Elink407 用户手册

www.bangkr.cn

版本： 1.0.0-alpha

2015.04.01

版本

0.1 文档版本

下面是每次的版本变化清单，你也可以在下面站点上查看最新版本： http://elink407.bangkr.cn/

0.1.1 2015年2月14日–1.0.0

建立初始版本，60页左右。

确定初始的目录结构。

确定基本的内容结构。

确定排版书的格式 。

0.2 开发板版本

开发板各版本差异说明

** 本文档的版权归*所有。任何公司或者个人未经许可，不得 将本文档用于商业目的

目录

版本 i

0.1 文档版本 i

0.1.1 2015年2月14日 1.0.0 i

0.2 开发板版本 i

目录 iii

1 ELink407 硬件平台 1

1.1 开发板简介 1

1.1.1 STM32407VG片上资源(芯片功能) 1

1.1.2 板载资源 2

1.2 开发板硬件详解 3

1.2.1 MCU 3

1.2.2 启动模式 5

1.2.3 复位电路 6

1.2.4 电源管理 6

1.2.5 ST-LINK 6

1.2.6 USB To RS232 8

1.2.7 ADC & DHT11 8

1.2.8 USBOTG 8

1.2.9 LCD & Touch 9

1.2.10 1062 9

1.2.11 SDcard 10

1.2.12 Key & LED 10

1.2.13 IIC 12

1.2.14 Connector 12

血压计 12

Ethernet 12

Camera 13

SPI 14

IIS 14

UART 14

CAN 14

arduino 14

扩展GPIO接口 16

1.3 开发板使用注意事项 16

1.3.1 基本注意事项 16
1.3.2 跳线对照表 17

2 Elink407开发板的使用 19
2.1 串口工具的安装使用 19
2.1.1 常用串口工具软件 19
2.1.2 USB2UART驱动的安装 19
2.1.3 串口工具的安装使用 19
2.2 二进制程序烧写方法 23
2.2.1 使用 STM32 STLINK Utility 工具烧写 23
2.2.2 使用 STM32 and STM8 Flash loader demonstrator 工具烧写 . . . 24
2.2.3 使用 FlyMcu 程序烧写 24
2.3 开发板测试程序烧写与使用 32

3 软件开发环境 33
3.1 STM32软件开发基本知识 33
3.1.1 STM32编程基础 33
3.1.2 C语言 34
位操作 34
struct 35
define 36
ifdef 条件编译 37
typedef 37
extern 38
static 39
volatile 39
3.2 ST 固件库 39
3.2.1 如何获得开发资源 39
3.2.2 固件库开发与寄存器开发 39
3.2.3 固件库与CMSIS标准 40
3.2.4 官方固件库安装 41
3.2.5 官方固件库使用 52
3.3 ARM MDK开发工具的安装与使用 54
3.3.1 ARM MDK安装 54
3.3.2 使用ARM MDK进行Elink407软件开发 57
第一个工程(工程模版目录结构) 57
3.3.3 配置工程 59
3.3.4 下载与调试 68
3.4 IAR for ARM 工具软件的安装使用 71
3.4.1 IAR for ARM 安装 71
3.4.2 使用IAR for ARM进行Elink407软件开发 71
3.4.3 第一个工程 71
3.4.4 工程目录结构 72
3.4.5 配置工程 72
3.4.6 下载与调试 73

4 STM32F407 的存储器模型 75
4.1 存储器空间映射 75
4.1.1 STM32的SRAM 77
4.1.2 STM32的Flash 77
4.2 位带区 78
4.3 不同启动模式对应的内存映射 79
4.4 端格式 80

5 启动过程 81
5.1 各版本启动文件 81
5.1.1 MDK启动文件 81
5.1.2 IAR 启动文件 81
5.2 启动代码分析 82

6 时钟系统 87
6.1 STM32F407 时钟系统 87
6.2 RCC相关库函数说明（STM32F4 时钟初始化配置） 89
6.3 修改系统时钟的方法 95
6.4 RCC相关库函数说明（STM32F4 时钟使能和配置） 96

7 LED灯实验 99
7.1 实验原理与基础知识 99
7.2 电路设计 103
7.3 不使用标准库，驱动代码设计 105
7.4 位带区功能，驱动代码设计 112
7.5 使用固件库驱动代码设计 115
7.5.1 库函数和数据结构 115
7.5.2 软件设计、开发 116
7.5.3 下载测试 119

8 蜂鸣器实验 121
8.1 实验原理与基础知识 121
8.2 电路设计 121
8.3 库函数和数据结构 121
8.4 软件设计、开发 121
8.5 下载测试 124

9 按键实验 125
9.1 实验原理与基础知识 125
9.2 电路设计 125
9.3 库函数和数据结构 125
9.4 软件设计、开发 125
9.5 软件下载、测试验证 128
9.6 实验原理与基础知识 129
9.7 电路设计 129
9.8 库函数和数据结构 129
9.9 软件设计、开发 133

9.10 软件下载、测试验证 135

10 Systick实验 137
10.1 实验原理与基础知识 137
10.2 电路设计 139
10.3 库函数和数据结构 139
10.4 软件设计、开发 140
10.5 软件下载、测试验证 141

11 NVIC实验 143
11.1 实验原理与基础知识 143
11.2 电路设计 144
11.3 库函数和数据结构 144
11.4 软件设计、开发 145
11.5 软件下载、测试验证 145

12 UART 实验 147
12.1 实验原理与基础知识 147
12.2 电路设计 147
12.3 库函数和数据结构 147
12.4 软件设计、开发 149
12.5 软件下载、测试验证 150

13 DMA实验 151
13.1 实验原理与基础知识 151
13.2 电路设计 151
13.3 库函数和数据结构 154
13.4 软件设计、开发 155
13.5 软件下载、测试验证 157

14 独立看门狗实验 159
14.1 实验原理与基础知识 159
14.2 电路设计 160
14.3 库函数和数据结构 160
14.4 软件设计、开发 161
14.5 软件下载、测试验证 162

15 窗口看门狗实验 163
15.1 实验原理与基础知识 163
15.2 电路设计 164
15.3 库函数和数据结构 164
15.4 软件设计、开发 166
15.5 软件下载、测试验证 166

16 ADC实验 167
16.1 实验原理与基础知识 167
16.2 电路设计 167
16.3 库函数和数据结构 168

16.4 软件设计、开发 168
16.5 软件下载、测试验证 171

17 IIC实验 173
17.1 实验原理与基础知识 173
17.2 电路设计 174
17.3 库函数和数据结构 174
17.4 软件设计、开发 175
17.5 软件下载、测试验证 179

18 SPI实验 181
18.1 实验原理与基础知识 181
18.2 电路设计 181
18.3 库函数和数据结构 182
18.4 软件设计、开发 183
18.5 软件下载、测试验证 187

19 RTC实验 189
19.1 实验原理与基础知识 189
19.2 电路设计 189
19.3 库函数和数据结构 189
19.4 软件设计、开发 191
19.5 软件下载、测试验证 195

20 定时器实验 197
20.1 实验原理与基础知识 197
20.2 电路设计 197
20.3 库函数和数据结构 198
20.4 软件设计、开发 198
20.5 软件下载、测试验证 199

21 PWM实验 201
21.1 实验原理与基础知识 201
21.2 电路设计 201
21.3 库函数和数据结构 201
21.4 软件设计、开发 202
21.5 软件下载、测试验证 204

22 DAC实验 205
22.1 实验原理与基础知识 205
22.2 电路设计 206
22.3 库函数和数据结构 206
22.4 软件设计、开发 207
22.5 软件下载、测试验证 208

23 LCD 实验 209
23.1 实验原理与基础知识 209
23.2 电路设计 220
23.3 库函数和数据结构 220
23.4 软件设计、开发 223
23.5 软件下载、测试验证 230

24 LCD 显示汉字实验 231
24.1 实验原理与基础知识 231
24.2 电路设计 232
24.3 库函数和数据结构 232
24.4 软件设计、开发 233
24.5 软件下载、测试验证 235

25 显示图片实验 237
25.1 实验原理与基础知识 237
25.2 电路设计 237
25.3 库函数和数据结构 237
25.4 软件设计、开发 237
25.5 软件下载、测试验证 239

26 触摸屏实验 241
26.1 实验原理与基础知识 241
26.2 电路设计 241
26.3 库函数和数据结构 241
26.4 软件设计、开发 241
26.5 软件下载、测试验证 241

27 待机唤醒实验 243
27.1 实验原理与基础知识 243
27.2 电路设计 244
27.3 库函数和数据结构 244
27.4 软件设计、开发 245
27.5 软件下载、测试验证 247

28 随机数实验 249
28.1 实验原理与基础知识 249
28.2 电路设计 249
28.3 库函数和数据结构 249
28.4 软件设计、开发 251
28.5 软件下载、测试验证 252

29 CRC 实验 253
29.1 实验原理与基础知识 253
29.2 电路设计 254
29.3 库函数和数据结构 254
29.4 软件设计、开发 255
29.5 软件下载、测试验证 255

30 CPU UID实验 257
30.1 实验原理与基础知识 257
30.2 电路设计 257
30.3 库函数和数据结构 257
30.4 软件设计、开发 257
30.5 软件下载、测试验证 258

31 DHT11 实验 259
31.1 实验原理与基础知识 259
31.2 电路设计 260
31.3 库函数和数据结构 260
31.4 软件设计、开发 260
31.5 软件下载、测试验证 264

附录 A 相关资源 265
A.1 工具链接 265
A.2 视频资料 265
A.3 开源项目 265

附录 B 常见问题汇总 267
B.1 编译下载 267
B.2 接口编程 267

第 1 章

ELink407 硬件平台

本章节主要介绍Elink407开发板的硬件资源，使大家对该开发板的功能及特点有个详细的了解。本章将分为如下几个部分：

1，开发板简介；

2，开发板硬件详解；

3，开发板使用注意事项；

1.1 开发板简介

1.1.1 STM32407VG片上资源(芯片功能)

STM32407VGARM 是基于Cortex -M4内核，可在高至168MHz的频率下运行，带有FPU单元。

◆ 具有在系统编程（ISP）和在应用编程（IAP）功能的1M片上Flash程序存储器；

◆ 时钟、复位和供电管理,2.0至3.6伏供电和I/O管脚，上电/断电复位(POR /PDR)、可编程电压监测器(PVD)，内嵌4至16MHz高速晶体振荡器，内嵌经出厂调校的8MHz RC振荡器，内部40kHz的RC振荡器，PLL供应CPU时钟，带校准的32kHzRTC振荡器；

◆ 低功耗睡眠、停机和待机模式，VBAT为RTC和后备寄存器供电；

◆ 调试模式串行线调试(SWD)和JTAG调试接口；

◆ 液晶并行接口，8080 /6800模式；

◆ 3×12位，2.4 MSPS A/D转换器，24个转换通道；

◆ 2×12位D/A转换器；

◆ 通用DMA：16通道DMA控制器与FIFO和突发的支持；

◆ 17个定时器；有12个16位和2个32位定时器的即数频率达168兆赫，每组包含4个输入捕获/输出比较/PWM或脉冲计数器通道；

◆ 具有15个通信接口：

- 2个I2C接口（SMBus/PMBus）；

- 4个USARTs/2个UARTs(10兆位/秒)接口，支持ISO7816,LIN,IrDA接口和调制解调控制；
- 3个SPI（37.5兆位/秒）和2个混合全双工I2S通过内部音频锁相环或外部时钟实现音频级的精度，复位和时钟的供给管理；
- 2个CAN（2.0B主动）、1个SDIO接口；

◆ 高级连接

- USB 2.0全速器件/主机/OTG；
- USB的2.0 high-speed/full-speed设备/主机/OTG控制器的专用DMA，片上全速PHY和ULPI；
- 10/100以太网MAC专用DMA：支持IEEE1588v2的硬件，MII/RMII；

◆ 8 14位并行摄像机接口（54 Mbytes /s）

◆ 唯一的96位ID

1.1.2 板载资源

Elink407提供如下个特点：

◆ STM32F407VGT6 微控制器有 1M 的 FLASH 存储器，192K 的 RAM，LQFP100 封装

◆ 板上的 ST-LINK/V2 可以使用选择的方式把开发板切换成一个独立的 ST-LINK/V2 来使用（用 SWD 连接器来编程和调试）

◆ 板子电源：通过 USB 总线或者外部 5V 电源来供电

◆ 对外部应用供电：3V 和 5V

◆ EMW1062无线wifi模块，结合TCP/IP库接入网络，瞬间开发出物联网应用

◆ 3.2TF-LCD,触摸屏

◆ DHT11数字温湿度传感器

◆ SPI,EEPROM存储芯片

◆ micro SD卡座

◆ USB转232，通过usb线与PC通信

◆ 扩展接口

- arduino接口兼容arduino外设
- CAMERA接口
- CAN接口
- 485接口
- 以太网接口
- IIS音频接口
- 独有的血压计接口

- 其他功能接口

◆ 5 个 LED 指示灯

- LED1,LED2(红/绿)ST1ink下载调试通讯
- LD3（红）用于 5V 上电
- 2 个用户 LED(绿）灯，LED4,LED5

◆ 5 个按钮，休眠按键,复位健和3个用户按键

◆ USB OTG，micro-AB 连接器

◆ LQFP100 IO 口的扩展头，可以快速连接成原型板，易于探测

1.2 开发板硬件详解

1.2.1 MCU

“最小系统（包括复位，boot，电源,晶振)” E1ink407 开发板使用的是100脚LQFP封装的STM32F407VGT6芯片，我们将所有该芯片所以引脚引出，并提供丰富的外围扩展接口，充分满足广大爱好者的学习开发需求。E1ink407 MCU部分原理图如下图所示:

虽然STM32F407VGT6内部已经包含了16MHz高速内部RC振荡电路和32kHz左右低速内部RC振荡电路LSE，但是其精准度不是很高，为此我们在外部增加了8MHz的晶振电路，为系统的可靠工作提供时序基准，和可以用于驱动独立看门狗和通过程序选择驱动RTC实时时钟。VDDA与VREF是STM32处理器数字/模拟转换（ADC）电路需要使用到的模拟参考电源和模拟电源输入。 为保证其稳定性需要隔离并增加独立的滤波储能电容。

MCU
M_O_INT/PA0
REFCLK
MDIO
PWM_MOTOR
DCMI_HSYNC
SPI1_SCK
DCMI_PIXCLK
CRS_DV
LED1
UART1_TX
UART1_RX
USB_D-
USB_D+
M_JTMS
M_JTCK
SPI1_CS
ADC
DC_BLOODSIGNAL
BOOT1/LAN_RST
M_JTDO
SPI1_MISO
SPI1_MOSI
IIC1_SCL/DCMI_D5
IIC1_SDA
LED2
IIS_WS/SPI2_CS
IIS_CK/USART3_TX
TX-EN/UART3_RX
TXD0/CAN2_RX
TXD1/CAN2_TX/SPI2_SCK
IIS_SDOUT/SPI2_MISO
IIS_SDA/SPI2_MOSI
BEEP/ADC0
MDC/WIFI_IRQ
AC_BLOODSIGNAL
PWM_VALVE/DCMI_SDA
RXD0
RXD1/WIFI_WAKE
IIS_MCK
DCMI_D1/PWM
SD_DAT0
SD_DAT1/DCMI_D3
SD_DAT2
SD_DAT3
SD_CLK
PC13/DCMI_RST
32K
WIFI_RST
KEY3
KEY2
KEY1
PE5
DCMI_D7
LCD_D4
LCD_D5
LCD_D6
LCD_D7
LCD_D8
LCD_D9
LCD_D10
LCD_D11
LCD_D12
LCD_D2
LCD_D3
SD_CMD
LCD_RST1
LCD_RD1
LCD_WR1
PD6/KEY4
LCD_CS1
LCD_D13
LCD_D14
LCD_D15
LCD_RS1
LCD_INT1
LCD_EN1
LCD_D0
LCD_D1
+3.3V
R68
0R/NC
R69
0R
PDR_ON
PH0
PH1
PC14
PC15
NRST
BOOT0
C20
20pF
Y2
8MHZ
C9
20pF
C21
6.8pF
C22
6.8pF
Y1
32.768K
U4-A
STM32F407VGT6
+3.3V
3.3V
L7
80R
L3
80R
R72
47R
C10
1uF
C3
100nF
R73
0R
C12
1uF
C11
100nF
VDD5
VDD12
VDD4
VDD1
VDD2
VDD3
VREF+
VDDA
VBAT
VSS5
VSS4
VSS2
VSSA
VCAP1
VCAP2
U4-B
STM32F407VGT6
C61
1uF
C23
2.2uF
C24
2.2uF
C13
C18
C32
C34
C35
C36
C37
100nF
3.3V
R74
R75
4.7K
4.7K
IIC
IIC1_SCL/DCMI_D5
IIC1_SDA
X3
X2
X4
X5
X8
X9
HOLE-0

1.2.2 启动模式

上图中的BOOT0，BOOT1用于设置STM32的启动方式，其对应启动模式如下表所示：

BOOT1	BOOT0	启动模式	说明
X	0	主闪存存储器	内部FLASH被选为启动区域，一般使用此模式。
0	1	系统存储器	运行出厂自举固件升级程序，用于串口下载固件
1	1	内置SRAM	内置SRAM被选为启动区域，用于在SRAM中调试代码。

● 从主闪存存储器启动：主闪存存储器被映射到启动空间(0x0000 0000)，但仍然能够在它原有的地址(0x0800 0000)访问它，即闪存存储器的内容可以在两个地址区域访问，0x0000 0000或0x0800 0000。

● 从系统存储器启动：系统存储器被映射到启动空间(0x0000 0000)，但仍然能够在它原有的地址访问它，神舟王STM32F407VGT原有地址为0x1FFF0000 – x1FFF77FF。

● 从内置SRAM启动：只能在0x2000 0000开始的地址区访问SRAM。

● 通过软件修改寄存器可以将FSMC的地址段映射到启动空间(0x0000 0000)。

如果内部Flash已经烧录了程序，一般正常运行时选择从主闪存存储器启动。如果要升级固件可以有多种方式，常见的有通过工具烧录和外设接口烧录两种方式。常用的烧录调试仿真工具如JLINK、STLink、Ulink等。

1.2.3 复位电路

Elink407开发板使用RC加按键来实现手动复位和上电复位。STM32芯片内部已经集成有上电复位和电压检测复位电路，所以外部的复位主要是实现手动复位功能，其原理图如下。

1.2.4 电源管理

Elink407开发板支持的供电方式主要有三种，分别是：

● USB接口供电，最大500mA
● 外部直流DC5V供电
● ST_LINK 5V供电

如上图所示，当J20接入5V电源适配器，开发板上就有了5V的电源，再通过DC-DC芯片，转换3.3V电源供MCU芯片和其他芯片或模块使用。大家也可以在下面的ST-LINK和USB To RS232部分电路图中找到USB供电电路

1.2.5 ST-LINK

DC-DC
J20
PJ-102A
V5
L10
0 ohm/SB
C7
10uF
C8
10uF
C196
22uF
C14
22uF
C30
10pF
U1
VIN
SW
EN
FB
GND
NC
NC
NC
PAM2310
R58
100K
10uF
C53
L11
4.7uH
3.3V
D2
LED_SMD0603
R91
2.2K
C1
100nF
C55
100nF
R81
26.1K 1%
R92
10K 1%
C33
10uF
C31
10uF
C54
3.3nF
R113
5.6K

Debugger

1.2.6 USB To RS232

开发板有一个USB To RS232接口，用户的PC或笔记本通过USB线就可以跟开发板进行串口通信，当然了，首先你要安装USB转232驱动，安装详见USB2USRT章节，电路如下。

1.2.7 ADC & DHT11

开发板板载一个DHT11温度传感器，可以用于采集环境温度或设备温度，板载有电位器可以实现AD模数转换实现，电路如下：

1.2.8 USBOTG

STM32F407含有一个USB 2.0 OTG全速接口，接口速率12Mbps，它既可以作为OTG与其它OTG设备相连通过ID信号协商主从角色；也可以固定作为USB SLAVE从设备与PC等主设备相连实现USB读卡器，USB鼠标，USB固件升级等功能；还可以固定作为USB HOST连接U盘，USB键鼠等从设备。当作为USB从设备设备时通过D+、D-引脚上的1500Ω上拉电阻可向主机发出设备已连接的信号，并指示设备的工作速度。电阻上拉至D+表示全速运行，电阻上拉至D-表示低速运行。当作为USB主设备时内部D+和D-含有15kΩ下拉电阻用于检测上拉电阻的状态。

OTG就是On The Go，正在进行中的意思。USB OTG使USB装置摆脱了原来主从架构的限制，实现了端对端的传输模式。USB OTG标准在完全兼容USB2.0标准的基础上，增添了电源管理（节省功耗）功能，它允许设备既可作为主机，也可作为外设操作。

1.2.9 LCD & Touch

开发板载有3.2寸带触摸液晶显示屏，LCD 接口连接在 FSMC 总线上面。支持触摸，IIC_SDA,IIC_SCL用来实现对触摸屏的控制其原理图如下：

1.2.10 1062

板载1062 wifi模块，该模块通过wifi 热点加入网络，通过socket编程进行网络数据通信。

1.2.11 SDcard

板载有标准的SD卡接口，采用STM32的高速SDIO接口访问SD卡，有了这个接口，我们就可以外扩大容量存储设备，可以用来记录数据。其原理图如下：

SD卡可通过SPI接口或者SDIO接口来通信，其中SDIO模式通信速率更高，在这里我们即是使用的SDIO模式进行通信。卡座的9脚SD用于卡插入检测。卡座内部有一个机械开关。当卡插入后，该机械开关闭合，和SD卡座的外壳（铁）搭接。一般外壳需要接GND，SD引脚接上拉电阻，这样即可实现无卡时SD口线为高电平，有卡时，SD口线为低电平。

1.2.12 Key & LED

开发板底板上总共有5个LED指示灯，其中1个是3.3V电源指示灯，还有两个下载调试指示灯。其它2个是用户LED指示灯。用户LED指示灯由GPIO管脚控制LED灯的亮灭，当GPIO管脚输出低电平时，LED指示灯亮。反之，当GPIO管脚输出高电平时，LED指示灯灭。这两个LED指示灯分别由PA8、PB8控制。

开发板总共有6个功能按键，分别是WAKEUP按键和RESET按键及四个用于自定义功能用户按键。四个用户按键分别与PD6、PE2、PE3和PE4四个GPIO管脚连接，当按键按下时，对应的GPIO管脚为低电平，反之，当没有按键按下时，对应的GPIO管脚为高电平。WKUP按键PA0（STM32的WKUP引脚）可以作为WK_UP功能，它除了可以用作普通输入按键外，还可以用作STM32的唤醒输入。

LED
+3.3V
LED1
R1
100R
LED_SMD0603/GREEN
D1
LED2
R3
100R
LED_SMD0603/GREEN
D3

KEY
3.3V
R51
4.7K
R50
4.7K
R49
4.7K
R37
4.7K
SW36
PUSHBUTTON
KEY1
SW35
PUSHBUTTON
KEY2
SW27
PUSHBUTTON
KEY3
SW1
PUSHBUTTON
PD6/KEY4

1.2.13 IIC

开发板自带了24C02的EEPROM芯片，该芯片的容量为2Kbit，也就是256个字节，对于我们普通应用来说是足够了的。你也可以选择换大的芯片，因为在原理上是兼容24C02～24C512全系列的EEPROM芯片的。其原理图如下：

1.2.14 Connector

血压计

本开发板独有的血压计接口，通过与配套的血压计模块配套使用，进行血压计的案例开发

Ethernet

STM32F407内部集成有高性能以太网模块，支持通过以太网收发数据，符合IEEE 802.3 2002标准。该以太网模块灵活可调，使之能适应各种不同的客户需求。该模块支持两种标准接口，连接到外接的物理层(PHY)模块：IEEE 802.3协议定义的独立于介质的接口(MII)和简

化的独立于介质的接口(RMII)。适用于各类应用，如交换机、网络接口卡等。开发板将以太网的管脚引出，通过与配套的模块进行使用。

Camera

开发板设置了 2x9 排座作为外置摄像头接口，该接口为 8bit 并行数据接口， 除了采集到的图像数据还有 VSYNC， HSYNC 以及像素时钟 PIXCLK 输出到 stm32f407。另外接口的 3,4 引脚是摄像头控制接口（SCCB） 。通过与之配套的摄像头模块连接使用

SPI

开发板载有SPI FLASH芯片W25X16，该芯片的容量为2M字节(16MBit)，和引出的标准2.54间距的spi 接口，其原理图如下：

IIS

开发板预留了IIS的音频接口，通过与配套的音频模块连接使用

UART

底板载引出了2个uart，uart1和uart3，其中uart1经过usb转串口芯片，用户之需要安装串口驱动，通过usb线与PC相连，便可以使用。同时uart1 和uart3 扩展出了标准2.54的排针，方便用户扩展使用 ,uart1和uart3的原理图如下图所示：

CAN

底板板载了 CAN 总线接口，支持 CAN 协议 2.0A 和 2.0B，板上兼容 5V电压的 CAN 物理层接口芯片和 3.3V 电压 CAN 物理层接口芯片来实现 CAN。

arduino

ELink407开发板引出了标准的arduino接口，可扩展使用arduino 的外设。

RS232&RS485
V5
3.3V
L13
80R
L9
NC
CON4
J9
UART1_TX
UART1_RX
USART1
V5
3.3V
L15
80R
L14
NC
CON4
J12
IIS_CK/USART3_TX
TX-EN/UART3_RX
USART3
3.3V
R35
R36
R24
R29
4.7K
UART1_TX
UART1_RX
IIS_CK/USART3_TX
TX-EN/UART3_RX

CAN
DM
USB_D-
CAN1_RX
DP
USB_D+
CAN1_TX
J11
CON3X2
V5
CON4
J15
CAN1_TX
CAN1_RX
V5
CON4
J17
TXD1/CAN2_TX/SPI2_SCK
TXD0/CAN2_RX

扩展GPIO接口

ELink407开发板将部分IO管脚使用标准双排插针引出，方便大家的实验和测试，调试其他模块或功能扩展

1.3 开发板使用注意事项

1.3.1 基本注意事项

为了让大家更好的学习使用ELink407，我们在这里使用的时候需要注意的一些问题。

1， 开发板一般情况是由 USB_DEBUG口(J16)供电，在使用电源适配器供电前，先检查适配器的输出是5V。PC的1 个 USB 供电最多 500mA，且由于导线电阻存在，供到开发板的电压，一般都不会有5V，如果使用了很多大负载外设，有可能引起 USB 供电不够，建用适配器供电。

2， 使用1062模块进行wifi通信时，请接上天线，如果没有天线，传输距离很短。

3， 当你想使用某个 IO 口用作其他用处的时候，请先看看开发板的原理图，该 IO 口是否有连接在开发板的某个外设上，如果有，该外设的这个信号是否会对你的使用造成干扰，先确定无干扰，再使用这个 IO。

4， 开发板上的跳线帽比较多，大家在使用某个功能的时候，要先查查这个是否需要设置跳线帽，以免浪费时间。具体可参见下面的跳线对照表

5， 开发板的 USB SLAVE 和 USB HOST 共用同一个 USB 口，所以，他们不可以同时使用。

1.3.2 跳线对照表

ELink407的跳线排针均为2.54间距的排针,通过跳线帽短接即可。需要说明的是带三角符号丝印的脚为1脚，或者看背面的焊盘形状也可以，1脚的焊盘为正方形，其它脚为圆形。

跳线标号	说明			
J35 ， J36	J36 --> BOOT0 J35 --> BOOT1	短接方式	BOOT 值	启动方式
		J36 Pin2,3 短接 J35 任意	BOOT0=0 BOOT1=X	主 Flash（STM32 内部 flash）启动
		J36 Pin1,2 短接 J35 Pin2,3 短接	BOOT0=1 BOOT1=0	系统存储器作为自举空间，串口下载
		J36 Pin1,2 短接 J35 Pin1,2 短接	BOOT0=1 BOOT1=1	从外部 SRAM 启动
J7	短接后由 J16 的 miniUSB 供电			
J5 J9	短接 J9 pin1,2 和 pin3,4 短接 J5 通过 JLink 将 STLink 固件烧录到 U6 中 短接 J9 pin1,2 和 pin3,4 取消短接 J5 通过 STLink 调试下载 stm32f407 代码			
J27	短接 J27 pin1,2 进行 ADC 实验，短接 J27 pin3,2 进行 DHT11 温湿度实验			
J21	短接 J21pin1,2 SPI flash 和 SPI 接口片选有效，短接 J21pin3,2 wifi 模块片选有效			
J11	短接 J11 pin2,4 和 Pin1,3 GPIO 引脚作为 USB 功能，短接 J11 pin6,4 和 Pin5,3 GPIO 引脚作为 CAN1 功能			
J24	GPIO 引脚功能复用选择作为以太网 RXD1 或 wifi 模块 wakeup			
J33	GPIO 引脚功能复用选择作为以太网 MDC 或 wifi 模块中断			
J34	GPIO 引脚功能复用选择作为蜂鸣器控制 IO 或其他功能			
J30	短接 J30pin1,2 对 arduino 接口进行 5V 供电			

图 1.1: 跳线对照表

第 2 章

Elink407开发板的使用

2.1 串口工具的安装使用

2.1.1 常用串口工具软件

串口工具，也就是串行通信接口调试软件，不仅广泛应用于嵌入式控制领域的数据监控、数据采集、数据分析等工作,而且在嵌入式开发的过程中打印调试信息，和运行的结果，是嵌入式开发的必备工具之一。我们例程的运行结果大多是通过串口打印出来的，在这里特别注明的是通过Elink407的USART0，进行数据通信，因此你需要通过USB串口线连接到PC机，先在PC上安装USB转串口的驱动，然后在PC机运行一个终端仿真软件（或者叫串口工具）查看结果。我们常用的串口调试工具包括串口调试助手、SecureCRT、超级终端、串口调试大师、putty等， 在这里推荐使用SecureCRT作为串口工具,它比其他的一些d串口工具类的软件功能丰富，稳定好用。

2.1.2 USB2UART驱动的安装

嵌入式领域使用的最广泛的应该莫过于RS232串行通信接口，我们的Elink407跟PC通信方式也是RS232接口。传统的主板台式机都有这个接口，笔记本就很少再带有这些老式接口，那么有什么方法使得我们笔记本的USB跟开发板的RS232串口进行通讯呢？针对这种情况我们在开发板上加了一个USB转串口的芯片，然后在PC上安装该芯片的驱动，通过USB转串口的方式来解决这个问题。 下面是我们驱动程序所在的路径：

```
Elink407\tools\usb2uart driver\PL-2303 Vista Driver Installer（Win7、8）.zip

Elink407\tools\usb2uart driver\PL2303_Prolific_DriverInstaller_v1_9_0（WinXP）.zip
```

他们分别是WIN7,8 和XP的驱动，Linux平台上默认安装了该驱动。驱动的安装过程，只需要解压压缩包，双击安装的exe文件即可。

2.1.3 串口工具的安装使用

SecureCRT是一款支持SSH和串口的终端仿真程序，功能非常强大，我们只使用其中的串口部分即可。安装文件位置：

```
光盘\Elink407\tools
```

压缩的文件夹工具
PL-2303 Vista Driver Installer (Win7、8) .zip
文件
主页
共享
查看
解压缩
详细信息窗格
导航窗格
预览窗格
窗格
超大图标
大图标
中图标
小图标
列表
详细信息
布局
排序方式
当前视图
项目复选框
文件扩展名
隐藏的项目
隐藏所选项目
显示/隐藏
选项
usb... ▸ PL-2303 Vista Driver Installer (Win7、8) .zip
搜索"PL-2303 Vista Driver I...
收藏夹
下载
桌面
最近访问的位置
这台电脑
Desktop
名称
修改日期
类型
大小
PL-2303 Vista Driver Installer.exe
2007/8/9 17:44
应用程序
2,018 KB
1 个项目

tarena - VI 未连接 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
连接
Sessions
192.168.0.108
192.168.0.108 (1)
Serial-COM3
192.168.0.108 (2)
192.168.170.225
Serial-COM3 (1)
192.168.1.110
192.168.1.159
启动时显示(S)
在标签页中打开(O)
连接(C)
关闭

解压 SecureCRT.rar文件，启动SecureCRT软件，使用时会弹出 快速连接 窗口，如下图：

设置串口属性，在标签下点右键弹出回话选项，如图所示：

设置串口属性，如图所示

这里用户具体使用哪个端口，需要根据设备管理器里面当前存在的端口号进行设置，如果没有发现端口，查看usb转串口的驱动是否安装成功。

会话选项 - Serial-COM3
类别(C):
连接
登陆动作
串行
SSH2
高级
终端
仿真
模式
Emacs
映射键
高级
外观
窗口
日志文件
打印
X/Y/Zmodem
文件传输
FTP/SFTP
串行选项
端口(O):
COM6
波特率(B):
115200
数据位(D):
8
奇偶校验(A):
None
停止位(S):
1
流控
DTR/DSR
RTS/CTS
XON/XOFF
串行中断长度(S):
100
毫秒
确定
取消

2.2 二进制程序烧写方法

程序代码写完以后，通过编译器编译生成可以在该平台可以运行的指令数据，这些数据通常是HEX文件的方式，那么如何将这个HEX文件下载（烧录）到嵌入式平台中的flash中呢？下面我们就来介绍一下Elink407的烧写方法。STM32F4 的程序下载有多种方法：USB、串口、JTAG、SWD 等，这几种方式，都可以用来给 STM32F4 下载代码。

2.2.1 使用 STM32 STLINK Utility 工具烧写

ST-LINK/V2是STM8和STM32微控制器系列的在线调试器和编程器。单线接口模块（SWIM）和串行线调试（SWD）接口用于与应用板上的STM8和STM32微控制器通讯。

STM8的应用使用USB全速接口与STVisualDevelop(STVD)，STVisualProgram(STVP)或IAREWSTM8等集成开发环境通讯。 STM32的应用使用USB全速接口与Atollic，IAR，Keil或TASKING等集成开发环境通讯。

烧录工具安装

在目录Elink407\tools\stlink2 下可以找到安装包STM32 ST-LINK Utility_v3.3.0.exe应用程序，双击安装，按照提示依次选择“Next”->“Yes”->“Next”，在弹出的窗口点击“下一步”->“完成”，最后点击finish完成安装。安装完成后将STLINKV2插入电脑的USB接口，此时计算机会提示发现新硬件，并提示安装驱动，请选择自动安装。此时，ST-LINKV2的D2LED会不停的闪烁，当D2长亮时，说明ST-LINKV2可以使用了。此时，打开计算机的设备管理器会发现在“通用串行总线控制器”选项里面，多了一个器件，这就是ST-LINKV2。

ST-LINKV2固件的升级 ST-LINKV2可以在线升级固件，目前的最新固件为V2.J17.S4。更新固件的方法为：打开STM32ST-LINKUtility软件(安装光盘里面带的2个压缩文件解压后的软件)，将ST-LINKV2插入计算机，在菜单中选择“ST-LINK”菜单，选中第一个子菜单“Firmwareupdate”

点击DeviceConnect按钮，此时对话框界面会提示当前固件版本及最新的固件版本，点击Yes按钮，固件就会自动升级

驱动安装 调试器这里我们采用ST-LINK，接口为USB，位于开发板屏幕下侧。驱动程序文件位于keil安装目录，如果采用默认安装，将位于Elink407\tools\stlink2\drivers\stsw-link009.zip压缩文件进行解压，进入在解压的目录，该目录双击脚本stlink_winusb_install.bat自动完成驱动安装。

烧写 将配套USB线连接在一端连接再PC另一端连接到开发板屏幕下侧的USB口，该USB接口专门用于烧写程序和调试代码使用。连接成功后红色电源灯会亮，如果电源不亮，请检查开发板J7跳线是否已经连接。 打开ST-LINK，在标签栏选择“Target”->“Connect”。如连接成功，ST-LINK会读取到硬件信息，如图所示：

在ST-LINK界面选择 Binary File，在弹出的对话窗口选择打开Elink407_test.hex文件，如图 3所示：

最后选择“Target”->“program&verify…”烧写到开发板并运行测试

2.2.2 使用 STM32 and STM8 Flash loader demonstrator 工具烧写

在Elink407\tools\目录中找到Flash loader demonstrator工具，双击运行安装；Flash loader demonstrator工具是通过串口下载烧录文件，因此，确保已经安装了usb转232的驱动，要将USB一端插入Elink407，一端插入电脑上，并在“设备管理器”中查看到虚拟串口的串口号。启动烧录工具，配置串口号，其余参数使用默认值即可。在点击next前，一定要将boot0引脚置高并复位。如果一切正常，软件会转向下一个页面。当然如果出错，则需要重复上述操作。

如果芯片的flash未被读、写保护，那么“Remove protection”按键是灰色的；如果芯片的flash被保护了，那么，这个按钮可以取消保护，对flash进行写入操作。

这个页面基本上是只读的，可以看到芯片的flash的基本信息显示

对flash的烧写与擦除就集中在这个页面里。在这个页面里我们可以擦除（erase）芯片现有的flash内容。我们可以下载（download）hex程序到芯片flash内，我们也可以使能/去除芯片的保护（disable/enable）功能，还是配置选项字的高级功能。

这里简单说明一下，擦除功能可以选择全部擦除（All），也可以选择擦除部分扇区（selection），点击后即弹出相应区块的页面。

下载flash（download），软件支持英特尔hex，binary与S19摩托罗拉格式。其余部分我们初级使用默认即可。

上传芯片内部flash内容（upload），该功能可以将芯片内部的flash读出来，并转存为相应的格式，其格式支持上面提到三种。

选项字配置，这个配置我们暂时用不到，留到后期高级应用时，我们再回过头来仔细研究其强大的作用与深远的意义。

如果进度条变成绿色的并提示下载成功，那么，恭喜你！程序已经成功下载至芯片内部了。这时，需要将boot0置低，复位微处理器，然后微处理器就执行新的程序了！

2.2.3 使用 FlyMcu 程序烧写

打开FlyMcu,切换到STMISP tab页面 ，然后选中USB转串口COM PORT

选择要烧录的HEX文件

设置烧写方式，一定要选 DTR低电平复位，RTS高电平进BootLoader

Flash Loader Demonstrator
STMicroelectronics
Select the communication port and set settings, then click next to open connection.
Common for all families
UART
Port Name
COM1
Parity
Even
Baud Rate
115200
Echo
Disabled
Data Bits
8
Timeout(s)
5
Back
Next
Cancel
Close

Flash Loader Demonstrator
STMicroelectronics
Target is readable. Please click "Next" to proceed.
Remove protection
Back
Next
Cancel
Close

Flash Loader Demonstrator
STMicroelectronics
Please, select your device in the target list
Target
STM32_Connectivity-line_256K
STM32_Connectivity-line_128K
STM32_Connectivity-line_256K
STM32_Connectivity-line_64K
PID (h)
Version
1.0
Flash mapping
Name
Start address
End address
Size
R
W
Page0 0x 8000000 0x 80007FF 0x800 (2K)
Page1 0x 8000800 0x 8000FFF 0x800 (2K)
Page2 0x 8001000 0x 80017FF 0x800 (2K)
Page3 0x 8001800 0x 8001FFF 0x800 (2K)
Page4 0x 8002000 0x 80027FF 0x800 (2K)
Page5 0x 8002800 0x 8002FFF 0x800 (2K)
Page6 0x 8003000 0x 80037FF 0x800 (2K)
Page7 0x 8003800 0x 8003FFF 0x800 (2K)
Page8 0x 8004000 0x 80047FF 0x800 (2K)
Page9 0x 8004800 0x 8004FFF 0x800 (2K)
Page10 0x 8005000 0x 80057FF 0x800 (2K)
Page11 0x 8005800 0x 8005FFF 0x800 (2K)
Legend :
Protected
UnProtected
Back
Next
Cancel
Close

Flash Loader Demonstrator
STMicroelectronics
Erase
All
Selection
Download to device
Download from file
Erase necessary pages
No Erase
Global Erase
@ (h)
8000000
Jump to the user program
Optimize (Remove some FFs)
Verify after download
Apply option bytes
Upload from device
Upload to file
Enable/Disable Flash protection
ENABLE
WRITE PROTECTION
Edit option bytes
Back
Next
Cancel
Close

Flash Loader Demonstrator
STMicroelectronics
Target STM32_Connectivity-line_256K
Map file STM32_Connectivity-line_256K.STmap
Operation DOWNLOAD
File name C:\MyFile.s19
File size 256.00 KB (262144 bytes)
Status 209.88 KB (214922 bytes) of 256.00 KB (262144 bytes)
Time 00:00:31
Downloading data 82%
Back
Next
Cancel
Close

FlyMcu V0.188--单片机在线编程专家--www.mcuisp.com
系统(X) 帮助(Y) Language 搜索串口(V) Port:COM5 bps:115200 www.mcuisp.com 编程器(W) 关于(Z)...
联机下载时的程序文件:
E:\orange\elink407\Examples\project\(1)GPIO_LED
COM5 : 空闲 Prolific USB-to-Serial Comm Port
手持万用编程器 STMISP 免费STMIAP NXP ISP EP968_RS232
开始编程(P)
校验
编程后执行
使用RamIsp
连续烧录模式
读器件信息(R)
清除芯片(Z)
读FLASH
选项字节区:
编程到FLASH时写选项字节
设定选项字节等
DTR的低电平复位,RTS高电平进BootLoader

FlyMcu V0.188--单片机在线编程专家--www.mcuisp.com
系统(X) 帮助(Y) Language 搜索串口(V) Port:COM5 bps:115200 www.mcuisp.com 编程器(W) 关于(Z)...
E:\orange\elink407\Examples\project\(1)GPIO_LED\out\gpio_led.hex
编程前重装文件
手持万用编程器 STMISP 免费STMIAP NXP ISP EP968_RS232
开始编程(P)
校验
编程后执行
使用RamIsp
连续烧录模式
读器件信息(R)
清除芯片(Z)
读FLASH
选项字节区:
编程到FLASH时写选项字节
设定选项字节等
DTR的低电平复位,RTS高电平进BootLoader

Flash Loader Demonstrator
STMicroelectronics
Target STM32_Connectivity-line_256K
Map file STM32_Connectivity-line_256K.STmap
Operation DOWNLOAD
File name C:\MyFile.s19
File size 256.00 KB (262144 bytes)
Status 209.88 KB (214922 bytes) of 256.00 KB (262144 bytes)
Time 00:00:31
Downloading data 82%
Back
Next
Cancel
Close

2.3 开发板测试程序烧写与使用

使用 STM32 STLINK Utility 工具烧写

第 3 章

软件开发环境

3.1 STM32软件开发基本知识

3.1.1 STM32编程基础

在学习前，用户必须掌握c语言语法、变量空间、程序空间、堆栈、c编译器和调试器工作原理等相关知识，也必须了解CPU内部Flash和内部RAM的特性和地址空间。为了方便广大初学者学习，我们对这些概念进行一些必要的说明。

首先，我们必须明确设计程序的最终目的是生成一个可用于烧写的文件（常见的就是hex格式，本文档的附录章节有hex文件格式说明），这个文件可以通过烧写软件烧写到CPU内部Flash。在CPU上电后，CPU的硬件取指系统会自动的读取并执行Flash中的指令序列。

