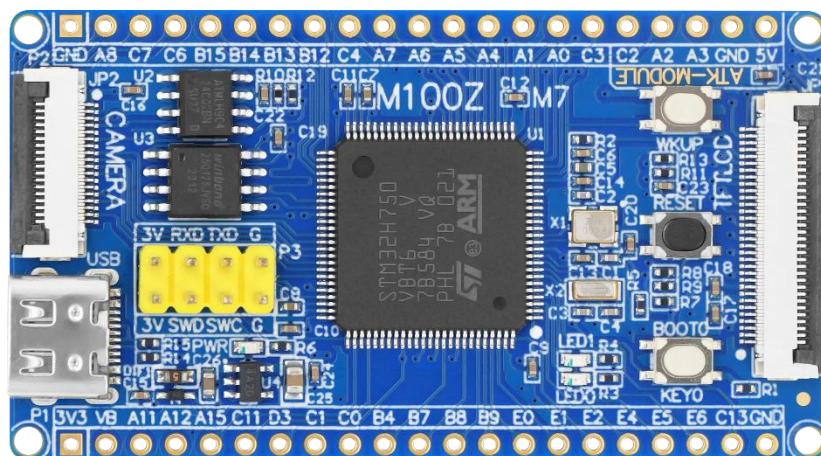


M100Z-M7 最小系统板使用指南

——STM32H750 版 V1.0

-M100Z-M7 最小系统板教程

注：本教程仅适用于 STM32H750 版



型号	版本	说明
M100Z-M3	STM32F103 版	主控芯片使用：STM32F103VET6
M100Z-M3	APM32E103 版	主控芯片使用：APM32E103VET6
M100Z-M4	STM32F407 版	主控芯片使用：STM32F407VET6
M100Z-M4	APM32F407 版	主控芯片使用：APM32F407VGT6
M100Z-M7	STM32H750 版	主控芯片使用：STM32H750VBT6

修订历史:

版本	日期	修改内容
V1.0	2023/05/25	第一次发布



正点原子公司名称 : 广州市星翼电子科技有限公司

原子哥在线教学平台 : www.yuanzige.com

开源电子网 / 论坛 : www.openedv.com

正点原子官方网站 : www.alientek.com

正点原子淘宝店铺 : <https://openedv.taobao.com>

正点原子 B 站视频 : <https://space.bilibili.com/394620890>

电话: 020-38271790 传真: 020-36773971

请下载原子哥 APP, 数千讲视频免费学习, 更快更流畅。

请关注正点原子公众号, 资料发布更新我们会通知。



扫码下载“原子哥”APP



扫码关注正点原子公众号

内容简介.....	1
第一篇 基础篇.....	2
第一章 本书学习方法.....	3
1.1 本书学习顺序.....	3
1.2 本书参考资料.....	3
1.3 本书编写规范.....	3
1.4 本书代码规范.....	4
1.5 例程资源说明.....	4
1.6 学习资料查找.....	5
第二章 STM32 简介	8
2.1 初识 STM32	8
2.2 STM32H750 资源简介.....	8
第三章 开发环境搭建.....	10
3.1 常用开发工具简介.....	10
3.2 MDK 安装	10
3.3 仿真器驱动安装.....	13
第四章 STM32 初体验	14
4.1 使用 MDK 编译实验例程	14
4.2 使用 DAP 烧录及调试程序.....	16
4.2.1 使用 DAP 烧录程序.....	17
4.2.2 使用 DAP 调试程序.....	18
4.3 MDK 使用技巧	23
4.3.1 自定义编辑器.....	23
4.3.2 动态语法检测&代码自动补全.....	25
4.3.3 代码编辑技巧.....	26
4.3.4 其他小技巧.....	28
第五章 STM32 基础知识入门	30
5.1 C 语言基础知识复习	30
5.1.1 位操作.....	30
5.1.2 define 宏定义.....	31
5.1.3 ifdef 条件编译	31
5.1.4 extern 外部申明.....	31
5.1.5 typedef 类型别名.....	31
5.1.6 struct 结构体.....	31
5.1.7 指针.....	32
5.2 寄存器基础知识.....	32
5.3 STM32H750 系统架构.....	33
5.3.1 Cortex M7 内核 & 芯片	33
5.3.2 STM32 系统架构	33
5.3.3 存储器映射	36
5.3.4 存储块功能介绍.....	37
第六章 认识 HAL 库	43

6.1 初识 STM32 HAL 库	43
6.1.1 CMSIS 标准.....	43
6.1.2 HAL 库简介	44
6.1.3 HAL 库能做什么	46
6.2 HAL 库驱动包	46
6.2.1 如何获取 HAL 库固件包.....	47
6.2.2 STM32Cube 固件包分析	48
6.2.3 CMSIS 文件夹关键文件.....	50
6.2.4 stdint.h 简介.....	52
6.3 HAL 库框架结构	52
6.3.1 HAL 库文件夹结构	52
6.3.2 HAL 库文件介绍	53
6.4 如何使用 HAL 库	56
6.4.1 学会用 HAL 库组织开发工具链.....	56
6.4.2 HAL 库的用户配置文件.....	57
6.4.3 stm32h7xx_hal.c 文件	59
6.4.4 HAL 库中断处理	64
6.4.5 正点原子对 HAL 库用法的个性化修改.....	65
6.5 HAL 库使用注意事项	65
第七章 HAL 库版本 MDK 工程创建	66
7.1 HAL 库版本 MDK 工程创建	66
7.1.1 构建工程文件夹.....	66
7.1.2 添加文件到工程目录.....	67
7.1.3 创建 MDK 工程	68
7.1.4 添加文件.....	70
7.1.5 魔术棒设置.....	76
7.1.6 添加 main.c， 并编写代码.....	81
7.2 下载验证.....	83
第八章 STM32 时钟系统	85
8.1 认识时钟树.....	85
8.1.1 时钟源.....	86
8.1.2 锁相环 PLL.....	87
8.1.3 系统时钟 SYSCLK	88
8.1.4 外设内核时钟.....	89
8.1.5 MCO 时钟输出.....	91
8.2 配置系统主频.....	91
8.2.1 STM32H7 时钟系统配置.....	91
8.2.2 STM32H7 时钟使能和配置.....	100
第九章 SYSTEM 文件夹介绍	102
9.1 deley 文件夹代码介绍	102
9.1.1 操作系统支持宏定义及相关函数.....	103
9.1.2 delay_init 函数.....	105
9.1.3 delay_us 函数	106
9.1.4 delay_ms 函数	107
9.1.5 HAL 库延时函数 HAL_Delay.....	108
9.2 sys 文件夹代码介绍.....	109
9.2.1 Cache 使能函数.....	109

9.2.2 QSPI_Enable_Memmapmode 函数.....	110
9.3 usart 文件夹代码介绍	112
9.3.1 printf 函数支持.....	112
第二篇 入门篇.....	115
第十章 跑马灯实验.....	116
10.1 硬件设计.....	116
10.1.1. 例程功能.....	116
10.1.2. 硬件资源.....	116
10.1.3. 原理图.....	116
10.2 程序设计.....	116
10.2.1 HAL 库的 GPIO 驱动	116
10.2.2 LED 驱动.....	118
10.2.3 实验应用代码.....	120
10.3 下载验证.....	120
第十一章 按键输入实验.....	121
11.1 硬件设计.....	121
11.1.1 例程功能.....	121
11.1.2 硬件资源.....	121
11.1.3 原理图.....	121
11.2 程序设计.....	122
11.2.1 HAL 库的 GPIO 驱动	122
11.2.2 按键驱动.....	122
11.2.3 实验应用代码.....	124
11.3 下载验证.....	125
第十二章 外部中断实验.....	126
12.1 硬件设计.....	126
12.1.1 例程功能.....	126
12.1.2 硬件资源.....	126
12.1.3 原理图.....	126
12.2 程序设计.....	127
12.2.1 HAL 库的 GPIO 驱动	127
12.2.2 外部中断驱动.....	128
12.2.3 实验应用代码.....	129
12.3 下载验证.....	130
第十三章 串口通信实验.....	131
13.1 硬件设计.....	131
13.1.1 例程功能.....	131
13.1.2 硬件资源.....	131
13.1.3 原理图.....	131
13.2 程序设计.....	131
13.2.1 HAL 库的 GPIO 驱动	131
13.2.2 HAL 库的 UART 驱动.....	132
13.2.3 串口通讯驱动.....	133
13.2.4 实验应用代码.....	135
13.3 下载验证.....	135

第十四章 独立看门狗实验.....	136
14.1 硬件设计.....	136
14.1.1 例程功能.....	136
14.1.2 硬件资源.....	136
14.1.3 原理图.....	136
14.2 程序设计.....	136
14.2.1 HAL 库的 IWDG 驱动.....	136
14.2.2 看门狗驱动.....	137
14.2.3 实验应用代码.....	138
14.3 下载验证.....	138
第十五章 窗口看门狗实验.....	139
15.1 硬件设计.....	139
15.1.1 例程功能.....	139
15.1.2 硬件资源.....	139
15.1.3 原理图.....	139
15.2 程序设计.....	139
15.2.1 HAL 库的 WWDG 驱动	139
15.2.2 看门狗驱动.....	140
15.2.3 实验应用代码.....	141
15.3 下载验证.....	141
第十六章 基本定时器中断实验.....	142
16.1 硬件设计.....	142
16.1.1 例程功能.....	142
16.1.2 硬件资源.....	142
16.1.3 原理图.....	142
16.2 程序设计.....	142
16.2.1 HAL 库的 TIM 驱动	142
16.2.2 基本定时器驱动.....	143
16.2.3 实验应用代码.....	144
16.3 下载验证.....	144
第十七章 通用定时器中断实验.....	145
17.1 硬件设计.....	145
17.1.1 例程功能.....	145
17.1.2 硬件资源.....	145
17.1.3 原理图.....	145
17.2 程序设计.....	145
17.2.1 HAL 库的 TIM 驱动	145
17.2.2 通用定时器驱动.....	145
17.2.3 实验应用代码.....	145
17.3 下载验证.....	146
第十八章 通用定时器 PWM 输出实验	147
18.1 硬件设计.....	147
18.1.1 例程功能.....	147
18.1.2 硬件资源.....	147
18.1.3 原理图.....	147
18.2 程序设计.....	147

18.2.1 HAL 库的 TIM 驱动	147
18.2.2 通用定时器驱动.....	149
18.2.3 实验应用代码.....	150
18.3 下载验证.....	151
第十九章 通用定时器输入捕获实验.....	152
19.1 硬件设计.....	152
19.1.1 例程功能.....	152
19.1.2 硬件资源.....	152
19.1.3 原理图.....	152
19.2 程序设计.....	152
19.2.1 HAL 库的 TIM 驱动	152
19.2.2 通用定时器驱动.....	154
19.2.3 实验应用代码.....	155
19.3 下载验证.....	156
第二十章 通用定时器脉冲计数实验.....	157
20.1 硬件设计.....	157
20.1.1 例程功能.....	157
20.1.2 硬件资源.....	157
20.1.3 原理图.....	157
20.2 程序设计.....	157
20.2.1 HAL 库的 TIM 驱动	157
20.2.2 通用定时器驱动.....	158
20.2.3 实验应用代码.....	160
20.3 下载验证.....	161
第二十一章 高级定时器输出指定个数 PWM 实验	162
21.1 硬件设计.....	162
21.1.1 例程功能.....	162
21.1.2 硬件资源.....	162
21.1.3 原理图.....	162
21.2 程序设计.....	162
21.2.1 HAL 库的 TIM 驱动	162
21.2.2 高级定时器驱动.....	162
21.2.3 实验应用代码.....	165
21.3 下载验证.....	165
第二十二章 高级定时器输出比较模式实验.....	166
22.1 硬件设计.....	166
22.1.1 例程功能.....	166
22.1.2 硬件资源.....	166
22.1.3 原理图.....	166
22.2 程序设计.....	166
22.2.1 HAL 库的 TIM 驱动	166
22.2.2 高级定时器驱动.....	168
22.2.3 实验应用代码.....	170
22.3 下载验证.....	171
第二十三章 高级定时器互补输出带死区控制实验.....	172

23.1 硬件设计.....	172
23.1.1 例程功能.....	172
23.1.2 硬件资源.....	172
23.1.3 原理图.....	172
23.2 程序设计.....	172
23.2.1 HAL 库的 TIM 驱动	172
23.2.2 高级定时器驱动.....	174
23.2.3 实验应用代码.....	176
23.3 下载验证.....	177
第二十四章 高级定时器 PWM 输入模式实验	178
24.1 硬件设计.....	178
24.1.1 例程功能.....	178
24.1.2 硬件资源.....	178
24.1.3 原理图.....	178
24.2 程序设计.....	178
24.2.1 HAL 库的 TIM 驱动	178
24.2.2 高级定时器驱动.....	179
24.2.3 实验应用代码.....	181
24.3 下载验证.....	181
第二十五章 OLED 实验.....	182
25.1 硬件设计.....	182
25.1.1 例程功能.....	182
25.1.2 硬件资源.....	182
25.1.3 原理图.....	182
25.2 程序设计.....	182
25.2.1 HAL 库的 GPIO 驱动	182
25.2.2 OLED 驱动	183
25.2.3 实验应用代码.....	186
25.3 下载验证.....	187
第二十六章 内存保护（MPU）实验	189
26.1 硬件设计.....	189
26.1.1 例程功能.....	189
26.1.2 硬件资源.....	189
26.1.3 原理图.....	189
26.2 程序设计.....	189
26.2.1 MPU 的 HAL 库驱动	189
26.2.2 MPU 配置步骤	190
26.2.3 程序解析.....	191
26.2.4 实验应用代码.....	193
26.3 下载验证.....	194
第二十七章 TFTLCD（MCU 屏）实验.....	196
27.1 硬件设计.....	196
27.1.1 例程功能.....	196
27.1.2 硬件资源.....	196
27.1.3 原理图.....	196
27.2 程序设计.....	197

27.2.1 HAL 库的 FSMC 驱动	197
27.2.2 TFTLCD 驱动	198
27.2.3 实验应用代码	203
27.3 下载验证	204
第二十八章 USMART 调试实验	205
28.1 硬件设计	205
28.1.1 例程功能	205
28.1.2 硬件资源	205
28.1.3 原理图	205
28.2 程序设计	205
28.2.1 USMART 组件	205
28.2.2 实验应用代码	207
28.3 下载验证	208
第二十九章 RTC 实时时钟实验	209
29.1 硬件设计	209
29.1.1 例程功能	209
29.1.2 硬件资源	209
29.1.3 原理图	209
29.2 程序设计	209
29.2.1 HAL 库的 PWR 驱动	209
29.2.2 HAL 库的 RTC 驱动	210
29.2.3 RTC 驱动	216
29.2.4 RTC 驱动	219
29.2.5 实验应用代码	222
29.3 下载验证	223
第三十章 硬件随机数实验	224
30.1 硬件设计	224
30.1.1 例程功能	224
30.1.2 硬件资源	224
30.1.3 原理图	224
30.2 程序设计	224
30.2.1 HAL 库的 RNG 驱动	224
30.2.2 RNG 驱动	225
30.2.3 实验应用代码	226
30.3 下载验证	227
第三十一章 PVD 电压监控实验	228
31.1 硬件设计	228
31.1.1 例程功能	228
31.1.2 硬件资源	228
31.1.3 原理图	228
31.2 程序设计	228
31.2.1 HAL 库的 PWR 驱动	228
31.2.2 PWR 驱动	229
31.2.3 实验应用代码	230
31.3 下载验证	231
第三十二章 睡眠模式实验	232

32.1 硬件设计.....	232
32.1.1 例程功能.....	232
32.1.2 硬件资源.....	232
32.1.3 原理图.....	232
32.2 程序设计.....	232
32.2.1 HAL 库的 PWR 驱动.....	232
32.2.2 PWR 驱动.....	233
32.2.3 实验应用代码.....	234
32.3 下载验证.....	235
第三十三章 停止模式实验.....	236
33.1 硬件设计.....	236
33.1.1 例程功能.....	236
33.1.2 硬件资源.....	236
33.1.3 原理图.....	236
33.2 程序设计.....	236
33.2.1 HAL 库的 PWR 驱动.....	236
33.2.2 PWR 驱动.....	237
33.2.3 实验应用代码.....	237
33.3 下载验证.....	238
第三十四章 待机模式实验.....	239
34.1 硬件设计.....	239
34.1.1 例程功能.....	239
34.1.2 硬件资源.....	239
34.1.3 原理图.....	239
34.2 程序设计.....	239
34.2.1 HAL 库的 PWR 驱动.....	239
34.2.2 PWR 驱动.....	240
34.2.3 实验应用代码.....	241
34.3 下载验证.....	241
第三十五章 DMA 实验	242
35.1 硬件设计.....	242
35.1.1 例程功能.....	242
35.1.2 硬件资源.....	242
35.1.3 原理图.....	242
35.2 程序设计.....	242
35.2.1 HAL 库的 DMA 驱动	242
35.2.2 DMA 驱动	244
35.2.3 实验应用代码.....	244
35.3 下载验证.....	246
第三十六章 单通道 ADC 采集实验	247
36.1 硬件设计.....	247
36.1.1 例程功能.....	247
36.1.2 硬件资源.....	247
36.1.3 原理图.....	247
36.2 程序设计.....	247
36.2.1 HAL 库的 ADC 驱动	247

36.2.2 ADC 驱动	250
36.2.3 实验应用代码.....	252
36.3 下载验证.....	253
第三十七章 单通道 ADC 采集（DMA 读取）实验	254
37.1 硬件设计.....	254
37.1.1 例程功能.....	254
37.1.2 硬件资源.....	254
37.1.3 原理图.....	254
37.2 程序设计.....	254
37.2.1 HAL 库的 ADC 驱动	254
37.2.2 ADC 驱动	255
37.2.3 实验应用代码.....	257
37.3 下载验证.....	258
第三十八章 多通道 ADC 采集（DMA 读取）实验	259
38.1 硬件设计.....	259
38.1.1 例程功能.....	259
38.1.2 硬件资源.....	259
38.1.3 原理图.....	259
38.2 程序设计.....	259
38.2.1 HAL 库的 ADC 驱动	259
38.2.2 ADC 驱动	259
38.2.3 实验应用代码.....	262
38.3 下载验证.....	263
第三十九章 内部温度传感器实验.....	264
39.1 硬件设计.....	264
39.1.1 例程功能.....	264
39.1.2 硬件资源.....	264
39.1.3 原理图.....	264
39.2 程序设计.....	264
39.2.1 HAL 库的 ADC 驱动	264
39.2.2 ADC 驱动	264
39.2.3 实验应用代码.....	265
39.3 下载验证.....	266
第四十章 DAC 输出实验	267
40.1 硬件设计.....	267
40.1.1 例程功能.....	267
40.1.2 硬件资源.....	267
40.1.3 原理图.....	267
40.2 程序设计.....	267
40.2.1 HAL 库的 DAC 驱动	267
40.2.2 DAC 驱动	269
40.2.3 实验应用代码.....	271
40.3 下载验证.....	272
第四十一章 DAC 输出三角波实验	273
41.1 硬件设计.....	273

41.1.1 例程功能.....	273
41.1.2 硬件资源.....	273
41.1.3 原理图.....	273
41.2 程序设计.....	273
41.2.1 HAL 库的 DAC 驱动	273
41.2.2 DAC 驱动	273
41.2.3 实验应用代码.....	274
41.3 下载验证.....	275
第四十二章 DAC 输出正弦波实验	276
42.1 硬件设计.....	276
42.1.1 例程功能.....	276
42.1.2 硬件资源.....	276
42.1.3 原理图.....	276
42.2 程序设计.....	276
42.2.1 HAL 库的 TIM 驱动	276
42.2.2 HAL 库的 DAC 驱动	277
42.2.3 DAC 驱动	278
42.2.4 实验应用代码.....	279
42.3 下载验证.....	281
第四十三章 IIC 实验	282
43.1 硬件设计.....	282
43.1.1 例程功能.....	282
43.1.2 硬件资源.....	282
43.1.3 原理图.....	282
43.2 程序设计.....	282
43.2.1 HAL 库的 GPIO 驱动	282
43.2.2 IIC 驱动	283
43.2.3 EEPROM 驱动	284
43.2.4 实验应用代码.....	284
43.3 下载验证.....	285
第四十四章 SPI 实验	286
44.1 硬件设计.....	286
44.1.1 例程功能.....	286
44.1.2 硬件资源.....	286
44.1.3 原理图.....	286
44.2 程序设计.....	286
44.2.1 HAL 库的 SPI 驱动.....	286
44.2.2 QSPI 驱动	288
44.2.3 NOR Flash 驱动.....	291
44.2.4 实验应用代码.....	292
44.3 下载验证.....	294
第四十五章 触摸屏实验.....	295
45.1 硬件设计.....	295
45.1.1 例程功能.....	295
45.1.2 硬件资源.....	295
45.1.3 原理图.....	295

45.2 程序设计.....	295
45.2.1 触摸屏驱动.....	295
45.2.2 实验应用代码.....	296
45.3 下载验证.....	299
第四十六章 FLASH 模拟 EEPROM 实验.....	300
46.1 硬件设计.....	300
46.1.1 例程功能.....	300
46.1.2 硬件资源.....	300
46.1.3 原理图.....	300
46.2 程序设计.....	300
46.2.1 HAL 库的 Flash 驱动.....	300
46.2.2 Flash 驱动.....	303
46.2.3 实验应用代码.....	307
46.3 下载验证.....	308
第四十七章 摄像头实验.....	309
47.1 硬件设计.....	309
47.1.1 例程功能.....	309
47.1.2 硬件资源.....	309
47.1.3 原理图.....	309
47.2 程序设计.....	310
47.2.1 HAL 库的 DCMI 驱动.....	310
47.2.2 DCMI 驱动代码	311
47.2.3 SCCB 驱动	315
47.2.4 OV5640 驱动.....	315
47.2.5 实验应用代码.....	315
47.3 下载验证.....	316
第三篇 提高篇.....	318
第四十八章 内存管理实验.....	319
48.1 硬件设计.....	319
48.1.1 例程功能.....	319
48.1.2 硬件资源.....	319
48.1.3 原理图.....	319
48.2 程序设计.....	319
48.2.1 内存管理库的使用.....	319
48.2.2 实验应用代码.....	320
48.3 下载验证.....	322
第四十九章 SD 卡实验.....	323
49.1 硬件设计.....	323
49.1.1 例程功能.....	323
49.1.2 硬件资源.....	323
49.1.3 原理图.....	323
49.2 程序设计.....	324
49.2.1 SD 卡驱动.....	324
49.2.2 实验应用代码.....	324
49.3 下载验证.....	327

第五十章 FATFS 实验	328
50.1 硬件设计.....	328
50.1.1 例程功能.....	328
50.1.2 硬件资源.....	328
50.1.3 原理图.....	328
50.2 程序设计.....	328
50.2.1 FATFS 的使用	328
50.2.2 实验应用代码.....	329
50.3 下载验证.....	331
第五十一章 汉字显示实验.....	332
51.1 硬件设计.....	332
51.1.1 例程功能.....	332
51.1.2 硬件资源.....	332
51.1.3 原理图.....	332
51.2 程序设计.....	332
51.2.1 字库管理库的使用.....	332
51.2.2 实验应用代码.....	333
51.3 下载验证.....	335
第五十二章 图片显示实验.....	336
52.1 硬件设计.....	336
52.1.1 例程功能.....	336
52.1.2 硬件资源.....	336
52.1.3 原理图.....	336
52.2 程序设计.....	336
52.2.1 图片解码库的使用.....	336
52.2.2 实验应用代码.....	337
52.3 下载验证.....	340
第五十三章 硬件 JPEG 解码实验.....	341
53.1 硬件设计.....	341
53.1.1 例程功能.....	341
53.1.2 硬件资源.....	341
53.1.3 原理图.....	341
53.2 程序设计.....	342
53.2.1 硬件 JPEG 解码 JPG/JPEG 的简要步骤.....	342
53.2.2 程序解析.....	342
53.3 下载验证.....	361
第五十四章 照相机实验.....	362
54.1 硬件设计.....	362
54.1.1 例程功能.....	362
54.1.2 硬件资源.....	362
54.1.3 原理图.....	362
54.2 程序设计.....	363
54.2.1 实验应用代码.....	363
54.3 下载验证.....	367
第五十五章 视频播放器实验.....	368

55.1 硬件设计.....	368
55.1.1 例程功能.....	368
55.1.2 硬件资源.....	368
55.1.3 原理图.....	368
55.2 程序设计.....	368
55.2.1 libjpeg 库的使用.....	368
55.2.2 AVI 文件解析	369
55.2.3 实验应用代码.....	369
55.3 下载验证.....	374
第五十六章 FPU 测试（Julia 分形）实验	375
56.1 硬件设计.....	375
56.1.1 例程功能.....	375
56.1.2 硬件资源.....	375
56.1.3 原理图.....	375
56.2 程序设计.....	375
56.2.1 实验应用代码.....	375
56.2.3 下载验证.....	377
第五十七章 DSP BasicMath 实验	378
57.1 硬件设计.....	378
57.1.1 例程功能.....	378
57.1.2 硬件资源.....	378
57.1.3 原理图.....	378
57.2 程序设计.....	378
57.2.1 DPS 库的使用	378
57.2.2 实验应用代码.....	379
57.3 下载验证.....	380
第五十八章 DSP FFT 实验.....	381
58.1 硬件设计.....	381
58.1.1 例程功能.....	381
58.1.2 硬件资源.....	381
58.1.3 原理图.....	381
58.2 程序设计.....	381
58.2.1 实验应用代码.....	381
58.3 下载验证.....	382
第五十九章 手写识别实验.....	383
59.1 硬件设计.....	383
59.1.1 例程功能.....	383
59.1.2 硬件资源.....	383
59.1.3 原理图.....	383
59.2 程序设计.....	383
59.2.1 手写识别库的使用.....	383
59.2.2 实验应用代码.....	385
59.3 下载验证.....	388
第六十章 T9 拼音输入法实验	389
60.1 硬件设计.....	389

60.1.1 例程功能.....	389
60.1.2 硬件资源.....	389
60.1.3 原理图.....	389
60.2 程序设计.....	389
60.2.1 T9 拼音输入法库的使用	389
60.2.2 实验应用代码.....	390
60.3 下载验证.....	393
第六十一章 串口 IAP 实验	394
61.1 硬件设计.....	394
61.1.1 例程功能.....	394
61.1.2 硬件资源.....	394
61.1.3 原理图.....	394
61.2 程序设计.....	394
61.2.1 IAP 的实现	394
61.2.2 APP 工程修改.....	395
61.2.3 实验应用代码.....	396
61.3 下载验证.....	399
第六十二章 USB 读卡器（Slave）实验	400
62.1 硬件设计.....	400
62.1.1 例程功能.....	400
62.1.2 硬件资源.....	400
62.1.3 原理图.....	400
62.2 程序设计.....	401
62.2.1 ST 的 USB 设备驱动库	401
62.2.2 实验应用代码.....	401
62.3 下载验证.....	404
第六十三章 USB 虚拟串口（Slave）实验	405
63.1 硬件设计.....	405
63.1.1 例程功能.....	405
63.1.2 硬件资源.....	405
63.1.3 原理图.....	405
63.2 程序设计.....	405
63.2.1 ST 的 USB 设备驱动库	405
63.2.2 实验应用代码.....	405
63.3 下载验证.....	407
第六十四章 FreeRTOS 移植实验.....	408
64.1 硬件设计.....	408
64.1.1 例程功能.....	408
64.1.2 硬件资源.....	408
64.1.3 原理图.....	408
64.2 程序设计.....	408
64.2.1 FreeRTOS 的移植	408
64.2.2 实验应用代码.....	408
64.3 下载验证.....	410

内容简介

本书将由浅入深，带领大家学习 STM32H750 的各个功能，为您开启 STM32H750 的学习之旅。

本书总共分为三篇：

- 1，基础篇，主要介绍 STM32H750 的基础入门知识，包括介绍 STM32、搭建开发环境、创建模板工程、介绍 STM32H750 的时钟系统以及 SYSTEM 文件夹等基础知识，必须好好学习并掌握；
- 2，入门篇，主要介绍 STM32H750 中常用片上外设的使用，包括 GPIO、TIM、ADC、DAC、DMA、FSMC 等，必须好好学习并掌握；
- 3，提高篇，主要介绍 STM32H750 中比较复杂的片上外设以及一些高级例程，包括 SDIO、USB、内存管理、文件系统、图片解码、RTOS 等，这些片上外设和例程的难度相对较高，应沉下心来，戒骄戒躁、循序渐进地学习，对于以上难度较高的软件库，仅需学习如何使用这些软件库，并不需要深入地了解软件库的实现原理和运行机制，例如文件系统、图片解码、USB 协议栈、RTOS 等。

本书为正点原子 M100Z-M7 最小系统板 STM32H750 版的配套教程，在板卡配套的资料包中提供了大量与本板卡相关的资料，其中就包括硬件原理图以及本书中所有实验例程的源代码，这些代码都作了详细的注释，并且都经过了测试，不会出现任何错误或警告，且每一个实验例程都提供了实现编译好的 Hex 文件，仅需将其通过仿真器烧录到板卡即可观察到实验现象，体验实验过程。

本书并没有详细介绍 M100Z-M7 最小系统板 STM32H750 版的硬件资源，而是单独提供了一份相关的文档，请参考：《M100Z-M7 最小系统板硬件参考手册——STM32H750 版.pdf》。

本书不仅非常适合广大学生和电子爱好者学习 STM32H750，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。本篇将详细介绍 STM32H750 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, 认识 HAL 库
- 7, 新建 HAL 库版本 MDK 工程
- 8, STM32H750 时钟系统
- 11, SYSTEM 文件夹介绍

第一章 本书学习方法

为了让读者能够更好地学习和使用本书，本章将介绍本书的学习方法。

本章分为如下几个小节：

- 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

1.1 本书学习顺序

为了让读者更好地学习和使用本书，我们做了以下几点考虑：

- 1, 坚持循序渐进的思路讲解，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；
- 3, 将板卡硬件资源介绍独立成一个文档（《M100Z-M7 最小系统板硬件参考手册——STM32H750 版.pdf》）。

因此，读者在学习本书的时候，我们建议：先通读一遍《M100Z-M7 最小系统板硬件参考手册——STM32H750 版.pdf》，对板卡的硬件资源有个大概的了解，然后从本书的基础篇开始，再到入门篇，最后是提高篇，循序渐进，逐一攻克。

对于初学者，更是要按照以上建议的学习路线进行学习，不要跳跃式学习，因为本书中的知识是环环相扣的，如果没有掌握前面的知识，就去学习后面的知识，就会学的非常吃力。

对于已经有了一定单片机基础的读者，就可以跳跃式地学习，学习效率，当然了，若是遇到不懂的知识点，也得查阅前面的知识点进行巩固。

1.2 本书参考资料

本书主要参考的资料有一下两份文档：

《RM0433: STM32H742, STM32H743/753, and STM32H750 value line advanced Arm®-based 32-bit MCUs》

《Arm® Cortex®-M7 Processor Technical Reference Manual》

前者是 ST 官方针对 STM32H750 系列 MCU 提供的参考手册，该参考手册向程序开发人员提供了如何使用 MCU 系统架构、存储器和外设所涉及的全部信息。

后者是 ARM 针对 ARM Cortex-M7 内核提供的技术参考手册，该技术参考手册包含了对 ARM Cortex-M7 内核及其使用的指令集、寄存器、内存映射、浮点和跟踪调试等的支持文档。

以上提及的两份文档也是读者在学习本书的过程中必不可少的参考资料，读者可以在 A 盘 → 8, STM32 参考资料中找到这两份文档。

1.3 本书编写规范

本书通过数十个实验例程为读者详细介绍了 STM32 几乎所有的功能和外设，按照难易程度以及知识结构，本书分为三大篇章：基础篇、入门篇和提高篇。

基础篇，共九章，主要是一些基础知识的介绍、包括开发环境搭建、HAL 库介绍、创建 MDK 工程、时钟系统介绍、SYSTEM 文件夹介绍等，这些章节在结构上没有共性，但相互有关联，即：必须先学习前面的知识，才能更好地学习后面的知识。

入门篇和提高篇，共 55 章，介绍了 STM32H750 的绝大部分外设及其驱动代码，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、T9 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，并且这些章节在结构上比较有

共性，一般分为三个部分，如下：

- 1, 硬件设计
- 2, 程序设计
- 3, 下载验证

硬件设计，包括具体章节实验例程实现的功能说明、使用到的硬件资源及其相关的硬件原理图，从而让读者清楚具体章节的实验例程要做什么？用那些硬件资源来做？这些硬件资源是如何进行连接的？便于在程序设计时编写驱动代码和应用代码。

程序设计，一般包括：驱动介绍、配置步骤、关键代码解析、main 函数讲解等及部分，一点一点地介绍程序代码是怎么来的和注意事项等，从而让读者掌握整个程序代码。

下载验证，属于实践环节，在程序设计完成之后，下载并验证设计的程序是否能按照预期工作，形成一个闭环的过程。

1.4 本书代码规范

为了提高读者编写代码的质量，本书对代码风格进行了统一，详细的代码规范说明文档，请参考 A 盘→1，入门资料→【正点原子】嵌入式单片机 C 代码规范与风格.pdf，对于初学者务必好好地学习一下这份文档。

下面总结几个代码编写规范的关键点：

- 1, 所有函数、变量名称，非特殊情况，一般使用小写字母；
- 2, 注释使用 Doxygen 风格，除屏蔽外，一律使用 “/* */” 的方式进行注释；
- 3, 代码统一使用 4 个空格进行缩进；
- 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, 全局变量的命名一般使用 “g_” 开头，全局指针变量的命名使用 “p_” 开头；
- 7, “if”、“for”、“while”、“do”、“case”、“switch”、“default” 等语句单独占一行，并且无论其有多少行执行语句，都加上 “{}”。

1.5 例程资源说明

M100Z-M7 最小系统板 STM32H750 版提供的标准例程多达 58 个，这些标准例程均是基于 ST 提供的 HAL 库进行编写的。提供的例程基本都是原创，并且拥有非常详细的注释，代码风格统一、内容循序渐进，非常适合初学者入门。

M100Z-M7 最小系统板 STM32H750 版的例程列表如下表所示：

编号	实验名字	编号	实验名字
1	跑马灯实验	19-3	单通道 ADC 过采样（26 位分辨
2	按键输入实验	19-4	内部温度传感器实验
3	外部中断实验	20	DAC 输出实验
4	串口通信实验	21	DAC 输出三角波实验
5	独立看门狗实验	22-1	DAC 输出正弦波实验
6	窗口看门狗实验	22-2	IIC 实验
7	基本定时器中断实验	22-3	QSPI 实验
8-1	通用定时器中断实验	23	触摸屏实验
9-1	通用定时器 PWM 输出实验	24	FLASH 模拟 EEPROM 实验
9-2	通用定时器输入捕获实验	25	摄像头实验
9-3	通用定时器脉冲计数实验	26	内存管理实验
9-4	高级定时器输出指定个数 PWM 实验	27	SD 卡实验
10-1	高级定时器输出比较模式实验	28	FATFS 实验

10-2	高级定时器互补输出带死区控制实验	29	汉字显示实验
10-3	高级定时器 PWM 输入模式实验	30	图片显示实验
10-4	OLED 实验	31	硬件 JPEG 解码实验
11	内存保护（MPU）实验	32	照相机实验
12	TFTLCD（MCU 屏）实验	33	视频播放器实验
13	USMART 调试实验	34	FPU 测试(Julia 分形)实验_开启
14	RTC 实验	35	FPU 测试(Julia 分形)实验_关闭
15	硬件随机数实验	36	DSP BasicMath 测试实验
16	PVD 电压监控实验	37	DSP FFT 测试实验
17-1	睡眠模式实验	38	手写识别实验
17-2	停止模式实验	39	T9 拼音输入法实验
17-3	待机模式实验	40-1	串口 IAP 实验
17-4	DMA 实验	40-2	USB 读卡器(Slave)实验
18	单通道 ADC 采集实验	41-1	USB 虚拟串口(Slave)实验
19-1	单通道 ADC 采集（DMA 读取）实验	41-2	FreeRTOS 移植实验
19-2	多通道 ADC 采集（DMA 读取）实验	42	综合测试实验

表 1.5.1 M100Z-M7 最小系统板 STM32H750 版例程表

从上表可以看出，正点原子 M100Z-M7 最小系统板 STM32H750 版的例程基本上涵盖了 STM32H750VBT6 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 M100Z-M7 最小系统板 STM32H750 版是非常适合初学者的。当然，对于想深入了解 STM32H750 内部资源的朋友，正点原子 M100Z-M7 最小系统板 STM32H750 版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 官方资料

ST 官方 (<https://www.st.com/>) 在其官网上提供了 STM32 芯片相关的 IC 文档和软件，索引方式如下图所示（若后续 ST 官网更新，索引方式可能会有差别）：

The screenshot shows the STMicroelectronics website's product search interface. The top navigation bar includes links for Products, Tools & Software, Applications, Solutions, and STM32 Developer Zone. A sidebar on the left lists various product categories such as Automotive Infotainment and Telematics, Automotive Logic ICs, Automotive Microcontrollers, Clocks and Timers, Data Converters, Digital Set-Top Box ICs, Diodes and Rectifiers, Imaging and Photonics Solutions, Interfaces and Transceivers, Memories, MEMS and Sensors, and Microcontrollers & Microprocessors. The main content area is titled 'Microcontrollers & Microprocessors' and contains sections for 'Legacy MCUs' (ST10 16-bit MCUs), 'STM32 32-bit Arm Cortex MCUs' (STM32 High Performance MCUs, STM32 Mainstream MCUs, STM32 Ultra Low Power MCUs, STM32 Wireless MCUs), 'STM32 Arm Cortex MPUs' (STM32MP1 Series), and 'STM8 8-bit MCUs' (STM8AF Series, STM8L Series, STM8S Series, STM8T Series). A red box highlights the 'STM32 32-bit Arm Cortex MCUs' section.

图 1.6.1 ST 官网微控制器芯片资料索引

随后根据芯片类型点击不同的标签，例如：STM32F1 系列对应“STM32 Mainstream MCUs”而 STM32F4、STM32F7、STM32H5 和 STM32H7 等则对应“STM32 High Performance MCUs”，随后点击页面中的“Product selector”便可看到该系列对应的所有芯片型号，如下图所示（以 STM32F103 系列为例）：

Part Number	Show Description	Status & Availability	Package	Core	Operating Frequency (MHz)	FPU	Co-Processor type	Co-Processor frequency max (MHz) max	Flash Size (kB)	Dual-bank Flash	Data E2 (B) no
STM32F103C4	ACTIVE	LQFP 48 7x7x1.4 mm	Arm Cortex-M3	72	-	-	-	-	16	-	-
STM32F103C6	ACTIVE	LQFP 48 7x7x1.4 mm, UFQFPN 48 7x7x0.55 mm	Arm Cortex-M3	72	-	-	-	-	32	-	-
STM32F103C8	ACTIVE	LQFP 48 7x7x1.4 mm	Arm Cortex-M3	72	-	-	-	-	64	-	-
STM32F103CB	ACTIVE	LQFP 48 7x7x1.4 mm, UFQFPN 48 7x7x0.55 mm	Arm Cortex-M3	72	-	-	-	-	128	-	-
STM32F103R4	ACTIVE	LQFP 64 10x10x1.4 mm, TFBGA 64 5x5x1.2 P 0.5 mm	Arm Cortex-M3	72	-	-	-	-	16	-	-

图 1.6.2 ST 官网 STM32F103 系列产品（部分截图）

点击芯片型号后便可根据指引进入芯片型号的产品页，在产品页中便可找到 ST 针对该型号芯片提供的所有资源，如下图所示（以 STM32F103VE 的数据手册为例）：

The screenshot shows the 'Documentation' section of the ST official website. At the top, there are tabs for Overview, Sample & Buy, Documentation (which is highlighted with a red box), CAD Resources, Tools & Software, and Quality & Reliability. Below the tabs are sections for Quick links, Product Specifications, User Manuals, Errata Sheets, Presentations, and Product Certifications. A search bar allows users to search by title, file type, and latest update, with a 'Reset' button. The main content area is titled 'All resources' and shows a list of 'Product Specifications (1)'. One item is highlighted with a red box: 'DS5792 High-density performance line Arm®-based 32-bit MCU with 256 to 512KB Flash, USB, CAN, 11 timers, 3 ADCs, 13 communication interfaces'. The page also includes filters for Download (0), Resource title, Version, and Latest update.

图 1.6.3 ST 官网 STM32F103VE 文档资源（部分截图）

2，正点原子的学习资料

正点原子提供了大量的学习资料，为方便读者下载所有正点原子最新最全的学习资料，这些资料都放在正点原子文档中心 (<http://www.openedv.com/docs/index.html>)，如下图所示（正点原子文档中心会不时地更新，以保证为读者提供最新的学习资料）：

The screenshot shows the '正点原子资料下载中心' (Zhiyidianziwu Document Download Center). The left sidebar lists various development boards: 正点原子开发板&核心板, 正点原子STM32开发板, 正点原子STM32电机开发板, 正点原子Linux开发板, 正点原子FPGA开发板, 正点原子IOT物联网板, and 正点原子NXP开发板. The main content area is titled '正点原子资料下载中心' and '正点原子开发板&核心板'. It lists several STM32 development boards with their names in blue, indicating they are clickable links:

- 正点原子STM32开发板
 - stm32f103-mini开发板
 - stm32f103精英开发板V2
 - stm32f103精英开发板V1 (停产)
 - stm32f103战舰开发板V4
 - stm32f103战舰开发板V3 (停产)
 - stm32f407探索者开发板V3
 - stm32f407探索者开发板V2 (停产)
 - stm32f429阿波罗开发板
 - stm32f767阿波罗开发板

图 1.6.4 正点原子文档中心（部分截图）

在正点原子文档中心中，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链接。

3，正点原子论坛

正点原子论坛 (<http://www.openedv.com/forum.php>)，即开源电子网，该论坛从 2010 年成立至今，已有十多年的时间，拥有数十万的注册用户和大量嵌入式相关的帖子，每天有数百人互动，是一个非常好的嵌入式学习交流平台。

第二章 STM32 简介

本章将为读者介绍 STM32 是什么？有什么资源？能做什么？以及如何选型等基础知识，让读者对 STM32 有一个基本的了解。

本章分为如下几个小节：

2.1 初识 STM32

2.2 STM32F407 资源简介

2.1 初识 STM32

STM32 是 ST（意法半导体）推出的基于 ARM Cortex®-M 内核 32 位通用 MCU，STM32 系列产片涵盖了高性能、实时性、数字信号处理、低功耗等多种应用场景，如下图所示（截取自 ST 官网，后续可能更新）：



图 2.1.1 STM32 系列产品

2.2 STM32H750 资源简介

下面来看看 STM32H750VBT6 具体的内部资源，如表 2.2.1 所示：

STM32H750VBT6 资源					
内核	Cortex M7	低功耗定时器	5	USART	9
主频	480Mhz	通用定时器	10	CAN	2
FLASH	128KB	高级定时器	2	SDIO	2
SRAM	1060KB	16 位 ADC	3	FMC	1
封装	LQFP100	ADC 通道数	20	DMA	4
IO 数量	82	12 位 DAC	2	RTC	1
工作电压	3.3V	SPI	6	IIC	4
USB OTG	2	QUADSPI	1	SAI	4
ETH MAC	100M	JPEG Codec	支持	DCMI	支持

表 2.2.1 STM32H750VBT6 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 STM32H750 有一个全面的了解和掌握。

关于 STM32H750 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→ STM32H750VBT6.pdf，该文档即 STM32H750 的数据手册，里面有 STM32H750 详细的资源说明和相关性能参数。

第三章 开发环境搭建

本章将介绍如何搭建 STM32 的开发环境。通过本章的学习，读者将了解到在进行 STM32 的开发过程中会涉及到的开发工具，例如 IDE、调试器、串口工具等。

本章分为以下几个小节：

3.1 常用开发工具简介

3.2 MDK 安装

3.3 仿真器驱动安装

3.1 常用开发工具简介

在进行 STM32 开发的过程中需要使用到一些开发工具，例如 IDE、仿真器、串口调试助手等。常用的工具如下表所示：

工具	名称	说明
IDE	MDK	全名：MDK-ARM，是 Keil 公司（已被 ARM 收购）开发的一款 IDE，界面美观、简单易用，是 STM32 最常用的 IDE
	EWARM	IAR 公司开发的一款 IDE，支持 STM32，使用 EWARM 的人相对 MDK 少一些，习惯使用 IAR 的读者可以选择该软件开发 STM32
仿真器	CMSIS-DAP	ARM 公司开源的仿真器，支持 STM32 仿真调试，且自带虚拟串口功能。有高速和低速两个版本，具有免驱、速度快、价格便宜等优点
	ST-LINK	ST 公司开发的仿真器
	J-Link	Segger 公司开发的仿真器，支持 STM32 仿真调试，具有稳定、高速等优点，但价格昂贵
串口调试助手	ATK-XCOM	正点原子开发的串口调试助手，具有稳定、多功能、使用简单等优点
	SSCOM	大虾丁丁开发的串口调试助手，具有稳定、小巧、使用简单等优点

表 3.1.1 常用开发工具

读者可以根据自己的需求和喜好，选个合适的开发工具。上表中工具名称加粗的开发工具为本书推荐的 STM32 开发工具，即 IDE 推荐使用 MDK、仿真器推荐使用 CMSIS-DAP、串口调试助手推荐使用 ATK-XCOM，接下来介绍如何安装这些开发工具及其相关的软件驱动。

3.2 MDK 安装

注意：MDK 作为免费软件使用时，仅能用于非商业用途，若要商用，请读者自行联系 Keil 公司进行购买，本书使用 MDK 仅用于教学用途。

MDK 的安装分为两个步骤：1，安装 MDK 软件本体；2，安装对应芯片的设备包。

MDK 安装过程中涉及的安装包可以在 A 盘→6，软件资料→1，软件→MDK 中找到，如下图所示（如有压缩包，请解压后查看，且后续若软件更新，则提供的软件可能非最新版本）：

```
6, 软件资料/1, 软件/MDK/
|--- ARMCompiler_506_Windows_x86_b960.zip
|--- Keil.STM32F4xx_DFP.2.17.0.pack
|--- MDK536.EXE
`--- MDK538a.EXE
```

图 3.2.1 MDK 软件及 STM32F4 设备包

当然，读者也可以自行到软件提供方的官网自行下载，其中 MDK 软件本体由 Keil 提供，下载链接为 <https://www.keil.com/download/product/>，截止至本书编写的时候，MDK 的最新版本为 5.38a（MDK538a.EXE），由于 MDK 从 5.37 版本开始，不再随软件附带 Arm Compiler 5（AC5）编译器，并且后续进行 STM32 开发的时候需要使用 AC5，因此安装 5.37 及以后版本的 MDK 需要读者再手动安装 AC5，AC5 安装包（ARMCompiler_506_Windows_x86_b960.zip）由 ARM 提

供，下载链接为 <https://developer.arm.com/downloads/view/ACOMP5>（可能需要注册并登入账号），当然读者也可以使用附带了 AC5 的 5.36 版本（MDK536.EXE）。设备包（Keil.STM32F4xx_DFP.2.17.0.pack）是由也是由 MDK 官方提供的，下载链接为 <https://www.keil.com/pack/>。

MDK 5.36 与 MDK 5.38a 的安装过程类似，本书以 MDK5.38a 为例展示如何安装 MDK 软件本体，首先打开 MDKxxx.EXE 安装包，接着一直点击“Next”按钮即可完成安装（需要同意许可协议，安装路径保持默认即可），安装完成后就可以在电脑桌面看到 MDK 的软件图标，如下图所示：



图 3.2.2 MDK 软件图标

MDK 软件本体安装好之后，还需要安装对应芯片的设备包，设备包的安装也很简单，仅需打开设备包安装程序，一路根据提示点击“Next”按钮即可完成安装，若读者后续要使用 MDK 软件基于其他芯片进行开发，也仅需安装对应芯片的设备包即可。

至此 MDK 5.36 软件安装完成，若读者安装 MDK 5.38a，则还需进行以下步骤安装 AC5。

首先解压 AC5 安装包，然后打开安装包中的“setup.exe”可执行程序，其路径如下图所示：

```
ARMCompiler_506_Windows_x86_b960/Installer/
|--- data
\-- setup.exe
```

图 3.2.3 AC5 安装程序路径

打开 AC5 的安装程序后，也是一样在同意许可协议后一路点击“Next”按钮，但务必要记住 AC5 的安装路径，如下图所示：

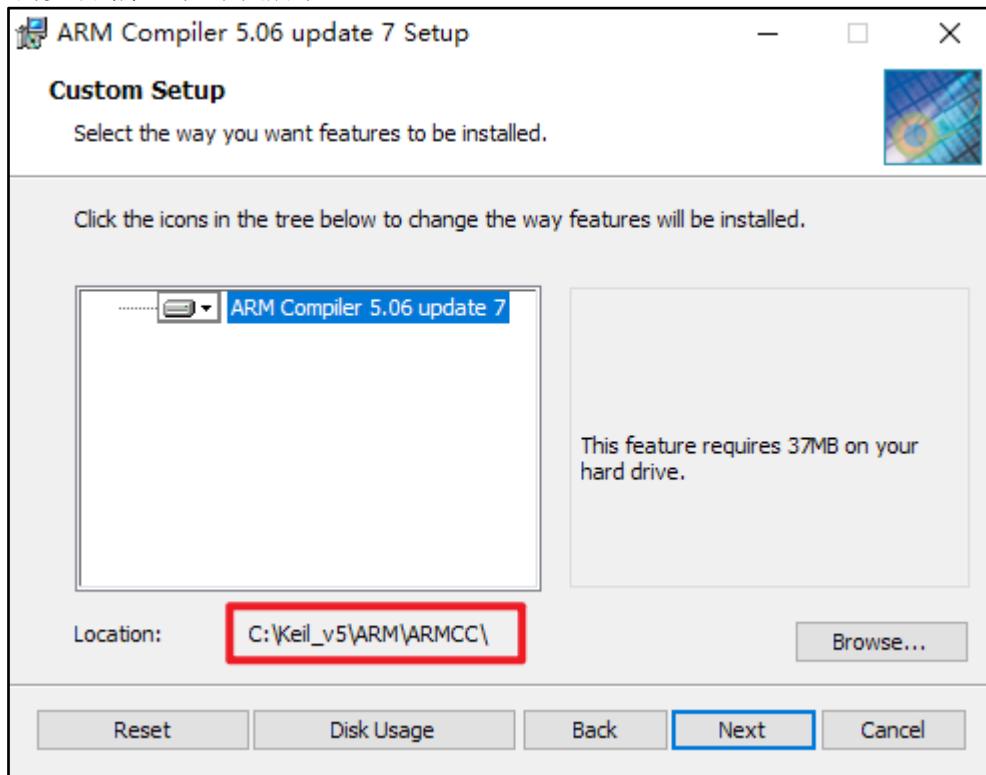


图 3.2.4 AC5 安装路径

如上图中，AC5 将被安装在 MDK 默认的安装路径里的“ARMCC”文件夹中，虽然这个路径可以是任意的，但是务必要记下这个路径。

安装完成 AC5 后，最后还要配置 MDK，使得 MDK 能够找到并使用 AC5 编译器。由于 MDK 在没有打开工程的状态下无法配置编译器，因此需要打开任意一个工程，读者可以在标准例程文

件夹（A 盘→4，程序源码→标准例程）中任意打开一个 MDK 工程，本书就以“跑马灯实验”为例，进入到标准例程→实验 1 跑马灯实验→Projects→MDK-ARM，可以看到该目录下有一个文件后缀为“uvprojx”的文件，该文件就是 MDK 的工程文件，如下图所示：

```
./4, 程序源码/标准例程/实验1 跑马灯实验/Projects/MDK-ARM/
|-- DebugConfig
|-- atk_h750.uvoptx
`-- atk_h750.uvprojx
```

图 3.2.5 MDK 工程文件

找到 MDK 工程后，使用 MDK 打开该文件，打开后如下图所示：

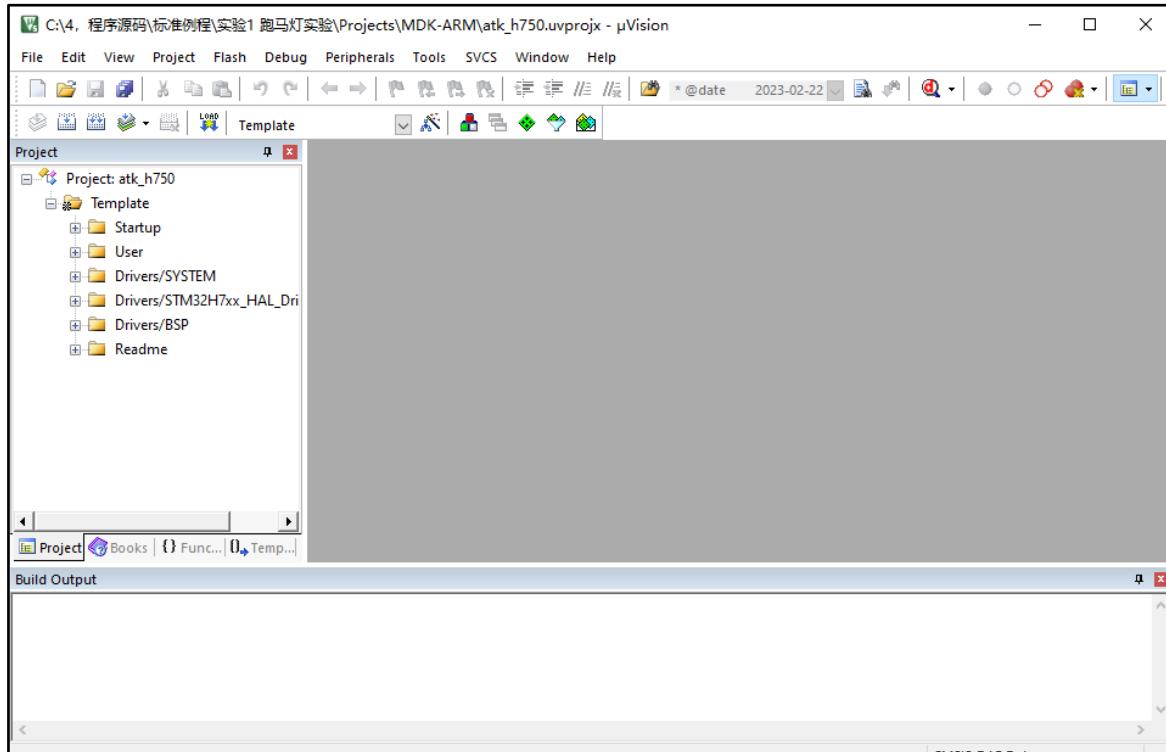


图 3.2.6 打开 MDK 工程

随后根据下图指示，打开工程项目管理窗口，如下图所示：



图 3.2.7 打开工程项目管理窗口

接着再根据下图指示，结合 AC5 的安装路径设置 MDK 使用的 AC5 编译器，如下图所示：

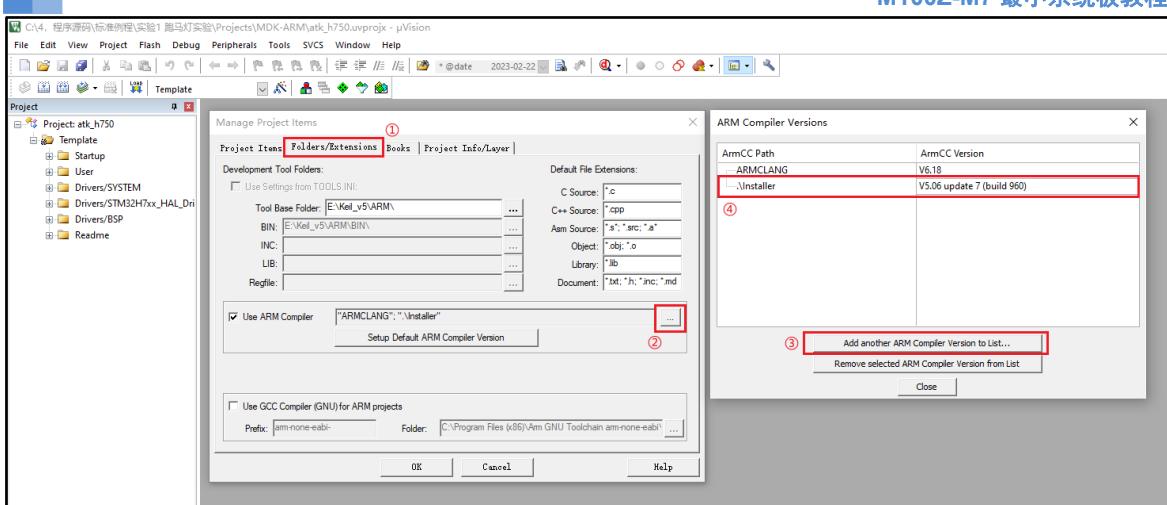


图 3.2.8 设置 MDK 使用 AC5 编译器

设置完成之后，点击工程项目管理窗口底部的“OK”按钮，关闭窗口即可，至此 MDK 安装完成。

3.3 仿真器驱动安装

STM32 可通过 CMSIS-DAP、J-Link 等仿真调试器进行程序的下载和仿真调试，本书推荐使用 CMSIS-DAP 仿真器，该仿真器在 Windows 操作系统下是免驱的，即无需安装驱动程序，即插即用，非常方便。

正点原子开发了两种规格的 CMSIS-DAP 仿真器：DAP 仿真器【高速版】ATK-HSDAP 和 DAP 仿真器【普速版】ATK-DAP，两款仿真器在使用上完全一样，只是高速版的速度更快，读者可根据实际需求进行选择。

第四章 STM32 初体验

本章并不涉及程序代码的编写，而是介绍如何编译工程、烧录程序以及进行程序的调试仿真，让读者体验 STM32 的开发流程，并会介绍一些 MDK 的使用技巧。通过本章的学习，读者将对 STM32 的开发流程及 MDK 的使用方法有一个大概的了解，为后续的深入学习打好基础。

本章分为如下几个小节：

- 4.1 使用 MDK 编译实验例程
- 4.2 使用串口烧录程序
- 4.3 使用 DAP 烧录及调试程序
- 4.4 MDK 使用技巧

4.1 使用 MDK 编译实验例程

在编写完程序代码后，需要对程序进行编译，只有在程序编译成功后，才能将编译出的二进制文件烧录至芯片中运行以及进行仿真调试等操作。

本书配套提供的实验例程在 A 盘 → 4，程序源码目录中，如下图所示：

4, 程序源码 /
'-- 标准例程

图 4.1.1 配套提供的实验例程源码

在上图“标准例程”文件夹中，就包含了基于 ST 提供的 HAL 库编写的各个实验例程，如下图所示：



图 4.1.2 标准例程

从上图中可以看到，除了“实验 0 基础入门实验”是用于创建 MDK 工程等 MDK 基础使用例程外，共有 42 个实验，其中部分实验内有多个实验例程，总的实验例程多达 58 个。

秉承着简单易懂的原则，本章以跑马灯实验例程（“实验 1 跑马灯实验”）作为示例例程，例程的根目录如下图所示：

```
实验1 跑马灯实验/
|-- Drivers
|-- Middlewares
|-- Output
|-- Projects
|-- User
|-- keilkill.bat
`-- readme.txt
```

图 4.1.3 跑马灯实验例程根目录

所有实验例程的目录结构都与上图中的目录结构类似，这会在后续创建 MDK 工程相关的章节中进行介绍（其实光看文件夹的名称，就能知道文件夹大概的作用了）。

接下来打开 MDK 工程文件，其路径为 Projects→MDK-ARM→atk_h750.uvprojx，如下图所示：

```
. /4, 程序源码/标准例程/实验1 跑马灯实验/Projects/MDK-ARM/
|-- DebugConfig
|-- atk_h750.uvoptx
`-- atk_h750.uvprojx
```

图 4.1.4 跑马灯实验例程 MDK 工程文件

注意：打开 MDK 工程文件的前提是正确地安装了 MDK 软件。

使用 MDK 打开跑马灯的 MDK 工程文件后，如下图所示：

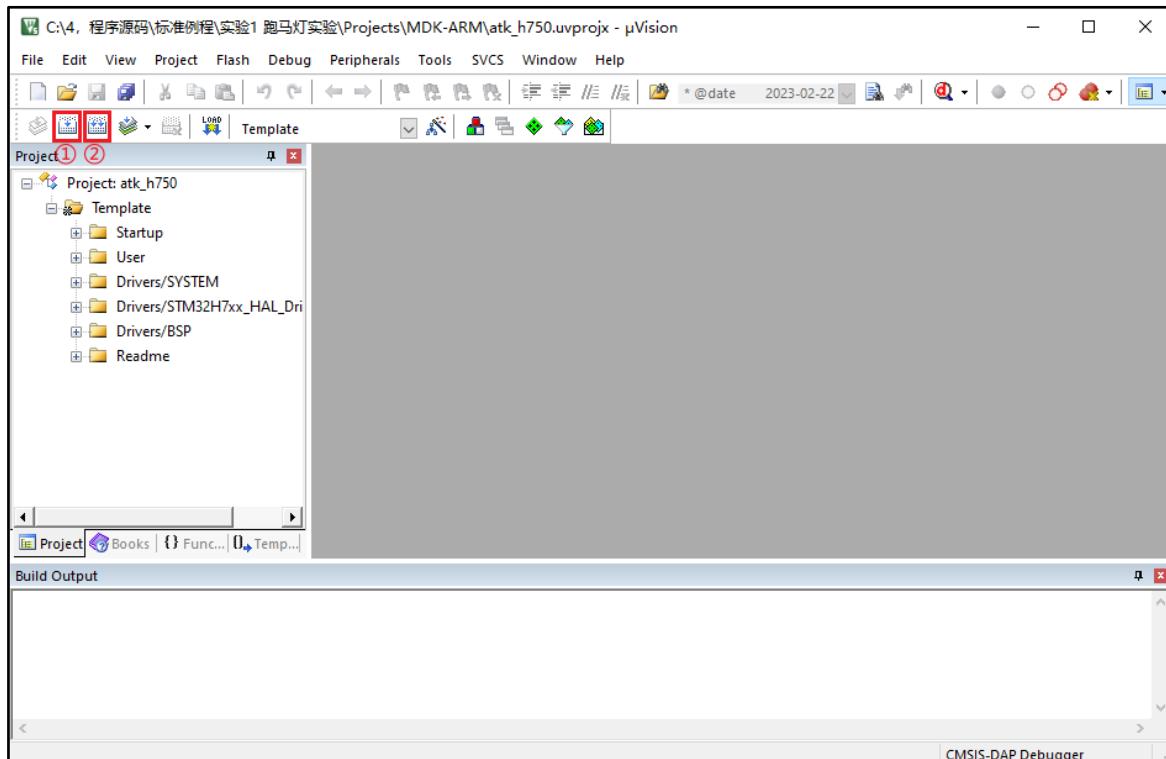


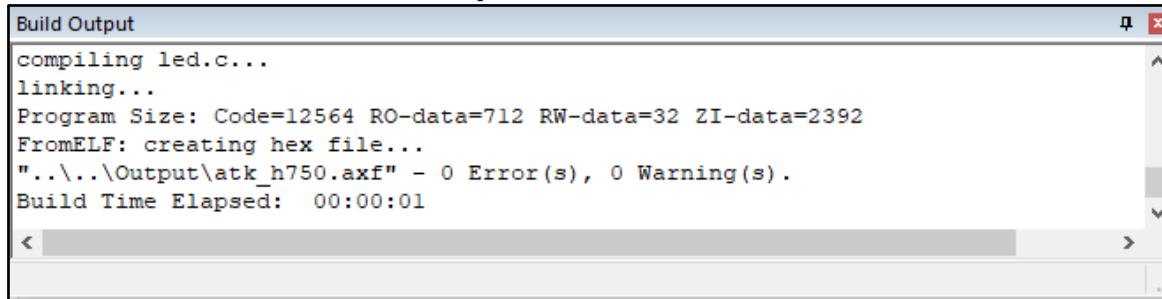
图 4.1.5 使用 MDK 软件打开跑马灯 MDK 工程

如上图所示，在 MDK 软件的左上方可以看到两个快捷按钮，这两个按钮都是用于编译工程的，区别如下：

①：Build 按钮，快捷键 F7，该按钮用于编译工程，并且该按钮比较“聪明”，使用该按钮编译工程时，只会编译在上次编译后被改动或需要重新编译的文件，并不会重新地编译整个工程，从而可以大大地缩短编译时间，推荐使用。

②：Rebuild 按钮，无快捷键，该按钮会重新地编译整个工程，因此整体上会比 Build 按钮耗时。

虽然 Build 按钮和 Rebuild 按钮存在一些差别，但是对于没有编译过的工程，这两个按钮的作用都是一样的，因此接下来按下任意一个编译按钮或直接点击快捷键 F7 编译跑马灯工程，编译完成后，MDK 软件底部的 Build Output 窗口会输出编译信息，如下图所示：



```
Build Output
compiling led.c...
linking...
Program Size: Code=12564 RO-data=712 RW-data=32 ZI-data=2392
FromELF: creating hex file...
"..\..\Output\atk_h750.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:01
```

图 4.1.6 编译结果输出信息

从上图中，就可以查看到本次编译的许多信息，这里介绍输出信息的后四行，如下：

Code: 程序代码占用的空间大小 (12564Bytes);

RO-data: 只读数据（一般是 const 关键字修饰的常量）占用的空间大小 (712Bytes);

RW-data: 有初值（初值不为 0）的可读写数据（Flash 中存放初值，取出至 RAM 中被访问）占用的空间（同时占用 Flash 和 RAM）大小 (32Bytes);

ZI-data: 无初值（初值为 0）的可读写数据占用的空间大小 (2392Bytes);

从上面的信息就可以看出，在不考虑动态内存分配的情况下，编写的程序将占用 13308Bytes (Code+RO-data+RW-data) 的 Flash 空间和 2424Bytes (RW-data+ZI-data) 的 RAM 空间；

创建了 Hex 文件（能够被单片机执行的文件，可直接烧录至单片机内部的 Flash 执行，使用串口烧录程序时使用）；

创建了 AXF 文件（不仅包含了能够被单片机执行的文件，还附加了其他的调试信息，能够进行仿真调试都靠它，使用调试器烧录程序时使用）；

编译结果有 0 个错误和 0 个警告；

编译耗时为 1 秒钟。

实验例程编译出的文件默认会输出在 Output 文件夹下，如下图所示：

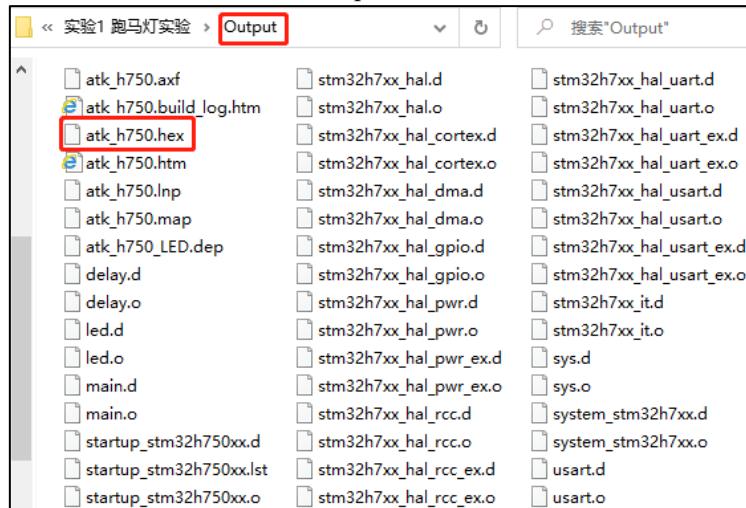


图 4.1.7 编译文件和编译过程产生的中间文件

从上图中可以看到，编译成功后，不仅生成了 Hex 和 AXF 文件，还生成了许多中间文件，对此，本书将在后续的 MAP 文件分析时进行介绍。

至此就完成了实验例程的编译，对于其他的实验例程，读者可以用同样的方式进行编译。

4.2 使用 DAP 烧录及调试程序

在需要调试程序代码的时候，最好使用仿真器进行程序的烧录和调试，本节就将介绍，如何使用正点原子 DAP 仿真器给 STM32 进行程序的烧录和调试，对于其他型号的仿真器，具体的操

作步骤也都是类似的。

正点原子 DAP 仿真器与 M100Z-M7 最小系统板 STM32H750 版的连接步骤如下：

- ①：通过板卡 USB 接口为板卡供电，也可以连接至 PC；
- ②：通过 USB 接口将正点原子 DAP 仿真器与 PC 进行连接，若连接正常，DAP 仿真器的蓝灯常亮；
- ③：通过 4P 排线将正点原子 DAP 的 SWD 接口与板卡的 SWD 接口相连。

4.2.1 使用 DAP 烧录程序

以第 4.1 小节中的跑马灯实验例程为例，在 MDK 软件界面下，点击 按钮（快捷键 Alt+F7）打开“Options for Target”窗口，同时打开窗口顶部的“Debug”选项卡，如下图所示：

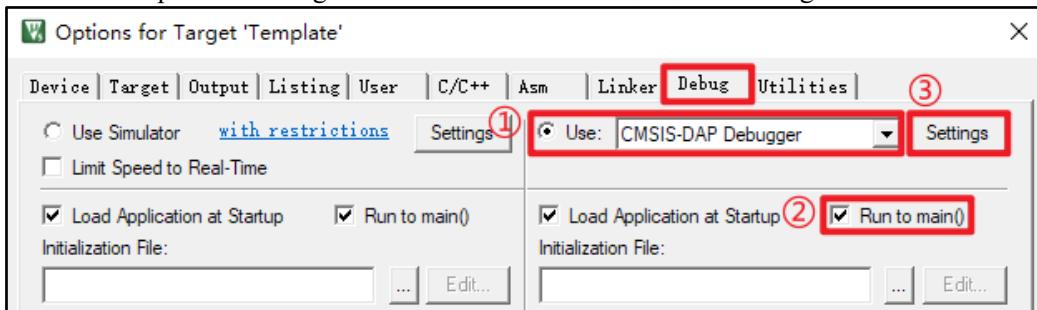


图 4.2.1.1 Debug 选项卡配置

如上图所示，打开“Debug”选项卡后，在①处根据使用的仿真器型号选择对应的调试工具，本章使用 DAP 仿真器，因此选择“CMSIS-DAP Debugger”，读者若使用其他仿真器，则对应选择调试工具即可。勾选②处的“Run to main()”复选框可以在开启调试后，程序自动运行到 main() 函数，否则程序会从启动文件中的“Reset_Handler”（复位异常）标号开始运行，读者可根据实际情况选择勾选或不勾选改复选框。接着再打开③处的“Settings”按钮，打开对应调试器的配置窗口，如下图所示：

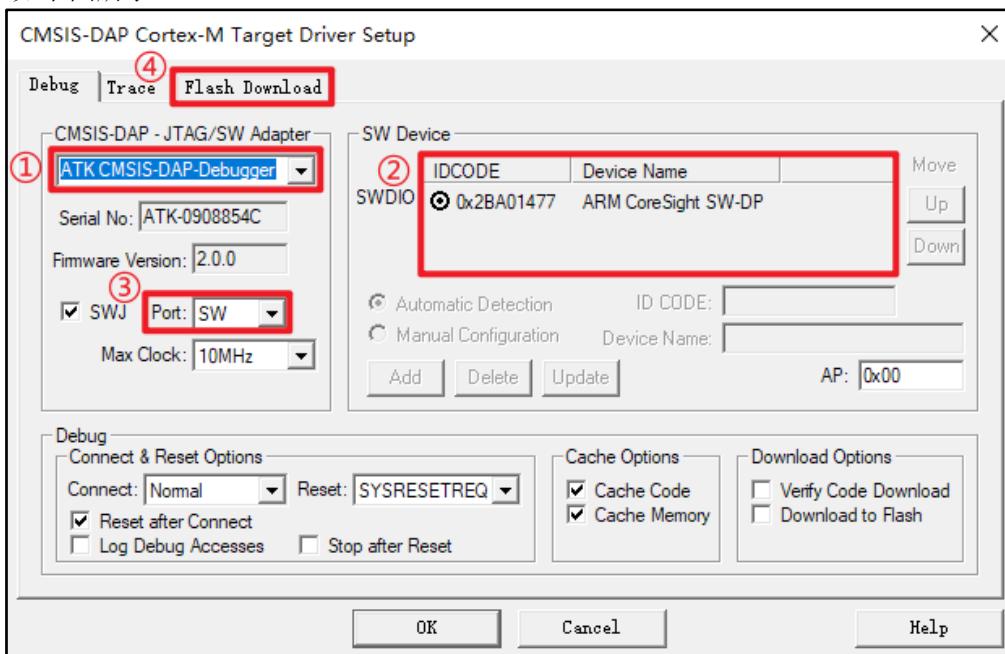


图 4.2.1.2 DAP 仿真器配置窗口

如上图所示，若 PC 上连接了多个 DAP 仿真器，则需要在①处选择对应的 DAP 仿真器，若 DAP 仿真器与板卡连接正常，则会在②处显示“IDCODE”和“Device Name”，若连接异常，则会显示“Error”，此时应该检查接线和供电。接着在③处的“Port”下拉框选择“SW”，该选项对应 4Pin 的 SWD 接口，另外一个选项为“JTAG”，其对应的是 20Pin 的 JTAG 接口。该窗口下的其他配置项保持默认，或根据实际情况进行配置。接着打开④处的“Flash

Download”选项卡，如下图所示：

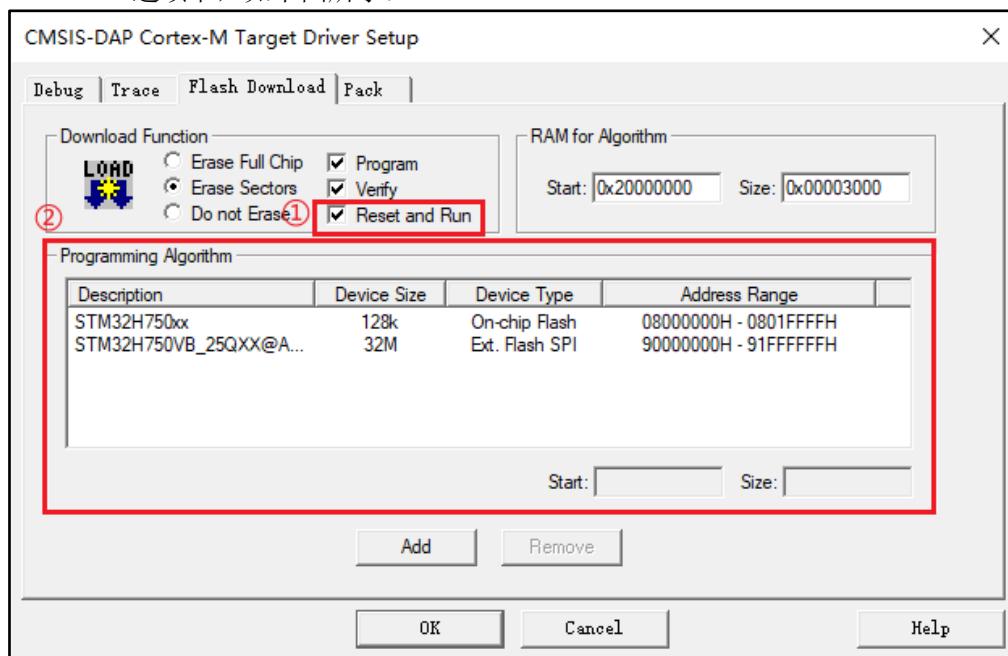


图 4.2.1.3 烧录选项配置

如上图所示，勾选①处的“Reset and Run”复选框，可以在烧录完成后自动完成复位并运行程序，若不勾选，则需要手动进行复位，程序才能运行。在②处可以选择烧录算法，程序代码一般是烧录到芯片内部的 Flash 中的，在烧录过程中，需要对芯片内部的 Flash 进行读写等操作，烧录算法就是用来告诉 MDK 软件，在烧录程序的时候应怎样读写芯片内部的 Flash。烧录算法一般有芯片厂商在芯片的设备包中提供，在安装设备包时，会一并安装烧录算法，在创建 MDK 工程时候，MDK 会自动选择芯片对应的烧录算法，若②处没有显示烧录算法，或烧录算法不符合实际使用的芯片，则需点击改窗口底部的“Add”按钮，添加对应的烧录算法。至此，使用 DAP 烧录程序的配置项均已配置完毕。

配置完成后，回到 MDK 软件主界面对工程进行编译，编译完成后，点击 按钮（快捷键 F8），随后 MDK 会自动将编译好的程序通过仿真器烧录到芯片上，同时会在“Build Output”窗口中提示烧录信息，如下图所示：

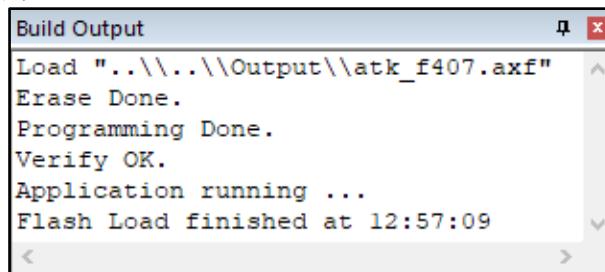


图 4.2.1.4 烧录结果输出信息

程序烧录完成后，就能看到 M100Z-M7 最小系统板 STM32F407 版上的 LED0 和 LED1 闪烁。

4.2.2 使用 DAP 调试程序

本小节依然以第 4.1 小节中的跑马灯实验例程为例，按照第 4.3.1 小节中的介绍配置好 MDK 软件中仿真器相关的配置并成功编译好程序后，点击 按钮（快捷键 Ctrl+F5）即可进入调试界面，进入调试界面前，若编译好的程序还未进行烧录操作，MDK 自会自动进行烧录操作，然后进入调试界面，调试界面如下图所示：

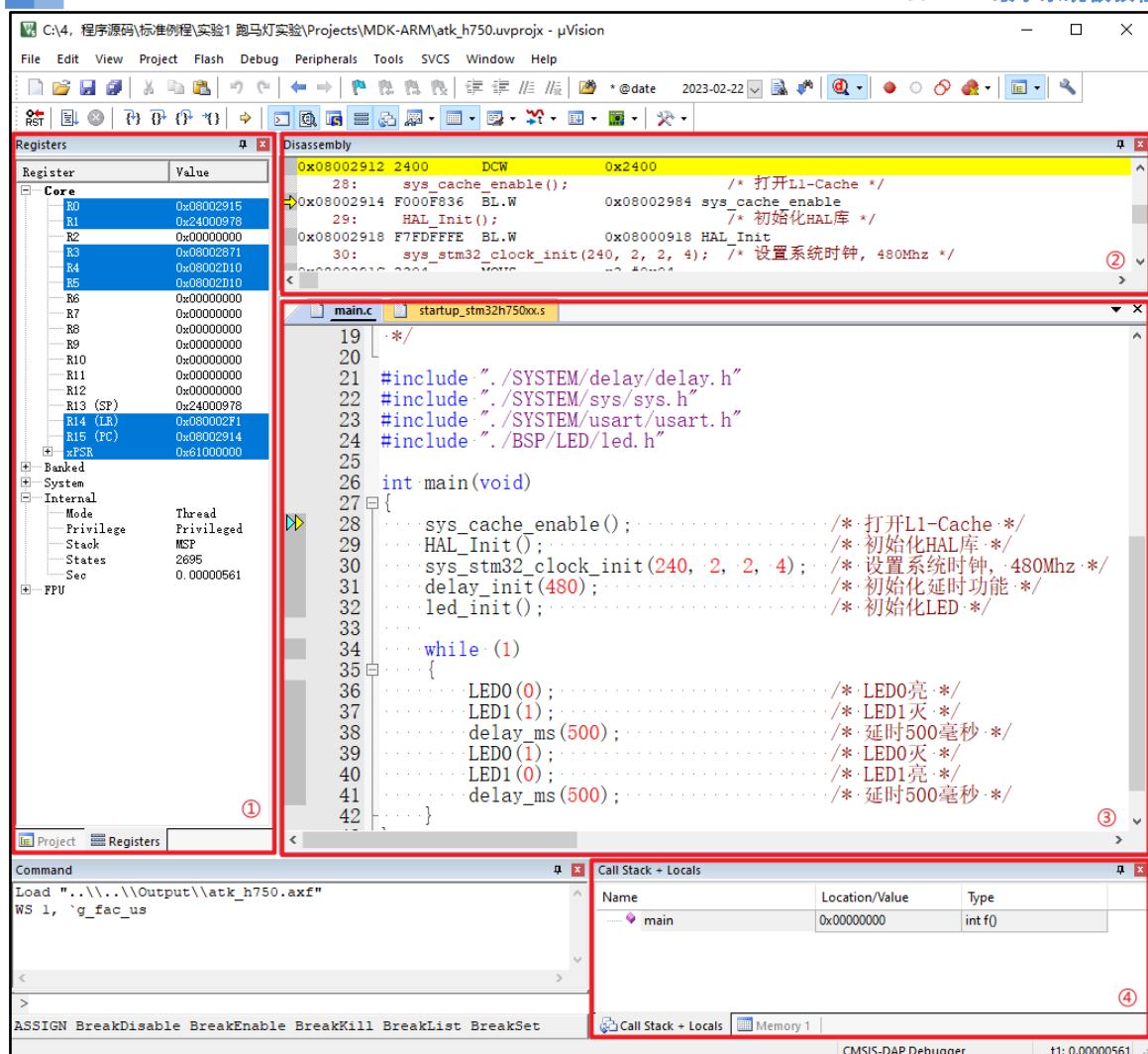


图 4.2.2.1 MDK 调试界面

上图中各个窗口的作用如所示：

①：Registers 窗口，该窗口显示了 Cortex-M4 内核寄存器 R0~R15 的值，并且还显示了内核当前的模式（Handler 模式或 Thread 模式）、当前使用的堆栈指针（MSP 或 PSP）等等一系列信息。Registers 窗口对查找 Bug 和掌握程序运行窗台有很大的帮助。

②：Disassembly 窗口，该窗口显示了反汇编代码和 C 语言代码的对比，包含了指令的保存地址、指令代码和指令等一系列信息，并且窗口的左侧有一个黄色的箭头用于指示程序当前将要执行的位置。Disassembly 窗口对查找 Bug 和优化程序有很大的帮助。

③：文本窗口，该窗口主要用于展示代码（C 代码或汇编代码），窗口左侧有一个青色和一个黄色的箭头，其中青色箭头用于指示当前光标所在的行，黄色箭头用于指示程序当前将要执行的位置。

④：Call Stack + Locals 窗口，该窗口用于显示当前函数的调用关系和函数局部变量的信息。该窗口在调试代码的时候非常有用的。

在调试界面中，还有一系列非常有用的工具，这些工具可通过 Debug 工具栏上的快捷按钮进行访问，Debug 工具栏，如下图所示：

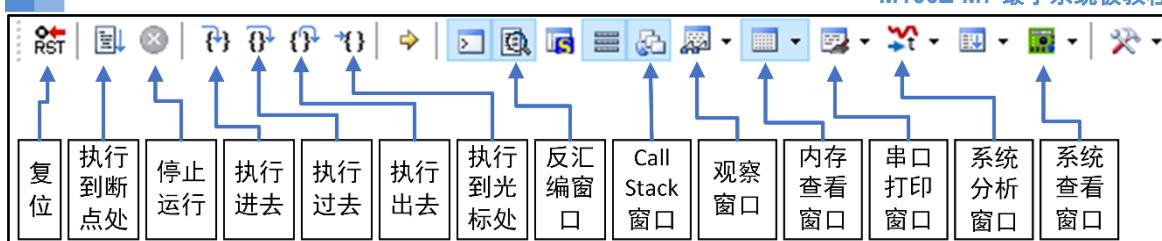


图 4.2.2.2 Debug 工具栏

复位: 其功能等同于硬件上按复位按钮。按下该按钮之后，代码会重新从头开始执行（跳转到复位异常服务函数）。

执行到断点处: 该按钮用来快速执行到断点处，有时候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，但前提是在需要查看的地方设置了断点，否则程序将全速运行。

停止运行: 该按钮在程序执行过程中变为有效，按下该按钮，可以使程序停止运行。

执行进去: 该按钮用于跳转进当前将要执行的函数中单步执行。

执行过去: 该按钮用于单步执行过当前将要执行的函数，而不跳转进函数里面。

执行出去: 该按钮用于执行出当前函数，跳转到函数返回后的位置。

执行到光标处: 该按钮用于执行到光标所在的位置，相当于在光标处打了一个临时的断点，当然，如果在执行到光标处前遇到了断点，程序也是会停止运行的。

Disassembly 窗口: 该按钮用于打开或关闭 Disassembly 窗口。

Call Stack 窗口: 该按钮用于打开或关闭 Call Stack + Locals 窗口。

Watch 窗口: 该按钮用于打开或关闭 Watch 窗口，MDK 提供了两个 Watch 窗口，在调试界面，可以将想要观察的变量或表达式添加到 Watch 窗口中，便可以实时地查看变量或表达式的值等信息。

Memory 窗口: 该按钮用于打开或关闭 Memory 窗口，MDK 提供了四个 Memory 窗口，在 Memory 窗口中可以查看一段指定起始地址内存的值。

Serial 窗口: 该按钮用于打开或关闭 Serial 窗口，MDK 提供了四个 Serial 窗口，通过前三个 Serial 窗口可以查看 MCU 串口输出的数据，通过最后一个 Serial 窗口可以查看 MDK 提供的 Event Recorder 等工具标准输出的数据。

系统分析窗口: 该按钮可以使用 MDK 提供的一些工具来查看系统运行的状态，不过部分功能对仿真器有一定要求。

系统查看窗口: 该按钮可以用来查看 MCU 各个外设的寄存器值。

以上就介绍了 Debug 工具栏中比较常用的几个快捷按钮，这些快捷按钮对调试程序是很有帮助的。接下来就实际地演示一下简单的调试。

首先打开 Watch 1 窗口，并双击“Enter expression”添加一个变量名为“g_fac_us”的表达式，该变量是在 delay.c 中定义的全局变量，主要用于微秒级延时，此时可以看到这个变量的值显示为“<cannot evaluate>”，意思是无法计算，这时可以单击 main() 函数中调用的 delay_init() 函数（第 31 行）左侧灰色的部分来添加断点，如下图所示：

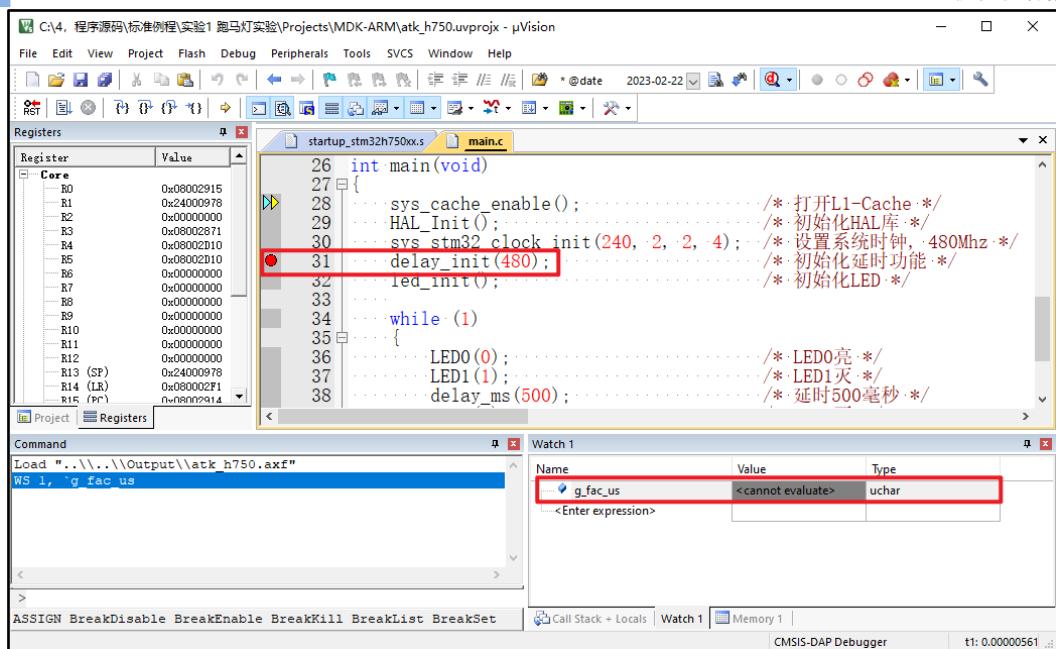


图 4.2.2.3 添加观察表达式和断点

随后点击 Debug 工具栏中的 按钮（快捷键 F5），即可运行到 delay_init() 函数调用前（断点处），然后点击 Debug 工具栏中的 按钮（快捷键 F11），即可跳转到 delay_init() 函数中，如下图所示：

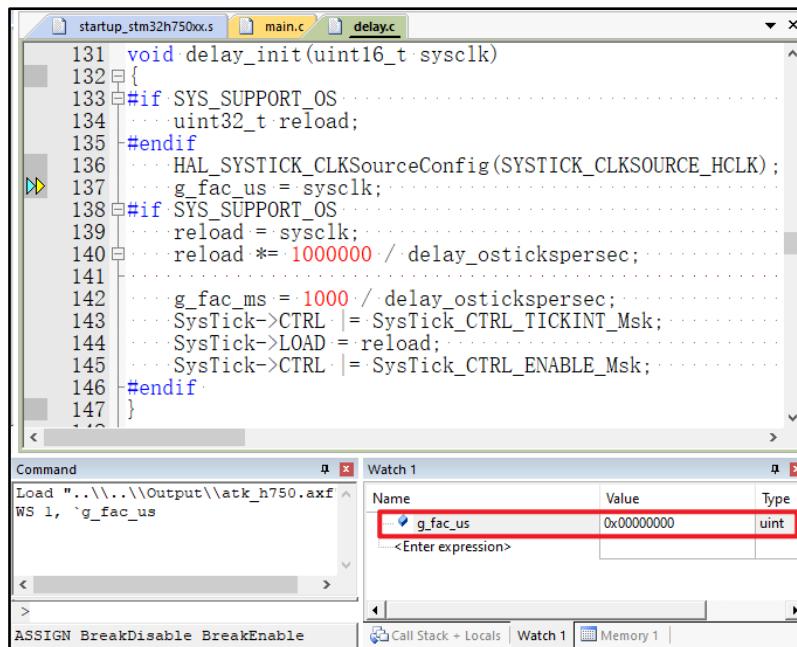


图 4.2.2.4 执行进入 delay_init() 函数

因此函数 delay_init() 会对变量 g_fac_us 进行初始化，因此可以看到 Watch 1 窗口中的表达式已经显示了具体的值，接下来可以连续单击 Debug 工具栏上的 按钮（快捷键 F10），让程序单步执行到变量 g_fac_us 被赋值的地方（大概第 137 行），此时再次单击 Debug 工具栏上的 按钮，随后便可在 Watch 1 窗口中看到变量 g_fac_us 被赋值后的结果，也可将鼠标光标放置在变量 g_fac_us 上停留一会，MDK 可会自动的提示该变量当前的值，如下图所示：

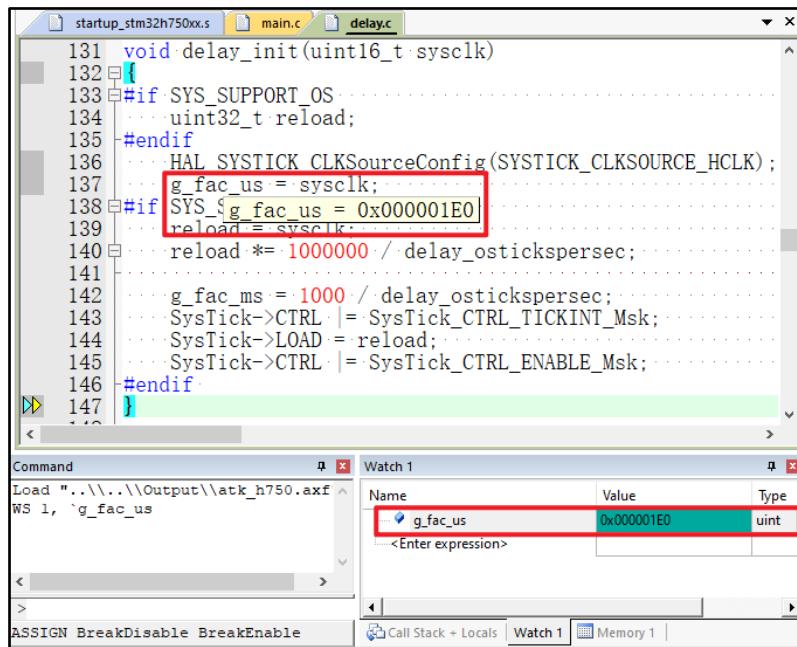
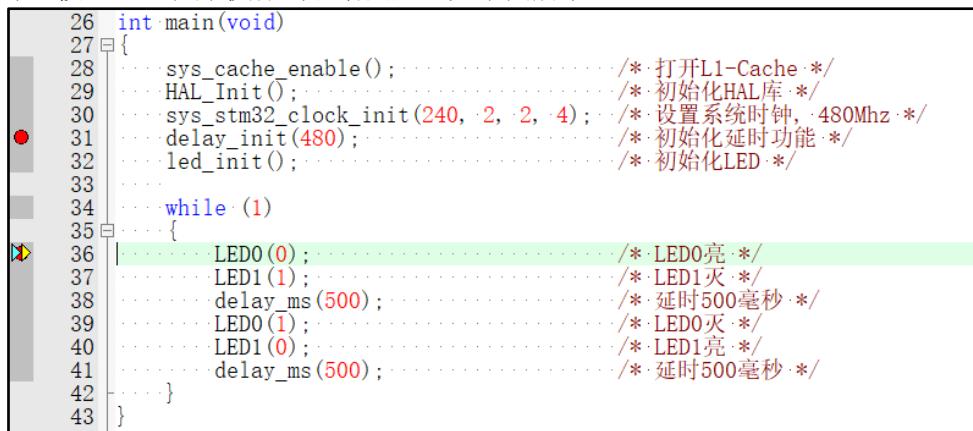


图 4.2.2.5 观察变量或表达式的值

接下来可以单击 Debug 工具栏上的 按钮（快捷键 Ctrl+F11），跳出 `delay_init()` 函数回到 `main()` 函数，然后在 `main()` 函数中 `while` 循环的第一个“`LED0(0)`”语句前打上断点单击 Debug 工具栏上的 按钮，让程序执行到该断点处，如下图所示：

图 4.2.2.6 执行到“`LED0(0)`”语句处

此时，不断点击 Debug 工具栏上的 按钮，便可以看到板卡上的 LED0 和 LED1 依次闪烁（执行到 `LED0(0)` 语句和 `LED1(1)` 语句时，需多次单击 按钮才能跳到下一行并看到对应的板载 LED 状态发生变化，这是因为 `LED0(0)` 语句和 `LED1(1)` 语句是两个宏定义，这两个宏定义由多条语句组成，因此需要多次单击 按钮才能执行到下一行）。

此时，单击 “`LED0(0)`” 语句前的断点，将其取消掉，然后点击 Debug 工具栏上的 按钮，使程序全速执行，则可以看到板载 LED 自动地交替闪烁，随后打开 `delay.c` 文件并定位到 `delay_us` 函数（大概 215 行），并在该函数左侧任意有灰色的地方打上断点，可以看到程序马上停在了断点的位置，如下图所示：

```

214 void delay_us(uint32_t nus)
215 {
216     uint32_t temp;
217     SysTick->LOAD = nus * g_fac_us; /* 时间加载 */
218     SysTick->VAL = 0x00; /* 清空计数器 */
219     SysTick->CTRL |= 1 << 0; /* 开始倒数 */
220     ...
221     do
222     {
223         temp = SysTick->CTRL;
224     } while ((temp & 0x01) && !(temp & (1 << 16))); /* CTRL.ENABLE位必须为1，并等待 */
225     SysTick->CTRL &= ~(1 << 0); /* 关闭SYSTICK */
226     SysTick->VAL = 0x00; /* 清空计数器 */
227 }

```

图 4.2.2.7 执行进入 delay_us() 函数

此时，便可在 Call Stack + Locals 窗口查看到函数的调用情况，如下图所示：

Name	Location/Value	Type
delay_us	0x080028C8	void f(uint)
nus	0x00007530	param - uint
temp	<not in scope>	auto - uint
delay_ms	0x080028B0	void f(ushort)
main	0x00000000	int f()

图 4.2.2.8 Call Stack + Locals 窗口

如上图所示，可以很清楚地查看到 delay_us() 函数是如何被调用的，从下往上看，main() 函数调用了 delay_ms() 函数，然后 delay_ms() 函数调用了 delay_us() 函数。

以上就是使用 DAP 调试程序的全部内容，这方便的内容也是比较重要的，掌握好调试程序的方法，对解决程序 Bug 是很有帮助的，并且对了解第三方库等的代码也很有帮助。

4.3 MDK 使用技巧

本小节将介绍 MDK 的一些使用技巧，这些技巧对提供开发效率是很有帮助的，读者可以实际操作一下，加深印象。

4.3.1 自定义编辑器

点击 MDK 软件顶部的 按钮，打开 MDK 配置窗口，并切换到顶部的 Editor 选项卡，如下图所示：

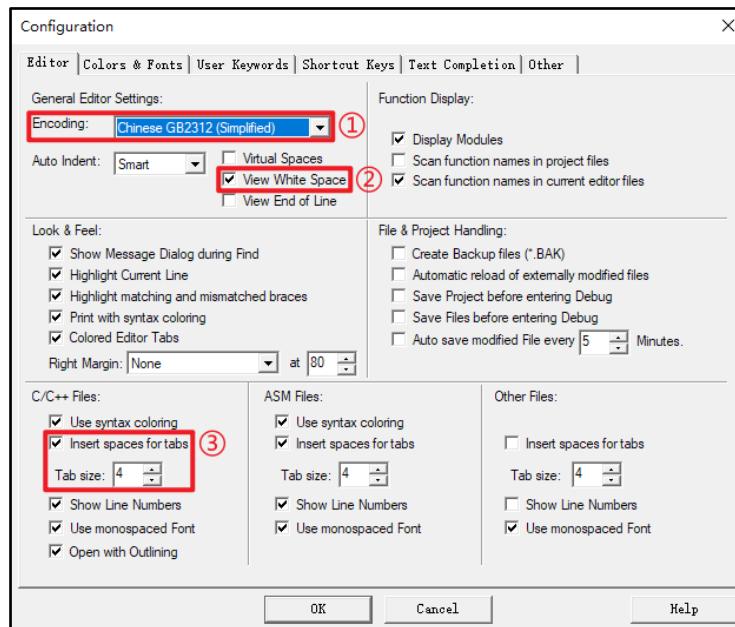


图 4.3.1.1 Editor 选项卡配置

①：“Encoding”用于设置在 MDK 中新建文件保存时使用的编码格式，推荐使用“Chinese GB2312(Simplified)”，该编码格式对中文的支持比较好。

②：勾选“View White Space”复选框可以在文本编辑器中以一个浅灰色的“·”代替一个空格进行显示，有助于代码对齐。

③：勾选“Insert spaces for tabs”复选框可以设置在文本编辑器输入时，使用空格来代替 TAB，TAB 键常用于代码对齐，但由于不同的文本编辑器对 TAB 长度的定义各不相同，因此使用 TAB 对齐可能在不同的文本编辑器中会出现不对齐的情况，因此可以使用空格来代替 TAB，以解决这个问题，同时可以设置使用多少个空格来代替一个 TAB，正点原子的编写代码均是以四个空格代替一个 TAB，建议读者进行相同的设置。

接着是 Colors & Fonts 选项卡，如下图所示：

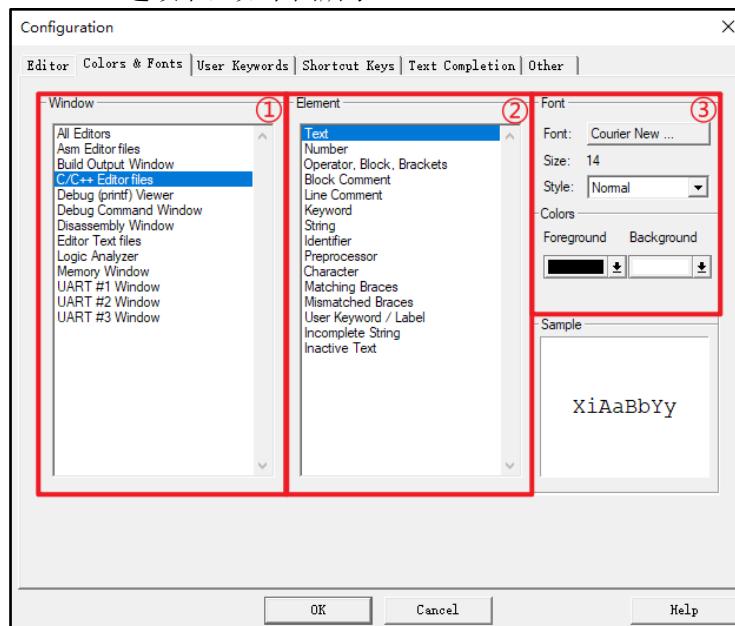


图 4.3.1.2 Colors & Fonts 选项卡配置

①：“Window”窗口用于选择要修改的窗口对象，例如要修改的对象为 C/C++文本编辑器窗口，则选择“C/C++ Editor files”。

②：“Element”窗口用于选择窗口对象中要修改的元素，例如要修改的对象为 C/C++文本编

辑器窗口中的文本，则选择“Text”。

③：“Font”和“Colors”用于修改被选中窗口对象中元素的字体、字号、样式、前景色和背景色等参数，读者可以根据自己的喜好进行修改。

接着是 User Keywrds 选项卡，如下图所示：

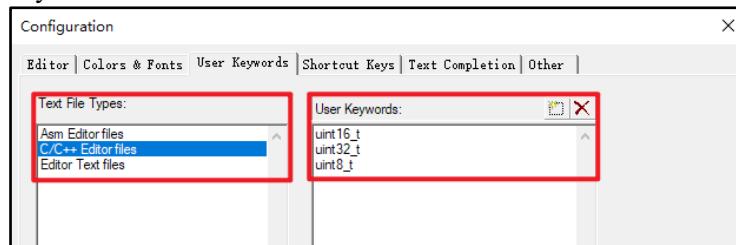


图 4.3.1.3 User Keywords 选项卡配置

该选项卡用于添加一些自定义的关键字，例如，在 C 语言中大约有 37 个关键字(C99 标准)，MDK 在 C/C++ 文本编辑器可以高亮显示这些关键字(可以在 Colors & Fonts 选项卡→C/C++ Editor files→Keyword 中进行自定义高亮的颜色)，让用户更容易地阅读代码，但是在代码中用 `typedef` 定义的变量类型名称或 `struct` 定义的结构体名称等，MDK 默认并不会高亮显示这些用户自定义的“关键字”，因此可以在 User Keyword 选项卡中添加自定义的“关键字”，如上图所示，就添加了三个自定义的关键字，以上文提到的 `delay_init()` 函数为例，该函数如使用到上述被添加的三个关键字，则都会被高亮，如下图所示：

<pre>void delay_init(uint16_t sysclk) { #if SYS_SUPPORT_OS uint32_t reload; #endif HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK); g_fac_us = sysclk; #if SYS_SUPPORT_OS reload = sysclk; reload *= 1000000 / delay_ostickspersec; g_fac_ms = 1000 / delay_ostickspersec; SysTick->CTRL = SysTick_CTRL_TICKINT_Msk; SysTick->LOAD = reload; SysTick->CTRL = SysTick_CTRL_ENABLE_Msk; #endif }</pre>	<pre>void delay_init uint16_t sysclk // 自定义关键字后 { #if SYS_SUPPORT_OS uint32_t reload; #endif HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK); g_fac_us = sysclk; #if SYS_SUPPORT_OS reload = sysclk; reload *= 1000000 / delay_ostickspersec; g_fac_ms = 1000 / delay_ostickspersec; SysTick->CTRL = SysTick_CTRL_TICKINT_Msk; SysTick->LOAD = reload; SysTick->CTRL = SysTick_CTRL_ENABLE_Msk; #endif }</pre>
---	---

图 4.3.1.4 自定义关键字效果

4.3.2 动态语法检测&代码自动补全

在 Configuration 窗口的 Text Completion 选项卡中可以配置动态语法检测和代码自动补全功能，如下图所示：

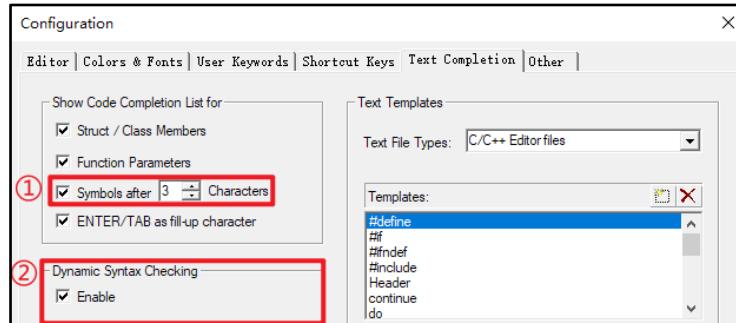


图 4.3.2.1 Text Completion 选项卡配置

①：用于使能代码自动补全功能，并可以配置在手动输入多少个字符后提示匹配自动补全的内容，实际的使用效果，如下图所示：

```

int main(void)
{
    sys_cache_enable();
    HAL_Init();
    sys_stm32_clock_init(240, 2, 2, 4);
    delay_init(480);
    led_init();
    del
        ADC_TWOSAMPLINGDELAY_9CYCLES
        ♦DelayBlock_Configure
        ♦DelayBlock_Disable
        ♦DelayBlock_Enable
        ♦delay_init
        ♦delay_ms
        ♦delay_us
        ETH_MACIVIR_VLC_VLANTAGDELETE
        ETH_MACIVIR_VLC_VLANTAGDELETE_Msk
}

```

图 4.3.2.2 代码自动补全功能

②：用于使能动态语法检测，这可以帮助用户在编写代码时，实时地观察代码语法的问题，实际的使用效果，如下图所示：

```

26 int main(void)
27 {
28     sys_cache_enable();
29     HAL_Init();
30     sys_stm32_clock_init(240, 2, 2, 4);
31     delay_init(480);
32     led_init();
33     delay_ms(65536); // Warning: Value too large for type
34     delay_ms(10); // Error: Syntax error - missing semicolon
35
36     while (1)
37     {
38         LED0(0);
39         LED1(1);
40         delay_ms(500);
41         LED0(1);
42         LED1(0);
43         delay_ms(500);
44     }
45 }

```

图 4.3.2.3 动态语法检测功能

如上图所示，动态语法检测功能可以对编写的代码进行提示，没有语法错误的代码则不会有任何提示，若语法有误，则会有警告（⚠）和错误（✖）两种提示，如第 33 行有警告提示，是因为 `delay_ms()` 函数的形参是 16 位无符号整型，但实际传入的参数已经超出了 16 位无符号整型的范围，因此给出了警告的提示，警告并不会导致编译出错，但是在程序实际运行过程中可能会有意料之外的结果；再如第 34 行有错误提示，是因为 `delay_ms()` 函数调用后面少了一个分号（“;”），错误提示会直接导致程序编译失败。

4.3.3 代码编辑技巧

本小节介绍几个常用的 MDK 代码编辑技巧，希望对读者提高 MDK 代码编辑效率有所帮助。

1. TAB 键的妙用

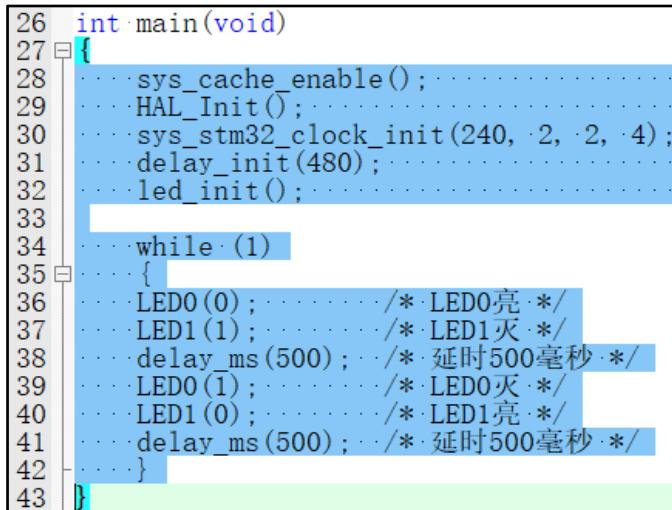
在前面的章节中也提到了可以使用 TAB 键来进行代码对齐，那具体的使用方法呢？按下 TAB 键可以使被选中的代码段或光标后的代码向右侧对齐，同时也可以按下 Shift+TAB 键使被选中的代码段或光标后的代码向左侧对齐，如下图所示：

```

26 int main(void)
27 {
28     sys_cache_enable();
29     HAL_Init();
30     sys_stm32_clock_init(240, 2, 2, 4);
31     delay_init(480);
32     led_init();
33
34     while (1)
35     {
36         LED0(0); /* LED0亮 */
37         LED1(1); /* LED1灭 */
38         delay_ms(500); /* 延时500毫秒 */
39         LED0(1); /* LED0灭 */
40         LED1(0); /* LED1亮 */
41         delay_ms(500); /* 延时500毫秒 */
42     }
43 }
```

图 4.3.3.1 未对齐的代码

如上图所示，未对齐的代码一看就令人头大，若在每一行需要对齐的代码前手动敲入多个空格，那效率是非常低的，此时可以多行选中需要对齐的代码段，然后按以下 TAB 键，即可使被选中的代码段向右对齐，如下图所示：

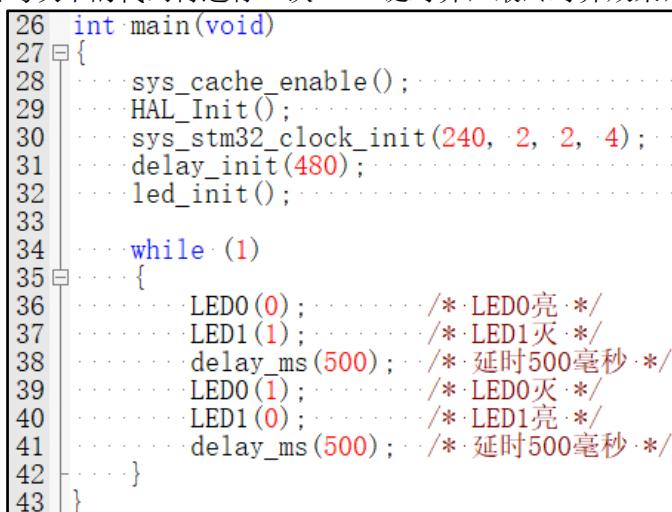


```

26 int main(void)
27 {
28     sys_cache_enable();
29     HAL_Init();
30     sys_stm32_clock_init(240, 2, 2, 4);
31     delay_init(480);
32     led_init();
33
34     while (1)
35     {
36         LED0(0); /* LED0亮 */
37         LED1(1); /* LED1灭 */
38         delay_ms(500); /* 延时500毫秒 */
39         LED0(1); /* LED0灭 */
40         LED1(0); /* LED1亮 */
41         delay_ms(500); /* 延时500毫秒 */
42     }
43 }
```

图 4.3.3.2 用 TAB 键进行对齐

接着对于 while 语句块中的代码再进行一次 TAB 键对齐，最终对齐效果，如下图所示：



```

26 int main(void)
27 {
28     sys_cache_enable();
29     HAL_Init();
30     sys_stm32_clock_init(240, 2, 2, 4);
31     delay_init(480);
32     led_init();
33
34     while (1)
35     {
36         LED0(0); /* LED0亮 */
37         LED1(1); /* LED1灭 */
38         delay_ms(500); /* 延时500毫秒 */
39         LED0(1); /* LED0灭 */
40         LED1(0); /* LED1亮 */
41         delay_ms(500); /* 延时500毫秒 */
42     }
43 }
```

图 4.3.3.3 使用 TAB 键的最终对齐效果

2. 快速定位函数/变量被定义的地方

在编写或调试代码的时候，经常会需要查看函数或变量的定义，如果每个都手动查找，那效率就太低了，MDK 软件提供了一键跳转到函数或变量定义地方的功能，但该功能是可选的，需要开启该功能后才能使用，开启方式也很简单，在 Options for Target 窗口的 Output 选项卡中勾选 Browse Information 复选框即可开启该功能，如下图所示：

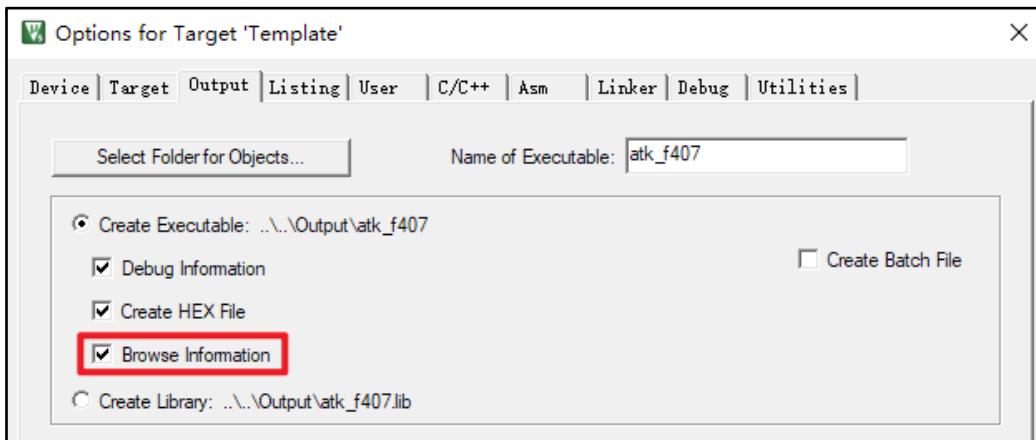


图 4.3.3.4 开启 Browse Information 功能

开启 Browse Information 功能后，还需进行一次编译才能使用跳转功能，使用方式也很简单，仅需单击或自定义选中函数或变量名，然后点击快捷键 F12 即可进行跳转，在查看完相关定义后，可以直接点击工具栏上的 \leftarrow 按钮（快捷键 Ctrl+-），即可一步一步地返回。

3. 快速注释/取消注释

MDK 提供了一键注释或取消注释代码段的功能，使用方法也很简单，若要注释或取消一段代码，则先选中代码段，然后点击工具栏上的 H （注释）或 H （取消注释），即可一键对代码段进行注释或取消注释。

4.3.4 其他小技巧

本小节再介绍几个 MDK 软件使用的小技巧。

①：右键#include 包含的头文件，弹出的菜单中有“Open document “xxx””的选项，可以快速地在文本编辑器中打开头文件，如下图所示：

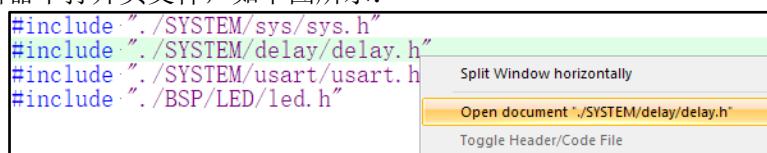


图 4.3.4.1 快速打开头文件

②：“Ctrl+H”、“Ctrl+F”和“Ctrl+Shift+F”的组合快捷键可一键打开替换、查找和跨文件查找的功能窗口，如下图所示：



图 4.3.4.2 替换、查找和跨文件查找的功能窗口

MDK 软件是一个非常强大的代码编辑和编辑工具，熟悉 MDK 的使用方法还是很有必要的。

第五章 STM32 基础知识入门

本章着重介绍 STM32 的一些基础知识，令读者对 STM32 有一个初步的了解，为后面的章节学习做铺垫。对于本章的内容，读者可以先只看一个大概，在后面需要使用到这部分知识点的时候，再回过头来仔细看。

本章分为如下几个小节：

5.1 C 语言基础知识复习

5.2 寄存器基础知识

5.3 STM32F407 系统架构

5.1 C 语言基础知识复习

本节介绍 C 语言的基础知识，对 C 语言比较熟练的读者，可以跳过本节，对于基础比较薄弱的读者，建议好好学习一下本节的内容。

由于 C 语言博大精深，不可能由一小节的内容就全讲明白，所以本节知识回顾在进行 STM32 开发时常用的几个 C 语言知识点，以便读者更好的学习本书后续的内容，并编写相关的代码。

