态'0' | | | | | | | | | | | | | |
| 位0 | | BSY: 忙 该位指示闪存操作正在进行。在闪存操作开始时, 该位被设置为'1'; 在操作结束或发生错误时该位被清除为'0'。 | | | | | | | | | | | | | |

图 45.1.3.4 FLASH_SR 寄存器

该寄存器主要用来指示当前 FPEC 的操作编程状态。由于寄存器中描述比较详细, 这里就不重复了。

● 闪存地址寄存器 (FLASH_AR)

闪存地址寄存器描述如图 45.1.3.5 所示:

| | | | | | | | | | | | | | | | |
|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| FAR[31:16] | | | | | | | | | | | | | | | |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FAR[15:0] | | | | | | | | | | | | | | | |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 这些位由硬件修改为当前/最后使用的地址。在页擦除操作中, 软件必须修改这个寄存器以指定要擦除的页。 | | | | | | | | | | | | | | | |
| 位31~0 | | FAR: 闪存地址 当进行编程时选择要编程的地址, 当进行页擦除时选择要擦除的页。 <i>注意: 当FLASH_SR中的BSY位为'1'时, 不能写这个寄存器。</i> | | | | | | | | | | | | | |

图 45.1.3.5 FLASH_AR 寄存器

该寄存器在本章, 我们主要用来设置要擦除的页。

关于 STM32 FLASH 的介绍, 我们就介绍到这里。更详细的介绍, 可以参考《STM32F10xxx 闪存编程参考手册》。

45.2 硬件设计

1. 例程功能

按键 KEY1 控制写入 FLASH 的操作, 按键 KEY0 控制读出操作, 并在 TFTLCD 模块上显示相关信息, 还可以借助 USART 进行读取或者写入操作。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) 独立按键
KEY0 - PE4 KEY1 - PE3

45.3 程序设计

45.3.1 FLASH 的 HAL 库驱动

FLASH 在 HAL 库中的驱动代码在 stm32f1xx_hal_flash.c 和 stm32f1xx_hal_flash_ex.c 文件(及其头文件)中。

1. HAL_FLASH_Unlock 函数

解锁闪存控制寄存器访问的函数, 其声明如下:

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);
```

- **函数描述:**
用于解锁闪存控制寄存器的访问, 在对 FLASH 进行写操作前必须先解锁, 解锁操作也就是必须在 FLASH_KEYR 寄存器写入特定的序列 (KEY1 和 KEY2)。
- **函数形参:**
无
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

2. HAL_FLASH_Lock 函数

锁定闪存控制寄存器访问的函数, 其声明如下:

```
HAL_StatusTypeDef HAL_FLASH_Lock (void);
```

- **函数描述:**
用于锁定闪存控制寄存器的访问。
- **函数形参:**
无
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

3. HAL_FLASH_Program 函数

闪存写操作函数, 其声明如下:

```
HAL_StatusTypeDef HAL_FLASHEx_Program(uint32_t TypeProgram, uint32_t Address,
                                         uint64_t Data);
```

- **函数描述:**
该函数用于 FLASH 的写入。
- **函数形参:**
形参 1 是 TypeProgram 用来区分要写入的数据类型, 取值可为字节、半字、字和双字, 用户根据写入数据类型选择即可。
形参 2 是 Address 用来设置要写入数据的 FLASH 地址。
形参 3 是 Data 是要写入的数据类型。该参数默认 64 位, 如果你要写入小于 64 位的数据, 比如 16 位, 程序会进行类型转换。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

4. HAL_FLASHEx_Erase 函数

闪存擦除函数, 其声明如下:

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                                       uint32_t *SectorError);
```

- **函数描述:**
该函数用于大量擦除或擦除指定的闪存扇区。
- **函数形参:**
形参 1 是 FLASH_EraseInitTypeDef 结构体类型指针变量。

```
typedef struct
{
    uint32_t TypeErase;          /* 擦除类型 (Page 擦除 / BANK 级别批量擦除) */
    uint32_t Banks;              /* 擦除的 Bank 编号 (批量擦除时才有效) */
    uint32_t PageAddress;        /* 擦除页面地址 */
    uint32_t NbPages;            /* 擦除的页面数 */
} FLASH_EraseInitTypeDef;
```

成员变量 TypeErase 用来设置擦除类型，是 page 擦除还是 BANK 级别的批量擦除，取值为 FLASH_TYPEERASE_PAGES 或者 FLASH_TYPEERASE_MASSERASE，这个比较好理解，如果一次擦除一个 Bank 下面的所有 Page，那么需要选择 FLASH_TYPEERASE_MASSERASE。成员变量 Banks 用来设置要擦除的 Bank 编号，这个只有设置为批量擦除的时候才有效。成员变量 PageAddress 用来设置要擦除页面的地址。成员变量 NbPages 用来设置要擦除的页面数。

形参 2 是 uint32_t 类型指针变量，存放错误码，0xFFFFFFFF 值表示扇区已被正确擦除，其它值表示擦除过程中的错误扇区。

- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

5. FLASH_WaitForLastOperation 函数

等待 FLASH 操作完成函数，其声明如下：

```
HAL_StatusTypeDef FLASH_WaitForLastOperation(uint32_t Timeout);
```

- **函数描述:**
该函数用于等待 FLASH 操作完成。
- **函数形参:**
形参 1 是 FLASH 操作超时时间。
- **函数返回值:**
HAL_StatusTypeDef 枚举类型的值。

45.3.2 程序流程图

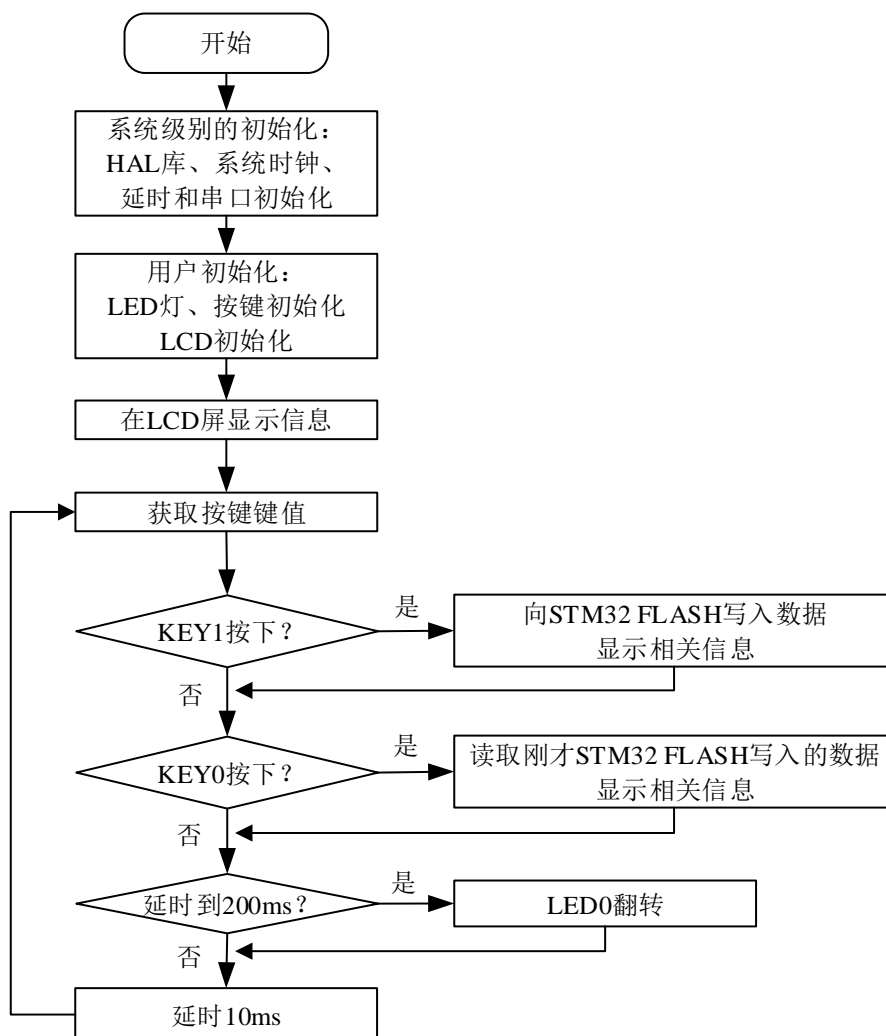


图 45.3.2.1 FLASH 模拟 EEPROM 实验程序流程图

45.3.3 程序解析

1. STM FLASH 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。STM FLASH 驱动源码包括两个文件：stmflash.c 和 stmflash.h。

stmflash.h 头文件做了一些比较重要的宏定义，定义如下：

```

/* FLASH 起始地址 */
#define STM32_FLASH_BASE      0x08000000      /* STM32 FLASH 起始地址 */
#define STM32_FLASH_SIZE     0x80000         /* STM32 FLASH 总大小 */
/* STM32F103 扇区大小 */
#if STM32_FLASH_SIZE < 256 * 1024
#define STM32_SECTOR_SIZE 1024      /* 容量小于 256K 的 F103，扇区大小为 1K 字节 */
#else
#define STM32_SECTOR_SIZE 2048      /* 容量大于等于 256K 的 F103，扇区大小为 2K 字节 */
#endif

```

STM32_FLASH_BASE 和 STM32_FLASH_SIZE 分别是 FLASH 的起始地址和 FLASH 总大小，这两个宏定义随着芯片是固定的，我们战舰开发板的 F103 芯片 FLASH 是 512K 字节，所以 STM32_FLASH_SIZE 宏定义值为 0x80000。

下面我们开始介绍 stmflash.c 的程序，下面先介绍一下 stmflash 写操作函数，源码如下：

```

/**
 * @brief      在 FLASH 指定位置，写入指定长度的数据(自动擦除)
 * @note      该函数往 STM32 内部 FLASH 指定位置写入指定长度的数据
 *            该函数会先检测要写入的扇区是否是空(全 0xFFFF)的？，如果
 *            不是，则先擦除，如果是，则直接往扇区里面写入数据。
 *            数据长度不足扇区时，自动被回擦除前的数据
 * @param      waddr    : 起始地址 (此地址必须为 2 的倍数!!，否则写入出错!)
 * @param      pbuf     : 数据指针
 * @param      length   : 要写入的 半字(16 位)数
 * @retval     无
 */
uint16_t g_flashbuf[STM32_SECTOR_SIZE / 2]; /* 最多是 2K 字节 */
void stmflash_write(uint32_t waddr, uint16_t *pbuf, uint16_t length)
{
    uint32_t secpos;          /* 扇区地址 */
    uint16_t secoff;          /* 扇区内偏移地址(16 位字计算) */
    uint16_t secremain;       /* 扇区内剩余地址(16 位字计算) */
    uint16_t i;
    uint32_t offaddr;          /* 去掉 0x08000000 后的地址 */
    FLASH_EraseInitTypeDef flash_eraseop;
    uint32_t erase_addr;       /* 擦除错误，这个值为发生错误的扇区地址 */

    if(waddr < STM32_FLASH_BASE || (waddr >= (STM32_FLASH_BASE + 1024 * STM32_FLASH_SIZE)))
    {
        return;               /* 非法地址 */
    }

    HAL_FLASH_Unlock();       /* FLASH 解锁 */

    offaddr = waddr - STM32_FLASH_BASE;          /* 实际偏移地址 */
    secpos = offaddr / STM32_SECTOR_SIZE;         /* 得到扇区编号 */
    secoff = (offaddr % STM32_SECTOR_SIZE) / 2; /* 在扇区内的偏移(2B 为基本单位) */
    secremain = STM32_SECTOR_SIZE / 2 - secoff; /* 扇区剩余空间大小 */

    if (length <= secremain)
    {
        secremain = length; /* 不大于该扇区范围 */
    }

    while (1)
    {
        stmflash_read(secpos * STM32_SECTOR_SIZE + STM32_FLASH_BASE,
                       g_flashbuf, STM32_SECTOR_SIZE / 2); /* 读出整个扇区的内容 */

        for (i = 0; i < secremain; i++) /* 校验数据 */
        {
            if (g_flashbuf[secoff + i] != 0xFFFF)
            {
                break; /* 需要擦除 */
            }
        }

        if (i < secremain) /* 需要擦除 */
        {
            flash_eraseop.TypeErase = FLASH_TYPEERASE_PAGES; /* 选择页擦除 */
            flash_eraseop.NbPages = 1; /* 要擦除的页数 */
            flash_eraseop.PageAddress = secpos * STM32_SECTOR_SIZE +
                                         STM32_FLASH_BASE; /* 要擦除的起始地址 */
            HAL_FLASHEx_Erase(&flash_eraseop, &erase_addr);

            for (i = 0; i < secremain; i++) /* 复制 */
        }
    }
}

```



```

        g_flashbuf[i + secoff] = pbuf[i];
    }
    stmflash_write_nocheck(secpos * STM32_SECTOR_SIZE + STM32_FLASH_BASE,
        g_flashbuf, STM32_SECTOR_SIZE / 2); /* 写入整个扇区 */
}
else
{
    /* 写已经擦除了的,直接写入扇区剩余区间 */
    stmflash_write_nocheck(waddr, pbuf, secremain);
}

if (length == secremain)
{
    break; /* 写入结束了 */
}
else /* 写入未结束 */
{
    secpos++; /* 扇区地址增 1 */
    secoff = 0; /* 偏移位置为 0 */
    pbuf += secremain; /* 指针偏移 */
    waddr += secremain * 2; /* 写地址偏移(16 位数据地址,需要*2) */
    length -= secremain; /* 字节(16 位)数递减 */

    if (length > (STM32_SECTOR_SIZE / 2))
    {
        secremain = STM32_SECTOR_SIZE / 2; /* 下一个扇区还是写不完 */
    }
    else
    {
        secremain = length; /* 下一个扇区可以写完了 */
    }
}
}
HAL_FLASH_Lock(); /* 上锁 */
}

```

该函数用于在 STM32 的指定地址写入指定长度的数据。函数的实现基本类似 SPI 章节的 `norflash_write` 函数，不过该函数对于写入地址是有要求，必须保证以下两点：

- 1、写入地址必须是用户代码区以外的地址。
- 2、写入地址必须是 2 的倍数。

第 1 点比较好理解，如果把用户代码给擦了，可想而知你运行的程序可能就被废了，从而很可能出现死机的情况。第 2 点则是 STM32 FLASH 的要求，每次必须写入 16 位，如果你写的地址不是 2 的倍数，那么写入的数据，可能就不是写在你要写的地址了。

另外，该函数的 `g_flashbuf` 数组，也是根据所用 STM32 的 FLASH 容量来确定的，战舰 STM32 开发板的 FLASH 是 512K 字节，所以 `STM_SECTOR_SIZE` 的值为 2048，故该数组大小为 2K 字节。

`stmflash_write` 函数实质是调用 `stmflash_write_nocheck` 函数进行实现，下面再来看一下 `stmflash_write` 函数代码，其代码如下：

```

/**
 * @brief      不检查的写入
 *             这个函数的假设已经把原来的扇区擦除过再写入
 * @param      waddr    : 起始地址 (此地址必须为 2 的倍数!! , 否则写入出错!)
 * @param      pbuf     : 数据指针
 * @param      length   : 要写入的 半字(16 位)数
 * @retval     无
 */
void stmflash_write_nocheck(uint32_t waddr, uint16_t *pbuf, uint16_t length)
{
    uint16_t i;

    for (i = 0; i < length; i++)
    {

```

```

        HAL_FLASH_Program(FLASH_TYPEPROGRAM_HALFWORD, waddr, pbuf[i]);
        waddr += 2;      /* 指向下一个半字 */
    }
}

```

该函数的实现依靠 flash 的 HAL 库驱动 HAL_FLASH_Program 进行实现。由于前面已经对 HAL_FLASH_Program 进行说明，这里就不作展开说明了。

接下来，讲解一下 STM FLASH 读相关的函数，写函数也有调用到读函数，其代码如下：

```

/**
 * @brief      从指定地址读取一个半字 (16 位数据)
 * @param      faddr : 读取地址 (此地址必须为 2 的倍数!!)
 * @retval     读取到的数据 (16 位)
 */
uint16_t stmflash_read_halfword(uint32_t faddr)
{
    return *(volatile uint16_t *)faddr;
}

/**
 * @brief      从指定地址开始读出指定长度的数据
 * @param      raddr : 起始地址
 * @param      pbuf : 数据指针
 * @param      length: 要读取的半字 (16 位) 数, 即 2 个字节的整数倍
 * @retval     无
 */
void stmflash_read(uint32_t raddr, uint16_t *pbuf, uint16_t length)
{
    uint16_t i;

    for (i = 0; i < length; i++)
    {
        pbuf[i] = stmflash_read_halfword(raddr);    /* 读取 2 个字节 */
        raddr += 2;                                /* 偏移 2 个字节 */
    }
}

```

前面也提及到 STM32 对 FLASH 写入，其写入地址的值必须是 0xFFFFFFFF，所以读函数主要是读取地址的值，以给写函数调用检验，确保能写入成功。读函数实现比较简单，这里就不做展开了。

2. main.c 代码

在 main.c 里面编写如下代码：

```

const uint8_t g_text_buf[] = {"STM32 FLASH TEST"}; /* 要写入的 FLASH 字符串数组 */
#define TEXT_LENGTH sizeof(g_text_buf)              /* 数组长度 */

/* SIZE 表示半字长 (2 字节)，大小必须是 2 的整数倍，如果不是的话，强制对齐到 2 的整数倍 */
#define SIZE TEXT_LENGTH / 2 + ((TEXT_LENGTH % 2) ? 1 : 0)

/* 设置 FLASH 保存地址 (必须为偶数，且其值要大于本代码所占用的 FLASH 的大小 + 0x08000000) */
#define FLASH_SAVE_ADDR 0x08070000

int main(void)
{
    uint8_t key = 0;
    uint16_t i = 0;
    uint8_t datatemp[SIZE];

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
}

```

```
led_init();          /* 初始化 LED */
lcd_init();          /* 初始化 LCD */
key_init();          /* 初始化按键 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "FLASH EEPROM TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write KEY0:Read", RED);

while (1)
{
    key = key_scan(0);

    if (key == KEY1_PRES)    /* KEY1 按下,写入 STM32 FLASH */
    {
        lcd_fill(0, 150, 239, 319, WHITE);
        lcd_show_string(30, 160, 200, 16, 16, "Start Write FLASH...", RED);
        stmflash_write(FLASH_SAVE_ADDR, (uint16_t *)g_text_buf, SIZE);
        lcd_show_string(30, 150, 200, 16, 16, "FLASH Write Finished!", RED);
    }

    if (key == KEY0_PRES)    /* KEY0 按下,读取字符串并显示 */
    {
        lcd_show_string(30, 150, 200, 16, 16, "Start Read FLASH.... ", RED);
        stmflash_read(FLASH_SAVE_ADDR, (uint16_t *)datatemp, SIZE);
        lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is: ", RED);
        lcd_show_string(30, 170, 200, 16, 16, (char *)datatemp, BLUE);
    }

    i++;
    delay_ms(10);

    if (i == 20)
    {
        LED0_TOGGLE();    /* 提示系统正在运行 */
        i = 0;
    }
}
```

主函数代码逻辑比较简单，当检测到按键 KEY1 按下后往 FLASH 指定地址开始的连续地址空间写入一段数据，当检测到按键 KEY0 按下后读取 FLASH 指定地址开始的连续空间数据。

最后，我们将 stmflash_read_word 和 test_write 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，调用 STM32F103 的 FLASH 读写函数，方便测试。

45.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 45.4.1 所示：

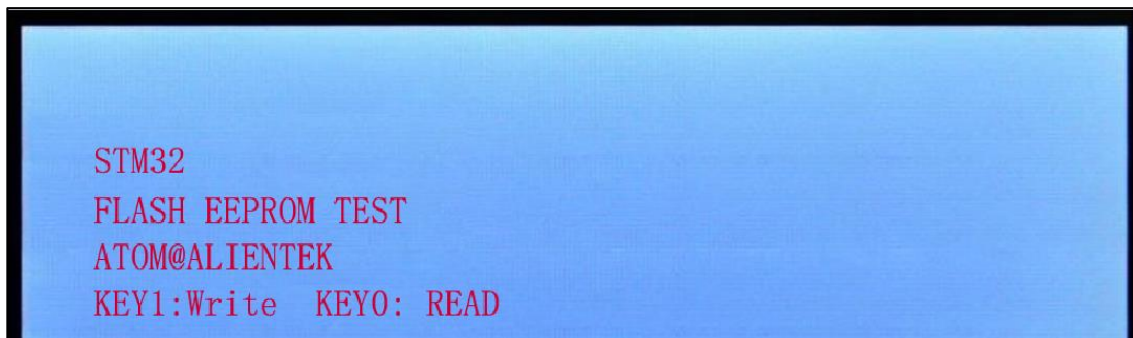


图 45.4.1 程序运行效果图

通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 45.4.2 所示：

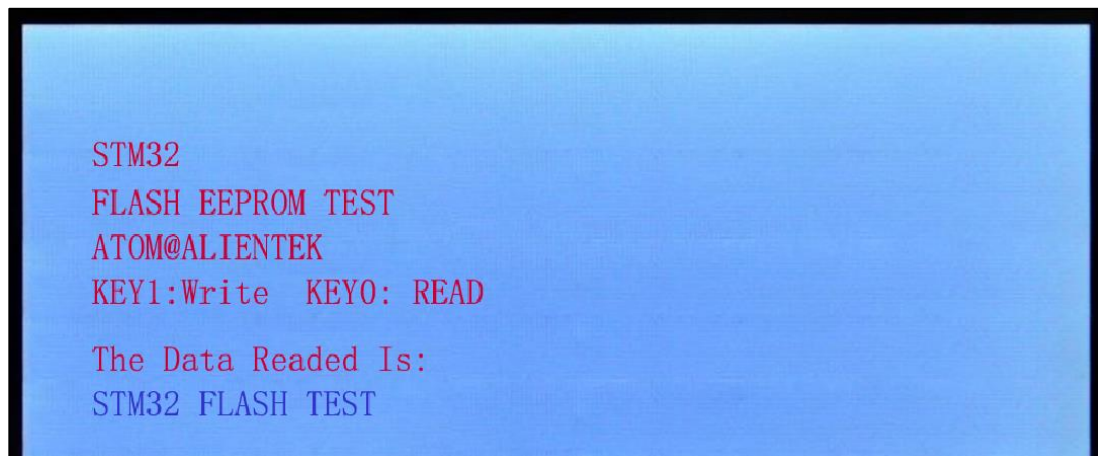


图 45.4.2 操作后的显示效果图

本实验的测试，我们还可以借助 USART，调用：stmflash_read_word 和 test_write 函数进行测试！

第四十六章 摄像头实验

正点原子战舰 STM32 开发板板载了一个摄像头接口（P6），该接口可以用来连接正点原子 OV7725 摄像头模块。本章，我们将使用 STM32 驱动正点原子 OV7725 摄像头模块，实现摄像头功能。

本章分为如下几个部分：

46.1 OV7725 简介

46.2 硬件设计

46.3 软件设计

46.4 下载验证

46.1 OV7725 模块简介

本部分将从正点原子的 OV7725 模块的原理和驱动方法讲述摄像头模块的使用方法。

46.1.1 正点原子 OV7725 模块

正点原子 OV7725 模块是正点原子推出的一款高性能 30W 像素高清摄像头模块。该模块采用 OmniVision 公司生产的一颗 1/4 英寸 CMOS VGA (640*480) 图像传感器：OV7725。正点原子的 OV7725 模块采用该 OV7725 传感器作为核心部件，集成有源晶振和 FIFO (AL422B)，可以调整缓存摄像头的图像数据，任意一款 MCU 都可控制该模块和读取图像。

它的外形如图 46.1.1.1 所示：

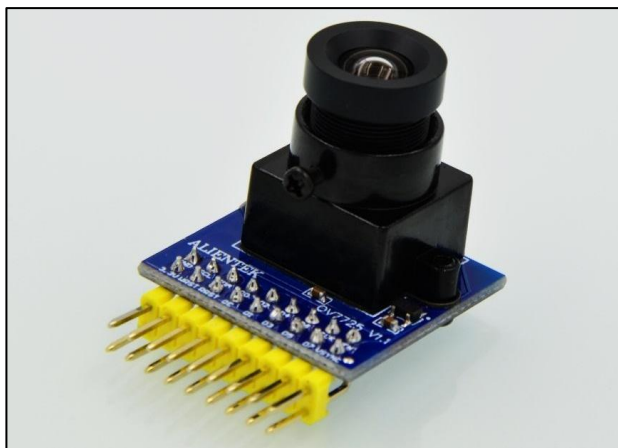


图 46.1.1.1 正点原子 OV7725 模块

正点原子 OV7725 模块的特点如下：

- 高灵敏度、低电压适合嵌入式应用
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(GBR4:2:2, RGB565/RGB555/RGB444), YUV(4:2:2) 和 YCbCr (4:2:2) 输出格式
- 支持 VGA、QVGA，和从 CIF 到 40*30 的各种尺寸输出
- 自动图像控制功能：自动曝光 (AEC)、自动白平衡 (AWB)、自动消除灯光条纹、自动黑电平校准 (ABLC) 和自动带通滤波器 (ABF)
- 支持图像质量控制：色饱和度调节、色调调节、gamma 校准、锐度和镜头校准等
- 支持图像缩放、平移和窗口设置
- 集成有源晶振，无需外部提供时钟
- 集成 FIFO 芯片 (AL422B)，方便 MCU 读取图像
- 自带嵌入式微处理器

OV7725 的功能框图如图 46.1.1.2 所示：

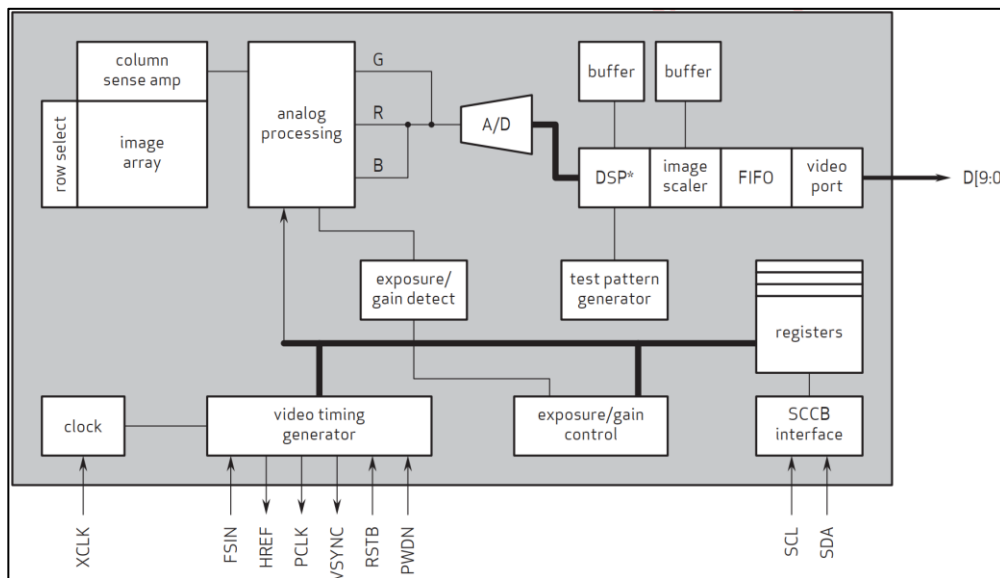


图 46.1.1.2 OV7725 功能框图

由上图可知，感光阵列（image array）在 XCLK 时钟的驱动下进行图像采样，输出 640*480 阵列的模拟数据；接着模拟信号处理器在时序发生器（video timing generator）的控制下对模拟数据进行算法处理（analog processing）；模拟数据处理完成后分成 G（绿色）和 R/B（红色/蓝色）两路通道经过 AD 转换器后转换成数字信号，并且通过 DSP 进行相关图像处理，最终输出所配置格式的 10 位视频数据流。模拟信号处理以及 DSP 等都可以通过寄存器（register）来配置，配置寄存器的接口就是 SCCB 接口。

正点原子 OV7725 模块原理图如下图 46.1.1.3 所示：

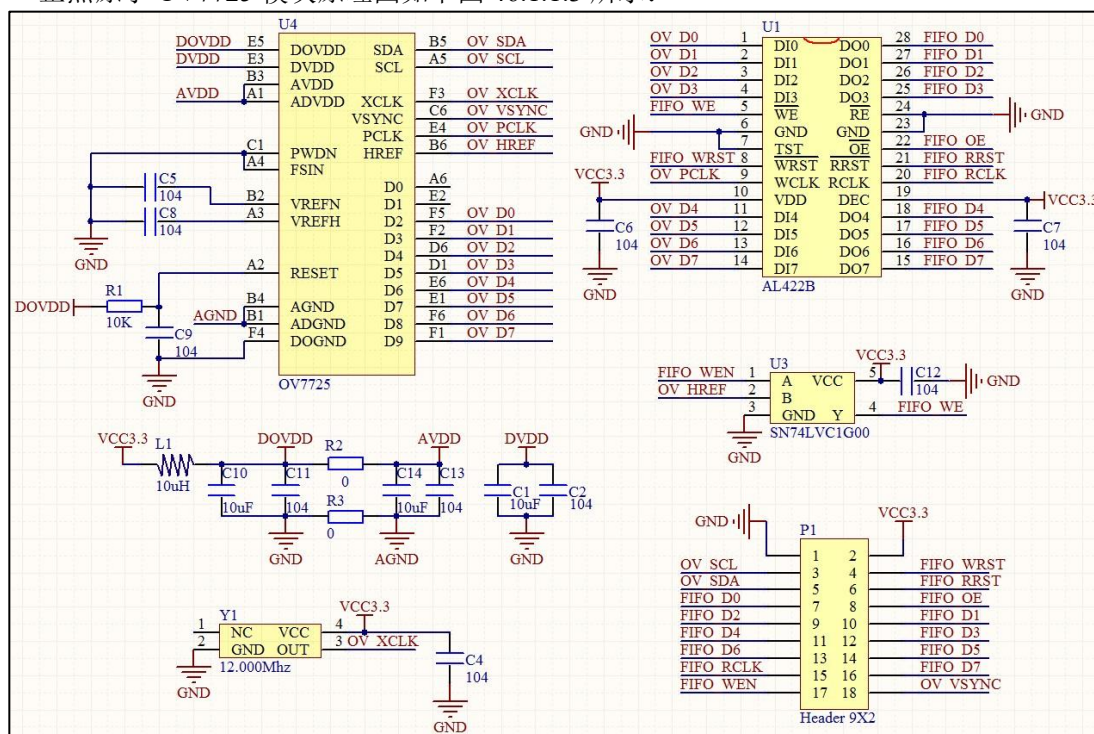


图 46.1.1.3 正点原子 OV7725 摄像头模块原理图

从上图可以看出，正点原子 OV7725 摄像头模块自带了有源晶振，用于产生 12M 时钟作为 OV7725 传感器的 XCLK 输入；带有一个 FIFO 芯片(AL422B)，该 FIFO 芯片的容量是 384K 字节，足够存储 2 帧 QVGA 的图像数据。当驱动好 OV7725 模块，图像数据就被存放到 FIFO 中，获取图像数据就是对 FIFO 进行读取，这个过程需要用到的引脚就是上图中的 2*9 双排座。模

块就是通过一个 2*9 的双排排针(P1)与外部通信，与外部通信信号如表 46.1.1.1 所示：

| 信号 | 作用描述 | 信号 | 作用描述 |
|-------------|---------------|-----------|--------------|
| VCC3.3 | 模块供电脚，接3.3V电源 | FIFO_WEN | FIFO写使能 |
| GND | 模块地线 | FIFO_WRST | FIFO写指针复位 |
| OV_SCL | SCCB通信时钟信号 | FIFO_RRST | FIFO读指针复位 |
| OV_SDA | SCCB通信数据信号 | FIFO_OE | FIFO输出使能（片选） |
| FIFO_D[7:0] | FIFO输出数据（8位） | OV_VSYNC | 帧同步信号 |
| FIFO_RCLK | 读FIFO时钟 | | |

表 46.1.1.1 OV7725 模块接口信号描述

46.1.2 串行摄像头控制总线（SCCB）简介

正点原子 OV7725 摄像头模块的所有配置，包括图像数据格式、分辨率以及图像处理参数等，都是通过 SCCB 总线来实现的。

SCCB 全称是：Serial Camera Control Bus 即串行摄像头控制总线，是由 OV（OmniVision 的简称）公司定义和发展的三线式串行总线。不过，OV 公司为了减少传感器引脚的封装，现在 SCCB 总线大多采用两线式接口总线。

OV7725 使用的两线式接口总线，由两条数据线组成：一条是用于传输时钟信号的 SIO_C（即 OV_SCL），另一条是用于传输数据信号的 SIO_D（即 OV_SDA），这两条数据线类型 IIC 协议中的 SCL 和 SDA 信号线。在前面提及到 SCCB 协议兼容 IIC 协议，是因为 SCCB 协议和 IIC 协议非常相似，有关 IIC 协议的详细介绍请大家参考前面的“IIC 实验”章节。

SCCB 包括三种传输周期（也就是协议），即 3 相写传输周期，2 相写传输周期和 2 相读传输周期。3 相写传输周期相当于写操作，而读操作是符合的需要 2 相写传输周期和 2 相读传输周期进行结合。这里的相指的是传输的单位，一字节称为一个相。

SCCB 的写传输协议如下图 46.1.2.1 所示：

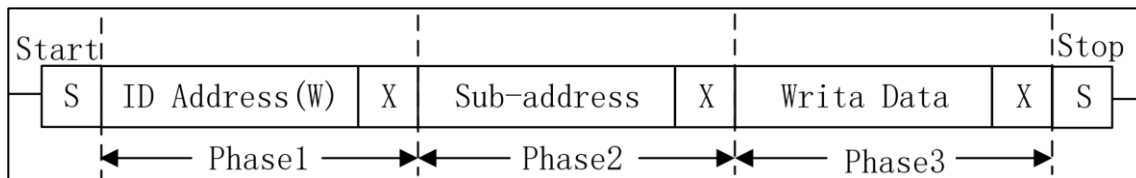


图 46.1.2.1 SCCB 写传输协议

上图中就是三相写传输周期，第一个相就是 ID Address，由 7 位器件地址和 1 位读写控制位构成（0：写 1：读），而 OV7725 器件地址为 0x21，所以在写传输时序中，ID Address(W)为 0x42（器件地址左移 1 位，低位补 0）；第二个相就是 Sub-address，即 8 位寄存器地址，在 OV7725 的数据手册中定义了 0x00~0xAC 共 173 个寄存器，有些寄存器是可写的，有些是只读的，只有可写的寄存器才能正确写入；第三个相就是 Write Data，即要写入寄存器的 8 位配置数据。而上图中的第 9 位 X 表示 Don't Care（不必关心位），该位是由从机（此处指 OV7725）发出应答信号来响应主机表示当前 ID Address、Sub-address 和 Write Data 是否传输完成，但是从机有可能不发出应答信号，因此主机（此处指 STM32）可不用判断此处是否有应答，直接默认当前传输完成即可。

SCCB 和 IIC 写传输协议是极为相似，只是在 SCCB 写传输时序中，第 9 位为不必关心位，而 IIC 写传输协议中为应答位。SCCB 的读传输时序和 IIC 有些差异，在 IIC 读传输协议中，写完寄存器地址后，会有一个 restart 即重复开始的操作；而 SCCB 读传输协议中没有重复开始的概念，在写完寄存器地址后，发起总线停止信号。

SCCB 读传输协议如下图 46.1.2.2 所示。

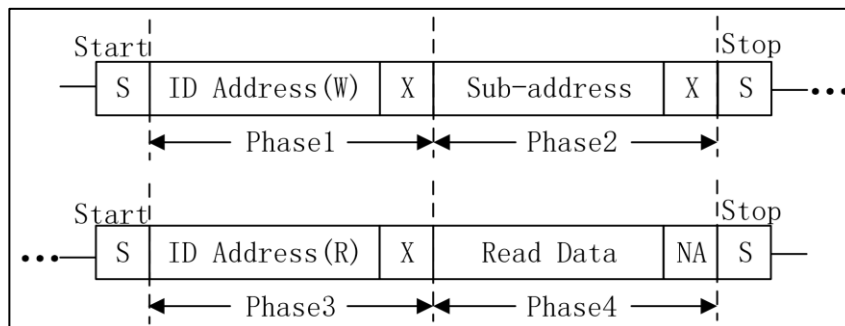


图 46.1.2.2 SCCB 读传输协议

SCCB 读传输协议由两个部分组成：二相写传输周期和二相读传输周期。跟 IIC 的读操作是相似的，都是复合的过程。第一部分是写器件地址和寄存器地址，即先进行一次虚写操作，通过这种虚写操作使地址指针向虚写操作中寄存器地址的位置。第二部分就是读器件地址和读数据，此时读取到的数据才是寄存器地址对应的数据，这里的读器件地址为 0x43（器件地址左移 1 位，低位补 1）。上图中的 NA 位是由主机（这里指 STM32）产生，由于 SCCB 总线不支持连续读写，因此 NA 位必须为高电平。

关于 SCCB 的详细介绍，请大家参考正点原子 OV7725 摄像头模块资料里《OmniVision Technologies Seril Camera Control Bus(SCCB) Specification.pdf》这个文档。

OV7725 的初始化，需要配置大量的寄存器，这里我们就不给大家多做介绍了，请大家参考正点原子 OV7725 摄像头模块资料里《OV7725 Software Application Note.pdf》。

46.1.3 输出时序说明

当使用SCCB总线对OV7725进行寄存器配置后，就会输出图像数据。通过查看输出时序图就可以知道如何进行图像数据的获取。由于OV7725支持多种尺寸(分辨率)输出，所以在这里简单介绍一下这些尺寸定义：

VGA，即分辨率为 640 * 480 的输出模式；

QVGA，即分辨率为 320 * 240 的输出模式；

QQVGA，即分辨率为 160 * 120 的输出模式；

这里，我们就以 VGA 模式为例子，即 OV7725 输出的图像分辨率为 640*480，下面分析一下图 46.1.3.1 所示的帧时序图。

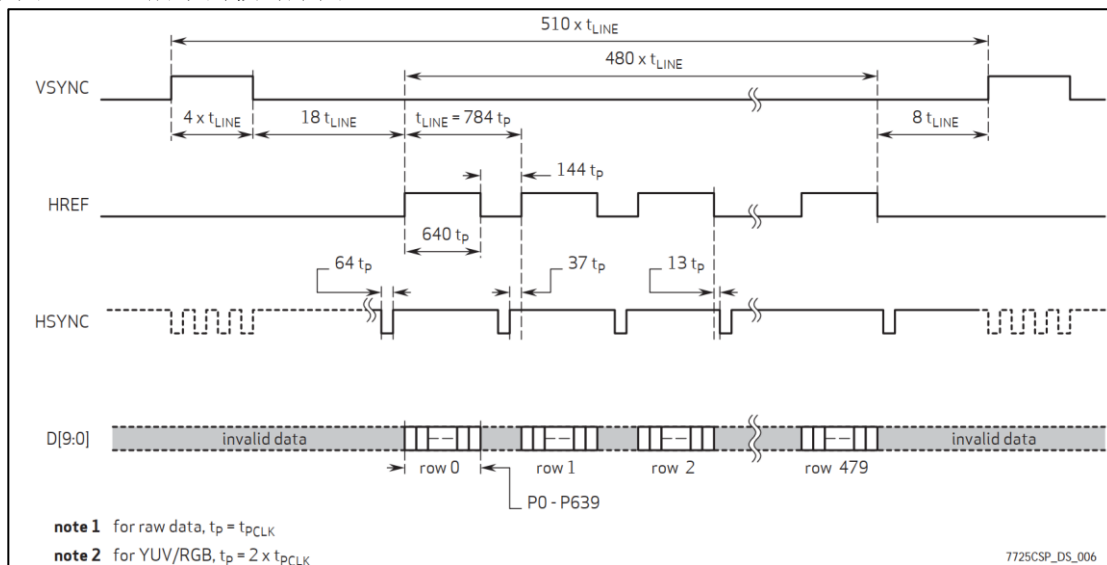


图 46.1.3.1 VGA 模式帧时序图

在分析时序图前，我们先了解几个基本的概念。

VSYNC: 场同步信号，也叫帧信号，由摄像头输出，用于标志一帧图像数据的开始与结束。上图中 VSYNC 的高电平作为一帧的同步信号，在低电平时输出的数据有效。场同步的极性可

以通过寄存器 0x15 去设置的，本实验使用的是和上图一致的默认设置。

HREF/HSYNC：行同步信号，由摄像头输出，用来标志一行数据的开始与结束。上图中的 HREF 和 HSYNC 是由同一引脚输出的，只是数据的同步方式不一样。HREF 上升沿就马上输出图像数据，而 HSYNC 会等待一段时间再输出图像数据，如果行中断里需要处理事情再开始采集，显然用 HREF 上升沿是不容易采集到第一个像素数据的。而我们本实验使用的是 HREF 格式输出，当 HREF 为高电平时，图像数据马上输出并有效。该引脚的极性也是可以通过寄存器 0x15 进行设置。

D[9:0]：数据信号，由摄像头输出，在 RGB 格式输出中，只用到 8 个数据引脚，即高 8 位 D[9:2]是有效的。

tPCLK：一个像素时钟周期。

tp：单个数据周期，这里我们需要注意上图中左下角的 Note1 和 2 描述，在 RGB 模式中，tp 代表两个 tPCLK(像素时钟)。以 RGB565 数据格式为例，RGB565 采用 16bit 数据表示一个像素点，而 OV7725 在一个像素周期(tPCLK)内只能传输 8bit 数据，因此需要两个时钟周期才能输出一个 RGB565 数据。

tLine：摄像头输出一行数据的时间，共 784 个 tp，包含 640tp 个高电平和 144tp 个低电平，其中，640tp 为有效像素数据输出的时间。以 RGB565 数据格式为例，640tp 实际上就是 $640 * 2 = 1280$ 个 tPCLK。

由图 46.1.3.1 可知，VSYNC 的上升沿作为一帧的开始，高电平同步脉冲时间为 $4 * tLine$ ，紧接着等待 $18 * tLine$ 时间后，HREF 开始拉高，此时 OV7725 输出一行有效图像数据，这里是一行数据即 640 个像素点（VGA 模式）；HREF 由 640tp 个高电平和 144tp 个低电平构成；输出 480 行数据之后等待 $8 * tLine$ 时间会产生一个 VSYNC 上升沿标志一帧数据传输结束。所以输出一帧图像的时间实际上是 $tFrame = (4 + 18 + 480 + 8) * tLine = 510 tLine$ 。

利用以上的公式，结合摄像头的输出时钟 fPCLK 得到 tPCLK，便可算出摄像头输出帧率：

摄像头输出帧率： $1s / tFrame = 1s / (510 * 784 * 2 tPCLK)$

OV7725 模块的输入时钟为 12MHz，通过 OV7725 初始化的配置（主要查看 0x0D 和 0x11 寄存器），输出时钟为 24MHz（周期为 42ns），所以代入以上公式，帧率达到 30Hz。这里要跟 LCD 的刷新率进行区分，由于我们本实验用到的是带 FIFO 的 OV7725 模块，MCU 不是直接去接收传感器输出的图像数据，而是通过从 FIFO 里进行获得，所以说这个刷新率是比摄像头的输出帧率要低很多。

从帧时序图中，可以清楚知道 OV7725 是如何输出图像数据，但是不清楚像素数据的情况，所以接下来，我们来看一下图 46.1.3.2 所示的输出 RGB565 时序图。

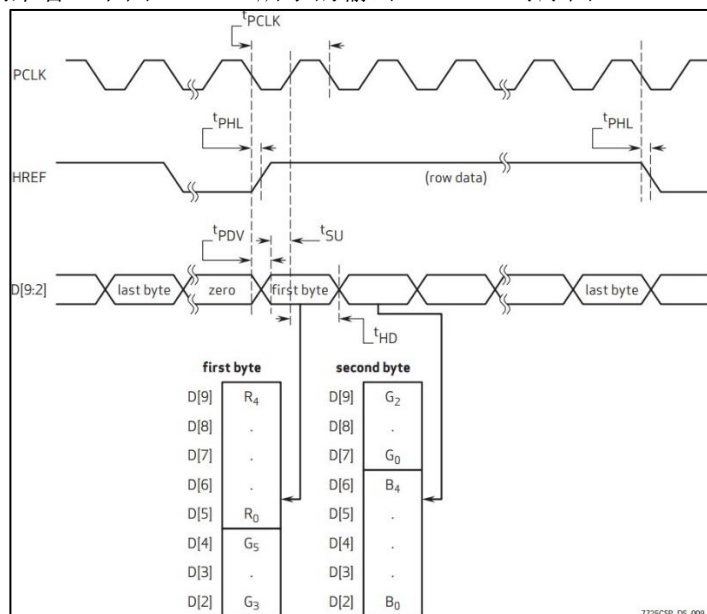


图 46.1.3.2 OV7725 RGB565 输出时序

从上图可看出，OV7725 的图像数据通过 D[9:2]输出一个字节，first byte 和 second byte 组

695

FIFO_WEN 引脚, WE 引脚就输出低电平, 允许图像数据写入到 FIFO。而 DI7~0 是直接和 OV7725 传感器的数据引脚 D2~D9 相连。

总的来说, 写 FIFO 时序就是: OV7725 输出有效的行图像数据时 (HREF 高电平), 我们需要保持写使能引脚为低电平 (FIFO_WEN 拉高)。在复位写指针 (拉低后又重新拉高) 后, 需要一定的复位周期, 然后才开始往 FIFO 的 0 地址处写数据, 且数据会按地址递增方式存入 FIFO。

通常, 我们会根据帧同步信号进而以上操作, 这个存储图像数据的工程为:

- 1) 等待 OV7725 帧同步信号;
- 2) FIFO 写指针复位;
- 3) FIFO 写使能;
- 4) 等待第二个 OV7725 帧同步信号;
- 5) FIFO 写禁止。

通过以上 5 个步骤, 我们就可以完成 1 帧图像数据在 AL422B 的存储。注意: FIFO 写禁止操作不是必须的, 只有当你想将一帧图片数据存储在 FIFO, 并在外部 MCU 读取完这帧图片数据之前, 不再采集新的图片数据的时候, 才需要进行 FIFO 写禁止。

接下来, 继续看图 46.1.4.2 所示的读 FIFO 时序图, 了解一下如何读取图像数据。

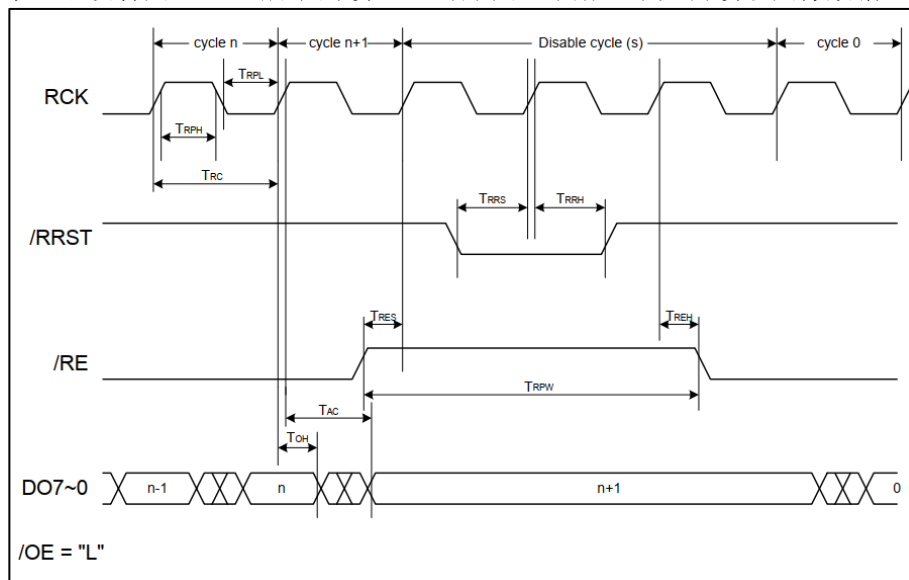


图 46.1.4.2 读 FIFO 时序图

上图中的 RCK 是读 FIFO 时钟, 由 MCU 控制; RRST 是 FIFO 读指针复位引脚, 由 MCU 控制, 低电平时, 读指针会复位到 FIFO 的 0 地址处; RE 是读使能引脚, 硬件设计直接接地; OE 是输出使能, 由 MCU 控制, 要保持低电平才能使能 FIFO 数据输出; DO7~0 是数据引脚, 获取图像数据。

在存储完一帧图像以后, 我们就可以开始读取图像数据了。读取过程如下:

- 1) FIFO 读指针复位 (RRST);
- 2) 给 FIFO 读时钟 (FIFO_RCLK);
- 3) 读取第一个像素高字节 (D[7:0]);
- 4) 给 FIFO 读时钟 (FIFO_RCLK);
- 5) 读取第一个像素低字节 (D[7:0]);
- 6) 给 FIFO 读时钟 (FIFO_RCLK);
- 7) 读取第二个像素高字节 (D[7:0]);
- 8) 循环读取剩余像素;
- 9) 结束。

可以看出, 摄像头模块数据的读取也是十分简单, 比如 QVGA 模式, RGB565 格式, 我们总共循环读取 $320 \times 240 \times 2$ 次, 就可以读取 1 帧图像数据, 把这些数据写入 LCD 模块, 我们就可

以看到摄像头捕捉到的画面了。

注意：如果摄像头要使用 VGA 模式输出，由于 FIFO 是没办法缓存一帧的 VGA 图像，就需要在 FIFO 写满之前开始读 FIFO 数据，否则数据可能被覆盖。OV7725 还可以对输出图像进行各种设置，数据手册和应用笔记详见光盘《OV7725_datasheet.pdf》和《OV7725 Software Application Note.pdf》。对 AL422B 的操作时序，请大家参考 AL422B 的数据手册。以上资料，大家可以通过正点原子的资料下载中心获取《正点原子 OV7725 摄像头模块资料》。

46.2 硬件设计

1. 例程功能

开机后，检测和初始化 OV7725 摄像头模块。初始化成功后需要先通过 KEY0 和 KEY1 选择为 QVGA 或 VGA 输出模式，然后 LCD 才会显示拍摄到的画面。

正常显示拍摄画面后，我们可以通过 KEY0 设置光照模式、通过 KEY1 设置色饱和度，通过 KEY2 设置亮度，通过 KEY_UP 设置对比度，通过 TPAD 设置特效（总共 7 种特效）。通过串口，我们可以查看当前的帧率（这里是指 LCD 显示的帧率，而不是指 OV7725 的输出帧率），同时可以借助 USART 设置 OV7725 的寄存器，方便大家调试。LED0 指示程序运行状态。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) 独立按键: KEY0 - PE4、KEY1 - PE3、KEY2 - PE2、WK_UP - PA0

5) 电容按键: PA1，用于控制触摸按键 TPAD。

6) 外部中断 8(连接 PA8，用于检测 OV7725 的帧信号)

7) 定时器 6(用于打印摄像头帧率等信息)

8) 正点原子 OV7725 摄像头模块，通讯线的连接关系为：

| OV7725 模块 | STM32 开发板 | OV7725 模块 | STM32 开发板 |
|-----------|-----------|-----------|-----------|
| OV_D0~D7 | PC0~7 | FIFO_OE | PG15 |
| OV_SCL | PD3 | FIFO_WRST | PD6 |
| OV_SDA | PG13 | FIFO_WEN | PB3 |
| OV_VSYNC | PA8 | FIFO_RCLK | PB4 |
| FIFO_RRST | PG14 | | |

表 46.2.1 OV7725 模块与开发板连接关系

这部分的连线，模块与开发板上的 P6 座子已经对应好了，安装时直接把模块镜头背离开板的方向安装即可(建议安装所有模块的时候断电安装)。



图 46.2.1 开发板上连接 OV7725 模块的座子

46.3 程序设计

OV7725 模块驱动步骤

1) 初始化 OV7725

这里的初始化工作包括：初始化用到的 IO 口以及 SCCB 接口、读取传感器 ID 以及执行初始化序列（配置参数）。

2) 存储图像数据

依照 OV7725 帧时序和 FIFO 写时序进行操作，详细过程参考前面对时序的分析。

3) 读取图像数据

依照 FIFO 读时序进行，详细过程参考前面对时序的分析。

46.3.1 程序流程图

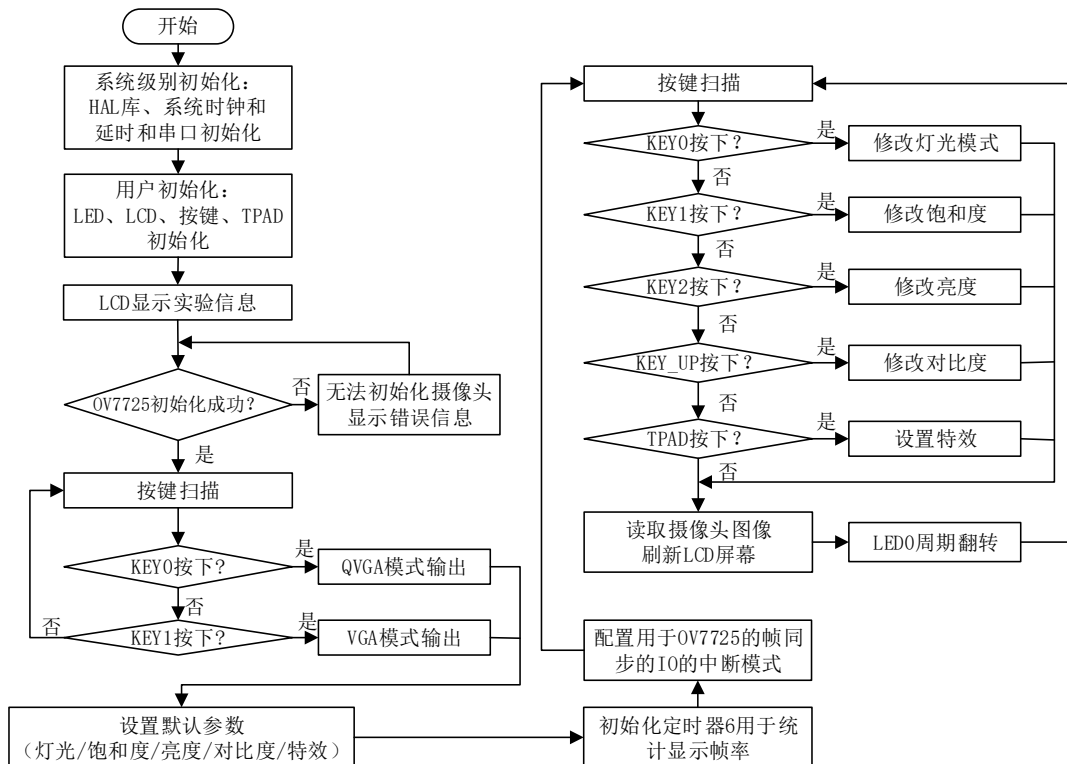


图 46.3.1.1 摄像头实验程序流程图

46.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SCCB 驱动源码包括两个文件：sccb.c 和 sccb.h。OV7725 驱动源码包括五个文件：ov7725.c、ov7725.h、ov7725cfg.h、exti.c 和 exti.h。

1. SCCB 驱动代码

我们已经介绍过 OV7725 是通过 SCCB 协议进行驱动的了，所以在实现 OV7725 驱动前，需要先加定义好 SCCB 对应 IO 的初始化、读写函数，这部分代码我们在 sccb.c/h 中实现。由于 sccb 与 I2C 的实现类似，但是根据 sccb 协议在数据收/发周期时必须实现第九位的传输。

我们直接复制 myiic.c/h 的代码，把它们改成 sccb 驱动，除了读写函数存在一点区别外，其他基本上没有太大的变化。这里看一下读写函数，代码如下：

```

/**
 * @brief      SCCB 发送一个字节
 * @param      data: 要发送的数据
 * @retval     无
 */
uint8_t sccb_send_byte(uint8_t data)
{
    uint8_t t, res;

    for (t = 0; t < 8; t++)

```

```

{
    SCCB_SDA((data & 0x80) >> 7); /* 高位先发送 */
    sccb_delay();
    SCCB_SCL(1);
    sccb_delay();
    SCCB_SCL(0);
    data <<= 1; /* 左移 1 位,用于下一次发送 */
}

SCCB_SDA(1); /* 发送完成, 主机释放 SDA 线 */
sccb_delay();
SCCB_SCL(1); /* 接收第九位, 以判断是否发送成功 */
sccb_delay();

if (SCCB_READ_SDA)
{
    res = 1; /* SDA=1 发送失败, 返回 1 */
}
else
{
    res = 0; /* SDA=0 发送成功, 返回 0 */
}

SCCB_SCL(0);
return res;
}

/**
 * @brief      SCCB 读取一个字节
 * @param      无
 * @retval     读取到的数据
 */
uint8_t sccb_read_byte(void)
{
    uint8_t i, receive = 0;

    for (i = 0; i < 8; i++) /* 接收 1 个字节数据 */
    {
        receive <<= 1; /* 高位先输出, 所以先收到的数据位要左移 */
        SCCB_SCL(1);
        sccb_delay();

        if (SCCB_READ_SDA)
        {
            receive++;
        }

        SCCB_SCL(0);
        sccb_delay();
    }
    return receive;
}

```

2. OV7725 驱动代码

OV7725 驱动代码实现对 OV7725 摄像头模块的操作。

首先看一下基于 SCCB 基本接口的 OV7725 读寄存器函数和写寄存器函数，代码如下：

```

/**
 * @brief      OV7725 读寄存器
 * @param      reg : 寄存器地址
 * @retval     读到的寄存器值
 */
uint8_t ov7725_read_reg(uint16_t reg)
{

```

```
uint8_t data = 0;

sccb_start(); /* 起始信号 */
sccb_send_byte(OV7725_ADDR); /* 写通讯地址 */
sccb_send_byte(reg); /* 寄存器地址 */
sccb_stop(); /* 停止信号 */

/* 设置寄存器地址后，才是读 */
sccb_start(); /* 起始信号 */
sccb_send_byte(OV7725_ADDR | 0X01); /* 读通讯地址 */
data = sccb_read_byte(); /* 读取数据 */
sccb_nack(); /* 非应答信号 */
sccb_stop(); /* 停止信号 */

return data;
}

/**
 * @brief      OV7725 写寄存器
 * @param      reg : 寄存器地址
 * @param      data: 要写入寄存器的值
 * @retval     0, 成功; 1, 失败;
 */
uint8_t ov7725_write_reg(uint8_t reg, uint8_t data)
{
    uint8_t res = 0;

    sccb_start(); /* 起始信号 */

    if (sccb_send_byte(OV7725_ADDR)) res = 1; /* 写通讯地址 */

    if (sccb_send_byte(reg)) res = 1; /* 寄存器地址 */

    if (sccb_send_byte(data)) res = 1; /* 写数据 */

    sccb_stop(); /* 停止信号 */

    return res;
}
```

按厂商建议的初始化序列，我们封装了需要进行初始化的寄存器，我们再结合 AL422B 的读写特性操作相关 IO。实现的初始化函数如下：

```
/**
 * @brief      初始化 OV7725
 * @param      无
 * @retval     0, 成功; 1, 失败;
 */
uint8_t ov7725_init(void)
{
    uint16_t i = 0;
    uint16_t reg = 0;

    /* .....这里删去 IO 和时钟初始化部分代码..... */

    __HAL_RCC_AFIO_CLK_ENABLE();
    /* 禁止 JTAG, 使能 SWD, 释放 PB3, PB4 两个引脚做普通 IO 用 */
    __HAL_AFIO_REMAP_SWJ_NOJTAG();

    OV7725_WRST(1); /* WRST = 1 */
    OV7725_RRST(1); /* RRST = 1 */
    OV7725_OE(1); /* OE = 1 */
    OV7725_RCLK(1); /* RCLK = 1 */
    OV7725_WEN(1); /* WEN = 1 */
}
```

```

sccb_init();          /* 初始化 SCCB 的 IO 口 */

if (ov7725_write_reg(0x12, 0x80)) /* 软件复位 */
{
    return 1;
}

delay_ms(50);
reg = ov7725_read_reg(0x1c); /* 读取厂家 ID 高八位 */
reg <= 8;
reg |= ov7725_read_reg(0x1d); /* 读取厂家 ID 低八位 */

if ((reg != OV7725_MID) && (reg != OV7725_MID1)) /* MID 不正确 ? */
{
    printf("MID:%d\r\n", reg);
    return 1;
}

reg = ov7725_read_reg(0x0a); /* 读取厂家 ID 高八位 */
reg <= 8;
reg |= ov7725_read_reg(0x0b); /* 读取厂家 ID 低八位 */

if (reg != OV7725_PID) /* PID 不正确 ? */
{
    printf("HID:%d\r\n", reg);
    return 2;
}

/* 初始化 ov7725,采用 QVGA 分辨率(320*240) */
for (i=0; i<sizeof(ov7725_init_reg_tbl)/sizeof(ov7725_init_reg_tbl[0]); i++)
{
    ov7725_write_reg(ov7725_init_reg_tbl[i][0], ov7725_init_reg_tbl[i][1]);
}

return 0; /* ok */
}

```

通过 ov7725_init 函数就完成了 OV7725 的基本配置,OV7725 就会以配置的 QVGA 模式输出图像数据。每当 OV7725 输出一个帧信号 VSYNC,就代表一帧图像数据就要通过数据引脚进行输出了,因此我们可以利用 STM32 的外部中断来捕获这个信号,进而在中断服务函数里,把图像数据写入到 FIFO 中。这里过程的实现,详看 exti.c 新添加的函数 exti_ov7725_vsync_init 和中断服务函数,源码如下:

```

/**
 * @brief      OV7725 VSYNC 外部中断初始化程序
 * @param      无
 * @retval     无
 */
void exti_ov7725_vsync_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    gpio_init_struct.Pin = OV7725_VSYNC_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_RISING; /* 上升沿触发 */
    HAL_GPIO_Init(OV7725_VSYNC_GPIO_PORT, &gpio_init_struct);

    HAL_NVIC_SetPriority(OV7725_VSYNC_INT_IRQn, 0, 0); /* 抢占 0, 子优先级 0 */
    HAL_NVIC_EnableIRQ(OV7725_VSYNC_INT_IRQn); /* 使能中断线 8 */
}

/**
 * @brief      OV7725 VSYNC 外部中断服务程序
 * @param      无

```

```

* @retval    无
*/
void OV7725_VSYNC_INT_IRQHandler(void)
{
    /* 是 OV7725_VSYNC_GPIO_PIN 线的中断? */
    if (__HAL_GPIO_EXTI_GET_IT(OV7725_VSYNC_GPIO_PIN))
    {
        if (g_ov7725_vsta == 0) /* 上一帧数据已经处理了? */
        {
            OV7725_WRST(0);      /* 复位写指针 */
            OV7725_WRST(1);      /* 结束复位 */
            OV7725_WEN(1);       /* 允许写入 FIFO */
            g_ov7725_vsta = 1;    /* 标记帧中断 */
        }
        else
        {
            OV7725_WEN(0);       /* 禁止写入 FIFO */
        }
    }
    /* 清除 OV7725_VSYNC_GPIO_PIN 上的中断标志位 */
    __HAL_GPIO_EXTI_CLEAR_IT(OV7725_VSYNC_GPIO_PIN);
}

```

因为 OV7725 的帧同步信号(OV_VSYNC)接在 PA8,所以用到的是 EXTI9_5_IRQHandler。在中断服务函数中,需要先判断中断是不是来自中断线 8 再进行处理。

中断处理的部分流程:每当帧中断到来后,先判断 g_ov7725_vsta 的值是否为 0,如果是 0,说明可以往 FIFO 里面写入数据,执行复位 FIFO 写指针,并允许 FIFO 写入。此时,AL422B 将从地址 0 开始,存储新一帧的图像数据。然后设置 g_ov7725_vsta 为 1,标记新的一帧数据正在存储中。如果 g_ov7725_vsta 不为 0,说明之前存储在 FIFO 里面的一帧数据还未被读取过,直接禁止 FIFO 写入,等待 MCU 读取 FIFO 数据,以免数据覆盖。

然而,STM32 只需要判断 g_ov7725_vsta 是否为 1,来读取 FIFO 里面的数据,读完一帧后,设置 g_ov7725_vsta 为 0,以免重复读取,同时还可以使能 FIFO 新一帧数据的写入。

2. main.c 代码

我们在实现 main 函数前,定义了一个 ov7725_camera_refresh()函数,用于读取摄像头模块自带 FIFO 里面的数据并显示在 LCD:

```

uint16_t g_ov7725_wwidth = 320; /* 默认窗口宽度为 320 */
uint16_t g_ov7725_wheight = 240; /* 默认窗口高度为 240 */
/**
 * @brief      更新 LCD 显示
 * @note      该函数将 OV7725 模块 FIFO 里面的数据拷贝到 LCD 屏幕上
 * @param      无
 * @retval     无
 */
void ov7725_camera_refresh(void)
{
    uint32_t i, j;
    uint16_t color;

    if (g_ov7725_vsta) /* 有帧中断更新 */
    {
        lcd_scan_dir(U2D_L2R); /* 从上到下,从左到右 */
        lcd_set_window((lcddev.width - g_ov7725_wwidth) / 2,
                       (lcddev.height - g_ov7725_wheight) / 2, g_ov7725_wwidth,
                       g_ov7725_wheight); /* 将显示区域设置到屏幕中央 */
        lcd_write_ram_prepare(); /* 开始写入 GRAM */

        OV7725_RRST(0); /* 开始复位读指针 */
        OV7725_RCLK(0);
        OV7725_RCLK(1);
        OV7725_RCLK(0);
    }
}

```

```

OV7725_RRST(1);          /* 复位读指针结束 */
OV7725_RCLK(1);

for (i = 0; i < g_ov7725_wheight; i++)
{
    for (j = 0; j < g_ov7725_wwidth; j++)
    {
        OV7725_RCLK(0);
        color = OV7725_DATA; /* 读数据 */
        OV7725_RCLK(1);
        color <<= 8;
        OV7725_RCLK(0);
        color |= OV7725_DATA; /* 读数据 */
        OV7725_RCLK(1);
        LCD->LCD_RAM = color;
    }
}

g_ov7725_vsta = 0;          /* 清零帧中断标记 */
g_ov7725_frame++;
lcd_scan_dir(DFT_SCAN_DIR); /* 恢复默认扫描方向 */
}
}

```

对于 OV7725，我们可以通过 g_ov7725_wwidth 和 g_ov7725_wheight 两个全局变量设置图像窗口输出的大小。对于分辨率大于 320*240 的屏幕，则通过开窗函数（lcd_set_window）将显示区域开窗在屏幕的正中央。注意，为了提高 FIFO 读取速度，我们将 OV7725_RCLK 的控制，采用快速 IO 控制，关键代码如下（在 ov7725.h 里面）：

```

#define OV7725_RCLK(x) x ? (OV7725_RCLK_GPIO_PORT->BSRR = OV7725_RCLK_GPIO_PIN) : \
(OV7725_RCLK_GPIO_PORT->BRR = OV7725_RCLK_GPIO_PIN)

```

控制 OV7725_RCLK 输出高电平或者低电平就用到 BSRR 和 BRR 两个寄存器，以实现快速 IO 设置，从而提高读取速度。

最后介绍的是 main 函数，其定义如下：

```

const char *LMODE_TBL[6] = {"Auto", "Sunny", "Cloudy", "Office",
                             "Home", "Night"};          /* 6 种光照模式 */
const char *EFFECTS_TBL[7] = {"Normal", "Negative", "B&W", "Redish",
                               "Greenish", "Bluish", "Antique"}; /* 7 种特效 */

int main(void)
{
    uint8_t key;
    uint8_t i = 0;
    char msgbuf[15];          /* 消息缓存区 */
    uint8_t tm = 0;
    uint8_t lightmode = 0, effect = 0;
    uint8_t saturation = 4, brightness = 4, contrast = 4;

    HAL_Init();              /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);          /* 延时初始化 */
    usart_init(115200);      /* 串口初始化为 115200 */
    usmart_dev.init(72);     /* 初始化 USMART */
    led_init();              /* 初始化 LED */
    lcd_init();              /* 初始化 LCD */
    key_init();              /* 初始化按键 */
    tpad_init(6);            /* TPAD 初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "OV7725 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Light Mode", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1:Saturation", RED);
    lcd_show_string(30, 150, 200, 16, 16, "KEY2:Brightness", RED);
}

```



```

lcd_show_string(30, 170, 200, 16, 16, "KEY_UP:Contrast", RED);
lcd_show_string(30, 190, 200, 16, 16, "TPAD:Effects", RED);
lcd_show_string(30, 210, 200, 16, 16, "OV7725 Init...", RED);

while (1)
{
    if (ov7725_init() == 0)                /* 初始化 ov7725 */
    {
        lcd_show_string(30, 210, 200, 16, 16, "OV7725 Init OK", RED);

        while (1)
        {
            key = key_scan(0);

            if (key == KEY0_PRES)
            {
                /* QVGA 模式输出 */
                g_ov7725_wwidth = 320;      /* 默认窗口宽度为 320 */
                g_ov7725_wheight = 240;    /* 默认窗口高度为 240 */
                ov7725_window_set(g_ov7725_wwidth, g_ov7725_wheight, 0);
                break;
            }
            else if (key == KEY1_PRES)
            {
                /* VGA 模式输出 */
                g_ov7725_wwidth = 320;      /* 默认窗口宽度为 320 */
                g_ov7725_wheight = 240;    /* 默认窗口高度为 240 */
                ov7725_window_set(g_ov7725_wwidth, g_ov7725_wheight, 1);
                break;
            }

            i++;

            if (i == 100)
                lcd_show_string(30, 230, 210, 16, 16, "KEY0:QVGA KEY1:VGA", RED);

            if (i == 200)
            {
                lcd_fill(30, 230, 210, 250 + 16, WHITE);
                i = 0;
            }

            delay_ms(5);
        }

        ov7725_light_mode(lightmode);
        ov7725_color_saturation(saturation);
        ov7725_brightness(brightness);
        ov7725_contrast(contrast);
        ov7725_special_effects(effect);

        OV7725_OE(0); /* 使能 OV7725 FIFO 数据输出 */

        break;
    }
    else
    {
        lcd_show_string(30, 190, 200, 16, 16, "OV7725 Error!!", RED);
        delay_ms(200);
        lcd_fill(30, 190, 239, 246, WHITE);
        delay_ms(200);
    }
}

btim_timx_int_init(10000, 7200 - 1);      /* 10Khz 计数频率, 1 秒钟中断 */
exti_ov7725_vsync_init();                 /* 使能 OV7725 VSYNC 外部中断 */
lcd_clear(BLACK);

```

```

while (1)
{
    key = key_scan(0);          /* 不支持连按 */

    if (key)
    {
        tm = 20;

        switch (key)
        {
            case KEY0_PRES:      /* 灯光模式 Light Mode */
                lightmode++;
                if (lightmode > 5) lightmode = 0;
                ov7725_light_mode(lightmode);
                sprintf((char *)msgbuf, "%s", LMODE_TBL[lightmode]);
                break;

            case KEY1_PRES:      /* 饱和度 Saturation */
                saturation++;
                if (saturation > 8) saturation = 0;
                ov7725_color_saturation(saturation);
                sprintf((char *)msgbuf, "Saturation:%d", saturation);
                break;

            case KEY2_PRES:      /* 饱和度 Saturation */
                brightness++;
                if (brightness > 8) brightness = 0;
                ov7725_brightness(brightness);
                sprintf((char *)msgbuf, "Brightness:%d", brightness);
                break;

            case WKUP_PRES:      /* 对比度 Contrast */
                contrast++;
                if (contrast > 8) contrast = 0;
                ov7725_contrast(contrast);
                sprintf((char *)msgbuf, "Contrast:%d", contrast);
                break;
        }
    }

    if (tpad_scan(0))          /* 检测到触摸按键 */
    {
        effect++;

        if (effect > 6) effect = 0;

        ov7725_special_effects(effect); /* 设置特效 */
        sprintf((char *)msgbuf, "%s", EFFECTS_TBL[effect]);
        tm = 20;
    }

    ov7725_camera_refresh(); /* 更新显示 */

    if (tm)
    {
        lcd_show_string((lcddev.width - 240) / 2 + 30,
                        (lcddev.height-320)/2 + 60, 200, 16, 16, msgbuf, BLUE);
        tm--;
    }

    i++;

    if (i >= 15)
    {

```

```
i = 0;
LED0_TOGGLE(); /* LED0 闪烁 */
}
}
```

main 函数的具体流程可以参考程序流程图，代码比较简单，这里我们就不展开说明了。

到此摄像头的使用过程我们讲解完了，大家可以参考光盘中的源码进行测试和修改，也可以在 USMART 中加入 OV7725 的测试接口 `ov7725_write_reg` 和 `ov7725_read_reg`，轻松调试摄像头。

最后，为了得到最快的显示速度，我们可以把 MDK 的代码优化等级设置为 -O2 级别（在 C/C++ 选项卡设置），这样 OV7725 的显示帧率可达 23 帧。注意：因为 `tpad_scan` 扫描会占用比较多的时间，所以帧率比较慢，屏蔽该函数，也可以提高帧率。

46.4 下载验证

在代码编译成功之后，下载代码到正点原子战舰 STM32F103 开发板上，得到如图 46.4.1 所示界面：

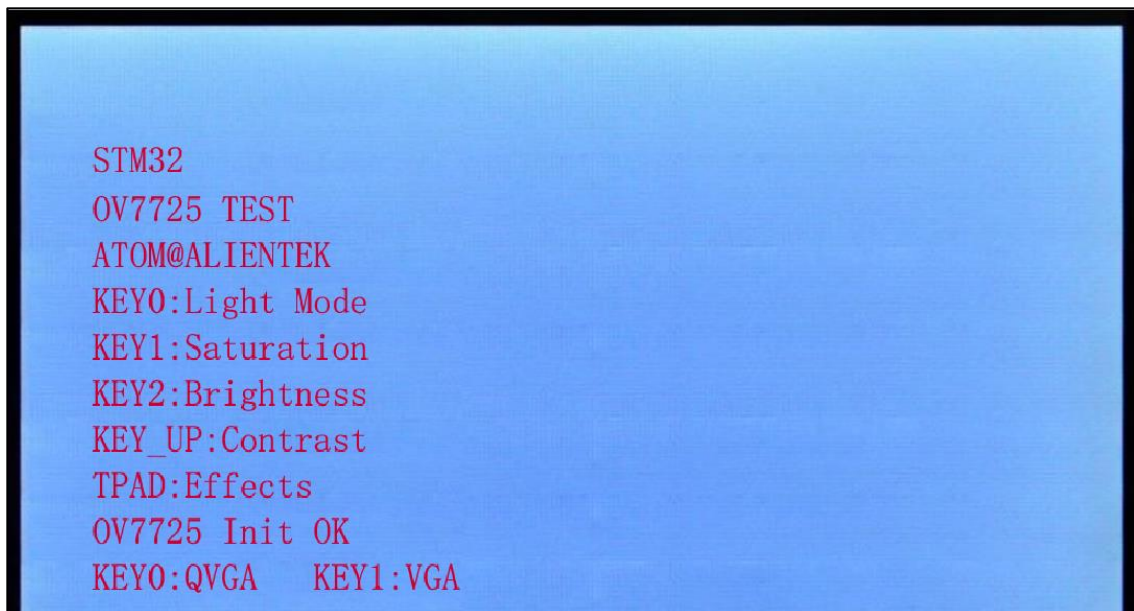


图 46.4.1 程序运行效果图

随后，我们通过 KEY0 和 KEY1 选择模式。当选择 QVGA 模式时，OV7725 直接输出 320*240 分辨率图像数据，该模式相对 VGA 模式视角较广但画面没有那么清晰细腻。当选择 VGA 模式时，实质是将 640*480 窗口截取中间 320*240 的图像输出，该模式拍出的图像较为清晰细腻但视角较小。

我们可以按不同的按键（KEY0~KEY2、KEY_UP、TPAD 等），来设置摄像头的相关参数和模式，得到不同的成像效果。同时，我们可以通过 USMART 调用 `ov7725_write_reg` 等函数，来设置 OV7725 的各寄存器等，达到调试测试的目的，具体如图 46.4.2 所示：

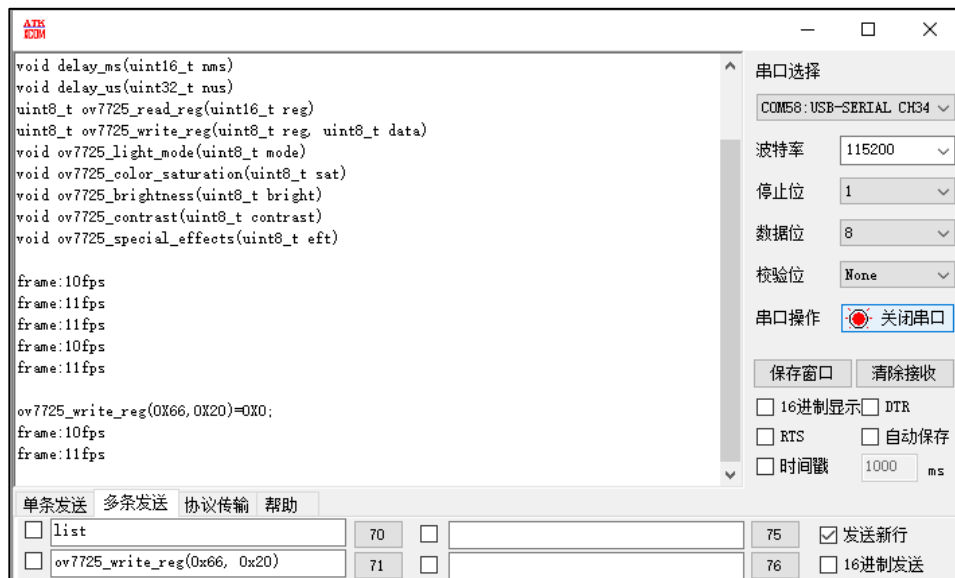


图 46.4.2 USART 调试 OV7725

从上图还可以看出，LCD 显示帧率为 10 帧左右（没有代码优化），而实际上 OV7725 的输出速度是 30 帧。图中，我们是可以通过 USART 发送 `ov7725_write_reg(0x66, 0x20)` 设置 OV7725 输出彩条，方便测试。

摄像头的实验我们就讲解到这里。

第四十七章 SRAM 实验

STM32F103ZET6 自带了 64K 字节的 RAM，对一般应用来说，已经足够了，不过在一些对内存要求高的场合，比如做华丽效果的 GUI，处理大量数据的应用等，STM32 自带的这些内存就可能不太够用了。好在嵌入式方案提供了扩展芯片 RAM 的方法，本章将介绍我们开发板上使用的 RAM 拓展方案：使用 SRAM 芯片，并驱动这个外部 SRAM 提供程序需要的一部分 RAM 空间，对其进行读写测试。

本章分为如下几个部分：

- 47.1 存储器简介
- 47.2 硬件设计
- 47.3 程序设计
- 47.4 下载验证

47.1 存储器简介

使用电脑时，我们会提到内存和内存条的概念，电脑维修的朋友有时候会说加个内存条电脑就不卡了。实际上对于 PC 来说一些情况下卡顿就是电脑同时运行的程序太多了，电脑处理速度变慢的现象。而程序是动态加载到内存中的，一种解决方法就是增加电脑的内存来增加同时可处理的程序的数量。对于单片机也是一样的，高性能有时候需要通过增加大内存来获得。内存是存储器的一种，由于微机架构设计了不同的存储器放置不同的数据，所以我们也简单来了解一下存储器。

存储器实际上是时序逻辑电路的一种，用来存放程序和数据信息。构成存储器的存储介质主要采用半导体器件和磁性材料。存储器中最小的存储单位就是一个双稳态半导体电路或一个 CMOS 晶体管或磁性材料的存储元，它可存储一个二进制代码。由若干个存储元组成一个存储单元，然后再由许多存储单元组成一个存储器。按不同的分类方式，存储器可以有表 47.1.1 所示的分类：

| 分类方式 | 类别 | 描述 |
|-------------|--------------|---|
| 按存储介质分类 | 半导体存储器 | 用半导体器件组成的存储器 |
| | 磁表面存储器 | 用磁性材料做成的存储器 |
| 按存储方式分类 | 随机存储器 | 任意存储单元的内容都能被随机存取，且存取时间和存储单元的物理位置无关 |
| | 顺序存储器 | 只能按某种顺序来存取，存取时间与存储单元的物理位置有关 |
| 按存储器的读写功能分类 | 只读存储器(ROM) | ROM 原为只能读而不能写的存储器。现在一般指掉电非易失性半导体存储器，STM32 的内部 Flash。 |
| | 随机读写存储器(RAM) | 通电状态下，能通过地址线在任意地址读/写数据的半导体存储器，读写速度极快。当电源关闭时，存于 RAM 中的数据会丢失。 |
| 按信息的可保存性分类 | 非永久记忆的存储器 | 断电后信息即消失的存储器 |
| | 永久记忆性存储器 | 断电后仍能保存信息的存储器 |

表 47.1.1 存储器的分类

对于上述分类，在我们 STM32 编程学习中我们常常只关心按读写功能分类的 ROM 和 RAM 两种，因为嵌入式程序主要对应到这两种存储器。对于 RAM，目前常见的是 SRAM 和 DRAM，它们因工作方式不同而得名，它们主要有以下的特性，如表 47.1.2 所示：

| | SRAM | DRAM |
|------|--|---|
| 描述 | 静态存储器/Static RAM, 存储单元一般为锁存器, 只要不掉电, 信息就不会丢失 | 动态存储器/Dynamic RAM, 利用 MOS (金属氧化物半导体) 电容存储电荷来储存信息, 保留数据的时间很短, 速度也比 SRAM 慢, 每隔一段时间, 要刷新充电一次, 否则内部的数据即会消失。 |
| 特点 | 存取速度快, 工作稳定, 不需要刷新电路, 集成度不高; 集成度较低且价格较高 | DRAM 的成本、集成度、功耗等明显优于 SRAM |
| 常见应用 | CPU 与主存间的高速缓冲、CPU 内部的一级/二级缓存、外部的高速缓存、SSRAM | DRAM 分为很多种, 按内存技术标准可分为 FPRAM/FastPage、EDO DRAM、SDRAM、DDR/DDR2/DDR3/DDR4/...、RDRAM、SGRAM 以及 WRAM 等。 |

表 47.1.2 SRAM 和 DRAM 特性

在 STM32 上, 我们编译的程序, 编译器一般会根据对应硬件的结构把程序中不同功能的数据段分为 ZIRWRO 这样的数据块, 执行程序时分别放到不同的存储器上, 这部分参考我们《第九章 STM32 启动过程分析》中关于 map 文件的描述。对于我们编写的 STM32 程序中的变量, 在默认配置下是加载到 STM32 的 RAM 区中执行的。而像程序代码和常量等编译后就固定不变的则会放到 ROM 区。

存储器的知识我们就介绍到这里, 限于篇幅只能作简单的引用和介绍, 大家可以查找资料拓展对各种存储器作一下加深了解。

47.2 SRAM 方案简介

RAM 的功能我们已经介绍过了, SRAM 更稳定, 但因为结构更复杂且造价更高, 所以有更大片上 SRAM 的 STM32 芯片造价也更高。而且由于 SRAM 集成度低的原因, MCU 也不会把片上 SRAM 做得特别大, 基于以上原因, 计算机/微机系统中都允许采用外扩 RAM 的方式提高性能。

1. SRAM 芯片介绍

IS62WV51216 方案

IS62WV51216 是 ISSI(Integrated Silicon Solution, Inc)公司生产的一颗 16 位宽 512K(512*16, 即 1M 字节)容量的 CMOS 静态内存芯片。该芯片具有如下几个特点:

- 高速。具有 45ns/55ns 访问速度。
- 低功耗。
- TTL 电平兼容。
- 全静态操作。不需要刷新和时钟电路。
- 三态输出。
- 字节控制功能。支持高/低字节控制。

IS62WV51216 的功能框图如图 47.2.1 所示:

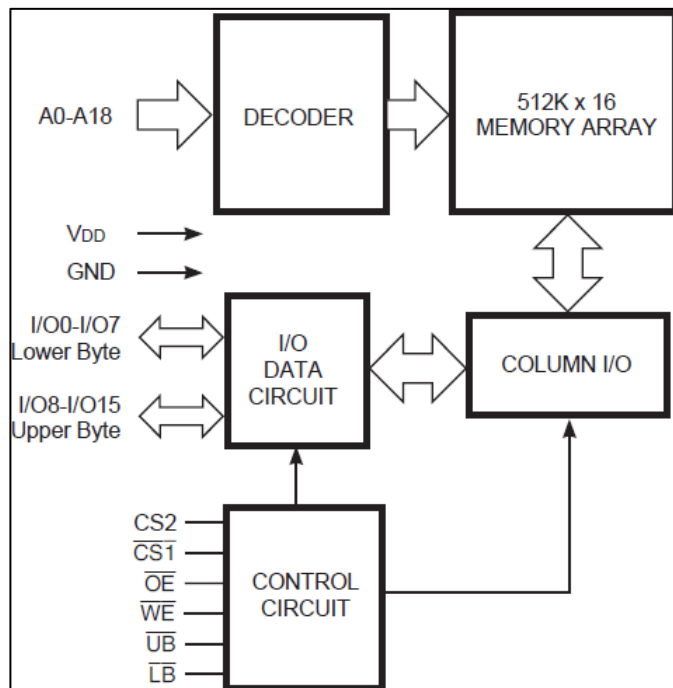


图 47.2.1 IS62WV51216 功能框图

图中 A0~18 为地址线，总共 19 根地址线（即 $2^{19}=512K$ ， $1K=1024$ ）；IO0~15 为数据线，总共 16 根数据线。CS2 和 CS1 都是片选信号，不过 CS2 是高电平有效 CS1 是低电平有效；OE 是输出使能信号（读信号）；WE 为写使能信号；UB 和 LB 分别是高字节控制和低字节控制信号；

XM8A51216 方案

国产替代一直是国内嵌入式领域的一个话题，国产替代的优势一般是货源稳定，售价更低，也有专门研发对某款芯片作 Pin to Pin 兼容的厂家，使用时无需修改 PCB，直接更换元件即可，十分方便。

正点原子开发板目前使用的一款替代 IS62WV51216 的芯片是 XM8A5121，它与 IS62WV51216 一样采用 TSOP44 封装，引脚顺序也与前者完全一致。

XM8A51216 是星忆存储生产的一颗 16 位宽 512K（ 512×16 ，即 1M 位）容量的 CMOS 静态内存芯片。采用异步 SRAM 接口并结合独有的 XRAM 免刷新专利技术，在大容量、高性能和高可靠及品质方面完全可以匹敌同类 SRAM，具有较低功耗和低成本优势，可以与市面上同类型 SRAM 产品硬件完全兼容，并且满足各种应用系统对高性能和低成本的要求，XM8A51216 也可以当做异步 SRAM 使用，该芯片具有如下几个特点：

- 高速。具有最高访问速度 10/12/15ns。
- 低功耗。
- TTL 电平兼容。
- 全静态操作。不需要刷新和时钟电路。
- 三态输出。
- 字节控制功能。支持高/低字节控制。

该芯片与 IS62WV51216 引脚和完全兼容，控制时序也类似，大家可以方便地直接替换。

本章，我们使用 FSMC 的 BANK1 区域 3 来控制 SRAM 芯片，关于 FSMC 的详细介绍，我们在学习 LCD 的章节已经介绍过，我们采用的是读写不同的时序来操作 TFTLCD 模块（因为 TFTLCD 模块读的速度比写的速度慢很多），但是在本章，因为 IS62WV51216/XM8A51216 的读写时间基本一致，所以，我们设置读写相同的时序来访问 FSMC。关于 FSMC 的详细介绍，请大家看“TFT LCD 实验”和《STM32F10xxx 参考手册_V10（中文版）.pdf》。

47.3 硬件设计

1. 例程功能

本章实验功能简介：开机后，显示提示信息，然后按下 KEY0 按键，即测试外部 SRAM 容量大小并显示在 LCD 上。按下 KEY1 按键，即显示预存在外部 SRAM 的数据。LED0 指示程序运行状态。

2. 硬件资源

1) LED 灯

LED0 - PB5

2) 按键：

KEY0: PE4

KEY1: PE3

3) SRAM 芯片：

XM8A51216/IS62WV51216

4) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

5) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

SRAM 芯片与 STM32 的连接关系，如下图所示：

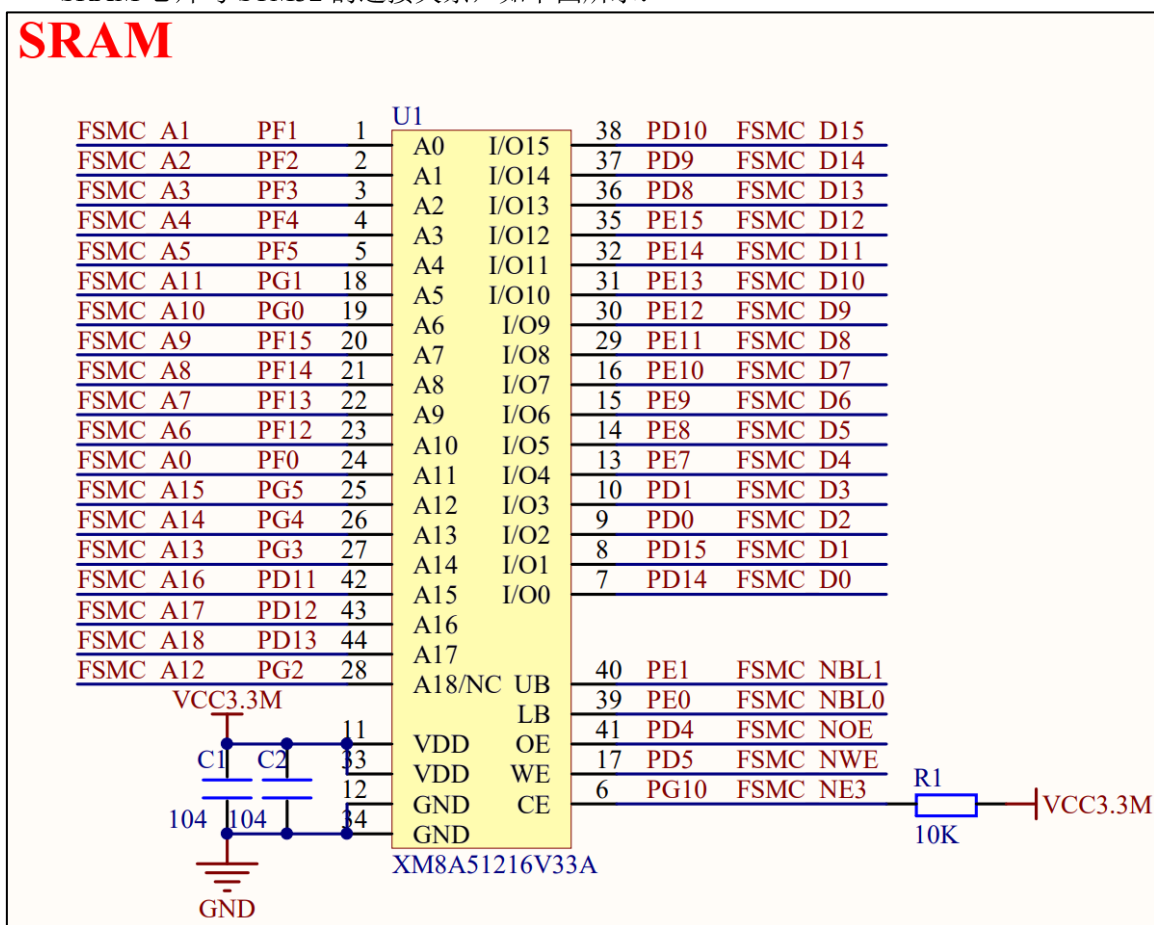


图 47.3.1 STM32 和 SRAM 连接原理图(XM8A51216/IS62WV51216 封装相同)

SRAM 芯片直接是接在 STM32F1 的 FSMC 外设上，具体的引脚连接关系如下表 47.3.1 所示。

| 战舰 | SRAM |
|---------|----------------------------------|
| A[0:18] | FMSC_A[0:18] (为了布线方便交换了部分 IO) |
| D[0:15] | FSMC_D[0:15] |
| UB | FSMC_NBL1 |
| LB | FSMC_NBL0 |
| OE | FSMC_OE |
| WE | FSMC_WE |
| CS | FSMC_NE3 |

表 47.3.1 STM32 和 SRAM 芯片的连接原理图

在上面的连接关系中,SRAM 芯片的 A[0:18]并不是按顺序连接 STM32F1 的 FMSC_A[0:18],这样设计的好处,就是可以方便我们的 PCB 布线。不过这并不影响我们正常使用外部 SRAM,因为地址具有唯一性,只要地址线不和数据线混淆,就可以正常使用外部 SRAM。

47.4 程序设计

操作 SRAM 时要通过多个地址线寻址,然后才可以读写数据,在 STM32 上可以使用 FSMC 来实现,在 TFT_LCD 一节我们已经讲解过 FSMC 接口的驱动,与之前的用法类似,关于 HAL 库部分函数介绍我们这里就不重复了。

使用 SRAM 的配置步骤

1) 使能 FSMC 时钟,并配置 FSMC 相关的 IO 及其时钟使能。

要使用 FSMC,当然首先得开启其时钟。然后需要把 FSMC_D0~15,FSMCA0~18 等相关 IO 口,全部配置为复用输出,并使能各 IO 组的时钟。

使能 FSMC 时钟的方法前面 LCD 实验已经讲解过,方法为:

```
HAL_RCC_FSMC_CLK_ENABLE();
```

配置 IO 口为复用输出的关键行代码为:

```
gpio_init_struct.Mode = GPIO_MODE_AF_PP; /* 复用推挽输出 */
```

2) 设置 FSMC BANK1 区域 3 的相关寄存器。

此部分包括设置区域 3 的存储器的工作模式、位宽和读写时序等。本章我们使用模式 A、16 位宽,读写共用一个时序寄存器。

这个是通过调用函数 HAL_SRAM_Init 来实现的,函数原型为:

```
HAL_StatusTypeDef HAL_SRAM_Init(SRAM_HandleTypeDef *hsram,
FSMC_NORSRAM_TimingTypeDef *Timing, FSMC_NORSRAM_TimingTypeDef *ExtTiming)
```

通过以上几个步骤,我们就完成了 FSMC 的配置,初始化 FSMC 后就可以访问 SRAM 芯片时行读写操作了,这里还需要注意,因为我们使用的是 BANK1 的区域 3,所以 HADDR[27:26]=10,故外部内存的首地址为 0X68000000。

47.4.1 程序流程图

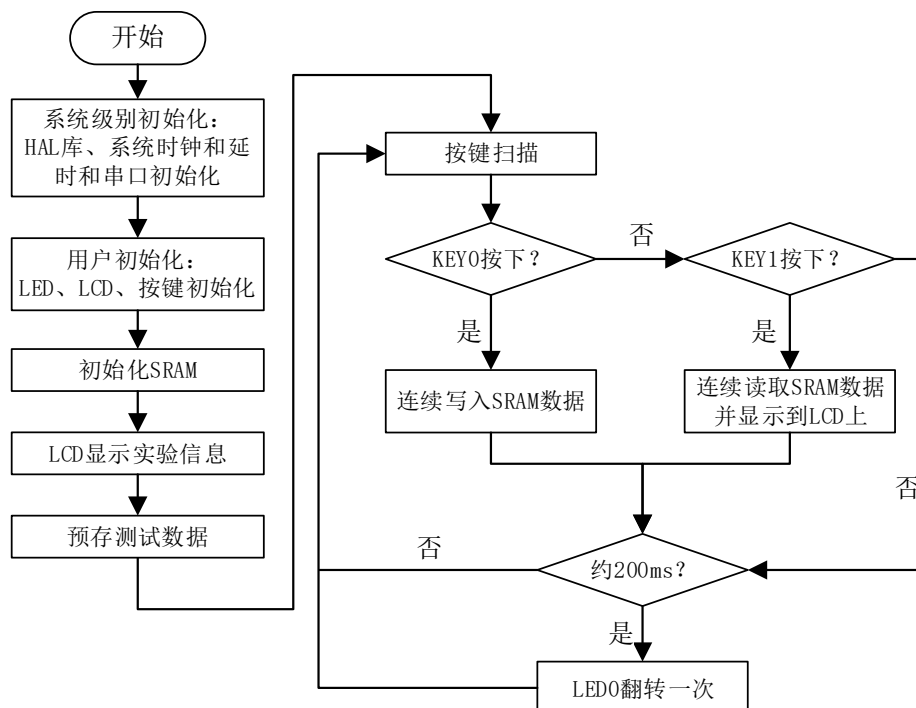


图 47.4.1.1 SRAM 实验程序流程图

47.4.2 程序解析

1. SRAM 驱动

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SRAM 驱动源码包括两个文件：sram.c 和 sram.h。

为方便修改，我们在 sram.h 中使用宏定义 SRAM 的读写控制和片选引脚，它们定义如下：

```

#define SRAM_WR_GPIO_PORT      GPIOD
#define SRAM_WR_GPIO_PIN      GPIO_PIN_5
#define SRAM_WR_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE();}while(0)

#define SRAM_RD_GPIO_PORT      GPIOD
#define SRAM_RD_GPIO_PIN      GPIO_PIN_4
#define SRAM_RD_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

/* SRAM_CS(需要根据 SRAM_FSMC_NEX 设置正确的 IO 口) 引脚 定义 */
#define SRAM_CS_GPIO_PORT      GPIOG
#define SRAM_CS_GPIO_PIN      GPIO_PIN_10
#define SRAM_CS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0)
  
```

根据 STM32F1 参考手册，SRAM 可以选择 FSMC 对应的存储块 1 上的 4 个区域之一作为访问地址，它上面有四块相互独立的 64M 的连续寻址空间，为了能灵活根据不同的计算出使用的地址空间，我们定义了以下的宏：

```

/* FSMC 相关参数 定义
 * 注意：我们默认是通过 FSMC 块 3 来连接 SRAM，块 1 有 4 个片选：FSMC_NE1~4
 *
 * 修改 SRAM_FSMC_NEX，对应的 SRAM_CS_GPIO 相关设置也得改
 */
#define SRAM_FSMC_NEX          3 /* 使用 FSMC_NE3 接 SRAM_CS,取值范围只能是：1~4 */

/*****
/* SRAM 基地址，根据 SRAM_FSMC_NEX 的设置来决定基址地址
 * 我们一般使用 FSMC 的块 1 (BANK1) 来驱动 SRAM，块 1 地址范围总大小为 256MB，均分成 4 块：
  
```

```
* 存储块 1 (FSMC_NE1) 地址范围: 0X6000 0000 ~ 0X63FF FFFF
* 存储块 2 (FSMC_NE2) 地址范围: 0X6400 0000 ~ 0X67FF FFFF
* 存储块 3 (FSMC_NE3) 地址范围: 0X6800 0000 ~ 0X6BFF FFFF
* 存储块 4 (FSMC_NE4) 地址范围: 0X6C00 0000 ~ 0X6FFF FFFF
*/
```

```
#define SRAM_BASE_ADDR (0X60000000 + (0X4000000 * (SRAM_FSMC_NEX - 1)))
```

上述定义 SRAM_FSMC_NEX 的值为 3, 即使用 FSMC 存储块 1 的第 3 个地址范围, 上面的 SRAM_BASE_ADDR 则根据我们使用的存储块计算出 SRAM 空间的首地址, 存储块 3 对应的是 0X68000000 ~ 0X6BFFFFFFF 的地址空间。

sram_init 的类似于 LCD, 我们需要根据原理图配置 SRAM 的控制引脚, 复用连接到 SRAM 芯片上的 IO 作为 FSMC 的地址线, 根据 SRAM 芯片上的进序设置地址线宽度、等待时间、信号极性等, 则 sram 的初始化函数我们编写如下:

```
void sram_init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    FSMC_NORSRAM_TimingTypeDef fsmc_readwritetim;

    SRAM_CS_GPIO_CLK_ENABLE(); /* SRAM_CS 脚时钟使能 */
    SRAM_WR_GPIO_CLK_ENABLE(); /* SRAM_WR 脚时钟使能 */
    SRAM_RD_GPIO_CLK_ENABLE(); /* SRAM_RD 脚时钟使能 */
    __HAL_RCC_FSMC_CLK_ENABLE(); /* 使能 FSMC 时钟 */
    __HAL_RCC_GPIOD_CLK_ENABLE(); /* 使能 GPIOD 时钟 */
    __HAL_RCC_GPIOE_CLK_ENABLE(); /* 使能 GPIOE 时钟 */
    __HAL_RCC_GPIOF_CLK_ENABLE(); /* 使能 GPIOF 时钟 */
    __HAL_RCC_GPIOG_CLK_ENABLE(); /* 使能 GPIOG 时钟 */

    GPIO_InitStructure.Pin = SRAM_CS_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Pull = GPIO_PULLUP;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(SRAM_CS_GPIO_PORT, &GPIO_InitStructure); /* SRAM_CS 引脚模式设置 */

    GPIO_InitStructure.Pin = SRAM_WR_GPIO_PIN;
    HAL_GPIO_Init(SRAM_WR_GPIO_PORT, &GPIO_InitStructure); /* SRAM_WR 引脚模式设置 */

    GPIO_InitStructure.Pin = SRAM_RD_GPIO_PIN;
    HAL_GPIO_Init(SRAM_RD_GPIO_PORT, &GPIO_InitStructure); /* SRAM_CS 引脚模式设置 */
    /* PD0,1,4,5,8~15 */
    GPIO_InitStructure.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_8 | GPIO_PIN_9 |
        GPIO_PIN_10 | GPIO_PIN_11 | GPIO_PIN_12 | GPIO_PIN_13 |
        GPIO_PIN_14 | GPIO_PIN_15;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP; /* 推挽复用 */
    GPIO_InitStructure.Pull = GPIO_PULLUP; /* 上拉 */
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH; /* 高速 */
    HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);

    /* PE0,1,7~15 */
    GPIO_InitStructure.Pin = GPIO_PIN_0 | GPIO_PIN_1 | 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_InitStructure);
    /* PF0~5,12~15 */
    GPIO_InitStructure.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 |
        GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_12 | GPIO_PIN_13 |
        GPIO_PIN_14 | GPIO_PIN_15;
    HAL_GPIO_Init(GPIOF, &GPIO_InitStructure);
    /* PG0~5,10 */
    GPIO_InitStructure.Pin = GPIO_PIN_0 | GPIO_PIN_1 |
        GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5;
    HAL_GPIO_Init(GPIOG, &GPIO_InitStructure);
    g_sram_handler.Instance = FSMC_NORSRAM_DEVICE;
    g_sram_handler.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;
```

```

g_sram_handler.Init.NSBank = (SRAM_FSMC_NEX == 1) ? FSMC_NORSRAM_BANK1 : \
                             (SRAM_FSMC_NEX == 2) ? FSMC_NORSRAM_BANK2 : \
                             (SRAM_FSMC_NEX == 3) ? FSMC_NORSRAM_BANK3 : \
                             FSMC_NORSRAM_BANK4; /* 根据配置选择 FSMC_NE1~4 */

/* 地址/数据线不复用 */
g_sram_handler.Init.DataAddressMux = FSMC_DATA_ADDRESS_MUX_DISABLE;
g_sram_handler.Init.MemoryType = FSMC_MEMORY_TYPE_SRAM; /* SRAM */
/* 16 位数据宽度 */
g_sram_handler.Init.MemoryDataWidth = SMC_NORSRAM_MEM_BUS_WIDTH_16;
/* 是否使能突发访问,仅对同步突发存储器有效,此处未用到 */
g_sram_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 */
g_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 */
g_sram_handler.Init.WaitSignalActive = FSMC_WAIT_TIMING_BEFORE_WS;
/* 存储器写使能 */
g_sram_handler.Init.WriteOperation = FSMC_WRITE_OPERATION_ENABLE;
/* 等待使能位,此处未用到 */
g_sram_handler.Init.WaitSignal = FSMC_WAIT_SIGNAL_DISABLE;
/* 读写使用相同的时序 */
g_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 */
g_sram_handler.Init.AsynchronousWait = FSMC_ASYNCHRONOUS_WAIT_DISABLE;
g_sram_handler.Init.WriteBurst = FSMC_WRITE_BURST_DISABLE; /* 禁止突发写 */
/* FSMC 读时序控制寄存器 */
/* 地址建立时间 (ADDSET) 为 1 个 HCLK 1/72M=13.8ns */
fsmc_readwritetim.AddressSetupTime = 0x00; /* 地址建立时间为 1 个 HCLK=13.8ns */
fsmc_readwritetim.AddressHoldTime = 0x00; /* 地址保持时间在模式 A 未用到 */
fsmc_readwritetim.DataSetupTime = 0x01; /* 数据保存时间为 2 个 HCLK=27.6ns */
fsmc_readwritetim.BusTurnAroundDuration = 0x00;
fsmc_readwritetim.AccessMode = FSMC_ACCESS_MODE_A; /* 模式 A */
HAL_SRAM_Init(&g_sram_handler,&fsmc_readwritetim,&fsmc_readwritetim);
}

```

初始化成功后,FSMC 控制器就能根据扩展的地址线访问 SRAM 的数据,于是我们可以直接根据地址指针来访问 SRAM,我们定义 SRAM 的写函数如下;

```

void sram_write(uint8_t *pbuf, uint32_t addr, uint32_t datalen)
{
    for (; datalen != 0; datalen--)
    {
        *(volatile uint8_t *) (SRAM_BASE_ADDR + addr) = *pbuf;
        addr++;
        pbuf++;
    }
}

```

同样地,也是利用地址,可以构造出一个 SRAM 的连续读函数:

```

void sram_read(uint8_t *pbuf, uint32_t addr, uint32_t datalen)
{
    for (; datalen != 0; datalen--)
    {
        *pbuf++ = *(volatile uint8_t *) (SRAM_BASE_ADDR + addr);
        addr++;
    }
}

```

注意以上两个函数是操作 uint8_t 类型的指针,当使用其它类型的指针时需要注意指针的偏移量。难点主要是根据 SRAM 芯片上的时序来初始化 FSMC 控制器,大家参考芯片手册上的时序结合代码来理解这部分初始化的过程。

2. main.c 代码

初始化好 SRAM,我们就可以使用 SRAM 中的存储进行编程了。我们利用 ARM 编译器的特性:可以在某一绝对地址定义变量。为方便测试,我们直接定义一个与 SRAM 容量大小类似

的数组，由于是 1M 位的 RAM，我们定义了 uint32_t 类型后，大小要除 4，故定义的测试数组如下：

```
/* 测试用数组，起始地址为：SRAM_BASE_ADDR */
#if ( __ARMCC_VERSION >= 6010050)
uint32_t g_test_buffer[250000] __attribute__((section(".bss.ARM.__at_0x68000000")));
#else
uint32_t g_test_buffer[250000] __attribute__((at(SRAM_BASE_ADDR)));
#endif
```

这里的 __attribute__(()) 是 ARM 编译器的一种关键字，它有很多种用法，可以通过特殊修饰指定变量或者函数的属性。大家可以去 MDK 的帮助文件里查找这个关键字的其它用法。这里我们要用这个关键字把变量放到指定的位置，而且用了条件编译，因为 MDK 的 AC5 和 AC6 下的语法不同。

通过前面的描述，我们知道 SRAM 的访问基地址是 0x68000000，如果我们定义一个与 SRAM 空间大小相同的数组，而且数组指向的位置就是 0x68000000 的话，则这通过数组就可以很方便直接操作这块存储空间。所以回来前面所说的 __attribute__ 这个关键字。对于 AC5，它可以用 __attribute__((at(地址))) 的方法来修饰变量，而且这个地址 **可以是一个算式**，这样编译器在编译时就会通过这个关键字判断并把这个数组放到我们定义的空间，如果硬件支持的情况下，我们就可以访问这些指定空间的变量或常量了。但是对于 AC6，同样指定地址，需要用 __attribute__((section(".bss.ARM.__at_地址"))) 的方法，指定一个绝对地址才能把变量或者常量放到我们所需要定义的位置。这里这个地址就 **不支持算式**了，但是这个语法对于相对而言更加地通用，其它平台的编译器如 gcc 也有类似的语法，而且 AC5 下也可以用 AC6 的这种语法来达到相同效果，两者之间的差异，大家可以多实践以进行区分。

在 main.c 文件中，我们还定义了一个 SRAM 读写测试函数 fsmc_sram_test，该函数内部实现比较简单，这里就不多说了。main 函数就是通过调用 fsmc_sram_test 函数实现对 SRAM 读写测试。

在 main 函数中编写代码如下：

```
int main(void)
{
    uint8_t key;
    uint8_t i = 0;
    uint32_t ts = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "SRAM TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Test Sram", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1:TEST Data", RED);

    for (ts = 0; ts < 250000; ts++)
    {
        g_test_buffer[ts] = ts; /* 预存测试数据 */
    }

    while (1)
    {
        key = key_scan(0); /* 不支持连接 */

        if (key == KEY0_PRES)
```

```

{
    fsmc_sram_test(30, 150); /* 测试 SRAM 容量 */
}
else if (key == KEY1_PRES) /* 打印预存测试数据 */
{
    for (ts = 0; ts < 250000; ts++)
    {
        /* 显示测试数据 */
        lcd_show_xnum(30, 170, g_test_buffer[ts], 6, 16, 0, BLUE);
    }
}
else
{
    delay_ms(10);
}

i++;
if (i == 20)
{
    i = 0;
    LED0_TOGGLE(); /* LED0 闪烁 */
}
}
}

```

47.5 下载验证

在代码编译成功之后，我们下载代码到开发板上，得到如图 47.5.1 所示界面：

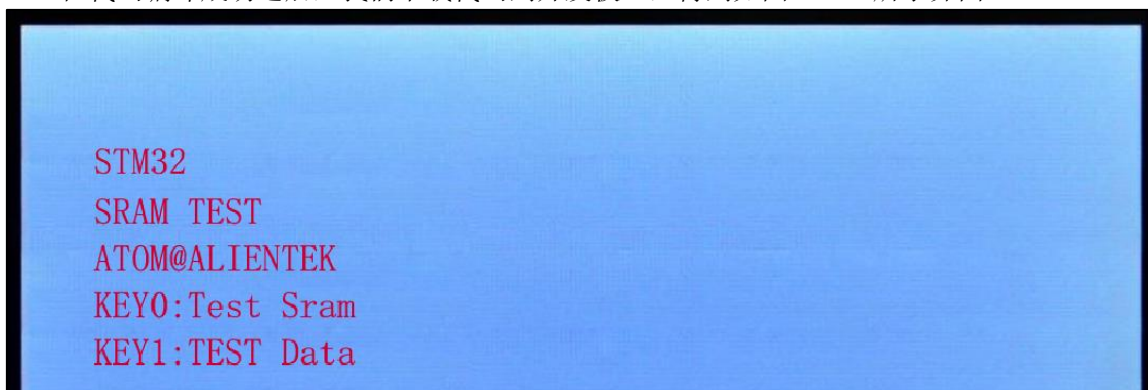


图 47.5.1 程序运行效果图

此时，我们按下 KEY0，就可以在 LCD 上看到内存测试的画面，同样，按下 KEY1，就可以看到 LCD 显示存放在数组 `g_test_buffer` 里面的测试数据，我们把数组的下标直接写到 SRAM 中，可以看到这个数据在不断地更新，SRAM 读写操作成功了，如图 47.5.2 所示：

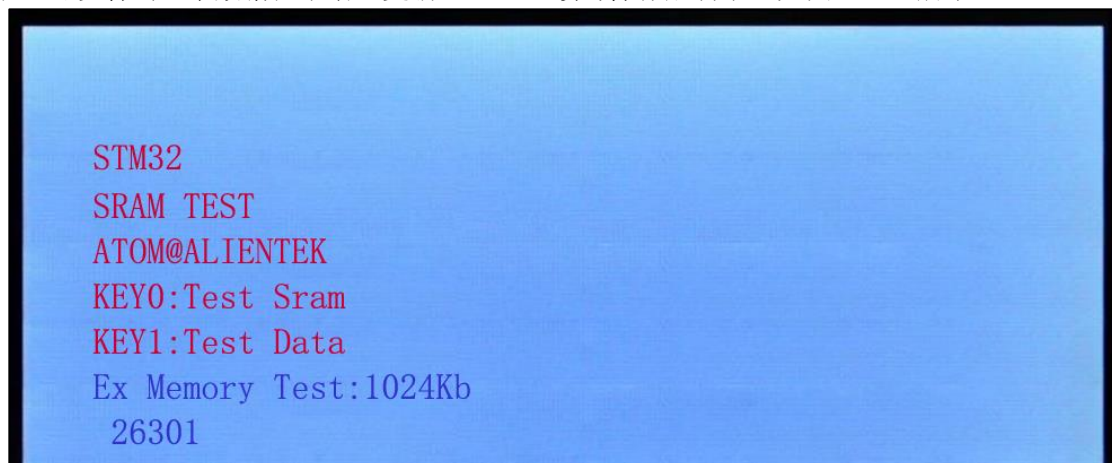


图 47.5.2 外部 SRAM 测试界面

该实验我们还可以借助 USART 来测试，如图 47.5.3 所示：



图 47.5.3 借助 USART 测试外部 SRAM 读写

第三篇 提高篇

现在我们已经学会如何使用 STM32 的一些基础外设了，STM32 我们算是入门了。接下来我们将学习提高篇的内容，本篇大部分内容都和 STM32 硬件底层关系不是很大，主要以各种第三方组件代码为主，比如：内存管理、文件系统、图片解码库、字库、手写识别库、T9 输入法、USB 库、网络协议栈、OS 等。这些内容偏向较复杂应用，掌握起来难度较大，需要大家认真学习，多多练习，做到举一反三。

本篇同样使用一章一实例的方式给大家介绍 STM32 的各种高级应用，通过本篇的学习，我们将可以完成更复杂的 STM32 项目开发。

第四十八章 内存管理实验

本章，我们将介绍内存管理。我们将使用内存的动态管理减少对内存的浪费。本章分为如下几个小节：

- 48.1 内存管理简介
- 48.2 硬件设计
- 48.3 程序设计
- 48.4 下载验证

48.1 内存管理简介

内存管理，是指软件运行时对计算机内存资源的分配和使用的技术。其最主要的目的是如何高效、快速的分配，并且在适当的时候释放和回收内存资源。内存管理的实现方法有很多种，其实最终都是要实现两个函数：`malloc` 和 `free`。`malloc` 函数用来内存申请，`free` 函数用于内存释放。

本章，我们介绍一种比较简单的办法来实现：分块式内存管理。下面我们介绍一下该方法的实现原理，如图 48.1.1 所示：

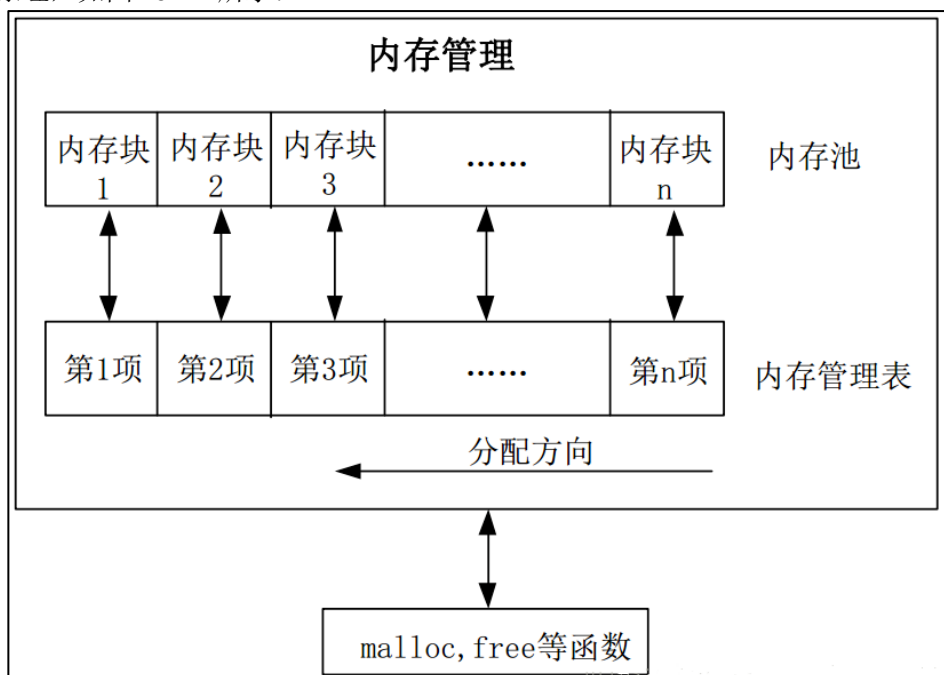


图 48.1.1 分块式内存管理原理

从上图可以看出，分块式内存管理由内存池和内存管理表两部分组成。内存池被等分为了 n 块，对应的内存管理表，大小也为 n ，内存管理表的每一个项对应内存池的一块内存。

内存管理表的项值代表的意义为：当该项值为 0 的时候，代表对应的内存块未被占用，当该项值非零的时候，代表该项对应的内存块已经被占用，其数值则代表被连续占用的内存块数。比如某项值为 10，那么说明包括本项对应的内存块在内，总共分配了 10 个内存块给外部的某个指针。

内存分配方向如上图所示，是从顶→底的分配方向。即首先从最末端开始找空内存。当内存管理刚初始化的时候，内存表全部清零，表示没有任何内存块被占用。

分配原理：

当指针 p 调用 `malloc` 申请内存的时候，先判断 p 要分配的内存块数 (m)，然后从第 n 开始，向下查找，直到找到 m 块连续的空内存块（即对应内存管理表项为 0），然后将这 m 个内存管理表项的值都设置为 m （标记被占用），最后，把最后的这个空内存块的地址返回指针 p ，

完成一次分配。注意：如果当内存不够的时候（找到最后也没有找到连续 m 块空闲内存），则返回 NULL 给 p ，表示分配失败。

释放原理：

当 p 申请的内存用完，需要释放的时候，调用 `free` 函数实现。`free` 函数先判断 p 指向的内存地址所对应的内存块，然后找到对应的内存管理表项目，得到 p 所占用的内存块数目 m （内存管理表项目的值就是所分配内存块的数目），将这 m 个内存管理表项目的值都清零，标记释放，完成一次内存释放。

48.2 硬件设计

1. 例程功能

开机后，LCD 显示提示信息，等待外部输入。KEY0 用于申请内存，每次申请 2K 字节内容，还向申请到的内存写入数据。KEY1 用于释放内存。WK_UP 用切换操作内存区（内部 SRAM / 外部 SRAM）。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY1 - PE3
KEY2 - PE2
WK_UP - PA0
- 3) 串口 1（PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面）
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) STM32 自带的 SRAM
- 6) 开发板板载的 SRAM

48.3 程序设计

48.3.1 程序流程图

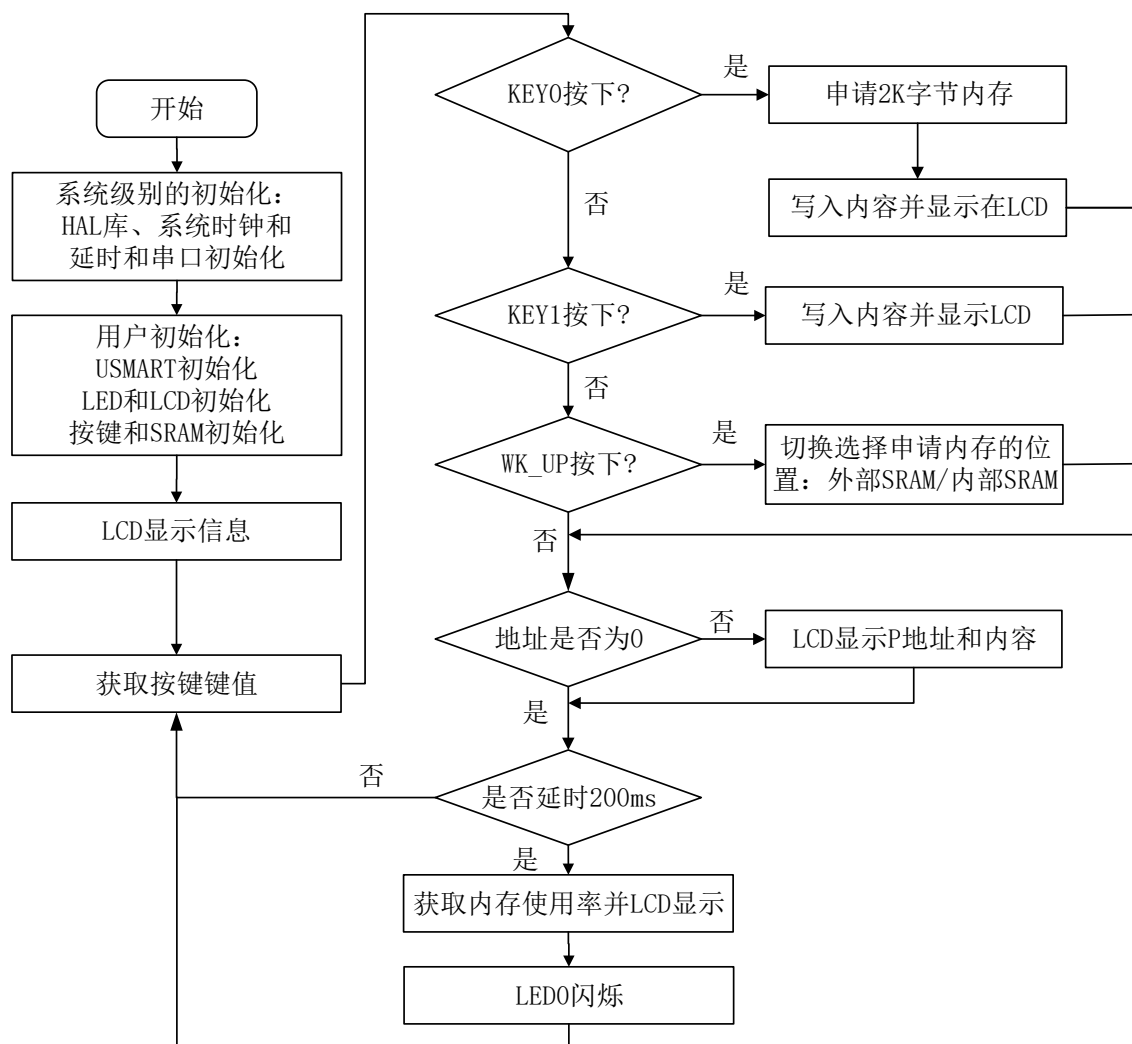


图 48.3.1.1 内存管理实验程序流程图

48.3.2 程序解析

1. 内存管理代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。内存管理驱动源码包括两个文件：`malloc.c` 和 `malloc.h`。这两个文件放在 `Middlewares` 文件夹下面的 `MALLOC` 文件夹。

下面我们直接介绍 `malloc.h` 中比较重要的一个结构体和内存参数宏定义，其定义如下：

```

/* 定义 3 个内存池 */
#define SRAMIN      0      /* 内部 SRAM，共 64KB */
#define SRAMEX      1      /* SRAMEX 内存池，外扩 SRAM，共 1024KB */
#define SRAMBANK    2      /* 定义支持的 SRAM 块数。 */

/* 定义内存管理表类型，当外扩 SDRAM 的时候，必须使用 uint32_t 类型，
   否则可以定义成 uint16_t，以节省内存占用 */
#define MT_TYPE      uint16_t
  
```

```

/* 单块内存，内存管理所占用的全部空间大小计算公式如下：
 * size = MEM1_MAX_SIZE + (MEM1_MAX_SIZE / MEM1_BLOCK_SIZE) * sizeof(MT_TYPE)
 * 以 SRAMEX 为例，size = 963 * 1024 + (963 * 1024 / 32) * 2 = 1047744 ≈ 1023KB

 * 已知总内存容量(size)，最大内存池的计算公式如下：
 * MEM1_MAX_SIZE = (MEM1_BLOCK_SIZE * size) / (MEM1_BLOCK_SIZE +
sizeof(MT_TYPE))
 * 以 SRAMIN 为例，MEM1_MAX_SIZE = (32 * 64) / (32 + 2) = 60.24KB ≈ 60KB
 * 但是我们为了给其他全局变量 / 数组等预留内存空间，这里设置最大管理为 40KB
 */

/* mem1 内存参数设定.mem1 是 F103 内部的 SRAM. */
#define MEM1_BLOCK_SIZE 32 /* 内存块大小为 32 字节 */
#define MEM1_MAX_SIZE 40 * 1024 /* 最大管理内存 40K, F103ZE 内部 SRAM 总共 64KB */
#define MEM1_ALLOC_TABLE_SIZE MEM1_MAX_SIZE/MEM1_BLOCK_SIZE /* 内存表大小 */
/* mem2 内存参数设定.mem2 是 F103 外扩 SRAM */
#define MEM2_BLOCK_SIZE 32 /* 内存块大小为 32 字节 */
#define MEM2_MAX_SIZE 963 * 1024 /* 最大管理内存 963K, F103 外扩 SRAM 大小 1024KB */
#define MEM2_ALLOC_TABLE_SIZE MEM2_MAX_SIZE/MEM2_BLOCK_SIZE /* 内存表大小 */
/* 内存管理控制器 */
struct _m_malloc_dev
{
    void (*init)(uint8_t); /* 初始化 */
    uint16_t (*perused)(uint8_t); /* 内存使用率 */
    uint8_t *membase[SRAMBANK]; /* 内存池 管理 SRAMBANK 个区域的内存 */
    MT_TYPE *memmap[SRAMBANK]; /* 内存管理状态表 */
    uint8_t memrdy[SRAMBANK]; /* 内存管理是否就绪 */
};

```

我们可以定义几个不同的内存管理表，再分配相应的指针给到管理控制器即可。程序中我们用宏定义 MEM1_BLOCK_SIZE 来定义 malloc 可以管理的内部内存池总大小，实际上我们定义为一个大小为 MEM1_BLOCK_SIZE 的数组，这样编译后就能获得一块实际的连续内存区域，这里是 40KB，MEM1_ALLOC_TABLE_SIZE 代表内存池的内存管理表大小。我们可以定义多个内存管理表，这样就可以同时管理多块内存。

从这里可以看出，如果内存分块越小，那么内存管理表就越大，当分块为 2 字节 1 个块的时候，内存管理表就和内存池一样大了（管理表的每项都是 uint16_t 类型），显然是不合适。我们这里取 32 字节，比例为 1: 16，内存管理表相对就比较小了。