“源文件“是指由汇编源文件(.s)和c源文件（.c、.h）构成的文件集合。我们通过编辑器（比如UltraEdit、SourceInsight或者IDE开发环境自带的编辑器）来书写源文件。源文件经过编译器（.s由汇编编译器负责编译，.c、.h由c编译器负责编译）编译后生成中间目标文件(.obj)。每个源文件会对应一个同名的obj文件，这些obj文件中的程序地址以及变量地址是未定位的。之后，由连接器负责将这些.obj文件链接为一个完整的目标文件（如axf文件），这个axf文件包含CPU可执行的二进制代码和一些必要的调试信息。生成axf文件后，我们就可以通过集成开发环境（IDE）提供的调试接口下载程序到Flash，并通过仿真器进行跟踪调试。为了便于产品批量烧写程序，我们可以借助于IDE提供的工具，将axf文件转换为可直接烧写的文件（如hex文件）。

“程序空间“指二进制可执行指令序列（包括一些常量）存放的空间，”变量空间“指程序运行时各种变量（局部的和全局的）以及堆栈的存放空间。程序空间可以定位在CPU内部Flash、CPU内部RAM、外部SRAM和外部NOR Flash。变量空间只能定位在CPU内部SRAM和外部SRAM。

在使用STM32开发产品时，程序空间必须定位在CPU内部Flash，因为上电后，CPU都是从内部Flash开始取值运行的。在调试开发阶段，为了节省程序装载时间和延长Flash寿命，可以将代码空间定位到CPU内部RAM或者外部SRAM。一个工程包含多个target的意义就在于此，可以方便地选择调试模式和最终成品模式两种连接配置。

在c程序中，栈（stack）用来保存函数返回地址、局部变量空间分配。堆（heap）用于通过malloc之类函数给变量分配空间。栈和堆的大小需要根据用户的具体程序进行合理设置，过小的栈和堆会引起程序执行异常。

3.1.2 C语言

C 语言知识博大精深，也不是我们三言两语能讲解清楚，同时我们相信学 STM32F4 这种级别 MCU 的用户，C 语言基础应该都是没问题的。我们这里主要是简单的复习一下几个 C 语言基础知识点，引导那些 C 语言基础知识不是很扎实的用户能够快速开发 STM32 程序。同时希望这些用户能够多去复习一下 C 语言基础知识，C 语言毕竟是单片机开发中的必备基础知识。对于 C 语言基础比较扎实的用户，这部分知识可以忽略不看。

位操作

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

C 语言支持如下 6 种位操作

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

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

1) 不改变其他位的值的状况下，对某几个位进行设值

这个场景单片机开发中经常使用，方法就是先对需要设置的位用&操作符进行清零操作，然后用|操作符设值。比如我要改变 GPIOA-> BSRRL 的状态,可以先对寄存器的值进行&清零操作

```
GPIOA-> BSRRL &=0XFF0F;  //将第 4-7 位清 0然后再与需要设置的值进行|或运算
GPIOA-> BSRRL |=0X0040;  //设置相应位的值，不改变其他位的值
```

2) 移位操作提高代码的可读性。

移位操作在单片机开发中也非常重要，我们来看看下面一行代码

```
GPIOx->ODR = (((uint32_t)0x01) << pinpos);
```

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

```
GPIOx->ODR =0x0030;
```

这样的代码就不好看也不好重用了。

3) ~取反操作使用技巧

SR 寄存器的每一位都代表一个状态，某个时刻我们希望去设置某一位的值为 0，同时其他位都保留为 1，简单的作法是直接给寄存器设置一个值:

```
TIMx->SR=0xFFF7 ;
```

这样的作法设置第 3 位为 0，但是这样的作法同样不好看，并且可读性很差。看看库函数代码中怎样使用的:

```
TIMx->SR = (uint16_t)~TIM_FLAG;
```

而 TIM_FLAG 是通过宏定义定义的值：

```
#define TIM_FLAG_Update                    ((uint16_t)0x0001)
#define TIM_FLAG_CC1                       ((uint16_t)0x0002)
```

看这个应该很容易明白，可以直接从宏定义中看出 TIM_FLAG_Update 就是设置的第 0 位了，可读性非常强。

struct

经常很多用户提到，他们对结构体使用不是很熟悉，但是 MDK 中太多地方使用结构体以及结构体指针，这让他们一下子摸不着头脑，学习 STM32 的积极性大大降低，其实结构体并不是那么复杂，这里我们稍微提一下结构体的一些知识，还有一些知识我们会在下一节的“寄存器地址名称映射分析”中讲到一些。声明结构体类型：

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

例如：

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

在结构体申明的时候可以定义变量，也可以申明之后定义，方法是：

```
Struct  结构体名字  结构体变量列表  ;
```

例如：struct U_TYPE usart1,usart2;
结构体成员变量的引用方法是：

```
结构体变量名字.成员名
```

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

例如：struct U_TYPE *usart3; //定义结构体指针变量 usart1;

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

```
usart3->BaudRate;
```

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

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

```
void USART_Init(u8 usartx,u32 u32 BaudRate,u8 parity,u8 mode);
```

这种方式是有效的同时在一定场合是可取的。但是试想，如果有一天，我们希望往这个函数里面再传入一个参数，那么势必我们需要修改这个函数的定义，重新加入字长这个入口参数。于是我们的定义被修改为：

```
void USART_Init (u8 usartx,u32 BaudRate, u8 parity,u8 mode,u8 wordlength );
```

但是如果我们这个函数的入口参数是随着开发不断的增多，那么是不是我们就要不断的修改函数的定义呢？这是不是给我们开发带来很多的麻烦？那又怎样解决这种情况呢？

这样如果我们使用到结构体就能解决这个问题了。我们可以在不改变入口参数的情况下，只需要改变结构体的成员变量，就可以达到上面改变入口参数的目的。结构体就是将多个变量组合为一个有机的整体。上面的函数，BaudRate,wordlength,Parity,mode,wordlength 这些参数，他们对于串口而言，是一个有机整体，都是来设置串口参数的，所以我们可以将他们通过定义一个结构体来组合在一个。MDK 中是这样定义的：

```
typedef struct
{
    uint32_t USART_BaudRate;
    uint16_t USART_WordLength;
    uint16_t USART_StopBits;
    uint16_t USART_Parity;
    uint16_t USART_Mode;
    uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

于是，我们在初始化串口的时候入口参数就可以是 USART_InitTypeDef 类型的变量或者指针变量了，MDK 中是这样做的：

```
void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);
```

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

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

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

define

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

```
#define  标识符  字符串
```

“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。例如：

```
#define PLL_M       8
```

定义标识符 PLL_M 的值为 8。至于 define 宏定义的其他一些知识，比如宏定义带参数这里我们就不多讲解。

ifdef 条件编译

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

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

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

```
#ifdef
    程序段 1
#endif
```

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

```
#if defined (STM32F40_41xxx)
    STM32F40x 系列和 STM32F41x 系列芯片需要的一些变量定义
#end
```

而(STM32F40_41xxx 则是我们通过#define 来定义的。条件编译也是 c 语言的基础知识。

typedef

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

```
struct _GPIO
{
    __IO uint32_t MODER;
    __IO uint32_t OTYPER;
};
```

定义了一个结构体 GPIO，这样我们定义变量的方式为：

```
struct    _GPIO    GPIOA;//定义结构体变量 GPIOA
```

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

```
typedef struct
{
    __IO uint32_t MODER;
    __IO uint32_t OTYPER;
    …
} GPIO_TypeDef;
```

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

```
GPIO_TypeDef _GPIOA,_GPIOB;
```

这里的 GPIO_TypeDef 就跟 struct _GPIO 是等同的作用了。 这样是不是方便很多？

extern

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

```
extern u16 USART_RX_STA;
```

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

```
u16 USART_RX_STA;
```

下面通过一个例子说明一下使用方法。
在 Main.c 定义的全局变量 id，id 的初始化都是在 Main.c 里面进行的。

```
**Main.c 文件**

u8 id;//定义只允许一次
main()
{
    id=1;
    printf("d%",id);//id=1
    test();
    printf("d%",id);//id=2
}
```

但是我们希望在 main.c 的 changeId(void)函数中使用变量 id,这个时候我们就需要在 main.c里面去申明变量 id 是外部定义的了，因为如果不申明，变量 id 的作用域是到不了 main.c 文件中。看下面 main.c 中的代码：

```
extern u8 id;//申明变量 id 是在外部定义的，申明可以在很多个文件中进行
void test(void){
    id=2;
}
```

在 main.c 中申明变量 id 在外部定义，然后在 main.c 中就可以使用变量 id 了。对于 extern 申明函数在外部定义的应用，这里我们就不多讲解了。

static

static 修饰用来限定符号的作用域，分为修饰变量和修饰函数之分。

static 修饰函数其作用域限于只在当前源文件中调用该函数，另外static 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。

static修饰变量可分为修饰全局变量和局部变量之分，static 局部变量只被初始化一次，下一次依据上一次结果值；tatic 全局变量只初使化一次，跟修饰函数一样，其作用域限于当前源文件，可以防止在其他文件单元中被引用，修改。

volatile

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是， 优化器在用到这个变量时必须每次都小心地重新读取这个变量的值， 而不是使用保存在寄存器里的备份。 下面是 volatile 变量的几个例子：

1)． 并行设备的硬件寄存器（如：状态寄存器）

2)． 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3)． 多线程应用中被几个任务共享的变量

3.2 ST 固件库

ST(意法半导体)为了方便用户开发程序，提供了一套丰富的 STM32F4 固件库。到底什么是固件库？它与直接操作寄存器开发有什么区别和联系？很多初学用户很是费解，这一节，我们将讲解 STM32 固件库相关的基础知识，希望能够让大家对 STM32F4 固件库有一个初步的了解

所谓的固件库就是将一组功能相关的特殊功能寄存器的操作过程封装为一些C语言函数，以方便应用程序调用，方便使用者记忆。其实固件库中的每一个函数，如果单步跟踪进去，就是原生的寄存器操作代码。也许你看不懂这种寄存器操作形式，不过没关系，我们后面将举个例子来介绍如何阅读这些代码。我在这个地方需要表达的是，直接操作寄存器和调用固件库中函数本质上没有区别，不会因为就多封装了1层就改变了性质。

3.2.1 如何获得开发资源

固件库包光盘路径（是压缩包形式，大家解压即可） ：

STM32_Lib\stm32f4_dsp_stdperiph_lib.zip

同时，大家也可以到ST官网 http://www.st.com 下载[1]

3.2.2 固件库开发与寄存器开发

我们所有的例程都是用ST官方固件库进行开发的，全部是C语言。 很多从51过来的初学者很排斥用固件库，认为固件库太复杂，看不懂，找个寄存器定义都找不到；用库学不到东西。如果你持有这种观点，只能说明你c语言没有过关。还停留在一个 main.c 文件就是全部程序的最初级阶段。

在 51 的开发中我们常常的作法是直接操作寄存器，比如要控制某些 IO 口的状态，我们直接操作寄存器：

[1] http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1939/PF257901

```
P0=0x11;
```

而在 STM32 的开发中，我们同样可以操作寄存器：

```
GPIOF->BSRRL=0x0001; //这里是针对 STM32F4 系列
```

这种方法当然可以，但是这种方法的劣势是你需要去掌握每个寄存器的用法，你才能正确使用STM32，而对于 STM32 这种级别的 MCU，数百个寄存器记起来又是谈何容易。于是ST(意法半导体)推出了官方固件库，固件库将这些寄存器底层操作都封装起来，提供一整套接口（API）供开发者调用，大多数场合下，你不需要去知道操作的是哪个寄存器，你只需要知道调用哪些函数即可。

比如上面的控制 BSRRL 寄存器实现电平控制，官方库封装了一个函数：

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    GPIOx->BSRRL = GPIO_Pin;
}
```

这个时候你不需要再直接去操作 BSRRL 寄存器了，你只需要知道怎么使用 GPIO_SetBits()这个函数就可以了。在你对外设的工作原理有一定的了解之后，你再去看固件库函数，基本上函数名字能告诉你这个函数的功能是什么，该怎么使用，这样是不是开发会方便很多？

任何处理器，不管它有多么的高级，归根结底都是要对处理器的寄存器进行操作。但是固件库不是万能的，您如果想要把 STM32 学透，光读 STM32 固件库是远远不够的。你还是要了解一下 STM32 的原理，了解 STM32 各个外设的运行机制。只有了解了这些原理，你在进行固件库开发过程中才可能得心应手游刃有余。 只有了解了原理，你才能做到“知其然知其所以然”，所以大家在学习库函数的同时，要了解一下寄存器大致配置过程。

3.2.3 固件库与CMSIS标准

STM32F4 固件库就是函数的集合，那么对这些函数有什么要求呢？？这里就涉及到一个 CMSIS 标准的基础知识。 经常有人问到 STM32 和 ARM 以及 ARM7 是什么关系这样的问题，其实 ARM 是一个做芯片标准的公司，它负责的是芯片内核的架构设计，而 TI，ST 这样的公司，他们并不做标准，他们是芯片公司，他们是根据 ARM 公司提供的芯片内核标准设计自己的芯片。所以，任何一个做 Cortex-M4 芯片，他们的内核结构都是一样的，不同的是他们的存储器容量，片上外设，IO 以及其他模块的区别。所以你会发现，不同公司设计的 Cortex-M4 芯片他们的端口数量，串口数量，控制方法这些都是有区别的，这些资源他们可以根据自己的需求理念来设计。同一家公司设计的多种 Cortex-M4 内核芯片的片上外设也会有很大的区别，比如 STM32F407 和 STM32F429，他们的片上外设就有很大的区别。

既然大家都使用的是 Cortex-M4 核，也就是说，本质上大家都是一样的，这样 ARM 公司为了能让不同的芯片公司生产的 Cortex-M4 芯片能在软件上基本兼容，和芯片生产商共同提出了一套标准 CMSIS 标准(Cortex Microcontroller Software Interface Standard)，翻译过来是“ARM Cortex 微控制器软件接口标准”。 ST 官方库就是根据这套标准设计的。这里我们又要引用参考资料里面的图片来看看基于 CMSIS 应用程序基本结构如下图

图 3 基于 CMSIS 应用程序基本结构

CMSIS 分为 3 个基本功能层：

1）核内外设访问层：ARM 公司提供的访问，定义处理器内部寄存器地址以及功能函数。

2）中间件访问层:定义访问中间件的通用 API。由 ARM 提供，芯片厂商根据需要更新。

3）外设访问层：定义硬件寄存器的地址以及外设的访问函数。

从图中可以看出， CMSIS 层在整个系统中是处于中间层，向下负责与内核和各个外设直接打交道，向上提供实时操作系统用户程序调用的函数接口。如果没有 CMSIS 标准，那么各个芯片公司就会设计自己喜欢的风格的库函数，而 CMSIS 标准就是要强制规定，芯片生产公司设计的库函数必须按照 CMSIS 这套规范来设计。 其实不用这么讲这么复杂的，一个简单的例子，我们在使用 STM32 芯片的时候首先要进行系统初始化， CMSIS 规范就规定，系统初始化函数名字必须为 SystemInit，所以各个芯片公司写自己的库函数的时候就必须用 SystemInit 对系统进行初始化。CMSIS 还对各个外设驱动文件的文件名 字规范化，以及函数名字规范化等等一系列规定。上一节讲的函数GPIO_ResetBits 这个函数名字也是不能随便定义的，是要遵循 CMSIS 规范的。

3.2.4 官方固件库安装

这一节内容主要讲解 ST 官方提供的 STM32F4 固件库包的结构 。固件库是不断完善升级的，所以有不同的版本，我们使用的是 V1.5 版本的固件库,大家可以到以下目录找到其压缩文件：

\STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib.zip

然后解压即可。 该压缩包是我们从ST官方下载[1]的，目前最新版V1.5 ，感兴趣的同学也可去检查是否有升级版下载。下面看看官方库包的目录结构，如下图所示：

```
Libraries
├──CMSIS
│  │ CMSIS_END_USER_LICENCE_AGREEMENT.pdf
│  │ CMSIS_END_USER_LICENCE_AGREEMENT.rtf
│  │ index.html
│  │ README.txt
│  │
│  ├──Device
│  │  └──ST
│  │  └──STM32F4xx
│  │  │ Release_Notes.html
│  │  │
│  │  ├──Include
│  │  │ stm32f4xx.h
│  │  │ system_stm32f4xx.h
│  │  │
│  │  └──Source
│  │  └──Templates
│  │  │ system_stm32f4xx.c
│  │  │
│  │  ├──arm
│  │  │ startup_stm32f401xx.s
│  │  │ startup_stm32f40xx.s
│  │  │ startup_stm32f40_41xxx.s
│  │  │ startup_stm32f411xe.s
```

[1] http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1939/PF257901

```
│  │  │ startup_stm32f427x.s
│  │  │ startup_stm32f427_437xx.s
│  │  │ startup_stm32f429_439xx.s
│  │  │ startup_stm32f446xx.s
│  │  │
│  │  ├──gcc_ride7
│  │  │ startup_stm32f401xx.s
│  │  │ startup_stm32f40xx.s
│  │  │ startup_stm32f40_41xxx.s
│  │  │ startup_stm32f427x.s
│  │  │ startup_stm32f427_437xx.s
│  │  │ startup_stm32f429_439xx.s
│  │  │
│  │  ├──iar
│  │  │ startup_stm32f401xx.s
│  │  │ startup_stm32f40xx.s
│  │  │ startup_stm32f40_41xxx.s
│  │  │ startup_stm32f411xe.s
│  │  │ startup_stm32f427x.s
│  │  │ startup_stm32f427_437xx.s
│  │  │ startup_stm32f429_439xx.s
│  │  │ startup_stm32f446xx.s
│  │  │
│  │  ├──TASKING
│  │  │ cstart_thumb2.asm
│  │  │
│  │  └──TrueSTUDIO
│  │ startup_stm32f401xx.s
│  │ startup_stm32f40xx.s
│  │ startup_stm32f40_41xxx.s
│  │ startup_stm32f411xe.s
│  │ startup_stm32f427xx.s
│  │ startup_stm32f427_437xx.s
│  │ startup_stm32f429_439xx.s
│  │ startup_stm32f446xx.s
│  │
│  ├──DSP_Lib
│  │  │ license.txt
│  │  │
│  │  └──Source
│  │  ├──BasicMathFunctions
│  │  │ arm_abs_f32.c
│  │  │ arm_abs_q15.c
│  │  │ arm_abs_q31.c
│  │  │ arm_abs_q7.c
```

```
│   │   │ arm_add_f32.c
│   │   │ arm_add_q15.c
│   │   │ arm_add_q31.c
│   │   │ arm_add_q7.c
│   │   │ arm_dot_prod_f32.c
│   │   │ arm_dot_prod_q15.c
│   │   │ arm_dot_prod_q31.c
│   │   │ arm_dot_prod_q7.c
│   │   │ arm_mult_f32.c
│   │   │ arm_mult_q15.c
│   │   │ arm_mult_q31.c
│   │   │ arm_mult_q7.c
│   │   │ arm_negate_f32.c
│   │   │ arm_negate_q15.c
│   │   │ arm_negate_q31.c
│   │   │ arm_negate_q7.c
│   │   │ arm_offset_f32.c
│   │   │ arm_offset_q15.c
│   │   │ arm_offset_q31.c
│   │   │ arm_offset_q7.c
│   │   │ arm_scale_f32.c
│   │   │ arm_scale_q15.c
│   │   │ arm_scale_q31.c
│   │   │ arm_scale_q7.c
│   │   │ arm_shift_q15.c
│   │   │ arm_shift_q31.c
│   │   │ arm_shift_q7.c
│   │   │ arm_sub_f32.c
│   │   │ arm_sub_q15.c
│   │   │ arm_sub_q31.c
│   │   │ arm_sub_q7.c
│   │   │
│   │   ├──CommonTables
│   │   │ arm_common_tables.c
│   │   │ arm_const_structs.c
│   │   │
│   │   ├──ComplexMathFunctions
│   │   │ arm_cmplx_conj_f32.c
│   │   │ arm_cmplx_conj_q15.c
│   │   │ arm_cmplx_conj_q31.c
│   │   │ arm_cmplx_dot_prod_f32.c
│   │   │ arm_cmplx_dot_prod_q15.c
│   │   │ arm_cmplx_dot_prod_q31.c
│   │   │ arm_cmplx_mag_f32.c
│   │   │ arm_cmplx_mag_q15.c
```

```
│   │   │ arm_cmplx_mag_q31.c
│   │   │ arm_cmplx_mag_squared_f32.c
│   │   │ arm_cmplx_mag_squared_q15.c
│   │   │ arm_cmplx_mag_squared_q31.c
│   │   │ arm_cmplx_mult_cmplx_f32.c
│   │   │ arm_cmplx_mult_cmplx_q15.c
│   │   │ arm_cmplx_mult_cmplx_q31.c
│   │   │ arm_cmplx_mult_real_f32.c
│   │   │ arm_cmplx_mult_real_q15.c
│   │   │ arm_cmplx_mult_real_q31.c
│   │   │
│   │   ├──ControllerFunctions
│   │   │ arm_pid_init_f32.c
│   │   │ arm_pid_init_q15.c
│   │   │ arm_pid_init_q31.c
│   │   │ arm_pid_reset_f32.c
│   │   │ arm_pid_reset_q15.c
│   │   │ arm_pid_reset_q31.c
│   │   │ arm_sin_cos_f32.c
│   │   │ arm_sin_cos_q31.c
│   │   │
│   │   ├──FastMathFunctions
│   │   │ arm_cos_f32.c
│   │   │ arm_cos_q15.c
│   │   │ arm_cos_q31.c
│   │   │ arm_sin_f32.c
│   │   │ arm_sin_q15.c
│   │   │ arm_sin_q31.c
│   │   │ arm_sqrt_q15.c
│   │   │ arm_sqrt_q31.c
│   │   │
│   │   ├──FilteringFunctions
│   │   │ arm_biquad_cascade_df1_32x64_init_q31.c
│   │   │ arm_biquad_cascade_df1_32x64_q31.c
│   │   │ arm_biquad_cascade_df1_f32.c
│   │   │ arm_biquad_cascade_df1_fast_q15.c
│   │   │ arm_biquad_cascade_df1_fast_q31.c
│   │   │ arm_biquad_cascade_df1_init_f32.c
│   │   │ arm_biquad_cascade_df1_init_q15.c
│   │   │ arm_biquad_cascade_df1_init_q31.c
│   │   │ arm_biquad_cascade_df1_q15.c
│   │   │ arm_biquad_cascade_df1_q31.c
│   │   │ arm_biquad_cascade_df2T_f32.c
│   │   │ arm_biquad_cascade_df2T_f64.c
│   │   │ arm_biquad_cascade_df2T_init_f32.c
```

```
| | | arm_biquad_cascade_df2T_init_f64.c
| | | arm_biquad_cascade_stereo_df2T_f32.c
| | | arm_biquad_cascade_stereo_df2T_init_f32.c
| | | arm_conv_f32.c
| | | arm_conv_fast_opt_q15.c
| | | arm_conv_fast_q15.c
| | | arm_conv_fast_q31.c
| | | arm_conv_opt_q15.c
| | | arm_conv_opt_q7.c
| | | arm_conv_partial_f32.c
| | | arm_conv_partial_fast_opt_q15.c
| | | arm_conv_partial_fast_q15.c
| | | arm_conv_partial_fast_q31.c
| | | arm_conv_partial_opt_q15.c
| | | arm_conv_partial_opt_q7.c
| | | arm_conv_partial_q15.c
| | | arm_conv_partial_q31.c
| | | arm_conv_partial_q7.c
| | | arm_conv_q15.c
| | | arm_conv_q31.c
| | | arm_conv_q7.c
| | | arm_correlate_f32.c
| | | arm_correlate_fast_opt_q15.c
| | | arm_correlate_fast_q15.c
| | | arm_correlate_fast_q31.c
| | | arm_correlate_opt_q15.c
| | | arm_correlate_opt_q7.c
| | | arm_correlate_q15.c
| | | arm_correlate_q31.c
| | | arm_correlate_q7.c
| | | arm_fir_decimate_f3.c
| | | arm_fir_decimate_fast_q15.c
| | | arm_fir_decimate_fast_q31.c
| | | arm_fir_decimate_init_f32.c
| | | arm_fir_decimate_init_q15.c
| | | arm_fir_decimate_init_q31.c
| | | arm_fir_decimate_q15.c
| | | arm_fir_decimate_q31.c
| | | arm_fir_f32.c
| | | arm_fir_fast_q15.c
| | | arm_fir_fast_q31.c
| | | arm_fir_init_f32.c
| | | arm_fir_init_q15.c
| | | arm_fir_init_q31.c
| | | arm_fir_init_q7.c
```

```
│   │   │ arm_fir_interpolate_f32.c
│   │   │ arm_fir_interpolate_init_f32.c
│   │   │ arm_fir_interpolate_init_q15.c
│   │   │ arm_fir_interpolate_init_q31.c
│   │   │ arm_fir_interpolate_q15.c
│   │   │ arm_fir_interpolate_q31.c
│   │   │ arm_fir_lattice_f32.c
│   │   │ arm_fir_lattice_init_f32.c
│   │   │ arm_fir_lattice_init_q15.c
│   │   │ arm_fir_lattice_init_q31.c
│   │   │ arm_fir_lattice_q15.c
│   │   │ arm_fir_lattice_q31.c
│   │   │ arm_fir_q15.c
│   │   │ arm_fir_q31.c
│   │   │ arm_fir_q7.c
│   │   │ arm_fir_sparse_f32.c
│   │   │ arm_fir_sparse_init_f32.c
│   │   │ arm_fir_sparse_init_q15.c
│   │   │ arm_fir_sparse_init_q31.c
│   │   │ arm_fir_sparse_init_q7.c
│   │   │ arm_fir_sparse_q15.c
│   │   │ arm_fir_sparse_q31.c
│   │   │ arm_fir_sparse_q7.c
│   │   │ arm_iir_lattice_f32.c
│   │   │ arm_iir_lattice_init_f32.c
│   │   │ arm_iir_lattice_init_q15.c
│   │   │ arm_iir_lattice_init_q31.c
│   │   │ arm_iir_lattice_q15.c
│   │   │ arm_iir_lattice_q31.c
│   │   │ arm_lms_f32.c
│   │   │ arm_lms_init_f32.c
│   │   │ arm_lms_init_q15.c
│   │   │ arm_lms_init_q31.c
│   │   │ arm_lms_norm_f32.c
│   │   │ arm_lms_norm_init_f32.c
│   │   │ arm_lms_norm_init_q15.c
│   │   │ arm_lms_norm_init_q31.c
│   │   │ arm_lms_norm_q15.c
│   │   │ arm_lms_norm_q31.c
│   │   │ arm_lms_q15.c
│   │   │ arm_lms_q31.c
│   │   │
│   │   ├──MatrixFunctions
│   │   │ arm_mat_add_f32.c
│   │   │ arm_mat_add_q15.c
```

```
│   │   │ arm_mat_add_q31.c
│   │   │ arm_mat_cmplx_mult_f32.c
│   │   │ arm_mat_cmplx_mult_q15.c
│   │   │ arm_mat_cmplx_mult_q31.c
│   │   │ arm_mat_init_f32.c
│   │   │ arm_mat_init_q15.c
│   │   │ arm_mat_init_q31.c
│   │   │ arm_mat_inverse_f32.c
│   │   │ arm_mat_inverse_f64.c
│   │   │ arm_mat_mult_f32.c
│   │   │ arm_mat_mult_fast_q15.c
│   │   │ arm_mat_mult_fast_q31.c
│   │   │ arm_mat_mult_q15.c
│   │   │ arm_mat_mult_q31.c
│   │   │ arm_mat_scale_f32.c
│   │   │ arm_mat_scale_q15.c
│   │   │ arm_mat_scale_q31.c
│   │   │ arm_mat_sub_f32.c
│   │   │ arm_mat_sub_q15.c
│   │   │ arm_mat_sub_q31.c
│   │   │ arm_mat_trans_f32.c
│   │   │ arm_mat_trans_q15.c
│   │   │ arm_mat_trans_q31.c
│   │   │
│   │   ├──StatisticsFunctions
│   │   │ arm_max_f32.c
│   │   │ arm_max_q15.c
│   │   │ arm_max_q31.c
│   │   │ arm_max_q7.c
│   │   │ arm_mean_f32.c
│   │   │ arm_mean_q15.c
│   │   │ arm_mean_q31.c
│   │   │ arm_mean_q7.c
│   │   │ arm_min_f32.c
│   │   │ arm_min_q15.c
│   │   │ arm_min_q31.c
│   │   │ arm_min_q7.c
│   │   │ arm_power_f32.c
│   │   │ arm_power_q15.c
│   │   │ arm_power_q31.c
│   │   │ arm_power_q7.c
│   │   │ arm_rms_f32.c
│   │   │ arm_rms_q15.c
│   │   │ arm_rms_q31.c
│   │   │ arm_std_f32.c
```

```
│   │   │ arm_std_q15.c
│   │   │ arm_std_q31.c
│   │   │ arm_var_f32.c
│   │   │ arm_var_q15.c
│   │   │ arm_var_q31.c
│   │   │
│   │   ├──SupportFunctions
│   │   │ arm_copy_f32.c
│   │   │ arm_copy_q15.c
│   │   │ arm_copy_q31.c
│   │   │ arm_copy_q7.c
│   │   │ arm_fill_f32.c
│   │   │ arm_fill_q15.c
│   │   │ arm_fill_q31.c
│   │   │ arm_fill_q7.c
│   │   │ arm_float_to_q15.c
│   │   │ arm_float_to_q31.c
│   │   │ arm_float_to_q7.c
│   │   │ arm_q15_to_float.c
│   │   │ arm_q15_to_q31.c
│   │   │ arm_q15_to_q7.c
│   │   │ arm_q31_to_float.c
│   │   │ arm_q31_to_q15.c
│   │   │ arm_q31_to_q7.c
│   │   │ arm_q7_to_float.c
│   │   │ arm_q7_to_q15.c
│   │   │ arm_q7_to_q31.c
│   │   │
│   │   └──TransformFunctions
│   │ arm_bitreversal.c
│   │ arm_bitreversal2.S
│   │ arm_cfft_f32.c
│   │ arm_cfft_q15.c
│   │ arm_cfft_q31.c
│   │ arm_cfft_radix2_f32.c
│   │ arm_cfft_radix2_init_f32.c
│   │ arm_cfft_radix2_init_q15.c
│   │ arm_cfft_radix2_init_q31.c
│   │ arm_cfft_radix2_q15.c
│   │ arm_cfft_radix2_q31.c
│   │ arm_cfft_radix4_f32.c
│   │ arm_cfft_radix4_init_f32.c
│   │ arm_cfft_radix4_init_q15.c
│   │ arm_cfft_radix4_init_q31.c
│   │ arm_cfft_radix4_q15.c
```

```
│   │ arm_cfft_radix4_q31.c
│   │ arm_cfft_radix8_f32.c
│   │ arm_dct4_f32.c
│   │ arm_dct4_init_f32.c
│   │ arm_dct4_init_q15.c
│   │ arm_dct4_init_q31.c
│   │ arm_dct4_q15.c
│   │ arm_dct4_q31.c
│   │ arm_rfft_f32.c
│   │ arm_rfft_fast_f32.c
│   │ arm_rfft_fast_init_f32.c
│   │ arm_rfft_init_f32.c
│   │ arm_rfft_init_q15.c
│   │ arm_rfft_init_q31.c
│   │ arm_rfft_q15.c
│   │ arm_rfft_q31.c
│   │
│   ├──Include
│   │ arm_common_tables.h
│   │ arm_const_structs.h
│   │ arm_math.h
│   │ core_cm0.h
│   │ core_cm0plus.h
│   │ core_cm3.h
│   │ core_cm4.h
│   │ core_cm7.h
│   │ core_cmFunc.h
│   │ core_cmInstr.h
│   │ core_cmSimd.h
│   │ core_sc000.h
│   │ core_sc300.h
│   │
│   ├──Lib
│   │   │ license.txt
│   │   │
│   │   ├──ARM
│   │   │ arm_cortexM0b_math.lib
│   │   │ arm_cortexM0l_math.lib
│   │   │ arm_cortexM3b_math.lib
│   │   │ arm_cortexM3l_math.lib
│   │   │ arm_cortexM4bf_math.lib
│   │   │ arm_cortexM4b_math.lib
│   │   │ arm_cortexM4lf_math.lib
│   │   │ arm_cortexM4l_math.lib
│   │   │ arm_cortexM7bfdp_math.lib
```

```
│   │   │ arm_cortexM7bfsp_math.lib
│   │   │ arm_cortexM7b_math.lib
│   │   │ arm_cortexM7lfdp_math.lib
│   │   │ arm_cortexM7lfsp_math.lib
│   │   │ arm_cortexM7l_math.lib
│   │   │
│   │   └──GCC
│   └──RTOS
│   └──Template
│ cmsis_os.h
│
└──STM32F4xx_StdPeriph_Driver
│ Release_Notes.html
│
├──inc
│ misc.h
│ stm32f4xx_adc.h
│ stm32f4xx_can.h
│ stm32f4xx_cec.h
│ stm32f4xx_crc.h
│ stm32f4xx_cryp.h
│ stm32f4xx_dac.h
│ stm32f4xx_dbgmcu.h
│ stm32f4xx_dcmi.h
│ stm32f4xx_dma.h
│ stm32f4xx_dma2d.h
│ stm32f4xx_exti.h
│ stm32f4xx_flash.h
│ stm32f4xx_flash_ramfunc.h
│ stm32f4xx_fmc.h
│ stm32f4xx_fmpi2c.h
│ stm32f4xx_fsmc.h
│ stm32f4xx_gpio.h
│ stm32f4xx_hash.h
│ stm32f4xx_i2c.h
│ stm32f4xx_iwdg.h
│ stm32f4xx_ltdc.h
│ stm32f4xx_pwr.h
│ stm32f4xx_qspi.h
│ stm32f4xx_rcc.h
│ stm32f4xx_rng.h
│ stm32f4xx_rtc.h
│ stm32f4xx_sai.h
│ stm32f4xx_sdio.h
│ stm32f4xx_spdifrx.h
```

```
│  stm32f4xx_spi.h
│  stm32f4xx_syscfg.h
│  stm32f4xx_tim.h
│  stm32f4xx_usart.h
│  stm32f4xx_wwdg.h
│
└──src
misc.c
stm32f4xx_adc.c
stm32f4xx_can.c
stm32f4xx_cec.c
stm32f4xx_crc.c
stm32f4xx_cryp.c
stm32f4xx_cryp_aes.c
stm32f4xx_cryp_des.c
stm32f4xx_cryp_tdes.c
stm32f4xx_dac.c
stm32f4xx_dbgmcu.c
stm32f4xx_dcmi.c
stm32f4xx_dma.c
stm32f4xx_dma2d.c
stm32f4xx_exti.c
stm32f4xx_flash.c
stm32f4xx_flash_ramfunc.c
stm32f4xx_fmc.c
stm32f4xx_fmpi2c.c
stm32f4xx_fsmc.c
stm32f4xx_gpio.c
stm32f4xx_hash.c
stm32f4xx_hash_md5.c
stm32f4xx_hash_sha1.c
stm32f4xx_i2c.c
stm32f4xx_iwdg.c
stm32f4xx_ltdc.c
stm32f4xx_pwr.c
stm32f4xx_qspi.c
stm32f4xx_rcc.c
stm32f4xx_rng.c
stm32f4xx_rtc.c
stm32f4xx_sai.c
stm32f4xx_sdio.c
stm32f4xx_spdifrx.c
stm32f4xx_spi.c
stm32f4xx_syscfg.c
stm32f4xx_tim.c
```

stm32f4xx_usart.c
stm32f4xx_wwdg.c

3.2.5 官方固件库使用

文件目录介绍：

◆ Libraries 文件夹下面有 CMSIS 和 STM32F4xx_StdPeriph_Driver 两个目录，这两个目录包含固件库核心的所有子文件夹和文件。CMSIS 文件夹存放的是符合 CMSIS 规范的一些文件。包括 STM32F4 核内外设访问层代码，DSP 软件库，RTOS API，以及 STM32F4 片上外设访问层代码等。我们新建工程的时候会添加文件到我们工程，需要注意的是该目录下的文件设定为只读权限，非必要的情况下不要修改里面的源文件，以免影响其他项目工程。

◆ STM32F4xx_StdPeriph_Driver 放的是 STM32F4 标准外设固件库源码文件和对应的头文件。

◆ inc 目录存放的是 stm32f4xx_ppp.h 头文件,无需改动。 src 目录下面放的是 stm32f4xx_ppp.c 格式的固件库源码文件。每一个.c 文件和一个相应的.h 文件对应。这里的文件也是固件库外设的关键文件，每个外设对应一组文件。

◆ Libraries 文件夹里面的文件在我们建立工程的时候都会使用到。

◆ Project 文件夹下面有两个文件夹。顾名思义， STM32F4xx_StdPeriph_Examples 文件夹下面存放的的 ST 官方提供的固件实例源码，在以后的开发过程中，可以参考修改这个官方提供的实例来快速驱动自己的外设，很多开发板的实例都参考了官方提供的例程源码，这些源码对以后的学习非常重要。

◆ STM32F4xx_StdPeriph_Template 文件夹下面存放的是工程模板。

◆ Utilities 文件下就是官方评估板的一些对应源码，这个对于本手册学习可以忽略不看。根目录中还有一个 stm32f4xx_dsp_stdperiph_lib_um.chm 文件，直接打开可以知道，这是一个固件库的帮助文档，这个文档非常有用，只可惜是英文的，在开发过程中，这个文档会经常被使用到。

关键文件介绍：

在介绍一些关键文件之前，首先我们来看看一个基于固件库的 STM32F4 工程需要哪些关键文件，这些文件之间有哪些关联关系。其实这个可以从 ST 提供的英文版的 STM32F4 固件库说明里面找到。

◆ core_cm4.h 文件位于\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Include 目录下面的，这个就是 CMSIS 核心文件，提供进入 M4 内核接口，这是 ARM 公司提供，对所有CM4 内核的芯片都一样。

◆ stm32f4xx.h 和 system_stm32f4xx.h 文件存放在文件夹\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Librari
\CMSIS\Device\ST\STM32F4xx\Include 下面。

◆ system_stm32f4xx.h 是片上外设接入层系统头文件。主要是申明设置系统及总线时钟相关的函数。与其对应的源文件 system_stm32f4xx.c 在目录\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Proj
\STM32F4xx_StdPeriph_Templates 可以找到。这个里面有一个非常重要的 SystemInit()函数申明，这个函数在我们系统启动的时候都会调用，用来设置系统的整个系统和总线时钟。

◆ stm32f4xx.h 是 STM32F4 片上外设访问层头文件。这个文件就相当重要了，只要你做STM32F4 开发，你几乎时刻都要查看这个文件相关的定义。这个文件打开可以看到，里面非常多的结构体以及宏定义。这个文件里面主要是系统寄存器定义申明以及包装内存操作，对于这里是怎样申明以及怎样将内存操作封装起来的。

◆ stm32f4xx_it.c,stm32f4xx_it.h 以及 stm32f4xx_conf.h 等文件，我们可以从\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Project\STM32F4xx_StdPeriph_Templates 文件夹中找到。这几个文件我们后面新建工程也有用到。stm32f4xx_it.c 和 stm32f4xx_it.h 里面是用来编写中断服务函数，中断服务函数也可以随意编写在工程里面的任意一个文件里面

◆ stm32f4xx_conf.h 是外设驱动配置文件。 文件打开可以看到一堆的#include,这里你建立工程的时候，可以注释掉一些你不用的外设头文件。

◆ 对于上图中的 misc.c,misc.h,stm32f4xx_ppp.c,stm32f4xx_ppp.h 以及 stm32f4xx_rcc.c 和stm32f4xx_rcc.h 文件，这些文件存放在目录 Libraries\STM32F4xx_StdPeriph_Driver。这些文件是 STM32F4 标准的外设库文件。其中 misc.c 和 misc.h 是定义中断优先级分组以及 Systick 定时器相关的函数。

◆ stm32f3xx_rcc.c 和 stm32f4xx_rcc.h 是与 RCC 相关的一些操作函数，作用主要是一些时钟的配置和使能。在任何一个 STM32 工程 RCC 相关的源文件和头文件是必须添加的。

◆ 对于文件 stm32f4xx_ppp.c 和 stm32f4xx_ppp.h，这就是 stm32F4 标准外设固件库对应的源文件和头文件。包括一些常用外设 GPIO,ADC,USART 等。

◆ 文件 Application.c 实际就是说是应用层代码。这个文件名称可以任意取了。我们工程中，直接取名为 main.c.实际上一个完整的 STM32F4 的工程光有上面这些文件还是不够的。还缺少非常关键的启动文件。STM32F4 的启动文件存放在目录\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm 下面。对于不同型号的 STM32F4 系列对应的启动文件也不一样。我们的开发板是 STM32F407 系列所以我们选择的启动文件为startup_stm32f40_41xxx.s。启动文件到底什么作用，其实我们可以打开启动文件进去看看。启动文件主要是进行堆栈之类的初始化，中断向量表以及中断函数定义。启动文件要引导进入main 函数。 Reset_Handler 中断函数是唯一实现了的中断处理函数，其他的中断函数基本都是死循环。 Reset_handler 在我们系统启动的时候会调用，下面让我们看看 Reset_handler 这段代码：

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

这段代码的作用是在系统复位之后引导进入 main 函数，同时在进入 main 函数之前，首先要调用 SystemInit 系统初始化函数。

3.3 ARM MDK开发工具的安装与使用

RealView MDK开发套件源自德国Keil公司（后被ARM公司收购），是ARM公司目前最新推出的针对各种嵌入式处理器的软件开发工具。RealView MDK集成了业内最领先的技术，包括μVision5v集成开发环境与 RealView编译器。支持ARM7、ARM9和最新的Cortex-M3,Cortex-M3核处理器，自动配置启动代码，集成Flash烧写模块，强大的Simulation设备模拟，性能分析等功能，与ARM之前的工具包ADS等相比，RealView编译器的最新版本可将性能改善超过20%。

3.3.1 ARM MDK安装

1、 在Elink407\tools\arm mdk目录下找到MDK_510.exe软件，双击mdk_510.exe应用程序，进入安装界面。依次选择“next”->“勾选同意协议，next”->“选择安装目录，next“->输入姓名、公司名、email信息(任意填写)->“next，等待安装”->“finish”。完成安装之后会弹出补丁安装界面，选择“OK“后，有提示部分补丁读取失败，直接点击确定忽略即可，如图所示：

在Pack Intaller窗口，依次选择打开“File”->“Impot..”，导入arm mdk 目录下的补丁文件(Keil.STM32F4xx_DFP.1.0.6)，如图所示 ：

2、运行 MDK(如果是win8 win7 建议用管理员权限打开)，打开”File“下的”License Management…“，复制 CID 编号（11 字符）。

3、 解压keygen.zip运行破解软件 keygen.exe。有些杀毒软件（如卡巴斯基）会误报为病毒程序，请临时屏蔽掉杀毒软件后再运行。

4、 keygen.exe的“target”下拉列表框选择“ARM”。第 2 行下拉列表框选择“Prof.Developers Kit/RealView MDK”

Pack Installer
File Packs Window Help
Device:
Packs
Examples
Pack
Action
Description
Devices
Boards
Search:
Device
Summary
Updating Pack descriptions , downloading http://www.wolfSSL.com/files/ide/wolfSSL.CyaSSL.pdsc
96%
ONLINE

