

# 零死角玩转STM32

与野火同行 乐竟惬意无边

1

原创教程，完全开源。

3

由浅入深，结合实操。

2

通俗易懂，详尽解读。

4

配套板子，全面玩转。

5

强强联合，不断更新。



野火团队 Wild Fire Team

## 野火 ISO / 硬件介绍篇

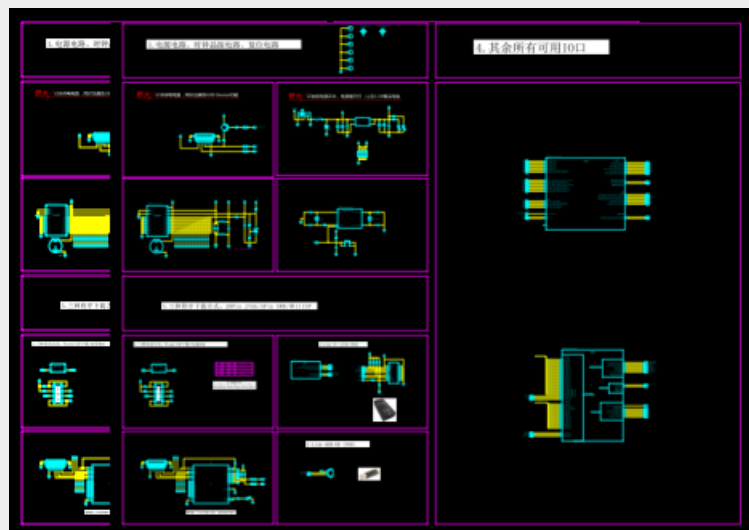
## 1. 原理图与硬件图

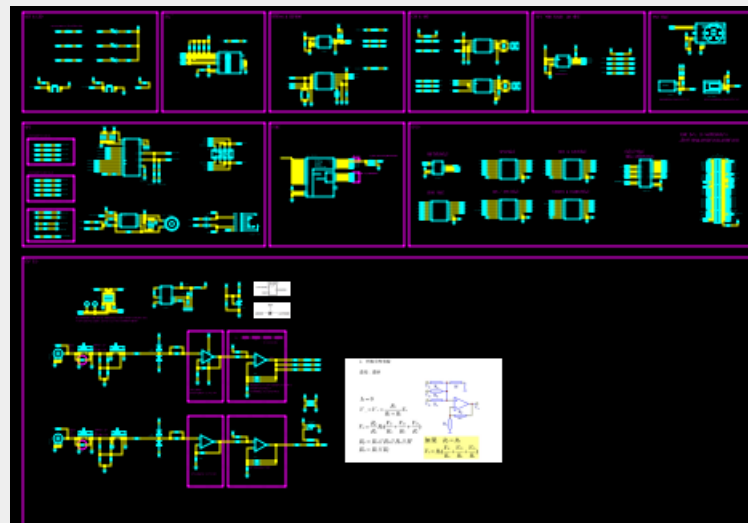
### 1.1. 原理图

ISO 原理图分为两个 sheet，sheet1 为最小系统部分，sheet2 为板载资源和一些接口。

STM32 入门，看原理图当如从最小系统开始看起。

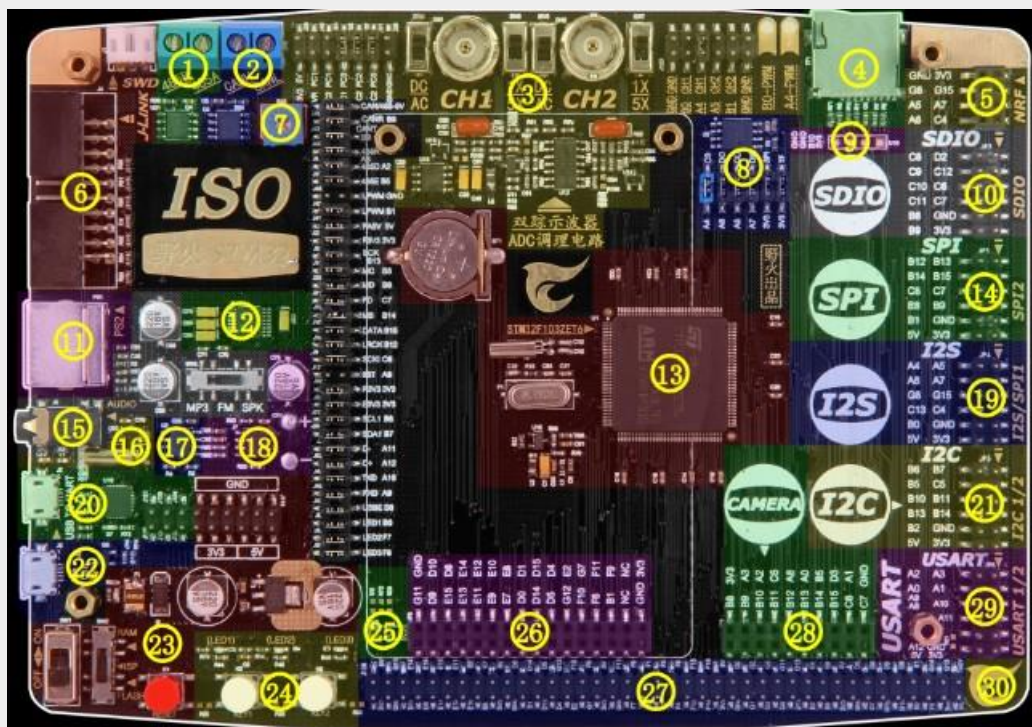
Sheet1 里面最主要的是：供电电路、复位电路、晶振电路、启动模式、下载电路。其中定位孔、mark 点、gpio 等我们也把放到了 sheet1。定位孔的直径是 3mm。



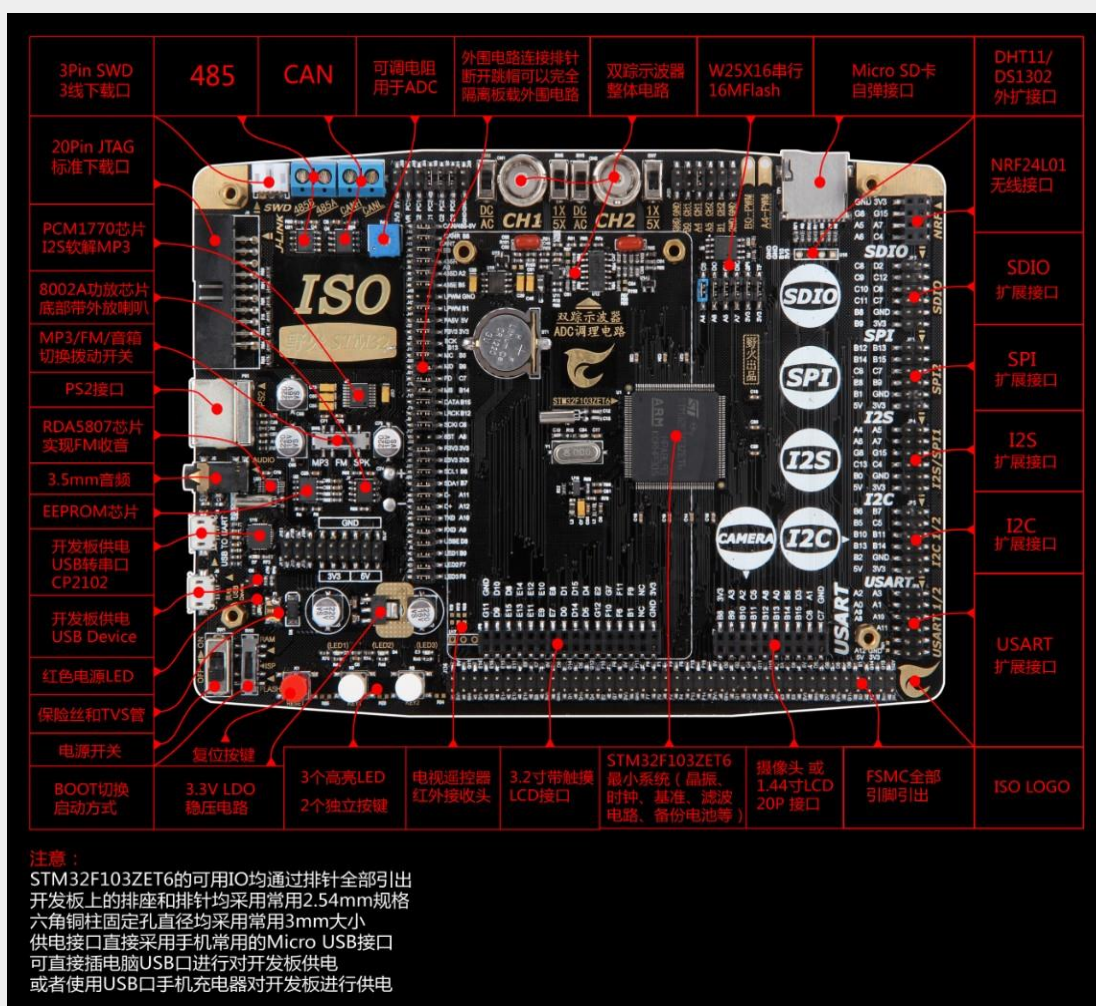




## 1.2. 硬件图



|    |                                        |
|----|----------------------------------------|
| 1  | 485通讯接口                                |
| 2  | CAN通讯接口                                |
| 3  | 双踪示波器                                  |
| 4  | Micro SD卡接口                            |
| 5  | NRF24L01无线接口                           |
| 6  | J-Link V8 20Pin和SW 3Pin下载或者调试接口        |
| 7  | 20K可变电阻，用于ADC电压检测                      |
| 8  | 串行Flash及其引脚引出                          |
| 9  | DHT11温湿度传感器接口或者DS18B20接口               |
| 10 | SDIO接口引出                               |
| 11 | PS2接口（可接键盘或者鼠标）                        |
| 12 | I2S软解码MP3电路                            |
| 13 | 最小系统所需的电路外围（时钟，晶振，备份电池，基准电路，滤波电路）      |
| 14 | SPI接口引出                                |
| 15 | 3.5MM音频引出，可以通过拨动开关实现MP3、FM输出，或者音频输入。   |
| 16 | FM电路                                   |
| 17 | EEPROM电路                               |
| 18 | 喇叭功放电路，底部带有2w外置喇叭一个，可外放MP3、FM          |
| 19 | I2S接口引出                                |
| 20 | CP2102 USB转串口电路（可作为开发板供电接口）            |
| 21 | I2C接口引出                                |
| 22 | USB Device接口（可作为开发板供电接口）               |
| 23 | 开发板电源开关、Boot启动方式、保险丝、LDO稳压、复位电路、电源引出接口 |
| 24 | 2个按键+3个LED电路                           |
| 25 | 红外接口                                   |
| 26 | 3.2寸带触摸LCD接口                           |
| 27 | FSMC全部接口，可以直接外扩SRAM、NOR/NAND Flash等    |
| 28 | 摄像头接口                                  |
| 29 | USART串口接口引出                            |
| 30 | 野火 ISO LOGO                            |



## 野火 ISO/基础入门篇

## 1. 搭建开发环境

在拿到开发板之后，很多朋友都跃跃欲试，想自己烧写个程序到开发板上，看看效果。下面我们将演示如何将光盘里面的程序烧写到 **ISO**（**ISO** 是野火最新的 **STM32** 开发板的代号）上，但前提是我们的电脑上必须已经安装了 **JLINK** 驱动和 **MDK** 开发环境，没有安装的朋友请参考我们的教程。光盘上提供的代码都是已经编译好的，直接下载即可，如果不行，请重新编译一次。

| 名称                       | 修改日期            | 类型  |
|--------------------------|-----------------|-----|
| 00-野火ISO 视频教程            | 2013/9/7 21:08  | 文件夹 |
| 01-野火ISO PDF教程           | 2013/9/12 16:40 | 文件夹 |
| 02-野火ISO 程序              | 2013/9/10 9:18  | 文件夹 |
| 03-原理图                   | 2013/9/9 11:52  | 文件夹 |
| 04-数据手册                  | 2013/9/9 10:56  | 文件夹 |
| 05-开发软件                  | 2013/9/9 21:28  | 文件夹 |
| 06-配套书籍《STM32裸开发实战指南》... | 2013/9/11 22:46 | 文件夹 |

### 1.1. 安装 **JLINK** 驱动

在用 **JLINK** 下载和调试程序之前，我们需要在电脑上安装 **JLINK** 驱动，如果电脑上已经安装 **JLINK** 驱动，则可跳过这一步。在光盘目录下：**05-开发软件** \ **1-JLINK 驱动** 点击 **Setup\_JLinkARM\_V458c.exe** 安装即可。

| 05-开发软件 ▶ 1-JLINK驱动 ▶         |                  |     |  |
|-------------------------------|------------------|-----|--|
| 帮助(H)                         |                  |     |  |
| 刻录 新建文件夹                      |                  |     |  |
| 名称                            | 修改日期             | 类型  |  |
| Setup_JLinkARM_V458a.zip      | 2012/12/10 21:18 | 好压  |  |
| Setup_JLinkARM_V460a.zip      | 2013/1/24 1:39   | 好压  |  |
| 读完我.txt                       | 2013/1/24 1:55   | TXT |  |
| 使用SAM-BA更新jlink固件20121126.rar | 2013/1/24 14:30  | 好压  |  |



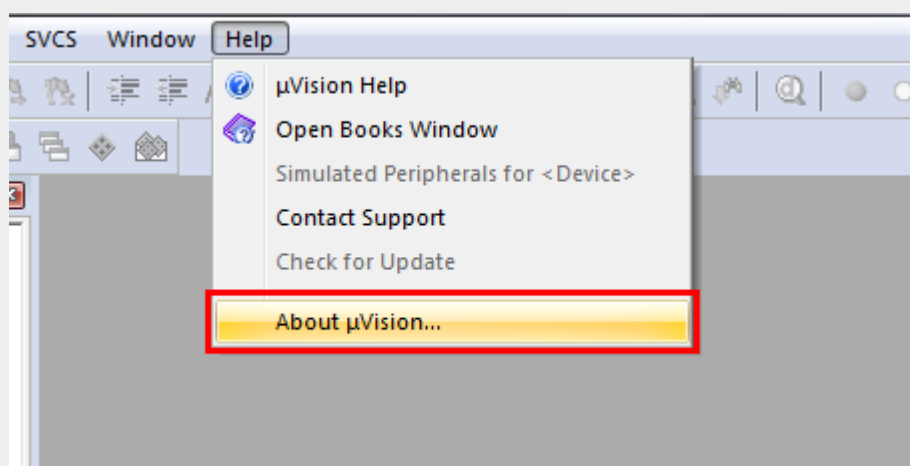
安装过程非常简单，这里将跳过。

在安装完成后，我们将 JLINK 插接到电脑的 USB 口，即可在我的电脑\管理\设备管理器\通用串行总线控制器中看到一个 J-Link driver。要注意的是在安装完 JLINK 驱动后，一定要将 JLINK 插接到电脑的 USB 口，否则在电脑的设备管理器中是查看不到 J-Link driver 的。当你把 JLINK 拔出电脑的 USB 口时候，J-Link driver 就会消失。



## 1.2. 安装 MDK

在我们编写代码之前需要先把 MDK 这个软件安装好，野火用的版本是 V4.72，在安装完成之后可以在工具栏 help->about uVision 选项卡中查看到版本信息。

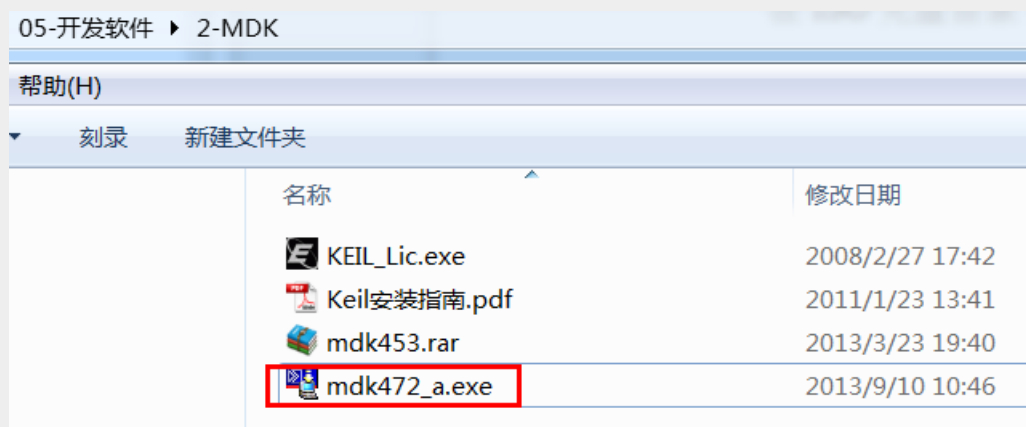


4.72 版本较低版本多了个语法检查的功能，非常人性化，当你写代码的时候，想犯错都犯不了，就像在 padslogic 中你不可能把原理图画错一个样（protel 则没有这个功能）。当出现语法错误的时候，则有个红色的 X。

```
17 L
18 #include "stm32f10x.h"
19
20 int variable;
21 int va
22
23 /**
24  * @brief 主函数
25  * @param 无
26  * @retval 无
27  */
```

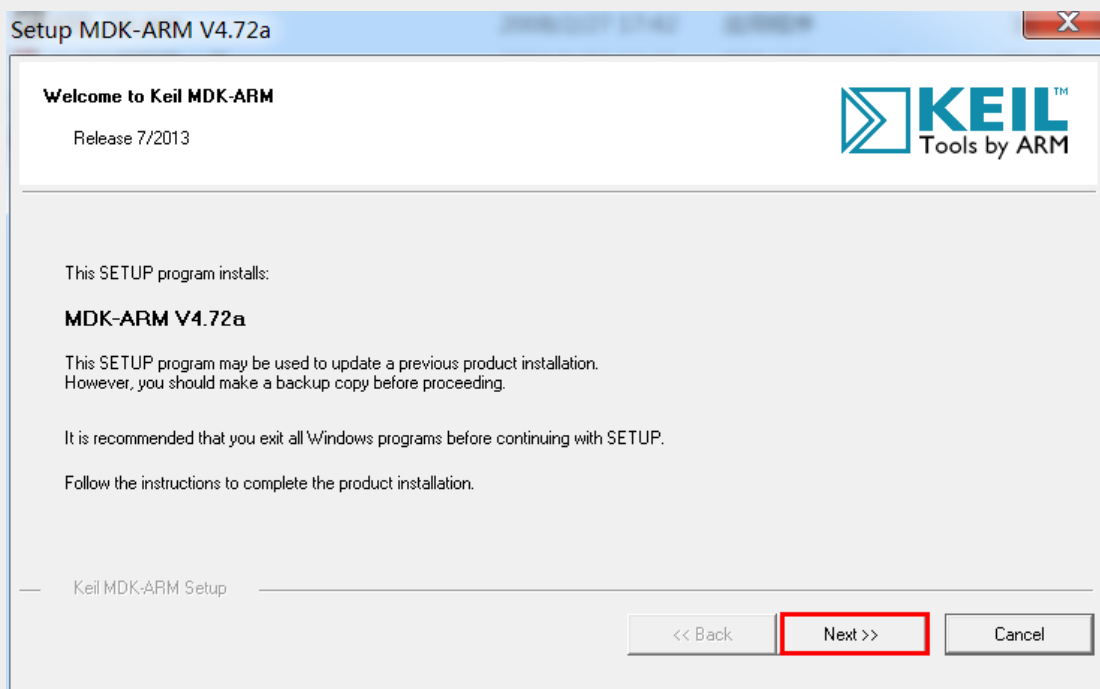
MDK 是一个集代码编辑，编译，链接和下载于一体的集成开发环境（KDE）。MDK 这个名字我们可能不熟悉，但说到 KEIL，学过 51 的朋友就再熟悉不过了。后来 KEIL 被 ARM 公司收购之后就改名为 MDK 了，所以学过 51 的朋友是很快就可以熟悉这个开发环境的。

在光盘目录下：05-开发软件\2-MDK 找到 MDK472\_a.exe。

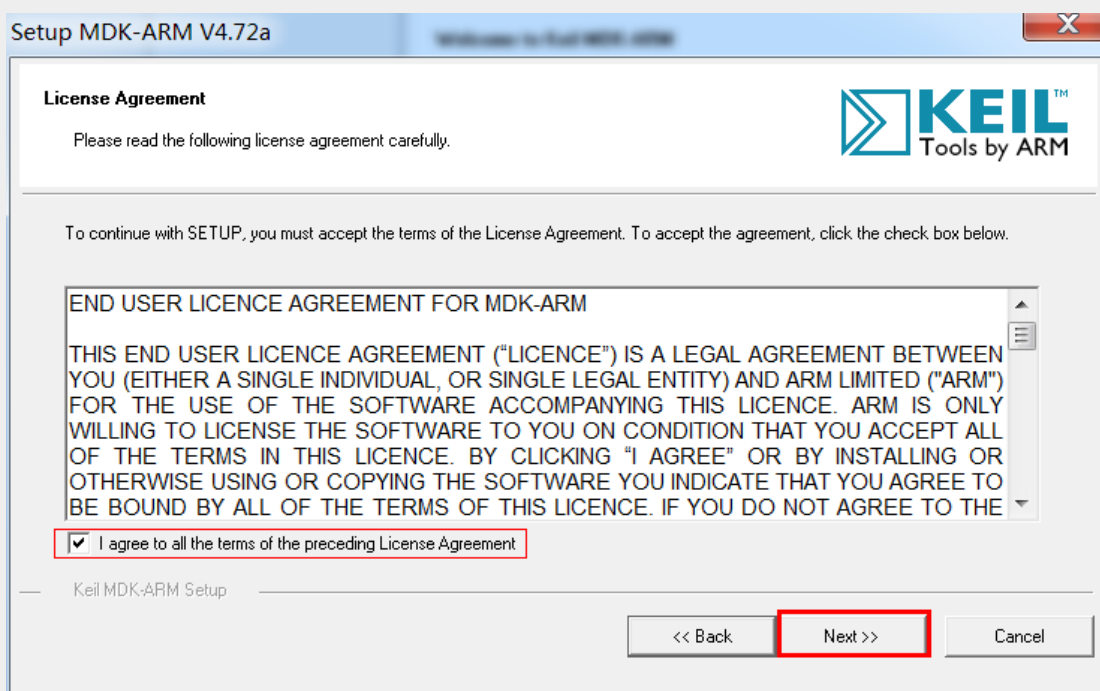


点击 MDK472\_a.exe，在弹出 MDK 安装界面后，按照如下步骤操作即可。如果之前你已经安装了低于 4.72 版本的 MDK 的话，发现打不开工程，那么请升级到 4.72 版本的 MDK。

- 点击 Next。

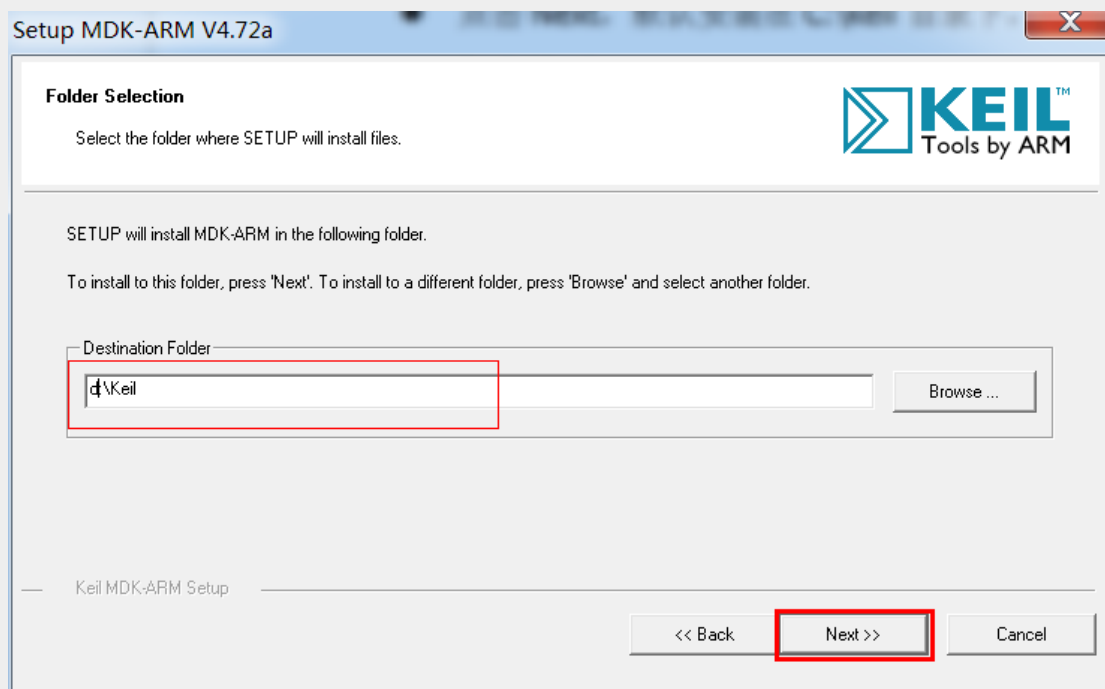


- 把勾勾上，点击 Next。

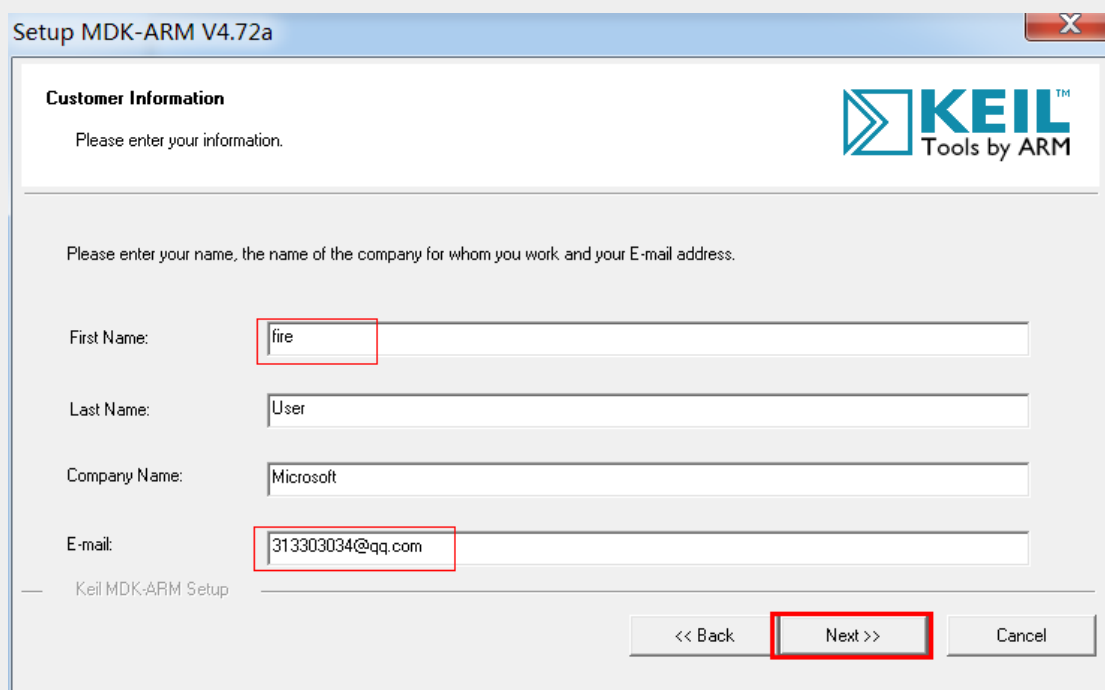


- 点击 Next，默认安装在 C:\keil 目录下，可根据自己的需要安装在其他盘，如 D:\keil。

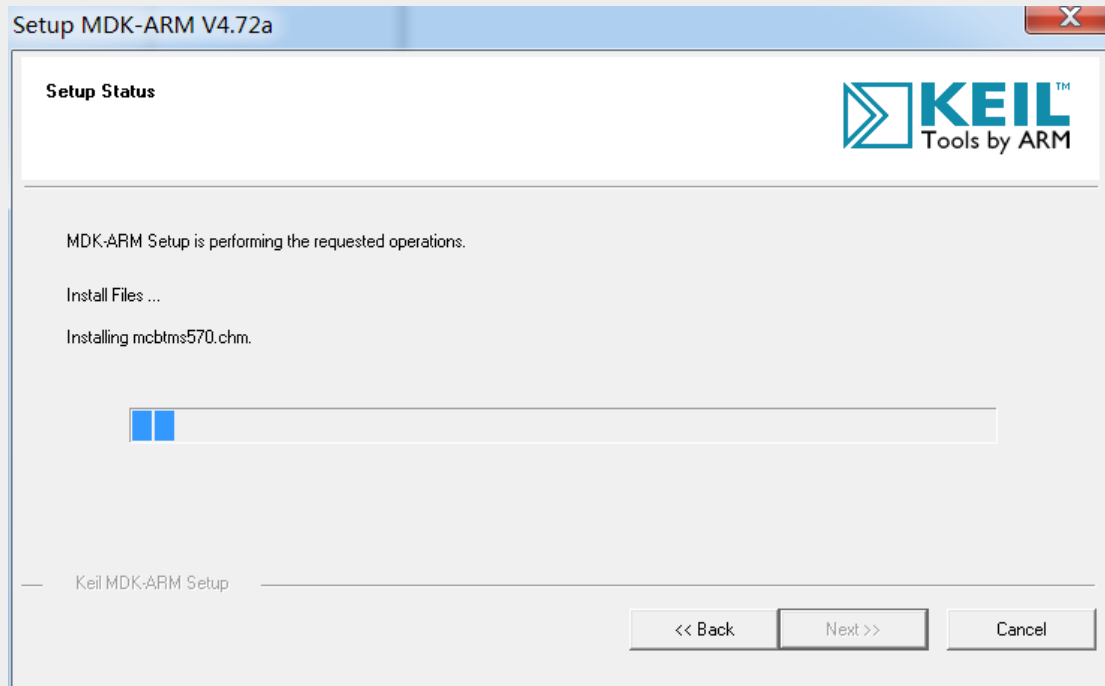




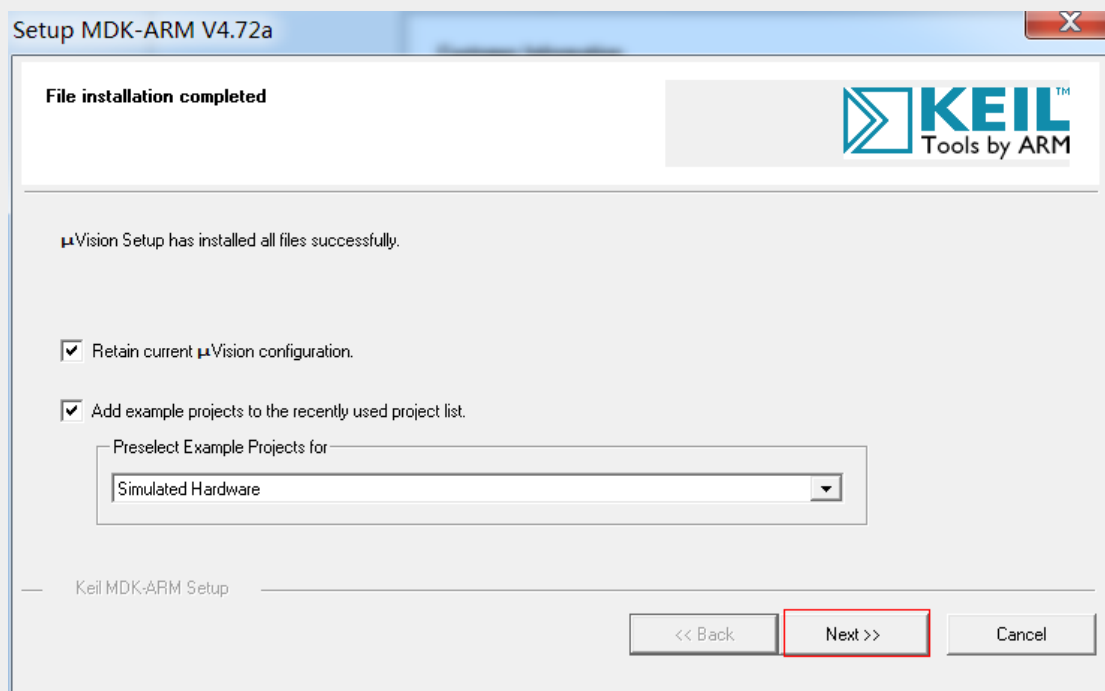
- 在用户名中填入名字（可随便写或者空格），在邮件地址那里填入邮件地址（可随便写或者可空格），点击 **Next**。

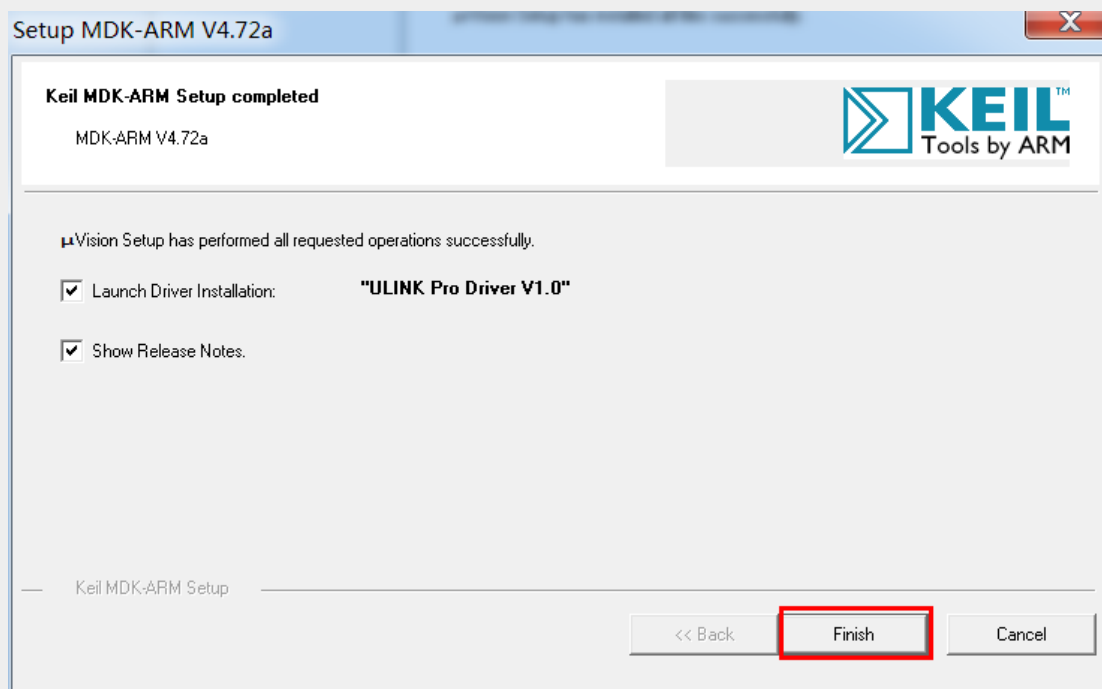


- 正在安装，请耐心等待。



- 点击 Finish，安装完成。

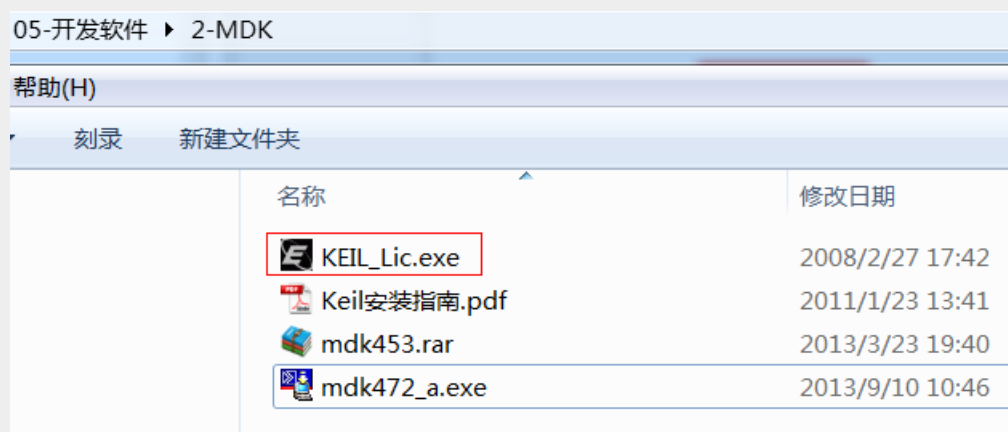




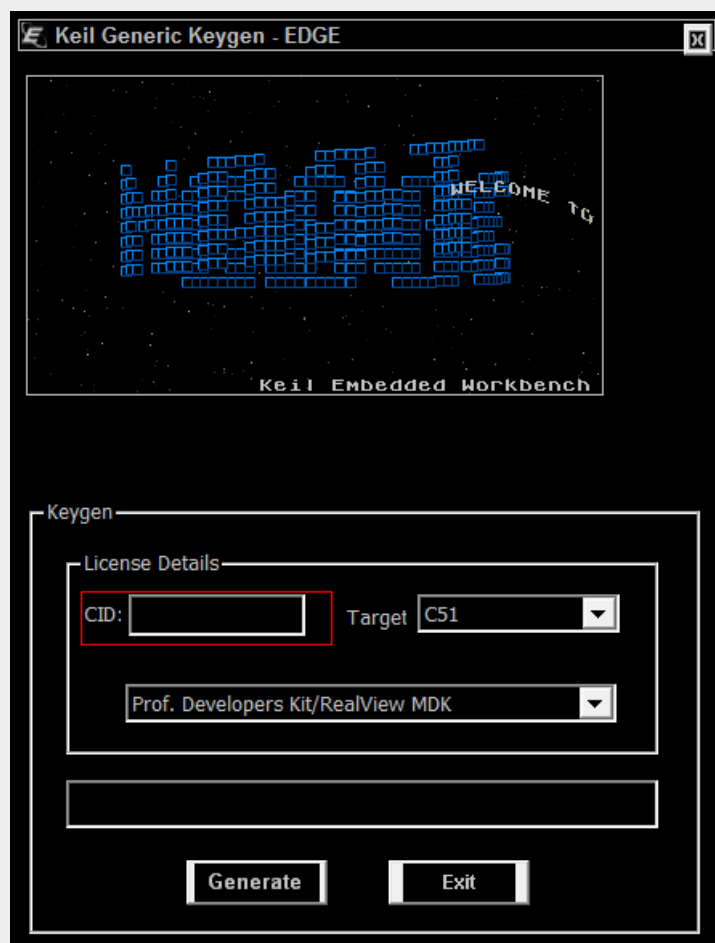
- 此时就可在桌面看到 MDK 的快捷图标，如右所示：

### 1.3. 和谐 MDK

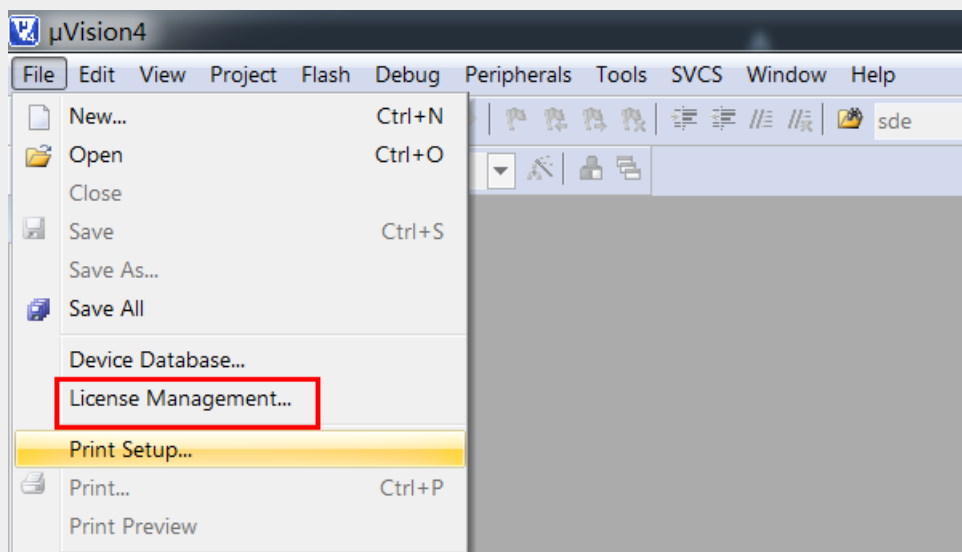
安装完 MDK 开发环境后，在编译程序的时候会有 40K 的代码限制，我们只需要和谐下即可搞定。在光盘目录下：05-开发软件\2-MDK 找到 KEIL\_Lic.exe。

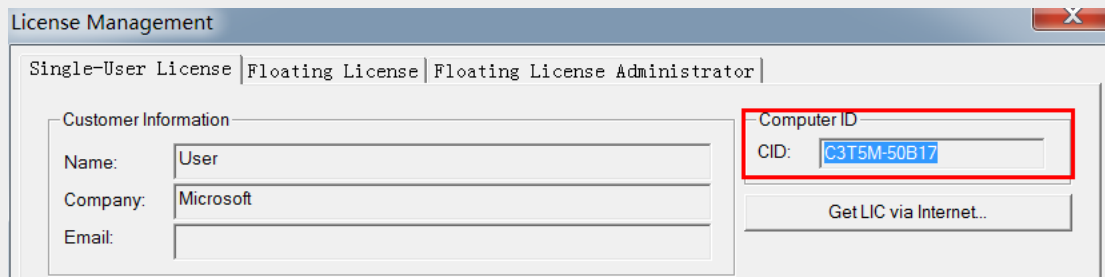


点击 KEIL\_Lic.exe，在弹出的界面中的 CID 选项框中填入 MDK 的 CID。

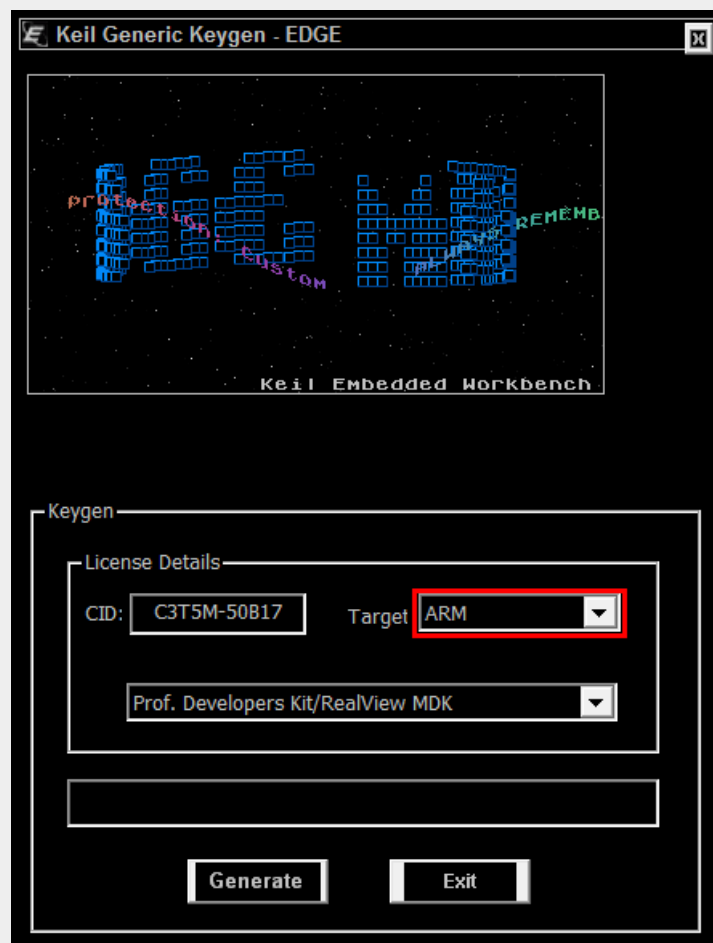


其中 MDK 的 CID 在 MDK 开发环境中的菜单栏 File\License Management 中获取到

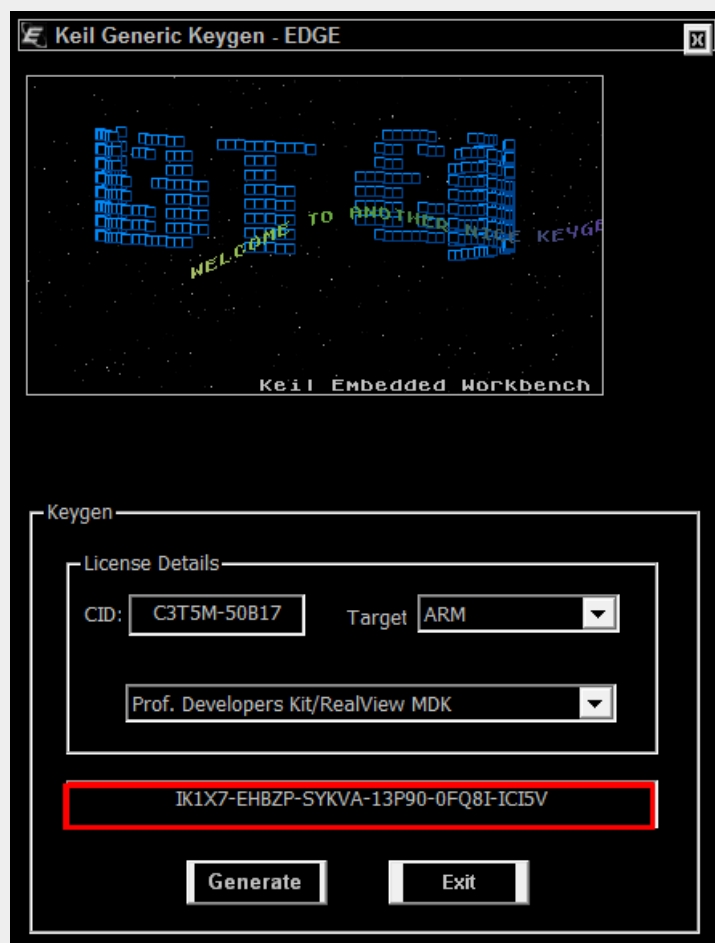




在 Target 下拉框中选择 ARM

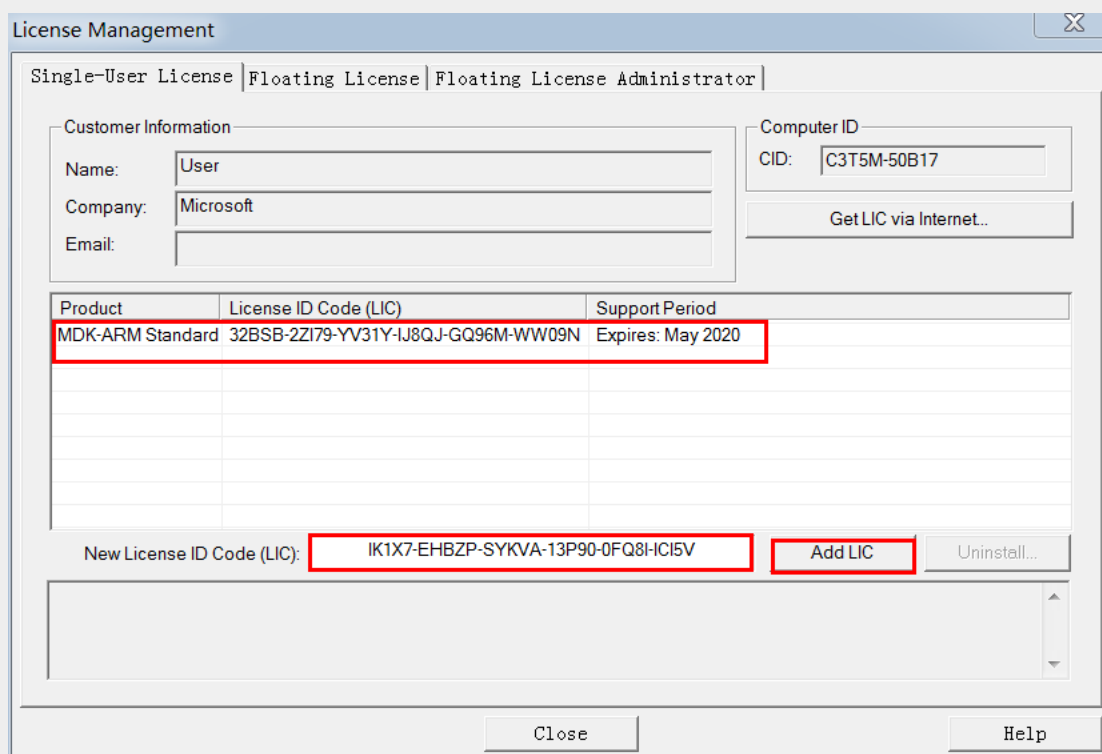


然后点击 Generate 按钮，复制产生的 CID Code



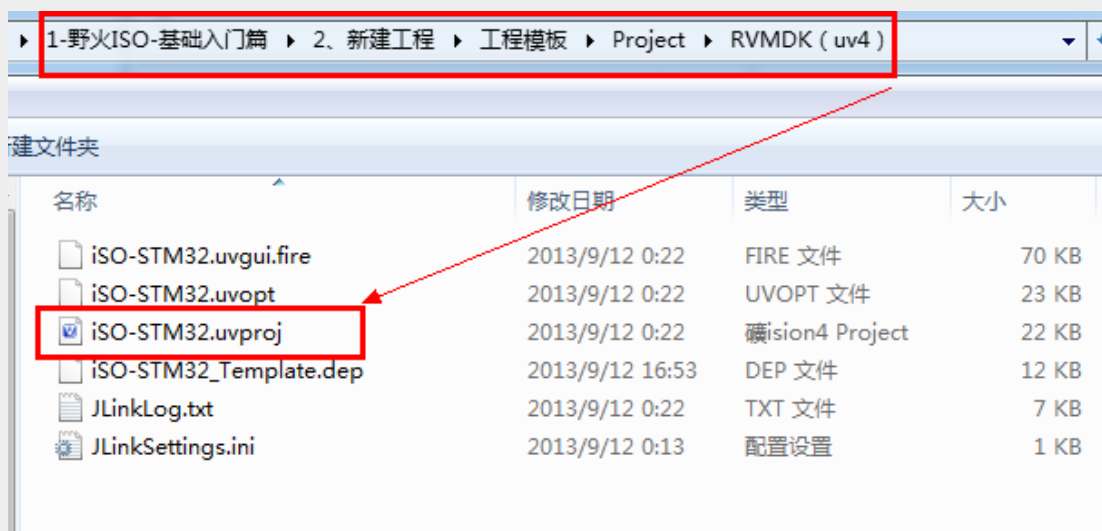
然后回到 MDK 开发环境中的菜单栏 File\License Managemant 中，把刚刚在注册机复制到的 CID Code 粘贴到 New License ID Code (LIC):框中，然后点击 Add LIC，点击 close，显示可以用到 2020 年，大功告成。



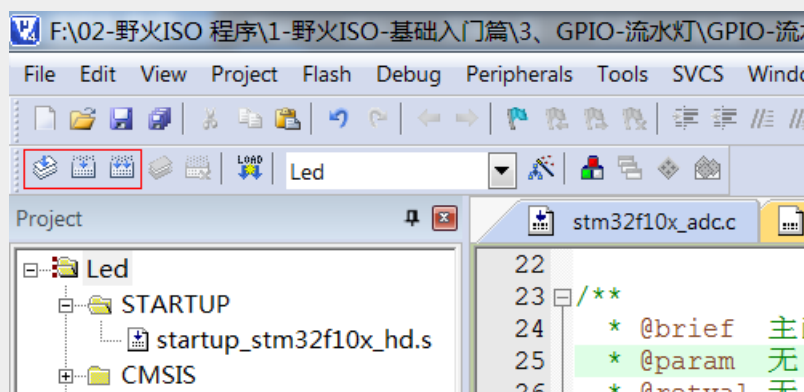


#### 1.4. 1.4 编译程序

在光盘目录下，随便打开一个 程序的工程。



在弹出的 MDK 界面中，我们可以看到左边的工具栏中有三个按钮，现在我们从左往右来介绍下这三个按钮的功能。



- 第一个按钮：**Translate** 就是翻译当下修改过的文件，说明白点就是检查下有没有语法错误，并不会去链接库文件，也不会生成可执行文件。
- 第二个按钮：**Build** 就是编译当下修改过的文件，它包含了语法检查，链接动态库文件，生成可执行文件。
- 第三个按钮：**Rebuild** 重新编译整个工程，跟 **Build** 这个按钮实现的功能是一样的，但有所不同的是它编译的是整个工程的所有文件，耗时巨大。

综上：当我们编辑好我们的程序之后，只需要用第二个 **Build** 按钮就可以，即方便又省时。第一个跟第三个按钮用的比较少。

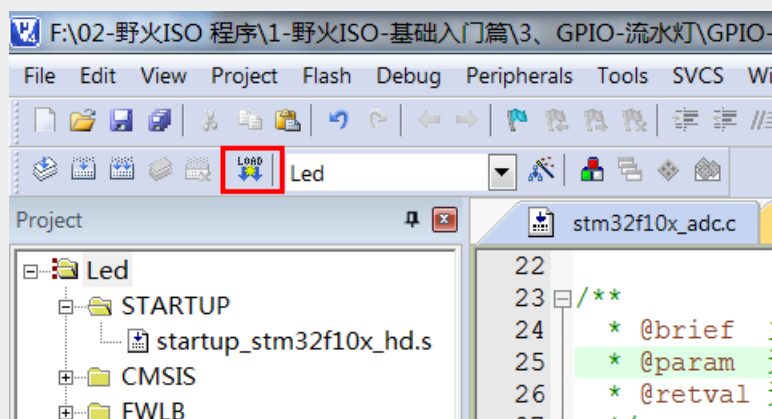
### 1.5. 下载程序

**ISO** 有两种下载方式，**JLINKV8** 下载和串口下载（**ISP**）。要记住，无论是 **JLINK** 下载还是串口下载，最终都要把 **BOOT** 开关打到 **FLAHS** 档位，程序才可以运行。特别是在串口下载时 **BOOT** 开关打到了 **ISP**，在下载完程序后，要记得把开关打回 **FLASH** 档位。

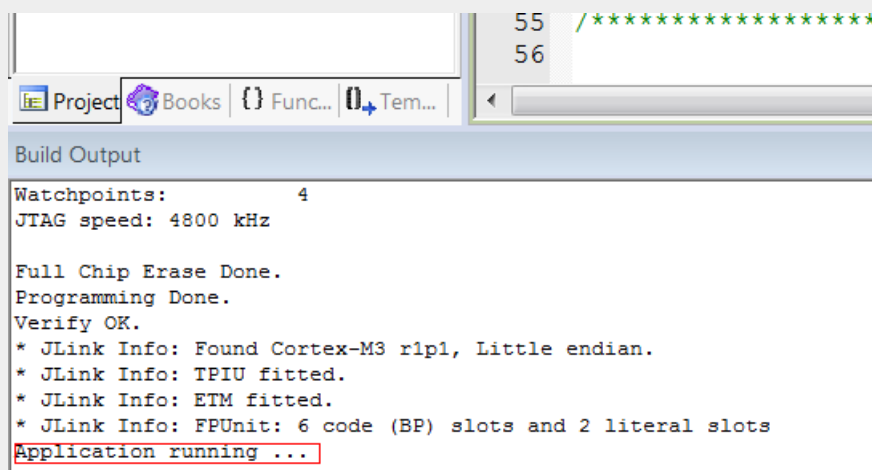
### 1.5.1. JLINK 下载



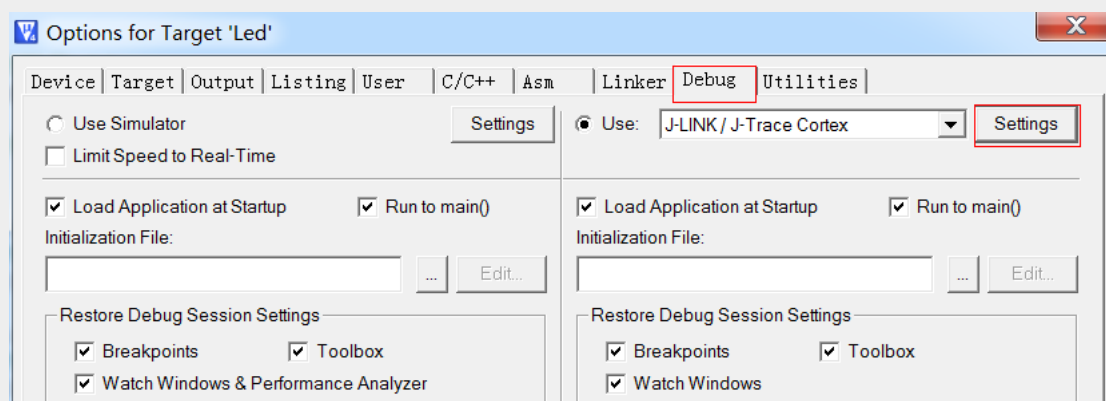
- ① 用 JLINK 连接电脑和开发板，给开发板上电。
- ② 点击 MDK 工具栏中的 Load 按钮就可将编译好的程序下载到开发板中。

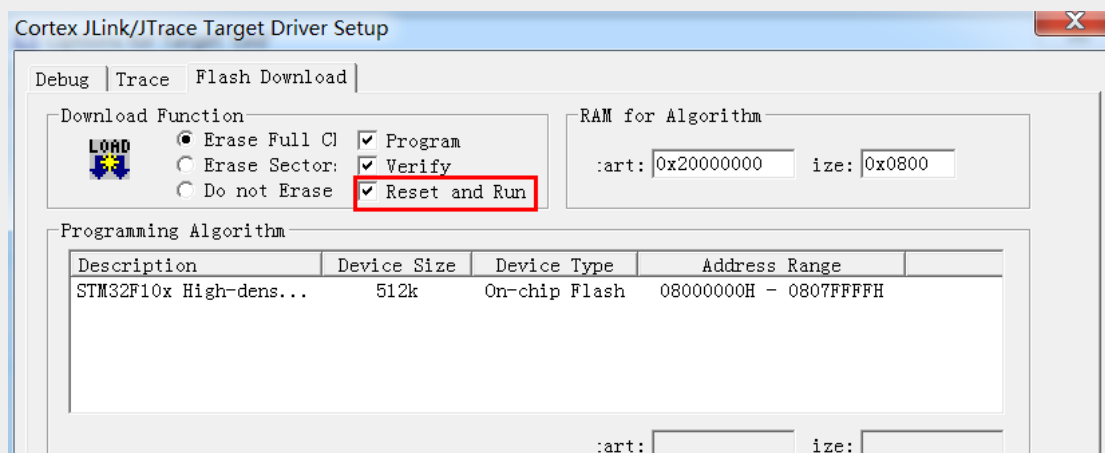


- ③ 下载成功之后，程序就会自动运行。如果发现程序没有运行，则可按下开发板中的复位按键（红色的那个）。



这里要注意的是：程序在烧写到开发板后是否自动运行，是可以在 MDK 开发环境：Target Options...->Debug->Setting->Flash Download 中设置的：



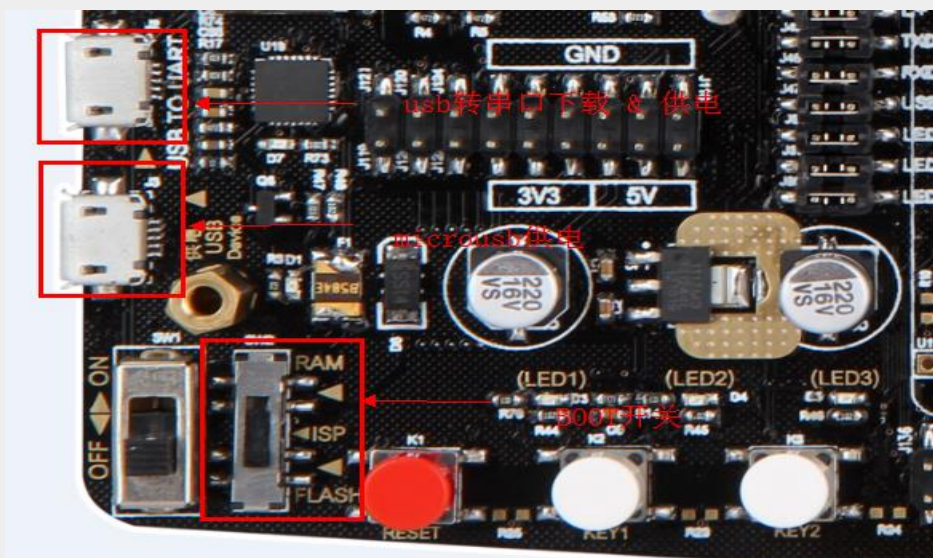


这里的 **Reset** 是通过 **JLINK** 里面的管脚来复位 **CPU**，如果这个勾没有勾上的话，我们可以手动复位。

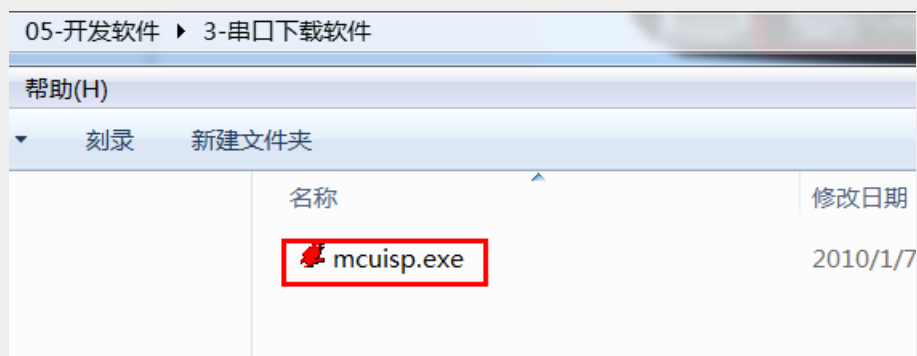
### 1.5.2. 串口下载

**ISO** 里面板载了 **USB** 转串口芯片电路，在开发板用 **microusb** 线连接好电脑后就可以用 **mcuisp** 下载程序。板子上用的 **USB** 转串口芯片是 **CP2012**，波特率可稳定在 **115200** 传输。

- ① 给开发板供电。
- ② 将 **BOOT** 开关拨到中间的 **ISP** 档位。



在这里我们用的串口下载软件是 **mcuisp**，这个一个绿色的软件。



③ 点击 `mcuisp.exe`，打开 `mcuisp`，`mcuisp` 是很智能的，只要开发板上电且连接好了 `usb` 线，它就会自动搜索串口，如果没有自动搜索到的话，我们可以手动搜索。

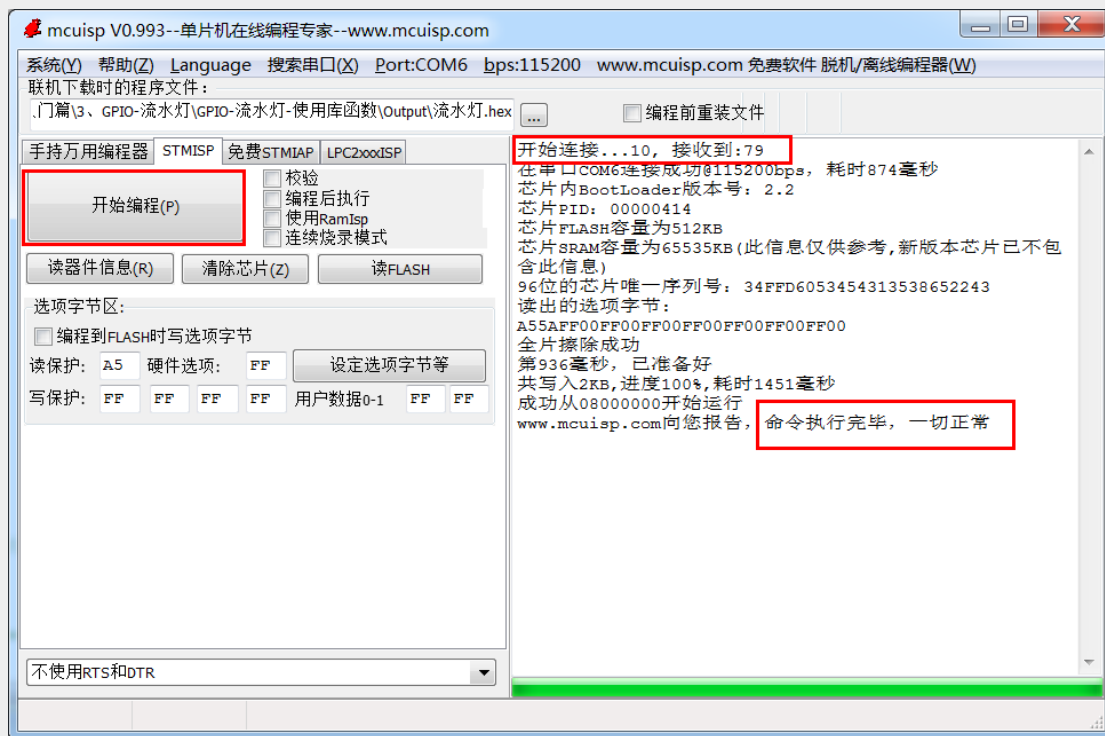


④ 然后设置波特率为 115200，也可设置为 9600。





⑤ 然后点击 开始编程 按钮，当右边的框框出现开始连接...的时候，按一下复位键，则程序就会开始下载。当下载完成的情况下，则会显示“命令执行完毕，一切正常”。



⑥ 这时程序还不会运行，需要把 BOOT 开关拨到 FLASH 档位，然后再按一下复位键，这个时候程序就运行起来了。

### 1.5.3. 串口下载与 JLINK 下载对比

#### ● 串口下载

- 优点：速度快，下载稳定，特别是下载大型程序的时候。如果你的板子用的 MAX3232 或者是 CH340 国产的毛片的话，则没有这个优点。
- 缺点：不能够在线调试。程序不能在线调试的话就等于到了东莞 ISO 发现没有带钱那么难受，瞎蒸腾。

#### ● JLINK 下载

- 优点：可以在线调试，开发一大利器，必不可少。有 **JLINK**，犹如倚天屠龙在手。要想提高自己的编程能力，学会调试是你永远绕不过去的坎。

缺点：正版 **JLINK** 价格昂贵，动辄上千，比我们的 **ISO** 还贵，着实坑爹。但是我朝攻城狮已经破解了 **JLINK** 里面的固件，生产出了具有特色社会主义的 **JLINK**，便宜又好用，真是壮哉我等屌丝啊。

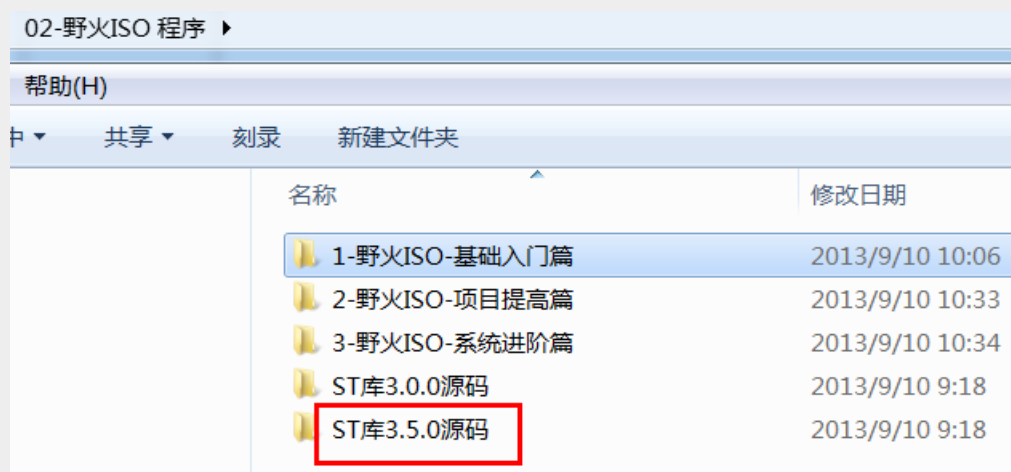
所以，建议大家在购买开发板的时候，一定要买一个我朝攻城狮开发的 **JLINK**。



## 2. 新建工程

### 2.1.1. 获取 ST 库源码

在新建工程模板之前，我们首先需要获取到 st 库的源码，源码可从 st 的官方网站下载到，也可在我们的光盘里面找到。

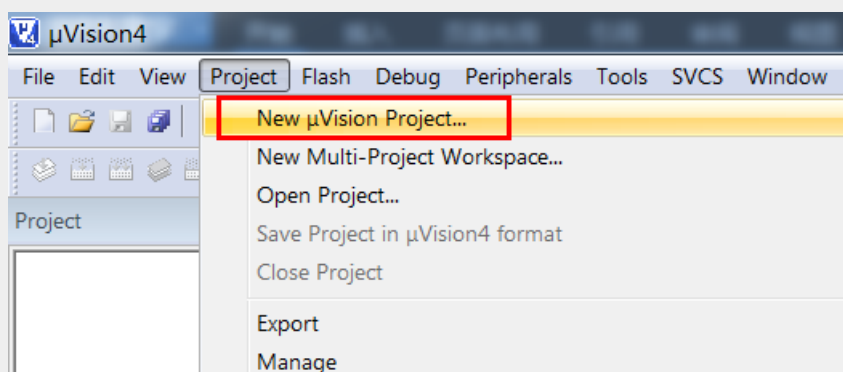


这里面有 V3.0.0 和 V3.5.0 版本的库，这两个库的版本区别很小，可以兼容。在这里我们以 V3.5.0 来新建我们的工程模板。

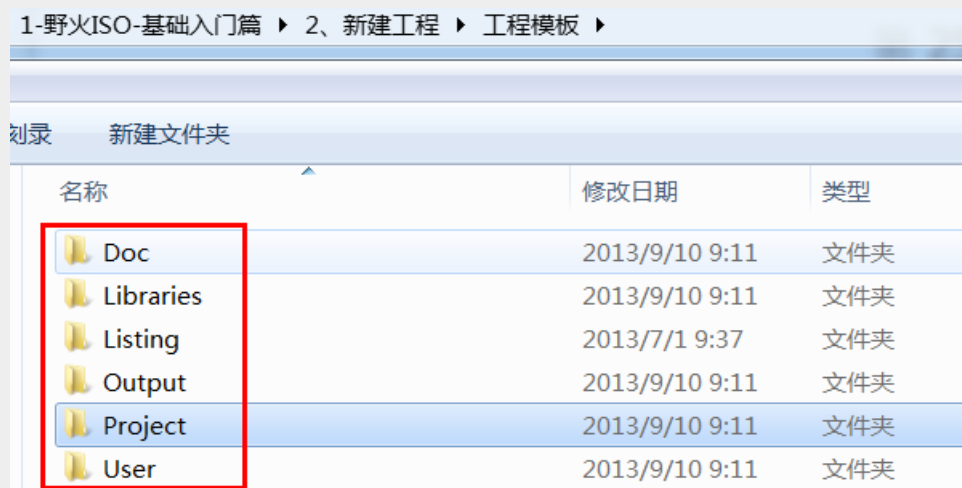
### 2.1.2. 开始新建工程

① 点击桌面 UVision4 图标，启动软件。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏 Project->Close Project 选项把它关掉。

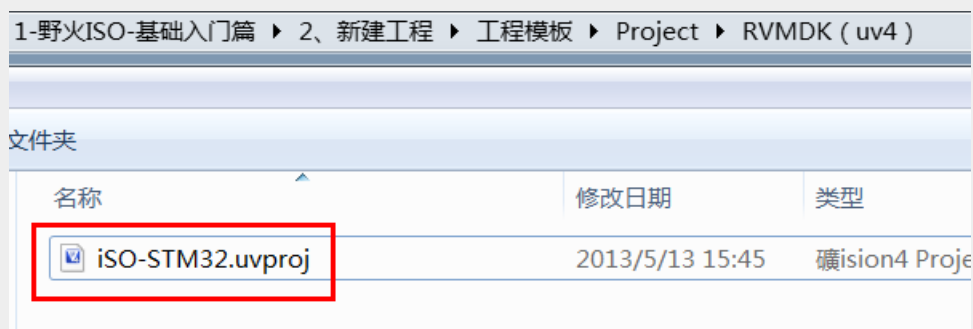
② 在工具栏 Project->New  $\mu$  Vision Project...新建我们的工程文件。



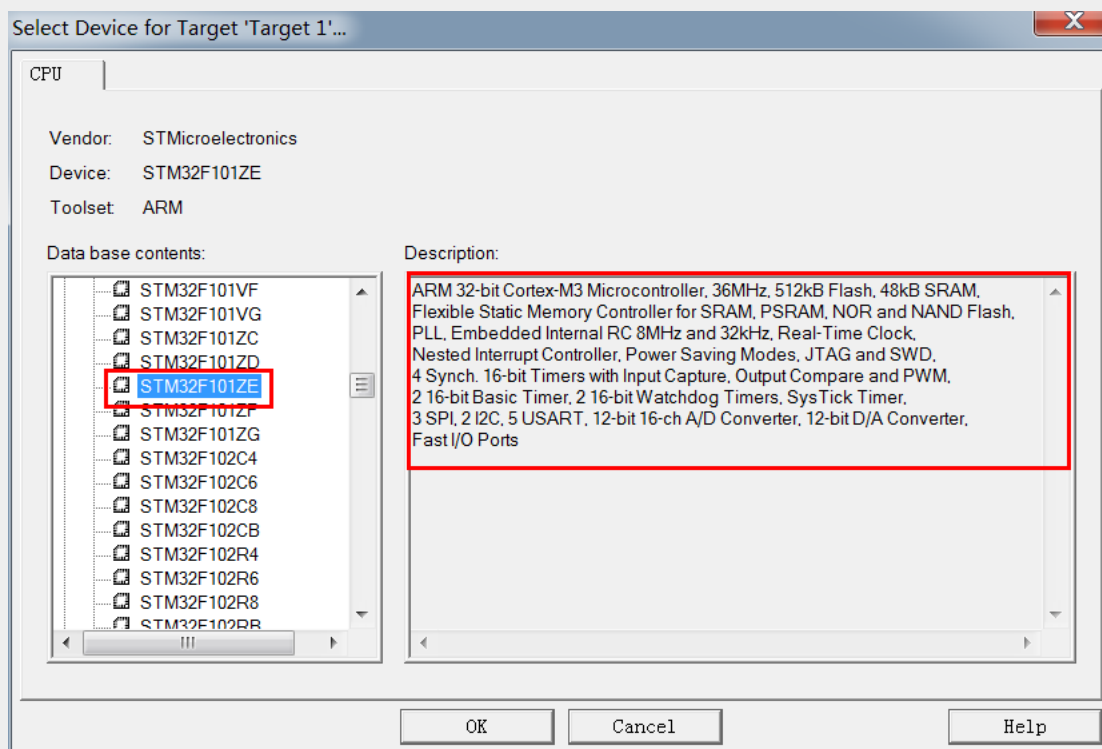
③ 我们将新建的工程文件保存在一个文件夹里面。首先我们建一个名为工程模板的文件夹，再在工程模板里面建几个文件夹（如下图所示）：Doc、Libraries、Listing、Output、Project、User。



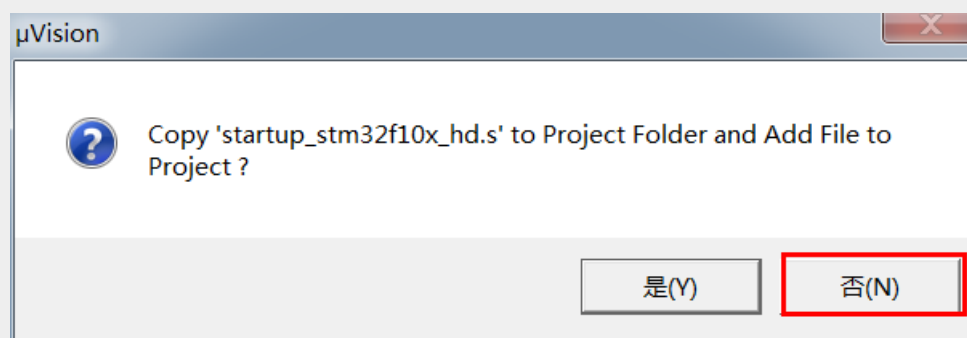
④ 在 Project 文件夹里面再建一个 RVMDK (uv4) 文件夹，这个文件夹就是我们用来存放我们的工程的，其中工程名字我在这里取为 ISO-STM32，大家可以根据自己的爱好取名字，不必跟我一样。



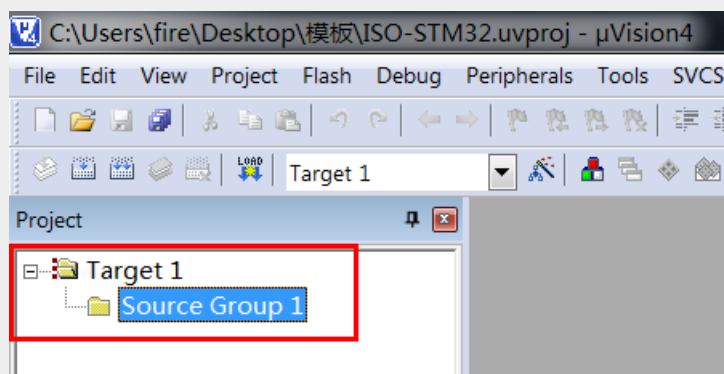
⑤ 接下来的窗口是让我们选择公司跟芯片的型号，我们用的芯片是 ST 公司的 STM32F103ZET6，有 64K SRAM, 512K Flash，属于高集成度的芯片。按如下选择即可。



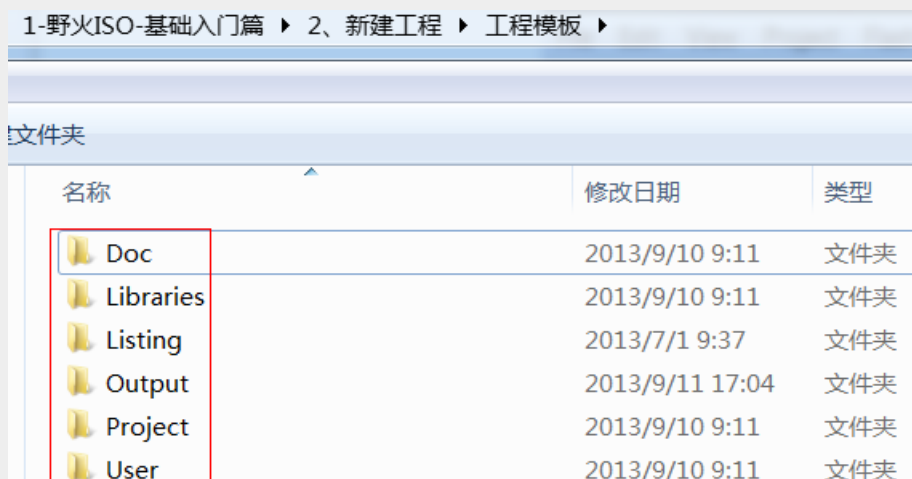
⑥ 接下来的窗口问我们是否需要拷贝 **STM32** 的启动代码到工程文件中，这份启动代码在 **M3** 系列中都是适用的，一般情况下我们都点击是，但我们这里用的是 **ST** 的库，库文件里面也自带了这一份启动代码，所以为了保持库的完整性，我们就不需要开发环境为我们自带的启动代码了，稍后我们自己手动添加，这里我们点击否。



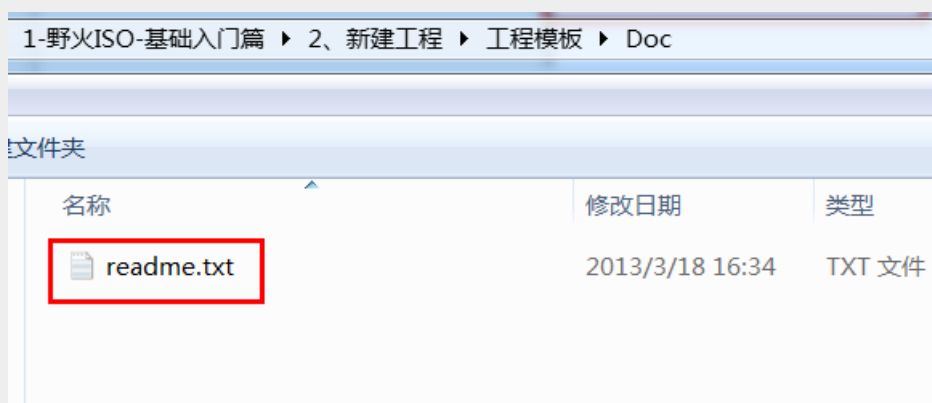
⑦ 此时我们的工程新建成功，如下图所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。



⑧ 在工程模板文件夹下，我们在刚开始建的文件夹下添加我们所需要的文件。



**Doc** 用来放一些说明文档，一般文件格式为 **XXX.TXT**，我一般习惯用 **readme.txt**。



**USER** 用来存放用户写的驱动文件。其中还包含了下面这四个文件：  
main.c、stm32f10x\_conf.h、stm32f10x\_it.c、stm32f10x\_it.h。

1-野火ISO-基础入门篇 ▶ 2、新建工程 ▶ 工程模板 ▶ User

文件夹

| 名称               | 修改日期             | 类型   |
|------------------|------------------|------|
| main.c           | 2013/5/13 15:45  | C 文件 |
| stm32f10x_conf.h | 2011/4/4 19:03   | H 文件 |
| stm32f10x_it.c   | 2012/10/26 11:44 | C 文件 |
| stm32f10x_it.h   | 2011/4/4 19:03   | H 文件 |

这四个文件来自于 ST 的库，我们需要拷贝过来即可，根据下面红色框中的路径即可找到这三个文件。

ST库3.5.0源码 ▶ STM32F10x\_StdPeriph\_Lib\_V3.5.0 ▶ Project ▶ STM32F10x\_StdPeriph\_Template ▶

文件夹

| 名称                 | 修改日期           | 类型   | 大小    |
|--------------------|----------------|------|-------|
| EWARM              | 2013/9/10 9:18 | 文件夹  |       |
| HiTOP              | 2013/9/10 9:18 | 文件夹  |       |
| MDK-ARM            | 2013/9/10 9:18 | 文件夹  |       |
| RIDE               | 2013/9/10 9:18 | 文件夹  |       |
| TrueSTUDIO         | 2013/9/10 9:18 | 文件夹  |       |
| main.c             | 2011/4/4 19:03 | C 文件 | 8 KB  |
| stm32f10x_conf.h   | 2011/4/4 19:03 | H 文件 | 4 KB  |
| stm32f10x_it.c     | 2011/4/4 19:03 | C 文件 | 5 KB  |
| stm32f10x_it.h     | 2011/4/4 19:03 | H 文件 | 3 KB  |
| system_stm32f10x.c | 2011/4/4 19:03 | C 文件 | 36 KB |

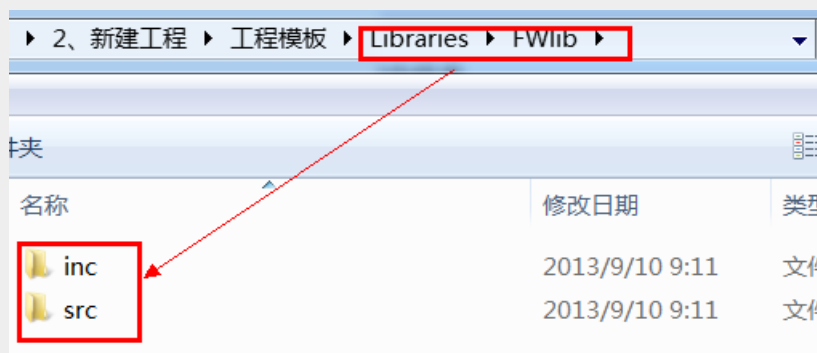
Libraries 用来放放 ST 库里面最核心的文件，里面包含了 FWlib 和 CMSIS 这两个文件夹。

1-野火ISO-基础入门篇 ▶ 2、新建工程 ▶ 工程模板 ▶ Libraries ▶

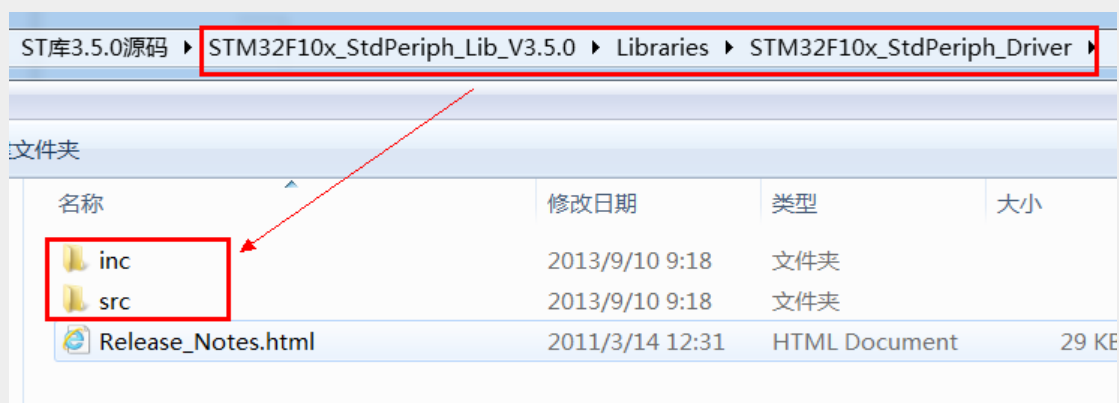
文件夹

| 名称    | 修改日期           | 类型  |
|-------|----------------|-----|
| CMSIS | 2013/9/10 9:11 | 文件夹 |
| FWlib | 2013/9/10 9:11 | 文件夹 |

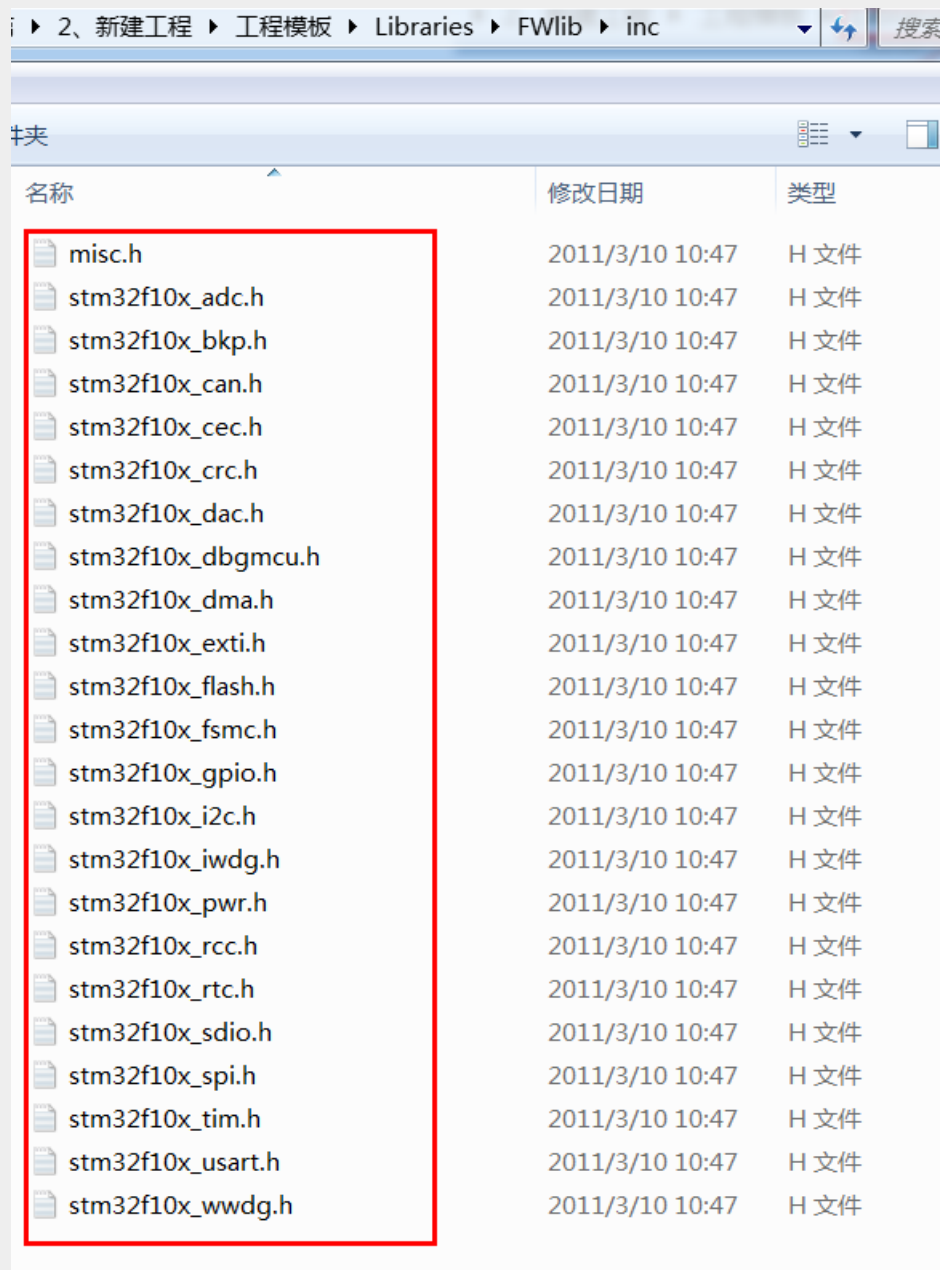
FWlib:用来存放 STM32 库里面的 inc 和 src 这两个文件，这两个文件包含了芯片上的所有驱动。



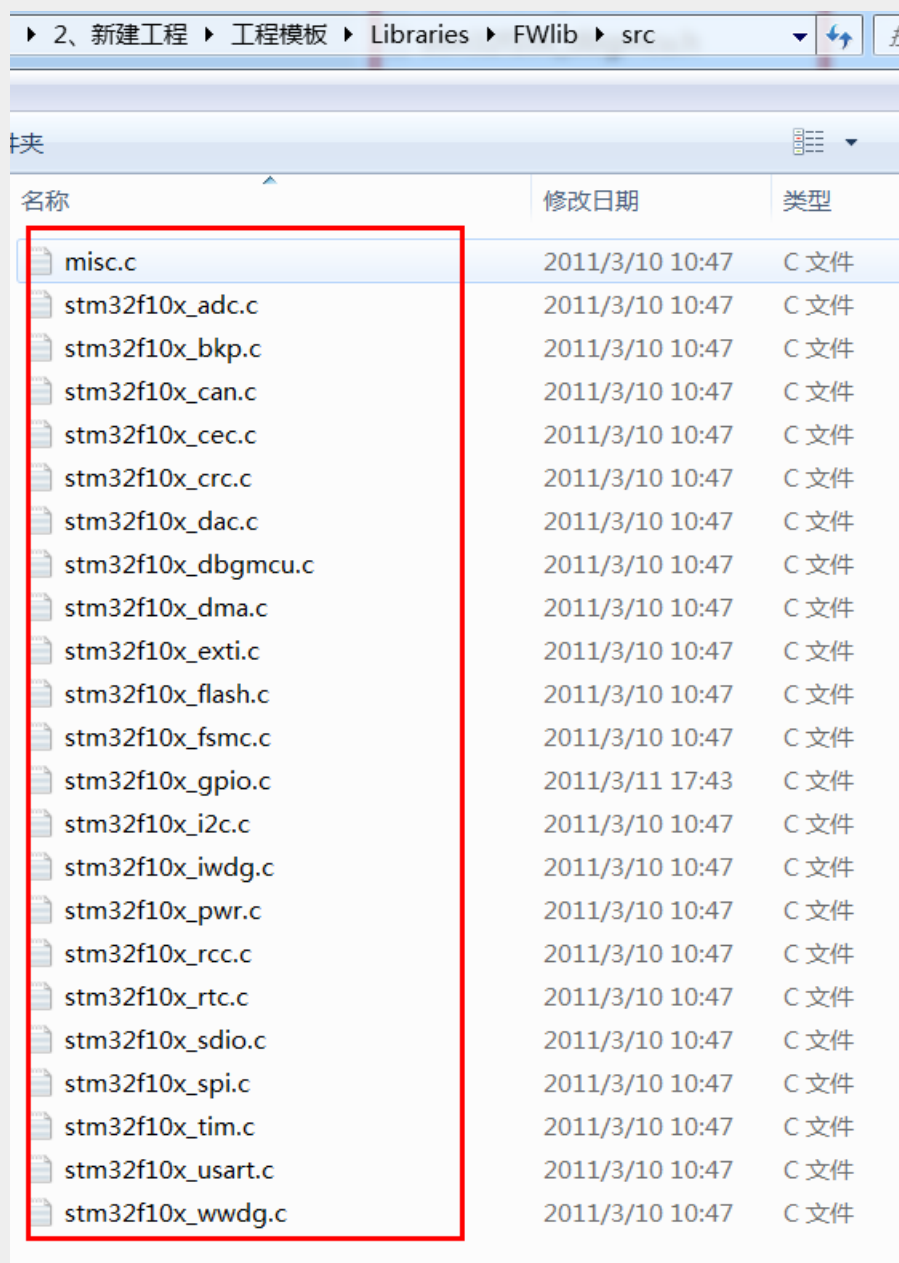
Inc 和 src 这两个文件夹也是直接从 ST 的库里面复制过来的。



inc 里面是 ST 片上资源的驱动的头文件，如要用到某个资源，则必须把相应的头文件包含进来。



src 里面是 ST 片上资源的驱动文件，这些驱动里面涉及了大量的 c 语言的知识，是我们学习库的重点。



CMSIS 用来存放库为我们自带的启动文件和一些 M3 系列通用的文件。  
CMSIS 里面存放的文件适合任何 M3 内核的单片机。CMSIS 的缩写为：  
Cortex Microcontroller Software Interface Standard，是 ARM Cortex 微控制器软件接口标准，是 ARM 公司为芯片厂商提供的一套通用的且独立于芯片厂商的处理器软件接口。



1-野火ISO-基础入门篇 ▶ 2、新建工程 ▶ 工程模板 ▶ Libraries ▶ CMSIS ▶

| 名称                 | 修改日期            | 类型   |
|--------------------|-----------------|------|
| startup            | 2013/9/10 9:11  | 文件夹  |
| core_cm3.c         | 2010/6/7 10:25  | C 文件 |
| core_cm3.h         | 2011/2/9 14:59  | H 文件 |
| stm32f10x.h        | 2011/3/10 10:51 | H 文件 |
| system_stm32f10x.c | 2011/3/10 10:51 | C 文件 |
| system_stm32f10x.h | 2011/3/10 10:51 | H 文件 |

1-野火ISO-基础入门篇 ▶ 2、新建工程 ▶ 工程模板 ▶ Libraries ▶ CMSIS ▶ startup

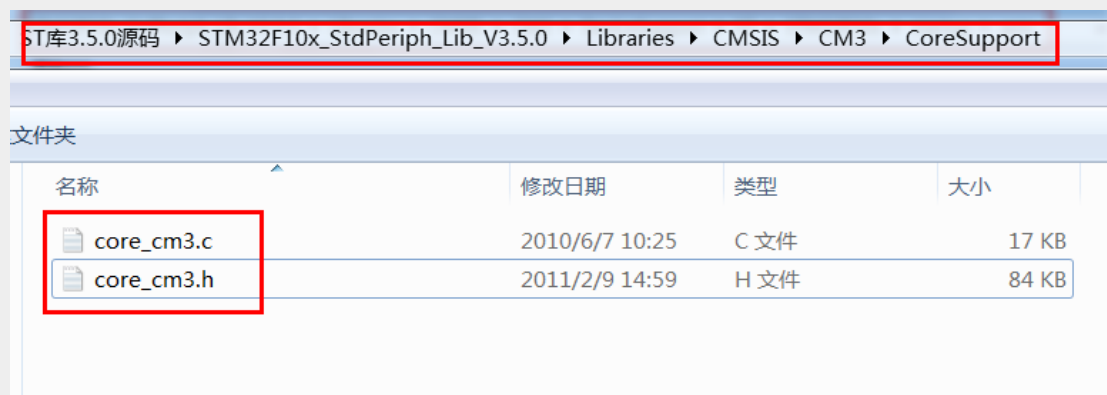
| 名称                        | 修改日期            | 类型   | 大小    |
|---------------------------|-----------------|------|-------|
| startup_stm32f10x_cl.s    | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_hd.s    | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_hd_vl.s | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_ld.s    | 2011/3/10 10:52 | S 文件 | 13 KB |
| startup_stm32f10x_ld_vl.s | 2011/3/10 10:52 | S 文件 | 14 KB |
| startup_stm32f10x_md.s    | 2011/3/10 10:52 | S 文件 | 13 KB |
| startup_stm32f10x_md_vl.s | 2011/3/10 10:51 | S 文件 | 14 KB |
| startup_stm32f10x_xl.s    | 2011/3/10 10:51 | S 文件 | 16 KB |

其中 startup 文件夹下的文件是从 ST 库里面复制过来的，路径如下红框中所示。

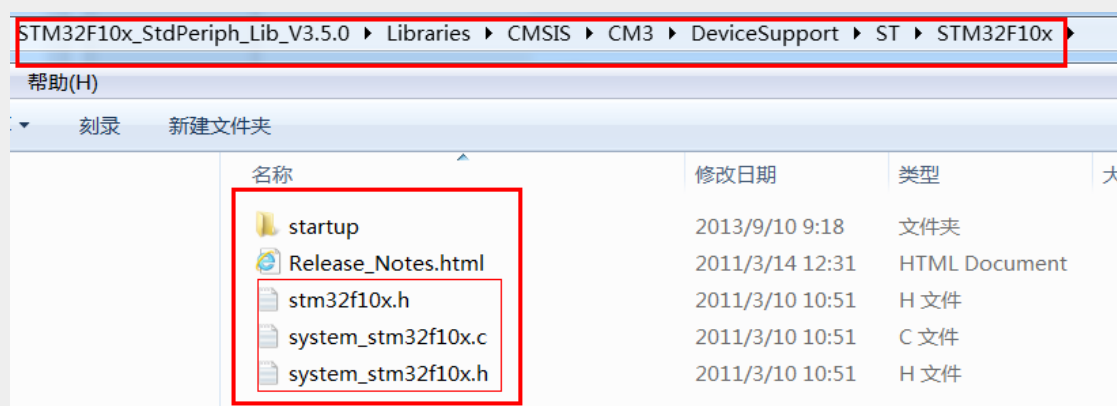
STM32F10x\_StdPeriph\_Lib\_V3.5.0 ▶ Libraries ▶ CMSIS ▶ CM3 ▶ DeviceSupport ▶ ST ▶ STM32F10x ▶ startup ▶ arm

| 名称                        | 修改日期            | 类型   | 大小    |
|---------------------------|-----------------|------|-------|
| startup_stm32f10x_cl.s    | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_hd.s    | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_hd_vl.s | 2011/3/10 10:52 | S 文件 | 16 KB |
| startup_stm32f10x_ld.s    | 2011/3/10 10:52 | S 文件 | 13 KB |
| startup_stm32f10x_ld_vl.s | 2011/3/10 10:52 | S 文件 | 14 KB |
| startup_stm32f10x_md.s    | 2011/3/10 10:52 | S 文件 | 13 KB |
| startup_stm32f10x_md_vl.s | 2011/3/10 10:51 | S 文件 | 14 KB |
| startup_stm32f10x_xl.s    | 2011/3/10 10:51 | S 文件 | 16 KB |

core\_cm3.c 和 core\_cm3.h 这两个文件也可从 ST 库里面复制过来，路径如下：

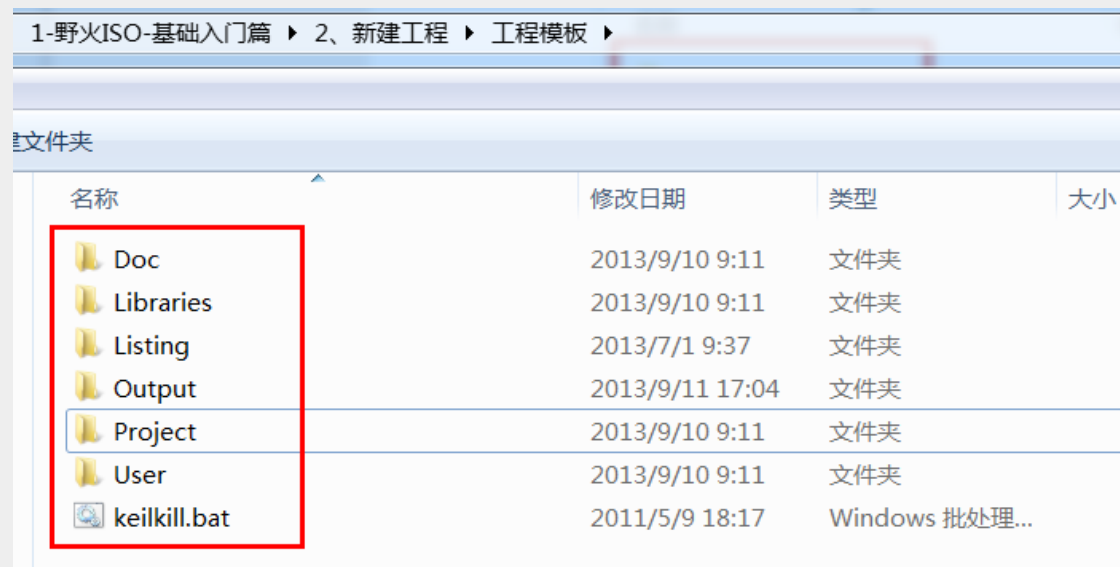


stm32f10x.h、system\_stm32f10x.c、system\_stm32f10x.h 这三个文件也可从 ST 库里面复制过来，路径如下：



Uotput 用来保存软件编译后输出的文件，当我们编译出的.axf 或者.hex 文件就放在这个文件夹，.axf 是 JLINK 下载程序用到的文件，.hex 则为串口下载用到的文件，现在暂时为空。

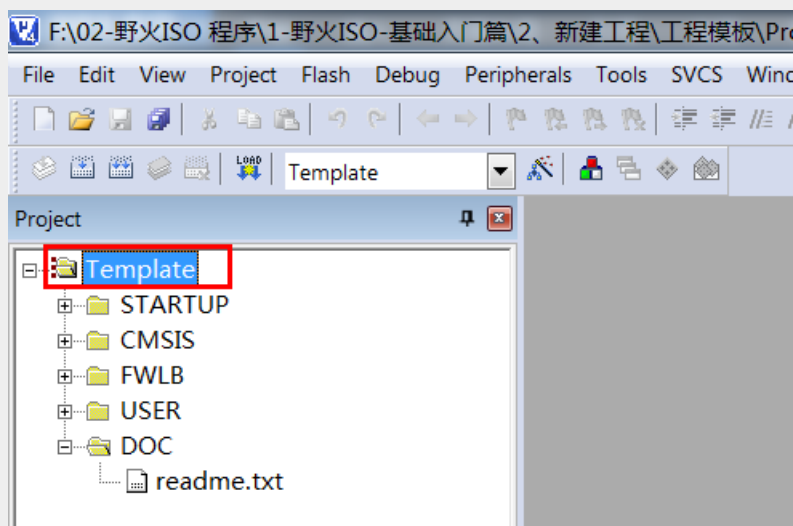
Listing 用来存放一些编译过程中产生的文件，暂时为空。



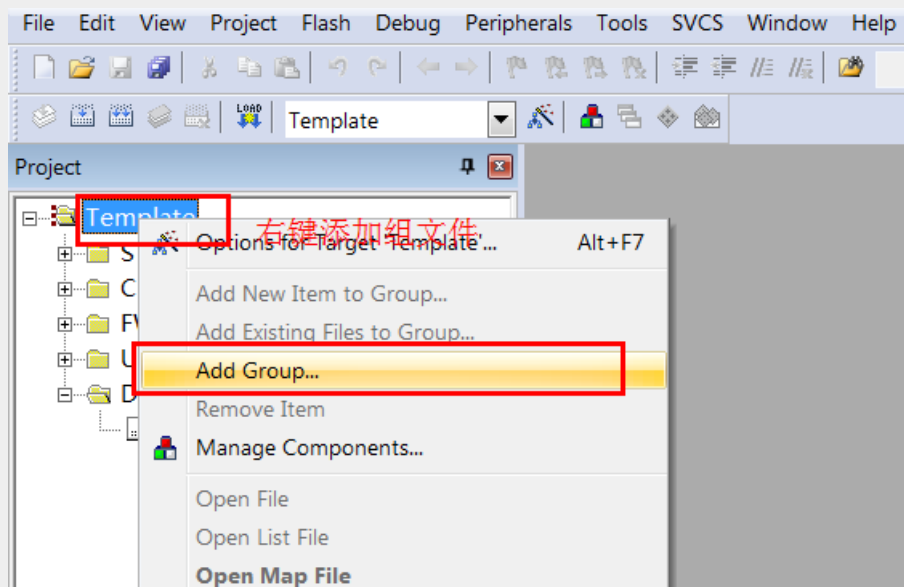
现在我们就已经基本完成了往模板里面添加库文件的工作。

⑨ 回到我们刚刚新建的工程界面，把刚刚那些文件夹里面的内容添加进工程，这样我们在 MDK 界面里面就可以管理那些文件了。

1. 把 Target 的名称改为 Template，如果我们建的工程是 LED 的就改为 LED，其实不改也可以，改了只是为了见名知义。

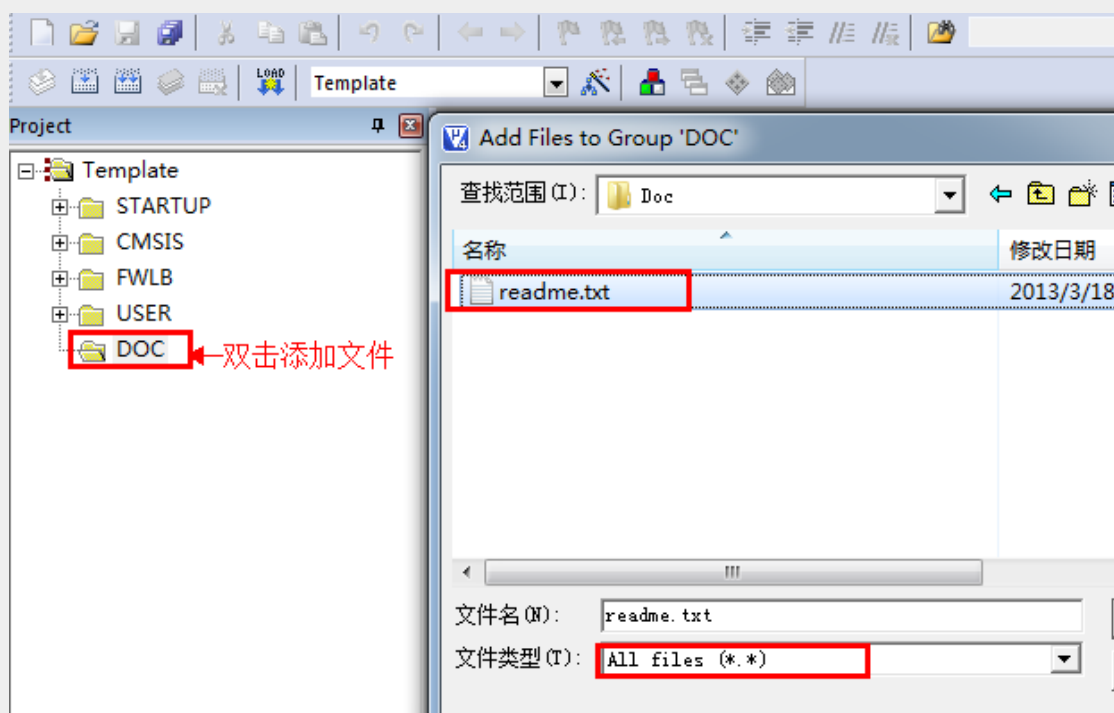


2. 往工程里面添加 5 个组文件夹，并命名为 STARTUP、CMSIS 、FWLIB 、USER 、DOC 。



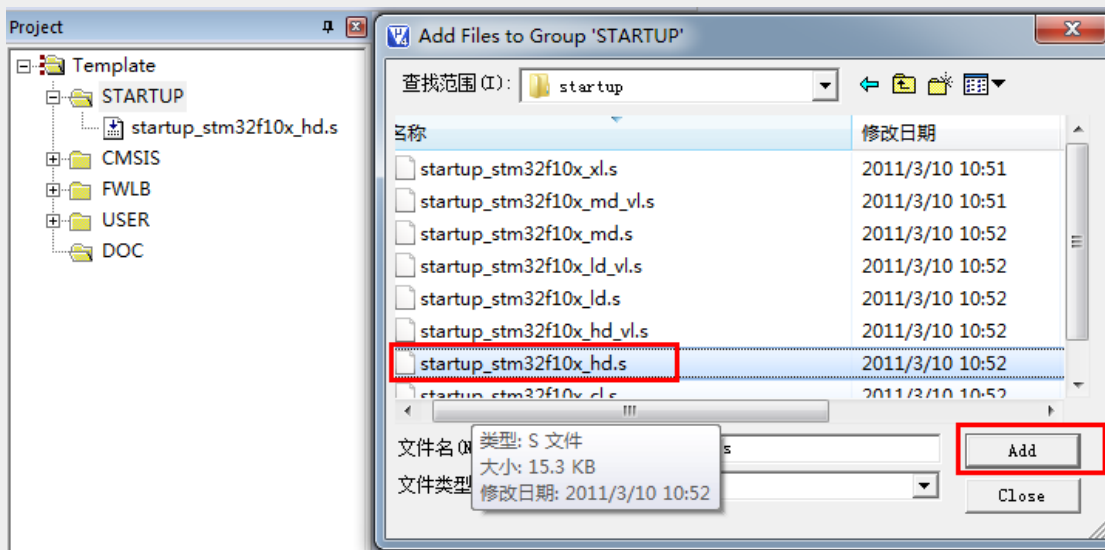
从名字就可以看得出我们是用它来放我们的启动代码的，USER 用来存放用户自定义的应用程序，FWlib 用来存放库文件，CMSIS 用来存放 M3 系列单片机通用的文件。

3. 接下来我们往这些新建的组中添加文件，双击哪个组就可以往哪个组里面添加文件，如果该组里面已经有文件的话，双击则把组里面的文件都显示出来，然后再双击该组的话，则可以继续添加文件。

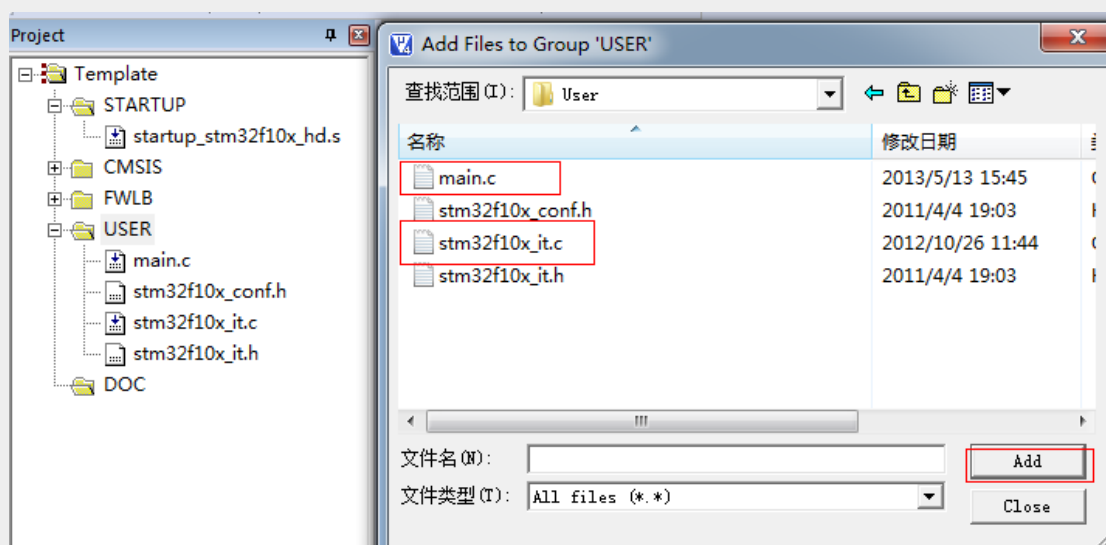


在对话框里面的文件类型里面我们选择 ALL files (\*.\*)，不然有些文件会显示不出来。

a. 在 STARTCOKE 里面添加 startup\_stm32f10x\_hd.s。



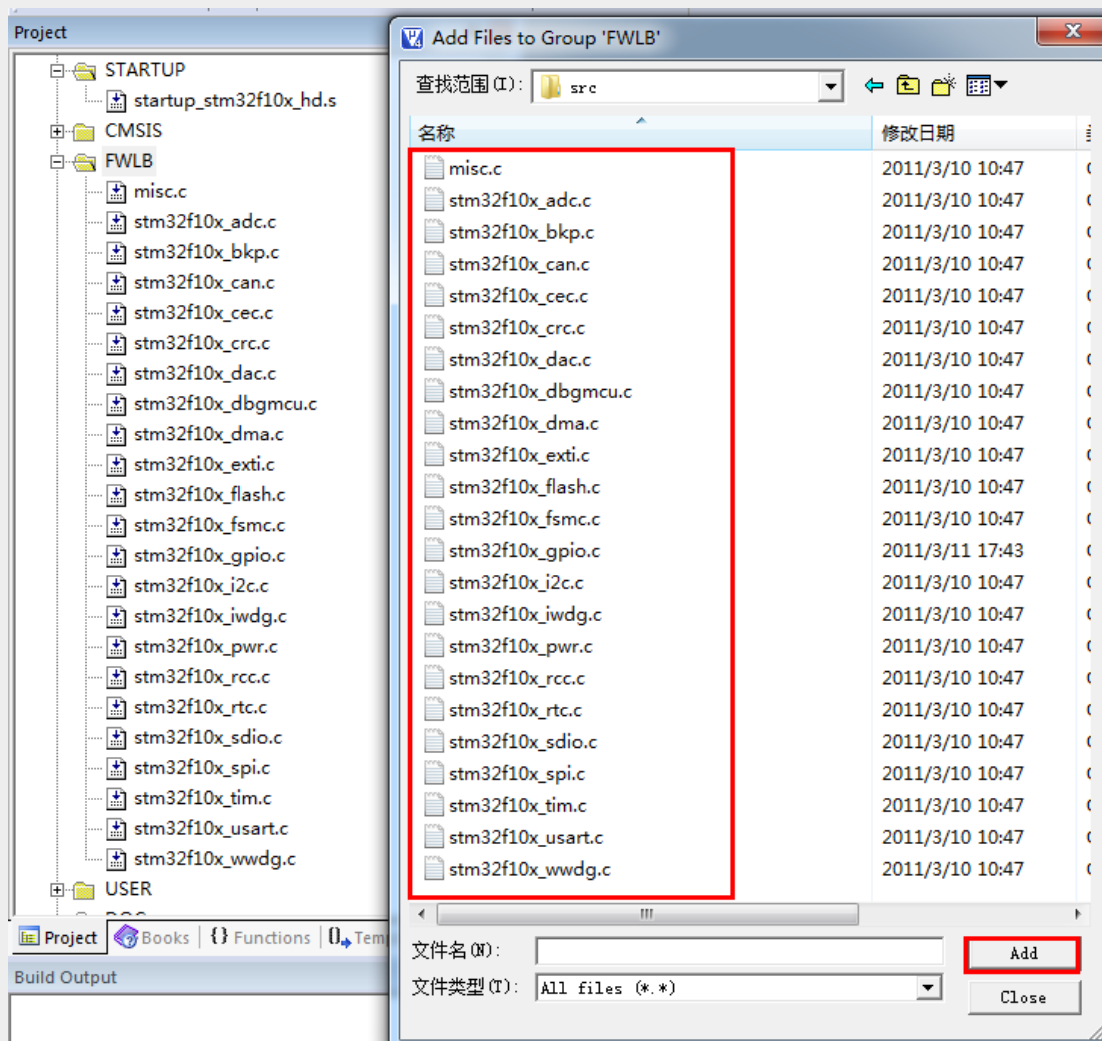
b. 在 USER 组里面添加 main.c 和 stm32f10x\_it.c 这两个文件。



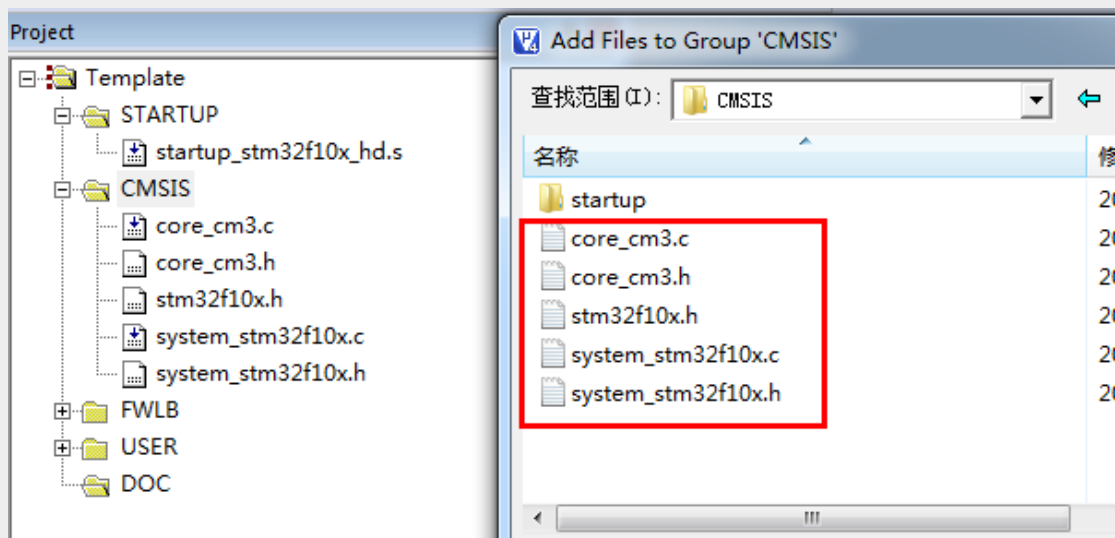
其中 stm32f10x\_conf.h 和 stm32f10x\_it.h 这两个头文件不需要添加，即使你添加也添加不进来，因为.h 文件不是通过这种方式进入到工程里面的。头文件都包含在了.c 文件中，当我们编译工程的时候.h 文件就自然包含进来了。

c. 在 FWlib 组里面添加 src 里面的全部驱动文件，当然，src 里面的驱动文件也可以需要哪个就添加哪个。这里将它们全部添加进去是为了后续开发的

方便，况且我们可以通过配置 `stm32f10x_conf.h` 这个头文件来选择性添加，只有在 `stm32f10x_conf.h` 文件中配置的文件才会被编译。

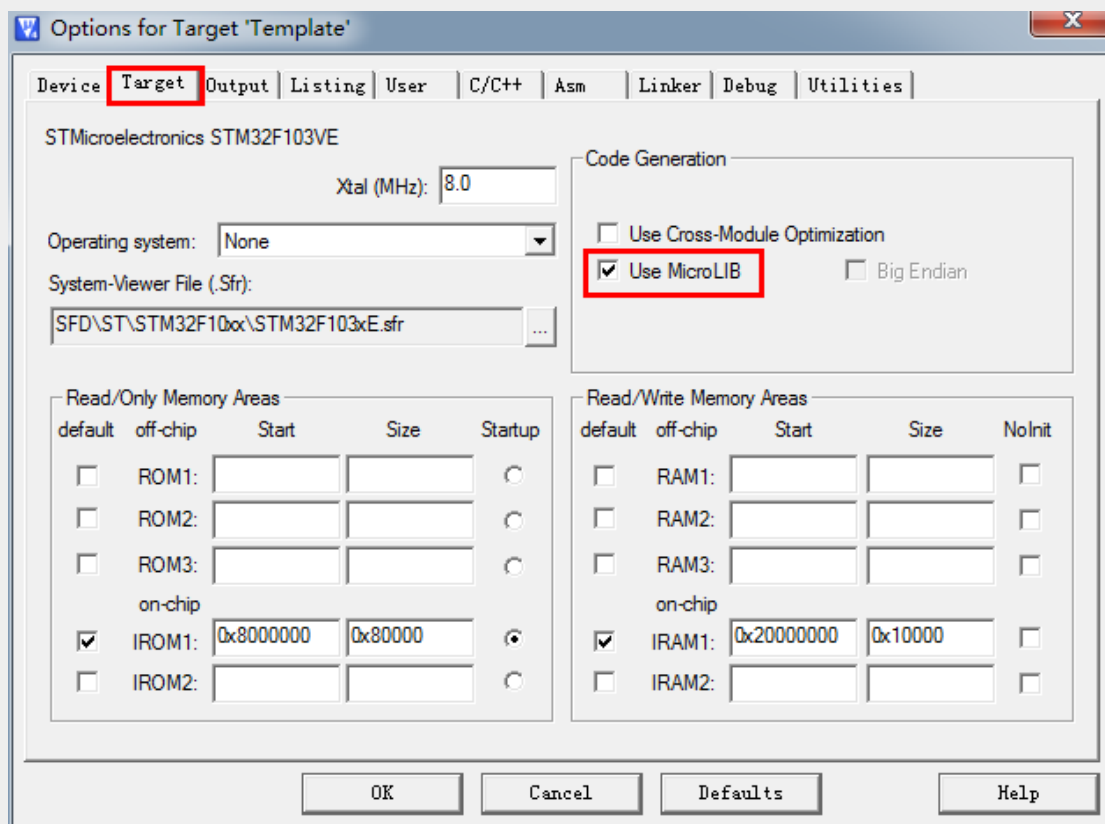


- d. 在 CMSIS 里面添加以下文件。注意，这些组里面添加的都是汇编文件跟 C 文件，头文件是不需要添加的，也是添加不了的。

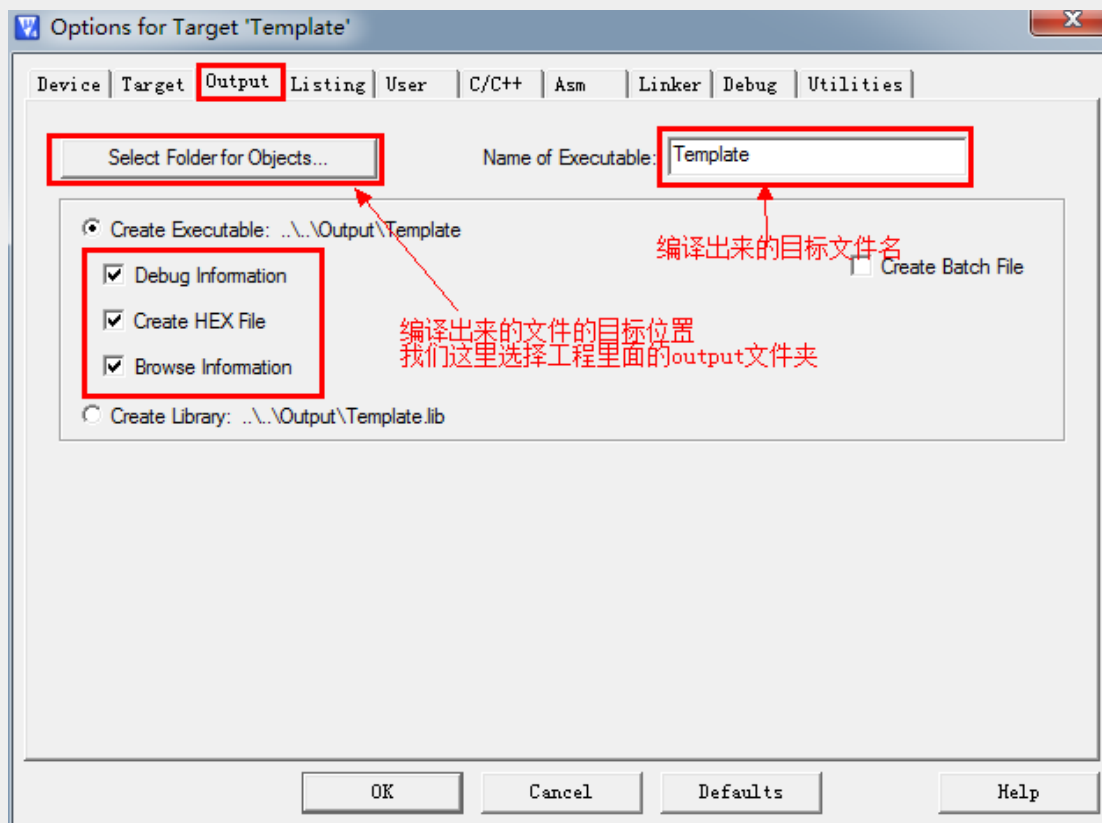


至此，我们的工程已经基本建好，但是还有一些工作需要完成。下面来配置一下 MDK 的配置选项，点击工具栏中的魔术棒按钮。

① 在 Target 这个选项卡中，把微库的勾打上，这样是为了后面的串口例程可以使用 printf 这个函数。

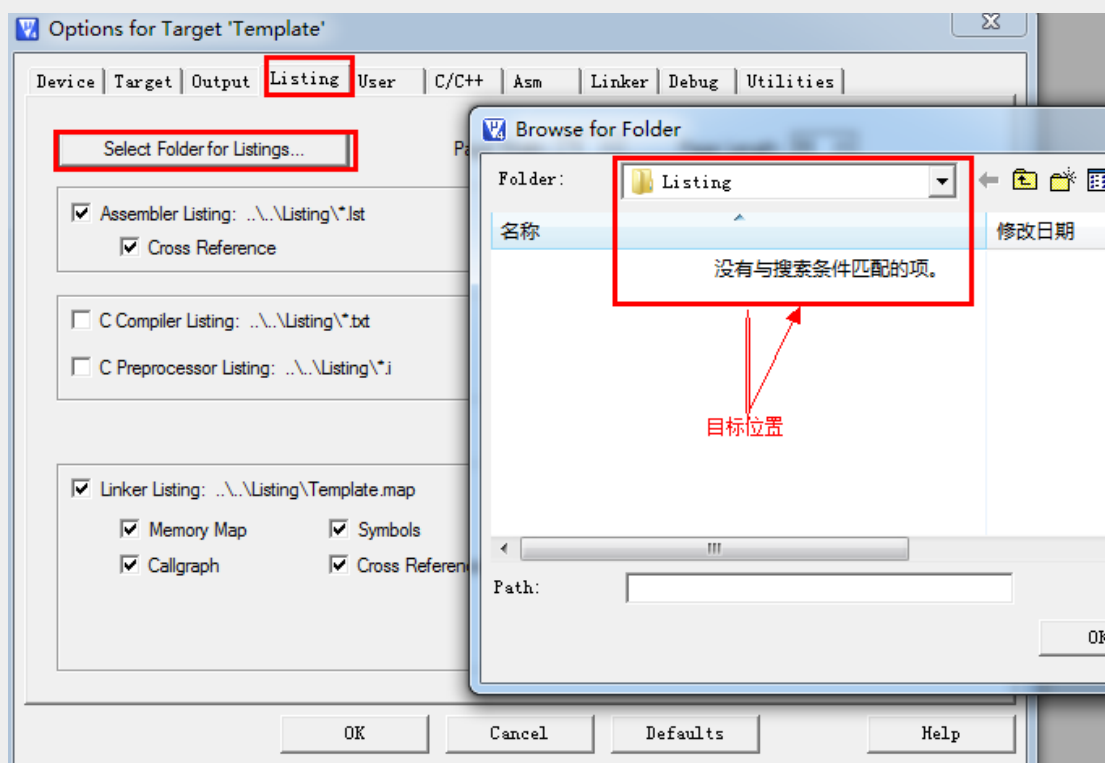


③ Output 选项卡如下设置，点击 Select Folder for Objects... 设置编译后输出文件保存的位置。同时把 Debug Information 、 Create HEX File 和 Browse information 这三个选项框也选上。



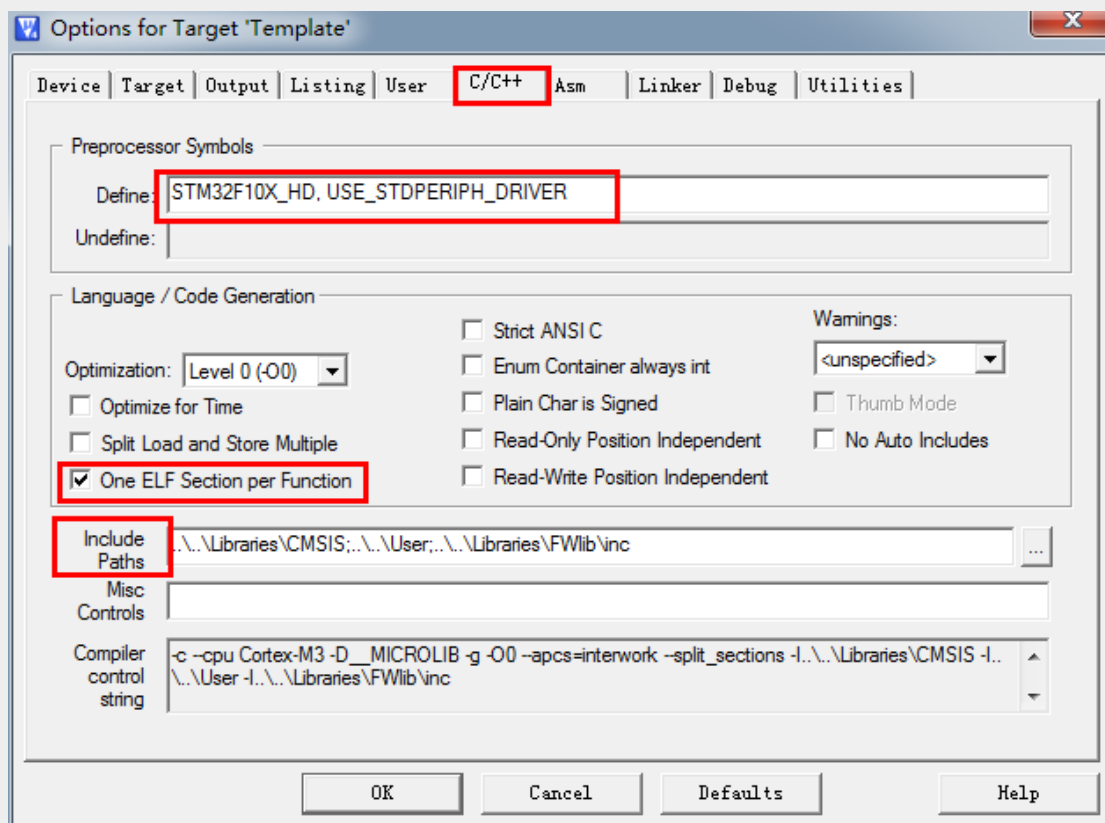
④ 在 Listing 这个选项卡中，点击 Select Folder listings...定位到模板中的 Listing 文件夹。



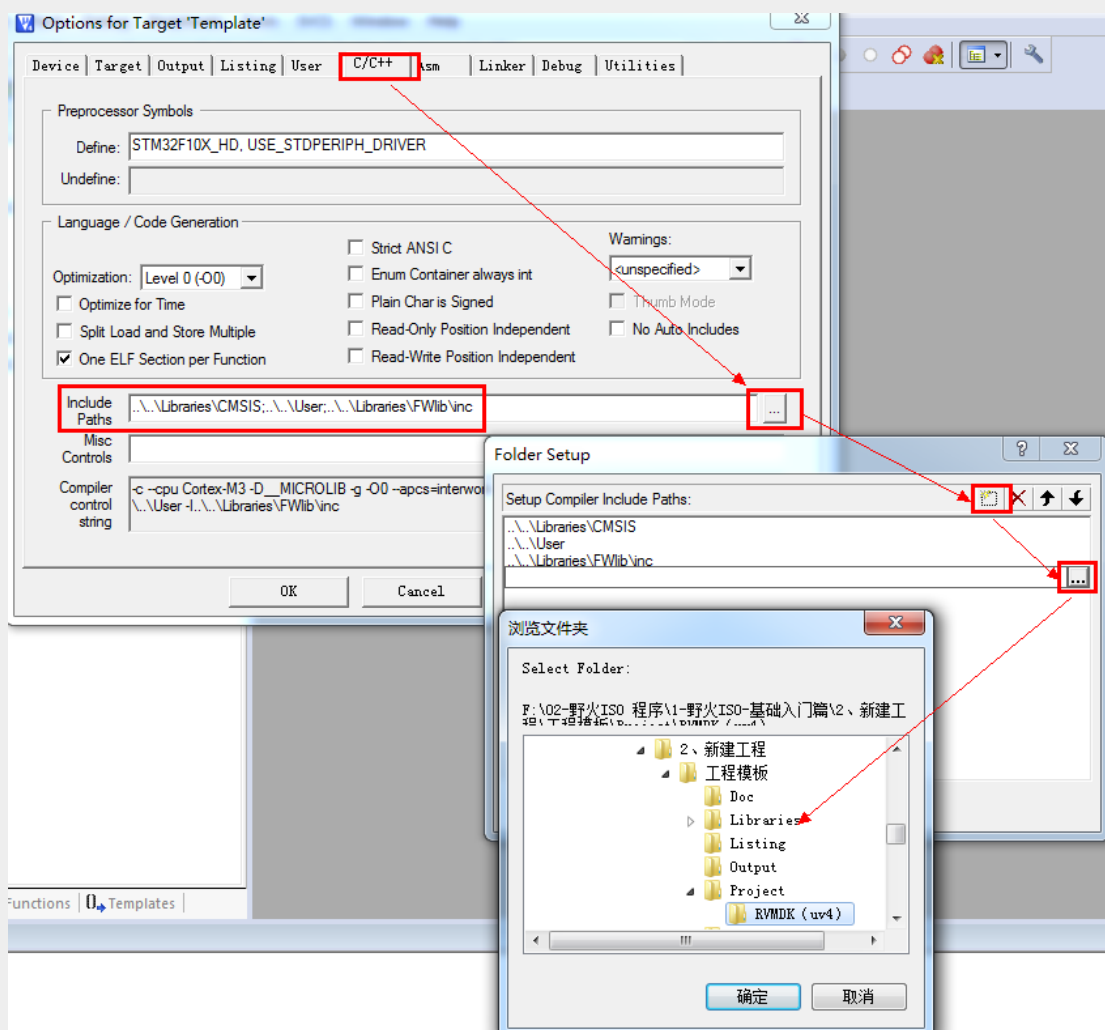


⑤ 在 C/C++ 这个选项卡上需要 设置的比较多。

在 Define 里面输入添加 STM32F10X\_HD, USE\_STDPERIPH\_DRIVER 这两个宏。添加 USE\_STDPERIPH\_DRIVER 是为了屏蔽编译器的默认搜索路径，转而使用我们添加到工程中的 ST 的库，添加 STM32F10X\_HD 是因为我们用的芯片是大容量的，添加了 STM32F10X\_HD 这个宏之后，库文件里面为大容量定义的寄存器我们就可以用了。芯片是小或中容量的时候宏要换成 STM32F10X\_LD 或者 STM32F10X\_MD。其实不管是什么容量的，我们只要添加上 STM32F10X\_HD 这个宏即可，当你用小或者中容量的芯片时，那些为大容量定义的寄存器我不去访问就是了，反正也访问不了。

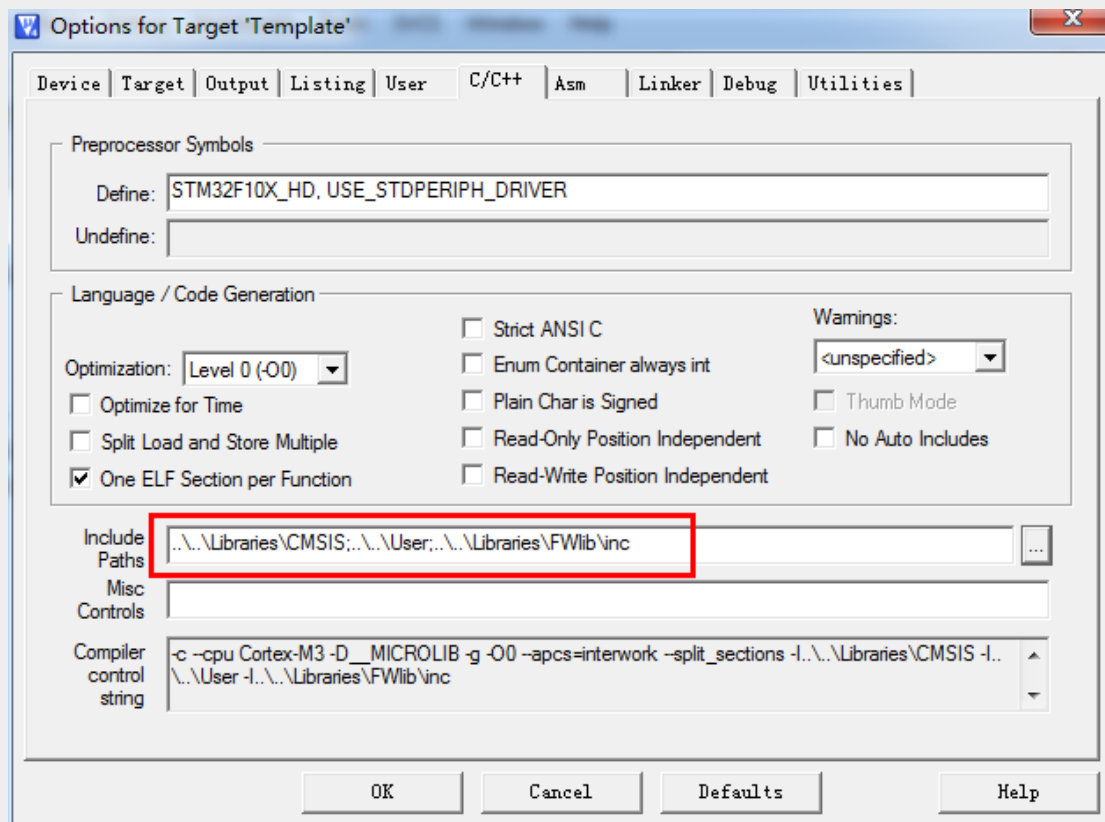


在 Include Paths 栏里添加库文件的搜索路径，这样就可以屏蔽掉默认搜索路径。



但当编译器在我们指定的路径下 搜索不到的话还是会回到标准目录去搜索，就像有些 ANSI C 的库文件，如 `stdin.h`、`stdio.h`。

库文件路径修改成功之后如下所示：



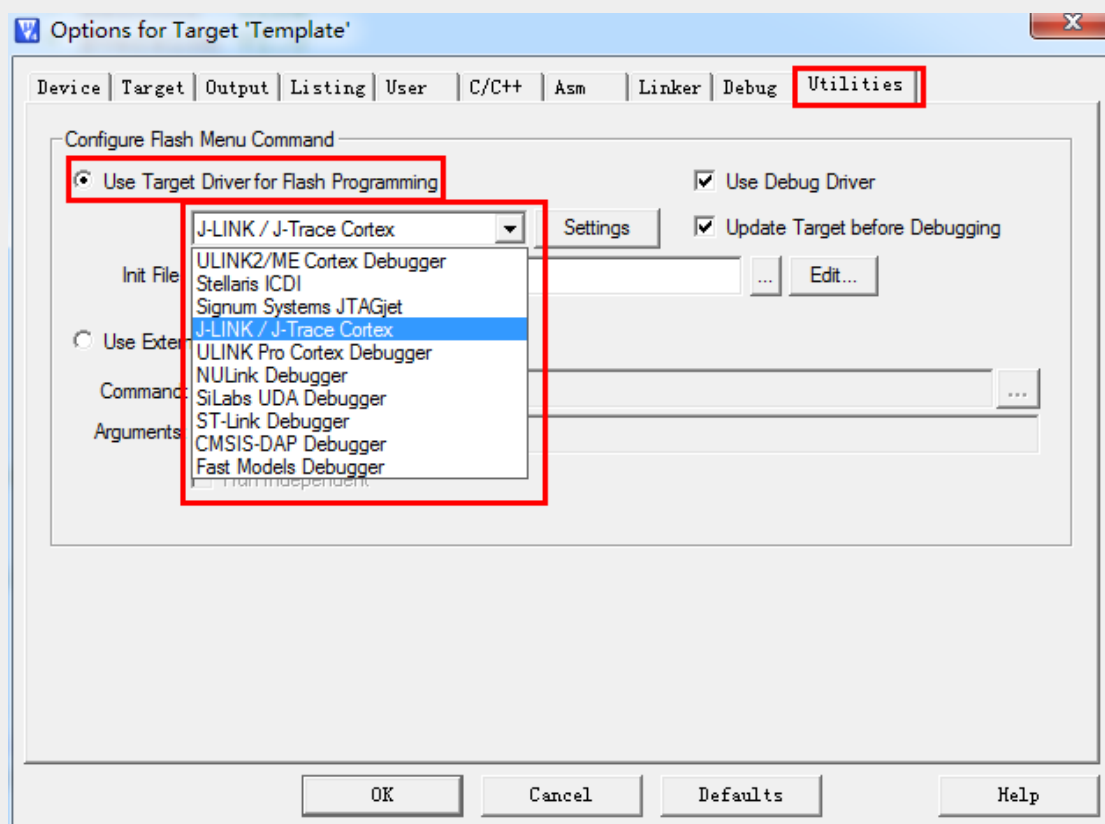
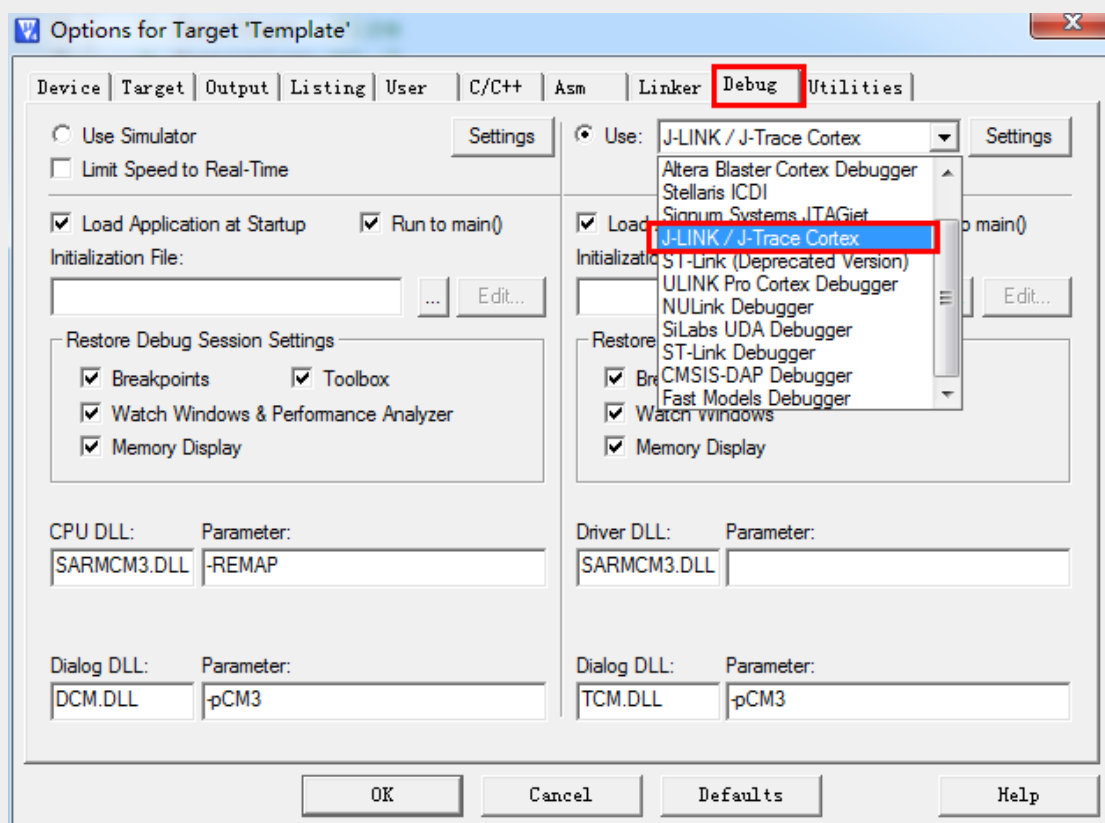
⑥ 修改 main.c 文件。因为刚刚我们的 main.c 文件是从官方库里面复制过来的，里面有许多东西我们是不需要的，为了简化 main.c 文件，我们将修改如下。

- 至此，我们的工程模板就建成了。学会新建工程，是学习 stm32 的第一步。

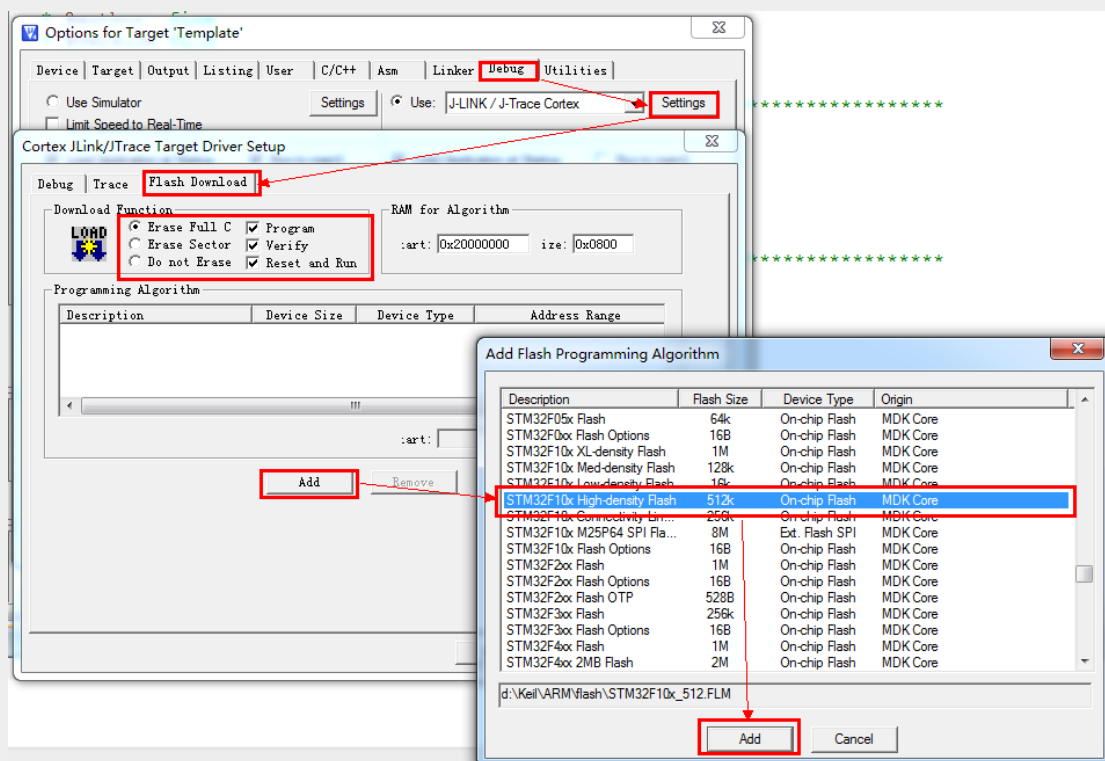
### 2.1.3. 硬件调试配置

这个工程默认的是软件仿真，如果开发板要用 J-LINK 调试的话，还需要在开发环境中做如下修改。实际上，我们开发程序的时候 90% 都是在硬件上调试的。

具体配置如下图



在选项卡 Debug\Setting\Flash download 中我们设置成如下：



到了这里就算是大功告成了。如果在新建工程中遇到什么问题，先不要急，可先参考野火提供的已经新建好的工程模板。

### 3. 初识 STM32 库

本章通过简单介绍 STM32 库的各个文件及其关系，让读者建立 STM32 库的概念，看完后对库有个总体印象即可，在后期实际开发时接触了具体的库时，再回头看看这一章，相信你对 STM32 库又会有一个更深刻的认识。

#### 3.1. STM32 神器之库开发

##### 3.1.1. 什么是 STM32 库？

在 51 单片机的程序开发中，我们直接配置 51 单片机的寄存器，控制芯片的工作方式，如中断，定时器等。配置的时候，我们常常要查阅寄存器表，看到用到哪些配置位，为了配置某功能，该置 1 还是置 0。这些都是很琐碎的、机械的工作，因为 51 单片机的软件相对来说较简单，而且资源很有限，所以可以直接配置寄存器的方式来开发。

STM32 库是由 ST 公司针对 STM32 提供的函数接口，即 **API (Application Program Interface)**，开发者可调用这些函数接口来配置 STM32 的寄存器，使开发人员得以**脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。**

当我们调用库的 API 的时候可以不用挖空心思去了解库底层的寄存器操作，就像当年我们学习 C 语言的时候，用 printf() 函数时只是学习它的使用格式，并没有去研究它的源码实现，如非必要，可以说是老死不相往来。

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

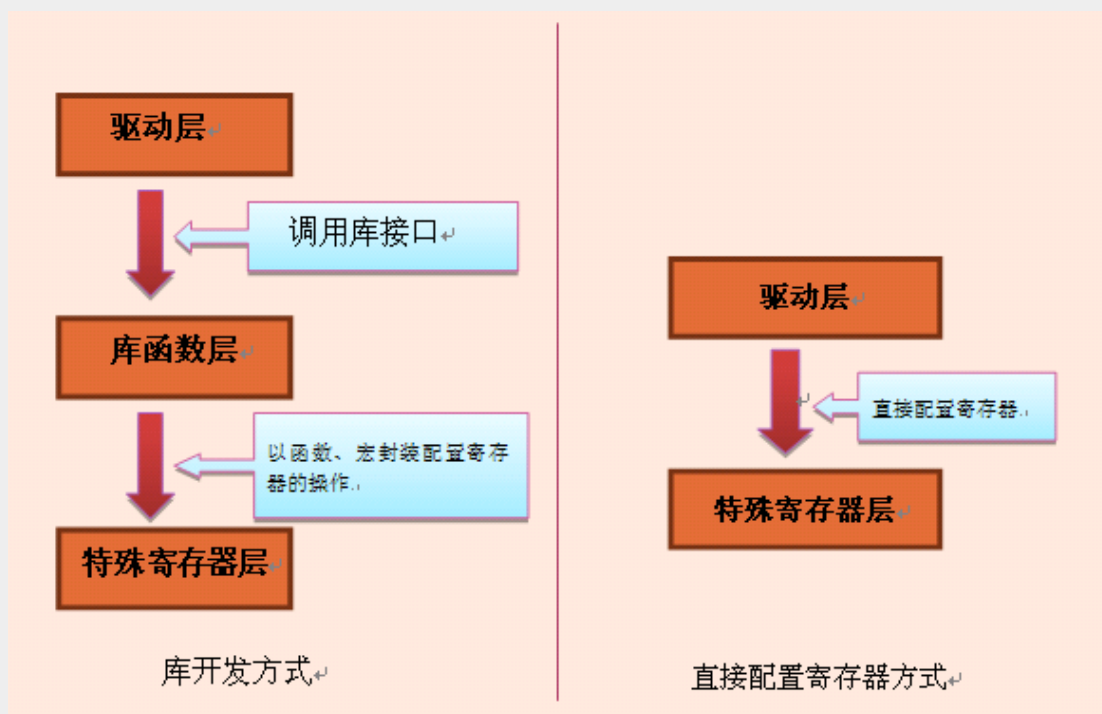


图 4-1

### 3.1.2. 为什么采用库来开发？

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

- 1) 开发速度慢
- 2) 程序可读性差

这两个缺陷直接影响了开发效率，程序维护成本，交流成本。库开发方式则正好弥补了这两个缺陷。

而坚持采用直接配置寄存器的方式开发的程序员，会列举以下原因：

- 1) 更直观
- 2) 程序运行占用资源少

初学 STM32 的读者，普遍因为第一个原因而选择以直接配置寄存器的方法来学习。认为这种方法直观，能够了解到是配置了哪些寄存器，怎样配置寄存器。事实上，库函数的底层实现恰恰是直接配置寄存器方式的最佳例子，想深入了解芯片是如何工作的话，只要追踪到库的最底层实现就能理解，相信你会



为它严谨、优美的实现方式而陶醉。要想修炼 C 语言，就从 ST 的库开始吧。

这将在第 4 章进行详细分析。

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

对于库开发与直接配置寄存器的方式，在 STM32 刚推出时引起程序员的激烈争论，但是，随着 ST 库的完善与大家对库的了解，更多的程序员选择了库开发。

本书采用 STM32 的库进行讲解，既介绍如何使用库接口，也讲解库接口的实现方式。使读者既能利用库进行快速开发，也能深入了解 STM32 的工作原理。

为进一步解答读者为什么使用库开发，请读者先思考一下为什么会采用 c 语言开发软件而不是采用汇编。相比之下，可以发现调用库接口开发与直接配置寄存器开发的关系，犹如 c 语言与汇编的关系。见表 4-1

| 特点              | 汇编语言                       | 直接配置寄存器方式                                   |
|-----------------|----------------------------|---------------------------------------------|
| 更接近机器思维<br>(直观) | 汇编指令为机器码的助记符，能直接了解 cpu 的操作 | 直接针对寄存器的某些位进行置 1 或清 0 操作，能清晰地看到驱动代码使用了什么寄存器 |
| 运行效率            | 代码为 cpu 直接执行的指令，与编译器优化无关   | 没有库函数层，省去代码为分层而消耗的资源                        |

和表 4-2。

据某无从考证的 IT 大师说过，“一切计算机科学的问题都可以用分层来解决。”从汇编到 c，从直接配置寄存器到使用库，从裸机到系统，从操作系统到应用层软件，无不体现着这样的分层思想。开发的软件多了，跨越的软件层次多了，会深刻地认同他这句话，分层思想在软件开发上体现得淋漓尽致，分

层使得问题变得更简单，使得能够屏蔽下层实现方式的差异，使得软件开发变成简单的调用函数接口，而不用管它的实现，大大提高效率。

库就是建立了一个新的软件抽象层，库的优点，其实就是分层的优点，库的缺点，也是软件分层带来的缺点，而对于 STM32 这样高性能的芯片，我想我们会愿意承受分层带来的缺点的。

| 特点              | C 语言                     | 库开发方式                                            |
|-----------------|--------------------------|--------------------------------------------------|
| 更接近人的思维<br>(易读) | 程序控制语句结构化。以函数作为程序单位便于模块化 | 1)用结构体封装寄存器参数<br>2)用宏表示参数，意义明确<br>3)用函数封装对寄存器的操作 |
| 移植性好            | 程序基本上不做修改就可应用于各种计算机上     | 代码的易读性使驱动代码的修改变得非常方便                             |

表 4-1

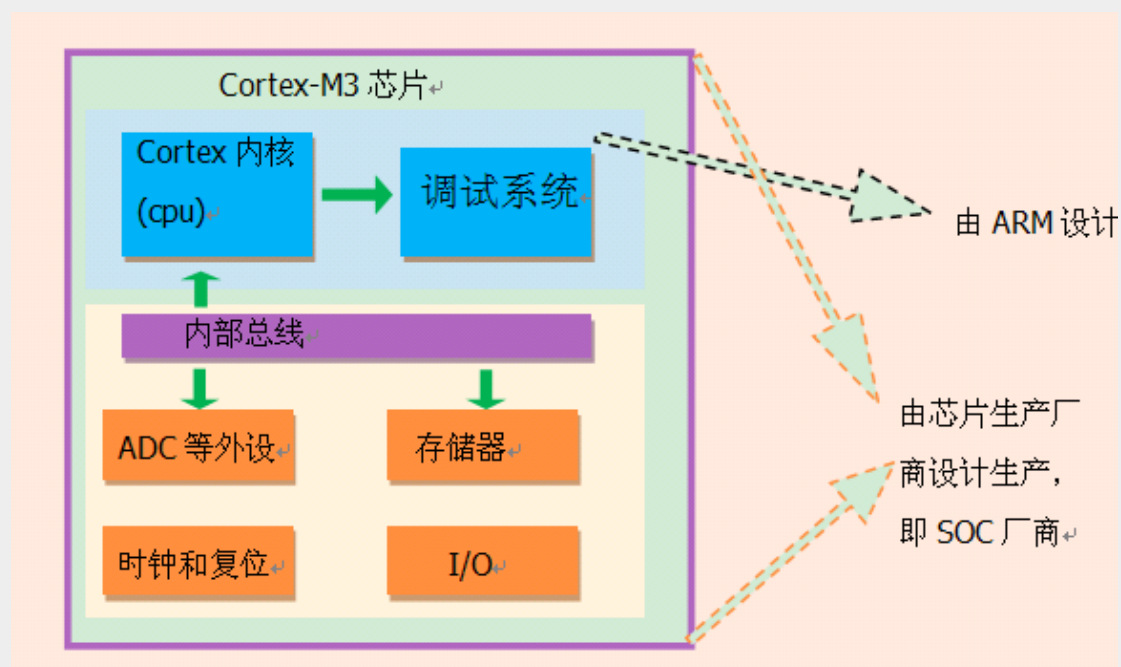
| 特点              | 汇编语言                       | 直接配置寄存器方式                                   |
|-----------------|----------------------------|---------------------------------------------|
| 更接近机器思维<br>(直观) | 汇编指令为机器码的助记符，能直接了解 cpu 的操作 | 直接针对寄存器的某些位进行置 1 或清 0 操作，能清晰地看到驱动代码使用了什么寄存器 |
| 运行效率            | 代码为 cpu 直接执行的指令，与编译器优化无关   | 没有库函数层，省去代码为分层而消耗的资源                        |