通过这个内存管理控制器 _m_malloc_dev 结构体，我们把分块式内存管理的相关信息，其初始化函数、获取使用率、内存池、内存管理表以及内存管理的状态保存下来，实现对内存池的管理控制。其中，内存池的定义为：

```

/* 内存池 (64 字节对齐) */
static __align(64) uint8_t mem1base[MEM1_MAX_SIZE]; /* 内部 SRAM 内存池 */
static __align(64) uint8_t mem2base[MEM2_MAX_SIZE]
    __attribute__((at(SRAM_BASE_ADDR))); /* 外扩 SRAM 内存池 */

/* 内存管理表 */
static MT_TYPE mem1mapbase[MEM1_ALLOC_TABLE_SIZE]; /* 内部 SRAM 内存池 MAP */
static MT_TYPE mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute-
    __((at(SRAM_BASE_ADDR + MEM2_MAX_SIZE))); /* 外扩 SRAM 内存池 MAP */

```

这里我们定义了两个内存池：一个是内部的 SRAM，另一个是外部的 SRAM。

MDK 支持用 __attribute__((at(地址))) 的方法把变量定义到指定的区域，而且这个变量支持是算式，大家可以去 MKD 的帮助文件中查找 __attribute__ 这个关键字查找相关信息，有比较详细的介绍。（SRAM 实验一章我们也有过说明了）我们这里是通过这个关键字，指定 mem2mapbase 这个大数组的存放位置为 SRAM 上的空间，如果不加这个关键字修饰，MDK 会默认把这些变量定义到 STM32 的内部空间，这样的话就超出了 STM32 内部的 SRAM 空间，编译时会直接报错。当然还有其它把变量定义到指定位置的方法，大家可以自行研究下。

其中, MEM1_MAX_SIZE 是在头文件中定义的内存池大小。__align(64)定义内存池为 64 字节对齐, **这个非常重要!** 如果不加上这个限制, 在某些情况下 (比如分配内存给结构体指针), 可能出现错误, 所以一定要加上这个。

上面的写法是对于 AC5 来说的, 但是如果你想换成 AC6 编译器的话就比较麻烦了, 指定变量位置的函数变成 __attribute__((section(““.bss.ARM.__at_地址”))) 的方式, 其中的 .bss 表示初始化为 0, 而且这个方式不支持算式, 所以还用上面的方法直接用宏计算出 SRAM 的地址的方法不可行了, 所以我们需要直接手动算出 SRAM 对应的内存地址, 同样地 __align(64) 在 AC6 下的写法也变成了 __ALIGNED(64), 还有其它差异的部分, 大家参考 MDK 官方提供的 AC5 到 AC6 的迁移方法的文档, 这样定义的方法就变成下面的方式:

```
/* 内存池(64字节对齐) */
static __ALIGNED(64) uint8_t mem1base[MEM1_MAX_SIZE]; /* 内部 SRAM 内存池 */
static __ALIGNED(64) uint8_t mem2base[MEM2_MAX_SIZE] __attribute__((section(““.bss.ARM.__at_0x68000000”))); /* 外扩 SRAM 内存池 */

/* 内存管理表 */
static MT_TYPE mem1mapbase[MEM1_ALLOC_TABLE_SIZE]; /* 内部 SRAM 内存池 MAP */
static MT_TYPE mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((section(““.bss.ARM.__at_0x680f0c00”))); /* 外扩 SRAM 内存池 MAP */
```

整个 malloc 代码的核心函数: my_mem_malloc 和 my_mem_free, 分别用于内存申请和内存释放。思路就是前面 48.1 所介绍的分配内存和释放内存, 不过在这里, 这两个函数只是内部调用, 外部调用我们另外定义了 mymalloc 和 myfree 两个函数, 其他函数我们就不多介绍了。下面看一下分配内存和释放内存相关函数, 其定义如下:

```
/**
 * @brief      内存分配(内部调用)
 * @param      memx : 所属内存块
 * @param      size : 要分配的内存大小(字节)
 * @retval     内存偏移地址
 * @arg        0 ~ 0xFFFFFFFF : 有效的内存偏移地址
 * @arg        0xFFFFFFFF : 无效的内存偏移地址
 */
static uint32_t my_mem_malloc(uint8_t memx, uint32_t size)
{
    signed long offset = 0;
    uint32_t nmemb; /* 需要的内存块数 */
    uint32_t cmemb = 0; /* 连续空内存块数 */
    uint32_t i;

    if (!mallco_dev.memrdr[memx])
    {
        mallco_dev.init(memx); /* 未初始化, 先执行初始化 */
    }
    if (size == 0) return 0xFFFFFFFF; /* 不需要分配 */
    nmemb = size / memblksize[memx]; /* 获取需要分配的连续内存块数 */
    if (size % memblksize[memx]) nmemb++;
    for (offset=memtblsize[memx] - 1; offset >= 0; offset--) /* 搜索整个内存控制区 */
    {
        if (!mallco_dev.memmap[memx][offset])
        {
            cmemb++; /* 连续空内存块数增加 */
        }
        else
        {
            cmemb = 0; /* 连续内存块清零 */
        }

        if (cmemb == nmemb) /* 找到了连续 nmemb 个空内存块 */
        {
            for (i = 0; i < nmemb; i++) /* 标注内存块非空 */
            {
```

```

        malloc_dev.memmap[memx][offset + i] = nmemb;
    }
    return (offset * memblksize[memx]); /* 返回偏移地址 */
}
}
return 0xFFFFFFFF; /* 未找到符合分配条件的内存块 */
}

/**
 * @brief      释放内存(内部调用)
 * @param      memx    : 所属内存块
 * @param      offset  : 内存地址偏移
 * @retval     释放结果
 * @arg        0, 释放成功;
 * @arg        1, 释放失败;
 * @arg        2, 超区域了(失败);
 */
static uint8_t my_mem_free(uint8_t memx, uint32_t offset)
{
    int i;

    if (!malloc_dev.memrdy[memx]) /* 未初始化, 先执行初始化 */
    {
        malloc_dev.init(memx);
        return 1; /* 未初始化 */
    }
    if (offset < memsize[memx]) /* 偏移在内存池内. */
    {
        int index = offset / memblksize[memx]; /* 偏移所在内存块号码 */
        int nmemb = malloc_dev.memmap[memx][index]; /* 内存块数量 */
        for (i = 0; i < nmemb; i++) /* 内存块清零 */
        {
            malloc_dev.memmap[memx][index + i] = 0;
        }
        return 0;
    }
    else
    {
        return 2; /* 偏移超区了. */
    }
}

/**
 * @brief      释放内存(外部调用)
 * @param      memx    : 所属内存块
 * @param      ptr     : 内存首地址
 * @retval     无
 */
void myfree(uint8_t memx, void *ptr)
{
    uint32_t offset;
    if (ptr == NULL) return; /* 地址为 0. */
    offset = (uint32_t)ptr - (uint32_t)malloc_dev.membase[memx];
    my_mem_free(memx, offset); /* 释放内存 */
}

/**
 * @brief      分配内存(外部调用)
 * @param      memx    : 所属内存块
 * @param      size    : 要分配的内存大小(字节)
 * @retval     分配到的内存首地址.
 */

```

```
void *mymalloc(uint8_t memx, uint32_t size)
{
    uint32_t offset;
    offset = my_mem_malloc(memx, size);
    if (offset == 0xFFFFFFFF) /* 申请出错 */
    {
        return NULL; /* 返回空(0) */
    }
    else /* 申请没问题, 返回首地址 */
    {
        return (void *)((uint32_t)mallco_dev.membase[memx] + offset);
    }
}
```

2. main.c 代码

在 main.c 里面编写如下代码:

```
const char *SRAM_NAME_BUF[SRAMBANK] = {" SRAMIN ", " SRAMEX "};
int main(void)
{
    uint8_t paddr[20]; /* 存放 P Addr:+p 地址的 ASCII 值 */
    uint16_t memused = 0;
    uint8_t key;
    uint8_t i = 0;
    uint8_t *p = 0;
    uint8_t *tp = 0;
    uint8_t sramx = 0; /* 默认为内部 sram */

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "MALLOC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Malloc & WR & Show", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:SRAMx KEY1:Free", RED);
    lcd_show_string(60, 160, 200, 16, 16, " SRAMIN ", BLUE);
    lcd_show_string(30, 176, 200, 16, 16, "SRAMIN USED:", BLUE);
    lcd_show_string(30, 192, 200, 16, 16, "SRAMEX USED:", BLUE);

    while (1)
    {
        key = key_scan(0); /* 不支持连按 */
        switch (key)
        {
            case KEY0_PRES: /* KEY0 按下 */
                p = mymalloc(sramx, 2048); /* 申请 2K 字节, 并写入内容, 显示在 lcd 屏幕上 */
                if (p != NULL)
                {
                    /* 向 p 写入一些内容 */
                    sprintf((char *)p, "Memory Malloc Test%03d", i);
                    /* 显示 p 的内容 */
                    lcd_show_string(30, 260, 209, 16, 16, (char *)p, BLUE);
                }
                break;
```

```

        case KEY1_PRES:           /* KEY1 按下 */
            myfree(sramx, p);      /* 释放内存 */
            p = 0;                 /* 指向空地址 */
            break;
        case WKUP_PRES:           /* KEY UP 按下 */
            sramx++;
            if (sramx > 1) sramx = 0;
            lcd_show_string(60, 160, 200, 16, 16,
                           (char *)SRAM_NAME_BUF[sramx], BLUE);
            break;
    }

    if (tp != p)
    {
        tp = p;
        sprintf((char *)paddr, "P Addr:0X%08X", (uint32_t)tp);
        /* 显示 p 的地址 */
        lcd_show_string(30, 240, 209, 16, 16, (char *)paddr, BLUE);
        if (p)
        {
            /* 显示 p 的内容 */
            lcd_show_string(30, 260, 280, 16, 16, (char *)p, BLUE);
        }
        else
        {
            lcd_fill(30, 260, 209, 296, WHITE); /* p=0,清除显示 */
        }
    }

    delay_ms(10);
    i++;
    if ((i % 20) == 0)
    {
        memused = my_mem_perused(SRAMIN);
        sprintf((char *)paddr, "%d.%01d%%", memused / 10, memused % 10);
        /* 显示内部内存使用率 */
        lcd_show_string(30 + 112, 176, 200, 16, 16, (char *)paddr, BLUE);
        memused = my_mem_perused(SRAMEX);
        sprintf((char *)paddr, "%d.%01d%%", memused / 10, memused % 10);
        /* 显示外部内存使用率 */
        lcd_show_string(30 + 112, 192, 200, 16, 16, (char *)paddr, BLUE);
        LED0_TOGGLE(); /* LED0 闪烁 */
    }
}
}

```

该部分代码比较简单，主要是对 `mymalloc` 和 `myfree` 的应用。不过这里提醒大家，如果对一个指针进行多次内存申请，而之前的申请又没释放，那么将造成“内存泄露”，这是内存管理所不希望发生的，久而久之，可能导致无内存可用的情况！所以，在使用的时候，请大家一定记得，申请的内存存在用完以后，一定要释放。

另外，本章希望利用 USMART 调试内存管理，所以在 USMART 里面添加了 `mymalloc` 和 `myfree` 两个函数，用于测试内存分配和内存释放。大家可以通过 USMART 自行测试。

48.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 48.4.1 所示：

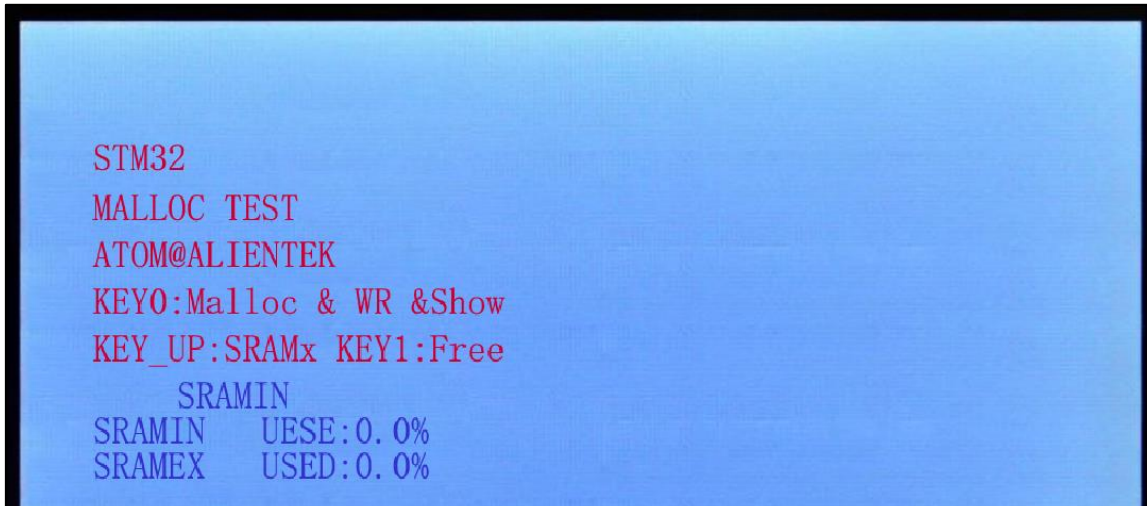


图 48.4.1 内存管理实验测试图

可以看到，内存的使用率均为 0%，说明还没有任何内存被使用。我们可以通过 KEY_UP 选择申请内存的位置：SRAMIN 为内部，SRAMEX 为外部。此时我们选择从内部申请内存，按下 KEY0，就可以看到申请了 5% 的一个内存块，同时看到下面提示了指针 p 所指向的地址（其实就是被分配到的内存地址）和内容，效果如图 48.4.2 所示。

KEY0 键用来申请内存并向 p 写入内容。多按几次 KEY0，可以看到内存使用率持续上升（注意比对 p 的值，可以发现是递减的，说明是从顶部开始分配内存！）。每次申请一个内存块后，可以通过按下 KEY1 释放本次申请的内存，如果我们每次申请完内存不再使用却不及时释放掉，再按 KEY1 将无法释放之前的内存了，当这样的情况重复了多次，就会造成“内存泄漏”。我们程序就是模拟这样一个情况，告诫大家在实际使用的时候，要注意到这种做法的危险性，必须在编程时严格避免内存泄漏的情况发生。

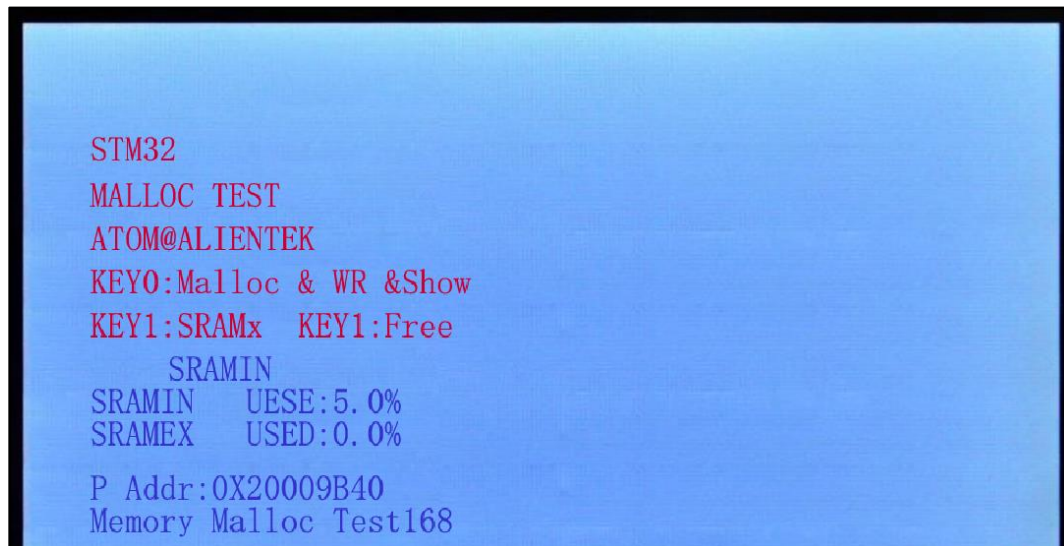


图 48.4.2 按下 KEY0 申请了部分内存

本章，我们还可以借助 USART，测试内存的分配和释放，有兴趣的朋友可以动手试试。如图 48.4.2 所示：

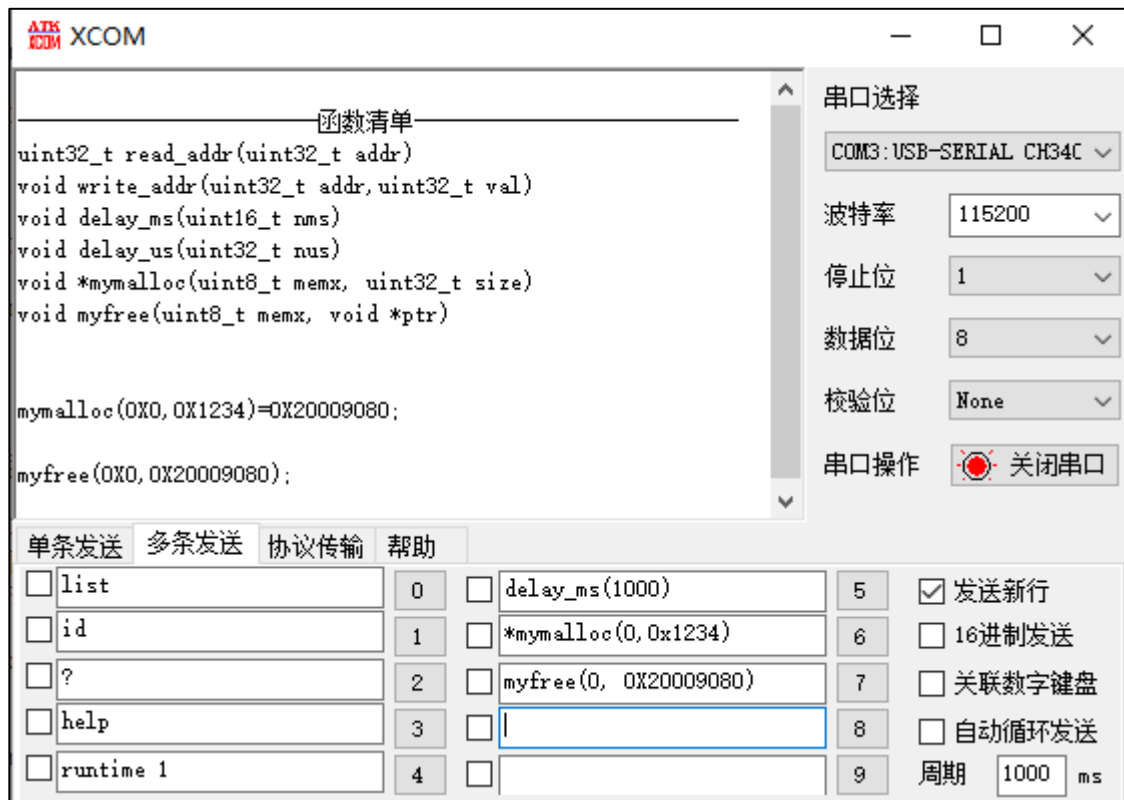


图 48.4.2 USART 测试内存管理图

图中，我们先申请了 4660 字节的内存，然后得到申请到的内存首地址为：0x20009080，说明我们申请内存成功（如果不成功，则会收到 0），然后释放内存的时候，参数是指针的地址，即执行：myfree(0, 0x20009080)，就可以释放我们申请到的内存。其他情况，大家可以自行测试并分析。

第四十九章 SD 卡实验

很多单片机系统都需要大容量存储设备，以存储数据。目前常用的有 U 盘，FLASH 芯片，SD 卡等。他们各有优点，综合比较，最适合单片机系统的莫过于 SD 卡了，它不仅容量可以做到很大（32GB 以上），支持 SPI/SDIO 驱动，而且有多种体积的尺寸可供选择（标准的 SD 卡尺寸及 Micro SD 卡尺寸等），能满足不同应用的要求。

只需要少数几个 IO 口即可外扩一个高达 32GB 或以上的外部存储器，容量从几十 M 到几十 G 选择范围很大，更换也很方便，编程也简单，是单片机大容量外部存储器的首选。

正点原子战舰 V4 版本以后，使用的接口是 Micro SD 卡接口，卡座带自锁功能，使用 STM32F1 自带的 SDIO 接口驱动，4 位模式，最高通信速度可达 24Mhz，最高每秒可传输数据 12M 字节，对于一般应用足够了。在本章中，我们将向大家介绍，如何在正点原子战舰 STM32F103 上实现 Micro SD 卡的读取。本章分为如下几个部分：

- 49.1 SD 卡简介
- 49.2 SDIO 接口简介
- 49.3 硬件设计
- 49.4 软件设计
- 49.5 下载验证

49.1 SD 卡简介

49.1.1 SD 物理结构

SD 卡的规范由 SD 卡协会明确，可以访问 <https://www.sdcard.org> 查阅更多标准。SD 卡主要有 SD、Mini SD 和 microSD(原名 TF 卡，2004 年正式更名为 Micro SD Card，为方便本文用 microSD 表示)三种类型，Mini SD 已经被 microSD 取代，使用得不多，根据最新的 SD 卡规格列出的参数如表 49.1.1.1 所示：

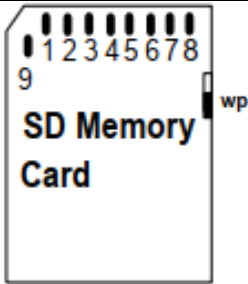
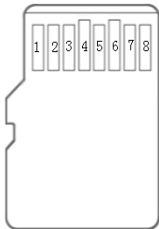
| 形状 | | SD | microSD |
|------|-------------------------------|---|--|
| 尺寸 | |  <p>32 x 24 x 2.1 mm, 32 x 24 x 1.4 mm, 约重 1.2g~2.5g</p> |  <p>11 x 15 x 1.0 mm, 约重 0.5g</p> |
| | | 卡片种类（容量范围） | |
| 硬件规格 | 脚位数 | SD(≤2GB)、SDHC(2GB~32GB)、SDXC(32GB ~2TB)、SDUC(2TB ~128TB) | |
| | | High Speed and UHS-I : 9 pins | High Speed and UHS-I : 8 pins |
| | | UHS-II and UHS-III: 17 pins | UHS-II and UHS-III: 16 pins |
| | | SD Express 1-lane: 17-19 pins | SD Express 1-lane: 17 pins |
| | SD Express 2-lane: 25-27 pins | | |
| | 电压范围 | 3.3V 版本: 2.7V - 3.6V | |
| | | 1.8V 低电压版: 1.70V-1.95V | |
| 防写开关 | 是 | 否 | |

表 49.1.1.1 SD 卡的主要规格参数

上述表格的“脚位数”，对应于实卡上的“金手指”数，不同类型的卡的触点数量不同，访问的速度也不相同。SD 卡允许了不同的接口来访问它的内部存储单元。最常见的是 SDIO 模式和 SPI 模式，根据这两种接口模式，我们也列出 SD 卡引脚对应于这两种不同的电路模式的引脚功能定义，如表 49.1.1.2 所示。

| SD 卡 | | SD Mode | | SPI Mode | | |
|------|---------|---------|------------|----------|------|----------|
| 引脚编号 | 引脚名 | 引脚类型 | 功能描述 | 引脚名 | 引脚类型 | 功能描述 |
| 1 | CD/DAT3 | I/O/PP | 卡识别/数据线位 3 | CS | I3 | 片选，低电平有效 |
| 2 | CMD | I/O/PP | 命令/响应 | DI | I | 数据输入 |
| 3 | VSS1 | S | 电源地 | VSS | S | 电源地 |
| 4 | VDD | S | DC 电源正极 | VDD | S | DC 电源正极 |
| 5 | CLK | I | Clock | SCLK | I | 时钟 |
| 6 | VSS2 | S | 电源地 | VSS2 | S | 电源地 |
| 7 | DAT0 | I/O/PP | 数据线位 0 | DO | O/PP | 数据输出 |
| 8 | DAT1 | I/O/PP | 数据线位 1 | RSV | | |
| 9 | DAT2 | I/O/PP | 数据线位 2 | RSV | | |

表 49.1.1.2 SD 卡引脚编号（注：S:电源 I: 输入 O: 推挽输出 PP: 推挽）

我们对比着来看一下 microSD 引脚，可见只比 SD 卡少了一个电源引脚 VSS2，其它的引脚功能类似。

| microSD | | SD Mode | | SPI Mode | | |
|---------|---------|---------|------------|----------|------|---------|
| 引脚编号 | 引脚名 | 引脚类型 | 功能描述 | 引脚名 | 引脚类型 | 功能描述 |
| 1 | DAT2 | I/O/PP | 数据线位 2 | RSV | | |
| 2 | CD/DAT3 | I/O/PP | 卡识别/数据线位 3 | CS | I | 片选低电平有效 |
| 3 | CMD | PP | 命令/响应 | DI | I | 数据输入 |
| 4 | VDD | S | DC 电源正极 | VDD | S | DC 电源正极 |
| 5 | CLK | I | 时钟 | SCLK | I | 时钟 |
| 6 | VSS | S | 电源地 | VSS | S | 电源地 |
| 7 | DAT0 | I/O/PP | 数据线位 0 | DO | O/PP | 数据输出 |
| 8 | DAT1 | I/O/PP | 数据线位 1 | RSV | | |

表 49.1.1.3 microSD 卡引脚编号（注：S:电源 I: 输入 O: 推挽输出 PP: 推挽）

SD 卡和 Micro SD 只有引脚和形状大小不同，内部结构类似，操作时序完全相同，可以使用完全相同的代码驱动，下面以 9’Pin SD 卡的内部结构为例，展示 SD 卡的存储结构，如图 49.1.1.1 所示。

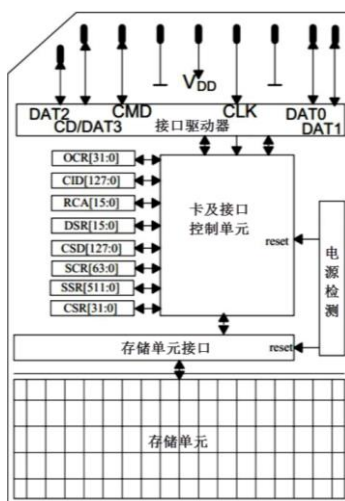


图 49.1.1.1 SD 卡内部物理结构（RCA 寄存器在 SPI 模式下不可访问）

SD 卡有自己的寄存器，但它不能直接进行读写操作，需要通过命令来控制，SDIO 协议定义了一些命令用于实现某一特定功能，SD 卡根据收到的命令要求对内部寄存器进行修改。图 49.1.1.4 中描述的 SD 卡的寄存器是我们和 SD 卡进行数据通讯的主要通道，如下：

| 名称 | 位宽 | 描述 |
|-----|-----|---|
| CID | 128 | 卡标识(Card identification): 每个卡都是唯一的 |
| 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，经主机核准。 |
| 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): 卡电源和状态标识 |
| 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 |

表 49.1.1.4 SD 卡寄存器信息

关于 SD 卡的更多信息和硬件设计规范可以参考 SD 卡协议《Physical Layer Simplified Specification Version 2.00》的相关章节(注：因为 STM32F1 的 SDIO 匹配的是 SD 协议 2.0 版本，后续版本也兼容此旧协议版本，故本章仍以 2.0 版本为介绍对象)。

49.1.2 命令和响应

一个完整的 SD 卡操作过程是：主机(单片机等)发起“命令”，SD 卡根据命令的内容决定是否发送响应信息及数据等，如果是数据读/写操作，主机还需要发送停止读/写数据的命令来结束本次操作，这意味着主机发起命令指令后，SD 卡可以没有响应、数据等过程，这取决于命令的含义。这一过程如图 49.1.2.1 所示。

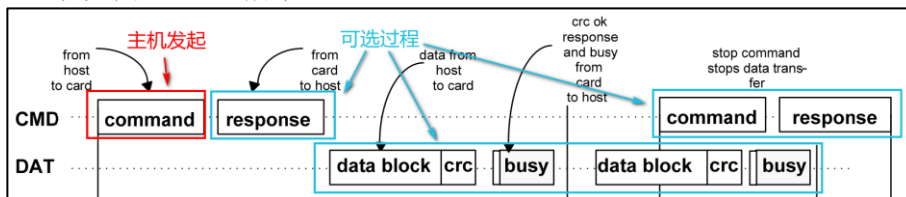


图 49.1.2.1 SD 卡命令格式

SD 卡有多种命令和响应，它们的格式定义及含义在《SD 卡协议 V2.0》的第三和第四章有详细介绍，发送命令时主机只能通过 CMD 引脚发送给 SD 卡，串行逐位发送时先发送最高位(MSB)，然后是次高位这样类推.....。

接下来，我们看看 SD 卡的命令格式，如表 49.1.2.1 所示：

| 字节1 | | 字节2--5 | 字节6 | |
|-----|----|---------|------|------|
| 47 | 46 | 45:40 | 39:8 | 7:1 |
| 0 | 1 | command | 命令参数 | CRC7 |
| | | | | 0 |
| | | | | 1 |

表 49.1.2.1 SD 卡控制命令格式

SD 卡的命令固定为 48 位，由 6 个字节组成，字节 1 的最高 2 位固定为 01，低 6 位为命令号（比如 CMD16，为 10000B 即 16 进制的 0X10，完整的 CMD16，第一个字节为 01010000，即 0X10+0X40）。字节 2~5 为命令参数，有些命令是没有参数的。字节 6 的高七位为 CRC 值，最低位恒定为 1。

SD 卡的命令总共有 12 类，分为 Class0~Class11，本章，我们仅介绍几个比较重要的命令，如表 49.1.2.2 所示：

| 命令 | 参数 | 响应 | 描述 |
|--------------|------------------|----|---------------------|
| CMD0(0X00) | NONE | 无 | 复位SD卡 |
| CMD8(0X08) | VHS+Checkpattern | R7 | 发送接口状态命令 |
| ACMD41(0X29) | HCS+VDD电压 | R3 | 主机发送容量支持信息和OCR寄存器内容 |

| | | | |
|-------------|------|-----|------------------|
| CMD2(0X02) | NONE | R2 | 读取SD卡的CID寄存器值 |
| CMD3(0X03) | NONE | R6 | 要求SD卡发布新的相对地址 |
| CMD9(0X09) | RCA | R2 | 获得选定卡的CSD寄存器的内容 |
| CMD7(0X07) | RCA | R1b | 选中SD卡 |
| CMD16(0X10) | 块大小 | R1 | 设置块大小（字节数） |
| CMD17(0X11) | 地址 | R1 | 读取一个块的数据 |
| CMD13(0x0D) | RCA | R1 | 被选中的卡返回其状态 |
| CMD24(0X18) | 地址 | R1 | 写入一个块的数据 |
| CMD55(0X37) | NONE | R1 | 告诉SD卡，下一个是特定应用命令 |

表 49.1.2.2 SD 卡部分命令

上表中，大部分的命令是初始化的时候用的，而表中的 R1、R1b、R2、R3、R6 和 R7 等是 SD 卡的应答信号。在主机发送有响应的命令后，SD 卡都会给出相对应的应答，以告知主机该命令的执行情况，或者返回主机需要获取的数据，具体场景如图 49.1.2.2 所示：

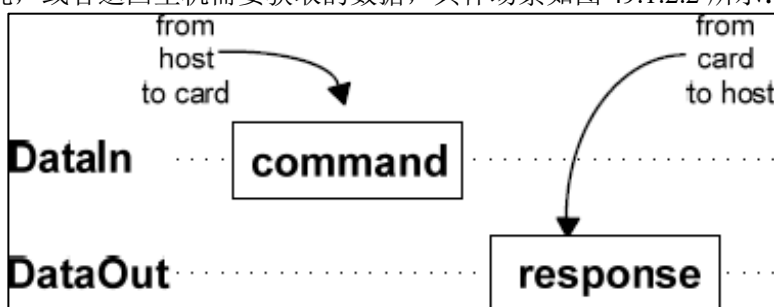


图 49.1.2.2 主机发送命令到 SD 卡应答通信过程

SD 的响应大体分为短响应 48bit 和长响应 136bit，每个响应也有规定好的格式。R1、R1b、R3、R6 和 R7 属于短响应，而 R2 属于长响应，它们具体作用如下表 49.1.2.3 所示。

| 响应 | 长度 | 描述 |
|-----|--------|--------------------------|
| R1 | 48bit | 正常响应命令，获知卡状态 |
| R1b | 48bit | 格式与R1相同，可选增加忙信号(数据线0上传输) |
| R2 | 136bit | 根据命令可返回CID/CSD寄存器值 |
| R3 | 48bit | ACMD41命令的响应，返回OCR寄存器值 |
| R6 | 48bit | 已发布的RCA响应 |
| R7 | 48bit | CMD8命令的响应(卡支持电压信息) |

表 49.1.2.3 SDIO 接口下的响应

SD 卡的响应因使用接口不同，比如 SDIO 和 SPI 接口，它们的响应种类以及响应格式也是不同。这里以 SDIO 接口下的 R1 响应为例，其内容格式如下表 49.1.2.4 所示：

| | 描述 | 起始位 | 传输位 | 命令号 | 卡状态 | CRC7 | 终止位 |
|-------|-----|-----|-----|---------|--------|-------|-----|
| R1 响应 | 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" |

表 49.1.2.4 R1 响应

R2~R7 的响应，限于篇幅，我们就不介绍了，但需要注意的是除了 R2 响应是 128 位外，其它的响应都是 48 位，请大家参考 SD 卡 2.0 协议。

49.1.3 SD 卡的操作模式

SD 卡系统(包括主机和 SD 卡)定义了 SD 卡的工作模式，在每个操作模式下，SD 卡都有几种状态，参考表 49.1.3.1，状态之间通过命令控制实现卡状态的切换。

| 操作模式 | 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) |

表 49.1.3.1 SD 卡状态与操作模式

对于我们来说两种有效操作模式：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，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；同时，SD 卡也处于卡识别模式，直到被主机识别到，即当 SD 卡在卡识别状态接收到 CMD3 (SEND_RCA)命令后，SD 卡就进入数据传输模式，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 49.1.3.1。

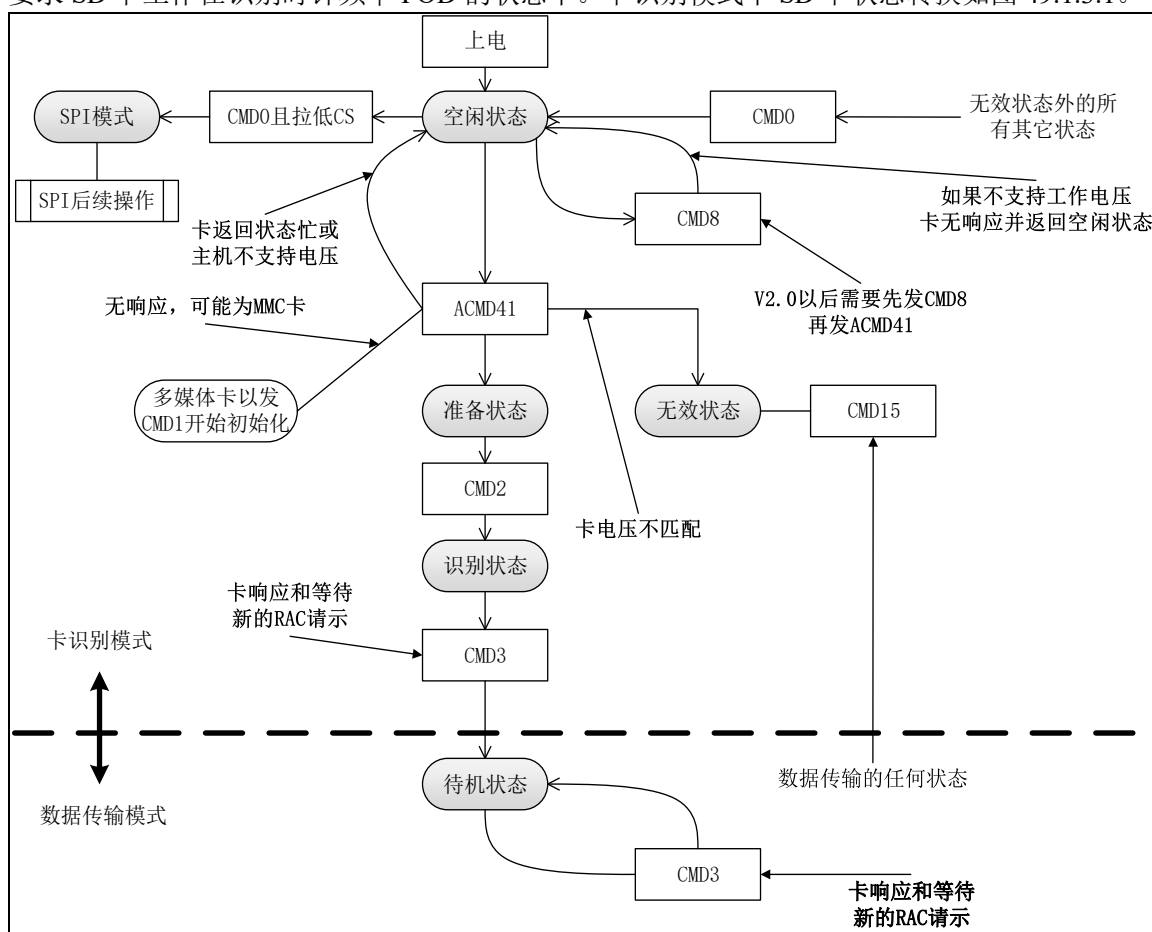


图 49.1.3.1 卡识别模式状态转换图

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

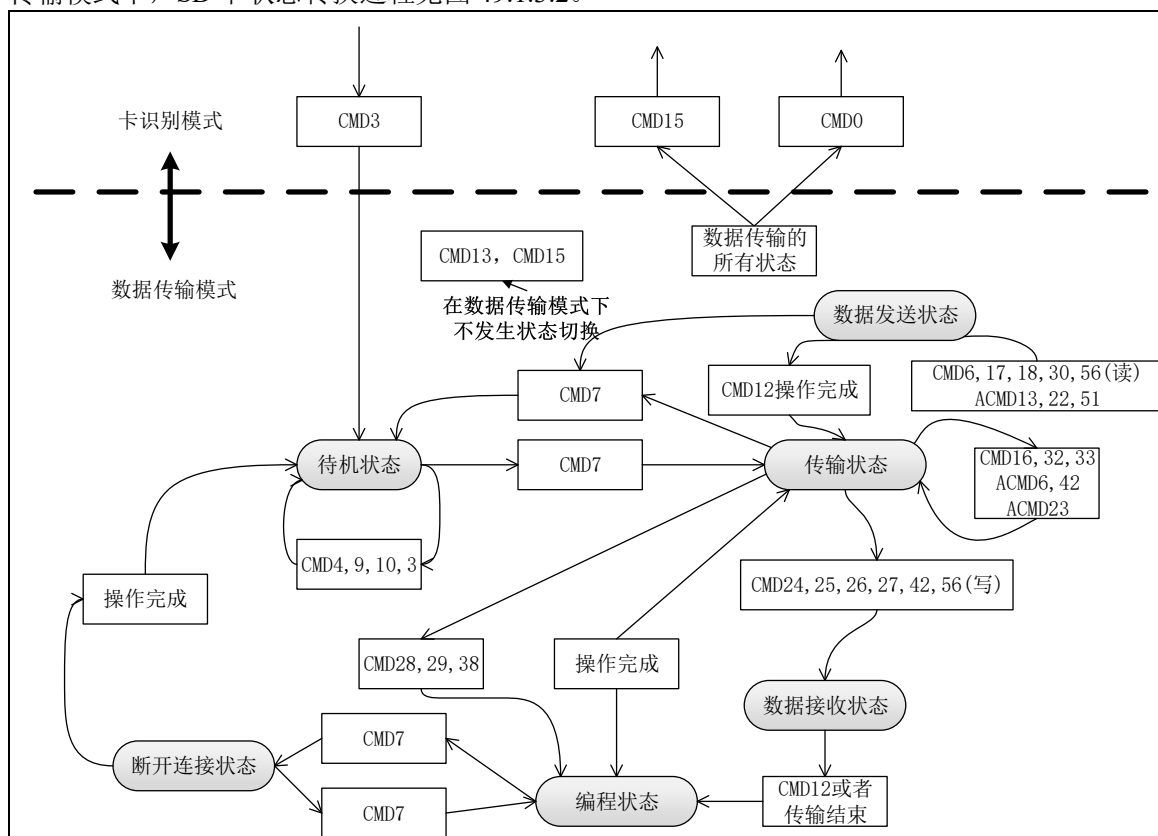
复位。

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。

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 49.1.3.2。



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通过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，注意谨慎使用，不当操作可能导致卡数据被损坏。

在数据模式下我们可以对 SD 卡的存储块进行读写访问操作。SD 卡上电后默认以一位数据总线访问，可以通过指令设置为宽总线模式，可以同时使有 4 位总线并行读写数据，这样对于支持宽总线模式的接口（如：SDIO）都能加快数据操作速度。

SD 卡有两种数据模式，一种是常规的 8 位宽，即一次按一字节传输，另一种是一次按 512 字节传输，我们只介绍前面一种。当按 8-bit 连续传输时，每次传输从最低字节开始，每字节从最高位(MSB)开始发送，当使用一条数据线时，只能通过 DAT0 进行数据传输，那它的数据传输结构如图 49.1.3.3 所示。

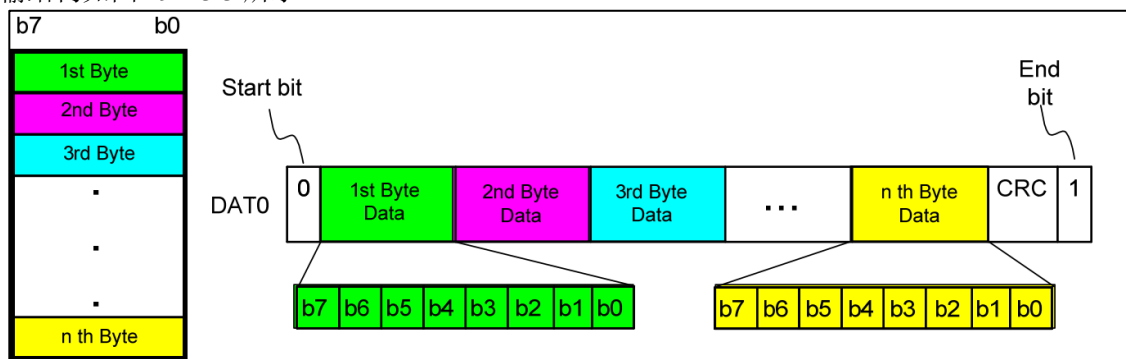


图 49.1.3.3 1 位数据线传输 8bit 的数据流格式

当使用 4 线模式传输 8-bit 结构的数据时，数据仍按 MSB 先发送的原则，DAT[3:0]的高位发送高数据位，低位发送低数据位。硬件支持的情况下，使用 4 线传输可以提升传输速率，其数据传输结构如图 49.1.3.4 所示。

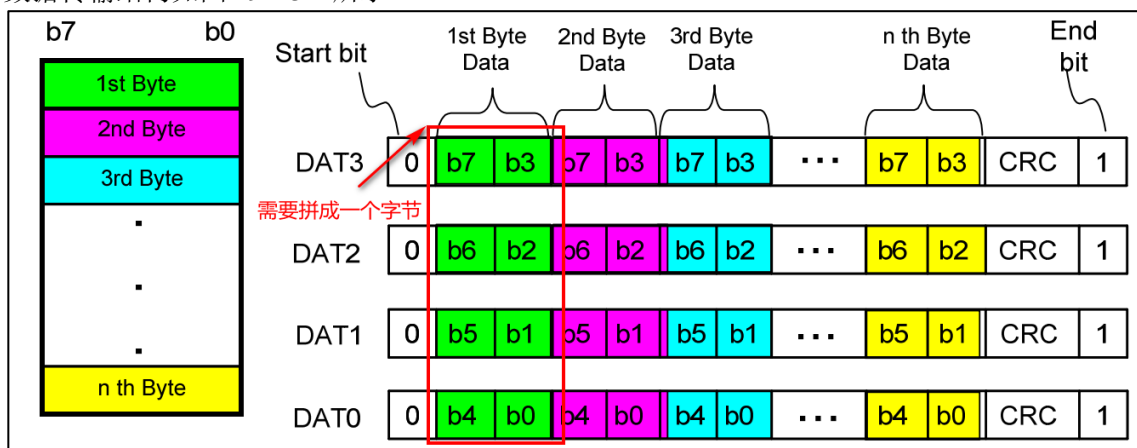


图 49.1.3.4 4 位数据线传输 8bit 格式的数据流格式

至此，我们已经介绍了 SD 卡操作的一些知识，并知道了 SD 卡操作的命令、响应和数据传输等状态，接下来我们来分析实际的硬件接口如何向 SD 卡发送我们需要的数据。

49.2 SDIO 接口简介

前面提到 SD 卡的驱动方式之一是用 SDIO 接口通讯，正点原子战舰 STM32F103 自带 SDIO 接口，本节，我们将简单介绍 STM32F1 的 SDIO 接口，包括：主要功能及框图、时钟、命令与响应和相关寄存器简介等。

49.2.1 SDIO 主要功能及框图

SDIO，全称为安全数字输入/输出接口，多媒体卡（MMC 卡）、SD 存储卡、SDI/O 卡和 CE-ATA 设备都有 SDIO 接口。SDIO 接口的设备整体概括如下图 49.2.1.1 所示。

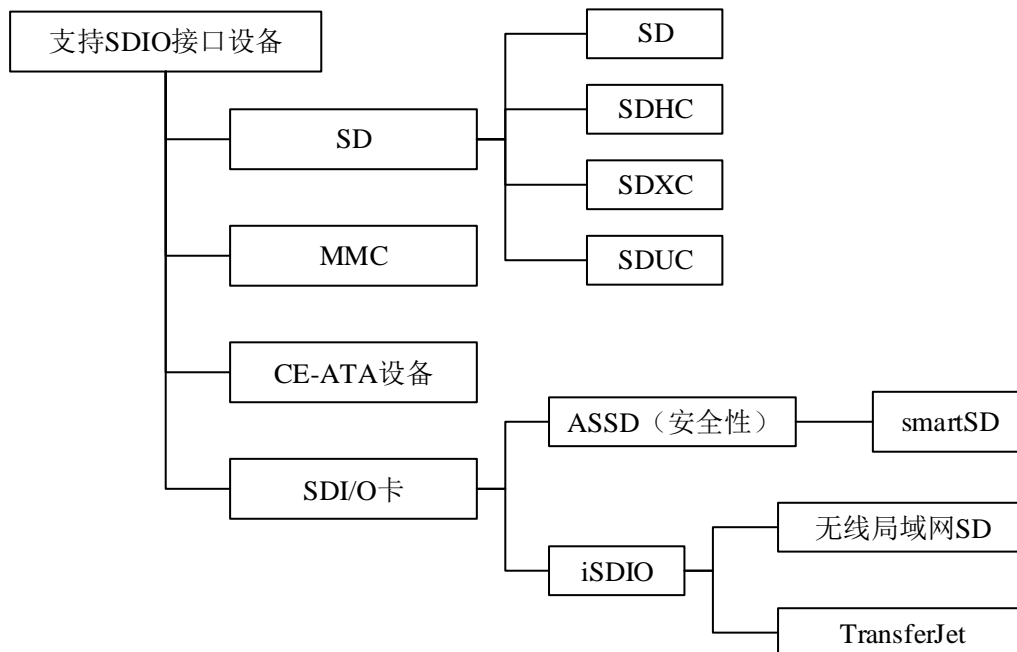


图 49.2.1.1 STM32F1 的 SDIO 支持的功能

STM32F1 的 SDIO 控制器支持多媒体卡（MMC 卡）、SD 存储卡、SDI/O 卡和 CE-ATA 设备等。SDIO 的主要功能如下：

- 与多媒体卡系统规格书版本 4.2 全兼容。支持三种不同的数据总线模式：1 位(默认)、4 位和 8 位。
- 与较早的多媒体卡系统规格版本全兼容(向前兼容)。
- 与 SD 存储卡规格版本 2.0 全兼容。SD 卡规范版本 2.0，包括 SD 和高容量 SDHC 标准卡，故不支持超大容量 SDXC/SDUC 标准卡，所以 STM32F1xx 的 SDIO 可以支持的最高卡容量是 32GB。
- 与 SDI/O 卡规格版本 2.0 全兼容：支持良种不同的数据总线模式：1 位(默认)和 4 位。
- 完全支持 CE-ATA 功能(与 CE-ATA 数字协议版本 1.1 全兼容)。8 位总线模式下数据传输速率可达 48MHz。
- 数据和命令输出使能信号，用于控制外部双向驱动器。
- SDIO 没有 SPI 兼容的通信模式。故用 SPI 方式驱动的 SD 卡具体看正点原子 MINI 板教程中的 SD 实验。

STM32F1 的 SDIO 控制器包含 2 个部分：SDIO 适配器模块和 AHB 总线接口，其功能框图如图 49.2.1.2 所示：

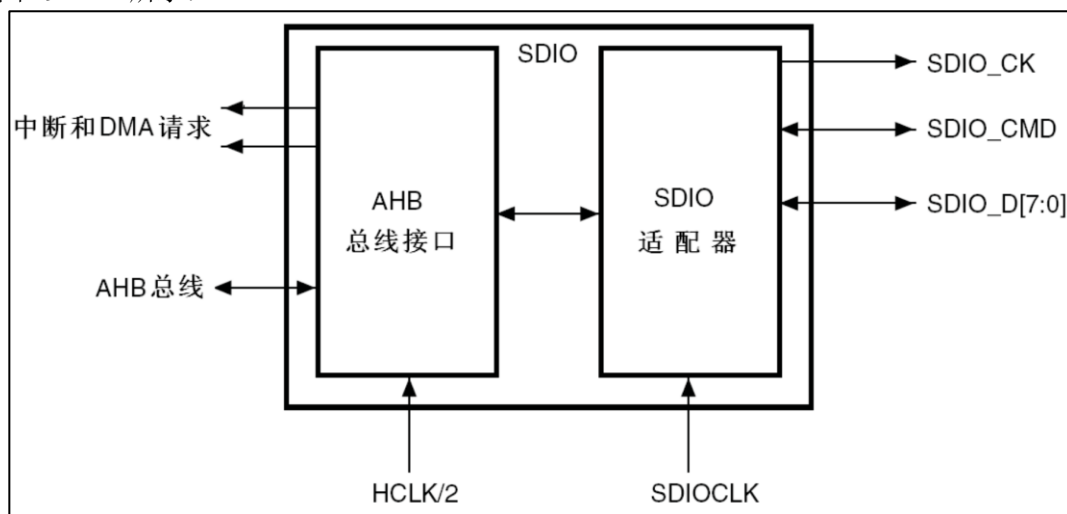


图 49.2.1.2 STM32F1 的 SDIO 控制器功能框图

复位后默认情况下 SDIO_D0 用于数据传输。初始化后主机可以改变数据总线的宽度（通过 ACMD6 命令设置）。

如果一个多媒体卡接到了总线上，则 SDIO_D0、SDIO_D[3:0]或 SDIO_D[7:0]可以用于数据传输。MMC 版本 V3.31 和之前版本的协议只支持 1 位数据线，所以只能用 SDIO_D0（为了通用性考虑，在程序里面我们只要检测到是 MMC 卡就设置为 1 位总线数据）。

如果一个 SD 或 SDI/O 卡接到了总线上，可以通过主机配置数据传输使用 SDIO_D0 或 SDIO_D[3:0]。所有的数据线都工作在推挽模式。

SDIO_CMD 有两种操作模式：

- ①用于初始化时的开路模式(仅用于 MMC 版本 V3.31 或之前版本)
- ②用于命令传输的推挽模式(SD/SD I/O 卡和 MMCV4.2 在初始化时也使用推挽驱动)

49.2.2 SDIO 的时钟

从图 49.2.1.2 我们可以看到 SDIO 总共有 3 个时钟，分别是：

①卡时钟（SDIO_CK）：每个时钟周期在命令和数据线上传输 1 位命令或数据。对于多媒体卡 V3.31 协议，时钟频率可以在 0MHz 至 20MHz 间变化；对于多媒体卡 V4.0/4.2 协议，时钟频率可以在 0MHz 至 48MHz 间变化；对于 SD 或 SDI/O 卡，时钟频率可以在 0MHz 至 25MHz 间变化。

②SDIO 适配器时钟（SDIOCLK）：该时钟用于驱动 SDIO 适配器，其频率等于 AHB 总线频率（HCLK），并用于产生 SDIO_CK 时钟。

③AHB 总线接口时钟（HCLK/2）：该时钟用于驱动 SDIO 的 AHB 总线接口，其频率为 HCLK/2。

前面提到，我们的 SD 卡时钟（SDIO_CK），根据卡的不同，可能有好几个区间，这就涉及到时钟频率的设置，SDIO_CK 与 SDIOCLK 的关系为：

$$SDIO_CK = \frac{SDIOCLK}{(2 + CLKDIV)}$$

其中，SDIO CLK 为 HCLK，一般是 72Mhz，而 CLKDIV 则是分配系数，可以通过 SDIO 的 SDIO_CLKCR 寄存器进行设置（确保 SDIO_CK 不超过卡的最大操作频率）。

这里要提醒大家，在 SD 卡刚刚初始化的时候，其时钟频率（SDIO_CK）是不能超过 400Khz 的，否则可能无法完成初始化。在初始化以后，就可以设置时钟频率到最大了（但不可超过 SD 卡的最大操作时钟频率）。

49.2.3 SDIO 的命令与响应

SDIO 的命令分为应用相关命令(ACMD)和通用命令(CMD)两部分，应用相关命令(ACMD)的发送，必须先发送通用命令（CMD55），然后才能发送应用相关命令（ACMD）。

SDIO 的所有命令和响应都只通过 SDIO_CMD 引脚传输的，任何命令的长度都是固定为 48 位，SDIO 的命令格式如表 49.2.3.1 所示：

| 位的位置 | 宽度 | 值 | 说明 |
|---------|----|---|------|
| 47 | 1 | 0 | 起始位 |
| 46 | 1 | 1 | 传输位 |
| [45:40] | 6 | - | 命令索引 |
| [39:8] | 32 | - | 参数 |
| [7:1] | 7 | - | CRC7 |
| 0 | 1 | 1 | 结束位 |

表 49.2.3.1 SDIO 命令格式

所有的命令都是由 STM32F1 发出，其中开始位、传输位、CRC7 和结束位由 SDIO 硬件控制，我们需要设置的就只有命令索引和参数部分。其中命令索引（如 CMD0，CMD1 之类的）在 SDIO_CMD 寄存器里面设置，命令参数则由寄存器 SDIO_ARG 设置。

一般情况下，选中的 SD 卡在接收到命令之后，都会回复一个应答（注意 CMD0 是无应答

的), 这个应答我们称之为响应, 响应也是在 CMD 线上串行传输的。STM32F1 的 SDIO 控制器支持 2 种响应类型, 即: 短响应 (48 位) 和长响应 (136 位), 这两种响应类型都带 CRC 错误检测 (注意不带 CRC 的响应应该忽略 CRC 错误标志, 如 CMD1 的响应)。

短响应的格式如表 49.2.3.2 所示:

| 位的位置 | 宽度 | 值 | 说明 |
|---------|----|---|------------------|
| 47 | 1 | 0 | 起始位 |
| 46 | 1 | 0 | 传输位 |
| [45:40] | 6 | - | 命令索引 |
| [39:8] | 32 | - | 参数 |
| [7:1] | 7 | - | CRC7 (或 1111111) |
| 0 | 1 | 1 | 结束位 |

表 49.2.3.2 SDIO 命令格式

长响应的格式如表 49.2.3.3 所示:

| 位的位置 | 宽度 | 值 | 说明 |
|-----------|-----|--------|-----------------------|
| 135 | 1 | 0 | 起始位 |
| 134 | 1 | 0 | 传输位 |
| [133:128] | 6 | 111111 | 保留 |
| [127:1] | 127 | - | CID 或 CSD (包括内部 CRC7) |
| 0 | 1 | 1 | 结束位 |

表 49.2.3.3 SDIO 命令格式

同样, 硬件为我们滤除了开始位、传输位、CRC7 以及结束位等信息, 对于短响应, 命令索引存放在 SDIO_RESPCMD 寄存器, 参数则存放在 SDIO_RESP1 寄存器里面。对于长响应, 则仅留 CID/CSD 位域, 存放在 SDIO_RESP1~SDIO_RESP4 等 4 个寄存器。

SD 存储卡总共有 5 类响应 (R1、R2、R3、R6、R7), 我们这里以 R1 为例简单介绍一下。R1 (普通响应命令) 响应属于短响应, 其长度为 48 位, R1 响应的格式如表 49.2.3.4 所示:

| 位的位置 | 宽度 (位) | 值 | 说明 |
|---------|--------|---|------|
| 47 | 1 | 0 | 起始位 |
| 46 | 1 | 0 | 传输位 |
| [45:40] | 6 | X | 命令索引 |
| [39:8] | 32 | X | 卡状态 |
| [7:1] | 7 | X | CRC7 |
| 0 | 1 | 1 | 结束位 |

表 49.2.3.4 R1 响应格式

在收到 R1 响应后, 我们可以从 SDIO_RESPCMD 寄存器和 SDIO_RESP1 寄存器分别读出命令索引和卡状态信息。关于其他响应的介绍, 请大家参考光盘:《SD 卡 2.0 协议.pdf》或《STM32 中文参考手册》第 20 章。

最后, 我们看看数据在 SDIO 控制器与 SD 卡之间的传输。对于 SDI/SDIO 存储器, 数据是以数据块的形式传输的, 而对于 MMC 卡, 数据是以数据块或者数据流的形式传输。本节我们只考虑数据块形式的数据传输。SDIO (多) 数据块读操作, 如图 49.2.3.1 所示:

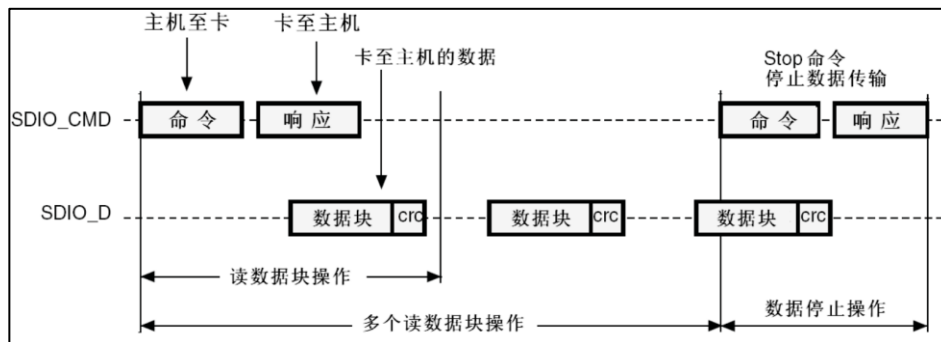


图 49.2.3.1 SDIO（多）数据块读操作

从上图，我们可以看出，从机在收到主机相关命令后，开始发送数据块给主机，所有数据块都带有 CRC 校验值（CRC 由 SDIO 硬件自动处理），单个数据块读的时候，在收到 1 个数据块以后即可以停止了，不需要发送停止命令（CMD12）。但是多块数据读的时候，SD 卡将一直发送数据给主机，直到接到主机发送的 STOP 命令（CMD12）。

SDIO（多）数据块写操作，如图 49.2.3.2 所示：

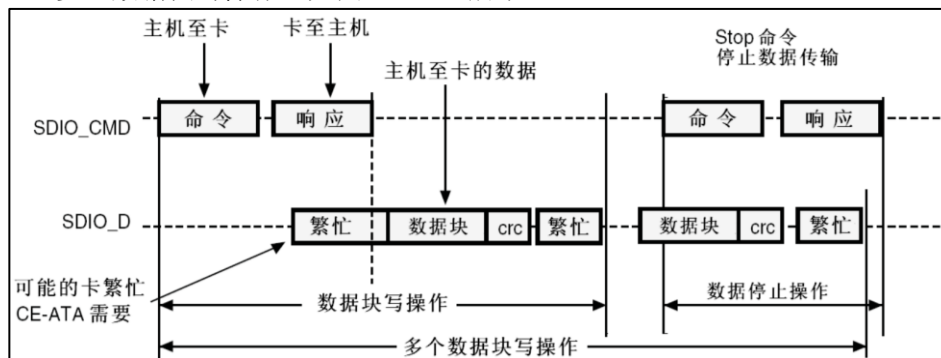


图 49.2.3.2 SDIO（多）数据块写操作

数据块写操作同数据块读操作基本类似，只是数据块写的时候，多了一个繁忙判断，新的数据块必须在 SD 卡非繁忙的时候发送。这里的繁忙信号由 SD 卡拉低 SDIO_D0，以表示繁忙，SDIO 硬件自动控制，不需要我们软件处理。

SDIO 的命令与响应就为大家介绍到这里。

49.2.4 SDIO 相关寄存器介绍

这部分将结合《STM32F10xxx 参考手册_V10（中文版）.pdf》的内容和大家一起分析使用 SDIO 时我们主要用的一些寄存器的情况。

● SDIO 电源控制寄存器（SDIO_POWER）

SDIO 电源控制寄存器（SDIO_POWER），该寄存器定义如图 49.2.4.1 所示：

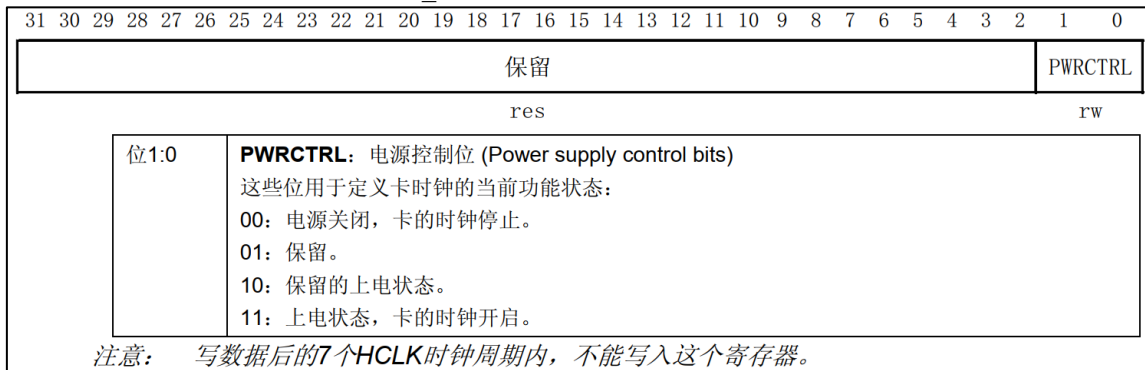


图 49.2.4.1 SDIO_POWER 寄存器位定义

该寄存器复位值为 0，所以 SDIO 的电源是关闭的，我们要启用 SDIO，第一步就是要设置该寄存器最低 2 个位均为 1，让 SDIO 上电，开启卡时钟。

● SDIO 时钟控制寄存器 (SDIO_CLKCR)

SDIO 时钟控制寄存器 (SDIO_CLKCR)，该寄存器主要用于设置 SDIO_CK 的分配系数，开关等，并可以设置 SDIO 的数据位宽，该寄存器的定义如图 49.2.4.2 所示：

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---------|---------|--------|--------|---------|-------|--------|---|---|---|---|---|---|---|---|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 保留 | | | | | | | | | | | | | | | | | HWFC_EN | NEGEDGE | WIDBUS | BYPASS | PWRSAPV | CLKEN | CLKDIV | | | | | | | | | |
| res | | | | | | | | | | | | | | | | | rw | rw | rw | rw | rw | rw | r/w | | | | | | | | | |
| 位12:11 | | WIDBUS : 宽总线模式使能位 (Wide bus mode enable bit) 00: 默认总线模式，使用SDIO_D0。 01: 4位总线模式，使用SDIO_D[3:0]。 10: 8位总线模式，使用SDIO_D[7:0]。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位10 | | BYPASS : 旁路时钟分频器 (Clock divider bypass enable bit) 0: 关闭旁路：驱动SDIO_CK输出信号之前，依据CLKDIV数值对SDIOCLK分频。 1: 使能旁路：SDIOCLK直接驱动SDIO_CK输出信号。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位8 | | CLKEN : 时钟使能位 (Clock enable bit) 0: SDIO_CK关闭。 1: SDIO_CK使能。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位7:0 | | CLKDIV : 时钟分频系数 (Clock divide factor) 这个域定义了输入时钟(SDIOCLK)与输出时钟(SDIO_CK)间的分频系数： SDIO_CK频率 = SDIOCLK/[CLKDIV + 2]。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 49.2.4.2 SDIO_CLKCR 寄存器位定义

上图仅列出了部分我们要用到的位设置，WIDBUS 用于设置 SDIO 总线位宽，正常使用的时候，设置为 1，即 4 位宽度。BYPASS 用于设置分频器是否旁路，我们一般要使用分频器，所以这里设置为 0，禁止旁路。CLKEN 则用于设置是否使能 SDIO_CK，我们设置为 1。最后，CLKDIV 则用于控制 SDIO_CK 的分频，设置为 1，即可得到 24Mhz 的 SDIO_CK 频率。

● SDIO 参数寄存器 (SDIO_ARG)

SDIO 参数寄存器 (SDIO_ARG)，该寄存器比较简单，就是一个 32 位寄存器，用于存储命令参数，不过需要注意的是，必须在写命令之前先写这个参数寄存器！

● SDIO 命令响应寄存器 (SDIO_RESPCMD)

SDIO 命令响应寄存器 (SDIO_RESPCMD)，该寄存器为 32 位，但只有低 6 位有效，比较简单，用于存储最后收到的命令响应中的命令索引。如果传输的命令响应不包含命令索引，则该寄存器的内容不可预知。

● SDIO 响应寄存器组 (SDIO_RESP1~SDIO_RESP4)

SDIO 响应寄存器组 (SDIO_RESP1~SDIO_RESP4)，该寄存器组总共由 4 个 32 位寄存器组成，用于存放接收到的卡响应部分信息。如果收到短响应，则数据存放在 SDIO_RESP1 寄存器里面，其他三个寄存器没有用到。而如果收到长响应，则依次存放在 SDIO_RESP1~SDIO_RESP4 里面，如表 49.2.4.1 所示：

| 寄存器 | 短响应 | 长响应 |
|------------|------------|--------------|
| SDIO_RESP1 | 卡状态 [31:0] | 卡状态 [127:96] |
| SDIO_RESP2 | 未使用 | 卡状态 [95:64] |
| SDIO_RESP3 | 未使用 | 卡状态 [63:32] |
| SDIO_RESP4 | 未使用 | 卡状态 [31:1]0b |

表 49.2.4.1 响应类型和 SDIO_RESPx 寄存器

● SDIO 命令寄存器 (SDIO_CMD)

SDIO 命令寄存器 (SDIO_CMD) 各位定义如图 49.2.4.3 所示：

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------|----|--|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----------|------|------------|-------------|--------|----------|---------|----------|----------|----|----|---|---|---|---|--|--|--|--|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | |
| 保留 | | | | | | | | | | | | | | | | | CE_ATACMD | nIEN | ENCMDcompl | SDIOSuspend | CPSMEN | WAITPEND | WAITINT | WAITRESP | CMDINDEX | | | | | | | | | | | |
| res | | | | | | | | | | | | | | | | | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | rW | | | | | | | | | |
| 位10 | | CPSMEN： 命令通道状态机(CPSM)使能位 (Command path state machine (CPSM) Enable bit) 如果设置该位，则使能CPSM。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位7:6 | | WAITRESP： 等待响应位 (Wait for response bits) 这2位指示CPSM是否需要等待响应，如果需要等待响应，则指示响应类型。 00：无响应，期待CMDSENT标志 01：短响应，期待CMDREND或CCRCFAIL标志 10：无响应，期待CMDSENT标志 11：长响应，期待CMDREND或CCRCFAIL标志 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位5:0 | | CMDINDEX： 命令索引 (Command index) 命令索引是作为命令的一部分发送到卡中。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 49.2.4.3 SDIO_CMD 寄存器位定义

图中只列出了部分位的描述, 其中低 6 位为命令索引, 也就是我们要发送的命令索引号 (比如发送 CMD1, 其值为 1, 索引就设置为 1)。位[7:6], 用于设置等待响应位, 用于指示 CPSM 是否需要等待, 以及等待类型等。这里的 CPSM, 即命令通道状态机, 我们就不详细介绍了, 请参阅《STM32F10xxx 参考手册_V10 (中文版).pdf》第 368 页, 有详细介绍。命令通道状态机我们一般都是开启的, 所以位 10 要设置为 1。

● SDIO 数据定时器寄存器 (SDIO_DTIMER)

SDIO 数据定时器寄存器 (SDIO_DTIMER) 用于存储以卡总线时钟 (SDIO_CK) 为周期的数据超时时间, 一个计数器将从 SDIO_DTIMER 寄存器加载数值, 并在数据通道状态机(DPSM)进入 Wait_R 或繁忙状态时进行递减计数, 当 DPSM 处在这些状态时, 如果计数器减为 0, 则设置超时标志。这里的 DPSM, 即数据通道状态机, 类似 CPSM, 详细请参考《STM32F10xxx 参考手册_V10 (中文版).pdf》第 372 页。注意: 在写入数据控制寄存器, 进行数据传输之前, 必须先写入该寄存器 (SDIO_DTIMER) 和数据长度寄存器 (SDIO_DLEN)!

● SDIO 数据长度寄存器 (SDIO_DLEN)

SDIO 数据长度寄存器 (SDIO_DLEN) 低 25 位有效, 用于设置需要传输的数据字节长度。对于块数据传输, 该寄存器的数值, 必须是数据块长度 (通过 SDIO_DCTRL 设置) 的倍数。

● SDIO 数据控制寄存器 (SDIO_DCTRL)

SDIO 数据控制寄存器 (SDIO_DCTRL) 各位定义如图 49.2.4.4 所示:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|--------|-------|--------|---------|------------|---|---|---|-------|--------|-------|------|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | |
| 保留 | | | | | | | | | | | | | | | | | | | | SDIOEN | RWMOD | RWSTOP | RWSTART | DBLOCKSIZE | | | | DMAEN | DTMODE | DTDIR | DTEN | | | | |
| res | | | | | | | | | | | | | | | | | | | | rW | rW | rW | rW | rW | | | | rW | | | | rW | rW | rW | rW |
| 位11 | | SDIOEN: SD I/O使能功能 (SD I/O enable functions) 如果设置了该位，则DPSM执行SD I/O卡特定的操作。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位10 | | RWMOD: 读等待模式 (Read wait mode) 0: 停止SDIO_CK控制读等待； | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 49.2.4.4 SDIO_DCTRL 寄存器位定义

该寄存器, 用于控制数据通道状态机 (DPSM), 包括数据传输使能、传输方向、传输模式、DMA 使能、数据块长度等信息, 都是通过该寄存器设置。我们需要根据自己的实际情况, 来配置该寄存器, 才可正常实现数据收发。

● **SDIO 状态寄存器 (STA) / 清除中断寄存器 (ICR) / 中断屏蔽寄存器 (MASK)**

接下来, 我们介绍几个位定义十分类似的寄存器, 他们是: 状态寄存器 (SDIO_STA)、清除中断寄存器 (SDIO_ICR) 和中断屏蔽寄存器 (SDIO_MASK), 这三个寄存器每个位的定义都相同, 只是功能各有不同。所以可以一起介绍, 以状态寄存器 (SDIO_STA) 为例, 该寄存器各位定义如图 49.2.4.5 所示:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----|----|----|----|----|----|----|----|--|--------|--------|--------|---------|---------|--------|--------|---------|----------|-------|-------|--------|---------|----------|---------|---------|---------|---------|----------|----------|----------|----------|----------|---|--|--|--|--|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | |
| 保留 | | | | | | | | CEATAEND | SDIOIT | RXDVAL | TXDVAL | RXFIFOE | TXFIFOE | RXFIFO | TXFIFO | RXFIFOH | TXFIFOHE | RXACT | TXACT | CMDACT | DBCKEND | STBITERR | DATAEND | CMDSENT | CMDREND | RXOVERR | TXUNDERR | DTIMEOUT | CTIMEOUT | DCRCFAIL | CCRCFAIL | | | | | |
| res | | | | | | | | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | | | | |
| 位23 | | | | | | | | CEATAEND: 在CMD61接收到CE-ATA命令完成信号 (CE-ATA command completion signal received for CMD61) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位22 | | | | | | | | SDIOIT: 收到SDIO中断 (SDIO interrupt received) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位21 | | | | | | | | RXDVAL: 在接收FIFO中的数据可用 (Data available in receive FIFO) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位20 | | | | | | | | TXDVAL: 在发送FIFO中的数据可用 (Data available in transmit FIFO) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位19 | | | | | | | | RXFIFOE: 接收FIFO空 (Receive FIFO empty) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位18 | | | | | | | | TXFIFOE: 发送FIFO空 (Transmit FIFO empty) 若使用了硬件流控制, 当FIFO包含2个字时, TXFIFOE信号变为有效。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位17 | | | | | | | | RXFIFO: 接收FIFO满 (Receive FIFO full) 若使用了硬件流控制, 当FIFO还差2个字满时, RXFIFO信号变为有效。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位16 | | | | | | | | TXFIFO: 发送FIFO满 (Transmit FIFO full) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位15 | | | | | | | | RXFIFOH: 接收FIFO半满 (Receive FIFO half full): FIFO中至少还有8个字。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位14 | | | | | | | | TXFIFOHE: 发送FIFO半空 (Transmit FIFO half empty): FIFO中至少还可以写入8个字。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位13 | | | | | | | | RXACT: 正在接收数据 (Data receive in progress) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位12 | | | | | | | | TXACT: 正在发送数据 (Data transmit in progress) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位11 | | | | | | | | CMDACT: 正在传输命令 (Command transfer in progress) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位10 | | | | | | | | DBCKEND: 已发送/接收数据块(CRC检测成功) (Data block sent/received (CRC check passed)) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位9 | | | | | | | | STBITERR: 在宽总线模式, 没有在所有数据信号上检测到起始位 (Start bit not detected on all data signals in wide bus mode) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位8 | | | | | | | | DATAEND: 数据结束 (数据计数器, SDIO_DCOUNT = 0) (Data end (data counter, SDIDCOUNT, is zero)) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位7 | | | | | | | | CMDSENT: 命令已发送(不需要响应) (Command sent (no response required)) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位6 | | | | | | | | CMDREND: 已接收到响应(CRC检测成功) (Command response) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位5 | | | | | | | | RXOVERR: 接收FIFO上溢错误 (Received FIFO overrun error) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位4 | | | | | | | | TXUNDERR: 发送FIFO下溢错误 (Transmit FIFO underrun error) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位3 | | | | | | | | DTIMEOUT: 数据超时 (Data timeout) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位2 | | | | | | | | CTIMEOUT: 命令响应超时 (Command response timeout) 命令超时时间是一个固定的值, 为64个SDIO_CLK时钟周期。 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位1 | | | | | | | | DCRCFAIL: 已发送/接收数据块(CRC检测失败) (Data block sent/received) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 位0 | | | | | | | | CCRCFAIL: 已收到命令响应(CRC检测失败) (Command response received) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 49.2.4.5 SDIO_STA 寄存器位定义

状态寄存器可以用来查询 SDIO 控制器的当前状态, 以便处理各种事务。比如 SDIO_STA 的位 2 表示命令响应超时, 说明 SDIO 的命令响应出了问题。我们通过设置 SDIO_ICR 的位 2 则可以清除这个超时标志, 而设置 SDIO_MASK 的位 2, 则可以开启命令响应超时中断, 设置为 0 关闭。其他位我们就不一一介绍了, 请大家自行学习。

● SDIO 数据 FIFO 寄存器 (SDIO_FIFO)

数据 FIFO 寄存器包括接收和发送 FIFO, 他们由一组连续的 32 个地址上的 32 个寄存器组成, CPU 可以使用 FIFO 读写多个操作数。例如我们要从 SD 卡读数据, 就必须读 SDIO_FIFO 寄存器, 要写数据到 SD 卡, 则要写 SDIO_FIFO 寄存器。SDIO 将这 32 个地址分为 16 个一组, 发送接收各占一半。而我们每次读写的时候, 最多就是读取发送 FIFO 或写入接收 FIFO 的一半大小的数据, 也就是 8 个字 (32 个字节), **这里特别提醒, 我们操作 SDIO_FIFO (不论读出还是写入) 必须是以 4 字节对齐的内存进行操作, 否则将导致出错!**

至此, SDIO 的相关寄存器介绍, 我们就介绍完了。还有几个不常用的寄存器, 我们没有介绍到, 请大家参考《STM32F10xxx 参考手册_V10 (中文版).pdf》第 20 章相关章节。

49.2.5 SDIO 模式下的 SD 卡初始化

这一节，我们来看看 SD 卡的初始化流程，要实现 SDIO 驱动 SD 卡，最重要的步骤就是 SD 卡的初始化，只要 SD 卡初始化完成了，那么剩下的（读写操作）就简单了，所以我们这里重点介绍 SD 卡的初始化。从《SD 卡 2.0 协议》（见光盘资料）文档，我们得到 SD 卡初始化流程图如图 49.2.5.1 所示：

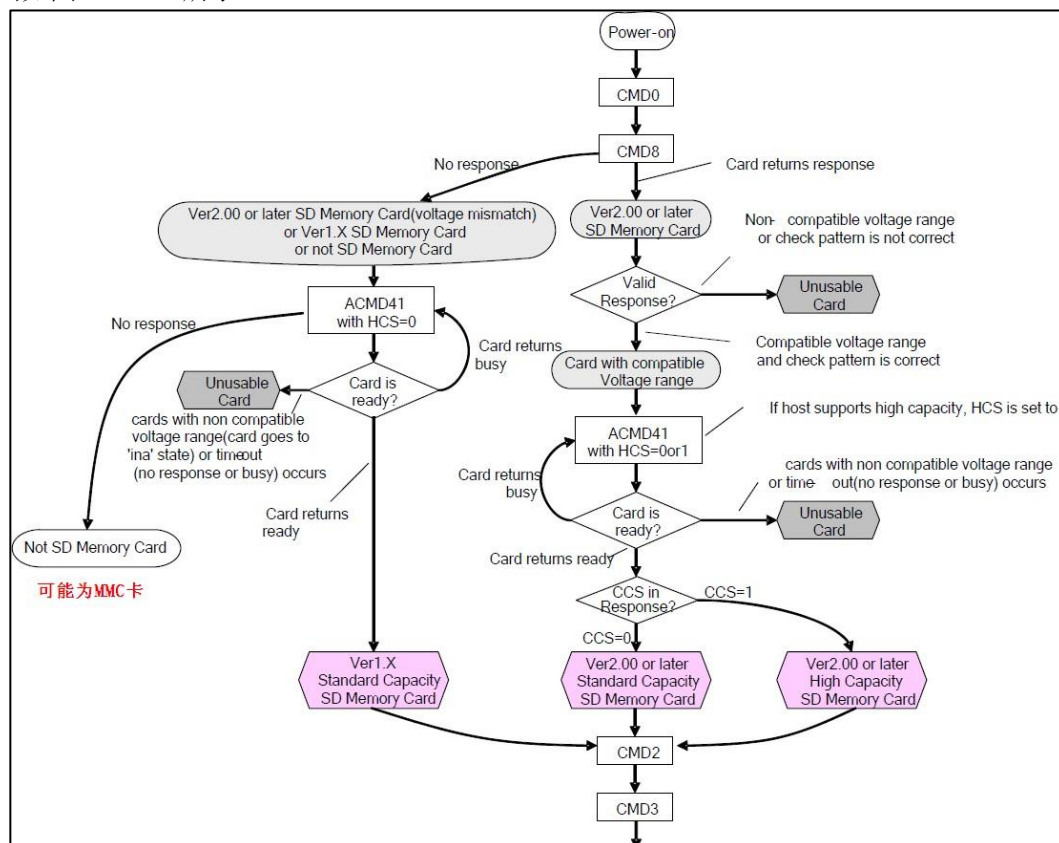


图 49.2.5.1 SD 卡初始化流程（Card Initialization and Identification Flow (SD mode)）

从图中，我们看到，不管什么卡（这里我们将卡分为 4 类：SD2.0 高容量卡（SDHC，最大 32G），SDV2.0 标准容量卡（SDSC，最大 2G），SD1.x 卡和 MMC 卡），首先我们要执行的是卡上电（需要设置 SDIO_POWER[1:0]=11），上电后发送 CMD0，对卡进行软复位，之后发送 CMD8 命令，用于区分 SD 卡 2.0，只有 2.0 及以后的卡才支持 CMD8 命令，MMC 卡和 V1.x 的卡，是不支持该命令的。CMD8 的格式如表 49.2.5.1 所示：

| 位序 | 47 | 46 | [45:40] | [39:20] | [19:16] | [15:8] | [7:1] | 0 |
|-----|-----|-----|----------|----------|---------|--------|-------|-----|
| 占用位 | 1 | 1 | 6 | 20 | 4 | 8 | 7 | 1 |
| 命令值 | '0' | '1' | '001000' | '00000h' | x | x | x | '1' |
| 描述 | 起始位 | 传输位 | 命令索引 | 保留位 | 电源(VHS) | 校验 | CRC7 | 结束位 |

表 49.2.5.1 CMD8 命令格式

这里，我们需要在发送 CMD8 的时候，通过其带的参数我们可以设置 VHS 位，以告诉 SD 卡，主机的供电情况，VHS 位定义如表 49.2.5.2 所示：

| Voltage Supplied | Value Definition |
|------------------|--------------------------------|
| 0000b | Not Defined |
| 0001b | 2.7-3.6V |
| 0010b | Reserved for Low Voltage Range |
| 0100b | Reserved |
| 1000b | Reserved |
| Others | Not Defined |

表 49.2.5.2 VHS 位定义

这里我们使用参数 0X1AA，即告诉 SD 卡，主机供电为 2.7~3.6V 之间，如果 SD 卡支持 CMD8，且支持该电压范围，则会通过 CMD8 的响应（R7）将参数部分原本返回给主机，如果不支持 CMD8，或者不支持这个电压范围，则不响应。

在发送 CMD8 后，发送 ACMD41（注意发送 ACMD41 之前要先发送 CMD55），来进一步确认卡的操作电压范围，并通过 HCS 位来告诉 SD 卡，主机是不是支持大容量卡（SDHC）。ACMD41 的命令格式如表 49.2.5.3 所示：

| ACMD INDEX | type | argument | resp | abbreviation | command description |
|------------|------|--|------|-----------------|--|
| ACMD41 | bcr | [31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V _{DD} Voltage Window(OCR[23:0]) | R3 | SD_SEND_OP_COND | Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30]. |

表 49.2.5.3 ACMD41 命令格式

ACMD41 得到的响应(R3)包含 SD 卡 OCR 寄存器内容,OCR 寄存器内容定义如表 49.2.5.4 所示：

| OCR bit position | OCR Fields Definition | VDD Voltage Window |
|--|--|--------------------|
| 0-6 | reserved | |
| 7 | Reserved for Low Voltage Range | |
| 8-14 | reserved | |
| 15 | 2.7-2.8 | |
| 16 | 2.8-2.9 | |
| 17 | 2.9-3.0 | |
| 18 | 3.0-3.1 | |
| 19 | 3.1-3.2 | |
| 20 | 3.2-3.3 | |
| 21 | 3.3-3.4 | |
| 22 | 3.4-3.5 | |
| 23 | 3.5-3.6 | |
| 24-29 | reserved | |
| 30 | Card Capacity Status (CCS) ¹ | |
| 31 | Card power up status bit (busy) ² | |
| 1) This bit is valid only when the card power up status bit is set. | | |
| 2) This bit is set to LOW if the card has not finished the power up routine. | | |

表 49.2.5.4 OCR 寄存器定义

对于支持 CMD8 指令的卡，主机通过 ACMD41 的参数设置 HCS 位为 1，来告诉 SD 卡主机支 SDHC 卡，如果设置为 0，则表示主机不支持 SDHC 卡，SDHC 卡如果接收到 HCS 为 0，则永远不会反回卡就绪状态。对于不支持 CMD8 的卡，HCS 位设置为 0 即可。

SD 卡在接收到 ACMD41 后，返回 OCR 寄存器内容，如果是 2.0 的卡，主机可以通过判断 OCR 的 CCS 位来判断是 SDHC 还是 SDSC；如果是 1.x 的卡，则忽略该位。OCR 寄存器的最后一个位用于告诉主机 SD 卡是否上电完成，如果上电完成，该位将会被置 1。

对于 MMC 卡，则不支持 ACMD41，不响应 CMD55，对 MMC 卡，我们只需要在发送 CMD0 后，在发送 CMD1（作用同 ACMD41），检查 MMC 卡的 OCR 寄存器，实现 MMC 卡的初始化。

至此，我们便实现了对 SD 卡的类型区分，图 49.2.5.1 中，最后发送了 CMD2 和 CMD3 命令，用于获得卡 CID 寄存器数据和卡相对地址（RCA）。

发送 CMD2，用于获得 CID 寄存器的数据，而 CID 寄存器数据各位定义如表 49.2.5.5 所示：

| Name | Field | Width | CID-slice |
|-----------------------|-------|-------|-----------|
| Manufacturer ID | MID | 8 | [127:120] |
| OEM/Application ID | OID | 16 | [119:104] |
| Product name | PNM | 40 | [103:64] |
| Product revision | PRV | 8 | [63:56] |
| Product serial number | PSN | 32 | [55:24] |
| reserved | -- | 4 | [23:20] |
| Manufacturing date | MDT | 12 | [19:8] |
| CRC7 checksum | CRC | 7 | [7:1] |
| not used, always 1 | - | 1 | [0:0] |

表 49.3.1.5 卡 CID 寄存器位定义

SD 卡在收到 CMD2 后，将返回 R2 长响应（136 位），其中包含 128 位有效数据（CID 寄存器内容），存放在 SDIO_RESP1~4 等 4 个寄存器里面。通过读取这四个寄存器，就可以获得 SD 卡的 CID 信息。

CMD3，用于设置卡相对地址（RCA，必须为非 0），对于 SD 卡（非 MMC 卡），在收到 CMD3 后，将返回一个新的 RCA 给主机，方便主机寻址。RCA 的存在允许一个 SDIO 接口挂多个 SD 卡，通过 RCA 来区分主机要操作的是哪个卡。而对于 MMC 卡，则不是由 SD 卡自动返回 RCA，而是主机主动设置 MMC 卡的 RCA，即通过 CMD3 带参数（高 16 位用于 RCA 设置），实现 RCA 设置。同样 MMC 卡也支持一个 SDIO 接口挂多个 MMC 卡，不同于 SD 卡的是所有的 RCA 都是由主机主动设置的，而 SD 卡的 RCA 则是 SD 卡发给主机的。

在获得卡 RCA 之后，我们便可以发送 CMD9（带 RCA 参数），获得 SD 卡的 CSD 寄存器内容，从 CSD 寄存器，我们可以得到 SD 卡的容量和扇区大小等十分重要的信息。CSD 寄存器我们在这里就不详细介绍了，关于 CSD 寄存器的详细介绍，请大家参考《SD 卡 2.0 协议.pdf》。

至此，我们的 SD 卡初始化基本就结束了，最后通过 CMD7 命令，选中我们要操作的 SD 卡，即可开始对 SD 卡的读写操作了，SD 卡的其他命令和参数，我们这里就不再介绍了，请大家参考《SD 卡 2.0 协议.pdf》，里面有非常详细的介绍。

49.3 硬件设计

1. 例程功能

本章实验功能简介：开机的时候先初始化 SD 卡，如果 SD 卡初始化完成，则提示 LCD 初始化成功。按下 KEY0，读取 SD 卡扇区 0 的数据，然后通过串口发送到电脑。如果没初始化通过，则在 LCD 上提示初始化失败。LED0 来指示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 按键
KEY0 - PE4
- 3) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) microSD Card

3. 原理图

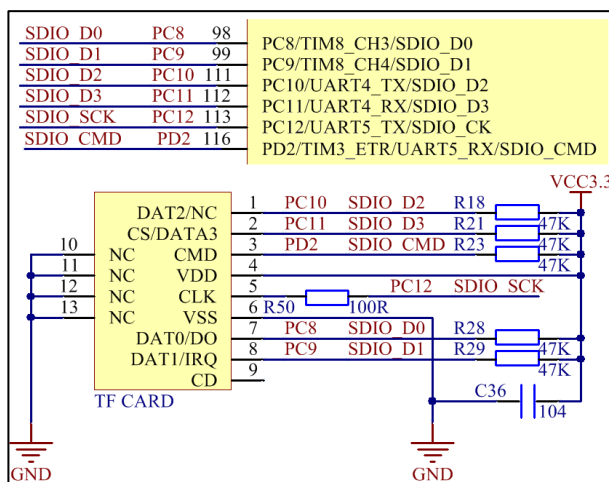


图 49.3.1 SD 卡接口与 STM32F1 连接原理图

SD 卡座在 JTAG 插座附近，SD 卡座与开发板的连接在开发板上是直接连接在一起的，硬件上不需要任何改动。大家准备好 SD 卡就可以开始我们的程序设计和验证了。

49.4 程序设计

49.4.1 SD 卡的 HAL 库驱动

STM32 的 HAL 库为 SD 卡操作封装了一些函数，主要存放在 stm32f1xx_hal_sd.c/h 下，下面我们来分析我们主要使用到的几个函数。

1. HAL_SD_Init 函数

要使用一个外设首先要对它进行初始化，所以先看 sdio 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_SD_Init(SD_HandleTypeDef *hsd)
```

● 函数描述：

根据 SD 参数，初始化 SDIO 外设以便后续操作 SD 卡。

● 函数形参：

形参 1 是 SD 卡的句柄，结构体类型是 SD_HandleTypeDef，我们不使用 USE_HAL_SD_REGISTER_CALLBACKS 宏来拓展 SD 卡的自定义函数，精简后其定义如下：

```
/**
 * @brief SD 操作句柄结构体定义
 */
typedef struct
{
    SD_TypeDef                *Instance;           /* SD 相关寄存器基地址 */
    SD_InitTypeDef            Init;                /* SDIO 初始化变量 */
    HAL_LockTypeDef            Lock;               /* 互斥锁，用于解决外设访问冲突 */
    uint8_t                   *pTxBuffPtr;        /* SD 发送数据指针 */
    uint32_t                   TxXferSize;        /* SD 发送缓存按字节数的大小 */
    uint8_t                   *pRxBuffPtr;        /* SD 接收数据指针 */
    uint32_t                   RxXferSize;        /* SD 接收缓存按字节数的大小 */
    __IO uint32_t              Context;            /* HAL 库对 SD 卡的操作阶段 */
    __IO HAL_SD_StateTypeDef   State;              /* SD 卡操作状态 */
    __IO uint32_t              ErrorCode;          /* SD 卡错误代码 */
    DMA_HandleTypeDef         *hdmatx;            /* SD DMA 数据发送指针 */
    DMA_HandleTypeDef         *hdmarx;            /* SD DMA 数据接收指针 */
    HAL_SD_CardInfoTypeDef     SdCard;            /* SD 卡信息的 */
    uint32_t                   CSD[4];            /* 保存 SD 卡 CSD 寄存器信息 */
    uint32_t                   CID[4];            /* 保存 SD 卡 CID 寄存器信息 */
} SD_HandleTypeDef;
```

上面的初始化结构体中 HAL_SD_CardInfoTypeDef 用于初始化后提取卡信息，包括卡类型、容量等参数。

```
typedef struct
{
    uint32_t CardType;           /* 存储卡类型标记：标准卡、高速卡 */
    uint32_t CardVersion;       /* 存储卡版本 */
    uint32_t Class;             /* 卡类型 */
    uint32_t RelCardAdd;        /* 卡相对地址 */
    uint32_t BlockNbr;         /* 卡存储块数 */
    uint32_t BlockSize;         /* SD 卡每个存储块大小 */
    uint32_t LogBlockNbr;       /* 以块表示的卡逻辑容量 */
    uint32_t LogBlockSize;      /* 以字节为单位的逻辑块大小 */
}HAL_SD_CardInfoTypeDef;
```

● 函数返回值：

HAL_StatusTypeDef 枚举类型的值，有 4 个，分别是 HAL_OK 表示成功，HAL_ERROR 表示错误，HAL_BUSY 表示忙碌，HAL_TIMEOUT 超时。后续遇到该结构体也是一样的。只有返回 HAL_OK 才是正常的卡初始化状态，遇到其它状态则需要结合硬件分析一下代码。

2. HAL_SD_ConfigWideBusOperation 函数

SD 卡上电后默认使用 1 位数据总线进行数据传输，卡如果允许，可以在初始化完成后重新设置 SD 卡的数据位宽以加快数据传输过程：

```
HAL_StatusTypeDef HAL_SD_ConfigWideBusOperation(SD_HandleTypeDef *hsd, uint32_t WideMode);
```

● 函数描述：

这个函数用于设置数据总线格式的数据宽度，用于加快卡的数据访问速度，当然前提是硬件连接和卡本身能支持这样操作。

● 函数形参：

形参 1 是 SD 卡的句柄，结构体类型是 SD_HandleTypeDef，此函数需要在 SDIO 初始化结束后才能使用，我们需要通过使用初始化后的 SDIO 结构体的句柄访问外设。

形参 2 是总线宽度，根据函数的形参检查规则我们可知它实际上只有三个可选值：

```
#define SDIO_BUS_WIDE_1B ((uint32_t)0x00000000U)
#define SDIO_BUS_WIDE_4B SDIO_CLKCR_WIDBUS_0
#define SDIO_BUS_WIDE_8B SDIO_CLKCR_WIDBUS_1
```

对于 STM32F103 实际不支持 8B 模式，使用时需要特别注意。

● 函数返回值：

HAL_StatusTypeDef 类型的函数，返回值同样是需要获取到 HAL_OK 表示成功。

3. HAL_SD_ReadBlocks 函数

SD 卡初始化后从 SD 卡的指定扇区读数据：

```
HAL_StatusTypeDef HAL_SD_ReadBlocks(SD_HandleTypeDef *hsd, uint8_t *pData,
uint32_t BlockAdd, uint32_t NumberOfBlocks, uint32_t Timeout);
```

这个函数是直接读取，不使用硬件中断。

● 函数描述：

从 SD 卡的指定扇区读取一定数量的数据。

● 函数形参：

形参 1 是 SD 卡的句柄，结构体类型是 SD_HandleTypeDef，此函数需要在 SDIO 初始化结束后才能使用，我们需要通过使用初始化后的 SDIO 结构体的句柄访问外设。

形参 2 pData 是一个指向 8 位类型的数据指针缓冲，它用于接收我们需要的数据。

形参 3 BlockAdd 指向我们需要访问的数据扇区，对于任意的存储都是类似的，像 SD 卡这样的大存储块也同样是位置标识来访问不同的数据。

形参 4 NumberOfBlocks 对应的是我们本次要从指定扇区读取的字节数。

形参 5 Timeout 表示读的超时时间。HAL 库驱动在达到超时时间前还没读到数据会进行重试和等待，达到超时时间后或者本次读取成功才退出本次操作。

● 函数返回值：

HAL_StatusTypeDef 类型的函数，返回值同样是需要获取到 HAL_OK 表示成功。

类似功能的函数还有，我们的例程没有使用 DMA 和中断方式，故不使用以下两个接口：

```
HAL_StatusTypeDef HAL_SD_ReadBlocks_IT(SD_HandleTypeDef *hsd, uint8_t *pData,
                                         uint32_t BlockAdd, uint32_t NumberOfBlocks);
HAL_StatusTypeDef HAL_SD_ReadBlocks_DMA(SD_HandleTypeDef *hsd, uint8_t *pData,
                                         uint32_t BlockAdd, uint32_t NumberOfBlocks);
```

它们分别使用了中断方式和 DMA 方式来实现类似的功能，它们的调用非常相似，这里就不重复介绍了，大家查看对应的函数实现即可。

4. HAL_SD_WriteBlocks 函数

SD 卡初始化后，在 SD 卡的指定扇区写入数据：

```
HAL_StatusTypeDef HAL_SD_WriteBlocks(SD_HandleTypeDef *hsd, uint8_t *pData,
                                       uint32_t BlockAdd, uint32_t NumberOfBlocks, uint32_t Timeout);
```

- **函数描述：**

向 SD 卡的指定扇区写入一定数量的数据。

- **函数形参：**

形参 1 是 SD 卡的句柄，结构体类型是 SD_HandleTypeDef，此函数需要在 SDIO 初始化结束后才能使用，我们需要通过使用初始化后的 SDIO 结构体的句柄访问外设。

形参 2 pData 是一个指向 8 位类型的数据指针缓冲，它用于存放我们需要写入的数据。

形参 3 BlockAdd 指向我们需要访问的数据扇区，对于任意的存储都是类似的，像 SD 卡这样的大存储块也是通过位置标识来访问不同的数据。

形参 4 NumberOfBlocks 对应的是我们本次要向指定扇区写入的字节数。

形参 5 Timeout 表示写动作的超时时间。HAL 库驱动在达到超时时间前还没读到数据会进行重试和等待，达到超时时间后或者本次写入成功才退出本次操作。

- **函数返回值：**

HAL_StatusTypeDef 类型的函数，返回值同样是需要获取到 HAL_OK 表示成功。

类似于读函数，写函数同样有中断版本，我们的例程没有使用 DMA 和中断方式，故不使用以下两个接口：

```
HAL_StatusTypeDef HAL_SD_WriteBlocks_IT(SD_HandleTypeDef *hsd, uint8_t *pData,
                                         uint32_t BlockAdd, uint32_t NumberOfBlocks);
HAL_StatusTypeDef HAL_SD_WriteBlocks_DMA(SD_HandleTypeDef *hsd, uint8_t *pData,
                                         uint32_t BlockAdd, uint32_t NumberOfBlocks);
```

它们分别使用了中断方式和 DMA 方式来实现类似的功能，它们的调用非常相似，这里就不重复介绍了，大家查看对应的函数实现即可。

5. HAL_SD_GetCardInfo 函数

SD 卡初始化后，根据设备句柄读 SD 卡的相关状态信息：

```
HAL_StatusTypeDef HAL_SD_GetCardInfo(SD_HandleTypeDef *hsd, HAL_SD_CardInfoTypeDef *pCardInfo);
```

- **函数描述：**

获取 SD 卡信息。

- **函数形参：**

形参 1 是 SD 卡的句柄，结构体类型是 SD_HandleTypeDef，此函数需要在 SDIO 初始化结束后才能使用，我们需要通过使用初始化后的 SDIO 结构体的句柄访问外设。

形参 2 pCardInfo 是一个指向 HAL_SD_CardInfoTypeDef 类型的数据指针缓冲，它用于接收卡信息，包括卡类型、容量等参数。

- **函数返回值：**

HAL_StatusTypeDef 类型的函数，返回值同样是需要获取到 HAL_OK 表示成功。

SDIO 驱动 SD 卡配置步骤

1) 初始化 SD 卡相关 GPIO 口

具体用到哪些 GPIO 口可以参考原理图，特别注意：IO 口的模式要设置为复用推挽输出。

2) SD 卡初始化

这里需要使能 SDIO 时钟并且设置 SDIO 的工作参数，使用 HAL_SD_Init 函数进行初始化。

3) 重新设置 SDIO 总线位宽

这个步骤是可选的，HAL 库通过 HAL_SD_WideBusOperation_config 函数实现 SDIO 总线位宽改变。通过 HAL_SD_Init 函数初始化后，总线位宽默认为 1 位，所以要想获得比较快的速度，就需要改变位宽。

4) 实现 SD 卡读取&写入函数

在初始化 SDIO 和 SD 卡完成以后，我们就可以访问 SD 卡了，HAL 库提供了两个基本的 SD 卡读写函数：HAL_SD_ReadBlocks 和 HAL_SD_WriteBlocks，用于读取和写入 SD 卡。我们对这两个函数再进行一次封装，以便更好的适配文件系统，再封装后，我们使用：sd_read_disk 来读取 SD 卡，使用：sd_write_disk 来写入 SD 卡，详见本例程源码。

49.4.2 程序流程图

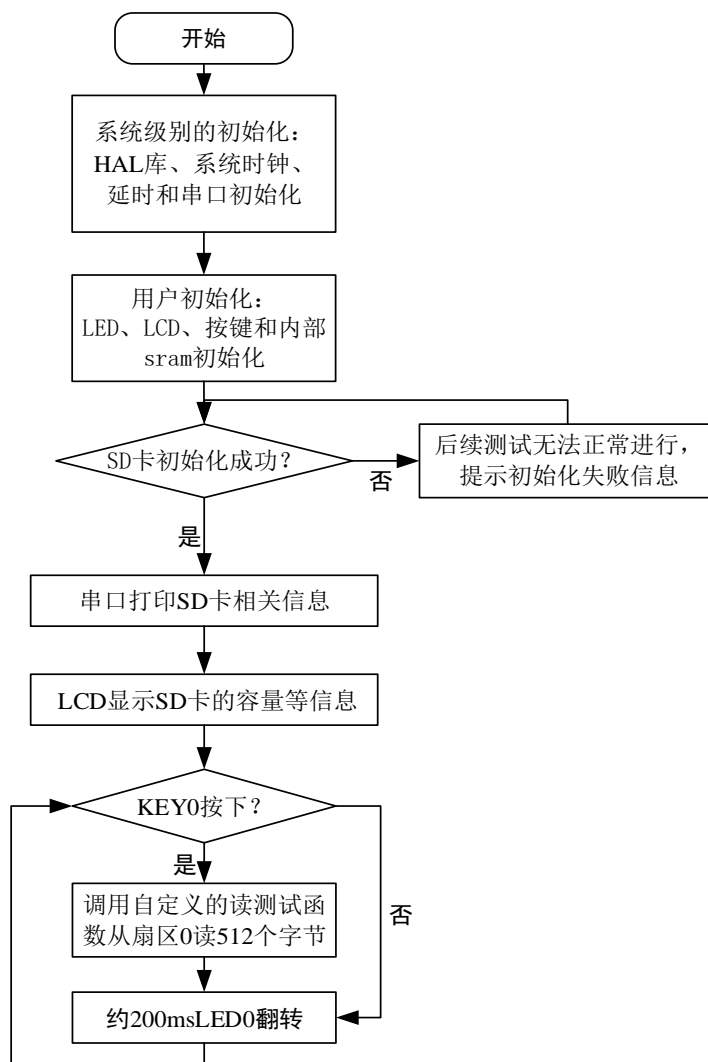


图 49.4.2.1 SD 读写实验程序流程图

49.4.3 程序解析

1. SDIO 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SDIO 驱动源码包括两个文件：sdio_sdcard.c 和 sdio_sdcard.h。

sdio_sdcard.h 我们主要介绍一下 GPIO 宏定义，根据我们 STM32 的复用功能和我们的硬件设计，我们把用到的管脚用宏定义，需要更换其它的引脚时也可以通过修改宏实现快速移植，

它们列出如下：

```
/* SDIO 的信号线：SD_D0 ~ SD_D3/SD_CLK/SD_CMD 引脚 定义
 * 如果你使用了其他引脚做 SDIO 的信号线，修改这里写定义即可适配。
 */
#define SD_D0_GPIO_PORT      GPIOC
#define SD_D0_GPIO_PIN      GPIO_PIN_8
#define SD_D0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define SD_D1_GPIO_PORT      GPIOC
#define SD_D1_GPIO_PIN      GPIO_PIN_9
#define SD_D1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define SD_D2_GPIO_PORT      GPIOC
#define SD_D2_GPIO_PIN      GPIO_PIN_10
#define SD_D2_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define SD_D3_GPIO_PORT      GPIOC
#define SD_D3_GPIO_PIN      GPIO_PIN_11
#define SD_D3_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define SD_CLK_GPIO_PORT      GPIOC
#define SD_CLK_GPIO_PIN      GPIO_PIN_12
#define SD_CLK_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define SD_CMD_GPIO_PORT      GPIOD
#define SD_CMD_GPIO_PIN      GPIO_PIN_2
#define SD_CMD_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)
```

sdio_sdcard.c 我们主要介绍三个函数：sd_init、sd_read_disk 和 sd_write_disk。

1) sd_init 函数

sd_init 的设计就比较简单了，我们只需要填充 SDIO 结构体的控制句柄，然后使用 HAL 库的初始化驱动即可。

```
/**
 * @brief      初始化 SD 卡
 * @param      无
 * @retval     返回值:0 初始化正确；其他值，初始化错误
 */
uint8_t sd_init(void)
{
    uint8_t SD_Error;
    /* 初始化时的时钟不能大于 400KHZ */
    g_sdcard_handler.Instance = SDIO;
    g_sdcard_handler.Init.ClockEdge = SDIO_CLOCK_EDGE_RISING; /* 上升沿 */
    /* 不使用 bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK */
    g_sdcard_handler.Init.ClockBypass = SDIO_CLOCK_BYPASS_DISABLE;
    /* 空闲时不关闭时钟电源 */
    g_sdcard_handler.Init.ClockPowerSave = SDIO_CLOCK_POWER_SAVE_DISABLE;
    g_sdcard_handler.Init.BusWide = SDIO_BUS_WIDE_1B; /* 1 位数据线 */
    g_sdcard_handler.Init.HardwareFlowControl =
        SDIO_HARDWARE_FLOW_CONTROL_ENABLE; /* 开启硬件流控 */
    /* SD 传输时钟频率最大 25MHZ */
    g_sdcard_handler.Init.ClockDiv = SDIO_TRANSFER_CLK_DIV;
    SD_Error = HAL_SD_Init(&g_sdcard_handler);
    if (SD_Error != HAL_OK)
    {
        return 1;
    }
    /* 使能宽总线模式，即 4 位总线模式，加快读取速度 */
    SD_Error = HAL_SD_ConfigWideBusOperation(&g_sdcard_handler, SDIO_BUS_WIDE_4B);
    if (SD_Error != HAL_OK)
    {
        return 2;
    }
    return 0;
}
```

}

2) sd_read_disk 函数

这个函数比较简单，实际上我们使用它来对 HAL 库的读函数 HAL_SD_ReadBlocks 进行了二次封装，并在最后加入了状态判断以使后续操作（实际上这部分代码也可以省略），直接根据读函数返回值自己作其它处理。为了保护 SD 卡的数据操作，我们在进行操作时暂时关闭了中断以防止数据读过程发生意外。

```
uint8_t sd_read_disk(uint8_t *pbuf, uint32_t saddr, uint32_t cnt)
{
    uint8_t sta = HAL_OK;
    uint32_t timeout = SD_TIMEOUT;
    long long lsector = saddr;
    __disable_irq();
    /* 关闭总中断(POLLING 模式, 严禁中断打断 SDIO 读写操作!!!) */
    sta = HAL_SD_ReadBlocks(&g_sdcard_handler, (uint8_t *)pbuf, lsector,
                           cnt, SD_TIMEOUT); /* 多个 sector 的读操作 */

    /* 等待 SD 卡读完 */
    while (get_sd_card_state() != SD_TRANSFER_OK)
    {
        if (timeout-- == 0)
        {
            sta = SD_TRANSFER_BUSY;
        }
    }
    __enable_irq(); /* 开启总中断 */
    return sta;
}
```

3) sd_write_disk 函数

这个函数比较简单，实际上我们使用它来对 HAL 库的读函数 HAL_SD_WriteBlocks 进行了二次封装，并在最后加入了状态判断以使后续操作（实际上这部分代码也可以省略），直接根据读函数返回值自己作其它处理。为了保护 SD 卡的数据操作，我们在进行操作时暂时关闭了中断以防止数据写过程发生意外。

```
uint8_t sd_write_disk(uint8_t *pbuf, uint32_t saddr, uint32_t cnt)
{
    uint8_t sta = HAL_OK;
    uint32_t timeout = SD_TIMEOUT;
    long long lsector = saddr;
    __disable_irq(); /* 关闭总中断(POLLING 模式, 严禁中断打断 SDIO 读写操作!!!) */
    sta = HAL_SD_WriteBlocks(&g_sdcard_handler, (uint8_t *)pbuf, lsector,
                             cnt, SD_TIMEOUT); /* 多个 sector 的写操作 */

    /* 等待 SD 卡写完 */
    while (get_sd_card_state() != SD_TRANSFER_OK)
    {
        if (timeout-- == 0)
        {
            sta = SD_TRANSFER_BUSY;
        }
    }
    __enable_irq(); /* 开启总中断 */
    return sta;
}
```

2. main.c 代码

main.c 里面的代码比较简单，按照我们程序流程图的思路进行设计。为了方便测试，我们编写了 sd_test_read()\sd_test_write()\show_sdcard_info ()三个函数分别用于读写测试和卡信息打印，主要就是调用 sdio_sdcard.c 文件中的函数接口，代码也比较容易看懂，这里就不单独介绍这几个函数了，大家查看光盘中的源代码即可。

最后，我们编写的主函数如下：

```
int main(void)
```

```

{
    uint8_t key;
    uint32_t sd_size;
    uint8_t t = 0;
    uint8_t *buf;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32F103", RED);
    lcd_show_string(30, 70, 200, 16, 16, "SD TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Read Sector 0", RED);

    while (sd_init()) /* 检测不到 SD 卡 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "SD Card Error!", RED);
        delay_ms(500);
        lcd_show_string(30, 130, 200, 16, 16, "Please Check! ", RED);
        delay_ms(500);
        LED0_TOGGLE(); /* 红灯闪烁 */
    }

    show_sdcard_info(); /* 打印 SD 卡相关信息 */
    lcd_show_string(30, 130, 200, 16, 16, "SD Card OK ", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "SD Card Size: MB", BLUE);
    lcd_show_num(30+13*8, 150, SD_TOTAL_SIZE_MB(&g_sdcard_handler), 5, 16, BLUE);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES) /* KEY0 按下了 */
        {
            sd_test_read(0, 1); /* 从 0 扇区读取 1*512 字节的内容 */
        }

        t++;
        delay_ms(10);

        if (t == 20)
        {
            LED0_TOGGLE(); /* 红灯闪烁 */
            t = 0;
        }
    }
}

```

main 函数先初始化相关外设和 SD 卡，初始化成功后，通过调用 show_sdcard_info 函数输出 SD 卡相关信息（卡类型、制造商 ID、卡相对地址、容量和块大小等），并在 LCD 上显示 SD 卡容量。然后进入死循环，如果有按键 KEY0 按下，则通过 sd_test_read 函数读取 SD 卡的扇区 0（物理磁盘，扇区 0），并将数据通过串口打印出来。

49.5 下载验证

在代码编译成功之后，我们通过下载代码到正点原子战舰 STM32F103 上，我们测试使用的是 16GB 标有“SDHC”标志的卡，安装方法如图 49.5.1 所示：



图 49.5.1 测试用的 microSD 卡与开发板的连接方式
SD 卡成功初始化后，LCD 显示本程序的一些必要信息，如图 49.5.2 所示：

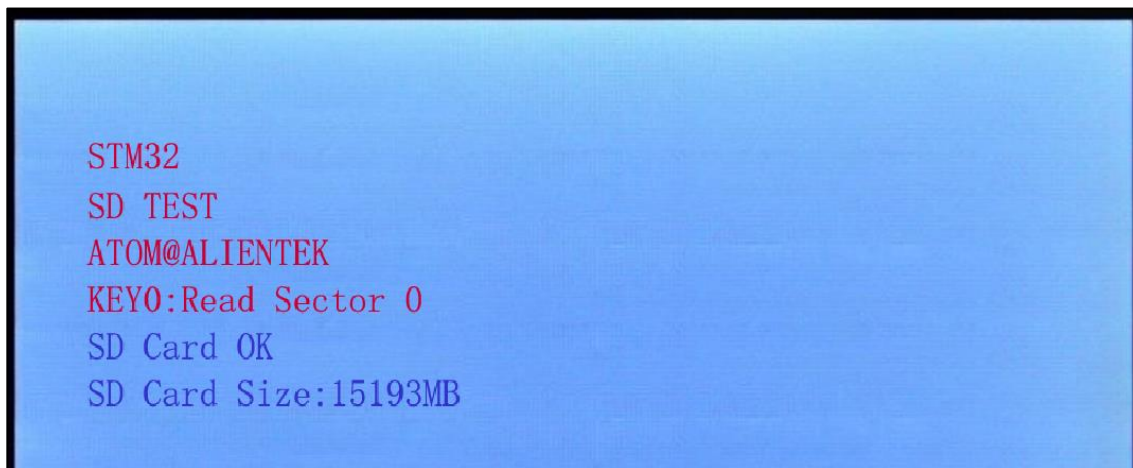


图 49.5.2 程序运行效果图

在进入测试的主循环前，我们如果已经通过 USB 连接开发板的串口 1 和电脑，可以看到串口端打印出 SD 卡的相关信息（也可以在接好 SD 卡后按 Reset 复位开发板），我们测试使用的是 16GB 标有“SDHC”标志的卡，SD 卡成功初始化后的信息，如图 49.5.3：



图 49.5.3 串口助手打印信息

可见我们用程序读到的 SD 卡信息与我们使用的 SD 卡一致。伴随 LED0 的不停闪烁，提示程序在运行。此时，我们按下 KEY0，调用我们编写的 SD 卡测试函数，这里我们只用到了读函数，写函数的测试大家可以添加代码进行演示。按下 KEY0 后，LCD 显示信息如图 49.5.4 所示，读取到的扇区数据将通过串口打印出来，具体数据情况如图 49.5.5 所示：

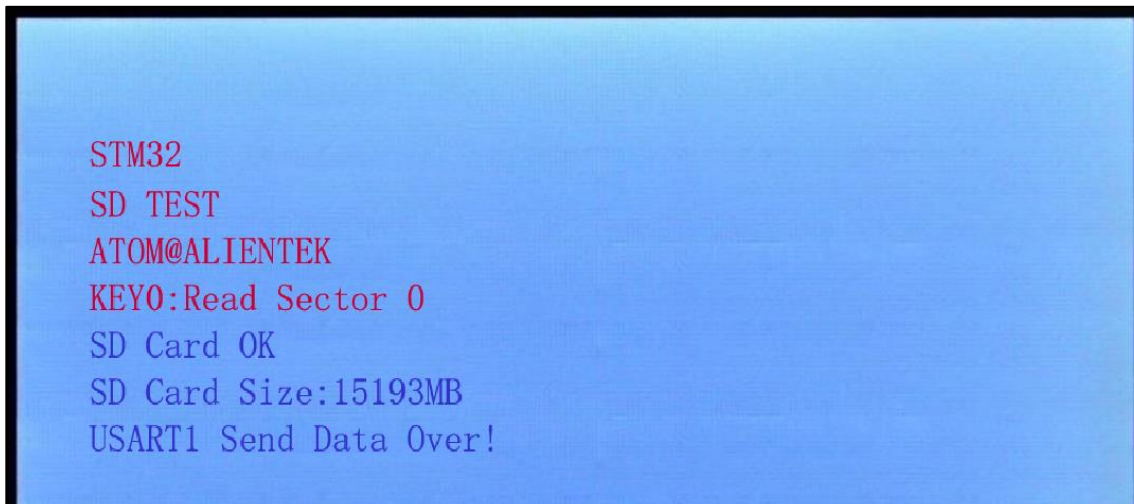


图 49.5.4 按下 KEY0 的开发板界面

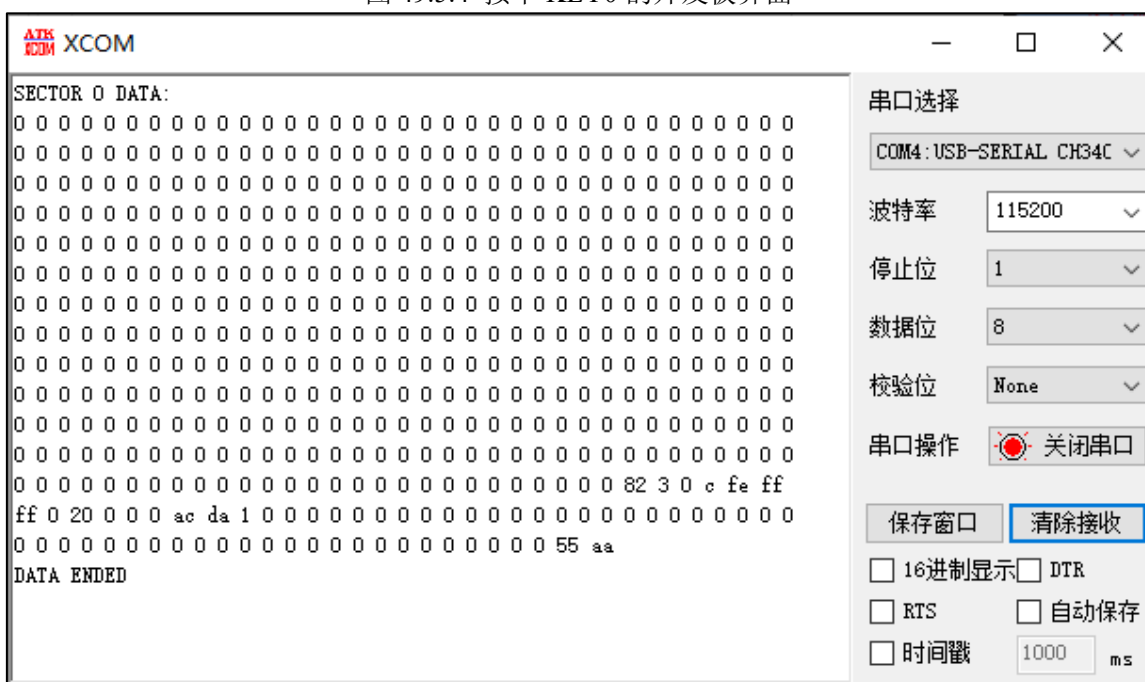


图 49.5.5 串口调试助手显示按下 KEY0 后读取到的信息

这里请大家注意，不同 SD 卡，读出来的扇区 0 是不尽相同的，所以不要因为你读出来的数据和上图不同而感到惊讶。另外，例程还支持 USMART 测试，这里就不做介绍了，大家可自行验证下。

第五十章 FATFS 实验

上一章，我们学习了 SD 卡的使用，并实现了简单的读写扇区功能。在电脑上我们的资料常以文件的形式保存，通过文件名我们可以快速对自己的文件数据等进行分类。对于 SD 卡这种容量可以达到非常大的存储介质，按扇区去管理数据已经变得不方便，我们希望单片机也可以像电脑一样方便地用文件的形式去管理，在需要做数据采集的场合也会更加便利。

本章，我们将介绍 FATFS 这个软件工具，利用它在 STM32 上实现类似电脑上的文件管理功能，方便管理 SD 卡上的数据，并设计例程在 SD 卡上生成文件并对文件实现读写操作。本章分为如下几个部分：

- 50.1 FATFS 简介
- 50.2 硬件设计
- 50.3 软件设计
- 50.4 下载验证

50.1 FATFS 简介

FATFS 是一个完全免费开源的 FAT/exFAT 文件系统模块，专门为小型的嵌入式系统而设计。它完全用标准 C 语言（ANSI C89）编写，所以具有良好的硬件平台独立性，只需做简单的修改就可以移植到 8051、PIC、AVR、ARM、Z80、RX 等系列单片机上。它支持 FAT12、FAT16 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位单片机做了优化。

FATFS 的特点有：

- Windows/dos 系统兼容的 FAT/exFAT 文件系统
- 独立于硬件平台，方便跨硬件平台移植
- 代码量少、效率高
- 多种配置选项
 - ✧ 支持多卷（物理驱动器或分区，最多 10 个卷）
 - ✧ 多个 ANSI/OEM 代码页包括 DBCS
 - ✧ 支持长文件名、ANSI/OEM 或 Unicode
 - ✧ 支持 RTOS
 - ✧ 支持多种扇区大小
 - ✧ 只读、最小化的 API 和 I/O 缓冲区等
 - ✧ 新版的 exFAT 文件系统，突破了原来 FAT32 对容量管理 32GB 的上限，可支持更巨大容量的存储器

FATFS 的这些特点，加上免费、开源的原则，使得 FATFS 应用非常广泛。FATFS 模块的层次结构如图 50.1.1 所示：

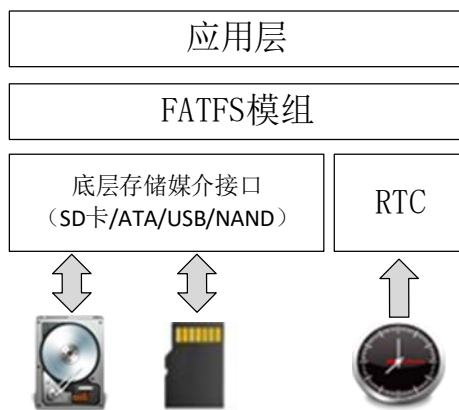


图 50.1.1 FATFS 层次结构图

最顶层是应用层,使用者无需理会 FATFS 的内部结构和复杂的 FAT 协议,只需要调用 FATFS 模块提供给用户的一系列应用接口函数,如 `f_open`, `f_read`, `f_write` 和 `f_close` 等,就可以像在 PC 上读 / 写文件那样简单。

中间层 FATFS 模块,实现了 FAT 文件读 / 写协议。FATFS 模块提供的是 `ff.c` 和 `ff.h`。除非有必要,使用者一般不用修改,使用时将头文件直接包含进去即可。

需要我们编写移植代码的是 FATFS 模块提供的底层接口,它包括存储媒介读 / 写接口 (`diskI/O`) 和供给文件创建修改时间的实时时钟。

FATFS 的源码及英文详述,大家可以在: http://elm-chan.org/fsw/ff/00index_e.html 网站下载到,教程目前使用的版本为 R0.14b。本章我们就使用最新版本的 FATFS 作为介绍,下载最新版本的 FATFS 软件包,解压后可以得到两个文件夹: `documents` 和 `source`。`documents` 里面主要是对 FATFS 的介绍,而 `source` 里面才是我们需要的源码。`source` 文件夹详情,如表 50.1.1 所示:

| 文件名 | 作用简述 | 备注 |
|--------------------------|--|----------|
| <code>diskio.h</code> | FATFS 和 <code>diskI/O</code> 模块公用的包含文件 | 与硬件平台无关 |
| <code>ff.c</code> | FATFS 模块 | |
| <code>ff.h</code> | FATFS 和应用模块公用的包含文件 | |
| <code>ffconf.h</code> | FATFS 模块配置文件,宏定义对应的功能,代码中都有说明,具体的配置范围可以见官方配置说明 http://elm-chan.org/fsw/ff/doc/config.html | |
| <code>ffsystem.c</code> | 根据是否有操作系统来修改这个文件 | |
| <code>ffunicode.c</code> | 可选,根据 <code>ffconf.h</code> 的配置,进行 Unicode 编码转换 | |
| <code>diskio.c</code> | FATFS 和 <code>diskI/O</code> 模块接口层文件,需要根据硬件修改这部分的代码 | 硬件平台相关代码 |

表 50.1.1 source 文件夹详情表

FATFS 模块在移植的时候,我们一般只需要修改 2 个文件,即 `ffconf.h` 和 `diskio.c`。FATFS 模块的所有配置项都是存放在 `ffconf.h` 里面,我们可以通过配置里面的一些选项,来满足自己的需求。接下来我们介绍几个重要的配置选项。

- 1) `FF_FS_TINY`。这个选项在 R0.07 版本中开始出现,之前的版本都是以独立的 C 文件出现 (FATFS 和 TinyFATFS),有了这个选项之后,两者整合在一起了,使用起来更方便。我们使用 FATFS,所以把这个选项定义为 0 即可。
- 2) `FF_FS_READONLY`。这个用来配置是不是只读,本章我们需要读写都用,所以这里设置为 0 即可。
- 3) `FF_USE_STRFUNC`。这个用来设置是否支持字符串类操作,比如 `f_putc`, `f_puts` 等,本章我们需要用到,故设置这里为 1。
- 4) `FF_USE_MKFS`。用来定时是否使能格式化,本章需要用到,所以设置这里为 1。
- 5) `FF_USE_FASTSEEK`。这个用来使能快速定位,我们设置为 1,使能快速定位。
- 6) `FF_USE_LABEL`。这个用来设置是否支持磁盘盘符 (磁盘名字) 读取与设置。我们设置为 1,使能,就可以通过相关函数读取或者设置磁盘的名字了。
- 7) `FF_CODE_PAGE`。这个用于设置语言类型,包括很多选项 (见 FATFS 官网说明),我们这里设置为 936,即简体中文 (GBK 码,同一个文件夹下的 `ffunicode.c` 根据这个宏选择对应的语言设置)。
- 8) `FF_USE_LFN`。该选项用于设置是否支持长文件名 (还需要 `CODE_PAGE` 支持),取值范围为 0~3。0,表示不支持长文件名,1~3 是支持长文件名,但是存储地方不一样,我们选择使用 3,通过 `ff_memalloc` 函数来动态分配长文件名的存储区域。
- 9) `FF_VOLUMES`。用于设置 FATFS 支持的逻辑设备数目,我们设置为 2,即支持 2 个设备。
- 10) `FF_MAX_SS`。扇区缓冲的最大值,一般设置为 512。
- 11) `FF_FS_EXFAT`。新版本增加的功能,使用 exFAT 文件系统,用于支持超过 32GB 的超大存储。它们使用的是 exFAT 文件系统,使用它时必须要根据设置 `FF_USE_LFN` 这个参数的值以决定 exFATs 系统使用的内存来自堆栈还是静态数组。

其他配置项，我们这里就不一一介绍了，FATFS 的说明文档里面有很详细的介绍，大家自己阅读 <http://elm-chan.org/fsw/ff/doc/config.html> 即可。下面我们来讲讲 FATFS 的移植，FATFS 的移植主要分为 3 步：

- ① 数据类型：在 `integer.h` 里面去定义好数据的类型。这里需要了解你用的编译器的数据类型，并根据编译器定义好数据类型。
- ② 配置：通过 `ffconf.h` 配置 FATFS 的相关功能，以满足你的需要。
- ③ 函数编写：打开 `diskio.c`，进行底层驱动编写，一般需要编写 5 个接口函数，如图 50.1.2 所示：

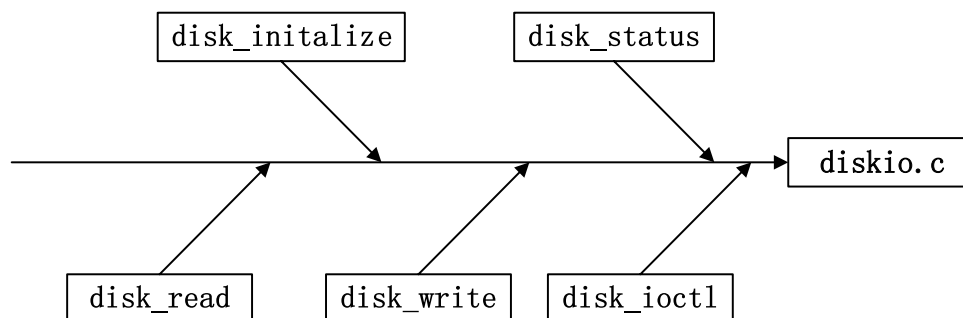


图 50.1.2 diskio 需要实现的函数

通过以下三步，我们即可完成对 FATFS 的移植。

第一步，我们使用的是 MDK5.34 编译器，器数据类型和 `integer.h` 里面定义的一致，所以此步，我们不需要做任何改动。

第二步，关于 `ffconf.h` 里面的相关配置，我们在前面已经有介绍（之前介绍的 11 个配置），我们将对应配置修改为我们介绍时候的值即可，其他的配置用默认配置。

第三步，因为 FATFS 模块完全与磁盘 I/O 层分开，因此需要下面的函数来实现底层物理磁盘的读写。底层磁盘 I/O 模块并不是 FATFS 的一部分，并且必须由用户提供。这些函数一般有 5 个，在 `diskio.c` 里面。

首先是 `disk_initialize` 函数，该函数介绍如表 50.1.2 所示：

| | |
|------|--|
| 函数名称 | <code>disk_initialize</code> |
| 函数原型 | <code>DSTATUS disk_initialize(BYTE Drive)</code> |
| 功能描述 | 初始化磁盘驱动器 |
| 函数参数 | Drive: 指定要初始化的逻辑驱动器号，即盘符，应当取值 0~9 |
| 返回值 | 函数返回一个磁盘状态作为结果，对于磁盘状态的细节信息，请参考 <code>disk_status</code> 函数 |
| 所在文件 | <code>ff.c</code> |
| 实例 | <code>disk_initialize(0);</code> /* 初始化驱动器 0 */ |
| 注意事项 | <code>disk_initialize</code> 函数初始化一个逻辑驱动器为读/写做准备，函数成功时，返回值的 <code>STA_NOINIT</code> 标志被清零； 应用程序不应调用此函数，否则卷上的 FAT 结构可能会损坏； 如果需要重新初始化文件系统，可使用 <code>f_mount</code> 函数； 在 FatFs 模块上卷注册处理时调用该函数可控制设备的改变； 此函数在 FatFs 挂在卷时调用，应用程序不应该在 FatFs 活动时使用此函数 |

表 50.1.2 disk_initialize 函数介绍

第二个函数是 `disk_status` 函数，该函数介绍如图 41.1.3 所示：

| | |
|------|---|
| 函数名称 | <code>disk_status</code> |
| 函数原型 | <code>DRESULT disk_status (BYTE Drive)</code> |
| 功能描述 | 返回当前磁盘驱动器的状态 |
| 函数参数 | Drive: 指定要确认的逻辑驱动器号，即盘符，应当取值 0~9 |
| 返回值 | 磁盘状态返回下列标志的组合，FatFs 只使用 <code>STA_NOINIT</code> 和 <code>STA_PROTECTED</code> <code>STA_NOINIT</code> : 表明磁盘驱动未初始化，下面列出产生该标志置位或清零的原因： |

| | |
|------|---|
| | 置位：系统复位，磁盘被移除和磁盘初始化函数失败 清零：磁盘初始化函数成功 STA_NODISK：表明驱动器中没有设备，安装磁盘驱动器后总为 0 STA_PROTECTED：表明设备被写保护，不支持写保护的总为 0，当 STA_NO-DISK 置位时非法 |
| 所在文件 | ff.c |
| 实例 | disk_status(0); /* 获取驱动器 0 的状态 */ |

表 50.1.3 disk_status 函数介绍

第三个函数是 disk_read 函数，该函数介绍如表 50.1.4 所示：

| | |
|------|---|
| 函数名称 | disk_read |
| 函数原型 | DRESULT disk_read (BYTE Drive, BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount) |
| 功能描述 | 从磁盘驱动器上读取扇区 |
| 函数参数 | Drive：指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向存储读取数据字节数组的指针，需要为所读取字节数的大小，扇区统计的扇区大小是需要的 （注：FaFts 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理） SectorNumber：指定起始扇区的逻辑块（LBA）上的地址 SectorCount：指定要读取的扇区数，取值 1~128 |
| 返回值 | RES_OK(0)：函数成功 RES_ERROR：读操作期间产生了任何错误且不能恢复它 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化 |
| 所在文件 | ff.c |

表 50.1.4 disk_read 函数介绍

第四个函数是 disk_write 函数，该函数介绍如表 50.1.5 所示：

| | |
|------|---|
| 函数名称 | disk_write |
| 函数原型 | DRESULT disk_write (BYTE Drive, const BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount) |
| 功能描述 | 向磁盘写入一个或多个扇区 |
| 函数参数 | Drive：指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向要写入字节数组的指针 （注：FaFts 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理） SectorNumber：指定起始扇区的逻辑块（LBA）上的地址 SectorCount：指定要写入的扇区数，取值 1~128 |
| 返回值 | RES_OK(0)：函数成功 RES_ERROR：写操作期间产生了任何错误且不能恢复它 RES_WRPERR：媒体被写保护 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化 |
| 所在文件 | ff.c |
| 注意事项 | 只读配置中不需要此函数 |

表 50.1.5 disk_write 函数介绍

第五个函数是 disk_ioctl 函数，该函数介绍如表 50.1.6 所示：

| | |
|------|---|
| 函数名称 | disk_ioctl |
| 函数原型 | DRESULT disk_ioctl (BYTE Drive, BYTE Command, void* Buffer) |
| 功能描述 | 控制设备指定特性和除了读/写外的杂项功能 |

| | |
|------|---|
| 函数参数 | Drive: 指定逻辑驱动器号, 即盘符, 应当取值 0~9 Command: 指定命令代码 Buffer: 指向参数缓冲区的指针, 取决于命令代码, 不使用时, 指定一个 NULL 指针 |
| 返回值 | RES_OK(0): 函数成功 RES_ERROR: 写操作期间产生了任何错误且不能恢复它 RES_PARERR: 非法参数 RES_NOTRDY: 磁盘驱动器没有初始化 |
| 所在文件 | ff.c |
| 注意事项 | CTRL_SYNC: 确保磁盘驱动器已经完成了写处理, 当磁盘 I/O 有一个写回缓存, 立即刷新原扇区, 只读配置下不适用此命令 GET_SECTOR_SIZE: 返回磁盘的扇区大小, 只用于 f_mkfs() GET_SECTOR_COUNT: 返回可利用的扇区数, _MAX_SS ≥ 1024 时可用 GET_BLOCK_SIZE: 获得擦除块大小, 只用于 f_mkfs() CTRL_ERASE_SECTOR: 强制擦除一块的扇区, _USE_ERASE > 0 时可用 |

表 50.1.6 disk_ioctl 函数介绍

以上五个函数, 我们将在软件设计部分一一实现。通过以上 3 个步骤, 我们就完成了对 FATFS 的移植, 就可以在我们的代码里面使用 FATFS 了。假如想要获得文件操作时的时间, 且 ffconf.h 文件中的 FF_FS_NORTC 宏为 0, 这时候还需要实现 get_fattime 函数。

FATFS 提供了很多 API 函数, 这些函数 FATFS 的自带介绍文件里面都有详细的介绍(包括参考代码), 我们这里就不多说了。这里需要注意的是, 在使用 FATFS 的时候, 必须先通过 f_mount 函数注册一个工作区, 才能开始后续 API 的使用, 关于 FATFS 的介绍, 我们就介绍到这里。大家可以通过 FATFS 自带的介绍文件进一步了解和熟悉 FATFS 的使用。

50.2 硬件设计

1. 例程功能

本章实验功能简介: 开机的时候先初始化 SD 卡, 初始化成功之后, 注册两个磁盘: 一个给 SD 卡用, 一个给 SPI FLASH 用, 之所以把 SPI FLASH 也当成磁盘来用, 一方面是为了演示大容量的 SPI Flash 也可以用 FATFS 管理, 说明 FATFS 的灵活性; 另一方面可以展示 FATFS 方式比原来直接按地址管理数据便利性, 使板载 SPI Flash 的使用更具灵活性。挂载成功后获取 SD 卡的容量和剩余空间, 并显示在 LCD 模块上, 最后定义 USMART 输入指令进行各项测试。本实验通过 LED0 指示程序运行状态。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) micro SD 卡
- 5) NOR FLASH

这几个外设原理图我们在之前的章节已经介绍过了, 这里就不重复介绍了, 不清楚的话可以查看本文之前章节的描述或对光盘资料提供的开发板原理图。

50.3 程序设计

FATFS 的驱动为一个硬件独立的组件, 因此我们把 FATFS 的移植代码放到“Middlewares”文件夹下。

本章, 我们在“SD 卡实验”的基础上进行拓展。在“Middlewares”下新建一个 FATFS 的文件夹, 然后将 FATFS R0.14b 程序包解压到该文件夹下。同时, 我们在 FATFS 文件夹里面新

建一个 exfuns 的文件夹，用于存放我们针对 FATFS 做的一些扩展代码（这个我们后面再讲）。操作结果如图 50.3.1 所示：

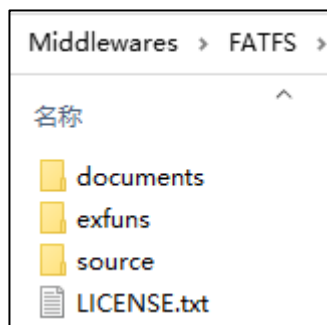


图 50.3.1 FATFS 文件夹子目录

50.3.1 程序流程图

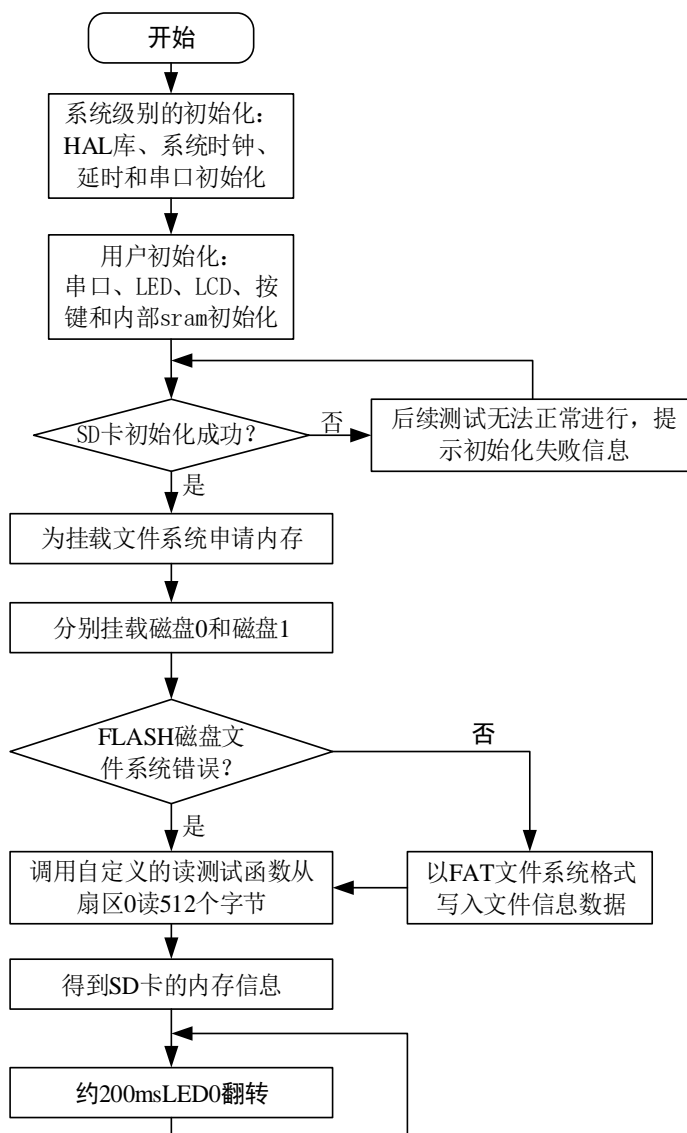


图 50.3.2.1 FATFS 实验程序流程图

50.3.2 程序解析

1. FATFS 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。

diskio.c/h 为我们提供了规定好的底层驱动接口的返回值。这个函数需要使用到我们的硬件接口，所以需要把使用到的硬件驱动的头文件包进来。