License Management
Single-User License
Floating License
Floating License Administrator
FlexLM License
Customer Information
Name:
dd dd
Company:
dd
Email:
dd
Computer ID
CID:
C51PZ-IFNTI
Get LIC via Internet...
Product
License ID Code...
Support Period
MDK-Lite
Evaluation Version
New License ID Code (LIC):
Add LIC
Uninstall...
Close
Help

Keil Generic Keygen - EDGE
Keil Embedded Workbench
Keygen
License Details
CID:
C51PZ-IFNTI
Target
ARM
MDK Professional
AK1Y8-3FI6M-KSKYM-XU4Y0-FB8KY-F9EVV
Generate
Exit

5、 将 MDK 的 CID 编号粘贴到 KEIL_lic 中的 CID 编辑框内，然后点击“Generate”按钮。

6、 把方框中生成的 30 位注册码，拷贝到 MDK 中的“New License ID Code”内。

7、 点击“AddLIC”完成注册。如果截止日期太短的话，就继续用 keil_Lic 生成注册码重新注册。

8、 点击“AddLIC”完成注册。

3.3.2 使用ARM MDK进行Elink407软件开发

第一个工程(工程模版目录结构)

良好的目录结构是个好的习惯，这是我们基于固件库第一个工程，以后我们基于固件库开发都以此为工程模版组织代码。 我们建立如下目录

```
.  ├── STM32_Lib
│      └── stm32标准库目录
├── 基础实验
│  ├── bsp4Elink //编写好的驱动源代码
│  │  ├── inc
│  │  │  ├── 编写的头文件.h
│     │     └── src
│     │     ├── 编写的头文件.c
│  ├── common
│  │  ├── IAR
│  │  ├── MDK
```

```
│ │ └── 一些公共的sys文件
│       └── project
│       ├── IAR //IAR案例目录
│       │     └── firstPrj //第一个工程，IAR工程模版目录
│       │     ├── bin //生成的烧录二进制文件目录
│       │     ├── project.ewp //第一个工程文件
│       │     ├── list //编译过程中生成的lst文件目录
│       │     ├── obj //编译过程中生成的obj文件目录
│       │     └── user //测试驱动代码的main.c等相关文件
│       │     └── main.c
│       └── MDK //MDK案例目录
│       └── firstPrj //第一个工程，MDK工程模版目录
│           ├── bin //生成的烧录二进制文件目录
│           ├── project.uvproj //第一个工程文件
│           ├── list //编译过程中生成的lst文件目录
│           ├── obj //编译过程中生成的obj文件目录
│           └── user //测试驱动代码的main.c等相关文件
│           └── main.c
└── 扩展实验
……..
```

创建新的工程firstPrj，双击“Keil uVision5”在菜单栏“Project”中选择“New uVision Project”。在firstPrj目录中保存工程文件为firePrj.uvproj，如图：

保存后，会提示选择CPU型号，在这里我们选择STM32F407VG，如图，弹出窗口再选择OK就可以了。

3.3.3 配置工程

进入工程配置界面的方法： 菜单 > project > Options For Target 或者选择如下图按钮

1） “Device” 选项。选择正确的CPU型号。

2） “Target” 选项。

勾选” Use MicroLIB” 。如果不选，会提示__main 函数不存在，启动代码需要跳到__main 然后再跳到用户的main函数中，其次我们习惯调用的printf函数将无法输出信息到串口。

3） “Output” 选项。

此处可修改可执行文件名。 勾选 “Create HEX File” ,表示生成可单独烧写的Hex文件。 取消 “Browse Information” 选项可以大大加快每个文件的编译速度。但将丧失鼠标右键定位函数和宏的功能。这个选项不影响最终的Hex。

在编译过程中，会生成大量的obj临时文件，你可以通过 “Select Folder for Objects” 按钮指定Obj文件的存放目录。这样将是工程文件目录变得非常简洁。

4） “Listing” 选项。

在编译过程中，会生成大量的lst临时文件，你可以通过 “Select Folder for Listings” 按钮指定lst文件的存放目录。这样将是工程文件目录变得非常简洁。

5） “User” 选项。可以指定编译之前和编译之后执行的用户程序。

为了将编译生成的Hex文件复制到指定目录并更名保存，我们可以在编译后执行一个bat批处理文件。 CopyHex_Flash.bat 文件是一个文本文件，内容如下： copy Flash\Obj\output.hex ..\output(mdk).hex

6） “C/C++” 选项。

此处 “USE_STDPERIPH_DRIVER” 表示使用ST的固件库；

E:\Elink407\source_dir\基础实验\project\led_nolib\MDK\led_onlib.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
error
Target 1
Project
Target 1
Options for Target...
Configure target options
Build Output
Configure target options
ULINK2/ME Cortex Debugger

Select Device for Target 'Target 1'...
CPU
Vendor: STMicroelectronics
Device: STM32F407VG
Toolset: ARM
Search:
Description:
ARM
STMicroelectronics
STM32F2 Series
STM32F4 Series
STM32F401
STM32F405
STM32F407
STM32F407IE
STM32F407IG
STM32F407VE
STM32F407VG
The STM32F4 family incorporates high-speed embedded memories and
- 64-Kbyte of CCM (core coupled memory) data RAM
- LCD parallel interface, 8080/6800 modes
- Timer with quadrature (incremental) encoder input
- 5 V-tolerant I/Os
- Parallel camera interface
- True random number generator
- RTC: subsecond accuracy, hardware calendar
- 96-bit unique ID
OK
Cancel
Help

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
STMicroelectronics STM32F407VG
Xtal (MHz): 168.0
Operating system: None
System-Viewer File (.Sfr):
STM32F40x.svd
Use Custom SVD File
Code Generation
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Floating Point Hardware: Use FPU
Read/Only Memory Areas
default off-chip Start Size Startup
ROM1:
ROM2:
ROM3:
on-chip
IROM1: 0x8000000 0x100000
IROM2:
Read/Write Memory Areas
default off-chip Start Size NoInit
RAM1:
RAM2:
RAM3:
on-chip
IRAM1: 0x20000000 0x20000
IRAM2: 0x10000000 0x10000
OK
Cancel
Defaults
Help

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Select Folder for Objects...
Name of Executable: led_nolib
Create Executable: .\obj\led_nolib
Debug Information
Create HEX File
Browse Information
Create Library: .\obj\led_nolib.lib
Create Batch File
OK
Cancel
Defaults
Help

Browse for Folder
Folder:
obj
名称
修改日期
类型
没有与搜索条件匹配的项。
Path:
ource_dir\基础实验\project\led_nolib\MDK\obj
OK

Browse for Folder
Folder:
list
名称
修改日期
类型
没有与搜索条件匹配的项。
Path:
ource_dir\基础实验\project\led_nolib\MDK\list\
OK

如果有其他的预编译宏，可以用逗号分割。

Include Path 指定头文件的路搜索路径 添加 diag_suppress=870 控制，可以避免中文字符串告警

7） “Asm” 选项。无需更改。

8） “Linker” 选项。无需更改。

9） “Debug” 选项。

点击” ” 按钮设置硬件仿真器，请选择对应的仿真器型号。我们的例程缺省都配置的是ST-Link 仿真器。 勾选 “Run to main()” 可以在启动Debug时，执行到main()函数暂停。

我们以ST-Link仿真器为例，点击 “Setting” 后，界面如下：

Port框，请选择” SW” 表示使用SWD调试接口。

10） “Utilities” 选项。

点击” Settings” 按钮设置硬件仿真器，请选择对应的仿真器型号。我们的例程缺省都配置的是ST-Link 仿真器。 我们以ST-Link仿真器为例，点击 “Setting” 后，界面如下：

选择 “Flash DownLoad”

点击” Add” 按钮，添加Flash编程算法。

Project Targets 一栏，我们将 Target 名字修改为 firstPrj,然后在 Groups 一栏删掉一个 Source Groupl，建立三个 Groups：sys,drv,bsp,user。然后点击 OK，可以看到我们的 Target名字以及 Groups 情况如下图：

我们的 CORE 下面需要添加的文件为 startup_stm32f40_41xxx.s(注意，默认添加的时候文件类型为.c,也就是添加 startup_stm32f40_41xxx.s 启动文件的时候，你需要选择文件类型为 All files才能看得到这个文件 ）， USER 目录下面需要添加的文件为 main.c ， stm32f4xx_it.c ， system_stm32f4xx.c。

Cortex-M Target Driver Setup
Debug
Trace
Flash Download
Debug Adapter
Unit: ST-LINK/V2
Serial Number: N/A
HW Version: V2
Firmware V2J20S4
Port: SW
Max 1MHz
SW Device
IDCODE
Device Name
SWDIO
0x2BA01477
ARM CoreSight SW-DP
Move
Up
Down
Automatic Detection
ID CODE:
Manual Configuration
Device Name:
Add
Delete
Update
IR len:
Debug
Connect & Reset Options
Connect: Normal
Reset: Autodetect
Reset after Connect
Cache Options
Cache Code
Cache Memory
Download Options
Verify Code Download
Download to Flash
确定
取消
应用(A)

Cortex-M Target Driver Setup
Debug
Trace
Flash Download
Download Function
LOAD
Erase Full Chip
Erase Sectors
Do not Erase
Program
Verify
Reset and Run
RAM for Algorithm
Start: 0x20000000
Size: 0x0800
Programming Algorithm
Description
Device Size
Device Type
Address Range
Start:
Size:
Add
Remove
确定
取消
应用(A)

Add Flash Programming Algorithm
Description
Flash Size
Device Type
Origin
STM32F40xxx/41xxx Flash ...
4B
On-chip Flash
Device Family Package
STM32F4xx Flash
1M
On-chip Flash
Device Family Package
STM32F4xx Flash OTP
528B
On-chip Flash
Device Family Package
LPC18xx/43xx S25FL032 SP...
4M
Ext. Flash SPI
MDK Core
RC28F640J3x Dual Flash
16M
Ext. Flash 32-bit
MDK Core
S29GL064N Dual Flash
16M
Ext. Flash 32-bit
MDK Core
d:\Keil_v5\ARM\PACK\Keil\STM32F4xx_DFP\1.0.6\Flash\STM32F4xx_1024.FLM
Add
Cancel

Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
firstPrj
Groups:
sys
drv
bsp
user
Files:
Set as Current Target
Add Files...
Add Files as Image...
OK
Cancel
Help

\source\STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm\startup_stm32f40xxx.s \source\STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\system_stm32f4xx.c

添加标准设备驱动库文件，源文件所在目录为：

\source_dir\STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\STM32F4xx_StdPeriph_Driver\src ctrl+a 选择全部，添加后如下图：

bsp组是我们编写的控制ELink407开发板的具体设备驱动文件，在我们的新工程中留空，以后将目录\基础实验\bsp4Elink\src的文件添加到该组，添加user文件main.c，源文件所在目录为：\基础实验\project\firstPrj\MDK\user

添加一个全局宏定义标识符。添加方法是点击魔术棒之后，进入 C/C++选项卡，然后在 Define 输入框连输入：STM32F40xxx,USE_STDPERIPH_DRIVER

添加头文件，设置头文件存放路径。也就是告诉 MDK 到那些目录下面去寻找包含了的头文件。这一步骤非常重要。如果没有设置头文件路径，那么工程会出现报错头文件路径找不到，添加相应的头文件路径：

基础实验\project\firstPrj\MDK\user

基础实验\bsp4Elink\inc

STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\STM32F4xx_StdPeriph_Driver\inc

STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\CMSIS\Device\ST\STM32F4xx\Include

添加后如下图

11） 配置完毕。

3.3.4 下载与调试

新建源文件

保存文件为main.c

0x4002 3800 - 0x4002 3BFF	RCC	AHB1	*第 171 页的第 6.3.32 节：RCC 寄存器映射*
0x4002 3000 - 0x4002 33FF	CRC		*第 85 页的第 4.4.4 节：CRC 寄存器映射*
0x4002 2000 - 0x4002 23FF	GPIOI		*第 192 页的第 7.4.11 节：GPIO 寄存器映射*
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		

7.4.1 GPIO 端口模式寄存器 (GPIOx_MODER) (x = A..I)

GPIO port mode register

偏移地址：0x00

E:\Elink407\source_dir\基础实验\project\led_nolib\MDK\led_onlib.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
error
New (Ctrl+N)
Create an empty document
Target 1
Source Group 1
Text1
Build Output
Create an empty document
ST-Link Debugger
linkedu77 - 与服务

Browse for Folder
Folder:
obj
名称
修改日期
类型
没有与搜索条件匹配的项。
Path:
ource_dir\基础实验\project\led_nolib\MDK\obj\
OK

3.4 IAR for ARM 工具软件的安装使用

IAR Embedded Workbench for ARM version 是一个针对ARM 处理器的集成开发环境，包含项目管理器、编辑器、编译连接工具和支持RTOS 的调试工具，在该环境下可以使用C/C++和汇编语言方便地开发嵌入式应用程序。对于30余种不同架构的处理器，IAR Embedded Workbench提供统一的使用界面，是一个真正的集成开发环境；方便了从8位/16位处理器转向ARM的用户。

3.4.1 IAR for ARM 安装

在Elink407\tools\EWARM目录下找到安装软件。

CD-EWARM-6301-3142.zip ：安装文件
EWARM-6310-3142-CRACK.zip ：软件注册机

安装步骤:

1、 将 CD-EWARM 6301 3142.zip 文件解压到硬盘任意文件夹。

2、 将 EWARM 6310 3142-CRACK.zip 文件解压到硬盘任意文件夹。

3、 首先运行CRACK下面的 !RUNME.BAT， 会生成license.txt文件 （每台计算机这个文件都不同）

a）用记事本打开license.txt文件，找到第4个Installserial；

b）安装EWARM 6.30中，需要系列号时拷贝license.txt文件的第4个Installserial后面的数字；

c）安装EWARM 6.30中，需要key时拷贝license.txt文件的与第4个Installserial相接的key后面的长字符串。

4、 运行安装目录下的 autorun.exe

5、 点击第2项“Install IAR Embedded Workbench”开始安装。等待若干秒后，将欢迎界面:

6、 点击“Next”按钮继续安装。出现软件授权声明界面:

7、 选择“I accept the terms of the license agreement ”同意，然后点击“Next”按钮继续安装。出现输入用户信息的界面:

8、 在 License# 栏输入第3步中获得 Installserial 数字。 然后点击 Next，出现注册窗口:

9、 在 License Key 栏输入第3步中获得 Key 数字。 然后点击 Next

10、 这一步可以选择目标文件夹，我们保持缺省路径不变，点击Next

11、 选择开始菜单中的程序文件夹名字，一般保持不变，点击Next，出现确认开始安装的界面:

12、 点击 Install， 开始安装

13、 等待十几分钟后，出现安装完成的界面

14、 点击“Finish”按钮。安装成功结束。

3.4.2 使用IAR for ARM进行Elink407软件开发

3.4.3 第一个工程

见MDK章节

3.4.4 工程目录结构

见MDK章节

3.4.5 配置工程

进入工程配置界面的方法: 菜单 project Options…

1) “General Options” - “Target” 选项。

“Processor variant” 栏选择

“Device” ,CPU型号选择 “ST STM32F4xxx”

“FPU” 浮点单元选择 “VFPv4”

2) “General Options - Output” 选项。

可以修改输出文件的目录名称。由于是定位在Flash的工程，我们可以修改名字为Flash。如果是定位在RAM的工程，可以修改名字为CpuRAM。目录名字可以自己随意起。

3) “General Options - Library Configuration” 选项。

“Library” 选择 “Full” 以支持中文字符串。

不要勾选 “Use CMSIS” ，因为我们在源代码中已经明确包含了DSP库的头文件。无需编译器自动寻找。

4) “General Options - Library Options” 选项。保持不变。

随后的 “MISRA-C:2004” 和 “MISRA-C:1998” 是C语言语法控制选项，可以保持不变。

5) “C/C++ Complier - Optimizations” 编译优化选项。

为了调试跟踪方便，我们将C编译器优化级别设置为 “None” 。如果你的工程对速度和代码有非常严格的要求，可以提高优化级别。

6) “C/C++ Complier - Preprocessor” C编译器预处理选项。

在Include 路径中必须正确添加所有的h文件的搜索路径。手工方式。 其中 “$PROJ_DIR$” 关键字表示工程文件所在的目录。” ..\” 表示上级目录，连续两个表示上上级目录。在Define符号栏，必须添加 “USE_STDPERIPH_DRIVER” 表示使用ST的标准外设固件库。 对于用到USB和以太网的工程，可能还需要添加：USB和PHY相关的宏

7) “C/C++ Complier” 中其它未提及的选项，比如 “Output” 、 “List” 等可以保持不变。

8) “Assembler” 汇编器中所有的选项均保持不变。

9) “Output Converter - Output” 选项。

为了能够编译生成可以直接烧写的可执行程序，需要勾选 “Generate additional output” 。 输出文件格式一般选择 Intel extened 格式，也就是hex文件。

输出的缺省hex文件名为 “Project.hex” ,存放路径为: “Fash\Exe\Project.hex”

10) “Build Actions” 选项。

为了将编译生成的Hex文件复制到指定目录并更名保存，我们可以在编译后执行一个bat批处理文件。 CopyHex_Flash.bat 文件是一个文本文件，内容如下: copy Flash\Exe\project.hex ..\output(iar).hex 11) “Linker - Config” 选项。这是比较重要的选项，决定程序定位地址，堆和栈的大小。

“icf” 文件用于保存这些参数，我们将其存放工程文件目录下，名字可以自定义。

点击 “Edit” 按钮，弹出对话框，用于设置ROM ，RAM，STACK，HEAP。

12) “Linker - Output” 选项。

在此处可更改输出的目标文件为更有意义的名字。

13) “Linker - List” 选项。

为了方便查看编译后的ROM、RAM占用大小、函数和变量定位地址、函数大小等信息，请勾选“Generate linker map file”。勾选后，连接器将生成一个文本格式的 map文件，可以使用记事本查看，也可以直接双击“Workspace”－“Output”分支下的 map文件打开。

14）“Linker”中其他未提及的选项，可以保持缺省设置不变。

15）“Debugger － Setup”选项。

选择使用的仿真器。这里以ST-Link仿真器为例。

勾选 Run to main，表示启动调试器后，程序执行到 main函数暂停。

16）“Debugger － Setup”选项。

请选择“Use flash loader”，调试器将选择缺省的下载配置文件。

17）“Debugger”中其它未提及的选项，可以保持缺省参数不变。

18）“ST-Link/ST-Trace － Setup”保持缺省参数不变。

19）“ST-Link/ST-Trace － Connection”选项。

20）其他未提及的选项，均保持缺省值不变。工程配置全部完成。

3.4.6 下载与调试

第 4 章

STM32F407 的存储器模型

4.1 存储器空间映射

Cortex-M4是32位的内核，因此其PC指针可以指向2 32=4G的地址空间，也就是0x0000_0000——0xFFFF_FFFF这一大块空间。程序存储器、数据存储器、寄存器和I/O 端口排列在同一个顺序的4 GB 地址空间内。

Cortex-M4内核的设计导致了，使用该内核的设计者必须按照这个进行各自芯片的存储器结构设计。这就可以去了解STM32的存储器结构，以及为什么这样设计STM32存储器的结构了。

Cortex-M4是一个内核，自身定义了一个存储器结构，ST公司按照Cortex-M4的这个存储器定义，设计出了自己的存储器结构；但是ST公司的STM32的存储器结构必须按照Cortex-M4这个定义的存储器结构来进行设计。Cortex-M4内核通过I-Code、D-Code、System总线与STM32内部的Flash、SROM相连接的，该种连接情况直接关系到STM32存储器的结构组织；也就是说，Cortex-M4的存储器结构决定了STM32的存储器结构存储器映射是指把芯片中或芯片外的FLASH，RAM，外设，BOOTBLOCK等进行统一编址。即用地址来表示对象。这个地址绝大多数是由厂家规定好的，用户只能用而不能改。用户只能在挂外部RAM或FLASH的情况下可进行自定义

首先，我们对比一下Cortex-M4存储器结构和STM32存储器结构：

0xFFFFFFFF
Vendor-specific memory
511MB
0xE0100000
0xE00FFFFF
Private peripheral bus
1.0MB
0xE0000000
0xDFFFFFFF
External device
1.0GB
0xA0000000
0x9FFFFFFF
External RAM
1.0GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code
0.5GB
0x00000000
0x43FFFFFF
32MB Bit band alias
0x42000000
0x400FFFFF
1MB Bit band region
0x40000000
0x23FFFFFF
32MB Bit band alias
0x22000000
0x200FFFFF
1MB Bit band region
0x20000000

图中可以很清晰的看到，STM32的存储器结构和Cortex-M4的很相似，不同的是，STM32加入了很多实际的东西，如：Flash、SRAM等。只有加入了这些东西，才能成为一个拥有实际意义的、可以工作的处理芯片——STM32。

Peripherals：外设的存储器映射，对该区域操作，就是对相应的外设进行操作；

SRAM：运行时临时存放代码的地方；

Flash：存放代码的地方；

System Memory：STM32出厂时自带的你只能使用，不能写或擦除；

Option Bytes：可以按照用户的需要进行配置（如配置看门狗为硬件实现还是软件实现）；

对STM32存储器知识的掌握，实际上就是对Flash和SRAM这两个区域知识的掌握。因此，下面将重点描述Flash和SRAM的知识。

4.1.1 STM32的SRAM

SRAM的理解比较简单，其作用是用来存取各种动态的输入输出数据、中间计算结果以及与外部存储器交换的数据和暂存数据。设备断电后，SRAM中存储的数据就会丢失。

不同类型的STM32单片机的SRAM大小是不一样的，但是他们的起始地址都是0x2000 0000，终止地址都是0x2000 0000+其固定的容量大小。

STM32F407带有4 KB 备份SRAM和192 KB 系统SRAM。

系统SRAM 可按字节、半字（16 位）或全字（32 位）访问。读写操作以CPU 速度执行，且等待周期为0。系统SRAM 分为三个块：

- 映射在地址0x2000 0000 的112 KB 和16 KB 块，可供所有AHB 主控总线访问。
- 映射在地址0x2002 0000 的64 KB 块，可供所有AHB 主控总线访问。AHB 主总线支持并发SRAM 访问（通过以太网或 USB OTG HS）：例如，当CPU 对112 KB 或64 KB SRAM 进行读/写操作时，以太网 MAC 可以同时对16 KB SRAM 进行读/写操作。
- 在地址0x1000 0000 映射的64 KB 块，只能供CPU 通过数据总线访问。

4.1.2 STM32的Flash

STM32的Flash，严格说，应该是Flash模块。Flash 模块可管理CPU 通过AHB I-Code 和D-Code 对Flash 进行的访问。该接口可针对Flash 执行擦除和编程操作，并实施读写保护机制。Flash 接口通过指令预取和缓存机制加速代码执行。

STM32F4 的闪存模块由：主存储器、系统存储器、OPT 区域和选项字节等 4 部分组成，四个组成部分分别在0x0000 0000——0xFFFF FFFF不同的区域，如图所示：

主存储器，该部分用来存放代码和数据常数（如 const 类型的数据）。分为 12 个扇区，前 4个扇区为 16KB 大小，然后扇区 4 是 64KB 大小，扇区 5～11 是 128K 大小，不同容量的 STM32F4，拥有的扇区数不一样，比如我们的STM32F407VGT6，则拥有全部 12 个扇区。 从上图可以看出主存储器的起始地址就是 0X08000000， B0、B1 都接 GND 的时候，就是从 0X08000000 开始运行代码的。

系统存储器，这个主要用来存放 STM32F4 的 bootloader 代码，此代码是出厂的时候就固化在 STM32F4 里面了，专门来给主存储器下载代码的。当 B0 接 V3.3，B1 接 GND 的时候，从该存储器启动（即进入串口下载模式）。

OTP 区域，即一次性可编程区域，共 528 字节，被分成两个部分，前面 512 字节（32 字节为 1 块，分成 16 块），可以用来存储一些用户数据（一次性的，写完一次，永远不可以擦除！！），后面 16 字节，用于锁定对应块。

选项字节，用于配置读保护、BOR 级别、软件/硬件看门狗以及器件处于待机或停止模式下的复位。

块	名称	块基址	大小
主存储器	扇区 0	0x0800 0000 - 0x0800 3FFF	16 KB
	扇区 1	0x0800 4000 - 0x0800 7FFF	16 KB
	扇区 2	0x0800 8000 - 0x0800 BFFF	16 KB
	扇区 3	0x0800 C000 - 0x0800 FFFF	16 KB
	扇区 4	0x0801 0000 - 0x0801 FFFF	64 KB
	扇区 5	0x0802 0000 - 0x0803 FFFF	128 KB
	扇区 6	0x0804 0000 - 0x0805 FFFF	128 KB
	. . .	. . .	. . .
	扇区 11	0x080E 0000 - 0x080F FFFF	128 KB
系统存储器		0x1FFF 0000 - 0x1FFF 77FF	30 KB
OTP 区域		0x1FFF 7800 - 0x1FFF 7A0F	528 字节
选项字节		0x1FFF C000 - 0x1FFF C00F	16 字节

4.2 位带区

玩过51的都知道51里面是有位操作，以一位（BIT）为数据对象的操作，以简单的将P1口的第2位独立操作：

```
P1.2=0;
P1.2=1;
```

这样就把P1口的第三个脚（BIT2）置0置1。

那么在STM32里面怎么实现位操作呢？通常我们通过读－改－写三个步骤来操作，先将需要修改地址的字读到变量中，修改变量相应的bit位，再写回地址中，C语言的操作代码如下：

通过跟51对位操作的对比，我们发现，上面对位操作指令多，并且非原子操作有可能被中断打断，那么怎么来实现类似于单片机代码缩小， 速度更快，效率更高，更安全位操作呢，STM32的位段、位带别名区就为了实现这样的功能。

它的对象可以是SRAM、I/O和外设空间。要实现对这些地方的某一位的操作。它是这样做的：在寻址空间（32位对应的地址空间为 4GB ）的另一地方，取个别名区空间，从这个地址开始处，每一个字（32BIT）对应SRAM或I/O的一位。这样，1MB SRAM 就可以有 32MB 的对应别名区空间，就是1位膨胀到32位（1 BIT 变为1个字节）。我们对这个别名区空间内的某一字操作（置0或置1），就等于它映射的 SRAM 或 I/O 相应的某地址的某一位的操作。

Cortex -M4F 存储器映射包括两个位段区域。这些区域将存储器别名区域中的每个字映射到存储器位段区域中的相应位。在别名区域写入字时，相当于对位段区域的目标位执行读-修改-写操作。

在STM32F4xx 器件中，外设寄存器和SRAM 均映射到一个位段区域，这样可实现单个位段的读写操作。这些操作仅适用于Cortex -M4F 访问，对于其它总线主接口（如DMA）无效。可通过一个映射公式说明别名区域中的每个字与位段区域中各个位之间的对应关系。映射公式为：

```
bit_word_addr= bit_band_base+ (byte_offsetx 32) + (bit_number× 4)
```

其中：— bit_word_addr 代表别名区域中将映射到目标位的字的地址 — bit_band_base 代表别名区域的起始地址 — byte_offset 代表目标位所在位段区域中的字节编号 — bit_number 代表目标位的位位置(0 7) 示例

下例说明如何将SRAM 地址0x20000300 处字节的位2 映射到别名区域：

```
0x22006008 = 0x22000000 + (0x300*32) + (2*4)
```

对地址0x22006008 执行写操作相当于在SRAM 地址0x20000300 处字节的位2 执行读-修改-写操作。

对地址0x22006008 执行读操作将返回SRAM 地址0x20000300 处字节的位2 的值（0x01 表示位置位，0x00 表示位复位）。

4.3 不同启动模式对应的内存映射

存储器采用固定的存储器映射，代码区域起始地址为0x0000 0000，而数据区域起始地址为0x2000 0000。Cortex -M4F CPU 复位后PC的起始地址始终为0x0000 0000,并通过ICode 总线获取复位向量，这意味着只有代码区域。STM32F4xx 微控制器实施一种特殊机制，可将代码区域（如内部SRAM） 进行启动。首先系统上电或复位后，在SYSCLK 的第四个上升沿锁存BOOT 引脚的值。所以用户可以通过设置BOOT1 和BOOT0 引脚来选择需要的启动模式。

在STM32F4xx 中，可通过BOOT1:0（section B.2）引脚选择三种不同的自举模式，如表所示。

自举模式选择引脚		自举模式	自举空间
BOOT1	BOOT0		
x	0	主 Flash	选择主 Flash 作为自举空间
0	1	系统存储器	选择系统存储器作为自举空间
1	1	嵌入式 SRAM	选择嵌入式 SRAM 作为自举空间

BOOT0 为专用引脚，而BOOT1 则与GPIO 引脚共用。一旦完成对BOOT1 的采样，相应GPIO 引脚即进入空闲状态，可用于其它用途。器件退出待机模式时，还会对BOOT 引脚重新采样。因此，当器件处于待机模式时，这些引脚必须保持所需的自举模式配置。这样的启动延迟结束后，CPU 将从地址0x0000 0000 获取栈顶值，然后从始于0x0000 0004 的自举存储器开始执行代码。

选择自举引脚后，应用程序软件可以将某些存储器设定为从代码空间进行访问（这样，可通过ICode 总线而非系统总线执行代码）。这样的修改通过在SYSCFG 控制器中编程SYSCFG 存储器重映射寄存器(SYSCFG_MEMRMP) 来实现。

SYSCFG 存储器重映射寄存器(SYSCFG_MEMRMP) SYSCFG memory remap register 偏移地址：0x00

此寄存器用于对存储器重映射进行配置：

● 使用两个位来配置可在地址0x0000 0000 访问的存储器区域。从而通过软件选择物理重映射，而旁路BOOT 引脚。

自举模式选择引脚		自举模式	自举空间
BOOT1	BOOT0		
x	0	主 Flash	选择主 Flash 作为自举空间
0	1	系统存储器	选择系统存储器作为自举空间
1	1	嵌入式 SRAM	选择嵌入式 SRAM 作为自举空间

● 这两个位的复位值和复位时BOOT 引脚的设置相同。当BOOT 引脚设为10 [(BOOT1,BOOT0) = (1,0)] 从主Flash 中自举时，寄存器值为0x00。当把FSMC 重映射到地址0x0000 0000 时，只有FSMC 的Bank1 的前两个区域（NOR/PSRAM 1和NOR/PSRAM 2）可被重映射到低端地址。在重映射模式下，CPU 可以通过ICode 总 线 （而不是System 总线）访问外部存储器来提高性能。

4.4 端格式

STM32F407 各字节按小端格式在存储器中编码。字中编号最低的字节被视为该字的最低有效字节，而编号最高的字节被视为最高有效字节。

上图所示的是小端存储方式。

第 5 章

启动过程

STM32F4XX 的启动过程，这里的启动过程是指从 CPU 上电复位执行第 1 条指令开始（汇编文件）到进入 C 程序 main()函数入口之间的部分。启动过程相对来说还是比较重要的，理解了这个过程，对于以后分析程序还是有些帮助的，要不每次看到这个启动过程都会跳过，直接去看主程序了。还有就是以后打算学习 RTOS 的话，对于这个过程必须有个了解，因为移植的时候需要用到这个。

对初学者来说，看这个可能有些吃力，不过不要紧，随着自己做过一些简单的应用之后再来看这章，应该会有很多的帮助，由于我们的ELink407是基于 STM32F407VGT6，所以我们这里主要针对 F4 系列的启动过程做一下分析。

5.1 各版本启动文件

这里各个版本的意思是指不同的编译器，不同的 F4 系列对应的启动文件。在这里我们主要分析STM32F407 在MDK和IAR中启动文件

5.1.1 MDK启动文件

打开我们为本期教程提供的工程文件，路径如下： \Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates 在这个文件里面有 ST 官方为各个编译器提供的启动文件。

MDK 带的启动文件

STM32F40xx 对应的如下系列

STM32F405RG，STM32F405VG，STM32F405ZG，STM32F415RG，STM32F415VG，STM32F415ZG,STM32F407VG，STM32F407VE， STM32F407ZG， STM32F407ZE， STM32F407IG， STM32F407IE， STM32F417VG，STM32F417VE，STM32F417ZG，STM32F417ZE，STM32F417IG，STM32F417IE。

5.1.2 IAR 启动文件

我们再来打开 IAR 文件夹里面的文件，对比下 MDK 和 IAR 带的文件，我们可以发现，除了文件大小不一样以外，文件名字是一样的。这个就跟大家说这么多，有兴趣的可以看看为其它编译器提供的启动文件。

5.2 启动代码分析

下面我们详细的分析一下启动文件startup_stm32f40_41xxx.s。说之前还得需要大家看一下 F4 系列的内存映射，记住 SRAM 的首地址和 FLASH 的首地址。

阅读汇编程序必须对汇编语言中的关键字有一定了解。如果你不能从字面意思理解汇编语句，那么可以使用 MDK 的联机帮助来查阅。具体方法如下：

启动 MDK 软件，在 Help 菜单点击 uVision Help 执行后将出现帮助文件窗口

在搜索栏输入你需要查询的单词进行查询即可。

我们先来看一下，启动文件前面的介绍 （固件库版本：V1.5.0）

```
;***********(C) COPYRIGHT 2012 STMicroelectronics ************
;* File Name : startup_stm32f4xx.s
;* Author : MCD Application Team
;* Version : V1.5.0
;* Date : 05-March-2015
;* Description : STM32F4xx devices vector table for MDK-ARM toolchain.
;* This module performs:
;* - Set the initial SP
;* - Set the initial PC == Reset_Handler
;* - Set the vector table entries with theexceptions ISR address
;* - Configure the system clock and the external SRAM mounted on
;* STM324xG-EVAL board to be used as data memory (optional,
;* to be enabled by user)
;* - Branches to __main in the C library (which eventually
;* calls main()).
;* After Reset the CortexM4 processor is in Thread mode,
;* priority is Privileged, and the Stack is set to Main.
;* <<< Use Configuration Wizard in Context Menu >>>
;*******************************************************
;
; Licensed under MCD-ST Liberty SW License Agreement V2, (the "License");
; You may not use this file except in compliance with the License.
; You may obtain a copy of the License at:
;
; http://www.st.com/software_license_agreement_liberty_v2
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS,
; WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
; See the License for the specific language governing permissions and
; limitations under the License.
;
;*******************************************************
; Amount of memory (in bytes) allocated for Stack
; Tailor this value to your application needs
; <h> Stack Configuration
```

```
; <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>
```

启动文件是后缀为.s 的汇编语言文本文件。行首的分号表示此行是注释行。 启动文件主要完成如下工作，即程序执行过程

- 设置堆栈指针 SP = __initial_sp
- 设置 PC 指针 = Reset_Handler
- 配置系统时钟
- 配置外部 SRAM 用于程序变量等数据存储（这是可选的）
- 跳转（Jump）到 C 库中的 __main ，最终会调用（Call）用户程序的 main()函数

CortexM4 处理器复位后，处于线程模式，指令权限是特权级别（最高级别），堆栈设置为主堆栈。

硬件复位之后，CPU 内的时序逻辑电路首先将 0x0800 0000 位置存放的堆栈栈顶地址装入 SP 寄存器。紧接着将 0x0800 0004 位置存放的向量地址装入 PC 程序计数器。CPU 从 PC 寄存器指向的物理地址取出第 1 条指令开始执行程序，也就是开始执行复位中断服务程序Reset_Handler.

复位中断服务程序会调用 SystemInit()函数(C 语言的) 来配置系统时钟、配置 FSMC 总线上的外部SRAM，然后跳转到 C 库中__main 函数。由 C 库中的__main 函数完成用户程序的初始化工作（比如：变量赋初值等），最后由__main 函数调用用户写的 main()函数开始执行 C 程序。

```
Stack_Size EQU 0x00004000
AREA STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem SPACE Stack_Size
__initial_sp
```

MDK 汇编语言的语法和关键字定义可以在 MDK 的连接帮助中找到定义。

EQU 是表示宏定义的伪指令，类似于 C 语言中的#define。伪指令的意思是指这个“指令”并不会生成二进制程序代码，也不会引起变量空间分配。 0x00004000 表示堆栈大小，注意这里是以字节为单位。 开辟一段数据空间可读可写，段名 STACK，按照 8 字节对齐。

ARER 伪指令表示下面将开始定义一个代码段或者数据段。此处是定义数据段。ARER 后面的关键字表示这个段的属性。

STACK ：表示这个段的名字，可以任意命名。

NOINIT：表示此数据段不需要填入初始数据。

READWRITE：表示此段可读可写。

ALIGN=3 ：表示首地址按照 2 的 3 次方对齐，也就是按照 8 字节对齐。

SPACE 这行指令告诉汇编器给 STACK 段分配 0x00004000 字节的连续内存空间。

__initial_sp 只是一个标号。这里解释一下什么是标号，标号主要用于表示一片内存空间的某个位置，等价于 C 语言中的“地址”概念。地址仅仅表示存储空间的一个位置，从 C 语言的角度来看，变量的地址，数组的地址或是函数的入口地址在本质上并无区别。

__initial_sp 紧接着 SPACE 语句放置，表示了栈空间顶地址。M4 堆栈是由高地址空间向低地址空间增长的。压栈（PUSH）时，堆栈指针 SP 递减。栈（POP）时，SP 递增。栈（STACK）用于存储局部变量、保存函数返回地址。

```
; <h> Heap Configuration
; <o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>
Heap_Size EQU 0x00000400
AREA HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem SPACE Heap_Size
__heap_limit
```

这几行语句和上面的一段代码类似。分配一片连续的内存空间给名字叫 HEAP 的段，也就是分配堆空间。堆的大小为 0x00000400。堆的首地址是 8 字节对齐。堆主要用于动态内存分配，也就是说用 malloc 函数分配的空间位于堆空间。

__heap_base 表示堆的开始地址。

__heap_limit 表示堆的结束地址。

PRESERVE8

THUMB

PRESERVE8 指定当前文件保持堆栈八字节对齐。

THUMB 表示后面的指令是 THUMB 指令集 （CM4 采用的是 16 位 THUMB 指令集，这是相对于ARM7，ARM9，ARM11 的 32 位的 ARM 指令集而言的）

```
; Vector Table Mapped to Address 0 at Reset
AREA RESET, DATA, READONLY
EXPORT __Vectors
EXPORT __Vectors_End
EXPORT __Vectors_Size
```

AREA 定义一块代码段，只读，段名字是 RESET。READONLY 表示只读，缺省就表示代码段了。 3 行 EXPORT 语句将 3 个标号申明为可被外部引用，主要提供给连接器用于连接库文件或其他其他文件。

```
__Vectors DCD __initial_sp ; Top of Stack
DCD Reset_Handler ; Reset Handler
DCD NMI_Handler ; NMI Handler
DCD HardFault_Handler ; Hard FaultHandler
DCD MemManage_Handler ; MPU Fault Handler
DCD BusFault_Handler ; Bus Fault Handler
DCD UsageFault_Handler ; Usage Fault Handler
… 中间的代码已省略 …
DCD DCMI_IRQHandler ; DCMI
DCD CRYP_IRQHandler ; CRYP crypto
DCD HASH_RNG_IRQHandler ; Hash and Rng
DCD FPU_IRQHandler ; FPU
__Vectors_End
__Vectors_Size EQU __Vectors_End - __Vectors
```

上面的这段代码是建立中断向量表，中断向量表定位在代码段的最前面。具体的物理地址由连接器的配置参数（IROM1 的地址）决定。如果程序在 Flash 运行，则中断向量表的起始地址是 0x08000000。 DCD 表示分配 1 个 4 字节的空间。每行 DCD 都会生成一个 4 字节的二进制代码。中断向量表存放的实际上是中断服务程序的入口地址。当异常（也即是中断事件）发生时，CPU 的中断系统会将相应的入口地址赋值给 PC 程序计数器，之后就开始执行中断服务程序。

```
AREA |.text|, CODE, READONLY
; Reset handler
Reset_Handler PROC
EXPORT Reset_Handler [WEAK]
IMPORT SystemInit
IMPORT __main
LDR R0, =SystemInit
BLX R0
LDR R0, =__main
BX R0
ENDP
… 下面是其他的中断服务程序 …
; Dummy Exception Handlers (infinite loops which can be modified)
NMI_Handler PROC
EXPORT NMI_Handler [WEAK]
B . <------ 死循环，用户可以自己编写中断服务程
ENDP
HardFault_Handler\
PROC
EXPORT HardFault_Handler [WEAK]
B .
ENDP
… 中间的代码已省略 …
Default_Handler PROC  <------ 缺省的中断服务程序（开始）
EXPORT WWDG_IRQHandler [WEAK]
EXPORT PVD_IRQHandler [WEAK]
… 中间的代码已省略 …
CRYP_IRQHandler
HASH_RNG_IRQHandler
FPU_IRQHandler
B . <------ 死循环
ENDP  <------ 缺省的中断服务程序(结束)
```

上面的程序就是用汇编代码实现的中断服务程序，下面还有其它中断的中断服务程序，这里就不说了，重点把这个复位中断服务程序说一下。

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

WEAK 声明其他的同名标号优先于该标号被引用,就是说如果外面声明了的话会调用外面的。这个申明很重要，它让我们可以在 C 文件中任意地方放置中断服务程序，只要保证 C 函数的名字和向量表中的名字一致即可。

IMPORT：伪指令用于通知编译器要使用的标号在其他的源文件中定义。但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

SystemInit 函数在文件 system_stm32f4xx.c 里面，这个文件在时钟系统有详细讲解。

这里重点说明一下__main 标号，__main 标号并不表示 C 程序中的 main 函数入口地址， 因此 LDR R0，=_main 也并不是跳转至 main 函数开始执行 C 程序。__main 标号表示 C/C++标准实时库函数里的一个初始化子程序__main 的入口地址。该程序的一个主要作用是初始化堆栈（跳转__user_initial_stackheap 标号进行初始化堆栈的，下面会讲到这个标号），并初始化映像文件，最后跳转到 C 程序中的 main 函数。这就解释了为何所有的 C 程序必须有一个 main 函数作为程序的起点。因为这是由 C/C++标准实时库所规，并且不能更改。 下面我们来看看启动文件最后一段代码，选择用户自己初始化堆和栈。

```
ALIGN
;*********************************
; User Stack and Heap initialization
;*********************************
IF :DEF:__MICROLIB
EXPORT __initial_sp
EXPORT __heap_base
EXPORT __heap_limit
ELSE
IMPORT __use_two_region_memory
EXPORT __user_initial_stackheap
__user_initial_stackheap
LDR R0, = Heap_Mem
LDR R1, =(Stack_Mem + Stack_Size)
LDR R2, = (Heap_Mem + Heap_Size)
LDR R3, = Stack_Mem
BX LR
ALIGN
ENDIF
END
;********* (C) COPYRIGHT STMicroelectronics *****END OF FILE*****
```

上面就是一个简单的汇编语言实现 IF…….ELSE……语句。如果定义了 MICROLIB，那么程序是不会 ELSE 分支的代码。 __MICROLIB 可能大家并不陌生，就在 MDK 的 Target Option 里面设置。

第 6 章

时钟系统

众所周知，时钟系统是 CPU 的脉搏，就像人的心跳一样。所以时钟系统的重要性就不言而喻了。 STM32F4 的时钟系统比较复杂，不像简单的 51 单片机一个系统时钟就可以解决一切。于是有人要问，采用一个系统时钟不是很简单吗？为什么 STM32 要有多个时钟源呢？ 因为首先 STM32 本身非常复杂，外设非常的多，但是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 k 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。