表 4-2

### 3.2. STM32 结构及库层次关系

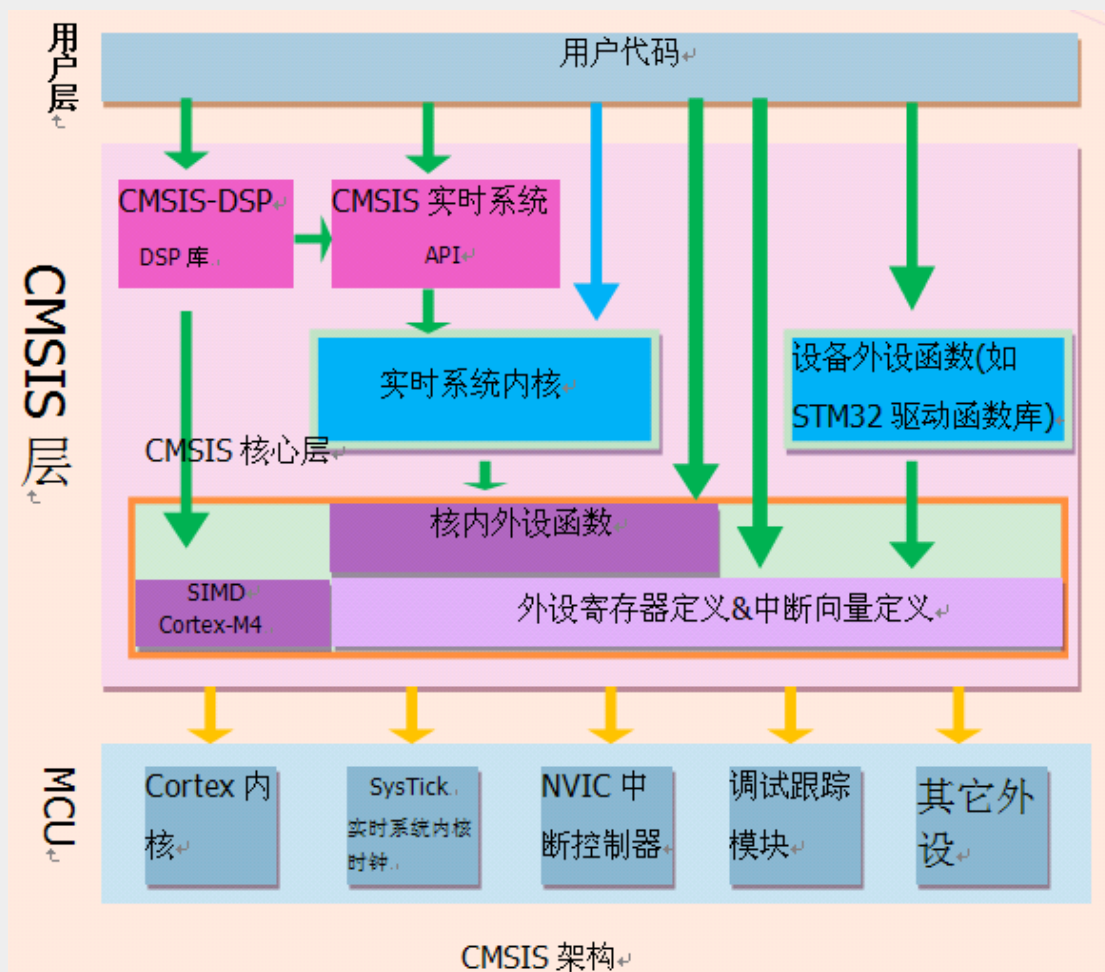
#### 3.2.1. CMSIS 标准

我们知道由 ST 公司生产的 STM32 采用的是 Cortex-M3 内核，内核是整个微控制器的 CPU。该内核是 ARM 公司设计的一个处理器体系架构。ARM 公司并不生产芯片，而是出售其芯片技术授权。ST 公司或其它芯片生产厂商如 TI，负责设计的是在内核之外的部件，被称为核外外设或片上外设、设备外设。如芯片内部的模数转换外设 ADC、串口 UART、定时器 TIM 等。内核与外设，如同 PC 上的 CPU 与主板、内存、显卡、硬盘的关系。



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

所谓 CMSIS 标准，实际是新建了一个软件抽象层。



CMSIS 标准中最主要的为 CMSIS 核心层，它包括了：

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

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

### 3.2.2. 库目录、文件简介

STM32 的 3.5 版库可以从官网获得，也可以直接从本书的附录光盘得到。本书主要采用最新版的 3.5 库文件，在高级篇的章节有部分代码是采用 3.0 的库开发的，因为 3.5 与 3.0 的库文件兼容性很好，对于旧版的代码我们仍然使用 3.0 版的。

解压后进入库目录：

***stm32f10x\_stdperiph\_lib\STM32F10x\_StdPeriph\_Lib\_V3.5.0***

各文件夹内容说明见图

4-1

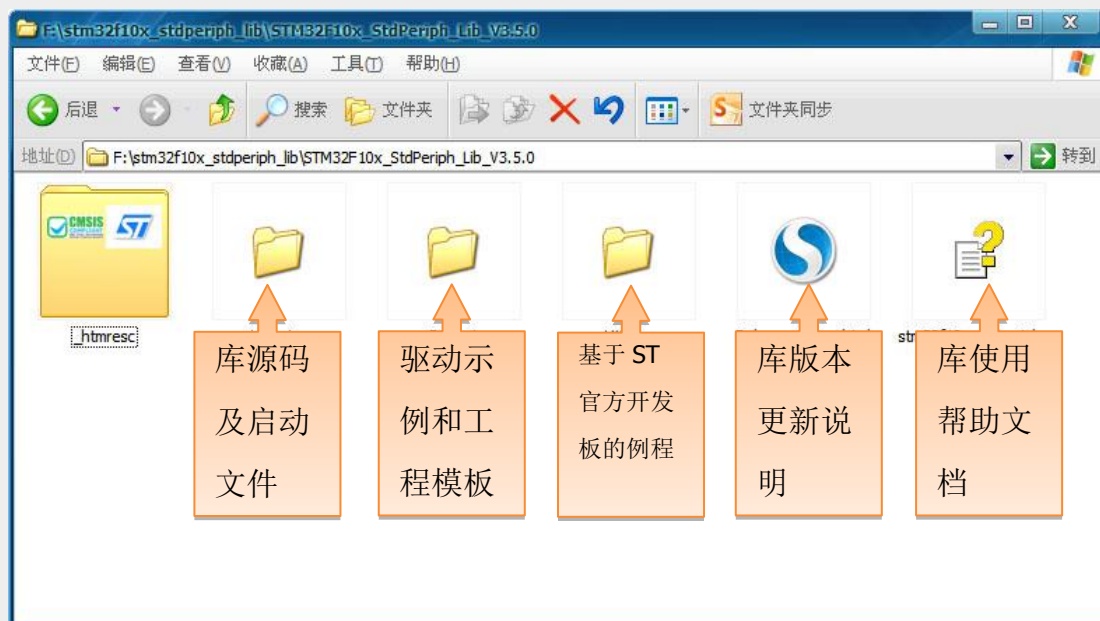


图 4-1

***Libraries*** 文件夹下是驱动库的源代码及启动文件。

***Project*** 文件夹下是用驱动库写的例子跟一个工程模板。

还有一个已经编译好的 ***HTML*** 文件，是***库帮助文档***，主要讲的是如何使用驱动库来编写自己的应用程序。说得形象一点，这个 ***HTML*** 就是告诉我们：ST 公司已经为你写好了每个外设的驱动了，想知道如何运用这些例子就来向我求救吧。不幸的是，这个帮助文档是英文的，这对很多英文不好的朋友来说是一个很大的障碍。但野火要告诉大家，英文仅仅是一种工具，绝对不能让它成为

我们学习的障碍。其实这些英文还是很简单的，我们需要的是拿下它的勇气。网上流传有一份中文版本的库帮助文档，但那个是 2.x 版本的，但 3.x 以上版本的目录结构和库函数接口跟 2.x 版本的区别还是比较大的，这点大家要注意下。

在使用库开发时，我们需要把 *libraries* 目录下的库函数文件添加到工程中，并查阅[库帮助文档](#)来了解 ST 提供的库函数，这个文档说明了每一个库函数的使用方法。

进入 *Libraries* 文件夹看到，关于内核与外设的库文件分别存放在 *CMSIS* 和 *STM32F10x\_StdPeriph\_Driver* 文件夹中。

*Libraries|CMSIS|CM3* 文件夹下又分为 *CoreSupport* 和 *DeviceSupport* 文件夹。

#### ■ core\_cm3.c 文件

在 *CoreSupport* 中的是位于 CMSIS 标准的 *核内设备函数层* 的 M3 核通用的源文件 *core\_cm3.c* 和头文件 *core\_cm3.h*，它们的作用是为那些采用 Cortex-M3 核设计 SOC 的芯片商设计的芯片外设提供一个进入 M3 内核的接口。这两个文件在其它公司的 M3 系列芯片也是相同的。至于这些功能是怎样用源码实现的，我们可以不用管它，我们只需把这个文件加进我们的工程文件即可，有兴趣的朋友可以深究。

*core\_cm3.c* 文件还有一些与编译器相关条件编译语句，用于屏蔽不同编译器的差异，我们在开发时不用管这部分，有兴趣可以了解一下。里面包含了一些跟编译器相关的信息，如：RealView Compiler (RVMDK)，ICC Compiler (IAR)，GNU Compiler。

```
1. /* define compiler specific symbols */
2. #if defined ( __CC_ARM )
3.     #define __ASM          __asm
4.     #define __INLINE       __inline
5.
6. #elif defined ( __ICCARM__ )
7.     #define __ASM          __asm
8.     #define __INLINE       inline
9. #elif defined ( __GNUC__ )
```

使用 RVMDK 编译器时的嵌入汇编与内联函数的关键字形式

使用 IAR 编译器时的形式



```
10. #define __ASM __asm
    #define __INLINE inline
11.
12. #elif defined ( __TASKING__ )
13. #define __ASM __asm
14. #define __INLINE inline
15. #endif
```

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

```
1. /* exact-width signed integer types */
2. typedef signed char int8_t;
3. typedef signed short int int16_t;
4. typedef signed int int32_t;
5. typedef signed __int64 int64_t;
6.
7. /* exact-width unsigned integer types */
8. typedef unsigned char uint8_t;
9. typedef unsigned short int uint16_t;
10. typedef unsigned int uint32_t;
11. typedef unsigned __int64 uint64_t;
```

这些新类型定义屏蔽了在不同芯片平台时，出现的诸如 `int` 的大小是 16 位，还是 32 位的差异。所以在我们以后的程序中，都将使用新类型如 `int8_t`、`int16_t`……

在稍旧版的程序中还可能会出现如 `u8`、`u16`、`u32` 这样的类型，请尽量避免这样使用，在这里提出来是因为初学时如果碰到这样的旧类型让人一头雾水，而且在以新的库建立的工程中是无法追踪到 `u8`、`u16`、`u32` 这些的定义的。

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

## ■ `system_stm32f10x.c` 文件

在 `DeviceSupport` 文件夹下的是启动文件、`外设寄存器定义&中断向量定义层` 的一些文件，这是由 ST 公司提供的。见图 4-2



图 4-2

***system\_stm32f10x.c***，是由 ST 公司提供的，遵守 CMSIS 标准。该文件的功能是设置系统时钟和总线时钟，M3 比 51 单片机复杂得多，并不是说我们外部给一个 8M 的晶振，M3 整个系统就以 8M 为时钟协调整个处理器的工作。我们还要通过 M3 核的核内寄存器来对 8M 的时钟进行倍频，分频，或者使用芯片内部的时钟。所有的外设都与时钟的频率有关，所以这个文件的时钟配置是很关键的。

***system\_stm32f10x.c*** 在实现系统时钟的时候要用到 PLL（锁相环），这就需要操作寄存器，寄存器都是以存储器映射的方式来访问的，所以该文件中包含了 ***stm32f10x.h*** 这个头文件。

#### ■ ***stm32f10x.h*** 文件

***stm32f10x.h*** 这个文件非常重要，是一个非常底层的文件。

所有处理器厂商都会将对内存的操作封装成一个宏，即我们通常说的寄存器，并且把这些实现封装成一个系统文件，包含在相应的开发环境中。这样，我们在开发自己的应用程序的时候只要将这个文件包含进来就可以了。



## ■ 启动文件

**Libraries\CMSIS\Core\CM3\startup\arm** 文件夹下是由汇编编写的系统启动文件，不同的文件对应不同的芯片型号，在使用时要注意。见图 4-3

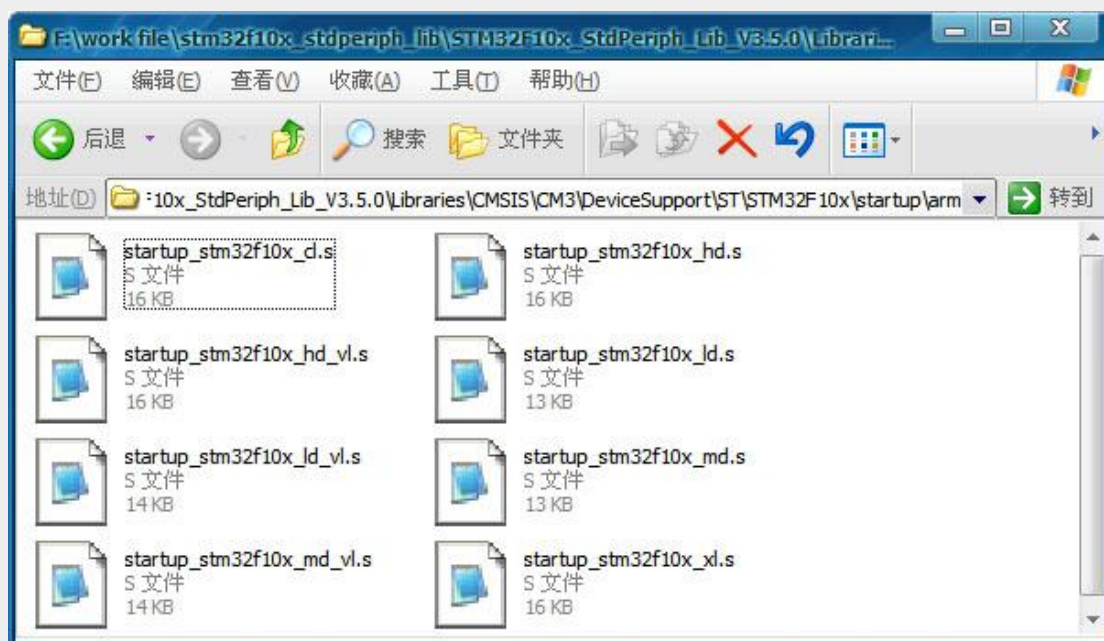


图 4-3

文件名的英文缩写的意义如下：

- cl: 互联型产品，stm32f105/107 系列
- vl: 超值型产品，stm32f100 系列
- xl: 超高密度（容量）产品，stm32f101/103 系列
- ld: 低密度产品，FLASH 小于 64K
- md: 中等密度产品，FLASH=64 or 128
- hd: 高密度产品，FLASH 大于 128

野火 M3 开发板中用的芯片是 **STM32F103VET6, 64KRAM, 512KROM**，是属于高密度产品，所以启动文件要选择 **startup\_stm32f10x\_hd.s**。

启动文件是任何处理器在上电复位之后最先运行的一段汇编程序。在我们编写的 c 语言代码运行之前，需要由汇编为 c 语言的运行建立一个合适的环境，接下来才能运行我们的程序。所以我们也要把启动文件添加进我们的的工程中去。

总的来说，启动文件的作用是：

1. 初始化堆栈指针 SP;
2. 初始化程序计数器指针 PC;
3. 设置堆、栈的大小;
4. 设置异常向量表的入口地址;
5. 配置外部 SRAM 作为数据存储器（这个由用户配置，一般的开发板可没有外部 SRAM）;
6. 设置 C 库的分支入口 \_\_main（最终用来调用 main 函数）;
7. 在 3.5 版的启动文件还调用了在 *system\_stm32f10x.c* 文件中的 *SystemIni()* 函数配置系统时钟，在旧版本的工程中要用户进入 main 函数自己调用 *SystemIni()* 函数。

#### ■ STM32F10x\_StdPeriph\_Driver 文件夹

*Libraries\STM32F10x\_StdPeriph\_Driver* 文件夹下有 *inc*（include 的缩写）跟 *src*（source 的简写）这两个文件夹，这都属于 CMSIS 的 *设备外设函数* 部分。*src* 里面是每个设备外设的驱动程序，这些外设是芯片制造商在 Cortex-M3 核外加进去的。

进入 *libraries* 目录下的 *STM32F10x\_StdPeriph\_Driver* 文件夹，见图 4-4。

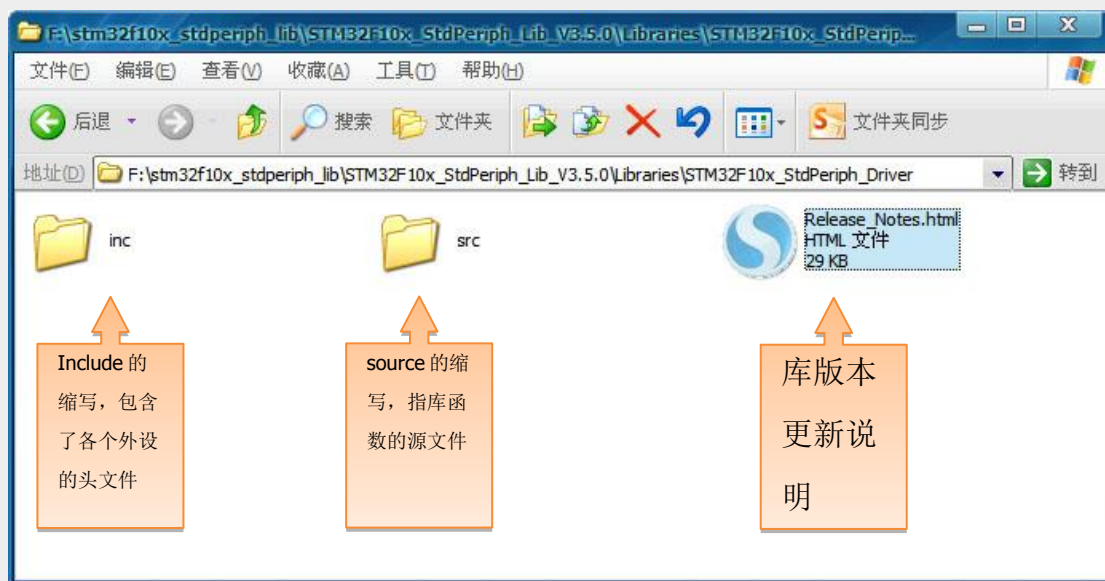


图 4-4

在 `src` 和 `inc` 文件夹里的就是 ST 公司针对每个 STM32 外设而编写的库函数文件, 每个外设对应一个 `.c` 和 `.h` 后缀的文件。我们把这类外设文件统称为: `stm32f10x_ppp.c` 或 `stm32f10x_ppp.h` 文件, PPP 表示外设名称。

如针对模数转换(ADC)外设, 在 `src` 文件夹下有一个 `stm32f10x_adc.c` 源文件, 在 `inc` 文件夹下有一个 `stm32f10x_adc.h` 头文件, 若我们开发的工程中用到了 STM32 内部的 ADC, 则至少要把这两个文件包含到工程里。

见图 4-5。

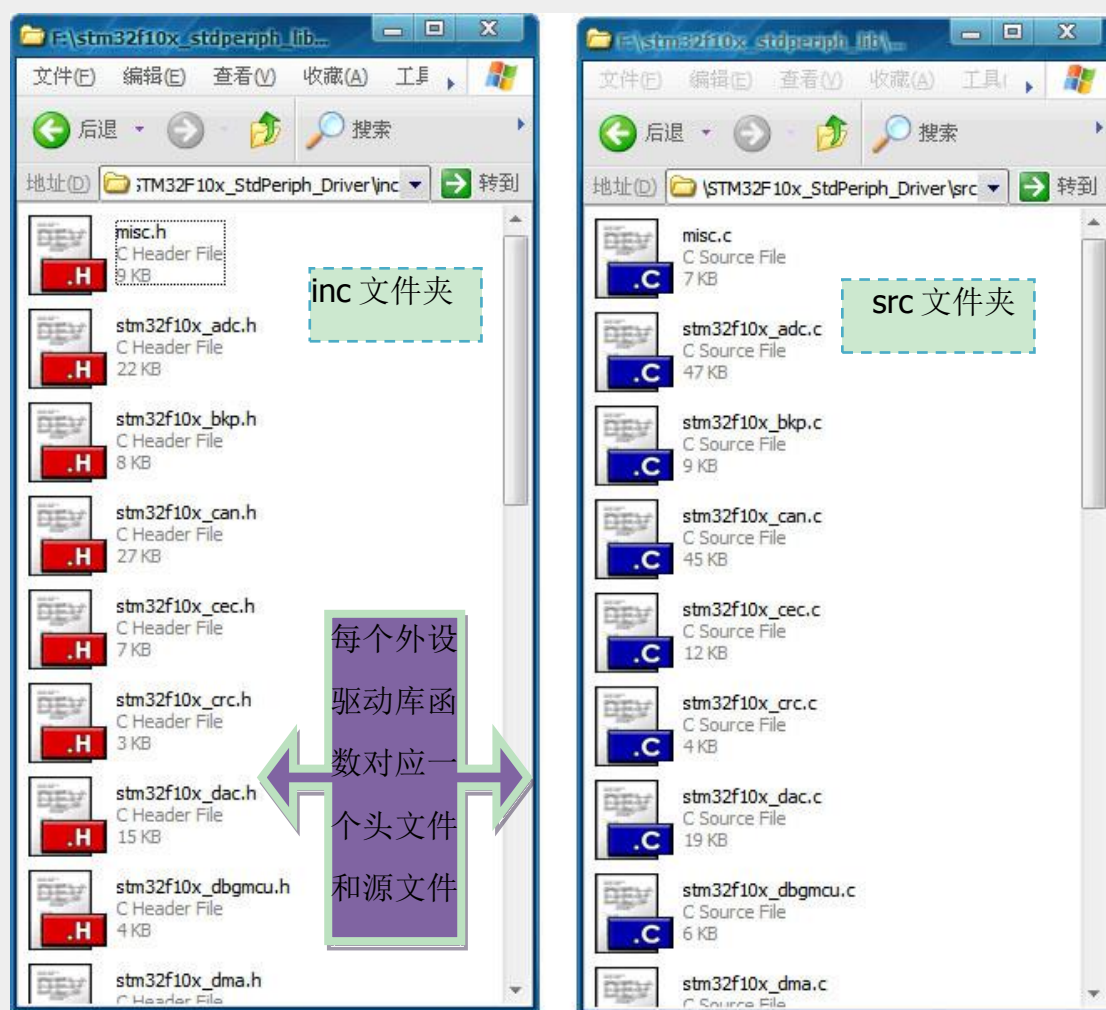


图 4-5

这两个文件夹中，还有一个很特别的 *misc.c* 文件，这个文件提供了外核对内核中的 NVIC(中断向量控制器)的访问函数，在配置中断时，我们必须把这个文件添加到工程中。

#### ■ *stm32f10x\_it.c*、*stm32f10x\_conf.h* 文件

在库目录的 *|Project|STM32F10x\_StdPeriph\_Template* 目录下，存放了官方的一个库工程模板，我们在用库建立一个完整的工程时，还需要添加这个目录下的 *stm32f10x\_it.c*、*stm32f10x\_it.h*、*stm32f10x\_conf.h* 这三个文件。

*stm32f10x\_it.c*，是专门用来编写中断服务函数的，在我们修改前，这个文件已经定义了一些系统异常的接口，其它普通中断服务函数由我们自己添

加。但是我们怎么知道这些中断服务函数的接口如何写呢？是不是可以自定义呢？答案当然不是的，这些都有可以在汇编启动文件中找到，具体的大家自个看库的启动文件的源码去吧。

**stm32f10x\_conf.h**，这个文件被包含进 **stm32f10x.h** 文件。是用来配置使用了什么外设的头文件，用这个头文件我们可以很方便地增加或删除上面 **driver** 目录下的外设驱动函数库。如下面的代码配置表示使用了 **gpio**、**rcc**、**spi**、**usart** 的外设库函数，其它的注释掉的部分，表示没有用到。

```
1.  /* Includes -----
   -----*/
2.  /* Uncomment/Comment the line below to enable/disable peripheral header
   file inclusion */
3.  //#include "stm32f10x_adc.h"
4.  //#include "stm32f10x_bkp.h"
5.  //#include "stm32f10x_can.h"
6.  //#include "stm32f10x_cec.h"
7.  //#include "stm32f10x_crc.h"
8.  //#include "stm32f10x_dac.h"
9.  //#include "stm32f10x_dbgmcu.h"
10. //#include "stm32f10x_dma.h"
11. //#include "stm32f10x_exti.h"
12. //#include "stm32f10x_flash.h"
13. //#include "stm32f10x_fsmc.h"
14. #include "stm32f10x_gpio.h"
15. //#include "stm32f10x_i2c.h"
16. //#include "stm32f10x_iwdg.h"
17. //#include "stm32f10x_pwr.h"
18. #include "stm32f10x_rcc.h"
19. //#include "stm32f10x_rtc.h"
20. //#include "stm32f10x_sdio.h"
21. #include "stm32f10x_spi.h"
22. //#include "stm32f10x_tim.h"
23. #include "stm32f10x_usart.h"
24. //#include "stm32f10x_wwdg.h"
25. //#include "misc.h" /* High level functions for NVIC and SysTick (add-
   on to CMSIS functions) */
```

**stm32f10x\_conf.h** 这个文件还可配置是否使用“断言”编译选项，在开发时使用断言可由编译器检查库函数传入的参数是否正确，软件编写成功后，去掉“断言”编译选项可使程序全速运行。可通过定义 **USE\_FULL\_ASSERT** 或取消定义来配置是否使用断言。

### 3.2.3. 库各文件间的关系

前面向大家简单介绍了各个库文件的作用，库文件是直接包含进工程即可，丝毫不用修改，而有的文件就要我们在使用的时候根据具体的需要进行配

置。接下来从整体上把握一下各个文件在库工程中的层次或关系，这些文件对应到 CMSIS 标准架构上。见图 3-6

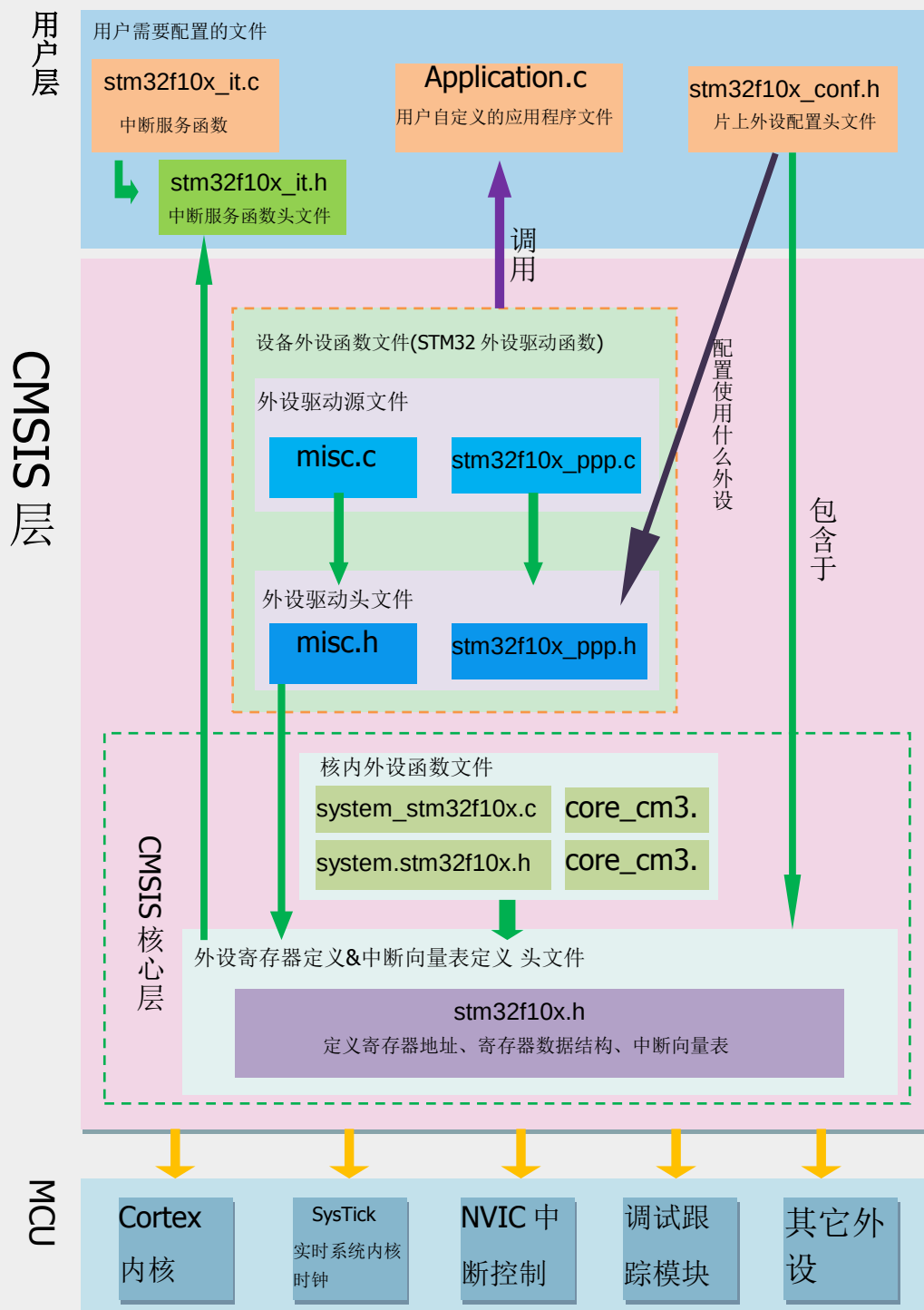


图 3-6 (STM32 库文件结构)



图 3-6 描述了 STM32 库各文件之间的调用关系，这个图省略了 DSP 核 (Cortex-M3 没有 DSP 核) 和实时系统层部分的文件关系。在实际的使用库开发工程的过程中，我们把位于 CMSIS 层的文件包含进工程，丝毫不用修改，也不建议修改。

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

### 3.2.4. 使用库帮助文档

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

#### ■ 常用官方资料

##### 1. 《stm32f10x\_stdperiph\_lib\_um.chm》

这个就是前面提到的库的帮助文档，在使用库函数时，我们最好通过查阅此文件来了解库函数原型，或库函数的 [调用](#) 的方法。也可以直接阅读源码里面的函数的函数说明。

##### 2. 《STM32 参考手册.pdf》

这个文件相当于 STM32 的 datasheet，它把 STM32 的时钟、存储器架构、及各种外设、寄存器都描述得清清楚楚。当我们对 STM32 的库函数的 [实现方式](#) 感到困惑时，可查阅这个文件，以直接配置寄存器方式开发的话查阅这个文档的频率会更高。但你会累死。

##### 3. 《Cortex-M3 权威指南》 宋岩译。

该手册详细讲解了 Cortex 内核的架构和特性，要深入了解 Cortex-M3 内核，这是首选，经典中的经典呀。

##### 4. 当然还有其他很有用的官方文档，这里就不再赘述……

## ■ 初识库函数

所谓库函数，就是 STM32 的库文件中为我们编写好的函数接口，我们只要调用这些库函数，就可以对 STM32 进行配置，达到控制目的。我们可以不知道库函数是如何实现的，但我们调用函数必须要知道[函数的功能](#)、[可传入的参数及其意义](#)、和[函数的返回值](#)。

于是，有读者就问那么多函数我怎么记呀？野火的回答是：会查就行了，哪个人记得了那么多。所以我们学会查阅[库帮助文档](#)是很有必要的。

打开库帮助文档 [stm32f10x\\_stdperiph\\_lib\\_um.chm](#) 见图 3-7

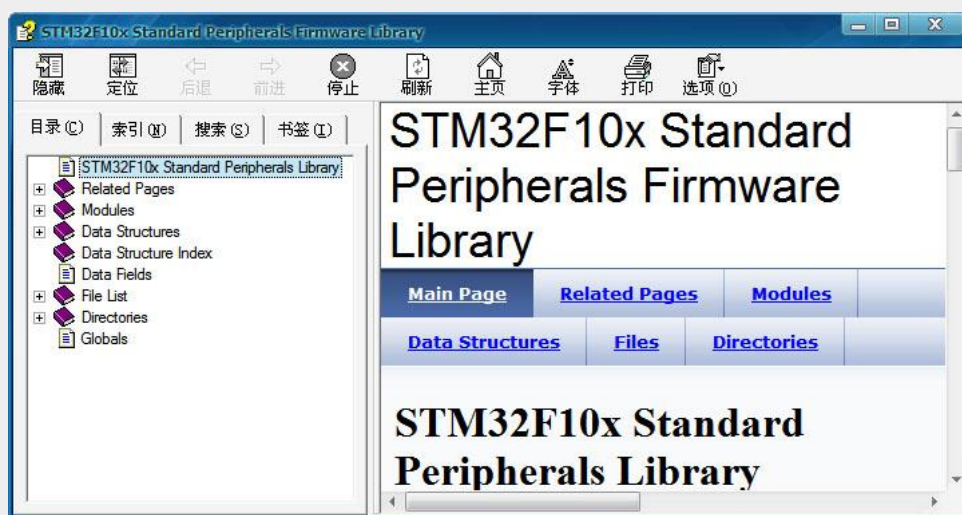


图 3-7

层层打开文档的目录标签 Modules\STM32F10x\_StdPeriph\_Driver，可看到

STM32F10x\_StdPeriph\_Driver 标签下有很多外设驱动文件的名字 MISC、ADC、BKP、CAN 等标签。我们试着查看 ADC 的[初始化库函数\(ADC\\_Init\)](#)看看，继续打开标签\ADC\ADC\_Exported\_Functions\Functions\ADC\_Init 见图



## 3-8

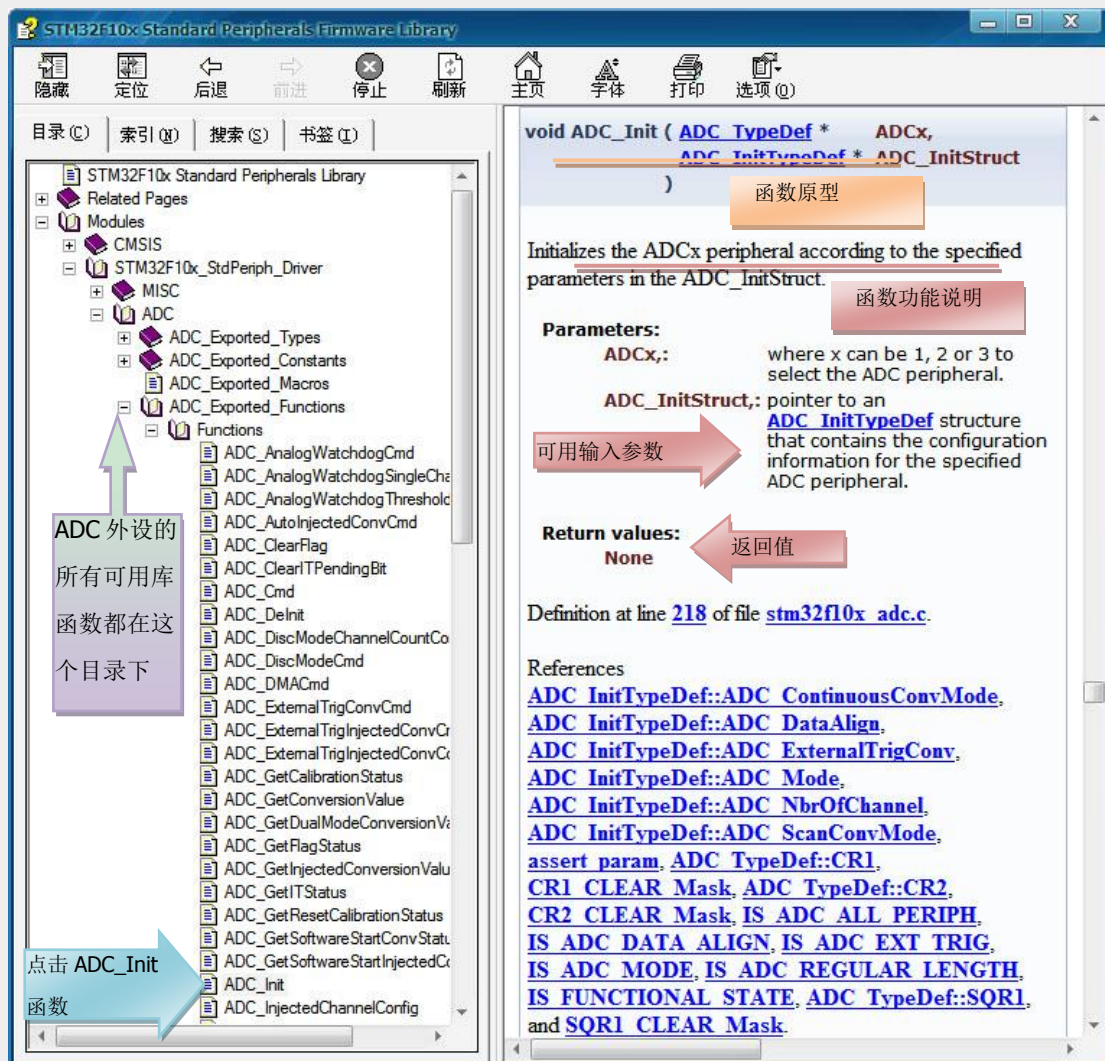


图 3-8

利用这个文档，我们即使没有去看它的具体代码，也知道要怎么利用它了。

如它的功能是：以 `ADC_InitStruct` 参数配置 ADC，进行初始化。原型为 `void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct)`

其中输入的参数 `ADCx` 和 `ADC_InitStruct` 均为库文档中定义的 **自定义数据类型**，这两个传入参数均为结构体指针。初学时，我们并不知道如 `ADC_TypeDef` 这样的类型是什么意思，可以点击函数原型中带下划线的 `ADC_TypeDef` 就可以查看这是什么类型了。

就这样初步了解了一下库函数，读者就可以发现 STM32 的库是写得很优美的。每个函数和数据类型都符合 [见名知义](#) 的原则，当然，这样的名称写起来特别长，而且对于我们来说要输入这么长的英文，很容易出错，所以在开发软件的时候，在用到库函数的地方，直接把[库帮助文档](#)中函数名称复制粘贴到工程文件就可以了。

## 4. GPIO-流水灯的前后今生

通过前面的内容，读者对库仅仅是建立了一个非常模糊的印象。

作为大家的第一个 STM32 例程，野火认为很有必要进行足够深入的分析，才能从**根本上**扫清读者对使用库函数的**困惑**。而且，只要读者利用这个 LED 例程，真正领会了库开发的流程以及原理，再进行其它外设的开发就变得相当简单了。

所以本章的任务是：

- 从 STM32 库的**实现原理**上解答 库到底是什么、为什么要用库、用库与直接配置寄存器的区别等问题。
- 让读者了解具体利用库的开发流程，熟悉库函数的结构，达到举一反三的效果，这次可就不是喝稀粥了，保证有吃干饭，所学就是所用的效果。

### 4.1. STM32 的 GPIO

想要控制 LED 灯，当然是通过控制 STM32 芯片的 I/O 引脚**电平的高低**来实现。在 STM32 芯片上，I/O 引脚可以被软件设置成各种**不同的功能**，如输入或输出，所以被称为 GPIO (General-purpose I/O)。而 GPIO 引脚又被分为 GPIOA、GPIOB……GPIOG 不同的组，每组端口分为 0~15，共 16 个**不同的引脚**，对于不同型号芯片，端口的组和引脚的数量不同，具体请参考相应芯片型号的 datasheet。

于是，控制 LED 的步骤就自然整理出来了：

1. GPIO 端口引脚多 --> 就要选定需要控制的**特定引脚**
2. GPIO 功能如此丰富 --> 配置需要的**特定功能**
3. 控制 LED 的亮和灭 --> 设置 GPIO 输出电压的**高低**

继续思考，要控制 GPIO 端口，就要涉及到控制相关的寄存器。这时我们就要查一查与 GPIO 相关的寄存器了，可以通过《STM32 参考手册》来查看，见图 5-1

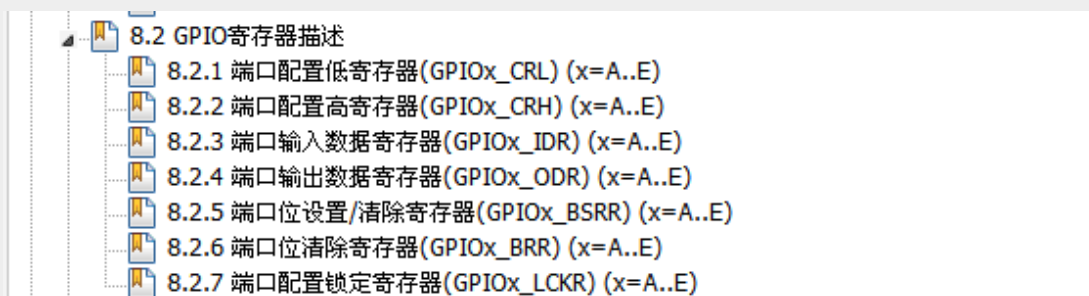


图 5-1

图中的 7 个寄存器，相应的功能在文档上有详细的说明。可以分为以下 4 类，其功能简要概括如下：

1. 配置寄存器：选定 GPIO 的**特定功能**，最基本的如：选择作为输入还是输出端口。
2. 数据寄存器：保存了 GPIO 的**输入电平**或 将要**输出的电平**。
3. 位控制寄存器：设置某引脚的**数据**为 1 或 0，控制输出的电平。
4. 锁定寄存器：设置某**锁定引脚**后，就不能修改其配置。

注：要想知道其功能严谨、详细的描述，请读者养成习惯在正式使用时，要以官方的 **datasheet** 为准，在这里只是简单地概括其功能进行说明。

关于寄存器名称上**标号 x** 的意义，如：GPIOx\_CRL、GPIOx\_CRH，这个 x 的取值可以为图中括号内的值(A……E)，表示这些寄存器也跟 GPIO 一样，也是**分组**的。也就是说，对于端口 GPIOA 和 GPIOB，**它们都有互不相干的一组寄存器**，如控制 GPIOA 的寄存器名为 GPIOA\_CRL、GPIOA\_CRH 等，而控制 GPIOB 的则是不同的、被命名为 GPIOB\_CRL、GPIOB\_CRH 等寄存器。

我们的程序代码以野火 STM32 第二代开发板为例，根据其硬件连接图来分析。

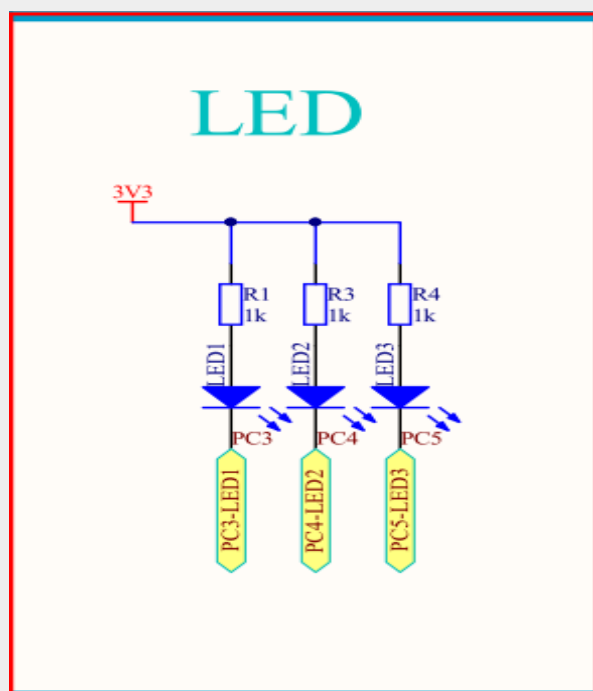


图 5-2

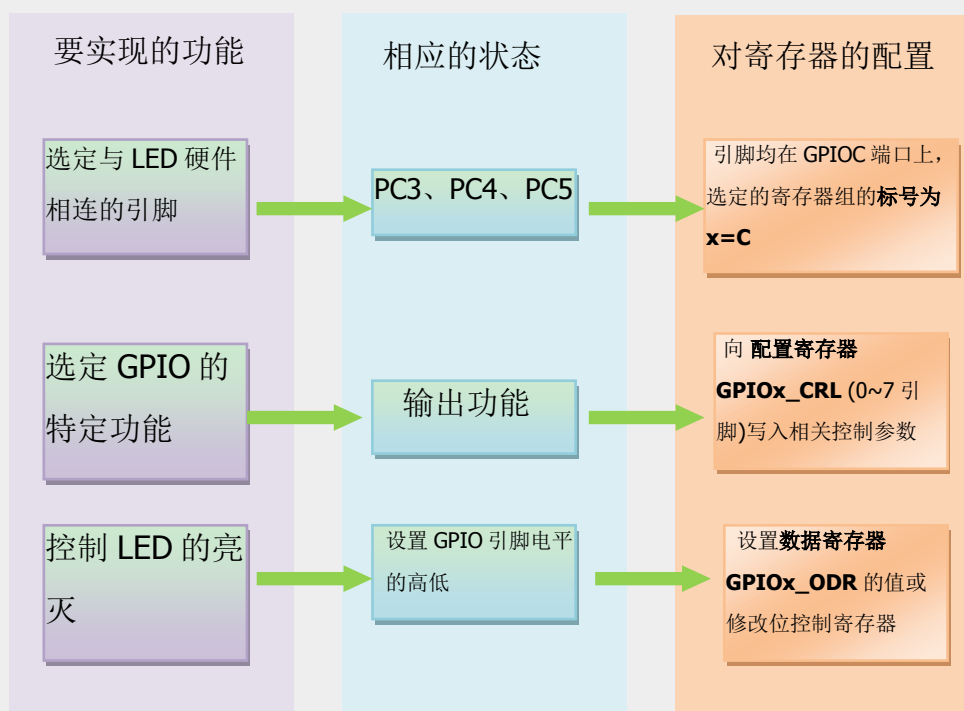


图 5-3

从这个图我们可以知道 STM32 的功能，实际上也是通过配置寄存器来实现的。配置寄存器的具体参数，需要参考《STM32 参考手册》的寄存器说明。见图 5-4。

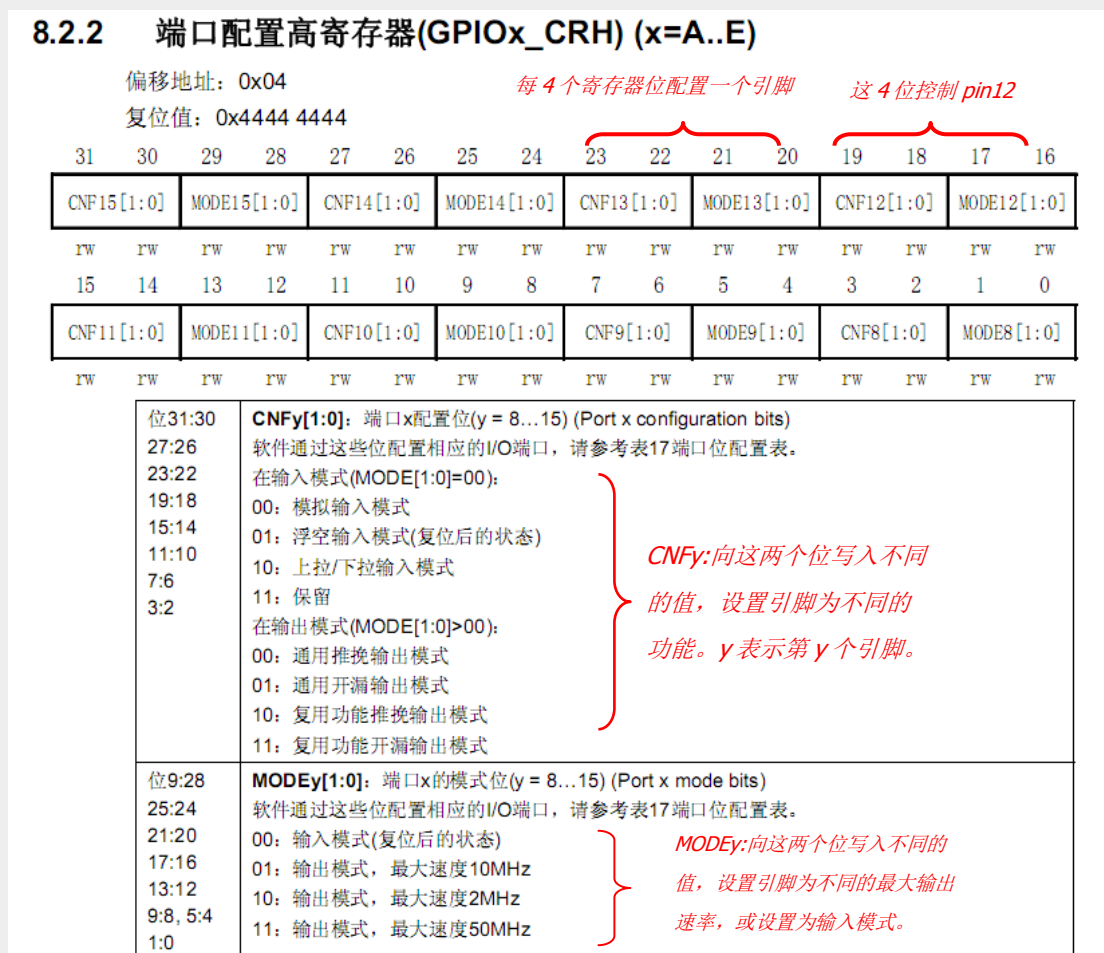


图 5-4

如图, 对于 GPIO 端口, 每个端口有 16 个引脚, 每个引脚的模式由寄存器的 4 个位控制, 每四位又分为两位控制引脚配置(CNFy[1:0]), 两位控制引脚的模式及最高速度(MODEy[1:0]), 其中 y 表示第 y 个引脚。这个图是 GPIOx\_CRH 寄存器的说明, 配置 GPIO 引脚模式的一共有两个寄存器, CRH 是高寄存器, 用来配置高 8 位引脚: pin8~pin15。还有一个称为 CRL 寄存器, 如果我们要配置 pin0~pin7 引脚, 则要在寄存器 CRL 中进行配置。

举例说明对 CRH 的寄存器的配置: 当给 GPIOx\_CRH 寄存器的第 28 至 29 位设置为参数 "11", 并在第 30 至 31 位设置为参数 "00", 则把 x 端口第 15



个引脚的模式配置成了“输出的最大速度为 50MHz 的通用推挽输出模式、”，其它引脚可通过其 GPIOx\_CRH 或 GPIOx\_CRL 的其它寄存器位来配置。至于 x 端口的 x 是指端口 GPIOA 还是 GPIOB 还要具体到不同的寄存器基址，这将在后面分析。

接下来分析要控制引脚电平高低，需要对寄存器进行什么具体的操作。见图 5-5。

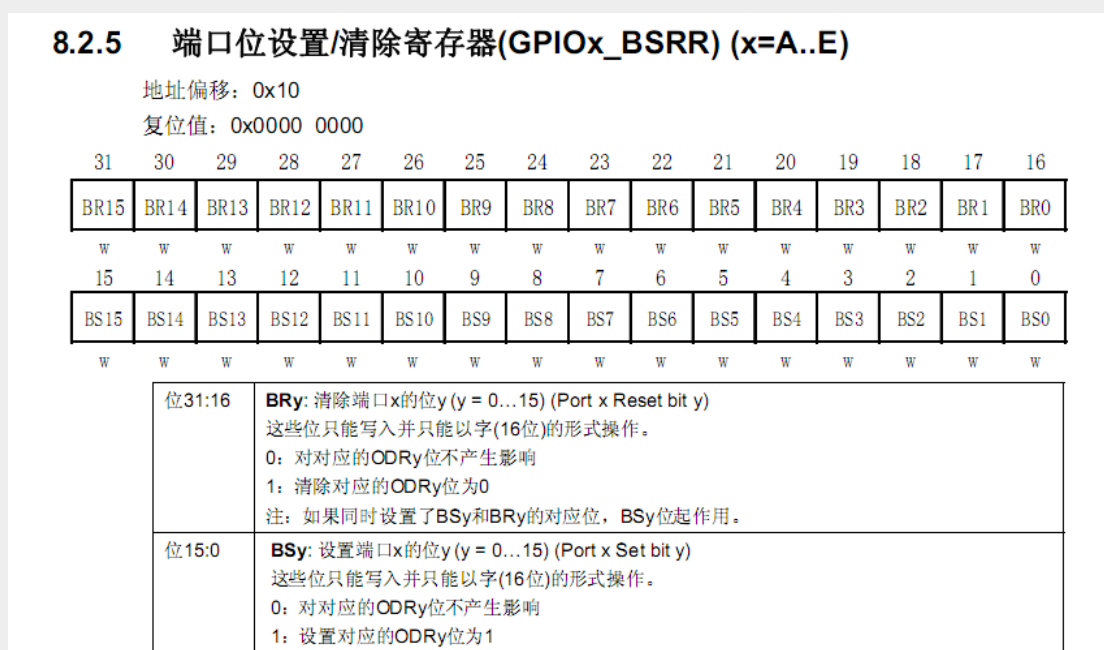


图 5-5

由寄存器说明图可知，一个引脚 y 的输出数据由 GPIOx\_BSRR 寄存器位的 2 个位来控制分别为 **BRy (Bit Reset y)**和 **BSy (Bit Set y)**，**BRy**位用于写 1 清零，使引脚输出低电平，**BSy**位用来写 1 置 1，使引脚输出高电平。而对这两个位进行写零都是无效的。(还可以通过设置寄存器 ODR 来控制引脚的输出。)

例如：对 x 端口的寄存器 GPIOx\_BSRR 的**第 0 位(BS0)**进行写 1，则 x 端口的第 0 引脚被设置为 1，输出高电平，若要令第 0 引脚再输出低电平，则需要向 GPIOx\_BSRR 的**第 16 位(BR0)**写 1。



## 4.2. STM32 的地址映射

### 温故而知新——stm32f10x.h 文件

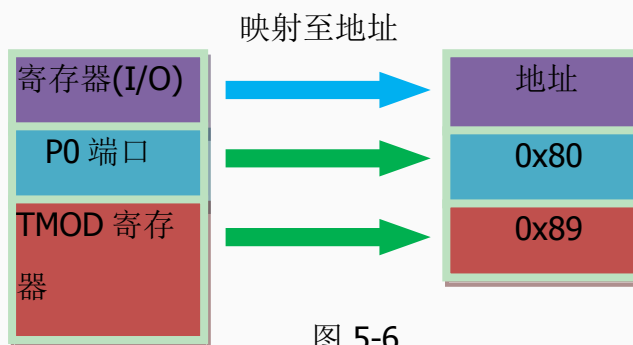
首先请大家回顾一下在 51 单片机上点亮 LED 是怎样实现的。这太简单了，几行代码就搞定。

```
1. #include<reg52.h>
2. int main (void)
3. {
4.     P0=0;
5.     while(1);
6. }
```

以上代码就可以点亮 P0 端口与 LED 阴极相连的 LED 灯了，当然，这里省略了启动代码。为什么这个 **P0=0;** 句子就能控制 P0 端口为低电平？很多刚入门 51 单片机的同学还真解释不来，关键之处在于这个代码所包含的头文件 **<reg52.h>**。

在这个文件下有以下的定义：

```
1. /* BYTE Registers */
2. sfr P0      = 0x80;
3. sfr P1      = 0x90;
4. sfr P2      = 0xA0;
5. sfr P3      = 0xB0;
6. sfr PSW     = 0xD0;
7. sfr ACC     = 0xE0;
8. sfr B       = 0xF0;
9. sfr SP      = 0x81;
10. sfr DPL     = 0x82;
11. sfr DPH     = 0x83;
12. sfr PCON   = 0x87;
13. sfr TCON   = 0x88;
14. sfr TMOD   = 0x89;
15. sfr TL0    = 0x8A;
16. sfr TL1    = 0x8B;
17. sfr TH0    = 0x8C;
18. sfr TH1    = 0x8D;
19. sfr IE     = 0xA8;
20. sfr IP     = 0xB8;
21. sfr SCON   = 0x98;
22. sfr SBUF   = 0x99;
```



这些定义被称为**地址映射**。

所谓地址映射，就是将芯片上的**存储器**甚至**I/O**等资源与地址建立一一对应的关系。如果某地址对应着某**寄存器**，我们就可以运用 c 语言的指针来寻址并修改这个地址上的内容，从而实现修改该寄存器的内容。

正是因为 `<reg52.h>` 头文件中有了对于各种寄存器和 I/O 端口的地址映射，我们才可以在 51 单片机程序中方便地使用 `P0 = 0xFF; TMOD = 0xFF` 等赋值句子对寄存器进行配置，从而控制单片机。

Cortex-M3 的地址映射也是类似的。Cortex-M3 有 32 根地址线，所以它的寻址空间大小为  $2^{32} \text{ bit} = 4\text{GB}$ 。ARM 公司设计时，预先把这 4GB 的寻址空间大致地分配好了。它把地址从 0x4000 0000 至 0x5FFF FFFF (512MB) 的地址分配给片上外设。通过把片上外设的寄存器映射到这个地址区，就可以简单地以访问内存的方式，访问这些外设的寄存器，从而控制外设的工作。结果，片上外设可以使用 C 语言来操作。M3 存储器映射见图 5-7

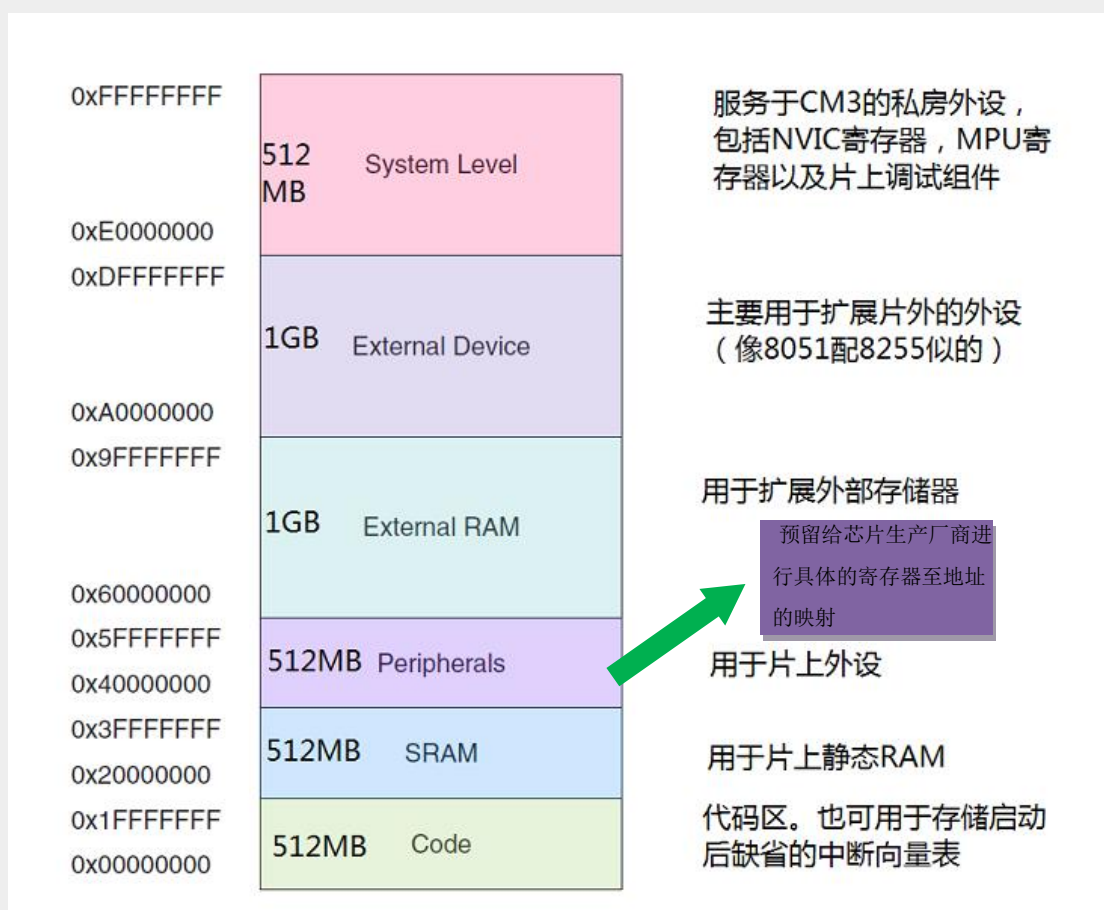


图 5-7

**stm32f10x.h** 这个文件中重要的内容就是把 STM32 的所有寄存器进行地址映射。如同 51 单片机的 **<reg52.h>** 头文件一样，**stm32f10x.h** 像一个大表格，我们在使用的时候就是通过宏定义进行类似查表的操作，大家想像一下没有这个文件的话，我们要怎样访问 STM32 的寄存器？有什么缺点？

不进行这些宏定义的缺点有：

- 1、地址容易写错
- 2、我们需要查大量的手册来确定哪个地址对应哪个寄存器
- 3、看起来还不好看，且容易造成编程的错误，效率低，影响开发进度。

当然，这些工作都是由 ST 的固件工程师来完成的，只有设计 M3 的人才是最了解 M3 的，才能写出完美的库。

在这里我们以外接了 LED 灯的外设 GPIOC 为例，在这个文件中有这样的一系列宏定义：

```
1. #define GPIOC_BASE          (APB2PERIPH_BASE + 0x1000)
2. #define APB2PERIPH_BASE     (PERIPH_BASE + 0x10000)
3. #define PERIPH_BASE         ((uint32_t)0x40000000)
```

这几个宏定义是从文件中的几个部分抽离出来的，具体的读者可参考 **stm32f10x.h 源码**。

### 外设基地址

首先看到 **PERIPH\_BASE** 这个宏，宏展开为 **0x4000 0000**，并把它强制转换为 **uint32\_t** 的 32 位类型数据，这是因为地 STM32 的地址是 32 位的，是不是觉得 **0x4000 0000** 这个地址很熟？是的，这个是 Cortex-M3 核分配给片上外设的从 **0x4000 0000** 至 **0x5FFF FFFF** 的 512MB 寻址空间中的第一个地址，我们把 **0x4000 0000** 称为外设基地址。

### 总线基地址

接下来是宏 **APB2PERIPH\_BASE**，宏展开为 **PERIPH\_BASE**（**外设基地址**）加上偏移地址 **0x1 0000**，即指向的地址为 **0x4001 0000**。这个 **APB2PERIPH\_BASE** 宏是什么地址呢？STM32 不同的外设是挂载在不同的总

线上的，见图 5-8。有 AHB 总线、APB2 总线、APB1 总线，挂载在这些总线上的外设有一定的地址范围。

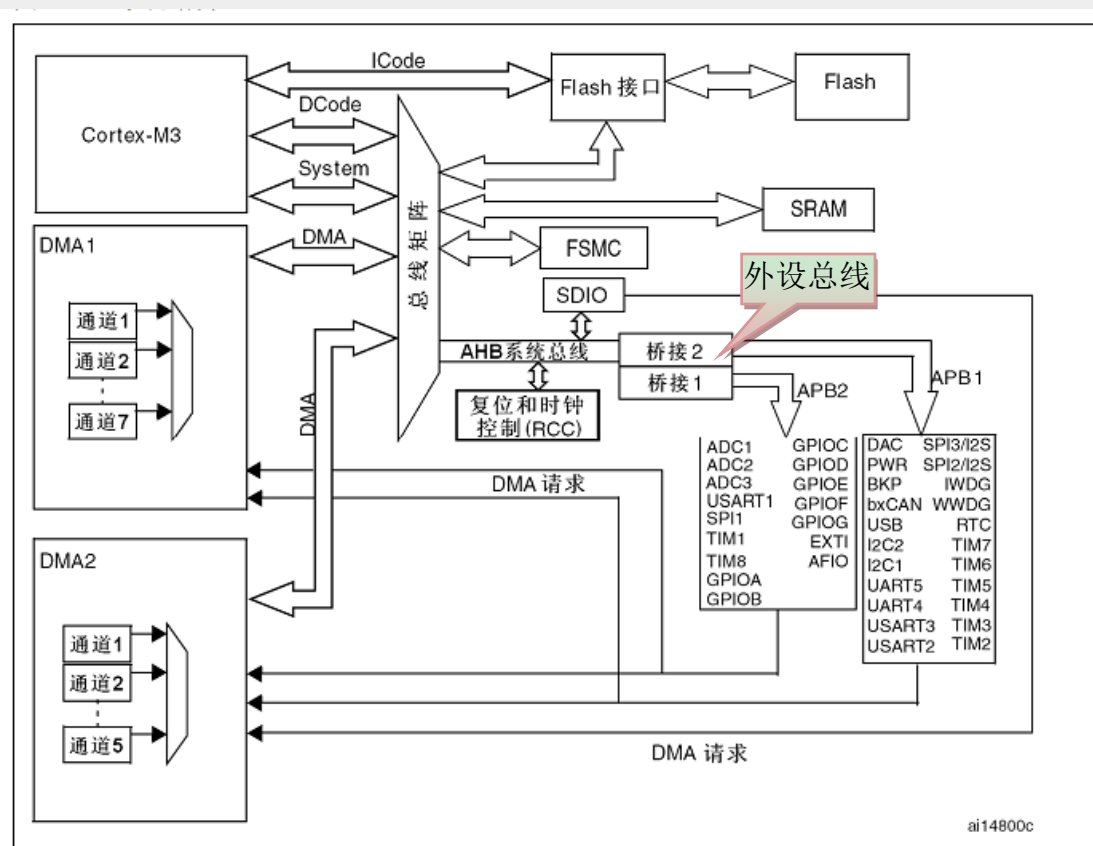


图 5-8

其中像 GPIO、串口 1、ADC 及部分定时器是挂载这个被称为 APB2 的总线上，挂载到 APB2 总线上的外设地址空间是从 0x4001 0000 至地址 0x4001 3FFF。这里的第一个地址，也就是 0x4001 0000，被称为 **APB2PERIPH\_BASE** (APB2 总线外设的基地址)。

而 APB2 总线基地址相对于外设基地址的偏移量为 0x1 0000 个地址，即为 APB2 相对外设基地址的偏移地址。

见表：

| 地址范围 | 总线 | 总线基地址 | 总线基地址相对外设基地址<br>(0x4000 000) 的偏移 |
|------|----|-------|----------------------------------|
|      |    |       |                                  |

|                           |      |             | 量        |
|---------------------------|------|-------------|----------|
| 0x4001 8000 - 0x5003 FFFF | AHB  | 0x4001 8000 | 0x1 8000 |
| 0x4001 0000 - 0x4001 7FFF | APB2 | 0x4001 0000 | 0x1 0000 |
| 0x4000 0000 - 0x4000FFFF  | APB1 | 0x4000 0000 | 0x0 0000 |

由这个表我们可以知道，[stm32f10x.h](#) 这个文件中必然还有以下的宏：

```
1. #define APB1PERIPH_BASE    PERIPH_BASE
```

因为偏移量为零，所以 APB1 的地址直接就等于外设基地址

### 寄存器组基地址

最后到了宏 [GPIOC\\_BASE](#)，宏展开为 [APB2PERIPH\\_BASE](#) (APB2 总线外设的基地址)加上相对 APB2 总线基地址的偏移量 [0x1000](#) 得到了 GPIOC 端口的寄存器组的基地址。这个所谓的寄存器组又是什么呢？它包括什么寄存器？

细看 [stm32f10x.h](#) 文件，我们还可以发现以下类似的宏：

```
1. #define GPIOA_BASE    (APB2PERIPH_BASE + 0x0800)
2. #define GPIOB_BASE    (APB2PERIPH_BASE + 0x0C00)
3. #define GPIOC_BASE    (APB2PERIPH_BASE + 0x1000)
4. #define GPIOD_BASE    (APB2PERIPH_BASE + 0x1400)
```

除了 GPIOC 寄存器组的地址，还有 GPIOA、GPIOB、GPIOD 的地址，并且这些地址是不一样的。

前面提到，每组 GPIO 都对应着独立的一组寄存器，查看 [stm32](#) 的 [datasheet](#)，看到寄存器说明如下图：

## 8.2.2 端口配置高寄存器(GPIOx\_CRH) (x=A..E)

偏移地址: 0x04

复位值: 0x4444 4444

|            |    |             |    |            |    |             |    |            |    |             |    |            |    |             |    |
|------------|----|-------------|----|------------|----|-------------|----|------------|----|-------------|----|------------|----|-------------|----|
| 31         | 30 | 29          | 28 | 27         | 26 | 25          | 24 | 23         | 22 | 21          | 20 | 19         | 18 | 17          | 16 |
| CNF15[1:0] |    | MODE15[1:0] |    | CNF14[1:0] |    | MODE14[1:0] |    | CNF13[1:0] |    | MODE13[1:0] |    | CNF12[1:0] |    | MODE12[1:0] |    |
| rw         | rw | rw          | rw | rw         | rw | rw          | rw | rw         | rw | rw          | rw | rw         | rw | rw          | rw |
| 15         | 14 | 13          | 12 | 11         | 10 | 9           | 8  | 7          | 6  | 5           | 4  | 3          | 2  | 1           | 0  |
| CNF11[1:0] |    | MODE11[1:0] |    | CNF10[1:0] |    | MODE10[1:0] |    | CNF9[1:0]  |    | MODE9[1:0]  |    | CNF8[1:0]  |    | MODE8[1:0]  |    |
| rw         | rw | rw          | rw | rw         | rw | rw          | rw | rw         | rw | rw          | rw | rw         | rw | rw          | rw |

图 5-9

注意到这个说明中有一个偏移地址: 0x04, 这里的偏移地址的是相对哪个地址的偏移呢? 下面进行举例说明。

对于 GPIOC 组的寄存器, GPIOC 含有的 端口配置高寄存器(GPIOC\_CRH) 寄存器地址为: **GPIOC\_BASE + 0x04**。

假如是 GPIOA 组的寄存器, 则 GPIOA 含有的 端口配置高寄存器 (GPIOA\_CRH) 寄存器地址为: **GPIOA\_BASE + 0x04**。

也就是说, 这个偏移地址, 就是该寄存器 相对所在寄存器组基地址的偏移量。

于是, 读者可能会想, 大概这个文件含有一个类似如下的宏(当初野火也是这么想的):

```
1. #define GPIOC_CRH    (GPIOC_BASE + 0x04)
```

这个宏, 定义了 GPIOC\_CRH 寄存器的具体地址, 然而, 在 stm32f10x.h 文件中并没有这样的宏。ST 公司的工程师采用了更巧妙的方式来确定这些地址, 请看下一小节——STM32 库对寄存器的封装。

## 4.3. STM32 库对寄存器的封装

ST 的工程师用结构体的形式, 封装了寄存器组, c 语言结构体学的不好的同学, 可以在这里补补课了。在 stm32f10x.h 文件中, 有以下代码:

```
1. #define GPIOA          ((GPIO_TypeDef *) GPIOA_BASE)
2. #define GPIOB          ((GPIO_TypeDef *) GPIOB_BASE)
3. #define GPIOC          ((GPIO_TypeDef *) GPIOC_BASE)
```

有了这些宏，我们就可以定位到具体的寄存器地址，在这里发现了一个陌生的类型 ***GPIO\_TypeDef***，追踪它的定义，可以在 ***stm32f10x.h*** 文件中找到如下代码：

```
1. typedef struct
2. {
3.     __IO uint32_t CRL;
4.     __IO uint32_t CRH;
5.     __IO uint32_t IDR;
6.     __IO uint32_t ODR;
7.     __IO uint32_t BSRR;
8.     __IO uint32_t BRR;
9.     __IO uint32_t LCKR;
10. } GPIO_TypeDef;
```

其中 ***\_\_IO*** 也是一个 ST 库定义的宏，宏定义如下：

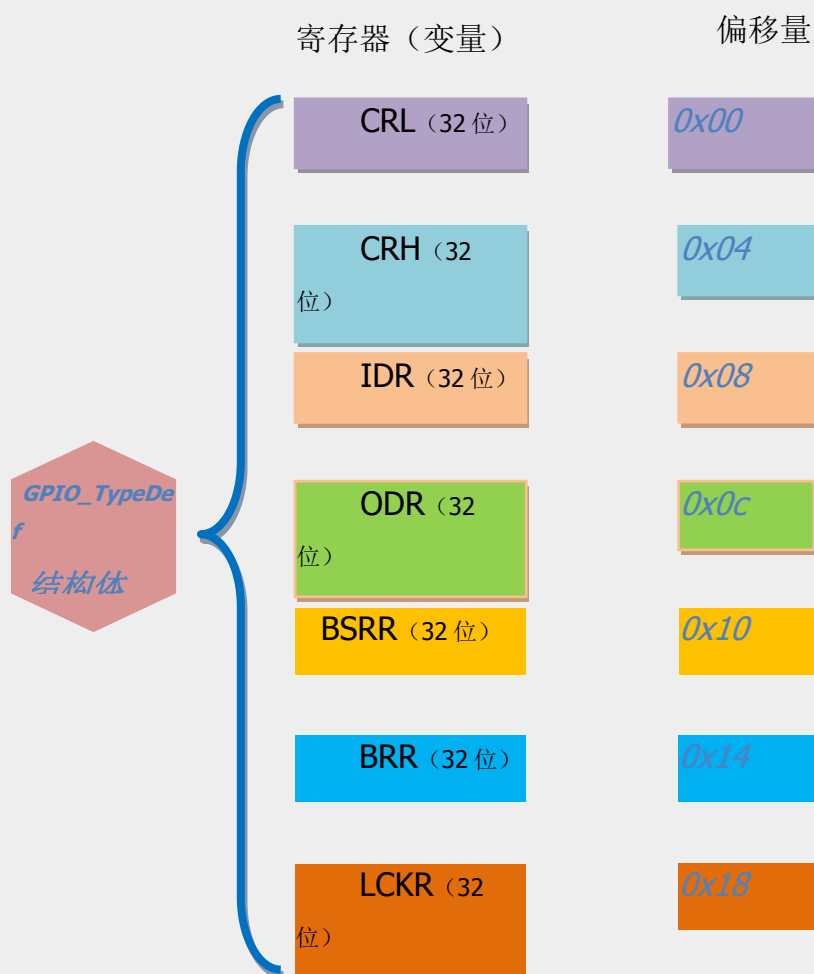
```
1. #define __O volatile /*!< defines 'write only' permissions */
2. #define __IO volatile /*!< defines 'read / write' permissions */
```

***volatile*** 是 c 语言的一个关键字，有关 ***volatile*** 的用法可查阅相关的 C 语言书籍。

回到 ***GPIO\_TypeDef*** 这段代码，这个代码用 ***typedef*** 关键字声明了名为 ***GPIO\_TypeDef*** 的结构体类型，结构体内又定义了 7 个 ***\_\_IO uint32\_t*** 类型的变量。这些变量每个都为 32 位，也就是每个变量占内存空间 4 个字节。在 c 语言中，结构体内变量的存储空间是连续的，也就是说假如我们定义了一个 ***GPIO\_TypeDef***，这个结构体的首地址（变量 CRL 的地址）若为 ***0x4001 1000***，那么结构体中第二个变量（CRH）的地址即为 ***0x4001 1000 + 0x04***，加上的这个 ***0x04***，正是代表 4 个字节地址的偏移量。

细心的读者会发现，这个 ***0x04*** 偏移量，正是 ***GPIOx\_CRH*** 寄存器相对于所在寄存器组的偏移地址，见图 5-9。同理，***GPIO\_TypeDef*** 结构体内其它变量的偏移量，也和相应的寄存器偏移地址相符。于是，只要我们匹配了结构体的首地址，就可以确定各寄存器的具体地址了。





有了这些准备，就可以分析本小节的第一段代码了：

```
4. #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
5. #define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
6. #define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
```

**GPIOA\_BASE** 在上一小节已解析，是一个代表 GPIOA 组寄存器的地址。**(GPIO\_TypeDef \*)** 在这里的作用则是把 **GPIOA\_BASE** 地址转换为 **GPIO\_TypeDef** 结构体指针类型。

有了这样的宏，以后我们写代码的时候，如果要修改 GPIO 的寄存器，就可以用以下的方式来实现。代码分析见注释。

```
1. GPIO_TypeDef * GPIOx;      //定义一个 GPIO_TypeDef 型结构体指针 GPIOx
2. GPIOx = GPIOA;             //把指针地址设置为宏 GPIOA 地址
3. GPIOx->CRL = 0xffffffff;    //通过指针访问并修改 GPIOA_CRL 寄存器
```

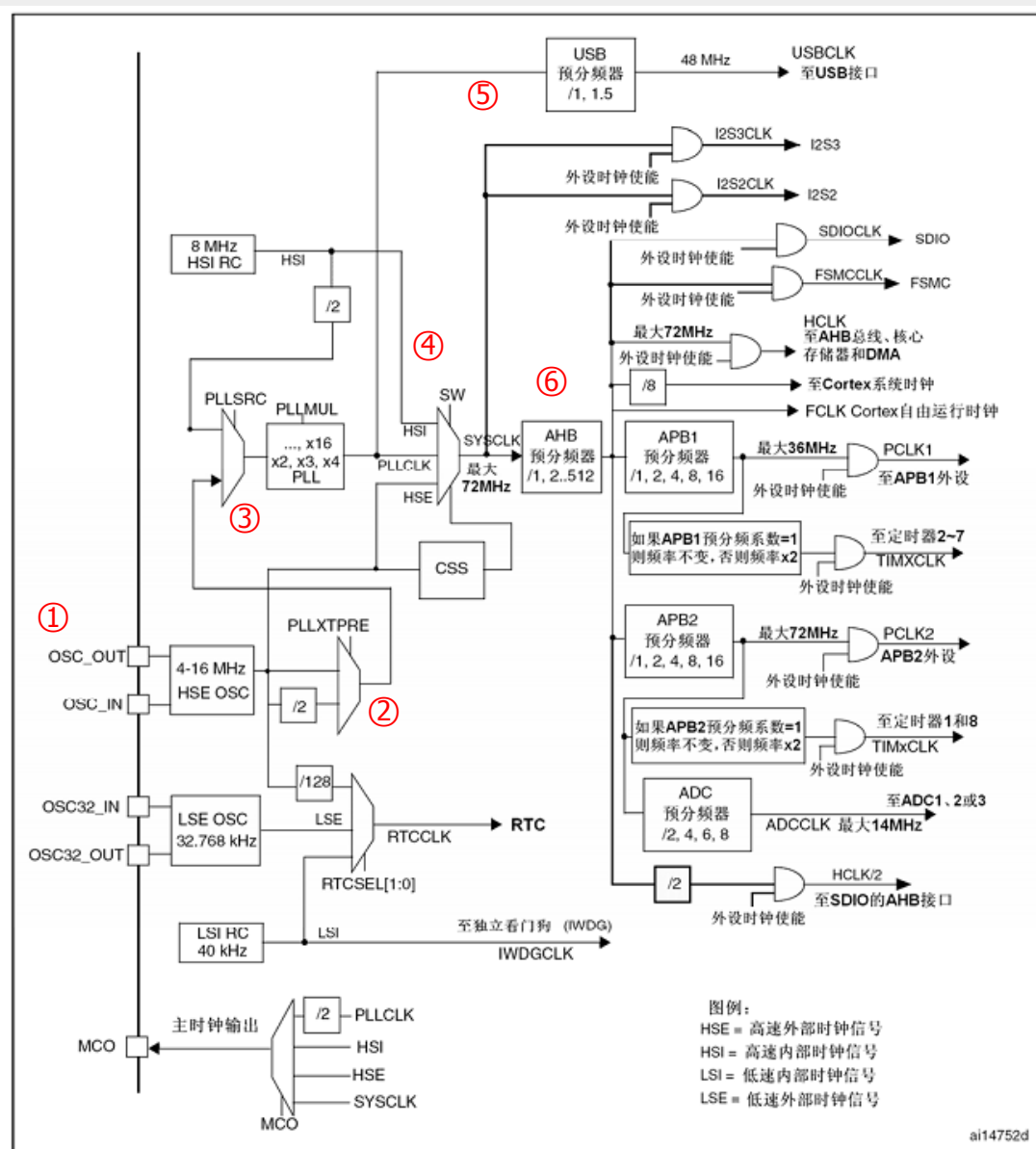
通过类似的方式，我们就可以给具体的寄存器写上适当的参数，控制 STM32 了。是不是觉得很巧妙？但这只是库开发的皮毛，而且实际上我们并不是这样使用库的，库为我们提供了更简单的开发方式。M3 的库可谓尽情绽放了 c 的魅力，如果你是单片机初学者，c 语言初学者，那么请你不要放弃与 M3 库邂逅的机会。是否选择库，就差你一个闪亮的回眸。

#### 4.4. STM32 的时钟系统

STM32 芯片为了实现低功耗，设计了一个功能完善但却非常复杂的时钟系统。普通的 MCU，一般只要配置好 GPIO 的寄存器，就可以使用了，但 STM32 还有一个步骤，就是开启外设时钟。

##### 4.4.1. 时钟树&时钟源

首先，从整体上了解 STM32 的时钟系统。



这个图说明了 STM32 的时钟走向，从图的左边开始，从时钟源一步步分配到外设时钟。

从时钟频率来说，又分为 **高速时钟** 和 **低速时钟**，高速时钟是提供给芯片主体的主时钟，而低速时钟只是提供给芯片中的 RTC（实时时钟）及独立看门狗使用。

从芯片角度来说，时钟源分为 **内部时钟** 与 **外部时钟源**，内部时钟是在芯片内部 RC 振荡器产生的，起振较快，所以时钟在芯片刚上电的时候，默认使用内部高速时钟。而外部时钟信号是由外部的晶振输入的，在精度和稳定性上都有很大优势，所以上电之后我们再通过软件配置，转而采用外部时钟信号。

所以，STM32 有以下 4 个时钟源：

高速外部时钟（HSE）：以外部晶振作时钟源，晶振频率可取范围为 4~16MHz，我们一般采用 8MHz 的晶振。

高速内部时钟（HSI）：由内部 RC 振荡器产生，频率为 8MHz，但不稳定。

低速外部时钟（LSE）：以外部晶振作时钟源，主要提供给实时时钟模块，所以一般采用 32.768KHz。野火 M3 实验板上用的是 32.768KHz，6p 负载规格的晶振。

低速内部时钟（LSI）：由内部 RC 振荡器产生，也主要提供给实时时钟模块，频率大约为 40KHz。

#### 4.4.2. 高速外部时钟（HSE）

我们以最常用的高速外部时钟为例分析，首先假定我们在外部提供的晶振的频率为 8MHz 的。

- 1、 从左端的 OSC\_OUT 和 OSC\_IN 开始，这两个引脚分别接到外部晶振的两端。
- 2、 8MHz 的时钟遇到了第一个分频器 **PLLXTPRE**（HSE divider for PLL entry），在这个分频器中，可以通过寄存器配置，选择它的输出。它的输出时钟可以是对输入时钟的二分频或不分频。本例子中，我们选择不分频，所以经过 PLLXTPRE 后，还是 8MHz 的时钟。
- 3、 8MHz 的时钟遇到开关 **PLLSRC**（PLL entry clock source），我们可以选择其输出，输出为外部高速时钟（HSE）或是内部高速时钟（HSI）。这里选择输出为 HSE，接着遇到锁相环 **PLL**，具有倍频作用，在这里我们可以输入倍频因子 **PLLMUL**（PLL multiplication factor），哥们，你要是想超频，就得在这个寄存器上做手脚啦。经过 PLL 的时钟称为 **PLLCLK**。倍频因子我们设定为 9 倍频，也就是说，经过 PLL 之后，我们的时钟从原来 8MHz 的 HSE 变为 72MHz 的 PLLCLK。

- 4、紧接着又遇到了一个**开关 SW**，经过这个开关之后就是 STM32 的**系统时钟 (SYSCLK)**了。通过这个开关，可以切换 SYSCLK 的时钟源，可以选择为 HSI、PLLCLK、HSE。我们选择为 PLLCLK 时钟，所以 SYSCLK 就为 72MHz 了。
- 5、PLLCLK 在输入到 SW 前，还流向了 USB 预分频器，这个分频器输出为 USB 外设的时钟 (USBCLK)。
- 6、回到 SYSCLK，SYSCLK 经过**AHB 预分频器**，分频后再输入到其它外设。如输出到称为 HCLK、FCLK 的时钟，还直接输出到 SDIO 外设的 SDIOCLK 时钟、存储器控制器 FSMC 的 FSMCCLK 时钟，和作为 APB1、APB2 的预分频器的输入端。本例子设置 AHB 预分频器不分频，即输出的频率为 72MHz。
- 7、**GPIO 外设**是挂载在 APB2 总线上的，APB2 的时钟是**APB2 预分频器**的输出，而 APB2 预分频器的时钟来源是**AHB 预分频器**。因此，把 APB2 预分频器设置为不分频，那么我们就可以得到 GPIO 外设的时钟也等于 HCLK，为 72MHz 了。

#### 4.4.3. HCLK、FCLK、PCLK1、PCLK2

从时钟树的分析，看到经过一系列的倍频、分频后得到了几个与我们开发密切相关的时钟。

**SYSCLK**: 系统时钟，STM32 大部分器件的时钟来源。主要由 AHB 预分频器分配到各个部件。

**HCLK**: 由 AHB 预分频器直接输出得到，它是高速总线 AHB 的时钟信号，提供给存储器，DMA 及 cortex 内核，是 cortex 内核运行的时钟，**cpu 主频**就是这个信号，它的大小与 STM32 运算速度，数据存取速度密切相关。

**FCLK**: 同样由 AHB 预分频器输出得到，是内核的“自由运行时钟”。“自由”表现在它不来自时钟 HCLK，因此在**HCLK 时钟停止时 FCLK 也继续运行**。它的存在，可以保证在处理器休眠时，也能够采样和到中断和跟踪休眠事件，它与 HCLK 互相同步。

PCLK1: 外设时钟, 由 **APB1 预分频器** 输出得到, 最大频率为 **36MHz**, 提供给挂载在 APB1 总线上的外设。

PCLK2: 外设时钟, 由 **APB2 预分频器** 输出得到, 最大频率可为 **72MHz**, 提供给挂载在 APB2 总线上的外设。

为什么 STM32 的时钟系统如此复杂, 有倍频、分频及一系列的外设时钟的开关。需要倍频是考虑到 **电磁兼容性**, 如外部直接提供一个 72MHz 的晶振, 太高的振荡频率可能会给制作电路板带来一定的难度。分频是因为 STM32 既有高速外设又有低速外设, 各种外设的 **工作频率不尽相同**, 如同 pc 机上的南北桥, 把高速的和低速的设备分开来管理。最后, 每个外设都配备了外设时钟的开关, 当我们不使用某个外设时, 可以把这个外设时钟关闭, 从而 **降低 STM32 的整体功耗**。所以, 当我们使用外设时, 一定要记得开启外设的时钟啊, 亲。

#### 4.5. LED 具体代码分析

有了以上对 STM32 存储器映像, 时钟系统, 以及基本的库函数知识, 我们就可以分析 LED 例程的代码了, 不知现在你有没饱饱的感觉了, 如果还饿, 那继续。

##### 4.5.1. 实验描述及工程文件清单

|         |                                                                                                                                                                    |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | 该实验讲解了如何运用 ST 的库来操作 I/O 口, 使 I/O 口产生置位(1)和复位(0)信号, 从而来控制 LED 的亮灭。                                                                                                  |
| 硬件连接    | PC3 – LED1、PC4 – LED2、PC5 – LED3                                                                                                                                   |
| 用到的库文件  | <i>startup/start_stm32f10x_hd.c</i><br><i>CMSIS/core_cm3.c</i><br><i>CMSIS/system_stm32f10x.c</i><br><i>FWlib/stm32f10x_gpio.c</i><br><i>FWlib/stm32f10x_rcc.c</i> |
| 用户编写的文件 | <i>USER/main.c</i>                                                                                                                                                 |

|  |                                                                      |
|--|----------------------------------------------------------------------|
|  | <i>USER/stm32f10x_it.c</i><br><i>USER/led.c</i><br><i>USER/led.h</i> |
|--|----------------------------------------------------------------------|

#### 4.5.2. 配置工程环境

LED 实验中用到了 GPIO 和 RCC(用于设置外设时钟)这两个片上外设，所以在操作 I/O 之前我们需要把关于这两个外设的库文件添加到工程模板之中。它们分别为 *stm32f10x\_gpio.c* 和 *stm32f10x\_rcc.c* 文件。其中 *stm32f10x\_gpio.c* 用于操作 I/O，而 *stm32f10x\_rcc.c* 用于配置系统时钟和外设时钟，由于每个外设都要配置时钟，所以它是每个外设都需要用到的库文件。

在添加完这两个库文件之后立即编译的话会出错，因为每个外设库对应于一个 *stm32f10x\_xxx.c* 文件的同时还对应着一个 *stm32f10x\_xxx.h* 头文件，头文件包含了相应外设的 C 语言函数实现的声明，只有我们把相应的头文件也包含进工程才能够使用这些外设库。在库中有一个专门的文件 *stm32f10x\_conf.h* 来管理所有库的头文件，*stm32f10x\_conf.h* 源码如下：

```
1.  * Includes -----  
    */  
  
2.  /* Uncomment the line below to enable peripheral header file inclusion */  
3.  /* #include "stm32f10x_adc.h" */  
4.  /* #include "stm32f10x_bkp.h" */  
5.  /* #include "stm32f10x_can.h" */  
6.  /* #include "stm32f10x_crc.h" */  
7.  /* #include "stm32f10x_dac.h" */  
8.  /* #include "stm32f10x_dbgmcu.h" */  
9.  /* #include "stm32f10x_dma.h" */  
10. /* #include "stm32f10x_exti.h" */  
11. /* #include "stm32f10x_flash.h" */
```



```
12. /* #include "stm32f10x_fsmc.h" */
13. /* #include "stm32f10x_gpio.h" */
14. /* #include "stm32f10x_i2c.h" */
15. /* #include "stm32f10x_iwdg.h" */
16. /* #include "stm32f10x_pwr.h" */
17. /* #include "stm32f10x_rcc.h" */
18. /* #include "stm32f10x_rtc.h" */
19. /* #include "stm32f10x_sdio.h" */
20. /* #include "stm32f10x_spi.h" */
21. /* #include "stm32f10x_tim.h" */
22. /* #include "stm32f10x_usart.h" */
23. /* #include "stm32f10x_wwdg.h" */
24. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

这是没有修改过的代码，默认情况下所有外设的头文件包含都被注释掉了。当我们需要用到某个外设驱动时直接把相应的注释去掉即可，非常方便。如本 LED 实验中我们用到了 **RCC** 跟 **GPIO** 这两个外设，所以我们应取消其注释，使第 13、17 行的代码 **#include "stm32f10x\_gpio.h"**、**#include "stm32f10x\_rcc.h"** 这两个语句生效，修改后如下所示：

```
1. /* Includes -----
    */
2. /* Uncomment the line below to enable peripheral header file inclusion */
3. /* #include "stm32f10x_adc.h" */
4. /* #include "stm32f10x_bkp.h" */
5. /* #include "stm32f10x_can.h" */
6. /* #include "stm32f10x_crc.h" */
7. /* #include "stm32f10x_dac.h" */
8. /* #include "stm32f10x_dbgmcu.h" */
9. /* #include "stm32f10x_dma.h" */
10. /* #include "stm32f10x_exti.h" */
11. /* #include "stm32f10x_flash.h" */
12. /* #include "stm32f10x_fsmc.h" */
13. #include "stm32f10x_gpio.h"
14. /* #include "stm32f10x_i2c.h" */
```

```
15. /* #include "stm32f10x_iwdg.h" */
16. /* #include "stm32f10x_pwr.h" */
17. #include "stm32f10x_rcc.h"
18. /* #include "stm32f10x_rtc.h" */
19. /* #include "stm32f10x_sdio.h" */
20. /* #include "stm32f10x_spi.h" */
21. /* #include "stm32f10x_tim.h" */
22. /* #include "stm32f10x_usart.h" */
23. /* #include "stm32f10x_wwdg.h" */
24. /*#include "misc.h"*/ /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

到这里，我们就可以用库自带的函数来操作 I/O 口了，这时我们可以编译一下，会发现既没有 Warning 也没有 Error。

#### 4.5.3. 编写用户文件

前期工程环境设置完毕，接下来我们就可以专心编写自己的应用程序了。我们把应用程序放在 USER 这个文件夹下，这个文件夹下至少包含了 *main.c*、*stm32f10x\_it.c*、*xxx.c* 这三个源文件。其中 main 函数就位于 *main.c* 这个 c 文件中，main 函数只是用来测试我们的应用程序。*stm32f10x\_it.c* 为我们提供了 M3 所有中断函数的入口，默认情况下这些中断服务程序都为空，等到用到的时候需要用户自己编写。所以现在我们把 *stm32f10x\_it.c* 包含到 USER 这个目录可以了。

而 *xxx.c* 就是由用户编写的文件，*xxx* 是应用程序的名字，用户可自由命名。我们把应用程序的具体实现放在了这个文件之中，程序的实现和应用分开在不同的文件中，这样就实现了很好的封装性。本书的例程都严格遵从这个规则，每个外设的用户文件都由独立的源文件与头文件构成，这样可以更方便地实现代码重用了。

于是，我们在工程中新建两个文件，分别为 *led.c* 和 *led.h*，保存在 USER 目录下，并把 *led.c* 添加到工程之中。*led.c* 文件中输入代码如下：

```
1. /***** (C) COPYRIGHT 2012 WildFire Team *****/
```

```
2.  * 文件名   : led.c
3.  * 描述     : led 应用函数库
4.  * 实验平台: 野火 STM32 开发板
5.  * 硬件连接: -----
6.  *          |      PC3 - LED1      |
7.  *          |      PC4 - LED2      |
8.  *          |      PC5 - LED3      |
9.  *          -----
10. * 库版本   : ST3.5.0
11. * 作者     : wildfire team
12. * 论坛     : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
13. * 淘宝     : http://firestm32.taobao.com
14. *****/
15. #include "led.h"
16.
17. /*
18. * 函数名: LED_GPIO_Config
19. * 描述   : 配置 LED 用到的 I/O 口
20. * 输入   : 无
21. * 输出   : 无
22. */
23. void LED_GPIO_Config(void)
24. {
25.     /*定义一个 GPIO_InitTypeDef 类型的结构体*/
26.     GPIO_InitTypeDef GPIO_InitStructure;
27.
28.     /*开启 GPIOC 的外设时钟*/
29.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);
30.
31.     /*选择要控制的 GPIOC 引脚
32.     */
33.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5
34.     ;
35.     /*设置引脚模式为通用推挽输出*/
36.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
37.
38.     /*设置引脚速率为 50MHz */
39.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
40.
41.     /*调用库函数，初始化 GPIOC*/
42.     GPIO_Init(GPIOC, &GPIO_InitStructure);
43.
44.     /* 关闭所有 led 灯 */
45.     GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
46. }
47.
48. /***** (C) COPYRIGHT 2012 WildFire Team *****/END OF FILE*****/
```

在这个文件中，我们定义了一个函数 `LED_GPIO_Config()`，在这个函数里，实现了所有为点亮 led 的配置。

#### 4.5.4. 初始化结构体——GPIO\_InitTypeDef 类型

**LED\_GPIO\_Config()**函数中，在文件的第 26 行的代码：

**GPIO\_InitTypeDef GPIO\_InitStructure;** 这是利用库，定义了一个名为 **GPIO\_InitStructure** 的结构体，结构体类型为 **GPIO\_InitTypeDef**。

**GPIO\_InitTypeDef** 类型与前面介绍的库对寄存器的封装类似，是库文件利用关键字 **typedef** 定义的新类型。追踪其定义原型如下，位于

**stm32f10x\_gpio.h** 文件中：

```
1. typedef struct
2. {
3.     uint16_t GPIO_Pin;           /*指定将要进行配置的 GPIO 引脚*/
4.     GPIOSpeed_TypeDef GPIO_Speed; /*指定 GPIO 引脚可输出的最高频率*/
5.     GPIOMode_TypeDef GPIO_Mode;  /*指定 GPIO 引脚将要配置成的工作状态*/
6. }GPIO_InitTypeDef;
```

于是我们知道，**GPIO\_InitTypeDef** 类型的结构体有三个成员，分别为 **uint16\_t** 类型的 **GPIO\_Pin**，**GPIOSpeed\_TypeDef** 类型的 **GPIO\_Speed** 及 **GPIOMode\_TypeDef** 类型的 **GPIO\_Mode**。

**uint16\_t** 类型的 **GPIO\_Pin** 为我们将要选择配置的引脚，在 **stm32f10x\_gpio.h** 文件中有如下宏定义：

```
1. #define GPIO_Pin_0      ((uint16_t)0x0001) /*!< Pin 0 selected */
2. #define GPIO_Pin_1      ((uint16_t)0x0002) /*!< Pin 1 selected */
3. #define GPIO_Pin_2      ((uint16_t)0x0004) /*!< Pin 2 selected */
4. #define GPIO_Pin_3      ((uint16_t)0x0008) /*!< Pin 3 selected */
```

这些宏的值，就是允许我们给结构体成员 **GPIO\_Pin** 赋的值，如我们给 **GPIO\_Pin** 赋值为宏 **GPIO\_Pin\_0**，表示我们选择了 **GPIO** 端口的第 0 个引脚，在后面会通过一个函数把这些宏的值进行处理，设置相应的寄存器，实现我们对 **GPIO** 端口的配置。如 **led.c** 代码中的第 32 行，意义为我们将要选择 **GPIO** 的 **Pin3**、**Pin4**、**Pin5** 引脚进行配置。

**GPIOSpeed\_TypeDef** 和 **GPIOMode\_TypeDef** 又是两个库定义的新类型，**GPIOSpeed\_TypeDef** 原型如下：

```
1. typedef enum
2. {
3.     GPIO_Speed_10MHz = 1, //枚举常量, 值为1, 代表输出速率最高为10MHz
4.     GPIO_Speed_2MHz,      //对不赋值的枚举变量, 自动加1, 此常量值为2
5.     GPIO_Speed_50MHz      //常量值为3
6. }GPIOSpeed_TypeDef;
```

这是一个枚举类型, 定义了三个枚举常量, 即

***GPIO\_Speed\_10MHz=1, GPIO\_Speed\_2MHz=2,***

***GPIO\_Speed\_50MHz=3***。这些常量可用于标识 GPIO 引脚可以配置成的各个最高速度。所以我们在为结构体中的 ***GPIO\_Speed*** 赋值的时候, 就可以直接用这些含义清晰的**枚举标识符**了。如 led.c 代码中的第 38 行, 给

GPIO\_Speed 赋值为 3, 意义为使其最高频率可达到 50MHz。

同样, GPIO\_Mode\_TypeDef 也是一个枚举类型定义符, 原型如下:

```
1. typedef enum
2. { GPIO_Mode_AIN = 0x0,           //模拟输入模式
3.   GPIO_Mode_IN_FLOATING = 0x04, //浮空输入模式
4.   GPIO_Mode_IPD = 0x28,         //下拉输入模式
5.   GPIO_Mode_IPU = 0x48,         //上拉输入模式
6.   GPIO_Mode_Out_OD = 0x14,      //开漏输出模式
7.   GPIO_Mode_Out_PP = 0x10,      //通用推挽输出模式
8.   GPIO_Mode_AF_OD = 0x1C,       //复用功能开漏输出
9.   GPIO_Mode_AF_PP = 0x18        //复用功能推挽输出
10. }GPIO_Mode_TypeDef;
```

这个枚举类型也定义了很多含义清晰的枚举常量, 是用来帮助配置 GPIO 引脚的模式, 如 GPIO\_Mode\_AIN 意义为模拟输入、GPIO\_Mode\_IN\_FLOATING 为浮空输入模式。在 led.c 代码中的第 35 行意义为把引脚设置为通用推挽输出模式。

于是, 我们可以总结 GPIO\_InitTypeDef 类型结构体的作用, 整个结构体包含 ***GPIO\_Pin***、***GPIO\_Speed***、***GPIO\_Mode*** 三个成员, 我们对这三个成员赋予不同的数值可以对 GPIO 端口进行不同的配置, 而这些可配置的数值, 已经由 ST 的库文件封装成见名知义的枚举常量。这使我们编写代码变得非常简便。

#### 4.5.5. 初始化库函数——GPIO\_Init()

在前面我们已经接触到 ST 的库文件，以及各种各样由 ST 库定义的新类型，但所有的这些，都只是为库函数服务的。在 led.c 文件的第 41 行，我们用到了第一个用于初始化的库函数 GPIO\_Init()。

在我们应用库函数的时候，只需要知道它的功能及输入什么类型的参数，允许的参数值就足够了，这些我们都可以能通过查找库帮助文档获得，详细方法见 3.2.4 使用库帮助文档小节。查询结果见图 4-10。

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

Initializes the GPIOx peripheral according to the specified parameters in the GPIO\_InitStruct.

**Parameters:**

- GPIOx,:** where x can be (A..G) to select the GPIO peripheral.  
可输入参数为 **GPIOA~GPIOG**
- GPIO\_InitStruct,:** pointer to a GPIO\_InitTypeDef structure that contains the configuration information for the specified GPIO peripheral.

**Return values:**

- None**  
可输入的为 **GPIO\_InitStruct** 结构体指针型参数，  
点击带下划线字体可弹出详细说明

Definition at line 173 of file stm32f10x\_gpio.c.

图 4-10 GPIO\_Init 函数

这个函数有两个输入参数，分别为 **GPIO\_TypeDef** 和 **GPIO\_InitTypeDef** 型的指针。其允许值为 GPIOA……GPIOG，和 **GPIO\_InitTypeDef** 型指针变量。

在调用的时候，如 led.c 文件的第 41 行，  
**GPIO\_Init(GPIOC, &GPIO\_InitStructure)**，第一个参数，说明它将要为 GPIOC 端口进行初始化。初始化的配置以第二个参数 GPIO\_InitStructure 结构体的成员值为准。这个结构体的成员，我们在调用 **GPIO\_Init()** 前，已对它们赋予了控制参数。

```
31.     /*选择要控制的 GPIOC 引脚  
    */  
32.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5  
    ;  
33.
```

```
34.  /*设置引脚模式为通用推挽输出*/
35.  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
36.
37.  /*设置引脚速率为 50MHz */
38.  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
39.
```

于是，在调用 `GPIO_Init()` 函数后，GPIOC 的 Pin3、Pin4、Pin5 就被配置成了最高频率为 50MHz 的通用推挽输出模式了。

在这个函数的内部，实现了把输入的这些参数按照一定的规则转化，进而写入寄存器，实现了配置 GPIO 端口的功能。函数的实现将在 4.6 小节进行详细分析。

#### 4.5.6. 开启外设时钟

调用了 `GPIO_Init()` 函数之后，对 GPIO 的初始化也就基本完成了，那还缺少什么呢？就是在前面强调过的必须要开启外设时钟，在开启外设时钟之前，我们首先要配置好系统时钟 `SYSCLK`，4.4 小节提到，为配置 `SYSCLK`，要设置一系列的时钟来源、倍频、分频等控制参数。这些工作由 `SystemInit()` 库函数完成。

#### ■ 启动文件及 `SystemInit()` 函数分析

在 `startup_stm32f10x_hd.s` 启动文件中，有如下一段启动代码：

```
1. ;Reset_Handler 子程序开始
2. Reset_Handler PROC
3.
4. ;输出子程序 Reset_Handler 到外部文件
5. EXPORT Reset_Handler [WEAK]
6.
7. ;从外部文件中引入 main 函数
8. IMPORT __main
9.
10. ;从外部文件引入 SystemInit 函数
11. IMPORT SystemInit
12.
13. ;把 SystemInit 函数调用地址加载到通用寄存器 r0
14. LDR R0, =SystemInit
15.
16. ;跳转到 r0 中保存的地址执行程序（调用 SystemInit 函数）
17. BLX R0
18.
19. ;把 main 函数调用地址加载到通用寄存器 r0
```



```
20.                LDR        R0, =__main
21.
22. ;跳转到 r0 中保存的地址执行程序（调用 main 函数）
23.                BX         R0
24.
25. ;Reset_Handler 子程序结束
26.                ENDP
```

注：这是一段汇编代码，对汇编比较陌生的读者请配以“；”后面的注释来阅读，“；”表示注释其后的单行代码，相当于c语言中的“//”和“/\* \*/”。

当芯片被复位(包括上电复位)的时候，将开始运行这一段代码，运行过程为先调用了 **SystemInit()** 函数，再进入c语言中的 **main** 函数执行。读者是否曾思考过？为什么c语言程序都从 **main** 函数开始执行？就是因为我们的启动文件中有了这一段代码，可以尝试一下把第8行引入 **main** 函数，及第20行的加载 **main** 函数的标识符修改掉，看其效果。如改成：

```
IMPORT __wildfire
.....

LDR    R0,=__wildfire
```

这样修改以后，内核就会从 **wildfire()** 函数中开始执行第一个c语言的代码啦。有些比较狡猾的朋友就会这么干，让人家看他的代码时找不到 **main** 函数，何其险恶呀:))。

但是，前面强调了，进入 **main** 函数之前调用了一个名为 **SystemInit()** 的函数。这个函数的定义在 **system\_stm32f10x.c** 文件之中。它的作用是设置系统时钟 **SYSCLK**。函数的执行流程是先将与配置时钟相关的寄存器都复位为默认值，复位寄存器后，调用了另外一个函数 **SetSysClock()**，**SetSysClock()** 代码如下：

```
1. static void SetSysClock(void)
2. {
3.     #ifdef SYSCLK_FREQ_HSE
4.         SetSysClockToHSE();
5.     #elif defined SYSCLK_FREQ_24MHz
6.         SetSysClockTo24();
7.     #elif defined SYSCLK_FREQ_36MHz
8.         SetSysClockTo36();
9.     #elif defined SYSCLK_FREQ_48MHz
10.        SetSysClockTo48();
11.    #elif defined SYSCLK_FREQ_56MHz
12.        SetSysClockTo56();
```

```
13. #elif defined SYSCLK_FREQ_72MHz
14.     SetSysClockTo72();
15. #endif
16.
17. /* If none of the define above is enabled, the HSI is used as System
    clock
18.     source (default after reset) */
19. }
```

从 **SetSysClock()** 代码可以知道，它是根据我们设置的条件编译宏来进行不同的时钟配置的。

在 **system\_stm32f10x.c** 文件的开头，已经默认有了如下的条件编译定义：

```
1. #if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined
   STM32F10X_HD_VL)
2. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
3. #define SYSCLK_FREQ_24MHz  24000000
4. #else
5. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
6. /* #define SYSCLK_FREQ_24MHz  24000000 */
7. /* #define SYSCLK_FREQ_36MHz  36000000 */
8. /* #define SYSCLK_FREQ_48MHz  48000000 */
9. /* #define SYSCLK_FREQ_56MHz  56000000 */
10. #define SYSCLK_FREQ_72MHz  72000000
11. #endif
```

在第 10 行定义了 **SYSCLK\_FREQ\_72MHz** 条件编译的标识符，所以在 **SetSysClock()** 函数中将调用 **SetSysClockTo72()** 函数把芯片的系统时钟 **SYSCLK** 设置为 **72MHz** 当然，前提是输入的外部时钟源 HSE 的振荡频率要为 **8MHz**。

其中的 **SetSysClockTo72()** 函数就是最底层的库函数了，那些跟寄存器打交道的活都是由它来完成的，如果大家想知道我们的系统时钟是如何配置成 72M 的话，可以研究这个函数的源码。但大可不必这样，我们应该抛开传统的直接跟寄存器打交道来学单片机的方法，而是直接用 ST 的库给我们提供的上层接口，这样会简化我们很多的工作，还能提高我们开发产品的效率，何乐而不为呢？对这一类直接跟寄存器打交道的函数分析在 4.6 小节以 **GPIO\_Init()** 函数为例来分析。

注意：3.5 版本的库在启动文件中调用了 **SystemInit()**，所以不必在 **main()** 函数中再次调用。但如果使用的是 3.0 版本的库则必须在 **main** 函数中调用 **SystemInit()**，以设置系统时钟，因为在 3.0 版本的启动代码中并没有调用 **SystemInit()** 函数。

## ■ 开启外设时钟

SYSCCLK 由 `SystemInit()`配置好了，而 GPIO 所用的时钟 `PCLK2` 我们采用默认值，也为 72MHz。我们采用默认值可以不修改分频器，但外设时钟默认是处在关闭状态的。所以外设时钟一般会在初始化外设的时候设置为开启(根据设计的产品功耗要求，也可以在使用的时候才打开)。开启和关闭外设时钟也有封装好的库函数 `RCC_APB2PeriphClockCmd()`。在 `led.c` 文件中的第 29 行，我们调用了这个函数。

查看其使用手册见图 4-11

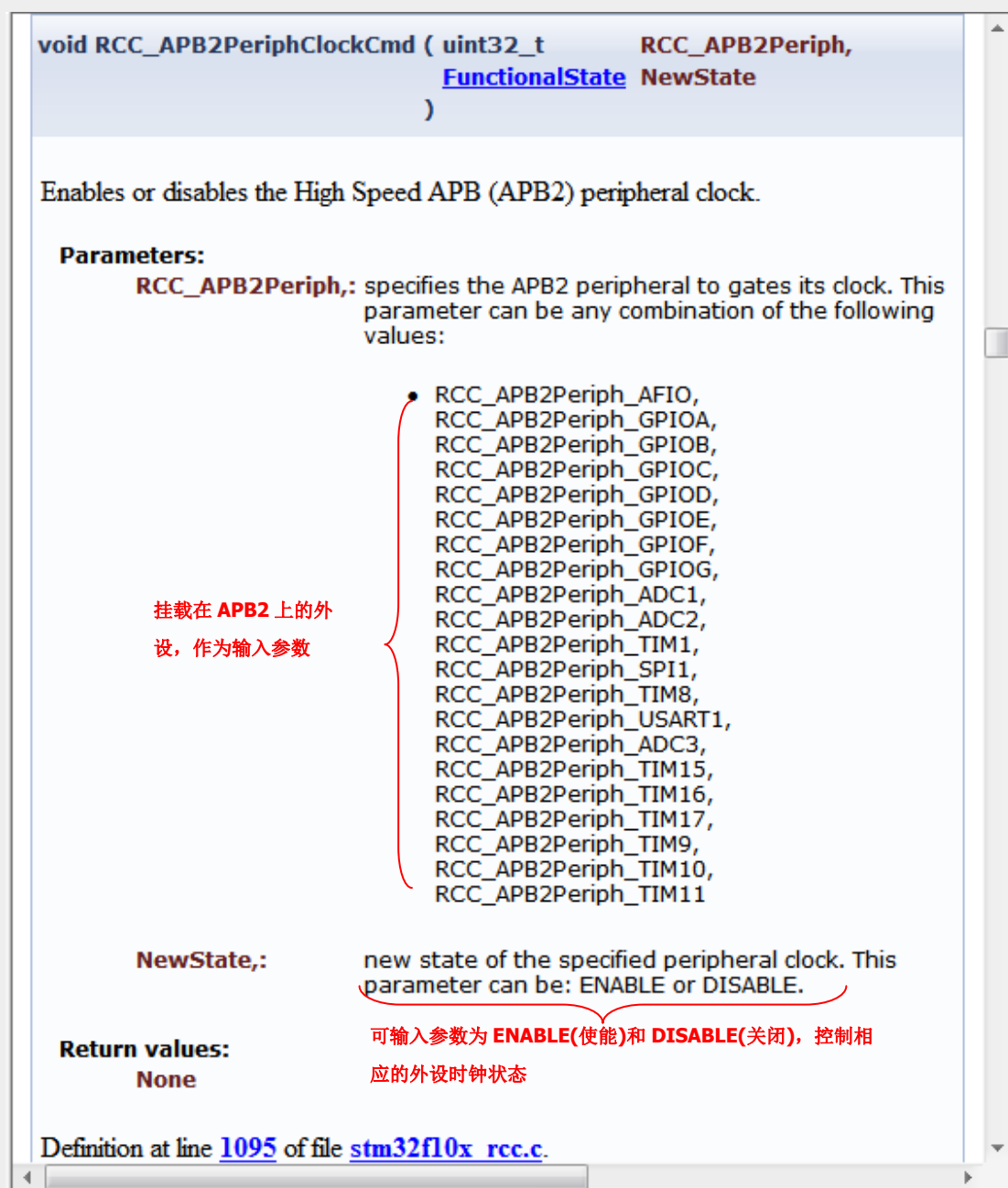


图 4-11 APB2 时钟使能函数

调用的时候需要向它输入两个参数，一个参数为将要控制的，挂载在 APB2 总线上的外设时钟，第二个参数为选择要开启还是关闭该时钟。

led.c 文件中对它的调用：

**RCC\_APB2PeriphClockCmd( RCC\_APB2Periph\_GPIOC, ENABLE);**

就表示将要 **ENABLE(使能)**GPIOC 外设时钟。

在这里强调一点，如果我们用到了 I/O 的引脚**复用功能**，还要开启其**复用功能时钟**。

如 GPIOC 的 Pin4 还可以作为 ADC1 的输入引脚，现在我们把它作为 ADC1 来使用，除了开启 GPIOC 时钟外，还要开启 ADC1 的时钟：

```
RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);  
RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC1, ENABLE);
```

我们知道有的外设是挂载在高速外设总线 APB2 上使用 PCLK2 时钟，还有的是挂载在低速外设总线 APB1 上，使用 PCLK1 时钟。既然时钟源是不同的，当然也就有另一个函数来开启 APB1 总线外设的时钟：

**RCC\_APB1PeriphClockCmd()**函数，这两个函数名，正是根据其挂载在的总线命名的。可输入的参数自然也就不一样，使用的时候要注意区分。其中所有的 GPIO 都是挂载在 APB2 上的。

#### 4.5.7. 控制 I/O 输出高、低电平

前面我们选择好了引脚，配置了其功能及开启了相应的时钟，我们可以终于可以正式控制 I/O 口的电平高低了，从而实现控制 LED 灯的亮与灭。

前面提到过，要控制 GPIO 引脚的电平高低，只要在 GPIOx\_BSRR 寄存器相应的位写入控制参数就可以了。ST 库也为我们提供了具有这样功能的函数，可以分别是用 **GPIO\_SetBits()**控制输出高电平，和用 **GPIO\_ResetBits()**控制输出低电平。见图 4-12 及图 4-13

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

Sets the selected data port bits.

**Parameters:**

**GPIOx,:** where x can be (A..G) to select the GPIO peripheral.  
**GPIO\_Pin,:** specifies the port bits to be written. This parameter can be any combination of GPIO\_Pin\_x where x can be (0..15).

**Return values:**

None

图 4-12 GPIO 引脚置 1 函数

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

Clears the selected data port bits.

**Parameters:**

**GPIOx,:** where x can be (A..G) to select the GPIO peripheral.  
**GPIO\_Pin,:** specifies the port bits to be written. This parameter can be any combination of GPIO\_Pin\_x where x can be (0..15).

**Return values:**

None

图 4-13 GPIO 引脚清零函数

输入参数有两个，第一个为将要控制的 GPIO 端口：GPIOA……GPIOG，第二个为要控制的引脚号：Pin0~Pin15。

在 led.c 文件的第 44 行，**LED\_GPIO\_Config()**函数中，我们在调用 GPIO\_Init()函数之后就调用了 **GPIO\_SetBits()**函数，从而让这几个引脚输出高电平，使三盏 LED 初始化后都处于灭状态。

#### 4.5.8. led.h 文件

接下来，分析 led.h 文件。其内容如下

```
1. #ifndef __LED_H
2. #define __LED_H
3.
4. #include "stm32f10x.h"
5.
6. /* the macro definition to trigger the led on or off
7.  * 1 - off
8.  * 0 - on
9.  */
10. #define ON 0
11. #define OFF 1
12.
13. //带参宏，可以像内联函数一样使用
14. #define LED1(a) if (a) \
15.                 GPIO_SetBits(GPIOC,GPIO_Pin_3);\
16.                 else \
17.                 GPIO_ResetBits(GPIOC,GPIO_Pin_3)
18.
19. #define LED2(a) if (a) \
20.                 GPIO_SetBits(GPIOC,GPIO_Pin_4);\
21.                 else \
22.                 GPIO_ResetBits(GPIOC,GPIO_Pin_4)
23.
24. #define LED3(a) if (a) \
25.                 GPIO_SetBits(GPIOC,GPIO_Pin_5);\
26.                 else \
27.                 GPIO_ResetBits(GPIOC,GPIO_Pin_5)
28.
29. void LED_GPIO_Config(void);
30.
31. #endif /* __LED_H */
```

这个头文件的内容不多，但也把它独立成一个头文件，方便以后扩展或移植使用。希望读者养成良好的工程习惯，在写头文件的时候，加上类似以下这样的条件编译。

```
#ifndef __LED_H
#define __LED_H
.....
#endif
```

这样可以防止头文件重复包含，使得工程的兼容性更好。读者问为什么要加两个下划线“\_\_”？在这里加两个下划线可以避免这个宏标识符与其它定义重名，因为在其它部分代码定义的宏或变量，一般都不会出现这样有下划线的名字。



在 `led.h` 头文件的部分，首先包含了前面提到的最重要的 ST 库必备头文件 `stm32f10x.h`。有了它我们才可以使用各种库定义、库函数。

在 `led.h` 文件的第 14~27 行，是我们利用 `GPIO_SetBits()`、`GPIO_ResetBits()` 库函数编写的带参宏定义，带参宏与 C++ 中的内联函数作用很类似。在编译过程，编译器会把带参宏展开，在相应的位置替换为宏展开代码。其中的反斜杠符号 “\” 叫做续行符，用来连接上下行代码，表示下面一行代码属于 “\” 所在的代码行，这在 ST 库经常出现。“\” 的语法要求极其严格，在它的后面不能有空格、注释等一切“杂物”，在论坛上经常有读者反映遇到编译错误，却不知道正是错在这里。群里很多朋友都问到 “\” 是个什么东西，那野火可要打你 pp 了，你这是 c 语言不及格呀，亲。

最后，在 `led.h` 文件中的第 29 行代码，声明了我们在 `led.c` 源文件定义的 `LED_GPIO_Config()` 用户函数。因此，我们要使用 `led.c` 文件定义的函数时，只要把 `led.h` 包含到调用到函数的文件中就可以了。

#### 4.5.9. main 文件

写好了 `led.c`、`led.h` 两个文件，我们控制 LED 灯的驱动程序就全部完成了。接下来，就可以利用写好的驱动文件，在 `main` 文件中编写应用程序代码了。本 LED 例程的 `main` 文件内容如下：

```
1.  /***** (C) COPYRIGHT 2012 WildFire Team *****/
2.  * 文件名   : main.c
3.  * 描述     : LED 流水灯，频率可调.....
4.  * 实验平台 : 野火 STM32 开发板
5.  * 库版本   : ST3.5.0
6.  *
7.  * 作者     : wildfire team
8.  * 论坛     : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
9.  * 淘宝     : http://firestm32.taobao.com
10. *****/
11. #include "stm32f10x.h"
12. #include "led.h"
13.
14. void Delay(__IO u32 nCount);
15.
16. /*
17.  * 函数名: main
18.  * 描述   : 主函数
19.  * 输入   : 无
20.  * 输出   : 无
```

```
21. */
22. int main(void)
23. {
24.     /* LED 端口初始化 */
25.     LED_GPIO_Config();
26.
27.     while (1)
28.     {
29.         LED1( ON );           // 亮
30.         Delay(0xFFFFEF);
31.         LED1( OFF );          // 灭
32.
33.         LED2( ON );
34.         Delay(0xFFFFEF);
35.         LED2( OFF );
36.
37.         LED3( ON );
38.         Delay(0xFFFFEF);
39.         LED3( OFF );
40.     }
41. }
42.
43. void Delay(__IO u32 nCount) //简单的延时函数
44. {
45.     for(; nCount != 0; nCount--);
46. }
47.
48.
49. /***** (C) COPYRIGHT 2012 WildFire Team *****/END OF FILE*****/
```

main 文件的开头部分首先包含所需的头文件，*stm32f10x.h* 和 *led.h*。

在第 14 行还声明了一个简单的延时函数，其定义在 main 文件的末尾。它是利用 for 循环实现的，用作短暂的，对精度要求不高的延时，延时的时间与输入的参数并无准确的计算公式，请不要深究。需要精准的延时的时候，我们会采用定时器来精确控制。

在芯片上电(复位)后，经过启动文件中 *SystemInit()* 函数配置好了时钟，就进入 main 函数了。接下来，从 main 函数开始分析代码的执行。

首先，调用了在 led.c 文件编写好的 *LED\_GPIO\_Config()* 函数，完成了对 GPIOC 的 Pin3、Pin4、Pin5 的初始化。紧接着就在 while 死循环里不断执行在 led.h 文件中编写的带参宏代码，并加上延时函数，使各盏 LED 轮流亮灭。当然，在 LED 控制的部分，如果不习惯带参宏的方式，读者也可以直接使用 *GPIO\_SetBits()* 和 *GPIO\_ResetBits()* 函数实现对 LED 的控制。

如果使用的是 3.0 版本的库，由于启动文件中没有调用 *SystemInit()* 函数，所以要在初始化 GPIO 等外设之前，也就是在 main 函数的第 1 行代码，就调用 *SystemInit()* 函数，以完成对系统时钟的配置。

到此，我们整个控制 LED 灯的工程的讲解就完成了。

#### 4.5.10. 实验现象

将程序烧写到野火 STM32 开发板中，即可看到 3 个 LED 一定的频率闪烁。

### 4.6. GPIO\_Init()函数的实现

在我们控制 LED 灯的工程中，调用了很多库函数，有 *SystemInit()*、*GPIO\_Init()*、*GPIO\_SetBits()*、*GPIO\_ResetBits()*等等。虽说为了开发速度，我们只管函数的功能和如何调用就行了，但免不了有种不踏实的感觉。

所以在本小节以 *GPIO\_Init()*函数实现的分析为例，可以帮助读者理解 ST 库的本质，让读者在使用库开发的时候心里更有底。

#### 4.6.1. 规范的位操作方法

由于库函数的实现涉及到不少位操作，首先为读者介绍一下几个常用的位操作方法，排除阅读代码的障碍。

1、将 char 型变量 a 的第七位(bit6)清 0，其它位不变。

```
1、 a &= ~(1<<6);    //括号内 1 左移 6 位，得二进制数：0100 0000
2、                //按位取反，得 1011 1111，所得的数与 a 作“位与&”运算，
3、                // a 的第 7 位 (bit6) 被置零，而其它位不变。
```

2、同理，将变量 a 的第七位(bit6)置 1，其它位不变的方法如下。

```
1、 a |= (1<<6);      //把第七位 (bit6) 置 1，其它为不变
```

3、将变量 a 的第七位(bit6)取反，其它位不变。

```
1、 a ^= (1<<6);      //把第七位 (bit6) 取反，其它位不变
```

#### 4.6.2. GPIO\_Init()实现代码分析

有了上面的位操作知识准备后，就可以分析 GPIO\_Init()函数的定义代码了。

```
1. void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
2. {
3.     uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
4.     uint32_t tmpreg = 0x00, pinmask = 0x00;
5.     /* 断言，用于检查输入的参数是否正确 */
6.     assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
7.     assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
8.     assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
9.
10.    /*----- GPIO 的模式配置 -----*/
11.    /*把输入参数 GPIO_Mode 的低四位暂存在 currentmode*/
12.    currentmode = ((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x0F);
13.    /*判断是否为输出模式，输出模式，可输入参数中输出模式的 bit4 位都是 1*/
14.    if (((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
15.    {
16.        /* 检查输入参数 */
17.        assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
18.        /* 输出模式，所以要配置 GPIO 的速率:00(输入模式) 01(10MHz) 10(2MHz) 11 */
19.        currentmode |= (uint32_t)GPIO_InitStruct->GPIO_Speed;
20.    }
21.    /*-----配置 GPIO 的 CRL 寄存器 -----*/
22.    /* 判断要配置的是否为 pin0 ~~ pin7 */
23.    if (((uint32_t)GPIO_InitStruct->GPIO_Pin & ((uint32_t)0x00FF)) != 0x00)
24.    {
25.        /*备份原 CRL 寄存器的值*/
26.        tmpreg = GPIOx->CRL;
27.        /*循环，一个循环设置一个寄存器位*/
28.        for (pinpos = 0x00; pinpos < 0x08; pinpos++)
29.        {
30.            /*pos 的值为 1 左移 pinpos 位*/
31.            pos = ((uint32_t)0x01) << pinpos;
32.            /* 令 pos 与输入参数 GPIO_PIN 作位与运算，为下面的判断作准备 */
33.            currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
34.            /*判断，若 currentpin=pos,说明 GPIO_PIN 参数中含的第 pos 个引脚需要配置*/
35.            if (currentpin == pos)
36.            {
37.                /*pos 的值左移两位（乘以 4），因为寄存器中 4 个寄存器位配置一个引脚*/
38.                pos = pinpos << 2;
39.                /*以下两个句子，把控制这个引脚的 4 个寄存器位清零，其它寄存器位不变*/
40.                pinmask = ((uint32_t)0x0F) << pos;
41.                tmpreg &= ~pinmask;
42.                /* 向寄存器写入将要配置的引脚的模式 */
43.                tmpreg |= (currentmode << pos);
44.                /* 复位 GPIO 引脚的输入输出默认值 */
45.            }
46.            /*判断是否为下拉输入模式*/
47.            if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
48.            {
49.                /*下拉输入模式，引脚默认置 0，对 BRR 寄存器写 1 可对引脚置 0*/
50.            }
51.        }
52.    }
53.    /* 对 BRR 寄存器写 1 可对引脚置 0 */
54.    GPIOx->BRR = (uint32_t)currentpin;
55.}
```

```

49.         GPIOx->BRR = (((uint32_t)0x01) << pinpos);
50.     }
51.     else
52.     {
53.         /*判断是否为上拉输入模式*/
54.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
55.         {
56.             /*上拉输入模式，引脚默认值为 1，对 BSRR 寄存器写 1 可对引脚置 1*/
57.             GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
58.         }
59.     }
60. }
61. }
62. /*把前面处理后的暂存值写入到 CRL 寄存器之中*/
63.     GPIOx->CRL = tmpreg;
64. }
65. /*----- 以下部分是对 CRH 寄存器配置的 -----

66. -----当要配置的引脚为 pin8 ~~ pin15 的时候，配置 CRH 寄存器，-----
67. -----这过程和配置 CRL 寄存器类似-----
68. -----读者可自行分析，看看自己是否了解了上述过程--^_^-----*/
69. /* Configure the eight high port pins */
70. if (GPIO_InitStruct->GPIO_Pin > 0x00FF)
71. {
72.     tmpreg = GPIOx->CRH;
73.     for (pinpos = 0x00; pinpos < 0x08; pinpos++)
74.     {
75.         pos = (((uint32_t)0x01) << (pinpos + 0x08));
76.         /* Get the port pins position */
77.         currentpin = ((GPIO_InitStruct->GPIO_Pin) & pos);
78.         if (currentpin == pos)
79.         {
80.             pos = pinpos << 2;
81.             /* Clear the corresponding high control register bits */
82.             pinmask = ((uint32_t)0x0F) << pos;
83.             tmpreg &= ~pinmask;
84.             /* Write the mode configuration in the corresponding bits */
85.             tmpreg |= (currentmode << pos);
86.             /* Reset the corresponding ODR bit */
87.             if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
88.             {
89.                 GPIOx->BRR = (((uint32_t)0x01) << (pinpos + 0x08));
90.             }
91.             /* Set the corresponding ODR bit */
92.             if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
93.             {
94.                 GPIOx->BSRR = (((uint32_t)0x01) << (pinpos + 0x08));
95.             }
96.         }
97.     }
98.     GPIOx->CRH = tmpreg;
99. }
100. }

```

这部分代码比较长，请读者配合代码中的注释，《STM32 中文参考手册》中的 CRL 寄存器的说明图，来理解这个函数。

### 8.2.1 端口配置低寄存器(GPIOx\_CRL) (x=A..E)

偏移地址: 0x00

复位值: 0x4444 4444

每 4 个寄存器位配置一个引脚 这 4 位控制 pin4

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

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

CNFy: 向这两个位写入不同的值, 设置引脚为不同的功能。y 表示第 y 个引脚。

MODEy: 向这两个位写入不同的值, 设置引脚为不同的最大输出速率, 或设置为输入模式。

图 4-14 GPIOx\_CRL 寄存器



图 4-15 GPIO\_Init 分析



以我们 *led.c* 文件中对 `GPIO_Init()` 函数的调用为例。在调用函数前：

1、 *led.c* 代码的 32 行，对 `GPIO_InitStructure.GPIO_Pin` 结构体成员赋值为 `GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5`，宏展开为 `(0000 0000 0011 1000)B`，表明我们将要对这三个引脚进行配置。

2、 第 35 行，对 `.GPIO_Mode` 赋值为 `GPIO_Mode_Out_PP`，宏展开为 `(0001 0100)B`，表明我们要把这三个引脚都设置为通用推挽模式。

3、 第 38 行，对 `.GPIO_Speed` 赋值为 `GPIO_Speed_50MHz`，宏展开为 `(0011)B`，表明我们设置这三个引脚的输出最大速度都为 50MHz。

*led.c* 的第 41 行调用 `GPIO_Init()` 的时候，就把 GPIOC 和上面这三个参数输入到函数了，经过这个函数处理，最终它向 GPIOC 组的 CRL 配置寄存器写入了一个值：

```
1. GPIOC->CRL = 0x44333444;  
2. //二进制表示为(0100 0100 0011 0011 0011 0100 0100 0100)
```

把这个值化为二进制为：`(0100 0100 0011 0011 0011 0100 0100 0100)B`；这个值的每 4 个二进制位代表一组引脚的控制值。Pin3、Pin4、Pin5 的控制值都是 `(0011)B`，有心的读者可以对比一下 `CRL` 寄存器的说明，这些控制值正好可以把 GPIO 设置为符合我们输入参数要求的状态，为最大速率为 50MHz 的通用推挽输出模式。

#### 4.6.3. 再论开发方式

了解库函数的实现后，我们现在就可以用实例来分析使用库函数与直接配置寄存器的区别了。

用直接配置寄存器的方法，只需要一个语句：

```
1. GPIOC->CRL = 0x44333444;
```

这样直接向寄存器赋值就完成了，以这样的方式配置是内核执行效率最高的方式，内核的工作是简单了，但我们为实现所需的配置，确定这样的值，却是一件麻烦事，工程量大的时候，缺点就显而易见了。

配置寄存器还可以用一些相对缓和的方法，前面提到的三种[位操作方式](#)。如：

```
1. GPIOC->CRL &=~(uint32_t)(1111<<4*3); //清空 Pin3 的 4 个控制位
2. GPIOC->CRL |=(uint32_t)(0011<<4*3); //配置 Pin3 的 4 个控制位
3. GPIOC->CRL &=~(uint32_t)(1111<<4*4); //清空 Pin4 的 4 个控制位
4. GPIOC->CRL |=(uint32_t)(0011<<4*4); //配置 Pin4 的 4 个控制位
5. GPIOC->CRL &=~(uint32_t)(1111<<4*5); //清空 Pin5 的 4 个控制位
6. GPIOC->CRL |=(uint32_t)(0011<<4*5); //配置 Pin5 的 4 个控制位
```

这个方法也可以实现我们所需的配置，而且修改起来比较容易，但执行的效率就比第一个方法要低了。

最后就是我们的调用库函数的方法，从内核的执行效率上看，首先库函数在被[调用](#)的时候要耗费[调用时间](#)；在函数内部，把输入参数[转换](#)为可以直接写入到寄存器的值也耗费了一些[运算时间](#)。而其它的宏、枚举等[解释](#)操作是作[编译过程完成](#)的，这部分并不消耗内核的时间。而优点呢？则是我们可以快速上手 STM32 控制器；配置外设状态时，不需要再纠结要向寄存器写入什么数值；交流方便，查错简单。这就是我们选择库的原因。

现在的处理器的主频是越来越高，我们需不需要担心 cpu 耗费那么多时间来干活会不会被累倒，野火要告诉你的是，不需要，还是担心下自己字字查询 datasheet 会不会被累倒吧。

至此，我们就把 GPIO\_Init()库函数的实现分析完毕了。分析它纯粹是为了满足自己的求知欲，学习其编程的方式、思想，这对提高我们的编程水平是很有好处的，顺便感受一下 ST 库设计的严谨性，野火认为这样的代码不仅严谨且华丽优美，不知读者你是否也有这样的感受。就像野火在论坛里面说过：要我操作寄存器，我宁愿回家种田。

我们在以后开发的工程中，一般不会去分析 ST 的库函数的实现了。因为这些库函数是很类似的，都是把原来封装好的宏或枚举标识符转化成相应的值，

写入到寄存器之中。这些都是十分枯燥和机械的工作，既然我们已经知道它的原理，又有现成的函数可供调用，就没必要再去探究了。

到了这里流水灯这个例程就算讲完了，如果你搞明白了流水灯编程的来龙去脉，那么后面的 M3 的学习路程将会简单而有趣。后面的例程也不再会像这个例程那么详细，所以大家要重点把握《4、初始 STM32 库》和《5、流水灯的前后今生》，把库的编程思想了然于胸。





## 5. POLLING-按键

在 LED 灯例程中我们已经简单体验了 GPIO 的强大之处。更强大的还在后头，野火开发板使用的芯片型号是 STM32F103VET6，具有 100 个管脚，除去晶振输入、电源输入、Boot 引脚，剩下的 80 个引脚均为 GPIO。它们分布在 GPIOA~GPIOE 的 5 个端口组之中，每个小组有 16 个引脚，所有的 GPIO 引脚都可以用作外部中断源的输入，每个 GPIO 引脚可配置为 8 种模式，不同的引脚还有相应的复用功能，复用功能重映射等，足以满足应用需求，也足以把初学者弄得晕头转向。

本章以按键工程为例，着重分析 GPIO 的模式配置。

### 5.1. GPIO 的 8 种工作模式

在初始化 GPIO 的时候，根据我们的使用要求，必须把 GPIO 设置为相应的模式。如 LED 例程中的 GPIO 引脚如果配置为模拟输入模式是必然会导致错误的。

我们配合 GPIO 结构图，来看看 GPIO 的 8 种模式及其应用场合：

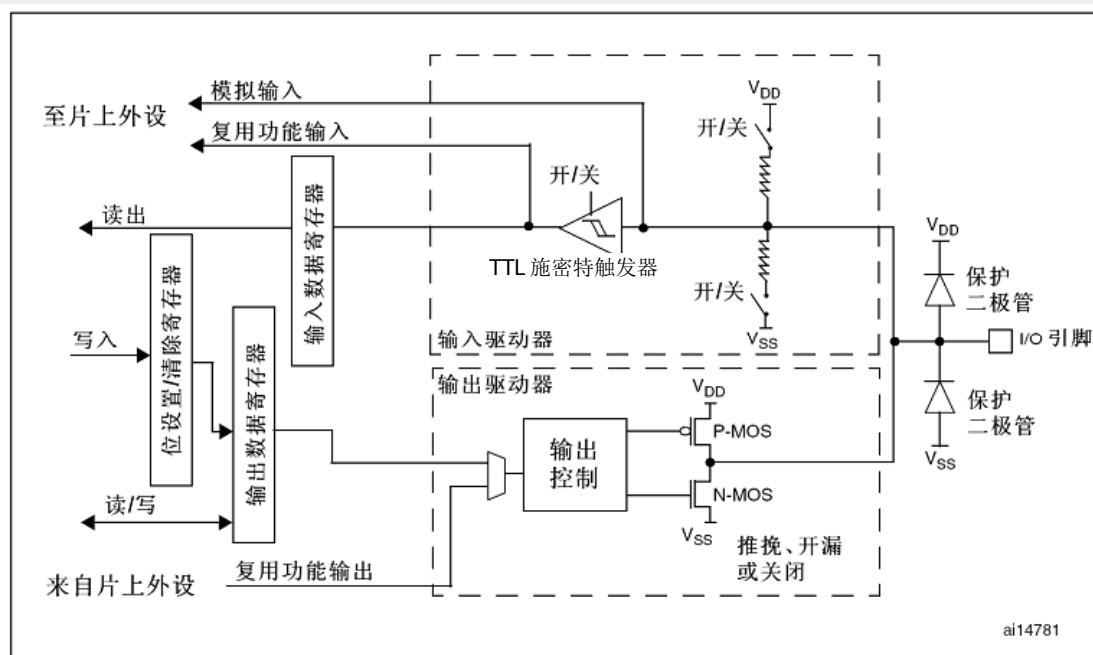


图 7-1 GPIO 结构图

图的最右端为 I/O 引脚，左端的器件位于芯片内部。I/O 引脚并联了两个用于保护的二极管。

#### 5.1.1. 四种输入模式

结构图的上半部分为输入模式结构。

接下来就遇到了两个开关和电阻，与  $V_{DD}$  相连的为 *上拉电阻*，与  $V_{SS}$  相连的为 *下拉电阻*。再连接到 *施密特触发器* 就把电压信号转化为 0、1 的数字信号存储在输入数据寄存器(IDR)。我们可以通过设置配置寄存器(CRL、CRH)，控制这两个开关，于是就可以得到 GPIO 的 *上拉输入(GPIO\_Mode\_IPU)* 和 *下拉输入模式(GPIO\_Mode\_IPD)* 了。

从它的结构我们就可以理解，若 GPIO 引脚配置为上拉输入模式，在 *默认状态下*(GPIO 引脚无输入)，读取得的 GPIO 引脚 *数据为 1，高电平*。而下拉模式则相反，在默认状态下其引脚 *数据为 0，低电平*。

而 STM32 的 *浮空输入模式(GPIO\_Mode\_IN\_FLOATING)* 在芯片内部既没有接上拉，也没有接下拉电阻，经由触发器输入。配置成这个模式直接用电压表测量其引脚电压为 1 点几伏，这是个不确定值。*由于其输入阻抗较大，一般把这种模式用于标准的通讯协议如 I2C、USART 的接收端。*

*模拟输入模式(GPIO\_Mode\_AIN)* 则关闭了施密特触发器，不接上、下拉电阻，经由另一线路把电压信号传送到片上外设模块。如传送至给 ADC 模块，由 ADC 采集电压信号。所以 *使用 ADC 外设的时候，必须设置为模拟输入模式。*

#### 5.1.2. 四种输出模式

结构图的下半部分为输出模式结构。

线路经过一个由 P-MOS 和 N-MOS 管组成的单元电路。而所谓 *推挽输出模式*，则是根据其工作方式来命名的。在输出高电平时，P-MOS 导通，低电平



时，N-MOS 管导通。两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都比普通的方式有很大的提高。推挽输出的供电平为 0 伏，高电平为 3.3 伏。

在 **开漏输出模式** 时，如果我们控制输出为 0，低电平，则使 N-MOS 管导通，使输出接地，若控制输出为 1 (无法直接输出高电平)，则既不输出高电平，也不输出低电平，为高阻态。为正常使用时必须在外部接上一个 **上拉电阻**。它具有“**线与**”特性，即很多个 **开漏模式** 引脚连接到一起时，只有当 **所有** 引脚都输出高阻态，才由上拉电阻提供高电平，此高电平的电压为外部上拉电阻所接的电源的电压。若其中一个引脚为低电平，那线路就相当于短路接地，使得整条线路都为低电平，0 伏。

STM32 的 GPIO 输出模式就分为 **普通推挽输出**(GPIO\_Mode\_Out\_PP)、**普通开漏输出** (GPIO\_Mode\_Out\_OD)及 **复用推挽输出**(GPIO\_Mode\_AF\_PP)、**复用开漏输出**(GPIO\_Mode\_AF\_OD)。

**普通推挽输出**模式一般应用在输出电平为 0 和 3.3 伏的场合。而 **普通开漏输出**一般应用在电平不匹配的场合，如需要输出 5 伏的高电平，就需要在外部接一个上拉电阻，电源为 5 伏，把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5 伏的电平。

对于相应的复用模式，则是根据 GPIO 的复用功能来选择的，如 GPIO 的引脚用作串口的输出，则使用复用 **推挽输出模式**。如果用在 IC、SMBUS 这些需要线与功能的复用场合，就使用 **复用开漏模式**。其它不同的复用场合的复用模式引脚配置将在具体的例子中讲解。

在使用任何一种开漏模式，都需要接上拉电阻。

## 5.2. 按键实验分析

了解了 GPIO 的 8 种工作模式之后，立即进行一下小测。如果采用以下的电路，我们的按键 GPIO 端口应该如何进行配置？

有两个方案可以选择，一是采用上拉输入模式，因为按键在没按下的时候，是默认为高电平的，采用内部上拉模式正好符合这个要求。

第二个方案是直接采用浮空输入模式，因为按照这个硬件电路图，在芯片外部接了上拉电阻，其实就没必要再配置成内部上拉输入模式了，因为在外部上拉与内部上拉效果都是一样的。

野火 **STM32** 开发板 按键 硬件原理图（GPIO 端口相对第一版有小改动）：

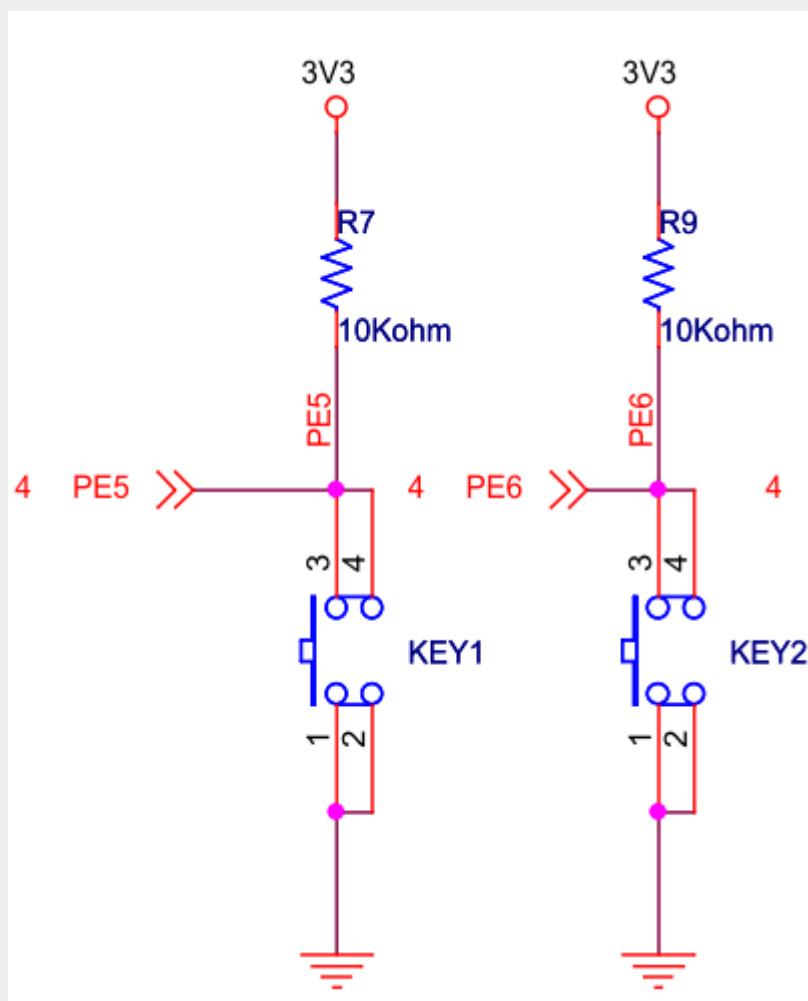


图 7-2 野火 STM32 开发板按键硬件图

## 5.3 按键代码分析

## 5.2.1. 实验描述及工程文件清单

|         |                                                                                                                                                                    |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | PE5 连接到 key1，用扫描的方式查询是否有按键按下，key1 按下时，LED1 状态取反。                                                                                                                   |
| 硬件连接    | PE5 – key1、PE6 – key2                                                                                                                                              |
| 用到的库文件  | <i>startup/start_stm32f10x_hd.c</i><br><i>CMSIS/core_cm3.c</i><br><i>CMSIS/system_stm32f10x.c</i><br><i>FWlib/stm32f10x_gpio.c</i><br><i>FWlib/stm32f10x_rcc.c</i> |
| 用户编写的文件 | <i>USER/main.c</i><br><i>USER/stm32f10x_it.c</i><br><i>USER/led.c</i><br><i>USER/key.c</i>                                                                         |

## 5.2.2. 配置工程环境

本按键实验中用到了 GPIO 和 RCC 片上外设，所以要把外设函数库文件 *FWlib/stm32f10x\_gpio.c* 和 *FWlib/stm32f10x\_rcc.c* 文件添加到工程模板之中。实验中还使用了 LED 灯，为了重用代码，我们把在前面写好的 *led.c* 和 *led.h* 用户文件复制到 *USER* 目录下，并添加到工程之中。配置工程环境最重要的一步就是别忘记在 *stm32f10x\_conf.h* 文件中把使用到的外设头文件包含进来。

```
1. /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9.
10. #include "stm32f10x_gpio.h"
```

```
11. #include "stm32f10x_rcc.h"
```

### 5.2.3. main 文件

顺着代码的执行流程，从 main 函数开始分析，这样阅读和分析别人写的代码更有条理。

```
1. /*
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7. int main(void)
8. {
9.     /* config the led */
10.    LED_GPIO_Config();
11.    LED1( ON );
12.
13.    /*config key*/
14.    Key_GPIO_Config();
15.
16.    while(1)
17.    {
18.        if( Key_Scan(GPIOE,GPIO_Pin_5) == KEY_ON )
19.        {
20.            /*LED1 反转*/
21.            GPIO_WriteBit(GPIOC, GPIO_Pin_3,
22.                (BitAction)((1-
23.                    GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
24.        }
25.    }
```

由于采用的为 3.5 版本的库，上电后，启动文件已经调用了 *SystemInit()* 将我们的系统时钟 SYSCLK 配置为 72MHz。接着进入到 main 函数，第一步先调用了在 LED 灯例程中编写的 *LED\_GPIO\_Config()*，配置 LED 用到的 I/O。再使用 *LED1( ON )* 宏，把 LED 设置为点亮状态。为了使用 LED 这部分代码，我们只要把前面写的 *led.c* 和 *led.h* 文件复制一份，放到本工程目录下，把 *led.c* 添加到工程就可以了，这样重用代码，变得非常方便。关于这部分的具体分析可参考 LED 代码讲解部分。

### 5.2.4. GPIO 初始化配置

现在我们分析一下紧接下来调用到的 *Key\_GPIO\_Config()* 函数。

```
1.  /*
2.  * 函数名: Key_GPIO_Config
3.  * 描述   : 配置按键用到的 I/O 口
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7. void Key_GPIO_Config(void)
8. {
9.     GPIO_InitTypeDef GPIO_InitStructure;
10.
11.     /*开启按键端口 (PE5) 的时钟*/
12.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
13.
14.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
15. // GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
16.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
17.
18.     GPIO_Init(GPIOE, &GPIO_InitStructure);
19. }
```

*Key\_GPIO\_Config()* 跟 LED 的 GPIO 初始化函数 *LED\_GPIO\_Config()* 是很类似的，区别只是在这个函数中，要开启的 GPIO 外设时钟为 *GPIOE* 的时钟，并且把检测按键用的引脚 *PE5* 的模式，设置为适合按键应用的 *上拉输入模式*（由于接了外部上拉电阻，也可以使用 *浮空输入*，读者可自行修改代码做实验）。在这个函数的第 15 行，读者注意到这行代码是被 *注释* 掉的，若 GPIO 被设置为 *输入模式*，是 *不需要* 设置 GPIO 端口的 *最大输出速度* 的，当然，如果配置了这个速度也没关系，*GPIO\_Init()* 函数会自动忽略它的。

在 *RCC\_APB2PeriphClockCmd()* 和 *GPIO\_InitStructure.GPIO\_Pin* 的输入参数设置之中，我们可以用符号“|”，同时配置多个参数。如：

```
1. RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE|RCC_APB2Periph_GPIOC, ENABLE);
```

输入参数为 *RCC\_APB2Periph\_GPIOE| RCC\_APB2Periph\_GPIOC*，这样调用之后，就把 *GPIOE* 和 *GPIOC* 的时钟都开启了。

```
1. GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6;
```

以上代码则表示将要同时配置 GPIO 端口的 *Pin5* 和 *Pin6*。

### 5.2.5. 利用库的数据类型

回到 `main` 函数中的应用代码，初始化了按键的 `GPIO` 之后，就在死循环里不断调用一个函数 `Key_Scan()`，用于扫描按键是否被按下。我们使用 `keil` 使用技巧中介绍的“GO ToDefinition of”功能追踪它的定义：

```
1.  /*
2.  * 函数名: Key_Scan(GPIO_TypeDef* GPIOx,u16 GPIO_Pin)
3.  * 描述   : 检测是否有按键按下
4.  * 输入   : GPIOx: x 可以是 A, B, C, D 或者 E
5.             GPIO_Pin: 待读取的端口位
6.  * 输出   : KEY_OFF(没按下按键)、KEY_ON (按下按键)
7.  */
8. u8 Key_Scan(GPIO_TypeDef* GPIOx,u16 GPIO_Pin)
9. {
10.     /*检测是否有按键按下 */
11.     if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON )
12.     {
13.         /*延时消抖*/
14.         Delay(10000);
15.         if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON )
16.         {
17.             /*等待按键释放 */
18.             while(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON);
19.             return KEY_ON;
20.         }
21.         else
22.             return KEY_OFF;
23.     }
24.     else
25.         return KEY_OFF;
26. }
```

相信延时消抖的原理大家在学习其它单片机的时候是非常了解了，本函数的功能就是扫描输入参数中指定的引脚，检测其电平变化，并作延时消抖处理，最终对按键消息进行确认。

1. 利用 `GPIO_ReadInputDataBit()` 读取输入数据，若从相应引脚读取的数据等于 0 (`KEY_ON`)，低电平，表明可能有按键按下，调用延时函数。否则返回 `KEY_OFF`，表示按键没有被按下。
2. 延时之后再次利用 `GPIO_ReadInputDataBit()` 读取输入数据，若依然为低电平，表明确实有按键被按下了。否则返回 `KEY_OFF`，表示按键没有被按下。

3. 循环调用 `GPIO_ReadInputDataBit()` 一直检测按键的电平，直至按键被释放，被释放后，返回表示按键被按下的标志 `KEY_ON`。

以上是按键消抖的流程，调用了一个库函数

`GPIO_ReadInputDataBit()`。输入参数为要读取的端口、引脚，返回引脚的输入电平状态，高电平为 1，低电平为 0；

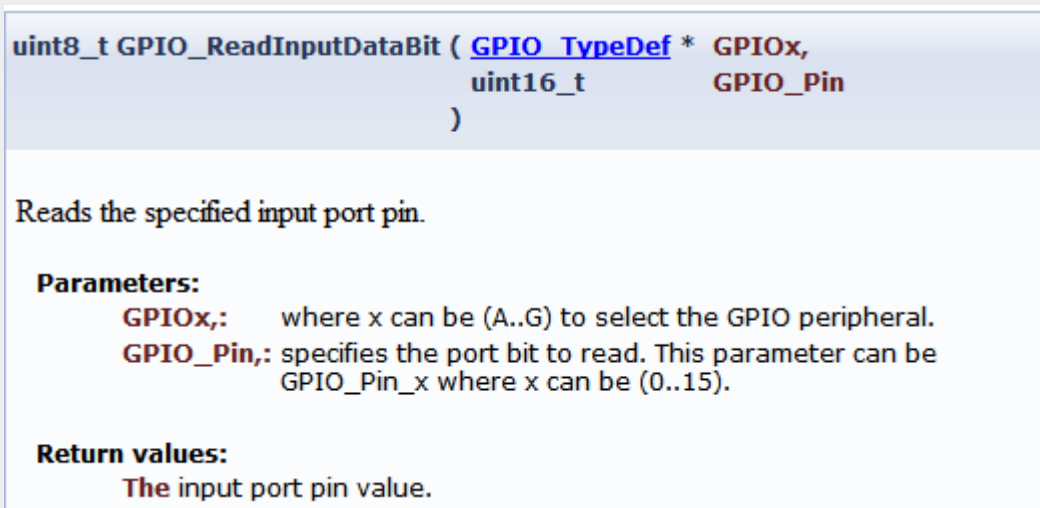


图 5-3 GPIO 输入数据读取函数

但按键消抖并不是本小节的重点，而且这样的消抖在实际的工程应用中并无价值。重点是教会大家如何利用 ST 库定义的新数据类型来编写用户函数。

`Key_Scan()`函数的形参，其实跟 `GPIO_ReadInputDataBit()`的形参是一样的，都是 `(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)`，如果再在 `Key_Scan()`的定义中加入断言，用于输入参数检查，看起来是不是很像 ST 官方的库函数？其实这是野火写的一个用户函数。

这个例子告诉大家，在 `stm32f10x.h` 文件中的新数据类型，我们不但可以利用它们来定义变量，还应善于利用这些数据类型来编写用户函数。如这个 `Key_Scan()`函数，由于使用了这些引脚类型形参，在其它不同的工程之中，我们就可以在调用时，通过输入不同的实参，来检测其它按键的引脚了。

如在调用 `Key_Scan()`函数时，把实参改成 `(GPIOE, GPIO_Pin_6)`，就可以用 `Key-2`来控制 LED1 啦(当然，`GPIO_Pin_6`要在 `Key_GPIO_Config()`中初始

化)。是不是很方便呢，利用官方的库，我们可以很方便地开发出这一类用户函数，这就是库的魅力呀！

#### 5.2.6. 实现 LED 反转

在 main 函数中，检测到有按键被按下之后，就开始执行 LED 反转的操作。

```
1. GPIO_WriteBit(GPIOC, GPIO_Pin_3,  
2.               (BitAction)((1-  
   GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3)))));
```

这一段代码首先调用了 `GPIO_ReadOutputDataBit()` 函数，读取 PC3 的当前输出电平，然后再用 1 减去读取得的电平数据状态，相当于获取一个与当前输出相反的状态，再把这个相反的状态利用 `GPIO_WriteBit()` 函数写入到 PC3，从而实现了输出状态取反的功能。大家会发现，这实在太复杂了，我们只不过是取反输出，在 51 单片机可以直接 `PA0=~PA0` 就可以完成了，而在这里使用库的时候，我们竟然要先读取状态，再计算出反状态，最后再写入新状态。能不能也像单片机那样使用呢？答案是肯定的，我们可以采用 Cortex-M3 的位带操作方式，实现同样的功能。

#### 5.2.7. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板，LED1 亮，按下按键时 LED1 灭，再按下按键时 LED1 亮，如此循环。



## 6. 6。EXTI-按键

*EXTI (External interrupt)* 就是指外部中断，通过 GPIO 检测输入脉冲，引起中断事件，打断原来的代码执行流程，进入到中断服务函数中进行处理，处理完后，再返回到中断之前的代码中执行。

前面提到，STM32 的 *所有 GPIO* 都可以用作外部中断源的输入端，利用这个特性，我们可以把按键 *轮询检测* 改为由 *中断* 来处理，大大提高软件执行的效率。

### 6. 1. STM32 的中断和异常

Cortex 内核具有强大的异常响应系统，它把能够打断当前代码执行流程的事件分为 *异常(exception)* 和 *中断(interrupt)*，并把它们用一个表管理起来，编号为 0~15 的称为 *内核异常*，而 16 以上的则称为 *外部中断*（外，相对内核而言），这个表就称为 *中断向量表*。