5.1.1 位操作

C 语言的位操作就是对基本类型变量进行位级别的操作。本节的内容比较简单，这里也就点到为止，不深入探讨。下面先讲解几种位操作运算符，然后介绍其相关的使用技巧。C 语言支持如下 6 种位操作的运算符：

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 6 种位操作的运算符

下面介绍这些位操作运算符的使用技巧。

①：在不改变其他位的情况下，对某几个位进行设值

这个场景在单片机开发中经常使用，方法就是先对需要设置的位用&运算符进行清零操作，然后用|运算符设值。例如要设置 GPIOA 的 ODATA 寄存器 Bit6（第 6 位）为 1，则可以先使用&运算符对该寄存器的 Bit6 进行清零操作：

```
GPIOA->ODATA &= 0xFFFFFFFBF; /* 将 Bit6 清 0 */
```

然后再使用|运算符对该寄存器的 Bit6 进行置 1 操作：

```
GPIOA->ODATA |= 0x00000040; /* 将 Bit6 置 1 */
```

②：移位操作提高代码的可读性

例如①中|操作使用到的 0x00000040，虽然通过换算，可以知道是将 Bit6 置 1，但是这样的表达可读性比较差，可以通过移位操作对其进行改进：

```
GPIOA->ODATA |= (1 << 6); /* 将 Bit6 置 1 */
```

这么一来，就可以非常直观地看出是将 Bit6 置 1 了。

③：按位取反操作使用技巧

②中使用移位操作改进①中仅使用|运算将 Bit6 置 1 的可读性，但要改进①中使用&运算将 Bit6 清 0 的可读性还需借助按位取反操作：

```
GPIOA->ODATA &= ~(1 << 6); /* 将 Bit6 清 0 */
```

这么一来，就可以非常直观地看出是将 Bit6 清 0 了。

④：按位异或操作使用技巧

按位异或可以很方便地对 Bit 位进行翻转，例如不考虑 LED 当前是何种状态，只要求控制 LED 翻转状态等情况（假设 LED 的亮灭状态由 PA6 输出的高低电平控制）：

```
GPIOA->ODATA ^= (1 << 6); /* Bit6 的值取反 */
```

这么一来，就可以很方便地操作 PA6 引脚输出相反的电平，而不用先读取 PA6 输出的电平状态然后才输出相反的电平。

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，其常见的格式如下：

```
#define 标识符 字符串
```

“标识符”为所定义宏的名称；“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 8000000U，数字后的 U 是 unsigned（无符号）的意思，随后便可在程序代码中使用 HSE_VALUE 来代替 8000000U。

至于 define 宏定义的一些其他高级用法，例如宏定义带参数等，本章不过多介绍。

5.1.3 ifdef 条件编译

在单片机程序开发过程中，经常会遇到需要在满足某些条件时对一段代码进行编译，而当条件不满足或满足另一条件时编译另一段代码，这就可以使用条件编译，条件编译最常见的形式如下：

```
#ifdef 标识符
代码段 1
#else
代码段 2
#endif
```

如上的条件编译，当标识符被定义过（一般使用 define 进行定义），则代码段 1 会被编译，否则会编译代码段 2。

5.1.4 extern 外部申明

C 语言中 extern 关键字用于修饰变量或函数，以表示变量或函数定义在别的文件中，提示编译器遇到此变量或函数时，需在其他文件中寻找其定义。这里要注意的是，可以使用 extern 多次在不同文件中修饰同一个变量或函数，但该变量或函数只能被定义一次

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。例如在编写程序时经常使用到的 uint8_t、uint16_t 和 uint32_t 等都是由 typedef 定义的类型别名，其定义如下：

```
typedef unsigned char      uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int       uint32_t;
```

这么一来就可以在编写程序代码的时候使用 uint8_t 等代替 unsigned char 等，极大地提高了代码的可读性可编写代码的效率。

5.1.6 struct 结构体

struct 用于定义结构体，结构体就是一堆变量的集合，结构体中的成员变量的作用一般都是相互关联的，定义结构体的形式如下：

```
struct 结构体名
{
    成员变量 1 的定义;
    成员变量 2 的定义;
    .....
};
```

例如：

```
struct lcd_device_struct
{
```

```

    uint16_t width;
    uint16_t height;
};

```

如上举例的结构体定义，一堆描述 LCD 屏幕的变量的集合，其中包含了 LCD 屏幕的宽度和高度。

结构体变量的定义如下：

```
struct lcd_device_struct lcd_device;
```

如上，就定义了一个名为 lcd_device 的结构体变量，那么怎么访问这个结构体变量中的成员变量呢？如下：

```

lcd_device.width = 240;
printf("LCD Height: %d\n", lcd_device.height);

```

如上就展示了结构体变量中成员变量的访问操作。

5.1.7 指针

指针是另一个变量的变量，指针变量在内存中保存的是另一个变量在内存中的地址，通过指针可以访问到另一个变量所在内存地址中的数据。

举一个定义指针的例子，如下：

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

如上，就定义一个名为 p_str 的指针变量，并将 p_str 指针指向了字符串 “This is a string!” 保存在内存中首地址，对于 STM32 来说，此时 p_str 的值就是一个 32 位的数，这个数就是一个内存地址，这个内存地址就是上述字符串保存在内存中的首地址。

通过 p_str 指针就可以访问到字符串 “This is a string!”，那具体是如何访问的呢？前面说 p_str 保存的是一个内存地址，那么就可以通过这个内存地址去内存中读取数据，通过*p_str 就可以访问地址为 p_str 的内存数据，*(p_str + 1) 可以访问下一个内存地址中的数据。

知道了如何访问内存中的数据，但是读取到的数据要如何解析呢？这就有 p_str 指针的类型决定了。在这个例子中 p_str 是一个 char 类型的指针，那么访问*p_str 就是访问地址为 p_str，大小为 sizeof(char)（一般为一个字节）的一段内存数据，在这个例子中就可以读取到字符 “T”，读取*(p_str + 1) 就是 “h”，以此类推。

指针是 C 语言的精髓，但也是初学者望而生畏的一个知识点，若读者一时半会无法理解指针的用法也没关系，善用搜索引擎，网上有很多参考的学习资料。

5.2 寄存器基础知识

寄存器（Register）是一种特殊的内存，它主要用于实现控制和访问 MCU 的内核和各个片上外设。

对于 STM32 来说，寄存器一般都是 32 位的，但由于 MCU 上的内存资源十分宝贵，因此在一个 32 位寄存器中，会使用其中的 1 位或多为来控制或访问 MCU 的内核或各个片上外设的一种功能，但即使如此，STM32 上也还是有上百个寄存器，这实际上是因为 STM32 有很多的片上外设导致的，只要将这些寄存器按照功能分好类，学起来就不难了。

从大方向来区分，STM32 中的寄存器可分为两大类，分别为内核寄存器和外设寄存器，大类下还可以分出许多小类，如下表所示：

大类	小类	说明
内核寄存器	内核相关寄存器	R0~R15、xPSR 和特殊功能寄存器等
	中断相关寄存器	NVIC、SCB 相关的寄存器
	SysTick 寄存器	SysTick 相关寄存器
	DBGMCU 寄存器	DBGMCU 相关寄存器
外设寄存器	GPIO、UART、IIC、SPI 等片上外设	GPIO、UART、IIC、SPI、TIM、DMA、ADC、DAC、RTC、FSMC、RCC、PWR 等片上外设相关寄存器

表 5.2.1 STM32 寄存器分类

对于初学者来说仅需在学习 MCU 的各个片上外设时，再去学习该片上外设相关的寄存器即可。

5.3 STM32H750 系统架构

STM32H750 是 ST 公司基于 ARM 授权 Cortex M7 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 M7 内核 & 芯片

ARM公司提供内核（如Cortex M7，简称CM7，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CM7内核授权后，就可以把CM7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 M7内核和芯片的关系如图5.3.1.1所示：

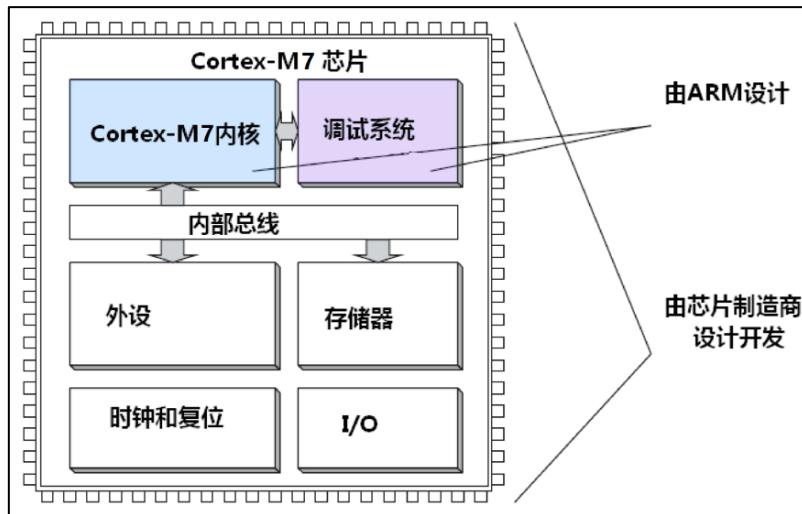


图 5.3.1.1 Cortex M7 内核 & 芯片关系

可以看到，ARM公司提供CM3内核和调试系统，其他的东西（外设（IIC、SPI、UART、TIM等）、存储器（SRAM、FLASH等）、I/O等）由芯片制造商设计开发。这里ST公司就是STM32H750芯片的制造商。

5.3.2 STM32 系统架构

STM32H750VBT6 内部系统结构如图 5.3.2.1：

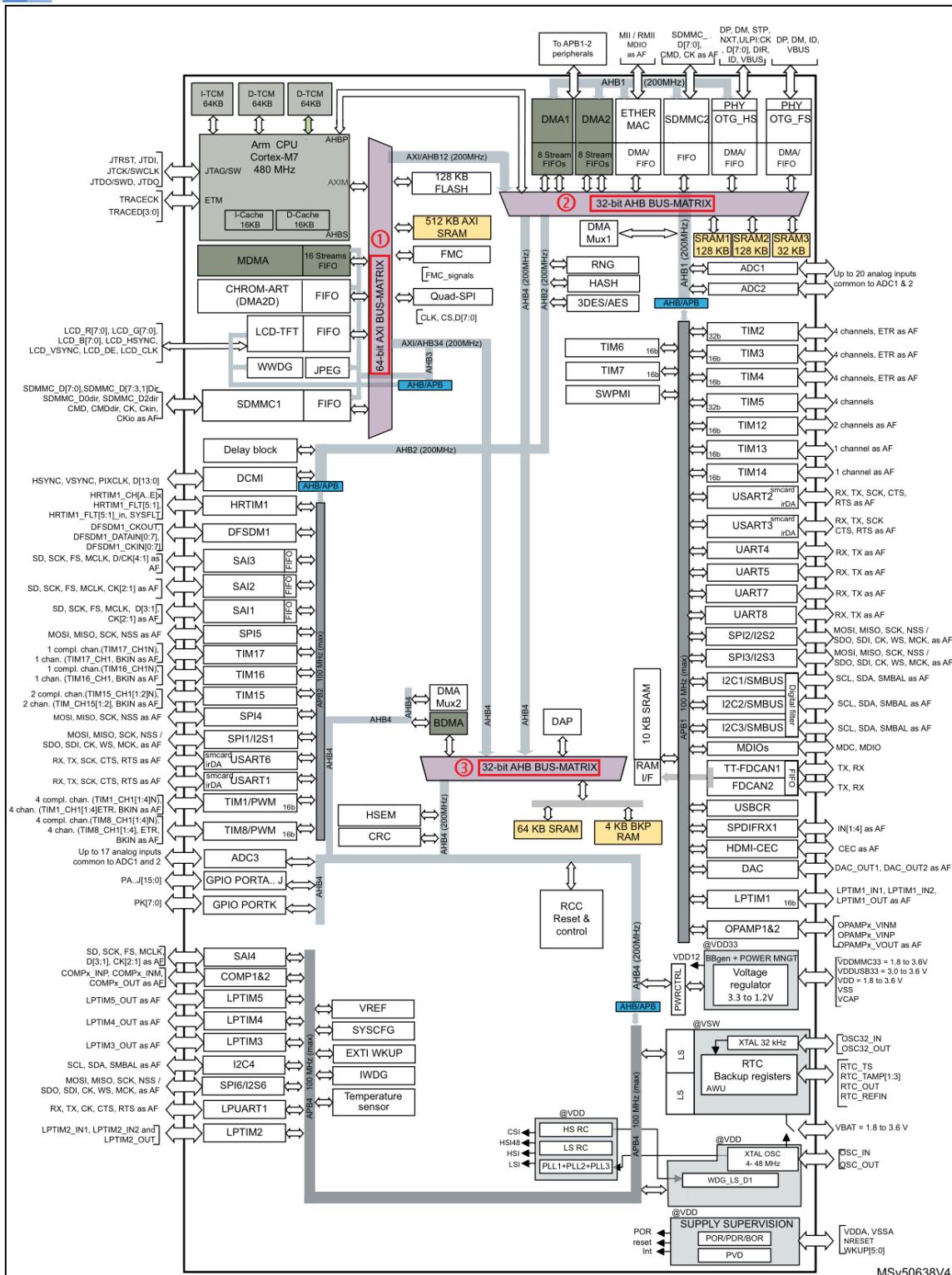


图5.3.2.1 STM32H750系统结构

上图把 STM32H7 的每条总线、总线时钟速度，以及每条总线挂载了那些外设都整理出来，这对我们理解整个 STM32H750 总的框架非常有帮助。

这个图这么复杂，该怎么看？我一般习惯以内核为起点，先看总线，再看总线连接关系，最后再看各个总线分别连接了什么外设，按照这样的顺序查阅，就会清晰很多。

在图 11.3.1 中，我们标记了①②③三个标号，①是 64 位的 AXI 总线矩阵，在 D1 域，②和③都是 32 位的 AHB 总线矩阵，不同的是②在 D2 域，而③是在 D3 域。①②③的关系，我们看

下面这张图更能清楚的表达出来，如图 5.3.2.2 所示。

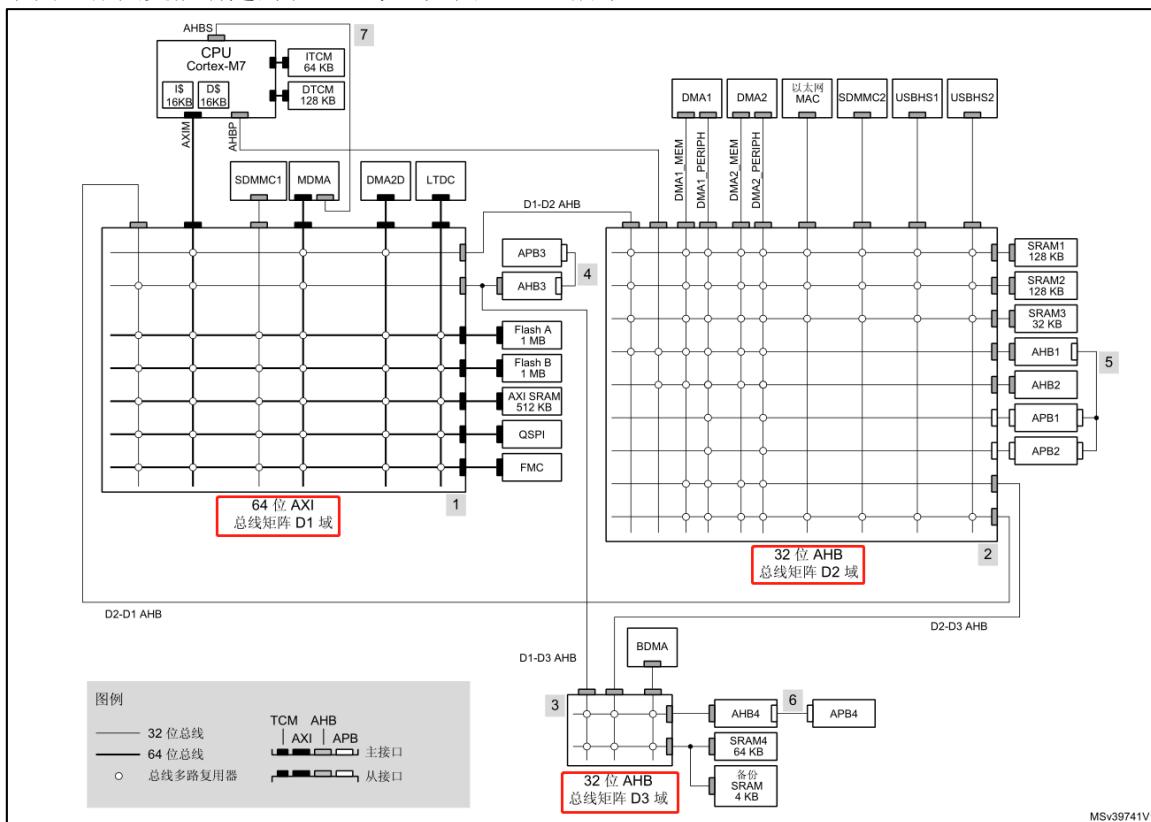


图 5.3.2.2 STM32H750 的总线架构图

从上图中可以看到 STM32H7 的总线架构图比 STM32H7 的系统总架构图有简化些，就是把 STM32H7 的系统总架构图中的总线部分提取出来，大家可以把它们结合一起看。

图 11.3.2 中的三个域就对应了前面的①②③，其中 D1 域是 AXI 总线，D2 和 D3 都是 AHB 总线的。下面分别介绍一下这三个域：

● D1 域

D1 域中的各个外设是挂在 64 位 AXI 总线组成 6×7 的矩阵上。6 个从接口端外接的主控分别是 LTDC、DMA2D、MDMA、SDMMC1、AXIM 和 D2-to-D1 AHB 总线。7 个主接口端外接的从设备分别是 D1-to-D2 AHB 总线、AHB3 总线、Flash A、Flash B、AXI SRAM、QSPI 和 FMC。另外 AHB3 再分支出 APB3 总线。

● D2 域

D2 域中的各个外设是挂在 32 位 AHB 总线组成 10×9 的矩阵上。10 个从接口端外接的主控分别是 D1-to-D2 AHB 总线、AHBP 总线、DMA1、DMA2、以太网 MAC、SDMMC2、USB HS1 和 USB HS2。9 个主接口端外接的从设备分别是 SRAM1、SRAM2、SRAM3、AHB1、AHB2、APB1、APB2、D2-to-D1 AHB 总线和 D2-to-D3 AHB 总线。

● D3 域

D3 域中的各个外设是挂在 32 位 AHB 总线组成 3×2 的矩阵上。3 个从接口端外接的主控分别是 D1-to-D3 AHB 总线、D2-to-D3 AHB 总线和 BDMA。2 个主接口端外接的从设备分别是 AHB4、SRAM4 和备份 SRAM。另外 AHB4 分支出 APB4 总线。

从 STM32H7 的总线架构图里，我们要注意域间总线互连，因为这决定哪些设备（总线）可以相互通信，共有四条域间总线，具体如下：

D2-D1 AHB 总线

该 32 位总线将 D2 域连接到 D1 域中的 AXI 总线矩阵。它使得 D2 域中的总线主设备能够访问 D1 域中的资源（总线从设备），以及通过 D1-D3 AHB 间接访问 D3 域中的资源（总线从设备）。例如：D2 域里面的 DMA2 访问 D1 域里面的 AXI SRAM。

D1-D2 AHB 总线

该 32 位总线将 D1 域连接到 D2 域 AHB 总线矩阵。它使得 D1 域中的总线主设备能够访问 D2 域中的资源(总线从设备)。例如:D1 域里面的 DMA2D 访问 D2 域里面的 SRAM1。

D1-D3 AHB 总线

该 32 位总线将 D1 域连接到 D3 域 AHB 总线矩阵。它使得 D1 域中的总线主设备能够访问 D3 域中的资源(总线从设备)。例如 :D1 域里面的 DMA2D 访问 D3 域里面的 SRAM4。

D2-D3 AHB 总线

该 32 位总线将 D2 域连接到 D3 域 AHB 总线矩阵。它使得 D2 域中的总线主设备能够访问 D3 域中的资源(总线从设备)。例如: D2 域里面的 DMA2 访问 D3 域里面的 SRAM4。

总线结构的内容多且复杂，大家也不要一次阅读就能掌握，把后面的实验都学习了，不懂就回来查阅（请结合官方原文），把后面的实验学习完，就会越来越清晰了。

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 ($2^{32} = 4\text{GB}$)，因此 Cortex M7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：

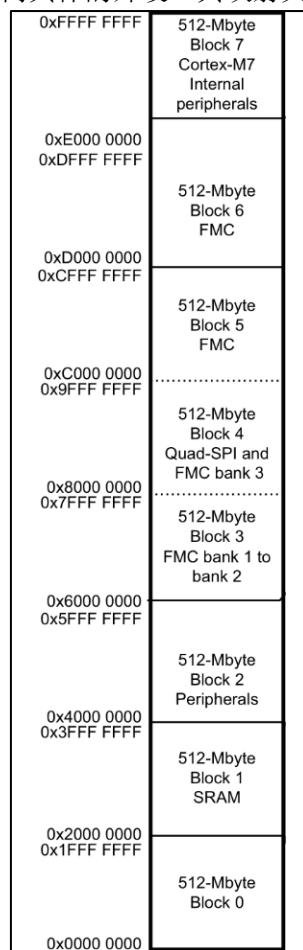


图5.3.3.1 STM32H750存储器映射

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	FMC Bank1&2	0X6000 0000 ~ 0x7FFF FFFF (512MB)
Block 4	FMC Bank3&4	0X8000 0000 ~ 0x9FFF FFFF (512MB)
Block 5	FMC、Quad-SPI 寄存器	0XA000 0000 ~ 0xBFFF FFFF (512MB)
Block 6	FMC Bank5&6	0XC000 0000 ~ 0xDFFF FFFF (512MB)
Block 7	Cortex M7 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	ITCM RAM，只能被 CPU 和 MDMA 访问，不用经过总线矩阵，属于高速的 RAM	0X0000 0000 ~ 0x0000 FFFF (64KB)
	保留	0X0001 0000 ~ 0x000F FFFF
	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	0X0010 0000 ~ 0x0010 EDBF
	保留	0X0011 0000 ~ 0x001F FFFF
	FLASH 存储基于 ITCM 总线接口，不支持写操作，即只读。	0X0020 0000 ~ 0x003F FFFF
	保留	0X0040 0000 ~ 0x07FF FFFF
	用户 FLASH，用于存储用户代码	0X0800 0000 ~ 0x0801 FFFF (128KB)
	保留	0X0802 0000 ~ 0x1FFF EFFF
	16 个字节用于锁定对应的 OTP 数据块	0X1FFF 0000 ~ 0X1FFF 001F
	保留	0X1FFF 0020 ~ 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128KB，这是对于我们使用的STM32H750VBT6来说，如果是其他型号，FLASH就可能不一样了，当然，如果ST喜欢，也是可以随时推出更大容量的STM32H750单片机的，因为这里保留了一大块地址空间。但是ST为降低成本，打造高性能、低价位的芯片，所以设置了这款STM32H750VBT6芯片。因为只有128KB的FLASH，所以我们用到外部FLASH来扩容。STM32H7的出厂固化BootLoader占用差不多60KB FLASH空间。DTCM RAM 可用于运行指令，也可以存储数据。

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	DTCM RAM	0x2000 0000 ~ 0x2001 FFFF (128KB)
	AXI SRAM	0x2400 0000 ~ 0x2407 FFFF (512KB)
	SRAM1	0x3000 0000 ~ 0x3001 FFFF (128KB)
	SRAM2	0x3002 0000 ~ 0x3003 FFFF (128KB)
	SRAM3	0x3004 0000 ~ 0x3004 7FFF (32KB)
	SRAM4	0x3800 0000 ~ 0x3800 FFFF (64KB)
	Backup SRAM	0x3880 0000 ~ 0x3880 0FFF (4KB)

表5.3.3.3 STM32 存储块1的功能划分

可以看到STMH750的SRAM分配基本不是连续的地址空间，我们重点注意各块SRAM的特性

对比，具体如下：

DTCM RAM，用于数据存取，特点是速度很快，和内核一样。

AXI SRAM，位于D1域，数据带宽是64bit，挂在AXI总线上。

SRAM1，SRAM2和SRAM3区，位于D2域，数据带宽是32bit，挂在AHB总线上。

SRAM1：用于D2域中的DMA缓冲，也可以当D1域断电后用于运行程序代码。

SRAM2：用于D2域中的DMA缓冲，也可以用于用户数据存取。

SRAM3：主要用于以太网和USB的缓冲。

SRAM4，位于D3域，数据带宽是32bit，挂在AHB总线上。可以用于D3域中的DMA缓冲，也可以当D1和D2域进入DStandby待机方式后，继续保存用户数据。

Backup SRAM，位于D3域，数据带宽是32bit，挂在AHB总线上。主要用于系统进入低功耗模式后，继续保存数据（Vbat引脚外接电池）。

大家可以参考5.3.2小节，进行理解。

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。该存储块被分成了APB和AHB两部分，其中APB又被分为APB1, APB2, APB3和APB4。AHB分为AHB1, AHB2, AHB3和AHB4。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 D3FF
	保留	0x4000 D400 ~ 0x4000 FFFF
	APB2 总线外设	0x4001 0000 ~ 0x4001 77FF
	保留	0x4001 7800 ~ 0x4001 FFFF
	AHB1 总线外设	0x4002 0000 ~ 0x400B FFFF
	保留	0x400C 0000 ~ 0x4801 FFFF
	AHB2 总线外设	0x4802 0000 ~ 0x4802 2BFF
	保留	0x4802 2C00 ~ 0x4FFF FFFF
	APB3 总线外设	0x5000 0000 ~ 0x5000 3FFF
	保留	0x5000 4000 ~ 0x50FF FFFF
	AHB3 总线外设	0x5100 0000 ~ 0x5200 8FFF
	保留	0x5200 9000 ~ 0x57FF FFFF
	APB4 总线外设	0x5800 0000 ~ 0x5800 6BFF
	保留	0x5800 6C00 ~ 0x5801 FFFF
	AHB4 总线外设	0x5802 0000 ~ 0x5802 67FF
	保留	0x5802 6800 ~ 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各个外设具体挂接在哪个总线上面，大家可以参考前面的 STM32H750系统结构图进行查找对应。

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《STM32H7xx参考手册_V3（中文版）.pdf》相关章节的寄存器描述部分，有详细的描述。

1. 寄存器描述解读

我们以GPIO的ODR寄存器为例，其参考手册的描述如图5.3.4.1所示：

GPIO 端口输出数据寄存器 (GPIOx_ODR) (x = A..K)																①
GPIO port output data register																
偏移地址: 0x14																②
复位值: 0x0000 0000																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
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	

③ 位 31:16 保留，必须保持复位值。
 ④ 位 15:0 ODRy: 端口输出数据位 (Port output data bit) (y = 0..15)
 这些位可通过软件读取和写入。
 注：对于原子置位/复位，通过写入 GPIOx_BSRR 或 GPIOx_BRR 寄存器，可分别置位和/或复位 ODR 位 (x = A..F)。

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~K，说明有11个这样的寄存器。

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《STM32H7xx参考手册_V3 (中文版).pdf》文档的第100页，可知其外设基地址是：0x58020400。那么GPIOB_ODR寄存器的地址就是：0x58020414。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。这里都为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: 0x58020414，假设我们要控制 GPIOB 的 16 个 IO 口都输出 1，则可以写成:

```
* (unsigned int *) (0x58020414) = 0xFFFF;
```

这里我们先要将 0x58020414 强制转换成 unsigned int 类型指针，然后用*对这个指针的值进行设置，从而完成对 GPIOB_ODR 寄存器的写入。

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

```
#define GPIOB_ODR      * (unsigned int *) (0x58020414)
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，stm32h750xx.h 里面使用结构体方式对 STM32H750 的寄存器做了详细映射。

3. 寄存器地址计算

STM32H750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、总线基址（BUS_BASE_ADDR）；2、外设基于总线基址的偏移量（PERIPH_OFFSET）；3、寄存器相对外设基址的偏移量（REG_OFFSET）。可以表示为：

$$\text{寄存器地址} = \text{BUS_BASE_ADDR} + \text{PERIPH_OFFSET} + \text{REG_OFFSET}$$

总线基址（BUS_BASE_ADDR），STM32H750内部的总线基址如表5.3.4.1所示：

总线	基址	偏移量
APB1	0X4000 0000	0X0
APB2	0X4001 0000	0X0001 0000
AHB1	0X4002 0000	0X0002 0000
AHB2	0X4802 0000	0X0802 0000
APB3	0X5000 0000	0X1000 0000
AHB3	0X5100 0000	0X1100 0000
APB4	0X5800 0000	0X1800 0000
AHB4	0X5802 0000	0X1802 0000

表5.3.4.1 总线基址

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。

外设基于总线基址的偏移量（PERIPH_OFFSET），这个不同外设偏移量不一样，我们可以在STM32H750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	基址	偏移量
AHB4 0X5802 0000	GPIOA	0X5802 0000	0X0
	GPIOB	0X5802 0400	0X0400
	GPIOC	0X5802 0800	0X0800
	GPIOD	0X5802 0C00	0X0C00
	GPIOE	0X5802 1000	0X1000
	GPIOF	0X5802 1400	0X1400
	GPIOG	0X5802 1800	0X1800
	GPIOH	0X5802 1C00	0X1C00
	GPIOI	0X5802 2000	0X2000
	GPIOJ	0X5802 2400	0X2400
	GPIOK	0X5802 2800	0X2800

表5.3.4.2 GPIO外设基址及相对总线偏移量

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

知道了外设基址，再在参考手册里面找到具体某个寄存器相对外设基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
AHB4 0X5802 0000	GPIOB 0X5802 0400	GPIOB_MODER	0X5802 0400	0X00
		GPIOB_OTYPER	0X5802 0404	0X04
		GPIOB_OSPEEDR	0X5802 0408	0X08
		GPIOB_PUPDR	0X5802 040C	0X0C
		GPIOB_IDR	0X5802 0410	0X10
		GPIOB_ODR	0X5802 0414	0X14
		GPIOB_BSRR	0X5802 0418	0X18
		GPIOB_LCKR	0X5802 041C	0X1C
		GPIOB_AFRL	0X5802 0420	0X20
		GPIOB_AFRH	0X5802 0424	0X24

表5.3.4.3 GPIOB寄存器相对外设基址的偏移量

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 = AHB4总线基址 + GPIOB外设偏移量 + 寄存器偏移量

所以得到：GPIOB_ODR 地址 = 0X5802 0000 + 0X0400 + 0X14 = 0X5802 0414

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. stm32h750xx.h 寄存器映射说明

STM32H750所有寄存器映射都在stm32h750xx.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2万多行，非常庞大。我们没有必要对该文件进行全面分析，因为很多内容都是相似的，我们只需要知道寄存器是如何被映射的，就可以了，至于寄存器位定义这些内容，知道是怎么回事就可以了。

我们还是以GPIO为例进行说明，看看stm32h750xx.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stm32h750xx.h的映射规则。

stm32h750xx.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
stm32h750xx.h	中断编号定义	定义 IRQn_Type 枚举类型，包含 STM32H750 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 stm32h750xx.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 MODER;      /* GPIO_MODE 寄存器，相对外设基地址偏移量: 0X00 */
    __IO uint32_t OTYPER;     /* GPIO_OTYPER 寄存器，相对外设基地址偏移量: 0X04 */
    __IO uint32_t OSPEEDR;    /* GPIO_OSPEEDR 寄存器，相对外设基地址偏移量: 0X08 */
    __IO uint32_t PUPDR;      /* GPIO_PUPDR 寄存器，相对外设基地址偏移量: 0X0C */
    __IO uint32_t IDR;        /* GPIO_IDR 寄存器，相对外设基地址偏移量: 0X10 */
    __IO uint32_t ODR;        /* GPIO_ODR 寄存器，相对外设基地址偏移量: 0X14 */
    __IO uint32_t BSRR;       /* GPIO_BSRR 寄存器，相对外设基地址偏移量: 0X18 */
    __IO uint32_t LCKR;       /* GPIO_LCKR 寄存器，相对外设基地址偏移量: 0X1C */
    __IO uint32_t AFR[2];     /* GPIO_AFR 寄存器，相对外设基地址偏移量: 0X20 */
} GPIO_TypeDef;
```

GPIO外设基地址定义如下：

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

#define D3_AHB1PERIPH_BASE   (PERIPH_BASE + 0x18020000UL) /* AHB4 总线基地址 */

#define GPIOA_BASE            (D3_AHB1PERIPH_BASE + 0x00000UL) /* GPIOA 基地址 */
#define GPIOB_BASE            (D3_AHB1PERIPH_BASE + 0x04000UL) /* GPIOB 基地址 */
#define GPIOC_BASE            (D3_AHB1PERIPH_BASE + 0x08000UL) /* GPIOC 基地址 */
```

```
#define GPIOD_BASE (D3_AHB1PERIPH_BASE + 0x0C00UL) /* GPIOD 基地址 */
#define GPIOE_BASE (D3_AHB1PERIPH_BASE + 0x1000UL) /* GPIOE 基地址 */
#define GPIOF_BASE (D3_AHB1PERIPH_BASE + 0x1400UL) /* GPIOF 基地址 */
#define GPIOG_BASE (D3_AHB1PERIPH_BASE + 0x1800UL) /* GPIOG 基地址 */
#define GPIOH_BASE (D3_AHB1PERIPH_BASE + 0x1C00UL) /* GPIOH 基地址 */
#define GPIOI_BASE (D3_AHB1PERIPH_BASE + 0x2000UL) /* GPIOI 基地址 */
#define GPIOJ_BASE (D3_AHB1PERIPH_BASE + 0x2400UL) /* GPIOJ 基地址 */
#define GPIOK_BASE (D3_AHB1PERIPH_BASE + 0x2800UL) /* GPIOK 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 */
#define GPIOH ((GPIO_TypeDef *) GPIOH_BASE) /* GPIOH 寄存器地址映射 */
#define GPIOI ((GPIO_TypeDef *) GPIOI_BASE) /* GPIOI 寄存器地址映射 */
#define GPIOJ ((GPIO_TypeDef *) GPIOJ_BASE) /* GPIOJ 寄存器地址映射 */
#define GPIOK ((GPIO_TypeDef *) GPIOK_BASE) /* GPIOK 寄存器地址映射 */
```

以上三部分代码，就完成了STM32H750内部GPIOA~GPIOK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：MODER、OTYPER、OSPEEDR、PUPDR、IDR、ODR、BSRR、LCKR和AFR[2]，每个成员是uint32_t类型，也就是4个字节，这样假设：MODER地址是0的话，OTYPER就是0X04，OSPEEDR就是0X08，PUPDR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（0X5802 0400），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->MODER的地址就是：GPIOB_BASE（0X5802 0400），GPIOB->OTYPER的地址就是：GPIOB_BASE + 0X04（0X5802 0404），GPIOB->OSPEEDR的地址就是：GPIOB_BASE + 0X08（0X5802 0408），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stm32h750xx.h的寄存器映射，我们就介绍到这里。

第六章 认识 HAL 库

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

本章将分为以下几个小节：

- 6.1 初识 STM32 HAL 库
- 6.2 HAL 库驱动包
- 6.3 HAL 库框架结构
- 6.4 如何使用 HAL 库
- 6.5 HAL 库使用注意事项

6.1 初识 STM32 HAL 库

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

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

6.1.1 CMSIS 标准

根据一些调查研究表明，软件开发已经被嵌入式行业公认为最主要的开发成本，为了降低这个成本，ARM 与 Atmel、IAR、KEIL、SEGGER 和 ST 等诸多芯片和软件工具厂商合作，制定了一个将所有 Cortex 芯片厂商的产品的软件接口标准化的标准 CMSIS（Cortex Microcontroller Software Interface Standard）。下面来看 ARM 官方提供的 CMSIS 规范架构图，如图 6.1.1.1 所示：

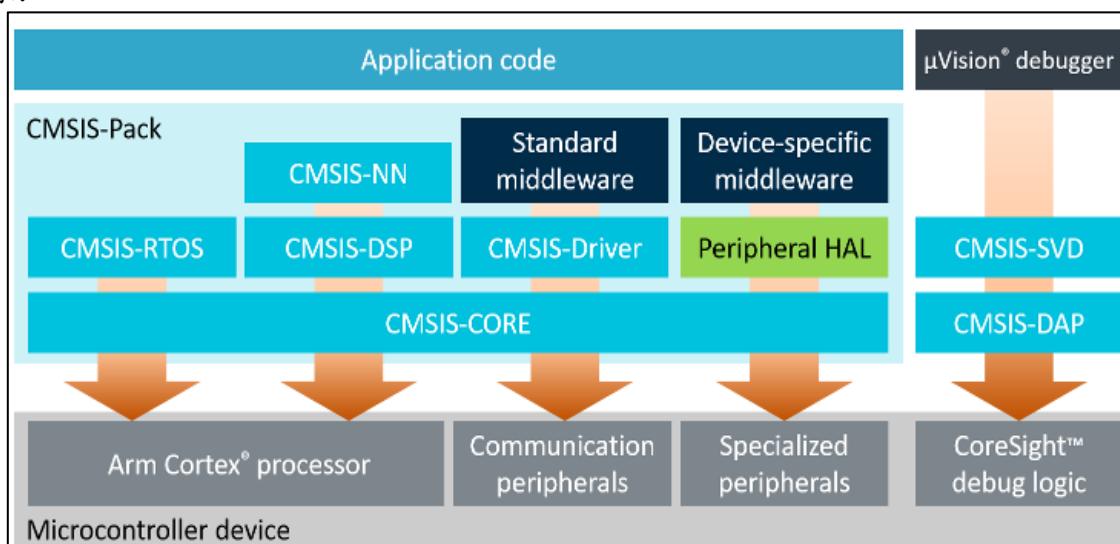


图 6.1.1.1 Cortex 芯片的 CMSIS 分级实现

从图中可以看出这个标准分级明显，从用户程序到内核底层实现做了分层。按照这个分级，HAL 库属于 CMSIS-Pack 中的“Peripheral HAL”层。CMSIS 规定的最主要的 3 个部分为：核内外设访问层（由 ARM 负责实现）、片上外设访问层和外设访问函数（后面两个由芯片

厂商负责实现)。ARM 整合并提供了大量的模版，各厂商根据自己的芯片差异修改模版，这其中包括汇编文件 startup_device.s、system_.h 和 system_.c 这些与初始化和系统相关的函数。

结合 STM32H7 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 6.1.1.2 所示。

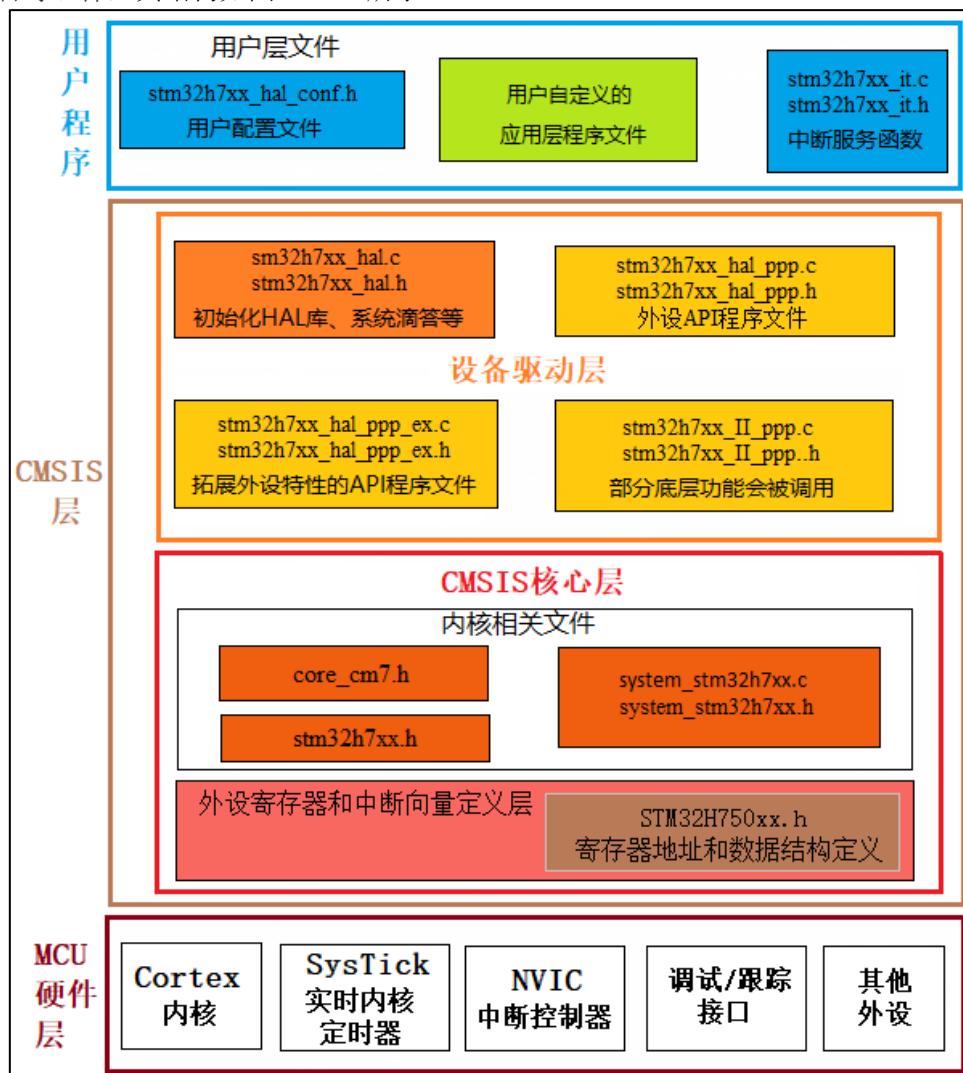


图 6.1.1.2 CMSIS 分级下的 STM32H7 的文件分布

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

6.1.2 HAL 库简介

库函数的引入，大大降低 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 的库函数维护重点已经转移到 HAL 库和 LL 库上，下面我们分别为这三种库作一下简单的介绍。

1. 标准外设库（Standard Peripheral Libraries）

标准外设库（Standard Peripherals Library）是对 STM32 芯片的一个完整的封装，包括所有标准器件外设的器件驱动器，是 ST 最早推出的针对 STM 系列主控的库函数。标准库的设计的初衷是减少用户的程序编写时间，进而降低开发成本。几乎全部使用 C 语言实现并严格按照“Strict ANSI-C”、MISRA-C 2004 等多个 C 语言标准编写。但标准外设库仍然接近于寄存器操

作，主要就是将一些基本的寄存器操作封装成了 C 函数。开发者仍需要关注所使用的外设是在哪个总线之上，具体寄存器的配置等底层信息。

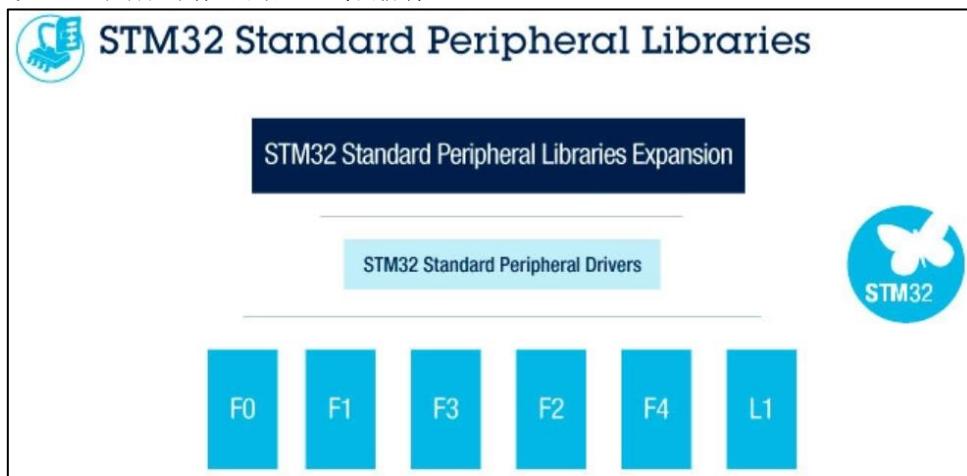


图 6.1.2.1 ST 的标准库函数家族

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，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，更多。但 ST 从 L1 以后的芯片 L0、L4 和 F7 等系列就没有再推出相应的标准库支持包了。

2.HAL 库

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

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

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

3.LL 库

LL 库（Low Layer）目前与 HAL 库捆绑发布，它的设计为比 HAL 库更接近于硬件底层的操作，代码更轻量级，代码执行效率更高的库函数组件，可以完全独立于 HAL 库来使用，但 LL 库不匹配复杂的外设，如 USB 等。所以 LL 库并不是每个外设都有对应的完整驱动配置程序。使用 LL 库需要对芯片的功能有一定的认知和了解，它可以：

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

- 混合使用，和 HAL 库结合使用。

对于 HAL 库和 LL 库的关系，如图 6.1.2.2 Cube 的软件框架所示，可以看出它们设计为彼此独立的分支，但又同属于 HAL 库体系。

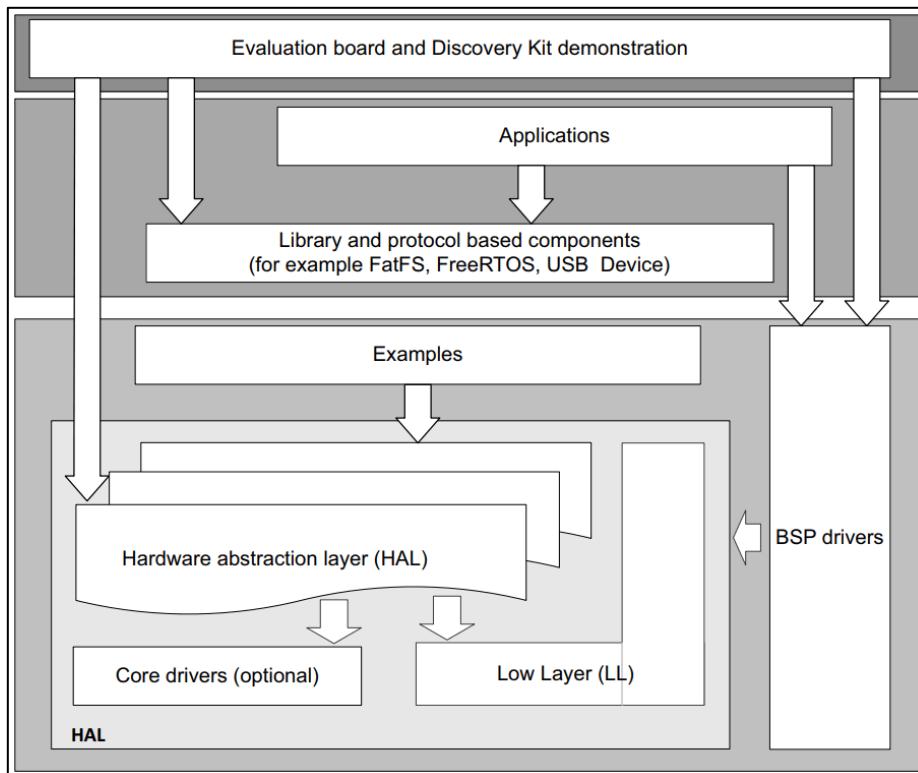


图 6.1.2.2 Cube 的软件框架

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

6.1.3 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。

6.2 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。

6.2.1 如何获取 HAL 库固件包

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

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

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

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

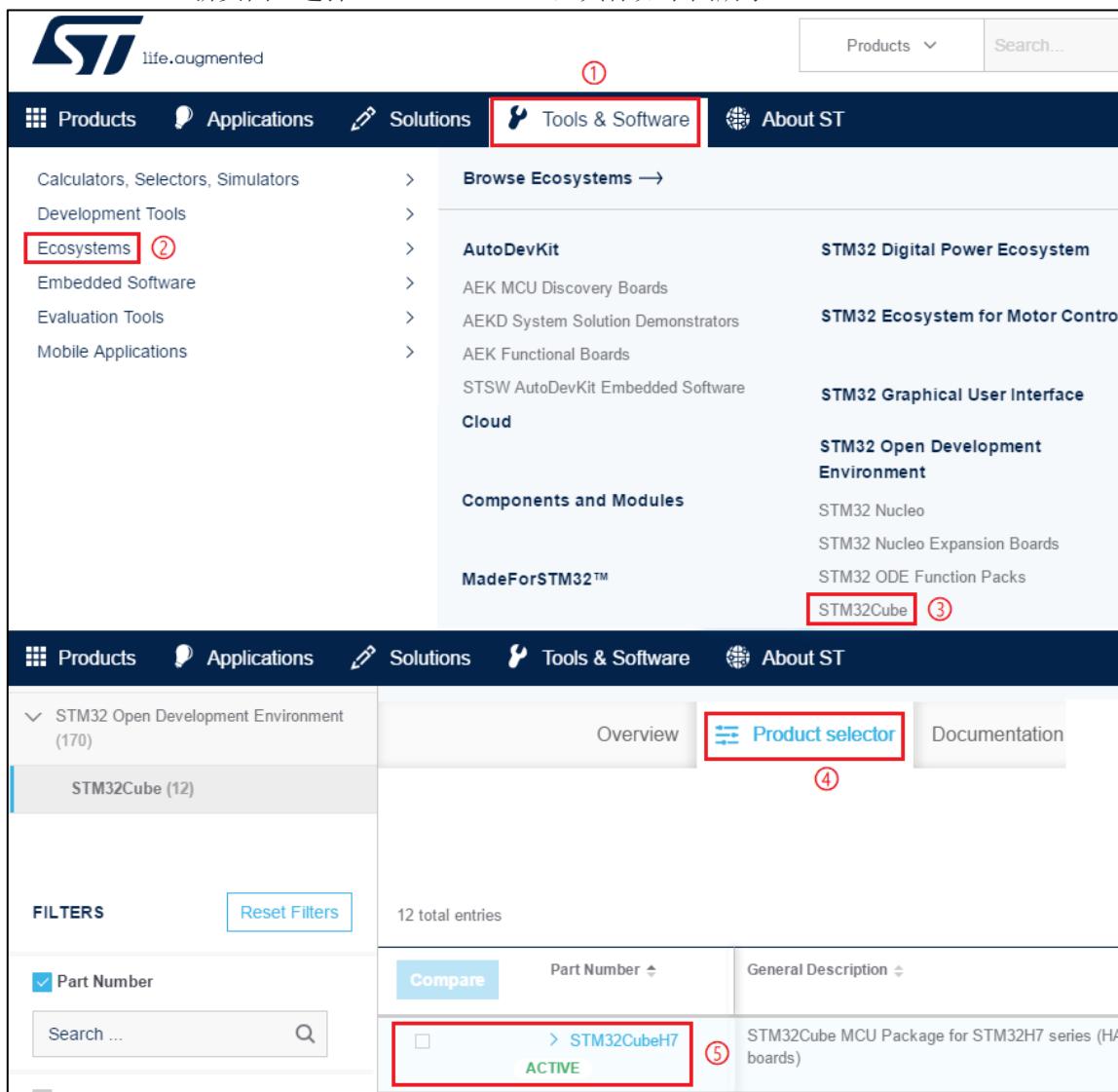


图 6.2.1.1 找到 STM32CubeH7 固件包的下载位置

在展开的页面中选择我们需要和固件，这里展开“STM32CubeH7”即可看到我们需要的 H7 的固件包，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 6.2.1.2 下载 STM32CubeH7 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, STM32CubeH7 固件包**，当前固件包版本是：STM32Cube_FW_H7_V1.6.0。因为现在是 STM32H750 的学习，所以我们准备好的固件包是 H7 的。大家要根据自己学习的芯片，下载对应的固件包。如果需要最新的固件包，大家按照上述的方法到官网重新获取即可。

6.2.2 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 STM32CubeH7 固件包后，我们看看其目录结构，如图 6.2.2.1 所示。

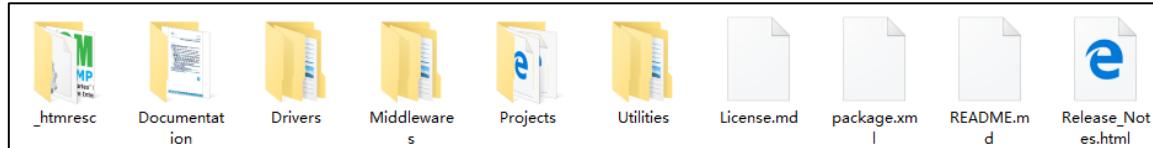


图 6.2.2.1 STM32CubeH7 固件包的目录结构

下面对 STM32CubeH7 固件包进行简要介绍。对于 Documentation 文件夹，里面是一个 STM32CubeH7 英文说明文档，这里我们就不做过多解释。接下来我们通过几个表格依次来介绍一下 STM32CubeH7 中几个关键的文件夹。

(1) Drivers 文件夹

Drivers 文件夹包含 BSP, CMSIS 和 STM32H7xx_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。

	STM32H7xx_HAL_Driver 文件夹	这个文件夹非常重要，它包含了所有的 STM32H7xx 系列 HAL 库头文件和源文件，也就是所有底层硬件抽象层 API 声明和定义。它的作用是屏蔽了复杂的硬件寄存器操作，统一了外设的接口函数。该文件夹包含 Src 和 Inc 两个子文件夹，其中 Src 子文件夹存放的是.c 源文件，Inc 子文件夹存放的是与之对应的.h 头文件。每个.c 源文件对应一个.h 头文件。源文件名称基本遵循 stm32H7xx_hal_ppp.c 定义格式，头文件名称基本遵循 stm32H7xx_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 6.2.2.1 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，这些中间件都是非常成熟的开源解决方案。具体说明请见下表 6.2.2.2:

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。
	Third_Party 子文件夹	STM32_USB_Host_Library 文件夹	USB 主机设备支持包。
		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 6.2.2.2 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。比如我们要查看 STM32H750 相关工程，所以我们直接打开子文件夹 STM32H750B-DK 即可。里面有很多实例，我们都可以用来参考。这里大家注意，每个工程下面都有一个 MDK-ARM 子文件夹，该子文件夹内部会有名称为 Project.uvprojx 的工程文件，我们只需要双击它就可在 MDK 中打开工程。

(4) Utilities 文件夹

该文件夹是一些公用组件，也是主要为 ST 官方的 Demo 板提供的，在我们的例程中使用得不多。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。

6.2.3 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 6.1.1 小节中提到的 CMSIS 标准是一致的, CMSIS 为软件包的内容制定了标准, 包括文件目录的命名和内容构成, CMSIS 版本 5.7.0 的规定软件包目录如表 6.2.3.1 所示:

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

表 6.2.3.1 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_H7_V1.6.0\Drivers\CMSIS”。打开文件夹内容如图 6.2.3.1 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。

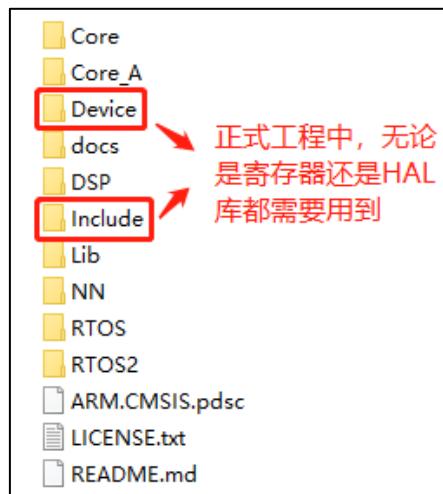


图 6.2.3.1 STM32CubeH7 固件包的 CMSIS 文件夹

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

(1) Device 文件夹

Device 文件夹关键文件介绍如下表 6.2.3.1 所示：

文件	描述
stm32h7xx.h	该文件内容看起来不多，却非常重要，是所有 stm32h7 系列的顶层头文件。使用 STM32H7 任何型号的芯片，都需要包含这个头文件。该文件包含了很多条件定义和常用的枚举变量类型，与宏定义配合，选择性包含某一特定的 STM32H7 系列芯片的头文件。这个文件使我们在使用 STM32H7 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 H7 芯片。
stm32h750xx.h	STM32H750 系列芯片通用的片上外设访问层头文件。只要我们使用 STM32H750 系列芯片，都需要包括这个头文件。这个文件的主要作用是定义声明寄存器以及封装内存操作，以结构体和宏定义标识符的形式。
system_stm32h7xx.c system_stm32h7xx.h	主要是声明和定义了系统初始化函数 SystemInit 以及系统时钟更新函数 SystemCoreClockUpdate。SystemInit 函数的作用是进行时钟系统的一些初始化操作以及中断向量表偏移地址设置，但它并没有设置具体的时钟值，这是与标准库的最大区别，在使用标准库的时候，SystemInit 函数会帮我们配置好系统时钟配置相关的各个寄存器。在启动文件 startup_stm32h750xx.s 中会设置系统复位后，直接调用 SystemInit 函数进行系统初始化。SystemCoreClockUpdate 函数是在系统时钟配置进行修改后，调用这个函数来更新全局变量 SystemCoreClock 的值，变量 SystemCoreClock 是一个全局变量，开放这个变量可以方便我们在用户代码中直接使用这个变量来进行一些时钟运算。
startup_stm32h750xx.s	STM32H750 系列芯片的启动文件，每个系列都有与之对应的启动文件。启动文件的作用主要是进行堆栈的初始化，中断向量表以及中断函数定义等。启动文件有一个很重要的作用就是系统复位后引导进入 main 函数。后面会细讲。

表 6.2.3.1 Device 文件夹关键文件介绍

表 6.1.2.1 列出的文件都是正式工程中必须的文件。固件包的 CMSIS 文件包括了所有 STM32H7 芯片型号的文件，而我们只用到 STM32H750 系列，所以只是挑我们用到的系列文件来讲。

(2) Include 文件夹

Include 文件夹存放了符合 CMSIS 标准的 Cortex-M 内核头文件。想要深入学习内核的朋友可以配合内核相关的手册去学习。对于 STM32H7 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_armlang.h、cmsis_compiler.h、cmsis_version.h、core_cm7.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_cm7.h。

core_cm7.h 是内核底层的文件，由 ARM 公司提供，包含一些 AMR 内核指令，如软件复位，开关中断等功能。今后在需要的例程再去讲解其程序，现在之所以提到，是因为它包含了很重要的头文件 stdint.h。

6.2.4 stdint.h 简介

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

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

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

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

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

6.3 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，

6.3.1 HAL 库文件夹结构

HAL 库头文件和源文件在 STM32Cube 固件包的 STM32H7xx_HAL_Driver 文件夹中，打开该文件夹，如图 6.3.1.1 所示。

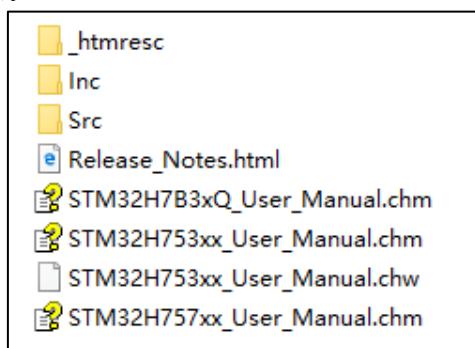


图 6.3.1.1 STM32H7xx_HAL_Driver 文件夹目录结构

STM32H7xx_HAL_Driver 文件夹下的 Src (Source 的简写) 文件夹存放是所有外设的驱动程序源码, Inc (Include 的简写) 文件夹存放的是对应源码的头文件。Release_Notes.html 是 HAL 库的版本更新信息。最后三个是库的用户手册, 这个需要可以去熟悉一下, 查阅起来很方便。

打开 Src 和 Inc 文件夹, 大家会发现基本都是 stm32h7xx_hal_ 和 stm32h7xx_ll_ 开头的.c 和 .h 文件。刚学 HAL 库的朋友可能会说, stm32h7xx_hal_ 开头的是 HAL 库, 我能理解。那么 stm32h7xx_ll_ 开头的文件又是什么? stm32h7xx_ll_ 开头的文件是前面介绍过的 LL 库的文件。

6.3.2 HAL 库文件介绍

HAL 库关键文件介绍如下表 6.3.2.1 所示, 表中 **ppp** 代表任意外设。

文件	描述
stm32h7xx_hal.c stm32h7xx_hal.h	初始化 HAL 库, (比如 HAL_Init, HAL_DeInit, HAL_Delay 等), 主要实现 HAL 库的初始化、系统滴答, HAL 库延时函数、IO 重映射和 DBGMCU 功能等。
stm32h7xx_hal_conf.h HAL 库中本身没有这个文件, 可以自行定义, 也可以直接使用《Inc》文件夹下 stm32f1xx_hal_conf_template.h 的内容作为参考模版	HAL 的用户配置文件, stm32h7xx_hal.h 引用了这个文件, 用来对 HAL 库进行裁剪。由于 HAL 库的很多配置都是通过预编译的条件宏来决定是否使用这一 HAL 库的功能, 这也是当前的主流库如 LWIP /FreeRTOS 等的做法, 无需修改库函数的源码, 通过使能/不使能一些宏来实现库函数的裁剪。
stm32hxx_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。
stm32h7xx_hal_ppp.c stm32h7xx_hal_ppp.h	外设驱动函数。对于所有的 STM32 该驱动名称都相同, ppp 代表一类外设, 包含该外设的操作 API 函数。例如: 当 ppp 为 adc 时, 那么这个文件就是 stm32h7xx_hal_adc.c/h, 可以分别在 Src/Inc 目录下找到。包含该外设的操作 API 函数。其中 stm32h7xx_hal_cortex.c/h 比较特殊, 它是一些 Cortex 内核通用函数声明和定义, 例如中断优先级 NVIC 配置, MPU, 系统软复位以及 Systick 配置等。
stm32h7xx_hal_ppp_ex.c stm32h7xx_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。
stm32h7xx_ll_ppp.c stm32h7xx_ll_ppp.h	LL 库文件, 在一些复杂外设中实现底层功能, 在部分 stm32h7xx_hal_ppp.c 中被调用。

表 6.3.2.1 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, stm32h7xx_hal_ppp (c/h) 中的函数和变量命名也严格按照命名规则, 如表 6.3.2.2 所示的命名规则在大部分情况下都是正确的:

文件名	stm32h7xx_hal_ppp (c/h)	stm32h7xx_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 6.3.2.2 HAL 库函数、变量命名规则

对于 HAL 的 API 函数，常见的有以下几种：

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

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

- 配置和初始化用的结构体

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

- 特殊处理的结构体

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

- 外设句柄结构体

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

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

表 6.3.2.3 HAL 库驱动部分与外设句柄相关的宏

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

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

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

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

表 6.3.2.4 HAL 库驱动中常用的回调函数接口

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

ST 官方给我们提供了快速查找 API 函数的帮助文档。在路径“STM32Cube_FW_H7_V1.6.0\Drivers\STM32H7xx_HAL_Driver”下有几个 chm 格式的文档，根据我们开发板主控芯片 STM32H750VBT6 我们没有找到直接可用的，但可以查看型号接近的：STM32H753xx_User_Manual.chm。双击打开后，可以看到左边目录下有四个主题，我们来查看 Modules。以外设 GPIO 为例，讲一下怎么使用这个文档。点击 GPIO 外设的主题下的 IO operation functions /functions 看看里面的 API 函数接口描述，如图 6.3.2.1 所示。

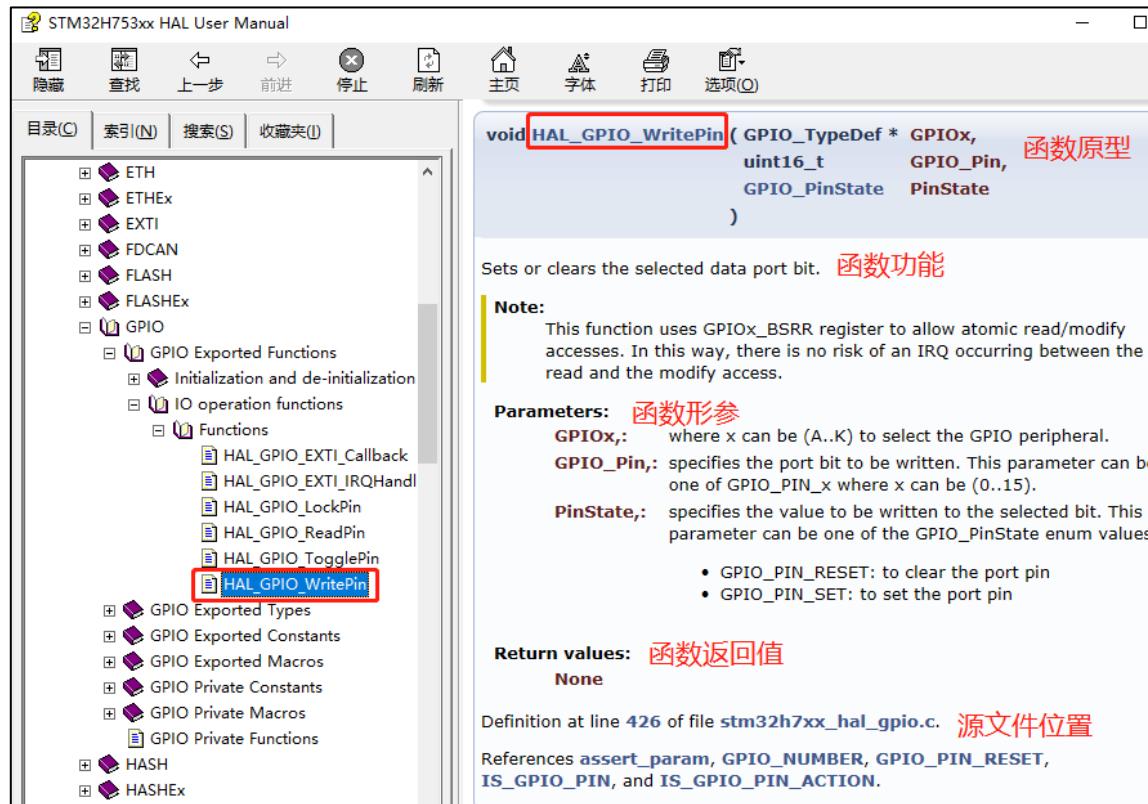


图 6.3.2.1 文档的 API 函数描述

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

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

函数有三个形参：

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

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

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

最后看函数返回值：`None`，没有返回值。

所以最后得出我们要调用的函数是：

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

文档的使用就讲到这。

6.4 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。

6.4.1 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，也不是由这个库能就直接开发。如果我们对芯片的功能不作了解的话，仍然不知道按照怎样的步骤和寻找哪些可用的接口去实现想要实现的功能。

ST 提供芯片使用手册《STM32H7xx 参考手册_V7（英文版）.pdf》告诉我们使用某一外设功能时如何具体地去操作每一个用到的寄存器的细节，后面我们的例程讲解过程也会结合这个手册来分析配置过程。

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

在 HAL 库学习前期，建议以模仿和操作体验为基础，通过例程来学习如何配置和驱动外设。下面根据我们后续要学习的工程梳理出来的基于 CMSIS 的一个 HAL 库应用程序文件结构，帮助读者学习和体会这些文件的组成意义，如下表 6.4.1.1 所示。

类别	文件名	描述	是否必须
用户程序文件	<code>main.c</code>	存放 main 函数，不一定在这个文件	否
	<code>main.h</code>	包含头文件、声明等作用，已删除	否
	<code>stm32f7xx_it.c</code>	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
	<code>stm32f7xx_it.h</code>		否
	<code>stm32f7xx_hal_conf.h</code>	用户配置文件	是
	<code>stm32f7xx_hal_msp.c</code>	回调函数存放文件，已删除	否
设备驱动层	<code>sm32h7xx_hal.c</code>	HAL 库的初始化、系统滴答，HAL 库延时函数等功能	是
	<code>sm32h7xx_hal.h</code>		是
	<code>stm32h7xx_hal_def.h</code>	通用 HAL 库资源定义	是
	<code>stm32h7xx_hal_ppp.c</code>	外设的操作 API 函数文件	是
	<code>stm32h7xx_hal_ppp.h</code>		是
	<code>stm32h7xx_hal_ppp_ex.c</code>	拓展外设特性的 API 函数文件	是
	<code>stm32h7xx_hal_ppp_ex.h</code>		是
	<code>stm32h7xx_ll_ppp.c</code>	LL 库文件，在一些复杂外设中实现底层功能	是
	<code>stm32h7xx_ll_ppp.h</code>		是
CMSIS 核心层	<code>stm32h7xx.h</code>	STM32H7 系列的顶层头文件	是
	<code>stm32h750xx.h</code>	STM32H750 系列片上外设头文件	是
	<code>system_stm32h7xx.c</code>	主要存放系统初始化函数 SystemInit	是
	<code>system_stm32h7xx.h</code>		是
	<code>startup_stm32h750xx.s</code>	启动文件，运行到 main 函数前的准备	是
	<code>core_cm7.h</code>	内核寄存器定义，如 Systick、SCB 等	是
	<code>cmsis_armcc.h</code>	内核头文件，一般都不需要去了解	是
	<code>cmsis_armclang.h</code>		
	<code>cmsis_compiler.h</code>		
	<code>cmsis_version.h</code>		
	<code>mpu_armv7.h</code>		

表 6.4.1.1 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，这将提前告诉一下大家组成我们需要的编译工具链大概需要哪些文件。

6.4.2 HAL 库的用户配置文件

stm32h7xx_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_H7_V1.6.0\Drivers\STM32H7xx_HAL_Driver\Inc 这个路径下提供了一个模板文件《stm32h7xx_hal_conf_template.h》，我们可以直接复制这个文件重命名为 stm32h7xx_hal_conf.h，做一些简单的修改即可，也可以从在官方的例程中直接复制过来。我们一般都直接从官方的模板例程中直接复制过来即可。因为我们的芯片是 STM32H750 系列，所以选择的路径是：STM32Cube_FW_H7_V1.6.0\Projects\STM32H750B-DK\Templates\Template_Project\Inc。

(1) stm32h7xx_hal_conf.h 文件里面的内容不多，对我们来说最重要的是 HSE_VALUE 这个参数，这个参数表示我们的外部高速晶振的频率。这个参数请务必根据我们板子外部焊接的晶振频率来修改，官方默认是 25M。正点原子 STM32H750MINI PRO 开发板外部高速晶振的频率是 8MHZ。**注意事项：**我们要修改这个参数，源码在 99 行，具体修改如下：

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

(2) 还有一个参数就是外部低速晶振频率，这个官方默认是 32.768KHZ，我们开发板的低速晶振也是这个频率，所以不用修改，源码在 128 行。

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

其他源码都可以不作修改，按照默认的配置即可。下面我们再来了解一下其他程序。

(3) 用户配置文件可以用来选择使能何种外设，源码配置在 37 行到 90 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_UART_MODULE_ENABLED
#define HAL_USART_MODULE_ENABLED
#define HAL_WWDG_MODULE_ENABLED
```

我们只要屏蔽某个外设的宏，则这个外设的驱动代码机会被屏蔽，从而不可用。比如我们屏蔽 GPIO 外设的宏，源码在 53 行，屏蔽就是把这个宏注释掉，具体如下。

```
#define HAL_GPIO_MODULE_ENABLED
```

然后打开 stm32h7xx_hal_gpio.c 文件，看到第 118 行。

```
#ifdef HAL_GPIO_MODULE_ENABLED
#include "stm32h7xx_hal_gpio.h"
#endif
```

这是一个条件编译符，与#endif 配合使用。这里的要表达的意思是，只要工程中定义了 HAL_GPIO_MODULE_ENABLED 这个宏，#ifdef 到#endif 之间的程序（119 行到 550 行）就会参与编译，否则不编译。所以只要我们屏蔽了 stm32h7xx_hal_conf.h 文件 53 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。

可以看官方的示范例程，就是通过屏蔽外设的宏的方法来选择使能何种外设。好处就是编译时间会变短，因为屏蔽了没有用的程序，编译时间自然就短了。正点原子的例程选择另外一种方法，就是工程中只保留需要的 stm32h7xx_hal_ppp.c，不需要的不添加到工程里，这样编译时间就不会太长。

(4) 大家看到 stm32h7xx_hal_conf.h 文件的 159 行。

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

宏定义 TICK_INT_PRIORITY 是滴答定时器的优先级。这个优先级很重要，因为如果其它的外设驱动程序的延时是通过滴答定时器提供的时间基准，来实现延时的话，又由于实现方式是滴答定时器对寄存器进行计数，所以当我们在其它中断服务程序里调用基于此时间基准的延

迟函数 HAL_Delay，那么假如该中断的优先级高于滴答定时器的优先级，就会导致滴答定时器中断服务函数一直得不到运行，从而程序卡死在这里。所以滴答定时器的中断优先级一定要比这些中断高。

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

(5) 下面说一下关于断言这个功能，这个功能用来判断函数的形参是否有效，在 HAL 库的 API 里面有用到。这个功能的使能开关代码是一个宏，在源码的 180 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 413 行到 432，代码如下。
/* Exported macro -----
#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 */

#ifndef __cplusplus
}
#endif

#endif /* __STM32H7xx_HAL_CONF_H */
```

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 180 行的注释去掉即可。然后看到源码 423 行的 assert_failed() 这个函数。其实这个函数是需要我们自己实现的，我们把这个函数定义在正点原子提供的 sys.c 文件里面。后面再跟大家讲 sys.c 文件，现在我们把 assert_failed() 这个函数拿出来给大家先讲，assert_failed() 函数的定义在 sys.c 的 176 行到 190 行，具体如下：

```
#ifdef USE_FULL_ASSERT

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

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

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

6.4.3 stm32h7xx_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 134 行到 172 行，简化函数如下：

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

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

    /* 初始化硬件 */
    HAL_MspInit();

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

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

- 1) 设置 NVIC 优先级分组为 4。
- 2) 配置滴答定时器每 1ms 产生一个中断。
- 3) 在这个阶段，系统时钟还没有配置好，因此系统还是默认使用内部高速时钟源 HSI 在跑程序。对于 H7 来说，HSI 的主频是 64MHZ。所以如果用户不配置系统时钟的话，那么系统将会使用 HIS 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 stm32h7xx_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 145 行改为如下代码：

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);
```

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

2. HAL_DeInit() 函数

源码在 179 行到 214 行，函数如下：

```
HAL_StatusTypeDef HAL_DeInit(void)
{
    /* 复位所有外设 */
    HAL_RCC_AHB3_FORCE_RESET();
    HAL_RCC_AHB3_RELEASE_RESET();
    HAL_RCC_AHB1_FORCE_RESET();
```

```

    __HAL_RCC_AHB1_RELEASE_RESET();
    __HAL_RCC_AHB2_FORCE_RESET();
    __HAL_RCC_AHB2_RELEASE_RESET();

    __HAL_RCC_AHB4_FORCE_RESET();
    __HAL_RCC_AHB4_RELEASE_RESET();

    __HAL_RCC_APB3_FORCE_RESET();
    __HAL_RCC_APB3_RELEASE_RESET();

    __HAL_RCC_APB1L_FORCE_RESET();
    __HAL_RCC_APB1L_RELEASE_RESET();

    __HAL_RCC_APB1H_FORCE_RESET();
    __HAL_RCC_APB1H_RELEASE_RESET();

    __HAL_RCC_APB2_FORCE_RESET();
    __HAL_RCC_APB2_RELEASE_RESET();

    __HAL_RCC_APB4_FORCE_RESET();
    __HAL_RCC_APB4_RELEASE_RESET();

/* 对底层硬件初始化进行复位 */
HAL_MspDeInit();

/* 返回函数状态 */
return HAL_OK;
}
}

```

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

- 1) 复位了 AHB1、AHB2、AHB3、AHB4、APB1L、APB1H、APB2、APB3、APB4 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 stm32h7xx_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 254 行到 302 行，简化函数如下：

```

weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /* uwTickFreq 是个枚举类型，如果检测到 uwTickFreq 为零，则返回 */
    if((uint32_t)uwTickFreq == 0UL)
    {
        return HAL_ERROR;
    }

    /* 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;
    }
#endif

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

```

```

    return HAL_ERROR;
}

/* 返回函数状态 */
return HAL_OK;
}

```

该函数用于初始化滴答定时器的时钟基准，主要功能如下：

- 1) 配置滴答定时器 1ms 产生一次中断。
- 2) 配置滴答定时器的中断优先级。
- 3) 该函数是 `_weak` 定义的“弱函数”，用户可以重新定义该函数。

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

4. 滴答定时器相关的函数

源码在 331 行到 463 行，相关函数如下：

```

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

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

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

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

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

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

```

```

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

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

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

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

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

```

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

5. HAL 库版本相关的函数

源码在 465 行到 517 行，相关函数声明如下：

```

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

```

这些函数了解一下就好了，用得不多。

6. 芯片内部电压基准相关函数

源码在 519 行到 602 行，函数声明如下：

```

void HAL_SYSCFG_VREFBUF_VoltageScalingConfig(uint32_t VoltageScaling);
void HAL_SYSCFG_VREFBUF_HighImpedanceConfig(uint32_t Mode);
void HAL_SYSCFG_VREFBUF_TrimmingConfig(uint32_t TrimmingValue);
HAL_StatusTypeDef HAL_SYSCFG_EnableVREFBUF(void);
void HAL_SYSCFG_DisableVREFBUF(void);

```

`HAL_SYSCFG_VREFBUF_VoltageScalingConfig` 函数用于配置芯片内部电压基准大小，形参有四个值可以选择：

- 1) 当形参为 `SYSCFG_VREFBUF_VOLTAGE_SCALE0` 时，

电压输出基准为 2.048V，条件是 VDDA >= 2.4V。

- 2) 当形参为 SYSCFG_VREFBUF_VOLTAGE_SCALE1 时，
电压输出基准为 2.5V，条件是 VDDA >= 2.8V。
- 3) 当形参为 SYSCFG_VREFBUF_VOLTAGE_SCALE2 时，
电压输出基准为 1.5V，条件是 VDDA >= 1.8V。
- 4) 当形参为 SYSCFG_VREFBUF_VOLTAGE_SCALE3 时，
电压输出基准为 1.8V，条件是 VDDA >= 2.1V。

HAL_SYSCFG_VREFBUF_HighImpedanceConfig 函数用于配置芯片内部电压是否与 VREF+ 引脚连接，即是否选择高阻抗模式，有两个形参选择：

- 1) 当形参为 SYSCFG_VREFBUF_HIGH_IMPEDANCE_DISABLE，表示导通。
- 2) 当形参为 SYSCFG_VREFBUF_HIGH_IMPEDANCE_ENABLE，表示高阻抗，即不导通。

HAL_SYSCFG_VREFBUF_TrimmingConfig 函数用于调整校准内部电压基准。

HAL_SYSCFG_EnableVREFBUF 函数用于使能内部电压基准参考。

HAL_SYSCFG_DisableVREFBUF 函数用于禁止内部电压基准参考。

7. 以太网 PHY 接口选择函数

源码在 605 行到 619 行，函数声明如下：

```
void HAL_SYSCFG_ETHInterfaceSelect(uint32_t SYSCFG_ETHInterface);
```

该函数用于以太网 PHY 接口的选择，可以是 MII 或 RMII 接口。

8. HAL_SYSCFG_AnalogSwitchConfig() 函数

源码在 622 行到 650 行，函数声明如下：

```
void HAL_SYSCFG_AnalogSwitchConfig(uint32_t SYSCFG_AnalogSwitch ,  
                                     uint32_t SYSCFG_SwitchState );
```

当 PA0、PA1、PC2、PC3 引脚复用为 ADC 的时候，还有一组对应的可选引脚 PA0_C、PA1_C、PC2_C、PC3_C。该函数的作用就是切换这些可选的引脚。关于这个不理解，请参考图 6.4.3.1。该函数操作了 SYSCFG_PMCR 寄存器，关于该寄存器请查阅参考手册。

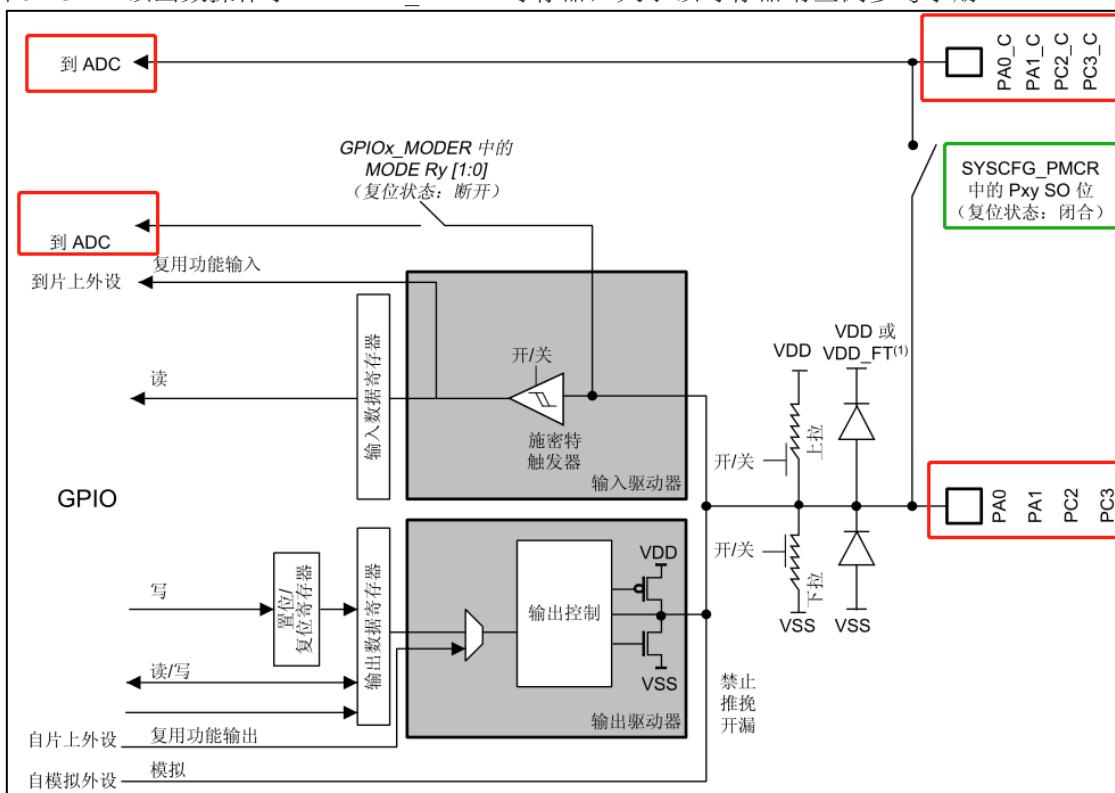


图 6.4.3.1 连接到 ADC 输入的模拟输入

9. Booster 的使能和禁止函数（用于 ADC）

源码在 653 行到 676 行，函数声明如下：

```
void HAL_SYSCFG_EnableBOOST(void); /* 使能 Booster */
void HAL_SYSCFG_DisableBOOST(void); /* 禁止 Booster */
```

如果使能 Booster，当供电电压低于 2.7V 时，能够减少模拟开关总的谐波失真。这样就使得模拟开关的性能和供电正常的情况时一样，能够正常工作。

10. HAL_SYSCFG_CM7BootAddConfig()函数

源码在 680 行到 712 行，函数如下：

```
void HAL_SYSCFG_CM7BootAddConfig(uint32_t BootRegister, uint32_t BootAddress)
```

该函数用于配置程序启动模式，BOOT=0 或者 BOOT=1，来选择启动地址。详细的内容请看第九章的 9.1 小节。

11. IO 补偿、低功耗、EXTI 等相关函数

IO 补偿、低功耗、EXTI 等相关函数，这里先不进行讲解了，后面用到再进行说明。

6.4.4 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 6.4.4.1 所表示的业务逻辑：

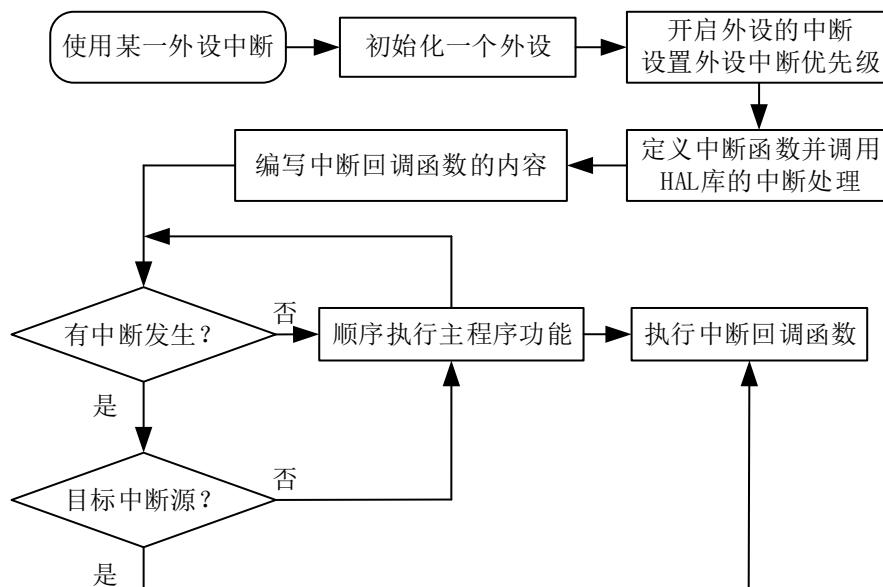


图 6.4.4.1 HAL 驱动中断处理流程

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_HandleTypeDef 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。

6.4.5 正点原子对 HAL 库用法的个性化修改

前面按 ST 官方建议的 HAL 库的使用方法给介绍了一个 HAL 库。

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

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

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

6.5 HAL 库使用注意事项

本小节根据经验跟大家讲述一些关于 HAL 库使用的注意事项，供读者参考。

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

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

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

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

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

第七章 HAL 库版本 MDK 工程创建

在上一章的内容中简单介绍了 HAL 库，本章将详细讲解使用 HAL 库创建 MDK 工程的步骤，本章创建好的 MDK 工程存放路径为 A 盘→4，程序源码→标准例程→实验 0 基础入门实验→实验 0-2 新建工程实验-HAL 库版本，读者在创建工程时若遇到问题，可打开该工程进行对比。

本章分为如下几个小节：

7.1 HAL 库版本 MDK 工程创建

7.2 下载验证

7.1 HAL 库版本 MDK 工程创建

HAL 库版本 MDK 工程的创建可分为四个步骤，如下：

- ①：构建工程文件夹
- ②：添加文件到工程目录
- ③：创建 MDK 工程
- ④：配置 MDK 工程

7.1.1 构建工程文件夹

构建工程文件夹前需要先确定工程文件夹结构，建议初学者使用本书配套实验例程的文件夹结构，该工程文件夹结构也是正点原子所有实验例程的工程文件夹结构，读者可任意打开一个本书配到的实验例程根目录，如下图所示：

```
实验0-2 新建工程实验-HAL库版本/
|-- Drivers
|-- Middlewares
|-- Output
|-- Projects
|-- User
|-- keillkill.bat
`-- readme.txt
```

图 7.1.1.1 工程文件夹结构

该工程文件夹结构清晰，其中各个子文件夹的作用如下表所示：

名称	描述
Drivers	存放于硬件相关的驱动文件
Middlewares	存放正点原子提供的中间组件或第三方中间组件
Output	存放工程编译的输出文件
Projects	存放 MDK 等工程文件
User	存放用户编写的代码文件，例如 main.c 或其他应用程序

表 7.1.1.1 工程子文件夹介绍

接下来，就按照上面介绍的工程文件夹结构构件工程文件夹，如下图所示：

```
新建工程实验-HAL库版本/
|-- Drivers
|-- Middlewares
|-- Output
|-- Projects
|-- User
|-- keillkill.bat
`-- readme.txt
```

图 7.1.1.2 构建工程文件夹

上图中还另外添加了 keillkill.bat 的脚本文件和 readme.txt 文本文件，其中 keillkill.bat 脚本文

件用于清除工程文件夹中编译工程产生的中间文件，方便工程的存档和传输，readme.txt 文本文件主要是对该工程的描述，包括工程功能、使用的硬件资源等信息。

7.1.2 添加文件到工程目录

在该步骤中，涉及了五个文件夹，如下：

① : Drivers 文件夹

Drivers 文件夹用于存放与硬件相关的驱动文件，如下表所示：

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)、stm32h7xx.h 等
STM32H7xx_HAL_Driver	存放 HAL 库驱动代码源文件
SYSTEM	存放正点原子系统级核心驱动代码，如 sys.c、delay.c 和 usart.c

表 7.1.2.1 Drivers 文件夹介绍

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、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)、stm32h7xx.h 等各种头文件。该文件夹我们可以直接从 STM32CubeH7 固件包（路径：A 盘→8, STM32 参考资料→1, STM32CubeH7 固件包）里面拷贝，不过由于固件包里面的 CMISIS 兼容了太多芯片，导致非常大（300 多 MB），因此我们根据实际情况，对其进行大幅精简，精简后的 CMSIS 文件夹大小为 2.3MB 左右。精简后的 CMSIS 文件夹大家可以在：A 盘→4, 程序源码→1，标准例程-HAL 库版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

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

STM32H7xx_HAL_Driver 文件夹，用于存放 ST 提供的 H7xx HAL 库驱动代码。该文件夹我们可以直接从 STM32CubeH7 固件包里面拷贝。直接拷贝“STM32CubeH7 固件包→Drivers”路径下的“STM32H7xx_HAL_Driver”文件夹到我们工程的 Drivers 下，只保留 Inc 和 Src 文件夹即可。

执行完以上操作后，Drivers 文件夹最终结构如下图所示：



图 7.1.2.1 Drivers 文件夹

②: Middlewares 文件夹

Middlewares 文件夹用于存放正点原子提供的中间组件（USMART、MALLOC 等）和第三方中间组件（FatFs、FreeRTOS 等）在初次创建工程时，并不会用到这些中间组件，因此留空即可。

③: Output 文件夹

Output 文件夹用户存放 MDK 工程编译后产生的二进制文件和编译过程文件等，这些文件均由 MDK 软件在进行工程编译的时候产生，留空即可。

④: Projects 文件夹

Projects 文件夹主要用于存放 MDK 的工程文件，在 Projects 文件夹中创建一个 MDK-ARM 的空文件夹，用于下文创建 MDK 工程时，存放 MDK 的工程文件，如下图所示：

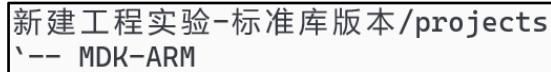


图 7.1.2.2 Projects 文件夹

⑤: User 文件夹

User 文件夹用于存放用户编写的应用代码，例如 main.c 等，main.c 文件中提供了 main() 函数（必须有该函数），对于初次创建的工程，可以从“实验 0-2 新建工程实验-HAL 库版本”的工

程中复制。另外，在该文件夹下还会存放 `stm32f4xx_hal_conf.h`、`stm32f4xx_it.c` 和 `stm32f4xx_it.h` 这三个文件，其中 `stm32f4xx_hal_conf.h` 文件主要用于配置 HAL 库，其他两个文件主要是用于管理程序中的中断服务函数，这三个文件直接从“实验 0-2 新建工程实验-HAL 库版本”的工程中复制即可。

添加好文件的 User 文件夹，如下图所示：

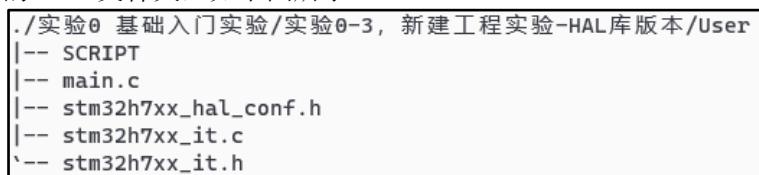


图 7.1.2.3 User 文件夹

7.1.3 创建 MDK 工程

在完成了以上两大步骤后，接下来可以开始创建 MDK 工程了，首先打开 MDK 软件，依次点击菜单栏中的 Project→New uVision Project，如下图所示：

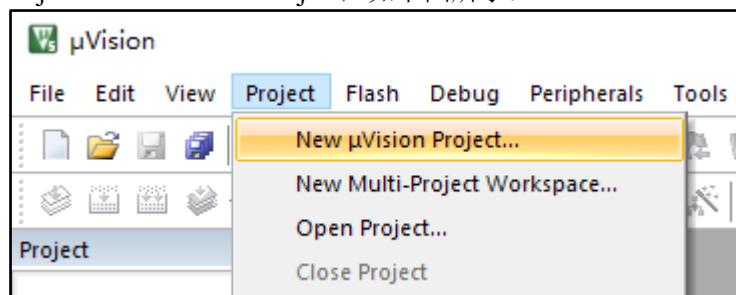


图 7.1.3.1 新建 MDK 工程

随后在弹出的窗口中选择工程文件的保存路径和工程文件名，如下图所示：

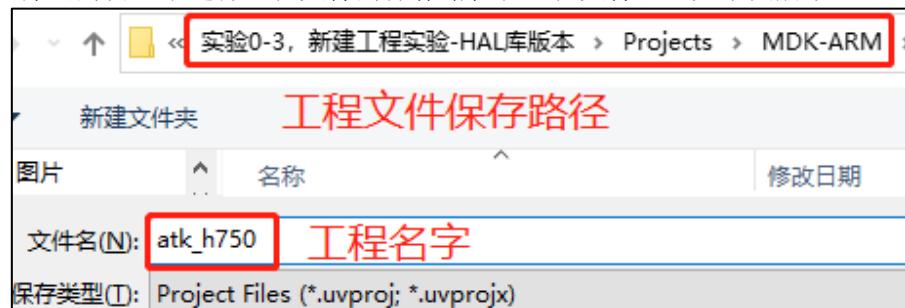


图 7.1.3.2 保存工程文件

之后，弹出器件选择对话框，如图 8.1.2.3 所示。因为 M100Z-M7 最小系统板 STM32H750 版所使用的 STM32 型号为 STM32H750VBT6，所以我们选择：STMicroelectronics→STM32H7 Series→STM32H750→STM32H750VBTx（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.STM32H7xx_DFP.2.5.0.pack 这个安装包后重试）。

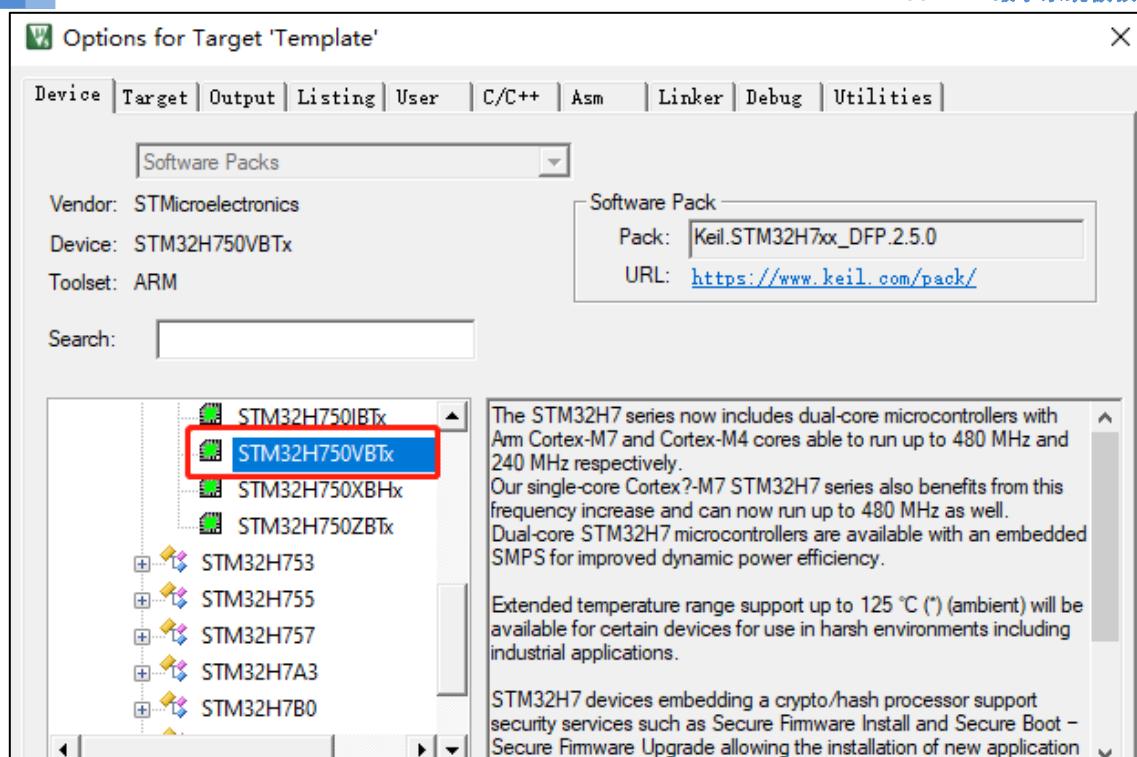


图 7.1.3.3 选择 MCU 型号

点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，如下图所示：

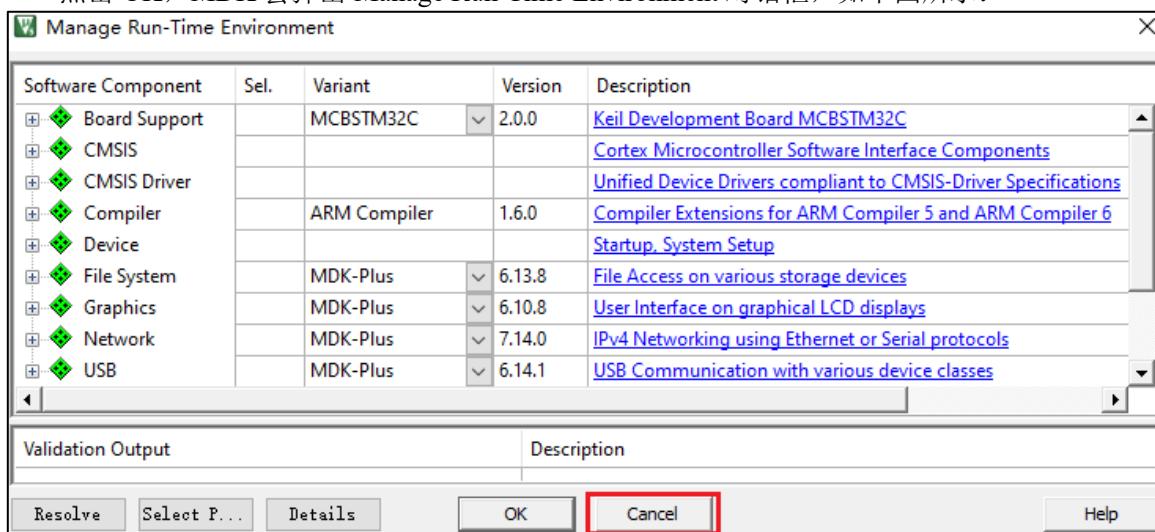


图 7.1.3.4 Manage Run-Time Environment 界面

这是 MDK5 新增的一个功能，在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，不过这里我们不做介绍。所以在图 8.1.2.4 所示界面，我们直接点击 Cancel，即可，得到如下图所示界面：

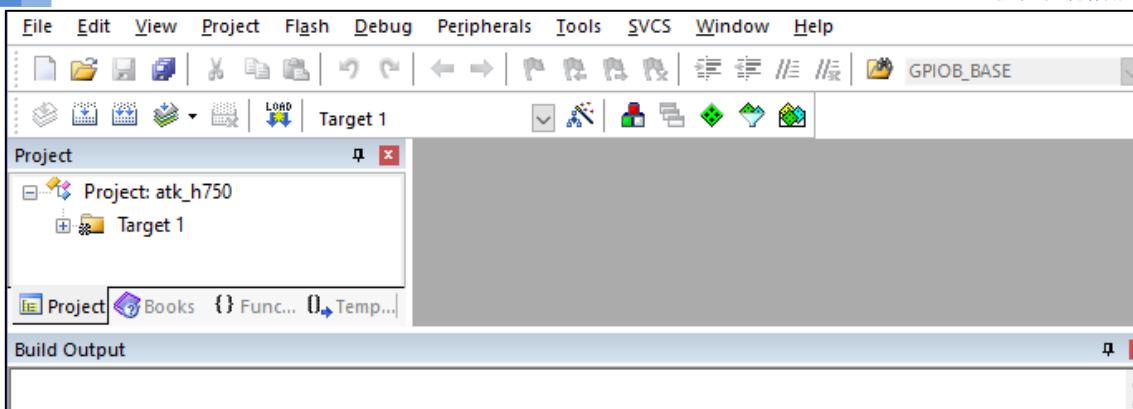


图 7.1.3.5 工程初步建立

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），如下图所示：



图 7.1.3.6 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 7.1.3.1 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 7.1.3.1 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。

7.1.4 添加文件

本节将分 5 个步骤：1，设置工程名和分组名；2，添加启动文件；3，添加 SYSTEM 源码 4，添加 User 源码；5，添加 STM32H7xx_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，如下图所示：

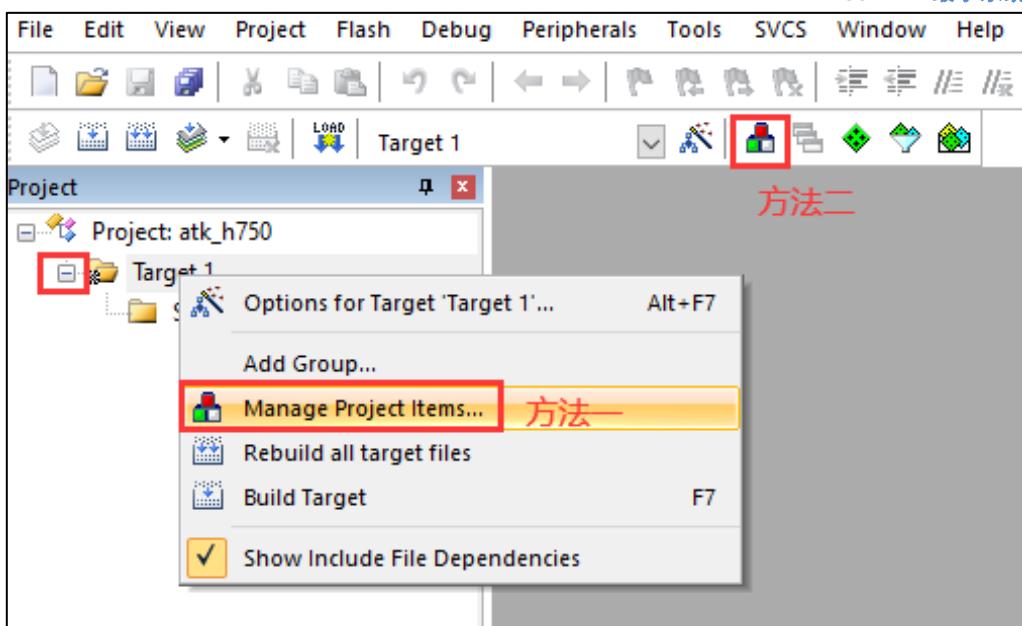


图 7.1.4.1 进入工程管理界面

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），如下图所示：

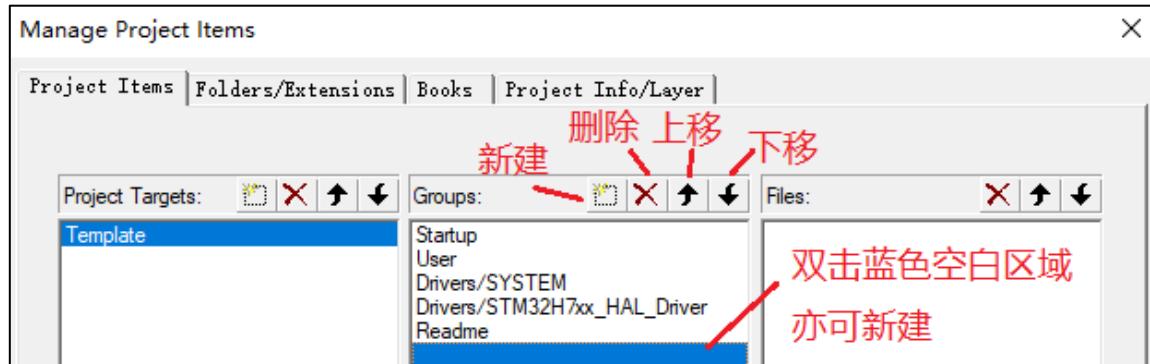


图 7.1.4.2 设置工程名和分组名

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下图所示：

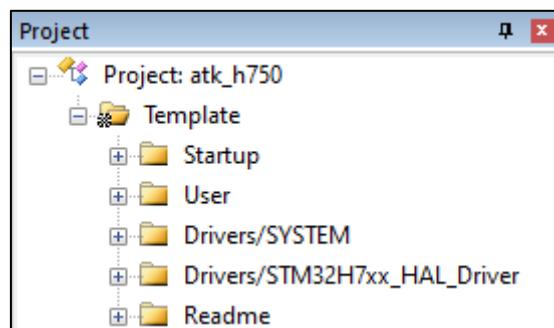


图 7.1.4.3 设置成功

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

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

如：User 分组对应 User 文件夹里面的源码，Drivers/SYSTEM 分组，对应 Drivers/SYSTEM 文件夹里面的源码，Drivers/BSP 分组对应 Drivers/BSP 文件夹里面的源码等。

2. 添加启动文件

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

启动文件由 ST 官方提供，存放在 STM32CubeH7 软件包的：Drivers→CMSIS→Device→ST→STM32H7xx→Source→Templates→arm 文件夹下。因为我们开发板使用的是 STM32H750VBT6，对应的启动文件为：startup_stm32h750xx.s，为了节省空间，在精简版 CMSIS 文件夹里面我们把其他启动文件都删了。

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

这了我们使用方法 1 添加（路径：实验 0-3，新建工程实验-HAL 库版本\Drivers\CMSIS\Device\ST\STM32H7xx\Source\Templates\arm），如下图所示：

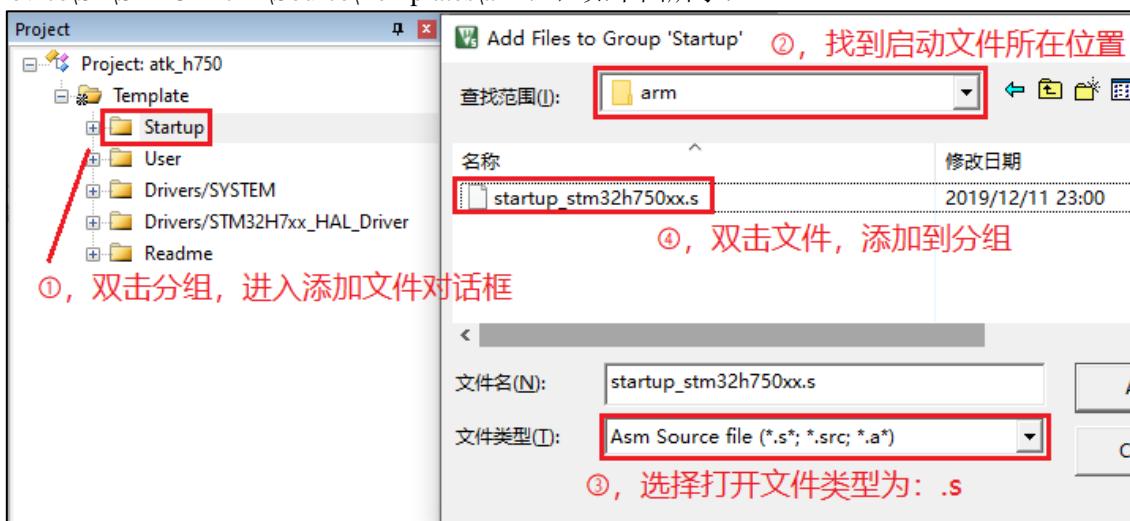


图 7.1.4.4 双击分组添加启动文件 (startup_stm32h750xx.s)

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，得到工程分组如下图所示：

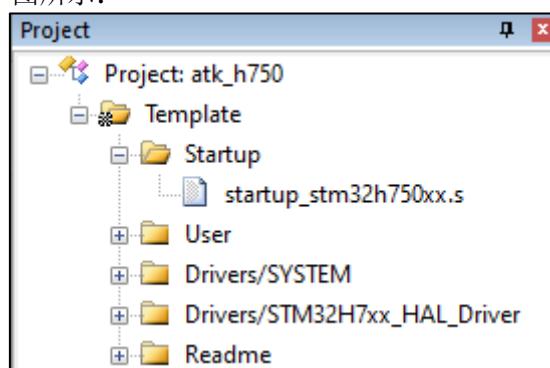


图 7.1.4.5 启动文件添加成功

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，如下图所示：

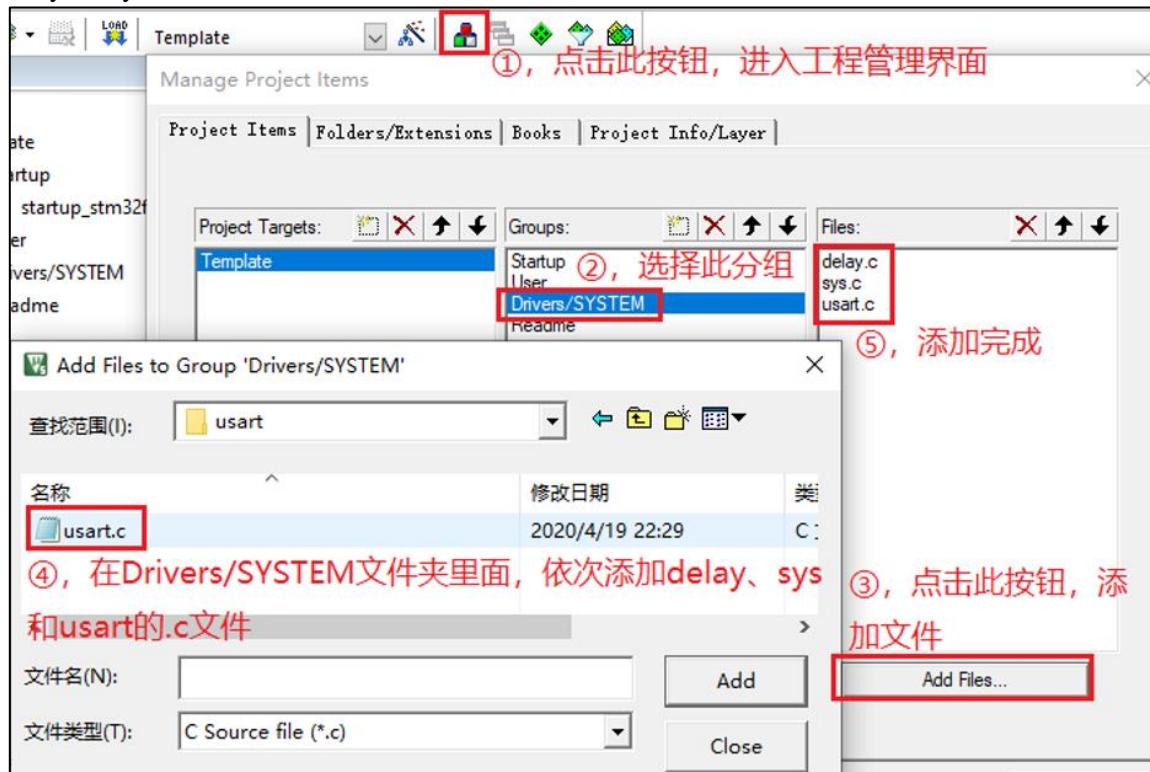


图 7.1.4.6 添加 SYSTEM 源码

注意：这些源码都是在第 7.1.1 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，如下图所示：

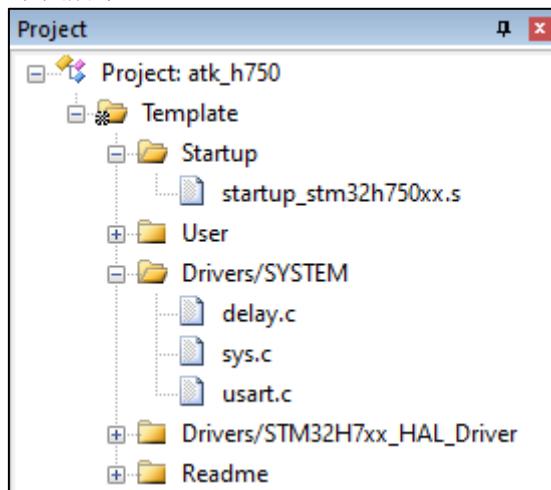


图 7.1.4.7 SYSTEM 源码添加完成

4. 添加 User 源码

这里我们在工程管理界面（方法 2）进行 User 源码添加。点击：按钮，进入工程管理界面，选中 User 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32h7xx_it.c 和 system_stm32h7xx.c 到该分组下，如下图所示：

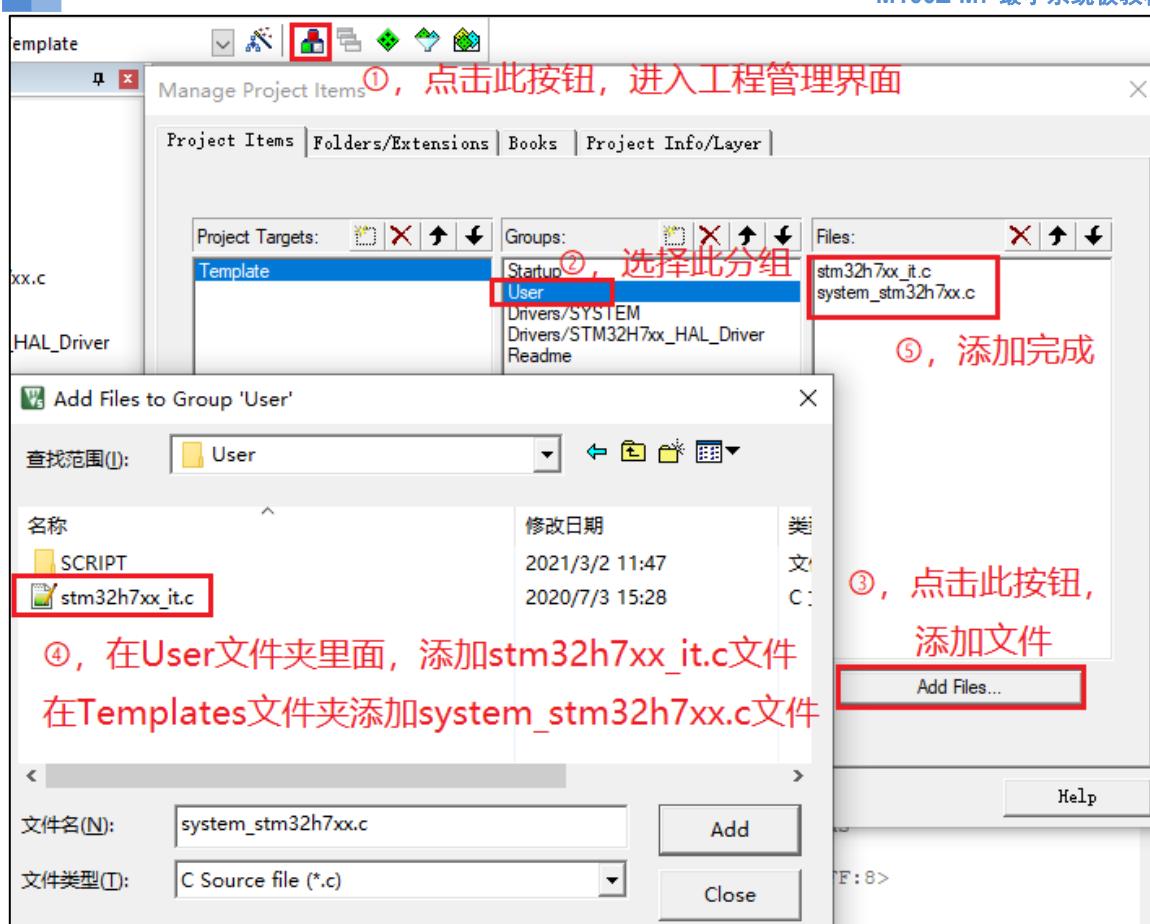


图 7.1.4.8 添加 User 源码

注意：这些源码都是在第 7.1.1 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，如下图所示：

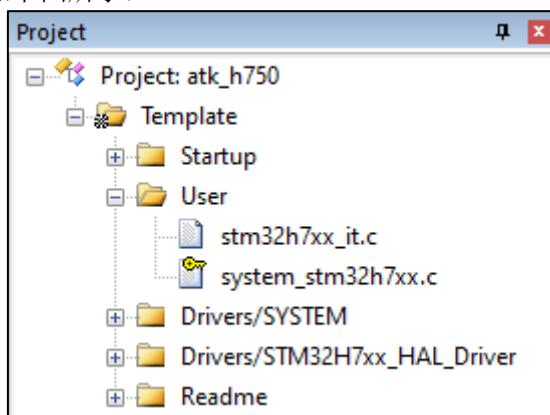


图 7.1.4.9 User 源码添加完成

5. 添加 STM32H7xx_HAL_Driver 源码

接下来我们往 Drivers/STM32H7xx_HAL_Driver 分组里添加文件。点击：按钮，进入工程管理界面，选中 Drivers/STM32H7xx_HAL_Driver 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32h7xx_hal.c、stm32h7xx_hal_cortex.c、stm32h7xx_hal_dma.c、stm32h7xx_hal_gpio.c、stm32h7xx_hal_pwr.c、stm32h7xx_hal_pwr_ex.c、stm32h7xx_hal_rcc.c、stm32h7xx_hal_rcc_ex.c、stm32h7xx_hal_uart.c、stm32h7xx_hal_uart_ex.c、stm32h7xx_hal_usart.c 和 stm32h7xx_hal_usart_ex.c 到该分组下，如下图所示：

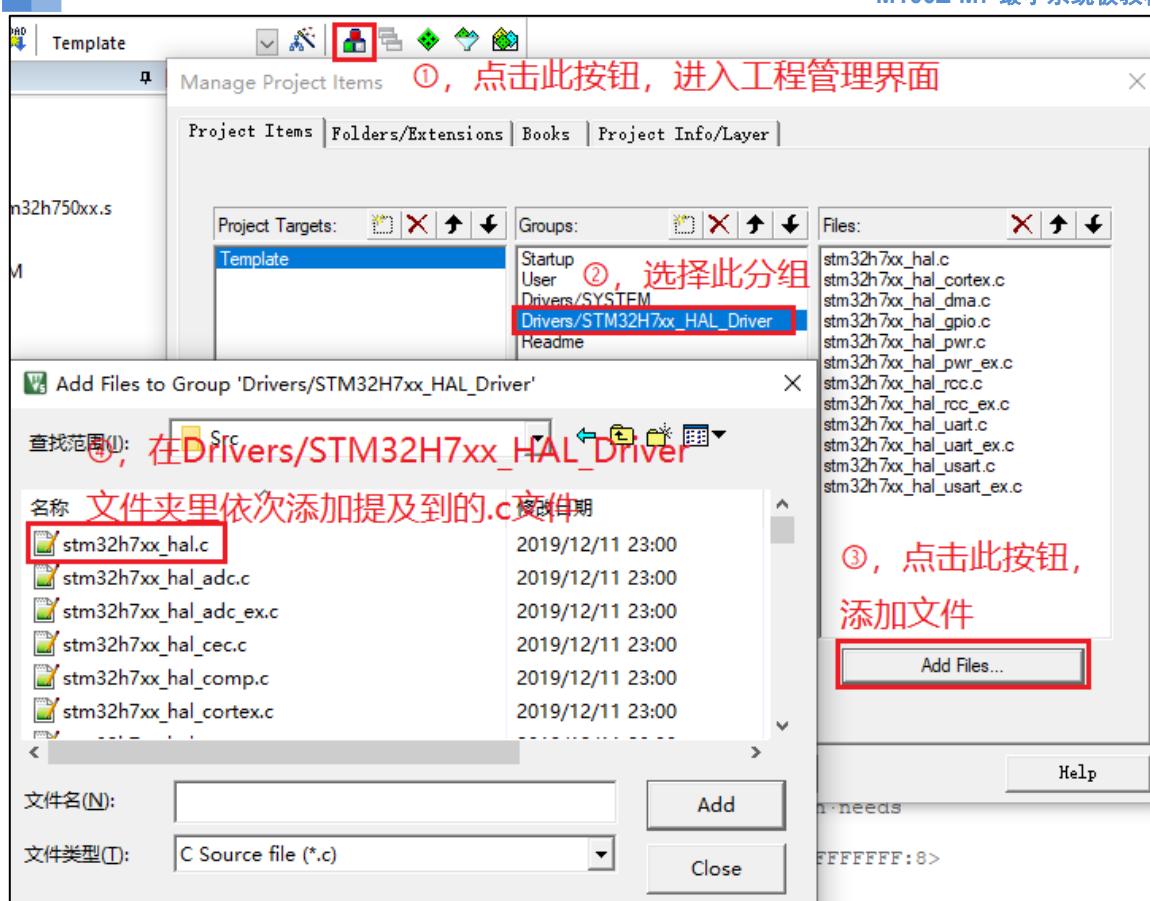


图 7.1.4.10 添加 STM32H7xx_HAL_Driver 源码

添加完成后, 如下图所示:

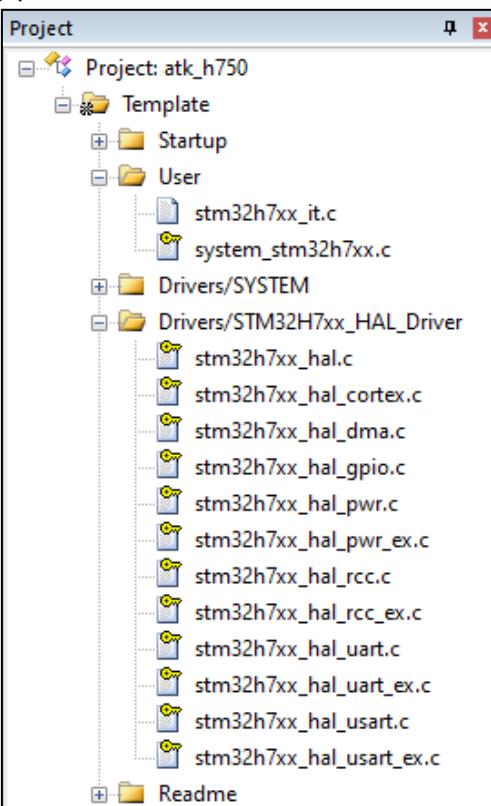


图 7.1.4.11 STM32H7xx_HAL_Driver 源码添加完成

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

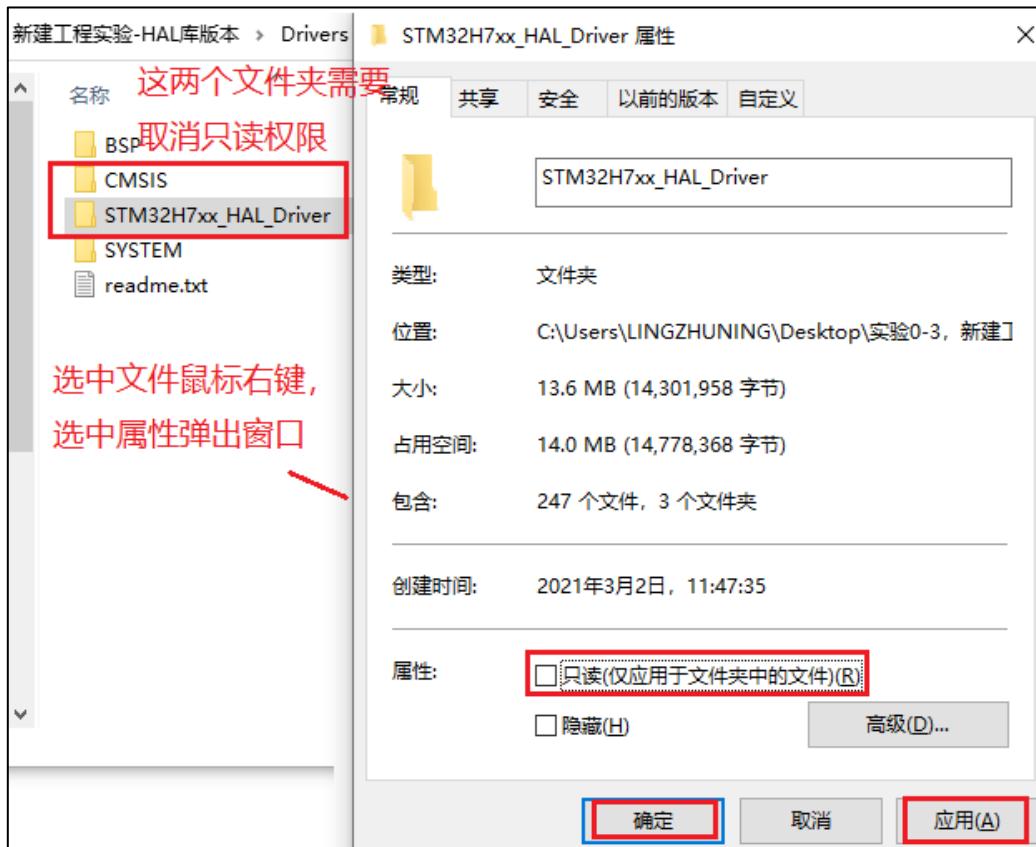


图 7.1.4.12 取消工程文件夹的只读权限

7.1.5 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，我们将进行以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，我们进行如下图所示设置：



图 7.1.5.1 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 7.1.5.1 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 7.1.5.1 AC5&AC6 简单对比

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

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 7.1.5.2 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 7.1.5.2 所示设置：



图 7.1.5.2 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，进行如下图所示设置：

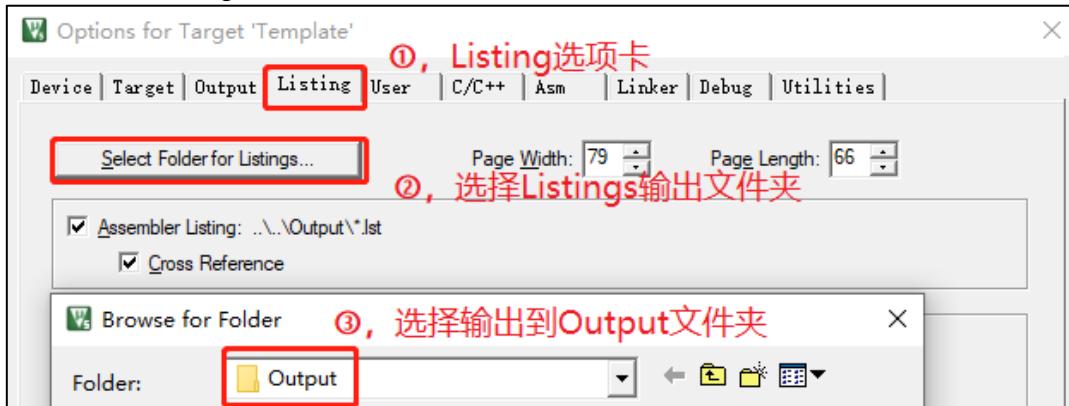


图 7.1.5.3 设置 Listing 选项卡

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，进行如下图所示设置：

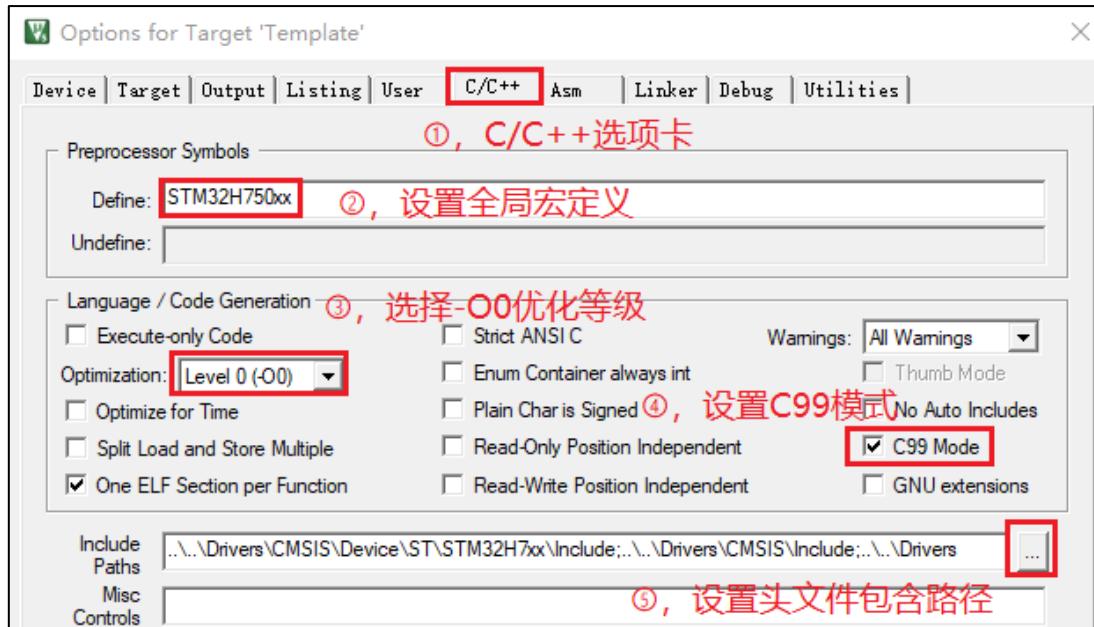


图 7.1.5.4 设置 C/C++ 选项卡

在②处设置了全局宏定义：STM32H750xx，用于定义所用 STM32 型号，在 stm32h7xx.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，进行如下图所示设置：

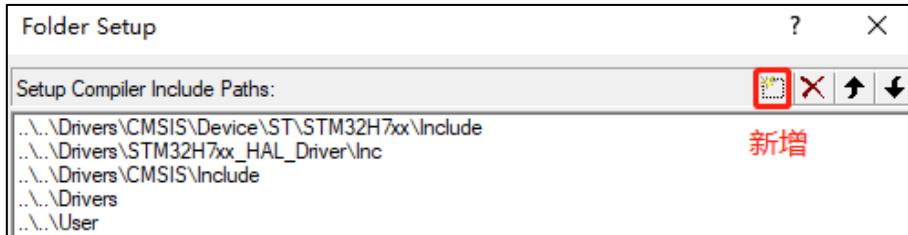


图 7.1.5.5 设置头文件包含路径

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\STM32H7xx\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，具体解释如下图所示：

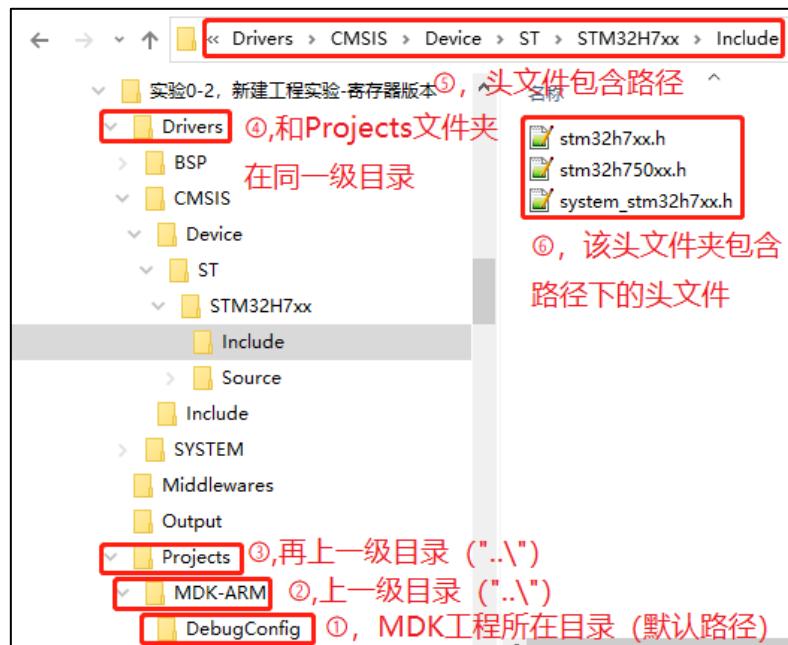


图 7.1.5.6 ..\Drivers\CMSIS\Device\ST\STM3H7xx\Include 的解释

上图表示根据头文件包含路径: ..\Drivers\CMSIS\Device\ST\STM32H7xx\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, 进行如下图所示设置:

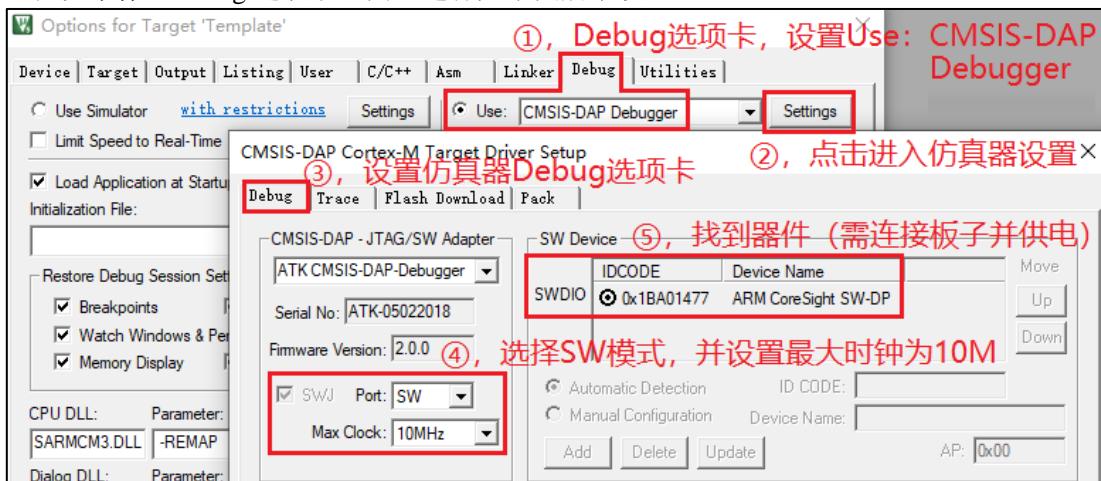


图 7.1.5.7 Debug 选项卡设置

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，进行如下图所示设置：

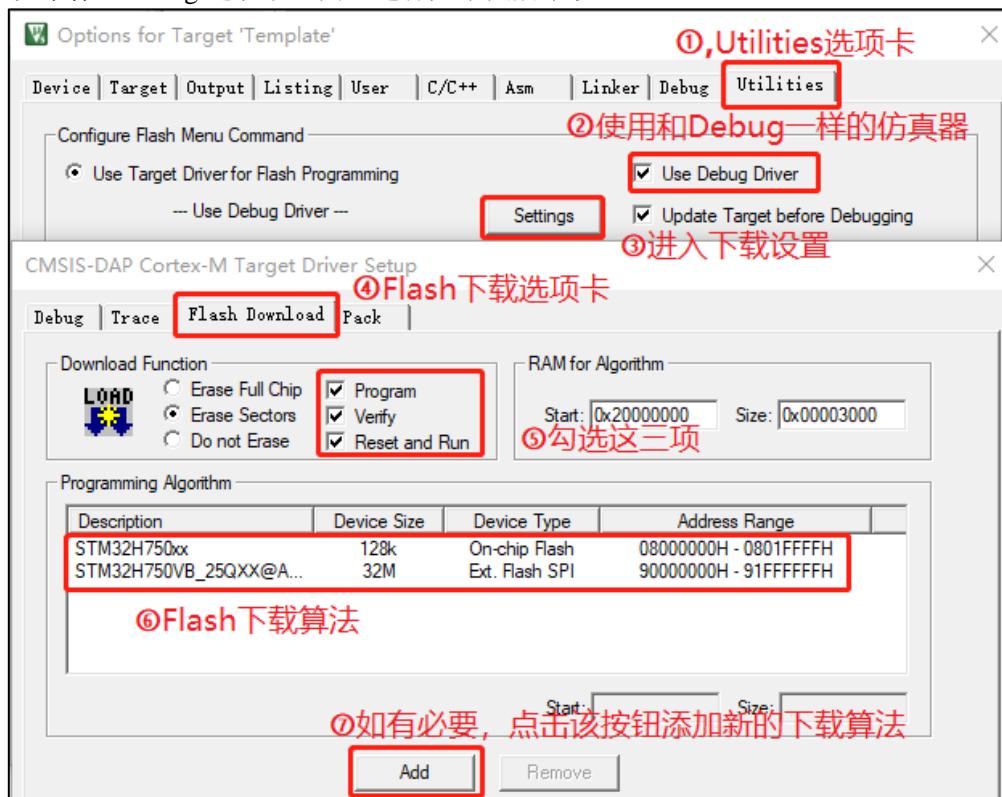


图 7.1.5.8 Utilities 选项卡设置

图中⑥处下载算法 STM32H750，是 MDK 默认添加的，针对 STM32H750 系列产品。除此之外，我们还要添加 STM32H750VB_W25Q64@ALIENTEK.FLM 算法，点击⑦处按钮添加即可。添加好算法后，设置算法使用的 RAM 地址和大小，这里设置的起始地址为：0X2000 0000 (DTCM)，大小为：0X3000。必须按这个大小设置，否则下载会出错（无法加载算法）。

7. 添加分散加载文件

由于 STM32H750VBT6 芯片内部的 FLASH 的空间比较少（只有 128KB）。对于大的工程，这个 FLASH 空间是不够用的，为了解决这个问题，同时方便后续工程的新建，我们统一使用分散加载的方式来决定 FLASH 内存的分配，而不用 MDK 默认的设置。关于分散加载是什么？我们后面 8.3 小节会讲解，请大家先跟着我们把新建工程完成。

分散加载的文件已经为大家准备好了，可以在 **实验 0-3，新建工程实验-HAL 库版本 \User\SCRIPT**，或者在 **(A 盘) /程序源码/STM32 启动文件/分散加载_HAL 库版本/SCRIPT** 中拷贝 **qspi_code.scf** 文件到我们的工程 **User\SCRIPT** 路径下，如图 8.1.4.9 所示。

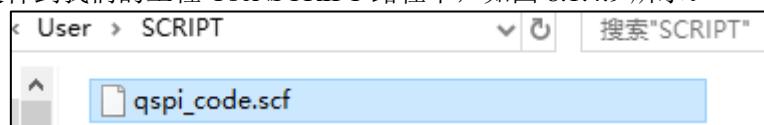


图 7.1.5.9 拷贝分散加载文件到工程

注意：这里的分散加载文件寄存器跟 HAL 库是不一样的，我们建立 HAL 库工程，所以必

需用 HAL 库版本的分散加载文件。

接下来我们需要对 MDK 进行配置，相当于把分散加载文件关联到工程里。方法：点击魔术棒 ，Linker 选项卡 → 取消勾选：Use Memory Layout from Target Dialog → Scatter File 路径 → 选择 SCRIPT 文件夹 → 选择 qspi_code.scf 文件，然后，在 disable Warnings 一栏，添加：6314,6329，屏蔽 6314 和 6329 这两个警告。如不屏蔽，当分散加载里面有某些段（section）没用到，则会报警告，所以我们需要屏蔽这两个警告。如下图所示。

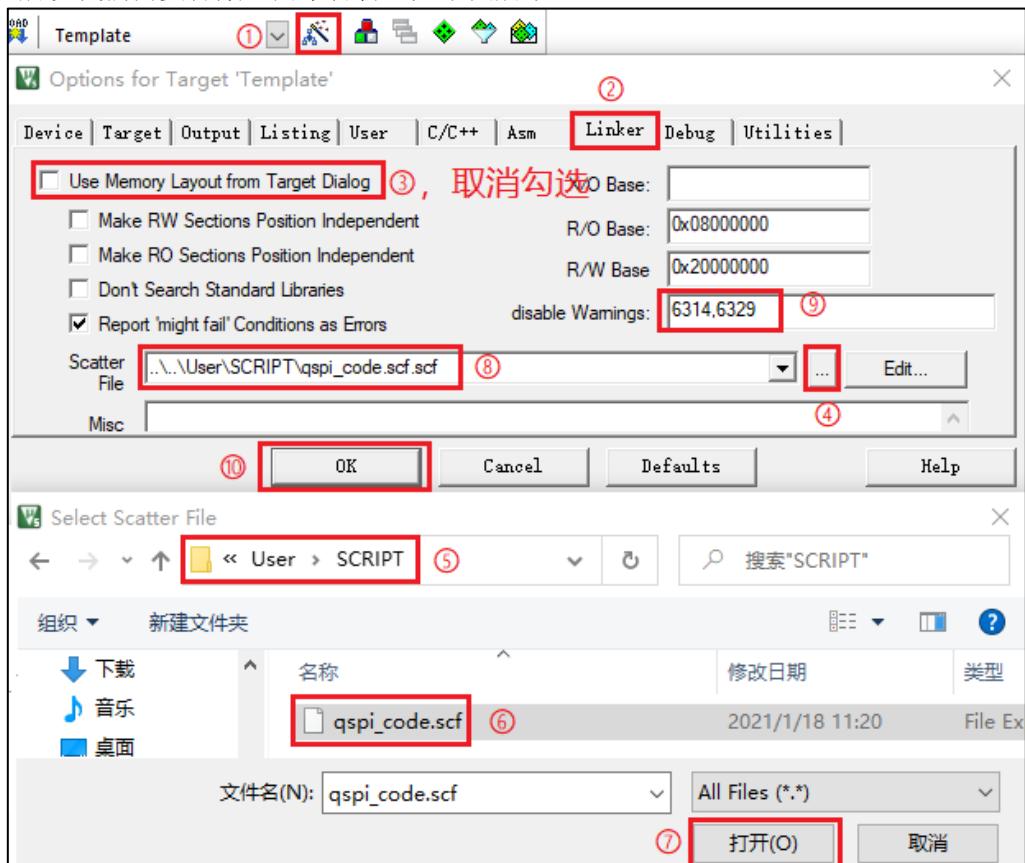


图 8.1.4.10 添加分散加载文件

至此，添加分散加载文件的相应操作就完成了。

7.1.6 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。得到如下图所示的界面：

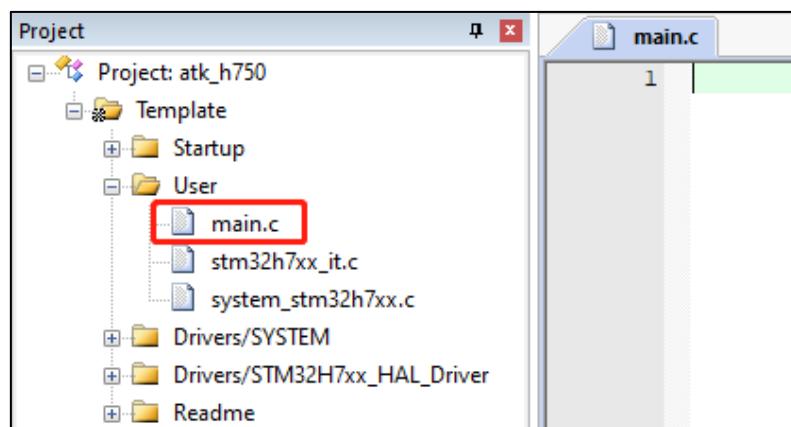


图 7.1.6.1 在 User 分组下加入 main.c 文件

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)
{
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */

    while (1)
    {
        printf("Hello, M100Z-M7-STM32H750\r\n");
        delay_ms(1000);
    }
}
```