6.1 STM32F407 时钟系统

首先让我们来看看 STM32F407 的时钟系统图：

在 STM32F4 中，有 5 个最重要的时钟源，为 HSI、HSE、LSI、LSE、PLL。其中 PLL 实际是分为两个时钟源，分别为主 PLL 和专用 PLL。 从时钟频率来分可以分为高速时钟源和低速时钟源，在这 5 个中 HSI，HSE 以及 PLL 是高速时钟，LSI 和 LSE 是低速时钟。从来源可分为外部时钟源和内部时钟源，外部时钟源就是从外部通过接晶振的方式获取时钟源，其中 HSE 和LSE 是外部时钟源，其他的是内部时钟源,我们按图中红圈的顺序讲解。

1、LSI 是低速内部时钟， RC 振荡器，频率为 32kHz 左右，供独立看门狗和自动唤醒单元使用。

2、LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体。这个主要是 RTC 的时钟源。

3、HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz～26MHz。ELink407接的是 8M 的晶振。HSE 也可以直接做为系统时钟或者 PLL 输入。

4、HSI 是高速内部时钟，RC 振荡器，频率为 16MHz。可以直接作为系统时钟或者用作 PLL输入。

5、PLL 为锁相环倍频输出。STM32F4 有两个 PLL：

1） 主 PLL(PLL)由 HSE 或者 HSI 提供时钟信号，并具有两个不同的输出时钟。

- 第一个输出 PLLP 用于生成高速的系统时钟（最高 168MHz）
- 第二个输出 PLLQ 用于生成 USB OTG FS 的时钟（48MHz），随机数发生器的时钟和 SDIO时钟。

2）专用 PLL(PLLI2S)用于生成精确时钟，从而在 I2S 接口实现高品质音频性能。

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

从图可以看出，SYSCLK 系统时钟来源有三个方面： HSI,HSE 和 PLL。在我们实际应用中，因为对时钟速度要求都比较高所以一般情况下，都是采用 PLL (PLLCLK)作为 SYSCLK 时钟源。HSI或者HSE输入到一个分频系数为 M 的分频器，然后经过倍频系数为 N 的倍频器出来之后的时候还需要经过一个分频系数为 P（第一个输出 PLLP）或者 Q（第二个输出 PLLQ）的分频器分频之后，最后才生成最终的主 PLL 时钟。

例如我们的外部晶振选择 8MHz，设置HSE 为 PLL 时钟源。同时我们设置相应的分频器 M=8，倍频器倍频系数 N=336，分频器分频系数 P=2，那么主 PLL 生成的第一个输出高速时钟 PLLP 为：

```
PLL=8MHz * N/ (M*P)=8MHz* 336 /(8*2) = 168MHz
```

如果设置的 SYSCLK 时钟源为 PLL，那么 SYSCLK 时钟为 168MHz。后面的实验都是采用这样的配置。

设置了内核的SYSCLK 时钟源，我们再看各个外设以及系统提供时钟，这里我们选择一些比较常用的时钟知识来讲解。从上到下的顺序进行说明。

看门狗时钟源。从图中可以看出，看门狗时钟源只能是低速的 LSI 时钟。

RTC 时钟源，从图上可以看出，RTC 的时钟源可以选择 LSI，LSE，以及HSE 分频后的时钟，HSE 分频系数为 2～31。

STM32F4 输出时钟 MCO1 和 MCO2。 MCO1 是向芯片的 PA8 引脚输出时钟。它有四个时钟来源分别为：HSI,LSE,HSE 和 PLL 时钟。MCO2 是向芯片的PC9 输出时钟，它同样有四个时钟来源分别为： HSE,PLL， SYSCLK 以及 PLLI2S时钟。MCO 输出时钟频率最大不超过 100MHz。

以太网 PTP 时钟， AHB 时钟， APB2 高速时钟， APB1 低速时钟。这些时钟都是来源于 SYSCLK 系统时钟。其中以太网 PTP 时钟是使用系统时钟。AHB,APB2 和 APB1 时钟是经过 SYSCLK 时钟分频得来。这里大家记住，AHB最大时钟为 168MHz，APB2 高速时钟最大频率为 84MHz,而 APB1 低速时钟最大频率为 42MHz。

I2S 时钟源。从图可以看出，I2S 的时钟源来源于 PLLI2S 或者映射到 I2S_CKIN 引脚的外部时钟。 I2S 出于音质的考虑，对时钟精度要求很高。

STM32F4 内部以太网 MAC 时钟的来源。对于 MII 接口来说，必须向外部PHY 芯片提供 25Mhz 的时钟，这个时钟，可以由 PHY 芯片外接晶振，或者使用STM32F4 的 MCO 输出来提供。然后，PHY 芯片再给 STM32F4 提供ETH_MII_TX_CLK 和 ETH_MII_RX_CLK 时钟。对于 RMII 接口来说，外部必须提供 50Mhz 的时钟驱动 PHY 和 STM32F4 的 ETH_RMII_REF_CLK，这个 50Mhz时钟可以来自 PHY、有源晶振或者 STM32F4 的 MCO。我们的开发板使用的是RMII 接口，使用 PHY 芯片提供 50Mhz 时钟驱动 STM32F4 的 ETH_RMII_REF_CLK。

外部 PHY 提供的 USB OTG HS (60MHZ) 时钟。这里还需要说明一下，Cortex 系统定时器 Systick 的时钟源可以是 AHB 时钟 HCLK 或HCLK 的 8 分频。

在以上的时钟输出中，有很多是带使能控制的，例如 AHB 总线时钟、内核时钟、各种 APB1外设、APB2 外设等等。当需要使用某模块时，记得一定要先使能对应的时钟。后面我们讲解实例的时候会讲解到时钟使能的方法。

6.2 RCC相关库函数说明 (STM32F4 时钟初始化配置)

上一小节我们对 STM32F4 时钟树进行了初步的讲解，接下来我们来讲解一下 STM32F4 的系统时钟配置。

STM32F4 时钟系统初始化是在 system_stm32f4xx.c 中的 SystemInit()函数中完成的。对于系统时钟关键寄存器设置主要是在 SystemInit 函数中调用 SetSysClock()函数来设置的。我们可以先看看 SystemInit ()函数体:

```
void SystemInit(void)
{
    /* FPU settings */
    #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
  SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access */
    #endif

    /* Reset the RCC clock configuration to the default reset state */
    /* Set HSION bit */
    RCC->CR |= (uint32_t)0x00000001;

    /* Reset CFGR register */
    RCC->CFGR = 0x00000000;

    /* Reset HSEON, CSSON and PLLON bits */
    RCC->CR &= (uint32_t)0xFEF6FFFF;

    /* Reset PLLCFGR register */
    RCC->PLLCFGR = 0x24003010;

    /* Reset HSEBYP bit */
    RCC->CR &= (uint32_t)0xFFFBFFFF;

    /* Disable all interrupts */
    RCC->CIR = 0x00000000;

    #if defined (DATA_IN_ExtSRAM) || defined (DATA_IN_ExtSDRAM)
    SystemInit_ExtMemCtl();
    #endif /* DATA_IN_ExtSRAM || DATA_IN_ExtSDRAM */

    /* Configure the System clock source, PLL Multiplier and Divider factors,
    AHB/APBx prescalers and Flash settings */
    SetSysClock();

    /* Configure the Vector Table location add offset address */
    #ifdef VECT_TAB_SRAM
    /* Vector Table Relocation in Internal SRAM */
    SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET;
    #else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in
    Internal FLASH */
    #endif
```

```
}
```

SystemInit 函数开始先进行浮点运算单元设置，然后是复位 PLLCFGR,CFGR 寄存器，同时通过设置 CR 寄存器的 HSI 时钟使能位来打开 HSI 时钟。默认情况下如果 CFGR 寄存器复位，那么是选择 HSI 作为系统时钟，这点大家可以查看 RCC->CFGR 寄存器的位描述最低 2 位可以得知，当低两位配置为 00 的时候（复位之后），会选择 HSI 振荡器为系统时钟。也就是说，调用 SystemInit 函数之后，首先是选择 HSI 作为系统时钟。下面是 RCC->CFGR 寄存器配置描述：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2		MCO2 PRE[2:0]			MCO1 PRE[2:0]			I2SSCR	MCO1		RTCPRE[4:0]				
rw		rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PPRE2[2:0]			PPRE1[2:0]			Reserved		HPRE[3:0]				SWS1	SWS0	SW1	SW0
rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

位31:30 MCO21:0 (section B.2)：微控制器时钟输出2 (Microcontroller clock output 2)由软件置1 和清零。时钟源选择可能会造成对MCO2 的干扰。强烈建议仅在复位后但在使能外部振荡器和PLL 之前来配置这些位。

00：选择系统时钟(SYSCLK) 输出到MCO2 引脚

01:选择PLLI2S 时钟输出到MCO2 引脚

10：选择HSE 振荡器时钟输出到MCO2 引脚

11：选择PLL 时钟输出到MCO2 引脚

位27:29 MCO2PRE：MCO2 预分频器(MCO2 prescaler)由软件置1 和清零，用于配置MCO2 的预分频器。对此预分频器进行修改可能会对 M C O 2 造成干扰。强烈建议仅在复位后且在使能外部振荡器和PLL 之前进行此分频器的更改。

0xx：无分频

100：2 分频

101：3 分频

110：4 分频

111：5 分频

位24:26 MCO1PRE：MCO1 预分频器(MCO1 prescaler)由软件置1 和清零，用于配置MCO1 的预分频器。对此预分频器进行修改可能会对 M C O 1 造成干扰。强烈建议仅在复位后且在使能外部振荡器和PLL 之前进行此分频器的更改。

0xx：无分频

100：2 分频

101：3 分频

110：4 分频

111：5 分频

位23 I2SSRC：I2S 时钟选择(I2S clock selection)由软件置1 和清零。通过此位可在PLLI2S 时钟和外部时钟之间选择I2S 时钟源。强烈建议仅在复位之后使能I2S 模块之前对此位进行更改。

0：PLLI2S 时钟用作I2S 时钟源

1：在I2S_CKIN 引脚上映射的外部时钟用作I2S 时钟源

位22:21 MCO1：微控制器时钟输出1 (Microcontroller clock output 1)由软件置1 和清零。时钟源选择可能会造成对MCO1 的干扰。强烈建议仅在复位后且在使能 外部振荡器和PLL 之前来配置这些位。 * 00：选择HSI 时钟输出到MCO1 引脚
01：选择LSE 振荡器输出到MCO1 引脚
10：选择HSE 振荡器时钟输出到MCO1 引脚
11：选择PLL 时钟输出到MCO1 引脚

位20:16 RTCPRE：适用于RTC 时钟的HSE 分频系数由软件置1 和清零，用于对HSE 时钟输入时钟进行分频，进而为RTC 生成1 MHz 的时钟。

00000：无时钟

00001：无时钟

00010：HSE/2

00011：HSE/3

00100：HSE/4

…

11110：HSE/30

11111：HSE/31

位15:13 PPRE2：APB 高速预分频器(APB2) (APB high-speed prescaler (APB2))由软件置位和清零，用于控制APB 高速时钟分频系数。

0xx：AHB 时钟不分频

100：AHB 时钟2 分频

101：AHB 时钟4 分频

110：AHB 时钟8 分频

111：AHB 时钟16 分频

位12:10 PPRE1：APB 低速预分频器(APB1) (APB Low speed prescaler (APB1))由软件置位和清零，用于控制APB 低速时钟分频系数。

0xx：AHB 时钟不分频

100：AHB 时钟2 分频

101：AHB 时钟4 分频

110：AHB 时钟8 分频

111：AHB 时钟16 分频

位9:8 保留，必须保持复位值。

位7:4 HPRE：AHB 预分频器(AHB prescaler)由软件置位和清零，用于控制AHB 时钟分频系数。

0xxx：系统时钟不分频

1000：系统时钟2 分频

1001：系统时钟4 分频

1010：系统时钟8 分频

1011：系统时钟16 分频

1100：系统时钟64 分频

1101：系统时钟128 分频

1110：系统时钟256 分频

1111：系统时钟512 分频

位3:2 SWS：系统时钟切换状态(System clock switch status)由硬件置1 和清零，用于指示用作系统时钟的时钟源。

00：HSI 振荡器用作系统时钟

01：HSE 振荡器用作系统时钟

10：PLL 用作系统时钟

11：不适用

位1:0 SW：系统时钟切换(System clock switch)由软件置1 和清零，用于选择系统时钟源。由硬件置1，用于在退出停机或待机模式时或者在直接或间接用作系统时钟的HSE 振荡器发生故障时强制HSI 的选择。 * 00：选择HSI 振荡器作为系统时钟
01：选择HSE 振荡器作为系统时钟
10：选择PLL 作为系统时钟
11：不允许

在设置完相关寄存器后，接下来 SystemInit 函数内部会调用 SetSysClock 函数。这个函数比较长，我们就把函数一些关键代码行截取出来给大家讲解一下。这里我们省略一些宏定义标识符值的判断而直接把针对 STM32F407 比较重要的内容贴出来：

```
static void SetSysClock(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;

    /*使能 HSE*/
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);
```

```
/*  等待 HSE 稳定*/
do
{
    HSEStatus = RCC->CR & RCC_CR_HSERDY;
    StartUpCounter++;
} while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));

if ((RCC->CR & RCC_CR_HSERDY) != RESET)
{
    HSEStatus = (uint32_t)0x01;
}
else
{
    HSEStatus = (uint32_t)0x00;
}

if (HSEStatus == (uint32_t)0x01)
{
    /* Select regulator voltage output Scale 1 mode */
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    PWR->CR |= PWR_CR_VOS;

    /* HCLK = SYSCLK / 1*/
    RCC->CFGR |= RCC_CFGR_HPRE_DIV1;

    /* PCLK2 = HCLK / 2*/
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;

    /* PCLK1 = HCLK / 4*/
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;

    /* PCLK2 = HCLK / 2*/
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV1;

    /* PCLK1 = HCLK / 4*/
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV2;

    /* Configure the main PLL */
    RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
    (RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);

    /*  使能主 PLL*/
    RCC->CR |= RCC_CR_PLLON;

    /*  等待主 PLL 就绪  */
```

```
        while((RCC->CR & RCC_CR_PLLRDY) == 0)
        {
        }

    /* Configure Flash prefetch, Instruction cache, Data cache and wait state */
        FLASH->ACR = FLASH_ACR_PRFTEN | FLASH_ACR_ICEN
        |FLASH_ACR_DCEN |FLASH_ACR_LATENCY_5WS;

        /*  设置主 PLL 时钟为系统时钟源  */
        RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
        RCC->CFGR |= RCC_CFGR_SW_PLL;

        /*  等待设置稳定（主 PLL 作为系统时钟源）*/
        while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS ) != RCC_CFGR_SWS_PLL);
        {
        }

    }
    else
    {
        /* If HSE fails to start-up, the application will have wrong clock
       configuration. User can add here some code to deal with this error */
    }
}
```

这段代码的大致流程是这样的：先使能外部时钟 HSE ，等待 HSE 稳定之后，配置AHB，APB1，APB2 时钟相关的分频因子，也就是相关外设的时钟。等待这些都配置完成之后，打开主 PLL 时钟，然后设置主 PLL 作为系统时钟 SYSCLK 时钟源。如果 HSE 不能达到就绪状态（比如外部晶振不能稳定或者没有外部晶振），那么依然会是 HSI 作为系统时钟。在这里要特别提出来，在设置主 PLL 时钟的时候，会要设置一系列的分频系数和倍频系数参数。大家可以从 SetSysClock 函数的这行代码看出：

```
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
(RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);
```

6.3 修改系统时钟的方法

这些参数是通过宏定义标识符的值来设置的。默认的配置在 System_stm32f4xx.c 文件开头的地方配置。对于我们开发板，我们的设置参数值如下：

```
#define PLL_M      8
#define PLL_Q      7
#define PLL_N      336
#define PLL_P      2
```

所以我们的主 PLL 时钟为：

```
PLL=8MHz * N/ (M*P)=8MHz* 336 /(8*2) = 168MHz
```

在开发过程中，我们可以通过调整这些值来设置我们的系统时钟。这里还有个特别需要注意的地方，就是我们还要同步修改 stm32f4xx.h 中宏定义标识符HSE_VALUE 的值为我们的外部时钟：

```
#if !defined   (HSE_VALUE)

/*!< Value of the External oscillator in Hz */
#define HSE_VALUE     ((uint32_t)8000000)

#endif /* HSE_VALUE */
```

这里默认固件库配置的是 25000000，我们外部时钟为 8MHz，所以我们根据我们硬件情况修改为 8000000 即可。

讲到这里，大家对 SystemInit 函数的流程会有个比较清晰的理解。那么 SystemInit 函数是怎么被系统调用的呢？SystemInit 是整个设置系统时钟的入口函数。这个函数对于我们使用 ST提供的 STM32F4 固件库的话，会在系统启动之后先执行 main 函数，然后再接着执行 SystemInit函数实现系统相关时钟的设置。这个过程设置是在启动文件 startup_stm32f40_41xxx.s 中间设置的，我们接下来看看启动文件中这段启动代码：

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

这段代码的作用是在系统复位之后引导进入 main 函数，同时在进入 main 函数之前，首先要调用 SystemInit 系统初始化函数完成系统时钟等相关配置。

最后我们总结一下 SystemInit()函数中设置的系统时钟大小：

```
SYSCLK（系统时钟）                =168MHz
AHB 总线时钟(HCLK=SYSCLK)         =168MHz
APB1 总线时钟(PCLK1=SYSCLK/4)     =42MHz
APB2 总线时钟(PCLK2=SYSCLK/2)     =84MHz
PLL 主时钟                        =168MHz
```

6.4 RCC相关库函数说明 (STM32F4 时钟使能和配置)

上小节我们讲解了系统复位之后调用 SystemInit 函数之后相关时钟的默认配置。如果在系统初始化之后，我们还需要修改某些时钟源配置，或者我们要使能相关外设的时钟该怎么设置呢？这些设置实际是在 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《STM32F4 中文参考手册》 6.3 小节查看所有 RCC 相关寄存器的配置。所以这里我们不直接讲解寄存器配置，而是通过 STM32F4 标准固件库配置方法给大家讲解。

在 STM32F4 标准固件库里，时钟源的选择以及时钟使能等函数都是在 RCC 相关固件库文件 stm32f4xx_rcc.h 和 stm32f4xx_rcc.c 中声明和定义的。大家打开 stm32f4xx_rcc.h 文件可以看到文件开头有很多宏定义标识符，然后是一系列时钟配置和时钟使能函数申明。这些函数大致可以归结为三类，一类是外设时钟使能函数，一类是时钟源和分频因子配置函数，还有一类是外设复位函数。当然还有几个获取时钟源配置的函数。下面我们以几种常见的操作来简要介绍一下这些库函数的使用。 首先是时钟使能函数。时钟使能相关函数包括外设设置使能和时钟源使能两类。首先我们来看看外设时钟使能相关的函数：

```
void   RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
void   RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
void   RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
void   RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
void   RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

这里主要有 5 个外设时钟使能函数。5 个函数分别用来使能 5 个总线下面挂载的外设时钟，这些总线分别为：AHB1 总线，AHB2 总线，AHB3 总线，APB1 总线以及 APB2 总线。要使能某个外设，调用对应的总线外设时钟使能函数即可。这里我们要特别说明一下，STM32F4 的外设在使用之前，必须对时钟进行使能，如果没有使能时钟，那么外设是无法正常工作的。对于哪个外设是挂载在哪个总线之下，虽然我们也可以查手册查询到，但是这里如果大家使用的是库函数的话，实际上是没有必要去查询手册的，这里我们给大家介绍一个小技巧。

比如我们要使能 GPIOA,我们只需要在 stm32f4xx_rcc.h 头文件里面搜索 GPIOA，就可以搜索到对应的时钟使能函数的第一个入口参数为 RCC_AHB1Periph_GPIOA，从这个宏定义标识符一眼就可以看出，GPIOA 是挂载在 AHB1 下面。同理，对于串口 1 我们可以搜索 USART1，找到标识符为 RCC_APB2Periph_USART1，那么很容易知道串口 1 是挂载在 APB2 之下。

如果我们要使能 GPIOA，那么我们可以在头文件 stm32f4xx_rcc.h 里面查看到宏定义标识符 RCC_AHB1Periph_GPIOA，顾名思义 GPIOA 是挂载在 AHB1 总线之下，所以，我们调用AHB1 总线下外设时钟使能函数 RCC_AHB1PeriphClockCmd 即可。具体调用方式入如下：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);//使能 GPIOA 时钟
```

同理，如果我们要使能串口 1 的时钟，那么我们调用的函数为：

```
void  RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
```

具体的调用方法是：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);
```

还有一类时钟使能函数是时钟源使能函数，前面我们已经讲解过 STM32F4 有 5 大类时钟源。这里我们列出来几种重要的时钟源使能函数：

```
void   RCC_HSICmd(FunctionalState NewState);
void   RCC_LSICmd(FunctionalState NewState);
void   RCC_PLLCmd(FunctionalState NewState);
void   RCC_PLLI2SCmd(FunctionalState NewState);
void   RCC_PLLSAICmd(FunctionalState NewState);
void   RCC_RTCCLKCmd(FunctionalState NewState);
```

这些函数是用来使能相应的时钟源。比如我们要使能 PLL 时钟，那么调用的函数为：

```
void   RCC_PLLCmd(FunctionalState NewState);
```

具体调用方法如下：

```
RCC_PLLCmd(ENABLE);
```

我们要使能相应的时钟源，调用对应的函数即可。

接下来我们要讲解的是第二类时钟功能函数：时钟源选择和分频因子配置函数。这些函数是用来选择相应的时钟源以及配置相应的时钟分频系数。比如我们之前讲解过系统时钟 SYSCLK，我们可以选择 HSI,HSE 以及 PLL 三个中的一个时钟源为系统时钟。那么到底选择哪一个，这是可以配置的。下面我们列举几种时钟源配置函数：

```
void       RCC_LSEConfig(uint8_t RCC_LSE);
void       RCC_SYSCLKConfig(uint32_t RCC_SYSCLKSource);
void       RCC_HCLKConfig(uint32_t RCC_SYSCLK);
void       RCC_PCLK1Config(uint32_t RCC_HCLK);
void       RCC_PCLK2Config(uint32_t RCC_HCLK);
void       RCC_RTCCLKConfig(uint32_t RCC_RTCCLKSource);
void       RCC_PLLConfig(uint32_t RCC_PLLSource, uint32_t PLLM,
uint32_t PLLN, uint32_t PLLP, uint32_t PLLQ);
```

比如我们要设置系统时钟源为 HSI，那么我们可以调用系统时钟源配置函数：

```
void   RCC_HCLKConfig(uint32_t RCC_SYSCLK);
```

具体配置方法如下：

```
RCC_HCLKConfig(RCC_SYSCLKSource_HSI);//配置时钟源为 HSI
```

又如我们要设置 APB1 总线时钟为 HCLK 的 2 分频，也就是设置分频因子为 2 分频，那么如果我们要使能 HSI，那么调用的函数为：

```
void    RCC_PCLK1Config(uint32_t RCC_HCLK);
```

具体配置方法如下：

```
RCC_PCLK1Config(RCC_HCLK_Div2);
```

接下来我们看看第三类外设复位函数。如下：

```
void   RCC_AHB1PeriphResetCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
void   RCC_AHB2PeriphResetCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
void   RCC_AHB3PeriphResetCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
void   RCC_APB1PeriphResetCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
void   RCC_APB2PeriphResetCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

这类函数跟前面讲解的外设时钟函数使用方法基本一致，不同的是一个是用来使能外设时钟，一个是用来复位对应的外设。这里大家在调用函数的时候一定不要混淆。对于这些时钟操作函数，我们就不一一列举出来，大家可以打开 RCC 对应的文件仔细了解。

第 7 章

LED灯实验

我们常常通过观察设备电源指示灯（LED灯）状态来判断设备是否加电或者设备供电是否正常，通过这一直观的现象我们很容易获得计算机运行的输出信息，这是一种非常简单也是常用的人机交互的方式，本次实验我们就需要实现计算机控制LED灯的交替闪烁。

7.1 实验原理与基础知识

LED是发光二极管的简称，它是半导体二极管的一种，可以把电能转化成光能。发光二极管与普通二极d管一样是由一个PN结组成，也具有单向导电性。当给发光二极管加上正向电压后，从P区注入到N区的空穴和由N区注入到P区的电子，在PN结附近数微米内分别与N区的电子和P区的空穴复合，产生自发辐射的荧光。不同的半导体材料中电子和空穴所处的能量状态不同。当电子和空穴复合时释放出的能量多少不同，释放出的能量越多，则发出的光的波长越短。波长不一样那么发光颜色就不一样。

那么我们这个LED灯的实验，我们怎么通过编码（或者说指令）控制LED灯的状态呢？我们可以先看一下我们LED部分的原理图

我们可以看到LED1，LED2 的P端接电源的正极，N端分别接在我们的STM32F407VGT6芯片的PA8，PB8管脚上，根据LED的发光原理，如果管脚上为低电平，PN两端产生压差，电子移动，发生电子从低轨道到高轨道的跃迁，释放能量，LED发光。如果管脚上为高电平，PN结两端没有压差，没有电子跃迁，LED熄灭。

所以，对LED灯的控制就变成了，对PA8，PB8管脚输出电平的控制了，那么在STM32F407VGT6芯片上，如果控制管脚的呢？下面我们开启STM32F407 GPIO的学习。

所谓GPIO就是通用输入输出管脚（General Purpose Input Output），是芯片与外部设备进行通信的基本方式，打开芯片手册文档或我们的原理图，我们可以看到STM32F407VGT6总共有6组GPIO（不同的芯片型号GPIO管脚数量可能不一样，组数也不一样）分别是PA，PB，PC，PD，PE，PH 。PH组只有两个管脚，其他组有16个管脚，STM32F4 每组通用 I/O 端口包括 4 个 32 位配置寄存器（MODER、 OTYPER、 OSPEEDR 和 PUPDR）、 2 个 32 位数据寄存器（IDR ODR）、 2 个 32 位复用功能选择寄存器（AFRH 和 AFRL）和 1 个 32 位置位/复位寄存器（BSRR）、1 个 32 位锁定寄存器（LCKR）。这样， STM32F4 每组 IO 有 10 个 32 位寄存器控制，其中常用的有 4 个配置寄存器+2 个数据寄存器+2 个复用功能选择寄存器，共 8 个。

接下来我们详细介绍 这8个常用的寄存器： MODER、 OTYPER、 OSPEEDR、 PUPDR、 ODR、IDR 、 AFRH 和 AFRL。介绍完后我们就可以通过对这些寄存器的操作，来编写LED的驱动代码。先使用自己操作寄存器（自己的代码操作寄存器）的方法来配置，稍后我们讲解对应的库函数配置方法。（section B.2）

P型電洞
N型電子
+
−
elecChinaCBE.com

MODER 寄存器

该寄存器是 GPIO 端口模式控制寄存器，用于控制 GPIOx（STM32F4 最多有 9 组 IO，分别用大写字母表示，即x=A/B/C/D/E/F/G/H/I，下同）寄存器用于选择I/O 方向（输入、输出、AF、模拟），该寄存器各位描述如表所示：

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

位 2y:2y+1 **MODERy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 方向模式。
00：输入（复位状态）
01：通用输出模式
10：复用功能模式
11：模拟模式

该寄存器各位在复位后，一般都是 0 （个别不是 0，比如 JTAG 占用的几个 IO 口），也就是默认条件下一般是输入状态的。每组 IO 下有 16 个 IO 口，该寄存器共 32 位，每 2 个位控制 1个 IO。

OTYPER 寄存器

该寄存器用于控制 GPIOx 的输出类型（推挽或开漏），该寄存器各位描述见表所示：

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

位 31:16 保留，必须保持复位值。

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

该寄存器仅用于输出模式，在输入模式（MODER1:0（section B.2)=00/11 时）下不起作用。该寄存器低 16 位有效，每一个位控制一个 IO 口，复位后，该寄存器值均为 0。

OSPEEDR寄存器

该寄存器用于控制 GPIOx 的输出速度，无论采用哪种I/O 方向，都会直接将I/O 速度引脚连接到相应的GPIOx_OSPEEDR 寄存器位，该寄存器各位描述见表所示：

该寄存器也仅用于输出模式，在输入模式（MODER1:0（section B.2)=00/11 时）下不起作用。该寄存器每 2 个位控制一个 IO 口，复位后，该寄存器值一般为 0。

PUPDR寄存器

该寄存器用于控制 GPIOx 的上拉/下拉，该寄存器各位描述见表所示：

该寄存器每 2 个位控制一个 IO 口，用于设置上下拉，复位后，该寄存器值一般为 0。

ODR 寄存器

该寄存器用于控制 GPIOx 的输出，该寄存器各位描述见表所示：

该寄存器用于设置某个 IO 输出低电平(ODRy=0)还是高电平(ODRy=1)，该寄存器也仅在输出模式下有效，在输入模式（MODER1:0（section B.2)=00/11 时）下不起作用。。

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

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

这些位通过软件写入，用于配置 I/O 输出速度。

00：2 MHz（低速）

01：25 MHz（中速）

10：50 MHz（快速）

11：30 pF 时为 100 MHz（高速）（15 pF 时为 80 MHz 输出（最大速度））

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

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

这些位通过软件写入，用于配置 I/O 上拉或下拉。

00：无上拉或下拉

01：上拉

10：下拉

11：保留

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

位 15:0 **IDRy[15:0]**：端口输入数据 (Port input data) (y = 0..15)

这些位为只读形式，只能在字模式下访问。它们包含相应 I/O 端口的输入值。

IDR 寄存器

该寄存器用于读取 GPIOx 的输入，该寄存器各位描述见表所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 保留，必须保持复位值。

位 15:0 **ODRy[15:0]**：端口输出数据 (Port output data) (y = 0..15)
这些位可通过软件读取和写入。

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

该寄存器用于读取某个 IO 的电平，如果对应的位为 0(IDRy=0)，则说明该 IO 输入的是低电平，如果是 1(IDRy=1)，则表示输入的是高电平。

AFRL 和 AFRH 寄存器

这两个寄存器用来设置 IO 引脚复用和映射的，STM32F4 每个 IO 引脚通过一个复用器连接到板载外设/模块，该复用器一次仅允许一个外设的复用功能（AF）连接到IO 引脚如图 1.1.10，这样，可以确保共用同一个 IO 引脚的外设之间不会发生冲突。引脚复用功能选择，正是通过 AFRL 和 AFRH 来控制的，其中 AFRL 控制0～7 这 8 个 IO 口，AFRH 控制 8～15 这 8 个 IO 口。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **AFRLy**：端口 x 位 y 的复用功能选择 (Alternate function selection for port x bit y) (y = 0..7)
这些位通过软件写入，用于配置复用功能 I/O。

AFRLy 选择：

0000：AF0	1000：AF8
0001：AF1	1001：AF9
0010：AF2	1010：AF10
0011：AF3	1011：AF11
0100：AF4	1100：AF12
0101：AF5	1101：AF13
0110：AF6	1110：AF14
0111：AF7	1111：AF15

AFRL 寄存器各位描述如图所示：

AFRH 寄存器各位描述如图所示：

GPIO的寄存器介绍的差不多了，接下来，我们就要结合我们的开发板LED部分的原理图编写驱动代码了

7.2 电路设计

如图LED部分的电路设计，电路的工作原理我们在上一小节中已经说明，这里不再讲解，在这里再次给出原理图，希望大家注意LED1，LED2对应的GPIO管脚，以便于下面我们设计其驱动代码。

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

位 31:16 **BRy**：端口 x 复位位 y (Port x reset bit y) (y = 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0：不会对相应的 ODRx 位执行任何操作

1：对相应的 ODRx 位进行复位

注意：如果同时对 BSx 和 BRx 置位，则 BSx 的优先级更高。

位 15:0 **BSy**：端口 x 置位位 y (Port x set bit y) (y= 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0：不会对相应的 ODRx 位执行任何操作

1：对相应的 ODRx 位进行置位

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:17 保留，必须保持复位值。

位 16 **LCKK[16]**：锁定键 (Lock key)

可随时读取此位。可使用锁定键写序列对其进行修改。

0：端口配置锁定键未激活。

1：端口配置锁定键已激活。直到 MCU 复位时，才锁定 GPIOx_LCKR 寄存器。

锁定键写序列：

WR LCKR[16] = ‘1’ + LCKR[15:0]

WR LCKR[16] = ‘0’ + LCKR[15:0]

WR LCKR[16] = ‘1’ + LCKR[15:0]

RD LCKR

RD LCKR[16] = ‘1’（此读操作为可选操作，但它可确认锁定已激活）

注意：在锁定键写序列期间，不能更改 LCK[15:0] 的值。

锁定序列中的任何错误都将中止锁定操作。

在任一端口位上的第一个锁定序列之后，对 LCKK 位的任何读访问都将返回“1”，直到下一次 CPU 复位为止。

位 15:0 **LCKy**：端口 x 锁定位 y (Port x lock bit y) (y= 0..15)

这些位都是读/写位，但只能在 LCKK 位等于“0”时执行写操作。

0：端口配置未锁定

1：端口配置已锁定

7.3 不使用标准库，驱动代码设计

通过对GPIO寄存器的了解，我们知道控制一个LED灯闪烁大概需要一下几个步骤，首先是配置管脚作为输出，然后输出低电平，LED导通发光，延时一段时间后，我们再关闭LED，即将相应管脚输出高电平，管脚上高低电平如此反复，LED灯产生闪烁。

双击Kei15 新建工程，在这里我们使用的是直接操作GPIO寄存器，所以不需要拷贝我们第一个工程模版（工程模版我们会在用标准库的操作寄存器的时候使用），点击新建文件按钮，我们开始编写我们的头文件：

```
#define     __IO    volatile

#define GPIOA_BASE    ((uint32_t)0x40020000)
#define GPIOB_BASE    ((uint32_t)0x40020400)

typedef unsigned  int       uint32_t;
typedef unsigned short     int uint16_t;

typedef struct
{
__IO uint32_t MODER;   /*!< GPIO port mode register,           offset: 0x00    */
__IO uint32_t OTYPER;  /*!< GPIO port output type register,      offset: 0x04    */
__IO uint32_t OSPEEDR; /*!< GPIO port output speed register,     offset: 0x08    */
__IO uint32_t PUPDR;   /*!< GPIO port pull-up/pull-down register, offset: 0x0C    */
__IO uint32_t IDR;    /*!< GPIO port input data register,       offset: 0x10    */
__IO uint32_t ODR;    /*!< GPIO port output data register,      offset: 0x14    */
__IO uint16_t BSRRL;   /*!< GPIO port bit set/reset low register, offset: 0x18    */
__IO uint16_t BSRRH;   /*!< GPIO port bit set/reset high register, offset: 0x1A    */
__IO uint32_t LCKR;    /*!< GPIO port configuration lock register, offset: 0x1C    */
__IO uint32_t AFR[2];  /*!< GPIO alternate function registers,    offset: 0x20-0x24 */
} GPIO_TypeDef;

#define GPIOA_Reg            ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB_Reg            ((GPIO_TypeDef *) GPIOB_BASE)

#define RCC_BASE                ((uint32_t)0x40023800)

typedef struct
{
__IO uint32_t CR;         /*!< RCC clock control register,       offset: 0x00 */
__IO uint32_t PLLCFGR;     /*!< RCC PLL configuration register,    offset: 0x04 */
__IO uint32_t CFGR;       /*!< RCC clock configuration register,  offset: 0x08 */
__IO uint32_t CIR;        /*!< RCC clock interrupt register,     offset: 0x0C */
__IO uint32_t AHB1RSTR;    /*!< RCC AHB1 peripheral reset register, offset: 0x10 */
__IO uint32_t AHB2RSTR;    /*!< RCC AHB2 peripheral reset register, offset: 0x14 */
__IO uint32_t AHB3RSTR;    /*!< RCC AHB3 peripheral reset register, offset: 0x18 */
 uint32_t     RESERVED0;    /*!< Reserved, 0x1C                              */
```

```
__IO uint32_t APB1RSTR;     /*!< RCC APB1 peripheral reset register, offset: 0x20 */
__IO uint32_t APB2RSTR;     /*!< RCC APB2 peripheral reset register, offset: 0x24 */
 uint32_t     RESERVED1[2];  /*!< Reserved, 0x28-0x2C                              */
__IO uint32_t AHB1ENR;      /*!< RCC AHB1 peripheral clock register, offset: 0x30 */
__IO uint32_t AHB2ENR;      /*!< RCC AHB2 peripheral clock register, offset: 0x34 */
__IO uint32_t AHB3ENR;      /*!< RCC AHB3 peripheral clock register, offset: 0x38 */
 uint32_t     RESERVED2;     /*!< Reserved, 0x3C                                   */

  /*!< RCC APB1 peripheral clock enable register, offset: 0x40 */
  __IO uint32_t APB1ENR;

  /*!< RCC APB2 peripheral clock enable register, offset: 0x44 */
  __IO uint32_t APB2ENR;

 uint32_t     RESERVED3[2];  /*!< Reserved, 0x48-0x4C                              */

  /*!< RCC AHB1 peripheral low power mode register, offset: 0x50 */
  __IO uint32_t AHB1LPENR;

  /*!< RCC AHB2 peripheral low power mode register, offset: 0x54 */
  __IO uint32_t AHB2LPENR;

  /*!< RCC AHB3 peripheral low power mode register, offset: 0x58 */
  __IO uint32_t AHB3LPENR;

  uint32_t      RESERVED4;     /*!< Reserved, 0x5C */

  /*!< RCC APB1 peripheral low power mode register, offset: 0x60 */
  __IO uint32_t APB1LPENR;

  /*!< RCC APB2 peripheral low power mode register, offset: 0x64 */
  __IO uint32_t APB2LPENR;

  uint32_t      RESERVED5[2];  /*!< Reserved, 0x68-0x6C         */

__IO uint32_t BDCR;       /*!< RCC Backup domain control register,offset: 0x70 */

__IO uint32_t CSR;        /*!< RCC clock control & status register,offset: 0x74 */

  uint32_t      RESERVED6[2];  /*!< Reserved, 0x78-0x7C          */

  /*!< RCC spread spectrum clock generation register,offset: 0x80 */
  __IO uint32_t SSCGR;

  /*!< RCC PLLI2S configuration register,            offset: 0x84 */
```

```
  __IO uint32_t PLLI2SCFGR;

  /*!< RCC PLLSAI configuration register,           offset: 0x88 */
  __IO uint32_t PLLSAICFGR;

  /*!< RCC Dedicated Clocks configuration register,  offset: 0x8C */
  __IO uint32_t DCKCFGR;

} RCC_TypeDef;

#define RCC                 ((RCC_TypeDef *) RCC_BASE)


#define LED1    1
#define LED2  2

void init_led(void);
void led_on(int ledx);
void led_off(int ledx);
```

编辑完成后，保存文件到“\基础实验\project\led_nolib\MDK\user” 文件夹中，命名为led.h文件。

同样的方式创建一个led.c源文件，保存在 \基础实验\project\led_nolib\MDK\user目录，编写代码如下：

```
#include "led.h"

void init_led(void)
{

    RCC->AHB1ENR|=1<<0;              //使能PORTA时钟
    RCC->AHB1ENR|=1<<1;              //使能PORTB时钟

    GPIOA_Reg-> MODER&=~(3<<(8*2));  //先清除 PA8 原来的设置
    GPIOA_Reg-> MODER|=1<<(8*2);     //设置 PA8 为输出模式
    GPIOA_Reg-> OTYPER&=~(1<<8) ;    //清除 PA8 原来的设置
    GPIOA_Reg-> OTYPER|=0<<8;        //设置 PA8 为推挽输出
    GPIOA_Reg-> OSPEEDR&=~(3<<(8*2));//先清除 PA8 原来的设置
    GPIOA_Reg-> OSPEEDR|=3<<(8*2);   //设置 PA8 输出速度为 100Mhz
    GPIOA_Reg-> PUPDR&=~(3<<(8*2));  //先清除 PA8 原来的设置
    GPIOA_Reg-> PUPDR|=0<<(8*2);     //设置 PA8 不带上下拉
    GPIOA_Reg-> ODR|=1<<8;          //设置 PA8 输出 1（高电平,LED1初始为关闭
状态）

    GPIOB_Reg-> MODER&=~(3<<(8*2));  //先清除 PB8 原来的设置
    GPIOB_Reg-> MODER|=1<<(8*2);     //设置 PB8 为输出模式
    GPIOB_Reg-> OTYPER&=~(1<<8) ;    //清除 PB8 原来的设置
```

```
    GPIOB_Reg-> OTYPER|=0<<8;        //设置 PB8 为推挽输出
    GPIOB_Reg-> OSPEEDR&=~(3<<(8*2));//先清除 PB8 原来的设置
    GPIOB_Reg-> OSPEEDR|=3<<(8*2);   //设置 PB8 输出速度为 100Mhz
    GPIOB_Reg-> PUPDR&=~(3<<(8*2));  //先清除 PB8 原来的设置
    GPIOB_Reg-> PUPDR|=0<<(8*2);     //设置 PB8 不带上下拉
    GPIOB_Reg-> ODR|=1<<8;          //设置 PB8 输出 1（高电平,LED1初始为关闭
状态）
}


void led_on(int ledx)
{

    if(LED1==ledx)
    {
        GPIOA_Reg-> ODR &=~(1<<8);   //设置 PA8 输出 0（低电平,LED1亮）

    }
    else if(LED2==ledx)
    {
        GPIOB_Reg-> ODR &=~(1<<8);   //设置 PB8 输出 0（低电平,LED2亮）

    }
    else
    {
        ;                           //ledx 不存在处理
    }

}

void led_off(int ledx)
{

    if(LED1==ledx)
    {
        GPIOA_Reg-> ODR|=1<<8;     //设置 PA8 输出 1（高电平,LED1关闭）

    }
    else if(LED2==ledx)
    {
        GPIOB_Reg-> ODR|=1<<8;     //设置 PB8 输出 1（高电平,LED2关闭）

    }
    else
    {
```

```
        ;                                   //ledx 不存在处理
    }
}
```

同样的方式创建一个main.c源文件，在main函数中调用刚刚写的led驱动程序，保存在\基础实验\project\led_nolib\MDK\user 目录，编写代码如下：

```
#include "led.h"

void mydelay(int delaytime);

int main(void)
{
    init_led();

    while(1)
    {
        led_on(LED1);
        led_off(LED2);

        mydelay(1000);

        led_off(LED1);
        led_on(LED2);

        mydelay(1000);
    }
}

void mydelay(int delaytime)
{
    int i;

    for(;delaytime>0;delaytime--)
                for(i=0;i<1000;i++)
                        ;

}
```

复制MDK的stm32F407启动文件 \STM32_Lib\STM32F4xx_DSP_StdPeriph_Lib\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm\startup_stm32f40xx.s 到\基础实验\project\led_nolib\MDK\user目录下，并取消文件只读权限。好了，所有文件都已经到位，我们开始将文件加入到我们刚刚新的工程，如下图：

这个工程主要为了说明，直接寄存器操作方式，所以这里我们先不使用库函数的对系统做初始化，而是使用默认系统初始化值，因此我们需要修改startup_stm32f40xx.s文件，双击工程窗口的文件startup_stm32f40xx.s，打开文件，修改如下：