```
#include "../MALLOC/malloc.h"
#include "../FATFS/source/diskio.h"
#include "../BSP/SDIO/sdio_sdcard.h"
#include "../BSP/NORFLASH/norflash.h"
```

按照 50.1 的描述，接下来我们来对这几个接口进行补充实现。本章，我们用 FATFS 管理了 2 个磁盘：SD 卡和 SPI FLASH，我们设置 SD_CARD 为 0，EX_FLASH 位为 1，对应到 disk_read/disk_write 函数里面。SD 卡比较好说，但是 SPI FLASH，因为其扇区是 4K 字节大小，我们为了方便设计，强制将其扇区定义为 512 字节，这样带来的好处就是设计使用相对简单，坏处就是擦除次数大增，所以不要随便往 SPI FLASH 里面写数据，非必要最好别写，如果频繁写的话，很容易将 SPI FLASH 写坏。

```
#define SD_CARD 0 /* SD卡,卷标为 0 */
#define EX_FLASH 1 /* 外部 spi flash,卷标为 1 */
/**
 * 对于 25Q128 FLASH 芯片，我们规定前 12M 给 FATFS 使用，12M 以后
 * 紧跟字库，3 个字库 + UNIGBK.BIN，总大小 3.09M，共占用 15.09M
 * 15.09M 以后的存储空间大家可以随便使用。
 */
#define SPI_FLASH_SECTOR_SIZE 512 /* 扇区大小 */
#define SPI_FLASH_SECTOR_COUNT 12 * 1024 * 2 /* 扇区数目 */
#define SPI_FLASH_BLOCK_SIZE 8 /* 每个 BLOCK 有 8 个扇区 */
#define SPI_FLASH_FATFS_BASE 0 /* FATFS 在外部 FLASH 起始地址为 0 */
```

另外，diskio.c 里面的函数，直接决定了磁盘编号（盘符/卷标）所对应的具体设备，比如，以上代码中，我们就通过 switch 来判断，到底要操作 SD 卡，还是 SPI FLASH，然后，分别执行对应设备的相关操作。以此实现磁盘编号和磁盘的关联。

(1) disk_initialize 函数

要使用 FATFS 管理，首先要对磁盘进行初始化，所以先看磁盘的初始化函数，其声明如下：

```
DSTATUS disk_initialize ( BYTE pdrv)
```

● 函数描述：

初始化指定编号的磁盘，磁盘所指定的存储区。使用每个磁盘前进行初始化，那在代码中直接根据编号调用硬件的初始化接口即可，这样也能保证代码的扩展性，硬件的顺序可以根据自己的喜好定义。

● 函数形参：

形参 1 是 FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。

代码实现如下：

```
/**
 * @brief 初始化磁盘
 * @param pdrv : 磁盘编号 0~9
 * @retval DSTATUS:FATFS 规定的返回值
 */
DSTATUS disk_initialize (
    BYTE pdrv /* Physical drive nmuber to identify the drive */
)
{
    uint8_t res = 0;

    switch (pdrv)
    {
        case SD_CARD: /* SD卡 */
            res = sd_init(); /* SD卡初始化 */
    }
}
```



```

        break;

    case EX_FLASH:          /* 外部 flash */
        norflash_init();    /* 外部 flash 初始化 */
        break;

    default:
        res = 1;
}

if (res)
{
    return STA_NOINIT;
}
else
{
    return 0; /* 初始化成功 */
}
}

```

● 函数返回值:

DSTATUS 枚举类型的值, FATFS 规定了自己的返回值来管理各接口函数的操作结果, 方便后续函数的操作和判断, 我们看看它的定义:

```

/* Status of Disk Functions */
typedef BYTE    DSTATUS;
/* Disk Status Bits (DSTATUS) */
#define STA_NOINIT      0x01  /* Drive not initialized */
#define STA_NODISK      0x02  /* No medium in the drive */
#define STA_PROTECT     0x04  /* Write protected */

```

定义时也写出了各个参数的含义, 根据 ff.c 中的调用实例可知操作返回 0 才是正常的状态, 其它情况发生的话就需要结合硬件进行了。

(2) disk_status 函数

便于我们知道当前磁盘驱动器的状态, 所以 FATFS 也有 disk_status 函数提供给我们使用, 其声明如下:

```
DSTATUS disk_status (BYTE pdrv)
```

● 函数描述:

返回当前磁盘驱动器的状态。

● 函数形参:

FATFS 管理的磁盘编号 pdrv: 磁盘编号 0~9。我们配置 FF_VOLUMES 为 2 来支持两个磁盘, 因此可选值为 0 和 1。

● 函数返回值:

直接返回 RES_OK。

(3) disk_read 函数

disk_read 实现直接从硬件接口读取数据, 这个函数接口是给 FATFS 的其它读操作接口函数调用, 其声明如下:

```
DRESULT disk_read (BYTE pdrv, BYTE *buff, DWORD sector, UINT count)
```

● 函数描述:

从磁盘驱动器上读取扇区数据。

● 函数形参:

形参 1: 是 FATFS 管理的磁盘编号 pdrv: 磁盘编号 0~9, 我们配置 FF_VOLUMES 为 2 来支持两个磁盘, 因此可选值为 0 和 1。

形参 2: buff 指向要保存数据的内存区域指针, 为字节类型。

形参 3: sector 为实际物理操作时要访问的扇区地址。

形参 4: count 为本次要读取的数据量, 最长为 unsigned int, 读到的数量为字节数。

代码实现如下, 同样要根据我们定义的设备标号, 在 swich-case 中添加对应硬件的驱动:

```

DRESULT disk_read (
    BYTE pdrv,          /* Physical drive number to identify the drive */
    BYTE *buff,         /* Data buffer to store read data */

```

```

DWORD sector,      /* Sector address in LBA */
UINT count         /* Number of sectors to read */
{
    uint8_t res = 0;
    if (!count) return RES_PARERR; /* count 不能等于 0，否则返回参数错误 */

    switch (pdrv)
    {
        case SD_CARD: /* SD 卡 */
            res = sd_read_disk(buff, sector, count);

            while (res) /* 读出错 */
            {
                if (res != 2) sd_init(); /* 重新初始化 SD 卡 */

                res = sd_read_disk(buff, sector, count);
                //printf("sd rd error:%d\r\n", res);
            }
            break;

        case EX_FLASH: /* 外部 flash */
            for (; count > 0; count--)
            {
                norflash_read(buff, SPI_FLASH_FATFS_BASE + sector *
                               SPI_FLASH_SECTOR_SIZE, SPI_FLASH_SECTOR_SIZE);

                sector++;
                buff += SPI_FLASH_SECTOR_SIZE;
            }
            res = 0;
            break;

        default:
            res = 1;
    }
    /* 处理返回值，将返回值转成 ff.c 的返回值 */
    if (res == 0x00)
    {
        return RES_OK;
    }
    else
    {
        return RES_ERROR;
    }
}

```

● 函数返回值：

DRESULT 为枚举类型，diskio.h 中有其定义，根据返回值的含义确认操作结果即可。在这里也把该结果展示出来，如下所示：

```

/* Results of Disk Functions */
typedef enum
{
    RES_OK = 0,      /* 0: 操作成功 */
    RES_ERROR,       /* 1: 读/写错误 */
    RES_WRPRT,       /* 2: 写保护状态 */
    RES_NOTRDY,      /* 3: 设备忙 */
    RES_PARERR       /* 4: 其它情形 */
} DRESULT;

```

根据返回值的含义确认操作结果即可。

(4) disk_write 函数

disk_write 实现直接从硬件接口写入数据，这个函数接口给 FATFS 的其它写操作接口函数调用，其声明如下：

```

DRESULT disk_write ( BYTE pdrv, const BYTE *buff, DWORD sector, UINT count)

```

● 函数描述：

向磁盘驱动器上写入扇区数据。

● 函数形参：

形参 1：是 FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。

形参 2：buff 指向要发送数据的内存区域指针，为字节类型。

形参 3：sector 为实际物理操作时要访问的扇区地址。

形参 4：count 为本次要写入的数据量。

代码实现如下，同样要根据我们定义的设备标号，在 switch-case 中添加对应硬件的驱动：

```
DRESULT disk_write (
    BYTE pdrv,          /* Physical drive nmuber to identify the drive */
    const BYTE *buff,    /* Data to be written */
    DWORD sector,        /* Sector address in LBA */
    UINT count           /* Number of sectors to write */
)
{
    uint8_t res = 0;

    if (!count) return RES_PARERR; /* count 不能等于 0，否则返回参数错误 */

    switch (pdrv)
    {
        case SD_CARD:    /* SD 卡 */
            res = sd_write_disk((uint8_t *)buff, sector, count);
            while (res)    /* 写出错 */
            {
                sd_init(); /* 重新初始化 SD 卡 */
                res = sd_write_disk((uint8_t *)buff, sector, count);
                //printf("sd wr error:%d\r\n", res);
            }
            break;

        case EX_FLASH:   /* 外部 flash */
            for (; count > 0; count--)
            {
                norflash_write((uint8_t *)buff,
                               SPI_FLASH_FATFS_BASE + sector* SPI_FLASH_SECTOR_SIZE,
                               SPI_FLASH_SECTOR_SIZE);

                sector++;
                buff += SPI_FLASH_SECTOR_SIZE;
            }
            res = 0;
            break;

        default:
            res = 1;
    }

    /* 处理返回值，将返回值转成 ff.c 的返回值 */
    if (res == 0x00)
    {
        return RES_OK;
    }
    else
    {
        return RES_ERROR;
    }
}
```

● 函数返回值：

DRESULT 为枚举类型,diskio.h 中有其定义，编写读函数时已经介绍了，注意要把返回值转成这个枚举类型的参数。

(5) disk_ioctl 函数

disk_ioctl 实现一些控制命令，这个接口为 FATFS 提供一些硬件操作信息，其声明如下：

```
DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void *buff)
```

- **函数描述:**
控制设备指定特性和除了读/写外得杂项功能。
- **函数形参:**
形参 1: FATFS 管理的磁盘编号 pdrv:磁盘编号 0~9, 我们配置 FF_VOLUMES 为 2 来支持两个磁盘, 因此可选值为 0 和 1。
形参 2: cmd 是 FATFS 定义好的一些宏, 用于访问硬盘设备的一些状态。我们实现几个简单的操作接口, 用于获取磁盘容量这些基础信息 (diskio.h 中已经定义好了), 为方便, 我们先只实现几个标准的应用接口, 关于 SDIO 的一些扩展命令我们再根据需要进行支持。

```
/* Command code for disk_ioctl fucntion */
/* Generic command (Used by FatFs) */
#define CTRL_SYNC          0      /* 完成挂起的写入过程 (当 FF_FS_READONLY == 0) */
#define GET_SECTOR_COUNT  1      /* 获取磁盘扇区数 (当 FF_USE_MKFS == 1) */
#define GET_SECTOR_SIZE   2      /* 获取磁盘存储空间大小 (当 FF_MAX_SS != FF_MIN_SS) */
#define GET_BLOCK_SIZE    3      /* 每个扇区块的大小 (当 FF_USE_MKFS == 1) */
```

下面是从 <http://elm-chan.org/fsw/ff/doc/dioctl.html> 得到的参数实现效果, 我们也可以参考原有的 disk_ioctl 的实现来理解这几个参数。

| 命令 | 说明 |
|------------------|---|
| CTRL_SYNC | 确保设备已完成挂起的写入过程。如果磁盘 I/O 层或存储设备具有回写缓存, 则必须立即将缓存数据提交到介质。如果对介质的每个写入操作都正常完成, 则此命令无任何动作。 |
| GET_SECTOR_COUNT | 返回对应标号的硬盘的可用扇区数。此命令由 f_mkfs/f_disk 函数用于确定要创建的卷/分区的大小。使用时需要设置 FF_USE_MKFS 为 1。 |
| GET_SECTOR_SIZE | 将用于读/写函数的扇区大小检索到 buff 指向的 WORD 变量中。有效扇区大小为 512、1024、2048 和 4096。只有在 FF_MAX_SS>FF_MIN_SS 执行此命令时。当 FF_MAX_SS=FF_MIN_SS 时, 将永远不会使用此命令, 并且读/写函数必须仅在 FF_MAX_SSbytes/扇区中工作。 |
| GET_BLOCK_SIZE | 将扇区单位中闪存介质的块大小以 DWORD 指针存到 buff 中。允许的值为 1 到 32768。如果擦除块大小未知或非闪存介质, 则返回 1。此命令仅由 f_mkfs 函数使用, 并尝试对齐擦除块边界上的数据区域。使用时需要设置 FF_USE_MKFS 为 1。 |

形参 3: buff 为 void 形指针, 根据命令的格式和需要, 把对应的值转成对应的形式传给它。参考原有的 disk_ioctl 的实现, 我们的函数实现如下。

```
DRESULT disk_ioctl (
    BYTE pdrv,      /* Physical drive nmuber (0..) */
    BYTE cmd,       /* Control code */
    void *buff      /* Buffer to send/receive control data */
)
{
    DRESULT res;

    if (pdrv == SD_CARD)    /* SD 卡 */
    {
        switch (cmd)
        {
            case CTRL_SYNC:
                res = RES_OK;
                break;
            case GET_SECTOR_SIZE:
                *(DWORD *)buff = 512;
                res = RES_OK;
                break;
            case GET_BLOCK_SIZE:
                *(WORD *)buff = g_sdcard_handler.SdCard.BlockSize;
```

```

        res = RES_OK;
        break;
    case GET_SECTOR_COUNT:
        *(DWORD *)buff = ((long long) g_sdcard_handler.SdCard.Block
                           Nbr*g_sdcard_handler.SdCard.BlockSize)/512;

        res = RES_OK;
        break;
    default:
        res = RES_PARERR;
        break;
    }
}
else if (pdrv == EX_FLASH) /* 外部 FLASH */
{
    switch (cmd)
    {
        case CTRL_SYNC:
            res = RES_OK;
            break;
        case GET_SECTOR_SIZE:
            *(WORD *)buff = SPI_FLASH_SECTOR_SIZE;
            res = RES_OK;
            break;
        case GET_BLOCK_SIZE:
            *(WORD *)buff = SPI_FLASH_BLOCK_SIZE;
            res = RES_OK;
            break;
        case GET_SECTOR_COUNT:
            *(DWORD *)buff = SPI_FLASH_SECTOR_COUNT;
            res = RES_OK;
            break;
        default:
            res = RES_PARERR;
            break;
    }
}
else
{
    res = RES_ERROR; /* 其他的不支持 */
}
return res;
}

```

● 函数返回值:

DRESULT 为枚举类型，diskio.h 中有其定义，编写读函数时已经介绍了，注意要把返回值转成这个枚举类型的参数。

以上实现了 50.1 节提到的 5 个函数，ff.c 中需要实现 get_fatime(void)，同时因为在 ffconf.h 里面设置对长文件名的支持为方法 3，所以必须在 ffsystem.c 中实现 get_fatime、ff_memalloc 和 ff_memfree 这三个函数。这部分比较简单，直接参考我们修改后的 ffsystem.c 的源码。

至此，我们已经可以直接使用 FATFS 下的 ff.c 下的 f_mount 的接口挂载磁盘，然后使用类似标准 C 的文件操作函数，就可以实现文件操作。但 f_mount 还需要一些文件操作的内存，为了方便操作，我们在 FATFS 文件夹下还新建了一个 exfuncs 的文件夹，该文件夹用于保存一些 FATFS 一些针对 FATFS 的扩展代码，如刚才提到的 FATFS 相关函数的内存申请方法等。

本章，我们编写了 4 个文件，分别是：exfuncs.c、exfuncs.h、fattester.c 和 fattester.h。其中 exfuncs.c 主要定义了一些全局变量，方便 FATFS 的使用，同时实现了磁盘容量获取等函数。而 fattester.c 文件则主要是为了测试 FATFS 用，因为 FATFS 的很多函数无法直接通过 USMART 调用，所以我们在 fattester.c 里面对这些函数进行了一次再封装，使得可以通过 USMART 调用。

这几个文件的代码，大家可以直接使用本例程源码，我们将 exfuncs.c/h 和 fattester.c/h 加入 FATFS 组下的 exfuncs 文件下，直接使用即可。

(6) exfuncs_init 函数

我们在使用文件操作前，需要用 `f_mount` 函数挂载磁盘，我们在挂载 SD 卡前需要一些文件系统的内存，为了方便管理，我们定义一个全局的 `fs[FF_VOLUMES]` 指针，定成数组是我们管理多个磁盘，而 `f_mount` 也需要一个 FATFS 类型的指针，定义如下：

```
/* 逻辑磁盘工作区(在调用任何 FATFS 相关函数之前,必须先给 fs 申请内存) */
FATFS *fs[FF_VOLUMES];
```

接下来只要用内存管理部分的知识来实现对 `fs` 指针的内存申请即可。

```
/**
 * @brief      为 exfuncs 申请内存
 * @param      无
 * @retval     0, 成功; 1, 失败.
 */
uint8_t exfuncs_init(void)
{
    uint8_t i;
    uint8_t res = 0;
    for (i = 0; i < FF_VOLUMES; i++)
    {
        /* 为磁盘 i 工作区申请内存 */
        fs[i] = (FATFS *)mymalloc(SRAMIN, sizeof(FATFS));
        if (!fs[i])break;
    }

    #if USE_FATTESTER == 1      /* 如果使能了文件系统测试 */
        res = mf_init();      /* 初始化文件系统测试(申请内存) */
    #endif

    if (i == FF_VOLUMES && res == 0)
        return 0; /* 申请有一个失败,即失败. */
    else
        return 1;
}
```

其它的函数我们暂时没有使用到，这里先不介绍了，大家使用时查阅源码即可。

2. main.c 代码

最后，我们编写的主函数如下：

```
int main(void)
{
    uint32_t total, free;
    uint8_t t = 0;
    uint8_t res = 0;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32F103", RED);
    lcd_show_string(30, 70, 200, 16, 16, "FATFS TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "Use USMART for test", RED);

    while (sd_init()) /* 检测不到 SD 卡 */
    {
        lcd_show_string(30, 150, 200, 16, 16, "SD Card Error!", RED);
        delay_ms(500);
        lcd_show_string(30, 150, 200, 16, 16, "Please Check! ", RED);
    }
```



```

    delay_ms(500);
    LED0_TOGGLE(); /* LED0 闪烁 */
}

exfuns_init(); /* 为 fatfs 相关变量申请内存 */
f_mount(fs[0], "0:", 1); /* 挂载 SD 卡 */
res = f_mount(fs[1], "1:", 1); /* 挂载 FLASH */

if (res == 0X0D) /* FLASH 磁盘, FAT 文件系统错误, 重新格式化 FLASH */
{
    /* 格式化 FLASH */
    lcd_show_string(30, 150, 200, 16, 16, "Flash Disk Formatting...", RED);
    res = f_mkfs("1:", 0, 0, FF_MAX_SS); /* 格式化 FLASH */
    if (res == 0) /* 格式化完成 */
    {
        /* 设置 Flash 磁盘的名字为: ALIENTEK */
        f_setlabel((const TCHAR *) "1:ALIENTEK");
        lcd_show_string(30, 150, 200, 16, 16, "Flash Disk Format Finish", RED);
    }
    else /* 格式化失败 */
        lcd_show_string(30, 150, 200, 16, 16, "Flash Disk Format Error ", RED);

    delay_ms(1000);
}
lcd_fill(30, 150, 240, 150 + 16, WHITE); /* 清除显示 */

while (exfuns_get_free("0", &total, &free)) /* 得到 SD 卡的总容量和剩余容量 */
{
    lcd_show_string(30, 150, 200, 16, 16, "SD Card Fatfs Error!", RED);
    delay_ms(200);
    lcd_fill(30, 150, 240, 150 + 16, WHITE); /* 清除显示 */
    delay_ms(200);
    LED0_TOGGLE(); /* LED0 闪烁 */
}

lcd_show_string(30, 150, 200, 16, 16, "FATFS OK!", BLUE);
lcd_show_string(30, 170, 200, 16, 16, "SD Total Size:    MB", BLUE);
lcd_show_string(30, 190, 200, 16, 16, "SD Free Size:    MB", BLUE);
lcd_show_num(30 + 8 * 14, 170, total >> 10, 5, 16, BLUE); /* 显示卡总容量 MB */
lcd_show_num(30 + 8 * 14, 190, free >> 10, 5, 16, BLUE); /* 显示卡剩余容量 */

while (1)
{
    t++;
    delay_ms(200);
    LED0_TOGGLE(); /* LED0 闪烁 */
}
}

```

在 main 函数里面, 我们为 SD 卡和 FLASH 都注册了工作区 (挂载), 在初始化 SD 卡并显示其容量信息后, 进入死循环, 等待 USMART 测试。

在 usmart_config.c 里面的 usmart_nametab 数组添加如下内容:

```

struct _m_usmart_nametab usmart_nametab[] =
{
    #if USMART_USE_WRFUNS == 1 /* 如果使能了读写操作 */
    (void *)read_addr, "uint32_t read_addr(uint32_t addr)",
    (void *)write_addr, "void write_addr(uint32_t addr, uint32_t val)",
    #endif
    (void *)delay_ms, "void delay_ms(uint16_t nms)",
    (void *)delay_us, "void delay_us(uint32_t nus)",
    (void *)mf_mount, "uint8_t mf_mount(uint8_t* path, uint8_t mt)",
    (void *)mf_open, "uint8_t mf_open(uint8_t* path, uint8_t mode)",
    (void *)mf_close, "uint8_t mf_close(void)",
    (void *)mf_read, "uint8_t mf_read(uint16_t len)",
    (void *)mf_write, "uint8_t mf_write(uint8_t* dat, uint16_t len)",
}

```

```
(void *)mf_opendir, "uint8_t mf_opendir(uint8_t* path)",
(void *)mf_closedir, "uint8_t mf_closedir(void)",
(void *)mf_readdir, "uint8_t mf_readdir(void)",
(void *)mf_scan_files, "uint8_t mf_scan_files(uint8_t * path)",
(void *)mf_showfree, "uint32_t mf_showfree(uint8_t *path)",
(void *)mf_lseek, "uint8_t mf_lseek(uint32_t offset)",
(void *)mf_tell, "uint32_t mf_tell(void)",
(void *)mf_size, "uint32_t mf_size(void)",
(void *)mf_mkdir, "uint8_t mf_mkdir(uint8_t*path)",
(void *)mf_fmks, "uint8_t mf_fmks(uint8_t* path,uint8_t opt,uint16_t au)",
(void *)mf_unlink, "uint8_t mf_unlink(uint8_t *path)",
(void *)mf_rename, "uint8_t mf_rename(uint8_t *oldname,uint8_t * newname)",
(void *)mf_getlabel, "void mf_getlabel(uint8_t *path)",
(void *)mf_setlabel, "void mf_setlabel(uint8_t *path)",
(void *)mf_gets, "void mf_gets(uint16_t size)",
(void *)mf_putc, "uint8_t mf_putc(uint8_t c)",
(void *)mf_puts, "uint8_t mf_puts(uint8_t *str)",
};
```

50.4 下载验证

在代码编译成功之后，我们通过下载代码到正点原子战舰 STM32F103 上，我们测试使用的是 16GB 标有“SDHC”标志的 Micro SD 卡，可以看到 LCD 显示如图 50.4.1 所示：

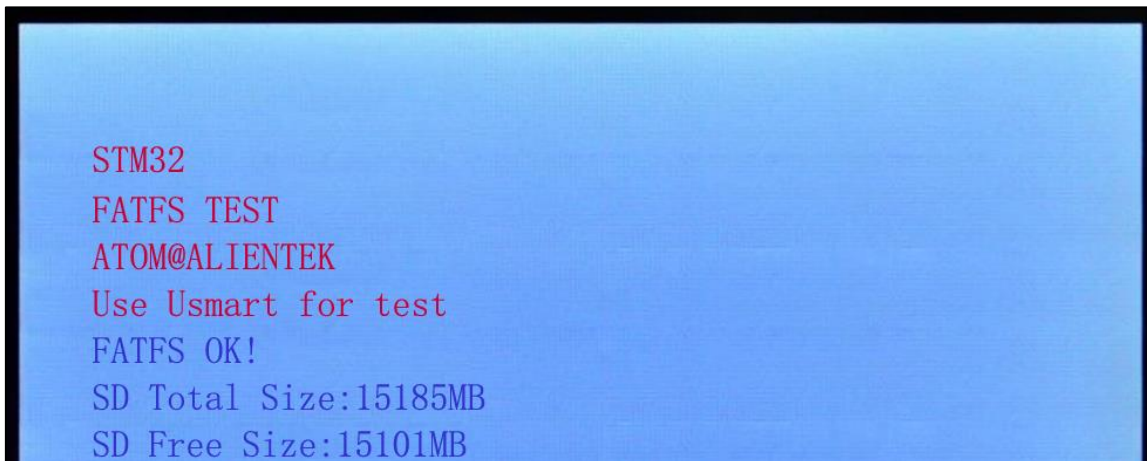


图 50.4.1 程序运行效果图

打开串口调试助手，我们就可以串口调用前面添加的各种 FATFS 测试函数了，比如我们输入 mf_scan_files("0:");，即可扫描 SD 卡根目录的所有文件，如图 50.4.2 所示：



图 50.4.2 扫描 SD 卡根目录所有文件

其他函数的测试，用类似的办法即可实现。注意这里 0 代表 SD 卡，1 代表 SPI FLASH。另外，提醒大家，mf_unlink 函数，在删除文件夹的时候，必须保证文件夹是空的，才可以正常删除，否则不能删除。

第五十一章 汉字显示实验

本章，我们将介绍如何使用 STM32 控制 LCD 显示汉字。在本章中，我们将使用外部 SPI FLASH 来存储字库，并可以通过 SD 卡更新字库。STM32 读取存在 SPI FLASH 里面的字库，然后将汉字显示在 LCD 上面。

本章分为如下几个小节：

51.1 汉字显示介绍

51.2 硬件设计

51.3 程序设计

51.4 下载验证

51.1 汉字显示原理简介

汉字的显示和 ASCII 显示其实是一样的原理，如图 51.1.1 所示：

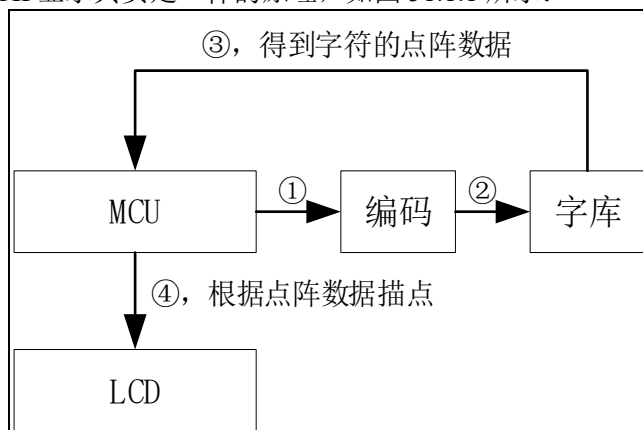


图 51.1.1 单个汉字显示原理框图

上图显示了单个汉字显示的原理框图，单片机（MCU）先根据汉字编码（①，②）从字库里面找到该汉字的点阵数据（③），然后通过描点函数，按字库取模方式，将点阵数据在 LCD 上画出来（④），就可以实现一个汉字的显示。

接下来，重点介绍一下汉字的：编码、字库及显示等相关知识。

51.1.1 字符编码介绍

单片机只能识别 0 和 1，所有信息都是以 0 和 1 的形式存储的，单片机本身并不能识别字符，所以我们需要对字符进行编码（也叫内码，特定的编码对应特定的字符），单片机通过编码来识别具体的汉字。常见的字符集编码如表 51.1.1.1 所示：

| 字符集 | 编码长度 | 说明 |
|---------|----------|--------------------------------|
| ASCII | 1 个字节 | 拉丁字母编码，仅 128 个编码，最简单 |
| GB2312 | 2 个字节 | 简体中文字符编码，包含约 6000 多汉字编码 |
| GBK | 2 个字节 | 对 GB2312 的扩充，支持繁体中文，约 2W 多汉字编码 |
| BIG5 | 2 个字节 | 繁体中文字符编码，在台湾、香港用的多 |
| UNICODE | 一般 2 个字节 | 国际标准编码，支持各国文字 |

表 51.1.1.1 常见字符集编码

其中 ASCII 编码最简单，采用单字节编码，在前面的 OLED 和 LCD 实验，我们已经有所接触。ASCII 是基于拉丁字母的一套电脑编码系统，仅包括 128 个编码，其中 95 个显示字符，使用一个字节即可编码完所有字符，我们常见的英文字母和数字，就是使用 ASCII 字符编码，另外 ASCII 字符显示所占宽度为汉字宽度的一半！也可以理解成：ASCII 字符的宽度 = 高度的一半。

GB2312、GBK 和 BIG5 都是汉字编码，GBK 码是 GB2312 的扩充，是国内计算机系统默认的汉字编码，而 BIG5 则是繁体汉字字符集编码，在香港和台湾的计算机系统汉字编码一般默认使用 BIG5 编码。一般来说，汉字显示所占的宽度等于高度，即宽度和高度相等。

UNICODE 是国际标准编码，支持各国文字，一般是 2 字节编码（也可以是 3 字节），这里不做讨论。想详细了解的可以执行百度学习。

接下来，我们重点介绍一下 GBK 编码。

GBK 是一套汉字编码规则，采用双字节编码，共 23940 个码位，收录汉字和图形符号 21886 个，其中汉字（含繁体字和构件）21003 个，图形符号 883 个。

每个 GBK 码由 2 个字节组成，第一个字节范围：0X81~0XFE，第二个字节分为两部分，一是：0X40~0X7E，二是：0X80~0XFE。其中与 GB2312 相同的区域，字完全相同。GBK 编码规则如表 51.1.1.2 所示：

| 字节 | 范围 | 说明 |
|-------------|-----------|---------------------------------|
| 第一字节 (高) | 0X81~0XFE | 共 126 个区（不包括 0X00~0X80，以及 0XFF） |
| 第二字节 (低) | 0X40~0X7E | 63 个编码（不包括 0X00~0X39，以及 0X7F） |
| | 0X80~0XFE | 127 个编码（不包括 0XFF） |

表 51.1.1.2 GBK 编码规则

我们把第一个字节（高字节）代表的意义称为区，那么 GBK 里面总共有 126 个区（0XFE - 0X81 + 1），每个区内有 190 个汉字（0XFE - 0X80 + 0X7E - 0X40 + 2），总共就有 126*190=23940 个汉字。

第一个编码：0X8140，对应汉字：丿；

第二个编码：0X8141，对应汉字：丨；

第三个编码：0X8142，对应汉字：丁；

第四个编码：0X8143，对应汉字：丄；

依次对所有汉字进行编码，详见：www.qqxiuzi.cn/zh/hanzi-gbk-bianma.php。

51.1.2 汉字字库简介

光有汉字编码，单片机还是无法在 LCD 上显示这个汉字的，必须有对应汉字编码的点阵数据，才可以通过描点的方式，将汉字显示在 LCD 上。所有汉字点阵数据的集合，就叫做汉字字库。而不同大小的汉字，其字库大小也不一样，因此又有不同大小汉字的字库（如：12*12 汉字字库、16*16 汉字字库、24*24 汉字字库等）。

单个汉字的点阵数据，也称之为字模。汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画。为了方便取模和描点，我们一般规定一个取模方向，当取模和描点都按取模方向来操作，就可以实现一个汉字的点阵数据提取和显示。

以 12*12 大小的“好”字为例，假设我们规定取模方向为：从上到下，从左到右，且高位在前，则其取模原理如图 51.1.2.1 所示：

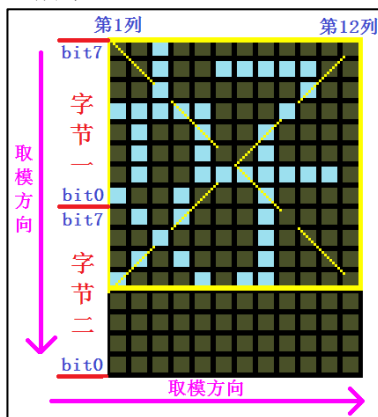


图 51.1.2.1 从上到下，从左到右取模原理

图中，我们取模的时候，从最左上方的点开始取（从上到下，从左到右），且高位在前（bit7 在表示第一个位），那么：

第一个字节是：0X11（1，表示浅蓝色的点，即要画出来的点，0 则表示不要画出来）；

第二个字节是：0X10；

第三个字节是：0X1E（到第二列了，每列 2 个字节）；

第四个字节是：0XA0；

以此类推，共 12 列，每列 2 个字节，总共 24 字节，12*12 “好” 字完整的字模如下：

```
uint8_t hzm_1212[24]={
0x11,0x10,0x1E,0xA0,0xF0,0x40,0x11,0xA0,0x1E,0x10,0x42,0x00,
0x42,0x10,0x4F,0xF0,0x52,0x00,0x62,0x00,0x02,0x00,0x00,0x00}; /* 好字字模 */
```

在显示的时候，我们只需要读取这个汉字的点阵数据（12*12 字体，一个汉字的点阵数据为 24 个字节），然后将这些数据，按取模方式，反向解析出来（坐标要处理好），每个字节，是 1 的位，就画出来，不是 1 的位，就忽略，这样，就可以显示出这个汉字了。

知道显示一个汉字的原理，就可以推及整个汉字库了，要显示任意汉字，我们首先要知道该汉字的点阵数据，整个 GBK 字库是比较大的（2W 多个汉字），这些数据可以由专门的软件来生成。

字库的制作

字库的制作，我们用到一款由正点原子团队开发的软件：ATK_XFONT。该软件可以在 WINDOWS 系统下生成任意点阵大小的 ASCII，GB2312(中文)、GBK(中文)、BIG5(繁体中文)和 Unicode 等共十几种编码的字库，不但支持生成二进制文件格式的文件，还支持生成图片功能，并支持横向，纵向等多种扫描方式。该软件的默认界面如图 51.1.2.2 所示：

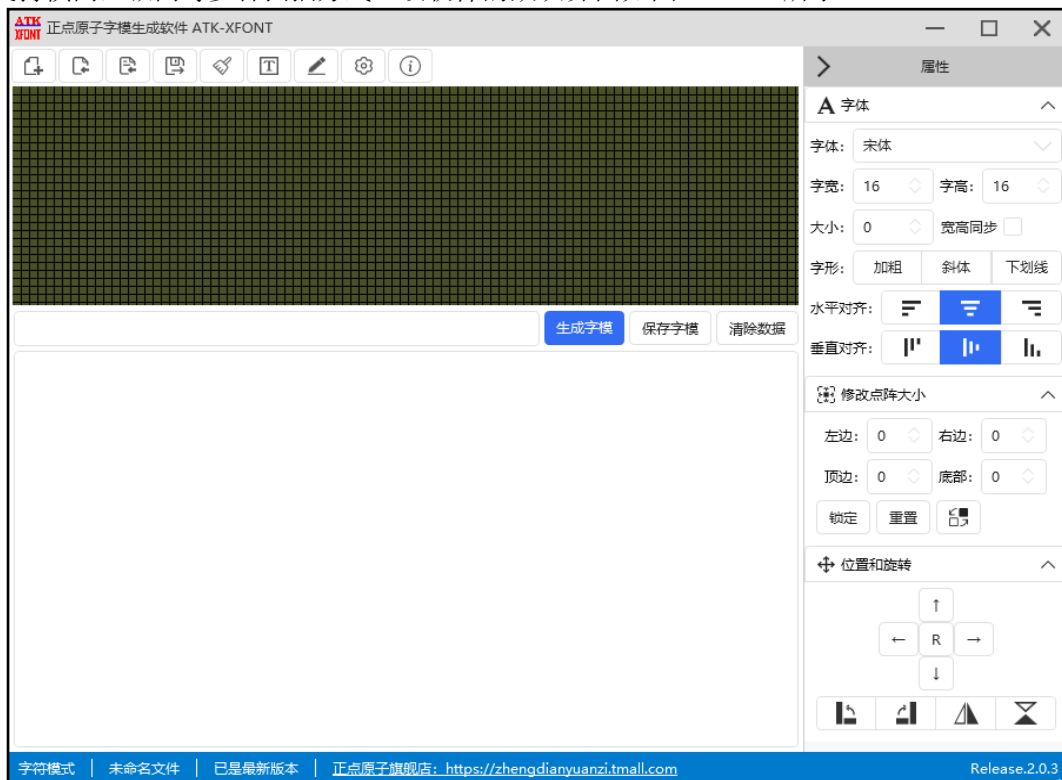


图 51.1.2.2 点阵字库生成器默认界面

要生成 16*16 的 GBK 字库，第一步进入 XFONT 软件的字库模式；第二步设置编码和字体大小，这里需要选择 GBK 编码以及设置字体大小为 16*16，另外还需要设置输出路径，这个由用户自己去设置，后面生成的字库文件会出现在该路径下；第三步设置取模方式，这里需要设置为从上到下，从左到右，高位在前；第四步，按下生成字库按钮，等待字库生成。这一系列的具体操作如图 51.1.2.3 所示：

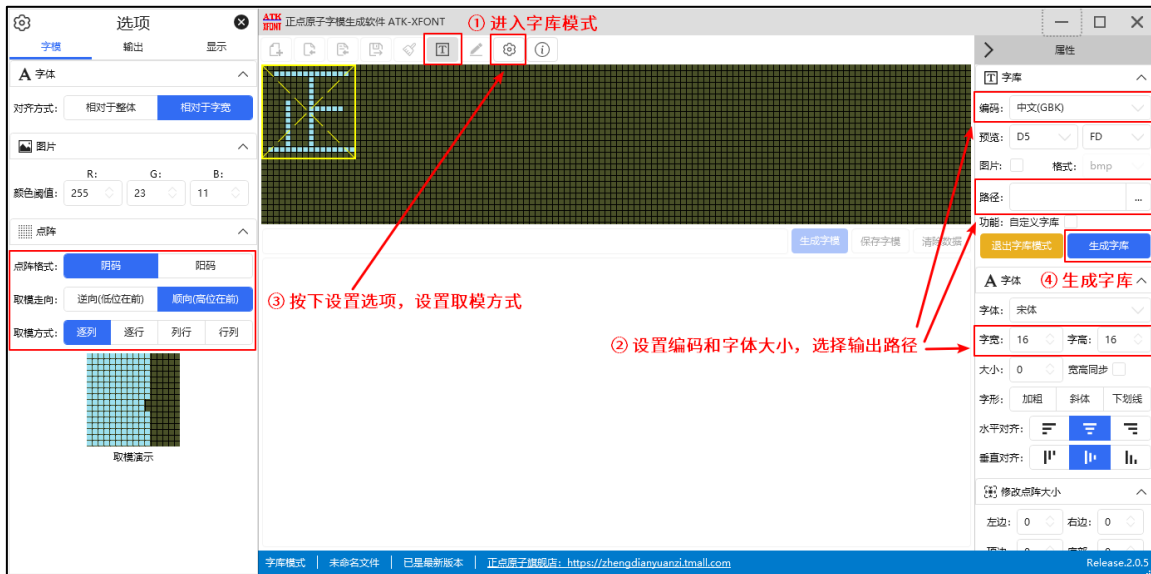


图 51.1.2.3 生成 GBK16*16 字库的设置方法

注意：电脑端的字体大小与我们生成点阵大小的关系为：

$$fsize = dsize * 6 / 8$$

其中，fsize 是电脑端字体的大小，dsize 是点阵大小（12、16、24 等）。所以 16*16 点阵大小对应的是 12 号字体。

生成完以后，我们把文件名和后缀改成：GBK16.FON（这里是手动修改后缀!!）。用类似的方法，生成 12*12 的点阵库（GBK12.FON）和 24*24 的点阵库（GBK24.FON），总共制作 3 个字库。

另外，该软件还可以生成其他很多字库，字体也可选，大家可以根据自己的需要按照上面的方法生成即可。该软件的详细介绍请看《ATK-XFONT 软件用户手册》。

最后，由于汉字字库比较大，我们不可能将其烧录在 MCU 内部 FLASH 里面。因此，我们生成的字库，要先放入 TF 卡，然后通过 TF 卡将字库文件复制到单片机外挂的 SPI FLASH 芯片（25Qxx）里面。使用的时候，单片机从 SPI FLASH 里面获取汉字点阵数据，这样，SPI FLASH 就相当于一个汉字字库芯片了。

51.1.3 汉字显示原理

经过以上两个小节的学习，我们可以归纳出汉字显示的过程：

MCU→汉字编码→汉字字库→汉字点阵数据→描点

编码和字库的制作我们已经学会了，所以只剩下一个问题：如何通过汉字编码在汉字字库里面查找对应汉字的点阵数据？

根据 GBK 编码规则，我们的汉字点阵字库只要按照这个编码规则从 0X8140 开始，逐一建立，每个区的点阵大小为每个汉字所用的字节数*190。这样，我们就可以得到在这个字库里面定位汉字的方法：

当 GBKL < 0X7F 时：Hp = ((GBKH - 0x81) * 190 + GBKL - 0X40) * csize;

当 GBKL > 0X80 时：Hp = ((GBKH - 0x81) * 190 + GBKL - 0X41) * csize;

其中 GBKH、GBKL 分别代表 GBK 的第一个字节和第二个字节(也就是高字节和低字节)，csize 代表单个汉字点阵数据的大小（字节数），Hp 则为对应汉字点阵数据在字库里面的起始地址(假设是从 0 开始存放，如果是非 0 开始，则加上对应偏移量即可)。

单个汉字点阵数据大小（csize）计算公式如下：

$$csize = (size / 8 + ((size \% 8) ? 1 : 0)) * (size);$$

其中 size 为汉字点阵长宽尺寸，如：12（对应 12*12 字体）、16（对应 16*16 字体）、24（对应 24*24 字体）。对于 12*12 字体，csize 大小为 24 字节，对于 16*16 字体，csize 大小为 32 字节。

通过以上方法，从字库里面获取到某个汉字点阵数据后，按取模方式（使用：从上到下、

从左到右，高位在前）进行描点还原即可将汉字显示在 LCD 上面。这就是汉字显示的原理。

51.1.4 ffunicode.c 优化（补充说明）

本小节内容和汉字显示无关，仅做补充说明，可选择性学习。

在上一章，我们提到要用 ffunicode.c，以支持长文件名，但是 ffunicode.c 文件里面中文转换（中文的页面编码代号为：936）的两个数组太大了（172KB），直接刷在单片机里面，太占用 flash 了，所以我们必须把这两个数组存放在外部 flash。数组 uni2oem936 和 oem2uni936 存放 unicode 和 gbk 的互相转换对照表，这两个数组很大，这里我们利用正点原子提供的一个 C 语言数组转 BIN（二进制）的软件：C2B 转换助手 V2.0.exe，将这两个数组转为 BIN 文件，我们将这两个数组拷贝出来存放为一个新的文本文件，假设为 UNIGBK.TXT，然后用 C2B 转换助手打开这个文本文件，如图 51.1.4.1 所示：

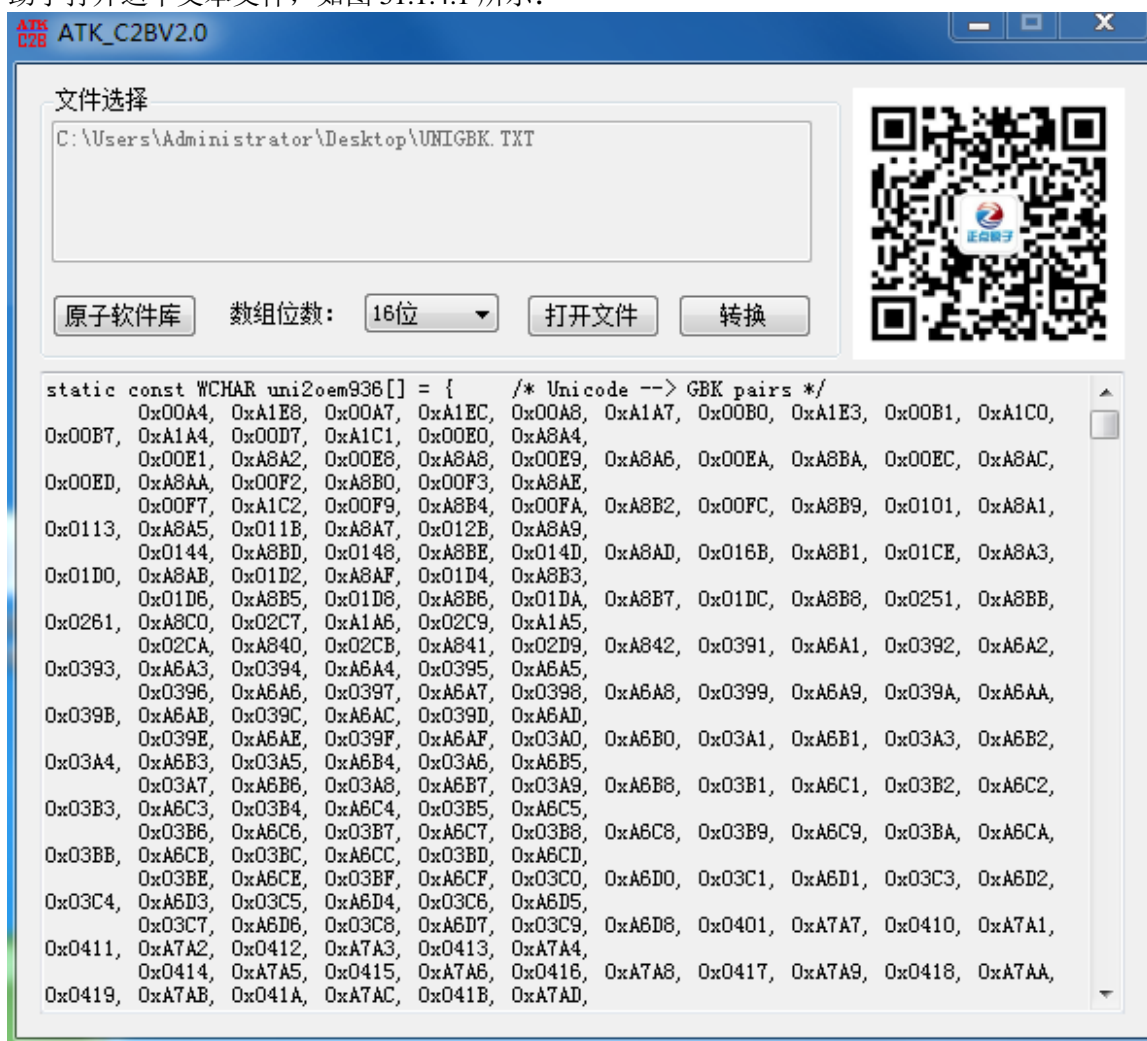


图 51.1.4.1 C2B 转换助手

然后点击转换，就可以在当前目录下（文本文件所在目录下）得到一个 UNIGBK.bin 的文件。这样就完成将 C 语言数组转换为.bin 文件，然后只需要将 UNIGBK.bin 保存到外部 FLASH 就实现了该数组的转移。

在 ffunicode.c 里面，通过 ff_uni2oem 和 ff_oem2uni 调用这两个数组，实现 UNICODE 和 GBK 的互转，该函数原代码如下：

```
WCHAR ff_uni2oem ( /* Returns OEM code character, zero on error */
    DWORD uni, /* UTF-16 encoded character to be converted */
    WORD cp /* Code page for the conversion */
)
{
```

```

const WCHAR *p;
WCHAR c = 0, uc;
UINT i = 0, n, li, hi;

if (uni < 0x80) /* ASCII? */
{
    c = (WCHAR)uni;
}
else /* Non-ASCII */
{
    if (uni < 0x10000 && cp == FF_CODE_PAGE) /* in BMP and valid code page? */
    {
        uc = (WCHAR)uni;
        p = CVTBL(uni2oem, FF_CODE_PAGE);
        hi = sizeof CVTBL(uni2oem, FF_CODE_PAGE) / 4 - 1;
        li = 0;

        for (n = 16; n; n--)
        {
            i = li + (hi - li) / 2;

            if (uc == p[i * 2]) break;

            if (uc > p[i * 2])
            {
                li = i;
            }
            else
            {
                hi = i;
            }
        }

        if (n != 0) c = p[i * 2 + 1];
    }
}

return c;
}

WCHAR ff_oem2uni ( /* Returns Unicode character in UTF-16, zero on error */
    WCHAR oem, /* OEM code to be converted */
    WORD cp /* Code page for the conversion */
)
{
    const WCHAR *p;
    WCHAR c = 0;
    UINT i = 0, n, li, hi;

    if (oem < 0x80) /* ASCII? */
    {
        c = oem;
    }
    else /* Extended char */
    {
        if (cp == FF_CODE_PAGE) /* Is it valid code page? */
        {
            p = CVTBL(oem2uni, FF_CODE_PAGE);
            hi = sizeof CVTBL(oem2uni, FF_CODE_PAGE) / 4 - 1;
            li = 0;

            for (n = 16; n; n--)
            {
                i = li + (hi - li) / 2;
            }

```

```

        if (oem == p[i * 2]) break;

        if (oem > p[i * 2])
        {
            li = i;
        }
        else
        {
            hi = i;
        }
    }

    if (n != 0) c = p[i * 2 + 1];
}

return c;
}

```

以上两个函数，我们只需要关心对中文的处理，也就是对 936 的处理，这两个函数通过二分法来查找 UNICODE（或 GBK）码对应的 GBK（或 UNICODE）码。当我们将两个数组存放在外部 flash 的时候，这两个函数该可以修改为：

```

WCHAR ff_uni2oem ( /* Returns OEM code character, zero on error */
    DWORD uni,      /* UTF-16 encoded character to be converted */
    WORD cp         /* Code page for the conversion */
)
{
    WCHAR t[2];
    WCHAR c;
    uint32_t i, li, hi;
    uint16_t n;
    uint32_t gbk2uni_offset = 0;

    if (uni < 0x80)
    {
        c = uni; /* ASCII, 直接不用转换 */
    }
    else
    {
        hi = ftinfo.ugbksize / 2; /* 对半开 */
        hi = hi / 4 - 1;
        li = 0;

        for (n = 16; n; n--) /* 二分法查找 */
        {
            i = li + (hi - li) / 2;
            norflash_read((uint8_t *)&t, ftinfo.ugbkaddr + i * 4 +
                           gbk2uni_offset, 4); /* 读出 4 个字节 */

            if (uni == t[0]) break;

            if (uni > t[0])
            {
                li = i;
            }
            else
            {
                hi = i;
            }
        }

        c = n ? t[1] : 0;
    }

    return c;
}

```

```

}

WCHAR ff_oem2uni ( /* Returns Unicode character, zero on error */
    WCHAR oem,      /* OEM code to be converted */
    WORD  cp         /* Code page for the conversion */
)
{
    WCHAR t[2];
    WCHAR c;
    uint32_t i, li, hi;
    uint16_t n;
    uint32_t gbk2uni_offset = ftinfo.ugbksize / 2;

    if (oem < 0x80)
    {
        c = oem;    /* ASCII, 直接不用转换 */
    }
    else
    {
        hi = ftinfo.ugbksize / 2;          /* 对半开 */
        hi = hi / 4 - 1;
        li = 0;

        for (n = 16; n; n--)              /* 二分法查找 */
        {
            i = li + (hi - li) / 2;
            norflash_read((uint8_t *)&t, ftinfo.ugbkaddr + i * 4 +
                           gbk2uni_offset, 4); /* 读出 4 个字节 */

            if (oem == t[0]) break;

            if (oem > t[0])
            {
                li = i;
            }
            else
            {
                hi = i;
            }
        }

        c = n ? t[1] : 0;
    }

    return c;
}

```

代码中的 `ftinfo.ugbksize` 为我们刚刚生成的 `UNIGBK.bin` 的大小，而 `ftinfo.ugbkaddr` 是我们存放 `UNIGBK.bin` 文件的首地址，这里同样采用的是二分法查找。

修改后的 `ffunicode.c`，我们将其命名为 `myffunicode.c`，并保存在 `exfuns` 文件夹下，将工程 `FATFS` 组下的 `ffunicode.c` 删除，然后重新添加 `myffunicode.c` 到 `FATFS` 组下，`myffunicode.c` 的源码就不贴出来了，其实就是在 `ffunicode.c` 的基础上去掉了两个大数组，然后对 `ff_uni2oem` 和 `ff_oem2uni` 两个函数进行了修改，详见本例程源码。

关于 `ffunicode.c` 的修改，我们就介绍到这。

51.2 硬件设计

1. 例程功能

本实验开机的时候程序通过预设值的标记位检测 `norflash` 中是否已经存在字库，如果存在，则按次序显示汉字（三种字体都显示）。如果没有，则检测 `SD` 卡和文件系统，并查找 `SYSTEM` 文件夹下的 `FONT` 文件夹，在该文件夹内查找 `UNIGBK.BIN`、`GBK12.FON`、`GBK16.FON` 和 `GBK24.FON` 这几个文件的由来，我们在前面已经介绍过了。在检测到这些文件之后，就开始更

新字库，更新完毕才开始显示汉字。通过按按键 KEY0，可以强制更新字库。

LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 – PB5
- 2) 独立按键
KEY0 – PE4
- 3) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) micro SD 卡
- 6) NOR FLASH，需要用到它来存储汉字库

51.3 程序设计

51.3.1 程序流程图

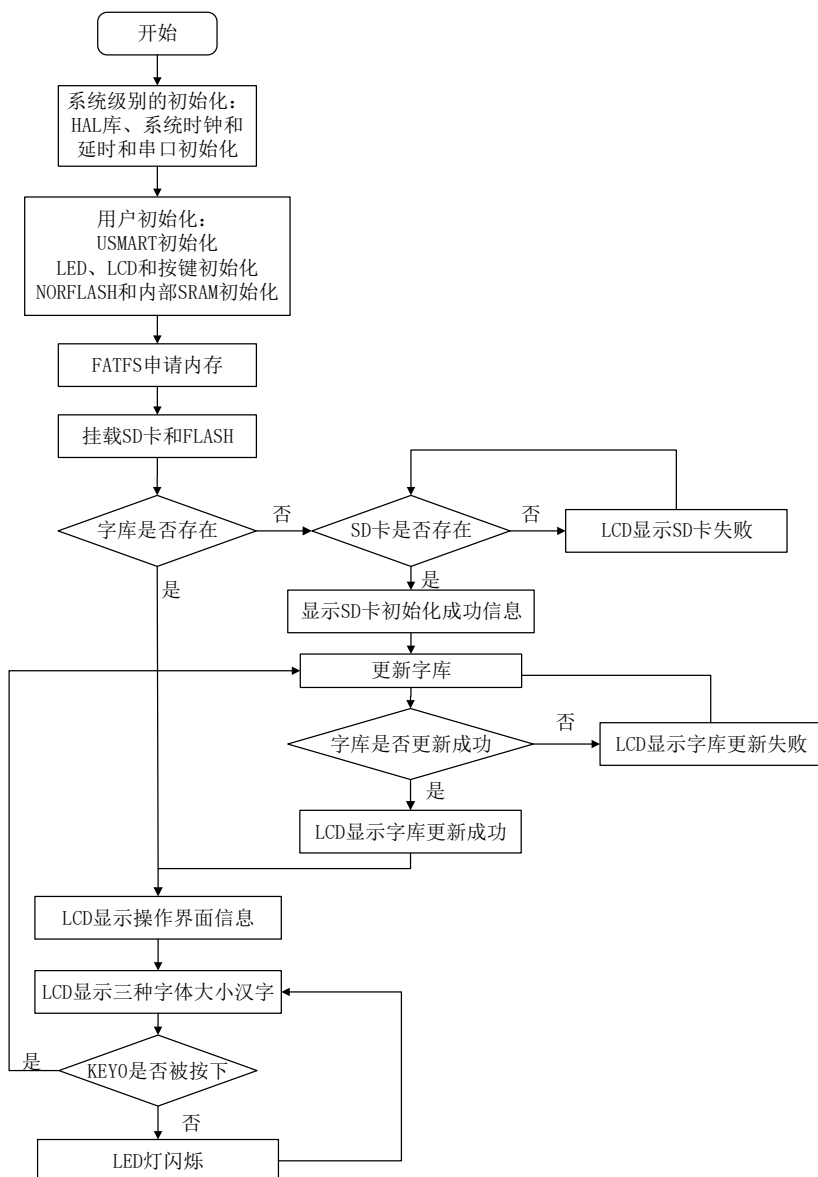


图 51.3.1.1 汉字显示实验程序流程图

51.3.2 程序解析

1. TEXT 代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。TEXT 驱动源码包括四个文件：text.c、text.h、font.c 和 font.h。

汉字显示实验代码主要分为两部分：一部分就是对字库的更新，另一部分就是对汉字的显示。字库的更新代码放在 font.c 和 font.h 文件中，汉字的显示代码就放在 text.c 和 text.h 中。

下面我们介绍一下有关字库操作的代码，首先我们先看一下 font.h 文件中字库信息结构体定义，其代码如下：

```
/* 字库信息结构体定义
 * 用来保存字库基本信息，地址，大小等
 */
__packed typedef struct
{
    uint8_t fontok;           /* 字库存在标志，0XAA，字库正常；其他，字库不存在 */
    uint32_t ugbkaddr;        /* unigbk 的地址 */
    uint32_t ugbksize;        /* unigbk 的大小 */
    uint32_t f12addr;         /* gbk12 地址 */
    uint32_t gbk12size;       /* gbk12 的大小 */
    uint32_t f16addr;         /* gbk16 地址 */
    uint32_t gbk16size;       /* gbk16 的大小 */
    uint32_t f24addr;         /* gbk24 地址 */
    uint32_t gbk24size;       /* gbk24 的大小 */
} _font_info;
```

这个结构体用于记录字库的首地址以及字库大小等信息，总共占用 33 个字节，第一个字节用来标识字库是否 OK，其他的用来记录地址和文件大小。因为我们将 NORFLASH（25Q128）的前 12M 字节给了 FATFS 管理（用做本地磁盘），12M 字节后紧跟 3 个字库+UNIGBK.BIN 总大小 3.09M 字节 791 个扇区，在 15.10M 字节后，预留了 100K 字节给用户自己使用。所以，我们的存储地址是从 12*1024*1024 处开始的。最开始的 33 个字节给 _font_info 用，用于保存 _font_info 结构体数据，之后是 UNIGBK.BIN、GBK12.FON、GBK16.FON 和 GBK24.FON。

下面介绍 font.c 文件中几个重要的函数。

字库初始化函数也是利用其存储顺序，进行检查字库，其定义如下：

```
/**
 * @brief      初始化字体
 * @param      无
 * @retval     0，字库完好；其他，字库丢失；
 */
uint8_t fonts_init(void)
{
    uint8_t t = 0;

    while (t < 10) /* 连续读取 10 次，都是错误，说明确实是有问题，得更新字库了 */
    {
        t++;
        /* 读出 ftinfo 结构体数据 */
        norflash_read((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo));

        if (ftinfo.fontok == 0XAA)
        {
            break;
        }

        delay_ms(20);
    }

    if (ftinfo.fontok != 0XAA)
    {

```



```

        return 1;
    }

    return 0;
}

```

这里就是把 NORFLASH 的 12M 地址的 33 个字节数据读取出来，进而判断字库结构体 ftinfo 的字库标记 fontok 是否为 AA，确定字库是否完好。

有人会有疑问：ftinfo.fontok 是在哪里赋值 AA 呢？肯定是字库更新完毕后，给该标记赋值的，那下面就来看一下是不是这样子，字库更新函数定义如下：

```

/**
 * @brief      更新字体文件
 * @note       所有字库一起更新 (UNIGBK, GBK12, GBK16, GBK24)
 * @param      x, y      : 提示信息的显示地址
 * @param      size      : 提示信息字体大小
 * @param      src       : 字库来源磁盘
 * @arg        "0:", SD 卡;
 * @arg        "1:", FLASH 盘
 * @param      color     : 字体颜色
 * @retval     0, 成功; 其他, 错误代码;
 */
uint8_t fonts_update_font(uint16_t x, uint16_t y, uint8_t size, uint8_t *src,
uint16_t color)
{
    uint8_t *pname;
    uint32_t *buf;
    uint8_t res = 0;
    uint16_t i, j;
    FIL *fftemp;
    uint8_t rval = 0;
    res = 0xFF;
    ftinfo.fontok = 0xFF;
    pname = mymalloc(SRAMIN, 100);          /* 申请 100 字节内存 */
    buf = mymalloc(SRAMIN, 4096);          /* 申请 4K 字节内存 */
    fftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 分配内存 */

    if (buf == NULL || pname == NULL || fftemp == NULL)
    {
        myfree(SRAMIN, fftemp);
        myfree(SRAMIN, pname);
        myfree(SRAMIN, buf);
        return 5; /* 内存申请失败 */
    }

    for (i = 0; i < 4; i++) /* 先查找文件 UNIGBK, GBK12, GBK16, GBK24 是否正常 */
    {
        strcpy((char *)pname, (char *)src);          /* copy src 内容到 pname */
        strcat((char *)pname, (char *)FONT_GBK_PATH[i]); /* 追加具体文件路径 */
        res = f_open(fftemp, (const TCHAR *)pname, FA_READ); /* 尝试打开 */

        if (res)
        {
            rval |= 1 << 7; /* 标记打开文件失败 */
            break;          /* 出错了, 直接退出 */
        }
    }

    myfree(SRAMIN, fftemp); /* 释放内存 */

    if (rval == 0)          /* 字库文件都存在. */
    {
        /* 提示正在擦除扇区 */
        lcd_show_string(x, y, 240, 320, size, "Erasing sectors... ", color);
    }
}

```

```

for (i = 0; i < FONTSECSIZE; i++) /* 先擦除字库区域,提高写入速度 */
{
    fonts_progress_show(x+20*size/2,y,size,FONTSECSIZE,i,color);/*进度显示*/
    /* 读出整个扇区的内容 */
    norflash_read((uint8_t *)buf, ((FONTINFOADDR / 4096) + i) * 4096,4096);

    for (j = 0; j < 1024; j++) /* 校验数据 */
    {
        if (buf[j] != 0xFFFFFFFF)break; /* 需要擦除 */
    }

    if (j != 1024)
    {
        norflash_erase_sector((FONTINFOADDR / 4096) + i); /*需要擦除的扇区*/
    }
}

for (i = 0; i < 4; i++) /* 依次更新 UNIGBK,GBK12,GBK16,GBK24 */
{
    lcd_show_string(x,y,240,320,size,FONT_UPDATE_REMIND_TBL[i],color);
    strcpy((char *)pname, (char *)src); /* copy src 内容到 pname */
    strcat((char *)pname, (char *)FONT_GBK_PATH[i]); /* 追加具体文件路径 */
    res = fonts_update_fontx(x+20*size/2,y,size,pname,i,color);/*更新字库*/
    if (res)
    {
        myfree(SRAMIN, buf);
        myfree(SRAMIN, pname);
        return 1 + i;
    }
}

ftinfo.fontok = 0XAA; /* 全部更新好了 */
norflash_write((uint8_t *)&ftinfo,FONTINFOADDR,sizeof(ftinfo));/*保存字库信息*/
}

myfree(SRAMIN, pname); /* 释放内存 */
myfree(SRAMIN, buf);
return rval; /* 无错误. */
}

```

函数的实现：动态申请内存→尝试打开文件(UNIGBK、GBK12、GBK16 和 GBK24)，确定文件是否存在→擦除字库→依次更新 UNIGBK、GBK12、GBK16 和 GBK24→写入 ftinfo 结构体信息。

在字库更新函数中能直接看到的是 ftinfo.fontok 成员被赋值，而其他成员在单个字库更新函数中被赋值，接下来分析一下更新某个字库函数，其代码如下：

```

/**
 * @brief      更新某一个字库
 * @param      x, y      : 提示信息的显示地址
 * @param      size      : 提示信息字体大小
 * @param      fpath      : 字体路径
 * @param      fx          : 更新的内容
 * @arg        0, ungbk;
 * @Arg        1, gbk12;
 * @arg        2, gbk16;
 * @arg        3, gbk24;
 * @param      color      : 字体颜色
 * @retval     0, 成功; 其他, 错误代码;
 */
static uint8_t fonts_update_fontx(uint16_t x, uint16_t y, uint8_t size, uint8_t
*fpath, uint8_t fx, uint16_t color)
{
    uint32_t flashaddr = 0;
    FIL *fftemp;
    uint8_t *tempbuf;

```

```
uint8_t res;
uint16_t bread;
uint32_t offx = 0;
uint8_t rval = 0;
fftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 分配内存 */
if (fftemp == NULL) rval = 1;

tempbuf = mymalloc(SRAMIN, 4096); /* 分配 4096 个字节空间 */

if (tempbuf == NULL) rval = 1;

res = f_open(fftemp, (const TCHAR *)fpath, FA_READ);

if (res) rval = 2; /* 打开文件失败 */

if (rval == 0)
{
    switch (fx)
    {
        case 0: /* 更新 UNIGBK.BIN */
            /* 信息头之后, 紧跟 UNIGBK 转换码表 */
            ftinfo.ugbkaddr = FONTINFOADDR + sizeof(ftinfo);
            ftinfo.ugbksize = fftemp->obj.objsize; /* UNIGBK 大小 */
            flashaddr = ftinfo.ugbkaddr;
            break;

        case 1: /* 更新 GBK12.FONT */
            /* UNIGBK 之后, 紧跟 GBK12 字库 */
            ftinfo.fl2addr = ftinfo.ugbkaddr + ftinfo.ugbksize;
            ftinfo.gbk12size = fftemp->obj.objsize; /* GBK12 字库大小 */
            flashaddr = ftinfo.fl2addr; /* GBK12 的起始地址 */
            break;

        case 2: /* 更新 GBK16.FONT */
            /* GBK12 之后, 紧跟 GBK16 字库 */
            ftinfo.fl6addr = ftinfo.fl2addr + ftinfo.gbk12size;
            ftinfo.gbk16size = fftemp->obj.objsize; /* GBK16 字库大小 */
            flashaddr = ftinfo.fl6addr; /* GBK16 的起始地址 */
            break;

        case 3: /* 更新 GBK24.FONT */
            /* GBK16 之后, 紧跟 GBK24 字库 */
            ftinfo.f24addr = ftinfo.fl6addr + ftinfo.gbk16size;
            ftinfo.gbk24size = fftemp->obj.objsize; /* GBK24 字库大小 */
            flashaddr = ftinfo.f24addr; /* GBK24 的起始地址 */
            break;
    }

    while (res == FR_OK) /* 死循环执行 */
    {
        res = f_read(fftemp, tempbuf, 4096, (UINT *)&bread); /* 读取数据 */
        if (res != FR_OK) break; /* 执行错误 */

        norflash_write(tempbuf, offx+flashaddr, bread); /* 从 0 开始写入 bread 个数据 */
        offx += bread;
        fonts_progress_show(x, y, size, fftemp->obj.objsize, offx, color); /* 进度显示 */
        if (bread != 4096) break; /* 读完了. */
    }

    f_close(fftemp);
}

myfree(SRAMIN, fftemp); /* 释放内存 */
```

```
myfree(SRAMIN, tempbuf);    /* 释放内存 */
return res;
}
```

单个字库更新函数，主要是对把字库从 SD 卡中读取数据，写入 NORFLASH。同时把字库大小和起始地址保存在 `ftinfo` 结构体里，在前面的整个字库更新函数中使用函数：

```
norflash_write((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo)); /*保存字库信息*/
```

结构体的所有成员一并写入到那 33 字节。有了这个字库信息结构体，就能很容易进行定位。结合前面的说到的根据地址偏移寻找汉字的点阵数据，我们就可以开始真正把汉字搬上屏幕中去了。

首先我们肯定需要获得汉字的 GBK 码，这里 MDK 已经帮我们实现了。这里用一个例子说明：

```
char* HZ_str = "正点原子";
printf("正点原子的'正'字GBK高位码: %#x\r\n", *HZ_str);
printf("正点原子的'正'字GBK低位码: %#x\r\n", *(HZ_str+1));
```

串口打印

```
正点原子的'正'字GBK高位码: 0xd5
正点原子的'正'字GBK低位码: 0xfd
```

在这里可以看出 MDK 识别汉字的方式是 GBK 码，换句话说就是 MDK 自动会把汉字看成是两个字节表示的东西。知道了要表示的汉字和其 GBK 码，那么就可以去找对应的点阵数据。在这里我们就定义了一个获取汉字点阵数据的函数，其定义如下：

```
/**
 * @brief      获取汉字点阵数据
 * @param      code    : 当前汉字编码 (GBK 码)
 * @param      mat     : 当前汉字点阵数据存放地址
 * @param      size    : 字体大小
 * @note       size 大小的字体,其点阵数据大小为: (size / 8 + ((size % 8) ? 1 : 0)) * (size) 字节
 *
 * @retval     无
 */
static void text_get_hz_mat(unsigned char *code, unsigned char *mat,
                           uint8_t size)
{
    unsigned char qh, ql;
    unsigned char i;
    unsigned long foffset;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    uint8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size);
    qh = *code;
    ql = *(++code);

    if (qh < 0x81 || ql < 0x40 || ql == 0xff || qh == 0xff) /* 非 常用汉字 */
    {
        for (i = 0; i < csize; i++)
        {
            *mat++ = 0x00; /* 填充满格 */
        }

        return; /* 结束访问 */
    }

    if (ql < 0x7f)
    {
        ql -= 0x40; /* 注意! */
    }
    else
    {
        ql -= 0x41;
    }
}
```

```
qh -= 0x81;
foffset = ((unsigned long)190 * qh + ql) * csize; /* 得到字库中的字节偏移量 */

switch (size)
{
    case 12:
        norflash_read(mat, foffset + ftinfo.fl12addr, csize);
        break;
    case 16:
        norflash_read(mat, foffset + ftinfo.fl16addr, csize);
        break;
    case 24:
        norflash_read(mat, foffset + ftinfo.fl24addr, csize);
        break;
}
```

函数实现的依据就是前面 51.1.3 小节讲到的两条公式：

当 $GBKL < 0X7F$ 时： $Hp = ((GBKH - 0x81) * 190 + GBKL - 0X40) * csize$;

当 $GBKL > 0X80$ 时： $Hp = ((GBKH - 0x81) * 190 + GBKL - 0X41) * csize$;

目标汉字的 GBK 码满足上面两条公式其一，就会得出与一个 GBK 对应的汉字点阵数据的偏移。在这个基础上，通过汉字点阵的大小，就可以从对应的字库提取目标汉字点阵数据。

在获取到点阵数据后，接下来就可以进行汉字显示，下面看一下汉字显示函数，其定义如下：

```
/**
 * @brief      显示一个指定大小的汉字
 * @param      x,y    : 汉字的坐标
 * @param      font    : 汉字 GBK 码
 * @param      size    : 字体大小
 * @param      mode    : 显示模式
 * @note       0, 正常显示 (不需要显示的点, 用 LCD 背景色填充, 即 g_back_color)
 * @note       1, 叠加显示 (仅显示需要显示的点, 不需要显示的点, 不做处理)
 * @param      color   : 字体颜色
 * @retval     无
 */
void text_show_font(uint16_t x, uint16_t y, uint8_t *font, uint8_t size,
                    uint8_t mode, uint16_t color)
{
    uint8_t temp, t, t1;
    uint16_t y0 = y;
    uint8_t *dzk;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    uint8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size);
    if (size != 12 && size != 16 && size != 24 && size != 32)
    {
        return; /* 不支持的 size */
    }

    dzk = mymalloc(SRAMIN, size); /* 申请内存 */
    if (dzk == 0) return; /* 内存不够了 */

    text_get_hz_mat(font, dzk, size); /* 得到相应大小的点阵数据 */

    for (t = 0; t < csize; t++)
    {
        temp = dzk[t]; /* 得到点阵数据 */

        for (t1 = 0; t1 < 8; t1++)
        {
            if (temp & 0x80)
            {
                lcd_draw_point(x, y, color); /* 画需要显示的点 */
            }
        }
    }
}
```

```

    }
    else if (mode == 0)      /* 如果非叠加模式，不需要显示的点，用背景色填充 */
    {
        lcd_draw_point(x, y, g_back_color); /* 填充背景色 */
    }

    temp <= 1;
    y++;
    if ((y - y0) == size)
    {
        y = y0;
        x++;
        break;
    }
}
}
myfree(SRAMIN, dzk);      /* 释放内存 */
}

```

汉字显示函数通过调用获取汉字点阵数据函数 `text_get_hz_mat` 就获取到点阵数据，使用 `lcd` 画点函数把点阵数据中“1”的点都画出来，最终会在 LCD 显示你所要表示的汉字。

其他函数就不多讲解，大家可以自行消化。

2. main.c 代码

在 `main.c` 里编写代码如下：

```

int main(void)
{
    uint32_t fontcnt;
    uint8_t i, j;
    uint8_t fontx[2];          /* GBK 码 */
    uint8_t key, t;

    HAL_Init();                /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);            /* 延时初始化 */
    usart_init(115200);        /* 串口初始化为 115200 */
    usmart_dev.init(72);      /* 初始化 USMART */
    led_init();                /* 初始化 LED */
    lcd_init();                /* 初始化 LCD */
    key_init();                /* 初始化按键 */
    norflash_init();          /* 初始化 NORFLASH */
    my_mem_init(SRAMIN);      /* 初始化内部 SRAM 内存池 */
    exfuncs_init();           /* 为 fatfs 相关变量申请内存 */
    f_mount(fs[0], "0:", 1);   /* 挂载 SD 卡 */
    f_mount(fs[1], "1:", 1);   /* 挂载 FLASH */

    while (fonts_init())      /* 检查字库 */
    {
        UPD:
        lcd_clear(WHITE);     /* 清屏 */
        lcd_show_string(30, 30, 200, 16, 16, "STM32F103", RED);

        while (sd_init())     /* 检测 SD 卡 */
        {
            lcd_show_string(30, 50, 200, 16, 16, "SD Card Failed!", RED);
            delay_ms(200);
            lcd_fill(30, 50, 200 + 30, 50 + 16, WHITE);
            delay_ms(200);
        }

        lcd_show_string(30, 50, 200, 16, 16, "SD Card OK", RED);
        lcd_show_string(30, 70, 200, 16, 16, "Font Updating...", RED);
    }
}

```



```

key = fonts_update_font(20, 90, 16, (uint8_t *) "0:", RED); /* 更新字库 */
while (key) /* 更新失败 */
{
    lcd_show_string(30, 90, 200, 16, 16, "Font Update Failed!", RED);
    delay_ms(200);
    lcd_fill(20, 90, 200 + 20, 90 + 16, WHITE);
    delay_ms(200);
}

lcd_show_string(30, 90, 200, 16, 16, "Font Update Success! ", RED);
delay_ms(1500);
lcd_clear(WHITE); /* 清屏 */
}

text_show_string(30, 30, 200, 16, "正点原子开发板", 16, 0, RED);
text_show_string(30, 50, 200, 16, "GBK 字库测试程序", 16, 0, RED);
text_show_string(30, 70, 200, 16, "正点原子@ALIENTEK", 16, 0, RED);
text_show_string(30, 90, 200, 16, "按 KEY0,更新字库", 16, 0, RED);

text_show_string(30, 110, 200, 16, "内码高字节:", 16, 0, BLUE);
text_show_string(30, 130, 200, 16, "内码低字节:", 16, 0, BLUE);
text_show_string(30, 150, 200, 16, "汉字计数器:", 16, 0, BLUE);

text_show_string(30, 180, 200, 24, "对应汉字为:", 24, 0, BLUE);
text_show_string(30, 204, 200, 16, "对应汉字(16*16)为:", 16, 0, BLUE);
text_show_string(30, 220, 200, 16, "对应汉字(12*12)为:", 12, 0, BLUE);

while (1)
{
    fontcnt = 0;
    for (i = 0x81; i < 0xff; i++) /* GBK 内码高字节范围为 0x81~0xFE */
    {
        fontx[0] = i;
        lcd_show_num(118, 110, i, 3, 16, BLUE); /* 显示内码高字节 */

        for (j = 0x40; j < 0xfe; j++) /* GBK 内码低字节范围 0x40~0x7E, 0x80~0xFE */
        {
            if (j == 0x7f) continue;
            fontcnt++;
            lcd_show_num(118, 130, j, 3, 16, BLUE); /* 显示内码低字节 */
            lcd_show_num(118, 150, fontcnt, 5, 16, BLUE); /* 汉字计数显示 */
            fontx[1] = j;
            text_show_font(30 + 132, 180, fontx, 24, 0, BLUE);
            text_show_font(30 + 144, 204, fontx, 16, 0, BLUE);
            text_show_font(30 + 108, 220, fontx, 12, 0, BLUE);
            t = 200;

            while (t-- > 0) /* 延时,同时扫描按键 */
            {
                delay_ms(1);
                key = key_scan(0);
                if (key == KEY0_PRES)
                {
                    goto UPD; /* 跳转到 UPD 位置(强制更新字库) */
                }
            }
            LED0_TOGGLE();
        }
    }
}
}

```

main 函数实现了我们在硬件设计例程功能所表述的一致，至此整个软件设计就完成了。

51.4 下载验证

本例程支持 12*12、16*16 和 24*24 等三种字体的显示，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 开始显示三种大小的汉字及内码如图 51.4.1 所示：

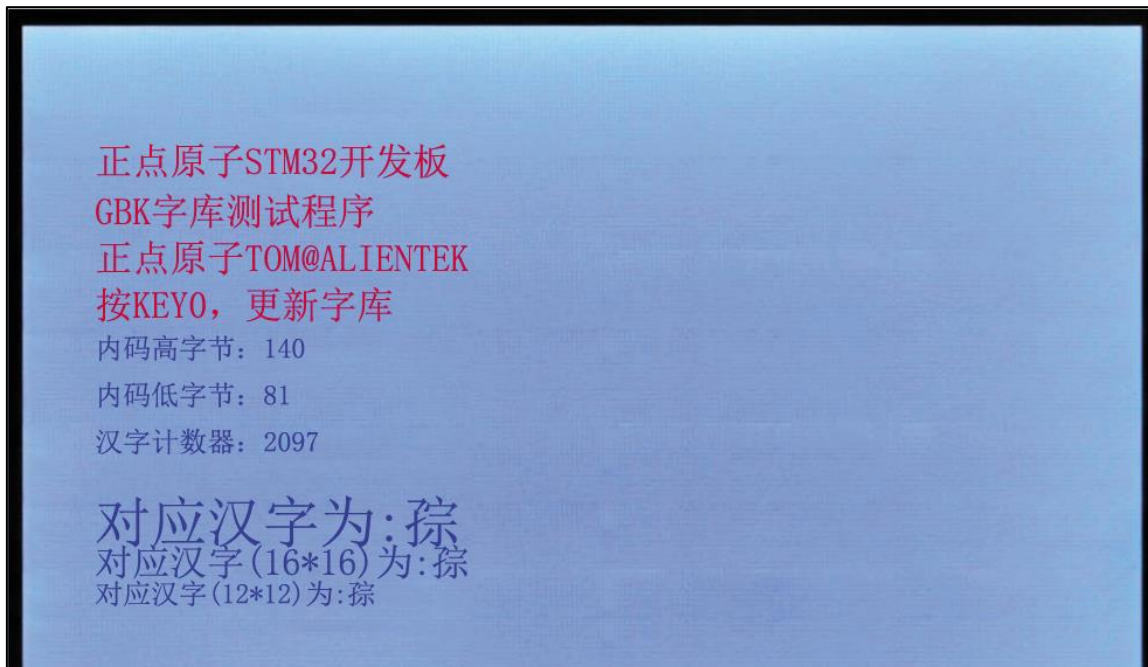


图 51.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为板子在出厂的时候都是测试过的，里面刷了综合测试程序，已经把字库写入到 NORFLASH 里面，所以并不会提示更新字库。如果你想要更新字库，就需要先找一张 SD 卡，把 A 盘资料\5，SD 卡根目录文件 文件夹下面的 SYSTEM 文件夹拷贝到 SD 卡根目录下，插入开发板，并按复位，之后，在显示汉字的时候，按下 KEY0，就可以开始更新字库。字库更新界面如图 51.4.2 所示：



图 51.4.2 汉字字库更新界面

此外我们还可以使用 USART 来测试该实验。通过 USART 调用 text_show_string 或者 text_show_string_middle 来实现任意位置显示任何字符串，有兴趣的朋友可以尝试一下。

第五十二章 图片显示实验

在开发产品的时候，很多时候，我们都会用到图片解码，在本章中，我们将向大家介绍如何通过 STM32F1 来解码 BMP/JPG/JPEG/GIF 等图片，并在 LCD 上显示出来。本章分为如下几个部分：

- 52.1 图片格式简介
- 52.2 硬件设计
- 52.3 软件设计
- 52.4 下载验证

52.1 图片格式介绍

我们常用的图片格式有很多，一般最常用的有三种：JPEG（或 JPG）、BMP 和 GIF。其中 JPEG（或 JPG）和 BMP 是静态图片，而 GIF 则是可以实现动态图片。下面，我们简单介绍一下这三种图片格式。

52.1.1 BMP 编码简介

我们常用的图片格式有很多，一般最常用的有三种：JPEG（或 JPG）、BMP 和 GIF。其中 JPEG（或 JPG）和 BMP 是静态图片，而 GIF 则是可以实现动态图片。下面，我们简单介绍一下这三种图片格式。

首先，我们来看看 BMP 图片格式。BMP（全称 Bitmap）是 Window 操作系统中的标准图像文件格式，文件后缀名为“.bmp”，使用非常广。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，BMP 文件所占用的空间很大，但是没有失真。BMP 文件的图像深度可选 1bit、4bit、8bit、16bit、24bit 及 32bit。BMP 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

典型的 BMP 图像文件由四部分组成：

- ① 位图头文件数据结构，它包含 BMP 图像文件的类型、显示内容等信息；
- ② 位图信息数据结构，它包含有 BMP 图像的宽、高、压缩方法，以及定义颜色等信息
- ③ 调色板，这个部分是可选的，有些位图需要调色板，有些位图，比如真彩色图（24 位的 BMP）就不需要调色板；
- ④ 位图数据，这部分的内容根据 BMP 位图使用的位数不同而不同，在 24 位图中直接使用 RGB，而其他的小于 24 位的使用调色板中颜色索引值。

关于 BMP 的详细介绍，请参考光盘的《BMP 图片文件详解.pdf》。

52.1.2 JPEG 编码简介

JPEG 是 Joint Photographic Experts Group(联合图像专家组)的缩写，文件后缀名为“.jpg”或“.jpeg”，是最常用的图像文件格式，由一个软件开发联合会组织制定，同 BMP 格式不同，JPEG 是一种有损压缩格式，能够将图像压缩在很小的储存空间，图像中重复或不重要的资料会被丢失，因此容易造成图像数据的损伤（BMP 不会，但是 BMP 占用空间大）。尤其是使用过高的压缩比例，将使最终解压缩后恢复的图像质量明显降低，如果追求高品质图像，不宜采用过高压缩比例。但是 JPEG 压缩技术十分先进，它用有损压缩方式去除冗余的图像数据，在获得极高的压缩率的同时能展现十分丰富生动的图像，换句话说，就是可以用最少的磁盘空间得到较好的图像品质。而且 JPEG 是一种很灵活的格式，具有调节图像质量的功能，允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在 10:1 到 40:1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好。比如可以把 1.37Mb 的 BMP 位图文件压缩至 20.3KB。当然也可以在图像质量和文件尺寸之间找到平衡点。JPEG 格式压缩的主要是高频信息，对色彩的信息保留较好，适合应用于互联网，可减少图像的传输时间，可以支持 24bit 真彩

色，也普遍应用于需要连续色调的图像。

JPEG/JPG 的解码过程可以简单的概述为如下几个部分：

1) 从文件头读出文件的相关信息。

JPEG 文件数据分为文件头和图像数据两大部分，其中文件头记录了图像的版本、长宽、采样因子、量化表、哈夫曼表等重要信息。所以解码前必须将文件头信息读出，以备图像数据解码过程之用。

2) 从图像数据流读取一个最小编码单元(MCU)，并提取出里边的各个颜色分量单元。

3) 将颜色分量单元从数据流恢复成矩阵数据。

使用文件头给出的哈夫曼表，对分割出来的颜色分量单元进行解码，将其恢复成 8×8 的数据矩阵。

4) 8×8 的数据矩阵进一步解码。

此部分解码工作以 8×8 的数据矩阵为单位，其中包括相邻矩阵的直流系数差分解码、使用文件头给出的量化表反量化数据、反 Zig-zag 编码、隔行正负纠正、反向离散余弦变换等 5 个步骤，最终输出仍然是一个 8×8 的数据矩阵。

5) 颜色系统 YCrCb 向 RGB 转换。

将一个 MCU 的各个颜色分量单元解码结果整合起来，将图像颜色系统从 YCrCb 向 RGB 转换。

6) 排列整合各个 MCU 的解码数据。

不断读取数据流中的 MCU 并对其解码，直至读完所有 MCU 为止，将各 MCU 解码后的数据正确排列成完整的图像。JPEG 的解码本身是比较复杂的，这里 FATFS 的作者，提供了一个轻量级的 JPG/JPEG 解码库：TjpgDec，最少仅需 3KB 的 RAM 和 3.5KB 的 FLASH 即可实现 JPG/JPEG 解码，本例程采用 TjpgDec 作为 JPG/JPEG 的解码库，关于 TjpgDec 的详细使用，请参考“A 盘→6，软件资料→图片编解码→TjpgDec 技术手册”这个文档。

52.1.2 GIF 编码简介

GIF(Graphics Interchange Format)是 CompuServe 公司开发的图像文件存储格式，1987 年开发的 GIF 文件格式版本号是 GIF87a，1989 年进行了扩充，扩充后的版本号定义为 GIF89a。GIF 图像文件以数据块(block)为单位来存储图像的相关信息。一个 GIF 文件由表示图形/图像的数据块、数据子块以及显示图形/图像的控制信息块组成，称为 GIF 数据流(DataStream)。数据流中的所有控制信息块和数据块都必须在文件头(Header)和文件结束块(Trailer)之间。

GIF 文件格式采用了 LZW(Lempel-Ziv-Walch)压缩算法来存储图像数据，定义了允许用户为图像设置背景的透明(transparency)属性。此外，GIF 文件格式可在一个文件中存放多幅彩色图形/图像。如果在 GIF 文件中存放有多幅图，它们可以像演幻灯片那样显示或者像动画那样演示。

一个 GIF 文件的结构可分为文件头(FileHeader)、GIF 数据流(GIFDataStream)和文件终结器(Trailer)三个部分。文件头包含 GIF 文件署名(Signature)和版本号(Version)；GIF 数据流由控制标识符、图象块(ImageBlock)和其他的一些扩展块组成；文件终结器只有一个值为 0x3B 的字符(';')表示文件结束。

关于 GIF 的详细介绍，请参考光盘 GIF 解码相关资料。图片格式简介，我们就介绍到这里。

52.2 硬件设计

1. 例程功能

开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY2 可以快速浏览下一张和上一张，KEY_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。如果未找到 PICTURE 文件夹/任何图片文件，则提示错误。同样我们也是用 DS0 来指示程序正在运行。

2. 硬件资源

- 1) LED 灯
DS0: LED0 - PB5
DS1: LED1 - PE5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 4) 独立按键: KEY0 - PE4、KEY1 - PE3、WK_UP - PA0
- 5) SD 卡, 通过 SDIO 连接
- 6) NOR FLASH(SPI FLASH 芯片, 连接在 SPI 上)

52.3 程序设计

52.3.1 程序流程图

下面看看本实验的程序流程图:

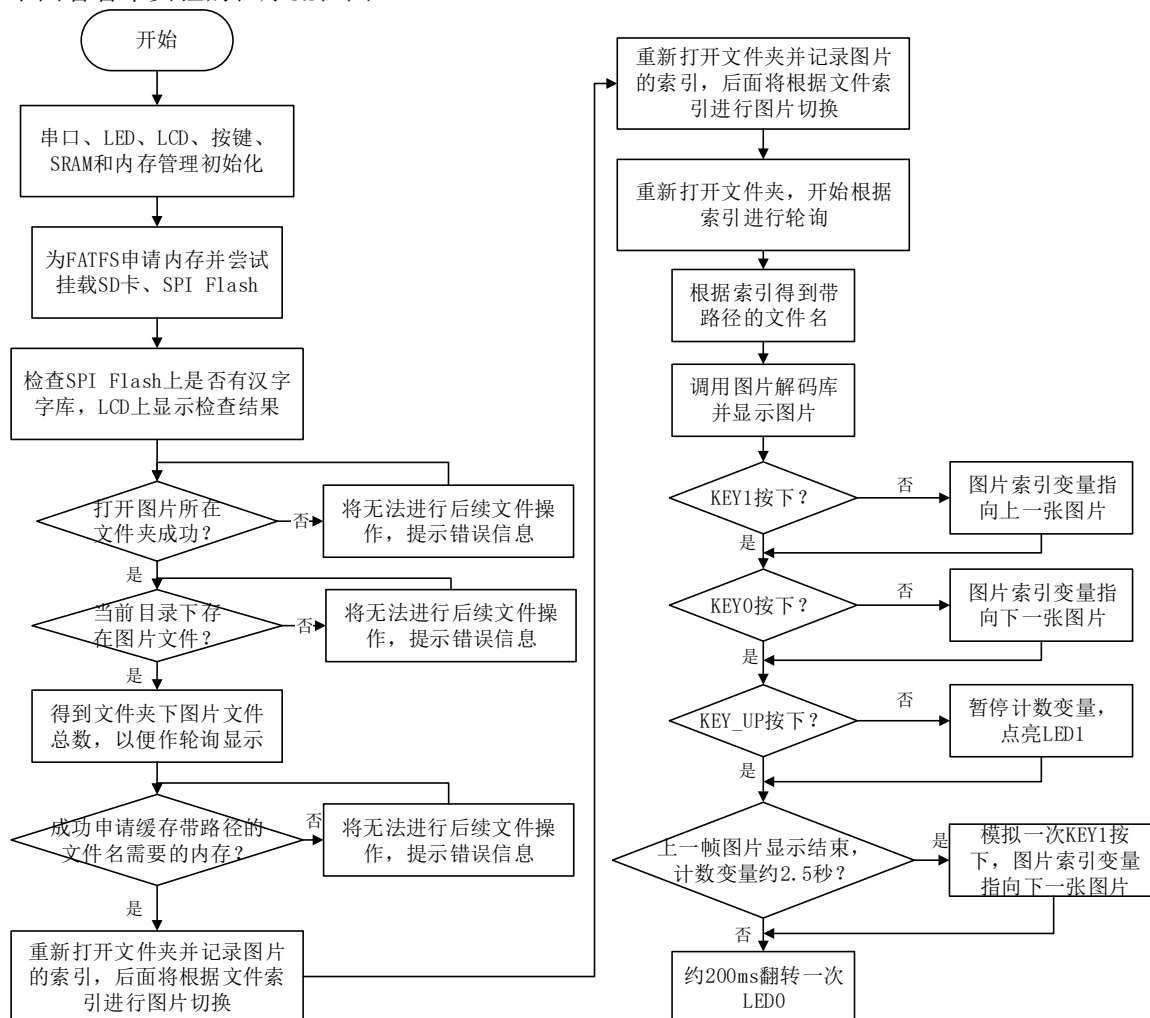


图 52.3.1.1 照相机实验程序流程图

本程序主要靠文件操作，打开指定位置的图片并调用图片解码库解码后显示不同格式的图片。我们加入了按键进行人机交互，以控制图片的显示切换等。

52.3.2 程序解析

1. PICTURE 代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PICTURE 驱动源码包括八个文件：bmp.c、bmp.h、tjpgd.c、tjpgd.h、gif.c、gif.h、piclib.c 和 piclib.h。

其中：

bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；

tjpgd.c 和 tjpgd.h 用于实现对 jpeg/jpg 文件的解码；

gif.c 和 gif.h 用于实现对 gif 文件的解码；

这几个代码太长了，而且也有规定的标准，需要结合各个图片编码的格式来编写，所以我们在这里不贴出来，大家查看光盘中的源码的实现过程即可。下面我们重点讲解这几个解码库对应到我们的 LCD 的显示部分。

1) 解码库的控制句柄_pic_phy 和_pic_info

我们使用这个接口，把解码后的图形数据与 LCD 的实际操作对应起来。为了方便去显示图片，我们需要将图片的信息与我们的 LCD 联系上。这里我们定义了_pic_phy 和_pic_info 分别用于定义图片解码库的 LCD 操作和存放解码后的图片尺寸颜色信息。它们的定义如下：

```
/* 在移植的时候,必须由用户自己实现这几个函数 */
typedef struct
{
    /* 读点函数 */
    uint32_t(*read_point)(uint16_t, uint16_t);
    /* 画点函数 */
    void(*draw_point)(uint16_t, uint16_t, uint32_t);
    /* 单色填充函数 */
    void(*fill)(uint16_t, uint16_t, uint16_t, uint16_t, uint32_t);
    /* 画水平线函数 */
    void(*draw_hline)(uint16_t, uint16_t, uint16_t, uint16_t);
    /* 颜色填充 */
    void(*fillcolor)(uint16_t, uint16_t, uint16_t, uint16_t, uint16_t *);
} _pic_phy;

/* 图像信息 */
typedef struct
{
    uint16_t lcdwidth;      /* LCD 的宽度 */
    uint16_t lcdheight;     /* LCD 的高度 */
    uint32_t ImgWidth;      /* 图像的实际宽度和高度 */
    uint32_t ImgHeight;
    uint32_t Div_Fac;       /* 缩放系数 (扩大了 8192 倍的) */
    uint32_t S_Height;      /* 设定的高度和宽度 */
    uint32_t S_Width;
    uint32_t S_XOFF;        /* x 轴和 y 轴的偏移量 */
    uint32_t S_YOFF;
    uint32_t staticx;       /* 当前显示到的 x y 坐标 */
    uint32_t staticy;
} _pic_info;
```

在 piclib.c 文件中，我们用上述类型定义了两个结构体，声明如下：

```
_pic_info picinfo;      /* 图片信息 */
_pic_phy pic_phy;       /* 图片显示物理接口 */
```

2) piclib_init 函数

piclib_init 函数，该函数用于初始化图片解码的相关信息，用于定义解码后的 LCD 操作。具体定义如下：

```
/**
 * @brief      画图初始化
 * @note      在画图之前,必须先调用此函数, 指定相关函数
 * @param      无
```



```

* @retval    无
*/
void piclib_init(void)
{
    pic_phy.read_point = lcd_read_point;    /* 读点函数实现,仅 BMP 需要 */
    pic_phy.draw_point = lcd_draw_point;    /* 画点函数实现 */
    pic_phy.fill = lcd_fill;                /* 填充函数实现,仅 GIF 需要 */
    pic_phy.draw_hline = lcd_draw_hline;    /* 画线函数实现,仅 GIF 需要 */
    pic_phy.fillcolor = piclib_fill_color;  /* 颜色填充函数实现,仅 TJPGD 需要 */

    picinfo.lcdwidth = lcddev.width;        /* 得到 LCD 的宽度像素 */
    picinfo.lcdheight = lcddev.height;      /* 得到 LCD 的高度像素 */

    picinfo.ImgWidth = 0;                   /* 初始化宽度为 0 */
    picinfo.ImgHeight = 0;                  /* 初始化高度为 0 */
    picinfo.Div_Fac = 0;                    /* 初始化缩放系数为 0 */
    picinfo.S_Height = 0;                   /* 初始化设定的高度为 0 */
    picinfo.S_Width = 0;                    /* 初始化设定的宽度为 0 */
    picinfo.S_XOFF = 0;                     /* 初始化 x 轴的偏移量为 0 */
    picinfo.S_YOFF = 0;                     /* 初始化 y 轴的偏移量为 0 */
    picinfo.staticx = 0;                    /* 初始化当前显示到的 x 坐标为 0 */
    picinfo.staticy = 0;                    /* 初始化当前显示到的 y 坐标为 0 */
}

```

● 函数描述:

初始化图片解码的相关信息,这些函数必须由用户在外部实现。我们使用之前 LCD 的操作函数对这个结构体中的绘制操作:画点、画线、画圆等定义与我们的 LCD 操作对应起来。具体这些操作可以查看 TFT LCD 一节的描述。

● 函数形参:

无。

● 函数返回值:

无。

3) piclib_alpha_blend 函数

RGB 色彩中,一个标准像素由 32 位组成:透明度(8bit)+R(8bit)+B(8bit)+B(8bit),8 位的 α 通道(alpha channel)位表示该像素如何产生特技效果,即通常我们说的半透明。alpha 的取值一般为 0 到 255。为 0 时,表示是全透明的,即图片是看不见的。为 255 时,表示图片是显示原始图的。中间值即为半透明状态。计算 alpha blending 时,通常的方法是将源像素的 RGB 值,分别与目标像素(如背景)的 RGB 按比例混合,最后得到一个混合后的 RGB 值。函数定义如下:

```

/**
 * @brief      快速 ALPHA BLENDING 算法
 * @param      src          : 颜色数
 * @param      dst          : 目标颜色
 * @param      alpha        : 透明程度(0~32)
 * @retval     混合后的颜色
 */
uint16_t piclib_alpha_blend(uint16_t src, uint16_t dst, uint8_t alpha)
{
    uint32_t src2;
    uint32_t dst2;
    /* Convert to 32bit |----GGGGGG----RRRRR-----BBBBB| */
    src2 = ((src << 16) | src) & 0x07E0F81F;
    dst2 = ((dst << 16) | dst) & 0x07E0F81F;
    dst2 = (((dst2 - src2) * alpha) >> 5) + src2 & 0x07E0F81F;
    return (dst2 >> 16) | dst2;
}

```

● 函数描述:

piclib_alpha_blend 函数,该函数用于实现半透明效果,在小格式(图片分辨率小于 LCD 分辨率)bmp 解码的时候,可能被用到。

- **函数形参:**

形参 1 是为 RGB 色彩编号, 这里我们使用的是 RGB565 模式, 故只有 16 位;

形参 2 是目标像素, 使用时我们一般指背景颜色。

形参 3 是透明度: 有效范围为 0~255, 0 表示全透明, 255 表示不透明。

- **函数返回值:**

返回计算后的透明度颜色数值。

4) piclib_ai_draw_init 函数

对于给定区域, 为了显示更好看, 一般会选择图片居中显示, 此函数实现此功能, 把图片在显示区域中居中。

```
/**
 * @brief      初始化智能画点
 * @param      无
 * @retval     无
 */
void piclib_ai_draw_init(void)
{
    float temp, temp1;
    temp = (float)picinfo.S_Width / picinfo.ImgWidth;
    temp1 = (float)picinfo.S_Height / picinfo.ImgHeight;
    if (temp < temp1) temp1 = temp; /* 取较小的那个 */
    if (temp1 > 1) temp1 = 1;
    /* 使图片处于所给区域的中间 */
    picinfo.S_XOFF += (picinfo.S_Width - temp1 * picinfo.ImgWidth) / 2;
    picinfo.S_YOFF += (picinfo.S_Height - temp1 * picinfo.ImgHeight) / 2;
    temp1 *= 8192; /* 扩大 8192 倍 */
    picinfo.Div_Fac = temp1;
    picinfo.staticx = 0xffff;
    picinfo.staticy = 0xffff; /* 放到一个不可能的值上面 */
}
```

- **函数描述:**

piclib_ai_draw_init 函数, 该函数使解码后的图片信息处于所给的区域的中间。

- **函数形参:**

无。

- **函数返回值:**

无。我们可以在显示实例中测试加与不加此函数的显示效果差异。

5) piclib_is_element_ok 函数

piclib_is_element_ok 函数定义如下:

```
/**
 * @brief      判断这个像素是否可以显示
 * @param      x, y      : 像素原始坐标
 * @param      chg       : 功能变量
 * @param      无
 * @retval     操作结果
 * @arg        0      , 不需要显示
 * @arg        1      , 需要显示
 */
inline uint8_t piclib_is_element_ok(uint16_t x, uint16_t y, uint8_t chg)
{
    if (x != picinfo.staticx || y != picinfo.staticy)
    {
        if (chg == 1)
        {
            picinfo.staticx = x;
            picinfo.staticy = y;
        }
        return 1;
    }
    else
    {

```

```

        return 0;
    }
}

```

● 函数描述:

piclib_is_element_ok 函数，该函数用于判断一个点是不是应该显示出来，在图片缩放的时候该函数是必须用到的。这里用__inline 修饰，保证该部分的代码不被优化。

● 函数形参:

无。

● 函数返回值:

1: 需要显示。

0: 不需要显示。

其它函数使用到时，根据此返回值进行判定显示操作。

6) piclib_ai_load_picfile 函数

piclib_ai_load_picfile 帮助我们得到需要显示的图片信息并助于下一步的绘制。本函数需要结合文件系统来操作，图片根据后缀区分并且在文件夹在保存是我们在 PC 下的习分类，也是我们处理和分类图片的最方便的方式。

```

/**
 * @brief      智能画图
 * @note      图片仅在 x,y 和 width, height 限定的区域内显示.
 *
 * @param      filename      : 包含路径的文件名(.bmp/.jpg/.jpeg/.gif 等)
 * @param      x, y          : 起始坐标
 * @param      width, height : 显示区域
 * @param      fast          : 使能快速解码
 * @arg        0, 不使能
 * @arg        1, 使能
 * @note      图片尺寸小于等于液晶分辨率,才支持快速解码
 * @retval     res           : 操作结果 0, 成功 其他, 错误码
 */
uint8_t piclib_ai_load_picfile(const uint8_t *filename, uint16_t x, uint16_t y,
uint16_t width, uint16_t height, uint8_t fast)
{
    uint8_t res; /* 返回值 */
    uint8_t temp;
    if((x + width) > picinfo.lcdwidth)return PIC_WINDOW_ERR; /* x 坐标超范围了 */
    if((y + height) > picinfo.lcdheight)return PIC_WINDOW_ERR; /* y 坐标超范围了 */
    /* 得到显示方框大小 */
    if (width == 0 || height == 0)return PIC_WINDOW_ERR; /* 窗口设定错误 */
    picinfo.S_Height = height;
    picinfo.S_Width = width;
    /* 显示区域无效 */
    if (picinfo.S_Height == 0 || picinfo.S_Width == 0)
    {
        picinfo.S_Height = lcddev.height;
        picinfo.S_Width = lcddev.width;
        return FALSE;
    }
    if (pic_phy.fillcolor == NULL)fast = 0; /* 颜色填充函数未实现,不能快速显示 */
    /* 显示的开始坐标点 */
    picinfo.S_YOFF = y;
    picinfo.S_XOFF = x;
    /* 文件名传递 */
    temp = exfuns_file_type((uint8_t *)filename); /* 得到文件的类型 */
    switch (temp)
    {
        case T_BMP:
            res = stdbmp_decode(filename); /* 解码 bmp */
            break;
        case T_JPG:

```

```

    case T_JPEG:
        res = jpg_decode(filename, fast);    /* 解码 JPG/JPEG */
        break;
    case T_GIF:
        res = gif_decode(filename, x, y, width, height);    /* 解码 gif */
        break;
    default:
        res = PIC_FORMAT_ERR;                /* 非图片格式!!! */
        break;
}
return res;
}

```

● 函数描述:

piclib_ai_load_picfile 函数，整个图片显示的对外接口，外部程序，通过调用该函数，可以实现 bmp、jpg/jpeg 和 gif 的显示，该函数根据输入文件的后缀名，判断文件格式，然后交给相应的解码程序（bmp 解码/jpeg 解码/gif 解码），执行解码，完成图片显示。

● 函数形参:

形参 1 filename 是文件的路径名，具体可以参考 FATFS 一节的描述，为字符口，我们的例程采用的是 SD 卡存图片，故一般为"0:/PICTURE/*.GIF"等类似格式。

形参 2 为画图的起始 x 坐标;

形参 3 为画图的起始 y 坐标;

形参 4 的 width 和形参 5 的 height 形成了以 x、y 为起点的(x,y)~(x+width,y+height)的矩形显示区域，对屏幕坐标不理解的可能参考我们的 TFT LCD 一节的描述。

形参 6 根据我们的 LCD 进行适应的一个快速解的操作，仅 jpg/jpeg 模式下有效。

这里用到的 exfuns_file_type()函数是我们前面 FATFS 一节提到的 FATFS 扩展应用，我们用这个函数来判断文件类型，方便我们进行程序设计。这部分参考文件系统下 exfuns 文件夹下的相关文件。

● 函数返回值:

0: 成功 其他: 错误码

由于图片显示需要用到大内存，我们使用动态内存分配来实现，我们仍使用我们自定的内存管理函数来管理程序内存。申请内存函数 piclib_mem_malloc() 和内存释放函数 piclib_mem_free()的实现就比较简单了，大家参考光盘的源码即可。

2. main.c 代码

main.c 函数我们利用 FATFS 的接口来操作和查找图片文件，我们在 microSD/SD 卡的根目录下新建一个 PICTURE 文件夹，然后放置我们准备要显示的 BMP、JPG、GIF 图片，接下来按我们程序流程图设置的思路，先扫描图像文件的数量并切换显示，并加入按键支持图片翻页，主要的代码如下所示：

```

int main(void)
{
    uint8_t res;
    DIR picdir;                /* 图片目录 */
    FILINFO *picfileinfo;      /* 文件信息 */
    uint8_t *pname;            /* 带路径的文件名 */
    uint16_t totpicnum;        /* 图片文件总数 */
    uint16_t curindex;         /* 图片当前索引 */
    uint8_t key;               /* 键值 */
    uint8_t pause = 0;         /* 暂停标记 */
    uint8_t t;
    uint16_t temp;
    uint32_t *picoffsettbl;    /* 图片文件 offset 索引表 */

    ... /* 省略 LED、按键、LCD、文件系统和 malloc 等初始化过程 */.....

    while (f_opendir(&picdir, "0:/PICTURE"))    /* 打开图片文件夹 */

```

```

{
    text_show_string(30, 150, 240, 16, "PICTURE 文件夹错误!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 150, 240, 186, WHITE);          /* 清除显示 */
    delay_ms(200);
}
totpicnum = pic_get_tnum((uint8_t *)"0:/PICTURE"); /* 得到总有效文件数 */
while (totpicnum == NULL)                          /* 图片文件为 0 */
{
    text_show_string(30, 150, 240, 16, "没有图片文件!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 150, 240, 186, WHITE);          /* 清除显示 */
    delay_ms(200);
}
picfileinfo = (FILINFO *)mymalloc(SRAMIN, sizeof(FILINFO)); /* 申请内存 */
pname = mymalloc(SRAMIN, FF_MAX_LFN * 2 + 1); /* 为带路径的文件名分配内存 */
/* 申请 4*totpicnum 个字节的内存,用于存放图片索引 */
picoffsettbl = mymalloc(SRAMIN, 4 * totpicnum);
while (!picfileinfo || !pname || !picoffsettbl) /* 内存分配出错 */
{
    text_show_string(30, 150, 240, 16, "内存分配失败!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 150, 240, 186, WHITE);          /* 清除显示 */
    delay_ms(200);
}
/* 记录索引 */
res = f_opendir(&picdir, "0:/PICTURE");          /* 打开目录 */
if (res == FR_OK)
{
    curindex = 0;                                /* 当前索引为 0 */
    while (1)                                    /* 全部查询一遍 */
    {
        temp = picdir.dptr;                      /* 记录当前 dptr 偏移 */
        res = f_readdir(&picdir, picfileinfo); /* 读取目录下的一个文件 */
        if (res != FR_OK || picfileinfo->fname[0] == 0)
            break;                                /* 错误了/到末尾了,退出 */
        res = exfuns_file_type((uint8_t *)picfileinfo->fname);
        if ((res & 0XF0) == 0X50) /* 取高四位,看看是不是图片文件 */
        {
            picoffsettbl[curindex] = temp;        /* 记录索引 */
            curindex++;
        }
    }
}
text_show_string(30, 150, 240, 16, "开始显示...", 16, 0, RED);
delay_ms(1500);
piclib_init();                                  /* 初始化画图 */
curindex = 0;                                  /* 从 0 开始显示 */
res = f_opendir(&picdir, (const TCHAR *)"0:/PICTURE"); /* 打开目录 */
while (res == FR_OK)                            /* 打开成功 */
{
    dir_sdi(&picdir, picoffsettbl[curindex]);    /* 改变当前目录索引 */
    res = f_readdir(&picdir, picfileinfo);        /* 读取目录下的一个文件 */
    if (res != FR_OK || picfileinfo->fname[0] == 0)
        break; /* 错误了/到末尾了,退出 */
    strcpy((char *)pname, "0:/PICTURE/");        /* 复制路径(目录) */
    /*将文件名接在后面 */
    strcat((char *)pname, (const char *)picfileinfo->fname);
    lcd_clear(BLACK);
    /* 显示图片 */
    piclib_ai_load_picfile(pname, 0, 0, lcddev.width, lcddev.height, 1);
}

```

```

/* 显示图片名字 */
text_show_string(2, 2, lcddev.width, 16, (char *)pname, 16, 1, RED);
t = 0;
while (1)
{
    key = key_scan(0); /* 扫描按键 */
    if (t > 250)
    {
        key = 1; /* 模拟一次按下 KEY0 */
        if ((t % 20) == 0)
        {
            LED0_TOGGLE(); /* LED0 闪烁,提示程序正在运行. */
        }
        if (key == KEY1_PRES) /* 上一张 */
        {
            if (curindex)
            {
                curindex--;
            }
            else
            {
                curindex = totpicnum - 1;
            }
            break;
        }
        else if (key == KEY0_PRES) /* 下一张 */
        {
            curindex++;
            if (curindex >= totpicnum)
                curindex = 0; /* 到末尾的时候,自动从头开始 */
            break;
        }
        else if (key == WKUP_PRES)
        {
            pause = !pause;
            LED1(!pause); /* 暂停的时候 LED1 亮. */
        }
        if (pause == 0)
            t++;
        delay_ms(10);
    }
    res = 0;
}
myfree(SRAMIN, picfileinfo); /* 释放内存 */
myfree(SRAMIN, pname); /* 释放内存 */
myfree(SRAMIN, picoffsettbl); /* 释放内存 */
}

```

可以看到整个设计思路是跟据图片解码库来设计的, `piclib_ai_load_picfile()` 是这套代码的核心, 其它的交互是围绕它和图片解码后的图片信息作的显示。大家再仔细对照光盘中的源码进一步了解整个设置思路。另外, 我们的程序中只分配了 4 个文件索引, 故更多数量的图片无法直接在本程序下演示, 大家根据自己的需要再进行修改即可。

52.4 下载验证

在代码编译成功之后, 我们下载代码到开发板上, 可以看到 LCD 开始显示图片 (假设 SD 卡及文件都准备好了, 即: 在 SD 卡根目录新建: PICTURE 文件夹, 并存放一些图片文件 (.bmp/.jpg/.gif) 在该文件夹内), 如图 52.4.1 所示:



图 52.4.1 图片显示实验显示效果

按 KEY0 和 KEY2 可以快速切换到下一张或上一张，KEY_UP 按键可以暂停自动播放，同时 DS1 亮，指示处于暂停状态，再按一次 KEY_UP 则继续播放。同时，由于我们的代码支持 gif 格式的图片显示（注意尺寸不能超过 LCD 屏幕尺寸），所以可以放一些 gif 图片到 PICTURE 文件夹，来看动画了。

本章，同样可以通过 USMART 来测试该实验，将 `piclib_ai_load_picfile` 函数加入 USMART 控制（方法前面已经讲了很多次了），就可以通过串口调用该函数，在屏幕上任何区域显示任何你想要显示的图片了！同时，可以发送：`runtime1`，来开启 USMART 的函数执行时间统计功能，从而获取解码一张图片所需时间，方便验证。

注意：本例程在支持 AC6 时，jpeg 解码库中的函数指针容易被优化，故如果使用 AC6 进行本实验时，建议单独对其进行优化设置，MDK 也支持对单一文件进行优化等级设置，操作方法如图 52.4.2 所示，我们设置 AC6 下不对这个解码库进行优化：

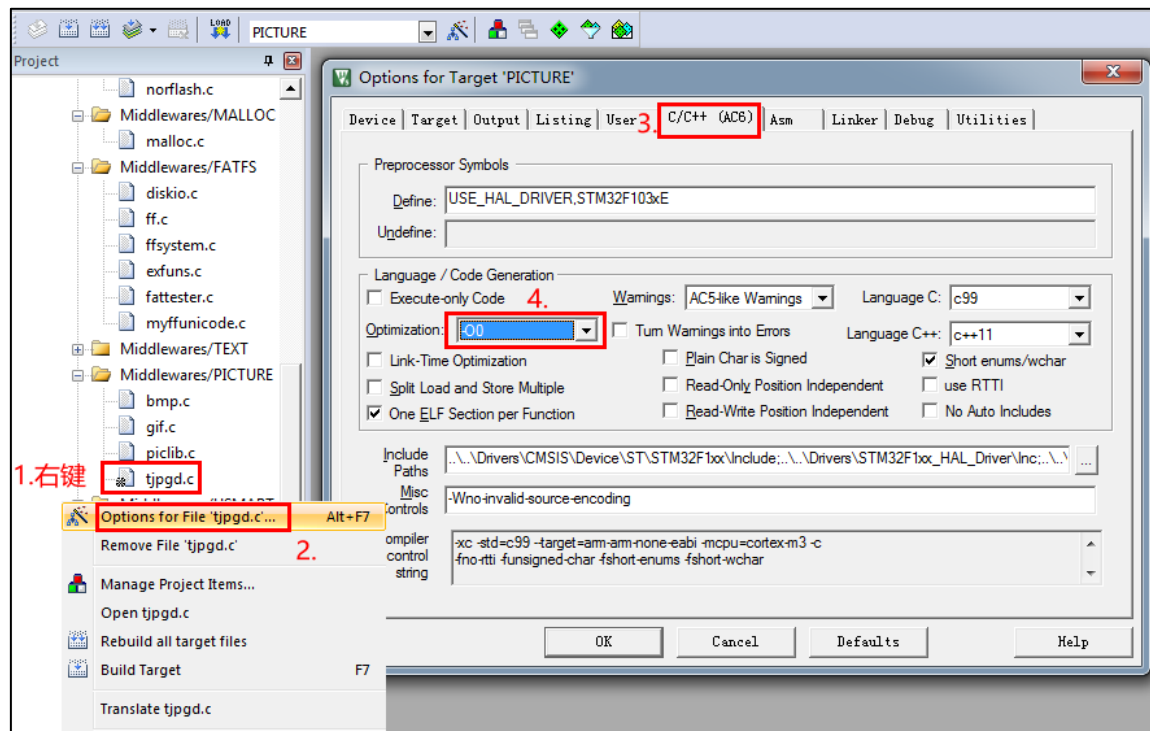


图 52.4.2 对 tjpgd.c 进行单独的优化设置

大家也可以想想其它解决优化的办法，图片显示实验我们就讲解到这里。

第五十三章 照相机实验

上一章，我们学习了图片解码，本章我们将学习 BMP 编码，结合前面的摄像头实验，实现一个简单的照相机功能。本章分为如下几个部分：

- 53.1 BMP 编码简介
- 53.2 硬件设计
- 53.3 软件设计
- 53.4 下载验证

53.1 BMP 编码简介

前面的章节中，我们学习了各种图片格式的解码。本章，我们介绍最简单的图片编码方法：BMP 图片编码。通过前面的了解，我们知道 BMP 文件是由文件头、位图信息头、颜色信息和图形数据等四部分组成。我们先来了解下这几个部分。

1、BMP 文件头（14 字节）：BMP 文件头数据结构含有 BMP 文件的类型、文件大小和位图起始位置等信息。

这里的 `__PACKED_STRUCT` 是强制对齐，这里是把结构体中间的留白空间移除。默认定义的变量是按 CUP 字长（STM32 为 32 位）对齐的，这样可以增加程序的访问速度，但这样定义一个结构体时如果结构体成员并不全部按 CUP 的字长去定义，如有 `uint8_t`, `uint16_t` 时，编译器默认按照的长度将占用 2 个 `uint32_t` 类型的长度，而对于嵌入式产品尤其是内存紧张的产品，这样定义的结构体变量就会浪费内存空间。嵌入式的编译器支持通过强制对齐，可以优化结构体变量的空间，大家同样可以在 MDK 的帮助文件中查找 `__packed` 关键字去看这部分的知识点。我们这里用了在 MDK 下同时兼容 AC5 和 AC6 编译器的写法。BMP 的文件头定义如下：