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

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

编写完 main.c，我们点击： (Rebuild) 按钮，编译整个工程，编译结果如下图所示：

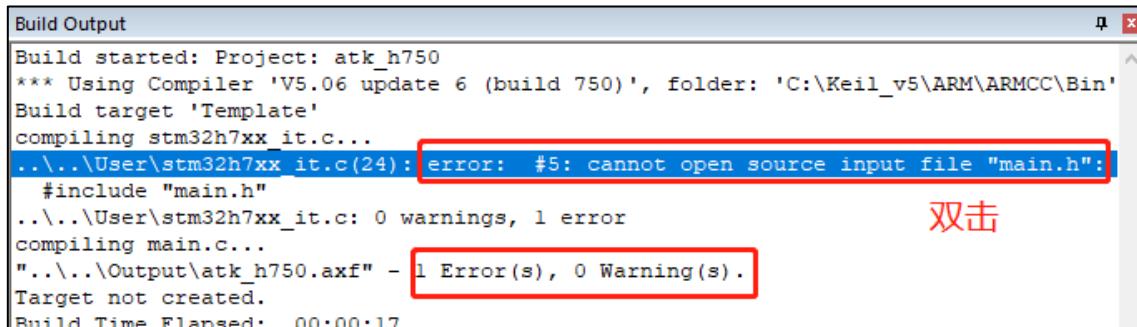


图 7.1.6.2 编译结果

编译结果可以看到 1 个错误，0 个警告。这个错误说找不到 main.h，因为我们也不需要用到 main.h，双击这个错误会弹出下面的 stm32h7xx_it.c 文件对应包含 main.h 的语句。我们只需要把它删除，然后重新编译，如图 8.1.5.3 所示。

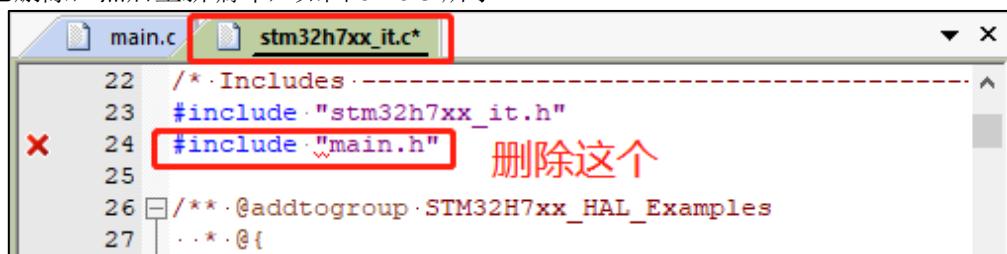


图 7.1.6.3 删除包含 main.h 的语句

编译后发现又有一个警告，警告 HAL_IncTick 函数没有声明，如下图所示。

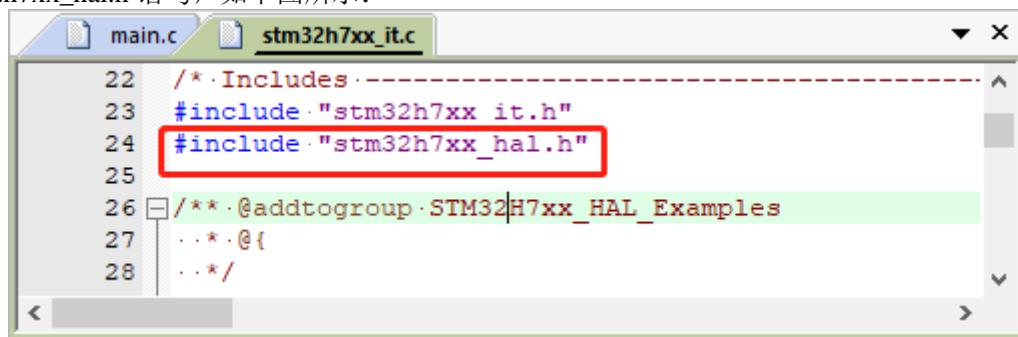
```

Build Output
compiling stm32h7xx_it.c...
...\\User\\stm32h7xx_it.c(140): warning: #223-D: function "HAL_IncTick" declared implicitly
    HAL_IncTick();
...\\User\\stm32h7xx_it.c: 1 warning, 0 errors
linking...
Program Size: Code=12718 RO-data=714 RW-data=32 ZI-data=1976
FromELF: creating hex file...
\"..\\Output\\atk_h750.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed: 00:00:00

```

图 7.1.6.4 编译报警告

因为这个函数是在 `stm32h7xx_hal.c` 定义了，并且在 `stm32h7xx_hal.h` 声明了。我们把 `stm32h7xx_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 `stm32h7xx_hal.h` 的语句，我们需要用到它，所以在里面把它包含进来。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 `stm32h7xx_it.c` 文件刚才删除包含 `main.h` 的语句的位置，编写包含 `stm32h7xx_hal.h` 语句，如下图所示：

图 7.1.6.5 包含 `stm32h7xx_hal.h` 头文件到工程

再进行编译就会发现 0 错误 0 警告，结果如下图所示：

```

Build Output
compiling stm32h7xx_hal_usart.c...
linking...
Program Size: Code=12774 RO-data=714 RW-data=32 ZI-data=1976
FromELF: creating hex file...
\"..\\Output\\atk_h750.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:07

```

图 7.1.6.6 编译结果

编译结果提示：代码总大小（Program Size）为：FLASH 占用 13520 字节（Code+RO+RW），SRAM 占用 2008 字节（RW+ZI）；并成功创建了 Hex 文件（可执行文件，放在 Output 目录下）。

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

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

至此，新建 HAL 库版本 MDK 工程完成。

7.2 下载验证

点击 MDK 软件上方工具栏上的 按钮进行程序烧录，程序烧录成功后，便可通过串口调试助手（推荐使用 ATK-XCOM，该软件可在 A 盘→6，软件资料→1，软件→串口调试助手软件（ATK-XOMC）下找到）看到 USART1 不断输出“Hello, M100Z-M7 STM32H750”字符串，如下图所示：



图 7.2.1 串口输出结果

第八章 STM32 时钟系统

STM32H7 时钟系统的知识在《STM32H7xx 参考手册_V7（英文版）.pdf》第八章复位和时钟控制章节有较详细的讲解。这里我们对 STM32H7 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 STM32H7 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：

8.1 认识时钟树

8.2 如何修改主频

8.1 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。回顾第五章的知识点，我们知道 STM32 内部也是由多种多样的电路模块组合在一起实现的。当一个电路越复杂，在达到正确的输出结果前，它可能因为延时会有一些短暂的中间状态，而这些中间状态有时会导致输出结果会有一个短暂的错误，这叫做电路中的“毛刺现象”，如果电路需要运行得足够快，那么这些错误状态会被其它电路作为输入采样，最终形成一系列的系统错误。为了解决这个问题，在单片机系统中，设计时以时序电路控制替代纯粹的组合电路，在每一级输出结果前对各个信号进行采样，从而使得电路中某些信号即使出现延时也可以保证各个信号的同步，可以避免电路中发生的“毛刺现象”，达到精确控制输出的效果。

由于时序电路的重要性，因此在 MCU 设计时就设计了专门用于控制时序的电路，在芯片设计中称为时钟树设计。由此设计出来的时钟，可以精确控制我们的单片机系统，这也是我们这节要展开分析的时钟分析。为什么是时钟树而不是时钟呢？一个 MCU 越复杂，时钟系统也会相应地变得复杂，如 STM32H7 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 STM32H7 系列的芯片，正常工作的主频可以达到 480MHz，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 内部非常复杂，外设也非常多，为了更低的功耗，STM32 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 芯片要使用哪些外设，从而再开启该外设的时钟，否则该外设不工作。

下面来看一下 STM32H7 时钟系统图。STM32H7 时钟系统图看起来有点多且复杂，但是分开几部分各个理解其实没有那么难了。

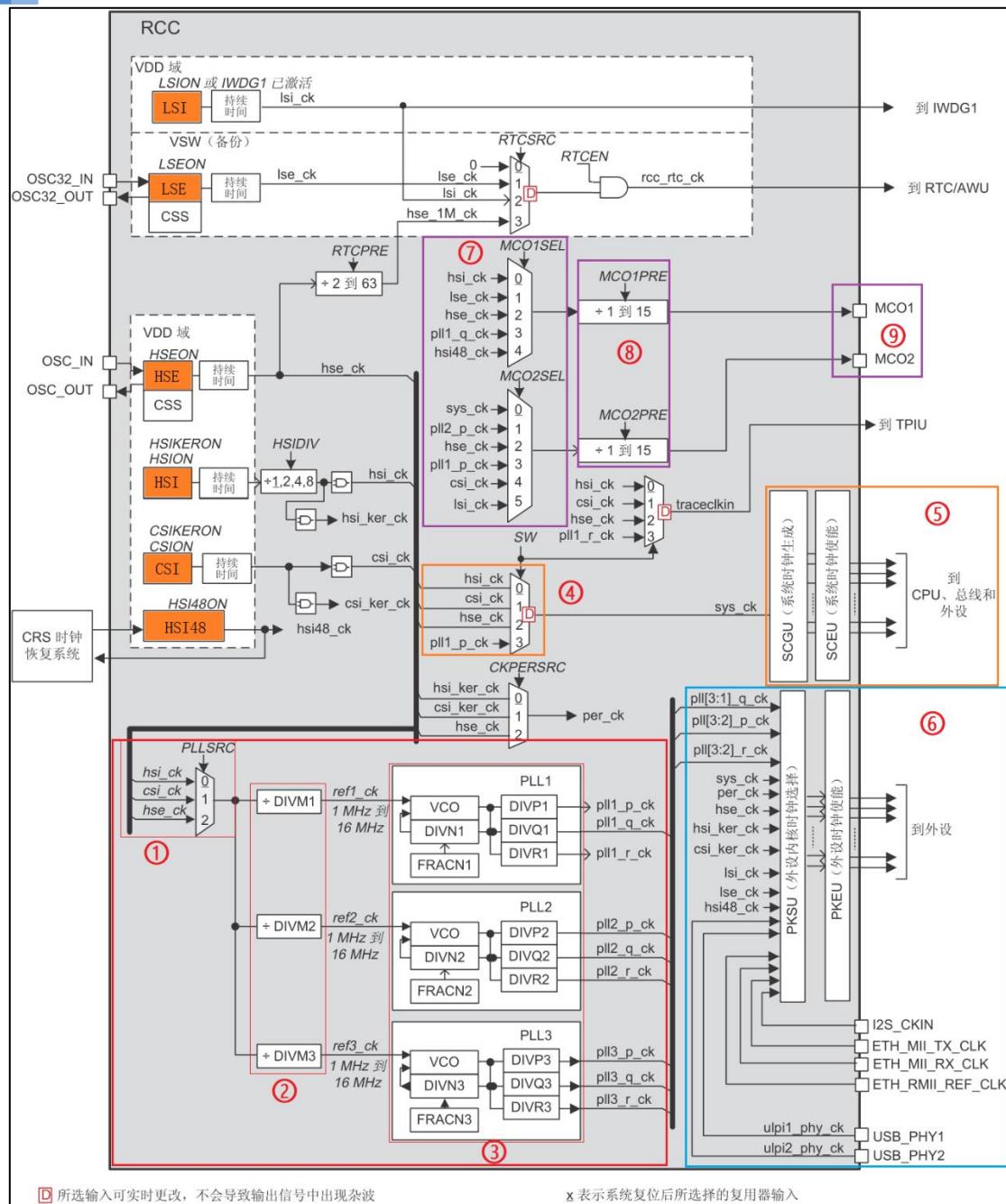


图 8.1.1 STM32H7 时钟系统图

8.1.1 时钟源

对于 STM32H7 有六个时钟源可供使用，可分为外部时钟源和内部时钟源。具体如图 8.1.1 所示，在 STM32H7 的时钟系统图里，我们把这六个时钟源标记成橘黄色。

(1) 2 个外部时钟源：

- 高速外部振荡器 (HSE)

支持 4 MHz 到 48 MHz 频率范围内的晶振。

- 低速外部振荡器 (LSE)

一般是 32.768 kHz 晶振，可用于看门狗和 RTC 实时时钟。

2 个外部时钟源都是芯片外部晶振产生的时钟频率，故而都有精度高的优点。

(2) 4 个内部时钟源：

- 高速内部振荡器 (HSI)

频率 64MHz 的高速 RC 振荡器，可用于系统时钟。

- 48 MHz RC 振荡器 (HSI48)

频率 48MHz，用于给特定的外设提供时钟，比如配合 CRS 可以作为 USB 的时钟源。

- 低功耗内部振荡器 (CSI)

频率约是 4MHz，主要用于低功耗。

- 低速内部振荡器 (LSI)

频率约 32 kHz 的低速 RC 振荡器，可用于 RTC 实时时钟、独立看门狗和自动唤醒。

8.1.2 锁相环 PLL

STM32H7 时钟系统图中看到红色框的部分，可以知道 STM32H7 有三个锁相环，分别是：PLL1、PLL2、PLL3。而且它们的时钟源只有一个，一起共用，但是进入到锁相环的时钟源分别经过三个不同的预分频器，这样我们就可以让锁相环有不同的输入时钟频率。

下面看到 STM32H7 时钟系统图标号①②③这三个组成部分：

- ①锁相环时钟源选择器

锁相环时钟源有三个：his_ck、csi_ck、hse_ck。我们一般选择 hse_ck，即来自外部高速晶振，正点原子 MiniPRO STM32H750 外部高速晶振为 8MHz。

- ②PLL1\PLL2\PLL3 时钟源预分频器

DIVM1、DIVM2 和 DIVM3 分别是 PLL1、PLL2 和 PLL3 输入时钟的预分频系数，取值范围都是：1~63，请根据实际需要设置。

- ③PLL1\PLL2\PLL3 锁相环

PLL1、PLL2 和 PLL3 锁相环都是一个输入，三个输出。his_ck、csi_ck、hse_ck 三个时钟源中的一个经过 DIVM 预分频器处理作为锁相环的输入时钟源。PLL1 三个输出分别是：

pll1_p_ck、pll1_q_ck 和 pll1_r_ck，PLL2 和 PLL3 同理。

下面以 PLL1 为例介绍一下锁相环内部框图，PLL2 和 PLL3 同理。

DIVN1 是 PLL1 中 VCO 的倍频系数，其取值范围是：4~512。

DIVP1 是 pll1_p_ck 的预分频系数，取值范围是：2、4、6...128（必须是偶数）。

DIVQ1 是 pll1_q_ck 的预分频系数，取值范围是：1~128。

DIVR1 是 pll1_r_ck 的预分频系数，取值范围是：1~128。

FRACN1 是 PLL1 中 VCO 的倍频系数的小数部分，它和 DIVN1 一起组成 PLL1 的倍频系数，但是我们一般情况下都是不需要用到小数倍频的，所以小数部分我们不讨论。

这里以 pll1_p_ck 为例，简单介绍下 PLL 输出频率的计算公式(时钟 PLL 输入频率为 hse_ck)：

$$pll1_p_ck = \frac{(hse_ck)}{DIVP1} * DIVN1$$

假设外部晶振为 8MHz，需要得到 480MHz 的 pll1_p_ck 频率，则可以设置：DIVM1=2，DIVN1=240，DIVP1=2 即可。其他分频输出 (pll1_q_ck/pll1_r_ck) 计算方法与上式相似。我们把它们对应到 STM32CubeMX 工具的时钟系统图上理解就很直观了，如图 8.1.2.1 所示。

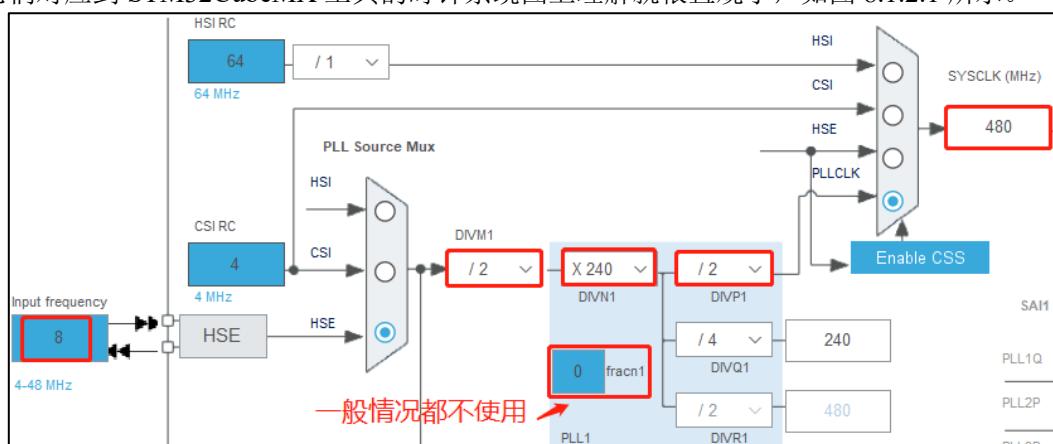


图 8.1.2.1 STM32CubeMX 对应的设置

关于时钟系统，大家如果看数据手册不太理解，结合 STM32CubeMX 的时钟系统配置界面理解会有很好的效果。

8.1.3 系统时钟 SYSCLK

下面看到 STM32H7 时钟系统图标号④⑤两个组成部分：

- **④sys_ck 时钟源选择器**

sys_ck (即系统时钟) 的时钟源有四个：hs1_ck、csi_ck、hse_ck 和 pll1_p_ck，系统默认是选择 hsi_ck (64MHz) 作为时钟源的。当我们把 PLL1 配置好并且选择 pll1_p_ck 作为系统时钟的时钟源，则系统时钟可以得到 480MHZ 的时钟频率，从而得到最高性能。

- **⑤系统时钟生成和使能单元**

SCGU (System Clock Generation Unit, 系统时钟生成单元)，用于将 sys_ck 分成各种时钟频率，供给 CPU、SysTick，以及各条总线使用，从而有了许多的时钟名称。这里大家对以下的总线及其相对应的时钟名称有印象即可，比如：在 AHB1~4 总线上的时钟就是 rcc_hclk1~4，在 APB1~4 总线上的时钟就是 rcc_pclk1~4。这 8 条总线和对应的 8 个时钟在后面会经常提及。

SCEU (System Clock Eable Unit, 系统时钟使能单元)，用于使能各个外设、总线等时钟，是一个时钟开关。

接下来，我们介绍系统时钟生成单元图（非常重要），如图 8.1.3.1 所示；

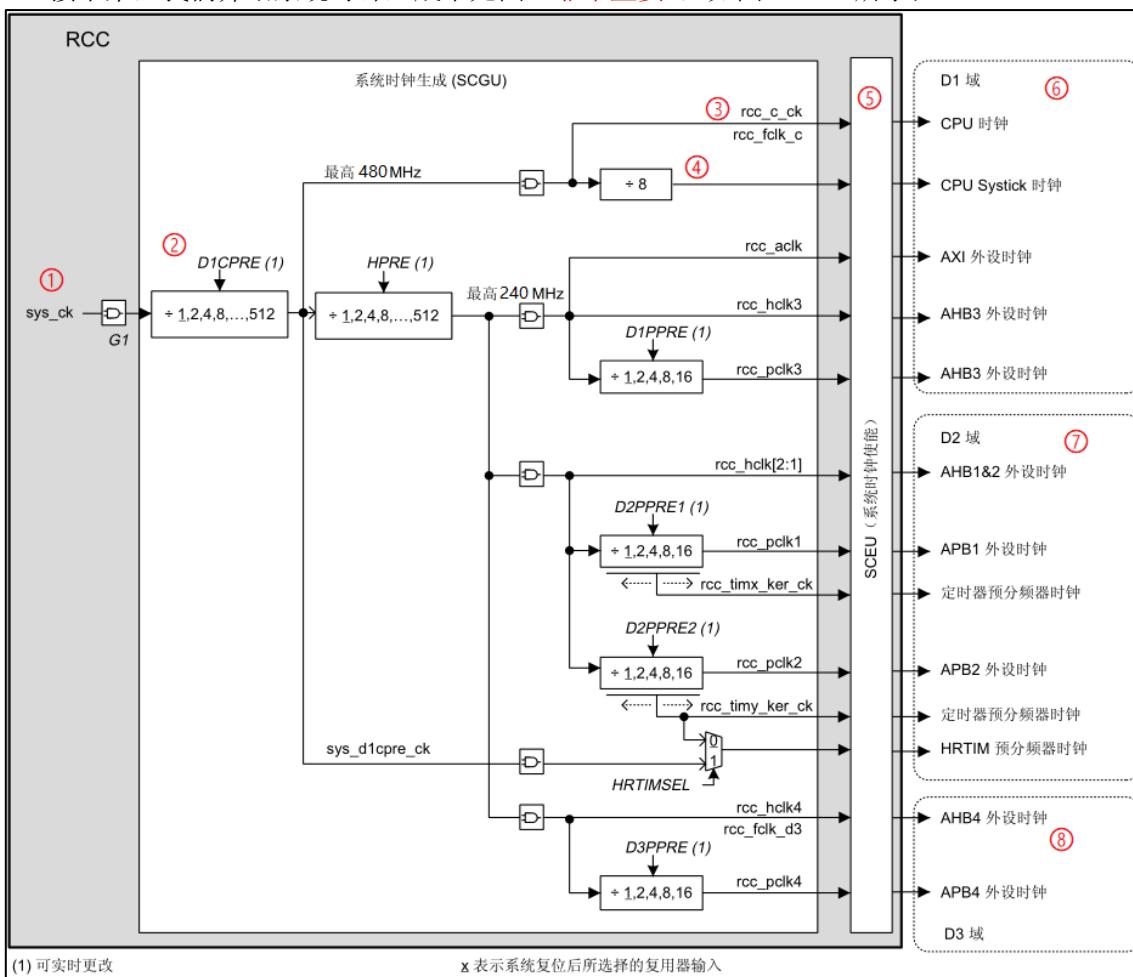


图 8.1.3.1 STM32H750 系统时钟生成图

上图主要列出了 STM32H750 系统时钟的生成原理，包括 CPU 时钟、SysTick 时钟、AXI 时钟、AHB1~4 总线和 APB1~4 总线时钟等，这些时钟对整个系统运行来说非常重要，所以该图必须要看懂。

图中，D1、D2 和 D3 域是 ST 为了支持动态能效管理，所设计的 3 个独立的电源域，每个域

都能独立开启/关闭。系统时钟由 SCGU 产生，然后经过 SCEU 做开关，最终输出到各个时钟域（D1、D2 和 D3），从而能够控制和访问各类外设，保证系统的正常运行。

我们挑选出 8 个重要的地方进行介绍（图 8.1.3.1 中标出的①~⑧）：

① SCGU 输入时钟 (sys_ck)，该时钟我们一般选择来自 pll1_p_ck，频率为 480MHz。

② sys_d1cpref_ck 时钟的分频系数，取值范围为 1~512，通过 RCC_D1CFGR 寄存器的 D1CPRE[3:0]位设置，我们一般设置为 1 分频，以得到最高的 sys_d1cpref_ck 频率，480Mhz。

③ CPU 时钟 (rcc_c_ck、rcc_fclk_c)，CPU 时钟是直接来自 sys_d1cpref_ck，没有分频器，频率为 480Mhz。

④ SysTick 时钟预分频器（可以选择 1 分频或者 8 分频），在例程中，我们选择 1 分频，因此 SysTick 的时钟源直接来自 sys_d1cpref_ck，频率为 480Mhz。

⑤ 系统时钟使能单元 (SCEU)，它能够对 D1、D2 和 D3 域内的所有外设时钟进行开/关控制，所以在使用外设的时候，必须设置 SCEU，使能其时钟，否则外设无法使用。这里所谓的 SCEU 设置，实际上就是通过 AHB1ENR 之类的寄存器设置的。

⑥ D1 域，高性能处理域，包含 CPU、SysTick、AXI、AHB3 和 APB3 等部分。CPU 可以从 TCM 和 Cache 中提取紧急的或优先级较高的用户程序，在 480M 的主频下执行，确保实现最快速响应。

⑦ D2 域，外设接口域，包含大部分外设，包括：AHB1、AHB2、APB1 和 APB2 等时钟部分。**注意：定时器的时钟都是在 D2 域进行控制，而且当 D2PPRE1 或 D2PPRE2 的分频系数不为 1 的时候，定时器的时钟频率为 rcc_pclk1 或 rcc_pclk2 的 2 倍。**

⑧ D3 域，低功耗处理域，包括 AHB4 和 APB4 等时钟部分。此部分中的 ADC 可以在整个系统深度休眠时仍然进行数据处理。在电池驱动的情况下，D3 可以保证在低功耗条件下仍然进行必要的数据处理工作。

我们可以把 STM32H750 系统时钟生成图对应到 STM32CubeMX 工具上理解，如图 8.1.3.2 所示。

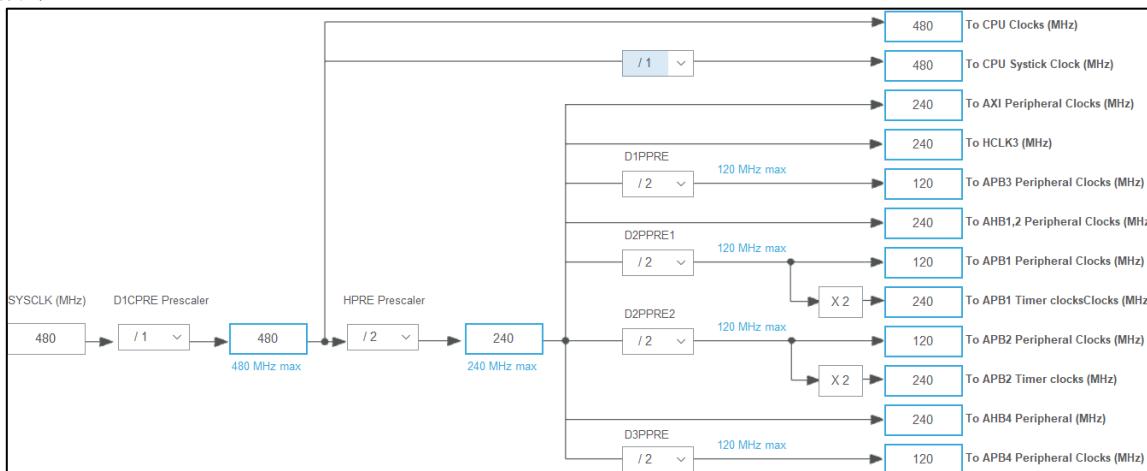


图 8.1.3.2 对应 STM32CubeMX 配置图

8.1.4 外设内核时钟

下面看到 STM32H7 时钟系统图标号⑥部分：

- **⑥外设内核时钟选择和使能单元**

PKSU (Peripheral Kernel clock Selection Unit，外设内核时钟选择单元)，用于选择部分外设的内核时钟源，STM32H7 内部，很多外设都可以自由选择内核时钟来源，从而提高使用的灵活性。

PKEU (Peripheral Kernel clock Enable Unit，外设内核时钟使能单元)，用于使能部分外设的内核时钟，从而控制外设内核时钟的开/关。

接下来我们看看外设时钟使能框图，如图 8.1.4.1 所示：

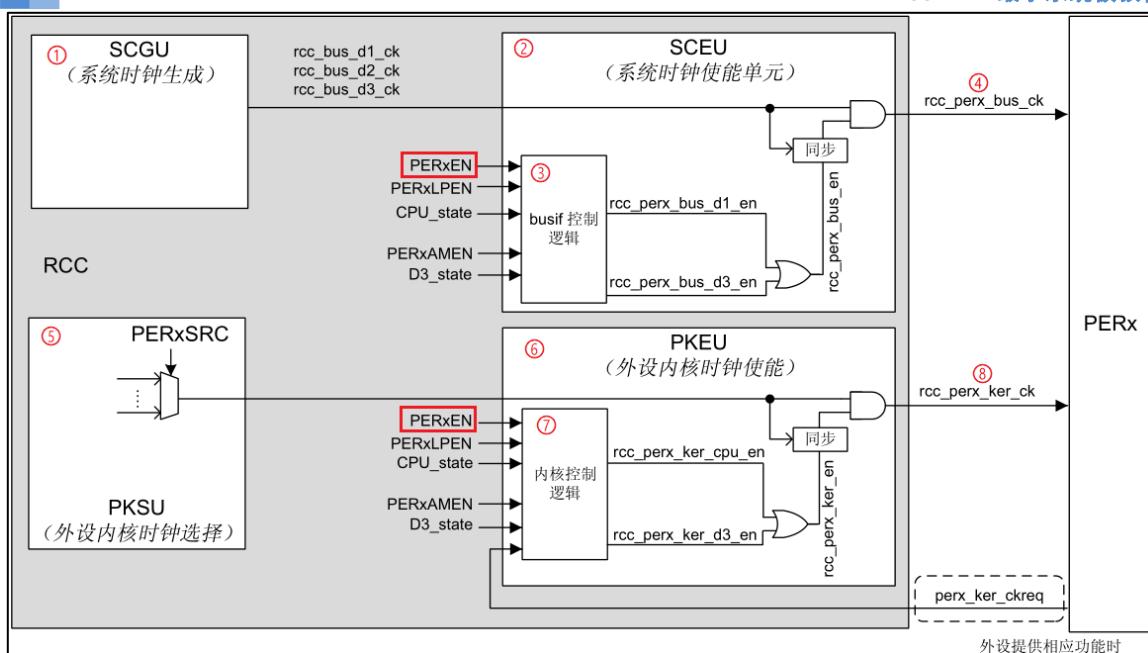


图 8.1.4.1 STM32H750 外设时钟使能框图

图中，PERx (perx) 代表外设 x (x 是 STM32H7 内部的外设，比如 USART、TIM、SPI 等)，通过此图，我们可以知道 STM32H7 内部外设时钟的由来。

我们挑选出 8 个重要的地方进行介绍（图 8.1.4.1 中标出的①~⑧）：

① 系统时钟生成单元 (SCGU)，此部分在图 8.1.3.1 我们进行了详细介绍，这里通过 SCGU 产生了三大总线时钟：rcc_bus_d1_ck、rcc_bus_d2_ck、rcc_bus_d3_ck，这些时钟实际上是指 rcc_hclk1~4、rcc_pclk1~4 等时钟。

② 系统时钟使能单元 (SCEU)，此部分将 SCGU 生成的时钟进行使能/禁止操作，最终控制是否输出时钟 (rcc_perx_bus_ck) 给外设。

③ 总线接口时钟控制逻辑 (busif Control Logic)，用于控制 SCEU 是否输出时钟给外设，它有很多控制信号，其中我们常用的是 PERxEN (图中红框框出部分)，通过这个位的设置就可以控制具体外设的时钟开/关。

④ 外设总线时钟 (rcc_perx_bus_ck)，该时钟主要用于访问外设寄存器，对外设进行设置。任何外设都必须使能该时钟才可以正常使用。

⑤ 外设内核时钟选择单元 (PKSU，即：Peripheral Kernel clock Selection Unit)，用于选择某个外设的内核时钟来源选择。

⑥ 外设内核时钟使能单元 (PKEU，即：Peripheral Kernel clock Enable Unit)，此部分将⑤处选择的外设内核时钟进行使能/禁止操作，最终控制是否输出内核时钟(rcc_perx_ker_ck)给外设。

⑦ 内核时钟控制逻辑 (Kernel Control Logic)，用于控制 PKEU 是否输出内核时钟给外设，它有很多控制信号，其中我们常用的是 PERxEN (图中红圈圈出部分)，通过这个位的设置就可以控制具体外设内核时钟的开/关。注意：③和⑦，都可以通过 PERxEN 控制使能。

⑧ 外设内核时钟 (rcc_perx_ker_ck)，该时钟用于驱动外设产生时序，如波特率、时钟脉冲等。大部分外设都需要用到 rcc_perx_ker_ck，比如串口、SPI、IIC、FMC、SAI、LTDC 和 CAN 等，但是，并不是所有的外设都需要用内核时钟来产生时序，比如：OPAMP、DMA 等。

因此，大部分外设的使用，需要同时用到外设总线时钟 (rcc_perx_bus_ck) 和外设内核时钟 (rcc_perx_ker_ck)，一般情况下，这两个时钟都是由 PERxEN 控制使能。至于哪些外设有内核时钟，请参考：《STM32H7xx 参考手册_V7 (英文版).pdf》352 页，Table 59。

SCEU 和 PKEU，一个用于控制外设的访问时钟（访问寄存器），一个用于控制外设的内核时钟（生成控制时序，如波特率等）。但是，并不是所有的外设都需要用到 PKEU，因为有些外设并不需要生成时序，没有所谓的外设内核时钟，比如 DMA、OPAMP 等，这些外设只需要在 SCEU 进行使能即可。

8.1.5 MCO 时钟输出

MCO 时钟输出，其作用是为外部器件提供时钟，STM32H750 有两个 MCO。下面看到 STM32H7 时钟系统图标号⑦⑧⑨三个部分：

- ⑦MCO1\MCO2 时钟源选择器

MCO1（外部器件的输出时钟 1）时钟源有五个：his_ck、lse_ck、hse_ck、pll1_q_ck 和 hsi48_ck。

MCO2（外部器件的输出时钟 2）时钟源有六个：sys_ck、pll2_p_ck、hse_ck、pll1_p_ck、csi_ck 和 lsi_ck。

- ⑧MCO1\MCO2 时钟分频器

MCO1PRE 是 MCO1 的预分频器，取值范围：1 到 15。

MCO2PRE 是 MCO2 的预分频器，取值范围：1 到 15。

- ⑨MCO1\MCO2 时钟输出引脚

MCO1、MCO2 两个时钟输出引脚给外部器件提供时钟源（分别由 PA8 和 PC9 复用功能实现），每个引脚可以选择一个时钟源，通过 RCC 时钟配置寄存器 (RCC_CFG) 进行配置。

关于 STM32H7xx 时钟的详细介绍，请参考《STM32H7xx 参考手册_V7（英文版）.pdf》第 8.5 节，有不明白的地方，可以对照手册仔细研究。**具体哪个外设是连接在哪个时钟总线上，以及对应的时钟总线最高主频是多少，请参考 STM32H750VBT6.pdf 数据手册（路径：A 盘→7，硬件资料→2，芯片资料→STM32H750VBT6.pdf）。**

通过以上内容的介绍，我们知道 STM32H750 的时钟设计的比较复杂，各个时钟基本都是可控的，任何外设都有对应的时钟控制开关，这样的设计，对降低功耗是非常有用的，不用的外设不开启时钟，就可以大大降低其功耗。

8.2 配置系统主频

STM32H750 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 64M 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 STM32H750 芯片在 480MHz 的频率下工作，480MHz 是官方推荐使用的最高的稳定时钟频率。正点原子的 MiniPRO STM32H750 开发板的外部高速晶振的频率是 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 480MHz 的系统工作频率。

8.2.1 STM32H7 时钟系统配置

下面我们将分几步给大家讲解 STM32H7 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 stm32h7xx_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），代码如下：

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

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
                IMPORT SystemInit
                IMPORT __main
```

```

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

```

下面我们来看看 system_stm32h7xx.c 文件下定义的 SystemInit 程序，源码在 140 行到 245 行，简化函数如下。

```

void SystemInit (void)
{
    /* FPU 设置-----*/
    #if (_FPU_PRESENT == 1) && (_FPU_USED == 1)
        SCB->CPACR |= ((3UL << (10*2)) | (3UL << (11*2))); /* 设置 CP10 和 CP11 为全访问 */
    #endif
    /* 复位 RCC 时钟配置为默认配置 */
    /* 打开 HSION 位 */
    RCC->CR |= RCC_CR_HSION;
    RCC->CFGGR = 0x00000000; /* 复位 CFGGR 寄存器 */
    /* 复位 HSEON, CSSON , CSION,RC48ON, CSIKERON PLL1ON, PLL2ON and PLL3ON 位 */
    RCC->CR &= 0xEAF6ED7FU;
    RCC->D1CFGR = 0x00000000; /* 复位 D1CFGR 寄存器 */
    RCC->D2CFGR = 0x00000000; /* 复位 D2CFGR 寄存器 */
    RCC->D3CFGR = 0x00000000; /* 复位 D3CFGR 寄存器 */
    RCC->PLLCSEL = 0x00000000; /* 复位 PLLCSEL 寄存器 */
    RCC->PLLCFGGR = 0x00000000; /* 复位 PLLCFGGR 寄存器 */
    RCC->PLL1DIVR = 0x00000000; /* 复位 PLL1DIVR 寄存器 */
    RCC->PLL1FRACR = 0x00000000; /* 复位 PLL1FRACR 寄存器 */
    RCC->PLL2DIVR = 0x00000000; /* 复位 PLL2DIVR 寄存器 */
    RCC->PLL2FRACR = 0x00000000; /* 复位 PLL2FRACR 寄存器 */
    RCC->PLL3DIVR = 0x00000000; /* 复位 PLL3DIVR 寄存器 */
    RCC->PLL3FRACR = 0x00000000; /* 复位 PLL3FRACR 寄存器 */
    RCC->CR &= 0xFFFFBFFFFU; /* 清除 CR 寄存器的 HSEBYP 位 */
    RCC->CIER = 0x00000000; /* 关闭所有的中断 */

    /* 双核 CM7 或单核线 */
    if((DBGMCU->IDCODE & 0xFFFF0000U) < 0x20000000U)
    {
        /* if stm32h7 revY */
        /* Change the switch matrix read issuing capability
           to 1 for the AXI SRAM target (Target 7) */
        *((__IO uint32_t*)0x51008108) = 0x00000001U;
    }

    /* Configure the Vector Table location add offset address for cortex-M7 ---*/
    #ifdef VECT_TAB_SRAM
        SCB->VTOR = D1_AXISRAM_BASE | VECT_TAB_OFFSET;
    #else
        SCB->VTOR = FLASH_BANK1_BASE | VECT_TAB_OFFSET;
    #endif
}

```

从上面代码可以看出，SystemInit 主要做了如下三个方面工作：

- 1) 设置 FPU
- 2) 复位 RCC 时钟配置为默认复位值（默认开启 HSI）
- 3) 配置中断向量表地址

HAL 库的 SystemInit 函数并没有像标准库的 SystemInit 函数一样进行时钟的初始化配置。HAL 库的 SystemInit 函数除了打开 HSI 之外，没有任何时钟相关配置，所以使用 HAL 库，我们必须编写自己的时钟配置函数。

第 3 步：在 main 函数里调用用户编写的时钟设置函数

我们打开 HAL 库例程实验 1 跑马灯实验，看看我们在工程目录 Drivers\SYSTEM 分组下面

定义的 sys.c 文件中的时钟设置函数 sys_stm32_clock_init 的内容:

```
/***
 * @brief      时钟设置函数
 * @param[in]   plln: PLL1 倍频系数(PLL 倍频), 取值范围: 4~512.
 * @param[in]   pllm: PLL1 预分频系数(进 PLL 之前的分频), 取值范围: 2~63.
 * @param[in]   pllP: PLL1 的 p 分频系数(PLL 之后的分频), 分频后作为系统时钟, 取值范围:
 *                  2~128. (且必须是 2 的倍数)
 * @param[in]   pllq: PLL1 的 q 分频系数(PLL 之后的分频), 取值范围: 1~128.
 * @note
 *
 *          Fvco: VCO 频率
 *          Fsyst: 系统时钟频率, 也是 PLL1 的 p 分频输出时钟频率
 *          Fq:    PLL1 的 q 分频输出时钟频率
 *          Fs:    PLL 输入时钟频率, 可以是 HSI, CSI, HSE 等.
 *          Fvco = Fs * (plln / pllm);
 *          Fsyst = Fvco / pllP = Fs * (plln / (pllM * pllP));
 *          Fq = Fvco / pllq = Fs * (plln / (pllM * pllq));
 *
 *          外部晶振为 25M 的时候, 推荐值:plln = 192, pllm = 5, pllP = 2, pllq = 4.
 *          外部晶振为 8M 的时候, 推荐值:plln = 240, pllm = 2, pllP = 2, pllq = 4.
 *          得到:Fvco = 8 * (240 / 2) = 960Mhz
 *          Fsyst = plln_p_ck = 960 / 2 = 480Mhz
 *          Fq = plln_q_ck = 960 / 4 = 240Mhz
 *
 *          H750 默认需要配置的频率如下:
 *          CPU 频率(rcc_c_ck) = sys_d1cpre_ck = 480Mhz
 *          rcc_aclk = rcc_hclk3 = 240Mhz
 *          AHB1/2/3/4(rcc_hclk1/2/3/4) = 240Mhz
 *          APB1/2/3/4(rcc_pclk1/2/3/4) = 120Mhz
 *          pll2_p_ck = (8 / 8) * 440 / 2 = 220Mhz
 *          pll2_r_ck = FMC 时钟频率 = ((8 / 8) * 440 / 2) = 220Mhz
 *
 * @retval     错误代码: 0, 成功; 1, 错误;
 */
uint8_t sys_stm32_clock_init(uint32_t plln, uint32_t pllm,
                             uint32_t pllP, uint32_t pllq)
{
    HAL_StatusTypeDef ret = HAL_OK;
    RCC_OscInitTypeDef rcc_osc_init;
    RCC_ClkInitTypeDef rcc_clk_init;
    RCC_PeriphCLKInitTypeDef rcc_periph_clk_init;
    /*SCUEN = 0, 锁定 LDOEN 和 BYPASS 位的设置 */
    MODIFY_REG(PWR->CR3, PWR_CR3_SCUEN, 0);
    /* VOS = 0, Scale0, 1.35V 内核电压, 使用最高主频 480MHz */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE0);
    while ((PWR->D3CR & (PWR_D3CR_VOSRDY)) != PWR_D3CR_VOSRDY); /* 等待电压稳定 */

    /* 使能 HSE, 并选择 HSE 作为 PLL 时钟源, 配置 PLL1, 开启 USB 时钟 */
    rcc_osc_init.OscillatorType = RCC_OSCILLATORTYPE_HSE
                                | RCC_OSCILLATORTYPE_HSI48;
    rcc_osc_init.HSEState = RCC_HSE_ON;
    rcc_osc_init.HSIState = RCC_HSI_OFF;
    rcc_osc_init.CSIState = RCC_CSI_OFF;
    rcc_osc_init.HSI48State = RCC_HSI48_ON;
    rcc_osc_init.PLL.PLLState = RCC_PLL_ON;
    rcc_osc_init.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    rcc_osc_init.PLL.PLLN = plln;
    rcc_osc_init.PLL.PLLM = pllm;
    rcc_osc_init.PLL.PLLP = pllP;
    rcc_osc_init.PLL.PLLQ = pllq;
    rcc_osc_init.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;
    rcc_osc_init.PLL.PLLRGE = RCC_PLL1VCIRANGE_2;
    rcc_osc_init.PLL.PLLFRACN = 0;
}
```

```
ret=HAL_RCC_OscConfig(&rcc_osc_init);
if(ret != HAL_OK)
{
    return 1;
}

/*
 * 选择 PLL 的输出作为系统时钟
 * 配置 RCC_CLOCKTYPE_SYSCLK 系统时钟, 480M
 * 配置 RCC_CLOCKTYPE_HCLK 时钟, 240Mhz, 对应 AHB1, AHB2, AHB3 和 AHB4 总线
 * 配置 RCC_CLOCKTYPE_PCLK1 时钟, 120Mhz, 对应 APB1 总线
 * 配置 RCC_CLOCKTYPE_PCLK2 时钟, 120Mhz, 对应 APB2 总线
 * 配置 RCC_CLOCKTYPE_D1PCLK1 时钟, 120Mhz, 对应 APB3 总线
 * 配置 RCC_CLOCKTYPE_D3PCLK1 时钟, 120Mhz, 对应 APB4 总线
*/
rcc_clk_init.ClockType = (RCC_CLOCKTYPE_SYSCLK \
                           | RCC_CLOCKTYPE_HCLK \
                           | RCC_CLOCKTYPE_PCLK1 \
                           | RCC_CLOCKTYPE_PCLK2 \
                           | RCC_CLOCKTYPE_D1PCLK1 \
                           | RCC_CLOCKTYPE_D3PCLK1);

rcc_clk_init.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
rcc_clk_init.SYSCLKDivider = RCC_SYSCLK_DIV1;
rcc_clk_init.AHBCLKDivider = RCC_HCLK_DIV2;
rcc_clk_init.APB1CLKDivider = RCC_APB1_DIV2;
rcc_clk_init.APB2CLKDivider = RCC_APB2_DIV2;
rcc_clk_init.APB3CLKDivider = RCC_APB3_DIV2;
rcc_clk_init.APB4CLKDivider = RCC_APB4_DIV2;
ret = HAL_RCC_ClockConfig(&rcc_clk_init, FLASH_LATENCY_4);
if(ret != HAL_OK)
{
    return 1;
}

/*
 * 配置 PLL2 的 R 分频输出, 为 220Mhz
 * 配置 FMC 时钟源是 PLL2R
 * 配置 QSPI 时钟源是 PLL2R
 * 配置串口 1 和 串口 6 的时钟源来自: PCLK2 = 120Mhz
 * 配置串口 2 / 3 / 4 / 5 / 7 / 8 的时钟源来自: PCLK1 = 120Mhz
 * USB 工作需要 48MHz 的时钟, 可以由 PLL1Q, PLL3Q 和 HSI48 提供, 这里配置时钟源是 HSI48
*/
rcc_periph_clk_init.PeriphClockSelection = RCC_PERIPHCLK_USART2
                                         | RCC_PERIPHCLK_USART1 | RCC_PERIPHCLK_USB
                                         | RCC_PERIPHCLK_QSPI | RCC_PERIPHCLK_FMC;
rcc_periph_clk_init.PLL2.PLL2M = 8;
rcc_periph_clk_init.PLL2.PLL2N = 440;
rcc_periph_clk_init.PLL2.PLL2P = 2;
rcc_periph_clk_init.PLL2.PLL2Q = 2;
rcc_periph_clk_init.PLL2.PLL2R = 2;
rcc_periph_clk_init.PLL2.PLL2RGE = RCC_PLL2VCIRANGE_0;
rcc_periph_clk_init.PLL2.PLL2VCOSEL = RCC_PLL2VCOWIDE;
rcc_periph_clk_init.PLL2.PLL2FRACN = 0;
rcc_periph_clk_init.FmcClockSelection = RCC_FMCLKSOURCE_PLL2;
rcc_periph_clk_init.QspiClockSelection = RCC_QSPICLKSOURCE_PLL2;
rcc_periph_clk_init.Usart234578ClockSelection =
                                         RCC_USART234578CLKSOURCE_D2PCLK1;
rcc_periph_clk_init.Usart16ClockSelection = RCC_USART16CLKSOURCE_D2PCLK2;
rcc_periph_clk_init.UsbClockSelection = RCC_USBCLKSOURCE_HSI48;
ret = HAL_RCCEx_PeriphCLKConfig(&rcc_periph_clk_init);
if(ret != HAL_OK)
{
```

```

    return 1;
}

sys_qspi_enable_memmapmode(0); /* 使能 QSPI 内存映射模式, FLASH 容量为普通类型 */

HAL_PWREx_EnableUSBVoltageDetector(); /* 使能 USB 电压电平检测器 */
__HAL_RCC_CSI_ENABLE(); /* 使能 CSI 时钟 */
__HAL_RCC_SYSCFG_CLK_ENABLE(); /* 使能 SYSCFG 时钟 */
HAL_EnableCompensationCell(); /* 使能 IO 补偿单元 */
return 0;
}

```

函数 `sys_stm32_clock_init` 就是用户的时钟系统配置函数，除了配置 PLL 相关参数确定 SYSCLK 值之外，还配置了 AHB、APB1、APB2、APB3 和 APB4 总线时钟的预分频系数，也就是确定了 `rcc_hclk1~4` 和 `rcc_pclk1~4` 等时钟频率。我们首先来看看使用 HAL 库配置 STM32H7 时钟系统的一般步骤：

- 1) 设置调压器输出电压级别：调用函数 `_HAL_PWR_VOLTAGESCALING_CONFIG()`。
- 2) 配置时钟源相关参数：调用函数 `HAL_RCC_OscConfig()`。
- 3) 选择系统时钟 SYSCLK 的时钟来源以及配置 AHB、APB1、APB2、APB3 和 APB4 总线时钟的预分频系数，调用 `HAL_RCC_ClockConfig()` 函数。
- 4) 配置 PLL2/PLL3 以及扩展外设的时钟，调用 `HAL_RCCEx_PeriphCLKConfig()` 函数。
- 5) 使能其他一些时钟和 IO 补偿单元。

下面我们详细讲解这个 5 个步骤。

步骤 1：调压器输出电压级别 VOS，它是由 PWR 控制寄存器 D3CR 的位 15:14 来确定的：

位 15:14 VOS[1:0]
00:保留（默认选择电压调节 3）
01:电压调节 3（默认）
10:电压调节 2
11:电压调节 1

电压调节级别数值越小工作频率越高，所以如果我们要配置 H750 的主频为 480MHz，那么我们需要配置调压器输出电压调节级别 VOS 为级别 1，详细的表格参考在步骤 3 会讲解。所以函数 `sys_stm32_clock_init` 中步骤 1 源码如下：

```
/* VOS = 1, Scale1, 1.2V 内核电压, FLASH 访问可以得到最高性能 */
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
```

步骤 2：配置时钟源和锁相环 PLL1 的相关参数，使能并选择 HSE 作为 PLL 时钟源，我们调用的函数为 `HAL_RCC_OscConfig()`，该函数在 HAL 库头文件 `stm32h7xx_hal_rcc.h` 中声明，在文件 `stm32h7xx_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 `RCC_OscInitTypeDef` 结构体类型指针。接下来我们看看 `RCC_OscInitTypeDef` 结构体的定义：

```

typedef struct
{
    uint32_t OscillatorType; /* 需要选择配置的振荡器类型 */
    uint32_t HSEState; /* HSE 状态 */
    uint32_t LSEState; /* LSE 状态 */
    uint32_t HSIState; /* HIS 状态 */
    uint32_t HSICalibrationValue; /* HIS 校准值 */
    uint32_t LSIState; /* LSI 状态 */
    uint32_t HSI48State; /* HSI48 的状态 */
    uint32_t CSIState; /* CSI 状态 */
    uint32_t CSICalibrationValue; /* CSI 校准值 */
    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、CSI 和 HSI48，配置方法类似，这

里我们还开启了 HSI48 的时钟。

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;          /* PLL1 状态 */
    uint32_t PLLSource;         /* PLL1 时钟源 */
    uint32_t PLLM;              /* PLL1 分频系数 M */
    uint32_t PLLN;              /* PLL1 倍频系数 N */
    uint32_t PLLP;              /* PLL1 分频系数 P */
    uint32_t PLLQ;              /* PLL1 分频系数 Q */
    uint32_t PLLR;              /* PLL1 分频系数 R */
    uint32_t PLLRGE;             /* PLL1 时钟输入范围 */
    uint32_t PLLVCOSEL;          /* PLL1 时钟输出范围 */
    uint32_t PLLFRACN;          /* PLL1 VCO 乘数因子的小数部分 */
} 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。这个结构体定义的相关内容请结合图 8.1.1 时钟树中红色框部分内容一起理解。

接下来看看系统时钟初始化函数 sys_stm32_clock_init 中的配置内容：

```
/* 使能 HSE，并选择 HSE 作为 PLL 时钟源，配置 PLL1，开启 USB 时钟 */
rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_HSE |  
RCC_OSCILLATORTYPE_HSI48;  
rcc_osc_init_handle.HSEState = RCC_HSE_ON;           /* 打开 HSE */  
rcc_osc_init_handle.HSISState = RCC_HSI_OFF;          /* 关闭 HSI */  
rcc_osc_init_handle.CSISState = RCC_CSI_OFF;          /* 关闭 CSI */  
rcc_osc_init_handle.HSI48State = RCC_HSI48_ON;         /* 打开 HSI48 */  
rcc_osc_init_handle.PLL.PLLState = RCC_PLL_ON;        /* 打开 PLL */  
rcc_osc_init_handle.PLL.PLLSource = RCC_PLLSOURCE_HSE; /* 设置 PLL 时钟源为 HSE */  
rcc_osc_init_handle.PLL.PLLN = plln;  
rcc_osc_init_handle.PLL.PLLM = pllm;  
rcc_osc_init_handle.PLL.PLLP = pllp;  
rcc_osc_init_handle.PLL.PLLQ = pllq;  
rcc_osc_init_handle.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;  
rcc_osc_init_handle.PLL.PLLRGE = RCC_PLL1VCIRANGE_2;  
rcc_osc_init_handle.PLL.PLLFRACN = 0;  
ret=HAL_RCC_OscConfig(&rcc_osc_init_handle);
```

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 4 个形式参数直接设置作为 PLL 的参数 N、M、P 和 Q 的值，这样就达到了设置 PLL 时钟源相关参数的目的。另外我们还开启了 HSI48 时钟，为 USB 相关实验做准备。

设置好时钟源和 PLL1 的相关参数后，就完成了步骤 2 的内容。

步骤 3：选择系统时钟 SYSCLK 的时钟来源以及配置 AHB、APB1、APB2、APB3 和 APB4 总线时钟的预分频系数，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 RCC_ClkInitTypeDef 结构体类型指针变量，用于设置 SYSCLK 时钟源以及 SYSCLK、AHB、APB1、APB2、APB3 和 APB4 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 SYSCLKDivider;      /* SYSCLK 分频系数 */  
    uint32_t AHBCLKDivider;      /* AHB 分频系数 */  
    uint32_t APB3CLKDivider;      /* APB3 分频系数 */
```

```

    uint32_t APB1CLKDivider;           /* APB1 分频系数 */
    uint32_t APB2CLKDivider;           /* APB2 分频系数 */
    uint32_t APB4CLKDivider;           /* APB4 分频系数 */
}RCC_ClkInitTypeDef;

```

我们在 sys_stm32_clock_init 函数中的实际应用配置内容如下：

```

/*
 * 选择 PLL 的输出作为系统时钟
 * 配置 RCC_CLOCKTYPE_SYSCLK 系统时钟, 480M
 * 配置 RCC_CLOCKTYPE_HCLK 时钟, 240Mhz, 对应 AHB1, AHB2, AHB3 和 AHB4 总线
 * 配置 RCC_CLOCKTYPE_PCLK1 时钟, 120Mhz, 对应 APB1 总线
 * 配置 RCC_CLOCKTYPE_PCLK2 时钟, 120Mhz, 对应 APB2 总线
 * 配置 RCC_CLOCKTYPE_D1PCLK1 时钟, 120Mhz, 对应 APB3 总线
 * 配置 RCC_CLOCKTYPE_D3PCLK1 时钟, 120Mhz, 对应 APB4 总线
 */

rcc_clk_init_handle.ClockType = (RCC_CLOCKTYPE_SYSCLK \
| RCC_CLOCKTYPE_HCLK \
| RCC_CLOCKTYPE_PCLK1 \
| RCC_CLOCKTYPE_PCLK2 \
| RCC_CLOCKTYPE_D1PCLK1 \
| RCC_CLOCKTYPE_D3PCLK1);

rcc_clk_init_handle.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
rcc_clk_init_handle.SYSCLKDivider = RCC_SYSCLK_DIV1;
rcc_clk_init_handle.AHBCLKDivider = RCC_HCLK_DIV2;
rcc_clk_init_handle.APB1CLKDivider = RCC_APB1_DIV2;
rcc_clk_init_handle.APB2CLKDivider = RCC_APB2_DIV2;
rcc_clk_init_handle.APB3CLKDivider = RCC_APB3_DIV2;
rcc_clk_init_handle.APB4CLKDivider = RCC_APB4_DIV2;
ret = HAL_RCC_ClockConfig(&rcc_clk_init_handle, FLASH_LATENCY_4);

```

sys_stm32_clock_init 函数中的 RCC_ClkInitTypeDef 结构体配置内容：

第一个成员 ClockType 配置表示我们要配置到的时钟，把所以配置到的时钟的宏都赋值给该成员，这里我们包含了六个宏定义。

第二个成员 SYSCLKSource 配置选择系统时钟源为 PLL。

第三个成员 SYSCLKDivider 配置 SYSCLK 分频系数为 1。

第四个成员 AHBCLKDivider 配置 AHB 总线时钟预分频系数为 2。

第五个成员 APB1CLKDivider 配置 APB1 总线时钟预分频系数为 2。

第六个成员 APB2CLKDivider 配置 APB2 总线时钟预分频系数为 2。

第七个成员 APB3CLKDivider 配置 APB3 总线时钟预分频系数为 2。

第八个成员 APB4CLKDivider 配置 APB4 总线时钟预分频系数为 2。

根据我们在 main 函数中调用 sys_stm32_clock_init(240, 2, 2, 4) 时设置的形参数值，代入 PLL 输出频率的计算公式可得：

$$pll1_p_ck = \frac{\left(\frac{hse_ck}{\text{DIVM1}}\right) * \text{DIVN1}}{\text{DIVP1}} = \frac{\left(\frac{hse_ck}{M}\right) * N}{P} = \frac{\left(\frac{8MHz}{2}\right) * 240}{2} = 480MHz$$

因为我们选择系统时钟源为 PLL (pll1_p_ck)，所以系统时钟 SYSCLK=480MHz。再结合各个总线时钟的预分频系数，总结一下调用函数 sys_stm32_clock_init(240, 2, 2, 4) 之后，关键的时钟频率值如下：(HCLK 代表 AHB1~4 总线上的时钟)

sys_ck(系统时钟)	= 480MHz
sys_ck 分频后的时钟 (sys_d1cpre_ck)	= 480MHz
AHB 总线时钟 (HCLK= sys_d1cpre_ck /2)	= 240MHz
APB1 总线时钟 (rcc_pclk1=HCLK/2)	= 120MHz
APB2 总线时钟 (rcc_pclk2=HCLK/2)	= 120MHz
APB3 总线时钟 (rcc_pclk3=HCLK/2)	= 120MHz
APB4 总线时钟 (rcc_pclk4=HCLK/2)	= 120MHz

最后我们来看看函数 HAL_RCC_ClockConfig 第二个入口参数 FLatency 的含义，对于 STM32H7 系列，FLASH 延迟配置参数值是通过下表来确定的：

Number of wait states (LATENCY)	Programming delay (WRHIGHFREQ)	AXI Interface clock frequency vs V _{CORE} range			
		VOS3 range 0.95 V - 1.05 V	VOS2 range 1.05 V - 1.15 V	VOS1 range 1.15 V - 1.26 V	VOS0 range 1.26 V - 1.40 V
0 WS (1 FLASH clock cycle)	00	[0;45 MHz]	[0;55 MHz]	[0;70 MHz]	[0;70 MHz]
1 WS (2 FLASH clock cycles)	01]45 MHz;90 MHz]]55 MHz;110 MHz]]70 MHz;140 MHz]]70 MHz;140 MHz]
2 WS (3 FLASH clock cycles)	01]90 MHz;135 MHz]]110 MHz;165 MHz]]140 MHz;185 MHz]]140 MHz;185 MHz]
	10	-	-]185 MHz;210 MHz]]185 MHz;210 MHz]
3 WS (4 FLASH clock cycles)	10]135 MHz;180 MHz]]165 MHz;225 MHz]]210 MHz;225 MHz]]210 MHz;225 MHz]
4 WS (5 FLASH clock cycles)	10]180 MHz;225 MHz]	225 MHz	-]225 MHz;240 MHz]

表 8.2.1.1 FLASH 推荐的等待状态和编程延迟数

上表中的时钟频率为 ACLK(即 AXI Interface clock), ACLK=HCLK=240MHz (我们现在实际是 240MHz)，上表中，当调压器设置为 VOS1 级别的时候(我们设置的就是 VOS1 级别)，如果需要 ACLK 最高为 225MHz，那么 FLASH 延迟等待周期要为 3WS，也就是总共 4 个 FLASH 时钟周期才访问一次 FLASH。但是我们的 ACLK=HCLK=240MHz，这里最高只有 225MHz，这是因为之前的 H7 最高是 400MHz 的，现在出了 480MHz，但是官方文档没有更新完全，所以我们先将就参考这个表。下面我们看看我们在 sys_stm32_clock_init 中调用函数 HAL_RCC_ClockConfig 的时候，第二个形参设置值：

```
ret = HAL_RCC_ClockConfig(&rcc_clk_init_handle, FLASH_LATENCY_4);
```

从上可以看出，我们设置值为 FLASH_LATENCY_4，也就是每次访问 FLASH 需要延迟 4WS，总共要 5 个 FLASH 周期。为什么选 FLASH_LATENCY_4 呢？因为 ST 官方例程使用的就是 4 个 FLASH 四个延迟周期，其实大于 2 个 WS 都可以(最大不能超过 7 个 WS)，延时越久越稳定，但是肯定影响性能。综合考虑性能和稳定性，我们也就选择 4 个 FLASH 四个延迟周期。这个值也是我们实践出来的，大家直接选用这个值就行。

最后，再给大家用 STM32CubeMX 工具的时钟配置界面，标记一下：sys_ck、sys_d1cpref_ck、rcc_hclk1~4 以及 rcc_pclk1~4 对应的位置，方便大家更容易理解。如下图：

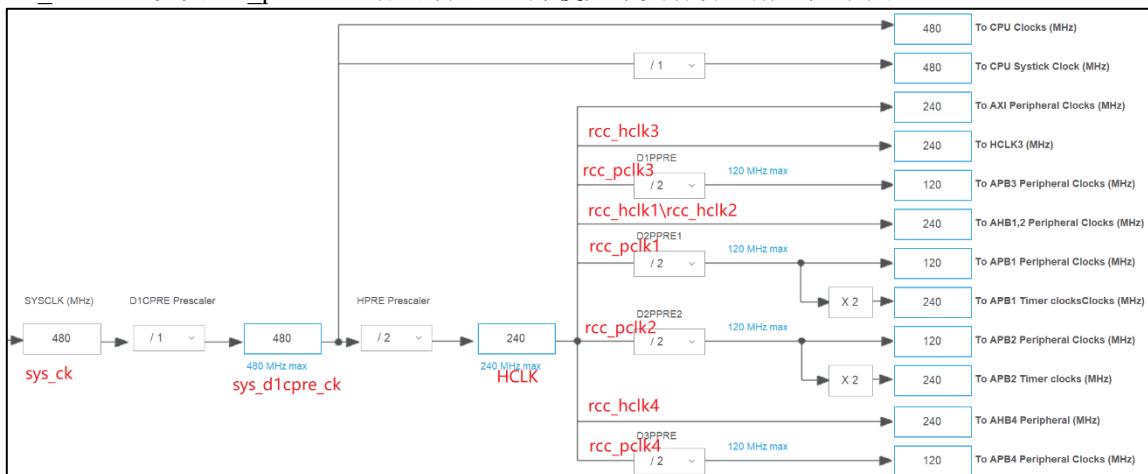


图 8.2.1.1 STM32CubeMX 时钟配置界面标记图

由上图可知道：

rcc_hclk1~4 分别对应的是 AHB1, AHB2, AHB3 和 AHB4 总线的时钟。

rcc_pclk1~4 分别对应的是 APB1, APB2, APB3 和 APB4 总线的时钟。

步骤4: 配置 PLL2, DIVM2 预分频系数为 8, DIVN2 倍频系数为 440, PLL2 的三个时钟输出对应的预分频系数 DIVP2、DIVQ2 和 DIVR2 都是 2, 即 2 分频, 所以得到它们的时钟频率都是 220MHz。因为我们要用到 PLL2R 输出作为 FMC 和 QSPI 的时钟源, 所以我们用 PLL 输出频率计算公式算出 PLL2R 的输出频率:

$$pll2_p_ck = \frac{\left(\frac{hse_ck}{DIVM2}\right) * DIVN2 \left(\frac{8MHz}{8}\right) * 440}{DIVR2} = 220MHz$$

由上面的公式得: PLL2R 输出频率为 220MHz, PLL2P 和 PLL2Q 计算方法同理。

这里我们除了配置 PLL2R 的输出频率外, 还配置了串口和 USB 的时钟源, 都是为后续实验准备。

配置 PLL2/PLL3 以及扩展外设的时钟, 我们要调用 HAL_RCCEEx_PeriphCLKConfig()函数, 其声明如下:

```
HAL_StatusTypeDef HAL_RCCEEx_PeriphCLKConfig(RCC_PeriphCLKInitTypeDef *PeriphClkInit);
```

该函数只有一个形参, 就是 RCC_PeriphCLKInitTypeDef 结构体类型指针。该结构体类型涉及的内容很多, 我们只要把用到的搞懂就好。其简化定义如下:

```
typedef struct
{
    uint32_t PeriphClockSelection; /* 要配置的扩展时钟 */
    RCC_PLL2InitTypeDef PLL2; /* PLL2 时钟配置结构体 */
    RCC_PLL3InitTypeDef PLL3; /* PLL3 时钟配置结构体 */
    uint32_t FmcClockSelection; /* FMC 时钟源 */
    uint32_t QspiClockSelection; /* QSPI 时钟源 */
    uint32_t Usart234578ClockSelection; /* USART2/3/4/5/7/8 时钟源 */
    uint32_t Usart16ClockSelection; /* USART1/6 时钟源 */
    uint32_t UsbClockSelection; /* USB 时钟源 */
}RCC_PeriphCLKInitTypeDef;
```

我们只把现在用到的定义列出来, 其他以后用到再讲。PeriphClockSelection 是需要配置的时钟, 结构体变量 PLL2 和 PLL3 就是对应的 PLL2 和 PLL3 的时钟配置, 其他的变量都是对应的外设时钟源配置。我们要用到 PLL2 的 DIVR2 分频输出, 所以看看 RCC_PLL2InitTypeDef 结构体的定义:

```
typedef struct
{
    uint32_t PLL2M; /* PLL2 分频系数 M */
    uint32_t PLL2N; /* PLL2 倍频系数 N */
    uint32_t PLL2P; /* PLL2 倍频系数 P */
    uint32_t PLL2Q; /* PLL2 倍频系数 Q */
    uint32_t PLL2R; /* PLL2 倍频系数 R */
    uint32_t PLL2RGE; /* PLL2 时钟输入范围 */
    uint32_t PLL2VCOSEL; /* PLL2 时钟输出范围 */
    uint32_t PLL2FRACN; /* PLL2 VCO 乘数因子的小数部分 */
}RCC_PLL2InitTypeDef;
```

该结构体跟我们前面讲过的 RCC_PLLInitTypeDef 结构体很像。RCC_PLL3InitTypeDef 的也是差不多的, 就不列出来了, 现在也没用到。

步骤4, 我们在 sys_stm32_clock_init 函数中的实际配置内容如下:

```
/*
 * 配置 PLL2 的 R 分频输出, 为 220Mhz
 * 配置 FMC 时钟源是 PLL2R
 * 配置 QSPI 时钟源是 PLL2R
 * 配置串口 1 和 串口 6 的时钟源来自: PCLK2 = 120Mhz
 * 配置串口 2 / 3 / 4 / 5 / 7 / 8 的时钟源来自: PCLK1 = 120Mhz
 * USB 工作需要 48MHz 的时钟, 可以由 PLL1Q, PLL3Q 和 HSI48 提供, 这里配置时钟源是 HSI48
 */
rcc_periph_clk_init_handle.PeriphClockSelection = RCC_PERIPHCLK_USART2
                                | RCC_PERIPHCLK_USART1 | RCC_PERIPHCLK_USB
                                | RCC_PERIPHCLK_QSPI | RCC_PERIPHCLK_FMC;
rcc_periph_clk_init_handle.PLL2.PLL2M = 8;
rcc_periph_clk_init_handle.PLL2.PLL2N = 440;
```

```
rcc_periph_clk_init_handle.PLL2.PLL2P = 2;
rcc_periph_clk_init_handle.PLL2.PLL2Q = 2;
rcc_periph_clk_init_handle.PLL2.PLL2R = 2;
rcc_periph_clk_init_handle.PLL2.PLL2RGE = RCC_PLL2VCIRANGE_0;
rcc_periph_clk_init_handle.PLL2.PLL2VCOSEL = RCC_PLL2VCOWIDE;
rcc_periph_clk_init_handle.PLL2.PLL2FRACN = 0;
rcc_periph_clk_init_handle.FmcClockSelection = RCC_FMCCLKSOURCE_PLL2;
rcc_periph_clk_init_handle.QspiClockSelection = RCC_QSPICLKSOURCE_PLL2;
rcc_periph_clk_init_handle.Usart234578ClockSelection =
    RCC_USART234578CLKSOURCE_D2PCLK1;
rcc_periph_clk_init_handle.Usart16ClockSelection =
    RCC_USART16CLKSOURCE_D2PCLK2;
rcc_periph_clk_init_handle.UsbClockSelection = RCC_USBCLKSOURCE_HSI48;
ret = HAL_RCCEx_PeriphCLKConfig(&rcc_periph_clk_init_handle);
```

经过以上的配置，得到 `pll2_p_ck`, `pll2_q_ck` 和 `pll2_r_ck` 这三个时钟的频率都为 220MHz, 配置外设 FMC 和 QSPI 的时钟源都是 `pll2_r_ck`, 串口 1、6 时钟源是 `rcc_pclk2`, 串口 2、3、4、5、7、8 时钟源是 `rcc_pclk1`, USB 时钟源是 HSI48。

步骤 5: 使能其他一些时钟和 IO 补偿单元，我们直接看代码：

```
sys_qspi_enable_memmapmode(0); /* 使能 QSPI 内存映射模式, FLASH 容量为普通类型 */

HAL_PWREX_EnableUSBVoltageDetector(); /* 使能 USB 电压电平检测器 */
__HAL_RCC_CSI_ENABLE(); /* 使能 CSI 时钟 */
__HAL_RCC_SYSCFG_CLK_ENABLE(); /* 使能 SYSCFG 时钟 */
HAL_EnableCompensationCell(); /* 使能 IO 补偿单元 */
```

`sys_qspi_enable_memmapmode` 函数我们在 12.2.2 小节再讲解。后面的函数就比较简单，都是一些使能函数。其中使能 IO 补偿单元功能，对于要求 IO 翻转速度较快的应用，一般都建议使能 IO 补偿单元功能。使能 CSI 时钟，为 IO 补偿单元提供时钟。

时钟系统配置相关知识就给大家讲解到这里。

8.2.2 STM32H7 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《STM32H7xx 参考手册_V7(英文版).pdf》8.7 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 STM32H7 的 HAL 库使能外设时钟的方法。

在 STM32H7 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 `stm32h7xx_hal_rcc.h` 定义的。大家打开 `stm32h7xx_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->AHB4ENR, RCC_AHB4ENR_GPIOAEN); \
    /* Delay after an RCC peripheral clock enabling */ \
    tmpreg = READ_BIT(RCC->AHB4ENR, RCC_AHB4ENR_GPIOAEN); \
    UNUSED(tmpreg); \
} while(0)
```

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->AHB4ENR, RCC_AHB4ENR_GPIOAEN);
```

这行代码的作用是，设置寄存器 `RCC->AHB4ENR` 的相关位为 1，至于是哪个位，是由宏定义标识符 `RCC_AHB4ENR_GPIOAEN` 的值决定的，而它的值为：

```
#define RCC_AHB4ENR_GPIOAEN_Pos (0U)
#define RCC_AHB4ENR_GPIOAEN_Msk (0x1UL << RCC_AHB4ENR_GPIOAEN_Pos)
#define RCC_AHB4ENR_GPIOAEN RCC_AHB4ENR_GPIOAEN_Msk
```

上面三行代码很容易计算出来 `RCC_AHB4ENR_GPIOAEN=0x00000001`, 因此上面代码的作用是设置寄存器 `RCC->AHB4ENR` 寄存器的最低位为 1。我们可以从 STM32H7 的参考手册中搜

索 AHB4ENR 寄存器定义，最低位的作用是用来使用 GPIOA 时钟。AHB4ENR 寄存器的位 0 描述如下：

位 0 GPIOAEN: GPIOA 外设时钟使能

由软件置 1 和清零

0: 禁止 IO 端口 A 时钟（复位后的默认值）

1: 使能 IO 端口 A 时钟

那么我们只需要在我们的用户程序中调用宏定义标识符就可以实现 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 stm32h7xx_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->AHB4ENR) &= ~ (RCC_AHB4ENR_GPIOAEN)
```

同样，宏定义标识符 __HAL_RCC_GPIOA_CLK_DISABLE() 的作用是设置 RCC->AHB4ENR 寄存器的最低位为 0，也就是禁止 GPIOA 时钟。具体使用方法我们这里就不做过多讲解，我们这里同样列出几个常用的禁止外设时钟的宏定义标识符使用方法：

```
__HAL_RCC_DMA1_CLK_DISABLE(); /* 禁止 DMA1 时钟 */
```

```
__HAL_RCC_USART2_CLK_DISABLE(); /* 禁止串口 2 时钟 */
```

```
__HAL_RCC_TIM1_CLK_DISABLE(); /* 禁止 TIM1 时钟 */
```

关于 STM32H7 的外设时钟使能和禁止方法我们就给大家讲解到这里。

第九章 SYSTEM 文件夹介绍

SYSTEM 文件夹里面的代码由正点原子提供，是 STM32H7xx 系列的底层核心驱动函数，可以用在 STM32H7xx 系列的各个型号上面，方便大家快速构建自己的工程。

SYSTEM 文件夹下包含了 delay、sys、uart 等三个文件夹。分别包含了 delay.c、sys.c、uart.c 及其头文件。通过这 3 个 c 文件，可以快速的给任何一款 STM32H7 构建最基本的框架，使用起来是很方便的。

本章，我们将向大家介绍这些代码，通过这章的学习，大家将了解到这些代码的由来，也希望大家可以灵活使用 SYSTEM 文件夹提供的函数，来快速构建工程，并实际应用到自己的项目中去。

本章将分为如下几个小节：

9.1 delay 文件夹代码介绍

9.2 sys 文件夹代码介绍

9.3 uart 文件夹代码介绍

9.1 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 rms);
```

前面 4 个函数，仅在支持操作系统（OS）的时候，需要用到，而后面 3 个函数，则不论是否支持 OS 都需要用到。

在介绍这些函数之前，我们先了解一下编程思想：CM7 内核和 CM3/CM4 内核一样，内部都包含了一个 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。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《STM32H7xx 参考手册_V7（英文版）.pdf》里面基本没有介绍，其详细介绍，请参阅《STM32H7 编程手册.pdf》第 212 页，4.4 节。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

这里我们将介绍的是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*480（假设系统时钟为 480Mhz，因为 systick 的频率等于系统时钟频率，那么 systick 每增加 1，就是 1/480us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*480，一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。

9.1.1 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 */
#ifndef OS_CRITICAL_METHOD             /* OS_CRITICAL_METHOD 定义了, 说明要支持 UCOSII */
#define delay_osrunning   OSRunning      /* OS 是否运行标记, 0, 不运行; 1, 在运行 */
#define delay_ostickspersec OS_TICKS_PER_SEC /* OS 时钟节拍, 即每秒调度次数 */
#define delay_osintnesting OSIntNesting  /* 中断嵌套级别, 即中断嵌套次数 */
#endif

/* 支持 UCOSIII */
#ifndef CPU_CFG_CRITICAL_METHOD /* CPU_CFG_CRITICAL_METHOD 定义了, 说明要支持 UCOSIII */
#define delay_osrunning   OSRunning      /* OS 是否运行标记, 0, 不运行; 1, 在运行 */
#define delay_ostickspersec OSCfg_TickRate_Hz /* OS 时钟节拍, 即每秒调度次数 */
#define delay_osintnesting OSIntNestingCtr /* 中断嵌套级别, 即中断嵌套次数 */
#endif

/**
 * @brief    us 级延时时, 关闭任务调度(防止打断 us 级延迟)
 * @param    无
 * @retval   无
 */
void delay_osschedlock(void)
{
#ifdef CPU_CFG_CRITICAL_METHOD /* 使用 UCOSIII */
    OS_ERR err;
    OSSchedLock(&err);           /* UCOSIII 的方式, 禁止调度, 防止打断 us 延时 */
#else
    OSSchedLock();                /* UCOSII 的方式, 禁止调度, 防止打断 us 延时 */
#endif
}

/**
 * @brief    us 级延时时, 恢复任务调度
 * @param    无
 * @retval   无
 */
void delay_osschedunlock(void)
{
#ifdef CPU_CFG_CRITICAL_METHOD /* 使用 UCOSIII */

```

```

OS_ERR err;
OSSchedUnlock(&err);           /* UCOSIII 的方式, 恢复调度 */
#else                           /* 否则 UCOSII */
OSSchedUnlock();               /* UCOSII 的方式, 恢复调度 */
#endif
}

/***
 * @brief      us 级延时时, 恢复任务调度
 * @param      ticks: 延时的节拍数
 * @retval     无
 */
void delay_ostimedly(uint32_t ticks)
{
#ifdef CPU_CFG_CRITICAL_METHOD
    OS_ERR err;
    OSTimeDly(ticks, OS_OPT_TIME_PERIODIC, &err); /* UCOSIII 延时采用周期模式 */
#else
    OSTimeDly(ticks); /* UCOSII 延时 */
#endif
}

/***
 * @brief      systick 中断服务函数, 使用 OS 时用到
 * @param      ticks: 延时的节拍数
 * @retval     无
 */
void SysTick_Handler(void)
{
    HAL_IncTick();
    if (delay_osrunning == 1) /* OS 开始跑了, 才执行正常的调度处理 */
    {
        OSIntEnter();          /* 进入中断 */
        OSTimeTick();          /* 调用 ucos 的时钟服务程序 */
        OSIntExit();           /* 触发任务切换软中断 */
    }
}
#endif

```

以上代码，仅支持 UCOSII 和 UCOSIII，不过，对于其他 OS 的支持，也只需要对以上代码进行简单修改即可实现。

支持 OS 需要用到的三个宏定义（以 UCOSII 为例）即：

```

#define delay_osrunning OSRunning           /* OS 是否运行标记, 0, 不运行; 1, 在运行 */
#define delay_ostickspersec OS_TICKS_PER_SEC /* OS 时钟节拍, 即每秒调度次数 */
#define delay_osintnesting OSIntNesting     /* 中断嵌套级别, 即中断嵌套次数 */

```

宏定义：delay_osrunning，用于标记 OS 是否正在运行，当 OS 已经开始运行时，该宏定义值为 1，当 OS 还未运行时，该宏定义值为 0。

宏定义：delay_ostickspersec，用于表示 OS 的时钟节拍，即 OS 每秒钟任务调度次数。

宏定义：delay_osintnesting，用于表示 OS 中断嵌套级别，即中断嵌套次数，每进入一个中断，该值加 1，每退出一个中断，该值减 1。

支持 OS 需要用到的 4 个函数，即：

函数：delay_osschedlock，用于 delay_us 延时，作用是禁止 OS 进行调度，以防打断 us 级延时，导致延时时间不准。

函数：delay_osschedunlock，同样用于 delay_us 延时，作用是在延时结束后恢复 OS 的调度，继续正常的 OS 任务调度。

函数：delay_ostimedly，则是调用 OS 自带的延时函数，实现延时。该函数的参数为时钟节拍数。

函数：SysTick_Handler，则是 systick 的中断服务函数，该函数为 OS 提供时钟节拍，同时可以引起任务调度。

以上就是 delay_ms 和 delay_us 支持操作系统时，需要实现的 3 个宏定义和 4 个函数。

9.1.2 delay_init 函数

该函数用来初始化 2 个重要参数：fac_us 以及 fac_ms；同时把 SysTick 的时钟源选择为外部时钟，如果需要支持操作系统（OS），只需要在 sys.h 里面，设置 SYS_SUPPORT_OS 宏的值为 1 即可，然后，该函数会根据 delay_ostickspersec 宏的设置，来配置 SysTick 的中断时间，并开启 SysTick 中断。具体代码如下：

```
/***
 * @brief    初始化延迟函数
 * @param    sysclk: 系统时钟频率, 即 CPU 频率(rcc_c_ck), 480Mhz
 * @retval   无
 */
void delay_init(uint16_t sysclk)
{
#if SYS_SUPPORT_OS /* 如果需要支持 OS */
    uint32_t reload;
#endif
/* SYSTICK 使用内核时钟源, 同 CPU 同频率 */
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
    g_fac_us = sysclk; /* 不论是否使用 OS, g_fac_us 都需要使用 */
#if SYS_SUPPORT_OS /* 如果需要支持 OS. */
    reload = sysclk; /* 每秒钟的计数次数 单位为 M */
/* 根据 delay_ostickspersec 设定溢出时间, reload 为 24 位
   寄存器, 最大值:16777216, 在 480M 下, 约合 0.035s 左右 */
    reload *= 1000000 / delay_ostickspersec;
    g_fac_ms = 1000 / delay_ostickspersec; /* 代表 OS 可以延时的最少单位 */
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk; /* 开启 SYSTICK 中断 */
    SysTick->LOAD = reload; /* 每 1/delay_ostickspersec 秒中断一次 */
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk; /* 开启 SYSTICK */
#endif
}
```

可以看到，delay_init 函数使用了条件编译，来选择不同的初始化过程，如果不使用 OS 的时候，只是设置一下 SysTick 的时钟源以及确定 fac_us 值。而如果使用 OS 的时候，则会进行一些不同的配置，这里的条件编译是根据 SYS_SUPPORT_OS 这个宏来确定的，该宏在 sys.h 里面定义。

SysTick 是 MDK 定义了一个结构体（在 core_m7.h 里面），里面包含 CTRL、LOAD、VAL、CALIB 等 4 个寄存器。

SysTick->CTRL 的各位定义如图 9.1.2.1 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 9.1.2.1 SysTick->CTRL 寄存器各位定义

SysTick->LOAD 的定义如图 9.1.2.2 所示：

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时，将被重装载的值

图 9.1.2.2 SysTick->LOAD 寄存器各位定义

SysTick->VAL 的定义如图 9.1.2.3 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 9.1.2.3 SysTick->VAL 寄存器各位定义

SysTick->CALIB 不常用, 在这里我们也用不到, 故不介绍了。

HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK); 这句代码把 SysTick 的时钟选择为内核时钟, 这里需要注意的是: SysTick 的时钟源自 HCLK, 假设我们外部晶振为 8M, 然后倍频到 480MHZ, 那么 SysTick 的时钟即为 480Mhz, 也就是 SysTick 的计数器 VAL 每减 1, 就代表时间过了 1/480us。

在不使用 OS 的时候: fac_us, 为 us 延时的基数, 也就是延时 1us, Systick 定时器需要走过的时钟周期数。

当使用 OS 的时候, fac_us, 还是 us 延时的基数, 不过这个值不会被写到 SysTick->LOAD 寄存器来实现延时, 而是通过时钟摘取的办法实现的(前面已经介绍了)。而 fac_ms 则代表 ucos 自带的延时函数所能实现的最小延时时间(如 delay_ostickspersec=200, 那么 fac_ms 就是 5ms)。

9.1.3 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 34952us(最大值即 2^24/g_fac_us @g_fac_us = 480)
 * @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。利用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: 0~8947848(最大值即 2^32 / g_fac_us @g_fac_us = 480)
* @retval 无
*/
void delay_us(uint32_t nus)
{
    uint32_t ticks;
    uint32_t told, tnow, tcnt = 0;
    uint32_t reload = SysTick->LOAD; /* LOAD 的值 */
    ticks = nus * g_fac_us; /* 需要的节拍数 */
    delay_osschedlock(); /* 阻止 OS 调度, 防止打断 us 延时 */
    told = SysTick->VAL; /* 刚进入时的计数器值 */
    while (1)
    {
        tnow = SysTick->VAL;
        if (tnow != told)
        {
            if (tnow < told)
            {
                tcnt += told - tnow; /* 这里注意一下 SYSTICK 是一个递减的计数器就可以了 */
            }
            else
            {
                tcnt += reload - tnow + told;
            }
            told = tnow;
            if (tcnt >= ticks)
            {
                break; /* 时间超过/等于要延迟的时间, 则退出 */
            }
        }
    }
    delay_osschedunlock(); /* 恢复 OS 调度 */
}

```

这里就正是利用了我们前面提到的时钟摘取法, ticks 是延时 nus 需要等待的 SysTick 计数次数(也就是延时时间), told 用于记录最近一次的 SysTick->VAL 值, 然后 tnow 则是当前的 SysTick->VAL 值, 通过他们的对比累加, 实现 SysTick 计数次数的统计, 统计值存放在 tcnt 里面, 然后通过对 tcnt 和 ticks, 来判断延时是否到达, 从而达到不修改 SysTick 实现 nus 的延时, 从而可以和 OS 共用一个 SysTick。

上面的 delay_osschedlock 和 delay_osschedunlock 是 OS 提供的两个函数, 用于调度上锁和解锁, 这里为了防止 OS 在 delay_us 的时候打断延时, 可能导致的延时不准, 所以我们利用这两个函数来实现免打断, 从而保证延时精度! 同时, 此时的 delay_us,, 可以实现最长 $2^{32}/\text{fac_us}$, 在 480M 主频下, 最大延时, 大概是 8.9 秒。

9.1.4 delay_ms 函数

该函数是用来延时指定的 ms 的, 其参数 nms 为要延时的毫秒数。该函数有使用 OS 和不使用 OS 两个版本, 这里我们分别介绍, 首先是不使用 OS 的时候, 实现函数如下:

```

/***
* @brief 延时 nms
* @param nms: 要延时的 ms 数 (0< nms <= 65535)
* @retval 无
*/
void delay_ms(uint16_t nms)
{
    uint32_t repeat = nms / 30; /* 这里用 30, 是考虑到可能有超频应用, 比如 500Mhz
                                   的时候, delay_us 最大只能延时 33554us 左右了 */
    uint32_t remain = nms % 30;
}

```

```

while (repeat)
{
    delay_us(30 * 1000); /* 利用 delay_us 实现 1000ms 延时 */
    repeat--;
}

if (remain)
{
    delay_us(remain * 1000); /* 利用 delay_us, 把尾数延时 (remain ms) 给做了 */
}
}

```

该函数其实就是多次调用 `delay_us` 函数，来实现毫秒级延时的。我们做了一些处理，使得调用 `delay_us` 函数的次数减少，这样时间会更加精准。再来看看使用 OS 的时候，`delay_ms` 的实现函数如下：

```

/**
 * @brief 延时 nms
 * @param nms: 要延时的 ms 数 (0 < nms <= 65535)
 * @retval 无
 */
void delay_ms(uint16_t nms)
{
    /* 如果 os 已经在跑了，并且不是在中断里面（中断里面不能任务调度）*/
    if (delay_osrunning && delay_osintnesting == 0)
    {
        if (nms >= g_fac_ms) /* 延时的时间大于 os 的最少时间周期 */
        {
            delay_ostimedly(nms / g_fac_ms); /* OS 延时 */
        }
        nms %= g_fac_ms; /* OS 已经无法提供这么小的延时了，采用普通方式延时 */
    }
    delay_us((uint32_t)(nms * 1000)); /* 普通方式延时 */
}

```

该函数中，`delay_osrunning` 是 OS 正在运行的标志，`delay_osintnesting` 则是 OS 中断嵌套次数，必须 `delay_osrunning` 为真，且 `delay_osintnesting` 为 0 的时候，才可以调用 OS 自带的延时函数进行延时（可以进行任务调度），`delay_ostimedly` 函数就是利用 OS 自带的延时函数，实现任务级延时的，其参数代表延时的时钟节拍数（假设 `delay_ostickspersec=200`，那么 `delay_ostimedly(1)`，就代表延时 5ms）。

当 OS 还未运行的时候，我们的 `delay_ms` 就是直接由 `delay_us` 实现的，OS 下的 `delay_us` 可以实现很长的延时（达到 204 秒）而不溢出！，所以放心的使用 `delay_us` 来实现 `delay_ms`，不过由于 `delay_us` 的时候，任务调度被上锁了，所以还是建议不要用 `delay_us` 来延时很长的时间，否则影响整个系统的性能。

当 OS 运行的时候，我们的 `delay_ms` 函数将先判断延时时长是否大于等于 1 个 OS 时钟节拍 (`fac_ms`)，当大于这个值的时候，我们就通过调用 OS 的延时函数来实现（此时任务可以调度），不足 1 个时钟节拍的时候，直接调用 `delay_us` 函数实现（此时任务无法调度）。

9.1.5 HAL 库延时函数 HAL_Delay

前面我们在 7.4.2 章节介绍 `stm32h7xx_hal.c` 文件时，已经讲解过 Systick 实现延时相关函数。实际上，HAL 库提供的延时函数，只能实现简单的毫秒级别延时，没有实现 us 级别延时。我看一下 HAL 库的 `HAL_Delay` 函数原定义：

```

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

    /* Add a freq to guarantee minimum wait */
    if (wait < HAL_MAX_DELAY)
    {

```

```

    wait += (uint32_t) (uwTickFreq);
}

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

```

HAL 库实现延时功能非常简单，首先定义了一个 32 位全局变量 uwTick，在 Systick 中断服务函数 SysTick_Handler 中通过调用 HAL_IncTick 实现 uwTick 值不断增加，也就是每隔 1ms 增加 uwTickFreq，而 uwTickFreq 默认是 1。而 HAL_Delay 函数在进入函数之后先记录当前 uwTick 的值，然后不断在循环中读取 uwTick 当前值，进行减运算，得出的就是延时的毫秒数，整个逻辑非常简单也非常清晰。

但是，HAL 库的延时函数有一个局限性，在中断服务函数中使用 HAL_Delay 会引起混乱（虽然一般禁止在中断中使用延时函数），因为它是通过中断方式实现，而 Systick 的中断优先级是最低的，所以在中断中运行 HAL_Delay 会导致延时出现严重误差。所以一般情况下，推荐大家使用 ALIENTEK 提供的延时函数库。

HAL 库的 ms 级别的延时函数 `_weak void HAL_Delay(uint32_t Delay);` 它是弱定义函数，所以用户可以自己重新定义该函数。例如：我们在 `deley.c` 文件可以这样重新定义该函数：