修改保存后，开始通过usb线将开发板连接PC，设置

E:\Elink407\source_dir\基础实验\project\led_nolib\MDK\led_nolib.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
user
led.c
main.c
sys
startup_stm32f4xx.s
startup_stm32f4xx.s
led.c
main.c
led.h
void mydelay(int delaytime);
int main(void)
{
init_led();
while(1)
{
led_on(LED1);
led_off(LED2);
Build Output
compiling main.c...
assembling startup_stm32f4xx.s...
linking...
Program Size: Code=656 RO-data=408 RW-data=0 ZI-data=1632
FromELF: creating hex file...
".\obj\led_nolib.axf" - 0 Error(s), 0 Warning(s).
Load "E:\\Elink407\\source_dir\\基础实验\\project\\led_nolib\\MDK\\obj\\led_nolib.axf"
Erase Done.
Programming Done.
Verify OK.
Target stopped.
ST-Link Debugger

E:\Elink407\source_dir\基础实验\project\led_nolib\MDK\led_nolib.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
user
led.c
main.c
sys
startup_stm32f4xx.s
startup_stm32f4xx.s
led.c
main.c
led.h
; Reset handler
Reset_Handler PROC
EXPORT Reset_Handler [WEAK]
; IMPORT SystemInit ;//不使用库函数进行初始化
IMPORT main
; LDR R0, =SystemInit ;//不使用库函数进行初始化
; BLX R0 ;//不使用库函数进行初始化
LDR R0, =main
BX R0
ENDP
Text Editor
Configuration Wizard
Build Output
compiling main.c...
assembling startup_stm32f4xx.s...
linking...
Program Size: Code=656 RO-data=408 RW-data=0 ZI-data=1632
FromELF: creating hex file...
Target stopped.
ST-Link Debugger

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

Cortex-M Target Driver Setup
Debug
Trace
Flash Download
Debug Adapter
Unit:
ST-LINK/V2
Serial Number:
N/A
HW Version:
V2
Firmware
V2J20S4
Port:
SW
Max
1MHz
SW Device
IDCODE
Device Name
SWDIO
0x2BA01477
ARM CoreSight SW-DP
Move
Up
Down
Automatic Detection
ID CODE:
Manual Configuration
Device Name:
Add
Delete
Update
IR len:
Debug
Connect & Reset Options
Connect:
Normal
Reset:
Autodetect
Reset after Connect
Cache Options
Cache Code
Cache Memory
Download Options
Verify Code Download
Download to Flash
确定
取消
应用(A)

编译调试代码,并察看结果

7.4 位带区功能，驱动代码设计

在STM32F4xx 器件中，外设寄存器和SRAM 均映射到一个位段区域，这样可实现单个位段的读写操作。这些操作仅适用于Cortex -M4F 访问，对于其它总线主接口（如DMA）无效。可通过一个映射公式说明别名区域中的每个字与位段区域中各个位之间的对应关系。映射公式为:

bit_word_addr= bit_band_base+ (byte_offsetx 32) + (bit_number× 4)

其中:

— bit_word_addr 代表别名区域中将映射到目标位的字的地址

— bit_band_base 代表别名区域的起始地址

— byte_offset 代表目标位所在位段区域中的字节编号

— bit_number 代表目标位的位位置(0 7)

示例

下例说明如何将SRAM 地址0x20000300 处字节的位2 映射到别名区域:

0x22006008 = 0x22000000 + (0x300*32) + (2*4)

对地址0x22006008 执行写操作相当于在SRAM 地址0x20000300 处字节的位2 执行读-修改-写操作。

那么在我们实验中如何使用位带区功能呢，在led.h文件中添加如下代码:

```
#define BIT_BAND_ENB

#ifdef  BIT_BAND_ENB

#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))
#define BIT_ADDR(addr, bitnum)  *((volatile unsigned long  *)(BITBAND(addr, bitnum)))

#define PAout(n)  BIT_ADDR((unsigned int)(&(GPIOA_Reg->ODR)),n)//PA 输出位带区地址
#define PBout(n)  BIT_ADDR((unsigned int)(&(GPIOB_Reg->ODR)),n)//PB 输出位带区地址

#endif
```

修改led.c代码如下：

```
void led_on(int ledx)
{
#ifdef  BIT_BAND_ENB
    if(LED1==ledx)
    {
        PAout(8)=0;                //设置 PA8 输出 0（高电平,LED1亮）

    }
    else if(LED2==ledx)
    {
        PBout(8)=0;;               //设置 PB8 输出 0（低电平,LED2亮）

    }
```

```
    else
    {
        ;                           //ledx 参数错误处理
    }

#else

    if(LED1==ledx)
    {
        GPIOA_Reg-> ODR &=~(1<<8); //设置 PA8 输出 0（低电平,LED1亮）
    }
    else if(LED2==ledx)
    {
        GPIOB_Reg-> ODR &=~(1<<8); //设置 PB8 输出 0（低电平,LED2亮）

    }
    else
    {
        ;                           //ledx 参数错误处理
    }


#endif
}

void led_off(int ledx)
{
#ifdef  BIT_BAND_ENB

    if(LED1==ledx)
    {
        PAout(8)=1;                 //设置 PBA 输出 1（高电平,LED1关闭）

    }
    else if(LED2==ledx)
    {
        PBout(8)=1;                 //设置 PB8 输出 1（高电平,LED2关闭）

    }
    else
    {
        ;                           //ledx参数错误处理
    }

    #else
```

```
    if(LED1==ledx)
    {
        GPIOA_Reg-> ODR|=1<<8;     //设置 PA8 输出 1（高电平,LED1关闭）

    }
    else if(LED2==ledx)
    {
        GPIOB_Reg-> ODR|=1<<8;     //设置 PB8 输出 1（高电平,LED2关闭）

    }
    else
    {
        ;                          //ledx参数错误处理
    }


#endif

}
```

代码编辑完成，编译调试代码,并查看结果

7.5 使用固件库驱动代码设计

上面是我们在讲解固件库之前会首先对寄存器对GPIO操作进行一个讲解，这样的话大家对驱动硬件有个初步的了解(实际上就是在读写相应的寄存器达到对硬件的控制)，对以后的学习也很有帮助。在编写上面的代码的时候，我们发现在使用的时候，每次都直接操作寄存器配置IO，代码会比较多，也不容易记住，另外，用库函数实现的好处是在各个 STM32 芯片上面的移植性非常好，不需要修改任何代码，所以我们在讲解寄存器的同时会讲解是用库函数配置 IO的方法。

7.5.1 库函数和数据结构

本案例涉及的库函数函及相关定义在文件stm32f4xx_gpio.h,stm32f4xx_rcc.h和 stm32f4xx_gpio.c, stm32f4xx_rcc.c中。

结构体:

```
typedef struct
{
    /*This parameter can be any value of @ref GPIO_pins_define */
    uint32_t GPIO_Pin;

    /*!< Specifies the operating mode for the selected pins.*/
    GPIOMode_TypeDef GPIO_Mode;

    /*!< Specifies the speed for the selected pins.*/
```

```
    GPIOSpeed_TypeDef GPIO_Speed;

    /*!the operating output type for the selected pins.*/
    GPIOOType_TypeDef GPIO_OType;

    /*!< Specifies the operating Pull-up/Pull down  pins.*/
    GPIOPuPd_TypeDef GPIO_PuPd;
}GPIO_InitTypeDef;
```

GPIO_InitTypeDef结构体定义了GPIO管脚的大部分属性包括组内编号，输入输出模式，速度，输出类型，上下拉。

库函数：

```
void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
```

外设时钟使能函数，GPIO是挂载在 AHB1 时钟下,所以要使用RCC_AHB1PeriphClockCmd这个函数,挂在非AHB1时钟的外设不能使用该函数进行使能。 第一个参数为外设名称，可以在stm32f4xx_rcc.h文件中找到 RCC_AHB1Periph外设的定义。第二个参数为枚举类型，其定义如下：

```
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;

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

这个函数主要实现 GPIO 的初始化，第一个参数为GPIOx组寄存器的起始地址，第二个参数为已经初始化的结构体GPIO_InitTypeDef的地址。

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

设置ODR 寄存器 输出管脚为高电平，第一个参数为GPIOx组寄存器的起始地址，第二个参数为组内GPIO编号。

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

清ODR 寄存器 输出管脚为低电平，第一个参数为GPIOx组寄存器的起始地址，第二个参数为组内GPIO编号。

7.5.2 软件设计、开发

首先我们复制我们的第一个工程firstPrj文件夹，重命名为led，修改工程文件名为led.uvproj。通常情况下双击工程文件便可以打开这个新的工程，在菜单 >Project >option > Output选项卡 中 配置Name of Executable 为led。

drv文件组中，我们只需要保留，stm32f4xx_gpio.c，stm32f4xx_rcc.c两个文件即可。也可以都编译，只不过其他编译了也不会链接，浪费时间。

新建led.h led.c源文件分别保存在 \基础实验\bsp4Elink\inc\，\基础实验\bsp4Elink\src\ 目录中，以后我们用库函数写的版级驱动程序(bsp)都放在bsp4Elink这个目录中，以后其他实验或工程需要使用到该驱动只需要include 头文件即可，复制led_nolib工程中的main.c 到led\user\ 目录。最终我们建立的工程目录结构如下：

(1) 编辑led.h，代码如下：

```
#ifndef __LED_H
#define __LED_H

#define LED1 1
#define LED2 2

void init_led(void);
void led_on(int ledx);
void led_off(int ledx);

#endif
```

(2) 编辑led.c，代码如下：

```
#include "led.h"
#include "stm32f4xx.h"

void init_led(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;

    /* Enable the GPIO_LED Clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

```
        /* Configure the GPIO_LED pin */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
        GPIO_Init(GPIOA, &GPIO_InitStructure);
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        GPIO_SetBits(GPIOA,GPIO_Pin_8);       //熄灭LED1
        GPIO_SetBits(GPIOB,GPIO_Pin_8);       //熄灭LED2

    }

    void led_on(int ledx)
    {
        if(LED1==ledx)
            GPIO_ResetBits(GPIOA,GPIO_Pin_8); //点亮LED1
        if(LED2==ledx)
            GPIO_ResetBits(GPIOB,GPIO_Pin_8); //点亮LED2
    }

    void led_off(int ledx)
    {
        if(LED1==ledx)
            GPIO_SetBits(GPIOA,GPIO_Pin_8);   //熄灭LED1
        if(LED2==ledx)
            GPIO_SetBits(GPIOB,GPIO_Pin_8);   //熄灭LED2

    }
```

（3）编写延时函数实现开关状态延时保持。

delay.h

```
    #ifndef __DELAY_H
    #define __DELAY_H

    void mydelay(int delaytime);

    #endif
```

delay.c

```
    void mydelay(int delaytime)
    {
       int i;
```

```
    for(;delaytime>0;delaytime--)
        for(i=0;i<1000;i++)
            ;

}
```

(4) 编写main函数进行测试

```
#include "led.h"
#include "delay.h"

int main(void)
{
    init_led();

    while(1)
    {
        led_on(LED1);
        led_off(LED2);

        mydelay(1000);

        led_off(LED1);
        led_on(LED2);

        mydelay(1000);
    }
}
```

7.5.3 下载测试

代码编辑完成，编译调试代码，并查看结果

第 8 章

蜂鸣器实验

上一章，我们介绍了 STM32F4 的 IO 口作为输出的使用，这一章我们继续巩固前面的知识，通过GPIO控制蜂鸣器，希望大家看完本章的实验原理与基础知识和电路设计部分的内容后能独立完成驱动蜂鸣器驱动程序的编写。

8.1 实验原理与基础知识

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

这里的有源不是指电源的“源”，而是指有没有自带震荡电路，有源蜂鸣器自带了震荡电路，一通电就会发声；无源蜂鸣器则没有自带震荡电路，必须外部提供 2~5Khz 左右的方波驱动，才能发声。

ELink407开发板板载的蜂鸣器是电磁式的有源蜂鸣器，如图所示：

8.2 电路设计

STM32F4 的单个 IO 最大可以提供 25mA 电流，而蜂鸣器的驱动电流是 30mA左右，考虑STM32F4 整个芯片的电流，最大也就 150mA，如果用IO 口直接驱动蜂鸣器，其他地方用电就会不足，所以我们不用 STM32F4 的 IO 直接驱动蜂鸣器，而是通过三极管扩流后再驱动蜂鸣，这样 STM32F4 的 IO 只需要提供不到 1mA 的电流就足够了。所以我们的硬件电路设计如下：

从电路原理图上，我们可以看到BEEP 和 ADC0 复用了PC0 管脚，在开发板上有一个跳线J34，如果我们需要使用蜂鸣器，需要跳线帽将J34的 1，2脚短接。

8.3 库函数和数据结构

跟GPIO使用库函数一样

8.4 软件设计、开发

首先我们复制我们的第一个工程firstPrj文件夹，重命名为buzzer，修改工程文件名为buzzer.uvproj。双击打开工程，在菜单 >Project >option > Output选项卡 中 配

J10
R32
U7
C43
Q2
R140
L6
L8
J23
J22

BEEP
3.3V
BEEP
R140
470R
Q2
S8050
J22
8 OHM?
1
2
3
J34
CON3
BEEP
BEEP/ADC0
ADC0

置Name of Executable 为buzzer。

在固件库中，GPIO 端口操作对应的库函数函数以及相关定义在文件stm32f4xx_gpio.h 和 stm32f4xx_gpio.c 中，所以drv文件组中，我们只需要保留，stm32f4xx_gpio.c，stm32f4xx_rcc.c 两个文件即可。

(1) buzzer.h

```
#ifndef __BUZZER_H
#define __BUZZER_H

void init_buzzer(void);
void buzzer_on(void);
void buzzer_off(void);

#endif
```

(2) buzzer.c

```
#include "buzzer.h"
#include "stm32f4xx.h"

void buzzer_led(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;

    /* Enable the GPIO_BUZZER Clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    /* Configure the GPIO_BUZZER pin */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_SetBits(GPIOC,GPIO_Pin_0);       //配置蜂鸣器 不响

}

void buzzer_on(void)
{
        GPIO_ResetBits(GPIOC,GPIO_Pin_0); //蜂鸣器响
}

void buzzer_off(void)
{
        GPIO_SetBits(GPIOC,GPIO_Pin_0);   //不响
```

```
}
```

将编写好的buzzer驱动分别保存在 \基础实验\bsp4Elink\inc\，\基础实验\bsp4Elink\src\ 目录中。

(3) 编写案例测试代码main.c

```
#include "buzzer.h"
#include "delay.h"

int main(void)
{
    init_buzzer();

    while(1)
    {
        buzzer_on();
        mydelay(1000);

        buzzer_off();
        mydelay(10000);
    }
}
```

8.5 下载测试

第 9 章

按键实验

前面的实验我们学会了，通过STM32F407输出信息，控制外部设备（LED，BUZZER）。那么CPU从外界获取信息的过程又是怎么样的呢？本案例我们让大家实现通过按键输入来控制LED灯。

9.1 实验原理与基础知识

在前面的案例中，我们是通过库函数 GPIO_ResetBits()和GPIO_SetBits()来设置两个输出数据寄存器。STM32F4 的 IO口做输入使用的时候，是通过调用函数 GPIO_ReadInputDataBit()来读取 IO 口的状态的。那么可以开始我们的代码编写了。

9.2 电路设计

9.3 库函数和数据结构

本案例涉及的库函数函及相关定义在文件stm32f4xx_gpio.h,stm32f4xx_rcc.h和 stm32f4xx_gpio.c，stm32f4xx_rcc.c 中。

库函数：

```
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

函数读取 IDR 寄存器组内管脚上电平值，第一个参数为GPIOx组寄存器的起始地址，第二个参数为组内GPIO编号。

9.4 软件设计、开发

在工程中添加button_noexti.c 文件和button_noexti.h 文件，按键相关的函数和定义都在这两个文件中。

(1) 初始化GPIOD，GPIOE 上的PIN,配置为输入模式

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

    /* Enable the GPIO_BUTTON Clock */
```

KEY
3.3V
R51
R50
R49
R37
4.7K
4.7K
4.7K
4.7K
SW36
SW35
SW27
SW1
PUSHBUTTON
KEY1
KEY2
KEY3
PD6/KEY4

```
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

    /* Configure the GPIO_BUTTON pin */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;

    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3 |GPIO_Pin_4 ;
    GPIO_Init(GPIOE, &GPIO_InitStructure);

}
```

(2)读取按键状态

```
int button_read(void)
{
    int status=0;

    status=  GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_4)<<3
                | GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3)<<2
                | GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2)<<1
                | GPIO_ReadInputDataBit(GPIOD,GPIO_Pin_6)<<0;

    return (~status) &0xF;

}
```

(3) 编写测试程序

```
#include "stm32f4xx.h"
#include "led.h"
#include "delay.h"

#include "button_noexti.h"

int main()
{
    int button_value=0;

    init_led();

    while(1)
    {
```

```
        button_value=button_read();

        if(button_value>0)
        {
            mydelay(10);        //延时去抖

            if(button_value==button_read())
            {
                    switch(button_value)
                    {
                        case 1:
                            led_on(LED1);
                            break;

                        case 2:
                            led_on(LED2);
                            break;

                        case 4:
                            led_on(LED1);
                            led_on(LED2);
                            break;

                        case 8:
                            led_off(LED1);
                            led_off(LED2);
                            break;

                        default:            //两个键同时按下，不处理
                            break;
                    }
                }
            }
        }
    }
```

9.5　软件下载、测试验证

EXTI验 # 上个一实验，我们通过CPU不断读取按键GPIO 输入寄存器的值，查询是否有按键按下，并做相应的控制操作，但是这种轮询的方式存在两个问题，首先浪费CPU的资源，无论按键是否发生，CPU都要做查询操作。其次，如果CPU的任务比较繁多，CPU有可能不能及时响应按键的请求，甚至不响应。解决这个问题，我们引入了中断机制，当按键按下以后，产生一个中断请求，CPU收到请求后，中断当前用户程序，转而执行中断响应代码。下面的案例就是在ELink407上实现按键中断控制LED灯。

9.6 实验原理与基础知识

STM32F4 的每个 IO 都可以作为外部中断的中断输入口。 STM32F407 的中断控制器支持 22 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。 STM32F407的 22 个外部中断为：

EXTI 线 0～15：对应外部 IO 口的输入中断。

EXTI 线 16：连接到 PVD 输出。

EXTI 线 17：连接到 RTC 闹钟事件。

EXTI 线 18：连接到 USB OTG FS 唤醒事件。

EXTI 线 19：连接到以太网唤醒事件。

EXTI 线 20：连接到 USB OTG HS(在 FS 中配置)唤醒事件。

EXTI 线 21：连接到 RTC 入侵和时间戳事件。

EXTI 线 22：连接到 RTC 唤醒事件。

从上面可以看出，STM32F4 供 IO 口使用的中断线只有 16 个，但是 STM32F4 的 IO 口却远远不止 16 个，那么 STM32F4 是怎么把 16 个中断线和 IO 口一一对应起来的呢？于是 STM32就这样设计， GPIO 的管脚 GPIOx.0～GPIOx.15 (x=A,B,C,D,E, F,G,H,I)分别对应中断线 0～15。这样每个中断线对应了最多 9 个 IO 口，以线 0 为例：它对应了 GPIOA.0、GPIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0、GPIOG.0,GPIOH.0,GPIOI.0。而中断线每次只能连接到 1 个 IO口上，这样就需要通过配置来决定对应的中断线配置到哪个 GPIO 上了。下面我们看看 GPIO跟中断线的映射关系图：

要产生中断，必须先配置好并使能中断线。根据需要的边沿检测设置2 个触发寄存器，同时在中断屏蔽寄存器的相应位写“1”使能中断请求。当外部中断线上出现选定信号沿时，便会产生中断请求，对应的挂起位也会置1。在挂起寄存器的对应位写“1”，将清除该中断请求。

9.7 电路设计

9.8 库函数和数据结构

本案例涉及的库函数函及相关定义在文件stm32f4xx_gpio.h,stm32f4xx_rcc.h，stm32f4xx_exti.h和 stm32f4xx_gpio.c，stm32f4xx_rcc.c,stm32f4xx_exti.c中。

结构体：

```
typedef struct
{
    /*!< Specifies the IRQ channel to be enabled or disabled.*/
  uint8_t NVIC_IRQChannel;

  /*!< Specifies the pre-emption priority for the IRQ channel*/
  uint8_t NVIC_IRQChannelPreemptionPriority;

  /*!< Specifies the subpriority level for the IRQ channel */
  uint8_t NVIC_IRQChannelSubPriority;

  /*!< Specifies whether the IRQ channel defined in NVIC_IRQChannel*/
  FunctionalState NVIC_IRQChannelCmd;
```

图 33. 外部中断/事件 GPIO 映射

KEY
3.3V
R51
R50
R49
R37
4.7K
4.7K
4.7K
4.7K
SW36
SW35
SW27
SW1
PUSHBUTTON
KEY1
KEY2
KEY3
PD6/KEY4

```
} NVIC_InitTypeDef;

typedef struct
{
    /*!< Specifies the EXTI lines to be enabled or disabled.*/
  uint32_t EXTI_Line;

  /*!< Specifies the mode for the EXTI lines.*/
  EXTIMode_TypeDef EXTI_Mode;

  /*!< Specifies the trigger signal active edge for the EXTI lines.*/
  EXTITrigger_TypeDef EXTI_Trigger;

  /*!< Specifies the new state of the selected EXTI lines.*/
  FunctionalState EXTI_LineCmd;
}EXTI_InitTypeDef;
```

EXTI_Line中断线的标号，对于我们的外部中断，取值范围为 EXTI_Line0～EXTI_Line15。这个在上面已经讲过中断线的概念。也就是说，这个函数配置的是某个中断线上的中断参数。

EXTI_Mode中断模式，可选值为中断EXTI_Mode_Interrupt 和事件 EXTI_Mode_Event。

EXTI_Trigger触发方式，可以是下降沿触发EXTI_Trigger_Falling，上升沿触发 EXTI_Trigger_Rising，或者任意电平（上升沿和下降沿）触发 EXTI_Trigger_Rising_Falling，。

EXTI_LineCmd使能中断线为DISABLE或 ENABLE。

库函数：

```
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

开启 SYSCFG 时钟函数，SYSCFG 是挂载在 APB2 时钟下,所以要使用RCC_APB2PeriphClockCmd这个函数。 第一个参数为外设名称，可以在stm32f4xx_rcc.h文件中找到 RCC_AHP2Periph外设的定义。第二个参数为DISABLE或 ENABLE

```
void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex);
```

配置 GPIO 与中断线的映射。

```
void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct);
```

中断线上中断的初始化函数。

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct);
```

中断优先级分组初始化函数

```
ITStatus EXTI_GetITStatus(uint32_t EXTI_Line);
```

这个函数一般使用在中断服务函数的开头判断中断是否发生。

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
```

清除某个中断线上的中断标志位函数

9.9 软件设计、开发

在工程中添加button.c 文件和button.h 文件，按键中断相关的函数和定义都在这两个文件中。

使用 IO 口外部中断的步骤：

1）使能 IO 口时钟，初始化 IO 口为输入首先，我们要使用 IO 口作为中断输入，所以我们要使能相应的 IO 口时钟，以及初始化相应的 IO 口为输入模式，具体的使用方法跟我们按键实验是一致的。然后开启 SYSCFG 时钟，

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);//使能 SYSCFG 时钟
```

2）使能 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

```
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOD, EXTI_PinSource6);
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource2);
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource3);
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource4);
```

3）初始化线上中断，设置触发条件等。

```
EXTI_InitTypeDef    EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line=EXTI_Line4;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);      //初始化外设 EXTI 寄存器
```

4）配置中断分组（NVIC），并使能中断。

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;          //使能按键外部中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;  //抢占优先级 2,
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;    //响应优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;      //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);                          //中断优先级分组初始化
```

5）编写中断服务函数。

中断服务函数的名字是在标准库中事先有定义的。这里需要说明一下， STM32F4 的 IO 口外部中断函数只有 7 个，分别为：

```
EXPORT   EXTI0_IRQHandler
EXPORT   EXTI1_IRQHandler
EXPORT   EXTI2_IRQHandler
EXPORT   EXTI3_IRQHandler
EXPORT   EXTI4_IRQHandler
EXPORT   EXTI9_5_IRQHandler
EXPORT   EXTI15_10_IRQHandler
```

中断线 0　4 每个中断线对应一个中断函数，中断线 5　9 共用中断函数 EXTI9_5_IRQHandler，中断线 10　15 共用中断函数 EXTI15_10_IRQHandler。我们以按键1和按键2中断服务程序为例：

按键1对应GPIOD_pin_6，所以中断服务程序名称为EXTI9_5_IRQHandler

```
void EXTI9_5_IRQHandler(void)
 {
    if(EXTI_GetITStatus(EXTI_Line6)!=RESET)//判断某个线上的中断是否发生
    {
        //…中断处理…
        led_on()
        EXTI_ClearITPendingBit(EXTI_Line6);   //清除 LINE 上的中断标志位
    }
}
```

按键1对应GPIOE_Pin_2，所以中断服务程序名称为EXTI2_IRQHandler

```
void EXTI2_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line2)!=RESET)//判断某个线上的中断是否发生
    {
        //…中断处理…
        led_on()
        EXTI_ClearITPendingBit(EXTI_Line2);   //清除 LINE 上的中断标志位
    }
}
```

(6) 编写main函数进行测试

```
#include "led.h"
#include "delay.h"
#include "button_noexti.h"

#include "button.h"

int main()
{
    init_led();

   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2

    EXTIX_Init();       //初始化外部中断输入


        while(1)
        {

           mydelay(1000);
```

```
        }
    }
```

9.10 软件下载、测试验证

第 10 章

Systick实验

在前面的实验中，我们实现了一个延时函数mydelay()，这个函数帮助我们实现了LED闪烁，按键去抖等效果，满足效果是没有问题，但实际使用这样的延时函数是不合适的，它占用了宝贵的CPU资源，而且延时时间不准确，那么在本案例中，我们学习systich部分的知识，完成一个实用的毫秒延时函数，并改写mydelay()函数。

10.1 实验原理与基础知识

大多操作系统需要一个定时器来产生周期性的中断,作为整个系统的时基，而且最好还让用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。该定时器的时钟源可以是内部时钟（FCLK，上的自由运行时钟），或者是外部时钟（处理器上的 STCLK 信号）。 CM4 内核的处理内部都包含了一个 SysTick 定时器，SysTick 定时器能产生中断，CM4 为它专门开出一个异常类型（异常号：15）并且在向量表中有它的一席之地。SysTick 是一个 24 位的倒计数定时器，当计到 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。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。

SysTick control and status register (STK_CTRL)
偏移地址：0x00

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSOURCE	TICKINT	ENABLE
													rw	rw	rw

Bits 31:17 保留，但必须保持清零
Bit 16 COUNTFLAG：如果在上次读取本寄存器后，SysTick 已经计到 0，那么该位是 1
Bits 15:3 保留，但必须保持清零
Bit 2 CLKSOURCE：时钟源选择

0： AHB/8

1: Processor clock (AHB)

Bit 1 TICKINT:嘀嗒定时器中断请求使能

1: SysTick 倒数计数到 0 时产生 SysTick 异常请求

0: 数到 0 的时候无动作

Bit 0 ENABLE: 计数使能

0 : 计数禁止

1: 计数使能

SysTick reload value register (STK_LOAD) 偏移地址: 0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSOURCE	TICKINT	ENABLE
													rw	rw	rw

Bits 31:24 保留，但必须保持清零

Bits 23:0 RELOAD:重载数值 当使能了计数，并且计数到 0 的时候，这个寄存器的数据将被重新的加载到寄存器。 计算重载数值 RELOAD 寄存器的数值从 0x00000001 0x00FFFFFF，寄存器的数值可以赋值为 0，但是没有影响，只有数值从 1 到 0 的时候嘀嗒定时器的中断请求和计数标志才有效。

产生一个周期为 N 的多节拍定时器，需要将寄存器 RELOAD 设置为 N 1，比如我们需要嘀嗒定时器每 100 个时钟脉冲产生一个中断，那么 RELOAD=99。

N 个时钟周期后产生一个滴答中断，需要 RELOAD = 99，比如我们需要嘀嗒定时器每100个时钟脉冲产生一个中断，那么 RELOAD=99.

SysTick current value register (STK_VAL)
偏移地址：0x08

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CURRENT[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CURRENT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:24 保留，但必须保持清零

Bits 23:0 CURRENT:当前计数值 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。

SysTick calibration value register (STK_CALIB)
偏移地址：0x0c

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NO REF	SKEW	Reserved						TENMS[23:16]							
r	r							r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TENMS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bit 31 NOREF：读取结果为 0 表示分离的参考时钟可用，频率时 HCLK/8

Bit 30 SKEW：表示 TENMS 的值是否是准确的，校准值是 1ms 不准确的定时是未知的，因为 TENMS 是不知道的，这可能会影响系统定时器是否适合作为一个软件实时时钟

Bits 29:24 保留，但必须保持清零

Bits 23:0 校准值，当嘀嗒定时器运行在 HCLK/8 时表示校准值，该值取决于产品，详细信息请看参考手册的嘀嗒定时器校验值部分，当 HCLK 被编程为最高频率时，嘀嗒定时器的周期是 1ms，如果没有已知的校准信息，需要从核心频率和外部时钟计数校准。

10.2 电路设计

无

10.3 库函数和数据结构

相关的函数主要分布在两个文件里面，一个是文件 misc.c 另一个是core_cm4.h。下面我把相关的函数贴出来说明一下。

SysTick_Config()

函数SysTick_Config()是属于CMSIS里面的一个函数，实现配置如下

- 函数的参数是重载寄存器reload要赋予的初值
- 配置嘀嗒定时器的优先级是最低优先级15
- 配置HCLK作为系统的时钟源
- 使能嘀嗒定时器中断
- 启动嘀嗒定时器计数

  void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)

这个函数主要实现嘀嗒定时器时钟源的选择. 用户可以在调用函数SysTick_Config()后通过函数 SysTick_CLKSourceConfig() 更改嘀嗒定时器的时钟源为HCLK/8。SysTick_CLKSourceConfig()在文件misc.c里面

NVIC_SetPriority()

用户可以在调用函数SysTick_Config()后通过函数 NVIC_SetPriority()修改优先级，函数NVIC_SetPriority()在文件core_cm4.h文件里面 通过下面的公式调整时基

Reload Value = SysTick Counter Clock (Hz) x Desired Time base (s)
Reload Value 是函数SysTick_Config()的参数，但是不能超过0xFFFFFF

这个函数主要是实现嘀嗒定时器的配置，这里面有个对嘀嗒定时器优先级的默认设置，这里将其设置成了15系统最低优先级

10.4 软件设计、开发

在工程中添加delay.c 文件和delay.h 文件，所有延时相关的函数和定义都在这两个文件中。

(1)定义延时计数全局变量

```
unsigned int TimingDelay;
```

(2)设置嘀嗒定时器是1us中断

```
void init_delay()
{
    if (SysTick_Config(SystemCoreClock / 1000000))
    {
        /* 出错 */
        while (1);
    }
}
```

(3)编写异常向量中断服务程序

```
void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0)
    {
        TimingDelay--;
    }
}
```

(4)编写毫秒延时代码

```
void delay_ms( unsigned int mTime)
{
    TimingDelay = mTime*1000;
    while(TimingDelay != 0);
}
```

(5)编写微秒延时代码

```
void delay_us( unsigned int uTime)
{
    TimingDelay = mTime;
    while(TimingDelay != 0);
}
```

(6)安装异常向量中断服务程序

打开文件 \基础实验\common\MDK\stm32f4xx_it.c，去掉中断服务程序的注释

```
void SysTick_Handler(void)
{
  //去掉下面函数的注释
    TimingDelay_Decrement();
}
```

(7)为了使原来的mydelay()函数能用，改写代码

```
void mydelay(int delaytime)
{
    delay_ms(delaytime);

#if 0     //使用标准延时函数
    int i;

    for(;delaytime>0;delaytime--)
                for(i=0;i<1000;i++)
#endif
                        ;
}
```

10.5 软件下载、测试验证

第 11 章

NVIC实验

STM32F40xx 总共有 92 个中断 包括 10 个内核中断和 82 个可屏蔽中断，具有 16 级可编程的中断优先级，那么这么多的中断源，可是我的COTEX M4只有一个IRQ中断信号，怎么办呢？NVIC（Nested vectored interrupt controller）可嵌套向量中断控制器，就是协助内核管理中断的使能，优先级，中断嵌套，中断挂起等待，中断触发方式等。前面的案例我们使用了外部中断（按键中断）和系统中断(systick)，那么是否系统中断要高呢？本实验我们验证一下，也是为了加深对NVIC的学习。

11.1 实验原理与基础知识

Cortex-M4 内核有一个 8位的寄存器支持 128 级中断嵌套，用户可以通过这个寄存器设置中断的优先级，STM32F407 只使用了这个 8 位寄存器的高四位7:4（section B.2），低四位取零，这样407只使用了 16 级中断嵌套。支持 5 种优先级分组，系统复位默认使用的是优先级分组 0，也就是没有抢占式优先级，只有子优先级。

关于这个抢占优先级和这个子优先级的说明：

1. 具有高抢占式优先级的中断可以在具有低抢占式优先级的中断处理过程中被响应，即中断嵌套，或者说高抢占式优先级的中断可以嵌套低抢占式优先级的中断。

2. 在抢占式优先级相同的情况下，有几个子优先级不同的中断同时到来，那么高子优先级的中断优先被响应。

3 在抢占式优先级相同的情况下，如果有低副优先级中断正在执行，高副优先级的中断要等待已被响应的低副优先级中断执行结束后才能得到响应——非抢断式响应(不能嵌套)。

4 判断中断是否被相应的依据

先是占先式优先级，其次是副优先级

* 占先式优先级决定是否会有中断嵌套
* Reset、NMI、Hard Fault 优先级为负高于普通中断优先级)且不可调整

中断使能寄存器 NVIC_ISERx

如果使能了挂起中断，NVIC 将激活基于此优先级的中断，如果没有使能，就是中断信号改变了挂起状态，但是 NVIC 不会激活此中断，并且忽略它的优先级。

中断屏蔽寄存器 NVIC_ICERx

中断挂起寄存器 NVIC_ISPRx

清除中断挂起寄存器 NVIC_ICPRx

中断激活状态寄存器 NVIC_IABRx
中断优先级寄存器
软件触发中断寄存器 NVIC_STIR

11.2 电路设计

11.3 库函数和数据结构

涉及到 NVIC 的库文件主要有 misc.c 文件以及 core_cm4.h 文件，下面就把这些相关的函数做一下解释，先介绍 misc.c 文件里面的函数，其实有了前面的介绍，这个文件里面的函数还是比较的容易理解的。

```
NVIC_PriorityGroupConfig (uint32_t NVIC_PriorityGroup)
```

这个函数用于设置优先级分组，记住系统复位默认是 0 分组，也就是没有抢占优先级，只有 0 15 的子优先级。

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
```

为了正确的配置优先级，之前需要先调用一下优先级分组 NVIC_PriorityGroupConfig 函数，这两个函数应该是平时用的最多的两个函数了

```
void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset)

void NVIC_SystemLPConfig(uint8_tLowPowerMode, FunctionalState NewState)
```

这个函数用于设置低功耗模式
下面我们再讲一下 core_cm4.h 文件里面的函数，这个里面的函数用到的时候比较少，不过都是访问NVIC 寄存器最直接的函数，有时候用这些函数更加的方便，下面就讲解一下。

```
void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
```

这个函数和前面的 NVIC_PriorityGroupConfig 函数是一样的，只不过一个是 ST 公司做的，一个是ARM 公司做的。

```
uint32_t NVIC_GetPriorityGrouping(void)
```

此函数用于读取优先级分组。

```
void NVIC_EnableIRQ(IRQn_Type IRQn)
void NVIC_DisableIRQ(IRQn_Type IRQn)
```

这两个函数是配套使用的，注意这两个函数不能设置系统中断，系统中断有自己的函数。

```
NVIC_SetPendingIRQ(IRQn_Type IRQn)
NVIC_ClearPendingIRQ(IRQn_Type IRQn)
NVIC_GetPendingIRQ(IRQn_Type IRQn)
```

这三个函数是一组的，设置挂起和清除挂起。

```
NVIC_GetActive(IRQn_Type IRQn)
```

这个函数用于获得激活状态，也就是获得是否进入了相应中断程序。

```
NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
uint32_t NVIC_GetPriority(IRQn_Type IRQn)
```

这两个函数是一组的，一个用于设置中断的优先级，一个用于得到中断的优先级，它们可以设置系统中断也可以设置外设中断，前面提到我们是16级的中断优先级，所以优先级的输入范围和输出范围是 0～15。

```
uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriorit
void NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t* pPreemptPriority, uir
```

这两个函数是一组的，一个用于优先级的编码，一个用于优先级的解码。

```
void NVIC_SystemReset(void)
```

这个函数用于实现系统的软件复位。

11.4 软件设计、开发

11.5 软件下载、测试验证

第 12 章

UART 实验

UART是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。在嵌入式设计中，UART常用来主机与辅助设备通信，另外通过串口信息对代码进行调试也是常用手段之一。

12.1 实验原理与基础知识

stm32f407系列，串口有3种工作模式：查询、中断、DMA。查询最简单，效率也最低，因为mcu速度很快，而串口发送效率很低，所以实际项目中基本不用。中断模式效率高，但如果发送/接收大量字节，会产生大量中断，影响整个系统效率。DMA方式适合于高速设置的数据传输，如果应用要求Uart 的速度不是很高不建议采用，另外使用DMA方式会占用DMA通道，stm32如果其它设备占用了DMA通道，那么久不能使用。

本案例采用中断方式，利用st库函数对编写UART1串口驱动，并通过重写fputc函数，实现使用printf函数打印串口消息。

USART_SR 寄存器

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。

RXNE（读数据寄存器非空），当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TC （发送完成），当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如果设置了这个位的中断，则会产生中断。该位也有两种清零方式：1）读 USART_SR，写USART_DR。2）直接向该位写 0。

12.2 电路设计

UART1串口对应收发引脚连接在开发GPIO的PA9和PA10，PA9,PA10经过USB转串口电路，跟PC用mini USB线连接，如图

我们把的UART1 UART3 的Pin也扩展出来了，可以参看原理图J9,J12。

12.3 库函数和数据结构

串口基本配置直接相关的几个固件库函数。这些函数和定义主要分布在 stm32f4xx_usart.h 和 stm32f4xx_usart.c 文件中。

5V_RS
R38
R39
R41
27R
27R
100K
D-
D+
J2
MINIUSB
UART1_RX
UART1_TX
R40
33R
3.3V_RS
C29
100nF
R46
33R
RI
DSR
DCD
CTS
3.3V_RS
R28
R31
4.7K/NC
4.7K/NC
TXD
DTR
RTS
VDD
RXD
RI
GND
NC
DSR
DCD
CTS
SHTD
CLK
DATA
OSC2
OSC1
PLL_TEST
GND
NC
GP0
GP1
GND
VCC
NC
GND
VO
DM
DP
U10
PL2303
C25
15pF
Y3
12MHZ
C26
15pF
5V_RS
C27
100nF
3.3V_RS
C28
100nF
R30
1.5K
D-
D+
RP1
220K/NC
RI
DSR
DCD
CTS

RS232&RS485
V5
3.3V
L13
80R
L9
NC
CON4
J9
UART1_TX
UART1_RX
USART1
V5
3.3V
L15
80R
L14
NC
CON4
J12
IIS_CK/USART3_TX
TX-EN/UART3_RX
USART3
3.3V
R35
R36
R24
R29
4.7K
4.7K
4.7K
4.7K
UART1_TX
UART1_RX
IIS_CK/USART3_TX
TX-EN/UART3_RX

```
RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

串口是挂载在 APB2 下面的外设，所以使用RCC_APB2PeriphClockCmd()函数：

```
void GPIO_PinAFConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF);
```

设置引脚复用器映射函数，对于 GPIO_PinAFConfig 在相应的配置文件找到外设对应的 AF 配置宏定义标识符即可，宏定义在文件 stm32f4xx_gpio.h中，串口 1 为 GPIO_AF_USART1。

```
void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)
```

串口参数初始化：设置波特率，字长，奇偶校验等参数

```
void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
```

使能串口函数，将 USART_CR1 寄存器的 UE 置 1 来激活 USART。

```
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);
```

库函数操作 USART_DR 寄存器发送数据的函数，通过该函数向串口寄存器 USART_DR 写入一个数据。

```
uint16_t USART_ReceiveData(USART_TypeDef* USARTx);
```

库函数操作 USART_DR 寄存器读取串口接收到的数据的函数，通过该函数可以读取串口接受到的数据。

12.4 软件设计、开发

在工程中添加uart.c 文件和uart.h 文件，所有 uart相关的驱动代码和定义都在这两个文件中。

1）串口时钟和 GPIO 时钟使能。

串口是挂载在 APB2 下面的外设，所以使能函数为：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);//使能 USART1 时钟
```

GPIO 时钟使能，串口 1 对应着芯片引脚 PA9,PA10，所以这里我们需要使能 GPIOA 时钟：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
```

2）设置引脚复用器映射

PA9 和 PA10 都映射到串口 1，引脚复用器映射，调用函数为：

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1); //PA9 复用为 USART1
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1);//PA10 复用为 USART1
```

对于 GPIO_PinAFConfig 在相应的配置文件找到外设对应的 AF 配置宏定义标识符即可，串口 1 为 GPIO_AF_USART1。

3）GPIO 端口模式设置：PA9 和 PA10 要设置为复用功能。

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //GPIOA9 与 GPIOA10
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  //速度 50MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9, PA10
```

4）串口参数初始化：设置波特率，字长，奇偶校验等参数

串口初始化是调用函数 USART_Init 来实现的，具体设置方法如下：

```
USART_InitStructure.USART_BaudRate = bound;//设置为 115200;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为 8 位数据格式
USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//收发模式
USART_Init(USART1, &USART_InitStructure); //初始化串口
```

5）调用函数 USART_Cmd 使能串口 1 方法如下：

```
USART_Cmd(USART1, ENABLE);   //使能串口
```

6）引入 printf 和scanf 函数支持需要 重定义 fputc() 和 fgetc()函数

```
int fputc(int ch, FILE *f)
{
    /* 写一个字节到USART1 */
    USART_SendData(USART1, (uint8_t) ch);

    /* 等待发送结束 */
    while (USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET)
    {
    }
    return ch;
}

int fgetc(FILE *f)
{
    /* 等待串口1输入数据 */
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    return (int)USART_ReceiveData(USART1);
}
```

7）清发送完成标志