而 STM32 对这个表重新进行了编排，把编号从 -3 至 6 的中断向量定义为系统异常，*编号为负* 的内核异常不能被设置优先级，如复位(Reset)、不可屏蔽中断(NMI)、硬错误(Hardfault)。从编号 7 开始的为外部中断，这些中断的优先级都是可以自行设置的。详细的 STM32 中断向量表见图 8-1，STM32 中断向量表。

| 位置 | 优先级 | 优先级类型 | 名称                 | 说明                               | 地址                          |
|----|-----|-------|--------------------|----------------------------------|-----------------------------|
|    | -   | -     | -                  | 保留                               | 0x0000_0000                 |
|    | -3  | 固定    | Reset              | 复位                               | 0x0000_0004                 |
|    | -2  | 固定    | NMI                | 不可屏蔽中断<br>RCC时钟安全系统(CSS)联接到NMI向量 | 0x0000_0008                 |
|    | -1  | 固定    | 硬件失效(HardFault)    | 所有类型的失效                          | 0x0000_000C                 |
|    | 0   | 可设置   | 存储管理(MemManage)    | 存储器管理                            | 0x0000_0010                 |
|    | 1   | 可设置   | 总线错误(BusFault)     | 预取指失败, 存储器访问失败                   | 0x0000_0014                 |
|    | 2   | 可设置   | 错误应用(UsageFault)   | 未定义的指令或非法状态                      | 0x0000_0018                 |
|    | -   | -     | -                  | 保留                               | 0x0000_001C<br>~0x0000_002B |
|    | 3   | 可设置   | SVCall             | 通过SWI指令的系统服务调用                   | 0x0000_002C                 |
|    | 4   | 可设置   | 调试监控(DebugMonitor) | 调试监控器                            | 0x0000_0030                 |
|    | -   | -     | -                  | 保留                               | 0x0000_0034                 |
|    | 5   | 可设置   | PendSV             | 可挂起的系统服务                         | 0x0000_0038                 |
|    | 6   | 可设置   | SysTick            | 系统嘀嗒定时器                          | 0x0000_003C                 |
| 0  | 7   | 可设置   | WWDG               | 窗口定时器中断                          | 0x0000_0040                 |
| 1  | 8   | 可设置   | PVD                | 连到EXTI的电源电压检测(PVD)中断             | 0x0000_0044                 |
| 2  | 9   | 可设置   | TAMPER             | 侵入检测中断                           | 0x0000_0048                 |
| 3  | 10  | 可设置   | RTC                | 实时时钟(RTC)全局中断                    | 0x0000_004C                 |
| 4  | 11  | 可设置   | FLASH              | 闪存全局中断                           | 0x0000_0050                 |

|    |    |     |                |                  |             |
|----|----|-----|----------------|------------------|-------------|
| 5  | 12 | 可设置 | RCC            | 复位和时钟控制(RCC)中断   | 0x0000_0054 |
| 6  | 13 | 可设置 | EXTI0          | EXTI线0中断         | 0x0000_0058 |
| 7  | 14 | 可设置 | EXTI1          | EXTI线1中断         | 0x0000_005C |
| 8  | 15 | 可设置 | EXTI2          | EXTI线2中断         | 0x0000_0060 |
| 9  | 16 | 可设置 | EXTI3          | EXTI线3中断         | 0x0000_0064 |
| 10 | 17 | 可设置 | EXTI4          | EXTI线4中断         | 0x0000_0068 |
| 11 | 18 | 可设置 | DMA1通道1        | DMA1通道1全局中断      | 0x0000_006C |
| 12 | 19 | 可设置 | DMA1通道2        | DMA1通道2全局中断      | 0x0000_0070 |
| 13 | 20 | 可设置 | DMA1通道3        | DMA1通道3全局中断      | 0x0000_0074 |
| 14 | 21 | 可设置 | DMA1通道4        | DMA1通道4全局中断      | 0x0000_0078 |
| 15 | 22 | 可设置 | DMA1通道5        | DMA1通道5全局中断      | 0x0000_007C |
| 16 | 23 | 可设置 | DMA1通道6        | DMA1通道6全局中断      | 0x0000_0080 |
| 17 | 24 | 可设置 | DMA1通道7        | DMA1通道7全局中断      | 0x0000_0084 |
| 18 | 25 | 可设置 | ADC1_2         | ADC1和ADC2的全局中断   | 0x0000_0088 |
| 19 | 26 | 可设置 | USB_HP_CAN_TX  | USB高优先级或CAN发送中断  | 0x0000_008C |
| 20 | 27 | 可设置 | USB_LP_CAN_RX0 | USB低优先级或CAN接收0中断 | 0x0000_0090 |
| 21 | 28 | 可设置 | CAN_RX1        | CAN接收1中断         | 0x0000_0094 |
| 22 | 29 | 可设置 | CAN_SCE        | CAN SCE中断        | 0x0000_0098 |
| 23 | 30 | 可设置 | EXTI9_5        | EXTI线[9:5]中断     | 0x0000_009C |
| 24 | 31 | 可设置 | TIM1_BRK       | TIM1刹车中断         | 0x0000_00A0 |
| 25 | 32 | 可设置 | TIM1_UP        | TIM1更新中断         | 0x0000_00A4 |
| 26 | 33 | 可设置 | TIM1_TRG_COM   | TIM1触发和通信中断      | 0x0000_00A8 |

|    |    |     |              |                       |             |
|----|----|-----|--------------|-----------------------|-------------|
| 27 | 34 | 可设置 | TIM1_CC      | TIM1捕获比较中断            | 0x0000_00AC |
| 28 | 35 | 可设置 | TIM2         | TIM2全局中断              | 0x0000_00B0 |
| 29 | 36 | 可设置 | TIM3         | TIM3全局中断              | 0x0000_00B4 |
| 30 | 37 | 可设置 | TIM4         | TIM4全局中断              | 0x0000_00B8 |
| 31 | 38 | 可设置 | I2C1_EV      | I <sup>2</sup> C1事件中断 | 0x0000_00BC |
| 32 | 39 | 可设置 | I2C1_ER      | I <sup>2</sup> C1错误中断 | 0x0000_00C0 |
| 33 | 40 | 可设置 | I2C2_EV      | I <sup>2</sup> C2事件中断 | 0x0000_00C4 |
| 34 | 41 | 可设置 | I2C2_ER      | I <sup>2</sup> C2错误中断 | 0x0000_00C8 |
| 35 | 42 | 可设置 | SPI1         | SPI1全局中断              | 0x0000_00CC |
| 36 | 43 | 可设置 | SPI2         | SPI2全局中断              | 0x0000_00D0 |
| 37 | 44 | 可设置 | USART1       | USART1全局中断            | 0x0000_00D4 |
| 38 | 45 | 可设置 | USART2       | USART2全局中断            | 0x0000_00D8 |
| 39 | 46 | 可设置 | USART3       | USART3全局中断            | 0x0000_00DC |
| 40 | 47 | 可设置 | EXTI15_10    | EXTI线[15:10]中断        | 0x0000_00E0 |
| 41 | 48 | 可设置 | RTCAlarm     | 连到EXTI的RTC闹钟中断        | 0x0000_00E4 |
| 42 | 49 | 可设置 | USB唤醒        | 连到EXTI的从USB待机唤醒中断     | 0x0000_00E8 |
| 43 | 50 | 可设置 | TIM8_BRK     | TIM8刹车中断              | 0x0000_00EC |
| 44 | 51 | 可设置 | TIM8_UP      | TIM8更新中断              | 0x0000_00F0 |
| 45 | 52 | 可设置 | TIM8_TRG_COM | TIM8触发和通信中断           | 0x0000_00F4 |
| 46 | 53 | 可设置 | TIM8_CC      | TIM8捕获比较中断            | 0x0000_00F8 |
| 47 | 54 | 可设置 | ADC3         | ADC3全局中断              | 0x0000_00FC |
| 48 | 55 | 可设置 | FSMC         | FSMC全局中断              | 0x0000_0100 |

图 8-1 中断向量表

这个表可以从《STM32 中文参考手册》找到，但野火一般是从启动文件 *startup\_stm32f10x\_hd.s* 中查找的，因为不同型号的 STM32 芯片，中断向量表稍微有点区别，在启动文件中，已经有相应芯片可用的全部中断向量。而且在编写 *中断服务函数* 时，需要从启动文件中定义的中断向量表查找中断服务函数名。

## 6.2. NVIC 中断控制器

STM32 的中断如此之多，配置起来并不容易，因此，我们需要一个强大而方便的中断控制器 NVIC (Nested Vectored Interrupt Controller)。NVIC 是属于 Cortex 内核的器件，不可屏蔽中断 (NMI) 和外部中断都由它来处理，而 SYSTICK 不是由 NVIC 来控制的。

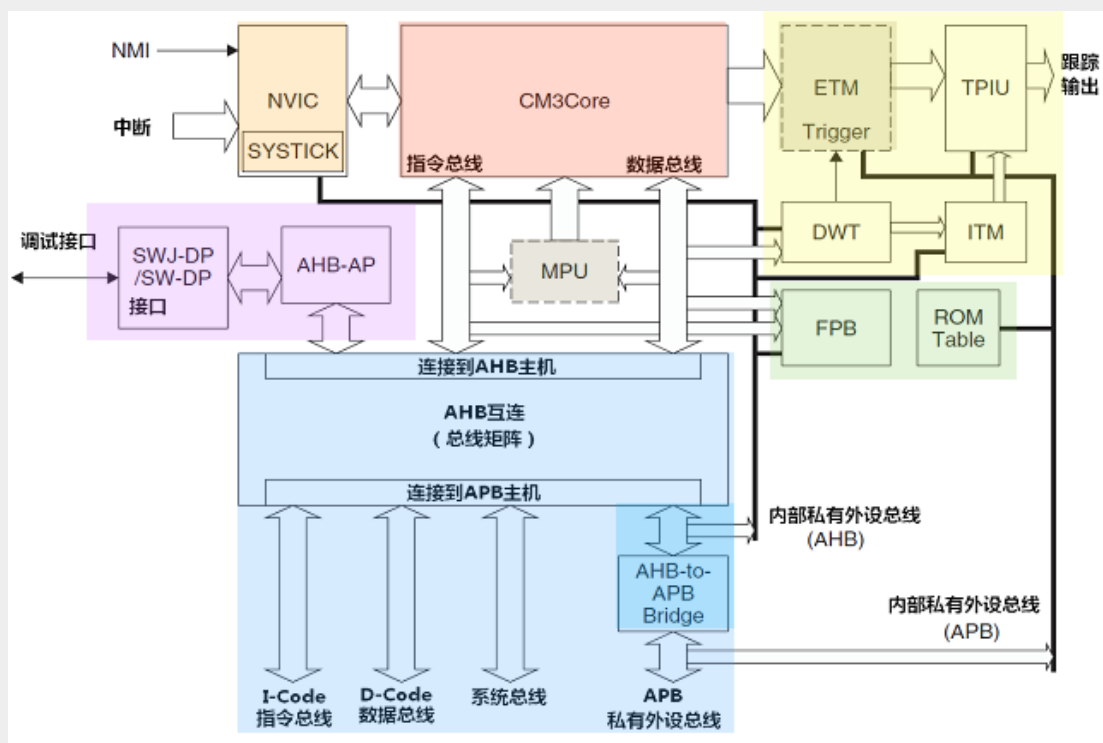


图 8-2 NVIC 在内核中的位置

### 6.2.1. NVIC 结构体成员

当我们要使用 NVIC 来配置中断时，自然想到 ST 库肯定也已经把它封装成库函数了。查找库帮助文档，发现在 *Modules->STM32F10x\_StdPeriph\_Driver->misc* 查找到一个 *NVIC\_Init()* 函数，对 NVIC 初始化，首先要定义并填充一个 *NVIC\_InitTypeDef* 类型的结构体。

这个结构体有四个成员

|                                          |                  |
|------------------------------------------|------------------|
| <i>NVIC_IRQChannel</i>                   | 需要配置的中断向量        |
| <i>NVIC_IRQChannelCmd</i>                | 使能或关闭相应中断向量的中断响应 |
| <i>NVIC_IRQChannelPreemptionPriority</i> | 配置相应中断向量抢占优先级    |
| <i>NVIC_IRQChannelSubPriority</i>        | 配置相应中断向量的响应优先级   |

前面两个结构体成员都很好理解，首先要用 *NVIC\_IRQChannel* 参数来选择将要配置的中断向量，用 *NVIC\_IRQChannelCmd* 参数来进行使能(ENABLE)或

关闭 (*DISABLE*) 该中断。在 *NVIC\_IRQChannelPreemptionPriority* 成员要配置中断向量的 *抢占优先级*，在 *NVIC\_IRQChannelSubPriority* 需要配置中断向量的 *响应优先级*。对于中断的配置，最重要的便是配置其优先级，但 STM32 的同一个中断向量为什么需要设置两种优先级？这两种优先级有什么区别？

### 6.2.2. 抢占优先级和响应优先级

STM32 的中断向量具有两个属性，一个为 *抢占属性*，另一个为 *响应属性*，其属性 *编号越小*，表明它的 *优先级别越高*。

抢占，是指打断其它中断的属性，即因为具有这个属性，会出现嵌套中断（在执行中断服务函数 A 的过程中被中断 B 打断，执行完中断服务函数 B 再继续执行中断服务函数 A），抢占属性由 *NVIC\_IRQChannelPreemptionPriority* 的参数配置。

而响应属性则应用在抢占属性相同的情况下，当两个中断向量的抢占优先级相同时，如果两个中断同时到达，则先处理响应优先级高的中断，响应属性由 *NVIC\_IRQChannelSubPriority* 的参数配置。

例如，现在有三个中断向量：

| 中断向量 | 抢占优先级 | 响应优先级 |
|------|-------|-------|
| A    | 0     | 0     |
| B    | 1     | 0     |
| C    | 1     | 1     |

若内核正在执行 C 的中断服务函数，则它能被抢占优先级更高的中断 A 打断，由于 B 和 C 的抢占优先级相同，所以 C 不能被 B 打断。但如果 B 和 C 中断是同时到达的，内核就会首先响应响应优先级别更高的 B 中断。

### 6.2.3. NVIC 的优先级组

在配置优先级的时候，还要注意一个很重要的问题，中断种类的数量。NVIC 只可以配置 16 种中断向量的优先级，也就是说，抢占优先级和响应优先级的数量由一个 4 位的数字来决定，把这个 4 位数字的位数分配成抢占优先级部分和响应优先级部分。有 5 组分配方式：

第 0 组：所有 4 位用来配置抢占优先级，即 NVIC 配置的  $2^4=16$  种中断向量都是只有抢占属性，没有响应属性。

第 1 组：最高 1 位用来配置抢占优先级，低 3 位用来配置响应优先级。表示有  $2^1=2$  种级别的抢占优先级(0 级, 1 级)，有  $2^3=8$  种响应优先级，即在 16 种中断向量之中，有 8 种中断，其抢占优先级都为 0 级，而它们的响应优先级分别为 0~7，其余 8 种中断向量的抢占优先级则都为 1 级，响应优先级别分别为 0~7。

第 2 组：2 位用来配置抢占优先级，2 位用来配置响应优先级。即  $2^2=4$  种抢占优先级， $2^2=4$  种响应优先级。

第 3 组：高 3 位用来配置抢占优先级，最低 1 位用来配置响应优先级。即有 8 种抢占优先级，2 种响应 2 优先级。

第 4 组：所有 4 位用来配置响应优先级。即 16 种中断向量具有都不相同的响应优先级。

要配置这些优先级组，可以采用库函数 `NVIC_PriorityGroupConfig()`，可输入的参数为 `NVIC_PriorityGroup_0 ~ NVIC_PriorityGroup_4`，分别为以上介绍的 5 种分配组。

于是，有读者觉得疑惑了，如此强大的 STM32，所有 GPIO 都能够配置成外部中断，USART、ADC 等外设也有中断，而 NVIC 只能配置 16 种中断向量，那在某个工程中使用了超过 16 个的中断怎么办呢？注意 NVIC 能配置的是 16 种中断向量，而不是 16 个，当工程之中有超过 16 个中断向量时，必然有 2 个以上的中断向量是使用相同的中断种类，而具有相同中断种类的中断向量不能互相嵌套。

STM32 单片机的所有 I/O 端口都可以配置为 EXTI 中断模式，用来捕捉外部信号，可以配置为下降沿中断，上升沿中断和上升下降沿中断这三种模式。它们以下图的方式连接到 16 个外部中断/事件线上

### 6.3. EXTI 外部中断

STM32 的所有 GPIO 都引入到 EXTI 外部中断线上，使得所有的 GPIO 都能作为外部中断的输入源。GPIO 与 EXTI 的连接方式见图 6-3

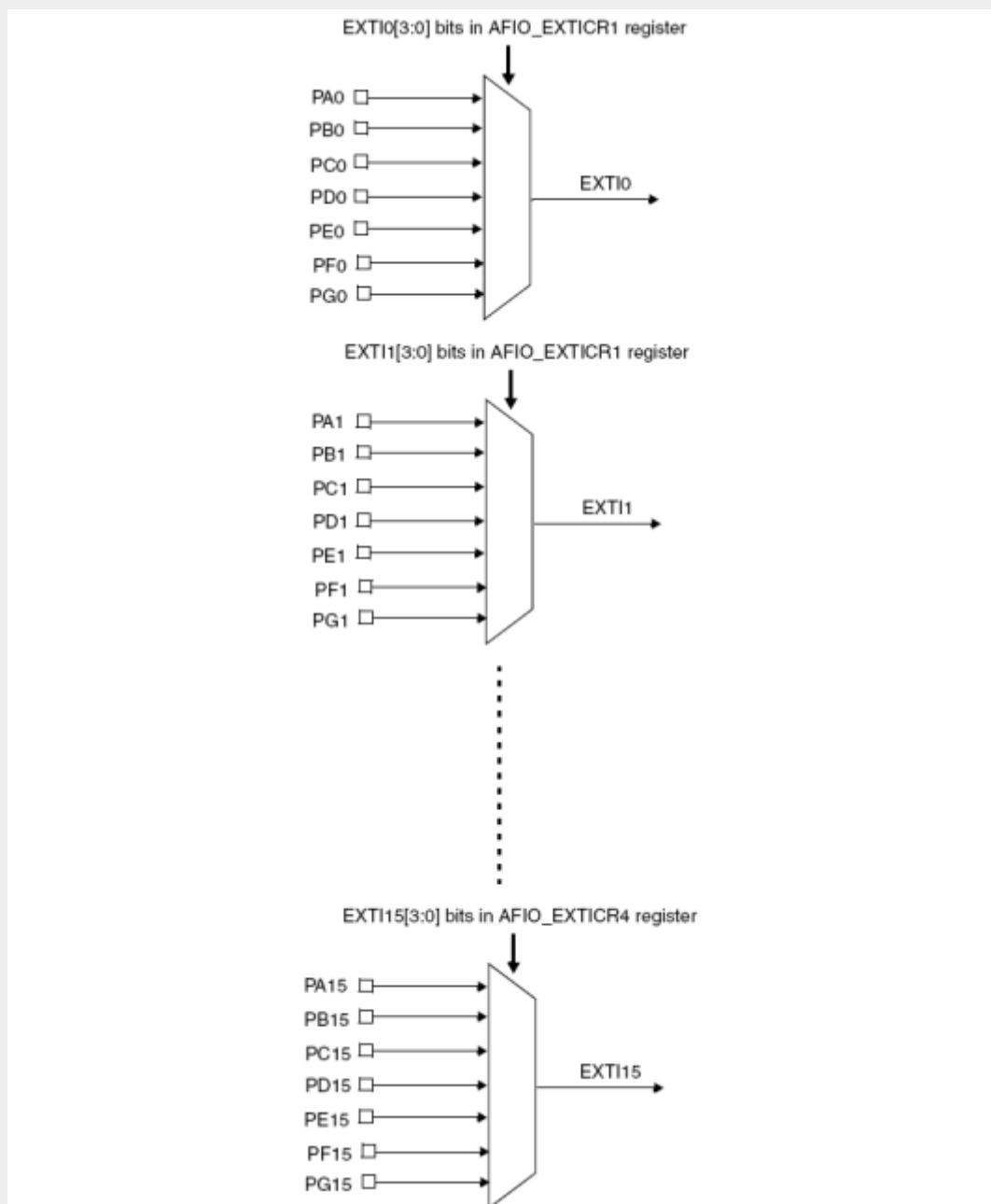


图 6-3 EXTI 与 GPIO 连接图



观察这个图知道，*PA0~PG0* 连接到 *EXTI0*、*PA1~PG1* 连接到 *EXTI1*、.....、*PA15~PG15* 连接到 *EXTI15*。这里大家要注意的是：*PAx~PGx* 端口的中断事件都连接到了 *EXTIx*，即同一时刻 *EXTx* 只能相应一个端口的事件触发，不能够同一时间响应所有 *GPIO* 端口的事件，但可以分时复用。它可以配置为上升沿触发，下降沿触发或双边沿触发。*EXTI* 最普通的应用就是接上一个按键，设置为下降沿触发，用中断来检测按键。

#### 6.4. 中断检测按键实验分析

##### 6.4.1. 实验描述及工程文件清单

|         |                                                                                                                                                                                                                            |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | PB0 连接到 key1，PB0 配置为线中断模式，key1 按下时，进入线中断处理函数，LED1 状态取反。                                                                                                                                                                    |
| 硬件连接    | PE5 – key1、PE6 – key2                                                                                                                                                                                                      |
| 用到的库文件  | <i>startup/start_stm32f10x_hd.c</i><br><i>CMSIS/core_cm3.c</i><br><i>CMSIS/system_stm32f10x.c</i><br><i>FWlib/stm32f10x_gpio.c</i><br><i>FWlib/stm32f10x_rcc.c</i><br><i>FWlib/stm32f10x_exti.c</i><br><i>FWlib/misc.c</i> |
| 用户编写的文件 | <i>USER/main.c</i><br><i>USER/stm32f10x_it.c</i><br><i>USER/led.c</i><br><i>USER/exti.c</i>                                                                                                                                |

##### 6.4.2. 配置工程环境

本中断检测按键实验照例使用了 *GPIO* 和 *RCC* 片上外设，由于还使用到了中断，所以比上一个按键实验要多使用两个库文件，分别为

*FWlib/stm32f10x\_exti.c*和 *FWlib/misc.c*，必须把这两个文件也添加到工程之中。其中 *stm32f10x\_exti.c*文件包含了支持 *exti*配置和操作的相关库函数；而 *misc.c*文件则包含了 *NVIC*的配置函数。本实验中我们还会在 *stm32f10x\_it.c*文件中编写中断服务函数。

添加了所需要的库文件、用户文件之后，要在 *stm32f10x\_conf.h*文件中配置使用到的头文件。

```
1.  /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****/
9.  #include "stm32f10x_exti.h"
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "misc.h" /
```

### 6.4.3. main 文件

我们从 main 函数开始分析：

```
1.  /**
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      /* config the led */
10.     LED_GPIO_Config();
11.     LED1( ON );
12.
13.     /* exti line config */
14.     EXTI_PE5_Config();
15.
16.     /* wait interrupt */
17.     while(1)
18.     {
19.     }
20. }
```

使用 *LED\_GPIO\_Config()* 配置好 LED 用到的 I/O 后，调用 *LED1()*点亮一盏 LED 灯。这两个函数的具体讲解可参考前面的教程。

#### 6.4.4. 配置外部中断

现在我们重点分析下 *EXTI\_PE5\_Config()* 这个函数，这是一个在用户文件 *exti.c* 中实现的函数，它完成了一般配置一个 I/O 为 EXTI 中断的步骤，主要为功能：

1. 使能 EXTIx 线的时钟和第二功能 AFIO 时钟
2. 配置 EXTIx 线的中断优先级
3. 配置 EXTI 中断线 I/O
4. 选定要配置为 EXTI 的 I/O 口线和 I/O 口的工作模式
5. EXTI 中断线工作模式配置

EXTI\_PE5\_Config()代码：

```
1. /*
2. * 函数名: EXTI_PE5_Config
3. * 描述   : 配置 PE5 为线中断口，并设置中断优先级
4. * 输入   : 无
5. * 输出   : 无
6. * 调用   : 外部调用
7. */
8. void EXTI_PE5_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     EXTI_InitTypeDef EXTI_InitStructure;
12.
13.     /* config the extiline(PE5) clock and AFIO clock */
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE | RCC_APB2Periph_AFIO,
15.     ENABLE);
16.
17.     /* config the NVIC(PE5) */
18.     NVIC_Configuration();
19.
20.     /* EXTI line gpio config(PE5) */
21.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
22.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD; // 上拉输入
23.     GPIO_Init(GPIOE, &GPIO_InitStructure);
24.
25.     /* EXTI line(PE5) mode config */
26.     GPIO_EXTILineConfig(GPIO_PortSourceGPIOE, GPIO_PinSource5);
27.     EXTI_InitStructure.EXTI_Line = EXTI_Line5;
28.     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
29.     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿中
30.     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
31.     EXTI_Init(&EXTI_InitStructure);
32. }
```

#### 6.4.5. AFIO 时钟

`EXTI_PE5_Config()`代码的第 14 行，调用 `RCC_APB2PeriphClockCmd()` 时还输入了参数 `RCC_APB2Periph_AFIO`，表示开启 AFIO 的时钟。见图 6-4。

**AFIO (alternate-function I/O)**，指 GPIO 端口的复用功能，GPIO 除了用作普通的输入输出(**主功能**)，还可以作为片上外设的复用输入输出，如串口，ADC，这些就是复用功能。大多数 GPIO 都有一个**默认复用功能**，有的 GPIO 还有**重映射功能**，重映射功能是指把原来属于 A 引脚的默认复用功能，转移到了 B 引脚进行使用，前提是 B 引脚具有这个重映射功能

当把 GPIO 用作 **EXTI 外部中断** 或使用 **重映射功能** 的时候，必须开启 AFIO 时钟，而在使用 **默认复用功能** 的时候，就不必开启 AFIO 时钟了。

| 脚位     |        |        |        |         |         | 管脚名称 | 类型 <sup>(1)</sup> | I/O电平 <sup>(2)</sup> | 主功能 <sup>(3)</sup><br>(复位后) | 可选的复用功能  |           |
|--------|--------|--------|--------|---------|---------|------|-------------------|----------------------|-----------------------------|----------|-----------|
| BGA144 | BGA100 | WCSP64 | LQFP64 | LQFP100 | LQFP144 |      |                   |                      |                             | 默认复用功能   | 重定义功能     |
| L6     | -      | -      | -      | -       | 55      | PF15 | I/O               | FT                   | PF15                        | FSMC_A9  |           |
| K6     | -      | -      | -      | -       | 56      | PG0  | I/O               | FT                   | PG0                         | FSMC_A10 |           |
| J6     | -      | -      | -      | -       | 57      | PG1  | I/O               | FT                   | PG1                         | FSMC_A11 |           |
| M7     | H5     | -      | -      | 38      | 58      | PE7  | I/O               | FT                   | PE7                         | FSMC_D4  | TIM1_ETR  |
| L7     | J5     | -      | -      | 39      | 59      | PE8  | I/O               | FT                   | PE8                         | FSMC_D5  | TIM1_CH1N |
| K7     | K5     | -      | -      | 40      | 60      | PE9  | I/O               | FT                   | PE9                         | FSMC_D6  | TIM1_CH1  |

图 6-4 GPIO 引脚功能说明

#### 6.4.6. NVIC 初始化配置

在 `EXTI_PE5_Config()`代码的第 17 行调用了 `NVIC_Configuration()`，这是用户编写的用来配置 **NVIC** 控制器的函数。其实现如下：

```

1.  /*
2.   * 函数名: NVIC_Configuration
3.   * 描述   : 配置嵌套向量中断控制器 NVIC
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 内部调用
7.   */
8. static void NVIC_Configuration(void)
9. {
10.    NVIC_InitTypeDef NVIC_InitStructure;

```

```
11.  
12.  /* Configure one bit for preemption priority */  
13.  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);  
14.  
15.  /* 配置 P[A|B|C|D|E]5 为中断源 */  
16.  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;  
17.  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;  
18.  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;  
19.  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  
20.  NVIC_Init(&NVIC_InitStructure);  
21. }
```

本代码的第 13 行调用了 `NVIC_PriorityGroupConfig()` 库函数，把 NVIC 中断优先级分组设置为第 1 组。接下来开始向 NVIC 初始化结构体写入参数 `.NVIC_IRQChannel = EXTI9_5_IRQn`，表示要配置的为 EXTI 第 5~9 线的中断向量。因为按键 PE5 对应的 EXTI 线为 EXTI5，而从 EXTI5~EXTI9 线，由于它们是使用同一个中断向量的，所以只能写入 `EXTI9_5_IRQn` 这个参数。这些可写入的参数可以在 `stm32f10x.h` 文件的 `IRQn` 类型定义中查找到。

然后配置 **抢占优先级** 和 **响应优先级**，因为这个工程简单，就直接把它设置为最高级中断。填充完结构体，别忘记最后要调用 `NVIC_Init()` 函数来向寄存器写入参数。

#### 6.4.7. EXTI 初始化配置

回到 `EXTI_PE5_Config()` 代码中，配置好 NVIC 后，还要对 GPIOE 进行初始化，这部分和按键轮询的设置类似。

接下来，调用 `GPIO_EXTILineConfig()` 函数把 `GPIOE`，`Pin5` 设置为 `EXTI` 输入线。

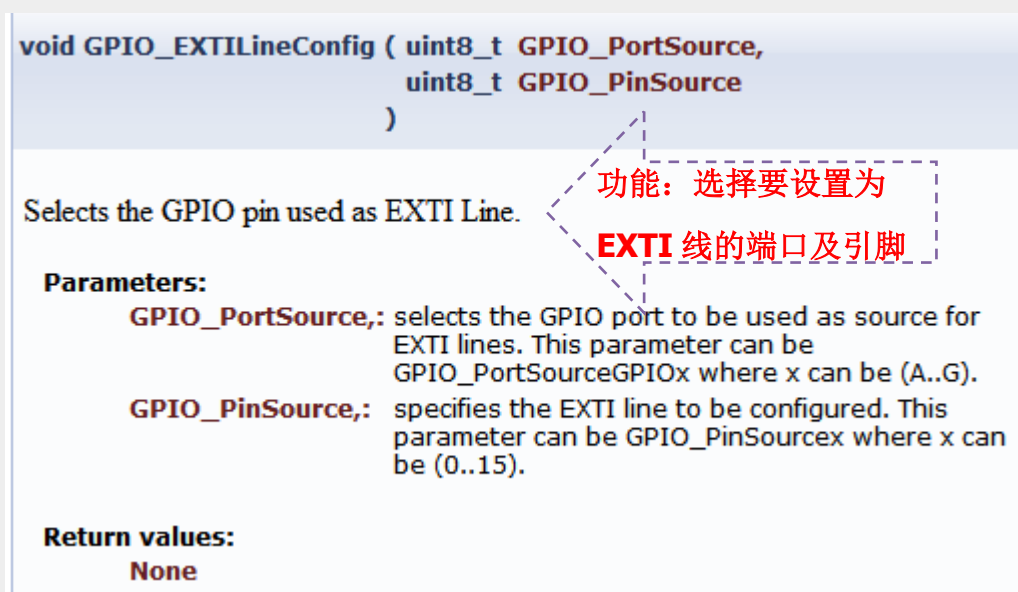


图 6-5 EXTI 中断源配置函数

选择好了 GPIO，开始填写 EXTI 的初始化结构体。从这些参数的名字，读者就已经可以知道野火是如何把它应用到按键检测中了吧？

```
.EXTI_Line = EXTI_Line5;
```

给 *EXTI\_Line* 成员赋值。选择 *EXTI\_Line5* 线进行配置，因为按键的 PE5 连接到了 *EXTI\_Line5*。

```
.EXTI_Mode = EXTI_Mode_Interrupt;
```

给 *EXTI\_Mode* 成员赋值。把 *EXTI\_Line5* 的模式设置为中断模式 *EXTI\_Mode\_Interrupt*。这个结构体成员也可以赋值为事件模式 *EXTI\_Mode\_Event*，这个模式不会立刻触发中断，而只是在寄存器上把相应的事件标置位置 1，应用这个模式要不停地查询相应的寄存器。

```
.EXTI_Trigger = EXTI_Trigger_Falling;
```

给 *EXTI\_Trigger* 成员赋值。把触发方式 (*EXTI\_Trigger*) 设置为下降沿触发 (*EXTI\_Trigger\_Falling*)。

```
.EXTI_LineCmd = ENABLE;
```

给 *EXTI\_LineCmd* 成员赋值。把 *EXTI\_LineCmd* 设置为使能。

最后调用 *EXTI\_Init()* 把 *EXTI* 初始化结构体的参数写入寄存器。

#### 6.4.8. 编写中断服务函数

在这个 EXTI 设置中我们把 *PE5* 连接到内部的 *EXTI5*, GPIO 配置为上拉输入, 工作在下降沿中断。在外围电路上我们将 *PE5* 接到了 *key1* 上。当按键没有按下时, *PE5* 始终为高, 当按键按下时 *PE5* 变为低, 从而 *PE5* 上产生一个下降沿跳变, *EXTI5* 会捕捉到这一跳变, 并产生相应的中断, 中断服务程序在 *stm32f10x\_it.c* 中实现。

*stm32f10x\_it.c* 文件是专门用来存放中断服务函数的。文件中默认只有几个关于系统异常的中断服务函数, 而且都是空函数, 在需要的时候自己进行编写。那么中断服务函数名是不是可以自己定义呢? 不可以。中断服务函数的名字必须要跟启动文件 *startup\_stm32f10x\_hd.s* 中的中断向量表定义一致。以下为启动文件中定义的部分向量表:

```
1. DCD     EXTI0_IRQHandler      ; EXTI Line 0
2. DCD     EXTI1_IRQHandler      ; EXTI Line 1
3. DCD     EXTI2_IRQHandler      ; EXTI Line 2
4. DCD     EXTI3_IRQHandler      ; EXTI Line 3
5. DCD     EXTI4_IRQHandler      ; EXTI Line 4
6. DCD     DMA1_Channel1_IRQHandler ; DMA1 Channel 1
7. DCD     DMA1_Channel2_IRQHandler ; DMA1 Channel 2
8. DCD     DMA1_Channel3_IRQHandler ; DMA1 Channel 3
9. DCD     DMA1_Channel4_IRQHandler ; DMA1 Channel 4
10. DCD     DMA1_Channel5_IRQHandler ; DMA1 Channel 5
11. DCD     DMA1_Channel6_IRQHandler ; DMA1 Channel 6
12. DCD     DMA1_Channel7_IRQHandler ; DMA1 Channel 7
13. DCD     ADC1_2_IRQHandler     ; ADC1 & ADC2
14. DCD     USB_HP_CAN1_TX_IRQHandler ; USB High Priority or CAN1 TX
15. DCD     USB_LP_CAN1_RX0_IRQHandler ; USB Low Priority or CAN1 RX0
16. DCD     CAN1_RX1_IRQHandler   ; CAN1 RX1
17. DCD     CAN1_SCE_IRQHandler   ; CAN1 SCE
18. DCD     EXTI9_5_IRQHandler    ; EXTI Line 9..5
19. DCD     TIM1_BRK_IRQHandler    ; TIM1 Break
20. DCD     TIM1_UP_IRQHandler    ; TIM1 Update
21. DCD     TIM1_TRG_COM_IRQHandler ; TIM1 Trigger and Commutation
```

```
22. DCD      TIM1_CC_IRQHandler      ; TIM1 Capture Compare
23. DCD      TIM2_IRQHandler         ; TIM2
24. DCD      TIM3_IRQHandler         ; TIM3
25. DCD      TIM4_IRQHandler         ; TIM4
26. DCD      I2C1_EV_IRQHandler      ; I2C1 Event
27. DCD      I2C1_ER_IRQHandler      ; I2C1 Error
28. DCD      I2C2_EV_IRQHandler      ; I2C2 Event
29. DCD      I2C2_ER_IRQHandler      ; I2C2 Error
30. DCD      SPI1_IRQHandler         ; SPI1
31. DCD      SPI2_IRQHandler         ; SPI2
32. DCD      USART1_IRQHandler       ; USART1
33. DCD      USART2_IRQHandler       ; USART2
34. DCD      USART3_IRQHandler       ; USART3
35. DCD      EXTI15_10_IRQHandler     ; EXTI Line 15..10
```

第 18 行，为 *EXTI9\_5\_IRQHandler*，表示为 *EXTI9~EXTI5* 中断向量的服务函数名。

于是，我们就可以在 *stm32f10x\_it.c* 文件中加入名为 *EXTI9\_5\_IRQHandler()* 的函数：

```
1.  /* I/O 线中断，中断线为 PE5 */
2.  void EXTI9_5_IRQHandler(void)
3.  {
4.      if(EXTI_GetITStatus(EXTI_Line5) != RESET) //确保是否产生了 EXTI Line
           中断
5.      {
6.          // LED1 取反
7.          GPIO_WriteBit(GPIOC, GPIO_Pin_3,
8.              (BitAction)((1-
           GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
9.          EXTI_ClearITPendingBit(EXTI_Line5);    //清除中断标志位
10.     }
11. }
```

其内容比较容易理解，进入中断后，调用库函数 *EXTI\_GetITStatus()* 来重新检查是否产生了 *EXTI\_Line* 中断，接下来把 LED 取反，操作完毕后，调用 *EXTI\_ClearITPendingBit()* 清除中断标志位再退出中断服务函数。这两个函数的解释见图 6-6 及图 6-7。



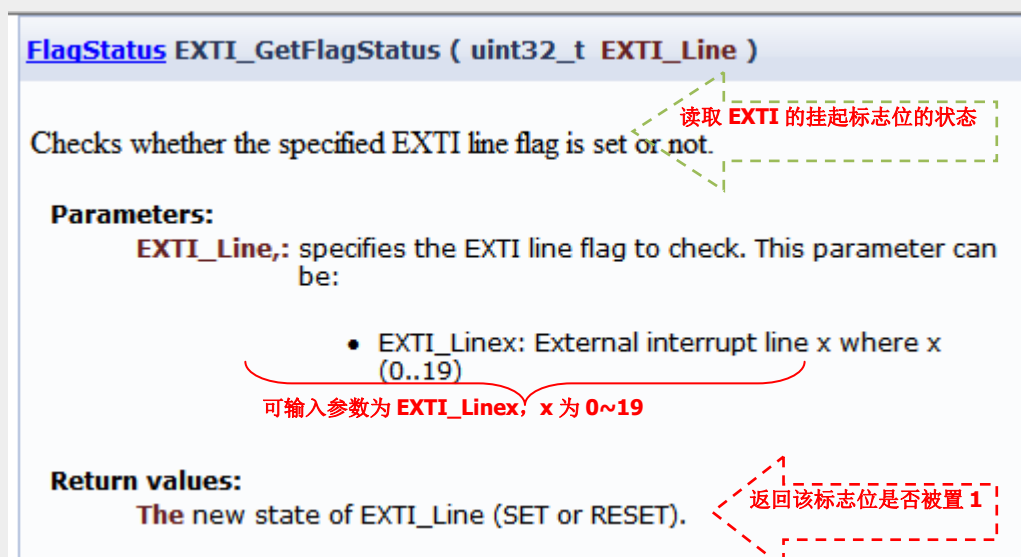


图 6-6EXTI 状态检查函数

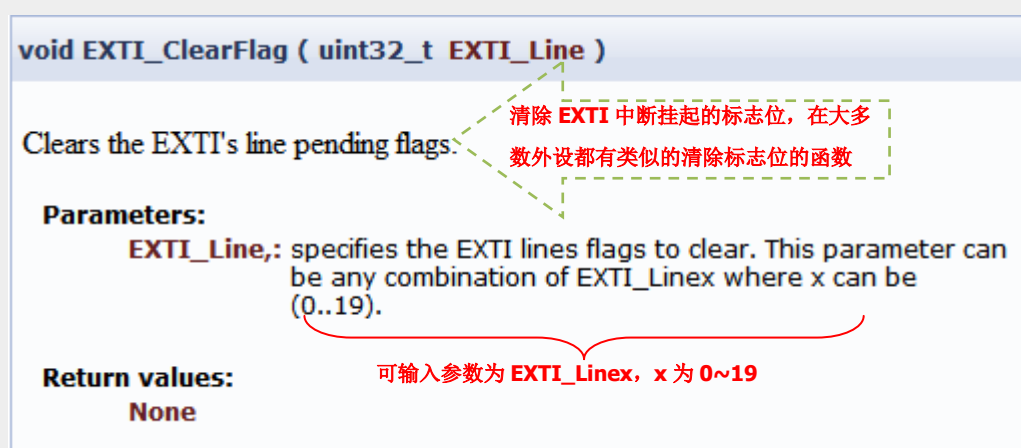


图 6-7 EXTI 清除标志位函数

这两种函数在 ST 库函数非常见，当我们要读取某外设的状态时，可调用该外设的 XXX\_GetFlagStatus()函数来获取该状态。一般也有 XXX\_ClearFlag()库函数可供调用，进行相应的标志位清除。

中断服务程序比较简单，很容易读懂，但我们在写中断函数入口的时候要注意函数名的写法，函数名只有两种命名方法：

```
1-> EXTI0_IRQHandler      ; EXTI Line 0
    EXTI1_IRQHandler      ; EXTI Line 1
    EXTI2_IRQHandler      ; EXTI Line 2
    EXTI3_IRQHandler      ; EXTI Line 3
    EXTI4_IRQHandler      ; EXTI Line 4
2-> EXTI9_5_IRQHandler    ; EXTI Line 9..5
```

```
EXTI15_10_IRQHandler ; EXTI Line 15..10
```

只要是中断线在 5 之后的就不能像 0~4 那样单独一个函数名，都必须写成 *EXTI9\_5\_IRQHandler* 和 *EXTI15\_10\_IRQHandler*。假如写成 *EXTI5\_IRQHandler*、*EXTI6\_IRQHandler*.....*EXTI15\_IRQHandler* 这样的话编译器是不会报错的，只是中断服务程序不能工作罢了。如果你不知道的话，会让你搞半天也不知问题出现在哪。

#### 6.4.9. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板，LED1 亮，按下按键时 LED1 灭，再按下按键时 LED1 亮，如此循环。

## 7. SysTick (系统定时器)

### 7.1. SysTick——操作系统的心跳

SysTick 定时器被捆绑在 NVIC 中，用于产生 SysTick 异常（异常号：15）。在以前，操作系统和有所有使用了时基的系统，都必须需要一个硬件定时器来产生需要的“滴答”中断，作为整个系统的时基。滴答中断对操作系统尤其重要。例如，操作系统可以为多个任务许以不同数目的时间片，确保没有一个任务能霸占系统；或者把每个定时器周期的某个时间范围赐予特定的任务等，还有操作系统提供的各种定时功能，都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，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。

Cortex-M3 在 *内核部分* 包含了一个简单的定时器——*SysTick timer*。因为所有的 CM3 芯片都带有这个定时器，软件在不同芯片生产厂商的 CM3 器件间的移植工作就得以化简。该定时器的时钟源可以是内部时钟（FCLK，CM3 上的自由运行时钟），或者是外部时钟（CM3 处理器上的 STCLK 信号）。不过，STCLK 的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能会大不相同。因此，需要阅读芯片的使用手册来确定选择什么作为时钟源。

在 STM32 中 SysTick 以 *HCLK(AHB 时钟)* 或 *HCLK/8* 作为运行时钟。见图 6-1。

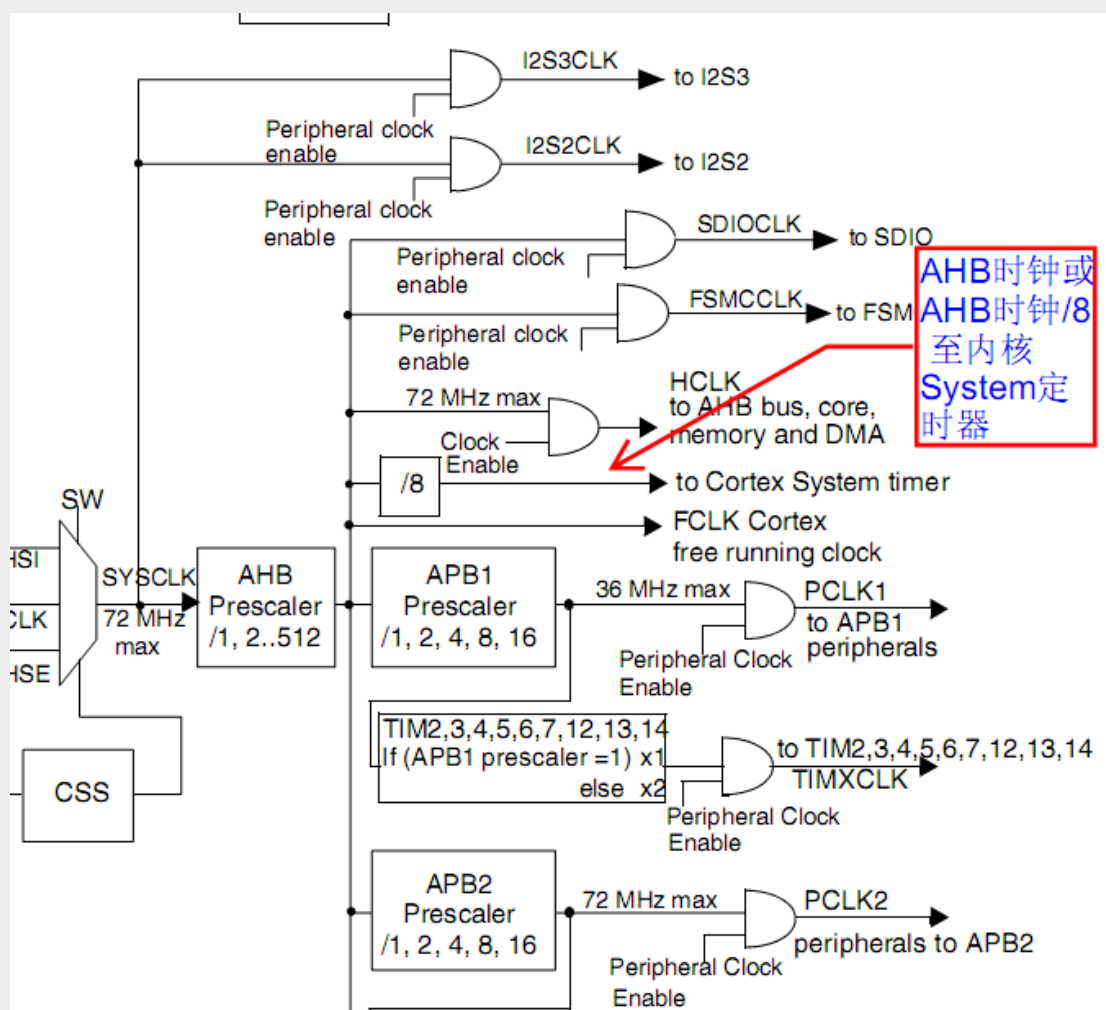


图 6-1 时钟树（部分）-SysTick timer 时钟来源

SysTick 定时器能产生中断，CM3 为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其它系统软件在 CM3 器件间的移植变得简单多了，因为在所有 CM3 产品间，SysTick 的处理方式都是相同的。

*SysTick 定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。*

SysTick 定时器属于 cortex 内核部件，可以参考《CortexM3 权威指南》或《STM32xxx-Cortex 编程手册》来了解

## 7.2. SysTick timer 工作分析

SysTick 是一个 24 位的定时器，即一次最多可以计数  $2^{24}$  个时钟脉冲，这个脉冲计数值被保存到 当前计数值寄存器 *STK\_VAL* (*SysTick current value*)

register) 中，只能向下计数，每接收到一个时钟脉冲 *STK\_VAL* 的值就向下减 1，直至 0，当 *STK\_VAL* 的值被减至 0 时，由硬件自动把 *重载寄存器 STK\_LOAD (SysTick reload value register)* 中保存的数据加载到 *STK\_VAL*，重新向下计数。当 *STK\_VAL* 的值被计数至 0 时，触发异常，就可以在中断服务函数中处理定时事件了。

当然，要使 SysTick 进行以上工作必须要进行 SysTick 进行配置。它的控制配置很简单，只有三个控制位和一个标志位，都位于寄存器 *STK\_CTRL (SysTick control and status register)* 中，见图 6-。

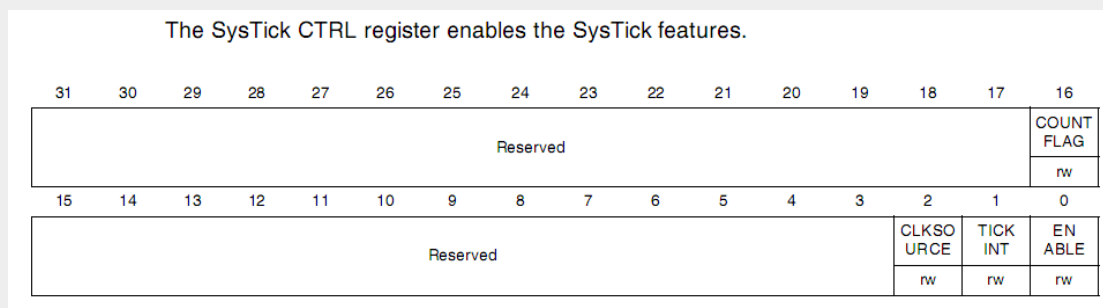


图 6-2 SysTick CTRL 寄存器

Bit0: *ENABLE*

为 *SysTick timer* 的使能位，此位为 1 的时候使能 SysTick timer，此位为 0 的时候关闭 SysTick timer。

Bit1: *TICKINT*

为异常触发使能位，此位为 1 的时候并且 *STK\_VAL* 计数至 0 时会触发 SysTick 异常，此位被配置为 0 的时候不触发异常

Bit2: *CLKSOURCE*

为 *SysTick* 的时钟选择位，此位为 1 的时候 SysTick 的时钟为 AHB 时钟，此位为 0 的时候 SysTick 时钟为 AHB/8（AHB 的八分频）。

Bit16: *COUNTFLAG*

为计数为 0 标志位，若 *STK\_VAL* 计数至 0，此标志位会被置 1。

与 SysTick 控制相关的所有寄存器如图 7-2，其中上面没有介绍的 *STK\_CALIB* 寄存器是用于校准的，不常用。

| Offset | Register                 | 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      |
|--------|--------------------------|----------|----|----|----|----|----|----|----|----|---------------|----|----|----|----|----|-----------|----------|----|----|----|----|----|---|---|---|---|---|---|---|-----------|----------|--------|
| 0x00   | STK_CTRL<br>Reset Value  | Reserved |    |    |    |    |    |    |    |    |               |    |    |    |    |    | COUNTFLAG | Reserved |    |    |    |    |    |   |   |   |   |   |   |   | CLKSOURCE | TICK INT | ENABLE |
| 0x04   | STK_LOAD<br>Reset Value  | Reserved |    |    |    |    |    |    |    |    | RELOAD[23:0]  |    |    |    |    |    |           |          |    |    |    |    |    |   |   |   |   |   | 0 | 0 |           |          |        |
| 0x08   | STK_VAL<br>Reset Value   | Reserved |    |    |    |    |    |    |    |    | CURRENT[23:0] |    |    |    |    |    |           |          |    |    |    |    |    |   |   |   |   |   | 0 | 0 |           |          |        |
| 0x0C   | STK_CALIB<br>Reset Value | Reserved |    |    |    |    |    |    |    |    | TENMS[23:0]   |    |    |    |    |    |           |          |    |    |    |    |    |   |   |   |   |   | 0 | 0 |           |          |        |

图 7-2 SysTick 寄存器映像

### 7.3. SysTick 精确延时实例精讲

前面的实验例程中，当有延时需要的时候，我们都是利用内核循环执行变量自减的代码来实现，延时的时间无法精确测量，有很大的局限性，当我们需要精确延时时，就可以利用 SysTick timer 实现，理论上它的最小计时单位为 AHB 的时钟周期，即  $1/72000000$  秒，72 分之一的微秒，足以满足大部分极端应用需求。本小节以实例讲解如何利用 SysTick 进行精确延时。

#### 7.3.1. 实验描述及工程文件清单

|         |                                                                                                                                                                    |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | 3 个 LED 在 SysTick 的控制下，以 500ms 的频率闪烁。                                                                                                                              |
| 硬件连接    | PC3 – LED1、PC4 – LED2、PC5 – LED3                                                                                                                                   |
| 用到的库文件  | <i>startup/start_stm32f10x_hd.c</i><br><i>CMSIS/core_cm3.c</i><br><i>CMSIS/system_stm32f10x.c</i><br><i>FWlib/stm32f10x_gpio.c</i><br><i>FWlib/stm32f10x_rcc.c</i> |
| 用户编写的文件 | <i>USER/main.c</i><br><i>USER/stm32f10x_it.c</i><br><i>USER/led.c</i><br><i>USER/led.h</i><br><i>USER/SysTick.c</i>                                                |

### 7.3.2. 配置工程环境

本 SysTick timer 精确延时实验中我们用到了 *GPIO*、*RCC* 外设，所以我们先要把以下库文件添加到工程 *stm32f10x\_gpio.c*、*stm32f10x\_rcc.c*。

由于本实验中，SysTick 的中断是在文件 *core\_cm3.h* 的函数配置的，没有使用 NVIC 来配置中断，所以可不添加 *misc.c* 文件。而 *core\_cm3.h* 在包含 *stm32f10x.h* 头文件时已被添加进工程了。

接下来添加旧工程中的外设用户文件 *led.c*，新建 *SysTick.c* 及 *SysTick.h* 文件，并在 *stm32f10x\_conf.h* 中把使用到的 ST 库的头文件注释去掉。

```
1.  /**
2.  ****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  ****
9.
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
```

### 7.3.3. main 文件

我们从看 main 函数看起：

```
1.  /*
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      /* LED 端口初始化 */
10.     LED_GPIO_Config();
11.
12.     /* 配置 SysTick 为 10us 中断一次 */
13.     SysTick_Init();
14.
15.     for(;;)
16.     {
17.
18.         LED1( 0 );
19.         Delay_us(50000);           // 50000 * 10us = 500ms
20.         LED1( 1 );
21.
22.         LED2( 0 );
```

```
23.         Delay_us(50000);           // 50000 * 10us = 500ms
24.         LED2( 1 );
25.
26.         LED3( 0 );
27.         Delay_us(50000);           // 50000 * 10us = 500ms
28.         LED3( 1 );
29.
30.     }
31.
32. }
```

在 `main` 函数中，我们只见到 `SysTick_Init()` 和 `Delay_us()` 这两个函数比较陌生，它们的功能分别是配置好 `SysTick` 定时器和进行精确延时。

整个 `main` 函数的流程就是先初始化好 LED 及 `SysTick` 定时器之后，就进入死循环，轮流点亮 LED1、LED2、LED3，点亮的时间为精确的 500ms。

#### 7.3.4. 配置并启动 SysTick timer

接下来我们看一下 `SysTick_Init()` 这个函数，它是由用户在 `SysTick.c` 这个文件中实现的，其功能是启动系统滴答定时器 `SysTick`，并将 `SysTick` 配置为 10 us 中断一次：

```
1.  /*
2.   * 函数名: SysTick_Init
3.   * 描述   : 启动系统滴答定时器 SysTick
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8. void SysTick_Init(void)
9. {
10.     /* SystemFrequency / 1000    1ms 中断一次
11.      * SystemFrequency / 100000   10us 中断一次
12.      * SystemFrequency / 1000000  1us 中断一次
13.      */
14. // if (SysTick_Config(SystemFrequency / 100000)) // ST3.0.0 库版本
15. if (SysTick_Config(SystemCoreClock / 100000)) // ST3.5.0 库版本
16. {
17.     /* Capture error */
18.     while (1);
19. }
20.     // 关闭滴答定时器
21.     SysTick->CTRL &= ~ SysTick_CTRL_ENABLE_Msk;
22. }
```

本函数实际上只是调用了 `SysTick_Config()` 函数，它是属于内核层的 Cortex-M3 通用函数，位于 `core_cm3.h` 文件中，若调用 `SysTick_Config()` 配置



SysTick 不成功，则进入死循环，初始化 SysTick 成功后，先关闭定时器，在需要的时候再开启。

`SysTick_Config()` 函数无法在《STM32 外设固件库帮助手册.chm》文件中找到其使用方法。所以我们在 keil 环境下直接跟踪这个函数到 `core_cm3.h` 文件，查看函数的定义：

```
1. /**
2.  * @brief Initialize and start the SysTick counter and its interrupt.
3.  *
4.  * @param ticks number of ticks between two interrupts
5.  * @return 1 = failed, 0 = successful
6.  *
7.  * Initialise the system tick timer and its interrupt and start the
8.  * system tick timer / counter in free running mode to generate
9.  * periodical interrupts.
10. */
11. static __INLINE uint32_t SysTick_Config(uint32_t ticks)
12. {
13.     /* Reload value impossible */
14.     if (ticks > SysTick_LOAD_RELOAD_Msk) return (1);
15.
16.     /* set reload register */
17.     SysTick->LOAD = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;
18.
19.     /* set Priority for Cortex-M0 System Interrupts */
20.     NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
21.
22.     /* Load the SysTick Counter Value */
23.     SysTick->VAL = 0;
24.
25.     SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
26.                     SysTick_CTRL_TICKINT_Msk |
27.                     SysTick_CTRL_ENABLE_Msk; /* Enab
28.     le SysTick IRQ and SysTick Timer */
29.     return (0); /* Func
30.     tion successful */
31. }
```

在这个函数定义的前面，有关于它的注释，如果我们不想去研究它的具体实现，可以根据这段注释了解函数的功能：这个函数启动了 SysTick timer；并把它配置为计数至 0 时引起中断；输入的参数 ticks 为两个中断之间的脉冲数，即相隔 ticks 个时钟周期会引起一次中断；配置 SysTick 成功时返回 0，出错返回 1。

但是，这段注释并没有告诉我们它把 SysTick 的时钟设置为 AHB 时钟还是 AHB/8，这是一个十分关键的问题，于是，野火对这个函数的具体实现进行分析，与大家再分享一下如何分析底层库函数。

分析底层库函数，要有 7.2 小节关于 SysTick timer 工作分析的知识准备。

## ■ 检查输入参数

*SysTick\_Config()*第 1 行代码是检查输入参数 *ticks*，因为 *ticks* 是脉冲计数值，要被保存到 *重载寄存器 STK\_LOAD* 寄存器中，再由硬件把 *STK\_LOAD* 值加载到 *当前计数值寄存器 STK\_VAL* 使用的，*STK\_LOAD* 和 *STK\_VAL* 都是 24 位的，所以当输入参数 *ticks* 大于其可存储的最大值时，将由这行代码检查出错误返回。

## ■ 位指示宏及位屏蔽宏

检查 *ticks* 参数没有错误后，就稍稍处理一下把 *ticks-1* 赋值给 *STK\_LOAD* 寄存器，要注意的是 *减 1*，若 *STK\_VAL* 从 *ticks-1* 向下计数至 0，实际上就经过了 *ticks* 个脉冲。这句赋值代码中使用到了宏 *SysTick\_LOAD\_RELOAD\_Msk*，与其它库函数类似，这个宏是用来指示寄存器的 *特定位置* 或进行 *位屏蔽* 用的。它及类似的宏定义如下：

```
1. /* SysTick Control / Status Register Definitions */
2. #define SysTick_CTRL_COUNTFLAG_Pos      16
   /*!< SysTick CTRL: COUNTFLAG Position */
3. #define SysTick_CTRL_COUNTFLAG_Msk      (1ul << SysTick_CTRL_COUNTF
   LAG_Pos)
   /*!< SysTick CTRL: COUNTFLAG Mask */
4.
5. #define SysTick_CTRL_CLKSOURCE_Pos      2
   /*!< SysTick CTRL: CLKSOURCE Position */
6. #define SysTick_CTRL_CLKSOURCE_Msk      (1ul << SysTick_CTRL_CLKSOU
   RCE_Pos)
   /*!< SysTick CTRL: CLKSOURCE Mask */
7.
8. #define SysTick_CTRL_TICKINT_Pos        1
   /*!< SysTick CTRL: TICKINT Position */
9. #define SysTick_CTRL_TICKINT_Msk        (1ul << SysTick_CTRL_TICKIN
   T_Pos)
   /*!< SysTick CTRL: TICKINT Mask */
10.
11. #define SysTick_CTRL_ENABLE_Pos         0
   /*!< SysTick CTRL: ENABLE Position */
12. #define SysTick_CTRL_ENABLE_Msk         (1ul << SysTick_CTRL_ENABLE
   _Pos)
   /*!< SysTick CTRL: ENABLE Mask */
13.
14. /* SysTick Reload Register Definitions */
15. #define SysTick_LOAD_RELOAD_Pos         0
   /*!< SysTick LOAD: RELOAD Position */
16. #define SysTick_LOAD_RELOAD_Msk         (0xFFFFFul << SysTick_LOAD
   _RELOAD_Pos)
   /*!< SysTick LOAD: RELOAD Mask */
17.
```

```
18. /* SysTick Current Register Definitions */
19. #define SysTick_VAL_CURRENT_Pos          0
    /*!< SysTick VAL: CURRENT Position */
20. #define SysTick_VAL_CURRENT_Msk          (0xFFFFFul << SysTick_VAL_
    CURRENT_Pos) /*!< SysTick VAL: CURRENT Mask */
21.
22. /* SysTick Calibration Register Definitions */
23. #define SysTick_CALIB_NOREF_Pos          31
    /*!< SysTick CALIB: NOREF Position */
24. #define SysTick_CALIB_NOREF_Msk          (1ul << SysTick_CALIB_NOREF_
    _Pos) /*!< SysTick CALIB: NOREF Mask */
25.
26. #define SysTick_CALIB_SKEW_Pos          30
    /*!< SysTick CALIB: SKEW Position */
27. #define SysTick_CALIB_SKEW_Msk          (1ul << SysTick_CALIB_SKEW_
    Pos) /*!< SysTick CALIB: SKEW Mask */
28.
29. #define SysTick_CALIB_TENMS_Pos          0
    /*!< SysTick CALIB: TENMS Position */
30. #define SysTick_CALIB_TENMS_Msk          (0xFFFFFul << SysTick_VAL_
    CURRENT_Pos) /*!< SysTick CALIB: TENMS Mask */
31. /* @} */ /* end of group CMSIS_CM3_SysTick */
```

其中的寄存器位指示宏: *SysTick\_xxx\_Pos* , 宏展开后即为 **xxx** 在相应寄存器中的位置, 如控制 SysTick 时钟源的 *SysTick\_CTRL\_CLKSOURCE\_Pos* , 宏展开为 **2** , 这个寄存器位正是在寄存器 *STK\_CTRL* 中的 *Bit2*。

而寄存器位屏蔽宏: *SysTick\_xxx\_Msk* , 宏展开是 **xxx** 的位全部置 1 后, 左移 *SysTick\_xxx\_Pos* 位。如控制 SysTick 时钟源的 *SysTick\_CTRL\_CLKSOURCE\_Msk* , 宏展开为 *(1ul << SysTick\_CTRL\_CLKSOURCE\_Pos)* , 把无符号长整型数值(**ul**) 1 左移 2 位, 得到了一个只有 *Bit2:CLKSOURCE* 位被置 1, 其它位为 0 的数值, 这样的数值配合位操作 **&**(按位与)、**|** (按位或)可以很方便地修改寄存器的某些位。假如控制 *CLKSOURCE* 需要 **四个**寄存器位, 这个宏就应该被改为 *(0xf ul << SysTick\_CTRL\_CLKSOURCE\_Pos)* , 这样就会得到一个关于 *CLKSOURCE* 的四位被置 1 的值, 这些宏的参数就是这样被确定的。

寄存器位指示宏和位屏蔽宏在操作寄存器的代码(大部分库函数)中用得十分广泛, 在前面 *GPIO\_Init()*函数分析时也遇到很多, 为了方便以后再使用, 野火就给这两类宏取了这两个名字。

## ■ 配置中断向量及重置 STK\_VAL 寄存器

回到 *SysTick\_Config()* 函数，接下来调用了 *NVIC\_SetPriority()* 函数配置了 SysTick 中断，这就是为什么我们在外部没有再使用 *NVIC* 配置 SysTick 中断的原因。配置好 SysTick 中断后把 *STK\_VAL* 寄存器重新赋值为 0（在使能 SysTick 时，硬件会把存储在 STK\_LOAD 寄存器中的 ticks 值加载给它）。

## ■ 配置 SysTick timer 时钟为 AHB

在这段代码最后，向 *STK\_CTRL* 寄存器写入了 SysTick timer 的控制参数，配置为使用 AHB 时钟，使能计数至 0 时引起中断，使能 SysTick。执行了这行代码，SysTick 就开始运行，进行脉冲计数了。

若读者想要使用 AHB/8 作为时钟，可以调用库函数 *SysTick\_CLKSourceConfig()* 进行修改，也可以直接对 *SysTick\_Config()* 函数的代码进行修改。

## ■ 使能、关闭定时器

由于调用 *SysTick\_Config()* 函数之后，SysTick 定时器就被开启了，但我们在初始化的时候并不希望这样，而是在需要的时候再开启。所以在 *SysTick\_Init()* 函数中，调用完 *SysTick\_Config()* 配置好后先把定时器关闭了。SysTick 的开启和关闭由寄存器 *STK\_CTRL* 的 Bit0: ENABLE 位来控制，使用位屏蔽宏，以操作寄存器的方式实现：

```
1. // 使能滴答定时器
2. SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
3. // 失能滴答定时器
4. SysTick->CTRL &= ~ SysTick_CTRL_ENABLE_Msk;
```

### 7.3.5. 定时时间的计算

现在回到函数 `SysTick_Init()`，在调用 `SysTick_Config()`函数时，向它输入的参数为：`SystemCoreClock / 100000`，`SystemCoreClock` 为定义了系统时钟（`SYSCCLK`）频率的宏，即等于 AHB 的时钟频率，本书的所有例程中 AHB 都是被配置为 72MHz 的，也就是这个 `SystemCoreClock` 宏展开为数值 `7200 0000`。

根据前面对 `SysTick_Config()`函数的介绍，它的输入参数为 SysTick 将要计的脉冲数，经过 `ticks` 个脉冲(经过 `ticks` 个时钟周期)后将触发中断，触发中断后又重新开始计数。

由此我们可以算出定时的时间，下面为计算公式：

$$T = \text{ticks} * (1/f)$$

$T$  为要定时的总时间。

`ticks` 为 `SysTick_Config()`的输入参数。

$1/f$  即为 SysTick timer 使用的时钟源的 **时钟周期**， $f$  为该时钟源的时钟频率，当时钟源确定后为常数。

例如:本实验例子中，使用时钟源为 AHB 时钟，其频率被配置为 72MHz。调用函数时，把 `ticks` 赋值为 `ticks=SystemFrequency / 10 000 =720`，表示 720 个时钟周期中断一次；（ $1/f$ ）是时钟周期的时间，此时（ $1/f = 1/72 \text{ us}$ ），所以最终定时总时间  $T=720*(1/72)$ ，为 720 个时钟周期，正好是 10us。

`SysTick` 定时器的定时时间(配置为触发中断，即为 **中断周期**)，由 `ticks` 参数决定，最大定时周期不能超过  $2^{24}$  个。以下是几种常用的中断周期配置，就是根据上面的公式计算出来的。

```
1. /* ticks 常取以下值 */
2. SystemFrequency / 1000           // 1ms 中断一次
3. SystemFrequency / 100000        // 10us 中断一次
4. SystemFrequency / 1000000       // 1us 中断一次
```

### 7.3.6. 编写中断服务函数

回到 `main` 函数，我们使 LED 工作在一个无限循环中，在 LED 的开与关之间调用了 `Delay_us()` 函数：

```
1. while (1)
2. {
3.     //SysTick->CTRL = 1 << SYSTICK_ENABLE;           // 使能滴答定时器
4.     LED1( 0 );
5.     Delay_us(50000);    // 50000 * 10us = 500ms
6.     LED1( 1 );
7.
8.     LED2( 0 );
9.     Delay_us(50000);    // 50000 * 10us = 500ms
10.    LED2( 1 );
11.
12.    LED3( 0 );
13.    Delay_us(50000);    // 50000 * 10us = 500ms
14.    LED3( 1 );
15.    //SysTick->CTRL = 0 << SYSTICK_ENABLE;           // 失能滴答定时器
16. }
```

一旦我们调用了 `Delay_us()` 函数，SysTick 定时器就被开启，按照设定好的定时周期递减计数，SysTick 的计数寄存器里面的值减为 0 时，就进入中断函数，当中断函数执行完毕之后由重新计时，如此循环，除非它被关闭。

`Delay_us()` 函数实现如下：

```
1. /*
2.  * 函数名: Delay_us
3.  * 描述   : us 延时程序,10us 为一个单位
4.  * 输入   : - nTime
5.  * 输出   : 无
6.  * 调用   : Delay_us( 1 ) 则实现的延时为 1 * 10us = 10us
7.  *       : 外部调用
8.  */
9.
10. void Delay_us(__IO u32 nTime)
11. {
12.     TimingDelay = nTime;
13.
14.     // 使能滴答定时器
15.     SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
16.
17.     while(TimingDelay != 0);
18. }
```

使能了 SysTick 之后，就使用 `while(TimingDelay != 0)` 语句等待 `TimingDelay` 变量变为 0，这个变量是在中断服务函数中被修改的。

因此，我们需要编写相应的中断服务程序，在本实验室中我们配置为 10us 中断一次，每次中断把 *TimingDelay* 减 1。中断程序在 *stm32f10x\_it.c* 中实现：

```
1. /**
2.  * @brief This function handles SysTick Handler.
3.  * @param None
4.  * @retval : None
5.  */
6. void SysTick_Handler(void)
7. {
8.     TimingDelay_Decrement();
9. }
```

SysTick 中断属于系统异常向量，在 *stm32f10x\_it.c* 文件中已经默认有了它的中断服务函数 *SysTick\_Handler()*，但内容为空。我们在找到这个函数，在里面调用了用户函数 *TimingDelay\_Decrement()*。

*TimingDelay\_Decrement()* 是由用户编写的一个应用程序，在 *SysTick.c* 中实现：

```
1. /*
2.  * 函数名: TimingDelay_Decrement
3.  * 描述   : 获取节拍程序
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 在 SysTick 中断函数 SysTick_Handler() 调用
7.  */
8. void TimingDelay_Decrement(void)
9. {
10.     if (TimingDelay != 0x00)
11.     {
12.         TimingDelay--;
13.     }
14. }
```

每次进入 SysTick 中断就调用一次 *TimingDelay\_Decrement()* 函数，把全局变量 *TimingDelay* 自减一次。用户函数 *Delay\_us ()* 在 *TimingDelay* 被减至等于 0 时，才退出延时循环，即我们对 *TimingDelay* 赋的值为要中断的次数。

所以总的延时时间  $T_{\text{延时}} = T_{\text{中断周期}} * \text{TimingDelay}$ 。

至此，SysTick 的精确延时功能讲解完毕。

### 7.3.7. 使用 SysTick 的测量时间的功能

稍微改变一下用法，我们就可以利用 SysTick 进行时间测量。

当我们开启 SysTick 定时器后，定时器开始工作，我们可以定义一个变量 `a` 来对中断次数进行记录，在定时器进入中断时，这个变量就 `a++`，当我们关闭定时器后，将变量的数值乘与定时器的中断周期就等于测量时间。这个功能野火一般用于测量程序的运行时间，特别是涉及到算法的程序，这对于优化算法是有非常大的帮助。假如你的算法的是 `us` 级别的，那么 SysTick 就应该设定为 `us` 级中断，如果是 `ms` 级别的，就将 SysTick 设定为 `ms` 级中断。

### 7.3.8. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板，即可看到板载的 3 个 LED 以 500ms 的频率闪烁。



## 8. 串口通信

当我们在学习一款 CPU 的时候，最经典的实验莫过于流水灯了，会了流水灯的话就基本等于学会操作 I/O 口了。那么在学会操作 I/O 之后，面对那么多的片上外设我们又应该先学什么呢？有些朋友会说用到什么就学什么，听起来这也不无道理呀。

但对于野火来说会把学习串口的操作放在第二位。在程序运行的时候我们可以通过点亮一个 LED 来显示代码的执行的状态，但有时候我们还想把某些中间量或者其他程序状态信息打印出来显示在电脑上，那么这时串口的作用就可想而知了。

### 8.1. 异步串口通讯协议

阅读过《STM32 中文参考手册》的读者会发现，STM32 的串口非常强大，它不仅支持最基本的通用串口同步、异步通讯，还具有 LIN 总线功能(局域互联网)、IRDA 功能(红外通讯)、SmartCard 功能。

为实现最迫切的需求，利用串口来帮助我们调试程序，本章介绍的为串口最基本、最常用的方法，全双工、异步通讯方式。图 1-1 为串口异步通讯协议。

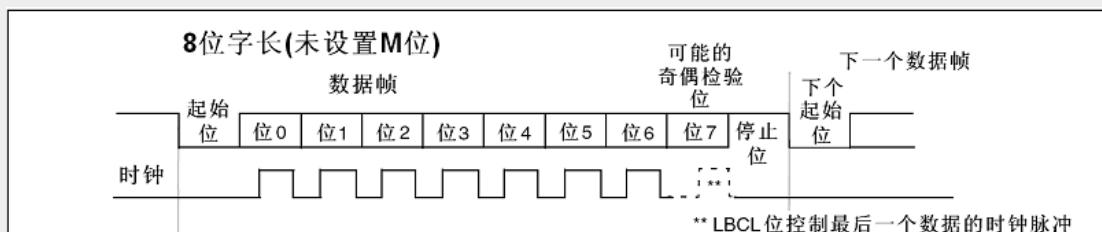


图 1-1 异步串口通讯协议

重温串口的通讯协议，我们知道要配置串口通讯，至少要设置以下几个参数：**字长**(一次传送的数据长度)、**波特率**(每秒传输的数据位数)、**奇偶校验位**、**还有停止位**。对 ST 库函数的使用已经上手的读者应该能猜到，在初始化串口的时候，必然有一个**串口初始化结构体**，这个结构体的几个成员肯定就是用来存储这些控制参数的。

## 8.2. 直通线和交叉线

## 野火 STM32 开发板串口硬件原理图

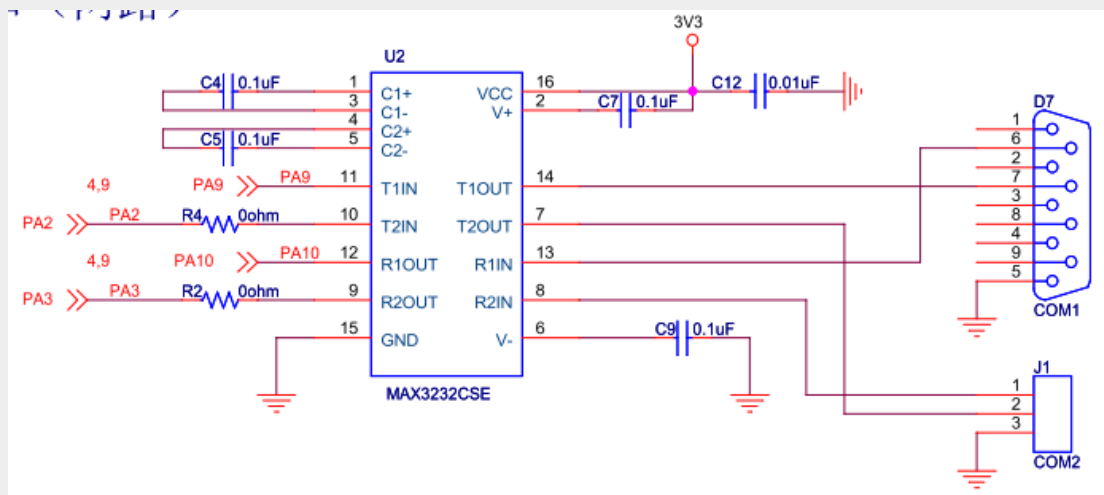


图 1-2 野火开发板串口硬件图

见图 1-2，这是野火 STM32 开发板的接线图，使用的为 MAX3232 芯片，把 STM32 的 *PA10* 引脚(复用功能为 *USART1* 的 *Rx*)接到了 DB9 接口的第 2 针脚，把 *PA9* 引脚(复用功能为 *USART* 的 *Tx*)连接到了 DB9 接口的第 3 针脚。

*Tx* (发送端) 接第 3 针脚, *Rx* (接收端) 接第 2 针脚。这种接法是跟 PC 的串口接法一样的, 如果要实现 PC 跟野火板子通讯, 就要使用两头都是母的交叉线。

串口线主要分两种，**直通线（平行线）**和**交叉线**。它们的区别见图 1-3。

假如 PC 与板子之间要实现全双工串口通讯，必然是 PC 的 Tx 针脚要连接到板子的 Rx 针脚，而 PC 的 Rx 针脚则要连接至板子的 Tx 针脚了。由于板子和 pc 的串口接法是**相同的**，就要使用**交叉线**来连接了。如果有的开发板是 Tx 连接至 DB9 的第 2 针脚，而 Rx 连接至第 3 针脚，这与 PC 接法是**相反的**，这样的板子与 PC 通讯就需要使用**直通线**了。

## 为什么野火板子要使用 PC 的接法？

假如使用非 PC 接法，由于板子与 PC 的接法相反，通讯就要使用直通线；但两个板子之间想要进行串口通讯时，由于接法相同，就要使用交叉线。如果

使用 PC 接法，板子与 PC 之间接法相同，通讯使用交叉线；两个相同板子之间接法也相同，通讯也是使用交叉线。

所以野火建议大家设计板子时，尽量采用与 PC 相同的标准串口接法。

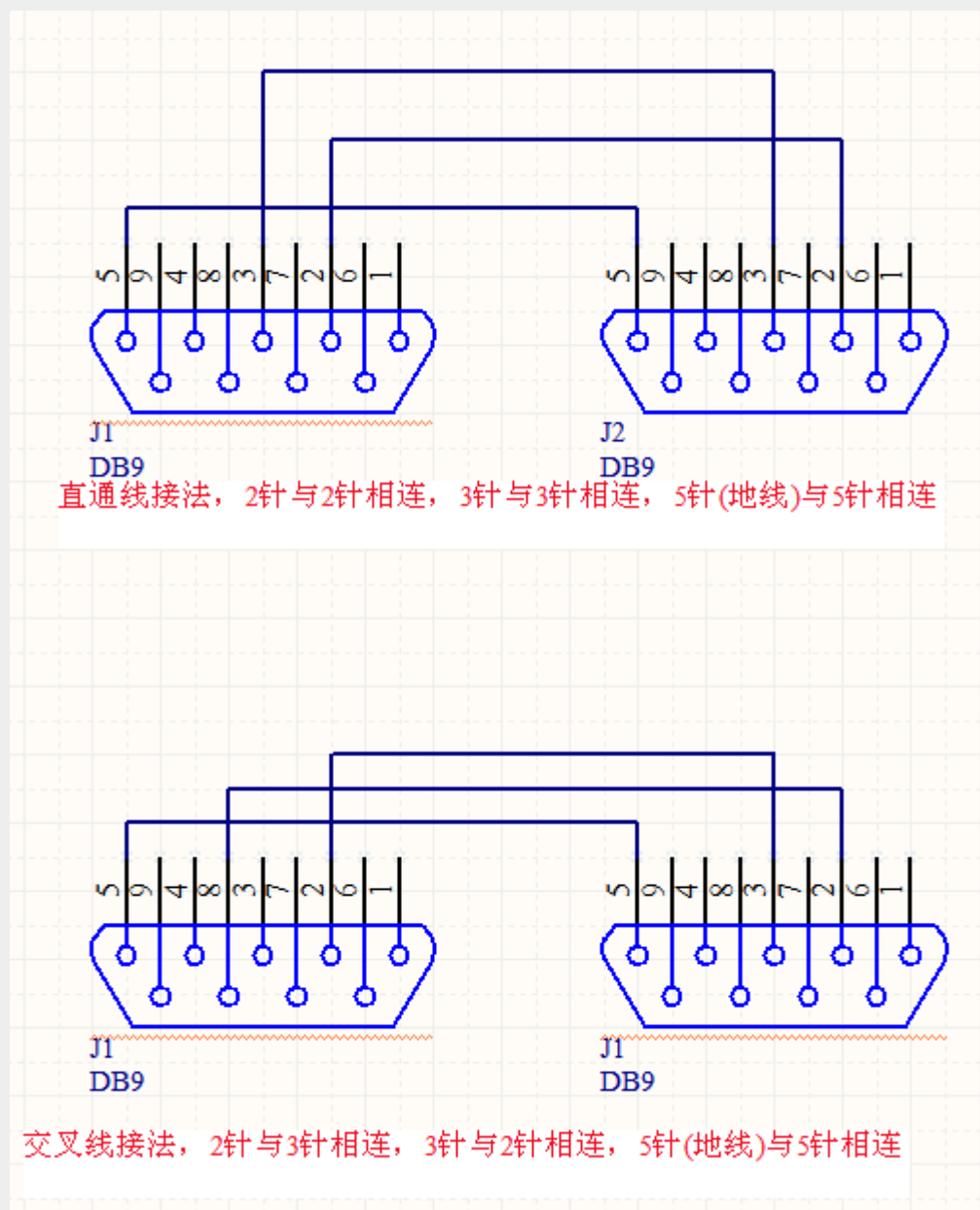


图 1-3 交叉线与直通线的区别

介绍直通线与交叉线的区别，一来是野火发现某些读者因为线的问题而花费了大量宝贵的时间。二来是介绍串口线的 DIY 方法。要实现基本的全双工异步通讯，只要 3 条线，分别为 *Rx*、*Tx*、和 *GND*。如果读者正在为直通线、交

叉线、公头、母头不匹配而烦恼，可以根据自己的需要，参照图 1-3，用三根杜邦线连接即可。

### 8.3. 串口工作过程分析

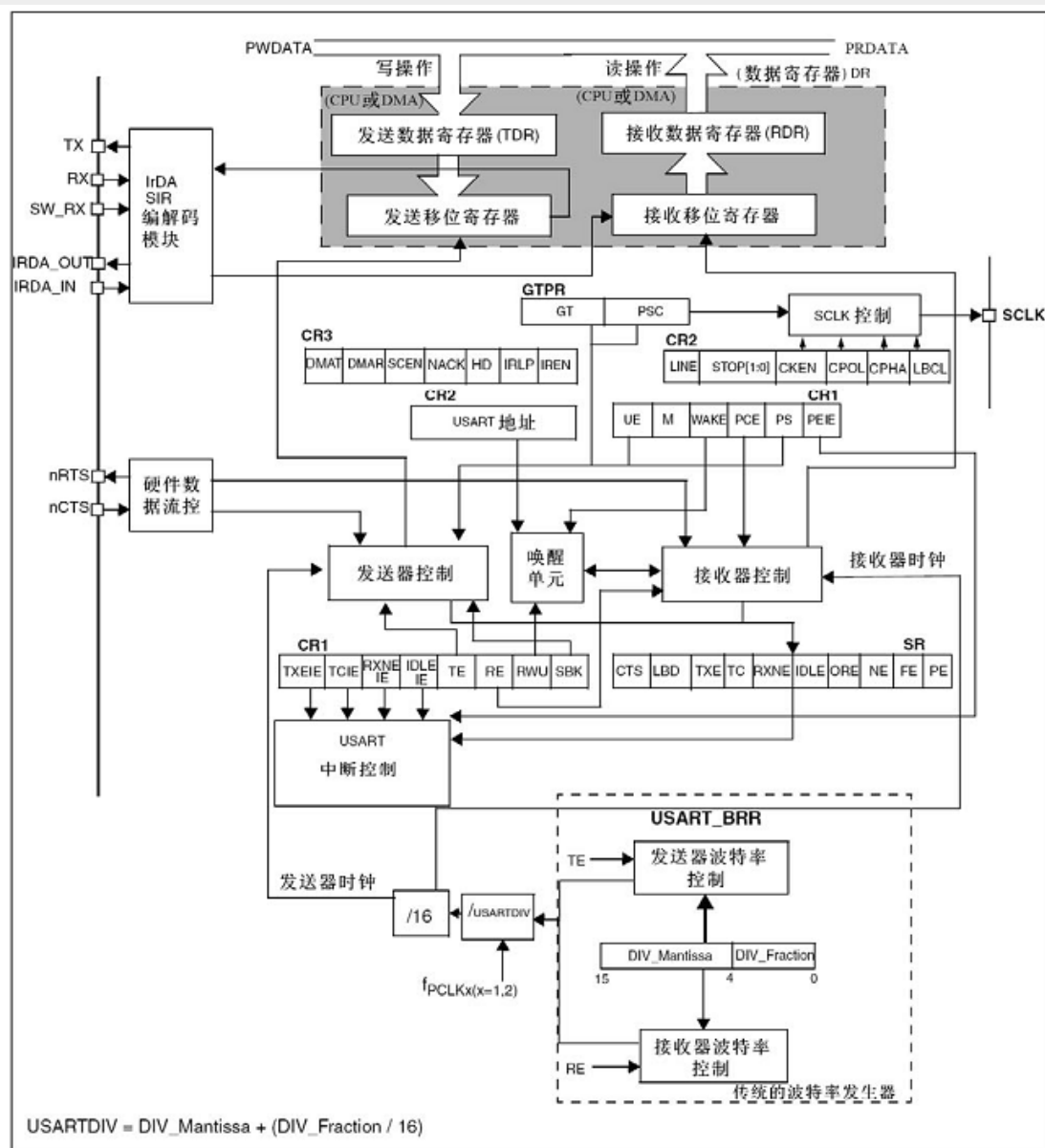


图 1-4 串口架构图

串口外设的架构图看起来十分复杂，实际上对于软件开发人员来说，我们只需要大概了解串口发送的过程即可。

从下至上，我们看到串口外设主要由三个部分组成，分别是**波特率的控制部分**、**收发控制部分**及**数据存储转移部分**。

### 8.3.1. 波特率控制

波特率，即每秒传输的二进制位数，用 b/s (bps)表示，通过对时钟的控制可以改变波特率。在配置波特率时，我们向波特比率寄存器 *USART\_BRR* 写入参数，修改了串口时钟的分频值 *USARTDIV*。*USART\_BRR* 寄存器包括两部分，分别是 *DIV\_Mantissa*(*USARTDIV* 的整数部分)和 *DIVFraction*(*USARTDIV* 的小数)部分，最终，计算公式为

$$USARTDIV = DIV\_Mantissa + (DIVFraction/16)。$$

*USARTDIV*是对串口外设的时钟源进行分频的，对于 *USART1*，由于它是挂载在 *APB2*总线上的，所以它的时钟源为  $f_{PCLK2}$ ；而 *USART2、3*挂载在 *APB1*上，时钟源则为  $f_{PCLK1}$ ，串口的时钟源经过 *USARTDIV*分频后分别输出作为发送器时钟及接收器时钟，控制发送和接收的时序。

### 8.3.2. 收发控制

围绕着发送器和接收器控制部分，有好多个寄存器：CR1、CR2、CR3、SR，即 *USART* 的三个控制寄存器(Control Register)及一个状态寄存器(Status Register)。通过向寄存器写入各种控制参数，来控制发送和接收，如奇偶校验位，停止位等，还包括对 *USART* 中断的控制；串口的状态在任何时候都可以从状态寄存器中查询得到。具体的控制和状态检查，我们都是使用库函数来实现的，在此就不具体分析这些寄存器位了。

### 8.3.3. 数据存储转移部分

收发控制器根据我们的寄存器配置，对数据存储转移部分的移位寄存器进行控制。

当我们需要发送数据时，内核或 DMA 外设(一种数据传输方式，在下一章介绍)把数据从内存(变量)写入到发送数据寄存器 *TDR*后，发送控制器将适时地自动把数据从 *TDR*加载到发送移位寄存器，然后通过串口线 *Tx*，把数据一

位一位地发送出去，在数据从 *TDR* 转移到 *移位寄存器* 时，会产生 *发送寄存器 TDR 已空事件 TXE*，当数据从 *移位寄存器* 全部发送出去时，会产生数据 *发送完成事件 TC*，这些事件可以在 *状态寄存器* 中查询到。

而 *接收数据* 则是一个 *逆过程*，数据从 *串口线 Rx* 一位一位地输入到 *接收移位寄存器*，然后自动地转移到 *接收数据寄存器 RDR*，最后用内核指令或 DMA 读取到内存(变量)中。

#### 8.4. 串口通讯实验分析

##### 8.4.1. 实验描述及工程文件清单

|         |                                                                                                                                                                                                      |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | 重新实现 C 库中的 <code>printf()</code> 函数到串口 1，这样我们就可以像用 C 库中的 <code>printf()</code> 函数一样将信息通过串口打印到电脑，非常方便我们程序的调试。                                                                                         |
| 硬件连接    | PA9 - USART1(Tx)<br>PA10 - USART1(Rx)                                                                                                                                                                |
| 用到的库文件  | <i>startup/start_stm32f10x_hd.c</i><br><i>CMSIS/core_cm3.c</i><br><i>CMSIS/system_stm32f10x.c</i><br><i>FWlib/stm32f10x_gpio.c</i><br><i>FWlib/stm32f10x_rcc.c</i><br><i>FWlib/stm32f10x_usart.c</i> |
| 用户编写的文件 | <i>USER/main.c</i><br><i>USER/stm32f10x_it.c</i><br><i>USER/usart1.c</i>                                                                                                                             |

### 8.4.2. 配置工程环境

串口实验中我们用到了 *GPIO*、*RCC*、*USART* 这三个外设的库文件 *stm32f10x\_gpio.c*、*stm32f10x\_rcc.c*、*stm32f10x\_usart.c*，所以我们要把这个库文件添加进工程，新建用户文件 *usart1.c*。并在 *stm32f10x\_conf.h* 中把相应的头文件的注释去掉。

```
1.  /**
2.  ****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  ****
9.
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "stm32f10x_usart.h"
```

### 8.4.3. main 文件

配置好要用的库的环境之后，我们就从 **main** 函数看起，层层剥离源代码。

```
1.  /**
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      /* USART1 config 115200 8-N-1 */
10.     USART1_Config();
11.
12.     printf("\r\n this is a printf demo \r\n");
13.
14.     printf("\r\n 欢迎使用野火 M3 实验板:) \r\n");
15.
16.     USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
17.
18.     USART1_printf(USART1, "\r\n ("__DATE__" -
19.     " __TIME__ ") \r\n");
20.     for(;;)
21.     {
22.
23.     }
24. }
```

首先调用函数 `USART1_Config()`, 函数 `USART1_Config()` 主要做了如下工作:

1. 使能了串口 1 的时钟
2. 配置好了 `usart1` 的 I/O
3. 配置好了 `usart1` 的工作模式, 具体为波特率为 115200、8 个数据位、1 个停止位、无硬件流控制。即 115200 8-N-1。

#### 8.4.4. USART 初始化配置

具体的 `USART1_Config()` 在 `usart1.c` 这个用户文件中实现:

```
1.  /*
2.  * 函数名: USART1_Config
3.  * 描述   : USART1 GPIO 配置,工作模式配置。115200 8-N-1
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void USART1_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     USART_InitTypeDef USART_InitStructure;
12.
13.     /* config USART1 clock */
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIO
A, ENABLE);
15.
16.     /* USART1 GPIO config */
17.     /* Configure USART1 Tx (PA.09) as alternate function push-
pull */
18.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
19.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
20.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21.     GPIO_Init(GPIOA, &GPIO_InitStructure);
22.     /* Configure USART1 Rx (PA.10) as input floating */
23.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
24.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
25.     GPIO_Init(GPIOA, &GPIO_InitStructure);
26.
27.     /* USART1 mode config */
28.     USART_InitStructure.USART_BaudRate = 115200;
29.     USART_InitStructure.USART_WordLength = USART_WordLength_8b;
30.     USART_InitStructure.USART_StopBits = USART_StopBits_1;
31.     USART_InitStructure.USART_Parity = USART_Parity_No ;
32.     USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlow
Control_None;
33.     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
34.     USART_Init(USART1, &USART_InitStructure);
35.     USART_Cmd(USART1, ENABLE);
36. }
```



在代码的第 14 行，调用了库函数 `RCC_APB2PeriphClockCmd()` 初始化了 `USART1` 和 `GPIOA` 的时钟，这是因为使用了 `GPIOA` 的 `PA9` 和 `PA10` 的默认复用 `USART1` 的功能，在使用复用功能的时候，要开启相应的功能时钟 `USART1`。

接下来，这段串口初始化代码分为两个部分，第一部分为 `GPIO` 的初始化，第二部分才是串口的模式、波特率的初始化。

## ■ GPIO 初始化

在前面提到过，`GPIO` 具有默认的复用功能，在使用它的复用功能的时候，我们首先要将相应的 `GPIO` 进行初始化。此时我们使用的 `GPIO` 的复用功能为串口，但为什么是 `PA9` 和 `PA10` 用作串口的 `Tx` 和 `Rx`，而不是其它 `GPIO` 引脚呢？这是从《STM32F103CDE 增强型系列数据手册》的引脚功能定义中查询到的。

|     |     |    |    |    |     |      |     |    |      |                                                       |  |
|-----|-----|----|----|----|-----|------|-----|----|------|-------------------------------------------------------|--|
| D12 | C9  | D2 | 42 | 68 | 101 | PA9  | I/O | FT | PA9  | USART1_TX <sup>(7)</sup><br>TIM1_CH2 <sup>(7)</sup>   |  |
| D11 | D10 | D3 | 43 | 69 | 102 | PA10 | I/O | FT | PA10 | USART1_RX <sup>(7)</sup> /<br>TIM1_CH3 <sup>(7)</sup> |  |

图 1-5 GPIO 引脚功能说明图

选定了这两个引脚，并且 `PA9` 为 `Tx`，`PA10` 为 `Rx`，那么它们的 `GPIO` 模式要如何配置呢？`Tx` 为发送端，输出引脚，而且现在 `GPIO` 是使用复用功能，所以要把它配置为复用推挽输出 (`GPIO_Mode_AF_PP`)；而 `Rx` 引脚为接收端，输入引脚，所以配置为浮空输入模式 `GPIO_Mode_IN_FLOATING`。如果在使用复用功能的时候，对 `GPIO` 的模式不太确定的话，我们可以从《STM32 参考手册》的 `GPIO` 章节中查询得到，见图 1-6。

| USART 引脚   | 配置      | GPIO 配置      |
|------------|---------|--------------|
| USARTx_TX  | 全双工模式   | 推挽复用输出       |
|            | 半双工同步模式 | 推挽复用输出       |
| USARTx_RX  | 全双工模式   | 浮空输入或带上拉输入   |
|            | 半双工同步模式 | 未用，可作为通用 I/O |
| USARTx_CK  | 同步模式    | 推挽复用输出       |
| USARTx_RTS | 硬件流量控制  | 推挽复用输出       |
| USARTx_CTS | 硬件流量控制  | 浮空输入或带上拉输入   |

图 1-6 GPIO 复用功能模式设置

## ■ USART 初始化

从代码的第 28 行开始，进行 USART1 的初始化，也就是填充 USART 的初始化结构体。这部分内容，是根据串口通讯协议来设置的。

### 1. `.USART_BaudRate = 115200;`

波特率设置，利用库函数，我们可以直接这样配置波特率，而不需要自行计算 `USARTDIV` 的分频因子。在这里把串口的波特率设置为 115200，也可以设置为 9600 等常用的波特率，在《STM32 参考手册》中列举了一些常用的波特率设置及其误差，见图 1-7。如果配置成 9600，那么在和 PC 通讯的时候，也应把 PC 的串口传输波特率设置为相同的 9600。通讯协议要求两个通讯器件之间的波特率、字长、停止位奇偶校验位都相同。

| 波特率 |       | f <sub>PCLK</sub> = 36MHz |             |       | f <sub>PCLK</sub> = 72MHz |             |       |
|-----|-------|---------------------------|-------------|-------|---------------------------|-------------|-------|
| 序号  | Kbps  | 实际                        | 置于波特率寄存器中的值 | 误差%   | 实际                        | 置于波特率寄存器中的值 | 误差%   |
| 1   | 2.4   | 2.400                     | 937.5       | 0%    | 2.4                       | 1875        | 0%    |
| 2   | 9.6   | 9.600                     | 234.375     | 0%    | 9.6                       | 468.75      | 0%    |
| 3   | 19.2  | 19.2                      | 117.1875    | 0%    | 19.2                      | 234.375     | 0%    |
| 4   | 57.6  | 57.6                      | 39.0625     | 0%    | 57.6                      | 78.125      | 0%    |
| 5   | 115.2 | 115.384                   | 19.5        | 0.15% | 115.2                     | 39.0625     | 0%    |
| 6   | 230.4 | 230.769                   | 9.75        | 0.16% | 230.769                   | 19.5        | 0.16% |
| 7   | 460.8 | 461.538                   | 4.875       | 0.16% | 461.538                   | 9.75        | 0.16% |
| 8   | 921.6 | 923.076                   | 2.4375      | 0.16% | 923.076                   | 4.875       | 0.16% |
| 9   | 2250  | 2250                      | 1           | 0%    | 2250                      | 2           | 0%    |
| 10  | 4500  | 不可能                       | 不可能         | 不可能   | 4500                      | 1           | 0%    |

图 1-7 STM32 常用波特率及其误差

### 2. `.USART_WordLength = USART_WordLength_8b;`

配置串口传输的字长。本例程把它设置为最常用的 8 位字长，也可以设置为 9 位。

### 3. `.USART_StopBits = USART_StopBits_1;`

配置停止位。把通讯协议中的停止位设置为 1 位。

### 4. `.USART_Parity = USART_Parity_No ;`

配置奇偶校验位。本例程不设置奇偶校验位。

#### 5. `.USART_HardwareFlowControl= USART_HardwareFlowControl_None;`

配置硬件流控制。不采用硬件流。

**硬件流**，在 STM32 的很多外设都具有硬件流的功能，其功能表现为：当外设硬件处于准备好的状态时，硬件启动自动控制，而不需要软件再进行干预。

在串口这个外设的硬件流具体表现为：使用串口的 **RTS (Request to Send)** 和 **CTS (Clear to Send)** 针脚，当串口已经准备好接收新数据时，由硬件流自动把 RTS 针拉低(向外表示可接收数据)；在发送数据前，由硬件流自动检查 CTS 针是否为低(表示是否可以发送数据)，再进行发送。本串口例程没有使用到 CTS 和 RTS，所以不采用硬件流控制。

#### 6. `USART_Mode = USART_Mode_Rx | USART_Mode_Tx;`

配置串口的模式。为了配置双线全双工通讯，需要把 Rx 和 Tx 模式都开启。

7. 填充完结构体，调用库函数 `USART_Init()`向寄存器写入配置参数。

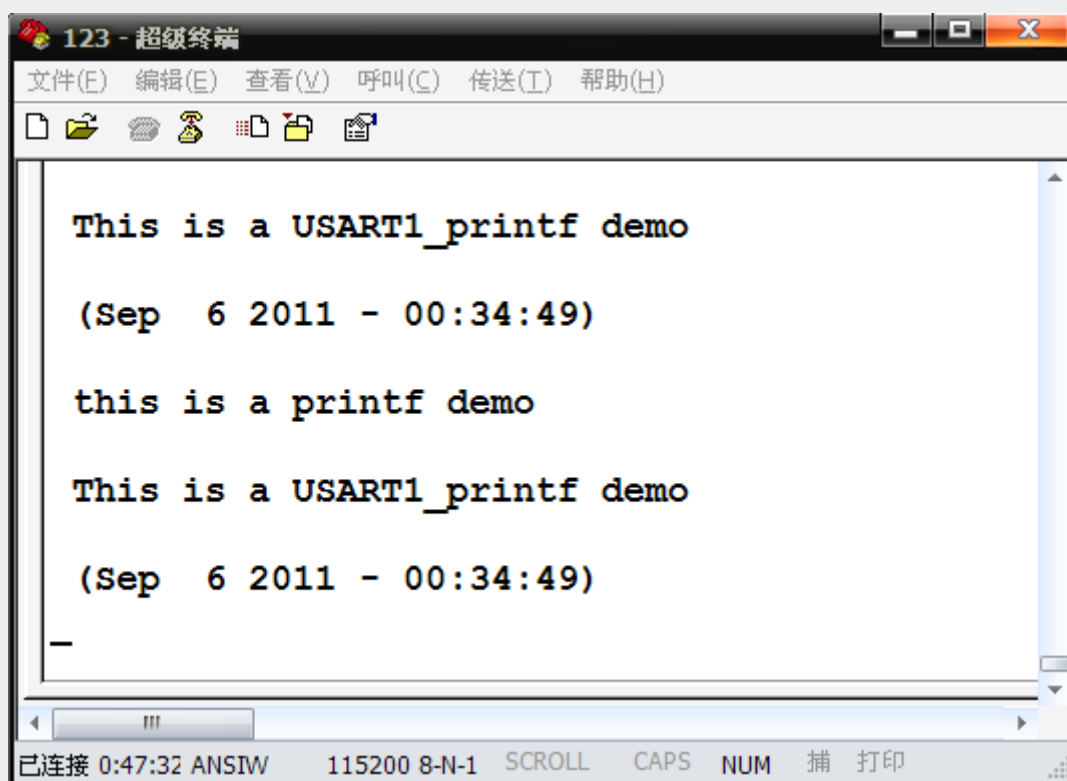
8. 最后，调用 `USART_Cmd()`使能 `USART1` 外设。在使用外设时，不仅要使能其时钟，还要调用此函数使能外设才可以正常使用。

#### 8.4.5. `printf()` 函数重定向

在 main 文件中，配置好串口之后，就通过下面的几行代码由串口往电脑里面的超级终端打印信息，打印的信息为一些字符串和当前的日期。

```
1. printf("\r\n this is a printf demo \r\n");
2.
3. printf("\r\n 欢迎使用野火 M3 实验板:) \r\n");
4.
5. USART1_printf(USART1, "\r\n This is a USART1_printf demo \r\n");
6.
7. USART1_printf(USART1, "\r\n (" __DATE__ " - " __TIME__ ") \r\n");
```

下面是电脑超级终端的截图，从图可以看出程序是运行正确的。



调用这三个函数看似很简单，但在这三个函数的背后还得做些工作，我们先来看 `printf()` 这个函数。要想 `printf()` 函数工作的话，我们需要把 `printf()` 重新定向到串口中。**重定向**，是指用户可以自己重写 c 的库函数，当连接器检查到用户编写了与 C 库函数相同名字的函数时，优先采用用户编写的函数，这样用户就可以实现对库的修改了。

为了实现 **重定向** `printf()` 函数，我们需要重写 `fputc()` 这个 c 标准库函数，因为 `printf()` 在 c 标准库函数中实质是一个宏，最终是调用了 `fputc()` 这个函数的。

重定向的这部分工作，由 `usart.c` 文件中的 `fputc(int ch, FILE *f)` 这个函数来完成，这个函数具体实现如下：

```
1.  /*
2.   * 函数名: fputc
3.   * 描述   : 重定向 c 库函数 printf 到 USART1
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 由 printf 调用
7.   */
8.  int fputc(int ch, FILE *f)
```

```

9. {
10.     /* 将 Printf 内容发往串口 */
11.     USART_SendData(USART1, (unsigned char) ch);
12. // while (!(USART1->SR & USART_FLAG_TXE));
13.     while( USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET);
14.     return (ch);
15. }

```

这个代码中调用了两个 ST 库函数。*USART\_SendData()* 和 *USART\_GetFlagStatus()* 其说明见图 1-8 及图 1-9。

重定向时，我们把 *fputc()* 的形参 *ch*，作为串口将要发送的数据，也就是说，当使用 *printf()*，它调用这个 *fputc()* 函数时，然后使用 ST 库的串口发送函数 *USART\_SendData()*，把数据转移到发送数据寄存器 *TDR*，触发我们的串口向 PC 发送一个相应的数据。调用完 *USART\_SendData()* 后，要使用 *while( USART\_GetFlagStatus(USART1, USART\_FLAG\_TC) != SET)* 语句不停地检查串口发送是否完成的标志位 *TC*，一直检测到标志为完成，才进入下一步的操作，避免出错。在这段 *while* 的循环检测的延时中，串口外设已经由发送控制器根据我们的配置把数据从移位寄存器一位一位地通过串口线 *Tx* 发送出去了。

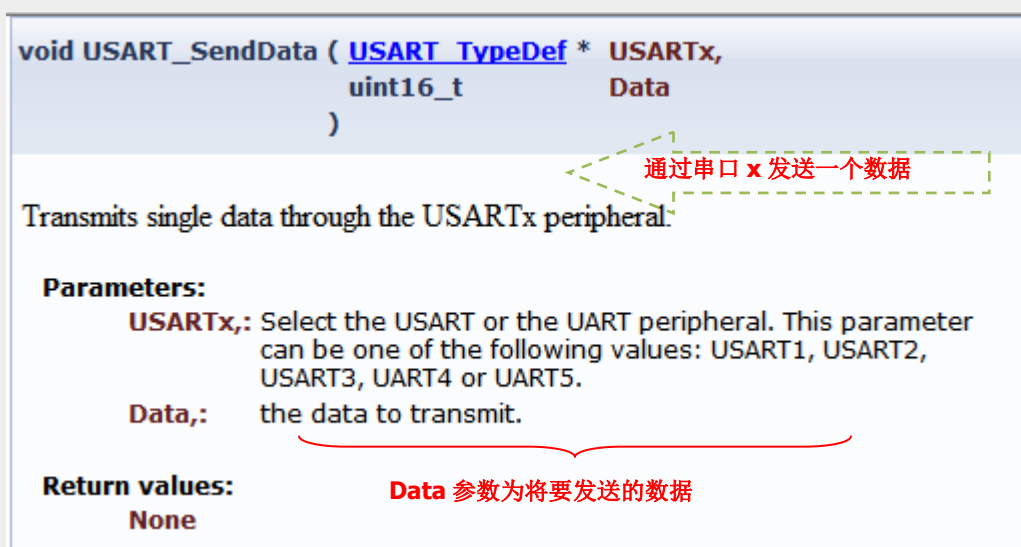


图 1-8 串口发送函数

FlagStatus USART\_GetFlagStatus ( USART\_TypeDef \* USARTx,  
uint16\_t USART\_FLAG  
)

检查 USART 的标志位

Checks whether the specified USART flag is set or not.

Parameters:

USARTx,:

Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, UART4 or UART5.

USART\_FLAG,:

specifies the flag to check. This parameter can be one of the following values:

这些为可输入的标志位参数

如 USART\_FLAG\_TXE 表示发送区是否为空的标志位。

USART\_FLAG\_TC 表示是否发送完成的标志位

- USART\_FLAG\_CTS: CTS Change flag (not available for UART4 and UART5)
- USART\_FLAG\_LBD: LIN Break detection flag
- USART\_FLAG\_TXE: Transmit data register empty flag
- USART\_FLAG\_TC: Transmission Complete flag
- USART\_FLAG\_RXNE: Receive data register not empty flag
- USART\_FLAG\_IDLE: Idle Line detection flag
- USART\_FLAG\_ORE: OverRun Error flag
- USART\_FLAG\_NE: Noise Error flag
- USART\_FLAG\_FE: Framing Error flag
- USART\_FLAG\_PE: Parity Error flag

返回标志位检查结果。

Return values:

The new state of USART\_FLAG (SET or RESET).

图 1-9 串口标志位检查函数

在使用 c 标准输出库函数，要添加什么头文件？搞嵌入式的可不要把这个忘记了哦。在我们的 *main.c* 文件中要把 *stdio.h* 这个头文件包含进来，还要在编译器中设置一个选项 Use MicroLIB (使用微库)，见图 1-10。这个微库是 keil MDK 为嵌入式应用量身定做的 C 库，我们要先具有库，才能重定向吧？勾选使用之后，我们就可以使用 *printf()* 这个函数了。

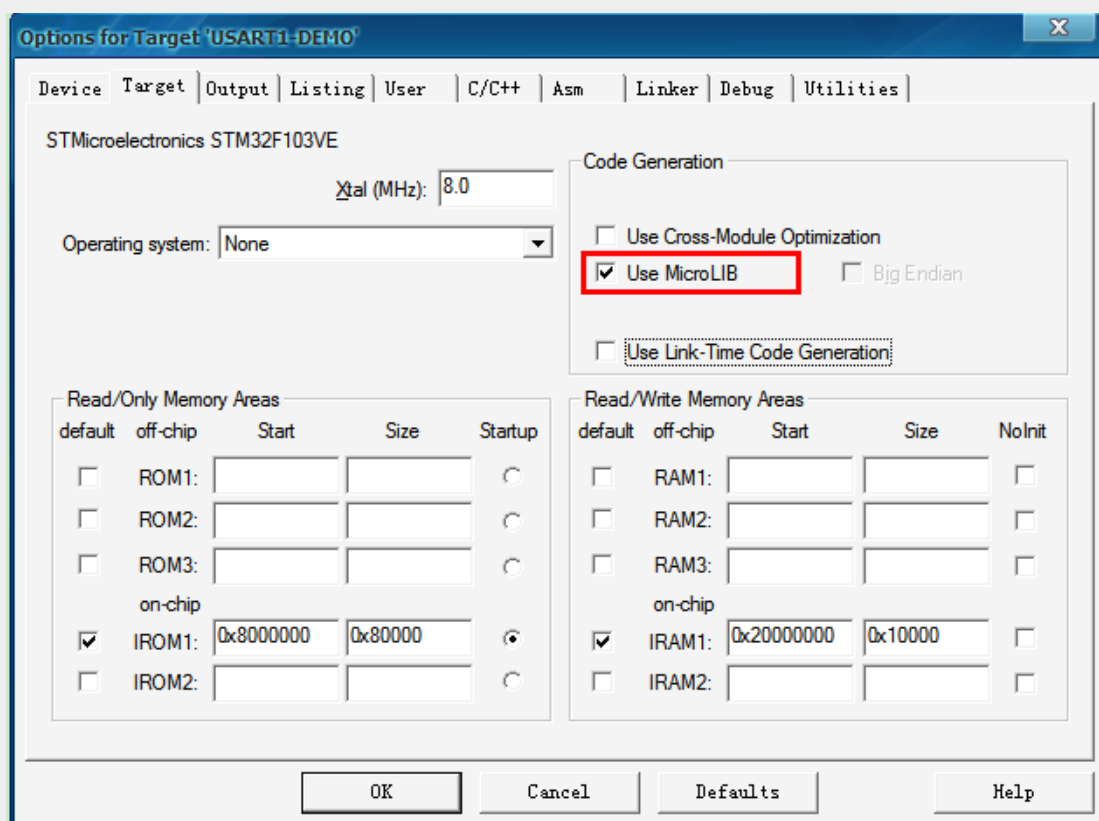


图 1-10 勾选使用微库

#### 8.4.6. USART1\_printf() 函数

除了重定向的方法，我们还可以自己编写格式输入输出函数。

*USART1\_printf()*便是一个完全自定义的格式输出函数，它的功能与重定向之后的 `printf` 类似。

让我们再来看看

*USART1\_printf(USART\_TypeDef\* USARTx, uint8\_t \*Data,...)* 这个函数的实现，它调用了 *itoa(int value, char \*string, int radix)* 函数。关于这两个函数的具体实现请看 *usart.c* 中的源代码。这两个函数中有些变量是定义在 *stdarg.h* 这个头文件中的，所以在 *usart.c* 中我们需要把这个头文件包含进来，这个头文件位于 KDE 的根目录下。我们可以在这个路径下找到它：

*C:\Keil\ARM\RV31\INC。*

```
1. /*
2.  * 函数名: itoa
```

```
3.  * 描述   : 将整形数据转换成字符串
4.  * 输入   : -radix =10 表示10进制, 其他结果为0
5.  *        -value 要转换的整形数
6.  *        -buf 转换后的字符串
7.  *        -radix = 10
8.  * 输出   : 无
9.  * 返回   : 无
10. * 调用   : 被 USART1_printf()调用
11. */
12. static char *itoa(int value, char *string, int radix)
13. {
14.     int     i, d;
15.     int     flag = 0;
16.     char    *ptr = string;
17.
18.     /* This implementation only works for decimal numbers. */
19.     if (radix != 10)
20.     {
21.         *ptr = 0;
22.         return string;
23.     }
24.
25.     if (!value)
26.     {
27.         *ptr++ = 0x30;
28.         *ptr = 0;
29.         return string;
30.     }
31.
32.     /* if this is a negative value insert the minus sign. */
33.     if (value < 0)
34.     {
35.         *ptr++ = '-';
36.         /* Make the value positive. */
37.         value *= -1;
38.     }
39.     for (i = 10000; i > 0; i /= 10)
40.     {
41.         d = value / i;
42.         if (d || flag)
43.         {
44.             *ptr++ = (char)(d + 0x30);
45.             value -= (d * i);
46.             flag = 1;
47.         }
48.     }
49.
50.     /* Null terminate the string. */
51.     *ptr = 0;
52.     return string;
53. } /* NCL_Itoa */
54.
```

```
55. /*
56. * 函数名: USART1_printf
57. * 描述   : 格式化输出, 类似于C库中的printf, 但这里没有用到C库
58. * 输入   : -USARTx 串口通道, 这里只用到了串口1, 即 USART1
59. *        -Data 要发送到串口的内容的指针
60. *        -... 其他参数
```



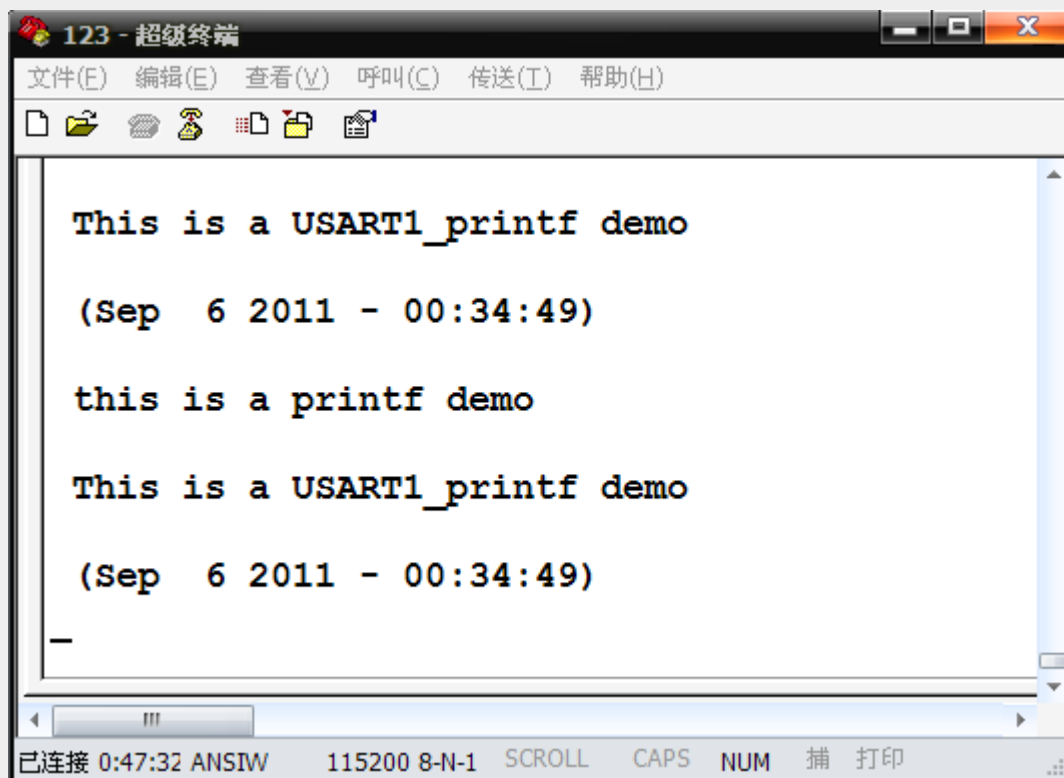
```
61.  * 输出 : 无
62.  * 返回 : 无
63.  * 调用 : 外部调用
64.  *      典型应用 USART1_printf( USART1, "\r\n this is a demo \r\n" );
65.  *      USART1_printf( USART1, "\r\n %d \r\n", i );
66.  *      USART1_printf( USART1, "\r\n %s \r\n", j );
67.  */
68. void USART1_printf(USART_TypeDef* USARTx, uint8_t *Data,...)
69. {
70.     const char *s;
71.     int d;
72.     char buf[16];
73.     va_list ap;
74.     va_start(ap, Data);
75.     while ( *Data != 0 )      // 判断是否到达字符串结束符
76.     {
77.         if ( *Data == 0x5c )  //'\'
78.         {
79.             switch ( *++Data )
80.             {
81.                 case 'r':      //回车符
82.                     USART_SendData(USARTx, 0x0d);
83.                     Data ++;
84.                     break;
85.
86.                 case 'n':      //换行符
87.                     USART_SendData(USARTx, 0x0a);
88.                     Data ++;
89.                     break;
90.
91.                 default:
92.                     Data ++;
93.                     break;
94.             }
95.         }
96.         else if ( *Data == '%' )
97.         {
98.             switch ( *++Data )
99.             {
100.                case 's':      //字符串
101.                    s = va_arg(ap, const char *);
102.                    for ( ; *s; s++)
103.                    {
104.                        USART_SendData(USARTx,*s);
105.                        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
106.                    }
107.                    Data++;
108.                    break;
109.
110.                case 'd':      //十进制
111.                    d = va_arg(ap, int);
112.                    itoa(d, buf, 10);
113.                    for (s = buf; *s; s++)
114.                    {
115.                        USART_SendData(USARTx,*s);
116.                        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
117.                    }
118.                    Data++;
119.                    break;
120.                default:
121.                    Data++;
122.                    break;
123.            }
124.        } /* end of else if */
125.        else USART_SendData(USARTx, *Data++);
126.        while( USART_GetFlagStatus(USARTx, USART_FLAG_TC) == RESET );
127.    }
128. }
```

这部分代码有点多，在格式上编排不是很好，野火推荐大家直接看源码好点。

综上，我们已经可以用 `printf()` 和 `USART1_printf()` 这两个函数来打印信息了，但到底用哪个比较好呢？其实各有千秋，`printf()` 函数会受缓冲区大小的影响，有时候在用它打印的时候程序会发生莫名奇妙的错误，而实际上就是由于使用 `printf()` 这个函数引起的，其优点就是这种情况很少见且支持的格式较多。而 `USART1_printf()` 则不会受缓冲区的影响产生莫名的错误，但其支持的格式较少。不过，相比之下，野火还是比较喜欢用 `USART1_printf()`。

#### 8.4.7. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线，没有的话就 DIY 一个吧 ^\_^)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：



```
123 - 超级终端
文件(E) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

this is a printf demo

This is a USART1_printf demo

(Sep 6 2011 - 00:34:49)

已连接 0:47:32 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

## 9. DMA—为 CPU 减负

### 9.1. DMA 功能简介

DMA (Direct Memory Access)，直接存储器存取，是一种可以大大减轻 CPU 工作量的数据存取方式，因而被广泛地使用。早在 8086 的应用中，就已经有 Intel 的 8237 这种典型的 DMA 控制器。而 STM32 的 DMA 则是以类似外设的形式，添加到 Cortex 内核之外的。

在硬件系统中，主要由 CPU(内核)、外设、内存(SRAM)、总线等结构组成，数据经常要在内存与外设之间转移，或从外设 A 转移到外设 B。

例如：当 CPU 需要处理由 ADC 外设采集回来的数据时，CPU 首先要把数据从 ADC 外设的寄存器读取到内存中(变量)，然后进行运算处理，这是一般的处理方法。

在转移数据的过程中会占用 CPU 十分宝贵的资源，所以我们希望 CPU 更多地被用在数据运算或响应中断之中，而数据转移的工作交由其它部件完成。DMA 正为 CPU 分担了数据转移的工作。因为 DMA 的存在，CPU 被解放出来，它可以在 DMA 转移数据的过程中同时 进行数据运算，响应中断，大大提高效率。

### 9.2. DMA 工作分析

见图 9-1 STM32 的系统结构，在这个图中，我们可以清晰地看到 STM32 内核、存储器、外设及 DMA 的连接。

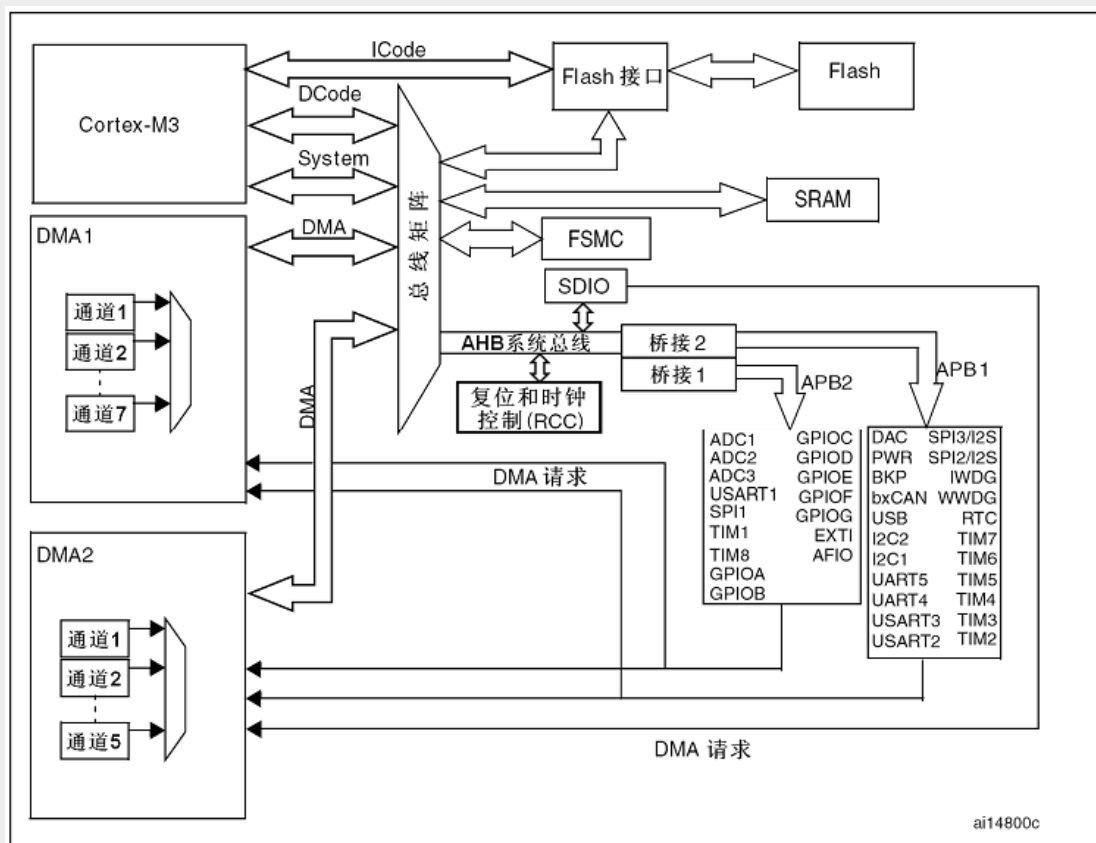


图 9-1 STM32 的系统结构

所有这些硬件结构最终都通过各种各样的线连接到总线矩阵之中，硬件结构之间的数据转移都经过总线矩阵的协调，使各个外设都能够和谐地使用总线来传输数据。

例如：在不使用 DMA 的情况下，内核通过 DCode 经过总线矩阵协调，使用 AHB 把外设 ADC 采集的数据读取到内核，然后内核 DCode 再通过总线矩阵协调，把数据存放到内存 SRAM 中。

DMA 正好可以取代这样的工作。由 DMA 控制器的 DMA 总线与总线矩阵协调，使用 AHB 把外设 ADC 的数据经由 DMA 通道存放到内存 SRAM。在这个数据传输的过程中，不需要内核的全程参与，所以内核可以同时进行数据运算，而且，DMA 的方式是点到点的数据转移，而不使用 DMA 的方式还要以内核来作为中转站，显然 DMA 的传输方式的效率更高，直接存储器存取中的“直接”不是徒有虚名的。

要使用 DMA，需要确定一系列的控制参数，如外设数据的地址，内存地址，传输方向等，在开启 DMA 传输前还要先发出 DMA 请求。

### 9.3. DMA 实例之串口通信

本小节以实例对 STM32 的 DMA 进行讲解，以 DMA 的方式使用串口发送数据。实际上是利用 DMA 把数据(数组)从内存转移到外设(串口)。我们知道外设工作的时候，除了转移数据，实质是不需要内核干预的，而数据转移的工作现在交给了 DMA，所以在串口发送数据的时候，内核同时还可以进行其它操作，例如点亮 LED 灯。

#### 9.3.1. 实验描述及工程文件清单

##### 实验描述：

利用 DMA 的方式，使用串口发送数据。并利用 LED 进行检验在使用 DMA 传输的过程中，内核是否可以同时进行其它的操作(点亮 LED)。

##### 硬件连接：

PA9 - USART1(Tx)

PA10 - USART1(Rx)

##### 库文件：

3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_dma.c

FWlib/misc.c

##### 用户文件：

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

### 9.3.2. 配置工程环境

本串口 DMA 实验中我们用到了 GPIO、RCC、USART、DMA 外设及中断，所以我们要把以下库文件添加到工程 stm32f10x\_gpio.c、stm32f10x\_rcc.c、stm32f10x\_usart.c、stm32f10x\_dma.c、misc.c，本实验主要在 usart1.c 文件中添加 DMA 配置的代码，添加旧工程中的外设用户文件 led.c、usart1.c，并在 stm32f10x\_conf.h 中把相应的头文件的注释去掉。见代码清单 9-1。

代码清单 9-1：DMA 例程的 stm32f10x\_conf.h 文件配置

```

1.  /*****
   ***
2.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
3.  * @author    MCD Application Team
4.  * @version    V3.5.0
5.  * @date      08-April-2011
6.  * @brief     Library configuration file.
7.  *****/
8.  #include "stm32f10x_dma.h"
9.  #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_usart.h"
12. #include "misc.h"

```

### 9.3.3. main 文件

依然从 main 函数，按照代码的执行流程进行分析见代码清单 9-2。

代码清单 9-2：DMA 例程的 main 函数

```

1.  #include "stm32f10x.h"
2.  #include "usart1.h"
3.  #include "led.h"
4.
5.  extern uint8_t SendBuff[SENDBUFF_SIZE];
6.  uint16_t i;
7.
8.
9.  /*

```

```

10. * 函数名: main
11. * 描述 : 主函数
12. * 输入 : 无
13. * 输出 : 无
14. */
15. int main(void)
16. {
17.     /* USART1 config 115200 8-N-1 */
18.     USART1_Config();
19.     DMA_Config();
20.     LED_GPIO_Config();
21.
22.
23.     /*填充将要发送的数据*/
24.     for(i=0;i<SENDER_BUFF_SIZE;i++)
25.     {
26.         SendBuff[i] = 0xff;
27.     }
28.
29.     /*串口向 DMA 发出请求 */
30.     USART_DMACmd(USART1, USART_DMAREQ_Tx, ENABLE);
31.
32.     /*在 DMA 尚未传送完成时, cpu 继续执行 main 函数中的代码*/
33.     /*点亮了 LED 灯*/
34.     /*而同时 DMA 在向串口运送数据, 当 DMA 发送完成时, 在中断函数关闭 LED
灯 */
35.
36.
37.     LED1(ON);
38.
39.
40.     while(1);
41. }

```

从 main 函数来看, 这个工程十分简单, 首先调用了用户函数 USART1\_Config()、DMA\_Config()及 LED\_GPIO\_Config()分别配置好了串口、DMA、及 LED 外设。其中 USART1\_Config()和 LED\_GPIO\_Config()函数跟前面的工程是完全相同的, 没有丝毫的改动。

#### 9.3.4. DMA 初始化

接下来我们开始分析本章的重点, DMA 的初始化配置。DMA 的初始化就是在 DMA\_Config() 函数中实现的。其代码位于 usart1.c 文件, 见代码清单 9-3。

代码清单 9-3: DMA\_Config()函数

```

1. /*
2. * 函数名: DMA_Config
3. * 描述 : DMA 串口的初始化配置
4. * 输入 : 无
5. * 输出 : 无
6. * 调用 : 外部调用

```

```

7.  */
8. void DMA_Config(void)
9. {
10.     DMA_InitTypeDef DMA_InitStructure;
11.
12.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
13.     //开启 DMA 时钟
14.
15.     NVIC_Config();           //配置 DMA 中断
16.
17.     /*设置 DMA 源：内存地址&串口数据寄存器地址*/
18.     DMA_InitStructure.DMA_PeripheralBaseAddr = USART1_DR_Base;
19.
20.     /*内存地址(要传输的变量的指针)*/
21.     DMA_InitStructure.DMA_MemoryBaseAddr = (u32)SendBuff;
22.
23.     /*方向：从内存到外设*/
24.     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
25.
26.     /*传输大小 DMA_BufferSize=SENDERBUFF_SIZE*/
27.     DMA_InitStructure.DMA_BufferSize = SENDERBUFF_SIZE;
28.
29.     /*外设地址不增*/
30.     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
31.
32.     /*内存地址自增*/
33.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
34.
35.     /*外设数据单位*/
36.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
37.
38.     /*内存数据单位 8bit*/
39.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
40.
41.     /*DMA 模式：一次传输，循环*/
42.     DMA_InitStructure.DMA_Mode = DMA_Mode_Normal ;
43.
44.     /*优先级：中*/
45.     DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;
46.
47.     /*禁止内存到内存的传输 */
48.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
49.
50.     /*配置 DMA1 的 4 通道*/
51.     DMA_Init(DMA1_Channel4, &DMA_InitStructure);
52.
53.     DMA_Cmd (DMA1_Channel4,ENABLE);           //使能 DMA
54.     DMA_ITConfig(DMA1_Channel4,DMA_IT_TC,ENABLE); //配置 DMA 发送完成后产生中断
55. }

```

初始化外设最主要的任务是什么？开启外设时钟、填充初始化结构体、使能外设。这个函数正是完成了这三个任务。



下面重点分析如何填充 DMA 的初始化结构体，先跳过 NVIC\_Config()函数，直接看给结构体成员赋值部分。

### (1) .DMA\_PeripheralBaseAddr

这个成员保存的是 外设数据寄存器的基地址，这个地址作为传输的源或目标。为什么说是“基”地址？因为 DMA 具有地址自增的功能，地址自增功使得可以方便地读取连续的数据单元。

现在给这个成员的赋值为 USART1\_DR\_Base，是一个自定义的宏，宏展开后它是 USART1 的数据寄存器地址：

```
1. #define USART1_DR_Base 0x40013804
```

涉及到的 USART 的数据寄存器及其地址，见表 9-1。

表 9-1 UART 数据寄存器(USART\_DR)

数据寄存器(USART\_DR)

地址偏移：0x04

1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6

|    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 保留 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|----|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

5 4 3 2 1 0

|    |         |
|----|---------|
| 保留 | DR[8:0] |
|----|---------|

| 位<br>数    | 描述                                                                                                                                        |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------|
| 位<br>31:9 | 保留位，硬件强制为 0                                                                                                                               |
| 位<br>8:0  | DR[8:0]：数据值 (Data value)<br>包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用 (TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄 |

|  |                                                                                                                                                   |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <p>寄存器和内部总线之间的并行接口。当使能校验位(USART_CR1 中 PCE 位被置位)进行发送时，写到 MSB 的值(根据数据的长度不同，MSB 是第 7 位或者第 8 位)会被后来的校验位该取代。</p> <p>当使能校验位进行接收时，读到的 MSB 位是接收到的校验位。</p> |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------|

从《STM32 参考手册》可知，串口外设会自动地把数据寄存器中的数据，送入它的移位寄存器，然后由硬件按照串口协议把该数据发送出去。

在本代码我们把这个数据寄存器的地址作为外设地址，那么由 DMA 通道转移过来的内存数据就会被保存到这个寄存器中，然后串口就会自动进行发送了。

那么这个寄存器的地址又是怎么算出来的呢？请注意上表中的地址偏移为 0x04，前面的章节已经介绍过，这些寄存器的地址偏移是相对于它所在外设的基地址进行偏移的。本代码中使用的为外设串口 1，所以我们在《STM32 参考手册》的存储器映像中找到 USART1(串口 1)的外设基地址为 0x4001 3800，见表 9-2，串口 1 的外设基地址加上数据寄存器的地址偏移就是我们在 DMA 传输中需要的目标地址了：

$$0x4001\ 3804 = 0x4001\ 3800 + 0x04。$$

表 9-2 存储器映射(部分)

| 起始地址                      | 外设           | 总线  |
|---------------------------|--------------|-----|
| 0x5000 0000 – 0x5003 FFFF | USB OTG 全速   | AHB |
| 0x4003 0000 – 0x4FFF FFFF | 保留           |     |
| 0x4002 8000 – 0x4002 9FFF | 以太网          |     |
| 0x4002 3400 - 0x4002 3FFF | 保留           | AHB |
| 0x4002 3000 - 0x4002 33FF | CRC          |     |
| 0x4002 2000 - 0x4002 23FF | 闪存存储器接口      |     |
| 0x4002 1400 - 0x4002 1FFF | 保留           |     |
| 0x4002 1000 - 0x4002 13FF | 复位和时钟控制(RCC) |     |
| 0x4002 0800 - 0x4002 0FFF | 保留           |     |

|                           |           |      |
|---------------------------|-----------|------|
| 0x4002 0400 - 0x4002 07FF | DMA2      |      |
| 0x4002 0000 - 0x4002 03FF | DMA1      |      |
| 0x4001 8400 - 0x4001 7FFF | 保留        |      |
| 0x4001 8000 - 0x4001 83FF | SDIO      |      |
| 0x4001 4000 - 0x4001 7FFF | 保留        | APB2 |
| 0x4001 3C00 - 0x4001 3FFF | ADC3      |      |
| 0x4001 3800 - 0x4001 3BFF | USART1    |      |
| 0x4001 3400 - 0x4001 37FF | TIM8 定时器  |      |
| 0x4001 3000 - 0x4001 33FF | SPI1      |      |
| 0x4001 2C00 - 0x4001 2FFF | TIM1 定时器  |      |
| 0x4001 2800 - 0x4001 2BFF | ADC2      |      |
| 0x4001 2400 - 0x4001 27FF | ADC1      |      |
| 0x4001 2000 - 0x4001 23FF | GPIO 端口 G |      |
| 0x4001 2000 - 0x4001 23FF | GPIO 端口 F |      |
| 0x4001 1800 - 0x4001 1BFF | GPIO 端口 E |      |
| 0x4001 1400 - 0x4001 17FF | GPIO 端口 D |      |
| 0x4001 1000 - 0x4001 13FF | GPIO 端口 C |      |
| 0x4001 0C00 - 0x4001 0FFF | GPIO 端口 B |      |
| 0x4001 0800 - 0x4001 0BFF | GPIO 端口 A |      |
| 0x4001 0400 - 0x4001 07FF | EXTI      |      |
| 0x4001 0000 - 0x4001 03FF | AFIO      |      |

## (2) .DMA\_MemoryBaseAddr

保存了内存的基地址，同样，这个地址可作为传输的源或目标。在使用时通常会给这个成员赋值为某个数组的基地址，然后利用 DMA 的地址自增功能把数组一个个地填满。

在本代码中向这个成员赋值为 **SendBuff**，这是一个自定义的数组变量名，它具有 5000 个数组元素：

```
1. #define SENDBUFF_SIZE 5000
```

```
2. uint8_t SendBuff[SENDBUFF_SIZE];
```

我们知道在 c 语言中数组名就是该数组的基地址，而数组(变量)是被保存到内存(SRAM)上的，所以我们实质上给.DMA\_MemoryBaseAddr 这个结构体成员赋予了一个内存地址。在本代码中，这个内存地址作为 DMA 传输的源地址。

### (3) .DMA\_DIR

保存了 DMA 数据传输方向，可以选择是外设到内存还是内存到外设。

本代码中该成员值为 DMA\_DIR\_PeripheralDST（外设作为目标地址），也就是说，这个成员的值决定了本次数据传输是从内存转移到外设的。

### (4) .DMA\_BufferSize

保存了 DMA 要传输的数据总大小，其单位为后面结构体成员.DMA\_PeripheralDataSize 中的参数。

本代码对它赋值为自定义的宏 SENDBUFF\_SIZE，宏展开为 5000。即本次 DMA 要传输 5000 个数据。

### (5) .DMA\_PeripheralInc 和.DMA\_MemoryInc

分别为外设和内存的地址是否开启自增功能。

本代码向这两个成员赋值为外设地址固定，而内存地址自增，所以 DMA 在传输过程中，数组的元素 0~4999 被一个一个地转移到同样的串口数据寄存器地址。

### (6) .DMA\_PeripheralDataSize 和.DMA\_MemoryDataSize

分别为外设和内存的数据单元大小。可以为字节，半字和字。

本代码设置为字节大小，即每个传输的数据大小为 1 字节，加上前面对.DMA\_BufferSize 成员的配置，本次 DMA 传输的数据总大小即为 5000 字节。

### (7) .DMA\_Mode

保存了 DMA 的模式，可以为循环模式或正常模式，在循环模式下在传输完一轮数据之后再重新传输，这种方式很适合在 ADC 不断采集数据的场合。

本代码只作一次 DMA 传输，采用 DMA\_Mode\_Normal（正常模式）。

### (8) .DMA\_Priority

DMA 通道的优先级，总线矩阵根据其 DMA 通道的优先级进行总线协调分配。

本代码配置为 DMA\_Priority\_Medium（中等优先级），其实在只使用一个 DMA 通道时，配置为任何优先级都没有区别。

### (9) .DMA\_M2M

保存了是否内存到内存的 DMA 传输。DMA 传输可以在外设与内存，外设与外设，还可以在内存与内存之间进行传输。

本代码是内存到外设的传输，所以要把本成员赋值为 DMA\_M2M\_Disable(禁止内存到内存的传输)。

填充完结构体后，就要调用其初始化函数及使能 DMA 外设，见代码清单 9-4。

代码清单 9-4：调用初始化函数和使能 DMA

```
1. DMA_Init(DMA1_Channel4, &DMA_InitStructure);  
2.  
3. DMA_Cmd (DMA1_Channel4,ENABLE);
```

在初始化的时候要选择恰当的 DMA 通道，本次函数使用的是 DMA1\_Channel4 即 DMA1 的通道 4，这个通道并不是随便选择的，而是要根据 DMA 的请求映像来设置。见图 9-2。DMA 请求是指外设在使用 DMA 前需要向 DMA 控制器发送请求信息，DMA 在接收到请求后才会根据 DMA 配置进行数据转移。

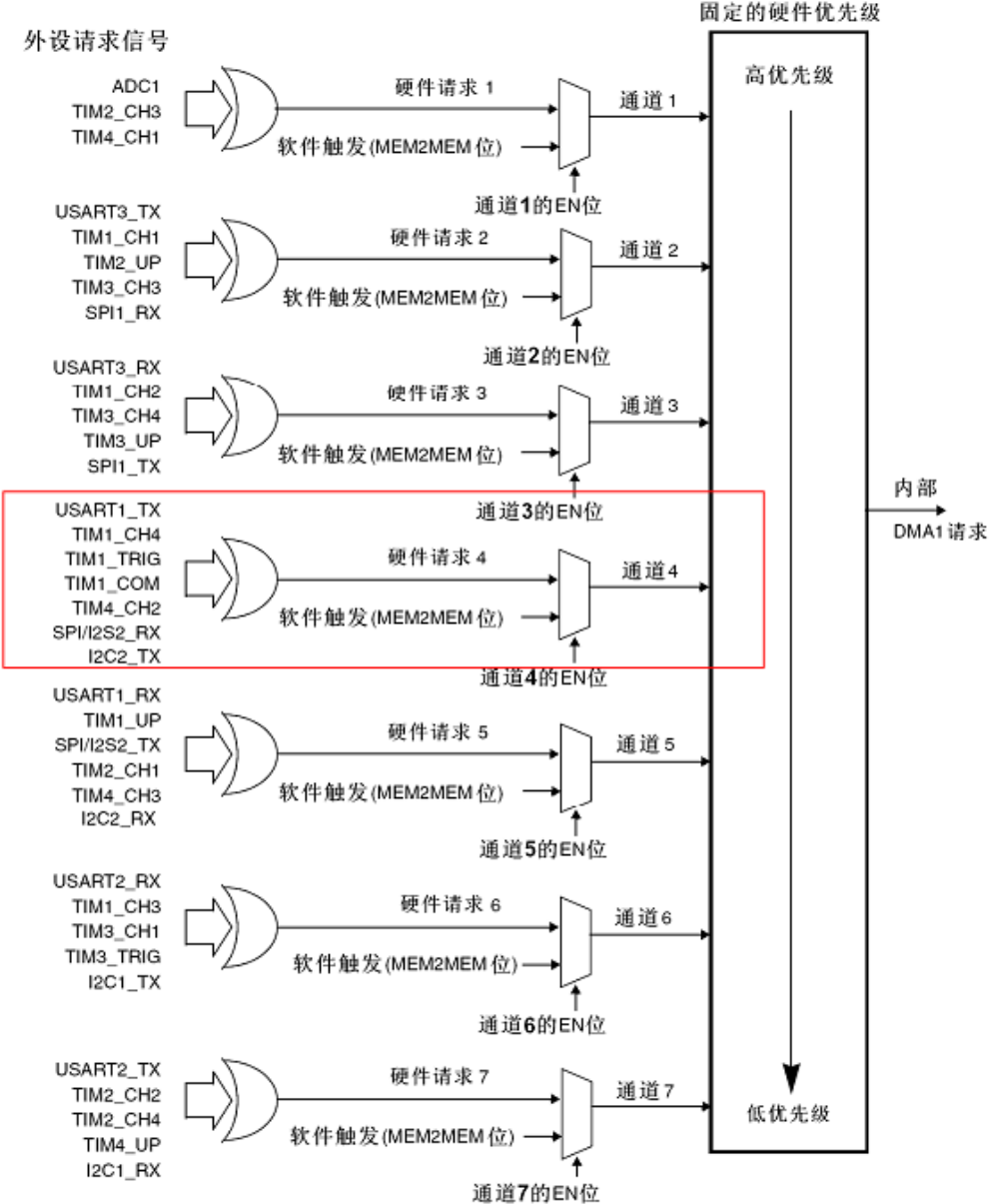


图 9-2 DMA1 请求映像

从图中看到即使是外设串口 1，串口 1 的发送数据 DMA 请求和串口 1 的接收数据 DMA 请求通道都是不一样的，分别为 DMA1 通道 4 和 DMA1 的通道 5。

从整体来说，DMA 被配置为：数据传输方向从内存(数组 SendBuff)到 USART1 外设的数据寄存器(USART1\_DR\_Base)，要传输的数据总量为 SENDBUFF\_SIZE(5000)个字节，并且传输时内存地址自增，外设地址固定，DMA 模式为非循环模式，DMA 通道为 DMA1 的 CHANNEL4。

### 9.3.5. 使用 DMA 中断

为了更好地看到实验现象，DMA\_Config() 还配置了 DMA 发送完成中断，在 stm32f10x\_it.c 文件里的中断服务函数对 LED 进行操作(实际上不使用 DMA 中断就已经可以完成 DMA 传输了)。

在 DMA\_Config()代码中的第 13 行调用了用户函数 NVIC\_Config() 配置了 DMA 的中断优先级，而用户函数 NVIC\_Config() 也跟 EXTI 配置外部中断时用到的很类似，只是其中断通道换成了 DMA1\_Channel4\_IRQn (DMA1 的 4 通道中断)。NVIC\_Config()代码见代码清单 9-5。

代码清单 9-5: NVIC\_Config()函数

```
1.  /*
2.  * 函数名: NVIC_Config
3.  * 描述   : DMA 中断配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. static void NVIC_Config(void)
9. {
10.     NVIC_InitTypeDef NVIC_InitStructure;
11.
12.     /* Configure one bit for preemption priority */
13.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
14.
15.     /* 配置 P[A|B|C|D|E]0 为中断源 */
16.     NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel4_IRQn;
17.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
18.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
19.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
20.     NVIC_Init(&NVIC_InitStructure);
21. }
```

在 DMA\_Config()函数的第 50 行调用了 ST 库函数 DMA\_ITConfig() 把 DMA 配置成 DMA\_IT\_TC (DMA 发送完成标志) 中断。DMA\_ITConfig()是用于配置 DMA 外设中断的函数，在**错误!未找到引用源**。已经介绍过其类似函数了。

接下来还要在 stm32f10x\_it.c 文件中编写中断服务函数，见代码清单 9-6。

代码清单 9-6: DMA 中断服务函数

```
1. void DMA1_Channel4_IRQHandler(void)
2. {
3.     //判断是否为 DMA 发送完成中断
4.     if(DMA_GetFlagStatus(DMA1_FLAG_TC4)==SET)
5.     {
6.         //LED 关闭
7.
8.         LED1(OFF);
9.
10.        DMA_ClearFlag(DMA1_FLAG_TC4);
11.    }
12. }
```

同样，这个中断服务函数的名字是从启动文件 `startup_stm32f10x_hd.s` 中查找到的。中断服务函数里，除了对标志位的检查、清除，主要就是把在 `main` 函数中点亮的 `LED1` 关闭了。

我们回到 `main` 函数进行流程分析。在初始化完成串口、DMA、LED 后，使用 `for` 循环先将要传送的数组全部填充为 `0xff`，然后调用库函数：

```
1. USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);
```

其说明见图 9-3。



```
void USART_DMACmd ( USART_TypeDef * USARTx,
                    uint16_t      USART_DMAReq,
                    FunctionalState NewState
                    )
```

Enables or disables the USART's DMA interface. 使能或关闭串口的 DMA 接口

**Parameters:**

**USARTx,:** Select the USART or the UART peripheral. This parameter can be one of the following values: USART1, USART2, USART3, UART4 or UART5.

**USART\_DMAReq,:** specifies the DMA request. This parameter can be any combination of the following values:

- USART\_DMAReq\_Tx: USART DMA transmit request
- USART\_DMAReq\_Rx: USART DMA receive request

此参数可以为串口 DMA 发送请求，可以为串口 DMA 接收

**NewState,:** new state of the DMA Request sources. This parameter can be: ENABLE or DISABLE.

**Note:** The DMA mode is not available for UART5 except in the STM32 High density value line devices(STM32F10X\_HD\_VL).

**Return values:** None

图 9-3 串口 DMA 使能函数

调用这个库函数允许串口外设向 DMA 发出请求，请求 DMA 传输数据。调用了这个函数之后，DMA 开始响应串口的请求，根据我们的 DMA 配置，把数组中的数据一个个地转移到串口数据寄存器，并由串口向外发送这些数据。

在调用了 USART\_DMACmd() 函数之后，接下来在 main 函数就把 LED 灯点亮了，实际上，在 DMA 还没传输完成数据的时候，因为内核并不参与 DMA 数据传输的全过程，所以内核在这个时候执行了点亮 LED1 的代码。而当 DMA 传输完成时，进入了中断，再把 LED1 关闭。

## 9.3.6. 实验现象

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上串口线, 打开串口调试助手, 配置调试助手为 115200 8-N-1, 十六进制显示。将编译好的程序下载到开发板, 即可看到超级终端打印出下图信息, 并且在 PC 端数据还没接收完的时候, 板子上的 LED1 被点亮, 在数据接收完成时, LED1 被关闭了, 说明内核的确在 DMA 传输的过程中同时进行了其它工作。



## 10. ADC 实验 (DMA 方式)

### 10.1. ADC 简介

ADC (Analog to Digital Converter), 模/数转换器。在模拟信号需要以数字形式处理、存储或传输时, 模/数转换器几乎必不可少。

STM32 在片上集成的 ADC 外设非常强大。在 STM32F103xC、STM32F103xD 和 STM32F103xE 增强型产品, 内嵌 3 个 12 位的 ADC, 每个 ADC 共用多达 21 个外部通道, 可以实现单次或多次扫描转换。如野火 STM32 开发板用的是 STM32F103VET6, 属于增强型的 CPU, 它有 18 个通道, 可测量 16 个外部和 2 个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阈值。

### 10.2. STM32 的 ADC 主要技术指标

对于 ADC 来说, 我们最关注的就是它的分辨率、转换速度、ADC 类型、参考电压范围。

#### □ 分辨率

12 位分辨率。不能直接测量负电压, 所以没有符号位, 即其最小量化单位

$$\text{LSB} = V_{\text{ref+}} / 212。$$

#### □ 转换时间

转换时间是可编程的。采样一次至少要用 14 个 ADC 时钟周期, 而 ADC 的时钟频率最高为 14MHz, 也就是说, 它的采样时间最短为 1us。足以胜任中、低频数字示波器的采样工作。

#### □ ADC 类型

ADC 的类型决定了它性能的极限, STM32 的是逐次比较型 ADC。

#### □ 参考电压范围

STM32 的 ADC 参考电压输入见表 10-1。

表 10-1 ADC 参考电压

| 引脚名称          | 信号类型      | 注解                                                    |
|---------------|-----------|-------------------------------------------------------|
| VREF+         | 输入，模拟参考正极 | ADC 使用的高端/正极参考电压， $2.4V \leq V_{REF+} \leq V_{DDA}$   |
| VDDA(1)       | 输入，模拟电源   | 等效于 VDD 的模拟电源且： $2.4V \leq V_{DDA} \leq V_{DD}(3.6V)$ |
| VREF-         | 输入，模拟参考负极 | ADC 使用的低端/负极参考电压， $V_{REF-} = V_{SSA}$                |
| VSSA(1)       | 输入，模拟电源地  | 等效于 VSS 的模拟电源地                                        |
| ADCx_IN[15:0] | 模拟输入信号    | 16 个模拟输入通道                                            |

1. VDDA 和 VSSA 应该分别连接到 VDD 和 VSS。

从图中可知，它的参考电压负极是要接地的，即  $V_{REF-} = 0V$ 。而参考电压正极的范围为  $2.4V \leq V_{REF+} \leq 3.6V$ ，所以 STM32 的 ADC 是不能直接测量负电压的，而且其输入的电压信号的范围为： $V_{REF-} \leq V_{IN} \leq V_{REF+}$ 。当需要测量负电压或测量的电压信号超出范围时，要先经过运算电路进行平移或利用电阻分压。

### 10.3. ADC 工作过程分析

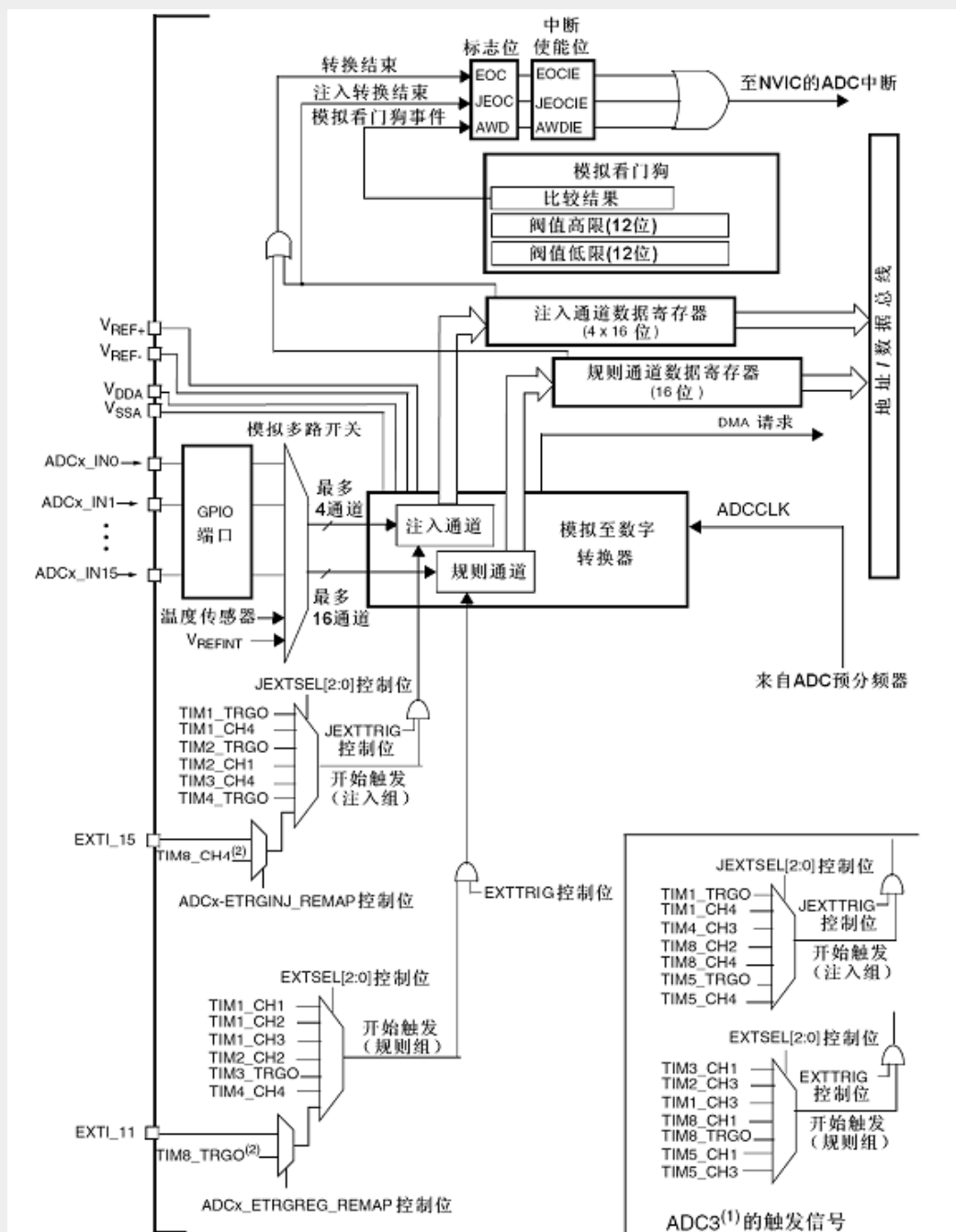


图 10-1 ADC 架构图

我们以 ADC 的规则通道转换来进行过程分析。所有的器件都是围绕中间的模拟至数字转换器部分（下面简称 ADC 部件）展开的。它的左端为  $V_{REF+}$ 、 $V_{REF-}$  等 ADC 参考电压，ADCx\_IN0~ADCx\_IN15 为 ADC 的输入信号通道，即某些 GPIO 引脚。输入信号经过这些通道被送到 ADC 部件，ADC 部件需要受到触

发信号才开始进行转换，如 EXTI 外部触发、定时器触发，也可以使用软件触发。ADC 部件接收到触发信号之后，在 ADCCLK 时钟的驱动下对输入通道的信号进行采样，并进行模数转换，其中 ADCCLK 是来自 ADC 预分频器的。

ADC 部件转换后的数值被保存到一个 16 位的规则通道数据寄存器(或注入通道数据寄存器)之中，我们可以通过 CPU 指令或 DMA 把它读取到内存(变量)。模数转换之后，可以触发 DMA 请求，或者触发 ADC 的转换结束事件。如果配置了模拟看门狗，并且采集得的电压大于阈值，会触发看门狗中断。

#### 10.4. ADC 采集数据实例(采用 DMA 模式)

使用 ADC 时常常需要不间断采集大量的数据，在一般的器件中会使用中断进行处理，但使用中断的效率还是不够高。在 STM32 中，使用 ADC 时往往采用 DMA 传输的方式，由 DMA 把 ADC 外设转换得的数据传输到 SRAM，再进行处理，甚至直接把 ADC 的数据转移到串口发送给上位机。本小节对 ADC 的 DMA 方式采集数据实例进行讲解，在讲解 ADC 的同时让读者进一步熟悉 DMA 的使用。

##### 10.4.1. 实验描述及工程文件清单

###### 实验描述：

串口 1(USART1)向电脑的超级终端以一定的时间间隔打印当前 ADC1 的转换电压值。

###### 硬件连接：

PC1 - ADC1 连接外部电压(通过一个滑动变阻器分压而来)。

###### 库文件：

3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_adc.c

FWlib/stm32f10x\_dma.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

USER/adc.c

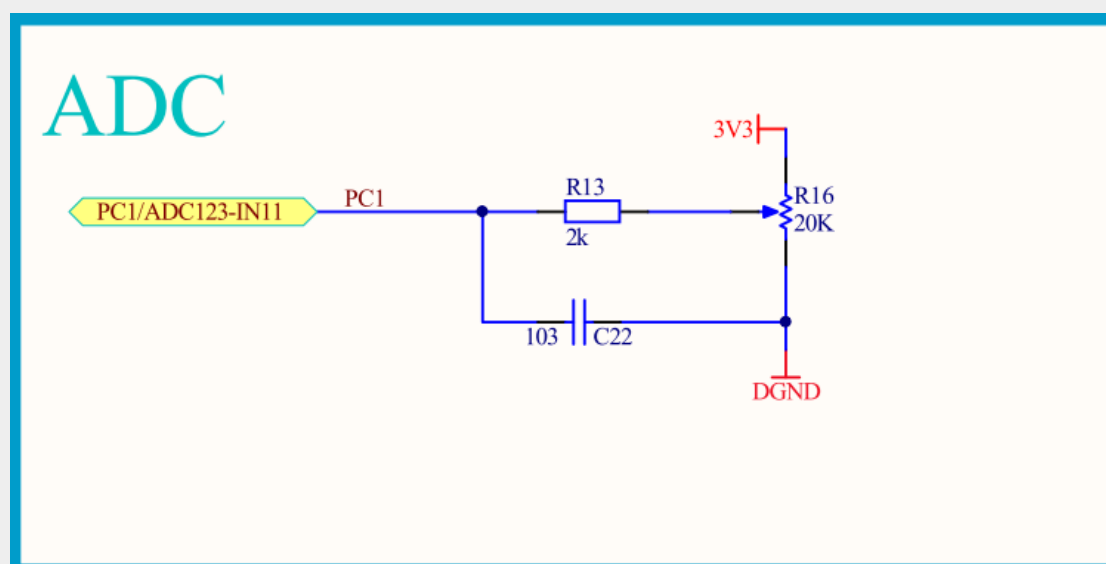


图 10-2 野火 STM32 开发板 ADC 硬件原理图

#### 10.4.2. 配置工程环境

本 ADC (DMA 方式) 实验中我们用到了 GPIO、RCC、USART、DMA 及 ADC 外设，所以我们要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_usart.c`、`stm32f10x_dma.c`、`stm32f10x_adc.c`，添加旧工程中的外设用户文件 `usart1.c`，新建 `adc.c` 及 `adc.h` 文件，并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉，见代码清单 10-1。

代码清单 10-1: ADC 例程的 stm32f10x\_conf.h 文件配置