```

/***
 * @brief HAL 库延时函数重定义
 * @param Delay 要延时的毫秒数
 * @retval None
 */
void HAL_Delay(uint32_t Delay)
{
    delay_ms(Delay);
}

```

9.2 sys 文件夹代码介绍

sys 文件夹内包含了 `sys.c` 和 `sys.h` 两个文件，主要实现下面的几个函数，以及一些汇编函数。

```

/* 设置中断向量表偏移量 */
void sys_nvic_set_vector_table(uint32_t baseaddr, uint32_t offset);
void sys_cache_enable(void); /* 使能 STM32H7 的 L1-Cache */
uint8_t sys_stm32_clock_init(uint32_t plln, uint32_t pllm, uint32_t pllp,
                             uint32_t pllq); /* 配置系统时钟 */
void sys_qspi_enable_memmapmode(uint8_t ftype); /* QSPI 进入内存映射模式 */

/* 以下为汇编函数 */
void sys_wfi_set(void); /* 执行 WFI 指令 */
void sys_intx_disable(void); /* 关闭所有中断 */
void sys_intx_enable(void); /* 开启所有中断 */
void sys_msr_msp(uint32_t addr); /* 设置栈顶地址 */

```

`sys_nvic_set_vector_table` 函数用于设置中断向量表偏移地址。`sys_stm32_clock_ini` 函数的讲解请参考 11.2.1 小节 STM32H7 时钟系统配置章节内容。

接下来我们重点看一下 `sys_cache_enable` 函数和 `sys_qspi_enable_memmapmode` 函数。

9.2.1 Cache 使能函数

STM32H7 自带了指令 Cache (I Cache) 和数据 Cache (D Cache)，使用 I/D Cache 可以缓存指令/数据，提高 CPU 访问指令/数据的速度，从而大大提高 MCU 的性能。不过，MCU 在复位后，I/D Cache 默认都是关闭的，为了提高性能，我们需要开启 I/D Cache，在 `sys.c` 里面，我们提供了如下函数：

```

/***
 * @brief      使能 STM32H7 的 L1-Cache，同时开启 D cache 的强制透写
 * @param      无
 */

```

```

 * @retval    无
 */
void sys_cache_enable(void)
{
    SCB_EnableICache(); /* 使能 I-Cache, 函数在 core_cm7.h 里面定义 */
    SCB_EnableDCache(); /* 使能 D-Cache, 函数在 core_cm7.h 里面定义 */
    SCB->CACR |= 1 << 2; /* 强制 D-Cache 透写, 如不开启透写, 实际使用中可能遇到各种问题 */
}

```

该函数，通过调用 SCB_EnableICache 和 SCB_EnableDCache 这两个函数来使能 I Cache 和 D Cache。不过，在使能 D Cache 之后，SRAM 里面的数据有可能会被缓存在 Cache 里面，此时如果有 DMA 之类的外设访问这个 SRAM 里面的数据，就有可能和 Cache 里面数据不同步，导致数据出错，为了防止这种问题，保证数据的一致性，我们设置了 D Cache 的强制透写功能（Write Through），这样 CPU 每次操作 Cache 里面的数据，同时也会更新到 SRAM 里面，保证 D Cache 和 SRAM 里面数据一致。关于 Cache 的详细介绍，请参考《STM32F7 Cache Overview》和《Level 1 cache on STM32F7 Series》（见光盘：8，STM32 参考资料 文件夹）。注意：F7 和 H7 这部分知识是通用的，所以参考 F7 的这两份文档即可，H7 的这两个资料暂时没出来。

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

- 1, SCB_DisableICache 函数，用于关闭 I Cache。
- 2, SCB_DisableDCache 函数，用于关闭 D Cache。
- 3, SCB_InvalidateDCache 函数，用于丢弃 D Cache 当前数据，重新从 SRAM 获取数据。
- 4, SCB_CleanDCache 函数，用于将 D Cache 数据回写到 SRAM 里面，同步数据。
- 5, SCB_CleanInvalidateDCache 函数，用于回写数据到 SRAM，并重新获取 D Cache 数据。

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

9.2.2 QSPI_Enable_Memmapmode 函数

该函数有三个作用：

- 1, 初始化 QSPI 接口，并使能内存映射模式；
- 2, 初始化外部 SPI FLASH，使能 QPI（QSPI）模式。
- 3, 设置 QSPI FLASH 空间的 MPU 保护。

其代码如下：

```

/**
 * @brief      QSPI 进入内存映射模式（执行 QSPI 代码必备前提）
 * @note       必须根据所使用 QSPI FLASH 的容量设置正确的 ftype 值！
 * @param      ftype: flash 类型
 * @arg        0, 普通 FLASH, 容量在 128Mbit 及以内的
 * @arg        1, 大容量 FLASH, 容量在 256Mbit 及以上的.
 * @retval     无
 */
void sys_qspi_enable_memmapmode(uint8_t ftype)
{
    uint32_t tempreg = 0;
    GPIO_InitTypeDef qspi_gpio;

    __HAL_RCC_GPIOB_CLK_ENABLE(); /* 使能 PORTB 时钟 */
    __HAL_RCC_GPIOD_CLK_ENABLE(); /* 使能 PORTD 时钟 */
    __HAL_RCC_GPIOE_CLK_ENABLE(); /* 使能 PORTE 时钟 */
    __HAL_RCC_QSPI_CLK_ENABLE(); /* QSPI 时钟使能 */
}

```

```

qspi_gpio.Pin = GPIO_PIN_6;                                /* PB6 AF10 */
qspi_gpio.Mode = GPIO_MODE_AF_PP;
qspi_gpio.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
qspi_gpio.Pull = GPIO_PULLUP;
qspi_gpio.Alternate = GPIO_AF10_QUADSPI;
HAL_GPIO_Init(GPIOB, &qspi_gpio);

qspi_gpio.Pin = GPIO_PIN_2;                                /* PB2 AF9 */
qspi_gpio.Alternate = GPIO_AF9_QUADSPI;
HAL_GPIO_Init(GPIOB, &qspi_gpio);

qspi_gpio.Pin = GPIO_PIN_11 | GPIO_PIN_12 | GPIO_PIN_13; /* PD11,12,13 AF9 */
qspi_gpio.Alternate = GPIO_AF9_QUADSPI;
HAL_GPIO_Init(GPIOD, &qspi_gpio);

qspi_gpio.Pin = GPIO_PIN_2;                                /* PE2 AF9 */
qspi_gpio.Alternate = GPIO_AF9_QUADSPI;
HAL_GPIO_Init(GPIOE, &qspi_gpio);

/* QSPI 设置，参考 QSPI 实验的 QSPI_Init 函数 */
RCC->AHB3RSTR |= 1 << 14;      /* 复位 QSPI */
RCC->AHB3RSTR &= ~(1 << 14); /* 停止复位 QSPI */

while (QUADSPI->SR & (1 << 5)); /* 等待 BUSY 位清零 */

/* QSPI 时钟源已经在 sys_stm32_clock_init() 函数中设置 */
/* 设置 CR 寄存器，这些值怎么来的，请参考 QSPI 实验/看 H750 参考手册寄存器描述分析 */
QUADSPI->CR = 0X01000310;
/* 设置 DCR 寄存器(FLASH 容量 32M(最大容量设置为 32M, 默认用 16M 的), tSHSL=3 个时钟) */
QUADSPI->DCR = 0X00180201;
QUADSPI->CR |= 1 << 0;          /* 使能 QSPI */

/*
 * 注意：QSPI QE 位的使能，在 QSPI 烧写算法里面，就已经设置了
 * 所以，这里可以不用设置 QE 位，否则需要加入对 QE 位置 1 的代码
 * 不过，代码必须通过仿真器下载，直接烧录到外部 QSPI FLASH，是不可用的
 * 如果想直接烧录到外部 QSPI FLASH 也可以用，则需要在这里添加 QE 位置 1 的代码
 *
 * 另外，对与 W25Q256，还需要使能 4 字节地址模式，或者设置 S3 的 ADP 位为 1。
 * 我们在 QSPI 烧写算法里面已经设置了 ADP=1(上电即 32 位地址模式)，因此这里也
 * 不需要发送进入 4 字节地址模式指令/设置 ADP=1 了，否则还需要设置 ADP=1
 */

/* BY/W25QXX 写使能 (0X06 指令) */
while (QUADSPI->SR & (1 << 5)); /* 等待 BUSY 位清零 */

QUADSPI->CCR = 0X000000106;        /* 发送 0X06 指令，BY/W25QXX 写使能 */

while ((QUADSPI->SR & (1 << 1)) == 0); /* 等待指令发送完成 */

QUADSPI->FCR |= 1 << 1;           /* 清除发送完成标志位 */

/* MemroyMap 模式设置 */
while (QUADSPI->SR & (1 << 5)); /* 等待 BUSY 位清零 */

QUADSPI->ABR = 0; /* 交替字节设置为 0，实际上就是 25QXX 0XEB 指令的，M0~M7 = 0 */
tempreg = 0XEB; /* INSTRUCTION[7:0]=0XEB，发送 0XEB 指令 (Fast Read QUAD I/O) */
tempreg |= 1 << 8;           /* IMODE[1:0] = 1，单线传输指令 */
tempreg |= 3 << 10;          /* ADDRESS[1:0] = 3，四线传输地址 */
tempreg |= (2 + ftype) << 12; /* ADSIZE[1:0] = 2/3,
                                24 位(ftype = 0) / 32 位(ftype = 1) 地址长度 */
tempreg |= 3 << 14;          /* ABMODE[1:0] = 3，四线传输交替字节 */

```

```

tempreg |= 0 << 16;           /* ABSIZE[1:0] = 0, 8 位交替字节(M0~M7) */
tempreg |= 4 << 18;           /* DCYC[4:0] = 4, 4 个 dummy 周期 */
tempreg |= 3 << 24;           /* DMODE[1:0] = 3, 四线传输数据 */
tempreg |= 3 << 26;           /* FMODE[1:0] = 3, 内存映射模式 */
QUADSPI->CCR = tempreg;      /* 设置 CCR 寄存器 */

/* 设置 QSPI FLASH 空间的 MPU 保护 */
SCB->SHCSR |= ~(1 << 16);    /* 禁止 MemManage */
MPU->CTRL |= ~(1 << 0);      /* 禁止 MPU */
MPU->RNR = 0;                 /* 设置保护区域编号为 0(1~7 可以给其他内存用) */
MPU->RBAR = 0x90000000;        /* 基址址为 0x9000 000, 即 QSPI 的起始地址 */
MPU->RASR = 0x0303002D;        /* 设置保护参数(禁止共用, 允许 cache, 允许缓冲),
                                 详见 MPU 实验解析 */
MPU->CTRL = (1 << 2) | (1 << 0); /* 使能 PRIVDEFENA, 使能 MPU */
SCB->SHCSR |= 1 << 16;         /* 使能 MemManage */
}

```

以上代码，可以分为 4 个部分解读：

第一部分，初始化 QSPI 相关 GPIO 及 QSPI 接口。

第二部分，设置 QSPI FLASH 进入 QSPI 模式，并设置相关参数，这里使用的是直接操作寄存器的方式，并没有调用任何函数，目的就是简化代码。这部分代码的详细理解，读者可以参考我们后续的 QSPI 实验（实验 24 QSPI 实验）的相关介绍。这里只需要直接调用就好。

第三部分，设置 STM32H750 的 QSPI 的内存映射模式，从而可以执行外部 SPI FLASH 的代码。

第四部分，设置 QSPI FLASH 空间的内存保护。

为了简化 sys_qspi_enable_memmapmode 函数，我们很多操作都做了精简处理，所以理解起来会有一定困难，如果实在看不懂，可以绕过这个，先学习后面的知识点，回头再来学习该函数的具体实现。

需要注意的是：在 sys_qspi_enable_memmapmode 函数之前的所有代码，必须不能放到外部 QSPI FLASH，因为在该函数之前，STM32H750 都是无法访问外部 QSPI FLASH 的，所以如果遇到一些代码启动不了的情况，请重点怀疑是不是在该函数之前，就调用了 QSPI FLASH 的程序？如果有，需要将这部分代码放到该函数之后才行！或者将这部分代码放内部 FLASH 运行（[方法见 3.2.8 节](#)）。

函数/指令地址，我们可以通过仿真，看反汇编窗口（Disassembly），会有地址显示，从而快速找到问题。

9.3 usart 文件夹代码介绍

该文件夹下面有 usart.c 和 usarts.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](#)。本节我们只给大家讲解比较独立的 printf 函数支持相关的知识。

9.3.1 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 Debug 模式下程序可能正常工作的情况。半主机是 ARM 目标的一种机制，用于将输入/输出请求从应用程序代码通信到运行调试器的主机。例如，此机制可用于允许 C 库中的函数（如 printf() 和 scanf()）使用主机的屏幕和键盘，而不是在目标系统上设置屏幕和键盘。这很有用，因为开发硬件通常不具有最终系统的所有输入和输出设备，如屏幕、键盘等。半主机是通过一组定义好的软件指令（如

SVC) SVC 指令（以前称为 SWI 指令）来实现的，这些指令通过程序控制生成异常。应用程序调用相应的半主机调用，然后调试代理处理该异常。调试代理（这里的调试代理是仿真器）提供与主机之间的必需通信。也就是说使用半主机模式必须使用仿真器调试。

如果想在独立环境下运行调试功能的函数，我们这里是 printf，printf 对字符 ch 处理后写入文件 f，最后使用 fputc 将文件 f 输出到显示设备。对于 PC 端的设备，fputc 通过复杂的源码，最终把字符显示到屏幕上。那我们需要做的，就是把 printf 调用的 fputc 函数重新实现，重定向 fputc 的输出，同时避免进入半主模式。

要避免半主机模式，现在主要有两种方式：一是使用 MicroLib，即微库；另一种方法是确保 ARM 应用程序中没有链接 MicroLib 的半主机相关函数，我们要取消 ARM 的半主机工作模式，这可以通过代码实现。

先说微库，ARM 的 C 微库 MicroLib 是为嵌入式设备开发的一套类似于标准 C 接口函数的精简代码库，用于替代默认 C 库，是专门针对专业嵌入式应用开发而设计的，特别适合那些对存储空间有特别要求的嵌入式应用程序，这些程序一般不在操作系统下运行。使用微库编写程序要注意其与默认 C 库之间存在的一些差异，如 main() 函数不能声明带参数，也无须返回；不支持 stdio，除了无缓冲的 stdin、stdout 和 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。

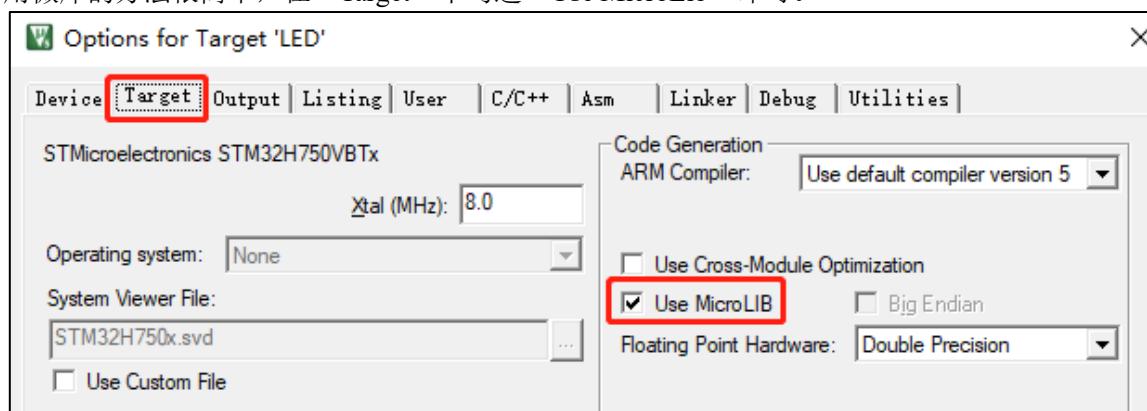


图 9.3.1.1 MDK 工程下使用微库的方法

在 keil5 中，不管是否使用半主机模式，使用 printf,scanf,fopen,fread 等都需要自己填充底层函数，以 printf 为例，需要补充定义 fputc，启用微库后，在我们初始化和使能串口 1 之后，我们只需要重新实现 fputc 的功能即可将每个传给 fputc 函数的字符 ch 重定向到串口 1，如果这时接上串口调试助手的话，可以看到串口的数据。实现的代码如下：

```
/* 重定义 fputc 函数，printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while ((USART_UX->ISR & 0X40) == 0); /* 等待上一个字符发送完成 */

    USART_UX->TDR = (uint8_t)ch;           /* 将要发送的字符 ch 写入到 DR 寄存器 */
    return ch;
}
```

上面说到了微库的一些限制，使用时注意某些函数与标准库的区别就不会影响到我们代码的正常功能。如果不想使用微库，那就要用到我们提到的第二种方法：取消 ARM 的半主机工作模式；只需在代码中添加不使用半主机的声明即可，对于 AC5 和 AC6 编译器版本，声明半主机的语法不同，为了同时兼容这两种语法，我们在利用编译器自带的宏 __ARMCC_VERSION 判定编译器版本，并根据版本不同选择不同的语法声明不使用半主机模式，具体代码如下：

```
#if (__ARMCC_VERSION >= 6010050)          /* 使用 AC6 编译器时 */
__asm(".global __use_no_semihosting\n\t"); /* 声明不使用半主机模式 */
```

```

__asm(".global __ARM_use_no_argv \n\t"); /* AC6 下需要声明 main 函数为无参数格式，否则
部分例程可能出现半主机模式 */

#else
/* 使用 AC5 编译器时，要在这里定义__FILE 和 不使用半主机模式 */
#pragma import(__use_no_semihosting)

/* 解决 HAL 库使用时，某些情况可能报错的 bug */
struct __FILE
{
    int handle;
    /* Whatever you require here. If the only file you are using is */
    /* standard output using printf() for debugging, no file handling */
    /* is required. */
};

#endif

```

使用的上面的代码，Keil 的编译器就不会把标准库的这部分函数链接到我们的代码里。如果用到原来半主机模式下的调试函数，需要重新实现它的一些依赖函数接口，对于 printf 函数需要实现的接口，我们的代码中将它们实现如下：

```

/* 不使用半主机模式，至少需要重定义_ttywrch\sys_exit\sys_command_string 函数，以同时兼容
AC6 和 AC5 模式 */
int _ttywrch(int ch)
{
    ch = ch;
    return ch;
}

/* 定义_sys_exit() 以避免使用半主机模式 */
void _sys_exit(int x)
{
    x = x;
}
char *_sys_command_string(char *cmd, int len)
{
    return NULL;
}

/* FILE 在 stdio.h 里面定义。 */
FILE __stdout;

```

fputc 的重定向和之前一样，重定向到串口 1 即可，如果硬件资源允许，读者有特殊需求，也可以重定向到 LCD 或者其它串口。

```

/* 重定义 fputc 函数，printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while (((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 STM32H750 之旅。通过本章的学习，我们将了解到 STM32H750 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 RGB 灯：LED0、LED1 和 LED2 交替闪烁，实现类似跑马灯的效果。

本章分为如下 4 个小节：

10.1 硬件设计

10.2 程序设计

10.3 下载验证

10.1 硬件设计

10.1.1. 例程功能

1. LED0 和 LED1 交替闪烁。

10.1.2. 硬件资源

1. LED

LED0 - PE5
LED1 - PE6

10.1.3. 原理图

本章实验用的两个 M100Z-M7 最小系统板 STM32H750 版板载 LED，分别为 LED0（红色）和 LED1（绿色），其与板载 MCU 的连接原理图，如下图所示：

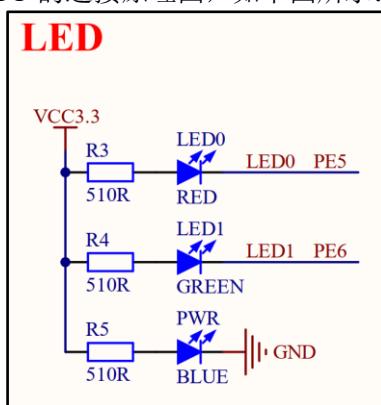


图 10.1.3.1 LED 与 MCU 连接原理图

从上面的原理图中可以看出，LED0 和 LED1 的正负极分别通过一个限流电阻接到了电源正极，而负极分别与 MCU 的 PE5 引脚和 PE6 引脚相连接，因此只需通过控制 PE5 引脚或 PE6 引脚输出低电平，则能分别控制 LED0 和 LED1 点亮，反之，则熄灭。

10.2 程序设计

10.2.1 HAL 库的 GPIO 驱动

本章实验中要通过控制 GPIO 引脚输出高低电平来控制 LED 的亮灭状态，因此需要以下两个步骤：

- ①：配置 GPIO 引脚为输出模式
- ②：设置 GPIO 引脚输出电平

在 HAL 库中对应的驱动函数如下：

①：配置 GPIO 引脚

该函数用于配置 GPIO 引脚的功能和各项参数，其函数原型如下所示：

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

该函数的形参描述，如下表所示：

形参	描述
GPIOx	指向 GPIO 端口结构体的指针 例如：GPIOA、GPIOB 等（在 stm32h750xx.h 文件中有定义）
GPIO_InitStruct	指向 GPIO 初始化结构体的指针 需自行定义，并根据 GPIO 的配置参数填充结构体中的成员变量

表 10.2.1.1 函数 HAL_GPIO_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 10.2.1.2 函数 HAL_GPIO_Init()返回值描述

该函数使用 GPIO_InitTypeDef 类型的结构体变量传入 GPIO 引脚的配置参数，该结构体的定义如下所示：

```
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 */
    uint32_t Alternate;   /* 复用功能 */
} GPIO_InitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    GPIO_InitTypeDef gpio_init_struct = {0};

    /* 配置 PA0 引脚为输出模式 */
    gpio_init_struct.Pin = GPIO_PIN_0;
    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(GPIOA, &gpio_init_struct);
}
```

②：设置 GPIO 引脚输出电平

该函数用于设置 GPIO 引脚输出指定电平（高电平或低电平），其函数原型如下所示：

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

该函数的形参描述，如下表所示：

形参	描述
GPIOx	指向 GPIO 端口结构体的指针 例如：GPIOA、GPIOB 等（在 stm32f407xx.h 文件中有定义）
GPIO_Pin	待设置的 GPIO 引脚号 例如：GPIO_PIN_0、GPIO_PIN_1 等（在 stm32f4xx_hal_gpio.h 文件中有定义）
PinState	GPIO 引脚输出的电平 GPIO_PIN_RESET：输出低电平 GPIO_PIN_SET：输出高电平

表 10.2.1.3 函数 HAL_GPIO_WritePin()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 10.2.1.4 函数 HAL_GPIO_WritePin()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32f4xx_hal.h"

void example_fun(void)
{
    /* 设置 PA0 引脚输出低电平 */
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);

    /* 设置 PA0 引脚输出高电平 */
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);
}
```

③：翻转 GPIO 引脚输出电平（补充）

该函数用于翻转 GPIO 引脚的输出电平，其函数原型如下所示：

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

该函数的形参描述，如下表所示：

形参	描述
GPIOx	指向 GPIO 端口结构体的指针 例如：GPIOA、GPIOB 等（在 stm32f407xx.h 文件中有定义）
GPIO_Pin	待设置的 GPIO 引脚号 例如：GPIO_PIN_0、GPIO_PIN_1 等（在 stm32f4xx_hal_gpio.h 文件中有定义）

表 10.2.1.5 HAL_GPIO_TogglePin()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 10.2.1.6 HAL_GPIO_TogglePin()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32f4xx_hal.h"

void example_fun(void)
{
    /* 翻转 PA0 引脚输出电平 */
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);
}
```

10.3.2 LED 驱动

LED 驱动主要就是控制 GPIO 引脚输出高低电平，来控制 LED 亮起或熄灭。本章实验中，LED 的驱动代码包括 led.c 和 led.h 两个文件（本书配套实验例程中，器件或外设的驱动代码基本都由一个 C 源文件和一个对应的头文件组成）。

根据原理图可知，应当将 PE5 引脚和 PE6 引脚配置为通用输出模式，并在需要控制 LED0 (LED1) 亮起的时候，设置 PE5 引脚 (PE6 引脚) 输出低电平，在需要控制 LED0 (LED1) 熄灭的时候，设置 PE5 引脚 (PE6 引脚) 输出高电平。

LED 驱动中，对引脚的定义，如下所示：

```
#define LED0_GPIO_PORT      GPIOE
#define LED0_GPIO_PIN        GPIO_PIN_5
#define LED0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define LED1_GPIO_PORT      GPIOE
#define LED1_GPIO_PIN        GPIO_PIN_6
#define LED1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

在后续的实验代码中，基本都会使用上述的宏定义的方式定义 GPIO 引脚的各种信息（GPIO 端口、GPIO 引脚号、GPIO 端口时钟使能和 GPIO 复用功能等信息）。

LED 驱动中，操作引脚的定义，如下所示：

```
#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)

#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)

#define LED0_TOGGLE()
do{
    HAL_GPIO_TogglePin(LED0_GPIO_PORT, LED0_GPIO_PIN);
}while(0)

#define LED1_TOGGLE()
do{
    HAL_GPIO_TogglePin(LED1_GPIO_PORT, LED1_GPIO_PIN);
}while(0)
```

在后续实验代码中，基本都会使用上述的宏定义的方式定义 GPIO 引脚的操作（设置 GPIO 引脚输出高电平或低电平、翻转 GPIO 引脚输出电平和读取 GPIO 引脚输出电平等操作）。

LED 驱动中，LED 的初始化函数，如下所示：

```
/***
 * @brief      初始化 LED
 * @param      无
 * @retval     无
 */
void led_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    LED0_GPIO_CLK_ENABLE();
    LED1_GPIO_CLK_ENABLE();

    /* 配置 LED0 引脚 */
    gpio_init_struct.Pin = LED0_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct);

    /* 配置 LED1 引脚 */
    gpio_init_struct.Pin = LED1_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct);

    /* 默认关闭所有 LED */
    LED0(1);
    LED1(1);
}
```

LED 的初始化函数中，使能了 LED0 和 LED1 控制引脚的 GPIO 端口时钟，并将其配置为通用输出模式，最后默认将 LED 的状态设置为熄灭状态。

10.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    led_init();                  /* 初始化 LED */

    while (1)
    {
        LED0(0);                /* LED0 亮 */
        LED1(1);                /* LED1 灭 */
        delay_ms(500);           /* 延时 500 毫秒 */
        LED0(1);                /* LED0 灭 */
        LED1(0);                /* LED1 亮 */
        delay_ms(500);           /* 延时 500 毫秒 */
    }
}
```

可以看到，调用 LED 初始化之前，会先调用一下函数：

```
sys_cache_enable();           /* 打开 L1-Cache */
HAL_Init();                   /* 初始化 HAL 库 */
sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
delay_init(480);              /* 初始化延时功能 */
```

- ①：第一行代码是调用系统级别的初始化：sys_cache_enable 函数使能 I-Cache 和 D-Cache。
- ②：第二行代码用于初始化 HAL 库
- ③：第三行代码用于配置系统时钟，因为大部分实验例程都无需考虑低功耗应用，因此大部分实验例程配置系统时钟的参数都与本实验一致，将系统主频配置为 480MHz。
- ④：第四行代码用于初始化延时功能，为后续的应用代码提供微秒级和毫秒级的延时的功能。
- ⑤：以上四行代码都是执行一些必要的初始化，基本在后续的每一个实验例程中都会看见，后续不再赘述。

执行完必要的初始化后，紧接着初始化 LED，然后在一个 while 循环中重复地控制板载 LED0 和 LED1 轮流亮起和熄灭，轮流时间为 500 毫秒，以此实现跑马灯的效果。

10.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 和 LED1 轮流亮起和熄灭，轮流的时间大约为 500 毫秒，与预期的实验效果相符。

第十一章 按键输入实验

上一章中介绍了 GPIO 的输出模式，并用其控制 LED 的亮灭。在实际的应用尝试中，还会需要使用到 GPIO 的输入模式，来获取外部的输入信号，例如获取按键的状态。通过本章的学习，读者将学习到 GPIO 作为输入模式的使用。

本章分为如下几个小节：

11.1 硬件设计

11.2 程序设计

11.3 下载验证

11.1 硬件设计

11.1.1 例程功能

1. 按下 WKUP 按键，LED0 状态翻转。
2. 按下 KEY0 按键，LED1 状态翻转。

11.1.2 硬件资源

1. LED

LED0 - PE5
LED1 - PE6

2. 按键

WKUP - PA0
KEY0 - PA15

11.1.3 原理图

本章实验使用的两个 M100Z-M7 最小系统板 STM32H750 版板载按键，分别为 KEY0 按键和 WKUP 按键，其于板载 MCU 的连接原理图，如下图所示：

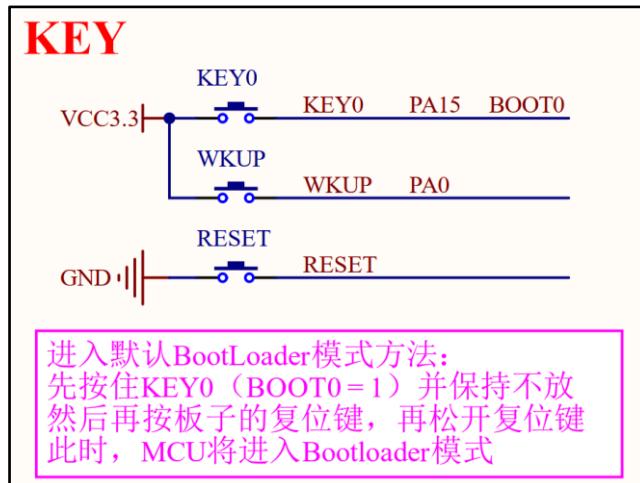


图 11.1.3.1 按键与 MCU 的连接原理图

从上面的原理图中可以看出，WKUP 按键和 KEY0 按键的一端连接到了电源正极，而另一端分别与 MCU 的 PA0 引脚和 PA15 引脚相连接，因此当任意一个按键被按下时，MCU 对应的引脚都能够读取到高电平的状态，而当松开按键后，MCU 对应的引脚读取到的电平状态却是不确定的，因此用于读取 WKUP 按键和 KEY0 按键的 PA0 引脚和 PA15 引脚不仅要配置为输入模式，还需要配置成下拉，使对应引脚在悬空时被下拉在电源负极。

11.2 程序设计

11.2.1 HAL 库的 GPIO 驱动

本章实验中要通过读取 GPIO 引脚的输入状态来判断按键是否被按下，以此来控制对应 LED 的状态翻转，因此对于判断按键的状态，需要以下几个步骤：

①：配置 GPIO 引脚为输入模式和下拉

②：读取 GPIO 引脚的输入状态

在 HAL 库中对应的驱动函数如下：

①：配置 GPIO 引脚

请见第 10.2.1 小节中配置 GPIO 引脚的相关内容。

②：读取 GPIO 引脚输入电平

该函数用于读取 GPIO 引脚的输入电平（高电平或低电平），其函数原型如下所示：

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

该函数的形参描述，如下表所示：

形参	描述
GPIOx	指向 GPIO 端口结构体的指针 例如：GPIOA、GPIOB 等（在 stm32f407xx.h 文件中有定义）
GPIO_Pin	待设置的 GPIO 引脚号 例如：GPIO_PIN_0、GPIO_PIN_1 等（在 stm32f4xx_hal_gpio.h 文件中有定义）

表 11.2.1.1 函数 HAL_GPIO_ReadPin()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
GPIO_PIN_SET	GPIO 引脚输入的电平为高电平
GPIO_PIN_RESET	GPIO 引脚输入的电平为低电平

表 11.2.1.2 函数 HAL_GPIO_ReadPin()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    GPIO_PinState value;

    /* 读取 PA0 引脚输入电平 */
    value = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0);

    if (value == GPIO_PIN_SET)
    {
        /* Do something. */
    }
    else
    {
        /* Do something. */
    }
}
```

11.2.2 按键驱动

按键驱动主要就是读取 GPIO 引脚的输入状态，以此判断按键是否被按下，本章实验中，按键的驱动代码包括 key.c 和 key.h 两个文件。

根据原理图可知，应当将 PA0 引脚和 PA15 引脚配置为输入模式和下拉，并在需要读取 WKUP 按键（KEY0 按键）状态的时候，读取 PA0 引脚（PA15 引脚）的输入电平，若读取到 PA0 引脚（PA15 引脚）的输入电平为高电平，则说明 WKUP 按键（KEY0 按键）被按下，反之，则说明 WKUP 按键（KEY0 按键）没有被按下。

按键驱动中，对引脚的定义，如下所示：

```
#define KEY0_GPIO_PORT      GPIOA
#define KEY0_GPIO_PIN        GPIO_PIN_15
#define KEY0_GPIO_CLK_ENABLE() do { __HAL_RCC_GPIOA_CLK_ENABLE(); } while (0)

#define WKUP_GPIO_PORT       GPIOA
#define WKUP_GPIO_PIN        GPIO_PIN_0
#define WKUP_GPIO_CLK_ENABLE() do { __HAL_RCC_GPIOA_CLK_ENABLE(); } while (0)
```

按键驱动中，操作引脚的定义，如下所示：

```
#define KEY0          HAL_GPIO_ReadPin(KEY0_GPIO_PORT, KEY0_GPIO_PIN)
#define WK_UP         HAL_GPIO_ReadPin(WKUP_GPIO_PORT, WKUP_GPIO_PIN)
```

按键驱动中，按键的初始化函数，如下所示：

```
/***
 * @brief      初始化按键
 * @param      无
 * @retval     无
 */
void key_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    KEY0_GPIO_CLK_ENABLE();
    WKUP_GPIO_CLK_ENABLE();

    /* 配置 KEY0 引脚 */
    gpio_init_struct.Pin = KEY0_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_INPUT;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(KEY0_GPIO_PORT, &gpio_init_struct);

    /* 配置 WKUP 引脚 */
    gpio_init_struct.Pin = WKUP_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_INPUT;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(WKUP_GPIO_PORT, &gpio_init_struct);
}
```

按键的初始化函数中，使能了 KEY0 按键和 WKUP 按键对应引脚的 GPIO 端口时钟，并将其配置为输入模式和下拉。

按键的驱动中，扫描按键状态的函数，如下所示：

```
/***
 * @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 > 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
 *             WKUP_PRES, 2, WKUP 按下
 */
uint8_t key_scan(uint8_t mode)
{
    static uint8_t key_up = 1;
    uint8_t keyval = 0;

    if (mode == 1)
    {
        key_up = 1;
    }
```

```
if (key_up && (KEY0 == 1 || WK_UP == 1))
{
    delay_ms(10);
    key_up = 0;

    if (KEY0 == 1)
    {
        keyval = KEY0_PRES;
    }

    if (WK_UP == 1)
    {
        keyval = WKUP_PRES;
    }
}
else if (KEY0 == 0 && WK_UP == 0)
{
    key_up = 1;
}

return keyval;
}
```

以上函数代码就实现了按键扫描，且具有按键消抖的功能。

11.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    uint8_t key;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */

    while (1)
    {
        key = key_scan(0); /* 扫描按键 */

        switch (key)
        {
            case KEY0_PRES: /* KEY0 被按下 */
            {
                LED0_TOGGLE(); /* LED0 状态翻转 */
                break;
            }
            case WKUP_PRES: /* WKUP 被按下 */
            {
                LED1_TOGGLE(); /* LED1 状态翻转 */
                break;
            }
            default:
            {
                break;
            }
        }

        delay_ms(10);
    }
}
```

可以看到应用代码中，在初始化完 LED 和按键后，就进入了一个 while 循环，在循环中，每隔 10 毫秒就调用 key_scan() 函数扫描以此按键的状态，如果扫描到 KEY0 按键或 WKUP 按键被按下，则反转对应 LED 的亮灭状态。

11.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 和 LED1 默认是处于熄灭的状态，若此时按下并释放一次 WKUP 按键，则能够看到 LED0 的亮灭状态发生了一次翻转，同样的，若此时按下并释放一次 KEY0 按键，则能够看到 LED1 的亮灭状态发生了一次翻转，与预期的实验现象效果相符。

第十二章 外部中断实验

前两章介绍了 GPIO 在输出模式和输入模式下最基本的操作，本章将介绍如何将 GPIO 引脚作为外部中断输入来使用。通过本章的学习，读者将学习到 GPIO 作为外部中断输入的使用。

本章分为如下几个小节：

- 12.1 硬件设计
- 12.2 程序设计
- 12.3 下载验证

12.1 硬件设计

12.1.1 例程功能

1. 按下 WKUP 按键，LED0 状态翻转。
2. 按下 KEY0 按键，LED1 状态翻转。

12.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 按键
 - WKUP - PA0
 - KEY0 - PA15

12.1.3 原理图

本章实验使用的两个 M100Z-M4 最小系统板 STM32F407 版板载按键，分别为 KEY0 按键和 WKUP 按键，其于板载 MCU 的连接原理图，如下图所示：

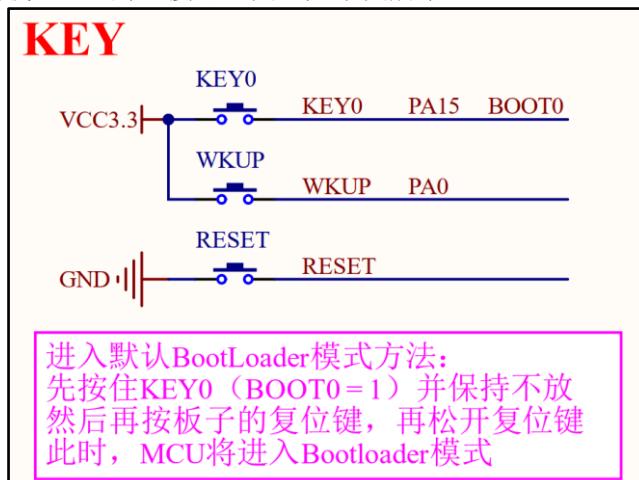


图 12.1.3.1 按键与 MCU 的连接原理图

从上面的原理图中可以看出，WKUP 按键和 KEY0 按键的一端连接到了电源正极，而另一端分别与 MCU 的 PA0 引脚和 PA15 引脚相连接，并且在上一小节的介绍中，也说明了对于该硬件设计，PA0 引脚和 PA15 引脚应当配置为下拉，这样一来，在按键被按下的时候，PA0 引脚或 PA15 引脚就会从原来的低电平状态变为高电平状态，在这期间就会有一个上升沿的跳变，因此可以使用该上升沿信号作为中断的触发源。

12.2 程序设计

12.2.1 HAL 库的 GPIO 驱动

针对本章的实验要求，仅需将对应的 GPIO 引脚配置为下拉并且上升沿触发中断模式即可，请见第 10.2.1 小节中配置 GPIO 引脚的相关内容。

①：配置 EXTI 中断

该函数用于配置 NVIC（嵌套向量中断控制器）中外设的中断优先级，因此不仅能用于配置 EXTI 的中断，还能配置其他外设等的中断，其函数原型如下所示：

```
void HAL_NVIC_SetPriority( IRQn_Type IRQn,
                           uint32_t PreemptPriority,
                           uint32_t SubPriority);
```

该函数的形参描述，如下表所示：

形参	描述
IRQn	NVIC 中断请求 例如：EXTI0_IRQHandler、USART1_IRQHandler 等（在 stm32h750xx.h 文件中有定义）
PreemptPriority	抢占优先级
SubPriority	子优先级

表 12.2.1.1 函数 HAL_NVIC_SetPriority()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 12.2.1.2 函数 HAL_NVIC_SetPriority()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 配置 EXTI0 的中断优先级 */
    HAL_NVIC_SetPriority(EXTI0_IRQHandler, 0, 0);
}
```

②：使能 EXTI 中断

该函数用于使能 NVIC（嵌套向量中断控制器）中外设的中断优先级，因此不仅能用于使能 EXTI 的中断，还能使能其他外设等的中断，其函数原型如下所示：

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

该函数的形参描述，如下表所示：

形参	描述
IRQn	NVIC 中断请求 例如：EXTI0_IRQHandler、USART1_IRQHandler 等（在 stm32h750xx.h 文件中有定义）

表 12.2.1.3 函数 HAL_NVIC_EnableIRQ()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 12.2.1.4 函数 HAL_NVIC_EnableIRQ()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 使能 EXTI0 中断 */
    HAL_NVIC_EnableIRQ(EXTI0_IRQHandler);
}
```

12.2.2 外部中断驱动

本实验的外部中断驱动主要就是配置 GPIO 引脚作为 EINT 的外部中断源，并在其对应的中断回调函数中处理按键被按下后执行的操作。

外部中断驱动中，对 GPIO 引脚、SYSCFG、EINT 的相关定义，如下所示：

```
#define KEY0_INT_GPIO_PORT      GPIOA
#define KEY0_INT_GPIO_PIN        GPIO_PIN_15
#define KEY0_INT_GPIO_CLK_ENABLE() \
    do { \
        __HAL_RCC_GPIOA_CLK_ENABLE(); \
    } while (0)
#define KEY0_INT_EXTI_LINE       EXTI_LINE_15
#define KEY0_INT IRQn
#define KEY0_INT_IRQHandler      EXTI15_10_IRQHandler

#define WKUP_INT_GPIO_PORT      GPIOA
#define WKUP_INT_GPIO_PIN        GPIO_PIN_0
#define WKUP_INT_GPIO_CLK_ENABLE() \
    do { \
        __HAL_RCC_GPIOA_CLK_ENABLE(); \
    } while (0)
#define WKUP_INT_EXTI_LINE       EXTI_LINE_0
#define WKUP_INT IRQn
#define WKUP_INT_IRQHandler      EXTI0_IRQHandler
```

外部中断驱动中，外部中断的初始化函数，如下所示：

```
/***
 * @brief      外部中断初始化程序
 * @param      无
 * @retval     无
 */
void extix_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    KEY0_INT_GPIO_CLK_ENABLE();
    WKUP_INT_GPIO_CLK_ENABLE();

    /* 配置 KEY0 引脚和外部中断 */
    gpio_init_struct.Pin = KEY0_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(KEY0_INT_GPIO_PORT, &gpio_init_struct);
    HAL_NVIC_SetPriority(KEY0_INT_IRQn, 2, 0);
    HAL_NVIC_EnableIRQ(KEY0_INT_IRQn);

    /* 配置 WKUP 引脚和外部中断 */
    gpio_init_struct.Pin = WKUP_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_RISING;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(WKUP_INT_GPIO_PORT, &gpio_init_struct);
    HAL_NVIC_SetPriority(WKUP_INT_IRQn, 2, 0);
    HAL_NVIC_EnableIRQ(WKUP_INT_IRQn);
}
```

在外部中断的初始化函数中，除了使能按键对应 GPIO 端口的时钟，还配置了对应的 EXTI 中断。

在外部中断的初始化函数中，开启了两个 EXTI 的中断，因此需要编写这两个 EXTI 中断对应的中断回调函数，如下所示：

```
/***
 * @brief      WKUP 按键外部中断服务函数
 * @param      无
 * @retval     无
 */
```

```

/*
void EXTI0_IRQHandler(void)
{
    delay_ms(20); /* 消抖 */
    HAL_GPIO_EXTI_IRQHandler(WKUP_INT_GPIO_PIN); /* 调用中断处理公用函数 */
}

/**
 * @brief      KEY0 按键外部中断服务函数
 * @param      无
 * @retval     无
 */
void EXTI15_10_IRQHandler(void)
{
    delay_ms(10); /* 消抖 */
    HAL_GPIO_EXTI_IRQHandler(KEY0_INT_GPIO_PIN); /* 调用中断处理公用函数 */
}

/**
 * @brief      外部中断回调函数
 * @param      GPIO_Pin: 中断引脚号
 * @note       在 HAL 库中所有的外部中断服务函数都会调用此函数
 * @retval     无
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    delay_ms(20);
    switch (GPIO_Pin)
    {
        case KEY0_INT_GPIO_PIN:
        {
            if (KEY0 == 0) /* KEY0 中断 */
            {
                LED0_TOGGLE(); /* LED0 状态翻转 */
            }
            break;
        }
        case WKUP_INT_GPIO_PIN:
        {
            if (WK_UP == 1) /* KEY1 中断 */
            {
                LED1_TOGGLE(); /* LED1 状态翻转 */
            }
            break;
        }
    }
}

```

在按键被按下的一瞬间，其与 MCU 连接的 GPIO 引脚会收到一个上升沿的信号，就会触发 EXTI 的中断，并运行对应中断回调函数。

12.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    led_init(); /* 初始化 LED */
    extix_init(); /* 初始化外部中断 */
}

```

```
while (1)
{
    delay_ms(1000);
}
```

本实验的应用代码很简单，在初始化完 LED 和外部表中断后，就进入一个设置了 1000ms 的延时函数的 while 循环。

12.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 和 LED1 默认是处于熄灭的状态，若此时按下 WKUP 按键，则能够看到 LED0 的亮灭状态发生了一次翻转，同样的，若此时按下 KEY0 按键，则能够看 LED1 的亮灭状态发生了一次翻转，与预期的实验现象效果相符。

第十三章 串口通信实验

本章将介绍使用串口进行数据的收发操作，具体实现 STM32H750 与上位机软件的数据通信，STM32H750 将接受自上位机软件的数据原原本本地发送回给上位机软件。通过本章的学习，读者将学习到 USART 和 GPIO 引脚复用的使用。

本章分为以下几个小节：

- 13.1 硬件设计
- 13.2 程序设计
- 13.3 下载验证

13.1 硬件设计

13.1.1 例程功能

1. 回显串口接收到的数据
2. 每间隔一定时间，串口发送一段提示信息
3. LED0 闪烁，提示程序正在运行

13.1.2 硬件资源

1. USART1
 - USART1_TX - PA9
 - USART1_RX - PA10
2. LED
 - LED0 - PE5

13.1.3 原理图

本章实验使用的 USART1 引脚通过排针引出，其原理图如下图所示：

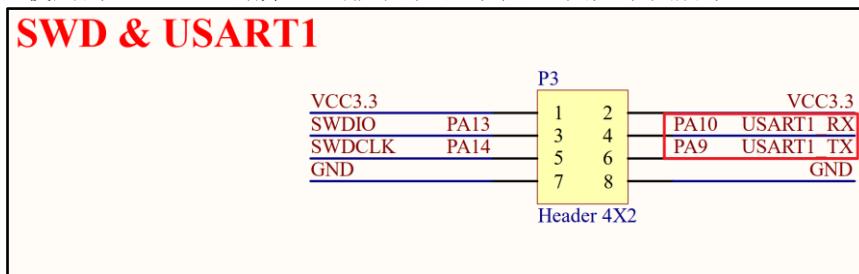


图 13.1.3.1 USART1 引脚引出

13.2 程序设计

13.2.1 HAL 库的 GPIO 驱动

在前面的章节中，介绍了 GPIO 引脚在输出模式和输入模式下的使用，但本章要使用 GPIO 引脚作为串口输出的收发引脚，因此不再是使用输出模式或输入模式，而是要使用 GPIO 的复用功能模式，例如将 PA9 引脚复用为 USART1 的数据发送引脚或将 PA10 引脚复用为 USART1 的数据接收引脚，具体的步骤如下：

①：配置 GPIO 的复用功能

在 HAL 库中对应的驱动函数如下：

①：配置 GPIO 的复用功能

请见第 10.2.1 小节中配置 GPIO 引脚的相关内容。

13.2.2 HAL 库的 UART 驱动

HAL 库的 UART 驱动提供了操作 STM32F407 片上 UART 的各种 API 函数，其中就包括配置 UART、使能 UART 等函数，本章实验还使能了 UART 中断用于接收 UART 数据。配置并使用 UART 收发数据的步骤。配置 UART 的具体步骤如下所示：

①：配置 UART

②：UART 中断接收数据

在 HAL 库中对应的驱动函数如下：

①：配置 USART

该函数用于配置 USART 的各项参数，其函数的原型如下所示：

```
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart);
```

该函数的形参描述，如下表所示：

形参	描述
huart	指向 UART 句柄的指针

表 13.2.2.1 函数 HAL_UART_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 13.2.2.2 函数 HAL_UART_Init()返回值描述

该函数需要传入 UART 的句柄指针，该句柄中就包含了 UART 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t BaudRate;      /* 波特率 */
    uint32_t WordLength;    /* 位宽 */
    uint32_t StopBits;      /* 停止位 */
    uint32_t Parity;        /* 校验位 */
    uint32_t Mode;          /* 模式 */
    uint32_t HwFlowCtl;     /* 硬件流控 */
    uint32_t OverSampling;  /* 过采样 */
} UART_HandleTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    UART_HandleTypeDef uart_handle = {0};

    /* 初始化 USART1 */
    uart_handle.Instance = USART1;
    uart_handle.Init.BaudRate = 115200;
    uart_handle.Init.WordLength = UART_WORDLENGTH_8B;
    uart_handle.Init.StopBits = UART_STOPBITS_1;
    uart_handle.Init.Parity = UART_PARITY_NONE;
    uart_handle.Init.Mode = UART_MODE_TX_RX;
    uart_handle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    uart_handle.Init.OverSampling = UART_OVERSAMPLING_16;
    HAL_UART_Init(&uart_handle);
}
```

①：UART 中断接收数据

该函数用于 UART 中断接收数据，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData,
                                      uint16_t Size)
```

该函数的形参描述，如下表所示：

形参	描述
----	----

huart	指向 UART 句柄的指针
pData	指向数据接收缓冲区的指针
Size	接收数据的大小

表 13.2.2.3 函数 HAL_UART_Receive_IT()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 13.2.2.4 函数 HAL_UART_Receive_IT()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint8_t buf[10];

    /* UART 中断接收数据 */
    HAL_UART_Receive_IT(&uart_handle, buf, sizeof(buf));
}
```

13.2.3 串口通讯驱动

本实验的串口通信驱动主要是配置 USART1 并完成一些相关的初始化操作，并支持将 printf 函数重定向到 USART1 进行输出，同时向应用层提供了一个数据接收缓冲区和接收完成标志，通过这些，应用层就能够很方便地使用 USART1 进行数据传输了，本章实验中，串口通讯的驱动代码包括 usart.c 和 usart.h 两个文件。

串口通讯驱动中，对 GPIO、USART 的相关宏定义，如下所示：

```
#define USART_TX_GPIO_PORT      GPIOA
#define USART_TX_GPIO_PIN        GPIO_PIN_9
#define USART_TX_GPIO_AF         GPIO_AF7_USART1
#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
#define USART_RX_GPIO_AF         GPIO_AF7_USART1
#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
#define USART_UX_CLK_ENABLE() \
    do { \
        __HAL_RCC_USART1_CLK_ENABLE(); \
    } while (0)
```

串口通讯驱动中，USART1 的初始化函数，如下所示：

```
/***
 * @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)
{
    g_uart1_handle.Instance = USART_UX;                                /* USART1 */
```

```

g_uart1_handle.Init.BaudRate = baudrate;           /* 波特率 */
g_uart1_handle.Init.WordLength = UART_WORDLENGTH_8B; /* 字长为 8 位数据格式 */
g_uart1_handle.Init.StopBits = UART_STOPBITS_1;      /* 一个停止位 */
g_uart1_handle.Init.Parity = UART_PARITY_NONE;       /* 无奇偶校验 */
g_uart1_handle.Init.HwFlowCtl = UART_HWCONTROL_NONE; /* 无硬件流控 */
g_uart1_handle.Init.Mode = UART_MODE_TX_RX;          /* 收发模式 */
HAL_UART_Init(&g_uart1_handle);                  /* HAL_UART_Init() 会使能 UART1 */

/* 该函数会开启接收中断:标志位 UART_IT_RXNE, 并且设置接收缓冲以及接收缓冲接收最大数据量 */
HAL_UART_Receive_IT(&g_uart1_handle, (uint8_t *)g_rx_buffer, RXBUFFERSIZE);
}

```

在串口通信的初始化函数中，在使能 USART1 收发引脚的 GPIO 端口时钟和 USART1 外设时钟后，配置了 USART1 收发引脚的 GPIO 复用功能和配置其为复用功能模式，并在最后配置并使能了 USART1，同时也使能并配置了 USART1 的接收缓冲区非空中断，这么一来，USART1 的中断回调函数就会在 USART1 接收到数据的时候被调用。

串口通讯驱动中，USART1 的接收完成回调函数，如下所示：

```

/** 
 * @brief    Rx 传输回调函数
 * @param    huart: USART 句柄类型指针
 * @retval   无
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance == USART_UX)
    {
        if((g_usart_rx_sto & 0x8000) == 0)
        {
            if(g_usart_rx_sto & 0x4000)
            {
                if(g_rx_buffer[0] != 0xa)
                {
                    g_usart_rx_sto = 0;
                }
                else
                {
                    g_usart_rx_sto |= 0x8000;
                }
            }
            else
            {
                if(g_rx_buffer[0] == 0xd)
                {
                    g_usart_rx_sto |= 0x4000;
                }
                else
                {
                    g_usart_rx_buf[g_usart_rx_sto & 0X3FFF] = g_rx_buffer[0];
                    g_usart_rx_sto++;
                    if(g_usart_rx_sto > (USART_REC_LEN - 1))
                    {
                        g_usart_rx_sto = 0;
                    }
                }
            }
        }
    }
}

```

在 USART1 的接收完成回调函数中主要用于读取 USART1 接收到的数据，并将其逐一存入接收的缓冲区，并在接收到“回车”和“换行”后标志数据接收完成。

13.2.4 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t len;
    uint16_t times;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mh */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */

    while (1)
    {
        if (g_usart_rx_sto & 0x8000) /* 接收到数据 */
        {
            len = g_usart_rx_sto & 0x3FFF;
            printf("\r\n您发送的消息为: \r\n");
            HAL_UART_Transmit(&g_uart1_handle,
                               (uint8_t *)g_usart_rx_buf,
                               len,
                               HAL_MAX_DELAY);
            printf("\r\n\r\n\r\n");
            g_usart_rx_sto = 0;
        }
        else
        {
            times++;
            if ((times % 5000) == 0)
            {
                printf("\r\n正点原子 M100Z-M7 最小系统板 STM32H750 版 \
                        串口通信实验\r\n");
                printf("正点原子@ALIENTEK\r\n\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();
            }
            delay_ms(10);
        }
    }
}

```

本实验的实验代码很简单，在完成初始化后，就不断地通过串口通信驱动提供的数据接收完成标志判断数据是否接收完毕，若还未完成数据接收，则每间隔一段时间就使用 printf 函数通过 USART1 打印一段提示信息，若数据接收完毕，则将数据原原本本的使用 printf 函数通过 USART1 打印出去，实现数据的回显。

13.3 下载验证

在完成编译和烧录操作后，需要将引出 USART1 引脚的排针通过串口转 TTL 模块等工具连接至 PC。接着打开 PC 上的 ATK-XCOM 串口调试助手软件，选择好正确的 COM 端口和相关的配置后，就能看到串口调试助手上每间隔一段时间就打印一次“请输入数据，以回车键结束”，接下来就可以根据提示通过串口调试助手发送一段任意的数据（以回车换行结束），随后立马就能看到串口调试助手上显示发送出去的数据，这就是本实验实现的数据回显。

第十四章 独立看门狗实验

本章介绍 STM32H750 独立看门狗 (IWDG) 的使用，独立看门狗能够帮助 CPU 在进入错误状态或程序跑飞时进行复位。通过本章的学习，读者将学习到 IWDG 的使用。

本章分为如下几个小节：

- 14.1 硬件设计
- 14.2 程序设计
- 14.3 下载验证

14.1 硬件设计

14.1.1 例程功能

1. 若每间隔 1 秒内按一次 WKUP 按键，则 LED0 常亮，否则 LED0 闪烁

14.1.2 硬件资源

1. LED
LED0 - PE5
2. 按键
WKUP - PA0
3. IWDG

14.1.3 原理图

本章实验使用的独立看门狗为 STM32H750 的片上资源，因此并没有相应的连接原理图。

14.2 程序设计

14.2.1 HAL 库的 IWDG 驱动

在使用 IWDG 前，需要先对其进行初始化，在初始化操作中要开启 IWDG 并配置 IWDG 的预分频系数和重装载值，预分频系数和重装载值就决定了 IWDG 单次溢出的时间，因此这两个值应该根据实际的应用场景妥善进行设置，若 IWDG 的溢出时间太长，则对异常情况的反应将变得迟钝，但若设置地太短，则会误触发复位，因此需要妥善设置。使用 IWDG 的具体步骤如下：

- ①：初始化 IWDG
- ②：在 IWDG 溢出前不断地进行“喂狗”操作

在 HAL 库中对应的驱动函数如下：

①：初始化 IWDG

该函数用于初始化 IWDG，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef *hiwdg);
```

该函数的形参描述，如下表所示：

形参	描述
hiwdg	指向 IWDG 句柄的指针

表 14.2.1.1 函数 HAL_IWDG_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 14.2.1.2 函数 HAL_IWDG_Init()返回值描述

该函数需要传入 IWDG 的句柄指针，该句柄中就包含了 IWDG 的初始化配置参数结构体，

该结构体的定义如下所示：

```
typedef struct
{
    uint32_t Prescaler; /* 分频系数 */
    uint32_t Reload;   /* 重装载值 */
} IWDG_HandleTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    IWDG_HandleTypeDef iwdg_handle = {0};

    /* 初始化 IWDG */
    iwdg_handle.Instance = IWDG;
    iwdg_handle.Init.Prescaler = IWDG_PRESCALER_64;
    iwdg_handle.Init.Reload = 500;
    HAL_IWDG_Init(&iwdg_handle);
}
```

②：重装载 IWDG

该函数用于重装载 IWDG，也就是所谓的“喂狗”，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg);
```

该函数的形参描述，如下表所示：

形参	描述
hiwdg	指向 IWDG 句柄的指针

表 14.2.1.3 函数 HAL_IWDG_Refresh()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 14.2.1.4 函数 HAL_IWDG_Refresh()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 重装载 IWDG (喂狗) */
    HAL_IWDG_Refresh();
}
```

14.2.2 看门狗驱动

本章实验的看门狗驱动主要负责向应用层提供 IWDG 初始化和喂狗的操作函数，本章实验中，看门狗的驱动代码包括 wdg.c 和 wdg.h 两个文件。

看门狗驱动中，IWDG 的初始化函数，如下所示：

```
/**
 * @brief      初始化独立看门狗
 * @param      prer: IWDG_PRESCALER_4~IWDG_PRESCALER_256, 对应 4~256 分频
 * @param      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) / 32 (ms).
 * @retval     无
 */
void iwdg_init(uint8_t prer, uint32_t rlr)
{
    g_iwdg_handle.Instance = IWDG1;
    g_iwdg_handle.Init.Prescaler = prer;
    g_iwdg_handle.Init.Reload = rlr;
    g_iwdg_handle.Init.Window = IWDG_WINDOW_DISABLE;
    HAL_IWDG_Init(&g_iwdg_handle);
```

}

IWDG 的初始化函数中使能了 IWDG 并配置其预分频系数和重装载值，并进行了一次“喂狗”防止 IWDG 一使能就溢出引发复位。

看门狗驱动中，IWDG 的“喂狗”函数，如下所示：

```
/***
 * @brief 喂狗独立看门狗
 * @param 无
 * @retval 无
 */
void iwdg_feed(void)
{
    HAL_IWDG_Refresh(&g_iwdg_handler);
}
```

该函数很简单，就是重装载 IWDG 的计数值。

14.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */
    delay_ms(100); /* 延时 100ms，LED0 的变化“可见” */
    iwdg_init(IWDG_PRESCALER_64, 500); /* 初始化独立看门狗，溢出时间约 1 秒 */
    LED0(0);

    while (1)
    {
        if (key_scan(0) == WKUP_PRES) /* 如果 WK_UP 按下，则喂狗 */
        {
            iwdg_feed(); /* 喂狗 */
        }

        delay_ms(10);
    }
}
```

可以看到应用代码中，LED 初始化后，LED0 会处于默认的熄灭状态 100 毫秒，随后初始化 IWDG 并点亮 LED0，接着在 while 循环中重复判断 WKUP 按键是否被按下，若按下则进行“喂狗”操作，若在 IWDG 溢出前都为按下 WKUP 按键，则 IWDG 会触发复位，复位会导致 LED0 熄灭大约 100 毫秒（便于观察）。

14.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 每间隔一段时间（大约 1 秒）就闪烁一次，这是因为 IWDG 不断地溢出，导致的复位。接下来若以时间间隔小于 1 秒（大约）的速度频繁地按下 WKUP 按键，则可以在 IWDG 溢出前及时“喂狗”，具体的现象为 LED0 不再闪烁。

第十五章 窗口看门狗实验

本章介绍 STM32H750 窗口看门狗（WWDG）的使用，窗口看门狗与独立看门狗一样能够帮助 CPU 在进入错误状态或程序跑飞时进行复位，不过窗口看门狗相对于独立看门狗限制了“喂狗”的最小间隔，若两次“喂狗”的间隔太短，一样会产生复位。通过本章的学习，读者将学习到 WWDG 的使用。

本章分为以下几个小节：

- 15.1 硬件设计
- 15.2 程序设计
- 15.3 下载验证

15.1 硬件设计

15.1.1 例程功能

1. LED1 闪烁

15.1.2 硬件资源

1. LED
LED0 - PE5
2. WWDG

15.1.3 原理图

本章实验使用的窗口看门狗为 STM32H750 的片上资源，因此并没有相应的连接原理图。

15.2 程序设计

15.2.1 HAL 库的 WWDG 驱动

本章实验使用到了 WWDG 的提前唤醒中断，提前唤醒中断指的是 WWDG 在“喂狗”超时即将进行复位前由 WWDG 产生的中断，本章实验就在 WWDG 的提前唤醒中断服务函数中进行“喂狗”。在使用 WWDG 前，需要先对其进行初始化，在初始化中，需要使能 WWDG 并配置 WWDG 的预分频系数和窗口值，还要进行使能 WWDG 中断的相关操作，具体的步骤如下：

- ①：初始化 WWDG
- ②：在 WWDG 提前唤醒回调函数对其进行“喂狗”

在 HAL 库中对应的驱动函数如下：

①：初始化 WWDG

该函数用于初始化 WWDG，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_WWDG_Init(WWDG_HandleTypeDef *hwwdg);
```

该函数的形参描述，如下表所示：

形参	描述
hwwdg	指向 WWDG 句柄的指针

表 15.2.1.1 函数 HAL_WWDG_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 15.2.1.2 函数 HAL_WWDG_Init()返回值描述

该函数需要传入 WWDG 的句柄指针，该句柄中就包含了 WWDG 的初始化配置参数结构体，该结构体的定义如下所示：

```

typedef struct
{
    uint32_t Prescaler; /* 分频系数 */
    uint32_t Window;   /* 窗口值 */
    uint32_t Counter;  /* 计数值 */
    uint32_t EWIMode;  /* 提前唤醒模式 */
} WWDG_HandleTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    WWDG_HandleTypeDef wwdg_handle = {0};

    /* 初始化 WWDG */
    wwdg_handle.Instance = WWDG;
    wwdg_handle.Init.Prescaler = WWDG_PRESCALER_8;
    wwdg_handle.Init.Window = 0x5F;
    wwdg_handle.Init.Counter = 0x7F;
    wwdg_handle.Init.EWIMode = WWDG_EWI_ENABLE;
    HAL_WWDG_Init(&wwdg_handle);
}

```

②：重装载 WWDG 的计数值

该函数用于重装载 WWDG 的计数值，也就是所谓的“喂狗”，其函数原型如下：

```
HAL_StatusTypeDef HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwwdg);
```

该函数的形参描述，如下表所示：

形参	描述
hwwdg	指向 WWDG 句柄的指针

表 15.2.1.3 函数 HAL_WWDG_Refresh()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 15.2.1.4 函数 HAL_WWDG_Refresh()返回值描述

该函数的使用示例，如下表所示：

```

#include "apm32h7xx_hal.h"

void example_fun(void)
{
    /* 重装载 WWDG 的计数值（喂狗） */
    HAL_WWDG_Refresh();
}

```

15.2.2 看门狗驱动

本章实验的看门狗驱动主要负责向应用层提供 WWDG 的初始化函数，并实现 WWDG 的提前唤醒中断服务函数，在 WWDG 的提前唤醒中断服务函数中执行“喂狗”操作。本章实验中，看门狗的驱动代码包括 wdt.c 和 wdt.h 两个文件。

看门狗驱动中 WWDG 的初始化函数，如下所示：

```

/**
 * @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_128, 表示
 *              2^WDGTB 分频
 * @param      Fwwdg=PCLK3/(4096*2^fprer). 一般 PCLK3=120Mhz
 * @param      溢出时间=(4096*2^fprer)*(tr-0X3F)/PCLK3
 * @param      假设 fprer=4, tr=7f, PCLK3=120Mhz
 * @param      则溢出时间=4096*16*64/120Mhz=34.95ms
 */

```

```

/* @retval    无
*/
void wwdg_init(uint8_t tr, uint8_t wr, uint32_t fprer)
{
    g_wwdg_handle.Instance = WWDG1;
    g_wwdg_handle.Init.Prescaler = fprer;           /* 设置分频系数 */
    g_wwdg_handle.Init.Window = wr;                 /* 设置窗口值 */
    g_wwdg_handle.Init.Counter = tr;                /* 设置计数器值 */
    g_wwdg_handle.Init.EWIMode = WWDG_EWI_ENABLE; /* 使能窗口看门狗提前唤醒中断 */
    HAL_WWDG_Init(&g_wwdg_handle);                /* 初始化 WWDG */
}

```

从上面的代码中可以看出，WWDG 的初始化函数就是调用了函数 HAL_WWDG_Init()函数来初始化 WWDG。

看门狗驱动中，WWDG 提前唤醒回调函数，如下所示：

```

/***
 * @brief      窗口看门狗喂狗提醒中断服务回调函数
 * @param      wwdg 句柄
 * @note       此函数会被 HAL_WWDG_IRQHandler() 调用
 * @retval     无
*/
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 */
}

```

可以看到，在窗口看门狗的提前唤醒回调函数中，对 WWDG 进行了喂狗操作，同时还对 LED1 的状态进行了以此翻转（方便观察）。

15.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    sys_cache_enable();                      /* 打开 L1-Cache */
    HAL_Init();                            /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);   /* 设置时钟，480Mhz */
    delay_init(480);                      /* 延时初始化 */
    usart_init(115200);                  /* 串口初始化为 115200 */
    led_init();                           /* 初始化 LED */
    LED0(0);                             /* 点亮 LED0 (红灯) */
    delay_ms(300);                       /* 延时 300ms，LED0 的变化“可见” */
    wwdg_init(0X7F, 0X5F, WWDG_PRESCALER_16); /* 初始化窗口看门狗 */

    while (1)
    {
        LED0(1);                         /* 关闭红灯 */
    }
}

```

可以看到应用代码中，LED 初始化后会初始化窗口看门狗，若窗口看门狗正常被“喂狗”，则可以看到 LED1 闪烁。

15.3 下载验证

在完成编译和烧录操作后，可以看 LED1 不断闪烁，说明 WWDG “喂狗”正常。

第十六章 基本定时器中断实验

STM32H750 内部有多种多个定时器 (TIM)，本章介绍 STM32H750 基本定时器的使用，基本定时器是一类只能实现定时功能、没有外部接口的定时器。通过本章的学习，读者将学习到基本定时器的使用。

本章分为如下几个小节：

- 16.1 硬件设计
- 16.2 程序设计
- 16.3 下载验证

16.1 硬件设计

16.1.1 例程功能

1. LED1 以 1Hz 的频率闪烁
2. LED0 闪烁，提示程序正在运行

16.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. TIM6

16.1.3 原理图

本章实验使用的 TIM6 为 STM32H750 的片上资源，因此并没有相应的连接原理图。

16.2 程序设计

16.2.1 HAL 库的 TIM 驱动

本章实验将配置 TIM6 每间隔 500 毫秒产生一次中断，并在其中断服务函数中改变一次 LED1 的状态，具体的步骤如下：

- ①：初始化 TIM6
 - ②：开启 TIM6 中断模式计数
- 在 HAL 库中对应的驱动函数如下：

①：初始化 TIM

该函数用于初始化 TIM 的各项参数，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针

表 16.2.1.1 函数 HAL_TIM_Base_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 16.2.1.2 函数 HAL_TIM_Base_Init()返回值描述

该函数需要传入 TIM 的句柄指针，该句柄中就包含了 TIM 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct
```

```
{
    uint32_t Prescaler;          /* 分频系数 */
    uint32_t CounterMode;        /* 计数模式 */
    uint32_t Period;            /* 重装载值 */
    uint32_t ClockDivision;     /* 计数分频 */
    uint32_t RepetitionCounter; /* 重复计数器 */
    uint32_t AutoReloadPreload; /* 自动重装载 */
} TIM_Base_InitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_HandleTypeDef tim_handle = {0};
    /* 初始化 TIM6 */
    tim_handle.Instance = TIM6;
    tim_handle.Init.Prescaler = 480 - 1;
    tim_handle.Init.Period = 1000 - 1;
    HAL_TIM_Base_Init(&tim_handle);
}
```

②：开启 TIM 中断模式计数

该函数用于开启 TIM 中断模式计数，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针

表 16.2.1.3 函数 HAL_TIM_Base_Start_IT()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 16.2.1.4 函数 HAL_TIM_Base_Start_IT()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启 TIM6 中断模式计数 */
    HAL_TIM_Base_Start_IT(&tim_handle);
}
```

16.2.2 基本定时器驱动

本章实验的基本定时器驱动主要负责向应用层提供基本定时器的初始化函数，并实现基本定时器的中断回调函数。本章实验中，基本定时器的驱动代码包括 btmr.c 和 btmr.h 两个文件。

基本定时器驱动中，对 TIM 的相关宏定义，如下所示：

```
#define BTIM_TIMX_INT          TIM6
#define BTIM_TIMX_INT IRQn       TIM6_DAC_IRQHandler
#define BTIM_TIMX_INT IRQHandler TIM6_DAC_IRQHandler
#define BTIM_TIMX_INT_CLK_ENABLE() do { \
                                HAL_RCC_TIM6_CLK_ENABLE(); \
                            } while (0)
```

基本定时器驱动中 TIM6 的初始化函数，如下所示：

```
/** 
 * @brief      初始化基本定时器中断
 * @note
 *             基本定时器的时钟来自 APB1，当 D2PPRE1≥2 分频的时候
 *             基本定时器的时钟为 APB1 时钟的 2 倍，而 APB1 为 120M，所以定时器时钟 = 240Mhz
 *             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;
    g_timx_handle.Init.Prescaler = psc;
    g_timx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_timx_handle.Init.Period = arr;
    HAL_TIM_Base_Init(&g_timx_handle);
    HAL_TIM_Base_Start_IT(&g_timx_handle);
}

```

从 TIM6 的初始化代码中可以看到，该函数调用了函数 HAL_TIM_Base_Init() 和函数 HAL_TIM_Base_Start_IT() 分别用来初始化 TIM6 和开启了 TIM6 的中断模式计数。

基本定时器驱动中 TIM6 的计数溢出回调函数，如下所示：

```

/**
 * @brief    HAL 库基本定时器超时中断回调函数
 * @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();
    }
}

```

从上面的代码中可以看出，在 TIM6 每次计数溢出后都会翻转一次 LED1 的状态。

16.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    sys_cache_enable();                                /* 打开 L1-Cache */
    HAL_Init();                                       /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);             /* 设置系统时钟, 480Mhz */
    delay_init(480);                                  /* 初始化延时功能 */
    usart_init(115200);                             /* 初始化串口 */
    led_init();                                     /* 初始化 LED */
    btim_timx_int_init(5000 - 1, 24000 - 1);       /* 初始化基本定时器, 溢出频率为 2Hz */
    while (1)
    {
        LED0_TOGGLE();
        delay_ms(200);
    }
}

```

从上面的代码中可以看到，TIM6 的自动重装载配置为(5000-1)，TIM6 的预分频器数值配置为(24000-1)，并且 TIM6 的时钟频率为 240MHz，因此 TIM6 的计数频率为 10KHz，且 TIM6 计数 5000 次溢出一次，因此溢出频率为 2Hz，又因为 TIM6 每溢出一次都会改变一次 LED1 的状态，因此 LED1 的闪烁频率为 1Hz。

16.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 和 LED1 都在闪烁，但闪烁频率不同，LED0 每间隔 200 毫秒改变一次状态，LED1 在 TIM6 的中断回调函数中被改变状态，其闪烁的频率约为 1Hz。

第十七章 通用定时器中断实验

本章介绍 STM32H750 通用定时器的使用，通用定时器相较于基本定时器，拥有输入捕获和输出比较等功能，这些功能可以用来测量脉冲宽度、频率和占空比，并且可以产生并输出波形等。通过本章的学习，读者将学习到通用定时器的基本使用。

本章分为如下几个小节：

- 17.1 硬件设计
- 17.2 程序设计
- 17.3 下载验证

17.1 硬件设计

17.1.1 例程功能

1. LED1 以 1Hz 的频率闪烁
2. LED0 闪烁，提示程序正在运行

17.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. TIM3

17.1.3 原理图

本章实验使用的 TIM3 为 STM32H750 的片上资源，因此没有对应的连接原理图。

17.2 程序设计

17.2.1 HAL 库的 TIM 驱动

本章实验仅是使用通用 TIM3 代替上一章中基本 TIM6，实现通用定时器的一些基本功能，具体的原理和使用的 HAL 库函数都是一样的，因此请参考第 16.2.1 小节中对 HAL 库中 TIM 驱动的相关介绍。

17.2.2 通用定时器驱动

本章实验仅是使用通用 TIM3 代替上一章中基本 TIM6，实现通用定时器的一些基本功能，具体的驱动代码都是一样的，因此请参考第 16.2.2 小节中基本定时器驱动的相关介绍。

17.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    sys_cache_enable();                                /* 打开 L1-Cache */
    HAL_Init();                                         /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);               /* 设置系统时钟，480Mhz */
    delay_init(480);                                    /* 初始化延时 */
    usart_init(115200);                                /* 初始化串口 */
    led_init();                                         /* 初始化 LED */
    gtim_timx_int_init(5000 - 1, 24000 - 1);          /* 初始化通用定时器，溢出频率为 2Hz */
}
```

```
while (1)
{
    LED0_TOGGLE();
    delay_ms(200);
}
```

与上一章实验一样，TIM3 的计数频率为 10KHz，溢出频率为 2Hz，因此 LED1 的闪烁频率也为 1Hz。

17.3 下载验证

在完成编译和烧录操作后，可以看到板子上的 LED0 和 LED1 都在闪烁，但闪烁的频率不同，LED0 每间隔 200 毫秒改变一次状态，LED1 在 TIM3 的中断回调函数中被改变状态，其闪烁的频率约为 1Hz。

第十八章 通用定时器 PWM 输出实验

本章将介绍使用 STM32H750 的通用定时器输出 PWM。通过本章的学习，读者将学习到通用定时器输出比较的使用。

本章分为如下几个小节：

- 18.1 硬件设计
- 18.2 程序设计
- 18.3 下载验证

18.1 硬件设计

18.1.1 例程功能

1. PB4 引脚输出频率为 2KHz 占空比不断变化的 PWM
2. LED0 闪烁，提示程序正在运行

18.1.2 硬件资源

1. LED
LED0 - PE5
2. TIM3
CH1 - PB4

18.1.3 原理图

本章实验使用的 TIM3 为 STM32F407 的片上资源，因此没有对应的连接原理图。

18.2 程序设计

18.2.1 HAL 库的 TIM 驱动

本章实验将使用 TIM3 从 CH1 (PB4 引脚) 输出 PWM，因此需要配置通用定时器从指定通道输出 PWM，具体的步骤如下：

- ①：初始化定时器 PWM
- ②：配置定时器 PWM 输出通道
- ③：开始定时器 PWM 输出

在 HAL 库中对应的驱动函数如下：

①：初始化定时器 PWM

该函数用于初始化定时器 PWM，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef *htim);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针

表 18.2.1.1 函数 HAL_TIM_PWM_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 18.2.1.2 函数 HAL_TIM_PWM_Init()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
```

```

{
    TIM_HandleTypeDef tim_handle = {0};

    /* 初始化 TIM3 PWM */
    tim_handle.Instance = TIM3;
    tim_handle.Init.Prescaler = 240 - 1;
    tim_handle.Init.Period = 500 - 1;
    HAL_TIM_PWM_Init(&tim_handle);
}

```

②：配置定时器 PWM 输出通道

该函数用于配置定时器的 PWM 输出通道，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_PWM_ConfigChannel(  TIM_HandleTypeDef *htim,
                                              TIM_OC_InitTypeDef *sConfig,
                                              uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sConfig	指向输出比较通道配置结构体的指针 需自行定义，并根据输出比较通道配置参数填充结构体中的成员变量
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 18.2.1.3 函数 HAL_TIM_PWM_ConfigChannel()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 18.2.1.4 函数 HAL_TIM_PWM_ConfigChannel()返回值描述

该函数使用 TIM_OC_InitTypeDef 类型结构体指针传入了定时器输出比较通道的配置参数，该结构体的定义如下所示：

```

typedef struct
{
    uint32_t OCMode;          /* 输出比较模式 */
    uint32_t Pulse;           /* 占空比 */
    uint32_t OCPolarity;      /* 输出极性 */
    uint32_t OCNPolarity;     /* 互补输出极性 */
    uint32_t OCFastMode;      /* 快速模式 */
    uint32_t OCIIdleState;    /* 空闲状态 */
    uint32_t OCNIdleState;    /* 互补空闲状态 */
} TIM_OC_InitTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_OC_InitTypeDef tim_oc_init_struct = {0};

    /* 配置定时器通道 1 PWM 输出 */
    tim_oc_init_struct.OCMode = TIM_OCMODE_PWM1;
    tim_oc_init_struct.Pulse = 250 - 1;
    tim_oc_init_struct.OCPolarity = TIM_OCPOLARITY_HIGH;
    HAL_TIM_PWM_ConfigChannel(&tim_handle, &tim_oc_init_struct, TIM_CHANNEL_1);
}

```

③：开始定时器 PWM 输出

该函数用于开始定时器的 PWM 输出，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
Channel	定时器通道 例如： TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 18.2.1.5 函数 HAL_TIM_PWM_Start()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 18.2.1.6 函数 HAL_TIM_PWM_Start()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启定时器通道 1 PWM 输出 */
    HAL_TIM_PWM_Start(&tim_handle, TIM_CHANNEL_1);
}
```

18.2.2 通用定时器驱动

本章实验的通用定时器驱动主要负责向应用层提供通用定时器的初始化函数。本章实验中，通用定时器的驱动代码包括 gtim.c 和 gtim.h 两个文件。

通用定时器驱动中，对 TIM、GPIO 的相关宏定义，如下所示：

```
#define GTIM_TIMX_PWM          TIM15
#define GTIM_TIMX_PWM_CHY        TIM_CHANNEL_1
#define GTIM_TIMX_PWM_CHY_CCRX   TIM15->CCR1
#define GTIM_TIMX_PWM_CHY_CLK_ENABLE()
do{ __HAL_RCC_TIM15_CLK_ENABLE(); }while(0)

#define GTIM_TIMX_PWM_CHY_GPIO_PORT      GPIOE
#define GTIM_TIMX_PWM_CHY_GPIO_PIN       GPIO_PIN_5
#define GTIM_TIMX_PWM_CHY_GPIO_AF        GPIO_AF4_TIM15
#define GTIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

通用定时器驱动中 TIM15 的初始化函数，如下所示：

```
/**
 * @brief      初始化通用定时器 PWM 输出
 * @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 120M，所以定时器时钟 = 240Mhz
 *             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)
{
    GPIO_InitTypeDef gpio_init_struct;
    TIM_OC_InitTypeDef timx_oc_pwm_chy = {0};

    /* 使能时钟 */
    GTIM_TIMX_PWM_CHY_CLK_ENABLE();
    GTIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE();

    /* 配置 PWM 输出引脚 */
    gpio_init_struct.Pin = GTIM_TIMX_PWM_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;
gpio_init_struct.Alternate = GTIM_TIMX_PWM_CHY_GPIO_AF;
HAL_GPIO_Init(GTIM_TIMX_PWM_CHY_GPIO_PORT, &gpio_init_struct);

/* 配置通用定时器 */
g_timx_pwm_chy_handle.Instance = GTIM_TIMX_PWM;
g_timx_pwm_chy_handle.Init.Prescaler = psc;
g_timx_pwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
g_timx_pwm_chy_handle.Init.Period = arr;
HAL_TIM_PWM_Init(&g_timx_pwm_chy_handle);

/* 配置输出比较通道 */
timx_oc_pwm_chy.OCMode = TIM_OCMODE_PWM1;
timx_oc_pwm_chy.Pulse = (arr + 1) >> 1;
timx_oc_pwm_chy.OCPolarity = TIM_OCPOLARITY_LOW;

/* .....此处省略部分代码..... */

HAL_TIM_PWM_Start(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY);
}

```

从 TIM15 的初始化代码中可以看到, 初始化函数中初始化了 TIM15 的 PWM 并配置了 TIM15 的 PWM 输出通道 1, 最后开始 TIM15 的 PWM 输出。

18.2.3 实验应用代码

本章实验的应用代码, 如下所示:

```

int main(void)
{
    uint16_t ledrpwmval = 0;
    uint8_t dir = 1;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    gtim_timx_pwm_chy_init(500 - 1, 240 - 1); /* 初始化通用定时器 PWM 输出 */

    while (1)
    {
        delay_ms(10);

        /* 根据方向修改 ledrpwmval */
        if (dir == 1)
        {
            ledrpwmval++; /* dir==1 ledrpwmval 递增 */
        }
        else
        {
            ledrpwmval--; /* dir==0 ledrpwmval 递减 */
        }

        if (ledrpwmval > 300)
        {
            dir = 0; /* ledrpwmval 到达 300 后, 方向为递减 */
        }

        if (ledrpwmval == 0)
        {
            dir = 1; /* ledrpwmval 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,
                      ledrpwmval);
}
```

从上面的代码中可以看到，在初始化完 TIM15 输出 PWM 后，就不断地改变 TIM15 通道 1 的比较值，以达到改变 PWM 占空比。

18.3 下载验证

下载代码后，定时器 15 通道 1 输出 PWM 信号到 PB4 口。可以看到 LED0 不停的由暗变到亮，然后又从亮变到暗。

第十九章 通用定时器输入捕获实验

本章将介绍使用 STM32H750 通用定时器的输入捕获功能。通过本章的学习，读者将学习到通用定时器输入捕获的使用。

本章分为以下几个小节：

19.1 硬件设计

19.2 程序设计

19.3 下载验证

19.1 硬件设计

19.1.1 例程功能

1. 按下并松开 WKUP 按键，串口输出 WKUP 被按下的时长
2. LED0 闪烁，提示程序正在运行

19.1.2 硬件资源

1. LED
LED0 - PE5
2. 按键
WKUP - PA0
3. TIM5
CH1 - PA0
4. USART1
USART1_TX - PA9
USART1_RX - PA10

19.1.3 原理图

本章实验使用的 TIM5 为 STM32F407 的片上资源，因此没有对应的连接原理图。

19.2 程序设计

19.2.1 HAL 库的 TIM 驱动

本章实验将使用 TIM5 的通道 1 捕获 WKUP 按键被按下的高电平脉冲宽度，具体的步骤如下：

- ①：初始化定时器输入捕获
- ②：配置定时器输入捕获通道
- ③：开始定时器输入捕获

在 HAL 库中对应的驱动函数如下：

①：初始化定时器输入捕获

该函数用于初始化定时器输入捕获，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针

表 19.2.1.1 函数 HAL_TIM_IC_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
-----	----

HAL_StatusTypeDef	HAL 状态
-------------------	--------

表 19.2.1.2 函数 HAL_TIM_IC_Init()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_HandleTypeDef tim_handle = {0};

    /* 初始化 TIM5 输入捕获 */
    tim_handle.Instance = TIM5;
    tim_handle.Init.Prescaler = 240 - 1;
    tim_handle.Init.Period = 0xFFFF;
    HAL_TIM_IC_Init(&tim_handle);
}
```

②：配置定时器输入捕获通道

该函数用于配置定时器的输入捕获通道，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_IC_ConfigChannel(      TIM_HandleTypeDef *htim,
                                                TIM_IC_InitTypeDef *sConfig,
                                                uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sConfig	指向输入捕获通道配置结构体的指针 需自行定义，并根据输入捕获通道配置参数填充结构体中的成员变量
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 19.2.1.3 函数 HAL_TIM_IC_ConfigChannel()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 19.2.1.4 函数 HAL_TIM_IC_ConfigChannel()返回值描述

该函数使用 TIM_IC_InitTypeDef 类型结构体指针传入了定时器输入捕获通道的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t IC_Polarity;    /* 输入捕获极性 */
    uint32_t IC_Selection;   /* 输入 */
    uint32_t IC_Prescaler;   /* 输入分频系数 */
    uint32_t IC_Filter;     /* 输入滤波 */
} TIM_IC_InitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_IC_InitTypeDef tim_ic_init_struct = {0};

    /* 配置定时器通道 1 输入捕获 */
    tim_ic_init_struct.IC_Polarity = TIM_ICPOLARITY_RISING;
    tim_ic_init_struct.IC_Selection = TIM_ICSELECTION_DIRECTTI;
    tim_ic_init_struct.IC_Prescaler = TIM_ICPSC_DIV1;
    tim_ic_init_struct.IC_Filter = 0;
    HAL_TIM_IC_ConfigChannel(&tim_handle, &tim_ic_init_struct, TIM_CHANNEL_1);
}
```

③：开始定时器中断模式输入捕获

该函数用于开始定时器中断模式输入捕获，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_IC_Start_IT( TIM_HandleTypeDef *htim,
                                         uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在stm32h7xx_hal_tim.h 文件中有定义）

表 19.2.1.5 函数 HAL_TIM_IC_Start_IT()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 19.2.1.6 函数 HAL_TIM_IC_Start_IT()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启定时器通道 1 中断模式输入捕获 */
    HAL_TIM_IC_Start_IT(&tim_handle, TIM_CHANNEL_1);
}
```

19.2.2 通用定时器驱动

本章实验的通用定时器驱动主要负责向应用层提供通用定时器的初始化函数，并实现通用定时器的中断回调函数。本章实验中，通用定时器的驱动代码包括 gtim.c 和 gtim.h 两个文件。

通用定时器驱动中，对 TIM、GPIO 的相关宏定义，如下所示：

```
#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
#define GTIM_TIMX_CAP CHY_CCRX              TIM5->CCR1
#define GTIM_TIMX_CAP CHY_CLK_ENABLE()       do{ __HAL_RCC_TIM5_CLK_ENABLE(); }while(0)

#define GTIM_TIMX_CAP_CHY_GPIO_PORT          GPIOA
#define GTIM_TIMX_CAP_CHY_GPIO_PIN           GPIO_PIN_0
#define GTIM_TIMX_CAP_CHY_GPIO_AF            GPIO_AF2_TIM5
#define GTIM_TIMX_CAP_CHY_CLK_ENABLE()        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)
```

通用定时器驱动中 TIM5 的初始化函数，如下所示：

```
/**
 * @brief      初始化通用定时器输入捕获
 * @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 120M，所以定时器时钟 = 240Mhz
 *             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
 * @param      None
 */
void gtim_timx_cap_chy_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;
    TIM_IC_InitTypeDef timx_ic_cap_chy = {0};
```

```

/* 使能时钟 */
GTIM_TIMX_CAP_CHY_CLK_ENABLE();
GTIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE();

/* 配置输入捕获引脚 */
gpio_init_struct.Pin = GTIM_TIMX_CAP_CHY_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_AF_PP;
gpio_init_struct.Pull = GPIO_PULLDOWN;
gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
gpio_init_struct.Alternate = GTIM_TIMX_CAP_CHY_GPIO_AF;
HAL_GPIO_Init(GTIM_TIMX_CAP_CHY_GPIO_PORT, &gpio_init_struct);
HAL_NVIC_SetPriority(GTIM_TIMX_CAP_IRQn, 1, 3);
HAL_NVIC_EnableIRQ(GTIM_TIMX_CAP_IRQn);

/* 配置通用定时器 */
g_timx_cap_chy_handle.Instance = GTIM_TIMX_CAP;
g_timx_cap_chy_handle.Init.Prescaler = psc;
g_timx_cap_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
g_timx_cap_chy_handle.Init.Period = arr;
HAL_TIM_IC_Init(&g_timx_cap_chy_handle);

/* 配置输入捕获通道 */
timx_ic_cap_chy.ICPolarity = TIM_ICPOLARITY_RISING;
timx_ic_cap_chy.ICSelection = TIM_ICSELECTION_DIRECTTI;
timx_ic_cap_chy.ICPrescaler = TIM_ICPSC_DIV1;
timx_ic_cap_chy.ICFilter = 0;
HAL_TIM_IC_ConfigChannel(&g_timx_cap_chy_handle,
                        &timx_ic_cap_chy,
                        GTIM_TIMX_CAP_CHY);
HAL_TIM_IC_Start_IT(&g_timx_cap_chy_handle,
                    GTIM_TIMX_CAP_CHY);
}

}

```

从 TIM5 的初始化代码中可以看到，初始化函数中初始化了 TIM5 的输入捕获并配置了 TIM5 的输入捕获通道 1，最后开始 TIM5 的中断模式输入捕获。

19.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint32_t temp = 0;
    uint8_t t = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    gtim_timx_cap_chy_init(0xFFFF, 240 - 1); /* 初始化通用定时器输入捕获 */

    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 *= 0xFFFF;
            temp += g_timxchy_cap_val;
            printf("HIGH:%d us\r\n", temp);
            g_timxchy_cap_sta = 0;
        }
    }
}

```

```
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);
}
```

从上面的代码中可以看到，TIM5 的预分频系数被配置为(240-1)，同时因为 TIM5 的时钟频率为 240MHz，因此 TIM5 的计数频率为 1MHz，因此在 TIM5 成功捕获到外部输入信号的高电平后，可以直接计算出捕获到高电平的脉宽时间。

19.3 下载验证

在完成编译和烧录操作后，短暂按下并抬起 WKUP 按键，可以通过串口调试助手观察到捕获到的 WKUP 按键被接下的时间。

第二十章 通用定时器脉冲计数实验

本章将介绍使用 STM32H750 通用定时器对输入脉冲的个数进行计数。通过本章的学习，读者将学习到通用定时器从模式的使用。

本章分为如下几个小节：

- 20.1 硬件设计
- 20.2 程序设计
- 20.3 下载验证

20.1 硬件设计

20.1.1 例程功能

1. 串口输出 WKUP 按键被按下的次数，按下 KEY0 可清零计数值
2. LED0 闪烁，提示程序正在运行

20.1.2 硬件资源

1. LED
LED0 - PE5
2. 按键
WKUP - PA0
KEY0 - PA15
3. TIM2
CH1 - PA0
4. USART1
USART1_TX - PA9
USART1_RX - PA10

20.1.3 原理图

本章实验使用的 TIM2 为 STM32H750 的片上资源，因此没有对应的连接原理图。

20.2 程序设计

20.2.1 HAL 库的 TIM 驱动

本章实验将使用 WKUP 按键按下时产生的上升沿作为 TIM2 计数的触发源，因此除了像第十九章实验配置定时器输入捕获外，该需要配置通用定时器的从模式，具体的步骤如下：

①：配置定时器从模式

在 HAL 库中对应的驱动函数如下：

①：配置定时器从模式

该函数用于配置定时器从模式，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_SlaveConfigSynchro(
    TIM_HandleTypeDef *htim,
    TIM_SlaveConfigTypeDef *sSlaveConfig);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sSlaveConfig	指向从模式配置结构体的指针 需自行定义，根据从模式配置参数填充结构体中的成员变量

表 20.2.1.1 函数 HAL_TIM_SlaveConfigSynchro()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 20.2.1.2 函数 HAL_TIM_SlaveConfigSynchro() 返回值描述

该函数使用 TIM_SlaveConfigTypeDef 类型结构体指针传入了定时器从模式的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t SlaveMode;          /* 从模式 */
    uint32_t InputTrigger;       /* 触发源 */
    uint32_t TriggerPolarity;   /* 触发极性 */
    uint32_t TriggerPrescaler;  /* 触发输入分频系数 */
    uint32_t TriggerFilter;     /* 触发输入滤波 */
} TIM_SlaveConfigTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32f4xx_hal.h"

void example_fun(void)
{
    TIM_SlaveConfigTypeDef tim_slave_config_struct = {0};

    /* 配置定时器从模式 */
    tim_slave_config_struct.SlaveMode = TIM_SLAVERESET;
    tim_slave_config_struct.InputTrigger = TIM_TS_TI1FP1;
    tim_slave_config_struct.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
    tim_slave_config_struct.TriggerPrescaler = TIM_TRIGGERPRESCALER_DIV1;
    tim_slave_config_struct.TriggerFilter = 0;
    HAL_TIM_SlaveConfigSynchro(&tim_handle, &tim_slave_config_struct);
}
```

20.2.2 通用定时器驱动

本章实验的通用定时器驱动主要负责向应用层提供通用定时器的初始化、获取和清零计数值等函数，并实现通用定时器的中断回调函数。本章实验中，通用定时器的驱动代码包括 gtim.c 和 gtim.h 两个文件。

通用定时器驱动中，对 TIM、GPIO 的相关宏定义，如下所示：

```
#define GTIM_TIMX_CNT_CHY_GPIO_PORT      GPIOA
#define GTIM_TIMX_CNT_CHY_GPIO_PIN        GPIO_PIN_0
#define GTIM_TIMX_CNT_CHY_GPIO_AF         GPIO_AF1_TIM2
#define GTIM_TIMX_CNT_CHY_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#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
#define GTIM_TIMX_CNT_CHY_CLK_ENABLE()
do{ __HAL_RCC_TIM2_CLK_ENABLE(); }while(0)
```

通用定时器驱动中 TIM2 的初始化函数，如下所示：

```
/**
 * @brief      初始化通用定时器脉冲计数
 * @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)
 *            这样 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};

    /* 使能时钟 */
    GTIM_TIMX_CNT_CHY_CLK_ENABLE();
    GTIM_TIMX_CNT_CHY_GPIO_CLK_ENABLE();
    /* 配置通用定时器 */
    g_timx_cnt_chy_handle.Instance = GTIM_TIMX_CNT;
    g_timx_cnt_chy_handle.Init.Prescaler = psc;
    g_timx_cnt_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_timx_cnt_chy_handle.Init.Period = 0xFFFF;
    HAL_TIM_IC_Init(&g_timx_cnt_chy_handle);

    /* 配置输入捕获引脚 */
    gpio_init_struct.Pin = GTIM_TIMX_CNT_CHY_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GTIM_TIMX_CNT_CHY_GPIO_AF;
    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_SLAVERESET_EXTERNAL1;
    tim_slave_config.InputTrigger = TIM_TS_TI1FP1;
    tim_slave_config.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
    tim_slave_config.TriggerPrescaler = TIM_TRIGGERPRESCALER_DIV1;
    tim_slave_config.TriggerFilter = 0x0;
    HAL_TIM_SlaveConfigSynchronization(&g_timx_cnt_chy_handle,
                                       &tim_slave_config);

    /* 使能通用定时器及其相关中断 */
    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);
    HAL_TIM_IC_Start(&g_timx_cnt_chy_handle, GTIM_TIMX_CNT_CHY);
}

```

从 TIM2 的初始化代码中可以看到，初始化函数中初始化了 TIM2 的输入捕获并配置了 TIM2 的从模式，最后开始 TIM2 的中断模式输入捕获。

通用定时器驱动中，获取和清零计数值的函数，如下所示：

```

/**
 * @brief      获取通用定时器当前计数值（含溢出）
 * @param      无
 * @retval     当前计数值
 */
uint32_t gtim_timx_cnt_chy_get_count(void)
{
    uint32_t count = 0;

    count = g_timxchycnt_ofcnt * (0xFFFF + 1);           /* 计算溢出次数对应的计数值 */
    count += __HAL_TIM_GET_COUNTER(&g_timx_cnt_chy_handle); /* 加上当前 CNT 的值 */

    return count;
}

/**
 * @brief      重启通用定时器计数
 * @param      无
 * @retval     无
 */
void gtim_timx_cnt_chy_restart(void)
{
    __HAL_TIM_DISABLE(&g_timx_cnt_chy_handle);           /* 关闭定时器 TIMX */
}

```

```

g_timxchy_cnt_ofcnt = 0; /* 累加器清零 */
__HAL_TIM_SET_COUNTER(&g_timx_cnt_chy_handle, 0); /* 计数器清零 */
__HAL_TIM_ENABLE(&g_timx_cnt_chy_handle); /* 使能定时器 TIMX */
}

```

从上面的代码中可以看到，获取计数值时，就是对定时器当前的计数值和溢出次数进行计算得出的，清零计数值就是清零计数值和溢出次数。定时器的溢出次数是在 TIM2 的更新中断中累加的。

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      通用定时器中断服务函数
 * @param      无
 * @retval     无
 */
void GTIM_TIMX_CNT_IRQHandler(void)
{
    if(__HAL_TIM_GET_FLAG(&g_timx_cnt_chy_handle, TIM_FLAG_UPDATE) == SET)
    {
        g_timxchy_cnt_ofcnt++;
    }
    __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。

20.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t t = 0;
    uint8_t key = 0;
    uint32_t curcnt = 0;
    uint32_t oldcnt = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */
    gtim_timx_cnt_chy_init(0); /* 初始化通用定时器脉冲计数 */
    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。

20.3 下载验证

在完成编译和烧录操作后，连续按下 WKUP 按键，模拟产生多个脉冲，可以通过串口调试助手观察到 WKUP 按键被按下的次数，同时按下 KEY0 按键，可清零统计值。

第二十一章 高级定时器输出指定个数 PWM 实验

本章将介绍使用 STM32H750 高级定时器输出指定个数的 PWM。通过本章的学习，读者将学习到高级定时器重复计数器的使用。

本章分为如下几个小节：

- 21.1 硬件设计
- 21.2 程序设计
- 21.3 下载验证

21.1 硬件设计

21.1.1 例程功能

1. 按下 KEY0 按键，PC6 引脚输出 5 个频率为 2Hz 的 PWM
2. LED0 闪烁，提示程序正在运行

21.1.2 硬件资源

- 1. LED
LED0 - PE5
- 2. 按键
KEY0 - PA15
- 3. TIM8
CH1 - PC6

21.1.3 原理图

本章实验使用的 TIM8 为 STM32H750 的片上资源，因此没有对应的连接原理图。

21.2 程序设计

21.2.1 HAL 库的 TIM 驱动

本章实验将使用 KEY0 按键控制 TIM8 通过通道 1 (PC6 引脚) 输出指定个数的 PWM，其具体的配置步骤如下：

- ①：初始化定时器 PWM
- ②：配置定时器 PWM 输出通道
- ③：开始定时器 PWM 输出

在 HAL 库中对应的驱动函数如下：

①：初始化定时器 PWM 输出

请见第 18.2.1 小节中初始化定时器 PWM 的相关内容。

②：配置定时器 PWM 输出通道

请见第 18.2.1 小节中配置定时器 PWM 输出通道的相关内容。

③：开始定时器 PWM 输出

请见第 18.2.1 小节中开始定时器 PWM 输出的相关内容。

21.2.2 高级定时器驱动

本章实验的高级定时器驱动主要负责向应用层提供高级定时器的初始化和输出指定个数 PWM 的函数，并实现高级定时器的中断回调函数。本章实验中，高级定时器的驱动代码包括 atim.c 和 atim.h 两个文件。

高级定时器驱动中，对 TIM、GPIO 的相关宏定义，如下所示：

```

#define ATIM_TIMX_NPWM_CHY_GPIO_PORT      GPIOC
#define ATIM_TIMX_NPWM_CHY_GPIO_PIN        GPIO_PIN_6
#define ATIM_TIMX_NPWM_CHY_GPIO_AF         GPIO_AF3_TIM8
#define ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_NPWM                      TIM8
#define ATIM_TIMX_NPWM_IRQn                TIM8_UP_TIM13_IRQn
#define ATIM_TIMX_NPWM_IRQHandler          TIM8_UP_TIM13_IRQHandler
#define ATIM_TIMX_NPWM_CHY                 TIM_CHANNEL_1
#define ATIM_TIMX_NPWM_CHY_CCRX           TIM8->CCR1
#define ATIM_TIMX_NPWM_CHY_CLK_ENABLE()    do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)

```

高级定时器驱动中 TIM8 的初始化函数，如下所示：

```

/***
 * @brief      初始化高级定时器输出指定个数 PWM
 * @note       高级定时器的时钟来自 APB1, 当 D2PPRE1≥2 分频的时候
 * @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 = {0};

    /* 使能时钟 */
    ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE();
    ATIM_TIMX_NPWM_CHY_CLK_ENABLE();

    /* 配置高级定时器 */
    a_timx_npwm_chy_handle.Instance = ATIM_TIMX_NPWM;
    a_timx_npwm_chy_handle.Init.Prescaler = psc;
    a_timx_npwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    a_timx_npwm_chy_handle.Init.Period = arr;
    a_timx_npwm_chy_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    a_timx_npwm_chy_handle.Init.RepetitionCounter = 0;
    HAL_TIM_PWM_Init(&a_timx_npwm_chy_handle);

    /* 配置 PWM 输出引脚 */
    gpio_init_struct.Pin = ATIM_TIMX_NPWM_CHY_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ VERY HIGH;
    gpio_init_struct.Alternate = ATIM_TIMX_NPWM_CHY_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_NPWM_CHY_GPIO_PORT, &gpio_init_struct);

    /* 配置输出比较通道 */
    timx_oc_npwm_chy.OCMode = TIM_OCMODE_PWM1;
    timx_oc_npwm_chy.Pulse = (arr + 1) >> 1;
    timx_oc_npwm_chy.OCPolarity = TIM_OCPOLARITY_LOW;
    if (ATIM_TIMX_NPWM_CHY == TIM_CHANNEL_1)
    {
        HAL_TIM_PWM_ConfigChannel(&a_timx_npwm_chy_handle,
                                &timx_oc_npwm_chy,
                                TIM_CHANNEL_1);
    }
    else if(ATIM_TIMX_NPWM_CHY == TIM_CHANNEL_2)
    {
        HAL_TIM_PWM_ConfigChannel(&a_timx_npwm_chy_handle,
                                &timx_oc_npwm_chy,

```

```

        TIM_CHANNEL_2);
}
else if(ATIM_TIMX_NPWM_CHY == TIM_CHANNEL_3)
{
    HAL_TIM_PWM_ConfigChannel(&a_timx_n pwm_chy_handle,
                             &timx_oc_n pwm_chy,
                             TIM_CHANNEL_3);
}
else if(ATIM_TIMX_NPWM_CHY == TIM_CHANNEL_4)
{
    HAL_TIM_PWM_ConfigChannel(&a_timx_n pwm_chy_handle,
                             &timx_oc_n pwm_chy,
                             TIM_CHANNEL_4);
}

/* 使能高级定时器及其相关中断 */
HAL_NVIC_SetPriority(ATIM_TIMX_NPWM IRQn, 1, 3);
HAL_NVIC_EnableIRQ(ATIM_TIMX_NPWM IRQn);
__HAL_TIM_ENABLE_IT(&a_timx_n pwm_chy_handle, TIM_IT_UPDATE);
HAL_TIM_PWM_Start(&a_timx_n pwm_chy_handle, ATIM_TIMX_NPWM_CHY);

```

从 TIM8 的初始化代码中可以看到，初始化函数中初始化了 TIM8 的 PWM 并配置了 TIM8 的 PWM 输出通道 1，最后开始 TIM8 的 PWM 输出。

高级定时器驱动中，开启高级定时器输出指定个数 PWM 的函数，如下所示：

```

/***
 * @brief      设置高级定时器 PWM 个数
 * @param      npwm: PWM 的个数, 范围 1~2^32
 * @retval     无
 */
void atim_timx_n_pwm_chy_set(uint32_t npwm)
{
    if (npwm == 0)
    {
        return;
    }

    g_npwm_remain = npwm;
    HAL_TIM_GenerateEvent(&a_timx_n_pwm_chy_handle, TIM_EVENTSOURCE_UPDATE);
    __HAL_TIM_ENABLE(&a_timx_n_pwm_chy_handle);
}

```

该函数记录下了需要产生的 PWM 个数，因为会在 TIM8 的更新中断中处理输出 PWM，因此手动产生了一次更新事件，由于初始化函数中开启了 TIM8 的更新中断，因此随后会进入 TIM 的中断服务函数中。

高级定时器驱动中，TIM8 的中断回调函数，如下所示：

```

/***
 * @brief      高级定时器中断服务函数
 * @param      无
 * @retval     无
 */
void ATIM_TIMX_NPWM_IRQHandler(void)
{
    uint16_t npwm = 0;

    if(__HAL_TIM_GET_FLAG(&a_timx_n_pwm_chy_handle, TIM_FLAG_UPDATE) == SET)
    {
        if (g_npwm_remain >= 256)
        {
            g_npwm_remain -= 256;
            npwm = 256;
        }
        else if (g_npwm_remain % 256)
        {
            npwm = g_npwm_remain % 256;
        }
    }
}

```

```

        g_npwm_remain = 0;
    }
    if (npwm != 0)
    {
        ATIM_TIMX_NPWM->RCR = npwm - 1;
        HAL_TIM_GenerateEvent(&a_timx_npwm_chy_handle,
                               TIM_EVENTSOURCE_UPDATE);
        __HAL_TIM_ENABLE(&a_timx_npwm_chy_handle);
    }
    else
    {
        ATIM_TIMX_NPWM->CR1 &= ~(1 << 0);
    }

    __HAL_TIM_CLEAR_IT(&a_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 (TIM15 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。

21.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t key = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    key_init(); /* 初始化按键 */
    atim_timx_npwm_chy_init(5000-1, 24000-1); /* 10Khz 的计数频率, 2Hz 的 PWM 频率 */

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES) /* KEY0 按下 */
        {
            atim_timx_npwm_chy_set(5); /* 输出 5 个 PWM 脉冲 */
        }

        delay_ms(10);
    }
}

```

从上面的代码中可以看到，TIM8 的自动重装载值配置为(5000-1)，TIM8 的预分频器数值配置为(24000-1)，并且 TIM8 的时钟频率为 480MHz，因此 TIM8 的计数频率为 10KHz，且 TIM8 每计数 5000 次溢出一次，因此溢出频率为 2Hz，也就是输出 PWM 的频率为 2Hz。

初始化完 TIM8 后，就重复地扫描按键，若 KEY0 按键被按下，则输出 5 个 PWM。

21.3 下载验证

在完成编译和烧录操作后，按下 KEY0 按键，此时可以通过示波器或外接 LED 的方式观察 PC6 引脚输出了 5 个频率为 2Hz，占空比为 50% 的 PWM。

第二十二章 高级定时器输出比较模式实验

本章将介绍使用 STM32H750 的高级定时器输出多个频率、占空比相同但相位不同的 PWM。通过本章的学习，读者将学习到高级定时器输出比较的匹配时输出翻转模式的使用。

本章分为如下几个小节：

- 22.1 硬件设计
- 22.2 程序设计
- 22.3 下载验证

22.1 硬件设计

22.1.1 例程功能

1. PC6 和 PC7 引脚输出频率同为 500Hz，但相位相差 25% 的 PWM
2. LED0 闪烁，提示程序正在运行

22.1.2 硬件资源

1. LED
LED0 - PE5
2. TIM8
CH1 - PA8
CH2 - PA9
CH3 - PA10
CH4 - PA11

22.1.3 原理图

本章实验使用的 TIM1 为 STM32H750 的片上资源，因此没有对应的连接原理图。

22.2 程序设计

22.2.1 HAL 库的 TIM 驱动

本章实验将使用 TIM1 输出四个频率、占空比相同，但相位不同的 PWM，其具体步骤如下：

- ①：初始化定时器输出比较
 - ②：配置定时器输出比较通道
 - ③：开始定时器输出比较
- 在 HAL 库中对应的驱动函数如下：

①：初始化定时器输出比较

该函数用于初始化定时器输出比较，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_OC_Init(TIM_HandleTypeDef *htim);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针

表 22.2.1.1 函数 HAL_TIM_OC_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 22.2.1.2 函数 HAL_TIM_OC_Init()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_HandleTypeDef tim_handle = {0};

    /* 初始化 TIM1 输出比较 */
    tim_handle.Instance = TIM1;
    tim_handle.Init.Prescaler = 480 - 1;
    tim_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    tim_handle.Init.Period = 1000 - 1;
    tim_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    tim_handle.Init.RepetitionCounter = 0;
    HAL_TIM_OC_Init(&tim_handle);
}
```

②：配置定时器输出比较通道

该函数用于配置定时器的输出比较通道，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_OC_ConfigChannel(      TIM_HandleTypeDef *htim,
                                                TIM_OC_InitTypeDef *sConfig,
                                                uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sConfig	指向输出比较通道配置结构体的指针 需自行定义，并根据输出比较通道配置参数填充结构体中的成员变量
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 22.2.1.3 函数 HAL_TIM_OC_ConfigChannel()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 22.2.1.4 函数 HAL_TIM_OC_ConfigChannel()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_OC_InitTypeDef tim_oc_init_struct = {0};

    /* 配置定时器通道 1 输出比较 */
    tim_oc_init_struct.OCMode = TIM_OCMODE_TOGGLE;
    tim_oc_init_struct.Pulse = 500 - 1;
    tim_oc_init_struct.OCPolarity = TIM_OCPOLARITY_HIGH;
    HAL_TIM_OC_ConfigChannel(&tim_handle, &tim_oc_init_struct, TIM_CHANNEL_1);
}
```

③：开始定时器输出比较

该函数用于开始定时器的输出比较，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIM_OC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 22.2.1.5 函数 HAL_TIM_OC_Start()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 22.2.1.6 函数 HAL_TIM_OC_Start()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启定时器通道 1 输出比较 */
    HAL_TIM_OC_Start(&tim_handle, TIM_CHANNEL_1);
}
```

22.2.2 高级定时器驱动

本章实验的高级定时器驱动主要负责向应用层提供高级定时器的初始化函数。本章实验中，高级定时器的驱动代码包括 atim.c 和 atim.h 两个文件。

高级定时器驱动中，对 TIM、GPIO 的相关宏定义，如下所示：

```
#define ATIM_TIMX_COMP_CH1_GPIO_PORT      GPIOA
#define ATIM_TIMX_COMP_CH1_GPIO_PIN        GPIO_PIN_8
#define ATIM_TIMX_COMP_CH1_GPIO_AF         GPIO_AF1_TIM1
#define ATIM_TIMX_COMP_CH1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_COMP_CH2_GPIO_PORT      GPIOA
#define ATIM_TIMX_COMP_CH2_GPIO_PIN        GPIO_PIN_9
#define ATIM_TIMX_COMP_CH2_GPIO_AF         GPIO_AF1_TIM1
#define ATIM_TIMX_COMP_CH2_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_COMP_CH3_GPIO_PORT      GPIOA
#define ATIM_TIMX_COMP_CH3_GPIO_PIN        GPIO_PIN_10
#define ATIM_TIMX_COMP_CH3_GPIO_AF         GPIO_AF1_TIM1
#define ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_COMP_CH4_GPIO_PORT      GPIOA
#define ATIM_TIMX_COMP_CH4_GPIO_PIN        GPIO_PIN_11
#define ATIM_TIMX_COMP_CH4_GPIO_AF         GPIO_AF1_TIM1
#define ATIM_TIMX_COMP_CH4_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_COMP
#define ATIM_TIMX_COMP_CH1_CCRX          TIM1
#define ATIM_TIMX_COMP_CH2_CCRX          ATIM_TIMX_COMP->CCR1
#define ATIM_TIMX_COMP_CH3_CCRX          ATIM_TIMX_COMP->CCR2
#define ATIM_TIMX_COMP_CH4_CCRX          ATIM_TIMX_COMP->CCR3
#define ATIM_TIMX_COMP_CLK_ENABLE()       ATIM_TIMX_COMP->CCR4
#define ATIM_TIMX_COMP_CLK_ENABLE()       do{ __HAL_RCC_TIM1_CLK_ENABLE(); }while(0)
```

高级定时器驱动中 TIM1 的初始化函数，如下所示：

```
/***
 * @brief      初始化高级定时器输出比较模式
 * @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 APB1，当 D2PPRE2≥2 分频的时候
 *             高级定时器的时钟为 APB2 时钟的 2 倍，而 APB2 为 120M，所以定时器时钟 = 240Mhz
 *             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};
    GPIO_InitTypeDef gpio_init_struct = {0};

    /* 使能时钟 */
    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();

    /* 配置输出比较通道 1 输出引脚 */
    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;
    gpio_init_struct.Alternate = ATIM_TIMX_COMP_CH1_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_COMP_CH1_GPIO_PORT, &gpio_init_struct);

    /* 配置输出比较通道 2 输出引脚 */
    gpio_init_struct.Pin = ATIM_TIMX_COMP_CH2_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_NOPULL;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    gpio_init_struct.Alternate = ATIM_TIMX_COMP_CH2_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_COMP_CH2_GPIO_PORT, &gpio_init_struct);

    /* 配置输出比较通道 3 输出引脚 */
    gpio_init_struct.Pin = ATIM_TIMX_COMP_CH3_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_NOPULL;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    gpio_init_struct.Alternate = ATIM_TIMX_COMP_CH3_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_COMP_CH3_GPIO_PORT, &gpio_init_struct);

    /* 配置输出比较通道 4 输出引脚 */
    gpio_init_struct.Pin = ATIM_TIMX_COMP_CH4_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_NOPULL;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    gpio_init_struct.Alternate = ATIM_TIMX_COMP_CH4_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_COMP_CH4_GPIO_PORT, &gpio_init_struct);

    /* 配置高级定时器 */
    g_timx_comp_pwm_handle.Instance = ATIM_TIMX_COMP;
    g_timx_comp_pwm_handle.Init.Prescaler = psc;
    g_timx_comp_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_timx_comp_pwm_handle.Init.Period = arr;
    g_timx_comp_pwm_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    HAL_TIM_OC_Init(&g_timx_comp_pwm_handle);

    /* 配置输出比较通道 1 */
    timx_oc_comp_pwm.OCMode = TIM_OCMODE_TOGGLE;
    timx_oc_comp_pwm.Pulse = (arr + 1) >> 1;
    timx_oc_comp_pwm.OCPolarity = TIM_OCPOLARITY_HIGH;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle,
                            &timx_oc_comp_pwm,
                            TIM_CHANNEL_1);
    HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_1);
}
```

```

/* 配置输出比较通道 2 */
timx_oc_comp_pwm.OCMode = TIM_OCMODE_TOGGLE;
timx_oc_comp_pwm.Pulse = (arr + 1) >> 1;
timx_oc_comp_pwm.OCPolarity = TIM_OCPOLARITY_HIGH;
HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle,
                         &timx_oc_comp_pwm,
                         TIM_CHANNEL_2);
__HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_2);

/* 配置输出比较通道 3 */
timx_oc_comp_pwm.OCMode = TIM_OCMODE_TOGGLE;
timx_oc_comp_pwm.Pulse = (arr + 1) >> 1;
timx_oc_comp_pwm.OCPolarity = TIM_OCPOLARITY_HIGH;
HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle,
                         &timx_oc_comp_pwm,
                         TIM_CHANNEL_3);
__HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_3);

/* 配置输出比较通道 4 */
timx_oc_comp_pwm.OCMode = TIM_OCMODE_TOGGLE;
timx_oc_comp_pwm.Pulse = (arr + 1) >> 1;
timx_oc_comp_pwm.OCPolarity = TIM_OCPOLARITY_HIGH;
HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle,
                         &timx_oc_comp_pwm,
                         TIM_CHANNEL_4);
__HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_4);

/* 使能高级定时器和输出比较通道输出 */
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 TIM1 的初始化函数中, 初始化了 TIM1 的输出比较同时配置了 TIM1 的输出比较通道 1、通道 2、通道 3 以及通道 4, 最后开始了 TIM1 的输出比较通道 1、通道 2、通道 3 和通道 4。

22.2.3 实验应用代码

本章实验的应用代码, 如下所示:

```

int main(void)
{
    uint8_t t = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    atim_timx_comp_pwm_init(1000 - 1, 240 - 1); /* 1MHz 的计数频率, 1KHz 的周期 */

    /* atim_timx_comp_pwm_init 已经设置过输出比较寄存器, 这是寄存器操作的方法 */
    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 = 1000 - 1; /* 通道 4 相位 100% */

    while (1)
    {
        delay_ms(10);
        t++;

        if (t >= 20)

```

```
{  
    LED0_TOGGLE(); /* LED0 闪烁 */  
    t = 0;  
}  
}
```

从上面的代码中可以看到，TIM1 的自动重装载值配置为(1000-1)，TIM1 的预分频器数值配置为(480-1)，并且 TIM1 的时钟频率为 240MHz，因此 TIM1 的计数频率为 1MHz，且 TIM1 每计数 1000 次溢出一次，因此溢出频率为 1KHz，但 TIM1 的四个通道配置为了匹配时输出翻转模式，因此四个通道输出的 PWM 频率应为 500Hz。

22.3 下载验证

在完成编译和烧录操作后，可以通过示波器观察 PA8、PA9、PA10、PA11 这 4 个引脚输出的四路 PWM，可以发现这四路 PWM 的频率都为 500Hz、占空比都为 50%，但相位两两相差了 25%。

第二十三章 高级定时器互补输出带死区控制实验

本章将介绍使用 STM32H750 输出带死区和刹车控制的两路互补 PWM。通过本章的学习，读者将学习到高级定时器的互补输出、死区插入和刹车的功能的使用。

本章分为如下几个小节：

23.1 硬件设计

23.2 程序设计

23.3 下载验证

23.1 硬件设计

23.1.1 例程功能

1, 利用 TIM1_CH1(PC6)输出 70%占空比的 PWM, 它的互补输出通道(PA7)则是输出 30%占空比的 PWM。

2, 刹车功能, 当给刹车输入引脚(PA6)输入低电平时, 进行刹车, 即 PC6 和 PA7 停止输出 PWM。

3, LED0 闪烁指示程序运行。

23.1.2 硬件资源

1. LED

LED0 - PE5

2. TIM1

CH1 - PC6

CH1N - PA7

BKIN - PA6

23.1.3 原理图

本章实验使用的 TIM8 为 STM32H750 的片上资源, 因此没有对应的连接原理图。

23.2 程序设计

23.2.1 HAL 库的 TIM 驱动

本章实验将使用 TIM8 的通道 1 和通道 1 的互补通道输出两路带死区的互补 PWM, 同时还使用到了刹车功能, 其具体的配置步骤如下:

①: 初始化定时器 PWM

②: 配置定时器 PWM 输出通道

③: 开始定时器 PWM 输出

④: 开始定时器互补通道 PWM 输出

在 HAL 库中对应的驱动函数如下:

①: 初始化定时器 PWM 输出

请见第 18.2.1 小节中初始化定时器 PWM 的相关内容。

②: 配置定时器 PWM 输出通道

请见第 18.2.1 小节中配置定时器 PWM 输出通道的相关内容。

③: 配置定时器刹车死区时间

该函数用于配置定时器的刹车死区时间, 其函数原型如下所示:

```
HAL_StatusTypeDef HAL_TIMEx_ConfigBreakDeadTime(
    TIM_HandleTypeDef *htim,
    TIM_BreakDeadTimeConfigTypeDef *sBreakDeadTimeConfig);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sBreakDeadTimeConfig	指向刹车死区时间配置结构体的指针 需自行定义，并根据刹车死区时间配置参数填充结构体中的成员变量

表 23.2.1.1 函数 HAL_TIMEx_ConfigBreakDeadTime()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 23.2.1.2 函数 HAL_TIMEx_ConfigBreakDeadTime()返回值描述

该函数使用 TIM_BreakDeadTimeConfigTypeDef 类型结构体指针传入了定时器刹车死区时间的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t OffStateRunMode; /* 运行模式 */
    uint32_t OffStateIDLEMode; /* 空闲模式 */
    uint32_t LockLevel; /* Lock 等级 */
    uint32_t DeadTime; /* 死区时间 */
    uint32_t BreakState; /* 刹车状态 */
    uint32_t BreakPolarity; /* 刹车极性 */
    uint32_t BreakFilter; /* 刹车滤波 */
    uint32_t AutomaticOutput; /* 刹车自动恢复 */
} TIM_BreakDeadTimeConfigTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32fh7xx_hal.h"

void example_fun(void)
{
    TIM_BreakDeadTimeConfigTypeDef tim_break_dead_time_config_struct = {0};
    /* 配置定时器刹车死区时间 */
    tim_break_dead_time_config_struct.OffStateRunMode = TIM_OSSR_DISABLE;
    tim_break_dead_time_config_struct.OffStateIDLEMode = TIM_OSSI_DISABLE;
    tim_break_dead_time_config_struct.LockLevel = TIM_LOCKLEVEL_OFF;
    tim_break_dead_time_config_struct.DeadTime = 100;
    tim_break_dead_time_config_struct.BreakState = TIM_BREAK_ENABLE;
    tim_break_dead_time_config_struct.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
    tim_break_dead_time_config_struct.BreakFilter = 0;
    tim_break_dead_time_config_struct.AutomaticOutput =
        TIM_AUTOMATICOUTPUT_ENABLE;
    HAL_TIMEx_ConfigBreakDeadTime( &tim_handle,
                                  &tim_break_dead_time_config_struct);
}
```

④：开始定时器 PWM 输出

请见第 18.2.1 小节中开始定时器 PWM 输出的相关内容。

⑤：开始定时器互补通道 PWM 输出

该函数用于开始定时器的互补通道 PWM 输出，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIMEx_PWMN_Start(      TIM_HandleTypeDef *htim,
                                            uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
Channel	定时器通道 例如：TIM_CHANNEL_1、TIM_CHANNEL_2 等（在 stm32h7xx_hal_tim.h 文件中有定义）

表 23.2.1.3 函数 HAL_TIMEx_PWMN_Start()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 23.2.1.4 函数 HAL_TIMEx_PWMN_Start()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启定时器通道 1 互补通道 PWM 输出 */
    HAL_TIMEx_PWMN_Start(&tim_handle, TIM_CHANNEL_1);
}
```

23.2.2 高级定时器驱动

本章实验的高级定时器驱动主要负责向应用层提供高级定时器的初始化函数和 PWM 占空比、死区时间的配置函数。本章实验中，高级定时器的驱动代码包括 atim.c 和 atim.h 两个文件。

高级定时器驱动中，对 TIM、GPIO 相关的宏定义，如下所示：

```
#define ATIM_TIMX_CPLM_CHY_GPIO_PORT      GPIOC
#define ATIM_TIMX_CPLM_CHY_GPIO_PIN        GPIO_PIN_6
#define ATIM_TIMX_CPLM_CHY_GPIO_AF        GPIO_AF3_TIM8
#define ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_CPLM_CHYN_GPIO_PORT      GPIOA
#define ATIM_TIMX_CPLM_CHYN_GPIO_PIN        GPIO_PIN_7
#define ATIM_TIMX_CPLM_CHYN_GPIO_AF        GPIO_AF3_TIM8
#define ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_CPLM_BKIN_GPIO_PORT      GPIOA
#define ATIM_TIMX_CPLM_BKIN_GPIO_PIN        GPIO_PIN_6
#define ATIM_TIMX_CPLM_BKIN_GPIO_AF        GPIO_AF3_TIM8
#define ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_CPLM                TIM8
#define ATIM_TIMX_CPLM_CHY            TIM_CHANNEL_1
#define ATIM_TIMX_CPLM_CHY_CCRY        ATIM_TIMX_CPLM->CCR1
#define ATIM_TIMX_CPLM_CLK_ENABLE()    do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)
```

高级定时器驱动中 TIM8 的初始化函数，如下所示：

```
/** 
 * @brief      初始化高级定时器互补输出带死区控制
 * @note       配置高级定时器 TIMX 互补输出,一路 OCy1 一路 OCyN,并且可以设置死区时间
 *             高级定时器的时钟来自 APB1,当 D2PPRE2≥2 分频的时候
 *             高级定时器的时钟为 APB2 时钟的 2 倍,而 APB2 为 120M,所以定时器时钟 = 240Mhz
 *             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)
{
    TIM_OC_InitTypeDef tim_oc_cplm_pwm = {0};
    GPIO_InitTypeDef gpio_init_struct = {0};

    /* 使能时钟 */
    ATIM_TIMX_CPLM_CLK_ENABLE();
    ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE();
```

```

ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE();
ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE();

/* 配置 PWM 输出引脚 */
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 ;
gpio_init_struct.Alternate = ATIM_TIMX_CPLM_CHY_GPIO_AF;
HAL_GPIO_Init(ATIM_TIMX_CPLM_CHY_GPIO_PORT, &gpio_init_struct);

/* 配置 PWM 互补输出引脚 */
gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHYN_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_AF_PP;
gpio_init_struct.Pull = GPIO_PULLUP;
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH ;
gpio_init_struct.Alternate = ATIM_TIMX_CPLM_CHYN_GPIO_AF;
HAL_GPIO_Init(ATIM_TIMX_CPLM_CHYN_GPIO_PORT, &gpio_init_struct);

/* 配置刹车输入引脚 */
gpio_init_struct.Pin = ATIM_TIMX_CPLM_BKIN_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_AF_PP;
gpio_init_struct.Pull = GPIO_PULLUP;
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH ;
gpio_init_struct.Alternate = ATIM_TIMX_CPLM_BKIN_GPIO_AF;
HAL_GPIO_Init(ATIM_TIMX_CPLM_BKIN_GPIO_PORT, &gpio_init_struct);

/* 配置高级定时器 */
g_timx_cplm_pwm_handle.Instance = ATIM_TIMX_CPLM;
g_timx_cplm_pwm_handle.Init.Prescaler = psc;
g_timx_cplm_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
g_timx_cplm_pwm_handle.Init.Period = arr;
g_timx_cplm_pwm_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV4;
g_timx_cplm_pwm_handle.Init.AutoReloadPreload=TIM_AUTORELOAD_PRELOAD_ENABLE;
HAL_TIM_PWM_Init(&g_timx_cplm_pwm_handle);

/* 配置输出比较通道 */
tim_oc_cplm_pwm.OCMode = TIM_OCMODE_PWM1;
tim_oc_cplm_pwm.OCPolarity = TIM_OCPOLARITY_LOW;
tim_oc_cplm_pwm.OCNPolarity = TIM_OCNPOLARITY_LOW;
tim_oc_cplm_pwm.OCIdleState = TIM_OCIDLESTATE_SET;
tim_oc_cplm_pwm.OCNIdleState = TIM_OCNIDLESTATE_SET;
HAL_TIM_PWM_ConfigChannel(&g_timx_cplm_pwm_handle,
                         &tim_oc_cplm_pwm,
                         ATIM_TIMX_CPLM_CHY);

/* 设置刹车和死区 */
g_sbreak_dead_time_config.OffStateRunMode = TIM_OSSR_DISABLE;
g_sbreak_dead_time_config.OffStateIDLEMode = TIM_OSSI_DISABLE;
g_sbreak_dead_time_config.LockLevel = TIM_LOCKLEVEL_OFF;
g_sbreak_dead_time_config.BreakState = TIM_BREAK_ENABLE;
g_sbreak_dead_time_config.BreakPolarity = TIM_BREAKPOLARITY_LOW;
g_sbreak_dead_time_config.BreakFilter = 0;
g_sbreak_dead_time_config.Break2State = TIM_BREAK2_DISABLE;
g_sbreak_dead_time_config.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
HAL_TIMEx_ConfigBreakDeadTime(&g_timx_cplm_pwm_handle,
                             &g_sbreak_dead_time_config);

/* 使能高级定时器和输出比较通道输出 */
HAL_TIM_PWM_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
HAL_TIMEx_PWMN_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
}

```

从上面的代码中可以看出，初始化函数初始化了 TIM8 的 PWM 输出，同时配置了 TIM8 的 PWM 输出通道，并且也配置了 TIM8 的刹车和死区时间，最后开始了 TIM8 的 PWM 输出。

高级定时器驱动中配置 PWM 占空比和死区时间的函数，如下所示：

```
/** @brief      设置高级定时器输出比较值和死区时间
 * @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[5:0]) * 2 * tDTS
 * @arg        dtg[7:5]=110 时, 死区时间 = (32 + dtg[4:0]) * 8 * tDTS
 * @arg        dtg[7:5]=111 时, 死区时间 = (32 + dtg[4:0]) * 16 * tDTS
 * @note      tDTS = 1 / (Ft / CKD[1:0]) = 1 / 60M = 16.67ns
 * @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 */

    /* 设置输出比较值 */
    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 */
}
```

从上面的代码中可以看出，该函数配置了 TIM8 的死区时间和输出比较值，因为配置 PWM 的占空比就是配置对应通道的输出比较值。

23.2.3 实验应用代码

本章实验的应用代码，如下所示：

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

    HAL_Init();                                /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);       /* 配置时钟, 168MHz */
    delay_init(168);                          /* 初始化延时 */
    usart_init(115200);                      /* 初始化串口 */
    led_init();                                /* 初始化 LED */
    atim_timx_cplm_init(1000 - 1, 168 - 1);   /* 初始化高级定时器互补输出带死区控制 */
    atim_timx_cplm_set(300 - 1, 100);          /* 设置高级定时器互补输出带死区控制 */

    while (1)
    {
        if (++t == 20)
        {
            t = 0;
            LED0_TOGGLE();
        }

        delay_ms(10);
    }
}
```

先看 `atim_timx_cplm_pwm_init(1000 - 1, 24 - 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 23。先看预分频系数，我们设置为 24 分频，定时器 1 的时钟源频率是 2 倍的 APB2 总线时钟频率，即 240MHz，可以得到计数器的计数频率是 10MHz，即每 0.1us 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 $(999+1)*0.1\mu s = 100\mu s$ 。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 100μs，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 1.667us。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= 16.67ns。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 = $100*16.67$ ns = 1.67us。

23.3 下载验证

在完成编译和烧录操作后，可以通过示波器观察 PC6 引脚和 PA7 引脚输出的两路 PWM，可以发现，这两路 PWM 为互补 PWM，且频率为 10KHz、占空比为 30%、死区时间大约为 1.67us。应为是能了刹车和自动输出功能，因此将 PA6 引脚接入有效的低电平后，可以看到两路 PWM 都被禁止输出了，撤销 PA6 引脚接入的高电平后，可以看到两路 PWM 有自动恢复输出了。

第二十四章 高级定时器 PWM 输入模式实验

本章将介绍使用 STM32H750 的高级定时器检测输入 PWM 的占空比和周期。通过本章的学习，读者将学习到高级定时器 PWM 输入模式的使用。

本章分为如下几个小节：

24.1 硬件设计

24.2 程序设计

24.3 下载验证

24.1 硬件设计

24.1.1 例程功能

1. PC6 引脚输出频率为 100Hz 的 PWM，串口输出 PC6 引脚输入 PWM 的周期和高电平脉宽
2. LED0 闪烁，提示程序正在运行

24.1.2 硬件资源

1. LED
LED0 - PE5
2. TIM8
CH1 - PC6
3. TIM14
CH1 - PA7
4. USART1
USART1_TX - PA9
USART1_RX - PA10

24.1.3 原理图

本章实验使用的 TIM8 为 STM32H750 的片上资源，因此没有对应的连接原理图。

24.2 程序设计

24.2.1 HAL 库的 TIM 驱动

本章实验将使用 TIM8 的通道 1(PC6 引脚)在 PWM 输入模式下捕获 TIM14 从通道 1(PA7)引脚输出的 PWM (有关通用定时器输出 PWM 的相关内容，请见第十八章“通用定时器 PWM 输出实验”)，将分别捕获输入 PWM 信号的上升沿和下降沿，以此来计算输入 PWM 信号的占空比和周期，要是实现以上功能都依赖于高级定时器的 PWM 输入模式，其具体的配置步骤如下：

- ①：初始化定时器输入捕获
- ②：配置定时器从模式
- ③：配置定时器输入捕获通道
- ④：开始定时器输入捕获

在 HAL 库中对应的驱动函数如下：

①：初始化定时器输入捕获

请见第 19.2.1 小节中初始化定时器输入捕获的相关内容。

②：配置定时器从模式

请见第 20.2.1 小节中配置定时器从模式的相关内容。

③：配置定时器输入捕获通道

请见第 19.2.1 小节中配置定时器输入捕获通道的相关内容。

④：开始定时器输入捕获

请见第 19.2.1 小节中开始定时器输入捕获的相关内容。

24.2.2 高级定时器驱动

本章实验的高级定时器驱动主要负责向应用层提供高级定时器的初始化函数，并实现高级定时器的中断服务函数。本章实验中，高级定时器的驱动代码包括 atim.c 和 atim.h 两个文件。

高级定时器驱动中，对 TIM、GPIO 相关的宏定义，如下所示：

```
#define ATIM_TIMX_PWMIN_CHY_GPIO_PORT      GPIOC
#define ATIM_TIMX_PWMIN_CHY_GPIO_PIN        GPIO_PIN_6
#define ATIM_TIMX_PWMIN_CHY_GPIO_AF        GPIO_AF3_TIM8
#define ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define ATIM_TIMX_PWMIN                      TIM8
#define ATIM_TIMX_PWMIN_CC_IRQn            TIM8_CC_IRQn
#define ATIM_TIMX_PWMIN_CC_IRQHandler    TIM8_CC_IRQHandler
#define ATIM_TIMX_PWMIN_CHY                TIM_CHANNEL_1
#define ATIM_TIMX_PWMIN_CHY_CLK_ENABLE()   do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)
```

高级定时器驱动中 TIM8 的初始化函数，如下所示：

```
/***
 * @brief      初始化高级定时器 PWM 输入模式
 * @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 120M，所以定时器时钟 = 240Mhz
 *             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      psc: 预分频器数值
 * @retval     无
 */
void atim_timx_pwmin_chy_init(uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct = {0};
    TIM_SlaveConfigTypeDef slave_config = {0};
    TIM_IC_InitTypeDef tim_ic_pwmin_chy = {0};

    /* 使能时钟 */
    ATIM_TIMX_PWMIN_CHY_CLK_ENABLE();
    ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE();

    /* 配置 PWM 输入引脚 */
    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_VERY_HIGH ;
    gpio_init_struct.Alternate = ATIM_TIMX_PWMIN_CHY_GPIO_AF;
    HAL_GPIO_Init(ATIM_TIMX_PWMIN_CHY_GPIO_PORT, &gpio_init_struct);

    /* 配置高级定时器 */
    g_timx_pwmin_chy_handle.Instance = ATIM_TIMX_PWMIN;
    g_timx_pwmin_chy_handle.Init.Prescaler = psc;
    g_timx_pwmin_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_timx_pwmin_chy_handle.Init.Period = 0xFFFF;
    HAL_TIM_IC_Init(&g_timx_pwmin_chy_handle);

    /* 配置从模式 */
    slave_config.SlaveMode = TIM_SLAVEMODE_RESET;
    slave_config.InputTrigger = TIM_TS_TI1FP1;
    slave_config.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
    slave_config.TriggerFilter = 0;
    HAL_TIM_SlaveConfigSynchronization(&g_timx_pwmin_chy_handle, &slave_config);
```

```

/* IC1 捕获：上升沿触发 TI1FP1 */
tim_ic_pwmin_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
tim_ic_pwmin_chy.ICSelection = TIM_ICSELECTION_DIRECTTI;
tim_ic_pwmin_chy.ICPrescaler = TIM_ICPSC_DIV1;
tim_ic_pwmin_chy.ICFilter = 0;
HAL_TIM_IC_ConfigChannel(&g_timx_pwmin_chy_handle,
                         &tim_ic_pwmin_chy,
                         TIM_CHANNEL_1);

/* IC2 捕获：下降沿触发 TI1FP2 */
tim_ic_pwmin_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
tim_ic_pwmin_chy.ICSelection = TIM_ICSELECTION_INDIRECTTI;
tim_ic_pwmin_chy.ICPrescaler = TIM_ICPSC_DIV1;
tim_ic_pwmin_chy.ICFilter = 0;
HAL_TIM_IC_ConfigChannel(&g_timx_pwmin_chy_handle,
                         &tim_ic_pwmin_chy,
                         TIM_CHANNEL_2);

/* 使能高级定时器及其相关中断 */
HAL_NVIC_SetPriority(ATIM_TIMX_PWMIN_CC_IRQn, 1, 0);
HAL_NVIC_EnableIRQ(ATIM_TIMX_PWMIN_CC_IRQn);
__HAL_TIM_ENABLE_IT(&g_timx_pwmin_chy_handle, TIM_IT_UPDATE);
HAL_TIM_IC_Start_IT(&g_timx_pwmin_chy_handle, TIM_CHANNEL_1);
HAL_TIM_IC_Start_IT(&g_timx_pwmin_chy_handle, TIM_CHANNEL_2);
__HAL_TIM_ENABLE_IT(&g_timx_pwmin_chy_handle, TIM_IT_CC1);
__HAL_TIM_ENABLE_IT(&g_timx_pwmin_chy_handle, TIM_IT_CC2);
__HAL_TIM_ENABLE(&g_timx_pwmin_chy_handle);
}

```

从上面的代码中可以看出，在配置定时器输入捕获通道时，配置了 IC1 捕获上升沿，IC2 捕获下降沿，并同时开启了捕获比较 1 和 2 的中断，这么一来就可以在中断回调函数中获取通道 1 和通道 2 的捕获比较寄存器值来计算输入 PWM 信号的周期和占空比了。

高级定时器驱动中，TIM8 的中断回调函数，如下所示：

```

/**
 * @brief      高级定时器中断服务函数
 * @param      无
 * @retval     无
 */
void ATIM_TIMX_PWMIN_CC_IRQHandler(void)
{
    if (__HAL_TIM_GET_FLAG(&g_timx_pwmin_chy_handle, TIM_FLAG_CC1) == SET)
    {
        /* 捕获到上升沿 */
        g_timxchy_pwmin_cval = HAL_TIM_ReadCapturedValue(&g_timx_pwmin_chy_handle,
                                                          TIM_CHANNEL_1) + 1;
        g_timxchy_pwmin_sta = 1;

        __HAL_TIM_CLEAR_FLAG(&g_timx_pwmin_chy_handle, TIM_FLAG_CC1);
    }

    if (__HAL_TIM_GET_FLAG(&g_timx_pwmin_chy_handle, TIM_FLAG_CC2) == SET)
    {
        /* 捕获到下降沿 */
        g_timxchy_pwmin_hval = HAL_TIM_ReadCapturedValue(&g_timx_pwmin_chy_handle,
                                                          TIM_CHANNEL_2) + 1;
        __HAL_TIM_CLEAR_FLAG(&g_timx_pwmin_chy_handle, TIM_FLAG_CC2);
    }
}

```

从上面的代码中可以看出，在捕获比较通道 1 中断中获取通道 1 的捕获比较寄存器值就是 TIM8 在输入 PWM 信号的两个上升沿之间的计数值，通过该值可以计算出输入 PWM 信号的周期；在捕获比较通道 2 中断中获取通道 2 的捕获比较寄存器值就是 TIM8 在输入 PWM 信号的上升沿和下降沿之间的计数值，通过该值可以计算出输入 PWM 信号的高电平占空比。

24.2.3 实验应用代码

本章实验的应用代码，如下所示：

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

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    gtim_timx_pwm_chy_init(100-1, 24000-1); /* 初始化高级定时器 PWM 输入模式 */
    atim_timx_pwmin_chy_init(240 - 1); /* 初始化 PWM 输入捕获 */

    while (1)
    {
        if (g_timxchy_pwmin_sta == 1) /* 判断成功捕获标志 */
        {
            g_timxchy_pwmin_sta = 0; /* 清除捕获成功标志 */

            printf("高电平时间: %d us\r\n", g_timxchy_pwmin_hval);
            printf("PWM 周期: %d us\r\n", g_timxchy_pwmin_cval);
            printf("PWM 频率: %d Hz\r\n", 1000000 / g_timxchy_pwmin_cval);
            printf("\r\n");
        }
        t++;

        if (t >= 20)
        {
            t = 0;
            LED0_TOGGLE(); /* LED0 闪烁 */
        }
        delay_ms(10);
    }
}
```

从上面的代码中可以看出，会配置通用定时器输出一个频率为 100Hz、占空比为 50% 的 PWM，该 PWM 信号用于作为本实验中高级定时器的 PWM 输入信号。随后会初始化高级定时器 TIM8，TIM8 的预分频计数器值配置为(240-1)，并且 TIM8 的时钟频率为 240MHz，因此 TIM8 的计数频率为 1MHz，即 1 个计数值对应 1 微秒，方便后面的时间计算。

初始化完成后，就等待高级定时器中断函数中的捕获成功标志为真，捕获成功后，便将捕获到的 PWM 输入信号的周期和占空比输出值串口调试助手。

24.3 下载验证

在完成编译和烧录操作后，将 TIM14 通道 1 输出的 PWM 信号接入 TIM8 的 PWM 输入引脚，即将 PA7 引脚与 PC6 引脚进行短接，随后便可在串口调试助手上看到，捕获到 PWM 输入信号的高电平时间为 5000 微秒、PWM 周期为 10000 微秒，即 PWM 的占空比为 50%，并且也能看到 PWM 输入信号的频率为 100Hz。

第二十五章 OLED 实验

本章将介绍使用 STM32H750 驱动 OLED 模块进行显示。通过本章的学习，读者将学习到使用 GPIO 模拟 8080 时序以及 OLED 模块的驱动。

本章分为以下几个小节：

25.1 硬件设计

25.2 程序设计

25.3 下载验证

25.1 硬件设计

25.1.1 例程功能

1. OLED 上不断刷新显示字符
2. LED0 闪烁，提示程序正在运行

25.1.2 硬件资源

1. LED
LED0 - PE5
2. ATK-MD0096 模块

25.1.3 原理图

本章实验使用了一个正点原子 0.96 寸 OLED 模块，该模块需通过 OLED 模块延长线与板载的 OLED 接口进行连接，该接口也可与摄像头模块进行连接，该接口与板载 MCU 的连接原理图，如下图所示：

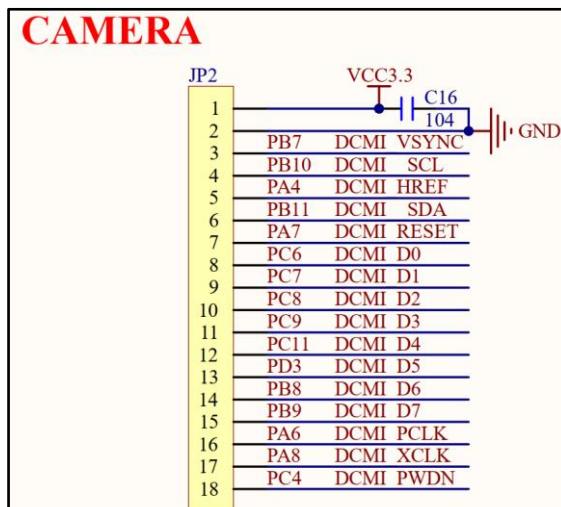


图 25.1.3.1 OLED 模块与 MCU 的连接原理图

25.2 程序设计

25.2.1 HAL 库的 GPIO 驱动

本章实验通过控制与 OLED 模块连接的 GPIO 模拟 8080 时序，实现配置 OLED 模块并在 OLED 模块上显示指定字符，对于 GPIO，主要涉及 GPIO 的配置和写操作，需要以下几个步骤：

- ①：配置 GPIO 引脚为通用输出模式

②：在与 OLED 模块通讯时，根据需求控制指定 GPIO 引脚输出指定电平

在 HAL 库中对应的驱动函数如下：

①：配置 GPIO 引脚

请见第 10.2.1 小节中配置 GPIO 引脚的相关内容。

②：设置 GPIO 引脚输出电平

使用 HAL 库提供的驱动函数设置 GPIO 引脚输出电平的方式，请见第 10.2.1 小节中设置 GPIO 引脚输出电平的相关内容。虽然 HAL 库已经提供了完整的 GPIO 驱动函数，但在大多数场景下，调用 HAL 库中 GPIO 驱动函数的效率并没有直接操作 GPIO 寄存器的方式高，例如，本章实验中将以直接操作寄存器的方式写与 OLED 模块通讯时模拟的 8080 时序通讯的数据总线，具体的方式是操作 GPIO 端口的 GPIO 端口输出数据寄存器 (GPIOx_ODR)，其使用示例，如下所示：

```
#include "stm32h7xx.h"

void example_fun(void)
{
    /* 设置 PA0 输出高电平 */
    GPIOA->ODR |= (1 << 0);

    /* 设置 PB1 输出低电平 */
    GPIOB->ODR &= ~(1 << 1);
}
```

25.2.2 OLED 驱动

本章介绍使用 GPIO 模拟 8080 时序驱动 OLED 模块，本质就是控制 GPIO 输出高低电平。OLED 驱动主要负责向应用层提供 OLED 的初始化函数，和各种 OLED 显示的操作函数。本章实验中，OLED 的驱动代码包括 oled.c、oled.h 和字体文件 oledfont.h 三个文件。

OLED 驱动中，对 GPIO 相关的宏定义，如下所示：

```
#define OLED_SPI_RST_PORT          GPIOA
#define OLED_SPI_RST_PIN            GPIO_PIN_7
#define OLED_SPI_RST_CLK_ENABLE()   __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define OLED_SPI_CS_PORT           GPIOB
#define OLED_SPI_CS_PIN             GPIO_PIN_7
#define OLED_SPI_CS_CLK_ENABLE()   __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define OLED_SPI_RS_PORT           GPIOB
#define OLED_SPI_RS_PIN             GPIO_PIN_10
#define OLED_SPI_RS_CLK_ENABLE()   __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define OLED_SPI_SCLK_PORT         GPIOC
#define OLED_SPI_SCLK_PIN           GPIO_PIN_6
#define OLED_SPI_SCLK_CLK_ENABLE()  __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define OLED_SPI_SDIN_PORT         GPIOC
#define OLED_SPI_SDIN_PIN           GPIO_PIN_7
#define OLED_SPI_SDIN_CLK_ENABLE()  __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define OLED_RST(x) do { (x) ? \
    HAL_GPIO_WritePin(OLED_SPI_RST_PORT, OLED_SPI_RST_PIN, GPIO_PIN_SET): \
    HAL_GPIO_WritePin(OLED_SPI_RST_PORT, OLED_SPI_RST_PIN, GPIO_PIN_RESET); \
} while (0)

#define OLED_CS(x) do { (x) ? \
    HAL_GPIO_WritePin(OLED_SPI_CS_PORT, OLED_SPI_CS_PIN, GPIO_PIN_SET): \
    HAL_GPIO_WritePin(OLED_SPI_CS_PORT, OLED_SPI_CS_PIN, GPIO_PIN_RESET); \
} while (0)
```

```

} while (0)

#define OLED_RS(x) do { (x) ? \
    HAL_GPIO_WritePin(OLED_SPI_RS_PORT, OLED_SPI_RS_PIN, GPIO_PIN_SET): \
    HAL_GPIO_WritePin(OLED_SPI_RS_PORT, OLED_SPI_RS_PIN, GPIO_PIN_RESET); \
} while (0)

#define OLED_WR(x) do { (x) ? \
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_SET): \
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_RESET); \
} while (0)

#define OLED_RD(x) do { (x) ? \
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_11, GPIO_PIN_SET): \
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_11, GPIO_PIN_RESET); \
} while (0)

```

OLED 驱动中，OLED 的初始化函数，如下所示：

```

/** 
 * @brief  初始化 OLED
 * @param  无
 * @retval 无
 */
void oled_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();

#if OLED_MODE==1           /* 使用 8080 并口模式 */

    gpio_init_struct.Pin = GPIO_PIN_4 | \
                          GPIO_PIN_6 | \
                          GPIO_PIN_7 | \
                          GPIO_PIN_8;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    gpio_init_struct.Pin = GPIO_PIN_7 | \
                          GPIO_PIN_8 | \
                          GPIO_PIN_9 | \
                          GPIO_PIN_10 | \
                          GPIO_PIN_11;
    HAL_GPIO_Init(GPIOB, &gpio_init_struct);

    gpio_init_struct.Pin = GPIO_PIN_4 | \
                          GPIO_PIN_6 | \
                          GPIO_PIN_7 | \
                          GPIO_PIN_8 | \
                          GPIO_PIN_9 | \
                          GPIO_PIN_11;
    HAL_GPIO_Init(GPIOC, &gpio_init_struct);

    /* PD3 设置 */
    gpio_init_struct.Pin=GPIO_PIN_3;
    HAL_GPIO_Init(GPIOD,&gpio_init_struct);

    OLED_WR(1);
    OLED_RD(1);

```

```

#else /* 使用 4 线 SPI 串口模式 */

    gpio_init_struct.Pin=OLED_SPI_RST_PIN;
    gpio_init_struct.Mode=GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull=GPIO_PULLUP;
    gpio_init_struct.Speed=GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(OLED_SPI_RST_PORT,&gpio_init_struct);

    gpio_init_struct.Pin=OLED_SPI_CS_PIN;
    HAL_GPIO_Init(OLED_SPI_CS_PORT,&gpio_init_struct);

    gpio_init_struct.Pin=OLED_SPI_RS_PIN;
    HAL_GPIO_Init(OLED_SPI_RS_PORT,&gpio_init_struct);

    gpio_init_struct.Pin=OLED_SPI_SCLK_PIN;
    HAL_GPIO_Init(OLED_SPI_SCLK_PORT,&gpio_init_struct);

    gpio_init_struct.Pin=OLED_SPI_SDIN_PIN;
    HAL_GPIO_Init(OLED_SPI_SDIN_PORT,&gpio_init_struct);

    OLED_SDIN(1);
    OLED_SCLK(1);
#endif
    OLED_CS(1);
    OLED_RS(1);

    OLED_RST(0);
    delay_ms(100);
    OLED_RST(1);

    oled_wr_byte(0xAE, OLED_CMD);
    oled_wr_byte(0xD5, OLED_CMD);
    oled_wr_byte(80, OLED_CMD);
    oled_wr_byte(0xA8, OLED_CMD);
    oled_wr_byte(0x3F, OLED_CMD);
    oled_wr_byte(0xD3, OLED_CMD);
    oled_wr_byte(0x00, OLED_CMD);
    oled_wr_byte(0x40, OLED_CMD);
    oled_wr_byte(0x8D, OLED_CMD);
    oled_wr_byte(0x14, OLED_CMD);
    oled_wr_byte(0x20, OLED_CMD);
    oled_wr_byte(0x02, OLED_CMD);
    oled_wr_byte(0xA1, OLED_CMD);
    oled_wr_byte(0xC8, OLED_CMD);
    oled_wr_byte(0xDA, OLED_CMD);
    oled_wr_byte(0x12, OLED_CMD);
    oled_wr_byte(0x81, OLED_CMD);
    oled_wr_byte(0xEF, OLED_CMD);
    oled_wr_byte(0xD9, OLED_CMD);
    oled_wr_byte(0xF1, OLED_CMD);
    oled_wr_byte(0xDB, OLED_CMD);
    oled_wr_byte(0x30, OLED_CMD);
    oled_wr_byte(0xA4, OLED_CMD);
    oled_wr_byte(0xA6, OLED_CMD);
    oled_wr_byte(0xAF, OLED_CMD);
    oled_clear();
}

```

从上面的代码中可以看出，OLED 的初始化函数，就是配置与 OLED 模块连接的 GPIO 引脚为输出模式，同时根据通信协议控制 GPIO 输出高低电平，以完成配置 OLED 模块的操作。

OLED 驱动中，向 OLED 写一字节数据的函数如下所示：

```

/**
 * @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);
}

```

从上面的代码中可以看出，向 OLED 写一个字节就是按照 8080 时序来控制 GPIO 完成的，其中操作数据总线的函数 oled_data_out() 是通过直接操作 GPIO 寄存器完成的，其函数如下所示：

```

/** @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)
{
    uint16_t dat = data & 0XF;
    GPIOC->ODR &= ~(0XF << 6);           /* 清空 6~9 */
    GPIOC->ODR |= dat << 6;                /* D[3:0]-->PC[9:6] */

    GPIOC->ODR &= ~(0X1 << 11);          /* 清空 11 */
    GPIOC->ODR |= ((data >> 4) & 0x01) << 11; /* D4 */

    GPIOD->ODR &= ~(0X1 << 3);          /* 清空 3 */
    GPIOD->ODR |= ((data >> 5) & 0x01) << 3; /* D5 */

    GPIOB->ODR &= ~(0X3<<8);           /* 清空 8,9 */
    GPIOB->ODR |= ((data >> 6) & 0x01) << 8; /* D6 */
    GPIOB->ODR |= ((data >> 7) & 0x01) << 9; /* D7 */
}

```

通过上面介绍的驱动函数就能够往 OLED 模块写入命令或数据了，而在 OLED 模块的显示屏上显示出特定的字符或图案，都是通过 OLED 模块规定的特定命令来完成的，想深究的读者可以查看正点原子 ATK-MD0096 模块的用户手册或查看实际使用的 OLED 模块相关的文档。

25.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t t = 0;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480MHz */
}

```

```
delay_init(480);                                /* 初始化延时功能 */
uart_init(115200);                             /* 初始化串口 */
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, 40, "ATOM 2020/3/22", 12);
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 闪烁 */
    LED0_TOGGLE();
}
}
```

从上面的代码中可以看出，在初始化完 OLED 后，便使用函数 `oled_show_string()` 将一些字符串写入到本地的显存中，然后使用函数 `oled_refresh_gram()` 将本地显存中的数据刷新到 OLED 上进行显示，最后在 while 循环中遍历 ASCII 表中的部分字符，将其轮流显示到 OLED 上，同时显示字符对应的 ASCII 码值。

25.3 下载验证

在完成编译和烧录操作后，先将开发板断电，随后将 OLED 模块通过 OLED 延长线与开发板进行连接，最后再给开发板供电（在插拔开发板上的接插件和模块时，要求必须断电操作，否则容易烧毁硬件）。程序运行后，可以看到 OLED 上显示了预期的字符串，同时不断地遍历显示 ASCII 表中的部分字符，即字符对应的 ASCII 码值。

第二十六章 内存保护（MPU）实验

STM32 的 Cortex M4 (STM32F3/F4 系列) 和 Cortex M7 (STM32H7/H7 系列) 系列的产品，都带有内存保护单元 (memory protection unit)，简称：MPU。使用 MPU 可以设置不同存储区域的存储器访问特性 (如只支持特权访问或全访问) 和存储器属性 (如可缓存、可共享)，从而提高嵌入式系统的健壮性，使系统更加安全。接下来，我们将以 STM32H750 为例，给大家介绍 STM32H7 内存保护单元 (MPU) 的使用。

本章分为以下几个小节：

26.1 硬件设计

26.2 程序设计

26.3 下载验证

26.1 硬件设计

26.1.1 例程功能

本实验使用 STM32H7 自带的 MPU 功能，对一个特定的内存空间(数组，地址:0X20002000) 进行写访问保护。开机时，串口调试助手显示：MPU closed，表示默认是没有写保护的。按 KEY0 可以往数组里面写数据，按 KEY1，可以读取数组里面的数据。按 KEY_UP 则开启 MPU 保护，此时，如果再按 KEY0 往数组写数据，就会引起 MemManage 错误，进入 MemManage_Handler 中断服务函数，此时 LED1 点亮，同时打印错误信息，最后软件复位，系统重启。LED0 用于提示程序正在运行，所有信息都是通过串口 1 输出，需要用串口调试助手查看。

26.1.2 硬件资源

1. LED

LED0 - PE5

LED1 - PE6

2) 独立按键

KEY1 - PA15

WK_UP - PA0

3) 串口 1

26.1.3 原理图

MPU 属于 STM32H750 的内部资源，只需要软件设置好即可正常工作。我们借助按键和 LED 灯验证 MPU 工作是否正常，然后通过电脑串口上位机软件观察打印出来的信息。

26.2 程序设计

26.2.1 MPU 的 HAL 库驱动

MPU 在 HAL 库中的驱动代码在 stm32h7xx_hal_cortex.c 文件 (及其头文件) 中。

①: HAL_MPU_ConfigRegion 函数

MPU 的初始化函数，其声明如下：

```
void HAL_MPU_ConfigRegion(MPU_Region_InitTypeDef *MPU_Init);
```

该函数的形参描述，如下表所示：

形参	描述
MPU_Init	指向 MPU_Region_InitTypeDef 结构的指针

表 26.2.1.1 函数 HAL_MPU_ConfigRegion ()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 26.2.1.2 函数 HAL_MPU_ConfigRegion ()返回值描述

该形参是 MPU_Region_InitTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
    uint8_t           Enable;          /* 区域使能/禁止 */
    uint8_t           Number;          /* 区域编号 */
    uint32_t          BaseAddress;     /* 配置区域地址 */
    uint8_t           Size;            /* 区域容量 */
    uint8_t           SubRegionDisable; /* 子 region 使能位段设置 */
    uint8_t           TypeExtField;   /* 类型扩展级别 */
    uint8_t           AccessPermission; /* 设置访问权限 */
    uint8_t           DisableExec;    /* 允许/禁止执行 */
    uint8_t           IsShareable;    /* 禁止/允许共享 */
    uint8_t           IsCacheable;    /* 禁止/允许缓存 */
    uint8_t           IsBufferable;   /* 禁止/允许缓冲 */
}MPU_Region_InitTypeDef;
```

该结构体成员变量很多，每个成员变量的含义我们都在上面定义中有注释。这里大家注意，除了 BaseAddress 和 Number 两个成员变量是分别用来配置 MPU->RBAR 和 MPU->RNR 寄存器之外，其他成员变量都是用来配置 MPU->RASR 寄存器相关位，大家如果对这些配置项不理解的话，可以直接对照我们前面讲解的寄存器 MPU->RASR 各个位含义来理解。

②: HAL_MPU_Enable 函数

HAL_MPU_Enable 函数是 MPU 使能函数。其声明如下：

```
void HAL_MPU_Enable(uint32_t MPU_Control);
```

该函数的形参描述，如下表所示：

形参	描述
MPU_Control	指定 MPU 在硬故障期间的控制模式

表 26.2.1.3 函数 HAL_MPU_ConfigRegion ()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 26.2.1.4 函数 HAL_MPU_ConfigRegion ()返回值描述

③: HAL_MPU_Disable 函数

HAL_MPU_Disable 函数是 MPU 失能函数。其声明如下：

```
void HAL_MPU_Disable (void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 26.2.1.5 函数 HAL_MPU_ConfigRegion ()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 26.2.1.6 函数 HAL_MPU_ConfigRegion ()返回值描述

26.2.2 MPU 配置步骤

①: 禁止 MPU 以及 MemManage 中断

调用 HAL_MPU_Disable 函数禁止 MPU 以及 MemManage 中断。

②: 配置某个区域的 MPU 保护参数

在进行 MPU 配置之前我们必须先通过 MPU_RNR 区域编号寄存器来选择下一个要配置的区域，然后通过配置 MPU_RBAR 基址寄存器来配置基址，最后通过区域属性和容量寄存器 RASR 来配置区域相关属性和参数。这些过程都可以通过调用 HAL_MPU_ConfigRegion 函数

来完成。除了 BaseAddress 和 Number 两个成员变量是分别用来配置 MPU->RBAR 和 MPU->RNR 寄存器之外，其他成员变量都是用来配置 MPU->RASR 寄存器相关位，大家如果对这些配置项不理解的话，可以直接对照我们前面讲解的寄存器 MPU->RASR 各个位含义来理解。

③：使能 MPU 以及 MemManage 中断

调用 HAL_MPU_Enable 函数禁止 MPU 以及 MemManage 中断。

26.2.3 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。MPU 驱动源码包括两个文件：mpu.c 和 mpu.h。

mpu.h 头文件只有函数声明，就不看了，所以我们直接看 mpu.c 文件，先看设置某个区域的 MPU 保护函数：

```
/***
 * @brief      设置某个区域的 MPU 保护
 * @param      baseaddr: MPU 保护区域的基址(首地址)
 * @param      size:MPU 保护区域的大小(必须是 32 的倍数,单位为字节)
 * @param      rnum:MPU 保护区编号,范围:0~7,最大支持 8 个保护区域
 * @param      de:禁止指令访问;0,允许指令访问;1,禁止指令访问
 * @param      ap:访问权限,访问关系如下:
 *             0,无访问(特权&用户都不可访问)
 *             1,仅支持特权读写访问
 *             2,禁止用户写访问(特权可读写访问)
 *             3,全访问(特权&用户都可访问)
 *             4,无法预测(禁止设置为 4!!!)
 *             5,仅支持特权读访问
 *             6,只读(特权&用户都不可以写)
 * @note       详见:STM32H7 编程手册.pdf,4.6.6 节,Table 91.
 * @param      sen:是否允许共用;0,不允许;1,允许
 * @param      cen:是否允许 cache;0,不允许;1,允许
 * @param      ben:是否允许缓冲;0,不允许;1,允许
 * @retval     0, 成功; 1, 错误;
 */
uint8_t mpu_set_protection(uint32_t baseaddr, uint32_t size, uint32_t rnum,
                           uint8_t de, uint8_t ap, uint8_t sen, uint8_t cen, uint8_t ben)
{
    MPU_Region_InitTypeDef mpu_region_init_handle; /* MPU 初始化句柄 */
    HAL_MPU_Disable(); /* 配置 MPU 之前先关闭 MPU,配置完成以后在使能 MPU */

    mpu_region_init_handle.Enable = MPU_REGION_ENABLE; /* 使能该保护区域 */
    mpu_region_init_handle.Number = rnum; /* 设置保护区域 */
    mpu_region_init_handle.BaseAddress = baseaddr; /* 设置基址 */
    mpu_region_init_handle.Size = size; /* 设置保护区域大小 */
    mpu_region_init_handle.SubRegionDisable = 0X00; /* 禁止子区域 */
    mpu_region_init_handle.TypeExtField = MPU_TEX_LEVEL0; /* 设置类型扩展域为 level0 */
    mpu_region_init_handle.AccessPermission = ap; /* 设置访问权限 */
    mpu_region_init_handle.DisableExec = de; /* 是否允许指令访问 */
    mpu_region_init_handle.IsShareable = sen; /* 是否允许共用 */
    mpu_region_init_handle.IsCacheable = cen; /* 是否允许 cache */
    mpu_region_init_handle.IsBufferable = ben; /* 是否允许缓冲 */
    HAL_MPU_ConfigRegion(&mpu_region_init_handle); /* 配置 MPU */
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT); /* 开启 MPU */
    return 0;
}
```

该函数使用 HAL_MPU_ConfigRegion 函数对 MPU_Region_InitTypeDef 结构体变量进行配置。注意：配置之前我们要调用 HAL_MPU_Disable 函数禁止 MPU 才能进行配置，最后再调用 HAL_MPU_Enable 函数开启 MPU。

接着，要介绍的是 mpu_memory_protection 函数，其定义如下：

```

/***
 * @brief      设置需要保护的存储块
 * @note       必须对部分存储区域进行 MPU 保护, 否则可能导致程序运行异常
 *           比如 MCU 屏不显示, 摄像头采集数据出错等等问题
 * @param      无
 * @retval     nbytes 以 2 为底的指数值
 */
void mpu_memory_protection(void)
{
    /* 保护整个 DTCM, 共 128K 字节, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲 */
    mpu_set_protection(0x20000000, MPU_REGION_SIZE_128KB, MPU_REGION_NUMBER1,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_CACHEABLE, MPU_ACCESS_BUFFERABLE);

    /* 保护整个 AXI SRAM, 共 512K 字节, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲 */
    mpu_set_protection(0x24000000, MPU_REGION_SIZE_512KB, MPU_REGION_NUMBER2,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_CACHEABLE, MPU_ACCESS_BUFFERABLE);

    /* 保护整个 SRAM1~SRAM3, 共 288K 字节, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲 */
    mpu_set_protection(0x30000000, MPU_REGION_SIZE_512KB, MPU_REGION_NUMBER3,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_CACHEABLE, MPU_ACCESS_BUFFERABLE);

    /* 保护整个 SRAM4, 共 64K 字节, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲 */
    mpu_set_protection(0x38000000, MPU_REGION_SIZE_64KB, MPU_REGION_NUMBER4,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_CACHEABLE, MPU_ACCESS_BUFFERABLE);

    /* 保护 MCU LCD 屏所在的 FMC 区域,, 共 64M 字节, 允许指令访问, 禁止共用, 禁止 cache, 禁止缓冲 */
    mpu_set_protection(0x60000000, MPU_REGION_SIZE_64MB, MPU_REGION_NUMBER5,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_NOT_CACHEABLE,
                        MPU_ACCESS_NOT_BUFFERABLE);

    /* 保护 SDRAM 区域, 共 64M 字节, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲 */
    mpu_set_protection(0XC0000000, MPU_REGION_SIZE_64MB, MPU_REGION_NUMBER6,
                        MPU_INSTRUCTION_ACCESS_ENABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_CACHEABLE, MPU_ACCESS_BUFFERABLE);

    /* 保护整个 NAND FLASH 区域, 共 256M 字节, 禁止指令访问, 禁止共用, 禁止 cache, 禁止缓冲 */
    mpu_set_protection(0X80000000, MPU_REGION_SIZE_256MB, MPU_REGION_NUMBER7,
                        MPU_INSTRUCTION_ACCESS_DISABLE, MPU_REGION_FULL_ACCESS,
                        MPU_ACCESS_NOT_SHAREABLE, MPU_ACCESS_NOT_CACHEABLE,
                        MPU_ACCESS_NOT_BUFFERABLE);
}

```

mpu_memory_protection 函数，用于设置整个代码里面，我们需要保护的存储块，这里我们对 7 个存储块（使用了 7 个区域（region））进行了保护：

1, 从 0x20000000 地址开始的 128KB DTCM 地址空间, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲。

2, 从 0x24000000 地址开始的 512KB AXI SRAM 地址空间, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲。

3, 从 0x30000000 地址开始的 288KB SRAM1~SRAM3 地址空间, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲。

4, 从 0x38000000 地址开始的 64KB SRAM4 地址空间, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲。

5, 从 0x60000000 地址开始的 64MB 地址空间, 允许指令访问, 禁止共用, 禁止 cache, 禁止缓冲, 保护 MCU LCD 屏的访问地址取件, 如不进行设置, 可能导致 MCU LCD 白屏。

6, 从 0XC0000000 地址开始的 64MB 地址空间, 即 SDRAM 的地址范围, 允许指令访问, 禁止共用, 允许 cache, 允许缓冲。

7，从 0X80000000 地址开始的 256MB 地址空间，即 NAND FLASH 区域，禁止指令访问，禁止共用，禁止 cache，禁止缓冲，如不进行设置，可能导致 NAND FLASH 访问异常。

这七个地址空间的保护设置，可以提高代码的稳定性（其实就是减少使用 cache 导致的各种莫名其妙的问题），请大家不要随意改动。此函数在本例程没有用到，不过我们在后续代码都会用到。

最后，MemManage_Handler 函数，用于处理产生 MemManage 错误的中断服务函数，在该函数里面点亮了 LED1，并输出一些串口信息，对系统进行软复位，以便观察本例程的实验结果。函数定义如下：

```
/***
 * @brief      MemManage 错误处理中断
 * @note       进入此中断以后,将无法恢复程序运行!!
 * @param      无
 * @retval     nbytes 以 2 为底的指数值
 */
void MemManage_Handler(void)
{
    LED1(0);                                /* 点亮 LED1(GREEN LED) */
    printf("Mem Access Error!!\r\n");        /* 输出错误信息 */
    delay_ms(1000);
    printf("Soft Resetting...\r\n");          /* 提示软件重启 */
    delay_ms(1000);
    NVIC_SystemReset();                      /* 软复位 */
}
```

mpu.c 的内容介绍就到此，下面开始 main.c 文件的介绍。

26.2.4 实验应用代码

在 main.c 文件代码如下：

```
uint8_t mpudata[128] __attribute__((at(0X20002000))); /* 定义一个数组 */

int main(void)
{
    uint8_t key = 0;
    uint8_t t = 0;
    sys_cache_enable();                         /* 打开 L1-Cache */
    HAL_Init();                                /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);        /* 设置时钟, 480Mhz */
    delay_init(480);                           /* 延时初始化 */
    usart_init(115200);                        /* 串口初始化为 115200 */
    led_init();                                /* 初始化 LED */
    key_init();                                /* 初始化按键 */
    printf("\r\n\r\nMPU closed!\r\n");

    while (1)
    {
        key = key_scan(0);

        if (key == WKUP_PRES)                  /* 使能 MPU 保护数组 mpudata */
        {
            mpu_set_protection(0X20002000, MPU_REGION_SIZE_128B,
                MPU_REGION_NUMBER0, MPU_INSTRUCTION_ACCESS_ENABLE,
                MPU_REGION_PRIV_RO_URO, MPU_ACCESS_NOT_SHAREABLE,
                MPU_ACCESS_NOT_CACHEABLE,
                MPU_ACCESS_BUFFERABLE);           /* 只读, 禁止共用, 禁止 catch, 允许缓冲 */
            printf("MPU open!\r\n");
        }
        /* 向数组中写入数据, 如果开启了 MPU 保护的话会进入内存访问错误! */
        else if (key == KEY0_PRES)
        {
            printf("Start Writing data...\r\n");
        }
    }
}
```

```

        sprintf((char *)mpudata, "MPU test array %d", t);
        printf("Data Write finshed!\r\n");
    }

/* 从数组中读取数据，不管有没有开启 MPU 保护都不会进入内存访问错误！ */
else if (key == KEY1_PRES)
{
    printf("Array data is:%s\r\n", mpudata);
}
else
{
    delay_ms(10);
}

t++;

if ((t % 50) == 0) LED0_TOGGLE(); /* LED0 取反 */
}
}

```

在 main 函数前面，我们定义了一个 128 字节大小的数组：mpudata，其首地址为 0X20002000，默认情况下，MPU 保护关闭，可以对该数组进行读写访问。当我们按下 KEY_UP 按键的时候，通过 mpu_set_protection 函数，对其 0X20002000 为起始地址，大小为 128 字节的内存空间进行保护，仅支持特权读访问，此时如果再按 KEY0，对数组进行写入操作，则会引起 MemManage 访问异常，进入 MemManage_Handler 中断服务函数，执行相关操作。

其他的代码比较简单，这里就不多做说明了，在整个代码编译通过之后，我们就可以开始下载验证了。

26.3 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。然后，打开串口调试助手（XCOM V2.6），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 COM8，另外，请注意：波特率是 115200），可以看到如图 26.3.1 所示信息（如果没有提示信息，请先按复位）：



图 26.3.1 复位后串口调试助手收到的信息

从图 26.3.1 可以看出，此时串口助手提示：MPU Closed，即 MPU 保护是关闭的，我们可以按 KEY0 往数组里面写入数据，按 KEY1，可以读取刚刚写入的数据，按 KEY_UP，则开启 MPU 保护，提示：MPU open!，此时，如果再按 KEY0，往数组里面写数据的话，则会引起 MemManage 访问异常，进入 MemManage_Handler 中断服务函数，点亮 LED1，并提示：Mem Access Error!!，并在 1 秒钟以后，重启系统（软复位），如图 26.3.2 所示：



图 26.3.2 串口调试助手显示运行结果

整个过程，验证了我们代码的正确性，通过 MPU 实现了对特定内存的写保护功能。通过 MPU，我们可以提高系统的可靠性，使代码更加安全的运行。

第二十七章 TFTLCD（MCU 屏）实验

本章将介绍使用 STM32H750 驱动 TFTLCD（MCU 屏）进行显示。通过本章的学习，读者将学习到 FSMC 的使用。

本章分为以下几个小节：

27.1 硬件设计

27.2 程序设计

27.3 下载验证

27.1 硬件设计

27.1.1 例程功能

1. TFTLCD 上显示实验信息，并不断刷新底色
2. LED0 闪烁，提示程序正在运行

27.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块

27.1.3 原理图

本章实验使用了正点原子的 TFTLCD 模块（兼容正点原子 2.8/3.5/4.3/7/10 寸的 TFTLCD 模块），该模块需通过 LCD 转接板与板载的 TFTLCD 接口进行连接，该接口与板载 MCU 的连接原理图，如下图所示：

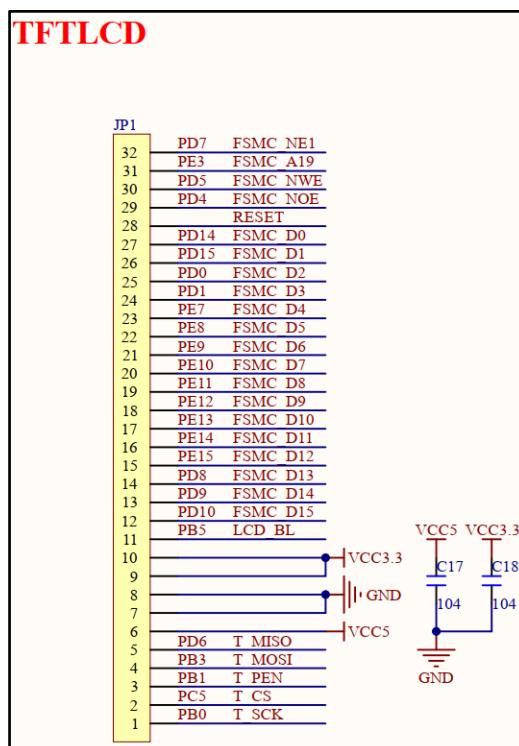


图 27.1.3.1 TFTLCD 模块与 MCU 的连接原理图

从上图中可以看到，TFTLCD 模块连接至 FSMC 的存储块 1 的区域块 1，并且使用了地址线 A19 作为 TFTLCD 模块命令或数据的选择信号，可计算出配置好 FSMC 后，TFTLCD 模块的命令访问地址被映射到 0x600FFFFE，TFTLCD 模块的数据访问地址被映射到 0x60100000。

27.2 程序设计

27.2.1 HAL 库的 FSMC 驱动

本章实验通过 FSMC 驱动 8080 并口的 TFTLCD 模块，通过 FSMC 可以将 TFTLCD 模块的命令和数据寄存器映射为两个地址，往这两个地址写入或读取数据就可直接与 TFTLCD 模块进行通讯，因此需要对 FSMC 做相应的配置，具体的步骤如下：

①：配置 FSMC 存储块 1 的区域块 1

在 HAL 库中对应的驱动函数如下：

①：配置 FSMC 存储块 1 的区域块

该函数用于配置 FSMC 存储块 1 的区域块，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_SRAM_Init(      SRAM_HandleTypeDef *hsram,
                                      FMC_NORSRAM_TimingTypeDef *Timing,
                                      FMC_NORSRAM_TimingTypeDef *ExtTiming);
```

该函数的形参描述，如下表所示：

形参	描述
hsram	指向 SRAM 句柄的指针
Timing	指向 SRAM 读时序配置结构体的指针 需自行定义，并根据 SRAM 读时序配置参数填充结构体中的成员变量
ExtTiming	指向 SRAM 写时序配置结构体的指针 需自行定义，并根据 SRAM 写时序配置参数填充结构体中的成员变量

表 27.2.1.1 函数 HAL_SRAM_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 27.2.1.2 函数 HAL_SRAM_Init()返回值描述

该函数使用 FMC_NORSRAM_TimingTypeDef 类型的结构体变量传入 FSMC 存储块 1 区域块的读写时序配置参数，该结构体的定义如下所示：

```
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; /* 总线转换时间 */
    uint32_t CLKDivision;          /* 时钟分频系数 */
    uint32_t DataLatency;          /* 数据延时 */
    uint32_t AccessMode;           /* 访问模式 */
} FMC_NORSRAM_TimingTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    SRAM_HandleTypeDef sram_handle = {0};
    FMC_NORSRAM_TimingTypeDef read_timing = {0};
    FMC_NORSRAM_TimingTypeDef write_timing = {0};

    /* 填充读数据结构体 */
    read_timing.AddressSetupTime = 0xFF;
```

```

read_timing.AddressHoldTime = 0xFF;
read_timing.DataSetupTime = 0xFF;
read_timing.AccessMode = FSMC_ACCESS_MODE_A;

/* 填充写时序结构体 */
write_timing.AddressSetupTime = 0xFF;
write_timing.AddressHoldTime = 0xFF;
write_timing.DataSetupTime = 0xFF;
write_timing.AccessMode = FSMC_ACCESS_MODE_A;

/* 配置 FSMC 存储块 1 的区域块 */
sram_handle.Instance = FSMC_NORSRAM_DEVICE;
sram_handle.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;
sram_handle.Init.NSBank = FSMC_NORSRAM_BANK1;
sram_handle.Init.DataAddressMux = FSMC_DATA_ADDRESS_MUX_DISABLE;
sram_handle.Init.MemoryType = FSMC_MEMORY_TYPE_SRAM;
sram_handle.Init.MemoryDataWidth = FSMC_NORSRAM_MEM_BUS_WIDTH_16;
sram_handle.Init.BurstAccessMode = FSMC_BURST_ACCESS_MODE_DISABLE;
sram_handle.Init.WaitSignalPolarity = FSMC_WAIT_SIGNAL_POLARITY_LOW;
sram_handle.Init.WrapMode = FSMC_WRAP_MODE_DISABLE;
sram_handle.Init.WaitSignalActive = FSMC_WAIT_TIMING_BEFORE_WS;
sram_handle.Init.WriteOperation = FSMC_WRITE_OPERATION_ENABLE;
sram_handle.Init.WaitSignal = FSMC_WAIT_SIGNAL_DISABLE;
sram_handle.Init.ExtendedMode = FSMC_EXTENDED_MODE_ENABLE;
sram_handle.Init.AynchronousWait = FSMC_ASYNCHRONOUS_WAIT_DISABLE;
sram_handle.Init.WriteBurst = FSMC_WRITE_BURST_DISABLE;
sram_handle.Init.PageSize = FSMC_PAGE_SIZE_NONE;
HAL_SRAM_Init(&sram_handle, &read_timing, &write_timing);
}

```

27.2.2 TFTLCD 驱动

本章实验的 TFTLCD 驱动主要负责向应用层提供 TFTLCD 的初始化和各种 TFTLCD 显示的操作函数。本章实验中，TFTLCD 的驱动代码包括 lcd.c、lcd_ex.c、lcd.h 和字体文件 lcdfont.h 四个文件。

由于 TFTLCD 模块需要使用到大量的 GPIO 引脚，因此对于 GPIO 的相关定义，请读者自行查看 lcd.c 和 lcd.h 这两个文件。

TFTLCD 驱动中，TFTLCD 的初始化函数，如下所示：

```

/**
 * @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;
    FMC_NORSRAM_TimingTypeDef fmc_read_handle;
    FMC_NORSRAM_TimingTypeDef fmc_write_handle;

    LCD_CS_GPIO_CLK_ENABLE();
    LCD_WR_GPIO_CLK_ENABLE();
    LCD_RD_GPIO_CLK_ENABLE();
    LCD_RS_GPIO_CLK_ENABLE();
    LCD_BL_GPIO_CLK_ENABLE();

    gpio_init_struct.Pin = LCD_CS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF12_FMC;
    HAL_GPIO_Init(LCD_CS_GPIO_PORT, &gpio_init_struct);
}

```

```
gpio_init_struct.Pin = LCD_WR_GPIO_PIN;
HAL_GPIO_Init(LCD_WR_GPIO_PORT, &gpio_init_struct);

gpio_init_struct.Pin = LCD_RD_GPIO_PIN;
HAL_GPIO_Init(LCD_RD_GPIO_PORT, &gpio_init_struct);

gpio_init_struct.Pin = LCD_RS_GPIO_PIN;
HAL_GPIO_Init(LCD_RS_GPIO_PORT, &gpio_init_struct);

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);

g_sram_handle.Instance = FMC_NORSRAM_DEVICE;
g_sram_handle.Extended = FMC_NORSRAM_EXTENDED_DEVICE;

g_sram_handle.Init.NSBank = FMC_NORSRAM_BANK1;
g_sram_handle.Init.DataAddressMux = FMC_DATA_ADDRESS_MUX_DISABLE;
g_sram_handle.Init.MemoryType = FMC_MEMORY_TYPE_SRAM;
g_sram_handle.Init.MemoryDataWidth = FMC_NORSRAM_MEM_BUS_WIDTH_16;
g_sram_handle.Init.BurstAccessMode = FMC_BURST_ACCESS_MODE_DISABLE;
g_sram_handle.Init.WaitSignalPolarity = FMC_WAIT_SIGNAL_POLARITY_LOW;
g_sram_handle.Init.WaitSignalActive = FMC_WAIT_TIMING_BEFORE_WS;
g_sram_handle.Init.WriteOperation = FMC_WRITE_OPERATION_ENABLE;
g_sram_handle.Init.WaitSignal = FMC_WAIT_SIGNAL_DISABLE;
g_sram_handle.Init.ExtendedMode = FMC_EXTENDED_MODE_ENABLE;
g_sram_handle.Init.AsynchronousWait = FMC_ASYNCHRONOUS_WAIT_DISABLE;
g_sram_handle.Init.WriteBurst = FMC_WRITE_BURST_DISABLE;
g_sram_handle.Init.ContinuousClock = FMC_CONTINUOUS_CLOCK_SYNC_ASYNC;

/* FMC 读时序控制寄存器 */
fmc_read_handle.AddressSetupTime = 0x15;
fmc_read_handle.AddressHoldTime = 0x00;
fmc_read_handle.DataSetupTime = 0x78;

fmc_read_handle.AccessMode = FMC_ACCESS_MODE_A;
/* FMC 写时序控制寄存器 */
fmc_write_handle.AddressSetupTime = 0x15;
fmc_write_handle.AddressHoldTime = 0x00;
fmc_write_handle.DataSetupTime = 0x15;

fmc_write_handle.AccessMode = FMC_ACCESS_MODE_A;

HAL_SRAM_Init(&g_sram_handle, &fmc_read_handle, &fmc_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 0X93 */
lcddev.id <= 8;
lcddev.id |= lcd_rd_data(); /* 读取 0X41 */

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 NT35510 */
    {
        /* 发送秘钥 (厂家提供,照搬即可) */
        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 高 8 位 */
        lcddev.id = lcd_rd_data();     /* 读回 0X55 */
        lcddev.id <= 8;

        lcd_wr_regno(0xC501);           /* 读取 ID 低 8 位 */
        lcddev.id |= lcd_rd_data();    /* 读回 0X10 */
        delay_ms(5);

        /* 也不是 NT5510, 尝试看看是不是 SSD1963 */
        if (lcddev.id != 0X5510)
        {
            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 == 0x5510)
{
    lcd_ex_nt35510_reginit();           /* 执行 NT35510 初始化 */
}
else if (lcddev.id == 0X1963)
{
    lcd_ex_ssd1963_reginit();          /* 执行 SSD1963 初始化 */
    lcd_ssdl_backlight_set(100);       /* 背光设置为最亮 */
}

/* 初始化完成以后,提速 */
if (lcddev.id == 0X7789)                  /* ST7789 提速 */
{
    /* 重新配置写时序控制寄存器的时序 */
    fmc_write_handle.AddressSetupTime = 5;
    fmc_write_handle.DataSetupTime = 5;
    FMC_NORSRAM_Extended_Timing_Init(g_sram_handle.Extended,
                                       &fmc_write_handle,
                                       g_sram_handle.Init.NSBank,
                                       g_sram_handle.Init.ExtendedMode);
}

/* 如果是这几个 IC,则设置 WR 时序为最快 */
if (lcddev.id == 0X9341 || lcddev.id == 0X1963)
{
    /* 重新配置写时序控制寄存器的时序 */
    fmc_write_handle.AddressSetupTime = 3;
    fmc_write_handle.DataSetupTime = 3;
    FMC_NORSRAM_Extended_Timing_Init(g_sram_handle.Extended,
&fmc_write_handle, g_sram_handle.Init.NSBank, g_sram_handle.Init.ExtendedMode);
}

/* 如果是这几个 IC,则设置 WR 时序为最快 */
if (lcddev.id == 0X5310 || lcddev.id == 0X5510)
{
    /* 重新配置写时序控制寄存器的时序 */
    fmc_write_handle.AddressSetupTime = 2;
    fmc_write_handle.DataSetupTime = 2;
    FMC_NORSRAM_Extended_Timing_Init(g_sram_handle.Extended,
                                       &fmc_write_handle,
                                       g_sram_handle.Init.NSBank,
                                       g_sram_handle.Init.ExtendedMode);
}

lcd_display_dir(0);           /* 默认为竖屏 */
LCD_BL(1);
lcd_clear(WHITE);
}

```

从上的代码中可以看出，本章实验的 TFTLCD 驱动是兼容了正点原子的多款 TFTLCD 模块的，因此在初始化完 FSMC 后，会与 TFTLCD 进行通讯，确定 TFTLCD 的型号，然后根据型号针对性地对 TFTLCD 模块进行配置。

TFTLCD 驱动中与 TFTLCD 模块通讯的函数，如下所示：

```

/***
 * @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;
    LCD->LCD_RAM = data;
}

/**

```

```

* @brief    LCD 写寄存器编号或地址
* @param    regno: 寄存器编号或地址
* @retval   无
*/
void lcd_wr_regno(volatile uint16_t regno)
{
    regno = regno;
    LCD->LCD_REG = regno;
}

/**
* @brief    LCD 写寄存器
* @param    regno: 寄存器编号
* @param    data : 要写入的数据
* @retval   无
*/
void lcd_write_reg(uint16_t regno, uint16_t data)
{
    LCD->LCD_REG = regno;
    LCD->LCD_RAM = data;
}

/**
* @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--);
}

/**
* @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;

    lcd_opt_delay(2);
    ram = LCD->LCD_RAM;

    return ram;
}

```

从上面的代码中可以看出，与 TFTLCD 的通讯都是通过 LCD 这一结构体对象来完成的，对于 LCD 结构体的相关定义，如下所示：

```

typedef struct
{
    volatile uint16_t LCD_REG;
    volatile uint16_t LCD_RAM;
} LCD_TypeDef;

#define LCD_FSMC_NEX    1
#define LCD_FSMC_AX     19
#define LCD_BASE \
    (uint32_t)((0x60000000+(0x4000000*(LCD_SMC_NEX-1)))|(((1<<LCD_SMC_AX)*2)-2))
#define LCD ((LCD_TypeDef *)LCD_BASE)

```

从 LCD 结构体的相关定义中可以看出，与 TFTLCD 模块的通讯地址是与 TFTLCD 连接的 FSMC 存储块 1 的区域块和 TFTLCD 模块命令、数据选择信号所连接的 FSMC 地址线是有关的。通过上面宏定义的计算，可以算出 LCD_BASE 宏定义的值为 0x600FFFFE，因此访问

LCD->LCD_REG 就是访问 0x600FFFFE 这一地址, 访问 LCD->LCD_RAM 就是访问 0x60100000 这一地址, 这两个地址也就是通过 FSMC 映射的 TFTLCD 命令和数据寄存器的访问地址。

通过上面介绍的驱动函数就能够与 TFTLCD 模块进行通讯了, 而在 TFTLCD 模块的显示屏上显示出特定的图案或字符或设置 TFTLCD 模块的显示方向等等的操作都是能够通过 TFTLCD 模块规定的特定命令来完成的, 想深究的读者可以观看正点原子 TFTLCD 模块的用户手册或查看实际使用的 TFTLCD 模块的相关文档。

27.2.3 实验应用代码

本章实验的应用代码, 如下所示:

```
int main(void)
{
    uint8_t x = 0;
    uint8_t lcd_id[12];

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置时钟, 480MHz */
    delay_init(480);             /* 初始化延时 */
    usart_init(115200);          /* 初始化串口 */
    led_init();                  /* 初始化 LED */
    mpu_memory_protection();     /* 保护相关存储区域 */
    lcd_init();                  /* 初始化 LCD */

    sprintf((char *)lcd_id, "LCD ID: %04X", lcddev.id);
    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);
x++;

if (x == 12)
{
    x = 0;
}

LED0_TOGGLE();
delay_ms(1000);
}
}
```

从上面的代码中可以看出，在初始化完 LCD 后，便在 LCD 上显示一些本实验的相关信息，随后便每间隔 1000 毫秒就更换一次 LCD 屏幕显示的背景色。

27.3 下载验证

在完成编译和烧录操作后，先将开发板断电，随后将 TFTLCD 模块通过 LCD 转接板与开发板进行连接，最后再给开发板供电（在插拔开发板上的接插件和模块时，要求必须断电操作，否则容易烧毁硬件）。程序运行后，可以看到 LCD 上显示了本实验的相关信息，同时 LCD 显示的背景色被间隔 1000 毫秒就切换一次。

第二十八章 USMART 调试实验

USMART 调试组件是正点原子开发的一款灵巧的串口调试交互组件，其功能类似 Linux 的 Shell，支持通过串口调用程序中的任意函数，对调试代码有很大的帮助。通过本章的学习，读者将学习到 USMART 调试组件的使用。

本章分为如下几个小节：

- 28.1 硬件设计
- 28.2 程序设计
- 28.3 下载验证

28.1 硬件设计

28.1.1 例程功能

1. 通过 USMART 调用程序中的函数，实现对 LCD、LED 和延时操作
2. LED0 闪烁，指示程序正在运行

28.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. USART1
 - USART1_TX - PA9
 - USART1_RX - PA10
4. TIM4

28.1.3 原理图

本章实验使用的 USMART 组件为应用软件，因此没有对应的连接原理图。

28.2 程序设计

28.2.1 USMART 组件

使用 USMART 组件需要先进行相应的移植操作，USMART 组件的移植也非常简单，只需要实现 usmart_port.c 文件中的 5 个函数即可完成移植。

第一个函数为 usmart_get_input_string()，该函数用于 USMART 获取串口输入的数据流，该函数的实现，如下所示：

```
/**  
 * @brief    获取输入数据流(字符串)  
 * @note     USMART 通过解析该函数返回的字符串以获取函数名及参数等信息  
 * @param    无  
 * @retval   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_sto & 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_sto = 0; /* 开启下一次接收 */
    }

    return pbuf;
}

```

从上面的代码中可以看出，该函数就是从 SYSTEM 文件夹 USART 驱动中获取 USART1 输入的数据。

剩余的四个函数，在宏 USMART_ENTIMX_SCAN 开启后才需要定义，该宏用于使能 runtime 统计等功能。

第二个函数为 usmart_timx_reset_time()，该函数用于复位 runtime，该函数的实现，如下所示：

```

/***
 * @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)
{
    /* 清除中断标志位 */
    __HAL_TIM_CLEAR_FLAG(&g_timx_usmart_handle, TIM_FLAG_UPDATE);
    /* 将重装载值设置到最大 */
    __HAL_TIM_SET_AUTORELOAD(&g_timx_usmart_handle, 0xFFFF);
    /* 清空定时器的 CNT */
    __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0);
    usmart_dev.runtime = 0;
}

```

该函数复位了 runtime 功能和用于 runtime 功能的相关 TIM。

第三个函数为 usmart_timx_get_time()，用于 runtime 功能获取时间，该函数的实现，如下所示：

```

/***
 * @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;
}

```

应为该函数能够处理一次定时器的溢出情况，因此能获取到的时间上限为定时器计数最大值的两倍。

第四个函数为 usmart_timx_init()，用于初始化用于 runtime 功能的定时器，该函数的实现，如下所示：

```

/***
 * @brief 定时器初始化函数
 * @param arr: 自动重装载值
 * @param psc: 定时器分频系数
 * @retval 无
 */
void usmart_timx_init(uint16_t arr, uint16_t psc)

```

```
{
    USMART_TIMX_CLK_ENABLE();

    g_timx_usmart_handle.Instance = USMART_TIMX;
    g_timx_usmart_handle.Init.Prescaler = psc;
    g_timx_usmart_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_timx_usmart_handle.Init.Period = arr;
    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);
    HAL_NVIC_SetPriority(USMART_TIMX_IRQn, 3, 3);
    HAL_NVIC_EnableIRQ(USMART_TIMX_IRQn);
}
}
```

可以看到，该函数初始化了一个用于 runtime 功能的定时器，同时使能了该定时器的更新中断。

第五个函数就是用于 runtime 功能的定时器的中断服务函数，该函数的实现，如下所示：

```
/**
 * @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();                                /* 执行 usmart 扫描 */
        __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0); /* 清空定时器的 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 */
}
```

以上就是移植 USMART 组件时需要实现的五个函数，至此 USMART 组件的移植也就基本完成了，接下来便可在 usart_config.c 文件中的 usmart_nametab 数组中添加需要调试的函数。

28.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
/**
 * @brief LED 状态设置
 * @param 无
 * @retval 无
 */
void led_set(uint8_t sta)
{
    LED1(sta);
}

/**
 * @brief 测试函数参数调用
 * @param 无
 * @retval 无
 */
void test_fun(void (*ledset)(uint8_t), uint8_t sta)
{
    ledset(sta);
}

int main(void)
{
    HAL_Init();                                     /* 初始化 HAL 库 */
}
```

```
sys_stm32_clock_init(336, 8, 2, 7); /* 配置时钟, 168MHz */
delay_init(168); /* 初始化延时 */
uart_init(115200); /* 初始化串口 */
usmart_dev.init(84); /* 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();
    delay_ms(500);
}
```

从上面的代码中可以看出，main()函数中初始化了 USMART 组件，并且另外定义了两个函数，分别为：函数 led_set()和函数 test_fun()，这两个函数都被添加到了 usmart_config.c 文件的 usmart_nametab 数组中，用于测试 USMART 组件，同时也添加了 LCD 操作和延时等函数，读者也可以添加自行编写的函数进行测试和调试。

28.3 下载验证

在完成编译和烧录操作后，便可通过串口调试助手“体验”USMART 组件，例如通过串口调试助手发送“led_set(0)\r\n”或“led_set(1)\r\n”，即可看到板载的 LED1 亮起或熄灭，也可使用同样的方式调用 LCD 的操作函数操作 LCD 进行显示。

第二十九章 RTC 实时时钟实验

本章，我们将介绍 STM32H750 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：

29.2 硬件设计

29.3 程序设计

29.4 下载验证

29.1 硬件设计

29.1.1 例程功能

本实验通过 LCD 显示 RTC 时间，并可以通过 usmart 设置 RTC 时间，从而调节时间，或设置 RTC 闹钟，还可以写入或者读取 RTC 后备区域 SRAM。LED1 每两秒闪烁一次，表示进入 WAKE UP 中断。LED0 闪烁，提示程序运行。

29.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. USART1
 - USART1_TX - PA9
 - USART1_RX - PA10
4. RTC

29.1.3 原理图

RTC 属于 STM32H750 内部资源，通过软件设置好就可以了。不过 RTC 不能断电，否则数据就丢失了，我们如果想让时间在断电后还可以继续走，那么必须确保开发板的电池有电。

29.2 程序设计

29.2.1 HAL 库的 PWR 驱动

本章实验要使用到 RTC，因此需要对 RTC 及其相关的寄存器进行配置，但是为了防止误操作，系统复位后备份区域（指 RTC、备份寄存器）是被禁止写访问的，备份区域的写访问是由 PWR 进行配置的，具体的配置步骤如下：

①：使能访问备份区域

在 HAL 库中对应的驱动函数如下：

①：使能访问备份区域

该函数用于使能访问备份区域，其函数原型如下所示：

```
void HAL_PWR_EnableBkUpAccess(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 29.2.1.1 函数 HAL_PWR_EnableBkUpAccess()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
-----	----

无	无
---	---

表 29.2.1.2 函数 HAL_PWR_EnableBkUpAccess()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 使能访问备份区域 */
    HAL_PWR_EnableBkUpAccess();
}
```

29.2.2 HAL 库的 RTC 驱动

本章实验使用了 RTC，RTC 最基本的操作就是设置和获取时间和日期，同时还需要读写 RTC 备份寄存器保存是否已经在初始化过程中设置过时间的标志，其具体的步骤如下：

- ①：初始化 RTC
- ②：读取 RTC 备份寄存器判断是否进行设置过时间
- ③：若未设置过时间，则设置 RTC 的时间
- ④：若未设置过时间，则设置 RTC 的日期
- ⑤：将设置过时间的标志写入 RTC 备份寄存器
- ⑥：读取 RTC 的时间
- ⑦：读取 RTC 的日期

在 HAL 库中对应的驱动函数如下：

①：初始化 RTC

该函数用于初始化 RTC，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针

表 29.2.2.1 函数 HAL_RTC_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.2 函数 HAL_RTC_Init()返回值描述

该函数需要传入 RTC 的句柄指针，该句柄中就包含了 RTC 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t HourFormat;          /* 时间格式 */
    uint32_t AsynchPrediv;        /* 异步预分频器系数 */
    uint32_t SynchPrediv;         /* 同步预分频器系数 */
    uint32_t OutPut;              /* 信号输出 */
    uint32_t OutPutPolarity;      /* 输出极性 */
    uint32_t OutPutType;          /* 输出引脚模式 */
} RTC_InitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_HandleTypeDef rtc_handle = {0};

    /* 初始化 RTC */
    rtc_handle.Instance = RTC;
    rtc_handle.Init.HourFormat = RTC_HOURFORMAT_24;
    rtc_handle.Init.AsynchPrediv = 0x7F;
```

```

    rtc_handle.Init.SynchPrediv = 0xFF;
    rtc_handle.Init.OutPut = RTC_OUTPUT_DISABLE;
    rtc_handle.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
    rtc_handle.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
    HAL_RTC_Init(&rtc_handle);
}

```

②：读取 RTC 备份寄存器

该函数用于读取 RTC 备份寄存器，其函数原型如下所示：

```
uint32_t HAL_RTCEx_BKUPRead(RTC_HandleTypeDef *hrtc, uint32_t BackupRegister);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
BackupRegister	RTC 备份寄存器编号

表 29.2.2.3 函数 HAL_RTCEx_BKUPRead()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
uint32_t 类型数据	RTC 备份寄存器值

表 29.2.2.4 函数 HAL_RTCEx_BKUPRead()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint32_t rtc_backup_reg0;

    /* 读取 RTC 备份寄存器 0 */
    rtc_backup_reg0 = HAL_RTCEx_BKUPRead(&rtc_handle, RTC_BKP_DR0);

    /* Do something. */
}
```

③：配置 RTC 时间

该函数用于配置 RTC 的时间，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RTC_SetTime( RTC_HandleTypeDef *hrtc,
                                    RTC_TimeTypeDef *sTime,
                                    uint32_t Format);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
sTime	指向 RTC 时间结构体的指针 需自行定义，并根据 RTC 的时间填充结构体中的成员变量
Format	时间格式 例如：RTC_FORMAT_BIN 和 RTC_FORMAT_BCD（在 stm32h7xx_hal_rtc.h 文件中有定义）

表 29.2.2.5 函数 HAL_RTC_SetTime()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.6 函数 HAL_RTC_SetTime()返回值描述

该函数使用 RTC_TimeTypeDef 类型的结构体变量传入 RTC 的时间参数，该结构体的定义如下所示：

```

typedef struct
{
    uint8_t Hours;           /* 时 */
    uint8_t Minutes;         /* 分 */
    uint8_t Seconds;         /* 秒 */
}

```

```

    uint8_t TimeFormat;      /* 上下午 */
    uint32_t SubSeconds;    /* 亚秒 */
} RTC_TimeTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_TimeTypeDef time = {0};

    /* 配置 RTC 时间 */
    RTC_TimeTypeDef.Hours = 0;
    RTC_TimeTypeDef.Minutes = 0;
    RTC_TimeTypeDef.Seconds = 0;
    RTC_TimeTypeDef.TimeFormat = RTC_HOURFORMAT12_AM;
    RTC_TimeTypeDef.SubSeconds = 0;
    HAL_RTC_SetTime(&rtc_handle, &time, RTC_FORMAT_BIN);
}

```

④：配置 RTC 日期

该函数用于配置 RTC 的日期，其函数原型如下所示：

```

HAL_StatusTypeDef HAL_RTC_SetDate( RTC_HandleTypeDef *hrtc,
                                    RTC_DateTypeDef *sDate,
                                    uint32_t Format);

```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
sDate	指向 RTC 日期结构体的指针 需自行定义，并根据 RTC 的日期填充结构体中的成员变量
Format	时间格式 例如：RTC_FORMAT_BIN 和 RTC_FORMAT_BCD（在 stm32h7xx_hal_rtc.h 文件中有定义）

表 29.2.2.7 函数 HAL_RTC_SetDate()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.8 函数 HAL_RTC_SetDate()返回值描述

该函数使用 RTC_DateTypeDef 类型的结构体变量传入 RTC 的日期参数，该结构体的定义如下所示：

```

typedef struct
{
    uint8_t WeekDay; /* 星期 */
    uint8_t Month;   /* 月 */
    uint8_t Date;    /* 日 */
    uint8_t Year;    /* 年 */
} RTC_DateTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_DateTypeDef date = {0};

    /* 配置 RTC 日期 */
    date.WeekDay = RTC_WEEKDAY_SATURDAY;
    date.Month = RTC_MONTH_JANUARY;
    date.Date = 1;
    date.Year = 0;
    HAL_RTC_SetDate(&rtc_handle, &date, RTC_FORMAT_BIN);
}

```

}

⑤：写入 RTC 备份寄存器

该函数用于写入 RTC 备份寄存器，其函数原型如下所示：

```
void HAL_RTCEx_BKUPWrite( RTC_HandleTypeDef *hrtc,
                           uint32_t BackupRegister,
                           uint32_t Data);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
BackupRegister	RTC 备份寄存器编号
Data	写入 RTC 备份寄存器的值

表 29.2.2.9 函数 HAL_RTCEx_BKUPWrite()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 29.2.2.10 函数 HAL_RTCEx_BKUPWrite()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 往 RTC 备份寄存器 0 写入 0x50505050 */
    RTC_WriteBackup(&rtc_handle, RTC_BKP_DR0, 0x50505050);
}
```

⑥：读取 RTC 时间

该函数用于读取 RTC 的时间，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RTC_GetTime( RTC_HandleTypeDef *hrtc,
                                    RTC_TimeTypeDef *sTime,
                                    uint32_t Format);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
sTime	指向 RTC 时间结构体的指针 需自行定义，并根据 RTC 的时间填充结构体中的成员变量
Format	时间格式 例如：RTC_FORMAT_BIN 和 RTC_FORMAT_BCD（在 stm32h7xx_hal_rtc.h 文件中有定义）

表 29.2.2.11 函数 HAL_RTC_GetTime()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.12 函数 HAL_RTC_GetTime()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_TimeTypeDef time;

    /* 读取 RTC 的时间 */
    HAL_RTC_GetTime(&rtc_handle, &time, RTC_FORMAT_BIN);

    /* Do something. */
}
```

⑦：读取 RTC 日期

该函数用于读取 RTC 的日期，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RTC_GetDate( RTC_HandleTypeDef *hrtc,
                                    RTC_DateTypeDef *sDate,
                                    uint32_t Format);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
sDate	指向 RTC 日期结构体的指针 需自行定义，并根据 RTC 的日期填充结构体中的成员变量
Format	时间格式 例如：RTC_FORMAT_BIN 和 RTC_FORMAT_BCD（在 stm32h7xx_hal_rtc.h 文件中有定义）

表 29.2.2.13 函数 HAL_RTC_GetDate()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.14 函数 HAL_RTC_GetDate()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_DateTypeDef date;

    /* 读取 RTC 的日期 */
    HAL_RTC_GetDate(&rtc_handle, &date, RTC_FORMAT_BIN);

    /* Do something. */
}
```

本实验还使能了 RTC 的周期性唤醒功能，该功能是低功耗唤醒的一个不错的方案，其具体的使用步骤如下：

①：设置 RTC 中断唤醒定时器

在 HAL 库中对应的驱动函数如下：

①：设置 RTC 中断唤醒定时器

该函数用于设置 RTC 中断唤醒定时器，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RTCEx_SetWakeUpTimer_IT( RTC_HandleTypeDef *hrtc,
                                                uint32_t WakeUpCounter,
                                                uint32_t WakeUpClock);
```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
WakeUpCounter	唤醒计数器
WakeUpClock	唤醒时钟

表 29.2.2.15 函数 HAL_RTCEx_SetWakeUpTimer_IT()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.16 函数 HAL_RTCEx_SetWkaeUpTimer_IT()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 设置 RTC 中断唤醒计数器 */
    HAL_RTCEx_SetWakeUpTimer_IT(&rtc_handle,
```

```

    0,
    RTC_WAKEUPCLOCK_CK_SPRE_16BITS);
}

```

本实验同时也使能了 RTC 的闹钟功能，闹钟功能可以在 RTC 时间到达设定值时触发中断，其具体的使用步骤如下：

①：设置 RTC 中断闹钟

在 HAL 库中对应的驱动函数如下：

①：设置 RTC 中断闹钟

该函数用于设置 RTC 的中断闹钟，其函数原型如下所示：

```

HAL_StatusTypeDef HAL_RTC_SetAlarm_IT( RTC_HandleTypeDef *hrtc,
                                       RTC_AlarmTypeDef *sAlarm,
                                       uint32_t Format);

```

该函数的形参描述，如下表所示：

形参	描述
hrtc	指向 RTC 句柄的指针
sAlarm	指向 RTC 闹钟结构体的指针 需自行定义，并根据 RTC 闹钟的配置参数充结构体中的成员变量
Format	时间格式 例如：RTC_FORMAT_BIN 和 RTC_FORMAT_BCD（在 stm32h7xx_hal_rtc.h 文件中有定义）

表 29.2.2.23 函数 HAL_RTC_SetAlarm_IT()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 29.2.2.24 函数 HAL_RTC_SetAlarm_IT()返回值描述

该函数使用 RTC_AlarmTypeDef 类型的结构体变量传入 RTC 闹钟的配置参数，该结构体的定义如下所示：

```

typedef struct
{
    RTC_TimeTypeDef AlarmTime;      /* 闹钟时间 */
    uint32_t AlarmMask;           /* 闹钟掩码 */
    uint32_t AlarmSubSecondMask;   /* 闹钟亚秒掩码 */
    uint32_t AlarmDateWeekDaySel; /* 日闹钟或星期闹钟 */
    uint8_t AlarmDateWeekDay;     /* 闹钟日期 */
    uint32_t Alarm;               /* 闹钟对象 */
} RTC_AlarmTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    RTC_AlarmTypeDef alarm = {0};

    /* 设置 RTC 闹钟中断 */
    alarm.AlarmTime.Hours = 0;
    alarm.AlarmTime.Minutes = 0;
    alarm.AlarmTime.Seconds = 0;
    alarm.AlarmTime.TimeFormat = RTC_HOURFORMAT12_AM;
    alarm.AlarmTime.SubSeconds = 0;
    alarm.AlarmMask = RTC_ALARMMASK_NONE;
    alarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_NONE;
    alarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_WEEKDAY;
    alarm.AlarmDateWeekDay = RTC_WEEKDAY_MONDAY;
    alarm.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&rtc_handle, &alarm, RTC_FORMAT_BIN);
}

```

29.2.3 RTC 驱动

本章实验的 RTC 驱动主要负责向应用层提供 RTC 的初始化和配置自动唤醒及闹钟的函数。本章实验中，RTC 的驱动代码包括 `rtc.c` 和 `rtc.h` 两个文件。

RTC 驱动中，RTC 的初始化函数，如下所示：

```
/***
 * @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 BKPO==0X5050 时, 使用的是 LSE
 *             当 BKPO==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;

    g_RTC_handle.Instance = RTC;
    g_RTC_handle.Init.HourFormat = RTC_HOURFORMAT_24; /* RTC 设置为 24 小时格式 */
    g_RTC_handle.Init.AsynchPrediv = 0X7F;           /* RTC 异步分频系数(1~0X7F) */
    g_RTC_handle.Init.SynchPrediv = 0xFF;            /* RTC 同步分频系数(0~7FFF) */
    g_RTC_handle.Init.OutPut = RTC_OUTPUT_DISABLE;
    g_RTC_handle.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
    g_RTC_handle.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;

    bkpflag = rtc_read_bkr(0);                         /* 读取 BKPO 的值 */

    if (HAL_RTC_Init(&g_RTC_handle) != HAL_OK)
    {
        return 1;
    }

    if ((bkpflag != 0X5050) && (bkpflag != 0x5051)) /* 之前未初始化过, 重新配置 */
    {
        rtc_set_time(23, 59, 56, RTC_HOURFORMAT12_AM); /* 设置时间, 根据实际时间修改 */
        rtc_set_date(23, 4, 24, 7);                   /* 设置日期 */
    }

    return 0;
}
```

从上面的代码中可以看出，RTC 的初始化函数中，会先读取 RTC 备份寄存器来判断是否初始化过 RTC 然后再初始化 RTC，RTC 初始化完成后，若是第一次初始化 RTC，则设置 RTC 的时间，否则不设置 RTC 的时间。

RTC 驱动中设置、获取 RTC 时间、日期的四个函数，如下所示：

```
/***
 * @brief      RTC 时间设置
 * @param      hour,min,sec: 小时,分钟,秒钟
 * @param      ampm       : AM/PM, 0=AM/24H; 1=PM/12H;
 * @retval     0, 成功
 *             1, 进入初始化模式失败
 */
HAL_StatusTypeDef rtc_set_time(uint8_t hour, uint8_t min, uint8_t sec, uint8_t ampm)
{
```

```
RTC_TimeTypeDef rtc_time;

rtc_time.Hours = hour;
rtc_time.Minutes = min;
rtc_time.Seconds = sec;
rtc_time.TimeFormat = ampm;
rtc_time.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
rtc_time.StoreOperation = RTC_STOREOPERATION_RESET;

return HAL_RTC_SetTime(&g_rtc_handle, &rtc_time, RTC_FORMAT_BIN);
}

/**
 * @brief      RTC 日期设置
 * @param      year,month,date : 年(0~99),月(1~12),日(0~31)
 * @param      week           : 星期(1~7,0,非法!)
 * @retval     0,成功
 *             1,进入初始化模式失败
 */
HAL_StatusTypeDef rtc_set_date(uint8_t year, uint8_t month, uint8_t date,
                               uint8_t week)
{
    RTC_DateTypeDef rtc_date;

    rtc_date.Date = date;
    rtc_date.Month = month;
    rtc_date.WeekDay = week;
    rtc_date.Year = year;

    return HAL_RTC_SetDate(&g_rtc_handle, &rtc_date, RTC_FORMAT_BIN);
}

/**
 * @brief      获取 RTC 时间
 * @param      *hour,*min,*sec : 小时,分钟,秒钟
 * @param      *ampm           : AM/PM,0=AM/24H,1=PM.
 * @retval     无
 */
void rtc_get_time(uint8_t *hour, uint8_t *min, uint8_t *sec, uint8_t *ampm)
{
    RTC_TimeTypeDef rtc_time;

    HAL_RTC_GetTime(&g_rtc_handle, &rtc_time, RTC_FORMAT_BIN);

    *hour = rtc_time.Hours;
    *min = rtc_time.Minutes;
    *sec = rtc_time.Seconds;
    *ampm = rtc_time.TimeFormat;
}

/**
 * @brief      获取 RTC 日期
 * @param      *year,*mon,*date: 年,月,日
 * @param      *week           : 星期
 * @retval     无
 */
void rtc_get_date(uint8_t *year, uint8_t *month, uint8_t *date, uint8_t *week)
{
    RTC_DateTypeDef rtc_date;

    HAL_RTC_GetDate(&g_rtc_handle, &rtc_date, RTC_FORMAT_BIN);

    *year = rtc_date.Year;
    *month = rtc_date.Month;
    *date = rtc_date.Date;
```

```
*week = rtc_date.WeekDay;
}
```

以上四个获取、设置 RTC 时间、日期的函数，均是对 HAL 库中 RTC 驱动的简单封装。

RTC 驱动中，配置 RTC 唤醒中断及其对应的唤醒定时器事件回调函数，如下所示：

```
/***
 * @breif      周期性唤醒定时器设置
 * @param      wksel
 * @arg        RTC_WAKEUPCLOCK_RTCCLK_DIV16      ((uint32_t)0x00000000)
 * @arg        RTC_WAKEUPCLOCK_RTCCLK_DIV8       ((uint32_t)0x00000001)
 * @arg        RTC_WAKEUPCLOCK_RTCCLK_DIV4       ((uint32_t)0x00000002)
 * @arg        RTC_WAKEUPCLOCK_RTCCLK_DIV2       ((uint32_t)0x00000003)
 * @arg        RTC_WAKEUPCLOCK_CK_SPRE_16BITS    ((uint32_t)0x00000004)
 * @arg        RTC_WAKEUPCLOCK_CK_SPRE_17BITS    ((uint32_t)0x00000006)
 * @note       000,RTC/16;001,RTC/8;010,RTC/4;011,RTC/2;
 * @note       注意：RTC 就是 RTC 的时钟频率，即 RTCCLK！
 * @param      cnt：自动重装载值。减到 0，产生中断。
 * @retval     无
 */
void rtc_set_wakeup(uint8_t wksel, uint16_t cnt)
{
    __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_WUTF);

    HAL_RTCEx_SetWakeUpTimer_IT(&g_rtc_handle, cnt, wksel);

    HAL_NVIC_SetPriority(RTC_WKUP_IRQn, 2, 2);
    HAL_NVIC_EnableIRQ(RTC_WKUP_IRQn);
}

/***
 * @breif      RTC WAKE UP 中断处理处理回调函数
 * @param      hrtc:RTC 句柄
 * @retval     无
 */
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef *hrtc)
{
    LED1_TOGGLE();
}
```

从上面的代码中可以看出，在 RTC 的唤醒定时器事件回调函数中翻转了 LED1 的亮灭状态，因此 RTC 唤醒时间将会周期性地发生，也就能看到 LED1 周期性地改变状态。

RTC 驱动中，配置 RTC 闹钟中断及其对应的闹钟事件回调函数，如下所示：

```
/***
 * @breif      设置闹钟时间(按星期闹铃,24 小时制)
 * @param      week      : 星期几(1~7)
 * @param      hour,min,sec: 小时,分钟,秒钟
 * @retval     无
 */
void rtc_set_alarm(uint8_t week, uint8_t hour, uint8_t min, uint8_t sec)
{
    RTC_AlarmTypeDef rtc_alarm;

    rtc_alarm.AlarmTime.Hours = hour;
    rtc_alarm.AlarmTime.Minutes = min;
    rtc_alarm.AlarmTime.Seconds = sec;
    rtc_alarm.AlarmTime.SubSeconds = 0;
    rtc_alarm.AlarmTime.TimeFormat = RTC_HOURFORMAT12_AM;

    rtc_alarm.AlarmMask = RTC_ALARMMASK_NONE;
    rtc_alarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_NONE;
    rtc_alarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_WEEKDAY;
    rtc_alarm.AlarmDateWeekDay = week;
    rtc_alarm.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&g_rtc_handle, &rtc_alarm, RTC_FORMAT_BIN);
}
```

```

    HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 1, 2);
    HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
}

/***
 * @brief      RTC 闹钟 A 中断处理回调函数
 * @param      hrtc:RTC 句柄
 * @retval     无
 */
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("ALARM A! \r\n");
}

```

从上面的代码中可以看到，在 RTC 的闹钟中断中通过串口打印了闹钟的提示。

29.2.4 RTC 驱动

本章实验的 RTC 驱动主要负责向应用层提供 RTC 的初始化和配置自动唤醒及闹钟的函数。本章实验中，RTC 的驱动代码包括 `rtc.c` 和 `rtc.h` 两个文件。

RTC 驱动中，RTC 的初始化函数，如下所示：

```

/***
 * @brief      初始化 RTC
 * @param      无
 * @retval     初始化结果
 *             0: 初始化成功
 *             1: 初始化失败
 */
uint8_t rtc_init(void)
{
    uint16_t flag;

    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_RCC RTC_ENABLE();
    HAL_PWR_EnableBkUpAccess();

    /* 初始化 RTC */
    g_rtc_handle.Instance = RTC;
    g_rtc_handle.Init.HourFormat = RTC_HOURFORMAT_24;
    g_rtc_handle.Init.AsynchPrediv = 0x7F;
    g_rtc_handle.Init.SynchPrediv = 0xFF;
    g_rtc_handle.Init.Output = RTC_OUTPUT_DISABLE;
    g_rtc_handle.Init.OutputPolarity = RTC_OUTPUT_POLARITY_HIGH;
    g_rtc_handle.Init.OutputType = RTC_OUTPUT_TYPE_OPENDRAIN;

    /* 从后备寄存器读取 RTC 初始化标志 */
    flag = rtc_read_bkr(0);

    if (HAL_RTC_Init(&g_rtc_handle) != HAL_OK)
    {
        return 1;
    }

    /* RTC 第一次初始化 */
    if ((flag != 0x5051) && (flag != 0x5050))
    {
        /* 设置 RTC 时间和日期信息 */
        rtc_set_time(8, 0, 0, 0);
        rtc_set_date(23, 4, 23, 7);
    }

    return 0;
}

```

从上面的代码中可以看出，RTC 的初始化函数中，会先读取 RTC 备份寄存器来判断是否初始化过 RTC 然后再初始化 RTC，RTC 初始化完成后，若是第一次初始化 RTC，则设置 RTC 的时间，否则不设置 RTC 的时间。

RTC 驱动中设置、获取 RTC 时间、日期的四个函数，如下所示：

```
/** * @brief 设置 RTC 时间信息 * @param hour: 时 * @param minute: 分 * @param second: 秒 * @param ampm: 上下午 * @arg 0: 上午 * @arg 1: 下午 * @retval 设置结果 * @arg 0: 设置成功 * @arg 1: 设置失败 */ uint8_t rtc_set_time(uint8_t hour, uint8_t minute, uint8_t second, uint8_t ampm) {    RTC_TimeTypeDef rtc_time_struct = {0};    rtc_time_struct.Hours = hour;    rtc_time_struct.Minutes = minute;    rtc_time_struct.Seconds = second;    rtc_time_struct.TimeFormat = ampm;    rtc_time_struct.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;    rtc_time_struct.StoreOperation = RTC_STOREOPERATION_RESET;    if (HAL_RTC_SetTime(&g_rtc_handle, &rtc_time_struct, RTC_FORMAT_BIN) != HAL_OK)    {        return 1;    }    return 0; } /** * @brief 设置 RTC 日期信息 * @param year: 年 * @param month: 月 * @param date: 日 * @param week: 星期 * @retval 设置结果 * @arg 0: 设置成功 * @arg 1: 设置失败 */ uint8_t rtc_set_date(uint8_t year, uint8_t month, uint8_t date, uint8_t week) {    RTC_DateTypeDef rtc_date_struct = {0};    rtc_date_struct.WeekDay = week;    rtc_date_struct.Month = month;    rtc_date_struct.Date = date;    rtc_date_struct.Year = year;    if (HAL_RTC_SetDate(&g_rtc_handle, &rtc_date_struct, RTC_FORMAT_BIN) != HAL_OK)    {        return 1;    }    return 0; }
```

```

/***
 * @brief    获取 RTC 时间信息
 * @param    hour: 时
 * @param    minute: 分
 * @param    second: 秒
 * @param    ampm: 上下午
 * @arg      0: 上午
 * @arg      1: 下午
 * @retval   无
 */
void rtc_get_time( uint8_t *hour,
                    uint8_t *minute,
                    uint8_t *second,
                    uint8_t *ampm)
{
    RTC_TimeTypeDef rtc_time_struct = {0};

    HAL_RTC_GetTime(&g_rtc_handle, &rtc_time_struct, RTC_FORMAT_BIN);

    *hour = rtc_time_struct.Hours;
    *minute = rtc_time_struct.Minutes;
    *second = rtc_time_struct.Seconds;
    *ampm = rtc_time_struct.TimeFormat;
}

/***
 * @brief    获取 RTC 日期信息
 * @param    year: 年
 * @param    month: 月
 * @param    date: 日
 * @param    week: 星期
 * @retval   无
 */
void rtc_get_date(uint8_t *year, uint8_t *month, uint8_t *date, uint8_t *week)
{
    RTC_DateTypeDef rtc_date_struct = {0};

    HAL_RTC_GetDate(&g_rtc_handle, &rtc_date_struct, RTC_FORMAT_BIN);

    *year = rtc_date_struct.Year;
    *month = rtc_date_struct.Month;
    *date = rtc_date_struct.Date;
    *week = rtc_date_struct.WeekDay;
}

```

以上四个获取、设置 RTC 时间、日期的函数，均是对 HAL 库中 RTC 驱动的简单封装。

RTC 驱动中，配置 RTC 唤醒中断及其对应的唤醒定时器事件回调函数，如下所示：

```

/***
 * @brief    设置 RTC 周期性唤醒中断
 * @param    clock: 唤醒时钟
 * @param    count: 唤醒计数器
 * @retval   无
 */
void rtc_set_wakeup(uint8_t clock, uint8_t count)
{
    HAL_NVIC_SetPriority(RTC_WKUP_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(RTC_WKUP_IRQn);

    HAL_RTCEx_SetWakeUpTimer_IT(&g_rtc_handle, count, clock);
}

/***
 * @brief    HAL 库 RTC 唤醒中断回调函数
 * @param    hrtc: RTC 句柄
 */

```

```

 * @retval 无
 */
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef *hrtc)
{
    LED1_TOGGLE();
}

```

从上面的代码中可以看出，在 RTC 的唤醒定时器事件回调函数中翻转了 LED1 的亮灭状态，因此 RTC 唤醒时间将会周期性地发生，也就能看到 LED1 周期性地改变状态。

RTC 驱动中，配置 RTC 闹钟中断及其对应的闹钟事件回调函数，如下所示：

```

/***
 * @brief   设置 RTC 闹钟时间信息
 * @param   week: 星期
 * @param   hour: 时
 * @param   minute: 分
 * @param   second: 秒
 * @retval 无
 */
void rtc_set_alarm(uint8_t week, uint8_t hour, uint8_t minute, uint8_t second)
{
    RTC_AlarmTypeDef rtc_alarm_struct = {0};

    HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);

    /* 设置闹钟中断 */
    rtc_alarm_struct.AlarmTime.Hours = hour;
    rtc_alarm_struct.AlarmTime.Minutes = minute;
    rtc_alarm_struct.AlarmTime.Seconds = second;
    rtc_alarm_struct.AlarmTime.TimeFormat = RTC_HOURFORMAT12_AM;
    rtc_alarm_struct.AlarmTime.SubSeconds = 0;
    rtc_alarm_struct.AlarmMask = RTC_ALARMMASK_NONE;
    rtc_alarm_struct.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_NONE;
    rtc_alarm_struct.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_WEEKDAY;
    rtc_alarm_struct.AlarmDateWeekDay = week;
    rtc_alarm_struct.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&g_rtc_handle, &rtc_alarm_struct, RTC_FORMAT_BIN);
}

/***
 * @brief   HAL 库 RTC 闹钟 A 中断回调函数
 * @param   hrtc: RTC 句柄
 * @retval 无
 */
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("Alarm A!\r\n");
}

```

从上面的代码中可以看到，在 RTC 的闹钟中断中通过串口打印了闹钟的提示。

29.2.5 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t hour;
    uint8_t min;
    uint8_t sec;
    uint8_t ampm;
    uint8_t year;
    uint8_t month;
    uint8_t date;
    uint8_t week;
    uint8_t tbuf[40];

```

```

uint8_t t = 0;

sys_cache_enable();           /* 打开 L1-Cache */
HAL_Init();                  /* 初始化 HAL 库 */
sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
delay_init(480);             /* 延时初始化 */
usart_init(115200);          /* 串口初始化为 115200 */
usmart_dev.init(240);         /* 初始化 USMART */
led_init();                  /* 初始化 LED */
mpu_memory_protection();     /* 保护相关存储区域 */
lcd_init();                  /* 初始化 LCD */
rtc_init();                  /* 初始化 RTC */

/* 配置 WAKE UP 中断, 1 秒钟中断一次 */
rtc_set_wakeup(RTC_WAKEUPCLOCK_CK_SPRE_16BITS, 0);

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "RTC TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

while (1)
{
    t++;

    if ((t % 10) == 0)           /* 每 100ms 更新一次显示数据 */
    {
        rtc_get_time(&hour, &min, &sec, &ampm);
        sprintf((char *)tbuf, "Time:%02d:%02d:%02d", hour, min, sec);
        lcd_show_string(30, 130, 210, 16, 16, (char*)tbuf, RED);
        rtc_get_date(&year, &month, &date, &week);
        sprintf((char *)tbuf, "Date:20%02d-%02d-%02d", year, month, date);
        lcd_show_string(30, 150, 210, 16, 16, (char*)tbuf, RED);
        sprintf((char *)tbuf, "Week:%d", week);
        lcd_show_string(30, 170, 210, 16, 16, (char*)tbuf, RED);
    }

    if ((t % 20) == 0)
    {
        LED0_TOGGLE();           /* 每 200ms, 翻转一次 LED0 */
    }

    delay_ms(10);
}
}

```

从上面的代码中可以看到，在初始化完 RTC 后，配置了 RTC 周期性地唤醒，且唤醒周期为 1Hz，因此应该能看到 LED1 以 0.5Hz 的频率闪烁，随后便每隔 100 毫秒获取一次 RTC 的时间和日期，并在 LCD 上进行显示。

本实验同时也使用的 USMART 调试组件，并在 `uart_config.c` 文件中添加了 RTC 驱动中相关的函数，以便调试。

29.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时地显示着 RTC 的时间，并且可以看到 LED1 在 RTC 周期性唤醒的驱动下以 0.5Hz 的频率闪烁着，此时可以通过串口调试助手调用 USMART 调试组件的 `rtc_set_alarma()` 函数来设置 RTC 的闹钟，当通过 LCD 观察到 RTC 的时间达到设置的闹钟时间后，可以看到串口调试助手上打印了“ALARM A!\r\n”的字符串提示。

第三十章 硬件随机数实验

本章介绍 STM32H750 随机数 (RNG) 的使用，RNG 能提供 32 位的随机数。通过本章的学习，读者将学习到 RNG 的使用。

本章分为如下几个小节：

30.1 硬件设计

30.2 程序设计

30.3 下载验证

30.1 硬件设计

30.1.1 例程功能

1. TFTLCD 上不断显示 0-9 之间的随机数
2. 按下 KEY0 按键后，TFTLCD 上显示一次 32 位宽的随机数
3. LED0 闪烁，提示程序正在运行

30.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
KEY0 - PA15
4. RNG

30.1.3 原理图

本章实验使用的 RNG 为 STM32H750 的片上资源，因此没有对应的连接原理图。

30.2 程序设计

30.2.1 HAL 库的 RNG 驱动

本章实验要使用 RNG 生成随机数，其使用方式时非常简单的，具体的使用步骤如下：

- ①：初始化 RNG
 - ②：读取 RNG 生成的 32 位随机数
- 在 HAL 库中对应的驱动函数如下：

①：初始化 RNG

该函数用于初始化 RNG，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_RNG_Init(RNG_HandleTypeDef *hrng);
```

该函数的形参描述，如下表所示：

形参	描述
hrng	指向 RNG 句柄的指针

表 30.2.1.1 函数 HAL_RNG_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 30.2.1.2 函数 HAL_RNG_Init()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"
```

```

void example_fun(void)
{
    RNG_HandleTypeDef rng_handle = {0};

    /* 初始化 RNG */
    rng_handle.Instance = RNG;
    HAL_RNG_Init(&rng_handle);
}

```

②：读取 32 位随机数

该函数用于读取 RNG 生成的 32 位随机数，其函数原型如下所示：

```

HAL_StatusTypeDef HAL_RNG_GenerateRandomNumber(    RNG_HandleTypeDef *hrng,
                                                uint32_t *random32bit);

```

该函数的形参描述，如下表所示：

形参	描述
hrng	指向 RNG 句柄的指针
random32bit	32 位随机数

表 30.2.1.3 函数 HAL_RNG_GenerateRandomNumber()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 30.2.1.4 函数 HAL_RNG_GenerateRandomNumber()返回值描述

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint32_t random;

    /* 获取 RNG 产生的 32 位随机数 */
    HAL_RNG_GenerateRandomNumber(&rng_handle, &random);

    /* Do something. */
}

```

30.2.2 RNG 驱动

本章实验的 RNG 驱动主要负责向应用层提供 RNG 的初始化及获取随机数的函数。本章实验中，RNG 的驱动代码包括 rng.c 和 rng.h 两个文件。

RNG 驱动中，RNG 的初始化函数，如下所示：

```

/**
 * @brief      初始化 RNG
 * @param      无
 * @retval     0,成功;1,失败
 */
uint8_t rng_init(void)
{
    uint16_t retry = 0;

    g_rng_handle.Instance = RNG;
    HAL_RNG_DeInit(&g_rng_handle);
    HAL_RNG_Init(&g_rng_handle);

    while (_HAL_RNG_GET_FLAG(&g_rng_handle,
                             RNG_FLAG_DRDY) == RESET && retry < 10000)
    {
        retry++;
        delay_us(10);
    }
    if (retry >= 10000)

```

```

    {
        return 1;
    }
    return 0;
}

```

RNG 驱动中提供了两个获取随机数的函数，一个是直接获取 RNG 生成的 32 位随机数，另一个是将 RNG 生成的 32 位随机数进行处理得到一个在指定范围内的随机数，这两个函数如下所示：

```

/***
 * @brief      得到随机数
 * @param      无
 * @retval     获取到的随机数(32bit)
 */
uint32_t rng_get_random_num(void)
{
    uint32_t randomnum;

    HAL_RNG_GenerateRandomNumber(&g_rng_handle, &randomnum);

    return randomnum;
}

/***
 * @brief      得到某个范围内的随机数
 * @param      min,max: 最小,最大值.
 * @retval     得到的随机数(rval),满足:min<=rval<=max
 */
int rng_get_random_range(int min, int max)
{
    uint32_t randomnum;

    HAL_RNG_GenerateRandomNumber(&g_rng_handle, &randomnum);

    return randomnum%(max-min+1) + min;
}

```

30.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t key;
    uint32_t random;
    uint8_t t = 0;

    sys_cache_enable();                                /* 打开 L1-Cache */
    HAL_Init();                                       /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);             /* 配置系统时钟, 480Mhz */
    delay_init(480);                                 /* 初始化延时功能 */
    usart_init(115200);                             /* 初始化串口 */
    usmart_dev.init(240);                           /* 初始化 USMART */
    mpu_memory_protection();                         /* 保护相关存储区域 */
    led_init();                                     /* 初始化 LED */
    lcd_init();                                    /* 初始化 LCD */
    key_init();                                   /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "RNG TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (rng_init())                               /* 初始化随机数发生器 */
    {

```

```
lcd_show_string(30, 110, 200, 16, 16, "RNG Error!    ", RED);
delay_ms(200);
lcd_show_string(30, 110, 200, 16, 16, "RNG Trying...", RED);

}

lcd_show_string(30, 110, 200, 16, 16, "RNG Ready!    ", RED);
lcd_show_string(30, 130, 200, 16, 16, "KEY0:Get Random Num", RED);
lcd_show_string(30, 150, 200, 16, 16, "Random Num:", RED);
lcd_show_string(30, 180, 200, 16, 16, "Random Num[0-9]:", RED);

while (1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)                                /* 获取随机数并显示至 LCD */
    {
        random = rng_get_random_num();
        lcd_show_num(30 + 8 * 11, 150, random, 10, 16, BLUE);
    }

    if ((t % 20) == 0)                                  /* 获取 0~9 间的随机数并显示至 LCD */
    {
        LED0_TOGGLE();                                 /* 每 200ms, 翻转一次 LED0 */
        random = rng_get_random_range(0, 9); /* 取[0,9]区间的随机数 */
        lcd_show_num(30 + 8 * 16, 180, random, 1, 16, BLUE); /* 显示随机数 */
    }

    delay_ms(10);
    t++;
}
}
```

从上面的代码中可以看出，在进行完相关的初始化后，便会在 LCD 屏幕上不断显示一个 0-9 的随机数，若按下 KEY0 按键，则会在 LCD 屏幕上显示 RNG 生成的 32 位随机数。

30.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上不断地刷新显示一个 0-9 的随机数，此时若按下 KEY0 按键，则可以看到 LCD 屏幕上刷新显示了一个 32 位的随机数。

第三十一章 PVD 电压监控实验

本章介绍 STM32H750 电源电压监测器（PVD）的使用，PVD 可以设置一个电压阈值，当监测到电源电压低于该阈值后，可以触发中断，以完成一些紧急处理。通过本章的学习，读者将学习到 PVD 的使用。

本章分为如下几个小节：

- 31.1 硬件设计
- 31.2 程序设计
- 31.3 下载验证

31.1 硬件设计

31.1.1 例程功能

1. TFTLCD 上显示实验信息
2. 供电不足时，LED1 亮起，LCD 上提示“PVD Low Voltage!”
3. 供电正常时，LED1 熄灭，LCD 上提示“PVD Voltage OK! ”
4. LED0 闪烁，提示程序正在运行

31.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. PVD

31.1.3 原理图

本章实验使用的 PVD 为 STM32H750 的片上资源，因此没有对应的连接原理图。

31.2 程序设计

31.2.1 HAL 库的 PWR 驱动

本章实验使用的 PVD 为 PWR 的子模块，因此对 PVD 的操作函数都由 HAL 库中的 PWR 驱动提供，使用 PVD 的具体步骤如下：

- ①：配置 PVD
- ②：使能 PVD

在 HAL 库中对应的驱动函数如下：

①：配置 PVD

该函数用于配置 PVD，其函数原型如下所示：