```
USART_ClearFlag(USART1, USART_FLAG_TC);
```

12.5 软件下载、测试验证

第 13 章

DMA实验

本章我们将向大家介绍 STM32F4 的 DMA。在本章中，我们将利用 STM32F4 的 DMA 来实现UART数据的发送，以此学习DMA的使用。

13.1 实验原理与基础知识

Direct Memory Access(DMA)，即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。

STM32F4 最多有 2 个 DMA 控制器（DMA1 和 DMA2），共 16 个数据流（每个控制器 8 个），每一个 DMA 控制器都用于管理一个或多个外设的存储器访问请求。每个数据流总共可以有多达 8个通道（或称请求）。每个数据流通道都有一个仲裁器，用于处理 DMA 请求间的优先级。

DMA 的框图

DMA 控制器执行直接存储器传输：因为采用 AHB 主总线，它可以控制 AHB 总线矩阵来启动 AHB 事务。它可以执行下列事务：

1，外设到存储器的传输

2，存储器到外设的传输

3，存储器到存储器的传输

DMA 控制器提供两个AHB 主端口：AHB 存储器端口（用于连接存储器）和 AHB 外设端口（用于连接外设）。但是，要执行存储器到存储器的传输，AHB 外设端口必须也能访问存储器。

两个DMA 控制器的系统实现，见图

DMA1 控制器AHB 外设端口与DMA2 控制器的情况不同，不连接到总线矩阵，因此，仅DMA2数据流能够执行存储器到存储器的传输。

每个数据流都与一个DMA 请求相关联，此DMA 请求可以从8 个可能的通道请求中选出。此选择由DMA_SxCR 寄存器中的CHSEL2:0（section B.2）位控制。

下表是DMA2 请求映射

我们要实现串口 1的 DMA 发送，即 USART1_TX，就必须选择 DMA2 的数据流 7，通道 4，来进行 DMA 传输。

13.2 电路设计

DMA 控制器
REQ_STR0_CH0
REQ_STR0_CH1
REQ_STR0_CH7
REQ_STR1_CH0
REQ_STR1_CH1
REQ_STR1_CH7
REQ_STR7_CH0
REQ_STR7_CH1
REQ_STR7_CH7
通道选择
REQ_STREAM0
REQ_STREAM1
REQ_STREAM2
REQ_STREAM3
REQ_STREAM4
REQ_STREAM5
REQ_STREAM6
REQ_STREAM7
仲裁器
流 0
流 1
流 2
流 3
流 4
流 5
流 6
流 7
FIFO
AHB 主器件
存储器端口
外设端口
AHB 从器件
编程接口
编程端口

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

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

13.3 库函数和数据结构

DMA_InitTypeDef结构体的定义：

```
typedef struct
{
    uint32_t DMA_Channel;
    uint32_t DMA_PeripheralBaseAddr;
    uint32_t DMA_Memory0BaseAddr;
    uint32_t DMA_DIR;
    uint32_t DMA_BufferSize;
    uint32_t DMA_PeripheralInc;
    uint32_t DMA_MemoryInc;
    uint32_t DMA_PeripheralDataSize;
    uint32_t DMA_MemoryDataSize;
    uint32_t DMA_Mode;
    uint32_t DMA_Priority;
    uint32_t DMA_FIFOMode;
    uint32_t DMA_FIFOThreshold;
    uint32_t DMA_MemoryBurst;
    uint32_t DMA_PeripheralBurst;
}DMA_InitTypeDef;
```

```
void DMA_Init(DMA_Stream_TypeDef* DMAy_Streamx, DMA_InitTypeDef* DMA_InitStruct);
```

函数的第一个参数是指定初始化的 DMA 的数据流编号，入口参数范围为：y=1,2;x=1 7。第二个参数DMA_InitTypeDef结构体

```
void DMA_Cmd(DMA_Stream_TypeDef* DMAy_Streamx, FunctionalState NewState);
```

使能 DMA 数据流的函数。

```
void USART_DMACmd(USART_TypeDef* USARTx, uint16_t USART_DMAReq, FunctionalState NewState) ;
```

开启串口的 DMA 发送/接送功能，第一个参数为串口编号，第二个为串口的收发功能，第三个参数为使能状态。

```
FlagStatus DMA_GetFlagStatus(DMA_Stream_TypeDef* DMAy_Streamx, uint32_t DMA_FLAG)
```

查询 DMA 传输通道的状态

```
void DMA_ClearFlag(DMA_Stream_TypeDef* DMAy_Streamx, uint32_t DMA_FLAG) ;
```

清除DMA传输完成标志

```
void USART_ClearFlag(USART_TypeDef* USARTx, uint16_t USART_FLAG);
```

清除串口传输完成标志

13.4 软件设计、开发

在uart.c文件中添加如下代码：

```
#define UART_DMA

#ifdef UART_DMA

#define TIMEOUT_MAX 1000 /* 定义最大的溢出数值 */

void USART_DMA_TxConfig(uint32_t *BufferSRC, uint32_t BufferSize)
{
    DMA_InitTypeDef DMA_InitStructure;
    __IO uint32_t Timeout = TIMEOUT_MAX;
    /* 复位 DMA Stream 寄存器（用于调试目的）*/
    /* DMA_DeInit(USARTx_TX_DMA_STREAM); */
    /*
     使能前检测DMA是否被禁止。请注意，在使用相同数据流的多次传输时，这一步很有用：
      使能，然后禁止，然后使  能..... 在这种情况下，只有在数据传输结束的时候，DMA的禁止才有效，
     在确保使能位被硬件清除之前做确认是不可能的，如果DMA传输只做一次，这一步可以忽略。
    */
    DMA_Cmd(DMA2_Stream7, DISABLE);
    while (DMA_GetCmdStatus(DMA2_Stream7) != DISABLE)
    {
    }
    DMA_InitStructure.DMA_BufferSize = BufferSize;
    /* 配置DMA大小 */
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable;
    /* 在这个程序里面使能或者禁止都可以的 */
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
    /* 设置阀值 */
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single ;
    /* 设置内存为单字节突发模式 */
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
    /* 设置内存数据的位宽是字节 */
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    /* 使能地址自增 */
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;

    /* 设置DMA是正常模式 */
    DMA_InitStructure.DMA_PeripheralBaseAddr =(uint32_t) (&(USART1->DR)) ;
    /* 设置外设地址 */
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
```

```
        /* 设置外设为单字节突发模式 */
        DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;;
        /* 设置外设数据的位宽是字节 */
        DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
        /* 禁止外设地址自增 */
        DMA_InitStructure.DMA_Priority = DMA_Priority_High;
        /* 设置优先级 */
        /* 配置 TX DMA */
        DMA_InitStructure.DMA_Channel = DMA_Channel_4 ;
        /* 配置发送通道 */
        DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral ;
        /* 设置从内存到外设 */
        DMA_InitStructure.DMA_Memory0BaseAddr =(uint32_t)BufferSRC ;
        /* 设置内存地址 */
        DMA_Init(DMA2_Stream7,&DMA_InitStructure);
        /* 使能 DMA USART TX Stream */
        DMA_Cmd(DMA2_Stream7, ENABLE);
        /*
        检测DMA Stream 是否被正确的使能.如果DMA的参数配置错误了,
         那么DMA Stream   的使能位会被硬件清除掉, 从而使得传输停止比如FIFO的阀值
配置错误   */
        Timeout = TIMEOUT_MAX;
        while ((DMA_GetCmdStatus(DMA2_Stream7) != ENABLE) && (Timeout-- > 0))
        {
        }
        /* 检测是否超时 */
        if (Timeout == 0)
        {
            /* 处理错误 */

        }
    }
```

在main.c文件中添加如下代码:

```
    //printf("%s",helpstr);

    /* 使能 DMA clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);

    /* 使能 USART DMA TX 请求 */
    USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);

    USART_DMA_TxConfig((uint32_t *)helpstr, STRLEN);
   /* 等待数据传输完成, 这个函数可以不是用的, 直接用下面的DMA等待即可 */

    while(DMA_GetFlagStatus(DMA2_Stream7, DMA_FLAG_TCIF7)==RESET);
```

```
/* 清除DMA传输完成标志 */
DMA_ClearFlag(DMA2_Stream7,DMA_FLAG_TCIF7);
/* 清除串口传输完成标志 */
USART_ClearFlag(USART1,USART_FLAG_TC);
```

13.5 软件下载、测试验证

第 14 章

独立看门狗实验

按下 K1 按键,通过使得无法执行喂狗程序，从而造成系统复位。

14.1 实验原理与基础知识

嵌入式系统常常会工作在复杂的环境中，因此会受到来自电磁场，振动，温湿度等外部环境的干扰，造成各种寄存器和内存的数据混乱，会导致程序指针错误，不在程序区，取出错误的程序指令等，都会陷入死循环，程序的正常运行被打断，设备工作不正常，会造成整个系统的陷入停滞状态，发生不可预料的后果，为了实现无人值守状态下设备连续正常工作，于是看门狗机制孕育而生。

看门狗的原理在系统运行以后也就启动了看门狗的计数器，看门狗就开始自动计数，正常运行程序中会有一段喂狗的代码，如果到了一定的时间还没有喂狗，说明代码已经跑离轨道，那么看门狗计数器就会溢出从而引起看门狗中断，系统复位。

STM32F4xx 内置两个看门狗，独立看门狗和窗口看门狗（下一章节讲解），独立看门狗(IWDG)由专用的低速时钟(LSI)驱动，即使主时钟发生故障它也仍然有效。独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 32Khz，而是在 15~47Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 32Khz 的频率来计算，所以它适合应用于那些需要看门狗作为一个在主程序之外，能够完全独立工作，并且对时间精度要求较低的场合。独立看门狗的功能框图

独立看门狗寄存器

关键字寄存器(IWDG_KR)　　Key register

偏移地址：0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																KEY[15:0]															
																w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:16 保留，必须保持复位值。

位 15:0 **KEY[15:0]：**键值 (Key value)（只写位，读为 0000h）

必须每隔一段时间便通过软件对这些位写入键值 AAAAh，否则当计数器计数到 0 时，看门狗会产生复位。

写入键值 5555h 可使能对 IWDG_PR 和 IWDG_RLR 寄存器的访问

写入键值 CCCCh 可启动看门狗（选中硬件看门狗选项的情况除外）

预分频器寄存器(IWDG_PR)　Prescaler register

偏移地址：0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																													PR[2:0]		
																													rw	rw	rw

位 31:3 保留，必须保持复位值。

位 2:0 **PR[2:0]：**预分频器 (Prescaler divider)

这些位受写访问保护，通过软件设置这些位来选择计数器时钟的预分频因子。若要更改预分频器的分频系数，IWDG_SR 的 PVU 位必须为 0。

000：4 分频
001：8 分频
010：16 分频
011：32 分频
100：64 分频
101：128 分频
110：256 分频
111：256 分频

注意：读取该寄存器会返回 VDD 电压域的预分频器值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 PVU 位为 0 时，从寄存器读取的值才有效。

重载寄存器(IWDG_RLR)　Reload register

偏移地址：0x08

状态寄存器(IWDG_SR)　Status register

偏移地址：0x0C

14.2 电路设计

14.3 库函数和数据结构

独立看门狗相关的库函数操作函数在文件 stm32f4xx_iwdg.c 和对应的头文件 stm32f4xx_iwdg.h 中

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																				RL[11:0]											
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:12 保留，必须保持复位值。

位 11:0 **RL[11:0]：**看门狗计数器重载值 (Watchdog counter reload value)

这些位受写访问保护，这个值由软件设置，每次对 IWDR_KR 寄存器写入值 AAAAh 时，这个值就会重装载到看门狗计数器中。之后，看门狗计数器便从该装载的值开始递减计数。超时周期由该值和时钟预分频器共同决定。

若要更改重载值，IWDG_SR 中的 RVU 位必须为 0。

注意：读取该寄存器会返回 VDD 电压域的重载值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 RVU 位为 0 时，从寄存器读取的值才有效。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																														RVU	PVU
																														r	r

位 31:2 保留，必须保持复位值。

位 1 **RVU：**看门狗计数器重载值更新 (Watchdog counter reload value update)

可通过硬件将该位置 1 以指示重载值正在更新。当在 V_{DD} 电压域下完成重载值更新操作后（需要多达 5 个 RC 40 kHz 周期），会通过硬件将该位复位。

重载值只有在 RVU 位为 0 时才可更新。

位 0 **PVU：**看门狗预分频器值更新 (Watchdog prescaler value update)

可通过硬件将该位置 1 以指示预分频器值正在更新。当在 V_{DD} 电压域下完成预分频器值更新操作后（需要多达 5 个 RC 40 kHz 周期），会通过硬件将该位复位。

预分频器值只有在 PVU 位为 0 时才可更新。

```
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
```

取消寄存器写保护函数,向 IWDG_KR 写入 0X5555

```
void IWDG_SetPrescaler(uint8_t IWDG_Prescaler);
```

设置看门狗的分频系数的函数

```
void IWDG_SetReload(uint16_t Reload);
```

设置看门狗的重装载值的函数

```
IWDG_ReloadCounter();
```

重载计数值喂狗函数，向 IWDG_KR 写入 0XAAAA

```
IWDG_Enable();
```

启动看门狗函数 向 IWDG_KR 写入 0XCCCC

14.4 软件设计、开发

在工程中添加iwdg.c文件和iwdg.h 文件，所有 IWDG 相关的驱动代码和定义都在这两个文件中。

iwdg.c文件代码如下：

iwdg.h代码如下：

main.c测试代码如下：

14.5 软件下载、测试验证

第 15 章

窗口看门狗实验

和独立看门狗不一样的地方是，窗口看门狗可以触发中断，本实验我们将使用窗口看门狗的中断功能来喂狗。

15.1 实验原理与基础知识

窗口看门狗(WWDG) 时钟由APB1 时钟经预分频后提供，通过可配置的时间窗口来检测应用程序非正常的过迟或过早的操作。WWDG 最适合那些要求看门狗在精确计时窗口起作用的应用程序

窗口看门狗通常被用来监测，由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在T6 位变成0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个MCU 复位。如果在递减计数器达到窗口寄存器值之前刷新控制寄存器中的7 位递减计数器值，也会产生MCU 复位。这意味着必须在限定的时间窗口内刷新计数器。

窗口看门狗框图

窗口看门狗时序图

窗口看门狗的 3 个寄存器

控制寄存器(WWDG_CR)　　Status register

超时值的计算公式如下：

$$t_{WWDG} = t_{PCLK1} \times 4096 \times 2^{WDGTB} \times (t[5:0]+1) \quad (ms)$$

其中：

t_{WWDG}：WWDG 超时

t_{PCLK1}：APB1 时钟周期，以 ms 为测量单位

有关 T_{WWDG} 的最小值和最大值，请参见下表。

偏移地址：0x00

配置寄存器(WWDG_CFR)　　Configuration register

偏移地址：0x04

状态寄存器(WWDG_SR)　　Status register

偏移地址：0x08

15.2 电路设计

15.3 库函数和数据结构

窗口看门狗库函数相关源码和定义分布在文件 stm32f4xx_wwdg.c 文件和头文件 stm32f4xx_wwdg.h 中

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);
```

WWDG 不同于 IWDG，IWDG 有自己独立的 32Khz 时钟，不存在使能问题。而 WWDG使用的是 PCLK1 的时钟，需要先使能时钟。

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								WDGA	T[6:0]						
								rs	rw						

位 31:8 保留，必须保持复位值。

位 7 **WDGA**：激活位 (Activation bit)

此位由软件置 1，只有复位后才由硬件清零。当 WDGA = 1 时，看门狗可产生复位。

0:禁止看门狗

1:使能看门狗

位 6:0 **T[6:0]**：7 位计数器 (MSB 到 LSB)

这些位用来存储看门狗计数器的值。它每隔 (4096 x 2^{WDGTB}) PCLK1 个周期递减一次。当它从 0x40 滚动到 0x3F（T6 清零）时会产生复位。

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						EWI	WDGTB[1:0]		W[6:0]						
						rs	rw		rw						

位 31:10 保留，必须保持复位值。

位 9 **EWI**：提前唤醒中断 (Early wakeup interrupt)

置 1 后，只要计数器值达到 0x40 就会产生中断。此中断只有在复位后才由硬件清零。

位 8:7 **WDGTB[1:0]**：定时器时基 (Timer base)

可按如下方式修改预分频器的时基：

00：CK 计数器时钟 (PCLK1 div 4096) 分频器 1

01：CK 计数器时钟 (PCLK1 div 4096) 分频器 2

10:CK 计数器时钟 (PCLK1 div 4096) 分频器 4

11:CK 计数器时钟 (PCLK1 div 4096) 分频器 8

位 6:0 **W[6:0]**：7 位窗口值 (7-bit window value)

这些位包含用于与递减计数器进行比较的窗口值。

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															EWIF
															rc_w0

位 31:1保留，必须保持复位值。

位 0 **EWIF**：提前唤醒中断标志 (Early wakeup interrupt flag)

当计数器值达到 0x40 时此位由硬件置 1。它必须由软件通过写入 0 来清零。写入 1 不起作用。如果不使能中断，此位也会被置 1。

```
void WWDG_SetWindowValue(uint8_t WindowValue);
```

设置窗口值的函数,一个参数为窗口值。

```
void WWDG_SetPrescaler(uint32_t WWDG_Prescaler);
```

设置分频数的函数,一个参数就是分频值。

```
WWDG_EnableIT(); //开启窗口看门狗中断
```

开启 WWDG 中断的函数。

```
void WWDG_Enable(uint8_t Counter);
```

设置计数器初始值并使能看门狗函数

```
void WWDG_SetCounter(uint8_t Counter);
```

独立的设置计数器值的函数

15.4 软件设计、开发

在工程中添加wwdg.c文件和wwdg.h 文件，所有 WWDG 相关的驱动代码和定义都在这两个文件中。

15.5 软件下载、测试验证

第 16 章

ADC实验

ADC即（Analog-to-Digital Converter）的缩写，模/数转换器或者模拟/数字转换器，意为将连续变量的模拟信号转换为离散的数字信号的器件。

16.1 实验原理与基础知识

stm32f407系列ADC支持DMA(Direct Memory Access，直接内存存取)模式，利用DMA对数据进行传输，从而提高ADC效率，在ADC每次转换完之后，由DMA控制器把转换值从ADC数据寄存器中转移到内存，即把数据从外设直接写入内存，把得到数据加以滤波算法分析，最终将得到结果通过串口打印显示。

对于stm32f407，共有两个DMA控制器，总共有16个stream（流），每个DMA控制器包括8个stream，每个stream多达8个channel（通道），本案例选择DMA2stream0->Channel_0。

stm32f407的每个ADC有19个复用channel（通道），可以检测16个外部信号源和两个内部信号源和电池信号。可以对应电路原理图和stm32f07数据手册进行选择，这里选择使用ADC1的8通道。

16.2 电路设计

本案例使用的ADC硬件为电位器，通过旋转电位器的螺丝得到不同的模拟信号，电位器连接到CPU的PB0，该引脚支持ADC1和ADC2的channel_8，注意电位器与温湿度传感器复用ADC引脚，这里需将J27到1、2用跳线冒连接。

16.3 库函数和数据结构

ADC基本配置直接相关的几个固件库函数。这些函数和定义主要分布在 stm32f4xx_adc.h 和 stm32f4xx_adc.c 文件中。

使能ADC时钟总线：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

ADC初始化相关数据结构：

```
ADC_InitTypeDef       ADC_InitStructure;
ADC_CommonInitTypeDef ADC_CommonInitStructure;
```

初始化ADC外设接口：

```
ADC_CommonInit(&ADC_CommonInitStructure);
```

初始化ADC配置：

```
ADC_Init(ADC1, &ADC_InitStructure);
```

规则通道设置，1个channel1，转换时间为28个周期

```
ADC_RegularChannelConfig(ADC1, ADC_Channel_8, 1, ADC_SampleTime_28Cycles);
```

启用ADC1的DMA请求后开始数据转移：

```
ADC_DMARequestAfterLastTransferCmd(ADC1, ENABLE);
```

启用ADC1到DMA请求：

```
ADC_DMACmd(ADC1, ENABLE);
```

启用ADC1外设：

```
ADC_Cmd(ADC1, ENABLE);
```

16.4 软件设计、开发

在工程中添加adc.c 文件和adc.h 文件，ADC相关的函数和定义都在这两个文件中。

1）使能ADC、DMA和GPIO到时钟总线。

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2 | RCC_AHB1Periph_GPIOB, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

2）定义数据结构变量

```
ADC_InitTypeDef       ADC_InitStructure;
ADC_CommonInitTypeDef ADC_CommonInitStructure;
DMA_InitTypeDef       DMA_InitStructure;
GPIO_InitTypeDef      GPIO_InitStructure;
```

3）配置DMA，选择DMA2的stream0

```
DMA_InitStructure.DMA_Channel = DMA_Channel_0;
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)ADC1_DR_ADDRESS;
DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)&ADC1ConvertedValue;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
DMA_InitStructure.DMA_BufferSize = 1;
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA2_Stream0, &DMA_InitStructure);
DMA_Cmd(DMA2_Stream0, ENABLE);
```

4）配置ADC对应GPIO为Analog Mode

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

5）初始化ADC1

```
ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
ADC_InitStructure.ADC_ScanConvMode = DISABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfConversion = 1;
ADC_Init(ADC1, &ADC_InitStructure);
```

6）开启ADC1

```
/* ADC1 regular channel_8 configuration ************************************/
ADC_RegularChannelConfig(ADC1, ADC_Channel_8, 1, ADC_SampleTime_28Cycles);
/* Enable DMA request after last transfer (Single-ADC mode) */
ADC_DMARequestAfterLastTransferCmd(ADC1, ENABLE);
/* Enable ADC1 DMA */
ADC_DMACmd(ADC1, ENABLE);
/* Enable ADC1 */
ADC_Cmd(ADC1, ENABLE);
```

7）编写滤波函数，让ADC采样十次，去掉最高和最低，再取剩下8次平均值

```
uint32_t ADC_filter(void)
{
    uint32_t i,min,max,sum;
    uint32_t arr[10]={0};
    sum=0;
    for(i=0;i<10;i++){
        arr[i]=ADC1ConvertedValue *3300/0xFFF;
        delay_ms(10);
        sum += arr[i];
    }

    min=arr[0];
    max=arr[0];
    for(i=0;i<10;i++){
        if(arr[i]<min)
            min=arr[i];
        if(arr[i]>max)
            max=arr[i];
    }

    return (sum-min-max)>>3;
}
```

8）串口配置（略），请参考实验

9）编写main函数

```
#define ADC1_DR_ADDRESS     ((uint32_t)0x4001204C)
__IO uint16_t ADC1ConvertedValue = 0;

int main(void)
{
    float adc_value;

    /*uart1 configure*/
    uart_config();

    /* ADC1 configuration */
    ADC1_CH8_DMA_Config();

    /* Start ADC1 Software Conversion */
    ADC_SoftwareStartConv(ADC1);

    while (1)
    {
        adc_value=ADC_filter()*1.0/1000;
        printf("value=%f\r\n",adc_value);
    }
```

```
}
```

16.5 软件下载、测试验证

第 17 章

IIC实验

IIC（Inter－Integrated Circuit）总线是由PHILIPS公司开发的两线式串行总线，用于连接微控制器及其外围设备。是微电子通信控制领域广泛采用的一种总线标准。

17.1 实验原理与基础知识

Elink407系列开发板通过IIC总线外接EEPROM设备（AT24C64），即带电可擦写可编程只读存储器，它是用户可更改的只读存储器，其可通过高于普通电压的作用来擦除和重编程（重写）。EEPROM不需从计算机中取出即可修改，EEPROM是一种特殊形式的闪存，其应用通常是在接口卡等嵌入式系统中，用来存放硬件设置数据。

AT24C64是低工作电压的64Kbit位串行电可擦除只读存储器，内部组织为8192个字节，每个字节8位，该芯片被广泛应用于低电压及低功耗的工商业领域。

AT25C64使用两线串行接口和微处理器进行通信，遵循IIC总线协议。

IIC协议术语：

发送器：发送数据到总线的器件；

接收器：从总线接收数据的器件；

主机：启动数据传送并产生时钟信号的设备；

从机：被主机寻址的器件；

多主机：同时有多于一个主机尝试控制总线但不破坏传输；

主模式：用I2CNDAT支持自动字节计数的模式；位I2CRM,I2CSTT,I2CSTP控制数据的接收和发送；

从模式：发送和接收操作都是由I2C模块自动控制的；

仲裁：是一个在有多个主机同时尝试控制总线但只允许其中一个控制总线并使传输不被破坏的过程；

同步：两个或多个器件同步时钟信号的过程 。

IIC总线——起始和停止信号：

SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件；

SCL 线是高电平时，SDA 线由低电平向高电平切换，这个情况表示停止条件。

起始和停止条件一般由主机产生，总线在起始条件后被认为处于忙的状态起始和停止条件起始和停止条件，在停止条件的某段时间后总线被认为再次处于空闲状态。如果产生重复起始条件而不产生停止条件，总线会一直处于忙的状态，此时的起始条件（S）和重复起始条件（Sr）在功能上是一样的。

IIC总线——数据传输格式：

字节格式

发送到SDA 线上的每个字节必须为8 位，每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位。首先传输的是数据的最高位（MSB），如果从机要完成一些其他功能后（例如一个内部中断服务程序）才能接收或发送下一个完整的数据字节，可以使时钟线SCL 保持低电平，迫使主机进入等待状态，当从机准备好接收下一个数据字节并释放时钟线SCL 后数据传输继续。

应答响应

数据传输必须带响应，相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间发送器释放SDA 线（高）。

在响应的时钟脉冲期间，接收器必须将SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平。

如果从机接收器响应了从机地址，但是在传输了一段时间后不能接收更多数据字节，主机必须再一次终止传输。这个情况用从机在第一个字节后没有产生响应来表示。从机使数据线保持高电平，主机产生一个停止或重复起始条件。

如果传输中有主机接收器，它必须通过在从机发出的最后一个字节时产生一个响应，向从机发送器通知数据结束。从机发送器必须释放数据线，允许主机产生一个停止或重复起始条件。

17.2　电路设计

本案例使用的IIC器件AT24C64封装类型为SOP8，其数据线SCL和时钟线SDA分别与stm32407的PB6、PB7相连接。

17.3　库函数和数据结构

IIC总线基本配置直接相关的几个固件库函数，这些函数的定义主要分布在stm32f4xx_i2c.h和 stm32f4xx_i2c.c 文件中。

发送起始信号：

```
I2C_GenerateSTART(I2C1, ENABLE);
```

发送停止信号：

```
I2C_GenerateSTOP(I2C1, ENABLE);
```

读取I2C指定寄存器到值：

```
I2C_ReadRegister(I2C1, I2C_Register);
```

获取状态标志：

```
I2C_GetFlagStatus(I2C1, I2C_FLAG)
```

开启I2C响应：

```
I2C_AcknowledgeConfig(I2C1, ENABLE)
```

17.4 软件设计、开发

在工程中添加iic.c 文件和iic.h 文件，IIC相关的函数和定义都在这两个文件中。

1）IIC的GPIO配置。

```
void I2C_Configuration(void)
{
    I2C_InitTypeDef  I2C_InitStructure;
    GPIO_InitTypeDef  GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(I2C_SDA_GPIO_CLK,ENABLE);
    RCC_AHB1PeriphClockCmd(I2C_SCL_GPIO_CLK,ENABLE);

    RCC_APB1PeriphClockCmd(I2C_CLK,ENABLE);

    GPIO_PinAFConfig(I2C_SDA_GPIO_PORT, I2C_SDA_SOURCE, I2C_SDA_AF);
    GPIO_PinAFConfig(I2C_SCL_GPIO_PORT, I2C_SCL_SOURCE, I2C_SCL_AF);

    GPIO_InitStructure.GPIO_Pin =  I2C_SDA_PIN | I2C_SCL_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //PB6和PB7设置为开漏输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    I2C_DeInit(I2C);
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1 = 0x0;//当I2C出于从模式时,自身的地址
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
   I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = I2C_SPEED;
    I2C_Init(I2C, &I2C_InitStructure);

    I2C_Cmd(I2C, ENABLE);
    I2C_AcknowledgeConfig(I2C, ENABLE);
}
```

2) 读数据操作 uint8_t I2C_Read(I2C_TypeDef I2Cx,uint8_t I2C_Addr,uint16_t addr,uint8_t buf,uint16_t num) { if(num==0){ return 1; } while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

```
    I2C_AcknowledgeConfig(I2Cx, ENABLE);

    I2C_GenerateSTART(I2Cx, ENABLE);
    while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    I2C_Send7bitAddress(I2Cx,  I2C_Addr, I2C_Direction_Transmitter);
   while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

    //I2C_SendData(I2Cx, addr);
    I2C_SendData(I2C1, addr/256);  //高位
    I2C_SendData(I2C1, addr%256);  //低位
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

    I2C_GenerateSTART(I2Cx, ENABLE);
    while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

    I2C_Send7bitAddress(I2Cx, I2C_Addr, I2C_Direction_Receiver);
    while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED))
    {}

    while (num)
    {
        if(num==1)
        {
            I2C_AcknowledgeConfig(I2Cx, DISABLE);
            I2C_GenerateSTOP(I2Cx, ENABLE);
        }

        while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED))
        {}
        *buf = I2C_ReceiveData(I2Cx);
        buf++;
        /* Decrement the read bytes counter */
        num--;
    }

    I2C_AcknowledgeConfig(I2Cx, ENABLE);

    return 0;
}
```

3) 写数据操作

```
 //写一个字节
uint8_t I2C_WriteOneByte(I2C_TypeDef *I2Cx,uint8_t I2C_Addr,uint16_t addr,uint8_t value)
 {
     //判断BUSY是否为1,BUSY=1总线处于忙状态,如果BUSY=1则等待一直等到BUSY=0
     while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY)){
         if((I2CTimeout--) == 0) return I2C_TIMEOUT_UserCallback();
     }

       /* Send STRAT condition */
     I2C_GenerateSTART(I2C1, ENABLE);

     /* Test on EV5 and clear it */
     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

     /* Send EEPROM address for write */
     I2C_Send7bitAddress(I2C1, I2C_Addr, I2C_Direction_Transmitter);

     /* Test on EV6 and clear it */
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

     /* Send the EEPROM's internal address to write to */
     I2C_SendData(I2C1, addr/256);  //高位
     I2C_SendData(I2C1, addr%256);  //低位
     /* Test on EV8 and clear it */
     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

     /* Send the byte to be written */
     I2C_SendData(I2C1, value);
     /* Test on EV8 and clear it */
     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

     /* Send STOP condition */
     I2C_GenerateSTOP(I2C1, ENABLE);

     I2C_EE_WaitEepromStandbyState();

     I2C_delay(1000);

     return 0;
 }


 //写多个字节
uint8_t I2C_Write(I2C_TypeDef *I2Cx,uint8_t I2C_Addr,uint16_t addr,uint8_t *buf,uint16_t num)
 {
```

```
    uint8_t err=0;
    while(num--)
    {
        if(I2C_WriteOneByte(I2Cx, I2C_Addr,addr++,*buf))
        {
            err++;
        }
        buf++;
    }
    if(err)
        return 1;
    else
        return 0;
}
```

4）擦除数据

```
void I2C_Erase(void)
{
    uint16_t i;
    for (i = 0; i <8*1024; i++)
    {
        I2C_WriteOneByte(I2C,ADDR_24L64,i,0xFF);
    }
}
```

5）编写main函数

```
int main(void)
{
    uint16_t Addr;
    uint8_t WriteBuffer[256]={0};
    uint8_t ReadBuffer[256]={0};
    uart_config();
    printf("\r\n**********************r\n");

    I2C_Configuration();

    for(Addr=0; Addr<256; Addr++){
        WriteBuffer[Addr]=Addr+100;  /* 填充WriteBuffer */
    }

    //全部擦出，每个字节置0xff
    //I2C_Erase();
    //printf("Erase ok\r\n");

    /* 开始向EEPROM写数据 */
```

```
    printf("\r\n EEPROM 24C64 Write Test \r\n");

    for(Addr=0;Addr<sizeof(WriteBuffer);Addr++){
        printf("%d ",WriteBuffer[Addr]);
    }
    printf("\r\n");

    I2C_Write(I2C,ADDR_24L64,0x0,WriteBuffer,sizeof(WriteBuffer) );
    printf("\r\n EEPROM 24C64 Write Test OK \r\n");

    /* EEPROM读数据 */
    printf("\r\n EEPROM 24C64 Read Test \r\n");
    I2C_Read(I2C,ADDR_24L64,0x0,ReadBuffer,sizeof(ReadBuffer) );

    for(Addr=0;Addr<sizeof(ReadBuffer);Addr++){
        printf("%d ",ReadBuffer[Addr]);
    }
    printf("\r\n");

    if(strcmp((const char *)ReadBuffer,(const char *)WriteBuffer)==0){
        printf("\r\n EEPROM 24C64 Read Test ok \r\n");
    }

    while (1)
    {
    }
}
```

17.5 软件下载、测试验证

第 18 章

SPI实验

串行外设接口SPI（Serial Peripheral Interface）是一种高速的、全双工、同步的通信总线，SPI总线系统可直接与各个厂家生产的多种标准外围器件直接连接，该接口一般使用4条线：串行时钟线（SCLK）、主机输入/从机输出数据线MISO、主机输出/从机输入数据线MOSI和低电平有效的从机选择线NSS。

18.1 实验原理与基础知识

Elink407系列开发板通过SPI外接flash设备（SST25VF016B）和网络设备（EMW1062），本案例使用flash，需将J21的1、2引脚用条线帽短接。

接口信号

（1）MOSI　主器件数据输出，从器件数据输入

（2）MISO　主器件数据输入，从器件数据输出

（3）SCLK　时钟信号，由主器件产生，最大为fPCLK/2，从模式频率最大为fCPU/2

（4）NSS　从器件使能信号，由主器件控制，有的SPI器件会标注为CS(Chip select)

在点对点的通信中，SPI接口不需要进行寻址操作，且为全双工通信，显得简单高效。多个从器件硬件连接示意图在多个从器件的系统中，每个从器件需要独立的使能信号，硬件上比I2C系统要稍微复杂一些。

SPI接口在内部硬件实际上是两个简单的移位寄存器，传输的数据为8位，在主器件产生的从器件使能信号和移位脉冲下，按位传输，高位在前，低位在后。如下图所示，在SCLK的上升沿上数据改变，同时一位数据被存入移位寄存器。

18.2 电路设计

本案例使用的SPI器件SST25VF016B封装类型为SOP8，引脚与处理器GPIO连接的对应关系如下：

PA15——SPI1_CS

PA5——SPI1_SCK

PB4——SPI1_MISO

PB5——SPI1_MOSI

18.3　库函数和数据结构

SPI总线基本配置直接相关的几个固件库函数，这些函数的定义主要分布在stm32f4xx_spi.h和 stm32f4xx_spi.c 文件中。

使能SPI1时钟总线：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
```

定义片选宏函数，信号为低时生效：

```
#define SST25V_CS_LOW()              GPIO_ResetBits(GPIOA, GPIO_Pin_15)
#define SST25V_CS_HIGH()             GPIO_SetBits(GPIOA, GPIO_Pin_15)
```

等待SPI1的发送缓冲区为空：

```
SPI_GetFlagStatus(SPI1, SPI_FLAG_TXE);
```

发送数据字节：

```
SPI_SendData(SPI1, byte);
```

等待SPI1设备完成接收数据：

```
SPI_GetFlagStatus(SPI1, SPI_FLAG_RXNE);
```

接受读到的数据：

```
SPI_ReceiveData(SPI1);
```

开启SPI1：

```
SPI_Cmd(SPI1, ENABLE);
```

18.4 软件设计、开发

在工程中添加spiflash.c 文件和spiflash.h 文件，SPI flash相关的函数和定义都在这两个文件中。

1）时钟总线配置

```
void RCC_Configuration(void)
{
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
 RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA | RCC_AHB1Periph_GPIOB, ENABLE);
}
```

2） SPI的GPIO配置。

```
void GPIO_Configuration(void)
{
    //PA5--SPI1_SCK  PB4--SPI1--MISO PB5--SPI1--MOSI  PA15--SPI1CS
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 ;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 |GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource5,GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource4,GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource5,GPIO_AF_SPI1);

    //PA15--SPI1_CS
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    //GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

3）配置SPI1总线为master模式

```
void SPI_configuration()  //SPI1 configuration
{
    SPI_InitTypeDef  SPI_InitStructure;
```

```
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(SPI1, &SPI_InitStructure);
    SPI_Cmd(SPI1, ENABLE);  //Enable SPI1
}
```

4）读取flash数据

```
//SPI发送一个字节数据
u8 SPI_Flash_SendByte(u8 byte)
{
    while(SPI_GetFlagStatus(SPI1, SPI_FLAG_TXE) == RESET);     //等待SPI1的
Tx buffer为空
   SPI_SendData(SPI1, byte);                                //发送数据字节byte
    while(SPI_GetFlagStatus(SPI1, SPI_FLAG_RXNE) == RESET);    //等待SPI1完
成接收数据
   return SPI_ReceiveData(SPI1);                            //返回读到的数据
}

//从ReadAddr读取一个字节数据
u8 SST25V_ByteRead(u32 ReadAddr)
{
    u8 Temp = 0;
    SST25V_CS_LOW();
    SPI_Flash_SendByte(Read_Data);
    SPI_Flash_SendByte((ReadAddr & 0xFF0000) >> 16);  //24位地址
    SPI_Flash_SendByte((ReadAddr& 0xFF00) >> 8);
    SPI_Flash_SendByte(ReadAddr & 0xFF);

    Temp = SPI_Flash_ReceiveByte();
    SST25V_CS_HIGH();
    return Temp;
}

//从ReadAddr读取NumByteToRead字节数据
void SST25V_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
{
    SST25V_CS_LOW();
    SPI_Flash_SendByte(Read_Data);
```

```
    SPI_Flash_SendByte((ReadAddr & 0xFF0000) >> 16);
    SPI_Flash_SendByte((ReadAddr& 0xFF00) >> 8);
    SPI_Flash_SendByte(ReadAddr & 0xFF);
    while(NumByteToRead--)
    {
        *pBuffer = SPI_Flash_ReceiveByte();
        pBuffer++;
    }
    SST25V_CS_HIGH();
}
```

5) 写数据到flash

```
//写入一个字节数据
void SST25V_ByteWrite(u8 Byte, u32 WriteAddr)
{
    SST25V_WriteEnable();
    SST25V_CS_LOW();
    SPI_Flash_SendByte(Byte_Program);
    SPI_Flash_SendByte((WriteAddr & 0xFF0000) >> 16);
    SPI_Flash_SendByte((WriteAddr & 0xFF00) >> 8);
    SPI_Flash_SendByte(WriteAddr & 0xFF);

    SPI_Flash_SendByte(Byte);
    SST25V_CS_HIGH();
    SST25V_WaitForWriteEnd();
}
//以WriteAddr为起始地址,写入ByteLength个Byte数据(相同的Byte)
void SST25V_WriteBytes(u8 Byte, u32 WriteAddr,u32 ByteLength)
{
    while(ByteLength--)
    {
        SST25V_ByteWrite(Byte,WriteAddr);
        WriteAddr++;
    }
}

//等待写操作完成
void SST25V_WaitForWriteEnd(void)
{
    u8 FLASH_Status = 0;
    SST25V_CS_LOW();
    do
    {
        FLASH_Status = SST25V_ReadStatusRegister();

    } while((FLASH_Status & 0x01) == SET);
```

```
    SST25V_CS_HIGH();
}
```

6）格式化flash

```
void SST25V_ChipErase(void)            //擦出整个芯片数据空间
{
    SST25V_WriteEnable();
    SST25V_CS_LOW();
    SPI_Flash_SendByte(ChipErace);
    SST25V_CS_HIGH();
    SST25V_WaitForWriteEnd();
}
```

7）编写main函数

```
int main()
{
    uart_config();
    printf("test usart!\r\n");

    Setup_System();
    SST25V_Init();
    SST25V_ChipErase();

    //test flash 1,擦出读测试
    DataByte = SST25V_ByteRead(0x000000);
    printf("data1=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x000010);
    printf("data2=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x000100);
    printf("data4=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x001000);
    printf("data5=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x010000);
    printf("data6=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x100000);
    printf("data7=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x600000);
    printf("data8=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xF00000);
    printf("data9=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xFFFFFF);
    printf("data10=%x\r\n",DataByte);

    //test flash 2, 向地址0xfff000写入0x1000个字符0x72
    SST25V_AAI_WriteBytes(0x72, 0xFFF000,0x001000);
```

```
    DataByte = SST25V_ByteRead(0x000000);
    printf("data11=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x000001);
    printf("data12=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0x000002);
    printf("data12=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xFFF000);
    printf("data13=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xFFF0EE);
    printf("data14=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xFFFEEE);
    printf("data15=%x\r\n",DataByte);
    DataByte = SST25V_ByteRead(0xFFFFFF);

    //test flash 3, 向地址0xfff000写字符串
    SST25V_ChipErase();
    SST25V_AAI_BufferProgram(Tx_Buffer,0xFFF000,BufferSize);
    printf("tx_buffer=%s\r\n",Tx_Buffer);
    SST25V_BufferRead(Rx_Buffer,0xFFF000,BufferSize);
    printf("rx_buffer=%s\r\n",Rx_Buffer);

    DataByte = SST25V_ByteRead(0xFFEEEE);
    printf("data16=%d\r\n",DataByte);
    return 0;
}
```

18.5 软件下载、测试验证

第 19 章

RTC实验

RTC是一个实时时钟是一个独立的定时器，RTC模块拥有一组连续计数的计数器，在相应软件配置下，可以提供时钟、日历的功能。修改计数器的值可以重新设置系统当前的时间和日期

RTC可以用来设置实时时钟，并产生秒中断或闹钟中断。在芯片有电源供电的情况下，系统掉电后仍能保持时钟计时。

19.1 实验原理与基础知识

RTC设置需要涉及对电源（PWR）、备份区域（BKP）和RCC部分寄存器的改写，对于RTC的使用，首先我们要判断是否是第一次使用RTC，如果是第一次使用首先要对RTC进行相应的配置，并初始化其时间。如果不是第一次使用，那么我们就无需再设置时间的初始值以及对RTC进行相应的配置，只需让RTC计数器继续计数就可以了。

RTC的时钟源可以选择内部低速时钟LSI、外部低速时钟LSE和时钟HSE，内部时钟LSI不够准确，本案例选择外部时钟LSE。

19.2 电路设计

19.3 库函数和数据结构

RTC配置直接相关的几个固件库函数，这些函数的定义主要分布在stm32f4xx_rtc.h 和stm32f4xx_rtc.c 文件中。

使能电源时钟总线：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR , ENABLE);
```

选择rtc晶振（外部晶振LSE）：

```
RCC_LSEConfig(RCC_LSE_ON);//使用外部晶振
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);//选择RTC时钟源
```

开启rtc时钟：

```
RCC_RTCCLKCmd(ENABLE);
```

rtc时间数据结构：

```
typedef struct
{
  uint8_t RTC_Hours;
  uint8_t RTC_Minutes;
  uint8_t RTC_Seconds;
  uint8_t RTC_H12;
}RTC_TimeTypeDef;
```

rtc日期数据结构：

```
typedef struct
{
  uint8_t RTC_WeekDay;
  uint8_t RTC_Month;
  uint8_t RTC_Date;
  uint8_t RTC_Year;
}RTC_DateTypeDef;
```

rtc闹钟数据结构：

```
typedef struct
{
  RTC_TimeTypeDef RTC_AlarmTime;
  uint32_t RTC_AlarmMask;

  uint32_t RTC_AlarmDateWeekDaySel;
  uint8_t RTC_AlarmDateWeekDay;
}RTC_AlarmTypeDef;
```

读取rtc日期和时间：

```
RTC_GetTime(RTC_Format_BCD,&RTC_TimeStructure); //读日期
RTC_GetDate(RTC_Format_BCD,&RTC_DateStructure); //读时间
```

设置闹钟报警时间：

```
RTC_SetAlarm(RTC_Format_BCD,RTC_Alarm_A,&RTC_AlarmStructure);
```

获取rtc闹钟时间：