```

1.  /**
2.  ****
3.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf
f.h
4.  * @author    MCD Application Team
5.  * @version    V3.5.0
6.  * @date      08-April-2011
7.  * @brief     Library configuration file.
8.  ****
/
9.  #include "stm32f10x_adc.h"
10. #include "stm32f10x_dma.h"
11. #include "stm32f10x_gpio.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_usart.h"

```

### 10.4.3. main 文件

配置好工程环境之后，我们就从 main 文件开始分析，见代码清单 10-2。

代码清单 10-2: ADC 例程的 main 函数

```

1.  #include "stm32f10x.h"
2.  #include "usart1.h"
3.  #include "adc.h"
4.
5.  // ADC1 转换的电压值通过 MDA 方式传到 SRAM
6.  extern __IO uint16_t ADC_ConvertedValue;
7.
8.  // 局部变量，用于保存转换计算后的电压值
9.
10. float ADC_ConvertedValueLocal;
11.
12. // 软件延时
13. void Delay(__IO uint32_t nCount)
14. {
15.     for(; nCount != 0; nCount--);
16. }
17.
18. /**
19.  * @brief    Main program.
20.  * @param    None
21.  * @retval   None
22.  */
23.
24. int main(void)
25. {
26.     /* USART1 config */
27.     USART1_Config();
28.
29.     /* enable adc1 and config adc1 to dma mode */
30.     ADC1_Init();
31.
32.     printf("\r\n -----这是一个 ADC 实验-----\r\n");
33.
34.     while (1)
35.     {

```



```

36.         ADC_ConvertedValueLocal = (float) ADC_ConvertedValue/409
6*3.3; // 读取转换的AD值
37.
38.         printf("\r\n The current AD value = 0x%04X \r\n", ADC_C
onvertedValue);
39.         printf("\r\n The current AD value = %f V \r\n",ADC_Conv
ertedValueLocal);
40.
41.         Delay(0xffffee); // 延
时
42.
43.
44.     }
45. }

```

浏览一遍 main 函数，在调用了用户函数 USART1\_Config() 及 ADC1\_Init() 配置好串口和 ADC 之后，就可以直接使用保存了 ADC 转换值的变量 ADC\_ConvertedValue 了，在 main 函数中并没有对 ADC\_ConvertedValue 重新赋值，这个变量是在什么时候改变的呢？除了可能在中断服务函数修改了变量值，就只有 DMA 有这样的能耐了，而且大家知道，在使用 DMA 传输时，由于不是内核执行的指令，所以修改变量值是绝对不会出现赋值语句的。

#### 10.4.4. ADC 初始化

本实验代码中完全没有使用中断，而 ADC 及 DMA 的配置工作都由用户函数 ADC1\_Init() 完成了。配置完成 ADC 及 DMA 后，ADC 就不停地采集数据，而 DMA 自动地把 ADC 采集得的数据转移至内存中的变量 ADC\_ConvertedValue 中，所以在 main 函数的 while 循环中使用的 ADC\_ConvertedValue 都是实时值。接下来重点分析 ADC1\_Init() 这个函数是如何配置 ADC 的。

ADC1\_Init() 函数使能了 ADC1，并使 ADC1 工作于 DMA 方式。ADC1\_Init() 这个函数是由在用户文件 adc.c 中实现的用户函数，见代码清单 10-3。

代码清单 10-3: ADC1\_Init() 函数

```

1.  /*
2.  * 函数名: ADC1_Init
3.  * 描述   : 无
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */

```

```
8. void ADC1_Init(void)
9. {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }
```

ADC1\_Init()调用了 ADC1\_GPIO\_Config()和 ADC1\_Mode\_Config()。这两个函数的作用分别是配置好 ADC1 所用的 I/O 端口；配置 ADC1 初始化及 DMA 模式。

## ■ 配置 GPIO 端口

关于 ADC 使用的 GPIO 口配置，见代码清单 10-4。

代码清单 10-4: ADC1\_GPIO\_Config()代码。

```
1. /*
2.  * 函数名: ADC1_GPIO_Config
3.  * 描述   : 使能 ADC1 和 DMA1 的时钟，初始化 PC.01
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void ADC1_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* Enable DMA clock */
13.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
14.
15.     /* Enable ADC1 and GPIOC clock */
16.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph
_GPIOIC, ENABLE);
17.
18.     /* Configure PC.01 as analog input */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
20.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
21.     GPIO_Init(GPIOC, &GPIO_InitStructure); // PC1,输入时不用设
置速率
22. }
```

ADC1\_GPIO\_Config() 代码非常简单，就是使能 DMA 时钟，GPIO 时钟及 ADC1 时钟。然后把 ADC1 的通道 11 使用的 GPIO 引脚 PC1 配置成模拟输入模式，在作为 ADC 的输入时，必须使用模拟输入。

这里涉及到 ADC 通道的知识，每个 ADC 通道都对应着一个 GPIO 引脚端口，GPIO 的引脚在设置为模拟输入模式后可用于模拟电压的输入端。

STM32F103VET6 有三个 ADC，这三个 ADC 共用 16 个外部通道，从《STM32 数据手册》的引脚定义可找到 ADC 的通道与 GPIO 引脚的关系。见表 10-2。

表 10-2 ADC 通道引脚图(部分)

| 脚位 |   |   |   |   |   | 管脚名称    | I/O 电平 | 功能      | 可选的复用功能         |       |
|----|---|---|---|---|---|---------|--------|---------|-----------------|-------|
|    |   |   |   |   |   |         |        |         | 默认复用功能          | 重映射功能 |
| 1  | 1 | 8 |   | 5 | 6 | P<br>C0 | /O     | P<br>C0 | ADC123_I<br>N10 |       |
| 2  | 2 | 8 |   | 6 | 7 | P<br>C1 | /O     | P<br>C1 | ADC123_I<br>N11 |       |
| 3  | 2 | 6 | 0 | 7 | 8 | P<br>C2 | /O     | P<br>C2 | ADC123_I<br>N12 |       |
| 4  | 3 |   | 1 | 8 | 9 | P<br>C3 | /O     | P<br>C3 | ADC123_I<br>N13 |       |

表中的引脚名称标注中出现的 ADC12\_INx(x 表示 4~9 或 14~15 之间的整数)，表示这个引脚可以是 ADC1\_INx 或 ADC2\_INx。例如：ADC12\_IN9 表示这个引脚可以配置为 ADC1\_IN9，也可以配置为 ADC2\_IN9。

本实验中使用的 PC1 对应的默认复用功能为 ADC123\_IN11，也就是说可以使用 ADC1 的通道 11、ADC2 的通道 11 或 ADC3 的通道 11 来采集 PC1 上的模拟电压数据，我们选择 ADC1 的通道 11 来采集。

## ■ 配置 DMA

ADC 模式及其 DMA 传输方式都是在用户函数 ADC1\_Mode\_Config()中实现的，见代码清单 10-5。

代码清单 10-5: ADC1\_Mode\_Config()函数代码

```

1.  /* 函数名: ADC1_Mode_Config
2.   * 描述   : 配置 ADC1 的工作模式为 MDA 模式
3.   * 输入   : 无
4.   * 输出   : 无
5.   * 调用   : 内部调用
6.   */
7.  static void ADC1_Mode_Config(void)
8.  {
9.      DMA_InitTypeDef DMA_InitStructure;
10.     ADC_InitTypeDef ADC_InitStructure;
11.
12.     /* DMA channel1 configuration */
13.     DMA_DeInit(DMA1_Channel1);
14.     DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
15.
16.     /*ADC 地址*/
17.     DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
18.     /*内存地址*/
19.     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //外设为数据源
20.     DMA_InitStructure.DMA_BufferSize = 1;
21.     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
22.     /*外设地址固定*/
23.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
24.     /*内存地址固定*/
25.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //半字
26.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
27.     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular; //循环传输
28.     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
29.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
30.     DMA_Init(DMA1_Channel1, &DMA_InitStructure);
31.
32.     /* Enable DMA channel1 */
33.     DMA_Cmd(DMA1_Channel1, ENABLE);
34.
35.     /* ADC1 configuration */
36.     ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
37.     /*独立 ADC 模式*/
38.     ADC_InitStructure.ADC_ScanConvMode = DISABLE ;
39.     /*禁止扫描模式，扫描模式用于多通道采集*/
40.     ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
41.     /*开启连续转换模式，即不停地进行 ADC 转换*/
42.     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; /*不使用外部触发转换*/
43.     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;

```

```

44. /*采集数据右对齐*/
45.     ADC_InitStructure.ADC_NbrOfChannel = 1;      /*要转换的通道数目
1*/
46.     ADC_Init(ADC1, &ADC_InitStructure);
47.
48. /*配置 ADC 时钟，为 PCLK2 的 8 分频，即 9Hz*/
49.     RCC_ADCCLKConfig(RCC_PCLK2_Div8);
50.     /*配置 ADC1 的通道 11 为 55.5 个采样周期 */
51.     ADC-RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_Sampl
eTime_55Cycles5);
52.
53.     /* Enable ADC1 DMA */
54.     ADC_DMACmd(ADC1, ENABLE);
55.
56.     /* Enable ADC1 */
57.     ADC_Cmd(ADC1, ENABLE);
58.
59.     /*复位校准寄存器 */
60.     ADC_ResetCalibration(ADC1);
61.     /*等待校准寄存器复位完成 */
62.     while(ADC_GetResetCalibrationStatus(ADC1));
63.
64.     /* ADC 校准 */
65.     ADC_StartCalibration(ADC1);
66.     /* 等待校准完成*/
67.     while(ADC_GetCalibrationStatus(ADC1));
68.
69.     /* 由于没有采用外部触发，所以使用软件触发 ADC 转换 */
70.     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
71. }

```

ADC 的 DMA 配置部分跟串口 DMA 配置部分很类似，它的 DMA 整体上被配置为：

使用 DMA1 的通道 1，数据从 ADC 外设的数据寄存器(ADC1\_DR\_Address)转移到内存(ADC\_ConvertedValue 变量)，内存、外设地址都固定，每次传输的数据大小为半字(16 位)，使用 DMA 循环传输模式。

其中 ADC1 外设的 DMA 请求通道为 DMA1 的通道 1，初始化时要注意。

DMA 传输的外设地址 ADC1\_DR\_Address 是一个自定义的宏：

```
1. #define ADC1_DR_Address    ((u32)0x40012400+0x4c)
```

ADC\_DR 数据寄存器保存了 ADC 转换后的数值，以它作为 DMA 的传输源地址。它的地址是由 ADC1 外设的基地址(0x4001 2400) 加上 ADC 数据寄存器(ADC\_DR)的地址偏移 (0x4c)计算得到的。见表 10-3 及表 10-4。

表 10-3 《SM32 参考手册》的 ADC1 起始地址说明

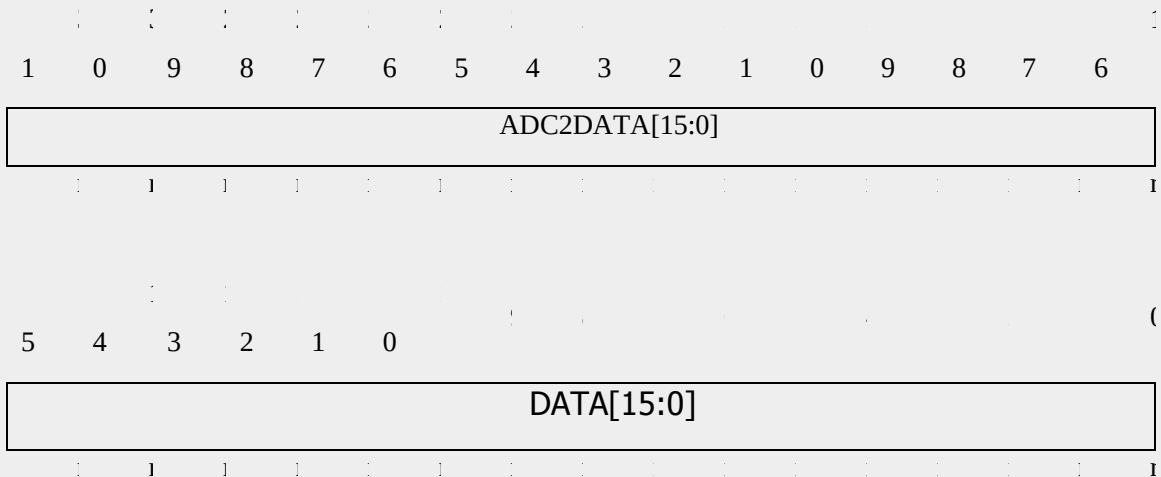
| 起始地址 | 外设 |
|------|----|
|------|----|

|                           |      |
|---------------------------|------|
| 0x4001 2800 - 0x4001 2BFF | ADC2 |
| 0x4001 2400 - 0x4001 27FF | ADC1 |

表 10-4 ADC\_DR 寄存器描述及其地址偏移

ADC 规则数据寄存器(ADC\_DR)

地址偏移: 0x4C



| 位<br>数     | 描述                                                                                                                      |
|------------|-------------------------------------------------------------------------------------------------------------------------|
| 位<br>31:16 | <b>ADC2DATA[15:0]:</b> ADC2 转换的数据 (ADC2 data)<br>- 在 ADC1 中: 双模式下, 这些位包含了 ADC2 转换的规则通道数据。<br>- 在 ADC2 和 ADC3 中: 不使用这些位。 |
| 位<br>15:0  | <b>DATA[15:0]:</b> 规则转换的数据 (Regular data)<br>这些位为只读, 包含了规则通道的转换结果。数据是左对齐或右对齐。                                           |

## ■ 配置 ADC 模式

从 ADC1\_Mode\_Config()函数代码的第 32 行开始, 为 ADC 模式的配置, 主要为对 ADC 的初始化结构体进行赋值。下面对这些结构体成员进行介绍:

### (1) .ADC\_Mode

STM32 具有多个 ADC，而不同的 ADC 又是共用通道的，当两个 ADC 采集同一个通道的先后顺序、时间间隔不同，就演变出了各种各样的模式，如同步注入模式、同步规则模式等 10 种，根据应用要求选择适合的模式以适应采集数据的要求。

本实验用于测量电阻分压后的电压值，要求不高，只使用一个 ADC 就可以满足要求了，所以本成员被赋值为 `ADC_Mode_Independent` (独立模式)。

## (2) `.ADC_ScanConvMode`

当有多个通道需要采集信号时，可以把 ADC 配置为按一定的顺序来对各个通道进行扫描转换，即轮流采集各通道的值。若采集多个通道，必须开启此模式。

本实验只采集一个通道的信号，所以 **DISABLE**(禁止)使用扫描转换模式。

## (3) `.ADC_ContinuousConvMode`

连续转换模式，此模式与单次转换模式相反，单次转换模式 ADC 只采集一次数据就停止转换。而连续转换模式则在上一次 ADC 转换完成后，立即开启下一次转换。

本实验需要循环采集电压值，所以 **ENABLE**(使能)连续转换模式。

## (4) `.ADC_ExternalTrigConv`

ADC 需要在接收到触发信号才开始进行模数转换，这些触发信号可以是外部中断触发(`EXTI` 线)、定时器触发。这两个为外部触发信号，如果不使用外部触发信号可以使用软件控制触发。

本实验中使用软件控制触发所以该成员被赋值为 `ADC_ExternalTrigConv_None` (不使用外部触发)。

## (5) `.ADC_DataAlign`

数据对齐方式。ADC 转换后的数值是被保存到数据寄存器(`ADC_DR`)的 0~15 位或 16~32 位，数据宽度为 16 位，而 ADC 转换精度为 12 位。把 12 位的数据保存到 16 位的区域，就涉及左对齐和右对齐的问题。这里的左、右对齐跟 word 文档中的文本左、右对齐是一样的意思。

左对齐即 ADC 转换的数值最高位 D12 与存储区域的最高位 Bit 15 对齐，存储区域的低 4 位无意义。右对齐则相反，ADC 转换的数值最低位 D0 保存在存储区域的最低位 Bit 0，高 4 位无意义。见表 10-5(本例程使用的是规则组)。

表 10-5 ADC 数据对齐  
数据右对齐

注入组

|     |     |     |     |    |    |   |   |   |   |   |   |   |   |   |   |
|-----|-----|-----|-----|----|----|---|---|---|---|---|---|---|---|---|---|
| S   | S   | S   | S   |    |    |   |   |   |   |   |   |   |   |   |   |
| EXT | EXT | EXT | EXT | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

规则组

|   |   |   |   |    |    |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|----|----|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 |    |    |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

数据左对齐

注入组

|     |    |    |   |   |   |   |   |   |   |   |   |   |  |  |  |
|-----|----|----|---|---|---|---|---|---|---|---|---|---|--|--|--|
| S   | D  | D  | D |   |   |   |   |   |   |   |   |   |  |  |  |
| EXT | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |  |  |

规则组

|    |    |   |   |   |   |   |   |   |   |   |   |  |  |  |  |
|----|----|---|---|---|---|---|---|---|---|---|---|--|--|--|--|
| D  | D  | D | D |   |   |   |   |   |   |   |   |  |  |  |  |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |  |  |  |

本实验中 ADC 的转换值最后被保存在一个 16 位的变量之中，选择 ADC\_DataAlign\_Right (右对齐) 会比较方便。

#### (6) .ADC\_NbrOfChannel

这个成员保存了要进行 ADC 数据转换的通道数，可以为 1~16 个。

本实验中只需要采集 PC1 这个通道，所以把成员赋值为 1 就可以了。

填充完结构体，就可以调用外设初始化函数进行初始化了，ADC 的初始化使用 ADC\_Init()函数，初始化完成后别忘记调用 ADC\_Cmd() 函数来使能 ADC



外设，用 `ADC_DMACmd()` 函数来使能 ADC 的 DMA 接口。在本实验中初始化 ADC1。

## ■ ADC 转换时间配置

配置好了 ADC 的模式，还要设置 ADC 的时钟(ADCCLK)，ADC 时钟频率越高，转换速度也就越快，但 ADC 时钟有上限值，不能超过 14MHz。

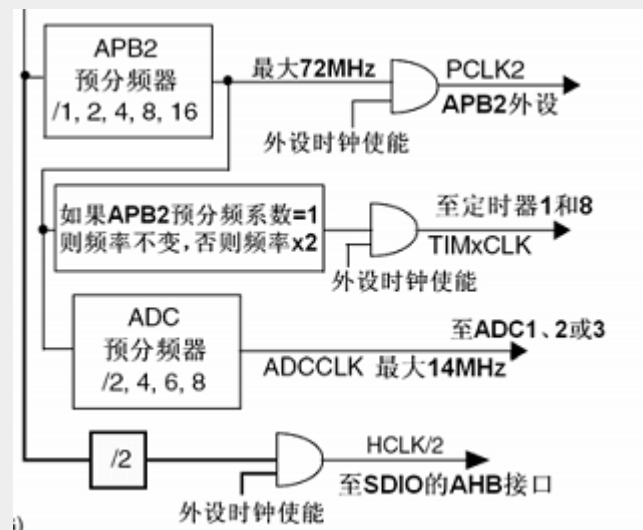


图 10-3 ADC 时钟

配置 ADC 时钟，可使用函数 `RCC_ADCCLKConfig()` 配置，见图 10-4

```
void RCC_ADCCLKConfig ( uint32_t RCC_PCLK2 )
```

Configures the ADC clock (ADCCLK).

### Parameters:

**RCC\_PCLK2,:** defines the ADC clock divider. This clock is derived from the APB2 clock (PCLK2). This parameter can be one of the following values:

- `RCC_PCLK2_Div2`: ADC clock =  $PCLK2/2$
- `RCC_PCLK2_Div4`: ADC clock =  $PCLK2/4$
- `RCC_PCLK2_Div6`: ADC clock =  $PCLK2/6$
- `RCC_PCLK2_Div8`: ADC clock =  $PCLK2/8$

### Return values:

None

14MHz

可以对 PCLK2 进行 2、4、6、8 分频，但 ADCCLK 时钟的最大值为 14MHz

图 10-4 ADCCLK 时钟配置函数

ADC 的时钟(ADCCLK)为 ADC 预分频器的输出，而 ADC 预分频器的输入则为高速外设时钟(PCLK2)。使用 `RCC_ADCCLKConfig()`库函数实质就是设置 ADC 预分频器的分频值，可设置为 PCLK2 的 2、4、6、8 分频。

PCLK2 的常用时钟频率为 72MHz，而 ADCCLK 必须低于 14MHz，所以在这个情况下，ADCCLK 最高频率为 PCLK2 的 8 分频，即  $ADCCLK=9MHz$ 。若希望使 ADC 以最高的频率 14MHz 运行，可以把 PCLK2 配置为 56MHz，然后再 4 分频得到 ADCCLK。

ADC 的转换时间不仅与 ADC 的时钟有关，还与采样周期相关。

每个不同的 ADC 通道，都可以设置为不同的的采样周期。配置时用库函数 `ADC_RegularChannelConfig()`，它的说明见图 10-5。

```
void ADC_RegularChannelConfig ( ADC_TypeDef * ADCx,
                                uint8_t      ADC_Channel,
                                uint8_t      Rank,
                                uint8_t      ADC_SampleTime
                                )
```

Configures for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.

**Parameters:**

**ADCx,:** where x can be 1, 2 or 3 to select the ADC peripheral.

**ADC\_Channel,:** the ADC channel to configure. This parameter can be one of the following values:

- ADC\_Channel\_0: ADC Channel0 selected
- ADC\_Channel\_1: ADC Channel1 selected
- ADC\_Channel\_2: ADC Channel2 selected
- ADC\_Channel\_3: ADC Channel3 selected
- ADC\_Channel\_4: ADC Channel4 selected
- ADC\_Channel\_5: ADC Channel5 selected
- ADC\_Channel\_6: ADC Channel6 selected
- ADC\_Channel\_7: ADC Channel7 selected
- ADC\_Channel\_8: ADC Channel8 selected
- ADC\_Channel\_9: ADC Channel9 selected
- ADC\_Channel\_10: ADC Channel10 selected
- ADC\_Channel\_11: ADC Channel11 selected
- ADC\_Channel\_12: ADC Channel12 selected
- ADC\_Channel\_13: ADC Channel13 selected
- ADC\_Channel\_14: ADC Channel14 selected
- ADC\_Channel\_15: ADC Channel15 selected
- ADC\_Channel\_16: ADC Channel16 selected
- ADC\_Channel\_17: ADC Channel17 selected

**Rank,:** The rank in the regular group sequencer. This parameter must be between 1 to 16.

**ADC\_SampleTime,:** The sample time value to be set for the selected channel. This parameter can be one of the following values:

- ADC\_SampleTime\_1Cycles5: Sample time equal to 1.5 cycles
- ADC\_SampleTime\_7Cycles5: Sample time equal to 7.5 cycles
- ADC\_SampleTime\_13Cycles5: Sample time equal to 13.5 cycles
- ADC\_SampleTime\_28Cycles5: Sample time equal to 28.5 cycles
- ADC\_SampleTime\_41Cycles5: Sample time equal to 41.5 cycles
- ADC\_SampleTime\_55Cycles5: Sample time equal to 55.5 cycles
- ADC\_SampleTime\_71Cycles5: Sample time equal to 71.5 cycles
- ADC\_SampleTime\_239Cycles5: Sample time equal to 239.5 cycles

**Return values:**  
None

选择要配置的 ADC 通道

RANK 参数: 配置的数值为多通道扫描时, 此通道的采样顺序

可配置的采样周期: 1.5、7.5、13.5 等 ADC 时钟周期

图 10-5 规则通道配置函数

ADC\_RegularChannelConfig()函数中的 RANK 值是指在多通道扫描模式时, 本通道的扫描顺序。例如通道 1、4、7 的 RANK 值分别为被配置为 3、2、1 的话, 在 ADC 扫描时, 扫描的顺序为通道 7、通道 4、最后扫描通道 1。

本实验中只采集一个 ADC 通道, 把以把 ADC1 的通道 11RANK 值配置为 1。

ADC\_SampleTime 的参数值则用于配置本通道的采样周期, 最短可配置为 1.5 个采样周期, 这里的周期指 ADCCLK 时钟周期。

本实验中把 ADC1 通道 11 配置为 55.5 个采样周期，而 ADCCLK 在前面已经配置为 9MHz，根据 STM32 的 ADC 采样时间计算公式：

$$T_{\text{CONV}} = \text{采样周期} + 12.5 \text{ 个周期}$$

公式中的采样周期就是本函数中配置的 ADC\_SampleTime，而后边加上的 12.5 个周期为固定的数值。

所以，本实验中 ADC1 通道 11 的转换时间  $T_{\text{CONV}} = (55.5 + 12.5) * 1/9 \approx 7.56\mu\text{s}$ 。

## ■ ADC 自校准

在开始 ADC 转换之前，需要启动 ADC 的自校准。ADC 有一个内置自校准模式，校准可大幅减小因内部电容器组的变化而造成的精度误差。在校准期间，在每个电容器上都会计算出一个误差修正码(数字值)，这个码用于消除在随后的转换中每个电容器上产生的误差。

见代码清单 10-6。这是在 ADC1\_Mode\_Config()函数中的 ADC 自校准时调用的库函数和使用步骤。

代码清单 10-6：ADC 自校准

```
1. /*复位校准寄存器 */
2. ADC_ResetCalibration(ADC1);
3. /*等待校准寄存器复位完成 */
4. while(ADC_GetResetCalibrationStatus(ADC1));
5.
6. /* ADC 校准 */
7. ADC_StartCalibration(ADC1);
8. /* 等待校准完成*/
9. while(ADC_GetCalibrationStatus(ADC1));
```

在调用了复位校准函数 ADC\_ResetCalibration() 和开始校准函数 ADC\_StartCalibration() 后，要检查标志位等待校准完成，确保完成后才开始 ADC 转换。建议在每次上电后都进行一次自校准。

在校准完成后，就可以开始进行 ADC 转换了。本实验代码中配置的 ADC 模式为软件触发方式，我们可以调用库函数 ADC\_SoftwareStartConvCmd() 来开启软件触发。其说明见图 10-6。

```
void ADC_SoftwareStartConvCmd ( ADC_TypeDef * ADCx,  
                                FunctionalState NewState  
                                )
```

Enables or disables the selected ADC software start conversion .

**Parameters:**

**ADCx,:** where x can be 1, 2 or 3 to select the ADC peripheral.  
**NewState,:** new state of the selected ADC software start conversion. This parameter can be: ENABLE or DISABLE.

**Return values:**

None

图 10-6 ADC 软件触发控制函数

调用这个函数使能了 ADC1 的软件触发后，ADC 就开始进行转换了，每次转换完成后，由 DMA 控制器把转换值从 ADC 数据寄存器(ADC\_DR)中转移到变量 ADC\_ConvertedValue 中，当 DMA 传输完成后，在 main 函数中使用的 ADC\_ConvertedValue 的内容就是 ADC 转换值了。

## ■ volatile 变量

现在我们来认识 ADC\_ConvertedValue 这个变量：

```
1. // ADC1 转换的电压值通过 MDA 方式传到 flash  
2. extern __IO u16 ADC_ConvertedValue;  
3.
```

ADC\_ConvertedValue 在文件 adc.c 中定义，这个。这里要注意一点的是，这个变量要用 c 语言的 volatile 关键字来修饰，为的是让编译器不要去优化这个变量。这样每次用到这两个变量时都要回到相应变量的内存中去取值，而 volatile 字面意思就是“可变的，不确定的”。

例如：不使用 volatile 关键字修饰的变量 a 在被访问的时候可能会直接从 CPU 的寄存器中取出（因为之前变量 a 被访问过，也就是说之前就从内存中取出 a 的值保存到某个 CPU 寄存器中），之所以直接从寄存器中取值，而不去内存中取值，是因为编译器优化代码的结果（访问 CPU 寄存器比访问内存快的

多)。这里的 CPU 寄存器指 R0、R1 等 CPU 通用寄存器，用于 CPU 运算是暂存数据的，不是指外设中的寄存器。

用 `volatile` 声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。

因为 `ADC_ConvertedValue` 这个变量值随时都是会被 DMA 控制器改变的，所以我们用 `volatile` 来修饰它，确保每次读取到的都是实时的 ADC 转换值。

#### 10.4.5. 计算电压值

回到 `main` 函数中循环打印电压值部分见代码清单错误!未找到引用源。-7。

代码清单错误!未找到引用源。-7：计算并打印电压值

```
1. while (1)
2. {
3.     ADC_ConvertedValueLocal = (float) ADC_ConvertedValue/4096*3.
3; // 读取转换的 AD 值
4.
5.     printf("\r\n The current AD value = 0x%04X \r\n", ADC_Conve
rtedValue);
6.     printf("\r\n The current AD value = %f V \r\n",ADC_Converte
dValueLocal);
7.
8.     Delay(0xffffee); // 延时
9.
10.
11. }
```

`float` 型变量 `ADC_ConvertedValueLocal` 保存了由转换值计算出来的电压值，其计算公式是 ADC 通用的：

实际电压值 = ADC 转换值\*LSB

STM32 的 ADC 的精度为 12 位，而野火板子中 `Vref+` 接的参考电压值为 3.3v，所以  $LSB = 3.3/2^{12}$ 。

#### 10.4.6. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：

当旋转开发板开发板上的滑动变阻器时，ADC1 转换的电压值则会改变。  
板载的是 20K 的精密电阻，旋转的圈数要多点才能看到 ADC 值的明显变化。

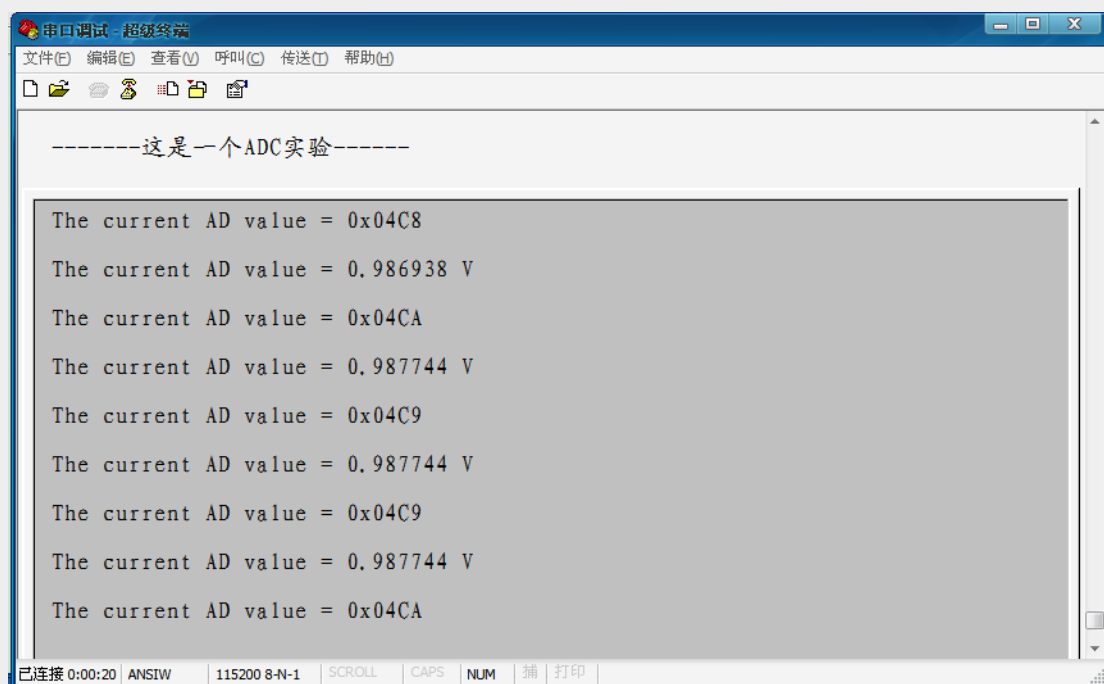


图 10-7 ADC 采集外部电压实验

### 10.5. ADC1 芯片内部温度检测实验

STM32 的 ADC 除了可以用外部通道来采集外部模拟电压信号之外，还有两个内部通道 16、17。其中通道 16 被连接至芯片上的温度传感器，而通道 17 被连接至内部电源模块。图 10-8。

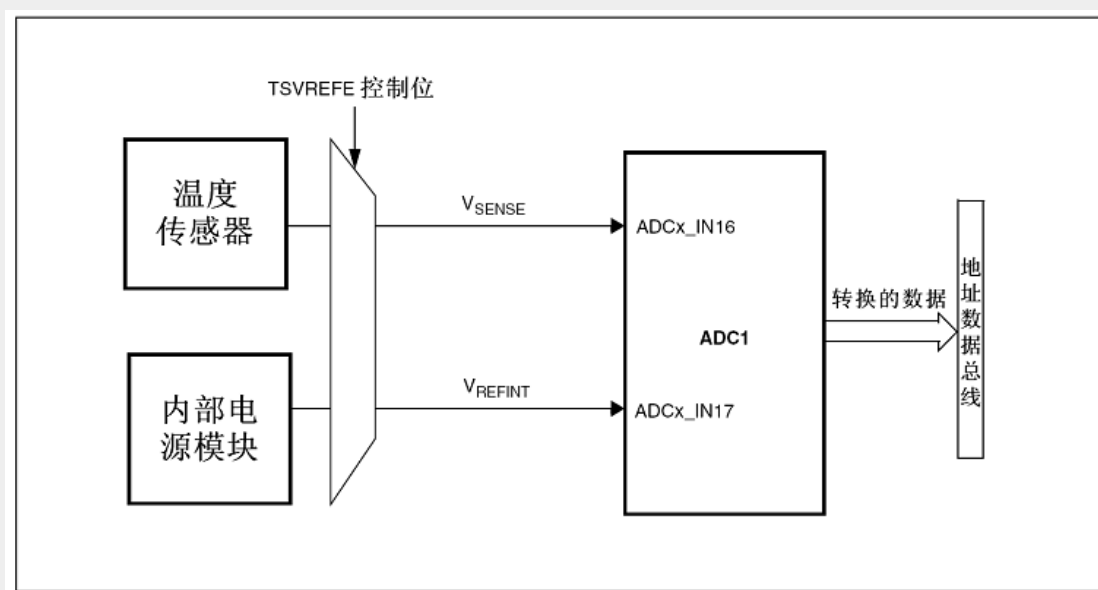


图 10-8 ADC 内部通道

所以这两个通道可以用来检测 STM32 芯片的温度和电源的状态，达到监测芯片运行状态的目的。

通道 16 把内部温度传感器输出的模拟电压转换成数字值，通过公式运算我们就可以得出温度值。内部温度传感器支持的温度范围从 -40 到 125 摄氏度。精度较差，误差为+（-）1.5 度左右，因此更适合于检测温度的变化，而不是测量绝对的温度。如果需要测量精确的温度，应该使用一个外置的温度传感器。

本小节以实例讲解如何使用内部温度传感器检测芯片温度。



### 10.5.1. 实验描述及工程文件清单

#### 实验描述:

串口 1(USART1)向电脑的超级终端以 1s 为时间间隔打印当前 STM32 芯片内部的温度值。（此实验内容与实验 PC1\_ADC1 连接外部电压的内容很相似，可以互相参考）

#### 硬件连接:

无 （温度传感器在芯片内部和 ADCx\_IN16 输入通道相连接）

#### 库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_adc.c

FWlib/stm32f10x\_dma.c

#### 用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

USER/ tempad.c

### 10.5.2. 配置工程环境

本温度传感器实验中我们用到了 GPIO、RCC、USART、DMA 及 ADC 外设，所以我们要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_usart.c`、`stm32f10x_dma.c`、`stm32f10x_adc.c`，添加旧工程中的外设用户文件 `usart1.c`，新建 `tempad.c` 及 `tempad.h` 文件，并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。

代码清单 10-8: ADC 内部温度检测例程的 `stm32f10x_conf.h` 文件配置

```
14. /**
15.  *****
16.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
17.  * @author    MCD Application Team
18.  * @version    V3.5.0
19.  * @date      08-April-2011
20.  * @brief     Library configuration file.
21.  *****
22.  /
23. #include "stm32f10x_adc.h"
24. #include "stm32f10x_dma.h"
25. #include "stm32f10x_gpio.h"
26. #include "stm32f10x_rcc.h"
27. #include "stm32f10x_usart.h"
```

### 10.5.3. main 文件

从 main 文件开始分析。

代码清单 10-9: ADC 内部温度检测例程的 `stm32f10x_conf.h` 文件配置

```
1. #include "stm32f10x.h"
2. #include "usart1.h"
3. #include "tempad.h"
4.
5. /* ADC1 转换的电压值通过 MDA 方式传到 sram */
6. extern __IO u16 ADC_ConvertedValue;
7.
8.
9. /* 计算后的温度值 */
10. u16 Current_Temp;
11.
12.
13. /* 软件延时 */
14. void Delay(__IO u32 nCount)
15. {
16.     for(; nCount != 0; nCount--);
17. }
```

```

18.
19.      /*
20.      * 函数名: main
21.      * 描述   : 主函数
22.      * 输入    : 无
23.      * 输出    : 无
24.      */
25.      int main(void)
26.      {
27.
28.          /* USART1 config 115200 8-N-1 */
29.          USART1_Config();
30.
31.          Temp_ADC1_Init();
32.
33.          printf("\r\n 这是一个内部温度传感器实验 \r\n");
34.          printf(" \r\n Print current Temperature \r\n");
35.
36.          while (1)
37.          {
38.
39.              Delay(0xffffee);          // 延时
40.
41.              //计算方法 1
42.              //Current_Temp= (1.43-
ADC_ConvertedValue*3.3/4096)*1000 / 4.3+ 25 ;
43.
44.              //计算方法 2
45.              Current_Temp=(V25-
ADC_ConvertedValue)/Avg_Slope+25;
46.
47.              //10 进制显示
48.              printf("\r\n The current temperature= %3d °C
\r\n", Current_Temp);
49.
50.              //16 进制显示
51.              // printf("\r\n The current temperature= %04x \r\n", Cu
rrent_Temp);
52.          }
53.      }
54.      /*** (C) COPYRIGHT 2011 野火嵌入式开发工作
室 *****END OF FILE*****/

```

在使用 USART1\_Config() 和 Temp\_ADC1\_Init()配置好串口及初始化好通道 16 之后，ADC 转换得的值被 DMA 传输到 ADC\_ConvertedValue 变量之中。

STM32 内部温度传感器检测温度的计算公式：

$$\text{温度}(^{\circ}\text{C}) = \{(V25 - V_{\text{SENSE}}) / \text{Avg\_Slope}\} + 25$$

$V_{\text{SENSE}}$  即为我们通过 ADC 通道 16 测得的 ADC 转换值，变量。

V25 为 25° C 时的 ADC 转换值，常量。

Avg\_Slope 为温度与  $V_{SENSE}$  曲线的平均斜率(单位为  $mV/^\circ C$  或  $\mu V/^\circ C$ )，常量。

V25 和 Avg\_Slope 两个参数可以在《STM32 数据手册》的电气特性小节中找到，见表 10-6。

表 10-6 STM32 内部温度传感器特性

| 符号        | 参数                  | 最小值  | 典型值     | 最大值     | 单位            |
|-----------|---------------------|------|---------|---------|---------------|
| TL        | VSENSE 相对于温度的线性度    |      | $\pm 1$ | $\pm 2$ | $^\circ C$    |
| Avg_Slope | 平均斜率                | 4.0  | 4.3     | 4.6     | $mV/^\circ C$ |
| V25       | 在 $25^\circ C$ 时的电压 | 1.34 | 1.43    | 1.52    | V             |
| tSTART    | 建立时间                | 4    |         | 10      | $\mu s$       |
| TS_temp   | 当读取温度时，ADC 采样时间     |      |         | 17.1    | $\mu s$       |

Main 文件中的第 45 行代码表示 ADC 转换值转化为温度的计算表达式。其中 V25 和 AVG\_SLOPE 是在文件 tempad.h 中自定义的宏，见代码清单 10-10。

代码清单 10-10: V25 和 AVG\_SLOPE 宏

```

1. //对于 12 位的 ADC，3.3V 的 ADC 值为 0xfff,温度为 25 度时对应的电压值为
   1.43V 即 0x6EE
2. #define V25 0x6EE
3.
4. //斜率 每摄氏度 4.3mV 对应每摄氏度 0x05
5. #define AVG_SLOPE 0x05

```

这两个宏都是根据数据手册来定义的。

#### 10.5.4. ADC 初始化

接下来我们分析检测温度的 ADC 初始化，这与 ADC 采集外部数据时很类似。

初始化是在 Temp\_ADC1\_Init()函数中完成的，见代码清单 10-11。

代码清单 10-11: Temp\_ADC1\_Init()函数

```

1.  /*
2.  * 函数名: Temp_ADC1_Init
3.  * 描述   : 无
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void Temp_ADC1_Init(void)
9. {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }
```

Temp\_ADC1\_Init() 调用了 ADC1\_GPIO\_Config() 和 ADC1\_Mode\_Config()。这两个函数的作用分别配置好 ADC1 及 DMA 时钟；配置 ADC1 初始化及 DMA 模式，见代码清单 10-12 和代码清单 10-13。

代码清单 10-12: ADC1\_GPIO\_Config()函数

```

1.  /*
2.  * 函数名: ADC1_GPIO_Config
3.  * 描述   : 使能 ADC1 和 DMA1 的时钟
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void ADC1_GPIO_Config(void)
9. {
10.
11.     /* Enable DMA clock */
12.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
13.
14.     /* Enable ADC1 and GPIOC clock */
15.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
16.
17. }
```

代码清单 10-13: ADC1\_Mode\_Config()函数

```

1.  /* 函数名: ADC1_Mode_Config
2.  * 描述   : 配置 ADC1 的工作模式为 MDA 模式
3.  * 输入   : 无
```

```
4.  * 输出 : 无
5.  * 调用 : 内部调用
6.  */
7.  static void ADC1_Mode_Config(void)
8.  {
9.      DMA_InitTypeDef DMA_InitStructure;
10.     ADC_InitTypeDef ADC_InitStructure;
11.
12.     /* DMA channel1 configuration */
13.
14.     DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
15.     DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
16.     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
17.     DMA_InitStructure.DMA_BufferSize = 1;
18.     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
19.     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
20.     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
21.     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
22.     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
23.     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
24.     DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
25.     DMA_Init(DMA1_Channel1, &DMA_InitStructure);
26.
27.     /* Enable DMA channel1 */
28.     DMA_Cmd(DMA1_Channel1, ENABLE);
29.
30.     /* ADC1 configuration */
31.     ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
32.     ADC_InitStructure.ADC_ScanConvMode = ENABLE;
33.     ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
34.     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
35.     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
36.     ADC_InitStructure.ADC_NbrOfChannel = 1;
37.     ADC_Init(ADC1, &ADC_InitStructure);
38.
39.     /*配置ADC时钟, 为PCLK2的8分频, 即9MHz*/
40.     RCC_ADCCLKConfig(RCC_PCLK2_Div8);
41.
42.     /* ADC1 regular channel16 configuration */
43.     /*设置采样通道IN16, 设置采样时间
44.     ADC-RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_239Cycles5);
45.
46.     /*使能温度传感器和内部参考电压
```

```
47. ADC_TempSensorVrefintCmd(ENABLE);

48.
49. /* Enable ADC1 DMA */
50. ADC_DMACmd(ADC1, ENABLE);
51.
52. /* Enable ADC1 */
53. ADC_Cmd(ADC1, ENABLE);
54.
55. /* Enable ADC1 reset calibration register */
56. ADC_ResetCalibration(ADC1);

57. /* Check the end of ADC1 reset calibration register */
58. while(ADC_GetResetCalibrationStatus(ADC1));

59.
60. /* Start ADC1 calibration */
61. ADC_StartCalibration(ADC1);

62. /* Check the end of ADC1 calibration */
63. while(ADC_GetCalibrationStatus(ADC1));
64.
65. /* Start ADC1 Software Conversion */
66. ADC_SoftwareStartConvCmd(ADC1, ENABLE);
67. }
```

这个部分对 ADC 的配置与 ADC 采集数据例程中的区别为第 44~47 行，使用 ADC\_RegularChannelConfig()函数配置时选择配置 ADC 通道 16，并且把采样周期改为更长的 ADC\_SampleTime\_239Cycles5（239.5 个周期），配置成如此长的时间主要是因为根据手册推荐的内部温度传感器的 ADC 转换时间为 17.1us，本代码这样的配置则为 28us。

在 ADC1\_GPIO\_Config() 代码中的第 47 行：

```
1. //使能温度传感器和内部参考电压
2. ADC_TempSensorVrefintCmd(ENABLE);
```

它调用了库函数 ADC\_TempSensorVrefintCmd()函数，这个函数的功能是使能或关闭内部温度传感器。它是使用内部温度传感器时必须调用的，默认的温度传感器是不开启的，当然也就测量不到正确的温度值。

#### 10.5.5. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序

下载到开发板，即可看到超级终端打印出如下信息：（检测到的是芯片的内部温度）

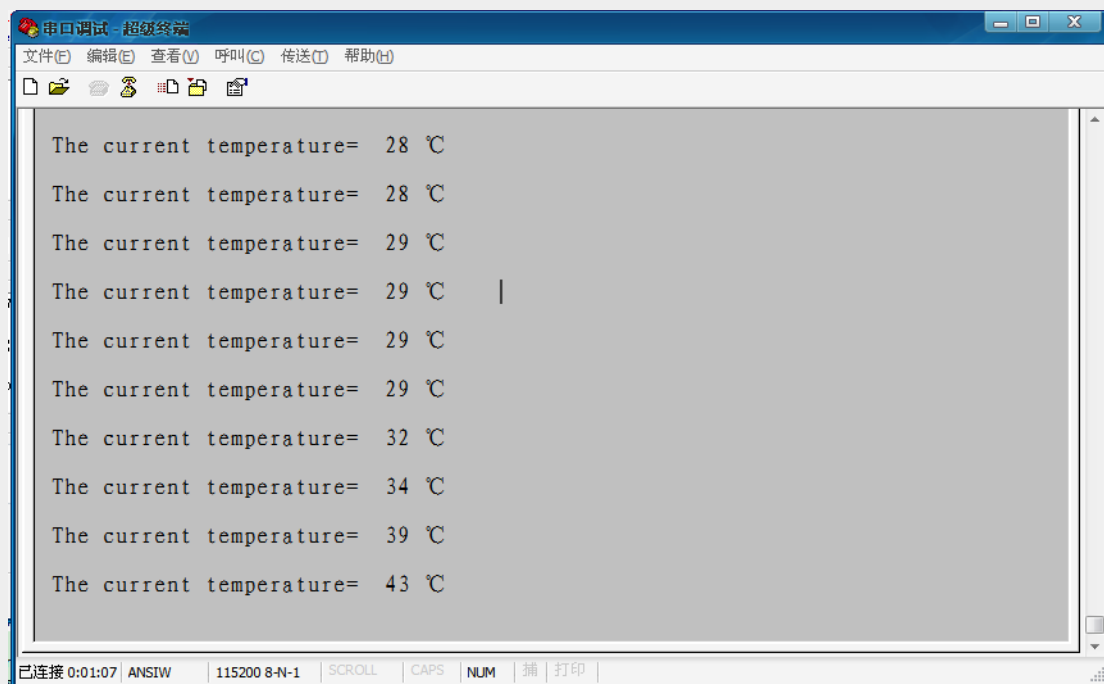


图 10-9 温度传感器实验

当用加热的电铬铁轻轻地放在我们的芯片上（或者用热风筒吹芯片）时会看到温度有明显的变化。





## 11. FSMC-液晶显示-英文

### 11.1. LCD 控制器简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为 STM32 内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，STM32 芯片通过驱动芯片来控制液晶屏和触摸屏。以野火 3.2 寸液晶屏(240\*320)为例，它使用 *ILI9341* 芯片控制液晶屏，通过 *TSC2046* 芯片控制触摸屏。关于触摸我们在后面的章节中再讲，这里我们先学习液晶显示的内容。

#### 11.1.1. ILI9341 控制器结构

液晶屏的控制芯片内部结构非常复杂，见下图。最主要的是位于中间的GRAM(Graphics RAM)，可以理解为显存。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；使用 SPI 接口还是 8080 接口与 MCU 进行通讯；使用 8080 接口的哪种模式。MCU 通过 SPI 或 8080 接口与 ILI9341 进行通讯，从而访问它的控制寄存器(CR)、地址计数器(AC)、及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器(LED Controller)。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。



16 位像素点格式

|     |     |     |     |     |     |     |     |    |    |    |    |    |    |    |    |    |    |
|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|
| D17 | D16 | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|----|

|      |      |      |      |      |  |      |      |      |      |      |      |      |      |      |      |      |  |
|------|------|------|------|------|--|------|------|------|------|------|------|------|------|------|------|------|--|
| R[4] | R[3] | R[2] | R[1] | R[0] |  | G[5] | G[4] | G[3] | G[2] | G[1] | G[0] | B[4] | B[3] | B[2] | B[1] | B[0] |  |
|------|------|------|------|------|--|------|------|------|------|------|------|------|------|------|------|------|--|

图中的是默认 18 条数据线时，像素点三原色的分配状况，D1~D5 为蓝色，D6~D11 为绿色，D13~D17 为红色。这样分配有 D0 和 D12 位是无效的。若使用 16 根数据线传送像素点的数据，则 D0~D4 为蓝色，D5~D10 为绿色，D11~D15 为红色，使得刚好使用完整的 16 位。

RGB 比例为 5: 6: 5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。如黑色的编码为 0x0000，白色的编码为 0xffff，红色为 0xf800。

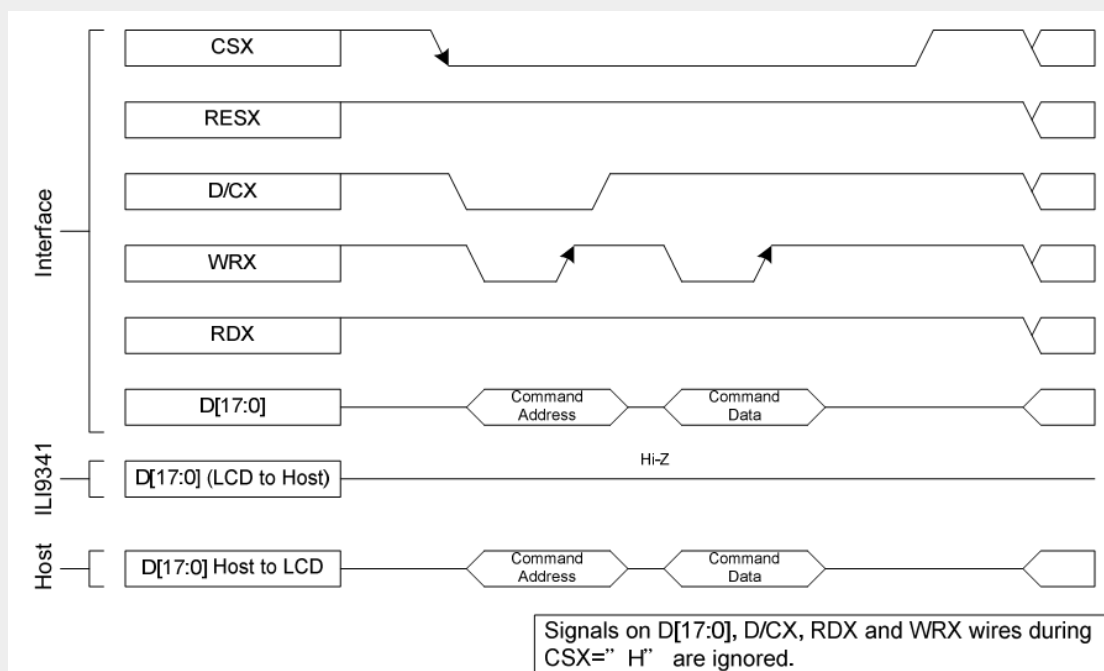
### 11.1.3. ILI9341 的通讯时序

目前，大多数的液晶控制器都使用 8080 或 6800 接口与 MCU 进行通讯，它们的时序十分相似，野火以 ILI9341 使用的 8080 通讯时序进行分析，实际上 ILI9341 也可以使用 SPI 接口来控制。

ILI9341 的 8080 接口有 5 条基本的控制信号线：

|             |         |
|-------------|---------|
| <b>CSX</b>  | 片选      |
| <b>WRX</b>  | 写使能     |
| <b>RDX</b>  | 读使能     |
| <b>D/CX</b> | 区分数据和命令 |
| <b>RESX</b> | 复位      |

其中带 X 的表示低电平有效。除了控制信号，还有数据信号线，它的数目不定，可根据 ILI9341 框图中的 IM[3:0]来设定，这部分一般由制作液晶屏的厂家完成。为便于传输像素点数据，野火使用的液晶屏设定为 16 条数据线 D[15:0]。使用 8080 接口的写命令时序图见下图。



使用 18 条数据线的 8080 接口写命令时序

由图可知，写命令时序由 CSX 信号线拉低开始，D/CX 信号线也置低电平表示写入的是命令地址(可理解为命令编码，如软件复位命令：0x01)，以 WRX 信号线为低，RDX 信号为高表示数据传输方向为写入，同时，在数据线[17:0]输出命令地址，在第第二个传输阶段传送的为命令的参数，所以 D/CX 要置高电平，表示写入的是命令数据。

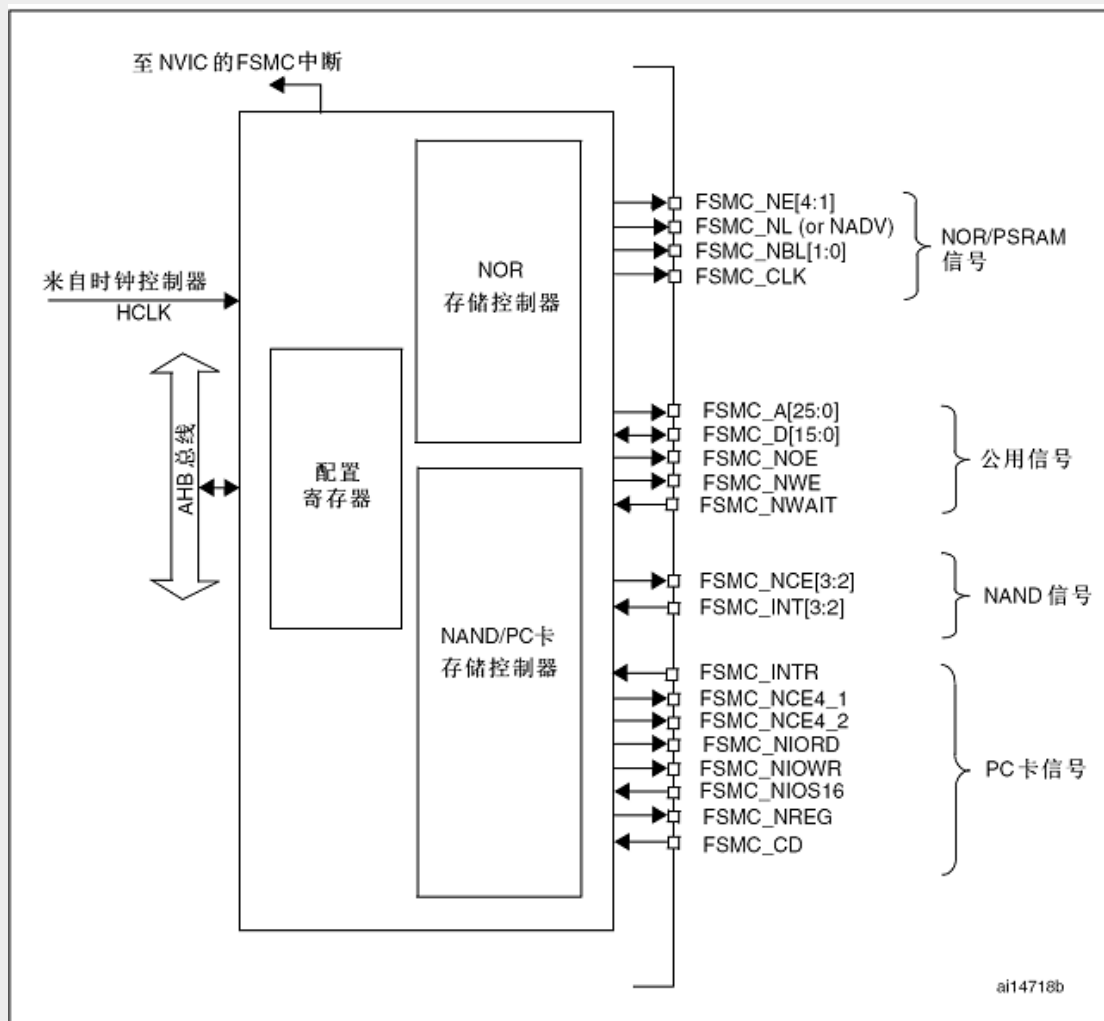
当我们需要向 GRAM 写入数据的时候，把 CSX 信号线拉低后，把 D/CX 信号线置为高电平，这时由 D[17:0]传输的数据则会被 ILI9341 保存至它的 GRAM 中。

## 11.2. 用 STM32 驱动 LCD

ILI9341 的 8080 通讯接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率较低，它提供了一种特别的控制方法——使用 FSMC 接口。

## 11.2.1. FSMC 简介

FSMC(flexible static memory controller), 译为静态存储控制器。可用于 STM32 芯片控制 NOR FLASH、PSRAM、和 NAND FLASH 存储芯片。其结构见下图。



FSMC 结构图

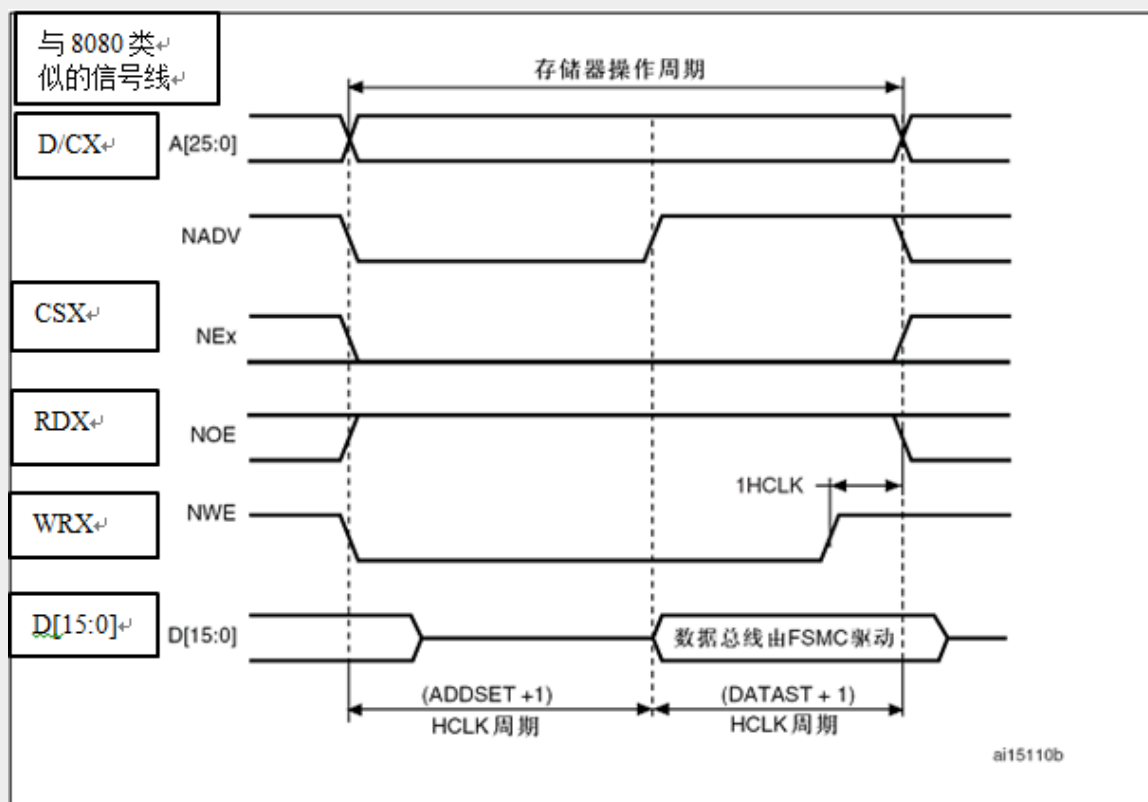
我们是使用 FSMC 的 NOR\PSRAM 模式控制 LCD，所以我们重点分析框图中 NOR FLASH 控制信号线部分。控制 NOR FLASH 主要使用到下表的信号线：

FSMC 控制 NOR FLASH 的信号线

| FSMC 信  | 信号   | 功能            |
|---------|------|---------------|
| CLK     | 输出   | 时钟(同步突发模式使用)  |
| A[25:0] | 输出   | 地址总线          |
| D[15:0] | 输入/输 | 双向数据总线        |
| NE[x]   | 输出   | 片选, x = 1...4 |
| NOE     | 输出   | 输出使能          |

|       |    |                     |
|-------|----|---------------------|
| NWE   | 输出 | 写使能                 |
| NWAIT | 输入 | NOR 闪存要求 FSMC 等待的信号 |

根据 STM32 对寻址空间的地址映射，见前面章节的图 5-7，地址 0x6000 0000 ~0x9FFF FFFF 是映射到外部存储器的，而其中的 0x6000 0000 ~0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。当 FSMC 外设被配置为正常工作，并且外部接了 NOR FLASH，这时若向 0x60000000 地址写入数据 0xffff，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。该过程的时序图见下图。



FSMC 写 NOR 时序图

它会控制片选信号  $NE[X]$  选择相应的某块 NOR 芯片，然后使用地址线  $A[25:0]$  输出 0X6C00 0000，在  $NWE$  写使能信号线上发出写使能信号，而要写入的数据信号 0xffff 则从数据线  $D[15:0]$  输出，然后数据就被保存到 NOR FLASH 中了。

### 11.2.2. 用 FSMC 模拟 8080 时序

在 FSMC 写 NORD 的时序图中 NADV 信号是在地址、信号线复用作为锁存信号的，在此，我们忽略它。然后读者会发现，这个 FSMC 写 NOR 时序是跟 8080 接口的时序是十分相似的，对它们的信号线对比见下表。

FSMC 的 NOR 与 8080 信号线对比

| 8080 信号线 | 功能      | FSMC-NOR<br>信号线 | 功能   |
|----------|---------|-----------------|------|
| CSX      | 片选信号    | NE <sub>x</sub> | 片选   |
| WRX      | 写使能     | NWR             | 写使能  |
| RDX      | 读使能     | NOE             | 读使能  |
| D[15:0]  | 数据信号    | D[15:0]         | 数据信号 |
| D/CX     | 数据/命令选择 | A[25:0]         | 地址信号 |

前四种信号线都是完全一样的，仅在 8080 的数据\命令选择线与 FSMC 的地址信号线有区别。为了模拟出 8080 时序，我们把 FSMC 的 **A23 地址线**(也可以使用其它地址线)连接 8080 的 **D/CX**，即 A23 为**高电平**时，数据线 D[15:0]的信号会被理解 ILI9341 为**数值**，若 A23 为**低电平**时，传输的信号则会被理解为**命令**。

也就是说，当向地址为 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5…这些**奇数地址**写入数据时，地址线 A0(D/CX)会为**高电平**，这个数据被理解为**数值**；若向 0x6xxx xxx0、0x6xxx xxx2、0x6xxx xxx4…这些**偶数地址**写入数据时，地址线 A0(D/CX)会为**低电平**，这个数据会被理解为**命令**。

有了这个基础，只要我们在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

### 11.3. 实验代码分析

```
1.      /**
2.      * @brief 主函数
```



```
3.      * @param  无
4.      * @retval 无
5.      */
6.      int main(void)
7.      {
8.          LCD_Init();
9.
10.         LCD_Clear(0, 0, 240, 320, RED);
11.
12.         LCD_DisStr(10, 10, (uint8_t *) "This is a lcd test demo", B
LACK);
13.
14.         LCD_DisStr(40, 100, (uint8_t *) "count:", BLACK);
15.
16.         for( n=0; n<500000; n++ )
17.         {
18.             LCD_DisNum(100, 100, n, BLACK);
19.             Delay(0xAFFf>>4);
20.         }
```

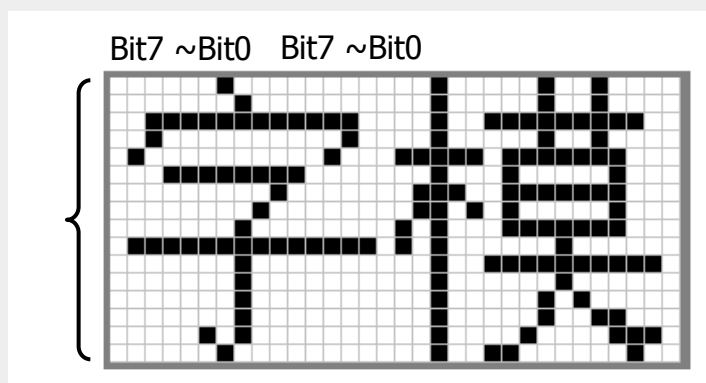
### 11.3.1. 液晶清屏显示

### 11.3.2. 液晶显示英文

#### ■ 什么是字模

我们知道其实液晶屏就是一个由像素点组成的点阵，若要显示文字，则需要很多像素点的共同构成。见下图 21-1，图中是两个由 16\*16 的点阵显示的两个汉字。

每个字  
16x2Byte



字模

如果我们规定：每个汉字都由这样  $16 \times 16$  的点阵来显示，把笔迹经过的像素点以“1”表示，没有笔迹的点以“0”表示，每个像素点的状态以一个二进制位来记录，用  $16 \times 16 / 8 = 32$  个字节就可以把这个字记录下来。这 32 个字节数据就称为该文字的字模，还有其它常用字模是  $24 \times 24$ 、 $32 \times 32$  的。 $16 \times 16$  的“字”的字模数据见代码清单。

代码清单：“字”的字模数据

```
1. /* 字 */
2. unsigned char code Bmp003[]=
3. {
4. /*-----*/
5. ; 源文件 / 文字 : 字
6. ; 宽×高（像素）: 16×16
7. ; 字模格式/大小 : 单色点阵液晶字模，横向取模，字节正序/32 字节
8. -----*/
9.
10. 0x02,0x00,0x01,0x00,0x3F,0xFC,0x20,0x04,0x40,0x08,0x1F,0xE0,0x0
11. 0,0x40,0x00,0x80,
12. 0xFF,0xFF,0x7F,0xFE,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x0
13. 5,0x00,0x02,0x00,
14. };
```

在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模。如果使用 LCD 的画点函数，按位来扫描这些字模数据，把为 1 的位以黑色来显示（也可以使用其它颜色），即可把整个点阵还原出来，显示在液晶屏上。

## ■ 制作 ASCII 码字模

汉字字模比较大，必须放到 SD 卡里面来读写，其中又必须用到 FATFS 文件系统，这部分的知识稍微难一点，所以我们把液晶显示汉字放到项目提高篇里面，这里我们先来学习在液晶上显示英文和数字。

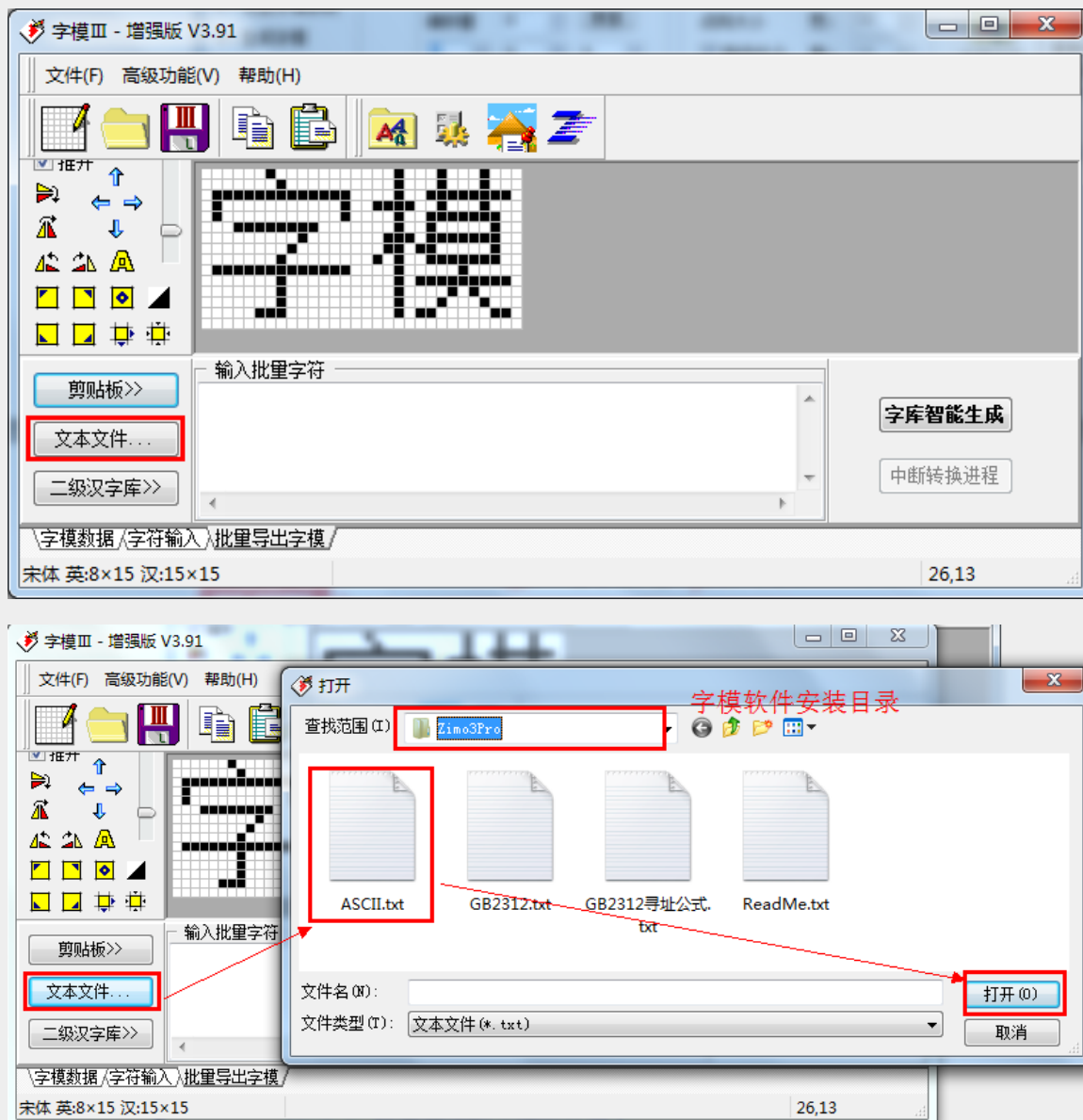
### 1、启动字模制作软件



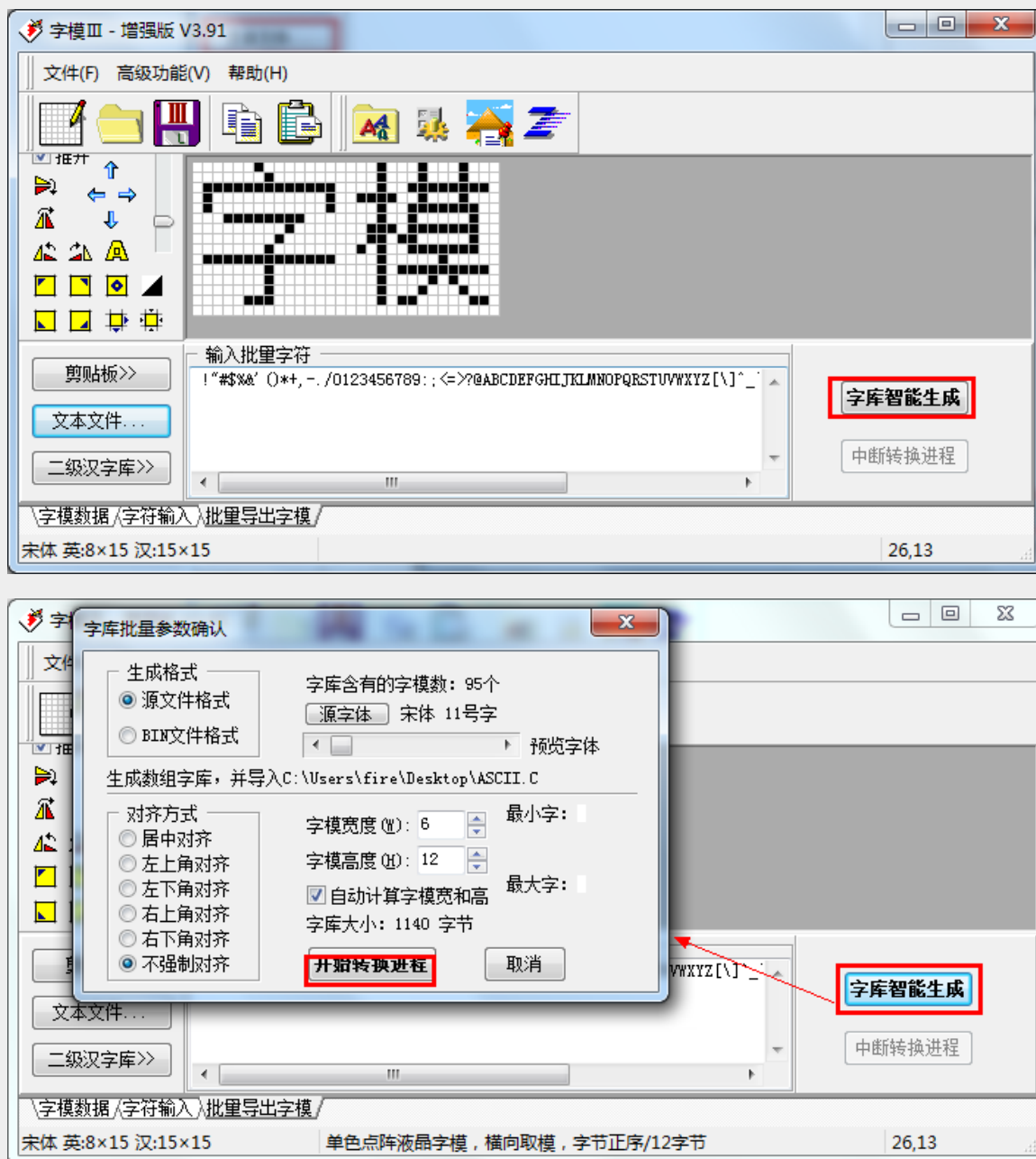
### 2、批量导出字模



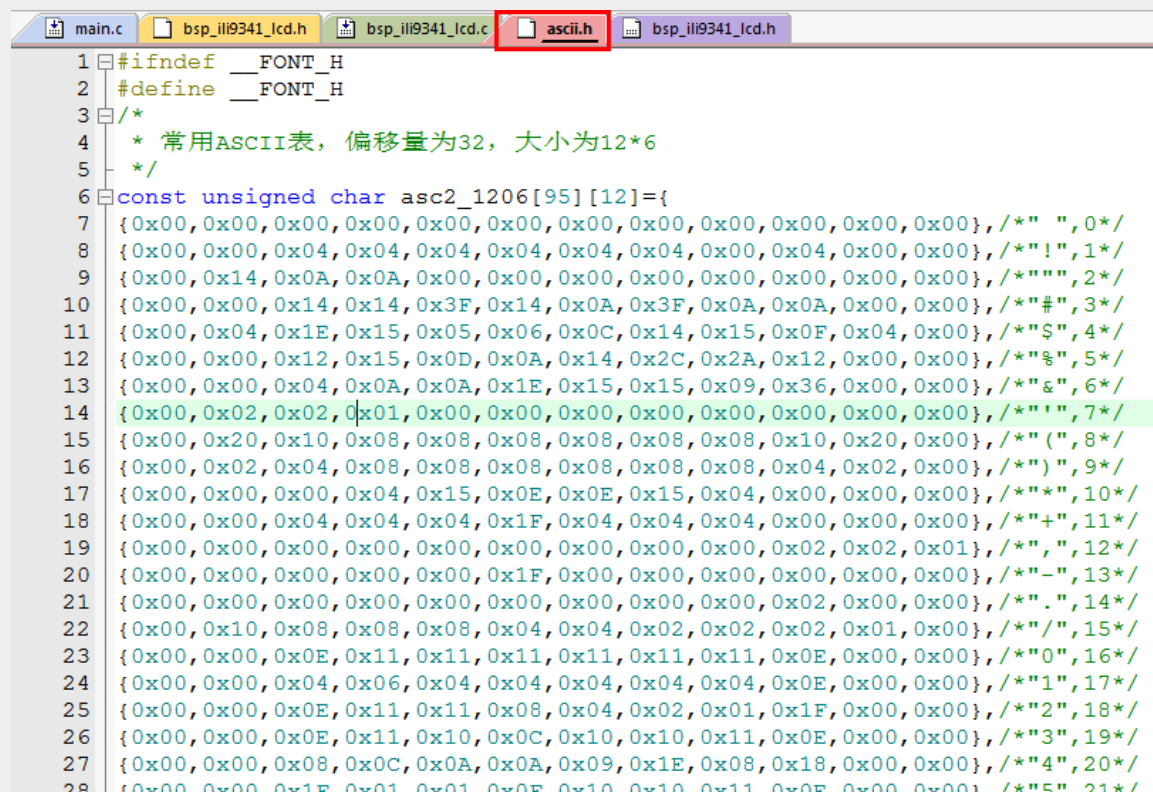
#### 4、选择文本文件



## 5 字库智能生成



我们讲生成的 ASCII 码字库修改成如下格式：



```
1 #ifndef __FONT_H
2 #define __FONT_H
3 /*
4  * 常用ASCII表，偏移量为32，大小为12*6
5  */
6 const unsigned char asc2_1206[95][12]={
7 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*" ",0*/
8 {0x00,0x00,0x04,0x04,0x04,0x04,0x04,0x04,0x00,0x04,0x00,0x00},/*"! ",1*/
9 {0x00,0x14,0x0A,0x0A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*" " ",2*/
10 {0x00,0x00,0x14,0x14,0x3F,0x14,0x0A,0x3F,0x0A,0x0A,0x00,0x00},/*"# ",3*/
11 {0x00,0x04,0x1E,0x15,0x05,0x06,0x0C,0x14,0x15,0x0F,0x04,0x00},/*"$ ",4*/
12 {0x00,0x00,0x12,0x15,0x0D,0x0A,0x14,0x2C,0x2A,0x12,0x00,0x00},/*"% ",5*/
13 {0x00,0x00,0x04,0x0A,0x0A,0x1E,0x15,0x15,0x09,0x36,0x00,0x00},/*"& ",6*/
14 {0x00,0x02,0x02,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*"' ",7*/
15 {0x00,0x20,0x10,0x08,0x08,0x08,0x08,0x08,0x08,0x10,0x20,0x00},/*"( ",8*/
16 {0x00,0x02,0x04,0x08,0x08,0x08,0x08,0x08,0x08,0x04,0x02,0x00},/*") ",9*/
17 {0x00,0x00,0x00,0x04,0x15,0x0E,0x0E,0x15,0x04,0x00,0x00,0x00},/*"* ",10*/
18 {0x00,0x00,0x04,0x04,0x04,0x1F,0x04,0x04,0x04,0x00,0x00,0x00},/*"+ ",11*/
19 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x02,0x01},/*", ",12*/
20 {0x00,0x00,0x00,0x00,0x00,0x1F,0x00,0x00,0x00,0x00,0x00,0x00},/*"- ",13*/
21 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00},/*". ",14*/
22 {0x00,0x10,0x08,0x08,0x08,0x04,0x04,0x02,0x02,0x02,0x01,0x00},/*"/ ",15*/
23 {0x00,0x00,0x0E,0x11,0x11,0x11,0x11,0x11,0x11,0x0E,0x00,0x00},/*"0 ",16*/
24 {0x00,0x00,0x04,0x06,0x04,0x04,0x04,0x04,0x04,0x0E,0x00,0x00},/*"1 ",17*/
25 {0x00,0x00,0x0E,0x11,0x11,0x08,0x04,0x02,0x01,0x1F,0x00,0x00},/*"2 ",18*/
26 {0x00,0x00,0x0E,0x11,0x10,0x0C,0x10,0x10,0x11,0x0E,0x00,0x00},/*"3 ",19*/
27 {0x00,0x00,0x08,0x0C,0x0A,0x0A,0x09,0x1E,0x08,0x18,0x00,0x00},/*"4 ",20*/
28 {0x00,0x00,0x1F,0x01,0x01,0x0F,0x10,0x10,0x11,0x0F,0x00,0x00},/*"5 ",21*/
```

我们这里制作的 ASCII 字模的宽度为 6 个像素，高度为 12 个像素。ASCII 码空格 ' ' 前面还有 32 个液晶显示不了的字符，故偏移量为 32。

### 11.3.3. 液晶显示变量

## 12. I2C-EEPROM

### 12.1. I2C 协议简介

I<sup>2</sup>C (Inter-Integrated Circuit)协议是由 Philips 公司开发的，由于它引脚少，硬件实现简单，可扩展性强，不需要如 USART、CAN 的外部收发设备，现在被广泛地使用在系统内多个集成电路(IC)间的通讯。

根据《I<sup>2</sup>C 总线协议版本 2.1-2000》的说明，我们可以更详细地了解 I<sup>2</sup>C 协议。

#### 12.1.1. 物理层

(1) 它只使用两条总线线路，一条双向串行数据线(SDA)，一条串行时钟线 (SCL)。

(2) 每个连接到总线的设备都有一个独立的地址，主机可以利用这个地址进行不同设备之间的访问

(3) 多主机同时使用总线时，为了防止数据冲突，会利用仲裁方式决定由哪个设备占用总线。

(4) 具有三种传输模式：标准模式传输速率为 100kbit/s，快速模式为 400kbit/s，高速模式下可达 3.4Mbit/s，但目前大多 I<sup>2</sup>C 设备尚不支持高速模式。

(5) 片上的滤波器可以滤去总线数据线上的毛刺波 保证数据完整。

(6) 连接到相同总线的 IC 数量受到总线的最大电容 400pF 限制。

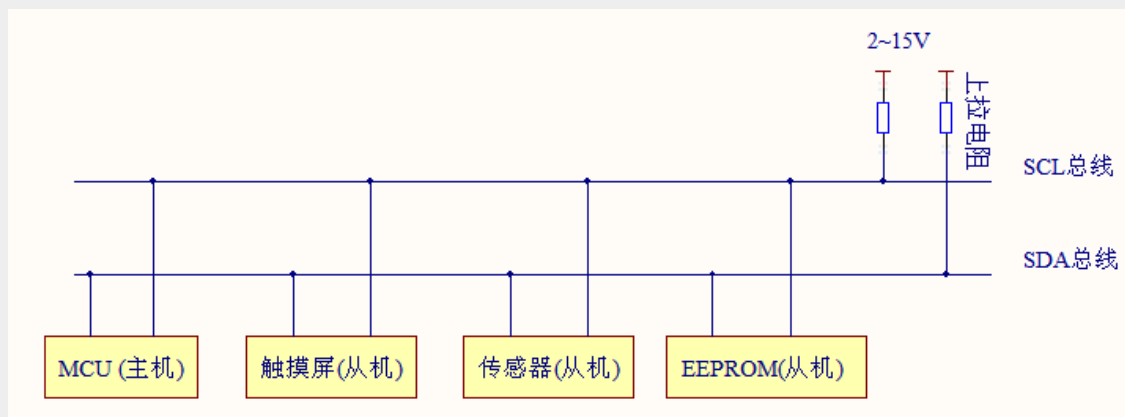


图 12-1 常见的 I2C 通讯系统

### 12.1.2. 协议层

I<sup>2</sup>C 的协议包括起始和停止条件、数据有效性、响应、仲裁、时钟同步和地址广播等环节，由于我们使用的是 STM32 集成的硬件 I<sup>2</sup>C 接口，并不需要用软件去模拟 SDA 和 SCL 线的时序，所以野火直接以 I<sup>2</sup>C 通讯的流程为大家讲解。见图 12-2 主机写数据到从机。

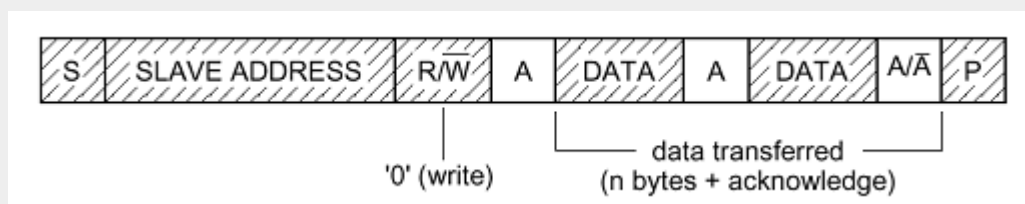


图 12-2 主机写数据到从机

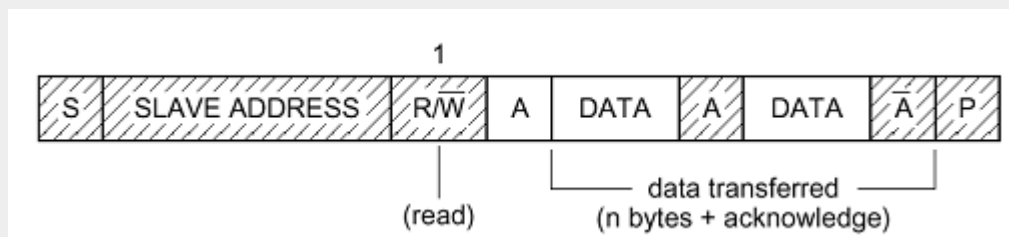


图 12-3 主机由从机中读数据

图例：▨ 数据由主机传输至从机 S：传输开始信号

SLAVE\_ADDRESS: 从机地址

□ 数据由从机传输至主机 R/W: 传输方向选择位，1 为读，0

为写

A/A：应答或非应答信号

P：停止传输信号

这两幅图表示的是主机和从机通讯时，SDA 线的数据包序列。

其中 S 表示由主机的 I<sup>2</sup>C 接口产生的传输起始信号(S)，这时连接到 I<sup>2</sup>C 总线上的所有从机都会接收到这个信号。

起始信号产生后，所有从机就开始等待主机紧接下来广播 的从机地址信号 (SLAVE\_ADDRESS)，在 I<sup>2</sup>C 总线上，每个设备的地址都是唯一的。当主机广播



的地址与某个设备地址相同时，这个设备就被选中了，没被选中的设备将会忽略之后的数据信号。根据 I<sup>2</sup>C 协议，这个从机地址可以是 7 位或 10 位。

在地址位之后，是传输方向的选择位，该位为 0 时，表示后面的数据传输方向是由主机传输至从机。该位为 1 时，则相反。

从机接收到匹配的地址后，主机或从机会返回一个应答(A)或非应答( $\overline{A}$ )信号，只有接收到应答信号后，主机才能继续发送或接收数据。

若配置的方向传输位为写数据，广播完地址，接收到应答信号后，主机开始正式向从机传输数据(DATA)，数据包的大小为 8 位，主机每发送完一个数据，都要等待从机的应答信号(A)，重复这个过程，可以向从机传输 N 个数据，这个 N 没有大小限制。当数据传输结束时，主机向从机发送一个停止传输信号(P)，表示不再传输数据。

若配置的方向传输位为读数据，广播完地址，接收到应答信号后，从机开始向主机返回数据(DATA)，数据包大小也为 8 位，从机每发送完一个数据，都会等待主机的应答信号(A)，重复这个过程，可以返回 N 个数据，这个 N 也没有大小限制。当主机希望停止接收数据时，就向从机返回一个非应答信号( $\overline{A}$ )，则从机自动停止数据传输。

## 12.2. STM32 的 I2C 特性及架构

### 12.2.1. I2C 接口特性

- (1) STM32 的中等容量和大容量型号的芯片均有多达 2 个的 I<sup>2</sup>C 总线接口。
- (2) 能够工作于多主模式或从模式，分别为主接收器，主发送器，从接收器及从发。
- (3) 支持标准模式 100Kbit/s 和快速模式 400Kbit/s，不支持高速模式。
- (4) 支持 7 位或 10 位寻址。
- (5) 内置了硬件 CRC 发生器/ 校验器。

- (6) I<sup>2</sup>C 的接收和发送都可以使用 DMA 操作。
- (7) 支持系统管理总线(SMBus)总线 2.0 版。

### 12.2.2. I2C 架构

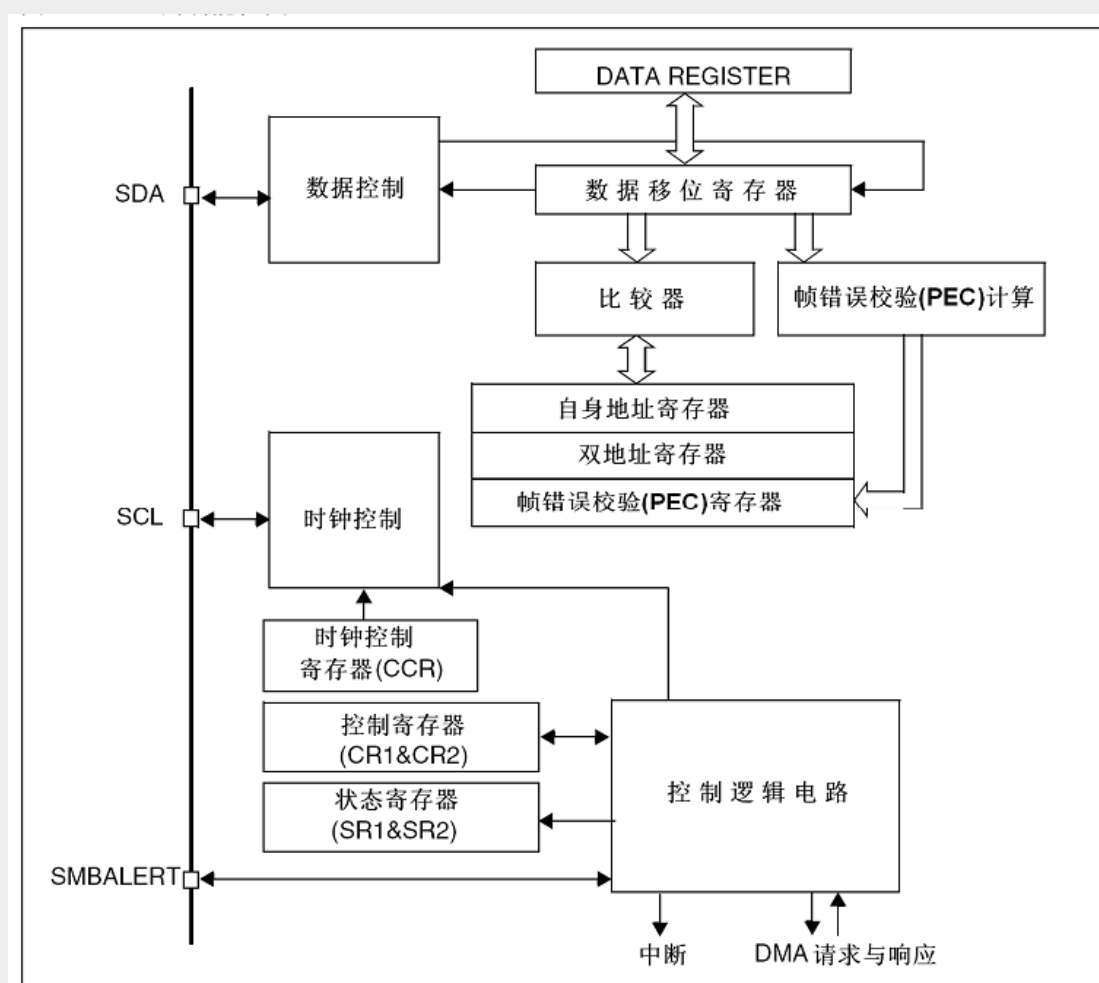


图 12-4 I2C 架构图

我们可以看到，I<sup>2</sup>C 的所有硬件架构就是根据 SCL 线和 SDA 线展开的(其中的 SMBALERT 线用于 SMBUS)。

SCL 线的时序即为 I<sup>2</sup>C 协议中的时钟信号，它由 I<sup>2</sup>C 接口根据时钟控制寄存器(CCR)控制，控制的参数主要为时钟频率。

而 SDA 的信号则通过一系列数据控制架构，在将要发送的数据的基础上，根据协议添加各种起始信号、应答信号、地址信号，实现以 I<sup>2</sup>C 协议的方式发

送出去。读取数据时则从 SDA 线上的信号中取出接收到的数据值。发送和接收的数据都是被保存在数据寄存器(DR)上的。

### 12.3. I2C 接口读写 EEPROM 实验

对 EEPROM 读写最常用的方式是使用 I<sup>2</sup>C 协议。本章以 EEPROM 的读写实验为大家讲解 STM32 的 I<sup>2</sup>C 使用方法。

野火 STM32 开发板用的是 STM32F103VET6。它有 2 个 I<sup>2</sup>C 接口。本实验使用 I<sup>2</sup>C1，对应地连接到 EERPOM（型号：AT24C02）的 SCL 和 SDA 线。实现 I<sup>2</sup>C 通讯，对 EERPOM 进行读写。

本实验采用主模式，分别用作主发送器和主接收器。通过查询事件的方式来确保正常通讯。

#### 12.3.1. 实验描述及工程文件清单

##### 实验描述：

向 EERPOM 写入数据，再读取出来，进行校验，通过串口打印写入与读取出来的数据，并输出校验结果。

##### 硬件连接：

PB6-I2C1\_SCL,

PB7-I2C1\_SDA

##### 库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_i2c.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

USER/i2c\_ee.c

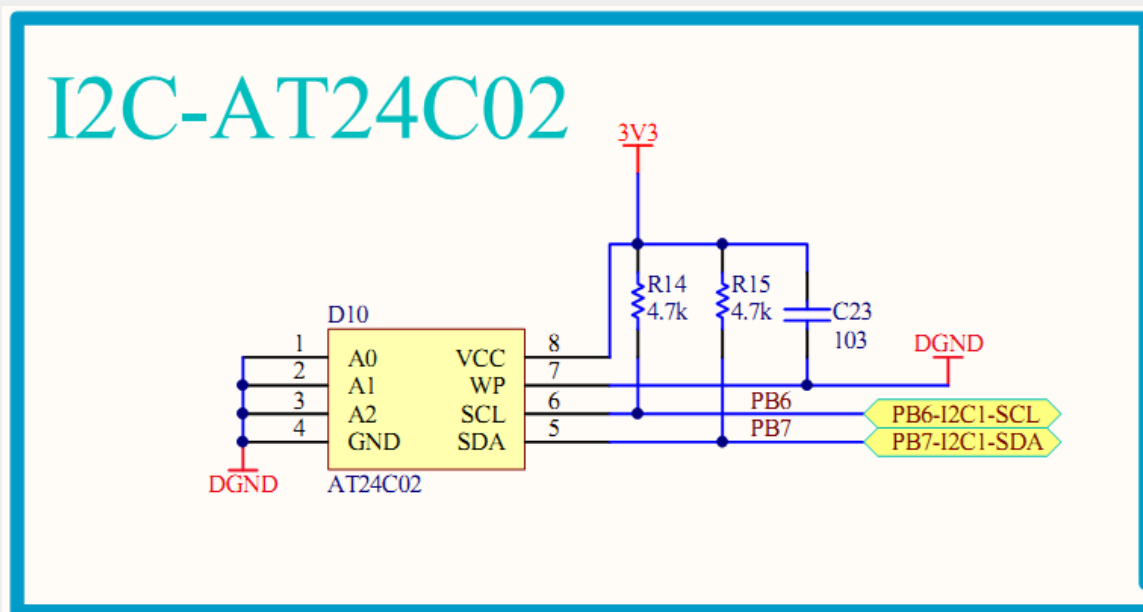


图 12-5 野火 STM32 开发板 I<sup>2</sup>C-EEPROM 硬件原理图

### 12.3.2. 配置工程环境

本 I<sup>2</sup>C-EEPROM 实验中我们用到了 GPIO、RCC、USART 及 I<sup>2</sup>C 外设，所以我们先要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_usart.c`、`stm32f10x_i2c.c`。本实验中没有使用中断，是使用轮询的方式来完成 I<sup>2</sup>C 的收发数据的。

接下来添加旧工程中的外设用户文件 `usart.c`，以便调试和观察实验效果。新建 `i2c_ee.c` 及 `i2c_ee.h` 文件。并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。见代码清单 12-1。

代码清单 12-1: I<sup>2</sup>C 接口实例的 `stm32f10x_conf.h` 文件配置

```

1.  /**
2.  * *****
3.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author    MCD Application Team
5.  * @version   V3.5.0
6.  * @date      08-April-2011
7.  * @brief     Library configuration file.
8.  * *****/
9.  *
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_i2c.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_usart.h"

```

### 12.3.3. main 文件

配置好所需的库文件之后，我们就从 main 函数开始分析，见。

代码清单 12-2: I<sup>2</sup>C 接口实例的 main 函数

```

1.  /**
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  */
8.  int main(void)
9.  {
10.     /* 配置系统时钟为 72M */
11.     //SystemInit();
12.
13.     /* 串口1 初始化 */
14.     USART1_Config();
15.
16.     /* I2C 外设初(AT24C02)始化 */
17.     I2C_EE_Init();
18.
19.     USART1_printf(USART1, "\r\n 这是一个 I2C 外设(AT24C02)读写测试例
程 \r\n");
20.     USART1_printf(USART1, "\r\n (\"__DATE__ \" -
\" __TIME__ \" ) \r\n");
21.
22.     I2C_Test();
23.
24.     while (1)
25.     {
26.     }
27. }

```

代码中的系统库函数 **SystemInit()**被注释掉了，这是野火在升级代码时修改的，3.5 版本的代码不需要在 main 函数中再调用 **SystemInit()**函数初始化时

钟。接下来调用用户函数 `USART1_Config()`和 `I2C_EE_Init()` 配置串口及 I<sup>2</sup>C 接口初始化。

初始化完成后由串口向终端输出调试信息，接着调用用户函数 `I2C_Test()`，它利用配置好的 I<sup>2</sup>C 接口向 EEPROM 写入数据，再把这些数据读取出来进行校验，检查我们的 I<sup>2</sup>C 是否正常工作。

#### 12.3.4. I2C 接口初始化

我们先来分析 I<sup>2</sup>C 接口的初始化函数 `I2C_EE_Init()`。它是用户编写的函数，与其它对 GPIO 复用的外设一样，它先调用了用户函数 `I2C_GPIO_Config()` 配置好 I<sup>2</sup>C 所用的 I/O 端口，然后再调用用户函数 `I2C_Mode_Configu()` 设置 I<sup>2</sup>C 的工作模式。并使能相关外设的时钟。见代码清单 12-3。

代码清单 12-3: `I2C_EE_Init()`函数

```
1.  /*
2.  * 函数名: I2C_EE_Init
3.  * 描述   : I2C 外设 (EEPROM) 初始化
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void I2C_EE_Init(void)
9. {
10.
11.     I2C_GPIO_Config();
12.
13.     I2C_Mode_Configu();
14.
15.     /* 根据头文件 i2c_ee.h 中的定义来选择 EEPROM 要写入的地址 */
16.     #ifdef EEPROM_Block0_ADDRESS
17.         /* 选择 EEPROM Block0 来写入 */
18.         EEPROM_ADDRESS = EEPROM_Block0_ADDRESS;
19.     #endif
20.
21.     #ifdef EEPROM_Block1_ADDRESS
22.         /* 选择 EEPROM Block1 来写入 */
23.         EEPROM_ADDRESS = EEPROM_Block1_ADDRESS;
24.     #endif
25.
26.     #ifdef EEPROM_Block2_ADDRESS
27.         /* 选择 EEPROM Block2 来写入 */
28.         EEPROM_ADDRESS = EEPROM_Block2_ADDRESS;
29.     #endif
30.
31.     #ifdef EEPROM_Block3_ADDRESS
32.         /* 选择 EEPROM Block3 来写入 */
33.         EEPROM_ADDRESS = EEPROM_Block3_ADDRESS;
34.     #endif
```

35. }

## ■ EEPROM 的地址

我们知道，每一个连接到 I<sup>2</sup>C 总线上的设备都需要有唯一的地址，本实验中的 EEPROM 地址是如何决定的呢？阅读 EEPROM 的 datasheet，我们可以看到以下说明：

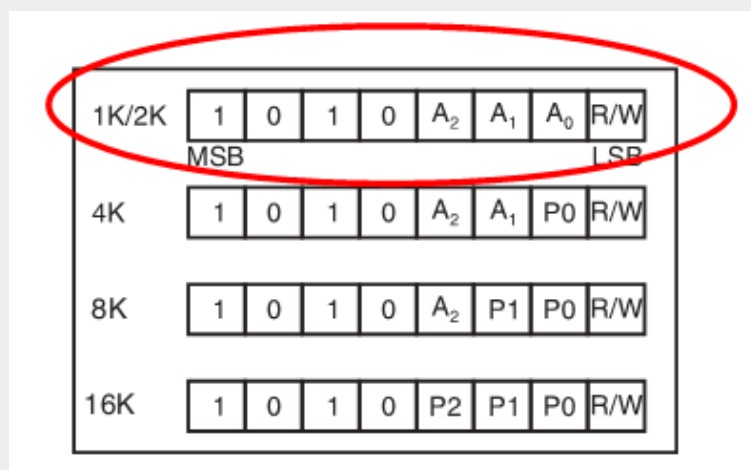


图 12-6 EEPROM 地址说明

本实验中使用的 EEPROM 型号为 AT24C02，容量为 2K bits，说明中已经对 EEPROM 地址的高四位作了硬性规定，而最低位又为 R/W(传输方向选择位)。在制作硬件时，我们可以根据需要改变的是地址位中的 A<sub>2</sub>、A<sub>1</sub>、A<sub>0</sub> 位，查看本实验中的硬件接法图可知 EEPROM 的 A<sub>2</sub>、A<sub>1</sub>、A<sub>0</sub> 位均接地，所以它的地址为：0xA0 或 0xA1，为什么有两个呢？因为最低位为数据方向选择位，在我们使用这个地址变量时，调用的库函数还会对数据方向位重新处理，所以最低位的数据并不影响我们实际的使用，但这也不代表 EEPROM 的实际地址有两个。如果不是使用库函数，那就得小心使用了。

在 I2C\_EE\_Init()函数中，用条件编译确定了 EEPROM 的器件地址，这个宏在 i2c\_ee.h 文件中进行了定义，见代码清单 12-4。若大家对 EEPROM 的编址不一样，就要使用其它宏。

代码清单 12-4：EEPROM 器件地址宏

```
1. /* EEPROM Addresses defines */
2. #define EEPROM_Block0_ADDRESS 0xA0 /* E2 = 0 */
3. // #define EEPROM_Block1_ADDRESS 0xA2 /* E2 = 0 */
```

```
4. //define EEPROM_Block2_ADDRESS 0xA4 /* E2 = 0 */
5. //define EEPROM_Block3_ADDRESS 0xA6 /* E2 = 0 */
```

条件编译中的变量值 `EEPROM_ADDRESS` 在正式使用时我们再来分析。

## ■ GPIO 端口初始化

现在先来看看 GPIO 的初始化，它在 `I2C_GPIO_Config()` 函数中实现了，见代码清单 12-5。

代码清单 12-5: `I2C_GPIO_Config()` 函数

```
1. /*
2.  * 函数名: I2C_GPIO_Config
3.  * 描述   : I2C1 I/O 配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void I2C_GPIO_Config(void)
9. {
10.    GPIO_InitTypeDef GPIO_InitStructure;
11.
12.    /* 使能与 I2C1 有关的时钟 */
13.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
14.    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
15.
16.    /* PB6-I2C1_SCL、PB7-I2C1_SDA */
17.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
18.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
19.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;    // 开漏输
    出
20.    GPIO_Init(GPIOB, &GPIO_InitStructure);
21. }
```

在本函数中，我们开启了 GPIO 时钟，I<sup>2</sup>C1 的时钟，并且对 I<sup>2</sup>C1 复用的两个引脚进行初始化配置，模式设置为开漏输出。若不清楚它 I<sup>2</sup>C1 的复用引脚和复用模式，可分别查询《STM32 数据手册》的引脚定义部分和《STM32 参考手册》的 GPIO 章节。

表 12-1 I<sup>2</sup>C1 引脚定义

| 脚位 | 脚名 | I/O | 功能 | 可选的复用功能 |
|----|----|-----|----|---------|
|----|----|-----|----|---------|



|   |   |   |   |   |    | 称       |    | 电<br>平 |         | 默认复<br>用功能                         | 重映<br>射功能                  |
|---|---|---|---|---|----|---------|----|--------|---------|------------------------------------|----------------------------|
| 6 | 5 | 5 | 7 | 1 | 35 | P<br>B5 | /O |        | P<br>B5 | I2C1_SMBA/<br>SPI3_MOSI<br>I2S3_SD | TIM3_<br>CH2/<br>SPI1_MOSI |
| 6 | 5 | 5 | 8 | 2 | 36 | P<br>B6 | /O | T      | P<br>B6 | I2C1_SCL/TI<br>M4_CH1              | USART<br>1_TX              |
| 6 | 5 | 5 | 9 | 3 | 37 | P<br>B7 | /O | T      | P<br>B7 | I2C1_SDA<br>/FSMC_NADV<br>TIM4_CH2 | USART<br>1_RX              |

表 12-2 I<sup>2</sup>C 复用 GPIO 模式设置

| I2C 引脚   | 配置     | GPIO 配置 |
|----------|--------|---------|
| I2Cx_SCL | I2C 时钟 | 开漏复用输出  |
| I2Cx_SDA | I2C 数据 | 开漏复用输出  |

## ■ I2C 模式初始化

接下来我们再来分析一下在 I2C\_Mode\_Configu()函数中进行的 I<sup>2</sup>C 的模式初始化，见代码清单 12-6。

代码清单 12-6: I2C\_Mode\_Configu()函数

```

1.  /*
2.   * 函数名: I2C_Configuration
3.   * 描述  : I2C 工作模式配置
4.   * 输入  : 无
5.   * 输出  : 无

```

```
6.  * 调用   : 内部调用
7.  */
8.  static void I2C_Mode_Configu(void)
9.  {
10.   I2C_InitTypeDef  I2C_InitStructure;
11.
12.   /* I2C 配置 */
13.   I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
14.   I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
15.   I2C_InitStructure.I2C_OwnAddress1 = I2C1_OWN_ADDRESS7;
16.   I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
17.   I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedA
ddress_7bit;
18.   I2C_InitStructure.I2C_ClockSpeed = I2C_Speed;
19.
20.   /* I2C1 初始化 */
21.   I2C_Init(I2C1, &I2C_InitStructure);
22.   /* 使能 I2C1 */
23.   I2C_Cmd(I2C1, ENABLE);
24. }
```

与其它外设初始化一样，I<sup>2</sup>C 模式的初始化还是对它的初始化结构体进行参数配置。我们来分析它这些结构体成员：

### (1) I2C\_Mode

本成员是选择 I<sup>2</sup>C 的使用方式，有 I<sup>2</sup>C 模式(I2C\_Mode\_I2C )和 SMBus 模式(I2C\_Mode\_SMBusDevice、I2C\_Mode\_SMBusHost )。

本实验中使用的是 I<sup>2</sup>C 模式。

### (2) I2C\_DutyCycle

本成员设置的是 I<sup>2</sup>C 的 SCL 线时钟的占空比。SCL 线的时钟信号的高电平时间与低电平时间是没必要相同的，由于 SDA 线是在 SCL 线维持在高电平时读取或写入数据的，而在 SCL 的低电平期间 SDA 的数据发生变化，所以高电平时间较长就不容易出现数据错误。根据 I<sup>2</sup>C 协议，在快速模式和高速模式下 SCL 的高低电平时间可以不同。在 STM32 的 I<sup>2</sup>C 占空比配置中有两个选择，分别为高电平时间比低电平时间为 16：9 (I2C\_DutyCycle\_16\_9) 和 2：1 (I2C\_DutyCycle\_2)。

本实验中使用的是 2：1。

### (3) I2C\_OwnAddress1

本成员配置的是 STM32 的 I2C 设备自己的地址，每个连接到 I2C 总线上的设备都要有一个自己的地址，作为主机也不例外。就如同前面介绍的连接到

I2C 的 EEPROM 的设备地址，只不过 STM32 的 I2C 地址是可软件编程的。这个地址可以被配置为 7 位和 10 位地址。

本实验中把这个地址设置为 0x0A (自定义宏 I2C1\_OWN\_ADDRESS7)。

#### (4) I2C\_Ack\_Enable

本成员是关于 I<sup>2</sup>C 应答设置，设置为使能则每接收到一个字节就返回一个应答信号。

本实验配置为允许应答(I2C\_Ack\_Enable)，这是绝大多数遵循 I<sup>2</sup>C 标准的设备通讯的要求，改为禁止应答(I2C\_Ack\_Disable)往往会导致通讯错误。

#### (5) I2C\_AcknowledgeAddress

本成员选择 I2C 的寻址模式是 7 位还是 10 位地址。这需要根据实际连接到 I2C 总线上设备的地址进行选择。

本实验是与 EEPROM 进行通讯，使用的为 7 位寻址模式(I2C\_AcknowledgedAddress\_7bit)。

#### (6) I2C\_ClockSpeed

本成员设置的是 I2C 的传输速率，在调用初始化函数时，函数会根据我们输入的数值经过运算后把分频值写入到 I2C 的时钟控制寄存器。而我们写入的这个参数值不得高于 400KHz。

本实验向这个成员写入参数为 400000(自定义宏 I2C\_Speed)。实际上由于 I2C 使用的 APB1 时钟为 36MHz，不是 10MHz 的整数倍，因此最终分频后输出的 SCL 线时钟并不是精确的 400KHz。

对结构体成员赋值完成后，我们调用库函数 I2C\_Init()根据我们的配置对 I2C 进行初始化，并调用库函数 I2C\_Cmd()使能 I2C 外设。

### 12.3.5. 对 EEPROM 的读写操作

对 I<sup>2</sup>C 接口初始化完成后,我们就可以利用 I<sup>2</sup>C 接口,根据 EEPROM 设备的要求对它进行读写操作。在本实验中的 I2C\_Test()函数就是一个对 EEPROM 进行读写的例子,见代码清单 12-7。

代码清单 12-7: I2C\_Test()函数

```
1.  /*
2.  * 函数名: I2C_EE_Test
3.  * 描述   : I2C(AT24C02)读写测试。
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  */
8. void I2C_Test(void)
9. {
10.     u16 i;
11.
12.     printf("写入的数据\n\r");
13.
14.     for ( i=0; i<=255; i++ ) //填充缓冲
15.     {
16.         I2c_Buf_Write[i] = i;
17.
18.         printf("0x%02X ", I2c_Buf_Write[i]);
19.         if(i%16 == 15)
20.             printf("\n\r");
21.     }
22.
23.     //将 I2c_Buf_Write 中顺序递增的数据写入 EEPROM 中
24.     I2C_EE_BufferWrite( I2c_Buf_Write, EEP_Firstpage, 256);
25.
26.     printf("\n\r 读出的数据\n\r");
27.     //将 EEPROM 读出数据顺序保持到 I2c_Buf_Read 中
28.     I2C_EE_BufferRead(I2c_Buf_Read, EEP_Firstpage, 256);
29.
30.     //将 I2c_Buf_Read 中的数据通过串口打印
31.     for (i=0; i<256; i++)
32.     {
33.         if(I2c_Buf_Read[i] != I2c_Buf_Write[i])
34.         {
35.             printf("0x%02X ", I2c_Buf_Read[i]);
36.             printf("错误:I2C EEPROM 写入与读出的数据不一致\n\r");
37.             return;
38.         }
39.         printf("0x%02X ", I2c_Buf_Read[i]);
40.         if(i%16 == 15)
41.             printf("\n\r");
42.     }
43.
44.     printf("I2C(AT24C02)读写测试成功\n\r");
45. }
```

这个函数实现的功能是把数值 0~255 按顺序填入缓冲区数组，并通过串口打印到终端，接着通过用户函数 I2C\_EE\_BufferWrite()把缓冲区的数据写入 EEPROM。写入成功之后，利用用户函数 I2C\_EE\_BufferRead()把数据读取出来，进行校验，判断是数据是否被正确写入。

其中的 I2C\_EE\_BufferWrite() 和 I2C\_EE\_BufferRead() 函数就是读写 EEPROM 的重点，编写这个函数主要是根据所使用的 EEPROM 说明。野火通过介绍这个 I2C\_EE\_BufferWrite()函数让大家熟悉 STM32 的 I<sup>2</sup>C 通讯流程，而类似的用户函数 I2C\_EE\_BufferRead()则留给读者进行分析。

I2C\_EE\_BufferWrite() 具有三个输入参数，pBuffer 缓冲区指针，WriteAddr 写入数据到 EEPROM 的存储地址，NumByteToWrite 要写入 EEPROM 的字节数。我们在调用时的形式如下 I2C\_EE\_BufferWrite( I2c\_Buf\_Write, EEP\_Firstpage, 256)，这两个参数定义见代码清单 12-8。I2C\_EE\_BufferWrite()函数见代码清单 12-9。

代码清单 12-8: I2C\_EE\_BufferWrite 的输入参数

```
1. u8 I2c_Buf_Write[256];  
2. #define EEP_Firstpage 0x00
```

代码清单 12-9: I2C\_EE\_BufferWrite()函数

```
1. /*  
2. * 函数名: I2C_EE_BufferWrite  
3. * 描述 : 将缓冲区中的数据写到 I2C EEPROM 中  
4. * 输入 : -pBuffer 缓冲区指针  
5. *         -WriteAddr 接收数据的 EEPROM 的地址  
6. *         -NumByteToWrite 要写入 EEPROM 的字节数  
7. * 输出 : 无  
8. * 返回 : 无  
9. * 调用 : 外部调用  
10. */  
11. void I2C_EE_BufferWrite(u8* pBuffer, u8 WriteAddr, u16 NumByteToWrite)  
12. {  
13.     u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0;  
14.  
15.     Addr = WriteAddr % I2C_PageSize;  
16.     count = I2C_PageSize - Addr;  
17.     NumOfPage = NumByteToWrite / I2C_PageSize;  
18.     NumOfSingle = NumByteToWrite % I2C_PageSize;  
19.  
20.     /* If WriteAddr is I2C_PageSize aligned */  
21.     if(Addr == 0)  
22.     {
```

```
23.  /* If NumByteToWrite < I2C_PageSize */
24.  if(NumOfPage == 0)
25.  {
26.      I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
27.      I2C_EE_WaitEepromStandbyState();
28.  }
29.  /* If NumByteToWrite > I2C_PageSize */
30.  else
31.  {
32.      while (NumOfPage--)
33.      {
34.          I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
35.          I2C_EE_WaitEepromStandbyState();
36.          WriteAddr += I2C_PageSize;
37.          pBuffer += I2C_PageSize;
38.      }
39.
40.      if(NumOfSingle!=0)
41.      {
42.          I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
43.          I2C_EE_WaitEepromStandbyState();
44.      }
45.  }
46. }
47. /* If WriteAddr is not I2C_PageSize aligned */
48. else
49. {
50.     /* If NumByteToWrite < I2C_PageSize */
51.     if(NumOfPage== 0)
52.     {
53.         I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
54.         I2C_EE_WaitEepromStandbyState();
55.     }
56.     /* If NumByteToWrite > I2C_PageSize */
57.     else
58.     {
59.         NumByteToWrite -= count;
60.         NumOfPage = NumByteToWrite / I2C_PageSize;
61.         NumOfSingle = NumByteToWrite % I2C_PageSize;
62.
63.         if(count != 0)
64.         {
65.             I2C_EE_PageWrite(pBuffer, WriteAddr, count);
66.             I2C_EE_WaitEepromStandbyState();
67.             WriteAddr += count;
68.             pBuffer += count;
69.         }
70.
71.         while (NumOfPage--)
72.         {
73.             I2C_EE_PageWrite(pBuffer, WriteAddr, I2C_PageSize);
74.             I2C_EE_WaitEepromStandbyState();
75.             WriteAddr += I2C_PageSize;
76.             pBuffer += I2C_PageSize;
77.         }
78.         if(NumOfSingle != 0)
79.         {
80.             I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
81.             I2C_EE_WaitEepromStandbyState();
82.         }
83.     }
84. }
85. }
```

首先，我们要了解 EEPROM 的数据组织形式，EEPROM 设备把它的存储矩阵进行了分页处理，见图 12-7。

|                            |                                                                                                                                                                    |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Memory Organization</b> | <b>AT24C01A, 1K SERIAL EEPROM:</b> Internally organized with 16 pages of 8 bytes each, the 1K requires a 7-bit data word address for random word addressing.       |
|                            | <b>AT24C02, 2K SERIAL EEPROM:</b> Internally organized with 32 pages of 8 bytes each, the 2K requires an 8-bit data word address for random word addressing.       |
|                            | <b>AT24C04, 4K SERIAL EEPROM:</b> Internally organized with 32 pages of 16 bytes each, the 4K requires a 9-bit data word address for random word addressing.       |
|                            | <b>AT24C08A, 8K SERIAL EEPROM:</b> Internally organized with 64 pages of 16 bytes each, the 8K requires a 10-bit data word address for random word addressing.     |
|                            | <b>AT24C16A, 16K SERIAL EEPROM:</b> Internally organized with 128 pages of 16 bytes each, the 16K requires an 11-bit data word address for random word addressing. |

图 12-7 EEPROM 分页方式

型号为 AT24C02 的 EEPROM 分为 32 页，每页可存储 8 个字节的数据，若在同一页写入超过 8 字节，则超过的部分会被写在该页的起始地址，这样部分数据会被覆盖。

因此，为了把连续的缓冲区数组按页写入到 EEPROM，就需要对缓冲区进入分页处理。I2C\_EE\_BufferWrite() 函数根据我们输入的缓冲区大小参数 NumByteToWrite，计算出我们需要写入多少页，计算写入位置。

分页处理好之后，调用 I2C\_EE\_PageWrite() 函数，这个函数是与 EEPROM 进行 I<sup>2</sup>C 通讯的最底层函数，它与 STM32 的 I<sup>2</sup>C 库函数使用密切相关。见代码清单 12-10。

代码清单 12-10: I2C\_EE\_PageWrite() 函数

```
1.  /*
2.   * 函数名: I2C_EE_PageWrite
3.   * 描述   : 在 EEPROM 的一个写循环中可以写多个字节，但一次写入的字节数
4.   *           不能超过 EEPROM 页的大小。AT24C02 每页有 8 个字节。
5.   * 输入    : -pBuffer 缓冲区指针
6.   *           -WriteAddr 接收数据的 EEPROM 的地址
7.   *           -NumByteToWrite 要写入 EEPROM 的字节数
8.   * 输出    : 无
9.   * 返回    : 无
10.  * 调用    : 外部调用
11.  */
12. void I2C_EE_PageWrite(u8* pBuffer, u8 WriteAddr, u8 NumByteToWrite)
13. {
14.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by
Najoua 27/08/2008
15.
16.     /* Send START condition */
17.     I2C_GenerateSTART(I2C1, ENABLE);
18. }
```

```

19.  /* Test on EV5 and clear it */
20.  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

21.
22.  /* Send EEPROM address for write */
23.  I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
24.
25.  /* Test on EV6 and clear it */
26.  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
27.
28.  /* Send the EEPROM's internal address to write to */
29.  I2C_SendData(I2C1, WriteAddr);
30.
31.  /* Test on EV8 and clear it */
32.  while(! I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
33.
34.  /* While there is data to be written */
35.  while(NumByteToWrite--)
36.  {
37.      /* Send the current byte */
38.      I2C_SendData(I2C1, *pBuffer);
39.
40.      /* Point to the next byte to be written */
41.      pBuffer++;
42.
43.      /* Test on EV8 and clear it */
44.      while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
45.  }
46.  /* Send STOP condition */
47.  I2C_GenerateSTOP(I2C1, ENABLE);
48. }

```

## ■ EEPROM 的页写入时序

这个页写入的函数是根据 EEPROM 的页写入时序来编写的，见图 12-8。

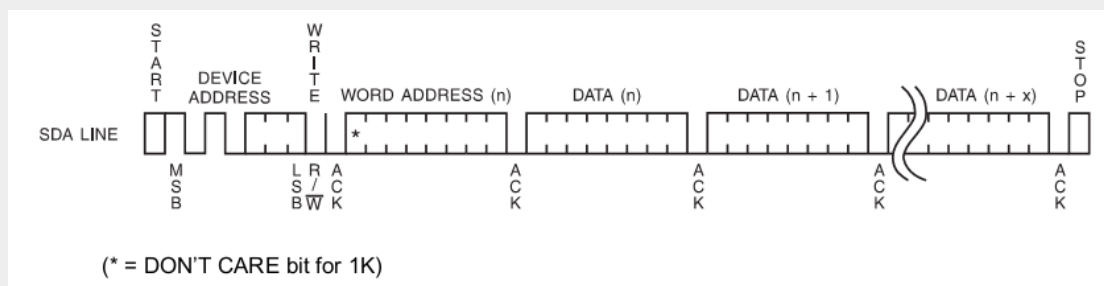


图 12-8 EEPROM 页写入时序

在 `I2C_EE_PageWrite()` 函数中，我们先不去理会使用 `while` 语句循环检测事件的语句。这样我们就可以清晰地看到这个函数的代码执行流程就是 EEPROM 的页写入时序流程。



(1) 第 17 行，调用库函数 `I2C_GenerateSTART()` 产生  $I^2C$  的通讯起始信号 `S`。其库函数说明如图 12-9。

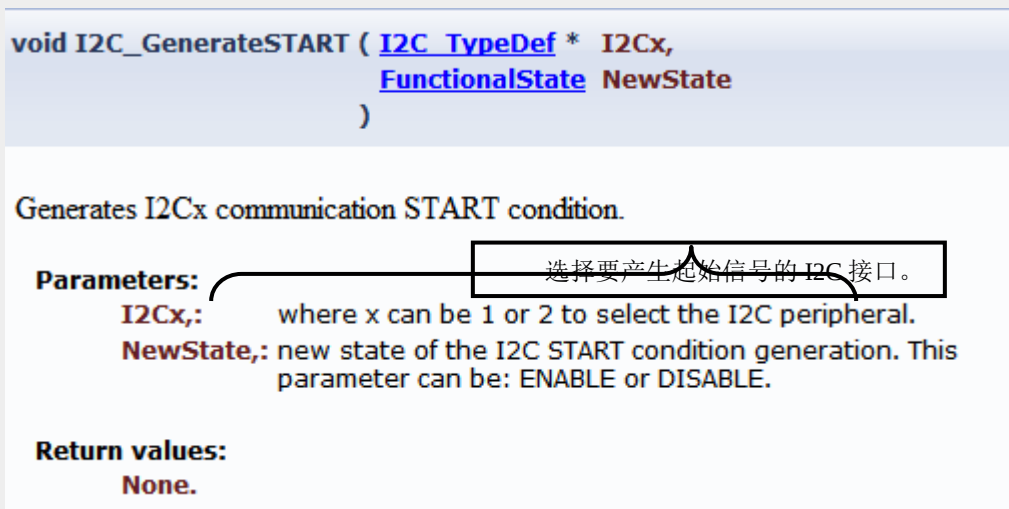


图 12-9 I2C\_GenerateSTART 函数说明

(2) 第 23 行，调用库函数 `I2C_Send7bitAddress()` 把前面条件编译中赋值的变量 `EEPROM_ADDRESS` 地址通过  $I^2C1$  接口发送出去，数据传输方向为 STM32 的  $I^2C$  发送数据 (`I2C_Direction_Transmitter`)。其中的 `EEPROM_ADDRESS` 地址是指 EEPROM 作为挂载在  $I^2C$  总线上设备寻址，而不是 EEPROM 内存存储矩阵的地址。

本库函数的说明如图 12-10：

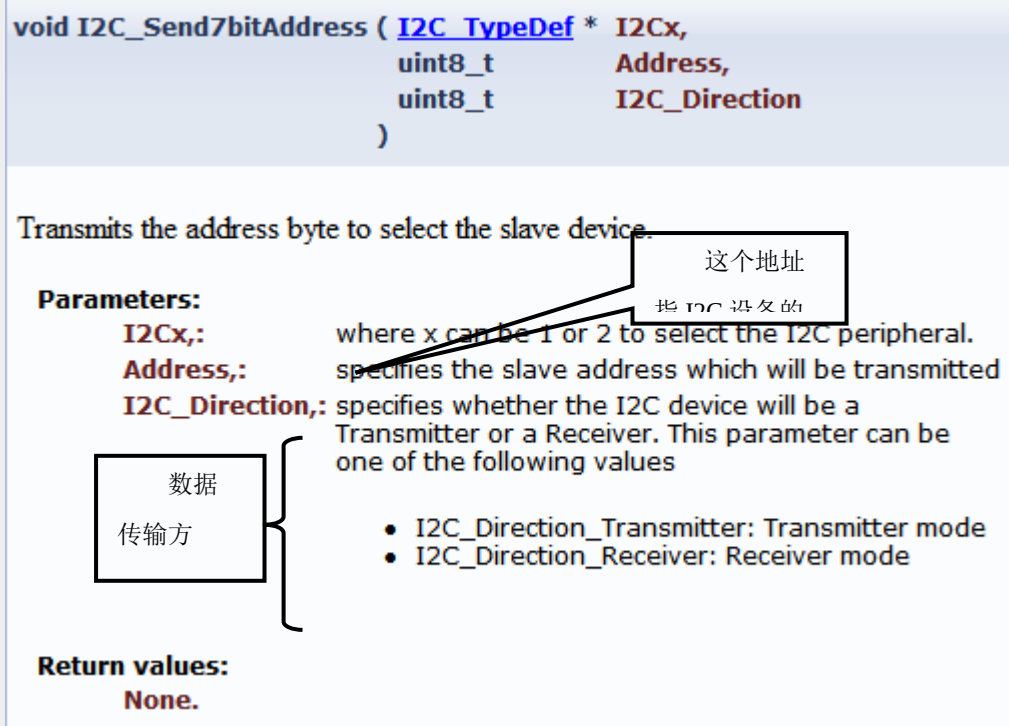


图 12-10 I2C\_Send7bitAddress 函数说明

(3) 第 29 行，调用库函数 I2C\_SendData()，请注意这个库函数的输入参数为 WriteAddr，根据 EEPROM 的页写入时序，发送完 I<sup>2</sup>C 的地址后的第一个数据，并不就是要写入 EEPROM 的数据，EEPROM 对这个数据解释为将要对存储矩阵写入的地址，这个参数 WriteAddr 是在我们调用 I2C\_EE\_PageWrite()函数时作为参数输入的，在本实验中 WriteAddr 的值由 I2C\_EE\_BufferWrite()计算得出。

I2C\_SendData()函数说明如图 12-11。

```
void I2C_SendData ( I2C_TypeDef * I2Cx,
                    uint8_t      Data
                  )
```

Sends a data byte through the I2Cx peripheral.

**Parameters:**

**I2Cx,:** where x can be 1 or 2 to select the I2C peripheral.

**Data,:** Byte to be transmitted..

**Return values:**

**None**

将要通过 I2C 接口发送的数据

图 12-11 I2C\_SendData 函数说明

这个库函数实际上是把数据传输到数据寄存器，再由 I<sup>2</sup>C 模块根据 I<sup>2</sup>C 协议发送出去。

(4) 第 35~45 行，调用 I2C\_SendData()函数，向 EEPROM 发送要写入的数据，根据 EEPROM 的页写入时序，这些数据将会被写入到前面发送的页地址中，若连续写入超过一页的最大字节数(本实验为 8 个)，则多出来的数据会重新从该页的起始地址连续写入，覆盖前面的数据。

(5) 第 47 行，调用库函数 I2C\_GenerateSTOP()产生 I<sup>2</sup>C 传输结束信号，完成一次 I<sup>2</sup>C 通讯。

其库函数说明如图 12-12。

```
void I2C_GenerateSTOP ( I2C_TypeDef * I2Cx,
                        FunctionalState NewState
                      )
```

Generates I2Cx communication STOP condition.

**Parameters:**

**I2Cx,:** where x can be 1 or 2 to select the I2C peripheral.

**NewState,:** new state of the I2C STOP condition generation. This parameter can be: ENABLE or DISABLE.

**Return values:**

**None.**

图 12-12 I2C\_GenerateSTOP 函数说明

## ■ I2C 事件检测

在以上的分析中，我们忽略了 I<sup>2</sup>C 的事件检测，而在 STM32 的 I<sup>2</sup>C 通讯中，事件的检测是必不可少的。从 stm32 参考手册的序列图可以看到，在 I<sup>2</sup>C 的通讯过程中，会产生一系列的事件，出现事件后在相应的寄存器中会产生标志位。

图245 主发送器传送序列图

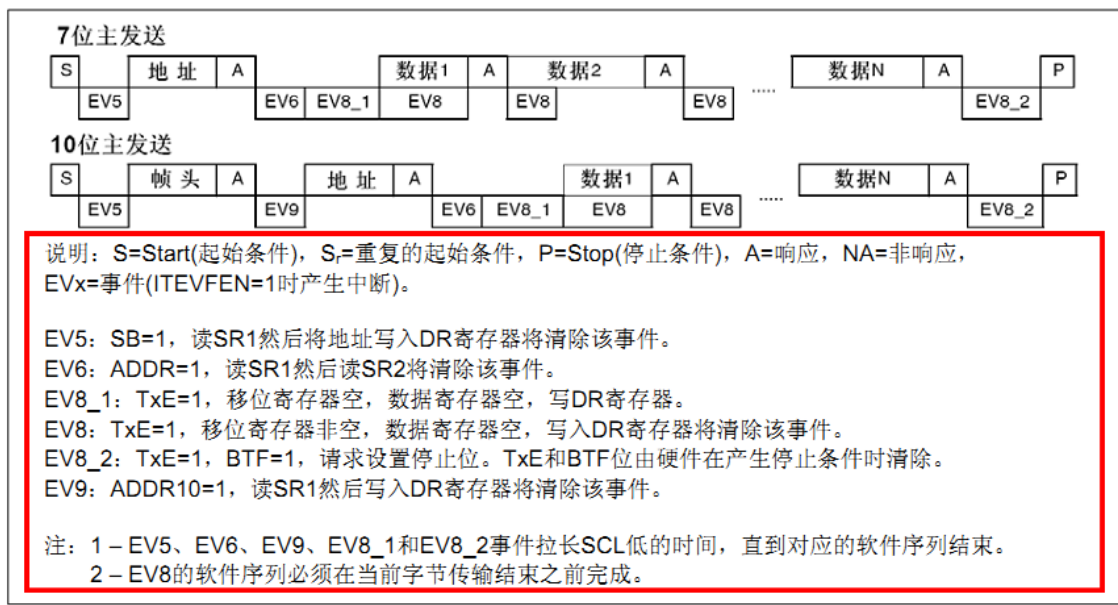


图 12-13 I2C 主发送通讯过程产生的事件

这个图的意思是：若发出了起始信号，会产生事件 5(EV5)，即 STM32 的 I2C 成为主机模式；继续发送完 I<sup>2</sup>C 设备寻址并得到应答后，会产生 EV6，即 STM32 的 I2C 成为数据发送端；之后发送数据完成会产生 EV8 等。

我们在作出 I<sup>2</sup>C 通讯操作时，可以通过循环调用库函数 I2C\_CheckEvent() 进行事件查询，以确保上一操作完成后才进行下一操作。

如：在确定 SDA 总线空闲的之后，作为主发送器的 stm32 发出起始讯号，若成功，这时会产生 EV5，我们在第 20 行：

使 用 语 句 `while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT))`

来检测这个事件，确保检测到之后再执行下一操作。其中的参数“`I2C_EVENT_MASTER_MODE_SELECT`”在固件函数说明中可以查到这就是“EV5”的宏，后面对 EV6、EV8 的相应检测操作类似。

其中库函数 `I2C_CheckEvent()` 的参数说明如图 12-14。

**Parameters:**  
**I2Cx,:** where x can be 1 or 2 to select the I2C peripheral.  
**I2C\_EVENT,:** specifies the event to be checked. This parameter can be one of the following values:

各种 I2C  
事件的参数  
及对应的事  
件编号。  
  
如：  
“I2C\_EVENT\_MASTER”

{

- `I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED` : EV1
- `I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED` : EV1
- `I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MATCHED` : EV1
- `I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED` : EV1
- `I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED` : EV1
- `I2C_EVENT_SLAVE_BYTE_RECEIVED` : EV2
- `(I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_DUALF)` : EV2
- `(I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_GENCALL)` : EV2
- `I2C_EVENT_SLAVE_BYTE_TRANSMITTED` : EV3
- `(I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_DUALF)` : EV3
- `(I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_GENCALL)` : EV3
- `I2C_EVENT_SLAVE_ACK_FAILURE` : EV3\_2
- `I2C_EVENT_SLAVE_STOP_DETECTED` : EV4
- `I2C_EVENT_MASTER_MODE_SELECT` : EV5
- ~~`I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED` : EV6~~
- `I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED` : EV6
- `I2C_EVENT_MASTER_BYTE_RECEIVED` : EV7
- `I2C_EVENT_MASTER_BYTE_TRANSMITTING` : EV8
- `I2C_EVENT_MASTER_BYTE_TRANSMITTED` : EV8\_2
- `I2C_EVENT_MASTER_MODE_ADDRESS10` : EV9

图 12-14 I2C\_CheckEvent 参数说明

## ■ 等待 EEPROM 内部写入完成

现在我们重新回到 `I2C_EE_BufferWrite()` 这个函数，在每次调用完 `I2C_EE_PageWrite()` 后，都调用了一个用户函数 `I2C_EE_WaitEepromStandbyState()`，见代码清单 12-11。

代码清单 12-11: `I2C_EE_WaitEepromStandbyState()` 函数

```
1.  /*
2.   * 函数名: I2C_EE_WaitEepromStandbyState
3.   * 描述  : Wait for EEPROM Standby state
4.   * 输入  : 无
5.   * 输出  : 无
6.   * 返回  : 无
7.   * 调用  :
8.   */
9. void I2C_EE_WaitEepromStandbyState(void)
10. {
11.     vu16 SR1_Tmp = 0;
12.
13.     do
14.     {
```

```

15.  /* Send START condition */
16.  I2C_GenerateSTART(I2C1, ENABLE);
17.  /* Read I2C1 SR1 register */
18.  SR1_Tmp = I2C_ReadRegister(I2C1, I2C_Register_SR1);
19.  /* Send EEPROM address for write */
20.  I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
21.  }while(!(I2C_ReadRegister(I2C1, I2C_Register_SR1) & 0x0002));

22.
23.  /* Clear AF flag */
24.  I2C_ClearFlag(I2C1, I2C_FLAG_AF);
25.  /* STOP condition */
26.  I2C_GenerateSTOP(I2C1, ENABLE); // Added by Najoua 27/08/20
08
27. }

```

这是利用了 EEPROM 在接收完数据后，启动内部周期写入数据的时间内不会对主机的请求作出应答的特性。所以利用这个函数循环发送起始讯号，若检测到 EEPROM 的应答，则说明 EEPROM 已经完成上一步的数据写入，进入 Standby 状态，可以进行下一步的操作了。

## ■ EEPROM 读取函数

关于 EEPROM 的数据读取函数 I2C\_EE\_BufferRead()，它的实现是跟写入函数很类似的，请读者尝试亲自分析吧，见代码清单 12-12。

代码清单 12-12： I2C\_EE\_BufferRead()函数

```

1.  /*
2.  * 函数名: I2C_EE_BufferRead
3.  * 描述   : 从 EEPROM 里面读取一块数据。
4.  * 输入   : -pBuffer 存放从 EEPROM 读取的数据的缓冲区指针。
5.  *          -WriteAddr 接收数据的 EEPROM 的地址。
6.  *          -NumByteToWrite 要从 EEPROM 读取的字节数。
7.  * 输出   : 无
8.  * 返回   : 无
9.  * 调用   : 外部调用
10. */
11. void I2C_EE_BufferRead(u8* pBuffer, u8 ReadAddr, u16 NumByteToRead)
12. {
13.  /**((u8 *)0x4001080c) |=0x80;
14.  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)); // Added by
Najoua 27/08/2008
15.
16.
17.  /* Send START condition */
18.  I2C_GenerateSTART(I2C1, ENABLE);
19.  /**((u8 *)0x4001080c) &=~0x80;
20.
21.  /* Test on EV5 and clear it */

```

```

22. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

23.
24. /* Send EEPROM address for write */
25. I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Transmitter);
26.
27. /* Test on EV6 and clear it */
28. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
29.
30. /* Clear EV6 by setting again the PE bit */
31. I2C_Cmd(I2C1, ENABLE);
32.
33. /* Send the EEPROM's internal address to write to */
34. I2C_SendData(I2C1, ReadAddr);
35.
36. /* Test on EV8 and clear it */
37. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
38.
39. /* Send STRAT condition a second time */
40. I2C_GenerateSTART(I2C1, ENABLE);
41.
42. /* Test on EV5 and clear it */
43. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

44.
45. /* Send EEPROM address for read */
46. I2C_Send7bitAddress(I2C1, EEPROM_ADDRESS, I2C_Direction_Receiver);
47.
48. /* Test on EV6 and clear it */
49. while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
50.
51. /* While there is data to be read */
52. while(NumByteToRead)
53. {
54.     if(NumByteToRead == 1)
55.     {
56.         /* Disable Acknowledgement */
57.         I2C_AcknowledgeConfig(I2C1, DISABLE);
58.
59.         /* Send STOP Condition */
60.         I2C_GenerateSTOP(I2C1, ENABLE);
61.     }
62.
63.     /* Test on EV7 and clear it */
64.     if(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED))
65.     {
66.         /* Read a byte from the EEPROM */
67.         *pBuffer = I2C_ReceiveData(I2C1);
68.
69.         /* Point to the next location where the byte read will be saved */
70.         pBuffer++;
71.
72.         /* Decrement the read bytes counter */
73.         NumByteToRead--;
74.     }
75. }
76.
77. /* Enable Acknowledgement to be ready for another reception */
/

```

```
78. I2C_AcknowledgeConfig(I2C1, ENABLE);  
79. }
```

这个读 EEPROM 函数与写的类似，也是利用 I2C\_CheckEvent()来确保通讯正常进行的，要注意的是读取数据时遵循 I<sup>2</sup>C 的标准，主发送器 stm32 要发出两次起始 I2C 讯号才能建立通讯。在这个函数中的第 18 行和第 40 行各产生了一次起始信号。

### 12.3.6. 使用 I2C 读写 EEPROM 流程总结

最后，总结一下在 stm32 如何建立与 EEPROM 的通讯。

(1) 配置 I/O 端口，确定并配置 I2C 的模式，使能 GPIO 和 I2C 时钟。

(2) 写：

- 1) 检测 SDA 是否空闲；
- 2) 按 I<sup>2</sup>C 协议发出起始信号；
- 3) 发出 7 位器件地址和写模式；
- 4) 要写入的存储区首地址；
- 5) 用页写入方式或字节写入方式写入数据；
- 6) 发送 I<sup>2</sup>C 通讯结束讯信号

每个操作之后要检测“事件”确定是否成功。写完后检测 EEPROM 是否进入 standby 状态。

(3) 读：

- 1) 检测 SDA 是否空闲；
- 2) 按 I2C 协议发出起始讯号；
- 3) 发出 7 位器件地址和写模式（伪写）；
- 4) 发出要读取的存储区首地址；
- 5) 重发起始讯号；



6) 发出 7 位器件地址和读模式;

7) 接收数据;

类似写操作, 每个操作之后要检测“事件”确定是否成功。

### 12.3.7. 实验现象

将野火 STM32 开发板供电(DC5V), 插上 JLINK, 插上串口线(两头都是母的交叉线), 打开超级终端, 配置超级终端为 115200 8-N-1, 将编译好的程序下载到开发板, 即可看到超级终端打印出如图 12-15 的信息。

写入的数据:

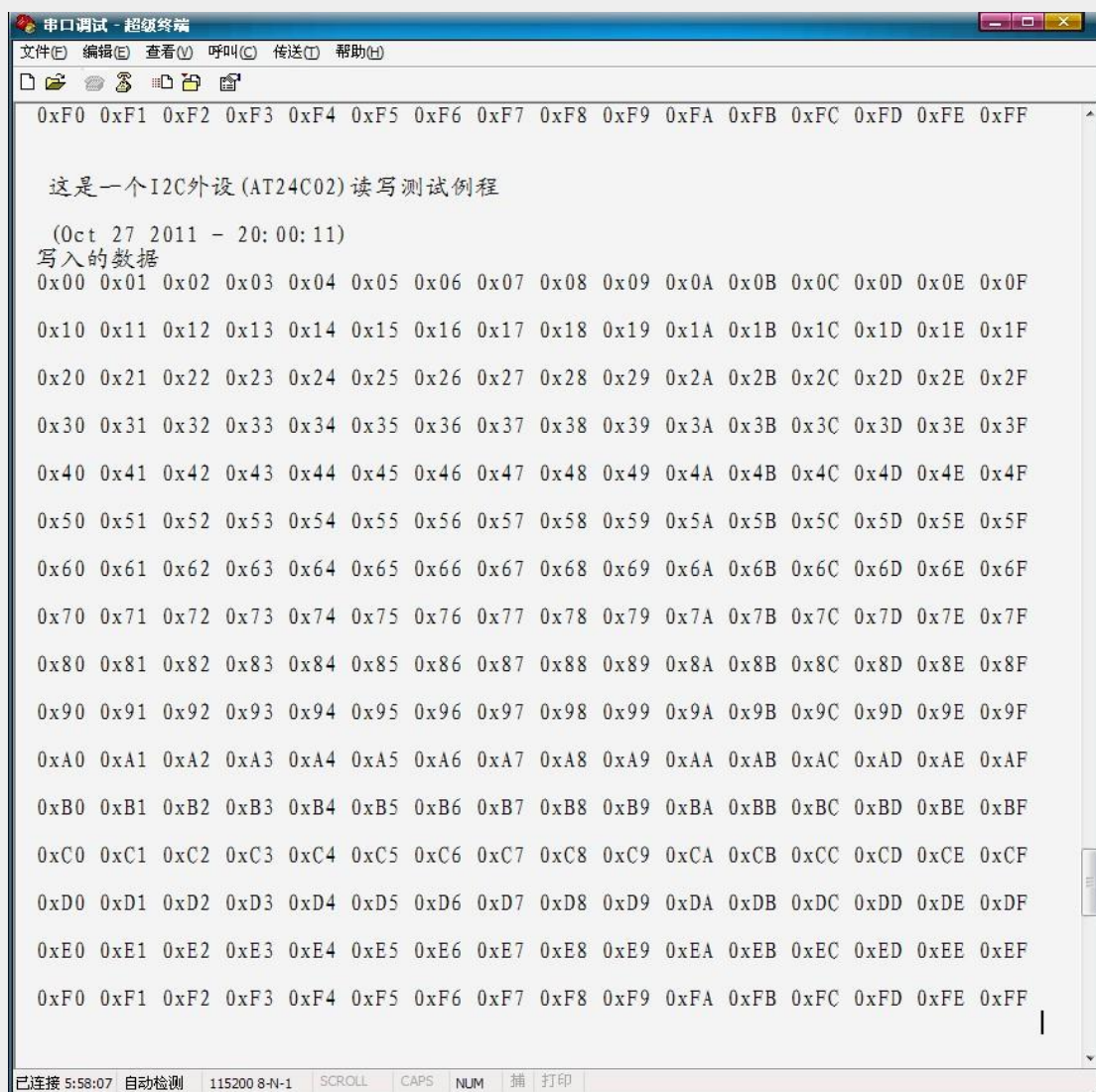


图 12-15 写入数据图

读出的数据：校验结果，读取出的数据与写入的一致，实验成功！

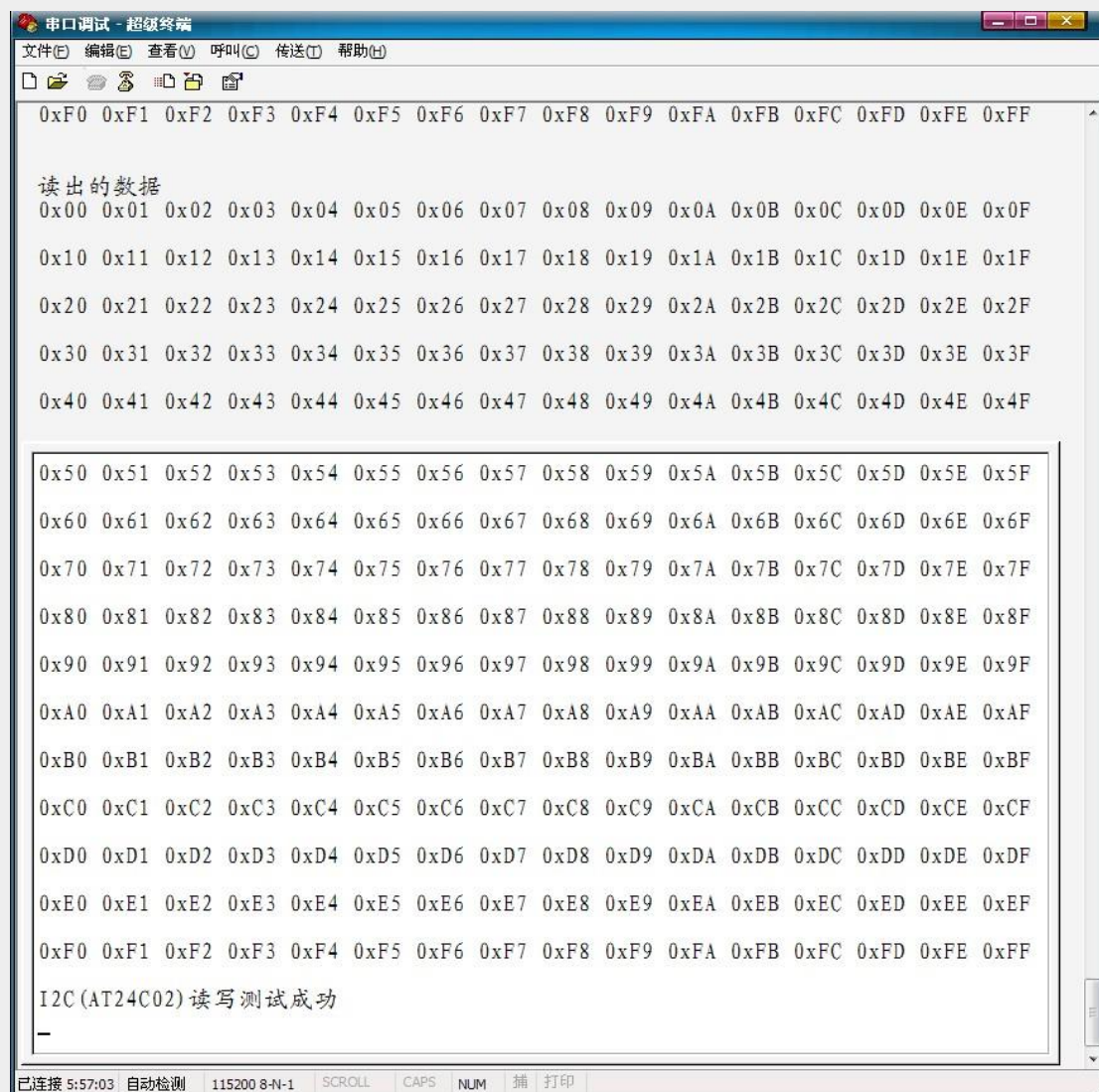


图 12-16 读出数据图

## 13. SPI-FLASH

### 13.1. SPI 协议简介

SPI 协议(Serial Peripheral Interface)，即串行外围设备接口，是一种高速全双工的通信总线，它由摩托罗拉公司提出，当前最新的为 V04.01-2004 版。它被广泛地使用在 ADC、LCD 等设备与 MCU 间通讯的场合。

#### 13.1.1. SPI 信号线

SPI 总线包含 4 条总线，分别为  $\overline{SS}$ 、SCK、MOSI、MISO。它们的作用介绍如下：

(1)  $\overline{SS}$  (Slave Select): 片选信号线，当有多个 SPI 设备与 MCU 相连时，每个设备的这个片选信号线是与 MCU 单独的引脚相连的，而其它的 SCK、MOSI、MISO 线则为多个设备并联到相同的 SPI 总线上，见图 13-1。当  $\overline{SS}$  信号线为低电平时，片选有效，开始 SPI 通讯。

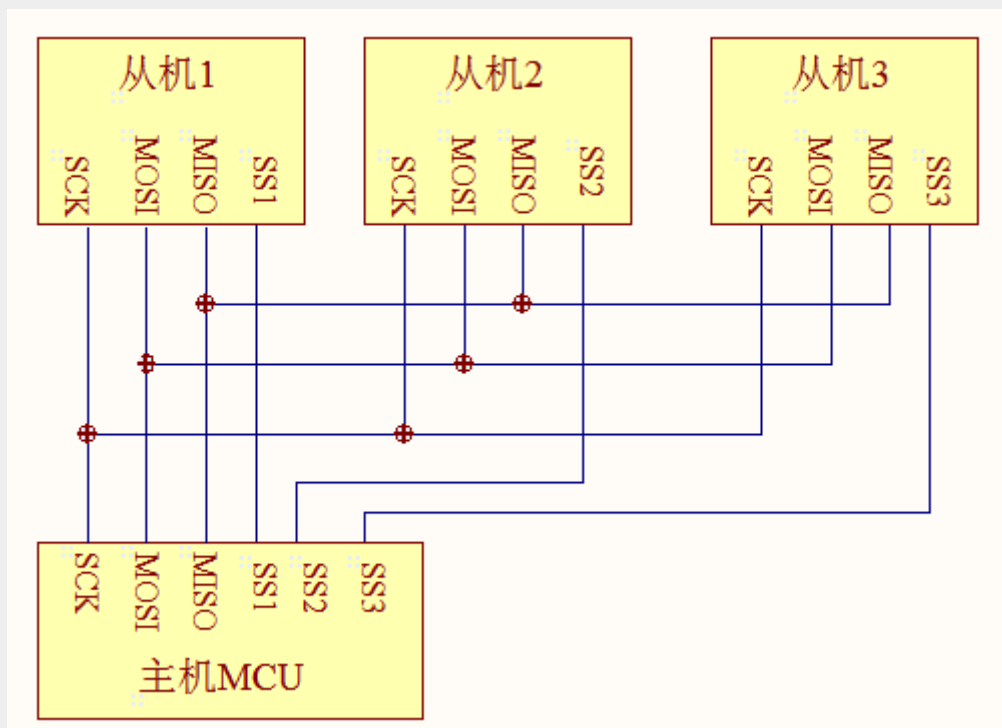


图 13-1 SPI 多设备通讯

(2) SCK (Serial Clock): 时钟信号线，由主通讯设备产生，不同的设备支持的时钟频率不一样，如 STM32 的 SPI 时钟频率最大为  $f_{\text{pclk}}/2$ 。

(3) MOSI (Master Output, Slave Input):主设备输出/从设备输入引脚。主机的数据从这条信号线输出,从机由这条信号线读入数据,即这条线上数据的方向为主机到从机。

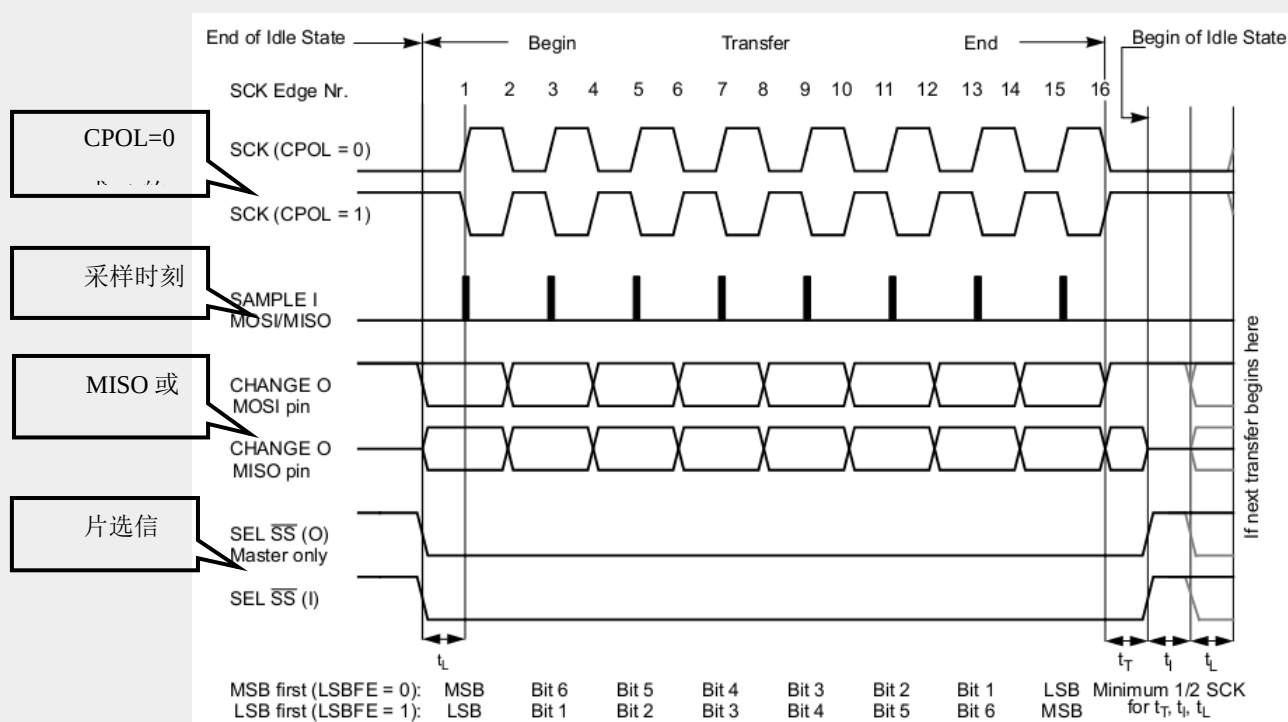
(4) MISO(Master Input, Slave Output):主设备输入/从设备输出引脚。主机从这条信号线读入数据,从机的数据则由这条信号线输出,即在这条线上数据的方向为从机到主机。

### 13.1.2. SPI 模式

根据 SPI 时钟极性(CPOL)和时钟相位(CPHA) 配置的不同,分为四种 SPI 模式。

时钟极性是指 SPI 通讯设备处于空闲状态时(也可以认为这是 SPI 通讯开始时,即 $\overline{SS}$ 线为低电平时), SCK 信号线的电平信号。CPOL=0 时, SCK 在空闲状态时为低电平, CPOL=1 时, 则相反。

时钟相位是指数据的采样的时刻,当 CPHA=0 时, MOSI 或 MISO 数据线上的信号将会在 SCK 时钟线的奇数边沿被采样。当 CPHA=1 时,数据线在 SCK 的偶数边沿采样。见图 13-2。

图 13-2  $CPHA=0$  时, SPI 时序

我们来分析这个  $CPHA=0$  的时序图。

首先, 由主机把片选信号线  $\overline{SS}$  拉低, 即为图中的  $\overline{SS}(O)$  时序, 意为主机输出,  $\overline{SS}(I)$  时序实际上也是  $\overline{SS}$  线信号,  $\overline{SS}(I)$  时序表示从机接收到  $\overline{SS}$  片选被拉低的信号。

在  $\overline{SS}$  被拉低的时刻,  $SCK$  分为两种情况, 若我们设置为  $CPOL=0$ , 则  $SCK$  时序在这个时刻为低电平, 若设置为  $CPOL=1$ , 则  $SCK$  在这个时刻为高电平。

无论  $CPOL=0$  还是  $=1$ , 因为我们配置的时钟相位  $CPHA=0$ , 在采样时刻的时序中我们可以看到, 采样时刻都是在  $SCK$  的奇数边沿(注意奇数边沿有时为下降沿, 有时为上升沿)。

因此,  $MOSI$  和  $MISO$  数据线的有效信号在  $SCK$  的奇数边沿保持不变, 这个信号将在  $SCK$  奇数边沿时被采集, 在非采样时刻,  $MOSI$  和  $MISO$  的有效信号才发生切换。

对于  $CPHA=1$  的情况也很类似, 但数据信号的采样时刻为偶数边沿, 其时序图见图 13-3。

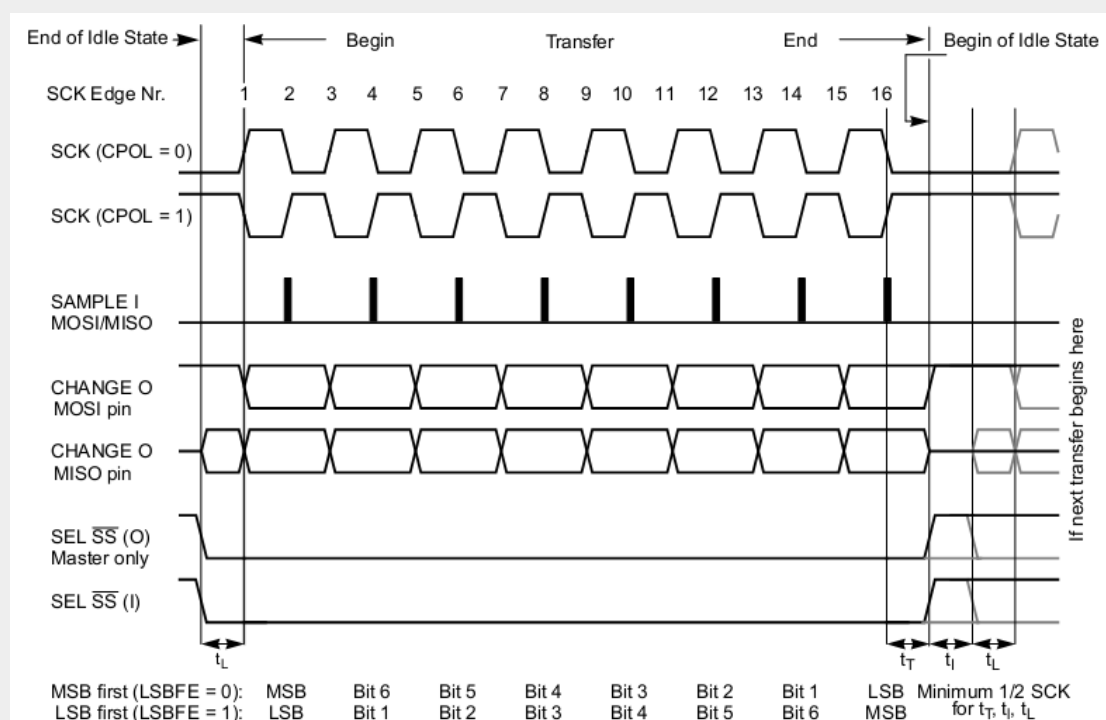


图 13-3 CPHA =1 时，SPI 时序

使用 SPI 协议通讯时，主机和从机的时序要保持一致，即两者都选择相同的 SPI 模式。

## 13.2. STM32 的 SPI 特性及架构

### 13.2.1. STM32 的 SPI 特性

STM32 的小容量产品有 1 个 SPI 接口，中容量的有 2 个，而大容量的则有 3 个。其特性如下：

- ❑ 单次传输可选择为 8 或 16 位。
- ❑ 波特率预分频系数(最大为  $f_{PCLK}/2$ )。
- ❑ 时钟极性(CPOL)和相位(CPHA)可编程设置。
- ❑ 数据顺序的传输顺序可进行编程选择，MSB 在前或 LSB 在前。
- ❑ 可触发中断的专用发送和接收标志。
- ❑ 可以使用 DMA 进行数据传输操作。



## 13.2.2. STM32 的 SPI 架构分析

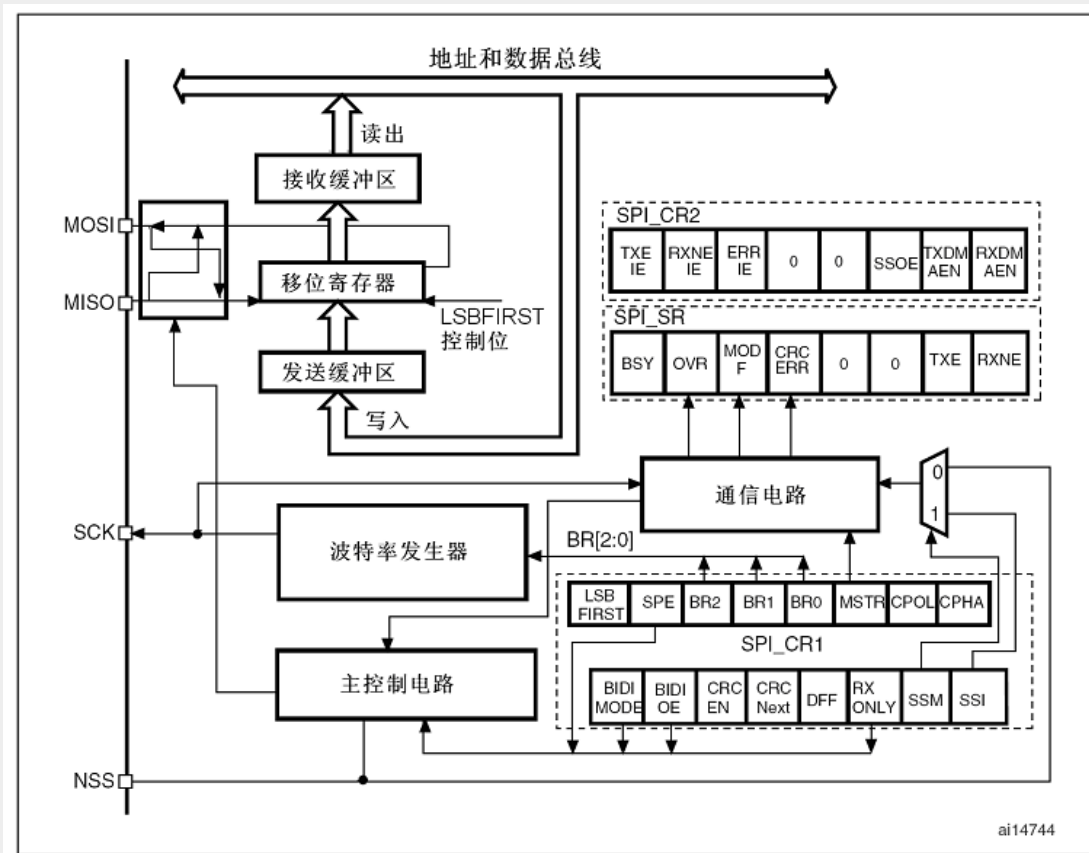


图 13-4 STM32 的 SPI 架构图

可以看到 **MISO** 数据线接收到的信号经移位寄存器处理后把数据转移到接收缓冲区，然后这个数据就可以由我们的软件从接收缓冲区读出了。

当要发送数据时，我们把数据写入到发送缓冲区，硬件将会把它用移位寄存器处理后输出到 **MOSI** 数据线。

**SCK** 的时钟信号则由波特率发生器产生，我们可以通过波特率控制位(**BR**)来控制它输出的波特率。

控制寄存器 **CR1** 掌管着主控制电路，**STM32** 的 **SPI** 模块的协议设置(时钟极性、相位等)就是由它来制定的。而控制寄存器 **CR2** 则用于设置各种中断使能。

最后的为 **NSS** 引脚，这个引脚扮演着 **SPI** 协议中的 **SS** 片选信号线的角色，如果我们把 **NSS** 引脚配置为硬件自动控制，**SPI** 模块能够自动判别它能否

成为 SPI 的主机，或自动进入 SPI 从机模式。但实际上我们用得更多的是由软件控制某些 GPIO 引脚单独作为  $\overline{SS}$  信号，这个 GPIO 引脚可以随便选择。

### 13.3. SPI 接口读取 FLASH 实例分析

本章以 SPI 读写 FLASH 的例程为大家讲解 SPI 的使用。

野火 STM32 开发板用的是 STM32F103VET6。它有 3 个 SPI 接口。本实验使用 SPI1，各信号线相应连接到 FLASH（型号：W25X16）的 CS、CLK、DO 和 DIO 线。实现 SPI 通讯，对 FLASH 进行读写。

本实验采用主模式，全双工通讯。通过查询发送数据寄存器和接收数据寄存器状态确保通讯正常。

#### 13.3.1. 实验描述及工程文件清单

##### 实验描述：

读取 FLASH 的 ID 信息，写入数据，并读取出来进行校验，通过串口打印写入与读取出来的数据，输出测试结果

##### 硬件连接：

PA4-SPI1-NSS : W25X16-CS  
PA5-SPI1-SCK : W25X16-CLK  
PA6-SPI1-MISO : W25X16-DO  
PA7-SPI1-MOSI : W25X16-DIO

##### 库文件：

使用 3.5 版本固件库  
startup/start\_stm32f10x\_hd.c  
CMSIS/core\_cm3.c  
CMSIS/system\_stm32f10x.c  
FWlib/stm32f10x\_gpio.c  
FWlib/stm32f10x\_rcc.c



FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_spi.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

USER/ spi\_flash.c

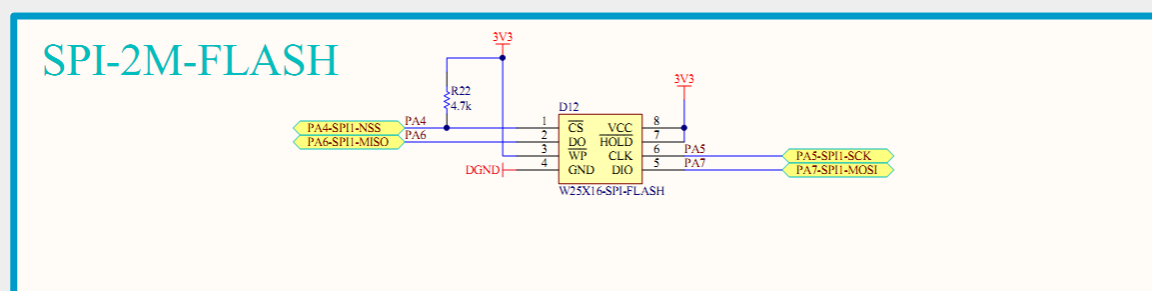


图 13-5 野火 STM32 开发板 SPI-FLASH 硬件原理图

### 13.3.2. 配置工程环境

本 SPI-2M-FLASH 实验中我们用到了 GPIO、RCC、USART 及 SPI 外设，所以我们要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_usart.c`、`stm32f10x_spi.c`。本实验中没有使用中断，采用轮询标志位的方式来确保 SPI 正常通讯。

接下来添加旧工程中的外设用户文件 `usart.c`，以便调试和观察实验效果。新建 `spi_flash.c` 及 `spi_flash.h` 文件，并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。见代码清单 13-1。

代码清单 13-1: FLASH 例程的 `stm32f10x_conf.h` 文件配置