```
void HAL_PWR_ConfigPVD(PWR_PVDTTypeDef *sConfigPVD);
```

该函数的形参描述，如下表所示：

形参	描述
sConfigPVD	指向 PVD 配置结构体的指针 需自行定义，并根据 PVD 的配置参数充结构体中的成员变量

表 31.2.1.1 函数 HAL_PWR_ConfigPVD()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
-----	----

无	无
---	---

表 31.2.1.2 函数 HAL_PWR_ConfigPVD()返回值描述

该函数使用 PWR_PVDTTypeDef 类型的结构体变量传入 PVD 的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t PVDLevel; /* 等级 */
    uint32_t Mode; /* 模式 */
} PWR_PVDTTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    PWR_PVDTTypeDef config = {0};

    /* 配置 PVD */
    config.PVDLevel = PWR_PVDLEVEL_7;
    config.Mode = PWR_PVD_MODE_IT_RISING_FALLING;
    HAL_PWR_ConfigPVD(&config);
}
```

②：使能 PVD

该函数用于使能 PVD，其函数原型如下所示：

```
void HAL_PWR_EnablePVD(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 31.2.1.3 函数 HAL_PWR_EnablePVD()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 31.2.1.4 函数 HAL_PWR_EnablePVD()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 使能 PVD */
    HAL_PWR_EnablePVD();
}
```

31.2.2 PWR 驱动

本章实验的 PWR 驱动主要负责向应用层提供 PVD 的初始化函数，并实现 PVD 的中断回调函数。本章实验中，PWR 的驱动代码包括 pwr.c 和 pwr.h 两个文件。

PWR 驱动中，初始化 PVD 的函数，如下所示：

```
/***
 * @brief      初始化 PVD 电压监视器
 * @param      pls: 电压等级
 * @arg        PWR_PVDLEVEL_0, 1.95V; PWR_PVDLEVEL_1, 2.1V
 * @arg        PWR_PVDLEVEL_2, 2.25V; PWR_PVDLEVEL_3, 2.4V;
 * @arg        PWR_PVDLEVEL_4, 2.55V; PWR_PVDLEVEL_5, 2.7V;
 * @arg        PWR_PVDLEVEL_6, 2.85V; PWR_PVDLEVEL_7, 使用 PVD_IN 脚上的电压 (与 Vrefint
 * 比较)
 * @retval     无
 */
void pwr_pvd_init(uint32_t pls)
{
```

```

PWR_PVDTTypeDef pwr_pvd = {0};

HAL_PWR_EnablePVD();

pwr_pvd.PVDLevel = pls;
pwr_pvd.Mode = PWR_PVD_MODE_IT_RISING_FALLING;
HAL_PWR_ConfigPVD(&pwr_pvd);

HAL_NVIC_SetPriority(PVD_AVD_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(PVD_AVD_IRQn);
HAL_PWR_EnablePVD();
}

```

从上面的代码中可以看到，PVD 的初始化函数中，根据函数传入的参数配置了 PVD 的电压阈值，并开启了 PVD 的相关中断和使能 PVD。

PWR 驱动中，PVD 的中断回调函数，如下所示：

```

/**
 * @brief      PVD 中断回调函数
 * @param      无
 * @retval     无
 */
void PVD_AVD_IRQHandler(void)
{
    if (__HAL_PWR_GET_FLAG(PWR_FLAG_PVDO) == SET)
    {
        lcd_show_string(30, 130, 200, 16, 16, "PVD Low Voltage!", RED);
        LED1(0);
    }
    else
    {
        lcd_show_string(30, 130, 200, 16, 16, "PVD Voltage OK! ", BLUE);
        LED1(1);
    }
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_PVDO);
}

```

从 PVD 的中断回调函数中可以看到，当 PVD 监测到电源电压小于设定的电压阈值时，会在 LCD 上显示电压低的提示并点亮 LED1，在 PVD 监测到电源电压恢复至设定的电压阈值时，会在 LCD 上显示电压正常的提示并熄灭 LED1。

虽然 PVD 在监测到电源电压低于设定的电压阈值时，会有相应的操作，但是由于电压过低可能导致无法观察到部分操作的现象。

31.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t t = 0;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    pwr_pvd_init(PWR_PVDELEVEL_5); /* PVD 2.7V 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);
    lcd_show_string(30, 110, 200, 16, 16, "PVD Voltage OK! ", BLUE);
}

```

```
while (1)
{
    if ((t % 20) == 0)
    {
        LED0_TOGGLE();
    }
    delay_ms(10);
    t++;
}
```

本实验的应用代码很简单，主要就是配置了 PVD 的阈值电压为 2.7V。

31.3 下载验证

在完成编译和烧录操作后，若系统的供电正常，便可以在 LCD 上看到“PVD Voltage OK!”的提示，并且 LED1 也处于熄灭状态；若系统的供电低于设置的 PVD 电压阈值 2.7V 时，便可以在 LCD 上看到“PVD Low Voltage!”的提示，并且 LED1 也会亮起（系统供电过低时，可能出现很多意想不到的结果，因此可能会看不到部分现象）。

正常情况下开发板的供电都是正常的，若读者强制影响开发板的供电，可能导致不可逆的硬件损坏。

第三十二章 睡眠模式实验

本章介绍 STM32H750 低功耗模式中的睡眠模式，通过调用 WFI 命令进入睡眠模式后，其内核将停止以降低功耗，在该睡眠模式下可以被任意中断唤醒。通过本章的学习，读者将学习到低功耗模式中睡眠模式的使用。

本章分为如下几个小节：

- 32.1 硬件设计
- 32.2 程序设计
- 32.3 下载验证

32.1 硬件设计

32.1.1 例程功能

1. TFTLCD 上显示实验信息
2. LED0 闪烁时，按下 KEY0 按键后，LED1 亮起，LED0 停止闪烁
3. LED0 停止闪烁时，按下 WKUP 按键后，LED1 熄灭，LED0 闪烁

32.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15

32.1.3 原理图

本章实验介绍 STM32H750 低功耗模式中的睡眠模式，不涉及连接原理图。

32.2 程序设计

32.2.1 HAL 库的 PWR 驱动

HAL 库中提供了配置进入睡眠模式的驱动函数，其配置步骤如下：

①：进入睡眠模式

在 HAL 库中对应的驱动函数如下：

①：进入睡眠模式

该函数用于进入睡眠模式，其函数原型如下所示：

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

该函数的形参描述，如下表所示：

形参	描述
Regulator	睡眠模式下调压器的工作状态 例如：PWR_MAINREGULATOR_ON 和 PWR_LOWPOWER_REGULATOR_ON（在 stm32h7xx_hal_pwr.h 文件中有定义）
SLEEPEntry	进入睡眠模式的方式 例如：PWR_SLEEPENTRY_WFI、PWR_SLEEPENTRY_WFE（在 stm32h7xx_hal_pwr.h 文件中有定义）

表 32.2.1.1 函数 HAL_PWR_EnterSLEEPMode()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 32.2.1.2 函数 HAL_PWR_EnterSLEEPMode()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 进入睡眠模式 */
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
}
```

32.2.2 PWR 驱动

本章实验的 PWR 驱动主要负责向应用层提供睡眠模式唤醒按键的初始化和进入睡眠模式的函数，同时实现唤醒按键的中断回调函数。本章实验中，PWR 的驱动代码包括 pwr.c 和 pwr.h 两个文件。

PWR 驱动中，睡眠模式唤醒按键的相关宏定义，如下所示：

```
#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_IRQHandler
#define PWR_WKUP_INT_IRQHandler EXTI0_IRQHandler
```

PWR 驱动中，睡眠模式唤醒按键的初始化函数，如下所示：

```
/**
 * @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();

    gpio_init_struct.Pin = PWR_WKUP_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_RISING;
    gpio_init_struct.Pull = GPIO_PULLDOWN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM;
    HAL_GPIO_Init(PWR_WKUP_GPIO_PORT, &gpio_init_struct);

    HAL_NVIC_SetPriority(PWR_WKUP_INT_IRQn, 2, 2);
    HAL_NVIC_EnableIRQ(PWR_WKUP_INT_IRQn);
}
```

因为进入睡眠模式后，能够被任意的中断唤醒，因此睡眠模式唤醒按键的初始化只需要配置好按键的外部中断即可。

PWR 驱动中，睡眠模式唤醒按键对应的中断回调函数，如下所示：

```
/**
 * @brief      唤醒引脚外部中断服务函数
 * @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)
    {
        /* */
    }
}

```

因为 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不做任何事。

3.2.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

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

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480);             /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
    pwr_wkup_key_init();         /* 初始化唤醒按键 */

    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 LED1, 提示进入睡眠模式 */
            pwr_enter_sleep(); /* 进入睡眠模式 */
            HAL_ResumeTick();  /* 恢复滴答时钟 */
            LED1(1);           /* 熄灭 LED1, 提示退出睡眠模式 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE();     /* 每 200ms, 翻转一次 LED0 */
        }

        t++;
        delay_ms(10);
    }
}

```

从上面的代码中可以看出，在完成相关的初始化操作后，并不断地扫描按键，若扫描到 KEY0 按键被按下，则会点亮 LED1 后进入睡眠模式，此时 ARM Cortex-M4 内核便停止了，从该睡眠模式唤醒需要有任意的中断产生，因此可以由 WKUP 按键产生外部中断来唤醒睡眠。退出睡眠模式后，LED1 将被熄灭。

32.3 下载验证

在完成编译和烧录操作后，可以看到 LED0 闪烁提示系统程序正在运行，此时可以按下 KEY0 按键，可以看到 LED1 亮起，但 LED0 不再闪烁，这是因为系统已经进入睡眠模式了，此时再按下 WKUP 按键，即可从睡眠模式下唤醒，可以看到 LED1 熄灭，LED0 继续闪烁。

第三十三章 停止模式实验

本章介绍 STM32H750 低功耗模式中的停止模式，进入停止模式后，所有的时钟都将被停止以降低功耗，在停止模式下可以被任意中断唤醒。通过本章的学习，读者将学习到低功耗模式中停止模式的使用。

本章分为如下几个小节：

- 33.1 硬件设计
- 33.2 程序设计
- 33.3 下载验证

33.1 硬件设计

33.1.1 例程功能

1. TFTLCD 上显示实验信息
2. LED0 闪烁时，按下 KEY0 按键后，LED1 亮起，LED0 停止闪烁
3. LED0 停止闪烁时，按下 WKUP 按键后，LED1 熄灭，LED0 闪烁

33.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15

33.1.3 原理图

本章实验介绍 STM32H750 低功耗模式中的停止模式，不涉及连接原理图。

33.2 程序设计

33.2.1 HAL 库的 PWR 驱动

HAL 库中提供了配置进入停止模式的驱动函数，其配置步骤如下：

①：进入停止模式

在 HAL 库中对应的驱动函数如下：

①：进入停止模式

该函数用于进入停止模式，其函数原型如下所示：

```
void HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry);
```

该函数的形参描述，如下表所示：

形参	描述
Regulator	停止模式下调压器的工作状态 例如：PWR_MAINREGULATOR_ON 和 PWR_LOWPOWER_REGULATOR_ON（在 stm32h7xx_hal_pwr.h 文件中有定义）
STOPEntry	进入停止模式的方式 例如：PWR_STOPENTRY_WFI、PWR_STOPENTRY_WFE（在 stm32xx_hal_pwr.h 文件中有定义）

表 33.2.1.1 函数 HAL_PWR_EnterSTOPMode()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 33.2.1.2 函数 HAL_PWR_EnterSTOPMode()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32f4xx_hal.h"

void example_fun(void)
{
    /* 进入停止模式 */
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}
```

33.2.2 PWR 驱动

本章实验的 PWR 驱动主要负责向应用层提供停止模式唤醒按键的初始化和进入停止模式的函数，同时实现唤醒按键的中断回调函数。本章实验中，PWR 的驱动代码包括 pwr.c 和 pwr.h 两个文件。

因为本章实验中停止模式的唤醒方式与上一章实验中睡眠模式的唤醒方式一致，都是可以由任意的中断唤醒，且均使用了 WKUP 按键的外部中断进行唤醒，因此本章实验中停止模式唤醒按键的相关宏定义、停止模式唤醒按键的初始化及其中断回调函数均与上一章实验一致，因此请读者自行查看上一章中的相关内容。

PWR 驱动中，进入停止模式的函数，如下所示：

```
/***
 * @brief      进入停止模式
 * @param      无
 * @retval     无
 */
void pwr_enter_stop(void)
{
    sys_stm32_clock_init(200, 2, 2, 4); /* 设置时钟, 400Mhz, 降频 */

    /* 当 SVOS3 进入停止模式时，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);
}
```

从上面的代码中可以看出，该函数调用了函数 HAL_PWR_EnterSTOPMode()以 WFI 方式进入停止模式，接下来 MCU 便会进入停止模式，等待任意的中断唤醒，因此在执行此函数前，需要先关闭部分中断，以免误唤醒。

33.2.3 实验应用代码

本章实验的应用代码，如下所示：

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

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
    pwr_wkup_key_init();         /* 初始化唤醒按键 */
}
```

```
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 LED1, 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, 480Mhz */
        sys_stm32_clock_init(240, 2, 2, 4);
        LED1(1);                                /* 熄灭 LED1, 提示退出停止模式 */
    }

    if ((t % 20) == 0)
    {
        LED0_TOGGLE();                         /* 每 200ms, 翻转一次 LED0 */
    }

    t++;
    delay_ms(10);
}
```

从上面的代码中可以看出,在完成相关的初始化操作后,便不断地扫描按键,若扫描到 KEY0 按键被按下,则会点亮 LED1 后进入停止模式,此时所有的时钟将会停止,从该停止模式唤醒需要有任意的中断产生,因此可以由 WKUP 按键产生外部中断来唤醒停止模式。退出停止模式后,LED1 将被熄灭。

33.3 下载验证

在完成编译和烧录操作后,可以看到 LED0 闪烁提示系统正在运行,此时可以按下 KEY0 按键,可以看到 LED1 亮起,但 LED0 不再闪烁,这是因为系统已经进入停止模式了,此时再按下 WKUP 按键,即可从停止模式下唤醒,可以看到 LED1 熄灭,LED0 继续闪烁。

第三十四章 待机模式实验

本章介绍 STM32H750 低功耗模式中的待机模式，进入待机模式后，MCU 内部的电压调压器将断开 1.3V 电源域的电源，这意味着内核和外设都将停止工作，并且内核寄存器和内存中的数据都将丢失，但这也是功耗最低的模式，待机模式下可被 WKUP 引脚的上升沿唤醒。通过本章的学习，读者将学习到低功耗模式下待机模式的使用。

本章分为以下几个小节：

- 34.1 硬件设计
- 34.2 程序设计
- 34.3 下载验证

34.1 硬件设计

34.1.1 例程功能

1. TFTLCD 上显示实验信息
2. LED0 闪烁时，按下 KEY0 按键后，MCU 进入待机模式
3. MCU 进入待机模式后，按下 WKUP 按键，MCU 重新运行

34.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15

34.1.3 原理图

本章实验介绍 STM32H750 低功耗模式中的待机模式，不涉及连接原理图。

34.2 程序设计

34.2.1 HAL 库的 PWR 驱动

本实验配置了 WKUP 按键(连接至 WKUP 引脚)唤醒待机模式，因此具体的配置步骤如下：

①：使能 WKUP 引脚功能

②：进入待机模式

在 HAL 库中对应的驱动函数如下：

①：使能 WKUP 引脚功能

该函数用于使能 WKUP 引脚功能，其函数原型如下所示：

```
void HAL_PWR_EnableWakeUpPin(uint32_t WakeUpPinx);
```

该函数的形参描述，如下表所示：

形参	描述
WakeUpPinx	指定使能的唤醒引脚 例如：PWR_WAKEUP_PIN1 等（在 stm32h7xx_hal_pwr.h 文件中有定义）

表 34.2.1.1 函数 HAL_PWR_EnableWakeUpPin()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 34.2.1.2 函数 HAL_PWR_EnableWakeUpPin()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 使能 WKUP 引脚功能 */
    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);
}
```

②：进入待机模式

该函数用于进入待机模式，其函数原型如下所示：

```
void HAL_PWR_EnterSTANDBYMode(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 34.2.1.3 函数 HAL_PWR_EnterSTANDBYMode()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
无	无

表 34.2.1.4 函数 HAL_PWR_EnterSTANDBYMode()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 进入待机模式 */
    HAL_PWR_EnterSTANDBYMode();
}
```

34.2.2 PWR 驱动

本章实验的 PWR 驱动主要负责向应用层提供进入待机模式的函数。本章实验中，PWR 的驱动代码包括 pwr.c 和 pwr.h 两个文件。

PWR 驱动中，进入待机模式的函数，如下所示：

```
/***
 * @brief      进入待机模式
 * @param      无
 * @retval     无
 */
void pwr_enter_standby(void)
{
    __HAL_GPIO_EXTI_CLEAR_IT(PWR_WKUP_GPIO_PIN);

    HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1_HIGH);

    __HAL_RCC_BACKUPRESET_FORCE();
    HAL_PWR_EnableBkUpAccess();
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
    __HAL_RTC_WRITEPROTECTION_DISABLE(&g_rtc_handle);

    /* 关闭 RTC 相关中断，可能在 RTC 实验打开了 */
    __HAL_RTC_WAKEUPTIMER_DISABLE_IT(&g_rtc_handle, RTC_IT_WUT);
    __HAL_RTC_TIMESTAMP_DISABLE_IT(&g_rtc_handle, RTC_IT_TS);
    __HAL_RTC_ALARM_DISABLE_IT(&g_rtc_handle, RTC_IT_ALRA | RTC_IT_ALRB);

    /* 清除 RTC 相关中断标志位 */
    __HAL_RTC_ALARM_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_ALRAF | RTC_FLAG_ALRBF);
}
```

```

    __HAL_RTC_TIMESTAMP_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_TSF);
    __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_WUTF);

    __HAL_RCC_BACKUPRESET_RELEASE();
    __HAL_RTC_WRITEPROTECTION_ENABLE(&g_rtc_handle);

    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1_HIGH);
    //__HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
    HAL_PWR_EnterSTANDBYMode();
}

```

该函数首先是关闭 RTC 相关中断，清除 RTC 相关中断标志位。然后使能 WKUP 引脚上升沿来唤醒待机模式。最后调用 HAL_PWR_EnterSTANDBYMode 函数进入待机模式。

34.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t t = 0;
    uint8_t key = 0;
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    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 按键进行唤醒。注意待机模式唤醒后，系统会进行复位。

34.3 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，此时 LED0 继续闪烁，说明系统从待机模式中唤醒相当于复位。

第三十五章 DMA 实验

本章介绍 STM32H750 直接存储访问 (DMA) 的使用, DMA 能够在无 CPU 干预的情况下, 实现外设与存储器或存储器与存储器之间数据的高速传输, 从而节省 CPU 资源来做其他操作。通过本章的学习, 读者将学习到 DMA 的使用。

本章分为如下几个小节:

- 35.1 硬件设计
- 35.2 程序设计
- 35.3 下载验证

35.1 硬件设计

35.1.1 例程功能

1. 按下 KEY0 按键后, 串口输出数据
2. LED0 闪烁, 提示程序正在运行

35.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
KEY0 – PA15
4. USART1
USART1_TX - PA9
USART1_RX - PA10
5. DMA2
Stream7 Channel4

35.1.3 原理图

本章实验使用的 DMA 为 STM32F407 的片上资源, 因此没有对应的连接原理图。

35.2 程序设计

35.2.1 HAL 库的 DMA 驱动

在使用 DMA 之前, 需要先根据需求对 DMA 的数据流进行配置, 例如本章实验要使用 DMA 进行串口数据的发送, 因此应将 DMA 的数据流配置为存储器到外设的模式等等。使用 DMA 的具体步骤如下:

①: 初始化 DMA

②: UART DMA 发送数据

在 HAL 库中对应的驱动函数如下:

①: 初始化 DMA

该函数用于初 DMA 始化的各项参数, 其函数原型如下所示:

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

该函数的形参描述, 如下表所示:

形参	描述
hdma	指向 DMA 句柄的指针

表 35.2.1.1 函数 HAL_DMA_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 35.2.1.2 函数 HAL_DMA_Init()返回值描述

该函数需要传入 DMA 的句柄指针，该句柄中就包含了 DMA 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t Channel;           /* 通道 */
    uint32_t Direction;        /* 方向 */
    uint32_t PeriphInc;        /* 外设递增 */
    uint32_t MemInc;           /* 存储器递增 */
    uint32_t PeriphDataAlignment; /* 外设数据格式 */
    uint32_t MemDataAlignment;  /* 存储器数据格式 */
    uint32_t Mode;              /* 模式 */
    uint32_t Priority;          /* 优先级 */
    uint32_t FIFOMode;          /* Fifo 模式 */
    uint32_t FIFOThreshold;     /* Fifo 阈值 */
    uint32_t MemBurst;          /* 存储区突发传输 */
    uint32_t PeriphBurst;       /* 外设突发传输 */
} DMA_InitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    DMA_HandleTypeDef dma_handle = {0};
    /* 初始化 DMA */
    dma_handle.Instance = DMA1_Stream0;
    dma_handle.Init.Channel = DMA_CHANNEL_0;
    dma_handle.Init.Direction = DMA_MEMORY_TO_PERIPH;
    dma_handle.InitPeriphInc = DMA_PINC_DISABLE;
    dma_handle.InitMemInc = DMA_MINC_ENABLE;
    dma_handle.InitPeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    dma_handle.InitMemDataAlignment = DMA_MDATAALIGN_BYTE;
    dma_handle.InitMode = DMA_NORMAL;
    dma_handle.InitPriority = DMA_PRIORITY VERY_HIGH;
    dma_handle.InitFIFOMode = DMA_FIFOMODE_DISABLE;
    dma_handle.InitFIFOThreshold = DMA_FIFO_THRESHOLD_1QUARTERFULL;
    dma_handle.InitMemBurst = DMA_MBURST_SINGLE;
    dma_handle.InitPeriphBurst = DMA_PBURST_SINGLE;
    HAL_DMA_Init(&dma_handle);
}
```

②：UART DMA 发送数据

该函数用于 UART DMA 发送数据，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_UART_Transmit_DMA(  UART_HandleTypeDef *huart,
                                         const uint8_t *pData,
                                         uint16_t Size);
```

该函数的形参描述，如下表所示：

形参	描述
huart	指向 UART 句柄的指针
pData	指向待发送数据的指针
Size	待发送数据的大小

表 35.2.1.3 函数 HAL_UART_Transmit_DMA()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 35.2.1.4 函数 HAL_UART_Transmit_DMA()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    const uint8_t buf[] = {"Hello, World!"};

    /* UART DMA 发送数据 */
    HAL_UART_Transmit_DMA(&uart_handle, buf, sizeof(buf));
}
```

35.2.2 DMA 驱动

本章实验的 DMA 驱动主要负责向应用层提供 DMA 的初始化函数。本章实验中，DMA 的驱动代码包括 dma.c 和 dma.h 两个文件。

DMA 驱动中，DMA 的初始化函数，如下所示：

```
/***
 * @brief      初始化 DMA
 * @note       这里的传输形式是固定的，这点要根据不同的情况来修改
 *             从存储器 -> 外设模式/8 位数据宽度/存储器增量模式
 * @param      dma_stream_handle : DMA 数据流, DMA1_Stream0~7/DMA2_Stream0~7
 * @retval     无
 */
void dma_init(DMA_HandleTypeDef *dma_stream_handle, uint32_t ch)
{
    if ((uint32_t)dma_stream_handle > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();
    }

    __HAL_LINKDMA(&g_uart1_handle, hdmatx, g_dma_handle);

    /* Tx DMA 配置 */
    g_dma_handle.Instance = dma_stream_handle;
    g_dma_handle.Init.Request = ch;
    g_dma_handle.Init.Direction = DMA_MEMORY_TO_PERIPH;
    g_dma_handle.Init.PeriphInc = DMA_PINC_DISABLE;
    g_dma_handle.Init.MemInc = DMA_MINC_ENABLE;
    g_dma_handle.InitPeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    g_dma_handle.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
    g_dma_handle.Init.Mode = DMA_NORMAL;
    g_dma_handle.Init.Priority = DMA_PRIORITY_MEDIUM;
    g_dma_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    g_dma_handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    g_dma_handle.Init.MemBurst = DMA_MBURST_SINGLE;
    g_dma_handle.Init.PeriphBurst = DMA_PBURST_SINGLE;

    HAL_DMA_DeInit(&g_dma_handle);
    HAL_DMA_Init(&g_dma_handle);
}
```

从上面的代码中可以看到，DMA 的初始化函数提供了 DMA 数据流通道、外设地址、存储器地址、传输数据项数据的配置参数，结合这些传入的参数，通过 DMA 初始化函数 HAL_DMA_Init() 初始化了 DMA。

35.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
/* 要循环发送的字符串 */
```

```

const uint8_t TEXT_TO_SEND[]{"正点原子 M100Z-M7 最小系统板 STM32H750 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 */

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */

    dma_init(DMA2_Stream7, DMA_REQUEST_USART1_TX); /* 初始化 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] = 0xa;
                k = 0;
            }
            else
            {
                g_sendbuf[i] = 0xd;
                mask++;
            }
        }
        else /* 复制 TEXT_TO_SEND 语句 */
        {
            mask = 0;
            g_sendbuf[i] = TEXT_TO_SEND[k];
            k++;
        }
    }
    i = 0;
    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES) /* KEY0 按下 */
        {
            printf("\r\nDMA DATA:\r\n");
            lcd_show_string(30, 130, 200, 16, 16, "Start Transimit....", BLUE);
            lcd_show_string(30, 150, 200, 16, 16, "%", BLUE); /* 显示百分号 */
        }
    }
}

```

```
/* 开始一次 DMA 传输! */
HAL_UART_Transmit_DMA(&g_uart1_handle, g_sendbuf, SEND_BUF_SIZE);

/* 等待 DMA 传输完成, 此时我们来做另外一些事情, 比如点灯
 * 实际应用中, 传输数据期间, 可以执行另外的任务 */
while (1)
{
    /* 等待 DMA1_Stream7 传输完成 */
    if (_HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TCIF3_7))
    {
        /* 清除 DMA1_Stream7 传输完成标志 */
        _HAL_DMA_CLEAR_FLAG(&g_dma_handle, DMA_FLAG_TCIF3_7);
        HAL_UART_DMAStop(&g_uart1_handle); /* 传输完成以后关闭串口 DMA */
        break;
    }
    Pro = _HAL_DMA_GET_COUNTER(&g_dma_handle); /* 得到当前还剩余多少个数据 */
    len = SEND_BUF_SIZE; /* 总长度 */
    pro = 1 - (pro / len); /* 得到百分比 */
    pro *= 100; /* 扩大 100 倍 */
    lcd_show_num(30, 150, pro, 3, 16, BLUE);
}
lcd_show_num(30, 150, 100, 3, 16, BLUE); /* 显示 100% */
/* 提示传送完成 */
lcd_show_string(30, 130, 200, 16, 16, "Transimit Finished!", BLUE);
}

i++;
delay_ms(10);

if (i == 20)
{
    LED0_TOGGLE(); /* LED0 闪烁, 提示系统正在运行 */
    i = 0;
}
}
```

main 函数的流程大致是：先初始化发送数据缓冲区 g_sendbuf 的值，然后通过 KEY0 开启串口 DMA 发送，在发送过程中，通过 _HAL_DMA_GET_COUNTER(&g_dma_handle) 获取当前还剩余的数据量来计算传输百分比，最后在传输结束之后清除相应标志位，提示已经传输完成。

35.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验的实验信息，此时若按下 KEY0 键，可以看到串口调试助手不断打印“正点原子 M100Z-M7 最小系统板 STM32H750 DMA 串口实验\r\n”（200 次）。

第三十六章 单通道 ADC 采集实验

本章介绍使用 STM32H750 模数转换器（ADC）进行带通道的电压采集。通过本章的学习，读者将学习到单通道 ADC 的使用。

本章分为如下几个小节：

- 36.1 硬件设计
- 36.2 程序设计
- 36.3 下载验证

36.1 硬件设计

36.1.1 例程功能

1. LCD 上不断刷新显示 PA5 引脚输入电压采样的数字量和模拟量。
2. LED0 闪烁，提示程序正在运行

36.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. ADC1
Channel 19 - PA5

36.1.3 原理图

本章实验使用的 ADC1 为 STM32H750 的片上资源，因此没有对应的连接原理图。

36.2 程序设计

36.2.1 HAL 库的 ADC 驱动

本章实验将使用 ADC1 的通道 1（PA5 引脚）采集外部输入电压的模拟量，并将其转换为数字量，其具体的步骤如下：

- ①：初始化 ADC
- ②：配置 ADC 通道
- ③：开启 ADC
- ④：ADC 轮询转换
- ⑤：获取 ADC 值

在 HAL 库中对应的驱动函数如下：

①：初始化 ADC

该函数用于初始化 ADC，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef* hadc);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针

表 36.2.1.1 函数 HAL_ADC_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 36.2.1.2 函数 HAL_ADC_Init()返回值描述

该函数需要传入 ADC 的句柄指针，该句柄中就包含了 ADC 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct {
    uint32_t ClockPrescaler; /* 设置预分频系数，即 PRESC[3:0]位 */
    uint32_t Resolution; /* 配置 ADC 的分辨率 */
    uint32_t ScanConvMode; /* 扫描模式 */
    uint32_t EOCSelection; /* 转换完成标志位 */
    FunctionalState LowPowerAutoWait; /* 低功耗自动延时 */
    FunctionalState ContinuousConvMode; /* 设置单次转换模式还是连续转换模式 */
    uint32_t NbrOfConversion; /* 设置转换通道数目，赋值范围是 1~16 */
    FunctionalState DiscontinuousConvMode; /* 设置常规转换组不连续模式 */
    uint32_t NbrOfDiscConversion; /* 常规转换组不连续模式转换通道的数目 */
    uint32_t ExternalTrigConv; /* ADC 外部触发源选择 */
    uint32_t ExternalTrigConvEdge; /* ADC 外部触发极性 */
    uint32_t ConversionDataManagement; /* 数据管理 */
    uint32_t Overrun; /* 发生溢出时，进行的操作 */
    uint32_t LeftBitShift; /* 数据左移几位 */
    FunctionalState OversamplingMode; /* 过采样模式 */
    ADC_OversamplingTypeDef Oversampling; /* 过采样的参数配置 */
} ADC_HandleTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    ADC_HandleTypeDef adc_handle = {0};

    /* 初始化 ADC */
    adc_handle.Instance = ADC1;
    adc_handle.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
    adc_handle.Init.Resolution = ADC_RESOLUTION_12B;
    adc_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    adc_handle.Init.ScanConvMode = DISABLE;
    adc_handle.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    adc_handle.Init.ContinuousConvMode = DISABLE;
    adc_handle.Init.NbrOfConversion = 1;
    adc_handle.Init.DiscontinuousConvMode = DISABLE;
    adc_handle.Init.NbrOfDiscConversion = 1;
    adc_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    adc_handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    adc_handle.Init.DMAContinuousRequests = DISABLE;
    HAL_ADC_Init(&adc_handle);
}
```

②：配置 ADC 通道

该函数用于配置 ADC 通道，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef* hadc,
                                         ADC_ChannelConfTypeDef* sConfig);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针
sConfig	指向 ADC 通道配置结构体的指针 需自行定义，并根据 ADC 通道配置参数填充结构体中的成员变量

表 36.2.1.3 函数 HAL_ADC_ConfigChannel()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 36.2.1.4 函数 HAL_ADC_ConfigChannel()返回值描述

该函数使用 ADC_ChannelConfTypeDef 类型结构体指针传入了 ADC 通道的配置参数，该结构体的定义如下所示：

```
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 */
    uint32_t SingleDiff; /* 输入信号线的类型 */
    uint32_t OffsetNumber; /* 采用偏移量的通道 */
    uint32_t Offset; /* 偏移量 */
    FunctionalState OffsetRightShift; /* 数据右移位数 */
    FunctionalState OffsetSignedSaturation; /* 转换数据格式为有符号位数据 */
} ADC_ChannelConfTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    ADC_ChannelConfTypeDef config = {0};

    /* 配置 ADC 通道 */
    config.Channel = ADC_CHANNEL_0;
    config.Rank = 1;
    config.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    config.Offset = 0;
    HAL_ADC_ConfigChannel(&adc_handle, &config);
}
```

③：开启 ADC

该函数用于开启 ADC，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针

表 36.2.1.5 函数 HAL_ADC_Start()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 36.2.1.6 函数 HAL_ADC_Start()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启 ADC */
    HAL_ADC_Start(&adc_handle);
}
```

④：ADC 轮询转换

该函数用于 ADC 轮询转换，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc,
                                            uint32_t Timeout);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针
Timeout	等待轮询超时时间

表 36.2.1.7 函数 HAL_ADC_PollForConversion()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
-----	----

HAL_StatusTypeDef	HAL 状态
-------------------	--------

表 36.2.1.8 函数 HAL_ADC_PollForConversion()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* ADC 轮询转换 */
    HAL_ADC_PollForConversion(&adc_handle, HAL_MAX_DELAY);
}
```

⑤：获取 ADC 值

该函数用于获取 ADC 值，其函数原型如下所示：

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针

表 36.2.1.9 函数 HAL_ADC_GetValue()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
uint32_t 类型数据	ADC 值

表 36.2.1.10 函数 HAL_ADC_GetValue()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint32_t value;

    value = HAL_ADC_GetValue(&adc_handle);

    /* Do something. */
}
```

36.2.2 ADC 驱动

本章实验的 ADC 驱动主要负责向应用层提供 ADC 的初始化和获取 ADC 转换结果的函数。本章实验中，ADC 的驱动代码包括 adc.c 和 adc.h 两个文件。

ADC 驱动中，对 ADC、GPIO 的相关宏定义，如下所示：

```
#define ADC_ADCX_CHY_GPIO_PORT      GPIOA
#define ADC_ADCX_CHY_GPIO_PIN        GPIO_PIN_5
#define ADC_ADCX_CHY_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ADC_ADCX                      ADC1
#define ADC_ADCX_CHY                  ADC_CHANNEL_19
#define ADC_ADCX_CHY_CLK_ENABLE()
do{ __HAL_RCC_ADC12_CLK_ENABLE(); }while(0)
```

ADC 驱动中，ADC 的初始化函数，如下所示：

```
/**
 * @brief      初始化 ADC
 * @note       本函数支持 ADC1/ADC2 任意通道，但是不支持 ADC3
 *             我们使用 16 位精度，ADC 采样时钟=32M，转换时间为：采样周期 + 8.5 个 ADC 周期
 *             设置最大采样周期：810.5，则转换时间 = 819 个 ADC 周期 = 25.6us
 * @param      无
 * @retval     无
 */
void adc_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
```

```

/* 使能时钟 */
ADC_ADCX_CHY_CLK_ENABLE();
ADC_ADCX_CHY_GPIO_CLK_ENABLE();
__HAL_RCC_ADC_CONFIG(RCC_ADCCLKSOURCE_CLKP);

/* 配置 ADC 输入引脚 */
gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_ANALOG;
HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);

/* 配置 ADC */
g_adc_handle.Instance = ADC_ADCX;
g_adc_handle.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV2;
g_adc_handle.Init.Resolution = ADC_RESOLUTION_16B;
g_adc_handle.Init.ScanConvMode = ADC_SCAN_DISABLE;
g_adc_handle.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
g_adc_handle.Init.LowPowerAutoWait = DISABLE;
g_adc_handle.Init.ContinuousConvMode = DISABLE;
g_adc_handle.Init.NbrOfConversion = 1;
g_adc_handle.Init.DiscontinuousConvMode = DISABLE;
g_adc_handle.Init.NbrOfDiscConversion = 0;
g_adc_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
g_adc_handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
g_adc_handle.Init.ConversionDataManagement = ADC_CONVERSIONDATA_DR;
g_adc_handle.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
g_adc_handle.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
g_adc_handle.Init.OversamplingMode = DISABLE;
HAL_ADC_Init(&g_adc_handle);
HAL_ADCEx_Calibration_Start(&g_adc_handle,
                            ADC_CALIB_OFFSET,
                            ADC_SINGLE_ENDED);
}

```

从上面的代码中可以看出，初始化 ADC 就是调用函数 HAL_ADC_Init() 来初始化 ADC。

接下来要介绍的是 adc_get_result 函数，其定义如下：

```

/**
 * @brief      获得 ADC 转换后的结果
 * @param      ch: 通道值 0~19, 取值范围为: ADC_CHANNEL_0~ADC_CHANNEL_19
 * @retval     返回值:转换结果
 */
uint32_t adc_get_result(uint32_t ch)
{
    ADC_ChannelConfTypeDef adc_ch_conf;

    adc_ch_conf.Channel = ch;                                /* 通道 */
    adc_ch_conf.Rank = ADC_REGULAR_RANK_1;                  /* 1 个序列 */
    /* 采样时间, 设置最大采样周期: 810.5 个 ADC 周期 */
    adc_ch_conf.SamplingTime = ADC_SAMPLETIME_810CYCLES_5;
    adc_ch_conf.SingleDiff = ADC_SINGLE_ENDED;              /* 单边采集 */
    adc_ch_conf.OffsetNumber = ADC_OFFSET_NONE;             /* 不使用偏移量的通道 */
    adc_ch_conf.Offset = 0;                                  /* 偏移量为 0 */
    HAL_ADC_ConfigChannel(&g_adc_handle, &adc_ch_conf);   /* 通道配置 */

    HAL_ADC_Start(&g_adc_handle);                          /* 启动 ADC */
    HAL_ADC_PollForConversion(&g_adc_handle, 10);          /* 轮询转换 */
    return HAL_ADC_GetValue(&g_adc_handle); /* 返回最近一次 ADC1 常规组的转换结果 */
}

```

该函数先是调用 HAL_ADC_ConfigChannel 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。

下面介绍的是获取 ADC 某通道的转换多次后的平均值函数，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，0~19
 * @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 */
}
```

该函数用于多次获取 ADC 值，取平均，用来提高准确度。

36.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    uint16_t adpdata;
    float voltage;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480Mhz */
    delay_init(480);            /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    mpu_memory_protection();    /* 保护相关存储区域 */
    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_CH19_VAL:", BLUE);
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH19_VOL:0.000V", BLUE);

    while (1)
    {
        /* 获取通道 19 的转换值，10 次取平均 */
        adpdata = adc_get_result_average(ADC_ADCX_CHY, 10);
        /* 显示 ADC 采样后的原始值 */
        lcd_show_xnum(142, 110, adpdata, 5, 16, 0, BLUE);
        /* 获取计算后的带小数的实际电压值 */
        voltage = (float)adpdata * (3.3 / 65536);
        adpdata = voltage;           /* 赋值整数部分给 adcx 变量 */
        /* 显示电压值的整数部分 */
        lcd_show_xnum(142, 130, adpdata, 1, 16, 0, BLUE);

        voltage -= adpdata;          /* 把已经显示的整数部分去掉，留下小数部分 */
        voltage *= 1000;             /* 小数部分乘以 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(158, 130, voltage, 3, 16, 0x80, BLUE);
    }
}
```

```
    LED0_TOGGLE();  
    delay_ms(100);  
}  
}
```

从上面的代码中可以看出，在进行完包括 ADC 的所有初始化工作后，便不断地获取 ADC2 通道 1 进行 10 次转换后经过均值滤波后的结果，并将该原始值显示在 LCD 上，同时还通过该电压的原始值计算出了电压的模拟量，并在 LCD 上进行显示。

36.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时刷新显示着 ADC1 通道 1 (PA5 引脚) 采集到电压的数字量和模拟量，此时可以通过杜邦线给 PA1 引脚接入不同的电压值（注意共地，且输入电压不能超过 3.3V，否则可能损坏开发板），可以看到 LCD 上显示的电压数字量和模拟量也随之改变。

第三十七章 单通道 ADC 采集（DMA 读取）实验

本章介绍使用 STM32H750 的 DMA 进行单通道的 ADC 采集。通过本章的学习，读者将学习到 DMA、ADC 的使用。

本章分为如下几个小节：

- 37.1 硬件设计
- 37.2 程序设计
- 37.3 下载验证

37.1 硬件设计

37.1.1 例程功能

1. LCD 上不断刷新显示 PA1 引脚输入电压采样的数字量和模拟量。
2. LED0 闪烁，提示程序正在运行

37.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. ADC1
Channel19 - PA5
4. DMA1
Stream7

37.1.3 原理图

本章实验使用的 ADC1 为 STM32H750 的片上资源，因此没有对应的连接原理图。

37.2 程序设计

37.2.1 HAL 库的 ADC 驱动

本章实验中使用 DAM1 Stream7 的外设到存储器模式，将 ADC1 的转换结果保存到指定的存储器中，其具体的步骤如下：

- ①：初始化 ADC
- ②：配置 ADC 通道
- ③：开始 ADC DMA 传输

在 HAL 库中对应的驱动函数如下：

①：初始化 ADC

请见第 35.2.1 小节中初始化 ADC 的相关内容。

②：配置 ADC 通道

请见第 35.2.1 小节中配置 ADC 通道的相关内容。

③：开始 ADC DMA 传输

该函数用于开始 ADC DMA 传输，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_ADC_Start_DMA(    ADC_HandleTypeDef* hadc,
                                         uint32_t* pData,
                                         uint32_t Length);
```

该函数的形参描述，如下表所示：

形参	描述
hadc	指向 ADC 句柄的指针

pData	目标存储器的首地址
Length	传输数据的长度

表 37.2.1.1 函数 HAL_ADC_Start_DMA()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 37.2.1.2 函数 HAL_ADC_Start_DMA()形参描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint16_t buf;

    /* 开始 ADC DMA 传输 */
    HAL_ADC_Start_DMA(&adc_handle, &buf, sizeof(buf) / sizeof(uint16_t));
}
```

37.2.2 ADC 驱动

本章实验的 ADC 驱动主要负责向应用层提供 ADC 初始化和启动 ADC 的 DMA 采集的函数，同时实现 DMA 的中断回调函数。本章实验中，ADC 的驱动代码包括 adc.c 和 adc.h 两个文件。

ADC 驱动中，对 DMA、GPIO、ADC 的相关宏定义，如下所示：

```
#define ADC_ADCX_CHY_GPIO_PORT      GPIOA
#define ADC_ADCX_CHY_GPIO_PIN        GPIO_PIN_5
#define ADC_ADCX_CHY_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ADC_ADCX                      ADC1
#define ADC_ADCX_CHY                  ADC_CHANNEL_19
#define ADC_ADCX_CHY_CLK_ENABLE()     do{ __HAL_RCC_ADC12_CLK_ENABLE(); }while(0)

#define ADC_ADCX_DMASx                DMA1_Stream7
#define ADC_ADCX_DMASx_REQ             DMA_REQUEST_ADC1
#define ADC_ADCX_DMASx_IRQn            DMA1_Stream7_IRQn
#define ADC_ADCX_DMASx_IRQHandler       DMA1_Stream7_IRQHandler

#define ADC_ADCX_DMASx_IS_TC()         ( __HAL_DMA_GET_FLAG(&g_dma_adc_handle,
DMA_FLAG_TCIF3_7) )
#define ADC_ADCX_DMASx_CLR_TC()        do{ __HAL_DMA_CLEAR_FLAG(&g_dma_adc_handle, DMA_FLAG_TCIF3_7); }while(0)
```

ADC 驱动中，ADC 的初始化函数，如下所示：

```
/***
 * @brief      初始化 ADC 和 DMA
 * @param      par: 外设地址
 * @param      mar: 存储器地址
 * @retval     无
 */
void adc_dma_init(uint32_t par, uint32_t mar)
{
    GPIO_InitTypeDef gpio_init_struct;
    ADC_ChannelConfTypeDef adc_ch_conf = {0};

    /* 使能时钟 */
    ADC_ADCX_CHY_GPIO_CLK_ENABLE();
    ADC_ADCX_CHY_CLK_ENABLE();

    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    {
```

```
    __HAL_RCC_DMA2_CLK_ENABLE();
}
else
{
    __HAL_RCC_DMA1_CLK_ENABLE();
}

__HAL_RCC_ADC_CONFIG(RCC_ADCCLKSOURCE_CLKP);

/* 初始化 ADC 采集通道对应的 IO 引脚 */
gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_ANALOG;
HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);

/* 初始化 DMA */
g_dma_adc_handle.Instance = ADC_ADCX_DMASx;
g_dma_adc_handle.Init.Request = ADC_ADCX_DMASx_REQ;
g_dma_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
g_dma_adc_handle.InitPeriphInc = DMA_PINC_DISABLE;
g_dma_adc_handle.InitMemInc = DMA_MINC_ENABLE;
g_dma_adc_handle.InitPeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
g_dma_adc_handle.InitMemDataAlignment = DMA_MDATAALIGN_HALFWORD;
g_dma_adc_handle.Init.Mode = DMA_NORMAL;
g_dma_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM;
g_dma_adc_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
HAL_DMA_Init(&g_dma_adc_handle);

__HAL_LINKDMA(&g_adc_dma_handle, DMA_Handle, g_dma_adc_handle);

/* 初始化 ADC */
g_adc_dma_handle.Instance = ADC_ADCX;
g_adc_dma_handle.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV2;
g_adc_dma_handle.Init.Resolution = ADC_RESOLUTION_16B;
g_adc_dma_handle.Init.ScanConvMode = ADC_SCAN_DISABLE;
g_adc_dma_handle.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
g_adc_dma_handle.Init.LowPowerAutoWait = DISABLE;
g_adc_dma_handle.Init.ContinuousConvMode = ENABLE;
g_adc_dma_handle.Init.NbrOfConversion = 1;
g_adc_dma_handle.Init.DiscontinuousConvMode = DISABLE;
g_adc_dma_handle.Init.NbrOfDiscConversion = 0;
g_adc_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
g_adc_dma_handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONEDGE_NONE;
g_adc_dma_handle.Init.ConversionDataManagement =
ADC_CONVERSIONDATA_DMA_ONESHOT;
g_adc_dma_handle.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
g_adc_dma_handle.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
g_adc_dma_handle.Init.OversamplingMode = DISABLE;
HAL_ADC_Init(&g_adc_dma_handle);
HAL_ADCEx_Calibration_Start(&g_adc_dma_handle, ADC_CALIB_OFFSET,
ADC_SINGLE_ENDED);

/* 配置 ADC 通道 */
adc_ch_conf.Channel = ADC_ADCX_CHY;
adc_ch_conf.Rank = ADC_REGULAR_RANK_1;
adc_ch_conf.SamplingTime = ADC_SAMPLETIME_810CYCLES_5;
adc_ch_conf.SingleDiff = ADC_SINGLE_ENDED;
adc_ch_conf.OffsetNumber = ADC_OFFSET_NONE;
adc_ch_conf.Offset = 0;
adc_ch_conf.OffsetRightShift = DISABLE;
adc_ch_conf.OffsetSignedSaturation = DISABLE;
HAL_ADC_ConfigChannel(&g_adc_dma_handle, &adc_ch_conf);

/* 配置 DMA 数据流请求中断优先级 */
HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);
```

```

    HAL_DMA_Start_IT(&g_dma_adc_handle, par, mar, 0);
    HAL_ADC_Start_DMA(&g_adc_dma_handle, &mar, 0);
}

```

从上面的代码中可以看出，该 ADC 初始化函数初始化了 ADC，同时也使能了 ADC 的 DMA 请求，并且在 HAL 库 ADC 初始化 MSP 函数中配置了 DMA。

ADC 驱动中，启动 ADC 的 DMA 采集的函数，如下所示：

```

/***
 * @brief      使能一次 DMA 传输 ADC 数据
 * @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，也是为了兼容后续实验
 * @param      ndtr: DMA 传输的次数
 * @retval     无
 */
void adc_dma_enable(uint16_t ndtr)
{
    ADC_ADCX->CR &= ~(1 << 0);

    ADC_ADCX_DMASx->CR &= ~(1 << 0);
    while (ADC_ADCX_DMASx->CR & 0x1);
    ADC_ADCX_DMASx->NDTR = ndtr;
    ADC_ADCX_DMASx->CR |= 1 << 0;

    ADC_ADCX->CR |= 1 << 0;
    ADC_ADCX->CR |= 1 << 2;
}

```

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_DMASx->NDTR = ndtr;语句给 DMA_SxNDTR 寄存器写入要传输的数据项数目，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_DMASx_IRQHandler(void)
{
    if (ADC_ADCX_DMASx_IS_TC())
    {
        g_adc_dma_sta = 1;           /* 标记 DMA 传输完成 */
        ADC_ADCX_DMASx_CLR_TC();    /* 清除 DMA1 数据流 7 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：

37.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

#define ADC_DMA_BUF_SIZE 100          /* ADC DMA 缓冲区大小 */
uint16_t g_adc_dma_buf[ADC_DMA_BUF_SIZE]; /* ADC DMA 缓冲区 */
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;
}

```

```

sys_cache_enable();           /* 打开 L1-Cache */
HAL_Init();                  /* 初始化 HAL 库 */
sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
delay_init(480);             /* 初始化延时功能 */
uart_init(115200);           /* 初始化串口 */
mpu_memory_protection();     /* 保护相关存储区域 */
led_init();                  /* 初始化 LED */
lcd_init();                  /* 初始化 LCD */

/* 初始化 ADC DMA 采集 */
adc_dma_init((uint32_t)&ADC1->DR, (uint32_t)&g_adc_dma_buf);
adc_dma_enable(ADC_DMA_BUF_SIZE); /* 启动 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_CH19_VAL:", BLUE);
lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH19_VOL:0.000V", BLUE);

while (1)
{
    if (g_adc_dma_sta == 1)
    {
        SCB_InvalidateDCache();

        sum = 0;
        for (i = 0; i < ADC_DMA_BUF_SIZE; i++)
        {
            sum += g_adc_dma_buf[i];
        }
        adcx= sum / ADC_DMA_BUF_SIZE;

        lcd_show_xnum(142, 110, adcx, 5, 16, 0, BLUE);

        temp = (float)adcx * (3.3 / 65536);
        adcx = temp;
        lcd_show_xnum(142, 130, adcx, 1, 16, 0, BLUE);

        temp -= adcx;
        temp *= 1000;
        lcd_show_xnum(158, 130, temp, 3, 16, 0X80, BLUE);

        g_adc_dma_sta = 0;
        adc_dma_enable(ADC_DMA_BUF_SIZE);
    }

    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。首先我们在液晶固定位置显示了小数点，然后后面计算步骤中，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就在液晶上面显示转换结果的整数和小数部分。

37.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上不断地刷新显示 ADC1 通道 19 (PA5 引脚) 采集到电压的数字量和模拟量，此时可以通过杜邦线给 PA5 引脚接入不同的电压值（注意共地，且输入电压不能超过 3.3V，否则可能损坏开发板），可以看到 LCD 上显示的电压数字量和模拟量也随之改变。

第三十八章 多通道 ADC 采集（DMA 读取）实验

本章介绍 STM32H750 的 DMA 进行多通道的 ADC 采集。通过本章的学习，读者将学习到 DMA、ADC 的使用。

本章分为如下几个小节：

- 38.1 硬件设计
- 38.2 程序设计
- 38.3 下载验证

38.1 硬件设计

38.1.1 例程功能

使用 ADC1 采集（DMA 读取）通道 14\15\16\17\18\19 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。

38.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. ADC1
 - Channel14 - PA2 Channel15 - PA3 Channel16 - PA0
 - Channel17 - PA1 Channel18 - PA4 Channel19 - PA5
4. DMA1
Stream7

38.1.3 原理图

本章实验使用的 ADC1 为 STM32H750 的片上资源，因此没有对应的连接原理图。

38.2 程序设计

38.2.1 HAL 库的 ADC 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。

38.2.2 ADC 驱动

本章实验的 ADC 驱动主要负责向应用层提供 ADC 的初始化和启动 ADC 的 DMA 采集函数，同时实现了 DMA 的中断回调函数。本章实验中，ADC 的驱动代码包括 adc.c 和 adc.h 两个文件。

ADC 驱动中，对 DMA、GPIO、ADC 的相关宏定义，如下所示：

```
#define ADC_ADCX_CHY_GPIO_PORT      GPIOA
#define ADC_ADCX_CHY_GPIO_PIN        GPIO_PIN_5
#define ADC_ADCX_CHY_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)

#define ADC_ADCX_CHY                ADC1
#define ADC_ADCX_CHY_CHANNEL        ADC_CHANNEL_19
#define ADC_ADCX_CHY_CLK_ENABLE()
do{ __HAL_RCC_ADC12_CLK_ENABLE(); }while(0)
```

```
#define ADC_ADCX_DMASx DMA1_Stream7
#define ADC_ADCX_DMASx_REQ DMA_REQUEST_ADC1
#define ADC_ADCX_DMASx IRQn DMA1_Stream7_IRQHandler
#define ADC_ADCX_DMASx_IRQHandler DMA1_Stream7_IRQHandler

#define ADC_ADCX_DMASx_IS_TC() ( __HAL_DMA_GET_FLAG(&g_dma_nch_adc_handle, DMA_FLAG_TCIF3_7) )
#define ADC_ADCX_DMASx_CLR_TC() do{ __HAL_DMA_CLEAR_FLAG(&g_dma_nch_adc_handle, DMA_FLAG_TCIF3_7); }while(0)
```

ADC 驱动中，ADC 的初始化函数，如下所示：

```
/***
 * @brief      初始化 ADC 多通道和 DMA
 * @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的方式来修改通道了，
 *             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） 和 ADC_ADCX_DMASx（默认=DMA1_Stream7） 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      par: 外设地址
 * @param      mar: 存储器地址
 * @retval     无
 */
void adc_nch_dma_init(uint32_t par, uint32_t mar)
{
    GPIO_InitTypeDef gpio_init_struct;
    ADC_ChannelConfTypeDef adc_ch_conf = {0};

    /* 使能时钟 */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    ADC_ADCX_CHY_CLK_ENABLE();

    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();
    }

    __HAL_RCC_ADC_CONFIG(RCC_ADCCLKSOURCE_CLKP);

    /* 配置 ADC1 通道 1/2/3/4/5/6/7 输入引脚 */
    gpio_init_struct.Pin = GPIO_PIN_0 |
                           GPIO_PIN_1 |
                           GPIO_PIN_2 |
                           GPIO_PIN_3 |
                           GPIO_PIN_4 |
                           GPIO_PIN_5;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    /* 配置 DMA */
    g_dma_nch_adc_handle.Instance = ADC_ADCX_DMASx;
    g_dma_nch_adc_handle.Init.Request = ADC_ADCX_DMASx_REQ;
    g_dma_nch_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    g_dma_nch_adc_handle.InitPeriphInc = DMA_PINC_DISABLE;
    g_dma_nch_adc_handle.InitMemInc = DMA_MINC_ENABLE;
    g_dma_nch_adc_handle.InitPeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    g_dma_nch_adc_handle.InitMemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_nch_adc_handle.Init.Mode = DMA_NORMAL;
    g_dma_nch_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM;
```

```
g_dma_nch_adc_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
HAL_DMA_Init(&g_dma_nch_adc_handle);

__HAL_LINKDMA(&g_adc_nch_dma_handle, DMA_Handle, g_dma_nch_adc_handle);

/* 配置 ADC */
g_adc_nch_dma_handle.Instance = ADC_ADCX;
g_adc_nch_dma_handle.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV2;
g_adc_nch_dma_handle.Init.Resolution = ADC_RESOLUTION_16B;
g_adc_nch_dma_handle.Init.ScanConvMode = ADC_SCAN_ENABLE;
g_adc_nch_dma_handle.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
g_adc_nch_dma_handle.Init.LowPowerAutoWait = DISABLE;
g_adc_nch_dma_handle.Init.ContinuousConvMode = ENABLE;
g_adc_nch_dma_handle.Init.NbrOfConversion = 6;
g_adc_nch_dma_handle.Init.DiscontinuousConvMode = DISABLE;
g_adc_nch_dma_handle.Init.NbrOfDiscConversion = 0;
g_adc_nch_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
g_adc_nch_dma_handle.Init.ExternalTrigConvEdge =
ADC_EXTERNALTRIGCONEDGE_NONE;
g_adc_nch_dma_handle.Init.ConversionDataManagement =
ADC_CONVERSIONDATA_DMA_ONESHOT;
g_adc_nch_dma_handle.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
g_adc_nch_dma_handle.Init.LeftBitShift = ADC_LEFTBITSHIFT_NONE;
g_adc_nch_dma_handle.Init.OversamplingMode = DISABLE;
HAL_ADC_Init(&g_adc_nch_dma_handle);

HAL_ADCEx_Calibration_Start(&g_adc_nch_dma_handle, ADC_CALIB_OFFSET,
ADC_SINGLE_ENDED);

/* 配置 ADC 通道 */
adc_ch_conf.Channel = ADC_CHANNEL_14;
adc_ch_conf.Rank = ADC_REGULAR_RANK_1;
adc_ch_conf.SamplingTime = ADC_SAMPLETIME_810CYCLES_5;
adc_ch_conf.SingleDiff = ADC_SINGLE_ENDED;
adc_ch_conf.OffsetNumber = ADC_OFFSET_NONE;
adc_ch_conf.Offset = 0;
adc_ch_conf.OffsetRightShift = DISABLE;
adc_ch_conf.OffsetSignedSaturation = DISABLE;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

adc_ch_conf.Channel = ADC_CHANNEL_15;
adc_ch_conf.Rank = ADC_REGULAR_RANK_2;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

adc_ch_conf.Channel = ADC_CHANNEL_16;
adc_ch_conf.Rank = ADC_REGULAR_RANK_3;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

adc_ch_conf.Channel = ADC_CHANNEL_17;
adc_ch_conf.Rank = ADC_REGULAR_RANK_4;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

adc_ch_conf.Channel = ADC_CHANNEL_18;
adc_ch_conf.Rank = ADC_REGULAR_RANK_5;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

adc_ch_conf.Channel = ADC_CHANNEL_19;
adc_ch_conf.Rank = ADC_REGULAR_RANK_6;
HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &adc_ch_conf);

/* 配置 DMA 数据流请求中断优先级 */
HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 3, 3);
HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);

HAL_DMA_Start_IT(&g_dma_nch_adc_handle, par, mar, 0);
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，请见第 36.2.2 小节中的相关内容。

38.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
#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_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */

    /* 初始化 ADC DMA 采集 */
    adc_nch_dma_init((uint32_t)&ADC1->DR, (uint32_t)&g_adc_dma_buf);

    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, 130, 200, 12, 12, "ADC1_CH14_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 142, 200, 12, 12, "ADC1_CH14_VOL:0.000V", BLUE);

    lcd_show_string(30, 160, 200, 12, 12, "ADC1_CH15_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 172, 200, 12, 12, "ADC1_CH15_VOL:0.000V", BLUE);

    lcd_show_string(30, 190, 200, 12, 12, "ADC1_CH16_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 202, 200, 12, 12, "ADC1_CH16_VOL:0.000V", BLUE);

    lcd_show_string(30, 220, 200, 12, 12, "ADC1_CH17_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 232, 200, 12, 12, "ADC1_CH17_VOL:0.000V", BLUE);

    lcd_show_string(30, 250, 200, 12, 12, "ADC1_CH18_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 262, 200, 12, 12, "ADC1_CH18_VOL:0.000V", BLUE);

    lcd_show_string(30, 280, 200, 12, 12, "ADC1_CH19_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 292, 200, 12, 12, "ADC1_CH19_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)
{
    /* 清除 D Cache 数据 */
    SCB_InvalidateDCache();
    /* 循环显示通道 14~通道 19 的结果 */
    for(j = 0; j < 6; j++) /* 遍历 6 个通道 */
    {
        sum = 0; /* 清零 */
        /* 每个通道采集了 50 次数据, 进行 50 次累加 */
        for (i = 0; i < ADC_DMA_BUF_SIZE / 6; i++)
        {
            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(114, 130 + (j * 30), adcx, 5, 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 / 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(114, 142 + (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(126, 142 + (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 14 的转换数据存放在 `g_adc_dma_buf[0]` 到 `g_adc_dma_buf[49]`，输入通道 15 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。

38.3 下载验证

使用 ADC1 采集 (DMA 读取) 通道 14\15\16\17\18\19 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 到你想测量的电压源 (0~3.3V)。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。

第三十九章 内部温度传感器实验

本章介绍使用 ADC 采集 STM32H750 内部温度传感器输出的电压值，并根据该电压值计算出 STM32H750 芯片的大致温度。通过本章的学习，读者将学习到 ADC、内部温度传感器的使用。

本章分为如下几个小节：

- 39.1 硬件设计
- 39.2 程序设计
- 39.3 下载验证

39.1 硬件设计

39.1.1 例程功能

1. LCD 上不断刷新显示温度
2. LED0 闪烁，提示程序正在运行

39.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. ADC1
Channel16

39.1.3 原理图

本章实验使用的内部温度传感器为 STM32H750 的片上资源，因此没有对应的连接原理图。

39.2 程序设计

39.2.1 HAL 库的 ADC 驱动

本章实验与第三十六章“单通道 ADC 采集实验”十分相似，第三十六章实验中使用 ADC1 的 Channel1 采集 PA5 引脚上的电压，而本章实验是使用 ADC3 的 Channel TEMPSENSOR 采集 STM32H750 片内温度传感器的输出电压，因此大部分的内容都是相似的，以及操作 ADC 的方式都一样，因此请见第 36.2.1 小节中 HAL 库的 ADC 驱动的相关内容。

39.2.2 ADC 驱动

本章实验的 ADC 驱动主要负责向应用层提供 ADC 的初始化以及获取内部温度传感器温度的函数。本章实验中，ADC 的驱动代码包括 adc.c 和 adc.h 两个文件。

因为本章实验的 ADC 驱动与第三十五章“单通道 ADC 采集实验”的 ADC 驱动代码十分相似，因此请参考第 36.2.1 小节 ADC 驱动的相关内容，本小节仅讲解获取内部温度传感器温度的函数，如下所示：

```
/**  
 * @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: °C.)  
 */  
short adc3_get_temperature(void)  
{  
    uint32_t adcx;
```

```

short result;
double temperature;
float temp = 0;
uint16_t ts_call1, ts_call2;

ts_call1 = *(volatile uint16_t *) (0X1FF1E820); /* 获取 TS_CAL1 */
ts_call2 = *(volatile uint16_t *) (0X1FF1E840); /* 获取 TS_CAL2 */
temp = (float) ((110.0 - 30.0) / (ts_call2 - ts_call1)); /* 获取比例因子 */

/* 读取内部温度传感器通道,10 次取平均 */
adcx = adc3_get_result_average(adc3_handle, ADC3_TEMPSENSOR_CHX, 10);
temperature = (float) (temp * (adcx - ts_call1) + 30); /* 计算温度 */

result = temperature *= 100; /* 扩大 100 倍. */
return result;
}

```

从上面的代码中可以看到，在获取温度传感器的输出电压后，需要将电压值转换为实际的温度值，转换公式涉及内部温度传感器的物理特性，由芯片的设计厂家给出。

39.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    short temp;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    adc3_init(); /* 初始化 ADC3(使能内部温度传感器) */

    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 = adc3_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 */
        }
        /* 显示整数部分 */
        lcd_show_xnum(30 + 11 * 8, 120, temp / 100, 2, 16, 0, BLUE);
        /* 显示小数部分 */
        lcd_show_xnum(30 + 14 * 8, 120, temp % 100, 2, 16, 0x80, BLUE);
    }
}

```

```
    LED0_TOGGLE(); /* LED0 闪烁, 提示程序运行 */
    delay_ms(250);
}
```

本章实验的应用代码很简单，在完成包括 ADC 之内的初始化后，便重复通过函数 adc_get_temperature() 获取内部温度传感器采集的温度值，并在 LCD 上实时显示。

39.3 下载验证

在完成编译和烧录操作后，便可看到 LCD 上不断地刷新显示 STM32H750 内部温度传感器采集到的温度值。

第四十章 DAC 输出实验

本章介绍使用 STM32H750 的 DAC 输出指定的电压值。通过本章的学习。读者将学习到 DAC 的使用。

本章分为如下几个小节：

40.1 硬件设计

40.2 程序设计

40.3 下载验证

40.1 硬件设计

40.1.1 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 ADC1 的通道 19 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。

40.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
4. ADC1
Channel19 - PA5
5. DAC
Channel1 - PA4

40.1.3 原理图

本章实验使用的 DAC 为 STM32H750 的片上资源，因此没有对应的连接原理图。

40.2 程序设计

40.2.1 HAL 库的 DAC 驱动

本章实验主要使用 DAC 的 Channel1 (PA4 引脚) 输出指定的电压值 (0V~3.3V)，其具体的步骤如下：

- ①：初始化 DAC
- ②：配置 DAC 通道
- ③：开启 DAC 转换
- ④：设置 DAC 输出值

在 HAL 库中对应的驱动函数如下：

①：初始化 DAC

该函数用于初始化 DAC 的相关参数，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DAC_Init(DAC_HandleTypeDef *hdac);
```

该函数的形参描述，如下表所示：

形参	描述
----	----

hdac	指向 DAC 句柄的指针
------	--------------

表 40.2.1.1 函数 HAL_DAC_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 40.2.1.2 函数 HAL_DAC_Init()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    DAC_HandleTypeDef dac_handle = {0};

    /* 初始化 DAC */
    dac_handle.Instance = DAC;
    HAL_DAC_Init(&dac_handle);
}
```

②：配置 DAC 通道

该函数用于配置 DAC 通道，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(  DAC_HandleTypeDef *hdac,
                                            DAC_ChannelConfTypeDef *sConfig,
                                            uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
hdac	指向 DAC 句柄的指针
sConfig	指向 DAC 通道配置结构体的指针 需自行定义，并根据 DAC 通道配置参数填充结构体中的成员变量
Channel	DAC 通道 例如：DAC_CHANNEL_1 和 DAC_CHANNEL_2（在 stm32h7xx_hal_dac.h 文件中有定义）

表 40.2.1.3 函数 HAL_DAC_ConfigChannel()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 40.2.1.4 函数 HAL_DAC_ConfigChannel()返回值描述

该函数使用 DAC_ChannelConfTypeDef 类型结构体指针传入了 DAC 通道的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t DAC_Trigger;      /* 外部触发源 */
    uint32_t DAC_OutputBuffer; /* 输出缓冲 */
} DAC_ChannelConfTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    DAC_ChannelConfTypeDef config = {0};

    /* 配置 DAC 通道 */
    config.DAC_Trigger = DAC_TRIGGER_SOFTWARE;
    config.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;
    HAL_DAC_ConfigChannel(&dac_handle, &config);
}
```

③：开启 DAC 转换

该函数用于开启 DAC 转换，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

该函数的形参描述，如下表所示：

形参	描述
hdac	指向 DAC 句柄的指针
Channel	DAC 通道 例如：DAC_CHANNEL_1 和 DAC_CHANNEL_2（在 stm32h7xx_hal_dac.h 文件中有定义）

表 40.2.1.5 函数 HAL_DAC_Start()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 40.2.1.6 函数 HAL_DAC_Start()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启 DAC 转换 */
    HAL_DAC_Start(&dac_handle, DAC_CHANNEL_1);
}
```

④：设置 DAC 输出值

该函数用于设置 DAC 输出值，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef *hdac,
                                    uint32_t Channel,
                                    uint32_t Alignment,
                                    uint32_t Data);
```

该函数的形参描述，如下表所示：

形参	描述
hdac	指向 DAC 句柄的指针
Channel	DAC 通道 例如：DAC_CHANNEL_1 和 DAC_CHANNEL_2（在 stm32h7xx_hal_dac.h 文件中有定义）
Alignment	数据对齐方式 例如：DAC_ALIGN_8B_R、DAC_ALIGN_12B_L 等（在 stm32h7xx_hal_dac.h 文件中有定义）
Data	输出值

表 40.2.1.7 函数 HAL_DAC_SetValue()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 40.2.1.8 函数 HAL_DAC_SetValue()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 设置 DAC 输出值 */
    HAL_DAC_SetValue(&dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R, 1000);
}
```

40.2.2 DAC 驱动

本章实验的 DAC 驱动主要负责向应用层提供 DAC 的初始化和配置其输出指定电压的函数。

本章实验中，DAC 的驱动代码包括 dac.c 和 dac.h 两个文件。

DAC 驱动中，DAC 的初始化函数，如下所示：

```
/***
 * @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=120Mhz=8.3ns
 *             DAC 在输出 buffer 关闭的时候，输出建立时间：tSETTLING = 2us (H750 数据手册有写)
 *             因此 DAC 输出的最高速度约为：500Khz，以 10 个点为一个周期，最大能输出 50Khz 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)
{
    DAC_ChannelConfTypeDef dac_ch_conf; /* DAC 通道配置结构体 */
    GPIO_InitTypeDef gpio_init_struct;
    __HAL_RCC_DAC12_CLK_ENABLE(); /* 使能 DAC12 时钟，本芯片只有 DAC1 */
    __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; /* 模拟 */
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    g_dac_handle.Instance = DAC1; /* DAC1 寄存器基址 */
    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 2 */
            break;
        case 2 :
            /* DAC 通道 1 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 2 */
            break;
        default : break;
    }
}
```

从上面的代码中可以看出，DAC 的初始化函数会初始化 DAC、配置 DAC 通道并开启 DAC 转换。

DAC 驱动中，配置 DAC 通道输出指定电压的函数，如下所示：

```
/***
 * @brief      设置 DAC 输出电压
 * @param      voltage: DAC 输出电压(扩大 1000 倍)
 * @retval     无
 */
void dac_set_voltage(uint16_t voltage)
{
    uint16_t value;

    value = (voltage * 4095) / 3300;
    value &= 0xFF;
    HAL_DAC_SetValue(&g_dac_handle, DAC_DACX_CHY, DAC_ALIGN_12B_R, value);
}
```

该函数将输入的电压模拟量转换为 DAC 输出的数字量后，将该值写入指定 DAC 通道的数

据保持寄存器。

40.2.3 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint16_t adcx;
    float temp;
    uint8_t t = 0;
    uint16_t dacval = 0;
    uint8_t key;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    adc_init(); /* 初始化 ADC */
    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, 150, 200, 16, 16, "DAC VAL:", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "DAC VOL:0.000V", BLUE);
    lcd_show_string(30, 190, 200, 16, 16, "ADC VOL:0.000V", BLUE);
    /* 初始值为 0 */
    HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R, 0);
    while (1)
    {
        t++;
        key = key_scan(0); /* 按键扫描 */
        if (key == WKUP_PRES)
        {
            if (dacval < 4000) dacval += 200;
            HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
                             dacval); /* 输出增大 200 */
        }
        else if (key == KEY1_PRES)
        {
            if (dacval > 200) dacval -= 200;
            else dacval = 0;
            HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
                             dacval); /* 输出减少 200 */
        }
        /* 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_dac_handle, 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*/
/* 得到 ADC 通道 19 的转换结果 */
adcx=adc_get_result_average(ADC_ADCX_CHY, 10);
temp = (float)adcx * (3.3 / 65536); /* 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。

40.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时刷新显示着 DAC 输出电压的模拟量和数字量以及 ADC 采集到电压的模拟量，此时可以将 PA4 引脚 (DAC Channel1 输出引脚) 和 PA4 引脚 (ADC Channel1 采集引脚) 通过杜邦线相连，在按下 KEY0 或 WKUP 按键来调整 DAC 通道 1 的输出电压，可以看到 LCD 上显示的 DAC 输出电压的模拟量和数字量以及 ADC 采集到电压的模拟量也随之变化，并且 ADC 采集到电压的模拟量也十分接近 DAC 输出电压的模拟量。

第四十一章 DAC 输出三角波实验

本章将使用软件的方式控制 STM32H750 的 DAC 输出指定幅值、频率和个数的三角波。通过本章的学习，读者将学习到 DAC 的使用。

本章分为如下几个小节：

- 41.1 硬件设计
- 41.2 程序设计
- 41.3 下载验证

41.1 硬件设计

41.1.1 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。

41.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
4. DAC
Channel1 - PA4

41.1.3 原理图

本章实验使用的 DAC 为 STM32H750 的片上资源，因此没有对应的连接原理图。

41.2 程序设计

41.2.1 HAL 库的 DAC 驱动

本章实验与上一章实验十分相似，上一章实验使用按键控制 DAC Channel1 输出“离散”的电压，而本章实验的是使用软件算法控制 DAC Channel1 输出的电压，使之输出一个三角波，因此大部分内容都是相似的，以及操作 DAC 的方式都是一样的，因此请见第 40.2.1 小节中 HAL 库的 DAC 驱动的相关内容。

41.2.2 DAC 驱动

本章实验的 DAC 驱动主要负责向应用层提供 DAC 的初始化以及控制 DAC 输出指定幅值、频率和个数的三角波的函数。本章实验中，DAC 的驱动代码包括 dac.c 和 dac.h 两个文件。

本章实验 DAC 驱动中对 DAC 的初始化与上一章实验中对 DAC 的初始化方式一致，请见第 40.2.2 小节中 DAC 初始化的相关内容提供。本小节仅介绍通过软件控制 DAC 输出三角波的函数，如下所示：

```
/**  
 * @brief      设置 DAC_OUT1 输出三角波  
 * @note       输出频率 ≈ 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, samples 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++)
    {
        /* 先输出 0 */
        HAL_DAC_SetValue(&g_dac_handle, 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 */
            /* 用寄存器操作波形会更稳定 */
            HAL_DAC_SetValue(&g_dac_handle, 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 */
            /* 用寄存器操作波形会更稳定 */
            HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R, Curval);
            delay_us(dt);
        }
    }
}
}

```

可以看到, 该函数就是每间隔一段时间就修改一次 DAC 的输出电压, 以控制 DAC 输出指定的三角波, 对于该函数的实现, 读者无需深究, 仅需会使用该函数即可。

41.2.3 实验应用代码

本章实验的应用代码, 如下所示:

```

int main(void)
{
    uint8_t t = 0;
    uint8_t key;
    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
    delay_init(480);             /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    usmart_dev.init(240);         /* 初始化 USMART */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
    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 0.1Khz 波形 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "DAC Wave1 ", BLUE);
        /* 幅值 4095, 采样点间隔 5us, 2000 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 0.1Khz 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 $\approx 1000/(dt * samples)$ KHz，得到三角波 1 和三角波 2 的频率都是 0.1KHz。

41.3 下载验证

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。

第四十二章 DAC 输出正弦波实验

本章将使用 TIM7 的更新事件用于 TRGO 触发 DAC Channel1 输出 DMA2 Stream6 从使用软件算法生成的正弦波数据，以输出正弦波。通过本章的学习，读者将学习到 DAC、TIM、DMA 的使用。

本章分为如下几个小节：

- 42.1 硬件设计
- 42.2 程序设计
- 42.3 下载验证

42.1 硬件设计

42.1.1 例程功能

1. 按下 WKUP 按键，PA4 引脚输出正弦波 1（幅值 3.3V，频率 1KHz，采样点 100）
2. 按下 KEY0 按键，PA4 引脚输出正弦波 2（幅值 3.3V，频率 10KHz，采样点 10）
3. LED0 闪烁，提示程序正在运行

42.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
4. DAC
Channel1 - PA4
5. DMA1
Stream6
6. TIM7

42.1.3 原理图

本章实验使用的 DAC 为 STM32H750 的片上资源，因此没有对应的连接原理图。

42.2 程序设计

42.2.1 HAL 库的 TIM 驱动

本章实验中将使用 TIM7 的更新事件产生 TRGO 事件，该 TRGO 事件将用于触发 DAC，其具体的步骤如下：

- ①：初始化 TIM
- ②：配置 TIM 主模式

在 HAL 库中对应的驱动函数如下：

①：初始化 TIM

请见第 16.2.1 小节中初始化 TIM 的相关章节。

②：配置 TIM 主模式

该函数用于配置 TIM 的主模式，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_TIMEx_MasterConfigSynchronization(
    TIM_HandleTypeDef *htim,
    TIM_MasterConfigTypeDef *sMasterConfig);
```

该函数的形参描述，如下表所示：

形参	描述
htim	指向 TIM 句柄的指针
sMasterConfig	指向 TIM 主模式配置结构体的指针 需自行定义，并根据 TIM 主模式配置参数填充结构体中的成员变量

表 42.2.1.1 函数 HAL_TIMEx_MasterConfigSynchronization()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 42.2.1.2 函数 HAL_TIMEx_MasterConfigSynchronization()返回值描述

该函数使用 TIM_MasterConfigTypeDef 类型结构体指针传入了 TIM 主模式的配置参数，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t MasterOutputTrigger; /* 触发输出选择 */
    uint32_t MasterSlaveMode; /* 主从模式 */
} TIM_MasterConfigTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    TIM_MasterConfigTypeDef config = {0};

    /* 配置 TIM 主模式 */
    config.MasterOutputTrigger = TIM_TRGO_RESET;
    config.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
    HAL_TIMEx_MasterConfigSynchronization(&tim_handle, &config);
}
```

42.2.2 HAL 库的 DAC 驱动

本章实验与第四十一章一样，使用 DAC 通道 1 (PA4 引脚) 输出电压，不同之处在于，本章使用 DMA 自动将 DAC 通道 1 待输出的数据写入 DAC 的数据保持寄存器，以输出正弦波。其中对 DAC 的操作请见第四十章中 HAL 库的 DAC 驱动的相关内容，本小节仅介绍 DAC 使用 DMA 的相关步骤，其具体的步骤如下：

①：开启 DAC DMA 转换

在 HAL 库中对应的驱动函数如下：

①：开启 DAC DMA 转换

该函数用于开启 DAC DMA 转换，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DAC_Start_DMA(DAC_HandleTypeDef *hdac,
                                     uint32_t Channel,
                                     uint32_t *pData,
                                     uint32_t Length,
                                     uint32_t Alignment);
```

该函数的形参描述，如下表所示：

形参	描述
hdac	指向 DAC 句柄的指针
Channel	DAC 通道 例如：DAC_CHANNEL_1 和 DAC_CHANNEL_2 (在 stm32h7xx_hal_dac.h 文件中有定义)
pData	数据首地址
Length	数据长度
Alignment	数据对齐方式

	例如：DAC_ALIGN_8B_R、DAC_ALIGN_12B_L 等（在stm32h7xx_hal_dac.h文件中有定义）
--	---

表 42.2.2.1 函数()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 42.2.2.2 函数()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint16_t buf;

    /* 开启 DAC DMA 传输 */
    HAL_DAC_Start_DMA(&dac_handle,
    DAC_CHANNEL_1,
    buf,
    sizeof(buf) / sizeof(buf),
    DAC_ALIGN_12B_R);
}
```

42.2.3 DAC 驱动

本章实验的 DAC 驱动主要负责向应用层提供 DAC 的初始化和使能 DAC 输出指定幅值、频率的正弦波。本章实验中，DAC 的驱动代码包括 dac.c 和 dac.h 两个文件。

DAC 驱动中，DAC 的初始化函数，如下所示：

```
/**
 * @brief      初始化 DAC 输出波形
 * @note       DAC 的输入时钟来自 APB1，时钟频率=120Mhz=8.3ns
 *             DAC 在输出 buffer 关闭的时候，输出建立时间：tSETTLING = 2us (H750 数据手册有
 * 写)
 *             因此 DAC 输出的最高速度约为:500Khz，以 10 个点为一个周期，最大能输出 50Khz 左右
 * 的波形
 *
 * @retval     无
 */
void dac_dma_wave_init(uint8_t outx)
{
    DAC_ChannelConfTypeDef dac_ch_conf={0};
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_DAC12_CLK_ENABLE();
    __HAL_RCC_DMA2_CLK_ENABLE();

    /* 配置 DAC 输出引脚 */
    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);

    /* 配置 DMA */
    g_dma_dac_handle.Instance = DMA2_Stream6;
    g_dma_dac_handle.Init.Request = DMA_REQUEST_DAC1_CH1;
    g_dma_dac_handle.Init.Direction = DMA_MEMORY_TO_PERIPH;
    g_dma_dac_handle.Init.PeriphInc = DMA_PINC_DISABLE;
    g_dma_dac_handle.Init.MemInc = DMA_MINC_ENABLE;
    g_dma_dac_handle.InitPeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    g_dma_dac_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_dac_handle.Init.Mode = DMA_CIRCULAR;
```

```

g_dma_dac_handle.Init.Priority = DMA_PRIORITY_MEDIUM;
g_dma_dac_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
HAL_DMA_Init(&g_dma_dac_handle);
__HAL_LINKDMA(&g_dac_dma_handle, DMA_Handle1, g_dma_dac_handle);

/* 配置 DAC */
g_dac_dma_handle.Instance = DAC1;
HAL_DAC_Init(&g_dac_dma_handle);
dac_ch_conf.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
dac_ch_conf.DAC_Trigger = DAC_TRIGGER_T7_TRGO;
dac_ch_conf.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
dac_ch_conf.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
dac_ch_conf.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
HAL_DAC_ConfigChannel(&g_dac_dma_handle, &dac_ch_conf, DAC_CHANNEL_1);
}

```

可以看到本章实验 DAC 驱动中的 DAC 初始化函数与第四十章“DAC 输出实验”中对 DAC 的初始化基本一致，不过本章配置 DAC Channel1 的触发源为 TIM7 的 TRGO 事件，同时还配置了 DMA。

DAC 驱动中使能 DAC 输出指定幅值、频率的正弦波的函数，如下所示：

```

/***
 * @brief      使能 DAC 输出波形
 * @note       TIM7 的输入时钟频率(f)来自 APB1, f = 120M * 2 = 240Mhz.
 *             DAC 触发频率 ftrgo = f / ((psc + 1) * (arr + 1))
 *             波形频率 = ftrgo / ndtr;
 *
 * @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 ndtr, uint16_t arr, uint16_t psc)
{
    TIM_HandleTypeDef tim7_handle = {0};
    TIM_MasterConfigTypeDef master_config = {0};

    /* 使能时钟 */
    __HAL_RCC_TIM7_CLK_ENABLE();

    /* 配置 TIM7 */
    tim7_handle.Instance = TIM7;
    tim7_handle.Init.Prescaler = psc;
    tim7_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    tim7_handle.Init.Period = arr;
    tim7_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    HAL_TIM_Base_Init(&tim7_handle);
    master_config.MasterOutputTrigger = TIM_TRGO_UPDATE;
    master_config.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&tim7_handle, &master_config);
    HAL_TIM_Base_Start(&tim7_handle);

    /* 配置 DAC 和 DMA */
    HAL_DAC_Stop_DMA(&g_dac_dma_handle, DAC_CHANNEL_1);
    HAL_DAC_Start_DMA(&g_dac_dma_handle, DAC_CHANNEL_1, (uint32_t
*)g_dac_sin_buf, ndtr, DAC_ALIGN_12B_R);
}

```

该函数配置了 TIM7 的溢出频率，因此 TIM7 的溢出频率就决定了 DAC 输出正弦波的频率，同时也开启了 DAC DMA 传输。

42.2.4 实验应用代码

本章实验的应用代码，如下所示：

```
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 * 3.1415926) / samples; /* ω=2π/T*/
    float outdata = 0;

    for (i = 0; i < samples; i++)
    {
        outdata = maxval * (1 + sin(inc * i)); /* y=Asin(ωx+φ)+b */

        if (outdata > 4095)
        {
            outdata = 4095; /* 上限限定 */
        }
        g_dac_sin_buf[i] = outdata;
    }
}

/***
 * @brief      通过 USMART 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)
{
    uint8_t key;
    uint8_t t = 0;
    uint16_t dacdata;
    uint16_t dac_voltage;
    uint16_t adcdata;
    uint16_t adc_voltage;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    adc_init(); /* 初始化 ADC */
    dac_dma_wave_init(1); /* 初始化 DAC 通道 1 DMA 波形输出 */
    dac_creat_sin_buf(2048, 100); /* 生成正弦数据, 振幅约 3.3(V), 100 个数据 */
    /* 定时器触发速率 100KHz, 100 个数据, 输出约 1KHz 的正弦波 */
    dac_dma_wave_enable(100, 100 - 1, 24 - 1);
}
```

```

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, "KEY0:5Khz WK_UP:50Khz", 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 == KEY0_PRES)                            /* 高采样率，约 5Khz 波形 */
    {
        dac_creat_sin_buf(2048, 100);                /* 产生正弦波函序列 */
        /* 500Khz 触发频率，100 个点，得到最高 5Khz 的正弦波。 */
        dac_dma_wave_enable(100, 20 - 1, 24 - 1);
    }
    else if (key == WKUP_PRES)                      /* 低采样率，约 50Khz 波形 */
    {
        dac_creat_sin_buf(2048, 10);                 /* 产生正弦波函序列 */
        /* 500Khz 触发频率，10 个点，可以得到最高 50Khz 的正弦波。 */
        dac_dma_wave_enable(10, 20 - 1, 24 - 1);
    }

    dacdata = DAC1->DHR12R1;                      /* 获取 DAC1_OUT1 的输出状态 */
    lcd_show_xnum(94, 130, dacdata, 5, 16, 0, BLUE);

    dac_voltage = (dacdata * 3300) / 4095;           /* 计算并显示 DAC 输出电压的模拟量 */
    lcd_show_xnum(94, 150, dac_voltage / 1000, 1, 16, 0, BLUE);
    lcd_show_xnum(110, 150, dac_voltage % 1000, 3, 16, 0x80, BLUE);
    /* 得到 ADC 通道 19 的转换结果 */
    adpdata = adc_get_result_average(ADC_ADCX_CHY, 20);
    adc_voltage = (adpdata * 3300) / 65535;
    lcd_show_xnum(94, 170, adc_voltage / 1000, 1, 16, 0, BLUE);
    lcd_show_xnum(110, 170, adc_voltage % 1000, 3, 16, 0x80, BLUE);

    if (t == 10)                                    /* 定时时间到了 */
    {
        LED0_TOGGLE();                            /* LED0 闪烁 */
        t = 0;
    }

    delay_ms(5);
}
}

```

可以看到，在实验应用代码中，定义了函数 `dac_creat_sin_buf()`，该函数用于生成正弦波数据，并保存至数组 `g_dac_sin_buf` 中。在完成 DAC 初始化后，便生成了一组正弦波数据，并调用函数 `dac_dma_wave_enable()` 使能 DAC Channel1 输出指定的正弦波，随后便通过扫描到的不同按键值，输出不同频率的正弦波。

42.3 下载验证

在完成编译和烧录操作后，便可分别按下 KEY0 按键和 WKUP 按键控制 DAC Channel1 输出不同类型的正弦波，DAC Channel1 输出的正弦波可通过示波器观察 PA4 引脚（DAC Channel1 输出引脚）看到。

第四十三章 IIC 实验

本章将介绍使用 STM32H750 驱动板载的 EEPROM 进行读写操作。通过本章的学习，读者将学习到使用 GPIO 模拟 IIC 时序以及 EEPROM 的驱动。

本章分为如下几个小节：

- 43.1 硬件设计
- 43.2 程序设计
- 43.3 下载验证

43.1 硬件设计

43.1.1 例程功能

1. 按下 WKUP 按键可往 24C02 写入 “STM32 IIC TEST”
2. 按下 KEY0 按键可从 24C02 读取到 “STM32 IIC TEST”，并显示至 LCD
3. LED0 闪烁，提示程序正在运行

43.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
4. 24C02
IIC_SCL - PB10
IIC_SDA - PB11

43.1.3 原理图

本章实验使用了板载的 24C02 芯片，该芯片是一个 EEPROM，MCU 是通过两个 GPIO 与该 EEPROM 进行连接与通信的，该 EEPROM 与 MCU 的连接原理图，如下图所示：

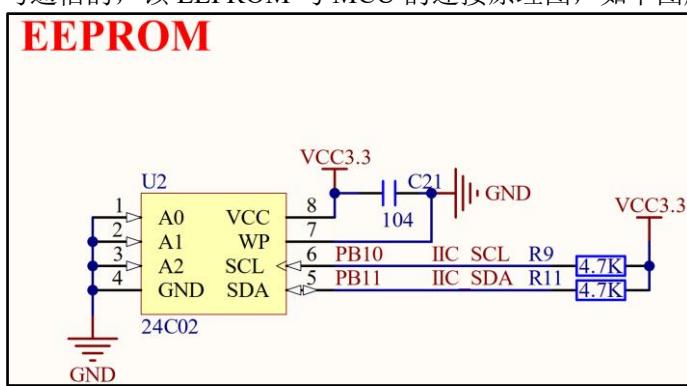


图 43.1.3.1 EEPROM 与 MCU 的连接原理图

43.2 程序设计

43.2.1 HAL 库的 GPIO 驱动

本章实验通过控制与 EEPROM 连接的 GPIO 模拟 IIC 时序，实现 EEPROM 的读写操作，对于 GPIO，主要涉及 GPIO 的配置和读写操作，需要一下几个步骤：

- ①: 配置 GPIO 引脚为通用输出模式、开漏输出和禁止上拉/下拉
- ②: 在与 EEPROM 通信时, 根据需求控制指定 GPIO 引脚输出指定电平
- ③: 在与 EEPROM 通信时, 根据需求读取指定 GPIO 引脚上的输入电平
在 HAL 库中对应的驱动函数如下:

①: 配置 GPIO 引脚

请见第 10.2.1 小节中配置 GPIO 引脚的相关内容。

②: 设置 GPIO 引脚输出电平

请见第 10.2.1 小节中设置 GPIO 引脚输出电平的相关内容。

③: 读取 GPIO 引脚输入电平

请见第 11.2.1 小节中读取 GPIO 引脚输入电平的相关内容。

43.2.2 IIC 驱动

本章实验使用的 IIC 的软件模拟的 IIC, 即控制 GPIO 模拟 IIC 的时序与外部器件进行通信。IIC 驱动主要负责向 EEPROM 驱动提供 IIC 操作的各种函数, 例如: IIC 起始信号、IIC 停止信号等。本章实验中, IIC 的驱动代码包括 myiic.c 和 myiic.h 两个文件。

IIC 驱动中, 对 GPIO 相关的宏定义, 如下所示:

```
#define IIC_SCL_GPIO_PORT      GPIOB
#define IIC_SCL_GPIO_PIN        GPIO_PIN_10
#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_11
#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)

#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)

#define IIC_READ_SDA  HAL_GPIO_ReadPin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN)
```

IIC 驱动中, IIC 的初始化函数, 如下所示:

```
/***
 * @brief      初始化 IIC
 * @param      无
 * @retval     无
 */
void iic_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    IIC_SCL_GPIO_CLK_ENABLE();
    IIC_SDA_GPIO_CLK_ENABLE();

    /* 配置 IIC SCL 引脚 */
    gpio_init_struct.Pin = IIC_SCL_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(IIC_SCL_GPIO_PORT, &gpio_init_struct);
```

```

/* 配置 IIC SDA 引脚 */
gpio_init_struct.Pin = IIC_SDA_GPIO_PIN;
gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD;
gpio_init_struct.Pull = GPIO_PULLUP;
gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
HAL_GPIO_Init(IIC_SDA_GPIO_PORT, &gpio_init_struct);

iic_stop();
}

```

可以看到，IIC 的初始化实际上就是配置 IIC 时钟与数据信号对应的 GPIO 引脚。

IIC 驱动中对 IIC 的各种操作，例如产生 IIC 起始信号、产生 IIC 停止信号等，请读者结合 IIC 的时序规定查看本实验的配套实验源码。

43.2.3 EEPROM 驱动

本章实验的 EEPROM 驱动主要负责向应用层提供 EEPROM 的初始化和读写数据等操作函数。本章实验中，EEPROM 的驱动代码包括 24cxx.c 和 24cxx.h 两个文件。

EEPROM 驱动中，EEPROM 的初始化函数，如下所示：

```

/**
 * @brief    初始化 AT24CXX
 * @param    无
 * @retval   无
 */
void at24cxx_init(void)
{
    iic_init();
}

```

可以看到，该函数实际上就是初始化了与 EEPROM 通讯的 IIC。

EEPROM 驱动中的其他对 EEPROM 的操作函数，例如：EEPROM 的读写函数，请读者结合 24C02 EEPROM 芯片的数据手册查看本实验的配套实验源码。

43.2.4 实验应用代码

本章实验的应用代码，如下所示：

```

static const uint8_t g_text_buf[] = {"STM32 IIC TEST"};
#define TEXT_SIZE sizeof(g_text_buf)

int main(void)
{
    uint8_t key;
    uint8_t t = 0;
    uint8_t datatemp [TEXT_SIZE];

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
    at24cxx_init();              /* 初始化 AT24CXX */

    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, "WK_UP:Write KEY0:Read", RED);

    /* 检测 AT24CXX 是否正常 */
    while (at24cxx_check())

```

```
{  
    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();  
}  
lcd_show_string(30, 130, 200, 16, 16, "24C02 Ready!      ", RED);  
  
while (1)  
{  
    t++;  
    key = key_scan(0);  
  
    if (key == WKUP_PRES)  
    {  
        /* 往 AT24CXX 写入数据 */  
        lcd_fill(0, 150, 239, 319, WHITE);  
        lcd_show_string(30, 150, 200, 16, 16, "Start Write 24C02...", BLUE);  
        at24cxx_write(0, (uint8_t *)g_text_buf, TEXT_SIZE);  
        lcd_show_string(30, 150, 200, 16, 16, "24C02 Write Finished!", BLUE);  
    }  
    else if (key == KEY0_PRES)  
    {  
        /* 从 AT24CXX 读取数据 */  
        lcd_show_string(30, 150, 200, 16, 16, "Start Read 24C02... ", BLUE);  
        at24cxx_read(0, datatemp, 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);  
    }  
  
    if (t == 20)  
    {  
        LED0_TOGGLE();  
        t = 0;  
    }  
  
    delay_ms(10);  
}  
}
```

从本章实验的应用代码中可以看到，在初始化完 EEPROM 后，会检测与 EEPROM 的连接是否正常，若与 EEPROM 的连接正常，则会不断地等待按键输入，若检测到 WKUP 按键被按下，则会往 EEPROM 的指定地址中写入指定的数据，若检测到 KEY_0 按键被按下，则会从 EEPROM 的指定地址中读取数据，并在 LCD 上进行显示。

43.3 下载验证

在完成编译和烧录操作后，若 MCU 与 EEPROM 的连接无误，则可以在 LCD 上看到“24C02 Ready!”的提示信息，此时可以按下 WKUP 按键往 EEPROM 的指定地址写入指定数据，然后再按下 KEY_0 按键从 EEPROM 的指定地址将写入的数据读回来在 LCD 上进行显示，此时便可以看到在 LCD 上显示了“STM32 IIC TEST”的提示信息，该提示信息就是从 EEPROM 中读回的数据。

第四十四章 SPI 实验

本章将介绍使用 STM32H750 驱动板载的 NOR Flash 进行读写操作。通过本章的学习，读者将学习到使用 SPI 驱动 NOR Flash 的使用。

本章分为如下几个小节：

- 44.1 硬件设计
- 44.2 程序设计
- 44.3 下载验证

44.1 硬件设计

44.1.1 例程功能

1. 按下 WKUP 按键可往 NOR Flash 写入“STM32 SPI TEST”
2. 按下 KEY0 按键可从 NOR Flash 读取到“STM32 SPI TEST”，并显示至 LCD
3. LED0 闪烁，提示程序正在运行

44.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
5. QSPI
QSPI_BK1_CLK - PB2
QSPI_BK1_NCS - PB6
QSPI_BK1_IO0 - PD11
QSPI_BK1_IO1 - PD12
QSPI_BK1_IO2 - PE2
QSPI_BK1_IO3 - PD13
6. norflash(QSPI FLASH 芯片,连接在 QSPI 接口上)

44.1.3 原理图

板载的 QSPI FLASH 芯片与 STM32H750 的连接关系，如下图所示：

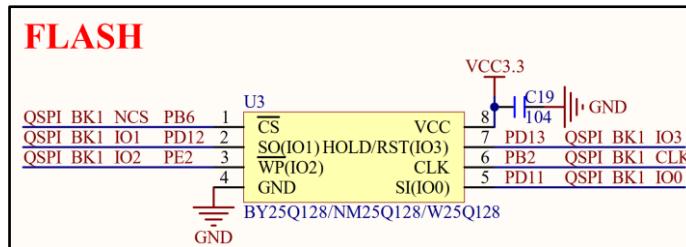


图 44.1.3.1 NOR Flash 与 MCU 的连接原理图

44.2 程序设计

44.2.1 HAL 库的 SPI 驱动

本章实验支持多种型号的 QSPI FLASH 芯片，比如：BY25Q128/NM25Q128/W25Q128 等等，其具体的步骤如下所示：

①：初始化 QSPI

②：QSPI 发送并接收数据

在 HAL 库中对应的驱动函数如下：

①：初始化 QSPI

该函数用于初始化 QSPI，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_QSPI_Init(QSPI_HandleTypeDef *hqspi);
```

该函数的形参描述，如下表所示：

形参	描述
hqspi	指向 QSPI 句柄的指针

表 44.2.1.1 函数 HAL_QSPI_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 44.2.1.2 函数 HAL_QSPI_Init()返回值描述

该函数需要传入 QSPI 的句柄指针，该句柄中就包含了 QSPI 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct
{
    QUADSPI_TypeDef            *Instance;          /* QSPI 寄存器基址 */
    QSPI_InitTypeDef           Init;              /* QSPI 参数配置结构体 */
    uint8_t                     *pTxBuffPtr;        /* 要发送数据的地址 */
    __IO uint16_t               TxXferSize;        /* 要发送数据的大小 */
    __IO uint16_t               TxXferCount;       /* 剩余要发送数据的个数 */
    uint8_t                     *pRxBuffPtr;        /* 要接收数据的地址 */
    __IO uint16_t               RxXferSize;        /* 要接收数据的大小 */
    __IO uint16_t               RxXferCount;       /* 剩余要接收数据的个数 */
    DMA_HandleTypeDef           *hmdma;            /* DMA 配置结构体 */
    __IO HAL_LockTypeDef        Lock;              /* 锁对象 */
    __IO HAL_QSPI_StateTypeDef State;             /* QSPI 通信状态 */
    __IO uint32_t                ErrorCode;          /* 错误代码 */
    uint32_t                     Timeout;            /* 配置 QSPI 内存访问超时时间 */
} QSPI_HandleTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    g_qspi_handle.Instance = QUADSPI;
    g_qspi_handle.Init.ClockPrescaler = 1;
    g_qspi_handle.Init.FifoThreshold = 4;
    g_qspi_handle.Init.SampleShifting = QSPI_SAMPLE_SHIFTING_HALFCYCLE;
    g_qspi_handle.Init.FlashSize = 25-1;
    g_qspi_handle.Init.ChipSelectHighTime = QSPI_CS_HIGH_TIME_3_CYCLE;
    g_qspi_handle.Init.ClockMode = QSPI_CLOCK_MODE_3;
    g_qspi_handle.Init.FlashID = QSPI_FLASH_ID_1;
    g_qspi_handle.Init.DualFlash = QSPI_DUALFLASH_DISABLE;

    if(HAL_QSPI_Init(&g_qspi_handle) == HAL_OK)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
```

②：HAL_QSPI_Command 函数

QSPI 设置命令配置函数，其声明如下：

```
HAL_StatusTypeDef HAL_QSPI_Command(QSPI_HandleTypeDef *hqspi,
                                    QSPI_CommandTypeDef *cmd, uint32_t Timeout);
```

该函数的形参描述，如下表所示：

形参	描述
hqspi	指向 QSPI 句柄的指针
cmd	指向 QSPI 指令配置信息的指针
Timeout	超时等待时间

表 44.2.1.3 函数 HAL_QSPI_Command ()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 44.2.1.4 函数 HAL_QSPI_Command ()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    QSPI_CommandTypeDef qspi_command_handle;
    QSPI_HandleTypeDef g_qspi_handle;

    /* QSPI 发送并接收数据 */
    HAL_QSPI_Command(&g_qspi_handle, &qspi_command_handle, 5000);
}
```

44.2.2 QSPI 驱动

本章实验的 QSPI 驱动主要负责向 NOR Flash 驱动提供 QSPI 的各种操作函数，例如：SPI 初始化、QSPI 读写等。本章实验中，QSPI 的驱动代码包括 qspi.c 和 qspi.h 两个文件。

QSPI 驱动中，对 QSPI、GPIO 相关的宏定义，如下所示：

```
#define QSPI_BK1_CLK_GPIO_PORT      GPIOB
#define QSPI_BK1_CLK_GPIO_PIN        GPIO_PIN_2
#define QSPI_BK1_CLK_GPIO_AF         GPIO_AF9_QUADSPI
#define QSPI_BK1_CLK_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define QSPI_BK1_NCS_GPIO_PORT       GPIOB
#define QSPI_BK1_NCS_GPIO_PIN        GPIO_PIN_6
#define QSPI_BK1_NCS_GPIO_AF         GPIO_AF10_QUADSPI
#define QSPI_BK1_NCS_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define QSPI_BK1_IO0_GPIO_PORT       GPIOD
#define QSPI_BK1_IO0_GPIO_PIN        GPIO_PIN_11
#define QSPI_BK1_IO0_GPIO_AF         GPIO_AF9_QUADSPI
#define QSPI_BK1_IO0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

#define QSPI_BK1_IO1_GPIO_PORT       GPIOD
#define QSPI_BK1_IO1_GPIO_PIN        GPIO_PIN_12
#define QSPI_BK1_IO1_GPIO_AF         GPIO_AF9_QUADSPI
#define QSPI_BK1_IO1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

#define QSPI_BK1_IO2_GPIO_PORT       GPIOE
#define QSPI_BK1_IO2_GPIO_PIN        GPIO_PIN_2
#define QSPI_BK1_IO2_GPIO_AF         GPIO_AF9_QUADSPI
#define QSPI_BK1_IO2_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define QSPI_BK1_IO3_GPIO_PORT       GPIOD
#define QSPI_BK1_IO3_GPIO_PIN        GPIO_PIN_13
```

```
#define QSPI_BK1_IO3_GPIO_AF          GPIO_AF9_QUADSPI
#define QSPI_BK1_IO3_GPIO_CLK_ENABLE()  __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)
```

QSPI 驱动中，QSPI 的初始化函数，如下所示：

```
/** @brief      初始化 QSPI
 * @param      无
 * @retval     0, 成功; 1, 失败.
 */
uint8_t qspi_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    /* 使能时钟 */
    __HAL_RCC_QSPI_CLK_ENABLE();
    QSPI_BK1_CLK_GPIO_CLK_ENABLE();
    QSPI_BK1_NCS_GPIO_CLK_ENABLE();
    QSPI_BK1_IO0_GPIO_CLK_ENABLE();
    QSPI_BK1_IO1_GPIO_CLK_ENABLE();
    QSPI_BK1_IO2_GPIO_CLK_ENABLE();
    QSPI_BK1_IO3_GPIO_CLK_ENABLE();

    /* 配置 QSPI_BK1_CLK 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_CLK_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF9_QUADSPI;
    HAL_GPIO_Init(QSPI_BK1_CLK_GPIO_PORT, &gpio_init_struct);

    /* 配置 QSPI_BK1_NCS 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_NCS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF10_QUADSPI;
    HAL_GPIO_Init(QSPI_BK1_NCS_GPIO_PORT, &gpio_init_struct);

    /* 配置 QSPI_BK1_IO0 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_IO0_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF9_QUADSPI;
    HAL_GPIO_Init(QSPI_BK1_IO0_GPIO_PORT, &gpio_init_struct);

    /* 配置 QSPI_BK1_IO1 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_IO1_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF9_QUADSPI;
    HAL_GPIO_Init(QSPI_BK1_IO1_GPIO_PORT, &gpio_init_struct);

    /* 配置 QSPI_BK1_IO2 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_IO2_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    gpio_init_struct.Alternate = GPIO_AF9_QUADSPI;
    HAL_GPIO_Init(QSPI_BK1_IO2_GPIO_PORT, &gpio_init_struct);

    /* 配置 QSPI_BK1_IO3 引脚 */
    gpio_init_struct.Pin = QSPI_BK1_IO3_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
```

```

gpio_init_struct.Pull = GPIO_PULLUP;
gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
gpio_init_struct.Alternate = GPIO_AF9_QUADSPI;
HAL_GPIO_Init(QSPI_BK1_IO3_GPIO_PORT, &gpio_init_struct);

/* 配置 QSPI */
g_qspi_handle.Instance = QUADSPI;
g_qspi_handle.Init.ClockPrescaler = 1;
g_qspi_handle.Init.FifoThreshold = 4;
g_qspi_handle.Init.SampleShifting = QSPI_SAMPLE_SHIFTING_HALFCYCLE;
g_qspi_handle.Init.FlashSize = 25-1;
g_qspi_handle.Init.ChipSelectHighTime = QSPI_CS_HIGH_TIME_3_CYCLE;
g_qspi_handle.Init.ClockMode = QSPI_CLOCK_MODE_3;
g_qspi_handle.Init.FlashID = QSPI_FLASH_ID_1;
g_qspi_handle.Init.DualFlash = QSPI_DUALFLASH_DISABLE;

if(HAL_QSPI_Init(&g_qspi_handle) == HAL_OK)
{
    return 0;
}
else
{
    return 1;
}
}

```

可以看到，该函数会初始化 QSPI1。

接下来介绍 QSPI 发送命令函数，其定义如下：

```

/**
 * @brief   QSPI 发送命令
 * @param cmd : 要发送的指令
 * @param addr: 发送到的目的地址
 * @param mode: 模式, 详细位定义如下:
 * @arg mode[1:0]: 指令模式; 00, 无指令; 01, 单线传输指令; 10, 双线传输指令; 11, 四线传输指令
 * @arg mode[3:2]: 地址模式; 00, 无地址; 01, 单线传输地址; 10, 双线传输地址; 11, 四线传输地址
 * @arg mode[5:4]: 地址长度; 00, 8 位地址; 01, 16 位地址; 10, 24 位地址; 11, 32 位地址
 * @arg mode[7:6]: 数据模式; 00, 无数据; 01, 单线传输数据; 10, 双线传输数据; 11, 四线传输数据
 * @param dmcycle: 空指令周期数
 * @retval 无
 */
void qspi_send_cmd(uint8_t cmd, uint32_t addr, uint8_t mode, uint8_t dmcycle)
{
    QSPI_CommandTypeDef qspi_command_handle;

    qspi_command_handle.Instruction = cmd;
    qspi_command_handle.Address = addr;
    qspi_command_handle.DummyCycles = dmcycle;

    if(((mode >> 0) & 0x03) == 0)
        qspi_command_handle.InstructionMode = QSPI_INSTRUCTION_NONE;
    else if(((mode >> 0) & 0x03) == 1)
        qspi_command_handle.InstructionMode = QSPI_INSTRUCTION_1_LINE;
    else if(((mode >> 0) & 0x03) == 2)
        qspi_command_handle.InstructionMode = QSPI_INSTRUCTION_2_LINES;
    else if(((mode >> 0) & 0x03) == 3)
        qspi_command_handle.InstructionMode = QSPI_INSTRUCTION_4_LINES;

    if(((mode >> 2) & 0x03) == 0)
        qspi_command_handle.AddressMode = QSPI_ADDRESS_NONE;
    else if(((mode >> 2) & 0x03) == 1)
        qspi_command_handle.AddressMode = QSPI_ADDRESS_1_LINE;
    else if(((mode >> 2) & 0x03) == 2)
        qspi_command_handle.AddressMode = QSPI_ADDRESS_2_LINES;
    else if(((mode >> 2) & 0x03) == 3)

```

```

qspi_command_handle.AddressMode = QSPI_ADDRESS_4_LINES;

if(((mode >> 4)&0x03) == 0)
qspi_command_handle.AddressSize = QSPI_ADDRESS_8_BITS;
else if(((mode >> 4) & 0x03) == 1)
qspi_command_handle.AddressSize = QSPI_ADDRESS_16_BITS;
else if(((mode >> 4) & 0x03) == 2)
qspi_command_handle.AddressSize = QSPI_ADDRESS_24_BITS;
else if(((mode >> 4) & 0x03) == 3)
qspi_command_handle.AddressSize = QSPI_ADDRESS_32_BITS;

if(((mode >> 6) & 0x03) == 0)
qspi_command_handle.DataMode=QSPI_DATA_NONE;
else if(((mode >> 6) & 0x03) == 1)
qspi_command_handle.DataMode = QSPI_DATA_1_LINE;
else if(((mode >> 6) & 0x03) == 2)
qspi_command_handle.DataMode = QSPI_DATA_2_LINES;
else if(((mode >> 6) & 0x03) == 3)
qspi_command_handle.DataMode = QSPI_DATA_4_LINES;

qspi_command_handle.SIOOMode = QSPI_SIOO_INST_EVERY_CMD;
qspi_command_handle.AlternateByteMode = QSPI_ALTERNATE_BYTES_NONE;
qspi_command_handle.DdrMode = QSPI_DDR_MODE_DISABLE;
qspi_command_handle.DdrHoldHalfCycle = QSPI_DDR_HHC_ANALOG_DELAY;

HAL_QSPI_Command(&g_qspi_handle, &qspi_command_handle, 5000);
}

```

该函数主要就是配置 QSPI_CommandTypeDef 结构体的参数，并调用 HAL_QSPI_Command 函数配置发送命令，是一个重要的基础函数。

44.2.3 NOR Flash 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。NORFLASH 驱动源码包括两个文件：norflash.c、norflash.h、norflash_ex.c 和 norflash_ex.h。

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;
    qspi_init();           /* 初始化 QSPI */
    norflash_qspi_disable(); /* 退出 QPI 模式(避免芯片之前进入这个模式,导致下载失败) */
    norflash_qe_enable();   /* 使能 QE 位 */
    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_write_enable(); /* 写使能 */
            /* QPI,使能 4 字节地址指令,地址为 0,无数据_8 位地址_无地址_单线传输指令,
               无空周期,0 个字节数据 */
            qspi_send_cmd(FLASH_Enable4ByteAddr, 0, (0 << 6) | (0 << 4))
        }
    }
}

```

```

        | (0 << 2) | (1 << 0), 0);
    }
    //printf("ID:%x\r\n", g_norflash_type);
}

```

该函数用于初始化 NOR FLASH，首先调用 qspi_init 函数，初始化 STM32H750 的 QSPI 接口。然后退出 QPI 模式（避免芯片之前进入这个模式，导致下载失败），使能 FLASH 的 QE 位，使能 IO2/IO3。最后读取 FLASH ID，如果 SPI FLASH 为 W25Q256，还必须使能 4 字节地址模式。调用本函数在初始化完成以后，我们便可以通过 QSPI 接口读写 NOR FLASH 的数据了。

```

/***
 * @brief      读取 QSPI FLASH,仅支持 QSPI 模式
 * @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)
{
    /* QSPI,快速读数据,地址为 addr,4 线传输数据_24/32 位地址_4 线传输地址_1 线传输指令,
       6 空周期,datalen 个数据 */
    qspi_send_cmd(FLASH_FastReadQuad, addr, (3 << 6) | (g_norflash_addrw << 4)
                  | (3 << 2) | (1 << 0), 6);
    qspi_receive(pbuf, datalen);
}

```

该函数用于从 NOR FLASH 的指定地址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，这个代码相对来说就比较简单了，通过 qspi_send_cmd 函数，发送 FLASH_FastReadQuad 指令，并发送读数据首地址（addr），然后通过 qspi_receive 函数循环读取数据，存放在 pbuf 里面。

44.2.4 实验应用代码

本章实验的应用代码，如下所示：

```

/* 要写入到 FLASH 的字符串数组 */
const uint8_t g_text_buf[] = {"M100ZM7 STM32H7 QSPI TEST"};

/* 待写入 NOR Flash 数据的长度 */
#define TEXT_SIZE sizeof(g_text_buf)

int main(void)
{
    uint8_t key;
    uint16_t i = 0;
    uint8_t datatemp[TEXT_SIZE];
    uint32_t flashsize;
    uint16_t id = 0;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    usmart_dev.init(240);         /* 初始化 USMART */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "QSPI TEST", RED);
}

```

```

lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "WK_UP:Write KEY0:Read", RED);

id = norflash_ex_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();
}

lcd_show_string(30, 130, 200, 16, 16, "QSPI FLASH Ready!", BLUE);
flashsize = 16 * 1024 * 1024; /* FLASH 大小为 16M 字节 */

while (1)
{
    key = key_scan(0);

    if (key == WKUP_PRES) /* WK_UP 按下, 写入数据 */
    {
        lcd_fill(0, 150, 239, 319, WHITE);
        lcd_show_string(30, 150, 200, 16, 16, "Start Write FLASH...", BLUE);
        sprintf((char *)datatemp, "%s%d", (char *)g_text_buf, i);
        /* 从倒数第 100 个地址处开始, 写入 SIZE 长度的数据 */
        norflash_ex_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 */
    {
        lcd_show_string(30, 150, 200, 16, 16, "Start Read FLASH... . .", BLUE);
        /* 从倒数第 100 个地址处开始, 读出 SIZE 个字节 */
        norflash_ex_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();
        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。在 main 中初始化外部设备 NOR FLASH 需要注意，这里不需要调用 norflash_init 函数了，因为 sys.c 里面的 sys_qspi_enable_memmapmode 函数已经初始化了 QSPI 接口。如果再调用，则内存映射模式的设置被破坏，导致 QSPI 代码执行异常！如果不使用分散加载，即所有代码加载到内部 FLASH，才可以调用 norflash_init 函数。后面的无限循环就是 KEY1 按下，就写入 NOR FLASH。KEY0 按下，读取刚才写入的字符串并显示。

最后，我们将 norflash_ex_read_id、norflash_ex_erase_chip 和 norflash_ex_erase_sector 函数加入 USMART 控制，大家还可以把其他的函数加进来，这样，我们就可以通过串口调试助手，操作 NOR FLASH，方便大家测试。norflash_ex_erase_chip 函数大家谨慎调用，因为会把 NOR FLASH 的程序指令也擦除掉，会导致死机。如果不使用分散加载，就没关系。

44.3 下载验证

在完成编译和烧录操作后，若 MCU 与 NOR Flash 的连接无误，则可以在 LCD 上看到“QSPI FLASH Ready！”的提示信息，此时可以按下 WK_UP 按键往 NOR Flash 的指定地址写入指定的数据，然后再按下 KEY_0 按键从 NOR Flash 的指定地址将写入的数据读回来在 LCD 上进行显示，此时便可以看到 LCD 上显示“M100ZM7 STM32H7 QSPI TEST”的提示信息，该提示信息就是从 NOR Flash 中读回的数据。

第四十五章 触摸屏实验

本章将介绍使用 STM32H750 驱动 TFTLCD 模块上的触摸屏，实现一个类似画板的应用。通过本章的学习，读者将学习到使用 GPIO 模拟 IIC 和 SPI 时序以及触摸屏的驱动。

本章分为如下几个小节：

- 45.1 硬件设计
- 45.2 程序设计
- 45.3 下载验证

45.1 硬件设计

45.1.1 例程功能

1. LCD 上实时显示触摸屏被触摸的触摸轨迹，并可通过触摸右上角的 RST 来清空轨迹
2. 按下 KEY0 按键可进行电阻触摸屏的触摸校准
3. LED0 闪烁，提示程序正在运行

45.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
4. 24C02
IIC_SCL - PB10
IIC_SDA - PB11

45.1.3 原理图

本章实验使用了正点原子的 TFTLCD 模块（兼容正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块），该模块需通过 LCD 转接板与板载的 TFTLCD 接口进行连接，该接口与板载 MCU 的连接原理图，如图 27.1.3.1 所示。