```
RTC_GetAlarm(RTC_Format_BCD,RTC_Alarm_A, &RTC_AlarmStruct);
```

开启rtc闹钟：

```
RTC_AlarmCmd(RTC_Alarm_A,ENABLE);
```

19.4 软件设计、开发

在工程中添加rtc.c 文件和rtc.h 文件，RTC相关的函数和定义都在这两个文件中。

1）配置rtc实时时钟

```
void rtc_config(void)
{
    /* RTC_ReadBackupRegister(RTC_BKP_DR0) */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR , ENABLE);//打开PWR时钟

    /* Allow access to BKP Domain */
    PWR_BackupAccessCmd(ENABLE);//打开配置寄存器权限

    /* Clear Wakeup flag */
    PWR_ClearFlag(PWR_FLAG_WU);

    /* Reset Backup Domain */
    RCC_BackupResetCmd(ENABLE);
    RCC_BackupResetCmd(DISABLE);

    /* Enable the LSE */
    RCC_LSEConfig(RCC_LSE_ON);//使用外部晶振
    while(RCC_GetFlagStatus(RCC_FLAG_LSERDY)==RESET){}//等待外部晶振准备好

    /* Select the RTC Clock Source */
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);//选择RTC时钟源

    /* Enable the RTC Clock */
    RCC_RTCCLKCmd(ENABLE);

      /* Wait for RTC APB registers synchronisation (needed after start-
up from Reset)*/
```

```
    RTC_WaitForSynchro();//等待RTC APB寄存器同步
}
```

2） 初始化设置rtc时间。

```
void rtc_time_init(void)
{
    RTC_TimeTypeDef RTC_TimeStructure;
    RTC_DateTypeDef RTC_DateStructure;
    RTC_InitTypeDef RTC_InitStructure;
    struct RtcTime time = {0x14,0x09,0x10,0x03,0x0,0x0,0x0};

    //if the time not set(first boot)
    if (RTC_ReadBackupRegister(RTC_BKP_DR0) != 0x32F2){
        //printf("init...\r\n");
        rtc_config();
        RTC_TimeStructure.RTC_H12 = RTC_H12_AM;
        RTC_TimeStructure.RTC_Hours = time.hour;
        RTC_TimeStructure.RTC_Minutes = time.min;
        RTC_TimeStructure.RTC_Seconds = time.sec;
        RTC_DateStructure.RTC_Year = time.year;
        RTC_DateStructure.RTC_Month = time.month;
        RTC_DateStructure.RTC_Date = time.date;
        RTC_DateStructure.RTC_WeekDay = time.week;

        // prescaler factor
        RTC_InitStructure.RTC_AsynchPrediv = 0x7F;
        RTC_InitStructure.RTC_SynchPrediv = 0XFF;

        // Specifies the RTC Hour Format
        RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;

        RTC_Init(&RTC_InitStructure);
        RTC_SetTime(RTC_Format_BCD,&RTC_TimeStructure);
        RTC_SetDate(RTC_Format_BCD,&RTC_DateStructure);

        //Writes a data in a specified RTC Backup data register
        RTC_WriteBackupRegister(RTC_BKP_DR0,0x32F2);
    }//the time already set(reboot)
    else{
        //printf("reset..\r\n");
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
        PWR_BackupAccessCmd(ENABLE);
        RTC_WaitForSynchro();
    }
}
```

3）配置rtc闹钟中断

```
void rtc_alarm_config(void)
{
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RTC_AlarmTypeDef RTC_AlarmStructure;

    /* 初始化外部中断17（参考数据手册） */
    EXTI_ClearITPendingBit(EXTI_Line17);
    EXTI_InitStructure.EXTI_Line=EXTI_Line17;
    EXTI_InitStructure.EXTI_Mode=EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger=EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_LineCmd=ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    /* 初始化RTC_Alarm_IRQn中断 */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_3);
    NVIC_InitStructure.NVIC_IRQChannel = RTC_Alarm_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2 ;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority=0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* RTC闹钟初始化， 每到5秒闹钟触发 */
    RTC_AlarmCmd(RTC_Alarm_A,DISABLE);
    RTC_AlarmStructure.RTC_AlarmDateWeekDay = 0x10;
  RTC_AlarmStructure.RTC_AlarmDateWeekDaySel = RTC_AlarmDateWeekDaySel_Date;

    /* 可选择屏蔽日期，星期和时间，如果全部屏蔽，每隔一秒产生一次闹钟 */
    //RTC_AlarmStructure.RTC_AlarmMask = RTC_AlarmMask_DateWeekDay;
    RTC_AlarmStructure.RTC_AlarmMask=RTC_AlarmMask_DateWeekDay|
            RTC_AlarmMask_Hours|RTC_AlarmMask_Minutes;

    /* 设置报警时间 */
    RTC_AlarmStructure.RTC_AlarmTime.RTC_H12 = RTC_H12_AM;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Hours = 0x0;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Minutes = 0x0;
    RTC_AlarmStructure.RTC_AlarmTime.RTC_Seconds = 0x5;
    RTC_SetAlarm(RTC_Format_BCD,RTC_Alarm_A,&RTC_AlarmStructure);

    /* 设置RTC中断源 */
    RTC_ITConfig(RTC_IT_ALRA,ENABLE);
    /* 使能RTC闹钟中断*/
    RTC_AlarmCmd(RTC_Alarm_A,ENABLE);
```

```
    /* 清除闹钟等待 */
    RTC_ClearFlag(RTC_FLAG_ALRAF);
    /* 清除中断线17等待，内存连接至闹钟*/
    EXTI_ClearITPendingBit(EXTI_Line17);

    get_alarm_time();
}
```

4）闹钟中断处理函数

```
void RTC_Alarm_IRQHandler(void)
{
    //检查是否指定的中断发生
    if(RTC_GetITStatus(RTC_IT_ALRA)!=RESET)
    {
        printf("alarm interrupt handler:");
        get_rtc_time();
        PWR_BackupAccessCmd(ENABLE);//必须有，否则RTC_ClearITPendingBit 不会
成功
        RTC_ClearITPendingBit(RTC_IT_ALRA);
        PWR_BackupAccessCmd(DISABLE);
        EXTI_ClearITPendingBit(EXTI_Line17);
    }
}
```

5）编写main函数

```
int main(void)
{
    /*uart1 configure*/
    uart_config();
    printf(" RTC example\r\n");

    /*Configuration delay millisecond precision*/
    SysTick_Configuration();

    /*configuration rtc*/
    rtc_time_init();

    /*configuration alarm time*/
    rtc_alarm_config();//必须在rtc初始化之后

    //get current time
    while(1){
        delay_ms(500);
        delay_ms(500);
        get_rtc_time();
```

```
            //get_alarm_time();
        }
    }
```

19.5 软件下载、测试验证

第 20 章

定时器实验

STM32F407的定时器包括TIME1和TIME8两个高级定时器，也有TIME2～TIME5，TIM9～TIM14通用定时器，还有TIME6和TIME7等基本定时器，总共达14个定时器之多。在案例将使用TIM3的定时器中断，设置定时器时间为1秒，并在定时器中断处理函数中通过串口输出到时消息。

20.1 实验原理与基础知识

STM32F4的通用定时器包含一个16位或32位自动重载计数器（CNT），该计数器由可编程预分频器（PSC）驱动。 STM32F4的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和PWM)。

使用定时器预分频器和RCC时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。 STM32F4的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM3的通用TIMx（TIM2～TIM5和TIM9～TIM14)定时器功能包括：

1)16位/32 位(仅TIM2和TIM5)向上、向下、向上/向下自动装载计数器（TIMx_CNT），注意：TIM9～TIM14只支持向上（递增）计数方式。 2)16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为1～65535之间的任意数值。 3）4个独立通道（TIMx_CH1～4，TIM9～TIM14最多2个通道），这些通道可以用来作为：

A．输入捕获

B．输出比较

C．PWM生成(边缘或中间对齐模式），注意：TIM9～TIM14不支持中间对齐模式 D．单脉冲模式输出

4）可使用外部信号（TIMx_ETR）控制定时器和定时器互连（可以用1个定时器控制另外一个定时器）的同步电路。 5）如下事件发生时产生中断/DMA（TIM9～TIM14不支持DMA）：

A．更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)

B．触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)

C．输入捕获

D．输出比较

E．支持针对定位的增量(正交)编码器和霍尔传感器电路（TIM9～TIM14不支持）

F．触发输入作为外部时钟或者按周期的电流管理（TIM9～TIM14不支持）

20.2 电路设计

20.3 库函数和数据结构

使能TIM3时钟： RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE)；初始化定时器参数,设置自动重装值，分频系数，计数方式等： voidTIM_TimeBaseInit(TIM_TypeDefTIMx, TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct)；第一个参数是确定是哪个定时器，这个比较容易理解。第二个参数是定时器初始化参数结构体指针，结构体类型为TIM_TimeBaseInitTypeDef，下面我们看看这个结构体的定义： typedef struct
{ uint16_t TIM_Prescaler;
uint16_t TIM_CounterMode;
uint16_t TIM_Period;
uint16_t TIM_ClockDivision;
uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
对于通用定时器只有前面四个参数有用，最后一个参数TIM_RepetitionCounter是高级定时器才有用的。 TIM_Prescaler：用来设置分频系数 TIM_CounterMode：用来设置计数方式。TIM_Period：设置自动重载计数周期值 TIM_ClockDivision：用来设置时钟分频因子。

设置TIM3_DIER允许更新中断： TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE)；

TIM3中断优先级设置： NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn; //定时器3中断 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x01; //抢占优先级1 NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x03; //响应优先级3 NVIC_InitStructure.NVIC_IRQChannelC NVIC_Init(&NVIC_InitStructure);//④初始化NVIC

开启TIM3定时器： TIM_Cmd(TIM3, ENABLE)；//使能TIM3外设

编写中断服务函数： //判断定时器3是否发生更新（溢出）中断 if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET){} //清除定时器TIMx的中断TIM_IT标志位 TIM_ClearITPendingBit(TIM3, TIM_IT_Update

20.4 软件设计、开发

1）初始化通用TIM3 //arr：自动重装值。 psc：时钟预分频数 //定时器到时时间计算方法:Tout=((arr+1)*(psc+1)) void timer_init(u16 arr,u16 psc) { TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure; NVIC_InitTypeDef NVIC_InitStructure; RCC_APB1PeriphClockCmd(RCC_APB1Periph //使能TIM3时钟

```
TIM_TimeBaseInitStructure.TIM_Period = arr;  //自动重装载值
TIM_TimeBaseInitStructure.TIM_Prescaler=psc;    //定时器分频
TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数
模式
TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;

TIM_TimeBaseInit(TIM3,&TIM_TimeBaseInitStructure);// 初始化定时器TIM3

TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE); //允许定时器3更新中断

NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn; //定时器3中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x01; //抢占优先级1
NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x03; //响应优先级3
```

```
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);// 初始化NVIC
    TIM_Cmd(TIM3,ENABLE); //使能定时器3
}
```

2）定时器中断处理函数

```
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)==SET) //溢出中断
    {
        printf("timer out !\r\n");
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update);    //清除中断标志位
}
```

3）编写主函数

```
int main()
{
    uart_config();
    //Tout=((arr+1)*(psc+1)),

    //84M等于APB1定时器时钟主频，即每秒产生一次定时器中断
    timer_init(10000-1,8400-1) ;
    while(1);
}
```

20.5 软件下载、测试验证

第 21 章

PWM实验

PWM（Pulse Width Modulation）即脉冲宽度调制的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。

21.1 实验原理与基础知识

STM32F4的定时器除了TIM6和TIM7其它都可以用来产生PWM输出。其中高级定时器TIM1和TIM8可以同时产生多达7路的PWM输出。而通用定时器也能同时产生多达4路的PWM输出。

本案例使用TIM4的CH3产生一路PWM输出，通过改变占空比实现对D3亮度的控制。

21.2 电路设计

使用指示灯D3，复用功能可设置使用TIM4_CH3

61	95	139	A5	167	PB8	I/O	FT		TIM4_CH3/SDIO_D4/ TIM10_CH1 / DCMI_D6 / ETH_MII_TXD3 / I2C1_SCL/ CAN1_RX/ EVENTOUT

21.3 库函数和数据结构

使能TIM4和GPIO时钟，配置PB8选择复用功能AF8（TIM4）输出：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE); //TIM4 时钟使能
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //使能PB时钟
```

初始化TIM14,设置TIM4的ARR和PSC等参数:

```
//设置ARR和PSC两个寄存器的值来控制输出PWM的周期。
TIM_TimeBaseStructure.TIM_Prescaler=psc;    //定时器分频
TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseStructure.TIM_Period=arr;      //自动重装载值
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
TIM_TimeBaseInit(TIM4,&TIM_TimeBaseStructure);//初始化定时器4
```

设置TIM4_CH3的PWM模式，使能TIM4的CH3输出:

```
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //PWM调制模式1
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //输出极性低
TIM_OC3Init(TIM4, &TIM_OCInitStructure);    //初始化外设TIM4
TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);    //使能预装载寄存器
TIM_ARRPreloadConfig(TIM4,ENABLE);//ARPE使能
TIM_Cmd(TIM4, ENABLE);    //使能TIM4
```

设置占空比，通过修改TIM4_CCR1来控制占空比，PWM开始输出的占空比和频率都是固定的，而通过修改TIM4_CCR3则可以控制CH3的输出占空比，继而控制D3的亮度。

```
TIM_SetCompare3(TIM4,led0pwmval);  //修改比较值，修改占空比
```

21.4 软件设计、开发

1）初始化定时器PWM模式

```
//arr：自动重装值  psc：时钟预分频数
void timer_pwm_init(u32 arr,u32 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;
    TIM_OCInitTypeDef    TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE); //TIM4 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //使能PB时钟

    GPIO_PinAFConfig(GPIOB,GPIO_PinSource8,GPIO_AF_TIM4); //PB8复用为TIM4
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;                    //GPIOB8
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;            //复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;  //速度50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;            //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;               //上拉
   GPIO_Init(GPIOB,&GPIO_InitStructure);                        //初始化PB8

    TIM_TimeBaseStructure.TIM_Prescaler=psc;    //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
```

```
        TIM_TimeBaseStructure.TIM_Period=arr;      //自动重装载值
        TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
        TIM_TimeBaseInit(TIM4,&TIM_TimeBaseStructure);//初始化定时器4

        //初始化TIM4 Channel3 PWM模式
        TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //PWM调制模式1
        TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出
使能
        TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //输出极性低
        TIM_OC3Init(TIM4, &TIM_OCInitStructure);    //初始化外设TIM4
        TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);    //使能预装载寄存器
        TIM_ARRPreloadConfig(TIM4,ENABLE);//ARPE使能
        TIM_Cmd(TIM4, ENABLE);    //使能TIM4

        //高级定时器，使能主输出，普通定时器不用加
        TIM_CtrlPWMOutputs(TIM4, ENABLE);
    }
```

2） 编写主函数

```
    int main()
    {
        u16 led0pwmval=0;
        u8 dir=1;

        uart_config();
        timer_pwm_init(500-1,84-1);
        /*定时器4时钟为84M，分频系数为84，所以计数频率
           为84M/84=1Mhz，重装载值500，所以PWM频率为:1M/500=2Khz,脉宽周期为：
1/2000=0.5ms*/
        while(1)
        {
            delay_ms(10);
            if(dir)
                led0pwmval++;//dir==1 led0pwmval 递增
            else
                led0pwmval--; //dir==0 led0pwmval递减
            if(led0pwmval>300){
                dir=0;//led0pwmval 到达300后，方向为递减
            }
            if(led0pwmval==0){
                dir=1;  //led0pwmval 递减到0后，方向改为递增
            }
            TIM_SetCompare3(TIM4,led0pwmval);  //修改比较值，修改占空比
        }
    }
```

21.5 软件下载、测试验证

第 22 章

DAC实验

STM32F4的DAC(数字/模拟转换）模块是12位数字输入，电压输出型的DAC。

本案例使用DAC输出电压，在利用ADC采样获取DAC到值并用过串口打印转换结果。

22.1 实验原理与基础知识

STM32F4的DAC可以配置为8位或12位模式，也可以与DMA控制器配合使用。DAC工作在12位模式时，数据可以设置成左对齐或右对齐。DAC模块有2个输出通道，每个通道都有单独的转换器。在双DAC模式下，2个通道可以独立地进行转换，也可以同时进行转换并同步地更新2个通道的输出。

STM32F4的DAC模块主要特点有：

① 2个DAC转换器：每个转换器对应1个输出通道

② 8位或者12位单调输出

③ 12位模式下数据左对齐或者右对齐

④ 同步更新功能

⑤ 噪声波形生成

⑥ 三角波形生成

⑦ 双DAC通道同时或者分别转换
⑧ 每个通道都有DMA功能

22.2 电路设计

使用2.0转2.54杜邦线连接J28到Pin5（PA4）和J21的Pin2（PB0）相连接

22.3 库函数和数据结构

开启DAC1时钟：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);//使能DAC时钟
```

初始化DAC，设置DAC的工作模式：

```
DAC_InitTypeDef DAC_InitType;
DAC_InitType.DAC_Trigger=DAC_Trigger_None;  //不使用触发功能  TEN1=0
DAC_InitType.DAC_WaveGeneration=DAC_WaveGeneration_None;//不使用波形发生
DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude=DAC_LFSRUnmask_Bit0;
 DAC_InitType.DAC_OutputBuffer=DAC_OutputBuffer_Disable ;  //DAC1 输出缓存关闭
DAC_Init(DAC_Channel_1,&DAC_InitType);    //初始化DAC通道1
```

使能DAC转换通道：

```
DAC_Cmd(DAC_Channel_1, ENABLE);    //使能DAC通道
```

设置DAC的输出值。

```
 DAC_SetChannel1Data(DAC_Align_12b_R, 0);    //12位右对齐数据格式设置DAC值，初始值0
```

读取DAC通道转换的数值：

```
DAC_GetDataOutputValue(DAC_Channel_1);
```

22.4 软件设计、开发

在ADC案例中直接添加dac相关代码

1）初始化DAC1

```
//DAC通道1输出初始化
void dac1_Init(void)
{
    GPIO_InitTypeDef    GPIO_InitStructure;
    DAC_InitTypeDef DAC_InitType;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//①使能PA时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);//②使能DAC时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;//模拟输入
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;//下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);//①初始化GPIO
    DAC_InitType.DAC_Trigger=DAC_Trigger_None; //不使用触发功能  TEN1=0
    DAC_InitType.DAC_WaveGeneration=DAC_WaveGeneration_None;//不使用波形发生
    DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude=DAC_LFSRUnmask_Bit0;

    //屏蔽、幅值设置
    DAC_InitType.DAC_OutputBuffer=DAC_OutputBuffer_Disable ;//输出缓存关闭
    DAC_Init(DAC_Channel_1,&DAC_InitType);   //③初始化DAC通道1
    DAC_Cmd(DAC_Channel_1, ENABLE);    //④使能DAC通道1
    DAC_SetChannel1Data(DAC_Align_12b_R, 0);    //⑤12位右对齐数据格式
}
```

2）编写住主函数，让DAC输出电压逐渐增大，并通过ADC获取打印结果

```
int main()
{
    double adc_value;
    u16 dacval=0;
    uart_config();
    ADC1_CH8_DMA_Config();

    dac1_Init();          //DAC通道1初始化
    DAC_SetChannel1Data(DAC_Align_12b_R,dacval);//初始值为0

    /* Start ADC1 Software Conversion */
    ADC_SoftwareStartConv(ADC1);
```

```
    while (1)
    {
        adc_value=ADC_filter()*1.0/1000;
        printf("get ADC input value=%.1f V\r\n",adc_value);

        if(dacval >= 4000){
            dacval = 0;
        }
        else{
            dacval += 200;
        }
        /*电压值
            Val/3.3 = dacval/4096
            Val = dacval*3.3/4096;
        */
        DAC_SetChannel1Data(DAC_Align_12b_R, dacval);//设置DAC值
        printf("set DAC output val=%.1f;  ",dacval*3.3/4096);
        delay_ms(500);
    }
}
```

22.5 软件下载、测试验证

第 23 章

LCD 实验

编写LCD驱动，通过 STM32F4 的 FSMC 接口来控制 LCD 进行显示。

23.1 实验原理与基础知识

LCD 液晶屏是 Liquid Crystal Display 的简称，屏幕能显示的基本原理就是在两片平行的玻璃当中放置液态的晶体，两片玻璃中间有许多垂直和水平的细小电线，通过通电与否或电压的大小改变来控制液晶分子的转动方，以达到遮光和透光的目的来显示深浅不一，错落有致的图象，而且只要在两块平板间再加上三元色的滤光层，就可实现显示彩色图象。LCD显示原理如图所示

因此要在LCD上显示一幅画的过程就是在正确的位置写入正确的电压，如果要维持该画面的显示就需要保持该位置的电压，通常情况下LCD控制电路会完成定时刷新电压的工作，

这个跟RAM控制器非常类似。但是根据LCD不一样，具体配置相应的控制寄存器。Elink407使用的是LCD模组是LX T30WH006T1，具体参见LCD的datasheet。

LCD与stm32F4通过FSMC（灵活的静态存储控制器）接口连接，FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接，STM32F4 的 FSMC 接口支持包括 SRAM、 NAND FLASH、 NOR FLASH 和 PSRAM 等存储器。FSMC 的框图。

FSMC 框图

FSMC 包含四个主要模块：

- AHB 接口（包括FSMC 配置寄存器）
- NOR Flash/PSRAM 控制器
- NAND Flash/PC 卡控制器
- 外部器件接口

STM32F4 的 FSMC 将外部设备分为 2 类：NOR/PSRAM 设备、NAND/PC 卡设备。他们共用地址数据总线等信号，他们具有不同的 CS 以区分不同的设备，将 LCD 当成 RAM 来控制，LCD的地址线、数据线、写信号（WE）、读信号（OE）、片选信号（CS），命令数据选择信号(RS)，分别与FSMC对应的功能管脚相连，需要说明的是 LCD 有 RS 信号，但是没有地址信号，我们把 RS 接在 A16 上面，那么当 FSMC 控制器写地址 0 的时候，就是写命令。写地址 1 的时候，就是写数据了。这样实现在同一数据总线上面区分地址和数据。

STM32F4 的 FSMC 支持 8/16/32 位数据宽度，我们这里用到的 LCD 是 16 位宽度的，所以在设置的时候，选择 16 位宽。STM32F4的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，如图所示FSMC 存储块地址映像

FSMC 总共管理 1GB 空间，拥有 4 个存储块（Bank），LCD用到的是NOR/PSRAM块，该区块可连接多达4 个NOR Flash 或PSRAM 存储器器件。此存储区域被划分为4 个 NOR/PSRAM 区域，带4 个专用片选信号,内 部 AHB 地址总 线HADDR25:0（section B.2)用来表示外部存储器地址,而HADDR27:26（section B.2）位用于四个存储区域之中选择其中一个存储区域如表所示。

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0X6000,0000~63FF,FFFF	00	FSMC_A[25:0]
第 2 区	FSMC_NE2	0X6400,0000~67FF,FFFF	01	
第 3 区	FSMC_NE3	0X6800,0000~6BFF,FFFF	10	
第 4 区	FSMC_NE4	0X6C00,0000~6FFF,FFFF	11	

对于 NOR/PSRAM控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1～4，对应 4 个区,Elink407 LCD使用的是1）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数。

FSMC 的 NOR FLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

1，HCLK 与 FSMC_CLK 的分频系数(CLKDIV)，可以为 2～16 分频；

2，同步突发访问中获得第 1 个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。

NOR/PSRAM 的可编程访问参数

参数	功能	访问模式	单位	最小值	最大值
地址建立	地址建立阶段的持续时间	异步	AHB 时钟周期 (HCLK)	0	15
地址保持	地址保持阶段的持续时间	异步，复用 I/O	AHB 时钟周期 (HCLK)	1	15
数据建立	数据建立阶段的持续时间	异步	AHB 时钟周期 (HCLK)	1	256
总线周转	总线周转阶段的持续时间	异步和同步读取	AHB 时钟周期 (HCLK)	0	15
时钟分频比	构建一个存储器时钟周期 (CLK) 所需的 AHB 时钟周期 (HCLK) 数量	同步	AHB 时钟周期 (HCLK)	2	16
数据延迟	在发出突发的第一个数据前向存储器发出的时钟周期数量	同步	存储器时钟周期 (CLK)	2	17

FSMC 定义了 4 种不同的异步时序模型(A,B,C,D)。选用不同的时序模型时，需要设置不同的时序参数，ELink407 LCD 使用的是A模式，其读写时序如图所示

模式 A 读取访问

FSMC_BCRx 位字段

FSMC_BTRx 位字段

FSMC_BWTRx 位字段

NOR/PSRAM 控制寄存器

SRAM/NOR-Flash 片选控制寄存器 偏移地址：0xA000 0000 + 8 * (x — 1)，x = 1…4

模式 A 写入访问

表 198. FSMC_BTRx 位字段

位号	位名	要设置的值
31:30	保留	0x0
29-28	ACCMOD	0x0
27-24	DATLAT	无关
23-20	CLKDIV	无关
19-16	BUSTURN	NEx 变为高电平到 NEx 变为低电平之间的时间 (BUSTURN HCLK)
15-8	DATAST	第二个访问阶段的持续时间（写入访问为 DATAST+1 个 HCLK 周期，读取访问为 DATAST 个 HCLK 周期）。
7-4	ADDHLD	无关
3-0	ADDSET	第一个访问阶段的持续时间（ADDSET 个 HCLK 周期）。 ADDSET 的最小值为 0。

FSMC_BTRx 位字段

位号	位名	要设置的值
31:30	保留	0x0
29-28	ACCMOD	0x0
27-24	DATLAT	无关
23-20	CLKDIV	无关
19-16	BUSTURN	NEx 变为高电平到 NEx 变为低电平之间的时间 (BUSTURN HCLK)
15-8	DATAST	第二个访问阶段的持续时间（写入访问为 DATAST+1 个 HCLK 周期，读取访问为 DATAST 个 HCLK 周期）。
7-4	ADDHLD	无关
3-0	ADDSET	第一个访问阶段的持续时间（ADDSET 个 HCLK 周期）。 ADDSET 的最小值为 0。

FSMC_BWTRx 位字段

位号	位名	要设置的值
31:30	保留	0x0
29-28	ACCMOD	0x0
27-24	DATLAT	无关
23-20	CLKDIV	无关
19-16	BUSTURN	NEx 变为高电平到 NEx 变为低电平之间的时间 (BUSTURN HCLK)
15-8	DATAST	第二个访问阶段的持续时间（写入访问为 DATAST+1 个 HCLK 周期）
7-4	ADDHLD	无关
3-0	ADDSET	第一个访问阶段的持续时间（ADDSET 个 HCLK 周期）。 ADDSET 的最小值为 0。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												CBURSTRW	Reserved			ASCYCWAIT	EXTMOD	WAITEN	WREN	WAITCFG	WRAPMOD	WAITPOL	BURSTEN	Reserved	FACCEN	MWID		MTYP		MUXEN	MBKEN
												rw				rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位31：20 保留，必须保持复位值。

位19 CBURSTRW：写入突发使能(Write burst enable)。对于Cellular RAM (PSRAM)，该位可在写操作时使能同步突发协议。读取访问期间同步突发协议的使能位为FSMC_BCRx 寄存器中的BURSTEN 位。

0：始终在异步模式下进行写入操作

1：在同步模式下进行写入操作。

位18：16 保留，必须保持复位值。

位15 ASYNCWAIT：异步传输期间的等待信号(Wait signal during asynchronous transfers)该位可使能/禁止FSMC 使用等待信号，即使是在异步协议期间该位也有作用。

0：运行异步协议时不考虑NWAIT 信号（复位后的默认值）

1：运行异步协议时考虑NWAIT 信号

位14 EXTMOD：扩展模式使能(Extended mode enable)。FSMC 可对FSMC_BWTR 寄存器中的写入时间进行配置，此配置由EXTMOD 位使能，进而使读取和写入操作采用不同时序。

0：不考虑FSMC_BWTR 寄存器中的值（复位后的默认值）

1：考虑FSMC_BWTR 寄存器中的值

位13 WAITEN：等待使能位(Wait enable bit)。该位可使能/禁止在同步模式下访问Flash 时通过NWAIT 信号插入等待周期。

0：禁止NWAIT 信号（不考虑其电平，不在配置过的Flash 延迟周期后插入等待周期）

1：使能NWAIT 信号（考虑其电平，如果使能，在配置过的Flash 延迟周期后插入等待周期）（复位后的默认值）

位12 WREN：写入使能位(Write enable bit)。该位指示FSMC 是否在存储区域内使能/禁止了写入操作：

0：FSMC 在存储区域内禁止了写入操作，如果进行写操作将报告AHB 错误，

1：FSMC 在存储区域内使能了写入操作（复位后的默认值）。

位11 WAITCFG：等待时序配置(Wait timing configuration)。NWAIT 信号指示存储器中的数据是否有效，或者在同步模式下访问Flash 时是否必须插入等待周期。该配置位决定存储器是在等待周期之前的一个时钟周期还是等待周期期间使能NWAIT：

0：NWAIT 信号在等待周期之前的一个数据周期有效（复位后的默认值），

1：NWAIT 信号在等待周期期间有效（不适用于Cellular RAM）。

位10 WRAPMOD：环回突发模式支持(Wrapped burst mode support)。定义控制器是否会将一个AHB 突发环回访问分割成为两个线性访问。仅在突发模式下访 问存储器时有效 * 0：未使能直接环回突发（复位后的默认值）， * 1：使能直接环回突发。

位9 WAITPOL：等待信号极性位(Wait signal polarity bit)。定义存储器的等待信号极性。仅在突发模式下访问存储器时有效：

0：NWAIT 低电平有效（复位后的默认值），

1：NWAIT 高电平有效。

位8 BURSTEN：突发使能位(Burst enable bit)。该位可使能/禁止读取操作期间的同步突发访问。仅对同步突发存储器有效：

0：禁止突发访问模式（复位后的默认值）

1：使能突发访问模式

位7 保留，必须保持复位值。

位6 FACCEN：Flash 访问使能(Flash access enable)使能NOR Flash 访问操作。

0：禁止相应的NOR Flash 访问

1：使能相应的NOR Flash 访问（复位后的默认值）

位5:4 MWID：存储器数据总线宽度(Memory databus width)。定义外部存储器器件宽度，对所有类型的存储器均有效。

00：8 位，

01：16 位（复位后的默认值），

10：保留，不使用，

11：保留，不使用。

位3:2 MTYP：存储器类型(Memory type)。定义与相应存储区域相连的外部存储器类型：

00：SRAM、ROM（对于存储区域2…4，复位后的默认值）

01：PSRAM（Cellular RAM：CRAM）

10：NOR Flash/OneNAND Flash （对于存储区域1，复位后的默认值）

11：保留

位1 MUXEN：地址/数据复用使能位(Address/data multiplexing enable bit)。该位置1 时，地址和数据值在数据总线上复用，仅对NOR 和PSRAM 存储器有效：

0：地址/数据非复用

1：地址/数据在数据总线上复用（复位后的默认值）

位0 MBKEN：存储区域使能位(Memory bank enable bit)。使能存储区域。复位后使能存储区域1，其它存储区域均禁止。访问禁止的存储区域会引起AHB 总线上的错误。

0：禁止相应的存储区域

1：使能相应的存储区域

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		ACCMOD		DATLAT				CLKDIV				BUSTURN				DATAST								ADDHLD				ADDSET			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SRAM/NOR-Flash 片选时序寄存器(FSMC_BTRx)

偏移地址：0xA000 0000 + 0x04 + 8 * (x — 1)，x = 1..4

该寄存器包含每个存储区域的控制信息，用于SRAM、ROM 和NOR Flash。如果FSMC_BCRx寄存器中的EXTMOD 位置1，该寄存器将和另外一个寄存器配合来配置写入和读取的时序参数。也就是说有2 个寄存器可用：一个用于配置读取访问（此寄存器），另一个用于配置写入访问（FSMC_BWTRx 寄存器）。

位31:30 保留，必须保持复位值。

位29:28 ACCMOD：访问模式(Access mode)指定异步访问模式，如时序图所示。仅当FSMC_BCRx 寄存器中的ECTMOD 位为1 时才考虑这些位。

00：访问模式A

01：访问模式B

10：访问模式C

11：访问模式D

位27:24 DATLAT：同步突发NOR Flash 的数据延迟(Data latency for synchronous burst NOR Flash memory)对于使能了同步突发模式的NOR Flash，该位定义了获取首个数据前要发送给存储器的存储器 时钟周期数(+2)：

0000：首次突发访问时，2 个CLK 时钟周期的数据延迟

1111：首次突发访问时，17 个CLK 时钟周期的数据延迟（复位后的默认值）

注意：该时序参数以Flash 时钟(CLK) 周期而非HCLK 周期表示。在异步NOR Flash、SRAM或ROM 访问模式下，该值为无关值。如果是CRAM，该字段必须设置为“0”。

位23:20 CLKDIV：CLK 信号的时钟分频比(Clock divide ratio (for CLK signal))定义CLK 时钟输出信号的周期，以HCLK 周期数表示：

0000：Reserved

0001：CLK 周期= 2 × HCLK 周期

0010：CLK 周期= 3 × HCLK 周期

1111：CLK 周期= 16 × HCLK 周期（复位后的默认值）

在异步NOR Flash、SRAM 或ROM 访问模式下，该值为无关值。

位19:16 BUSTURN：总线周转阶段的持续时间(Bus turnaround phase duration)通过软件写入这些位可在写入/读取事务的结尾添加延迟。该延迟可以匹配连续事务之间的最短时间（tEHEL 由NEx 高电平变为NEx 低电平）以及存储器在读取访问后释放数据总线所需 的最长时间(tEHQZ)：

(BUSTRUN + 1)HCLK 周期 tEHELmin 和(BUSTRUN + 2)HCLK 周期 tEHQZmax（如果EXTMOD = “0”）

(BUSTRUN + 2)HCLK 周期 max（tEHELmin，tEHQZmax）（如果EXTMOD = “1”）。

0000：BUSTURN 阶段的持续时间= 增加0 个HCLK 时钟周期

…

1111：BUSTURN 阶段的持续时间= 15 × HCLK 时钟周期（复位后的默认值）

位15:8 DATAST：数据阶段的持续时间(Data-phase duration)通过软件写入这些位可定义数据阶段的持续时间，适用于SRAM、ROM 和异步NOR Flash 访问模式：

0000 0000：Reserved

0000 0001：DATAST 阶段的持续时间= 1 × HCLK 时钟周期

0000 0010：DATAST 阶段的持续时间= 2 × HCLK 时钟周期

…

1111 1111：DATAST 阶段的持续时间= 255 × HCLK 时钟周期（复位后的默认值）

位7:4 ADDHLD：地址保持阶段的持续时间(Address-hold phase duration)通过软件写入这些位可定义地址保持阶段的持续时间，适用于模式 D 和复用访问：

0000：Reserved

0001：ADDHLD 阶段的持续时间= 1 × HCLK 时钟周期

0010：ADDHLD 阶段的持续时间= 2 × HCLK 时钟周期

…

1111：ADDHLD 阶段的持续时间= 15 × HCLK 时钟周期（复位后的默认值）

位3:0 ADDSET：地址设置阶段的持续时间(Address setup phase duration)通过软件写入这些位可定义地址设置阶段的持续时间，适用于SRAM、ROM 和异步NOR Flash 访问模式：

0000：ADDSET 阶段的持续时间= 0 × HCLK 时钟周期

…

1111：ADDSET 阶段的持续时间= 1615 × HCLK 时钟周期（复位后的默认值）

SRAM/NOR-Flash 写入时序寄存器 (FSMC_BWTRx)

偏移地址：0xA000 0000 + 0x104 + 8 * (x — 1)，x = 1…4

该寄存器包含每个存储区域的控制信息，用于SRAM、ROM 和NOR Flash。当FSMC_BCRx 寄存器中的EXTMOD 位置1 时，该寄存器将处于有效状态，可以进行写入访问。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.		ACCMOD		DATLAT				CLKDIV				BUSTURN				DATAST								ADDHLD				ADDSET			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 保留，必须保持复位值。

位29:28 ACCMOD：访问模式(Access mode)。指定异步访问模式，如下一个时序图所示。仅当FSMC_BCRx 寄存器中的EXTMOD 位置1 时 ， 这些位才有效。

00：访问模式A

01：访问模式B

10：访问模式C

11：访问模式D

位27:24 DATLAT：数据延迟（用于同步突发NOR Flash）（Data latency (for synchronous burst NOR Flash)）。对于使能了同步突发模式的NOR Flash，该位定义了获取首个数据前要发送给存储器的存储器时钟 周期数(+2)：

0000：(0x0) 首次突发访问时，2 个CLK 时钟周期的数据延迟

…

1111：(0xF) 首次突发访问时，17 个CLK 时钟周期的数据延迟（复位后的默认值）

位23:20 CLKDIV：CLK 信号的时钟分频比(Clock divide ratio (for CLK signal))。定义CLK 时钟输出信号的周期，以HCLK 周期数表示：

0000：保留

0001 CLK 周期= 2 × HCLK 周期

0010 CLK 周期= 3 × HCLK 周期

1111：CLK 周期= 16 × HCLK 周期（复位后的默认值）

位19:16 BUSTURN：总线周转阶段的持续时间(Bus turnaround phase duration)通过软件写入这些位可在写入事务结束时增加一个延迟，进而匹配两个连续事务之间的最小时间（tEHEL 由ENx 高电平变为ENx 低电平）：(BUSTRUN + 1) HCLK 周期 tEHELmin。

0000：BUSTURN 阶段的持续时间= 增加0 个HCLK 时钟周期

…

1111：BUSTURN 阶段的持续时间= 增加15 个HCLK 时钟周期（复位后的默认值）

位15:8 DATAST：数据阶段的持续时间(Data-phase duration)。通过软件写入这些位可定义数据阶段的持续时间，适用于SRAM、ROM 和异步NOR Flash 访问模式：

0000 0000：保留

0000 0001:DATAST 阶段的持续时间= 1 × HCLK 时钟周期

0000 0010:DATAST 阶段的持续时间= 2 × HCLK 时钟周期

…

1111 1111:DATAST 阶段的持续时间= 255 × HCLK 时钟周期（复位后的默认值）

位7:4 ADDHLD：地址保持阶段的持续时间(Address-hold phase duration)。通过软件写入这些位可定义地址保持 阶段的持续时间，适用于SRAM、ROM 和异步复用NOR Flash 访问模式：

0000：保留

0001：ADDHLD 阶段的持续时间= 1 × HCLK 时钟周期

0010：ADDHLD 阶段的持续时间= 2 × HCLK 时钟周期

…

1111：ADDHLD 阶段的持续时间= 15 × HCLK 时钟周期（复位后的默认值）

位3:0 ADDSET：地址设置阶段的持续时间(Address setup phase duration)。通过软件写入这些位可定义以HCLK 周期表示的地址设置 阶段的持续时间，适用于SRAM、ROM 和异步NOR Flash 访问模式：

0000：ADDSET 阶段的持续时间= 0 × HCLK 时钟周期

…

1111：ADDSET 阶段的持续时间= 15 × HCLK 时钟周期（复位后的默认值）

23.2 电路设计

23.3 库函数和数据结构

相关的操作在文件 stm32f4xx_fsmc.c 和对应的头文件 stm32f4xx_fsmc.h 中

NORSRAM初始化结构体，前面有 13 个32位整形的成员变量，是用来配置片选控制寄存器 FSMC_BCRx。最后面还有两个SMC_NORSRAMTimingInitTypeDef 指针类型的成员变量是用来设置读时序和写时序的参数。

```
/**
  * @brief  FSMC NOR/SRAM Init structure definition
  */

typedef struct
```

```
{
    /*!< Specifies the NOR/SRAM memory bank that will be used.
    This parameter can be a value of @ref FSMC_NORSRAM_Bank */
  uint32_t FSMC_Bank;

    /*!< Specifies whether the address and data values are
    multiplexed on the data bus or not.
   This parameter can be a value of @ref FSMC_Data_Address_Bus_Multiplexing */
  uint32_t FSMC_DataAddressMux;

    /*!< Specifies the type of external memory attached to
    the corresponding memory bank.
    This parameter can be a value of @ref FSMC_Memory_Type */
  uint32_t FSMC_MemoryType;

    /*!< Specifies the external memory device width.
    This parameter can be a value of @ref FSMC_Data_Width */
  uint32_t FSMC_MemoryDataWidth;

    /*!< Enables or disables the burst access mode for Flash memory,
    valid only with synchronous burst Flash memories.
    This parameter can be a value of @ref FSMC_Burst_Access_Mode */
  uint32_t FSMC_BurstAccessMode;

    /*!< Enables or disables wait signal during asynchronous transfers,
    valid only with asynchronous Flash memories.
    This parameter can be a value of @ref FSMC_AsynchronousWait */
  uint32_t FSMC_AsynchronousWait;

    /*!< Specifies the wait signal polarity, valid only when accessing
    the Flash memory in burst mode.
    This parameter can be a value of @ref FSMC_Wait_Signal_Polarity */
  uint32_t FSMC_WaitSignalPolarity;

    /*!< Enables or disables the Wrapped burst access mode for Flash
    memory, valid only when accessing Flash memories in burst mode.
    This parameter can be a value of @ref FSMC_Wrap_Mode */
  uint32_t FSMC_WrapMode;

    /*!< Specifies if the wait signal is asserted by the memory one
    clock cycle before the wait state or during the wait state,
    valid only when accessing memories in burst mode.
    This parameter can be a value of @ref FSMC_Wait_Timing */
  uint32_t FSMC_WaitSignalActive;
```

```
  /*!< Enables or disables the write operation in the selected bank by the FSMC.
    This parameter can be a value of @ref FSMC_Write_Operation */
  uint32_t FSMC_WriteOperation;

    /*!< Enables or disables the wait state insertion via wait
    signal, valid for Flash memory access in burst mode.
    This parameter can be a value of @ref FSMC_Wait_Signal */
  uint32_t FSMC_WaitSignal;

    /*!< Enables or disables the extended mode.
    This parameter can be a value of @ref FSMC_Extended_Mode */
  uint32_t FSMC_ExtendedMode;

    /*!< Enables or disables the write burst operation.
    This parameter can be a value of @ref FSMC_Write_Burst */
  uint32_t FSMC_WriteBurst;

/*!< Timing Parameters for write and read access if the  Extended Mode is not used*/
  FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;

     /*!< Timing Parameters for write access if the  Extended Mode is used*/
  FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;

  }FSMC_NORSRAMInitTypeDef;
```

FSMC 有读时序和写时序之分，这里定义的FSMC_NORSRAMTimingInitTypeDef结构体，用来设置读时序和写时序的参数，即配置寄存器 FSMC_BTRx 和 FSMC_BWTRx。 结构体有 7 个参数用来设置 FSMC 读写时序，主要是设计地址建立保持时间，数据建立时间等等配置 。

```
  /**
    * @brief  Timing parameters For NOR/SRAM Banks
    */
typedef struct
{
    /*!< Defines the number of HCLK cycles to configure
    the duration of the address setup time.
    This parameter can be a value between 0 and 0xF.
    @note This parameter is not used with synchronous NOR Flash memories. */
    uint32_t FSMC_AddressSetupTime;

    /*!< Defines the number of HCLK cycles to configure
    the duration of the address hold time.
    This parameter can be a value between 0 and 0xF.
    @note This parameter is not used with synchronous NOR Flash memories.*/
    uint32_t FSMC_AddressHoldTime;