```
1.  **
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
```

```
6.  * @date    08-April-2011
7.  * @brief    Library configuration file.
8.  *****/

9.  #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_spi.h"
12. #include "stm32f10x_usart.h"
```

### 13.3.3. main 文件

配置好所需的库文件之后，我们就从 main 函数开始分析见代码清单

13-2。

代码清单 13-2: FLASH 例程的 main 函数

```
1.  /*
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.
10.     /* 配置串口 1 为: 115200 8-N-1 */
11.     USART1_Config();
12.     printf("\r\n 这是一个 2M 串行 flash (W25X16) 实验 \r\n");
13.
14.     /* 2M 串行 flash W25X16 初始化 */
15.     SPI_FLASH_Init();
16.
17.     /* Get SPI Flash Device ID */
18.     DeviceID = SPI_FLASH_ReadDeviceID();
19.
20.     Delay( 200 );
21.
22.     /* Get SPI Flash ID */
23.     FlashID = SPI_FLASH_ReadID();
24.
25.     printf("\r\n FlashID is 0x%X, Manufacturer Device ID is 0x%X\n", FlashID, DeviceID);
26.
27.     /* Check the SPI Flash ID */
28.     if (FlashID == sFLASH_ID) /* #define sFLASH_ID 0xEF3015 */
29.     {
30.
31.         printf("\r\n 检测到华邦串行 flash W25X16 !\r\n");
32.
33.         /* Erase SPI FLASH Sector to write on */
34.         SPI_FLASH_SectorErase(FLASH_SectorToErase);
35.
36.         /* 将发送缓冲区的数据写到 flash 中 */
37.         SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, Buffer
Size);
38.         printf("\r\n 写入的数据为: %s \r\t", Tx_Buffer);
39.
```

```
40.  /* 将刚刚写入的数据读出来放到接收缓冲区中 */
41.  SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
42.      printf("\r\n 读出的数据为: %s \r\n", Tx_Buffer);
43.
44.  /* 检查写入的数据与读出的数据是否相等 */
45.  TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
46.
47.      if( PASSED == TransferStatus1 )
48.      {
49.          printf("\r\n 2M 串行 flash(W25X16) 测试成功!\n\r");
50.      }
51.  else
52.      {
53.          printf("\r\n 2M 串行 flash(W25X16) 测试失败!\n\r");
54.      }
55.  } // if (FlashID == sFLASH_ID)
56.  else
57.  {
58.      printf("\r\n 获取不到 W25X16 ID!\n\r");
59.  }
60.
61.  SPI_Flash_PowerDown();
62.  while(1);
63. }
```

这个例程的 `main` 函数很长，主要是因为后面有很多与实验验证有关的测试代码，实际上我们使用 `SPI` 读写 `FLASH` 可以写得十分简洁。

先沿着 `main` 函数的流程先走一遍，大家要清楚地认识到，本实验中 `main` 函数调用的所有函数都是用户函数。

- (1) 调用 `USART1Config()` 初始化串口。
- (2) 调用 `SPI_FLASH_Init()` 初始化 `SPI` 模块。
- (3) 调用 `SPI_FLASH_ReadDeviceID()` 读取 `FLASH` 器件生产厂商的 `ID` 信息。
- (4) 调用 `SPI_FLASH_ReadID()` 读取 `FLASH` 器件的设备 `ID` 信息。

读取器件的 `ID` 信息可以让我们知道设备与主机是否能够正常工作，也便与我们区分不同的器件。根据本实验使用的 `datasheet` 说明可以查询到器件的 `ID`，见表 13-1。我们可以在程序中进行验证。

表 13-1 `FLASH` 数据手册的设备 `ID` 说明

| MANUFACTURER ID      | (M7-M0) |  |
|----------------------|---------|--|
| Winbond Serial Flash | EFH     |  |
|                      |         |  |

| Device ID   | (ID7-ID0) | (ID15-ID0) |
|-------------|-----------|------------|
| Instruction | ABh, 90h  | 9Fh        |
| W25X16      | 14h       | 3015h      |
| W25X32      | 15h       | 3016h      |
| W25X64      | 16h       | 3017h      |

(5) 若读取得的 ID 正确，则调用 `SPI_FLASH_SectorErase()`把 FLASH 的内容擦除，擦除后调用 `SPI_FLASH_BufferWrite()`向 FLASH 写入数据，然后再调用 `SPI_FLASH_BufferRead()`从刚刚写入的地址中读出数据。最后调用 `Buffercmp()`函数对写入的数据与读取出的数据进行比较，若写入的数据与读出的数据相同，则把标志变量 `TransferStatus1` 赋值为 `PASSED`(自定义的枚举变量)。

(6) 根据标志变量 `TransferStatus1` 判断 FLASH 数据的“擦除—写入—读取”是否正常，分情况输出测试信息到终端。

(7) 若在读取 FLASH 的 ID 信息时就出错，则直接向终端输出“检测不到 FLASH”的信息。

(8) 最后调用 `SPI_Flash_PowerDown()`函数关闭 FLASH 设备的电源，因为数据写入到 FLASH 后并不会因断电而丢失，我们在使用它时才重新开启 FLASH 的电源。

接下来我们详细分析 `main` 函数中调用的以上用户函数是怎样编写的。

#### 13.3.4. SPI 初始化

`SPI_FLASH_Init()`函数初始化了 SPI1 复用到的 GPIO 引脚，启动了 GPIO 及 SPI1 外设的时钟，并初始化了 SPI 的模式。它实现如见代码清单 13-3。

代码清单 13-3: `SPI_FLASH_Init()`函数

```
1.  /*****
   **
```

```

2.  * Function Name   : SPI_FLASH_Init
3.  * Description     : Initializes the peripherals used by the SPI
FLASH driver.
4.  * Input           : None
5.  * Output          : None
6.  * Return          : None
7.  ****/
8.  void SPI_FLASH_Init(void)
9.  {
10.     SPI_InitTypeDef SPI_InitStructure;
11.     GPIO_InitTypeDef GPIO_InitStructure;
12.
13.     /* Enable SPI1 and GPIO clocks */
14.     /*!< SPI_FLASH_SPI_CS_GPIO, SPI_FLASH_SPI_MOSI_GPIO,
15.         SPI_FLASH_SPI_MISO_GPIO, SPI_FLASH_SPI_DETECT_GPIO
16.         and SPI_FLASH_SPI_SCK_GPIO Periph clock enable */
17.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_
GPIOB, ENABLE);
18.
19.     /*!< SPI_FLASH_SPI Periph clock enable */
20.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
21.
22.     /*!< Configure SPI_FLASH_SPI pins: SCK */
23.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
24.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
25.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
26.     GPIO_Init(GPIOA, &GPIO_InitStructure);
27.
28.     /*!< Configure SPI_FLASH_SPI pins: MISO */
29.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
30.     GPIO_Init(GPIOA, &GPIO_InitStructure);
31.
32.     /*!< Configure SPI_FLASH_SPI pins: MOSI */
33.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
34.     GPIO_Init(GPIOA, &GPIO_InitStructure);
35.
36.     /*!< Configure SPI_FLASH_SPI_CS_PIN pin: SPI_FLASH Card CS pi
n */
37.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
38.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
39.     GPIO_Init(GPIOA, &GPIO_InitStructure);
40.
41.     /* Deselect the FLASH: Chip Select high */
42.     SPI_FLASH_CS_HIGH();
43.
44.     /* SPI1 configuration */
45.     // W25X16: data input on the DIO pin is sampled on the rising
edge of the CLK.
46.     // Data on the DO and DIO pins are clocked out on the falling
edge of CLK.
47.     SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDu
plex;
48.     SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
49.     SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
50.     SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
51.     SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
52.     SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
53.     SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescal
er_4;
54.     SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
55.     SPI_InitStructure.SPI_CRCPolynomial = 7;
56.     SPI_Init(SPI1, &SPI_InitStructure);
57.
58.     /* Enable SPI1 */
59.     SPI_Cmd(SPI1, ENABLE);

```

60. }

SPI\_FLASH\_Init() 分为两部分。从开始到第 42 行为 GPIO 配置部分，之后为 SPI 模式配置。

## ■ GPIO 端口初始化

我们根据《STM32 数据手册》和《STM32 参考手册》把 PA5(SCK)、PA6(MISO)、PA7(MOSI)配置成复用的推挽输出模式，而由于我们的 NSS 引脚使用的是软件模式，所以我们把 PA4(NSS)配置成通用推挽输出。

表 13-2 SPI1 引脚定义

| 脚位 |   |   |   |   |   | 管脚名称 | 管脚类型 | I/O 电平 | 管脚功能 | 可选的复用功能                                       |           |
|----|---|---|---|---|---|------|------|--------|------|-----------------------------------------------|-----------|
|    |   |   |   |   |   |      |      |        |      | 默认复用功能                                        | 重映射功能     |
| 3  | 3 | 7 | 0 | 9 | 0 | A4   | P    | /O     | A4   | SPI1_NSS/USA<br>RT2_CK<br>DAC_OUT1/ADC12_IN4  |           |
| 3  | 3 | 5 | 1 | 0 | 1 | A5   | P    | /O     | A5   | SPI1_SCK<br>DAC_OUT2/ADC12_IN5                |           |
| 3  | 3 | 5 | 2 | 1 | 2 | A6   | P    | /O     | A6   | SPI1_MISO<br>/TIM8_BKIN<br>ADC12_IN6/TIM3_CH1 | TIM1_BKIN |

|   |   |   |   |   |   |    |   |    |    |   |                                                   |               |
|---|---|---|---|---|---|----|---|----|----|---|---------------------------------------------------|---------------|
| 3 | 3 | 4 | 3 | 2 | 3 | A7 | P | /O | A7 | P | SPI1_MOSI<br>/TIM8_CH1N<br>ADC12_IN7/TIM3_C<br>H2 | TIM1_C<br>H1N |
|---|---|---|---|---|---|----|---|----|----|---|---------------------------------------------------|---------------|

表 13-3 SPI 的 GPIO 引脚复用设置

| SPI 引脚    | 配置             | GPIO 配置          |
|-----------|----------------|------------------|
| SPIx_SCK  | 主模式            | 推挽复用输出           |
|           | 从模式            | 浮空输入             |
| SPIx_MOSI | 全双工模式/主模式      | 推挽复用输出           |
|           | 全双工模式/从模式      | 浮空输入或带上拉输入       |
|           | 简单的双向数据线/主模式   | 推挽复用输出           |
|           | 简单的双向数据线/从模式   | 未用，可作为通用 I/O     |
| SPIx_MISO | 全双工模式/主模式      | 浮空输入或带上拉输入       |
|           | 全双工模式/从模式      | 推挽复用输出           |
|           | 简单的双向数据线/主模式   | 未用，可作为通用 I/O     |
|           | 简单的双向数据线/从模式   | 推挽复用输出           |
| SPIx_NSS  | 硬件主/从模式        | 浮空输入或带上拉输入或带下拉输入 |
|           | 硬件主模式/NSS 输出使能 | 推挽复用输出           |
|           | 软件模式           | 未用，可作为通用 I/O     |

在本函数中第 42 行的 SPI\_FLASH\_CS\_HIGH()实际上是一个自定义的宏，如下：

```
1. #define SPI_FLASH_CS_HIGH()      GPIO_SetBits(GPIOA, GPIO_Pin_4  
)
```

实际上这个宏是用来把 SPI 的 NSS(PA4)引脚拉高，从而禁止 SPI 的通讯，在我们需要开始 SPI 通讯时，我们会使用下面的宏来把 NSS(PA4)引脚置低，开始 SPI 通讯的时序：

```
1. #define SPI_FLASH_CS_LOW()      GPIO_ResetBits(GPIOA, GPIO_Pin  
_4)
```

这样控制片选信号的方式，就是前面提到的软件控制方式，如果我们连接了多个 SPI 设备，这个片选信号可以由其它 GPIO 引脚产生。

## ■ SPI 模式初始化

从代码的第 48 行至 56 行，为 SPI 模式的初始化，对 STM32 的 SPI 初始化配置，是根据将要与之通讯的 FLASH 设备的 SPI 特性来制定的。初始化时，有如下相关的结构体成员：

### (1) SPI\_Mode

STM32 的 SPI 设备可以工作在主机模式(SPI\_Mode\_Master)或从机模式(SPI\_Mode\_Slave)，这两个模式的最大区别为 SPI 的 SCK 信号线的时序，SCK 的时序是由通讯中的主机产生的。若被配置为从机模式，STM32 的 SPI 模块将接受外来的 SCK 信号。

本实验中 STM32 作为 SPI 通讯中的主机，我们向这个成员赋值为主机模式(SPI\_Mode\_Master)。

### (2) SPI\_DataSize

这个成员可以选择 SPI 每次通讯的数据大小(称为数据帧)为 8 位还是 16 位。

从 FLASH 的数据手册我们可以查到，本 FLASH 通讯的数据帧大小为 8 位，我们把 STM32 的 SPI 模块设置要与之相同。

### (3) SPI\_CPOL 和 SPI\_CPHA



这两个成员就是配置 SPI 的时钟极性(CPOL)和时钟相位(CPHA)，这两个配置影响到 SPI 的通讯模式，要设置成符合将要互相通讯的设备的要求。

CPOL 分别可以取 SPI\_CPOL\_High(SPI 通讯空闲时 SCK 为高电平)和 SPI\_CPOL\_Low(SPI 通讯空闲时 SCK 为低电平)。

CPHA 则可以取 SPI\_CPHA\_1Edge(在 SCK 的奇数边沿采集数据)和 SPI\_CPHA\_2Edge(在 SCK 的偶数边沿采集数据)。

查阅本 FLASH 的使用手册，见图 13-6。可以了解到这个 FLASH 支持以 SPI 的模式 0 和模式 3 通讯。即在 SPI 空闲时，SCK 为低电平，奇数边沿采样（模式 0）。也可以在 SPI 空闲时，SCK 为高电平，偶数边沿采样（模式 3）。即无论 CPOL 的状态是什么，FLASH 的数据采样时刻为 SCK 的上升沿。

野火在本实验配置使用它的模式 3。即把 CPOL 赋值为 SPI\_CPOL\_High(SPI 通讯空闲时 SCK 为高电平)；CPHA 赋值为 SPI\_CPHA\_2Edge(在 SCK 的偶数边沿采集数据)。

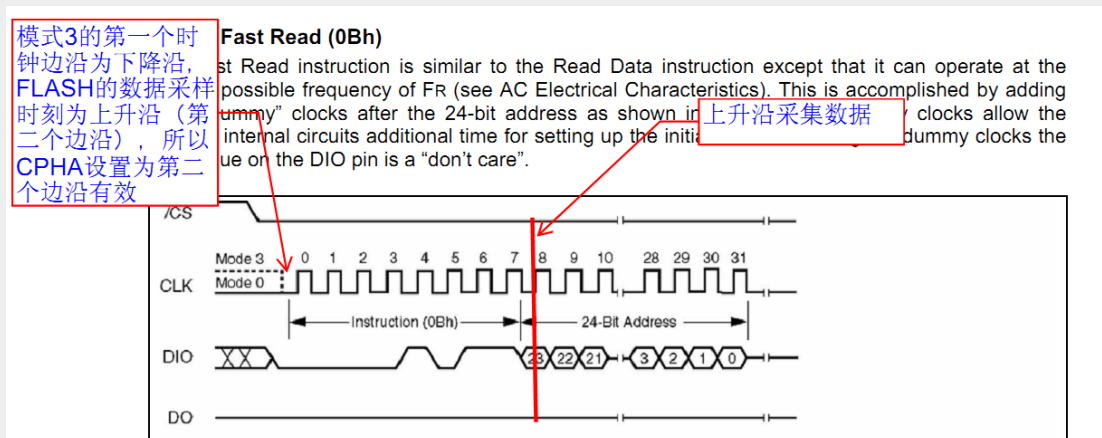


图 13-6 FLASH 时序说明图

#### (4) SPI\_NSS

本成员配置 NSS 引脚的使用模式，可以选择为硬件模式(SPI\_NSS\_Hard)与软件模式(SPI\_NSS\_Soft)，在硬件模式中的 SPI 片选信号由硬件自动产生，而软件模式则需要我们亲自把相应的 GPIO 端口拉高或置低产生非片选和片选信号。如果外界条件允许，硬件模式还会自动将 STM32 的 SPI 设置为主机。

本实验使用软件产生模式，向这个成员赋值为 SPI\_NSS\_Soft。

#### (5) SPI\_BaudRatePrescaler

本成员设置波特率分频值，分频后的时钟即为 SPI 的 SCK 信号线的时钟频率。这个成员参数可设置为 fclk 的 2、4、6、8、16、32、64、128、256 分频。

本实验向这个成员赋值为 SPI\_BaudRatePrescaler\_4，即 fclk 的 4 分频。

#### (6) SPI\_FirstBit

所有串行的通讯协议都会有 MSB 先行(高位数据在前)还是 LSB 先行(低位数据在前)的问题，而 STM32 的 SPI 模块可以通过这个结构体成员，对这个特性编程控制。

据 FLASH 的通讯时序，我们向这个成员赋值为 MSB 先行 (SPI\_FirstBit\_MSB)。

#### (7) SPI\_CRCPolynomial

这是 SPI 的 CRC 校验中的多项式，若我们使用 CRC 校验时，就使用这个成员的参数(多项式)，来计算 CRC 的值。

由于本实验的 FLASH 不支持 CRC 校验，所以我们向这个结构体成员赋值为 7 实际上是没有意义的。

配置完这些结构体成员后，我们要调用 SPI\_Init()函数把这些参数写入到寄存器中，实现 SPI 的初始化，然后调用 SPI\_Cmd()来使能 SPI1 外设。

### 13.3.5. 控制 FLASH 的命令

实际上，编写设备的驱动都是有一定的规律可循的。

首先我们要确定设备使用的是什么通讯协议。如上一章的 EEPROM 使用的是 I<sup>2</sup>C，本章的 FLASH 使用的是 SPI。那么我们就先根据它的通讯协议，选择好 STM32 的硬件模块，并进行相应的 I<sup>2</sup>C 或 SPI 模块初始化。

因为不同的设备，都会相应的有不同的指令，如 EEPROM 中会把第一个数据解释为存储矩阵的地址(实质就是指令)。而 FLASH 则定义了更多的指令，有写指令，读指令，读 ID 指令等等。

这些指令，对主机来说，只是它遵守最基本的通讯协议发送出的数据。但设备把这些数据解释成不同的意义(指令编码)，所以才成为指令。在我们配置好 STM32 的协议模块后，想要控制设备，就要遵守相应设备所定义的命令规则。

所以写驱动并不难：就是确定通讯协议模块，通过协议收发命令、数据，进而驱动设备。野火把这个称为驱动编写原理。这也是设备厂商的设计准则。

我们先来查看 FLASH 的各种指令和命令解释时序，见表 13-4。

表 13-4 FLASH 命令

| INSTR<br>UCTION<br>NAME     | B<br>YTE 1<br>CODE | BY<br>TE 2                 | B<br>YTE<br>3 | B<br>YTE<br>4 | B<br>YTE<br>5   | BYT<br>E 6     | N-<br>BYTES                  |
|-----------------------------|--------------------|----------------------------|---------------|---------------|-----------------|----------------|------------------------------|
| Write<br>Enable             | 06h                |                            |               |               |                 |                |                              |
| Write<br>Disable            | 04h                |                            |               |               |                 |                |                              |
| Read<br>Status<br>Register  | 05h                | (S7–<br>S0) <sup>(1)</sup> |               |               |                 |                | <sup>(2)</sup>               |
| Write<br>Status<br>Register | 01h                | S7–<br>S0                  |               |               |                 |                |                              |
| Read<br>Data                | 03h                | A23–<br>A16                | A<br>15–A8    | A<br>7–A0     | (<br>D7–<br>D0) | (Next<br>byte) | contin<br>uous               |
| Fast<br>Read                | 0Bh                | A23–<br>A16                | A<br>15–A8    | A<br>7–A0     | d<br>ummy       | (D7–<br>D0)    | (Next<br>Byte)<br>continuous |

|                                         |     |                      |                        |                    |                          |                                       |                                     |
|-----------------------------------------|-----|----------------------|------------------------|--------------------|--------------------------|---------------------------------------|-------------------------------------|
| Fast Read Dual Output                   | 3Bh | A23–A16              | A15–A8                 | A7–A0              | dummy                    | I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1) | (one byte per 4 clocks, continuous) |
| Page Program                            | 02h | A23–A16              | A15–A8                 | A7–A0              | (D7–D0)                  | (Next byte)                           | Up to 256 bytes                     |
| Block Erase (64KB)                      | D8h | A23–A16              | A15–A8                 | A7–A0              |                          |                                       |                                     |
| Sector Erase (4KB)                      | 20h | A23–A16              | A15–A8                 | A7–A0              |                          |                                       |                                     |
| Chip Erase                              | C7h |                      |                        |                    |                          |                                       |                                     |
| Power-down                              | B9h |                      |                        |                    |                          |                                       |                                     |
| Release Power- down / Device ID         | ABh | dummy                | dummy                  | duummy             | (ID7–ID0) <sup>(4)</sup> |                                       |                                     |
| Manufac turer/ Device ID <sup>(3)</sup> | 90h | dummy                | dummy                  | 00h                | (M7–M0)                  | (ID7–ID0)                             |                                     |
| JEDEC ID                                | 9Fh | (M7–M0) Manufacturer | (ID15–ID8) Memory Type | (ID7–ID0) Capacity |                          |                                       |                                     |

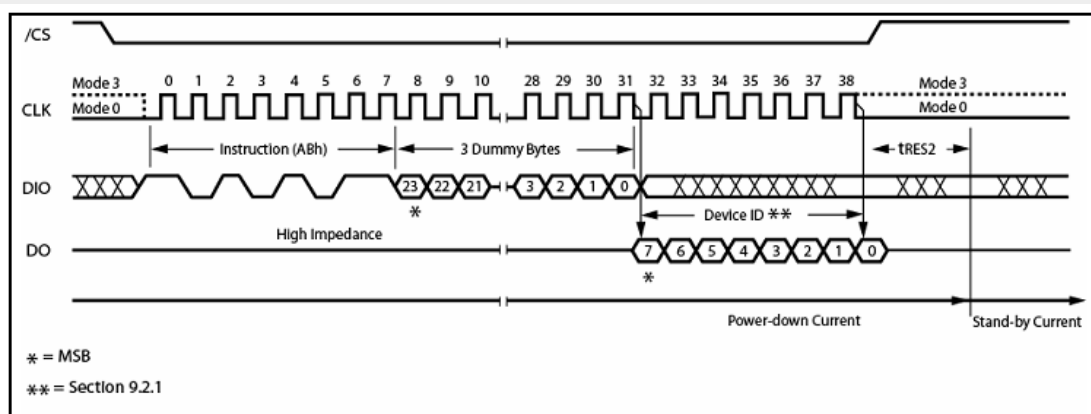


图 13-7 FLASH 读 ID 命令解释时序

指令表中的 A0~A23 指地址；M0~M7 为器件的制造商 ID (MANUFACTURER ID)；D0~D7 为数据。

在命令列表可以了解到读取设备 ID 的命令(Device ID)编码为 ABh、dummy、dummy、dummy。表示此命令由这四个字节组成，其中 dummy 意为任意编码，即这几个字节必须发送数据，但这些数据是任意的。命令列表中带括号的字节数据表示由 FLASH 返回给主机的响应，可以看到 Device ID 命令的第 5 个字节为从机返回的响应，(ID7~ID0)，即返回设备的 ID 号。

使用 Device ID 命令时的时序图见图 13-7，可以看到主机首先通过 MOSI 线(即 FLASH 的 DIO 线)发送第一个字节为 ABh 编码，紧接着三个字节的 dummy 编码，然后 FLASH 就忽略 DIO 线上的信号，通过 MISO 线(即 FLASH 的 DO 线)把它的 FLASH 设备 ID 发送给主机。

了解了 Device ID 命令及其时序，我们分析 SPI\_FLASH\_ReadDeviceID()函数来说明驱动编写原理。这个函数实现了读取 FLASH 的 ID 的功能，见代码清单 13-4。

代码清单 13-4: SPI\_FLASH\_ReadDeviceID()函数

```
1.  /*****
*
2.  * Function Name   : SPI_FLASH_ReadID
3.  * Description    : Reads FLASH identification.
4.  * Input          : None
5.  * Output         : None
6.  * Return         : FLASH identification
7.  *****/
8.  u32 SPI_FLASH_ReadDeviceID(void)
9.  {
```

```

10.  u32 Temp = 0;
11.  /* Select the FLASH: Chip Select low */
12.  SPI_FLASH_CS_LOW();
13.  /* Send "RDID " instruction */
14.  SPI_FLASH_SendByte(W25X_DeviceID);
15.  SPI_FLASH_SendByte(Dummy_Byte);
16.  SPI_FLASH_SendByte(Dummy_Byte);
17.  SPI_FLASH_SendByte(Dummy_Byte);
18.  /* Read a byte from the FLASH */
19.  Temp = SPI_FLASH_SendByte(Dummy_Byte);
20.  /* Deselect the FLASH: Chip Select high */
21.  SPI_FLASH_CS_HIGH();
22.  return Temp;
23. }

```

这个函数的代码流程严格遵从 Device ID 命令的时序：

(1) 第 12 行，调用宏 SPI\_FLASH\_CS\_LOW()，这是一个自定义的宏，使 FLASH 的片选有效，以使能 FLASH 设备；

(2) 第 14 行，利用用户函数 SPI\_FLASH\_SendByte() 来向 FLASH 发送第一个命令字节编码：“W25X\_DeviceID”（自定义的宏：0xAB）；

(3) 根据指令表，发送完这个指令后，后面紧跟着三个字节的“dummy byte”，野火把 Dummy\_Byte 宏定义为“0xff”，实际上改成其它宏编码也无影响。

(4) 第 19 行，完整的命令在前面已经发送完毕，根据时序，在第 5 个字节，FLASH 通过 DIO 端口输出它的器件 ID，我们调用函数 SPI\_FLASH\_SendByte() 接收返回的数据，并赋值给 Temp 变量。SPI\_FLASH\_ReadDeviceID()函数的返回值即为读取得的器件 ID

(5) 把片选信号拉高，结束通讯。

这就完成了读 FLASH ID。

在这个读 FLASH ID 函数中多次调用了一个相对底层的用户函数，它实现了利用 SPI 发送和接收数据的功能，见代码清单 13-5。

代码清单 13-5: SPI\_FLASH\_SendByte()函数

```

1.  /*****
2.  * Function Name   : SPI_FLASH_SendByte
3.  * Description    : Sends a byte through the SPI interface and r
4.  *                : eceived from the SPI bus.
5.  * Input          : byte : byte to send.
6.  * Output         : None

```

```

7.  * Return          : The value of the received byte.
8.  ****
**/
9.  u8 SPI_FLASH_SendByte(u8 byte)
10. {
11.  /* Loop while DR register in not empty */
12.  while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET
);
13.
14.  /* Send byte through the SPI1 peripheral */
15.  SPI_I2S_SendData(SPI1, byte);
16.
17.  /* Wait to receive a byte */
18.  while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET
);
19.
20.  /* Return the byte read from the SPI bus */
21.  return SPI_I2S_ReceiveData(SPI1);
22. }

```

- (1) 调用库函数 `SPI_I2S_GetFlagStatus()` 等待发送数据寄存器清空。
- (2) 发送数据寄存器准备好后，调用库函数 `SPI_I2S_SendData()` 向从机发送数据。
- (3) 调用库函数 `SPI_I2S_GetFlagStatus()` 等待接收数据寄存器非空。
- (4) 接收寄存器非空时，调用 `SPI_I2S_ReceiveData()` 获取接收寄存器中的数据并作为函数的返回值，这个数据即由从机发送给主机的数据。

这是最底层的发送数据和接收数据的函数，利用了库函数的标志检测确保通讯正常。

### 13.3.6. 读取厂商 ID

对于其它函数，编写的方法是很类似的，如读取厂商 ID 的函数 `SPI_FLASH_ReadID()` 见代码清单 13-6。

代码清单 13-6: `SPI_FLASH_ReadID()` 函数

```

1.  /*****
**
2.  * Function Name   : SPI_FLASH_ReadID
3.  * Description    : Reads FLASH identification.
4.  * Input          : None
5.  * Output         : None
6.  * Return         : FLASH identification
7.  ****
**/

```

```

8. u32 SPI_FLASH_ReadID(void)
9. {
10.     u32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;
11.
12.     /* Select the FLASH: Chip Select low */
13.     SPI_FLASH_CS_LOW();
14.
15.     /* Send "RDID " instruction */
16.     SPI_FLASH_SendByte(W25X_JedecDeviceID);
17.
18.     /* Read a byte from the FLASH */
19.     Temp0 = SPI_FLASH_SendByte(Dummy_Byte);
20.
21.     /* Read a byte from the FLASH */
22.     Temp1 = SPI_FLASH_SendByte(Dummy_Byte);
23.
24.     /* Read a byte from the FLASH */
25.     Temp2 = SPI_FLASH_SendByte(Dummy_Byte);
26.
27.     /* Deselect the FLASH: Chip Select high */
28.     SPI_FLASH_CS_HIGH();
29.
30.     Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;
31.
32.     return Temp;
33. }

```

这个函数根据命令流程，发送一个字节的命令编码 JEDEC ID (9Fh)，后，从机就通过 DO 线返回厂商 ID 及 0~16 位的设备 ID。

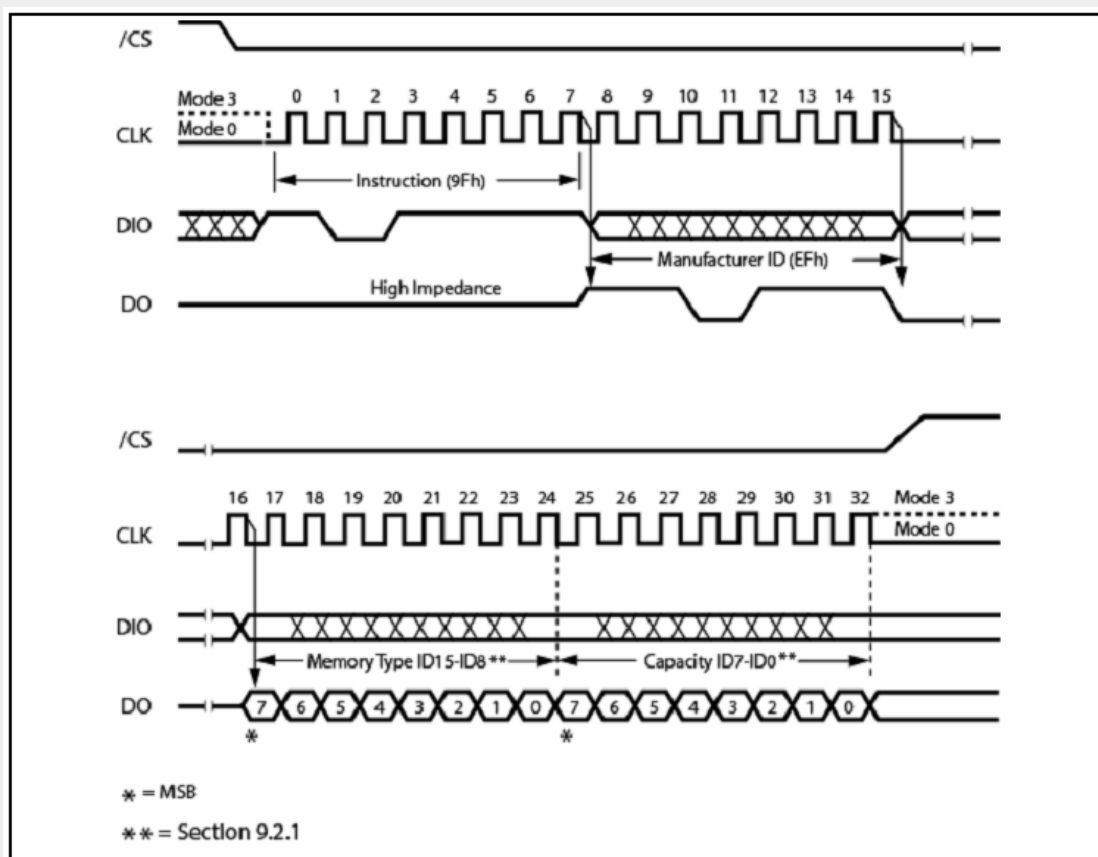


图 13-8 读厂商 ID 时序图



### 13.3.7. 擦除 FLASH 内容

#### ■ 扇区擦除

根据 FLASH 的存储原理，在写入数据前，要先对存储区域进行擦除，所以执行 `SPI_FLASH_SectorErase()` 函数对要写入的扇区进行擦除，也称为预写。见代码清单 13-7。

代码清单 13-7: `SPI_FLASH_SectorErase()` 函数

```
1.  /*****
***
2.  * Function Name   : SPI_FLASH_SectorErase
3.  * Description    : Erases the specified FLASH sector.
4.  * Input          : SectorAddr: address of the sector to erase.
5.  * Output         : None
6.  * Return         : None
7.  *****/
8.  void SPI_FLASH_SectorErase(u32 SectorAddr)
9.  {
10. /* Send write enable instruction */
11. SPI_FLASH_WriteEnable();
12. SPI_FLASH_WaitForWriteEnd();
13. /* Sector Erase */
14. /* Select the FLASH: Chip Select low */
15. SPI_FLASH_CS_LOW();
16. /* Send Sector Erase instruction */
17. SPI_FLASH_SendByte(W25X_SectorErase);
18. /* Send SectorAddr high nibble address byte */
19. SPI_FLASH_SendByte((SectorAddr & 0xFF0000) >> 16);
20. /* Send SectorAddr medium nibble address byte */
21. SPI_FLASH_SendByte((SectorAddr & 0xFF00) >> 8);
22. /* Send SectorAddr low nibble address byte */
23. SPI_FLASH_SendByte(SectorAddr & 0xFF);
24. /* Deselect the FLASH: Chip Select high */
25. SPI_FLASH_CS_HIGH();
26. /* Wait the end of Flash writing */
27. SPI_FLASH_WaitForWriteEnd();
28. }
```

先忽略其中的 `SPI_FLASH_WriteEnable()`和 `SPI_FLASH_WaitForWriteEnd()` 函数，从第 15 行至第 25 行是纯粹的关于 FLASH 擦除操作，扇区擦除命令时序如下：

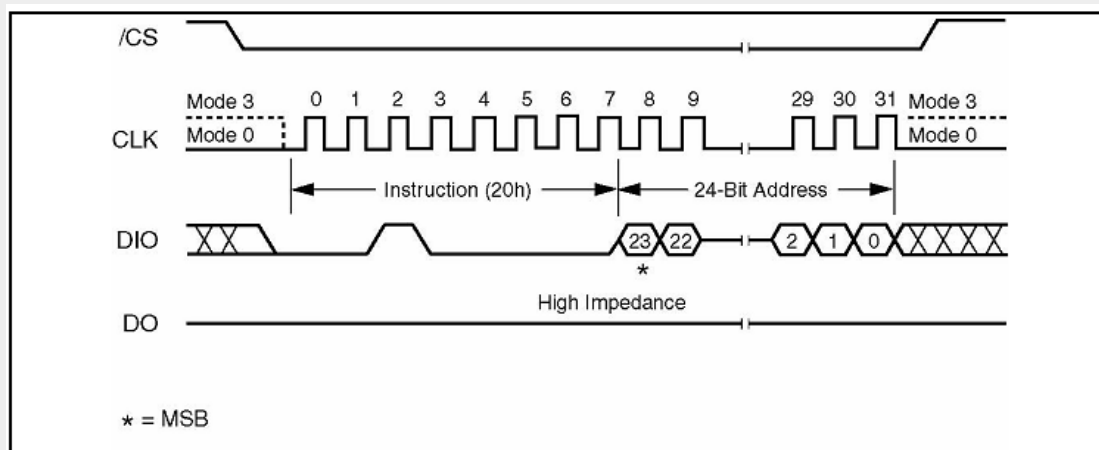


图 13-9 扇区擦除时序

其中的第一个字节为扇区擦除命令编码(20h)，紧跟其后的为要进行擦除的 24 位起始地址。根据 FLASH 的说明，它把整个存储矩阵分为块区和扇区，每块(Block)的大小为 64KB，每个扇区(Sector)的大小为 4KB，对存储矩阵进行擦除时，最小的单位为扇区。由于篇幅问题，就不贴上它的地址分布图了。

## ■ 写使能

根据 FLASH 的读写要求，在页写入、扇区擦除、块擦除、整片擦除及写状态寄存器前，都需要先发送写使能命令。在 SPI\_FLASH\_SectorErase()函数的第 11 行就调用了用户函数 SPI\_FLASH\_WriteEnable()，它的具体实现见代码清单 13-8。

代码清单 13-8: SPI\_FLASH\_WriteEnable()函数

```
1.  /*****
2.  **
3.  * Function Name : SPI_FLASH_WriteEnable
4.  * Description   : Enables the write access to the FLASH.
5.  * Input         : None
6.  * Output        : None
7.  * Return        : None
8.  *****/
9.  void SPI_FLASH_WriteEnable(void)
10. {
11.     /* Select the FLASH: Chip Select low */
12.     SPI_FLASH_CS_LOW();
13.     /* Send "Write Enable" instruction */
14.     SPI_FLASH_SendByte(W25X_WriteEnable);
15.     /* Deselect the FLASH: Chip Select high */
16.     SPI_FLASH_CS_HIGH();
17. }
```

```
18. }
```

本函数十分简单，就是根据写使能命令的时序，发送了写使能命令 `write enable`（06h 编码）。

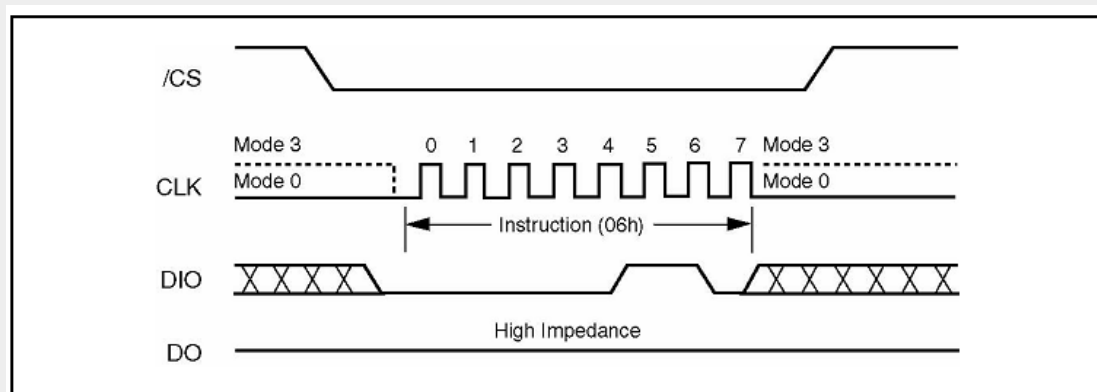


图 13-10 写使能命令时序

## ■ 读 FLASH 状态

在擦除函数 `SPI_FLASH_SectorErase()` 中，还调用用户函数 `SPI_FLASH_WaitForWriteEnd()` 来确保在 FLASH 不忙碌的时候，才发送命令或数据。这个函数通过读取 FLASH 的状态寄存器来获知它的工作状态，实现见代码清单 13-9。

代码清单 13-9: `SPI_FLASH_WaitForWriteEnd()` 函数

```
1.  /*****
**
2.  * Function Name   : SPI_FLASH_WaitForWriteEnd
3.  * Description    : Polls the status of the Write In Progress (W
IP) flag in the
4.  *                FLASH's status register and loop until w
rite opertaion
5.  *                has completed.
6.  * Input          : None
7.  * Output         : None
8.  * Return         : None
9.  *****/

10. void SPI_FLASH_WaitForWriteEnd(void)
11. {
12.     u8 FLASH_Status = 0;
13.
14.     /* Select the FLASH: Chip Select low */
15.     SPI_FLASH_CS_LOW();
16.
17.     /* Send "Read Status Register" instruction */
18.     SPI_FLASH_SendByte(W25X_ReadStatusReg);
```

```
19.
20.  /* Loop as long as the memory is busy with a write cycle */
21.  do
22.  {
23.      /* Send a dummy byte to generate the clock needed by the FLASH
24.      and put the value of the status register in FLASH_Status variable */
25.      FLASH_Status = SPI_FLASH_SendByte(Dummy_Byte);
26.  }
27.  while ((FLASH_Status & WIP_Flag) == SET); /* Write in progress */
28.
29.  /* Deselect the FLASH: Chip Select high */
30.  SPI_FLASH_CS_HIGH();
31. }
```

本函数实质是不断循环地检测 FLASH 状态寄存器的 Busy 位，直到 FLASH 的内部写时序完成，从而确保下一通讯操作正常。主机通过发送读状态寄存器命令 Read Status Register(05h 编码)，返回的为它的 8 位状态寄存的值，这个命令的时序及状态寄存器位的说明如图 13-11 和图 13-12。

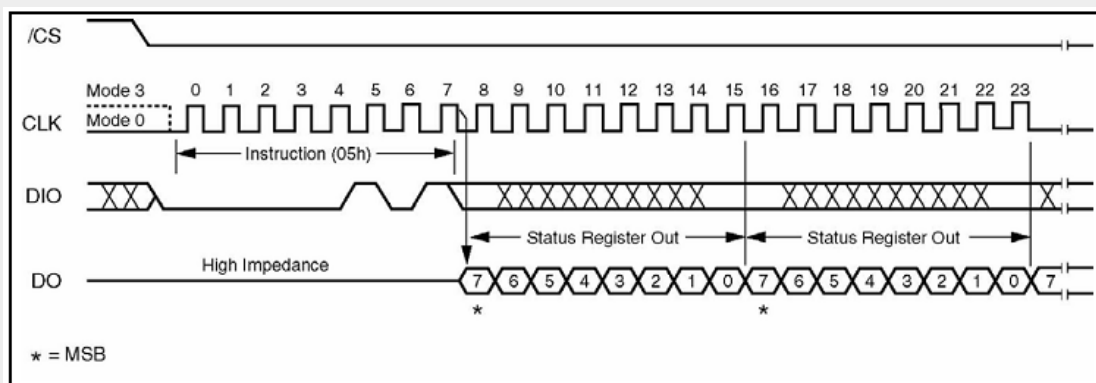


图 13-11 读状态寄存器时序

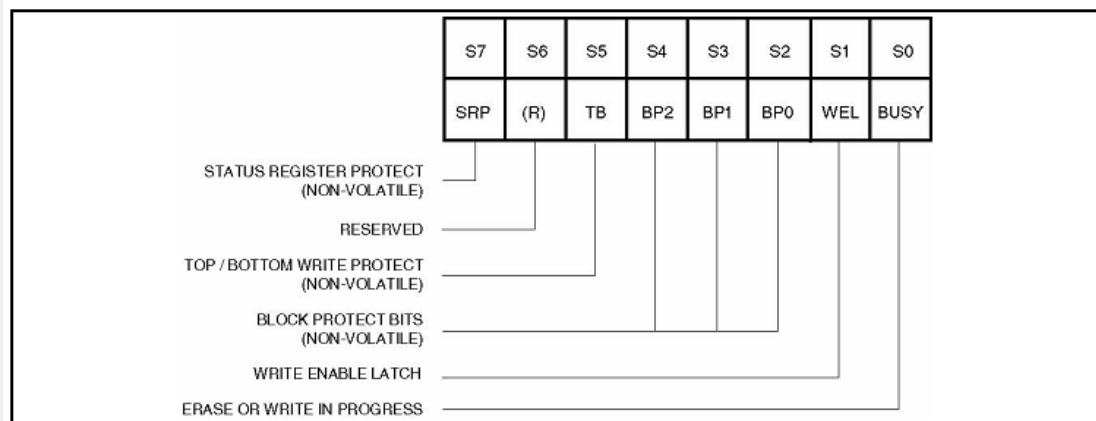


图 13-12 状态寄存器位说明

本函数检测的就是状态寄存器的 Bit0, BUSY 位，当 FLASH 在执行内部写时序的时候，除了读状态寄存器命令，其它一切命令它都会忽略，并且

BUSY 位保持为 1，即我们需要等待 BUSY 位为 0 的时候，再向 FLASH 发送其它命令。

### 13.3.8. 向 FLASH 写入数据

对 FLASH 写入数据，最小单位是 256 字节，厂商把这个单位称为页。写入时，一般也只有页写入的方式。因而我们为了方便地把一个很长的数组写入到 FLASH 中时，一般需要进行转换，把数组按页分好，再写入到 FLASH 中，如同 I<sup>2</sup>C 例子中对 EEPROM 的页写入一样，只是页的大小不同而已，这个函数见代码清单 13-10。

代码清单 13-10: SPI\_FLASH\_BufferWrite()函数

```

1.
2. /*****
**
3. * Function Name   : SPI_FLASH_BufferWrite
4. * Description    : Writes block of data to the FLASH. In this f
unction, the
5. *                number of WRITE cycles are reduced, using Pa
ge WRITE sequence.
6. * Input          : -
pBuffer : pointer to the buffer containing the data to be
7. *                written to the FLASH.
8. *                -
WriteAddr : FLASH's internal address to write to.
9. *                -
NumByteToWrite : number of bytes to write to the FLASH.
10. * Output         : None
11. * Return          : None
12. *****/
/
13. void SPI_FLASH_BufferWrite(u8* pBuffer, u32 WriteAddr, u16 NumB
yteToWrite)
14. {
15.     u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp
= 0;
16.
17.     Addr = WriteAddr % SPI_FLASH_PageSize;
18.     count = SPI_FLASH_PageSize - Addr;
19.     NumOfPage = NumByteToWrite / SPI_FLASH_PageSize;
20.     NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;
21.
22.     if (Addr == 0) /* WriteAddr is SPI_FLASH_PageSize aligned */
23.     {
24.         if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize
*/
25.         {
26.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
27.         }
28.         else /* NumByteToWrite > SPI_FLASH_PageSize */
29.         {
30.             while (NumOfPage--)
31.             {
32.                 SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageS
ize);
33.                 WriteAddr += SPI_FLASH_PageSize;
34.                 pBuffer += SPI_FLASH_PageSize;
35.             }
36.
37.             SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
38.         }
39.     }
40.     else /* WriteAddr is not SPI_FLASH_PageSize aligned */
41.     {
42.         if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize
*/
43.         {
44.             if (NumOfSingle > count) /* (NumByteToWrite + WriteAddr)
> SPI_FLASH_PageSize */
45.             {

```

```

46.         temp = NumOfSingle - count;
47.
48.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
49.         WriteAddr += count;
50.         pBuffer += count;
51.
52.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, temp);
53.     }
54.     else
55.     {
56.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite)
;
57.     }
58. }
59. else /* NumByteToWrite > SPI_FLASH_PageSize */
60. {
61.     NumByteToWrite -= count;
62.     NumOfPage = NumByteToWrite / SPI_FLASH_PageSize;
63.     NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;
64.
65.     SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
66.     WriteAddr += count;
67.     pBuffer += count;
68.
69.     while (NumOfPage--)
70.     {
71.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageS
ize);
72.         WriteAddr += SPI_FLASH_PageSize;
73.         pBuffer += SPI_FLASH_PageSize;
74.     }
75.
76.     if (NumOfSingle != 0)
77.     {
78.         SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
79.     }
80. }
81. }
82. }

```

在 SPI\_FLASH\_BufferWrite()中，对数组进行分页后，它调用了用户函数 SPI\_FLASH\_PageWrite()来对数据进行按页写入见代码清单 13-11。

代码清单 13-11: SPI\_FLASH\_PageWrite()函数

```

1.  /*****
2.  * Function Name   : SPI_FLASH_PageWrite
3.  * Description    : Writes more than one byte to the FLASH with
a single WRITE
4.  *                cycle(Page WRITE sequence). The number of by
te can't exceed
5.  *                the FLASH page size.
6.  * Input          : -
pBuffer : pointer to the buffer containing the data to be
7.  *                written to the FLASH.
8.  *                -
WriteAddr : FLASH's internal address to write to.
9.  *                -
NumByteToWrite : number of bytes to write to the FLASH,

```

```
10.*                                     must be equal or less than "SPI_FLASH_Page
Size" value.
11.* Output                           : None
12.* Return                           : None
13.*****
/
14.void SPI_FLASH_PageWrite(u8* pBuffer, u32 WriteAddr, u16 NumByte
eToWrite)
15.{
16.    /* Enable the write access to the FLASH */
17.    SPI_FLASH_WriteEnable();
18.
19.    /* Select the FLASH: Chip Select low */
20.    SPI_FLASH_CS_LOW();
21.    /* Send "Write to Memory " instruction */
22.    SPI_FLASH_SendByte(W25X_PageProgram);
23.    /* Send WriteAddr high nibble address byte to write to */
24.    SPI_FLASH_SendByte((WriteAddr & 0xFF0000) >> 16);
25.    /* Send WriteAddr medium nibble address byte to write to */
26.    SPI_FLASH_SendByte((WriteAddr & 0xFF00) >> 8);
27.    /* Send WriteAddr low nibble address byte to write to */
28.    SPI_FLASH_SendByte(WriteAddr & 0xFF);
29.
30.    if(NumByteToWrite > SPI_FLASH_PerWritePageSize)
31.    {
32.        NumByteToWrite = SPI_FLASH_PerWritePageSize;
33.        //printf("\n\r Err: SPI_FLASH_PageWrite too large!");
34.    }
35.
36.    /* while there is data to be written on the FLASH */
37.    while (NumByteToWrite-->0)
38.    {
39.        /* Send the current byte */
40.        SPI_FLASH_SendByte(*pBuffer);
41.        /* Point on the next byte to be written */
42.        pBuffer++;
43.    }
44.
45.    /* Deselect the FLASH: Chip Select high */
46.    SPI_FLASH_CS_HIGH();
47.
48.    /* Wait the end of Flash writing */
49.    SPI_FLASH_WaitForWriteEnd();
50.}
```

其页写入时序如图 13-13，发送完页写入命令 Page Program (编码 02h)及地址之后，可以连续写入最多 256 个字节的数据，在发送完数据之后，我们调用 SPI\_FLASH\_WaitForWriteEnd()等待 FLASH 内部写时序完成再退出函数。



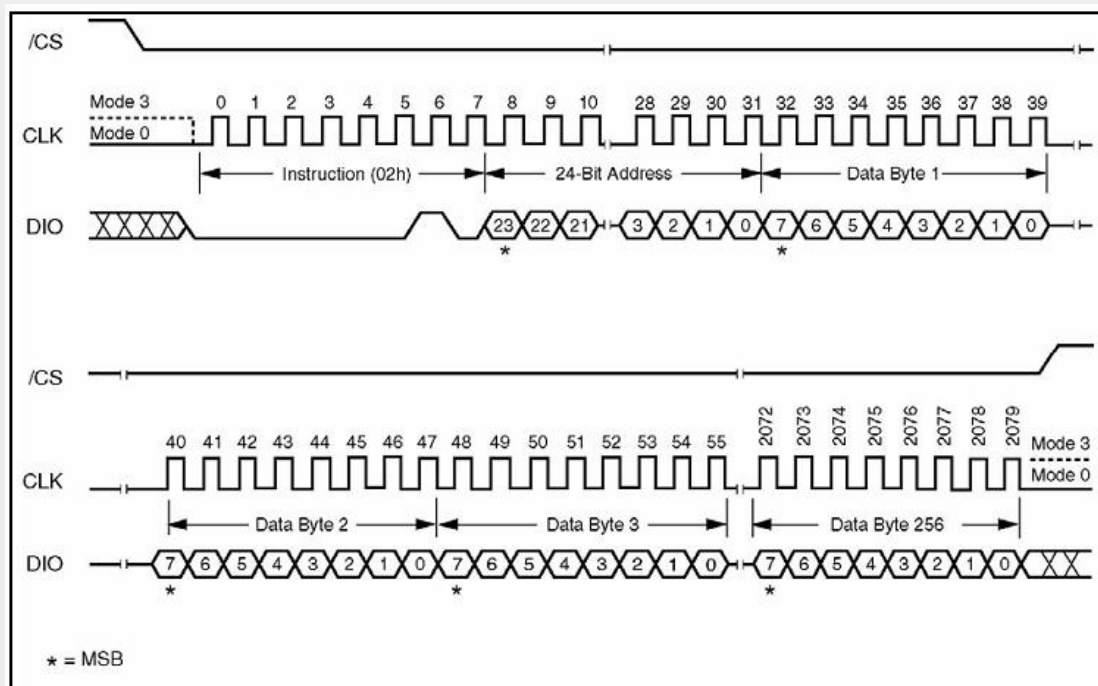


图 13-13 页写入时序

### 13.3.9. 从 FLASH 读取数据

对于读数据，发出一个命令后，可以无限制地一直把整个 FLASH 的数据都读取完，若认为读取的数据量足够了，可以拉高片选信号以表示读取数据结束，我们使用的读数据函数 `SPI_FLASH_BufferRead()` 见代码清单 13-12。

代码清单 13-12: `SPI_FLASH_BufferRead()` 函数

```

1.  /*****
***
2.  * Function Name   : SPI_FLASH_BufferRead
3.  * Description    : Reads a block of data from the FLASH.
4.  * Input          : -
   pBuffer : pointer to the buffer that receives the data read
5.  *              from the FLASH.
6.  *              -
   ReadAddr : FLASH's internal address to read from.
7.  *              -
   NumByteToRead : number of bytes to read from the FLASH.
8.  * Output         : None
9.  * Return         : None
10. *****/
11. void SPI_FLASH_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
12. {
13.     /* Select the FLASH: Chip Select low */
14.     SPI_FLASH_CS_LOW();
15.
16.     /* Send "Read from Memory " instruction */

```

```
17. SPI_FLASH_SendByte(W25X_ReadData);
18.
19. /* Send ReadAddr high nibble address byte to read from */
20. SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
21. /* Send ReadAddr medium nibble address byte to read from */
22. SPI_FLASH_SendByte((ReadAddr & 0xFF00) >> 8);
23. /* Send ReadAddr low nibble address byte to read from */
24. SPI_FLASH_SendByte(ReadAddr & 0xFF);
25.
26. while (NumByteToRead--
) /* while there is data to be read */
27. {
28.     /* Read a byte from the FLASH */
29.     *pBuffer = SPI_FLASH_SendByte(Dummy_Byte);
30.     /* Point to the next location where the byte read will be s
aved */
31.     pBuffer++;
32. }
33.
34. /* Deselect the FLASH: Chip Select high */
35. SPI_FLASH_CS_HIGH();
36. }
```

读数据命令时序如图 13-14。

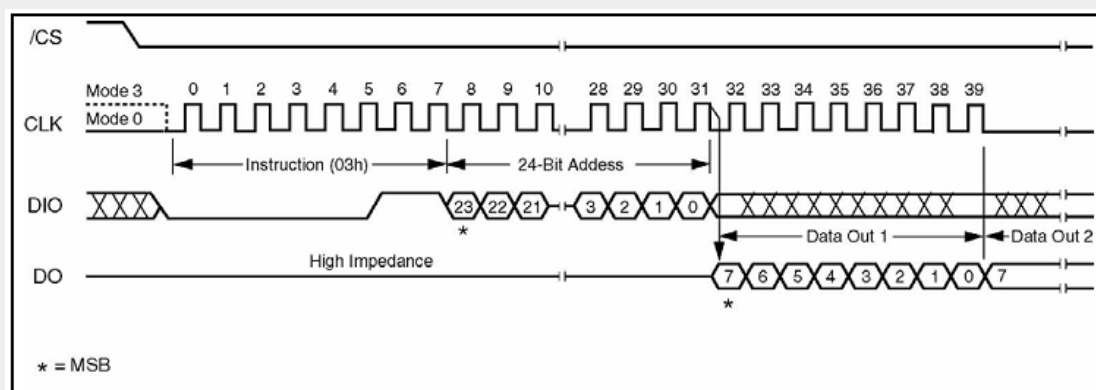


图 13-14 读数据命令时序

`SPI_FLASH_BufferRead()` 函数首先发送了读数据命令 `Read Data`(编码 03h)，紧接着发送 24 位的读数据起始地址，STM32 再通过 DO 线接收数据，并把它使用指针的方式记录起来。

### 13.3.10. 小结

主要通过以上的函数，我们就可以实现对 FLASH 的擦除、写入和读取操作了，还有一小部分函数没有分析到，大家根据需求，在使用到的时候再分析一下源码吧。

最后，总结一下在 STM32 如何建立与 SPI-FLASH 的通讯。

- (1) 配置 I/O 端口，使能 GPIO；
- (2) 根据将要进行通讯器件的 SPI 模式，配置 STM32 的 SPI，使能 SPI 时钟；
- (3) 配置好 SPI 后，根据各种 FLASH 定义的命令控制对它的读写。

---

注意在写操作前要先进行存储扇区的擦除操作，擦除操作前也要先发出“写使能”命令。

---

### 13.3.11. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如**错误!未找到引用源。**的信息：





## 14. 定时器

### 14.1. 定时器功能简介

区别于 SysTick 一般只用于系统时钟的计时，STM32 的定时器外设功能强大得超出了想像力，《STM32 参考手册》中仅对定时器的介绍就已经占了 100 多页。STM32 一共有 8 个都为 16 位的定时器。其中 TIM6、TIM7 是基本定时器；TIM2、TIM3、TIM4、TIM5 是通用定时器；TIM1 和 TIM8 是高级定时器。这些定时器使 STM32 具有定时、信号的频率测量、信号的 PWM 测量、PWM 输出、三相 6 步电机控制及编码器接口等功能，都是专门为工控领域量身订做的。

### 14.2. 定时器工作分析

#### 14.2.1. 基本定时器

基本定时器 TIM6 和 TIM7，这两个定时器只具备最基本的定时功能，就是累加的时钟脉冲数超过预定值时，能触发中断或触发 DMA 请求。由于在芯片内部与 DAC 外设相连，可通过触发输出驱动 DAC。也可以作为其它通用定时器的时钟基准。见图 14-1。

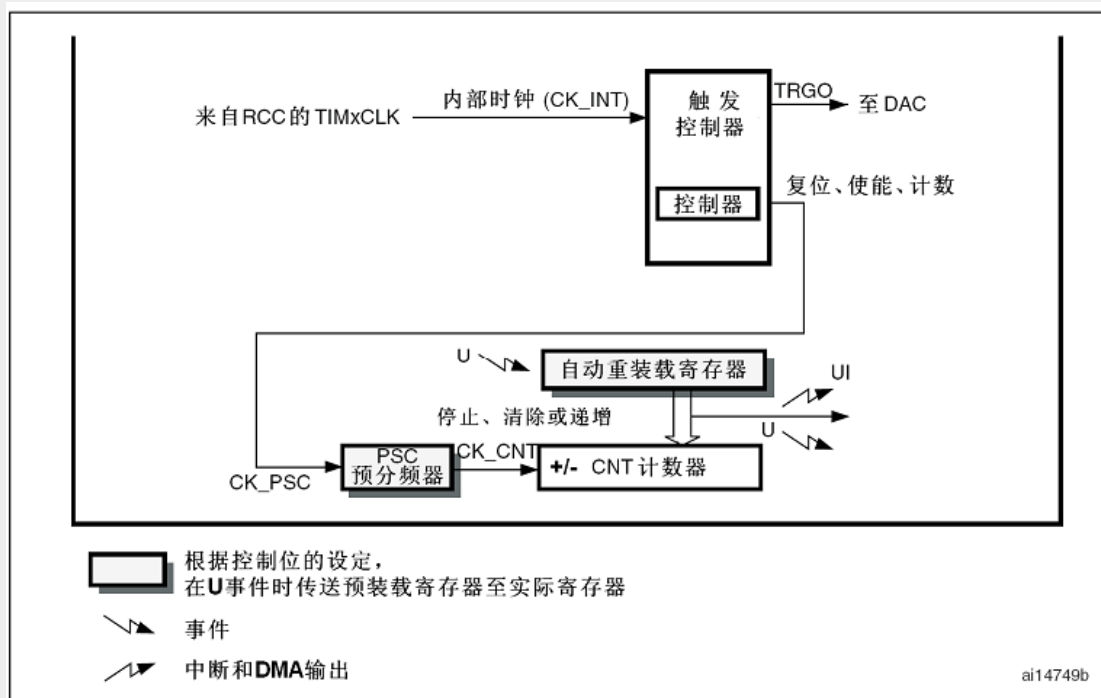


图 14-1 基本定时器结构

这两个基本定时器使用的时钟源都是  $TIMxCLK$ ，时钟源经过 PSC 预分频器输入至脉冲计数器 ( $TIMx\_CNT$ )，基本定时器只能工作在向上计数模式，在重载寄存器 ( $TIMx\_ARR$ ) 保存的是定时器的溢出值。

工作时，脉冲计数器  $TIMx\_CNT$  由时钟触发进行计数，当  $TIMx\_CNT$  的计数值  $X$  等于重载寄存器  $TIMx\_ARR$  中保存的数值  $N$  时，产生溢出事件，可触发中断或 DMA 请求。然后  $TIMx\_CNT$  的值重新被置为 0，重新向上计数。

#### 14.2.2. 通用定时器

相比之下，通用定时器  $TIM2\sim TIM5$ ，就比基本定时器复杂得多了。除了基本的定时，它主要用在测量输入脉冲的频率、脉冲宽与输出 PWM 脉冲的场合，还具有编码器的接口。见图 14-5。

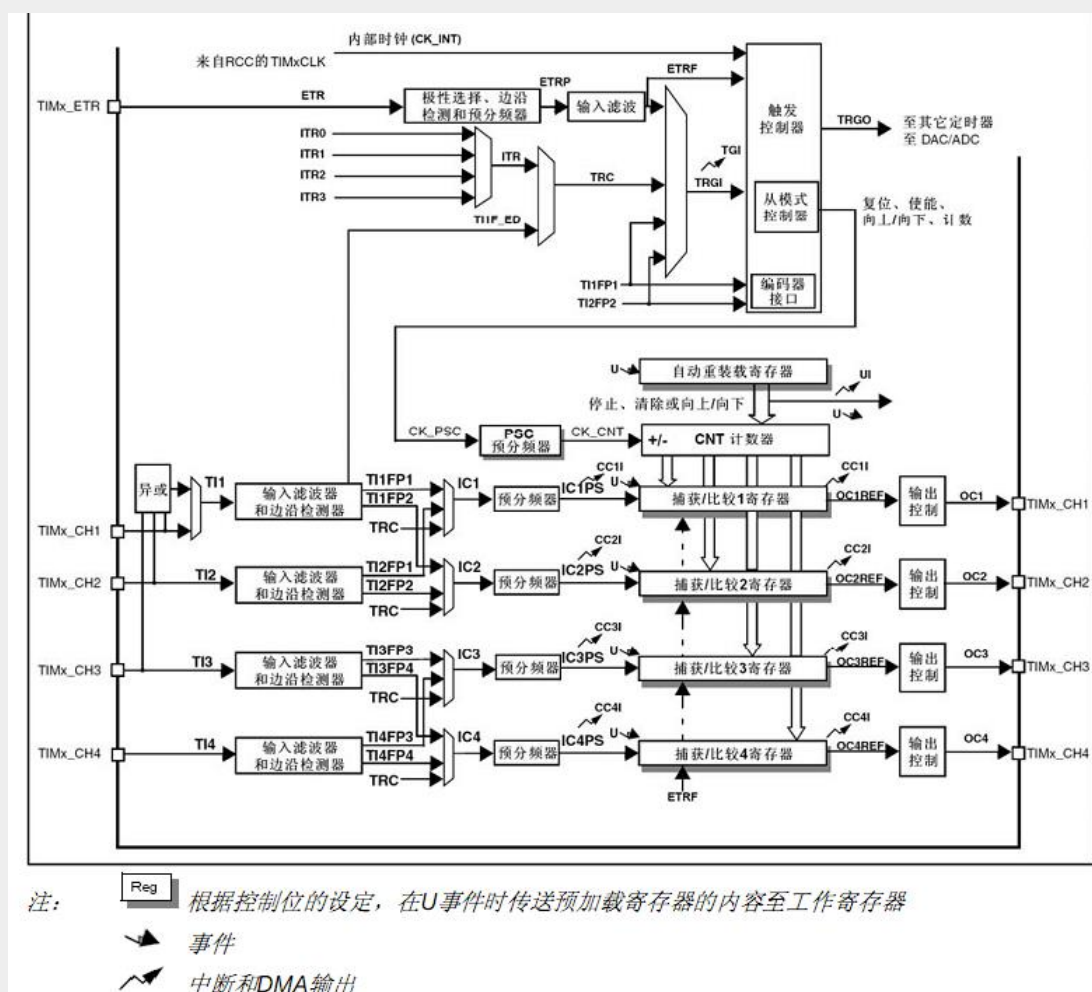


图 14-2 通用定时器结构

## ■ 捕获/比较寄存器

通用定时器的基本计时功能跟基本定时器的的工作方式是一样的，同样把时钟源经过预分频器输出到脉冲计数器 TIMx\_CNT 累加，溢出时就产生中断或 DMA 请求。

而通用定时器比基本定时器多出的强大功能，就是因为通用定时器多出了一种寄存器——捕获/比较寄存器，TIMx\_CCR(capture/compare register)，它在输入时被用于捕获(存储)输入脉冲在电平发生翻转时，脉冲计数器 TIMx\_CNT 的当前计数值，从而实现脉冲的频率测量；在输出时被用来存储一个脉冲数值，把这个数值用于与脉冲计数器 TIMx\_CNT 的当前计数值进行比较，根据比较结果进行不同的电平输出。

## ■ PWM 输出过程分析

通用定时器可以利用 GPIO 引脚进行脉冲输出，在配置为比较输出、PWM 输出功能时，捕获/比较寄存器 TIMx\_CCR 被用作比较功能，下面把它简称为比较寄存器。

这里直接举例说明定时器的 PWM 输出工作过程：若配置脉冲计数器 TIMx\_CNT 为向上计数，而重载寄存器 TIMx\_ARR 被配置为 N，即 TIMx\_CNT 的当前计数值数值 X 在 TIMxCLK 时钟源的驱动下不断累加，当 TIMx\_CNT 的数值 X 大于 N 时，会重置 TIMx\_CNT 数值为 0 重新计数。

而在 TIMx\_CNT 计数的同时，TIMx\_CNT 的计数值 X 会与比较寄存器 TIMx\_CCR 预先存储了的数值 A 进行比较，当脉冲计数器 TIMx\_CNT 的数值 X 小于比较寄存器 TIMx\_CCR 的值 A 时，输出高电平（或低电平），相反地，当脉冲计数器的数值 X 大于或等于比较寄存器的值 A 时，输出低电平（或高电平）。

如此循环，得到的输出脉冲周期就为重载寄存器 TIMx\_ARR 存储的数值 (N+1) 乘以触发脉冲的时钟周期，其脉冲宽度则为比较寄存器 TIMx\_CCR 的值 A 乘以触发脉冲的时钟周期，即输出 PWM 的占空比为  $A/(N+1)$ 。

见图 14-3，图中的为重载寄存器 TIMx\_ARR 被配置为 N=8，向上计数，比较寄存器 TIMx\_CCR 的值被设置为 A= 4、8、大于 8、等于 0 时的输出时序图。图中 OCxREF 即为 GPIO 引脚的输出时序、CCxIF 为触发中断的时序。



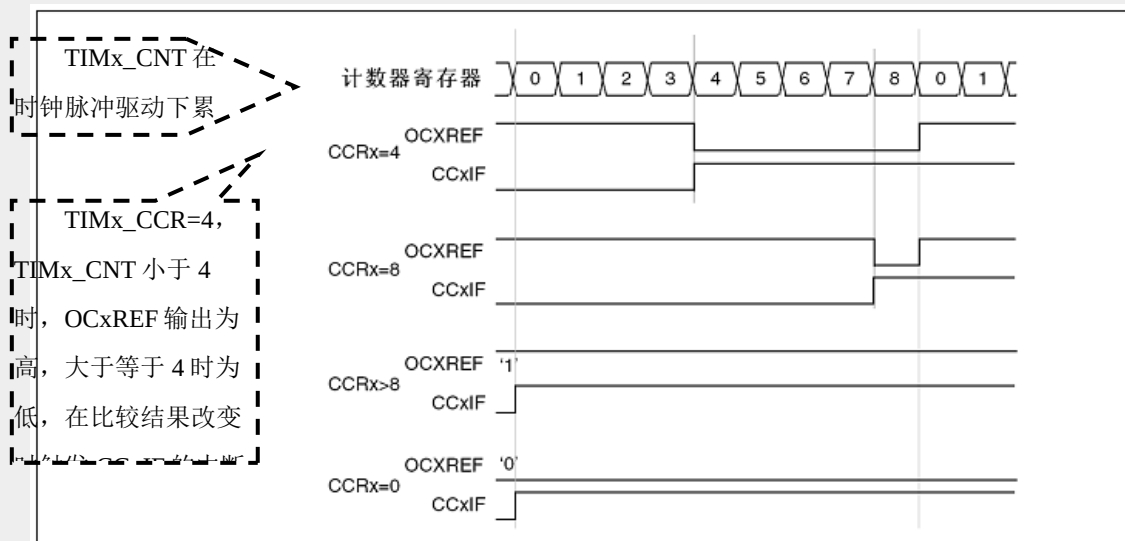


图 14-3 PWM 输出模式

## ■ 测量 PWM 输入过程分析

而当定时器被配置为输入功能时，可以用于检测输入到 GPIO 引脚的信号 (频率检测、输入 PWM 检测)，此时捕获/比较寄存器 TIMx\_CCR 被用作捕获功能，下面把它简称为捕获寄存器。

见图 14-4，为 PWM 输入时的脉冲宽检测时序图。

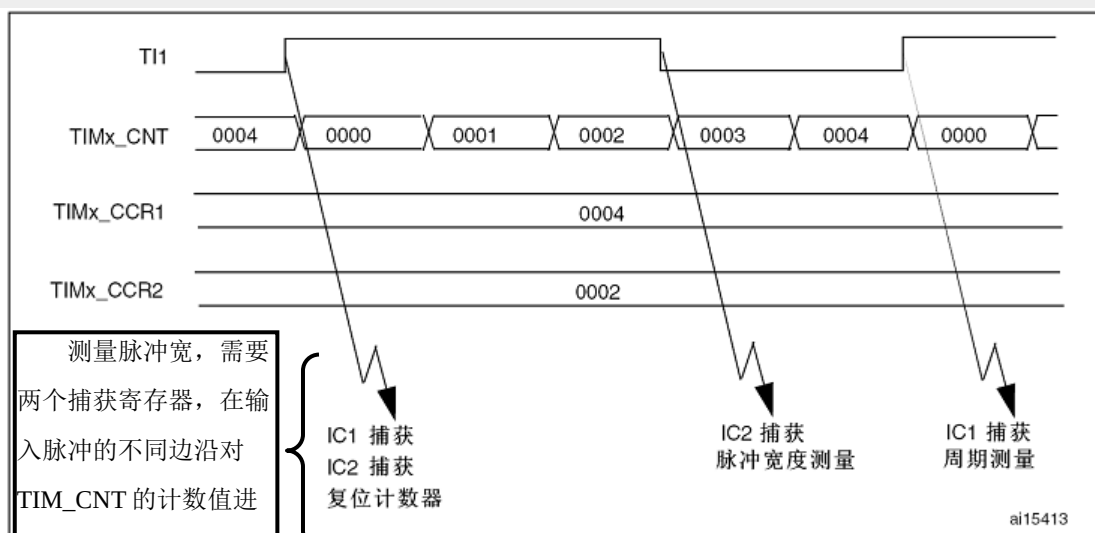


图 14-4 PWM 输入检测

按照上面的时序图来分析 PWM 输入脉冲宽检测的工作过程：要测量的 PWM 脉冲通过 GPIO 引脚输入到定时器的脉冲检测通道，其时序为图中的

TI1。把脉冲计数器 TIMx\_CNT 配置为向上计数，重载计数器重载寄存器 TIMx\_ARR 的 N 值配置为足够大。时序。

在输入脉冲 TI1 的上升沿到达时，触发 IC1 和 IC2 输入捕获中断，这时把脉冲计数器 TIMx\_CNT 的计数值复位为 0，于是 TIMx\_CNT 的计数值 X 在 TIMxCLK 的驱动下从 0 开始不断累加，直到 TI1 出现下降沿，触发 IC2 捕获事件，此时捕获寄存器 TIMx\_CCR2 把脉冲计数器 TIMx\_CNT 的当前值 2 存储起来，而 TIMx\_CNT 继续累加，直到 TI1 出现第二个上升沿，触发了 IC1 捕获事件，此时 TIMx\_CNT 的当前计数值 4 被保存到 TIMx\_CCR1。

很明显 TIMx\_CCR1(加 1)的值乘以 TIMxCLK 的周期，即为待检测的 PWM 输入脉冲周期，TIMx\_CCR2（加 1）的值乘以 TIMxCLK 的周期，就是待检测的 PWM 输入脉冲的高电平时间，有了这两个数值，就可以计算出 PWM 脉冲的频率、占空比了。

可以看出，正因为捕获/比较寄存器的存在，才使得通用定时器变得如此强大。

## ■ 定时器的时钟源

从时钟源方面来说，通用定时器比基本定时器多了一个选择，它可以使用外部脉冲作为定时器的时钟源，使用外部时钟源时，要使用寄存器进行触发边沿、滤波器带宽的配置。如果选择内部时钟源的话则跟基本定时器的是一样的，也为 TIMxCLK。但要注意的是，所有定时器(包括基本、通用、高级)使用内部时钟时，定时器的时钟源都被称为 TIMxCLK，但 TIMxCLK 的时钟来源并不是完全一样的，见图 14-5。

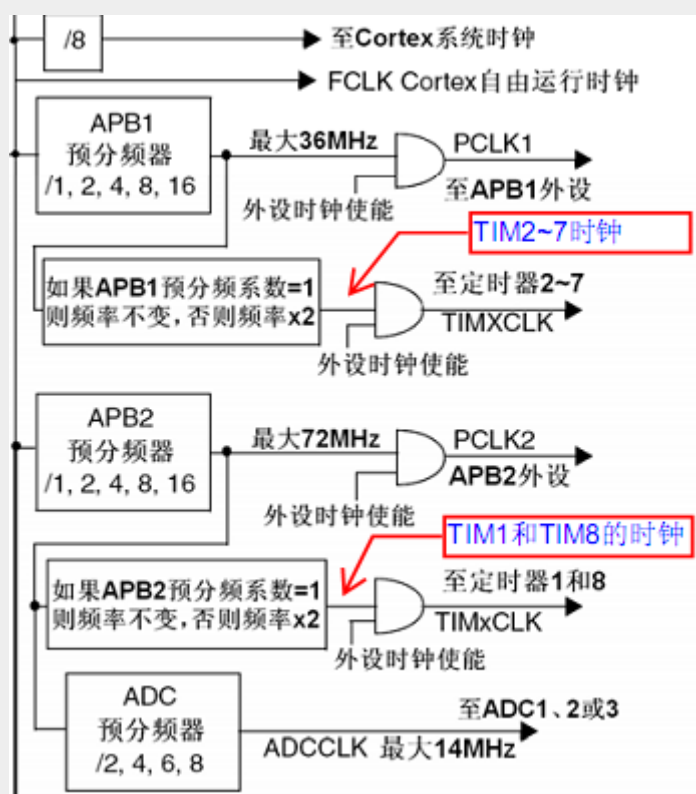


图 14-5 时钟树，TIMxCLK 部分

TIM2~7，也就是基本定时器和通用定时器，TIMxCLK 的时钟来源是 APB1 预分频器的输出，当 APB1 的分频系数为 1 时，则 TIM2~7 的 TIMxCLK 直接等于该 APB1 预分频器的输出，而 APB1 的分频系数不为 1 时，TIM2~7 的 TIMxCLK 则为 APB1 预分频器输出的 2 倍。

如在常见的配置中，AHB=72MHz，而 APB1 预分频器的分频系数被配置为 2，则 PCK1 刚好达到最大值 36MHz，而此时 APB1 的分频系数不为 1，则 TIM2~TIM7 的时钟  $TIMxCLK = (AHB/2) * 2 = 72MHz$ 。

而 TIM1 和 TIM8 这两个高级的定时器，TIMxCLK 的时钟来源则是 APB2 预分频器的输出，同样它 also 根据分频系数分为两种情况。

常见的配置中 AHB=72MHz，APB2 预分频器的分频系数被配置为 1，此时 PCKL2 刚好达到最大值 72MHz，而 TIMxCLK 则直接等于 APB2 分频器的输出，即 TIM1 和 TIM8 的时钟  $TIMxCLK = AHB = 72MHz$ 。

虽然这种配置下最终 TIMxCLK 的时钟频率相等，但必须清楚实质上它们的时钟来源的是有区别的。还要强调的是：TIMxCLK 是定时器内部的时钟源，但

在时钟输出到脉冲计数器 TIMx\_CNT 前，还经过一个预分频器 PSC，最终用于驱动脉冲计数器 TIMx\_CNT 的时钟频率为根据预分频器 PSC 的配置而定。

### 14.2.3. 高级定时器

TIM1 和 TIM8 是两个高级定时器，它们具有基本、通用定时器的所有功能，还具有三相六步电机的接口，刹车功能(break function)及用于 PWM 驱动电路的死区时间控制等，使得它非常适合于电机的控制。

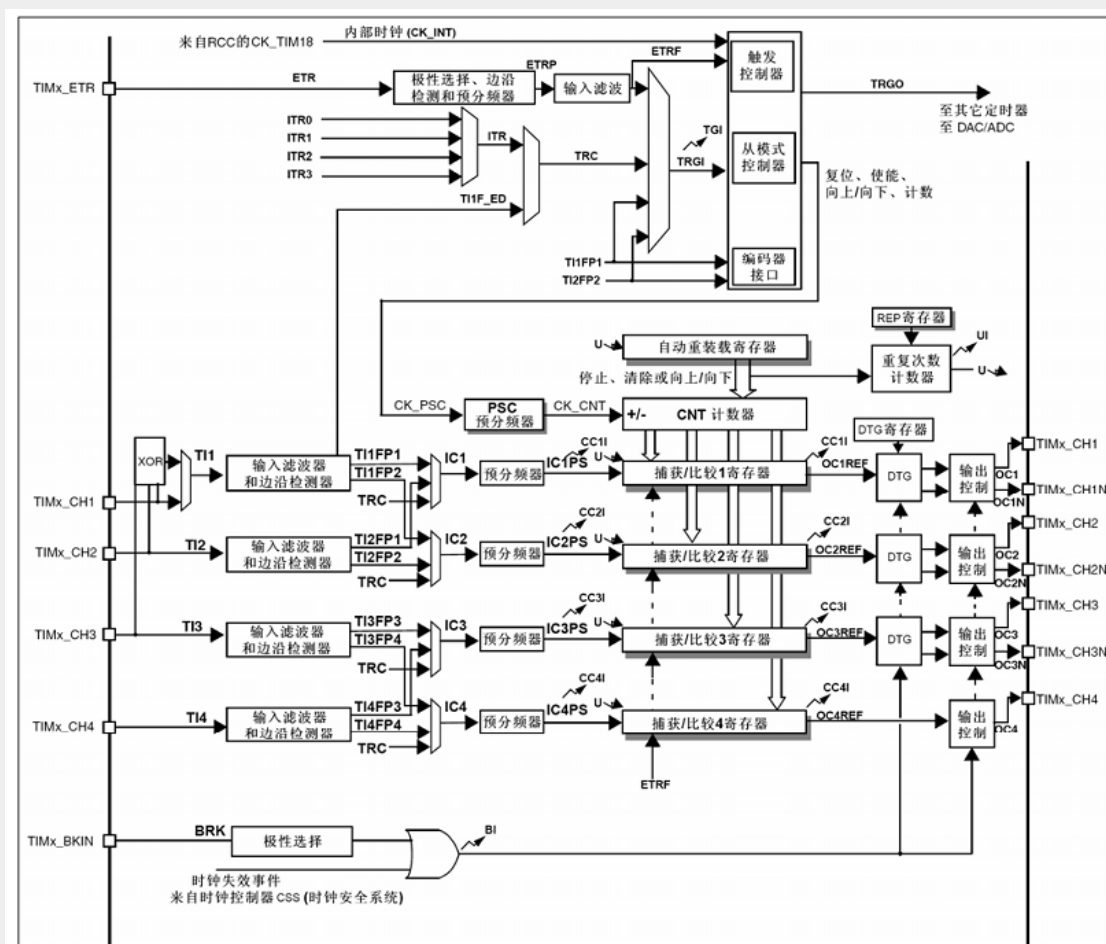


图 14-6 高级定时器结构

#### ■ 死区控制

见图 14-6，相比于通用定时器，主要多出了 BRK、DTG 两个结构，因而具有了死区时间的控制功能。

首先，死区时间是什么呢？在 H 桥，三相桥的 PWM 驱动电路中，上下两个桥臂的 PWM 驱动信号是互补的，即上下桥臂轮流导通，但实际上为了防止出现上下两个臂同时导通(会造成短路)，在上下两臂切换时留一小段时间，上下臂都施加关断信号，这个上下臂都关断的时间称为死区时间。

STM32 的高级定时器可以配置出输出互补的 PWM 信号，并且在这个 PWM 信号中加入死区时间，为电机的控制提供了极大的便利。见图 14-7。图中的 OCxREF 为参考信号(可理解为原信号)，OCx 和 OCxN 为定时器通过 GPIO 引脚输出的 PWM 互补信号。

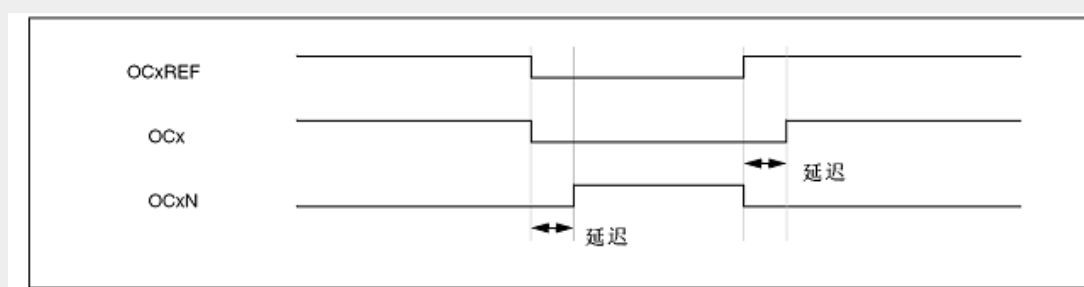


图 14-7 插入死区时间

若不加入死区时间，当 OCxREF 出现下降沿，OCx 同时输出下降沿，OCxN 则同时输出相反的上升沿，即这三个信号的跳变是同时的。

加入死区时间后，当 OCxREF 出现下降沿，OCx 同时输出下降沿，但 OCxN 则过了一小段延迟再输出上升沿，OCxREF 出现上升沿后，OCx 要经过一段延时再输出上升沿。假如 OCx、OCxN 分别控制上、下桥臂，有了延迟后，就不容易出现上、下桥臂同时导通的情况。这个延迟时间是根据 PWM 信号驱动的电子器件特性相关的，从事工控领域的读者对此应该比较熟悉。

在保证不出现短路的情况下，死区时间越短越好。

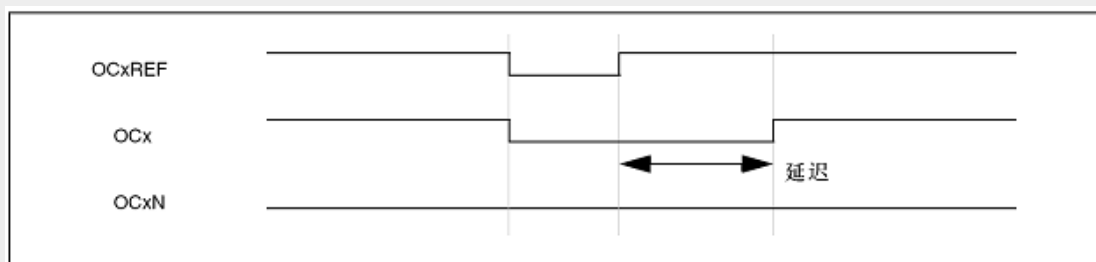


图 14-8 死区时间太长，OCxN 输出不正常

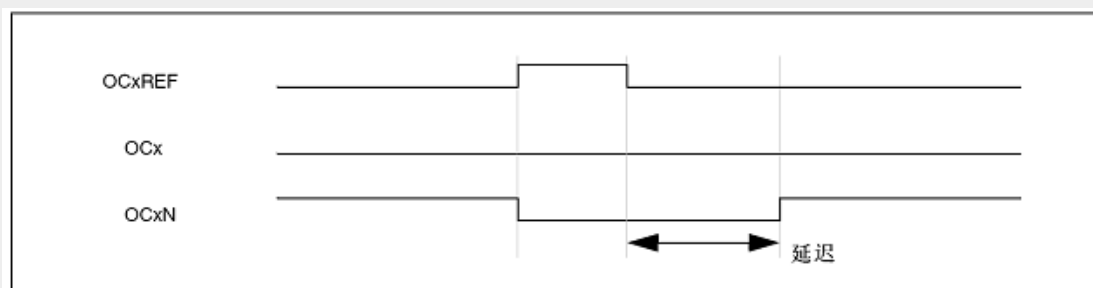


图 14-9 死区时间太长，OCx 输出不正常

### 14.3. PWM 输出实例分析

PWM 在电力电子技术中占据着重要的地位，被广泛地用在逆变电路之中。利用 STM32 定时器的 PWM 输出功能，可以直接获取 PWM 波。根据面积等效原理，利用规则采样法、查表法可以调制出 SPWM 波及各种调制 PWM 波形。

本实验向大家讲解如何输出占空比固定的 PWM 波形。

#### 14.3.1. 实验描述及工程文件清单

##### 实验描述：

通用定时器 TIM3 产生 4 路不同占空比的 PWM 波。

TIM3 Channel1 duty cycle =  $(\text{TIM3\_CCR1} / \text{TIM3\_ARR}) * 100 = 50\%$

TIM3 Channel2 duty cycle =  $(\text{TIM3\_CCR2} / \text{TIM3\_ARR}) * 100 = 37.5\%$

TIM3 Channel3 duty cycle =  $(\text{TIM3\_CCR3} / \text{TIM3\_ARR}) * 100 = 25\%$

TIM3 Channel4 duty cycle =  $(\text{TIM3\_CCR4} / \text{TIM3\_ARR}) * 100 = 12.5\%$

##### 硬件连接：

PA.06: (TIM3\_CH1)

PA.07: (TIM3\_CH2)

PB.00: (TIM3\_CH3)

PB.01: (TIM3\_CH4)

##### 库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_tim.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/pwm\_output.c

### 14.3.2. 配置工程环境

本定时器 PWM 输出实验中我们用到了 GPIO、RCC、TIM 外设，没有使用中断，所以我们要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_tim.c`。

新建 `pwm_output.c` 及 `pwm_output.h` 文件，并在 `stm32f10x_conf.h` 中使用到的 ST 库的头文件注释去掉。见代码清单 14-1。

代码清单 14-1: PWM 例程中的 `stm32f10x_conf.h` 文件配置

```
1.  /**
2.  *****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
8.  *****
9.  #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_tim.h"
```

### 14.3.3. main 文件

我们先从 main 文件看看本实验的执行流程，见代码清单 14-2。

代码清单 14-2: PWM 例程的 main 函数

```
1. #include "stm32f10x.h"
2. #include "pwm_output.h"
3.
4. /*
5.  * 函数名: main
6.  * 描述   : 主函数
7.  * 输入   : 无
8.  * 输出   : 无
9.  */
10. int main(void)
11. {
12.
13.
14.     /* TIM3 PWM 波输出初始化, 并使能 TIM3 PWM 输出 */
15.     TIM3_PWM_Init();
16.
17.     while (1)
18.     {}
19. }
20.
21.
22. /***** (C) COPYRIGHT 2012 WildFire Team *****/
*****/
```

我们看到，代码的执行流程十分简单，调用用户函数 TIM3\_PWM\_Init() 把 TIM 初始化成 PWM 输出模式后，内核就把所有的工作都交给 TIM 外设，完全由 TIM 来控制 GPIO 引脚输出 PWM 波。

### 14.3.4. 定时器初始化

接下来看看 TIM3\_PWM\_Init() 函数是怎样配置 TIM 外设的，它是在 pwm\_output.c 文件中实现的，见代码清单 14-3。

代码清单 14-3: TIM3\_PWM\_Init() 函数

```
1. /*
2.  * 函数名: TIM3_PWM_Init
3.  * 描述   : TIM3 输出 PWM 信号初始化, 只要调用这个函数
4.  *           TIM3 的四个通道就会有 PWM 信号输出
5.  * 输入   : 无
6.  * 输出   : 无
7.  * 调用   : 外部调用
8.  */
```



```
9. void TIM3_PWM_Init(void)
10. {
11.     TIM3_GPIO_Config();
12.     TIM3_Mode_Config();
13. }
```

一看，发现很有野火代码的风格。调用了 TIM3\_GPIO\_Config()对作为 TIM 外设通道复用的 GPIO 引脚进行初始化，再调用 TIM3\_Mode\_Config()对 TIM 外设进行初始化。

## ■ GPIO 初始化

来分析其中 TIM3\_GPIO\_Config()代码，见。

代码清单 14-4: TIM3\_GPIO\_Config()函数

```
1. /*
2.  * 函数名: TIM3_GPIO_Config
3.  * 描述   : 配置 TIM3 复用输出 PWM 时用到的 I/O
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void TIM3_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* TIM3 clock enable */
13.     //PCLK1 经过 2 倍频后作为 TIM3 的时钟源等于 72MHz
14.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
15.
16.     /* GPIOA and GPIOB clock enable */
17.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_
GPIOB, ENABLE);
18.
19.     /*GPIOA Configuration: TIM3 channel 1 and 2 as alternate func
tion push-pull */
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
21.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;    // 复用推挽输
出
22.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
23.
24.     GPIO_Init(GPIOA, &GPIO_InitStructure);
25.
26.     /*GPIOB Configuration: TIM3 channel 3 and 4 as alternate func
tion push-pull */
27.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
28.
29.     GPIO_Init(GPIOB, &GPIO_InitStructure);
30. }
```

在 TIM3\_GPIO\_Config()函数中，我们使能了 TIM3 外设的时钟，并对 TIM3 通道相应的 GPIO 引脚作了相应的配置，使能 GPIO 时钟，分别是 PA6、PA7、PB0 和 PB1。它们被配置为复用输出，翻转速率为 50MHz。

同样，关于 TIM3 通道的 GPIO 引脚映射可以在《STM32 数据手册》的引脚定义表找到，GPIO 复用模式配置可从《STM32 参考手册》的 GPIO 章节找到。见表 14-1 及

表 14-2。

表 14-1 TIM 外设 GPIO 引脚定义(部分)

| 脚位 |   |   |   |   |    | 管脚名称 | I/O 电平 | I/O 模式 | 功能 | 可选的复用功能                                       |           |
|----|---|---|---|---|----|------|--------|--------|----|-----------------------------------------------|-----------|
|    |   |   |   |   |    |      |        |        |    | 默认复用功能                                        | 重映射功能     |
| 3  | 3 | 5 | 2 | 1 | 2  | PA6  | /O     |        | P  | SPI1_MISO<br>/TIM8_BKIN<br>ADC12_IN6/TIM3_CH1 | TIM1_BKIN |
| 3  | 3 | 4 | 3 | 2 | 3  | PA7  | /O     |        | P  | SPI1_MOSI<br>/TIM8_CH1N<br>ADC12_IN7/TIM3_CH2 | TIM1_CH1N |
| 6  | 5 | 5 | 8 | 2 | 36 | PB6  | /O     | T      | P  | I2C1_SCL/TIM4_CH1                             | USART1_TX |
| 6  | 5 | 5 | 9 | 3 | 37 | PB7  | /O     | T      | P  | I2C1_SDA<br>/FSMC_NADV<br>TIM4_CH2            | USART1_RX |

表 14-2 TIM 的 GPIO 模式配置

| TIM2/3/4/5 引脚  | 配置       | GPIO 配置 |
|----------------|----------|---------|
| TIM2/3/4/5_CHx | 输入捕获通道 x | 浮空输入    |
|                | 输出比较通道 x | 推挽复用输出  |
| TIM2/3/4/5_ETR | 外部触发时钟输入 | 浮空输入    |

## ■ 时基初始化

配置好 GPIO 后，再来看看 TIM3\_Mode\_Config()函数是如何配置 TIM 的模式，函数的代码有点多，但主要是因为配置了 4 个通道，见代码清单 14-5。

代码清单 14-5: TIM3\_Mode\_Config()函数

```
1. /*
2.  * 函数名: TIM3_Mode_Config
3.  * 描述   : 配置 TIM3 输出的 PWM 信号的模式，如周期、极性、占空比
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
```

```
static void TIM3_Mode_Config(void)
{
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    /* PWM 信号电平跳变值 */
    u16 CCR1_Val = 500;
    u16 CCR2_Val = 375;
    u16 CCR3_Val = 250;
    u16 CCR4_Val = 125;

    /* -----
    TIM3 Configuration: generate 4 PWM signals with 4 different duty c
cycles:
    TIM3CLK = 72 MHz, Prescaler = 0x0, TIM3 counter clock = 72 MHz
    TIM3 ARR Register = 999 => TIM3 Frequency = TIM3 counter clock/(AR
R + 1)
    TIM3 Frequency = 72 KHz.
    TIM3 Channel1 duty cycle = (TIM3_CCR1/ TIM3_ARR)* 100 = 50%
    TIM3 Channel2 duty cycle = (TIM3_CCR2/ TIM3_ARR)* 100 = 37.5%
    TIM3 Channel3 duty cycle = (TIM3_CCR3/ TIM3_ARR)* 100 = 25%
    TIM3 Channel4 duty cycle = (TIM3_CCR4/ TIM3_ARR)* 100 = 12.5%
    ----- */

    /* Time base configuration */
    TIM_TimeBaseStructure.TIM_Period = 999;
    //当定时器从 0 计数到 999，即为 1000 次，为一个定时周期
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    //设置预分频：不预分频，即为 72MHz
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1 ;
```

```
//设置时钟分频系数：不分频
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
//向上计数模式

TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

/* PWM1 Mode configuration: Channel1 */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
//配置为 PWM 模式 1
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
//设置跳变值，当计数器计数到这个值时，电平发生跳变
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
//当定时器计数值小于 CCR1_Val 时为高电平

TIM_OC1Init(TIM3, &TIM_OCInitStructure);    //使能通道 1

TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);

/* PWM1 Mode configuration: Channel2 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR2_Val;
//设置通道 2 的电平跳变值，输出另外一个占空比的 PWM

TIM_OC2Init(TIM3, &TIM_OCInitStructure);    //使能通道 2

TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);

/* PWM1 Mode configuration: Channel3 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR3_Val;
//设置通道 3 的电平跳变值，输出另外一个占空比的 PWM

TIM_OC3Init(TIM3, &TIM_OCInitStructure);    //使能通道 3

TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);

/* PWM1 Mode configuration: Channel4 */
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = CCR4_Val;
//设置通道 4 的电平跳变值，输出另外一个占空比的 PWM

TIM_OC4Init(TIM3, &TIM_OCInitStructure);    //使能通道 4

TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable);

TIM_ARRPreloadConfig(TIM3, ENABLE);
// 使能 TIM3 重载寄存器 ARR

/* TIM3 enable counter */
TIM_Cmd(TIM3, ENABLE);    //使能定时器 3
}
```

TIM3\_Mode\_Config()函数被分为三个部分，分别为时基初始化、输出模式初始化和装载捕获/比较寄存器的数值。

首先是时基初始化，使用了一个 TIM\_TimeBaseInitTypeDef 类型的结构体。时基初始化，即配置基本定时器只具有的那部分功能，下面分析初始化结构体的成员：

### (1) .TIM\_Period

定时周期，实质是存储到重载寄存器  $TIMx\_ARR$  的数值，脉冲计数器从 0 累加到这个值上溢或从这个值自减至 0 下溢。这个数值加 1 然后乘以时钟源周期就是实际定时周期。

本实验中向该成员赋值为 999，即定时周期为  $(999+1)*T$ ， $T$  为时钟源周期。

### (2) .TIM\_Prescaler

对定时器时钟  $TIMxCLK$  的预分频值，分频后作为脉冲计数器  $TIMx\_CNT$  的驱动时钟，得到脉冲计数器的时钟频率为： $f_{CK\_CNT}=f_{TIMx\_CLK}/(N+1)$ ，其中  $N$  即为赋给本成员的时钟分频值。

本实验给  $TIM\_Prescaler$  成员赋值为 0，即不对  $TIMxCLK$  分频，已知 AHB 时钟频率为 72MHz、 $TIMxCLK$  为 72MHz，所以输出到脉冲计数器  $TIMx\_CNT$  的时钟频率为  $f_{CK\_CNT}=72MHz/1=72MHz$ 。

### (3) .TIM\_ClockDivision

时钟分频因子。怎么又出现一个配置时钟分频的呢？要注意这个  $TIM\_ClockDivision$  和上面的  $TIM\_Prescaler$  是不一样的。 $TIM\_Prescaler$  预分频配置是对  $TIMxCLK$  进行分频，分频后的时钟被输出到脉冲计数器  $TIMx\_CNT$ ，而  $TIM\_ClockDivision$  虽然也是对  $TIMxCLK$  进行分频，但它分频后的时钟频率为  $f_{DTS}$ ，是被输出到定时器的 ETRP 数字滤波器部分，会影响滤波器的采样频率。 $TIM\_ClockDivision$  可以被配置为 1 分频（ $f_{DTS}=f_{TIMxCLK}$ ）、2 分频及 4 分频。ETRP 数字滤波器的作用是对外部时钟  $TIMxETR$  进行滤波。

本实验中是使用内部时钟  $TIMxCLK$  作为定时器时钟源的，所以配置  $TIM\_ClockDivision$  为任何数值都没有影响。

### (4) .TIM\_CounterMode

本成员配置的为脉冲计数器  $TIMx\_CNT$  的计数模式，分别为向上计数，向下计数，及中央对齐模式。向上计数即  $TIMxCNT$  从 0 向上累加到  $TIM\_Period$  中的值，（重载寄存器  $TIMx\_ARR$  的值），产生上溢事件；向下计数则  $TIMxCNT$  从  $TIM\_Period$  的值累减至 0，产生下溢事件。而中央对齐模式则为

向上、向下计数的合体，TIMxCNT 从 0 累加到 TIM\_Period 的值减 1 时，产生一个上溢事件，然后向下计数到 1 时，产生一个计数器下溢事件，再从 0 开始重新计数。

本实验中 TIM\_CounterMode 成员被赋值为 TIM\_CounterMode\_Up(向上计数模式)。

填充完配置参数后，调用库函数 TIM\_TimeBaseInit()把这些控制参数写到寄存器中，定时器的时基配置就完成了。

## ■ 输出模式配置

通用定时器的输出模式由 TIM\_OCInitTypeDef 类型结构体的以下几个成员来配置，若是使用 TIM1 或 TIM8 还有其它的成员，本实验不介绍。

### (1) .TIM\_OCMode

输出模式配置，主要使用的为 PWM1 和 PWM2 模式。

PWM1 模式是：在向上计数时，当 TIMx\_CNT < TIMx\_CCRn (比较寄存器，其数值等于 TIM\_Pulse 成员的内容) 时，通道 n 输出为有效电平，否则为无效电平；在向下计数时，当 TIMx\_CNT > TIMx\_CCRn 时通道 n 为无效电平，否则为有效电平。PWM2 模式跟 PWM1 模式相反。

其中的有效电平和无效电平并不是固定地对应高电平和低电平，也是需要配置的，由下面介绍的 TIM\_OCPolarity 成员配置。

本实验中使用 PWM1 输出模式。

### (2) .TIM\_OutputState

配置输出模式的状态使能或关闭输出。

本实验中向该成员赋值为 TIM\_OutputState\_Enable(使能输出)。

### (3) .TIM\_OCPolarity

有效电平的极性，把 PWM 模式中的有效电平设置为高电平或低电平。

本实验中向该成员赋值为 `TIM_OCPolarity_High`(有效电平为高电平), 因为在上面把输出模式配置为 `PWM1` 模式, 向上计数, 所以在 `TIMx_CNT < TIMx_CCRn` 时, 通道 `n` 输出为高电平, 否则为低电平。

#### (4) .TIM\_Pulse

直译为跳动, 本成员的参数值即为比较寄存器 `TIMx_CCR` 的数值, 当脉冲计数器 `TIMx_CNT` 与 `TIMx_CCR` 的比较结果发生变化时, 输出脉冲将发生跳变。

本实验中向 1、2、3、4 通道的该成员分别赋值为 500、375、250、125。而定时器向上计数、`PWM1` 模式、有效电平为高, 定时周期为 1000(`.TIM_Period=999`), 所以当 `TIMx_CNT` 计时值小于 `.TIM_Pulse` 值时, 输出高电平, 否则为低电平, 即各通道输出 PWM 的占空比为  $D = \text{TIM\_Pulse} / (\text{TIM\_Period} + 1)$ , 即分别为 50%、37.5%、25%、12.5%。

(注: 上面占空比的计算公式仅针对本实验的配置。)

填充完输出模式初始化结构体后, 要调用输出模式初始化函数 `TIM_OCxInit()` 对各个通道进行初始化, 其中 `x` 表示定时器的通道。如 `TIM_OC1Init()` 是用来初始化定时器的通道 1 的, `TIM_OC2Init()` 是用来初始化定时器的通道 2 的, 在调用各个通道的初始化函数前, 需要对初始化结构体的 `.TIM_Pulse` 成员重新赋值, 这是因为本实验中其它成员的配置都是一样, 而占空比不同。

实验中还用 `TIM_OCxPreloadConfig()` 配置了各通道的比较寄存器 `TIM_CCR` 预装载使能; 使用 `TIM_ARRPreloadConfig()` 把重载寄存器 `TIMx_ARR` 使能, 最后用 `TIM_Cmd()` 使能定时器 `TIM3`, 定时器外设就开始工作了。

由于定时器的需要配置的参数较多, 下面为大家总结一下:

- (1) 设定 TIM 信号周期
- (2) 设定 TIM 预分频值(`.TIM_Prescaler`)
- (3) 设定 TIM 分频系数(`TIM_ClockDivision`)
- (4) 设定 TIM 计数模式

- (5) 根据 TIM\_TimeBaseInitStruct 这个结构体里面的值初始化 TIM
- (6) 设定 TIM 的 OC 模式
- (7) TIM 输出使能
- (8) 设定电平跳变值
- (9) 设定 PWM 信号的极性
- (10) 使能 TIM 信号通道
- (11) 使能 TIM 比较寄存器 CCRX 重载
- (12) 使能 TIM 重载寄存器 ARR
- (13) 使能 TIM 计数器

本实验中设置输出的为占空比固定的 PWM 波，若想利用定时器输出 SPWM 波，则要不断改变 PWM 的占空比，我们可以利用库函数 TIM\_SetCompare()查表修改比较寄存器中的值(即脉冲宽)，来达到目的。

#### 14.3.5. 实验现象

现在，TIM3 的通道 1(PA.06)、2(PA.07)、3(PB.00)、4(PB.01)就会输出不同占空比的 PWM 信号了。PWM 信号可以通过示波器看到。考虑到并不是每个用户手头上都有示波器，我们在这里采用软件仿真的方式来验证我们的程序。

以前我们都是通过 JLINK 直接将我们的代码烧到开发板的 flash 中去调试，现在要换成软件仿真，得首先设置下我们的开发环境，按照如下步骤所示：

- (1) 点击 Target Options...选项。



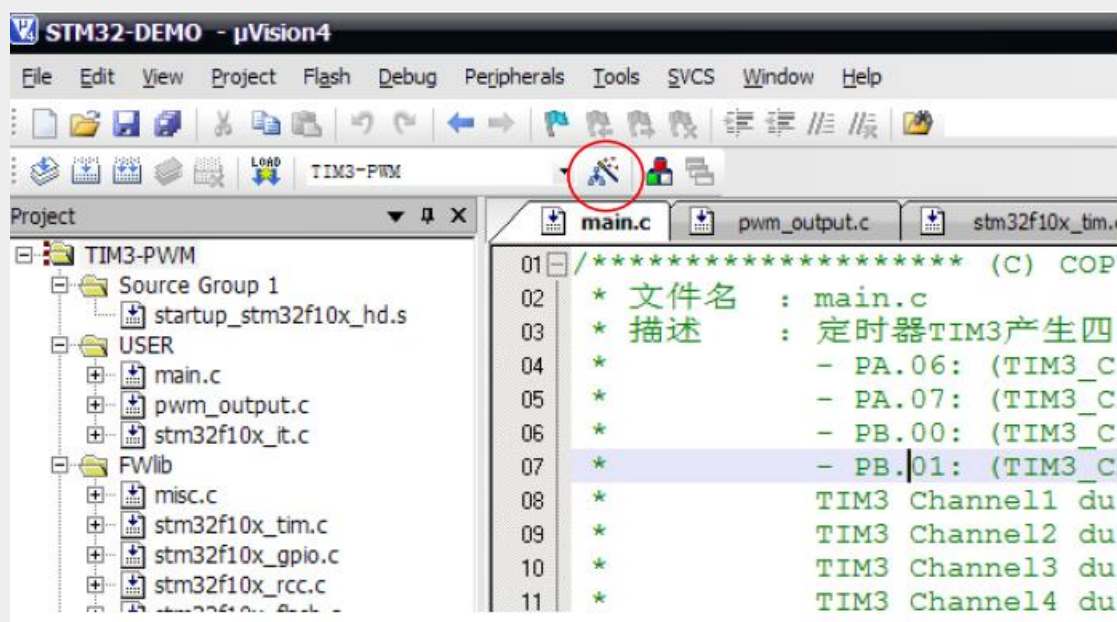


图 14-10 打开 Target Option 设置

(2) 选中 Debug 选项卡。

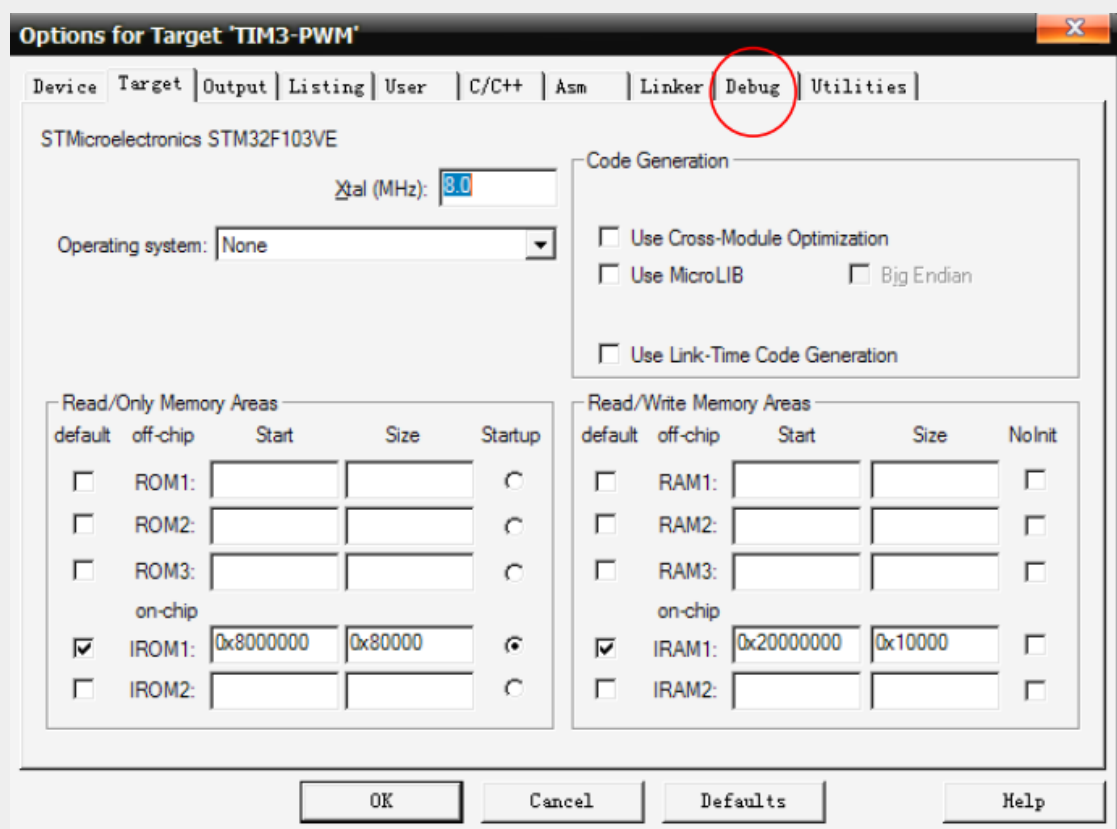


图 14-11 debug 选项卡

(3) 选中 Use Simulator 选项，然后点击 OK 即可。

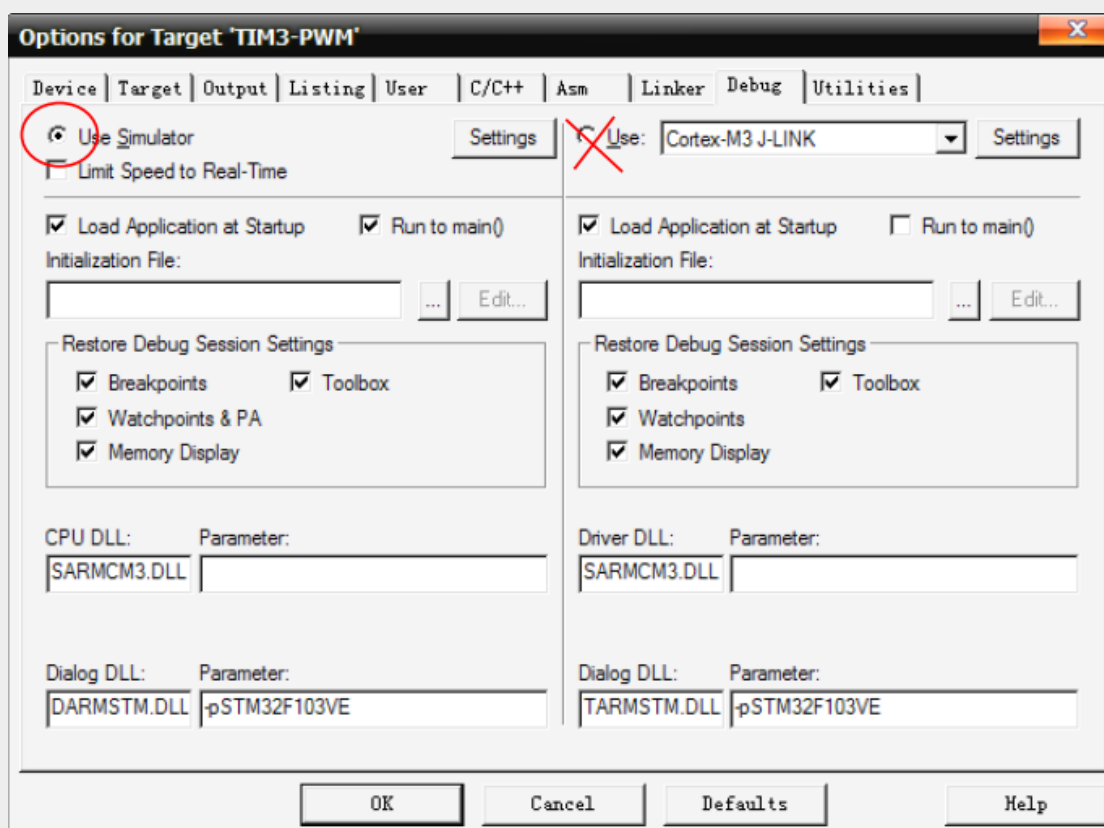


图 14-12 选择软件仿真

下面我们开始进行软件仿真，按照如下步骤进行：

- (1) 点击 Start/Stop Debug Session 选项。

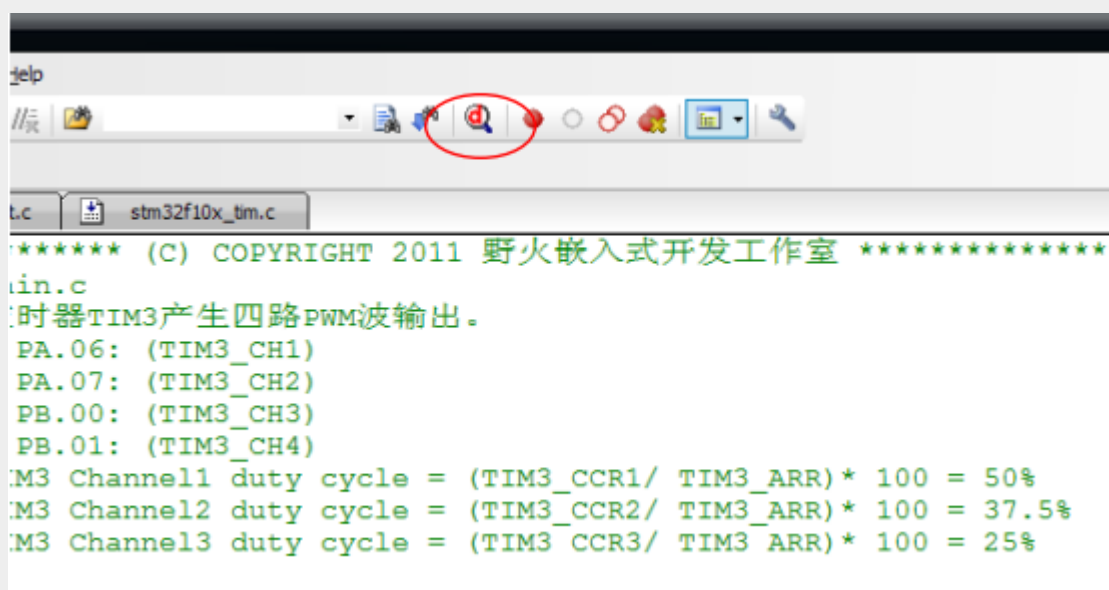


图 14-13 点击 Start debug session

(2) 点击 Analysis Windows 选项。

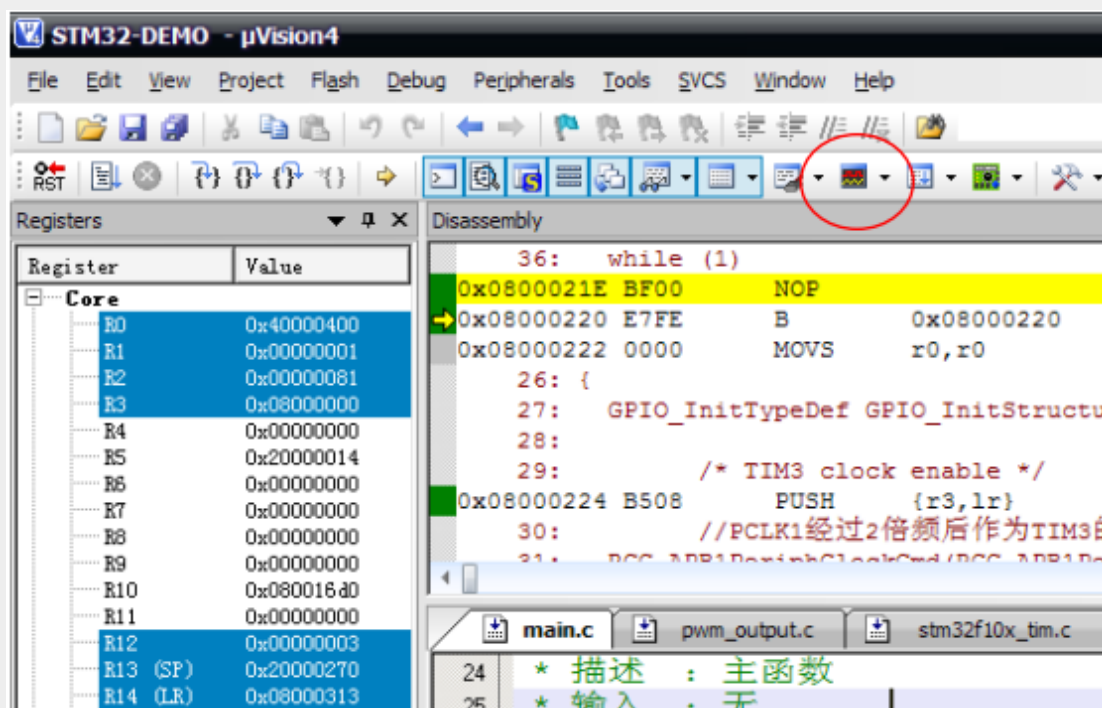


图 14-14 点击 Analysis Windows

(3) 点击 Setup... 选项卡。

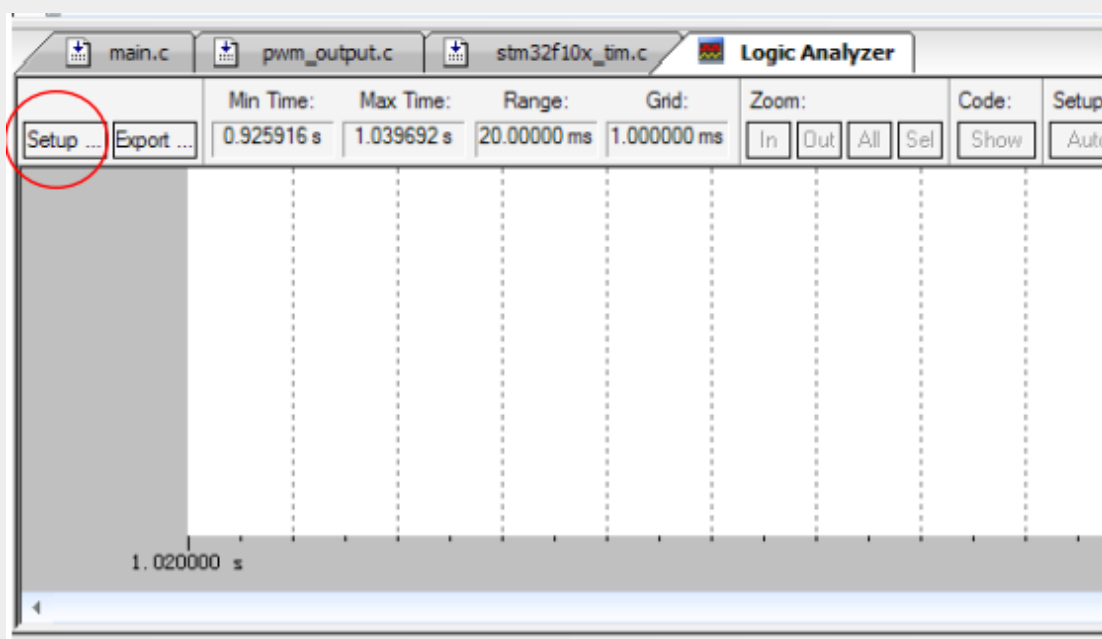


图 14-15 点击 Setup 选项

(4) 点击 NEW(Insert), 在下面的文本框中输入 TMI3 的 PWM 通道, 这里分别是: PORTA.6、PORTA.7、PORTB.0、PORTB.1。然后点击 Close。

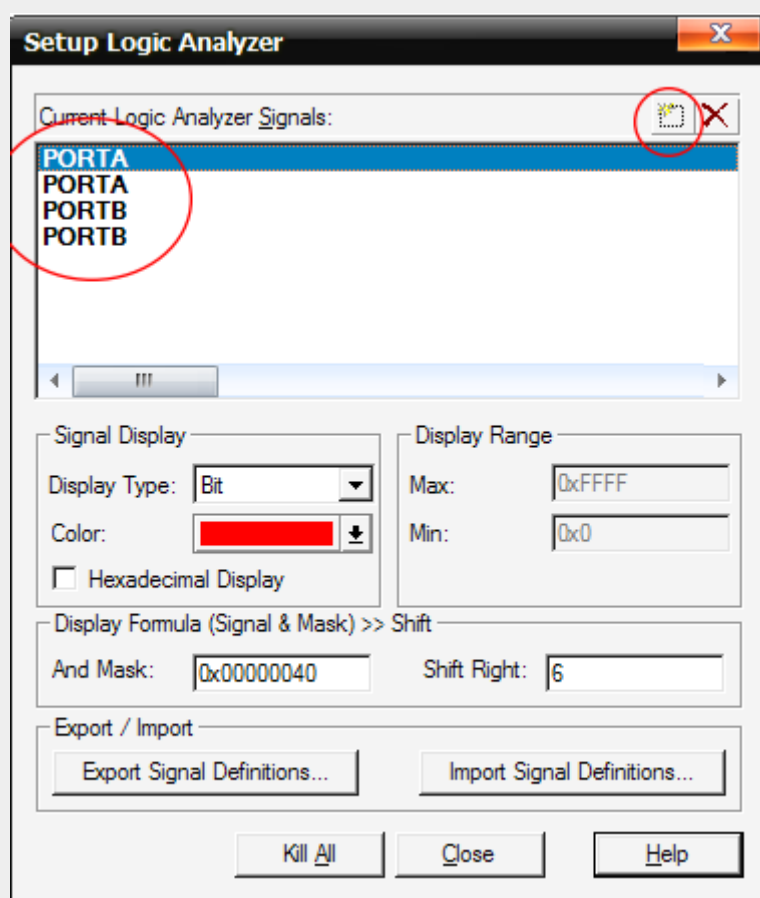


图 14-16 输入要查看的引脚

(5) 点击运行。

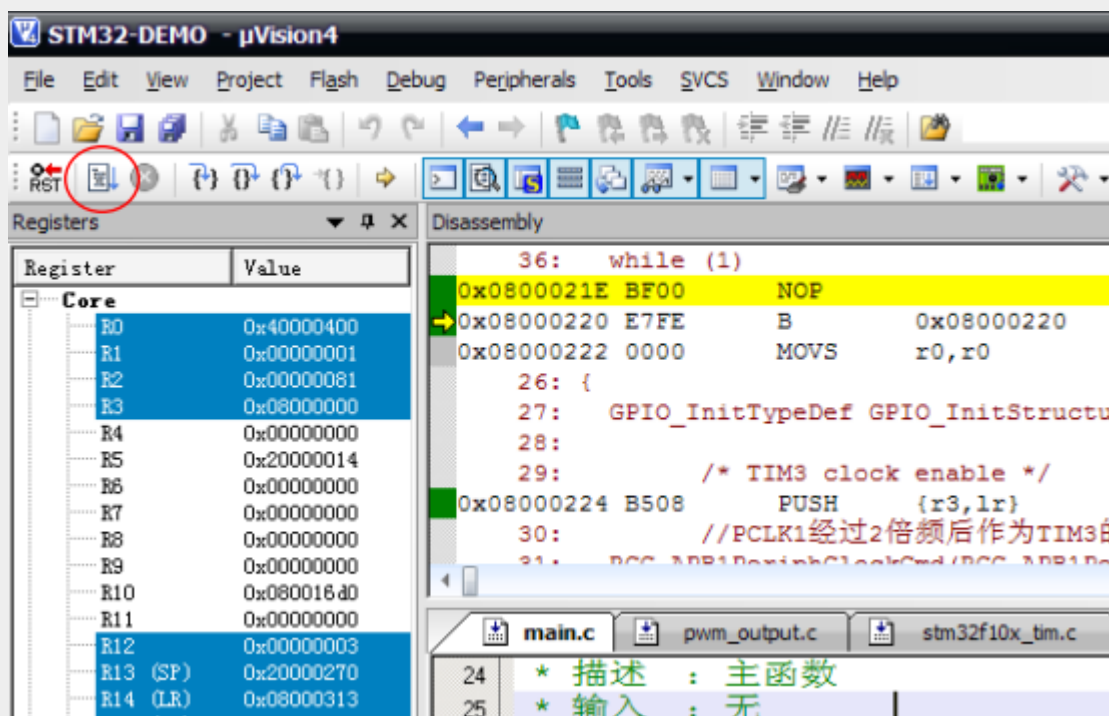


图 14-17 全速运行

(6) 这时候正常的话则是出现下面的 PWM 信号，若没有出现逻辑分析仪界面可点击工具栏中的  按钮。

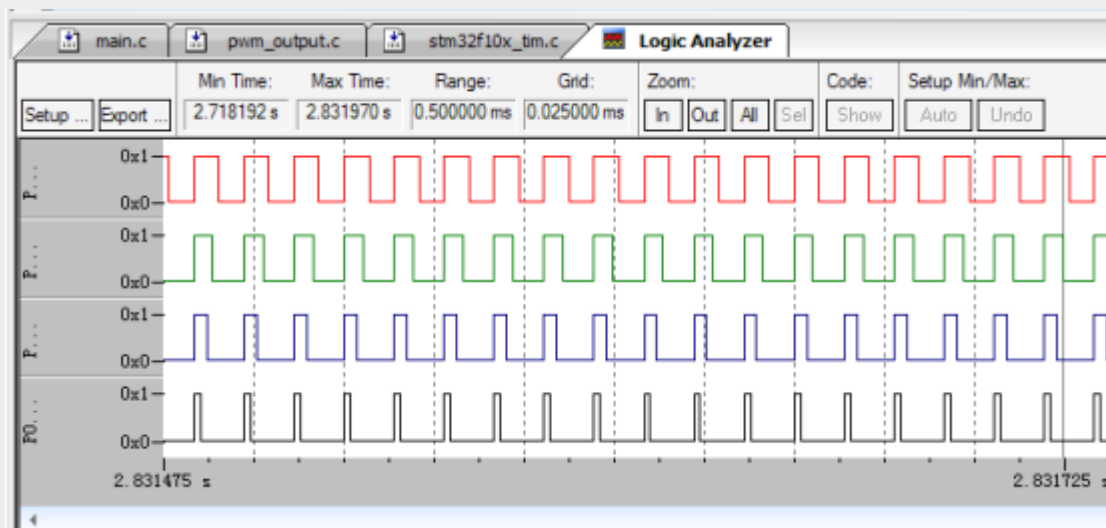


图 14-18 PWM 实验现象

不正常的话则出现下面的情况，一团糟，根本看不到 PWM 信号。这时我们不免会有些抓狂呀，但请大家放心，看看接下来我们是怎么解决的吧。

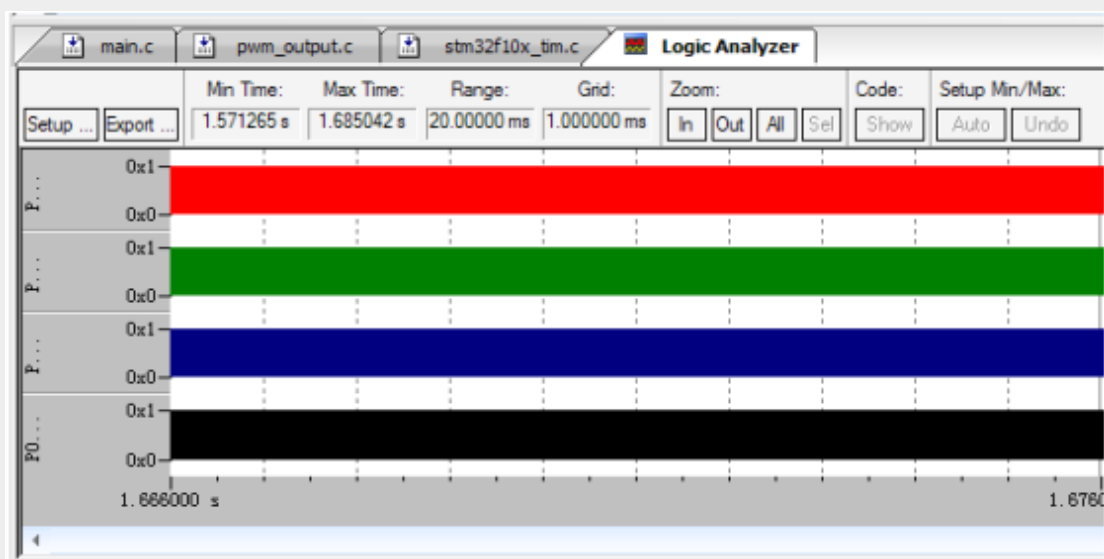


图 14-19 没有调整好时间轴的现象

(7) 其实出现上面的情况是因为我们显示 PWM 信号时没有放大的缘故，我们可以点击 **In** 这个按钮来将 PWM 信号显示的大点，如下所示，一切搞定。

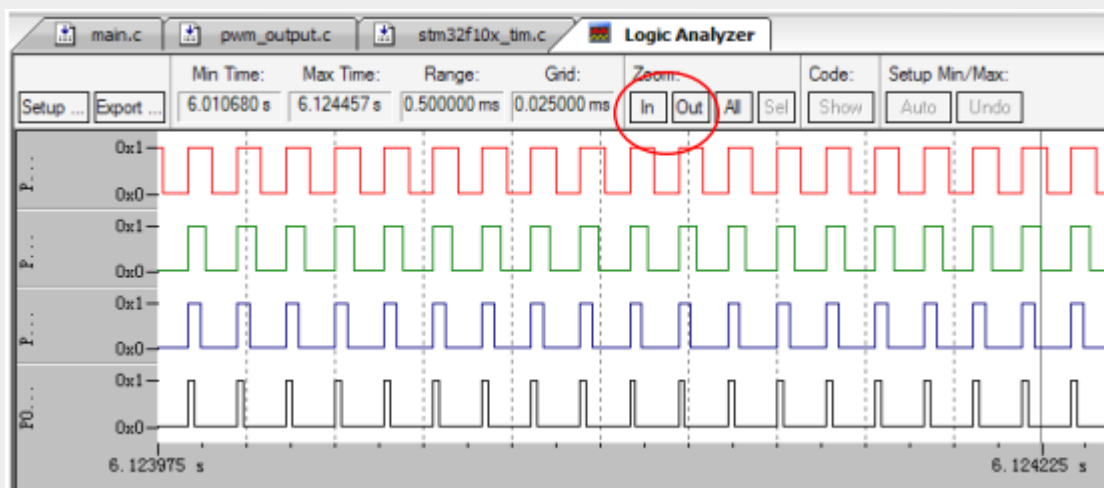


图 14-20 设置 Zoon IN OUT 调整时间轴

(8) 我们可以点击停止按钮，让 PWM 信号静止显示。

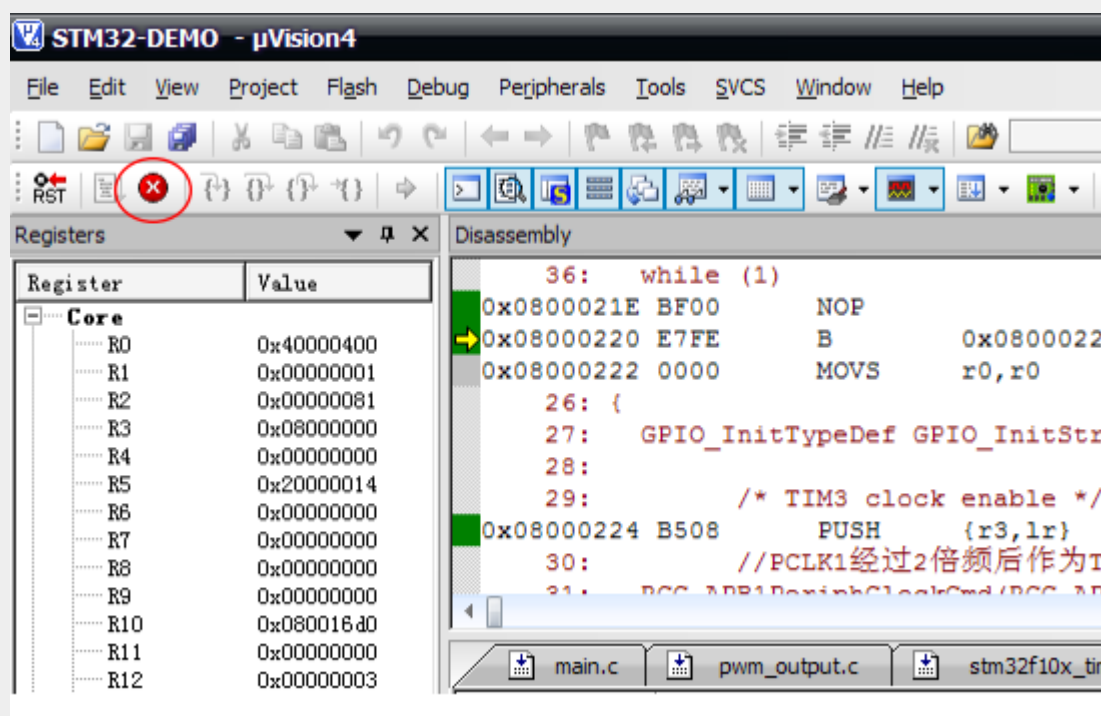


图 14-21 停止运行

(9) 多选分析仪界面中的 Cursor 选项，出现时间标签，可以用来测量周期，频率。

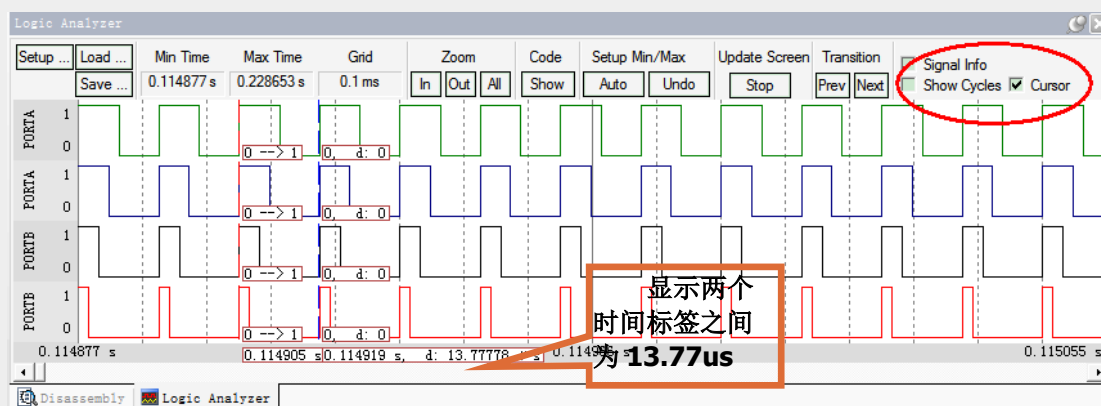


图 14-22 查看周期

我们可以看到，这样测量出来的一个脉冲周期约为 13.77us，与我们配置的 13.88us 一致，(误差是因为为时间标签的位置放得不够准确)

(10) 仿真完毕之后，点击 Start/Stop Debug Session 选项就可以回到正常的代码编辑模式。



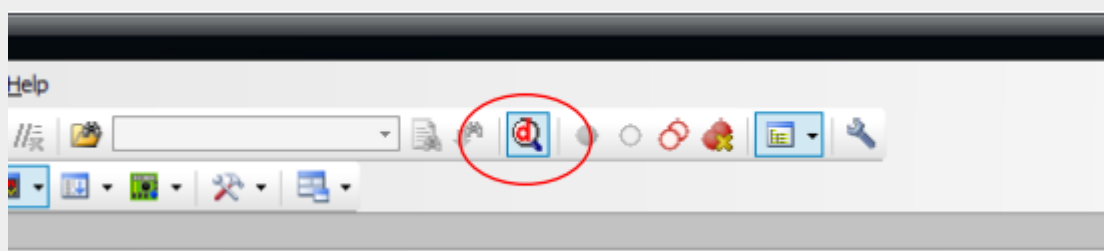


图 14-23 回到代码编辑模式

(11) 要是您有示波器的话，看到的效果则更真实、准确。

## 15. 呼吸灯



## 16. RTC 实时时钟

### 16.1. RTC 实时时钟简介

STM32 的 RTC 外设，实质是一个掉电后还继续运行的定时器。从定时器的角度来说，相对于通用定时器 TIM 外设，它十分简单，只有很纯粹的计时功能(当然，可以触发中断)；但从掉电还继续运行的角度来说，它却是 STM32 中唯一一个具有如此强大功能的外设。所以 RTC 外设的复杂之处并不在于它的定时功能，而在于它掉电还继续运行的特性。

以上所说的掉电，是指主电源  $V_{DD}$  断开的情况，为了 RTC 外设掉电继续运行，必须给 STM32 芯片通过  $V_{BAT}$  引脚接上锂电池。当主电源  $V_{DD}$  有效时，由  $V_{DD}$  给 RTC 外设供电。当  $V_{DD}$  掉电后，由  $V_{BAT}$  给 RTC 外设供电。但无论由什么电源供电，RTC 中的数据都保存在属于 RTC 的备份域中，若主电源  $V_{DD}$  和  $V_{BAT}$  都掉电，那么备份域中保存的所有数据将丢失。备份域除了 RTC 模块的寄存器，还有 42 个 16 位的寄存器可以在  $V_{DD}$  掉电的情况下保存用户程序的数据，系统复位或电源复位时，这些数据也不会被复位。

从 RTC 的定时器特性来说，它是一个 32 位的计数器，只能向上计数。它使用的时钟源有三种，分别为高速外部时钟的 128 分频：HSE/128；低速内部时钟 LSI；使 HSE 分频时钟或 LSI 的话，在主电源  $V_{DD}$  掉电的情况下，这两个时钟来源都会受到影响，因此没法保证 RTC 正常工作。因此 RTC 一般使用 低速外部时钟 LSE，频率为实时时钟模块中常用的 32.768KHz，这是因为  $32768 = 2^{15}$ ，分频容易实现，所以它被广泛应用到 RTC 模块。在主电源  $V_{DD}$  有效的情况下(待机)，RTC 还可以配置闹钟事件使 STM32 退出待机模式。

## 16.2. RTC 工作过程分析

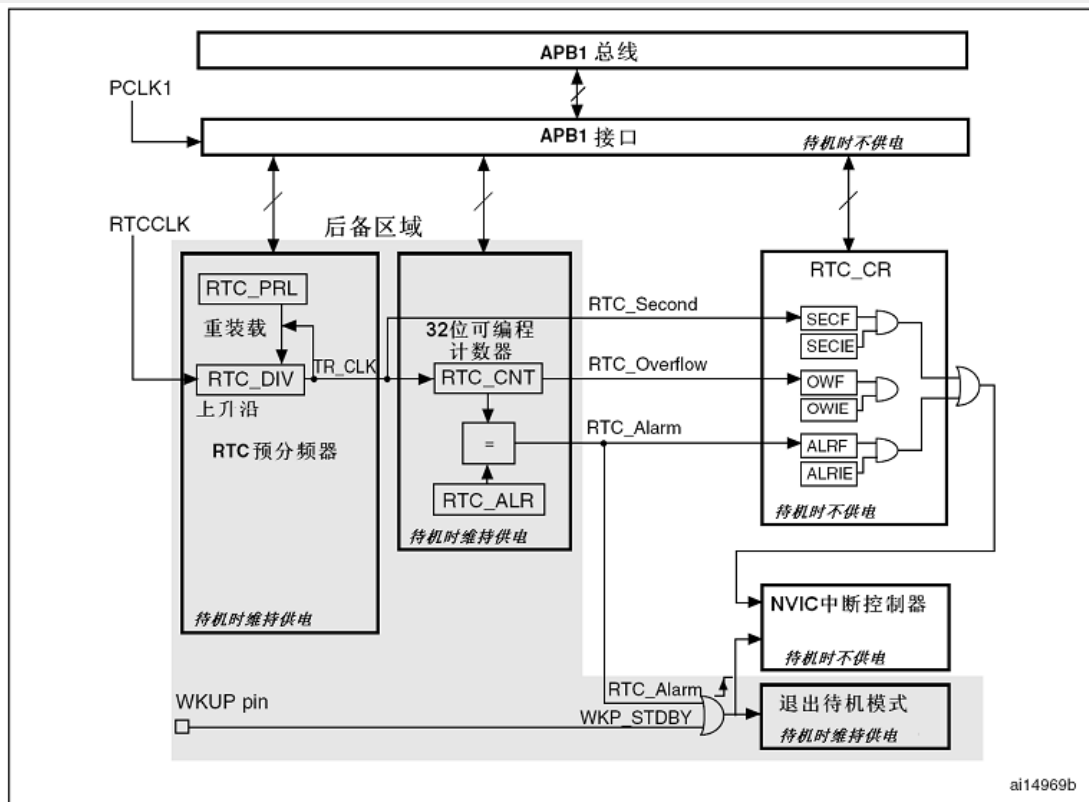


图 16-1 RTC 架构图

### 16.2.1. RTC 架构

图中浅灰色的部分都是属于备份域的，在  $V_{DD}$  掉电时可在  $V_{BAT}$  的驱动下继续运行。这部分仅包括 RTC 的分频器，计数器，和闹钟控制器。若  $V_{DD}$  电源有效，RTC 可以触发 `RTC_Second`(秒中断)、`RTC_Overflow`(溢出事件)和 `RTC_Alarm`(闹钟中断)。从结构图可以分析到，其中的定时器溢出事件无法被配置为中断。若 STM32 原本处于待机状态，可由闹钟事件或 `WKUP` 事件(外部唤醒事件，属于 `EXTI` 模块，不属于 RTC)使它退出待机模式。闹钟事件是在计数器 RTC CNT 的值等于闹钟寄存器 RTC ALR 的值时触发的。

由于 RTC 的寄存器是属于备份域，所以它的所有寄存器都是 16 位的。它的计数器 RTC\_CNT 的 32 位由 RTC\_CNTL 和 RTC\_CNTH 两个寄存器组成，分别保存计数值的低 16 位和高 16 位。我们配置 RTC 模块的时钟时，把输入的 32768Hz 的 RTCCLK 进行 32768 分频得到实际驱动计数器的时钟 TR\_CLK =

$RTCCLK/32768 = 1\text{ Hz}$ ，计时周期为 1 秒，计时器在 TR\_CLK 的驱动下计数，即每秒计数器 RTC\_CNT 的值加 1。

由于备份域的存在，使得 RTC 核具有了完全独立于 APB1 接口的特性，也因此对 RTC 寄存器的访问要遵守一定的规则。

系统复位后，禁止访问后备寄存器和 RTC，防止对后备区域(BKP)的意外写操作。执行以下操作使能对后备寄存器和 RTC 的访问：

(1) 设置 RCC\_APB1ENR 寄存器的 PWREN 和 BKPEN 位来使能电源和后备接口时钟。

(2) 设置 PWR\_CR 寄存器的 DBP 位使能对后备寄存器和 RTC 的访问。

设置为可访问后，在第一次通过 APB1 接口访问 RTC 时，必须等待 APB1 与 RTC 外设同步，确保被读取出来的 RTC 寄存器值是正确的。若在同步之后，一直没有关闭 APB1 的 RTC 外设接口，就不需要再次同步了。

如果内核要对 RTC 寄存器进行任何的写操作，在内核发出写指令后，RTC 模块在 3 个 RTCCLK 时钟之后，才开始正式的写 RTC 寄存器操作。我们知道 RTCCLK 的频率比内核主频低得多，所以必须要检查 RTC 关闭操作标志位 RTOFF，当这个标志被置 1 时，写操作才正式完成。

当然，以上的操作都具有库函数，读者不必具体地查阅寄存器。

### 16.2.2. UNIX 时间戳

如果我们从现在起，把计数器 RTC\_CNT 的计数值置 0，然后每秒加 1，RTC\_CNT 什么时候会溢出呢？RTC\_CNT 是 32 位寄存器，可存储的最大值为  $(2^{32}-1)$ ，即这样计时的话，在  $2^{32}$  秒后溢出。它是多少年呢？

$$N = 2^{32}/365/24/60/60 \approx 136 \text{ 年}$$

假如某个时刻我们读取到计数器的数值为  $X = 60*60*24*2$ ，而我们又知道计数器是在 2011 年 1 月 1 日的 0 时 0 分 0 秒置 0 的，那么我们就可以根据计数器的这个相对时间数值，计算得这个时刻是 2011 年 1 月 3 日的 0 时 0 分 0

秒了。而计数器则会在(2011+136)年左右溢出，穿越回到 2011 年。那时如果我们还在使用这个计数器提供时间的话就会出现时间问题。定时器被置 0 的这个时间被称为计时元年，相对计时元年经过的秒数称为时间戳，即为本例子中计数器的值。

大多数操作系统都是利用时间戳和计时元年来计算当前时间的，而这个时间戳和计时元年大家都取了同一个标准——UNIX 时间戳和 UNIX 计时元年。UNIX 计时元年被设置为格林威治时间 1970 年 1 月 1 日 0 时 0 分 0 秒，大概是为了纪念 UNIX 的诞生吧。而 UNIX 时间戳即为当前时间相对于 UNIX 计时元年经过的秒数。在这个计时系统中，使用的是有符号的 32 位整型变量来保存 UNIX 时间戳的，即实际可用计数位数比我们上面例子中的少了一位，少了这一位，UNIX 计时元年也相对提前了，这个计时方法在 2038 年 1 月 19 日 03 时 14 分 07 秒将会发生溢出。这个时间离我们并不远，UNIX 时间戳被广泛应用到各种系统中，溢出可能会导致系统发生严重错误，差不多到这个时候，记得提醒大家注意这个问题呀。

### 16.3. 利用 RTC 提供北京时间实验

从上面的分析，我们知道，RTC 外设这个连续计数的计数器，在相应软件配置下，可提供时钟日历的功能，修改计数器的值则可以重新设置系统当前的时间和日期。而由于它的时钟配置系统(RCC\_BDCR 寄存器)是在备份域，在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变，利用它，可以实现实时时钟的功能。

#### 16.3.1. 实验描述及工程文件清单

##### 实验描述：

利用 STM32 的 RTC 实现一个电子时钟。在超级终端显示 UNIX 时间戳，公历，农历，北京时间。

##### 硬件连接：

V<sub>BAT</sub> 引脚需外接电池。

### 库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_pwr.c

FWlib/stm32f10x\_bkp.c

FWlib/stm32f10x\_rtc.c

FWlib/stm32f10x\_misc.c

### 用户文件：

USER/main.c

USER/stm32f10x\_it.c

USER/calendar.c (农历算法)

USER/date.c (公历算法)

USER/usart.c

USER/rtc.c

## 16.3.2. 配置工程环境

本 RTC 实时时钟实验中我们用到了 GPIO、RCC、USART、BKP(备份域)、PWR(电源管理)及 RTC 外设，所以我们要把以下库文件添加到工程 stm32f10x\_gpio.c、stm32f10x\_rcc.c stm32f10x\_usart.c、stm32f10x\_bkp.c 、stm32f10x\_pwr.c、stm32f10x\_rtc.c。由于本实验中，我们使用了 RTC 的秒中



断，使开发板每秒向 PC 终端更新一次时间。所以还需要添加 `misc.c` 文件。但实际上 RTC 的计时并不需要使用中断。

接下来添加旧工程中的外设用户文件 `usart.c`，加入开源的万年历算法文件 `calendar.c` 和 `date.c` (野火修改了部分代码，使之符合我们常用的北京时间)，新建 `rtc.c` 及 `rtc.h` 文件，并在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。见代码清单 16-1。

代码清单 16-1: RTC 例程中的 `stm32f10x_conf.h` 文件配置