如图 27.1.3.1 所示，TFTLCD 模块上的触摸屏使用 T_MISO、T_MOSI、T_PEN、T_CS、T_SCK 信号与 MCU 进行通讯，正点原子的 TFTLCD 模块采用了两种触摸屏，分别为：电阻式触摸屏和电容式触摸屏，其中电阻式触摸屏使用 SPI 协议与 MCU 进行通信，而电容式触摸屏则是使用 IIC 协议与 MCU 进行通讯，因此 TFT 模块上的触摸屏引脚对于不同的触摸屏有不同的引脚定义，请读者自行查看 TFTLCD 模块的用户手册查看具体的引脚定义。

45.2 程序设计

45.2.1 触摸屏驱动

本章实验的触摸屏驱动主要负责向应用层提供触摸屏的初始化和获取触摸屏触摸坐标等函数。本章实验中，触摸屏的驱动代码包括 touch.c、touch.h、ctiic.c、ctiic.h、ft5206.c、ft5206.h、gt9xxx.c、gt9xxx.h 八个文件。

触摸屏驱动中，触摸屏的初始化函数，如下所示：

```
/**  
 * @brief    初始化触摸屏  
 * @param    无  
 * @retval   触摸屏是否校准  
 *           0: 否
```

```

/*
 * 1: 是
 */
uint8_t tp_init(void)
{
    tp_dev.touchtype = 0; /* 默认设置（电阻屏、竖屏） */
    tp_dev.touchtype |= lcddev.dir & 0x01; /* 根据 LCD 判断是横屏还是竖屏 */

    /* 电容触摸屏, 4.3 寸/10.1 寸屏 */
    if ((lcddev.id == 0x5510) ||
        (lcddev.id == 0x4342) ||
        (lcddev.id == 0x4342) ||
        (lcddev.id == 0x1018))
    {
        gt9xxx_init();
        tp_dev.scan = gt9xxx_scan; /* 设置触摸屏扫描函数 */
        tp_dev.touchtype |= 0x80; /* 电容屏 */
        return 0;
    }
    /* SSD1963 7 寸屏或 7 寸 800*480/1024*600 RGB 屏 */
    else if ((lcddev.id == 0x1963) ||
              (lcddev.id == 0x7084) ||
              (lcddev.id == 0x7016))
    {
        if (!ft5206_init())
        {
            tp_dev.scan = ft5206_scan;
        }
        else
        {
            gt9xxx_init();
            tp_dev.scan = gt9xxx_scan;
        }
        tp_dev.touchtype |= 0X80;
        return 0;
    }
    /* 电阻屏 */
    else
    {
        /* 电阻屏的初始化操作, 代码省略 */
    }
    return 1;
}

```

从上面的代码中可以看出，触摸屏的初始化会读取 TFTLCD 模块 LCD 的 ID 号，来判断触摸屏的型号，因此在使用本触摸屏驱动初始化触摸屏前，需要先进行 LCD 的初始化。触摸屏的初始化函数会依据 LCD 的 ID 对不同型号的触摸屏进行初始化。

对于触摸屏驱动中的其他内容，请读者自行结合实际使用的 TFTLCD 的用户手册产看本章配套实验例程的源码。

45.2.2 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
}

```

```

key_init();                                /* 初始化按键 */
tp_dev.init();                             /* 触摸屏初始化 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "TOUCH TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

if ((tp_dev.touchtype & 0x80) == 0)
{
    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 */
}
}

```

在进行完触摸屏的初始化后，便根据触摸屏的不同类型调用了不同的测试函数，这是因为电容屏是支持多点触控的，而电阻屏并不支持。

电容屏测试测试函数，如下所示：

```

/**
 * @brief      电容触摸屏测试函数
 * @param      无
 * @retval     无
 */
void ctp_test(void)
{
    uint8_t t = 0;
    uint8_t i = 0;
    uint16_t lastpos[10][2];
    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();  
}  
}  
}
```

因为电容屏触摸屏支持多点触控，因此电容屏测试函数在扫描到每一个触摸点坐标后，便将每一个触摸点对应 LCD 屏幕上的坐标在 LCD 屏幕上进行绘制，并且每个触摸点使用不同的颜色进行绘制。

电阻屏测试函数，如下所示：

```
/***
 * @brief      电阻触摸屏测试函数
 * @param      无
 * @retval     无
 */
void rtp_test(void)
{
    uint8_t key;
    uint8_t i = 0;

    while (1)
    {
        key = key_scan(0);
        tp_dev.scan(0);

        if (tp_dev.sta & TP_PRES_DOWN) /* 触摸屏被按下 */
        {
            if (tp_dev.x[0] < lcddev.width && tp_dev.y[0] < lcddev.height)
            {
                if (tp_dev.x[0] > (lcddev.width - 24) && tp_dev.y[0] < 16)
                {
                    load_draw_dialog(); /* 清除 */
                }
                else
                {
                    tp_draw_big_point(tp_dev.x[0], tp_dev.y[0], RED); /* 画点 */
                }
            }
        }
    }
}
```

```
    delay_ms(10);           /* 没有按键按下的时候 */

}

if (key == KEY0_PRES)      /* KEY0 按下，则执行校准程序 */
{
    lcd_clear(WHITE);     /* 清屏 */
    tp_adjust();           /* 屏幕校准 */
    tp_save_adjust_data();
    load_draw_dialog();
}

i++;

if (i % 20 == 0)
{
    LED0_TOGGLE();
}
}

}
```

电阻触摸屏就相对简单，因为电阻触摸屏仅支持单点触控，因此仅需将触摸的1个触摸点对应LCD屏幕上坐标的点进行绘制即可，同时因为电阻触摸屏是需要校准的，因此当检测到KEY0按键被按下时，便会进行电阻触摸屏校准。

45.3 下载验证

在完成编译和烧录操作后，可以看到LCD上显示了本实验的实验信息，随后便进入“白板”界面，此时便可在LCD上通过触摸屏绘制出任意的图案，若是电容屏，还支持多点触控，若是电阻屏出现触摸点与LCD上显示的绘制点坐标不吻合，可以按下KEY0按键进行电阻触摸屏的校准。

第四十六章 FLASH 模拟 EEPROM 实验

本章将介绍使用 STM32H750 的片上 Flash 模拟 EEPROM，并对齐进行读写操作。通过本章的学习，读者将学习到 Flash 的使用。

本章分为如下几个小节：

- 46.1 硬件设计
- 46.2 程序设计
- 46.3 下载验证

46.1 硬件设计

46.1.1 例程功能

1. 按下 WKUP 按键可往 Flash 写入“STM32 FLASH TEST”
2. 按下 KEY0 按键可从 Flash 读取到“STM32 FLASH TEST”，并显示至 LCD
3. LED0 闪烁，提示程序正在运行

46.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15

46.1.3 原理图

本章实验使用的 Flash 为 STM32H750 的片上资源，因此没有对应的连接原理图。

46.2 程序设计

46.2.1 HAL 库的 Flash 驱动

STM32H750 的片上 Flash 是可以直接读取的，但 Flash 无法直接写入，写入 Flash 前，需要先对其进行擦除操作，其具体的操作步骤如下：

- ①：解锁访问 Flash 控制寄存器
- ②：关闭 Flash 数据 Cache
- ③：擦除 Flash
- ④：编程 Flash
- ⑤：开启 Flash 数据 Cache
- ⑥：上锁访问 Flash 控制寄存器

在 HAL 库中对应的驱动函数如下：

①：解锁访问 Flash 控制寄存器

该函数用于解锁访问 Flash 控制寄存器，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 46.2.1.1 函数 HAL_FLASH_Unlock()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 46.2.1.2 函数 HAL_FLASH_Unlock()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32H7xx_hal.h"

void example_fun(void)
{
    /* 解锁访问 Flash 控制寄存器 */
    HAL_FLASH_Unlock();
}
```

②：关闭 Flash 数据 Cache

该函数实际上是一个宏定义，用于关闭 Flash 数据 Cache，该宏的定义如下所示：

```
#define __HAL_FLASH_DATA_CACHE_DISABLE() (FLASH->ACR &= (~FLASH_ACR_DCE))
```

该宏的使用示例，如下所示：

```
#include "stm32H7xx_hal.h"

void example_fun(void)
{
    /* 关闭 Flash 数据 Cache */
    __HAL_FLASH_DATA_CACHE_DISABLE();
}
```

③：擦除 Flash

该函数用于擦除 Flash，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                                    uint32_t *SectorError);
```

该函数的形参描述，如下表所示：

形参	描述
pEraseInit	指向 Flash 擦除信息结构体的指针 需自行定义，并根据 Flash 的擦除信息填充结构体中的成员变量
SectorError	擦除失败的扇区

表 46.2.1.3 函数 HAL_FLASHEx_Erase()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 46.2.1.4 函数 HAL_FLASHEx_Erase()返回值描述

该函数使用 FLASH_EraseInitTypeDef 类型结构体指针传入了 Flash 的擦除信息，该结构体的定义如下所示：

```
typedef struct
{
    uint32_t TypeErase;      /* 擦除类型 */
    uint32_t Banks;         /* 块编号 */
    uint32_t Sector;        /* 起始扇区 */
    uint32_t NbSectors;     /* 扇区数量 */
    uint32_t VoltageRange;  /* 电压范围 */
} FLASH_EraseInitTypeDef;
```

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    FLASH_EraseInitTypeDef erase = {0};
    uint32_t sector

    /* 擦除 Flash */
    HAL_FLASHEx_Erase(&erase, &sector);
```

④：编程 Flash

该函数用于编程 Flash，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_FLASH_Program( uint32_t TypeProgram,
                                     uint32_t Address,
                                     uint64_t Data);
```

该函数的形参描述，如下表所示：

形参	描述
TypeProgram	编程类型 例如：FLASH_TYPEPROGRAM_BYTE、FLASH_TYPEPROGRAM_HALFWORD 等（在 stm32h7xx_hal_flash.h 文件中有定义）
FlashAddress	起始地址
DataAddress	数据

表 46.2.1.5 函数 HAL_FLASH_Program()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 46.2.1.6 函数 HAL_FLASH_Program()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    uint8_t buf;

    /* 编程 Flash */
    HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE, 0, &buf);
}
```

⑤：开启 Flash 数据 Cache

该函数实际上是一个宏定义，用于开启 Flash 数据 Cache，该宏的定义如下所示：

```
#define __HAL_FLASH_DATA_CACHE_ENABLE() (FLASH->ACR |= FLASH_ACR_DCEN)
```

该宏的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 开启 Flash 数据 Cache */
    __HAL_FLASH_DATA_CACHE_ENABLE();
}
```

⑥：上锁访问 Flash 控制寄存器

该函数用于上锁访问 Flash 控制寄存器，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_FLASH_Lock(void);
```

该函数的形参描述，如下表所示：

形参	描述
无	无

表 46.2.1.7 函数 HAL_FLASH_Lock()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 46.2.1.8 函数 HAL_FLASH_Lock()返回值描述

该函数的使用示例，如下所示：

```
#include "stm32h7xx_hal.h"

void example_fun(void)
```

```
{
    /* 上锁访问 Flash 控制寄存器 */
    HAL_FLASH_Lock();
}
```

46.2.2 Flash 驱动

下面我们开始介绍 stmflash.c 的程序，具体程序源码如下：

```
/***
 * @brief      得到 FLASH 的错误状态
 * @param      无
 * @retval     错误代码
 * @arg        0    , 无错误
 * @arg        其他, 错误编号
 */
static uint8_t stmflash_get_error_status(void)
{
    uint32_t res = 0;
    res = FLASH->SR1;

    if (res & (1 << 17)) return 1; /* WRPERR=1, 写保护错误 */
    if (res & (1 << 18)) return 2; /* PGSERR=1, 编程序列错误 */
    if (res & (1 << 19)) return 3; /* STRBERR=1, 复写错误 */
    if (res & (1 << 21)) return 4; /* INCERR=1, 数据一致性错误 */
    if (res & (1 << 22)) return 5; /* OPERR=1, 写/擦除错误 */
    if (res & (1 << 23)) return 6; /* RDPERR=1, 读保护错误 */
    if (res & (1 << 24)) return 7; /* RDSERR=1, 非法访问加密区错误 */
    if (res & (1 << 25)) return 8; /* SNECCERR=1, 1bit ecc 校正错误 */
    if (res & (1 << 26)) return 9; /* DBECCERR=1, 2bit ecc 错误 */

    return 0; /* 没有任何状态/操作完成. */
}

/***
 * @brief      等待操作完成
 * @param      time : 要延时的长短
 * @retval     错误代码
 * @arg        0    : 已完成
 * @arg        1~9 : 错误代码
 * @arg        0xFF: 超时
 */
static uint8_t stmflash_wait_done(uint32_t time)
{
    uint8_t res = 0;
    uint32_t tempreg = 0;

    while (1)
    {
        tempreg = FLASH->SR1;

        if (((tempreg & 0x07) == 0)
        {
            break; /* BSY=0, WBNE=0, QW=0, 则操作完成 */
        }

        time--;
        if (time == 0) return 0xFF;
    }

    res = stmflash_get_error_status();
```

```
if (res)
{
    FLASH->CCR1 = 0X07EE0000; /* 清所有错误标志 */
}

return res;
}

/***
 * @brief      在 FLASH 指定地址写 8 个字,即 256bit
 * @note       必须以 256bit 为单位(32 字节)编程!!
 * @param      faddr : 指定地址(此地址必须为 4 的倍数!!)
 * @param      pdata : 要写入的数据
 * @retval     错误代码
 * @arg        0   : 写入成功
 * @arg        其他: 错误代码
 */
static uint8_t stmflash_write_8word(uint32_t faddr, uint32_t *pdata)
{
    volatile uint8_t nword = 8; /* 每次写 8 个字,256bit */
    uint8_t res;
    res = stmflash_wait_done(0xFFFF);

    if (res == 0) /* OK */
    {
        FLASH->CR1 &= ~(3 << 4); /* PSIZE1[1:0]=0,清除原来的设置 */
        FLASH->CR1 |= 2 << 4; /* 设置为 32bit 宽 */
        FLASH->CR1 |= 1 << 1; /* PG1=1,编程使能 */

        while (nword)
        {
            *(volatile uint32_t *)faddr = *pdata; /* 写入数据 */
            faddr += 4; /* 写地址+4 */
            pdata++;
            /* 偏移到下一个数据首地址 */
            nword--;
        }
        __DSB(); /* 写操作完成后,屏蔽数据同步,使 CPU 重新执行指令序列 */
        res = stmflash_wait_done(0xFFFF); /* 等待操作完成,一个字编程,最多 100us. */
        FLASH->CR1 &= ~(1 << 1); /* PG1=0,清除扇区擦除标志 */
    }

    return res;
}

/***
 * @brief      读取指定地址的一个字(32 位数据)
 * @param      faddr : 要读取的地址
 * @retval     读取到的数据
 */
uint32_t stmflash_read_word(uint32_t faddr)
{
    return *(volatile uint32_t *)faddr;
}

/***
 * @brief      从指定地址开始写入指定长度的数据
 * @note       特别注意: 因为 STM32H750 只有一个扇区 (128K),因此我们规定:
 *             前 16K 留作 BootLoader 用
 *             后 112K 用作 APP 用,我们要做写入测试,尽量使用 16K 以后的地址,否则容易出问题
 *             另外,由于写数据时,必须是 0xFF 才可以写入数据,因此不可避免的需要擦除扇区
 *             所以在擦除时需要先对前 16K 数据做备份保存(读取到 RAM),然后再写入,以保证
 *             前 16K 数据的完整性。且执行写入操作的时候,不能发生任何中断(凡是在写入时执
 *             行)
```

```
*          行内部 FLASH 代码，必将导致 hardfault)。
* @param    waddr : 起始地址(此地址必须为 32 的倍数!!，否则写入出错!)
* @param    pbuf  : 数据指针
* @param    length: 字数(就是要写入的 32 位数据的个数,一次至少写入 32 字节,即 8 个字)
* @retval   无
*/
/* FLASH 写入数据缓存 */
uint32_t g_flashbuf[BOOT_FLASH_SIZE / 4];

void stmflash_write(uint32_t waddr, uint32_t *pbuf, uint32_t length)
{
    FLASH_EraseInitTypeDef flash_erase_init_handle;
    HAL_StatusTypeDef hal_status = HAL_OK;
    uint32_t SectorError = 0;
    uint32_t addrx = 0;
    uint32_t endaddr = 0;
    uint16_t wbfycyc = BOOT_FLASH_SIZE/32; /* 写 bootflashbuf 时,需要执行的循环数 */
    uint32_t *wbfptr;
    uint32_t wbfaddr;
    /* 写入地址小于 STM32_FLASH_BASE+BOOT_FLASH_SIZE,非法. */
    if (waddr < (STM32_FLASH_BASE + BOOT_FLASH_SIZE)) return;
    /* 写入地址大于 STM32 总 FLASH 地址范围,非法. */
    if (waddr > (STM32_FLASH_BASE + STM32_FLASH_SIZE)) return;

    if (waddr % 32) return; /* 写入地址不是 32 字节倍数,非法. */

    HAL_FLASH_Unlock(); /* 解锁 */
    addrx = waddr; /* 写入的起始地址 */
    endaddr = waddr + length * 4; /* 写入的结束地址 */

    while (addrx < endaddr) /* 扫清一切障碍.(对非 FFFFFFFF 的地方,先擦除) */
    {
        /* 有非 0xFFFFFFFF 的地方,要擦除这个扇区 */
        if (stmflash_read_word(addrx) != 0xFFFFFFFF)
        {
            /* 读出 BOOT_FLASH_SIZE 大小数据 */
            stmflash_read(STM32_FLASH_BASE, g_flashbuf, BOOT_FLASH_SIZE / 4);
            INTX_DISABLE(); /* 禁止所有中断 */
            /* 擦除类型,扇区擦除 */
            flash_erase_init_handle.TypeErase = FLASH_TYPEERASE_SECTORS;
            flash_erase_init_handle.Sector = FLASH_SECTOR_0; /* 要擦除的扇区 */
            flash_erase_init_handle.Banks = FLASH_BANK_1; /* 操作 BANK1 */
            flash_erase_init_handle.NbSectors = 1; /* 一次只擦除一个扇区 */
            /* 电压范围, VCC=2.7~3.6V 之间 */
            flash_erase_init_handle.VoltageRange = FLASH_VOLTAGE_RANGE_3;
            hal_status = HAL_FLASHEx_Erase(&flash_erase_init_handle, &SectorError);
            if (hal_status != HAL_OK) /* 发生错误了 */
            {
                INTX_ENABLE(); /* 允许中断 */
                break; /* 发生错误了 */
            }
            SCB_CleanInvalidateDCache(); /* 清除无效的 D-Cache */
            wbfptr = g_flashbuf; /* 指向 g_flashbuf 首地址 */
            wbfaddr = STM32_FLASH_BASE; /* 指向 STM32 FLASH 首地址 */
            while (wbfycyc) /* 写数据 */
            {
                if (stmflash_write_8word(wbfaddr, wbfptr)) /* 写入数据 */
                {
                    break; /* 写入异常 */
                }
                wbfaddr += 32;
            }
        }
    }
}
```

```

        wbfptr += 8;
        wbfyc--;
    }
    INTX_ENABLE(); /* 允许中断 */
}
else
{
    addrx += 4; /* 偏移到下一个位置 */
}
/* 等待上次操作完成 */
FLASH_WaitForLastOperation(FLASH_WAITETIME, FLASH_BANK_1);
}
/* 等待上次操作完成 */
hal_status = FLASH_WaitForLastOperation(FLASH_WAITETIME, FLASH_BANK_1);

if (hal_status == HAL_OK)
{
    while (waddr < endaddr) /* 写数据 */
    {
        if (stmflash_write_8word(waddr, pbuf)) /* 写入数据 */
        {
            break; /* 写入异常 */
        }
        waddr += 32;
        pbuf += 8;
    }
    HAL_FLASH_Lock(); /* 上锁 */
}

/**
 * @brief      从指定地址开始读出指定长度的数据
 * @param      raddr : 起始地址
 * @param      pbuf : 数据指针
 * @param      length: 要读取的字(32)数, 即4个字节的整数倍
 * @retval     无
 */
void stmflash_read(uint32_t raddr, uint32_t *pbuf, uint32_t length)
{
    uint32_t i;

    for (i = 0; i < length; i++)
    {
        pbuf[i] = stmflash_read_word(raddr); /* 读取4个字节. */
        raddr += 4; /* 偏移4个字节. */
    }
}
/*****************************************/
/* 测试用代码 */
/**
 * @brief      测试写数据(写1个字)
 * @param      waddr : 起始地址
 * @param      wdata : 要写入的数据
 * @retval     读取到的数据
 */
void test_write(uint32_t waddr, uint32_t wdata)
{
    stmflash_write(waddr, &wdata, 1); /* 写入一个字 */
}

```

该部分代码，我们重点介绍一下 `stmflash_write` 函数，该函数用于在 STM32H750 的指定地址写入指定长度的数据，有几个要注意的点：

- 1, 写入地址必须是在 `BOOT_FLASH_SIZE` 以后。

2, 写入地址必须是 32 的倍数。

3, 单次写入长度必须是 32 字节的倍数 (8 个字)。

第 1 点重点说明一下, BOOT_FLASH_SIZE 是我们在 stmflash.h 里面定义的一个宏定义, 其值为: 0X8000, 即 32K, 也就是写入数据必须在 32K 以后的地址 (0X0800 0000 + 0X8000) 写入。因为 STM32H750 内部仅有 1 个扇区, 为了方便做 IAP 应用, 必须把这个扇区分为 2 部分: IAP 部分和 APP 部分, 我们预留 32K 地址范围给 IAP, 所以本例程写入地址必须在 32K 以后, 以便后面的 IAP 应用。

另外, 由于 H750 内部只有 1 个扇区, 在擦除扇区的时候, 连带所有数据都擦掉了 (IAP 也擦了), 所以, 为了能够实现保留 IAP 的效果, 我们在 stmflash_write 函数里面, 执行擦除扇区之前, 会先备份前 16K 的数据, 将前 16K 数据保存到 SRAM, 然后再擦除扇区, 擦除完了以后, 再从 SRAM 里面恢复这 16K 数据到扇区里面去, 这样就可以实现类似擦除扇区但是仍保留前 16K 数据的效果。需要特别注意的是: 在擦除扇区或写入扇区数据的时候, 不能执行任何内部 FLASH 上面的代码! 所有相关的代码必须存放到外部 QSPI FLASH。体现到本例程就是: stmflash.c 的代码, 都应该存放到外部 QSPI FLASH, 而不能存放到 H750 内部 FLASH。

第 2 点和第 3 点则是由于 STM32H7 的 FLASH 特性, 每次写入必须是 256 位宽, 也就是 32 字节, 因此写入首地址必须是 32 字节的倍数, 且写入数据长度必须是 32 字节的倍数。

另外, 在 STMFLASH_Write8Word 函数里面, 有一个 __DSB 函数, 该函数用于屏蔽数据同步, 该函数在 cmsis_armcc.h 里面定义, 这里在执行等待操作完成之前, 必须调用该函数, 否则将无法往 FLASH 写入数据。

由于我们使用了分散加载 (qspi_code.scf), stmflash.c 编译后是自动存放到外部 QSPI FLASH 的, 所以不需要做额外的设置。

46.2.3 实验应用代码

在 main.c 里面编写如下代码:

```
const uint8_t g_text_buf[] = {"STM32 FLASH TEST"}; /* 要写入到 FLASH 的字符串数组 */
#define TEXT_LENGTH sizeof(g_text_buf) /* 数组长度 */

/* SIZE 表示字长(4 字节), 大小必须是 4 的整数倍, 如果不是的话, 强制对齐到 4 的整数倍 */
#define SIZE TEXT_LENGTH / 4 + ((TEXT_LENGTH % 4) ? 1 : 0)

/* 设置 FLASH 保存地址(必须大于用户代码区地址范围, 且为 4 的倍数) */
#define FLASH_SAVE_ADDR 0X80008000

int main(void)
{
    uint8_t key = 0;
    uint16_t i = 0;
    uint8_t datatemp[SIZE];

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */

    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 == WKUP_PRES) /* WK_UP 按下,写入 STM32 FLASH */  
    {  
        lcd_fill(0, 150, 239, 319, WHITE);  
        lcd_show_string(30, 150, 200, 16, 16, "Start Write FLASH....", RED);  
        stmflash_write(FLASH_SAVE_ADDR, (uint32_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, (uint32_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 WK_UP 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 STM32H750 的 FLASH 读写函数，方便测试。

46.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验相关的信息，此时便可按下 WK_UP 按键往 Flash 的指定地址写入指定数据，然后再按下 KEY0 按键从 Flash 的指定地址将写入的数据读回来在 LCD 上进行显示，此时便可以看到在 LCD 上显示了“STM32 FLASH TEST”的提示信息，该提示信息就是从 Flash 中读回的数据。

第四十七章 摄像头实验

本章将介绍使用 STM32H750 驱动 OV5640 摄像头，从而获取摄像头输出的图像数据，并将其显示在 LCD 上，也可通过串口发送至 PC 上位机软件。通过本章的学习，读者将学习到数字摄像头接口（DCMI）的使用。

本章分为如下几个小节：

47.1 硬件设计

47.2 程序设计

47.3 下载验证

47.1 硬件设计

47.1.1 例程功能

1. 初始化摄像头后，可通过按下 WKUP 按键或 KEY0 按键进入 RGB565 或 JPEG 测试模式
2. RGB565 测试模式下，LCD 显示 DCMI 捕获的摄像头画面，可通过按键修改画面对比度或缩放
3. JPEG 测试模式下，USART2 输出 DCMI 捕获的 JPEG 数据，可通过按键对画面对比度或尺寸进行设置
4. 串口 1 输出 DCMI 接收摄像头数据帧率
5. LED1 闪烁，提示 DCMI 接收到一帧数据

47.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15
4. USART1
 - USART1_TX - PA9
 - USART1_RX - PA10
5. USART2
 - USART2_TX - PA2
6. TIM6
7. ATK-MC5640 摄像头模块
 - OV_D0~D7 - PC6/PC7/PC8/PC9/PC11/PD3/PB8/PB9
 - OV_SCL - PB10
 - OV_SDA - PB11
 - OV_VSYNC - PB7
 - OV_HREF - PA4
 - OV_PCLK - PA6
 - OV_PWDN - PC4
 - OV_RESET - PA7
 - OV_XCLK - PA8

47.1.3 原理图

本章实验使用了一个正点原子 OV5640 摄像头模块（ATK-MC5640），该模块需通过摄像头模块延长线与板载的 CAMERA 接口进行连接，该接口也可与 OLED 模块进行连接，该接口与板

载 MCU 的连接原理图, 如下图所示:

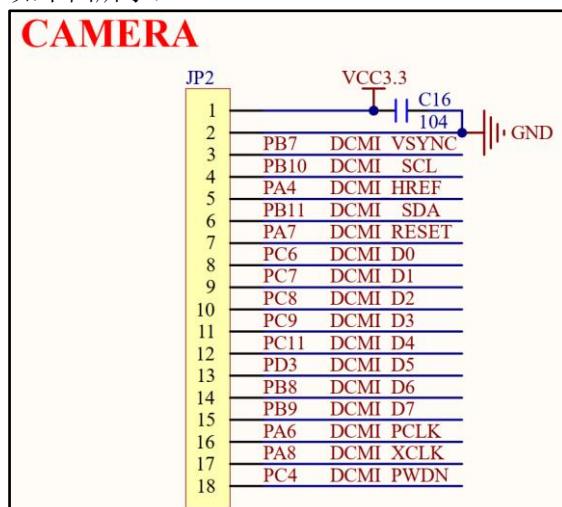


图 47.1.3.1 摄像头模块与 MCU 的连接原理图

47.2 程序设计

47.2.1 HAL 库的 DCMI 驱动

本章使用通过 DCMI 驱动摄像头模块，并获取摄像头模块返回的图像数据，分为 RGB565 模式和 JPEG 模式，其中 RGB565 模式直接将摄像头返回的数据流通过 DMA 传输至 LCD，JPEG 模式下将整帧的 JPEG 图像数据通过 USART2 传输至 PC 上位机软件，因为需要获取 JPEG 整帧的数据，因此 JPEG 模式下需要使用到 DCMI 的捕获完成中断，其具体的步骤如下：

- ①: 初始化 DCMI
- ②: 使能 DCMI 捕获

在 HAL 库中对应的驱动函数如下：

①: 初始化 DCMI

该函数用于初始化 DCMI 的各项参数，其函数原型如下所示：

```
HAL_StatusTypeDef HAL_DCMI_Init(DCMI_HandleTypeDef *hdcmi);
```

该函数的形参描述，如下表所示：

形参	描述
hdcmi	指向 DCMI 句柄的指针

表 47.2.1.1 函数 HAL_DCMI_Init()形参描述

该函数的返回值描述，如下表所示：

返回值	描述
HAL_StatusTypeDef	HAL 状态

表 47.2.1.2 函数 HAL_DCMI_Init()返回值描述

该函数需要传入 DCMI 的句柄指针，该句柄中就包含了 DCMI 的初始化配置参数结构体，该结构体的定义如下所示：

```
typedef struct {
    uint32_t SyncroMode; /* 同步方式选择硬件同步模式还是内嵌码模式 */
    uint32_t PCKPolarity; /* 设置像素时钟的有效边沿 */
    uint32_t VSPolarity; /* 设置垂直同步的有效电平 */
    uint32_t HSPolarity; /* 设置水平同步的有效边沿 */
    uint32_t CaptureRate; /* 设置图像的帧捕获率 */
    uint32_t ExtendedDataMode; /* 设置数据线的宽度（扩展数据模式） */
    DCMI_CodesInitTypeDef SyncroCode; /* 分隔符设置 */
    uint32_t JPEGMode; /* JPEG 模式选择 */
    uint32_t ByteSelectMode; /* 配置字节选项模式 */
    uint32_t ByteSelectStart; /* 字节选择开始 */
} DCMI_CodesInitTypeDef;
```

```

    uint32_t LineSelectMode;           /* 行选择模式 */
    uint32_t LineSelectStart;         /* 指定数据行是奇数还是偶数 */
} DCMI_HandleTypeDef;

```

该函数的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    DCMI_HandleTypeDef dcmi_handle = {0};

    /* 初始化 DCMI */
    dcmi_handle.Instance = DCMI;
    dcmi_handle.Init.SynchroMode = DCMI_SYNCHRO_HARDWARE;
    dcmi_handle.Init.PCKPolarity = DCMI_PCKPOLARITY_RISING;
    dcmi_handle.Init.VSPolarity = DCMI_VSPOLARITY_LOW;
    dcmi_handle.Init.HSPolarity = DCMI_HSPOLARITY_LOW;
    dcmi_handle.Init.CaptureRate = DCMI_CR_ALL_FRAME;
    dcmi_handle.Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B;
    dcmi_handle.Init.JPEGMode = DCMI_JPEG_DISABLE;
    HAL_DCMI_Init(&dcmi_handle);
}

```

②：使能 DCMI 捕获

该函数实际上是一个宏定义，用于使能 DCMI 捕获，该宏的定义如下所示：

```
#define __HAL_DMA_ENABLE(__HANDLE__) ((__HANDLE__)->Instance->CR |= DMA_SxCR_EN)
```

该宏的使用示例，如下所示：

```

#include "stm32h7xx_hal.h"

void example_fun(void)
{
    /* 使能 DCMI 捕获 */
    __HAL_DMA_ENABLE(&dcmi_handle);
}

```

47.2.2 DCMI 驱动代码

本章实验的 DCMI 驱动主要负责向应用层提供 DCMI 的初始化和启动 DCMI 传输等函数。本章实验中，DCMI 的驱动代码包括 dcmi.c 和 dcmi.h 两个文件。

由于 DCMI 使用了大量的 GPIO 引脚，因此对于 GPIO 的相关定义，请读者自行查看 dcmi.c 这个文件。

DCMI 驱动中，DCMI 的初始化函数，如下所示：

```

/***
 * @brief      DCMI 初始化
 * @note       IO 对应关系如下：
 *             摄像头模块 ----- STM32 开发板
 *             OV_D0~D7 ----- PC6/PC7/PC8/PC9/PC11/PD3/PB8/PB9
 *             OV_SCL ----- PB10
 *             OV_SDA ----- PB11
 *             OV_VSYNC ----- PB7
 *             OV_HREF ----- PA4
 *             OV_RESET ----- PA7
 *             OV_PCLK ----- PA6
 *             OV_PWDN ----- PC4
 *             本函数仅初始化 OV_D0~D7/OV_VSYNC/OV_HREF/OV_PCLK 等信号(11个).
 * @param      无
 * @retval     无
 */
void dcmi_init(void)
{
    g_dcmi_handle.Instance = DCMI;
    g_dcmi_handle.Init.SynchroMode = DCMI_SYNCHRO_HARDWARE; /* 硬件同步 */
}

```

```

g_dcmi_handle.Init.PCKPolarity = DCMI_PCKPOLARITY_RISING; /* PK 上升沿有效 */
g_dcmi_handle.Init.VSPolarity = DCMI_VSPOLARITY_LOW; /* VS 低电平有效 */
g_dcmi_handle.Init.HSPolarity = DCMI_HSPOLARITY_LOW; /* HS 低电平有效 */
g_dcmi_handle.Init.CaptureRate = DCMI_CR_ALL_FRAME; /* 全帧捕获 */
g_dcmi_handle.Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B; /* 8 位数据格式 */
HAL_DCMI_Init(&g_dcmi_handle); /* 初始化 DCMI，此函数会开启帧中断 */

/* 关闭行中断、VSYNC 中断、同步错误中断和溢出中断 */
//__HAL_DCMI_DISABLE_IT(&g_dcmi_handle, DCMI_IT_LINE | DCMI_IT_VSYNC
// | DCMI_IT_ERR | DCMI_IT_OVR);

/* 关闭所有中断，函数 HAL_DCMI_Init() 会默认打开很多中断，开启这些中断
以后我们就需要对这些中断做相应的处理，否则的话就会导致各种各样的问题，
但是这些中断很多都不需要，所以这里将其全部关闭掉，也就是将 IER 寄存器清零。
关闭完所有中断以后再根据自己的实际需求来使能相应的中断 */
DCMI->IER=0x0;
__HAL_DCMI_ENABLE_IT(&g_dcmi_handle, DCMI_IT_FRAME); /* 使能帧中断 */
__HAL_DCMI_ENABLE(&g_dcmi_handle); /* 使能 DCMI */
}

```

该函数主要对 DCMI_HandleTypeDef 结构体成员赋值并初始化，最后关闭所有中断，只开启帧中断，使能 DCMI。而 DCMI 接口的 GPIO 口的初始化是在 HAL_DCMI_MspInit 回调函数中完成，其定义如下：

```

/**
 * @brief      DCMI 底层驱动，引脚配置，时钟使能，中断配置
 * @param      hdcmi:DCMI 句柄
 * @note       此函数会被 HAL_DCMI_Init() 调用
 * @retval     无
 */
void HAL_DCMI_MspInit(DCMI_HandleTypeDef* hdcmi)
{
    GPIO_InitTypeDef gpio_init_struct;

    __HAL_RCC_DCMI_CLK_ENABLE(); /* 使能 DCMI 时钟 */
    __HAL_RCC_GPIOA_CLK_ENABLE(); /* 使能 GPIOA 时钟 */
    __HAL_RCC_GPIOB_CLK_ENABLE(); /* 使能 GPIOB 时钟 */
    __HAL_RCC_GPIOC_CLK_ENABLE(); /* 使能 GPIOC 时钟 */
    __HAL_RCC_GPIOD_CLK_ENABLE(); /* 使能 GPIOD 时钟 */

    gpio_init_struct.Pin = GPIO_PIN_4 | GPIO_PIN_6;
    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_VERY_HIGH; /* 高速 */
    gpio_init_struct.Alternate = GPIO_AF13_DCMI; /* 复用为 DCMI */
    HAL_GPIO_Init(GPIOA, &gpio_init_struct); /* 初始化 PA4, 6 引脚 */

    gpio_init_struct.Pin = GPIO_PIN_7 | GPIO_PIN_8 | GPIO_PIN_9;
    HAL_GPIO_Init(GPIOB, &gpio_init_struct); /* 初始化 PB7,8,9 引脚 */

    gpio_init_struct.Pin = GPIO_PIN_6 | GPIO_PIN_7 | GPIO_PIN_8
                         | GPIO_PIN_9 | GPIO_PIN_11;
    HAL_GPIO_Init(GPIOC, &gpio_init_struct); /* 初始化 PC6,7,8,9,11 引脚 */

    gpio_init_struct.Pin = GPIO_PIN_3;
    HAL_GPIO_Init(GPIOD, &gpio_init_struct); /* 初始化 PD3 引脚 */

    HAL_NVIC_SetPriority(DCMI_IRQn, 2, 2); /* 抢占优先级 2，子优先级 2 */
    HAL_NVIC_EnableIRQ(DCMI_IRQn); /* 使能 DCMI 中断 */
}

```

DCMI 接口的 GPIO 口前面都介绍过了，该函数最后设置了 DCMI 中断抢占优先级为 2，子优先级为 2，并且使能 DCMI 中断。

接下来介绍 DCMI DMA 配置初始化函数，其定义如下：

```
/** @brief      DCMI DMA 配置
 * @param mem0addr: 存储器地址 0      将要存储摄像头数据的内存地址(也可以是外设地址)
 * @param mem1addr: 存储器地址 1      当只使用 mem0addr 的时候, 该值必须为 0
 * @param memsize : 存储器长度        0~65535
 * @param memblen : 存储器位宽        DMA_MDATAALIGN_BYTE, 8 位;
 *                                     DMA_MDATAALIGN_HALFWORD, 16 位; DMA_MDATAALIGN_WORD, 32 位
 * @param meminc : 存储器增长方式    DMA_MINC_DISABLE, 不增长; DMA_MINC_ENABLE, 增长
 * @retval 无
 */
void dcmi_dma_init(uint32_t mem0addr, uint32_t mem1addr,
                    uint16_t memsize, uint32_t memblen, uint32_t meminc)
{
    __HAL_RCC_DMA1_CLK_ENABLE(); /* 使能 DMA1 时钟 */
    /* 将 DMA 与 DCMI 联系起来 */
    __HAL_LINKDMA(&g_dcmi_handle, DMA_Handle, g_dma_dcmi_handle);
    /* 先关闭 DMA 传输完成中断(否则在使用 MCU 屏的时候会出现花屏的情况) */
    __HAL_DMA_DISABLE_IT(&g_dma_dcmi_handle, DMA_IT_TC);

    g_dma_dcmi_handle.Instance = DMA1_Stream1; /* DMA1 数据流 1 */
    g_dma_dcmi_handle.Init.Request = DMA_REQUEST_DCMI; /* DCMI_DMA */
    g_dma_dcmi_handle.Init.Direction = DMA_PERIPH_TO_MEMORY; /* 外设到存储器 */
    g_dma_dcmi_handle.InitPeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
    g_dma_dcmi_handle.InitMemInc = meminc; /* 存储器增量模式 */

    /* 外设数据长度:32 位 */
    g_dma_dcmi_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
    g_dma_dcmi_handle.Init.MemDataAlignment = memblen; /* 存储器数据长度:8/16/32 位 */
    g_dma_dcmi_handle.Init.Mode = DMA_CIRCULAR; /* 使用循环模式 */
    g_dma_dcmi_handle.Init.Priority = DMA_PRIORITY_HIGH; /* 高优先级 */
    g_dma_dcmi_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE; /* 禁用 FIFO */
    g_dma_dcmi_handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL; /* 无效 */
    g_dma_dcmi_handle.Init.MemBurst = DMA_MBURST_SINGLE; /* 存储器突发传输 */
    g_dma_dcmi_handle.Init.PeriphBurst = DMA_PBURST_SINGLE; /* 外设突发单次传输 */
    HAL_DMA_DeInit(&g_dma_dcmi_handle); /* 先清除以前的设置 */
    HAL_DMA_Init(&g_dma_dcmi_handle); /* 初始化 DMA */

    /* 在开启 DMA 之前先使用 __HAL_UNLOCK() 解锁一次 DMA, 因为 HAL_DMA_Start() */
    /* 和 HAL_DMAEx_MultiBufferStart() 这两个函数一开始要先使用 __HAL_LOCK() 锁定 DMA, */
    /* 而函数 __HAL_LOCK() 会判断当前的 DMA 状态是否为锁定状态, 如果是锁定状态的话就直接 */
    /* 返回 HAL_BUSY, 这样会导致函数 HAL_DMA_Start() 和 HAL_DMAEx_MultiBufferStart() */
    /* 后续的 DMA 配置程序直接被跳过! DMA 也就不能正常工作, 为了避免这种现象, 所以在启动 */
    /* DMA 之前先调用 __HAL_UNLOCK() 先解锁一次 DMA。 */
    __HAL_UNLOCK(&g_dma_dcmi_handle);
    if (mem1addr == 0) /* 开启 DMA, 不使用双缓冲 */
    {
        HAL_DMA_Start(&g_dma_dcmi_handle, (uint32_t)&DCMI->DR, mem0addr, memsize);
    }
    else /* 使用双缓冲 */
    {
        HAL_DMAEx_MultiBufferStart(&g_dma_dcmi_handle, (uint32_t)&DCMI->DR,
                                    mem0addr, mem1addr, memsize); /* 开启双缓冲 */
        __HAL_DMA_ENABLE_IT(&g_dma_dcmi_handle, DMA_IT_TC); /* 开启传输完成中断 */
        HAL_NVIC_SetPriority(DMA1_Stream1_IRQn, 2, 3); /* DMA 中断优先级 */
        HAL_NVIC_EnableIRQ(DMA1_Stream1_IRQn);
    }
}
```

该函数用于配置 DCMI 的 DMA 传输，其外设地址固定为：DCMI->DR，而存储器地址可变（LCD 或者 SRAM）。DMA 被配置为循环模式，一旦开启，DMA 将不停的循环传输数据。

下面介绍的是 DCMI 启动传输和关闭传输函数，它们的定义分别如下：

```
/***
 * @brief      DCMI, 启动传输
 * @param      无
 * @retval     无
 */
void dcmi_start(void)
{
    lcd_set_cursor(0, 0);          /* 设置坐标到原点 */
    lcd_write_ram_prepare();      /* 开始写入 GRAM */
    __HAL_DMA_ENABLE(&g_dma_dcmi_handle); /* 使能 DMA */
    DCMI->CR |= DCMI_CR_CAPTURE;  /* DCMI 捕获使能 */
}

/***
 * @brief      DCMI, 关闭传输
 * @param      无
 * @retval     无
 */
void dcmi_stop(void)
{
    DCMI->CR &= ~DCMI_CR_CAPTURE; /* DCMI 捕获关闭 */
    while (DCMI->CR & 0X01);    /* 等待传输结束 */
    __HAL_DMA_DISABLE(&g_dma_dcmi_handle); /* 关闭 DMA */
}
```

下面介绍的是 DCMI 中断服务函数（及其回调函数）、DCMI DMA 接收回调函数和 DMA1 数据流 1 中断服务函数，它们的定义分别如下：

```
/***
 * @brief      DCMI 中断服务函数
 * @param      无
 * @retval     无
 */
void DCMI_IRQHandler(void)
{
    HAL_DCMI_IRQHandler(&g_dcmi_handle);
}

/***
 * @brief      DCMI 中断回调服务函数
 * @param      hdcmi:DCMI 句柄
 * @note       捕获到一帧图像处理
 * @retval     无
 */
void HAL_DCMI_FrameEventCallback(DCMI_HandleTypeDef *hdcmi)
{
    jpeg_data_process(); /* jpeg 数据处理 */
    LED1_TOGGLE();      /* LED1 闪烁 */
    g_ov_frame++;

    /* 重新使能帧中断, 因为 HAL_DCMI_IRQHandler() 函数会关闭帧中断 */
    __HAL_DCMI_ENABLE_IT(&g_dcmi_handle, DCMI_IT_FRAME);
}

void (*dcmi_rx_callback)(void); /* DCMI DMA 接收回调函数 */

/***
 * @brief      DMA1 数据流 1 中断服务函数(仅双缓冲模式会用到)
 * @param      无
 * @retval     无
 */
void DMA1_Stream1_IRQHandler(void)
```

```

{
    /* DMA 传输完成 */
    if (_HAL_DMA_GET_FLAG(&g_dma_dcmi_handle, DMA_FLAG_TCIF1_5) != RESET)
    {
        /* 清除 DMA 传输完成中断标志位 */
        _HAL_DMA_CLEAR_FLAG(&g_dma_dcmi_handle, DMA_FLAG_TCIF1_5);
        dcmi_rx_callback();      /* 执行摄像头接收回调函数,读取数据等操作在这里面处理 */
    }
}

```

其中：DCMI_IRQHandler 函数，用于处理帧中断，可以实现帧率统计（需要定时器支持）和 JPEG 数据处理等，实际上当捕获到一帧数据后，调用的是 HAL 库回调函数 HAL_DCMI_FrameEventCallback 进行处理。DMA1_Stream1_IRQHandler 函数，仅用于在使用 RGB 屏的时候，双缓冲存储时，数据的搬运处理（通过 dcmi_rx_callback 函数实现）。

最后还定义两个可以通过 usmart 调试、辅助测试使用的函数 dcmi_set_window 和 dcmi_cr_set 函数。dcmi_set_window 函数用于调节屏幕显示的范围，本实验 LCD 的起始坐标要设置为(0,0)，LCD 显示范围要设置为屏幕最大像素点范围内。dcmi_cr_set 函数用于设置 pclk/hsync/vsync 这三个信号的有效电平。

DCMI 驱动代码就介绍到这里。

47.2.3 SCCB 驱动

本章实验的 SCCB 驱动主要负责向 OV5640 驱动提供配置 OV5640 摄像头的各种函数，SCCB 协议与 IIC 协议十分相似，也可兼容 IIC 协议，因此请读者结合 SCCB 和 IIC 协议的相关文档查看本章实验配套实验源码中 SCCB 的相关驱动文件。本章实验中，SCCB 的驱动代码包括 sccb.c 和 sccb.h 两个文件。

47.2.4 OV5640 驱动

本章实验的 OV5640 驱动主要负责向应用层提供 OV5640 的初始化和各种配置函数，请读者结合正点原子 OV5640 模块(ATK-MC5640)的用户手册查看本章实验配套例程源码中的 OV5640 驱动代码。本章实验中，OV5640 的驱动代码包括 ov5640.c 和 ov5640.h 两个文件。

47.2.5 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint8_t key = 0;
    uint16_t t = 0;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟, 480Mhz */
    delay_init(480);             /* 初始化延时功能 */
    usart_init(115200);          /* 初始化串口 */
    usmart_dev.init(240);         /* 初始化 USMART */
    usart2_init(1500000);        /* 初始化串口 2 波特率为 1500000 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */

    btim_timx_int_init(10000 - 1, 24000 - 1); /* 1 秒钟中断一次，用于统计帧率 */
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "OV5640 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (ov5640_init() != 0)           /* 初始化 OV5640 */

```

```

{
    lcd_show_string(30, 130, 240, 16, 16, "OV5640 ERROR", RED);
    delay_ms(200);
    lcd_fill(30, 130, 239, 170, WHITE);
    delay_ms(200);
    LED0_TOGGLE();
}
lcd_show_string(30, 130, 200, 16, 16, "OV5640 OK", RED);

while (1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        g_ov_mode = 0; /* RGB565 模式 */
        break;
    }
    else if (key == WKUP_PRES)
    {
        g_ov_mode = 1; /* JPEG 模式 */
        break;
    }

    t++;
    if (t == 100) lcd_show_string(30, 150, 230, 16, 16,
                                  "KEY0:RGB565 WK_UP:JPEG", RED); /* 闪烁显示提示信息 */
    if (t == 200)
    {
        lcd_fill(30, 150, 210, 150 + 16, WHITE);
        t = 0;
        LED0_TOGGLE();
    }
    delay_ms(5);
}

if (g_ov_mode == 1)
{
    jpeg_test();
}
else
{
    rgb565_test();
}
}

```

从上面的代码中可以看出，在初始化完 OV5640 摄像头后，并不断地扫描按键，若检测到 KEY0 按键被按下，则进入 RGB565 模式的测试，即将 DCMI 获取到的摄像头图像数据，直接在 LCD 上进行显示，若检测到 WK_UP 按键被按下，则进入 JPEG 模式的测试，即将 DCMI 获取到的摄像头数据，通过 USART2 传输至 PC 上位机。

47.3 下载验证

在完成编译和烧录操作后，先将开发板断电，随后将摄像头模块通过摄像头延长线与开发板进行连接，同时连接好 LCD 模式和 USART2 与 PC 的连接，最后再给开发板供电（在插拔开发板上的接插件和模块时，要求必须断电操作，否则容易烧毁硬件）。程序运行后，可以看到 LCD 上提示按下 KEY0 按键进入 RGB565 模式的测试，按下 WK_UP 按键进入 JPEG 模式的测试。

若此时按下 KEY0 按键，若开发板连接了 LCD 模块，便能够在 LCD 显示屏上看到摄像头采集到的实时画面，并且按下 KEY0 按键可以改变画面的对比度，按下 WK_UP 按键可以改变画面的尺寸。

若此时按下 WK_UP 按键，若开发板的 USART2 通过 USB 转串口模式与 PC 进行连接，并配置好 PC 端的上位机（正点原子 视频传输上位机，读者可前往正点原子资料下载中心下载该

上位机软件)，便可以在上位机软件中看到摄像头采集到的实时画面，并且按下 KEY0 按键可以改变画面的对比度，按下 WK_UP 按键可以改变画面的尺寸。

在进入 RGB565 模式或 JPEG 模式测试后，可以通过串口调试助手查看到摄像头的实时帧率。

第三篇 提高篇

前面的章节介绍了开发 STM32 的工具和方法以及 STM32 的一些基础外设，本章将进入提高篇的学习，本篇的大部分内容都和 STM32 硬件关系不大，主要是介绍各种软件在 STM32 上的移植和使用，例如：内存管理、文件系统、图片解码库、中文字库、手写识别库、T9 拼音输入法、USB 库、OS 等。

本篇依然采用一章一示例的方式进行 STM32 各种高级应用的介绍，但将更多地介绍各种软件库的使用方法，而不会去分析各种软件库的实现方式，相信通过本篇的学习，读者能够使用 STM32 完成更复杂的项目开发。

第四十八章 内存管理实验

本章将介绍正点原子提供的内存管理库的使用，通过使用内存管理库可以提高内存的使用效率，在大型的应用开发中，是必不可少的工具。通过本章的学习，读者将学习到正点原子内存管理库的使用。

本章分为如下几个小节：

- 48.1 硬件设计
- 48.2 程序设计
- 48.3 下载验证

48.1 硬件设计

48.1.1 例程功能

1. 按下 WK_UP 按键可从内存 SRAM 和 CCM 申请内存，并在 LCD 上显示相关信息
2. 按下 KEY0 按键可释放最近一次从 SRAM 和 CCM 申请到的内存
3. LED0 闪烁，提示程序正在运行

48.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15

48.1.3 原理图

本章实验使用的内存管理库为软件库，因此没有对应的连接原理图。

48.2 程序设计

48.2.1 内存管理库的使用

正点原子提供的内存管理库包含两个文件，分别为：malloc.c 和 malloc.h，本章实验配套的实验例程中已经提供了这两个文件，并且已经针对正点原子 M100Z-M7 最小系统板 STM32H750 版进行了移植适配，用户在使用时，仅需将这两个文件添加到自己的工程即可，如下图所示：

```
./Middlewares/MALLOC/
|-- malloc.c
`-- malloc.h
```

图 48.2.1.1 正点原子内存管理库文件

内存管理库中提供了内存管理初始化、申请内存和释放内存等函数，使用非常方便。

在进行内存申请和释放之前，需要使用内存初始化函数对内存进行初始化，内存管理初始化的使用示例，如下所示：

```
#include "./MALLOC/malloc.h"

void example_fun(void)
{
    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMIN);
}
```

初始化内存后，便可以在需要内存的时候申请内存，申请内存函数的使用示例，如下所示：

```
#include "./MALLOC/malloc.h"

void example_fun(void)
{
    uint8_t ptr;

    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMIN);

    /* 申请 1KB 内存 */
    ptr = (uint8_t *)mymalloc(SRAMIN, 1024);

    /* Do something. */
}
```

在内存使用完毕后,需要及时释放内存,否则可能产生内存泄漏,释放内存函数的使用示例,如下所示:

```
#include "./MALLOC/malloc.h"

void example_fun(void)
{
    uint8_t ptr;

    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMIN);

    /* 申请 1KB 内存 */
    ptr = (uint8_t *)mymalloc(SRAMIN, 1024);

    /* Do something. */

    /* 释放无用内存 */
    myfree(SRAMIN, ptr);
}
```

48.2.2 实验应用代码

本章实验的应用代码, 如下所示:

```
int main(void)
{
    uint8_t t = 0;
    uint8_t key;
    uint8_t *p_sramin = NULL;
    uint8_t *p_sramccm = NULL;
    uint32_t tp_sramin = 0;
    uint32_t tp_sramccm = 0;
    uint8_t paddr[32];
    uint16_t memused;

    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 配置时钟, 168MHz */
    delay_init(168); /* 初始化延时 */
    usart_init(115200); /* 初始化串口 */
    led_init(); /* 初始化 LED */
    key_init(); /* 初始化按键 */
    lcd_init(); /* 初始化 LCD */
    my_mem_init(SRAMIN); /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM); /* 初始化 CCM 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:Free", RED);
lcd_show_string(30, 162, 200, 16, 16, "SRAMIN USED:", BLUE);
lcd_show_string(30, 178, 200, 16, 16, "SRAMCCM USED:", BLUE);

while (1)
{
    key = key_scan(0);
    if (key == WKUP_PRES)
    {
        /* 申请内存 */
        p_sramin = mymalloc(SRAMIN, 2048);
        p_sramccm = mymalloc(SRAMCCM, 2048);

        /* 判断内存申请是否成功 */
        if ((p_sramin != NULL) && (p_sramccm != NULL))
        {
            /* 使用申请到的内存 */
            sprintf((char *)p_sramin,
                    "SRAMIN: Malloc Test%03d", t + SRAMIN);
            lcd_show_string(30, 226, 239, 16, 16, (char *)p_sramin, BLUE);
            sprintf((char *)p_sramccm,
                    "SRAMCCM: Malloc Test%03d", t + SRAMCCM);
            lcd_show_string(30, 242, 239, 16, 16, (char *)p_sramccm, BLUE);
        }
        else
        {
            myfree(SRAMIN, p_sramin);
            myfree(SRAMCCM, p_sramccm);
            p_sramin = NULL;
            p_sramccm = NULL;
        }
    }
    else if (key == KEY0_PRES)
    {
        /* 释放内存 */
        myfree(SRAMIN, p_sramin);
        myfree(SRAMCCM, p_sramccm);
        p_sramin = NULL;
        p_sramccm = NULL;
    }

    /* 显示申请到内存的首地址 */
    if ((tp_sramin != (uint32_t)p_sramin) ||
        (tp_sramccm != (uint32_t)p_sramccm))
    {
        tp_sramin = (uint32_t)p_sramin;
        tp_sramccm = (uint32_t)p_sramccm;

        sprintf((char *)paddr,
                "SRAMIN: Addr: 0x%08X", (uint32_t)p_sramin);
        lcd_show_string(30, 194, 239, 16, 16, (char *)paddr, BLUE);
        sprintf((char *)paddr,
                "SRAMCCM: Addr: 0x%08X", (uint32_t)p_sramccm);
        lcd_show_string(30, 210, 239, 16, 16, (char *)paddr, BLUE);
    }
    else if ((p_sramin == NULL) || (p_sramccm == NULL))
    {
        lcd_fill(30, 194, 239, 319, WHITE);
    }

    if (++t == 20)
    {
        t = 0;
    }
}

/* 显示内部 SRAM 使用率 */
```

```
memused = my_mem_perused(SRAMIN);
sprintf((char *)paddr, "%d.%01d%%", memused / 10, memused % 10);
lcd_show_string(30 + 96, 162, 200, 16, 16, (char *)paddr, BLUE);

/* 显示CCM 使用率 */
memused = my_mem_perused(SRAMCCM);
sprintf((char *)paddr, "%d.%01d%%", memused / 10, memused % 10);
lcd_show_string(30 + 104, 178, 200, 16, 16, (char *)paddr, BLUE);

LED0_TOGGLE();
}

delay_ms(10);
}
```

可以看到，本实验的应用代码使用到了两个内存池，分别为内部 SRAM 和 CCM，在完成内存池初始化后，便在 LCD 上实时刷新显示两个内存池的使用量，以及检测按键输入，若检测到 WKUP 按键被按下，则从两个内存池中申请两块内存，并写入测试数据，然后将申请到的两块内存的起始地址即内存中写入的测试数据在 LCD 上进行显示，若检测到 KEY0 按键被按下，则释放最近一次申请的两块内存回对应的内存池中。

48.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时地显示了两个内存池的使用情况，此时按下 WKUP 按键申请内存，可以看到两个内存池的使用量增加，并且 LCD 上显示了申请到的两个内存的起始地址和内存中写入的测试数据，接着按下 KEY0 按键释放内存，可以看到，LCD 上显示的内存信息消失，并且因为内存已经被释放回内存池，因此内存池的使用量较少。

第四十九章 SD 卡实验

本章将介绍使用 STM32H750 驱动 SD 卡进行 SD 卡的识别、读写等操作。通过本章的学习，读者将学习到 SD 卡的使用。

本章分为如下几个小节：

- 49.1 硬件设计
- 49.2 程序设计
- 49.3 下载验证

49.1 硬件设计

49.1.1 例程功能

1. LCD 上显示 SD 卡信息
2. 按下 WKUP 按键可读取 SD 卡第 0 块的数据并通过串口显示
3. LED0 闪烁，提示程序正在运行

49.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
4. USART1
USART1_TX - PA9
USART1_RX - PA10
5. SD
SDIO_D0 - PC8
SDIO_D1 - PC9
SDIO_D2 - PC10
SDIO_D3 - PC11
SDIO_SCK - PC12
SDIO_CMD - PD2

49.1.3 原理图

本章实验使用 SDIO 接口与 SD 卡进行连接，开发板板载了一个 Micro SD 卡座，用于连接 SD 卡，SD 卡与 MCU 的连接原理图，如下图所示：

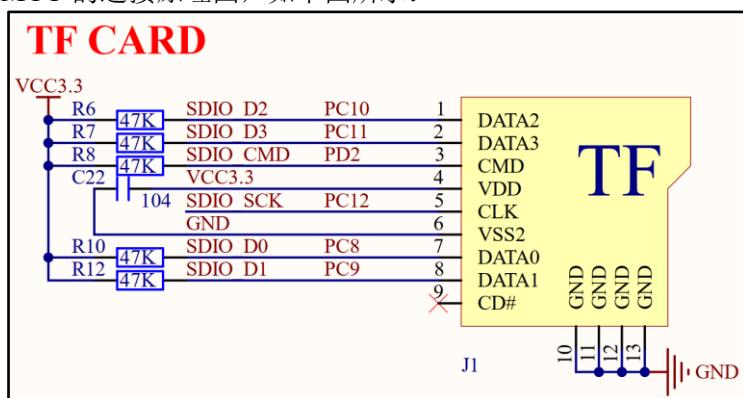


图 49.1.3.1 SD 卡与 MCU 的连接原理图

49.2 程序设计

49.2.1 SD 卡驱动

本章实验的 SD 卡驱动包含两个文件，分别为：sdmmc_sdcard.c 和 sdmmc_sdcard.h，SD 卡的驱动涉及 SD 的通信协议，本章实验配套实验例程的 SD 卡驱动已经根据 SD 协议实现了 SD 卡的初始化和读写等操作，对具体实现过程感兴趣的读者，可结合 SD 的协议查看本章实验配套实验例程的 SD 卡驱动。

49.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
/***
 * @brief      通过串口打印 SD 卡相关信息
 * @param      无
 * @retval     无
 */
void show_sdcard_info(void)
{
    uint64_t card_capacity;          /* SD 卡容量 */
    HAL_SD_CardCIDTypeDef sd_card_cid;

    HAL_SD_GetCardCID(&g_sd_handle, &sd_card_cid);      /* 获取 CID */
    get_sd_card_info(&g_sd_card_info_handle);           /* 获取 SD 卡信息 */

    switch (g_sd_card_info_handle.CardType)
    {
        case CARD_SDSC:
        {
            if (g_sd_card_info_handle.CardVersion == CARD_V1_X)
            {
                printf("Card Type:SDSC V1\r\n");
            }
            else if (g_sd_card_info_handle.CardVersion == CARD_V2_X)
            {
                printf("Card Type:SDSC V2\r\n");
            }
        }
        break;

        case CARD_SDHC_SDXC:
        {
            printf("Card Type:SDHC\r\n");
            break;
        }

        card_capacity = (uint64_t)(g_sd_card_info_handle.LogBlockNbr) *
                        (uint64_t)(g_sd_card_info_handle.LogBlockSize); /* 计算 SD 卡容量 */
        /* 制造商 ID */
        printf("Card ManufacturerID:%d\r\n", sd_card_cid.ManufacturerID);
        /* 卡相对地址 */
        printf("Card RCA:%d\r\n", g_sd_card_info_handle.RelCardAdd);
        printf("LogBlockNbr:%d \r\n",
               (uint32_t)(g_sd_card_info_handle.LogBlockNbr)); /* 显示逻辑块数量 */
        printf("LogBlockSize:%d \r\n",
               (uint32_t)(g_sd_card_info_handle.LogBlockSize)); /* 显示逻辑块大小 */
        /* 显示容量 */
        printf("Card Capacity:%d MB\r\n", (uint32_t)(card_capacity >> 20));
        /* 显示块大小 */
        printf("Card BlockSize:%d\r\n\r\n", g_sd_card_info_handle.BlockSize);
    }
}
/***
```

```
* @brief      测试 SD 卡的读取
* @note       从 secaddr 地址开始, 读取 seccnt 个扇区的数据
* @param      secaddr : 扇区地址
* @param      seccnt  : 扇区数
* @retval     无
*/
void sd_test_read(uint32_t secaddr, uint32_t seccnt)
{
    uint32_t i;
    uint8_t *buf;
    uint8_t sta = 0;
    buf = mymalloc(SRAMIN, seccnt * 512); /* 申请内存, 从 SDRAM 申请内存 */
    sta = sd_read_disk(buf, secaddr, seccnt); /* 读取 secaddr 扇区开始的内容 */

    if (sta == 0)
    {
        printf("SECTOR %d DATA:\r\n", secaddr);
        for (i = 0; i < seccnt * 512; i++)
        {
            printf("%x ", buf[i]); /* 打印 secaddr 开始的扇区数据 */
        }

        printf("\r\nDATA ENDED\r\n");
    }
    else printf("err:%d\r\n", sta);
    myfree(SRAMIN, buf); /* 释放内存 */
}

/**
* @brief      测试 SD 卡的写入
* @note       从 secaddr 地址开始, 写入 seccnt 个扇区的数据
*             慎用!! 最好写全是 0xFF 的扇区, 否则可能损坏 SD 卡.
*
* @param      secaddr : 扇区地址
* @param      seccnt  : 扇区数
* @retval     无
*/
void sd_test_write(uint32_t secaddr, uint32_t seccnt)
{
    uint32_t i;
    uint8_t *buf;
    uint8_t sta = 0;
    buf = mymalloc(SRAMIN, seccnt * 512); /* 从 SDRAM 申请内存 */

    for (i = 0; i < seccnt * 512; i++) /* 初始化写入的数据, 是 3 的倍数. */
    {
        buf[i] = i * 3;
    }
    /* 从 secaddr 扇区开始写入 seccnt 个扇区内容 */
    sta = sd_write_disk(buf, secaddr, seccnt);

    if (sta == 0)
    {
        printf("Write over!\r\n");
    }
    else printf("err:%d\r\n", sta);
    myfree(SRAMIN, buf); /* 释放内存 */
}

int main(void)
{
    uint8_t key;
    uint32_t sd_size;
    uint8_t t = 0;
```

```

uint8_t *buf;                                /* SD 卡容量 */

sys_cache_enable();                         /* 打开 L1-Cache */
HAL_Init();                                 /* 初始化 HAL 库 */
sys_stm32_clock_init(240, 2, 2, 4);        /* 设置时钟, 480Mhz */
delay_init(480);                           /* 延时初始化 */
uart_init(115200);                         /* 串口初始化为 115200 */
usmart_dev.init(240);                      /* 初始化 USMART */
mpu_memory_protection();                   /* 保护相关存储区域 */
led_init();                                /* 初始化 LED */
lcd_init();                                /* 初始化 LCD */
key_init();                                /* 初始化按键 */
my_mem_init(SRAMIN);                       /* 初始化内部内存池(AXI) */
my_mem_init(SRAM12);                       /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
my_mem_init(SRAM4);                        /* 初始化 SRAM4 内存池(SRAM4) */
my_mem_init(SRAMDTCM);                     /* 初始化 DTCM 内存池(DTCM) */
my_mem_init(SRAMITCM);                     /* 初始化 ITCM 内存池(ITCM) */

lcd_show_string(30, 50, 200, 16, 16, "STM32", 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, 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(); /* 红灯闪烁 */
}

/* 打印 SD 卡相关信息 */
show_sdcard_info();

/* 检测 SD 卡成功 */
lcd_show_string(30, 150, 200, 16, 16, "SD Card OK    ", BLUE);
lcd_show_string(30, 170, 200, 16, 16, "SD Card Size:   MB", BLUE);
card_capacity = (uint64_t)(g_sd_card_info_handle.LogBlockNbr) *
    (uint64_t)(g_sd_card_info_handle.LogBlockSize); /* 计算 SD 卡容量 */
/* 显示 SD 卡容量 */
lcd_show_num(30 + 13 * 8, 170, card_capacity >> 20, 5, 16, BLUE);

while (1)
{
    key = key_scan(0);
    if (key == KEY0_PRES)           /* KEY0 按下了 */
    {
        buf = mymalloc(0, 512); /* 申请内存 */
        key = sd_read_disk(buf, 0, 1);
        if (key == 0)              /* 读取 0 扇区的内容 */
        {
            lcd_show_string(30, 190, 200, 16, 16,
                "USART1 Sending Data...", BLUE);
            printf("SECTOR 0 DATA:\r\n");
            for (sd_size = 0; sd_size < 512; sd_size++)
            {
                printf("%x ", buf[sd_size]); /* 打印 0 扇区数据 */
            }
            printf("\r\nDATA ENDED\r\n");
        }
    }
}

```

```
    lcd_show_string(30, 190, 200, 16, 16,
                     "USART1 Send Data Over!", BLUE);
}
else printf("err:%d\r\n", key);
myfree(0, buf); /* 释放内存 */
}
t++;
delay_ms(10);

if (t == 20)
{
    LED0_TOGGLE(); /* 红灯闪烁 */
    t = 0;
}
}
}
```

这里总共 4 个函数：

1、show_sdcard_info 函数

该函数用于从串口输出 SD 卡相关信息，包括：卡类型、制造商 ID、卡相对地址、容量和块大小等信息。

2、sd_test_read 函数

该函数用于测试 SD 卡的读取，通过 USMART 调用，可以指定 SD 卡的任何地址，读取指定个数的扇区数据，将读到的数据，通过串口打印出来，从而验证 SD 卡数据的读取。

3、sd_test_write 函数

该函数用于测试 SD 卡的写入，通过 USMART 调用，可以指定 SD 卡的任何地址，写入指定个数的扇区数据，写入数据自动生成（都是 3 的倍数），写入完成后，在串口打印写入结果。我们可以通过 sd_test_read 函数，来检验写入数据是否正确。**注意：千万别乱写，否则可能把卡写成砖头/数据丢失！写之前，先读取该地址的数据，最好全部是 0xFF 才写（全部 0x00 也行），其他情况最好别写！**

4、main 函数函数

该函数，先初始化相关外设和 SD 卡，初始化成功，则调用 show_sdcard_info 函数，输出 SD 卡相关信息，并在 LCD 上面显示 SD 卡容量。然后进入死循环，如果有按键 KEY0 按下，则通过 SD_ReadDisk 读取 SD 卡的扇区 0（物理磁盘，扇区 0），并将数据通过串口打印出来。这里，我们对上一章学过的内存管理小试牛刀，稍微用了下，以后我们会尽量使用内存管理来设计。

最后，我们将 sd_test_read 和 sd_test_write 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，测试 SD 卡的读写了，方便测试。

49.3 下载验证

在完成编译和烧录操作后，将准备好的 SD 卡插入开发板板载的 SD 卡卡座（请确保 SD 卡中没有有用的数据，或已做好备份），接着便能在 LCD 上看到 SD 卡的容量，以及串口调试助手显示了 SD 卡的卡类型、容量等信息，接着可以按下 WKUP 按键读取 SD 卡扇区 0 的 512 字节数据，可以通过串口调试助手查看读出的 512 字节数据。

第五十章 FATFS 实验

上一章实验中已经成功驱动 SD 卡，并可对 SD 卡进行读写操作，但读写 SD 卡时都是直接读出或写入二进制数据，这样使用起来显得十分不方便，因此本章将介绍 FATFS，FATFS 是一个通用的 FAT 文件系统模块，能够帮助实现文件系统，方便对 SD 卡、NOR Flash 或其他存储介质中数据的管理。通过本章的学习，读者将学习到 FATFS 的基本使用。

本章分为以下几个小节：

- 50.1 硬件设计
- 50.2 程序设计
- 50.3 下载验证

50.1 硬件设计

50.1.1 例程功能

- 1. LCD 上显示 SD 卡容量和剩余容量
- 2. 可通过 USMART 对 SD 卡和 NOR Flash 的文件系统进行操作
- 3. LED0 闪烁，提示程序正在运行

50.1.2 硬件资源

- 1. LED
LED0 - PE5
- 2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
- 3. 按键
WKUP - PA0
- 4. USART1
USART1_TX - PA9
USART1_RX - PA10
- 5. NOR Flash
QSPI FLASH 芯片，连接在 QSPI 上
- 6. SD
SDIO_D0 - PC8
SDIO_D1 - PC9
SDIO_D2 - PC10
SDIO_D3 - PC11
SDIO_SCK - PC12
SDIO_CMD - PD2

50.1.3 原理图

本章实验使用的 FATFS 为软件库，因此没有对应的连接原理图。

50.2 程序设计

50.2.1 FATFS 的使用

FATFS 涉及多个文件，在本章实验的配套实验例程中，如下图所示：

```

./Middlewares/FATFS/source/
|-- 00history.txt
|-- 00readme.txt
|-- diskio.c
|-- diskio.h
|-- ff.c
|-- ff.h
|-- ffconf.h
|-- fffsystem.c
`-- ffunicode.c

```

图 50.2.1.1 实验例程中的 FATFS 源文件

对于不同的硬件，使用 FATFS 仅需修改 diskio.c 和 ffconf.h 文件即可，并且本章实验配套的实验例程中的 FATFS 文件已经针对正点原子 M100Z-M7 最小系统板 STM32H750 版进行了移植适配，用户在使用时，仅需直接将这些文件添加到自己的工程，并根据实际需求适当修改 diskio.c 和 ffconf.h 文件即可。

为了方便读者使用 FATFS，本章实验配套的实验例程另外提供了四个文件，如下图所示：

```

./Middlewares/FATFS/exfun/
|-- exfun.c
|-- exfun.h
|-- fattester.c
`-- fattester.h

```

图 50.2.1.2 实验 FATFS 扩展文件

这几个文件中提供了许多便捷使用和测试 FATFS 的函数，例如：获取文件类型、获取磁盘剩余容量和复制文件夹等。

50.2.2 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint32_t total, free;
    uint8_t t = 0;
    uint8_t res = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    my_mem_init(SRAMIN); /* 初始化内部内存池(AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池(SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池(DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池(ITCM) */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", 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);
LEDO_TOGGLE(); /* LED0 闪烁 */
}

exfun_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 == 0XD) /* 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);
    /* 格式化 FLASH, 1:, 盘符; 0, 使用默认格式化参数 */
    res = f_mkfs("1:", 0, 0, FF_MAX_SS);

    if (res == 0)
    {
        /* 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); /* 格式化完成 */
    }
    else lcd_show_string(30, 150, 200, 16, 16, "Flash Disk Format Error ",
                         RED); /* 格式化失败 */
    delay_ms(1000);
}

lcd_fill(30, 150, 240, 150 + 16, WHITE); /* 清除显示 */

while (exfun_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);
    LEDO_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);
/* 显示 SD 卡总容量 MB */
lcd_show_num(30 + 8 * 14, 170, total >> 10, 5, 16, BLUE);
/* 显示 SD 卡剩余容量 MB */
lcd_show_num(30 + 8 * 14, 190, free >> 10, 5, 16, BLUE);

while (1)
{
    t++;
    delay_ms(200);
    LEDO_TOGGLE(); /* LED0 闪烁 */
}
}

```

可以看到，本实验的应用代码中，使用函数 exfun_init() 函数为使用和测试 FATFS 完成一些必要的初始化后，便挂载了 SD 卡和 NOR Flash，并且在 NOR Flash 上没有文件系统的情况下对其进行格式化，随后借助函数 exfun_get_free() 获取 SD 卡的总容量和剩余容量，并且将其在 LCD 上进行显示。

本实验还使用到了 USMART 调试组件，方便使用串口调试助手测试 FATFS，usmart_config.c 文件中添加的函数，如下所示：

```
/* 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 *)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_fmkfs, "uint8_t mf_fmkfs(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)",
};

};
```

这么一来便可以通过串口调试助手借助 USMART 调试组件对 FATFS 进行测试。

50.3 下载验证

在完成编译和烧录操作后，将准备好的 SD 卡插入开发板板载的 SD 卡卡座（请确保 SD 卡中没有有用的数据，或已做好备份），接着若 SD 卡和 NOR Flash 都初始化并挂载和格式化成功，便能在 LCD 上看到 SD 卡的总容量和剩余容量。

接下来便可以通过串口调试助手借助 USMART 对 FATFS 进行测试，例如：“mf_scan_files(“0:”）” 和 “mf_scan_files(“1:”）” 可分别查看 SD 卡和 NOR Flash 中文件系统根目录下的目录结构等。

第五十一章 汉字显示实验

本章将介绍正点原子提供的字库管理库的使用，通过使用字库能够在 LCD 或其他显示设备上显示中文字符，对应中文应用的开发，是一个很有用的工具。通过本章的学习，读者将学习到正点原子字库管理库的使用。

本章分为如下几个小节：

- 51.1 硬件设计
- 51.2 程序设计
- 51.3 下载验证

51.1 硬件设计

51.1.1 例程功能

- 1. 自动检查并在无字库时自动更新字库
- 2. LCD 上不断刷新显示汉字
- 3. LED0 闪烁，提示程序正在运行

51.1.2 硬件资源

- 1. LED
LED0 - PE5
- 2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
- 3. 按键
WKUP - PA0
- 4. NOR Flash
QSPI FLASH 芯片，连接在 QSPI 上
- 5. SD
SDIO_D0 - PC8
SDIO_D1 - PC9
SDIO_D2 - PC10
SDIO_D3 - PC11
SDIO_SCK - PC12
SDIO_CMD - PD2

51.1.3 原理图

本章实验使用的字库管理库为软件库，因此没有对应的连接原理图。

51.2 程序设计

51.2.1 字库管理库的使用

正点原子提供的字库管理库包含了四个文件，分别为：fonts.c、fonts.h、text.c 和 text.h，本章实验配套实验例程中已经提供了这四个文件，并且已经针对正点原子 M100Z-M7 最小系统板 STM32H750 版软硬件进行了移植适配，用户在使用时，仅需将这四个文件添加到自己的工程中即可，如下图所示：

```
./Middlewares/TEXT/
|-- fonts.c
|-- fonts.h
|-- text.c
\-- text.h
```

图 51.2.1.1 正点原子字库管理库文件

字库管理库中 fonts.c 和 fonts.h 两个文件提供了字库更新和初始化的函数，test.c 和 test.h 文件中提供了在 LCD 上显示中文字符的函数。

字库管理库在显示中文字符至 LCD 上时会使用 NOR Flash 中的中文字库，因此需要确保 NOR Flash 中的中文字库无误，若 NOR Flash 中没有中文字库的数据，那么在进行字库初始化时就会提示失败，这时就需要使用字库管理库中提供的字库更新函数更新 NOR Flash 中的中文字库数据，更新字库是读取 SD 卡中的字库文件将其写入 NOR Flash，因此需确保 SD 卡中有对应的中文字库文件。

本章实验所需的中文字库文件可在 A 盘→5, SD 卡根目录文件→SYSTEM→FONT 中找到，建议将 A 盘→5, SD 卡根目录文件中的所有文件按照该目录的目录结构复制到 SD 卡中，方便后续实验的使用。

51.2.2 实验应用代码

本章实验的应用代码，如下所示：

```

int main(void)
{
    uint32_t fontcnt;
    uint8_t i, j;
    uint8_t fontx[2]; /* GBK 码 */
    uint8_t key, t;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    my_mem_init(SRAMIN); /* 初始化内部内存池(AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池(SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池(DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池(ITCM) */
    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 */
    {
        UPD:
        lcd_clear(WHITE); /* 清屏 */
        lcd_show_string(30, 30, 200, 16, 16, "STM32", 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 STM32 开发板", 16, 0, RED);
text_show_string(30, 50, 200, 16, "GBK 字库测试程序", 16, 0, RED);
text_show_string(30, 70, 200, 16, "ATOM@ALIENTEK", 16, 0, RED);
text_show_string(30, 90, 200, 16, "按 KEY0,更新字库", 16, 0, RED);

text_show_string(30, 130, 200, 16, "内码高字节:", 16, 0, BLUE);
text_show_string(30, 150, 200, 16, "内码低字节:", 16, 0, BLUE);
text_show_string(30, 170, 200, 16, "汉字计数器:", 16, 0, BLUE);

text_show_string(30, 200, 200, 32, "对应汉字为:", 32, 0, BLUE);
text_show_string(30, 232, 200, 24, "对应汉字为:", 24, 0, BLUE);
text_show_string(30, 256, 200, 16, "对应汉字(16*16)为:", 16, 0, BLUE);
text_show_string(30, 272, 200, 12, "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, 130, i, 3, 16, BLUE); /* 显示内码高字节 */
        /* GBK 内码低字节范围为 0X40~0X7E, 0X80~0XFE */
        for (j = 0x40; j < 0xfe; j++)
        {
            if (j == 0x7f) continue;
            fontcnt++;
            lcd_show_num(118, 150, j, 3, 16, BLUE); /* 显示内码低字节 */
            lcd_show_num(118, 170, fontcnt, 5, 16, BLUE); /* 汉字计数显示 */
            fontx[1] = j;
            text_show_font(30 + 176, 200, fontx, 32, 0, BLUE);
            text_show_font(30 + 132, 232, fontx, 24, 0, BLUE);
            text_show_font(30 + 144, 256, fontx, 16, 0, BLUE);
            text_show_font(30 + 108, 272, fontx, 12, 0, BLUE);
            t = 200;

            while (t--) /* 延时,同时扫描按键 */
            {
                delay_ms(1);
                key = key_scan(0);

                if (key == KEY0_PRES)
                {
                    goto UPD; /* 跳转到 UPD 位置(强制更新字库) */
                }
            }
            LED0_TOGGLE();
        }
    }
}
}

```

从上面的代码中可以看出，本实验的应用代码中，在完成文件系统等一系列初始化后，便调用函数 fonts_init() 初始化字库管理库，若初始化失败则说明 NOR Flash 中没有对应的字库数据，

因此需要进行更新字库，更新字库时，会从 SD 卡中读入字库数据并将其写入 NOR Flash 中，因此需要在 SD 中提前准备好字库文件。

当 NOR Flash 有了字库数据后，便在 LCD 上使用中文显示本实验的实验信息，并不断的显示中文字符，同时检测按键输入，若检测到 WKUP 按键被按下，则强制进行字库更新。

51.3 下载验证

在完成编译和烧录操作后，将根目录存放了 A 盘→5，SD 卡根目录文件中文件的 SD 卡插入开发板板载的 SD 卡卡座后，若 NOR Flash 中没有字库数据，则能够看到 LCD 显示了正在更新字库的提示，更新完成后，便可以在 LCD 上看到中文显示的实验信息和不断刷新显示的中文字符，此时可以按下 WKUP 按键强制更新字库。

第五十二章 图片显示实验

本章将介绍使用 STM32H750 软件解码 BMP、JPG 和 GIF 等格式的图片，并在 LCD 进行显示。通过本章的学习，读者将学习到图片解码库的使用。

52.1 硬件设计

52.1.1 例程功能

1. LCD 上不断刷新显示图片
2. 按下 WKUP 按键可切换至上一个图片
3. 按下 KEY0 按键可切换至下一个图片
4. LED0 闪烁，提示程序正在运行

52.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
4. NOR Flash
QSPI FLASH 芯片，连接在 QSPI 上
5. SD
 - SDIO_D0 - PC8
 - SDIO_D1 - PC9
 - SDIO_D2 - PC10
 - SDIO_D3 - PC11
 - SDIO_SCK - PC12
 - SDIO_CMD - PD2

52.1.3 原理图

本章实验使用的图片解码库为软件库，因此没有对应的连接原理图。

52.2 程序设计

52.2.1 图片解码库的使用

正点原子提供的图片解码库包含了九个文件，分别为：piclib.c、piclib.h、bmp.c、bmp.h、gif.c、gif.h、tjpgd.c、tjpgd.h、tjpgdcnf.h，这几个文件的描述，如下表所示：

文件	描述
piclib.c piclib.h	正点原子图片解码库文件，主要提供图片解码库和画图的初始化和根据文件类型自动解码并画图的函数等
bmp.c bmp.h	正点原子 BMP 图片编解码库，主要用于解码 BMP 图片文件
gif.c gif.h	正点原子 GIF 图片解码库，主要用于解码 GIF 图片文件
tjpgd.c	TjpgDec (JPEG 图片解码库) 的源文件和配置文件，为了方

tjpd.h tjpdconf.h	便使用，添加了用于正点原子图片解码库的函数
----------------------	-----------------------

表 52.2.1.1 图片解码库各文件描述

以上图片解码库的九个文件，在本章实验配套的实验例程中都已提供，并且针对正点原子 M100Z-M7 最小系统板 STM32H750 版软硬件进行了移植适配，用户在使用时，仅需将这九个文件添加到自己的工程即可，如下所示：

```
./Middlewares/PICTURE/
|-- bmp.c
|-- bmp.h
|-- gif.c
|-- gif.h
|-- piclib.c
|-- piclib.h
|-- tjpd.c
`-- tjpd.h
`-- tjpdconf.h
```

图 52.2.1.1 正点原子图片解码库文件

52.2.2 实验应用代码

本章实验的应用代码，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 */

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    my_mem_init(SRAMIN); /* 初始化内部内存池 (AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池 (SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池 (SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池 (DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池 (ITCM) */
    exfun_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, "STM32", 16, 0, RED);
text_show_string(30, 70, 200, 16, "图片显示 实验", 16, 0, RED);
text_show_string(30, 90, 200, 16, "KEY0:NEXT KEY1:PREV", 16, 0, RED);
text_show_string(30, 110, 200, 16, "KEY_UP:PAUSE", 16, 0, RED);
text_show_string(30, 130, 200, 16, "ATOM@ALIENTEK", 16, 0, RED);

while (f_opendir(&picdir, "0:/PICTURE")) /* 打开图片文件夹 */
{
    text_show_string(30, 170, 240, 16, "PICTURE 文件夹错误!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 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, 170, 240, 16, "没有图片文件!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 240, 186, WHITE); /* 清除显示 */
    delay_ms(200);
}

picfileinfo = (FILINFO *)mymalloc(SRAMIN, sizeof(FILINFO)); /* 申请内存 */
pname = mymalloc(SRAMIN, 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, 170, 240, 16, "内存分配失败!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 240, 186, WHITE); /* 清除显示 */
    delay_ms(200);
}

/* 记录索引 */
res = f_opendir(&picdir, "0:/PICTURE"); /* 打开目录 */
if (res == FR_OK)
{
    curindex = 0; /* 当前索引为 0 */

    while (1) /* 全部查询一遍 */
    {
        temp = picdir.dptr; /* 记录当前 dptr 偏移 */
        res = f_readdir(&picdir, picfileinfo); /* 读取目录下的一个文件 */
        /* 错误了/到末尾了, 退出 */
        if (res != FR_OK || picfileinfo->fname[0] == 0) break;

        res = exfun_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 */
    /* 错误了/到末尾了,退出 */
    if (res != FR_OK || picfileinfo->fname[0] == 0)break;

    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根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。

52.3 下载验证

在完成编译和烧录操作后，将根目录存放了 A 盘→5，SD 卡根目录文件中文件的 SD 卡插入开发板板载的 SD 卡卡座后，便能看到 LCD 上显示了 SD 卡 PICTURE 文件夹中的图片，并且按下 WKUP 按键或 KEY0 按键可以切换 LCD 显示 SD 卡 PICTURE 文件夹中的上一张或下一张图片。

第五十三章 硬件 JPEG 解码实验

上一章，我们学习了图片解码，学会了使用软件解码显示 bmp/jpg/jpeg/gif 等格式的图片，但是软件解码速度都比较慢，本章我们将学习如何使用 STM32H750 自带的硬件 JPEG 编解码器，实现对 JPG/JPEG 图片的硬解码，从而大大提高解码速度。

本章分为以下几个小节：

53.2 硬件设计

53.3 程序设计

53.4 下载验证

53.1 硬件设计

53.1.1 例程功能

1、本实验开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY1 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，LED1 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。

注意：本例程的实验现象，同上一章（图片显示实验）完全一模一样，唯一的区别，就是 JPEG 解码速度（要求图片分辨率小于等于 LCD 分辨率）变快了很多。

对比上一章的图片显示，大家可以利用 USMART 测试同一张 JPEG 图片，软件解码和硬件解码的时间差距。本实验也可以通过 USMART 调用 ai_load_picfile 和 minibmp_decode 解码任意指定路径的图片。

2、LED0 闪烁，提示程序运行。

53.1.2 硬件资源

1. LED

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. 按键

WKUP - PA0

KEY0 - PA15

5. NOR Flash

QSPI FLASH 芯片，连接在 QSPI 上

6. SD

SDIO_D0 - PC8

SDIO_D1 - PC9

SDIO_D2 - PC10

SDIO_D3 - PC11

SDIO_SCK - PC12

SDIO_CMD - PD2

7. 硬件 JPEG 解码内核(STM32H750 自带)

53.1.3 原理图

本章实验使用的图片解码库为软件库，因此没有对应的连接原理图。

53.2 程序设计

53.2.1 硬件 JPEG 解码 JPG/JPEG 的简要步骤

为了提高速度，我们直接操作寄存器，没有使用 HAL 库提供的函数。

所以 HAL 库的内容就讲解了，接下来，我们看看在 DMA 模式下，使用 STM32H7 的硬件 JPEG 解码 JPG/JPEG 的简要步骤：

① 初始化硬件 JPEG 内核。

首先，我们通过设置 AHB3ENR 的 bit5 位为 1，使能硬件 JPEG 内核时钟，然后通过 JPEG_CR 寄存器的 JCEN 位，使能硬件 JPEG。通过清零 JPEG_CONFR0 寄存器的 START 位，停止 JPEG 编解码进程。通过设置 JPEG_CONFR1 寄存器的 HDR 位，使能 JPEG 头解码。最后设置 JPEG 中断服务函数的中断优先级，完成初始化硬件 JPEG 内核过程。

② 初始化硬件 JPEG 解码。

在初始化硬件 JPEG 内核以后，我们配置 JPEG 内核工作在 JPEG 解码模式。通过设置 JPEG_CONFR1 寄存器的 DE 位，使能 JPEG 解码模式。然后设置 JPEG_CR 寄存器的 OFF、IFF、HPDIE、EOCIE 等位，清空输出/输入 FIFO，并开启 JPEG 头解码完成和 JPEG 解码完成中断。最后，设置 JPEG_CONFR0 寄存器的 START 位，启动 JPEG 解码进程。

注意：此时我并未开启 JPEG 的输入和输出 MDMA，只要我们不往输入 FIFO 写入数据，JPEG 内核就一直处于等待数据输入状态。

③ 配置硬件 JPEG 输入/输出 MDMA。

这一步，我们将配置 JPEG 的输入 MDMA 和输出 MDMA，分别负责 JPEG 输入 FIFO 和输出 FIFO 的数据传输。对于输入 MDMA，目标地址为 JPEG_DIR 寄存器地址，源地址为一片内存区域，利用输入 MDMA 实现 JPEG 输入 FIFO 数据的自动填充。对于输出 MDMA，目标地址为一片内存区域，源地址为 JPEG_DOR 寄存器地址，利用输出 MDMA 实现 JPEG 输出 FIFO 数据自动搬运到对应内存区域。对于输入 MDMA 和输出 MDMA，我们都需要开启传输完成中断，并设置相关中断服务函数。在传输完成中断里面，实现对输入输出数据的处理。

④ 编写相关中断服务函数，启动 MDMA。

我们总共开启了 4 个中断：JPEG 头解码完成中断、JPEG 解码完成中断、输入 MDMA 传输完成中断和输出 MDMA 传输完成中断。前两个和后两个中断分别共用一个中断服务函数，所以我们总共只需要编写 2 个中断服务函数。另外，我们采用回调函数的方式，对数据进行处理，总共需要编写 4 个回调函数，分别对应 4 个中断产生时的数据处理。在配置完这些以后，启动 MDMA，开始执行 JPEG 解码。

注意：输出 MDMA 的配置和启动，我们放在 JPEG 头解码完成中断回调函数里的，因为输出 YCbCr 数据的多少和单次输出行数，得根据抽样方式进行不同的设置，因此我们必须先等到解析完 JPEG 头以后，再来配置输出 MDMA。

⑤ 处理 JPEG 数据输出数据，执行 YUV→RGB 转换，并送 LCD 显示。

最后，在主循环里面，根据输入 MDMA 和输出 MDMA 的数据处理情况，持续从源文件读取 JPEG 数据流，并利用 DMA2D，将硬件 JPEG 解码完成的 YCbCr (YUV) 数据流转换成 RGB 格式。最后，在完成一张 JPEG 解码之后，将 RGB 数据直接一次性显示到 LCD 屏幕上，实现图片显示。

53.2.2 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。JPEGCODEC 驱动源码包括两个文件：jpegcodec.c 和 jpegcodec.h。

jpegcodec.h 头文件定义了两个结构体和一些宏定义，下面重点围绕 jpeg_codec_typedef 这个结构体介绍一下，jpegcodec.h 的相关内容定义如下：

#define JPEG_DMA_INBUF_LEN	4096	/* 单个 MDMA IN BUF 的大小 */
#define JPEG_DMA_INBUF_NB	10	/* MDMA IN BUF 的个数 */
#define JPEG_DMA_OUTBUF_NB	2	/* MDMA OUT BUF 的个数 */

```

/* JPEG 数据缓冲结构体 */
typedef struct
{
    uint8_t sta;          /* 状态:0,无数据;1,有数据 */
    uint8_t *buf;         /* JPEG 数据缓冲区 */
    uint16_t size;        /* JPEG 数据长度 */
} jpeg_databuf_type;

#define JPEG_STATE_NOHEADER      0      /* HEADER 未读取,初始状态 */
#define JPEG_STATE_HEADEROK      1      /* HEADER 读取成功 */
#define JPEG_STATE_FINISHED      2      /* 解码完成 */
#define JPEG_STATE_ERROR         3      /* 解码错误 */

#define JPEG_YCBCR_COLORSPACE    JPEG_CONFRL_COLORSPACE_0
#define JPEG_CMYK_COLORSPACE     JPEG_CONFRL_COLORSPACE

/* jpeg 编解码控制结构体 */
typedef struct
{
    JPEG_ConfTypeDef Conf;           /* 当前 JPEG 文件相关参数 */
    jpeg_databuf_type inbuf[JPEG_DMA_INBUF_NB];   /* MDMA IN buf */
    jpeg_databuf_type outbuf[JPEG_DMA_OUTBUF_NB];  /* MDMA OUT buf */
    volatile uint8_t inbuf_read_ptr;    /* MDMA IN buf 当前读取位置 */
    volatile uint8_t inbuf_write_ptr;   /* MDMA IN buf 当前写入位置 */
    volatile uint8_t indma_pause;      /* 输入 MDMA 暂停状态标识 */
    volatile uint8_t outbuf_read_ptr;   /* MDMA OUT buf 当前读取位置 */
    volatile uint8_t outbuf_write_ptr;  /* MDMA OUT buf 当前写入位置 */
    volatile uint8_t outdma_pause;     /* 输入 MDMA 暂停状态标识 */
    volatile uint8_t state;           /* 解码状态;0,未识别到 Header;1,识别到了 Header;2,解码完成*/
}

/* YUV 输出的字节数,使得完成一次 DMA2D YUV2RGB 转换,刚好是图片宽度的整数倍
 * YUV420 图片,每个像素占 1.5 个 YUV 字节,每次输出 16 行,yuvblk_size=图片宽度*16*1.5
 * YUV422 图片,每个像素占 2 个 YUV 字节和 RGB565 一样,每次输出 8 行,yuvblk_size=图片宽度*8*2
 * YUV444 图片,每个像素占 3 个 YUV 字节,每次输出 8 行,yuvblk_size=图片宽度*8*3 */
uint32_t yuvblk_size;

/* 每个 YUV 块输出像素的高度,对于 YUV420,为 16,对于 YUV422/YUV444 为 8 */
uint16_t yuvblk_height;
uint16_t yuvblk_curheight;           /* 当前输出高度,0~分辨率高度 */
} jpeg_codec_TYPEDEF;

```

该结构体用于控制整个 JPEG 解码，下面分别介绍一下它的成员：

Conf 用于存储当前 JPEG 文件的一些相关信息，其结构体类型定义如下：

```

/* JPEG 文件信息结构体 */
typedef struct
{
    u8 ColorSpace;           /* 图像的颜色空间: gray-scale/YCBCR/RGB/CMYK */
    /* YCBCR/CMYK 颜色空间的色度抽样情况: 0: 4:4:4; 1: 4:2:2; 2: 4:1:1; 3: 4:2:0 */
    u8 ChromaSubsampling;
    u32 ImageHeight;         /* 图像高度 */
    u32 ImageWidth;          /* 图像宽度 */
    u8 ImageQuality;         /* 图像编码质量:1~100 */
} JPEG_ConfTypeDef;

```

inbuf 和 **outbuf** 分别代表输入 MDMA FIFO 和输出 MDMA FIFO，使用 FIFO 来处理 MDMA 数据，可以提高读写效率。注意：这里的输入 MDMA FIFO 和输出 MDMA FIFO 同 JPEG 的输入 FIFO 和输出 FIFO 是不一样的，要注意区分。通过 **JPEG_DMA_INBUF_NB** 和 **JPEG_DMA_OUTBUF_NB** 宏定义，我们可以修改输入 DMA FIFO 和输出 DMA FIFO 的深度。

其它，还有输入输出 MDMA FIFO 的读写位置、暂停状态、解码状态、单次 YUV 输出字节数、单次输出图像高度和当前输出高度等信息。

下面开始介绍 **jpegcodec.c** 文件，首先是 JPEG 规范(ISO/IEC 10918-1 标准)的样本量化表，其

定义如下：

```
/* JPEG 规范(ISO/IEC 10918-1 标准)的样本量化表
 * 获取 JPEG 图片质量时需要用到
 */
const uint8_t JPEG_LUM_QuantTable [JPEG_QUANT_TABLE_SIZE] =
{
    16, 11, 10, 16, 24, 40, 51, 61, 12, 12, 14, 19, 26, 58, 60, 55,
    14, 13, 16, 24, 40, 57, 69, 56, 14, 17, 22, 29, 51, 87, 80, 62,
    18, 22, 37, 56, 68, 109, 103, 77, 24, 35, 55, 64, 81, 104, 113, 92,
    49, 64, 78, 87, 103, 121, 120, 101, 72, 92, 95, 98, 112, 100, 103, 99
};

const uint8_t JPEG_ZIGZAG_ORDER [JPEG_QUANT_TABLE_SIZE] =
{
    0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18, 11, 4, 5,
    12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13, 6, 7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36, 29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46, 53, 60, 61, 54, 47, 55, 62, 63
};
```

这两个数组在后面的 jpeg_get_quality 函数，获取 JPEG 图片质量时需要用到。

下面介绍的是 JPEG 硬件解码输入 MDMA 配置函数，其定义如下：

```
/**
 * @brief      JPEG 硬件解码输入 MDMA 配置
 * @param      meminaddr : JPEG 输入 MDMA 存储器地址
 * @param      meminsize  : 输入 MDMA 数据长度,0~262143,以字节为单位
 * @retval     无
 */
void jpeg_in_dma_init(uint32_t meminaddr, uint32_t meminsize)
{
    uint32_t regval = 0;
    uint32_t addrmask = 0;
    RCC->AHB3ENR |= 1 << 0;           /* 使能 MDMA 时钟 */
    MDMA_Channel17->CCR = 0;           /* 输入 MDMA 清零 */

    while (MDMA_Channel17->CCR & 0X01); /* 等待 MDMA_Channel17 关闭完成 */
    MDMA_Channel17->CIFCR = 0X1F;       /* 中断标志清零 */
    MDMA_Channel17->CCR |= 1 << 2;     /* CTCIE=1,使能通道传输完成中断 */
    MDMA_Channel17->CCR |= 2 << 6;     /* PL[1:0]=2,高优先级 */
    MDMA_Channel17->CBNDTR = meminsize; /* 传输长度为 meminsize */
    MDMA_Channel17->CDAR = (uint32_t)&JPEG->DIR; /* 目标地址为:JPEG->DIR */
    MDMA_Channel17->CSAR = meminaddr;   /* meminaddr 作为源地址 */
    regval = 0 << 28;                  /* TRGM[1:0]=0,每个 MDMA 请求触发一次 buffer 传输 */
    regval |= 1 << 25;                 /* PKE=1,打包使能 */
    regval |= (32 - 1) << 18;          /* TLEN[6:0]=31,buffer 传输长度为 32 字节 */
    regval |= 4 << 15;                 /* DBURST[2:0]=4,目标突发传输长度为 16 */
    regval |= 4 << 12;                 /* SBURST[2:0]=4,源突发传输长度为 16 */
    regval |= 0 << 8;                  /* SINCOS[1:0]=0,源地址变化单位为 8 位(字节) */
    regval |= 2 << 6;                  /* DSIZE[1:0]=2,目标位宽为 32 位 */
    regval |= 0 << 4;                  /* SSIZE[1:0]=0,源位宽为 8 位 */
    regval |= 0 << 2;                  /* DINC[1:0]=0,目标地址固定 */
    regval |= 2 << 0;                  /* SINC[1:0]=2,源地址自增 */
    MDMA_Channel17->CTCR = regval;    /* 设置 CTCR 寄存器 */
    /* MDMA 的硬件触发通道 17 触发 inmdma,通道 17=JPEG input FIFO threshold
     * 详见<STM32H7xx 参考手册 V7 (英文版).pdf>577 页,table 95 */
    MDMA_Channel17->CTBR = 17 << 0;
    addrmask = meminaddr & 0xFF000000; /* 获取掩码 */
    /* 使用 AHBS 总线访问 DTCM/ITCM */
    if (addrmask == 0X20000000 || addrmask == 0) MDMA_Channel17->CTBR |= 1 << 16;
    HAL_NVIC_SetPriority(MDMA_IRQn, 2, 3); /* 设置中断优先级, 抢占优先级 2, 子优先级 3 */
    HAL_NVIC_EnableIRQ(MDMA_IRQn);        /* 开启 MDMA 中断 */
}
```

该函数用于初始化 JPEG 输入 FIFO 的 MDMA 通道，使用 buffer 传输，单次触发传输 32 字节，满足 JPEG 输入 FIFO 的传输要求。

下面介绍的是 JPEG 硬件解码输出 MDMA 配置函数，其定义如下：

```
/***
 * @brief      JPEG 硬件解码输出 MDMA 配置
 * @param      memoutaddr : JPEG 输出 MDMA 存储器地址
 * @param      memoutsize : 输出 MDMA 数据长度, 0~262143, 以字节为单位
 * @retval     无
 */
void jpeg_out_dma_init(uint32_t memoutaddr, uint32_t memoutsize)
{
    uint32_t regval = 0;
    uint32_t addrmask = 0;
    RCC->AHB3ENR |= 1 << 0; /* 使能 MDMA 时钟 */
    MDMA_Channel16->CCR = 0; /* 输出 MDMA 清零 */

    while (MDMA_Channel16->CCR & 0X01); /* 等待 MDMA_Channel16 关闭完成 */

    MDMA_Channel16->CIFCR = 0x1F; /* 中断标志清零 */
    MDMA_Channel16->CCR |= 3 << 6; /* PL[1:0]=2, 最高优先级 */
    MDMA_Channel16->CCR |= 1 << 2; /* CTCIE=1, 使能通道传输完成中断 */
    MDMA_Channel16->CBNDTR = memoutsize; /* 传输长度为 memoutsize */
    MDMA_Channel16->CDAR = memoutaddr; /* 目标地址为:memoutaddr */
    MDMA_Channel16->CSAR = (uint32_t)&JPEG->DOR; /* JPEG->DOR 作为源地址 */
    regval = 0 << 28; /* TRGM[1:0]=0, 每个 MDMA 请求触发一次 buffer 传输 */
    regval |= 1 << 25; /* PKE=1, 打包使能 */
    regval |= (32 - 1) << 18; /* TLEN[6:0]=31, buffer 传输长度为 32 字节 */
    regval |= 4 << 15; /* DBURST[2:0]=4, 目标突发传输长度为 16 */
    regval |= 4 << 12; /* SBURST[2:0]=4, 源突发传输长度为 16 */
    regval |= 0 << 10; /* DINCOS[1:0]=0, 目标地址变化单位为 8 位(字节) */
    regval |= 0 << 6; /* DSIZE[1:0]=0, 目标位宽为 8 位 */
    regval |= 2 << 4; /* SSIZE[1:0]=2, 源位宽为 32 位 */
    regval |= 2 << 2; /* DINC[1:0]=2, 目标地址自增 */
    regval |= 0 << 0; /* SINC[1:0]=0, 源地址固定 */
    MDMA_Channel16->CTCR = regval; /* 设置 CTCR 寄存器 */
    /* MDMA 的硬件触发通道 19 触发 outmdma, 通道 19=JPEG output FIFO threshold */
    MDMA_Channel16->CTBR = 19 << 0;

    /* 详见< STM32H7xx 参考手册_v7 (英文版).pdf >577 页, table 95 */
    addrmask = memoutaddr & 0xFF000000; /* 获取掩码 */
    /* 使用 AHBS 总线访问 DTCM/ITCM */
    if (addrmask == 0x20000000 || addrmask == 0) MDMA_Channel16->CTBR |= 1 << 17;

    HAL_NVIC_SetPriority(MDMA IRQn, 2, 3); /* 设置中断优先级, 抢占优先级 2, 子优先级 3 */
    HAL_NVIC_EnableIRQ(MDMA IRQn); /* 开启 MDMA 中断 */
}
```

该函数用于初始化 JPEG 输出 FIFO 的 MDMA 通道，使用 buffer 传输，单次触发传输 32 字节，满足 JPEG 输出 FIFO 的传输要求。

下面介绍一些中断处理函数和回调函数，它们的定义分别如下：

```
void (*jpeg_in_callback)(void); /* JPEG MDMA 输入回调函数 */
void (*jpeg_out_callback)(void); /* JPEG MDMA 输出 回调函数 */
void (*jpeg_eoc_callback)(void); /* JPEG 解码完成 回调函数 */
void (*jpeg_hdp_callback)(void); /* JPEG Header 解码完成 回调函数 */

/***
 * @brief      MDMA 中断服务函数
 * @note       处理硬件 JPEG 解码时输入/输出数据流
 * @param      无
 * @retval     无

```

```

/*
void MDMA_IRQHandler(void)
{
    if (MDMA_Channel7->CISR & (1 << 1)) /* CTCIF, 通道 7 传输完成(输入) */
        if (MDMA_Channel7->CISR & (1 << 1)) /* CTCIF, 通道 7 传输完成(输入) */
    {
        MDMA_Channel7->CIFCR |= 1 << 1; /* 清除通道传输完成中断 */
        JPEG->CR &= ~(0X7E); /* 关闭 JPEG 中断, 防止被打断 */
        jpeg_in_callback(); /* 执行输入回调函数, 继续读取数据 */
        JPEG->CR |= 3 << 5; /* 使能 EOC 和 HPD 中断 */
    }
    if (MDMA_Channel6->CISR & (1 << 1)) /* CTCIF, 通道 6 传输完成(输出) */
    {
        MDMA_Channel6->CIFCR |= 1 << 1; /* 清除通道传输完成中断 */
        JPEG->CR &= ~(0X7E); /* 关闭 JPEG 中断, 防止被打断 */
        jpeg_out_callback(); /* 执行输出回调函数, 将数据转换成 RGB */
        JPEG->CR |= 3 << 5; /* 使能 EOC 和 HPD 中断 */
    }
}

/**
 * @brief      JPEG 解码中断服务函数
 * @param      无
 * @retval     无
 */
void JPEG_IRQHandler(void)
{
    if (JPEG->SR & (1 << 6)) /* JPEG Header 解码完成 */
    {
        jpeg_hdp_callback();
        JPEG->CR &= ~(1 << 6); /* 禁止 JPEG Header 解码完成中断 */
        JPEG->CFR |= 1 << 6; /* 清除 HPDF 位(header 解码完成位) */
    }

    if (JPEG->SR & (1 << 5)) /* JPEG 解码完成 */
    {
        jpeg_dma_stop();
        jpeg_eoc_callback();
        JPEG->CFR |= 1 << 5; /* 清除 EOC 位(解码完成位) */
        MDMA_Channel6->CCR &= ~(1 << 0); /* 关闭 MDMA 通道 6 */
        MDMA_Channel7->CCR &= ~(1 << 0); /* 关闭 MDMA 通道 7 */
    }
}

```

MDMA_IRQHandler 中断服务函数，用于处理输入/输出 FIFO MDMA 的传输完成中断，当发生输入 FIFO MDMA 传输完成中断时，调用 jpeg_in_callback 回调函数，处理输入 FIFO MDMA 传输完成事务。当发生输出 FIFO MDMA 传输完成中断时，调用 jpeg_out_callback 回调函数，处理输出 FIFO MDMA 传输完成事务。

JPEG_IRQHandler 中断服务函数，根据 JPEG_SR 的状态标志位，分别处理 JPEG 头解码完成中断和 JPEG 文件解码完成中断。当 JPEG 头解码完成时，调用 jpeg_hdp_callback 回调函数处理相关事务。当 JPEG 文件解码完成时，调用 jpeg_eoc_callback 回调函数处理相关事务，同时停止 MDMA 传输。

下面介绍的是初始化硬件 JPEG 内核函数，其定义如下：

```

/**
 * @brief      初始化硬件 JPEG 内核
 * @param      tjpeg      : JPEG 编解码控制结构体
 * @retval     0, 成功; 1, 失败;
 */
uint8_t jpeg_core_init(jpeg_codec_typedef *tjpeg)
{
    uint8_t i;

```

```

RCC->AHB3ENR |= 1 << 5; /* 使能硬件 jpeg 时钟 */

for (i = 0; i < JPEG_DMA_INBUF_NB; i++)
{
    tjpeg->inbuf[i].buf = mymalloc(SRAMDTCM, JPEG_DMA_INBUF_LEN);

    if (tjpeg->inbuf[i].buf == NULL)
    {
        jpeg_core_destroy(tjpeg);
        return 1;
    }
}

JPEG->CR = 0; /* 先清零 */
JPEG->CR |= 1 << 0; /* 使能硬件 JPEG */
JPEG->CONFR0 &= ~(1 << 0); /* 停止 JPEG 编解码进程 */
JPEG->CR |= 1 << 13; /* 清空输入 fifo */
JPEG->CR |= 1 << 14; /* 清空输出 fifo */
JPEG->CFR = 3 << 5; /* 清空标志 */
HAL_NVIC_SetPriority(JPEG IRQn, 1, 3); /* 设置中断优先级，抢占优先级 1，子优先级 3 */
HAL_NVIC_EnableIRQ(JPEG IRQn); /* 开启 JPEG 中断 */
JPEG->CONFR1 |= 1 << 8; /* 使能 header 处理 */
return 0;
}

```

`jpeg_core_init` 函数用于初始化硬件 JPEG 内核。在该函数里面，对 `tjpeg->inbuf[i].buf` 数组申请内存 (`tjpeg->outbuf[i]` 的内存申请，我们放到了 `jpeg_hdover_callback` 函数里面)。`tjpeg` 是 `jpeg_codec_typedef` 结构体类型变量，用于控制整个 JPEG 解码。`jpeg_codec_typedef` 结构体我们在 `jpegcodec.h` 里面定义的，前面讲过的。

下面介绍的是关闭硬件 JPEG 内核，并释放内存函数，其定义如下：

```

/**
 * @brief      关闭硬件 JPEG 内核，并释放内存
 * @param      tjpeg      : JPEG 编解码控制结构体
 * @retval     无
 */
void jpeg_core_destroy(jpeg_codec_typedef *tjpeg)
{
    uint8_t i;
    jpeg_dma_stop(); /* 停止 MDMA 传输 */

    for (i = 0; i < JPEG_DMA_INBUF_NB; i++)
    {
        myfree(SRAMDTCM, tjpeg->inbuf[i].buf); /* 释放内存 */
    }
    for (i = 0; i < JPEG_DMA_OUTBUF_NB; i++)
    {
        myfree(SRAMIN, tjpeg->outbuf[i].buf); /* 释放内存 */
    }
}

```

该函数用于关闭 JPEG 处理（停止 MDMA 传输），并释放内存。

下面介绍的是初始化硬件 JPEG 解码器函数，其定义如下：

```

/**
 * @brief      初始化硬件 JPEG 解码器
 * @param      tjpeg      : JPEG 编解码控制结构体
 * @retval     无
 */
void jpeg_decode_init(jpeg_codec_typedef *tjpeg)
{
    uint8_t i;
    tjpeg->inbuf_read_ptr = 0;
    tjpeg->inbuf_write_ptr = 0;
    tjpeg->indma_pause = 0;
    tjpeg->outbuf_read_ptr = 0;
}

```

```

tjpeg->outbuf_write_ptr = 0;
tjpeg->outdma_pause = 0;
tjpeg->state = JPEG_STATE_NOHEADER; /* 图片解码结束标志 */

for (i = 0; i < JPEG_DMA_INBUF_NB; i++)
{
    tjpeg->inbuf[i].sta = 0;
    tjpeg->inbuf[i].size = 0;
}

for (i = 0; i < JPEG_DMA_OUTBUF_NB; i++)
{
    tjpeg->outbuf[i].sta = 0;
    tjpeg->outbuf[i].size = 0;
}

MDMA_Channel16->CCR = 0;           /* MDMA 通道 6 禁止 */
MDMA_Channel17->CCR = 0;           /* MDMA 通道 7 禁止 */
MDMA_Channel16->CIFCR = 0X1F;      /* 中断标志清零 */
MDMA_Channel17->CIFCR = 0X1F;      /* 中断标志清零 */

JPEG->CONFR1 |= 1 << 3;          /* 硬件 JPEG 解码模式 */
JPEG->CONFR0 &= ~(1 << 0);       /* 停止 JPEG 编解码进程 */
JPEG->CR &= ~(0X3F << 1);       /* 关闭所有中断 */
JPEG->CR |= 1 << 13;             /* 清空输入 fifo */
JPEG->CR |= 1 << 14;             /* 清空输出 fifo */
JPEG->CR |= 1 << 6;              /* 使能 Jpeg Header 解码完成中断 */
JPEG->CR |= 1 << 5;              /* 使能解码完成中断 */
JPEG->CFR = 3 << 5;              /* 清空标志 */
JPEG->CONFR0 |= 1 << 0;          /* 使能 JPEG 编解码进程 */
}

```

该函数用于初始化硬件 JPEG 解码器，同时对输入 MDMA FIFO 和输出 MDMA FIFO 的相关标记进行清理处理，以便开始 JPEG 解码。

下面介绍的是启动输入 MDMA 函数和启动输出 MDMA 函数，它们的定义如下：

```

/***
 * @brief      启动 jpeg_in_mdma, 开始解码 JPEG
 * @param      无
 * @retval     无
 */
void jpeg_in_dma_start(void)
{
    MDMA_Channel17->CCR |= 1 << 0; /* 使能 MDMA 通道 7 的传输 */
}

/***
 * @brief      启动 jpeg_out_mdma, 开始输出 YUV 数据
 * @param      无
 * @retval     无
 */
void jpeg_out_dma_start(void)
{
    MDMA_Channel16->CCR |= 1 << 0; /* 使能 MDMA 通道 6 的传输 */
}

```

`jpeg_in_dma_start` 函数启动 `jpeg_in_mdma`, 开始解码 JPEG。`jpeg_out_dma_start` 函数启动 `jpeg_out_mdma`, 开始输出 YUV 数据。

下面介绍的是停止 JPEG MDMA 解码过程函数，其定义如下：

```

/***
 * @brief      停止 JPEG MDMA 解码过程
 * @param      无
 * @retval     无
 */

```

```

void jpeg_dma_stop(void)
{
    JPEG->CONFR0 &= ~(1 << 0);           /* 停止 JPEG 编解码进程 */
    JPEG->CR &= ~(0X3F << 1);           /* 关闭所有中断 */
    JPEG->CFR = 3 << 5;                  /* 清空标志 */
}

```

该函数用于停止 JPEG MDMA 解码过程。

下面介绍的是恢复 MDMA IN 过程函数和恢复 MDMA OUT 过程函数，它们的定义如下：

```

/***
 * @brief      恢复 MDMA IN 过程
 * @param      memaddr    : 存储区首地址
 * @param      memlen     : 要传输数据长度(以字节为单位)
 * @retval     无
 */
void jpeg_in_dma_resume(uint32_t memaddr, uint32_t memlen)
{
    if (memlen % 4) memlen += 4 - memlen % 4;      /* 扩展到 4 的倍数 */

    MDMA_Channel17->CIFCR = 0X1F;                /* 中断标志清零 */
    MDMA_Channel17->CBNDTR = memlen;             /* 传输长度为 memlen */
    MDMA_Channel17->CSAR = memaddr;               /* memaddr 作为源地址 */
    MDMA_Channel17->CCR |= 1 << 0;              /* 使能 MDMA 通道 7 的传输 */
}

/***
 * @brief      恢复 MDMA OUT 过程
 * @param      memaddr    : 存储区首地址
 * @param      memlen     : 要传输数据长度(以字节为单位)
 * @retval     无
 */
void jpeg_out_dma_resume(uint32_t memaddr, uint32_t memlen)
{
    if (memlen % 4) memlen += 4 - memlen % 4;      /* 扩展到 4 的倍数 */

    MDMA_Channel16->CIFCR = 0X1F;                /* 中断标志清零 */
    MDMA_Channel16->CBNDTR = memlen;             /* 传输长度为 memlen */
    MDMA_Channel16->CDAR = memaddr;               /* memaddr 作为源地址 */
    MDMA_Channel16->CCR |= 1 << 0;              /* 使能 MDMA 通道 6 的传输 */
}

```

`jpeg_in_dma_resume` 函数用于重启启动输入 MDMA。`jpeg_out_dma_resume` 函数用于重启启动输出 MDMA。

下面介绍的是获取图像信息函数，其定义如下：

```

/***
 * @brief      获取图像信息
 * @param      tjpeg      : JPEG 编解码控制结构体
 * @retval     无
 */
void jpeg_get_info(jpeg_codec_typedef *tjpeg)
{
    uint32_t yblockNb, cBblockNb, cRblockNb;

    switch (JPEG->CONFR1 & 0X03)
    {
        case 0:/* grayscale,1 color component */
            tjpeg->Conf.ColorSpace = JPEG_GRAYSCALE_COLORSPACE;
            break;

        case 2:/* YUV/RGB,3 color component */
            tjpeg->Conf.ColorSpace = JPEG_YCBCR_COLORSPACE;
            break;
    }
}

```