```
/* BMP 头文件 */
typedef __PACKED_STRUCT
{
    uint16_t  bfType;          /* 文件标志. 只对 'BM', 用来识别 BMP 位图类型 */
    uint32_t  bfSize;          /* 文件大小, 占四个字节 */
    uint16_t  bfReserved1;     /* 保留 */
    uint16_t  bfReserved2;     /* 保留 */
    uint32_t  bfOffBits;       /* 从文件开始到位图数据 (bitmap data) 开始之间的的偏移量 */
} BITMAPFILEHEADER;
```

2、位图信息头（40 字节）：BMP 位图信息头数据用于说明位图的尺寸等信息。

```
/* BMP 信息头 */
typedef __PACKED_STRUCT
{
    uint32_t  biSize;          /* 说明 BITMAPINFOHEADER 结构所需要的字数。 */
    long      biWidth;         /* 说明图象的宽度，以像素为单位 */
    long      biHeight;        /* 说明图象的高度，以像素为单位 */
    uint16_t  biPlanes;        /* 为目标设备说明位面数，其值将总是被设为 1 */
    uint16_t  biBitCount;      /* 说明比特数/像素，其值为 1、4、8、16、24、或 32 */
    uint32_t  biCompression;   /* 说明图象数据压缩的类型。其值可以是下述值之一
    * BI_RGB   : 没有压缩
    * BI_RLE8  : 每个像素 8bit 的 RLE 压缩编码，压缩格式由 2B 组成
    * BI_RLE4  : 每个像素 4bit 的 RLE 压缩编码，压缩格式由 2B 组成
    * BI_BITFIELDS: 每个像素的比特由指定的掩码决定
    */
    uint32_t  biSizeImage;     /* 说明图象大小，字节为单位。当用 BI_RGB 格式时，可设置为 0 */
    long      biXPelsPerMeter; /* 说明水平分辨率，用像素/米表示 */
    long      biYPelsPerMeter; /* 说明垂直分辨率，用像素/米表示 */
    uint32_t  biClrUsed;       /* 说明位图实际使用的彩色表中的颜色索引数 */
    /* 说明对图象显示有重要影响的颜色索引的数目，如果是 0，表示都重要 */
    uint32_t  biClrImportant;
```

```
}BITMAPINFOHEADER ;
```

3、颜色表：颜色表用于说明位图中的颜色，它有若干个表项，每一个表项是一个 RGBQUAD 类型的结构，定义一种颜色。

```
/* 彩色表 */
typedef __PACKED_STRUCT
{
    uint8_t rgbBlue;           /* 指定蓝色强度 */
    uint8_t rgbGreen;         /* 指定绿色强度 */
    uint8_t rgbRed;           /* 指定红色强度 */
    uint8_t rgbReserved;      /* 保留，设置为 0 */
}RGBQUAD ;
```

颜色表中 RGBQUAD 结构数据的个数由 biBitCount 来确定：当 biBitCount=1、4、8 时，分别有 2、16、256 个表项；当 biBitCount 大于 8 时，没有颜色表项。

BMP 文件头、位图信息头和颜色表组成位图信息（我们将 BMP 文件头也加进来，方便处理），BITMAPINFO 结构定义如下：

```
/* 位图信息头 */
typedef __PACKED_STRUCT
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    uint32_t RGB_MASK[3];      /* 调色板用于存放 RGB 掩码 */
    //RGBQUAD bmiColors[256];
}BITMAPINFO;
```

4、位图数据：位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右，扫描行之间是从下到上。位图的一个像素值所占的字节数：

- 当 biBitCount=1 时，8 个像素占 1 个字节；
- 当 biBitCount=4 时，2 个像素占 1 个字节；
- 当 biBitCount=8 时，1 个像素占 1 个字节；
- 当 biBitCount=16 时，1 个像素占 2 个字节；
- 当 biBitCount=24 时，1 个像素占 3 个字节；
- 当 biBitCount=32 时，1 个像素占 4 个字节；

biBitCount=1 表示位图最多有两种颜色，缺省情况下是黑色和白色，你也可以自己定义这两种颜色。图像信息头装调色板中将有二个调色板项，称为索引 0 和索引 1。图象数据阵列中的每一位表示一个像素。如果一个位是 0，显示时就使用索引 0 的 RGB 值，如果位是 1，则使用索引 1 的 RGB 值。

biBitCount=16 表示位图最多有 65536 种颜色。每个像素用 16 位（2 个字节）表示。这种格式叫作高彩色，或叫增强型 16 位色，或 64K 色。它的情况比较复杂，当 biCompression 成员的值是 BI_RGB 时，它没有调色板。16 位中，最低的 5 位表示蓝色分量，中间的 5 位表示绿色分量，高的 5 位表示红色分量，一共占用了 15 位，最高的一位保留，设为 0。这种格式也被称作 555 16 位位图。如果 biCompression 成员的值是 BI_BITFIELDS，那么情况就复杂了，首先是原来调色板的位置被三个 DWORD 变量占据，称为红、绿、蓝掩码。分别用于描述红、绿、蓝分量在 16 位中所占的位置。在 Windows 95（或 98）中，系统可接受两种格式的位域：555 和 565，在 555 格式下，红、绿、蓝的掩码分别是：0x7C00、0x03E0、0x001F，而在 565 格式下，它们则分别为：0xF800、0x07E0、0x001F。你在读取一个像素之后，可以分别用掩码“与”上像素值，从而提取出想要的颜色分量（当然还要再经过适当的左右移操作）。在 NT 系统中，则没有格式限制，只不过要求掩码之间不能有重叠。（注：这种格式的图像使用起来是比较麻烦的，不过因为它的显示效果接近于真彩，而图像数据又比真彩图像小的多，所以，它更多的被用于游戏软件）。

biBitCount=32 表示位图最多有 4294967296(2 的 32 次方)种颜色。这种位图的结构与 16 位位图结构非常类似，当 biCompression 成员的值是 BI_RGB 时，它也没有调色板，32 位中有 24 位用于存放 RGB 值，顺序是：最高位一保留，红 8 位、绿 8 位、蓝 8 位。这种格式也成为 888 32 位图。如果 biCompression 成员的值是 BI_BITFIELDS 时，原来调色板的位置将被三个

DWORD 变量占据，成为红、绿、蓝掩码，分别用于描述红、绿、蓝分量在 32 位中所占的位置。在 Windows 95(or 98)中，系统只接受 888 格式，也就是说三个掩码的值只能是：0xFF0000、0xFF00、0xFF。而在 NT 系统中，你只要注意使掩码之间不产生重叠就行。（注：这种图像格式比较规整，因为它是 DWORD 对齐的，所以在内存中进行图像处理时可进行汇编级的代码优化（简单））。

通过以上了解，我们对 BMP 有了一个比较深入的了解，本章，我们采用 16 位 BMP 编码（因为我们的 LCD 就是 16 位色的，而且 16 位 BMP 编码比 24 位 BMP 编码更省空间），故我们需要设置 biBitCount 的值为 16，这样得到新的位图信息（BITMAPINFO）结构体：

```
/* 位图信息头 */
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    uint32_t RGB_MASK[3]; /* 调色板用于存放 RGB 掩码 */
}BITMAPINFO;
```

其实就是颜色表由 3 个 RGB 掩码代替。最后，我们来看看将 LCD 的显存保存为 BMP 格式的图片文件的步骤：

1) 创建 BMP 位图信息，并初始化各个相关信息

这里，我们要设置 BMP 图片的分辨率为 LCD 分辨率、BMP 图片的大小（整个 BMP 文件大小）、BMP 的像素位数（16 位）和掩码等信息。

2) 创建新 BMP 文件，写入 BMP 位图信息

我们要保存 BMP，当然要存放在某个地方（文件），所以需要先创建文件，同时先保存 BMP 位图信息，之后才开始 BMP 数据的写入。

3) 保存位图数据。

这里就比较简单了，只需要从 LCD 的 GRAM 里面读取各点的颜色值，依次写入第二步创建的 BMP 文件即可。注意：保存顺序（即读 GRAM 顺序）是从左到右，从下到上。

4) 关闭文件。

使用 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！这个要特别注意，写完之后，一定要调用 f_close。

BMP 编码就介绍到这里。

53.2 硬件设计

1. 例程功能

开机的时候先检测字库，然后检测 SD 卡根目录是否存在 PHOTO 文件夹，如果不存在则创建，如果创建失败，则报错（提示拍照功能不可用）。在找到 SD 卡的 PHOTO 文件夹后，开始初始化 OV7725，在初始化成功之后，就一直在屏幕显示 OV7725 拍到的内容。当按下 KEY0 按键的时候，即进行拍照，此时 LED1 亮，拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功，同时 LED1 灭。LED0 还是用于指示程序运行状态。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 独立按键

KEY0 - PE4

3) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

5) micro SD 卡

6) NOR FLASH

7) 外部中断 8 (PA8，用于检测 OV7725 的帧信号)

- 8) 定时器 6 (用于打印摄像头帧率)
9) 正点原子 OV7725 摄像头模块, 连接关系为:

| OV7725 模块 | STM32 开发板 | OV7725 模块 | STM32 开发板 |
|-----------|-----------|-----------|-----------|
| OV_D0~D7 | PC0~7 | FIFO_OE | PG15 |
| OV_SCL | PD3 | FIFO_WRST | PD6 |
| OV_SDA | PG13 | FIFO_WEN | PB3 |
| OV_VSYNC | PA8 | FIFO_RCLK | PB4 |
| FIFO_RRST | PG14 | | |

53.3 程序设计

53.3.1 程序流程图

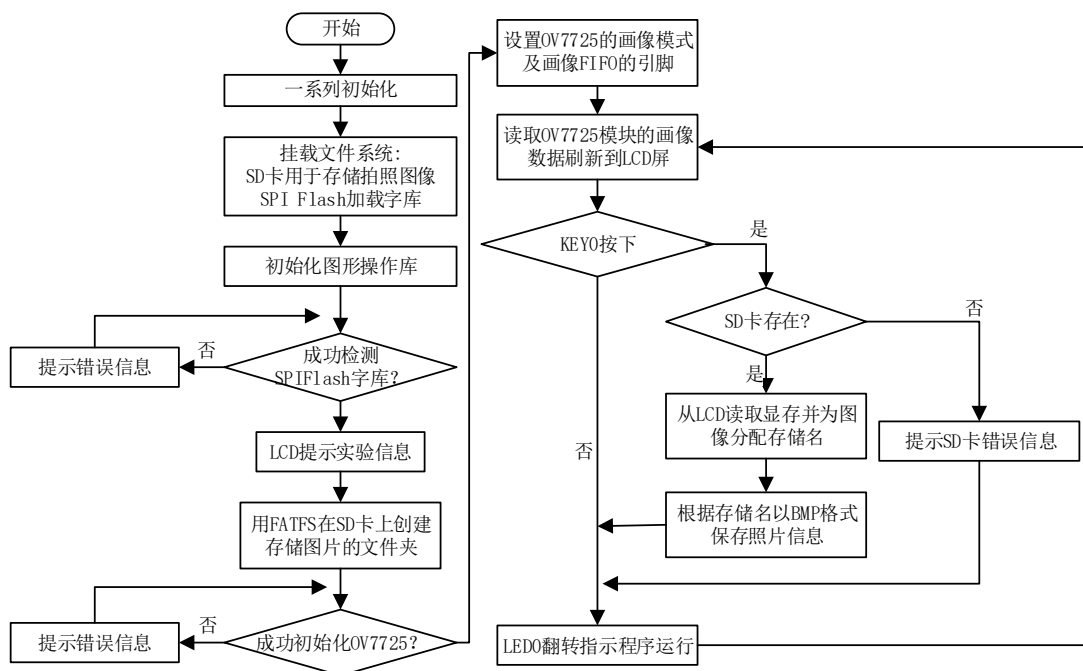


图 53.3.1.1 照相机实验程序流程图

53.3.2 程序解析

1. PICTURE 驱动代码

这里我们只讲解核心代码, 详细的源码请大家参考光盘本实验对应源码, PICTURE 的驱动主要包括两个文件: bmp.c 和 bmp.h。

bmp.h 头文件在 53.1.1 小节基本讲过, 具体请看源码。下面来看到 bmp.c 文件里面的 bmp 编码函数: bmp_encode, 该函数代码如下:

```
/**
 * @brief    BMP 编码函数
 * @note     将当前 LCD 屏幕的指定区域截图, 存为 16 位格式的 BMP 文件 RGB565 格式.
 *           保存为 rgb565 则需要掩码, 需要利用原来调色板位置增加掩码. 这里我们已经增加了掩码.
 *           保存为 rgb555 格式则需要颜色转换, 耗时间比较久, 所以保存为 565 是最快速的办法.
 *
 * @param    filename    : 包含存储路径的文件名 (.bmp)
 * @param    x, y        : 起始坐标
 * @param    width,height: 区域大小
 * @param    acolor      : 附加的 alphablend 的颜色 (这个仅对 32 位色 bmp 有效!!!)
 * @param    mode        : 保存模式
```



```

* @arg          0, 仅仅创建新文件的方式编码;
* @arg          1, 如果之前存在文件,则覆盖之前的文件.如果没有,则创建新的文件;
* @retval      操作结果
* @arg          0 , 成功
* @arg          其他, 错误码
*/
uint8_t bmp_encode(uint8_t *filename, uint16_t x, uint16_t y, uint16_t width,
uint16_t height, uint8_t mode)
{
    FIL *f_bmp;
    uint32_t bw = 0;
    uint16_t bmpheadsize; /* bmp 头大小 */
    BITMAPINFO hbmp; /* bmp 头 */
    uint8_t res = 0;
    uint16_t tx, ty; /* 图像尺寸 */
    uint16_t *databuf; /* 数据缓存区地址 */
    uint16_t pixcnt; /* 像素计数器 */
    uint16_t bi4width; /* 水平像素字节数 */

    if (width == 0 || height == 0) return PIC_WINDOW_ERR; /* 区域错误 */
    if ((x + width - 1) > lcddev.width) return PIC_WINDOW_ERR; /* 区域错误 */
    if ((y + height - 1) > lcddev.height) return PIC_WINDOW_ERR; /* 区域错误 */

    #if BMP_USE_MALLOC == 1 /* 使用 malloc */

    /* 开辟至少 bi4width 大小的字节的内存区域,对 240 宽的屏,480 个字节就够了.
    最大支持 1024 宽度的 bmp 编码 */
    databuf = (uint16_t *)piclib_mem_malloc(2048);
    if (databuf == NULL) return PIC_MEM_ERR; /* 内存申请失败. */
    f_bmp = (FIL *)piclib_mem_malloc(sizeof(FIL)); /* 开辟 FIL 字节的内存区域 */
    if (f_bmp == NULL) /* 内存申请失败 */
    {
        piclib_mem_free(databuf);
        return PIC_MEM_ERR;
    }
    #else
    databuf = (uint16_t *)bmpreadbuf;
    f_bmp = &f_bfile;
    #endif

    bmpheadsize = sizeof(hbmp); /* 得到 bmp 文件头的大小 */
    my_mem_set((uint8_t *)&hbmp, 0, sizeof(hbmp)); /* 置零空申请到的内存 */
    hbmp.bmiHeader.biSize = sizeof(BITMAPINFOHEADER); /* 信息头大小 */
    hbmp.bmiHeader.biWidth = width; /* bmp 的宽度 */
    hbmp.bmiHeader.biHeight = height; /* bmp 的高度 */
    hbmp.bmiHeader.biPlanes = 1; /* 恒为 1 */
    hbmp.bmiHeader.biBitCount = 16; /* bmp 为 16 位色 bmp */
    hbmp.bmiHeader.biCompression = BI_BITFIELDS; /* 每个像素的比特由指定的掩码决定 */
    hbmp.bmiHeader.biSizeImage = hbmp.bmiHeader.biHeight *
        hbmp.bmiHeader.biWidth * hbmp.bmiHeader.biBitCount/8; /* bmp 数据区大小 */
    hbmp.bmfHeader.bfType = ((uint16_t)'M' << 8) + 'B'; /* BM 格式标志 */
    /* 整个 bmp 的大小 */
    hbmp.bmfHeader.bfSize = bmpheadsize + hbmp.bmiHeader.biSizeImage;
    hbmp.bmfHeader.bfOffBits = bmpheadsize; /* 到数据区的偏移 */
    hbmp.RGB_MASK[0] = 0X00F800; /* 红色掩码 */
    hbmp.RGB_MASK[1] = 0X0007E0; /* 绿色掩码 */
    hbmp.RGB_MASK[2] = 0X00001F; /* 蓝色掩码 */

    if (mode == 1)
    {
        /* 尝试打开之前的文件 */
        res = f_open(f_bmp, (const TCHAR *)filename, FA_READ | FA_WRITE);
    }
}

```

```

if (mode == 0 || res == 0x04)
{
    /* 模式 0,或者尝试打开失败,则创建新文件 */
    res = f_open(f_bmp, (const TCHAR *)filename, FA_WRITE | FA_CREATE_NEW);
}

if ((hbm.bmiHeader.biWidth * 2) % 4) /* 水平像素(字节)不为 4 的倍数 */
{
    /* 实际要写入的宽度像素,必须为 4 的倍数 */
    bi4width = ((hbm.bmiHeader.biWidth * 2) / 4 + 1) * 4;
}
else
{
    bi4width = hbm.bmiHeader.biWidth * 2; /* 刚好为 4 的倍数 */
}

if (res == FR_OK) /* 创建成功 */
{
    res = f_write(f_bmp, (uint8_t *)&hbm, bmpheadsize, &bw); /* 写入 BMP 首部 */
    for (ty = y + height - 1; hbm.bmiHeader.biHeight; ty--)
    {
        pixcnt = 0;
        for (tx = x; pixcnt != (bi4width / 2);)
        {
            if (pixcnt < hbm.bmiHeader.biWidth)
            {
                databuf[pixcnt] = pic_phy.read_point(tx, ty); /* 读取坐标点的值 */
            }
            else
            {
                databuf[pixcnt] = 0xffff; /* 补充白色的像素 */
            }
            pixcnt++;
            tx++;
        }
        hbm.bmiHeader.biHeight--;
        res = f_write(f_bmp, (uint8_t *)databuf, bi4width, &bw); /* 写入数据 */
        f_close(f_bmp);
    }
}

#if BMP_USE_MALLOC == 1 /* 使用 malloc */
piclib_mem_free(databuf);
piclib_mem_free(f_bmp);
#endif
return res;
}

```

该函数实现了对 LCD 屏幕的任意指定区域进行截屏保存,用到的方法就是 53.1.1 节我们所介绍的方法,该函数实现了将 LCD 任意指定区域的内容,保存个为 16 位 BMP 格式,存放在指定位置(由 filename 决定)。注意,代码中的 BMP_USE_MALLOC 是在 bmp.h 定义的一个宏,用于设置是否使用 malloc,本章我们选择使用 malloc。

2. main.c 代码

main.c 函数我们在之前摄像头实验的基本上进行改动,首先我们要为图片分配一个与图片文件夹下名字不重复的文件名,主要通过 camera_new_pathname 实现,函数代码如下:

```

/**
 * @brief      文件名自增(避免覆盖)
 * @note      组合成形如 "0:PHOTO/PIC13141.bmp" 的文件名
 * @param      pname : 有效的文件名
 * @retval     无
 */
void camera_new_pathname(char *pname)
{

```

```
uint8_t res;
uint16_t index = 0;
FIL *ftemp;
ftemp = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 开辟 FIL 字节的内存区域 */
if (ftemp == NULL) return; /* 内存申请失败 */
while (index < 0xFFFF)
{
    sprintf((char *)pname, "0:PHOTO/PIC%05d.bmp", index);
    res = f_open(ftemp, (const TCHAR *)pname, FA_READ); /* 尝试打开这个文件 */
    if (res == FR_NO_FILE) break; /* 该文件名不存在，正是我们需要的 */
    index++;
}
myfree(SRAMIN, ftemp);
}
```

通过以上程序，可以生成一个与当前文件夹下图片不重名的文件名字符串，并传给针对应的缓冲区。

为了模拟照相机的效果，我们需要把 LCD 上显示的画像读取出来并用前面的 `bmp_encode()` 函数编码成*.bmp 格式的图片进行存储，以模拟实时的拍照效果。

最后，我们来看一下 main 函数，代码如下：

```
extern uint8_t g_ov7725_vsta; /* 在 exit.c 里面定义 */
extern uint8_t g_ov7725_frame; /* 在 timer.c 里面定义 */
int main(void)
{
    uint8_t res;
    char *pname; /* 带路径的文件名 */
    uint8_t key; /* 键值 */
    uint8_t i;
    uint8_t sd_ok = 1; /* 0, sd 卡不正常; 1, SD 卡正常 */
    uint8_t vga_mode = 0; /* 0, QVGA 模式(320 * 240); 1, VGA 模式(640 * 480) */

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    beep_init(); /* 蜂鸣器初始化 */
    norflash_init(); /* 初始化 NORFLASH */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */
    exfuncs_init(); /* 为 fatfs 相关变量申请内存 */
    f_mount(fs[0], "0:", 1); /* 挂载 SD 卡 */
    f_mount(fs[1], "1:", 1); /* 挂载 FLASH */

    piclib_init(); /* 初始化画图 */
    while (fonts_init()) /* 检查字库 */
    {
        lcd_show_string(30, 50, 200, 16, 16, "Font Error!", RED);
        delay_ms(200);
        lcd_fill(30, 50, 240, 66, WHITE); /* 清除显示 */
        delay_ms(200);
    }
    text_show_string(30, 50, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
    text_show_string(30, 70, 200, 16, "照相机 实验", 16, 0, RED);
    text_show_string(30, 90, 200, 16, "KEY0:拍照 (bmp 格式)", 16, 0, RED);
    res = f_mkdir("0:/PHOTO"); /* 创建 PHOTO 文件夹 */
    if (res != FR_EXIST && res != FR_OK) /* 发生了错误 */

```

```

{
    res = f_mkdir("0:/PHOTO");          /* 创建 PHOTO 文件夹 */
    text_show_string(30, 110, 240, 16, "SD 卡错误!", 16, 0, RED);
    delay_ms(200);
    text_show_string(30, 110, 240, 16, "拍照功能将不可用!", 16, 0, RED);
    delay_ms(200);
    sd_ok = 0;
}

while (ov7725_init() != 0)              /* 初始化 ov7725 失败? */
{
    lcd_show_string(30, 130, 200, 16, 16, "OV7725 Error!!", RED);
    delay_ms(200);
    lcd_fill(30, 150, 239, 246, WHITE);
    delay_ms(200);
}
lcd_show_string(30, 130, 200, 16, 16, "OV7725 Init OK", RED);
delay_ms(1500);

/* 输出窗口大小设置 QVGA / VGA 模式 */
g_ov7725_wwidth = 320;                  /* 默认窗口宽度为 320 */
g_ov7725_wheight = 240;                /* 默认窗口高度为 240 */
ov7725_window_set(g_ov7725_wwidth, g_ov7725_wheight, vga_mode);
ov7725_light_mode(0);                  /* 自动 灯光模式 */
ov7725_color_saturation(4);            /* 默认 色彩饱和度 */
ov7725_brightness(4);                 /* 默认 亮度 */
ov7725_contrast(4);                   /* 默认 对比度 */
ov7725_special_effects(0);            /* 默认 特效 */
OV7725_OE(0);                         /* 使能 OV7725 FIFO 数据输出 */
pname = mymalloc(SRAMIN, 30);          /* 为带路径的文件名分配 30 个字节的内存 */
btim_tmx_int_init(10000, 7200 - 1);    /* 10Khz 计数频率, 1 秒钟中断 */
exti_ov7725_vsync_init();             /* 使能 OV7725 VSYNC 外部中断, 捕获帧中断 */
lcd_clear(BLACK);
while (1)
{
    key = key_scan(0);
    if (key == KEY0_PRES)
    {
        if (sd_ok)
        {
            LED1(0);                   /* 点亮 DS1, 提示正在拍照 */
            camera_new_pathname(pname); /* 得到文件名 */
            /* 编码成 bmp 图片 */
            if (bmp_encode((uint8_t *)pname,
                (lcddev.width - g_ov7725_wheight) / 2,
                (lcddev.height - g_ov7725_wwidth) / 2,
                g_ov7725_wheight, g_ov7725_wwidth, 0))
            {
                text_show_string(40, 110, 240, 12, "写入文件错误!", 12, 0, RED);
            }
            else
            {
                text_show_string(40, 110, 240, 12, "拍照成功!", 12, 0, BLUE);
                text_show_string(40, 130, 240, 12, "保存为:", 12, 0, BLUE);
                text_show_string(40 + 42, 130, 240, 12, pname, 12, 0, BLUE);
                BEEP(1);                /* 蜂鸣器短叫, 提示拍照完成 */
                delay_ms(100);
            }
        }
        else /* 提示 SD 卡错误 */
        {
            text_show_string(40, 110, 240, 12, "SD 卡错误!", 12, 0, RED);
        }
    }
}

```

```

        text_show_string(40, 130, 240, 12, "拍照功能不可用!", 12, 0, RED);
    }
    BEEP(0);          /* 关闭蜂鸣器 */
    LED1(1);          /* 关闭 DS1 */
    delay_ms(1800);    /* 等待 1.8 秒钟 */
    lcd_clear(BLACK);
}
else
{
    delay_ms(5);
}
ov7725_camera_refresh(); /* 更新显示 */
i++;
if (i >= 15)          /* DS0 闪烁 */
{
    i = 0;
    LED0_TOGGLE();     /* LED0 闪烁 */
}
}
}

```

到此，本实验的代码基本就编写完成了，最后进行下载验证。

53.4 下载验证

将程序下载到开发板后，可以看到 LCD 首先显示一些实验相关的信息，如图 53.4.1 所示：

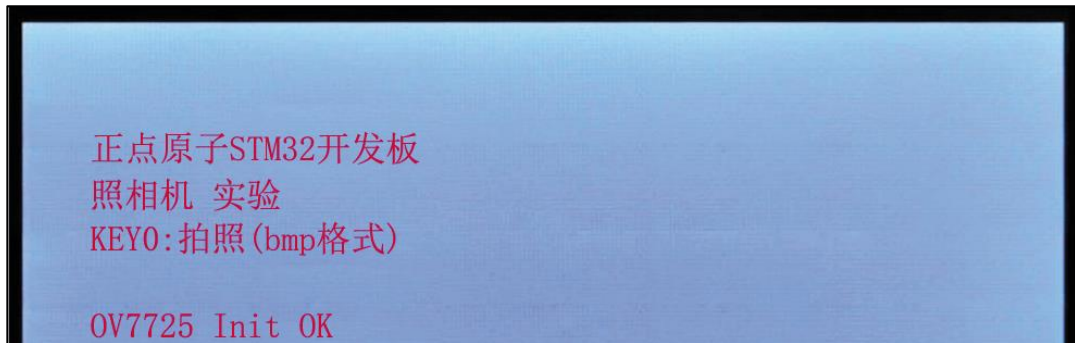


图 53.4.1 显示实验相关信息

随后，进入监控界面。此时，我们可以按下 KEY0 即可进行拍照。拍照得到的照片效果如图 53.4.2 所示：



图 53.4.2 拍照样图

最后，我们还可以通过 USART 调用 bmp_encode 函数，实现串口控制拍照，还可以拍成各种尺寸哦（不过必须小于 240*320）！

第五十四章 音乐播放器实验

正点原子战舰 STM32F103 板载了 VS1053B 这颗高性能音频编解码芯片，该芯片可以支持 wav/mp3/wma/flac/ogg/midi/aac 等音频格式的播放，并且支持录音（下一章介绍）。本章，我们将利用战舰 STM32F103 实现一个简单的音乐播放器（支持 wav/mp3/wma/flac/ogg/midi/aac 等格式）。本章分为如下几个部：

- 54.1 VS1053 简介
- 54.2 硬件设计
- 54.3 软件设计
- 54.4 下载验证

54.1 VS1053 简介

VS1053 是继 VS1003 后荷兰 VLSI 公司出品的又一款高性能解码芯片。该芯片可以实现对 MP3/OGG/WMA/FLAC/WAV/AAC/MIDI 等音频格式的解码，同时还可以支持 ADPCM/OGG 等格式的编码，性能相对以往的 VS1003 提升不少。VS1053 拥有一个高性能的 DSP 处理器核 VS_DSP，16K 的指令 RAM，0.5K 的数据 RAM，通过 SPI 控制，具有 8 个可用的通用 IO 口和一个串口，芯片内部还带了一个可变采样率的立体声 ADC（支持咪头/咪头+线路/2 线路）、一个高性能立体声 DAC 及音频耳机放大器。

VS1053 的特性如下：

- 支持众多音频格式解码，包括 OGG/MP3/WMA/WAV/FLAC（需要加载 patch）/MIDI/AAC 等。
- 对话筒输入或线路输入的音频信号进行 OGG（需要加载 patch）/IMA ADPCM 编码
- 高低音控制
- 带有 EarSpeaker 空间效果（用耳机虚拟现场空间效果）
- 单时钟操作 12..13MHz
- 内部 PLL 锁相环时钟倍频器
- 低功耗
- 内含高性能片上立体声 DAC，两声道间无相位差
- 过零交差侦测和平滑的音量调整
- 内含能驱动 30 欧负载的耳机驱动器
- 模拟，数字，I/O 单独供电
- 为用户代码和数据准备的 16KB 片上 RAM
- 可扩展外部 DAC 的 I2S 接口
- 用于控制和数据的串行接口（SPI）
- 可被用作微处理器的从机
- 特殊应用的 SPI Flash 引导
- 供调试用途的 UART 接口

VS1053 相对于它的前辈 VS1003，增加了编解码格式的支持（比如支持 OGG/FLAC，还支持 OGG 编码，VS1003 不支持）、增加了 GPIO 数量到 8 个（VS1003 只有 4 个）、增加了内部指令 RAM 容量到 16KiB（VS1003 只有 5.5KiB）、增加了 I2S 接口（VS1003 没有）、支持 EarSpeaker 空间效果（VS1003 不支持）等。同时 VS1053 的 DAC 相对于 VS1003 有不少提高，同样的歌曲，用 VS1053 播放，听起来比 1003 效果好很多。

VS1053 的封装引脚和 VS1003 完全兼容，所以如果你以前用的是 VS1003，则只需要把 VS1003 换成 VS1053，就可以实现硬件更新，电路板完全不用修改。不过需要注意的是 VS1003 的 CVDD 是 2.5V，而 VS1053 的 CVDD 是 1.8V，所以你还需要把稳压芯片也变一下，其他都可以照旧了。

VS1053 通过 SPI 接口来接受输入的音频数据流，它可以是一个系统的从机，也可以作为独立的主机。这里我们只把它当成从机使用。我们通过 SPI 口向 VS1053 不停的输入音频数据，

它就会自动帮我们解码了，然后从输出通道输出音乐，这时我们接上耳机就能听到所播放的乐曲了。

VS1053 的 SPI 支持两种模式：1，VS1002 有效模式（即新模式）。2，VS1001 兼容模式。这里我们仅介绍 VS1002 有效模式（此模式也是 VS1053 的默认模式）。表 48.1.2 是在新模式下 VS1053 的 SPI 信号线功能描述：

| SDI 引脚 | SCI 引脚 | 说明 |
|--------|--------|---|
| XDCS | XCS | 低有效的片选信号。高电平将强制串行接口结束当前操作，并进入备用模式，它将强行使串行输出进入高阻状态。如果 SM_SDISHARE 是 1，则 XDCS 引脚是不使用的，但这个信号是通过将 XCS 反向来获得的。 |
| SCK | | 串行时钟输入。此时钟也是用作内部寄存器接口的主时钟。SCK 电平在平时可以是脉冲状或平静的。不管在何种情况之下，只要在 XCS 信号变低之后，首个时钟上升沿将被定义为首个位。 |
| SI | | 串行输入。如果片选有效，SI 是在时钟 SCK 的上升沿上取样的。 |
| - | SO | 串行输出。在读取时，数据是逐位移动输出在时钟 SCK 的下降沿上。而在写入时，它是处于高阻态的。 |

表 54.1.2 VS1053 新模式下 SPI 口信号线功能

VS1053 的 SPI 数据传送，分为 SDI 和 SCI，分别用来传输数据/命令。SDI 和前面介绍的 SPI 协议一样的，不过 VS1053 的数据传输是通过 DREQ 控制的，主机在判断 DREQ 有效（高电平）之后，直接发送即可（一次可以发送 32 个字节）。

这里我们重点介绍一下 SCI。SCI 串行总线命令接口包含了一个指令字节、一个地址字节和一个 16 位的数据字。读写操作可以读写单个寄存器，在 SCK 的上升沿读出数据位，所以主机必须在下降沿刷新数据。SCI 的字节数据总是高位在前低位在后的。第一个字节指令字节，只有 2 个指令，也就是读和写，读为 0X03，写为 0X02。

一个典型的 SCI 读时序如图 54.1.2 所示：

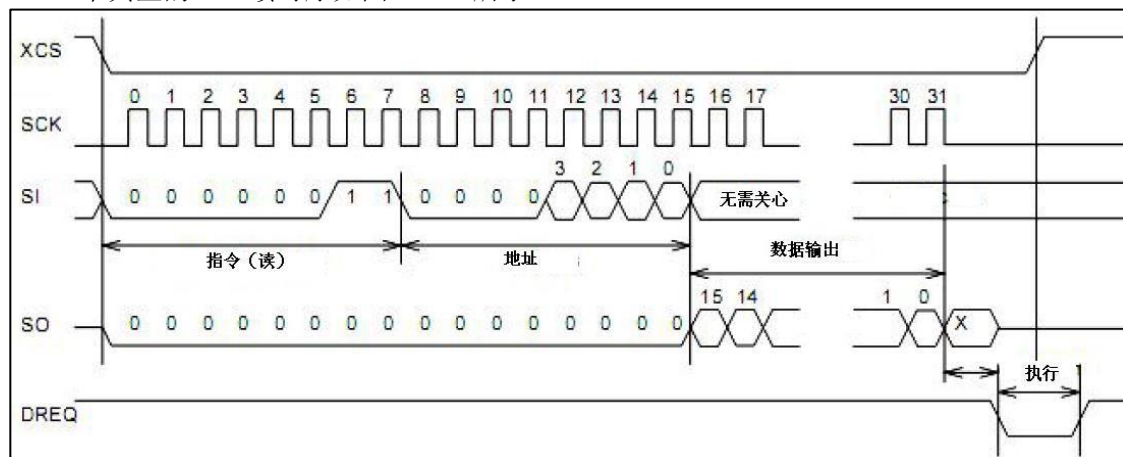


图 54.1.2 SCI 读时序

从图 54.1.2 可以看出，向 VS1053 读取数据，通过先拉低 XCS（VS_XCS），然后发送读指令（0X03），再发送一个地址，最后，我们在 SO 线（VS_MISO）上就可以读到输出的数据了。而同时 SI（VS_MOSI）上的数据将被忽略。

看完了 SCI 的读，我们再来看看 SCI 的写时序，如图 54.1.3 所示：

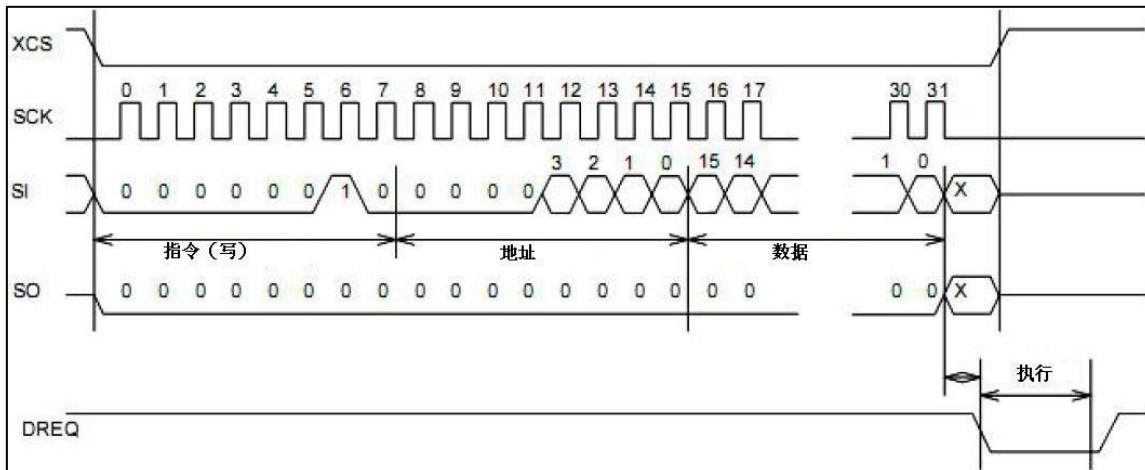


图 54.1.3 SCI 写时序

图 54.1.3 中，其时序和图 54.1.2 基本类似，都是先发指令，再发地址。不过写时序中，我们的指令是写指令（0X02），并且数据是通过 SI 写入 VS1053 的，SO 则一直维持低电平。细心的读者可能发现了，在这两个图中，DREQ 信号上都产生了一个短暂的低脉冲，也就是执行时间。这个不难理解，我们在写入和读出 VS1053 的数据之后，它需要一些时间来处理内部的事情，这段时间，是不允许外部打断的，所以，我们在 SCI 操作之前，最好判断一下 DREQ 是否为高电平，如果不是，则等待 DREQ 变为高。

了解了 VS1053 的 SPI 读写，我们再来看看 VS1053 的 SCI 寄存器，VS1053 的所有 SCI 寄存器如表 54.1.3 所示：

| SCI 寄存器 | | | | |
|---------|----|--------|-------------|------------|
| 寄存器 | 类型 | 复位值 | 缩写 | 描述 |
| 0X00 | RW | 0X0800 | MODE | 模式控制 |
| 0X01 | RW | 0X000C | STATUS | VS0153 状态 |
| 0X02 | RW | 0X0000 | BASS | 内置低音/高音控制 |
| 0X03 | RW | 0X0000 | CLOCKF | 时钟频率+倍频数 |
| 0X04 | RW | 0X0000 | DECODE_TIME | 解码时间长度（秒） |
| 0X05 | RW | 0X0000 | AUDATA | 各种音频数据 |
| 0X06 | RW | 0X0000 | WRAM | RAM 写/读 |
| 0X07 | RW | 0X0000 | WRAMADDR | RAM 写/读的基址 |
| 0X08 | R | 0X0000 | HDAT0 | 流的数据标头 0 |
| 0X09 | R | 0X0000 | HDAT1 | 流的数据标头 1 |
| 0X0A | RW | 0X0000 | AIADDR | 应用程序起始地址 |
| 0X0B | RW | 0X0000 | VOL | 音量控制 |
| 0X0C | RW | 0X0000 | AICTRL0 | 应用控制寄存器 0 |
| 0X0D | RW | 0X0000 | AICTRL1 | 应用控制寄存器 1 |
| 0X0E | RW | 0X0000 | AICTRL2 | 应用控制寄存器 2 |

表 54.1.3 SCI 寄存器

VS1053 总共有 16 个 SCI 寄存器，这里我们不介绍全部的寄存器，仅仅介绍几个我们在本章需要用到的寄存器。

首先是 MODE 寄存器，该寄存器用于控制 VS1053 的操作，是最关键的寄存器之一，该寄存器的复位值为 0x0800，其实就是默认设置为新模式。表 54.1.4 是 MODE 寄存器的各位描述：

| 位 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|------------------------|----------------------|-----------------|---------------------|------------------|----------------|---------------------|------------------------------|
| 名称 | SM_DIFF | SM_LAYER12 | SM_RESET | SM_CANCEL | SM_EARSPEAKER_LO | SM_TEST | SM_STREAM | SM_EARSPEAKER_HI |
| 功能 | 差分 | 允许MPEG I&II | 软件复位 | 取消当前文件的解码 | EarSpeaker 低设定 | 允许SDI 测试 | 流模式 | EarSpeaker 高设定 |
| 描述 | 0, 正常的同相音频 1, 左通道反相 | 0, 不允许 1, 允许 | 0, 不复位 1, 复位 | 0, 不取消 1, 取消 | 0, 关闭 1, 激活 | 0, 禁止 1, 允许 | 0, 不是 1, 是 | 0, 关闭 1, 激活 |
| 位 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 名称 | SM_DACT | SM_SDIORD | SM_SDISHARE | SM_SDINEW | SM_ADPCM | - | SM_LINE1 | SM_CLK_RANGE |
| 功能 | DCLK的有效边沿 | SDI位顺序 | 共享SPI片选 | VS1002本地SPI模式 | ADPCM激活 | - | 咪/线路1选择 | 输入时钟范围 |
| 描述 | 0, 上升沿 1, 下降沿 | 0, MSB在前 1, MSB在后 | 0, 不共享 1, 共享 | 0, 非本地模式 1, 本地模式 | 0, 不激活 1, 激活 | - | 0, MICP 1, LINE1 | 0, 12..13Mhz 1, 24..26Mhz |

表 54.1.4 MODE 寄存器各位描述

这个寄存器，我们这里只介绍一下第 2 和第 11 位，也就是 SM_RESET 和 SM_SDINEW。其他位，我们用默认的即可。这里 SM_RESET，可以提供一次软复位，建议在每播放一首歌曲之后，软复位一次。SM_SDINEW 为模式设置位，这里我们选择的是 VS1002 新模式(本地模式)，所以设置该位为 1（默认的设置）。其他位的详细介绍，请参考 VS1053 的数据手册。

接着我们看看 SCI_BASS 寄存器，该寄存器可以用于设置 VS1053 的高低音效。该寄存器的各位描述如表 54.1.5 所示：

| 名称 | 位域 | 说明 |
|--------------|-------|----------------------------|
| ST_AMPLITUDE | 15:12 | 高音控制,步长为 1.5dB(-8..7,0=关闭) |
| ST_FREQLIMIT | 11:08 | 下限频率,步长为 1000hz(1.15) |
| SB_AMPLITUDE | 7:04 | 低音增强,步长为 1dB(0..15,0=关闭) |
| SB_FREQLIMIT | 3:00 | 频率上限,步长为 10hz(2.15) |

表 54.1.5 SCI_BASS 寄存器各位描述

通过这个寄存器以上位的一些设置，我们可以随意配置自己喜欢的音效（其实就是高低音的调节）。VS1053 的 EarSpeaker 效果则由 MODE 寄存器控制，请参考表 54.1.4。

接下来，我们介绍一下 CLOCKF 寄存器，这个寄存器用来设置时钟频率、倍频等相关信息，该寄存器的各位描述如表 54.1.6 所示：

| CLOCKF 寄存器 | | | |
|------------|-------------------------------------|--------|---|
| 位 | 15:13 | 12:11 | 10:00 |
| 名称 | SC_MULT | SC_ADD | SC_FREQ |
| 描述 | 时钟倍频数 | 允许倍速频 | 时钟频率 |
| 说明 | $CLKI = XTALI * (Scmult * 0.5 + 1)$ | | 倍频增量 $= SC_ADD * 0.5$ 当时钟频率不为 12.288M 时，外部时钟为 12.288 时，此部分设置为 0 即可 |

表 54.1.6 CLOCKF 寄存器各位描述

此寄存器，重点说明 SC_FREQ，SC_FREQ 是以 4Khz 为步进的一个时钟寄存器，当外部时钟不是 12.288M 的时候，其计算公式为：

$$SC_FREQ = (XTALI - 8000000) / 4000$$

式中为 XTALI 的单位为 Hz。表 54.1.6 中 CLKI 是内部时钟频率，XTALI 是外部晶振的时钟频率。由于我们使用的是 12.288M 的晶振，在这里设置此寄存器的值为 0x9800，也就是设置内部时钟频率为输入时钟频率的 3 倍，倍频增量为 1.0 倍。

接下来，我们看看 DECODE_TIME 这个寄存器。该寄存器是一个存放解码时间的寄存器，以秒钟为单位，我们通过读取该寄存器的值，就可以得到解码时间了。不过它是一个累计时间，所以我们需要在每首歌播放之前把它清空一下，以得到这首歌的准确解码时间。

HDATA0 和 HDATA1 是两个数据流头寄存器，不同的音频文件，读出来的值意义不一样，我

们可以通过这两个寄存器来获取音频文件的码率，从而可以计算音频文件的总长度。这两个寄存器的详细介绍，请参考 VS1053 的数据手册。

最后我们介绍一下 VOL 这个寄存器，该寄存器用于控制 VS1053 的输出音量，该寄存器可以分别控制左右声道的音量，每个声道的控制范围为 0~254，每个增量代表 0.5db 的衰减，所以该值越小，代表音量越大。比如设置为 0X0000 则音量最大，而设置为 0XFEFE 则音量最小。注意：如果设置 VOL 的值为 0XFFFF，将使芯片进入掉电模式！

关于 VS1053 的介绍，我们就介绍到这里，更详细的介绍请看 VS1053 的数据手册。接下来我们说说如何控制 VS1053 播放一首歌曲。最简单的步骤如下：

- 复位 VS1053

这里包括了硬复位和软复位，是为了让 VS1053 的状态回到原始状态，准备解码下一首歌曲。这里建议大家在每首歌曲播放之前都执行一次硬件复位和软件复位，以便更好的播放音乐。

- 配置 VS1053 的相关寄存器

这里我们配置的寄存器包括 VS1053 的模式寄存器 (MODE)、时钟寄存器 (CLOCKF)、音调寄存器 (BASS)、音量寄存器 (VOL) 等。

- 发送音频数据

当经过以上两步配置以后，我们剩下来要做的事情，就是往 VS1053 里面扔音频数据了，只要是 VS1053 支持的音频格式，直接往里面丢就可以了，VS1053 会自动识别，并进行播放。不过发送数据要在 DREQ 信号的控制下有序的进行，不能乱发。这个规则很简单：只要 DREQ 变高，就向 VS1053 发送 32 个字节。然后继续等待 DREQ 变高，直到音频数据发送完。

经过以上三步，我们就可以播放音乐了。VS1053 就先介绍到这里。

54.2 硬件设计

1. 例程功能

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，则开始循环播放 SD 卡 MUSIC 文件夹里面的歌曲（必须在 SD 卡根目录建立一个 MUSIC 文件夹，并存放歌曲在里面），在 TFTLCD 上显示歌曲名字、播放时间、歌曲总时间、歌曲总数目、当前歌曲的编号等信息。KEY0 用于选择下一曲，KEY2 用于选择上一曲，KEY_UP 和 KEY1 用来调节音量。DS0 还是用于指示程序运行状态，DS1 用于指示 VS1053 正在初始化。

2. 硬件资源

本实验，大家需要准备 1 个 microSD/SD 卡（在里面新建一个 MUSIC 文件夹，并存放一些歌曲在 MUSIC 文件夹下）和一个耳机（非必备），分别插入 SD 卡接口和耳机接口，然后下载本实验就可以通过耳机来听歌了。实验用到的硬件资源如下：

- 1) LED 灯

LED0 - PB5

LED1 - PE5

- 2) 独立按键

KEY0 - PE4

KEY1 - PE3

KEY2 - PE2

KEY_UP - PA0 (程序中的宏名:WK_UP)

- 3) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

- 5) SD 卡:

通过 SDIO(SDIO_D0~D4(PC8~PC11),SDIO_SCK(PC12),SDIO_CMD(PD2))连接

- 6) NOR FLASH(SPI FLASH 芯片,连接在 SPI2 上)

- 7) VS1053 芯片，通过 SPI1 驱动

- 8) 功放芯片 HT6872,用于放大 1053 的输出以支持扬声器

正点原子战舰 STM32 开发板，自带了一颗 VS1053 音频编解码芯片，所以，我们直接可以

通过开发板来播放各种音频格式，实现一个音乐播放器。战舰 STM32 开发板板载了 VS1053 解码芯片的驱动电路，原理图如图 54.1.1 所示：

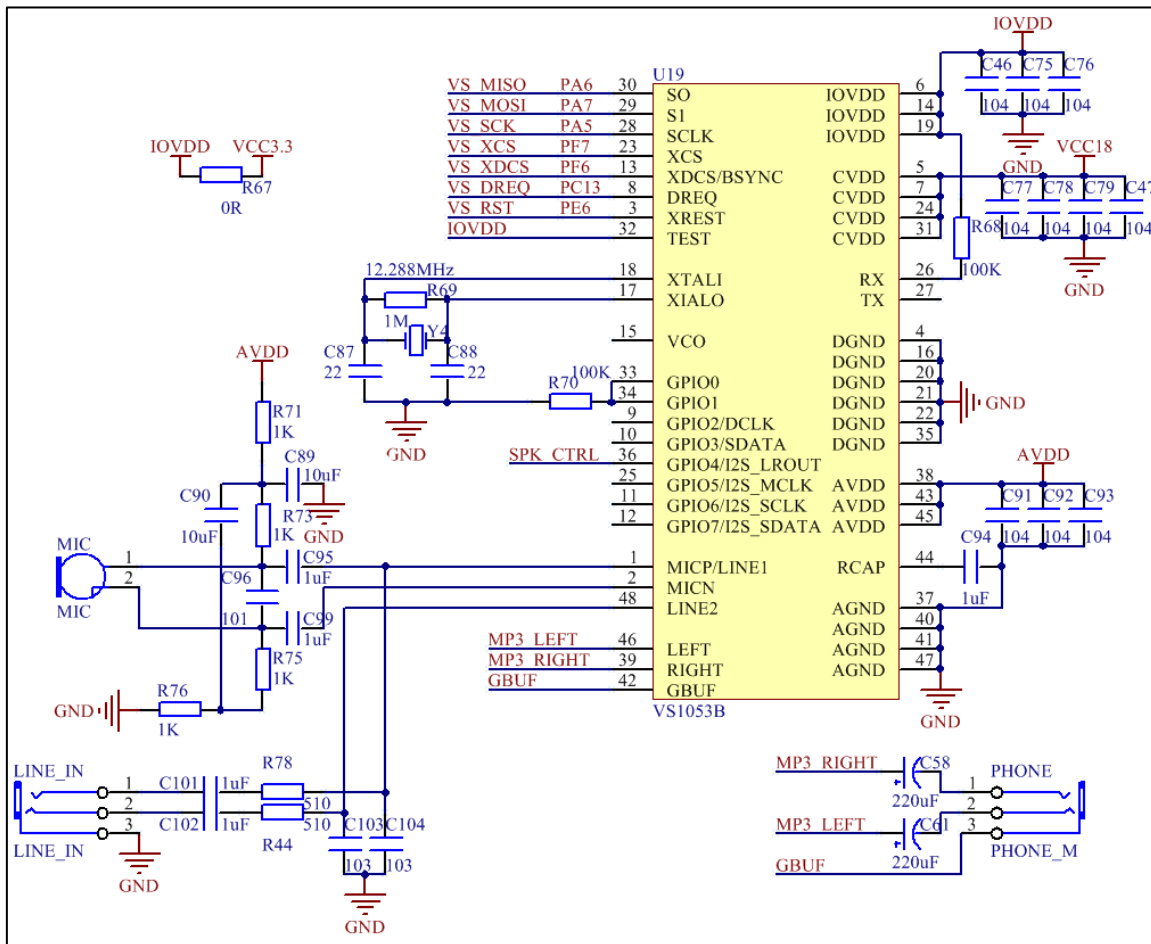


图 54.1.1 正点原子音频解码模块原理图

VS1053 通过 7 根线同 STM32 连接，他们是：VS_MISO、VS_MOSI、VS_SCK、VS_XCS、VS_XDCS、VS_DREQ 和 VS_RST。这 7 根线同 STM32 的连接关系如表 54.1.1 所示：

| 芯片 | 信号线 | | | | | | |
|---------------|---------|---------|--------|--------|---------|---------|--------|
| VS1053 | VS_MISO | VS_MOSI | VS_SCK | VS_XCS | VS_XDCS | VS_DREQ | VS_RST |
| STM32F103ZET6 | PA6 | PA7 | PA5 | PF7 | PF6 | PC13 | PE6 |

表 54.1.1 VS1053 各信号线与 STM32 连接关系

其中 VS_RST 是 VS1053 的复位信号线，低电平有效。VS_DREQ 是一个数据请求信号，用来通知主机，VS1053 可以接收数据与否。VS_MISO、VS_MOSI 和 VS_SCK 则是 VS1053 的 SPI 接口他们在 VS_XCS 和 VS_XDCS 下面来执行不同的操作。从上表可以看出，VS1053 的 SPI 是接在 STM32 的 SPI1 上面的。

MP3_RIGHT 和 MP3_LEFT 是来自 VS1053 的音频左右声道输出，由于我们用的 1053B 可以直接驱动一个 30 欧的负载，所以我们直接输出到耳机 PHONE 接口即可。PWM_AUDIO，则连接多功能接口的 AIN，可用于外接音频输入或者 PWM 音频，注意 PWM_AUDIO 仅仅输入了 TDA1308 的一个声道，所以插上耳机的时候，听到也是只有一边有声音。SPK_IN 则连接到了 HT6872，用于 HT6872 的输入。

HT6872 是一颗单声道、高功率（最大可达 4.7W）D 类功放 IC，驱动板载的 2W 喇叭（在板子背面），HT6872 原理图如图 48.2.2 所示：

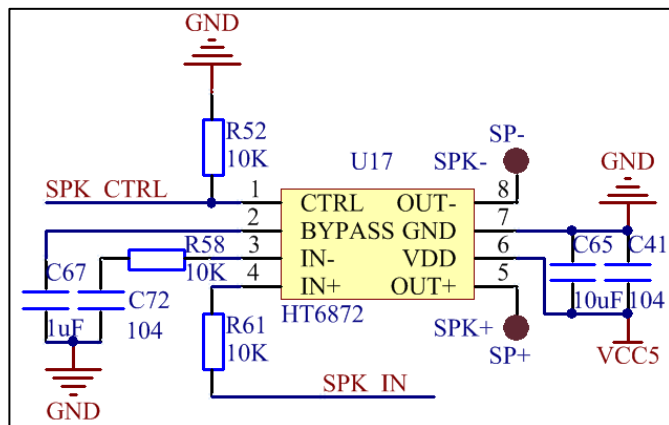


图 54.2.2 HT6872 原理图

图中 SPK_IN 就是 HT6872 的音频输入，来自图 48.2.1，然后 SP+和 SP-则分别连接喇叭的正负极。重点看看 SPK_CTRL，这个信号控制着 HT6872 的工作模式，该信号由 VS1053 的 36 脚 (GPIO4) 控制 (见图 48.1.1)，当 SPK_CTRL 脚为低电平时 HT6872 进入关断模式，也就是功放不工作了，当 SPK_CTRL 脚为高电平的时候，HT6872 进入正常工作模式，此时喇叭可以播放 SPK_IN 输入的音频信号。这样，我们通过 SPK_CTRL 就可以控制喇叭的开关了。

关于如何控制 VS1053 的 GPIO，详见 VS1053 中文数据手册的低 10.7 节 (67 页)，这里我们就不作介绍了。本例程播放歌曲的时候，喇叭输出是默认开启的，方便大家测试。

54.3 程序设计

54.3.2 程序流程图

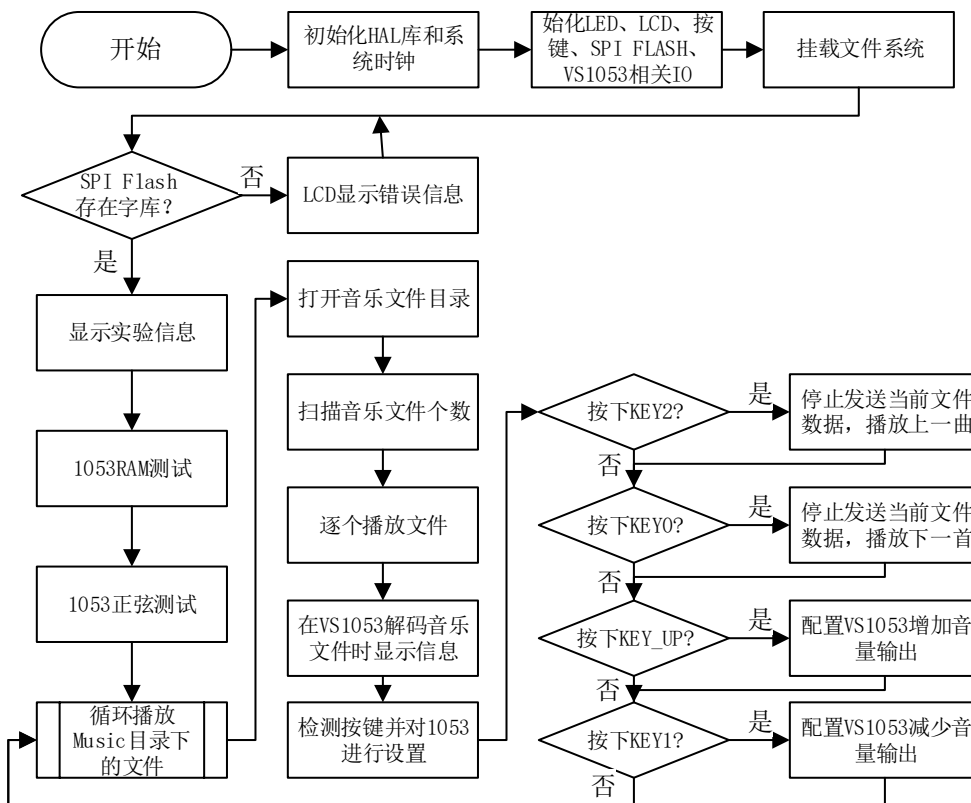


图 54.3.1.1 音乐播放器实验程序流程图

音乐播放我们从 SD 卡的指定目录读取音乐文件,解析格式正确后,通过 SPI 不断向 VS1053 发送文件数据至播放完成,VS1053 解码后通过选择扬声器或直接从耳机输出音乐。为了交互性,我们设置板载的按钮用于控制播放的歌曲切换及音量调节。

54.3.2 程序解析

音乐文件我们要通过 micro SD/SD 卡来传给单片机,那我们自然要用到文件系统。LCD、按键交互这些我们也需要实现,同样我们为了快速建立工程,复制之前的《实验 38 FATFS 实验》来修改成我们音乐播放器实验。

由于播放功能涉及到多个外设的配合使用,用文件系统读音频文件,做播放控制等,所以我们将 1053 的硬件驱动放到 BSP 目录下,播放功能作为 APP 放到 USER 目录下。

1. VS1053 驱动代码

这里我们只讲解核心代码,详细的源码请大家参考光盘本实验对应源码,VS1053 的驱动主要包括两个文件:vs1053.c 和 vs1053.h。

除去 SPI 的管脚,我们需要初始其它 IO 的模式,我们在头文件中定义 VS1053 的引脚,方便如果 IO 变更之后作修改:

```
/* VS10XX_RST/XCS/XDCS/DQ 引脚 定义 */
/* RST 口配置 */
#define VS10XX_RST_GPIO_PORT      GPIOE
#define VS10XX_RST_GPIO_PIN        GPIO_PIN_6
#define VS10XX_RST_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE();}while(0)
/* XCS 口配置 */
#define VS10XX_XCS_GPIO_PORT      GPIOF
#define VS10XX_XCS_GPIO_PIN        GPIO_PIN_7
#define VS10XX_XCS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOF_CLK_ENABLE();}while(0)
/* XDCS 口配置 */
#define VS10XX_XDCS_GPIO_PORT      GPIOF
#define VS10XX_XDCS_GPIO_PIN        GPIO_PIN_6
#define VS10XX_XDCS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOF_CLK_ENABLE();}while(0)
/* DQ 口配置 */
#define VS10XX_DQ_GPIO_PORT        GPIOC
#define VS10XX_DQ_GPIO_PIN          GPIO_PIN_13
#define VS10XX_DQ_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE();}while(0)
```

接下来就是用根据 1053 手册配置这些控制 IO 的功能,我们编写 1053 的初始化函数:

```
/**
 * @brief      VS10XX 初始化
 * @param      无
 * @retval     无
 */
void vs10xx_init(void)
{
    VS10XX_RST_GPIO_CLK_ENABLE(); /* VS10XX_RST 脚 时钟使能 */
    VS10XX_XCS_GPIO_CLK_ENABLE(); /* VS10XX_XCS 脚 时钟使能 */
    VS10XX_XDCS_GPIO_CLK_ENABLE(); /* VS10XX_XDCS 脚 时钟使能 */
    VS10XX_DQ_GPIO_CLK_ENABLE(); /* VS10XX_DQ 脚 时钟使能 */
    GPIO_InitTypeDef GPIO_InitStructure;
    /* RST 引脚模式设置,输出 */
    GPIO_InitStructure.Pin = VS10XX_RST_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStructure.Pull = GPIO_PULLUP;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(VS10XX_RST_GPIO_PORT, &GPIO_InitStructure);
    /* XCS 引脚模式设置,输出 */
    GPIO_InitStructure.Pin = VS10XX_XCS_GPIO_PIN;
    HAL_GPIO_Init(VS10XX_XCS_GPIO_PORT, &GPIO_InitStructure);
    /* XDCS 引脚模式设置,输出 */
    GPIO_InitStructure.Pin = VS10XX_XDCS_GPIO_PIN;
    HAL_GPIO_Init(VS10XX_XDCS_GPIO_PORT, &GPIO_InitStructure);
}
```

```

/* DQ 引脚模式设置,输入 */
GPIO_Initure.Pin = VS10XX_DQ_GPIO_PIN;
GPIO_Initure.Mode = GPIO_MODE_INPUT;
GPIO_Initure.Pull = GPIO_PULLUP;
GPIO_Initure.Speed = GPIO_SPEED_FREQ_HIGH;
HAL_GPIO_Init(VS10XX_DQ_GPIO_PORT, &GPIO_Initure);
spi1_init();
}

```

由于 VS1053 需要符合它自定的 SCI 时序来配置: 发送指令字节, 读写操作和 16 位的数据字, 在 SCK 下降沿更新数据, 先发送每个字节的 MSP, 所以我们需要根据 1053 的这种时序, 编写对应的配置的函数, 这里我们重新封装一下 SPI 读写函数为 vs10xx_spi_read_write_byte(), 当我们修改 SPI 的接口时也方便修改; 根据手册中的时序图我们编写 1053 的控制接口写命令函数如下:

```

/**
 * @brief      VS10XX 写命令
 * @param      address : 命令地址
 * @param      data    : 命令数据
 * @retval     无
 */
void vs10xx_write_cmd(uint8_t address, uint16_t data)
{
    while (VS10XX_DQ == 0); /* 等待空闲 */
    vs10xx_spi_speed_low(); /* 低速 */
    VS10XX_XDCS(1);
    VS10XX_XCS(0);
    vs10xx_spi_read_write_byte(VS_WRITE_COMMAND); /* 发送 VS10XX 的写命令 */
    vs10xx_spi_read_write_byte(address); /* 地址 */
    vs10xx_spi_read_write_byte(data >> 8); /* 发送高八位 */
    vs10xx_spi_read_write_byte(data); /* 低八位 */
    VS10XX_XCS(1);
    vs10xx_spi_speed_high(); /* 高速 */
}

```

该函数用于向 VS1053 发送命令, 这里要注意 VS1053 的写操作比读操作快 (写 1/4CLKI, 读 1/7 CLKI), 虽然说写寄存器最快可以到 1/4 CLKI, 但是经实测在 1/4 CLKI 的时候会出错, 所以在写寄存器的时候最好把 SPI 速度调慢点, 然后在发送音频数据的时候, 就可以 1/4 CLKI 的速度了。

有了上面的写寄存器操作, 我们还需要访问 1053 的工作状态、数据标头等信息, 还需要编写一个读函数, 同样参考 VS1053 的读时序, 我们编写如下接口:

```

/**
 * @brief      VS10XX 读寄存器
 * @param      address : 寄存器地址
 * @retval     读取到的数据
 */
uint16_t vs10xx_read_reg(uint8_t address)
{
    uint16_t temp = 0;
    while (VS10XX_DQ == 0); /* 非等待空闲状态 */
    vs10xx_spi_speed_low(); /* 低速 */
    VS10XX_XDCS(1);
    VS10XX_XCS(0);
    vs10xx_spi_read_write_byte(VS_READ_COMMAND); /* 发送 VS10XX 的读命令 */
    vs10xx_spi_read_write_byte(address); /* 地址 */
    temp = vs10xx_spi_read_write_byte(0xff); /* 读取高字节 */
    temp = temp << 8;
    temp += vs10xx_spi_read_write_byte(0xff); /* 读取低字节 */
    VS10XX_XCS(1);
    vs10xx_spi_speed_high(); /* 高速 */
    return temp;
}

```

对于 1053 的一些需要特殊配置的参数，如播放速度、硬件版本、比特率等，需要借助 RAM 和 RAM 地址扩展，所以对这部分我们也单独封装了函数用于操作这些特殊地址，我们定义为 vs10xx_read_ram()和 vs10xx_write_ram()，代码如下：

```
/**
 * @brief      VS10XX 读 RAM
 * @param      address : RAM 地址
 * @retval     读取到的数据
 */
static uint16_t vs10xx_read_ram(uint16_t address)
{
    uint16_t res;
    vs10xx_write_cmd(SPI_WRAMADDR, address);
    res = vs10xx_read_reg(SPI_WRAM);
    return res;
}

/**
 * @brief      VS10XX 写 RAM
 * @param      address : RAM 地址
 * @param      data    : 要写入的值
 * @retval     无
 */
static void vs10xx_write_ram(uint16_t address, uint16_t data)
{
    vs10xx_write_cmd(SPI_WRAMADDR, address);    /* 写 RAM 地址 */
    while (VS10XX_DQ == 0);                    /* 等待空闲 */
    vs10xx_write_cmd(SPI_WRAM, data);           /* 写 RAM 值 */
}
```

有了以上代码，我们就可以根据自己的需要配置 1053 工作在我们需要的模式下了，大家可以用这些读写函数测试相应的寄存器看看，其它的功能函数比较多，我们也实现了一些常用的函数，大家根据 1053 的手册和我们光盘提供的源码，对照着去实现。

2. audioplayer 代码

这部分我们需要根据 V1053 的手册推荐的初始化顺序时行配置。我们需要借助 SD 卡和文件系统把我们需要的歌曲传给 VS1053 播放。我们在 User 目录下新建一个《APP》文件夹，同时在该目录下新建 audioplayer.c 和 audioplayer.h 并加入到工程。

同样地，需要判断音乐文件类型，符合条件的再把相应的文件数据发送给 1053，同样需要用到文件类型判断的函数，我们在 FATFS 的扩展文件中已经实现了这个功能，在图片显示实验也演示了这部分代码的使用，我们把这个功能封装成了 audio_get_tnum()函数，这部分参考我们光盘源码即可。接下来我们来分析一下 audio_play_song()函数，它用于实现播放单个歌曲的功能，由于 1053 在解码的 FLAC 时有已知 Bug，我们在解码 FLAC 时需要根据官方给的 patches 文件利用 vs10xx_load_patch()刷入 1053 的 RAM 区，结合播放控制，我们的歌曲播放函数实现如下：

```
/**
 * @brief      播放一曲指定的歌曲
 * @param      pname   : 带路径的文件名
 * @retval     播放结果
 * @arg       KEY0_PRES , 下一曲
 * @arg       KEY2_PRES , 上一曲
 * @arg       其他      , 错误
 */
uint8_t audio_play_song(uint8_t *pname)
{
    FIL *fmp3;
    uint16_t br;
    uint8_t res, rval;
    uint8_t *databuf;
    uint16_t i = 0;
```

```

uint8_t key;
rval = 0;
fmp3 = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 申请内存 */
databuf = (uint8_t *)mymalloc(SRAMIN, 4096); /* 开辟 4096 字节的内存区域 */
if (databuf == NULL || fmp3 == NULL) rval = 0xFF; /* 内存申请失败 */
if (rval == 0)
{
    vs10xx_restart_play(); /* 重启播放 */
    vs10xx_set_all(); /* 设置音量等信息 */
    vs10xx_reset_decode_time(); /* 复位解码时间 */
    res = exfuns_file_type(pname); /* 得到文件后缀 */
    if (res == T_FLAC) /* 如果是 flac, 加载 patch */
    {
        vs10xx_load_patch((uint16_t *)vs1053b_patch, VS1053B_PATCHLEN);
    }
    res = f_open(fmp3, (const TCHAR *)pname, FA_READ); /* 打开文件 */
    if (res == 0) /* 打开成功 */
    {
        vs10xx_spi_speed_high(); /* 高速 */
        while (rval == 0)
        {
            res = f_read(fmp3, databuf, 4096, (UINT *)&br); /* 读出 4096 个字节 */
            i = 0;
            do /* 主播放循环 */
            {
                if (vs10xx_send_music_data(databuf + i) == 0)
                { /* 给 VS10XX 发送音频数据 */
                    i += 32;
                }
            } else
            {
                key = key_scan(0);
                switch (key)
                {
                    case KEY0_PRES: /* 下一曲 */
                    case KEY2_PRES: /* 上一曲 */
                        rval = key;
                        break;
                    case WKUP_PRES: /* 音量增加 */
                        if (vsset.mvol < 250)
                        {
                            vsset.mvol += 5;
                            vs10xx_set_volume(vsset.mvol);
                        }
                        else
                        {
                            vsset.mvol = 250;
                        }
                }
                /* 音量限制在:100~250, 显示的时候, 按照公式 (vol-100)/5, 显示, 也就是 0~30 */
                audio_vol_show((vsset.mvol - 100) / 5);
                break;
                    case KEY1_PRES: /* 音量减 */
                        if (vsset.mvol > 100)
                        {
                            vsset.mvol -= 5;
                            vs10xx_set_volume(vsset.mvol);
                        }
                        else
                        {
                            vsset.mvol = 100;
                        }
                }
                /* 音量限制在:100~250, 显示的时候, 按照公式 (vol-100)/5, 显示, 也就是 0~30 */
                audio_vol_show((vsset.mvol - 100) / 5);
            }
        }
    }
}

```

```

        break;
    }
    audio_msg_show(fmp3->obj.objsize); /* 显示信息 */
}
} while (i < 4096); /* 循环发送 4096 个字节 */
if (br != 4096 || res != 0)
{
    rval = KEY0_PRES;
    break; /* 读完了 */
}
}
f_close(fmp3);
}
else
{
    rval = 0xFF; /* 出现错误 */
}
}
myfree(SRAMIN, databuf);
myfree(SRAMIN, fmp3);
return rval;
}

```

测试通过单曲播放的功能后，我们可以根据需要，再封装一个目录循环播放的函数，利用 VS1053 解码音乐数据的间隔我们显示一些跟音乐文件相关的信息，来完成类似音乐播放器的功能，我们封装为了 `audio_play()` 函数，这部分代码实现起来就相对容易一些了，大家自行实现或者参考我们光盘的源代码即可。

3. main.c 代码

解决了音乐播放的问题，`main.c` 函数实现起来就比较简单了，我们可以按照流程图的设计思路进行编写即可：

```

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    norflash_init(); /* 初始化 NORFLASH */
    vs10xx_init(); /* VS10XX 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */
    exfuns_init(); /* 为 fatfs 相关变量申请内存 */
    f_mount(fs[0], "0:", 1); /* 挂载 SD 卡 */
    f_mount(fs[1], "1:", 1); /* 挂载 FLASH */
    while (fonts_init()) /* 检查字库 */
    {
        lcd_show_string(30, 50, 200, 16, 16, "Font Error!", RED);
        delay_ms(200);
        lcd_fill(30, 50, 240, 66, WHITE); /* 清除显示 */
        delay_ms(200);
    }
    text_show_string(30, 50, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
    text_show_string(30, 70, 200, 16, "音乐播放器 实验", 16, 0, RED);
    text_show_string(30, 110, 200, 16, "KEY0:NEXT KEY2:PREV", 16, 0, RED);
    text_show_string(30, 130, 200, 16, "KEY_UP:VOL+ KEY1:VOL-", 16, 0, RED);
    while (1)

```

```
{
    LED1(0);
    text_show_string(30, 150, 200, 16, "存储器测试...", 16, 0, RED);
    printf("Ram Test:0X%04X\r\n", vs10xx_ram_test()); /* 打印RAM测试结果 */
    text_show_string(30, 150, 200, 16, "正弦波测试...", 16, 0, RED);
    vs10xx_sine_test();
    text_show_string(30, 150, 200, 16, "<<音乐播放器>>", 16, 0, RED);
    LED1(1);
    audio_play();
}
```

到这里本实验的代码基本就编写完成了，我们准备好音乐文件放到 SD 卡根目录下的《MUSIC》夹下测试本实验的代码，可以把 VS1053 的配置函数加到 USMART 下，这样就能用 USMART 来测试和调试 VS1053 了。

54.4 下载验证

在代码编译成功之后，我们下载代码到正点原子战舰 STM32 开发板上，程序先执行字库监测，然后对 VS1053 进行 RAM 测试和正弦测试。

当检测到 SD 卡根目录的 MUSIC 文件夹有有效音频文件(VS1053 所支持的格式)的时候，就开始自动播放歌曲了，如图 54.4.1 所示：



图 54.4.1 音乐播放中

从上图可以看出，总共 3 首歌曲，当前正在播放第 3 首歌曲，歌曲名、播放时间、总时长、码率、音量等信息等也都有显示。此时 DS0 会随着音乐的播放而闪烁，2 秒闪烁一次。

此时我们便可以听到开发板板载喇叭播放出来的音乐了，也可以在开发板的 PHONE 端子插入耳机来听歌。同时，我们可以通过按 KEY0 和 KEY2 来切换下一曲和上一曲，通过 KEY_UP 按键来控制音量增加，通过 KEY1 控制音量减小。

本实验，我们还可以通过 USMART 来测试 VS1053 的其他功能，通过将 vs10xx.c 里面的部分函数加入 USMART 管理，我们可以很方便的设置/获取 VS1053 各种参数，达到验证测试的目的。有兴趣的朋友，可以实验测试一下。

至此，我们就完成了一个简单的 MP3 播放器了，在此基础上进一步完善，就可以做出一个比较实用的 MP3 了。大家可以自己发挥想象，做出一个你心仪的 MP3。

第五十五章 录音机实验

上一章,我们实现了一个简单的音乐播放器,本章我们将在上一章的基础上,继续用 VS1053 实现一个简单的录音机,录制 WAV 格式的录音。

55.1 WAV 格式简介

55.2 硬件设计

55.3 软件设计

55.4 下载验证

55.1 WAV 格式简介

WAV 即 WAVE 文件, WAV 是计算机领域最常用的数字化声音文件格式之一,它是微软专门为 Windows 系统定义的波形文件格式 (WaveformAudio), 由于其扩展名为 "*.wav"。它符合 RIFF(Resource Interchange File Format)文件规范,用于保存 Windows 平台的音频信息资源,被 Windows 平台及其应用程序所广泛支持,该格式也支持 MSADPCM, CCITTALAW 等多种压缩运算法,支持多种音频数字,取样频率和声道,标准格式化的 WAV 文件和 CD 格式一样,也是 44.1K 的取样频率,16 位量化数字,因此在声音文件质量和 CD 相差无几!

要想实现 WAV 录音得先了解一下 WAV 文件的格式。WAVE 文件的数据采用“Chunk”来存储。因此,如果想要在 WAVE 文件中补充一些新的信息,只需要在新 Chunk 中添加信息,而不需要改变整个文件。所以可以把 WAVE 文件看成是很多不同 Chunk 的集合。每个 Chunk 由块标识符、数据大小和数据三部分组成,如图 55.1.1 所示:



图 55.1.1 Chunk 结构示意图

其中块标识符由 4 个 ASCII 码构成,数据大小则标出紧跟其后的数据的长度(单位为字节),注意这个长度不包含块标识符和数据大小的长度,即不包含最前面的 8 个字节。所以实际 Chunk 的大小为数据大小加 8。

对于一个基本的 WAVE 文件而言,以下三种 Chunk 是必不可少的:文件中第一个 Chunk 是 RIFF Chunk,然后是 FMT Chunk,最后是 Data Chunk。对于其他的 Chunk,顺序没有严格的限制。使用 WAVE 文件的应用程序必须具有读取以上三种 chunk 信息的能力,如果程序想要复制 WAVE 文件,必须拷贝文件中所有的 chunk。本章,我们主要讨论 PCM,因为这个最简单,它只包含 3 个 Chunk,我们看一下它的文件构成,如图 55.1.2。

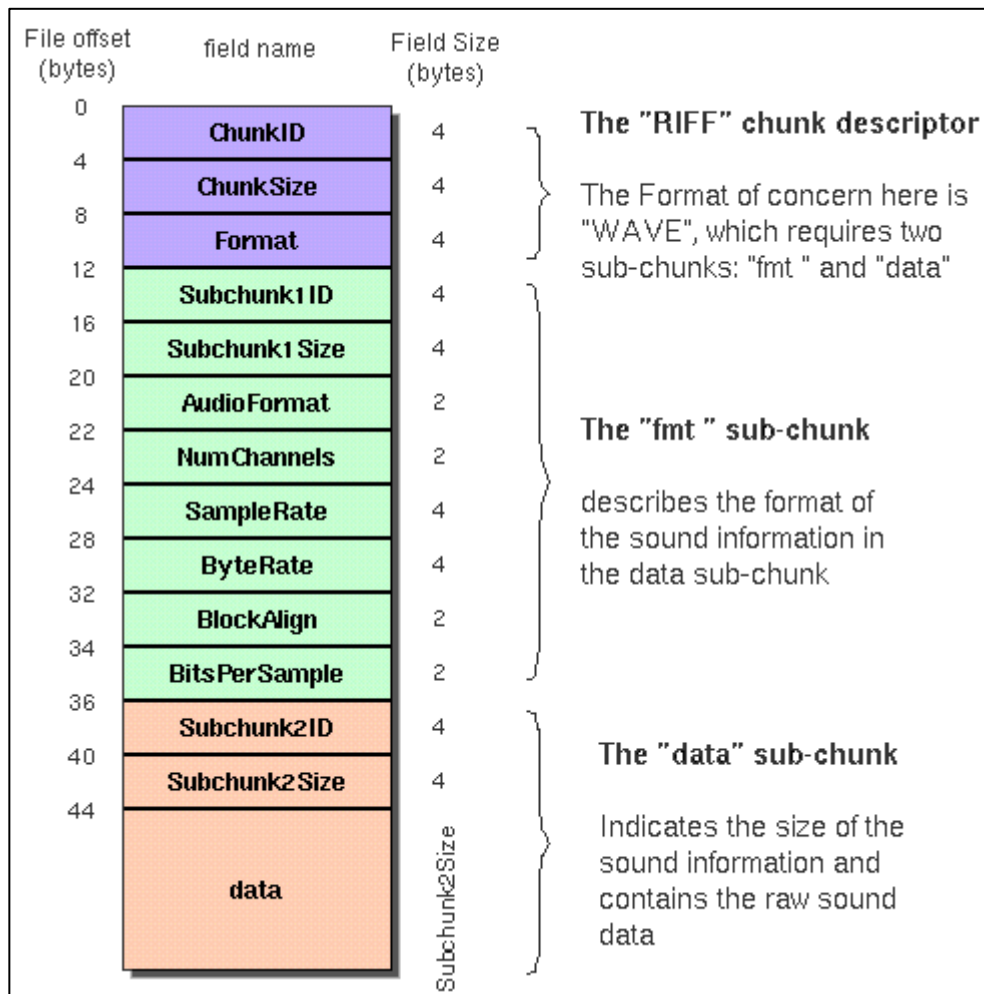


图 55.1.2 PCM 格式的 wav 文件构成

可以看到，不同的 Chunk 有不同的长度，编码文件时，按照 Chunk 的字节和位序排列好之后写入文件头，加上 wav 的后缀，就可以生成一个能被正确解析的 wav 文件了，对于 PCM 结构，我们只需要把获取到的音频数据填充到 Data Chunk 中即可。我们将利用 VS1053 实现 16 位，8KHz 采样率的单声道 WAV 录音(PCM 格式)。

首先，我们来看看 RIFF 块 (RIFF WAVE Chunk)，该块以“RIFF”作为标示，紧跟 wav 文件大小（该大小是 wav 文件的总大小-8），然后数据段为“WAVE”，表示是 wav 文件。RIFF 块的 Chunk 结构如下：

```
typedef __PACKED_STRUCT
{
    uint32_t ChunkID;           /* chunk id;这里固定为"RIFF",即 0X46464952 */
    uint32_t ChunkSize;         /* 集合大小;文件总大小-8 */
    uint32_t Format;            /* 格式;WAVE,即 0X45564157 */
} ChunkRIFF;
```

接着，我们看看 Format 块 (FormatChunk)，该块以“fmt”作为标示（注意有个空格！），一般情况下，该段的大小为 16 个字节，但是有些软件生成的 wav 格式，该部分可能有 18 个字节，含有 2 个字节的附加信息。Format 块的 Chunk 结构如下：

```
typedef __PACKED_STRUCT
{
    uint32_t ChunkID;           /* chunk id;这里固定为"fmt ",即 0X20746D66 */
    uint32_t ChunkSize;         /* 子集合大小(不包括 ID 和 Size);这里为:20 */
    uint16_t AudioFormat;       /* 音频格式;0X10,表示线性 PCM;0X11 表示 IMA ADPCM */
    uint16_t NumOfChannels;     /* 通道数量;1,表示单声道;2,表示双声道; */
    uint32_t SampleRate;        /* 采样率;0X1F40,表示 8KHz */
    uint32_t ByteRate;          /* 字节速率 */
}
```

```
uint16_t BlockAlign;          /* 块对齐(字节) */
uint16_t BitsPerSample;       /* 单个采样数据大小;4 位 ADPCM, 设置为 4 */
//uint16_t ByteExtraData;      /* 附加的数据字节;2 个; 线性 PCM, 没有这个参数 */
//uint16_t ExtraData;          /* 附加的数据, 单个采样数据块大小;0x1F9:505 字节 线性 PCM, 没有这个参数 */
} ChunkFMT;
```

接下来, 我们再看看 Fact 块 (Fact Chunk), 该块为可选块, 以 “fact” 作为标示, 不是每个 WAV 文件都有, 在非 PCM 格式的文件中, 一般会在 Format 结构后面加入一个 Fact 块, 该块 Chunk 结构如下:

```
typedef __PACKED_STRUCT
{
    uint32_t ChunkID;           /* chunk id;这里固定为"fact",即 0x74636166 */
    uint32_t ChunkSize;         /* 子集合大小(不包括 ID 和 Size);这里为:4 */
    uint32_t NumOfSamples;       /* 采样的数量 */
} ChunkFACT;
```

DataFactSize 是这个 Chunk 中最重要的数据, 如果这是某种压缩格式的声音文件, 那么从这里就可以知道他解压缩后的大小。对于解压时的计算会有很大的好处! 不过本章我们使用的是 PCM 格式, 所以不存在这个块。

最后, 我们来看看数据块 (Data Chunk), 该块是真正保存 wav 数据的地方, 以 “data” 作为该 Chunk 的标示。然后是数据的大小。紧接着就是 wav 数据。根据 Format Chunk 中的声道数以及采样 bit 数, wav 数据的 bit 位置可以分成如表 55.1.1 所示的几种形式:

| 单声道 | 取样1 | 取样2 | 取样3 | 取样4 |
|-------|----------------|----------------|----------------|----------------|
| 8位量化 | 声道0 | 声道0 | 声道0 | 声道0 |
| 双声道 | 取样1 | | 取样2 | |
| 8位量化 | 声道0(左) | 声道1(右) | 声道0(左) | 声道1(右) |
| 单声道 | 取样1 | | 取样2 | |
| 16位量化 | 声道0(低字节) | 声道0(高字节) | 声道0(低字节) | 声道0(高字节) |
| 双声道 | 取样1 | | | |
| 16位量化 | 声道0 (左,低字节) | 声道0 (左,高字节) | 声道1 (右,低字节) | 声道1 (右,高字节) |

表 55.1.1 WAVE 文件数据采样格式

本实验, 我们采用的是 16 位, 单声道, 所以每个取样为 2 个字节, 低字节在前, 高字节在后。数据块的 Chunk 结构如下:

```
typedef __PACKED_STRUCT
{
    uint32_t ChunkID;           /* chunk id;这里固定为"data",即 0x61746164 */
    uint32_t ChunkSize;         /* 子集合大小(不包括 ID 和 Size);文件大小-60 */
} ChunkDATA;
```

通过以上学习, 我们对 WAVE 文件结构有了个大概了解。如果对 WAV 的格式还存在疑问, 请参考我们 “A 盘→6, 软件资料→8, WAV 文件格式说明” 的内容。

接下来, 我们看看如何使用 VS1053 实现 WAV (PCM 格式) 录音。

1. 激活 PCM 录音

VS1053 激活 PCM 录音需要设置的寄存器和相关位如表 55.1.2 所示:

| 寄存器 | 位域 | 说明 |
|-------------|-----------|---|
| SCI_MODE | 2, 12, 14 | 开始 ADPCM 模式, 选择: 咪/线路1 |
| SCI_AICTRL0 | 15..0 | 采样率 8000..48000 Hz (在录音启动时读取的) |
| SCI_AICTRL1 | 15..0 | 录音增益 (1024 = 1×) 或 0 是自动增益控制 (AGC) |
| SCI_AICTRL2 | 15..0 | 自动增益放大器的最大值 (1024 = 1×, 65535 = 64×) |
| SCI_AICTRL3 | 1..0 | 0=联合立体声(共用 AGC), 1=双声道(各自的 AGC), 2=左通道, 3=右通道 |
| | 2 | 0=IMA ADPCM 模式, 1=线性 PCM 模式 |
| | 15..3 | 保留, 设置为 0 |

表 55.1.2 VS1053 激活 PCM 录音相关寄存器

通过设置 SCI_MODE 寄存器的 2、12、14 位, 来激活 PCM 录音, SCI_MODE 的各位描述见表 48.1.4(也可以参考 VS1053 的数据手册)。SCI_AICTRL0 寄存器用于设置采样率, 我们本章用的是 8K 的采样率, 所以设置这个值为 8000 即可。SCI_AICTRL1 寄存器用于设置 AGC, 1024 相当于数字增加 1, 这里建议大家设置 AGC 在 4 (4*1024) 左右比较合适。SCI_AICTRL2 用于设置自动 AGC 的时候的最大值, 当设置为 0 的时候表示最大 64(65536), 这个大家按自己的需要设置即可。最后, SCI_AICTRL3, 我们本章用到的是咪头线性 PCM 单声道录音, 所以设置该寄存器值为 6。

通过这几个寄存器的设置, 我们就激活 VS1053 的 PCM 录音了。不过, VS1053 的 PCM 录音有一个小 BUG, 必须通过加载 patch 才能解决, 如果不加载 patch, 那么 VS1053 是不输出 PCM 数据的, VLSI 提供了我们这个 patch, 只需要通过软件加载即可。

2. 读取 PCM 数据

在激活了 PCM 录音之后, SCI_HDAT0 和 SCI_HDAT1 有了新的功能。VS1053 的 PCM 采样缓冲区由 1024 个 16 位数据组成, 如果 SCI_HDAT1 大于 0, 则说明可以从 SCI_HDAT0 读取至少 SCI_HDAT1 个 16 位数据, 如果数据没有被及时读取, 那么将溢出, 并返回空的状态。

注意, 如果 SCI_HDAT1 ≥ 896, 最好等待缓冲区溢出, 以免数据混叠。所以, 对我们来说, 只需要判断 SCI_HDAT1 的值非零, 然后从 SCI_HDAT0 读取对应长度的数据, 即完成一次数据读取, 以此循环, 即可实现 PCM 数据的持续采集。

最后, 我们看看本章实现 WAV 录音需要经过哪些步骤:

1) 设置 VS1053 PCM 采样参数

这一步, 我们要设置 PCM 的格式 (线性 PCM)、采样率 (8K)、位数 (16 位)、通道数 (单声道) 等重要参数, 同时还要选择采样通道 (咪头), 还包括 AGC 设置等。可以说这里的设置直接决定了我们 wav 文件的性质。

2) 激活 VS1053 的 PCM 模式, 加载 patch

通过激活 VS1053 的 PCM 格式, 让其开始 PCM 数据采集, 同时, 由于 VS1053 的 BUG, 我们需要加载 patch, 以实现正常的 PCM 数据接收。

3) 创建 WAV 文件, 并保存 wav 头

在前两部设置成功之后, 我们即可正常的从 SCI_HDAT0 读取我们需要的 PCM 数据了, 不过在这之前, 我们需要先在创建一个新的文件, 并写入 wav 头, 然后才能开始写入我们的 PCM 数据。

4) 读取 PCM 数据

经过前面几步的处理, 这一步就比较简单了, 只需要不停的从 SCI_HDAT0 读取数据, 然后存入 wav 文件即可, 不过这里我们还需要做文件大小统计, 在最后的时候写入 wav 头里面。

5) 计算整个文件大小, 重新保存 wav 头并关闭文件

在结束录音的时候, 我们必须知道本次录音的大小 (数据大小和整个文件大小), 然后更新 wav 头, 重新写入文件, 最后因为 FATFS, 在文件创建之后, 必须调用 f_close, 文件才会真正写入到磁盘里面! 所以最后还需要调用 f_close, 以保存文件。

54.2 硬件设计

1. 例程功能

开机的时候先检测字库，然后初始化 VS1053，进行 RAM 测试和正弦测试，之后，检测 SD 卡根目录是否存在 RECORDER 文件夹，如果不存在则创建，如果创建失败，则报错。在找到 SD 卡的 RECORDER 文件夹后，即设置 VS1053 进入录音模式，此时可以在耳机听到 VS1053 采集的音频。KEY0 用于开始/暂停录音，KEY2 用于保存并停止录音，KEY_UP 用于 AGC 增加、KEY1 用于 AGC 减小，TPAD 用于播放最近一次的录音。当我们按下 KEY0 的时候，可以在屏幕上看到录音文件的名称，以及录音时间，然后通过 KEY2 可以保存该文件，同时停止录音（文件名和时间也都将清零），在完成一个录音后，我们可以通过按 TPAD 按键，来试听刚刚的录音。DS0 用于提示程序正在运行，DS1 用于指示当前是否处于录音暂停状态。

2. 硬件资源

本实验，大家需要准备 1 个 microSD/SD 卡（在里面新建一个 MUSIC 文件夹，并存放一些歌曲在 MUSIC 文件夹下）和一个耳机（非必备），分别插入 SD 卡接口和耳机接口，然后下载本实验就可以实现录音机的效果。实验用到的硬件资源如下：

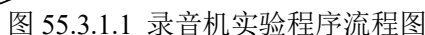
- 1) LED 灯
 - LED0 - PB5
 - LED1 - PE5
- 2) 独立按键
 - KEY0 - PE4
 - KEY1 - PE3
 - KEY2 - PE2
 - KEY_UP - PA0 (程序中的宏名:WK_UP)
- 3) 串口 1 (PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 5) SD 卡：通过 SDIO(SDIO_D0~D4(PC8~PC11),SDIO_SCK(PC12),SDIO_CMD(PD2))连接
- 6) NOR FLASH(SPI FLASH 芯片,连接在 SPI2 上)
- 7) VS1053 芯片，通过 SPI1 驱动
- 8) 功放芯片 HT6872,用于放大 1053 的输出以支持扬声器
- 9) 开发板板载的咪头或自备麦克风输入
- 10) TPAD:
 - 需要用跳线帽连接 TPAD 和 ADC 引脚

录音机实验与上一章(音乐播放器实验)用到的硬件资源基本一样，我们这里就不重复介绍了，有差异的是这次我们用到板载的咪头用于信号输入，也可以通过 3.5mm 的音频接口通过 LINE_IN 接入麦克风输入录音音源。

55.3 程序设计

55.3.1 程序流程图

程序的设计流程如下：



829


```

vs10xx_write_cmd(SPI_AICTRL2, 0); /* 设置增益最大值,0,代表最大值 65536=64X */
vs10xx_write_cmd(SPI_AICTRL3, 6); /* 左通道 (MIC 单声道输入), 线性 PCM */
/* 设置 VS10XX 的时钟, MULT:2 倍频; ADD:不允许; CLK:12.288Mhz */
vs10xx_write_cmd(SPI_CLOCKF, 0X2000);
vs10xx_write_cmd(SPI_MODE, 0x1804); /* MIC, 录音激活 */
delay_ms(5); /* 等待至少 1.35ms */
vs10xx_load_patch((uint16_t *)wav_plugin, 40); /* VS1053 的 WAV 录音需要 patch */
}

```

该函数就是用我们前面介绍的方法,激活 VS1053 的 PCM 模式,本章,我们使用的是 8Khz 采样率,16 位单声道线性 PCM 模式,AGC 通过函数参数设置。最后加载 patch(用于修复 VS1053 录音 BUG)。

由于最后要把录音写入到文件,这里需要准备 wav 的文件头,为方便,我们定义了一个 __WaveHeader 结构体来定义文件头的字节,这个结构体包含了前面提到的 wav 文件的数据结构块:

```

typedef __PACKED_STRUCT
{
    ChunkRIFF riff; /* riff 块 */
    ChunkFMT fmt; /* fmt 块 */
    //ChunkFACT fact; /* fact 块 线性 PCM,没有这个结构体 */
    ChunkDATA data; /* data 块 */
} __WaveHeader;

```

我们定义一个 recoder_wav_init()函数方便初始化文件信息,代码如下:

```

void recoder_wav_init(__WaveHeader *wavhead)
{
    wavhead->riff.ChunkID = 0X46464952; /* "RIFF" */
    wavhead->riff.Format = 0X45564157; /* "WAVE" */
    wavhead->fmt.ChunkID = 0X20746D66; /* "fmt " */
    wavhead->fmt.ChunkSize = 16; /* 大小为 16 个字节 */
    wavhead->fmt.AudioFormat = 1; /* 1, 表示 PCM; 0, 表示 IMA ADPCM; */
    wavhead->fmt.NumOfChannels = 1; /* 单声道 */
    wavhead->fmt.SampleRate = 8000; /* 8Khz 采样率 采样速率 */
    /* 字节速率,等于采样率*2(单声道,16位) */
    wavhead->fmt.ByteRate = wavhead->fmt.SampleRate * 2;
    wavhead->fmt.BlockAlign = 2; /* 块大小,2 个字节为一个块 */
    wavhead->fmt.BitsPerSample = 16; /* 16 位 PCM */
    wavhead->data.ChunkID = 0X61746164; /* "data" */
    wavhead->data.ChunkSize = 0; /* 数据大小,还需要计算 */
}

```

录音完成我们还要重新计算录音文件的大小写入文件头,以保证音频文件能正常被解析。录音时通过读取 SCI_HDAT0 寄存器中的 16 位数据获取到录音的 ADC 值,我们把这些数据直接按顺序写入文件即可完成录音操作,结合文件操作和按键功能定义,我们用 recoder_play()函数实现录音过程,代码如下:

```

/**
 * @brief      录音机
 * @note      所有录音文件,均保存在 SD 卡 RECORDER 文件夹内
 * @param      无
 * @retval     0, 成功; 0xFF, 播放出错;
 */
uint8_t recoder_play(void)
{
    uint8_t res;
    uint8_t key;
    uint8_t rval = 0;
    __WaveHeader *wavhead = 0;
    uint32_t sectorsize = 0;
    FIL *f_rec = 0; /* 文件 */
    DIR recdir; /* 目录 */
    UINT bw; /* 写入长度 */
    uint8_t *recbuf; /* 数据内存 */
}

```

```

uint16_t w;
uint16_t idx = 0;
char *pname = 0;
uint8_t timecnt = 0;    /* 计时器 */
uint32_t recsec = 0;    /* 录音时间 */
uint8_t recagc = 4;     /* 默认增益为 4 */
uint8_t rec_sta = 0;    /* 录音状态
    * [7] : 0, 没有录音; 1, 有录音;
    * [6:1]: 保留
    * [0] : 0, 正在录音; 1, 暂停录音
    */

while (f_opendir(&readdir, "0:/RECORDER")) /* 打开录音文件夹 */
{
    text_show_string(30, 230, 240, 16, "RECORDER 文件夹错误!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 230, 240, 246, WHITE); /* 清除显示 */
    delay_ms(200);
    f_mkdir("0:/RECORDER"); /* 创建该目录 */
}

pname = mymalloc(SRAMIN, 30); /* 申请 30 个字节内存, 类似 "0:RECORDER/REC00001.wav" */
f_rec = (FIL *)mymalloc(SRAMIN, sizeof(FIL)); /* 开辟 FIL 字节的内存区域 */
/* 开辟 __WaveHeader 字节的内存区域 */
wavhead = (__WaveHeader *)mymalloc(SRAMIN, sizeof(__WaveHeader));
recbuf = mymalloc(SRAMIN, 512);
if (pname == NULL || f_rec == NULL || wavhead == NULL || recbuf == NULL)
{
    rval = 1; /* 申请失败 */
}
if (rval == 0) /* 内存申请 OK */
{
    recoder_enter_rec_mode(1024 * recagc);
    while (vs10xx_read_reg(SPI_HDAT1) >> 8); /* 等到 buf 较为空闲再开始 */
    recoder_show_time(recsec); /* 显示时间 */
    recoder_show_agc(recagc); /* 显示 agc */
    pname[0] = 0; /* pname 没有任何文件名 */
    while (rval == 0)
    {
        key = key_scan(0);
        switch (key)
        {
            case KEY2_PRES: /* STOP&SAVE */
                if (rec_sta & 0X80) /* 有录音 */
                {
                    /* 整个文件的大小-8 */
                    wavhead->riff.ChunkSize = sectorsize * 512 + 36;
                    wavhead->data.ChunkSize = sectorsize * 512; /* 数据大小 */
                    f_lseek(f_rec, 0); /* 偏移到文件头 */
                    f_write(f_rec, (const void *)wavhead,
                        sizeof(__WaveHeader), &bw); /* 写入头数据 */
                    f_close(f_rec);
                    sectorsize = 0;
                }
                rec_sta = 0;
                recsec = 0;
                LED1(1); /* 关闭 DS1 */
                /* 清除显示, 清除之前显示的录音文件名 */
                lcd_fill(30, 230, 240, 246, WHITE);
                recoder_show_time(recsec); /* 显示时间 */
                break;
            case KEY0_PRES: /* REC/PAUSE */
                if (rec_sta & 0X01) /* 原来是暂停, 继续录音 */

```

```

    {
        rec_sta &= 0XFE;          /* 取消暂停 */
    }
    else if (rec_sta & 0X80) /* 已经在录音了, 暂停 */
    {
        rec_sta |= 0X01;          /* 暂停 */
    }
    else /* 还没开始录音 */
    {
        rec_sta |= 0X80;          /* 开始录音 */
        recoder_new_pathname((uint8_t *)pname); /* 得到新的名字 */
        /* 显示当前录音文件名字 */
        text_show_string(30, 230, 240, 16, pname + 11, 16, 0, RED);
        recoder_wav_init(wavhead); /* 初始化 wav 数据 */
        res = f_open(f_rec, pname, FA_CREATE_ALWAYS | FA_WRITE);
        if (res) /* 文件创建失败 */
        {
            rec_sta = 0;          /* 创建文件失败, 不能录音 */
            rval = 0XFE;          /* 提示是否存在 SD 卡 */
        }
        else
        {
            res = f_write(f_rec, (const void *)wavhead,
                           sizeof(__WaveHeader), &bw); /* 写入头数据 */
        }
    }
    LED1(!(rec_sta & 0X01)); /* 提示录音状态 */
    break;

case WKUP_PRES: /* AGC+ */
case KEY1_PRES: /* AGC- */
    if (key == WKUP_PRES)
    {
        recagc++;
    }
    else if (recagc)
    {
        recagc--;
    }
    /* 范围限定为 0~15.0, 自动 AGC; 其他, AGC 倍数; */
    if (recagc > 15) recagc = 15;
    recoder_show_agc(recagc);
    /* 设置增益, 0, 自动增益. 1024 相当于 1 倍, 512 相当于 0.5 倍 */
    vs10xx_write_cmd(SPI_AICTRL1, 1024 * recagc);
    break;
}
/* 读取数据 */
if (rec_sta == 0X80) /* 已经在录音了 */
{
    w = vs10xx_read_reg(SPI_HDAT1);
    if ((w >= 256) && (w < 896))
    {
        idx = 0;
        while (idx < 512) /* 一次读取 512 字节 */
        {
            w = vs10xx_read_reg(SPI_HDAT0);
            recbuf[idx++] = w & 0XFF;
            recbuf[idx++] = w >> 8;
        }
        res = f_write(f_rec, recbuf, 512, &bw); /* 写入文件 */
        if (res)
        {
            printf("err:%d\r\n", res);
            printf("bw:%d\r\n", bw);
        }
    }
}

```

```

        break;        /* 写入出错 */
    }
    sectorsize++;      /* 扇区数增加 1, 约为 32ms */
}
}
else /* 没有开始录音, 则检测 TPAD 按键 */
{
    if (tpad_scan(0) && pname[0]) /* 如果触摸按键被按下, 且 pname 不为空 */
    {
        text_show_string(30, 230, 240, 16, "播放:", 16, 0, RED);
        /* 显示当播放的文件名字 */
        text_show_string(30 + 40, 230, 240, 16, pname + 11, 16, 0, RED);
        rec_play_wav((uint8_t *)pname); /* 播放 pname */
        lcd_fill(30, 230, 240, 246, WHITE); /* 清除显示, 清除之前显示的录音文件名 */
        recoder_enter_rec_mode(1024 * recagc); /* 重新进入录音模式 */
        while (vs10xx_read_reg(SPI_HDAT1) >> 8); /* 等到 buf 较为空闲再开始 */
        recoder_show_time(recsec); /* 显示时间 */
        recoder_show_agc(recagc); /* 显示 agc */
    }
    delay_ms(5);
    timecnt++;
    if ((timecnt % 20) == 0) LED0_TOGGLE(); /* DS0 闪烁 */
}
if (recsec != (sectorsize * 4 / 125)) /* 录音时间显示 */
{
    LED0_TOGGLE(); /* DS0 闪烁 */
    recsec = sectorsize * 4 / 125;
    recoder_show_time(recsec); /* 显示时间 */
}
}
}

myfree(SRAMIN, wavhead);
myfree(SRAMIN, recbuf);
myfree(SRAMIN, f_rec);
myfree(SRAMIN, pname);
return rval;
}

```

2. main.c 代码

由于我们把大部分功能已经在 `recoder_play()` 中实现了, `main` 函数进行必要的外设初始化, 显示相关的数据信息后, 调用该接口即可实现我们需要的录音机功能了, 最后我们在 `main.c` 中实现代码如下:

```

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    sram_init(); /* SRAM 初始化 */
    norflash_init(); /* 初始化 NORFLASH */
    tpad_init(6); /* 初始化 TPAD */
    vs10xx_init(); /* VS10XX 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */
}

```

```

exfuncs_init(); /* 为 fatfs 相关变量申请内存 */
f_mount(fs[0], "0:", 1); /* 挂载 SD 卡 */
f_mount(fs[1], "1:", 1); /* 挂载 FLASH */
while (fonts_init()) /* 检查字库 */
{
    lcd_show_string(30, 50, 200, 16, 16, "Font Error!", RED);
    delay_ms(200);
    lcd_fill(30, 50, 240, 66, WHITE); /* 清除显示 */
    delay_ms(200);
}
text_show_string(30, 50, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
text_show_string(30, 70, 200, 16, "WAV 录音机 实验", 16, 0, RED);
text_show_string(30, 110, 200, 16, "KEY0:REC/PAUSE", 16, 0, RED);
text_show_string(30, 130, 200, 16, "KEY2:STOP&SAVE", 16, 0, RED);
text_show_string(30, 150, 200, 16, "KEY_UP:AGC+ KEY1:AGC-", 16, 0, RED);
text_show_string(30, 170, 200, 16, "TPAD:Play The File", 16, 0, RED);

while (1)
{
    LED1(0);
    text_show_string(30, 190, 200, 16, "存储器测试...", 16, 0, RED);
    printf("Ram Test:0X%04X\r\n", vs10xx_ram_test()); /* 打印 RAM 测试结果 */

    text_show_string(30, 190, 200, 16, "正弦波测试...", 16, 0, RED);
    vs10xx_sine_test();

    text_show_string(30, 190, 200, 16, "<<WAV 录音机>>", 16, 0, RED);
    LED1(1);
    recoder_play();
}

```

可以看到 main 函数与音乐播放器实验十分类似，封装好了 APP，main 函数会精简很多。

55.4 下载验证

在代码编译成功之后，我们下载代码到正点原子战舰 STM32 开发板上，程序先检测字库，然后对 VS1053 进行 RAM 测试和正弦测试，之后检测 SD 卡的 RECORDER 文件夹，一切顺利通过之后，激活 VS1053 的 PCM 录音模式，得到，如图 55.4.1 所示

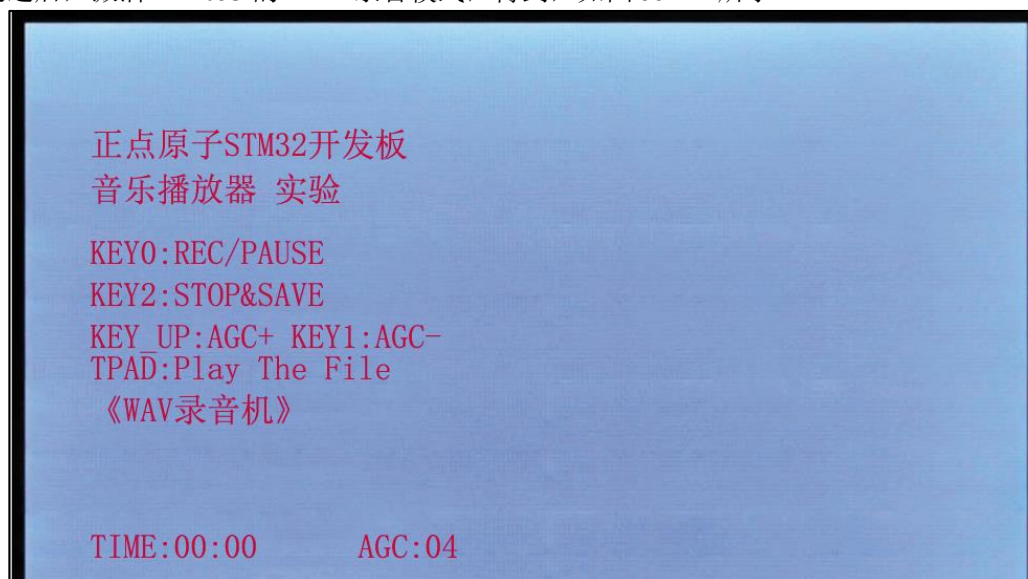


图 55.4.1 录音机实验界面

此时，我们按下 KEY0 就开始录音了，此时看到屏幕显示录音文件的名字以及录音时长，如图 55.4.2 所示：

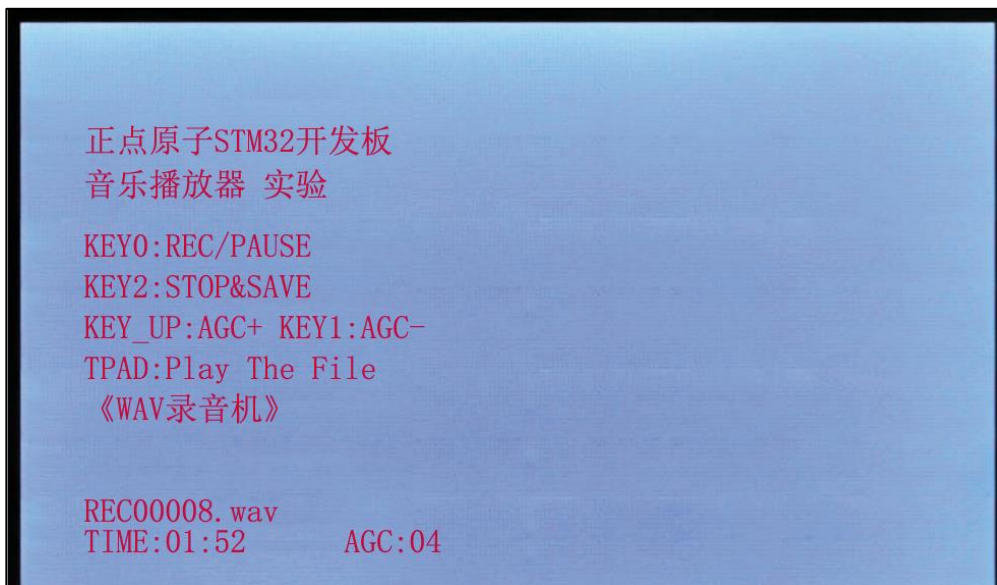


图 55.4.2 录音进行中

在录音的时候,按下 KEY0 执行暂停/继续录音的切换,通过 DS1 指示录音暂停,按 KEY_UP 和 KEY1 可以调节 AGC, AGC 越大,越灵敏,不过不建议设置太大,因为这可能导致失真。通过按下 KEY2,可以停止当前录音,并保存录音文件。在完成一次录音文件保存之后,我们可以通过按 TPAD 按键,来实现播放这个录音文件(即播放最近一次的录音文件),实现试听。

我们可以把录音完成的 wav 文件放到电脑上,可以通过一些播放软件播放并查看详细的音频编码信息,本例程使用的是 KMPlayer 播放,查看到的信息如图 55.4.3 所示:

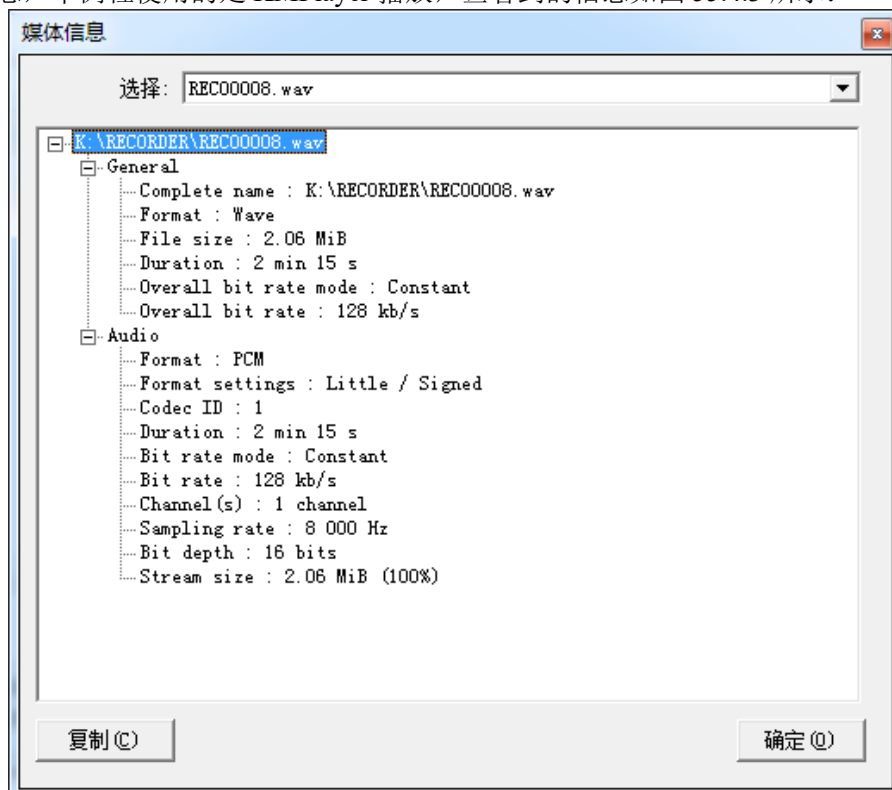


图 55.4.3 录音文件属性

这和我们程序设计时的效果一样,通过电脑端的播放器可以直接播放我们所录的音频。经实测,效果还是非常不错的。

第五十六章 DSP 测试实验

本章，我们将指导大家入门 STM32F103 的 DSP，手把手教大家搭建 DSP 库测试环境，同时通过对 DSP 库中的几个基本数学功能函数和 FFT 快速傅里叶变换函数的测试，让大家对 STM32F103 的 DSP 库有个基本的了解。

本章分为如下几个小节：

56.1 DSP 简介与环境搭建

56.2 硬件设计

56.3 软件设计

56.4 下载验证

56.1 DSP 简介与环境搭建

本节将分两个部分：1，STM32F103 DSP 简介；2，DSP 库运行环境搭建。

56.1.1 STM32F103 DSP 简介

STM32F103 采用 Cortex-M3 内核，没有内置硬件 FPU 单元，也没有 DSP 指令集。那么在数字信号处理能力方面比较有比较大的提升就要选择 Cortex-M4 和 Cortex-M7。我们这里只是通过调用 DSP 库进行测试。

接下来我们来介绍一下 STM32F1 的 DSP 库。STM32F1 的 DSP 库源码和测试实例在 ST 提供的 HAL 库固件包里就有提供：en.stm32cubeF1.zip 里面就有（该文件可以在 www.st.com 网站下载，搜索 STM32CubeF1 即可找到最新版本），该文件在：A 盘→8，STM32 参考资料→1，STM32CubeF1 固件包文件夹里面，解压该文件，即可找到 ST 提供的 DSP 库，详细路径为：光盘→8，STM32 参考资料→1，STM32CubeF1 固件包→STM32Cube_FW_F1_V1.8.0→Drivers→CMSIS→DSP，该文件夹下目录结构如图 56.1.1.3 所示。

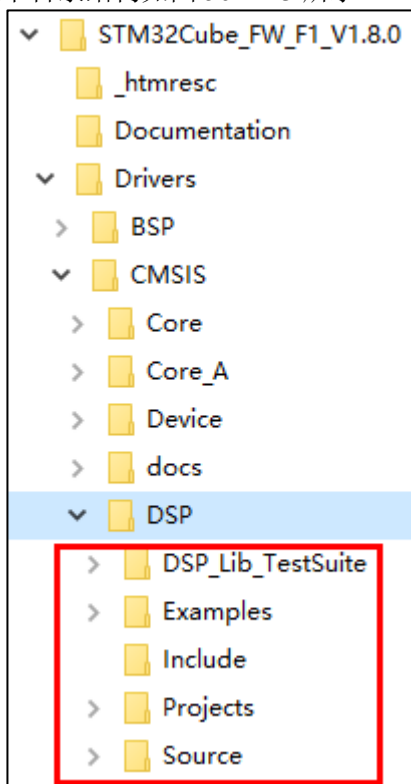


图 56.1.1.3 DSP 目录结构

DSP 源码包的 Source 文件夹是所有 DSP 库的源码，Examples 文件夹是相对应的一些测试实例。这些测试实例都是带 main 函数的，也就是拿到工程中可以直接使用。接下来我们一一讲解一下 Source 源码文件夹下面的子文件夹包含的 DSP 库的功能。

- **BasicMathFunctions**

基本数学函数：提供浮点数的各种基本运算函数，如向量加减乘除等运算。

- **CommonTables**

arm_common_tables.c 文件提供位翻转或相关参数表。arm_const_structs.c 文件提供一些常用的常量结构体，方便用户使用。

- **ComplexMathFunctions**

复数学功能，如向量处理，求模运算。

- **ControllerFunctions**

控制功能函数。包括正弦余弦，PID 电机控制，矢量 Clarke 变换，矢量 Clarke 逆变换。

- **FastMathFunctions**

快速数学功能函数。提供了一种快速的近似正弦，余弦和平方根等相比 CMSIS 计算库要快的数学函数。

- **FilteringFunctions**

滤波函数功能，主要为 FIR 和 LMS（最小均方根）等滤波函数。

- **MatrixFunctions**

矩阵处理函数。包括矩阵加法、矩阵初始化、矩阵反、矩阵乘法、矩阵规模、矩阵减法、矩阵转置等函数。

- **StatisticsFunctions**

统计功能函数。如求平均值、最大值、最小值、计算均方根 RMS、计算方差/标准差等。

- **SupportFunctions**

支持功能函数，如数据拷贝，Q 格式和浮点格式相互转换，Q 任意格式相互转换。

- **TransformFunctions**

变换功能。包括复数 FFT（CFFT）/复数 FFT 逆运算（CIFFT）、实数 FFT（RFFT）/实数 FFT 逆运算（RIFFT）、和 DCT（离散余弦变换）和配套的初始化函数。

所有这些 DSP 库代码合在一起是比较多的，因此，ST 为我们提了 .lib 格式的文件，方便使用。这些 .lib 文件就是由 Source 文件夹下的源码编译生成的，如果想看某个函数的源码，大家可以在 Source 文件夹下面查找。lib 格式文件 HAL 库包路径：Drivers→CMSIS→Lib→ARM，针对 M3 的总共有 2 个 .lib 文件，如下：

① arm_cortexM3b_math.lib (Cortex-M3 大端模式)

② arm_cortexM3l_math.lib (Cortex-M3 小端模式)

我们得根据所用 MCU 内核类型以及端模式来选择符合要求的 .lib 文件，本章我们所用 STM32F1 属于 CortexM3 内核，小端模式，应选择：

arm_cortexM3l_math.lib (Cortex-M7 小端模式)。

对于 DSP 的子文件夹 Examples 下面存放的文件，是 ST 官方提供的一些 DSP 测试代码，提供简短的测试程序，方便上手，有兴趣的朋友可以根据需要自行测试。

56.1.2 DSP 库运行环境搭建

本节我们将讲解怎么搭建 DSP 库运行环境，只要运行环境搭建好了，使用 DSP 库里面的函数来做相关处理就非常简单了。本节，我们将基于定时器实验，搭建 DSP 运行环境。

在 MDK 里面搭建 STM32F103 的 DSP 运行环境(使用 .lib 方式)是很简单的，分为 3 个步骤：

1、添加文件。

首先，我们在实验 32_1 DSP BasicMath 测试实验\Drivers\CMSIS 目录下新建：DSP 和 Lib 文件夹，然后把官方的相应文件拉到我们的工程的相应位置：arm_cortexM3l_math.lib 和相关头文件，再把没用到的文件删除，如图 56.1.2.1 和图 56.1.2.2 所示：

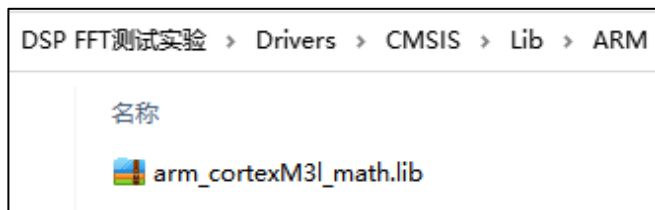


图 56.1.2.1 LIB 文件夹添加文件

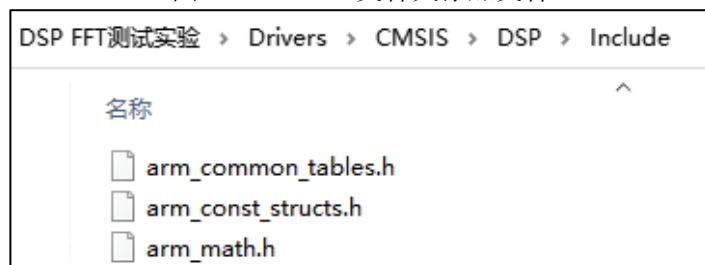


图 56.1.2.2 DSP 文件夹添加文件

这个步骤具体请参考工程源码文件, arm_cortexM3l_math.lib 文件在 56.1.1 节已经介绍过了。Include 文件夹里面包含了我们可能要用到的相关头文件, 所以要添加到工程中。

然后, 打开工程, 新建 Drivers/CMSIS 分组, 并将 arm_cortexM3l_math.lib 添加到工程里面, 如图 56.1.2.3 所示:

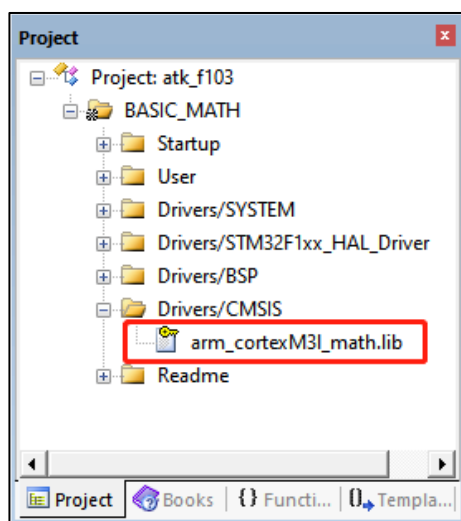


图 56.1.2.3 添加.lib 文件

这样, 添加文件就结束了 (就添加了一个.lib 文件)。

2、添加头文件包含路径

添加好.lib 文件后, 我们要添加头文件包含路径, 这个我们工程都统一添加好了, 所以不用额外再操作, 这里是提醒一下大家要添加头文件包含路径, 如图 56.1.2.4 所示:

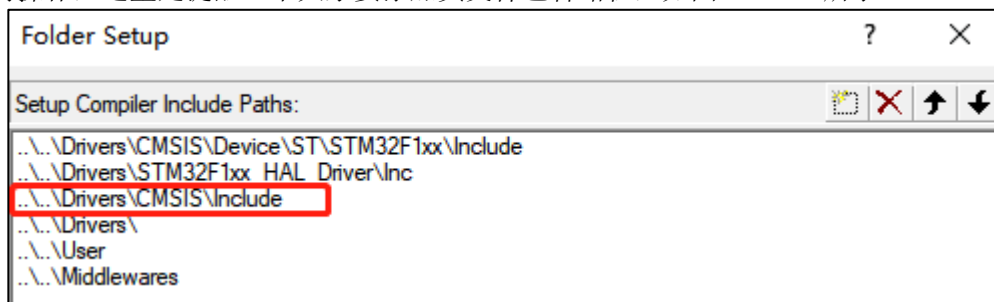



图 56.1.2.4 添加相关头文件包含路径

3、添加全局宏定义

最后, 为了使用 DSP 库的所有功能, 我们还需要添加几个全局宏定义:

- 1, ARM_MATH_CM3
- 2, __CC_ARM
- 3, ARM_MATH_MATRIX_CHECK
- 4, ARM_MATH_ROUNDING

添加方法：点击  → C/C++ 选项卡，然后在 Define 里面进行设置，如图 56.1.2.5 所示：

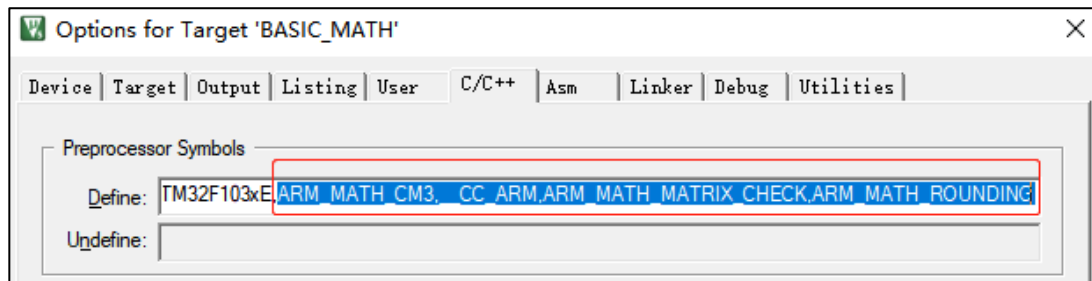


图 56.1.2.5 DSP 库支持全局宏定义设置

在这里，所有的宏之间都需要用英文格式下“,”进行隔开。在 Define 处要输入的所有宏为：STM32F103xE,USE_HAL_DRIVER,ARM_MATH_CM3,__CC_ARM,ARM_MATH_MATRIX_CHECK,ARM_MATH_ROUNDING 共 6 个。

至此，STM32F103 的 DSP 库运行环境就搭建完成了。

特别注意，为了方便调试，本章例程我们将 MDK 的优化设置为 -O0 优化，以得到最好的调试效果。

56.2 硬件设计

1. 例程功能

本例程包含 2 个源码：《实验 32_1 DSP BasicMath 测试实验》和《实验 32_2 DSP FFT 测试实验》，他们除了 main.c 里面内容不一样外，其他源码完全一模一样（包括 MDK 配置）。

《实验 32_1 DSP BasicMath 测试实验》功能简介：测试 STM32F103 的 DSP 库基础数学函数：arm_cos_f32 和 arm_sin_f32 和标准库基础数学函数：cosf 和 sinf 的速度差别，并在 LCD 屏幕上面显示两者计算所用时间。LED0 闪烁，提示程序运行。

《实验 32_2 DSP FFT 测试实验》功能简介：测试 STM32F103 的 DSP 库的 FFT 函数，程序运行后，自动生成 1024 点测试序列，然后，每当 KEY0 按下后，调用 DSP 库的 FFT 算法（基 4 法）执行 FFT 运算，在 LCD 屏幕上面显示运算时间，同时将 FFT 结果输出到串口。LED0 闪烁，提示程序运行。

2. 硬件资源

- 1) LED 灯
LED0 – PB5
- 2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
- 4) 独立按键
KEY0 – PE4
- 5) 定时器 6

56.3 程序设计

本章代码，分成两个工程：1，实验 32_1 DSP BasicMath 测试；2，实验 32_2 DSP FFT 测试，接下来我们分别介绍。

56.3.1 DSP BasicMath 测试

这是我们使用 STM32F103 的 DSP 库进行基础数学函数测试的一个例程。使用大家耳熟能

详的公式进行计算：

$$\sin(x)^2 + \cos(x)^2 = 1$$

这里我们用到的就是 sin 和 cos 函数，不过实现方式不同。MDK 的标准库（math.h）提供我们：sin、cos、sinf 和 cosf 等 4 个函数，带 f 的表示单精度浮点型运算，即 float 型，而不带 f 的表示双精度浮点型，即 double。

STM32F103 的 DSP 库，则提供我们另外两个函数：arm_sin_f32 和 arm_cos_f32（**注意：需要添加：arm_math.h 头文件才可使用**），这两个函数也是单精度浮点型的，用法同 sinf 和 cosf 一模一样。

本例程就是测试：arm_sin_f32&arm_cos_f32 同 sinf&cosf 的速度差别。

因为 56.1.2 节已经搭建好 DSP 库运行环境了，所以我们这里直接只需要修改 main.c 里面的代码即可，main.c 代码如下：