```
1.  /**
2.     ****
3.     * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.     * @author    MCD Application Team
5.     * @version    V3.5.0
6.     * @date      08-April-2011
7.     * @brief     Library configuration file.
8.     ****
9.     #include "stm32f10x_bkp.h"
10.    #include "stm32f10x_gpio.h"
11.    #include "stm32f10x_pwr.h"
12.    #include "stm32f10x_rcc.h"
13.    #include "stm32f10x_rtc.h"
14.    #include "stm32f10x_usart.h"
15.    #include "misc.h"
```

### 16.3.3. main 文件

配置好库的环境之后，我们就从 `main` 函数开始分析，为了给大家分析流程更方便，野火把部分凌乱的代码重新封装好，让大家更加清晰实验的流程，实际应用时大家可根据需求来改，见代码清单 16-2。

代码清单 16-2: RTC 例程中的 `main` 函数

```
1.  #include "stm32f10x.h"
2.  #include "usart1.h"
3.  #include "rtc.h"
4.
5.
6.  /*时间结构体*/
7.  struct rtc_time systmtime;
8.
9.
10. /**
11.     * @brief    Main program.
12.     * @param    None
13.     * @retval    None
14.     */
```

```
15. int main()
16. {
17.     USART1_Config();
18.     /* 配置 RTC 秒中断优先级 */
19.     NVIC_Configuration();
20.
21.     RTC_CheckAndConfig(&systmtime);
22.
23.
24.     /* Display time in infinite loop */
25.     Time_Show(&systmtime);
26. }
```

## ■ 实验代码执行流程

因为我们在实验中需要用到串口，所以我们调用 `USART1_Config()` 函数将串口配置好。

`NVIC_Configuration()` 函数用于配置 RTC 的秒中断优先级，我们将它的主优先级设置为 1，次优先级为 0。这里只用到了 RTC 一个中断，所以 RTC 的主和次优先级不必太关心。

紧接着调用用户函数 `RTC_CheckAndConfig()` 来查看 RTC 外设是否在本次  $V_{DD}$  上电之前被配置过了，若之前没有被配置过，则要求用户输入当前时间，重新初始化 RTC 和配置时间。

RTC 被配置好（或之前已经配置好）后，调用用户函数 `Time_Show()`，根据秒中断设置的标志位，每隔 1 秒向终端更新一次时间。

## ■ 时间管理结构体

在 `RTC_CheckAndConfig()` 和 `Time_Show()` 都使用了一个输入参数——结构体 `systmtime` 的指针，这个指针是自定义的 `rtc_time` 类型，位于 `date.h` 文件中，见代码清单 16-3。

代码清单 16-3：时间管理结构体的定义

```
1. struct rtc_time {
2.     int tm_sec;
3.     int tm_min;
4.     int tm_hour;
5.     int tm_mday;
6.     int tm_mon;
7.     int tm_year;
8.     int tm_wday;
```

```
9. };
```

这个类型的结构体具有时、分、秒、日、月、年及星期这 7 个成员。当我们需要给 RTC 的计时器重新配置时间时(更改时间戳),我们当然不会询问用户现在距离 UNIX 计时元年过了多少秒。而是向用户询问现在的公元纪年,及所在时区的时间。根据 RTC 计时器向用户输出时间,也是转化为这样容易接受的时间表示方式。

这就是 `rtc_time` 这个结构体的作用,配置 RTC 时,保存用户输入的时间,其它函数通过它求出 UNIX 时间戳,写入 RTC; RTC 正常运行后,需要输出时间时,其它函数通过 RTC 获取 UNIX 时间戳,转化成用户友好的时间表示方式保存在这个结构体上。其实对这个结构体的使用过程就是本 RTC 实验的整个过程。

其实在 C 语言标准库 ANSI C 中,也具有类似这样的结构体 `struct tm`,位于标准的 `time.h` 文件中,而转化函数则为 `mktime()`和 `localtime()`,分别把 `tm` 结构体成员转化成时间戳和用时间戳转化成结构体成员。由于 ANSI C 的函数是不开源的,野火为了把格林威治时间(GMT 标准时间)转化为北京时间(GMT+8),所以使用了开源的万年历算法源码,便于修改,也便于大家学习。

#### 16.3.4. 检查 RTC

`NVIC_Configuration()`函数跟前面例程中的类似,仅中断通道不一样,本次实验使用的中断通道为 `RTC_IRQn`。我们重点来分析 `RTC_CheckAndConfig()`函数,见代码清单 16-4。

代码清单 16-4: `RTC_CheckAndConfig()`函数

```
1. /*
2.  * 函数名: RTC_CheckAndConfig
3.  * 描述   : 检查并配置 RTC
4.  * 输入   : 用于读取 RTC 时间的结构体指针
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void RTC_CheckAndConfig(struct rtc_time *tm)
9. {
10.     /*在启动时检查备份寄存器 BKP_DR1, 如果内容不是 0xA5A5,
11.     则需重新配置时间并询问用户调整时间*/
```

```

12.     if (BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
13.     {
14.         printf("\r\n\r\n RTC not yet configured....");
15.
16.         /* RTC Configuration */
17.         RTC_Configuration();
18.         printf("\r\n\r\n RTC configured....");
19.
20.         /* Adjust time by users typed on the hyperterminal */
21.         Time_Adjust(tm);
22.
23.         BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
24.     }
25.     else
26.     {
27.         /*启动无需设置新时钟*/
28.         /*检查是否掉电重启*/
29.         if (RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
30.         {
31.             printf("\r\n\r\n Power On Reset occurred....");
32.         }
33.         /*检查是否 Reset 复位*/
34.         else if (RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
35.         {
36.             printf("\r\n\r\n External Reset occurred....");
37.         }
38.
39.         printf("\r\n No need to configure RTC....");
40.
41.         /*等待寄存器同步*/
42.         RTC_WaitForSynchro();
43.
44.         /*允许 RTC 秒中断*/
45.         RTC_ITConfig(RTC_IT_SEC, ENABLE);
46.
47.         /*等待上次 RTC 寄存器写操作完成*/
48.         RTC_WaitForLastTask();
49.     }
50.     /*定义了时钟输出宏, 则配置校正时钟输出到 PC13*/
51.     #ifndef RTCClockOutput_Enable
52.     /* Enable PWR and BKP clocks */
53.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_
h_BKP, ENABLE);
54.
55.     /* Allow access to BKP Domain */
56.     PWR_BackupAccessCmd(ENABLE);
57.
58.     /* Disable the Tamper Pin */
59.     BKP_TamperPinCmd(DISABLE); /* To output RTCCLK/64 on Tamper
er pin, the tamper
60.                                     functionality must be disabled */
61.
62.     /* Enable RTC Clock Output on Tamper Pin */
63.     BKP_RTCOutputConfig(BKP_RTCOutputSource_CalibClock);
64.     #endif
65.
66.     /* Clear reset flags */
67.     RCC_ClearFlag();
68.
69. }

```

本函数代码比较长，别急，我们先从整体看看它的流程，再一步步进入它具体调用的其它函数分析。

(1) `if` 语句部分首先调用库函数 `BKP_ReadBackupRegister()` 读取 RTC 备份域寄存器里面的值，看看备份寄存器里面的值是否正确——在后面的代码中，若 RTC 配置成功，我们会向备份域寄存器写入数值 `0XA5A5`，这个数值在  $V_{DD}$  掉电后仍然保存，如果  $V_{BAT}$  也掉电，那么备份域、RTC 所有寄存器将被复位，这时本寄存器的值将不等于 `0XA5A5`，RTC 的计数器的值自然也是无效的。简言之，我们写入的这个数值用作标志 RTC 是否从未被配置或配置是否已经失效，因此，其实我们可以写入任何数值到任何一个备份域寄存器，只要检查的时候与此匹配就行了，所以读者如果想修改时间的话，可以把  $V_{DD}$ 、 $V_{BAT}$  掉电，或在代码上修改这个检查值。

(2) 如果 RTC 从未被配置或配置已经失效(备份域寄存器写入数值不等于 `0XA5A5`)这两种情况有任何一种发生的话，则调用用户函数 `RTC_Configuration()` 来初始化 RTC，配置 RTC 外设的控制参数，时钟分频等，并往电脑的超级终端打印出相应的调试信息。初始化好 RTC 之后，调用函数用户函数 `Time_Adjust()` 让用户键入（通过超级终端输入）时间值，本实验使用的为北京时间。见图 16-2:

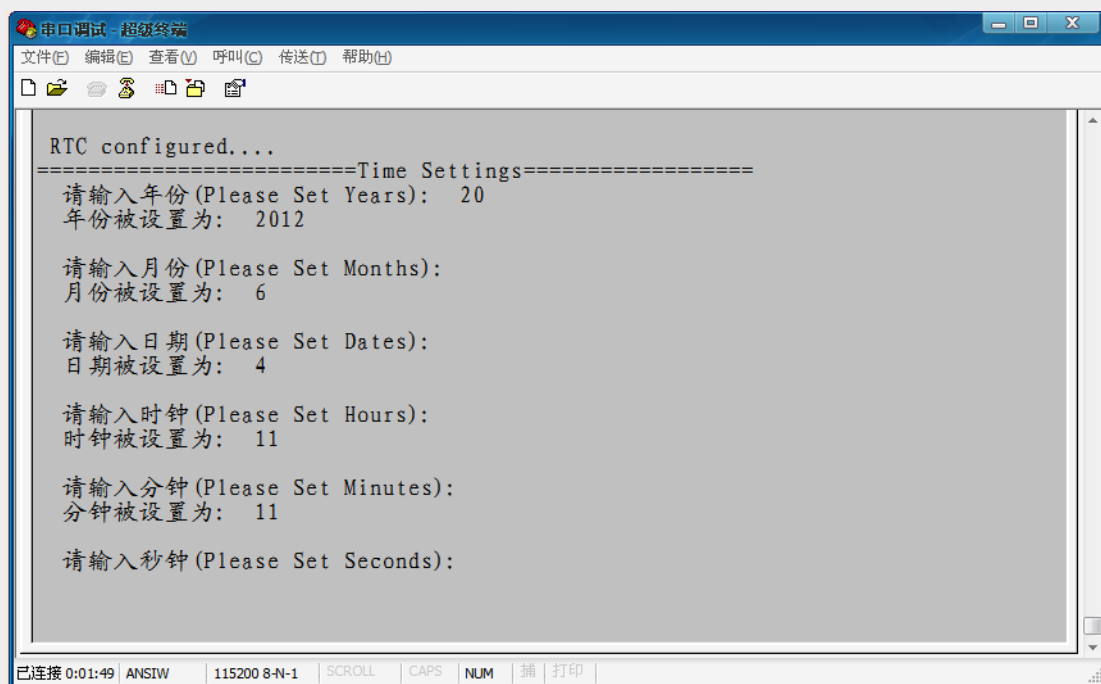


图 16-2 用户输入时间配置

当我们输入时间值后，`Time_Adjust()` 函数把用户输入的北京时间转化为 UNIX 时间戳，并把这个 UNIX 时间戳写入到 RTC 外设的计数寄存器 `RTC_CNT`。接着 RTC 外设在这个时间戳的基础上，每秒对 `RTC_CNT` 加 1，RTC 时钟就运行起来了，并且在  $V_{DD}$  掉电还运行，以后我们需要知道时间就读取 RTC 的计时值(即为该时刻 UNIX 时间戳)，就可以计算出时间了。

设置好时间后，我们调用库函数 `BKP_WriteBackupRegister()`把 `0XA5A5` 这个值写入备份域寄存器，作为我们配置成功的标志。

(3) 如果 if 语句检查备份域，确认 RTC 曾经被配置过，则进入 `else` 部分。`else` 部分用 `RCC_GetFlagStatus()`首先检测是上电复位还是按键复位，根据不同的复位情况在超级终端中打印出不同的调试信息，而两种复位都不需要重新设置 RTC 里面的时间值。接下来调用了库函数 `RTC_WaitForSynchro()`等待 APB1 接口与 RTC 外设同步，上电后第一次通过 APB1 接口访问 RTC 时必须等待同步。在同步完成后才调用 `RTC_ITConfig()`使能 RTC 外设的秒中断，使能 RTC 的秒中断是一个对 RTC 外设寄存器的一个写操作，进行写操作之后，必须要调用 `RTC_WaitForLastTask()`来等待，确保写操作完成。到这里，RTC 的配置就完成了，剩下的我们就是等待秒中断来触发串口向终端更新时间了。

当检测到系统上电复位时，打印出如下信息：

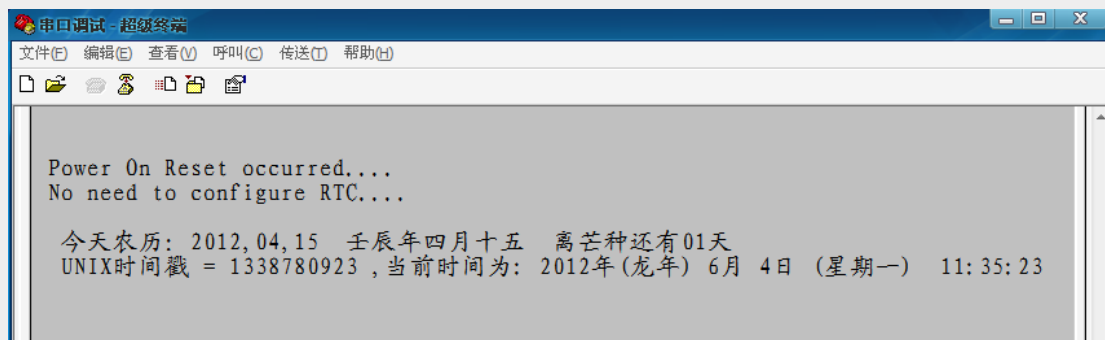


图 16-3 上电复位信息

当检测到系统按键复位时，打印出如下信息：

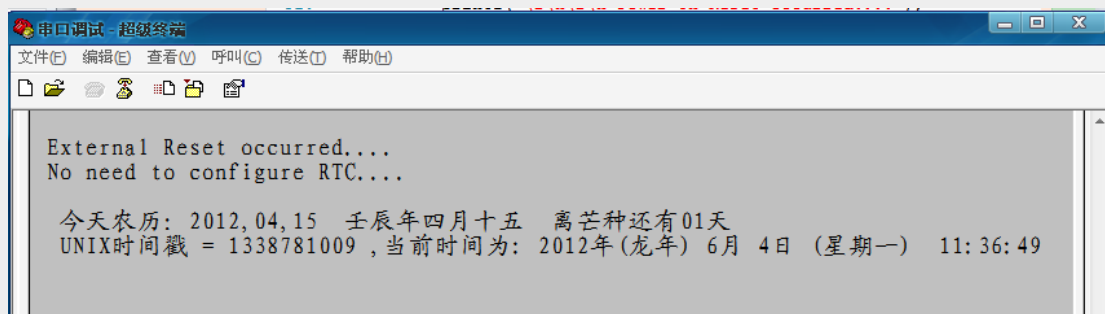


图 16-4 按键复位信息

(4) 在 `RTC_CheckAndConfig()` 函数部分还有条件编译选项询问我们是否需要 `output RTCCLK/64 on Tamper pin`，这是 RTC 的时钟输出配置，若我们在 `rtc.h` 的头文件中定义了 `RTCClockOutput_Enable` 这个宏，PC13 引脚会输出 RTCCLK 的 64 分频时钟，主要是用于 RTC 时钟频率的校准或调整时间补偿。假如用户需要用到这个时钟信号的话，只需在头文件中定义 `RTCClockOutput_Enable` 这个宏即可。

### 16.3.5. 初始化 RTC

从上面的分析知道，若从未配置过 RTC 或 RTC 失效时，会调用用户函数 `RTC_Configuration()` 来对 RTC 进行初始化，下面来分析它的具体实现，见代码清单 16-5。

代码清单 16-5: `RTC_Configuration()` 函数

```
1. /*
```

```

2.  * 函数名: RTC_Configuration
3.  * 描述   : 配置 RTC
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void RTC_Configuration(void)
9. {
10.     /* 使能 PWR 和 BKP 时钟 */
11.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_
BKP, ENABLE);
12.
13.     /* 允许访问 BKP, 备份域 */
14.     PWR_BackupAccessCmd(ENABLE);
15.
16.     /* 对备份域进行软件复位 */
17.     BKP_DeInit();
18.
19.     /* 使能低速外部时钟 LSE */
20.     RCC_LSEConfig(RCC_LSE_ON);
21.     /* 等待 LSE 起振稳定 */
22.     while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
23.     {}
24.
25.     /* 选择 LSE 作为 RTC 外设的时钟 */
26.     RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
27.
28.     /* 使能 RTC 时钟 */
29.     RCC_RTCCLKCmd(ENABLE);
30.
31.     /* 等待 RTC 寄存器与 APB1 同步 */
32.     RTC_WaitForSynchro();
33.
34.     /* 等待对 RTC 的写操作完成 */
35.     RTC_WaitForLastTask();
36.
37.     /* 使能 RTC 秒中断 */
38.     RTC_ITConfig(RTC_IT_SEC, ENABLE);
39.
40.     /* 等待对 RTC 的写操作完成 */
41.     RTC_WaitForLastTask();
42.
43.     /* 设置 RTC 时钟分频: 使 RTC 定时周期为 1 秒 */
44.     RTC_SetPrescaler(32767);
45.     /* RTC 周期 = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */
46.
47.     /*等待对 RTC 的写操作完成 */
48.     RTC_WaitForLastTask();
49. }

```

在这个初始化函数里，我们没有见到熟悉的初始化结构体，对 RTC 的每一个初始化参数都是使用相应的库函数来配置的。RTC 作为备份域的一份子，在访问前首先要使能备份域、电源管理外设的时钟，设置备份域访问权限；作为定时器，初始化时必须要选择好时钟来源，时钟分频。



### 16.3.6. 输入当前 UNIX 时间戳

#### ■ 配置流程

在初始化 RTC 的 `RTC_Configuration()` 函数中，还有一个最重要的定时器参数没有配置，就是计数器 `RTC_CNT` 的值。为了以后读取得的 `RTC_CNT` 值就是 UNIX 时间戳，我们必须在配置时给它输入一个当前时间戳的值，然后使这个值每秒加 1。`RTC_Configuration()` 函数仅完成了每秒加 1 的工作。`RTC_CheckAndConfig()` 函数调用完 `RTC_Configuration()` 之后，紧接着调用了函数 `Time_Adjust()`，输入当前时间戳的工作就由它完成，见。代码清单 16-6

代码清单 16-6: `Time_Adjust()` 函数

```
1.  /*
2.  * 函数名: Time_Adjust
3.  * 描述   : 时间调节
4.  * 输入   : 用于读取 RTC 时间的结构体指针
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void Time_Adjust(struct rtc_time *tm)
9. {
10.     /* 等待前面可能的 RTC 写操作完成 */
11.     RTC_WaitForLastTask();
12.
13.     /* 利用串口，在终端向用户询问当前北京时间(年月日时分秒)，
14.        写入到 rtc_time 型结构体 */
15.     Time_Regulate(tm);
16.
17.     /* 计算输入的日期是星期几，把 rtc_time 型结构体填充完整 */
18.     GregorianDay(tm);
19.
20.     /* 根据输入日期，计算出 UNIX 时间戳，修改当前 RTC 计数寄存器内
容 */
21.     RTC_SetCounter(mktimev(tm));
22.
23.     /* 等待 RTC 写操作完成 */
24.     RTC_WaitForLastTask();
25. }
```

本函数的流程为使用函数 `Time_Regulate()` 从终端获取当前北京时间，然后根据用户的输入，调用函数 `mktimev()` 根据用户输入的年、月、日、时、分、秒数据，计算出相应的 UNIX 时间戳，最后调用库函数 `RTC_SetCounter()` 把这个 UNIX 时间戳写入到计数器 `RTC_CNT`。之后，RTC 就正式运行了。

## ■ 获取时间

Time\_Regulate()主要是通过调用函数 USART\_Scanf()使用串口,通过终端获取到数据的。具体实现见代码清单 16-7。

代码清单 16-7: Time\_Regulate()函数

```
1.  /*
2.  * 函数名: Time_Regulate
3.  * 描述   : 返回用户在超级终端中输入的时间值,并将值储存在
4.  *         RTC 计数寄存器中。
5.  * 输入   : 用于读取 RTC 时间的结构体指针
6.  * 输出   : 用户在超级终端中输入的时间值,单位为 s
7.  * 调用   : 内部调用
8.  */
9. void Time_Regulate(struct rtc_time *tm)
10. {
11.     u32 Tmp_YY = 0xFF, Tmp_MM = 0xFF, Tmp_DD = 0xFF, Tmp_HH =
12.     0xFF, Tmp_MI = 0xFF, Tmp_SS = 0xFF;
13.     printf("\r\n=====Time Settings=====");
14.
15.     printf("\r\n  请输入年份(Please Set Years):  20");
16.     while (Tmp_YY == 0xFF)
17.     {
18.         Tmp_YY = USART_Scanf(99);
19.     }
20.
21.     printf("\n\r  年份被设置为:  20%0.2d\n\r", Tmp_YY);
22.     tm->tm_year = Tmp_YY+2000;
23.
24.     Tmp_MM = 0xFF;
25.
26.     printf("\r\n  请输入月份(Please Set Months):  ");
27.
28.     while (Tmp_MM == 0xFF)
29.     {
30.         Tmp_MM = USART_Scanf(12);
31.     }
32.
33.     printf("\n\r  月份被设置为:  %d\n\r", Tmp_MM);
34.
35.     tm->tm_mon= Tmp_MM;
36.
37.     Tmp_DD = 0xFF;
38.
39.     printf("\r\n  请输入日期(Please Set Dates):  ");
40.
41.     while (Tmp_DD == 0xFF)
42.     {
43.         Tmp_DD = USART_Scanf(31);
44.     }
45.
46.     printf("\n\r  日期被设置为:  %d\n\r", Tmp_DD);
47.
48.     tm->tm_mday= Tmp_DD;
```

```

51.         Tmp_HH = 0xFF;
52.
53.
54.         printf("\n\n 请输入时钟(Please Set Hours):  ");
55.
56.         while (Tmp_HH == 0xFF)
57.         {
58.             Tmp_HH = USART_Scanf(23);
59.         }
60.
61.         printf("\n\n 时钟被设置为:  %d\n\n", Tmp_HH );
62.
63.         tm->tm_hour= Tmp_HH;
64.
65.         Tmp_MI = 0xFF;
66.
67.         printf("\n\n 请输入分钟(Please Set Minutes):  ");
68.
69.         while (Tmp_MI == 0xFF)
70.         {
71.             Tmp_MI = USART_Scanf(59);
72.         }
73.
74.         printf("\n\n 分钟被设置为:  %d\n\n", Tmp_MI);
75.
76.         tm->tm_min= Tmp_MI;
77.
78.         Tmp_SS = 0xFF;
79.
80.         printf("\n\n 请输入秒钟(Please Set Seconds):  ");
81.
82.         while (Tmp_SS == 0xFF)
83.         {
84.             Tmp_SS = USART_Scanf(59);
85.         }
86.
87.         printf("\n\n 秒钟被设置为:  %d\n\n", Tmp_SS);
88.
89.         tm->tm_sec= Tmp_SS;
90.     }

```

USART\_Scanf()函数在前面的串口例程中没有介绍，在这里补充一下它的代码。分析并不难，当是野火留给大家的思考题吧，要注意的问题是从终端输入的是 ASCII 码，而需要的是实际数值，函数中的大部分代码就是为了 ASCII 码转换为数字而存在的，见代码清单 16-8。

代码清单 16-8: USART\_Scanf()函数

```

1.  /*****
2.  * 函数名   : USART_Scanf
3.  * 描述     : 从微机超级终端获取数字值(把 ASCII 码转换为数字)
4.             本函数专用于 RTC 获取时间，若进行其它输入应用，要修改一下
5.  * 输入     : - value 用户在超级终端中输入的数值上限
6.  * 输出     : 输入字符的 ASCII 码对应的数值
7.  * 调用     : 内部调用
8.  ***/
9.
10. static uint8_t USART_Scanf(uint32_t value)

```

```

11. {
12.     uint32_t index = 0;
13.     uint32_t tmp[2] = {0, 0};
14.
15.     while (index < 2)
16.     {
17.         /* Loop until RXNE = 1 */
18.         while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) ==
RESET)
19.         {
20.         }
21.
22.         tmp[index++] = (USART_ReceiveData(USART1));
23.         if ((tmp[index - 1] < 0x30) || (tmp[index -
1] > 0x39)) /*数字 0 到 9 的 ASCII 码为 0x30 至 0x39*/
24.         {
25.             printf("\n\rPlease enter valid number between 0 and 9 --
>: ");
26.             index--;
27.         }
28.     }
29.
30.     /* 计算输入字符的 ASCII 码转换为数字*/
31.     index = (tmp[1] - 0x30) + ((tmp[0] - 0x30) * 10);
32.     /* Checks */
33.     if (index > value)
34.     {
35.         printf("\n\rPlease enter valid number between 0 and %d"
, value);
36.         return 0xFF;
37.     }
38.     return index;
39. }

```

## ■ 计算 UNIX 时间戳

从用户端获取了北京时间后，就可以用它换成 UNIX 时间戳了，但不能忽略一个重要的问题——时差。UNIX 时间戳的计时元年是以标准时间(GMT 时区)为准的，而北京时间为 GMT+8，即时差为+8 小时。为了保证我们写入到 RTC\_CNT 的是标准的 UNIX 时间戳(主要是为了兼容)，以北京时间转化出的‘秒数’要减去 8\*60\*60 才是标准的 UNIX 时间戳。这个计算在 date.c 文件中的 mktimeev()函数中实现，见代码清单 16-9。

代码清单 16-9: mktimeev()函数

```

1. /*
2. 函数名: mktimeev
3. 输入: 时间结构体, 包含时间数据
4. 返回: UNIX 时间戳
5. *ADD by fire: 本函数在工程中的输入参数为北京时间,
6. 所以在转换成时间戳时最后要从北京时间转换为标准时间的时戳
7. */
8. u32 mktimeev(struct rtc_time *tm)

```

```

9.  {
10.     if (0 >= (int) (tm->tm_mon - 2)) { /* 1..12 -
> 11,12,1..10 */
11.         tm-
>tm_mon += 12;          /* Puts Feb last since it has leap day */
12.         tm->tm_year -= 1;
13.     }
14.
15.     return (((
16.         (u32) (tm->tm_year/4 - tm->tm_year/100 + tm-
>tm_year/400 + 367*tm->tm_mon/12 + tm->tm_mday) +
17.         tm->tm_year*365 - 719499
18.         )*24 + tm->tm_hour /* now have hours */
19.         )*60 + tm->tm_min /* now have minutes */
20.         )*60 + tm->tm_sec-8*60*60; /* finally seconds */
21.     /*Add by fire: -8*60*60 把输入的北京时间转换为标准时间,
22.     再写入计时器中, 确保计时器的数据为标准 UNIX 时间戳*/
23.
24. }

```

野火在这个函数中的返回值计算式的最后，添加了 $(-8*60*60)$ ，使得这个函数实现了输入北京时间，返回标准 UNIX 时间戳的功能。如果想使用其它时区，则根据不同的时区为修改这个值。这个 `mktimev()` 是在工程中的 `Time_Adjust()` 函数调用的，输入参数是从用户的终端得到的时间，它的返回值最终被写入到 `RTC_CNT` 计数器中：

```

1.  /* 根据输入日期，计算出 UNIX 时间戳，修改当前 RTC 计数寄存器内容 */
2.      RTC_SetCounter(mktimev(tm));

```

至此 `RTC_CheckAndConfig()` 函数就为大家分析完了，我们再看看这个工程是怎样利用 RTC 提供北京时间的。

### 16.3.7. 输出时间到终端

在 `main` 函数中，调用了 `Time_Show()` 进行时间的输出，其代码见代码清单 16-10。

代码清单 16-10: `Time_Show()` 函数

```

1.  /*
2.   * 函数名: Time_Show
3.   * 描述   : 在超级终端中显示当前时间值
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8.  void Time_Show(struct rtc_time *tm)
9.  {

```

```
10.  /* Infinite loop */
11.  while (1)
12.  {
13.      /* 每过1s */
14.      if (TimeDisplay == 1)
15.      {
16.          /* Display current time */
17.          Time_Display( RTC_GetCounter(), tm);
18.          TimeDisplay = 0;
19.      }
20.  }
21. }
```

其中变量 `TimeDisplay` 是 RTC 秒中断标志，RTC 的秒中断被触发后，进入中断服务函数，把这个变量 `TimeDisplay` 置 1。本函数死循环检查这个标志，变为 1 时，调用函数 `Time_Display()` 显示最新时间，就可以实现每隔 1 秒向终端更新一次时间了，更新完时间后再把变量 `TimeDisplay` 置 0，等待下一次标志。

RTC 的秒中断服务函数位于文件，实现见代码清单 16-11。

代码清单 16-11: RTC 中断服务函数

```
1. void RTC_IRQHandler(void)
2. {
3.     if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
4.     {
5.         /* 清除秒中断标志 */
6.         RTC_ClearITPendingBit(RTC_IT_SEC);
7.
8.         /* 把标志位置 1 */
9.         TimeDisplay = 1;
10.
11.        /* 等待写操作完成 */
12.        RTC_WaitForLastTask();
13.    }
14. }
```

从中断服务函数可以看到，在这个函数中并没有任何对 `RTC_CNT` 的操作。若  $V_{DD}$  掉电，RTC 是无法触发秒中断的，所以想利用秒中断的方案来实现实时时钟是不现实的，秒中断最适合用在类似本实验的触发显示的时间更新场合，而不是用于计数。

我们再来看看 `Time_Display()` 函数的代码，见代码清单 16-12。

代码清单 16-12: `Time_Display()` 函数

```
1.
2. /*
3.  * 函数名: Time_Display
4.  * 描述   : 显示当前时间值
5.  * 输入   : -TimeVar RTC 计数值，单位为 s
```

```

6.  * 输出 : 无
7.  * 调用 : 内部调用
8.  */
9.  void Time_Display(uint32_t TimeVar, struct rtc_time *tm)
10. {
11.     static uint32_t FirstDisplay = 1;
12.     uint32_t BJ_TimeVar;
13.     uint8_t str[15]; // 字符串暂存
14.
15.     /* 把标准时间转换为北京时间*/
16.     BJ_TimeVar = TimeVar + 8*60*60;
17.
18.     to_tm(BJ_TimeVar, tm); /*利用时间戳转换为北京时间*/
19.
20.     if((!tm->tm_hour && !tm->tm_min && !tm-
>tm_sec) || (FirstDisplay))
21.     {
22.
23.         GetChinaCalendar((u16) tm->tm_year, (u8) tm-
>tm_mon, (u8) tm->tm_mday, str);
24.
25.         printf("\r\n\r\n 今天农
历: %0.2d%0.2d,%0.2d,%0.2d", str[0], str[1], str[2], str[3]);
26.
27.         GetChinaCalendarStr((u16) tm->tm_year, (u8) tm-
>tm_mon, (u8) tm->tm_mday, str);
28.         printf("  %s", str);
29.
30.         if(GetJieQiStr((u16) tm->tm_year, (u8) tm-
>tm_mon, (u8) tm->tm_mday, str))
31.             printf("  %s\n\r", str);
32.
33.         FirstDisplay = 0;
34.     }
35.
36.     /* 输出时间戳, 公历时间 */
37.     printf("\r UNIX 时间戳 = %d ,当前时间为: %d 年(%s 年) %d 月 %d
日 (星期%s) %0.2d:%0.2d:%0.2d", TimeVar,
38.         tm->tm_year, zodiac_sign[(tm->tm_year-
3)%12], tm->tm_mon, tm->tm_mday,
39.         WEEK_STR[tm->tm_wday], tm->tm_hour,
40.         tm->tm_min, tm->tm_sec);
41.
42. }

```

本函数的第一个输入参数为 UNIX 时间戳，在 Time\_Show()调用它时，我们利用库函数 RTC\_GetCounter()读取了 RTC 的 RTC\_CNT 的当前计数值，并把这个计数值作为 Time\_Display()的输入参数。

根据我们的配置，RTC\_CNT 的计数值是标准时间 GMT 的 UNIX 时间戳，为了计算北京时间，我们在使用 RTC\_CNT 计数值转换北京时间时，要加上时差，在代码中的第 16 行实现：BJ\_TimeVar = TimeVar + 8\*60\*60。之后，把这个变量 BJ\_TimVar 作为函数 to\_tm()的输入参数，把时间戳转换成年、月、日、时、分、秒的格式，并保存到时间结构体中。函数 to\_tm()位于 date.c 文

件，由于这个函数是纯算法，就不展开了。而农历计算则由后面的 GetChinaCalendar()函数实现，也是位于 date.c 文件。在 Time\_Display()函数的最后，把 UNIX 标准时间戳，农历，公历，星期，时间都输出到终端。

如果使用普通 51 芯片，想实现一个 RTC 时钟的时候还需借助时钟芯片，如 DS1302 或 DS12C887，而现在我们用一个定时器就可以完成了。

### 16.3.8. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，根据 RTC 是否失效，复位方式的不同，实验现象会有所不同，这在实验分析中已经贴图。

在这里贴上 RTC 最终正常运行的实验现象。如果大家在网上搜索“UNIX 时间戳”，可以找到一些网站，它不断更新当前互联网上使用的标准 UNIX 时间戳。在此，野火用开发板返回到终端的 UNIX 时间戳进行对比。图中的背景是网站提供实时更新的时间戳，与返回终端的值相差不大，大家也可以尝试对比一下，过几天后看看误差，有必要的话调整一下 RTC 的时间补偿。





## 17. 电源管理

## 18. CAN 通信

CAN 是控制器局域网(Controller Area Network, CAN)的简称,是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发的,并最终成为国际标准(ISO11519)。是国际上应用最广泛的现场总线之一。

在北美和西欧,CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线,并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。近年来,其所具有的高可靠性和良好的错误检测能力受到重视,被广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强和振动大的工业环境。

### 18.1. CAN 协议简介

#### 18.1.1. 物理层

与 I<sup>2</sup>C、SPI 等具有时钟信号的通讯方式不同,CAN 通讯并不是以时钟信号来进行同步的。它只具有 CAN\_High 和 CAN\_Low 两条信号线,共同构成一组差分信号线,所以 CAN 是以差分信号的形式进行通讯的。见图 18-1。

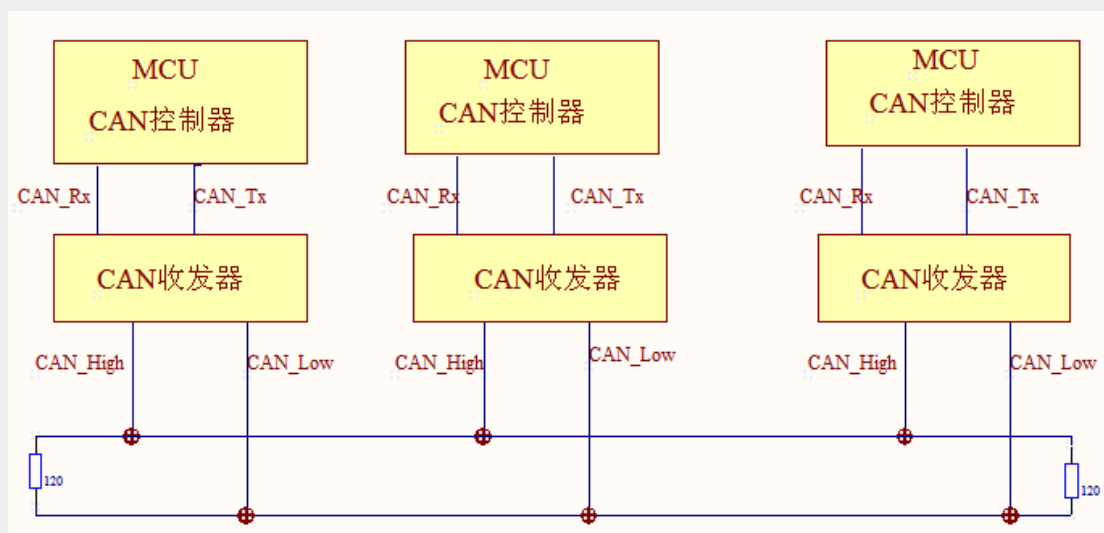


图 18-1 CAN 闭环总线通讯网络

图中的 CAN 通讯网络是遵循 ISO11898 标准的高速、短距离闭环网络,它的总线最大长度为 40m,通信速度最高为 1Mbps。还有一种是遵循 ISO11519-

2 标准的低速远离开环网络，它最大传输距离为 1km，最高通讯速率为 125kbps。

从 CAN 的通讯网络图可以了解到，它的通讯节点由一个 CAN 控制器，CAN 收发器组成。如 STM32 的 CAN 接口即为 CAN 控制器，为了构成完整的节点，还要给它外接一个 CAN 收发器。

在发送数据时，CAN 控制器把要发送的二进制编码通过 CAN\_Tx 线发送到 CAN 收发器，然后由收发器把这个普通的逻辑电平信号转化成差分信号，通过差分线 CAN\_High 和 CAN\_Low 线输出到 CAN 总线网络。在接收数据时，这个过程相反。

差分信号是什么意思呢？即信号的逻辑 0 和逻辑 1 由两根差分信号线的电压差来表示。ISO 11898 规定 CAN 协议中的逻辑 1 (隐性电平)时，CAN\_High 和 CAN\_Low 线上的电压均为 2.5v，即它们的电压差  $V_h - V_l = 0V$ 。而在逻辑 0 (显性电平)中，CAN\_High 的电平为 3.5V，CAN\_Low 线的电平为 1.5V，即它们的电压差为  $V_h - V_l = 2V$ 。

如当 CAN 收发器从 CAN\_Tx 线接收到来自 CAN 控制器的低电平信号时，它会将该信号转化，使 CAN\_High 输出 3.5V，同时 CAN\_Low 输出 1.5V，即显性电平。如同串口中的 MAX3232，用作电平转换。CAN 收发器的作用则是用作差分信号的转换。

在 CAN 总线中，必须处于隐性电平(逻辑 1)或显性电平(逻辑 0)中的一个状态。假如有两个 CAN 通讯节点，在同一时间，一个输出隐性电平，另一个输出显性电平，总线的“线与”特性将使它处于显性电平状态，即可以认为显性具有优先的意味。

### 18.1.2. CAN 的报文种类及结构

在 SPI 通讯中，片选、时钟信号、数据输入及数据输出这四个信号都有单独的信号线。而 CAN 使用的是两条差分信号线，只能表达一个信号。简洁的物

理层决定了 CAN 必然要配上一套更复杂的协议。如何用一个信号通道实现同样、甚至更强大的功能呢？答案是对数据或操作命令进行打包。

### 18.1.3. 报文的种类

在原始数据段的前面加上传输起始标签、片选(识别)标签、控制标签，在数据的尾段加上 CRC 校验标签、应答标签和传输结束标签。把这些内容按特定的格式打包好，就可以用一个通道表达各种信号了，各种各样的标签就如同 SPI 中各种通道上的信号，起到了协同传输的作用。当整个数据包被传输到其它设备时，只要这些设备按格式去解读，就能还原出原始数据。类似这样的数据包就被称为 CAN 的数据帧。为了更有效地控制通讯，CAN 一共规定了 5 种类型的帧，帧也称为报文，它们的类型及用途说明如表 18-1。

表 18-1 帧的种类及其用途

| 帧   | 帧用途                         |
|-----|-----------------------------|
| 数据帧 | 用于发送单元向接收单元传送数据的帧。          |
| 遥控帧 | 用于接收单元向具有相同 ID 的发送单元请求数据的帧。 |
| 错误帧 | 用于当检测出错误时向其它单元通知错误的帧。       |
| 过载帧 | 用于接收单元通知其尚未做好接收准备的帧。        |
| 帧间隔 | 用于将数据帧及遥控帧与前面的帧分离开来的帧。      |

### 18.1.4. 数据帧的结构

数据帧是在 CAN 通讯中最主要、最复杂的报文，我们来了解它的结构，见图 18-2。

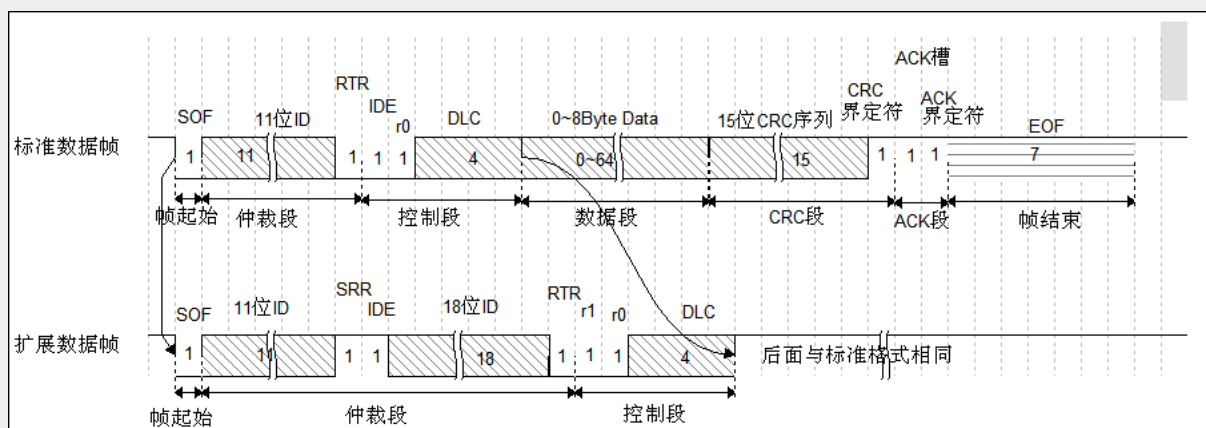


图 18-2 数据帧的结构

数据帧以一个显性位(逻辑 0)开始，以 7 个连续的隐性位(逻辑 1)结束。在它们之间，分为仲裁段、控制段、数据段、CRC 段和 ACK 段。

### ■ 仲裁段

仲裁段的内容主要为本数据帧的 ID 信息。数据帧分为标准格式和扩展格式两种，区别就在于 ID 信息的长度，标准格式的 ID 为 11 位，扩展格式的仲裁段为 29 位。

在 CAN 协议中，ID 起着重要的作用，它决定着数据帧发送的优先级，也决定着其它设备是否会接收这个数据帧。CAN 协议不对挂载在它之上的设备分配优先级，对总线的占有权是由信息的重要性决定的，即对于重要的信息，我们会给它打包上一个优先级高的 ID，使它能够及时地发送出去。也正因为它这样的优先级分配原则，使得 CAN 的扩展性大大加强，在总线上增加或减少节点并不影响其它设备。

报文的优先级，是通过对 ID 的仲裁来确定的。前面对物理层的分析我们知道如果总线上同时出现显性电平和隐性电平，总线的状态会被置为显性电平，优先级的仲裁就根据这个特性实现的。

见图 18-3。若两个节点同时竞争 CAN 总线的占有权，在它们发送报文时，若首先出现隐性电平，则会失去对总线的占有权，进入接收状态。在开始阶段，两个设备发送的电平一样，所以它们一直继续发送数据。到了图中箭头

所指的时序处，节点单元 1 发送的为隐性时序，而此时节点单元 2 发送的为显性时序，因此单元 2 竞争总线成功，这个报文得以被继续发送出去。

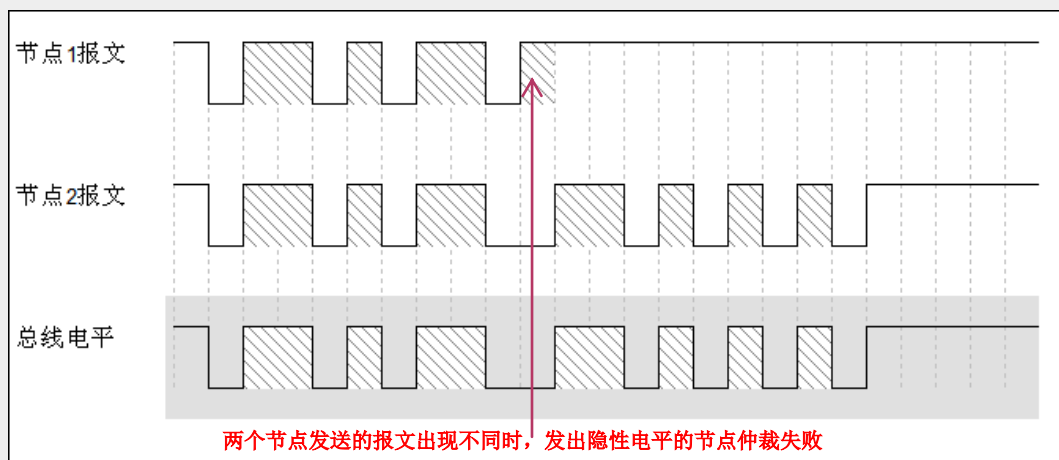


图 18-3 仲裁过程

仲裁段 ID 的优先级也影响着接收设备对报文的反应。因为在 CAN 总线上数据是以广播的形式发送的，所有连接在 CAN 总线的节点都会收到所有其它节点发出的有效数据，因而我们的 CAN 控制器大多具有根据 ID 过滤报文的功能，即只接收某些 ID 的报文。

仲裁段除了报文 ID 外，还有 RTR、IDE、SRR 位。

其中 RTR 位(Remote Transmission Request BIT)，是用于区分数据帧和遥控帧的，在数据帧里这一位为显性(逻辑 0)。

IDE 位(Identifier Extension Bit)，是用于区分标准格式与扩展格式，在标准格式中 IDE 位为显性，在扩展格式里 IDE 位为隐性。

SRR 位(Substitute Remote Request BIT)，只存在于扩展格式，它用于替代标准格式中的 RTR 位。SRR 位为隐性位，由于 RTR 在数据帧为显性位，所以在两个 ID 相同的标准格式报文与扩展格式报文中，标准格式的优先级较高。

## ■ 控制段

在控制段中的 `r1` 和 `r0` 为保留位，默认设置为显性位。最主要的为 `DLC` 段，`DLC` 段由 4 位组成，`MSB` 先行，它二进制编码用于表示本报文中的数据段含有多少个字节，`DLC` 段表示的数字为 0~8。

## ■ 数据段

数据段为数据帧的核心内容，它由 0~8 个字节组成，`MSB` 先行。

## ■ CRC 段

为了保证报文的正确传输，`CAN` 的报文包含了一段 15 位的 `CRC` 校验码，一旦接收端计算出的 `CRC` 码跟接收到的 `CRC` 码不同，则会向发送端反馈出错信息，重新发送。`CRC` 部分的计算和出错处理一般由 `CAN` 控制器硬件完成或由软件控制最大重发数。

在 `CRC` 校验码之后，有一个 `CRC` 界定符，它为隐性位，主要作用是把 `CRC` 校验码与后面的 `ACK` 段间隔起来。

## ■ ACK 段

`ACK` 段包括一个 `ACK` 槽位，和 `ACK` 界定符位。在 `ACK` 槽位中，发送端发送的为隐性位，而接收端则在这一位中发送显性位以示应答。在 `ACK` 槽和帧结束之间由 `ACK` 界定符间隔开。

## ■ 帧结束

帧结束段由发送端发送 7 个隐性位表示结束。

`CAN` 其它种类的帧结构请参阅 `CAN` 标准协议。

### 18.1.5. 同步

由于 CAN 没有时钟信号线，而且它的报文中并没有包含用于同步的标志。所以要使用位同步的方式来确保通讯时序，对总线的电平进行正确的采样。

#### ■ 位时序分解

为了实现位同步，CAN 协议把每一位的时序分解成如图所示的 SS 段、PTS 段、PBS1 段、PBS2 段，这四段的长度加起来即为一个 CAN 数据位的长度。分解后最小的时间单位是  $Tq$ ，而一个完整的位由 8~25 个  $Tq$  组成。图中的一位为  $19Tq$ 。见图 18-4。

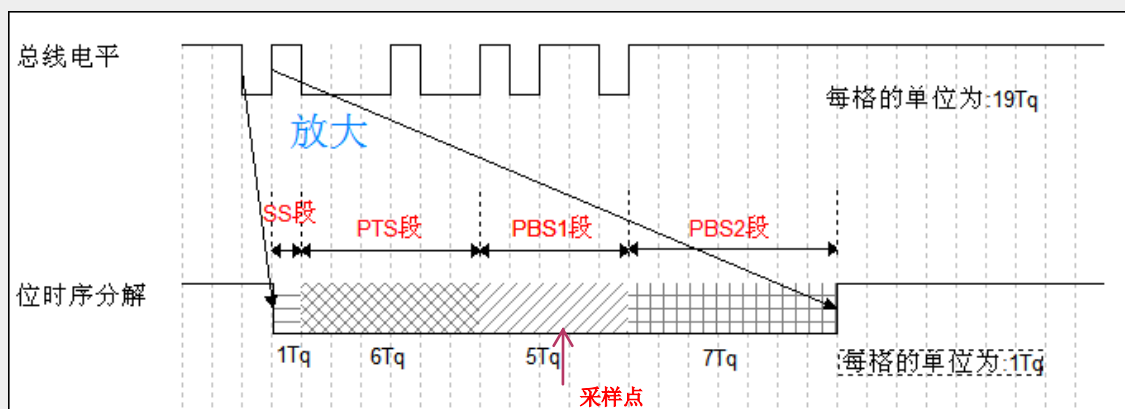


图 18-4 CAN 位时序分解图

每位中的各段作用如下：

□ SS 段(SYNC SEG)，译为同步段，若总线的跳变沿被包含在 SS 段的范围之内，则表示节点与总线的时序同步，节点与总线同步时，采样点采集到的总线电平即可被确定为该位的电平。如当总线上出现帧起始信号(SOF)时，其它节点上的控制器根据总线上的这个下降沿，对自己的位时序进行调整，把该下降沿包含到 SS 段内，这样根据起始帧来进行同步的方式称为硬同步。其中 SS 段的大小为  $1Tq$ 。

□ PTS 段(PROP SEG)，译为传播时间段，这个时间段是用于补偿网络的物理延时时间。是总线上输入比较器延时和输出驱动器延时总和的两倍。PTS 段的大小为  $1\sim 8Tq$ 。



□ PBS1 段(PHASE SEG1)，译为相位缓冲段，主要用来补偿边沿阶段的误差，它的时间长度在重新同步的时候可以加长。PBS1 段的初始大小可以为  $1\sim 8T_q$ 。

□ PBS2 段(PHASE SEG2)，这是另一个相位缓冲段，也是用来补偿边沿阶段误差的，它的时间长度在重新同步时可以缩短。PBS2 段的初始大小可以为  $2\sim 8T_q$ 。

□ 在重新同步的时候，PBS1 和 PBS2 段的允许加长或缩短的时间长度定义为：重新同步补偿宽度 SJW (reSynchronization Jump Width)。

## ■ 同步过程分析

CAN 的同步分为硬同步和重新同步。因为硬同步时只是在有帧起始信号时起作用，无法确保后续一连串的位时序都是同步的，所以 CAN 还引入了重新同步的方式。在检测到总线上的时序与节点使用的时序有相位差时(即总线上的跳变沿不在节点时序的 SS 段范围)，通过延长 PBS1 段或缩短 PBS2 段，来获得同步，这样的方式称为重新同步。见图 18-5。

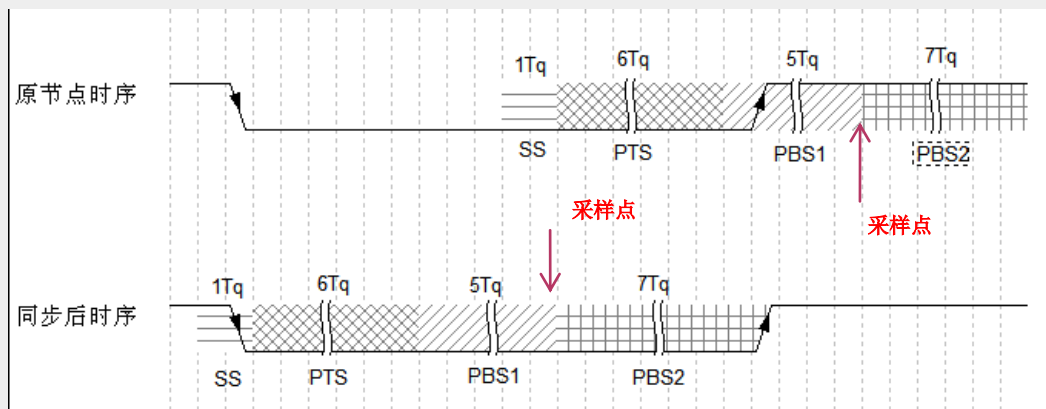


图 18-5 硬同步过程图

可以看到在总线出现帧起始信号时，该节点原来的位时序与总线时序不同步，因而这个状态的采样点采集得的数据是不正确的。所以节点以硬同步的方式调整，把自己的位时序中的 SS 段平移至总线出现下降沿的部分，获得同步，这时采样点采集得的为正确数据。

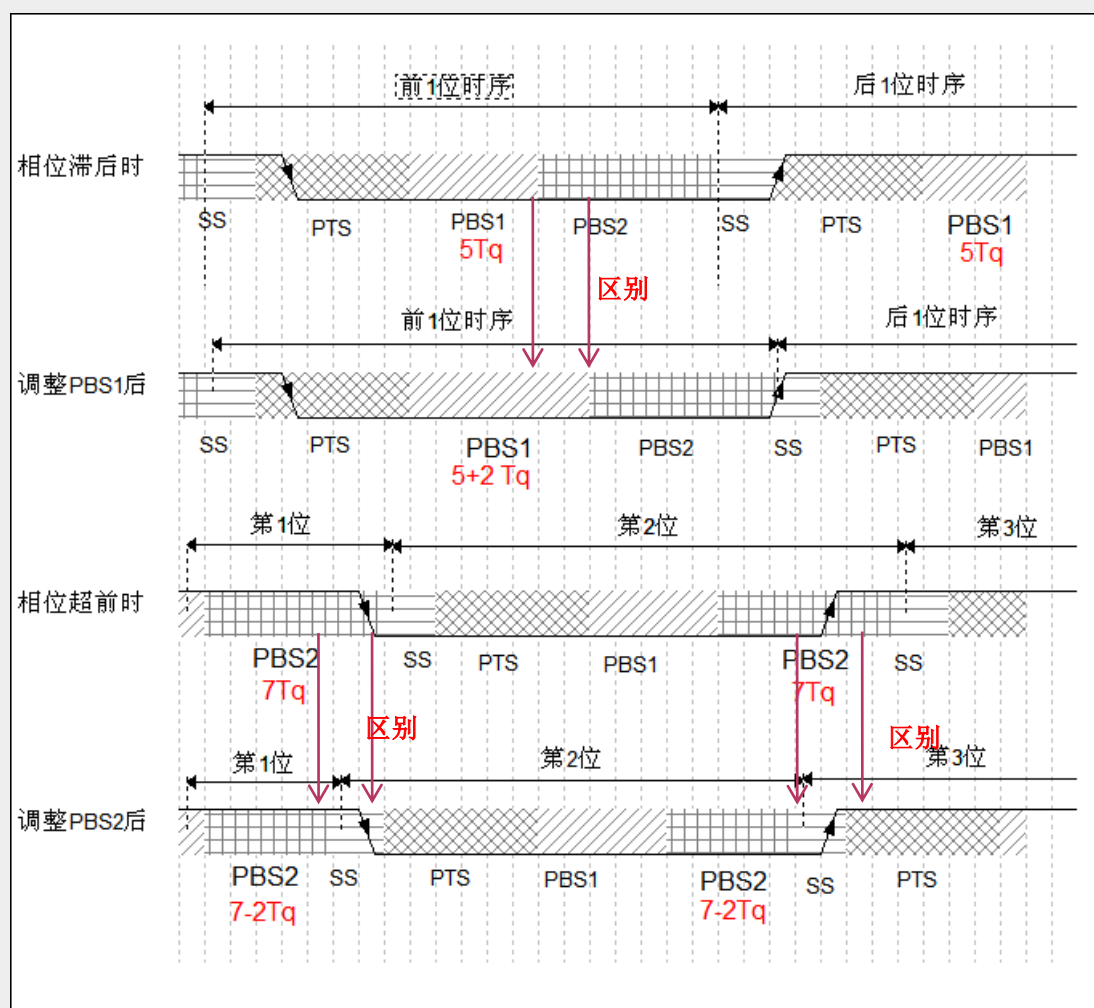


图 18-6 重新同步时的两种情况

重新同步的方式分为两种情况，如图。第一种，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对滞后  $2Tq$ ，这时控制器在下一个位时序中的 PBS1 段增加  $2Tq$  的时间长度，使得节点与总线时序重新同步。第二种，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对超前  $2Tq$ ，这时控制器在前一个位时序中的 PBS2 段减少  $2Tq$  的时间长度，获得同步。这里设置的 PBS1 和 PBS2 能够增减的最大时间长度为  $SJW=2Tq$ ，若 SJW 设置得太小，则重新同步的调整速度慢，若设置得太大，则影响传输速率。

## 18.2. STM32 的 CAN 特性及架构

### 18.2.1. CAN 特性

STM32 的所有型号芯片中都具有 **bxCAN** 控制器 (Basic Extended CAN)，它支持 CAN 协议 2.0A 和 2.0B。**bxCAN** 接口可以自动地接收和发送 CAN 报文，支持标准标识符和扩展标识符。

具有 3 个发送邮箱，发送报文的优先级可以使用软件，可以记录发送的时间。有 2 个 3 级深度的接收 FIFO，可以使用过滤功能只接收或不接收某些 ID 号的报文。可以配置成自动重发。不支持使用 DMA 进行数据收发。

### 18.2.2. CAN 架构

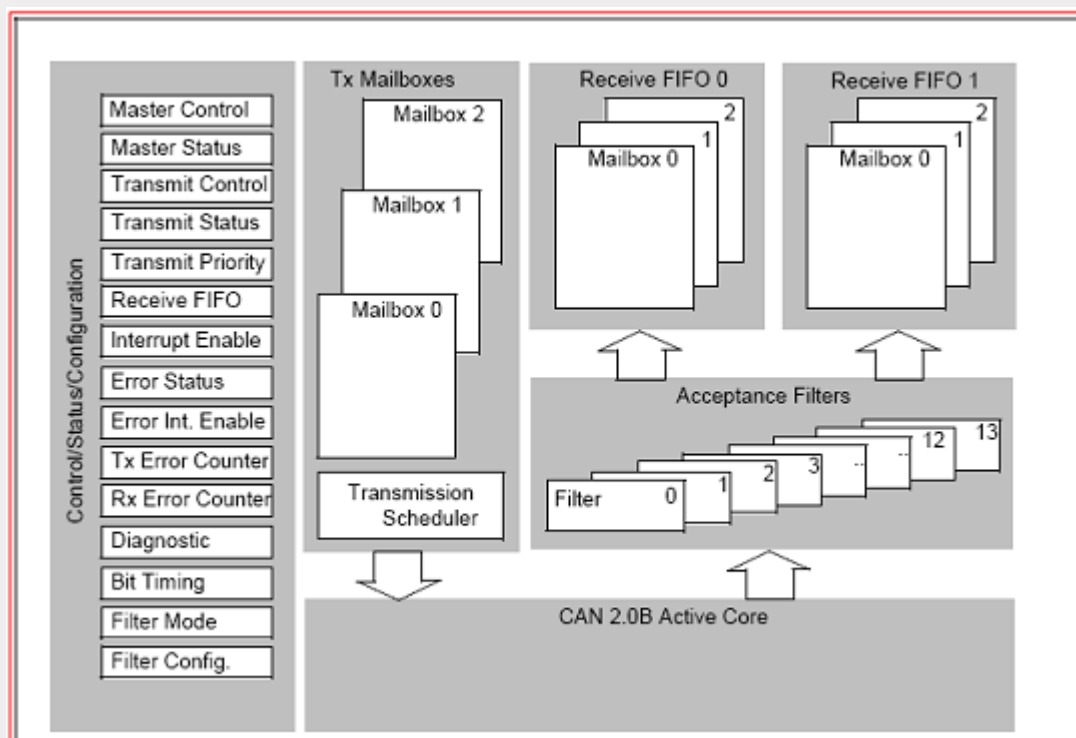


图 18-7 STM32 的 CAN 架构

上图中左边的 **Control / Status / Configuration** 的具体细节是关于 CAN 的各种配置。这个图右边有三个专业名词：

- **Tx Mailboxes**(发送邮箱)

STM32 的 CAN 中共有 3 个发送邮箱供软件来发送报文。发送调度器根据优先级决定哪个邮箱的报文先被发送。

#### ❑ Acceptance Filters( 接收过滤器 )

STM32 的 CAN 中共有 14 个位宽可变/可配置的标识符过滤器组，软件通过对它们编程，从而在 CAN 收到的报文中选择它需要的报文，而把其它报文丢掉。

#### ❑ Receive FIFO( 接收 FIFO )

STM32 的 CAN 中共有 2 个接收 FIFO，每个 FIFO 都可以存放 3 个完整的报文。它们完全由硬件来管理。

CAN 的工作就是围绕这三部分展开的，我们通过实例进行详细讲解。

### 18.3. 双 CAN 通信 实验分析

本节讲解的实验是在两个开发板之间使用 CAN 接口进行通讯。STM32 的 CAN 接口还可以使用回环模式，进行 CAN 的自检。若只有一个开发板的读者可以使用回环模式来进行测试。

#### 18.3.1. 实验描述及工程文件清单

##### 实验描述：

双 CAN 测试实验（中断模式），采用两块实验板充当某次 CAN 通信过程的主动发送和被动接受节点。主动设备首先发送信号“ABCD”，若被动设备接收到这个信号后，会向发送出“DCBA”。若主设备成功收到答复信号“DCBA”，则通过 USART1 向终端中打印信息。本次实验中，主动设备在被动设备之后运行。

##### 硬件连接：

PB8-CAN-RX

PB9-CAN-TX

库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_can.c

FWlib/misc.c

用户文件：

USER/main.c

USER/stm32f10x\_it.c

USER/led.c

USER/usart.c

USER/can.c

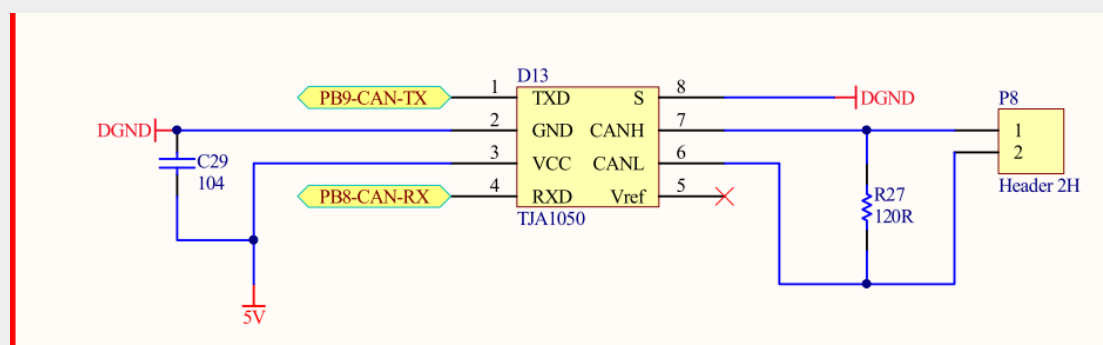


图 18-8 CAN 硬件原理图

野火开发板使用的 STM32 型号为：STM32F103VET6，它自带了一个 CAN 控制器(即我们说的 CAN 接口)。具体 I/O 定义为 PB8-CAN-RX、PB9-CAN-TX。板载的 CAN 外接了一个 TJA1050 型号的 CAN 收发器，它 RXD 的和 TXD 引脚

分别连接到 STM32 的 PB8 和 PB9，它的 CANH 和 CANL 引脚是外接到排针上的，并且在 CANH 和 CANL 之间连接了一个 120 欧姆的电阻。

在使用双 CAN 通讯时，我们可以利用双绞线或两条杜邦线把相应的信号连接起来，由两个板上的电阻和双绞线就构成了 CAN 总线网络。而两块板就成为接入到这个 CAN 网络的两个节点了。见图 18-9。

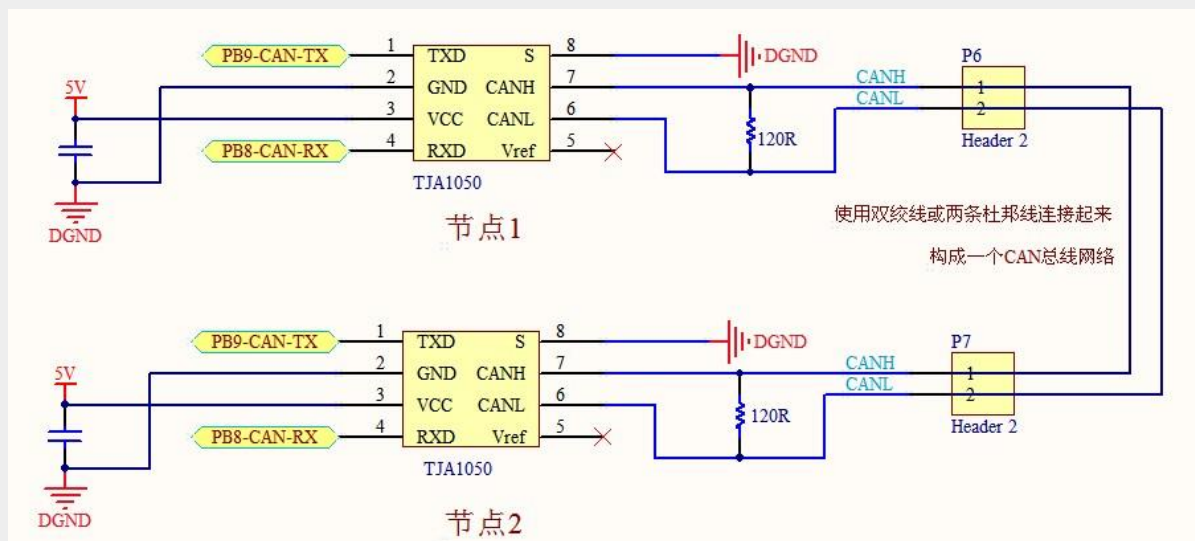
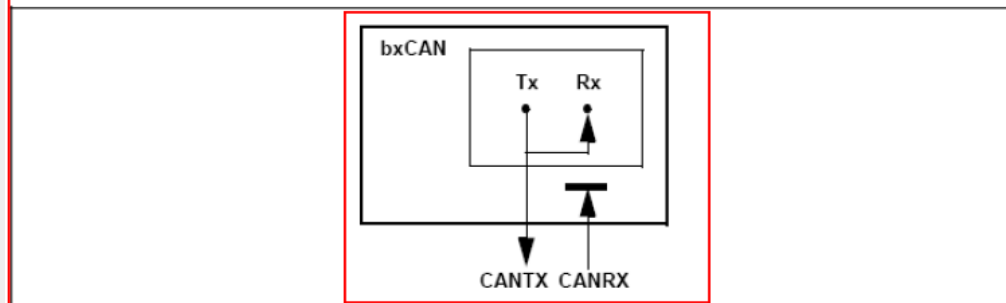


图 18-9 双 CAN 通讯连接图

若只是用了 CAN 的回环测试，就不需要挂接外部的 CAN 节点。当我们用 CAN 的回环测试时，硬件会在内部将 TX 和 RX 连接起来，实现内部的收和发，从而达到测试的目的。

通过对CAN\_BTR寄存器的LBKM位置'1'，来选择环回模式。在环回模式下，bxCAN把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。

图194 bxCAN工作在环回模式



环回模式可用于自测试。为了避免外部的影响，在环回模式下CAN内核忽略确认错误(在数据/远传帧的确认位时刻，不检测是否有显性位)。在环回模式下，bxCAN在内部把Tx输出回馈到Rx输入上，而完全忽略CANRX引脚的实际状态。发送的报文可以在CANTX引脚上检测到。

图 18-10 回环测试模式

### 18.3.2. 配置工程环境

本双 CAN 通讯中我们创建了两个工程，分别用于节点 1(主机)和节点 2(从机)它们的内容大体相似，讲解时主要使用主设备的代码进行讲解。

在实验环境配置部分，这两个工程是完全一样的。实验中我们用到了 GPIO、RCC、USART 及 CAN 外设，所以我们要把以下库文件添加到主、从设备的工程中：stm32f10x\_gpio.c、stm32f10x\_rcc.c stm32f10x\_usart.c、stm32f10x\_can.c。本实验中使用了中断来读取 CAN 接收邮箱中的数据，所以我们还要添加 misc.c 文件。

接下来添加旧工程中的外设用户文件 usart.c，以便调试和观察实验效果。新建 can.c 及 can.h 文件，并在 stm32f10x\_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 18-1。

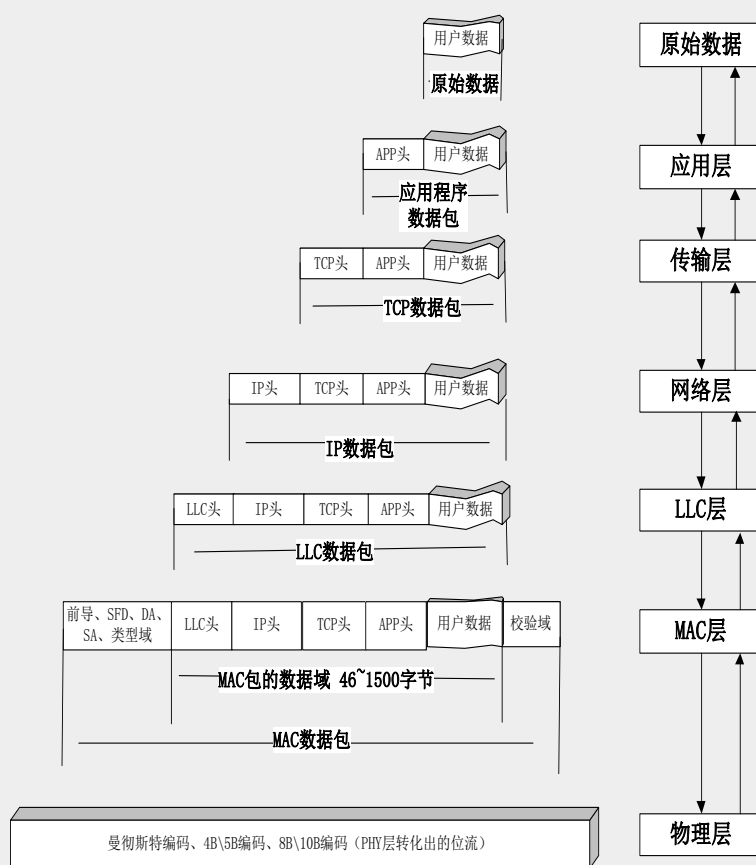
代码清单 18-1: CAN 例程中的 stm32f10x\_conf.h 文件配置

```
1.  /**
2.  ****
3.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author  MCD Application Team
5.  * @version V3.5.0
6.  * @date    08-April-2011
7.  * @brief   Library configuration file.
```

```
8.  *****/
9. #include "stm32f10x_can.h"
10. #include "stm32f10x_gpio.h"
11. #include "stm32f10x_rcc.h"
12. #include "stm32f10x_usart.h"
13. #include "misc.h"
```

### 18.3.3. main 文件

在本实验中，主机和从机的区别主要就体现在 main 函数中，即代码的执行流程不同。不同的 CAN 节点实际具有同等的地位，只是在本实验中为了方便区分而把它们分别称为主机和从机。本实验中的流程图如下：



#### ■ 主机

首先我们来分析主机的执行流程代码，见代码清单 18-2。

代码清单 18-2: CAN 例程的主机 main 函数

```
1. /*
```



```

2.  * 函数名: main
3.  * 描述   : "主机"的主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      /*初始化串口模块*/
10.     USART1_Config();
11.
12.     /* 配置 CAN 接口 */
13.     CAN_Config();
14.
15.     printf( "\r\n***** 这是一个双 CAN 通讯实验***** \r\n");
16.     printf( "\r\n 这是 “主机端” 的反馈信息:  \r\n");
17.
18.     /*设置要通过 CAN 发送的信息*/
19.     CAN_SetMsg();
20.
21.     printf("\r\n 将要发送的报文内容为: \r\n");
22.     printf("\r\n 扩展 ID 号 ExtId: 0x%x", TxMessage.ExtId);
23.     printf("\r\n 数据段的内容:Data[0]=0x%x ,
Data[1]=0x%x \r\n", TxMessage.Data[0], TxMessage.Data[1]);
24.
25.     /*发送消息 “ABCD”*/
26.     CAN_Transmit(CAN1, &TxMessage);
27.
28.
29.     while( flag == 0xff );           //flag =0 ,success
30.
31.     printf( "\r\n 成功接收到“从机”返回的数据\r\n ");
32.     printf("\r\n 接收到的报文为: \r\n");
33.     printf("\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId);
34.     printf("\r\n 数据段的内容:Data[0]= 0x%x ,
Data[1]=0x%x \r\n", RxMessage.Data[0], RxMessage.Data[1]);
35.
36.     while(1);
37.
38. }

```

在主机中，首先调用用户函数 `USART1_Config()`和 `CAN_Config()`初始化串口及配置 `CAN` 接口。接着调用用户函数 `CAN_SetMsg()`把我们的要发送的数据打包成报文并利用 `printf()`输出这个报文的 ID 及数据信息。生成报文后，调用库函数 `CAN_Transmit()`把这个报文广播到 `CAN` 网络上。发送完毕后进入等待状态，轮循标志变量 `flag` 直到它变 0。(这个变量在本 `CAN` 接口收到数据时在中断服务函数被赋值为 0)，本 `CAN` 接收器接到数据后，把接收到的报文信息打印到终端上。

## ■ 从机

从机的 main 函数代码见代码清单 18-3。

代码清单 18-3: CAN 例程的从机 main 函数

```
1.  /*
2.   * 函数名: main
3.   * 描述   : “从机”的主函数
4.   * 输入   : 无
5.   * 输出   : 无
6.   */
7.  int main(void)
8.  {
9.
10.     /* USART1 config */
11.     USART1_Config();
12.
13.     /* 配置 CAN 接口 */
14.     CAN_Config();
15.
16.     printf( "\r\n***** 这是一个双 CAN 通讯实验***** \r\n");
17.     printf( "\r\n 这是 “从机端” 的反馈信息:  \r\n");
18.
19.     /*等待主机端的数据*/
20.     while( flag == 0xff );
21.
22.     printf( "\r\n 成功接收到“主机”返回的数据\r\n ");
23.     printf( "\r\n 接收到的报文为: \r\n");
24.     printf( "\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId);
25.     printf( "\r\n 数据段的内容: Data[0]= 0x%x ,
Data[1]=0x%x \r\n", RxMessage.Data[0], RxMessage.Data[1]);
26.
27.     /*设置要通过 CAN 发送的信息*/
28.     CAN_SetMsg();
29.
30.     printf( "\r\n 将要发送的报文内容为: \r\n");
31.     printf( "\r\n 扩展 ID 号 ExtId: 0x%x", TxMessage.ExtId);
32.     printf( "\r\n 数据段的内容: Data[0]=0x%x ,
Data[1]=0x%x \r\n", TxMessage.Data[0], TxMessage.Data[1]);
33.
34.     /*发送消息 “CDAB”*/
35.     CAN_Transmit(CAN1, &TxMessage);
36.
37.     while(1);
38.
39. }
```

从机中配置好了串口和 CAN 接口后, 就开始等待标志变量 flag 被更新(接收到报文), 若接收到报文则把报文的信息打印到终端上, 接下来成为发送接点, 向 CAN 网络广播“DCBA”数据。主机若接收到该报文则会在主机端的 USART 打印报文信息。

#### 18.3.4. 配置 CAN 接口

了解代码的执行流程后，我们仔细分析在 main 函数中调用的函数。被调用的 CAN\_Config() 具有初始化 CAN 控制器的 GPIO 初始化、CAN 中断、CAN 模式、CAN 接收过滤器 的功能，这几个功能又被野火封装成一个个模块。CAN\_Config() 函数及它所调用的函数模块，在本实验的主、从机工程中都是完全一样的。它的实现见代码清单 18-4。

代码清单 18-4: CAN\_Config() 函数

```
1.  /*
2.  * 函数名: CAN_Config
3.  * 描述   : 完整配置 CAN 的功能
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void CAN_Config(void)
9. {
10.     CAN_GPIO_Config();
11.     CAN_NVIC_Config();
12.     CAN_Mode_Config();
13.     CAN_Filter_Config();
14. }
```

#### ■ GPIO 初始化及其重映射功能

CAN\_Config() 函数调用了 CAN\_GPIO\_Config() 函数开启了 GPIO 时钟、CAN 外设时钟，并把 CAN 复用的引脚配置成相应的模式。见代码清单 18-5。

代码清单 18-5: CAN\_GPIO\_Config() 函数

```
1.  /*
2.  * 函数名: CAN_GPIO_Config
3.  * 描述   : CAN 的 GPIO 配置, PB8 上拉输入, PB9 推挽输出
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void CAN_GPIO_Config(void)
9. {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* 外设时钟设置 */
13.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph
        _GPIOB, ENABLE);
```

```

14.   RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
15.
16.   /*IO 设置*/
17.   GPIO_PinRemapConfig(GPIO_Remap1_CAN1, ENABLE);
18.
19.   /* Configure CAN pin: RX PB8*/
20.
21.   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
22.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;    // 上拉输
入
23.   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
24.   GPIO_Init(GPIOB, &GPIO_InitStructure);
25.
26.   /* Configure CAN pin: TX PB9 */
27.
28.   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
29.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  // 复用推挽
输出
30.   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
31.   GPIO_Init(GPIOB, &GPIO_InitStructure);
32. }

```

关于 GPIO 的复用功能引脚号及其复用功能配置，见表 18-2 和表 18-3。

表 18-2 GPIO 引脚复用功能

| 脚位 |   |   |   |   |    | 管脚名称 | 管型  | I/O电平 | 管脚功能 | 可选的复用功能          |                     |
|----|---|---|---|---|----|------|-----|-------|------|------------------|---------------------|
|    |   |   |   |   |    |      |     |       |      | 默认复用功能           | 重映射功能               |
| 5  | 4 | 5 | 1 | 5 | 39 | PB8  | I/O | T     | PB8  | TIM4_CH3/SDIO_D4 | I2C1_SCL/<br>CAN_RX |
| 5  | 4 | 6 | 2 | 6 | 40 | PB9  | I/O | T     | PB9  | TIM4_CH4/SDIO_D5 | I2C1_SDA/<br>CAN_TX |

表 18-3 CAN 的 GPIO 复用配置

| BxCAN 引脚 | GPIO 配置    |
|----------|------------|
| CAN_TX   | 推挽复用输出     |
| CAN_RX   | 浮空输入或带上拉输入 |

代码根据手册的说明把 PB8 配置成上拉输入，PB9 配置成复用推挽输出。敏感的读者会发现，PB8 及 PB9 的默认复用功能是定时器 TIM 或 SDIO，GPIO 的重映射功能才是用作 CAN 的接口。

当我们使用 GPIO 的重映射功能时，调用 GPIO 重映射库函数 GPIO\_PinRemapConfig()开启此功能(代码的 17 行)，并开启 AFIO 时钟(代码的第 13 行)。

复用功能重映射可查询《STM32 参考手册》的 GPIO 的重映射章节。CAN 有三种映，见表 18-4。

表 18-4 GPIO 的 CAN 重映射表

| 复用<br>功能               | CAN_REMAP[<br>1:0]="00" | CAN_REMAP<br>[1:0]="10" | CAN_REMAP[<br>1:0]="11" |
|------------------------|-------------------------|-------------------------|-------------------------|
| CAN1<br>_RX 或<br>AN_RX | PA11                    | PB8                     | PD0                     |
| CAN1<br>_TX 或<br>AN_TX | PA12                    | PB9                     | PD1                     |

在代码第 17 行调用的 GPIO\_PinRemapConfig()函数中我们使用的参数为 GPIO\_Remap1\_CAN1，这个参数正是选择把 CAN 信号映射到 PB8、PB9 的 GPIO 引脚上。重映射复用功能对软件编写并无其它的影响，可见也没有难度。但这样的映射功能却为硬件布线提供了很大的便利，这也是 ST 公司设计重映射功能原因。

我们再来看看它的 GPIO\_PinRemapConfig()的库函数说明，见图 18-12。

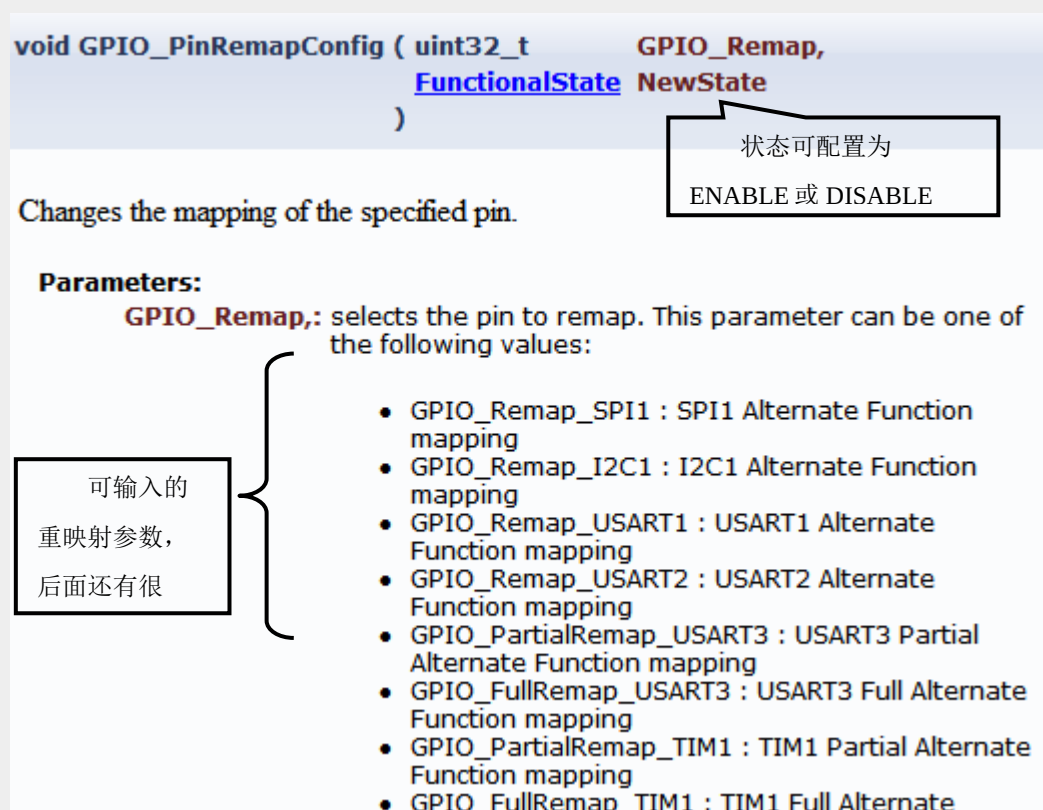


图 18-12 重映射库函数说明

## ■ NVIC 配置

本实验中使用了 CAN 的接收中断，因而需要配置中断向量，见代码清单 18-6。

代码清单 18-6: 配置 CAN 的中断向量

```

1.  /*
2.  * 函数名: CAN_NVIC_Config
3.  * 描述   : CAN 的 NVIC 配置,第 1 优先级组, 0, 0 优先级
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void CAN_NVIC_Config(void)
9. {
10.     NVIC_InitTypeDef NVIC_InitStructure;
11.     /* Configure one bit for preemption priority */
12.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
13.     /*中断设置*/ /*CAN1_RX0 中断 */
14.     NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;

15.
16.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    //抢占优先级 0

```

```
17.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //子优先  
级为0  
18.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  
19.     NVIC_Init(&NVIC_InitStructure);  
20. }
```

中断向量配置在 **EXTI** 章节中已详细解释，与本实验中的主要为 **NVIC** 的通道配置区别。本实验中的第 12 行的 **.NVIC\_IRQChannel** 被赋值为 **USB\_LP\_CAN1\_RX0\_IRQn**，即 **CAN** 的 **RX0** 中断。这个中断号 **USB\_LP\_CAN1\_RX0\_IRQn** 可以从启动文件 **startup\_stm32f10x\_hd.s** 中的中断向量列表中查找到。

那么 **RX0** 中断又是什么呢？见图 18-13。

**CAN** 的中断由发送中断、接收 **FIFO** 中断、错误中断构成。发送中断由三个发送邮箱任意一个为空的事件构成。接收 **FIFO** 中断分为 **FIFO0** 和 **FIFO1** 的中断，接收 **FIFO** 收到新的报文或报文溢出的事件可以引起中断。

本实验中使用的 **RX0** 中断通道即为 **FIFO0** 中断通道，当 **FIFO0** 收到新报文时，引起中断，我们就在相应的中断服务函数读取这个新报文。

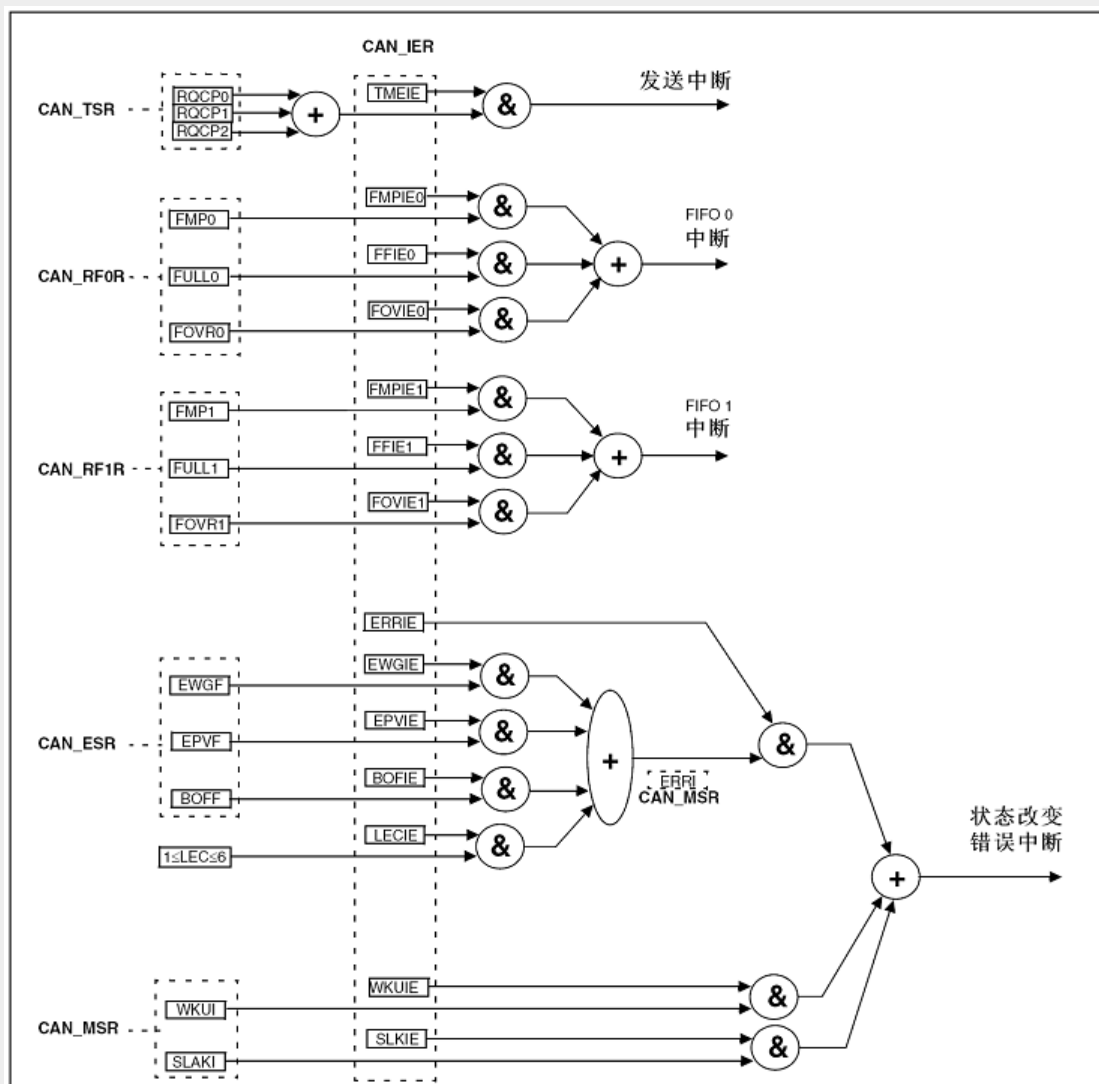


图 18-13 CAN 中断示意图

## ■ CAN 模式配置

CAN 的模式我们是通过调用用户函数 `CAN_Mode_Config()`来实现的，见代码清单 18-7。

### 代码清单 18-7: `CAN_Mode_Config()`函数

```
1.  /*
2.  * 函数名: CAN_Mode_Config
3.  * 描述   : CAN 的模式 配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 内部调用
7.  */
8. static void CAN_Mode_Config(void)
```



```
9. {
10.     CAN_InitTypeDef          CAN_InitStructure;
11.     /******CAN 通信参数设置******/
12.     /*CAN 寄存器初始化*/
13.     CAN_DeInit(CAN1);
14.     CAN_StructInit(&CAN_InitStructure);
15.     /*CAN 单元初始化*/
16.     CAN_InitStructure.CAN_TTCM=DISABLE;
17. /*MCR-TTCM 关闭时间触发通信模式使能*/
18.     CAN_InitStructure.CAN_ABOM=ENABLE;
19. /*MCR-ABOM 自动离线管理*/
20.     CAN_InitStructure.CAN_AWUM=ENABLE;
21. /*MCR-AWUM 自动唤醒模式*/
22.     CAN_InitStructure.CAN_NART=DISABLE;
23. /*MCR-NART 禁止报文自动重传 DISABLE=自动重传*/
24.     CAN_InitStructure.CAN_RFLM=DISABLE;
25. /*MCR-RFLM 接收 FIFO 锁定模式 DISABLE-溢出时新报文会覆盖原有报文
*/
26.     CAN_InitStructure.CAN_TXFP=DISABLE;
27. /*MCR-TXFP 发送 FIFO 优先级 DISABLE-优先级取决于报文标示符*/
28.     CAN_InitStructure.CAN_Mode = CAN_Mode_Normal; /*正常工作模式
*/
29.     CAN_InitStructure.CAN_SJW=CAN_SJW_2tq;
30. /*BTR-SJW 重新同步跳跃宽度 2 个时间单元*/
31.     CAN_InitStructure.CAN_BS1=CAN_BS1_6tq;
32. /*BTR-TS1 时间段 1 占用了 6 个时间单元*/
33.     CAN_InitStructure.CAN_BS2=CAN_BS2_3tq;
34. /*BTR-TS1 时间段 2 占用了 3 个时间单元*/
35.     CAN_InitStructure.CAN_Prescaler =4;
36. /*BTR-BRP 波特率分频器 定义了时间单元的时间长
度 36/(1+6+3)/4=0.8Mbps*/
37.     CAN_Init(CAN1, &CAN_InitStructure);
38. }
```

STM32 的 CAN 接口具有非常强大的功能，因而它的初始化结构体也有大量的参数可供配置。

CAN 初始化结构体具有以下成员：

#### (1) .CAN\_TTCM

本成员用于配置 CAN 的时间触发通信模式(Time triggered communication mode)，在此模式下，CAN 使用它内部定时器产生时间戳，被保存在 CAN\_RDTxR、CAN\_TDTxR 寄存器中。内部定时器在每个 CAN 位时间累加，在接收和发送的帧起始位被采样，并生成时间戳。

本实验不使用时间触发模式。

#### (2) .CAN\_ABOM

当 CAN 检测到发送错误(TEC)或接收错误(REC)超过一定值时，会自动进入离线状态。在离线状态中，CAN 不能接收或发送报文。其中的发送错误或接收

错误的计算原则由 CAN 协议规定，是 CAN 硬件自动检测的，不需要我们软件干预。我们软件可干预的是通过此 CAN\_ABOM 参数选择是否使用自动离线管理(Automatic bus-off management)，决定 CAN 硬件在什么条件下可以退出离线状态。

若我们把此成员赋值为 **ENABLE**，则使用硬件自动离线管理。一旦硬件检测到 128 次 11 位连续的隐性位，则自动退出离线状态。

若我们把此成员赋值为 **DISABLE**，离线状态由软件管理，首先由软件要对 CAN\_MCR 寄存器的 INRQ 位进行置'1'随后清'0'，再等到硬件检测到 128 次 11 位连续的隐性位，才退出离线状态。

本实验使用硬件自动离线管理。

### (3) .CAN\_AWUM

本成员选择是否开启自动唤醒功能(Automatic wakeup mode)。若使能了自动唤醒功能，并且 CAN 处于睡眠模式，检测到 CAN 总线活动时，会自动进入正常模式，以便收发数据。若禁止此功能，则只能由软件配置才可以使 CAN 退出睡眠模式。

本实验使用自动唤醒模式(ENABLE)。

### (4) .CAN\_NART

本成员用于选择是否禁止报文自动重传(No automatic retransmission)。按照 CAN 的标准，CAN 发送失败时会自动重传至成功为止。向本参数赋值 **ENABLE**，即禁止自动重传，若赋值为 **DISABLE**，则允许自动重传功能。

本实验允许 CAN 的自动重传(赋值 **DISABLE**)。

### (5) .CAN\_RFLM

本成员用于配置接收 FIFO 是否锁定(Receive FIFO locked mode)。若选择 **ENABLE** 则当 FIFO 溢出时，会丢弃下一个接收的报文。若选择 **DISABLE**，当 FIFO 溢出时，下一个接收到的报文会覆盖原报文。

本实验选择非锁定模式(DISABLE)。

### (6) .CAN\_TXFP

本成员用于选择 CAN 报文发送优先级的判定方法。STM32 的 CAN 接口可以对它邮箱内的几个将要发送的报文按照优先级来处理。对于这个优先级的判定可以设置为按照报文标识符来决定(DISABLE)，或按照报文的请求顺序来决定(ENABLE)。

本实验发送报文的优先级按照报文标识符来决定优先级。

#### (7) .CAN\_Mode

本成员用于选择 CAN 是处于工作模式状态还是测试模式状态。它有四个可赋值参数，分别是一个正常工作模式状态(CAN\_Mode\_Normal)和 静默模式(CAN\_Mode\_Silent)、回环模式(CAN\_Mode\_LoopBack)和 静默回环(CAN\_Mode\_Silent\_LoopBack)三个测试模式。

本实验使用的是正常的两个 CAN 节点间的通讯，所以向本成员赋值为正常工作模式。

#### (8) CAN\_SJW、CAN\_BS1、CAN\_BS2 及 CAN\_Prescaler

这几个成员是用来配置 CAN 通讯的位时序的。它们分别代表 CAN 协议中的 SJW 段(重新同步跳跃宽度)、PBS1 段(相位缓冲段 1)、PBS2 段(相位缓冲段 2)、及时钟分频。

在 STM32 的 CAN 接口配置中，SS 段(同步段)被固定为 1Tq，PTS(物理缓冲段)被省略。所以一个正常的位时间只由 SS 段(固定为 1Tq)、PBS1 段(CAN\_BS1 成员)及 PBS2 段(CAN\_BS2 成员)组成，PBS1 和 PBS2 的重新同步跳跃宽度由成员 CAN\_SJW 决定。而时间单元 Tq 则由 CAN\_Prescaler 成员决定，它决定 CAN 使用的时钟是由 APB1 的多少分频得到。

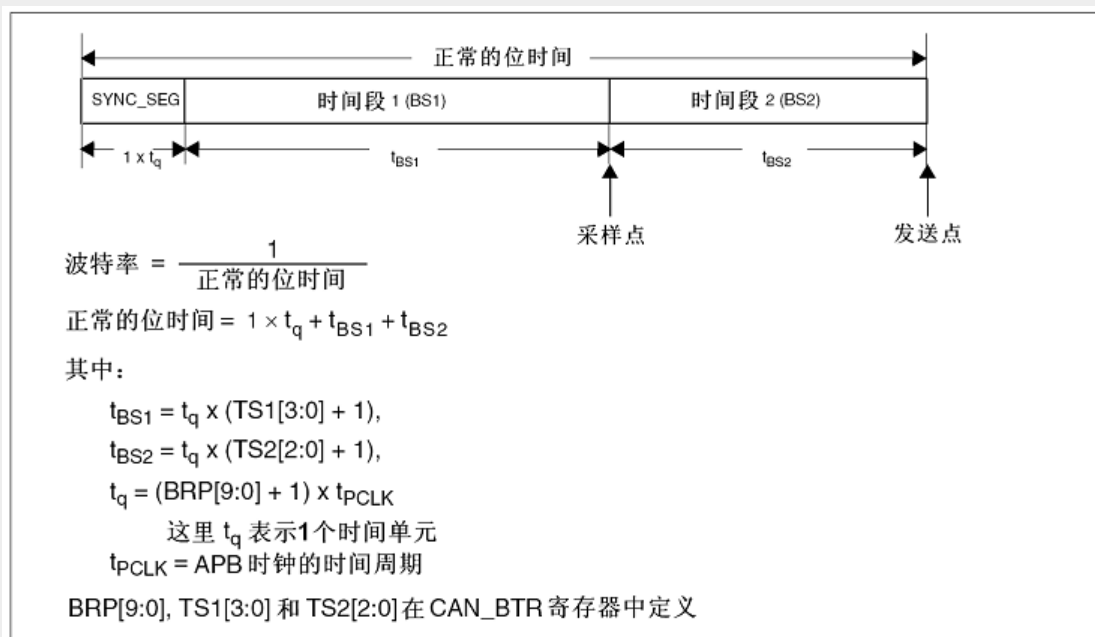


图 18-14 STM32 位时间段

我们知道 PBS1 与 PBS2 之间是采样点，一般配置在 75-80%的位置，保证总线上不同节点数据同步。在本实验中我们把采样点设置在 70%处。即  $SS=1Tq$ ， $CAN\_BS1 = 6Tq$ ， $CAN\_BS2=3Tq$ 。为提高同步调整的速度，把  $CAN\_SJW$  配置为  $2Tq$ 。利用图中的公式。时间单位  $Tq$  根据成员  $CAN\_Prescaler$  的值(4 分频)及  $APB1$  的时钟频率(36M)计算得出。 $Tq = 1/(36M/4)$  秒，即实际上每一个 CAN 位的时间为  $10Tq$  ( $1+6+3Tq$ )，波特率为  $36M/4/10 = 0.8Mbps$ 。

对于这些与 CAN 位时序相关的配置，在 CAN 网络上的各节点应设置成相同或相似以确保 CAN 通讯的正常。

配置完这些成员后，我们调用库函数 `CAN_Init()`把这些参数写进寄存器。

## ■ CAN 过滤器配置

在配置完 CAN 的 GPIO、中断、模式后，我们还要对 CAN 的过滤器进行配置，使接收 FIFO 只接收特定的报文。所谓的过滤就是 CAN 接口根据收到的报文的 ID，选择是否把该报文保存到接收 FIFO 中。

STM32 的 ID 过滤方式有两种，一种为标识符列表模式，它把要接收报文的 ID 列成一个表，要求报文 ID 与列表中的某一个标识符完全相同才可以接收，可以理解为白名单管理。

另一种称为标识符屏蔽模式，它把可接收报文 ID 的某几位作为列表，这几位被称为屏蔽位，可以把它理解成关键字搜索，只要屏蔽位(关键字)相同，就符合要求。即这种模式只要求报文 ID 的屏蔽位与列表中标识符相应屏蔽位相同，报文就被保存到接收 FIFO。见图 18-15。

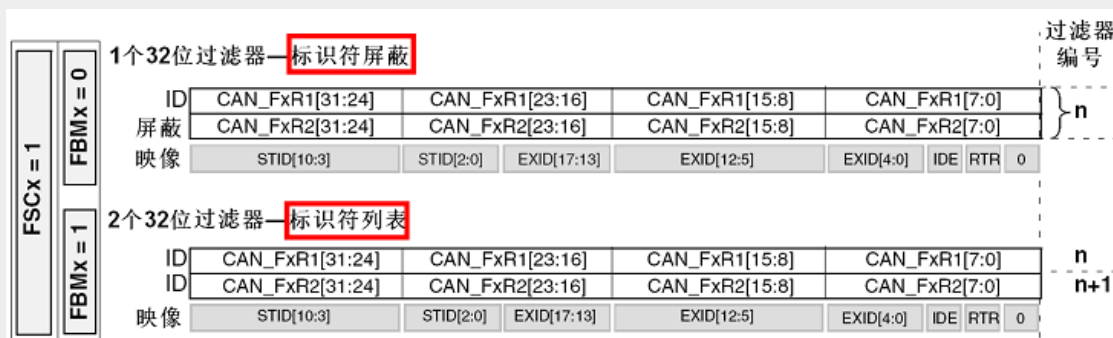


图 18-15 过滤器组

在非互联型号的 STM32 中的 CAN 接口具有 14 个过滤器组，每组过滤器由 2 个标识符寄存器组成。图中左端表示过滤器的配置，FSCx、FBMx 是属于过滤器的配置寄存器(CAN\_FS1R、CAN\_FM1R)中的寄存器位。FSCx 用来配置过滤器组 x 的寄存器长度为 16 位还是 32 位，本图只解释 32 位的情况，即 FSCx=1。而 FBMx 则用于选择过滤器组 x 是用作标识符屏蔽模式还是标识符列表模式。

在标识符屏蔽模式下，过滤器组 x 的第 1 个标识符寄存器(CAN\_FxR1)用来保存用于与报文 ID 比较的完整标识符。第 2 个标识符寄存器(CAN\_FxR2)则用来表示屏蔽位，即表明报文 ID 要与第 1 个标识符寄存器中的哪几位需要比较。

在标识符列表模式下，过滤器组 x 的 2 个标识符寄存器都用于保存完整的标识符。报文 ID 若与其中一个标识符完全相同，可被保存到接收 FIFO。

从过滤器的寄存器映像中看到，无论用作哪个模式，标识符寄存器的第 0 位保留、第 1 位为报文的 RTR 位，第 2 位是报文的 IDE 位，报文的扩展 ID 保存在第 3~20 位，而报文的标准 ID 则保存在第 21~32 位。使用的时候，我们要根据它的意义写入参数。

下面我们进入具体的代码进行分析，CAN 的过滤器设置在用户函数 CAN\_Filter\_Config()中实现，见代码清单 18-8。

代码清单 18-8: CAN\_Filter\_Config()函数

```
1.  /*
2.   * 函数名: CAN_Filter_Config
3.   * 描述   : CAN 的过滤器 配置
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 内部调用
7.   */
8. static void CAN_Filter_Config(void)
9. {
10.     CAN_FilterInitTypeDef  CAN_FilterInitStructure;
11.
12.     /*CAN 过滤器初始化*/
13.     CAN_FilterInitStructure.CAN_FilterNumber=0;           //过滤器组
14.     CAN_FilterInitStructure.CAN_FilterMode=CAN_FilterMode_IdMask;
15.     /*工作在标识符屏蔽位模式*/
16.     CAN_FilterInitStructure.CAN_FilterScale=CAN_FilterScale_32bit;
17.     /*过滤器位宽为单个 32 位。*/
18.     /* 使能报文标识符过滤器按照标识符的内容进行比对过滤，扩展 ID 不是如下的就
19.     抛弃掉，是的话，会存入 FIFO0。 */
20.     CAN_FilterInitStructure.CAN_FilterIdHigh= (((u32)0x1314<<3)
21.     &0xFFFF0000)>>16;           /*要过滤的 ID 高位*/
22.     CAN_FilterInitStructure.CAN_FilterIdLow= (((u32)0x1314<<3) |
23.     CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF; /*要过滤的 ID 低位*/
24.     CAN_FilterInitStructure.CAN_FilterMaskIdHigh= 0xFFFF;
25.     /*过滤器高 16 位每位必须匹配*/
26.     CAN_FilterInitStructure.CAN_FilterMaskIdLow= 0xFFFF;
27.     /*过滤器低 16 位每位必须匹配*/
28.     CAN_FilterInitStructure.CAN_FilterFIFOAssignment=CAN_Filter_
29.     FIFO0 ; /*过滤器被关联到 FIFO0*/
30.     CAN_FilterInitStructure.CAN_FilterActivation=ENABLE;
31.     /*使能过滤器*/
32.     CAN_FilterInit(&CAN_FilterInitStructure);
33.     /*CAN 通信中断使能*/
34.     CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
35. }
```

过滤器的初始化是使用过滤器初始化结构体与库函数 CAN\_FilterInit()实现的。过滤器初始化结构体成员如下：

#### (1) .CAN\_FilterNumber

本成员用于选择要配置的过滤器组，其参数值可以为 0~13，分别与过滤器组 0~13 一一对应。

本实验选择过滤器组 0。

## (2) .CAN\_FilterMode

本成员用于配置过滤器的工作模式，分别为标识符列表模式(CAN\_FilterMode\_IdList)和标识符屏蔽模式(CAN\_FilterMode\_IdMask)。

本实验使用标识符屏蔽模式。

## (3) .CAN\_FilterScale

本成员用于配置过滤器的长度，可以分别设置为 16 位(CAN\_FilterScale\_16bit)和 32 位(CAN\_FilterScale\_32bit)。

本实验设置跟过滤器解说图中的一样，为 32 位模式。

## (4) CAN\_FilterIdHigh 和 CAN\_FilterIdLow

这两个成员分别为过滤器组中的第 1 个标识符寄存器的高 16 位和低 16 位。

本实验中配置要接收的报文使用了扩展 ID，其标准 ID 部分(11 位)为 0x000，其扩展部分(18 位)ID 为 0x01314。

IDE 位为隐性位(宏 CAN\_ID\_EXT: (uint32\_t)0x00000004)并且报文为数据帧，即 RTR 位为显性位(宏 CAN\_RTR\_DATA: (uint32\_t)0x00000000)。

我们把这些信息整理好，写入到相应的寄存器位中，使过滤器比较的报文 ID 为扩展格式的数据帧。赋值时标识符 0x0000 1314 左移三位是因为寄存器的低三位意义为保留位、RTR 位和 IDE 位。

```
1. CAN_FilterIdHigh= (((u32)0x1314<<3)&0xFFFF0000)>>16
```

CAN\_FilterIdHigh 成员中(0x00001314)左移后再与 0xFFFF0000 作“&”运算是为了取这个标识符的高 16 位，最后再右移 16 位赋值给 CAN\_FilterIdHigh 成员。调用库函数时，函数会把本成员的值写入到标识符寄存器的高 16 位。

```
1. CAN_FilterIdLow= (((u32)0x1314<<3)|CAN_ID_EXT|CAN_RTR_DATA)&0xFFFF
```



`CAN_FilterIdLow` 成员的赋值运算类似, `(0x00001314)` 左移后与宏 `CAN_ID_EXT` 和宏 `CAN_RTR_DATA` 作“|”运算, 即把报文的 IDE 位、RTR 位也加入到过滤器进行过滤比较。

#### (5) `CAN_FilterMaskIdHigh` 和 `CAN_FilterMaskIdLow`

这两个成员为过滤器组中的第 2 个标识符寄存器的高 16 位与低 16 位。若在标识符列表模式中, 这个寄存器保存的内容也是用于过滤的标识符。在标识符屏蔽模式下, 这个寄存器保存的内容是屏蔽位。

本实验中过滤器模式是屏蔽模式, 本寄存器中, 值为 1 的寄存器位就是屏蔽位, 值为 0 的寄存器位不参与比较。如我们向这两个成员的参数赋值都为 `0xFFFF`, 即本寄存器位的值全部为 1, 表示我们在上面配置的 `CAN_FilterIdHigh` 和 `CAN_FilterIdLow` 每 1 位都要进行比较。其实在正常的屏蔽位模式使用中, 我们一般不会要求所有标识符都参与比较的(因为这样的比较结果与标识符列表模式相同), 而是选择某些位进行比较, 从而过滤出一组报文。

#### (6) `.CAN_FilterFIFOAssignment`

本成员用于设置过滤器与接收 FIFO 的关联, 即过滤成功后, 报文的存储位置。可配置为 `FIFIO0(CAN_Filter_FIFO0)` 和 `FIFO1(CAN_Filter_FIFO1)` 两个接收位置。

本实验设置存储位置为 `FIFO0`。

#### (7) `.CAN_FilterActivation`

本成员用于使能或关闭过滤器, 默认下它为关闭的。因而使用过滤器时, 我们要向它赋值 `ENABLE`。

对过滤器成员参数赋值完毕后我们调用库函数 `CAN_FilterInit()` 将它们写入寄存器。在本实验中由于我们是使用中断来读取 FIFO 数据的, 所以在函数 `CAN_Filter_Config()` 的最后我们调用库函数 `CAN_ITConfig()` 来开启 `FIFO0` 消息挂号中断使能(`CAN_IT_FMP0`), 在 CAN 接口的 `FIFO0` 收到报文时, 我们在中断服务函数中从 `FIFO0` 读数据到内存。库函数 `CAN_ITConfig()` 说明如图 18-16。



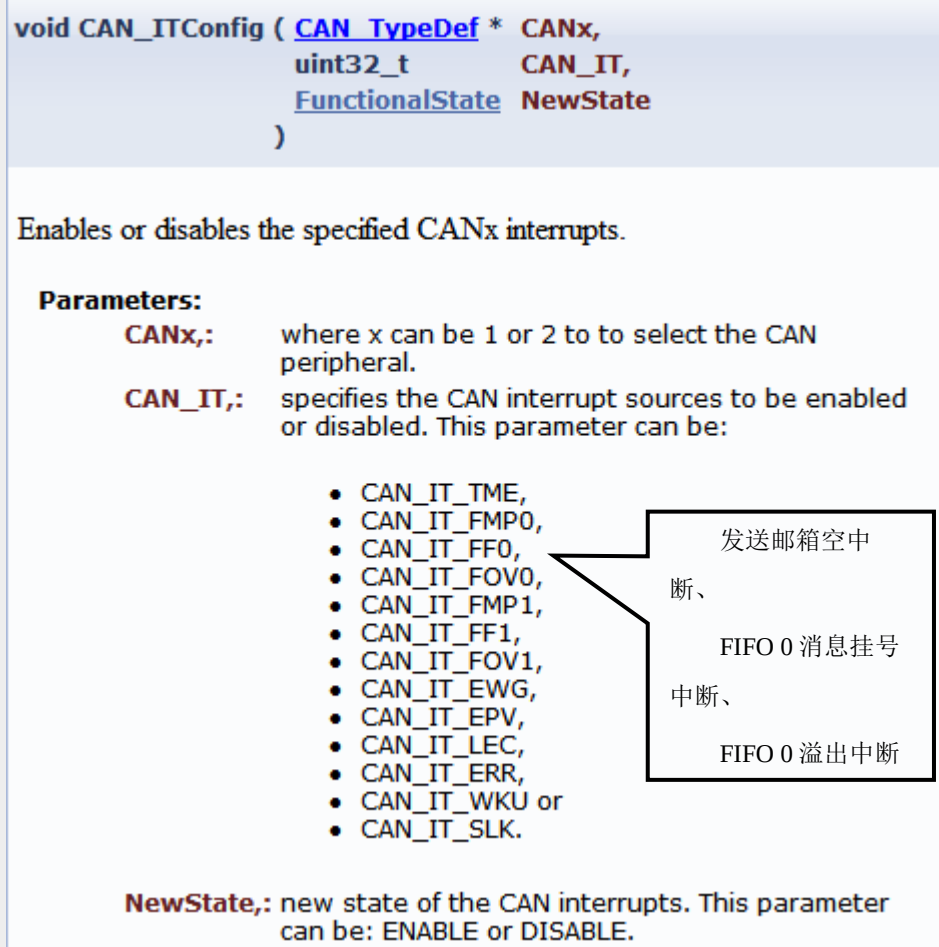


图 18-16 库函数 CAN\_ITConfig()说明

### 18.3.5. 打包报文

配置好 CAN 接口后，我们就可以复用它来发送数据了。利用 CAN 发送数据，要先把数据打包成完整的 CAN 报文格式。我们使用用户函数 CAN\_SetMsg() 来实现，见代码清单 18-9。

代码清单 18-9: CAN\_SetMsg()函数

```
1. /*
2.  * 函数名: CAN_SetMsg
3.  * 描述   : “主机”的 CAN 通信报文内容设置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void CAN_SetMsg(void)
9. {
10. // TxMessage.StdId=0x00;
```

```

11. TxMessage.ExtId=0x1314;           //使用的扩展 ID
12. TxMessage.IDE=CAN_ID_EXT;        //扩展模式
13. TxMessage.RTR=CAN_RTR_DATA;      //发送的是数据
14. TxMessage.DLC=2;                 //数据长度为 2 字
节
15. TxMessage.Data[0]=0xAB;
16. TxMessage.Data[1]=0xCD;
17. }

```

若清楚 CAN 协议的数据帧报文格式，阅读本代码就没有难度了。本函数使用的结构体变量 TxMessage 在 main 文件以全局变量的形式定义，见代码清单 18-10。

代码清单 18-10：结构体变量 TxMessage 和 RxMessage

```

1. CanTxMsg TxMessage;           //发送缓冲区
2. CanRxMsg RxMessage;          //接收缓冲区

```

TxMessage 的类型为 CanTxMsg，而接收报文时，我们使用 CanRxMsg 类型。它们都是由库文件定义的结构体类型。

我们先以 CAN\_SetMsg() 函数分析怎样利用类型 CanTxMsg 打包发送报文，它有如下成员：

#### (1) .StdId

本成员保存的是报文的标准 ID。在本实验中我们使用扩展 ID，所以不需要对它赋值。

#### (2) .ExtId

报文的扩展 ID。在实验中我们设置为 0x1314。

#### (3) .IDE

本成员用于设置 CAN 报文的 IDE 位，即使用标准 ID(CAN\_Id\_Standard) 还是使用扩展 ID(CAN\_ID\_EXT)。本实验使用扩展 ID。

#### (4) .RTR

本成员用于设置 CAN 报文的 RTR 位，即确定是数据帧(CAN\_RTR\_DATA) 还是遥控帧(CAN\_RTR\_REMOTE)。本实验发送的为数据帧。

#### (5) .DLC

用于设置 CAN 报文控制段中的 DLC 段，DLC 的大小为数据段的字节长度，可取值范围为 0~8。本实验要发送的数据为 2 个字节，所以向 DLC 赋值为 2。

#### (6) Data[0]~Data[7]

这些 Data 数组即为 CAN 报文中数据段的内容，Data 的数量要与 DLC 段的一致。

本实验中主机的数据段设置为 Data[0]=0xAB，Data[1]=0xCD。在从机的数据段则设置为 Data[0]=0xDC，Data[1]=0xBA。

从机端的报文打包函数如下，与主机仅有数据段的内容区别，见代码清单 18-11。

代码清单 18-11：从机端报文打包函数

```
1.  /*
2.  * 函数名: CAN_SetMsg
3.  * 描述   : “从机”CAN 通信报文内容设置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void CAN_SetMsg(void)
9. {
10. //TxMessage.StdId=0x00;
11. TxMessage.ExtId=0x1314;           //使用的扩展 ID
12. TxMessage.IDE=CAN_ID_EXT;       //扩展模式
13. TxMessage.RTR=CAN_RTR_DATA;     //发送的是数据
14. TxMessage.DLC=2;                //数据长度为 2 字节
15. TxMessage.Data[0]=0xDC;
16. TxMessage.Data[1]=0xBA;
17. }
```

#### 18.3.6. 发送报文

回到主机的 main 函数中，在初始化完毕后，调用了函数 CAN\_SetMsg()来打包报文，然后调用了库函数 CAN\_Transmit()，由这个函数把我们打包好的报文发送到 CAN 网络。见代码清单 18-12。

代码清单 18-12：发送报文到 CAN 网络

```
1. /*发送消息 "ABCD"*/  
2. CAN_Transmit(CAN1, &TxMessage);
```

其库函数说明如图 18-17。

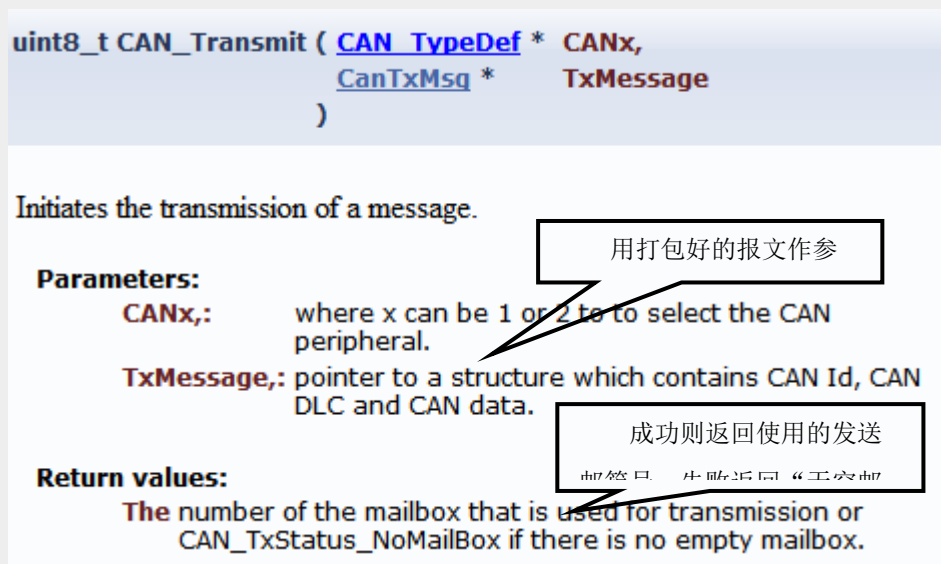


图 18-17 库函数 CAN\_Transmit 说明

### 18.3.7. 接收报文、编写中断服务函数

根据我们的实验配置，主机初始化完成后发送报文，而从机初始化完成后一直等待接收报文。我们来看从机端是如何接收的。

在从机的 main 函数中，初始化完成后就一直轮询标志变量 flag，见代码清单 18-13。

代码清单 18-13：轮询标志变量 flag

```
1. /*等待主机端的数据*/  
2. while( flag == 0xff );  
3.  
4. printf( "\r\n 成功接收到“主机”返回的数据\r\n " );  
5. printf( "\r\n 接收到的报文为: \r\n" );  
6. printf( "\r\n 扩展 ID 号 ExtId: 0x%x", RxMessage.ExtId );
```

当从机的 FIFO0 收到报文时，将会进入中断，在中断服务函数中我们把 flag 的值修改，中断服务函数结束后，main 函数中这句轮询代码因 flag 的改变而结束，执行紧接下来的 printf() 打印信息到串口。

我们来看看从机的中断服务函数是如何编写的，见代码清单 18-14。

## 代码清单 18-14: 从机中断服务函数

```
1. void USB_LP_CAN1_RX0_IRQHandler(void)
2. {
3.
4.     CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
5.     /* 比较是否是发送的数据和 ID */
6.     if((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT)
7.         && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xABCD))
8.     {
9.         flag = 0;                                //接收成功
10.    }
11. }
12. else
13. {
14.     flag = 0xff;                                //接收失败
15. }
16. }
```

我们可以从启动文件 `startup_stm32f10x_hd.s` 中查得 CAN1 的 RX0 的中断服务函数名为 `USB_LP_CAN1_RX0_IRQHandler`，然后在 `stm32f10x_it.c` 文件中添加这个函数。

在中断服务函数中，我们调用了库函数 `CAN_Receive()` 把 FIFO0 的报文读取到在 main 文件中的 `CanRxMsg` 类型全局变量 `RxMessage` 中。`CAN_Receive()` 库函数说明图 18-18。

```
void CAN_Receive ( CAN_TypeDef * CANx,
                  uint8_t      FIFONumber,
                  CanRxMsg *    RxMessage
                  )
```

Receives a message.

**Parameters:**

- CANx,:** where x can be 1 or 2 to select the CAN peripheral.
- FIFONumber,:** Receive FIFO number, CAN\_FIFO0 or CAN\_FIFO1.
- RxMessage,:** pointer to a structure receive message which contains CAN Id, CAN DLC, CAN datas and FMI number.

三个参数分别为：CAN 号，接收 FIFO 号，以及保存报文的结构体地址

图 18-18 CAN\_Receive 库函数说明

接收报文结构体 `CanRxMsg` 类型与报文的结构体很类似，见代码清单 18-15。

代码清单 18-15: 结构体 CanRxMsg 的定义

```
1. typedef struct
2. {
3.     uint32_t StdId; /*接收报文的标准 ID */
4.     uint32_t ExtId; /*接收报文的扩展 ID */
5.     uint8_t IDE; /*报文的 IDE 位 */
6.     uint8_t RTR; /*报文的 RTR 位 */
7.     uint8_t DLC; /*报文的 DLC 段 */
8.     uint8_t Data[8]; /*报文的数据段 */
9.     uint8_t FMI; /*过滤器匹配序号 */
10. } CanRxMsg;
```

使用 CAN\_Receive()函数接收了报文后, 我们使用 if 语句判断接收到的报文的 ID 信息、IDE 位、DLC 位及数据段是否等于 0xABCD, 若这接收到的报文跟我们主机预定发送的报文一样, 则把 flag 的位置 0, 退出中断服务函数。之后由 main 函数中的 printf()把接收到的报文信息打印到终端。

这样就完成了一次从主机发送报文到从机的过程。第一个主机发送过程结束后, 主机进入轮询状态, 等待从机的向它发送数据。而从机把接收到的报文打印到终端后, 打包好数据段为 0xDCBA 的报文发送到 CAN 网络。(这部分的流程详见主从机的 main 函数代码)。当主机收到报文后, 进入它的中断服务函数, 判断接收是否正确, 若正确则把 flag 置为 0, 退出中断函数, 由 main 的 printf()打印报文信息到终端, 实验结束。其中主机的中断服务函数见代码清单 18-16。

代码清单 18-16: 主机中断服务函数

```
1. void USB_LP_CAN1_RX0_IRQHandler(void)
2. {
3.
4.     /*从邮箱中读出报文*/
5.     CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
6.
7.     /* 比较 ID 和数据是否为 0x1314 及 DCBA */
8.     if((RxMessage.ExtId==0x1314) && (RxMessage.IDE==CAN_ID_EXT)
9.
10.         && (RxMessage.DLC==2) && ((RxMessage.Data[1]|RxMessage.Data[0]<<8)==0xDCBA))
11.     {
12.         flag = 0; /*接收成功
13.     }
14.     else
15.     {
16.         flag = 0xff; /*接收失败
17.     }
```

它与从机中断服务函数的区别仅为数据段的内容判别，为 0xDCBA。

#### 18.3.8. 实验小结

在本小节总结一下 STM32 的 CAN 通讯配置。

- (1) 初始化 CAN 的 GPIO。
- (2) 配置好 CAN 的工作模式，主要是对 CAN 的位时序的配置。
- (3) 若需要过滤报文，要设置 CAN 的过滤器，根据需求选用标识符列表或标识符屏蔽模式。
- (4) 在发送报文前需要先对报文打包。

只要对 CAN 协议有清晰的了解，STM32 的 CAN 配置实际上并不复杂，野火建议读者进一步阅读 CAN2.0 的标准协议。

#### 18.3.9. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序分别下载到两块开发板，根据连接图把两块板子的 CAN 信号线连接起来，先开启从机的电源，再开启主机的电源(这是由我们实验代码的执行流程决定的)。

若串口线接到“主机”板子上，超级终端打印出如图 18-19 信息：

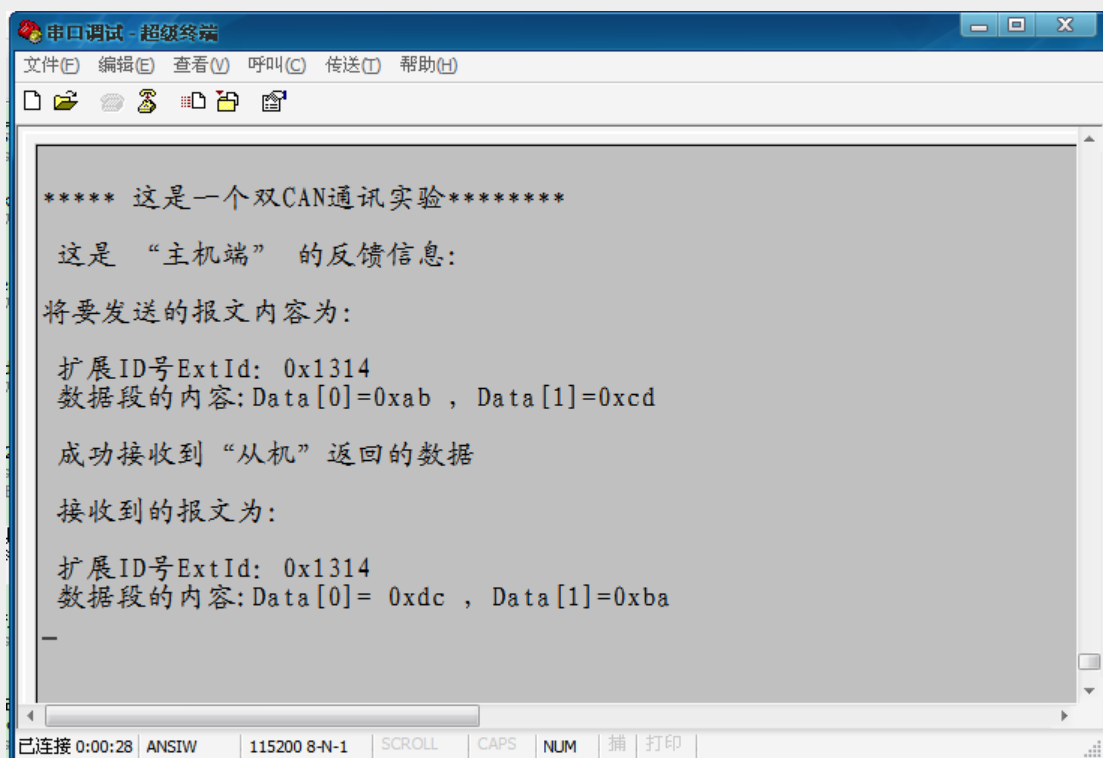


图 18-19 双 CAN 通讯主机端信息

若串口线接到 “从机” 板上，超级终端打印出如图 18-20 信息。

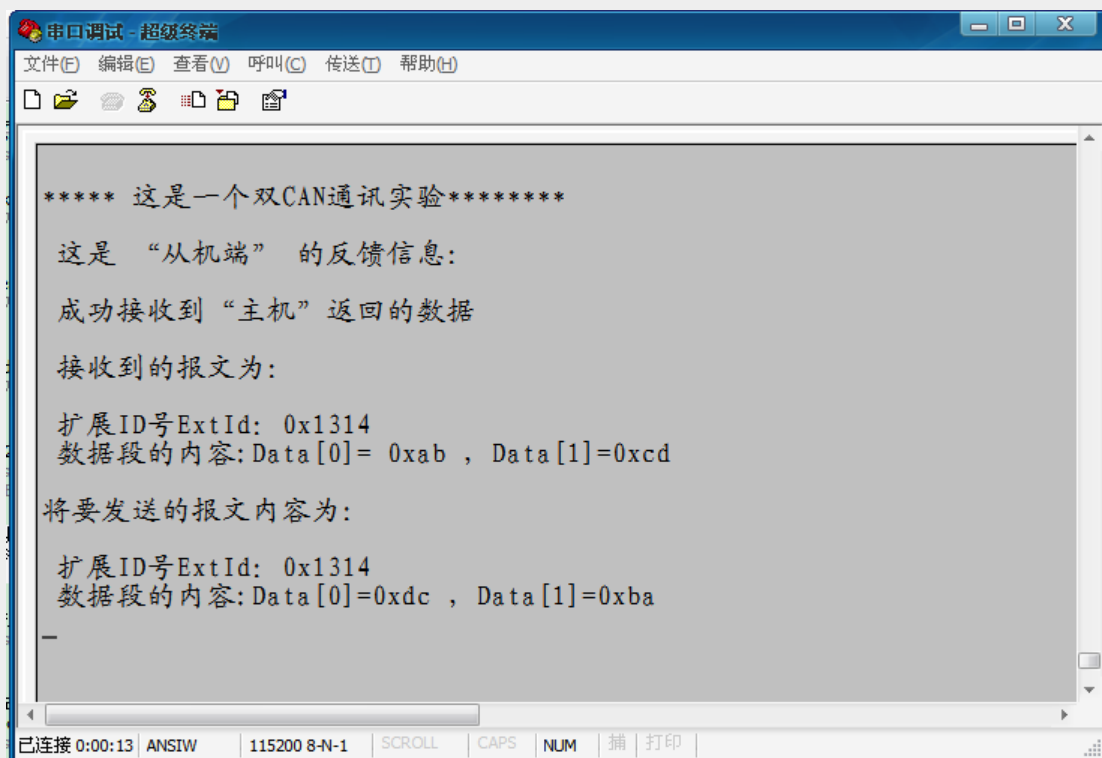


图 18-20 双 CAN 通讯从机端信息



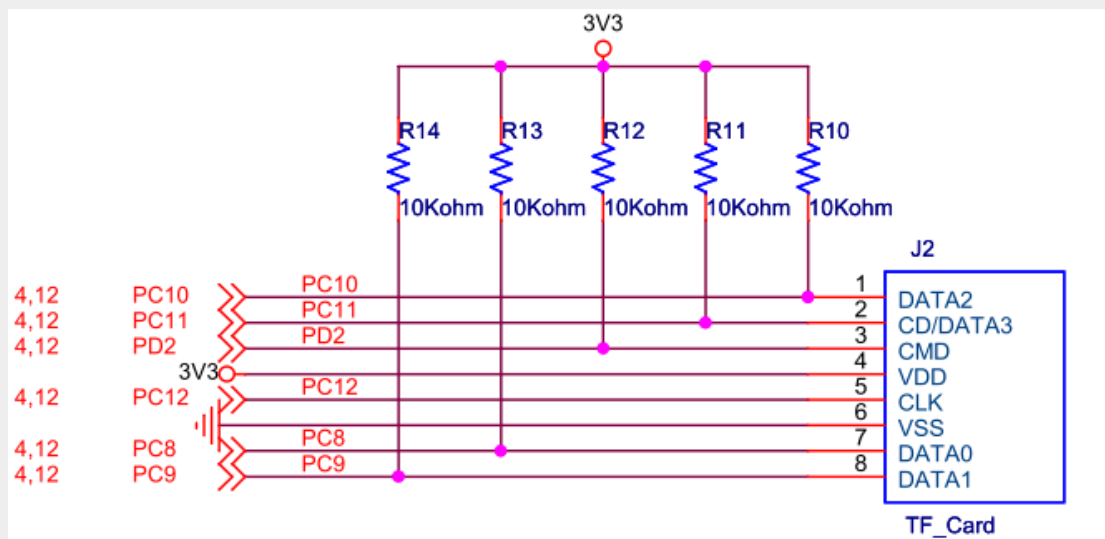


## 19. 1、SDIO-SD 卡

### 19. 1. 1.1 实验描述及工程文件清单

|         |                                                                                                                                                                                                                                  |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | MicroSD 卡(SDIO 模式)测试实验，采用 4bit 数据线模式。没有跑文件系统，只是单纯地读 block 并将测试信息通过串口 1 在电脑的超级终端上 打印出来。                                                                                                                                           |
| 硬件连接    | PC12-SDIO-CLK: CLK<br>PC10-SDIO-D2 : DATA2<br>PC11-SDIO-D3: CD/DATA3<br>PD2-SDIO-CMD : CMD<br>PC8-SDIO-D0: DATA0<br>PC9-SDIO-D1: DATA1                                                                                           |
| 用到的库文件  | startup/start_stm32f10x_hd.c<br>CMSIS/core_cm3.c<br>CMSIS/system_stm32f10x.c<br>FWlib/stm32f10x_gpio.c<br>FWlib/stm32f10x_rcc.c<br>FWlib/stm32f10x_usart.c<br>FWlib/ stm32f10x_sdio.c<br>FWlib/ stm32f10x_dma.c<br>FWlib/ misc.c |
| 用户编写的文件 | USER/main.c<br>USER/stm32f10x_it.c<br>USER/usart1.c<br>USER/ sdio_sdcard.c                                                                                                                                                       |

野火 STM32 开发板 MicroSD 卡硬件原理图：



## 19.2. 1.2 SDIO 简介

野火 STM32 开发板的 CPU ( STM32F103VET6 )具有一个 SDIO 接口。SD/SDIO/MMC 主机接口可以支持 MMC 卡系统规范 4.2 版中的 3 个不同的数据总线模式：1 位(默认)、4 位和 8 位。在 8 位模式下，该接口可以使数据传输速率达到 48MHz，该接口兼容 SD 存储卡规范 2.0 版。SDIO 存储卡规范 2.0 版支持两种数据总线模式：1 位(默认)和 4 位。

目前的芯片版本只能一次支持一个 SD/SDIO/MMC 4.2 版的卡，但可以同时支持多个 MMC 4.1 版或之前版本的卡。除了 SD/SDIO/MMC，这个接口完全与 CE-ATA 数字协议版本 1.1 兼容。

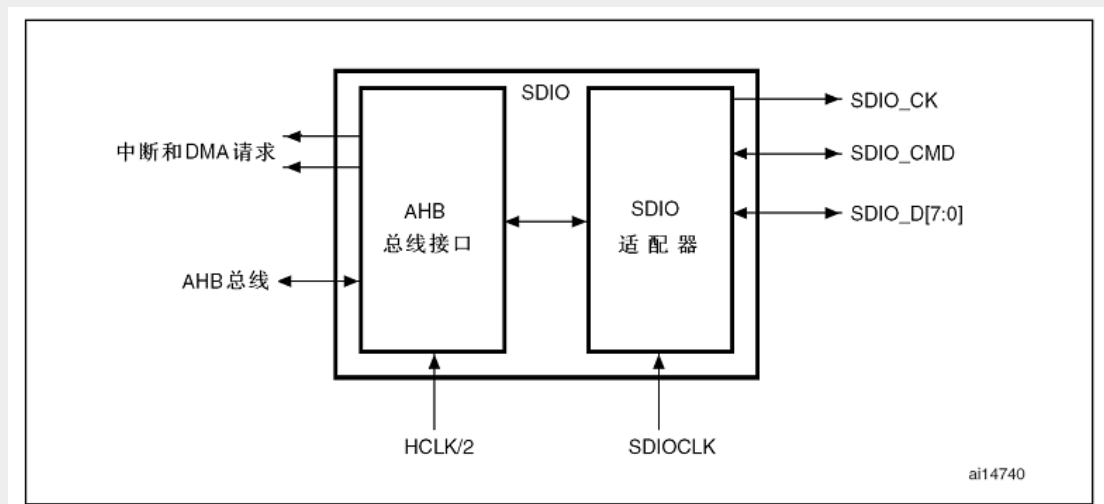
## 19.3. 1.3 SD 协议

大多数人原来没有了解过 SD 协议，又看到 SDIO 的驱动有 2000 多行，感觉无从下手。所以野火重新写了这个文档进行详细的解释，帮助大家更快地跨过这道槛。

附资料：《Simplified\_Physical\_Layer\_Spec.pdf》，这个资料包含了 SDIO 协议中 SD 存储卡的部分。

下面野火结合 STM32 的 SDIO，分析 SD 协议，让大家对它先有个大概了解，更具体的说明在代码中展开。

SDIO 接口图



### 一. 从 SDIO 的时钟说起。

SDIO\_CLK 时钟是通过 PC12 引脚连接到 SD 卡的，是 SDIO 接口与 SD 卡用于同步的时钟。

SDIO 适配器挂载到 AHB 总线上，通过 HCLK 二分频输入到适配器得到 SDIO\_CLK 的时钟，这时  $SDIO\_CLK = HCLK / (2 + CLKDIV)$ 。其中 CLKDIV 是 SDIO\_CLK(寄存器)中的 CLKDIV 位。

另外，SDIO\_CLK 也可以由 SDIOCLK 通过设置 bypass 模式直接得到，这时  $SDIO\_CLK = SDIOCLK = HCLK$ 。

通过下面的库函数来配置时钟：

```
SDIO_Init(&SDIO_InitStructure);
```

对 SD 卡的操作一般是大吞吐量的数据传输，所以采用 DMA 来提高效率，SDIO 采用的是 DMA2 中的通道 4。在数据传输的时候 SDIO 可向 DMA 发出请求。

### 二. 讲解 SDIO 的命令、数据传输方式。

SDIO 的所有命令及命令响应，都是通过 SDIO-CMD 引脚来传输的。

命令只能由 host 即 STM32 的 SDIO 控制器发出。SDIO 协议把命令分成了 11 种，包括基本命令，读写命令还有 ACMD 系列命令等。其中，在发送 ACMD 命令前，要先向卡发送编号为 CMD55 的命令。

参照下面的命令格式图，其中的 start bit, transmission bit, crc7, endbit, 都是由 STM32 中的 SDIO 硬件完成，我们在软件上配置的时候只需要设置 command index 和命令参数 argument。Command index 就是命令索引（编号），如 CMD0, CMD1...被编号成 0, 1...。有的命令会包含参数，读命令的地址参数等，这个参数被存放在 argument 段。

SD 卡命令格式

| Bit position | 47        | 46               | [45:40]       | [39:8]   | [7:1] | 0       |
|--------------|-----------|------------------|---------------|----------|-------|---------|
| Width (bits) | 1         | 1                | 6             | 32       | 7     | 1       |
| Value        | '0'       | '1'              | x             | x        | x     | '1'     |
| Description  | start bit | transmission bit | command index | argument | CRC7  | end bit |

Table 4-16: Command Format

可以通过下面的函数来配置、发送命令：

```
⑤ SDIO_SendCommand(&SDIO_CmdInitStructure); //发送命令
```

SD 卡对 host 的各种命令的回复称为响应，除了 CMD0 命令外，SD 卡在接收到命令都会返回一个响应。对于不同的命令，会有不同的响应格式，共 7 种，分为长响应型（136bit）和短响应型（48bit）。以下图，响应 6（R6）为例：

SD 卡命令响应格式（R6）

| Bit position | 47        | 46               | [45:40]                  | [39:8]<br>Argument field              |                                                         | [7:1] | 0       |
|--------------|-----------|------------------|--------------------------|---------------------------------------|---------------------------------------------------------|-------|---------|
| Width (bits) | 1         | 1                | 6                        | 16                                    | 16                                                      | 7     | 1       |
| Value        | '0'       | '0'              | x                        | x                                     | x                                                       | x     | '1'     |
| Description  | start bit | transmission bit | command index ('000011') | New published RCA [31:16] of the card | [15:0] card status bits: 23,22,19,12:0 (see Table 4-35) | CRC7  | end bit |

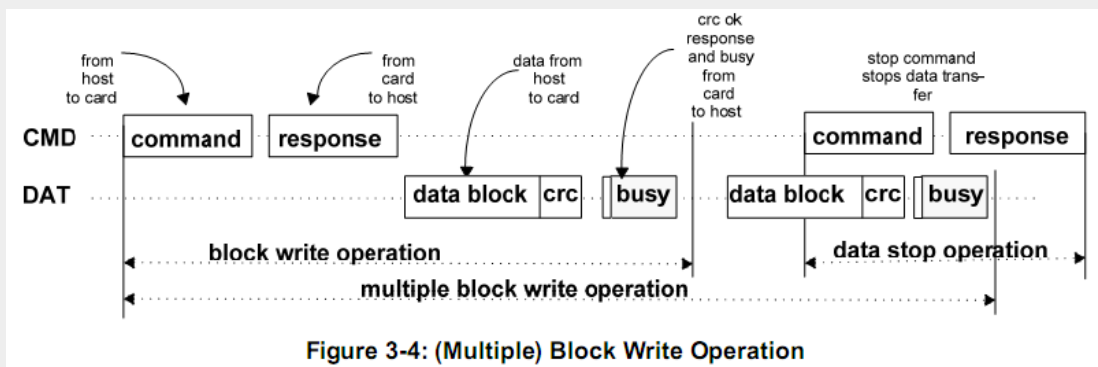
Table 4-32: Response R6

SDIO 通过 CMD 接收到响应后，硬件去除头尾的信息，把 **command index** 保存到 **SDIO\_RESPCMD 寄存器**，把 **argument field** 内容保存存储到 **SDIO\_RESPx 寄存器** 中。这两个值可以分别通过下面的库函数得到。

```
39. SDIO_GetCommandResponse(); //卡返回接收到的命令
40. SDIO_GetResponse(SDIO_RESP1); //卡返回的 argument field 内容
```

数据写入，读取。请看下面的写数据时序图，在软件上，我们要处理的只是读忙。另外，我们的实验中用的是 Micro SD 卡，有 4 条数据线，默认的时候 SDIO 采用 1 条数据线的传输方式，更改为 4 条数据线模式要通过向卡发送命令来更改。

SD 卡的多块写入时序图

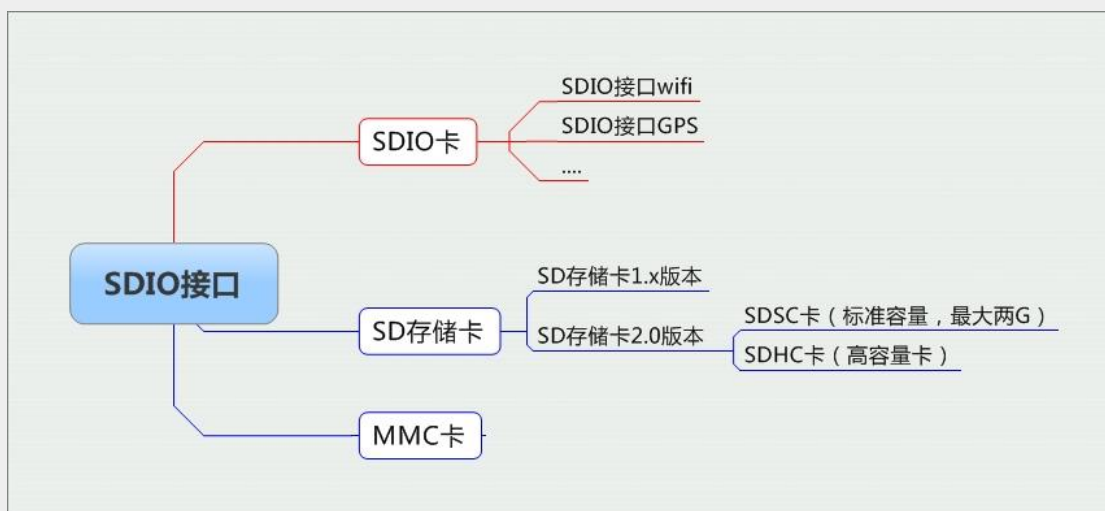


### 三. 卡的种类。

STM32 的 SDIO 支持 SD 存储卡，SD I/O 卡，MMC 卡。

其中 **SDI/O 卡** 与 **SD 存储卡** 是有区别的，SDI/O 卡实际上就是利用 SDIO 接口的一些模块，插入 SD 的插槽中，扩展设备的功能，如：SDI/O wifi, SDI/O cmos 相机等。而 SD 存储卡就是我们平时常见的单纯用于存储数据的卡。

可使用 SDIO 接口类型的卡



本实验中使用的 Micro SD 卡属于 **SDSC**(标准容量, 最大两 G)卡。介绍卡的种类是因为 SD 协议中的命令也支持这三种类型的卡, 因此对 STM32 中的 SDIO 接口进行初始化后, 上电后就要对接入的卡进行**检测、分类**, 这个过程是通过向卡发送一系列不同的命令, 根据卡**不同的响应**来进行分类。

下面进入代码展开具体讲解。

#### 19.4. 1.4 代码分析

首先要添加用的库文件, 在工程文件夹下 **Fwlib** 下我们需添加以下库文件:

```
FWlib/stm32f10x_gpio.c  
  
FWlib/stm32f10x_rcc.c  
  
FWlib/stm32f10x_usart.c  
  
FWlib/stm32f10x_sdio.c  
  
FWlib/stm32f10x_dma.c  
  
FWlib/misc.c
```

还要在 `stm32f10x_conf.h` 中把相应的头文件添加进来:

```
4. #include "stm32f10x_dma.h"
5. #include "stm32f10x_gpio.h"
6. #include "stm32f10x_rcc.h"
7. #include "stm32f10x_sdio.h"
8. #include "stm32f10x_usart.h"
9. #include "misc.h"
```

保持良好的习惯, 从 `main` 函数开始分析:

```
17. int main(void)
18. {
19.
20.     /*进入到 main 函数前, 启动文件 startup (startup_stm32f10x_xx.s) 已经调用
      了在
21.     system_stm32f10x.c 中的 SystemInit(), 配置好了系统时钟, 在外部晶振 8M
      的条件下,
22.     设置 HCLK = 72M */
23.
24.     /* Interrupt Config */
25.     NVIC_Configuration();
26.
27.     /* USART1 config */
28.     USART1_Config();
29.
30.     /*----- SD Init -----
      ----- */
31.     Status = SD_Init();
32.
33.     printf( "\r\n 这是一个MicroSD 卡实验 (没有跑文件系
      统).....\r\n " );
34.
35.
36.     if(Status == SD_OK) //检测初始化是否成功
37.     {
38.         printf( " \r\n SD_Init 初始化成功 \r\n " );
39.     }
40.     else
41.     {
42.         printf("\r\n SD_Init 初始化失败 \r\n" );
43.         printf("\r\n 返回的 Status 的值为:  %d \r\n",Status );
44.     }
45.
46.     printf( " \r\n CardType is : %d ", SDCardInfo.CardType );
47.     printf( " \r\n CardCapacity is : %d ", SDCardInfo.CardCapacity );
48.     printf( " \r\n CardBlockSize is : %d ", SDCardInfo.CardBlockSize )
      ;
49.     printf( " \r\n RCA is : %d ", SDCardInfo.RCA);
50.     printf( " \r\n ManufacturerID is : %d \r\n", SDCardInfo.SD_cid.Man
      ufacturerID );
51.
52.     SD_EraseTest();      //擦除测试
53.
54.     SD_SingleBlockTest(); //单块读写测试
55.
56.     SD_MultiBlockTest(); //多块读写测试
57.
```



```
58. while (1)
59. {}
60. }
```

main 函数的流程简单明了：

1. 用 `NVIC_Configuration()` 初始化好 SDIO 的中断；
2. 用 `USART1_Config()` 配置好用于返回调试信息的串口，`SD_Init()` 开始进行 SDIO 的初始化；
3. 最后分别用 `SD_EraseTest()`、`SD_SingleBlockTest()`、`SD_MultiBlockTest()` 进行擦除，单数据块读写，多数据块读写测试。

下面我们先进入 SDIO 驱动函数的大头——`SD_Init()` 进行分析：

```
39. /*
40. * 函数名: SD_Init
41. * 描述   : 初始化 SD 卡，使卡处于就绪状态 (准备传输数据)
42. * 输入   : 无
43. * 输出   : -SD_Error SD 卡错误代码
44. *         成功时则为 SD_OK
45. * 调用   : 外部调用
46. */
47. SD_Error SD_Init(void)
48. {
49.     /*重置 SD_Error 状态*/
50.     SD_Error errorstatus = SD_OK;
51.
52.     /* SDIO 外设底层引脚初始化 */
53.     GPIO_Configuration();
54.
55.     /*对 SDIO 的所有寄存器进行复位*/
56.     SDIO_DeInit();
57.
58.     /*上电并进行卡识别流程，确认卡的操作电压 */
59.     errorstatus = SD_PowerON();
60.
61.     /*如果上电，识别不成功，返回“响应超时”错误 */
62.     if (errorstatus != SD_OK)
63.     {
64.         /*!< CMD Response TimeOut (wait for CMDSENT flag) */
65.         return(errorstatus);
66.     }
67.
68.     /*卡识别成功，进行卡初始化 */
69.     errorstatus = SD_InitializeCards();
70.
71.     if (errorstatus != SD_OK)    //失败返回
72.     {
73.         /*!< CMD Response TimeOut (wait for CMDSENT flag) */
74.         return(errorstatus);
75.     }
76.
77.     /*!< Configure the SDIO peripheral
```

```

78.  上电识别,卡初始化都完成后,进入数据传输模式,提高读写速度
79.  速度若超过 24M 要进入 bypass 模式
80.  !< on STM32F2xx devices, SDIOCLK is fixed to 48MHz
81.  !< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_TRANSFER_CLK_DIV) */

82.  SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
83.  SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;           //上升沿采集数据
84.  SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;    //时钟频率若超过 24M,要开启此模式
85.  SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
    //若开启此功能,在总线空闲时关闭 sd_clk 时钟
86.  SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;
    //1 位模式
87.  SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_Disable; //硬件流,若开启,在 FIFO 不能进行发送和接收数据时,数据传输暂停
88.  SDIO_Init(&SDIO_InitStructure);
89.
90.  if (errorstatus == SD_OK)
91.  {
92.      /*----- Read CSD/CID MSD registers -----
    */
93.      errorstatus = SD_GetCardInfo(&SDCardInfo); //用来读取 csd/cid 寄存器
94.  }
95.
96.  if (errorstatus == SD_OK)
97.  {
98.      /*----- Select Card -----
    */
99.      errorstatus = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));
    //通过 cmd7 ,rca 选择要操作的卡
100.  }
101.
102.  if (errorstatus == SD_OK)
103.  {
104.      errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b); //开启 4bits 模式
105.  }
106.
107.  return(errorstatus);
108.  }

```

先从整体上了解这个 `SD_Init()` 函数:

1. 用 `GPIO_Configuration()` 进行 SDIO 的端口底层配置
2. 分别调用了 `SD_PowerON()` 和 `SD_InitializeCards()` 函数,这两个函数共同实现了上面提到的卡检测、识别流程。
3. 调用 `SDIO_Init(&SDIO_InitStructure)` 库函数配置 SDIO 的时钟,数据线宽度,硬件流(在读写数据的时候,开启硬件流是很必要的,可以减少出错)
4. 调用 `SD_GetCardInfo(&SDCardInfo)` 获取 sd 卡的 CSD 寄存器中的内容,在 main 函数里输出到串口的数据就是这个时候从卡读取得到的。
5. 调用 `SD_SelectDeselect()` 选定后面即将要操作的卡。
6. 调用 `SD_EnableWideBusOperation(SDIO_BusWide_4b)` 开启 4bit 数据线模式

如果 **SD\_Init()** 函数能够执行完整个流程，并且返回值是 **SD\_OK** 的话则说明初始化成功，就可以开始进行擦除、读写的操作了。

下面进入 **SD\_PowerON()** 函数，分析完这个函数大家就能了解 **SDIO** 如何接收、发送命令了。

```

1.  /*
2.  * 函数名: SD_PowerON
3.  * 描述   : 确保 SD 卡的工作电压和配置控制时钟
4.  * 输入   : 无
5.  * 输出   : -SD_Error SD 卡错误代码
6.  *          成功时则为 SD_OK
7.  * 调用   : 在 SD_Init() 调用
8.  */
9. SD_Error SD_PowerON(void)
10. {
11.     SD_Error errorstatus = SD_OK;
12.     uint32_t response = 0, count = 0, validvoltage = 0;
13.     uint32_t SDType = SD_STD_CAPACITY;
14.
15.     /*!< Power ON Sequence -----
        -----*/
16.     /*!< Configure the SDIO peripheral */
17.     /*!< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_INIT_CLK_DIV) */
18.     /*!< on STM32F2xx devices, SDIOCLK is fixed to 48MHz */
19.     /*!< SDIO_CK for initialization should not exceed 400 KHz */
20.     /*初始化时的时钟不能大于 400KHz*/
21.     SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV; /* HCLK = 72MHz,
        SDIOCLK = 72MHz, SDIO_CK = HCLK/(178 + 2) = 400 KHz */
22.     SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
23.     SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable; //不使用
        bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK
24.     SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable; /
        / 空闲时不关闭时钟电源
25.     SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b; /
        / 1 位数据线
26.     SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
        Disable; //硬件流
27.     SDIO_Init(&SDIO_InitStructure);
28.
29.     /*!< Set Power State to ON */
30.     SDIO_SetPowerState(SDIO_PowerState_ON);
31.
32.     /*!< Enable SDIO Clock */
33.     SDIO_ClockCmd(ENABLE);
34.
35.     /*下面发送一系列命令,开始卡识别流程*/
36.     /*!< CMD0: GO_IDLE_STATE -----
        -----*/
37.     /*!< No CMD response required */
38.     SDIO_CmdInitStructure.SDIO_Argument = 0x0;
39.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE; //cmd0
40.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No; //无响应
41.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
42.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable; //则 CPSM 在开始发送
        命令之前等待数据传输结束。
43.     SDIO_SendCommand(&SDIO_CmdInitStructure); //写命令进命令寄存器
44.

```

```

45. errorstatus = CmdError(); //检测是否正确接收到 cmd0
46.
47. if (errorstatus != SD_OK) //命令发送出错, 返回
48. {
49.     /*!< CMD Response TimeOut (wait for CMDSENT flag) */
50.     return(errorstatus);
51. }
52.
53. /*!< CMD8: SEND_IF_COND -----*/
    -----*/
54. /*!< Send CMD8 to verify SD card interface operating condition */
55. /*!< Argument: - [31:12]: Reserved (shall be set to '0')
56.                - [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
57.                - [7:0]: Check Pattern (recommended 0xAA) */
58. /*!< CMD Response: R7 */
59. SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN; //接收到命令 sd
    会返回这个参数
60. SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND; //cmd8
61. SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r7
62. SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No; //关闭等待中
    断
63. SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
64. SDIO_SendCommand(&SDIO_CmdInitStructure);
65.
66. /*检查是否接收到命令*/
67. errorstatus = CmdResp7Error();
68.
69. if (errorstatus == SD_OK) //有响应则 card 遵循 sd 协议 2.0 版本
70. {
71.     CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0; /*!< SD Card 2.0 , 先把它定
        义会 sdsc 类型的卡*/
72.     SDType = SD_HIGH_CAPACITY; //这个变量用作 acmd41 的参数, 用来询问是 sdsc 卡
        还是 sdhc 卡
73. }
74. else //无响应, 说明是 1.x 的或 mmc 的卡
75. {
76.     /*!< CMD55 */
77.     SDIO_CmdInitStructure.SDIO_Argument = 0x00;
78.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
79.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
80.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
81.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
82.     SDIO_SendCommand(&SDIO_CmdInitStructure);
83.     errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
84. }
85. /*!< CMD55 */ //为什么在 else 里和 else 外面都要发送 CMD55?
86. //发送 cmd55, 用于检测是 sd 卡还是 mmc 卡, 或是不支持的卡
87. SDIO_CmdInitStructure.SDIO_Argument = 0x00;
88. SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
89. SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r1
90. SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
91. SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
92. SDIO_SendCommand(&SDIO_CmdInitStructure);
93. errorstatus = CmdResp1Error(SD_CMD_APP_CMD); //是否响应, 没响应的是 mmc 或
    不支持的卡
94.
95. /*!< If errorstatus is Command TimeOut, it is a MMC card */
96. /*!< If errorstatus is SD_OK it is a SD card: SD card 2.0 (voltage rang
    e mismatch)
    or SD card 1.x */
97.
98. if (errorstatus == SD_OK) //响应了 cmd55, 是 sd 卡, 可能为 1.x, 可能为 2.0
99. {
100.     /*下面开始循环地发送 sdio 支持的电压范围, 循环一定次数*/

```