```

    case 3:/* CMYK,4 color component */
        tjpeg->Conf.ColorSpace = JPEG_CMYK_COLORSPACE;
        break;
    }

    tjpeg->Conf.ImageHeight = (JPEG->CONFR1 & 0xFFFF0000) >> 16; /* 获得图像高度 */
    tjpeg->Conf.ImageWidth = (JPEG->CONFR3 & 0xFFFF0000) >> 16; /* 获得图像宽度 */

    if ((tjpeg->Conf.ColorSpace == JPEG_YCBCR_COLORSPACE)
        || (tjpeg->Conf.ColorSpace == JPEG_CMYK_COLORSPACE))
    {
        yblockNb = (JPEG->CONFR4 & (0XF << 4)) >> 4;
        cBblockNb = (JPEG->CONFR5 & (0XF << 4)) >> 4;
        cRblockNb = (JPEG->CONFR6 & (0XF << 4)) >> 4;

        if ((yblockNb == 1) && (cBblockNb == 0) && (cRblockNb == 0))
        {
            tjpeg->Conf.ChromaSubsampling = JPEG_422_SUBSAMPLING; /* 16x8 block */
        }
        else if ((yblockNb == 0) && (cBblockNb == 0) && (cRblockNb == 0))
        {
            tjpeg->Conf.ChromaSubsampling = JPEG_444_SUBSAMPLING;
        }
        else if ((yblockNb == 3) && (cBblockNb == 0) && (cRblockNb == 0))
        {
            tjpeg->Conf.ChromaSubsampling = JPEG_420_SUBSAMPLING;
        }
        else
        {
            tjpeg->Conf.ChromaSubsampling = JPEG_444_SUBSAMPLING;
        }
    }
    else
    {
        tjpeg->Conf.ChromaSubsampling = JPEG_444_SUBSAMPLING; /* 默认用 4:4:4 */
    }
    /* 图像质量参数在整个图片的最末尾,刚开始的时候,是无法获取的,所以直接设置为 0 */
    tjpeg->Conf.ImageQuality = 0;
}

```

JPEG_Get_Info 函数，用于获取 JPEG 图像信息，在 JPEG 头解码完成后，被调用。该函数可以获取 JPEG 图片的宽度、高度、颜色空间和色度抽样等重要信息。

下面介绍的是得到 JPEG 图像质量函数，其定义如下：

```

/**
 * @brief      得到 JPEG 图像质量
 * @note       在解码完成后,可以调用并获得正确的结果.
 * @param      无
 * @retval     图像质量, 0~100
 */
uint8_t jpeg_get_quality(void)
{
    uint32_t quality = 0;
    uint32_t quantRow, quantVal, scale, i, j;
    uint32_t *tableAddress = (uint32_t *)JPEG->QMEM0;
    i = 0;
    while (i < JPEG_QUANT_TABLE_SIZE)
    {
        quantRow = *tableAddress;

        for (j = 0; j < 4; j++)
        {
            quantVal = (quantRow >> (8 * j)) & 0xFF;
            if (quantVal == 1) quality += 100; /* 100% */
            else
            {

```

```

        scale = (quantVal * 100)
            / ((uint32_t)JPEG_LUM_QualTable[JPEG_ZIGZAG_ORDER[i + j]]);

        if (scale <= 100)
        {
            quality += (200 - scale) / 2;
        }
        else
        {
            quality += 5000 / scale;
        }
    }

    i += 4;
    tableAddress++;
}

return (quality / ((uint32_t)64));
}

```

该函数用于获取当前 JPEG 图像的质量，返回值越大，说明图像质量越好，解码所要耗费的时间就越多。该函数我们一般用不到。

最后介绍的是将 YUV 数据转换成 RGB 数据函数，其定义如下：

```

/***
 * @brief      将 YUV 数据转换成 RGB 数据
 * @note       利用 DMA2D，将 JPEG 解码的 YUV 数据转换成 RGB 数据，全硬件完成，速度非常快
 * @param      tjpeg      : JPEG 编解码控制结构体
 * @param      pdst       : 输出数组首地址
 * @retval     0, 成功; 1, 超时, 失败;
 */
uint8_t jpeg_dma2d_yuv2rgb_conversion(jpeg_codec_t *tjpeg, uint32_t *pdst)
{
    uint32_t regval = 0;
    uint32_t cm = 0;           /* 采样方式 n */
    uint32_t timeout = 0;

    if (tjpeg->Conf.ChromaSubsampling == JPEG_420_SUBSAMPLING)
    {
        cm = DMA2D_CSS_420;          /* YUV420 转 RGB */
    }
    else if (tjpeg->Conf.ChromaSubsampling == JPEG_422_SUBSAMPLING)
    {
        cm = DMA2D_CSS_422;          /* YUV422 转 RGB */
    }
    else if (tjpeg->Conf.ChromaSubsampling == JPEG_444_SUBSAMPLING)
    {
        cm = DMA2D_NO_CSS;          /* YUV444 转 RGB */
    }

    RCC->AHB3ENR |= 1 << 4;      /* 使能 DMA2D 时钟 */
    RCC->AHB3RSTR |= 1 << 4;      /* 复位 DMA2D */
    RCC->AHB3RSTR &= ~(1 << 4);  /* 结束复位 */
    DMA2D->CR &= ~(1 << 0);      /* 先停止 DMA2D */
    DMA2D->CR = 1 << 16;          /* MODE[2:0]=001, 存储器到存储器, 带 PFC 模式 */
    DMA2D->OPFCCR = 2 << 0;        /* CM[2:0]=010, 输出为 RGB565 格式 */
    DMA2D->OOR = 0;                /* 设置行偏移为 0 */
    DMA2D->IFCR |= 1 << 1;        /* 清除传输完成标志 */
    regval = 11 << 0;              /* CM[3:0]=1011, 输入数据为 YCbCr 格式 */
    /* CSS[1:0]=cm, Chroma Sub-Sampling:0,4:4:4;1,4:2:2;2,4:2:0 */
    regval |= cm << 18;
    DMA2D->FGPCCR = regval;        /* 设置 FGPCCR 寄存器 */
    DMA2D->FGOR = 0;                /* 前景层行偏移为 0 */
    /* 设定行数寄存器 */
    DMA2D->NLR = tjpeg->yuvblk_height | (tjpeg->Conf.ImageWidth << 16);
    DMA2D->OMAR = (uint32_t)pdst;   /* 输出存储器地址 */
}

```

```

DMA2D->FGMAR = (uint32_t)tjpeg->outbuf[tjpeg->outbuf_read_ptr].buf; /*源地址*/
DMA2D->CR |= 1 << 0; /* 启动 DMA2D */
while ((DMA2D->ISR & (1 << 1)) == 0) /* 等待传输完成 */
{
    timeout++;

    if (timeout > 0X1FFFFFF) break; /* 超时退出 */
}

/* YUV2RGB 转码结束后,再复位一次 DMA2D */
RCC->AHB3RSTR |= 1 << 4; /* 复位 DMA2D */
RCC->AHB3RSTR &= ~(1 << 4); /* 结束复位 */
if (timeout > 0X1FFFFFF) return 1;
return 0;
}
}

```

该函数使用硬件 DMA2D 实现 YCbCr (YUV) 图像数据到 RGB 图像数据的格式转换，可以快速实现 YUV→RGB 数据的转换。具体的实现原理，我们在前面已经介绍过了。

JPEGCODEC 驱动代码就介绍到这里。下面我们来介绍 PICTURE 驱动代码。

PICTURE 驱动代码，本实验我们添加了 hjpgd.c 和 hjpgd.h，用于实现 JPG/JPEG 图片的硬件 JPEG 解码。

hjpgd.h 头文件中只是一些函数声明，下面直接介绍 hjpgd.c 文件的代码，首先是 JPEG 输入数据流回调函数，其定义如下：

```

jpeg_codec_TYPEDEF hjpgd; /* JPEG 硬件解码结构体 */

/**
 * @brief      JPEG 输入数据流回调函数
 * @note       用于获取 JPEG 文件原始数据，每当 JPEG DMA IN BUF 为空的时候，调用该函数
 * @param      无
 * @retval     无
 */
void jpeg_dma_in_callback(void)
{
    hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta = 0; /* 此 buf 已经处理完了 */
    hjpgd.inbuf[hjpgd.inbuf_read_ptr].size = 0; /* 此 buf 已经处理完了 */
    hjpgd.inbuf_read_ptr++; /* 指向下一个 buf */
    /* 归零 */
    if (hjpgd.inbuf_read_ptr >= JPEG_DMA_INBUF_NB) hjpgd.inbuf_read_ptr = 0;

    if (hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta == 0) /* 无效 buf */
    {
        hjpgd.indma_pause = 1; /* 标记暂停 */
    }
    else /* 有效的 buf */
    {
        /* 继续下一次 DMA 传输 */
        jpeg_in_dma_resume((uint32_t)hjpgd.inbuf[hjpgd.inbuf_read_ptr].buf,
                            hjpgd.inbuf[hjpgd.inbuf_read_ptr].size);
    }
}
}

```

该函数用于处理 JPEG 输入数据流，当 JPEG 输入 MDMA 传输完成时，调用该函数。对已处理的 buf 标记清零，然后切换到下一个 buf。当 buf 不够时，暂停 JPEG 输入 FIFO 获取数据，并标记暂停；当 buf 足够时，切换到下一个 buf，继续传输。

下面介绍的是 JPEG 输出数据流(YCBCR)回调函数，其定义如下：

```

/**
 * @brief      JPEG 输出数据流 (YCBCR) 回调函数
 * @note       用于输出 YCbCr 数据流 (YUV)
 * @param      无
 * @retval     无
 */
void jpeg_dma_out_callback(void)
{
}
}

```

```

{
    uint32_t *pdata = 0;
    hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta = 1; /* 此 buf 已满 */
    hjpgd.outbuf[hjpgd.outbuf_write_ptr].size = hjpgd.yuvblk_size
        - (MDMA_Channel6->CBNDTR & 0X1FFF); /* 此 buf 里面数据的长度 */
    /* 如果文件已经解码完成, 需要读取 DOR 最后的数据(<=32 字节) */
    if (hjpgd.state == JPEG_STATE_FINISHED)
    {
        pdata = (uint32_t *) (hjpgd.outbuf[hjpgd.outbuf_write_ptr].buf
            + hjpgd.outbuf[hjpgd.outbuf_write_ptr].size);

        while (JPEG->SR & (1 << 4))
        {
            *pdata = JPEG->DOR;
            pdata++;
            hjpgd.outbuf[hjpgd.outbuf_write_ptr].size += 4;
        }
    }
    hjpgd.outbuf_write_ptr++; /* 指向下一个 buf */
    /* 归零 */
    if (hjpgd.outbuf_write_ptr >= JPEG_DMA_OUTBUF_NB) hjpgd.outbuf_write_ptr = 0;

    if (hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta == 1) /* 无有效 buf */
    {
        hjpgd.outdma_pause = 1; /* 标记暂停 */
    }
    else /* 有效的 buf */
    {
        /* 继续下一次 DMA 传输 */
        jpeg_out_dma_resume((uint32_t)hjpgd.outbuf[hjpgd.outbuf_write_ptr].buf,
            hjpgd.yuvblk_size);
    }
}

```

该函数用于处理 JPEG 输出数据流，当 JPEG 输出 MDMA 传输完成时，调用该函数。对已满的 buf 标记满，并标记容量，然后切换到下一个 buf。当 buf 不够时，暂停获取 JPEG 输出 FIFO 的数据，并标记暂停；当 buf 足够时，切换到下一个 buf，继续传输。当解码状态结束时，需要手动读取 JPEG_DOR 寄存器的数据。

下面介绍的是 JPEG 整个文件解码完成回调函数，其定义如下：

```

/***
 * @brief      JPEG 整个文件解码完成回调函数
 * @param      无
 * @retval     无
 */
void jpeg_endofcovert_callback(void)
{
    hjpgd.state = JPEG_STATE_FINISHED; /* 标记 JPEG 解码完成 */
}

```

该函数在 JPG/JPEG 文件解码结束时调用。该函数处理非常简单，直接将当前解码状态标记为：JPEG 解码完成（JPEG_STATE_FINISHED）即可。

下面介绍的是 JPEG header 解析成功回调函数，其定义如下：

```

/***
 * @brief      JPEG header 解析成功回调函数
 * @param      无
 * @retval     无
 */
void jpeg_hdrover_callback(void)
{
    uint8_t i = 0;
    hjpgd.state = JPEG_STATE_HEADEROK; /* HEADER 获取成功 */
    jpeg_get_info(&hjpgd); /* 获取 JPEG 相关信息，包括大小，色彩空间，抽样等 */
    picinfo.ImgWidth = hjpgd.Conf.ImageWidth;
}

```

```

picinfo.ImgHeight = hjpgd.Conf.ImageHeight;

/* 需要获取 JPEG 基本信息以后, 才能根据 jpeg 输出大小和采样方式, 来计算输出缓冲大小,
并启动输出 MDMA */
switch (hjpgd.Conf.ChromaSubsampling)
{
    case JPEG_420_SUBSAMPLING:
        /* YUV420, 每个 YUV 像素占 1.5 个字节. 每次输出 16 行. 16*1.5=24 */
        hjpgd.yuvblk_size = 24 * hjpgd.Conf.ImageWidth;
        hjpgd.yuvblk_height = 16; /* 每次输出 16 行 */
        break;

    case JPEG_422_SUBSAMPLING:
        /* YUV422, 每个 YUV 像素占 2 个字节. 每次输出 8 行. 8*2=16 */
        hjpgd.yuvblk_size = 16 * hjpgd.Conf.ImageWidth;
        hjpgd.yuvblk_height = 8; /* 每次输出 8 行 */
        break;

    case JPEG_444_SUBSAMPLING:
        /* YUV444, 每个 YUV 像素占 3 个字节. 每次输出 8 行. 8*3=24 */
        hjpgd.yuvblk_size = 24 * hjpgd.Conf.ImageWidth;
        hjpgd.yuvblk_height = 8; /* 每次输出 8 行 */
        break;
}

hjpgd.yuvblk_curheight = 0; /* 当前行计数器清零 */

for (i = 0; i < JPEG_DMA_OUTBUF_NB; i++)
{
    /* 可能会多需要 32 字节内存 */
    hjpgd.outbuf[i].buf = mymalloc(SRAMIN, hjpgd.yuvblk_size + 32);

    if (hjpgd.outbuf[i].buf == NULL)
    {
        hjpgd.state = JPEG_STATE_ERROR; /* HEADER 获取失败 */
    }
}

if (hjpgd.outbuf[JPEG_DMA_OUTBUF_NB - 1].buf != NULL) /* 所有 buf 都申请 OK */
{
    /* 配置输出 DMA */
    jpeg_out_dma_init((uint32_t)hjpgd.outbuf[0].buf, hjpgd.yuvblk_size);
    jpeg_out_dma_start(); /* 启动 DMA OUT 传输, 开始接收 JPEG 解码数据流 */
}
piclib_ai_draw_init();
}

```

该函数在 JPEG 头解码成功后调用。该函数先标记状态为 JPEG 头解码成功 (JPEG_STATE_HEADEROK)，然后调用 JPEG_Get_Info 函数获取 JPEG 相关信息。在得到 JPEG 文件的抽样方式、图库宽度等信息后，计算：yuvblk_size 和 yuvblk_height 这两个关键参数，然后对申请 outbuf 内存并初始化输出 FIFO MDMA，最后启动输出 FIFO MDMA，开始接收硬件 JPEG 解码后的数据。

这里需要注意：因为 JPEG 输出 FIFO 的 MDMA 是在 JPEG 头解码成功回调函数里面才申请内存并初始化的，因此输入 FIFO 的 MDMA 必须先输入一部分数据给 JPEG 解码内核，才可以执行输出 FIFO MDMA 初始化。但是如果整张图片的大小，都不够一次输入 FIFO MDMA 的传输的话，那么可能会出现解码无法完成的情况，出现这种情况很好解决，就是把输入 FIFO MDMA 的大小改小一点，即 JPEG_DMA_INBUF_LEN 的大小改小一点。

下面介绍的是 JPEG 硬件解码图片函数，其定义如下：

```

/**
 * @brief      JPEG 硬件解码图片
 * @note       注意, 请保证:
 *             1, 待解码图片的分辨率, 必须小于等于屏幕的分辨率!

```

```
/*
 *      2, 请保证图片的宽度是 16 的倍数,否则解码出错!
 */

* @param      filename : 包含路径的文件名(.jpeg/jpg)
* @retval     操作结果
* @arg        0 , 成功
* @arg        其他, 错误码
*/
uint8_t hjpgd_decode(uint8_t *filename)
{
    FIL *ftemp;
    uint16_t *rgb565buf = 0;
    volatile uint32_t timecnt = 0;
    uint32_t br = 0;
    uint8_t fileover = 0;
    uint8_t i = 0;
    uint8_t res;
    res = jpeg_core_init(&hjpgd); /* 初始化 JPEG 内核 */

    if (res) return 1;

    ftemp = (FIL *)mymalloc(SRAMITCM, sizeof(FIL)); /* 申请内存 */

    if (f_open(ftemp, (char *)filename, FA_READ) != FR_OK) /* 打开图片失败 */
    {
        jpeg_core_destroy(&hjpgd);
        myfree(SRAMITCM, ftemp); /* 释放内存 */
        return 2;
    }

    jpeg_decode_init(&hjpgd); /* 初始化硬件 JPEG 解码器 */

    for (i = 0; i < JPEG_DMA_INBUF_NB; i++)
    {
        /* 填满所有输入数据缓冲区 */
        res = f_read(ftemp, hjpgd.inbuf[i].buf, JPEG_DMA_INBUF_LEN, &br);

        if (res == FR_OK && br)
        {
            hjpgd.inbuf[i].size = br; /* 读取 */
            hjpgd.inbuf[i].sta = 1; /* 标记 buf 满 */
        }
        if (br == 0) break;
    }

    /* 配置输入 DMA */
    jpeg_in_dma_init((uint32_t)hjpgd.inbuf[0].buf, hjpgd.inbuf[0].size);
    jpeg_in_callback = jpeg_dma_in_callback; /* JPEG DMA 读取数据回调函数 */
    jpeg_out_callback = jpeg_dma_out_callback; /* JPEG DMA 输出数据回调函数 */
    jpeg_eoc_callback = jpeg_endofcovert_callback; /* JPEG 解码结束回调函数 */
    jpeg_hdp_callback = jpeg_hdover_callback; /* JPEG Header 解码完成回调函数 */
    jpeg_in_dma_start(); /* 启动 DMA IN 传输,开始解码 JPEG 图片 */

    while (1)
    {
        /* rgb565buf 空,且 JPEG HEAD 解码完成 */
        if (rgb565buf == 0 && hjpgd.state == JPEG_STATE_HEADEROK)
        {
            rgb565buf = mymalloc(SRAMIN, hjpgd.Conf.ImageWidth
                * hjpgd.yuvblk_height * 2 + 32); /* 申请单次输出缓冲内存 */
        }
        /* 有 buf 为空 */
        if (hjpgd.inbuf[hjpgd.inbuf_write_ptr].sta == 0 && fileover == 0)
        {
            res = f_read(ftemp, hjpgd.inbuf[hjpgd.inbuf_write_ptr].buf,
```

```
JPEG_DMA_INBUF_LEN, &br); /* 填满一个缓冲区 */

if (res == FR_OK && br)
{
    hjpgd.inbuf[hjpgd.inbuf_write_ptr].size = br; /* 读取 */
    hjpgd.inbuf[hjpgd.inbuf_write_ptr].sta = 1; /* buf 满 */
}
else if (br == 0)
{
    timecnt = 0; /* 清零计时器 */
    fileover = 1; /* 文件结束了 */
}
if (hjpgd.indma_pause == 1 && hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta
    == 1) /* 之前是暂停的了,继续传输 */
{
    jpeg_in_dma_resume((uint32_t)hjpgd.inbuf[hjpgd.inbuf_read_ptr].buf,
    hjpgd.inbuf[hjpgd.inbuf_read_ptr].size); /* 继续下一次 DMA 传输 */
    hjpgd.indma_pause = 0;
}
hjpgd.inbuf_write_ptr++;
if (hjpgd.inbuf_write_ptr >= JPEG_DMA_INBUF_NB)
    hjpgd.inbuf_write_ptr = 0;
}
if (hjpgd.outbuf[hjpgd.outbuf_read_ptr].sta == 1) /* buf 里面有数据要处理 */
{
    SCB_CleanInvalidateDCache(); /* 清空 D catch */
    /* 利用 DMA2D,将 YUV 图像转成 RGB565 图像 */
    jpeg_dma2d_yuv2rgb_conversion(&hjpgd, (uint32_t *)rgb565buf);

    /* 输出到 LCD 屏幕上面 */
    pic_phy.fillcolor(picinfo.S_XOFF, picinfo.S_YOFF
                      + hjpgd.yuvblk_curheight, hjpgd.Conf.ImageWidth,
                      hjpgd.yuvblk_height, rgb565buf);
    hjpgd.yuvblk_curheight += hjpgd.yuvblk_height; /* 列偏移 */

    //SCB_CleanInvalidateDCache(); /* 清空 D catch */
    hjpgd.outbuf[hjpgd.outbuf_read_ptr].sta = 0; /* 标记 buf 为空 */
    hjpgd.outbuf[hjpgd.outbuf_read_ptr].size = 0; /* 数据量清空 */
    hjpgd.outbuf_read_ptr++;

    if (hjpgd.outbuf_read_ptr >= JPEG_DMA_OUTBUF_NB)
        hjpgd.outbuf_read_ptr = 0; /* 限制范围 */
    /* 当前高度等于或者超过图片分辨率的高度,则说明解码完成了,直接退出 */
    if (hjpgd.yuvblk_curheight >= hjpgd.Conf.ImageHeight) break;
}
/* out 暂停,且当前 writebuf 已经为空了,则恢复 out 输出 */
else if (hjpgd.outdma_pause == 1
         && hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta == 0)
{
    jpeg_out_dma_resume((uint32_t)hjpgd.outbuf[hjpgd.outbuf_write_ptr]
                         .buf, hjpgd.yuvblk_size); /* 继续下一次 DMA 传输 */
    hjpgd.outdma_pause = 0;
}
timecnt++;
if (hjpgd.state == JPEG_STATE_ERROR) /* 解码出错,直接退出 */
{
    res = 2;
    break;
}

if (fileover) /* 文件结束后,及时退出,防止死循环 */
{
    /* 当前处于暂停状态, 且没有解析到 JPEG 头 */
}
```

```

    if (hjpgd.state == JPEG_STATE_NOHEADER && hjpgd.indma_pause == 1)
    {
        break; /* 解码 JPEG 头失败了 */
    }
    if (timecnt > 0X3FFFF) break; /* 超时退出 */
}
f_close(ftemp); /* 关闭文件 */
myfree(SRAMITCM, ftemp); /* 释放申请的内存 */
myfree(SRAMIN, rgb565buf); /* 释放内存 */
jpeg_core_destroy(&hjpgd); /* 结束 JPEG 解码, 释放内存 */
return res;
}

```

该函数用于解码一张 JPG/JPEG 图片。该函数采用的思路就是按照我们在 53.3.1 节介绍的步骤来解码 JPG/JPEG 图片。请大家参考前面的介绍和源码进行理解。

另外，我们需要将 hjpgd_decode 函数加入到图片解码库里面，修改 piclib_ai_load_picfile 函数代码，具体如下：

```

/**
 * @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 无
 */
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 = exfun_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:
if (fast) /* 可能需要硬件解码 */
{
    res = jpg_get_size(filename, &picinfo.ImgWidth,
                         &picinfo.ImgHeight);

    if (res == 0)
    {
        /* 满足分辨率小于等于屏幕分辨率、满足图片宽度为 16 的整数倍，则可以硬件解码 */
        if (picinfo.ImgWidth <= lcddev.width && picinfo.ImgHeight
            <= lcddev.height && picinfo.ImgWidth <= picinfo.S_Width
            && picinfo.ImgHeight <= picinfo.S_Height &&
            (picinfo.ImgWidth % 16) == 0)
        {
            /* 采用硬解码 JPG/JPEG */
            res = hjpgd_decode((uint8_t *)filename);
        }
        else
        {
            res = jpg_decode(filename, fast); /* 采用软件解码 JPG/JPEG */
        }
    }
}
else
{
    res = jpg_decode(filename, fast); /* 统一采用软件解码 JPG/JPEG */
}
break;
case T_GIF:
res = gif_decode(filename, x, y, width, height); /* 解码 gif */
break;
default:
res = PIC_FORMAT_ERR; /* 非图片格式!!! */
break;
}
return res;
}

```

启用快速解码时，当 JPG/JPEG 图片尺寸满足小于等于屏幕分辨率，且图片宽度是 16 的倍数，则我们会通过调用 `hjpgd_decode` 函数实现硬件 JPEG 解码，从而大大提高速度。

PICTURE 驱动代码就介绍到这里，下面我们介绍一下本章的应用代码，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 */

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */

```

```

usmart_dev.init(240);           /* 初始化 USMART */
mpu_memory_protection();        /* 保护相关存储区域 */
led_init();                     /* 初始化 LED */
lcd_init();                     /* 初始化 LCD */
key_init();                     /* 初始化按键 */
my_mem_init(SRAMIN);           /* 初始化内部内存池 (AXI) */
my_mem_init(SRAM12);           /* 初始化 SRAM12 内存池 (SRAM1+SRAM2) */
my_mem_init(SRAM4);            /* 初始化 SRAM4 内存池 (SRAM4) */
my_mem_init(SRAMDTCM);          /* 初始化 DTCM 内存池 (DTCM) */
my_mem_init(SRAMITCM);          /* 初始化 ITCM 内存池 (ITCM) */
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, "硬件 JPEG 解码 实验", 16, 0, RED);
text_show_string(30, 90, 200, 16, "KEY0:NEXT KEY1:PREV", 16, 0, RED);
text_show_string(30, 110, 200, 16, "KEY_UP:PAUSE", 16, 0, RED);
text_show_string(30, 130, 200, 16, "ATOM@ALIENTEK", 16, 0, RED);

while (f_opendir(&picdir, "0:/PICTURE")) /* 打开图片文件夹 */
{
    text_show_string(30, 170, 240, 16, "PICTURE 文件夹错误!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 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, 170, 240, 16, "没有图片文件!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 240, 186, WHITE); /* 清除显示 */
    delay_ms(200);
}
picfileinfo = (FILINFO *)mymalloc(SRAMIN, sizeof(FILINFO)); /* 申请内存 */
pname = mymalloc(SRAMIN, 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, 170, 240, 16, "内存分配失败!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 170, 240, 186, WHITE); /* 清除显示 */
    delay_ms(200);
}
/* 记录索引 */
res = f_opendir(&picdir, "0:/PICTURE"); /* 打开目录 */
if (res == FR_OK)
{
    curindex = 0; /* 当前索引为 0 */
    while (1) /* 全部查询一遍 */
    {
        temp = picdir.dptr; /* 记录当前 dptr 偏移 */

```

```
res = f_readdir(&picdir, picfileinfo); /* 读取目录下的一个文件 */
/* 错误了/到末尾了,退出 */
if (res != FR_OK || picfileinfo->fname[0] == 0)break;
res = exfun_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, 170, 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 */
    /* 错误了/到末尾了,退出 */
    if (res != FR_OK || picfileinfo->fname[0] == 0)break;
    strcpy((char *)pname, "0:/PICTURE/");
    /* 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 */
}
```

main 函数里面我们通过读/写偏移量（图片文件在 PICTURE 文件夹下的读/写偏移位置，可以看做是一个索引），来查找上一个/下一个图片文件（使用 dir_sdi 函数）。通过 piclib_ai_load_picfile 函数，实现对 JPG/JPEG 图片的解码。这里将 fast 参数设置为 1，当图片文件的分辨率小于等于液晶分辨率的时候，将使用硬件 JPEG 进行解码。

至此本例程代码编写完成。最后，本实验可以通过 USMART 来调用相关函数，以对比性能。将 mf_scan_files、piclib_ai_load_picfile 和 hjpgd_decode 等函数添加到 USMART 管理，即可以通过串口调用这几个函数，测试对比软件 JPEG 解码和硬件 JPEG 解码的速度差别。

53.3 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 显示了一些实验信息之后就开始显示图片（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）。

按下 KEY0 或 WK_UP 按键，可分别进行切换下一张图片和切换上一张图片的操作。对比上一章实验，我们可以发现，对于小尺寸的 JPG/JPEG 图片（小于液晶分辨率），本例程解码速度明显提升。

第五十四章 照相机实验

本章将介绍使用 STM32H750 模拟照相机将摄像头采集到的图像数据以 BMP 或 JPEG 的文件格式保存至 SD 中，是前面章节中 SD 卡驱动、摄像头驱动、FATFS、字库管理库、图片编解码库等的综合应用。通过本章的学习，读者将学习到编码 BMP 文件以及保存摄像头采集到图像数据的保存。

本章分为以下几个小节：

- 54.1 硬件设计
- 54.2 程序设计
- 54.3 下载验证

54.1 硬件设计

54.1.1 例程功能

1. 按下 WKUP 按键对摄像头输出的数据进行 BMP 编码并保存值 SD 卡中
2. LED1 闪烁，提示 DCMI 接收到一帧数据
3. LED0 闪烁，提示程序正在运行

54.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
4. ALIENTEK OV5640 摄像头模块,连接关系为:
 - OV_D0~D7 - PC6/PC7/PC8/PC9/PC11/PD3/PB8/PB9
 - OV_SCL - PB10
 - OV_SDA - PB11
 - OV_VSYNC - PB7
 - OV_HREF - PA4
 - OV_RESET - PA7
 - OV_PCLK - PA6
 - OV_XCLK - PA8
 - OV_PWDN - PC4
5. SD
 - SDIO_D0 - PC8
 - SDIO_D1 - PC9
 - SDIO_D2 - PC10
 - SDIO_D3 - PC11
 - SDIO_SCK - PC12
 - SDIO_CMD - PD2

54.1.3 原理图

本章实验涉及的 SD 卡、摄像头、LCD 等的连接原理图，请读者自行查看前面对应章节中的连接原理图。

54.2 程序设计

54.2.1 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    uint8_t res;
    uint8_t *pname;
    uint8_t key;
    uint8_t t = 0;
    uint8_t sd_ok = 1;
    uint16_t outputheight = 0;

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    ov5640_init(); /* 初始化 OV5640 */
    sw_sdcard_mode(); /* 首先切换为 OV5640 模式 */
    piclib_init(); /* 初始化画图 */
    my_mem_init(SRAMIN); /* 初始化内部内存池(AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池(SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池(DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池(ITCM) */
    exfun_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() != 0) /* 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);
    }

    while (sd_init() != SD_OK) /* 初始化 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);
    }

    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);
    text_show_string(30, 110, 200, 16, "WK_UP:拍照(jpg 格式)", 16, 0, RED);

    /* 创建 PHOTO 文件夹 */
    res = (uint8_t)f_mkdir("0:/PHOTO");
    /* 发生了错误 */
}
```

```
if ((res != (uint8_t)FR_EXIST) && (res != FR_OK))
{
    text_show_string(30, 150, 240, 16, "SD 卡错误!", 16, 0, RED);
    text_show_string(30, 150, 240, 16, "拍照功能将不可用!", 16, 0, RED);
    sd_ok = 0;
}

/* 为 jpeg dma 接收申请内存 */
p_dcmi_line_buf[0] = mymalloc(SRAM12, jpeg_line_size * 4);
/* 为 jpeg dma 接收申请内存 */
p_dcmi_line_buf[1] = mymalloc(SRAM12, jpeg_line_size * 4);
/* 为 jpeg 文件申请内存 */
p_jpeg_data_buf = mymalloc(SRAMIN, jpeg_buf_size);
/* 为带路径的文件名分配 30 个字节的内存 */
pname = mymalloc(SRAMIN, 30);

/* 内存分配出错 */
while ((pname == NULL) ||
       (p_dcmi_line_buf[0] == NULL) ||
       (p_dcmi_line_buf[1] == NULL) ||
       (p_jpeg_data_buf == NULL))
{
    text_show_string(30, 150, 240, 16, "内存分配失败!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 150, 240, 146, WHITE);
    delay_ms(200);
}

while (ov5640_init() != 0) /* 初始化 OV5640 */
{
    text_show_string(30, 170, 240, 16, "OV5640 错误!", 16, 0, RED);
    delay_ms(200);
    lcd_fill(30, 150, 239, 206, WHITE);
    delay_ms(200);
}
delay_ms(100);
text_show_string(30, 170, 230, 16, "OV5640 正常", 16, 0, RED);

ov5640_rgb565_mode(); /* RGB565 模式 */
dcmi_init(); /* DCMI 配置 */
/* DCMI DMA 配置,MCU 屏,竖屏 */
dcmi_dma_init((uint32_t)&LCD->LCD_RAM,
               0,
               1,
               DMA_MDATAALIGN_HALFWORD,
               DMA_MINC_DISABLE);

if (lcddev.height >= 800)
{
    g_yoffset = (lcddev.height - 800) / 2;
    outputheight = 800;
    ov5640_write_reg(0x3035, 0x51); /* 降低输出帧率, 否则可能抖动 */
}
else
{
    g_yoffset = 0;
    outputheight = lcddev.height;
}

g_curline = g_yoffset; /* 行数复位 */
ov5640_outsize_set(16, /* 满屏缩放显示 */
                   4,
                   lcddev.width,
                   outputheight);
```

```
dcmi_start(); /* 启动传输 */
lcd_clear(BLACK);

while (1)
{
    t++;
    key = key_scan(0); /* 不支持连接 */

    if (key != 0) /* 有按键按下 */
    {
        dcmi_stop(); /* 先禁止 DCMI 传输 */

        if (sd_ok == 1) /* SD 卡正常 */
        {
            sw_sdcard_mode(); /* 切换为 SD 卡模式 */

            switch (key)
            {
                case KEY0_PRES:
                {
                    camera_new_pathname(pname, 0); /* 得到 BMP 格式文件名 */
                    /* 编码并保存 BMP 文件 */
                    res = bmp_encode(pname, 0, 0, lcddev.width, lcddev.height, 0);
                    break;
                }
                case WKUP_PRES:
                {
                    camera_new_pathname(pname, 1); /* 得到 JPG 格式文件 */
                    /* 保存 OV5640 输出的 JPEG 数据到 JPG 文件 */
                    res = ov5640_jpg_photo(pname);
                    ov5640_outsize_set(0, 0, lcddev.width, lcddev.height);
                    break;
                }
                default:
                {
                    break;
                }
            }
            sw_ov5640_mode(); /* 切换为 OV5640 模式 */

            if (res != 0)
            {
                text_show_string(30, 130, 240, 16, "写入文件错误!", 16, 0, RED);
            }
            else
            {
                text_show_string(30, 130, 240, 16, "拍照成功!", 16, 0, RED);
                text_show_string(30, 150, 240, 16, "保存为:", 16, 0, RED);
                text_show_string(30+56, 150, 240, 16, (char*)pname, 16, 0, RED);
            }
            delay_ms(1000);
            /* 这里先使能 dcmi, 然后立即关闭 DCMI, 后面再开启 DCMI, 可以防止 RGB 屏的侧移问题 */
            dcmi_start();
            dcmi_stop();
        }
        else /* SD 卡不可用 */
        {
            text_show_string(30, 130, 240, 16, "SD 卡错误!", 16, 0, RED);
            text_show_string(30, 150, 240, 16, "拍照功能不可用!", 16, 0, RED);
        }
        delay_ms(2000);
        dcmi_start(); /* 开始显示 */
    }
}
```

```

    if (t == 20)
    {
        LED0_TOGGLE();
        t = 0;
    }

    delay_ms(10);
}
}

```

可以看到，本章实验实际上是前面章节中 SD 卡驱动、摄像头驱动、FATFS、字库管理库、图片编解码库等的综合应用，并没有涉及新的内容。要注意的是，对于正点原子 M100Z-M7 最小系统板 STM32H750 版，其 SD 卡和摄像头是无法同时工作的，因为它们使用到了部分共同的 GPIO 引脚，因此在本章实验中，需要对 GPIO 引脚的复用模式进行切换，以正常使用 SD 卡或摄像头，用于切换 GPIO 引脚复用模式的函数，如下所示：

```

/***
 * @brief      切换为 OV5640 模式
 * @note       切换 PC8/PC9/PC11 为 DCMI 复用功能 (AF13)
 * @param      无
 * @retval     无
 */
void sw_ov5640_mode(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    ov5640_write_reg(0X3017, 0xFF); /* 开启 OV5650 输出 (可以正常显示) */
    ov5640_write_reg(0X3018, 0xFF);

    /* GPIOC8/9/11 切换为 DCMI 接口 */
    gpio_init_struct.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_11;
    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_VERY_HIGH; /* 高速 */
    gpio_init_struct.Alternate = GPIO_AF13_DCMI;      /* 复用为 DCMI */
    HAL_GPIO_Init(GPIOC, &gpio_init_struct);         /* 初始化 PC8, 9, 11 引脚 */
}

```

因为 SD 卡和 OV5640 有几个 IO 共用，所以这几个 IO 需要分时复用。该函数用于切换 GPIO8/9/11 的复用功能为 DCMI 接口，并开启 OV5640，这样摄像头模块，可以开始正常工作。

接下来介绍的是切换为 SD 卡模式函数，其定义如下：

```

/***
 * @brief      切换为 SD 卡模式
 * @note       切换 PC8/PC9/PC11 为 SDMMC 复用功能 (AF12)
 * @param      无
 * @retval     无
 */
void sw_sdcard_mode(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    ov5640_write_reg(0X3017, 0x00); /* 关闭 OV5640 全部输出 (不影响 SD 卡通信) */
    ov5640_write_reg(0X3018, 0x00);

    /* GPIOC8/9/11 切换为 SDIO 接口 */
    gpio_init_struct.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_11;
    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_VERY_HIGH; /* 高速 */
    gpio_init_struct.Alternate = GPIO_AF12_SDIO1;      /* 复用为 SDIO */
    HAL_GPIO_Init(GPIOC, &gpio_init_struct);         /* 初始化 PC8, 9, 11 引脚 */
}

```

该函数用于切换 GPIO8/9/11 的复用功能为 SDMMC 接口，并关闭 OV5640，这样，SD 卡可

以开始正常工作。

54.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上实时显示这摄像头采集到的图像数据，按下 KEY0 或 WK_UP 按键，可分别进行拍摄并保存 BMP 或 JPG 格式的图片文件至 SD 卡。拍照完成后，可通过 PC 或图片显示实验例程（需修改获取图片的文件夹路径为“0:PHOTO”）查看所拍摄的 BMP 或 JPG 格式的照片。

第五十五章 视频播放器实验

本章将介绍使用 STM32H750 实现一个简单的视频播放器应用。通过本章的学习，读者将学习到使用 libjpeg 库软解 JPEG 实现播放 AVI 视频的使用。

本章分为如下几个小节：

- 55.1 硬件设计
- 55.2 程序设计
- 55.3 下载验证

55.1 硬件设计

55.1.1 例程功能

1. LCD 上自动播放 AVI 格式视频
2. 按下 WKUP 按键可切换至上一个视频
3. 按下 KEY0 按键可切换至下一个视频
4. LED0 闪烁，提示程序正在运行

55.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15
4. NOR Flash
 - QSPI FLASH 芯片，连接在 QSPI 上
5. SD
 - SDIO_D0 - PC8
 - SDIO_D1 - PC9
 - SDIO_D2 - PC10
 - SDIO_D3 - PC11
 - SDIO_SCK - PC12
 - SDIO_CMD - PD2

55.1.3 原理图

本章实验主要涉及 libjpeg 软件库的使用和 AVI 文件的解析，因此没有对应的连接原理图。

55.2 程序设计

55.2.1 libjpeg 库的使用

本章实验要实现一个简单的视频播放器，以播放 AVI 格式的视频，AVI 格式的视频数据可以采用 MJPG 进行编码，因此在播放时，需要对 MJPG 数据流进行解码，这样便可以从 MJPG 数据流中获得视频的逐帧画面，只需要在 LCD 上连续显示这些画面，就能够实现视频播放。

在第五十二章“图片显示实验”中使用 TjpgDec 解码 JPEG 图片，虽然 TjpgDec 占用资源少，但解码速度慢，若用于解码 MJPG 数据流，会导致视频播放不够流畅，因此本章实验使用解码速度更快的 libjpeg 库解码 MJPG 数据流。

关于 libjpeg 库的移植和使用，请读者自行查看 libjpeg 源码中的介绍文件，可以重点看

readme.txt、filelist.txt、install.txt 和 libjpeg.txt 等文件，可以参考本章实验配套实验例程进行移植和使用。

为了更方便地使用 libjpeg 实现本章实验的功能，正点原子提供了 mjpeg.c 和 mjpeg.h 这两个文件，这两个文件提供了初始化、解码等三个函数，大大简化了 libjpeg 的使用流程，函数原型，如下所示：

```
uint8_t mjpegdec_init(uint16_t offx,uint16_t offy);
void mjpegdec_free(void);
uint8_t mjpegdec_decode(uint8_t* buf,uint32_t bsize);
```

这三个函数的使用也非常简单，光看函数名也能猜出各个函数的作用了，因此具体的使用方法，请读者自行查看本章实验的配套实验例程。

55.2.2 AVI 文件解析

有关 AVI 文件格式的介绍和解析方式，请感兴趣的读者自行查阅相关的资料。对于 AVI 文件的解析，正点原子提供了 avi.c 和 avi.h 两个文件，请读者结合本章实验配套的实验例程进行查看。

55.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    usmart_dev.init(240); /* 初始化 USMART */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    my_mem_init(SRAMIN); /* 初始化内部内存池 (AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池 (SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池 (SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池 (DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池 (ITCM) */
    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 */
    lcd_display_dir(1); /* 设置成横屏 */
    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(60, 50, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
    text_show_string(60, 70, 200, 16, "视频播放器 实验", 16, 0, RED);
    text_show_string(60, 90, 200, 16, "KEY0:NEXT KEY1:PREV", 16, 0, RED);
    text_show_string(60, 110, 200, 16, "KEY_UP:FF", 16, 0, RED);
    delay_ms(1500);
    btim_timx_int_init(10000-1, 24000-1); /* 10Khz 计数，1 秒钟中断一次 */
    while(1)
    {
        video_play();
    }
}
```

可以看到，本章实验在完成一些必要的软硬件初始化后，便调用函数 video_play()进行视频播放，该函数如下所示：

```
/***
 * @brief      播放视频
 * @param      无
 * @retval     无
 */
void video_play(void)
{
    uint8_t res;
    DIR vdir;
    FILINFO *vfileinfo;
    uint8_t *pname;
    uint16_t totavinum;
    uint16_t curindex;
    uint8_t key;
    uint32_t temp;
    uint32_t *voffsettbl;

    while (f_opendir(&vdir, "0:/VIDEO") != FR_OK)
    {
        text_show_string(60, 190, 240, 16, "VIDEO 文件夹错误!", 16, 0, RED);
        delay_ms(200);
        lcd_fill(60, 190, 240, 206, WHITE);
        delay_ms(200);
    }

    totavinum = video_get_tnum("0:/VIDEO");

    while (totavinum == 0)
    {
        text_show_string(60, 190, 240, 16, "没有视频文件!", 16, 0, RED);
        delay_ms(200);
        lcd_fill(60, 190, 240, 146, WHITE);
        delay_ms(200);
    }

    vfileinfo = (FILINFO *)mymalloc(SRAM12, sizeof(FILINFO));
    pname = (uint8_t *)mymalloc(SRAM12, 2 * FF_MAX_LFN + 1);
    voffsettbl = (uint32_t *)mymalloc(SRAM12, 4 * totavinum);

    while ((vfileinfo == NULL) || (pname == NULL) || (voffsettbl == NULL))
    {
        text_show_string(60, 190, 240, 16, "内存分配失败!", 16, 0, RED);
        delay_ms(200);
        lcd_fill(60, 190, 240, 146, WHITE);
        delay_ms(200);
    }

    res = (uint8_t)f_opendir(&vdir, "0:/VIDEO");

    if (res == FR_OK)
    {
        curindex = 0;

        while (1)
        {
            temp = vdir.dptr;
            res = (uint8_t)f_readdir(&vdir, vfileinfo);

            if ((res != FR_OK) || (vfileinfo->fname[0] == 0))
            {
                break;
            }
        }
    }
}
```

```

res = exfun_s_file_type(vfileinfo->fname);

if ((res & 0xF0) == 0x60)
{
    voffsettbl[curindex] = temp;
    curindex++;
}
}

curindex = 0;
res = (uint8_t)f_opendir(&vdir, "0:/VIDEO");

while (res == FR_OK)
{
    dir_sdi(&vdir, voffsettbl[curindex]);
    res = (uint8_t)f_readdir(&vdir, vfileinfo);

    if ((res != FR_OK) || (vfileinfo->fname[0] == 0))
    {
        break;
    }

    strcpy((char *)pname, "0:/VIDEO/");
    strcat((char *)pname, (const char *)vfileinfo->fname);
    lcd_clear(WHITE);
    video_bmsg_show((uint8_t *)vfileinfo->fname, curindex + 1, totavinum);
    key = video_play_mjpeg(pname);

    if (key == KEY0_PRES)
    {
        if (curindex != 0)
        {
            curindex--;
        }
        else
        {
            curindex = totavinum - 1;
        }
    }
    else if (key == WKUP_PRES)
    {
        curindex++;

        if (curindex >= totavinum)
        {
            curindex = 0;
        }
    }
    else
    {
        break;
    }
}

myfree(SRAM12, vfileinfo);
myfree(SRAM12, pname);
myfree(SRAM12, voffsettbl);
}

```

从上面的代码中可以看出，主要就是调用函数 video_play_mjpeg() 来播放 AVI 视频文件，该函数如下所示：

```
/***
 * @brief      播放一个 MJPEG 文件
 * @param      pname   : 文件名
 */
```

```
* @retval 执行结果
* @arg 无
* @arg 无
* @arg 其他 , 错误
*/
uint8_t video_play_mjpeg(uint8_t *pname)
{
    uint8_t *framebuf; /* 视频解码 buf */
    uint8_t *pbuff; /* buf 指针 */
    FIL *favi;
    uint8_t res = 0;
    uint32_t offset = 0;
    uint32_t nr;
    uint8_t key;
    psaibus = mymalloc(SRAM4, AVI_AUDIO_BUF_SIZE); /* 申请音频内存 */
    framebuf = mymalloc(SRAMIN, AVI_VIDEO_BUF_SIZE); /* 申请视频buf */
    favi = (FIL *)mymalloc(SRAM12, sizeof(FIL)); /* 申请 favi 内存 */
    memset(psaibus, 0, AVI_AUDIO_BUF_SIZE);
    if (!psaibus || !framebuf || !favi)
    {
        printf("memory error!\r\n");
        res = 0xFF;
    }
    while (res == 0)
    {
        res = f_open(favi, (char *)pname, FA_READ);
        if (res == 0)
        {
            pbuff = framebuf;
            res = f_read(favi, pbuff, AVI_VIDEO_BUF_SIZE, &nr); /* 开始读取 */
            if (res)
            {
                printf("fread error:%d\r\n", res);
                break;
            }
            /* 开始 avi 解析 */
            res = avi_init(pbuff, AVI_VIDEO_BUF_SIZE); /* avi 解析 */

            if (res || avix.Width > lcddev.width)
            {
                printf("avi err:%d\r\n", res);
                res = KEY0_PRES;
                break;
            }
            video_info_show(&avix);
            /* 10Khz 计数频率,加1是 100us */
            btim_tim7_int_init(avix.SecPerFrame / 100 - 1, 24000 - 1);
            /* 寻找 movi ID */
            offset = avi_srarch_id(pbuff, AVI_VIDEO_BUF_SIZE, "movi");
            avi_get_stremainfo(pbuff + offset + 4); /* 获取流信息 */
            f_lseek(favi, offset + 12); /* 跳过标志 ID, 读地址偏移到流数据开始处 */

            if (lcddev.height <= avix.Height)
            {
                res = mjpeg_init((lcddev.width - avix.Width) / 2,
                                  (lcddev.height - avix.Height) / 2,
                                  avix.Width, avix.Height); /* JPG 解码初始化 */
            }
            else
            {
                res = mjpeg_init((lcddev.width - avix.Width) / 2,
                                  110 + (lcddev.height - 110 - avix.Height) / 2,
                                  avix.Width, avix.Height); /* JPG 解码初始化 */
            }
        }
    }
}
```

```
if (res)
{
    mjpeg_free();
    break;
}
if (avix.SampleRate) /* 有音频信息,才初始化 */
{
    /* 没有音频硬件,所以并不播放音频出来 */
}
while (1) /* 播放循环 */
{
    if (avix.StreamID == AVI_VIDS_FLAG) /* 视频流 */
    {
        pbuf = framebuf;
        /* 读入整帧+下一数据流 ID 信息 */
        f_read(favi, pbuf, avix.StreamSize + 8, &nr);
        res = mjpeg_decode(pbuf, avix.StreamSize);
        if (res)
        {
            printf("decode error!\r\n");
        }
        while (g_frameup == 0); /* 等待时间到达(在 TIM7 的中断里面设置为 1) */
        g_frameup = 0;           /* 标志清零 */
        g_frame++;
    }
    else if (avix.StreamID == AVI_AUDS_FLAG) /* 音频流 */
    {
        video_time_show(favi, &avix);           /* 显示当前播放时间 */
        f_read(favi, psaibuf, avix.StreamSize + 8, &nr); /* 填充 psaibuf */
        pbuf = psaibuf;
    }
    key = key_scan(0);
    /* KEY0/KEY1 按下,播放下一个/上一个视频 */
    if (key == KEY0_PRES || key == KEY1_PRES)
    {
        res = key;
        break;
    }
    else if (key == KEY1_PRES || key == WKUP_PRES)
    {
        video_seek(favi, &avix, framebuf);
        pbuf = framebuf;
    }
    if (avi_get_streaminfo(pbuf + avix.StreamSize)) /* 读取下一帧 流标志 */
    {
        pbuf = framebuf;
        res = f_read(favi, pbuf, AVI_VIDEO_BUF_SIZE, &nr); /* 开始读取 */
        /* 读取成功,且读取了指定长度的数据 */
        if (res == 0 && nr == AVI_VIDEO_BUF_SIZE)
        {
            /* 寻找 AVI_VIDS_FLAG,00dc */
            offset = avi_srarch_id(pbuf, AVI_VIDEO_BUF_SIZE, "00dc");
            avi_get_streaminfo(pbuf + offset); /* 获取流信息 */
            if (offset)f_lseek(favi,
                                (favi->fptr - AVI_VIDEO_BUF_SIZE) + offset + 8);
        }
        else
        {
            printf("g_frame error \r\n");
            res = KEY0_PRES;
            break;
        }
    }
}
```

```
TIM7->CR1 &= ~(1 << 0); /* 关闭定时器 7 */
lcd_set_window(0, 0, lcddev.width, lcddev.height); /* 恢复窗口 */
mjpeg_free(); /* 释放内存 */
f_close(favi);
}
}

myfree(SRAM4, psaibuf);
myfree(SRAMIN, framebuf);
myfree(SRAM12, favi);
return res;
}
```

从上面的代码中可以看出，主要就是逐帧的读取 AVI 文件的数据流和下一帧数据流的 ID 信息，如果当前数据流为视频流则调用函数 mjpegdec_decode() 对齐进行解码并在 LCD 上进行显示，若不是视频流则直接跳过，不进行处理，并且在每一帧数据流处理完毕后，都会检测一次按键，以进行上（下）一个视频切换播放的操作。

55.3 下载验证

在完成编译和烧录操作后，将根目录存放了 A 盘 → 5，SD 卡根目录文件中文件的 SD 卡插入开发板板载的 SD 卡卡座后，便能看到 LCD 上显示了 SD 卡 VIDEO 文件夹中的视频信息，并且在 LCD 上自动播放了该视频文件，此时，若按下 WKUP 按键或 KEY0 按键可以切换 LCD 显示播放 SD 卡 VIDEO 文件夹中的上一个或下一个视频。

第五十六章 FPU 测试（Julia 分形）实验

本章介绍使用 STM32H750 的 FPU 加速浮点运算。通过本章的学习，读者将学习到如何开启 STM32H750 的 FPU。

本章分为如下几个小节：

- 56.1 硬件设计
- 56.2 程序设计
- 56.3 下载验证

56.1 硬件设计

56.1.1 例程功能

1. 自动计算 Julia 图形并在 LCD 进行显示，同时显示单次计算耗时
2. 按下 WKUP 按键可增加缩放因子
3. 按下 KEY0 按键可切换自动和手动增加缩放因子
4. LED0 闪烁，提示程序正在运行

56.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
 - WKUP - PA0
 - KEY0 - PA15
4. FPU
5. TIM6
6. USART1
 - USART1_TX - PA9
 - USART1_RX - PA10

56.1.3 原理图

本章实验使用的 FPU 为 STM32H750 的片上资源，因此没有对应的连接原理图。

56.2 程序设计

56.2.1 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    uint8_t key;
    uint8_t i = 0;
    uint8_t autorun = 0;
    float time;
    char buf[50];

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480); /* 延时初始化 */
```

```

uart_init(115200);           /* 串口初始化为 115200 */
mpu_memory_protection();     /* 保护相关存储区域 */
led_init();                  /* 初始化 LED */
key_init();                  /* 初始化按键 */
lcd_init();                  /* 初始化 LCD */

btim_timx_int_init(65535, 24000 - 1); /* 10Khz 计数频率,最大计时 6.5 秒超出 */
lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "FPU TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 110, 200, 16, 16, "KEY0:+ KEY1:-", RED); /*显示提示信息*/
lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:AUTO/MANUL", RED);
delay_ms(1200);

julia_clut_init(g_color_map); /* 初始化颜色表 */
while (1)
{
    key = key_scan(0);
    switch (key)
    {
        case KEY0_PRES:
            i++;
            if (i > sizeof(zoom_ratio) / 2 - 1)i = 0; /* 限制范围 */
            break;
        case KEY1_PRES:
            if (i)i--;
            else i = sizeof(zoom_ratio) / 2 - 1;
            break;
        case WKUP_PRES:
            autorun = !autorun; /* 自动/手动 */
            break;
    }
    if (autorun == 1) /* 自动时,自动设置缩放因子 */
    {
        i++;
        if (i > sizeof(zoom_ratio) / 2 - 1)
        {
            i = 0; /* 限制范围 */
        }
    }
    lcd_set_window(0, 0, lcddev.width, lcddev.height); /* 设置窗口 */
    lcd_write_ram_prepare();
    BTIM_TIMX_INT->CNT = 0; /* 重设 TIM6 定时器的计数器值 */
    g_timeout = 0;
    julia_generate_fpu(lcddev.width, lcddev.height, lcddev.width / 2,
                       lcddev.height / 2, zoom_ratio[i]);
    time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
    sprintf(buf, "%s: zoom:%d runtime:%0.1fms\r\n", SCORE_FPU_MODE,
            zoom_ratio[i], time / 10);
    lcd_show_string(5, lcddev.height - 5 - 12, lcddev.width - 5,
                    12, 12, buf, RED); /* 显示当前运行情况 */
    printf("%s", buf); /* 输出到串口 */
    LED0_TOGGLE();
}
}

```

从上面的代码中可以看出,本实验就是使用函数 `julia_genetate_fpu()`实时产生 Julia 分形图形,并在 LCD 上进行显示,同时通过 TIM6 计算函数 `julia_generate_fpu()`的执行时间,并将执行时间在 LCD 上进行显示。

本章实验要对比开启和关闭 FPU 的实验效果,在 Options for Target 窗口的 Target 选项卡中可以开启或关闭使用 FPU,如下图所示:

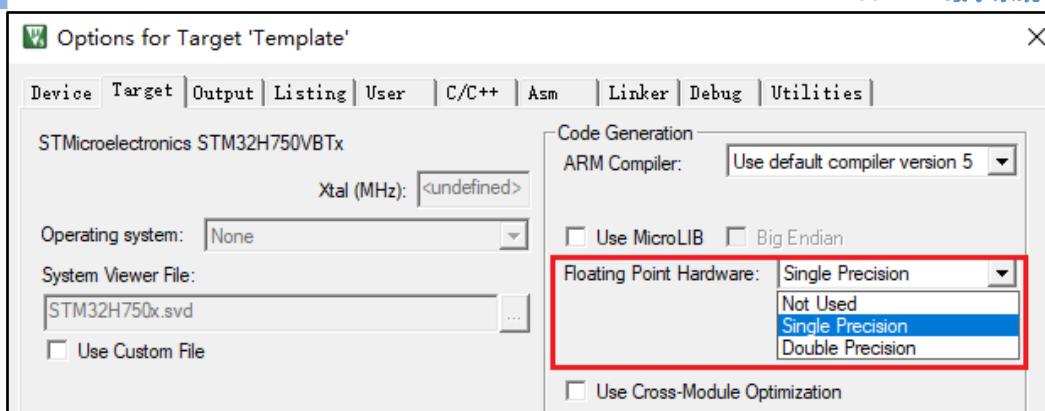


图 56.2.1.1 FPU 开关

上图中的 Floating Point Hardware 选择 “Not Used” 就表示不使用硬件 FPU，选择 “Single Precision” 就表示使用硬件 FPU。

56.3 下载验证

在 FPU 开启和关闭的情况下，分别完成编译和烧录操作后，可以看到，LCD 自动刷新显示了 Julia 分形图形，并且可以看到在不开启 FPU 的情况下生成并显示第一张 Julia 分形图像，大约耗时 3000+毫秒，而在开启 FPU 后生成并显示第一张 Julia 分形图像的耗时缩短为 300+毫秒，两者相差近 10 倍，由此可见使用 FPU 的优势。

通过可按下 WKUP 按键重新生成并显示不同缩放因子的 Julia 分形图像，也可按下 KEY0 按键使程序不断自动切换缩放因子，并重新生成和显示 Julia 分形图形。

第五十七章 DSP BasicMath 实验

本章介绍使用 DSP 库进行基本的数学运算。通过本章的学习，读者将学习到如何使用 DSP 库进行数学运算。

本章分为如下几个小节：

- 57.1 硬件设计
- 57.2 程序设计
- 57.3 下载验证

57.1 硬件设计

57.1.1 例程功能

1. 分别在使用和不使用 DSP 库的情况下进行正弦余弦计算，并分别在 LCD 上显示计算耗时
2. LED0 闪烁，提示程序正在运行

57.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块

57.1.3 原理图

本章实验使用的 DSP 库为软件库，因此没有对应的连接原理图。

57.2 程序设计

57.2.1 DPS 库的使用

本章实验使用 STM32H7 的 DSP 库源码以及 ST 提供的 HAL 库，将其添加到工程文件夹后，如下图所示：

```
./Drivers/CMSIS
|-- DSP
    '-- Include
        |-- arm_common_tables.h
        |-- arm_const_structs.h
        '-- arm_math.h
    '-- Device
        '-- ST
            '-- STM32H7xx
    '-- Include
        |-- cmsis_armcc.h
        |-- cmsis_armclang.h
        |-- cmsis_compiler.h
        |-- cmsis_version.h
        |-- core_cm7.h
        '-- mpu_armv7.h
    '-- Lib
        '-- ARM
            '-- arm_cortexM7lfdp_math.lib
```

图 57.2.1.1 DSP 库

DSP 中就提供了大量用于数学运算的函数，例如：正弦、余弦的计算等。使用 DSP 库能加快各种算法的实现，并且 DSP 中的函数是专门为 DSP 指令集做了相应的优化的，因此能以极高的效率在开启 FPU 且具有 DSP 指令集的 MCU 执行。

DSP 库的使用方法也很简单，仅需将 DSP 库文件添加到工程中，并包含相应的头文件，就

能调用 DSP 库中的函数了，具体请见本章实验的配套实验例程。

57.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
#define DELTA  0.0001f      /* 误差值 */

/**
 * @brief      sin cos 测试
 * @param      angle : 起始角度
 * @param      times : 运算次数
 * @param      mode  : 是否使用 DSP 库
 * @arg        0 , 不使用 DSP 库;
 * @arg        1 , 使用 DSP 库;
 *
 * @retval     无
 */
uint8_t sin_cos_test(float angle, uint32_t times, uint8_t mode)
{
    float sinx, cosx;
    float result;
    uint32_t i = 0;
    if (mode == 0)
    {
        for (i = 0; i < times; i++)
        {
            cosx = cosf(angle);           /* 不使用 DSP 优化的 sin, cos 函数 */
            sinx = sinf(angle);
            result = sinx * sinx + cosx * cosx; /* 计算结果应该等于 1 */
            result = fabsf(result - 1.0f);   /* 对比与 1 的差值 */
            if (result > DELTA) return 0xFF; /* 判断失败 */
            angle += 0.001f;               /* 角度自增 */
        }
    }
    else
    {
        for (i = 0; i < times; i++)
        {
            cosx = arm_cos_f32(angle);   /* 使用 DSP 优化的 sin, cos 函数 */
            sinx = arm_sin_f32(angle);
            result = sinx * sinx + cosx * cosx; /* 计算结果应该等于 1 */
            result = fabsf(result - 1.0f);   /* 对比与 1 的差值 */
            if (result > DELTA) return 0xFF; /* 判断失败 */
            angle += 0.001f;               /* 角度自增 */
        }
    }
    return 0; /* 任务完成 */
}

int main(void)
{
    float time;
    char buf[50];
    uint8_t res;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
    delay_init(480);             /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
}
```

```

lcd_init();                                /* 初始化 LCD */
btim_timx_int_init(65535, 24000 - 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, 120, 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);
while (1)
{
    /* 不使用 DSP 优化 */
    BTIM_TIMX_INT->CNT = 0; /* 重设 TIM3 定时器的计数器值 */
    g_timeout = 0;
    res = sin_cos_test(PI / 6, 1000000, 0);
    time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
    sprintf(buf, "%0.1fms\r\n", time / 10);
    if (res == 0)
    {
        /* 显示运行时间 */
        lcd_show_string(30 + 16 * 8, 120, 100, 16, 16, buf, BLUE);
    }
    else
    {
        /* 显示当前运行情况 */
        lcd_show_string(30 + 16 * 8, 120, 100, 16, 16, "error!", BLUE);
    }
    /* 使用 DSP 优化 */
    BTIM_TIMX_INT->CNT = 0; /* 重设 TIM3 定时器的计数器值 */
    g_timeout = 0;
    res = sin_cos_test(PI / 6, 1000000, 1);
    time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
    sprintf(buf, "%0.1fms\r\n", time / 10);
    if (res == 0)
    {
        /* 显示运行时间 */
        lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, buf, BLUE);
    }
    else
    {
        /* 显示错误 */
        lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, "error!", BLUE);
    }
    LED0_TOGGLE();
}
}

```

这里包括 2 个函数：sin_cos_test 和 main 函数，sin_cos_test 函数用于根据给定参数，执行 $\sin(x)^2+\cos(x)^2=1$ 的计算。计算完后，计算结果同给定的误差值（DELTA）对比，如果不大于误差值，则认为计算成功，否则计算失败。该函数可以根据给定的模式参数(mode)来决定使用哪个基础数学函数执行运算，从而得出对比。

main 函数则比较简单，这里我们通过定时器 3 来统计 sin_cos_test 运行时间，从而得出对比数据。主循环里面，每次循环都会两次调用 sin_cos_test 函数，首先采用不使用 DSP 库方式计算，然后采用使用 DSP 库方式计算，并得出两次计算的时间，显示在 LCD 上面。

sin_cos_test 函数使用 DSP 库计算正弦、余弦时，使用了 DPS 库中的函数 arm_cos_f32() 和函数 arm_sin_f32()，不使用 DSP 库正弦、余弦时，使用了 C 标准库中的函数 cosf() 和函数 sinf()。

DSP 基础数学函数测试的程序设计就讲解到这里。

57.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上分别显示了不使用 DSP 库和使用 DSP 库的运算耗时时间，从两者的耗时时间中，能看出使用 DSP 库进行数学运算有明显的优势。

第五十八章 DSP FFT 实验

本章将使用 DSP 进行快速傅里叶变换（FFT）的运算测试。通过本章的学习，读者将学习到 DSP 库中 FFT 运算的简单应用。

本章分为如下几个章节：

- 58.1 硬件设计
- 58.2 程序设计
- 58.3 下载验证

58.1 硬件设计

58.1.1 例程功能

1. 按下 WKUP 按键可进行 FFT 计算，并在 LCD 上显示计算耗时，同时通过串口输出计算结果
2. LED0 闪烁，提示程序正在运行

58.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
4. USART1
USART1_TX - PA9
USART1_RX - PA10

58.1.3 原理图

本章使用的 DSP 库为软件库，因此没有对应的连接原理图。

58.2 程序设计

58.2.1 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    float time;
    char buf[50];
    arm_cfft_radix4_instance_f32 scfft;
    uint8_t key, t = 0;
    uint16_t i;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480);             /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
}
```

```

btim_timx_int_init(65535, 24000 - 1); /* 10Khz 计数频率,最大计时 6.5 秒超出 */

lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
lcd_show_string(30, 70, 200, 16, 16, "DSP FFT TEST", RED);
lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
lcd_show_string(30, 130, 200, 16, 16, "KEY0:Run FFT", RED); /* 显示提示信息 */
lcd_show_string(30, 160, 200, 16, 16, "FFT runtime:", RED); /* 显示提示信息 */
/* 初始化 scfft 结构体, 设定 FFT 相关参数 */
arm_cfft_radix4_init_f32(&scfft, FFT_LENGTH, 0, 1);
while (1)
{
    key = key_scan(0);
    if (key == KEY0_PRES)
    {
        for (i = 0; i < FFT_LENGTH; i++) /* 生成信号序列 */
        {
            /* 生成输入信号实部 */
            fft_inputbuf[2 * i] = 100 +
                10 * arm_sin_f32(2 * PI * i / FFT_LENGTH) +
                30 * arm_sin_f32(2 * PI * i * 4 / FFT_LENGTH) +
                50 * arm_cos_f32(2 * PI * i * 8 / FFT_LENGTH);
            fft_inputbuf[2 * i + 1] = 0; /* 虚部全部为 0 */
        }
        BTIM_TIMX_INT->CNT = 0;; /* 重设 BTIM_TIMX_INT 定时器的计数器值 */
        g_timeout = 0;
        arm_cfft_radix4_f32(&scfft, fft_inputbuf); /* FFT 计算 (基 4) */
        /* 计算所用时间 */
        time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
        sprintf((char *)buf, "%0.3fms\r\n", time / 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。

58.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验的相关信息，此时若按下 WKUP 按键进行 FFT 运算测试，测可以看到，在 FFT 运算结束后，LCD 上显示了本次 FFT 运算的耗时长，同时串口调试助手也显示了本次 FFT 运算结果取模后的结果值。

第五十九章 手写识别实验

本章将介绍正点原子提供的手写识别库的使用，通过使用手写识别库可以实现一些简单的数字字母手写识别的应用。通过本章的学习，读者将学习到正点原子手写识别库的使用。

本章分为如下几个小节：

- 59.1 硬件设计
- 59.2 程序设计
- 59.3 下载验证

59.1 硬件设计

59.1.1 例程功能

1. LCD 上显示手写触摸轨迹，并自动识别手写输入
2. 按下 WKUP 按键可进行电容触摸屏触摸校准
3. 按下 KEY0 按键可切换手写识别模式
4. LED0 闪烁，提示程序正在运行

59.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. USART1
USART1_TX - PA9
USART1_RX - PA10
4. NOR Flash
QSPI FLASH 芯片，连接在 QSPI 上
5. SD
SDIO_D0 - PC8
SDIO_D1 - PC9
SDIO_D2 - PC10
SDIO_D3 - PC11
SDIO_SCK - PC12
SDIO_CMD - PD2
6. 24C02
IIC_SCL - PB10
IIC_SDA - PB11

59.1.3 原理图

本章实验使用的手写识别库为软件库，因此没有对应的连接原理图。

59.2 程序设计

59.2.1 手写识别库的使用

正点原子提供的手写识别库包含四个文件，分别为：ATKNCR_M_Vx.x.lib、ATKNCR_N_Vx.x.lib、atk_ncr.x 和 atk_ncr.h，本章实验配套的实验例程中已经提供了这四个文件，并且针对正点原子 M100Z-M4 最小系统板 STM32F407 版进行了移植适配，在使用时，仅需将 atk_ncr.c 和 atk_ncr.h 这两个文件添加到自己的工程中，并根据是否使用内存管理动态管理手写识别库运行时所需的内存，添加 ATKNCR_M_Vx.x.lib（使用内存管理）或 ATKNCR_N_Vx.x.lib

(不使用内存管理) 即可, 如下图所示:

```
./Middlewares/ATKNCR/
|-- ATKNCR_M_V2.0.lib
|-- ATKNCR_N_V2.0.lib
|-- atk_ncr.c
`-- atk_ncr.h
```

图 59.2.1.1 正点原子手写识别库文件

手写识别库中提供了手写识别初始化、识别和停止等函数, 使用非常方便。

在进行手写识别前, 需要使用手写识别初始化函数对其进行初始化, 手写识别初始化的使用示例, 如下所示:

```
#include "./ATKNCR/atk_ncr.h"

void example_fun(void)
{
    /* 初始化手写识别 */
    alientek_ncr_init();
}
```

手写识别初始化后, 便可开始进行手写识别, 手写识别函数的使用示例, 如下所示:

```
#include "./ATKNCR/atk_ncr.h"

void example_fun(void)
{
    /* 定义点阵数据缓存 */
    atk_ncr_point input_buf[200];
    int input_cnt;
    char output_buf[2];

    /* 初始化手写识别 */
    alientek_ncr_init();

    /* 从触摸屏或其他方式获取输入的点阵数据 */
    input_cnt = get_data(input_buf);

    /* 进行手写识别 */
    alientek_ncr(input_buf, cnt, 1, 1, output_buf);

    /* 处理识别结果 */
    /* printf("Detect result: %s\r\n", output_buf); */
}
```

在进行手写识别后, 若不再需要进行手写识别, 可以调用结束手写识别函数, 结束手写识别, 该函数的使用示例, 如下所示:

```
#include "./ATKNCR/atk_ncr.h"

void example_fun(void)
{
    /* 定义点阵数据缓存 */
    atk_ncr_point input_buf[200];
    int input_cnt;
    char output_buf[2];

    /* 初始化手写识别 */
    alientek_ncr_init();

    /* 从触摸屏或其他方式获取输入的点阵数据 */
    input_cnt = get_data(input_buf);

    /* 进行手写识别 */
    alientek_ncr(input_buf, cnt, 1, 1, output_buf);

    /* 处理识别结果 */
    /* printf("Detect result: %s\r\n", output_buf); */
```

```
/* 停止手写识别 */
alientek_ncr_stop();
}
```

59.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
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 */

    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 配置系统时钟，480Mhz */
    delay_init(480); /* 初始化延时功能 */
    usart_init(115200); /* 初始化串口 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    tp_dev.init(); /* 初始化触摸屏 */
    my_mem_init(SRAMIN); /* 初始化内部内存池(AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池(SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池(DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池(ITCM) */
    exfun_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() != 0) /* 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 STM32 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, "ATOM@ALIENTEK", 16, 0, RED);
    text_show_string(60, 70, 200, 16, "KEY0:MODE WK_UP: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);

        switch (key)
        {
            case KEY0_PRES: /* 切换识别模式 */

```

```
{  
    lcd_fill(20, 115, 219, 315, WHITE);  
    tcnt = 100;  
    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;  
        }  
        default:  
        {  
            break;  
        }  
    }  
  
    break;  
}  
case WKUP_PRES:  
{  
    if ((tp_dev.touchtype & 0x80) == 0)  
    {  
        tp_adjust();  
        goto RESTART;  
    }  
    break;  
}  
default:  
{  
    break;  
}  
}  
tp_dev.scan(0); /* 扫描 */  
  
if (tp_dev.sta & TP_PRES_DOWN) /* 有按键被按下 */  
{  
    delay_ms(1); /* 必要的延时，否则老认为有按键按下 */  
    tcnt = 0; /* 松开时的计数器清空 */  
  
    if (((tp_dev.x[0] < (lcddev.width - 20 - 2)) &&  
         (tp_dev.x[0] >= (20 + 2))) &&  
        ((tp_dev.y[0] < (lcddev.height - 5 - 2)) &&  
         (tp_dev.y[0] >= (115 + 2))))
```

```
{  
    if (lastpos[0] == 0xFFFF)  
    {  
        lastpos[0] = tp_dev.x[0];  
        lastpos[1] = tp_dev.y[0];  
    }  
  
    lcd_draw_bline(lastpos[0],  
                   lastpos[1],  
                   tp_dev.x[0],  
                   tp_dev.y[0],  
                   2,  
                   BLUE); /* 画线 */  
    lastpos[0] = tp_dev.x[0];  
    lastpos[1] = tp_dev.y[0];  
  
    if (pcnt < 200) /* 总点数少于 200 */  
    {  
        if (pcnt != 0)  
        { /* x,y 不相等 */  
            if ((ncr_input_buf[pcnt - 1].y != tp_dev.y[0]) &&  
                (ncr_input_buf[pcnt - 1].x != tp_dev.x[0]))  
            {  
                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);  
            alienteck_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 == 20)  
{  
    LED0_TOGGLE();  
    i = 0;  
}  
}  
}
```

由于本实验的应用代码过于冗长，因此上面代码仅保留了部分关键代码，完整代码请读者自行查看本实验的配套实验例程。从上面的代码中可以看出，本实验实现的应用中，通过触摸屏获取点阵数据，将点阵数据传入手写识别函数后，获取手写识别结果，然后将手写识别结果通过串口等方式输出，并且可通过 KEY0 按键修改手写识别的模式，也可通过 WKUP 按键随时进行电阻屏的触摸校准。

59.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验的相关实验信息，此时便可以在 LCD 上提示的“手写区”进行手写操作，完后手写操作后，可以看到 LCD 上提示了手写识别的结果，此时也可以按下 KEY0 按键修改手写识别的模式，也可以按下 WKUP 按键进行电阻屏的触摸校准。

第六十章 T9 拼音输入法实验

本章将介绍正点原子提供的 T9 拼音输入法库的使用，用其实现一个简单的 T9 拼音输入法应用。通过本章的学习，读者将学习到正点原子 T9 拼音输入法库的使用。

本章分为如下几个小节：

- 60.1 硬件设计
- 60.2 程序设计
- 60.3 下载验证

60.1 硬件设计

60.1.1 例程功能

1. 输入拼音后，LCD 和串口显示匹配结果
2. 按下 WKUP 按键可清除输入
3. 按下 KEY0 按键可换页匹配结果
4. LED0 闪烁，提示程序正在运行

60.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. USART1
USART1_TX - PA9
USART1_RX - PA10
4. NOR Flash
QSPI FLASH 芯片，连接在 QSPI 上
5. SD
SDIO_D0 - PC8
SDIO_D1 - PC9
SDIO_D2 - PC10
SDIO_D3 - PC11
SDIO_SCK - PC12
SDIO_CMD - PD2
6. 24C02
IIC_SCL - PB10
IIC_SDA - PB11

60.1.3 原理图

本章实验使用的 T9 拼音输入法库为软件库，因此没有对应的连接原理图。

60.2 程序设计

60.2.1 T9 拼音输入法库的使用

正点原子提供的 T9 拼音输入法库包含三个文件，分别为：pyinput.c、pyinput.h 和 pymb.h，本章实验配套的实验例程中已经提供了这三个文件，并且针对正点原子 M100Z-M7 最小系统板 STM32H750 版进行了移植适配，在使用时，仅需将这三个文件添加到自己的工程即可，如下图所示：

```
./Middlewares/T9INPUT/
|-- pyinput.c
|-- pyinput.h
`-- pymb.h
```

图 60.2.1.1 正点原子 T9 拼音输入法库文件

T9 拼音输入法库提供了匹配码表的函数，使用非常方便。

匹配码表函数的使用示例，如下所示：

```
#include "./T9INPUT/pyinput.h"

void example_fun(void)
{
    uint8_t inputstr[5];
    uint8_t res;
    uint8_t index;

    /* 拼音输入 */
    // inputstr[0] = 9; // wxyz
    // inputstr[1] = 4; // ghi
    // inputstr[2] = 3; // def
    // inputstr[3] = 6; // mno
    // inputstr[4] = 4; // ghi

    /* 匹配码表 */
    res = t9.getpymb(inputstr);
    /* 有匹配结果 */
    if (res != 0)
    {
        /* 展示所有匹配结果 */
        for (index=0; index<(res&0x7F); index++)
        {
            // printf("拼音: %s\r\n", t9.pymb[index]->py);
            // printf("结果: %s\r\n", t9.pymb[index]->pymb);
        }
    }
    /* 无匹配结果 */
    else
    {
        /* Do something */
    }
}
```

60.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    uint8_t t = 0;
    uint8_t result_num;
    uint8_t cur_index;
    uint8_t key;
    uint8_t pykey;
    uint8_t inputstr[7];
    uint8_t inputlen;

    sys_cache_enable();           /* 打开 L1-Cache */
    HAL_Init();                  /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */
    delay_init(480);             /* 延时初始化 */
    usart_init(115200);          /* 串口初始化为 115200 */
    usmart_dev.init(240);         /* 初始化 USMART */
    mpu_memory_protection();     /* 保护相关存储区域 */
    led_init();                  /* 初始化 LED */
```



```

lcd_init(); /* 初始化 LCD */
key_init(); /* 初始化按键 */
tp_dev.init(); /* 初始化触摸屏 */
my_mem_init(SRAMIN); /* 初始化内部内存池 (AXI) */
my_mem_init(SRAM12); /* 初始化 SRAM12 内存池 (SRAM1+SRAM2) */
my_mem_init(SRAM4); /* 初始化 SRAM4 内存池 (SRAM4) */
my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池 (DTCM) */
my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池 (ITCM) */
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 */

RESTART:
while (fonts_init() != 0) /* 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);
    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, "ATOM@ALIENTEK", 16, 0, RED);
text_show_string(30, 65, 200, 16, "WK_UP:清除", 16, 0, RED);
text_show_string(30, 85, 200, 16, "KEY0:翻页", 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;
    kbdysize = 43;
}
else if (lcddev.id == 0X5510)
{
    kbdxsize = 140;
    kbdysize = 70;
}
else
{
    kbdxsize = 60;
    kbdysize = 40;
}

py_load_ui(30, 195); /* 全部清零 */
my_mem_set(inputstr, 0, 7); /* 输入长度为 0 */
inputlen = 0; /* 总匹配数清零 */
result_num = 0;
cur_index = 0;

while (1)
{
    t++;
    pykey = py_get_keynum(30, 195); /* 得到触摸屏的输入 */
    if (pykey != 0) /* 有按键按下 */
    {
        if (pykey == 1) /* 删除键 */
        {
            if (inputstr[inputlen - 1] != '\0') /* 如果输入不为空 */
                inputstr[inputlen - 1] = '\0'; /* 移除最后一个字符 */
            else
                continue; /* 如果输入为空，忽略删除操作 */
            result_num--; /* 减少一个匹配数 */
        }
        else if (pykey == 2) /* 翻页键 */
        {
            if (inputlen > 0) /* 如果输入不为空 */
                result_num++; /* 增加一个匹配数 */
        }
    }
}

```

```
if (inputlen != 0)
{
    inputlen--;
}
inputstr[inputlen] = '\0';
}
Else /* 添加字符 */
{
    inputstr[inputlen] = pykey + '0';
    if (inputlen < 7)
    {
        inputlen++;
    }
}

if (inputstr[0] != NULL) /* 有字符，需要进行匹配 */
{
    pykey = t9.getpymb(inputstr); /* 获取匹配结果 */
    if (pykey != 0) /* 有匹配结果 */
    {
        result_num = pykey & 0x7F;
        cur_index = 1;
        if ((pykey & 0x80) != 0)
        {
            inputlen = pykey & 0x7F;
            inputstr[inputlen] = '\0';
            if (inputlen > 1)
            {
                result_num = t9.getpymb(inputstr);
            }
        }
    }
    Else /* 无匹配结果 */
    {
        inputlen--;
        inputstr[inputlen] = '\0';
    }
}
else
{
    cur_index = 0;
    result_num = 0;
}

lcd_fill(30 + 40, 105, 30 + 40 + 48, 110 + 16, WHITE);
lcd_show_num(30 + 144, 105, result_num, 1, 16, BLUE);
/* 显示有效的字符串 */
text_show_string(30+40, 105, 200, 16, (char *)inputstr, 16, 0, BLUE);
py_show_result(cur_index); /* 根据索引显示匹配结果 */
}

if (result_num != 0) /* 有匹配结果 */
{
    key = key_scan(0);
    switch (key)
    {
        case KEY0_PRES: /* 匹配结果翻页 */
        {
            if (cur_index < result_num)
            {
                cur_index++;
            }
            else
            {
                cur_index = 1;
            }
        }
    }
}
```

```
        }
        py_show_result(cur_index);
        break;
    }
    case WKUP_PRES:           /* 清除输入 */
    {
        lcd_fill(30 + 40, 145, lcddev.width - 1, 145 + 48, WHITE);
        goto RESTART;
    }
}
if (t == 20)
{
    LED0_TOGGLE();
    t = 0;
}
delay_ms(10);
}
```

由于本实验的应用代码过于冗长，因此上面代码仅保留了部分关键代码，完成代码请读者自行查看本实验的配套实验例程。从上面的代码中可以看出，本实验实验的应用中，通过触摸屏获取拼音的键值，然后调用函数 t9.getpymb()进行码表匹配，有匹配结果时，则在 LCD 上显示匹配结果，若有一个匹配结果，则可通过 KEY0 按键进行翻页，通过也可通过 WKUP 按键清除输入。

60.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验的相关实验信息，通过也显示了 T9 拼音的输入键盘，此时便可通过键盘输入拼音，随便可在 LCD 上看到输入拼音对应的汉字匹配结果，若有一个匹配结果，例如输入“64”，则会匹配到拼音“mi”和“ni”，此时按下 KEY0 按键，可对多个匹配结果进行翻页，若按下 WKUP 按键则会清除所有输入。

第六十一章 串口 IAP 实验

本章将介绍在 STM32H750 上使用串口进行 IAP，以实现简单的 IAP 功能。通过本章的学习，读者将学习到 IAP 的使用。

本章分为如下几个小节：

- 61.1 硬件设计
- 61.2 程序设计
- 61.3 下载验证

61.1 硬件设计

61.1.1 例程功能

1. 串口接收 APP 固件数据
2. 按下 WKUP 按键可将串口接收到的 APP 固件数据写入 Flash
3. 按下 KEY0 按键可执行写入 Flash 的 APP
4. LED0 闪烁，提示程序正在运行

61.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. 按键
WKUP - PA0
KEY0 - PA15
4. USART1
USART1_TX - PA9
USART1_RX - PA10

61.1.3 原理图

本章实验使用的 IAP 为软件算法，因此没有对应的连接原理图。

61.2 程序设计

61.2.1 IAP 的实现

本章实验提供了用于 IAP 的驱动代码，如下图所示：

```
./IAP Bootloader V1.0/User/IAP/
|-- iap.c
`-- iap.h
```

图 61.2.1.1 IAP 驱动代码

IAP 的实现最主要分为两个步骤，分别为加载 APP 固件至 Flash 和跳转到 APP 中运行，为此本实验实现了以上两个函数，如下所示：

```
void iap_write_appbin(uint32_t appxaddr, uint8_t *appbuf, uint32_t applen);
void iap_load_app(uint32_t appxaddr);
```

以上两个函数就分别实现了加载 APP 固件至 Flash 和跳转到 APP 运行的功能，这两个函数的使用示例，如下所示：

```
#include "./IAP/iap.h"

#define FLASH_APP_ADDR (0x08010000)
#define APP_MAX_SIZE (120*1024)
```

```

static uint8_t appbin[APP_MAX_SIZE];

void example_fun(void)
{
    uint32_t appsize;

    /* 通过串口等方式获取 APP 的二进制数据 */
    appsize = get_app_bin(appbin);

    /* 将 APP 的二进制数据写入 Flash 的指定地址中 */
    iap_write_appbin(FLASH_APP_ADDR, appbin, appsize);

    /* 跳转到 APP 在 Flash 中的起始地址运行 */
    iap_load_app(FLASH_APP_ADDR);
}

```

61.2.2 APP 工程修改

APP 程序要能够在指定的 Flash 起始地址运行，需要编译器在编译 APP 的时候知道 APP 将被保存在 Flash 的那个位置，对于 MDK 软件，可以在 Options for Target 窗口中设置，如下图所示：

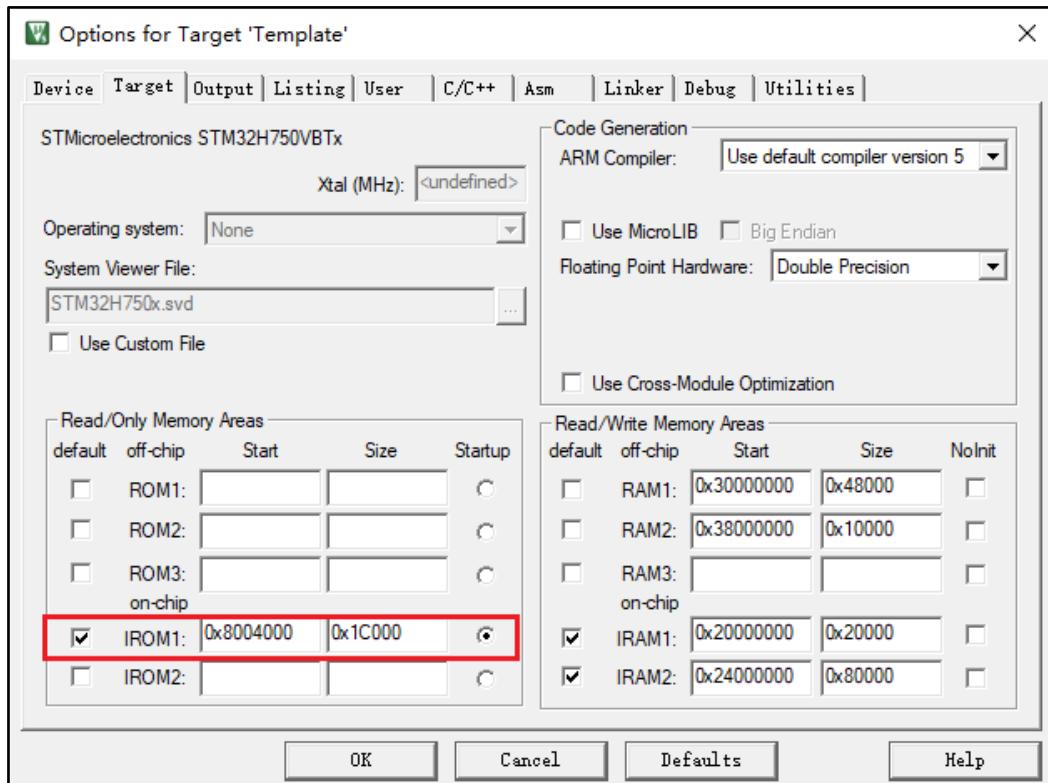


图 61.2.2.1 配置 MDK 中的 Flash 空间

默认的条件下，图中 IROM1 的起始地址(Start)一般为 0X08000000，大小(Size)为 0X20000，即从 0X08000000 开始的 128K 空间为我们的程序存储区。而图中，我们设置起始地址(Start)为 0X08004000，即偏移量为 0X4000 (16K 字节)，因而，留给 APP 用的 FLASH 空间(Size)只有 0X20000-0X2000=0X1C000 (112K 字节)大小了。设置好 Start 和 Size，就完成 APP 程序的起始地址设置。

这里的 16K 字节，一般是需要大家根据 Bootloader 程序大小进行选择，但是对 H750 来说，我们只保留了前 16K 给 IAP 用（详见前面的说明），因此必须保证内部 FLASH IAP 占用的内部 FLASH 不能超过 16K 大小。实际上，我们存放在内部 FLASH 的 IAP 代码实际上只做了一个拷贝操作（将 QSPI FLASH 的 IAP 程序拷贝到 DTCM 去运行），因此占用的 FLASH 非常小，因此 16K 的内存空间是够用的。

上一小节中将 APP 保存到 Flash 中的操作操作的是二进制数据，因此 APP 也应当被编译为二进制文件，然后 MDK 软件中并没有直接编译出二进制文件的选项，因此需要配置 MDK 在编译完成后执行指定的命令，如下图所示：

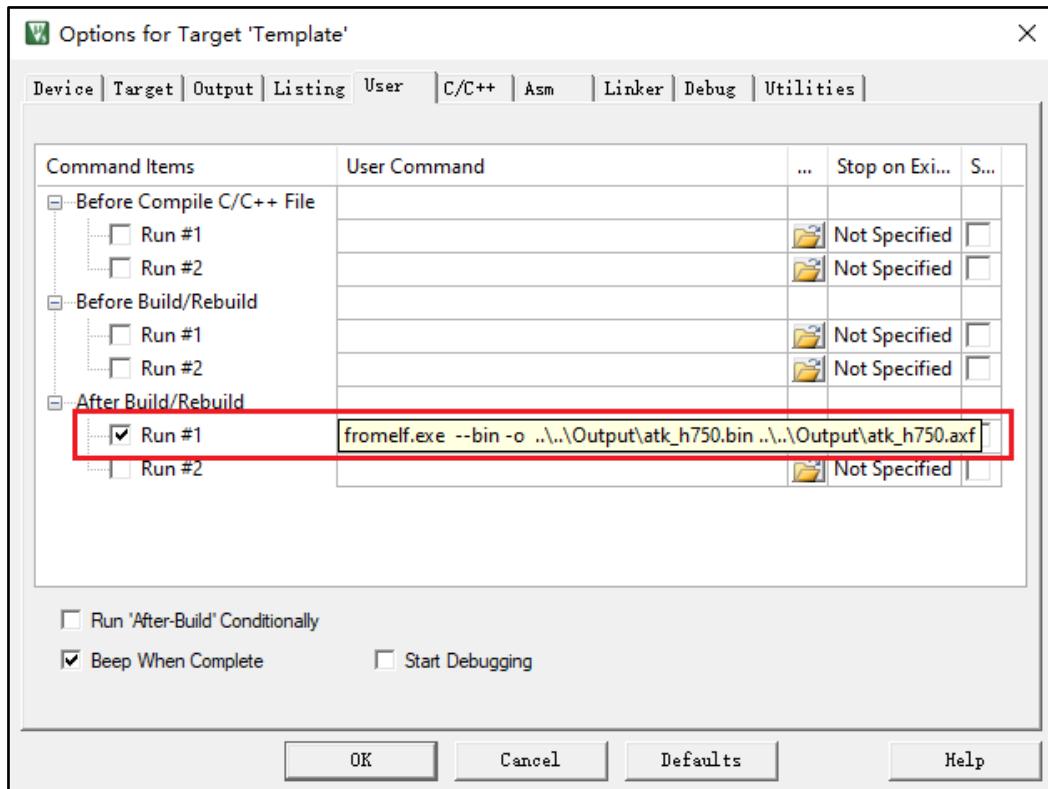


图 61.2.2.2 配置 MDK 生成二进制文件

完成以上配置后，便可在 APP 程序编译完成后在工程的 Output 目录下得到正确的二进制文件。

61.2.3 实验应用代码

本章实验的应用代码，如下所示：

```
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;
    uint16_t id = 0;

    sys_cache_enable();             /* 打开 L1-Cache */
    HAL_Init();                    /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟，480Mhz */

    /* 中断向量表放在 ITCM RAM,无偏移 */
    sys_nvic_set_vector_table(D1_ITCMRAM_BASE, 0);

    delay_init(480);               /* 延时初始化 */
    usart_init(115200);            /* 串口初始化为 115200 */
    mpu_memory_protection();       /* 保护相关存储区域 */
    led_init();                    /* 初始化 LED */
    lcd_init();                   /* 初始化 LCD */
    key_init();                   /* 初始化按键 */
    norflash_init();              /* NORFLASH(W25Q64) 初始化 */
}
```

```
id = norflash_read_id(); /* 读取 FLASH ID */

while ((id == 0) || (id == 0xFFFF)) /* 检测不到 FLASH 芯片 */
{
    lcd_show_string(30, 150, 200, 16, 16, "FLASH Check Failed!", RED);
    delay_ms(500);
    lcd_show_string(30, 150, 200, 16, 16, "Please Check!      ", RED);
    delay_ms(500);
    LED0_TOGGLE();
}

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, "KEY0:Run FLASH APP", RED);

while (1)
{
    if (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 */
        if (oldcount == g_usart_rx_cnt)
        {
            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(10);

    if (t == 30)
    {
        LED0_TOGGLE();
        t = 0;

        if (clearflag)
        {
            clearflag--;
            if (clearflag == 0)
            {
                lcd_fill(30, 210, 240, 210 + 16, WHITE); /* 清除显示 */
            }
        }
    }

    key = key_scan(0);

    /* 注意:QSPI FLASH 不做任何校验和检查,所以请确保数据正确!否则将运行出错 */
    if (key == WKUP_PRES) /* WKUP 按下,更新固件到外部 QSPI FLASH */
    {
        if (applenth)
        {
            printf("开始更新固件...\r\n");
            lcd_show_string(30, 210, 200, 16, 16, "Copying APP2QSPI...", BLUE);
            norflash_write((uint8_t *)g_usart_rx_buf,
                          QSPI_APP1_ADDR - 0x90000000,
                          applenth); /* 写入 QSPI FLASH! */
            lcd_show_string(30, 210, 200, 16, 16, "Copy APP Successed!!", BLUE);
            printf("固件更新完成!\r\n");
        }
    }
}
```

```
    }
    else
    {
        printf("没有可以更新的固件!\r\n");
        lcd_show_string(30, 210, 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 */
}
/* KEY0 按键按下，更新固件到内部 FLASH，更新完后自动运行 FLASH APP 代码 */
if (key == KEY0_PRES)
/* 注意：如果 APP 还有 QSPI FLASH 部分代码，则必须先更新 QSPI FLASH */
{
    if (applenth)
    {
        printf("开始更新固件...\r\n");
        lcd_show_string(30, 210, 200, 16, 16, "Copying APP2FLASH...", BLUE);
        /* 判断是否为 0X08XXXXXX */
        if (((*(volatile uint32_t *)  
        (SRAM_APP1_ADDR + 4)) & 0xFF000000) == 0x08000000)  

        { /* 更新 FLASH 代码 */  

            iap_write_appbin(FLASH_APP1_ADDR, g_usart_rx_buf, applenth);  

            lcd_show_string(30, 210, 200, 16, 16, "Copy APP Successed!!", BLUE);  

            printf("固件更新完成!\r\n");
            delay_ms(500);
            printf("开始执行 FLASH 用户代码!!\r\n\r\n");
            delay_ms(10);
            /* 判断是否为 0X08XXXXXX */
            if (((*(volatile uint32_t *)  
            (FLASH_APP1_ADDR + 4)) & 0xFF000000) == 0x08000000)  

            {
                iap_load_app(FLASH_APP1_ADDR); /* 执行 FLASH APP 代码 */
            }
            else
            {
                printf("非 FLASH 应用程序，无法执行!\r\n");
                lcd_show_string(30, 210, 200, 16, 16, "Illegal FLASH APP!", BLUE);
            }
        }
        else
        {
            lcd_show_string(30, 210, 200, 16, 16, "Illegal FLASH APP! ", BLUE);
            printf("非 FLASH 应用程序!\r\n");
        }
    }
    /* 判断 FLASH 里面是否有 APP，有的话执行 */
    else if (((*(volatile uint32_t *)  
        (FLASH_APP1_ADDR + 4)) & 0xFF000000) == 0x08000000)  

    {
        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("没有可以更新的固件!\r\n");
        lcd_show_string(30, 210, 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 */
}
}
```

该段代码，实现了串口数据处理，以及 IAP 更新和跳转等各项操作。DTCM IAP Bootloader

程序就设计完成了，一般要求 bootloader 程序越小越好（给 APP 节省空间），这里由于 H750 的 DTCM 比较大（128KB），而且存储在外部 QSPI FLASH（预留 1MB），所以是足够放代码的，但是在实际应用时，可以根据需要尽量精简代码来得到最小的 IAP。

因为存放在 RAM 的 DTCM 代码是无法掉电保存的，因此我们需要将 DTCM IAP Bootloader 编译生成.bin 文件（二进制文件），然后把.bin 转成 c 数组，添加到内部 FLASH IAP Bootloader 程序里面，由 FLASH IAP Bootloader 保存到外部 QSPI FLASH，并在每次复位时执行将 DTCM IAP 程序拷贝到 DTCM，并运行的操作，从而实现 IAP 功能。

6.1.3 下载验证

在完成编译和烧录操作后，可以看到 LCD 上显示了本实验的相关实验信息，此时程序正在等待串口接收 APP 的二进制数据，此时便可通过串口调试助手将实现编译生成的 APP 二进制文件发送给 MCU，待发送完毕后按下 WKUP 按键，将 APP 的二进制数据写入到 Flash 中，若写入成功，LCD 上将会有相应的提示，此时便可按下 KEY0 按键将程序跳转到 APP 的保存位置中运行，若指定的 Flash 位置保存了正确的 APP 固件，那么便可看到 MCU 正在运行 APP 程序，而非 IAP（Bootloader）程序。

第六十二章 USB 读卡器（Slave）实验

本章将介绍使用 STM32H750 作为 USB 从设备，模拟出 USB 读卡器，使得 PC 与 STM32H750 通过 USB 连接后，能够访问与 STM32H750 连接的 SD 中的数据。通过本章的学习，读者将学习到 STM32H750 作为 USB 从设备模拟出 USB 读卡器的使用。

本章分为如下几个小节：

- 62.1 硬件设计
- 62.2 程序设计
- 62.3 下载验证

62.1 硬件设计

62.1.1 例程功能

1. LCD 上显示 SD 卡和 NOR Flash 容量
2. 当通过 USB 接口连接至 USB Host 时，LCD 上显示 USB 连接状态和读写状态，同时 USB Host 设备可对 SD 卡和 NOR Flash 中的文件进行操作
3. LED0 闪烁，提示程序正在运行

62.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. NOR Flash
 - QSPI FLASH 芯片，连接在 QSPI 上
4. SD
 - SDIO_D0 - PC8
 - SDIO_D1 - PC9
 - SDIO_D2 - PC10
 - SDIO_D3 - PC11
 - SDIO_SCK - PC12
 - SDIO_CMD - PD2
5. USB
 - USB_DM - PA11
 - USB_DP - PA12

62.1.3 原理图

本章实验使用 USB 接口与 PC 进行连接，开发板板载了一个 USB 接口，用于连接其他 USB 设备，USB 接口与 MCU 的连接原理图，如下图所示：

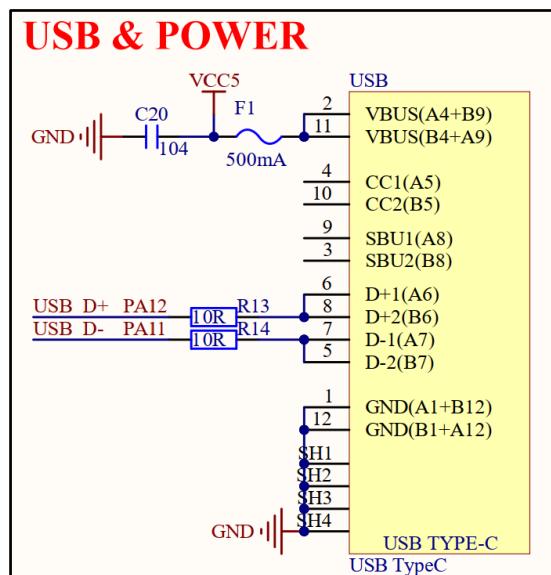


图 62.1.3.1 USB 接口与 MCU 的连接原理图

62.2 程序设计

62.2.1 ST 的 USB 设备驱动库

ST 针对 STM32H7 的 USB 设备驱动库为 STM32_USB_Device_Library，该 USB 设备驱动库由 ST 提供，如下图所示：

```
./Middlewares/USB/STM32_USB_Device_Library/Class
|-- AUDIO
|-- CDC
|-- CustomHID
|-- DFU
|-- HID
|-- MSC
`-- Template
```

图 62.2.1.1 STM32_USB_Device_library

从上图中也可以看出，ST 提供了 USB 主机和设备的驱动库，本书主要介绍 USB 设备的应用，因此主要关注 STM32_USB_Device_Library。

本书不展开讲解 USB 的相关原理，若读者想更加深入地了解 USB，请自行查阅相关资料。

62.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
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_sto;
    uint16_t id;
    uint64_t card_capacity;                  /* SD 卡容量 */

    sys_cache_enable();                     /* 打开 L1-Cache */
    HAL_Init();                            /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);    /* 设置时钟，480Mhz */
}
```

```
delay_init(480);                                /* 延时初始化 */
usart_init(115200);                            /* 串口初始化为 115200 */
mpu_memory_protection();                         /* 保护相关存储区域 */
led_init();                                     /* 初始化 LED */
lcd_init();                                     /* 初始化 LCD */
key_init();                                     /* 初始化按键 */
my_mem_init(SRAMIN);                            /* 初始化内部内存池 (AXI) */
my_mem_init(SRAM12);                            /* 初始化 SRAM12 内存池 (SRAM1+SRAM2) */
my_mem_init(SRAM4);                             /* 初始化 SRAM4 内存池 (SRAM4) */
my_mem_init(SRAMDTCM);                           /* 初始化 DTCM 内存池 (DTCM) */
my_mem_init(SRAMITCM);                           /* 初始化 ITCM 内存池 (ITCM) */

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, 130, 200, 16, 16, "SD Card Error!", RED);
}
else /* SD 卡正常 */
{
    lcd_show_string(30, 130, 200, 16, 16, "SD Card Size:      MB", RED);
    card_capacity = (uint64_t)(g_sd_card_info_handle.LogBlockNbr
        * (uint64_t)(g_sd_card_info_handle.LogBlockSize)); /* 计算 SD 卡容量 */
    lcd_show_num(134, 130, card_capacity >> 20, 5, 16, RED); /* 显示 SD 卡容量 */
}

id = norflash_ex_read_id();
if ((id == 0) || (id == 0xFFFF))
{
    /* 检测 W25Q128 错误 */
    lcd_show_string(30, 130, 200, 16, 16, "BY25Q128 Error!", RED);
}
else /* SPI FLASH 正常 */
{
    lcd_show_string(30, 150, 200, 16, 16, "SPI FLASH Size:7.25MB", RED);
}

/* 提示正在建立连接 */
lcd_show_string(30, 190, 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); /* 为 MSC 类添加回调函数 */
USBD_Start(&USBD_Device);                         /* 开启 USB */
delay_ms(1800);
while (1)
{
    delay_ms(1);
    if (usb_sto != g_usb_state_reg) /* 状态改变了 */
    {
        lcd_fill(30, 210, 240, 210 + 16, WHITE); /* 清除显示 */
        if (g_usb_state_reg & 0x01) /* 正在写 */
        {
            LED1(0);
            /* 提示 USB 正在写入数据 */
            lcd_show_string(30, 210, 200, 16, 16, "USB Writing...", RED);
        }
        if (g_usb_state_reg & 0x02) /* 正在读 */
        {
            LED1(0);
            /* 提示 USB 正在读出数据 */
        }
    }
}
```



```

        lcd_show_string(30, 210, 200, 16, 16, "USB Reading...", RED);
    }
    if (g_usb_state_reg & 0x04)
    {
        /* 提示写入错误 */
        lcd_show_string(30, 230, 200, 16, 16, "USB Write Err ", RED);
    }
    else
    {
        lcd_fill(30, 230, 240, 230 + 16, WHITE); /* 清除显示 */
    }

    if (g_usb_state_reg & 0x08)
    {
        /* 提示读出错误 */
        lcd_show_string(30, 250, 200, 16, 16, "USB Read Err ", RED);
    }
    else
    {
        lcd_fill(30, 250, 240, 250 + 16, WHITE); /* 清除显示 */
    }
    usb_sto = g_usb_state_reg; /* 记录最后的状态 */
}
if (device_sto != g_device_state)
{
    if (g_device_state == 1)
    {
        /* 提示 USB 连接已经建立 */
        lcd_show_string(30, 190, 200, 16, 16, "USB Connected     ", RED);
    }
    else
    {
        /* 提示 USB 被拔出了 */
        lcd_show_string(30, 190, 200, 16, 16, "USB DisConnected ", RED);
    }
    device_sto = g_device_state;
}
tct++;
if (tct == 200)
{
    tct = 0;
    LED1(1);           /* 关闭 LED1 */
    LED0_TOGGLE();      /* LED0 闪烁 */

    if (g_usb_state_reg & 0x10)
    {
        offline_cnt = 0; /* USB 连接了，则清除 offline 计数器 */
        g_device_state = 1;
    }
    else /* 没有得到轮询 */
    {
        offline_cnt++;
        if (offline_cnt > 10)
        {
            g_device_state = 0; /* 2s 内没收到在线标记，代表 USB 被拔出了 */
        }
    }
    g_usb_state_reg = 0;
}
}
}

```

从上面的代码中可以看出，在调用完 USB 设备驱动库中的相关初始化函数后并不断地获取 USB 的读写状态和连接状态，并在 LCD 上进行显示。

在调用完 USB 相关的初始化函数后，USB 设备驱动库便会自动模拟出一个 USB 读卡器，

当然也需要实现配置好 USB 读卡器模拟出的设备信息，以及读写、初始化 SD 卡、NOR Flash 等的操作，这些操作全部在 `usbd_storage.c` 文件中完成了，请读者自行参考该文件。

62.3 下载验证

在完成编译和烧录操作后，将 SD 卡正确插入开发板板载的 SD 卡卡座，可以看到 LCD 上显示了 SD 卡的容量信息以及 USB 的连接状态，若开发板还未通过 USB 接口与 PC 进行连接，那么 LCD 上会有 USB 连接断开的提示，此时可以将开发板通过 USB 接口与 PC 进行连接，稍等一会后，可以看到 LCD 上显示了 USB 已连接的提示，并且 PC 上也多出了两个磁盘，磁盘中的文件就是 SD 卡和 NOR Flash 中的文件，并且在 PC 上也可以直接对 SD 卡和 NOR Flash 中的文件进行读写操作。

第六十三章 USB 虚拟串口（Slave）实验

本章将介绍使用 STM32H750 作为 USB 从设备，虚拟出串口与 PC 进行通信。通过本章的学习，读者将学习到 STM32H750 作为 USB 从设备虚拟出串口的使用。

本章分为如下几个小节：

- 63.1 硬件设计
- 63.2 程序设计
- 63.3 下载验证

63.1 硬件设计

63.1.1 例程功能

1. 回显虚拟串口接收到的数据
2. 每间隔一定时间，虚拟串口发送一段提示信息
3. LED0 闪烁，提示程序正在运行

63.1.2 硬件资源

1. LED
LED0 - PE5
2. 正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块
3. USB
USB_DM - PA11
USB_DP - PA12

63.1.3 原理图

请见第 60.1.3 小节中 USB OTG 接口与 MCU 的连接原理图的相关内容。

63.2 程序设计

63.2.1 ST 的 USB 设备驱动库

请见第 62.2.1 小节中 ST 的 USB 设备驱动库的相关内容。

63.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
USBH_HandleTypeDef hUSBHost; /* USB Host 处理结构体 */

static void USBH_UserProcess(USBH_HandleTypeDef *phost, uint8_t id)
{
    uint32_t total, free;
    uint8_t res = 0;
    printf("id:%d\r\n", id);

    switch (id)
    {
        case HOST_USER_SELECT_CONFIGURATION:
            break;

        case HOST_USER_DISCONNECTION:
            f_mount(0, "2:", 1); /* 卸载 U 盘 */
            text_show_string(30, 140, 200, 16, "设备连接中...", 16, 0, RED);
    }
}
```

```
lcd_fill(30, 160, 239, 220, WHITE);
break;

case HOST_USER_CLASS_ACTIVE:
    text_show_string(30, 140, 200, 16, "设备连接成功!", 16, 0, RED);
    f_mount(fs[2], "2:", 1); /* 重新挂载 U 盘 */
    res = exfun_get_free("2:", &total, &free);
    if (res == 0)
    {
        lcd_show_string(30, 160, 200, 16, 16, "FATFS OK!", BLUE);
        lcd_show_string(30, 180, 200, 16, 16, "U Disk Total Size: MB",
                        BLUE);
        lcd_show_string(30, 200, 200, 16, 16, "U Disk Free Size: MB",
                        BLUE);
        /* 显示 U 盘总容量 MB */
        lcd_show_num(174, 180, total >> 10, 5, 16, BLUE);
        lcd_show_num(174, 200, free >> 10, 5, 16, BLUE);
    }
    else
    {
        printf("U 盘存储空间获取失败\r\n");
    }
    break;

case HOST_USER_CONNECTION:
    break;

default:
    break;
}

int main(void)
{
    uint8_t t = 0;
    sys_cache_enable(); /* 打开 L1-Cache */
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4); /* 设置时钟, 480Mhz */
    delay_init(480); /* 延时初始化 */
    usart_init(115200); /* 串口初始化为 115200 */
    mpu_memory_protection(); /* 保护相关存储区域 */
    led_init(); /* 初始化 LED */
    lcd_init(); /* 初始化 LCD */
    key_init(); /* 初始化按键 */
    my_mem_init(SRAMIN); /* 初始化内部内存池(AXI) */
    my_mem_init(SRAM12); /* 初始化 SRAM12 内存池(SRAM1+SRAM2) */
    my_mem_init(SRAM4); /* 初始化 SRAM4 内存池(SRAM4) */
    my_mem_init(SRAMDTCM); /* 初始化 DTCM 内存池(DTCM) */
    my_mem_init(SRAMITCM); /* 初始化 ITCM 内存池(ITCM) */
    exfun_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, "STM32", 16, 0, RED);
    text_show_string(30, 70, 200, 16, "USB U 盘 实验", 16, 0, RED);
    text_show_string(30, 90, 200, 16, "正点原子@ALIENTEK", 16, 0, RED);
}
```

```

text_show_string(30, 120, 200, 16, "设备连接中...", 16, 0, RED);
USBH_Init(&hUSBHost, USBH_UserProcess, 0);
USBH_RegisterClass(&hUSBHost, USBH_MSC_CLASS);
USBH_Start(&hUSBHost);
while (1)
{
    USBH_Process(&hUSBHost);
    delay_ms(10);
    t++;
    if (t == 50)
    {
        t = 0;
        LED0_TOGGLE();
    }
}
}

```

}从上面的代码中可以看出,在调用完USB设备驱动库中相关的初始化函数后便等待USB虚拟串口接收数据,若接收到数据则通过函数usb_printf()将数据发送出去,usb_printf()的实现如下所示:

```

/**
 * @brief 通过 USB 格式化输出函数
 * @note 通过 USB VCP 实现 printf 输出
 * 确保一次发送数据长度不超过 USB_USART_REC_LEN 字节
 * @param 格式化输出
 * @retval 无
 */
void usb_printf(char *fmt, ...)
{
    uint16_t i;
    va_list ap;
    va_start(ap, fmt);
    vsprintf((char *)g_usb_usart_printf_buffer, fmt, ap);
    va_end(ap);
    i = strlen((const char *)g_usb_usart_printf_buffer); /* 此次发送数据的长度 */
    cdc_vcp_data_tx(g_usb_usart_printf_buffer, i); /* 发送数据 */
}

```

63.3 下载验证

在完成编译和烧录操作后,可以看到LCD上显示了本实验的相关实验信息,此时可以将开发板通过USB接口与PC进行连接,待PC自动完成驱动安装后,便可看到PC上多出了一个端口设备,此时可以通过串口调试助手与该端口进行通信,可以看到无论串口调试助手发送任何数据,都会在串口调试助手的接受区看到发送出去的数据,这是因为STM32H750将虚拟串口接收到的数据通过虚拟串口发送回去,实现了“回显”的效果。

第六十四章 FreeRTOS 移植实验

前面章节中的实验都是在裸机环境下实现，本章将介绍 FreeRTOS 的简单使用，对 RTOS 感兴趣的读者，可以参考正点原子推出的 FreeRTOS 或 μC/OS-III 的全套教程资料，包含教学视频、开发指南并配套有例程源码。通过本章的学习，读者将学习到 FreeRTOS 的简单应用。

本章分为如下几个小节：

- 64.1 硬件设计
- 64.2 程序设计
- 64.3 下载验证

64.1 硬件设计

64.1.1 例程功能

1. LED0 由 LED0 任务控制 80 毫秒亮，920 毫秒灭
2. LED1 由 LED1 任务控制 300 毫秒亮，300 毫秒灭

64.1.2 硬件资源

1. LED
 - LED0 - PE5
 - LED1 - PE6

64.1.3 原理图

本章实验使用的 FreeRTOS 为软件库，因此没有对应的连接原理图。

64.2 程序设计

64.2.1 FreeRTOS 的移植

FreeRTOS 的源码可参考正点原子推出的《FreeRTOS 开发指南》或相关的配套教学视频至 FreeRTOS 的官网下载，也可在 A 盘 → 6，软件资料 → FreeRTOS 学习资料中找到。

FreeRTOS 的移植请读者参考正点原子推出的《FreeRTOS 开发指南》或相关的配套教学视频，在上述资料中，对 FreeRTOS 的移植和使用都做了非常详细地介绍，是入门 FreeRTOS 的一个非常好的选择。

64.2.2 实验应用代码

本章实验的应用代码，如下所示：

```
int main(void)
{
    sys_cache_enable();                                /* 打开 L1-Cache */
    HAL_Init();                                       /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);              /* 配置时钟，480MHz */
    delay_init(480);                                  /* 初始化延时 */
    usart_init(115200);                             /* 初始化串口 */
    led_init();                                      /* 初始化 LED */
    mpu_memory_protection();                         /* 保护相关存储区域 */

    xTaskCreate( (TaskFunction_t)start_task,           /* 任务函数 */
                (const char*)"start_task",             /* 任务名称 */
                (uint16_t)START_STK_SIZE,               /* 任务堆栈大小 */
                (void*)NULL,                           /* 传入给任务函数的参数 */
                /* 任务函数 */
                /* 任务名称 */
                /* 任务堆栈大小 */
                /* 传入给任务函数的参数 */

```

```

        (UBaseType_t      )START_TASK_PRIO,           /* 任务优先级 */
        (TaskHandle_t*   )&StartTask_Handler);    /* 任务句柄 */
vTaskStartScheduler();
}

```

从上面的代码中可以看出，在完成必要的初始化后，便创建了“start_task”任务，随后便开始了任务调度，“start_task”任务的入口函数为函数 start_task()，如下所示：

```

/***
 * @brief    start_task
 * @param    pvParameters: 传入参数(未用到)
 * @retval   无
 */
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();                                /* 进入临界区 */

    /* 创建 LED0 任务 */
    xTaskCreate( (TaskFunction_t )led0_task,
                (const char*     )"led0_task",
                (uint16_t       )LED0_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LED0_PRIO,
                (TaskHandle_t*  )&LED0Task_Handler);

    /* 创建 LED1 任务 */
    xTaskCreate( (TaskFunction_t )led1_task,
                (const char*     )"led1_task",
                (uint16_t       )LED1_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )LED1_PRIO,
                (TaskHandle_t*  )&LED1Task_Handler);

    vTaskDelete(StartTask_Handler);                      /* 删除开始任务 */
    taskEXIT_CRITICAL();                               /* 退出临界区 */
}

```

从上面的代码中可以看出，“start_task”任务主要用于创建“led0_task”和“led1_task”任务，这两个任务的入口函数，如下所示：

```

/***
 * @brief    LED0 任务
 * @param    pvParameters: 传入参数(未用到)
 * @retval   无
 */
void led0_task(void *pvParameters)
{
    while(1)
    {
        LED0(0);
        vTaskDelay(80);
        LED0(1);
        vTaskDelay(920);
    }
}

/***
 * @brief    LED1 任务
 * @param    pvParameters: 传入参数(未用到)
 * @retval   无
 */
void led1_task(void *pvParameters)
{
    while(1)
    {
        LED1(0);
        vTaskDelay(300);
    }
}

```

```
    LED1(1);  
    vTaskDelay(300);  
}
```

可以看到，两个任务分别控制开发板板载的两个 LED 闪烁，但其闪烁的频率和亮灭时间都不相同，因此因该能看到开发板板载的两个 LED 以不同的频率和不同的亮灭时间进行闪烁。

64.3 下载验证

在完成编译和烧录操作后，可以看到开发板板载的两个 LED 以不用的频率和不同的亮灭时间闪烁着，其中 LED0 在一个闪烁周期内的亮灭时间约分别为 80 毫秒和 920 毫秒，LED1 在一个闪烁周期内的亮灭时间约分别为 300 毫秒和 300 毫秒。