```
/**
 * @brief      sin cos 测试
 * @param      angle : 起始角度
 * @param      times : 运算次数
 * @param      mode : 是否使用 DSP 库
 * @arg        0 , 不使用 DSP 库;
 * @arg        1 , 使用 DSP 库;
 *
 * @retval     无
 */
uint8_t sin_cos_test(float angle, uint32_t times, uint8_t mode)
{
    float sinx, cosx;
    float result;
    uint32_t i = 0;

    if (mode == 0)
    {
        for (i = 0; i < times; i++)
        {
            cosx = cosf(angle);          /* 不使用 DSP 优化的 sin, cos 函数 */
            sinx = sinf(angle);
            result = sinx * sinx + cosx * cosx; /* 计算结果应该等于 1 */
            result = fabsf(result - 1.0f);      /* 对比与 1 的差值 */

            if (result > DELTA) return 0xFF;    /* 判断失败 */

            angle += 0.001f;    /* 角度自增 */
        }
    }
    else
    {
        for (i = 0; i < times; i++)
        {
            cosx = arm_cos_f32(angle);          /* 使用 DSP 优化的 sin, cos 函数 */
            sinx = arm_sin_f32(angle);
            result = sinx * sinx + cosx * cosx; /* 计算结果应该等于 1 */
            result = fabsf(result - 1.0f);      /* 对比与 1 的差值 */

            if (result > DELTA) return 0xFF;    /* 判断失败 */

            angle += 0.001f;    /* 角度自增 */
        }
    }

    return 0;          /* 任务完成 */
}
```

```
uint8_t g_timeout;

int main(void)
{
    float time;
    char buf[50];
    uint8_t res;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    btim_timx_int_init(65535, 7200 - 1); /* 10Khz 计数频率,最大计时 6.5 秒超出 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DSP BasicMath TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, " No DSP runtime:", RED); /* 显示提示信息 */
    lcd_show_string(30, 150, 200, 16, 16, "Use DSP runtime:", RED); /* 显示提示信息 */

    while (1)
    {
        /* 不使用 DSP 优化 */
        __HAL_TIM_SET_COUNTER(&timx_handler, 0); /* 重设 TIM3 定时器的计数器值 */
        g_timeout = 0;
        res = sin_cos_test(PI / 6, 10000, 0);
        time = __HAL_TIM_GET_COUNTER(&timx_handler) +
            (uint32_t)g_timeout * 65536;
        sprintf(buf, "%0.1fms\r\n", time / 10);

        if (res == 0)
        {
            lcd_show_string(30 + 16 * 8, 110, 100, 16, 16, buf, BLUE); /* 显示运行时间 */
        }
        else
        { /* 显示当前运行情况 */
            lcd_show_string(30 + 16 * 8, 110, 100, 16, 16, "error! ", BLUE);
        }
        /* 使用 DSP 优化 */
        __HAL_TIM_SET_COUNTER(&timx_handler, 0); /* 重设 TIM3 定时器的计数器值 */
        g_timeout = 0;
        res = sin_cos_test(PI / 6, 10000, 1);
        time = __HAL_TIM_GET_COUNTER(&timx_handler) +
            (uint32_t)g_timeout * 65536;
        sprintf(buf, "%0.1fms\r\n", time / 10);
        if (res == 0)
        {
            lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, buf, BLUE); /* 显示运行时间 */
        }
        else
        { /* 显示错误 */
            lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, "error! ", BLUE);
        }
        LED0_TOGGLE();
    }
}
```

这里包括 2 个函数：sin_cos_test 和 main 函数，sin_cos_test 函数用于根据给定参数，执行 $\sin(x)^2 + \cos(x)^2 = 1$ 的计算。计算完后，计算结果同给定的误差值（DELTA）对比，如果不大于误差值，则认为计算成功，否则计算失败。该函数可以根据给定的模式参数(mode)来决定使用哪

个基础数学函数执行运算，从而得出对比。

main 函数则比较简单，这里我们通过定时器 6 来统计 sin_cos_test 运行时间，从而得出对比数据。主循环里面，每次循环都会两次调用 sin_cos_test 函数，首先采用不使用 DSP 库方式计算，然后采用使用 DSP 库方式计算，并得出两次计算的时间，显示在 LCD 上面。

DSP 基础数学函数测试的程序设计就讲解到这里。

56.3.2 DSP FFT 测试

这是我们使用 STM32F103 的 DSP 库进行 FFT 函数测试的一个例程。

首先，我们简单介绍下 FFT：FFT 即快速傅里叶变换，可以将一个时域信号变换到频域。因为有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了，这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。简而言之，FFT 就是将一个信号从时域变换到频域方便我们分析处理。

在实际应用中，一般的处理过程是先对一个信号在时域进行采集，比如我们通过 ADC，按照一定大小采样频率 F 去采集信号，采集 N 个点，那么通过对这 N 个点进行 FFT 运算，就可以得到这个信号的频谱特性。

这里还涉及到一个采样定理的概念：在进行模拟/数字信号的转换过程中，当采样频率 F 大于信号中最高频率 f_{\max} 的 2 倍时 ($F > 2 * f_{\max}$)，采样之后的数字信号完整地保留了原始信号中的信息，采样定理又称奈奎斯特定理。举个简单的例子：比如我们正常人发声，频率范围一般在 8KHz 以内，那么我们要通过采样之后的数据来恢复声音，我们的采样频率必须为 8KHz 的 2 倍以上，也就是必须大于 16KHz 才行。

模拟信号经过 ADC 采样之后，就变成了数字信号，采样得到的数字信号，就可以做 FFT 变换了。N 个采样点数据，在经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F，对一个信号采样，采样点数为 N，那么 FFT 之后结果就是一个 N 点的复数，每一个点就对应着一个频率点（以基波频率为单位递增），这个点的模值（ $\sqrt{\text{实部}^2 + \text{虚部}^2}$ ）就是该频点频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 N/2 倍，而第一个点就是直流分量，它的模值就是直流分量的 N 倍。

这里还有个基波频率，也叫频率分辨率的概念，就是如果我们按照 F 的采样频率去采集一个信号，一共采集 N 个点，那么基波频率（频率分辨率）就是 $f_k = F/N$ 。这样，第 n 个点对应信号频率为： $F * (n-1)/N$ ；其中 $n \geq 1$ ，当 $n=1$ 时为直流分量。

关于 FFT 我们就介绍到这。

如果我们要自己实现 FFT 算法，对于不懂数字信号处理的朋友来说，是比较难的，不过，ST 提供的 STM32F103 DSP 库里面就有 FFT 函数给我们调用，因此我们只需要知道如何使用这些函数，就可以迅速的完成 FFT 计算，而不需要自己学习数字信号处理，去编写代码了，大大方便了我们的开发。

STM32F103 的 DSP 库里面，提供了定点和浮点 FFT 实现方式，并且有基 4 的也有基 2 的，大家可以根据需要自由选择实现方式。注意：对于基 4 的 FFT 输入点数必须是 4^n ，而基 2 的 FFT 输入点数则必须是 2^n ，并且基 4 的 FFT 算法要比基 2 的快。

本章我们将采用 DSP 库里面的基 4 浮点 FFT 算法来实现 FFT 变换，并计算每个点的模值，所用到的函数有：

```
/* Deprecated */
arm_status arm_cfft_radix4_init_f32(arm_cfft_radix4_instance_f32 * S,
                                     uint16_t fftLen, uint8_t ifftFlag, uint8_t bitReverseFlag);

/* Deprecated */
void arm_cfft_radix4_f32(const arm_cfft_radix4_instance_f32 * S,
                        float32_t * pSrc);

/**
 * @brief Floating-point complex magnitude
```

```

* @param[in] pSrc      points to the complex input vector
* @param[out] pDst      points to the real output vector
* @param[in] numSamples number of complex samples in the input vector
*/

```

```
void arm_cmplx_mag_f32(float32_t * pSrc, float32_t * pDst, uint32_t numSamples);
```

第一个函数 `arm_cfft_radix4_init_f32`，用于初始化 FFT 运算相关参数，其中：`fftLen` 用于指定 FFT 长度（16/64/256/1024/4096），本章设置为 1024；`ifftFlag` 用于指定是傅里叶变换(0)还是反傅里叶变换(1)，本章设置为 0；`bitReverseFlag` 用于设置是否按位取反，本章设置为 1；最后，所有这些参数存储在一个 `arm_cfft_radix4_instance_f32` 结构体指针 `S` 里面。

第二个函数 `arm_cfft_radix4_f32` 就是执行基 4 浮点 FFT 运算的，`pSrc` 传入采集到的输入信号数据（实部+虚部形式），同时 FFT 变换后的数据，也按顺序存放在 `pSrc` 里面，`pSrc` 必须大于等于 2 倍 `fftLen` 长度。另外，`S` 结构体指针参数是先由 `arm_cfft_radix4_init_f32` 函数设置好，然后传入该函数的。

第三个函数 `arm_cmplx_mag_f32` 用于计算复数模值，可以对 FFT 变换后的结果数据，执行取模操作。`pSrc` 为复数输入数组（大小为 `2*numSamples`）指针，指向 FFT 变换后的结果；`pDst` 为输出数组（大小为 `numSamples`）指针，存储取模后的值；`numSamples` 就是总共有多少个数据需要取模。

通过这三个函数，我们便可以完成 FFT 计算，并取模值。本节例程（实验 32_2 DSP FFT 测试实验）同样是在 56.1.2 节已经搭建好 DSP 库运行环境上面修改代码，只需要修改 `main.c` 里面的代码即可，本例程 `main.c` 代码如下：

```

/* FFT 长度，默认是 1024 点 FFT
 * 可选范围为：16, 64, 256, 1024.
 */
#define FFT_LENGTH      1024

float fft_inputbuf[FFT_LENGTH * 2];    /* FFT 输入数组 */
float fft_outputbuf[FFT_LENGTH];        /* FFT 输出数组 */

uint8_t g_timeout;

int main(void)
{
    float time;
    char buf[50];
    arm_cfft_radix4_instance_f32 scfft;
    uint8_t key, t = 0;
    uint16_t i;

    HAL_Init();                          /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);                       /* 延时初始化 */
    usart_init(115200);                   /* 串口初始化为 115200 */
    led_init();                           /* 初始化 LED */
    lcd_init();                           /* 初始化 LCD */
    key_init();                           /* 初始化按键 */
    btim_timx_int_init(65535, 7200 - 1); /* 10Khz 计数频率,最大计时 6.5 秒超出 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32F103", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DSP FFT TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "2020/4/5", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY0:Run FFT", RED); /* 显示提示信息 */
    lcd_show_string(30, 160, 200, 16, 16, "FFT runtime:", RED); /* 显示提示信息 */

    /* 初始化 scfft 结构体，设定 FFT 相关参数 */
    arm_cfft_radix4_init_f32(&scfft, FFT_LENGTH, 0, 1);

    while (1)
    {

```

```

key = key_scan(0);

if (key == KEY0_PRES)
{
    for (i = 0; i < FFT_LENGTH; i++)    /* 生成信号序列 */
    {
        /* 生成输入信号实部 */
        fft_inputbuf[2 * i] = 100 +
            10 * arm_sin_f32(2 * PI * i / FFT_LENGTH) +
            30 * arm_sin_f32(2 * PI * i * 4 / FFT_LENGTH) +
            50 * arm_cos_f32(2 * PI * i * 8 / FFT_LENGTH);
        fft_inputbuf[2 * i + 1] = 0; /* 虚部全部为 0 */
    }

    BTIM_TIMX_INT->CNT = 0;; /* 重设 BTIM_TIMX_INT 定时器的计数器值 */
    g_timeout = 0;

    arm_cfft_radix4_f32(&scfft, fft_inputbuf);    /* FFT 计算 (基 4) */

    time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536; /* 计算所用时间 */
    sprintf((char *)buf, "%0.3fms\r\n", time / 1000);
    /* 显示运行时间 */
    lcd_show_string(30 + 12 * 8, 160, 100, 16, 16, buf, BLUE);
    /* 把运算结果复数求模得幅值 */
    arm_cmplx_mag_f32(fft_inputbuf, fft_outputbuf, FFT_LENGTH);

    printf("\r\n%d point FFT runtime:%0.3fms\r\n", FFT_LENGTH, time/1000);
    printf("FFT Result:\r\n");

    for (i = 0; i < FFT_LENGTH; i++)
    {
        printf("fft_outputbuf[%d]:%f\r\n", i, fft_outputbuf[i]);
    }
}
else
{
    delay_ms(10);
}

t++;

if ((t % 20) == 0)
{
    LED0_TOGGLE();
}
}
}

```

以上代码只有一个 main 函数,里面通过我们前面介绍的三个函数:arm_cfft_radix4_init_f32、arm_cfft_radix4_f32 和 arm_cmplx_mag_f32 来执行 FFT 变换并取模值。每当按下 KEY0 就会重新生成一个输入信号序列,并执行一次 FFT 计算,将 arm_cfft_radix4_f32 所用时间统计出来,显示在 LCD 屏幕上面,同时将取模后的模值通过串口打印出来。

这里,我们在程序上生成了一个输入信号序列用于测试,输入信号序列表达式:

```

/* 生成输入信号实部 */
fft_inputbuf[2 * i] = 100 +
    10 * arm_sin_f32(2 * PI * i / FFT_LENGTH) +
    30 * arm_sin_f32(2 * PI * i * 4 / FFT_LENGTH) +
    50 * arm_cos_f32(2 * PI * i * 8 / FFT_LENGTH);

```

通过该表达式我们可知,信号的直流分量为 100,外加 2 个正弦信号和一个余弦信号,其幅值分别为 10、30 和 50。

56.4 下载验证

将实验 32_1 DSP BasicMath 测试实验的程序下载到开发板后，可以在屏幕看到两种实现方式的速度差别，如图 56.4.1 所示：

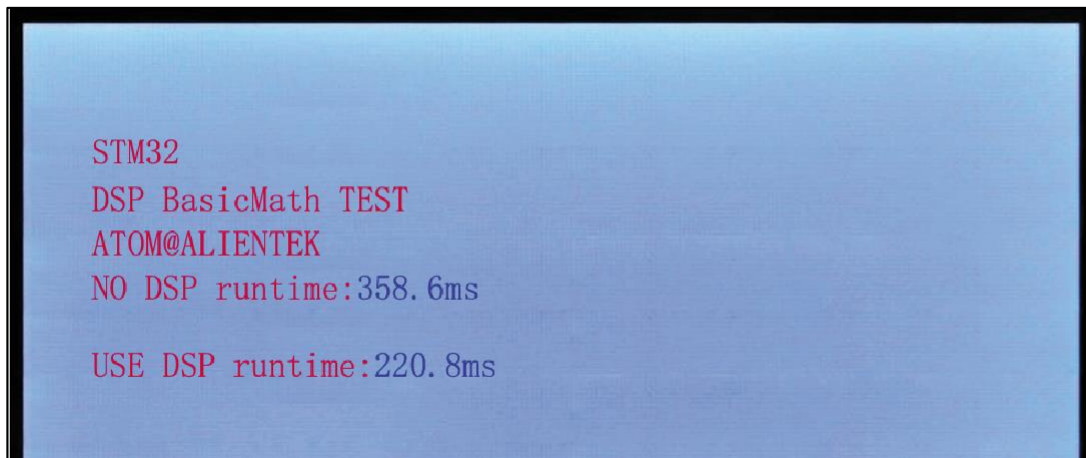


图 56.4.1 使用 DSP 库和不使用 DSP 库的基础数学函数速度对比

从图中可以看出，使用 DSP 库的基础数学函数计算所用时间比不使用 DSP 库的短，使用 STM32F103 的 DSP 库，速度上面比传统的实现方式提升了约 23%。

对于实验 32_2 DSP FFT 测试实验，下载后，屏幕显示提示信息，然后我们按下 KEY0 就可以看到 FFT 运算所耗时间，如图 56.4.2 所示：

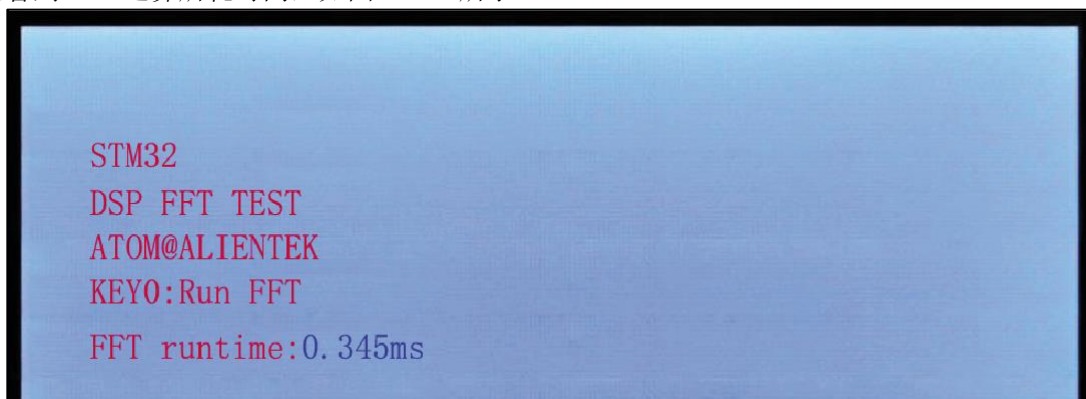


图 56.4.2 FFT 测试界面

可以看到，STM32F103 采用基 4 法计算 1024 个浮点数的 FFT，仅用了 0.345ms，速度是非常的快。同时，可以在串口看到 FFT 变换取模后的各频点模值，如图 56.4.3 所示：

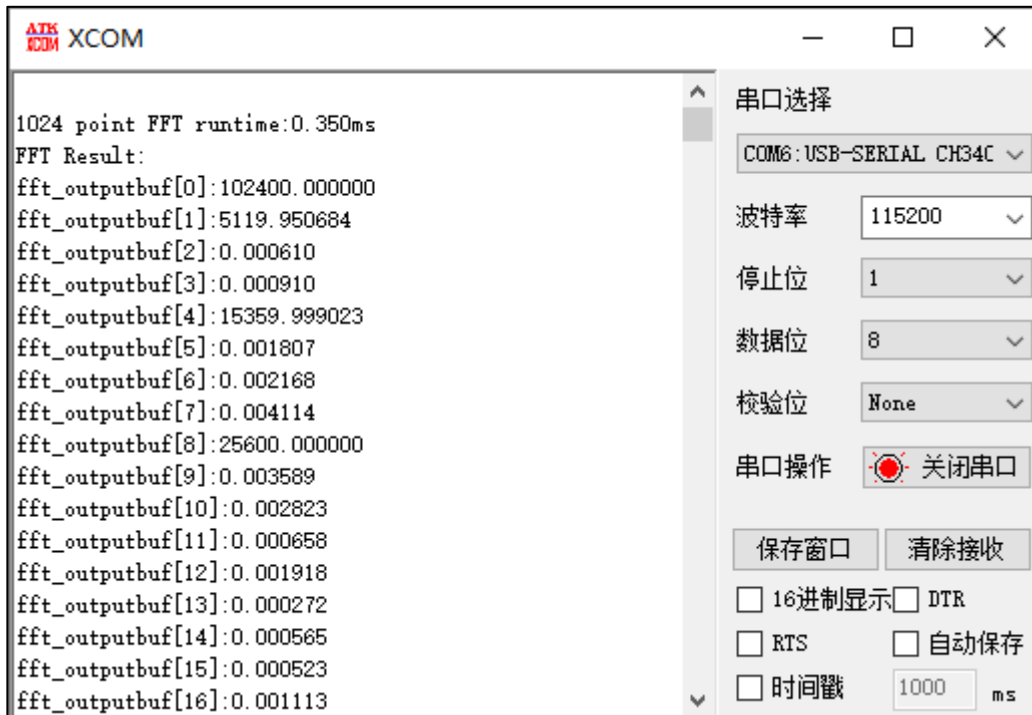


图 56.4.3 FFT 变换后个频点模值

查看所有数据，会发现：第 0、1、4、8、1016、1020、1023 这 7 个点的值比较大，其他点的值都很小，接下来我们就简单分析一下这些数据。

由于 FFT 变换后的结果具有对称性，所以，实际上有用的数据，只有前半部分，后半部分和前半部分是对称关系，比如 1 和 1023，4 和 1020，8 和 1016 等，就是对称关系，因此我们只需要分析前半部分数据即可。这样，就只有第 0、1、4、8 这四个点，比较大，重点分析。

假设我们采样频率为 1024Hz，那么总共采集 1024 个点，频率分辨率就是 1Hz，对应到频谱上面，两个点之间的间隔就是 1Hz。因此，上面我们生成的三个叠加信号： $10 \cdot \sin(2 \cdot \pi \cdot i / 1024) + 30 \cdot \sin(2 \cdot \pi \cdot i \cdot 4 / 1024) + 50 \cdot \cos(2 \cdot \pi \cdot i \cdot 8 / 1024)$ ，频率分别是：1Hz、4Hz 和 8Hz。

对于上述 4 个值比较大的点，结合 56.3.1 节的知识，很容易分析得出：第 0 点，即直流分量，其 FFT 变换后的模值应该是原始信号幅值的 N 倍， $N=1024$ ，所以值是 $100 \cdot 1024=102400$ ，与理论完全一样，然后其他点，模值应该是原始信号幅值的 N/2 倍，即 $10 \cdot 512$ 、 $30 \cdot 512$ 、 $50 \cdot 512$ ，而我们计算结果是：5119.950684、15359.999023、256000，同理论值非常接近。

DSP 测试实验，我们就讲解到这里，DSP 库的其他测试实例，大家可以自行研究下，我们这里就不再介绍了。

第五十七章 手写识别实验

本章，我们将利用正点原子提供的手写识别库，实现一个简单得数字字母手写识别。本章分为如下几个小节：

- 57.1 手写识别简介
- 57.2 硬件设计
- 57.3 程序设计
- 57.4 下载验证

57.1 手写识别简介

手写识别，是指对在手写设备上书写时产生的有序轨迹信息进行识别的过程，是人际交互最自然、最方便的手段之一。随着智能手机和平板电脑等移动设备的普及，手写识别的应用也被越来越多的设备采用。

手写识别能够使用户按照最自然、最方便的输入方式进行文字输入，易学易用，可取代键盘或者鼠标。用于手写输入的设备有许多种，比如电磁感应手写板、压感式手写板、触摸屏、触控板、超声波笔等。本实验通过使用 STM32 板子自带得 TFTLCD 触摸屏(2.8/3.5/4.3/7 寸)，可以用来作为手写识别的输入设备。接下来，我们将给大家简单介绍下手写识别的实现过程。

手写识别与其他识别系统如语音识别、图像识别一样分为两个过程：训练学习过程；识别过程。如图 57.1.1 所示：

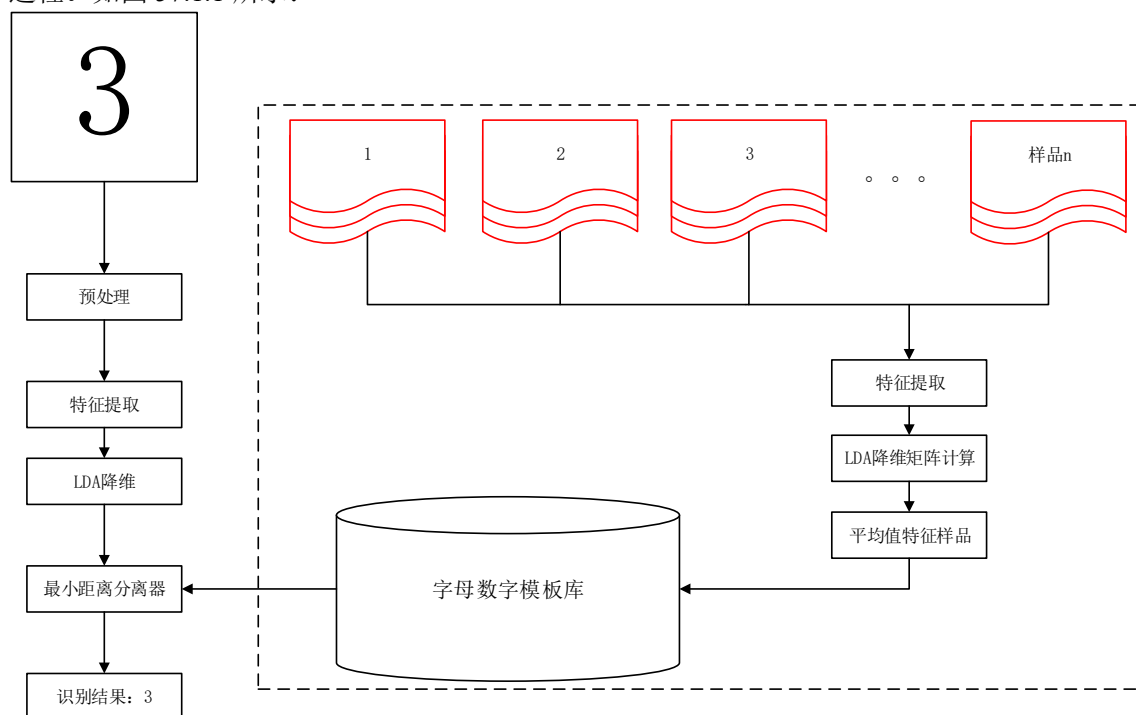


图 57.1.1 字母数字识别系统示意图

上图中虚线部分分为训练学习过程，该过程首先需要使用设备采集大量数据样本，样本类别数目为 0~9, a~z, A~Z 总过 62 类，每个类别 5~10 个样本不等（样本越多识别率就越高）。对这些样本进行传统的八方向特征提取，提取后特征维数为 512 维，这对于 STM32 来说，计算量和模板库的存储量都是难以接收，所以需要运行一些方法进行降维，这里采用 LDA 线性判决分析的方法进行降维。所谓的线性判决分析，即是假设所有样本服从高斯分布（正态分布）对样本进行低维投影，以达到各个样本间的距离最大化。关于 LDA（线性判别分析）的更多知识可以自行阅读（<http://wenku.baidu.com/view/f05c731452d380eb62946d39.html>）等参考文档。这里将维度降到 64 维度，然后针对各个样本类别进行平均计算的到该类别的样本模板。

而对于识别过程，首先得到触屏输入的有序轨迹，然后进行一些预处理，预处理主要包括重采样，归一化处理。重采样主要是因为不同的输入设备不同的输入处理方式产生的有序轨迹序列有所不同。为了达到更好的识别结果我们需要对训练样本和识别输入的样本进行重采样处理，这里主要应用隔点重采样的方法对输入序列进行重采样；而归一化就是因为不同的书写风格采用分辨率的差异会导致字体大小不同，因此需要对输入轨迹进行归一化。这里把样本进行线性缩放的方法归一化为 64×64 像素。

接下来进行同样的八方向特征提取操作。所谓的八方向特征就是首先将经过预处理后的 64×64 输入进行切分成 8×8 的小方格，每个方格 8×8 个像素；然后对每个 8×8 个小格进行各个方向的点数统计。如某个方格内一共有 10 个点，其中八个方向的点分别为：1、3、5、2、3、4、3、2，那么这个格子得到的八个特征向量为 $[0.1, 0.3, 0.5, 0.2, 0.3, 0.4, 0.3, 0.2]$ 。总共 64 个格子，于是一个样本最终能得到 $64 \times 8 = 512$ 维特征，更多八方向特征提取可以参考一下两个文档：

- 1, <http://wenku.baidu.com/view/d37e5a49e518964bcf847ca5.html>;
- 2, <http://wenku.baidu.com/view/3e7506254b35eefdc8d333a1.html>;

由于训练过程进行了 LDA 降维计算，所以识别过程同样需要对应的 LDA 降维过程得到最终的 64 维特征。这个计算过程就是在训练模板的过程中可以运算得到一个 512×64 维的矩阵，那么我们通过矩阵乘运算可以得到 64 维的最终特征值。

$$[d_1, d_2, \dots, d_{512}] \times \begin{bmatrix} l & \dots & l \\ \vdots & \ddots & \vdots \\ l & \dots & l \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_{64} \end{bmatrix}$$

最后将这 64 维特征分别与模板中的特征进行求距离运算。得到最小的距离为该输入的最佳识别结果输出。

$$output = \arg \min_{i \in [1, 62]} \{(f_1 - f_1^i)^2 + (f_2 - f_2^i)^2 + \dots + (f_{64} - f_{64}^i)^2\}$$

关于手写识别原理，我们就介绍到这里。如果想自己实现手写识别，那得花很多时间学习和研究，但是如果只是应用的话，那么就只需要知道怎么用就 OK 了，相对来说，简单得多。

正点原子提供了一个数字字母识别库，我们不需要关心手写识别是如何实现的，只需要知道这个库怎么用，就能实现手写识别。正点原子提供的手写识别库由 4 个文件组成：ATKNCR_M_V2.0.lib、ATKNCR_M_V2.0.lib、atk_ncr.c 和 atk_ncr.h。

ATKNCR_M_V2.0.lib 和 ATKNCR_M_V2.0.lib 是两个识别用的库文件（两个版本），使用的时候，选择其中之一即可。ATKNCR_M_V2.0.lib 用于使用内存管理的情况，用户必须自己实现 `alientek_ncr_malloc` 和 `alientek_ncr_free` 两个函数。而 ATKNCR_N_V2.0.lib 用于不使用内存管理的情况，通过全局变量来定义缓存区，缓存区需要提供至少 3K 左右的 RAM。大家根据自己的需要，选择不同的版本即可。正点原子手写识别库资源需求：FLASH 在 52K 左右，RAM 在 6K 左右。

57.2 硬件设计

1. 例程功能

开机的时候先初始化手写识别器，然后检测字库，之后进入等待输入状态。此时，我们在手写区写数字/字符，在每次写入结束后，自动进入识别状态，进行识别，然后将识别结果输出在 LCD 模块上，同时打印到串口。通过按 KEY0 可以进行模式切换（4 种模式都可以测试），通过按 KEY_UP，可以进入触摸屏校准（仅电阻屏需要校准，如果发现触摸屏不准，请执行此操作）。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY1 - PE3

- 3) 串口 1
- 4) ALIENTEK 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 5) NOR FLASH: 通过 SPI2 连接

57.3 程序设计

57.3.1 程序流程图

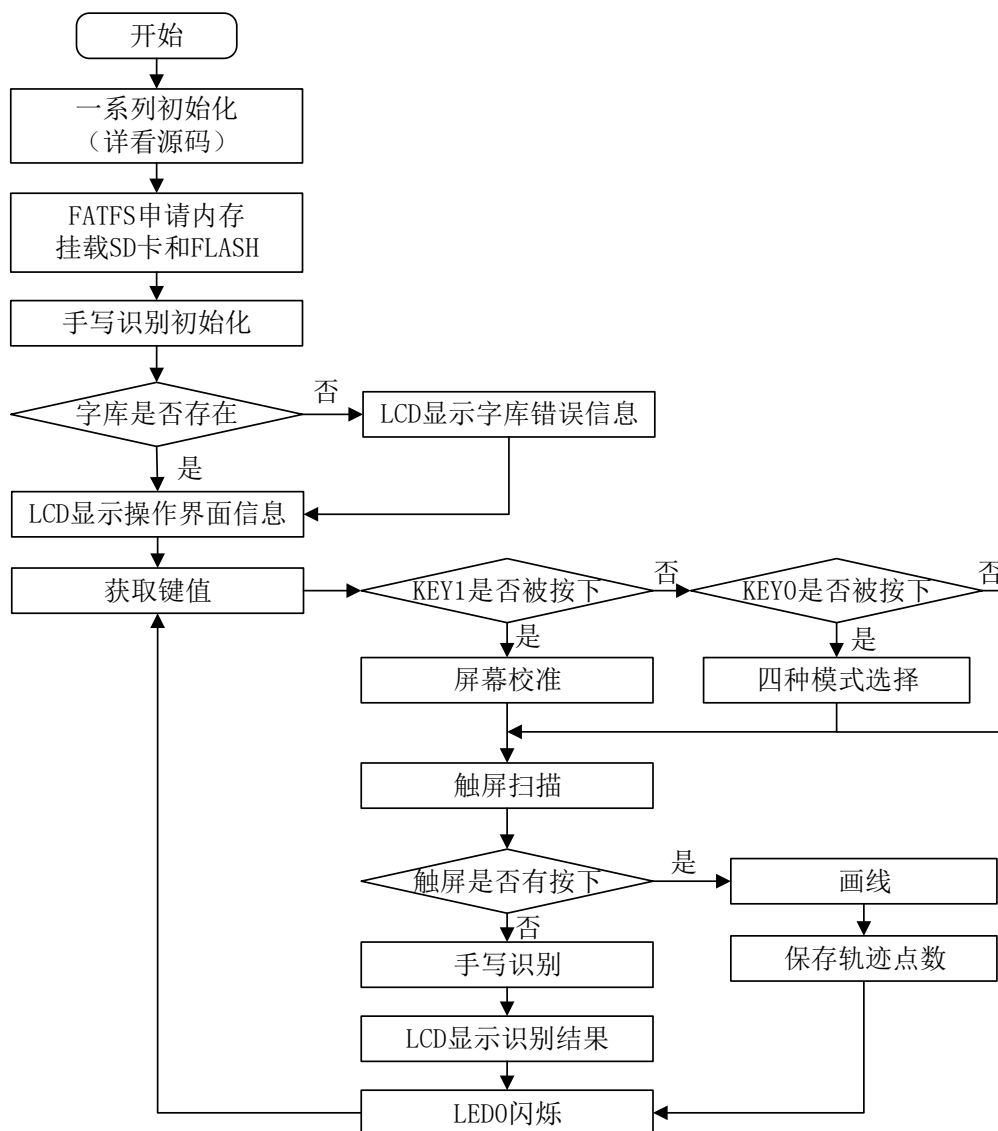


图 57.3.1 手写识别实验程序流程图

手写识别我们主要通过配合 LCD 屏的触摸识别功能, 将触摸信息传给解码库进行识别, 由于解码库存储触摸点需要内存, 所以注意保证内存够用。通过设定解码的点数就可以通过解码库封闭好的接口实现数据识别了。

57.3.2 程序解析

1. ATKNCR 代码

手写识别代码我们在前面也提到了有四种, 两个字母数字识别库至于用了哪个识别库, 我们工程中使用的是 ATKNCR_M_V2.0.lib。首先我们先看一下 `atk_nrc.h` 头文件中比较重要部分, 其代码如下:

```

/* 输入轨迹坐标类型 */
_packed typedef struct _atk_ncr_point
{
    short x;          /* x 轴坐标 */
    short y;          /* y 轴坐标 */
}atk_ncr_point;

/* 外部调用函数
 * 初始化识别器
 * 返回值 : 0, 初始化成功
 *         1, 初始化失败
 */
unsigned char alientek_ncr_init(void);

/* 停止识别器 */
void alientek_ncr_stop(void);

/* 识别器识别
 * track   : 输入点阵集合
 * potnum  : 输入点阵的点数, 就是 track 的大小
 * charnum : 期望输出的结果数, 就是你希望输出多少个匹配结果
 * mode    : 识别模式
 *          1, 仅识别数字
 *          2, 仅识别大写字母
 *          3, 仅识别小写字母
 *          4, 混合识别(全部识别)
 *
 * result  : 结果缓存区(至少为:charnum+1 个字节)
 */
void alientek_ncr(atk_ncr_point * track, int potnum, int charnum,
unsigned char mode, char*result);

```

在上面的代码中, 我们定义了一些外部接口函数以及轨迹结构体等。

`alientek_ncr_init` 函数用于初始化识别器, 该函数在 `.lib` 文件实现, 在识别开始之前, 我们应该调用该函数。

`alientek_ncr_stop` 函数用于停止识别器, 在识别完成之后(不需要再识别), 我们调用该函数, 如果一直处于识别状态, 则没必要调用。该函数也是在 `.lib` 文件实现。

`alientek_ncr` 函数就是识别函数了。它有 5 个参数, 第一个参数 `track`, 为输入轨迹点的坐标集(最好 200 以内); 第二个参数 `potnum`, 为坐标集点坐标的个数; 第三个参数 `charnum`, 为期望输出的结果数, 即希望输出多少个匹配结果, 识别器按匹配程度排序输出(最佳匹配排第一); 第四个参数 `mode`, 该函数用于设置模式, 识别器总共支持 4 种模式:

- 1, 仅识别数字
- 2, 仅识别大写字母
- 3, 仅识别小写字母
- 4, 混合识别(全部识别)

最后一个参数是 `result`, 用来输出结果, 注意这个结果是 ASCII 码格式的。

下面我们直接来介绍 `atk_ncr.c` 中内存管理部分, 其代码如下:

```

/**
 * @brief      内存设置函数
 * @param      *p      : 内存首地址
 * @param      c      : 要设置的值
 * @param      len     : 需要设置的内存大小(字节为单位)
 * @retval     无
 */
void alientek_ncr_memset(char *p, char c, unsigned long len)
{
    my_mem_set((uint8_t*)p, (uint8_t)c, (uint32_t)len);
}

```

```

/**
 * @brief      分配内存
 * @param      size : 要分配的内存大小(字节)
 * @retval     分配到的内存首地址.
 */
void *alientek_ncr_malloc(unsigned int size)
{
    return mymalloc(SRAMIN, size);
}

/**
 * @brief      释放内存
 * @param      ptr : 内存首地址
 * @retval     无
 */
void alientek_ncr_free(void *ptr)
{
    myfree(SRAMIN, ptr);
}

```

alientek_ncr_memset、alientek_ncr_free 和 alientek_ncr_free 三个函数的实现主要调用 malloc 中的函数实现。这里就不多讲，忘记这些函数的实现，可以回顾一下内存管理实验章节。

2. main.c 代码

在这里讲一下正点原子提供的手写数字识别库实现数字字母识别的步骤：

1) 调用 alientek_ncr_init 函数，初始化识别程序

该函数用来初始化识别器，在手写识别进行之前，必须调用该函数。

2) 获取输入点阵数据

此步，我们通过触摸屏获取输入轨迹点阵坐标，然后存放到一个缓冲区里面，注意至少要输入 2 个不同坐标的点阵数据，才能正常识别。注意输入点数不要太多，太多的话，需要更多的内存，我们推荐的输入点数范围：100~200 点。

3) 调用 alientek_ncr 函数，得到识别结果

通过调用 alientek_ncr 函数，我们可以得到输入点阵的识别结果，结果将保存在 result 参数里面，采用 ASCII 码格式存储。

4) 调用 alientek_ncr_stop 函数，终止识别

如果不需要继续识别，则调用 alientek_ncr_stop 函数，终止识别器。如果还需要继续识别，重复步骤 2 和步骤 3 即可。

以上 4 个步骤，就是使用正点原子手写识别库的方法，十分简单。这个操作流程也在主函数中清晰看到，下面看一下 main.c，其代码如下：

```

int main(void)
{
    uint8_t i = 0;
    uint8_t tcnt = 0;
    char sbuf[10];
    uint8_t key;
    uint16_t pcnt = 0;
    uint8_t mode = 4;           /* 默认是混合模式 */
    uint16_t lastpos[2];       /* 最后一次的数据 */

    HAL_Init();                /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);             /* 延时初始化 */
    usart_init(115200);         /* 串口初始化为 115200 */
    led_init();                 /* 初始化 LED */
    lcd_init();                 /* 初始化 LCD */
    key_init();                 /* 初始化按键 */
    tp_dev.init();              /* 初始化触摸屏 */
}

```

```

norflash_init();          /* 初始化 NORFLASH */
my_mem_init(SRAMIN);      /* 初始化内部 SRAM 内存池 */

exfuns_init();            /* 为 fatfs 相关变量申请内存 */
f_mount(fs[0], "0:", 1);  /* 挂载 SD 卡 */
f_mount(fs[1], "1:", 1);  /* 挂载 FLASH */

alientek_ncr_init();      /* 初始化手写识别 */

while (fonts_init())      /* 检查字库 */
{
    lcd_show_string(60, 50, 200, 16, 16, "Font Error!", RED);
    delay_ms(200);
    lcd_fill(60, 50, 240, 66, WHITE); /* 清除显示 */
    delay_ms(200);
}

RESTART:
text_show_string(60, 10, 200, 16, "正点原子 STM32F1 开发板", 16, 0, RED);
text_show_string(60, 30, 200, 16, "手写识别实验", 16, 0, RED);
text_show_string(60, 50, 200, 16, "正点原子@ALIENTEK", 16, 0, RED);
text_show_string(60, 70, 200, 16, "KEY0:MODE KEY1:Adjust", 16, 0, RED);
text_show_string(60, 90, 200, 16, "识别结果:", 16, 0, RED);
lcd_draw_rectangle(19, 114, lcddev.width - 20, lcddev.height - 5, RED);
text_show_string(96, 207, 200, 16, "手写区", 16, 0, BLUE);
tcnt = 100;

while (1)
{
    key = key_scan(0);

    if (key == KEY1_PRES && (tp_dev.touchtype & 0X80) == 0)
    {
        tp_adjust();          /* 屏幕校准 */
        lcd_clear(WHITE);
        goto RESTART;        /* 重新加载界面 */
    }

    if (key == KEY0_PRES)
    {
        lcd_fill(20, 115, 219, 314, WHITE); /* 清除当前显示 */
        mode++;

        if (mode > 4) mode = 1;

        switch (mode)
        {
            case 1:
                text_show_string(80, 207, 200, 16, "仅识别数字", 16, 0, BLUE);
                break;
            case 2:
                text_show_string(64, 207, 200, 16, "仅识别大写字母", 16, 0, BLUE);
                break;
            case 3:
                text_show_string(64, 207, 200, 16, "仅识别小写字母", 16, 0, BLUE);
                break;
            case 4:
                text_show_string(88, 207, 200, 16, "全部识别", 16, 0, BLUE);
                break;
        }
        tcnt = 100;
    }
}

```

```

tp_dev.scan(0); /* 扫描 */

if (tp_dev.sta & TP_PRES_DOWN) /* 有按键被按下 */
{
    delay_ms(1); /* 必要的延时, 否则老认为有按键按下 */
    tcnt = 0; /* 松开时的计数器清空 */

    if ((tp_dev.x[0] < (lcddev.width - 20 - 2) &&
        tp_dev.x[0] >= (20 + 2)) &&
        (tp_dev.y[0] < (lcddev.height - 5 - 2) &&
        tp_dev.y[0] >= (115 + 2)))
    {
        if (lastpos[0] == 0xFFFF)
        {
            lastpos[0] = tp_dev.x[0];
            lastpos[1] = tp_dev.y[0];
        }
        /* 画线 */
        lcd_draw_bline(lastpos[0], lastpos[1], tp_dev.x[0], tp_dev.y[0], 2, BLUE);
        lastpos[0] = tp_dev.x[0];
        lastpos[1] = tp_dev.y[0];

        if (pcnt < 200) /* 总点数少于 200 */
        {
            if (pcnt)
            {
                if((ncr_input_buf[pcnt - 1].y != tp_dev.y[0]) &&
                    (ncr_input_buf[pcnt - 1].x != tp_dev.x[0])) /* x,y 不相等 */
                {
                    ncr_input_buf[pcnt].x = tp_dev.x[0];
                    ncr_input_buf[pcnt].y = tp_dev.y[0];
                    pcnt++;
                }
            }
            else
            {
                ncr_input_buf[pcnt].x = tp_dev.x[0];
                ncr_input_buf[pcnt].y = tp_dev.y[0];
                pcnt++;
            }
        }
    }
}
else /* 按键松开了 */
{
    lastpos[0] = 0xFFFF;
    tcnt++;
    delay_ms(10);
    /* 延时识别 */
    i++;

    if (tcnt == 40)
    {
        if (pcnt) /* 有有效的输入 */
        {
            printf("总点数:%d\r\n", pcnt);
            alientek_ncr(ncr_input_buf, pcnt, 6, mode, sbuf);
            printf("识别结果:%s\r\n", sbuf);
            pcnt = 0;
            lcd_show_string(60 + 72, 90, 200, 16, 16, sbuf, BLUE);
        }

        lcd_fill(20, 115, lcddev.width-20-1, lcddev.height-5-1, WHITE);
    }
}

```



```
if (i == 30)
{
    i = 0;
    LED0_TOGGLE();
}
}
```

main 函数代码实现手写识别功能的步骤跟前面所说的一致。其中使用到了 `lcd_draw_bline` 函数，该函数是用来画粗线的，函数的实现也是通过调用 `lcd_fill_circle` 实现，这里就不做展开讲解该函数了。在获取触点数据需要注意的是：这里我们采用的都是不重复的点阵（即相邻的坐标不相等）。这样可以避免重复数据，而重复的点阵数对于识别是没有帮助的。

57.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 57.4.1 所示：

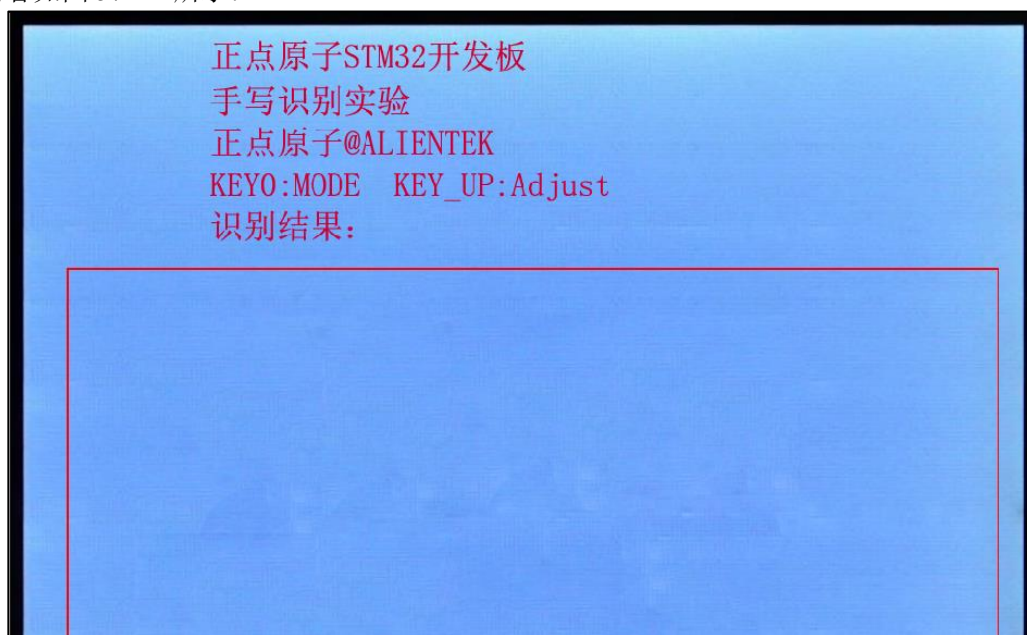


图 57.4.1 手写识别界面图

此时，我们在手写区写数字/字母，即可得到识别结果，如图 57.4.2 所示：

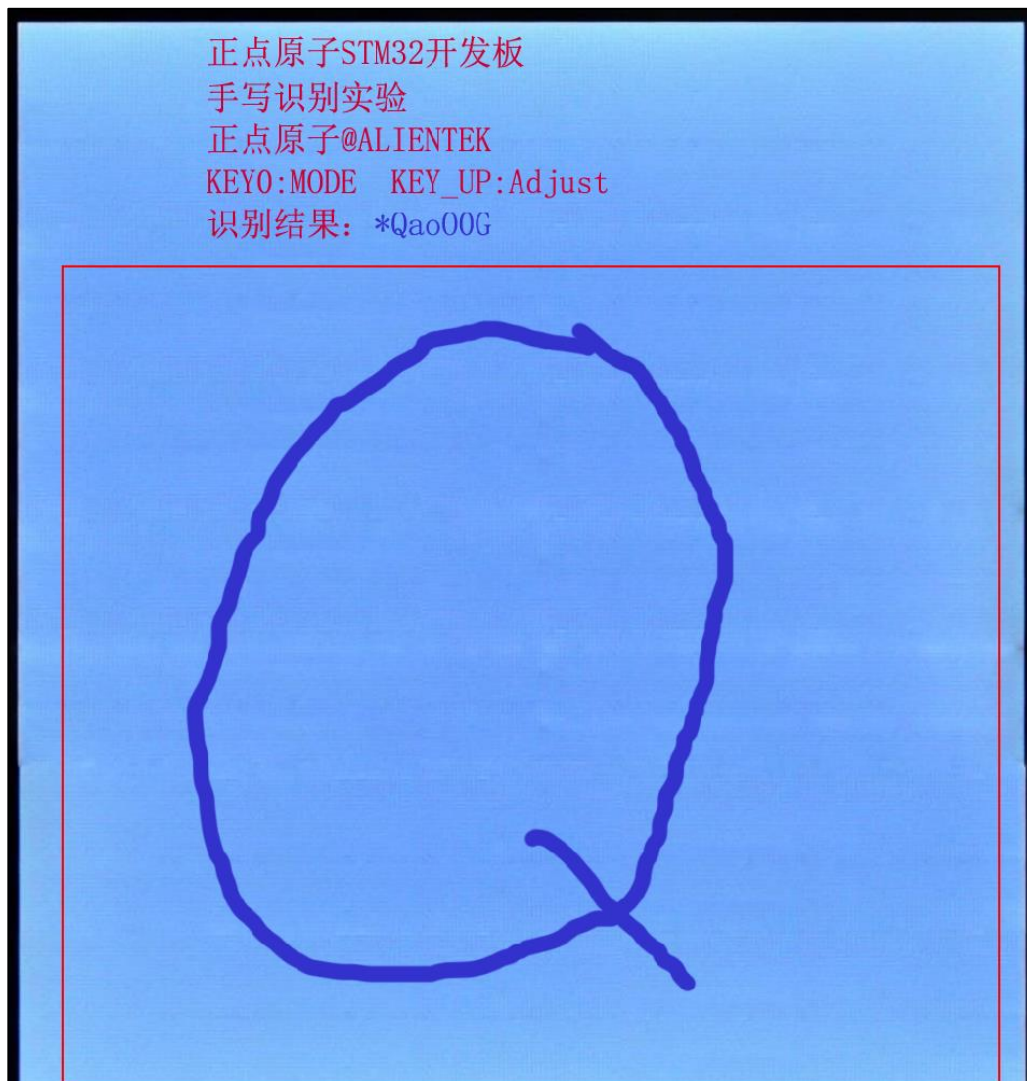


图 57.4.2 手写识别结果图

按下 KEY0 可以切换识别模式，同时在识别区提示当前模式。按下 KEY1 可以对屏幕进行校准（仅限电阻屏，电容屏无需校准）。每次识别结束，会在串口打印本次识别的输入点数和识别结果，大家可以通过串口助手查看。

第五十八章 T9 拼音输入法实验

本章，我们将介绍如何在 STM32 板子上实现一个简单的 T9 中文拼音输入法。本章分为如下几个小节：

- 58.1 拼音输入法简介
- 58.2 硬件设计
- 58.3 程序设计
- 58.4 下载验证

58.1 拼音输入法简介

在计算机上汉字的输入法有很多种，比如拼音输入法、五笔输入法、笔画输入法、区位输入法等。其中，又以拼音输入法用的最多。拼音输入法又可以分为很多类，比如全拼输入、双拼输入等。

而在手机上，用的最多的应该算是 T9 拼音输入法了，T9 输入法全名为智能输入法，字库容量九千多字，支持十多种语言。T9 输入法是由美国特捷通讯（Tegic Communications）软件公司开发的，该输入法解决了小型掌上设备的文字输入问题，已经成为全球手机文字输入的标准之一。

一般情况下，手机拼音输入键盘如图 58.1.1 所示：

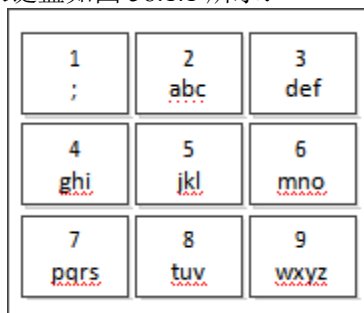


图 58.1.1 手机拼音输入键盘

在这个键盘上，我们对比下传统的输入法和 T9 输入法，输入“中国”两个字需要的按键次数。传统的方法，先按 4 次 9，输入字母 z，再按 2 次 4，输入字母 h，再按 3 次 6，输入字母 o，再按 2 次 6，输入字母 n，最后按 1 次 4，输入字母 g。这样，输入“中”字，要按键 12 次，接着同样的方法，输入“国”字，需要按 6 次，总共就是 18 次按键。

如果是 T9，我们输入“中”字，只需要输入：9、4、6、6、4，即可实现输入“中”字，在选择“中”字之后，T9 会联想出一系列同“中”字组合的词，如：文、国、断、山等。这样输入“国”字，我们直接选择即可，所以输入“国”字按键 0 次，这样使用 T9 输入法总共只需要 5 次按键。

这就是 T9 智能输入法的优越之处。正因为 T9 输入法高效便捷的输入方式得到了众多手机厂商的采用，以至于 T9 成为了使用频率最高知名度最大的手机输入法。

在本实验中，我们实现的 T9 拼音输入法，没有真正的 T9 那么强大，我们这里仅实现输入部分，不支持词组联想。

58.2 硬件设计

1. 例程功能

开机的时候先检测字库，然后显示提示信息和绘制拼音输入表，之后进入等待输入状态。此时用户可以通过屏幕上的拼音输入表输入拼音数字串（通过 DEL 可以实现退格），然后程序自动检测与之对应的拼音和汉字，并显示在屏幕上（同时输出到串口）。如果有多个匹配的拼音，

则通过 KEY_UP 和 KEY1 进行选择。按键 KEY0 用于清除一次输入，按键 KEY2 用于触摸屏校准。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

- 1) LED 灯
LED0 - PB5
- 2) 独立按键
KEY0 - PE4
KEY1 - PE3
WK_UP - PA0
- 3) 串口 1
- 4) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏, 16 位 8080 并口驱动)
- 5) NOR FLASH, 通过 SPI2 驱动, 我们需要里面的汉字库

58.3 程序设计

58.3.1 程序流程图

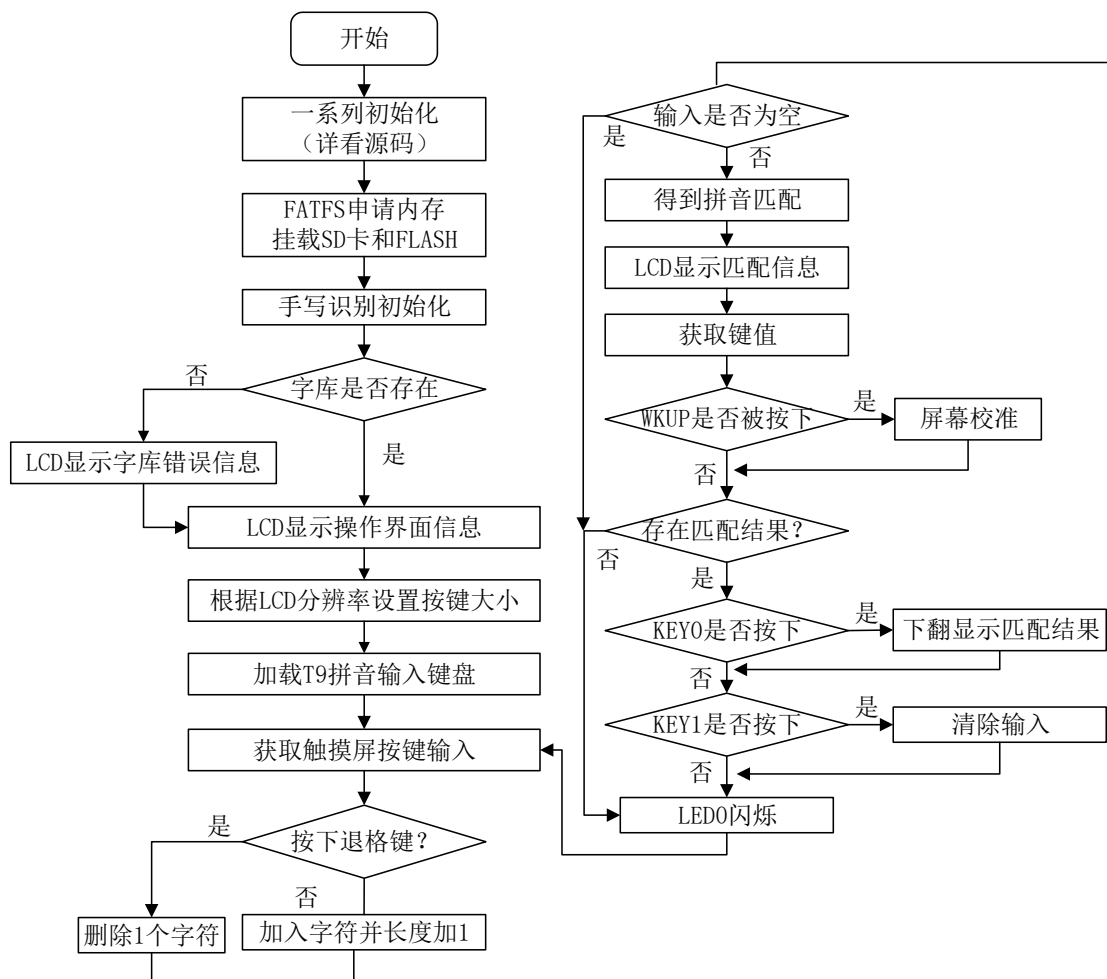


图 58.3.1.1 T9 拼音输入法实验程序流程图

我们通过 LCD 的绘制算法生成一个 9 宫格输入法, 采集到对应的合法输入点后, 我们把这些信息传给解码库进行解码, 识别出对应的拼音, 并调用汉字库显示对应的匹配汉字。

58.3.2 程序解析

1.T9 拼音输入法代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，T9 拼音输入法的驱动源码包括 pyinput.c、pyinput.h 和 pymb.h 三个文件。

首先我们先介绍一下定义在 pymb.h 中的拼音索引。先介绍一下汉字排列表，该表将汉字拼音所有可能的组合都列出来了，如下所示：

```
/* 汉字排列表 */
const uint8_t PY_mb_space []={" "};
const uint8_t PY_mb_a []={"啊阿腌吡铜屌嘎铜呵腌"};
const uint8_t PY_mb_ai []={"爱埃挨哎唉哀皑癌蒿矮艾碍隘掙暖嗑媛媛暖破镣靛"};
.....此处省略 N 多个组合.....
const uint8_t PY_mb_zu []={"足组卒族祖祖祖阻阻阻茁"};
const uint8_t PY_mb_zuan []={"钻攥纂缵"};
const uint8_t PY_mb_zui []={"最罪嘴醉蓑"};
const uint8_t PY_mb_zun []={"尊遵樽樽"};
const uint8_t PY_mb_zuo []={"左佐做作坐座昨撮昨昨琢昨昨昨昨昨昨昨"};
```

这里我们只列出了部分组合，我们将这些组合称之为码表，然后将这些码表和其对应的数字串对应起来，组成一个拼音索引表，如下所示：

```
/* 拼音索引表 */
const py_index py_index3[]=
{
    {"", "", (uint8_t*)PY_mb_space},
    {"2", "a", (uint8_t*)PY_mb_a},
    {"3", "e", (uint8_t*)PY_mb_e},
    {"6", "o", (uint8_t*)PY_mb_o},
    {"24", "ai", (uint8_t*)PY_mb_ai},
    .....此处省略 N 多个组合.....
    {"94664", "zhong", (uint8_t*)PY_mb_zhong},
    {"94824", "zhuai", (uint8_t*)PY_mb_zhuai},
    {"94826", "zhuan", (uint8_t*)PY_mb_zhuan},
    {"248264", "chuang", (uint8_t*)PY_mb_chuang},
    {"748264", "shuang", (uint8_t*)PY_mb_shuang},
    {"948264", "zhuang", (uint8_t*)PY_mb_zhuang},
};
```

其中 py_index 是一个结构体，定义如下：

```
/* 拼音码表与拼音的对应表 */
typedef struct
{
    uint8_t *py_input; /* 输入的字符串 */
    uint8_t *py; /* 对应的拼音 */
    uint8_t *pymb; /* 码表 */
}py_index;
```

其中 py_input，即与拼音对应的数字串，比如“94824”。py，即与 py_input 数字串对应的拼音，如果 py_input = “94824”，那么 py 就是“zhuai”。最后 pymb 就是我们前面说到的码表。注意，一个数字串可以对应多个拼音，也可以对应多个码表。

有了这个拼音索引表（py_index3）之后，我们只需要将输入的数字串和 py_index3 索引表里面所有成员的 py_input 对比，将所有完全匹配的情况记录下来，用户要输入的汉字就被确定了，然后由用户选择可能的拼音组成（假设有多个匹配的项目），再选择对应的汉字，即完成一次汉字输入。

当然还可能是找遍了索引表，也没有发现一个完全符合要求的成员，那么我们会统计匹配数最多的情况，作为最佳结果，反馈给用户。比如，用户输入“323”，找不到完全匹配的情况，那么我们就将能和“32”匹配的结果返回给用户。这样，用户还是可以得到输入结果，同时还可以知道输入有问题，提示用户需要检查输入是否正确。

我们归纳一下完整的 T9 拼音输入步骤：

1) 输入拼音数字串

我们用到的 T9 拼音输入法的核心思想就是对比用户输入的拼音数字串，所以必须先由用户输入拼音数字串。

2) 在拼音索引表里面查找和输入字符串匹配的项，并记录

在得到用户输入的拼音数字串之后，在拼音索引表里面查找所有匹配的项目，如果有完全匹配的项目，就全部记录下来，如果没有完全匹配的项目，则记录匹配情况最好的一个项目。

3) 显示匹配清单里面所有可能的文字，供用户选择

将匹配项目的拼音和对应的汉字显示出来，供用户选择。如果有多个匹配项（一个数字串对应多个拼音的情况），则用户还需要选择拼音。

4) 用户选择匹配项，并选择对应的汉字

用户对匹配的拼音和汉字进行选择，选中其真正想输入的拼音和汉字，实现一次拼音输入。

下面介绍一下 pyinput.c 中比较核心的函数 get_matched_pymb，代码如下：

```
/**
 * @brief      获取匹配的拼音码表
 * @param      strin      : 输入的字符串,形如:"726"
 * @param      matchlist  : 输出的匹配表
 * @retval     匹配状态
 *              [7] , 0,表示完全匹配; 1,表示部分匹配(仅在没有完全匹配的时候才会出现)
 *              [6:0], 完全匹配的时候,表示完全匹配的拼音个数
 *              部分匹配的时候,表示有效匹配的位数
 */
uint8_t get_matched_pymb(uint8_t *strin, py_index **matchlist)
{
    py_index *bestmatch = 0; /* 最佳匹配 */
    uint16_t pyindex_len = 0;
    uint16_t i = 0;
    uint8_t temp, mcnt = 0, bmcnt = 0;

    bestmatch = (py_index *)&py_index3[0]; /* 默认为 a 的匹配 */
    pyindex_len = sizeof(py_index3) / sizeof(py_index3[0]); /* 得到 py 索引表的大小 */

    for (i = 0; i < pyindex_len; i++)
    {
        temp = str_match(strin, (uint8_t *)&py_index3[i].py_input);

        if (temp)
        {
            if (temp == 0xFF)
            {
                matchlist[mcnt++] = (py_index *)&py_index3[i];
            }
            else if (temp > bmcnt) /* 找最佳匹配 */
            {
                bmcnt = temp;
                bestmatch = (py_index *)&py_index3[i]; /* 最好的匹配 */
            }
        }
    }

    if (mcnt == 0 && bmcnt) /* 没有完全匹配的结果,但是有部分匹配的结果 */
    {
        matchlist[0] = bestmatch;
        mcnt = bmcnt | 0x80; /* 返回部分匹配的有效位数 */
    }

    return mcnt; /* 返回匹配的个数 */
}
```

该函数实现的是将用户输入拼音数字串同拼音索引表里面的各个项对比，找出匹配结果，并将完全匹配的项目存放在 matchlist 里面，同时记录匹配数。对于那些没有完全匹配的输入串，则查找与其最佳匹配的项目，并将匹配的长度返回。

其中该文件还有一个函数 `test_py`，提供给 `usmart` 调用，实现串口测试，在串口测试的时候才能用到，如果不使用的话，可以去掉。本实验也是加入 `usmart` 控制，大家可以通过该函数实现串口调试拼音输入法。

其他两个函数比较简单，这里就不细说了。

前面提及的 `matchlist`，其定义在 `pyinput.h` 中，代码如下：

```
/* 拼音输入法 */
typedef struct
{
    uint8_t(*getpymb)(uint8_t *instr); /* 字符串到码表获取函数 */
    py_index *pymb[MAX_MATCH_PYMB]; /* 码表存放位置 */
}pyinput;
```

该结构体提供了两个成员，一个成员就是字符串到码表获取函数，另外一个成员也就是码表的存放位置。另外一个结构体 `py_index`，已经在前面讲解了，这里就不作展开。

2. main.c 代码

在 `main.c` 文件下，除了 `main` 函数之外，还有 `py_load_ui`、`py_key_staset`、`py_get_keynum` 和 `py_show_result` 函数。其中，`py_load_ui` 函数是用于加载输入键盘，在 LCD 上显示我们输入拼音数字串的虚拟键盘。`py_key_staset` 函数用于与设置虚拟键盘某个按键的状态（按下/松开）。`py_get_keynum` 函数用于得到触摸屏当前按下的按键键值，通过该函数实现拼音数字串的获取。`py_show_result` 函数用于显示输入串的匹配结果，并将结果打印到串口。这部分代码就不列出来了，大家可以自行理解。

下面看一下 `main` 主函数的代码：

```
int main(void)
{
    uint8_t i = 0;
    uint8_t result_num;
    uint8_t cur_index;
    uint8_t key;
    uint8_t temp;
    uint8_t inputstr[7]; /* 最大输入 6 个字符+结束符 */
    uint8_t inputlen; /* 输入长度 */

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev_init(72); /* 初始化 USMART */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    tp_dev_init(); /* 初始化触摸屏 */
    sram_init(); /* SRAM 初始化 */
    norflash_init(); /* 初始化 NORFLASH */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */

    exfuncs_init(); /* 为 fatfs 相关变量申请内存 */
    f_mount(fs[0], "0:", 1); /* 挂载 SD 卡 */
    f_mount(fs[1], "1:", 1); /* 挂载 FLASH */
    RESTART:
    while (fonts_init()) /* 检查字库 */
    {
        lcd_show_string(60, 50, 200, 16, 16, "Font Error!", RED);
        delay_ms(200);
        lcd_fill(60, 50, 240, 66, WHITE); /* 清除显示 */
        delay_ms(200);
    }
```

```

text_show_string(30, 5, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
text_show_string(30, 25, 200, 16, "拼音输入法实验", 16, 0, RED);
text_show_string(30, 45, 200, 16, "正点原子@ALIENTEK", 16, 0, RED);
text_show_string(30, 65, 200, 16, "KEY_UP:校准", 16, 0, RED);
text_show_string(30, 85, 200, 16, "KEY0:翻页 KEY1:清除", 16, 0, RED);
text_show_string(30, 105, 200, 16, "输入:          匹配: ", 16, 0, RED);
text_show_string(30, 125, 200, 16, "拼音:          当前: ", 16, 0, RED);
text_show_string(30, 145, 210, 32, "结果:", 16, 0, RED);

/* 根据 LCD 分辨率设置按键大小 */
if (lcddev.id == 0X5310)
{
    kbdxsize = 86;
    kbysize = 43;
}
else if (lcddev.id == 0X5510)
{
    kbdxsize = 140;
    kbysize = 70;
}
else
{
    kbdxsize = 60;
    kbysize = 40;
}

py_load_ui(30, 195);
my_mem_set(inputstr, 0, 7); /* 全部清零 */
inputlen = 0; /* 输入长度为 0 */
result_num = 0; /* 总匹配数清零 */
cur_index = 0;

while (1)
{
    i++;
    delay_ms(10);
    key = py_get_keynum(30, 195);

    if (key)
    {
        if (key == 1) /* 删除 */
        {
            if (inputlen)inputlen--;
            inputstr[inputlen] = '\0'; /* 添加结束符 */
        }
        else
        {
            inputstr[inputlen] = key + '0'; /* 输入字符 */
            if (inputlen < 7)inputlen++;
        }

        if (inputstr[0] != NULL)
        {
            temp = t9.getpymb(inputstr); /* 得到匹配的结果数 */
            if (temp) /* 有部分匹配/完全匹配的结果 */
            {
                result_num = temp & 0X7F; /* 总匹配结果 */
                cur_index = 1; /* 当前为第一个索引 */

                if (temp & 0X80) /* 是部分匹配 */
                {
                    inputlen = temp & 0X7F; /* 有效匹配位数 */
                }
            }
        }
    }
}

```

```

        inputstr[inputlen] = '\0'; /* 不匹配的位数去掉 */
        if (inputlen > 1)
        {
            temp = t9.getpymb(inputstr); /* 重新获取完全匹配字符数 */
            /* 如果还是部分匹配，直接匹配数为 0，则表示匹配数量 */
            result_num = (temp & 0X80)? 0 : (temp & 0X7F);
        }
    }
}
else /* 没有任何匹配 */
{
    inputlen--;
    inputstr[inputlen] = '\0';
}
else
{
    cur_index = 0;
    result_num = 0;
}
lcd_fill(30 + 40, 105, 30 + 40 + 48, 110 + 16, WHITE); /* 清除之前的显示 */
lcd_show_num(30 + 144, 105, result_num, 1, 16, BLUE); /* 显示匹配的结果数 */
/* 显示有效的数字串 */
text_show_string(30 + 40, 105, 200, 16, (char *)inputstr, 16, 0, BLUE);
py_show_result(cur_index); /* 显示第 cur_index 的匹配结果 */
}

key = key_scan(0);
if (key == WKUP_PRES && tp_dev.touchtype == 0) /* KEYUP 按下, 且是电阻屏 */
{
    tp_dev.adjust();
    lcd_clear(WHITE);
    goto RESTART;
}

switch (key)
{
    case KEY0_PRES: /* 下翻 */
        if (result_num) /* 存在匹配的结果 */
        {
            if (cur_index < result_num) cur_index++;
            else cur_index = 1;
            /* 显示第 cur_index 的匹配结果 */
            py_show_result(cur_index);
        }
        break;
    case KEY1_PRES: /* 清除输入 */
        /* 清除之前的显示 */
        lcd_fill(30 + 40, 145, lcddev.width - 1, 145 + 48, WHITE);
        goto RESTART;
}

if (i == 30)
{
    i = 0;
    LED0_TOGGLE();
}
}
}

```

在 main 函数里，实现了我们在 58.2.1 小节所说的功能，也是按照它表述的逻辑进行实现。在这里我们并没有实现汉字选择功能，但是由本例程作为基础，再实现汉字选择功能就比较简单了，大家自行实现即可。注意：kbdxsize 和 kbysize 代表虚拟键盘按键宽度和高度，程序根

据 LCD 分辨率不同而自动设置这两个参数，以达到较好的输入效果。

58.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 58.4.1 所示：

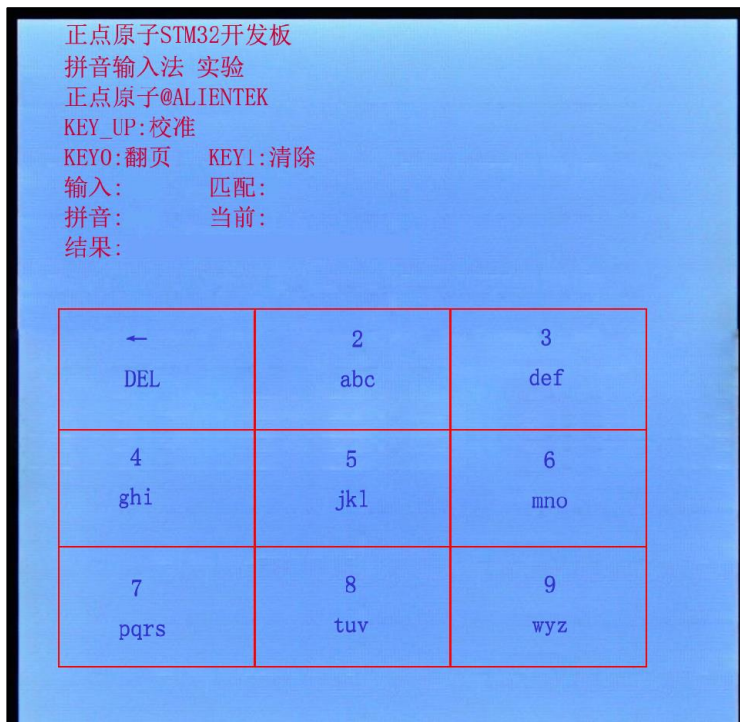


图 58.4.1 汉字输入法界面图

此时，我们在虚拟键盘上输入拼音数字串，即可实现拼音输入，如图 58.4.2 所示：



图 58.4.2 实现拼音输入

如果发现输入错误了，可以通过屏幕上的 DEL 按键，来退格。如果有多个匹配的情况（匹配值大于 1），则可以通过 KEY_UP 和 KEY1 来选择拼音。通过按下 KEY0，可以 KEY_UP 和

KEY1 来选择拼音。通过按下 KEY0，可以清除当前输入，通过按下 KEY2，可以实现触摸屏校准（仅限电阻屏，电容屏无需校准）。

触摸输入的匹配项可以通过串口调试助手打印出来：

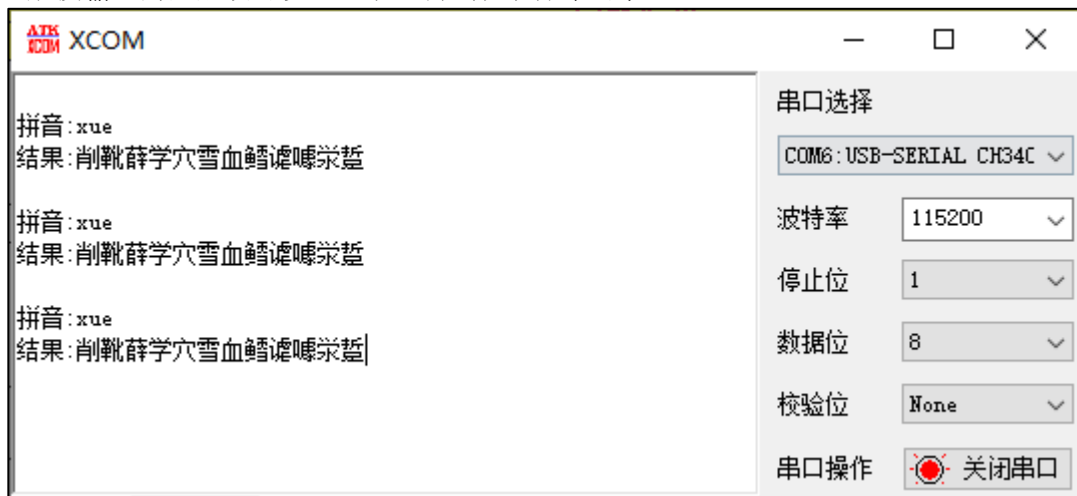


图 58.4.2 拼音输入有匹配结果时打印

本章，我们还可以借助 USART 调用 test_py 来实现输入法调试，如图 58.4.3 所示：



图 58.4.2 USART 测试 T9 拼音输入法图

第五十九章 串口 IAP 实验

IAP, 即在应用编程, 通俗地说法就是“程序升级”。产品阶段设计完成后, 在脱离实验室的调试环境下, 如果想对产品做功能升级或 BUG 修复会十分麻烦, 如果硬件支持, 在出厂时预留一套升级固件的流程, 就可以很好解决这个问题, IAP 技术就是为此而生的。在之前的 FLASH 模拟 EEPROM 实验里面, 我们学习了 STM32F103 的 FLASH 自编程, 本章我们将结合 FLASH 自编程的知识, 通过 STM32F103 的串口实现一个简单的 IAP 功能。

本章分为如下几个小节:

59.1 IAP 简介

59.2 硬件设计

59.3 软件设计

59.4 下载验证

59.1 IAP 简介

IAP (In Application Programming) 即在应用编程。在讲解 STM32 的启动模式时我们已经知道 STM32 可以通过设置 MSP 的方式从不同的地址启动: 包括 Flash 地址、RAM 地址等, 在默认方式下, 我们的嵌入式程序是以连续二进制的方式烧录到 STM32 的可寻址 Flash 区域上的。如果我们用的 Flash 容量大到可以存储两个或多个的完整程序, 在保证每个程序完整的情况下, 上电后的程序通过修改 MSP 的方式, 就可以保证一个单片机上有多个有功能差异的嵌入式软件, 这就是我们要讲解的 IAP 的设计思路。

IAP 是用户自己的程序在运行过程中对 User Flash 的部分区域进行烧写, 目的是为了在产品发布后可以方便地通过预留的通信口对产品中的固件程序进行更新升级, 由于用户可以自定义通讯方式和自定义加密, 使得 IAP 在使用上非常灵活。通常实现 IAP 功能时, 即用户程序运行中作自身的更新操作, 需要在设计固件程序时编写两个项目代码, 第一个程序检查有无升级需求, 并通过某种通信方式(如 USB、USART)接收程序或数据, 执行对第二部分代码的更新; 第二个项目代码才是真正的功能代码。这两部分项目代码都同时烧录在 User Flash 中, 当芯片上电后, 首先是第一个项目代码开始运行, 它做如下操作:

- 1) 检查是否需要第二部分代码进行更新
- 2) 如果不需要更新则转到 4)
- 3) 执行更新操作
- 4) 跳转到第二部分代码执行

第一部分代码必须通过其它手段, 如 JTAG、ISP 等方式烧录, 常常是烧录后就不再进行更改; 第二部分代码可以使用第一部分代码 IAP 功能烧入, 也可以和第一部分代码一起烧入, 以后需要程序更新时再通过第一部分 IAP 代码更新。

我们将第一个项目代码称之为 Bootloader 程序, 第二个项目代码称之为 APP 程序, 他们存放在 STM32F103 内部 FLASH 的不同地址范围, 一般从最低地址区开始存放 Bootloader, 紧跟其后的就是 APP 程序(注意, 如果 FLASH 容量足够, 是可以设计很多 APP 程序的, 本章我们只讨论一个 APP 程序的情况)。这样我们就是要实现 2 个程序: Bootloader 和 APP。

STM32F1 的 APP 程序不仅可以放到 FLASH 里面运行, 也可以放到 SRAM 里面运行, 本章, 我们将制作两个 APP, 一个用于 FLASH 运行, 一个用于内部 SRAM 运行。

我们先来看看 STM32F1 正常的程序运行流程(为了方便说明 IAP 过程, 我们先仅考虑代码全部存放在内部 FLASH 的情况), 如图 59.1.1 所示:

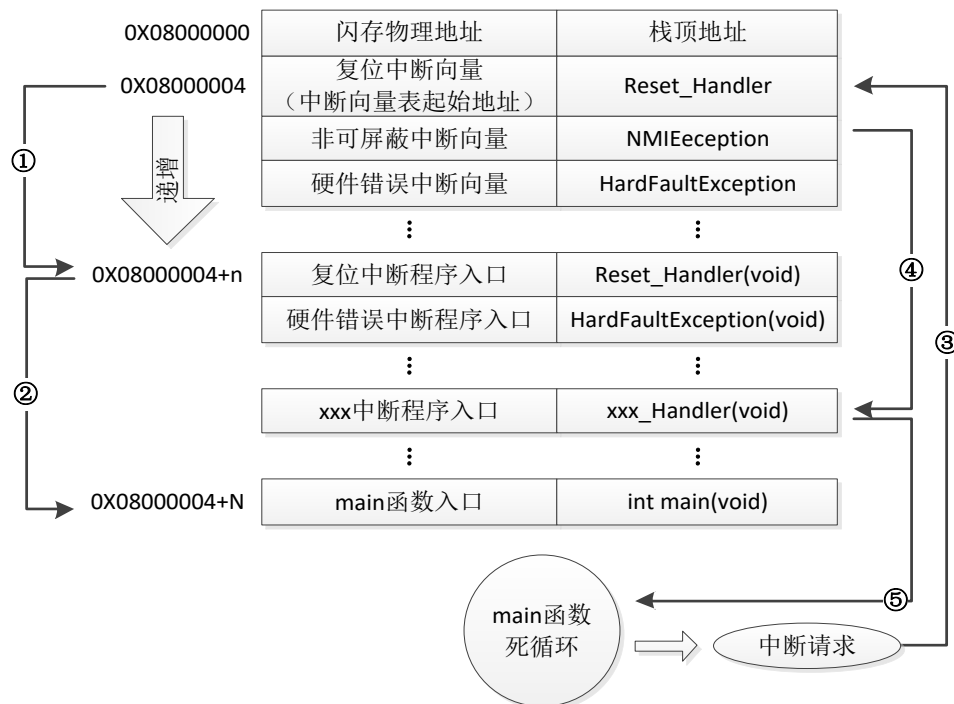


图 59.1.1 STM32F1 正常运行流程图

STM32F1 的内部闪存（FLASH）地址起始于 0X0800 0000，一般情况下，程序文件就从此地址开始写入。此外 STM32F103 是基于 Cortex-M3 内核的微控制器，其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，而这张“中断向量表”的起始地址是 0x08000004，当中断来临，STM32F103 的内部硬件机制亦会自动将 PC 指针定位到“中断向量表”处，并根据中断源取出对应的中断向量执行中断服务程序。

在图 59.1.1 中，STM32F103 在复位后，先从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，如图标号①所示；在复位中断服务程序执行完之后，会跳转到我们的 main 函数，如图标号②所示；而我们的 main 函数一般都是一个死循环，在 main 函数执行过程中，如果收到中断请求（发生了中断），此时 STM32F103 强制将 PC 指针指回中断向量表处，如图标号③所示；然后，根据中断源进入相应的中断服务程序，如图标号④所示；在执行完中断服务程序以后，程序再次返回 main 函数执行，如图标号⑤所示。

当加入 IAP 程序之后，程序运行流程如图 59.1.2 所示：

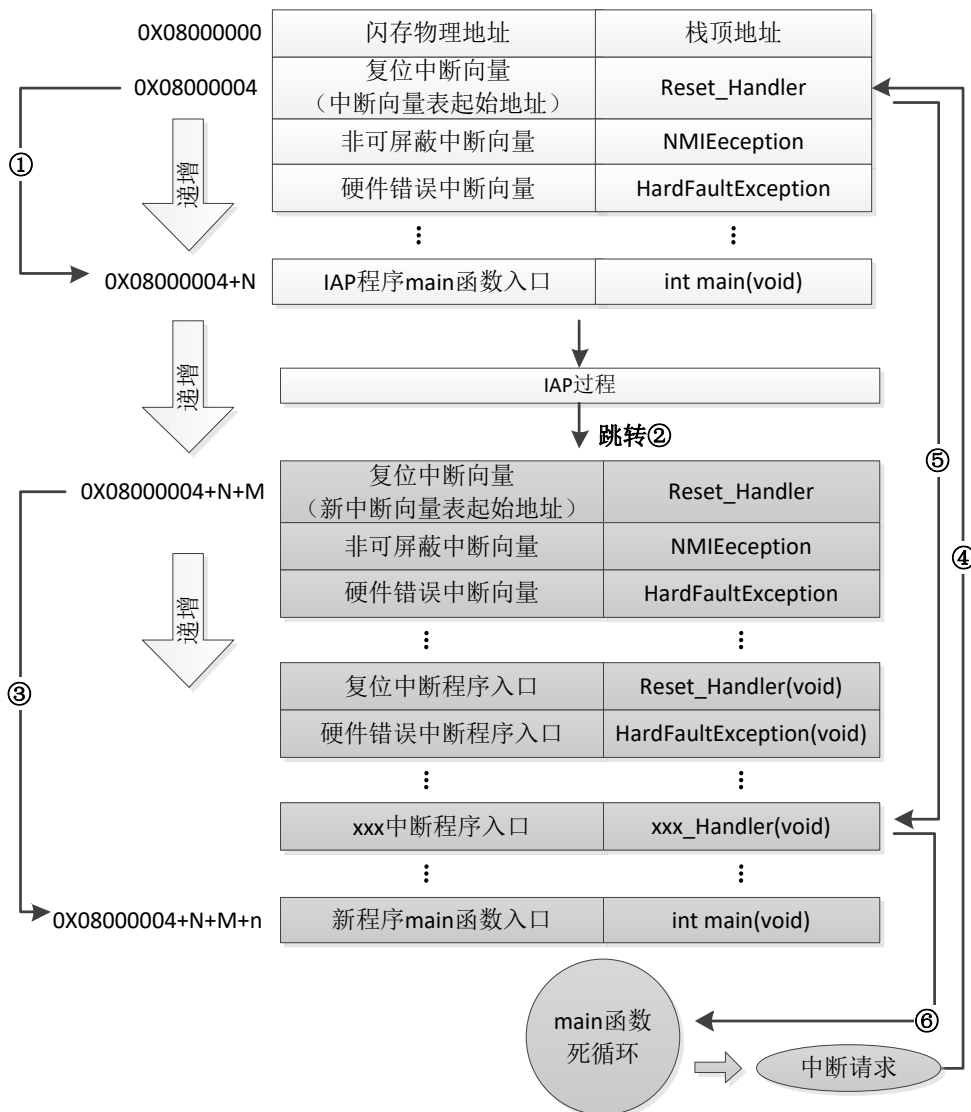


图 59.1.2 加入 IAP 之后程序运行流程图

在图 59.1.2 所示流程中，STM32F103 复位后，还是从 $0X08000004$ 地址取出复位中断向量的地址，并跳转到复位中断服务程序，在运行完复位中断服务程序之后跳转到 IAP 的 main 函数，如图标号①所示，此部分同图 59.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32F103 的 FLASH，灰底部分）。新程序的复位中断向量起始地址为 $0X08000004+N+M$ ），跳转至新写入程序的复位向量表，取出新程序的复位中断向量的地址，并跳转执行新程序的复位中断服务程序，随后跳转至新程序的 main 函数，如图标号②和③所示，同样 main 函数为一个死循环，并且注意到此时 STM32F103 的 FLASH，在不同位置上，共有两个中断向量表。

在 main 函数执行过程中，如果 CPU 得到一个中断请求，PC 指针仍然会强制跳转到地址 $0X08000004$ 中断向量表处，而不是新程序的中断向量表，如图标号④所示；程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑤所示；在执行完中断服务程序后，程序返回 main 函数继续运行，如图标号⑥所示。

通过以上两个过程的分析，我们知道 IAP 程序必须满足两个要求：

- 1) 新程序必须在 IAP 程序之后的某个偏移量为 x 的地址开始；
- 2) 必须将新程序的中断向量表相应的移动，移动的偏移量为 x ；

对 STM32F1 系列来说，闪存编程一次可以写入 16 位(半字)。闪存擦除操作可以按页面擦除或完全擦除(全擦除)。全擦除不影响信息块。根据类别的不同，Flash 有如下区别：

- 小容量产品主存储块最大为 $4K \times 64$ 位，每个存储块划分为 32 个 1K 字节的页。
- 中容量产品主存储块最大为 $16K \times 64$ 位，每个存储块划分为 128 个 1K 字节的页。

- 大容量产品主存储块最大为 64K×64 位，每个存储块划分为 256 个 2K 字节的页。
- 互联型产品主存储块最大为 32K×64 位，每个存储块划分为 128 个 2K 字节的页

使用时我们需要根据自己的芯片型号来选择，设计 IAP 程序时需要严格避免不同的程序占用相同 Flash 扇区的情形。

本章，我们有 2 个 APP 程序：

1，FLASH APP 程序，即只运行在内部 FLASH 的 APP 程序。

2，SRAM APP 程序，即只运行在内部 SRAM 的 APP 程序，其运行过程和图 59.1.2 相似，不过需要设置向量表的地址为 SRAM 的地址。

1.APP 程序起始地址设置方法

APP 我们使用以前的例程即可，不过需要对程序进行修改，

默认的条件下，图中 IROM1 的起始地址(Start)一般为 0x08000000，大小(Size)为 0x80000，即从 0x08000000 开始的 512K 空间为我们的程序存储区。

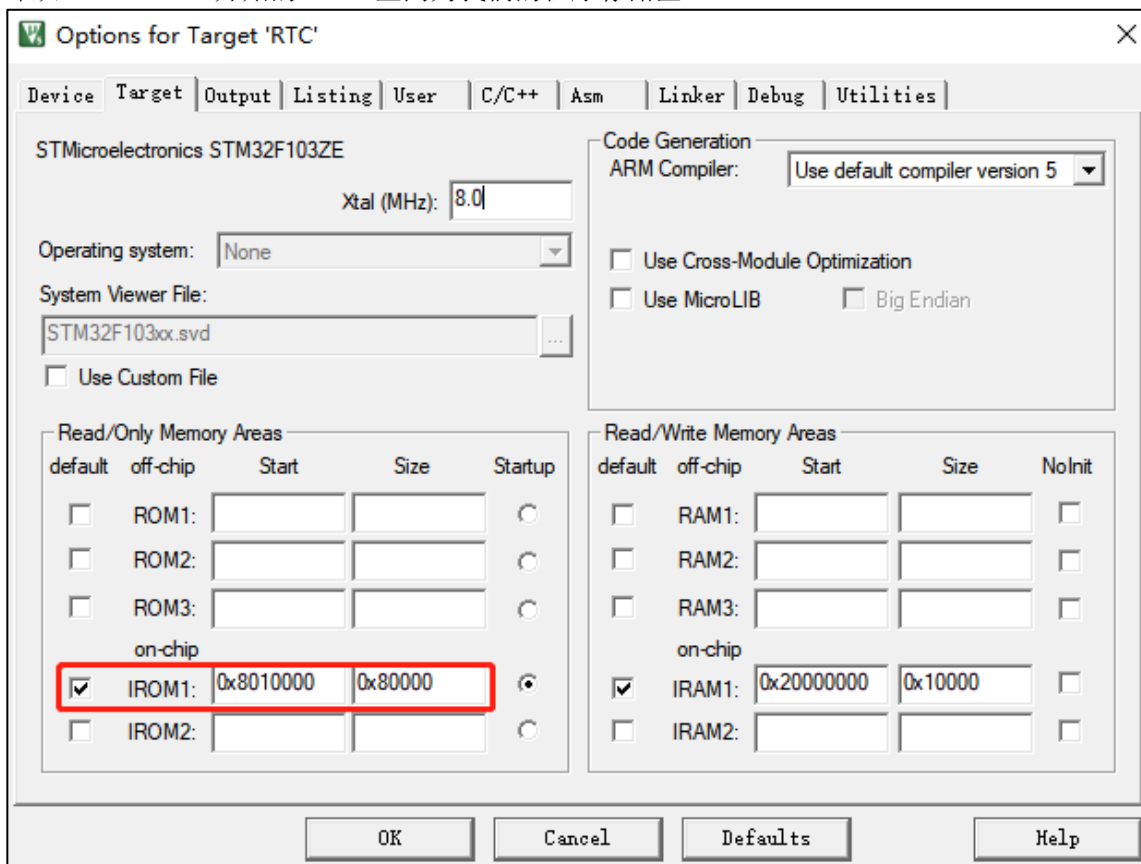


图 59.1.3 FLASH APP Target 选项卡设置

图 59.1.3 中，我们设置起始地址 (Start) 为 0X08010000，即偏移量为 0x10000 (64K 字节，即留给 BootLoader 的空间)，因而，留给 APP 用的 FLASH 空间 (Size) 为 0x80000-0x10000=0x70000 (448K 字节) 大小了。设置好 Start 和 Size，就完成 APP 程序的起始地址设置。IRAM 是内存的地址，APP 可以独占这些内存，我们不需要修改。

注意：需要确保 APP 起始地址在 Bootloader 程序结束位置之后，并且偏移量为 0X200 的倍数即可（相关知识，请参考：<http://www.openedv.com/posts/list/392.htm>）。

这是针对 FLASH APP 的起始地址设置，如果是 SRAM APP，那么起始地址设置如图 59.1.4 所示：

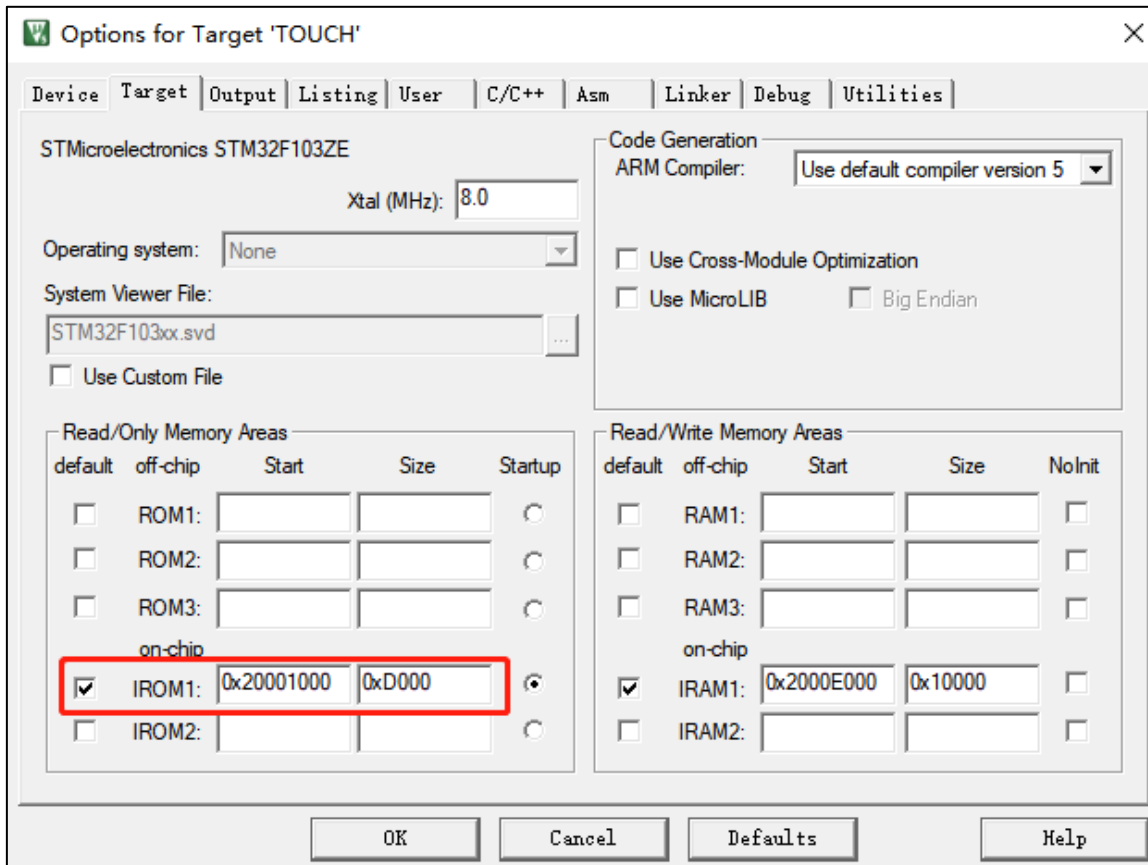


图 59.1.6 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址 (Start) 定义为: 0X20001000, 大小为 0XD000 (52K 字节), 即从地址 0X20000000 偏移 0X1000 开始, 存放 SRAM APP 代码。这个分配关系大家可以根据自己的实际情况修改, 由于 STM32F103ZE 只有一个 64K 的片内 SRAM, 存放程序的位置与变量的加载位置不能重复, 所以我们需要设置 IRAM1 中的地址到 SRAM 程序空间之外。

关于 APP 起始地址的设置方法, 我们就介绍到这里, 大家可以根据自己项目的实际需求进行修改。

2. 中断向量表的偏移量设置方法

VTOR 寄存器存放的是中断向量表的起始地址。默认的情况它由 BOOT 的启动模式决定, 对于 F103 来说就是指向 0x0800 0000 这个位置, 也就是从默认的启动位置加载中断向量等信息, 不过 ST 允许重定向这个位置, 这样就可以从 Flash 区域的任意位置启动我们的代码了。我们可以通过调用 sys.c 里面的 sys_nvic_set_vector_table 函数实现, 该函数定义如下:

```
/**
 * @brief      设置中断向量表偏移地址
 * @param      baseaddr: 基址
 * @param      offset: 偏移量
 * @retval     无
 */
void sys_nvic_set_vector_table(uint32_t baseaddr, uint32_t offset)
{
    /* 设置NVIC的向量表偏移寄存器,VTOR低9位保留,即[8:0]保留 */
    SCB->VTOR = baseaddr | (offset & (uint32_t)0xFFFFE00);
}
```

该函数用于设置中断向量偏移, baseaddr 为基地址 (即 APP 程序首地址), Offset 为偏移量, 需要根据自己的实际情况进行设置。比如 FLASH APP 设置中断向量表偏移量为 0x10000, 调用情况如下:

```
/* 设置中断向量表偏移量为 0x10000 */
sys_nvic_set_vector_table(FLASH_BASE, 0x10000);
```

这是设置 FLASH APP 的情况，SRAM APP 的情况可以参考触摸屏实验_SRAM APP 版本，其具体的调用情况请看到 main 函数。

通过以上两个步骤的设置，我们就可以生成 APP 程序了，只要 APP 程序的 FLASH 和 SRAM 大小不超过我们的设置即可。不过 MDK 默认生成的文件是 .hex 文件，并不方便我们用作 IAP 更新，我们希望生成的文件是 .bin 文件，这样可以方便进行 IAP 升级（至于为什么，请大家自行百度 HEX 和 BIN 文件的区别！）。这里我们通过 MDK 自带的格式转换工具 fromelf.exe，如果安装在 C 盘的默认路径，它的位置是 C:\Keil_v5\ARM\ARMCC\bin\fromelf.exe，来实现 .axf 文件到 .bin 文件的转换。该工具在 MDK 的安装目录\ARM\ARMCC\bin 文件夹里面。

fromelf.exe 转换工具的语法格式为：fromelf[options] input_file。其中 options 有很多选项可以设置，详细使用请参考光盘《mdk 如何生成 bin 文件.doc》。

本实验，我们可以通过在 MDK 点击 Options for Target→User 选项卡，在 After Build/Rebuild 一栏中，勾选 Run #1，我们推荐使用相对地址，在勾选的同一行后的输入框并写入命令行：fromelf --bin -o ..\..\Output\@L.bin ..\..\Output\%L，如图 59.1.7 所示：

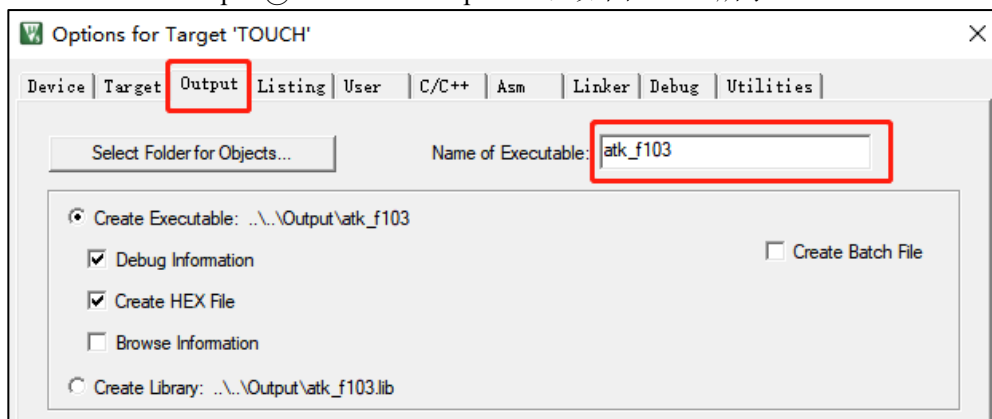


图 59.1.7 设置生成编译结果文件名

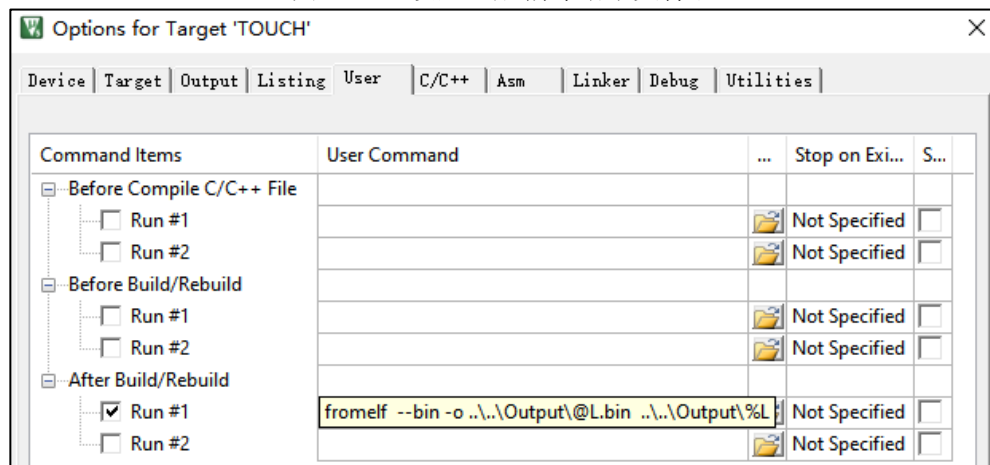


图 59.1.7 MDK 生成 .bin 文件设置方法

通过这一步设置，我们就可以在 MDK 编译成功之后，调用 fromelf.exe，..\..\Output\%L 表示当前编译的链接文件（..\是相对路径，表示上级目录，编译器默认从工程文件*.uvprojx 开始查找，根据我的工程文件 Output 的位置就能明白路径的含义），指令--bin -o ..\..\Output\@L.bin 表示在 Output 目录下生成一个 .bin 文件，@L 在 Keil 的下表示 Output 选项卡下的 Name of Executable 后面的字符串，即在 Output 文件夹下生成一个 atk_f103.bin 文件。在得到 .bin 文件之后，我们只需要将这个 bin 文件传送给单片机，即可执行 IAP 升级。

最后来看看 APP 程序的生成步骤：

1) 设置 APP 程序的起始地址和存储空间大小

对于在 FLASH 里面运行的 APP 程序，我们只需要设置 APP 程序的起始地址，和存储空间大小即可。而对于在 SRAM 里面运行的 APP 程序，我们还需要设置 SRAM 的起始地址和大小。

无论哪种 APP 程序，都需要确保 APP 程序的大小和所占 SRAM 大小不超过我们的设置范围。

2) 设置中断向量表偏移量

此步，通过调用 `sys_nvic_set_vector_table` 函数，实现对中断向量表偏移量的设置。这个偏移量的大小，其实就等于程序起始地址相对于 `0X08000000` 或者 `0X24000000` 的偏移。

3) 设置编译后运行 `fromelf.exe`，生成 `.bin` 文件

通过在 User 选项卡，设置编译后调用 `fromelf.exe`，根据 `.axf` 文件生成 `.bin` 文件，用于 IAP 更新。

以上 3 个步骤，就可以得到一个 `.bin` 的 APP 程序，通过 Bootlader 程序即可实现更新。

59.2 硬件设计

1. 例程功能

本章实验（Bootloader 部分）功能简介：开机的时候先显示提示信息，然后等待串口输入接收 APP 程序（无校验，一次性接收），在串口接收到 APP 程序之后，即可执行 IAP。如果是 SRAM APP，通过按下 KEY0 即可执行这个收到的 SRAM APP 程序。如果是 FLASH APP，则需要先按下 KEY_UP 按键，将串口接收到的 APP 程序存放到 STM32F1 的 FLASH，之后再按 KEY1 即可以执行这个 FLASH APP 程序。通过 KEY2 按键，可以手动清除串口接收到的 APP 程序。DS0 用于指示程序运行状态。

2. 硬件资源

1) LED 灯

DS0 : LED0 - PB5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) 独立按键 : KEY0 - PE4、KEY1 - PE3、WK_UP - PA0

59.3 程序设计

59.3.1 程序流程图

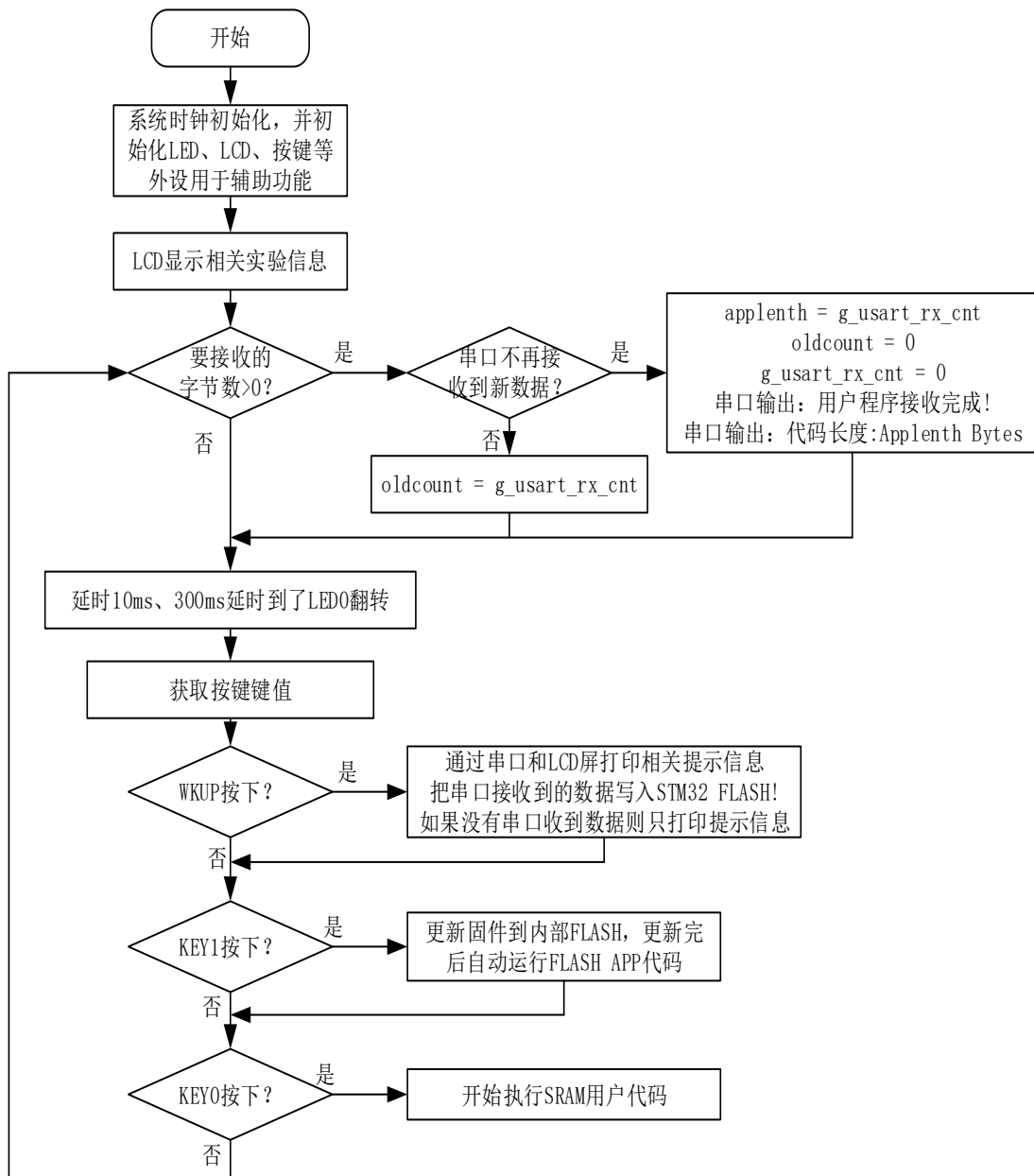


图 59.3.1.1 串口 IAP 实验程序流程图

IAP 我们设置为有按键才跳转的方式, 可以用串口接收不同的 APP, 再根据按键选择跳转到具体的 APP (Flash APP 或者 SRAM APP), 方便我们进行验证和记忆。

59.3.2 程序解析

本实验, 我们总共需要 3 个程序 (1 个 IAP, 2 个 APP):

1、FLASH IAP Bootloader, 起始地址为 0X08000000, 设置为我们用于升级的跳转的程序, 我们将用串口 1 来作数据接收程序, 通过按键功能手动跳转到指定 APP。

2、FLASH APP, 仅使用 STM32 内部 FLASH, 大小为 112KB。本程序使用: 实验 16 USART 调试实验, 作为 FLASH APP 程序 (起始地址为 0X08010000)

3、SRAM APP, 使用 STM32 内部 SRAM, 我们使用 -O2 优化, 生成的 bin 大小为 49KB。

本程序使用：实验 25 触摸屏实验，作为 SRAM APP 程序（起始地址为 0X20001000）。

本章关于 APP 程序的生成和修改比较简单，我们就不细说，请大家结合光盘源码以及 59.1 节的介绍，自行理解。本章程序解析小节仅针对 Bootloader 程序。

1. IAP 程序

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，IAP 的驱动主要包括两个文件：iap.c 和 iap.h。


由于 STM32 芯片的 Flash 的容量一般要比 SRAM 要大，所以我们只编写对 Flash 的写功能和对 MSP 的设置功能以实现程序的跳转。写 STM32 内部 Flash 的功能我们用到 STM32 的 Flash 操作，通过封装之前《实验 33 FLASH 模拟 EEPROM 实验》的驱动，我们实现 IAP 的写 Flash 操作如下：

```
/**
 * @brief      IAP 写入 APP BIN
 * @param      appxaddr : 应用程序的起始地址
 * @param      appbuf   : 应用程序 CODE
 * @param      appsize  : 应用程序大小(字节)
 * @retval     无
 */
void iap_write_appbin(uint32_t appxaddr, uint8_t *appbuf, uint32_t appsize)
{
    uint16_t t;
    uint16_t i = 0;
    uint16_t temp;
    uint32_t fwaddr = appxaddr; /* 当前写入的地址 */
    uint8_t *dfu = appbuf;
    for (t = 0; t < appsize; t += 2)
    {
        temp = (uint16_t)dfu[1] << 8;
        temp |= (uint16_t)dfu[0];
        dfu += 2; /* 偏移 2 个字节 */
        g_iapbuf[i++] = temp;
        if (i == 1024)
        {
            i = 0;
            stmflash_write(fwaddr, g_iapbuf, 1024);
            fwaddr += 2048; /* 偏移 2048 16 = 2 * 8 所以要乘以 2 */
        }
    }
    if (i)
    {
        stmflash_write(fwaddr, g_iapbuf, i); /* 将最后的一些内容字节写进去 */
    }
}
```

在保存了一个完整的 APP 到了对应的位置后，我们需要对栈顶进行检查操作，初步检查程序设置正确再进行跳转。我们以 FlashAPP 为例，用 bin 文件查看工具（A 盘→6，软件资料→1，软件→winhex），可以看到 bin 的内容默认为小端结构，如图 59.3.2.1 所示。

WinHex - [atk_f103.bin]

文件(F) 编辑(E) 搜索(S) 位置(P) 视图(V) 工具(T) 专业工具(I) 选项(O) 窗口(W) 帮助(H)



atk_f103.bin

0x200009A8 0x0801022D

| | Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
|-------------------------------|-----------|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| atk_f103.bin | 00000000 | A8 | 09 | 00 | 20 | 2D | 02 | 01 | 08 | 1B | 20 | 01 | 08 | 13 | 20 | 01 | 08 | |
| F:\OAlien_Work_Dir\2Projects\ | 00000016 | 17 | 20 | 01 | 08 | 5D | 07 | 01 | 08 | 99 | 24 | 01 | 08 | 00 | 00 | 00 | 00 | |
| 文件大小: | 46.7 KB | 00000032 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 41 | 21 | 01 | 08 | |
| | 47,772 字节 | 00000048 | 61 | 07 | 01 | 08 | 00 | 00 | 00 | 00 | 43 | 02 | 01 | 08 | 45 | 02 | 01 | 08 |

图 59.3.2.1 Flash APP 的 bin 文件

我们利用 stm32 的 bin 文件的特性，按 32 位取的数据，开始的第一个字为 SP 的地址，第二个为 Reset_Handler 的地址，我们利用这个特性在跳转前做一个初步的判定，然后设置主堆栈，这部分我们用到 sys.c 下的嵌入汇编函数 sys_msr_msp()，我们实现代码如下：

```
/**
 * @brief      跳转到应用程序段 (执行 APP)
 * @param      appxaddr : 应用程序的起始地址
 *
 * @retval     无
 */
void iap_load_app(uint32_t appxaddr)
{
    if (((*(volatile uint32_t *)appxaddr) & 0x2FFE0000) == 0x20000000)
    { /* 检查栈顶地址是否合法. 可以放在内部 SRAM 共 64KB (0x20000000) */
        /* 用户代码区第二个字为程序开始地址 (复位地址) */
        jump2app = (iapfun) * (volatile uint32_t *) (appxaddr + 4);
        /* 初始化 APP 堆栈指针 (用户代码区的第一个字用于存放栈顶地址) */
        sys_msr_msp(*(volatile uint32_t *)appxaddr);
        /* 跳转到 APP */
        jump2app();
    }
}
```

2. IAP Bootloader 程序

根据我们流程图的设想，所以我们需要用到 LCD、串口、按键和 stm32 内部 Flash 的操作，所以我们通过复制以前的《FLASH 模拟 EEPROM 实验》来修改，重命名为《串口 IAP 实验》，工程内的组重命名为 IAP。

我们需要修改串口接收部分的程序，为了便于测试，我们定义一个大的接收数组 g_usart_rx_buf[USART_REC_LEN]，并保证这个数组能接收并缓存一个完整的 bin 文件，程序中我们定了这个大小为 55KB，因为我们有 SRAM 程序（优化后为 49KB），所以把这部分的数组放用 __attribute__((at(0X20001000))) 直接放到 SRAM 程序的位置，这样接收完成的 SRAM 程序后我们直接跳转就可以了。

```
uint8_t g_usart_rx_buf[USART_REC_LEN] __attribute__((at(0X20001000)));
```

接收的数据处理方法与我们之前的串口处理方式类似，大家查看串口接收处理的源码即可。我们把接收标记的处理放在 main.c 中处理，具体如下：

```
int main(void)
{
    uint8_t t;
    uint8_t key;
    uint32_t oldcount = 0;          /* 老的串口接收数据值 */
    uint32_t applenth = 0;         /* 接收到的 app 代码长度 */
    uint8_t clearflag = 0;

    HAL_Init();                    /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72);                 /* 延时初始化 */
    usart_init(115200);             /* 串口初始化为 115200 */
    led_init();                     /* 初始化 LED */
    lcd_init();                     /* 初始化 LCD */
    key_init();                     /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "IAP TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY UP: Copy APP2FLASH!", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1: Run FLASH APP", RED);
    lcd_show_string(30, 150, 200, 16, 16, "KEY0: Run SRAM APP", RED);

    while (1)
    {
```

```

if (g_usart_rx_cnt)
{
    if (oldcount == g_usart_rx_cnt)
    { /* 新周期内,没有收到任何数据,认为本次数据接收完成 */
        applenth = g_usart_rx_cnt;
        oldcount = 0;
        g_usart_rx_cnt = 0;
        printf("用户程序接收完成!\r\n");
        printf("代码长度:%dBytes\r\n", applenth);
    }
    else oldcount = g_usart_rx_cnt;
}

t++;
delay_ms(100);
if (t == 3)
{
    LED0_TOGGLE();
    t = 0;
    if (clearflag)
    {
        clearflag--;
        if (clearflag == 0)
        {
            lcd_fill(30, 190, 240, 210 + 16, WHITE); /* 清除显示 */
        }
    }
}

key = key_scan(0);
if (key == WKUP_PRES) /* WKUP 按下,更新固件到 FLASH */
{
    if (applenth)
    {
        printf("开始更新固件...\r\n");
        lcd_show_string(30, 190, 200, 16, 16, "Copying APP2FLASH...", BLUE);
        if (((*(volatile uint32_t *) (FLASH_APP1_ADDR + 4)) & 0xFF000000) ==
            0x08000000) /* 判断是否为 0X08XXXXXX */
        { /* 更新 FLASH 代码 */
            iap_write_appbin(FLASH_APP1_ADDR, g_usart_rx_buf, applenth);
            lcd_show_string(30, 190, 200, 16, 16, "Copy APP Succeeded!!", BLUE);
            printf("固件更新完成!\r\n");
        }
        else
        {
            lcd_show_string(30, 190, 200, 16, 16, "Illegal FLASH APP! ", BLUE);
            printf("非 FLASH 应用程序!\r\n");
        }
    }
    else
    {
        printf("没有可以更新的固件!\r\n");
        lcd_show_string(30, 190, 200, 16, 16, "No APP!", BLUE);
    }
    clearflag = 7; /* 标志更新了显示,并且设置 7*300ms 后清除显示 */
}

if (key == KEY1_PRES) /* KEY1 按键按下,运行 FLASH APP 代码 */
{
    if (((*(volatile uint32_t *) (FLASH_APP1_ADDR + 4)) & 0xFF000000) ==
        0x08000000) /* 判断 FLASH 里面是否有 APP,有的话执行 */
    {
        printf("开始执行 FLASH 用户代码!!\r\n\r\n");
        delay_ms(10);
        iap_load_app(FLASH_APP1_ADDR); /* 执行 FLASH APP 代码 */
    }
}

```