```

101.
102.     /*!< SD CARD */
103.     /*!< Send ACMD41 SD_APP_OP_COND with Argument 0x80100000 */
104.     while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL))
105.     {
106.         //因为下面要用到 ACMD41, 是 ACMD 命令, 在发送 ACMD 命令前都要先向卡发送
        CMD55
107.         /*!< SEND CMD55 APP_CMD with RCA as 0 */
108.         SDIO_CmdInitStructure.SDIO_Argument = 0x00;
109.         SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;    //CMD55

110.         SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
111.         SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
112.         SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
113.         SDIO_SendCommand(&SDIO_CmdInitStructure);
114.
115.         errorstatus = CmdResp1Error(SD_CMD_APP_CMD); //检测响应
116.
117.         if (errorstatus != SD_OK)
118.         {
119.             return(errorstatus); //没响应 CMD55, 返回
120.         }
121.         //acmd41, 命令参数由支持的电压范围及 HCS 位组成, HCS 位置一来区分卡是 SDSc
        还是 sdhc
122.         SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_SD | SDT
        ype; //参数为主机可供电压范围及 hcs 位
123.         SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
124.         SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r3

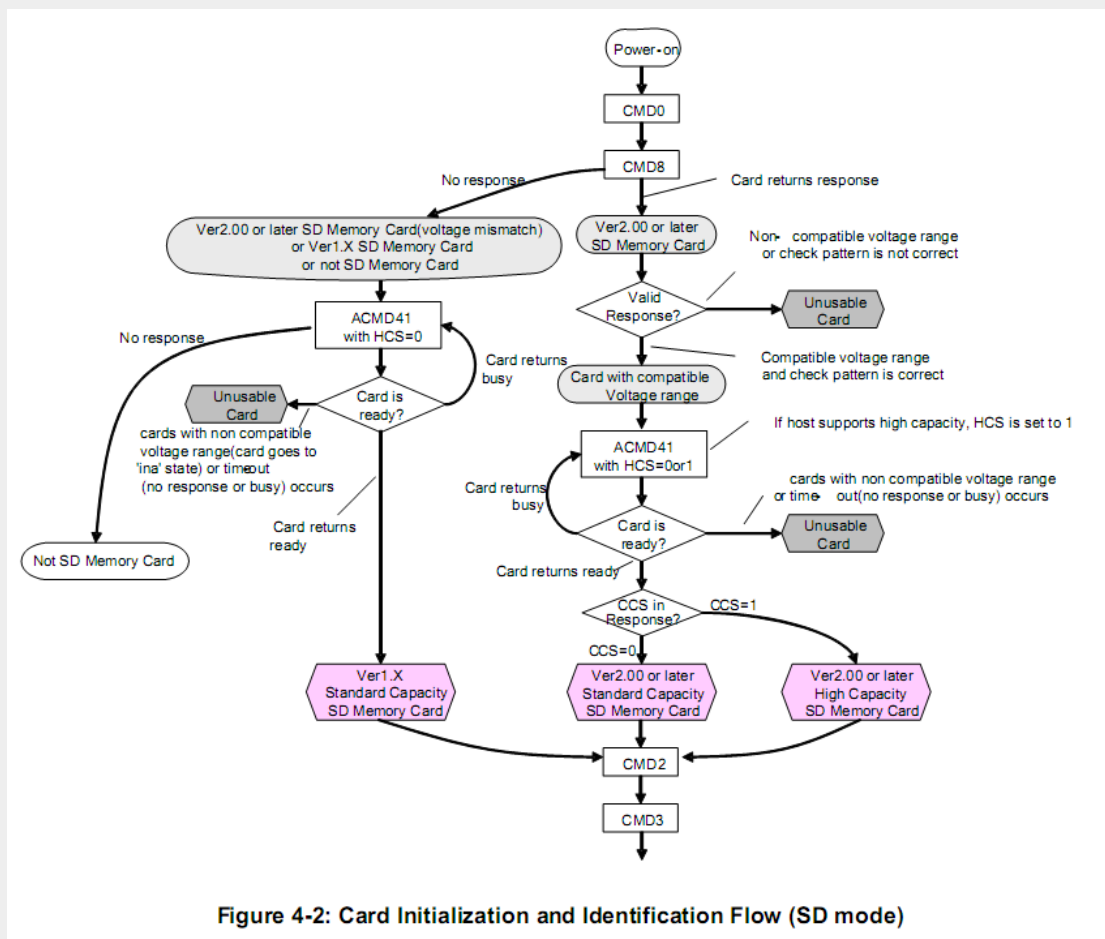
125.         SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
126.         SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
127.         SDIO_SendCommand(&SDIO_CmdInitStructure);
128.
129.         errorstatus = CmdResp3Error(); //检测是否正确接收到数据
130.         if (errorstatus != SD_OK)
131.         {
132.             return(errorstatus); //没正确接收到 acmd41, 出错, 返回
133.         }
134.         /*若卡需求电压在 SDIO 的供电电压范围内, 会自动上电并标志 pwr_up 位*/
135.         response = SDIO_GetResponse(SDIO_RESP1); //读取卡寄存器, 卡状
        态
136.         validvoltage = (((response >> 31) == 1) ? 1 : 0); //读取卡的 ocr
        寄存器的 pwr_up 位, 看是否已工作在正常电压
137.         count++; //计算循环次数
138.     }
139.     if (count >= SD_MAX_VOLT_TRIAL) //循环检测超过一定次数还没上电
140.     {
141.         errorstatus = SD_INVALID_VOLTRANGE; //SDIO 不支持 card 的供电电
        压
142.         return(errorstatus);
143.     }
144.     /*检查卡返回信息中的 HCS 位*/
145.     if (response &= SD_HIGH_CAPACITY) //判断 ocr 中的 ccs 位, 如果是 sdsc
        卡则不执行下面的语句
146.     {
147.         CardType = SDIO_HIGH_CAPACITY_SD_CARD; //把卡类型从初始化的 sdsc 型
        改为 sdhc 型
148.     }
149.
150.     } /*!< else MMC Card */
151.
152.     return(errorstatus);
153. }

```

这个函数的流程就是卡的上电、识别操作，如下图：

卡的上电，识别流程：

截图来自《Simplified\_Physical\_Layer\_Spec.pdf》page27



代码中所有的判断语句都是根据这个图的各个识别走向展开的，最终把卡分为 1.0 版的 SD 存储卡，2.0 版的 SDSC 卡和 2.0 版的 SDHC 卡。

在这个代码流程中有两点要注意一下：

1. 初始化的时钟。SDIO\_CLK 的时钟分为两个阶段，在初始化阶段 SDIO\_CLK 的频率要小于 400KHz，初始化完成后可把 SDIO\_CLK 调整成高速模式，高速模式时超过 24M 要开启 bypass 模式，对于 SD 存储卡即使开启 bypass，最高频率不能超过 25MHz。

2. CMD8 命令。

CMD8 命令格式。

Table 4-15 shows the format of CMD8.

| Bit position | 47        | 46               | [45:40]       | [39:20]       | [19:16]                | [15:8]        | [7:1] | 0       |
|--------------|-----------|------------------|---------------|---------------|------------------------|---------------|-------|---------|
| Width (bits) | 1         | 1                | 6             | 20            | 4                      | 8             | 7     | 1       |
| Value        | '0'       | '1'              | '001000'      | '00000h'      | x                      | x             | x     | '1'     |
| Description  | start bit | transmission bit | command index | reserved bits | voltage supplied (VHS) | check pattern | CRC7  | end bit |

CMD8 命令中的 **VHS** 是用来确认主机 SDIO 是否支持卡的工作电压的。**Check pattern** 部分可以是任何数值，若 SDIO 支持卡的工作电压，卡会把接收到的 check pattern 数值原样返回给主机。

CMD8 命令的响应格式 R7:

| Bit position | 47        | 46               | [45:40]       | [39:20]       | [19:16]          | [15:8]                     | [7:1] | 0       |
|--------------|-----------|------------------|---------------|---------------|------------------|----------------------------|-------|---------|
| Width (bits) | 1         | 1                | 6             | 20            | 4                | 8                          | 7     | 1       |
| Value        | '0'       | '0'              | '001000'      | '00000h'      | x                | x                          | x     | '1'     |
| Description  | start bit | transmission bit | command index | reserved bits | voltage accepted | echo-back of check pattern | CRC7  | end bit |

Table 4-33: Response R7

在驱动程序中调用了 [CmdResp7Error\(\)](#) 来检验卡接收命令后的响应。

### 3.ACMD41 命令。

这个命令也是用来进一步检查 SDIO 是否支持卡的工作电压的，协议要它在调用它之前必须先调用 CMD8，另外还可以通过它命令参数中的 **HCS** 位来区分卡是 **SDHC** 卡还是 **SDSC** 卡。

确认工作电压时循环地发送 ACMD41，发送后检查在 SD 卡上的 **OCR 寄存器** 中的 **pwr\_up** 位，若 **pwr\_up** 位置为 1，表明 SDIO 支持卡的工作电压，卡开始正常工作。

同时把 ACMD41 中的命令参数 **HCS 位置 1**，卡正常工作的时候检测 OCR 寄存器中的 **CCS** 位，若 **CCS** 位为 1 则说明该卡为 SDHC 卡，为零则为 SDSC 卡。

因为 ACMD41 命令属于 ACMD 命令，在发送 ACMD 命令前都要先发送 [CMD55](#)。

ACMD41 命令格式



| ACMD INDEX | type | argument                                                                                                         | resp | abbreviation    | command description                                                                                                                                                                                                                                                                          |
|------------|------|------------------------------------------------------------------------------------------------------------------|------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ACMD41     | bcr  | [31]reserved bit<br>[30]HCS(OCR[30])<br>[29:24]reserved bits<br>[23:0] V <sub>DD</sub> Voltage Window(OCR[23:0]) | R3   | SD_SEND_OP_COND | Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30]. |

ACMD41 命令的响应（R3），返回的是 OCR 寄存器的值

#### 4.9.4 R3 (OCR register)

Code length is 48 bits. The contents of the OCR register are sent as a response to ACMD41.

| Bit position | 47        | 46               | [45:40]  | [39:8]       | [7:1]     | 0       |
|--------------|-----------|------------------|----------|--------------|-----------|---------|
| Width (bits) | 1         | 1                | 6        | 32           | 7         | 1       |
| Value        | '0'       | '0'              | '111111' | x            | '1111111' | '1'     |
| Description  | start bit | transmission bit | reserved | OCR register | reserved  | end bit |

Table 4-31: Response R3

OCR 寄存器的内容

| OCR bit position | OCR Fields Definition                        |
|------------------|----------------------------------------------|
| 0-3              | reserved                                     |
| 4                | reserved                                     |
| 5                | reserved                                     |
| 6                | reserved                                     |
| 7                | Reserved for Low Voltage Range               |
| 8                | reserved                                     |
| 9                | reserved                                     |
| 10               | reserved                                     |
| 11               | reserved                                     |
| 12               | reserved                                     |
| 13               | reserved                                     |
| 14               | reserved                                     |
| 15               | 2.7-2.8                                      |
| 16               | 2.8-2.9                                      |
| 17               | 2.9-3.0                                      |
| 18               | 3.0-3.1                                      |
| 19               | 3.1-3.2                                      |
| 20               | 3.2-3.3                                      |
| 21               | 3.3-3.4                                      |
| 22               | 3.4-3.5                                      |
| 23               | 3.5-3.6                                      |
| 24-29            | reserved                                     |
| 30               | Card Capacity Status (CCS) <sup>1</sup>      |
| 31               | Card power up status bit (busy) <sup>2</sup> |

VDD Voltage Window

1) This bit is valid only when the card power up status bit is set.

2) This bit is set to LOW if the card has not finished the power up routine.



SD 卡上电确认成功后，进入 `SD_InitializeCards()` 函数：

```

5、 /*
6、  * 函数名: SD_InitializeCards
7、  * 描述   : 初始化所有的卡或者单个卡进入就绪状态
8、  * 输入   : 无
9、  * 输出   : -SD_Error SD 卡错误代码
10、        *          成功时则为 SD_OK
11、        * 调用   : 在 SD_Init() 调用，在调用 power_on () 上电卡识别完毕后，调用
        此函数进行卡初始化
12、        */
13、        SD_Error SD_InitializeCards(void)
14、        {
15、            SD_Error errorstatus = SD_OK;
16、            uint16_t rca = 0x01;
17、
18、            if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
19、            {
20、                errorstatus = SD_REQUEST_NOT_APPLICABLE;
21、                return(errorstatus);
22、            }
23、
24、            if (SDIO_SECURE_DIGITAL_IO_CARD != CardType) //判断卡的类型
25、            {
26、                /*!< Send CMD2 ALL_SEND_CID */
27、                SDIO_CmdInitStructure.SDIO_Argument = 0x0;
28、                SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
29、
30、                SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
31、
32、                SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
33、                SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
34、                SDIO_SendCommand(&SDIO_CmdInitStructure);
35、
36、                errorstatus = CmdResp2Error();
37、
38、                if (SD_OK != errorstatus)
39、                {
40、                    return(errorstatus);
41、                }
42、
43、                CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
44、                CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
45、                CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
46、                CID_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
47、            }
48、
49、            /*下面开始 SD 卡初始化流程*/
50、            if ((SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType) || (SDIO_STD_CAPACITY_SD_CARD_V2_0 == CardType) || (SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType)
51、                || (SDIO_HIGH_CAPACITY_SD_CARD == CardType)) //使用的是
52、                2.0 的卡
53、            {
54、                /*!< Send CMD3 SET_REL_ADDR with argument 0 */
55、                /*!< SD Card publishes its RCA. */
56、                SDIO_CmdInitStructure.SDIO_Argument = 0x00;

```

```
54、    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR;
55、    //cmd3
56、    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
57、    //r6
58、    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
59、    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
60、    SDIO_SendCommand(&SDIO_CmdInitStructure);
61、    errorstatus = CmdResp6Error(SD_CMD_SET_REL_ADDR, &rca); //
    把接收到的卡相对地址存起来。
62、
63、    if (SD_OK != errorstatus)
64、    {
65、        return(errorstatus);
66、    }
67、
68、    if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
69、    {
70、        RCA = rca;
71、
72、        /*!< Send CMD9 SEND_CSD with argument as card's RCA */
73、        SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16)
74、    ;
75、        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
76、        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
77、
78、        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
79、        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
80、        SDIO_SendCommand(&SDIO_CmdInitStructure);
81、
82、        errorstatus = CmdResp2Error();
83、
84、        if (SD_OK != errorstatus)
85、        {
86、            return(errorstatus);
87、        }
88、
89、        CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
90、        CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
91、        CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
92、        CSD_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
93、    }
94、
95、    errorstatus = SD_OK; /*!< All cards get intialized */
96、
97、    return(errorstatus);
98、}
```

这个函数向卡发送了 CMD2 和 CMD3 命令

### 1. CMD2

CMD2 命令是要求卡返回它的 CID 寄存器的内容。

命令的响应格式 (R2)。

|              |           |                  |           |                                            |         |
|--------------|-----------|------------------|-----------|--------------------------------------------|---------|
| Bit position | 135       | 134              | [133:128] | [127:1]                                    | 0       |
| Width (bits) | 1         | 1                | 6         | 127                                        | 1       |
| Value        | '0'       | '0'              | '111111'  | x                                          | '1'     |
| Description  | start bit | transmission bit | reserved  | CID or CSD register incl.<br>internal CRC7 | end bit |

Table 4-30: Response R2

因为命令格式是 136 位的，属于长响应。软件接收的信息有 128 位。在长响应的时候通过 `SDIO_GetResponse(SDIO_RESP4)` 中的不同参数来获取 CID 中的不同数据段的数据。

表151 响应类型和SDIO\_RESPx寄存器

| 寄存器        | 短响应       | 长响应         |
|------------|-----------|-------------|
| SDIO_RESP1 | 卡状态[31:0] | 卡状态[127:96] |
| SDIO_RESP2 | 不用        | 卡状态[95:64]  |
| SDIO_RESP3 | 不用        | 卡状态[63:32]  |
| SDIO_RESP4 | 不用        | 卡状态[31:1]   |

总是先收到卡状态的最高位，SDIO\_RESP3寄存器的最低位始终为0。

## 2. CMD3

CMD3 命令是要求卡向主机发送卡的相对地址。在接有多个卡的时候，主机要求接口上的卡重新发一个相对地址，这个地址跟卡的实际 ID 不一样。比如接口上接了 5 个卡，这 5 个卡的相对地址就分别为 1, 2, 3, 4, 5。以后主机 SDIO 对这几个卡寻址就直接使用相对地址。这个地址的作用就是为了寻址更加简单。

接下来我们回到 `SD_Init()` 函数。分析到这里大家应该对 SDIO 的命令发送和响应比较清楚了。在 `SD_InitializeCards()` 之后的 `SD_GetCardInfo(&SDCardInfo)`、`SD_SelectDeselect()` 和 `SD_EnableWideBusOperation(SDIO_BusWide_4b)` 的具体实现就不再详细分析了，实际就是发送相应的命令，对卡进行相应的操作。

接下来分析 main 函数中的 [SD\\_MultiBlockTest\(\)](#) 多块数据读写函数，让大家了解 SDIO 是怎样传输数据的。

```
(4) /*
(5) * 函数名: SD_MultiBlockTest
(6) * 描述   : 多数据块读写测试
(7) * 输入   : 无
(8) * 输出   : 无
(9) */
(10) void SD_MultiBlockTest(void)
(11) {
(12)     /*----- Multiple Block Read/Write -----
-----*/
(13)     /* Fill the buffer to send */
(14)     Fill_Buffer(Buffer_MultiBlock_Tx, MULTI_BUFFER_SIZE, 0x0);
(15)
(16)     if (Status == SD_OK)
(17)     {
(18)         /* Write multiple block of many bytes on address 0 */
(19)         Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, BLOCK_SIZE, NUMBER_OF_BLOCKS);
(20)         /* Check if the Transfer is finished */
(21)         Status = SD_WaitWriteOperation();
(22)         while (SD_GetStatus() != SD_TRANSFER_OK);
(23)     }
(24)
(25)     if (Status == SD_OK)
(26)     {
(27)         /* Read block of many bytes from address 0 */
(28)         Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, BLOCK_SIZE, NUMBER_OF_BLOCKS);
(29)         /* Check if the Transfer is finished */
(30)         Status = SD_WaitReadOperation();
(31)         while (SD_GetStatus() != SD_TRANSFER_OK);
(32)     }
(33)
(34)     /* Check the correctness of written data */
(35)     if (Status == SD_OK)
(36)     {
(37)         TransferStatus2 = Buffercmp(Buffer_MultiBlock_Tx, Buffer_MultiBlock_Rx, MULTI_BUFFER_SIZE);
(38)     }
(39)
(40)     if (TransferStatus2 == PASSED)
(41)         printf("\r\n 多块读写测试成功! ");
(42)
(43)     else
(44)         printf("\r\n 多块读写测试失败! ");
(45)
(46) }
```

把这个函数拿出来分析最重要的一点就是让大家注意在调用了

[SD\\_WriteMultiBlocks\(\)](#) 这一类读写操作的函数后，一定要调用

[SD\\_WaitWriteOperation\(\)](#) [在读数据时调用 [SD\\_WaitReadOperation](#)] 和 [SD\\_GetStatus\(\)](#)

来确保数据传输已经结束再进行其它操作。其中的 [SD\\_WaitWriteOperation\(\)](#) 是用来

等待 DMA 把缓冲的数据传输到 SDIO 的 FIFO 的；而 `SD_GetStatus()` 是用来等待卡与 SDIO 之间传输数据完毕的。

最后进入 `SD_WriteMultiBlocks ()` 函数分析：

```

1.  /*
2.  * 函数名: SD_WriteMultiBlocks
3.  * 描述   : 从输入的起始地址开始, 向卡写入多个数据块,
4.             只能在 DMA 模式下使用这个函数
5.             注意: 调用这个函数后一定要调用
6.                   SD_WaitWriteOperation () 来等待 DMA 传输结束
7.                   和   SD_GetStatus() 检测卡与 SDIO 的 FIFO 间是否已经完成传输
8.  * 输入   :
9.             * @param WriteAddr: Address from where data are to be read.
10.            * @param writebuff: pointer to the buffer that contain the
11.              data to be transferred.
12.            * @param BlockSize: the SD card Data block size. The Block
13.              size should be 512.
14.            * @param NumberOfBlocks: number of blocks to be written.
15.  * 输出   : SD 错误类型
16.  */
17. SD_Error SD_WriteMultiBlocks(uint8_t *writebuff, uint32_t WriteAddr, u
18.   int16_t BlockSize, uint32_t NumberOfBlocks)
19. {
20.     SD_Error errorstatus = SD_OK;
21.     __IO uint32_t count = 0;
22.     TransferError = SD_OK;
23.     TransferEnd = 0;
24.     StopCondition = 1;
25.     SDIO->DCTRL = 0x0;
26.     if (CardType == SDIO_HIGH_CAPACITY_SD_CARD)
27.     {
28.         BlockSize = 512;
29.         WriteAddr /= 512;
30.     }
31.
32.     /*****add, 没有这一段容易卡死在 DMA 检测中
33.     *****/
34.     /*!< Set Block Size for Card, cmd16,若是 sdsc 卡, 可以用来设置块大小, 若
35.       是 sdhc 卡, 块大小为 512 字节, 不受 cmd16 影响 */
36.     SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
37.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
38.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //r1
39.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
40.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
41.     SDIO_SendCommand(&SDIO_CmdInitStructure);
42.
43.     errorstatus = CmdResplError(SD_CMD_SET_BLOCKLEN);
44.
45.     if (SD_OK != errorstatus)
46.     {
47.         return(errorstatus);
48.     }
49.     /*****
50.     *****/
51.     /*!< To improve performance */

```

```
50. SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) (RCA << 16);
51. SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD; // cmd55
52. SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
53. SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
54. SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
55. SDIO_SendCommand(&SDIO_CmdInitStructure);
56.
57.
58. errorstatus = CmdResplError(SD_CMD_APP_CMD);
59.
60. if (errorstatus != SD_OK)
61. {
62.     return(errorstatus);
63. }
64. /*!< To improve performance */// pre-erased, 在多块写入时可发送此命令进
    行预擦除
65. SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)NumberOfBlocks; //
    参数为将要写入的块数目
66. SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCK_COUNT; //cmd
    23
67. SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
68. SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
69. SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
70. SDIO_SendCommand(&SDIO_CmdInitStructure);
71.
72. errorstatus = CmdResplError(SD_CMD_SET_BLOCK_COUNT);
73.
74. if (errorstatus != SD_OK)
75. {
76.     return(errorstatus);
77. }
78.
79.
80. /*!< Send CMD25 WRITE_MULT_BLOCK with argument data address */
81. SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)WriteAddr;
82. SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_WRITE_MULT_BLOCK;
83. SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
84. SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
85. SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
86. SDIO_SendCommand(&SDIO_CmdInitStructure);
87.
88. errorstatus = CmdResplError(SD_CMD_WRITE_MULT_BLOCK);
89.
90. if (SD_OK != errorstatus)
91. {
92.     return(errorstatus);
93. }
94.
95. SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
96. SDIO_DataInitStructure.SDIO_DataLength = NumberOfBlocks * BlockSize;
97.
98. SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;
99. SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToCard;
100. SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;
101. SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;
102. SDIO_DataConfig(&SDIO_DataInitStructure);
103. SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);
104. SDIO_DMAMCmd(ENABLE);
105. SD_DMA_TxConfig((uint32_t *)writebuff, (NumberOfBlocks * Bloc
    kSize));
106.
107. return(errorstatus);
108. }
```

写操作在发送正式的多块写入命令 [CMD25](#) 前调用了 [CMD23](#) 进行预写，这样有利于提高写入的速度。在代码的最后调用了 [SDIO\\_ITConfig\(\)](#)，SDIO 的数据传输结束中断就是这个时候开启的，数据传输结束时，就进入到 [stm32f10x\\_it.c](#) 文件中的中断服务函数 [SDIO\\_IRQHandler\(\)](#) 中处理了，中断服务函数主要就是负责清中断。

最后讲一下官方原版的驱动中的一个 bug。

在官方原版的 SDIO 驱动的 [SD\\_ReadBlock\(\)](#)、[SD\\_ReadMultiBlocks\(\)](#)、[SD\\_WriteBlock\(\)](#) 和 [SD\\_WriteMultiBlocks\(\)](#) 这几个函数中，发送读写命令前，漏掉了发送一个 [CMD16](#) 命令，这个命令用于设置读写 SD 卡的块大小。缺少这个命令很容易导致程序运行时卡死在循环检测 DMA 传输结束的代码中，网上很多人直接移植 ST 官方例程时，用 3.5 版库函数和这个 4.5 版的 SDIO 驱动移植失败，就是缺少了这段用 [CMD16](#) 设置块大小的代码。

到这里，终于讲解完毕啦！这个讲解如果能让你从对 SDIO 一无所知到大概了解的话，我的目标就达到啦，想要更深入了解还是要好好地配合这个例程中我在代码中的注释和附带资料 SD2.0 协议

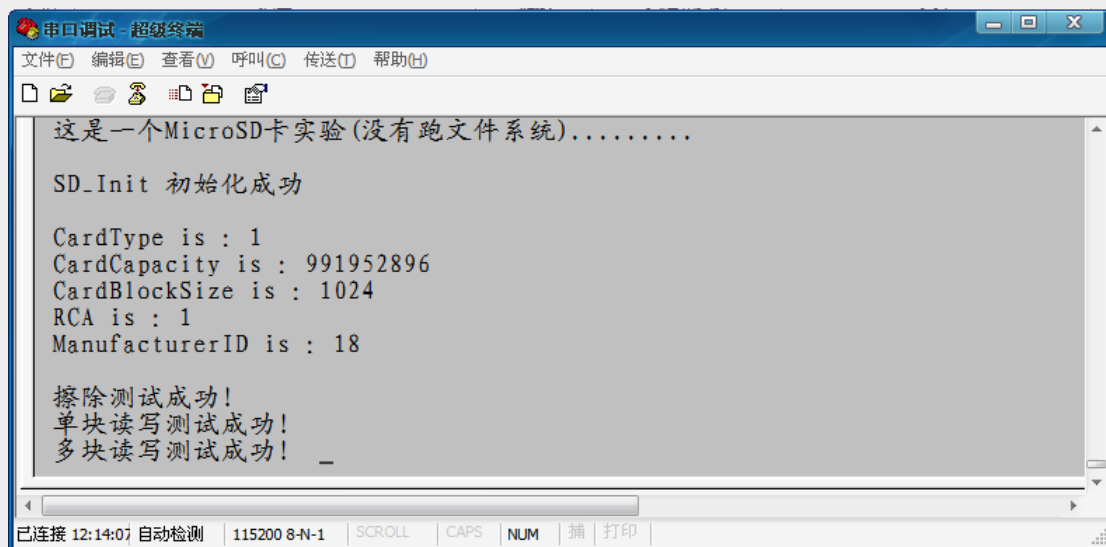
《[Simplified\\_Physical\\_Layer\\_Spec.pdf](#)》好好研究一番！^\_^

**注意：**这个例程是没有跑文件系统的，而是直接就去读卡的 block，这样的话就会破坏卡的分区，在实验完成之后，你再把卡插到电脑上时，电脑会提示你要重新初始化卡，这是正常现象，并不是本实验把你的卡弄坏了，如果卡原来有资料的请先把数据**备份了再进行测试**。但跑文件系统时就不会出现这种问题，有关文件系统的操作将在下一讲的教程中讲解。

### 19.5. 1.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(**两头都是母的交叉线**)，插上 MicroSD 卡(我用的是 1G，经测试，本驱动也适用于 2G 以

上的卡（sdhc 卡）），打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：





## 20. 2、FATFS-文件系统

### 20.1. 什么是文件系统

即使读者可能不了解文件系统，读者也一定对“文件”这个概念十分熟悉。数据在 PC 上是以文件的形式储存在磁盘中的，这些数据的形式一般为 ASCII 码或二进制形式。在上一章我们已经写好了 SD 卡的驱动，也就是说已经可以对 SD 卡进行读写数据。如需要记录本书的书名“STM32 库开发实战”，可以首先把这些文字转化成 ASCII 码，存储在数组中，然后调用块写入函数 `SD_WriteBlock()`，把数组内容写入到 SD 卡的某个地址，在需要的时候从该地址把数据读取出来，再对读出来的数据以 ASCII 码的格式进行解读。

但是，这样直接存储数据会带来极大的不便，如难以记录有效数据的位置，难以确定存储介质的剩余空间，以及应以何种格式来解读数据。就如同一个巨大的图书馆无人管理，杂乱无章地存放着各种书籍，难以查找所需的文档。想象一下图书馆的采购人员购书后，把书籍往馆内一扔，拍拍屁股走人，当有人来借阅某本书的时候，就不得不一本本地查找。这样直接存储数据的方式对于小容量的存储介质如 **EEPROM** 还可以接受，但对于 SD 卡这类大容量设备，我们需要一种高效的方式来管理它的存储内容。

这些管理方式即为文件系统，它是为了存储和管理数据，而在存储介质建立的一种组织结构，这些结构包括操作系统引导区、目录和文件。常见的 windows 下的文件系统格式包括 **FAT32**、**NTFS**、**exFAT**。在使用文件系统前，要先对存储介质进行格式化。格式化之后，在存储介质会创建一个文件分配表和目录。这样，文件系统就可以记录数据存放的物理地址，剩余空间。

使用文件系统时，数据都以文件的形式存储。写入新文件时，先在目录中创建一个文件索引，它指示了文件存放的物理地址，再把数据存储到该地址中。当需要读取数据时，可以从目录中找到该文件的索引，进而在相应的地址中读取数据。具体还涉及到逻辑地址、簇大小、不连续存储等一系列辅助结构或处理过程。

文件系统的存在使我们在存取数据时，不再是简单地向某物理地址直接读写，而是要遵循它的读写格式。如经过逻辑转换，一个完整的文件可能被分成多段存储到不连续的物理地址，使用目录或链表的方式来获知下一段的位置。

上一章的 SD 卡驱动只完成了向物理地址写入数据的工作，而根据文件系统格式的逻辑转换部分则需要额外的代码来完成。实质上，这个逻辑转换部分可以理解为当我们需要写入一段数据时，由它来求解向什么物理地址写入数据、以什么格式写入及写入一些原始数据以外的信息(如目录)。这个逻辑转换部分代码我们也习惯称之为文件系统。

## 20.2. FATFS 文件系统简介

上面提到的逻辑转换部分代码(文件系统)即为本章的要点，文件系统庞大而复杂，它需要根据应用的文件系统格式而编写，而且一般与驱动层分离开来，很方便移植，所以工程应用中一般是移植现成的文件系统源码。

FATFS 是面向小型嵌入式系统的一种通用的 FAT 文件系统。它完全是由 AISI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他的处理器当中，如 8051、PIC、AVR、SH、Z80、H8、ARM 等。FATFS 支持 FAT12、FAT16、FAT32 等格式，所以我们利用前面写好的 SDIO 驱动，把 FATFS 文件系统代码移植到工程之中，就可以利用文件系统的各种函数，对已格式化的 SD 卡的文件进行读写了。

FATFS 文件系统的源码可以从 fatfs 官网下载：

[http://elm-chan.org/fsw/ff/00index\\_e.html](http://elm-chan.org/fsw/ff/00index_e.html)

### 20.2.1. FATFS 的目录结构

在移植 FATFS 文件系统之前，我们先要到 FAT 的官网获取源码，版本为 R0.09。解压之后可看到里面有 doc 和 src 这两个文件夹。doc 文件夹里面是一些使用文档，src 里面是文件系统的源码。

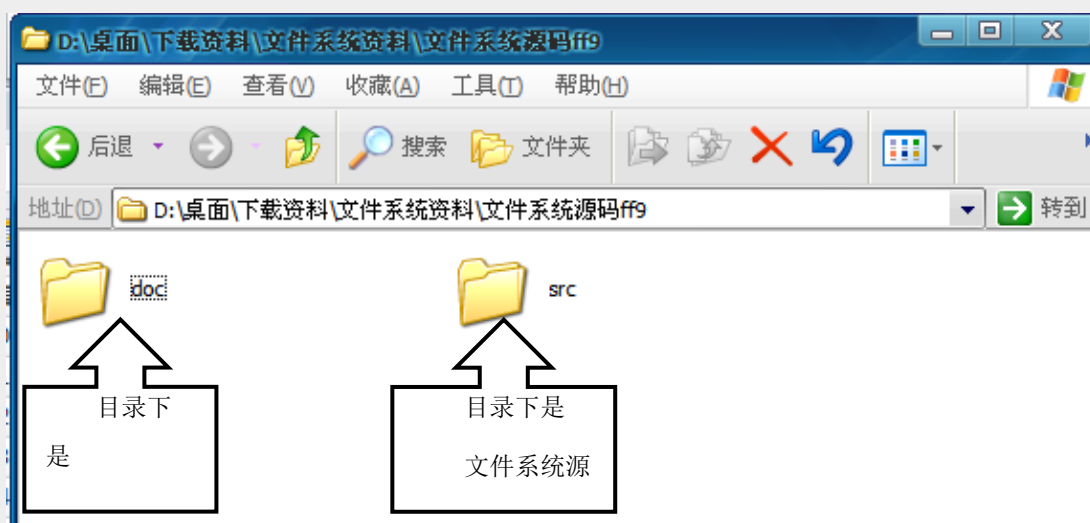


图 20-1 fatfs 源码结构

### 20.2.2. FATFS 帮助文档

打开 doc 文件夹，可看到如下文件目录：

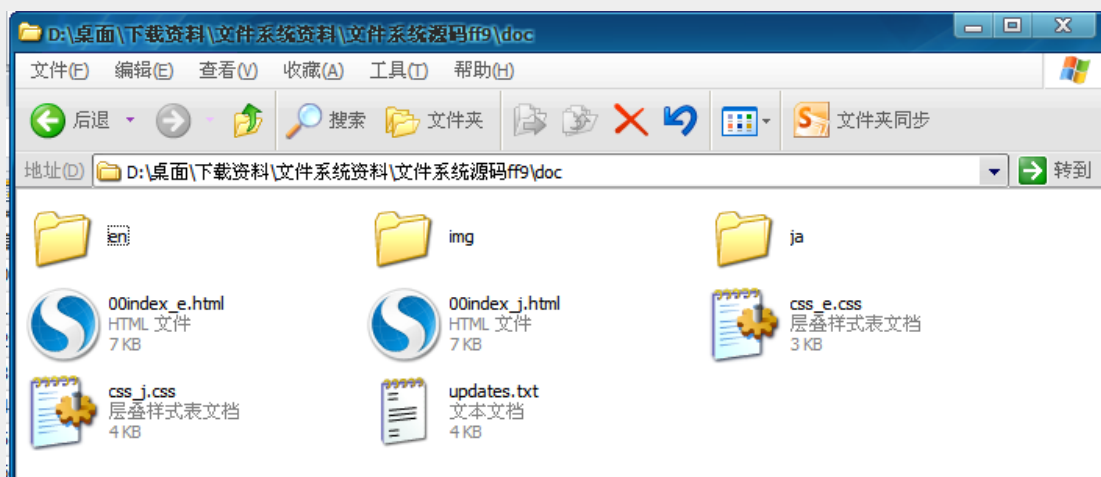


图 20-2 doc 文件夹下的内容

其中 en 和 ja 这两个文件夹里面是编译好的 html 文档，讲的是 FATFS 里面各个函数的使用方法，这些函数就如 LINUX 下的系统调用，是封装得非常好的函数，利用这些函数我们就可以操作我们的 MicroSD 卡了。有关具体的函数我们在用到的时候再讲解。这两个文件夹的唯一区别就是 en 文件夹下的文档是英文的，ja 文件夹下的是日文的。00index\_e.html 是一些关于 FATFS 的英文简介，updates.txt 是 FATFS 的更新信息，至于其他几个文件可以不看。

### 20.2.3. FATFS 源码

打开 src 文件夹，可看到如下目录：

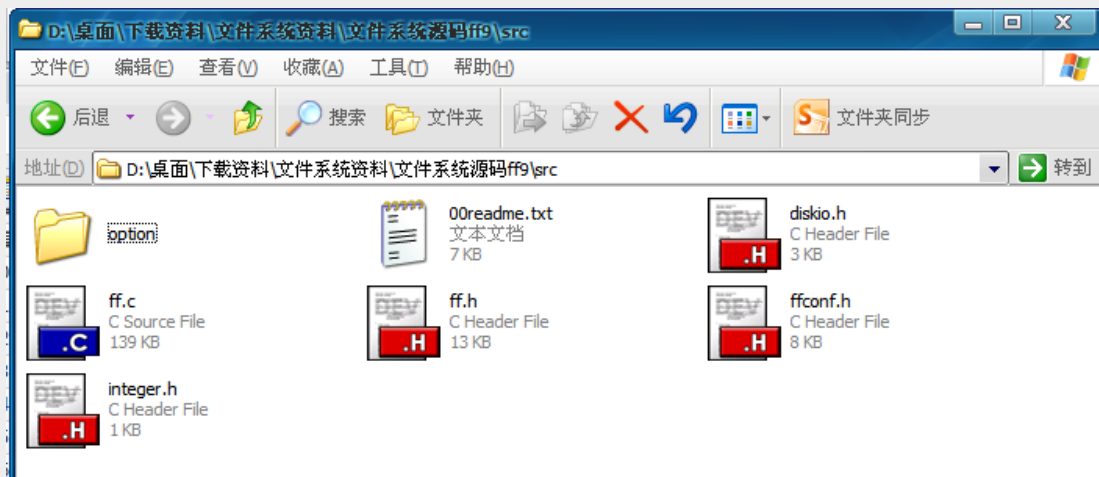


图 20-3 src 文件夹下的源码文件

option 文件夹下是一些可选的外部 c 文件，包含了多语言支持需要用到的文件和转换函数。

在新的 0.09 版源码中没有了 diskio.c 文件，但有 diskio.h。在 diskio.h 源码中有一些关于底层硬件接口的函数声明，我们可以从旧版本的源码中把 diskio.c 的函数定义搬过来，自己新建一个 diskio.c 文件。diskio.c 文件是移植中最关键的文件，它为文件系统提供了最底层的访问 SD 卡的方法，即调用了 SD 驱动函数。

00readme.txt 说明了当前目录下 diskio.c、diskio.h、ff.c、ff.h、integer.h 的功能、涉及了 FATFS 的版权问题(是自由软件)，还讲到了 FATFS 的版本更新信息。

src 文件夹下的源码文件功能简介如下：

- ❑ integer.h：文件中包含了一些数值类型定义。
- ❑ diskio.c：包含底层存储介质的操作函数，这些函数需要用户自己实现，主要添加底层驱动函数。
- ❑ ff.c：独立于底层介质操作文件的函数，利用这些函数实现文件的读写。

❑ **cc936.c** :本文件在 **option** 目录下，是简体中文支持所需要添加的文件，包含了简体中文的 **GBK** 和转换函数。

❑ **ffconf.h**:这个头文件包含了对文件系统的各种配置，有点类似 **stm32f10x\_conf.h** 在 **STM32** 库中的功能。如需要支持简体中文，需要把 **ffconf.h** 中的 **\_CODE\_PAGE** 的宏改成 **936** 并把上面的 **cc936.c** 文件加入到工程之中。

建议阅读这些源码的顺序为：**integer.h -> diskio.c -> ff.c** 。

阅读文件系统源码 **ff.c** 文件需要一定的功底，建议读者先阅读 **FAT32** 的文件格式，再去分析 **ff.c** 文件。若仅为使用文件系统，则只需要理解 **integer.h** 及 **diskio.c** 文件并会调用 **ff.c** 文件中的函数就可以了。本章主要讲解如何把 **FATFS** 文件系统移植到野火开发板上。

### 20.3. 移植 **FATFS** 文件系统实验

#### 20.3.1. 实验描述及工程文件清单

##### 实验描述：

**MicroSD** 卡文件系统 **FATFS R0.09** 测试实验。在 **MicroSD** 卡里面创建一个 **DEMO.TXT** 文本文件，并在文件里面写入字符串“感谢您选用 野火 **STM32** 开发板！”然后通过串口将这些内容打印在电脑的超级终端上。支持简体中文和长文件名，采用 **SDIO** 的 **4bit+DMA** 模式。关于文件系统的使用，将在紧接下来的 **mp3** 例程中进行演示。

##### 硬件连接：

与 **SDIO** 实验相同

**PC8-SDIO-D0** : **DATA0**

**PC9-SDIO-D1** : **DATA1**

**PC10-SDIO-D2** : **DATA2**

**PC11-SDIO-D3** : **CD/DATA3**

PC12-SDIO-CLK : CLK

PD2-SDIO-CMD : CMD

库文件 :

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_sdio.c

FWlib/stm32f10x\_dma.c

FWlib/misc.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/usart.c

USER/sdio\_sdcard.c

文件系统源文件:

使用 fatfs0.09 版本源码

ff9/diskio.c

ff9/ff.c

ff9/cc936.c

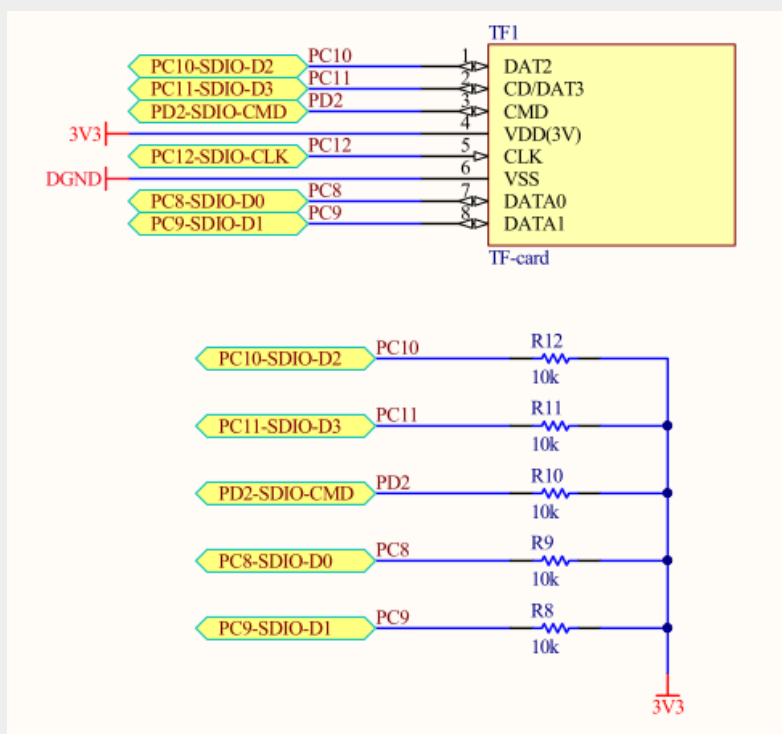


图 20-4 野火 STM32 开发板 MicroSD 卡硬件原理图

本实验是在上一章节的基础上讲解的，只有上一章的实验成功了，文件系统才能跑起来。有关卡的底层的初始化，不再详述，本章着重讲解文件系统的移植。

### 20.3.2. 配置工程环境

首先要获取一个完全没有修改过的文件系统源码，添加到 SD 读写实验中，在该工程的基础上进行文件系统移植。我们在移植这个文件系统的过程中会尽量保持文件系统源码的纯净，尽量做到在修改最少量的源码的情况下移植成功。

(1) 由于本实验与 SD 卡读写实验使用的外设完全一样，首先 SD 读写实验的工程复制一份。

(2) 直接将整个解压后的文件系统源码复制到工程目录下，见图 20-5。

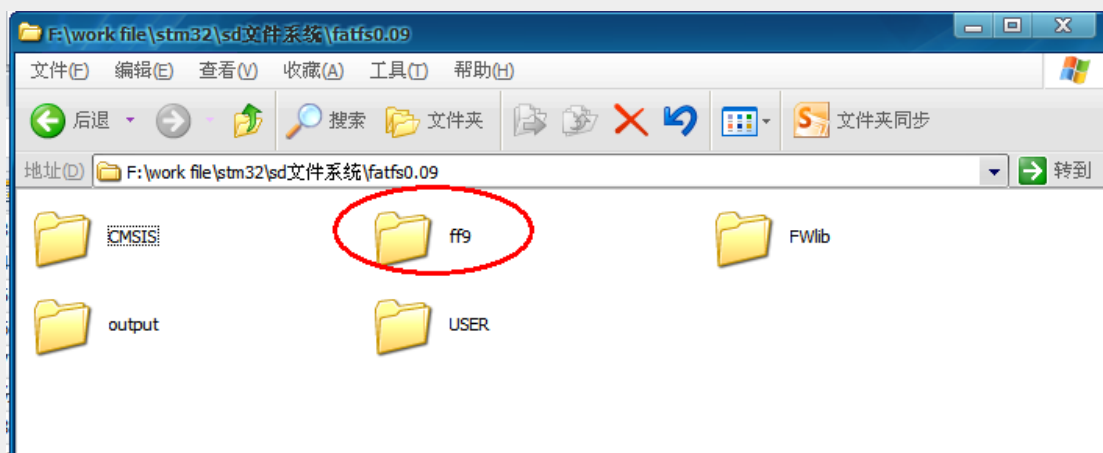


图 20-5 复制文件系统源码到工程目录

(3) 返回到 MDK 界面下，添加 ff9 项目组，并把 ff9\src\目录下的 ff.c, ff9\src\option 目录下的 cc936.c 和新建的 diskio.c 这三个文件添加到 ff9 项目组下，见图 20-6。

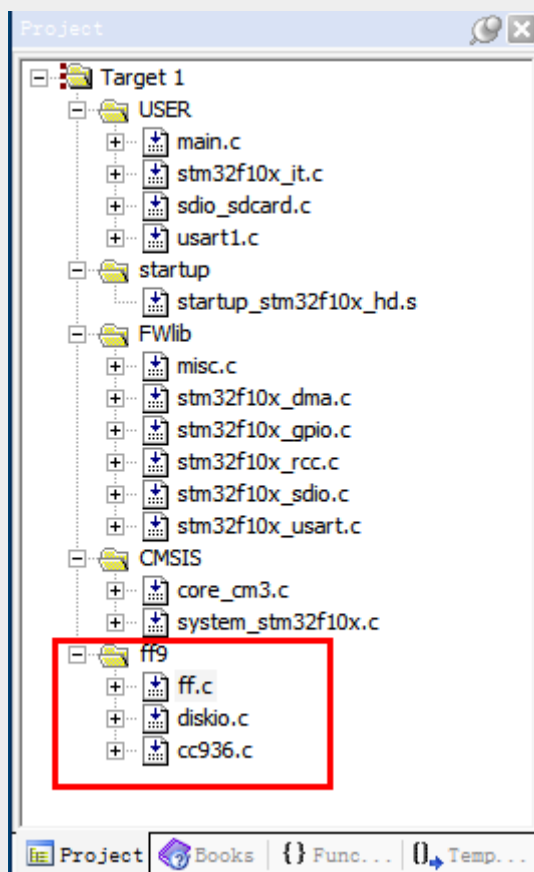


图 20-6 添加源文件到工程

(4) 在 option for project 的 c++编译器选项中添加文件包含路径:



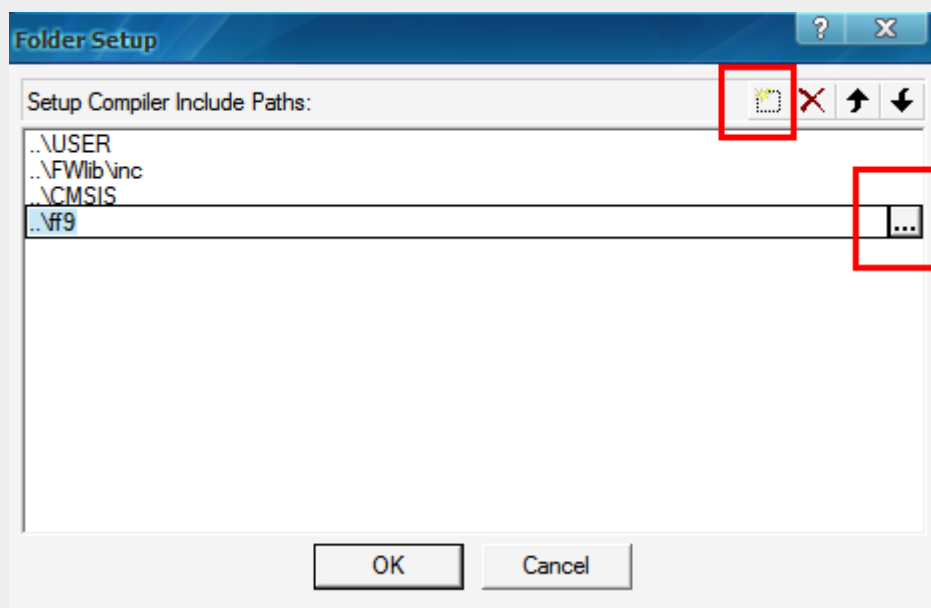
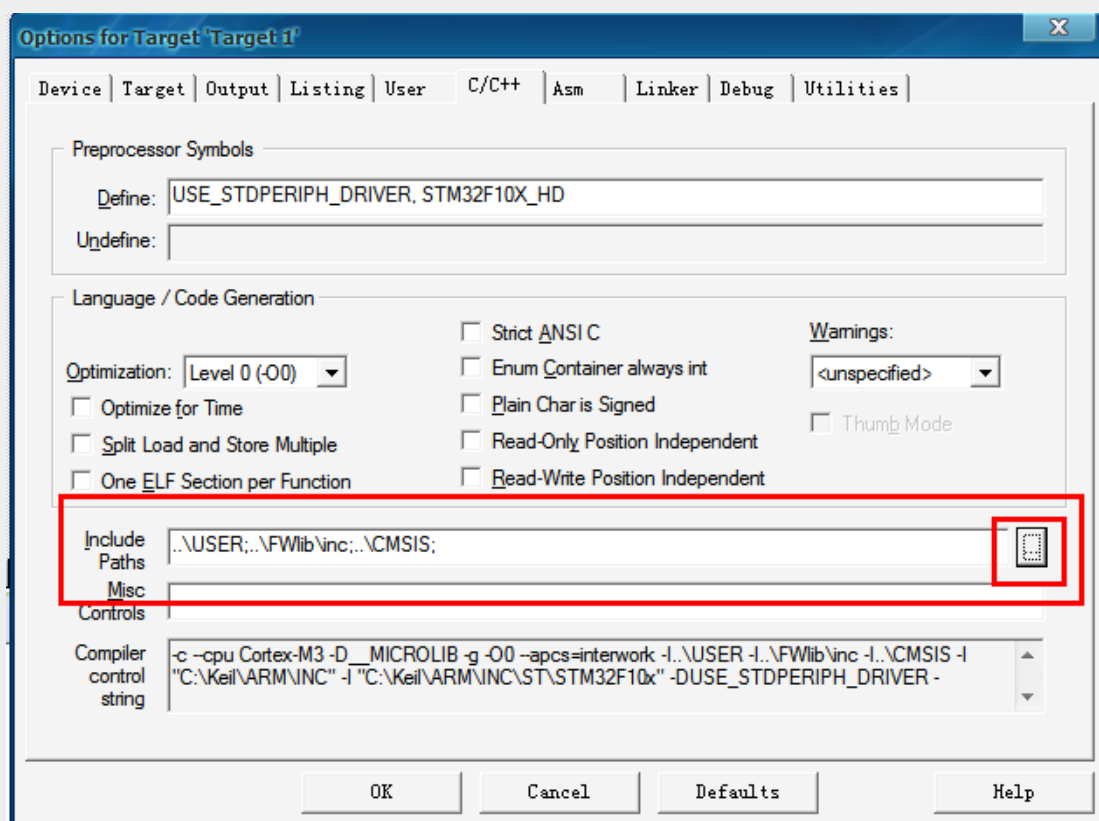


图 20-7 添加头文件搜索路径

接下来就可以直接编译，可以发现，新版本的文件系统源码与 **stm32** 的库文件没有冲突，这样我们就不用像使用旧版本文件系统那样修改重复定义的变

量。省了不少事，如果你使用的还是旧版本的文件系统源码，可以参照之前的野火例程处理重复定义的变量，这里就不详述了。

### 20.3.3. 为文件系统添加底层驱动

FATFS 文件系统与底层介质的驱动分离开来，对底层介质的操作都要交给用户去实现，它仅仅是提供了一个函数接口而已，函数为空，要用户添加代码。因此我们要把 `diskio.c` 中的函数接口与前面写的 SD 实验驱动连接起来。

根据 FATFS 帮助文档的说明，用户需要提供的几个函数的原型如下，在 `diskio.c` 中定义，见代码清单 20-1 至代码清单 20-5。

代码清单 20-1：存储介质初始化函数

```
1.  /* Inidialize a Drive */
2.  DSTATUS disk_initialize (
3.      BYTE drv          /* Physical drive nmuber (0..) */
4.  )
```

代码清单 20-2：存储介质状态函数：

```
1.  /* Return Disk Status */
2.  DSTATUS disk_status (
3.      BYTE drv          /* Physical drive nmuber (0..) */
4.  )
```

代码清单 20-3：扇区读取函数：

```
1.  /* Read Sector(s) */
2.  DRESULT disk_read (
3.      BYTE drv,          /* Physical drive nmuber (0..) */
4.      BYTE *buff,        /* Data buffer to store read data */
5.      DWORD sector,       /* Sector address (LBA) */
6.      BYTE count          /* Number of sectors to read (1..255) */
7.  )
```

代码清单 20-4：扇区写入函数：

```
1.  /* Write Sector(s) */
2.  #if _READONLY == 0
3.  DRESULT disk_write (
4.      BYTE drv,          /* Physical drive nmuber (0..) */
5.      const BYTE *buff,   /* Data to be written */
6.      DWORD sector,       /* Sector address (LBA) */
7.      BYTE count          /* Number of sectors to write (1..255) */
8.  )
```

代码清单 20-5：其它控制功能：

```
● /* Miscellaneous Functions
*/
● DRESULT disk_ioctl (
●     BYTE drv, /* Physical drive number (0..) */
●     BYTE ctrl, /* Control code */
●     void *buff /* Buffer to send/receive control data */
● )
```

这些函数都是操作底层介质的函数，都需要用户自己实现，然后 FATFS 的应用函数就可以调用这些函数来操作我们的卡了。

### ■ 初始化函数接口

以下是文件系统初始化函数接口，文件系统的一些函数会调用这个 `disk_initialize()` 接口来进行底层存储介质的初始化，所以我们在这里加入 SDIO 的 `SD_Init()` 初始化函数，见代码清单 20-6。

代码清单 20-6: `disk_initialize()` 接口

```
13. DSTATUS disk_initialize (
14.     BYTE drv /* Physical drive number (0..) */
15. )
16. {
17.     SD_Error Status;
18.     /* Supports only single drive */
19.     if (drv)
20.     {
21.         return STA_NOINIT;
22.     }
23. /*----- SD Init -----
*/
24.     Status = SD_Init();
25.     if (Status != SD_OK )
26.     {
27.         return STA_NOINIT;
28.     }
29.     else
30.     {
31.         return RES_OK;
32.     }
33. }
34. }
```

这个初始化函数接口，调用了 SDIO 的 `SD_Init()` 函数，返回初始化成功或失败的参数，当文件系统的上层函数调用 `disk_initialize()` 时，实质调用了 `SD_Init()` 函数对 SD 卡进行初始化。

在使用文件系统前，还要在 main 函数里调用 SDIO 的驱动层的一个函数 NVIC\_Configuration(), 用来配置 SDIO 的中断优先级，其实也可以把这个函数也放到 disk\_initialize ()这里边。要注意如果有多个中断要处理时，必须调整 SDIO 的优先级符合工程的应用，不然错了也不知道从哪里找。

## ■ 扇区读取函数

接下来是 disk\_read()函数，这个函数是文件系统读取 SD 卡数据会调用的一个函数，所以我们在这个函数中加入 SDIO 驱动的读取函数的接口。见代码清单 20-7。

代码清单 20-7: disk\_read()接口

```
1. DRESULT disk_read (
2.     BYTE drv,      /* Physical drive nmuber (0..) */
3.     BYTE *buff,    /* Data buffer to store read data */
4.     DWORD sector,  /* Sector address (LBA) */
5.     BYTE count     /* Number of sectors to read (1..255) */
6. )
7. {
8.
9.     if (count > 1)
10.    {
11.        SD_ReadMultiBlocks(buff, sector*BLOCK_SIZE, BLOCK_SIZE,
count);
12.
13.        /* Check if the Transfer is finished */
14.        SD_WaitReadOperation(); //循环查询 dma 传输是否结束
15.
16.        /* Wait until end of DMA transfer */
17.        while(SD_GetStatus() != SD_TRANSFER_OK);
18.
19.    }
20.    else
21.    {
22.
23.        SD_ReadBlock(buff, sector*BLOCK_SIZE, BLOCK_SIZE);
24.
25.        /* Check if the Transfer is finished */
26.        SD_WaitReadOperation(); //循环查询 dma 传输是否结束
27.
28.        /* Wait until end of DMA transfer */
29.        while(SD_GetStatus() != SD_TRANSFER_OK);
30.
31.    }
32.    return RES_OK;
33. }
```

这里分了两个部分，分为单块读取和多块数据读取。因为用一个多块读取函数 SD\_ReadMultiBlocks()比用多个单块读取函 SD\_ReadBlock()数次要简便快

捷得多，所以我们加入了一个判断函数来进行区分。由于文件系统的读写都是以块（512 字节）为单位的，所以你不必担心自己的 SD 驱动没有读写大小低于 512 字节的函数。

要强调的是在调用了读取函数之后要调用 `SD_WaitReadOperation()` 查询 SDIO 是否发送完毕，和用语句 `while(SD_GetStatus() != SD_TRANSFER_OK)` 等待 DMA 传输是结束，这是底层驱动结构决定的。

## ■ 扇区写入函数

扇区写入函数与扇区读取函数十分类似，也是根据写入的扇区数目是 1 个还是多个来分别调用不同的 SD 数据块写入函数。在调用了写入函数之后，也要等待 SDIO 发送完毕及 DMA 传输结束。其具体函数代码见代码清单 20-8。

代码清单 20-8: `disk_write()` 函数

```
1. DRESULT disk_write (
2.     BYTE drv,          /* Physical drive number (0..) */
3.     const BYTE *buff,   /* Data to be written */
4.     DWORD sector,       /* Sector address (LBA) */
5.     BYTE count,         /* Number of sectors to write (1..255)
*/
6. )
7. {
8.
9.     if (count > 1)
10.    {
11.        SD_WriteMultiBlocks((uint8_t *)buff, sector*BLOCK_SIZE,
BLOCK_SIZE, count);
12.
13.        /* Check if the Transfer is finished */
14.        SD_WaitWriteOperation();      //等待 dma 传输结束
15.        while(SD_GetStatus() != SD_TRANSFER_OK);
16.    //等待 sdio 到 sd 卡传输结束
17.    }
18.    else
19.    {
20.        SD_WriteBlock((uint8_t *)buff, sector*BLOCK_SIZE, BLOCK_
SIZE);
21.
22.        /* Check if the Transfer is finished */
23.        SD_WaitWriteOperation();      //等待 dma 传输结束
24.        while(SD_GetStatus() != SD_TRANSFER_OK);
25.    //等待 sdio 到 sd 卡传输结束
26.    }
27.    return RES_OK;
28. }
```

## ■ 时间接口函数

在 `diskio.c` 文件的最后我们还得提供获取时间的函数，因为 `ff.c` 中调用了它，用于记录文件的创建、修改时间，而 `FATFS` 库又没有给出这个函数的原型，所以需要用户实现，否则会编译出错。对于这部分，函数体为空，提供无意义的返回值 `0` 即可，也可以为它加载 `STM32` 的 `RTC` 驱动。若加载 `RTC` 驱动，返回值需要按照如下格式组织数据：

(4) `bit31:25` ——从 1980 至今是多少年，范围是 (0..127)。

(5) `bit24:21` ——月份，范围为 (1..12)

(6) `bit20:16` ——该月份中的第几日，范围为(1..31)

(7) `bit15:11`——时，范围为 (0..23)

(8) `bit10:5` ——分，范围为 (0..59)

(9) `bit4:0` ——秒/ 2，范围为 (0..29)

在本实验中没有添加 `RTC` 时间驱动，没有进行修改，见代码清单 20-9。

代码清单 20-9：空的 `RTC` 时间驱动

```
(7) /*-----  
*/  
(8) /* Get current time  
   */  
(9) /*-----  
*/  
(10)     DWORD get_fattime(void)  
(11)     {  
(12)  
(13)         return 0;  
(14)  
(15)     }
```

#### 20.3.4. 简体中文和长文件支持

添加完底层的驱动，实际上文件系统已经可以正常使用了，为了支持简体中文和长文件名，需要对文件系统进行配置。

优秀的工程一般都会一个源文件配套一个头文件，再外加一个头文件，包含对工程进行裁剪、配置的条件编译宏。如 `STM32` 的库文件中就有

stm32f10x\_conf.h 文件可对使用的库文件进行配置。FATFS0.09 文件系统也是一个很优秀的工程，我们可以在 ffconf.h 头文件中对文件系统进行裁剪。

在这个文件中，修改如下的宏，这是 ffconf.h 中的一部分，见代码清单 20-10。

代码清单 20-10: ffconf.h 文件中的宏

```
1.  /*-----*/
2.  /  Locale and Namespace Configurations
3.  /-----*/
4.
5.  #define _CODE_PAGE  936
6.  #define _USE_LFN     2      /* 0 to 3 */
7.  #define _MAX_LFN     255 /* Maximum LFN length to handle (12 to
255) */
8.  #define _LFN_UNICODE  0      /* 0:ANSI/OEM or 1:Unicode */
9.  #define _FS_RPATH     0      /* 0 to 2 */
```

- (1) 把\_CODE\_PAGE 宏配置成简体中中文的 code\_page，936。
- (2) 把\_USE\_LFN 宏 配置成 2，表示开启长文件名支持。
- (3) 把\_MAX\_LFN 宏配置成 255，表示可支持的长文件名的最大字节数。

这些配置参数可以从源文件中的注释或帮助文档中查找到，由于篇幅问题，野火省略了源文件中对于以上几个宏的注释。由于使用简体中文支持后，添加的字库会使代码变得什么庞大，下载代码的时候会花上较长的时间，建议读者在调试完成基本功能后，再添加简体中文支持。

按这样的配置就可以支持简体中文和长文件名，但在具体配置的时候要注意堆栈大小问题，使用长文件名时也与使用默认长度文件名不同。关于这部分的详细解释在下一章节。在本章中我们先这样把文件系统跑起来，感受一下。

到这里我们算是把文件系统移植成功了，接下来的任务就是调用文件系统的函数来操作我们的卡，往 SD 卡里读写文件。

### 20.3.5. main 文件

实现好底层介质的操作函数之后，我们就可以回到应用层了，下面我们从main文件开始看起，见代码清单 20-11。

代码清单 20-11: FATFS 例程的 main 函数

```

4.  /* Includes -----*/
5.  #include "stm32f10x.h"
6.  #include "sdio_sdcard.h"
7.  #include "usart1.h"
8.  #include "ff.h"
9.
10. int res;      //读写文件的返回值
11. int a;
12.
13. FIL fsrc, fdst; //文件系统结构体，包含文件指针等成员
14. FATFS fs;      //记录文件系统盘符信息的结构体
15. UINT br, bw;   // File R/W count
16. BYTE buffer[512]; // file copy buffer
17. BYTE textFileBuffer[] = "感谢您选用 野火 STM32 开发板 !
   ^_^ \r\n";
18.
19.
20. int main(void)
21. {
22.
23.
24.     /* USART1 config */
25.     USART1_Config();
26.
27.     /* Interrupt Config */
28.     NVIC_Configuration();
29.
30.     printf("\r\n 这是一个MicroSD卡文件系统实验
(FATFS R0.09)\n");
31.
32.     printf ( "\r\n disk_initialize starting.....\n " );
33.
34.     f_mount(0, &fs);
35.
36.     res = f_open(&fdst, "0:/Demo.TXT", FA_CREATE_NEW | FA_WRITE);
37.
38.     if ( res == FR_OK )
39.     {
40.         /* 将缓冲区的数据写到文件中 */
41.         res = f_write(&fdst, textFileBuffer, sizeof(textFileBuf
fer), &bw);
42.         printf( "\r\n 文件创建成功 \n" );
43.         /*关闭文件 */
44.         f_close(&fdst);
45.     }
46.     else if ( res == FR_EXIST )
47.     {
48.         printf( "\r\n 文件已经存在 \n" );
49.     }
50.
51.     /*----- 将刚刚新建的文件里面的内容打印到超级终端 -----*/
52.     /* 以只读的方式打开刚刚创建的文件 */

```



```

53.     res = f_open(&fdst, "0:/Demo.TXT", FA_OPEN_EXISTING | FA_RE
AD);
54. /* 打开文件 */
55.     br = 1;
56.     a = 0;
57.     for (;;)
58.     {
59.         for ( a=0; a<512; a++ )      /* 清缓冲区 */
60.             buffer[a]=0;
61.
62.         res = f_read( &fdst, buffer, sizeof(buffer), &br );
63. /* 将文件里面的内容读到缓冲区 */
64.         printf("\r\n %s ", buffer);
65.         if (res || br == 0) break;      /* 错误或者到了文件尾 */
66.
67.     }
68.     f_close(&fdst);      /* 关闭打开的文件 */
69.
70.
71.     while (1)
72.     {}
73. }
74.
75. /***** (C) COPYRIGHT 2012 野火嵌入式开发工作
室 **END OF FILE***/

```

首先因为 main 函数中要用到 ff.c 文件的函数接口，所以在 main.c 中将 c 文件对应的头文件 ff.h 包含进来。

## ■ 写入文件

定义了一系列文件系统需要用到的变量后，进入 main 函数，初始化了串口、NVIC，开始了写文件流程，这个流程调用了一系列文件系统的函数：

31. 第 31 行，调用函数 f\_mount() 创建一个工作区，它的另一个功能是调用了底层的 disk\_initialize () 函数，进行 sdio 接口的初始化，将我们的底层硬件初始化好，这一步非常重要，如果不成功，接下来什么都干不了。

32. 第 33 行，调用 f\_open() 函数在刚刚开辟的工作区的盘符 0 下打开一个名为 Demo.TXT 的文件，以创建新文件、或写入的方式打开 (参数 “ FA\_CREATE\_NEW | FA\_WRITE ”)，如果文件不存在的话则创建这个文件。本函数还将 Demo.TXT 这个文件关联到 fsrc 这个结构指

针，以后我们操作文件就是通过这个结构指针来完成的，可以把它理解为文件指针。

33. 第 35 行，对 `f_open()` 函数的返回值进行检查，若返回值 `res` 等于宏 `FR_OK`，表示创建新文件成功，若返回值 `res` 等于宏 `FR_EXIST`，表明文件已经存在。

34. 第 38 行，创建文件成功，调用 `f_write()` 将缓冲区的数组变量 `textFileBuffer` 的内容写到刚刚打开的 `Demo.TXT` 文件中。写完之后必须调用 `f_close()` 函数关闭已打开的文件，否则前面写入的数据无效，还可能导致其它错误。

### ■ 读取文件

从第 48 行开始，演示的是读文件流程，从刚刚写入的文件中读取出数据：

3. 第 50 行，调用函数 `f_open()` 以只读的方式打开刚刚的文件。

4. 第 59 行，调用函数 `f_read()` 将文件的内容读到缓冲区 `buffer` 数组变量中，然后调用 `printf()` 将读取得的数据打印到电脑的超级终端。

5. 第 62 行，根据 `f_read()` 函数的返回值和调用它时使用的参数 `br` 来判断是否读取到了文件尾。

6. 第 65 行，调用 `f_close()` 关闭文件。当被打开的文件操作完成之后都要调用 `f_close()` 将它关闭，这如同一块动态分配的内存存在用完之后都要调用 `free()` 来将它释放。

这里涉及到了 **FATFS** 文件系统库函数的操作，如果读者学习过 **LINUX** 系统调用，操作这些函数将是非常简单，如果没有学过，可以把本实验当是学习 **LINUX** 文件操作的基础。在 **FATFS** 源码目录 `doc` 这个文件夹中提供了每个应用函数的用法，如 图 20-8。

### f\_read

The `f_read` function reads data from a file.

```

FRESULT f_read (
    FIL* FileObject,    /* Pointer to the file object structure */
    void* Buffer,        /* Pointer to the buffer to store read data */
    UINT ByteToRead,    /* Number of bytes to read */
    UINT* ByteRead       /* Pointer to the variable to return number of bytes read */
);
        
```

#### Parameters

**FileObject**  
Pointer to the open file object.

**Buffer**  
Pointer to the buffer to store read data

**ByteToRead**  
Number of bytes to read in range of `UINT`.

**ByteRead**  
Pointer to the `UINT` variable to return number of bytes read

#### Return Values

[FR\\_OK](#), [FR\\_DISK\\_ERR](#), [FR\\_INT\\_ERR](#), [FR\\_NOT\\_READY](#), [FR\\_INVALID\\_OBJECT](#), [FR\\_TIMEOUT](#)

#### Description

The file pointer of the file object increases in number of bytes read. After the function succeeded, `*ByteRead` should be checked to detect the end of file. In case of `*ByteRead < ByteToRead`, it means the read/write pointer reached end of the file during read operation.

函数参数说明

- 1.文件指针：用于指示读取哪个文件
- 2.缓冲数组：用于存放读取得的数据
- 3.将要读取字节数：表示要读取多少个字节的

内容

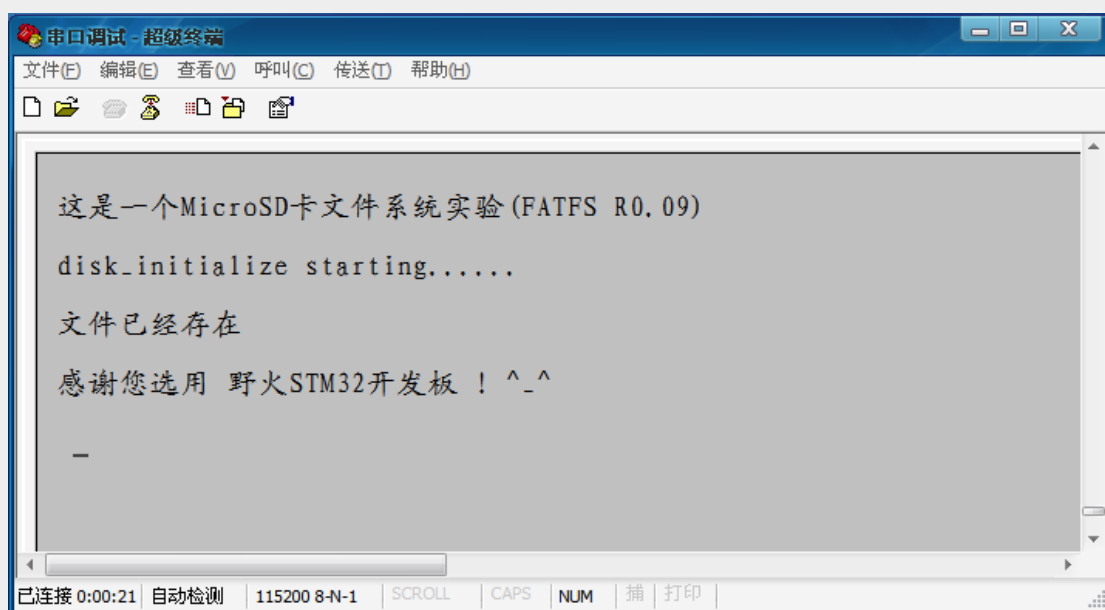
The value is always valid after the function call regardless of the result.

详细功能描述

图 20-8 f\_read 帮助文档说明

### 20.3.6. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，插上已格式化成 FAT32 格式的 MicroSD 卡( 野火用的是 1G，4G 的也已经测试通过，理论上 SDSC、SDHC 卡均可正常使用 )，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息，完成后可以用 PC 来查看 SD 卡内容，看看卡里是不是多了一个文件？



## 21.3、FSMC-液晶显示汉字/图片

在上一章节中，我们已经成功地实现了驱动 LCD 和触摸屏，并制作了触摸画板小应用，但是若要显示文字或图片文件，则还需要利用文件系统，读取保存在 SD 卡中的字库文件、图片文件。

### 21.1. 什么是字模？

我们知道其实液晶屏就是一个由像素点组成的点阵，若要显示文字，则需要很多像素点的共同构成。见图 21-1，图中是两个由 16\*16 的点阵显示的两个汉字。

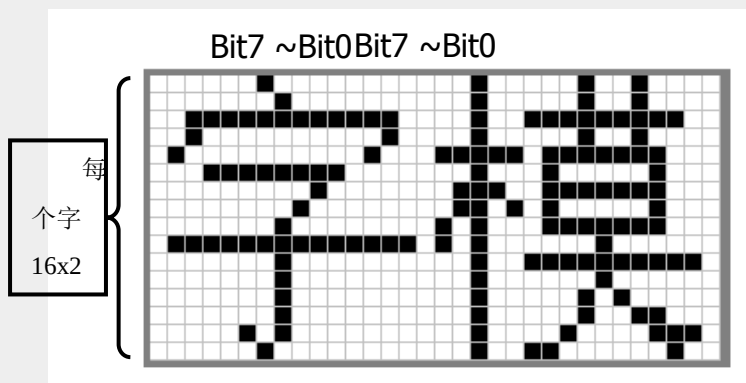


图 21-1 字模

如果我们规定：每个汉字都由这样 16\*16 的点阵来显示，把笔迹经过的像素点以“1”表示，没有笔迹的点以“0”表示，每个像素点的状态以一个二进制位来记录，用  $16*16/8 = 32$  个字节就可以把这个字记录下来。这 32 个字节数据就称为该文字的字模，还有其它常用字模是 24\*24、32\*32 的。16\*16 的“字”的字模数据见代码清单。

代码清单 21-1：“字”的字模数据

```
1. /* 字 */
2. unsigned char code Bmp003[] =
3. {
4. /*-----
5. ; 源文件 / 文字：字
6. ; 宽×高（像素）：16×16
```

7.; 字模格式/大小：单色点阵液晶字模，横向取模，字节正序/32 字节

8.-----\*/

9.

10. 0x02, 0x00, 0x01, 0x00, 0x3F, 0xFC, 0x20, 0x04, 0x40, 0x08, 0x1F, 0xE0, 0x00, 0x40, 0x00, 0x80,

11. 0xFF, 0xFF, 0x7F, 0xFE, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x05, 0x00, 0x02, 0x00,

12. };

在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模。如果使用 LCD 的画点函数，按位来扫描这些字模数据，把为 1 的位以黑色来显示 (也可以使用其它颜色)，即可把整个点阵还原出来，显示在液晶屏上。

## 21.2. 制作字模

我们采用“字模 III-增强版 v3.91” 软件来制作中文字库，步骤如下：

### 22. 打开字模软件

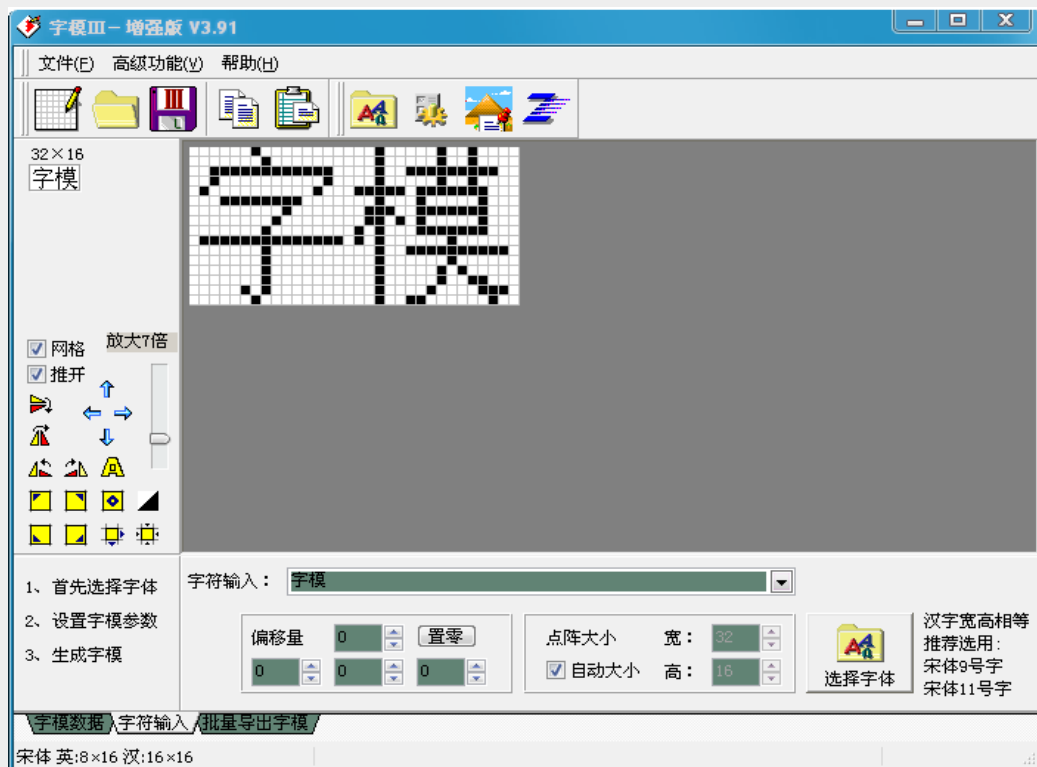


图 21-2 字模软件界面

23. 点击“自动批量生成字库”按钮选项。

软件界面左下角将出现一下几个按钮选项：



图 21-3 字库选项

24. 点击选择“二级汉字库”按钮。

在“输入批量字符”框里面将会列出二级汉字的所有汉字，其中共收录了 6768 个汉字字符，非特殊情况下都能够满足大家的要求，如图 21-4。

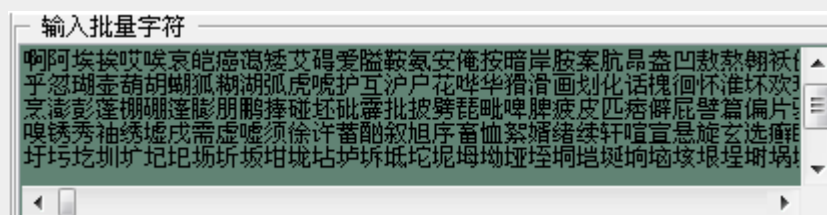
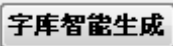


图 21-4 字库

25. 点击“字库智能生成”按钮，弹出“字库批量参数确认”对话框。

我们在“源字体”选项里面做如下设置，需要注意的是大小问题，因为我们本次的设计目标是实现 16\*16 的汉字，所以在此选择‘小四’字体。

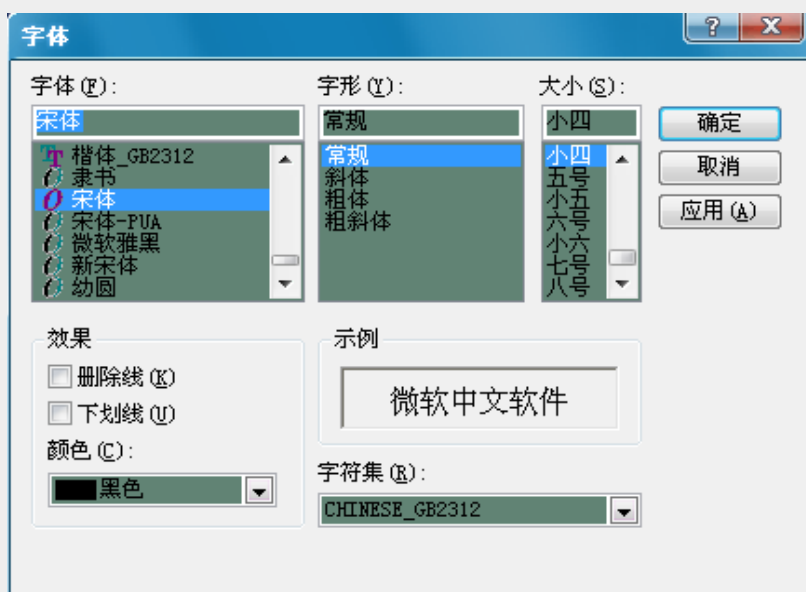


图 21-5 设置字体

设置好之后如下：

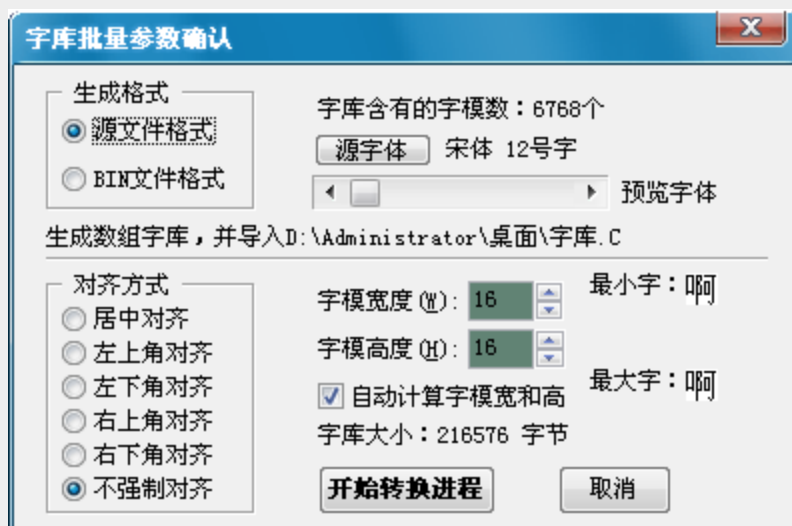


图 21-6 转换字库

26. 点击“开始转换进程”按钮 **开始转换进程**，就会在安装目录下或者你设置好的目录下生成.c 后缀的字库文件。

27. 对于 LCD 显示来说，只要能够在指定的位置描写制定颜色的点，那么就能够很好地根据汉字字模信息来描写汉字。在此，为了能够更好的清楚字模的取向和高低位的排列顺序，我们可以现先在 pc 测试我们刚才制作好的库文件。

在这里我们取“当”字符的数据来测试。



```

1459  /* 当 */
1460  0x00,0x80,0x10,0x90,0x08,0x98,0x0C,0x90,0x08,0xA0,0x00,0x80,0x3F,0xFC,0x00,0x04,
1461  0x00,0x04,0x1F,0xFC,0x00,0x04,0x00,0x04,0x00,0x04,0x3F,0xFC,0x00,0x04,0x00,0x00,

```

图 21-7 “当”字的字模

VC6.0 测试源码如下，该代码实现了把字模中为 1 的点都用数字“8”来表示，见代码清单 21-2。

代码清单 21-2：“当”字字模示例

```

1. #include <stdio.h>
2.
3. unsigned char cc[] =
4. { /* "当"字符 */
5. 0x00,0x80,0x10,0x90,0x08,0x98,0x0C,0x90,0x08,0xA0,0x00,0x80,0x3
F,0xFC,0x00,0x04,
6. 0x00,0x04,0x1F,0xFC,0x00,0x04,0x00,0x04,0x00,0x04,0x3F,0xFC,0x0
0,0x04,0x00,0x00
7. };
8.
9. void main()
10. {
11.     int i,j;
12.     unsigned char kk;
13.     for ( i=0; i<16; i++)
14.     {
15.         for(j=0; j<8; j++)
16.         {
17.             kk = cc[2*i] << j ;    //左移 j 位
18.
19.             if( kk & 0x80) //如果最高位为 1
20.             {
21.                 printf("8");
22.             }
23.             else
24.             {
25.                 printf(" ");
26.             }
27.         }
28.
29.         for(j=0; j<8; j++)
30.         {
31.
32.             kk = cc[2*i+1] << j ;    //左移 j 位
33.
34.             if( kk & 0x80) //如果最高位为 1
35.             {
36.                 printf("8");
37.             }
38.             else
39.             {
40.                 printf(" ");
41.             }
42.         }
43.     }
44.
45.     printf("\n");
46.

```

```
47.     }  
48.     printf("\n\n");  
49.  
50.     }  
51.  
52.
```

测试结果如图 21-8。

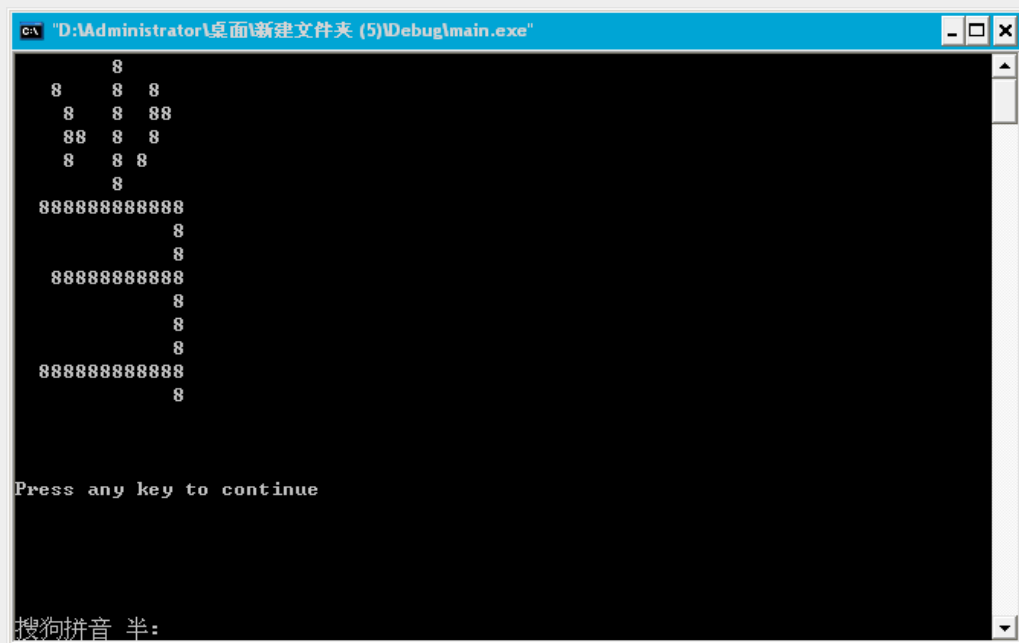


图 21-8 测试结果

看到以上的测试结果，相信大家对汉字的取模方向和高低位的排列顺序有了比较直观的了解。

28. 回到“字模 III-增强版 v3.91”软件，采用与之前同样的方式生成 bin 格式的字库文件(即“生成格式”选项设置为“bin 文件格式”)。

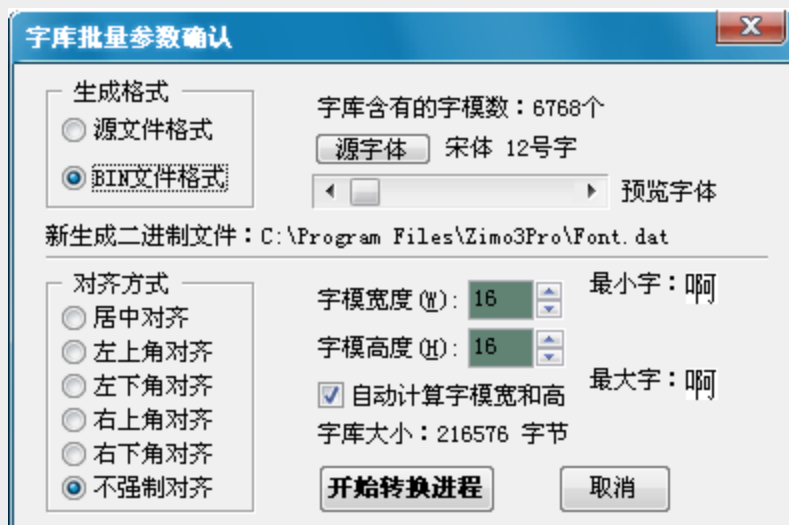


图 21-9 生成 bin 文件格式

在软件安装目录下会生成 Font.dat 文件,我们用“WinHex”软件查看他的具体内容,与刚才制作的.c字库的文件内容是一致的,对比如图 21-10。

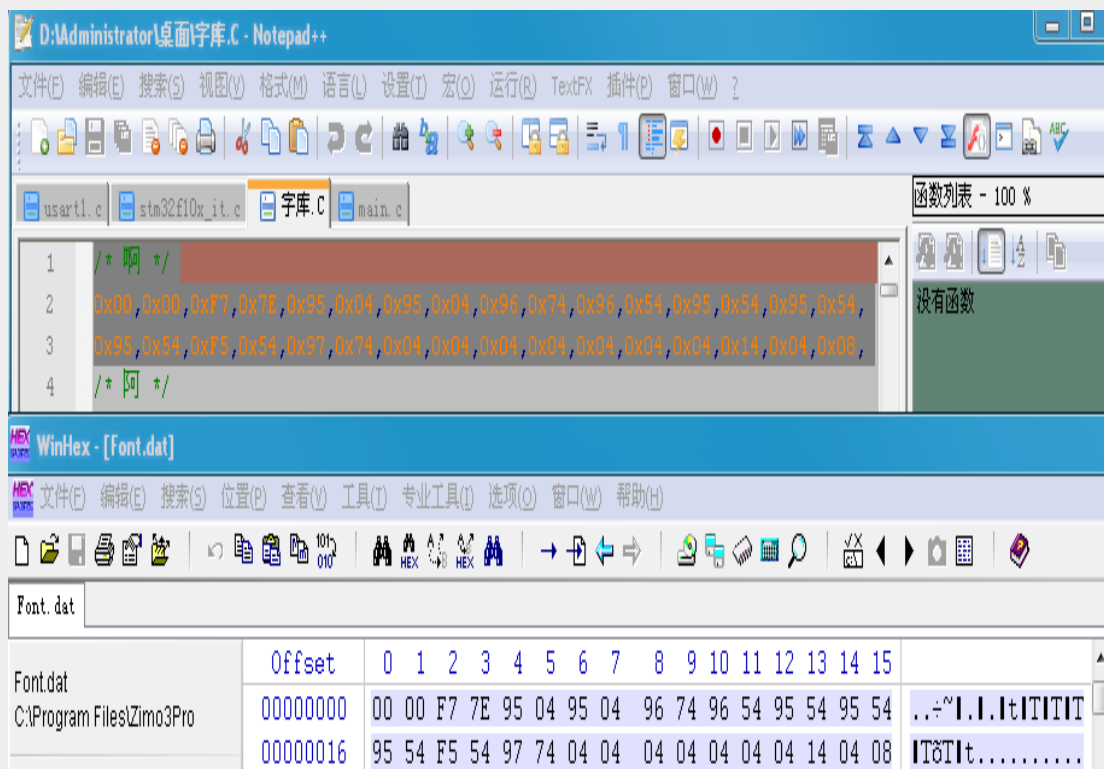


图 21-10 查看字库内容

将生成的汉字字库拷贝到 SD 卡根目录下并重命名为“HZLIB.bin”。把该文件保存到 SD 卡中,STM32 芯片通过文件系统读取文件即可获得字库的数据。

### 28.1. BMP 图片格式

使用 LCD 屏来显示 BMP 图片，就如同播放 MP3 文件一样，首先要了解其文件的格式。

BMP 文件格式，又称为位图（Bitmap）或是 DIB(Device-Independent Device, 设备无关位图)，是 Windows 系统中广泛使用的图像文件格式。BMP 文件保存了一幅图像中所有的像素。

BMP 格式可以保存单色位图、16 色或 256 色索引模式像素图、24 位真彩色图象，每种模式中单一像素点的大小分别为 1/8 字节，1/2 字节，1 字节和 32 字节。目前最常见的是 256 位色 BMP 和 24 位色 BMP。

BMP 文件格式还定义了像素保存的几种方法，包括不压缩、RLE 压缩等。常见的 BMP 文件大多是不压缩的。

Windows 所使用的 BMP 文件，在开始处有一个文件头，大小为 54 字节。保存了包括文件格式标识、颜色数、图像大小、压缩方式等信息，因为我们仅讨论 24 位色不压缩的 BMP，所以文件头中的信息基本不需要注意，只有“大小”这一项对我们比较有用。图像的宽度和高度都是一个 32 位整数，在文件中的地址分别为 0x0012 和 0x0016。54 个字节以后，如果是 16 色或 256 色 BMP，则还有一个颜色表，但在 24 位色 BMP 文件则没有，我们这里不考虑。接下来就是实际的像素数据了。因此总的来说 BMP 图片的优点是简单。

#### 28.1.1. BMP 图片分析

下面来用 WinHex 软件(跟 UltraEdit 软件功能类似)来分析一下 BMP 图像的文件内容：

测试图片为 123.bmp，先用 ACDSee 软件打开，查看图像，其图像内容见图 21-11。

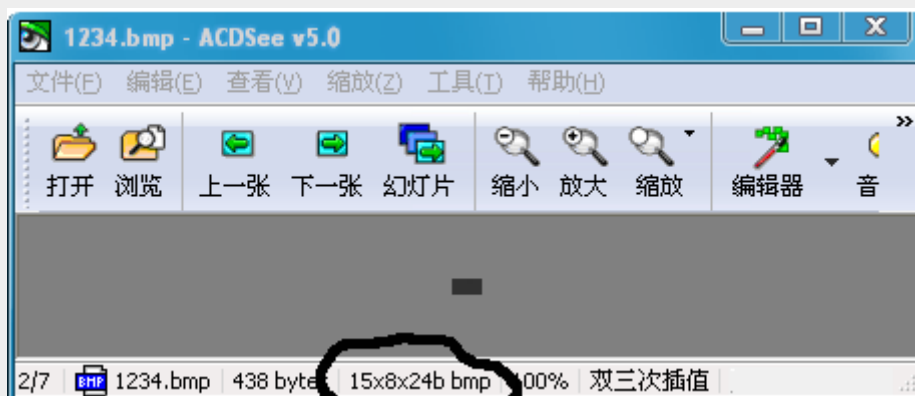


图 21-11 测试图片

可知，这幅图像的大小是 15\*8，是 24 位真彩色 bmp 图像。

接着使用 WinHex 软件打开，可读取到该文件的原始数据见图 21-12。

| 1234.bmp            |                     | Offset   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |                  |
|---------------------|---------------------|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------------|
| 1234.bmp            |                     | 00000000 | 42 | 4D | B6 | 01 | 00 | 00 | 00 | 00 | 00 | 00 | 36 | 00 | 00 | 00 | 28 | 00 | BM.....6...      |
| D:\Administrator\桌面 |                     | 00000016 | 00 | 00 | 0F | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 01 | 00 | 18 | 00 | 00 | 00 | .....            |
| 文件大小:               | 438 B               | 00000032 | 00 | 00 | 00 | 00 | 00 | 00 | C4 | 0E | 00 | 00 | C4 | 0E | 00 | 00 | 00 | 00 | .....Ä...Ä...    |
|                     | 438 字节              | 00000048 | 00 | 00 | 00 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | .....11111111    |
| 缺省编辑模式              |                     | 00000064 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 状态:                 | 原始的                 | 00000080 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 撤销级数:               | 0                   | 00000096 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
| 反向撤销:               | n/a                 | 00000112 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
|                     |                     | 00000128 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 创建时间:               | 2011-09-10 20:52:08 | 00000144 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
|                     |                     | 00000160 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 最后写入时间:             | 2011-09-10 20:52:08 | 00000176 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
|                     |                     | 00000192 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
| 属性:                 | A                   | 00000208 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 图标:                 | 1                   | 00000224 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
|                     |                     | 00000240 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
| 模式:                 | 十六进制                | 00000256 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 字符集:                | ANSI ASCII          | 00000272 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 偏移地址:               | decimal             | 00000288 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
| 每页字节数:              | 37x16=592           | 00000304 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 当前窗口:               | 1                   | 00000320 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 1111111111111111 |
| 窗口总数:               | 1                   | 00000336 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
| 剪贴板:                | 可用                  | 00000352 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 30 | 30 | 1111111111111110 |
| 暂存文件夹:              |                     | 00000368 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 0000111111111111 |
|                     | S:TEMP              | 00000384 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 111...1111111111 |
|                     |                     | 00000400 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 30 | 30 | 1111111111111110 |
|                     |                     | 00000416 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 0000111111111111 |
|                     |                     | 00000432 | 31 | 31 | 31 | 00 | 00 | 00 |    |    |    |    |    |    |    |    |    |    | 111...           |

图 21-12 测试图片的原始数据

(5) 文件头部信息部分(前面 54 字节)

| Offset   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000000 | 42 | 4D | B6 | 01 | 00 | 00 | 00 | 00 | 00 | 00 | 36 | 00 | 00 | 00 | 28 | 00 |
| 00000016 | 00 | 00 | 0F | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 01 | 00 | 18 | 00 | 00 | 00 |
| 00000032 | 00 | 00 | 00 | 00 | 00 | 00 | C4 | 0E | 00 | 00 | C4 | 0E | 00 | 00 | 00 | 00 |
| 00000048 | 00 | 00 | 00 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |

图 21-13 文件头部信息

图 21-13，阴影部分是文件头部信息。它可以分为两块：BMP 文件头和位图信息头。

### ■ 0~1 字节

BMP 文件的 0 和 1 字节用于表示文件的类型。如果是位图文件类型，必须分别为 0x42 和 0x4D，0x424D='BM'。

### ■ 3~14 字节

3 到 14 字节的意义可以用一个结构体来描述。见代码清单 21-3。

代码清单 21-3：BMP 文件信息数据结构体

```
1. typedef struct tagBITMAPFILEHEADER
2. {
3.     //attention: sizeof(DWORD)=4   sizeof(WORD)=2
4.     DWORD bfSize;           //文件大小
5.     WORD bfReserved1;      //保留字，不考虑
6.     WORD bfReserved2;      //保留字，同上
7.     DWORD bfOffBits;       //实际位图数据的偏移字节数，即前三个部分长度之和
8. } BITMAPFILEHEADER, tagBITMAPFILEHEADER;
```

### ■ 14~53 字节

头部信息剩下的部分就是位图信息头，14 到 53 字节内容的意义见代码清单 21-4。

代码清单 21-4：位图信息头内容

```
1. typedef struct tagBITMAPINFOHEADER
2. {
3.     //attention: sizeof(DWORD)=4   sizeof(WORD)=2
```

```

4.  DWORD biSize;           //指定此结构体的长度, 为 40
5.  LONG biWidth;           //位图宽, 说明本图的宽度, 以像素为单位
6.  LONG biHeight;          //位图高, 指明本图的高度, 像素为单位
7.  WORD biPlanes;          //平面数, 为 1
8.  WORD biBitCount;        //采用颜色位数, 可以是 1, 2, 4, 8, 16, 24 新的
    可以是 32
9.  DWORD biCompression;    //压缩方式, 可以是 0, 1, 2, 其中 0 表示不压缩
10. DWORD biSizeImage;      //实际位图数据占用的字节数
11. LONG biXPelsPerMeter;    //X 方向分辨率
12. LONG biYPelsPerMeter;    //Y 方向分辨率
13. DWORD biClrUsed;        //使用的颜色数, 如果为 0, 则表示默认值 (2^颜色
    位数)
14. DWORD biClrImportant;    //重要颜色数, 如果为 0, 则表示所有颜色都是重要
    的
15.
16. } BITMAPINFOHEADER, tagBITMAPINFOHEADER;

```

由上述分析与 WinHex 软件的分析内容结合得到该图片的信息如下:

- ☐ 文件大小:438
- ☐ 保留字:0
- ☐ 保留字:0
- ☐ 实际位图数据的偏移字节数:54
- ☐ 位图信息头:
- ☐ 结构体的长度:40
- ☐ 位图宽:15
- ☐ 位图高:8
- ☐ biPlanes 平面数:1
- ☐ biBitCount 采用颜色位数:24
- ☐ 压缩方式:0
- ☐ biSizeImage 实际位图数据占用的字节数:0
- ☐ X 方向分辨率:3780
- ☐ Y 方向分辨率:3780
- ☐ 使用的颜色数:0
- ☐ 重要颜色数:0

(6) 图像像素数据部分(如果是 24 位真彩色,则 54 字节之后就是像素部分):

|          |                                                 |
|----------|-------------------------------------------------|
| 00000048 | 00 00 00 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000064 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000080 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000096 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000112 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000128 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000144 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000160 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000176 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000192 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000208 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000224 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000240 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000256 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000272 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000288 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000304 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000320 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31    |
| 00000336 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000352 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 30 30 |
| 00000368 | 30 30 30 30 31 31 31 31 31 31 31 31 31 31 31    |
| 00000384 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31    |
| 00000400 | 31 31 31 31 31 31 31 31 31 31 31 31 31 31 30 30 |
| 00000416 | 30 30 30 30 31 31 31 31 31 31 31 31 31 31 31    |
| 00000432 | 31 31 31 00 00 00                               |

图 21-14 图像像素数据

有些读者可能已经发现,在像素部分夹杂着一些值为 0 的数据信息,如图 21-15 灰色部分所示。

|          |                                              |
|----------|----------------------------------------------|
| 00000096 | 31 31 31 00 00 00 31 31 31 31 31 31 31 31 31 |
|----------|----------------------------------------------|

图 21-15 像素中夹杂的 0 数据信息

本例子中整张图片都是灰色的,为什么会有 0 像素的出现呢?

对齐的数据,更容易被操作系统或者硬件调入 cache,使得 cache 的命中率提高,从而提高访问效率。也就是基于性能上得考虑,所以 Windows 规定一个扫描行所占的字节数必须是 4 的倍数(即以 long 为单位),不足的以 0 填充。由前面位图信息头的分析可知,位图宽为 15 个像素点,由于是 24 位图,每个像素点由 3 个字节构成。因此,未补充字节前字节数为 15\*3 等于 45 字节,



要到 4 的倍数，必须向上取 4 的倍数即 48，所以就有了不上 3 个字节 0 的结果。

因此，整个图像文件的大小为： $54 + (15 \times 3 + 3) \times 8$  等于 438 字节，和前面所得到的信息文件大小为 438 是一致的。

另外一点需要注意的是显示图像的顺序是 由下到上，由左到右。即像素数据部分的第一像素数据是我们见到的图像的左下角的像素的数据；而像素数据的最后一个有效数据是我们见到的图像的右上角的像素的数据。

下面我们修改图片来验证一下：

我们将左下角画上白色，右上角画上黑色，**图片放大之后**如图 21-16。

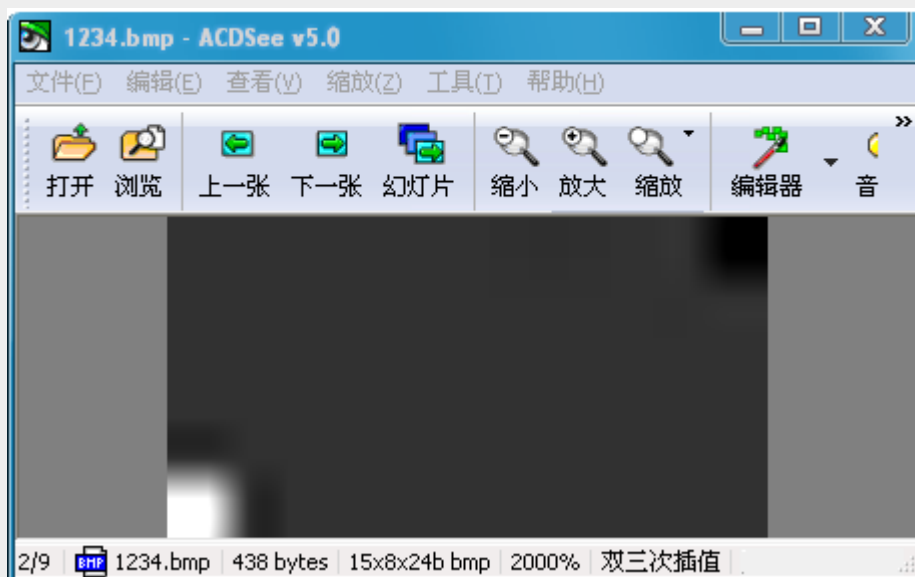


图 21-16 图片放大结果

图片数据分析如图 21-17。

| Offset   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000000 | 42 | 4D | B6 | 01 | 00 | 00 | 00 | 00 | 00 | 00 | 36 | 00 | 00 | 00 | 28 | 00 |
| 00000016 | 00 | 00 | 0F | 00 | 00 | 00 | 08 | 00 | 00 | 00 | 01 | 00 | 18 | 00 | 00 | 00 |
| 00000032 | 00 | 00 | 80 | 01 | 00 | 00 | C4 | 0E | 00 | 00 | C4 | 0E | 00 | 00 | 00 | 00 |
| 00000048 | 00 | 00 | 00 | 00 | 00 | 00 | FF | FF | FF | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000064 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000080 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000096 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000112 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000128 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000144 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000160 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000176 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000192 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000208 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000224 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000240 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000256 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000272 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000288 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000304 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000320 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000336 | 31 | 31 | 31 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000352 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 30 | 30 |
| 00000368 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 00 | 00 | 00 |
| 00000384 | 00 | 00 | 00 | 00 | 00 | 00 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 00000400 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 30 | 30 |
| 00000416 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 00 | 00 | 00 |
| 00000432 | 00 | 00 | 00 | 00 | 00 | 00 |    |    |    |    |    |    |    |    |    |    |

图 21-17 图像数据分析

我们可以看到像素部分开始部分是白色像素(灰色部分)，对应我们图像的左下角，见图 21-18。

|          |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000048 | 00 | 00 | 00 | 00 | 00 | 00 | FF | FF | FF | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

图 21-18 白色像素

后尾部分是黑色像素（灰色部分）对应我们图像的右上角，见图 21-19。

|          |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000416 | 30 | 30 | 30 | 30 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 00 | 00 | 00 |
| 00000432 | 00 | 00 | 00 | 00 | 00 | 00 |    |    |    |    |    |    |    |    |    |    |

图 21-19 黑色像素

对 BMP 图像文件内容有了具体的了解之后就可以开始编写我们的应用程序，根据图片的头部信息，合理地读出其中的像素部分，把读出的像素点数据送到 LCD 屏，就可以显示出该图片了。

## 28.2. 显示中英文及 BMP 图片实验

### 28.2.1. 实验描述及工程文件清单

**实验描述：**使用软件制作自定义类型的字库，然后将字库放入 SD 卡中，并且在 SD 卡中放入三张 BMP 图片。最后调用截屏函数截取 LCD 背景并保存为 BMP 图片。

**硬件连接：**本实验的硬件连接包括 MicroSD 卡控制信号、TFT 控制信号线及触摸屏 TSC2046 控制线，这些连接与前面章节的一样，可参照前面的电路图。

**库文件：**使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/misc.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_systick.c

FWlib/stm32f10x\_exti.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_sdio.c

FWlib/stm32f10x\_dma.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_fsmc.c

**用户文件：**

USER/main.c

USER/stm32f10x\_it.c

USER/systick.c

USER/usart1.c

USER/lcd.c

USER/ff.c

USER/ sdio\_sdcard.c

USER/lcd\_botton.c

USER/Sd\_bmp.c

USER/sd\_fs\_app.c

文件系统源文件:

使用 fatfs0.09 版本源码

ff9/diskio.c

ff9/ff.c

ff9/cc936.c

### 28.2.2. 配置工程环境

本实验需要制作字库，其文件名为：HZLIB.bin，三个 BMP 图片文件，文件名为：pic1.bmp、pic2.bmp、pic3.bmp，把这四个文件保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口。

本实验中要把旧文件：systick.c、usart1.c、lcd.c、ff.c、sdio\_sdcard.c、lcd\_botton.c 文件添加进新工程，新建 Sd\_bmp.c、sd\_fs\_app.c 文件，分别用于编写 BMP 文件相关的函数和字模获取函数。

### 28.2.3. main 文件

从 main 函数看起，它调用了很多函数，用于显示 BMP 图和沿各种方向排列的文字。见错误!未找到引用源。。

代码清单 21-5: 图片显示例程的 main 文件

```
1. int main(void)
```

```

2.  {
3.
4.     /* USART1 config */
5.     USART1_Config();
6.     SysTick_Init();
7.     LCD_Init();                               /* LCD 初始化
*/
8.     sd_fs_init();
9.
10.    /*显示图像*/
11.    Lcd_show_bmp(0, 0, "/pic3.bmp");
12.    Lcd_show_bmp( 0,0, "/pic2.bmp");
13.    Lcd_show_bmp( 0,0, "/pic1.bmp");
14.
15.    /*横屏显示*/
16.    LCD_Str_O(20, 10, "LCD_DEMO",0);
17.    LCD_Str_CH(20,30, "阿莫论坛野火专区",0,0xffff);
18.    LCD_Str_CH_O(20,50, "阿莫论坛野火专区",0);
19.    LCD_Num_6x12_O(20, 70, 65535, BLACK);
20.    LCD_Str_6x12_O(20, 90, "LOVE STM32", BLACK);
21.
22.    /*竖屏显示*/
23.    LCD_Str_O_P(300, 10, "Runing", 0);
24.    LCD_Str_CH_P(280,10, "阿莫论坛野火专区欢迎你
",0xff,0xffff);
25.    LCD_Str_CH_O_P(260,10, "阿莫论坛野火专区",0);
26.    LCD_Str_6x12_O_P(240, 10, "LOVE STM32", 0);
27.    LCD_Str_ENCH_O_P(220,10, "欢迎使用野火 stm32 开发板",0);
28.
29.    /*截图*/
30.    LCD_Str_CH(20,150, "正在截图",0,0xffff);
31.    Screen_shot(0, 0, 240, 320, "/myScreen");
32.    LCD_Str_CH(20,150, "截图完成",0,0xffff);
33.
34.
35.    while (1)
36.    {}
37. }

```

#### 28.2.4. 显示汉字

这里先说汉字字符的显示，第 17 行：LCD\_Str\_CH(20,30,"阿莫论坛野火专区",0,0xffff);

该函数功能是显示汉字字符串，源码见代码清单 21-6。

代码清单 21-6: LCD\_Str\_CH()函数

```

1.  /*****
****
2.   * 函数名: LCD_Str_CH
3.   * 描述   : 在指定坐标处显示 16*16 大小的指定颜色汉字字符串
4.   * 输入    :    - x: 显示位置横向坐标
5.   *              - y: 显示位置纵向坐标
6.   *              - str: 显示的中文字符串
7.   *              - Color: 字符颜色
8.   *              - bkColor: 背景颜色

```

```

9.  * 输出 : 无
10. * 举例 : LCD_Str_CH(0,0,"阿莫论坛野火专区",0,0xffff);
11.          LCD_Str_CH(50,100,"阿莫论坛野火专区",0,0xffff);
12.          LCD_Str_CH(320-16*8,240-16,"阿莫论坛野火专区
",0,0xffff);
13. * 注意 : 字符串显示方向为横向 已测试
14. *****/

15. void LCD_Str_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor
)
16. {
17.
18.     Set_direction(0);
19.     while(*str != '\0')
20.     {
21.         if(x>(320-16))
22.         {
23.             /*换行*/
24.             x =0;
25.             y +=16;
26.
27.         }
28.         if(y >(240-16))
29.         {
30.             /*重新归零*/
31.             y =0;
32.             x =0;
33.         }
34.         LCD_Char_CH(x,y,str,Color,bkColor);
35.         str += 2 ;
36.         x += 16 ;
37.     }
38. }

```

该函数其实没做到什么工作，对超出屏幕范围的显示坐标进行换行处理，并把字符串中的汉字一个一个提取出来调用单字符显示函数 LCD\_Char\_CH()显示出来，LCD\_Char\_CH()函数的源码见代码清单 21-7。

代码清单 21-7: LCD\_Char\_CH()函数

```

1.  /*****
2.  * 函数名: LCD_Char_CH
3.  * 描述 : 显示单个汉字字符
4.  * 输入 : x: 0~(319-16)
5.  *       y: 0~(239-16)
6.  *       str: 中文字符串首址
7.  *       Color: 字符颜色
8.  *       bkColor: 背景颜色
9.  * 输出 : 无
10. * 举例 : LCD_Char_CH(200,100,"好",0,0);
11. * 注意 : 如果输入大于1的汉字字符串，显示将会截断，只显示最前面一个汉字
12. *****/

13. void LCD_Char_CH(u16 x,u16 y,const u8 *str,u16 Color,u16 bkColor)
14. {
15.

```

```
16. #ifndef NO_CHNISEST_DISPLAY /*如果汉字显示功能没有关闭*/
17.     u8 i,j;
18.     u8 buffer[32];
19.     u16 tmp_char=0;
20.
21.
22.     GetGBKCode_from_sd(buffer,str); /* 取字模数据 */
23.
24.     for (i=0;i<16;i++)
25.     {
26.         tmp_char=buffer[i*2];
27.         tmp_char=(tmp_char<<8);
28.         tmp_char|=buffer[2*i+1];
29.         for (j=0;j<16;j++)
30.         {
31.             if ( (tmp_char >> 15-j) & 0x01 == 0x01)
32.             {
33.                 LCD_ColorPoint(x+j,y+i,Color);
34.             }
35.             else
36.             {
37.                 LCD_ColorPoint(x+j,y+i,bkColor);
38.             }
39.         }
40.     }
41.
42. #endif
43. }
```

函数中的条件编译 `#ifndef NO_CHNISEST_DISPLAY`，是用于开关汉字显示功能的，若定义了 `NO_CHNISEST_DISPLAY`，则本函数为空，关闭了显示汉字的功能。

在 `LCD_Char_CH()` 这个函数中，首先调用 `GetGBKCode_from_sd()` 从 SD 卡中读出我们需要显示在 LCD 上的指定汉字的字模数据。

接着在 22~40 行的代码就根据字模数据来描写，把字模中为 1 的数据位，在 LCD 屏中的像素点中使用画点函数 `LCD_ColorPoint()` 显示特定的颜色。思路和前面 VC 测试部分用数字“8”来显示“当”字是一样的。

## ■ 查找字模

读者现在可能在想，字库里面保存着大量的汉字字幕信息，现在输入 `GetGBKCode_from_sd(buffer,str)` 就能够拷贝出这个字符的字模数据，是怎样定位字模信息所在的位置的呢？换句话说，假如现在要显示“吾”字，是怎样根据这个字来确定“吾”字符在字库中的保存位置的呢？其实这里面有一定的映射关系，那就是接下来要说的汉字“区码”和“位码”。

在国标 GB2312—80 中规定，所有的国标汉字及符号在字库中的存储形式是：分配在一个 94 行、94 列的阵列中，阵列的每一行称为一个“区”，共有 01 区到 94 区；每一列称为一个“位”，共有 01 位到 94 位，阵列中的每一个汉字和符号所在的区号和位号组合在一起形成的四个阿拉伯数字就是它们的“区位码”。区位码的前两位是它的区号，后两位是它的位号。01~09 区是特殊符号，10~15 区未编码，16~55 区是一级汉字，56~87 区是二级汉字。例如，汉字“啊”的区位码是 1601。汉字“阿”的区位码为 1602。

在前面生成的 HZLIB.bin 文件，生成的字库是按国标 GB2312 编排的，包含了一级汉字和二级汉字，但不包含 01~15 区。所以，汉字“啊”，是存储在该文件的第 0 个位置中。汉字“阿”存储在该文件的第 1 个位置中。

汉字的机内码是指在计算机中表示一个汉字的编码。为避免与 ASCII 码混淆。用机内码的两个字节表示一个汉字，这两个字节分别称为高位字节和低位字节。

高位字节 = 区码 + 20H + 80H(或区码 + A0H)  
低位字节 = 位码 + 20H + 80H(或位码 + A0H)。

因此，我们就可以通过汉字的机内码，运算得出汉字在字库中的区位码，由区位码和文件中的偏移地址查找出该汉字的字模。

下面以 vc6.0 的测试源码来说明机内码、区位码的关系，见代码清单 21-8。

代码清单 21-8：机内码、区位码的关系

```
80. #include <stdio.h>
81. void main ()
82. {
83.     unsigned char * s , * e = "A" , * c = "古" ;
84.     unsigned char high_byte, lower_byte; //内码高字节，内码低字节
85.     printf ( "字母%s的ASCII码=", e ) ;
86.     s = e ;
87.
88.     while ( * s != 0 )                //c 的字符串以 0 为结束符*
89.     {
90.         printf ( "%3d," , * s ) ;
91.         s ++ ;
92.     }
93.     printf ( "\n 汉字内码(10 进制) '%s'=", c ) ;
94.
95.     s = c ;
96.     while ( * s != 0 )
```



```
97.     {
98.         printf ( "%3d," , * s );
99.         s ++ ;
100.     }
101.
102.         printf ( "\n 汉字内码(16 进制) '%s'=", c ) ;
103.
104.         s = c ;
105.         while ( *s != 0 )
106.         {
107.             printf ( "%0X," , * s );
108.             s ++ ;
109.         }
110.
111.
112.         s = c ;
113.         high_byte = *s;
114.
115.         s ++ ;
116.         lower_byte = *s;
117.
118.         printf("\n\n 汉字'%s'对应的\n 内码高字节:%d\n 内码低字
节:%d\n",c,high_byte,lower_byte);
119.         printf("\n\n 汉字'%s'对应的\n 区码为:%d-160 = %d\n 位码
为:%d-160 = %d\n",c,high_byte,high_byte-160,lower_byte,lower_byte-160);
120.
121.         printf("\n\n 汉字'%s'在区位码表中的位置
为%d%d\n",c,high_byte-160,lower_byte-160);
122.         printf("汉字区位码表可参考网
站:http://cs.scu.edu.cn/~wangbo/others/quweima.htm\n");
123.         printf("通过在线查阅,编号为%d%d 对应的汉字刚好就是
'%s'\n\n",high_byte-160,lower_byte-160,c);
124.
125.     }
```

测试结果如图 21-6。

```

C:\WINDOWS\system32\cmd.exe

D:\Administrator\桌面>main.exe
字母'A的ASCII码' = 65,
汉字内码<10进制>'古' =185,197,
汉字内码<16进制>'古' =B9,C5,

汉字'古'对应的
内码高字节:185
内码低字节:197

汉字'古'对应的
区码为:185-160 = 25
位码为:197-160 = 37

汉字'古'在区位码表中的位置为2537
汉字区位码表可参考网站:http://cs.scu.edu.cn/~wangbo/others/quweima.htm
通过在线查阅,编号为2537对应的汉字刚好就是'古'

D:\Administrator\桌面>pause
请按任意键继续. . .

搜狗拼音 半:
    
```

图 21-20 区位码测试效果

打开汉字区位码表在线查询网站: <http://www.jscj.com/index/gb2312.php>

查询“古”汉字的区位码刚好如计算所得, 为 。

上面的测试结果说明了每一个汉字的内码具体作用。回到本实验工程中获取字模函数 `GetGBKCode_from_sd()` 中, 它的具体定义见代码清单 21-9。

代码清单 21-9: `GetGBKCode_from_sd()` 函数

```

1.  /*****
2.   * 函数名: GetGBKCode_from_sd
3.   * 描述   : 从 sd 卡上的字库文件中拷贝指定汉字的字模数据
4.   * 输入    : pBuffer---数据保存地址
5.   *          c---汉字字符低字节码
6.   * 输出    :      0                (成功)
7.   *          -1                 (失败)
8.   *****/
9.
10. int GetGBKCode_from_sd(unsigned char* pBuffer,unsigned char * c
11. )
12. {
13.     unsigned char High8bit,Low8bit;
14.     unsigned int pos;
15.     High8bit=*c;
16.     Low8bit=*(c+1);
17.     pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16 ;
    
```

```

18.     f_mount(0, &myfs[0]);
19.     myres = f_open(&myfsrc, "0:/HZLIB.bin", FA_OPEN_EXISTING |
FA_READ);
20.
21.     if ( myres == FR_OK )
22.     {
23.         f_lseek (&myfsrc, pos);                //制定读取
位置
24.         myres = f_read( &myfsrc, pBuffer, 32, &mybr ); //16*16
大小的汉字其字模占用 16*2 个字节
25.         f_close(&myfsrc);
26.
27.         return 0;
28.     }
29.
30.     else
31.         return -1;
32.
33. }

```

第 17 行中的： $pos = ((High8bit-0xb0)*94+Low8bit-0xa1)*2*16$ ，该语句就是根据约定的映射关系，由汉字内码求得该汉字字模在字库文件中的存放位置，之后就到指定的位置去拷贝字模数据就可以了。

以汉字“啊”为例，解释该公式的含义。“啊”是 GB2312 标准中的第一个汉字，它的高位字节码(High8bit)为 16，低位字节码(Low8bit)为 01。而它的字模存放在我们的字库文件 HZLIB.bin 中的第 0 个位置。由 High8bit-0xa0 从机内码的高位字节求得其区位码 16，再由区码 16 减去 0x10(即公式中 High8bit-0xb0)，求得它在字库文件中的区码部分位置是 0。公式中还要 \*94，94 表示一个区有 94 位，由 Low8bit-0xa0 从机内码的低位字节求得其位码 1，再由位码减去 0x01(即公式中的 Low8bit-0xa1)，求得它在字库文件中的位码部分位置是 0，利用公式，求得了“啊”字的字模位置。

以上就是利用 SD 卡字库实现 LCD 显示汉字的具体流程。对于 ASCII 码(包括英文字符)的显示，原理实际上也是一样的，由于 ASCII 码占用空间较少，所以我们直接它的字模数据以数组的形式存储在代码中，这些数组在文件 ascii.h 和 asc\_font.h 中定义。

## 28.3. 实现 SD 卡 BMP 图像的读取与保存

## ■ 显示 BMP 图

再回到前面的 main 函数，其中调用了一个 BMP 图片显示函数 Lcd\_show\_bmp()，其定义见代码清单 21-10。

代码清单 21-10: Lcd\_show\_bmp()函数

```
1.  /*****
2.  * 函数名: Lcd_show_bmp
3.  * 描述   : LCD 显示 RGB888 位图图片
4.  * 输入   : x                      --显示横坐标 (0-319)
5.             y                      --显示纵坐标 (0-239)
6.             pic_name               --图片名称
7.  * 输出   : 无
8.  * 举例   : Lcd_show_bmp(0, 0, "/test.bmp");
9.  * 注意   : 图片为 24 为真彩色位图图片
10.             图片宽度不能超过 320
11.             图片在 LCD 上的粘贴范围为:纵向: [x, x+图像高
度] 横向 [Y, Y+图像宽度]
12.             当图片为 320*240 时--建议 x 输入 0 y 输入 0
13.  *****/

14. void Lcd_show_bmp(unsigned short int x, unsigned short int y, un
signed char *pic_name)
15. {
16.     int i, j, k;
17.     int width, height, l_width;
18.
19.     BYTE red, green, blue;
20.     BITMAPFILEHEADER bitHead;
21.     BITMAPINFOHEADER bitInfoHead;
22.     WORD fileType;
23.
24.     unsigned int read_num;
25.     unsigned char tmp_name[20];
26.     sprintf((char*)tmp_name, "0:%s", pic_name);
27.     f_mount(0, &bmpfs[0]);
28.
29.     BMP_DEBUG_PRINTF("file mount ok \r\n"); //使用串口输出调试信
息
30.
31.     bmpres = f_open(&bmpfsrc, (char *)tmp_name, FA_OPEN_EXIST
ING | FA_READ);
32.     Set_direction(0);
33.
34.     if(bmpres == FR_OK)
35.     {
36.         BMP_DEBUG_PRINTF("Open file success\r\n");
37.
38.         //读取位图文件头信息
39.         f_read(&bmpfsrc, &fileType, sizeof(WORD), &read_num);
40.
41.         if(fileType != 0x4d42)
```

```

42.         {
43.             BMP_DEBUG_PRINTF("file is not .bmp file!\r\n");
44.             return;
45.         }
46.         else
47.         {
48.             BMP_DEBUG_PRINTF("Ok this is .bmp file\r\n");
49.         }
50.
51.         f_read(&bmpfsrc, &bitHead, sizeof(tagBITMAPFILEHEADER), &read_num);
52.
53.         showBmpHead(&bitHead);
54.         BMP_DEBUG_PRINTF("\r\n");
55.
56.         //读取位图信息头信息
57.         f_read(&bmpfsrc, &bitInfoHead, sizeof(BITMAPINFOHEADER), &read_num);
58.         showBmpInforHead(&bitInfoHead);
59.         BMP_DEBUG_PRINTF("\r\n");
60.     }
61.     else
62.     {
63.         BMP_DEBUG_PRINTF("file open fail!\r\n");
64.         return;
65.     }
66.
67.     width = bitInfoHead.biWidth;
68.     height = bitInfoHead.biHeight;
69.
70.     l_width = WIDTHBYTES(width * bitInfoHead.biBitCount);
71.     //计算位图的实际宽度并确保它为 32 的倍数
72.     if(l_width > 960)
73.     {
74.         BMP_DEBUG_PRINTF("\nSORRY, PIC IS TOO BIG (<=320)\n");
75.         return;
76.     }
77.
78.     if(bitInfoHead.biBitCount >= 24)
79.     {
80.
81.         bmp_lcd(x, 240 - y - height, width, height);    //LCD 参数相关
82.
83.         for(i = 0; i < height + 1; i++)
84.         {
85.
86.             for(j = 0; j < l_width; j++)    //将一行数据全部读入
87.             {
88.
89.                 f_read(&bmpfsrc, pColorData + j, 1, &read_num);
90.             }
91.
92.             for(j = 0; j < width; j++) //一行有效信息
93.             {
94.                 k = j * 3;
95.
96.                 //一行中第 k 个像素的起点
97.                 red = pColorData[k + 2];
98.                 green = pColorData[k + 1];
99.                 blue = pColorData[k];
100.                LCD_WR_Data(RGB24TORGB16(red, green, blue));
101.                //写入 LCD-GRAM
102.            }

```

```

100.         }
101.         bmp_lcd_reset();    //lcd 扫描方向复原
102.     }
103.     else
104.     {
105.         BMP_DEBUG_PRINTF("SORRY, THIS PIC IS NOT A 24BITS REAL C
OLOR");
106.         return ;
107.     }
108.
109.     f_close(&bmpfsrc);
110. }

```

该函数的主要工作流程是：读取头部信息确定宽度和高度并确定每一行后面具体需要读出的字节数(保证是 4 字节的倍数)----->读取一行像素点并显示-->读取下一行并显示--> .....直至读完所有行。

另外一点就是：RGB24TORGB16 是个宏定义，因为图像数据是 RGB888 即 24 为真彩色，而我们的 LCD 是 RGB565 即 16 位色度的，所以我们需要按比例将 24 为真彩色压缩为 16 位。宏定义见代码清单 21-11。

代码清单 21-11：图像数据压缩宏

```

1. #define RGB24TORGB16(R,G,B) (((unsigned short int) (((R)>>3)<<11) | (((G)>>2)<<5) | ((B)>>3)))

```

## ■ LCD 截图功能

为了实现截图功能，我们可以根据用户截屏范围的宽和高来构造 BMP 文件的信息头，并且根据位图宽与四字节对齐的关系来补充需要的 0 大小字节。在 main 函数中，我们调用了 Screen\_shot()这个函数来截图。保存的图片是 24 位的真彩色。Screen\_shot()的源码见代码清单 21-12。

代码清单 21-12：Screen\_shot()函数

```

1. /*****
2.  * 函数名: Screen_shot
3.  * 描述   : 截取 LCD 指定位置 指定宽高的像素 保存为 24 位真彩色 bmp 格式图片

```

```

4.  * 输入   :   x                               ---水平位置
5.  *                               y               ---竖直位置
置
6.  *                               Width          ---水平宽度
7.  *                               Height         ---竖直高度
8.  *                               filename        ---文件名
9.  * 输出   :   0                               ---成功
10. *                               -1             ---失败
11. *                               8              ---文件已存在
12. * 举例   : Screen_shot(0, 0, 320, 240, "/myScreen");-----全屏截图
图
13. * 注意   : x 范围[0,319]  y 范围[0,239]  Width 范围[0,320-x]  Height
范围[0,240-y]
14. *                               如果文件已存在,将直接返回
15. *****
/
16.int Screen_shot(unsigned short int x, unsigned short int y, unsigned short int Width, unsigned short int Height, unsigned char *filename)
17.{
18.    unsigned char header[54] =
19.    {
20.        0x42, 0x4d, 0, 0, 0, 0,
21.        0, 0, 0, 0, 54, 0,
22.        0, 0, 40, 0, 0, 0,
23.        0, 0, 0, 0, 0, 0,
24.        0, 0, 1, 0, 24, 0,
25.        0, 0, 0, 0, 0, 0,
26.        0, 0, 0, 0, 0,
27.        0, 0, 0, 0, 0,
28.        0, 0, 0, 0, 0,
29.        0, 0, 0
30.    };
31.    int i;
32.    int j;
33.    long file_size;
34.    long width;
35.    long height;
36.    unsigned short int tmp_rgb;
37.    unsigned char r,g,b;
38.    unsigned char tmp_name[30];
39.    unsigned int mybw;
40.    char kk[4]={0,0,0,0};
41.
42.
43.    // if(!(Width%4))
44.    //     file_size = (long)Width * (long)Height * 3 + 54;
45.    //else
46.    file_size = (long)Width * (long)Height * 3 + Height*(Width%
4) + 54;    //宽*高 +补充的字节 + 头部信息
47.
48.    header[2] = (unsigned char)(file_size &0x000000ff);
49.    header[3] = (file_size >> 8) & 0x000000ff;
50.    header[4] = (file_size >> 16) & 0x000000ff;
51.    header[5] = (file_size >> 24) & 0x000000ff;
52.
53.
54.    width=Width;
55.    header[18] = width & 0x000000ff;
56.    header[19] = (width >> 8) & 0x000000ff;
57.    header[20] = (width >> 16) & 0x000000ff;
58.    header[21] = (width >> 24) & 0x000000ff;
59.
60.    height = Height;

```

```
61.     header[22] = height &0x000000ff;
62.     header[23] = (height >> 8) &0x000000ff;
63.     header[24] = (height >> 16) &0x000000ff;
64.     header[25] = (height >> 24) &0x000000ff;
65.
66.     sprintf((char*)tmp_name, "0:%s.bmp", filename);
67.     f_mount(0, &bmpfs[0]);
68.
69.
70.     bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_CREATE_NEW
| FA_WRITE);
71.
72.     f_close(&bmpfsrc); //新建文件之后要先关闭再打开才能写入
73.     bmpres = f_open( &bmpfsrc , (char*)tmp_name, FA_OPEN_EXISTI
NG | FA_WRITE);
74.     if ( bmpres == FR_OK )
75.     {
76.         bmpres = f_write(&bmpfsrc, header, sizeof(unsigned char)
*54, &mybw);
77.         for(i=0;i<Height;i++) //高
78.         {
79.             if(!(Width%4))
80.             {
81.                 for(j=0;j<Width;j++) //宽
82.                 {
83.                     #ifdef HX8347
84.                         tmp_rgb = bmp4(j+y, Height-i+x);
85.                     #else
86.                         tmp_rgb = bmp4(Height-
i+x, j+y);
87.                     #endif
88.
89.                     r = GETR_FROM_RGB16(tmp_rgb);
90.                     g = GETG_FROM_RGB16(tmp_rgb);
91.                     b = GETB_FROM_RGB16(tmp_rgb);
92.
93.                     bmpres = f_write(&bmpfsrc, &b, sizeof(unsign
ed char), &mybw);
94.                     bmpres = f_write(&bmpfsrc, &g, sizeof(unsign
ed char), &mybw);
95.                     bmpres = f_write(&bmpfsrc, &r, sizeof(unsign
ed char), &mybw);
96.
97.                 }
98.
99.             }
100.        else
101.        {
102.            for(j=0;j<Width;j++)
103.            {
104.                #ifdef HX8347
105.                    tmp_rgb = bmp4(j+y, Height-i+x);
106.                #else
107.                    tmp_rgb = bmp4(H
eight-i+x, j+y);
108.                #endif
109.
110.                r = GETR_FROM_RGB16(tmp_rgb);
111.                g = GETG_FROM_RGB16(tmp_rgb);
112.                b = GETB_FROM_RGB16(tmp_rgb);
113.
114.                bmpres = f_write(&bmpfsrc, &b, sizeof
(unsigned char), &mybw);
115.                bmpres = f_write(&bmpfsrc, &g, sizeof
(unsigned char), &mybw);
```



```

116.                                     bmpres = f_write(&bmpfsrc, &r, sizeof
(unsigned char), &mybw);
117.
118.
119.                                     }
120.
121.                                     bmpres = f_write(&bmpfsrc, kk, sizeof(uns
igned char)*(Width%4), &mybw);
122.
123.
124.                                     }
125.                                     }
126.
127.                                     f_close(&bmpfsrc);
128.                                     return 0;
129.                                     }
130.                                     else if ( bmpres == FR_EXIST ) //如果文件已经存在
131.                                     {
132.                                         f_close(&bmpfsrc);
133.                                         return FR_EXIST; //8
134.                                     }
135.
136.                                     else
137.                                     { f_close(&bmpfsrc);
138.                                         return -1;
139.                                     }
140.                                     }

```

该函数中，调用了 `bmp4()` 这个函数，该函数返回 LCD 上指定位置的像素信息。`GETG_FROM_RGB16`（绿色）、`GETB_FROM_RGB16`（蓝色）和 `GETR_FROM_RGB16`（红色）都是宏定义，将 RGB565 即 16 位色度抽取其中的 RGB 数据并分别将其线性映射为 8 位数据即映射为 RGB888 真彩色。宏定义的内容见代码清单 21-13。

代码清单 21-13：抽取 RGB 数据宏

```

22. #define GETR_FROM_RGB16(RGB565) ((unsigned char)(( (unsigned
short int )(RGB565) >>11)<<3)) //返回 8 位 R
23. #define GETG_FROM_RGB16(RGB565) ((unsigned char)(( (unsigned
short int )(RGB565 & 0x7ff) >>5)<<2)) //返回 8 位 G
24. #define GETB_FROM_RGB16(RGB565) ((unsigned char)(( (unsigned
short int )(RGB565 & 0x1f)<<3))) //返回 8 位 B

```

利用 `Screen_shot()` 函数，就实现了把屏幕的当前显示的图像转换为 BMP 文件的功能。

### 28.3.2. 实验现象

把文件 `HZLIB.bin`、`pic1.bmp`、`pic2.bmp`、`pic3.bmp`，保存到 SD 卡中，再把该 SD 卡插入开发板的 SD 卡接口（也可直接把工程下的 SD 字库备份文件夹

下的内容复制到 SD 卡的根目录），然后将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，接上液晶屏，将编译好的程序下载到开发板。

程序运行之后截图保存为“myScreen.bmp”如下：



图 21-21 截图结果

(截图保存图片功能+摄像头模块 就构成了一个完整的照相机啦！！)

## 29. 4、液晶触摸

本章向大家介绍如何使用 STM32 的 FSMC 接口驱动 LCD 屏，及使用触摸屏控制器检测触点坐标。

### 29. 1. LCD 控制器简介

LCD，即液晶显示器，因为其功耗低、体积小，承载的信息量大，因而被广泛用于信息输出、与用户进行交互，目前仍是各种电子显示设备的主流。

因为 STM32 内部没有集成专用的液晶屏和触摸屏的控制接口，所以在显示面板中应自带含有这些驱动芯片的驱动电路(液晶屏和触摸屏的驱动电路是独立的)，STM32 芯片通过驱动芯片来控制液晶屏和触摸屏。以野火 3.2 寸液晶屏(240\*320)为例，它使用 ILI9341 芯片控制液晶屏，通过 TSC2046 芯片控制触摸屏。

#### 29. 1. 1. ILI9341 控制器结构

液晶屏的控制芯片内部结构非常复杂，见图 4-1。最主要的是位于中间 GRAM(Graphics RAM)，可以理解为显存。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；使用 SPI 接口还是 8080 接口与 MCU 进行通讯；使用 8080 接口的哪种模式。MUC 通过 SPI 或 8080 接口与 ILI9341 进行通讯，从而访问它的控制寄存器(CR)、地址计数器(AC)、及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器(LED Controller)。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。

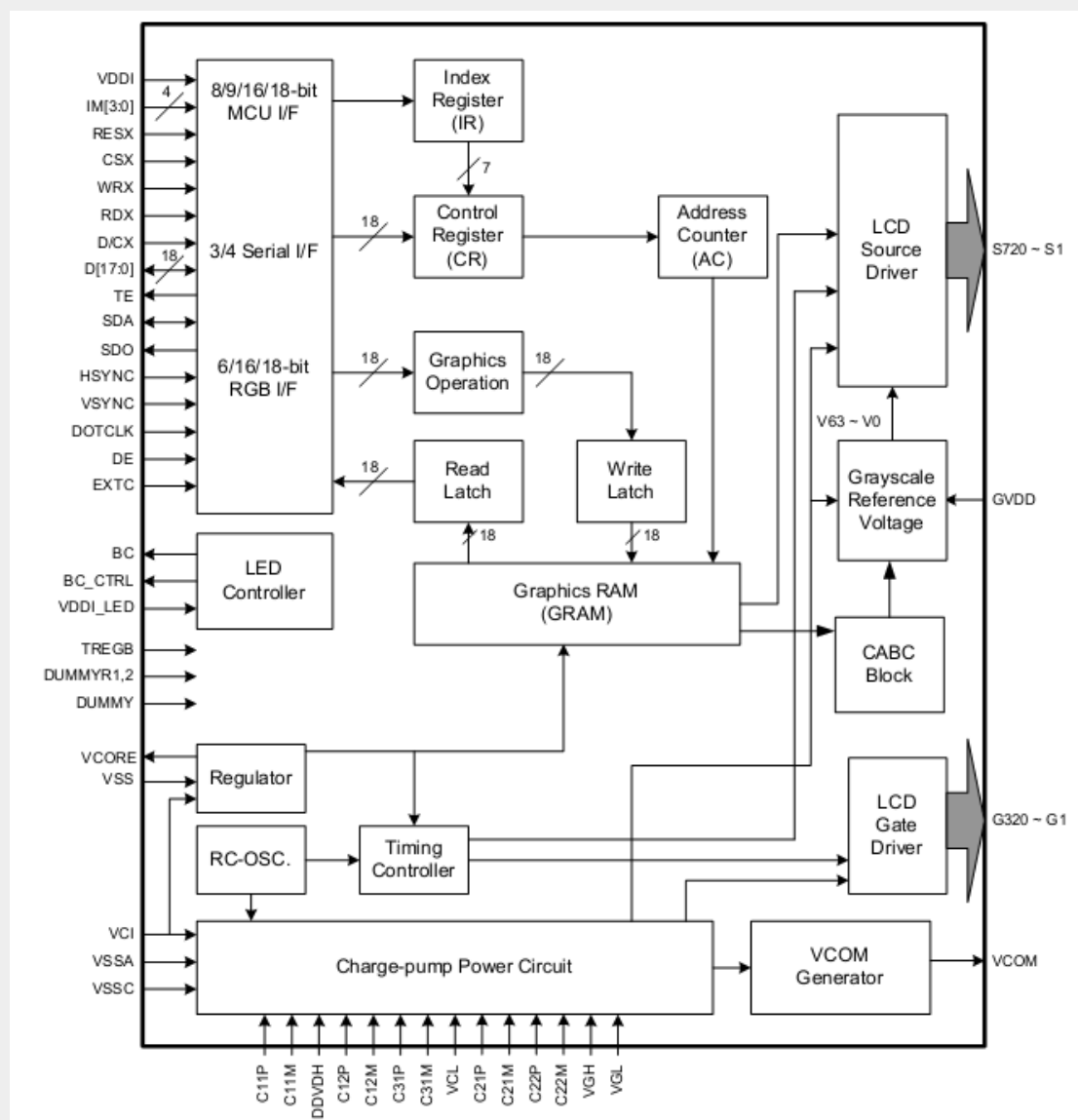


图 29-1 ILI9341 控制器内部框图

### 29.1.2. 像素点的数据格式

图像数据的像素点由红(R)、绿(G)、蓝(B)三原色组成，三原色根据其深浅程度被分为 0~255 个级别，它们按不同比例的混合可以得出各种色彩。如 R: 255, G255, B255 混合后为白色。根据描述像素点数据的长度，主要分为 8、16、24 及 32 位。如以 8 位来描述的像素点可表示  $2^8=256$  色，16 位描述的为  $2^{16}=65536$  色，称为真彩色，也称为 64K 色。实际上受人眼对颜色的识别能力的限制，16 位色与 12 位色已经难以分辨了。

ILI9341 最高能够控制 18 位的 LCD，但为了数据传输简便，我们采用它的 16 位控制模式，以 16 位描述的像素点。按照标准格式，16 位的像素点的三原色描述的位数为 R: G: B = 5: 6: 5，描述绿色的位数较多是因为人眼对绿色更为敏感。16 位的像素点格式见表 29-1 16 位像素点格式。

表 29-1 16 位像素点格式

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

图中的是默认 18 条数据线时，像素点三原色的分配状况，D1~D5 为蓝色，D6~D11 为绿色，D13~D17 为红色。这样分配有 D0 和 D12 位是无效的。若使用 16 根数据线传送像素点的数据，则 D0~D4 为蓝色，D5~D10 为绿色，D11~D15 为红色，使得刚好使用完整的 16 位。

RGB 比例为 5: 6: 5 是一个十分通用的颜色标准，在 GRAM 相应的地址中填入该颜色的编码，即可控制 LCD 输出该颜色的像素点。如黑色的编码为 0x0000，白色的编码为 0xffff，红色为 0xf800。

### 29.1.3. ILI9341 的通讯时序

目前，大多数的液晶控制器都使用 8080 或 6800 接口与 MCU 进行通讯，它们的时序十分相似，野火以 ILI9341 使用的 8080 通讯时序进行分析，实际上 ILI9341 也可以使用 SPI 接口来控制。

ILI9341 的 8080 接口有 5 条基本的控制信号线：

- ☐ 用于片选的 CSX 信号线；
- ☐ 用于写使能的 WRX 信号线；
- ☐ 用于读使能的 RDX 信号线；
- ☐ 用于区分数据和命令的 D/CX 信号线；
- ☐ 用于复位的 RESX 信号线。

其中带 X 的表示低电平有效。除了控制信号，还有数据信号线，它的数目不定，可根据 ILI9341 框图中的 IM[3:0]来设定，这部分一般由制作液晶屏的厂

家完成。为便于传输像素点数据，野火使用的液晶屏设定为 16 条数据线 D[15:0]。使用 8080 接口的写命令时序图见图 29-2。

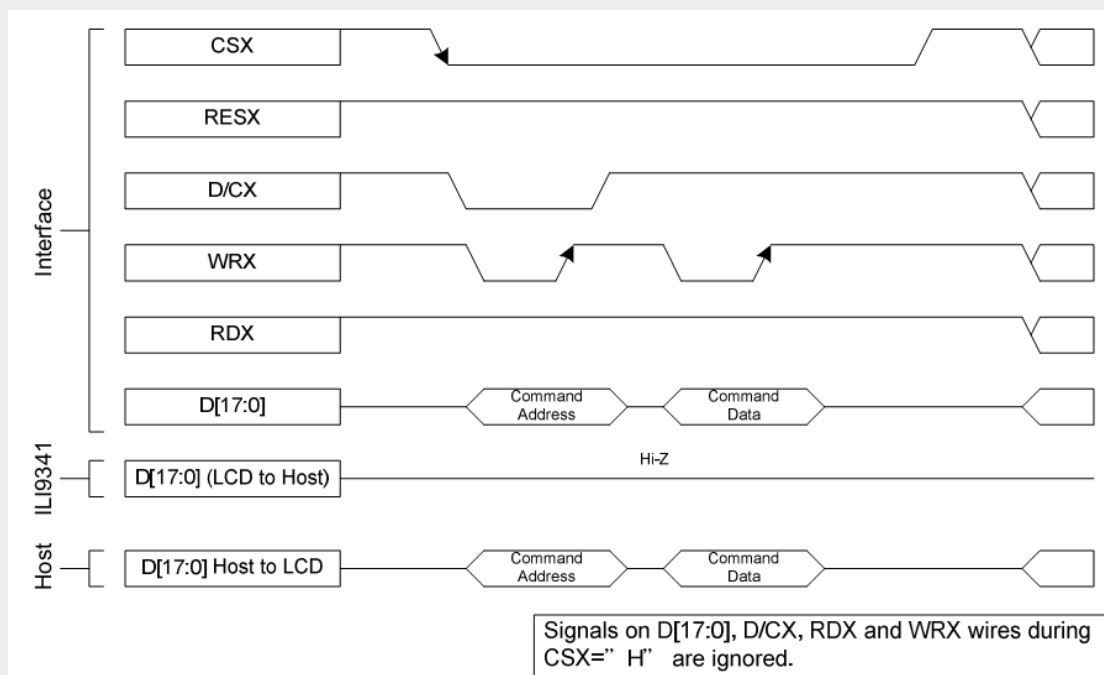


图 29-2 使用 18 条数据线的 8080 接口写命令时序

由图可知，写命令时序由 CSX 信号线拉低开始，D/CX 信号线也置低电平表示写入的是命令地址(可理解为命令编码，如软件复位命令：0x01)，以 WRX 信号线为低，RDX 信号为高表示数据传输方向为写入，同时，在数据线[17:0]输出命令地址，在第第二个传输阶段传送的为命令的参数，所以 D/CX 要置高电平，表示写入的是命令数据。

当我们需要向 GRAM 写入数据的时候，把 CSX 信号线拉低后，把 D/CX 信号线置为高电平，这时由 D[17:0]传输的数据则会被 ILI9341 保存至它的 GRAM 中。

## 29.2. 用 STM32 驱动 LCD

ILI9341 的 8080 通讯接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率较低，它提供了一种特别的控制方法——使用 FSMC 接口。

### 29.2.1. FSMC 简介

FSMC(flexible static memory controller), 译为静态存储控制器。可用于 STM32 芯片控制 NOR FLASH、PSRAM、和 NAND FLASH 存储芯片。其结构见图 29-3。

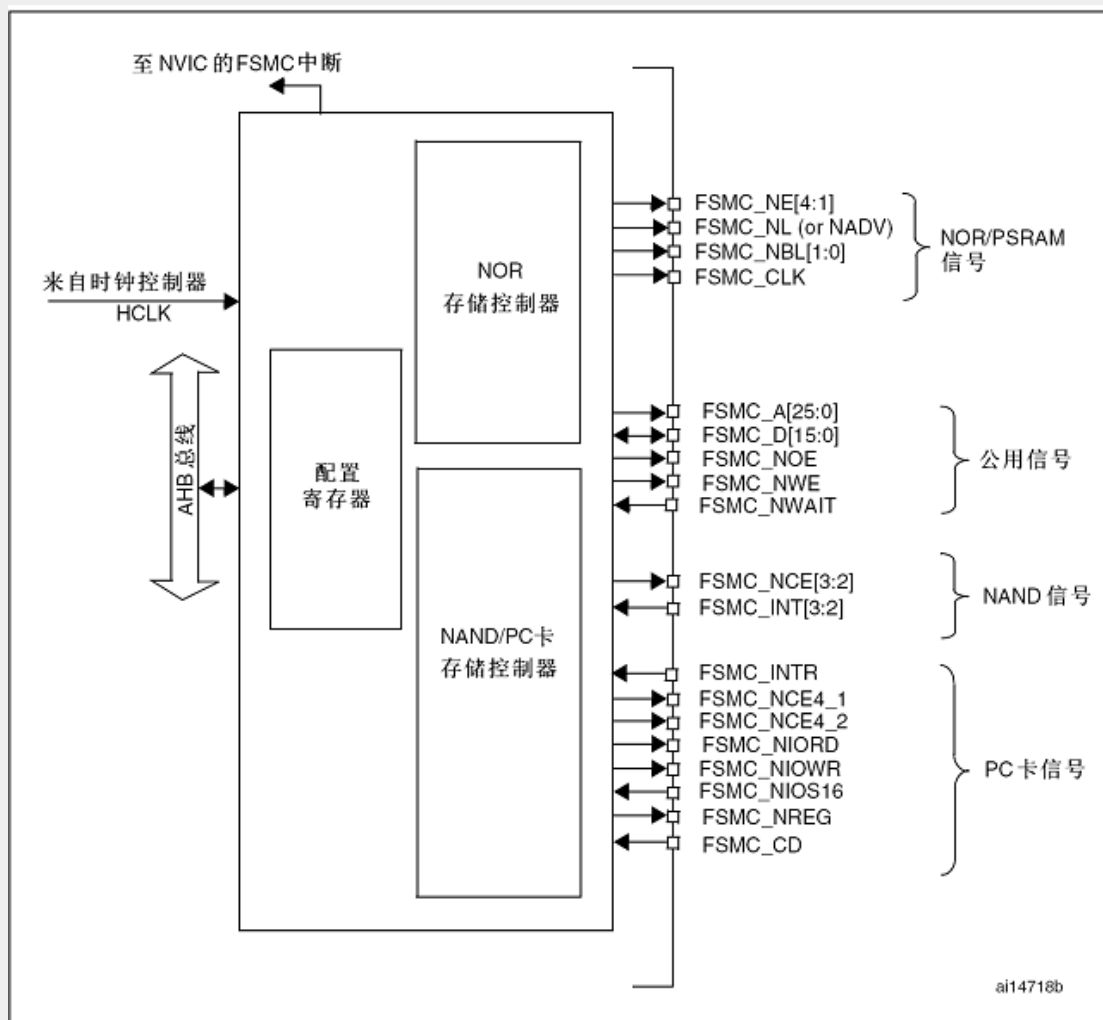


图 29-3 FSMC 结构图

我们是使用 FSMC 的 NOR\PSRAM 模式控制 LCD，所以我们重点分析框图中 NOR FLASH 控制信号线部分。控制 NOR FLASH 主要使用到表 29-2 的信号线：

表 29-2 FSMC 控制 NOR FLASH 的信号线

| FSMC 信  | 信号方   | 功能           |
|---------|-------|--------------|
| CLK     | 输出    | 时钟(同步突发模式使用) |
| A[25:0] | 输出    | 地址总线         |
| D[15:0] | 输入/输出 | 双向数据总线       |

|       |    |                     |
|-------|----|---------------------|
| NE[x] | 输出 | 片选, $x = 1...4$     |
| NOE   | 输出 | 输出使能                |
| NWE   | 输出 | 写使能                 |
| NWAIT | 输入 | NOR 闪存要求 FSMC 等待的信号 |

根据 STM32 对寻址空间的地址映射, 见前面章节的图 5-7, 地址 0x6000 0000 ~0x9FFF FFFF 是映射到外部存储器的, 而其中的 0x6000 0000 ~0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。当 FSMC 外设被配置为正常工作, 并且外部接了 NOR FLASH, 这时若向 0x60000000 地址写入数据 0xffff, FSMC 会自动在各信号线上产生相应的电平信号, 写入数据。该过程的时序图见图 29-4。

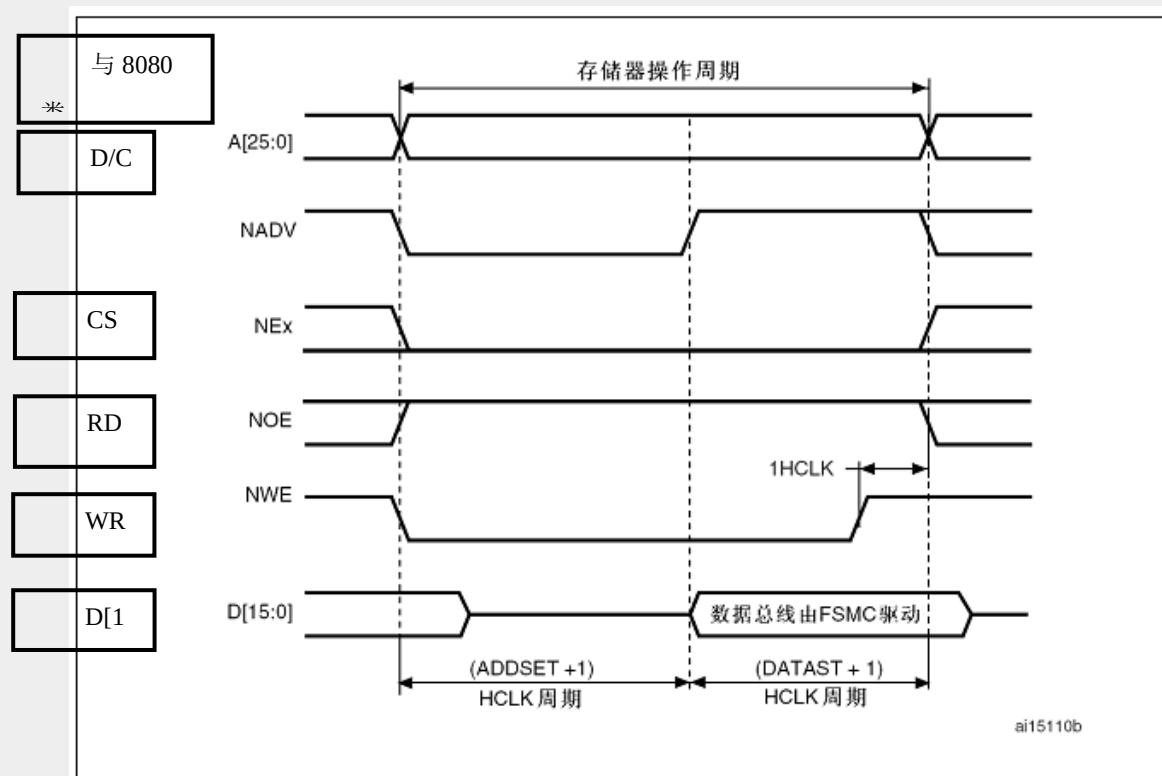


图 29-4 FSMC 写 NOR 时序图

它会控制片选信号  $NE[X]$  选择相应的某块 NOR 芯片, 然后使用地址线  $A[25:0]$  输出 0x60000000, 在  $NWE$  写使能信号线上发出写使能信号, 而要写入的数据信号 0xffff 则从数据线  $D[15:0]$  输出, 然后数据就被保存到 NOR FLASH 中了。



### 29.2.2. 用 FSMC 模拟 8080 时序

在图 29-4 的时序图中 NADV 信号是在地址、信号线复用作为锁存信号的，在此，我们忽略它。然后读者会发现，这个 FSMC 写 NOR 时序是跟 8080 接口的时序(见图 29-2)是十分相似的，对它们的信号线对比见表 29-3。

表 29-3 FSMC 的 NOR 与 8080 信号线对比

| 8080 信号线 | 功能      | FSMC-NOR<br>信号线 | 功能   |
|----------|---------|-----------------|------|
| CSX      | 片选信号    | NEx             | 片选   |
| WRX      | 写使能     | NWR             | 写使能  |
| RDX      | 读使能     | NOE             | 读使能  |
| D[15:0]  | 数据信号    | D[15:0]         | 数据信号 |
| D\CX     | 数据/命令选择 | A[25:0]         | 地址信号 |

前四种信号线都是完全一样的，仅在 8080 的数据\命令选择线与 FSMC 的地址信号线有区别。为了模拟出 8080 时序，我们把 FSMC 的 A0 地址线(也可以使用其它地址线)连接 8080 的 D\CX，即 A0 为高电平时，数据线 D[15:0]的信号会被理解 ILI9341 为数值，若 A0 为低电平时，传输的信号则会被理解为命令。

也就是说，当向地址为 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5…这些奇数地址写入数据时，地址线 A0(D/CX)会为高电平，这个数据被理解为数值；若向 0x6xxx xxx0、0x6xxx xxx2、0x6xxx xxx4…这些偶数地址写入数据时，地址线 A0(D/CX)会为低电平，这个数据会被理解为命令。

有了这个基础，只要我们在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

### 29.3. 触摸屏感应原理

触摸屏常与液晶屏配套使用，组合成为一个可交互的输入输出系统。除了熟悉的电阻、电容屏外，触摸屏的种类还有超声波屏、红外屏。由于电阻屏的控制系统简单、成本低，且能适应各种恶劣环境，被广泛采用。

电阻触摸屏的基本原理为分压，它由一层或两层阻性材料组成，在检测坐标时，在阻性材料的一端接参考电压  $V_{ref}$ ，另一端接地，形成一个沿坐标方向的均匀电场。当触摸屏受到挤压时，阻性材料与下层电极接触，阻性材料被分为两部分，因而在触摸点的电压，反映了触摸点与阻性材料的  $V_{ref}$  端的距离，而且为线性关系，而该触点的电压可由 ADC 测得。更改电场方向，以同样的方法，可测得另一方向的坐标。

### 29.4. TSC2046 触摸屏控制器

TSC2046 是专用在四线电阻屏的触摸屏控制器，MCU 可通过 SPI 接口向它写入控制字，由它测得 X、Y 方向的触点电压返回给 MCU。见图 29-5。

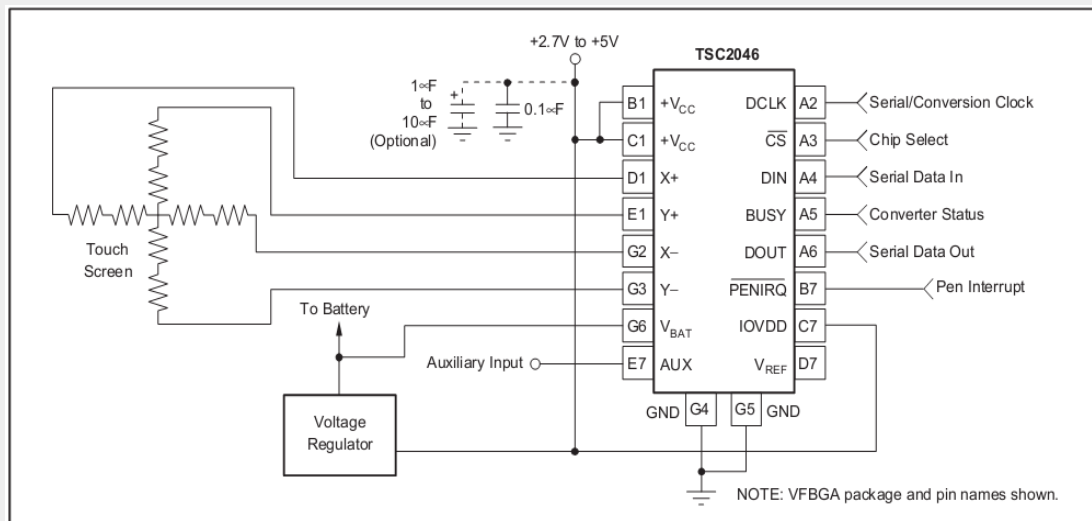


图 29-5 TSC2046 与电阻屏的连接图

图中，电阻屏两层阻性材料的两端分别接入到 TSC2046 的 X+、X-和 Y+、Y-。当要测量 X 坐标时，MCU 通过 SPI 接口写命令到 TSC2046，使它通过内部的模拟开关使 X+、X-接通电源，于是在电阻屏的 X 方向上产生一个匀强电场；把 Y+、Y-连接到 TSC2046 的 ADC。当电阻屏被触摸时，上、下两层的阻性材

料接触，在 PENIRQ 引脚产生一个中断信号，通知 MCU。该触点的电压由 Y+ 或 Y-(此时的 Y+Y-电阻很小，可忽略)引入到 ADC 进行测量，MCU 读取该电压，进行软件转换，就可以测得触点 X 方向的坐标。同理可以测得 Y 方向的坐标。

## 29.5. LCD 触摸屏画板实验

### 29.5.1. 实验描述及工程文件清单

#### 实验描述：

野火 STM32 开发板驱动配套的 3.2 寸液晶、触摸屏，使用 FSMC 接口控制该屏幕自带的液晶控制器 ILI9341，使用 SPI 接口与触摸屏控制器 TSC2046 通讯。驱动成功后可在屏幕上使用基本的触摸绘图功能。

#### 硬件连接：

TFT 数据线

|               |              |
|---------------|--------------|
| PD14-FSMC-D0  | ----LCD-DB0  |
| PD15-FSMC-D1  | ----LCD-DB1  |
| PD0-FSMC-D2   | ----LCD-DB2  |
| PD1-FSMC-D3   | ----LCD-DB3  |
| PE7-FSMC-D4   | ----LCD-DB4  |
| PE8-FSMC-D5   | ----LCD-DB5  |
| PE9-FSMC-D6   | ----LCD-DB6  |
| PE10-FSMC-D7  | ----LCD-DB7  |
| PE11-FSMC-D8  | ----LCD-DB8  |
| PE12-FSMC-D9  | ----LCD-DB9  |
| PE13-FSMC-D10 | ----LCD-DB10 |
| PE14-FSMC-D11 | ----LCD-DB11 |
| PE15-FSMC-D12 | ----LCD-DB12 |
| PD8-FSMC-D13  | ----LCD-DB13 |

|                 |                        |
|-----------------|------------------------|
| PD9-FSMC-D14    | ----LCD-DB14           |
| PD10-FSMC-D15   | ----LCD-DB15           |
| TFT 控制信号线       |                        |
| PD4-FSMC-NOE    | ----LCD-RD             |
| PD5-FSMC-NEW    | ----LCD-WR             |
| PD7-FSMC-NE1    | ----LCD-CS             |
| PD11-FSMC-A16   | ----LCD-DC             |
| PE1-FSMC-NBL1   | ----LCD-RESET          |
| PD13-FSMC-A18   | ----LCD-BLACK-LIGHT    |
| 触摸屏 TSC2046 控制线 |                        |
| PA5-SPI1-SCK    | ----TSC2046-SPI -SCK   |
| PA7-SPI1-MOSI   | ----TSC2046-SPI - MOSI |
| PA6-SPI1-MISO   | ----TSC2046-SPI - MISO |
| PB7-I2C1-SDA    | ----TSC2046-SPI-CS     |
| PB6-I2C1-SCL    | ----TSC2046- INT_IRQ   |

库文件：使用 3.5 版本固件库

Startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/misc.c

FWlib/stm32f10x\_spi.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_exti.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_fsmc.c

用户文件：

USER/main.c

USER/stm32f10x\_it.c

USER/lcd.c

USER/SysTick.c

USER/lcd\_botton.c

USER/Touch.c

## 2.4寸&3.2寸液晶接口

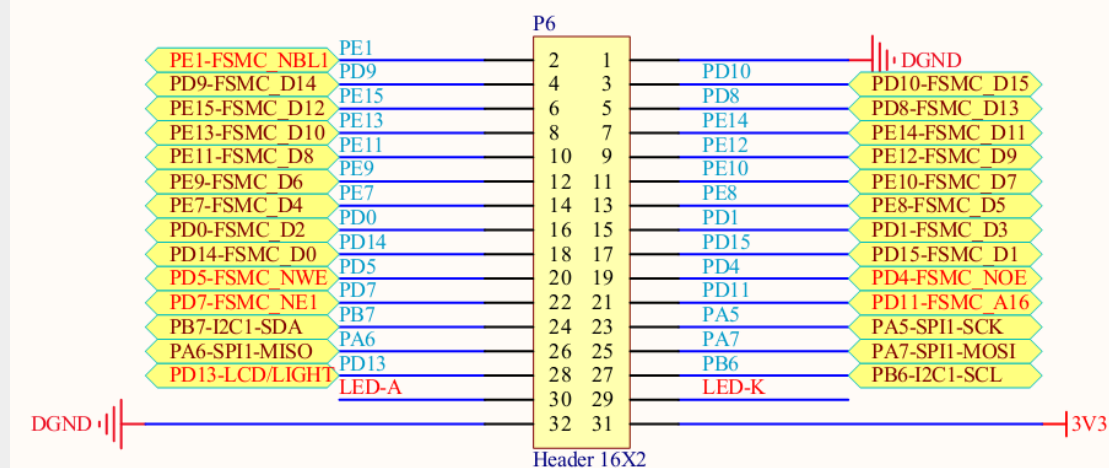


图 29-6 野火 2.4 和 3.2 寸的 LCD 接口图

### 29.5.2. 配置工程环境

本 LCD 触摸屏画板实验中我们用到了 GPIO、RCC、SPI、EXTI、FSMC 外设，所以我们先要把以下库文件添加到工程：stm32f10x\_gpio.c、stm32f10x\_rcc.c、stm32f10x\_spi.c、stm32f10x\_exti.c、stm32f10x\_fsmc.c。由于在 TSC2046 的触摸检测中使用了中断，所以还要把 misc.c 文件添加进工程。

本工程使用了旧的用户文件 SysTick.c，用作定时，把它添加到新工程之中，并新建 lcd\_botton.c、lcd.c、Touch.c 及相应的头文件。其中 lcd\_botton.c 文件定义了最底层的 LCD 控制函数，LCD 上层的函数如画点、显示字符等位于 lcd.c 文件中。

最后在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉。见代码清单 29-1。

代码清单 29-1: LCD 例程的 `stm32f10x_conf.h` 文件配置。