    /*!< Defines the number of HCLK cycles to configure
```

```
 the duration of the data setup time.
 This parameter can be a value between 0 and 0xFF.
 @note This parameter is used for SRAMs, ROMs and asynchronous
 multiplexed NOR Flash memories. */
uint32_t FSMC_DataSetupTime;

/*!< Defines the number of HCLK cycles to configure
the duration of the bus turnaround.
This parameter can be a value between 0 and 0xF.
@note This parameter is only used for multiplexed NOR Flash memories. */
uint32_t FSMC_BusTurnAroundDuration;

/*!< Defines the period of CLK clock output signal, expressed in
number of HCLK cycles.
This parameter can be a value between 1 and 0xF.
@note This parameter is not used for asynchronous NOR Flash,
SRAM or ROM accesses. */
uint32_t FSMC_CLKDivision;

/*!< Defines the number of memory clock cycles to issue
to the memory before getting the first data.
The parameter value depends on the memory type as shown below:
- It must be set to 0 in case of a CRAM
- It is don't care in asynchronous NOR, SRAM or ROM accesses
- It may assume a value between 0 and 0xF in NOR Flash memories
with synchronous burst mode enable */
uint32_t FSMC_DataLatency;

/*!< Specifies the asynchronous access mode.
This parameter can be a value of @ref FSMC_Access_Mode */
uint32_t FSMC_AccessMode;

}FSMC_NORSRAMTimingInitTypeDef;
```

```
void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState)
```

FSMC 时钟使能函数

```
void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);
```

FSMC 使能函数,这里我们使用的类型是NORSRAM，所以使用上面的函数，如果是其他的类型的存储调用相应的使能函数。

23.4 软件设计、开发

在工程中添加lcd.c 文件和lcd.h 文件，LCD相关的函数和定义都在这两个文件中。 编写LCD 用GPIO配置函数，初始化所有LCD相连的GPIO。

```
void LCD_CtrlLinesConfig(void)
{

    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable GPIOD, GPIOE, and AFIO clocks */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD |
                           RCC_AHB1Periph_GPIOE, ENABLE);

    /*-- GPIO Configuration ---*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14 | GPIO_Pin_15 |
        GPIO_Pin_0  | GPIO_Pin_1 |   GPIO_Pin_7 |
        GPIO_Pin_8  | GPIO_Pin_9 | GPIO_Pin_10 |
        GPIO_Pin_11 | GPIO_Pin_4 | GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    GPIO_PinAFConfig(GPIOD, GPIO_PinSource0, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource1, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource4, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource5, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource7, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource8, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource9, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource10, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource11, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource14, GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource15, GPIO_AF_FSMC);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9 |
            GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12 |
            GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
    GPIO_Init(GPIOE, &GPIO_InitStructure);

    GPIO_PinAFConfig(GPIOE, GPIO_PinSource7 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource8 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource9 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource10 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource11 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource12 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource13 , GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource14 , GPIO_AF_FSMC);
```

```
    GPIO_PinAFConfig(GPIOE, GPIO_PinSource15 , GPIO_AF_FSMC);

    /*PD13(LCD_BL) configuration*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    GPIO_SetBits(GPIOD, GPIO_Pin_13);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    {
        int i;
        GPIO_ResetBits(GPIOD, GPIO_Pin_3);
        for (i = 0; i < 1000000; i++)
        {
            ;
        }
        GPIO_SetBits(GPIOD, GPIO_Pin_3);
    }
}
```

为LCD配置 FSMC

```
void LCD_FSMCConfig(void)
{
    FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
    FSMC_NORSRAMTimingInitTypeDef  p;

    /* Enable FSMC clock */
    RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);

    /* FSMC_Bank1_NORSRAM_1 configuration */
    p.FSMC_AddressSetupTime = 5;
    p.FSMC_AddressHoldTime = 0;
    p.FSMC_DataSetupTime = 9;
    p.FSMC_BusTurnAroundDuration = 0;
    p.FSMC_CLKDivision = 0;
    p.FSMC_DataLatency = 0;
    p.FSMC_AccessMode = FSMC_AccessMode_A;

    FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
 FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
    FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM;
  FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
```

```
  FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
  FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
  FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
     FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
     FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = \
             FSMC_WaitSignalActive_BeforeWaitState;
   FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
     FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
     FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
     FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;

     FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p;
     FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p;

     FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);

     /* Enable FSMC NOR/SRAM Bank1 */
     FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);
}
```

LCD使用的是FSMC NOR/SRAM Bank1，其所对应地址为0x60000000，宏定义如下：

```
#define LCD_BASE          ((uint32_t)(0x60000000))
```

LCD的RS接地址A16，LCD使用的是 16 位数据宽度，所以 HADDR0（section B.2)并没有用到，只有 HADDR25:1（section B.2)是有效的，对应关 系变为：HADDR25:1（section B.2) > FSMC24:0（section B.2)，相当于右移了一位，故CMD地址为0x60020000，宏定义如下：

```
#define LCD_CMD_ADDR      ((uint32_t)(0x60020000))
```

根据LCD的datasheet 设置如下宏定义：

```
//RS=1 发送数据  命令的值或像数值
#define LCD_CMD_REG       (*( uint16_t *)LCD_CMD_ADDR)
#define LCD_RAM_REG       (*( uint16_t *)LCD_CMD_ADDR)

//RS=0 设置地址（寄存器索引）
#define LCD_IDX_REG       (*((__IO uint16_t *)LCD_BASE))
```

宏定义LCD屏的点阵范围

```
#define LCD_W     (240)
#define LCD_H     (400)
```

向LCD寄存器写数据，LCD_Reg 寄存器ID，LCD_RegValue 寄存器内容

```
void LCD_WriteReg(uint16_t LCD_Reg, uint16_t LCD_RegValue)
{
    /* Write 16-bit Index, then Write Reg */
```

```
    LCD_IDX_REG = LCD_Reg;          //选择LCD寄存器地址

    /* Write 16-bit Reg */
    LCD_CMD_REG = LCD_RegValue;     //发送数据
}
```

操作索引地址，设置像数点的颜色，REG_Code 像素点的颜色

```
__inline void LCD_REG_Select(uint16_t REG_Code)
{
    LCD_IDX_REG = REG_Code;
}
```

配置LCD使其处于接收显示数据的状态

```
__inline void LCD_WriteRAM_Prepare(void)
{
    LCD_IDX_REG =  LCD_REG_DATA;
}
```

向LCD显存写入显示数据

```
__inline void LCD_WriteRAM(uint16_t RGB_Code)
{
    LCD_RAM_REG = RGB_Code;
}
```

LCD初始化配置，首先初始化GPIO管脚，初始化LCD的FSMC配置寄存器，然后按照LCD datasheet 的配置顺序初始化配置寄存器。 void init_lcd(void) { /配置LCD 用到的GPIO /LCD_Ctrl1LinesConfig();

```
    /* Configure the FSMC Parallel interface */
    LCD_FSMCConfig();
    delay_ms(10); /*delay 10 ms*/

    LCD_WriteReg(0x001A,0x0002);
    LCD_WriteReg(0x001B,0x0088);

    //****VCOM offset**///
    LCD_WriteReg(0x0023,0x0000);
    LCD_WriteReg(0x0024,0x0064);
    LCD_WriteReg(0x0025,0x0015);
    LCD_WriteReg(0x002D,0x0003);
    delay_ms(10);

    //****OPON**//
    LCD_WriteReg(0x00E8,0x0060);

    //Power on Setting
```

```
LCD_WriteReg(0x0018,0x0004);
LCD_WriteReg(0x0019,0x0001);
LCD_WriteReg(0x0001,0x0000);
LCD_WriteReg(0x001F,0x0088);

delay_ms(10);
LCD_WriteReg(0x1F,0x80);

delay_ms(10);
LCD_WriteReg(0x1F,0x90);

delay_ms(10);
LCD_WriteReg(0x1F,0xD0);
delay_ms(10);

//262k/65k color selection
LCD_WriteReg(0x17,0x05);

//SET PANEL
LCD_WriteReg(0x36,0x03);
LCD_WriteReg(0x29,0x31);
LCD_WriteReg(0x71,0x1A);

//Gamma 2.2 Setting
LCD_WriteReg(0x40,0x00);
LCD_WriteReg(0x41,0x45);
LCD_WriteReg(0x42,0x45);
LCD_WriteReg(0x43,0x04);

LCD_WriteReg(0x44,0x00);
LCD_WriteReg(0x45,0x08);
LCD_WriteReg(0x46,0x23);
LCD_WriteReg(0x47,0x23);

LCD_WriteReg(0x48,0x77);
LCD_WriteReg(0x49,0x40);
LCD_WriteReg(0x4A,0x04);
LCD_WriteReg(0x4B,0x00);
LCD_WriteReg(0x4C,0x88);
LCD_WriteReg(0x4D,0x88);
LCD_WriteReg(0x4E,0x88);

LCD_WriteReg(0x02,0x00);
LCD_WriteReg(0x03,0x00);
```

```
    LCD_WriteReg(0x04,0x00);
    LCD_WriteReg(0x05,0xEF);

    LCD_WriteReg(0x06,0x00);
    LCD_WriteReg(0x07,0x00);

    LCD_WriteReg(0x08,0x01);
    LCD_WriteReg(0x09,0x8F);


    //Display ON Setting
    LCD_WriteReg(0x28,0x38);
    delay_ms(40);
    LCD_WriteReg(0x28,0x3F);

    LCD_REG_Select(0x22);
}
```

开启LCD显示

```
void LCD_DisplayOn(void)
{
    /* Display On */
    LCD_WriteReg(0x07, 0x0173); /* 262K color and display ON */
}
```

禁止LCD显示

```
void LCD_DisplayOff(void)
{
    /* Display Off */
    LCD_WriteReg(0x7, 0x0);
}
```

设置LCD显存当前的位置 Xpos x坐标，Ypos y坐标

```
void LCD_SetCursor(uint16_t Xpos, uint16_t Ypos)
{
    LCD_WriteReg(0x02, ((Xpos>>8)&0xFF));
    LCD_WriteReg(0x03, (Xpos&0xFF));

    LCD_WriteReg(0x06, ((Ypos>>8)&0xFF));
    LCD_WriteReg(0x07, (Ypos&0xFF));
}
```

在指定位置画一个点 x坐标，y坐标，Color RGB颜色值。

```
void PutPixel(int16_t x, int16_t y, uint16_t Color)
{
```

```
    if(y < 0 || y > (LCD_H - 1) || x < 0 || x > (LCD_W - 1))
    {
        return;
    }

    //设置光标位置
    LCD_SetCursor(x,y);
    LCD_WriteRAM_Prepare();
    LCD_WriteRAM(Color);
}
```

清屏函数 `void LCD_Clear(uint16_t Color) { uint32_t index = 0;`

```
    LCD_SetCursor(0x00, 0x00);
    LCD_WriteRAM_Prepare(); /* Prepare to write GRAM */
    for(index = 0; index < LCD_W * LCD_H; index++)
    {
        LCD_RAM_REG = Color;
    }
}
```

编写main函数进行测试

```
int main(void)
{

    init_delay();

    init_lcd();

    while(1)
    {
        LCD_Clear(Red);
        delay_ms(1000);

        LCD_Clear(White);
        delay_ms(1000);

    }
}
```

23.5 软件下载、测试验证

第 24 章

LCD 显示汉字实验

LCD驱动成功后，再考虑一个问题，如果让LCD上能显示文本信息？我们可以通过代码实现在屏幕的坐标位置显示我们想要的点阵信息，通过多个像数点让屏幕上的点阵显示成像一个字母，然后再组成文本信息。这样想的思路是对的，实际上不需要我们自己建立字符的点阵，而是通过调用现成的点阵字库，实现文本的显示，本案例将实现在LCD上显示中文文本信息。

24.1　实验原理与基础知识

计算机中的图像类型分为两种：位图和矢量图。我们用点阵表示图形叫位图，用数学公式来表达直线或圆的方式就是矢量方法。

字库也分为矢量字库和点阵字库，矢量字体中每一个字形是通过数学曲线来描述的，它包含了字形边界上的关键点，连线的导数信息等，字体的渲染引擎通过读取这些数学矢量，然后进行一定的数学运算来进行渲染。这类字体的好处是字体可以无限放大而不产生变形。点阵从本质上讲就是单色位图，它使用一个比特来表示一个点，如果这个比特为0，表示某个位置没有点，如果为1表示某个位置有点。毋庸置疑，矢量字库的显示效果是最好的，但是点阵字库运算量少，简单并且是免费的。所以在很多场合，特别是嵌入式产品，点阵字库仍然是产品开发的首选。本案例中我们介绍大家点阵字库的使用。

点阵字符是如何表示，其数据又是怎样存储。比如说 16*16 的点阵，也就是说每一行有16个点，由于一个点使用一个比特来表示，如果这个比特的值为1，则表示这个位置有点，如果这个比特的值为0，则表示这个位置没有点，那么一行也就需要16个比特，而8个比特就是一个字节，也就是说，这个点阵中，一行的数据需要两个字节来存放。

第一行的前八个点的数据存放在点阵数据的第一个字节里面，第一行的后面八个点的数据存放在点阵数据的第二个字节里面，第二行的前八个点的数据存放在点阵数据的第三个字节里面…，然后后面的就以此类推了。这样我们可以计算出存放一个点阵总共需要32个字节（2*16）。看看下面这个图形化的例子：

像这样的文字点阵数据就是被保存在点阵字库文件中，那么显示该字符的过程就是获取该字符点阵数据的过程了，下面我们来说明点阵字符的获取。

汉字在汉字库中的具体位置计算公式为：94(区号　1)+位号　1。减1是因为数组是以0为开始而区号位号是以1为开始的。这仅为以汉字为单位该汉字在汉字库中的位置，那么，如何得到以字节为单位得到该汉字在汉字库中的位置呢？只需乘上一个汉字字模占用的字节数即可，即：(94(区号　1)+位号　1)一个汉字字模占用字节数，而按每种汉字库的汉字大小不同又会得到不同的结果。以 1616点阵字库为例，计算公式则为：(94(区号　1)+(位号　1))32。汉字库文该从该位置起的32字节信息即该字的点阵信息。

```
|  |1|  |  |  |  |  |  |  |  |  |1|  |  |  |
|  |  |1|1|  |1|1|1|1|1|1|1|1|1|  |  |
|  |  |  |1|  |  |  |  |  |  |  |1|  |  |  |
|1|  |  |  |  |  |1|  |  |  |  |1|  |  |  |
|  |1|1|  |  |  |1|  |  |  |  |1|  |  |  |
|  |  |1|  |  |  |1|  |  |  |1|  |  |  |  |
|  |  |  |  |1|  |  |1|  |  |1|  |  |  |  |
|  |  |  |1|  |  |  |1|  |1|  |  |  |  |  |
|  |  |1|  |  |  |  |  |1|  |1|  |  |  |  |
|1|1|1|  |  |  |  |  |  |1|  |  |  |  |  |
|  |  |1|  |  |  |  |  |1|  |1|  |  |  |  |
|  |  |1|  |  |  |  |1|  |  |  |1|  |  |  |
|  |  |1|  |  |  |1|  |  |  |  |  |1|  |  |
|  |  |1|  |  |1|  |  |  |  |  |  |1|1|1|  |
|  |  |  |  |1|  |  |  |  |  |  |  |  |1|  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

24.2 电路设计

24.3 库函数和数据结构

\基础实验\common\font\将目录下的 font.c, font_data.s 和font.h头文件加入工程中

```
typedef struct
{
  int16_t FColor;       //字符颜色
  int16_t BColor;       //字符背景色
  int32_t Mode;         //背景色模式使能
} DrawMode, *pDrawMode;
```

字符显示模式结构体

```
typedef struct
{
    void * font;            // 暂时未使用
    unsigned int w;         //字符像数宽度
    unsigned int h;         //字符像数高度
    unsigned int str_bytes; //字符大小 汉字2个字节  ascii 为1个字节
    unsigned char *data;    //字符在字库中的偏移
} FontData, *pFontData;
```

字符信息结构体

```
unsigned char * FONT_GetDataGB2312(FontData *data, const unsigned char *str);
```

根据字符编码，获取字符属性，设置字符属性结构体的值

24.4 软件设计、开发

显示一个字符 x，y 字符显示坐标；Font 字体信息;ShowMode 显示模式

```
void LCD_DrawFont(uint16_t x, uint16_t y, FontData *Font, DrawMode *ShowMode)
{
    int row,col;
    int i;
    uint8_t temp;

    if (Font && Font->data && ShowMode)
    {
        for(row=0; row < Font->h; row++)
        {
            for(col=0; col < Font->w / 8; col++)
            {
                temp=Font->data[row * Font->w / 8 + col];
                for (i=0; i < 8; i++)
                {
                    if(temp & 0x80)
                    {
                        //设置光标位置
                        LCD_SetCursor(x+col*8+i, y+row);
                        LCD_WriteRAM_Prepare();
                        LCD_WriteRAM(ShowMode->FColor);
                    }
                    else
                    {
                        if(ShowMode->Mode & DRAW_BACK_COLOR)
                        {
                            //设置光标位置
                            LCD_SetCursor(x+col*8+i, y+row);
                            LCD_WriteRAM_Prepare();
                            LCD_WriteRAM(ShowMode->BColor);
                        }
                    }
                    temp<<=1;
                }
            }
        }
    }
```

```
}
```

显示字符串 x, y 字符显示坐标; Str 字符串;Mode 显示模式

```
void LCD_DrawString(uint32_t x, uint32_t y, const uint8_t *Str, DrawMode *Mode)
{
    FontData Font;
    uint32_t x0;

    x0 = x;

    while(*Str)
    {
        if (FONT_GetDataGB2312(&Font, Str))
        {
            LCD_DrawFont(x0, y, &Font, Mode);
            x0 += Font.w;
            if (x0 >= LCD_W)
            {
                x0 = x;
                y += Font.h;
                if (y >= LCD_H)
                {
                    break;
                }
            }
            Str += Font.str_bytes;
        }
        else
        {
            Str++;
        }
    }
    return ;
}
```

显示一个整数, x, y 显示起始坐标 ,Num整数值 ,Mode 显示模式

```
void LCD_DisplayNum(uint32_t x, uint32_t y, uint32_t Num, DrawMode *Mode)
{
    char  buf[16];
    sprintf(buf, "%d", Num);
    LCD_DrawString(x, y, (uint8_t *)buf, Mode);
}
```

编写main 函数进行测试

```
int main(void)
```

```
{

    DrawMode Mode;

    Mode.FColor = 0xF800;
    Mode.BColor = 0x001F;
    Mode.Mode = 0;

    init_delay();
    init_lcd();

    while(1)
    {
        LCD_Clear(Red);
        delay_ms(1000);

        LCD_Clear(White);
        delay_ms(1000);

        PutPixel(120, 200, Black);
        delay_ms(1000);

        LCD_DrawString(0, 0, (uint8_t*)"你好 Elink407", &Mode);
        delay_ms(1000);

        LCD_DisplayNum(0, 80, 100, &Mode);
        delay_ms(1000);
    }
}
```

24.5 软件下载、测试验证

第 25 章

显示图片实验

在LCD上显示图片

25.1 实验原理与基础知识

LCD显示图片的过程是将图片上每一个像数信息写到显存中，所以首先我们需要将图片转换为图片数组，然后编写函数将数组中的像数信息写到显存。

25.2 电路设计

25.3 库函数和数据结构

25.4 软件设计、开发

制作位图数组

利用tools目录中的image2lcd工具将elink_logo.bmp图片制作成数组文件logo.h，如图所示设置宽度和高度分别为126，400，设置输出灰度为24位，将图像数组到\基础实验\common\font\logo.h中。

在lcd.c文件中添加LCD_Draw_Imag函数进行画图，同时在lcd.h文件中进行声明。

```
void LCD_Draw_Imag(uint16_t x1,uint16_t y1,uint16_t weight,uint16_t hight,const u8 *img)
 {
     uint32_t color,pix,i,j;
     for(i=0; i<hight; i++)
     {
         LCD_SetCursor(x1, y1+i);
         LCD_WriteRAM_Prepare();
         for (j = 0; j <weight; j++)
         {
             color=0;
             pix=3*(i*weight+j);
             color=img[pix]|(img[pix+1]<<8) |(img[pix+2]<<16);
             LCD_RAM_REG = color;
         }
     }
 }
```

编写main函数进行测试

```
int main(void)
{

    DrawMode Mode;

    Mode.FColor = 0xF800;
    Mode.BColor = 0x001F;
    Mode.Mode = 0;

    init_delay();
    init_lcd();

    while(1)
    {
        LCD_Clear(Red);
        delay_ms(1000);

        LCD_Clear(White);
        delay_ms(1000);

        PutPixel(120, 200, Black);
        delay_ms(1000);

        LCD_DrawString(0, 0, (uint8_t*)"你好 Elink407", &Mode);
        delay_ms(1000);
```

```
        LCD_DisplayNum(0, 80, 100, &Mode);
        delay_ms(1000);

        LCD_Draw_Imag(60,0,126,400, gImage_logo);
        delay_ms(1000);

    }
}
```

25.5 软件下载、测试验证

第 26 章

触摸屏实验

26.1 实验原理与基础知识

26.2 电路设计

26.3 库函数和数据结构

26.4 软件设计、开发

26.5 软件下载、测试验证

第 27 章

待机唤醒实验

通过长按 WAKEUP按键来实现唤醒和进入待机模式的功能，类似手机开关按钮的唤醒功能

27.1 实验原理与基础知识

默认情况下，系统复位或上电复位后，微控制器进入运行模式。在运行模式下，CPU 通过HCLK 提供时钟，并执行程序代码。系统提供了多个低功耗模式，可在CPU 不需要运行时（例如等待外部事件时）节省功耗。由用户根据应用选择具体的低功耗模式，以在低功耗短启动时间和可用唤醒源之间寻求最佳平衡。

器件有三个低功耗模式：

- 睡眠模式（Cortex -M4F 内核停止，外设保持运行）
- 停止模式（所有时钟都停止）
- 待机模式（1.2 V 域断电）

此外，可通过下列方法之一降低运行模式的功耗：

- 降低系统时钟速度
- 不使用APBx 和AHBx 外设时，将对应的外设时钟关闭

27.2 电路设计

27.3 库函数和数据结构

相关的操作在文件 stm32f4xx_pwr.c 和对应的头文件 stm32f4xx_pwr.h 中

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);
```

使能电源时钟函数，因为要配置电源控制寄存器，所以必须先使能电源时钟。

```
PWR_WakeUpPinCmd(ENABLE);
```

使能唤醒管脚功能函数，设置 WK_UP 引脚作为唤醒源。使能时钟之后后再设置 PWR_CSR 的 EWUP 位，使能 WK_UP 用于将 CPU 从待机模式唤醒。在库函数中，设置使能 WK_UP 用于唤醒 CPU 待机模式

```
void PWR_EnterSTANDBYMode(void) ;
```

设置 SLEEPDEEP 位，设置 PDDS 位，执行 WFI 指令，进入待机模式。通过PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。

```
void EXTI0_IRQHandler(void);
```

WakeUP 中断函数。通过 WK_UP 中断（PA0 中断）来唤醒 CPU。

27.4 软件设计、开发

在工程中添加wakeup.c 文件和wakeup.h 文件，所有待机唤醒相关的驱动代码和定义都在这两个文件中。

编写wakeup GPIO 的初始化代码

```
void init_wkup(void)
{
    GPIO_InitTypeDef   GPIO_InitStructure;
    EXTI_InitTypeDef    EXTI_InitStructure;
    NVIC_InitTypeDef    NVIC_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);//使能 SYSCFG 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //PA0
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;//下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化

    EXTI_InitStructure.EXTI_Line = EXTI_Line0;//LINE0
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;//中断事件
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; //上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;//使能 LINE0
    EXTI_Init(&EXTI_InitStructure);//配置
     SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);//PA0 连接
到线 0

    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;//外部中断 0
     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;//抢占优先
级 2
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;//响应优先级 2
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能外部中断通道
    NVIC_Init(&NVIC_InitStructure);//配置 NVIC

   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
}
```

编写进入待机模式函数，当wakeup按键中断产生调用该函数进入低功耗模式

```
 void StandbyMode(void)
 {
    //关闭不需要的外设
    RCC_AHB1PeriphResetCmd(0X01FF,ENABLE);  //复位所有 IO 口
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);//使能 PWR 时钟
```

```
    PWR_ClearFlag(PWR_FLAG_WU);//清除 Wake-up  标志
    PWR_WakeUpPinCmd(ENABLE);//设置 WKUP 用于唤醒
    PWR_EnterSTANDBYMode();//进入待机模式

 }
```

编写函数 Check_WKUP 实现 WK_UP按下的延时去抖

```
 u8 Check_WKUP(void)
{
    u32 t=0;

    led_on(LED1); //亮灯LED1

    while(1)
    {
        if(0==(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)&0x01))//wakeup键按下
        {
            t++;
        }

        if(t>=600000)//按下延时
        {
            printf("待机\r\n");
            return 2;
        }
    }
}
```

编写wakeup 按键的中断服务函数，当wakeup按键按下产生中断进入待机模式。

```
void EXTI0_IRQHandler(void)
{
    EXTI->PR=1<<0; //清除LINE10上的中断标志位
    if(2==Check_WKUP())//待机?
    {
        Sys_Enter_Standby();
    }
}
```

编写main函数进行测试

```
int main()
{
    init_led();

    init_delay();
```

```
    init_uart();

    printf("开机唤醒\r\n");

    init_wkup();

    while(1)
    {

        led_on(LED2);

        delay_ms(1000);

        led_off(LED2);

        delay_ms(1000);

        printf("主程序while 循环\r\n");

    }

}
```

27.5 软件下载、测试验证

第 28 章

随机数实验

按K1键来获取硬件随机数，并且将获取到的随机数值打印出来

28.1 实验原理与基础知识

STM32F4 自带了硬件随机数发生器RNG 是一个以连续模拟噪声为基础的32位随机数发生器。随机数发生器框图

RNG 控制寄存器(RNG_CR) RNG control register 偏移地址：0x00
RNG 状态寄存器(RNG_SR) RNG status register 偏移地址：0x04
RNG 数据寄存器(RNG_DR) RNG data register 偏移地址：0x08

RNG_DR 寄存器是只读寄存器，在读取时提供32 位随机数值。读取后，此寄存器在最多40 个PLL48CLK 时钟周期后，提供新的随机数值。在读取RNDATA 值之前，软件必须检查DRDY 位是否已置1。

28.2 电路设计

28.3 库函数和数据结构

相关的操作在文件 stm32f4xx_rng.c 和对应的头文件 stm32f4xx_rng.h 中

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									SEIS rc_w0	CEIS rc_w0	Reserved		SECS r	CECS r	DRDY r

位 31:3 保留，必须保持复位值

位 6 **SEIS：**种子错误中断状态 (Seed error interrupt status)

此位与 SECS 同时设置，通过向其写入 0 来清零。

0：未检测到错误序列

1：检测到以下错误序列之一：

– 超过 64 个连续位具有相同值（0 或 1）

– 超过 32 个连续交替的 0 和 1 (0101010101...01)

如果 RNG_CR 寄存器中 IE = 1，则会挂起中断。

位 5 **CEIS：**时钟错误中断状态 (Clock error interrupt status)

此位与 CECS 同时设置，通过向其写入 0 来清零。

0：正确检测到 PLL48CLK 时钟

1：未正确检测到 PLL48CLK 时钟 ($f_{PLL48CLK} < f_{HCLK}/16$)

如果 RNG_CR 寄存器中 IE = 1，则会挂起中断。

位 4:3 保留，必须保持复位值

位 2 **SECS：**种子错误当前状态 (Seed error current status)

0：目前未检测到错误序列。如果 SEIS 位置 1，则意味着已检测到错误序列并已恢复正常。

1：检测到以下错误序列之一：

– 超过 64 个连续位具有相同值（0 或 1）

– 超过 32 个连续交替的 0 和 1 (0101010101...01)

位 1 **CECS：**时钟错误当前状态 (Clock error current status)

0：正确检测到 PLL48CLK 时钟。如果 CEIS 位置 1，则意味着已检测到时钟错误并已恢复正常。

1：未正确检测到 PLL48CLK 时钟 ($f_{PLL48CLK} < f_{HCLK}/16$)。

位 0 **DRDY：**数据就绪 (Data ready)

0：RNG_DR 寄存器尚未有效，无可用随机数据

1：RNG_DR 寄存器包含有效随机数据

注意：如果 RNG_CR 寄存器中 IE = 1，则会挂起中断。

读取 RNG_DR 寄存器后，此位恢复到 0，直到计算出新的有效值。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												IE rw	RNGEN rw	Reserved	

位 31:4 保留，必须保持复位值

位 3 **IE：**中断使能 (Interrupt enable)

0：禁止 RNG 中断。

1：使能 RNG 中断。只要 RNG_SR 寄存器中 DRDY=1 或 SEIS=1 或 CEIS=1，就会挂起中断。

位 2 **RNGEN：**随机数发生器使能 (Random number generator enable)

0：禁止随机数发生器。

1：使能随机数发生器。

位 1:0 保留，必须保持复位值

```
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);
```

AHB2 总线外设时钟使能函数,随机数发生器时钟来自 PLL48CK，通过AHB2ENR 寄存器使能。

```
RNG_Cmd(ENABLE);
```

使能随机数发生器函数 将RNG_CR 寄存器的最低位设置为 1，使能随机数发生器。

```
FlagStatus RNG_GetFlagStatus(uint8_t RNG_FLAG)
```

获取随机数发生器状态的函数,每次读取随机数值之前，必须先判断 RNG_SR寄存器的DRDY 位，如果该位为 1，则可以读取 RNG_DR 得到随机数值，如果不为 1，则需要等待。

```
uint32_t RNG_GetRandomNumber(void);
```

读取随机数发生器产生的随机数函数。

28.4 软件设计、开发

在工程中添加rng.c 文件和rng.h 文件，所有 RNG 相关的驱动代码和定义都在这两个文件中。

初始化硬件随机数发生器

```
int init_rng(void)
{
    u16 retry=0;
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);  //开启 RNG 时钟
    RNG_Cmd(ENABLE);  //使能 RNG
    while(RNG_GetFlagStatus(RNG_FLAG_DRDY)==RESET&&retry<10000)//等待就绪
    {
        retry++;
        delay_ms(1);
    }
    if(retry>=10000)return 1;//随机数产生器工作不正常
    return 0;
}
```

编写读取随机数值函数

```
u32 get_RandomNum(void)
{
    while(RNG_GetFlagStatus(RNG_FLAG_DRDY)==RESET);  //等待随机数就绪
    return RNG_GetRandomNumber();
}
```

硬件随机数发生器产生随机数是32位数，我们通过取模的方式实现产生随机数的范围

```
int get_RandomRange(int min,int max)
{
    return get_RandomNum()%(max-min+1) +min;
}
```

编写main函数进行测试

```
if(button_value==button_read())
{
    switch(button_value)
    {
        case 1:
            led_on(LED1);
            break;

        case 2:
            led_on(LED2);
            printf("random number %d\r\n", get_RandomRange(0,0xFF));
            break;
     }
 }
```

28.5 软件下载、测试验证

第 29 章

CRC 实验

29.1 实验原理与基础知识

CRC（循环冗余校验）计算单元使用一个固定的多项式发生器从一个32 位的数据字中产生 CRC 码。

在众多的应用中，基于CRC 的技术还常用来验证数据传输或存储的完整性。这些技术提供了验证Flash 完整性的方法。

CRC 计算单元有助于在运行期间计算软件的签名，并将该签名与链接时生成并存储在指定存储单元的参考签名加以比较。

CRC 计算单元框图

CRC 计算单元主要由单个32 位数据寄存器组成，该寄存器：

- 用作输入寄存器，向CRC 计算器中输入新数据 （向寄存器写入数据时）
- 可保存之前的CRC 计算结果

对数据寄存器的每个写操作都会把当前新输入的数值和之前生成在数据寄存器中的CRC 值再做一次CRC 计算 （CRC 计算针对整个32 位数据字完成，而非逐字节进行）。

CRC 计算的时候，写操作被阻塞，因此允许执行背靠背写访问或连续的写读访问。使用CRC_CR 寄存器中的RESET 控制位即可将CRC 计算器复位为0xFFFF FFFF。此操作不影响CRC_IDR 寄存器的内容。

CRC 寄存器

CRC 计算单元包含两个数据寄存器和一个控制寄存器。CRC 寄存器必须按字（32 位）访问。

数据寄存器(CRC_DR) Data register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DR [31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR [15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **数据寄存器位**

向 CRC 计算器写入新数据时用作输入寄存器。
读取寄存器时可读出之前的 CRC 计算结果。

偏移地址：0x00

独立数据寄存器(CRC_IDR) Independent data register

偏移地址：0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								IDR[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

位 31:8 保留，必须保持复位值。

位 7:0 **通用的 8 位数据寄存器位**

可用作一个字节的临时存储单元。
此寄存器不受 CRC_CR 寄存器中 RESET 位产生的 CRC 复位影响。

控制寄存器(CRC_CR) Control register

偏移地址：0x08

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															RESET
															w

位 31:1 保留，必须保持复位值。

位 0 **RESET 位**

复位 CRC 计算单元并将数据寄存器设为 0xFFFF FFFF。
此位只能置 1，将由硬件自动进行清零。

29.2 电路设计

29.3 库函数和数据结构

相关的操作在文件 stm32f4xx_crc.c 和对应的头文件 stm32f4xx_crc.h 中

void CRC_ResetDR(void)

用于复位 CRC 数据寄存器

uint32_t CRC_CalcCRC(uint32_t Data)

用于计算 32 位数的 CRC,参数Data 输入的 32 位数,返回 32 位数的 CRC 计算结果

uint32_t CRC_CalcBlockCRC(uint32_t pBuffer[], uint32_t BufferLength)

用于计算多个 32 位数的 CRC。参数: pBuffersection B.2 缓冲区地址 BufferLength 缓冲区长度 ,返回一个 32 位的 CRC 计算结果。

uint32_t CRC_GetCRC(void)

返回当前的 CRC 数值

void CRC_SetIDRegister(uint8_t IDValue)

存入一个 8 位数据到独立数据寄存器。

uint8_t CRC_GetIDRegister(void)

从独立数据寄存器读出 8 位数据。

29.4 软件设计、开发

29.5 软件下载、测试验证

第 30 章

CPU UID实验

编程实现CPU UID （唯一设备ID） 96 位，并打印出来。

30.1 实验原理与基础知识

唯一设备标识符可以用于:

● 用作序列号（例如USB 字符串序列号或其它终端应用程序）

● 在对内部Flash 进行编程前将唯一ID 与软件加密原语和协议结合使用时用作安全密钥以

提高Flash 中代码的安全性

● 激活安全自举过程等

96 位的唯一设备标识符提供了一个对于任何设备和任何上下文都唯一的参考号码。用户永远不能改变这些位。

96 位的唯一设备标识符也可以以单字节/半字/字等不同方式读取，然后使用自定义算法连接起来。

唯一设备ID存储在Flash 区,其所对应的基址为0x1FFF7A10连续3个字。

30.2 电路设计

30.3 库函数和数据结构

30.4 软件设计、开发

读取的方式跟读取内存地址的方式一样:

```
int UID_baseaddr=0x1FFF7A10;
unsigned int CPU_UID[3];

for(i=0;i<3;i++)
{
    CPU_UID[i]=* ((vu32*)UID_baseaddr +i*4) ;
    printf("CPU_UID[%d] =%d\r\n",i,CPU_UID[i]);
}
```

30.5 软件下载、测试验证

第 31 章

DHT11 实验

芯片内部有温度传感器，有兴趣的同学可以尝试编程读出温度值，本次实验我们使用的是板载的DTH11温湿度传感器，每隔1s将读取的温湿度值打印在串口终端和LCD上。

31.1 实验原理与基础知识

DHT11 是一款湿温度一体化的数字传感器。该传感器包括一个电阻式测湿元件和一个NTC测温元件，并与一个高性能 8 位单片机相连接。通过单片机等微处理器简单的电路连接就能够实时的采集本地湿度和温度。 DHT11 与单片机之间能采用简单的单总线进行通信，仅仅需要一个 I/O 口。传感器内部湿度和温度数据 40Bit 的数据一次性传给单片机，数据采用校验和方式进行校验，有效的保证数据传输的准确性。

技术参数

供电电压： 3.3～5.5V DC

输 出： 单总线数字信号

测量范围： 湿度20 90%RH， 温度0～50℃

测量精度： 湿度+ 5%RH， 温度+ 2℃

— 分 辨 率： 湿度1%RH， 温度1℃

— 互 换 性： 可完全互换 ，

— 长期稳定性： ＜±1%RH/年

DHT11 数字湿温度传感器采用单总线数据格式。即，单个数据引脚端口完成输入输出双向传输。其数据包由 5Byte（40Bit）组成。数据分小数部分和整数部分，一次完整的数据传输为40bit，高位先出。DHT11 的数据格式为：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据+8bit 校验和(前四个字节相加)。

用户MCU发送一次开始信号后,DHT11从低功耗模式转换到高速模式,等待主机开始信号结束后,DHT11发送响应信号,送出40bit的数据,并触发一次信号采集,用户可选择读取部分数据.从模式下,DHT11接收到开始信号触发一次温湿度采集,如果没有接收到主机发送开始信号,DHT11不会主动进行温湿度采集.采集数据后转换到低速模式。

DHT11 的传输时序

总线空闲状态为高电平,主机把总线拉低等待DHT11响应,主机把总线拉低必须大于18毫秒,保证DHT11能检测到起始信号。DHT11接收到主机的开始信号后,等待主机开始信号结束,

然后发送80us低电平响应信号.主机发送开始信号结束后,延时等待20 40us后,读取DHT11的响应信号,主机发送开始信号后,可以切换到输入模式,或者输出高电平均可,总线由上拉电阻拉高。

总线为低电平,说明DHT11发送响应信号,DHT11发送响应信号后,再把总线拉高80us,准备发送数据,每一bit数据都以50us低电平时隙开始,高电平的长短定了数据位是0还是1.格式见下面图示.如果读取响应信号为高电平,则DHT11没有响应,请检查线路是否连接正常.当最后一bit数据传送完毕后, DHT11拉低总线50us,随后总线由上拉电阻拉高进入空闲状态。数字0信号表示方法如图4所示

数字1信号表示方法.

31.2 电路设计

31.3 库函数和数据结构

31.4 软件设计、开发

在工程中添加 dht11.c 文件和dht11.h 文件, 所有 DHT11 相关的驱动代码和定义都在这两个文件中。

从原理图上可知STM32F407是通过GPB0与DHT11相连, 我们编写管脚初始化函数

70us
表示'1'
VCC
下一bit开始
GND
1Bit开始
单总线
50us
信号线说明:
主机信号
DHT信号

Variable resistor
Temperature Humidity Sensor
ADC1
R13
0R
3.3V
R14
RES_POT
3.3V
VCC
DATA
NC
GND
DHT11
J25
R169
5K
ADC2
C59
100nF
J27
CON3
ADC1
ADC
ADC2

```
void init_GPB0(void)
```

一总线传输，GPB0根据时序需要IO输入输出的方向，编写如下宏定义：

```
//IO输入输出设置
#define DHT11_IO_IN()  {GPIOB->MODER&=~(3<<(0*2));GPIOB->MODER|=0<<0*2;}   //
PB0输入模式
#define DHT11_IO_OUT() {GPIOB->MODER&=~(3<<(0*2));GPIOB->MODER|=1<<0*2;}   //
PB0输出模式
```

编写GPB0输出函数和读取函数：

```
void DHT11_OUT(u8 value)
{
    if(1==value)
        GPIO_SetBits(GPIOB,GPIO_Pin_0);
    else
        GPIO_ResetBits(GPIOB,GPIO_Pin_0);

}

u8 DHT11_IN(void)
{
    return 0x01 & GPIOB->IDR;

}
```

编写DHT11复位函数 和 响应检查函数：

```
void DHT11_Rst(void)
{
    DHT11_IO_OUT();     //SET OUTPUT
    DHT11_OUT(0);       //拉低
    delay_ms(20);       //拉低至少18ms
    DHT11_OUT(1);       //拉高
    delay_us(30);       //主机拉高20~40us
}
```

编写读一个bit 读一个字节，读整个40位的数据

```
u8 DHT11_Read_Bit(void)
{
    u8 retry=0;
    while(DHT11_IN()&&retry<100)//等待变为低电平
    {
        retry++;
        delay_us(1);
    }
    retry=0;
```

```
    while(!DHT11_IN()&&retry<100)//等待变高电平
    {
        retry++;
        delay_us(1);
    }
    delay_us(40);//等待40us
    if(DHT11_IN())
        return 1;
    else
    return 0;
}

u8 DHT11_Read_Byte(void)
{
    u8 i,dat;
    dat=0;
    for (i=0;i<8;i++)
    {
        dat<<=1;
        dat|=DHT11_Read_Bit();
    }
    return dat;
}

u8 DHT11_Read_Data(u8 *temp,u8 *humi)
{
    u8 buf[5];
    u8 i;
    DHT11_Rst();
    if(DHT11_Check()==0)
    {
        for(i=0;i<5;i++)//读取40位数据
        {
            buf[i]=DHT11_Read_Byte();
        }
        if((buf[0]+buf[1]+buf[2]+buf[3])==buf[4])
        {
            *humi=buf[0];
            *temp=buf[2];
        }
    }
    else
        return 1;
    return 0;
}
```

编写main函数进行测试

```
int main()
{
    u8 temperature;

    u8 humidity;

    DrawMode Mode;

    Mode.FColor = 0xF800;
    Mode.BColor = 0x001F;
    Mode.Mode = 0;

    init_delay();

    init_uart();

    init_DHT11();

    init_lcd();

    while(1)
    {

        DHT11_Read_Data(&temperature,&humidity);        //读取温湿度值

     printf("DHT11  temperature: %d  humidity: %d\r\n",temperature,humidity);

        LCD_DisplayNum(0, 80, temperature, &Mode);

        LCD_DisplayNum(0, 100, humidity, &Mode);

        delay_ms(1000);
    }


}
```

31.5 软件下载、测试验证

附录 A

相关资源

如果在公司培训，这是一个模板，建议根据实际情况修改。

A.1 工具链接

A.2 视频资料

A.3 开源项目

附录 B

常见问题汇总

B.1 编译下载

在推动技术改变的时候，首先要了解最新的技术状况，别学了一个旧了过时了的。

我们使用的主要是ClearCase，开始考虑这个转换的时候是在2009年初，SVN是第一个考虑的对象，因为它在开源中用的最多，sourceforge[1]和Eclipse的很多项目多用它，但我总觉得缺了点什么。

恰好我有个同事提到SVN和ClearCase都是集中式的，推荐我看看一个分布式版本控制工具：Mercurial，说实话听了介绍不是很懂，没有眼前一亮的感觉。聊了一下，感觉和SVN的分支没有多大区别，何况DVCS还需要两层提交呢。

同时我也了解到还有其它的分布式版本控制工具Git，Bazaar可供选择。

不管怎么样，我了解到这块领域有了最新的技术，它或许能解决我们的问题（要不时地问问自己为什么）。

B.2 接口编程

为了尽快了解DVCS，我决定要在日常的开发中用用它，实践它，尽快地掌握它的关键。

[1] http://sourceforge.net