```

else
{
    printf("没有可以运行的固件!\r\n");
    lcd_show_string(30, 190, 200, 16, 16, "No APP!", BLUE);
}
clearflag = 7; /* 标志更新了显示,并且设置 7*300ms 后清除显示 */
}
if (key == KEY0_PRES) /* KEY0 按下 */
{
    printf("开始执行 SRAM 用户代码!!\r\n\r\n");
    delay_ms(10);
    if (((*(volatile uint32_t *) (0x20001000 + 4)) & 0xFF000000) ==
        0x20000000) /* 判断是否为 0x20XXXXXX */
    {
        iap_load_app(0x20001000); /* SRAM 地址 */
    }
    else
    {
        printf("非 SRAM 应用程序,无法执行!\r\n");
        lcd_show_string(30, 190, 200, 16, 16, "Illegal SRAM APP!", BLUE);
    }
    clearflag = 7; /* 标志更新了显示,并且设置 7*300ms 后清除显示 */
}
}
}

```

APP 代码我们在这里就不做介绍了,大家可以参考本例程提供的源代码,注意在 `mian` 函数起始处重新设置中断向量表(寄存器 `SCB->VTOR`)的偏移量,否则 APP 无法正常运行,仍以 Flash APP 为例,我们编译通过后执行了 `fromelf.exe` 生成 bin 文件,如图 59.3.2.6 所示:

| << 串口IAP实验 > 实验15 RTC实验_FLASH_APP > Output | | |
|--|--------|--------|
| 名称 | 类型 | 大小 |
| atk_f103.bin | BIN 文件 | 47 KB |
| atk_f103.hex | HEX 文件 | 132 KB |

图 59.3.2.6 多存储段 APP 程序生成多个.bin 文件

59.4 下载验证

将程序下载到开发板后,可以看到 LCD 首先显示一些实验相关的信息,如图 59.4.1 所示:



图 59.4.1 IAP 程序界面

此时,我们可以通过 XCOM, 发送 FLASH APP、SRAM APP 到开发板,我们以 FLASH APP 为例进行演示,如图 59.4.2 所示:

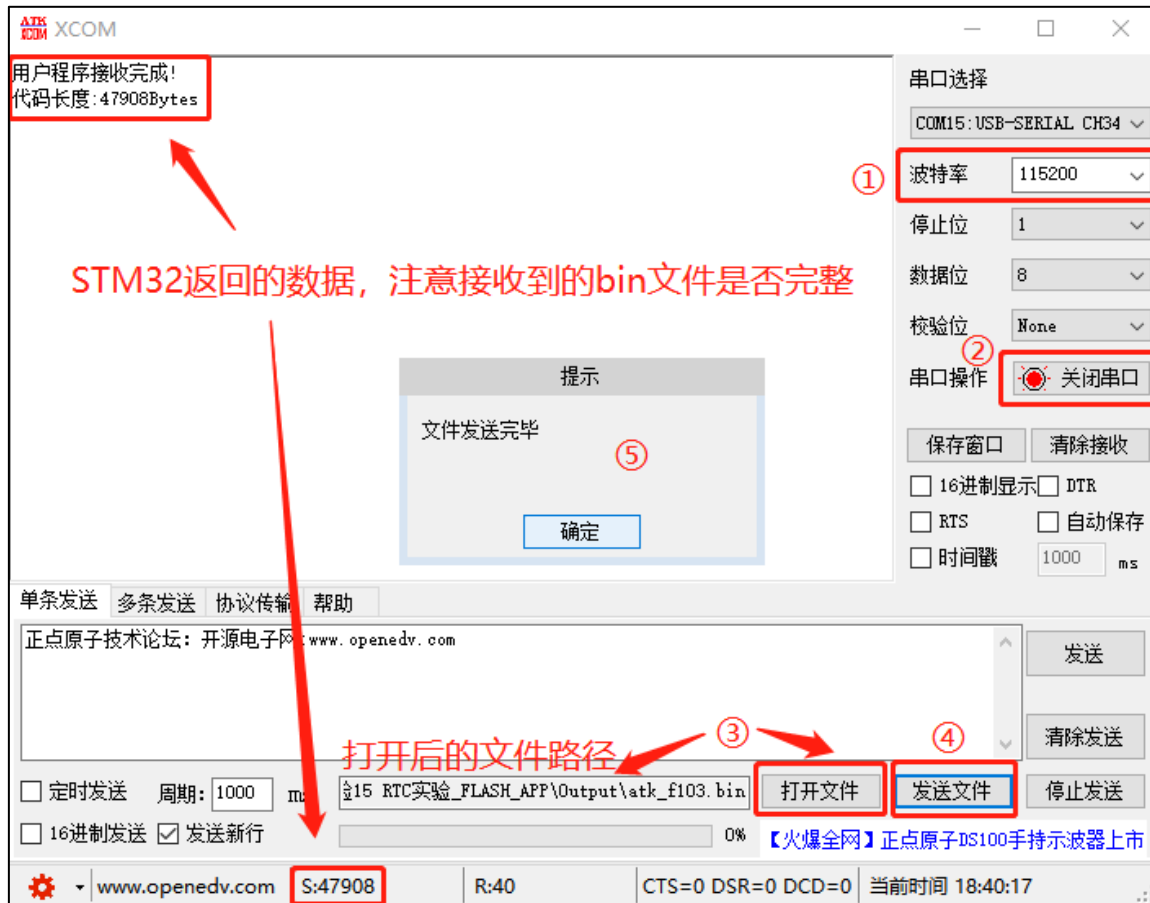


图 59.4.2 串口发送 APP 程序界面

首先找到开发板 USB 转串口的串口号，打开串口（我电脑是 COM15），然后设置波特率为 115200 并打开串口，然后，点击打开文件按钮（图中标号 3 所示），找到 APP 程序生成的 .bin 文件（注意：文件类型得选择所有文件！默认是只打开 txt 文件的），最后点击发送文件（图中标号 4 所示），将 .bin 文件发送给 STM32 开发板，发送完成后，XCOM 会提示文件发送完毕（图中标号 5 所示）。

开发板收到 APP 程序之后会打印提示信息，我们可以根据发送的数据与开发板的提示信息确认开发板接收到的 bin 文件是否完整，我们就可以通过 KEY0/KEY1 运行这个 APP 程序了（如果是 FLASH APP，需要通过 KEY1 将其存入对应 FLASH 区域），此时我们根据程序设计，按下 KEY1 即可执行 FLASH APP 程序，更新 SRAM APP 的过程类似，大家自行测试即可。

第六十章 USB 读卡器实验

本章，我们将向大家介绍如何利用 USB OTG FS 在 STM32F1 开发板实现一个 USB 读卡器。本章分为如下几个部分：

- 60.1 USB 读卡器简介
- 60.2 硬件设计
- 60.3 软件设计
- 60.4 下载验证

60.1 USB 简介

USB，即通用串行总线(Universal Serial Bus)，包括 USB 协议和 USB 硬件两个方面，支持热插拔功能。现在日常生活的很多方面都离不开 USB 的应用，如充电和数据传输等场景。

USB 经过多次修改，1996 年确定了初始规范版本 USB1.0，目前由非盈利组织 USB-IF(<https://www.usb.org>)管理。STM32 自带的 USB 符合 USB2.0 规范,故 2.0 版本仍是本文的重点介绍对象。

60.1.1 USB 简介

USB 本身的知识体系非常复杂，本节只能作一点知识点的引入。本书篇幅有限，不可能在这里详细介绍，想更系统地学习 USB 的知识可以参考《圈圈教你玩 USB》、塞普拉斯提供的《USB101：通用串行总线 2.0 简介》等文献，下面我们一起来看 USB 的简单特性：

- USB 的硬件接口

USB 协议有漫长的发展历程，为的不同的场合和硬件功能而发展出不同的接口：Type-A、Type-B、Type-C，Type-C 规范碰巧是跟着 USB3.1 的规范一起发布的。常见的接口类型列出如图 60.1.1 所示。

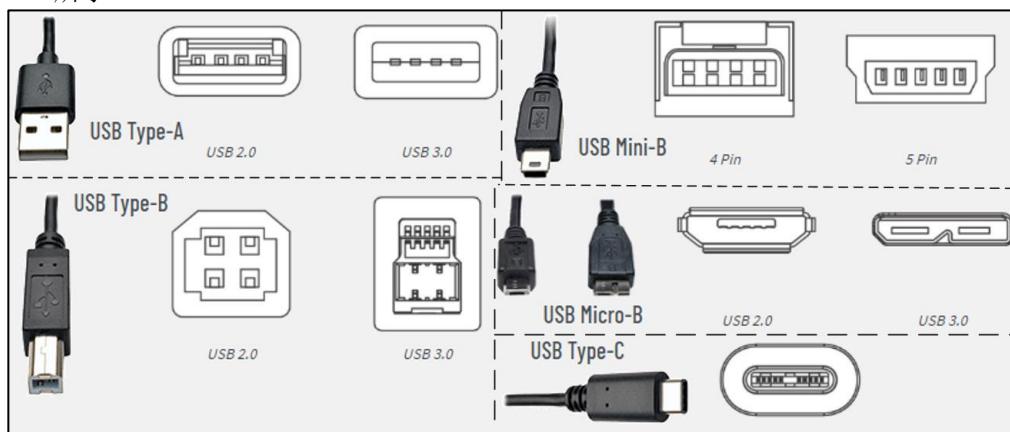


图 60.1.1 常见的 USB 连接器的形状

USB 发展到现在已经有 USB1.0/1.1/2.0/3.x/4 等多个版本。目前用的最多的就是版本 USB1.1 和 USB2.0，USB3.x/USB4 目前也在加速推广。从图中可以发现不同的版本的 USB 接口内的引脚数量是有差异的。USB3.0 以后为了提高速度，采用了更多数量的通讯线，比如同样的是 Type A 接口，USB2.0 版本内部只有四根线，采用半双工式广播式通讯，USB3.0 版本则将通讯线提高到了 9 根，并可以支持全双工非广播式的总线，允许两个单向数据管道分别处理一个单向通信。

USB2.0 常使用四根线：VCC（5V）、GND、D+（3.3V）和 D-（3.3V）（注：五线模式多了一个 DI 脚用于支持 OTG 模式，OTG 为 USB 主机+USB 设备双重角色），其中数据线采用差分电压的方式进行数据传输。在 USB 主机上，D-和 D+都是接了 15K 的电阻到地的，所以在没有设备接入的时候，D+、D-均是低电平。而在 USB 设备中，如果是高速设备，则会在 D+上接一个 1.5K

的电阻到 3.3V，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 3.3V。这样当设备接入主机的时候，主机就可以判断是否有设备接入，并能判断设备是高速设备还是低速设备。

关于 USB 硬件还有更多具体的细节规定，硬件设计时需要严格按照 USB 的器件的使用描述和 USB 标准所规定的参数来设计。

● USB 速度

USB 规范已经为 USB 系统定义了以下四种速度模式：低速(Low-Speed)、全速(Full-Speed)、高速(Hi-Speed)和超高速(SuperSpeedUSB)。接口的速度上限与设备支持的 USB 协议标准和导线长度、阻抗有关，不同协议版本对硬件的传输线数量、阻抗等要求各不相同，各个版本的能达到的理论速度上限对应如图 60.1.2。








| Standard | Also Known As | Logo | Year Introduced | Connector Types | Max. Data Transfer Speed | Cable Length |
|--------------------------------|--|---|----------------------------------|--|--------------------------|--------------|
| USB 1.1 | Basic Speed USB | | 1998 | USB-A USB-B | 12 Mbps | 3 m |
| USB 2.0 | Hi-Speed USB |  | 2000 | USB-A USB-B USB Micro A USB Micro B USB Mini A USB Mini B | 480 Mbps | 5 m |
| USB 3.2 Gen 1 | USB 3.0 USB 3.1 Gen 1 SuperSpeed USB |  | 2008 (USB 3.0) 2013 (USB 3.1) | USB-A USB-B USB Micro B USB-C* | 5 Gbps | 3 m |
| USB 3.2 Gen 2 | USB 3.1 USB 3.1 Gen 2 SuperSpeed+ SuperSpeed USB 10Gbps |  | 2013 (USB 3.1) | USB-A USB-B USB Micro B USB-C* | 10 Gbps | 3 m |
| USB 3.2 Gen 2x2 | USB 3.2 SuperSpeed USB 20Gbps |  | 2017 (USB 3.2) | USB-C* | 20 Gbps | 3 m |
| Thunderbolt™ 2 | |  | 2013 | Mini DisplayPort | 20 Gbps | 3 m |
| Thunderbolt™ 3 | |  | 2015 | USB-C* | 20 Gbps (Passive Cable) | 2 m |
| | | | | | 40 Gbps (Passive Cable) | 0.5 m |
| | | | | | 40 Gbps (Active Cable) | 2 m |
| USB 4 | |  | 2019 | USB-C* | Up to 40 Gbps | 0.8 m |
| Thunderbolt™ 4 | | | 2020 | USB-C* | 40 Gbps | 2 m |

图 60.1.2 USB 协议发展与版本对应的速度

USB 端口和连接器有时会标上颜色，以指示 USB 规格及其支持的功能。这些颜色不是 USB 规范所要求的，并且在设备制造商之间不一致。例如，常见的支持 USB3.0 的 U 盘和电脑等设备使用蓝色指示，英特尔使用橙色指示充电端口等。

● USB 系统

USB 系统主要包括三个部分：控制器 (Host Controller)、集线器 (Hub) 和 USB 设备。

控制器 (Host Controller)，主机一般可以有一个或多个控制器，主要负责执行由控制器驱动程序发出的命令。控制器驱动程序(Host Controller Driver)在控制器与 USB 设备之间建立通信信道。

集线器(Hub)连接到 USB 主机的根集线器，可用于拓展主机可访问的 USB 设备的数量。

USB 设备(USB Device)则是我们常用的如 U 盘，USB 鼠标这类受主机控制的设备。

● USB 通讯

USB 针对主机、集线器和设备制定了严格的协议。概括来讲，通过检测、令牌、传输控制、数据传输等多种方式，定义了主机和从机在系统中的不同职能。USB 系统通过“管道”进行通讯，

有“控制管道”和“数据管道”两种，“控制管道”是双向的，而每个“数据管道”则是单向的，这种关系如图 60.1.3 所示。

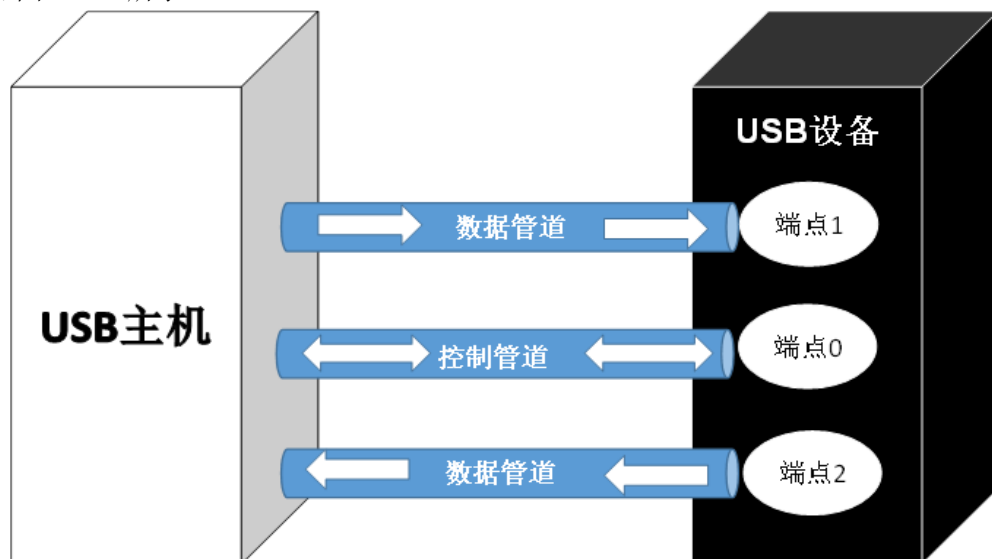


图 60.1.3 USB 管道模型

USB 通讯中的检测和断开总是由主机发起。USB 主机与设备首次进行连接时会交换信息，这一过程叫“USB 枚举”。枚举是设备和主机间进行的信息交换过程，包含用于识别设备的信息。此外，枚举过程主机需要分配设备地址、读取描述符（作为提供有关设备信息的数据结构），并分配和加载设备驱动程序，而从机需要提供相应的描述符使主机知悉如何操作此设备。整个过程需要数秒时间。完成该过程后设备才可以向主机传输数据。数据传输也有规定的三种类型，分别是：IN/读取/上行数据传输、OUT/写入/下行数据传输、控制数据传输。

USB 通过设备端点寻址，在主机和设备间实现信息交流。枚举发生前有一套专用的端点用于与设备进行通信。这些专用的端点统称为控制端点或端点 0，有端点 0 IN 和端点 0 OUT 两个不同的端点，但对开发者来说，它们的构建和运行方式是一样的。每一个 USB 设备都需要支持端点 0。因此，端点 0 不需要使用独立的描述符。除了端点 0 外，特定设备所支持的端点数量将由各自的设计要求决定。简单的设计（如鼠标）可能仅要一个 IN 端点。复杂的设计可能需要多个数据端点。

USB 规定的 4 种数据传输方式也是通过管道进行，分别是控制传输(Control Transfer)、中断传输(Interrupt Transfer)、批量传输或叫块传输(Bulk Transfer)、实时传输或叫同步传输(Isochronous Transfer)，每种模式规定了各自通讯时使用的管道类型。

关于 USB 还有很多更详细的时序和要求，像 USB 描述符、VID/PID 的规定、USB 类设备和调试等，因为 USB2.0 和之后的版本有差异，这里就不再为大家列举了，ST 对 USB2.0 也有专门的培训资料，这部分我们也放到“光盘资料 A 盘→8，STM32 参考资料→2，STM32 USB 学习资料”中了，感兴趣的朋友自行去查阅更的 USB 的相关扩展知识，我们对 USB 的简介就到这里。

60.1.2 STM32F1 的 USB 特性

本节结合《STM32F10xxx 参考手册_V10（中文版）.pdf》的内容，对 STM32F1 的 USB 外设作简介。STM32F1 系列芯片自带了 USB FS(FS, 即全速, 12Mbps)，支持从机(Slave/Device)，但其 USB 与 CAN 不能同时使用，因为硬件共享同一个 SRAM。

STM32F1 的 USB 外设实现了 USB2.0 的接口和 APB1 总线间的接口。它有以下特性：

- 符合 USB2.0 全速设备的技术规范
- 可配置 1 到 8 个 USB 端点
- CRC(循环冗余校验)生成/校验，反向不归零(NRZI)编码/解码和位填充
- 支持同步传输
- 支持批量/同步端点的双缓冲区机制

- 支持 USB 挂起/恢复操作
- 帧锁定时钟脉冲生成

STM32F1 的 USB 外设使用标准的 48Mhz 时钟，允许每个端点有独立的缓冲区，每个端点最大为 512 字节缓冲，最大 16 个单向或 8 个双向端点。USB 的传输格式由硬件完成，状态可以由寄存器标记，可以很大程度上简化我们的程序设计。USB 模块启动时间 $t_{STARTUP}$ 最大为 1us，这个需要在编程时注意。图 60.1.2.1 引用了 STM32F1 的 USB 设备框图，方便我们理解 USB 这个外设。

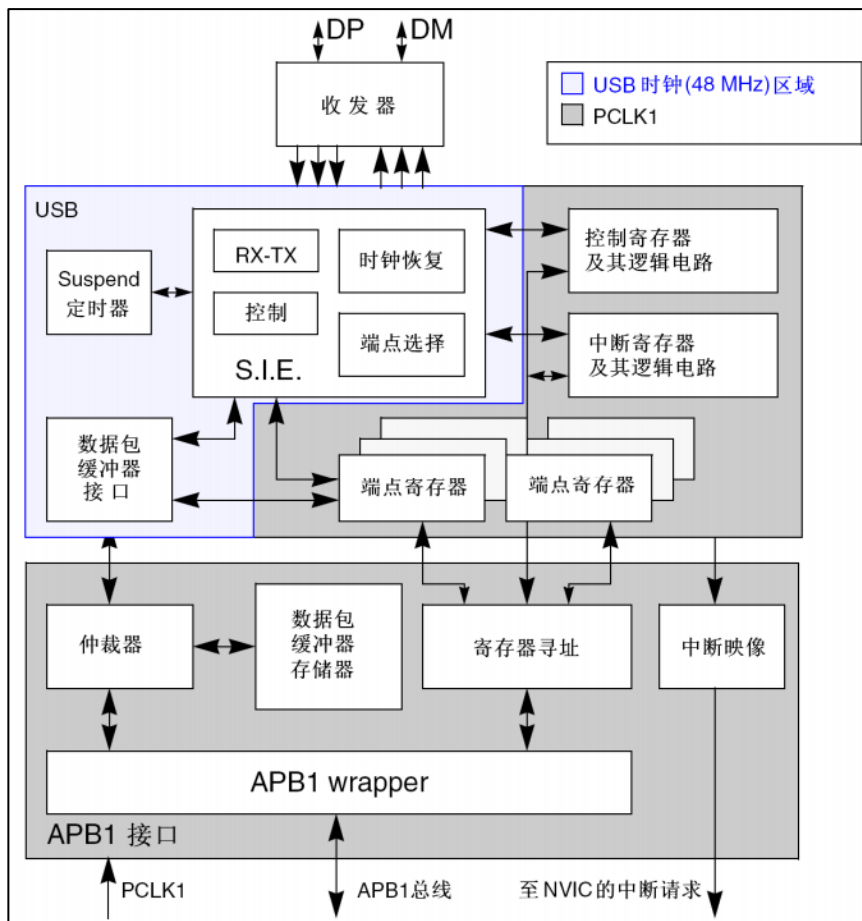


图 60.1.2.1 USB 设备框图

到这里我们已经对 USB 的硬件有了一定的了解，但由于 USB 协议的庞杂性，使得直接编写 USB 驱动的上手难度很大。故在本书编写 USB 驱动的思路是教大家学会移植 ST 官方的 USB 例程并学会使用 USB 驱动库。

ST 官方 Cube 库中提供的官方 USB 协议栈，主要是包含了 USB 内核与 USB 各种类。USB 内核一般是固定的，用户一般不需要修改，但 USB 类，如果用户需要修改或者扩展，比如复合设备或者用户自定义设备，则需要用户自行修改。

USB 协议栈将所有 USB 类都抽象成一个数据结构：USB_ClassTypeDef，USB 内核与 USB 类之间的纽带就是 USB_ClassTypeDef 这个结构体。这个结构体是一个抽象类，定义了一些虚拟函数，比如初始化，反初始化，类请求指令处理函数，端点 0 发送完成，端点 0 接收处理，数据发送完成，数据接收处理，SOF 中断处理，同步传输发送未完成，同步传输接收未完成处理等等；用户在实现自己具体的 USB 类的时候需要将它实例化，USB_ClassTypeDef 结构体是 USB 内核提供给外部定义一个 USB 设备类的窗口，而 USB 类文件实际就是实现这个结构体具体实例化的过程。最后将这个具体实例化的对象注册到 USB 内核的同时，USB 内核与 USB 类也进行了关联。

介绍 USB 时我们说过 USB 有很多的设备类，这一节的 USB 读卡器实际上是一个大容量存储设备(MSC: Mass Storage Class)，本实验实现过程我们需要对 MSC 类实例化，即定义这个类

可以操作的功能，上层应用通过 `USB_D_RegisterClass` 函数，将此对象注册到 `usb_d` 内核，它主要在 `usb_d_msc.c` 源文件中实现它的各个成员函数。

本实验中，STM32 作为设备连接到主机，我们需要使能 USB 外设，以便主机识别到 USB 设备进行扫描，同时我们需要在软件上设计好 USB 枚举所需要的一些设备描述符和注册信息，配置对应的端点以用于 USB 通讯。

这个过程比较复杂，好在 ST 已经提供了实现了类似的例程：通过 USB 来读写 SD 卡（SDIO 方式）和 NAND FLASH，支持 2 个逻辑单元。我们在官方例程的基础上，只需要修改 SD 驱动部分代码，并将对 NAND FLASH 的操作修改为对 SPI FLASH 的操作。只要这两步完成了，剩下的就比较简单了，对底层磁盘的读写，都是在 `usb-storage.c` 文件实现的，所以我们只需要修改文件中的对应接口使之与我们的 SD 卡和 SPI FLASH 对应起来即可。

60.2 硬件设计

1. 例程功能

本节实验功能简介：开机的时候先检测 SD 卡、SPI FLASH 是否存在，如果存在则获取其容量，并显示在 LCD 上（如果不存在，则报错）。之后开始 USB 配置，在配置成功之后，通过 USB 连接线可以在电脑上发现两个可移动磁盘。我们用 LED1 来指示 USB 正在读写，并在液晶上显示出来，同样，我们还是用 DS0 来指示程序正在运行。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 串口 1(PA9/PA10 连接在板载 USB 转串口芯片 CH340 上面)

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4) microSD Card（使用大卡的情况类似，大家可根据自己设计的硬件匹配选择）

5) SPI Flash

6) STM32 自带的 USB Slave 功能，要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。



图 60.2.1 USB SLAVE 接口

除了 USB 接口外，其它外设原理图我们在之前的章节已经介绍过了，这里就不重复介绍了，不清楚的话可以查看本文之前章节的描述或对光盘资料提供的开发板原理图。

60.3 程序设计

由于 USB 驱动的复杂性，如果我们要从零开始编写 USB 驱动，那将是一件相当困难的事情，尤其对于从没了解过 USB 的人来说，周期会更长。不过，ST 提供了 STM32F1 的 USB 驱动库，通过这个库，我们可以很方便的实现我们所要的功能，而不需要详细了解 USB 的整个驱动，大大缩短了我们的开发时间和精力。当能正常驱动起 USB 了，我们再去关联和研究 USB 底层的知识更容易达到事半功倍的效果。

USB 库和相关参考例程在 `en.stm32cube_f1.zip` 里面可以找到，该文件可以在 <http://www.st.com> 网站搜索：cube_f1 找到。不过，我们已经帮大家下载到开发板光盘：8，STM32 参考资料→1，STM32F1xx 固件库→stm32cube_fw_f1_v183.zip。解压可以得到 STM32F1 的固件支持包：STM32Cube_FW_F1_V1.8.0，该文件包含了我们常用的 STM32F1 的嵌入式软件源

码, 如果已经安装过 CubeMX 和 F1 对应的固件, 则这个文件夹在 CubeMX 的资源仓库路径下, 其中就有我们将要使用的 USB 库, 它的位置如图 60.3.1 所示:

- ①. USB 设备驱动库, 从机使用
- ②. USB Host 驱动库, 主机使用
- ③. 与我们使用的芯片型号近似的 ST 开发板的 USB 例程;

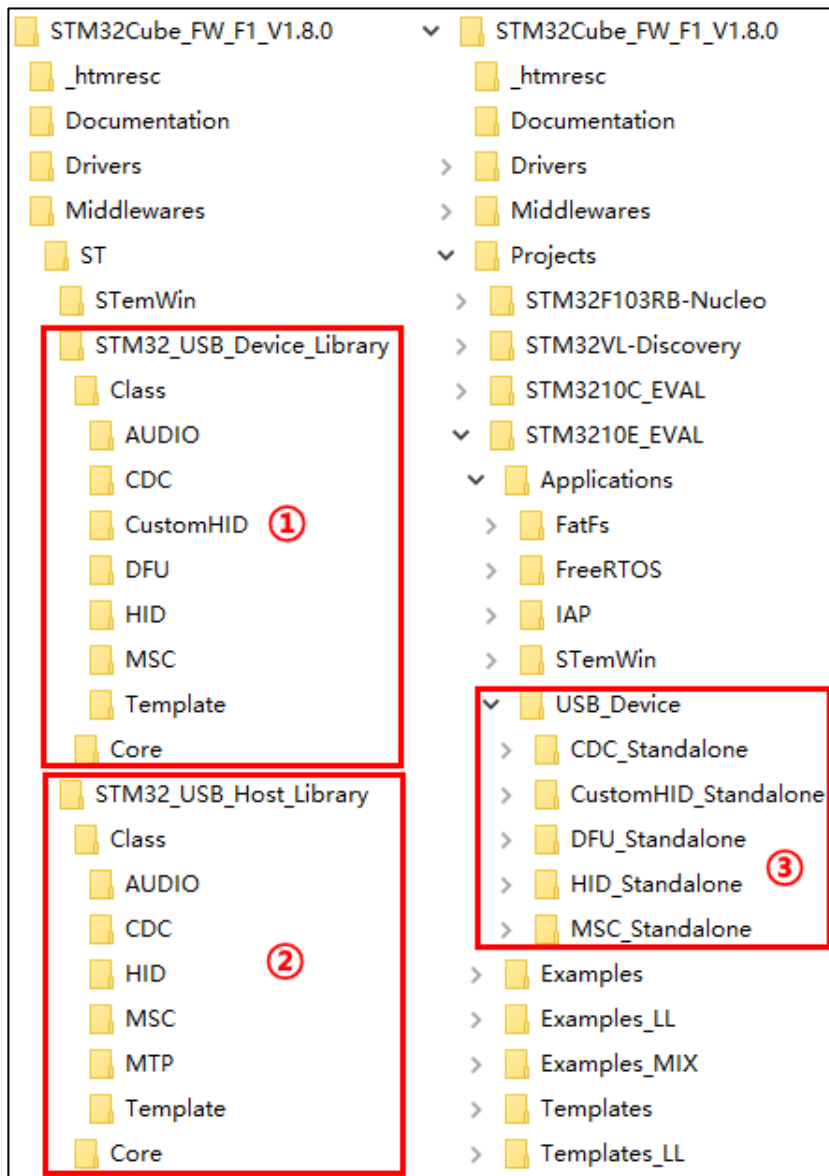


图 60.3.1 ST 以 HAL 库提供的 USB 组件

我们将通过图 60.3.1 序号③的例程, 移植并实现我们自己的 USB Device 设备。读卡器属于 USB 大存储设备, 所以本章要移植的是官方的 MSC_Standalone 例程。我们先打开该例程的 MDK 工程(MDK-ARM 文件夹下), 查看一下其工程结构:

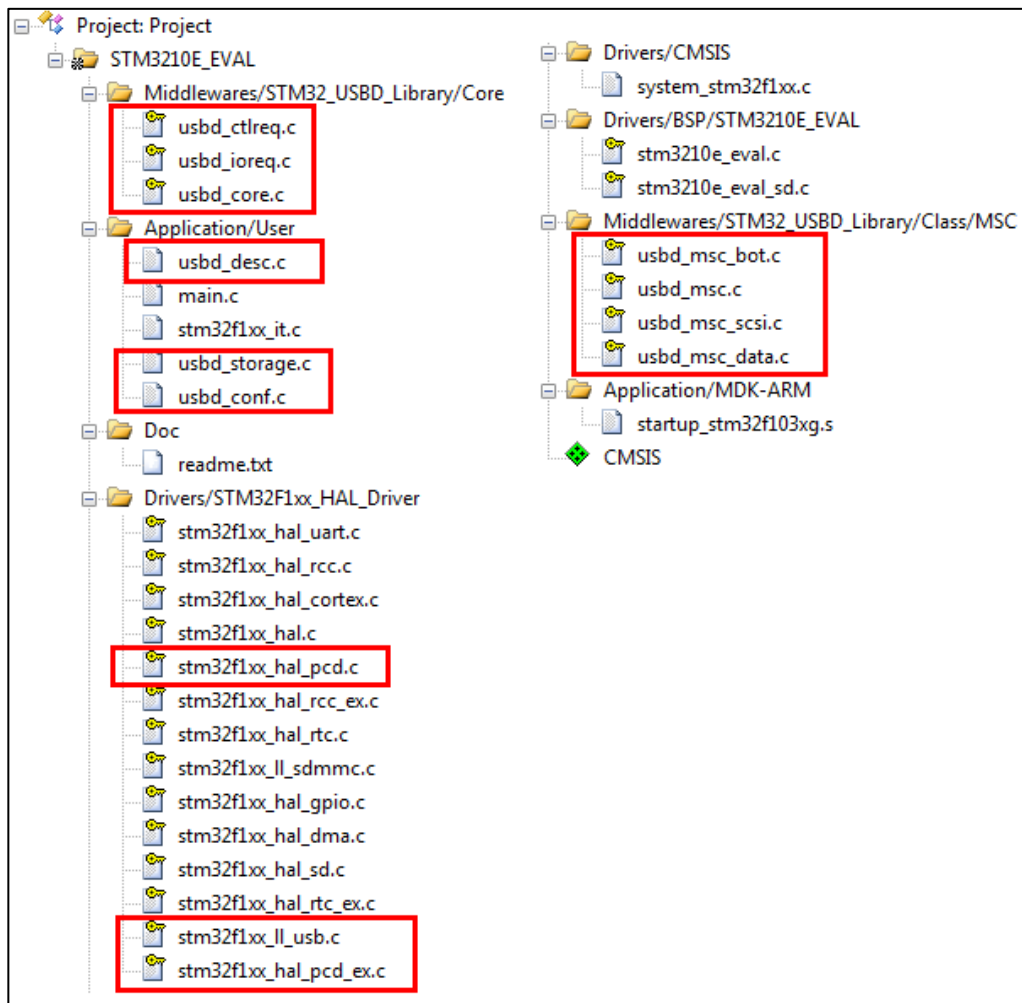


图 60.3.2 ST 官方例程的结构

从工程结构不难找出我们需要的 USB 功能代码，并且这些文件是加了只读属性的，我们移植后需要把只读属性去掉才能进行我们需要的修改。因为要用到 SD 卡和 SPI Flash，为了减少步骤，我们复制之前的 SD 卡实验工程文件夹，重命名为“USB 读卡器实验”，一方面是我们要用到 SD 卡，另一方面是 USB 的端点需要用到动态分配的内存。因为我们并不是所有例程都使用 USB 库驱动，故我们把 USB 作为一个第三方组件放到我们的“Middlewares”文件夹下，我们在该文件夹下新建一个 USB 文件目录，把 USB 相关的全部放到 USB 文件夹下使这部分驱动完全独立，这样可以方便我们以后移植到其它项目中。

首先是 `usbd_core.c`、`usbd_ioreq.c`、`usbd_req.c` 这三个文件，我们查看它们所在的位置发现它们都位于“STM32_USB_Device_Library”文件夹下，所以我们可以直接把该文件夹复制到我们的 USB 文件夹下，后面再考虑精简工程。

接着同样的方法，找到 `usbd_msc_bot.c`、`usbd_msc.c`、`usbd_msc_scsi.c`、`usbd_msc_data.c` 这四个文件，发现他们同样位于“STM32_USB_Device_Library”文件夹，上一步我们已经把整个文件夹复制到我们的工程目录了，所以这步不需要再操作。

接下来的 USB 应用程序 `usbd_desc.c`、`usbd_storage.c`、`usbd_conf.c` 三个文件，源文件和头文件分别位于图 61.3.1 所示 `USB_Device\MSC_Standalone\Src` 和 `USB_Device\MSC_Standalone\Inc` 下，我们在 USB 文件夹下新建一个“USB_APP”文件夹，把它们连同头文件都放到该文件夹的根目录下。



图 60.3.3 USB APP 文件夹下的文件

我们需要添加的文件已经准备好了，接下来我们添加到我们的工程中来。我们按原来的定义，在 MDK 中新建 Middlewares/USB_CORE、Middlewares/USB_CLASS、Middlewares/USB_APP 三个分组，把上面的文件的只读属性去掉后添加到我们的工程中，结果如图，然后把相关的 HAL 库的驱动加到 Drivers/STM32F1xx_HAL_Driver 目录下，如图 60.3.4 所示。

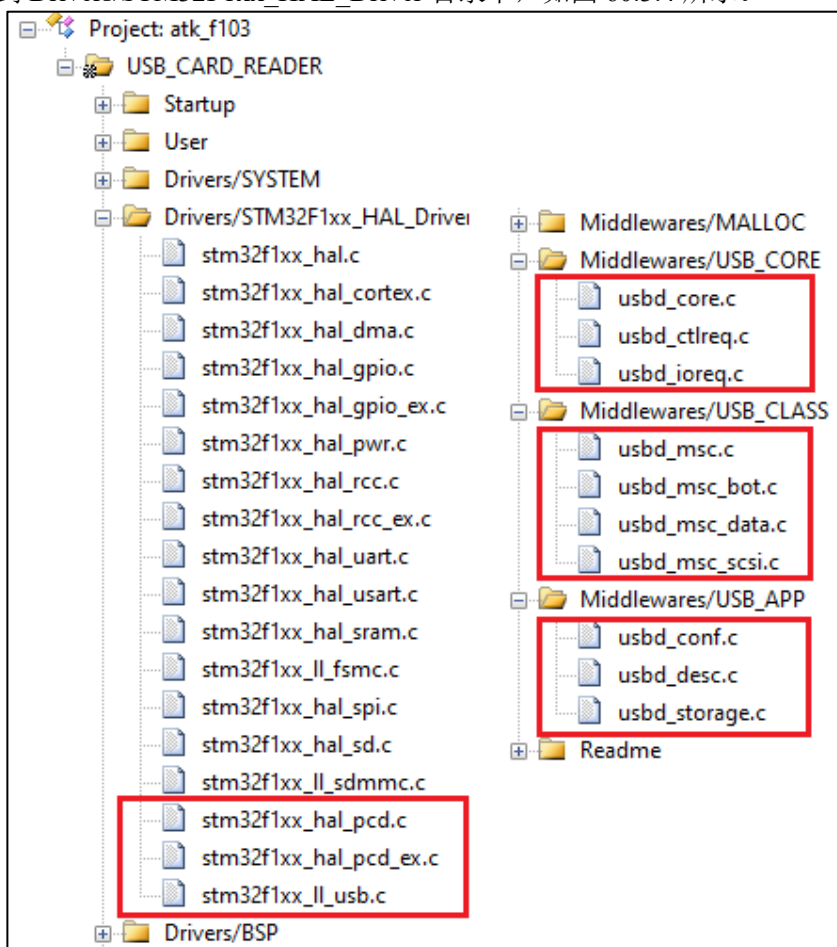


图 60.3.4 在 MDK 中添加需要的代码

为了保持 USB 驱动部分更少的改动，我们添加原有 USB 库的头文件的引用路径，结果如图 60.3.3 所示。

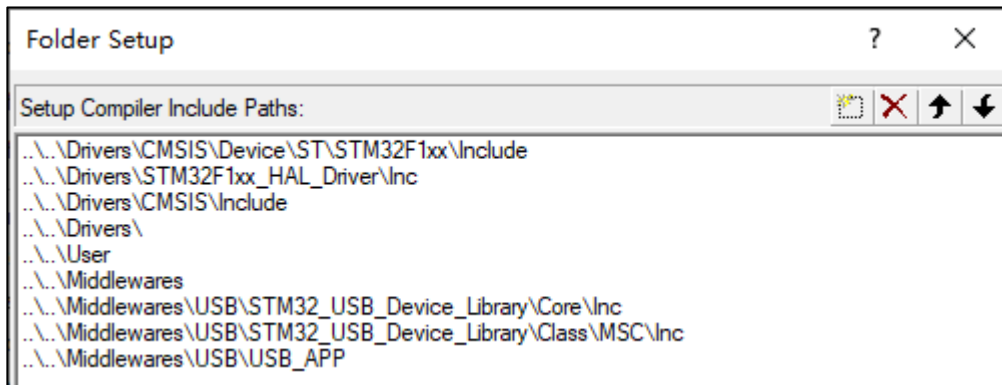


图 60.3.3 在 MDK 中添加 USB 引用的头文件的路径

这时我们直接编译会报错，因为我们没有引用 ST 开发板的 BSP 文件，这时我们还需要修改相关源码以匹配我们的底层的驱动，这部分与我们的开发板相关，我们在程序设计的时候再对应修改。

60.3.1 程序流程图

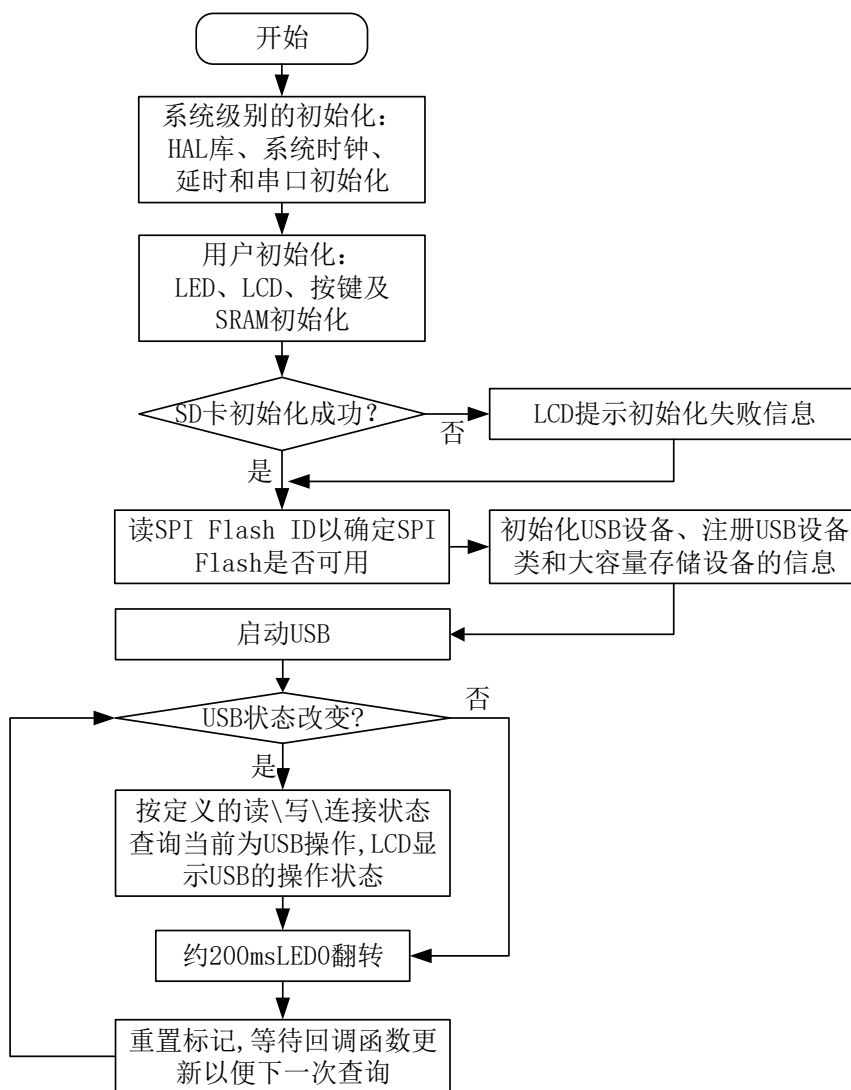


图 60.3.2.1 USB 读卡器程序流程图

我们按流程图编写的初始化顺序，在 STM32 注册 USB 内核，最后通过 USB 的中断和回调函数得到 USB 的操作状态和操作结果，主程序通过查询设定的标记变量的状态值后，在 LCD 上显示对应的 USB 操作状态。

60.3.2 usbd_stroage 驱动

usbd_storage.c/h 需要适配我们的硬件信息。这个函数需要使用到我们的硬件底层驱动，我们需要把我们要对 SD 卡和 SPI Flash 的信息识别和读写操作在这里实现。

```
#include "../MALLOC/malloc.h"
#include "../FATFS/source/diskio.h"
#include "../BSP/SDIO/sdio_sdcard.h"
#include "../BSP/NORFLASH/norflash.h"
```

按照 60.1 的描述，接下来我们来对这几个接口进行补充实现。本章，我们用 FATFS 管理了 2 个磁盘：SD 卡和 SPI FLASH，我们设置 SD_CARD 为 0，EX_FLASH 位为 1，对应到 disk_read/disk_write 函数里面。SD 卡比较好说，但是 SPI FLASH，因为其扇区是 4K 字节大小，我们为了方便设计，强制将其扇区定义为 512 字节，这样带来的好处就是设计使用相对简单，坏处就是擦除次数大增，所以不要随便往 SPI FLASH 里面写数据，非必要最好别写，如果频繁写的话，很容易将 SPI FLASH 写坏。

```
#define SD_CARD    0        /* SD 卡, 卷标为 0 */
#define EX_FLASH   1        /* 外部 qspi flash, 卷标为 1 */
/**
 * 对于 25Q128 FLASH 芯片，我们规定前 12M 给 FATFS 使用，12M 以后
 * 紧跟字库，3 个字库 + UNIGBK.BIN，总大小 3.09M，共占用 15.09M
 * 15.09M 以后的存储空间大家可以随便使用。
 */
#define SPI_FLASH_SECTOR_SIZE 512
#define SPI_FLASH_SECTOR_COUNT 12 * 1024 * 2 /*25Q128, 前 12M 字节给 FATFS 占用*/
#define SPI_FLASH_BLOCK_SIZE 8                /* 每个 BLOCK 有 8 个扇区 */
#define SPI_FLASH_FATFS_BASE 0                /* FATFS 在外部 FLASH 的起始地址从 0 开始 */
```

另外，diskio.c 里面的函数，直接决定了磁盘编号（盘符/卷标）所对应的具体设备，比如，以上代码中，我们就通过 switch 来判断，到底要操作 SD 卡，还是 SPI FLASH，然后，分别执行对应设备的相关操作。以此实现磁盘编号和磁盘的关联。

1. USB_DISK_fops 结构体

要实现大容量存储设备的一个标识信息，作为一个标准接口被封装在 USB 大容量存储设备的操作结构体 USB_DISK_fops 中，它是在 usbd_msc.h 定义好的 USB_StorageTypeDef 类型，基本为函数指针和数据指针，我们需要为这些 USB 操作实现跟我们硬件相关的底层代码，它们会在 USB 枚举过程中被调用。

```
USB_StorageTypeDef USB_DISK_fops =
{
    STORAGE_Init,                /* 外设初始化 */
    STORAGE_GetCapacity,         /* 获取容量 */
    STORAGE_IsReady,             /* 检查设备就绪状态 */
    STORAGE_IsWriteProtected,    /* 查询设备读保护状态 */
    STORAGE_Read,                /* 读操作 */
    STORAGE_Write,               /* 写操作 */
    STORAGE_GetMaxLun,           /* 获取磁盘数 */
    (int8_t *)STORAGE_Inquirydata, /* 设备信息标识 */
};
```

其中 STORAGE_Inquirydata 表示设备的基本描述信息，它会在 USB 设备注册时被 SCSI_Inquiry() 函数调用，大小与宏 STANDARD_INQUIRY_DATA_LEN 表示的数值相同，为 36 个字节，大容量存储设备对这个设备信息进行的格式有了标准的规定，具体如下面的表 60.3.2.1 所示，其中的设备类型也有作了相应的规定，由于我们使用的是存储设备，所以前面的信息保持一致即可：

| 字节编号 \ 位 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|---------------------|---|----|------|----|---|---|---|
| 0 | 保留 | | | 设备类型 | | | | |
| 1 | 0:不可移除设备 1:可移除设备 | | 保留 | | | | | |
| 2 | 保留 | | | | | | | |
| 3 | 保留 | | | | 保留 | | | |
| 4 | 剩余长度 | | | | | | | |
| 5-7 | 保留 | | | | | | | |
| 8-15 | 设备厂商 ID | | | | | | | |
| 16-31 | 产品 ID | | | | | | | |
| 32-35 | 产品版本号 | | | | | | | |

表 60.3.2.1 大容量存储设备信息

USB 大容量存储设备的每个函数操作时都会对应一个卷标号，需要支持多个设备时，保证每个设备的描述信息都为完整合法的 36 个字节即可。我们模仿 ST 的例程，修改为支持两个设备，则有如下的设备信息代码：

```
/* USB Mass storage 标准查询数据(每个 lun 占 36 字节) */
const int8_t STORAGE_Inquirydata[] =
{
    /* LUN 0 */
    0x00,
    0x80,
    0x02,
    0x02,
    (STANDARD_INQUIRY_DATA_LEN - 4),
    0x00,
    0x00,
    0x00,
    /* Vendor Identification */
    'A', 'L', 'I', 'E', 'N', 'T', 'E', 'K', ' ', /* 9 字节 */
    /* Product Identification */
    'S', 'P', 'I', ' ', 'F', 'l', 'a', 's', 'h', /* 15 字节 */
    ' ', 'D', 'i', 's', 'k', ' ',
    /* Product Revision Level */
    '1', '.', '0', ' ', /* 4 字节 */

    /* LUN 1 */
    0x00,
    0x80,
    0x02,
    0x02,
    (STANDARD_INQUIRY_DATA_LEN - 4),
    0x00,
    0x00,
    0x00,
    /* Vendor Identification */
    'A', 'L', 'I', 'E', 'N', 'T', 'E', 'K', ' ', /* 9 字节 */
    /* Product Identification */
    'S', 'D', ' ', 'F', 'l', 'a', 's', 'h', ' ', /* 15 字节 */
    'D', 'i', 's', 'k', ' ',
    /* Product Revision Level */
    '1', '.', '0', ' ', /* 4 字节 */
};
```

2. STORAGE_Init 函数

需要实现 USB 大容量设备的操作接口之一，会被 MSC_BOT_Init()接口调用。这里是初始化板载硬件的接口的操作，我们把 SD 卡和 SPI Flash 对应的初始化放到这个函数中：

```
int8_t STORAGE_Init (uint8_t lun)
```

● 函数描述:

初始存储设备的硬件接口，我们这里有两个设备，所以把 SPI Flash 和 SD 卡的设备信息存放在这个位置。

● 函数形参:

形参 1 lun 是存储设备的卷标，从 0 开始，有多个设备时应该与 STORAGE_Inquirydata 中定义的设备顺序一致。

我们定义了两个设备，则初始化函数的代码修改如下：

```
/**
 * @brief      初始化存储设备
 * @param      lun      : 逻辑单元编号
 * @arg        0, SD 卡
 * @arg        1, SPI FLASH
 * @retval     操作结果
 * @arg        0      , 成功
 * @arg        其他   , 错误代码
 */
int8_t STORAGE_Init (uint8_t lun)
{
    uint8_t res = 0;

    switch (lun)
    {
        case 0: /* SPI FLASH */
            norflash_init();
            break;
        case 1: /* SD 卡 */
            res = sd_init();
            break;
    }

    return res;
}
```

● 函数返回值:

返回硬件初始化结果：0：成功， 其它：错误或失败。

3. STORAGE_GetCapacity 函数

STORAGE_GetCapacity 从名字就可以知道这个接口需要返回存储器设备的存储容量，并将参数返回给对应的指针：

```
int8_t STORAGE_GetCapacity (uint8_t lun, uint32_t *block_num,
                             uint16_t *block_size)
```

● 函数描述:

获取指定标号的存储设备的容量信息。

● 函数形参:

形参 1 lun 是存储设备的卷标，从 0 开始，有多个设备时应该与 STORAGE_Inquirydata 中定义的设备顺序一致。

代码实现如下：

```
int8_t STORAGE_GetCapacity (uint8_t lun, uint32_t *block_num, uint16_t
*block_size)
{
    switch (lun)
    {
        case 0: /* SPI FLASH */
            *block_size = 512;
            *block_num = (1024 * 1024 * 12)/512; /*SPI FLASH 的 12MB 字节,文件系统用 */
            break;

        case 1: /* SD 卡 */
            *block_size = 512;
            *block_num = ((long long)g_sdcard_handler.SdCard.BlockNbr *
g_sdcard_handler.SdCard.BlockSize)/512;
```



```
        break;
    }
    return 0;
}
```

● 函数返回值:

返回硬件初始化结果: 0: 成功, 其它: 错误或失败。

4. STORAGE_IsReady 函数

STORAGE_IsReady 用于查询设备的就绪状态, 我们的代码比较简单, 只为了演示功能部分, 所以直接返回就绪状态即可。

```
int8_t STORAGE_IsReady (uint8_t lun)
```

为了标识 USB 的操作状态, 我们加入一个全局的标记来指示 USB 的操作状态, 也方便我们在其它位置查询到当前的 USB 执行结果:

```
/* 自己定义的一个标记 USB 状态的寄存器, 方便判断 USB 状态
 * bit0 : 表示电脑正在向 SD 卡写入数据
 * bit1 : 表示电脑正从 SD 卡读出数据
 * bit2 : SD 卡写数据错误标志位
 * bit3 : SD 卡读数据错误标志位
 * bit4 : 1, 表示电脑有轮询操作 (表明连接还保持着)
 */
volatile uint8_t g_usb_state_reg = 0;
```

● 函数描述:

初始化指定编号的磁盘, 磁盘所指定的存储区。

● 函数形参:

形参 1 lun 是存储设备的卷标, 从 0 开始, 有多个设备时应该与 STORAGE_Inquirydata 中定义的设备顺序一致。

```
/**
 * @brief      查看存储设备是否就绪
 * @param      lun      : 逻辑单元编号
 * @arg        0, SD 卡
 * @arg        1, SPI FLASH
 * @retval     就绪状态
 * @arg        0 , 就绪
 * @arg        其他 , 未就绪
 */
int8_t STORAGE_IsReady (uint8_t lun)
{
    g_usb_state_reg |= 0x10;    /* 标记轮询 */
    return 0;
}
```

● 函数返回值:

返回 0 表示设备就绪, 返回其它则未就绪。

5. STORAGE_Read 函数

STORAGE_Read 实现 USB 对物理设备的读操作, 其声明如下:

```
int8_t STORAGE_Read (uint8_t lun, uint8_t *buf, uint32_t blk_addr,
                    uint16_t blk_len)
```

● 函数描述:

初始化指定编号的磁盘, 磁盘所指定的存储区。

● 函数形参:

形参 1 lun 是存储设备的卷标, 从 0 开始, 有多个设备时应该与 STORAGE_Inquirydata 中定义的设备顺序一致。

形参 2 buf 为要写入的数据的缓冲区指针, 为字节类型。

形参 3 blk_addr 为要读数据的起始地址, 对应为 SPI Flash 和 SD 卡的扇区地址。

形参 4 blk_len 表示的是要读取到 buf 的字节数。

代码实现如下:

```
int8_t STORAGE_Read (uint8_t lun, uint8_t *buf, uint32_t blk_addr,
                    uint16_t blk_len)
```

```

{
    int8_t res = 0;
    g_usb_state_reg |= 0X02;    /* 标记正在读数据 */
    switch (lun)
    {
        case 0: /* SPI FLASH */
            norflash_read(buf, USB_STORAGE_FLASH_BASE + blk_addr * 512,
                           blk_len * 512);
            break;
        case 1: /* SD 卡 */
            res = sd_read_disk(buf, blk_addr, blk_len);
            break;
    }
    if (res)
    {
        printf("rerr:%d,%d\r\n", lun, res);
        g_usb_state_reg |= 0X08;    /* 读错误! */
    }
    return res;
}

```

● 函数返回值:

返回 0 表示设备就绪, 返回其它则未就绪。

usb_storage 的代码就讲到这里, 其它的几个函数的实现方法类似, 大家参考光盘的代码即可, 代码里也有相应的注释, 大家根据自己的硬件情况实现即可。

60.3.3 usbd_conf 驱动

usbd_conf.c/h 主要实现 USB 的硬件初始化和中断操作, 当 USB 状态机处理完不同事务的时候, 会调用这些回调函数, 我们通过这些回调函数, 就可以知道 USB 当前状态, 比如: 是否枚举成功了? 是否连接上了? 是否断开了? 根据这些状态, 用户应用程序可以执行不同操作, 完成特定功能。

USBD_Init()接口会调用 ST 芯片的 USBD_LL_Init()初始化函数, HAL_PCD_MspInit()函数在这个时候被调用, 所以我们需要实现 USBD_LL_Init()和 HAL_PCD_MspInit 函数, 使之与我们开发板上的 USB 接口对应。

1. USBD_LL_Ini 函数

这个部分配置 USB 的一些基础参数, 如 USB 速度, 端点数, 通讯 fifo 的分配等, ST 的例程中已经帮我们实现好了这个函数, 因为我们不需要添加新的特性, 所以这里直接沿用原来的配置即可。

```

USBD_StatusTypeDef USBD_LL_Init(USBD_HandleTypeDef *pdev)
{
    /* 设置 LL 驱动相关参数 */
    hpcd.Instance = USB;                /* 使用 USB */
    hpcd.Init.dev_endpoints = 8;        /* 端点数为 8 */
    hpcd.Init.phy_iface = PCD_PHY_EMBEDDED; /* 使用内部 PHY */
    hpcd.Init.speed = PCD_SPEED_FULL;    /* USB 全速 (12Mbps) */
    hpcd.Init.low_power_enable = 0;      /* 不使能低功耗模式 */
    hpcd.pData = pdev;                  /* hpcd 的 pData 指向 pdev */
    pdev->pData = &hpcd;                /* pdev 的 pData 指向 hpcd */
    HAL_PCD_Init((PCD_HandleTypeDef *) pdev->pData); /* 初始化 LL 驱动 */

    HAL_PCDEx_PMAConfig(pdev->pData, 0x00, PCD_SNG_BUF, 0x18);
    HAL_PCDEx_PMAConfig(pdev->pData, 0x80, PCD_SNG_BUF, 0x58);
    HAL_PCDEx_PMAConfig(pdev->pData, MSC_EPIN_ADDR, PCD_SNG_BUF, 0x98);
    HAL_PCDEx_PMAConfig(pdev->pData, MSC_EPOUT_ADDR, PCD_SNG_BUF, 0xD8);

    return USBD_OK;
}

```

2. HAL_PCD_MspInit 函数

HAL_PCD_MspInit 中我们需要开启 USB 的引脚的复用功能, 我们使用的是 PA11/PA12, 并开

启 USB 中断:

```
/**
 * @brief      初始化 PCD MSP
 * @note      这是一个回调函数，在 stm32f1xx_hal_pcd.c 里面调用
 * @param      hpcd      : PCD 结构体指针
 * @retval     无
 */
void HAL_PCD_MspInit(PCD_HandleTypeDef *hpcd)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    __HAL_RCC_GPIOA_CLK_ENABLE();          /* 使能 PORTA 时钟 */
    __HAL_RCC_USB_CLK_ENABLE();            /* 使能 USB 时钟 */

    /* PA11/PA12,复用为(USB DM/DP)功 */
    GPIO_InitStructure.Pin = (GPIO_PIN_11 | GPIO_PIN_12);
    GPIO_InitStructure.Mode = GPIO_MODE_AF_INPUT;
    GPIO_InitStructure.Pull = GPIO_PULLUP;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

    HAL_NVIC_SetPriority(USB_LP_CAN1_RX0_IRQn, 0, 3); /* 抢占 0, 子优先 3, 组 2 */
    HAL_NVIC_EnableIRQ(USB_LP_CAN1_RX0_IRQn);        /* 开启 USB 中断 */
}
```

由于开启了中断，我们还需要定义 USB 的中断处理函数，类似我们之前实验的中断处理函数定义，直接调用 HAL 库的 USB 中断处理接口，然后再实现对应的回调函数

```
/**
 * @brief      USB OTG 中断服务函数
 * @note      处理所有 USB 中断
 * @param      无
 * @retval     无
 */
void USB_LP_CAN1_RX0_IRQHandler(void)
{
    HAL_PCD_IRQHandler(&hpcd);
}
```

HAL 库为我们提供了回调事件的接口。可以监测 USB 在通讯中的各个状态，为了方便，我们加入了一个全局变量 `g_device_state` 标记 USB 的运行状态，以方便程序访问，具体可以查看我们添加的回调函数的代码。

3. USB 内存管理函数

ST 的例程提供了静态和动态两种内存分配方式，注意到 USB 默认使用堆来分配 USB 设备的工作内存，我不使用默认的堆，不去修改 `startup_stm32f103xe.s` 中的堆大小，我们对应地要把默认的内存管理函数变成我们自己的内存管理方式，在 `usbd_conf.h` 中把以下宏重定义：

```
/* Memory management macros */
#define USBBD_malloc(x)          mymalloc(SRAMIN, x)
#define USBBD_free(x)            myfree(SRAMIN, x)
```

到此就基本移植完成了，下面我们开始编写我们的 `main` 函数来测试移植效果。

60.3.4. main.c 代码

在 `main.c` 就比较简单了，按照我们的流程图的思路编写即可，我们初始化探按键，LCD 和 LED、SRAM 等外设辅助程序显示。

最后，我们编写的 `main` 函数如下：

```
USBD_HandleTypeDef USBD_Device;          /* USB Device 处理结构体 */
extern volatile uint8_t g_usb_state_reg;  /* USB 状态 */
extern volatile uint8_t g_device_state;   /* USB 连接 情况 */

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

```
uint8_t tct = 0;
uint8_t usb_sta;
uint8_t device_sta;
uint16_t id;

HAL_Init(); /* 初始化 HAL 库 */
sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72Mhz */
delay_init(72); /* 延时初始化 */
usart_init(115200); /* 串口初始化为 115200 */
led_init(); /* 初始化 LED */
lcd_init(); /* 初始化 LCD */
key_init(); /* 初始化按键 */
sram_init(); /* SRAM 初始化 */
norflash_init(); /* 初始化 NOR FLASH */
my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "USB Card Reader TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

if (sd_init()) /* 初始化 SD 卡 */
{ /* 检测 SD 卡错误 */
    lcd_show_string(30, 110, 200, 16, 16, "SD Card Error!", RED);
}
else /* SD 卡正常 */
{ /* 显示 SD 卡容量 */
    lcd_show_string(30, 110, 200, 16, 16, "SD Card Size:    MB", RED);
    lcd_show_num(134, 110, SD_TOTAL_SIZE_MB(&g_sdcard_handler), 5, 16, RED);
}

id = norflash_read_id();
if ((id == 0) || (id == 0xFFFF))
{ /* 检测 W25Q128/NM25Q 错误 */
    lcd_show_string(30, 110, 200, 16, 16, "BY25Q128 Error!", RED);
}
else /* SPI FLASH 正常 */
{
    lcd_show_string(30, 130, 200, 16, 16, "SPI FLASH Size:7.25MB", RED);
}

usb_dport_config(0); /* USB 先断开 */
delay_ms(500);
usb_dport_config(1); /* USB 再次连接 */
delay_ms(500);
/* 提示正在建立连接 */
lcd_show_string(30, 170, 200, 16, 16, "USB Connecting...", RED);
USBD_Init(&USBD_Device, &MSC_Desc, 0); /* 初始化 USB */
USBD_RegisterClass(&USBD_Device, USBD_MSC_CLASS); /* 添加类 */
USBD_MSC_RegisterStorage(&USBD_Device, &USBD_DISK_fops); /* 添加 MSC 类回调函数 */
USBD_Start(&USBD_Device); /* 开启 USB */
delay_ms(1800);

while (1)
{
    delay_ms(1);

    if (usb_sta != g_usb_state_reg) /* 状态改变了 */
    {
        lcd_fill(30, 190, 240, 210 + 16, WHITE); /* 清除显示 */
        if (g_usb_state_reg & 0x01) /* 正在写 */
        { /* 提示 USB 正在写入数据 */
            lcd_show_string(30, 190, 240, 210 + 16, "Writing Data", RED);
        }
    }
}
```

```

        LED1(0);
        lcd_show_string(30, 190, 200, 16, 16, "USB Writing...", RED);
    }

    if (g_usb_state_reg & 0x02)    /* 正在读 */
    { /* 提示 USB 正在读出数据 */
        LED1(0);
        lcd_show_string(30, 190, 200, 16, 16, "USB Reading...", RED);
    }

    if (g_usb_state_reg & 0x04)
    { /* 提示写入错误 */
        lcd_show_string(30, 210, 200, 16, 16, "USB Write Err ", RED);
    }
    else
    {
        lcd_fill(30, 210, 240, 230 + 16, WHITE); /* 清除显示 */
    }

    if (g_usb_state_reg & 0x08)
    { /* 提示读出错误 */
        lcd_show_string(30, 230, 200, 16, 16, "USB Read Err ", RED);
    }
    else
    {
        lcd_fill(30, 230, 240, 250 + 16, WHITE); /* 清除显示 */
    }

    usb_sta = g_usb_state_reg;                                /* 记录最后的状态 */
}

if (device_sta != g_device_state)
{
    if (g_device_state == 1)
    { /* 提示 USB 连接已经建立 */
        lcd_show_string(30, 170, 200, 16, 16, "USB Connected ", RED);
    }
    else
    { /* 提示 USB 被拔出了 */
        lcd_show_string(30, 170, 200, 16, 16, "USB DisConnected ", RED);
    }

    device_sta = g_device_state;
}
tct++;
if (tct == 200)
{
    tct = 0;
    LED1(1);                                /* 关闭 LED1 */
    LED0_TOGGLE();                            /* LED0 闪烁 */

    if (g_usb_state_reg & 0x10)
    {
        offline_cnt = 0;                    /* USB 连接了,则清除 offline 计数器 */
        g_device_state = 1;
    }
    else                                    /* 没有得到轮询 */
    {
        offline_cnt++;

        if (offline_cnt > 100)
        {
            g_device_state = 0; /* 20s 内没收到在线标记,代表 USB 被拔出了 */
        }
    }
}

```

```

    }

    g_usb_state_reg = 0;
}
}
}

```

我们通过前面的移植，我们已经能正常使用 USB 读卡器的功能了，但我们发现识别成功后，如果 USB 设备发生故障（比如我们复位开发板），这时主机会认为 USB 设备错误自动移除 USB 的连接，被主机断开后的设备不会再被扫描，所以我们设定 `usbd_port_config()` 函数用于控制 USB 的 IO，在 `while` 循环前，我们先把 USB 的 IO 复用成普通 IO，再复用成 USB 的引脚，以保证每次复位 USB 线都有一个物理“断开”，重新激活 USB 的枚举过程。

```

/**
 * @brief      USB 接口配置
 * @note      使能/关闭 USB 接口，以便每次启动都可以正常
 *            连接 USB
 * @param      state    : 接口状态
 * @arg        0, 断开 USB 连接
 * @arg        1, 使能 USB 连接
 * @retval     无
 */
void usbd_port_config(uint8_t state)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    __HAL_RCC_GPIOA_CLK_ENABLE(); /* 使能 PORTA 时钟 */

    if (state)
    {
        USB->CNTR &= ~(1 << 1); /* PWDN = 0, 退出断电模式 */

        /* PA11 引脚模式设置, 复用功能 */
        GPIO_InitStructure.Pin = GPIO_PIN_11;
        GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStructure.Pull = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

        /* PA12 引脚模式设置, 复用功能 */
        GPIO_InitStructure.Pin = GPIO_PIN_12;
        GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStructure.Pull = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
    }
    else
    {
        USB->CNTR |= 1 << 1; /* PWDN = 1, 进入断电模式 */
        /* PA11 引脚模式设置, 推挽输出 */
        GPIO_InitStructure.Pin = GPIO_PIN_11;
        GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStructure.Pull = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

        /* PA12 引脚模式设置, 推挽输出 */
        GPIO_InitStructure.Pin = GPIO_PIN_12;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11, GPIO_PIN_RESET); /* PA11 = 0 */
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, GPIO_PIN_RESET); /* PA12 = 0 */
    }
}

```


60.5 下载验证

在代码编译成功之后, 我们通过下载代码战舰 STM32F103 开发板上, 在 USB 配置成功后 (假设已经插入 SD 卡, 注意: USB 数据线, 要插在开发板的 USB_SLAVE 口! 而不是 USB_UART 端口, 且 P9 必须用跳线帽连接 PA11 和 D-以及 PA12 和 D+!!), LCD 显示效果如图 60.5.1 所示:

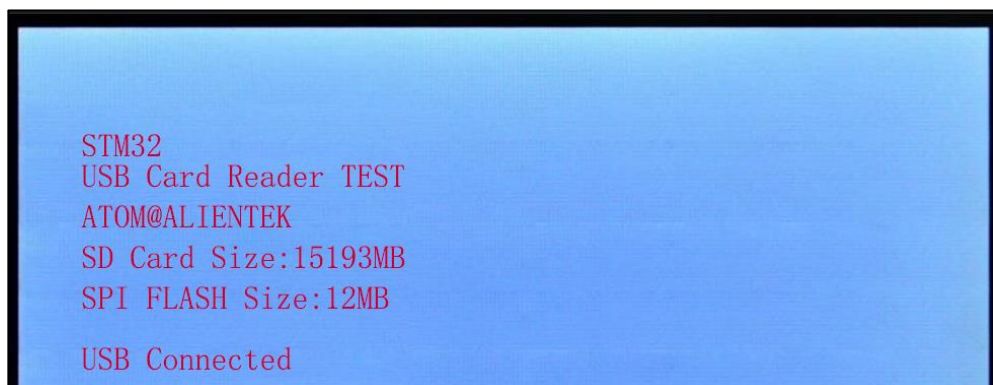


图 60.5.1 USB 连接成功

USB 识别成功后, 电脑提示发现新硬件, 并自动安装驱动, 如图 60.5.2 所示:

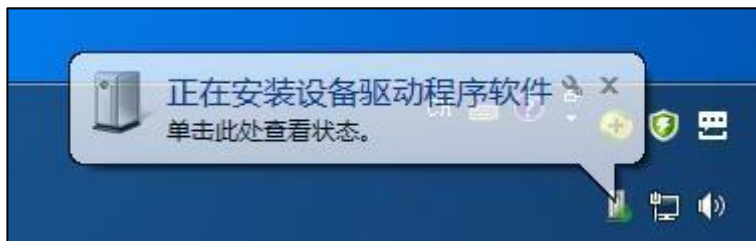


图 60.5.2 USB 读卡器被电脑找到

等 USB 配置成功后, DS1 不亮, DS0 闪烁, 并且在电脑上可以看到我们的磁盘, 如图 60.5.3 所示:

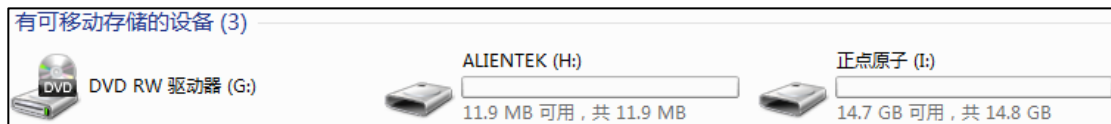


图 60.5.3 电脑找到 USB 读卡器的两个盘符

我们打开设备管理器, 在通用串行总线控制器里面可以发现多出了一个 USB Mass Storage Device, 同时看到磁盘驱动器里面多了 2 个磁盘, 如图 60.5.4 所示:

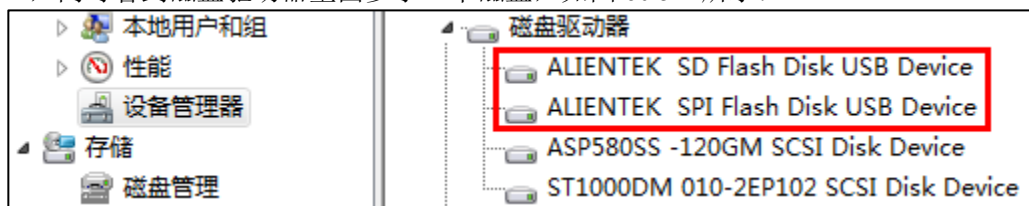


图 60.5.4 通过设备管理器查看磁盘驱动器

此时, 我们就可以通过电脑读写 SD 卡或者 SPI FLASH 里面的内容了。在执行读写操作的时候, 就可以看到 DS1 亮, 并且会在液晶上显示当前的读写状态。

注意, 因为 SPI FLASH 有写次数限制, 最好不要频繁的往里面写数据, 否则很容易将 SPI FLASH 写坏!!

第六十一章 USB 虚拟串口实验

本章,我们将向大家介绍如何利用 USB FS 在正点原子战舰 STM32F1 开发板实现一个 USB 虚拟串口,通过 USB 与电脑数据数据交互。本章分为如下几个部分:

- 61.1 USB 虚拟串口简介
- 61.2 硬件设计
- 61.3 软件设计
- 61.4 下载验证

61.1 USB 虚拟串口简介

USB 虚拟串口,简称 VCP,是 Virtual COM Port 的简写,它是利用 USB 的 CDC 类来实现的一种通信接口。CDC(Communication Device Class)类是 USB2.0 标准下的一个设备类,定义了通信相关设备的抽象集合。ST 官方文档《USB CDC 类入门培训》提供了关于 USB CDC 的详细知识,同样的,我们把资料放到“光盘资料(A 盘)→8, STM32 参考资料→2, STM32 USB 学习资料”中了。

我们可以利用 STM32 自带的 USB 功能,来实现一个 USB 虚拟串口,从而通过 USB,实现电脑与 STM32 的数据互传。上位机无需编写专门的 USB 程序,只需要一个串口调试助手即可调试,非常实用。

61.2 硬件设计

1. 例程功能

本章实验功能简介:本实验利用 STM32 自带的 USB 功能,连接电脑 USB,虚拟出一个 USB 串口,实现电脑和开发板的数据通信。本例程功能完全同实验 串口通信实验,只不过串口变成了 STM32 的 USB 虚拟串口。当 USB 连接电脑(USB 线插入 USB_SLAVE 接口),开发板将通过 USB 和电脑建立连接,并虚拟出一个串口(注意:虚拟串口需要先安装驱动,虚拟串口驱动在: A 盘→6, 软件资料→1, 软件→STM32 USB 虚拟串口驱动→en.stsw-stm32102.zip),USB 和电脑连接成功后,DS1 常亮。

在找到虚拟串口后,即可打开串口调试助手,实现同串口实验类似的功能,即:STM32 通过 USB 虚拟串口和上位机对话,STM32 在收到上位机发过来的字符串(以回车换行结束)后,原样返回给上位机。下载后,DS0 闪烁,提示程序在运行,同时每隔一定时间,通过 USB 虚拟串口输出一段信息到电脑。

2. 硬件资源

- 1) LED 灯
 - LED0 – PB5
 - LED1 – PE5
- 2) KEY0 按键
- 3) TFTLCD 模块
- 4) USB SLAVE 接口
- 6) 串口 1

这几个外设原理图我们在之前的章节已经介绍过了,这里就不重复介绍了,不清楚的话可以查看本文之前章节的描述或对光盘资料提供的开发板原理图。这里再次提醒大家, P10 的连接,要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。

61.3 程序设计

同上一章一样，我们直接移植官方的 USB VCP 例程，官方例程路径：8，STM32 参考资料 → 1，STM32F1xx 固件库 → STM32Cube_FW_F1_V1.8.0 → Projects → STM3210E_EVAL → Applications → USB_Device → CDC_Standalone，该例程采用 USB CDC 类来实现，利用 STM32 的 USB 接口，实现一个 USB 转串口的功能。我们先打开该例程的 MDK 工程(MDK-ARM 文件夹下)，查看一下其工程结构：

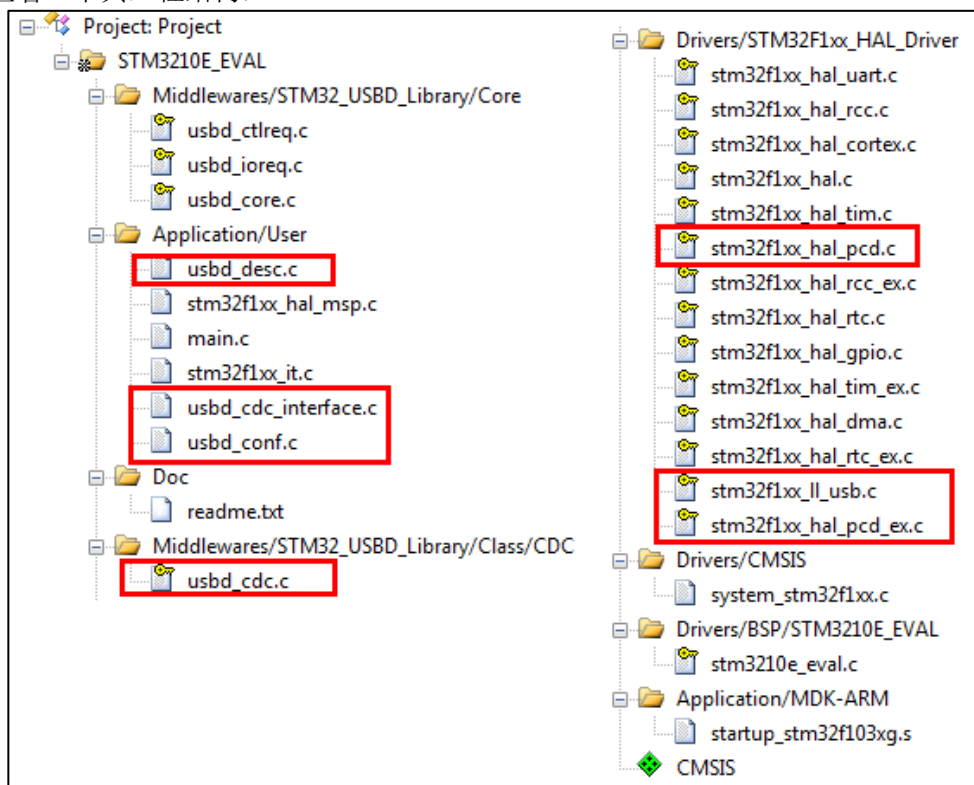


图 61.3.1 ST 官方例程的结构

从工程结构不难找出我们需要的 USB 功能代码，并且这些文件是加了只读属性的，我们移植后需要把只读属性去掉才能进行我们需要的修改。我们复制之前的“内存管理实验”工程，重命名为“USB 虚拟串口实验”，因为我们并不是所有例程都使用 USB 库驱动，故我们把 USB 作为一个第三方组件放到我们的“Middlewares”文件夹下，我们在该文件夹下新建一个 USB 文件目录，把 USB 相关的全部放到 USB 文件夹下使这部分驱动完全独立，这样可以方便我们以后事移植到其它项目中。

首先是 usbd_core.c、usbd_ioreq.c、usbd_ctlreq.c 这三个文件，我们查看它们所在的路径发现它们都位于“STM32_USB_Device_Library”文件夹下，所以我们可以直接把该文件夹复制到我们的“USB 虚拟串口/Middlewares/USB”文件夹下，后面再考虑精简工程。复制后 USB 文件夹下就有“STM32_USB_Device_Library/Class”和“STM32_USB_Device_Library/Core”两个目录了。

接着同样的方法，找到 usbd_cdc.c 文件，文件位于“STM32_USB_Device_Library/Class”文件夹下，上一步我们已经把整个文件夹复制到我们的 USB 目录下了，所以这步不需要再操作。

接下来的 USB 应用程序 usbd_cdc_interface.c、usbd_desc.c、usbd_conf.c 三个文件，源文件和头文件分别位于 USB_Device\CDC_Standalone\Src 和\USB_Device\CDC_Standalone\Inc 下，我们在 USB 文件夹下新建一个“USB_APP”文件夹，把它们连同头文件都放到该文件夹下。复制后的 USP_APP 目录结构如图 61.3.2 所示。

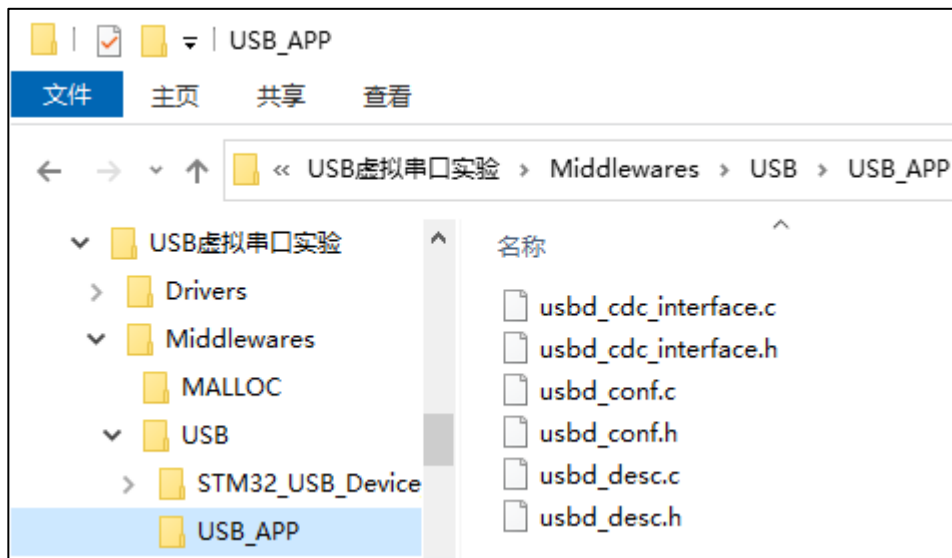


图 61.3.2 USB APP 文件夹下的文件

我们需要添加的文件已经准备好了，接下来需要把这些文件添加到我们移植的 MDK 工程中。为了使 USB 的驱动更加独立，我们按原来的定义，在 MDK 中新建 Middlewares/USB_CORE、Middlewares/USB_CLASS、Middlewares/USB_APP 三个分组，把上面的文件的只读属性去掉后添加到我们的工程中，然后把相关的 HAL 库的驱动加到 Drivers/STM32F1xx_HAL_Driver 目录下，添加后的工程目录如图 61.3.3 所示。

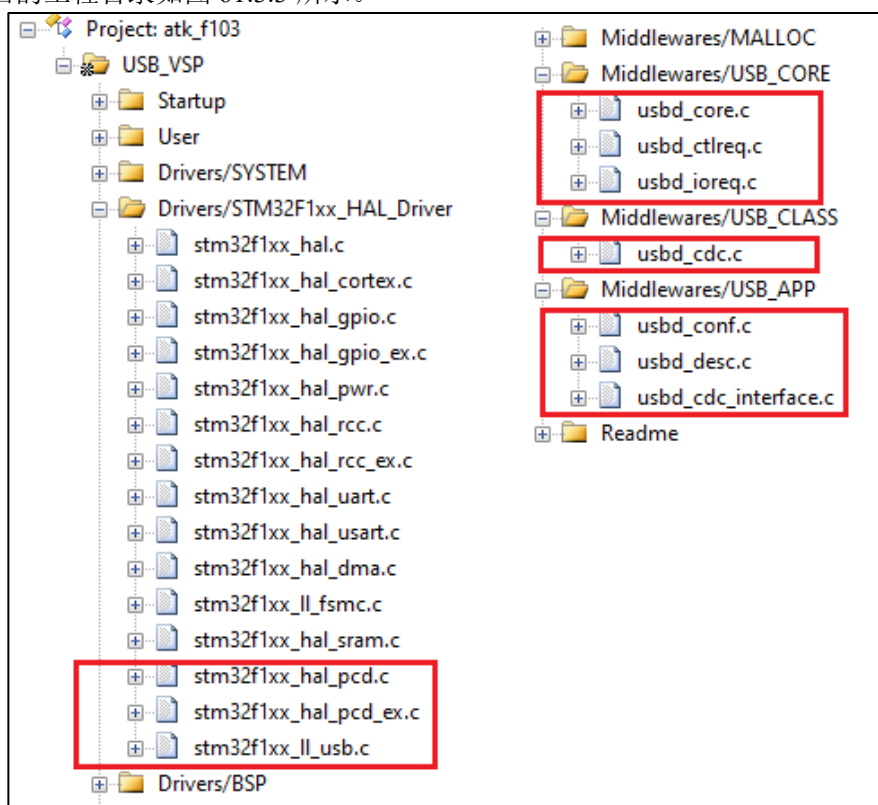


图 61.3.3 在 MDK 中添加需要的代码

为了保持 USB 驱动部分更少的改动，我们添加原有 USB 库的头文件的引用路径，结果如图 61.3.4 所示。

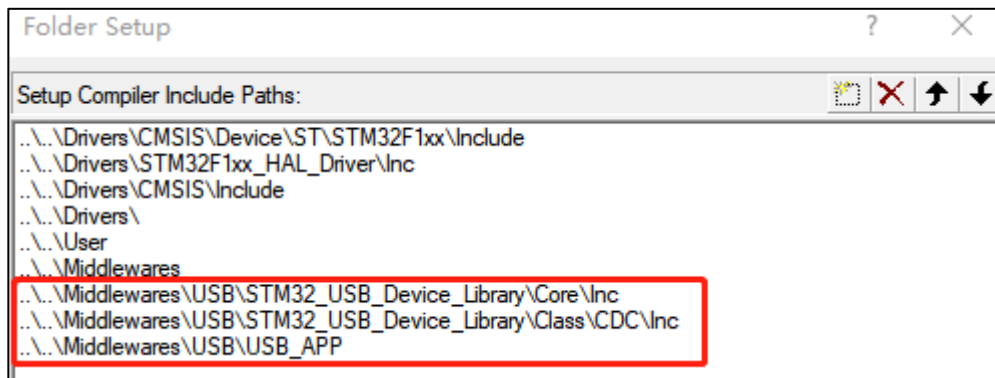


图 61.3.4 在 MDK 中添加 USB 引用的头文件的路径

这时我们直接编译会报错，因为我们没有引用 ST 开发板的 BSP 文件，这时我们还需要修改相关源码以匹配我们的底层的驱动，这部分与我们开发板硬件相关，我们在程序设计的部分再讲解。

61.3.1 程序流程图

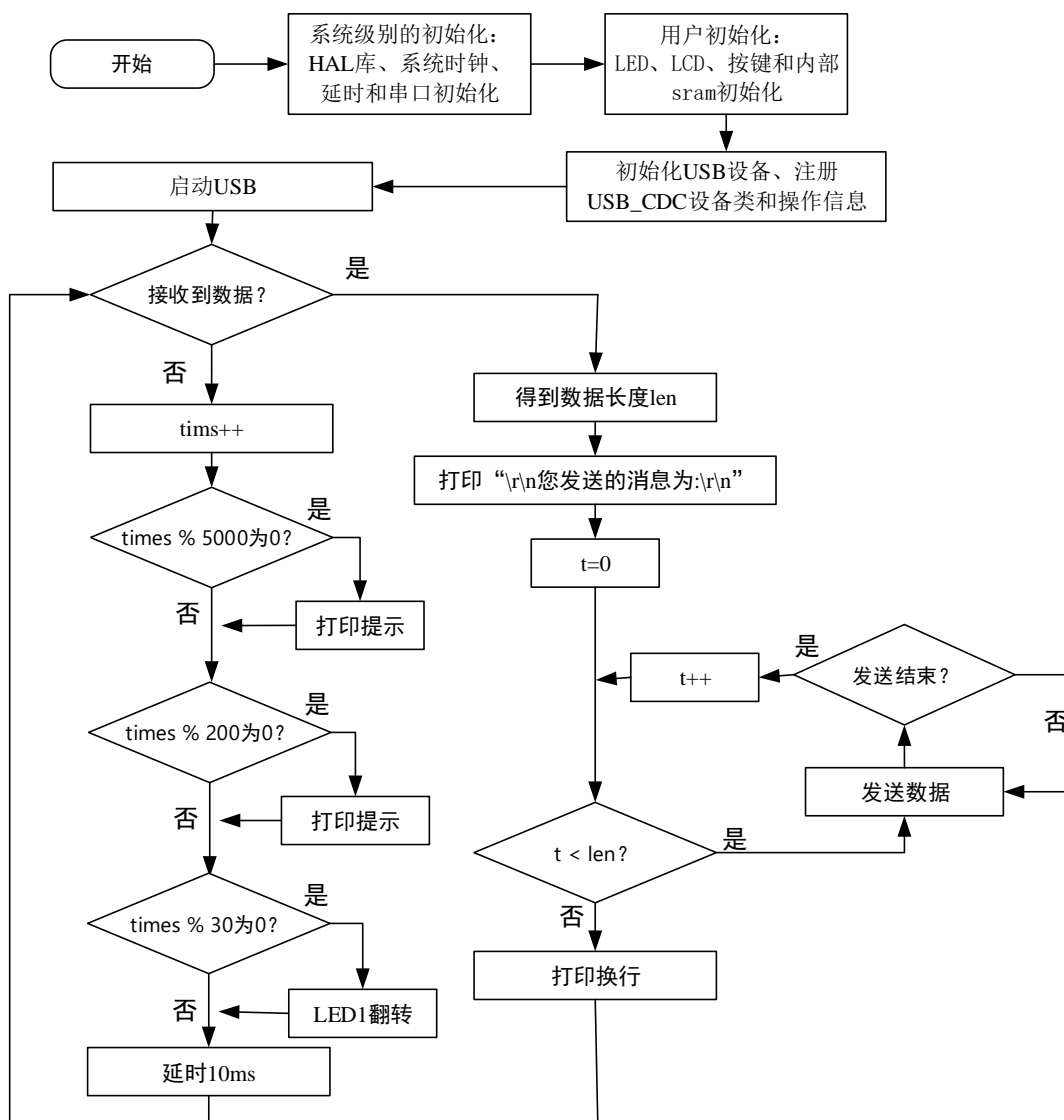


图 61.3.1.1 USB 虚拟串口程序流程图

我们按流程图编写的初始化顺序，在 STM32 注册 USB 内核，最后通过 USB 的中断和回调

函数得到 USB 的操作状态和操作结果，主程序通过查询设定的标记变量的状态值后，在 LCD 上显示对应的 USB 操作状态。最后通过 USB 和电脑实现数据交互。

61.3.2 usbd_cdc_interface 驱动

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，USB 虚拟串口的驱动主要包括两个文件：usbd_cdc_interface.c 和 usbd_cdc_interface.h。

usbd_cdc_interface.c/h 需要适配我们的硬件信息。本例中，我们需要把 USB 操作与我们的串口操作对应起来。为了实现我们流程图设想的功能，我们将对原例程中的这部分代码进行修改，在此之前，我们先大致了解一下需要用到的 USB 设备类的一些参数和操作接口。下面介绍对应接口时我们给出的是修改后的源码，详细的修改点大家可以参考对照光盘的源码即可。

1. USBD_CDC_fops 结构体

USBD_CDC_fops 结构体定义了 CDC 通讯设备的操作，它是 USBD_CDC 类下封装的一个结构体，用于 USBD_CDC 类与应用层的分离，我们通过对这个结构体进行实例化即可完成对 CDC 类的应用层的功能定义，它会在注册后被 USB 内核调用，我们需要为这些 USB 操作实现跟我们硬件相关的底层代码，它们会在 USB 枚举过程中被调用。

```
/* 虚拟串口配置函数(供 USB 内核调用) */
USBD_CDC_ItfTypeDef USBD_CDC_fops =
{
    CDC_Itf_Init,
    CDC_Itf_DeInit,
    CDC_Itf_Control,
    CDC_Itf_Receive
};
```

● 操作描述:

CDC_Itf_Init 用于初始化 VCP，在初始化的时候由 USB 内核调用，这里我们调用函数：USBD_CDC_SetRxBuffer，设置 USB 接收数据缓冲区。USB 虚拟串口收到的数据，会先缓存在这个 buf 里面，ST 原来的例程同时设置了写缓冲，我们为了方便单独编写了写发送函数。

CDC_Itf_DeInit 用于复位 VCP，我们用不到，所以直接返回 USBD_OK 即可。

CDC_Itf_Control 用于控制 VCP 的相关参数，根据 cmd 的不同，执行不同的操作，这里主要用到 CDC_SET_LINE_CODING 命令，该命令用于设置 VCP 的相关参数，比如波特率、数据类型（位数）、校验类型（奇偶校验）等，保存在 linecoding 结构体里面，在需要的时候，应用程序可以读取 LineCoding 结构体里面的参数，以获得当前 VCP 的相关信息。

CDC_Itf_Receive 和 VCP_DataRx，这两个函数一起，用于 VCP 数据接收，当 STM32 的 USB 接收到电脑端串口发送过来的数据时，由 USB 内核程序调用 CDC_Itf_Receive，然后在该函数里面再调用 VCP_DataRx 函数，实现 VCP 的数据接收，只需要在该函数里面，将接收到的数据，保存起来即可，接收的原理同串口通信实验完全一样。

2. CDC_Itf_Init 函数

CDC_Itf_Init 用于初始化 VCP，在初始化的时候由 USB 内核调用，这里我们可以编写实际的串口应用，使虚拟串口与物理串口对应上，但为了简化测试，我们没有加上这个操作，这里只为 CDC 设备通讯分配接收缓存：

```
static int8_t CDC_Itf_Init(void)
```

● 函数描述:

初始存储 CDC 通讯设备，这里主要包括跟开发板底层驱动相关的及 CDC 设备类相关的收发缓存的分配，所以在这个函数里面会调用 USBD_CDC_SetRxBuffer，设置 USB 接收数据缓冲区。USB 虚拟串口收到的数据，会先缓存在这个 buf 里面。

```
/**
 * @brief      初始化 CDC
 * @param      无
 * @retval     USB 状态
 * @arg        USBD_OK(0) , 正常;
 * @arg        USBD_BUSY(1) , 忙;
 * @arg        USBD_FAIL(2) , 失败;
 */
```



```
static int8_t CDC_Itf_Init(void)
{
    USBD_CDC_SetRxBuffer(&USBD_Device, g_usb_rx_buffer);
    return USBD_OK;
}
```

● 函数返回值:

返回硬件初始化结果: USBD_OK(0): 成功, 其它: 错误或忙。

3. CDC_Itf_Control 函数

CDC_Itf_Control 用于 CDC 设备的控制, 主机可以通过这些接口配置 CDC 设备:

```
static int8_t CDC_Itf_Control(uint8_t cmd, uint8_t *pbuf, uint16_t length)
```

● 函数描述:

控制接口主要用来做设备管理和电话管理(可选), 设备管理涉及到请求(request)和通知(notification), 端点 0 一般用做请求, 一般用来控制和配置设备的运行状态, 而非 0 端点(0x82)一般用作异步事件通知, 设备端通过此端点向主机端发送设备内部的一些事件, 比如串口状态变化事件, 电话状态改变等等

USB 请求在 USB 器件库文档中分为三类: 标准请求、特定类请求、特定厂商请求。其中标准请求由默认的控制端点 0 接收处理, 另外两类请求由回调函数传递到特定类代码进行处理, 这里的 CDC_Itf_Control()函数即为回调函数最终实现者, 其处理 CDC 类的特定请求, 这些请求可以在 PSTN120 协议文档以及 CDC120 协议文档中找到。

● 函数形参:

形参 1 cmd 为控制命令, PSTN 定义了三种模型:DLM(Direct Line Mode),ACM(Abstract Control Model)和 TCM(Telephone Control Model),STM32 的 CDC 设备采用的是抽象控制模型(ACM: Abstract Control Model), 对此控制指令的格式进行了定义, 这些请求可以在 PSTN120 协议文档以及 CDC120 协议文档中找到, 但能否实现还需要主机支持。ST 文档《USB CDC 类入门培训》也有对这些类进行描述, 这里我们就不列举了, 大家看代码实现即可。

形参 2 pbuf 用于需要传递数据的控制命令时使用。

形参 3 length 表示控制命令的传递过来的数据按字节计的数量。

```
/**
 * @brief      控制 CDC 的设置
 * @param      cmd      : 控制命令
 * @param      buf      : 命令数据缓冲区/参数保存缓冲区
 * @param      length   : 数据长度
 * @retval     USB 状态
 * @arg        USBD_OK(0) , 正常;
 * @arg        USBD_BUSY(1) , 忙;
 * @arg        USBD_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_Control(uint8_t cmd, uint8_t *pbuf, uint16_t length)
{
    switch (cmd)
    {
        case CDC_SEND_ENCAPSULATED_COMMAND:
            break;
        case CDC_GET_ENCAPSULATED_RESPONSE:
            break;
        case CDC_SET_COMM_FEATURE:
            break;
        case CDC_GET_COMM_FEATURE:
            break;
        case CDC_CLEAR_COMM_FEATURE:
            break;
        case CDC_SET_LINE_CODING:
            LineCoding.bitrate = (uint32_t) (pbuf[0] | (pbuf[1] << 8) |
                                             (pbuf[2] << 16) | (pbuf[3] << 24));
            LineCoding.format = pbuf[4];
            LineCoding.paritytype = pbuf[5];
            LineCoding.datatype = pbuf[6];
            /* 打印配置参数 */
    }
```

```
printf("linecoding.format:%d\r\n", LineCoding.format);
printf("linecoding.paritytype:%d\r\n", LineCoding.paritytype);
printf("linecoding.datatype:%d\r\n", LineCoding.datatype);
printf("linecoding.bitrate:%d\r\n", LineCoding.bitrate);
break;
case CDC_GET_LINE_CODING:
    pbuf[0] = (uint8_t) (LineCoding.bitrate);
    pbuf[1] = (uint8_t) (LineCoding.bitrate >> 8);
    pbuf[2] = (uint8_t) (LineCoding.bitrate >> 16);
    pbuf[3] = (uint8_t) (LineCoding.bitrate >> 24);
    pbuf[4] = LineCoding.format;
    pbuf[5] = LineCoding.paritytype;
    pbuf[6] = LineCoding.datatype;
    break;
case CDC_SET_CONTROL_LINE_STATE:
    break;
case CDC_SEND_BREAK:
    break;
default:
    break;
}
return USB_D_OK;
}
```

- **函数返回值:**

返回操作结果: 0(USB_D_OK): 成功, 其它: 错误或忙等状态。

4. CDC_Itf_Receive 函数

CDC_Itf_Receive 用于 CDC 数据接收处理, 我们虚拟串口的数据接收通过这个接口实现。

```
static int8_t CDC_Itf_Receive(uint8_t *buf, uint32_t *len)
```

- **函数描述:**

初始化指定编号的磁盘, 磁盘所指定的存储区。

- **函数形参:**

形参 1 buf 是接收数据缓冲, 我们收到的数据存放在这个缓冲中。

形参 2 为按字节计从虚拟串口接收到的数据。

```
/**
 * @brief      CDC 数据接收函数
 * @param      buf      : 接收数据缓冲区
 * @param      len      : 接收到的数据长度
 * @retval     USB 状态
 * @arg        USB_D_OK(0) , 正常;
 * @arg        USB_BUSY(1) , 忙;
 * @arg        USB_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_Receive(uint8_t *buf, uint32_t *len)
{
    USB_CDC_ReceivePacket(&USB_Device);
    cdc_vcp_data_rx(buf, *len);
    return USB_D_OK;
}
```

上面的 cdc_vcp_data_rx(buf, *len) 函数为我们编写的虚拟串口缓冲数据处理接收函数, 写法与我们的串口实验相同, 以全局 g_usb_uart_rx_sta 为接收标记和数据计数器, 以“\r\n”为结束符, 大家参考光盘中的代码即可。

- **函数返回值:**

返回操作结果: 0(USB_D_OK): 成功, 其它: 错误或忙等状态。

5. cdc_vcp_data_tx 函数

cdc_vcp_data_tx 实现 USB 对物理设备的读操作, 其声明如下:

```
void cdc_vcp_data_tx(uint8_t *data, uint32_t Len)
```

- **函数描述:**

用于 STM32 向虚拟串口发送数据。

- **函数形参:**

形参 1 data 为接收的数据指针。

形参 2 Len 同为按字节计的数据长度。

代码实现如下：

```
/**
 * @brief      通过 USB 发送数据
 * @param      buf      : 要发送的数据缓冲区
 * @param      len      : 数据长度
 * @retval     无
 */
void cdc_vcp_data_tx(uint8_t *data, uint32_t Len)
{
    USBDCDC_SetTxBuffer(&USBD_Device, data, Len);
    USBDCDC_TransmitPacket(&USBD_Device);
    delay_ms(CDC_POLLING_INTERVAL);
}
```

为了方便实现打参数格式化和测试打印，我们编写了 `usb_printf` 这个接口用于实际的写测试，这是一个变参函数，与 `printf` 的功能类似，本质上还是调用 `cdc_vcp_data_tx` 接口，主要还是调用了 USBD 下的 CDC 类的发送操作接口，大家参考我们的源代码即可。

● 函数返回值：

无。

`usbd_cdc` 的代码就讲到这里，`usbd_conf.c` 的配置和 USB 读卡器的章节一致，我们就不重复了。

2. main.c 代码

在 `main.c` 就比较简单了，按照我们的流程图的思路编写即可，我们初始化探按键，LCD 和 LED、SRAM 等外设辅助程序显示，同样地在 `while` 循环之前调用我们之前编写的 `usbd_port_config` 接口以保证每次复位后 USB 都能重连成功。大家也可以尝试其它的方法。

最后，我们编写的 `main` 函数如下：

```
USBD_HandleTypeDef USBD_Device; /* USB Device 处理结构体 */
extern volatile uint8_t g_device_state; /* USB 连接 情况 */

int main(void)
{
    uint16_t len;
    uint16_t times = 0;
    uint8_t usbstatus = 0;
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    sram_init(); /* SRAM 初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USB Virtual USART TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    /* 提示 USB 开始连接 */
    lcd_show_string(30, 110, 200, 16, 16, "USB Connecting...", RED);
    usbd_port_config(0); /* USB 先断开 */
    delay_ms(500);
    usbd_port_config(1); /* USB 再次连接 */
    delay_ms(500);
    USBD_Init(&USBD_Device, &VCP_Desc, 0);
    USBD_RegisterClass(&USBD_Device, USBD_CDC_CLASS);
    USBD_CDC_RegisterInterface(&USBD_Device, &USBD_CDC_fops);
    USBD_Start(&USBD_Device);
    while (1)
    {
```

```

if (usbstatus != g_device_state)          /* USB 连接状态发生了改变 */
{
    usbstatus = g_device_state;            /* 记录新的状态 */
    if (usbstatus == 1)
    { /* 提示 USB 连接成功 */
        lcd_show_string(30, 110, 200, 16, 16, "USB Connected ", RED);
        LED1(0);                          /* 绿灯亮 */
    }
    else
    { /* 提示 USB 断开 */
        lcd_show_string(30, 110, 200, 16, 16, "USB disConnected ", RED);
        LED1(1);                          /* 绿灯灭 */
    }
}
if (g_usb_usart_rx_sta & 0x8000)
{
    len = g_usb_usart_rx_sta & 0x3FFF;     /* 得到此次接收到的数据长度 */
    usb_printf("\r\n 您发送的消息长度为:%d\r\n\r\n", len);
    cdc_vcp_data_tx(g_usb_usart_rx_buffer, len);
    usb_printf("\r\n\r\n");               /* 插入换行 */
    g_usb_usart_rx_sta = 0;
}
else
{
    times++;
    if (times % 5000 == 0)
    {
        usb_printf("\r\nSTM32 开发板 USB 虚拟串口实验\r\n");
        usb_printf("正点原子@ALIENTEK\r\n\r\n");
    }
    if (times % 200 == 0)usb_printf("请输入数据,以回车键结束\r\n");
    if (times % 30 == 0)
    {
        LED0_TOGGLE(); /* 闪烁 LED,提示系统正在运行 */
    }
    delay_ms(10);
}
}
}

```

61.4 下载验证

本例程的测试，需要在电脑上先安装 ST 提供的 USB 虚拟串口驱动软件，该软件可以在 STM32 论坛搜索“STM32 Virtual COM Port Driver”获取最新的虚拟串口版本。我们光盘资料提供了 V1.5 版本的压缩包，位于 **A 盘→6，软件资料→1，软件→STM32 USB 虚拟串口驱动→en.stsw-stm32102.zip**，我们解压这个文件可以看到有多个版本的软件可选，如图 61.4.1 所示：








| (A盘) > 6, 软件资料 > 1, 软件 > STM32 USB虚拟串口驱动 | | |
|--|--------------|-----------|
| 名称 | 类型 | 大小 |
|  en.stsw-stm32102.zip | 360压缩 ZIP 文件 | 24,755 KB |
|  readme.txt | 文本文档 | 3 KB |
|  VCP_V1.5.0_Setup_W7_x64_64bits.exe | 应用程序 | 6,745 KB |
|  VCP_V1.5.0_Setup_W7_x86_32bits.exe | 应用程序 | 6,745 KB |
|  VCP_V1.5.0_Setup_W8_x64_64bits.exe | 应用程序 | 6,745 KB |
|  VCP_V1.5.0_Setup_W8_x86_32bits.exe | 应用程序 | 6,745 KB |
|  version.txt | 文本文档 | 3 KB |

图 61.4.1 选择适合自己电脑的软件版本安装

由于我们电脑系统是 Win10 64 位的系统，所以我们安装 win7 以后的版本，这里我们选择 VCP_V1.5.0_Setup_W8_x64_64bits.exe 这个版本，安装过程比较简单，我们填写一些简单的信息，一直点下一步安装到默认路径即可，安装完成后界面如图 61.4.2 所示：

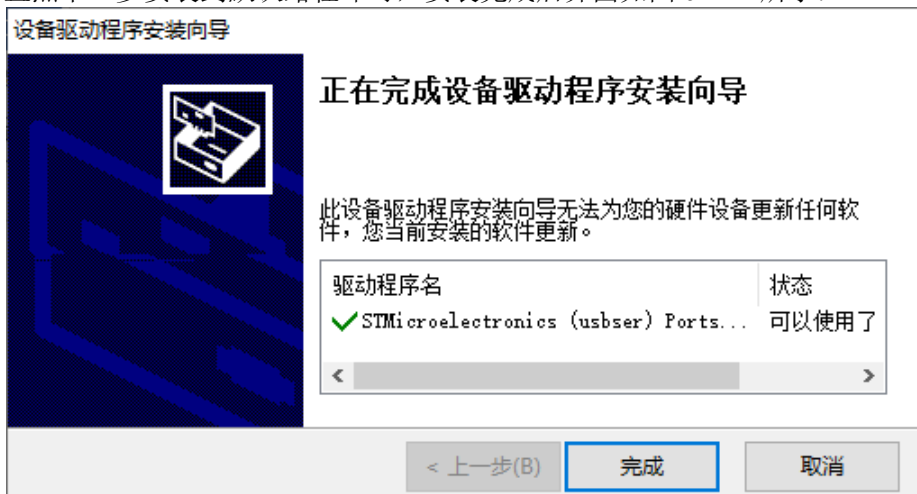


图 61.4.2 STM32 VCOM 驱动安装完成

安装完驱动后我们就可以开始实验了。在代码编译成功之后，我们下载代码到 STM32 开发板上，然后将 USB 数据线，插入 USB_SLAVE 口（此时要检查 P9 排针是处的 PA11、PA12 是否与 USB_D+、USB_D- 正确连接），通过 USB_Slave 连接电脑和开发板，此时电脑会提示找到新硬件，并自动安装驱动。不过，如果自动安装不成功（有惊叹号），如图 63.4.3 所示：

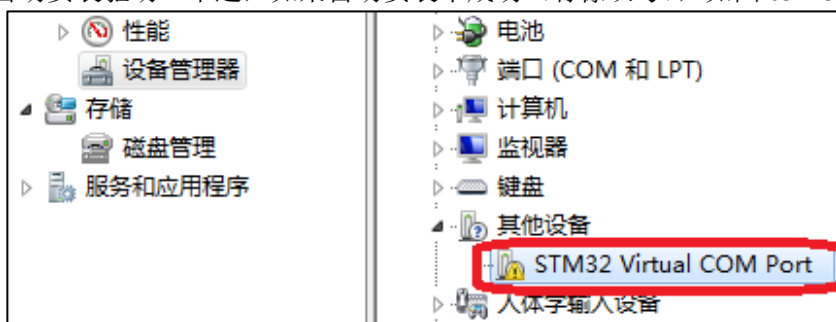


图 61.4.3 自动安装驱动失败

此时，我们可手动选择驱动(以 WIN7 为例)，进行安装，在如图 61.4.3 所示的条目上面，右键→更新驱动程序软件→浏览计算机以查找驱动程序软件→浏览，选择 STM32 虚拟串口的驱动的路径为：C:\ProgramFiles(x86)\STMicroelectronics\Software\Virtual comport driver，上述的文件夹下同样有对应对应的 WIN7 和 WIN8 版本，如果是 WIN7 以后的 windows 版本我们选择 WIN8 就可以了。然后点击下一步，即可完成安装。安装完成后，可以看到设备管理器里面多出了一个 STM32 的虚拟串口，这时我们就可以开始正常测试了，效果如图 63.4.4。

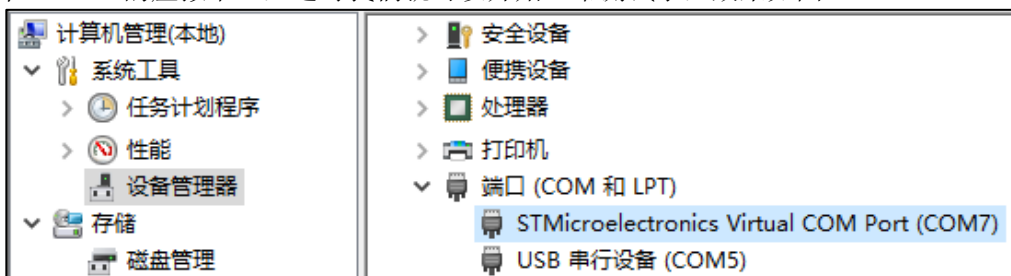


图 63.4.4 发现 STM32USB 虚拟串口

如图 63.4.4，STM32 通过 USB 虚拟的串口，被电脑识别了，端口号为：COM7（不同电脑有差异），字符串名字为：STMicroelectronics Virtual COM Port（固定）。此时，开发板的 DS1 常亮，同时，开发板的 LCD 显示 USBConnected，如图 63.4.5 所示：



图 63.4.5 USB 虚拟串口连接成功

正确安装好虚拟串口驱动后，我们打开 XCOM 进行测试，选择电脑识别到的虚拟串口号，我们这里是 COM7（需根据自己的电脑识别到的串口号选择），并打开串口（注意：波特率可以随意设置），我们例程的代码以接收到的数据中的回车换行为接收结束符，故测试时串口助手必须勾选：发送新行，STM32 才会把收到的数据回发给 USB 口，测试结果如图 63.4.6 所示：

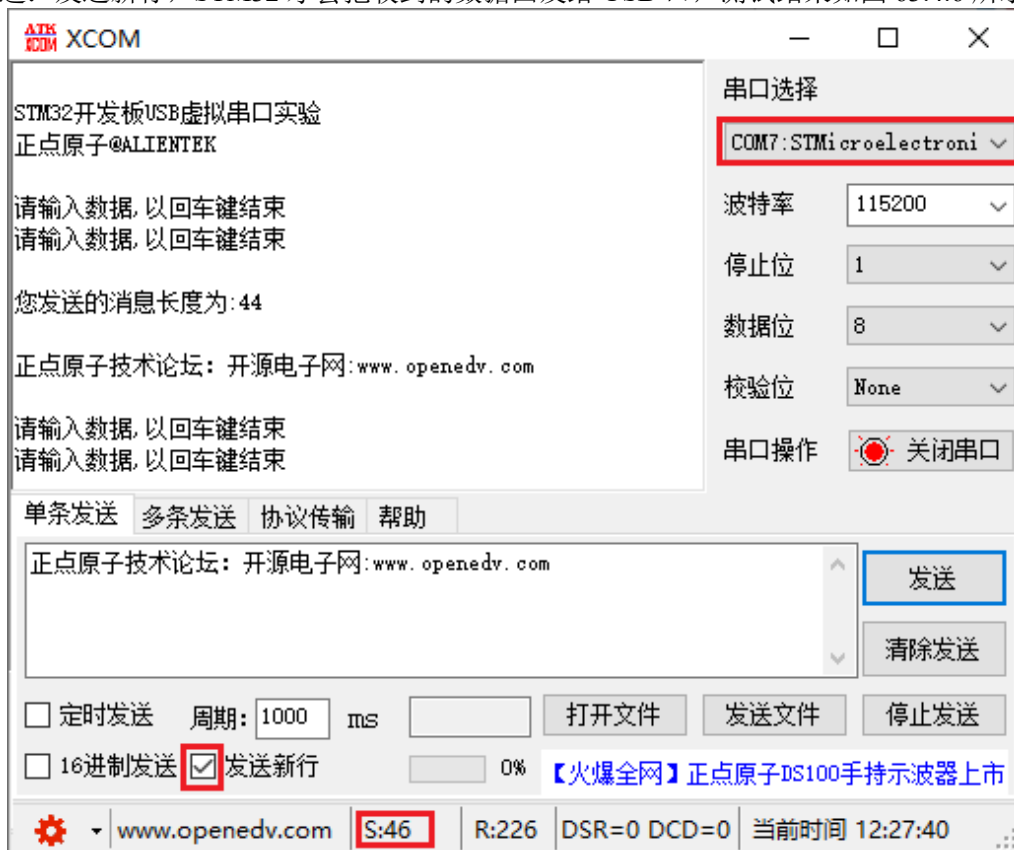


图 63.4.6 STM32 虚拟串口通信测试

可以看到，我们的串口调试助手，收到了来自 STM32 开发板的数据，同时，按发送按钮（串口助手必须勾选：发送新行），也可以收到电脑发送给 STM32 的数据（原样返回），说明我们的实验是成功的。实验现象同串口通讯实验完全一样。

至此，USB 虚拟串口实验就完成了，通过本实验，我们就可以利用 STM32 的 USB，直接和电脑进行数据互传了，具有广泛的应用前景。

第六十二章 网络通讯实验

战舰开发板板载以太网芯片的相关知识及配套例程,请参考战舰开发板的《网络开发指南》。

第六十三章 UCOSII 实验 1-任务调度

前面我们所有的例程都是跑裸机程序，简称裸跑，从本章开始，我们将分 3 个章节向大家介绍 UCOSII（实时多任务操作系统内核）的使用。本章，我们将向大家介绍 UCOSII 最基本也是最重要的应用：任务调度。本章分为如下几个小节：

- 63.1 嵌入式实时操作系统介绍
- 63.2 硬件设计
- 63.3 程序设计
- 63.4 下载验证

63.1 嵌入式实时操作系统介绍

63.1.1 裸机系统和多任务系统的区别

在嵌入式设备的开发过程中，我们使用的是两种程序，一是裸机程序，前面所有的实验章节使用的都是使用裸机程序；二是多任务程序，即接下来的三章都是多任务程序。很多人会有疑问用得好好的裸机程序，为什么要用多任务程序呢？

裸机程序最大的特点，就是主函数中会有一个大循环，大循环中就会有多个小任务的实现。任务间是按照顺序进行执行的，换句话说它们执行等级是一样的，下一个任务想要执行必须等上一个任务执行完成才能进行。这个运行着的大循环我们称之为后台程序。中断是可以打断系统当前的后台任务优先被执行，等待执行完后，再回到原来后台被打断处继续执行后台程序，中断处理程序称之前台程序。这种使用前后台裸机程序的叫做前后台系统，如图 63.1.1.1 所示：

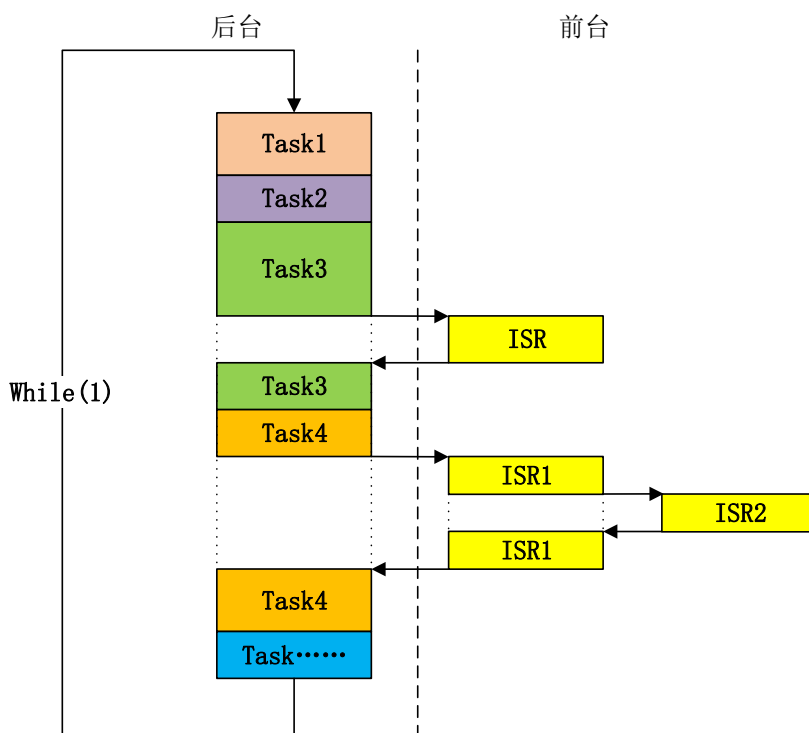


图 63.1.1.1 前后台系统

这样的前后台系统在实时性处理方便存在缺陷，例如 Task1 是重要任务，需要能够得到及时的响应，但是在执行 Task3 的时候，产生中断。现在的情况是执行 Task1 条件满足，理想的处理方式就是 Task1 需要立刻被执行，但是前后台程序中做不到，因为任务是被顺序执行的，即使 Task1 十万火急，也必须等待 Task3 处理完毕才能被执行。

前面的情况对于要求实时性比较强的产品来说，是不允许的。所以出现了多任务程序，这种使用多任务程序的系统，叫做嵌入式实时操作系统。它把任务分为不同的优先级，当运行条件被满足时，高优先级任务可以打断低优先级任务优先运行，从而极大地提高了系统的实时性。嵌入式实时操作系统执行任务示意图如图 63.1.1.2 所示：

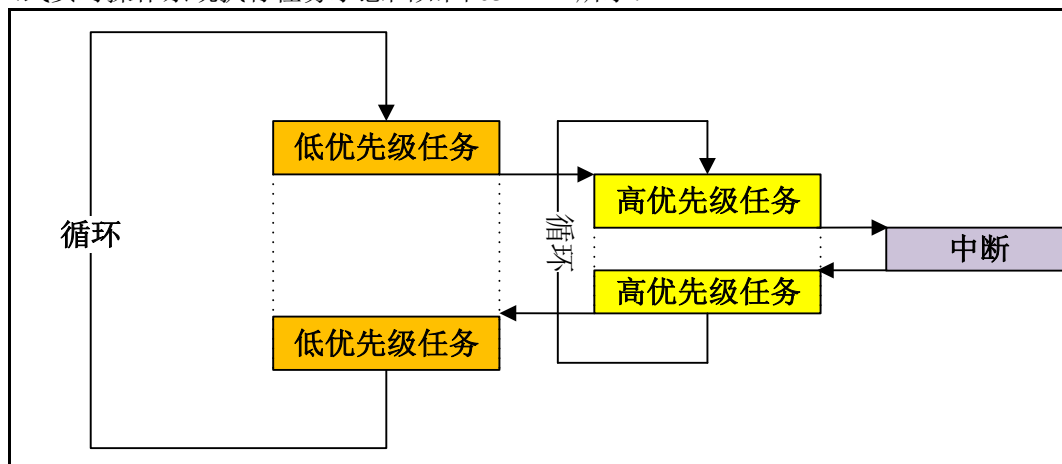


图 63.1.1.2 嵌入式实时操作系统执行任务示意图

嵌入式实时操作系统相比前后台系统明显体现在实时性方面，同时它在多任务管理、任务间通信、内存管理、定时器管理、设备管理等方面也提供了一套完整的机制，极大程度上便利了嵌入式应用程序的开发、管理和维护。

63.1.2 UCOSII 介绍

现在市面上有许多实时操作系统，国外的实时操作系统就有 FreeRTOS，UCOS 和 RTX，国内的实时操作系统就有 RT_Thread、LiteOS 等。其中 FreeRTOS 使用率世界最高，UCOS 发展历史最悠久。在这里我们主要是对 UCOSII 进行学习。

UCOSII，全称是 Micro Control Operation System Two，是由 Micrium 公司提供，是一个可移植、可固化、可裁剪的、占先式多任务实时内核，它适用于多种微处理器，微控制器和数字处理芯片。早在 1992 年就由美国嵌入式专家 Jean J.Labrosse 在《嵌入式系统编程》杂志中提出，并公布源码。UCOSII 只是一个实时操作系统内核，它仅仅包含了任务调度，任务管理，时间管理，内存管理和任务间的通信和同步等基本功能。没有提供输入输出管理，文件系统，网络等额外的服务。该实时系统十分适合初次接触嵌入式实时操作系统的朋友。

本章实验中我们使用的是 UCOSII V2.93.01 版本，它的体系结构如图 63.1.2.1 所示。

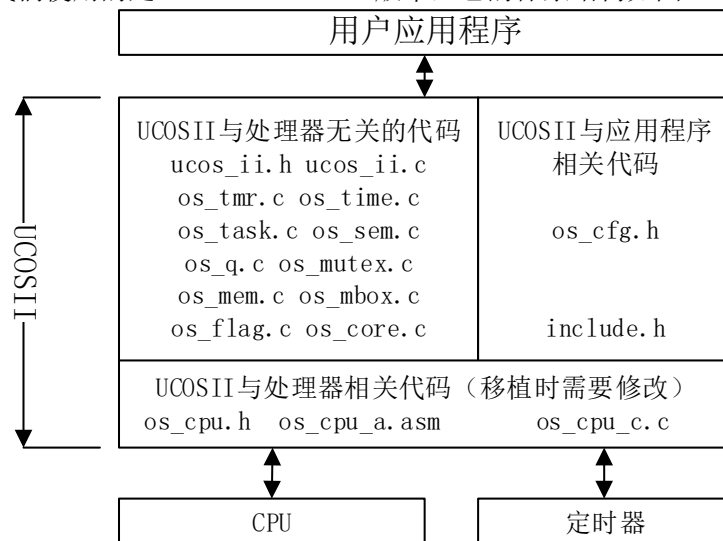


图 63.1.2.1 UCOSII 体系结构图

该版本比早期的 UCOSII(如 V2.52)多了很多功能，比如软件定时器，支持任务数最大达到

255 个等，而且修正了很多已知 BUG。

在 UCOSII 的移植，我们只需要修改：`cpu_cfg.h`、`lib_cfg.h` 和 `os_cfg.h`、`app_cfg.h` 四个文件即可，其中 `cpu_cfg.h` 该文件主要是配置 CPU 相关的一些宏定义，如时间戳、前导置零指令等相关内容；`lib_cfg.h`，该文件主要用于配置 μ C/LIB 组件，如使能内存库中优化的内存相关操作函数；`os_cfg.h` 该文件是系统的配置文件，用于配置系统默认的功能，比如消息队列、信号量、事件标志位等；`app_cfg.h` 该文件用于配置软件定时器的任务优先级等。

图中定时器的作用是为 UCOSII 提供系统时钟节拍，实现任务切换和任务延时等功能。这个时钟节拍由 `OS_TICKS_PER_SEC`(在 `os_cfg.h` 中定义)设置，一般我们设置 UCOSII 的系统时钟节拍为 1ms~100ms，具体根据你所用处理器和使用需要来设置。本章，我们利用 STM32F1 的 SYSTICK 定时器来提供 UCOSII 时钟节拍。

63.1.3 任务定义

在前面也说到有任务，在前面的多任务系统中，我们根据功能的不同，把整个系统分成一个个独立的循环函数，这些函数称为任务。而 UCOSII 就是一个能对这些任务的运行进行管理和调度的多任务操作系统。UCOSII 最大支持的任务数达到了 255 个，但是对于我们来说一般 64 个任务已经足够。

任务类型有两种：一种是系统任务，另一种是用户任务。由系统提供的任务叫系统任务，由用户编写的任务叫用户任务。系统任务是为应用程序提供某种服务或为系统本身服务的。UCOSII 具有 2 个系统任务，即空闲任务和统计任务，占用最低 2 个优先级。空闲任务是 UCOSII 优先级最低的任务，当所有其他任务均没有使用 CPU 时，空闲任务就会占用 CPU。统计任务是 UCOSII 优先级倒数第二低的任务，用于统计 CPU 的使用率和各个任务的堆栈使用情况。

相对于系统任务而言，我们开发者用得多的就是用户任务。用户任务需要注意的是：用户任务对应的函数是一个带有无限循环体的函数，没有返回值；每一个用户任务具有唯一的优先级号。

实时操作系统为了更好的调度任务，给每一个任务都定义了一个任务控制块 TCB (Task Control Block)。这个任务控制块就相当于任务在系统里的身份证，存放着任务的所有信息，比如任务函数指针，任务堆栈指针，任务优先级等。

由于 CPU 只有一个，所以一个时刻只会一个任务占用 CPU 处于运行状态，而其他任务只能处于其他状态。UCOSII 系统中的任务具有 5 种，系统运行起来的时候，每一个任务都处在以下 5 种状态之一的状态下，这 5 种状态分别是睡眠状态、就绪状态、运行状态、等待状态和中断服务状态。

睡眠状态，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态。

就绪状态，系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，任务已经准备好了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行，这时任务的状态叫做就绪状态。

运行状态，该任务获得 CPU 使用权，并正在运行中，此时的任务状态叫做运行状态。

等待状态，正在运行的任务，需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给别的任务而使任务进入等待状态。

中断服务状态，一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态。

UCOSII 任务的 5 个状态转换关系如图 63.1.3.1 所示：

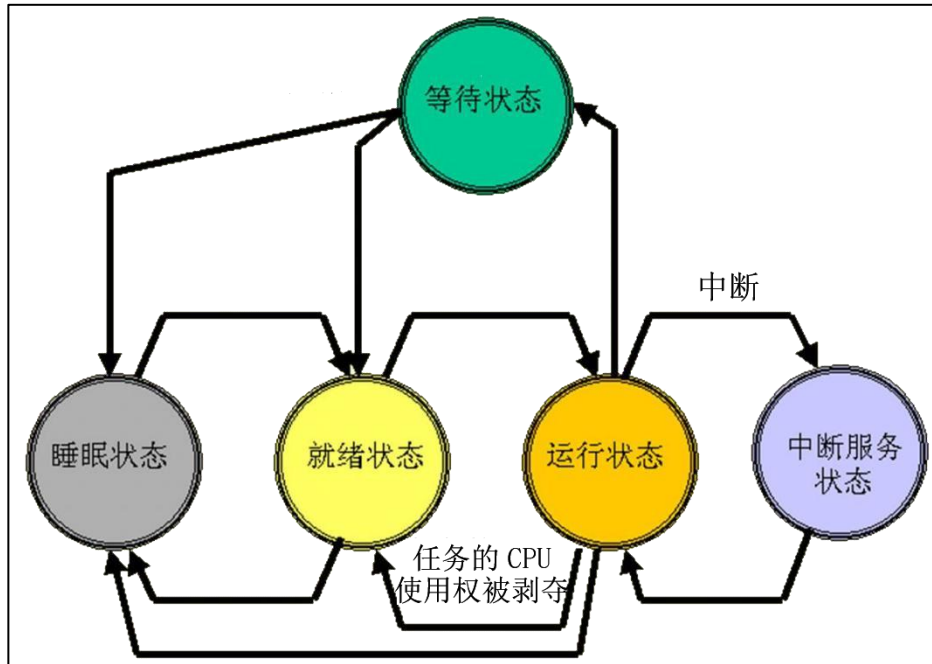


图 63.1.3.1 UCOSII 任务转换关系

63.1.4 任务调度

UCOSII 的任务调度思想是：“近似每时每刻让优先级最高的就绪任务处于运行状态”。在具体做法上，它在系统或者用户任务调用系统函数及执行中断服务程序结束时来调用调度器，以确定应该运行的任务并运行它。

在多任务系统中，令 CPU 中止当前正在运行的任务转而去运行另一个任务的工作叫任务切换，而按照某种规则进行任务切换的工作叫做任务调度。

在 UCOSII 中，任务调度是由任务调度器来完成。任务调度器的主要工作就有两个：①在任务就绪表中查找具有最高优先级别的就绪任务 ②实现任务切换

63.2 硬件设计

1. 例程功能

创建了 3 个任务 `start_task`、`led0_task` 和 `led1_task`。其中 `start_task` 是创建其他 2 个任务（`led0` 和 `led1`）的。当 `start_task` 创建完其他 2 个任务后就会挂起。`led0_task` 和 `led1_task` 这两个任务分别是让 LED0，LED1 闪烁。LED0 每秒钟亮 80ms；LED1 每秒钟亮 300ms。

2. 硬件资源

- 1) LED 灯
 - LED0 - PB5
 - LED1 - PE5

3. 原理图

本章用到的硬件用到 LED 灯：LED0 和 LED1。电路在开发板上已经连接好了，所以在硬件上不需要动任何东西，直接下载代码就可以测试使用。其连接原理图如图 63.2.1 所示：

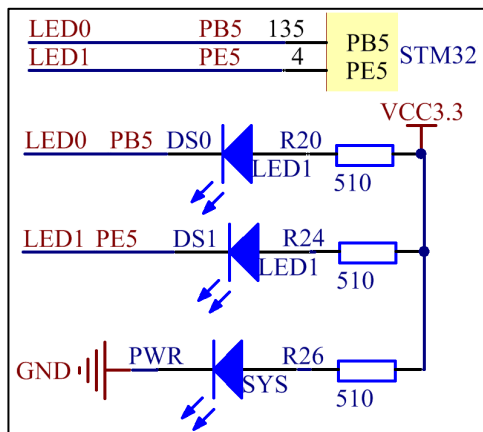


图 63.2.1 LED 与 STM32F103 连接原理图

63.3 程序设计

63.3.1 UCOSII 驱动函数

在这里主要对本实验用到的 UCOSII 驱动函数进行介绍。

1. OSTaskCreateExt 函数

创建任务函数，该函数是 OSTaskCreate 函数的扩展，并提供了一些附加功能。OSTaskCreateExt 函数创建任务更加灵活，不过会增加一些额外的开销。其声明如下：

```
INT8U OSTaskCreateExt (void (*task)(void *p_arg),
                        void *p_arg,
                        OS_STK *ptos,
                        INT8U prio,
                        INT16U id,
                        OS_STK *pbos,
                        INT32U stk_size,
                        void *pext,
                        INT16U opt)
```

- **函数描述：**
用于创建一个任务
- **函数形参：**

函数 OSTaskCreatExt 具有 9 个形参，如表 63.3.1.1 所示：

| 参数 | 描述 | |
|----------------------|---------------------|--------------------------|
| (*task)(void *p_arg) | 指向任务的指针 | |
| *p_arg | 传递给任务的参数 | |
| *ptos | 指向任务堆栈栈顶的指针 | |
| prio | 任务的优先级 | |
| id | 任务的标识 | |
| *pbos | 任务堆栈栈底的指针 | |
| stk_size | 任务堆栈的容量 | |
| *pext | 指向附加数据域的指针 | |
| opt | 用于设定操作选项 | |
| | OS_OPT_TASK_NONE | 表示没有任何选项。 |
| | OS_OPT_TASK_STK_CHK | 指定是否允许检测该任务的堆栈。 |
| | OS_OPT_TASK_STK_CLR | 指定是否清除该任务的堆栈。 |
| | OS_OPT_TASK_SAVE_FP | 指定是否存储浮点寄存器，CPU 需要有浮点运算。 |

- **函数返回值：**
OS_ERR_NONE：函数调用成功

OS_ERR_PRIO_EXIST: 具有该优先级的任务已经存在
 OS_ERR_PRIO_INVALID: 参数指定的优先级大于最大优先级
 OS_ERR_TASK_CREATE_ISR: 在 ISR 中创建任务
 OS_ERR_ILLEGAL_CREATE_RUN_TIME: 尝试在安全关键操作启动后创建任务

- **注意事项:**

- 1, 任务必须被创建在多任务开始之前或者运行的任务中
- 2, 任务不能由 ISR 创建
- 3, 任务必须在死循环中, 并且不能返回

2. OSTaskSuspend 函数

任务挂起函数, 其声明如下:

```
INT8U OSTaskSuspend (INT8U prio)
```

- **函数描述:**

用于将任务挂起

- **函数形参:**

Prio: 要挂起任务的优先级。

- **函数返回值:**

OS_ERR_NONE: 函数调用成功
 OS_ERR_TASK_SUSPEND_IDLE: 挂起空闲任务
 OS_ERR_PRIO_INVALID: 参数指定的优先级大于最大优先级
 OS_ERR_TASK_SUSPEND_PRIO: 需要挂起的任务不存在
 OS_ERR_TASK_NOT_EXISTS: 任务被分配到一个互斥执行

3. OSTaskDel 函数

删除任务函数, 其声明如下:

```
INT8U OSTaskDel (INT8U prio)
```

- **函数描述:**

用于删除任务

- **函数形参:**

Prio: 要删除任务的优先级。如果任务不知道自己优先级, 还可以传递参数 OS_PRIO_SELF。
 被删除的任务将回到休眠状态。

- **函数返回值:**

OS_ERR_NONE: 函数调用成功
 OS_ERR_TASK_DEL_IDLE: 删除空闲任务
 OS_ERR_PRIO_INVALID: 参数指定的优先级大于最大优先级
 OS_ERR_TASK_DEL: 任务被分配给互斥量执行
 OS_ERR_TASK_NOT_EXIST: 要删除的任务不存在
 OS_ERR_TASK_DEL_ISR: 在中断处理函数中删除任务

4. OSInit 函数

UCOSII 系统初始化函数, 其声明如下:

```
void OSInit (void)
```

- **函数描述:**

用于初始化 UCOSII 内部

- **函数形参:**

无

- **函数返回值:**

无

5. OSStart 函数

多任务启动函数, 其声明如下:

```
void OSStart (void)
```

- **函数描述:**

用于启动多任务

- **函数形参:**

无

- 函数返回值:

无

- 注意事项:

多任务的启动是通过调用 OSStart 实现的，而在启动 UCOSII 之前至少需要建立一个应用任务。

63.3.2 程序流程图

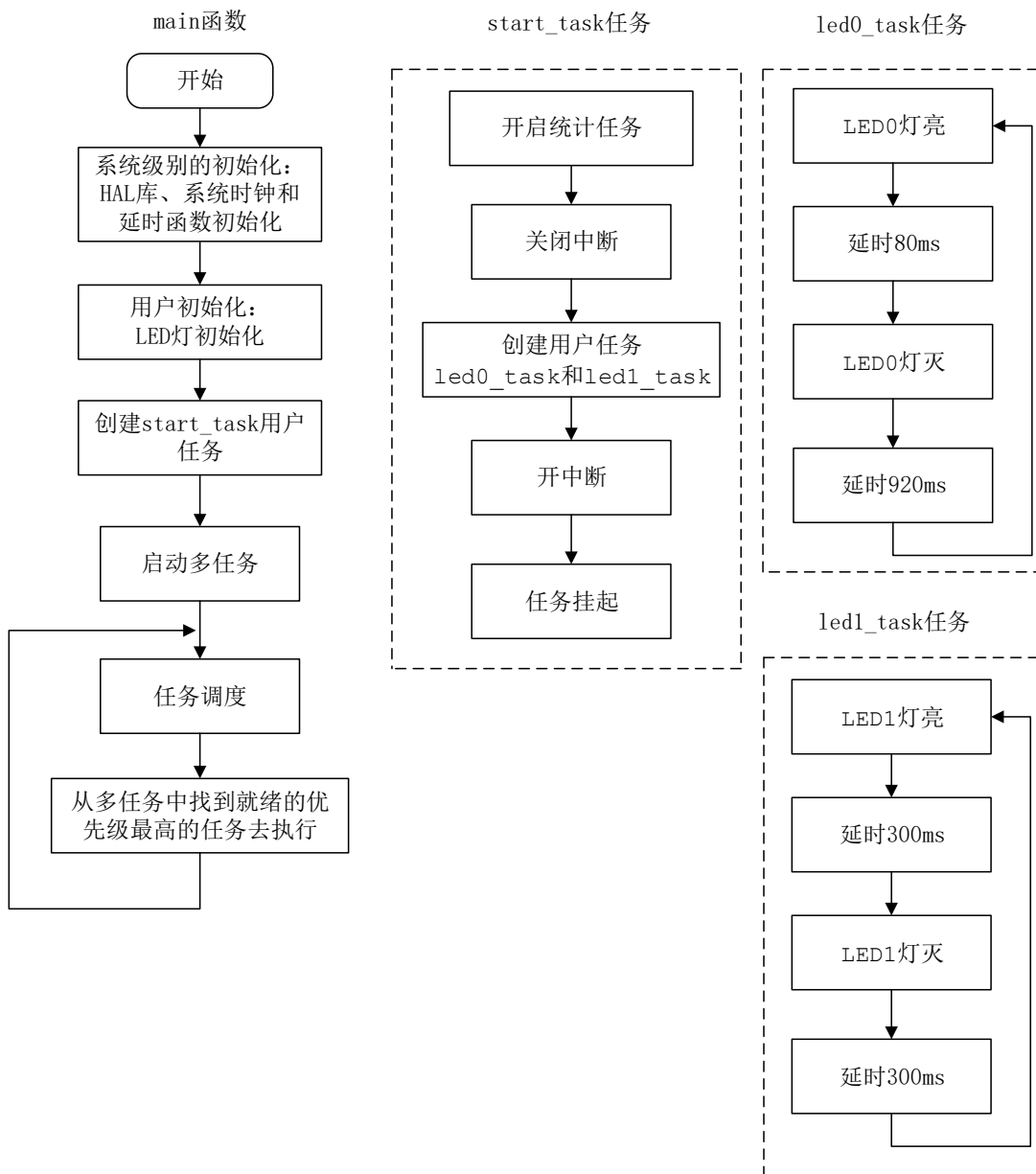


图 63.3.2.1 UCOSII 任务调度实验

我们在 main 函数中进行外设和 UCOS 的初始化后，通过创建起始任务，并在起始任务创建 LED0 和 LED1 任务。任务初始化完成后，程序不断在 LED0 和 LED1 任务间切换。

63.3.3 程序解析

在 STM32 上运行 UCOSII 的步骤:

1、移植 UCOSII

要使得 UCOSII 在 STM32 上正常运行，首先需要移植 UCOSII，这部分我们已经为大家做好了。（要学习更详细的 UCOS 移植方法，可以参考我们关于 UCOS 的专门教程）

这里我们需要注意的一个地方，SYSTEM 文件夹里面的系统函数直接支持 UCOSII，只需要在 sys.h 文件里将：SYSTEM_SUPPORT_UCOS 宏定义改为 1，即可通过 delay_init 函数初始化 UCOSII 的系统时钟节拍，为 UCOSII 提供时钟节拍。

2、编写任务函数并设置其堆栈大小和优先级等参数

编写任务函数，以便 UCOSII 调用。

设置函数堆栈大小，这个需要根据函数的需求来设置，如果任务函数的局部变量多，嵌套层数多，那么对应的堆栈就得大一些，如果堆栈设置小了，很可能出现的结果就是 CPU 进入 HardFault，遇到这种情况，你就必须把堆栈设置大一点了。另外，有些地方还需要注意堆栈字节对齐的问题，如果任务运行出现莫名其妙的错误（比如用到 sprintf 出错），请考虑是不是字节对齐的问题。

设置任务优先级，这个需要大家根据任务的重要性和实时性设置，记住高优先级的任务有优先使用 CPU 的权力。

3、初始化 UCOSII，并在 UCOSII 中创建任务

调用 OSInit，初始化 UCOSII，通过调用 OSTaskCreate 函数创建我们的任务。

4、启动 UCOSII

调用 OSStart，启动 UCOSII。

通过以上 4 个步骤，UCOSII 就开始在 STM32 上面运行了，这里还需要注意我们必须对 os_cfg.h 进行部分配置，以满足我们的需求。

1. main.c 代码

在 main.c 文件下，只初始化一些外设，然后调用 ucossii 的例程入口函数 uc_os2_demo()，如下代码所示：

```
#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/usart/usart.h"
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LED/led.h"
#include "uc-os2_demo.h"

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72M */
    delay_init(72); /* 初始化延时函数 */
    led_init(); /* 初始化 LED */
    uc_os2_demo(); /* 运行 uC/OS-II 例程 */
}
```

可以看到，在 main.c 文件中只包含了一个 main() 函数，main() 函数主要就是完成了一些外设的初始化，如系统时钟、延时、LED 等，并在最后调用了函数 uc_os2_demo()。

下面看一下 uc-os2_demo.c 的代码：

```
#include "uc-os2_demo.h"
#include "../SYSTEM/usart/usart.h"
#include "../BSP/LED/led.h"

/*uC/OS-II
*****
*****/
#include "os.h"
#include "cpu.h"

/*****
*****/
/* UCOSII 任务设置 */

/* START 任务 配置
```

```

* 包括：任务优先级 堆栈大小 等
*/
#define START_TASK_PRIO          10          /* 开始任务的优先级设置为最低 */
#define START_STK_SIZE           128         /* 堆栈大小 */

OS_STK START_TASK_STK[START_STK_SIZE];      /* 任务堆栈 */
void start_task(void *pdata);               /* 任务函数 */

/* LED0 任务 配置
* 包括：任务优先级 堆栈大小 等
*/
#define LED0_TASK_PRIO           7           /* 开始任务的优先级设置为最低 */
#define LED0_STK_SIZE            128         /* 堆栈大小 */
OS_STK LED0_TASK_STK[LED0_STK_SIZE];        /* 任务堆栈 */
void led0_task(void *pdata);                /* 任务函数 */

/* LED1 任务 配置
* 包括：任务优先级 堆栈大小 等
*/
#define LED1_TASK_PRIO           6           /* 开始任务的优先级设置为最低 */
#define LED1_STK_SIZE            128         /* 堆栈大小 */
OS_STK LED1_TASK_STK[LED0_STK_SIZE];        /* 任务堆栈 */
void led1_task(void *pdata);                /* 任务函数 */

/*****
*****/
/**
* @brief      uc/OS-II 例程入口函数
* @param      无
* @retval     无
*/
void uc_os2_demo(void)
{
    OSInit();                               /* UCOS 初始化 */
    OSTaskCreateExt((void(*) (void *))start_task, /* 任务函数 */
                    (void *)0,                /* 传递给任务函数的参数 */
                    (OS_STK *) &START_TASK_STK[START_STK_SIZE - 1],
                    (INT8U) START_TASK_PRIO,   /* 任务优先级 */
                    (INT16U) START_TASK_PRIO,  /* 任务 ID */
                    (OS_STK *) &START_TASK_STK[0], /* 任务堆栈栈底 */
                    (INT32U) START_STK_SIZE,   /* 任务堆栈大小 */
                    (void *)0,                 /* 用户补充的存储区 */
                    /* 任务选项,为了保险起见,所有任务都保存浮点寄存器的值 */
                    (INT16U) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);

    OSStart();                               /* 开始任务 */

    for (;;)
    {
        /* 不会进入这里 */
    }
}

/**
* @brief      开始任务
* @param      无
* @retval     无
*/
void start_task(void *pdata)
{

```

```

OS_CPU_SR cpu_sr = 0;
CPU_INT32U cnts;

OSStatInit(); /* 开启统计任务 */

/* 根据配置的节拍频率配置 SysTick */
cnts = (CPU_INT32U)(HAL_RCC_GetSysClockFreq() / OS_TICKS_PER_SEC);
OS_CPU_SysTickInit(cnts);
OS_ENTER_CRITICAL(); /* 进入临界区(关闭中断) */
/* LED0 任务 */
OSTaskCreateExt((void*)(void *) )led0_task,
                (void *) 0,
                (OS_STK *) &LED0_TASK_STK[LED0_STK_SIZE - 1],
                (INT8U) LED0_TASK_PRIO,
                (INT16U) LED0_TASK_PRIO,
                (OS_STK *) &LED0_TASK_STK[0],
                (INT32U) LED0_STK_SIZE,
                (void *) 0,
                (INT16U) ) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

/* LED1 任务 */
OSTaskCreateExt((void*)(void *) )led1_task,
                (void *) 0,
                (OS_STK *) &LED1_TASK_STK[LED1_STK_SIZE - 1],
                (INT8U) LED1_TASK_PRIO,
                (INT16U) LED1_TASK_PRIO,
                (OS_STK *) &LED1_TASK_STK[0],
                (INT32U) LED1_STK_SIZE,
                (void *) 0,
                (INT16U) ) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

OS_EXIT_CRITICAL(); /* 退出临界区(开中断) */
OSTaskSuspend(START_TASK_PRIO); /* 挂起开始任务 */
}

/**
 * @brief      LED0 任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void led0_task(void *pdata)
{
    while (1)
    {
        LED0(0);
        OSTimeDly(80);
        LED0(1);
        OSTimeDly(920);
    }
}

/**
 * @brief      LED1 任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void led1_task(void *pdata)
{
    while (1)
    {
        LED1(0);

```

```

        OSTimeDly(300);
        LED1(1);
        OSTimeDly(300);
    }
}

```

对于 uc-os2_demo.c 文件, 这里先简单介绍一下这个文件的代码结构, 这个文件的代码结构可分为五个部分, 分别是包含头文件、 μ C/OS-II 相关配置、应用程序入口函数、开始任务入口函数、其他任务入口函数, 接下来分别地介绍以上一个部分的代码。

① 包含的头文件分成两个部分, 分别为 μ C/OS-II 相关的头文件和其他头文件, 如下所示:

```

#include "uc-os2_demo.h"
#include "../SYSTEM/usart/usart.h"
#include "../BSP/LED/led.h"

/*uC/OSII***** */
#include "os.h"
#include "cpu.h"

```

② μ C/OS-II 的配置主要包括所创建 μ C/OS-II 任务的相关定义(任务优先级、任务栈大小、任务栈、任务函数)以及定义应用于全局的 μ C/OS-II 相关变量(信号量、事件标志、消息队列、软件定时器等), 如下所示:

```

/* START 任务 配置
 * 包括: 任务优先级 堆栈大小 等
 */
#define START_TASK_PRIO          10      /* 开始任务的优先级设置为最低 */
#define START_STK_SIZE           128     /* 堆栈大小 */

OS_STK START_TASK_STK[START_STK_SIZE]; /* 任务堆栈 */
void start_task(void *pdata);           /* 任务函数 */

/* LED0 任务 配置
 * 包括: 任务优先级 堆栈大小 等
 */
#define LED0_TASK_PRIO           7       /* 开始任务的优先级设置为最低 */
#define LED0_STK_SIZE            128     /* 堆栈大小 */
OS_STK LED0_TASK_STK[LED0_STK_SIZE];   /* 任务堆栈 */
void led0_task(void *pdata);            /* 任务函数 */

/* LED1 任务 配置
 * 包括: 任务优先级 堆栈大小 等
 */
#define LED1_TASK_PRIO           6       /* 开始任务的优先级设置为最低 */
#define LED1_STK_SIZE            128     /* 堆栈大小 */
OS_STK LED1_TASK_STK[LED0_STK_SIZE];   /* 任务堆栈 */
void led1_task(void *pdata);            /* 任务函数 */

```

③ 这部分就是函数 uc_os2_demo(), 函数 uc_os2_demo() 一开始就是对 μ C/OS-II 内核进行初始化, 接着创建一个开始任务, 最后开启 μ C/OS-II 内核的任务调度, 如下所示:

```

/**
 * @brief      uC/OS-II 例程入口函数
 * @param      无
 * @retval     无
 */
void uc_os2_demo(void)
{
    OSInit(); /* UCOS 初始化 */
    OSTaskCreateExt((void(*) (void *)) start_task, /* 任务函数 */
                    (void *) 0, /* 传递给任务函数的参数 */
                    (OS_STK *) &START_TASK_STK[START_STK_SIZE - 1],
                    (INT8U) START_TASK_PRIO, /* 任务优先级 */
                    (INT16U) START_TASK_PRIO, /* 任务 ID */
                    (OS_STK *) &START_TASK_STK[0], /* 任务堆栈栈底 */
                    0);
}

```



```

        (INT32U      ) START_STK_SIZE,          /* 任务堆栈大小 */
        (void *      ) 0,                      /* 用户补充的存储区 */
        /* 任务选项, 为了保险起见, 所有任务都保存浮点寄存器的值 */
        (INT16U      ) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

OSStart();                                     /* 开始任务 */

for (;;)
{
    /* 不会进入这里 */
}

```

④ 这部分就是开始任务的入口函数, 开始任务主要用于初始化 μ C/CPU 组件、配置 μ C/OS-II 内核节拍的时基定时器、并且还会创建其他用于实验演示的任务, 如下所示:

```

/**
 * @brief      开始任务
 * @param      无
 * @retval     无
 */
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr = 0;
    CPU_INT32U cnts;

    OSStatInit();                             /* 开启统计任务 */

    /* 根据配置的节拍频率配置 SysTick */
    cnts = (CPU_INT32U)(HAL_RCC_GetSysClockFreq() / OS_TICKS_PER_SEC);
    OS_CPU_SysTickInit(cnts);
    OS_ENTER_CRITICAL();                       /* 进入临界区(关闭中断) */
    /* LED0 任务 */
    OSTaskCreateExt((void(*) (void *)) led0_task,
                    (void *) 0,
                    (OS_STK *) &LED0_TASK_STK[LED0_STK_SIZE - 1],
                    (INT8U) LED0_TASK_PRIO,
                    (INT16U) LED0_TASK_PRIO,
                    (OS_STK *) &LED0_TASK_STK[0],
                    (INT32U) LED0_STK_SIZE,
                    (void *) 0,
                    (INT16U) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

    /* LED1 任务 */
    OSTaskCreateExt((void(*) (void *)) led1_task,
                    (void *) 0,
                    (OS_STK *) &LED1_TASK_STK[LED1_STK_SIZE - 1],
                    (INT8U) LED1_TASK_PRIO,
                    (INT16U) LED1_TASK_PRIO,
                    (OS_STK *) &LED1_TASK_STK[0],
                    (INT32U) LED1_STK_SIZE,
                    (void *) 0,
                    (INT16U) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

    OS_EXIT_CRITICAL();                       /* 退出临界区(开中断) */
    OSTaskSuspend(START_TASK_PRIO);          /* 挂起开始任务 */
}

```

从上面的代码可以看到 start_task 函数中创建了 led0_task 和 led1_task 两个任务, 创建这两个任务后, 将自己挂起。

我们单独创建 `start_task` 的目的是为了提供一个单一任务,实现应用程序开始之前的准备工作,比如外设初始化,创建任务,初始化统计任务,以及后面讲到的创建信号量、创建邮箱、创建消息队列、创建信号量集等。

在应用程序中经常有一些代码段必须不受任何干扰地连续运行,这样的代码段叫做临界段(或临界区)。因此,为了使临界段在运行时不受中断所打断,在临界段代码前必须用关中断指令使 CPU 屏蔽中断请求,而在临界段代码后必须用开中断指令接触屏蔽使得 CPU 可以响应中断请求。因为临界段代码不能被中断打断,将严重影响系统的实时性,所以临界段代码越短越好!

在 `start_task` 任务中,我们在创建 `led0_task` 和 `led1_task` 的时候,不希望中断打断,故使用了临界区。

⑤ 这部分包含了实验中用于演示的任务入口函数,如下所示:

```
/**
 * @brief      LED0 任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void led0_task(void *pdata)
{
    while (1)
    {
        LED0(0);
        OSTimeDly(80);
        LED0(1);
        OSTimeDly(920);
    }
}

/**
 * @brief      LED1 任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void led1_task(void *pdata)
{
    while (1)
    {
        LED1(0);
        OSTimeDly(300);
        LED1(1);
        OSTimeDly(300);
    }
}
```

上述两个任务较为简单,分别控制 `led0` 与 `led1` 不同频率的翻转,所以就不多做介绍。

以上就是 `uc_os2_demo.c` 文件的代码结构,本教程配套的 3 个 `ucosii` 的实验例程都会按照这个代码结构进行实验代码的编写,建议读者先熟悉以下这个代码结构。

另外,一个任务里面一般是必须有延时函数的,以释放 CPU 使用权,否则可能导致低优先级的任务因高优先级的任务不释放 CPU 使用权而一直无法得到 CPU 使用权,从而无法运行。

软件设计部分就为大家介绍到这里。

63.4 下载验证

将程序下载到开发板后,可以看到 `LED0` 一秒钟闪一次,而 `LED1` 则以固定的频率闪烁。说明两个任务(`led0_task` 和 `led1_task`)都已经正常运行,符合我们预期的设计。

第六十四章 UCOSII 实验 2-信号量和邮箱

上一章，我们学习了如何使用 UCOSII，学习了 UCOSII 的任务调度，但是并没有用到任务间的同步与通信，本章我们将学习两个最基本的任务间通讯方式：信号量和邮箱。本章分为如下几个小节：

- 64.1 UCOSII 信号量和邮箱简介
- 64.2 硬件设计
- 64.3 程序设计
- 64.4 下载验证

64.1 UCOSII 信号量和邮箱简介

系统中的多个任务在运行时，经常需要互相无冲突地访问同一个共享资源，或者需要互相支持和依赖，甚至有时还要互相加以必要的限制和制约，才能保证任务的顺利运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间可以无冲突、流畅地同步运行，而不导致灾难性的后果。

例如，任务 A 和任务 B 共享一台打印机，如果系统已经把打印机分配给了任务 A，则任务 B 因不能获得打印机的使用权而应该处于等待状态，只有当任务 A 把打印机释放后，系统才能唤醒任务 B，使其获得打印机的使用权。如果这两个任务不这样做，那么会造成极大的混乱。

任务间的同步依赖于任务间的通信。在 UCOSII 中，是使用信号量、邮箱（消息邮箱）和消息队列，这些被称作事件的中间环节来实现任务之间的通信的。这里我们仅介绍信号量和邮箱，消息队列将会在下一章介绍。

事件

两个任务通过事件进行通讯的示意图如图 64.1.1 所示：



图 64.1.1 两个任务使用事件进行通信的示意图

在上图中任务 1 是发信方，任务 2 是收信方。任务 1 负责把信息发送到事件上，这项操作叫做发送事件。任务 2 通过读取事件操作对事件进行查询：如果有信息则读取，否则等待。读事件操作叫做请求事件。

为了把描述事件的数据结构统一起来，UCOSII 使用叫做事件控制块(ECB)的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这些事件。事件控制块中包含包括等待任务表在内的所有有关事件的数据，事件控制块结构体定义如下：

```

typedef struct os_event {
    INT8U    OSEventType;          /* 事件的类型 */
    void     *OSEventPtr;          /* 消息或消息队列的指针 */
    INT16U   OSEventCnt;           /* 信号量计数器 */
    OS_PRIO  OSEventGrp;           /* 等待事件的任务组 */
    OS_PRIO  OSEventTbl[OS_EVENT_TBL_SIZE]; /* 任务等待表 */
#ifdef OS_EVENT_NAME_EN > 0u
    INT8U    *OSEventName;        /* 事件名 */
#endif
} OS_EVENT;
  
```

信号量

使用信号量的最初目的，是为了给共享资源设立一个标志，该标志表示该共享资源的占用情况。这样，当一个任务在访问共享资源之前，就可以先对这个标志进行查询，从而在了解资源被占用的情况之后，再来决定自己的行为。

信号量的实质是一个全局计数器的实现机制，释放信号量的任务使得该计数器的值加 1，请求到信号量的任务使得该计数器的值减 1。如果计数器的值为 0，则请求该信号量得任务将挂起等待，直到别的任务释放该信号量。通过这种方式，使得释放信号量的任务可以控制请求信号量的任务的运行。

信号量的工作原理如图 64.1.2 所示：

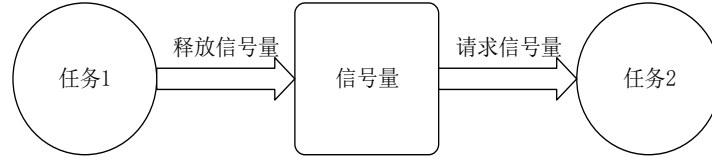


图 64.1.2 信号量工作原理图

信号量可以分为两种：一种是二值型信号量，另外一种是 N 值信号量。

UCOSII 将二值型信号量称之为也叫互斥型信号量，将 N 值信号量称之为计数型信号量，也就是普通的信号量。

信号量相关的主要操作有：创建信号量 `OSSemCreate`、请求信号量 `OSSemPend`、释放信号量 `OSSemPost` 和删除信号量 `OSSemDel`。后面再对这几个函数进行讲解。

邮箱

在多任务操作系统中，常常需要在任务与任务之间通过传递一个数据（这种数据叫做“消息”）的方式来进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区。如果把这个缓冲区称之为消息缓冲区，这样在任务间传递数据（消息）的最简单办法就是传递消息缓冲区的指针。我们把用来传递消息缓冲区指针的数据结构叫做邮箱（消息邮箱）。消息邮箱的工作情况如图 64.1.3 所示：

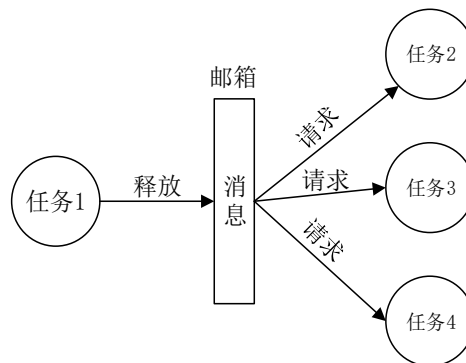


图 64.1.3 消息邮箱工作情况图

从上图可知道，只有任务才能请求消息，邮箱里仅能存放一条消息，如果释放消息的速度比请求消息的速度快，则释放的消息将会丢失。可以通过广播的方式，使得释放的消息传递给所有请求该消息邮箱的任务。如果当前邮箱为空，且有某一任务 2 正在请求邮箱，则当另一任务 1 向邮箱中释放消息时，释放的消息将直接发送给任务 2，而不用经过邮箱中转。

在 UCOSII 中，我们通过事件控制块的 `OSEventPrt` 来传递消息缓冲区指针，同时使事件控制块的成员 `OSEventType` 为常数 `OS_EVENT_TYPE_MBOX`，则该事件控制块就叫做消息邮箱。

与消息邮箱相关的主要操作有：创建邮箱函数 `OSMboxCreate`、向邮箱发送消息函数 `OSMboxPost`、请求邮箱函数 `OSMboxPend`、查询邮箱状态函数 `OSMboxQuery` 和删除邮箱函数 `OSMboxDel`。后面再对这几个进行函数进行讲解。

64.2 硬件设计

1. 例程功能

在 UCOSII 里面创建 6 个任务（不包含统计任务和空闲任务）：开始任务、LED0 任务、LED1 任务，触摸屏任务、按键扫描任务和主任务。开始任务用于创建信号量、创建邮箱、初始化统计任务以及其他任务的创建，之后挂起；LED0 任务用于 DS0 控制，提示程序运行状况；LED1 任务用于测试信号量，通过请求信号量函数，每得到一个信号量，DS1 就亮一下；触摸屏任务

用于在屏幕上画图，可以用于测试 CPU 使用率；按键扫描任务用于按键扫描，优先级最高，将得到的键值通过消息邮箱发送出去；主任务则通过查询消息邮箱获得键值，并根据键值执行信号量发送（DS1 控制）、触摸区域清屏和触摸屏校准等控制。

2. 硬件资源

1) LED 灯

LED0 - PB5

LED1 - PE5

2) 独立按键

KEY0 - PE4

KEY1 - PE3

WK_UP - PA0

3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

本章用到的硬件资源只有 LED 灯和按键。电路在开发板上已经连接好了，所以在硬件上不需要动任何东西，直接下载代码就可以测试使用。其连接原理图如图 64.2.1 所示：

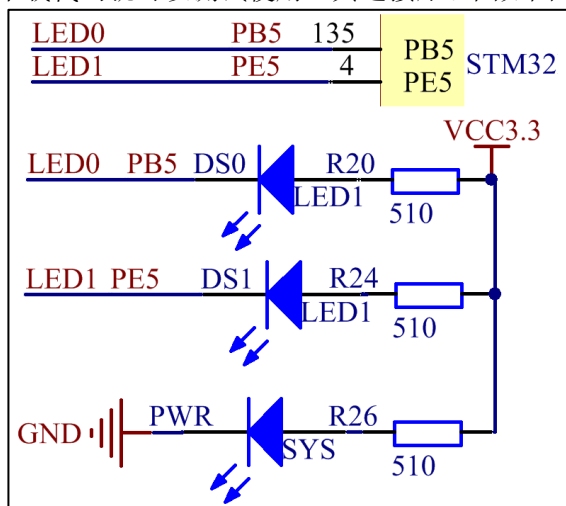


图 64.2.1 LED 与开发板连接原理图

64.3 程序设计

64.3.1 信号量函数

在这里对本实验用到的 UCOSII 信号量函数进行介绍，相关代码存放在 os_sem.c 中。

1. OSSemCreate 函数

创建信号量函数，其声明如下：

```
OS_EVENT *OSSemCreate (INT16U cnt)
```

- **函数描述：**
用于创建一个信号量
- **函数形参：**
cnt 是信号量计数器（OSEventCnt）的初始值
- **函数返回值：**
已创建的信号量的指针

2. OSSemPend 函数

请求信号量函数，其声明如下：

```
void OSSemPend (OS_EVENT *pevent, INT32U timeout, INT8U *perr)
```

- **函数描述:**
请求信号量
- **函数形参:**
perr: 错误信息
OS_ERR_NONE: 调用成功, 信号量不为零
OS_ERR_TIMEOUT: 信号量没有在指定数目的时钟周期内被设置
OS_ERR_PEND_ABOUT: 取消对信号量的等待
OS_ERR_EVENT_TYPE: 没有传递信号量的指针
OS_ERR_PEND_ISR: 从中断调用该函数时错误
OS_ERR_PEVENT_NULL: pevent 是一个空指针
OS_ERR_PEND_LOCKED: 调度器上锁了
- **函数返回值:**
无
- **注意事项:**
为了防止任务因得不到信号量而处于长期的等待状态, 函数 OSSemPend 允许用参数 timeout 设置一个等待时间的限制, 当任务等待的时间超过 timeout 时可以结束等待状态而进入就绪状态。如果参数 timeout 被设置为 0, 则表明任务的等待时间为无限长。

3. OSSemPost 函数

发送信号量函数, 其声明如下:

```
INT8U OSSemPost (OS_EVENT *pevent)
```

- **函数描述:**
用于发送信号量或者称为释放信号量
- **函数形参:**
pevent: 被请求信号量的指针
- **函数返回值:**
OS_ERR_NONE: 函数调用成功, 信号量被成功地设置
OS_ERR_SEM_OVF: 信号量的值溢出
OS_ERR_EVENT_TYPE: pevent 不是指向信号量的指针
OS_ERR_PEVENT_NULL: pevent 是一个空指针

4. OSSemDel 函数

删除信号量函数, 其声明如下:

```
OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

- **函数描述:**
用于删除信号量并准备挂起所有任务
- **函数形参:**
pevent: 要删除的信号量指针
opt: 删除条件选项
OS_DEL_NO_PEND: 在没有任务挂起时删除信号量
OS_DEL_ALWAYS: 删除信号量, 即使任务正在等待。
perr: 错误信息
OS_ERR_NONE: 函数调用成功, 成功删除信号量
OS_ERR_DEL_ISR: 尝试在中断中删除信号量
OS_ERR_INVALID_OPT: 指向一个无效的选项
OS_ERR_TASK_WAITING: 一个或多个任务在等待这个信号量
OS_ERR_EVENT_TYPE: 没有传递一个指向信号量的指针
OS_ERR_PEVENT_NULL: pevent 是一个空指针
- **函数返回值:**
pevent: 存在错误 (OS_EVENT *)0: 该信号量被成功删除。

消息邮箱函数

在这里对本实验用到的 UCOSII 消息邮箱函数进行介绍，相关代码存放在 os_mbox.c 中

1. OSMboxCreate 函数

创建邮箱函数，其声明如下：

```
OS_EVENT *OSMboxCreate (void *pmsg)
```

- **函数描述：**
用于创建邮箱函数
- **函数形参：**
pmsg: 消息的指针
- **函数返回值：**
消息邮箱的指针
- **注意事项：**
调用 OSMboxCreate 前，需先定义 msg 的初始值。在一般情况下，这个初始值为 NULL。但也可以事先定义一个邮箱，然后把这个邮箱的指针作为参数传递到函数 OSMboxCreate 中，使得一开始就指向一个邮箱。

2. OSMboxPost 函数

向邮箱发送消息函数，其声明如下：

```
INT8U OSMboxPost (OS_EVENT *pevent, void *pmsg)
```

- **函数描述：**
用于向消息邮箱发送消息
- **函数形参：**
pevent: 消息邮箱的指针
pmsg : 消息指针
- **函数返回值：**
OS_NO_ERR: 消息发送成功
OS_MBOX_FULL: 不能向满邮箱再发送消息
OS_ERR_EVENT_TYPE: 指定的事件不是消息邮箱类型
OS_ERR_PEVENT_NULL: 不能向不存在的消息邮箱发送消息
OS_ERR_POST_NULL_PTR: 消息缓冲区不能为空

3. OSMboxPend 函数

请求邮箱函数，其声明如下：

```
void *OSMboxPend (OS_EVENT *pevent, INT32U timeout, INT8U *perr)
```

- **函数描述：**
请求消息邮箱，就是等待一个消息传送到消息邮箱或取得一个消息数据。
- **函数形参：**
pevent: 消息邮箱的指针
timeout: 等待时限
perr: 错误信息
- **函数返回值：**
NULL: 未得到消息或者 pevent 指针 ! NULL: 预期消息的指针

4. OSMboxQuery 函数

查询邮箱状态函数，其声明如下：

```
UINT8U *OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *p_mbox_data)
```

- **函数描述：**
获取消息邮箱的相关信息
- **函数形参：**
pevent: 消息邮箱的指针
p_mbox_data: 存放邮箱信息的结构
- **函数返回值：**
OS_ERR_NONE: 调用成功

OS_ERR_EVENT_TYPE: pevent 不是指向消息邮箱的指针
 OS_ERR_PEVENT_NULL: 不能向不存在的消息邮箱发送消息
 OS_ERR_PDATA_NULL: p_mbox_data 是一个空指针

- **注意事项:**

必须先建立消息邮箱，然后使用

5.OSMboxDel 函数

删除邮箱函数，其声明如下：

```
OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

- **函数描述:**

对一个不再使用的消息邮箱及时删除以释放资源

- **函数形参:**

pevent: 要删除的邮箱指针

opt: 删除条件选项

OS_DEL_NO_PEND: 在没有任务挂起时删除邮箱

OS_DEL_ALWAYS: 无条件删除邮箱，所有等待该事件的任务转到就绪态。

perr: 错误信息

OS_ERR_NONE: 函数调用成功，成功删除邮箱

OS_ERR_DEL_ISR: 不支持在中断中删除邮箱

OS_ERR_INVALID_OPT: 指向一个无效的选项

OS_ERR_TASK_WAITING: 一个或多个任务在等待这个信号量

OS_ERR_EVENT_TYPE: 没有传递一个指向邮箱的指针

OS_ERR_PEVENT_NULL: pevent 是一个空指针

- **函数返回值:**

pevent : 存在错误

(OS_EVENT *)0 : 该邮箱被成功删除。

64.3.2 程序流程图

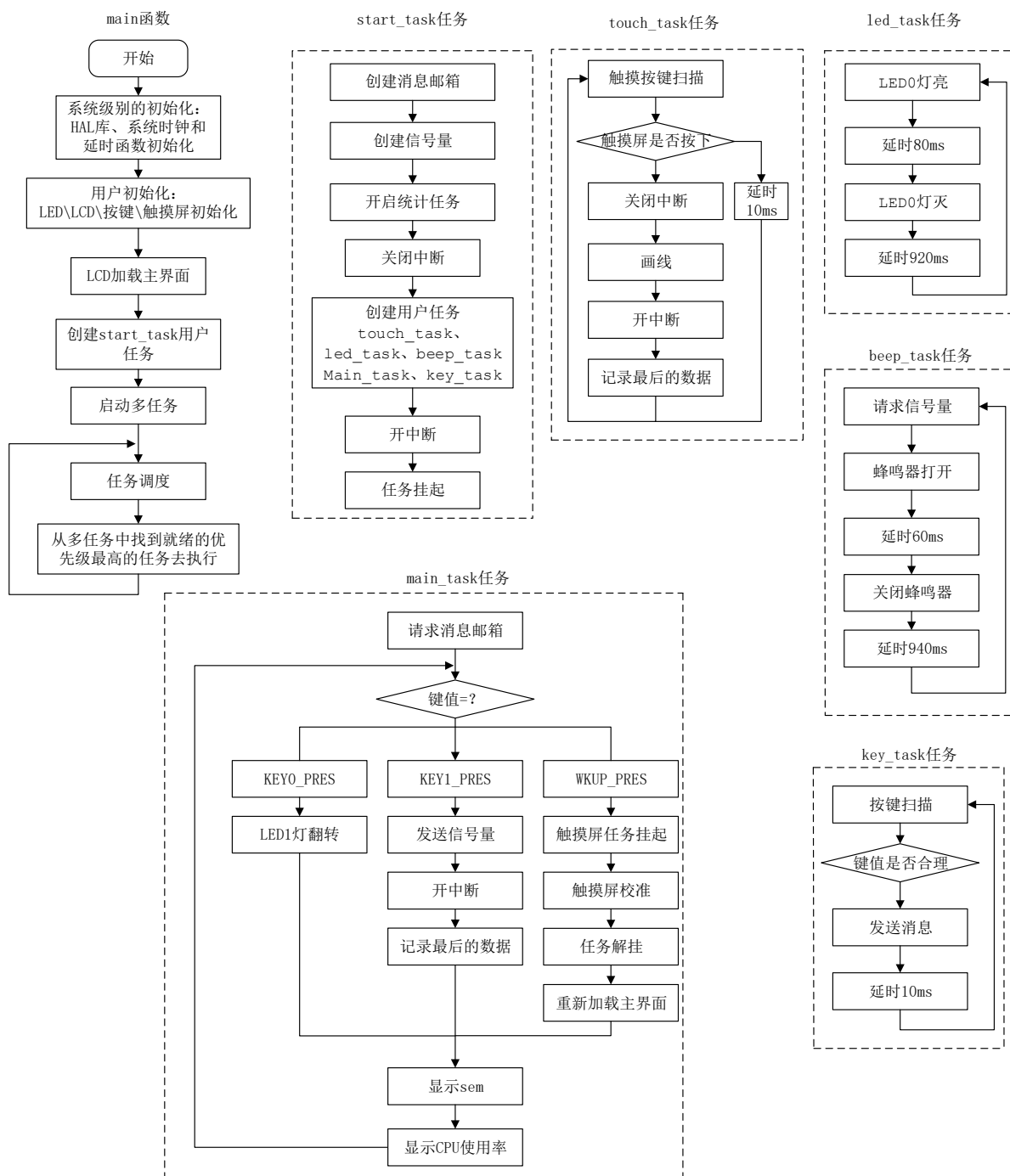


图 64.3.2.1 UCOSII 信号量和邮箱实验

64.3.3 程序解析

1. main.c 代码

在 main.c 文件下，只初始化一些外设，然后调用 ucossii 的例程入口函数 uc_os2_demo()，如下代码所示：

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

```
#include "../BSP/LED/led.h"
#include "../BSP/KEY/key.h"
#include "../BSP/LCD/lcd.h"
#include "../BSP/BEEP/beep.h"
#include "../BSP/TOUCH/touch.h"
#include "uc-os2_demo.h"

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72M */
    delay_init(72); /* 初始化延时函数 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    beep_init(); /* 初始化蜂鸣器 */
    tp_dev.init(); /* 触摸屏初始化 */
    uc_os2_demo(); /* 运行 uC/OS-II 例程 */
}
```

可以看到，在 `main.c` 文件中只包含了一个 `main()` 函数，`main()` 函数主要就是完成了一些外设的初始化，如串口、LED、LCD 等，并在最后调用了函数 `uc_os2_demo()`。下面看一下 `uc-os2_demo.c` 的代码：

```
/* UCOSII 任务设置 */

/* START 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define START_TASK_PRIO 10 /* 开始任务的优先级设置为最低 */
#define START_STK_SIZE 128 /* 堆栈大小 */

OS_STK START_TASK_STK[START_STK_SIZE]; /* 任务堆栈 */
void start_task(void *pdata); /* 任务函数 */

/* 触摸屏任务 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define TOUCH_TASK_PRIO 7 /* 优先级设置 (越小优先级越高) */
#define TOUCH_STK_SIZE 128 /* 堆栈大小 */

OS_STK TOUCH_TASK_STK[TOUCH_STK_SIZE]; /* 任务堆栈 */
void touch_task(void *pdata); /* 任务函数 */

/* LED 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define LED_TASK_PRIO 6 /* 优先级设置 (越小优先级越高) */
#define LED_STK_SIZE 128 /* 堆栈大小 */

OS_STK LED_TASK_STK[LED_STK_SIZE]; /* 任务堆栈 */
void led_task(void *pdata); /* 任务函数 */

/* 蜂鸣器任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define BEEP_TASK_PRIO 5 /* 优先级设置 (越小优先级越高) */
#define BEEP_STK_SIZE 128 /* 堆栈大小 */

OS_STK BEEP_TASK_STK[BEEP_STK_SIZE]; /* 任务堆栈 */
void beep_task(void *pdata); /* 任务函数 */
```

```

/* 主任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define MAIN_TASK_PRIO          4          /* 优先级设置 (越小优先级越高) */
#define MAIN_STK_SIZE          512        /* 堆栈大小 */

OS_STK MAIN_TASK_STK[MAIN_STK_SIZE];      /* 任务堆栈 */
void main_task(void *pdata);              /* 任务函数 */

/* 按键扫描 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define KEY_TASK_PRIO          3          /* 优先级设置 (越小优先级越高) */
#define KEY_STK_SIZE          128        /* 堆栈大小 */

OS_STK KEY_TASK_STK[KEY_STK_SIZE];        /* 任务堆栈 */
void key_task(void *pdata);              /* 任务函数 */

/*****
*****/
OS_EVENT *msg_key;                      /* 按键邮箱事件块指针 */
OS_EVENT *sem_beep;                    /* 蜂鸣器信号量指针 */

void uc_os_load_main_ui(void);
void lcd_draw_bline(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint8_t
size, uint16_t color);

/*****
*****/
/**
 * @brief      uC/OS-III 例程入口函数
 * @param      无
 * @retval     无
 */
void uc_os2_demo(void)
{
    uc_os_load_main_ui();                /* 加载主界面 */
    OSInit();                            /* UCOS 初始化 */
    OSTaskCreateExt((void(*) (void *))start_task,                /* 任务函数 */
                    /* 传递给任务函数的参数 */
                    (void *)0,
                    /* 任务堆栈栈顶 */
                    (OS_STK *) &START_TASK_STK[START_TASK_SIZE - 1],
                    (INT8U) START_TASK_PRIO,                /* 任务优先级 */
                    /* 任务 ID, 这里设置为和优先级一样 */
                    (INT16U) START_TASK_PRIO,
                    (OS_STK *) &START_TASK_STK[0],          /* 任务堆栈栈底 */
                    (INT32U) START_TASK_SIZE,                /* 任务堆栈大小 */
                    (void *)0,                                /* 用户补充的存储区 */
                    /* 任务选项, 为了保险起见, 所有任务都保存浮点寄存器的值 */
                    (INT16U) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);

    OSStart();                            /* 开始任务 */

    for (;;)
    {
        /* 不会进入这里 */
    }
}

```

```

/**
 * @brief      开始任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr = 0;
    CPU_INT32U cnts;

    msg_key = OSMboxCreate((void *)0); /* 创建消息邮箱 */
    sem_beep = OSSemCreate(0);          /* 创建信号量 */

    OSStatInit();                       /* 开启统计任务 */
    /* 根据配置的节拍频率配置 SysTick */
    cnts = (CPU_INT32U)(HAL_RCC_GetSysClockFreq() / OS_TICKS_PER_SEC);
    OS_CPU_SysTickInit(cnts);

    OS_ENTER_CRITICAL();                /* 进入临界区(关闭中断) */

    /* 触摸任务 */
    OSTaskCreateExt((void(*) (void *)) touch_task,
                    (void *) 0,
                    (OS_STK *) &TOUCH_TASK_STK[TOUCH_STK_SIZE - 1],
                    (INT8U) TOUCH_TASK_PRIO,
                    (INT16U) TOUCH_TASK_PRIO,
                    (OS_STK *) &TOUCH_TASK_STK[0],
                    (INT32U) TOUCH_STK_SIZE,
                    (void *) 0,
                    (INT16U) ) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);

    /* LED 任务 */
    OSTaskCreateExt((void(*) (void *)) led_task,
                    (void *) 0,
                    (OS_STK *) &LED_TASK_STK[LED_STK_SIZE - 1],
                    (INT8U) LED_TASK_PRIO,
                    (INT16U) LED_TASK_PRIO,
                    (OS_STK *) &LED_TASK_STK[0],
                    (INT32U) LED_STK_SIZE,
                    (void *) 0,
                    (INT16U) ) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);

    /* 蜂鸣器任务 */
    OSTaskCreateExt((void(*) (void *)) beep_task,
                    (void *) 0,
                    (OS_STK *) &BEEP_TASK_STK[BEEP_STK_SIZE - 1],
                    (INT8U) BEEP_TASK_PRIO,
                    (INT16U) BEEP_TASK_PRIO,
                    (OS_STK *) &BEEP_TASK_STK[0],
                    (INT32U) BEEP_STK_SIZE,
                    (void *) 0,
                    (INT16U) ) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);

    /* 主任务 */
    OSTaskCreateExt((void(*) (void *)) main_task,
                    (void *) 0,
                    (OS_STK *) &MAIN_TASK_STK[MAIN_STK_SIZE - 1],
                    (INT8U) MAIN_TASK_PRIO,
                    (INT16U) MAIN_TASK_PRIO,
                    (OS_STK *) &MAIN_TASK_STK[0],
                    (INT32U) MAIN_STK_SIZE,
                    (void *) 0,
                    (INT16U) ) OS_TASK_OPT_STK_CHK |
                    OS_TASK_OPT_STK_CLR |
                    OS_TASK_OPT_SAVE_FP);
}

```



```

        (INT16U      ) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

    /* 按键任务 */
    OSTaskCreateExt((void(*) (void *)) key_task,
                    (void *) 0,
                    (OS_STK *) &KEY_TASK_STK[KEY_STK_SIZE - 1],
                    (INT8U) KEY_TASK_PRIO,
                    (INT16U) KEY_TASK_PRIO,
                    (OS_STK *) &KEY_TASK_STK[0],
                    (INT32U) KEY_STK_SIZE,
                    (void *) 0,
                    (INT16U) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);

    OS_EXIT_CRITICAL(); /* 退出临界区 (开中断) */
    OSTaskSuspend(START_TASK_PRIO); /* 挂起开始任务 */
}

/**
 * @brief      LED0 任务
 * @param      pdata : 传入参数 (未用到)
 * @retval     无
 */
void led_task(void *pdata)
{
    uint8_t t;

    while (1)
    {
        t++;
        OSTimeDly(10);

        if (t == 8)
        {
            LED0(1); /* LED0 灭 */
        }

        if (t == 100)
        {
            t = 0;
            LED0(0); /* LED0 亮 */
        }
    }
}

/**
 * @brief      蜂鸣器任务
 * @param      pdata : 传入参数 (未用到)
 * @retval     无
 */
void beep_task(void *pdata)
{
    uint8_t err;

    while (1)
    {
        OSSemPend(sem_beep, 0, &err); /* 请求信号量 */
        BEEP(1); /* 打开蜂鸣器 */
        OSTimeDly(60);
        BEEP(0); /* 关闭蜂鸣器 */
        OSTimeDly(940);
    }
}

```

```

/**
 * @brief      触摸屏任务
 * @param      pdata : 传入参数 (未用到)
 * @retval     无
 */
void touch_task(void *pdata)
{
    uint32_t cpu_sr;
    uint16_t lastpos[2];
    /* 最后一次的数据 */

    while (1)
    {
        tp_dev.scan(0);
        /* 触摸屏被按下 */
        if (tp_dev.sta & TP_PRES_DOWN)
        {
            if ( tp_dev.x[0] < lcddev.width &&
                tp_dev.y[0] < lcddev.height && tp_dev.y[0] > 120)
            {
                if (lastpos[0] == 0xFFFF)
                {
                    lastpos[0] = tp_dev.x[0];
                    lastpos[1] = tp_dev.y[0];
                }
                /* 进入临界段,防止其他任务,打断 LCD 操作,导致液晶乱序 */
                OS_ENTER_CRITICAL();
                lcd_draw_bline(lastpos[0],
                               lastpos[1],
                               tp_dev.x[0],
                               tp_dev.y[0],
                               2,
                               RED); /* 画线 */
                OS_EXIT_CRITICAL();
                lastpos[0] = tp_dev.x[0];
                lastpos[1] = tp_dev.y[0];
            }
        }
        else
        {
            lastpos[0] = 0xFFFF;
            OSTimeDly(10); /* 没有按键按下时候 */
        }
    }
}

/**
 * @brief      主任务
 * @param      pdata : 传入参数 (未用到)
 * @retval     无
 */
void main_task(void *pdata)
{
    uint32_t key = 0;
    uint8_t err;
    uint8_t semmask = 0;
    uint8_t tcnt = 0;

    while (1)
    {
        key = (uint32_t)OSMboxPend(msg_key, 10, &err);

        switch (key)

```

```

{
    case KEY0_PRES:
    {
        /* 控制 DS1, 并清除触摸区域 */
        LED1_TOGGLE();
        lcd_fill(0, 121, lcddev.width - 1, lcddev.height - 1, WHITE);
        break;
    }
    case KEY1_PRES:
    {
        /* 发送信号量 */
        semmask = 1;
        OSSemPost(sem_beep);
        break;
    }
    case WKUP_PRES:
    {
        /* 校准 */
        OSTaskSuspend(TOUCH_TASK_PRIO); /* 挂起触摸屏任务 */

        if ((tp_dev.touchtype & 0X80) == 0)
        {
            tp_adjust();
        }
        OSTaskResume(TOUCH_TASK_PRIO); /* 解挂 */
        ucos_load_main_ui(); /* 重新加载主界面 */
        break;
    }
}

if (semmask || sem_beep->OSEventCnt) /* 需要显示 sem */
{
    /* 显示信号量的值 */
    lcd_show_xnum(192, 50, sem_beep->OSEventCnt, 3, 16, 0X80, BLUE);

    if (sem_beep->OSEventCnt == 0)
    {
        semmask = 0; /* 停止更新 */
    }
}

if (tcnt == 10) /* 0.6 秒更新一次 CPU 使用率 */
{
    tcnt = 0;
    /* 显示 CPU 使用率 */
    lcd_show_xnum(192, 30, OSCPUUsage, 3, 16, 0, BLUE);
}

tcnt++;
OSTimeDly(10);
}
}

/**
 * @brief      按键扫描任务
 * @param      pdata : 传入参数 (未用到)
 * @retval     无
 */
void key_task(void *pdata)
{
    uint32_t key;

    while (1)

```

```

    {
        key = key_scan(0);

        if (key)
        {
            OSMboxPost(msg_key, (void *)key); /* 发送消息 */
        }
        OSTimeDly(10);
    }
}

/**
 * @brief      加载主界面
 * @param      无
 * @retval     无
 */
void uclos_load_main_ui(void)
{
    lcd_clear(WHITE); /* 清屏 */
    lcd_show_string(30, 10, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 30, 200, 16, 16, "UCOSII TEST2", RED);
    lcd_show_string(30, 50, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 75, 200, 16, 16, "KEY0:DS1 AND CLEAR", RED);
    lcd_show_string(30, 95, 200, 16, 16, "KEY1:BEEP KEY_UP:ADJUST", RED);
    lcd_show_string(80, 210, 200, 16, 16, "Touch Area", RED);
    lcd_draw_line(0, 120, lcddev.width - 1, 120, RED);
    lcd_draw_line(0, 70, lcddev.width - 1, 70, RED);
    lcd_draw_line(150, 0, 150, 70, RED);

    lcd_show_string(160, 30, 200, 16, 16, "CPU:  %", BLUE);
    lcd_show_string(160, 50, 200, 16, 16, "SEM:000", BLUE);
}

/**
 * @brief      画粗线
 * @param      x1,y1: 起点坐标
 * @param      x2,y2: 终点坐标
 * @param      size : 线条粗细程度
 * @param      color: 线的颜色
 * @retval     无
 */
void lcd_draw_bline(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint8_t
size, uint16_t color)
{
    uint16_t t;
    int xerr = 0, yerr = 0, delta_x, delta_y, distance;
    int incx, incy, row, col;

    if (x1 < size || x2 < size || y1 < size || y2 < size)
    {
        return;
    }

    delta_x = x2 - x1; /* 计算坐标增量 */
    delta_y = y2 - y1;
    row = x1;
    col = y1;

    if (delta_x > 0)
    {
        incx = 1; /* 设置单步方向 */
    }
    else if (delta_x == 0)
    {

```

```

        incx = 0;                                /* 垂直线 */
    }
    else
    {
        incx = -1;
        delta_x = -delta_x;
    }

    if (delta_y > 0)
    {
        incy = 1;
    }
    else if (delta_y == 0)
    {
        incy = 0;                                /* 水平线 */
    }
    else
    {
        incy = -1;
        delta_y = -delta_y;
    }

    if ( delta_x > delta_y)
    {
        distance = delta_x;                        /* 选取基本增量坐标轴 */
    }
    else distance = delta_y;

    for (t = 0; t <= distance + 1; t++ ) /* 画线输出 */
    {
        /* 画点 */
        lcd_fill_circle(row, col, size, color);
        xerr += delta_x ;
        yerr += delta_y ;

        if (xerr > distance)
        {
            xerr -= distance;
            row += incx;
        }

        if (yerr > distance)
        {
            yerr -= distance;
            col += incy;
        }
    }
}

```

上面就是对创建的 `start_task`、`led0_task`、`touch_task`、`led1_task`、`main_task` 和 `key_task` 等 6 个任务的参数进行配置，例如优先级、堆栈大小和任务函数的声明及定义。这 6 个任务做的事情在前面程序流程图里面已经有说明，这里就不多说了。

这一章的运行流程比上一章复杂了一些，我们创建了消息邮箱 `msg_key`，用于按键任务和主任务之间的数据传输（传递键值）。另外创建了信号量 `sem_led1`，用于 LED 1 任务和主任务之间的通信。

在代码中，我们使用了 UCOSII 提供的 CPU 统计服务，通过 `OSStatInit` 初始化 CPU 统计任务，然后在主任务中显示 CPU 使用率。

另外，在主任务中，我们用到了任务的挂起和恢复函数，在执行触摸校准的时候，我们必须先将触摸屏任务挂起，待校准完成之后，再恢复触摸屏任务。这是因为触摸屏校准和触摸屏任务都用到了触摸屏和 TFTLCD，而这两个东西是不支持多任务占用的，所有必须采用独占的方式使用，否则可能导致数据错乱。

64.4 下载验证

将程序下载到开发板后，可以看到 LCD 显示界面如图 64.4.1 所示：

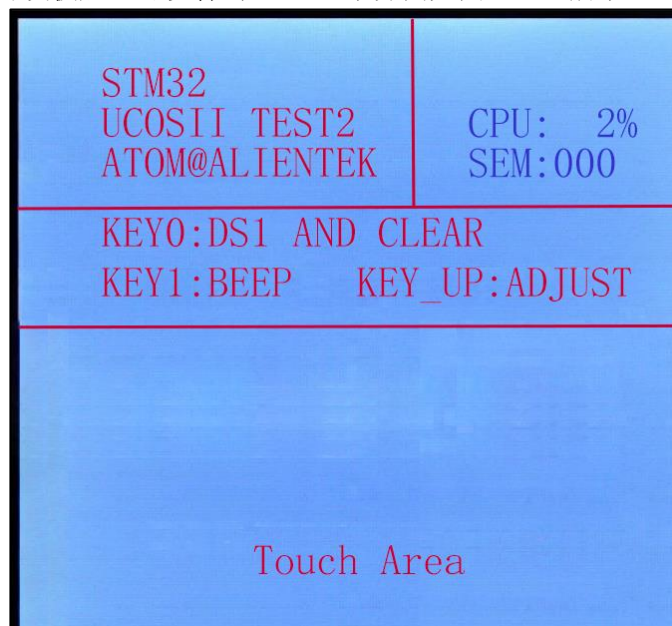


图 64.4.1 初始化界面

从上图可以看到，默认情况下，CPU 使用率仅为 1%。此时通过在触摸区域(Touch Area)画图，可以看到 CPU 使用率飙升，这说明触摸屏任务是一个很占 CPU 的任务；通过按 KEY0，可以控制 LED1 的亮灭；同时，可以清除触摸区域的笔迹；通过按下 KEY1 可以控制蜂鸣器响一次，如果多次按下，可以看到剩余信号量请求次数；通过按下 WK_UP 可以进入校准程序，对触摸屏进行校准（注意，电容触摸屏不需要校准，所以如果是电容屏，按 KEY_UP，就相当于清屏一次的效果，不会进行校准）。

第六十五章 UCOSII 实验 3-消息队列、信号量集和软件定时器

上一章，我们学习了如何使用 UCOSII 的信号量和邮箱的使用，本章，我们将学习消息队列、信号量集和软件定时器的使用。本章分为如下几个小节：

65.1 UCOSII 消息队列、信号量集和软件定时器简介

65.2 硬件设计

65.3 程序设计

65.4 下载验证

65.1 UCOSII 消息队列、信号量集和软件定时器简介

65.1.1 消息队列

消息队列可以视为消息邮箱的数组形式，消息邮箱一次传递一则消息，而消息队列可以在任务之间传递多条消息。消息队列的工作情况如图 65.1.1.1 所示：

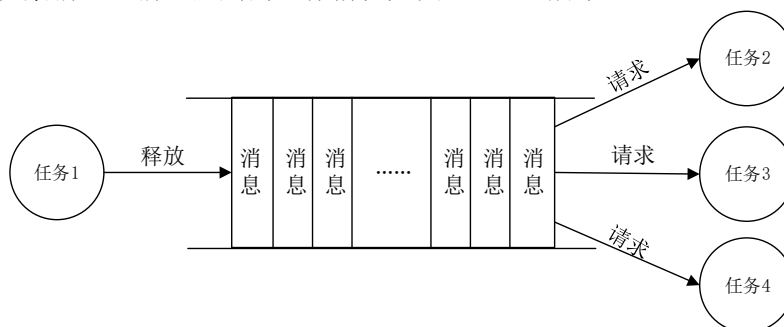


图 65.1.1.1 消息队列的工作情况图

从上图可知，任务可以向消息队列中释放消息，只有任务才能从消息队列中请求消息，任务可以始终请求消息，也可以周期性地请求消息。消息队列具有一定的长度，其长度可包含的消息个数，如果向队列中释放消息的速度大于从队列中请求消息的速度，那么消息队列将会溢出。消息队列的数据结构如图 65.1.1.2 所示：

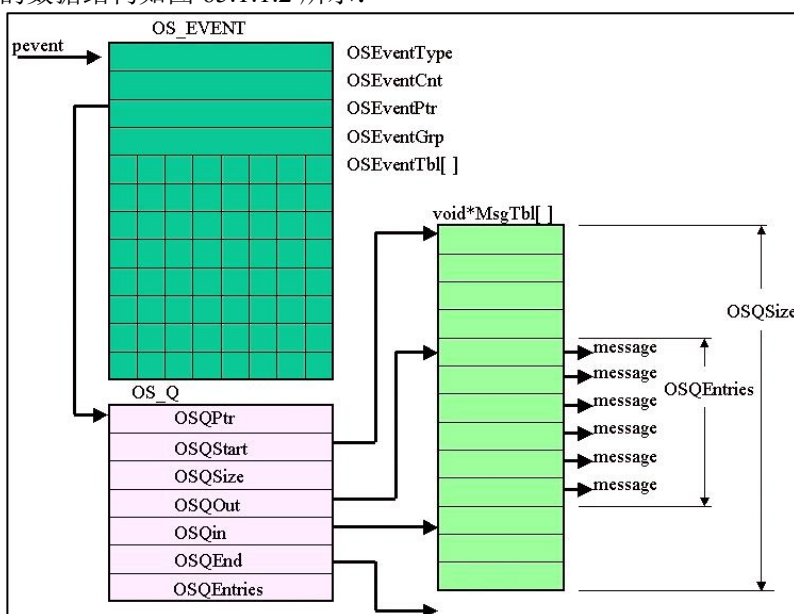


图 65.1.1.2 消息队列的数据结构图

消息队列由三个部分组成：事件控制块、消息队列和消息。当把事件控制块成员 `OSEventType` 的值设置为 `OS_EVENT_TYPE_Q` 时，该事件控制块描述的就是一个消息队列。

从上图可以知道，消息队列相当于一个任务等待列表的消息邮箱数组，事件控制块成员 `OSEventPtr` 指向了一个叫做队列控制块 (`OS_Q`) 的结构，该结构管理了一个数组 `MsgTbl[]`，该数组中的元素都是一些指向消息的指针。

队列控制块 (`OS_Q`) 的结构定义：

```
typedef struct os_q {
    struct os_q *OSQPtr;          /* 队列控制块 */
    void **OSQStart;              /* 指向下一个空的队列控制块 */
    void **OSQEnd;                /* 指向消息指针数组的起始地址 */
    void **OSQIn;                 /* 指向消息指针数组结束单元的下一个单元 */
    void **OSQOut;                /* 指向插入一条消息的位置 */
    INT16U OSQSize;               /* 指向被取出消息的位置 */
    INT16U OSQEntries;            /* 数组的长度 */
} OS_Q;                          /* 已存放消息指针的元素数目 */
```

其中，可以移动的指针为 `OSQIn` 和 `OSQOut`，而指针 `OSQStart` 和 `OSQEnd` 只是一个标志（常指针）。当可移动的指针 `OSQIn` 或 `OSQOut` 移动到数组末尾，也就是与 `OSQEnd` 相等时，可移动的指针将会被调整到数组的起始位置 `OSQStart`。从效果上看，指针 `OSQEnd` 与 `OSQStart` 等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个循环队列，如图 65.1.1.3 所示：

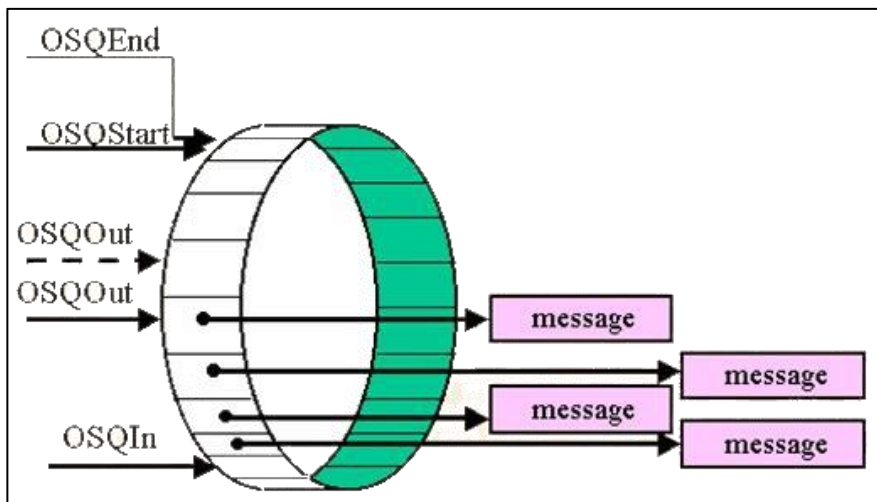


图 65.1.1.3 消息指针数组构成的环形数据缓冲区

在 UCOSII 初始化时，系统将按 `os_cfg.h` 文件中的 `OS_MAX_QS` 的数值定义 `OS_MAX_QS` 个队列控制块，并用队列控制块中的指针 `OSQPtr` 将所有队列控制块链接为链表。由于这时候还没有使用它们，所以这个链表叫做空队列控制块链表。

消息队列相关的主要操作有：创建消息队列函数 `OSQCreate`、请求消息队列函数 `OSQPend` 和向消息队列发送消息函数 `OSQPost`。后面再对这几个函数进行讲解。

消息到这里就介绍完成了，想了解更多朋友可以参考《嵌入式实时操作系统 UCOSII 原理及应用》第五章。

65.1.2 信号量集

在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。UCOSII 为了实现多个信号量组合的功能定义了一种特殊的数据结构——信号量集。

信号量集所能管理的信号量都是一些二值信号，所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如图 65.1.2.1 所示：

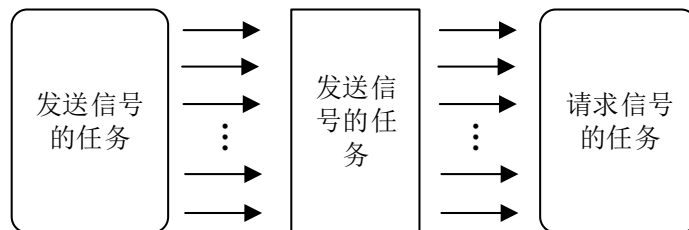


图 65.1.2.1 信号量集示意图

不同于信号量、消息邮箱、消息队列等事件，UCOSII 不使用事件控制块来描述信号量集，而使用了一个叫做标志组的结构 OS_FLAG_GRP 来描述。OS_FLAG_GRP 结构如下：

```
typedef struct os_flag_grp { /* 标志组 */
    INT8U      OSFlagType; /* 信号量集的标志 */
    void       *OSFlagWaitList; /* 指向等待任务链表的指针 */
    OS_FLAGS    OSFlagFlags; /* 所有信号列表 */
} OS_FLAG_GRP;
```

成员 OSFlagFlags 是一个指针，当一个信号量集被创建后，这个指针指向了这个信号量集的等待任务链表。

与其他前面介绍过的事件不同，信号量集用一个双向链表来组织等待任务，每一个等待任务都是该链表中的一个节点（node）。标志组 OS_FLAG_GRP 的成员 OSFlagWaitList 就指向了信号量集的这个等待任务链表。等待任务链表节点 OS_FLAG_NODE 的结构如下：

```
typedef struct os_flag_node { /* 等待任务链表节点 */
    void       *OSFlagNodeNext; /* 指向下一个节点的指针 */
    void       *OSFlagNodePrev; /* 指向前一个节点的指针 */
    void       *OSFlagNodeTCB; /* 指向对应任务控制块的指针 */
    void       *OSFlagNodeFlagGrp; /* 反向指向信号量集的指针 */
    OS_FLAGS    OSFlagNodeFlags; /* 信号过滤器 */
    INT8U      OSFlagNodeWaitType; /* 定义逻辑运算关系的数据 */
} OS_FLAG_NODE;
```

其中 OSFlagNodeWaitType 是定义逻辑运算关系的一个常数（根据需要设置），其可选值和对应的逻辑关系如表 65.1.2.1 所示：

| 常数 | 信号有效状态 | 等待任务的就绪条件 |
|-----------------------------|--------|--------------------|
| WAIT_CLR_ALL 或 WAIT_CLR_AND | 0 | 信号全部有效（全 0） |
| WAIT_CLR_ANY 或 WAIT_CLR_OR | 0 | 信号有一个或一个以上有效（有 0） |
| WAIT_SET_ALL 或 WAIT_SET_AND | 1 | 信号全部有效（全 1） |
| WAIT_SET_ANY 或 WAIT_SET_OR | 1 | 信号有一个或者一个以上有效（有 1） |

表 65.1.2.1 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、OSFlagNodeFlags、OSFlagNodeWaitType 三者的关系如图 65.1.2.2 所示：

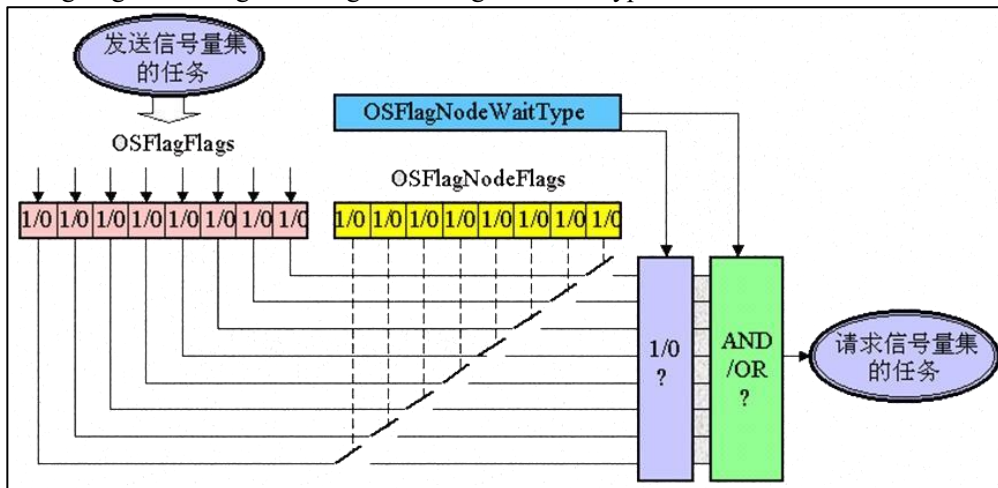


图 65.1.2.2 标志组与等待任务共同完成信号量集的逻辑运算及控制

为了方便说明，我们将 OSFlagFlags 定义为 8 位，但是 UCOSII 支持 8 位/16 位/32 位定义，这个通过修改 OS_FLAGS 的类型来确定（UCOSII 默认设置 OS_FLAGS 为 16 位）。

上图清楚表达了信号量集各成员的关系：OSFlagFlags 位信号量表，通过发送信号量集任务设置；OSFlagNodeFlags 为信号滤波器，由请求信号量集的任务设置，用于选择性的挑选 OSFlagFlags 中的部分（或全部）位作为有效信号；OSFlagNodeWaitType 定义有效信号的逻辑运算关系，也是由请求信号量集的任务设置，用于选择有效的组合方式（0/1？与/或？）。

举个简单的例子，假设请求信号量集的任务设置 OSFlagNodeFlags 的值为 0x0F，同时设置 OSFlagNodeWaitType 的值为 WAIT_SET_ANY，那么只要 OSFlagFlags 的低四位的任何一位为 1，请求信号量集的任务将得到有效的请求，从而执行相关操作；如果第四位都为 0，那么请求信号量集的任务将得到无效的请求。

信号量集相关的主要操作有：创建一个信号量集函数 OSFlagCreate，请求一个信号量集函数 OSFlagPend，向信号量集发送信号函数 OSFlagPost。后面再对这几个函数进行讲解。

信号量集就介绍到这里，更详细的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》第六章。

65.1.3 软件定时器

UCOSII 从 V2.83 版本以后，加入了软件定时器，这使得 UCOSII 的功能更加完善，在其上的应用程序开发与移植也更加方便。在实时操作系统中一个好的软件定时器实现要求有较高的精度、较小的处理器开销，且占用较少的储存器资源。

通过前面的学习，我们知道 UCOSII 通过 OSTimTick 函数对时钟节拍进行加 1 操作，同时遍历任务控制块，以判断任务延时是否到时。软件定时器同样由 OSTimTick 提供时钟，但是软件定时器的时钟还受 OS_TMR_CFG_TICKS_PER_SEC 设置的控制，也就是在 UCOSII 的时钟节拍上面在做了一次“分频”，软件定时器的最快时钟节拍就等于 UCOSII 的系统时钟节拍。这也决定了软件定时器的精度。

软件定时器定义了一个单独的计数器 OSTmrTime，用于软件定时器的计时，UCOSII 并不在 OSTimTick 中进行软件定时器的到时判断与处理，而是创建了一个高于应用程序中所有其他任务优先级的定时器管理任务 OSTmr_Task，在这个任务中进行定时器的到时判断和处理。时钟节拍函数通过信号量给这个高优先级任务发信号。这种方法缩短了中断服务程序的执行时间，但也使得定时器到时处理函数的响应收到中断推出时恢复现场和任务切换的影响。

UCOSII 中软件定时器的实现方法是，将定时器按定时事件分组，使得每次时钟节拍到来时只对部分定时器及逆行比较操作，缩短了每次处理的时间。但这就需要动态地维护一个定时器组。定时器组的维护只是在每次定时器到时的时候才发生，而且定时器从组中移除和再插入操作不需要排序。这是一种比较高效的算法，减少了维护所需的操作时间。

UCOSII 软件定时器实现了 3 类链表的维护：

```
OS_EXT OS_TMR OSTmrTbl[OS_TMR_CFG_MAX]; /* 定时器控制块数组 OS_EXT */
OS_EXT OS_TMR *OSTmrFreeList; /* 空闲定时器控制块链表指针 */
OS_EXT OS_TMR_WHEEL OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]; /* 定时器轮 */
```

其中 OS_TMR 为定时器控制块，定时器控制块是软件定时器管理的基本单元，包含软件定时器的名称、定时时间、在链表中的位置、使用状态、使用方式，以及到时回调函数及其参数等基本信息。

OSTmrTbl[OS_TMR_CFG_MAX]: 以数组的形式静态分配定时器控制块所需的 RAM 空间，并存储所有已建立的定时器控制块，OS_TMR_CFG_MAX 为最大软件定时器的个数。

OSTmrFreeList: 为空闲定时器控制块链表头指针。空闲态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针分别指向空闲控制块的前一个和后一个，组织了空闲控制块双向链表。建立定时器时，从这个链表中搜索空闲定时器控制块。

OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]: 该数组的每个元素都是已开启定时器的一个分组，元素中记录了指向该分组中第一个定时器控制块的指针，以及定时器控制块的个数。运行态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPre 两个指针同样也组织了所在分组中定时器控制块的双向链表。软件定时器管理所需的数据结构示意图如图 65.1.3.1 所示：

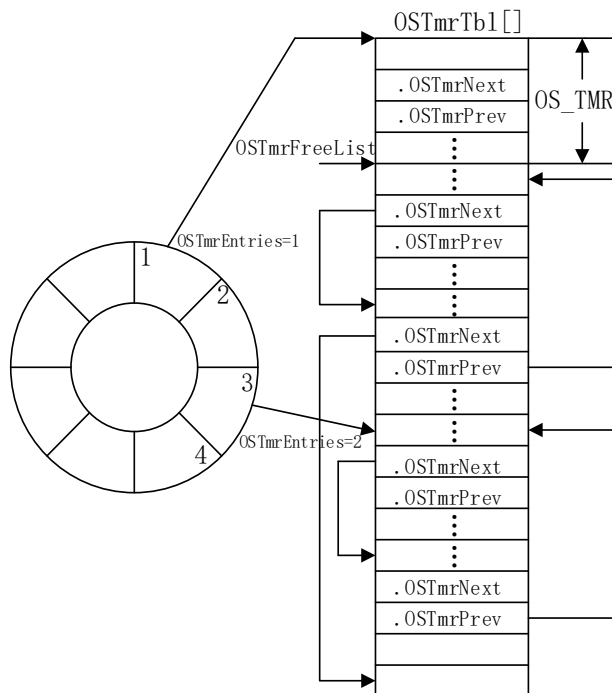


图 65.1.3.1 软件定时器管理所需的数据结构示意图

OS_TMR_CFG_WHEEL_SIZE 定义了 OSTmrWheelTbl 的大小，同时这个值也是定时器分组的依据。按照定时器到时值与 OS_TMR_CFG_WHEEL_SIZE 相除的余数进行分组：不同余数的定时器放在不同分组中；相同余数的定时器处在同一组中，由双向链表连接。这样，在余数值为 $0 \sim \text{OS_TMR_CFG_WHEEL_SIZE} - 1$ 的不同定时器控制块，正好分别对应了数组元素 OSTmrWheelTbl[0]~OSTmrWheelTbl[OS_TMR_CFGWHEEL_SIZE - 1] 的不同分组。每次时钟节拍到来时，时钟数 OSTmrTime 值加 1，然后也进行求余操作，只有余数相同的那组定时器才有可能到时，所以只对该组定时器进行判断。这种方法比循环判断所有定时器更高效。随着时钟数的累加，处理的分组也由 $0 \sim \text{OS_TMR_CFG_WHEEL_SIZE} - 1$ 循环。我们推荐这里 OS_TMR_CFG_WHEEL_SIZE 的取值为 2 的 N 次方，采用移位操作计算余数，缩短处理时间。

信号量唤醒定时器管理任务，计算出当前索要处理的分组后，程序遍历该分组中所有控制块，将当前 OSTmrTime 值与定时器控制块中的到时值（OSTmrMatch）相比较。若相等（即到时），则调用该定时器到时回调函数；若不相等，则判断该组中下一个定时器控制块。如此操作，直到该分组链表的结尾。软件定时器管理任务的流程如图 65.1.3.2 所示：

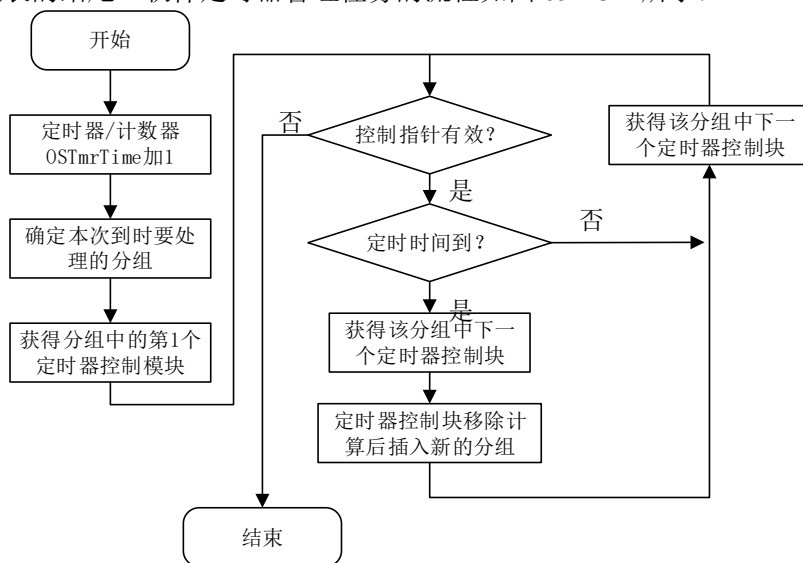


图 65.1.3.2 软件定时器管理任务流程

当运行完软件定时器的到时处理函数之后，需要进行该定时器控制块在链表中的移除和再插入操作。插入前需要重新计算定时器下次到时所处的分组。计算公式如下：

定时器下次到时的 OSTmrTime 值(OSTmrMatch) = 定时器定时值 + 当前 OSTmrTime 值
新分组 = 定时器下次到时的 OSTmrTime 值(OSTmrMatch) % OS_TMR_CFG_WHEEL_SIZE

软件定时器相关的主要操作有：创建软件定时器函数 OSTmrCreate，开启软件定时器函数 OSTmrStart，停止软件定时器函数 OSTmrStop。

65.2 硬件设计

1. 例程功能

在 UCOSII 里面创建 7 个任务：开始任务、LED 任务、触摸屏任务、队列消息显示任务、信号量集任务、按键扫描任务和主任务。开始任务用于创建邮箱、消息队列、信号量集以及其他任务，之后任务挂起；触摸屏任务用于在屏幕上画图，可以用于测试 CPU 使用率；队列消息显示任务请求消息队列，在得到消息后显示收到的消息数据；信号量集任务用于测试信号量集，采用 OS_FLAG_WAIT_SET_ANY 的方法，任何按键按下，该任务都会控制 DS1 闪以下；按键扫描任务用于按键扫描，优先级最高，将得到的键值通过消息邮箱发送出去；主任务创建 3 个软件定时器（定时器 1，100ms 溢出一次，显示 CPU 和内存使用率；定时器 2，200ms 溢出一次，在固定区域不停的显示不同颜色；定时器 3，100ms 溢出一次，用于自动发送消息到消息队列）。KEY0 控制软件定时器 3 的开关，从而控制消息队列的发送；KEY1 控制软件定时器 2 的开关，同时清除 LCD 触摸屏区域数据；WK_UP 按键用于触摸屏校准。

2. 硬件资源

- 1) LED 灯
 - LED0 - PB5
 - LED1 - PE5
- 2) 独立按键
 - KEY0 - PE4
 - KEY1 - PE3
 - WK_UP - PA0
- 3) 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

用到的硬件用到 LED 灯和按键。电路在开发板上已经连接好了，所以在硬件上不需要动任何东西，直接下载代码就可以测试使用。其连接原理图如图 65.2.1 所示：

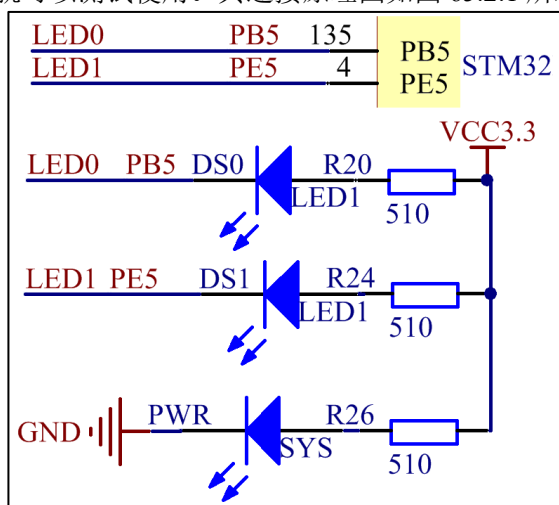


图 65.2.1 LED 与 STM32F103 连接原理图

65.3 程序设计

65.3.1 UCOSII 程序流程图

消息队列函数

在这里对本实验用到的 UCOSII 消息队列函数进行介绍，相关代码存放在 os_q.c 中。

1. OSQCreate 函数

创建消息队列函数，其声明如下：

```
OS_EVENT *OSQCreate (void **start, INT16U size)
```

- **函数描述：**
用于创建一个消息队列
- **函数形参：**
start: 存放消息缓冲区指针数组的地址
size: 该数组的大小
- **函数返回值：**
返回一个指向消息队列控制块的指针。如果没有空闲的控制块，返回空指针。

2. OSQPend 函数

请求消息队列函数，其声明如下：

```
void *OSQPend (OS_EVENT *pevent, INT32U timeout, INT8U *perr)
```

- **函数描述：**
用于向消息队列请求消息。
- **函数形参：**
pevent: 要访问的消息队列事件控制块的指针 timeout: 等待时限
timeout: 等待时限
perr: 错误信息
OS_ERR_NONE: 调用成功，消息被正确地接受
OS_ERR_TIMEOUT : 消息没有在指定数目的时钟周期内被接收
OS_ERR_PEND_ABOUT: 取消队列等待
OS_ERR_EVENT_TYPE: 没有传递一个指向队列的指针
OS_ERR_PEVENT_NULL: pevent 是一个空指针
OS_ERR_PEND_ISR : 从中断调用该函数，导致任务挂起
OS_ERR_PEND_LOCKED: 调度器上锁了
- **函数返回值：**
无
- **注意事项：**
如果队列中有消息，该消息传递给任务，并从队列中清除该消息；如果队列中没有消息，则调用该函数的任务被挂起等待，直到有消息或者等待超时。当有多个任务请求到同一消息队列时，UCOSII 进行任务调度，当前任务和所有请求该消息队列的任务中最高优先级的任务得到运行。

3. OSQPost 函数

向消息队列发送消息函数，其声明如下：

```
INT8U OSQPost (OS_EVENT *pevent, void *pmsg)
```

- **函数描述：**
用于向消息队列发送消息
- **函数形参：**
pevent: 消息队列指针 pmsg: 消息指针
- **函数返回值：**
OS_ERR_NONE: 函数调用成功，成功发送消息到消息队列
OS_ERR_Q_FULL: 队列已满
OS_ERR_EVENT_TYPE: pevent 不是指向消息队列的指针

OS_ERR_PEVENT_NULL: pevent 是一个空指针

信号量集函数

在这里对本实验用到的 UCOSII 信号量集函数进行介绍，相关代码存放在 os_flag.c 中。

1. OSFlagCreate 函数

创建信号量集函数，其声明如下：

```
OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags, INT8U *perr)
```

- **函数描述：**
用于创建一个信号量集
- **函数形参：**
flags: 信号的初始值
perr: 错误信息
OS_ERR_NONE: 函数调用成功
OS_ERR_CREATE_ISR: 在中断中创建信号量集
OS_ERR_FLAG_GRP_DEPLETED: 没有更多的事件标志组
- **函数返回值：**
这个信号量集的标志组的指针，应用程序可以用这个指针对信号量集进行相对应的操作。

2. OSFlagPend 函数

请求信号量集函数，其声明如下：

```
OS_FLAGS OSFlagPend (OS_FLAG_GRP *pgrp,  
                    OS_FLAGS flags,  
                    INT8U wait_type,  
                    INT32U timeout,  
                    INT8U *perr)
```

- **函数描述：**
用于请求一个信号量集
- **函数形参：**
pgrp: 消息邮箱的指针
flags : 消息指针
wait_type: 逻辑运算类型
OS_FLAG_WAIT_CLR_ALL: 等待“mask”中的所有位被清除
OS_FLAG_WAIT_SET_ALL: 等待“mask”中的所有位被设置
OS_FLAG_WAIT_CLR_ANY: 等待“mask”中的任何位被清除
OS_FLAG_WAIT_SET_ANY: 等待“mask”中的任何位被设置
timeout: 等待时延
perr: 错误信息
OS_ERR_NONE: 指定比特数已经被设置“超时”
OS_ERR_PEND_ISR: 从中断挂起
OS_ERR_FLAG_INVALID_PGRP: pgrp 是一个空指针
OS_ERR_TIMEOUT: 指定比特数未设置“超时”
OS_ERR_PEND_ABORT: 取消标志等待
OS_ERR_FLAG_WAIT_TYPE: 没有指定正确的“wait_type”参数
- **函数返回值：**
返回标志组成员 OSFlagFlags 的值

3. OSFlagPost 函数

向信号量集发送信号函数，其声明如下：

```
OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp,  
                   OS_FLAGS flags,  
                   INT8U opt,  
                   INT8U *perr)
```

- **函数描述：**
用于向信号量集发送信号

- **函数形参：**
pgrp: 信号量集指针
flags: 选择所要发送的信号
opt: 信号有效的选项
perr: 错误信息
OS_ERR_NONE: 函数调用成功
OS_ERR_FLAG_INVALID_PGRP: pgrp 是一个空指针
OS_ERR_EVENT_TYPE: 没有指向时间标志组
OS_ERR_FLAG_INVALID_OPT: 使用无效选项
- **函数返回值：**
返回标志组成员 OSFlagFlags 的值
- **注意事项：**
所谓任务向信号量集发信号，就是对信号量集标志组中的信号进行置“1”(置位)或者置“0”(复位)的操作。至于对信号量集标志组中的哪些信号进行操作，用函数中的参数 flags 来指定；对指定的信号是置“1”还是置“0”，用函数中的参数 opt 来指定(opt = OS_FLAG_SET 为置“1”操作；opt = OS_FLAG_CLR 为置“0”操作)。

软件定时器函数

在这里对本实验用到的 UCOSII 软件定时器函数进行介绍，相关代码存放在 os_tmr.c 中。

1. OSTmrCreate 函数

创建软件定时器函数，其声明如下：

```
OS_TMR *OSTmrCreate (INT32U      dly,
                    INT32U      period,
                    INT8U       opt,
                    OS_TMR_CALLBACK callback,
                    void        *callback_arg,
                    INT8U       *pname,
                    INT8U       *perr)
```

- **函数描述：**
用于创建软件定时器
- **函数形参：**
dly: 用于初始化定时时间
period: 软件定时器的周期溢出时间
opt: 设置软件定时器工作模式
OS_TMR_OPT_ONE_SHOT: 计时器只倒计时一次
OS_TMR_OPT_PERIODIC: 定时器倒计时，然后重新加载
callback_arg: 回调函数的参数
pname: 软件定时器的名字
perr: 错误信息
OS_ERR_NONE: 函数调用成功
OS_ERR_TMR_INVALID_DLY: 执行无效延时操作
OS_ERR_TMR_INVALID_PERIOD: 执行操作时输入了不合法的时间
OS_ERR_TMR_INVALID_OPT: 使用无效选项
OS_ERR_TMR_ISR: 在中断中创建
OS_ERR_TMR_NON_AVAIL: 定时器池中没空闲的定时器
- **函数返回值：**
返回定时器控制块
- **注意事项：**
软件定时器的回调函数有固定格式，我们必须按照这个格式编写，软件定时器的回调函数格式为：void (*OS_TMR_CALLBACK)(void *ptmr, void *parg)。其中，函数名我们可以自己随意设置，而 ptmr 这个参数，软件定时器用来传递当前定时器的控制块指针，所以我们一般设置

其类型为 OS_TMR*类型，第二个参数 parg 为回调函数的参数，这个就可以根据自己需要设置了，你也可以不用，但是必须有这个参数。

2. OSTmrStart 函数

开启软件定时器函数，其声明如下：

```
BOOLEAN OSTmrStart (OS_TMR *ptmr, INT8U *perr)
```

- **函数描述：**

用于开启某个软件定时器

- **函数形参：**

ptmr: 要开启的软件定时器指针

perr: 错误信息

OS_ERR_NONE: 函数调用成功

OS_ERR_TMR_INVALID: 定时器无效

OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向 OS_TMR

OS_ERR_TMR_ISR: 从中断中创建

OS_ERR_TMR_INACTIVE: 定时器没有被创建

OS_ERR_TMR_INVALID_STATE: 定时器处于无效状态

- **函数返回值：**

返回布尔值 OS_TRUE: 软件定时器开启 OS_FALSE: 错误发生

3. OSTmrStop 函数

停止软件定时器函数，其声明如下：

```
BOOLEAN OSTmrStop (OS_TMR *ptmr,
                   INT8U opt,
                   void *callback_arg,
                   INT8U *perr)
```

- **函数描述：**

用于停止某个软件定时器

- **函数形参：**

ptmr: 要停止的软件定时器指针

opt: 停止选项

OS_TMR_OPT_NONE, 直接停止，不做任何其他处理

OS_TMR_OPT_CALLBACK, 停止，用初始化的参数执行一次回调函数

OS_TMR_OPT_CALLBACK_ARG, 停止，用新的参数执行一次回调函数

callback_arg: 新的回调函数参数

perr: 错误信息

OS_ERR_NONE: 函数调用成功

OS_ERR_TMR_INVALID: 定时器无效

OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向 OS_TMR

OS_ERR_TMR_ISR: 从中断中创建

OS_ERR_TMR_INACTIVE: 定时器没有被创建

OS_ERR_TMR_INVALID_OPT: 操作时使用无效选项

OS_ERR_TMR_STOPPED: 定时器已经停止

OS_ERR_TMR_INVALID_STATE: 定时器处于无效状态

OS_ERR_TMR_NO_CALLBACK: 定时器没有定义回调函数

- **函数返回值：**

返回布尔值 OS_TRUE: 软件定时器停止 OS_FALSE: 软件定时器停止失败

65.3.2 程序流程图

消息队列我们在多任务间进行，所以我们需要定义多个任务，在不同任务中传递消息。

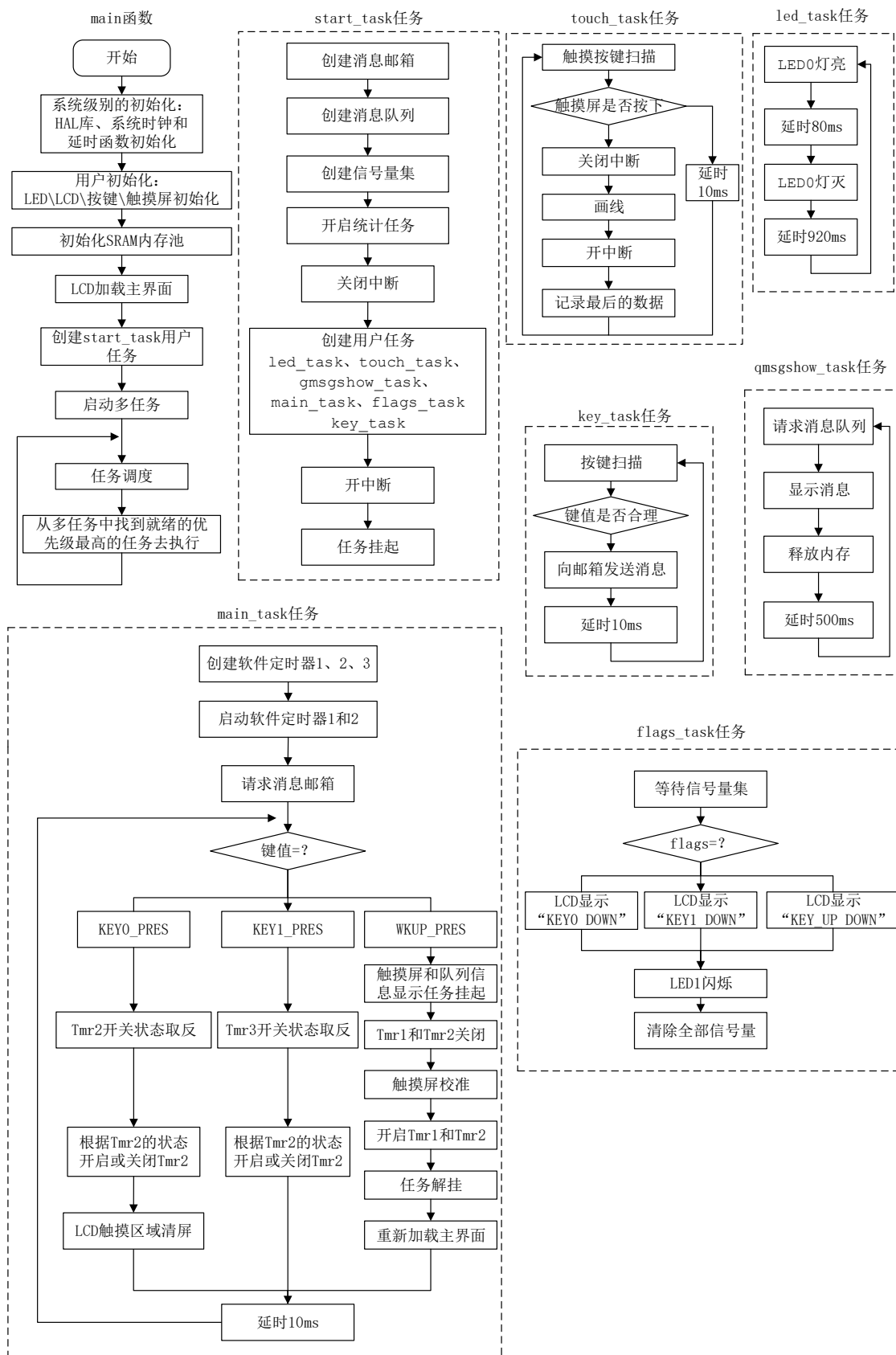


图 65.3.2.1 UCOSII 消息队列、信号量集和软件定时器实验

程序我们按流程图的设计来实现本节的功能代码。我们通过 `start_task` 创建其它任务，包括：LED 任务，用于定时闪烁 LED 灯；

触摸任务，用于实现触摸画线功能；
 按键扫描函数，根据按键发送不同的消息；
 qmsgshow_task，请求并显示收到的队列消息；
 flag_task 用于请求和显示按键信号量集，获取完的信号于一段时间后从队列中清除；
 main_task 用于创建定时器任务，并根据获取的队列消息得到按键值并投递给 flag_task 进行显示，根据按键值设置软件定时器 1，2，3 的状态，控制部分任务运行或挂起，重绘界面。

65.3.3 程序解析

1. main.c 代码

在 main.c 文件下，只初始化一些外设，然后调用 ucossii 的例程入口函数 uc_os2_demo()，如下代码所示：

```

#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"
#include "./BSP/LED/led.h"
#include "./BSP/KEY/key.h"
#include "./BSP/LCD/lcd.h"
#include "./BSP/BEEP/beep.h"
#include "./BSP/TOUCH/touch.h"
#include "./BSP/SRAM/sram.h"
#include "./MALLOC/malloc.h"
#include "uc-os2_demo.h"

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟, 72M */
    delay_init(72); /* 初始化延时函数 */
    usart_init(115200); /* 串口初始化为 115200 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    beep_init(); /* 初始化蜂鸣器 */
    sram_init(); /* SRAM 初始化 */
    tp_dev.init(); /* 触摸屏初始化 */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMEX); /* 初始化外部 SRAM 内存池 */
    uc_os2_demo(); /* 运行 uC/OS-II 例程 */
}
  
```

可以看到，在 main.c 文件中只包含了一个 main() 函数，main() 函数主要就是完成了一些外设的初始化，如串口、LED、LCD 等，并在最后调用了函数 uc_os2_demo()。

下面看一下 uc-os2_demo.c 的代码：

```

/* UCOSII 任务设置 */

/* START 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define START_TASK_PRIO 10 /* 开始任务的优先级设置为最低 */
#define START_STK_SIZE 128 /* 堆栈大小 */

OS_STK START_TASK_STK[START_STK_SIZE]; /* 任务堆栈 */
void start_task(void *pdata); /* 任务函数 */

/* 触摸屏任务 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define TOUCH_TASK_PRIO 7 /* 优先级设置 (越小优先级越高) */
  
```



```

#define TOUCH_STK_SIZE                128          /* 堆栈大小 */

OS_STK TOUCH_TASK_STK[TOUCH_STK_SIZE];           /* 任务堆栈 */
void touch_task(void *pdata);                     /* 任务函数 */

/* LED 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define LED_TASK_PRIO                  6           /* 优先级设置(越小优先级越高) */
#define LED_STK_SIZE                   128         /* 堆栈大小 */

OS_STK LED_TASK_STK[LED_STK_SIZE];                /* 任务堆栈 */
void led_task(void *pdata);                        /* 任务函数 */

/* 队列消息显示 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define QMSGSHOW_TASK_PRIO             5           /* 优先级设置(越小优先级越高) */
#define QMSGSHOW_STK_SIZE              128         /* 堆栈大小 */

OS_STK QMSGSHOW_TASK_STK[QMSGSHOW_STK_SIZE];      /* 任务堆栈 */
void qmsgshow_task(void *pdata);                  /* 任务函数 */

/* 主 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define MAIN_TASK_PRIO                 4           /* 优先级设置(越小优先级越高) */
#define MAIN_STK_SIZE                   512        /* 堆栈大小 */

OS_STK MAIN_TASK_STK[MAIN_STK_SIZE];              /* 任务堆栈 */
void main_task(void *pdata);                      /* 任务函数 */

/* 信号量集 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define FLAGS_TASK_PRIO                 3           /* 优先级设置(越小优先级越高) */
#define FLAGS_STK_SIZE                  512        /* 堆栈大小 */

OS_STK FLAGS_TASK_STK[FLAGS_STK_SIZE];            /* 任务堆栈 */
void flags_task(void *pdata);                     /* 任务函数 */

/* 按键扫描 任务 配置
 * 包括：任务优先级 堆栈大小 等
 */
#define KEY_TASK_PRIO                   2           /* 优先级设置(越小优先级越高) */
#define KEY_STK_SIZE                    512        /* 堆栈大小 */

OS_STK KEY_TASK_STK[KEY_STK_SIZE];                /* 任务堆栈 */
void key_task(void *pdata);                       /* 任务函数 */

/*****

OS_EVENT  *msg_key;                               /* 按键邮箱事件块 */
OS_EVENT  *q_msg;                                  /* 消息队列 */
OS_TMR     *tmr1;                                  /* 软件定时器 1 */
OS_TMR     *tmr2;                                  /* 软件定时器 2 */
OS_TMR     *tmr3;                                  /* 软件定时器 3 */
OS_FLAG_GRP *flags_key;                           /* 按键信号量集 */

```

```

/* 消息队列存储地址,最大支持 256 个消息 */
void *MsgGrp[256];

/* 这些函数在 main 函数后面实现 */
void tmr1_callback(OS_TMR *ptmr, void *p_arg);
void tmr2_callback(OS_TMR *ptmr, void *p_arg);
void tmr3_callback(OS_TMR *ptmr, void *p_arg);

void uc_os_load_main_ui(void);
void lcd_draw_bline(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint8_t
size, uint16_t color);

/**
 * @brief      uC/OS-II 例程入口函数
 * @param      无
 * @retval     无
 */
void uc_os2_demo(void)
{
    uc_os_load_main_ui();                /* 加载主界面 */
    OSInit();                            /* UCOS 初始化 */
    OSTaskCreateExt((void*)(void *) )start_task, /* 任务函数 */
        /* 传递给任务函数的参数 */
        (void *) 0,
        /* 任务堆栈栈顶 */
        (OS_STK * )&START_TASK_STK[START_STK_SIZE - 1],
        (INT8U )START_TASK_PRIO,        /* 任务优先级 */
        /* 任务 ID, 这里设置为和优先级一样 */
        (INT16U )START_TASK_PRIO,
        (OS_STK * )&START_TASK_STK[0], /* 任务堆栈栈底 */
        (INT32U )START_STK_SIZE,        /* 任务堆栈大小 */
        (void * )0,                      /* 用户补充的存储区 */
        /* 任务选项,为了保险起见,所有任务都保存浮点寄存器的值 */
        (INT16U ) OS_TASK_OPT_STK_CHK |
            OS_TASK_OPT_STK_CLR |
            OS_TASK_OPT_SAVE_FP);

    OSStart();                          /* 开始任务 */

    for (;;)
    {
        /* 不会进入这里 */
    }
}

/**
 * @brief      开始任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void start_task(void *pdata)
{
    uint8_t err;
    OS_CPU_SR cpu_sr = 0;
    CPU_INT32U cnts;
    /* 根据配置的节拍频率配置 SysTick */
    cnts = (CPU_INT32U)(HAL_RCC_GetSysClockFreq() / OS_TICKS_PER_SEC);
    OS_CPU_SysTickInit(cnts);

    msg_key = OSMsgboxCreate((void *)0); /* 创建消息邮箱 */
    q_msg = OSQCreate(&MsgGrp[0], 256); /* 创建消息队列 */
    flags_key = OSFlagCreate(0, &err); /* 创建信号量集 */
    OSStatInit();                      /* 开启统计任务 */
}

```

```

OS_ENTER_CRITICAL(); /* 进入临界区(关闭中断) */

/* LED 任务 */
OSTaskCreateExt((void*)(void *) )led_task,
    (void *) 0,
    (OS_STK *) &LED_TASK_STK[LED_STK_SIZE - 1],
    (INT8U) LED_TASK_PRIO,
    (INT16U) LED_TASK_PRIO,
    (OS_STK *) &LED_TASK_STK[0],
    (INT32U) LED_STK_SIZE,
    (void *) 0,
    (INT16U) OS_TASK_OPT_STK_CHK |
              OS_TASK_OPT_STK_CLR |
              OS_TASK_OPT_SAVE_FP);

/* 触摸任务 */
OSTaskCreateExt((void*)(void *) )touch_task,
    (void *) 0,
    (OS_STK *) &TOUCH_TASK_STK[TOUCH_STK_SIZE - 1],
    (INT8U) TOUCH_TASK_PRIO,
    (INT16U) TOUCH_TASK_PRIO,
    (OS_STK *) &TOUCH_TASK_STK[0],
    (INT32U) TOUCH_STK_SIZE,
    (void *) 0,
    (INT16U) OS_TASK_OPT_STK_CHK |
              OS_TASK_OPT_STK_CLR |
              OS_TASK_OPT_SAVE_FP);

/* 消息队列显示任务 */
OSTaskCreateExt((void*)(void *) )qmsgshow_task,
    (void *) 0,
    (OS_STK *) &QMSGSHOW_TASK_STK[QMSGSHOW_STK_SIZE - 1],
    (INT8U) QMSGSHOW_TASK_PRIO,
    (INT16U) QMSGSHOW_TASK_PRIO,
    (OS_STK *) &QMSGSHOW_TASK_STK[0],
    (INT32U) QMSGSHOW_STK_SIZE,
    (void *) 0,
    (INT16U) OS_TASK_OPT_STK_CHK |
              OS_TASK_OPT_STK_CLR |
              OS_TASK_OPT_SAVE_FP);

/* 主任务 */
OSTaskCreateExt((void*)(void *) )main_task,
    (void *) 0,
    (OS_STK *) &MAIN_TASK_STK[MAIN_STK_SIZE - 1],
    (INT8U) MAIN_TASK_PRIO,
    (INT16U) MAIN_TASK_PRIO,
    (OS_STK *) &MAIN_TASK_STK[0],
    (INT32U) MAIN_STK_SIZE,
    (void *) 0,
    (INT16U) OS_TASK_OPT_STK_CHK |
              OS_TASK_OPT_STK_CLR |
              OS_TASK_OPT_SAVE_FP);

/* 信号量集任务 */
OSTaskCreateExt((void*)(void *) )flags_task,
    (void *) 0,
    (OS_STK *) &FLAGS_TASK_STK[FLAGS_STK_SIZE - 1],
    (INT8U) FLAGS_TASK_PRIO,
    (INT16U) FLAGS_TASK_PRIO,
    (OS_STK *) &FLAGS_TASK_STK[0],
    (INT32U) FLAGS_STK_SIZE,
    (void *) 0,
    (INT16U) OS_TASK_OPT_STK_CHK |
              OS_TASK_OPT_STK_CLR |
              OS_TASK_OPT_SAVE_FP);

/* 按键任务 */
OSTaskCreateExt((void*)(void *) )key_task,
    (void *) 0,

```

```

        (OS_STK *      ) &KEY_TASK_STK[KEY_STK_SIZE - 1],
        (INT8U        ) KEY_TASK_PRIO,
        (INT16U       ) KEY_TASK_PRIO,
        (OS_STK *      ) &KEY_TASK_STK[0],
        (INT32U       ) KEY_STK_SIZE,
        (void *        ) 0,
        (INT16U       ) OS_TASK_OPT_STK_CHK |
                        OS_TASK_OPT_STK_CLR |
                        OS_TASK_OPT_SAVE_FP);
    OS_EXIT_CRITICAL(); /* 退出临界区(开中断) */
    OSTaskSuspend(START_TASK_PRIO); /* 挂起开始任务 */
}

/**
 * @brief      LED0 任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void led_task(void *pdata)
{
    uint8_t t;

    while (1)
    {
        t++;
        OSTimeDly(10);

        if (t == 8)
        {
            LED0(1); /* LED0 灭 */
        }

        if (t == 100)
        {
            t = 0;
            LED0(0); /* LED0 亮 */
        }
    }
}

/**
 * @brief      触摸屏任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void touch_task(void *pdata)
{
    uint32_t cpu_sr;
    uint16_t lastpos[2]; /* 最后一次的数据 */

    while (1)
    {
        tp_dev.scan(0);

        if (tp_dev.sta & TP_PRES_DOWN) /* 触摸屏被按下 */
        {
            if (tp_dev.x[0] < (130 - 1) &&
                tp_dev.y[0] < lcddev.height &&
                tp_dev.y[0] > (220 + 1))
            {
                if (lastpos[0] == 0xFFFF)
                {
                    lastpos[0] = tp_dev.x[0];
                    lastpos[1] = tp_dev.y[0];
                }
            }
        }
    }
}

```

```

        /* 进入临界段,防止其他任务,打断 LCD 操作,导致液晶乱序 */
        OS_ENTER_CRITICAL();
        lcd_draw_bline(lastpos[0],
                        lastpos[1],
                        tp_dev.x[0],
                        tp_dev.y[0],
                        2,
                        RED);          /* 画线 */
        OS_EXIT_CRITICAL();
        lastpos[0] = tp_dev.x[0];
        lastpos[1] = tp_dev.y[0];
    }
}
else
{
    lastpos[0] = 0xFFFF;
    OSTimeDly(10);                  /* 没有按键按下的时候 */
}
}
}

/**
 * @brief      队列消息显示任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void qmsgshow_task(void *pdata)
{
    char *p;
    uint8_t err;

    while (1)
    {
        printf("qmsgshow_task\r\n");
        p = OSQPend(q_msg, 0, &err);          /* 请求消息队列 */
        /* 显示消息 */
        lcd_show_string(5, 170, 240, 16, 16, p, RED);
        myfree(SRAMIN, p);
        OSTimeDly(500);
    }
}

/**
 * @brief      主任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void main_task(void *pdata)
{
    uint32_t key = 0;
    uint8_t err;
    uint8_t tmr2sta = 1;                /* 软件定时器 2 开关状态 */
    uint8_t tmr3sta = 0;                /* 软件定时器 3 开关状态 */
    uint8_t flagsclr = 0;               /* 信号量集显示清零倒计时 */
    tmr1 = OSTmrCreate(10, 10, OS_TMR_OPT_PERIODIC,
                      (OS_TMR_CALLBACK)tmr1_callback,
                      0,
                      (unsigned char*)"tmr1", &err);          /* 100ms 执行一次 */
    tmr2 = OSTmrCreate(10, 20, OS_TMR_OPT_PERIODIC,
                      (OS_TMR_CALLBACK)tmr2_callback,
                      0,
                      (unsigned char*)"tmr2", &err);          /* 200ms 执行一次 */
    tmr3 = OSTmrCreate(10, 10, OS_TMR_OPT_PERIODIC,
                      (OS_TMR_CALLBACK)tmr3_callback,

```

```

    0,
    (unsigned char*)"tmr3", &err);          /* 100ms 执行一次 */
OSTmrStart(tmr1, &err);                     /* 启动软件定时器 1 */
OSTmrStart(tmr2, &err);                     /* 启动软件定时器 2 */

while (1)
{
    key = (uint32_t)OSMboxPend(msg_key, 10, &err);

    if (key)
    {
        flagsclrt = 51;                      /* 500ms 后清除 */
        /* 设置对应的信号量为 1 */
        OSFlagPost(flags_key, 1 << (key - 1), OS_FLAG_SET, &err);
    }

    if (flagsclrt)                           /* 倒计时 */
    {
        flagsclrt--;

        /* 清除显示 */
        if (flagsclrt == 1)lcd_fill(140, 162, 239, 162 + 16, WHITE);
    }

    switch (key)
    {
        case KEY0_PRES:
        {
            /* 软件定时器 2 开关,并清屏 */
            tmr2sta = !tmr2sta;

            if (tmr2sta)
            {
                OSTmrStart(tmr2, &err);      /* 开启软件定时器 2 */
            }
            else
            {
                /* 关闭软件定时器 2 */
                OSTmrStop(tmr2, OS_TMR_OPT_NONE, 0, &err);
                /* 提示定时器 2 关闭了 */
                lcd_show_string(148, 262, 240, 16, 16, "TMR2 STOP", RED);
            }
            /* 顺便清屏 */
            lcd_fill(0, 221, 129, lcddev.height - 1, WHITE);
            break;
        }
        case KEY1_PRES:
        {
            /* 控制软件定时器 3 */
            tmr3sta = !tmr3sta;

            if (tmr3sta)
            {
                OSTmrStart(tmr3, &err);
            }
            else
            {
                /* 关闭软件定时器 3 */
                OSTmrStop(tmr3, OS_TMR_OPT_NONE, 0, &err);
            }
            break;
        }
        case WKUP_PRES:
        {
            /* 校准 */

```



```

        OSTaskSuspend(TOUCH_TASK_PRIO);          /* 挂起触摸屏任务 */
        OSTaskSuspend(QMSGSHOW_TASK_PRIO);        /* 挂起队列信息显示任务 */
        OSTmrStop(tmr1, OS_TMR_OPT_NONE, 0, &err); /* 关闭软件定时器 1 */

        if (tmr2sta)
        {
            /* 关闭软件定时器 2 */
            OSTmrStop(tmr2, OS_TMR_OPT_NONE, 0, &err);
        }

        if ((tp_dev.touchtype & 0X80) == 0)
        {
            tp_adjust();
        }
        /* 重新开启软件定时器 1 */
        OSTmrStart(tmr1, &err);

        if (tmr2sta)
        {
            OSTmrStart(tmr2, &err);                /* 重新开启软件定时器 2 */
        }

        OSTaskResume(TOUCH_TASK_PRIO);            /* 解挂 */
        OSTaskResume(QMSGSHOW_TASK_PRIO);          /* 解挂 */
        ucoss_load_main_ui();                       /* 重新加载主界面 */
        break;
    }
}
OSTimeDly(10);
}
}

/**
 * @brief      信号量集处理任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void flags_task(void *pdata)
{
    uint16_t flags;
    uint8_t err;

    while (1)
    {
        /* 等待信号量 */
        flags = OSFlagPend(flags_key, 0X0007, OS_FLAG_WAIT_SET_ANY, 0, &err);

        if (flags & 0X0001)
        {
            lcd_show_string(140, 162, 240, 16, 16, "KEY0 DOWN ", RED);
        }

        if (flags & 0X0002)
        {
            lcd_show_string(140, 162, 240, 16, 16, "KEY1 DOWN ", RED);
        }

        if (flags & 0X0004)
        {
            lcd_show_string(140, 162, 240, 16, 16, "KEY_UP DOWN ", RED);
        }

        BEEP(1);
        OSTimeDly(50);
        BEEP(0);
    }
}

```

```

        OSFlagPost(flags_key, 0X0007, OS_FLAG_CLR, &err); /* 全部信号量清零 */
    }
}

/**
 * @brief      按键扫描任务
 * @param      pdata : 传入参数(未用到)
 * @retval     无
 */
void key_task(void *pdata)
{
    uint32_t key;

    while (1)
    {
        key = key_scan(0);

        if (key)
        {
            OSMboxPost(msg_key, (void *)key); /* 发送消息 */
        }
        OSTimeDly(10);
    }
}

/**
 * @brief      软件定时器 1 的回调函数
 * @note       每 100ms 执行一次,用于显示 CPU 使用率和内存使用率
 * @param      ptmr : 软件定时器指针
 * @param      p_arg: 参数指针(未用到)
 * @retval     无
 */
void tmr1_callback(OS_TMR *ptmr, void *p_arg)
{
    static uint16_t cpuusage = 0;
    static uint8_t tcnt = 0;

    if (tcnt == 5)
    {
        /* 显示 CPU 使用率 */
        lcd_show_xnum(202, 10, cpuusage / 5, 3, 16, 0, BLUE);
        cpuusage = 0;
        tcnt = 0;
    }

    cpuusage += OSCPUUsage;
    tcnt++;
    /* 显示内存使用率 */
    lcd_show_xnum(202, 30, my_mem_perused(SRAMIN) / 10, 3, 16, 0, BLUE);
    /* 显示队列当前的大小 */
    lcd_show_xnum(202, 50, ((OS_Q *)
        (q_msg->OSEventPtr))->OSQEntries, 3, 16, 0X80, BLUE);
}

/**
 * @brief      软件定时器 2 的回调函数
 * @note       每 200ms 执行一次
 * @param      ptmr : 软件定时器指针
 * @param      p_arg: 参数指针(未用到)
 * @retval     无
 */
void tmr2_callback(OS_TMR *ptmr, void *p_arg)
{
    static uint8_t sta = 0;

```

```

switch (sta)
{
    case 0:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, RED);
        break;
    }
    case 1:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, GREEN);
        break;
    }
    case 2:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, BLUE);
        break;
    }
    case 3:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, MAGENTA);
        break;
    }
    case 4:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, CYAN);
        break;
    }
    case 5:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, YELLOW);
        break;
    }
    case 6:
    {
        lcd_fill(131, 221, lcddev.width - 1, lcddev.height - 1, BRRED);
        break;
    }
}

sta++;

if (sta > 6)
{
    sta = 0;
}
}

/**
 * @brief      软件定时器 3 的回调函数
 * @note       每 300ms 执行一次
 * @param      ptmr : 软件定时器指针
 * @param      p_arg: 参数指针 (未用到)
 * @retval     无
 */
void tmr3_callback(OS_TMR *ptmr, void *p_arg)
{
    uint8_t *p;
    uint8_t err;
    static uint8_t msg_cnt = 0;
    p = mymalloc(SRAMIN, 13);
    if (p)
    {
        sprintf((char *)p, "ALIENTEK %03d", msg_cnt);
    }
}

```

```

        msg_cnt++;
        err = OSQPost(q_msg, p); /* 发送队列 */

        if (err != OS_ERR_NONE) /* 发送失败 */
        {
            myfree(SRAMIN, p); /* 释放内存 */
            OSTmrStop(tmr3, OS_TMR_OPT_NONE, 0, &err); /* 关闭软件定时器 3 */
        }
    }
}

/**
 * @brief      加载主界面
 * @param      无
 * @retval     无
 */
void uclos_load_main_ui(void)
{
    lcd_clear(WHITE); /* 清屏 */
    lcd_show_string(10, 10, 200, 16, 16, "STM32", RED);
    lcd_show_string(10, 30, 200, 16, 16, "UCOSII TEST3", RED);
    lcd_show_string(10, 50, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(10, 75, 240, 16, 16, "KEY0:TMR2 SW & CLR KEY1:Q SW", RED);
    lcd_show_string(10, 95, 240, 16, 16, "KEY_UP:ADJUST", RED);
    lcd_draw_line(0, 70, lcddev.width - 1, 70, RED);
    lcd_draw_line(150, 0, 150, 70, RED);

    lcd_draw_line(0, 120, lcddev.width - 1, 120, RED);
    lcd_draw_line(0, 220, lcddev.width - 1, 220, RED);
    lcd_draw_line(130, 120, 130, lcddev.height - 1, RED);

    lcd_show_string(5, 125, 240, 16, 16, "QUEUE MSG", RED); /* 队列消息 */
    lcd_show_string(5, 150, 240, 16, 16, "Message:", RED);
    lcd_show_string(5 + 130, 125, 240, 16, 16, "FLAGS", RED); /* 信号量集 */
    lcd_show_string(5, 225, 240, 16, 16, "TOUCH", RED); /* 触摸屏 */
    lcd_show_string(5 + 130, 225, 240, 16, 16, "TMR2", RED); /* 队列消息 */

    lcd_show_string(170, 10, 200, 16, 16, "CPU:  %", BLUE);
    lcd_show_string(170, 30, 200, 16, 16, "MEM:  %", BLUE);
    lcd_show_string(170, 50, 200, 16, 16, " Q :000", BLUE);

    delay_ms(300);
}

/**
 * @brief      画粗线
 * @param      x1,y1: 起点坐标
 * @param      x2,y2: 终点坐标
 * @param      size : 线条粗细程度
 * @param      color: 线的颜色
 * @retval     无
 */
void lcd_draw_bline(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint8_t
size, uint16_t color)
{
    uint16_t t;
    int xerr = 0, yerr = 0, delta_x, delta_y, distance;
    int incx, incy, row, col;

    if (x1 < size || x2 < size || y1 < size || y2 < size)
    {
        return;
    }
}

```

```

delta_x = x2 - x1;                                /* 计算坐标增量 */
delta_y = y2 - y1;
row = x1;
col = y1;

if (delta_x > 0)
{
    incx = 1;                                       /* 设置单步方向 */
}
else if (delta_x == 0)
{
    incx = 0;                                       /* 垂直线 */
}
else
{
    incx = -1;
    delta_x = -delta_x;
}

if (delta_y > 0)
{
    incy = 1;
}
else if (delta_y == 0)
{
    incy = 0;                                       /* 水平线 */
}
else
{
    incy = -1;
    delta_y = -delta_y;
}

if ( delta_x > delta_y)
{
    distance = delta_x;                             /* 选取基本增量坐标轴 */
}
else
{
    distance = delta_y;
}

for (t = 0; t <= distance + 1; t++)                /* 画线输出 */
{
    lcd_fill_circle(row, col, size, color); /* 画点 */
    xerr += delta_x ;
    yerr += delta_y ;

    if (xerr > distance)
    {
        xerr -= distance;
        row += incx;
    }

    if (yerr > distance)
    {
        yerr -= distance;
        col += incy;
    }
}
}

```

上面就是对创建的 start_task、led_task、touch_task、qmsgshow_task、main_task、flags_task 和 key_task 等 7 个任务的参数进行配置，例如优先级、堆栈大小和任务函数的声明及定义。此外还有 3 个软件定时器及其回调函数。

软件定时器 tmr1、tmr2 和 tmr3，tmr1 用于显示 CPU 使用率和内存使用率，每 100ms 执行一次；tmr2 用于在 LCD 的右下角区域不停的显示各种颜色，每 200ms 执行一次；tmr3 用于定时器向队列发送消息（用到了对台内存申请），每 100ms 发送一次。

在本实验中，我们还是使用消息邮箱 msg_key 在按键任务和主任务之间传递键值数据，我们创建信号量集 flags_key，在主任务里面将按键键值通过信号量集传递给信号量集处理任务 flags_task，实现按键信息的显示以及 LED1 的提示性闪灯。

此外我们还创建了一个大小为 256 的消息队列 q_msg，通过软件定时器 tmr3 的回调函数向消息队列发送消息，然后在消息队列显示任务 qmsgshow_task 里面请求消息队列，并在 LCD 上面显示得到的消息。消息队列还用到了动态内存管理。

在主任务 main_task 里面，我们实现了前面介绍的功能：KEY0 控制软件定时器 3 的开关，间接控制消息队列的发送；KEY1 控制软件定时器 2 的开关，同时清除 LCD 触摸屏区域的数据；WK_UP 用于触摸屏校准，在校准的时候，要先挂起触摸屏任务、队列消息显示任务，并停止软件定时器 tmr1 和 tmr2，否则可能对校准时的 LCD 显示造成干扰；

其他任务做的事情在前面程序流程图里面已经有说明，这里就不多说了。

65.4 下载验证

将程序下载到开发板后，可以看到 LCD 显示界面如图 65.4.1 所示：

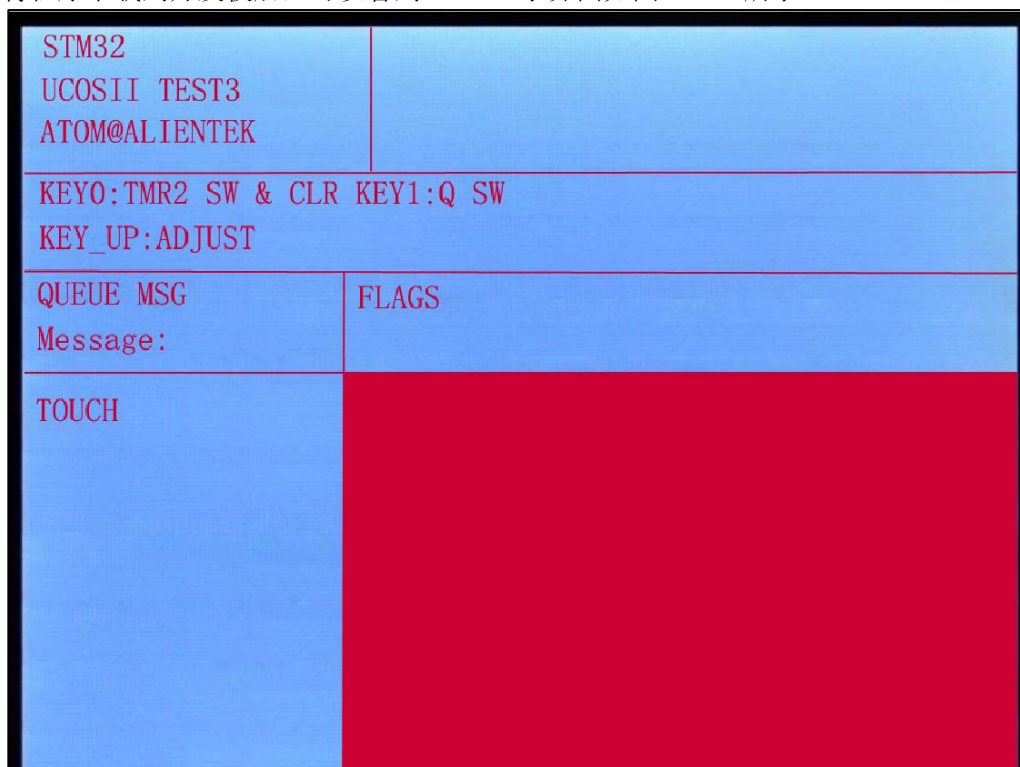


图 65.4.1 初始化界面

从上图可以看到，默认情况下，CPU 使用率为 25%，比上一章节多了一些，主要原因是软件定时器 2 不停的刷屏导致的。

通过按 KEY1，控制软件定时器 3 的开关，从而控制消息队列的发送，可以在 LCD 中看到 Q 和 MEM 的值慢慢变大，说明队列消息在增多，占用内存也随着消息增多而增大，在 QUEUE MSG 区，开始显示队列信息，再按一次 KEY1 停止软件定时器 3，此时可以看到 Q 和 MEM 逐渐减小。当 Q 值变为 0 的时候，QUEUE MSG 也停止显示（队列为空）。

通过按 KEY0 可以控制软件定时器 2 的开关，同时清除 LCD 触摸屏区域数据；通过按下 WK_UP 可以进入校准程序，对触摸屏进行校准。在 TOUCH 区域，可以输入手写内容。任何按键按下，DS1 都会闪一下，提示按键被按下，同时在 FLAGS 区域显示按键信息。

第六十六章 综合测试实验

为了方便大家使用和验证综合例程，本章内容是综合例程的使用介绍。目的是展示 STM32F1 的强大处理能力，并且可以测试开发板的大部分功能。本实验代码只提供寄存器版本，存放的路径是：**战舰 V4 STM32F103 开发板资料盘(A 盘)→4，程序源码→1，标准例程-寄存器版本→实验 54 综合测试实验**，代码非常多，这里不讲代码，只讲功能。

本章将分为如下 2 个小节：

66.1 综合测试实验简介

66.2 综合测试实验详解

66.1 综合测试实验简介

战舰 V4 是正点原子战舰 V3 开发板的升级版本，板载资源更合理，设计更精湛，软件功能也更强大，是新一代 STM32F103 开发板的旗舰型号。

开发板的硬件介绍请参考《战舰 V4 STM32F103 硬件参考手册_V1.0.pdf》。为了展示 STM32F1 系列强大的处理能力，综合例程实现了 19 个界面功能，分别是：电子图书、数码相框、音乐播放、TOM 猫、时钟、系统设置、FC 游戏机、记事本、运行器、手写画笔、照相机、录音机、USB 连接、网络通信、无线传书、计算器、拨号、应用中心和电压表。

电子图书：支持.txt/.c/.h/.lrc 等 4 种格式的文件阅读。

数码相框：支持.bmp/.jpeg/.jpb/.gif 等 4 种格式的图片文件播放。

音乐播放，支持.mp3/.wav/.wma/.ogg/.flac 等多种常见音频文件的播放。

TOM 猫，模仿手机游戏：TOM 猫，实现一个类似效果的游戏。

时钟：支持温度、时间、日期、星期的显示，同时具有指针式时钟显示。

系统设置：整个综合实验的设置。

FC 游戏机：支持绝大部分 NES 游戏(.nes)，支持 USB 手柄/键盘控制，带声音，超 Info NES。

记事本：可以实现文本(.txt/.c/.h/.lrc)记录编辑等功能，支持中英文输入，手写识别。

运行器：即 SRAM IAP 功能，支持.bin 文件的运行（文件大小+SRAM 大小≤55K）。

手写画笔：可以作画/对 bmp 图片进行编辑，支持画笔颜色/尺寸设置。

照相机：可以拍照(.bmp/.jpg 格式，需摄像头模块支持)，支持成像效果设置。

录音机：支持 wav 文件格式的录音（8KHz/16 位单声道录音），支持 AGC 设置。

USB 连接：支持和电脑连接读写 SD 卡/SPI FLASH 的内容。

网络通信：支持 10、100M 自适应，支持 DHCP，支持 UDP/TCP 测试。

无线传书：通过无线模块，实现两个开发板之间的无线通信。

计算器：一个科学计算器，支持各种运算，精度为 12 位，支持科学计数法表示。

拨号：支持拨打电话（需要 4G 模块支持）。

应用中心：可扩展 16 个应用程序，我们实现了其中 4 个（红外遥控&蜂鸣器&LED 测试&按键测试），其他预留。

电压表：用于测量 0~3.3V 的直流电压源。

以上，就是综合实验的 19 个功能简介，涉及到的内容包括：GUI（正点原子编写，非 ucGUI）、UCOSII、内存管理、图片解码、音频解码、文件系统、USB（从机）、IAP、网络通信（TCP/UDP）、NES 模拟器、手写识别、汉字输入等非常多的内容。下面，我们将详细介绍这 19 个功能。

66.2 综合测试实验详解

要测试战舰 V4 STM32F103 开发板综合测试实验的全部功能，大家得自备 1 个 TF 卡、1 根网线、1 个耳机（非必须）、1 个 DS18B20 温度传感器（非必须）、1 个 4G 模块、1 个正点原子 OV7725 带 FIFO 摄像头模块等。不过，就算没有这些东西，综合实验还是可以正常运行的，只是有些限制而已，比如：不能保存新建的记事本、不能保存新建的画图、不能使用录音机功能、不能使用摄像头功能、不能拨号等。除了这几个，其他功能基本都可以正常运行。

预备知识:

- 1, 系统支持: 正点原子 2.8 寸电阻屏、正点原子 3.5 寸电阻屏、正点原子 4.3 寸电容屏和正点原子 7 寸电容屏 (SSD1963 方案), 自动识别。
- 2, 系统针对不同分辨率的屏幕, 不同界面, 会采用不同的字体和图标, 以达到最佳效果。
- 3, 系统主界面, 对于 2.8 寸和 3.5 寸液晶模块, 将会有 2 页, 通过滑屏切换。每页 8 个图标+底部 3 个固定图标, 总共 19 个。对于其中 4.3 寸/7 寸液晶模块, 直接就是 1 页, 4.3 寸/7 寸屏不支持滑动。
- 4, 系统测试有可能需要比较大电流 (4.3 屏、7 寸屏、网络、喇叭) 供电, 强烈建议使用外部电源供电。

有了以上预备知识, 我们先来看看战舰 V4 STM32F103 开发板综合测试实验的启动界面, 启动界面如图 66.2.1 所示:

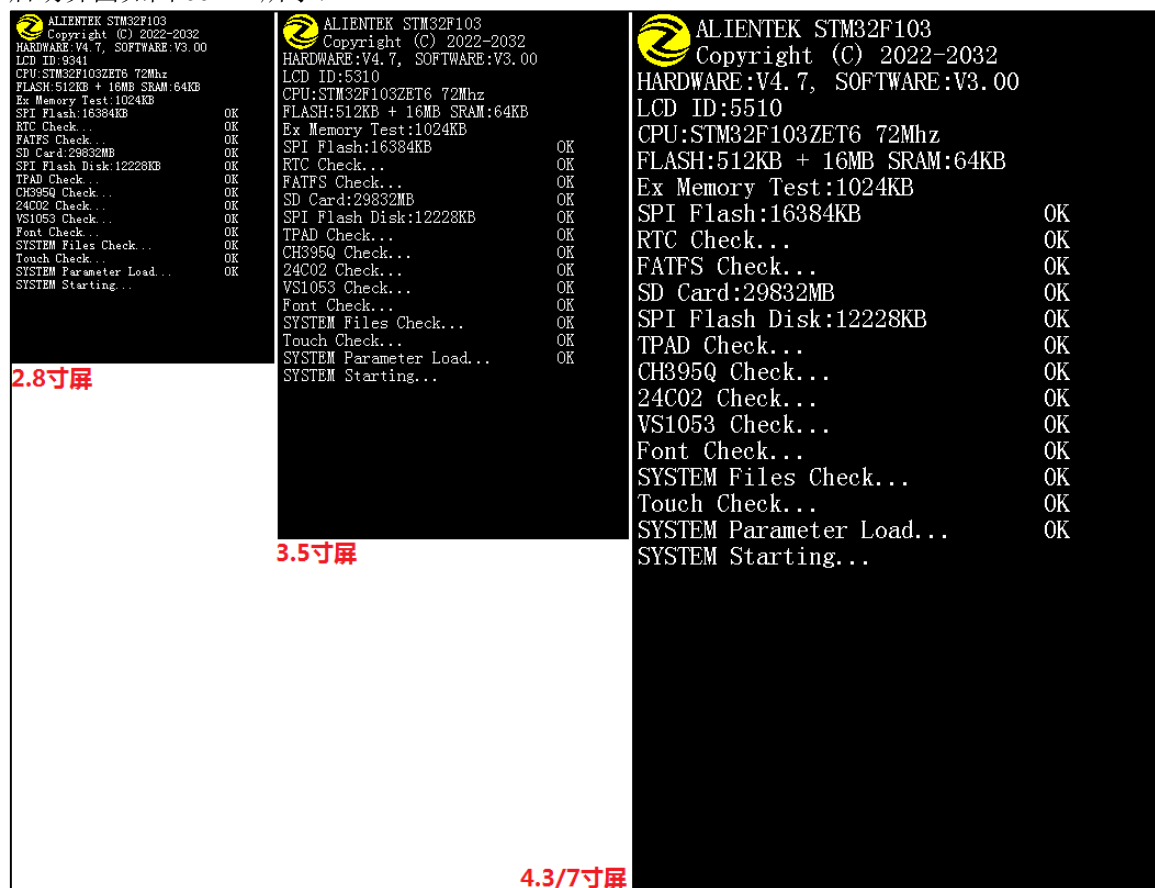


图 66.2.1 综合实验启动界面

注意: 综合实验支持屏幕截图 (通过 USMART 控制, 波特率为 115200), 本章所有图片均来自屏幕截图!

图 66.2.1 总共有 3 个截图拼成, 分别代表 2.8 寸、3.5 寸和 4.3 寸/7 寸屏模块, 显示内容都一样, 但是图标大小和文字大小各不相同。图片显示了综合实验的详细启动过程, 首先显示了版权信息, 软硬件版本, 接着显示了 LCD 驱动器的型号 (LCD ID), 然后显示 CPU 和内存信息 (16MB, 指的是 SPIFLASH 是 16MB), 之后显示 SPIFLASH 的大小, 接着开始初始化 RTC 和文件系统 (FATFS), 然后显示 SD 卡容量、FLASH Disk 容量 (注意 FLASH Disk 就是指 SPI FLASH, 因为我们划分了 12MB 空间给 FATFS 管理, 所以 FLASH Disk 的容量显示为 12228KB)。

接着, 就是硬件检测, 完了之后检测字库和系统文件, 再初始化触摸屏, 加载系统参数 (参数保存在 24C02 里面), 最后启动系统。在加载过程中, 任何一个地方出错, 都会显示相应的提示信息, 请在检查无误后, 按复位重启。

这里有几个注意的地方:

- ① 如果没插入 SD 卡, 其容量显示 0, 并提示 ERROR, 不过系统还是会继续启动, 因为

就算没有 SD 卡系统还是可以启动的（前提是 SPI FLASH（25Q128）里面的系统文件和字库文件都是正常的）。

- ② 系统文件和字库文件都是存在 SPI FLASH(25Q128)里面的，如这些文件被破坏了，在启动的时候，会提示 **Font Error / SYSTEM File Error**。解决方法如下：

准备一个 TF 卡，并拷贝 SYSTEM 文件夹（**注意：这个 SYSTEM 文件夹不是开发板例程里的 SYSTEM 文件夹，而是光盘根目录→SD 卡根目录文件→SYSTEM 文件夹**）到 TF 卡根目录，然后 TF 卡插入开发板，按复位重启，然后开发板会自动更新文件。

- ③ SPI Flash Disk 是从 SPI FLASH（25Q128）里面分割 12MB 空间出来实现的，强制将 4K 字节的扇区改为 512 字节使用，所以在写操作的时候擦除次数会明显提升（8 倍以上），因此，如非必要，请不要往 SPI Flash Disk 里面写文件。频繁的写操作，很容易将 SPI Flash Disk 写挂掉。
- ④ 在系统启动时，一直按着 KEY0 不放（加载到 Touch Check 时），可以进入**强制校准**（仅电阻屏支持）。当你发现触摸屏不准的时候，可以使用这个办法强制校准。
- ⑤ 在系统启动时，一直按着 KEY1 不放（加载到 Font Check 时），可以强制更新字库。
- ⑥ 在系统启动时，一直按着 WK_UP 不放（加载到 FLASH 容量时），可以选择是否擦除所有文件（清空 SPI FLASH），当需要重新更新的时候，建议先用此方法擦除，再更新。
- ⑦ 本系统用到触摸按键 TPAD 做返回（类似手机的 HOME 键，**TPAD 在开发板右下角，‘Z’ 图标丝印，该区域是触摸按键区域！！手指轻轻一摸，即可返回**），所以请确保多功能端口 P10 的 ADC 和 TPAD 用跳线帽短接！

在 SYSTEM Starting...之后，系统启动 UCOSII，并加载 SPB 界面，在加载成功之后，来到主界面，主界面如图 66.2.2~66.2.4 所示：



图 66.2.2 综合实验系统主界面(2.8 寸屏版本)



图 66.2.3 综合实验系统主界面(3.5 寸屏版本)



图 66.2.4 综合实验系统主界面(4.3/7 寸屏版本)

从上面三张图可以看出，2.8 寸屏和 3.5 寸屏主界面有 2 个页面（滑动切换），而 4.3/7 寸屏的只有 1 个页面（不支持滑动），总共是 19 个图标。每个图标代表一大功能，主界面顶部具有状态栏，显示 4G 模块信号质量、运营商、TF 卡状态、CPU 使用率和时间等信息。

注意：4G 模块信号质量和运营商，必须是接了 ATK-GM510 或者是 ATK-GM196 4G 模块后，才可能正常显示的，否则信号质量显示灰色，运营商显示：无移动网。

这里推荐大家使用性价比更高些的 ATK-GM196 4G 模块，模块与开发板的连接方式如下：用杜邦线将开发板的 PB11(RX)接 4G 模块的 TXD 脚，将开发板的 PB10(TX)接 4G 模块的 RXD 脚。最后共地，开发板上随便找一个 GND 用杜邦线和 4G 模块的 GND 连接起来。

只有 4G 模块连接好，SIM 卡正常，且长按 4G 模块的 POWER 键开机后，开发板才会显示如图 66.2.4 所示的信号质量和运营商，才可以进行拨号功能测试！！如果不显示信号质量和运营商，请检查是否有 4G 模块，或者 4G 模块是否工作正常！

回到主界面，主界面默认是简体中文的，我们可以在系统设置里面设置语言，综合测试实验支持 3 种语言选择：简体中文、繁体中文和英文。

在进入主界面之后，开发板上的 LED0 开始有规律的短亮（每 2.5 秒左右亮 100ms），提示系统运行正常，我们可以通过 LED0 判断系统的运行状况。另外，如果运行过程中，出现 HardFault 的情况，系统则会进入 HardFault 中断服务函数，此时 LED0 和 LED1 都会闪烁，提示系统故障。同时在串口打印故障信息。通过串口，系统会打印其他很多信息，最常打印的是内存使用率，然后我们还可以通过 USART 对系统进行调试。

我们可以通过点击任何一个图标，选中，然后再次点击，即可进入该图标的功能。接下来，我们主要以 4.3 屏为例，给大家讲解综合实验。2.8 屏、3.5 屏和 7 寸屏操作基本一模一样，下面就不再分别贴图了。

在任何界面下，都可以通过按 TPAD 返回上一级，直至返回到主界面。

上面已经介绍完系统启动，下面开始介绍各个功能。

66.2.1 电子图书

双击主界面的电子图书图标，进入如图 66.2.1.1 所示的文件浏览界面：



图 66.2.1.1 文件浏览界面

图 66.2.1.1 中，左侧的图是我们刚刚进入的时候看到的界面（类似在 WIN7/WIN10 上打开我的电脑），可以看到我们有 2 个盘，磁盘名字分别是：正点原子。正点原子是我们 TF 卡的卷标（即磁盘名字），是板载 SPI FLASH 磁盘的卷标。注意：如果没有插入 TF 卡，则只会显示 ALIENTEK 这一个卷标。我们可以选择任何一个磁盘打开，并浏览里面的内容。

界面的上方，显示文件/文件夹的路径。如果当前路径是磁盘/磁盘根目录则显示磁盘图标，如果是文件夹，则显示文件夹图标，另外，如果路径太深，则只显示部分路径（其余用...代替）。界面的下方显示磁盘/文件夹信息。

界面的下方，显示磁盘信息/当前文件夹信息。对磁盘，则显示当前选中磁盘的总容量和可用空间，对文件夹，则显示当前路径下文件夹总数和文件总数，并显示你当前选中的是第几个文件夹/文件。

双击图 66.2.1.1 中的“正点原子”，打开 TF 卡，得到图 66.2.1.1 右侧图片所示的界面，选中 TEXT 文件夹，双击打开得到如图 66.2.1.2 所示界面：

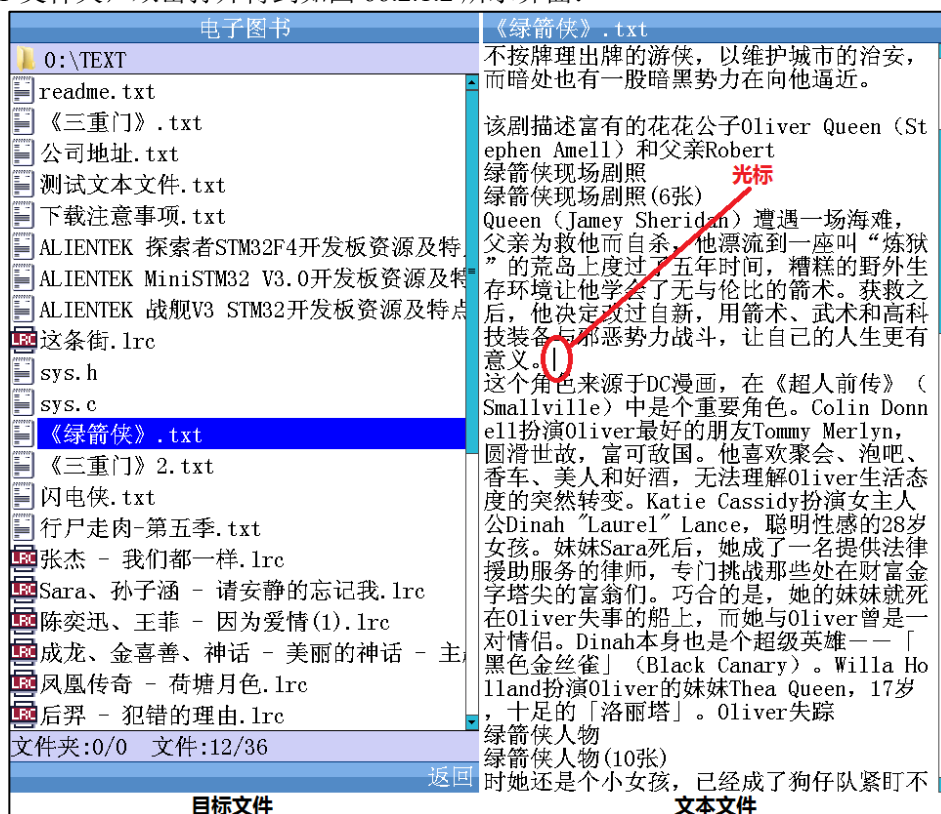


图 66.2.1.2 目标文件和文本阅读

图 66.2.1.2 左侧显示了当前文件夹下面的目标文件（即电子图书支持的文件，包括.txt/h/c/.lrc 等格式，其中.txt/h/c 文件共用 1 个图标，.lrc 文件单独一个图标）。另外，如果文件名太长，在我们选中该文件名后，系统会以走字的形式，显示整个文件名。

我们打开一个 txt 文件，开始文本阅读，如图 66.2.1.2 右侧的图片所示，同样我们可以通过滚动条/拖动的方式来浏览，图中我们还看到有一个光标，触摸屏点到哪，它就在哪里闪烁，可以方便大家阅读。

文本阅读是将整个文本文件加载到外部内存里面来实现的，所以文本文件最大不能超过外部内存总大小，即 963KB（这里仅指受内存管理的部分，不是整个外部 SRAM 的大小）。

当我们想退出文本阅读的时候，通过按 TPAD 触摸按键实现，按一下 TPAD，则又回到查找目标文件状态（左侧图），按返回按钮可以返回上一层目录，如果再按一次 TPAD 则直接返回主界面。

66.2.2 数码相框

双击主界面的数码相框图标，进入文件浏览界面，这个和 66.2.1 节差不多，我们找到存放

图片的文件夹，如图 66.2.2.1 所示：

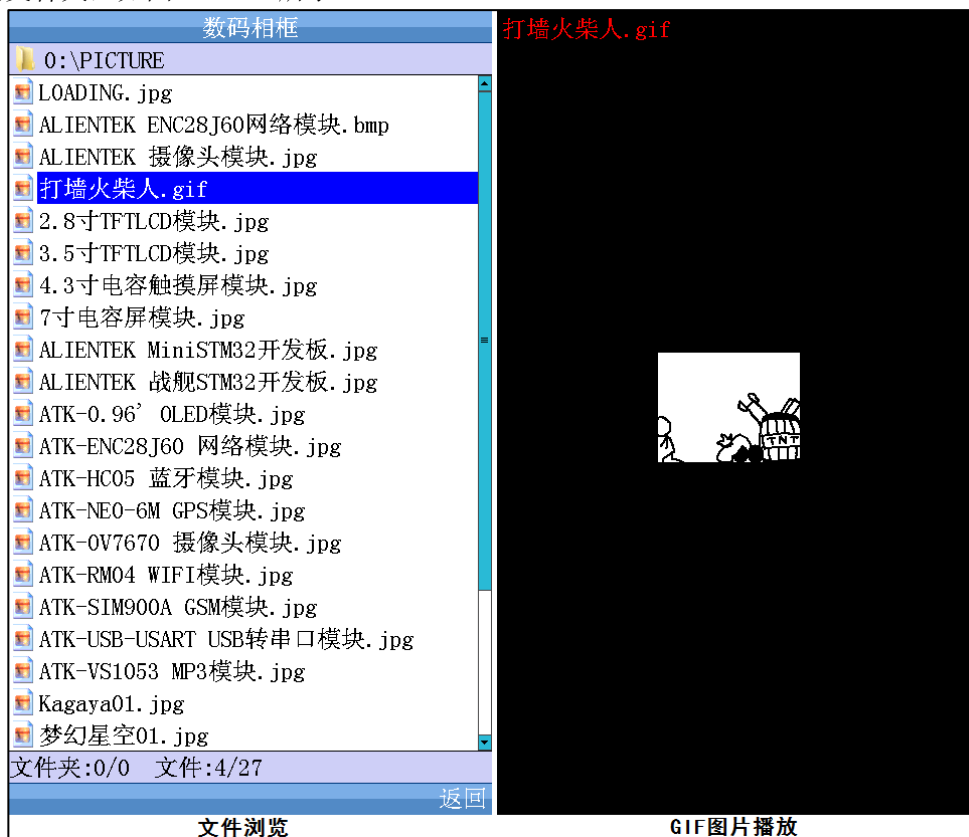


图 66.2.2.1 文件浏览和图片播放

左侧是文件浏览的界面，可以看到在 PICTURE 文件夹下总共有 27 个文件，包括 gif/jpg/bmp 等，这些都是数码相框功能所支持的格式。右侧图片显示了一个正在播放的 GIF 图片，并在其左上角显示当前图片的名字。当然，我们也可以播放 bmp 和 jpg 文件，如图 66.2.2.2 所示：



图 66.2.2.2 bmp 和 jpg 图片播放

对于 bmp 和 jpg 文件，基本没有尺寸限制（但图片越大，解码时间越久），但是对于 gif 文件，则只支持尺寸在 LCD 分辨率以内的文件（因为 gif 图片我们不好做尺寸压缩处理），超过这个尺寸的 gif 图片将无法显示！！

我们可以通过按屏幕的上方（1/3 屏幕）区域切换到上一张图片浏览；通过按屏幕的下方（1/3 屏幕）区域切换到下一章图片；通过单击屏幕的中间（1/3 屏幕）区域可以暂停自动播放，同时 LED1 亮，提示正在暂停状态，同样，通过按 TPAD 按钮，可以返回文件浏览状态。

图片浏览支持两种自动播放模式：循环播放/随即播放。大家可以在系统设置里面设置图片播放模式。系统默认是循环播放模式，在该模式下，每隔 4 秒左右自动播放下一张图片，依次播放所有图片。而随机播放模式，也是每隔 4 秒左右自动播放下一张图片，但是不是顺序播放，而是随机的播放下一张图片。

66.2.3 音乐播放

综合测试实验的音乐播放器性能非常不错，可作音乐播放器使用。支持的音频格式包括：MP3、WMV、OGG、WAV、FLAC、M4V、AAC 等。其中 MP3/WMA/OGG 支持全码率播放，WAV 和 FLAC 则不能支持高码率，如果播放出现不流畅，说明歌曲码率太高了，无法支持，请降低码率再试试。

注意：战舰 V4 自带了喇叭，播放音乐时，直接可以通过开发板自带的喇叭来听，不过需要在系统设置里面，开启板载喇叭（默认是开启的）。另外，如果有耳机，则可以插入开发板的 PHONE 接口，从而更好的享受音乐（立体声）。

双击主界面的音乐播放图标，进入文件浏览界面，这个和 66.2.1 节差不多，只是这里我们浏览的文件变为了 .mp3/.wma/.ogg/.wav/.flac/.m4a/.aac 等音频文件，我们找到存放音频文件的文件夹，如图 66.2.3.1 所示：



图 66.2.3.1 文件浏览和 wav 格式播放

左侧是文件浏览的界面，可以看到在音乐文件 文件夹下总共有 26 个音频文件，包括 wav/mp3/flac/ogg/wma 等格式，这些都是播放器所支持的格式。右侧图片则是我们播放器的主界面，该界面显示了当前播放歌曲的名字、播放进度、播放时长、总时长、采样率、位数、码率、音量、当前文件编号、总文件数、歌词等信息。下方的 5 个按键分别是：目录、上一曲、暂停/播放、下一曲、返回。点击播放进度条，可以直接设置歌曲播放位置，点击声音进度条，可以设置音量。上图为正在播放 mp3 文件，当然我们还可以播放其他音频格式，如图 66.2.3.2 所示：



图 66.2.3.2 ogg 和 wma 格式播放

图 66.2.3.2 中，分别展示了播放 ogg 格式和 wma 格式的音频文件。

播放器还可以设置音效和播放模式（均在系统设置里面设置）。音效包括 3D 效果、高音设置、低音设置等。播放模式有 3 种：全部循环、随机播放、单曲循环，默认为全部循环。

另外，关于歌词显示。歌词必须和歌曲在同一个文件夹里面，且名字必须相同（当然后缀是不同的，歌词后缀为.lrc），这样才能正常显示歌词。对于没有歌词文件的歌曲，则直接播放，不显示歌词。歌词采用多行显示，中间为当前正在演唱的歌词（粉红色字体显示），上下分别有预览歌词（白色字体显示），如果正在演唱的歌词太长，则会采用走字的形式来显示，走字时间由系统自动确定。

我们可以通过按目录按键，来选择其他音频文件；按返回按键（或 TPAD）则可以返回主界面，不过此时正在播放的歌曲还是会继续播放（后台播放），如果想关闭音乐播放器，则需要先按暂停，然后返回主界面，即可关闭音频播放器，否则音频播放器将一直播放音乐。

本音乐播放器支持多种无损音频格式播放，前面介绍的 mp3、ogg、wma 都是有损音频格式，本音乐播放器支持的无损音频格式为：wav 和 flac 两种，无损音频格式可以得到更好的音质，wav 和 flac 的播放，如图 66.2.3.3 所示：



图 66.2.3.3 wav 和 flac 格式播放

在播放 wav 和 flac 格式的时候，由于通过 VS1053 不能得到歌曲文件的正确码率，所以播放时间和歌曲总时间都是不正确的，因此歌词显示也不准确，在使用的时候，大家注意下这个问题。

66.2.4 TOM 猫

这是一个在智能手机上非常流行的游戏，你说一句话，游戏里的猫也跟着说一句，而且是以怪怪的音调（变调）模仿，十分有意思。

双击主界面的 TOM 猫图标，进入如图 66.2.4.1 所示界面：

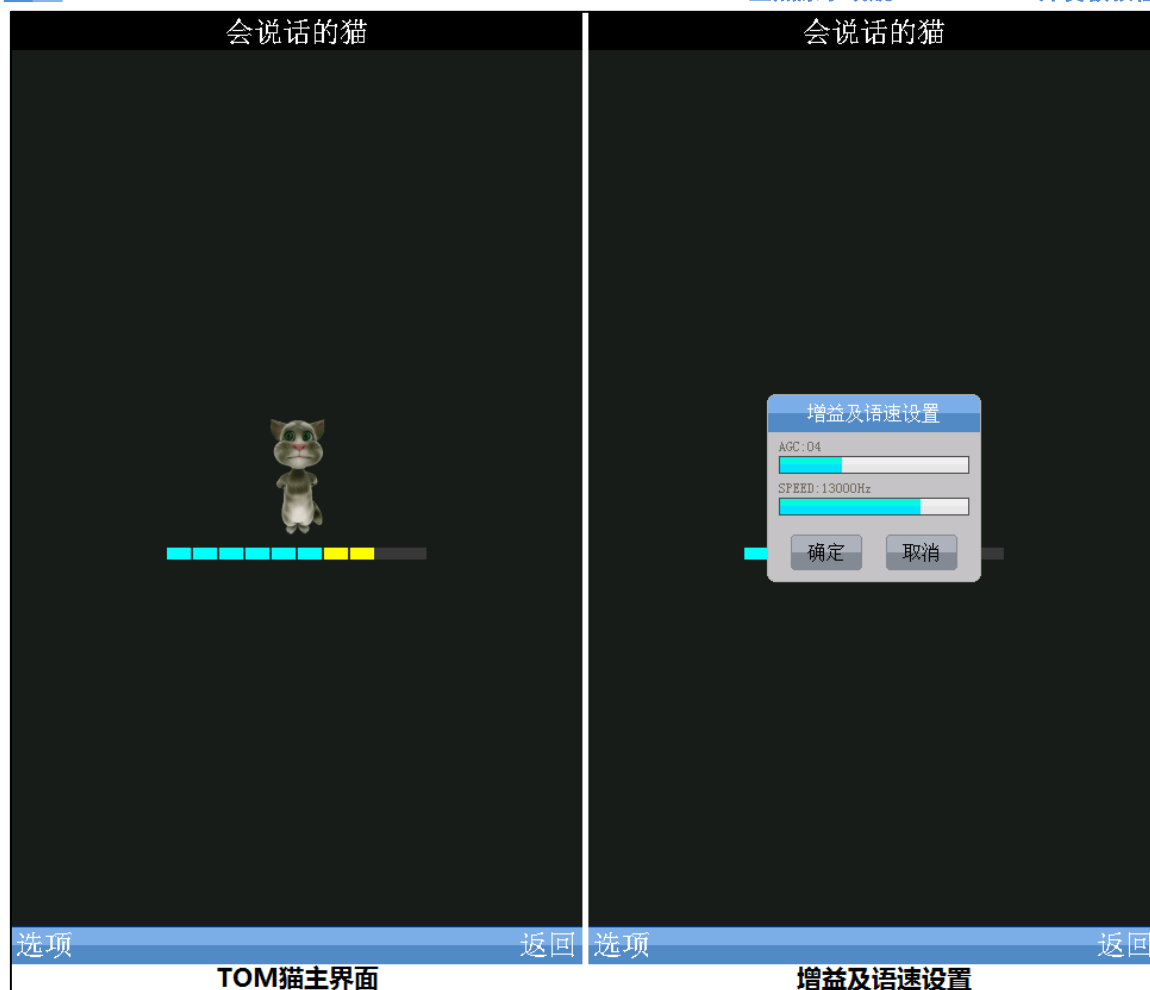


图 66.2.4.1 TOM 猫主界面和增益及语速设置界面

上图中，左侧图片为 TOM 猫游戏的主界面，图中显示了一个小猫和信号电平指示，此时我们可以对着 MIC（咪头）说话，你说一句，就可以通过板载喇叭或耳机（插开发板的耳机接口）里面听到 TOM 猫在重复你的句子，而且是以变调重复的，听起来和手机的 TOM 猫游戏差不多。

我们的 TOM 猫游戏还加入了语速设置，点击左侧图片里面的选项按钮，可以弹出增益及语速设置对话框，如右侧图片所示。在这个对话框里面，我们可以设置增益（AGC）和语速（SPEED），增益设置范围为 0~10，建议设置在 4 左右为最佳。

语速设置范围为 4000Hz~16000Hz，这里我们实现变调的原理很简单，就是人为改变 wav 文件的采样率，我们 wav 录音默认采样率为 8KHz，而如果我们强制修改采样率为其他值，那么语调就肯定发生了变化，我们通过将采样率设置为不同的值得到不同的语调，如果设置为 8KHz，就是正常语调了。默认我们设置语调为 13000Hz，这个语调比较接近手机的 TOM 猫效果，大家可以修改为其他值，比如设置为 4000Hz，听起来就像个老人的声音。

TOM 猫就为大家介绍到这里。

66.2.5 时钟

双击主界面的时钟图标，进入时钟界面，如图 66.2.5.1 所示：

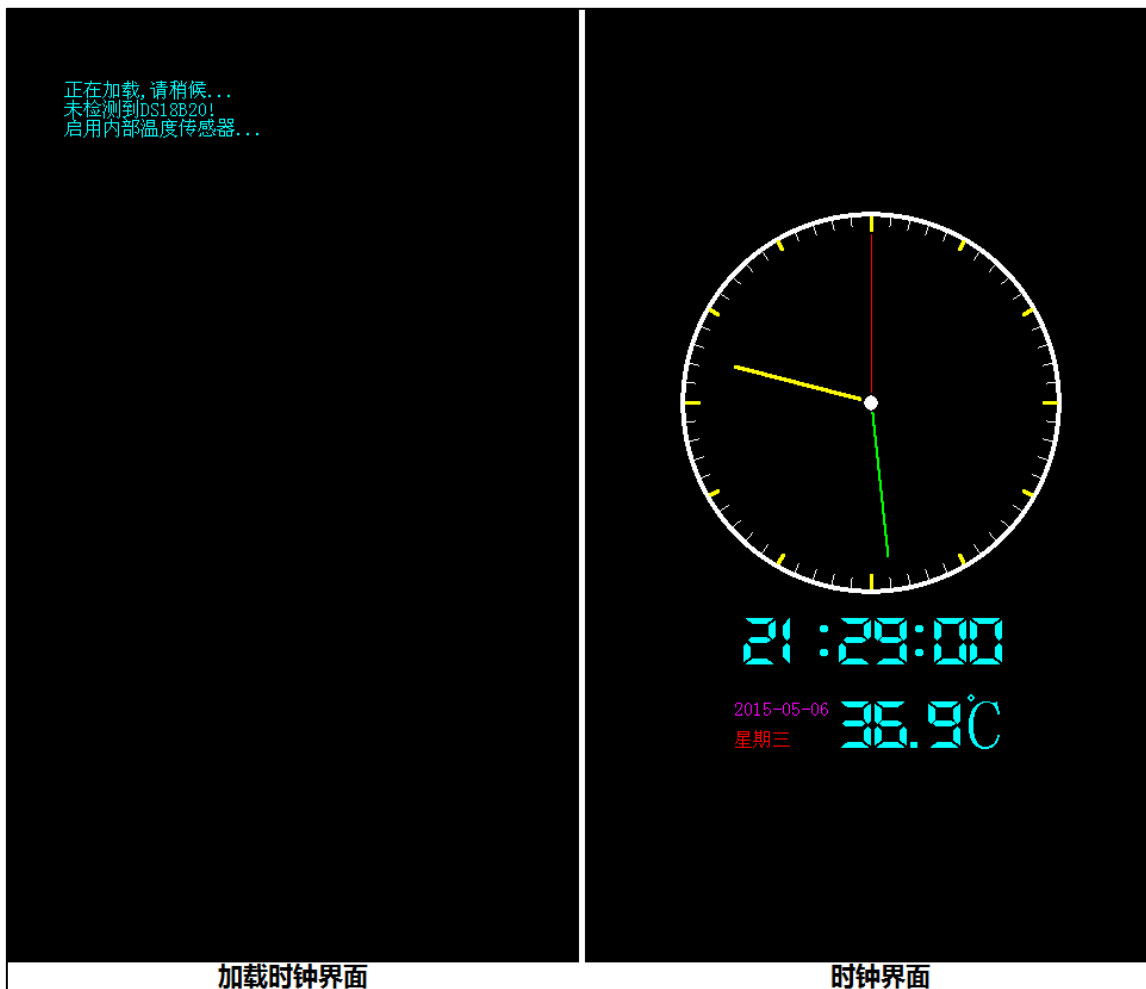


图 66.2.5.1 时钟界面

图 66.2.5.1 的左侧图片为加载时钟界面时的提示界面，表明没有检测到 DS18B20，启用 STM32F103 的内部温度传感器，之后进入时钟主界面，如右侧图片所示。在时钟界面，我们显示了日期、时间、温度、星期等信息，并且在屏幕上方区域，有一个指针式时钟显示。我们可以在系统设置里面设置时间和日期，并且还可以设置闹钟和闹铃，这个我们后面再介绍。

同样，按 TPAD 可以返回主界面。

66.2.6 系统设置

双击主界面的系统设置图标，进入系统设置界面，如图 66.2.6.1 所示：

| 系统设置 | 系统设置 |
|--------------------|--------------------|
| 1. 时间设置 | 1. 时间设置 |
| 2. 日期设置 | 2. 日期设置 |
| 3. 闹钟时间设置 | 3. 闹钟时间设置 |
| 4. 闹钟开关设置 | 4. 闹钟开关设置 |
| 5. 闹钟铃声设置 | 5. 闹钟铃声设置 |
| 6. 语言设置 | 6. 语言设置 |
| 7. 数码相框设置 | 7. 数码相框设置 |
| 8. 音乐播放器模式设置 | 8. 音乐播放器模式设置 |
| 9. 板载喇叭开关设置 | 9. 板载喇叭开关设置 |
| 10. VS1053 音量&3D设置 | 10. VS1053 音量&3D设置 |
| 11. VS1053 低音设置 | 11. VS1053 低音设置 |
| 12. VS1053 高音设置 | 12. VS1053 高音设置 |
| 13. 背光设置 | 13. 背光设置 |
| 14. 屏幕校准 | 14. 屏幕校准 |
| 15. 系统文件更新 | 15. 系统文件更新 |
| 16. 恢复默认设置 | 16. 恢复默认设置 |
| 17. 系统信息 | 17. 系统信息 |
| 18. 系统状态 | 18. 系统状态 |
| 19. 关于 | 19. 关于 |
| 确定 | 返回 |
| 系统设置主界面 | 时间设置 |

图 66.2.6.1 系统设置主界面和时间设置界面

上图中左侧的图片为系统设置主界面，在系统设置里面，总共有 19 个项目：时间设置、日期设置、闹钟时间设置、闹钟开关设置、闹钟铃声设置、语言设置、数码相框设置、音乐播放器模式设置、板载喇叭开关设置、VS1053 音量&3D 设置、VS1053 低音设置、VS1053 高音设置、背光设置、屏幕校准、系统文件更新、恢复默认设置、系统信息、系统状态、关于。通过这 19 个项目，我们可以设置和查看各种系统参数。下面我们将一一介绍这些设置。

首先是时间设置，如图 66.2.6.1 右侧图片所示，双击时间设置，就会弹出一个时间是指对话框，通过这个对话框，我们就可以设置开发板的时间了。设置好之后点击确定回到系统设置主界面，如果想放弃设置，则直接点击取消（或 TPAD）。

再来看看日期设置和闹钟时间设置，如图 66.2.6.2 所示：

| 系统设置 | 系统设置 |
|--|--|
| 1. 时间设置 | 1. 时间设置 |
| 2. 日期设置 | 2. 日期设置 |
| 3. 闹钟时间设置 | 3. 闹钟时间设置 |
| 4. 闹钟开关设置 | 4. 闹钟开关设置 |
| 5. 闹钟铃声设置 | 5. 闹钟铃声设置 |
| 6. 语言设置 | 6. 语言设置 |
| 7. 数码相框设置 | 7. 数码相框设置 |
| 8. 音乐播放器模式设置 | 8. 音乐播放器模式设置 |
| 9. 板载喇叭开关设置 | 9. 板载喇叭开关设置 |
| 10. VS1053 音量设置 | 10. VS1053 音量设置 |
| 11. VS1053 低音设置 | 11. VS1053 低音设置 |
| 12. VS1053 高音设置 | 12. VS1053 高音设置 |
| 13. 背光设置 | 13. 背光设置 |
| 14. 屏幕校准 | 14. 屏幕校准 |
| 15. 系统文件更新 | 15. 系统文件更新 |
| 16. 恢复默认设置 | 16. 恢复默认设置 |
| 17. 系统信息 | 17. 系统信息 |
| 18. 系统状态 | 18. 系统状态 |
| 19. 关于 | 19. 关于 |
| 日期设置 | 闹钟时间设置 |
| <div> <div>+</div> <div>+</div> <div>+</div> <div>2022</div> <div>12</div> <div>23</div> <div>-</div> <div>-</div> <div>-</div> <div>确定</div> <div>取消</div> </div> | <div> <div>+</div> <div>+</div> <div>16</div> <div>30</div> <div>-</div> <div>-</div> <div>确定</div> <div>取消</div> </div> |
| 日期设置 | 闹钟时间设置 |

图 66.2.6.2 日期设置和闹钟时间设置

上图中，左侧的对话框用来设置系统日期，右侧的对话框用来设置闹钟时间。操作上同前面介绍的时间设置的方法一模一样。关于闹钟，我们等下再详细介绍，先看闹钟开关设置和闹钟铃声设置两个界面，如图 66.2.6.3 所示：

| 系统设置 | 系统设置 |
|---|--------------------------------------|
| 1. 时间设置 | 1. 时间设置 |
| 2. 日期设置 | 2. 日期设置 |
| 3. 闹钟时间设置 | 3. 闹钟时间设置 |
| 4. 闹钟开关设置 | 4. 闹钟开关设置 |
| 5. 闹钟铃声设置 | 5. 闹钟铃声设置 |
| 6. 语言设置 | 6. 语言设置 |
| 7. 数码相框设置 | 7. 数码相框设置 |
| 8. 音乐播放器 | 8. 音乐播放器模式设置 |
| 9. 板载喇叭开 | 9. 板载喇叭开 |
| 10. VS1053 音量 | 10. VS1053 音量 |
| 11. VS1053 低音 | 11. VS1053 低音 |
| 12. VS1053 高音 | 12. VS1053 高音 |
| 13. 背光设置 | 13. 背光设置 |
| 14. 屏幕校准 | 14. 屏幕校准 |
| 15. 系统文件更 | 15. 系统文件更 |
| 16. 恢复默认设置 | 16. 恢复默认设置 |
| 17. 系统信息 | 17. 系统信息 |
| 18. 系统状态 | 18. 系统状态 |
| 19. 关于 | 19. 关于 |
| 闹钟开关设置 | 闹钟铃声设置 |
| 星期天 <input type="checkbox"/> | 铃声1 <input type="radio"/> |
| 星期一 <input checked="" type="checkbox"/> | 铃声2 <input type="radio"/> |
| 星期二 <input checked="" type="checkbox"/> | 铃声3 <input type="radio"/> |
| 星期三 <input checked="" type="checkbox"/> | 铃声4 <input checked="" type="radio"/> |
| 星期四 <input checked="" type="checkbox"/> | 确定 取消 |
| 星期五 <input checked="" type="checkbox"/> | |
| 星期六 <input type="checkbox"/> | |
| 确定 取消 | |
| 闹钟开关设置 | 闹钟铃声设置 |

图 66.2.6.3 闹钟开关设置和闹钟铃声设置

上图中，左侧对话框用来设置闹钟开关，右侧对话框用来设置闹钟铃声。这里，我们来介绍一下本系统的闹钟，本系统的闹钟以星期为周期，以时间为点实现闹钟，比如判断一个闹钟是否应该响铃的标准是：先判断星期的条件是否满足，比如上图我们设置是周一到周五闹铃，今天（2022 年 12 月 23 号）是周五，所以满足星期条件，接着看时间是否相等，如果两个条件都满足，则闹铃。假定当前时间是 11:56，我们设置的闹钟时间是 11:58，所以时间还不相等，故不闹铃，当时间来到 11:58 的时候，系统将会闹铃。闹铃铃声有 4 种，如上图右侧图片所示，铃声由蜂鸣器产生，铃声 1 对应“滴”，铃声 2 对应“滴、滴”，铃声 3 和 4 依此类推。当闹钟时间到来的时候，产生闹铃，如图 66.2.6.4 所示：

| 系统设置 | 系统设置 |
|---------------|---------------|
| 1. 时间设置 | 1. 时间设置 |
| 2. 日期设置 | 2. 日期设置 |
| 3. 闹钟时间设置 | 3. 闹钟时间设置 |
| 4. 闹钟开关设置 | 4. 闹钟开关设置 |
| 5. 闹钟铃声设置 | 5. 闹钟铃声设置 |
| 6. 语言设置 | 6. 语言设置 |
| 7. 数码相框设置 | 7. 数码相框设置 |
| 8. 音乐播放器模式设置 | 8. 音乐播放器模式设置 |
| 9. 板载喇叭开关设置 | 9. 板载喇叭开关设置 |
| 10. VS1053 音量 | 10. VS1053 音量 |
| 11. VS1053 低音 | 11. VS1053 低音 |
| 12. VS1053 高音 | 12. VS1053 高音 |
| 13. 背光设置 | 13. 背光设置 |
| 14. 屏幕校准 | 14. 屏幕校准 |
| 15. 系统文件更新 | 15. 系统文件更新 |
| 16. 恢复默认设置 | 16. 恢复默认设置 |
| 17. 系统信息 | 17. 系统信息 |
| 18. 系统状态 | 18. 系统状态 |
| 19. 关于 | 19. 关于 |
| 确定 | 返回 |
| 闹铃 | 语言设置 |

图 66.2.6.4 闹铃和语言设置

上图中，左侧的图片显示正在闹铃。此时会弹出一个闹钟的对话框，并显示当前时间，同时蜂鸣器发出“滴、滴、滴、滴”的闹铃声（铃声 4）。按取消（或 TPAD）可以关闭闹钟，按再响，则 5 分钟后（12:03）继续闹铃。右侧的图片为语言设置界面，系统支持 3 种语言设置，默认为简体中文，设置为繁体中文/English 之后如图 66.2.6.5 所示：

| 系統設置 | SYSTEM SET |
|--------------------|----------------------------|
| 1. 時間設置 | 1. TIME SET |
| 2. 日期設置 | 2. DATE SET |
| 3. 鬧鐘時間設置 | 3. ALARM TIME SET |
| 4. 鬧鐘開關設置 | 4. ALARM ON/OFF SET |
| 5. 鬧鐘鈴聲設置 | 5. ALARM RING SET |
| 6. 語言設置 | 6. LANGUAGE SET |
| 7. 數碼相框設置 | 7. DIGITAL PHOTO FRAME SET |
| 8. 音樂播放器模式設置 | 8. AUDIO PLAYER MODE SET |
| 9. 板載喇叭開關設置 | 9. ONBOARD SPEAKER SET |
| 10. VS1053 音量&3D設置 | 10. VS1053 VOL&3D SET |
| 11. VS1053 低音設置 | 11. VS1053 TREBLE SET |
| 12. VS1053 高音設置 | 12. VS1053 BASS SET |
| 13. 背光設置 | 13. BACKLIGHT SET |
| 14. 屏幕校準 | 14. TOUCH SCREEN ADJUST |
| 15. 系統文件更新 | 15. SYSTEM FILE UPDATE |
| 16. 恢復默認設置 | 16. RESTORE DEFAULT SET |
| 17. 系統信息 | 17. SYSTEM INFORMATION |
| 18. 系統狀態 | 18. SYSTEM STATUS |
| 19. 關於 | 19. ABOUT |
| 確定 | 返回 OK BACK |
| 繁體中文 | English |

图 66.2.6.5 繁体中文和 English

上图显示了繁体中文和 English 的设置，不过本章我们还是以简体中文为例进行介绍。下面，我们来看看数码相框设置和音乐播放器模式设置，如图 66.2.6.6 所示：

| 系統設置 | 系統設置 |
|---------------|---------------|
| 1. 時間設置 | 1. 時間設置 |
| 2. 日期設置 | 2. 日期設置 |
| 3. 鬧鐘時間設置 | 3. 鬧鐘時間設置 |
| 4. 鬧鐘開關設置 | 4. 鬧鐘開關設置 |
| 5. 鬧鐘鈴聲設置 | 5. 鬧鐘鈴聲設置 |
| 6. 語言設置 | 6. 語言設置 |
| 7. 數碼相框設置 | 7. 數碼相框設置 |
| 8. 音樂播放器模式設置 | 8. 音樂播放器模式設置 |
| 9. 板載喇叭開關設置 | 9. 板載喇叭開關設置 |
| 10. VS1053 音量 | 10. VS1053 音量 |
| 11. VS1053 低音 | 11. VS1053 低音 |
| 12. VS1053 高音 | 12. VS1053 高音 |
| 13. 背光設置 | 13. 背光設置 |
| 14. 屏幕校準 | 14. 屏幕校準 |
| 15. 系統文件更新 | 15. 系統文件更新 |
| 16. 恢復默認設置 | 16. 恢復默認設置 |
| 17. 系統信息 | 17. 系統信息 |
| 18. 系統狀態 | 18. 系統狀態 |
| 19. 關於 | 19. 關於 |
| 數碼相框設置 | 音樂播放器模式設置 |

图 66.2.6.6 数码相框设置和音乐播放器模式设置

前面提到数码相框支持全部循环播放和随机播放两种模式，就是通过上图左侧的界面设置的。而音乐播放器的三个播放模式，则通过右侧的界面进行设置。接下来看看板载喇叭开关设置和 VS1053 音量&3D 设置，如图 66.2.6.7 所示：



图 66.2.6.7 板载喇叭开关设置和 VS1053 音量&3D 设置

上图中，左侧的界面可以设置板载喇叭是否开启，战舰 V4 板载了一个喇叭，在开发板的背面，可以通过这个设置来选择是否开启。默认是开启板载喇叭的，如果设置为关闭喇叭，则板载的喇叭不会发声，需要插耳机到 PHONE 端口，才可以听到开发板的声音。

上图右侧是 VS1053 音量&3D 设置，可以设置 VS1053 的音量（在此处修改后，将写入 EEPROM 保存，而如果在音频播放的时候修改音量，是不会保存在 EEPROM 的），同时还可以设置 3D 效果，值越大，3D 效果越强，默认是 0，即关闭 3D 效果。

下面我们看看 VS1053 的音效设置和背光设置，如图 66.2.6.8 所示：

| 系统设置 | 系统设置 |
|---------------|---------------|
| 1. 时间设置 | 1. 时间设置 |
| 2. 日期设置 | 2. 日期设置 |
| 3. 闹钟时间设置 | 3. 闹钟时间设置 |
| 4. 闹钟开关设置 | 4. 闹钟开关设置 |
| 5. 闹钟铃声设置 | 5. 闹钟铃声设置 |
| 6. 语言设置 | 6. 语言设置 |
| 7. 数码相框设置 | 7. 数码相框设置 |
| 8. 音乐播放器模式设置 | 8. 音乐播放器模式设置 |
| 9. 板载喇叭开关 | 9. 板载喇叭开关设置 |
| 10. VS1053 音量 | 10. VS1053 音量 |
| 11. VS1053 低音 | 11. VS1053 低音 |
| 12. VS1053 高音 | 12. VS1053 高音 |
| 13. 背光设置 | 13. 背光设置 |
| 14. 屏幕校准 | 14. 屏幕校准 |
| 15. 系统文件更新 | 15. 系统文件更新 |
| 16. 恢复默认设置 | 16. 恢复默认设置 |
| 17. 系统信息 | 17. 系统信息 |
| 18. 系统状态 | 18. 系统状态 |
| 19. 关于 | 19. 关于 |



VS1053低音设置



背光设置

图 66.2.6.8 VS1053 音效（低音）设置和背光设置

上图中，左侧的界面用于设置 VS1053 的音效设置，VS1053 有 2 个音效设置：VS1053 低音设置和 VS1053 高音设置。这里我们仅以低音设置为例讲解，高音设置方法类似，这里就不列出来。图中的中心频率，即增益对应的频点，默认是 60Hz，可以通过中心频率滚动条设置（20Hz~150Hz），而增益默认设置的是 15dB，也可以通过滚动条设置（范围：0dB~15dB）。大家可以根据自己的喜好，来设置音效以达到最好的音乐享受。**注意：系统设置里面对 VS1053 的设置，都是立即生效的，并且会保存在 EEPROM 里面，当这里设置好之后，其他用到 VS1053 来播放音乐的程序（音乐播放、TOM 猫和 NES 游戏等），都会共用这些设置。**

右侧的界面用于设置 LCD 背光,背光通过 PWM 控制。当设置为 0 的时候,启动自动(auto)背光控制,其他值则是固定的背光亮度,值越大越亮。自动背光的时候,通过板载的光敏传感器(在摄像头座右侧,LS1)采集环境光强,自动调整背光。

第 14 项，屏幕校准，这里因为我们用的是 4.3 寸电容触摸屏为例讲解的，电容屏不需要校准，所以这个设置对 4.3 屏模块无效。如果是电阻屏，点击该项则可以进入屏幕校准，根据提示完成校准即可。

接下来，我们看看系统文件更新，如图 66.2.6.9 所示：



图 66.2.6.9 系统文件更新

上图中，左侧是双击系统文件更新提示，这里的系统文件是指 SYSTEM 文件夹里面除字库文件外的所有内容。综合测试实验之所以可以没有 TF 卡也能正常运行，主要是将 SYSTEM 文件夹（注意这个不是源码里面的 SYSTEM 文件夹!!）拷贝到了 SPI Flash Disk（即 25Q128）里面，这样，我们所有的系统资源都可以从 25Q128 里面获得，从而正常启动。

SYSTEM 文件夹目前是包含 171 个文件，总大小为 6.33MB。这些文件一般不要修改，如果你想自己 DIY 的话，那可以修改这些文件，以达到你要的效果，不过建议修改之前备份一下，搞坏了还可以还原。

如果在图 66.2.6.9 的系统文件更新提示时选择确定，则会执行系统文件更新，将 TF 卡的 SYSTEM 文件夹（拷贝自光盘：5，SD 卡根目录文件 里面的 SYSTEM 文件夹），拷贝到 FLASH Disk 里面。这里有个前提，就是你的 TF 卡里面必须有这个 SYSTEM 文件夹！更新时界面如图 66.2.6.9 右侧图片所示，该界面显示了当前更新的文件夹以及文件和进度等信息。

接下来，我们看看恢复默认设置和系统信息，如图 66.2.6.10 所示：



图 66.2.6.10 恢复默认设置和系统信息

上图左侧图片为恢复默认设置功能确认界面，当选择确定后，系统将恢复默认设置，除了 RTC 时间日期以外的所有设置，都将恢复默认值，方便大家在设置乱以后，恢复正常。

上图右侧图片为系统信息界面，通过该界面，可以看到软硬件的详细信息。

最后，我们来看看系统状态和关于界面，如图 66.2.6.11 所示：



图 66.2.6.11 系统状态和关于界面

上图中，左侧的界面显示了当前系统资源状况，显示了当前 CPU 使用率，CPU 温度以及内存使用率。

右侧的图片显示了战舰 STM32F103 开发板的软硬件版本以及产品序列号，这个序列号是全球唯一的，每个开发板都不一样。

66.2.7 FC 游戏机

综合测试实验移植了一个非常强大的 NES 模拟器，核心部分采用汇编实现，效率极高，支持音频输出，支持 MAP，支持绝大部分 NES 游戏的运行。综合性能超过 infoNES。该模拟器由开源电子网 (<http://www.openedv.com/forum.php>) 论坛网友：ye781205 编写，然后正点原子移植到本开发板上，即 FC 游戏机。

该 FC 游戏机特点如下：

- 1, 支持 MAP，可运行绝大部分小于 960K 的 NES 游戏。
- 2, 支持 FC 游戏手柄（9 针手柄，默认配送）。
- 3, 支持声音输出。
- 4, 支持全速运行（60 帧@2.8/3.5 寸屏），在 4.3 寸屏会放大 4 倍处理（480*480 分辨率），帧率约 43 帧/秒

双击主界面的 FC 游戏机图标，屏幕会弹出提示对话框，如图 66.2.7.1 所示：

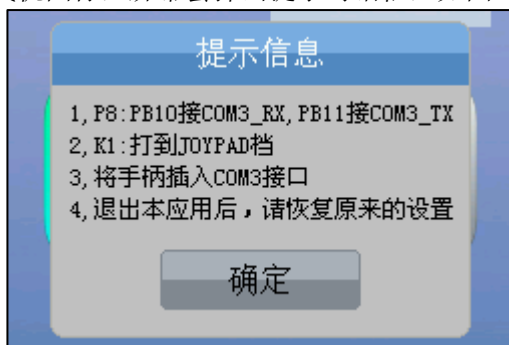


图 66.2.7.1 双击 FC 游戏机后提示界面

提示信息告诉我们，战舰 V4 的 P8 跳线帽，要连接 PB10 与 COM3_RX、PB11 与 COM3_TX，然后，将拨动开关（K1）打到 JOYPAD 位置，最后，将 FC 游戏手柄插入开发板的 COM3 接口。最后，在退出 NES 游戏后，需要将 K1 打到 RS232 位置，然后将 ATK-GM196 模块连接在 COM3，才可以进行拨号等操作。

确认好这些操作以后，点击确定，进入 nes 文件浏览界面，如图 66.2.7.2 所示：



图 66.2.7.2 文件浏览和超级玛丽游戏

上图中，左侧为 nes 文件浏览界面，我们随便选择一个打开即可开始游戏了，记得插上手柄哦！右侧的图片为经典的超级玛丽游戏界面，当然还可以玩很多其他经典游戏，如下面的图片所示：



图 66.2.7.3 冒险岛和魂斗罗



图 66.2.7.4 三木童子和双截龙

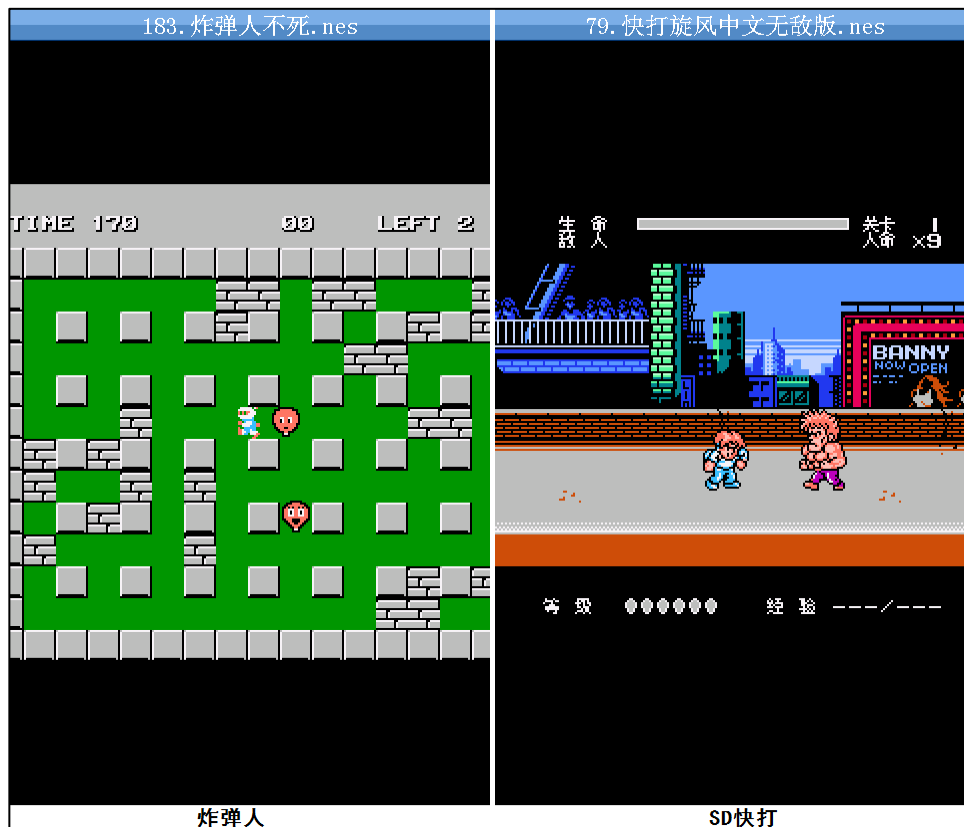


图 66.2.7.5 炸弹人和 SD 快打

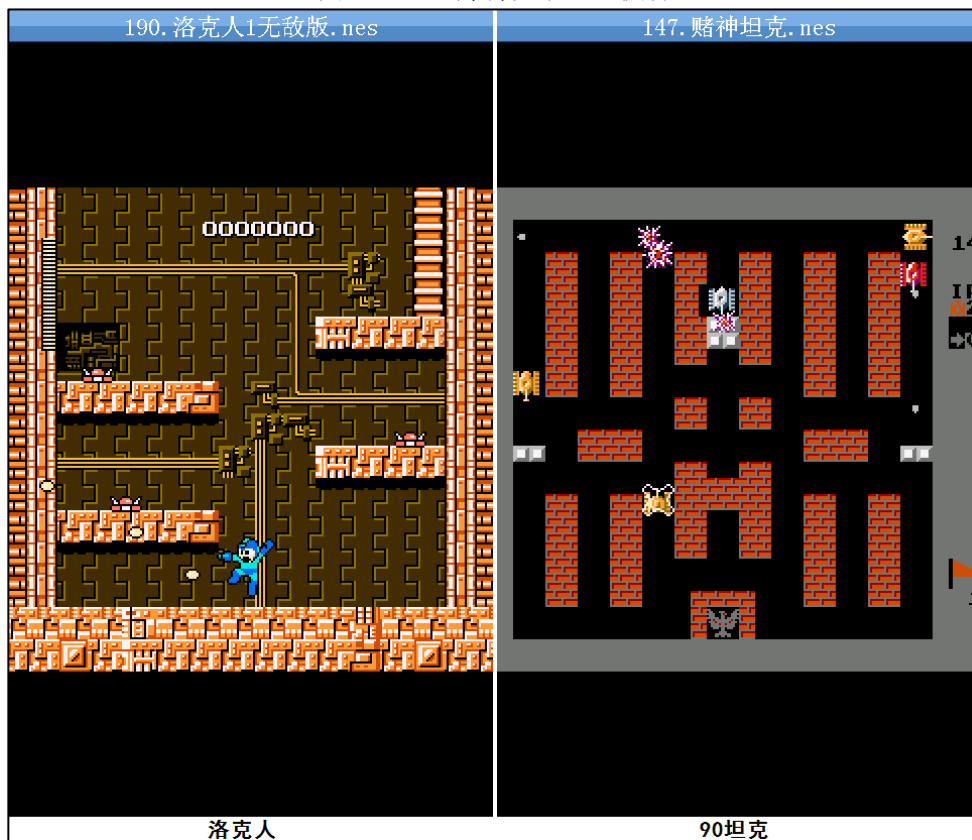


图 66.2.7.6 洛克人和 90 坦克

这里，我们仅列出了几种游戏，这都是 80 后童年时玩的经典游戏，如今，在战舰 V4 开发板上，大家可以回味一下当年的经典了。

66.2.8 记事本

双击主界面的记事本图标，首先弹出模式选择对话框，如图 66.2.8.1 所示：



图 66.2.8.1 模式选择和新建文本文件

记事本支持 2 种模式：1，新建文本文件，这种方式完全新建一个文本文件（以当前系统时间命名），用来输入信息。2，打开已有文件，这种方式可以对已有的文件进行编辑。

上图中，右侧的界面为我们选择新建文本文件后的界面，此时出现一个空白编辑区和一个闪烁的光标，我们通过下方的键盘输入信息即可，这个输入键盘和我们的手机键盘十分类似，输入方法也是一模一样，支持中文、字母、数字和手写识别输入等几种输入方式，如图 66.2.8.2 和图 66.2.8.3 所示：



图 66.2.8.2 中文输入和标点符号输入

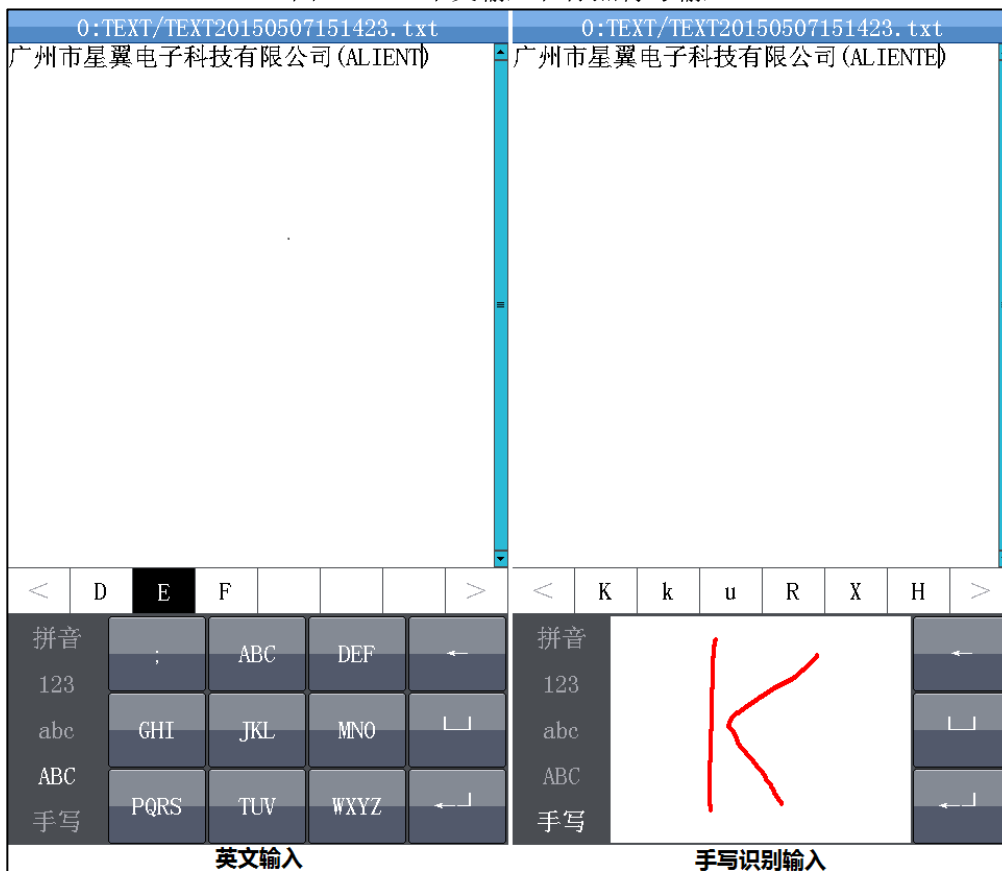


图 66.2.8.3 英文输入和手写识别输入

其中，中文输入就是我们前面 T9 拼音输入法实验的具体运用，而手写识别的输入界面，我们也是用到前面手写识别实验的知识实现的。

只要新建文本文件有被编辑过，那么在返回（按 TPAD 返回）的时候，系统会提示是否保存，如图 66.2.8.4 所示：

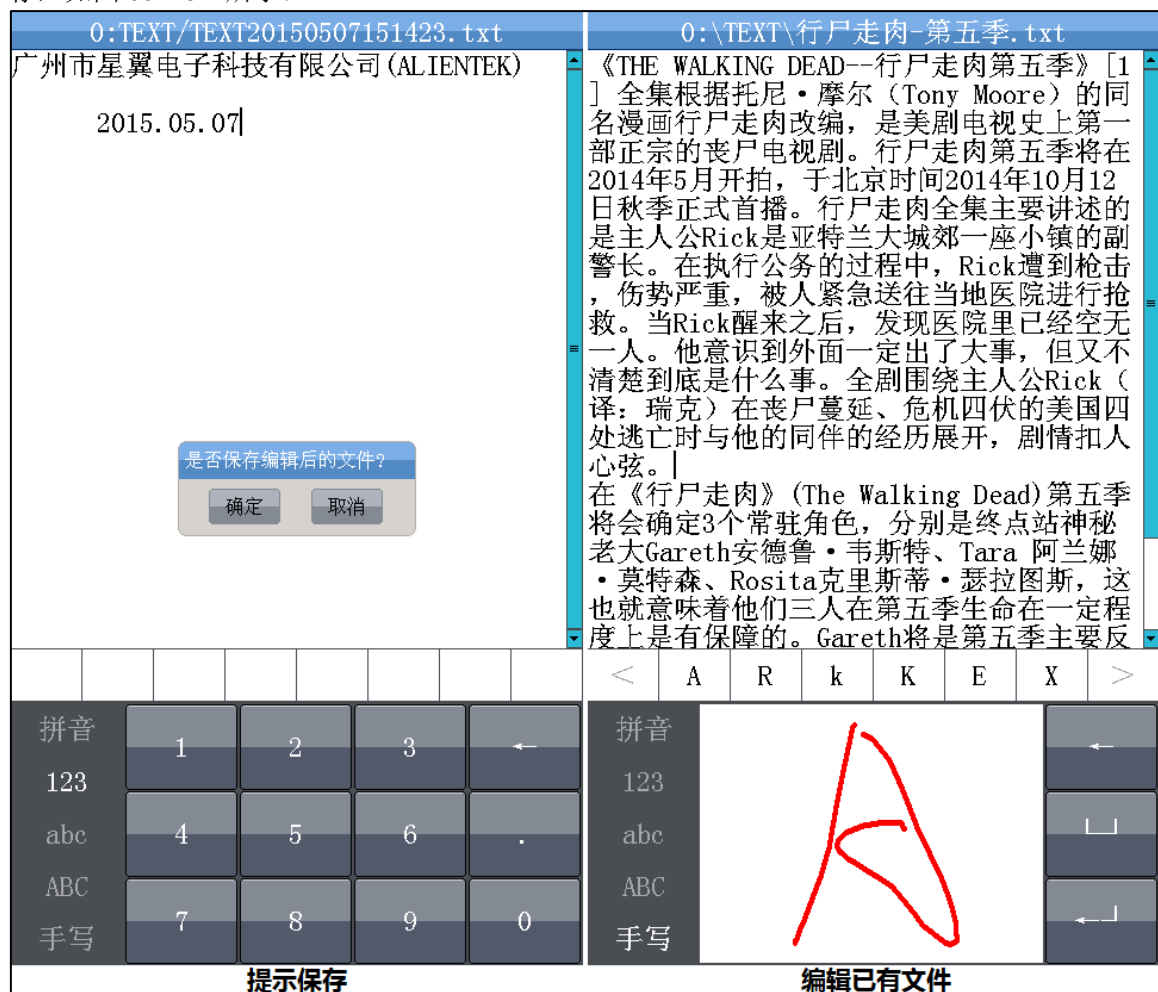


图 66.2.8.4 保存提示和编辑已有文件

上图中，左侧图片为提示保存界面，如果选择确定，该文件将被保存在 TF 卡根目录的 TEXT 文件夹里面。右侧图片为打开已有文件进行编辑的界面，这样我们就可以在开发板上编辑.txt/.h/.c/.lrc 文件了。

66.2.9 运行器

双击主界面的运行器图标，首先进入文件浏览界面，如图 66.2.9.1 所示：

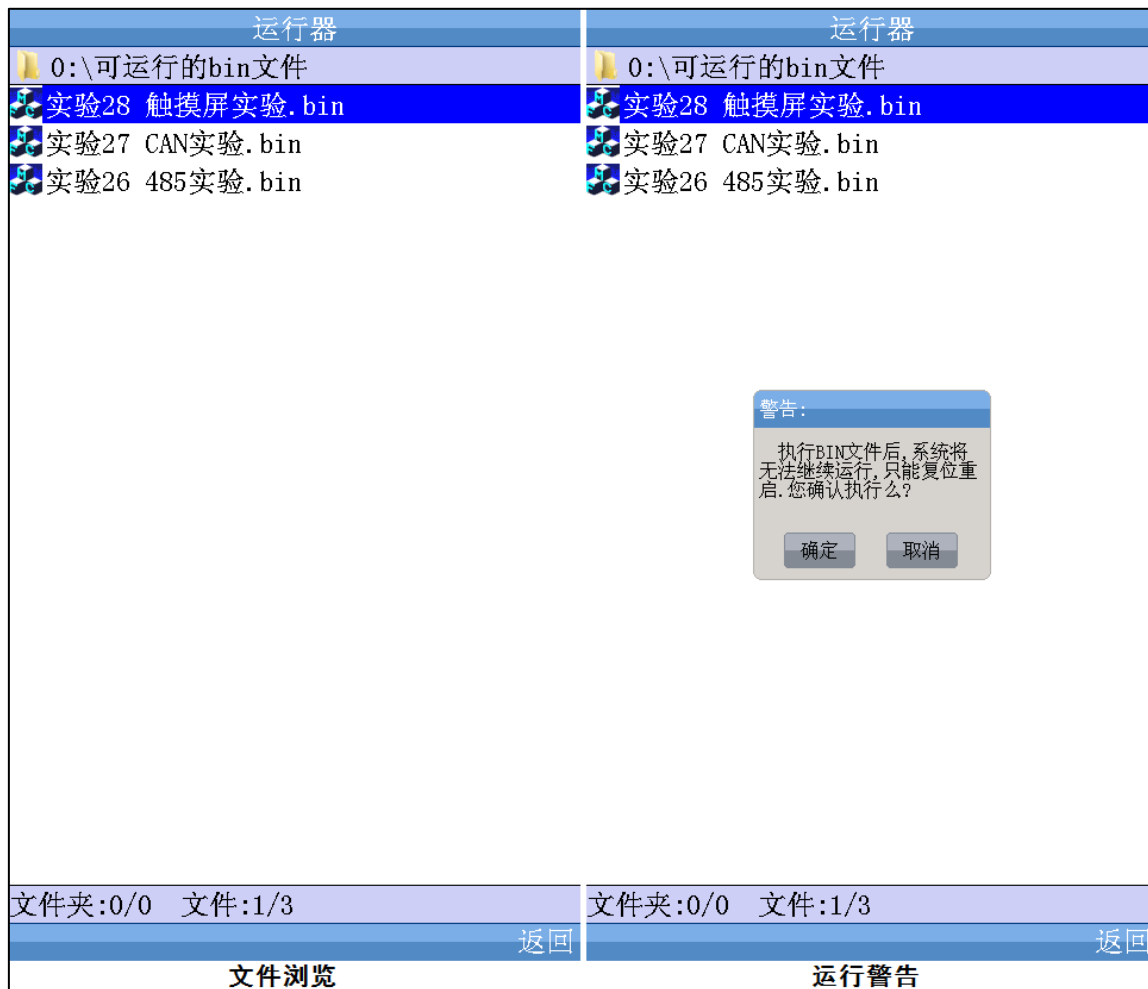


图 66.2.9.1 文件浏览和运行警告

上图中，左侧为文件浏览界面，图中显示了可运行的 bin 文件有 3 个，这些全部来自我们的标准例程对应的实验。本运行器支持 55K 字节以内的程序运行（FLASH+SRAM 总共不超过 55K），很多例程都可以生成 SRAM 版本的 bin 文件，在运行器里面运行。**SRAM 版本.bin 文件的生成办法，请参考串口 IAP 实验这个章节，里面有详细介绍。**通过运行器，大家可以直接运行我们大部分例程，而不用再去刷代码了，方便大家测试和验证我们的实验。

右侧的图片是运行前的警告界面，因为一旦执行 .bin 文件，我们的系统将无法恢复，只能靠复位重启。点击确定之后，STM32 就开始运行你所选择的 .bin 文件了，实验现象和对应实验所描述的现象一模一样。

66.2.10 手写画笔

双击主界面的手写画笔图标，首先弹出模式选择对话框，如图 66.2.10.1 所示：



图 66.2.10.1 模式选择和新建画板

上图中，左侧图片为我们双击手写画笔后，弹出的模式选择界面，我们可以选择新建画笔，建立一个新的文件；也可以选择打开一个已有的位图进行编辑。右侧的图片为我们新建画笔后输入的内容，默认画笔为最小尺寸，颜色为红色。画笔的颜色和尺寸是可以设置的，按 KEY_UP 按键，则弹出画笔设置对话框，然后，可以对画笔颜色和画笔尺寸进行设置。如图 66.2.10.2 和图 66.2.10.3 所示：



图 66.2.10.2 画笔设置和画笔颜色设置



图 66.2.10.3 画笔尺寸设置和完成后的画图

图 66.2.10.2 中，左侧的图片为按 KEY_UP 按键后弹出的画笔设置对话框，我们可以选择

对画笔颜色和画笔尺寸进行设置。右侧的图片为画笔颜色设置对话框，在该对话框里面，我们可以直接在颜色条快速输入要设置的颜色，也可以通过下方的三个滚动条进行精确设置，左侧的正方形区域为预览区。

图 66.2.10.3 中，左侧为画笔尺寸设置界面，我们可以通过滚动条设置画笔尺寸，对话框显示了画笔尺寸和对应的预览图。右侧的图片为我们完成的画图文件，在返回主界面（按 TPAD）的时候，会提示保存，如图 66.2.10.4 所示：



图 66.2.10.4 保存画图和编辑已有位图

上图中，左侧为我们退出时弹出的提示保存对话框，如果选择确定，则新的画图文件将会被保存在 TF 卡的 PAINT 文件夹里面，命名方式是以当前系统的时间命名的，如 PAINT20220620193550.bmp。

右侧的图片为对打开的位图进行编辑的界面，通过这个功能，我们可以在开发板上实现对一些相片（bmp 格式）进行涂鸦。

66.2.11 照相机

本照相机支持正点原子 OV7725 这款 30W 像素的 CMOS 摄像头模块，本照相机的特点有：

- 1，支持 BMP 拍照（拍下的 bmp 分辨率为 LCD 分辨率），按 KEY_UP 拍 BMP 照片。
- 2，支持 30 帧全速预览。
- 3，支持各种参数设置，包括：场景、特效、曝光、亮度、色度和对比度等。

双击主界面的照相机图标，首先初始化 OV7725 摄像头模块，如图 66.2.11.1 所示：



图 66.2.11.1 初始化 OV7725

在初始化 OV7725 之后，进入等待拍照模式，右侧的等待拍照时，显示区域只有一部分，并没有全屏显示，因为正点原子 OV7725 模块，输出分辨率一般是用 240*320，而我们演示用的屏幕分辨率为 480*800，所以，只有一部分区域有图像显示，其他区域用黑色填充。

此时我们可以通过点击屏幕，弹出相机设置对话框，对摄像头的参数进行设置，如图 66.2.11.2 所示：



图 66.2.11.2 相机设置

在相机设置界面，我们可以对很多参数进行调节。
再来看看场景设置和特效设置，如图 66.2.11.3 所示：

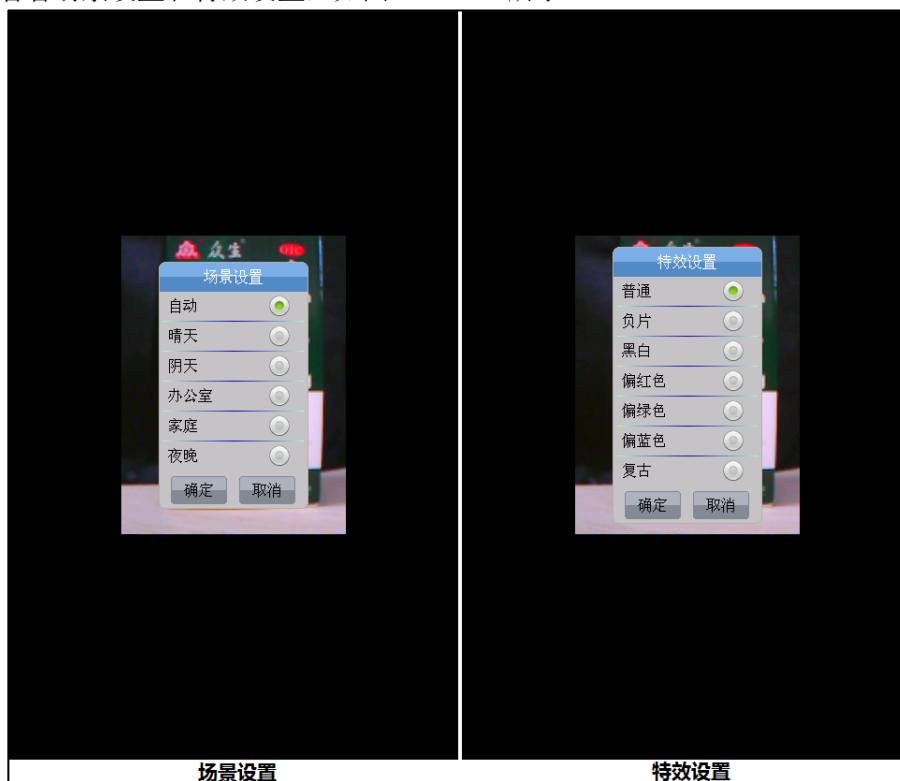


图 66.2.11.3 场景设置和特效设置

场景设置支持 6 种常用场景，特效设置支持 6 种特效（不含普通模式），我们可以根据自己的需要选择。

接下来看看亮度设置和色度设置，如图 66.2.11.4 所示：

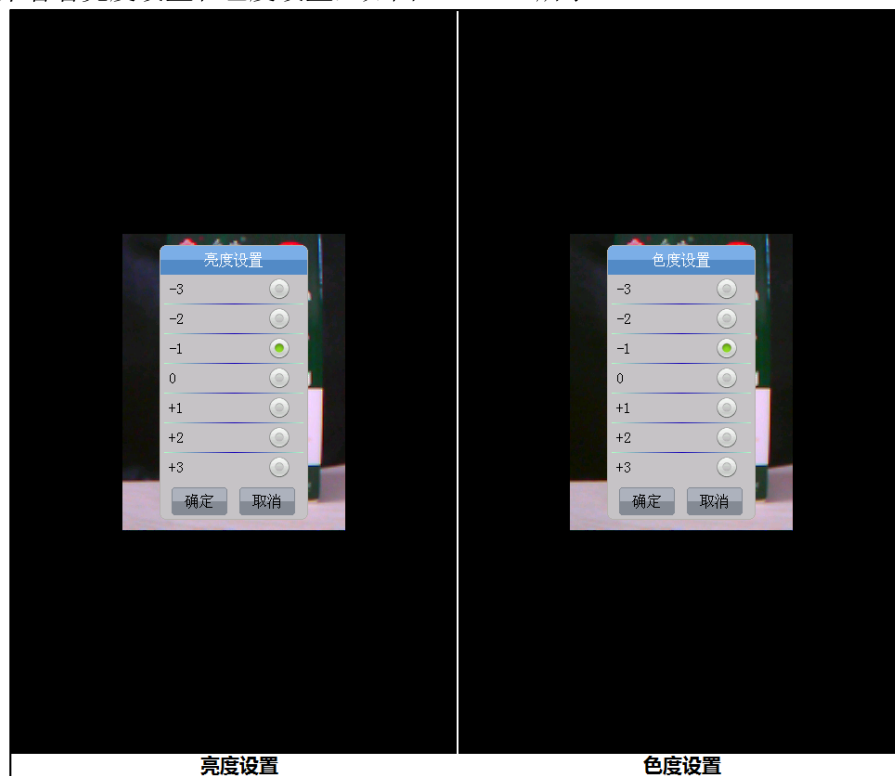


图 66.2.11.4 亮度设置和色度设置

亮度设置和色度设置各支持 7 个档位调节,我们可以根据自己的需要选择,默认都是-1 的。最后,看看对比度设置和拍照实现,如图 66.2.11.5 所示:



图 66.2.11.5 对比度设置和拍照

同样,对比度也支持 7 个档位设置,默认为-1。在个参数设置好之后,我们按下 KEY_UP 按键,就会执行拍照操作,在照片保存期间 LED1 亮,保存完后蜂鸣器发出“滴”的一声,提示拍照成功,同时弹出拍照成功对话框,如上图右侧图片所示。

从上图可以看出,照片文件的命名还是以当前时间为名字命名的。我们将所有的照片都保存在 TF 卡的 PHOTO 文件夹。如果你没有插入 TF 卡,拍照时会提示“创建文件失败,请检查!”的提示信息。

另外,如果你觉得照片模糊,可以手动调节摄像头模块的镜头,进行调焦,以达到最佳效果。

66.2.12 录音机

综合测试实验带了录音机功能,可以实现通过 MIC (咪头) 录音,并将录音文件保存在 TF 卡。录音文件为 WAV 文件,格式为:单声道、16 位、8Khz 采样率。

双击主界面的录音机图标,进入录音机主界面,如图 66.2.12.1 左侧图片所示:

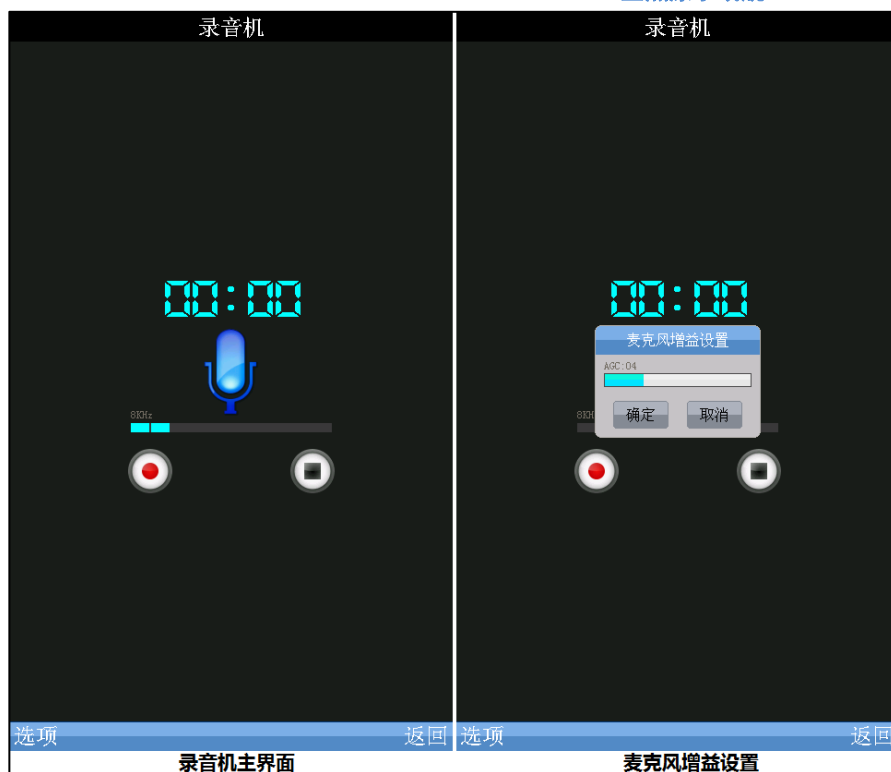


图 66.2.12.1 录音机主界面和麦克风增益设置

图中，左侧图片为录音机主界面，该界面显示了当前录音时间以及信号电平等，在该界面有两个按钮：左边的按钮用于开始/暂停录音，右边的按钮用于停止录音，并保存当前录音文件。

图中，右侧图片为麦克风增益设置界面，通过点击主界面左下角的选项按钮，可以弹出麦克风增益设置界面，麦克风增益设置，可以调节咪头(MIC)的灵敏度（其实就是调节：AGC），设置范围为：0~15，值越大，增益越高。0 代表自动增益，默认的增益值为 4。

我们在录音机主界面点击录音按钮，则开始录音，如图 66.2.12.2 所示：

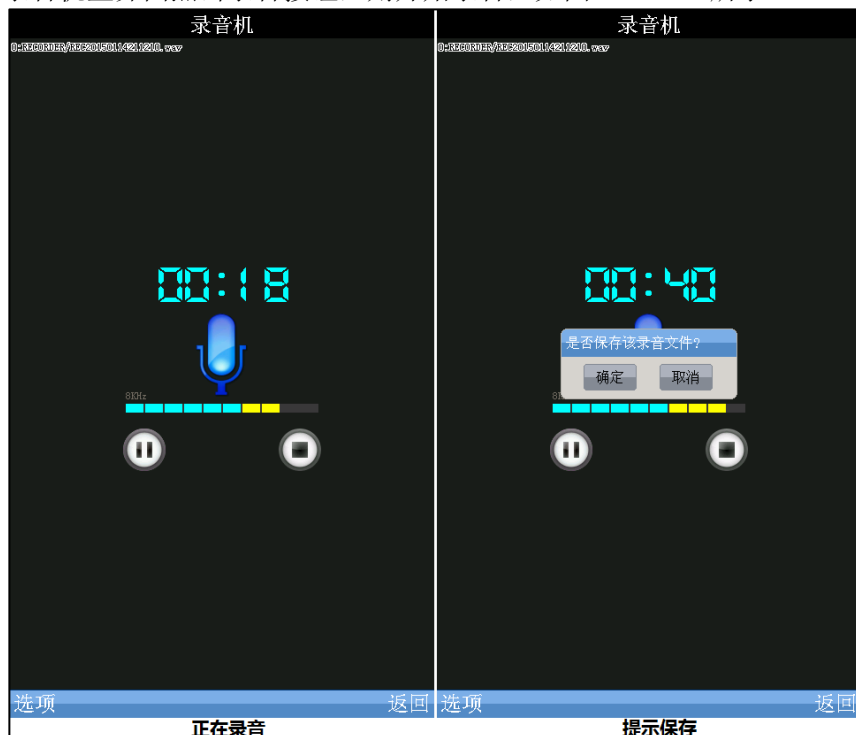


图 66.2.12.2 录音进行中和提示保存

在图 66.2.12.2 中，左侧的图片为正在录音的界面，此时我们可以按暂停/停止，按停止则自动保存当前录音文件，录音文件同样是以时间命名（见图中上方白字），所有录音文件都是被保存在 TF 卡根目录的 RECORDER 文件夹里面的。在录音的时候，按下 TPAD，会提示是否保存，如上图右侧图片所示，我们可以根据需要选择。

66.2.13 USB 连接

双击主界面的 USB 连接图标，如果开发板的 USB 端口没有连接电脑，则显示无连接，如图 66.2.13.1 所示：

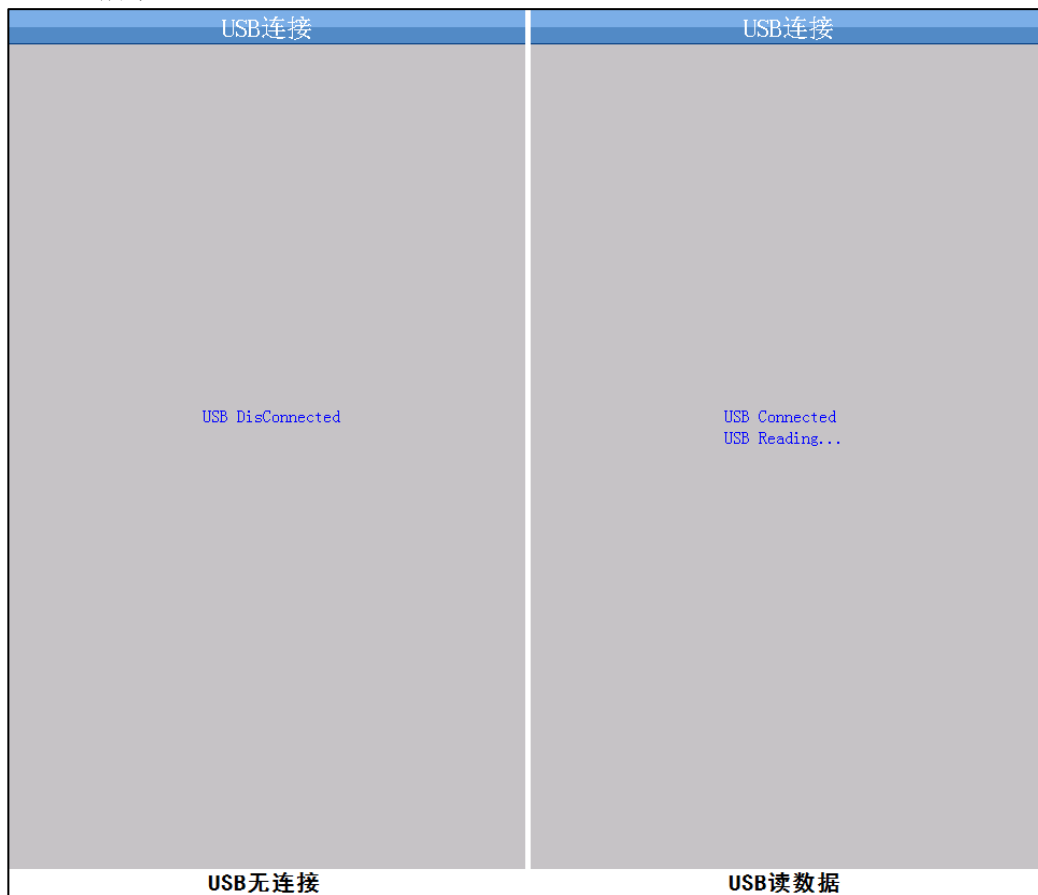


图 66.2.13.1 USB 无连接和 USB 读数据

上图中，左侧的图片显示开发板没有和电脑连接上，此时，我们找一根 USB 线，连接开发板的 USB_Slave 端口和电脑的 USB，然后，可以看到开发板提示 USB 已连接，并显示 USB 正在读数据，同时我们在电脑上面，可以看到右下角提示发现新硬件，并自动安装驱动（如果是第一次连接的话），如图 66.2.13.2 所示：

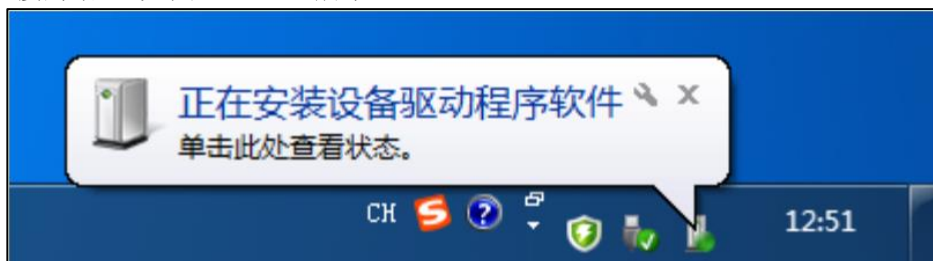


图 66.2.13.2 电脑发现新硬件

此时，我们打开我的电脑，即可找到可移动磁盘，如果有 TF 卡插入，那么会显示两个磁盘：ALIENTEK 磁盘和 TF 卡磁盘。如果 TF 卡没插入，则只显示 ALIENTEK 磁盘。这里的 ALIENTEK 磁盘即开发板板载的 SPI FLASH Disk。

这样，我们就实现了开发板和电脑的 USB 连接，可以直接从电脑拷贝文件到开发板的 TF 卡和 SPI Flash Disk（即 25Q128）。

这里再次提醒大家，如非必要，不要往 SPI Flash Disk 写入数据！否则容易写坏 SPI FLASH。

66.2.14 网络通信

开发板板载了一个 10M/100M 自适应以太网接口，可以实现网络通信。本系统的网络通信，就是给大家演示开发板的网络通信功能。

本系统自带的网络通信具有如下特点：

- 1，支持 DHCP，当 DHCP 失败时，使用静态 IP（静态 IP 地址为：192.168.1.100）。
- 2，自适应网线（支持交叉和直连网线）。
- 3，支持 TCP Client、TCP Server 和 UDP 等测试。

特别注意：本测试，必须在开发板网口接入网线，并连接正常后，才可以进行测试。也就是必须用网线连接开发板和电脑/路由器，才可以进行测试，否则，系统将使用默认 IP 地址：192.168.1.100!!!

双击主界面的网络通信图标，开始网卡初始化，在网卡初始化成功后，开始 DHCP 获取 IP 地址，在 DHCP 成功后，进入网络通信主界面，如图 66.2.14.1 所示：



图 66.2.14.1 初始化网卡和网络通信主界面

上图中，左侧图片显示正在初始化网卡，此时，我们需要在开发板的网口插入网线，并连接电脑或者路由器。初始化网卡成功后，则开始 DHCP 获取 IP 地址：

1，开发板连接到路由器，此种方式，DHCP 一般可以成功获取 IP 地址。

2，如果是直接连接电脑，那么 DHCP 肯定失败，最终会使用静态 IP 地址：192.168.1.100，此时需要设置电脑 IP 地址为 192.168.1.XXX，其中 XXX 可以由用户自己随意设置（但是不能是 1 和 100）。

这里我们以直接连接到电脑为例，DHCP 获取失败后，进入网络通信主界面，如图 66.2.14.1 右侧图片所示，该界面显示了开发板网卡的详细设置和参数，包括：本机 MAC 地址、本机 IP 地址（DHCP 获取）、子网掩码、网关和网速等。

能进入网络通信主界面，说明开发板与电脑的连接已经正常了，可以在电脑端 ping 屏幕显示的 IP 地址，即可查看网络是否连接正常，如图 66.2.14.2 所示：

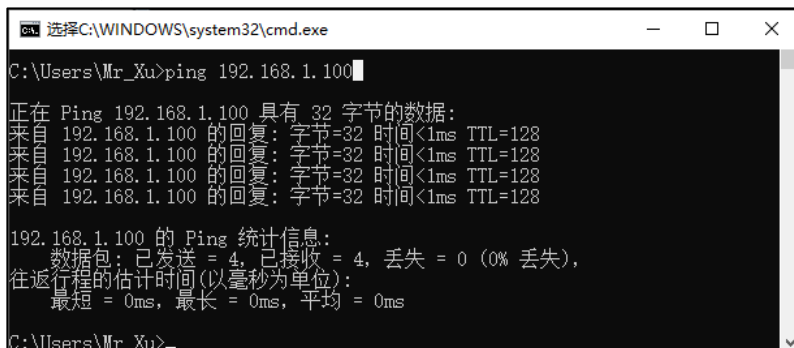


图 66.2.14.2 ping 192.168.1.100 成功

从上图可以看出，ping 开发板 IP 地址是成功的，说明网络连接正常了。

点击图 66.2.14.1 右侧图片所示的开始测试，即可进入：TCP Server、TCP Client 和 UDP 的测试界面，如图 66.2.14.3 所示：



图 66.2.14.3 TCP&UDP 测试界面和协议选择

从图 66.2.14.3 左侧图片可以看出：最顶部，显示了 TCP Server 的本机 IP 地址和端口号，其中 IP 地址是白色，表示不可以设置；端口号是绿色，表示可以设置，设办法：触摸点击该区域，看到光标闪烁后，即可输入数字进行设置。随后，显示 TX，RX，和协议等三个信息，分

别代表发送接收的数据量，和当前所使用的协议类型。

然后就是一个接收区和发送区，分别用于显示接收到的数据，和发送的数据。这个同电脑端的网络调试助手一样。右边的协议选择按钮，可以选择不同协议（如图 66.2.14.3 右侧图片所示），该选择按钮只有在连接断开的时候，才有效。连接按钮，用于启动连接（TCP Server/TCP Client 和 UDP 等），连接结果会有对话框提示（成功或者失败）。清除接收按钮，则可以清除接收区的所有数据，同时清除 RX 和 TX 计数器。发送按钮，则用于发送数据，每按一次，发送区的数据就发送一次。

进入 TCP&UDP 测试后，协议默认选择的是 TCP Server，可以通过协议选择按钮，选择不同协议（TCP Server、TCP Client 和 UDP），如上图右侧图片所示。

我们使用默认的 TCP Server 端口号（8088），然后点击链接，提示：连接成功后，即可开启开发板的 TCP Server 服务，然后在电脑端，打开网络调试助手，设置正确的网络参数后（**如不懂设置方法，请参考《网络开发指南》对应测试部分，下同！！**），即可连接上开发板的 TCP Server，并互相通信，如图 66.2.14.4 所示：

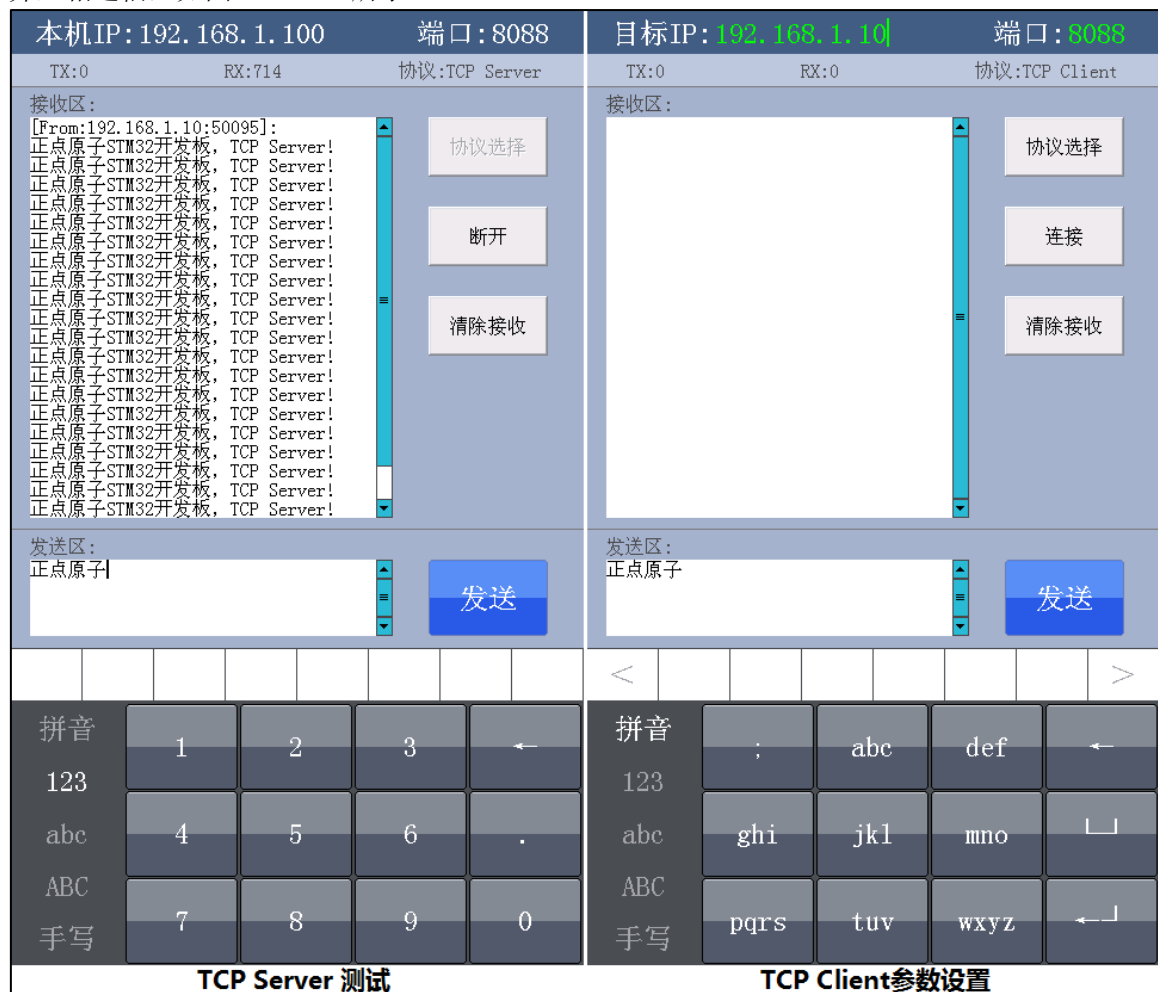


图 66.2.14.4 TCP Server 测试和 TCP Client 参数设置

上图中，左侧图片为 TCP Server 测试界面，可以看到，我们收到来自电脑的数据，接收数据时，首先会在接收区提示收到的数据来自何处（即电脑端的 IP 地址，本例可以看出，电脑 IP 地址为：192.168.1.10），随后才是电脑端发送过来的数据。同时，我们也发送了一些数据给电脑（电脑端网络调试助手可以看到，这里就不截图出来了）。此时，协议选择按钮变为了灰色，处于无效状态。只有在断开连接后，才可以选择新的协议。

TCP Client 的测试如图 66.2.14.4 中右侧图片所示，我们选择了 TCP Client 协议之后，顶部 IP 变为了目标 IP，也就是 TCP Client 要连接的 IP 地址，我们可以根据自己需要设置对应的 IP 地址和端口号。这里，我们设置为：192.168.1.100，端口为：8088。注意：这里的 IP 和端口号，

要根据自己的电脑实际情况修改。随后需要在电脑端开启网络调试助手，并设置正确的网络参数（主要是端口号和 IP 地址），然后开启 TCP Server 服务。

TCP Client 测试必须等电脑端 TCP Server 服务开启后，我们才可以在开发板点击连接，并连接成功，否则肯定是连接失败的。连接成功后，TCP Client 的测试如图 66.2.14.5 所示：



图 66.2.14.5 TCP Client 测试和 UDP 测试

上图中，左侧为 TCP Client 测试界面，右侧为 UDP 测试界面，这两个测试界面和 TCP Server 差不多，请参考前面的介绍。

UDP 的测试，同 TCP Client 基本一样，也是先设置目标 IP 和端口号，然后按连接，进行测试。UDP 测试的时候，一般需要开发板先发送一次数据给电脑端的网络调试助手，随后才可以实现数据互发。UDP 测试界面如图 66.2.14.5 右侧图片所示。注意：UDP 不是基于可靠连接的通信，所以程序提示连接成功的时候，仅仅是个提示作用，并不是说就一定连接上了目标 IP 和端口号，因此，不一定能成功发送数据给对方，这个在使用的时候，要注意。

网络通信就为大家介绍到这里。

66.2.15 无线传书

该功能用来实现两个开发板之间的无线数据传输，在开发板 A 输入的内容，会在开发板 B 上完整的“复制”一份，该功能需要 2 个战舰 V4 开发板（也可以一个战舰板与精英板或探索板或 Mini 板搭配用，不过都要刷综合实验!!）和 2 个 NRF24L01 无线模块。

双击主界面的无线传书图标（假定开发板已插上 NRF24L01 无线模块），会先弹出模式选择对话框，如图 66.2.15.1 所示：



图 66.2.15.1 模式选择和发送模式

从左侧的图片可以看出，模式设置，我们可以设置为发送模式或接收模式。右侧的图片则是选择发送模式后进入的界面。我们在另外一块开发板（开发板 B）设置模式为接收模式，然后在本开发板（开发板 A）手写输入一些内容，就可以看到在另外一个开发板也出现了同样的内容，如图 66.2.15.2 所示：

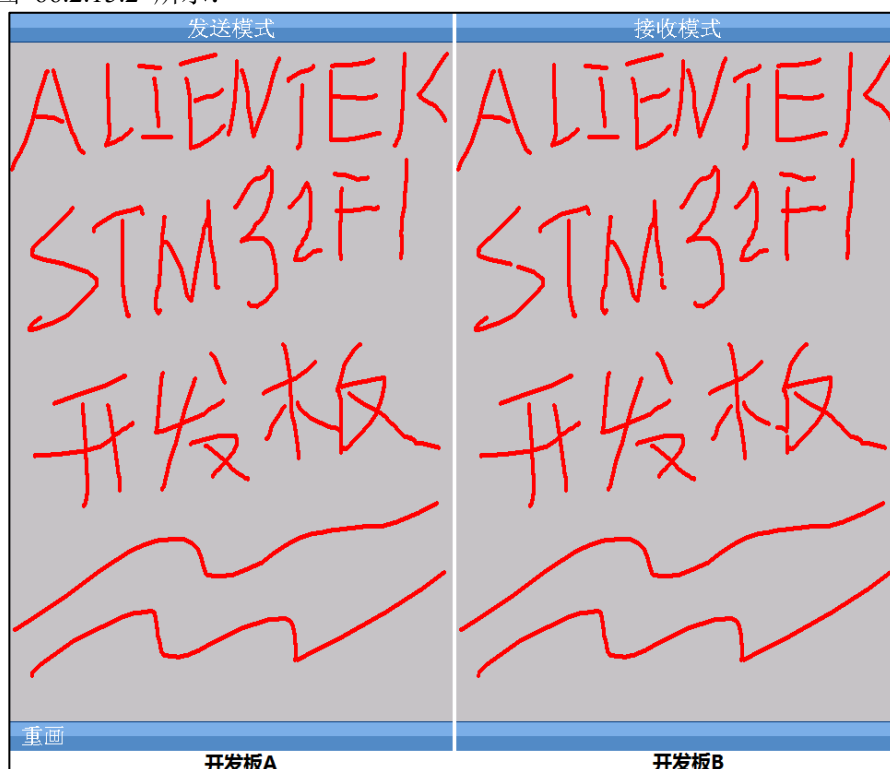


图 66.2.15.2 在开发板 A 输入的内容完整的显示在开发板 B 上

从上图可以看出，在开发板 A 上输入的内容，被完整的复制到开发板 B 上了。这就是无线传书功能。

66.2.16 计算器

本开发板实现了一个简单的科学计算器，可以计算加减乘除、开方、平方、 M^N 次方、正弦、余弦、正切、对数、倒数、格式转换等一些常见的计算器功能，精度为 12 位，支持科学计数法表示。双击主界面的计算器图标，进入计算器主界面，如图 66.2.16.1 所示：

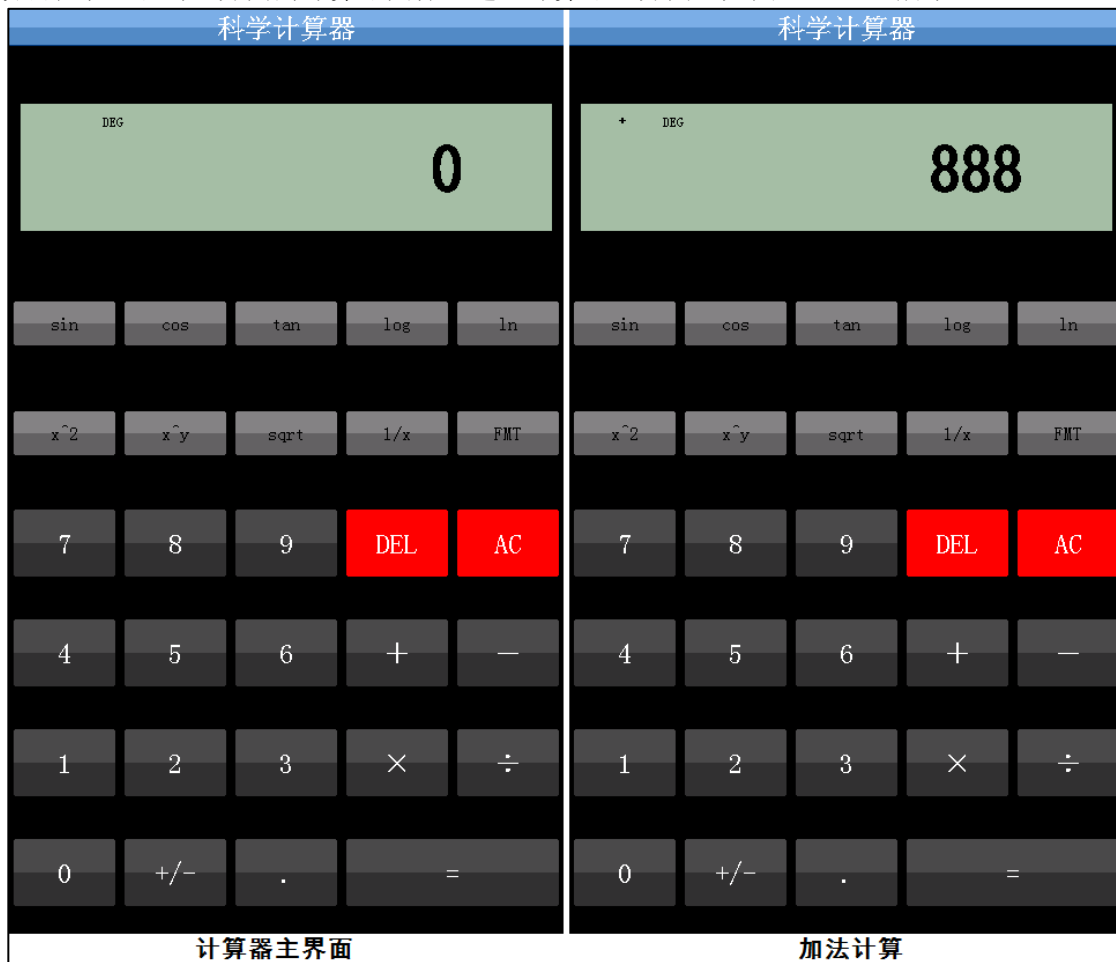


图 66.2.16.1 计算器主界面和加法计算

上图中，左侧的图片为科学计算器的主界面，和我们手机用的计算器基本一样，使用上非常简单，我们就不详细介绍了。右侧的图片为加法计算，支持累加功能。



图 66.2.16.2 计算器主界面和加法计算

上图为乘法计算和倒数计算，可以看到，结果是以科学计数法表示的，最大支持 200 位指数表示，超过范围直接显示错误 (E)。

该计算器还支持格式转换（按 FMT 键），可以将十进制数据（最大为 65535，超过部分将被丢弃）转换为 16 进制/二进制数据表示，如图 66.2.7.3 所示：



图 66.2.16.3 格式转换

上图显示我们将十进制的 65535 转换为 16 进制/二进制后的表示。计算器的其他功能，我们就再列举了，感兴趣的朋友可以慢慢摸索，当然也可以在这个基础上进行改进。通过按 TPAD 可以返回主界面。

66.2.17 拨号

本综合实验支持拨打和接听电话，不过需要正点原子 ATK-GM510/ATK-GM196 模块的支持，所以本功能的测试，请先确保有 4G 模块，并连接成功（详见 66.2 节开头部分）。

双击主界面的拨号图标，进入拨号界面，如图 66.2.17.1 所示：

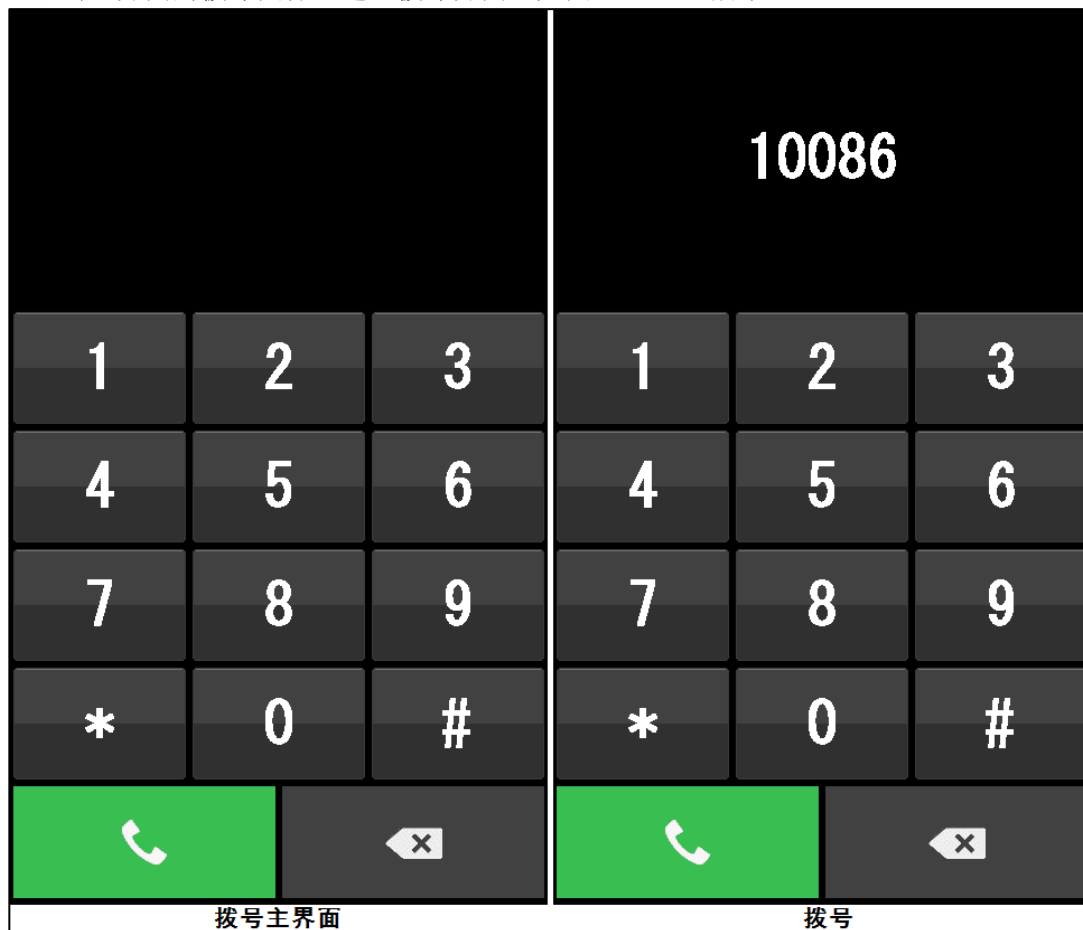


图 66.2.17.1 拨号主界面和拨号

上图中左侧图片就是拨号主界面，这个和手机拨号是一样的。右侧是我们输入的拨号号码，点击拨号图标，即可进行拨号。如图 66.2.17.2 所示。



图 66.2.17.2 拨号中和通话中

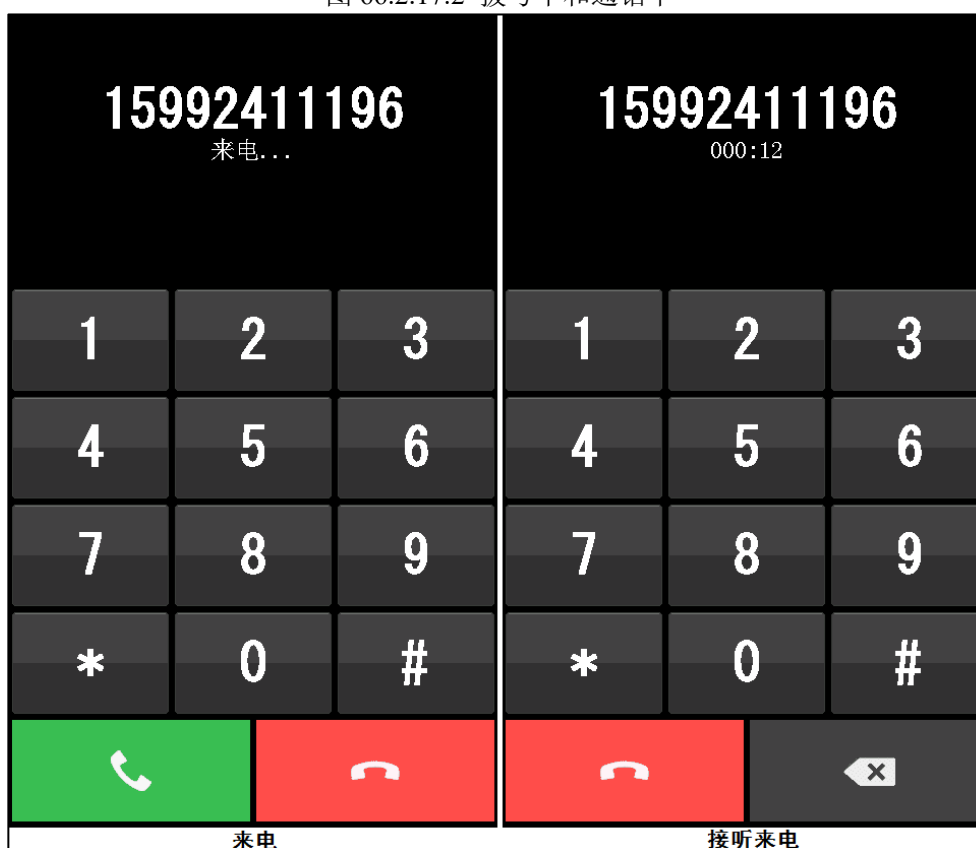


图 66.2.17.3 来电和接听来电

图 66.2.17.2 为拨号和拨通后的通话界面。图 66.2.17.3 为来电和接听来电后的通话界面，此

时蜂鸣器会发出“滴、滴”的提示声，提示有电话呼入。其他的操作和我们智能手机基本一模一样，就无需多说了。

注意，在通话状态，如果按 TPAD，则会挂断电话，结束通话。

66.2.18 应用中心

双击主界面的应用中心图标，进入应用中心界面，如图 66.2.18.1 所示：

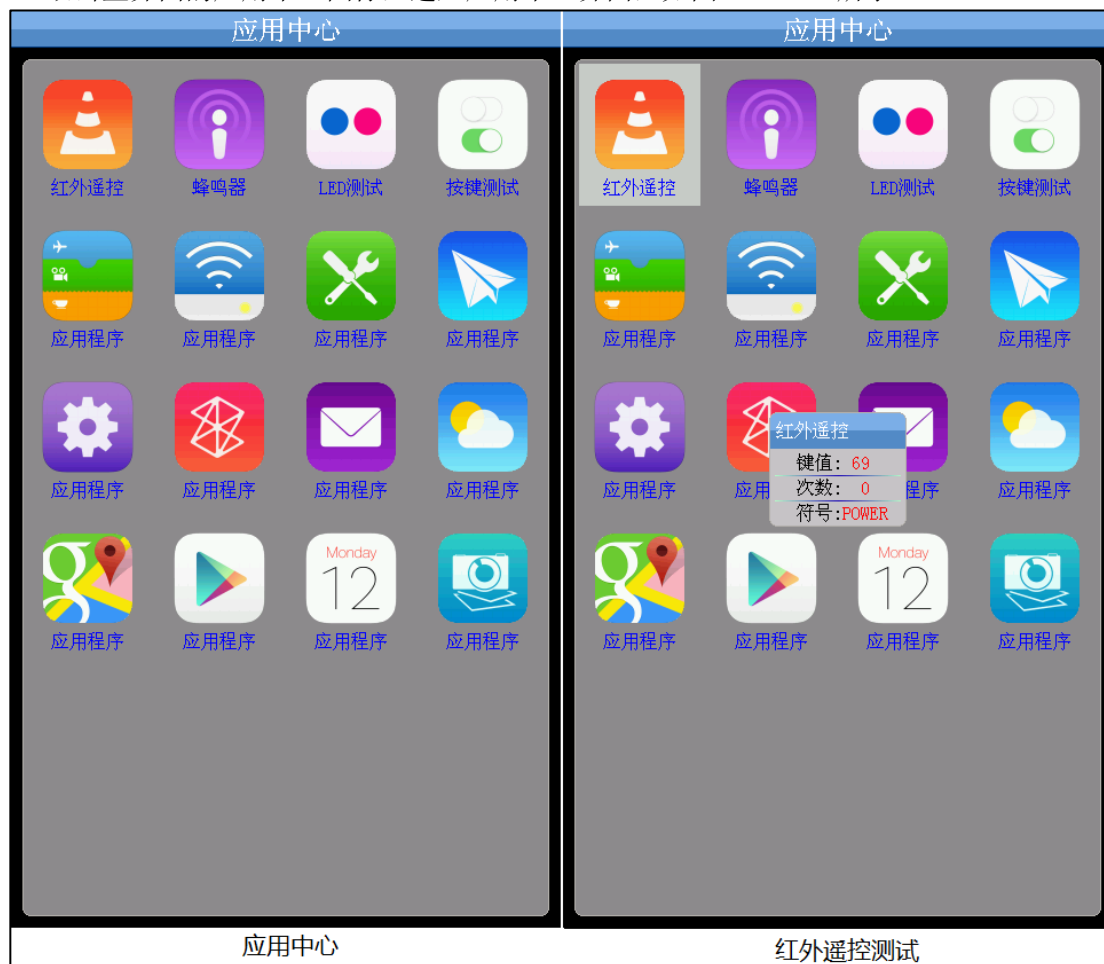


图 66.2.18.1 应用中心和红外遥控测试

左侧图片是我们刚进入应用中心看到的界面，在该界面下总共有 16 个图标，我们仅实现了第一行四个功能：红外遥控、蜂鸣器、LED 测试、按键测试。其他都没有实现，大家可以自由发挥，添加属于自己的东西。

双击第一个图标，会弹出一个红外遥控的小窗口，用于接收红外信号，此时，我们将红外遥控对准开发板的红外接收头，并按下按钮，就可以在红外遥控窗口里面显示键值、按键次数、符号等信息。如图 66.2.18.1 右侧图片所示。

按 TPAD 可以退出红外遥控功能，返回应用中心主界面，然后按第二个图标，即可进入蜂鸣器测试界面。蜂鸣器功能用于测试蜂鸣器，通过触摸屏来控制蜂鸣器响或不响。

进入蜂鸣器测试窗口后，就可以通过点击显示屏的“打开”按钮，让蜂鸣器响。此时“打开”按钮变成“关闭”按钮，之后点击“关闭”按钮则会关闭蜂鸣器，“关闭”按钮又会变成“打开”按钮，如图 66.2.18.2 所示：



图 66.2.18.2 蜂鸣器功能测试

按 TPAD 会退出蜂鸣器测试功能，回到应用中心主界面。然后按第三个图标，即可进入 LED 测试界面。LED 测试功能用于测试板载的 LED 灯，通过触摸屏来控制 LED 灯的亮灭。

进入 LED 测试窗口后，默认 LED 灯全部熄灭，显示屏的两个实心圆显示白色，如图 66.2.18.3 左侧图所示：

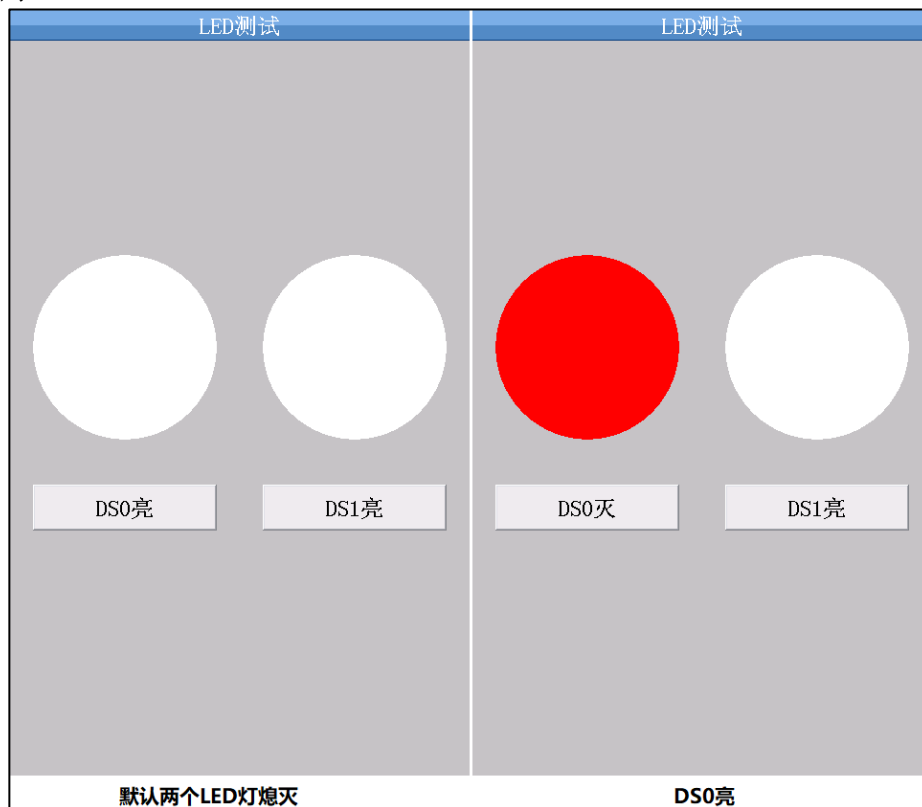


图 66.2.18.3 默认没有按键按下和按下 DS0 亮按键

我们可以通过点击显示屏的“DS0 亮”和“DS1 亮”这 2 个按钮，来控制板载 LED 灯亮。比如点击“DS0 亮”按钮，那么 DS0 对应的实心圆就变成红色，如图 66.2.18.1 右侧图所示，此时 LED0 红灯也会亮。LED1 灯的控制方法类似的。

按 TPAD，则会退出 LED 测试功能，回到应用中心主界面。然后按第四个图标，即可进入按键测试界面。按键测试功能用于测试板载的四个功能按键。

进入按键测试窗口后，此时，如果没有任何按键被按键，默认显示屏上显示四个黄色的实心圆（KEYUP、KEY2、KEY1、KEY0），如图 66.2.18.4 左侧图所示：

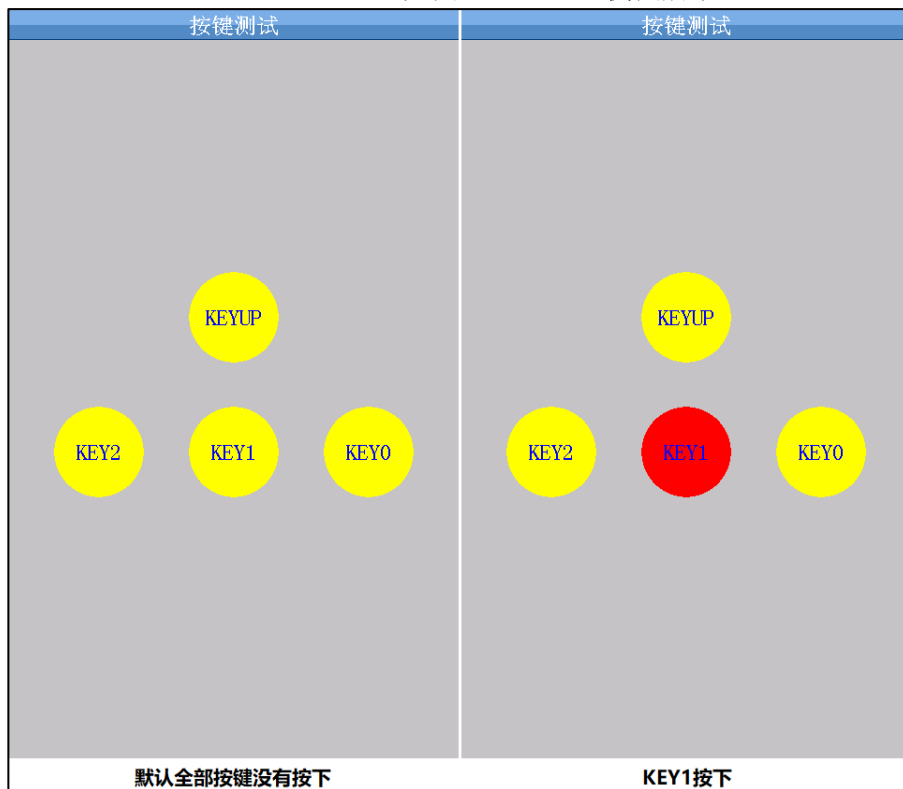


图 66.2.18.4 默认没有按键按下和按下 KEY1 按键

当我们按下其中一个按键，则对应的实心圆会显示红色，比如按下按键 KEY1，显示屏的实心圆 KEY1 变红色，如图 66.2.18.4 所示，此后松开 KEY1 按键，该实心圆又变回黄色。注意：当有两个或两个以上按键被同时按下，只有其中一个有效。

按 TPAD，则会退出按键测试功能，回到应用中心主界面，再按一次 TPAD 就可以返回主界面。

66.2.19 电压表

电压表功能用于测量 0~3.3V 的直流电压源，可以是开发板上的电压源，也可以是外部的，但是电压的范围一定要是 0~3.3V，否则容易烧毁 STM32 芯片。

下面我们一起来测量板载的可调电位器的电压，双击主界面的电压表图标，屏幕会弹出提示对话框，如图 66.2.19.1 所示：

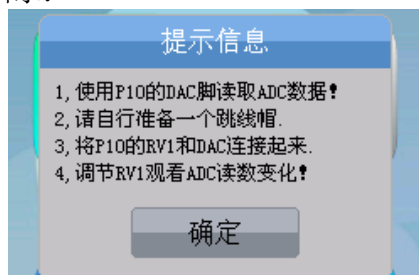


图 66.2.19.1 双击电压表后提示界面

提示信息告诉我们，需要使用短路帽将多功能端口 P10 的 DAC 和 RV1 排针连接。

进入电压表功能后，就可以通过调节可调点位旋钮改变接入 ADC 的电压，从而在屏幕上显示不同的电压值，如图 66.2.19.2 左侧图所示：



图 66.2.19.2 调节电位器得到不同电压和最大允许接入 3.3V 电压

上图中，ADC 值是电压的数字值，Uin 值是经过转换后的实际电压。由于 STM32F103 的 ADC 最大分辨率是 12 位，所以电压的数字值范围是 0~4095。

当我们把可调电压值调至最大，就可以得到 3.3V 的最大输入电压，如图 66.2.19.2 右侧图所示。电压的数字值是 4095，对应的实际电压是 3.3V。

至此，开发板的综合测试实验就介绍完了。其中，参考了不少网友的代码，对这些网友表示衷心的感谢，同时我也希望我们的这个代码，可以让大家有所受益，能开发出更强更好的产品。

综合实验整个代码编译后大小为 452K 左右（-O0 优化后），代码量是非常的大，希望大家慢慢理解，各个攻破，最后祝大家身体健康、学习进步！