```
1.  /**
2.  ****
3.  * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.  * @author    MCD Application Team
5.  * @version    V3.5.0
6.  * @date      08-April-2011
7.  * @brief     Library configuration file.
8.  ****
9.
10. #include "stm32f10x_exti.h"
11. #include "stm32f10x_fsmc.h"
12. #include "stm32f10x_gpio.h"
13. #include "stm32f10x_rcc.h"
14. #include "stm32f10x_spi.h"
15. #include "misc.h"
```

### 29.5.3. main 文件

从本工程的 `main` 文件分析代码的执行流程，见代码清单 29-2。

代码清单 29-2: LCD 例程的 `main` 函数

```
1.  /**
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  */
7.  int main(void)
8.  {
9.      SysTick_Init();           /*systick 初始化*/
10.     LCD_Init();               /*LCD 初始化*/
11.     Touch_init();             /*触摸初始化*/
12.
13.     while(Touch1_Calibrate() !=0); /*等待触摸屏校准完毕
*/
14.     Init_Palette();           /*画板初始化*/
15.
16.     while (1)
17.     {
18.         if(touch_flag == 1)    /*如果触笔按下了*/
19.         {
20.             /*获取点的坐标*/
21.             if(Get_touch_point(&display, Read_2046_2(), &touch_
para ) !=DISABLE)
22.             {
23.                 /*画点*/
24.                 Palette_draw_point(display.x,display.y);
25.                 /*画点*/
26.             }
27.         }
28.     }
```

```
27.     }  
28. }
```

其执行流程如下：

② 调用 `SysTick_Init()`、`LCD_Init()`、`Touch_init()` 初始化了 STM32 的 `Systick`、`FSMC`、`SPI` 外设，并用 `FSMC` 和 `SPI` 接口初始化了 `ILI9341` 和 `TSC2046` 控制器。

③ 调用 `Touch1_Calibrate()`函数进行触摸屏校准，使得触摸屏与液晶屏的坐标匹配。

④ 调用 `Init_Palette()`函数初始化触摸画板的应用程序，使得在 LCD 上显示画板界面，并能够正常响应触摸屏的信号。

⑤ 第 16~27 行的 `while` 循环，通过不断检测触笔按下标志 `touch_flag`，判断触摸屏是否被触笔按下。触摸屏控制器 `TSC2046` 检测到触笔信号时，由它的 `PENIRQ` 引脚触发 STM32 的中断，在中断服务函数中对 `touch_flag` 标志置 1。

⑥ 第 21 行，若检测到触笔按下，调用 `Get_touch_point()`函数读取 `TSC2046` 的寄存器，获得与触点的 X、Y 坐标相关的电压信号，转化成 LCD 的 X、Y 坐标。

⑦ 获取了触点坐标后，使 LCD 液晶屏在该坐标点显示为对应的颜色。

⑧ 循环触点捕捉、画点过程，就实现了触摸画板的功能。

#### 29.5.4. 初始化 FSMC 模式

##### ■ 初始化液晶屏流程

在 `main` 函数中调用的 `LCD_Init()`函数，它对液晶控制器 `ILI9341` 用到的 `GPIO`、`FSMC` 接口进行了初始化，并且向该控制器写入了命令参数，配置好了 LCD 液晶屏的基本功能。其函数定义位于 `lcd_botton.c` 文件，见代码清单 29-3。

代码清单 29-3: LCD\_Init()函数

```
12. /*****
13. * 函数名: LCD_Init
14. * 描述   : LCD 控制 I/O 初始化
15. *         LCD FSMC 初始化
16. *         LCD 控制器 HX8347 初始化
17. * 输入   : 无
18. * 输出   : 无
19. * 举例   : 无
20. * 注意   : 无
21. *****/
22. void LCD_Init(void)
23. {
24.     unsigned long i;
25.
26.     LCD_GPIO_Config();           //初始化使用到的 GPIO
27.     LCD_FSMC_Config();          //初始化 FSMC 模式
28.     LCD_Rst();                  //复位 LCD 液晶屏
29.     Lcd_init_conf();            //写入命令参数, 对液晶屏进行基本的初始化配置
30.     Lcd_data_start();           //发送写 GRAM 命令
31.     for(i=0; i<(320*240); i++)
32.     {
33.         LCD_WR_Data(GBLUE);     //发送颜色数据, 初始化屏幕为 GBLUE 颜色
34.     }
35. }
36. }
```

LCD\_Init()函数执行后, 最直观的结果是使 LCD 整个屏幕显示编码为 0X07FF 的 GBLUE 颜色。

函数中调用的 LCD\_GPIO\_Config()主要工作是把液晶屏(不包括触摸屏)中使用到的 GPIO 引脚和使能外设时钟, 除了背光、复位用的 PD13 和 PD1 设置为通用推挽输出外, 其它的与 FSMC 接口相关的地址信号、数据信号、控制信号的端口均设置为复用推挽输出。

### ■ 初始化 FSMC 模式

接下来 LCD\_Init()函数调用 LCD\_FSMC\_Config()设置 FSMC 的模式, 我们的目的是使用它的 NOR FLASH 模式模拟出 8080 接口, 在 LCD 接口中我们使用的是 FSMC 地址线 A16 作为 8080 的 D/CX 命令选择信号的。LCD\_FSMC\_Config()具体代码见代码清单 29-4。

代码清单 29-4: LCD\_FSMC\_Config()函数



```
25. /*****
26. * 函数名: LCD_FSMC_Config
27. * 描述   : LCD_FSMC 模式配置
28. * 输入   : 无
29. * 输出   : 无
30. * 举例   : 无
31. * 注意   : 无
32. *****/
33. void LCD_FSMC_Config(void)
34. {
35.     FSMC_NORSRAMInitTypeDef FSMC_NORSRAMInitStructure;
36.     FSMC_NORSRAMTimingInitTypeDef p;
37.
38.
39.     p.FSMC_AddressSetupTime = 0x02; //地址建立时间
40.     p.FSMC_AddressHoldTime = 0x00; //地址保持时间
41.     p.FSMC_DataSetupTime = 0x05; //数据建立时间
42.     p.FSMC_BusTurnAroundDuration = 0x00; //总线恢复时间
43.     p.FSMC_CLKDivision = 0x00; //时钟分频
44.     p.FSMC_DataLatency = 0x00; //数据保持时间
45.     p.FSMC_AccessMode = FSMC_AccessMode_B;
46.     //在地址数\据线不复用的情况下, ABCD 模式的区分不大
47.     //本成员配置只有使用扩展模式才有效
48.
49.
50.     FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
51.     //NOR FLASH 的 BANK1
52.     FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable; //数据线与地址线不复用
53.     FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_NOR; //存储器类型 NOR FLASH
54.     FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b; //数据宽度为 16 位
55.     FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable; //使用异步写模式, 禁止突发模式
56.     FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low; //本成员的配置只在突发模式下有效, 等待信号极性为低
57.     FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable; //禁止非对齐突发模式
58.     FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState; //本成员配置仅在突发模式下有效。NWAIT 信号在什么时期产生
59.     FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable; //本成员的配置只在突发模式下有效, 禁用 NWAIT 信号
60.     FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable; //禁止突发写操作
61.     FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable; //写使能
62.     FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable; //禁止扩展模式, 扩展模式可以使用独立的读、写模式
63.     FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &p; //配置读写时序
64.     FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &p; //配置写时序
65.
66.     FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
67.
68.     /* 使能 FSMC Bank1_SRAM Bank */
69.     FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);
70. }
```

本函数主要使用了两种类型的结构体对 FSMC 进行配置，第一种为 `FSMC_NORSRAMInitTypeDef` 类型的结构体主要用于 NOR FLASH 的模式配置，包括存储器类型、数据宽度等。另一种的类型为 `FSMC_NORSRAMTimingInitTypeDef`，这是用于配置 FSMC 的 NOR FLASH 模式下读写时序中的地址建立时间、地址保持时间等，代码中用它定义了结构体 `p`，这个第二种类型的结构体在前一种结构体中被指针调用。

下面先来分析 `FSMC_NORSRAMInitTypeDef` 的结构体成员：

#### ❑ FSMC\_Bank

用于选择外接存储器的区域(或地址)，见图 29-7 FSMC 存储块，STM32 的存储器映射中，把 `0x6000 0000~0x9fff ffff` 的地址都映射到被 FSMC 控制的外存储器中，其中属于 NOR FLASH 的为 `0x6000 0000~0x6fff ffff`。而属于 NOR FLASH 的这部分地址空间又被分为 4 份，每份大小为 64MB，编号为 `BANK1~BANK4`。分 BANK 是由 FSMC 寻址范围决定的，该接口的地址线最多为 26 条，即最大寻址空间为  $2^{26} = 64\text{MB}$ 。为了扩展寻址空间，可把地址与数据线与多片 NOR FLASH 并联，由不同的片选信号 `NE[3:0]` 区分不同的 BANK。

在本实验中，我们使用的是 FSMC 的信号线 `NE1` 作为控制 8080 的 CSX 片选信号，所以我们把本成配置为 `FSMC_Bank1_NORSRAM1` (`NE1` 片选 `BANK1`)。

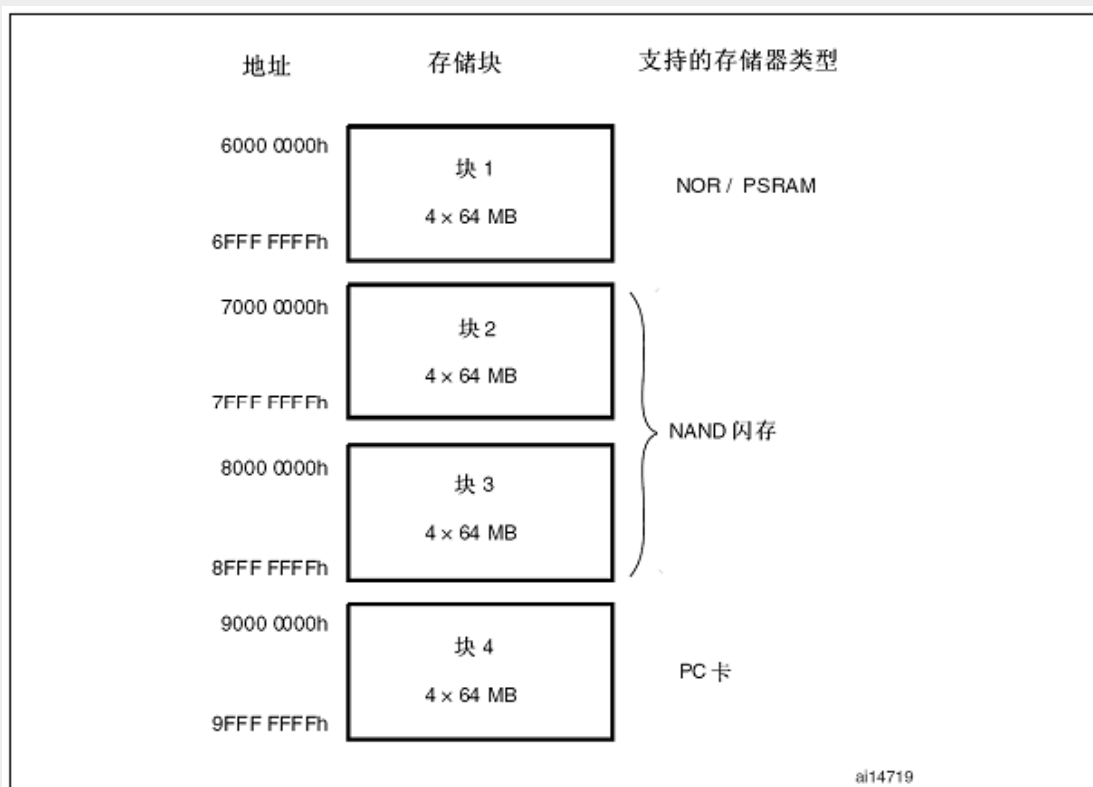


图 29-7 FSMC 存储块

#### ❑ FSMC\_DataAddressMux

本成员用于配置 FSMC 的数据线与地址线是否复用。FSMC 支持数据与地址线复用或非复用两种模式。在非复用模式下 16 位数据线及 26 位地址线分开始用；复用模式则低 16 位数据/地址线复用。在复用模式下，推荐使用地址锁存器以区分数据与地址。当 NADV 信号线为低时，复用信号线 ADx(x=0...15)上出现地址信号 Ax，当 NADV 变高时，ADx 上出现数据信号 Dx。

本实验中用 FSMC 模拟 8080 接口，地址线 A16 提供 8080 的 D/CX 信号，实际上就只使用了这一条地址线，I/O 资源并不紧张，所以把本成员配置为 FSMC\_DataAddressMux\_Disable (非复用模式)。

#### ❑ FSMC\_MemoryType

本成员用于配置 FSMC 外接的存储器的类型，可被配置为 NOR FLASH 模式、PSARM 模式及 SRAM 模式。

在本实验的应用中，由于 NOR FLASH 模式的时序与 8080 更接近，所以本结构体被配置为 FSMC\_MemoryType\_NOR(NOR FLASH 模式)。

#### ❑ FSMC\_MemoryDataWidth

本成员用于设置 FSMC 接口的数据宽度，可被设置为 8Bit 或 16bit。对于 16 位宽度的外部存储器。在 STM32 地址映射到 FSMC 接口的结构中，HADDR 信号线是需要转换到外部存储器的内部 AHB 地址线，是字节地址。

若存储器的数据线宽为 8Bit，FSMC 的 26 条地址信号线 FSMC\_A[25:0] 直接可以引入到与 AHB 相连的 HADDR[25:0]，26 条字节地址信号线最大寻址空间为 64MB。见表 29-4。

表 29-4 外部存储器地址

| 数据   | 连到存储器的地址线                      | 最大访问存储器空            |
|------|--------------------------------|---------------------|
| 8 位  | HADDR[25:0]与 FSMC_A[25:0]对应相连  | 64M 字节 x 8 = 512 M  |
| 16 位 | HADDR[25:1]与 FSMC_A[24:0]对应相连， | 64M 字节/2 x 16 = 512 |

若存储器的数据线宽 16Bit，则存储器的地址信号线是半字地址(16Bit)。为了使 HADDR 的字节地址信号线与存储器匹配，FSMC 的 25 条地址信号线 FSMC\_A[24:0]与 HADDR[25:1]相连，由于变成了半字地址(16Bit)，仅需要 25 条半字地址信号线就达到最大寻址空间 64MB。正因地址线的不对称相连，16bit 数据线宽下，实际的访问地址为右移一位之后的地址。

本实验中 8080 接口采用 16bit 模式，所以我们把本成员配置为 FSMC\_MemoryDataWidth\_16b，由于地址线不对称相连，这会影响到我们用地 址信号线 A16 控制的 8080 接口的 D/CX 信号。

#### ❑ FSMC\_BurstAccessMode

本成员用于配置访问模式。FSMC 对存储器的访问分为异步模式和突发模式(同步模式)。在异步模式下，每次传送数据都需要产生一个确定的地址，而突发模式可以在开始时提供一个地址之后，把数据成组地连续写入。

本实验中使用 FSMC 模拟 8080 端口，更适合使用异步模式，因而向本成员赋值为 FSMC\_WriteBurst\_Disable。

#### ❑ 突发模式参数配置

代码中的 30~34 行，都是关于使用突发模式时的一些参数配置，这些成员为：FSMC\_WaitSignalPolarity(配置等待信号极性)、FSMC\_WrapMode(配置是

否使用非对齐方式)、FSMC\_WaitSignalActive(配置等待信号什么时期产生)、FSMC\_WaitSignal(配置是否使用等待信号)、FSMC\_WriteBurst(配置是否允许突发写操作), 这些成员均需要在突发模式开启后配置才有效。

这些成员在开启突发模式时才有效, 本实验使用的是异步模式, 所以这些成员的参数没有意义。

#### ❑ FSMC\_WriteOperation

本成员用于配置写操作使能, 如果禁止了写操作, FSMC 不会产生写时序, 但仍可从存储器中读出数据。

本实验需要写时序, 所以向本成员赋值为FSMC\_WriteOperation\_Enable(写使能)。

#### ❑ FSMC\_ExtendedMode

本成员用于配置是否使用扩展模式, 在扩展模式下, 读时序和写时序可以使用独立时序模式。如读时序使用模式 A, 写时序使用模式 B, 这些 A、B、C、D 模式实际上差别不大, 主要是在使用数据/地址线复用的情况下, NADV 信号产生的时序不一样, 具体的时序图可查阅《STM32 参考手册》。

本实验中数据/地址线不复用, 所以读写时序中不同的 NADV 信号并没影响, 禁止使用扩展模式 SMC\_ExtendedMode\_Disable。

#### ❑ FSMC\_ReadWriteTimingStruct 及 FSMC\_WriteTimingStruct

这两个参数分别用来设置 FSMC 的读时序及写时序的时间参数。若使用了扩展模式, 则前者配置的是读时序, 后者为写时序; 若禁止了扩展模式, 则读写时序都使用 FSMC\_ReadWriteTimingStruct 结构体中的参数。

在配置这两个参数时, 使用的是类型 FSMC\_NORSRAMTimingInitTypeDef 时序初始化结构体, 对这种类型结构体的成员进行赋值。它的成员分别有: FSMC\_AddressSetupTime(地址建立时间)、FSMC\_AddressHoldTime(地址保持时间)、FSMC\_DataSetupTime(数据建立时间)、FSMC\_DataLatency(数据保持时间)、FSMC\_BusTurnAroundDuration(总线恢复时间)、FSMC\_CLKDivision(时钟分频)、FSMC\_AccessMode(访问模式)。对以上各个时间成员赋的数值 X 表示 X 个时钟周期, 它的时钟是由 HCLK 经过成员时钟分频得来的, 该分频值在成

员 FSMC\_CLKDivision(时钟分频)中设置。其中 FSMC\_AccessMode(访问模式)成员的设置只在开启了扩展模式才有效，而且开启了扩展模式后，读时序和写时序的设置可以是独立的。

本实验中的时序设置是根据 ILI9341 的 datasheet 设置的，调试的时候可以先把这些值设置得大一些，然后慢慢靠近 datasheet 要求的最小值，这样会取得比较好的效果。时序的参数设置对 LCD 的显示效果有一定的影响。

配置完初始化结构体后，要调用库函数 FSMC\_NORSRAMInit()把这些配置参数写到控制寄存器，还要调用 FSMC\_NORSRAMCmd()使能 BANK1。如果是使用 FSMC 配置其它存储器如 NAND FLASH，要使用其它的库函数及初始化结构体。

#### 29.5.5. FSMC 模拟 8080 读写参数、命令

回到 LCD\_Init()函数的执行流程，初始化完成 FSMC 接口后，就可以使用它控制 ILI9341 了。在 LCD\_Init()中调用了 Lcd\_init\_conf()函数向 ILI9341 写入了一系列的控制参数，见代码清单 29-5。

代码清单 29-5: Lcd\_init\_conf()函数

```
/******  
* 函数名: Lcd_init_conf  
* 描述   : ILI9341 LCD 寄存器初始配置  
* 输入   : 无  
* 输出   : 无  
* 举例   : 无  
* 注意   : 无  
*****/  
void Lcd_init_conf(void)  
{  
    DEBUG_DELAY();  
    LCD_ILI9341_CMD(0xCF);  
    LCD_ILI9341_Parameter(0x00);  
    LCD_ILI9341_Parameter(0x81);  
    LCD_ILI9341_Parameter(0x30);  
  
    DEBUG_DELAY();  
    LCD_ILI9341_CMD(0xED);  
    LCD_ILI9341_Parameter(0x64);  
    LCD_ILI9341_Parameter(0x03);  
    LCD_ILI9341_Parameter(0x12);  
    LCD_ILI9341_Parameter(0x81);  
  
    DEBUG_DELAY();  
}
```

```
LCD_ILI9341_CMD(0xE8);  
LCD_ILI9341_Parameter(0x85);  
LCD_ILI9341_Parameter(0x10);  
LCD_ILI9341_Parameter(0x78);  
// .....此处省略几十行.....
```

本函数十分长，由于篇幅问题，以上只是该函数其中的一部分，省略部分的代码也是这样的模板，只是写入的命令和参数不一样而已，这些命令和参数设置了像素点颜色格式、屏幕扫描方式、横屏\竖屏等初始化配置，这些命令的意义从 ILI9341 的 datasheet 命令列表中可以查到。该函数通过调用 LCD\_ILI9341\_CMD()写入命令，用 LCD\_ILI9341\_Parameter()写入参数。它们实质是两个宏，见代码清单 29-6。

代码清单 29-6: LCD\_ILI9341\_CMD()和 LCD\_ILI9341\_Parameter()宏

```
1.  /*****  
2.  #define LCD_ILI9341_CMD(index)      LCD_WR_REG(index)  
3.  #define LCD_ILI9341_Parameter(val)  LCD_WR_Data(val)  
4.  
5.  /****为了移植方便，上面的宏只是封装，以下才是最底层的宏*****/  
6.  /* 选择 BANK1-BORSRAM1 连接 TFT，地址范围为 0X60000000~0X63FFFFFF  
7.  * FSMC_A16 接 LCD 的 DC(寄存器/数据选择)脚  
8.  * 16 bit => FSMC[24:0]对应 HADDR[25:1]  
9.  * 寄存器基地址 = 0X60000000  
10. * RAM 基地  
址 = 0X60020000 = 0X60000000+2^16*2 = 0X60000000 + 0X20000 = 0X60020000  
  
11. * 当选择不同的地址线时，地址要重新计算。  
12. */  
13.  
14. #define Bank1_LCD_D      ((u32)0x60020000)      //Disp Data ADDR  
  
15. #define Bank1_LCD_C      ((u32)0x60000000)      //Disp Reg ADDR  
  
16.  
17. /*选定 LCD 指定寄存器（命令编码）*/  
18. #define LCD_WR_REG(index)      ((*(__IO u16 *) (Bank1_LCD_C)) = (  
(u16)index))  
19.  
20. /*往 LCD 写入数据*/  
21. #define LCD_WR_Data(val)      ((*(__IO u16 *) (Bank1_LCD_D)) =  
(u16)(val)))
```

这部分是 FSMC 模拟 8080 接口的精髓。



## ■ 读写参数、命令

先来看第 21 行的宏，LCD\_WR\_Data(val)，这是一个带参宏，用于向 LCD 控制器写入参数，参数为 val。它的宏展开为：

```
1. ((*(__IO u16 *) (Bank1_LCD_D)) = ((u16)(val)))
```

宏展开中的(Bank1\_LCD\_D)是一个在第 14 行定义的宏，它的值为 0x6002 0000，实质是一个地址，这个地址的计算在后面介绍。

(\_\_IO u16 \*) (Bank1\_LCD\_D)表示把(Bank1\_LCD\_D)强制转换成一个 16 位的地址。((\*(\_\_IO u16 \*) (Bank1\_LCD\_D))表示再对这个地址作“\*”指针运算，取该指针对象的内容，并把它的内容赋值为 = ((u16)(val)))。所以整个宏的操作就是：把参数 val 写入到地址为 0x6002 0000 的地址空间。

由于这个地址被 STM32 映射到外存储器，所以会由 FSMC 外设以访问 NOR FLASH 的形式、时序，在地址线上发出 0x6002 0000 地址信号，在数据线上发出 val 数据信号，写入参数到外存储器中。而 FSMC 接口又被我们模拟成了 8080 接口，最终 val 被 8080 接口理解为参数，传输到 ILI9341 控制器中。

## ■ 计算地址

见图 29-8。计算地址前，再明确一下在本实验中，使用的是 FSMC\_NE1 作为 8080\_CS 片选信号，以 FSMC\_A16 作为 8080\_D/CX 数据/命令信号(图中为 RS，意义相同)。

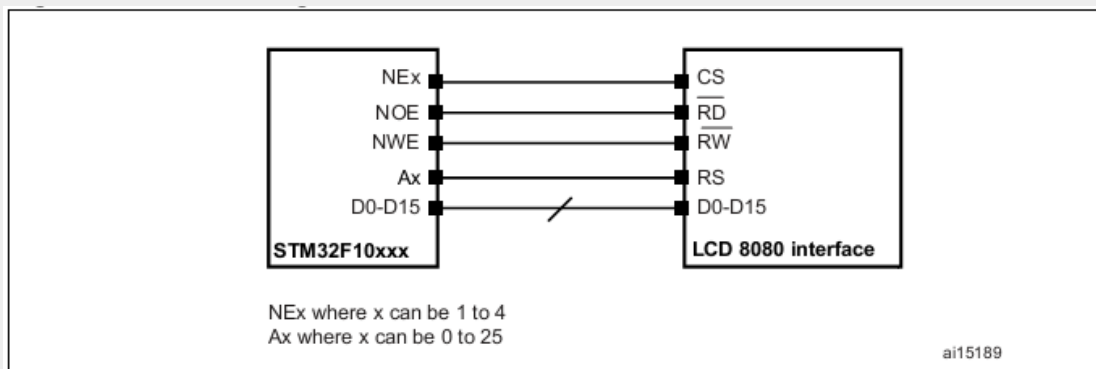


图 29-8 FSMC 与 8080 端口连接简图



按这种连接时，FSMC\_NE1 为低电平、FSMC\_A16 为高电平，表示通过 D[15:0]发送\接收的数据被 8080 接口解释为参数(数值)，当我们访问 0x6002 0000 这个地址的时候，正好符合这个条件。该地址的计算过程如下：

由于选择的是使用 FSMC\_NE1 片选信号线，片选的为 BANK1，所以基地址为 0x6000 0000。要把地址线 FSMC\_A16 置为高电平，可以采用下列算式：

```
(5) 0x6000 0000 |= 1<<16; //结果 = 0x6001 0000
```

但是，这样计算出来的地址只是数据线为 8Bit 模式下的字节地址。由于我们采用的是 16Bit 数据线，FSMC[24:0]与 HADDR[25:1]对齐，HADDR 地址要左移一位才是 FSMC 的访问地址。因此为了把 FSMC 中的 FSMC\_A16 (D/CX 线)置 1，实际上要对应到 HADDR 地址(AHB 地址)的 HADDR\_A17，即正确的计算公式应为：

```
1. 0x6000 0000 |= 1<<(16+1); //此为正确结果 = 0x6002 0000
```

对于 16 位数据线模式，能使 FSMC\_NEX 为低电平，FSMC\_A16 为高电平的地址，并不只有 0x6002 0000 一个，只要是属于 0x60000000~0x63FFFFFF 范围内(BANK1 地址范围)，HADDR\_A17 位为高电平的地址均可，这是因为我们只采用了 FSMC\_A16 用于 8080 的 D/CX 信号，所以地址线的电平状态并无影响。如 0x6002 0001 地址，在本实验中是与 0x6002 0000 等价的。

若修改这个地址，使 FSMC\_NEX 为低电平，FSMC\_A16 为低电平，即 D/CX 被置 0，8080 会把由数据线传输的信号理解为命令。以同样的方式计算，符合这样要求的其中一个地址为 0x6000 0000，向这个地址空间赋值，这个值最终会被 8080 接口解释为命令。宏 LCD\_WR\_REG(index) 就是这样实现的。

## ■ 给整个屏幕上色

再次回到 LCD\_Init()函数，它调用完 Lcd\_init\_conf()初始化了液晶屏后，向使用 Lcd\_data\_start()函数发送了一个命令——写 GRAM 内容，即后面发送的数据都被解析为显示到屏幕像素点的数据。代码中使用 for 循环把语句

LCD\_WR\_Data(GBLUE)执行了 320\*240 次，即把所有像素点都显示为 GBLUE 颜色。

### 29.5.6. 液晶屏画点函数

初始化了液晶屏后，就可以控制液晶上每个像素点的颜色了。如果能够实现一个画点函数，在指定的(x,y)坐标像素点上显示指定的颜色，那么就能够实现一切液晶屏最复杂的显示功能，如在液晶屏指定位置显示形状、文字、图像，都可以通过调用画点函数或以类似的方式控制液晶的像素点。本实验中的画点函数 LCD\_Point()在 lcd.c 文件中定义，见代码清单 29-7。

代码清单 29-7： LCD\_Point()函数

```
1.  /*****
2.   * 函数名: LCD_Point
3.   * 描述   : 在指定坐标处显示一个点
4.   * 输入   : -x 横向显示位置 0~319
5.               -y 纵向显示位置 0~239
6.   * 输出   : 无
7.   * 举例   :      LCD_Point(100,200);
8.               LCD_Point(10,200);
9.               LCD_Point(300,220);
10.  * 注意   :      (0,0)位置为液晶屏左上角 已测试
11.  *****/

12. void LCD_Point(u16 x,u16 y)
13. {
14.     LCD_open_windows(x,y,1,1);
15.     LCD_WR_Data(POINT_COLOR);
16. }
```

本函数首先调用了—个液晶显示窗口函数 LCD\_open\_windows()用于开辟液晶屏上的显示区域，后面写入的颜色数据将被显示到该区域中。示窗函数按(x,y,1,1)的参数调用时，该函数开辟了一个坐标为(x,y)的像素点。接着调用 LCD\_WR\_Data()向 ILI9341 写入颜色数据，像素的坐标即为示窗函数开辟的显示坐标。于是 LCD\_Point()函数就实现了控制特定坐标的像素点。

接下来分析 LCD\_open\_windows()函数是如何开辟显示区域的，见代码清单 29-8。

代码清单 29-8： LCD\_open\_windows()函数

```
1.  /*****
2.   * 函数名: LCD_open_windows
```

```

3.  * 描述   : 开窗(以 x,y 为坐标起点, 长为 len, 高为 wid)
4.  * 输入   : -x      窗户起点
5.             -y      窗户起点
6.             -len    窗户长
7.             -wid    窗户宽
8.  * 输出   : 无
9.  * 举例   : 无
10. * 注意   : 无
11. *****/
12. void LCD_open_windows(u16 x,u16 y,u16 len,u16 wid)
13. {
14.
15.     if(display_direction == 0)        /*如果是横屏选项*/
16.     {
17.
18.         LCD_ILI9341_CMD(0X2A);
19.         LCD_ILI9341_Parameter(x>>8);    //start 起始位置的高 8
位
20.         LCD_ILI9341_Parameter(x-((x>>8)<<8)); //起始位置的低 8
位
21.         LCD_ILI9341_Parameter((x+len-1)>>8); //end 结束位置的
高 8 位
22.         LCD_ILI9341_Parameter((x+len-1)-(((x+len-
1)>>8)<<8)); //结束位置的低 8 位
23.
24.         LCD_ILI9341_CMD(0X2B);
25.         LCD_ILI9341_Parameter(y>>8);    //start
26.         LCD_ILI9341_Parameter(y-((y>>8)<<8));
27.         LCD_ILI9341_Parameter((y+wid-1)>>8); //end
28.         LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-
1)>>8)<<8));
29.
30.     }
31.     else
32.     {
33.         LCD_ILI9341_CMD(0X2B);
34.         LCD_ILI9341_Parameter(x>>8);
35.         LCD_ILI9341_Parameter(x-((x>>8)<<8));
36.         LCD_ILI9341_Parameter((x+len-1)>>8);
37.         LCD_ILI9341_Parameter((x+len-1)-(((x+len-
1)>>8)<<8));
38.
39.         LCD_ILI9341_CMD(0X2A);
40.         LCD_ILI9341_Parameter(y>>8);
41.         LCD_ILI9341_Parameter(y-((y>>8)<<8));
42.         LCD_ILI9341_Parameter((y+wid-1)>>8);
43.         LCD_ILI9341_Parameter((y+wid-1)-(((y+wid-
1)>>8)<<8));
44.
45.     }
46.
47.     LCD_ILI9341_CMD(0x2c);
48. }

```

本函数主要使用了三个 ILI9341 的控制命令：

### (1) 0x2A 列地址控制命令

用于设置显示区域的列像素区域。它有四个参数，分别为 SC 的高 8 位、SC 的低 8 位及 EC 的高 8 位、EC 的低 8 位。SC 和 EC 的意义见图 29-9。SC 表示要控制的显示区域的列起始坐标，EC 表示显示区域的列结束坐标。

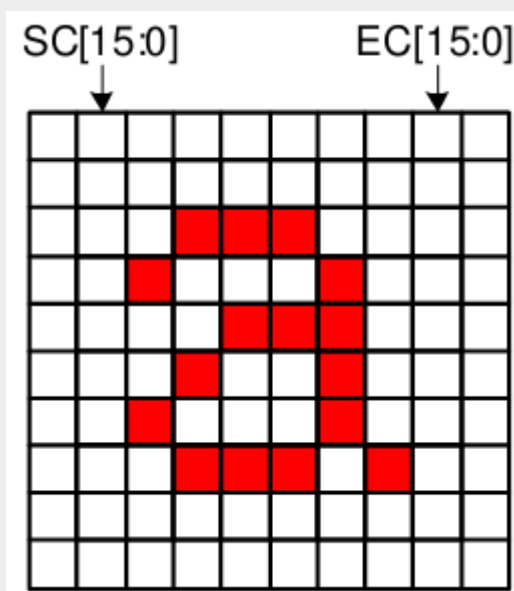


图 29-9 列控制命令 SC 和 EC

## (2) 0x2B 页地址控制命令

用于设置显示区域的 页(行)像素区域。与上一命令类似，也有四个参数，分别为 SP 的高 8 位、SP 的低 8 位及 EP 的高 8 位、EP 的低 8 位。SP 和 EP 的意义见图 29-10。SP 表示要控制的显示区域的行起始坐标，EP 表示显示区域的行结束坐标。

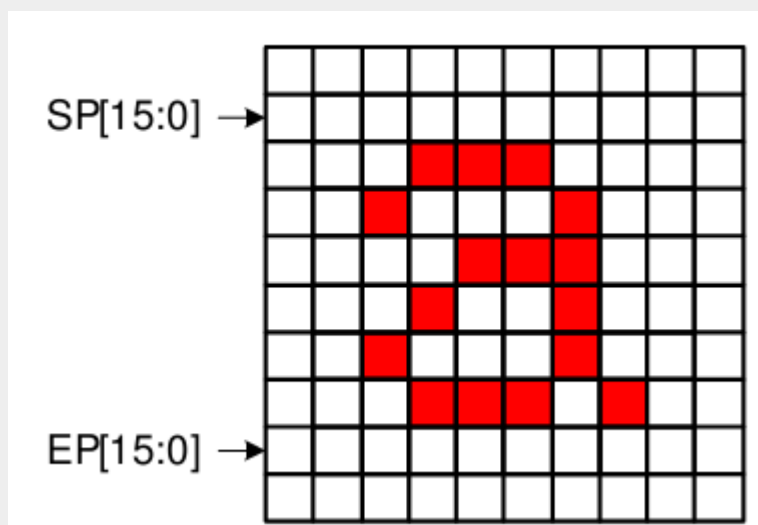


图 29-10 页控制命令 SP 和 EP

### (3) 0x2C 写 RAM 命令

本命令用于表示开始写入像素显示数据，紧跟着本命令后面的即为写入到GRAM 的 RGB 5: 6: 5 的颜色数据。在初始化液晶屏函数中也调用到本命令。使用本命令，后面的颜色数据按照一个接着一个预设的扫描方式(扫描方式决定了是横屏和竖屏)写入到由 0x2A 和 0x2B 设置的显示区域中。横屏扫描方式为从左到右，扫描完一行(页)像素点再扫描下一行(页)。竖屏扫描为从上到下，扫描完一列再扫描下一列。

了解这三个命令后，就知道示窗函数的执行流程了。图 29-11。

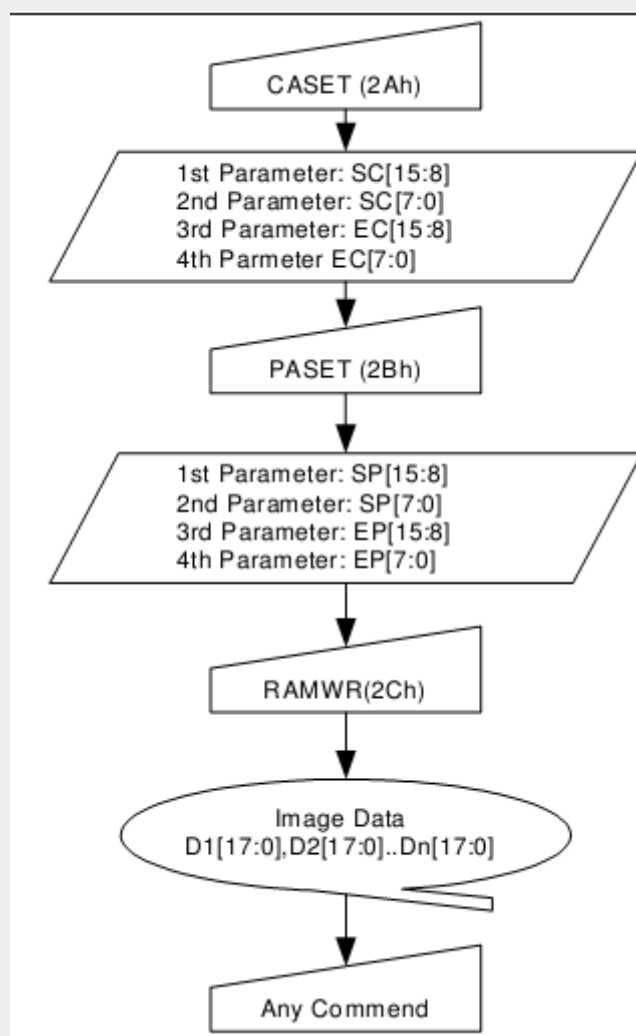


图 29-11 示窗函数执行流程

调用本示窗函数后，就可以开辟一个起始坐标为(x,y)，长为 len 宽为 wid 的矩形显示窗口了。特别地，当 len=wid=1 时，开辟的为一个坐标为(x,y)的像素点。

### 29.5.7. 触摸屏校正

在 main 函数初始化完成 LCD 之后，调用了 Touchl\_Calibrate()函数进行触摸屏校正，见代码清单 29-9。

代码清单 29-9: Touchl\_Calibrate()函数

```

1.  /*****
2.  * 函数名: Touchl_Calibrate
3.  * 描述   : 触摸屏校正函数
4.  * 输入   : 无
5.  * 输出   : 0    --- 校正成功
6.             1    --- 校正失败
7.  * 举例   : 无
8.  * 注意   : 无
9.  *****/

10. int Touchl_Calibrate(void)
11. {
12.     uint8_t i;
13.     u16 test_x=0, test_y=0;
14.     u16 gap_x=0, gap_y=0;
15.     Coordinate * Ptr;
16.     // delay_init();
17.     Set_direction(0);    //设置为横屏
18.     for(i=0;i<4;i++)
19.     {
20.         LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR); //使整个
           个屏幕显示背景颜色
21.         LCD_Str_6x12_O(10, 10,"Touch Calibrate", 0); //显示提示
           信息
22.         LCD_Num_6x12_O(10,25, i+1, 0); //显示触点
           次数
23.
24.         delay_ms(500);
25.         DrawCross(DisplaySample[i].x,DisplaySample[i].y); //显
           示校正用的“十”字
26.         do
27.         {
28.             Ptr=Read_2046(); //读取 TSC2046 数据到变量 ptr
29.         }
30.         while( Ptr == (void*)0 ); //当 ptr 为 0 时表示没有触点被按
           下
31.         ScreenSample[i].x= Ptr->x; //把读取的原始数据存放到
           ScreenSample 结构体
32.         ScreenSample[i].y= Ptr->y;
33.
34.     }
35.
36.     /* 用原始参数计算出 原始参数与坐标的转换系数。 */
37.     Cal_touch_para( &DisplaySample[0],&ScreenSample[0],&touch_p
           ara );
38.
39.     /*计算 x 值*/
40.     test_x = ( (touch_para.An * ScreenSample[3].x) +
41.                (touch_para.Bn * ScreenSample[3].y) +
42.                touch_para.Cn
43.                ) / touch_para.Divider ;

```

```
44.
45.     /*计算 Y 值*/
46.     test_y = ( (touch_para.Dn * ScreenSample[3].x) +
47.               (touch_para.En * ScreenSample[3].y) +
48.               touch_para.Fn
49.               ) / touch_para.Divider ;
50.
51.     gap_x = (test_x > DisplaySample[3].x)?(test_x -
DisplaySample[3].x):(DisplaySample[3].x - test_x);
52.     gap_y = (test_y > DisplaySample[3].y)?(test_y -
DisplaySample[3].y):(DisplaySample[3].y - test_y);
53.
54.
55.     LCD_Rectangle(0,0,320,240,CAL_BACKGROUND_COLOR);
56.     if((gap_x>11)|| (gap_y>11))
57.     {
58.
59.         LCD_Str_6x12_O(100, 100,"Calibrate fail", 0);
60.         LCD_Str_6x12_O(100, 120," try again ", 0);
61.         delay_ms(2000);
62.         return 1;
63.     }
64.
65.
66.     aa1 = (touch_para.An*1.0)/touch_para.Divider;
67.     bb1 = (touch_para.Bn*1.0)/touch_para.Divider;
68.     cc1 = (touch_para.Cn*1.0)/touch_para.Divider;
69.
70.     aa2 = (touch_para.Dn*1.0)/touch_para.Divider;
71.     bb2 = (touch_para.En*1.0)/touch_para.Divider;
72.     cc2 = (touch_para.Fn*1.0)/touch_para.Divider;
73.
74.     LCD_Str_6x12_O(100, 100,"Calibrate Success", 0);
75.     delay_ms(1000);
76.
77.     return 0;
78. }
```

本函数的主要作用是在指定的几个液晶屏坐标(逻辑坐标)显示“十”字交叉点，由用户使用触笔点击触摸屏交叉点，读取由 TSC2046 测得的触点电压(物理坐标)。采集 4 个不同位置的触点电压(物理坐标)，然后根据触摸校准算法把逻辑坐标与物理坐标转换公式的系数 A、B、C、D、E、F 计算出来。

若使用此函数校准成功后，用户再点击触摸屏时，可把测量出的触点电压(物理坐标)代入已知系数的转换公式，计算出对应的液晶屏坐标(逻辑坐标)。

转换公式的系数为以上代码 66~72 行中的 aa1、bb1、cc1、aa2、bb2、cc2 这几个全局变量，如果把这几个数据保存在非易失性存储器（SD 卡、EEPROM 等）中，上电后向这几个变量赋值，就不需要每次上电都进行一次触屏校准了。

本函数中大部分都是关于触摸屏校准算法的数学运算，有兴趣的读者可查阅其它相关资料来理解。在代码中的 18~30 行，与触摸屏的触点电压获取有关，分析如下：

22. 第 20~25 行，调用 LCD\_Rectangle()、LCD\_Str\_6x12\_O()、LCD\_Num\_6x12\_O()、DrawCross()由液晶屏显示背景、提示信息及校准用的“十”字。这些函数都与液晶的画点函数原理类似，关于字符显示的在下一个章节进行说明。

23. 第 28 行，调用 Read\_2046()函数获取触点的电压，该函数通过向 TSC2046 控制器发送控制命令：若触笔点击触摸屏时采集触点的电压，采集 10 个电压取平均值，结果返回给变量 Ptr；若没有触点，则 Ptr 的值为 0，由 do-while 循环等待至采集到数据为止。Ptr 中保存的电压数据在后面被用于校准算法计算。

Read\_2046()函数定义见代码清单 29-10。

代码清单 29-10: Read\_2046()函数

```
72. /*****
73. * 函数名: Read_2046
74. * 描述   : 得到滤波之后的 X Y
75. * 输入   : 无
76. * 输出   : Coordinate 结构体地址
77. * 举例   : 无
78. * 注意   : 速度相对比较慢
79. *****/

80. Coordinate *Read_2046(void)
81. {
82.     static Coordinate  screen;
83.     int m0,m1,m2,TP_X[1],TP_Y[1],temp[3];
84.     uint8_t count=0;
85.
86.     /* 坐标 x 和 y 进行 9 次采样*/
87.     int buffer[2][9]={0},0};
88.     do
89.     {
90.         Touch_GetAdXY(TP_X,TP_Y);
91.         buffer[0][count]=TP_X[0];
92.         buffer[1][count]=TP_Y[0];
93.         count++;
94.
95.     } /*用户点击触摸屏时即 TP_INT_IN 信号为低 并且 count<9*/
96.     while(!INT_IN_2046&& count<9);
97.
98. //由于篇幅问题，此处省略很多行，省略部分主要为计算 10 个采样电压的平均值
99.
100.     .....
```



101.

在 `Read_2046()`函数中，调用了 `Touch_GetAdXY()`，它用于获取一次触点 (x,y) 电压。实际上，驱动 TSC2046 最底层的是命令 `WR_CMD(CHX)` 和 `WR_CMD(CHY)`，发送了这两个命令后，TSC2046 开始采集相应的触点电压，通过 SPI 传送触点电压数据到 STM32。

命令语句中的 CHX 宏展开为 0xd0，CHY 为 0x90，它们是根据 TSC2046 的命令格式设定的。驱动 TSC2046 的命令控制字格式，见表 29-5。

表 29-5 TSC2046 命令控制字

| 位         | 位 6 | 位 5 | 位 4 | 位 3 | 位 2     | 位 1 | 位 0 |
|-----------|-----|-----|-----|-----|---------|-----|-----|
| TMCS<br>S | A2  | A1  | A0  | MOD | SER DFR | PD1 | PD0 |

其中 S 为数据传输起始标志位该位必为 1，A2~A0 进行通道选择，MODE 用于转换精度选择，1 为 8 位精度，0 为 12 位精度，见表 29-6。

表 29-6 通道选择

| 2   | 1   | 0 | + | - | Y  | X  | Y  | Y- | X- | Z1-  | Z | 驱     |
|-----|-----|---|---|---|----|----|----|----|----|------|---|-------|
| REF | REF |   |   |   | +  | +  | 位置 | 位置 | 位置 | 2- 置 | 位 | 动     |
|     |     |   | Y | Y |    | +  |    | M  |    |      |   | Y     |
|     |     |   | + | - |    | IN |    |    |    |      |   | + 9Y- |
|     |     |   | Y | X |    | +  |    |    |    | M    |   | Y     |
|     |     |   | + | - |    | IN |    |    |    |      |   | + 9X- |
|     |     |   | Y | Y | +  |    |    |    |    |      | M | Y     |
|     |     |   | + | + | IN |    |    |    |    |      |   | + 9X- |
|     |     |   | X | X |    |    | +  |    | M  |      |   | X     |
|     |     |   | + | - |    |    | IN |    |    |      |   | + 9X- |

所谓通道选择即为检测哪一个通道的坐标。如 A2~A0 为 001 时，即命令控制字为 0x90，根据表格知，芯片会给触摸屏的 Y 阻性材料层的两端提供 Y+、Y- 的电压，若有触笔点击，则 Y 触点电压可经过 X+ 利用 ADC 读取得。同

理命令控制字为 0xd0 时, A2~A0 为 101, 即给 X 阻性材料层提供电压, 触点电压经过 Y+ 由 ADC 读取得。这就是 TSC2046 采集触点电压的原理。

#### 29.5.8. 检测触点、画点

回到 main 函数。触摸屏也校准好后, 剩下的就是应用程序代码了, 调用 Init\_Palette() 使液晶屏显示出画板的界面, 见图 29-12 (由于印刷原因, 无法分辨具体颜色)。

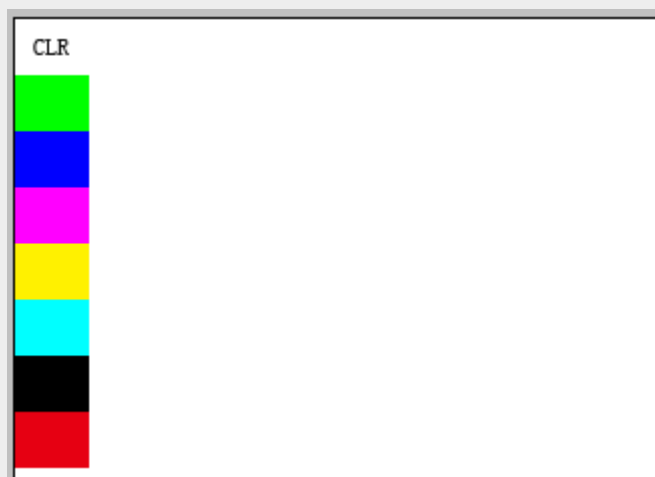


图 29-12 画板界面

该画板的界面左侧为各种颜色方块, 右侧为提供给用户进行绘画的空间。显示完该界面后, 循环检测 touch\_flag 标志, 它在中 stm32f10x\_it.c 文件中的中断服务函数被赋值。当触摸屏被按下时会进入该函数, 对 touch\_flag 赋值, 见代码清单 29-11。

代码清单 29-11: 触摸屏的中断服务函数

```
void EXTI9_5_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line6) != RESET)
    {
        GPIO_ResetBits(GPIOB, GPIO_Pin_5);

        touch_flag=1;

        EXTI_ClearITPendingBit(EXTI_Line6);
    }
}
```

若 `touch_flag` 标志为 1，即触摸屏被按下，`main` 中调用函数 `Get_touch_point()`，该函数通过 `Read_2046_2()` 获取触点电压，根据公式把电压转换为液晶坐标，并保存到 `display` 结构体中。得到液晶坐标后，`main` 调用 `Palette_draw_point()` 对相应的坐标点进行处理，若触点位于画板界面的颜色方块中，则使画笔变为该颜色，其后在空白界面的触点将显示该颜色的笔迹。

#### 29.5.9. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，接上液晶屏，将编译好的程序下载到开发板。运行后，可在 LCD 屏幕看到提示信息“Touch Calibrate”和触摸屏校正用的“十”字。点击“十”字的中间（共四个），若校正成功后会出现画板界面，在画板界面的右侧可进行绘画，点击左侧的颜色块可选择笔迹的颜色。

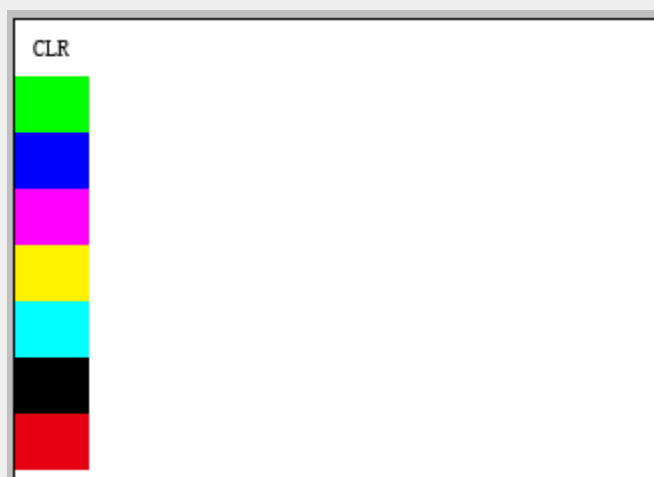


图 29-13 触摸画板界面

## 30. 5、摄像头-OV7725

在各种信息中，图像含有最丰富的信息，作为机器视觉领域的核心部件，摄像头被广泛地应用在安防、探险、车牌检测等场合。STM32 的处理速度比传统的 8、16 位机快得多，所以使用它驱动摄像头采集图像信息并进行基本的加工处理更为适合，本章讲解使用 STM32 驱动 OV7670 型号的摄像头。

### 30. 1. 摄像头的分类

摄像头按输出信号的类型来看可以分为数字摄像头和模拟摄像头，按照摄像头图像传感器材料构成来看可以分为 CCD 和 CMOS。

#### 30. 1. 1. 数字摄像头跟模拟摄像头区别

37. 输出信号类型：数字摄像头输出信号为数字信号，模拟摄像头输出信号为标准的模拟信号。

38. 接口类型：数字摄像头有 usb 接口(比如常见的 pc 端免驱摄像头)、IEEE1394 火线接口(由苹果公司领导的开发联盟开发的一种高速度传送接口，数据传输率高达 800Mbps)、千兆网接口（网络摄像头）。模拟摄像头多采用 AV 视频端子（信号线+地线）或 S-VIDEO（即莲花头--SUPER VIDEO，是一种五芯的接口，由两路视频亮度信号、两路视频色度信号和一路公共屏蔽地线共五条芯线组成）。

39. 分辨率：模拟摄像头的感光器件，其像素指标一般维持在 752(H)\*582(V)左右的水平，像素数一般情况下维持在 41W 左右。数字摄像头分辨率一般从数十万到数百万甚至数千万(比如诺基亚 808 手机搭载了 4100 万像素，被人称为可以打电话的相机)。但这并不能说明数字摄像头的成像分辨率就比模拟摄像头的高，原因在于模拟摄像头输出的是模拟视频信号，一般直接输入至电视或监视器，其感光器件的分辨率与电视信号的扫描数呈一定的换算关系，图像的显示介质已经确定，因此模拟摄像

头的感光器件分辨率不是不能做高，而是依据于实际情况没必要做这么高。

### 30.1.2. CCD 与 CMOS 的区别

CCD 与 CMOS 成像器主要区别如下：

40. 成像原理：CCD 是“电荷耦合器件”(Charge Coupled Device)的简称，而 CMOS 是“互补金属氧化物半导体”(Complementary Metal Oxide Semiconductor)的简称。

41. 功耗：由于 CCD 的像素由 MOS 电容构成，读取电荷信号时需使用电压相当大(至少 12V)的二相或三相或四相时序脉冲信号，才能有效地传输电荷。因此 CCD 的取像系统除了要有多个电源外，其外设电路也会消耗相当大的功率。有的 CCD 取像系统需消耗 2~5W 的功率。而 CMOS 光电成像器件只需使用一个单电源 5V 或 3V，耗电量非常小，仅为 CCD 的 1/8~1/10，有的 CMOS 取像系统只消耗 20~50mW 的功率。

42. 成像质量：CCD 成像器件制作技术起步早，技术成熟，采用 PN 结或二氧化硅(sio2)隔离层隔离噪声，所以噪声低，成像质量好。与 CCD 相比，CMOS 的主要缺点是噪声高及灵敏度低，不过随着 CMOS 电路消噪技术的不断进展，为生产高密度优质的 CMOS 成像器件提供了良好的条件(现在高级的 CMOS 并不比一般 CCD 差，但是 CMOS 工艺还不是十分成熟，普通的 CMOS 分辨率低而成像较差，太容易出现杂色点)，现在主流的单反相机普遍采用 CMOS 成像器。

### 30.2. OV7670 介绍

OV7670 是一个能够提供单片 VGA 摄像头和影像处理器的所有功能的图像传感器，它可以输出整帧，子采样，取窗口等方式的各种分辨率 8/10 位图像数据，支持的数据格式有很多种，包括 RAW RGB，RGB (GRB 4:2:2, RGB565/555/444) 以及 YCbCr (4:2:2) 等格式。它的体积小，工作电压低，具

有单片 VGA 摄像头影像处理器的功能，可以对图像进行伽玛曲线、白平衡、饱和度和色度等处理。

### 30.2.1. ov7670 功能框架

OV7670 芯片的各个功能模块见图 30-1。

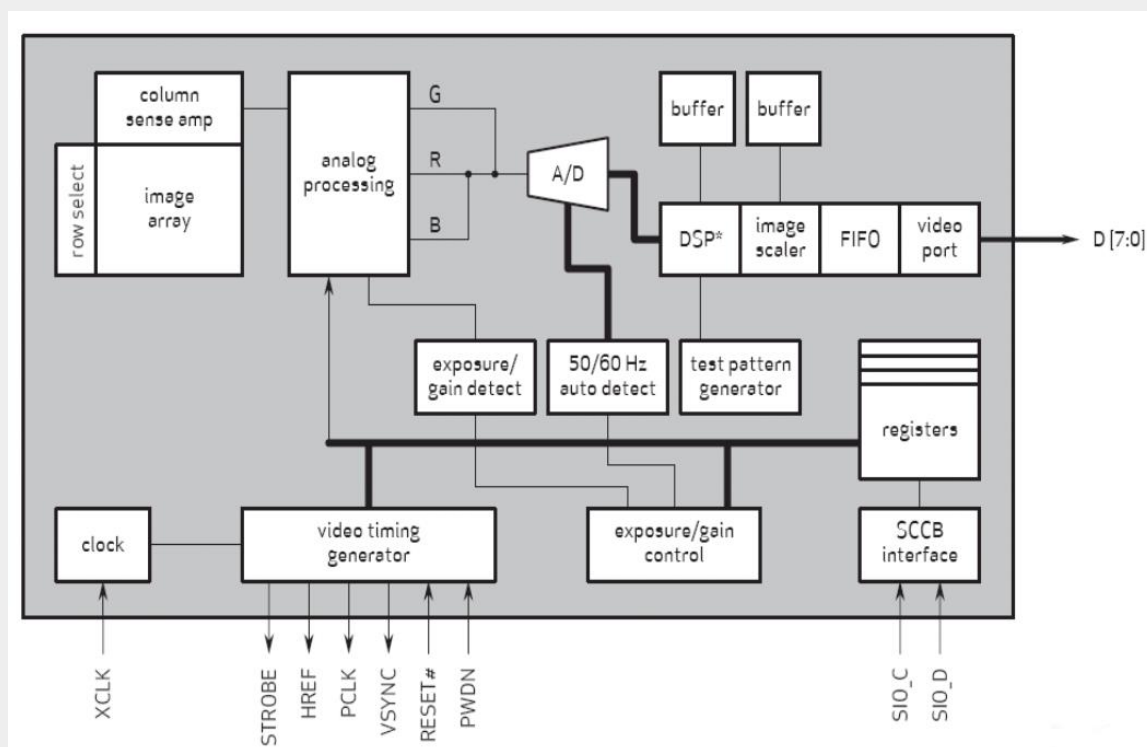


图 30-1 OV7670 功能框架图

OV7670 包含有  $656 \times 488$  图像阵列，共包含 320128 个像素，但是实际有效的还是  $640 \times 480$  个，边缘多出来的像素主要是为了去除一些影响边缘失真特性而设立的。图像阵列感知的原始信号(颜色&亮度)之后会输送入模拟处理器进行处理(比如通常所说的曝光控制和增益控制)，经处理后会分成 G 和 BR 两路通道进入一个 A/D 转换器，转换成数字信号后送入 DSP 处理器作进一步处理。

当然，用户还可以通过相关寄存器来切换选择把测试图案发生器产生的图形数据送入 DSP 处理器处理，DSP 模块在整个框架中起着非常重要的作用，它控制着从原始信号插值到 RGB 信号的整个过程，并且控制这图像的质量包括边缘增强、颜色空间转换、gamma 控制等，这些操作的实现可能还会依赖于 buffer 的支持，这跟我们平时做大量数据的复杂运算时类似，在这里同样需要开辟一定空间的 buffer，否则很多中间数据无法安置。

图像处理后会放入一个图像缩放模块，缩放模块按照默认配置或者用户配置的要求，支持将 YUV/RGB 信号从 VGA 缩小到 CIF 以下的任何尺寸，再进入一个 FIFO，最后通过视频端口将数据传递给用户。另外，我们还可以看到，系统时钟经过视频时序发生器后，根据系统或者用户配置，产生了视频采集常见的三种信号：行同步信号，场同步信号，像素时钟；SENSOR 内部的寄存器可以通过 SCCB 接口来访问，SCCB 总线之后我们会单独描述。

30.2.2. OV7670 管脚封装

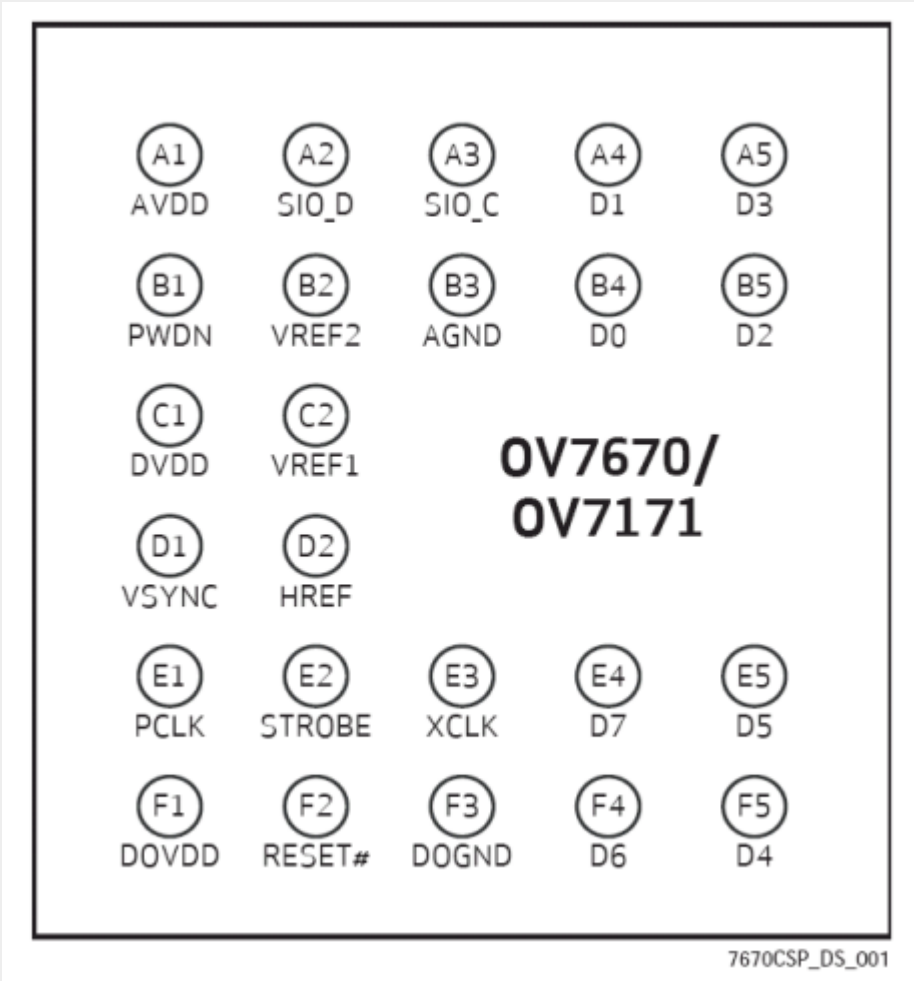


图 30-2 OV7670 管脚图

主要管脚的定义见表 30-1。

表 30-1 OV7670 管脚

| 管脚名称 | 管脚类型 | 管脚描述 |
|------|------|------|
|------|------|------|

|         |     |                |
|---------|-----|----------------|
| ADVDD   | 电源  | 供给 ADC 模块使用    |
| RSTB    | 输入  | 系统复位管脚，低电平有效   |
| SCL     | 输入  | SCCB 总线的时钟线    |
| SDA     | I/O | SCCB 总线的数据线    |
| AVDD    | 电源  | 模拟电源           |
| DOVDD   | 电源  | 数字电源           |
| HREF    | 输出  | 行同步信号          |
| PWDN    | 输入  | 掉电/省电模式(高电平有效) |
| VSYNC   | 输出  | 场同步信号          |
| PCLK    | 输出  | 像素时钟           |
| XCLK    | 输入  | 系统时钟输入端口       |
| D0...D9 | 输出  | 像素数据端口         |

当 SENSOR 配置为输出 RAW RGB 数据时，数字端口 D0 到 D9 都要用到，D0 作为最低有效位，D9 作为最高有效位。当 SENSOR 配置为输出 YUV 或者 RGB 565/RGB 555 格式时，数字端口的 D2 到 D9 才有效，D2 作为最低有效位，D9 作为最高有效位。

### 30.3. SCCB 总线

SCCB 总线全称为 Serial Camera Control Bus，它的工作方式与 I<sup>2</sup>C 十分类似，是由 OV 公司定义的 3 线串行摄像头控制总线，可以控制大部分 OV 系列图像传感器。SCCB 亦可以工作在 2 线串行模式(SIOC 与 SIOD)，一条 SCCB 总线下可以挂在多个从设备(通过从设备地址来区分)，另外 SCCB 还可以附带一根 PWDN 用于关闭或者开启从设备系统。

#### 30.3.1. sccb 接口定义

□ 标准的 3 线 SCCB 接口定义见表 30-2 和图 30-3。

表 30-2 3 线 SCCB 接口定义

| 信号 | 信号类 | 描述 |
|----|-----|----|
|----|-----|----|



| 名称     | 型   |                                                                                   |
|--------|-----|-----------------------------------------------------------------------------------|
| SCCB_E | 输出  | 片选信号。当总线处于空闲状态，主机应该将其逻辑电平设为 1；该信号的逻辑电平从 1 变成 0 标志着一个起始信号，从 0 变成 1 标志着一个终止信号。      |
| SIO_C  | 输出  | 时钟信号。标志着一个位的传输，当总线处于空闲状态，主机应该将其逻辑电平设为 1；当 SCCB_E 信号的裸机电平为 0 时，该管脚的 0、1 变换标志着位的传输。 |
| SIO_D  | I/O | 数据信号。当总线为空闲状态时将其浮空状态，当系统为挂起状态是将其设为裸机 0。在数据传输阶段，该管脚的逻辑电平变换只能发生在 SIO_C 为低电平时。       |
| PWDN   | 输出  | 控制从设备状态(附加的信号)                                                                    |

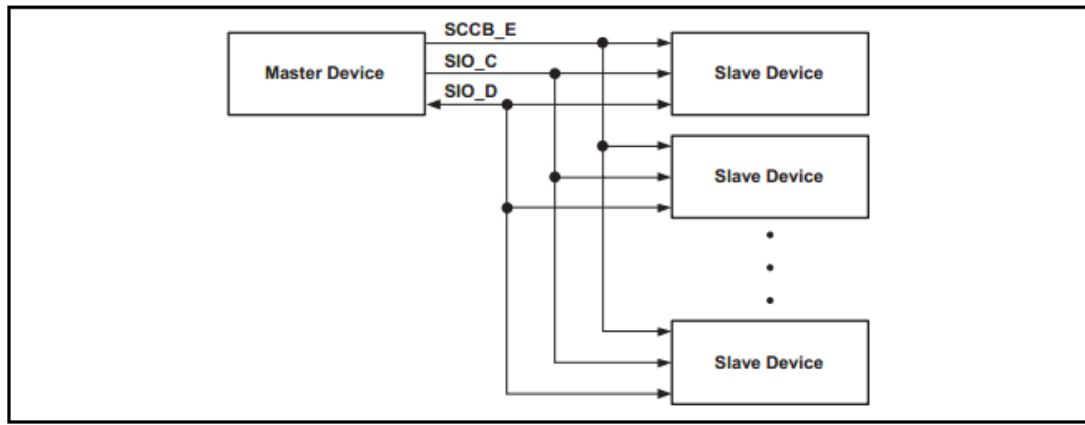


图 30-3 3 线 SCCB 设备连接范例

(8) 简化的 2 线 SCCB 接口定义见表 30-3。

表 30-3 2 线的 SCCB 接口定义

| 信号名称  | 信号类型 | 描述                                     |
|-------|------|----------------------------------------|
| SIO_C | 输出   | 时钟信号。空闲状态下该管脚的逻辑电平为 1。逻辑 0、1 变换标志着位的传输 |

|       |     |                                                                                                                               |
|-------|-----|-------------------------------------------------------------------------------------------------------------------------------|
| SIO_D | I/O | 数据信号。当 SIO_C 是高电平时 SIO_D 线由高电平向低电平切换表示起始传输；当 SIO_C 是高电平时 SIO_D 线由低电平向高电平切换表示停止终止。因此在数据传输过程中，只有当 SIO_C 的电平为 0 时，该管脚逻辑电平才能发生改变。 |
|-------|-----|-------------------------------------------------------------------------------------------------------------------------------|

表格描述可以看出，两线 SCCB 接口同样可以满足控制要求，如果需要控制从设备的工作状态，我们可以外加 GPIO 来控制从设备即可。

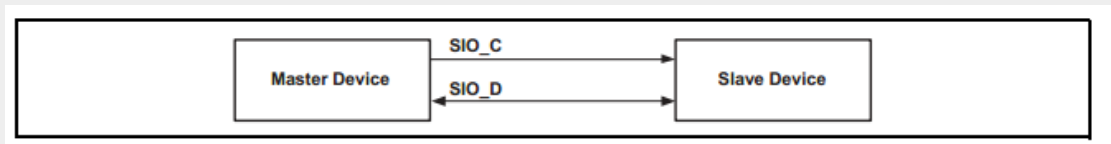


图 30-4 2 线 SCCB 设备连接范例

### 30.3.2. SCCB 时序描述

读者可以参考 I<sup>2</sup>C 协议与 SCCB 的时序进行对比，SCCB 时序见图 30-5  
SCCB 总线时序

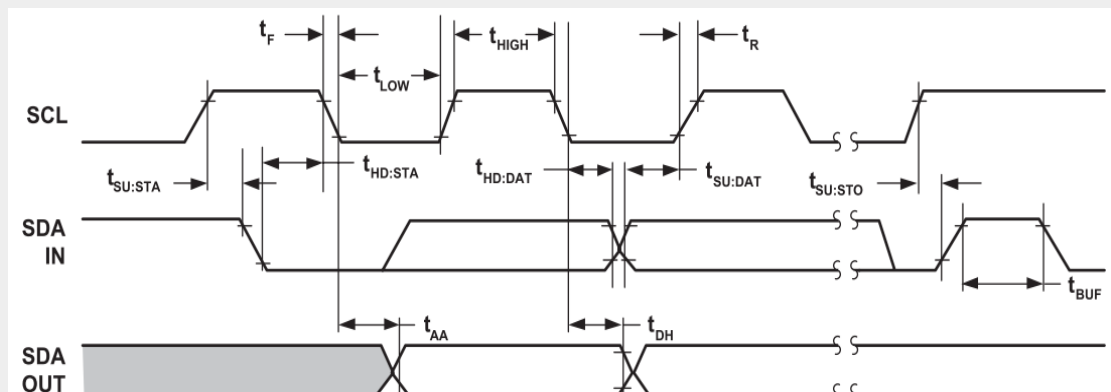


图 30-5 SCCB 总线时序

图中所引用的一些符号描述见表 30-4。

表 30-4 时序图描述

| 符号 | 描述 |
|----|----|
|----|----|

|  |                     |
|--|---------------------|
|  | 其实条件的建立时间           |
|  | SCCB 的上升或者下降时间      |
|  | 新的“START”信号前总线的空闲时间 |
|  | “START”条件的保持时间      |
|  | “START”条件的创建时间      |
|  | Data-in 保持时间        |
|  | Data-in 建立时间        |
|  | Data-out 保持时间       |

(9) 起始信号：在 SIO\_C 为高电平时，SIO\_D 出现一个下降沿，则 SCCB 开始传输。

(10) 停止信号：在 SIO\_C 为高电平时，SIO\_D 出现一个上升沿，则 SCCB 停止传输。

(11) 数据传输：除了开始和停止状态，当数据传输时，当 SIO\_C 为高电平时，必须保证 SIO\_D 上的数据稳定，也就是说，SIO\_D 上的电平变换只能发生在 SIO\_C 为低电平的时候。

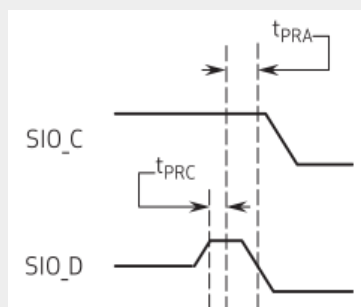


图 30-6 SCCB 起始信号

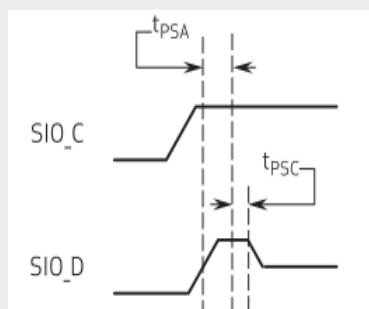


图 30-7 SCCB 停止信号

与 I<sup>2</sup>C 总线类似，SCCB 基本传输格式如图 30-8 所示，完整的数据传输包括两个或者三个阶段。每一阶段包中含 9 位二进制数据，其中高 8 位为所要传输的 8 位数据，第一阶段的 bit 0 根据主器件的数据传输是读操作还是写操作而确定。在进行主器件写操作时，全部阶段的最低位均是自由位(Don't care bit)，自由位中主机输出高电平，从机输出低电平响应(有些从机不会响应自由位)；读操作时，第一阶段的最低位是自由位，第二阶段的最低位 NA——主器件驱动为高电平有效。

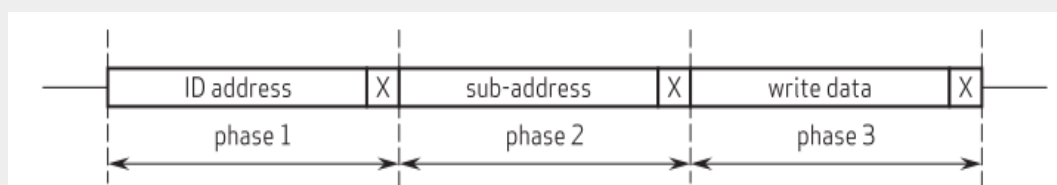


图 30-8 SCCB 的三相写操作

在 SCCB 协议定义了两种写操作，即三相写操作和两相写操作。三相写操作是向从器件的一个目的寄存器中写入数据，见图 30-8。在三相写操作中，第一阶段写从器件的 8 位 IDW(器件地址+写方向标志)和自由位，第二阶段写从器件目标寄存器的 8 位地址和自由位，第三阶段写要求写入寄存器的 8 位数据和自由位；两相写操作没有第三阶段，即只向从器件传输了器件 ID 和目的寄存器的地址，见图 30-9。

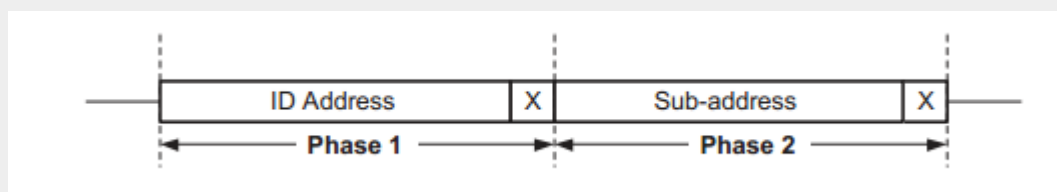


图 30-9 SCCB 的两相写操作

SCCB 协议定义了两相读操作，它用于读取从器件目的寄存器中的数据，见图 30-10。在第一阶段中写从件读操作 8 位 IDR(器件地址+读方向标志)和自由位，在第二阶段中读取寄存器中的 8 位数据和写 NA bit。由于两相读操作没有确定目的寄存器的地址，所以在读操作前，必需有一个两相或三相的写循环操作，以提供读操作中的寄存器地址(该过程类似 EEPROM 例程中的“伪写”)。

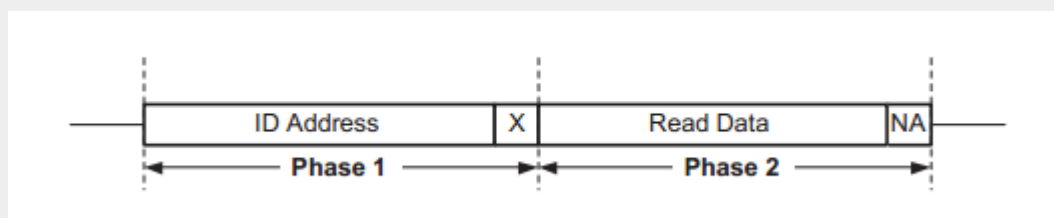


图 30-10 SCCB 的两相读操作

## 30.4. 摄像头模块

### 30.4.1. 摄像头模块硬件介绍。

野火 STM32 开发板具有一个 CAMERA 接口，可用于扩展摄像头模块，该接口可兼容野火的 OV7670、OV7725 及市面上的其它基于 OV 系列传感器的模块。考虑到一些性能比较低的单片机也能够搭载野火开发的摄像头模块，我们在 OV7670 后端搭载了 FIFO 来降低对单片机的性能依赖-----当前模块对处理器的硬件要求仅仅为一个中断，几个 GPIO 管脚即可。

### 30.4.2. ov7670 输出时序

摄像头采集的图像，也就是 OV7670 芯片的输出，它采用 VGA 时序，通过 VSYNC、HREF/HSYNC 和 PCLK 引脚输出图像。因此我们必须正确地理解好 VGA 工作时序图。

VGA 最早是指一种 480\*640 像素显示器的显示模式，它显示时，一行一行地对图像的像素进行扫描，所以它的时序分为行时序和帧时序。

#### 21. 行时序图

行时序即输出每一行像素的时序，见图 30-11。它以 PCLK 输出像素时钟，以 D0~D9 输出像素数据，以 HREF 输出行起始和行结束信号。

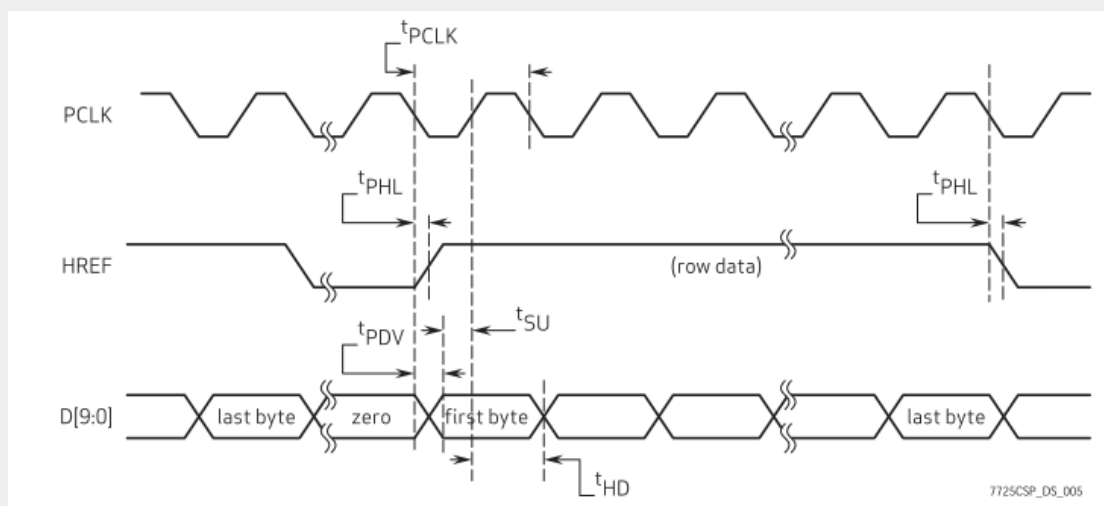


图 30-11 OV7670 输出的 VGA 行时序

从图中可以看出，当 **HREF** 为高电平时，摄像头数据端口随着像素时钟 **PCLK** 的运转，先后输出一行的像素数据，当一行数据传输完成时，**HREF** 转为低电平。在 **HREF** 为高电平期间，每一个 **PCLK** 时钟就输出一个基本数据单元，而且数据在 **PCLK** 在上升沿阶段维持稳定，因此，主控芯片配置中断时，应该配置为上升沿中断，在上升沿读取数据。

## 22. 帧时序图

**VGA** 显示图像时以一幅图像为一帧，所以其帧时序(也称为场时序)由多个行时序组成，见图 30-12。帧时序与行时序的区别主要是多了一条信号线 **VSYNC** 用于表示帧起始信号和帧结束信号。见图 30-12。

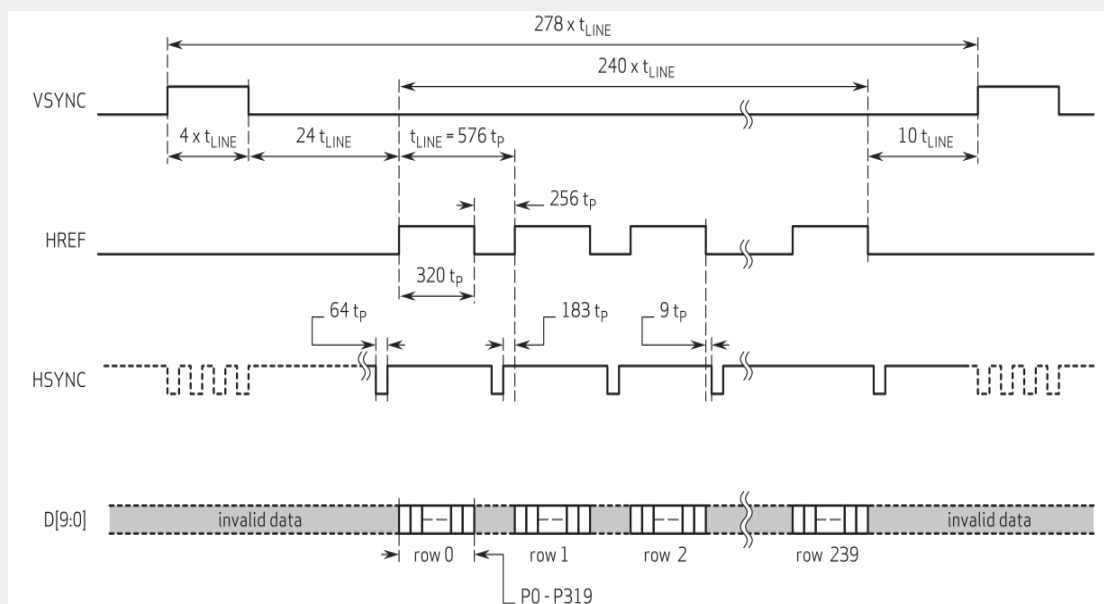


图 30-12 QVGA 帧时序

当 VSYNC 为低电平时，表明正在传送一幅图像，当 VSYNC 由低电平变为高电平时，表明一幅图像的数据已经传送完成。而在 VSYNC 为低电平期间，我们可以用 HREF 信号来换行，当 HREF 由高电平变为低电平时，表明一行数据已经传送完毕，当 HREF 由低电平变为高电平时，表明新的一行数据开始传输。信号 HSYNC 同样可用于行同步，它与 HREF 有一点区别(HSYNC 和 HREF 共用一个管脚，可以通过软件来配置)，由图可知，HREF 为高电平时，摄像头输出的数据都是有效的，而 HSYNC 高电平期间，在后面有一段时间是无意义的，即图中灰色部分。结合 FIFO 的特性，我们选用了 HREF 作为行同步。

OV7670 实际输出图像时，不是标准的 VGA 时序，而可以选择使用 QVGA、CIF、QCIF 时序(可通过配置 OV7670 的 COM7 寄存器选择)。QVGA 意为 Quarter VGA，译为 VGA 的四分之一，即 240\*320 像素，这正好符合野火 3.2 寸屏的像素大小，所以我们采用 QVGA 时序。图 30-12 中的 QVGA 时序，可以看到输出每行像素的时间(HREF 高电平的时间)为 320 T<sub>p</sub>，即 320 个像素点的时间，240\* T<sub>line</sub>就是指输出 240 行像素的基本时间。

### 23. 像素输出时序

像素输出时序即每个像素点数据的组织形式和输出方式。我们选择的是 RGB565 格式，所以，我们看看该模式下的像素输出时序，见图 30-13。

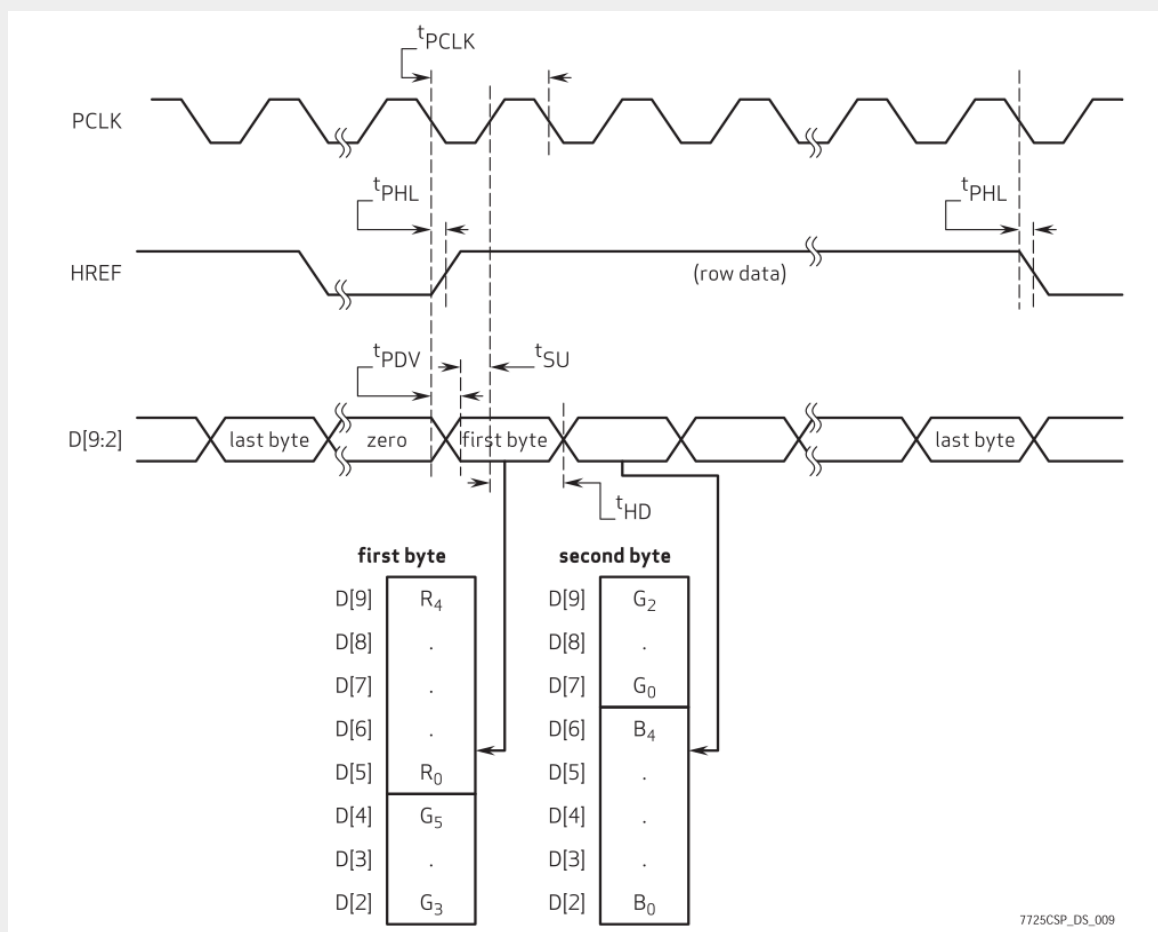


图 30-13 摄像头 RGB565 像素数据输出时序图 30-13

像素值从摄像头数据端口的 D2 到 D9 管脚输出，每一个像素由 RGB565 来表示，共 16 位，分两次来传输，之后用户组合起来即可。当用户收到 HREF 上升沿中断之后，之后开始捕捉到 PCLK 上升沿是有效的，该时刻读取到的数据即为第一字节有效像素数据。当收到 HREF 下降沿时，表明整行像素已经全部输出。

### 30.4.3. FIFO 时序

摄像头采集分辨率我们配置为 240\*320，每个像素用 RGB565 表示，即每个像素占用两个字节，因此，整幅图像占用的空间大小为 240\*320\*2，等于 153600 字节。摄像头模块中采用的 FIFO 型号为 AL422B，其容量高达 384K 字节，完全符合我们的空间要求。

AL422B 以下几个操作比较重要：读 FIFO，写 FIFO，读指针复位，写指针复位等。



## 24. 读 FIFO 时序

读 FIFO 时序见图 30-14。

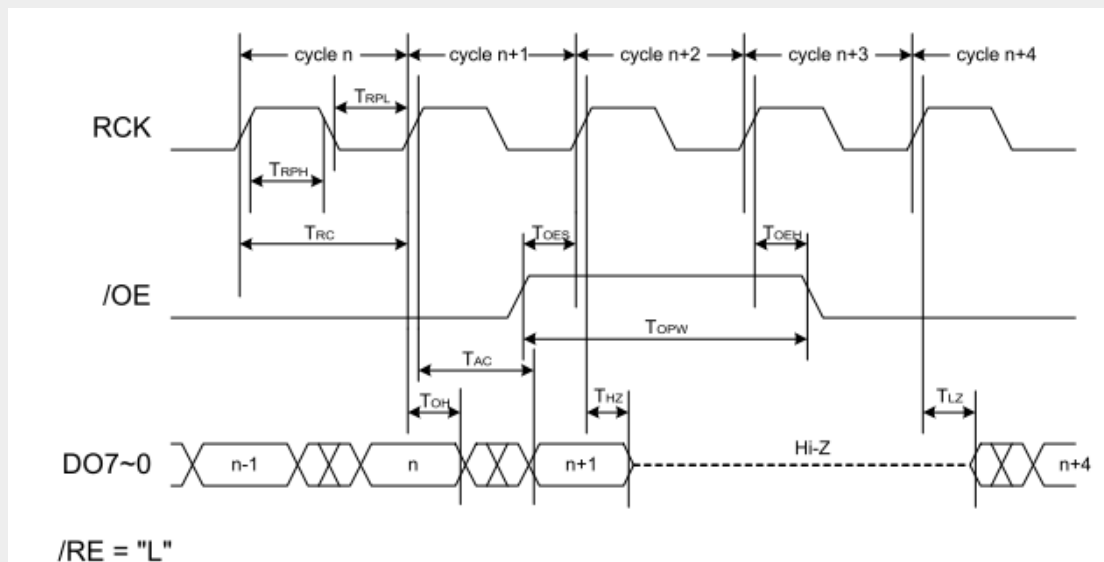


图 30-14 读 FIFO 时序

摄像头模块的 FIFO 硬件中已经将 RE 管脚设置为低电平，即硬件已经激活读操作。从时序图可知，当 OE 管脚为低电平时，输出处于使能状态，随着读时钟 RCK 的运转，在数据输出管脚 DO7-DO0 就会按地址递增的方式输出数据；当 OE 管脚为高电平时，关闭输出，随着读时钟 RCK 的运转，在数据输出管脚 DO7-DO0 会呈现高阻态，即在这个时候，单片机读取到的端口值是不稳定的、无效的，但是 FIFO 读指针的地址值仍会随着 RCK 的运转递增。

## 25. 写 FIFO 时序

写 FIFO 时序见图 30-15。

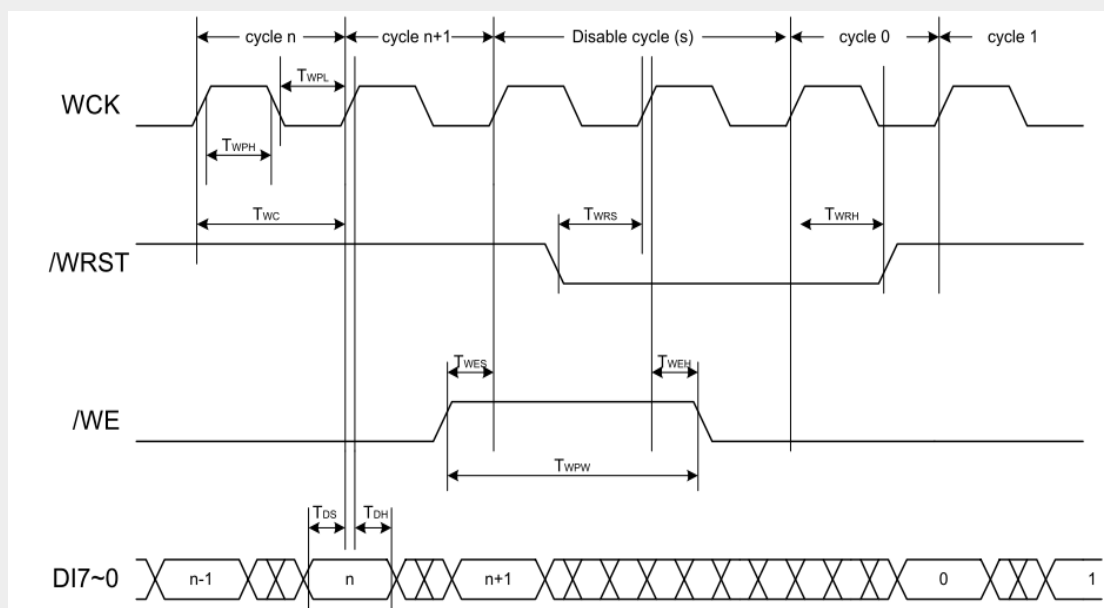


图 30-15 写 FIFO 时序

在写时序中，当 WE 管脚为低电平时，FIFO 写入处于使能状态，随着读时钟 WCK 的运转，数据管脚 DI7-DI0 的电平值将会按地址递增的方式存入 FIFO；当 WE 管脚为高电平时，关闭输入，数据端口管脚 DI7-DI0 的电平值不会随着写时钟 WCK 的运转被捕捉保存入 FIFO。

26. 读指针复位操作：当 RRST 为低电平时，读指针就复位到从 0 地址开始读取数据。

27. 写指针复位操作：当 WRST 为低电平时，写指针就复位到从 0 地址开始写入数据。

#### 30.4.4. 摄像头的驱动原理

摄像头模块的 OV7670 和 FIFO AL422B 的原理图见图 30-16 和图 30-17。

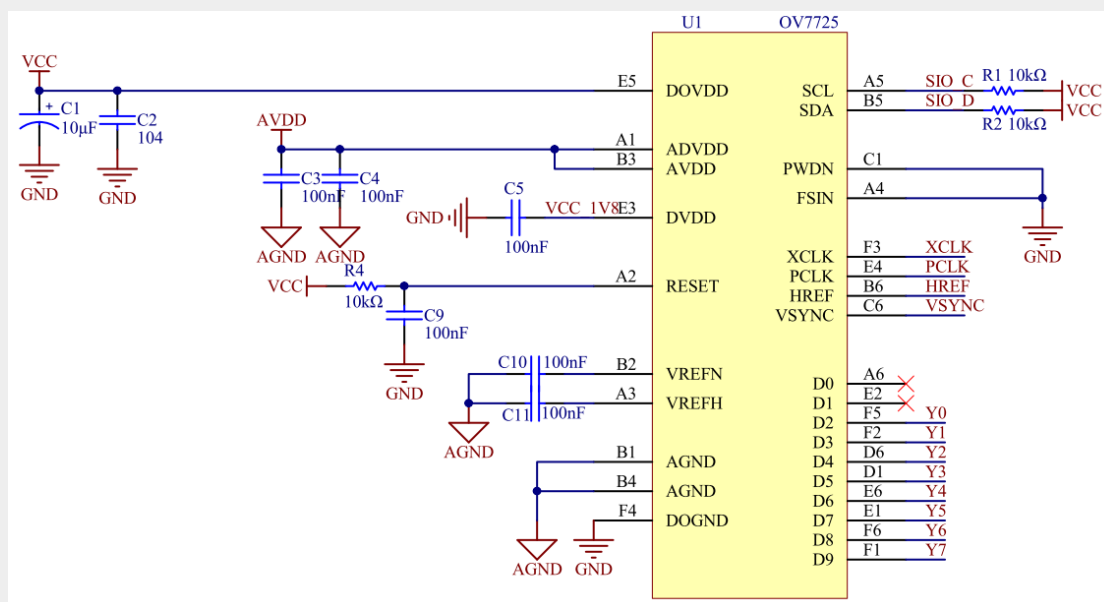


图 30-16 OV7670 硬件驱动电路图

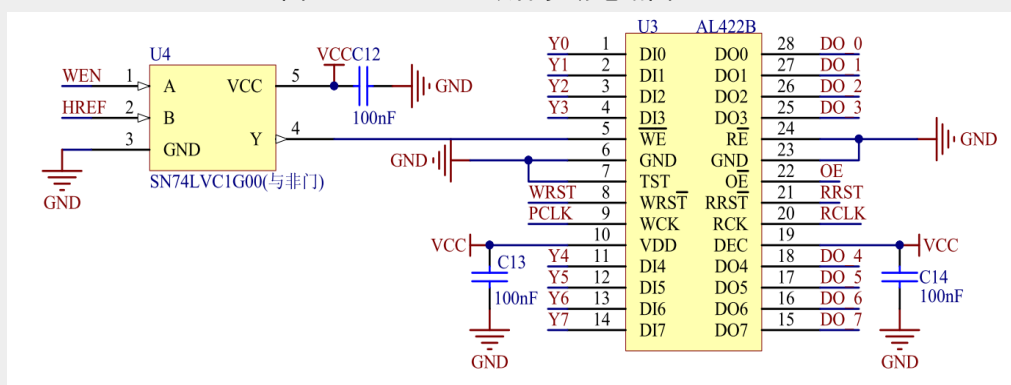


图 30-17 OV7670 与 AL422B 硬件连接图

从原理图可知，WEN 与 HREF 作为与非门的输入端，而与非门的输出端口连接到 FIFO 的 WE 端口，当 WEN 管脚跟 HREF 管脚电平都为高电平时，FIFO 的 WE 端口的电平值才为低电平。因此，如果我们允许图像数据传入 FIFO 时，我们可以把 WEN 拉高，当摄像头的 HREF 为高电平时，WE 为低电平，此时随着 PCLK 的运转，摄像头的像素数据会被送入 FIFO，当 HREF 为低电平时，WE 的电平值为高电平，因此，即使 PCLK 仍在运转，也不会传送数据进入 FIFO，从而防止采集到无效数据。

程序驱动部分思路流程大概如下：将摄像头的场中断信号送入单片机，当单片机捕捉到场中断信号时，拉高 WEN 管脚电平，之后整幅图像的数据就会存入 FIFO。当单片机再次捕捉到场中断时，表明一幅图像已经送入 FIFO，此时，关闭场中断，拉低 WEN 电平，防止摄像头数据再次写入 FIFO，并开始读

取 FIFO 数据到内存缓冲区，根据采集到的图像做我们的应用处理。应用完成后，我们再次开启场中断，当再次捕捉到场中断时，再次拉高 WEN 管脚电平，继续让摄像头数据存入 FIFO，完成后单片机读取 FIFO 数据并应用，如此循环。

### 30.5. 摄像头驱动实验

#### 30.5.1. 实验描述及工程文件清单

##### 实验描述：

在本实验中，采用野火 OV7670 摄像头模块，连接到野火 STM32 开发板的 CAMERA 扩展接口中，实现实时采集图像，并显示到 3.2 寸的屏幕上

##### 硬件连接：

硬件连接中的液晶屏幕部分请参考 LCD 章节。

摄像头硬件连接：

|                     |                    |
|---------------------|--------------------|
| PD6---CAMERA-OE     | PB8---CAMERA-D0    |
| PE0---CAMERA-RRST   | PB9---CAMERA-D1    |
| PE2---CAMERA-RCLK   | PB10---CAMERA-D2   |
| PA8---CAMERA-XCLK   | PB11---CAMERA-D3   |
| PA0---CAMERA- VSYNC | PB12---CAMERA-D4   |
| PB5---CAMERA- WRST  | PB13---CAMERA-D5   |
| PD3---CAMERA-WEN    | PB14---CAMERA-D6   |
| PA1---CAMERA-HREF   | PB15---CAMERA-D7   |
| PC6---CAMERA-SIO-C  | PC6---CAMERA-SIO-D |

##### 库文件：

使用 3.5 版本固件库

Startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/misc.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_exti.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_fsmc.c

FWlib/stm32f10x\_usart.c

用户文件:

USER/main.c

USER/stm32f10x\_it.c

USER/lcd.c

USER/SysTick.c

USER/lcd\_botton.c

USER/usart1.c

SENSOR/OV7670.c

SENSOR/SCCB.c

SENSOR/SensorConfig.c

SENSOR/Imag\_App.c

野火开发板上的 CAMERA 接口兼容 OV7670、OV7725 等系列的摄像头模块，原理图见图 30-18。

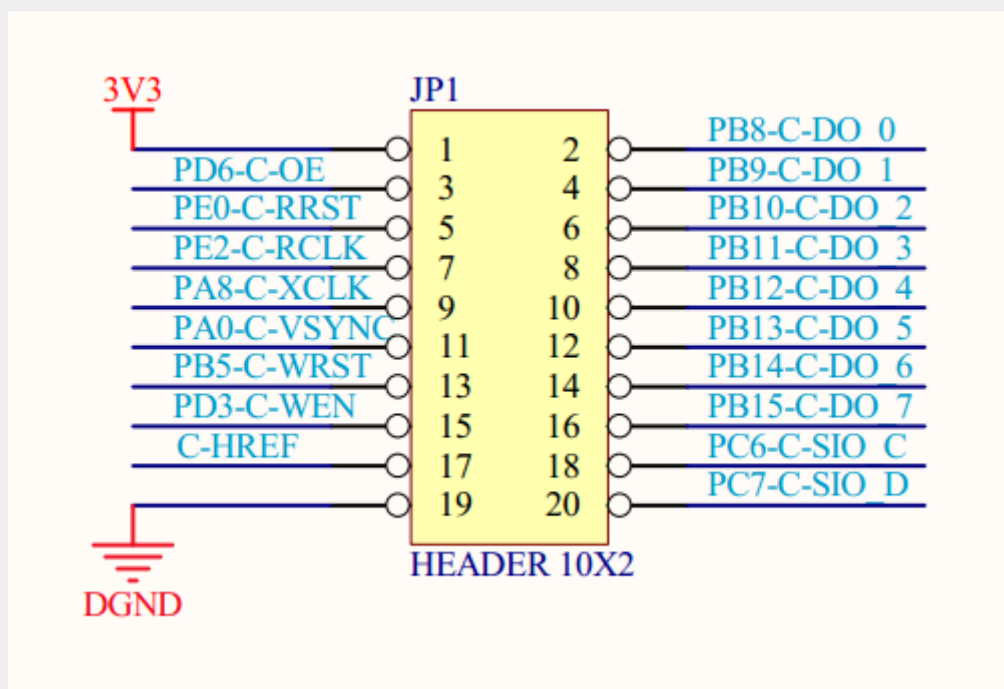


图 30-18 野火 CAMERA 接口原理图

### 30.5.2. 配置工程环境

本摄像头驱动实验中我们用到了 GPIO、RCC、EXTI、FSMC、USART 外设，所以我们要把以下库文件添加到工程：stm32f10x\_gpio.c、stm32f10x\_rcc.c、stm32f10x\_exti.c、stm32f10x\_fsmc.c、stm32f10x\_usart.c。由于在场中断 VSYNC 的检测中使用了中断，所以还要把 misc.c 文件添加进工程。

本工程使用了旧的用户文件 SysTick.c，用作定时，还使用了 lcd\_botton.c 文件驱动 LCD。最后在 stm32f10x\_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 30-1。

代码清单 30-1：摄像头例程的 stm32f10x\_conf.h 文件配置

```

16. /**
17.  *****
18.  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
19.  * @author  MCD Application Team
20.  * @version V3.5.0
21.  * @date    08-April-2011
22.  * @brief   Library configuration file.
23.  *****

```

```
24.
25. #include "stm32f10x_exti.h"
26. #include "stm32f10x_fsmc.h"
27. #include "stm32f10x_gpio.h"
28. #include "stm32f10x_rcc.h"
29. #include "stm32f10x_usart.h"
30. #include "misc.h"
```

### 30.5.3. main 文件

从本工程的 main 文件分析代码的执行流程，见代码清单 30-2。

代码清单 30-2：摄像头例程的 main 函数

```
55.
56. #include "stm32f10x.h"
57. #include "lcd_botton.h"
58. #include "OV7670.h"
59. #include "Imag_App.h"
60. #include "SysTick.h"
61. #include "usart1.h"
62. #include "SCCB.h"
63. extern volatile u8 Frame_Count;
64.
65.
66. /*使用 3.2 寸 ILI9341 LCD*/
67. int main(void)
68. {
69.
70.     SysTick_Init();
71.     USART1_Config();
72.     LCD_Init();
73.     SCCB_GPIO_Configuration();
74.     FIFO_GPIO_Configuration();
75.     while(OV7670_Init() != SUCCESS);
76.     VSYNC_Init();
77.     LCD_open_windows(0, 0, 320, 240);
78.     OV7670_VSYNC = 0;
79.     while(1)
80.     {
81.         if( 1 )
82.         {
83.             Delay_us(5);
84.             FIFO_PREPARE;           /*FIFO 准备*/
85.             Get_imag_and_discor(); /*采集并显示*/
86.             OV7670_VSYNC = 0;
87.             Frame_Count++;          /*帧计数器加 1*/
88.         }
89.     }
90. }
91.
92. /***** (C) COPYRIGHT 2012 WildFire Team ****END OF FILE****
93. */
```

main 函数的执行流程说明如下：

- main 函数首先调用了 SysTick()、USART1Config()、LCD\_Init()函数初始化了 SysTick、串口及 LCD，这些函数在前面的章节已经介绍。
- 接下来 SCCB\_GPIO\_Configuration() 和 FIFO\_GPIO\_Configuration()函数用于初始化 GPIO，这些 GPIO 分别用来模拟 SCCB 时序和读写 FIFO。
- 初始化好 SCCB 相关的引脚，就可向 OV7670 芯片写入配置参数，本例程中是通过调用 OV7670\_Init()函数实现的，该函数利用 SCCB 通讯协议，根据 OV7670 的寄存器说明，向寄存器写入相关的配置参数。
- 由于需要 VSYNC 提供中断，所以作为 VSYNC 引脚的 GPIO PA0 要设置 EXTI 中断，并配置中断优先级，这些工作由 VSYNC\_Init()函数完成。
- 调用 LCD\_open\_windows()使 LCD 开窗，做好显示的准备。
- 根据 OV7670\_VSYNC 标志，判断 FIFO 是否接收完了一幅图像。在中断服务程序中，若检测到两次 VSYNC 的上升沿(表示接收完一幅图像)，会把 OV7670\_VSYNC 变量设置为 2。
- 判断接收完成一幅图像后，调用宏 FIFO\_PREPARE 使读 FIFO 指针复位，使 Get\_imag\_and\_discor()读取 FIFO 时，读取得正确的数据并显示到 LCD 屏。
- 记录帧数目的变量 Frame\_Count 加 1，这个变量用来统计帧率，由 SysTick 的中断服务函数通过串口输出。最后把 OV7670\_VSYNC 置 0，使重新开始计数。

#### 30.5.4. SCCB 总线的软件实现

由于 STM32 没有 SCCB 的接口，我们需要用普通的 GPIO 引脚来模拟 SCCB 时序。首先需要用 SCCB\_GPIO\_Configuration()函数初始化 SCCB 总线所用到的 GPIO 引脚。再根据 SCCB 时序，编写读、写字节的函数，OV7670\_Init()



利用这些函数，向 OV7670 相应的寄存器写入配置参数，初始化摄像头。本小节介绍的与 SCCB 时序相关函数都位于 SCCB.c 文件中。

### 3. 初始化 SCCB 用到的 GPIO

在本实验中，摄像头的 SCL 和 SDA 连接到 STM32 的 PC6 和 PC7，使用 SCCB\_GPIO\_Configuration()函数初始化这两个引脚，其代码见代码清单 30-3。

代码清单 30-3: SCCB\_GPIO\_Configuration()函数

```
1.  /*****
2.  * 函数名: SCCB_Configuration
3.  * 描述   : SCCB 管脚配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 注意   : 无
7.  *****/
8.  void SCCB_GPIO_Configuration(void)
9.  {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     /* SCL(PC6)、SDA(PC7)管脚配置 */
12.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
13.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
14.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
15.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
16.     GPIO_Init(GPIOC, &GPIO_InitStructure);
17. }
```

### 4. SCCB 起始与结束时序

SCCB 通讯需要有起始与结束信号，这分别由 SCCB\_Start()和 SCCB\_Stop()函数实现。

SCCB\_Start()代码见代码清单 30-4。

代码清单 30-4: SCCB\_Start()函数

```
1.  /*****
2.  * 函数名: SCCB_Start
3.  * 描述   : SCCB 起始信号
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 注意   : 内部调用
7.  *****/
8.  static int SCCB_Start(void)
9.  {
10.     SDA_H;
```

```
11.     SCL_H;  
12.     SCCB_delay();  
13.     if(!SDA_read)  
14.         return DISABLE; /* SDA 线为低电平则总线忙,退出 */  
15.     SDA_L;  
16.     SCCB_delay();  
17.     if(SDA_read)  
18.         return DISABLE; /* SDA 线为高电平则总线出错,退出 */  
19.     SDA_L;  
20.     SCCB_delay();  
21.     return ENABLE;  
22. }
```

参照前面介绍的 SCCB 时序,当 SCL 线为高电平时,SDA 线出现下降沿,表示 SCCB 时序的起始信号,SCCB\_Start()函数就是实现了这样功能其中的 SDA\_H 和 SCL\_H 是用于控制 SDA 和 SCL 引脚电平的宏。为了提高程序的健壮性,还使用 SDA\_read()宏检测 SDA 线是否忙碌或是否正常。

SCCB 结束信号的函数实现类似,其 SCCB\_Stop()代码见代码清单 30-5。

代码清单 30-5: SCCB\_Stop()函数

```
e.  /*****  
*****  
f.  * 函数名: SCCB_Stop  
g.  * 描述   : SCCB 停止信号  
h.  * 输入   : 无  
i.  * 输出   : 无  
j.  * 注意   : 内部调用  
k.  *****/  
*****/  
l.  static void SCCB_Stop(void)  
m.  {  
n.      SCL_L;  
o.      SCCB_delay();  
p.      SDA_L;  
q.      SCCB_delay();  
r.      SCL_H;          /* SCL 线为高电平时,SDA 线状态变换表示停止信  
号 */  
s.      SCCB_delay();  
t.      SDA_H;  
u.      SCCB_delay();  
v.  }
```

参照前面的介绍,当 SCL 线为高电平的时候,使 SDA 线出现一个上升沿,表示 SCCB 时序的结束信号。

## 5. 写寄存器与读寄存器

与 I2C 时序类似,在 SCCB 时序也使用自由位(Don't care bit)和非应答(NA)信号来保证正常通讯。自由位和非应答信号位于 SCCB 每个传输阶段中的第九位。

在写数据的第一个传输阶段中，第 9 位为自由位，在一般的正常通讯中，第 9 位时，主机的 SDA 线输出高电平，而从机把 SDA 线拉低作为响应，第二、三阶段类似，只是传输的内容分别为目的寄存器地址和要写入的数据。见图 30-19。

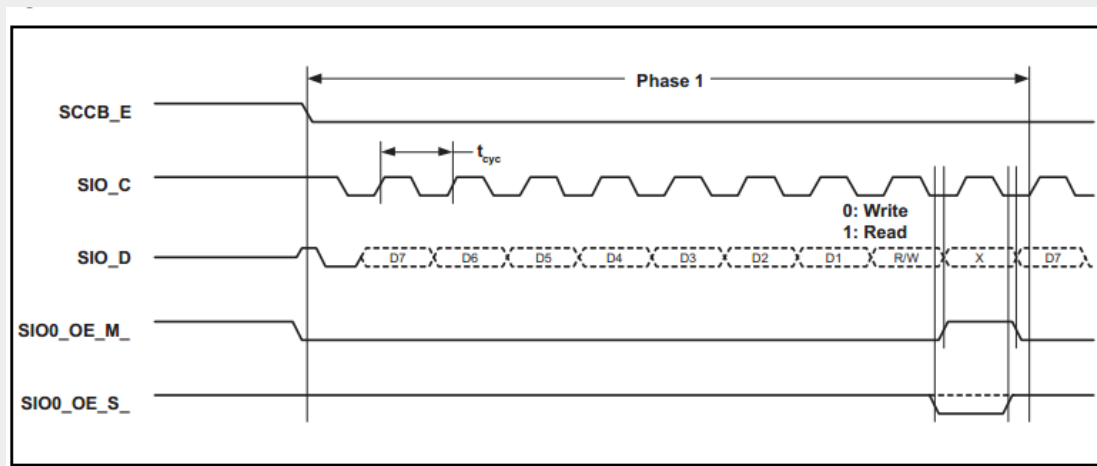


图 30-19 写操作第一阶段(传输器件地址)

因此，在数据传输的第 9 位，主机使用 SCCB\_WaitAck()函数来等待从机的应答，代码见代码清单 30-6。

代码清单 30-6: SCCB\_WaitAck()函数

```
68. /*****
*****
69. * 函数名: SCCB_WaitAck
70. * 描述 : SCCB 等待应答
71. * 输入 : 无
72. * 输出 : 返回为:=1 有 ACK,=0 无 ACK
73. * 注意 : 内部调用
74. *****/
*****/
75. static int SCCB_WaitAck(void)
76. {
77.     SCL_L;
78.     SCCB_delay();
79.     SDA_H;
80.     SCCB_delay();
81.     SCL_H;
82.     SCCB_delay();
83.     if(SDA_read)
84.     {
85.         SCL_L;
86.         return DISABLE;
87.     }
88.     SCL_L; //SDA 低电平应答
89.     return ENABLE;
90. }
```

该函数让主机把 SDA 线设为高电平，延时一段时间后再检测 SDA 线的电平，若为低则返回 **ENABLE** 表示接收到从机的应答，反之返回 **DISABLE**。

最后，整个三相写过程由函数 **SCCB\_WriteByte()**实现，它的具体代码见代码清单 30-7。

代码清单 30-7: **SCCB\_WriteByte()**函数

```
22.  /*****
23.  * 函数名: SCCB_WriteByte
24.  * 描述  : 写一字节数据
25.  * 输入  : - WriteAddress: 待写入地址    - SendByte: 待写入数据    -
DeviceAddress: 器件类型
26.  * 输出  : 返回为:=1 成功写入,=0 失败
27.  * 注意  : 无
28.  *****/

29. int SCCB_WriteByte( u16 WriteAddress , u8 SendByte )
30. {
31.     if(!SCCB_Start())
32.     {
33.         return DISABLE;
34.     }
35.     SCCB_SendByte( DEV_ADR );          /* 器件地址，第一阶段*/
36.     if( !SCCB_WaitAck() )
37.     {
38.         SCCB_Stop();
39.         return DISABLE;
40.     }
41.     SCCB_SendByte((u8)(WriteAddress & 0x00FF));
42.                                     /* 设置低起始地址，第二阶
段 */
43.     SCCB_WaitAck();
44.     SCCB_SendByte(SendByte);          /* 写内容，第三阶段 */
45.     SCCB_WaitAck();
46.     SCCB_Stop();
47.     return ENABLE;
48. }
```

**SCCB\_WriteByte()** 函数调用 **SCCB\_Start()** 产生起始信号，接着调用 **SCCB\_SendByte()** 把器件地址 **DEV\_ADR** (这是一个宏，数值是 **0x42**) 一位一位地发送出去，在第 9 位时，调用 **SCCB\_WaitAck()** 函数检测从机的应答，若接收到应答则进入第二阶段——发送目的寄存器地址，再进入第三阶段——发送要写入的内容。在第二、三阶段没有加条件判断语句判断是否接收到从机的应答是因为 **SCCB** 规定在数据传输阶段允许从机不应答(实际上,OV7670 芯片在这两个阶段都会有应答讯号)。在最后，三阶段都传输结束时，要调用 **SCCB\_Stop()** 函数结束本次 **SCCB** 传输。

与自由位相对应的非应答信号用在两相读操作的第二阶段的第 9 位，见图 30-20。在这第 9 位中，从机把 SDA 线置为高电平，而主机把 SDA 线拉低表示非应答，接着本次读数据的操作就结束了(SCCB 不支持连续读取数据)。

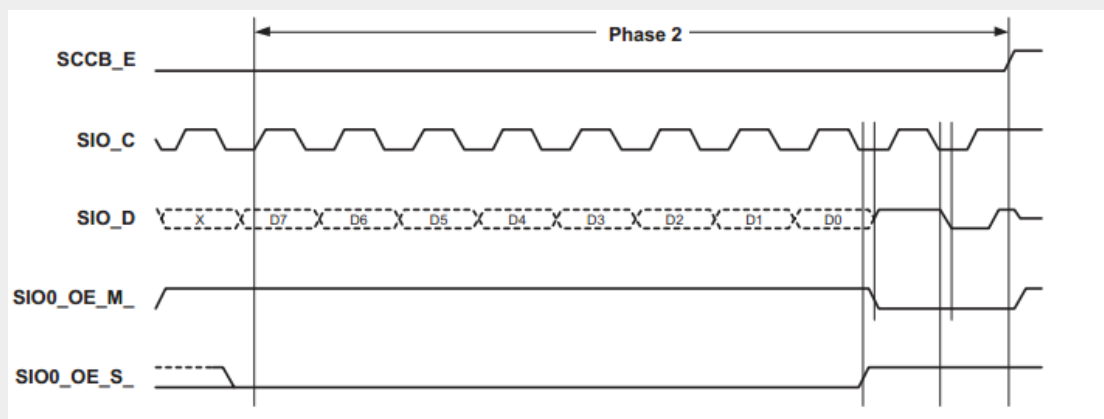


图 30-20 两相读操作第二阶段(读寄存器内容)

主机的非应答信号，由 SCCB\_NoAck()函数实现，其代码见代码清单 30-8。

#### 代码清单 30-8: SCCB\_NoAck()函数

```
8.  /*****
9.  * 函数名: SCCB_NoAck
10. * 描述 : SCCB 无应答方式
11. * 输入 : 无
12. * 输出 : 无
13. * 注意 : 内部调用
14. *****/
15. static void SCCB_NoAck(void)
16. {
17.     SCL_L;
18.     SCCB_delay();
19.     SDA_H; //高电平为非应答信号
20.     SCCB_delay();
21.     SCL_H;
22.     SCCB_delay();
23.     SCL_L;
24.     SCCB_delay();
25. }
```

最后，整个读寄存器的过程由 SCCB\_ReadByte()函数完成，其代码见代码清单 30-9。

#### 代码清单 30-9: SCCB\_ReadByte()函数

```
23. /*****
```

```

24. * 函数名: SCCB_ReadByte
25. * 描述 : 读取一串数据
26. * 输入 : - pBuffer: 存放读出数据 - ReadAddress: 待读出地址 -
DeviceAddress: 器件类型
27. * 输出 : 返回为:=1 成功读入,=0 失败
28. * 注意 : 无
29. *****/

30. int SCCB_ReadByte(u8* pBuffer, u8 ReadAddress)
31. {
32.
33.     if(!SCCB_Start())
34.     {
35.         return DISABLE;
36.     } /*在读操作前要有写操作以提供起
始地址*/
37.
38.     SCCB_SendByte( DEV_ADR ); /* 器件地址, 写操作第一阶
段 */
39.     if( !SCCB_WaitAck() )
40.     {
41.         SCCB_Stop();
42.         return DISABLE;
43.     }
44.     SCCB_SendByte( ReadAddress ); /* 设置低起始地址, 写操作第二阶
段 */
45.
46.     SCCB_WaitAck();
47.     SCCB_Stop();
48.
49.     if(!SCCB_Start())
50.     {
51.         return DISABLE;
52.     }
53.     SCCB_SendByte( DEV_ADR + 1 ); /* 器件地址, 读操作第一阶
段 */
54.     if(!SCCB_WaitAck())
55.     {
56.         SCCB_Stop();
57.         return DISABLE;
58.     }
59.
60.     *pBuffer = SCCB_ReceiveByte(); /* 读取内容, 读操作第二阶
段 */
61.     SCCB_NoAck();
62.
63.     SCCB_Stop();
64.     return ENABLE;
65. }

```

本函数在两相读操作前, 加入了一个两相写操作, 用于向从机发送要读取的寄存器地址。在两相写操作的第一个阶段(第 31 行), 使用 `SCCB_SendByte()` 函数发送的数据是 `DEV_ADR+1` (即 `0x43`), 这与写操作中发送的 `DEV_ADR` 有区别, 这是因为在第一阶段发送的这个器件地址的最低位是用于表示数据传送方向的, 最低位为 0 表示主机写数据, 最低位为 1 表示主机读数据, 所以 `DEV_ADR+1` 就表示读数据了。读操作的第二阶段是使用 `SCCB_ReceiveByte()`

函数，一位一位地接收数据，然后存放到 PBuffer 指向的单元中的。接收完 8 位数据后，主机调用 SCCB\_NoAck()发送非应答位，最后调用 SCCB\_Stop()结束本次读操作。

### 30.5.5. 初始化 ov7670

在上一小节编写了模拟 SCCB 的读写寄存器的时序后，就可以向 OV7670 的寄存器发送配置参数，对 OV7670 进行初始化了。该过程由 OV7670.c 文件中的 Sensor\_Init()函数完成，代码见代码清单 30-10。

代码清单 30-10: Sensor\_Init()函数

```
46.
47.  /*****
48.   * 函数名: Sensor_Init
49.   * 描述  : Sensor 初始化
50.   * 输入  : 无
51.   * 输出  : 返回 1 成功, 返回 0 失败
52.   * 注意  : 无
53.   *****/
54. ErrorStatus OV7670_Init(void)
55. {
56.     u16 i = 0;
57.     u8 Sensor_IDCode = 0;
58.
59.     if( 0 == SCCB_WriteByte ( 0x12, 0x80 ) ) /*复位 sensor */
60.     {
61.
62.         Delay_ms(50);
63.         return ERROR ;
64.     }
65.     Delay_ms(50);
66.     if( 0 == SCCB_ReadByte( &Sensor_IDCode, 0x0b ) )
67.         /* 读取 sensor ID 号*/
68.     {
69.         return ERROR;
70.     }
71.
72.
73.     if(Sensor_IDCode == OV7670_ID)
74.     {
75.         for( i = 0 ; i < OV7670_REG_NUM ; i++ )
76.         {
77.             if( 0 == SCCB_WriteByte(Sensor_Config[i].Address, S
78. ensor_Config[i].Value) )
79.             {
80.                 return ERROR;
81.             }
82.         }
83.     }
84.     else
85.     {
```

```
85.         return ERROR;
86.     }
87.         return SUCCESS;
88.     }
89.
```

这个函数的执行流程如下：

8) 调用 SCCB\_WriteByte()向地址为 0x12 的寄存器写入数据 0x80，进行复位操作。见表 30-5，根据 OV7670 的数据手册，把该寄存器的位 7 置 1，可控制 SCCB 的寄存器复位。

表 30-5 OV7670 部分寄存器

| 地址 | 寄存   | 默  | 读      | 描述                                                                                                                                                           |
|----|------|----|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | CLKR | 80 | 读<br>写 | 内部时钟 位[7]: 保留 位[6]: 直接使用外部时钟(没有预分频) 位[5: 0]: 内部时钟分频<br>$F(\text{内部时钟}) = F(\text{输入时钟}) / (\text{位}[5: 0] + 1)$                                              |
| 2  | COM7 | 00 | 读<br>写 | 通用控制 7 位[7]: SCCB 寄存器复位<br>0: 不复位<br>1: 复位<br>位[6]: 保留<br>位[5]: 输出格式-CIF<br>位[4]: 输出格式-QVGA<br>位[3]: 输出格式-QCIF<br>位[2]: 输出格式-RGB(见下面)<br>位[1]: 彩色条<br>0: 非使能 |
| 3  | COM8 | 8F | 读<br>写 | 通用控制 8 位[7]: 使能快速 AGC/AEC 算法 位[6]: AEC-步长限制<br>0: 步长限制与垂直同步<br>1: 不限制步长<br>位[5]: 条纹滤波器打开/关闭-打开条纹滤波器 ,BD50ST(0x9D) 或者 BD60ST(0x9E)要设成 1<br>0: 关               |



9) 调用 `SCCB_ReadByte()` 函数从地址为 `0x0b` 的寄存器读取出 `OV7670` 芯片的 ID 号，并与默认值进行对比，这个操作可以用来确保 `OV7670` 是否正常工作。

10) 利用 `for` 语句循环调用 `SCCB_WriteByte()` 函数，向各个寄存器写入配置参数，其中 `SCCB_WriteByte()` 的输入参数为 `Sensor_Config[i].Address` 和 `Sensor_Config[i].Value`，这是自定义的结构体数组，分别对应于要配置的寄存器地址和寄存器配置参数。`Sensor_Config` 数组是在 `SensorConfig.c` 文件定义的，首先定义了 `Register_Info` 结构体，它包含地址和寄存器两个结构体成员，见代码清单 30-11。

代码清单 30-11: `Register_Info` 结构体

```
24. typedef struct Register
25. {
26.     u8 Address;           /*寄存器地址*/
27.     u8 Value;            /*寄存器值*/
28. } Register_Info;
```

再利用这个自定义的结构体，定义结构体数组，每组的内容就表示了寄存器地址及相应的配置参数。若要修改对 `OV7670` 的配置，可参考 `OV7670` 数据手册的说明，修改相应地址的内容即可，`SCCB_WriteByte()` 函数会把这些参数写入 `OV7670` 芯片，下面是结构体数组的部分代码，省略了部分寄存器，完整部分请参考源代码，见代码清单 30-12。

代码清单 30-12: 部分寄存器控制参数

```
64.
65. Register_Info Sensor_Config[] =
66. {
67.
68.     /*以下为 OV7670 QVGA RGB565 参数 */
69.     {0x3a, 0x04}, //
70.     {0x40, 0x10},
71.     {0x12, 0x14},
72.     {0x32, 0x80},
73.     {0x17, 0x16},
74.
75.     {0x18, 0x04}, //5
76.     {0x19, 0x02},
77.     /******此处省略 N 行***** */
78.     {0x6f, 0x9f}, //0x9e for advance AWB
79.     {0x55, 0x00}, //亮度
80.     {0x56, 0x45}, //对比度
81.     {0x57, 0x80}, //0x40, change according to Jim's request
```

```
82.  
83.     };
```

### 30.5.6. 采集并显示图像

初始初始化 OV7670 完成后, OV7670 芯片开始正常工作, 由于我们使用的摄像头模块自带 FIFO, OV7670 采集得的图像保存到 FIFO, 我们使用 STM32 只需要检测摄像头模块的 VSYNC 输出的帧结束信号, 然后从 FIFO 中读取图像数据即可。

#### 49. 初始化 VSYNC 引脚

由于我们使用中断的方式来检测 VSYNC 的信号, 所以先要把相应的引脚初始化并为它配置 EXTI 中断。

在 main 函数中调用的 VSYNC\_Init()函数通过调用其它三个子函数来完成所有初始化流程, 代码见代码清单 30-13。

代码清单 30-13: VSYNC\_Init()函数

```
1.  /******  
2.  * 函数名: VSYNC_Init  
3.  * 描述   : OV7670 VSYNC 中断相关配置  
4.  * 输入   : 无  
5.  * 输出   : 无  
6.  * 注意   : 无  
7.  *****/  
8.  void VSYNC_Init(void)  
9.  {  
10.     VSYNC_GPIO_Configuration();  
11.     VSYNC_EXTI_Configuration();  
12.     VSYNC_NVIC_Configuration();  
13. }
```

子函数 VSYNC\_GPIO\_Configuration()完成了 GPIO 引脚模式、开启时钟的基本 I/O 配置工作, 本实验中使用的是 PA0 引脚检测 VSYNC 信号。VSYNC\_GPIO\_Configuration()代码见代码清单 30-14。

代码清单 30-14: VSYNC\_GPIO\_Configuration()函数

```
11. /******  
12. * 函数名: VSYNC_GPIO_Configuration  
13. * 描述   : OV7670 GPIO 配置  
14. * 输入   : 无  
15. * 输出   : 无  
16. * 注意   : 无  
17. *****/
```

```

18. static void VSYNC_GPIO_Configuration(void)
19. {
20.     GPIO_InitTypeDef GPIO_InitStructure;
21.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); /*PA0
-VSYNC*/
22.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
23.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
24.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
25.     GPIO_Init(GPIOA, &GPIO_InitStructure);
26. }

```

子函数 `VSYNC_EXTI_Configuration()` 为 PA0 引脚配置成上升沿触发的 EXTI 中断，根据前面关于 OV7670 输出时序的介绍，在 VSYNC 的两个上升沿期间，OV7670 输出了一幅完整的图像。`VSYNC_EXTI_Configuration()` 代码见。

代码清单 30-15: `VSYNC_EXTI_Configuration()` 函数

```

(7)  /*****
(8)  * 函数名: VSYNC_EXTI_Configuration
(9)  * 描述   : OV7670 VSYNC 中断管脚配置
(10) * 输入    : 无
(11) * 输出    : 无
(12) * 注意    : 无
(13) *****/
(14) static void VSYNC_EXTI_Configuration(void)
(15) {
(16)     EXTI_InitTypeDef EXTI_InitStructure;
(17)
(18)     GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);
(19)
(20)     EXTI_InitStructure.EXTI_Line = EXTI_Line0;
(21)     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
(22)     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; /*上
升沿触发*/
(23)     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
(24)     EXTI_Init(&EXTI_InitStructure);
(25)     EXTI_GenerateSWInterrupt(EXTI_Line0); /*中断挂
到 EXTI_Line0 线*/
(26) }

```

设置好了中断，还需要为它分配优先级，这由子函数 `VSYNC_NVIC_Configuration()` 完成，见代码清单 30-16。

代码清单 30-16: `VSYNC_NVIC_Configuration()` 函数

```

27. /*****
28. * 函数名: VSYNC_NVIC_Configuration
29. * 描述   : VSYNC 中断配置
30. * 输入    : 无
31. * 输出    : 无
32. * 注意    : 无
33. *****/
34. static void VSYNC_NVIC_Configuration(void)
35. {
36.     NVIC_InitTypeDef NVIC_InitStructure;

```

```
37.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
38.     NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
39.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;

40.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
41.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
42.     NVIC_Init(&NVIC_InitStructure);
43. }
```

## 50. 编写检测 VSYNC 的中断服务函数

由于 VSYNC 出现两次上升沿，才表示 FIFO 保存了一幅图像，所以在检测 VSYNC 上升沿的中断服务函数中，我们使用一个变量 OV7670\_VSYNC 作为标志。OV7670\_VSYNC 标志的初始值为 0，当检测到第一次上升沿时，我们控制 FIFO 的相应 GPIO 引脚，允许 OV7670 向 FIFO 写入图像数据，并把标志值设置为 1；检测到第二次上升沿时，禁止 OV7670 写 FIFO，把标志设置为 2，在 main 函数中的循环判断中，当 OV7670\_VSYNC=2 时，STM32 开始从 FIFO 读取数据并显示，读取完毕后把 OV7670\_VSYNC 标志复位为 0，重新开始下一幅图像的采集。

中断服务函数 EXTI0\_IRQHandler()位于 stm32f10x\_it.c 文件，见代码清单 30-17。

### 代码清单 30-17: EXTI0\_IRQHandler()中断服务函数

```
1.
2. void EXTI0_IRQHandler(void)
3. {
4.     if ( EXTI_GetITStatus(EXTI_Line0) != RESET )
5.         //检查 EXTI_Line0 线路上的中断请求是否发送到了 NVIC
6.     {
7.         if( OV7670_VSYNC == 0 )
8.         {
9.             FIFO_Wrst_L(); //拉低使 FIFO 写(数据 from 摄像头)指针复位
10.            FIFO_We_H();   //拉高使 FIFO 写允许
11.            OV7670_VSYNC = 1;
12.            FIFO_We_H();   //使 FIFO 写允许
13.            FIFO_Wrst_H(); //允许使 FIFO 写(数据 from 摄像头)指针运动
14.        }
15.        else if( OV7670_VSYNC == 1 )
16.        {
17.            OV7670_VSYNC = 2;
18.            FIFO_We_L();   //拉低使 FIFO 写暂停
19.        }
20.        EXTI_ClearITPendingBit(EXTI_Line0);
21.        //清除 EXTI_Line0 线路挂起标志位
22.    }
23.
24. }
```

## 51. 读 FIFO 并显示图像

采集得的图像数据都保存到摄像头模块的 FIFO 中，在 OV7670\_VSYNC 标志变为 2 的时候，STM32 即可读取它并显示到 LCD 上。与 FIFO 相关的函数有 FIFO\_GPIO\_Configuration()、FIFO\_PREPARE 和 Get\_imag\_and\_discor()。

□ FIFO\_GPIO\_Configuration() 类似 SCCB\_GPIO\_Configuration() 函数，完成了基本的 GPIO 初始化，见代码清单 30-18。

代码清单 30-18: FIFO\_GPIO\_Configuration() 函数

```
1.  /*****
2.  * 函数名: FIFO_GPIO_Configuration
3.  * 描述   : FIFO GPIO 配置
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 注意   : 无
7.  *****/
8.  void FIFO_GPIO_Configuration(void)
9.  {
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC | RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE, ENABLE);
12.
13.     /*PB5(FIFO_WRST--FIFO 写复位)*/
14.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
15.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
16.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
17.     GPIO_Init(GPIOB, &GPIO_InitStructure);
18.
19.     /*PD3(FIFO_WEN--FIFO 写使能) PD6(FIFO_REN--FIFO 读使能)*/
20.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_6;
21.     GPIO_Init(GPIOD, &GPIO_InitStructure);
22.
23.     /*PE0(FIFO_RRST--FIFO 读复位) PE2(FIFO_RCLK--FIFO 读时钟)*/
24.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_2;
25.     GPIO_Init(GPIOE, &GPIO_InitStructure);
26.
27.     /*PB8-PB15(FIFO_DATA--FIFO 输出数据)*/
28.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
29.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
30.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
31.     GPIO_Init(GPIOB, &GPIO_InitStructure);
32.
33.     FIFO_CS_L(); /*拉低使 FIFO 输出使能*/
34.     FIFO_WE_H(); /*拉高使 FIFO 写允许*/
35.
36. }
```

□ FIFO\_PREPAGE 实际是一个宏，它是在 main 函数中，判断到接收完成一幅图像后被调用的，它的作用是把 FIFO 读指针复位，使后面的数

据读取从 FIFO 的 0 地址开始，其代码见代码清单 30-19(在工程中，要把宏定义写在同一行或用续行符)。

代码清单 30-19: FIFO\_PREPAGE 宏

```

⑦
⑧ #define FIFO_PREPAGE \
⑨ do{\
⑩ FIFO_RRST_L();\
11 FIFO_RCLK_L();\
12 FIFO_RCLK_H();\
13 FIFO_RRST_H();\
14 FIFO_RCLK_L();\
15 FIFO_RCLK_H();\
16 }while(0)
17

```

□ Get\_imag\_and\_discor()函数完成了从 FIFO 读取图像及显示到 LCD 的工作，每当 OV7670 输出完一幅图像，它被调用一次，在调用前，要用上面的 FIFO\_PREPAGE 宏复位 FIFO 读指针。它的代码见代码清单 30-20。

代码清单 30-20: Get\_imag\_and\_discor()函数

```

16.
17. void Get_imag_and_discor(void)
18. {
19.     u16 i, j;
20.     u16 Camera_Data;
21.     for(i = 0; i < 240; i++)
22.     {
23.         for(j = 0; j < 319; j++)
24.         {
25.
26.             READ_FIFO_PIXEL(Camera_Data);
27.             //从 FIFO 读出一个 rgb565 像素到 Camera_Data 变
量
28.             LCD_WR_Data(Camera_Data);
29.         }
30.         READ_FIFO_PIXEL(Camera_Data);
31.         LCD_WR_Data(0);
32.     }
33. }
34.

```

在代码中，循环调用了宏 READ\_FIFO\_PIXEL 读取 FIFO 数据，并使用 LCD\_WR\_Data()函数把该图像数据显示到 LCD 上。宏 READ\_FIFO\_PIXEL 代码见代码清单 30-21(在工程中，要把宏定义写在同一行或用续行符)。

代码清单 30-21: READ\_FIFO\_PIXEL 宏

```
25.  
26. #define READ_FIFO_PIXEL( RGB565 ) \  
27. do{ RGB565=0; \  
28. FIFO_RCLK_L(); \  
29. RGB565 = (GPIOB->IDR) & 0xff00; \  
30. FIFO_RCLK_H(); \  
31. FIFO_RCLK_L(); \  
32. RGB565 |= (GPIOB->IDR >>8) & 0x00ff; \  
33. FIFO_RCLK_H(); \  
34. }while(0)  
35.
```

这个宏把 FIFO 读取到的数据按 RGB565 的处理，保存到一个 16 位的变量中，LCD\_WR\_Data()函数就可以直接利用这个数据，显示到 LCD 上了。

回顾一下摄像头的驱动流程，首先要设计 SCCB 时序，根据 OV7670 数据手册进行配置，然后利用 VSYNC 中断，利用标志变量作状态标记，最后在 main 函数中判断 OV7670 输出完成一幅图像后，调用 FIFO 相关函数进行读取并显示。

#### 30.5.7. 实验现象

把 OV7670 摄像头模块接入将野火 STM32 开发板的 CAMERA 接口，然后给开发板供电(DC5V)，插上 JLINK，将编译好的程序下载到开发板。即可看到 LCD 上输出摄像头拍到的图像，若图片显示不够清晰，可调整镜头进行调焦，使得到清晰的图像。

### 31. 6、I2S-MP3



## 32. 7. *Usb-device*

USB 作为电子设备中最常用连接方式，读者一定不会觉得陌生。USB 是 (Universal Serial BUS) 的缩写，译作通用串行总线，由于它易于扩充展、价格低廉、易于升级、速度快和支持热插拔等优点，被广泛用于与 PC 相连的设备中。

### 32. 1. USB 协议分析

#### 32. 1. 1. 协议版本

曾被广泛使用的 USB 协议版本有 USB1.1、USB2.0，而目前公布的最新 USB 协议为 USB3.0，主要由于数据线的增加，USB3.0 数据传输速度有了很大的提高。对于 USB1.1 协议支持低速模式(1.5Mb/s)、全速模式(12Mb/s)，而 USB2.0 协议还支持高速模式(480Mb/s)。不同的器件对 USB 协议的支持是不一样的，如于本书采用的 STM32 芯片仅支持 USB2.0 中的全速模式(12Mb/s)。在本文中关于 USB 协议的解释都是基于 USB2.0 全速模式的，在不同的协议或不同的模式下会有一些区别。

#### 32. 1. 2. USB 电气特性

##### ■ USB 接口

USB 协议规定的 USB 接口主要有 A 和 B 两个类型，每种类型有对应的插头和插座，见图 32-1。A 型接口主要用于 USB 主机，B 型则主要用于 USB 集线器或 USB 设备。

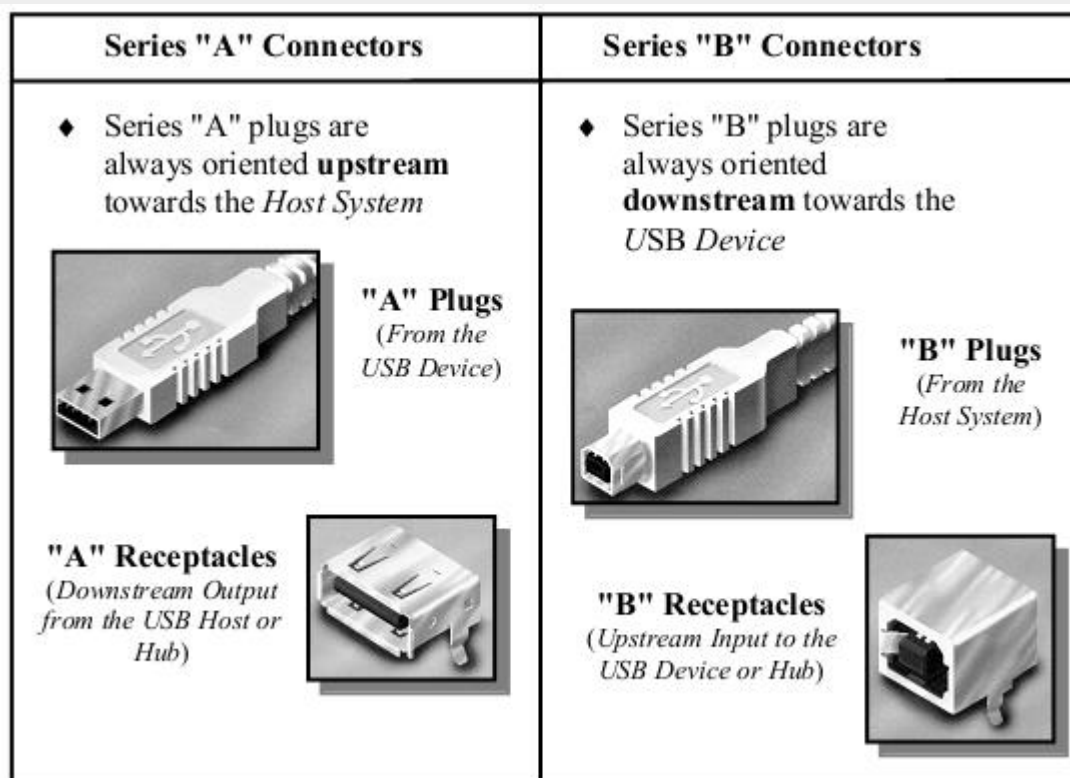


图 32-1 USB 的接口类型

拿起一根 USB 线缆，使 USB 标志  朝上，可以看到有四根露出的金属触点，对于 A 型插头，从左往右：第 1 脚为  $V_{BUS}$ 、2 脚为 D<sup>-</sup>、3 脚为 D<sup>+</sup>、4 脚为 GND，而对于 B 型插头，左上角的为第 1 脚，按顺时针方向数为 2、3、4 针脚。USB 插座的引脚安排则与相应的插头对应，在 USB2.0 协议书中有很详细的规格。

### ■ USB 线缆

USB 线缆中的四个金属触点对应的导线  $V_{BUS}$  为 5V 电源线，GND 为电源地线，而 D<sup>-</sup> 和 D<sup>+</sup> 组合成一组差分数据线。在 USB 插头的设计中， $V_{BUS}$  和 GND 的触点都比 D<sup>-</sup> 和 D<sup>+</sup> 的长，这是为了支持热插拔设计的。D<sup>-</sup> 和 D<sup>+</sup> 这一组差分数据线，在一个时刻只能传输一路信号，这也决定了 USB2.0 的全速模式只能是半双工的通讯方式。

## ■ NRZI 编码

在 CAN 的章节中，我们知道 CAN 总线它没有使用时钟信号，所以提出了一种特殊的位同步方法，协调通讯双方。而 USB 也是仅有一组差分线作为信号线，没有时钟信号，所以也必须解决信号同步的问题，首先它采用 NRZI 编码方式，见图 32-2 NRZI 编码方式。

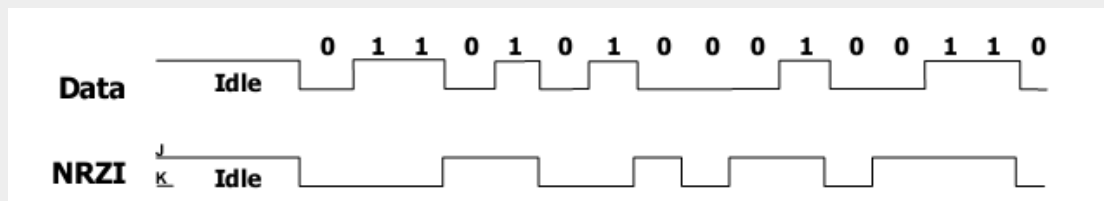


图 32-2 NRZI 编码方式

NRZI 编码方式是指把原数据流进行处理，原数据为 0 时，编码成电平翻转，原数据为 1 时，编码成电平不翻转。USB 要发送数据时，硬件把数据流转化成 NRZI 编码流，再经过两条差分线把 NRZI 编码发送出去，收数据时作逆处理。

在 USB 的各种数据包的头部，都有一个同步域，同步域实际是一串长度为 7 位，且 7 位都为 0 的数据，这串数据经 NRZI 编码后，就是一串 0101010 的方波，且方波的频率即为发送的波特率。接收方即可通过各种协调机制，利用这个方波同步域，与发送方协调通讯。

## 32.1.3. USB 通讯模型

从 USB 线缆及其接口可以发现，USB 的物理结构其实跟 CAN 总线是很相似的，都是使用一组差分数据线，只是多了电源和地。只要我们愿意，把 CAN 总线改装成使用 USB 接口，也可以利用这个接口为设备提供电源，而设备之意则使用 CAN 协议通讯，就成了山寨版的 USB 了。虽然 USB 与 CAN 的物理接口很相似，但它们的核心——通讯模型、协议却有着具大的差别，正是这些差别，使得它们在各自的领域大放异彩。

## ■ USB 系统架构

我们先从整体来看 USB 通讯的系统架构，见图 32-3 USB 系统架构。

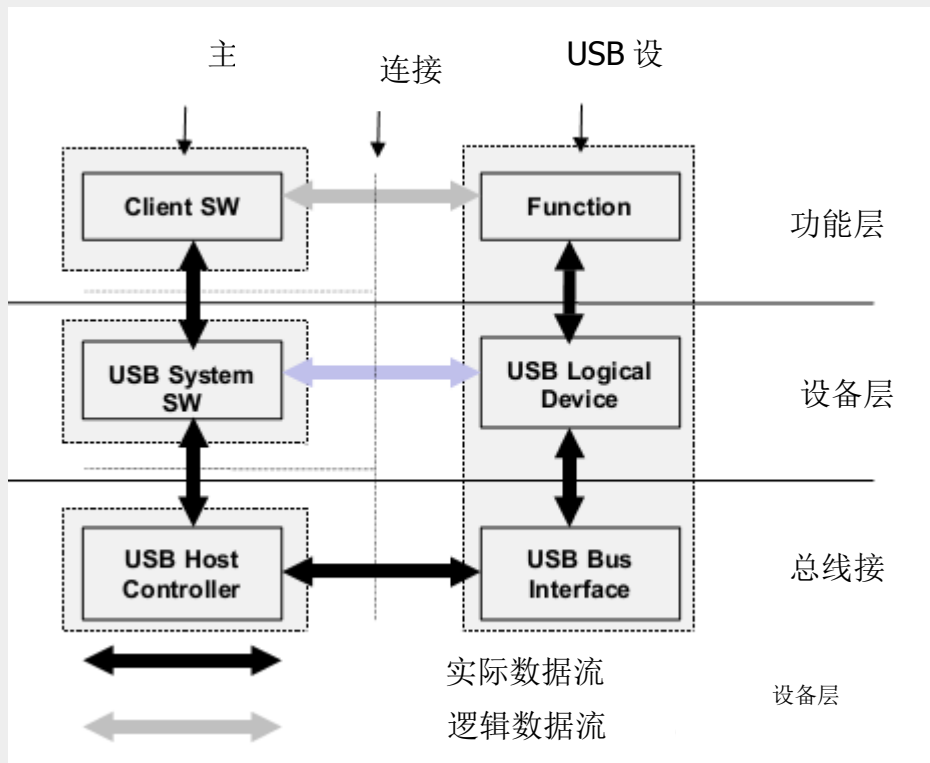


图 32-3 USB 系统架构

图中 USB 系统架构模型的分层思想与其它通讯系统是很类似的。对主机端和 USB 设备端的通讯结构分为三层，PC 主机端的客户端软件(Client SW)与 USB 的应用程序(Function)属于功能层，USB 系统软件(USB System SW)与 USB 逻辑设备(USB Logical Device)属于设备层，USB 主机控制器(USB Host Controller)与 USB 设备总线接口(USB Bus Interface)属于总线接口层。实际的数据通讯都通过总线接口层传输，软件封装，使得主机、设备在设备层、功能层的通讯可以忽略下层对数据包的处理，从而感觉上主机与设备的通讯是在同一个层次的，这样处理也更为简单、可扩展性强。

总线接口层主要负责包括 NRZI 编码、传输速率等，为功能层提供 USB 通讯中的端点、通道、缓冲区等物理支持。它可以理解为物理层。

设备层主要负责解释各种通讯包，进行各种 USB 事务管理，针对不同的情况按命令状态机进行状态切换等。它可以理解为逻辑层。

功能层主要就是根据不同的应用而开发的，需要主机的客户端软件进行配合。它可以理解为应用层。

## ■ USB 主机与设备

USB 主机与设备的连接、通讯方式是 USB 逻辑通讯结构的基础。在 USB 的通讯模型中要严格区分 USB 主机(Host)与 USB 设备(Device)，因为在 USB 的通讯中，有且只有一个主机。当检测到有 USB 设备接入时，首先由主机发起第一次通讯。

以 PC 中的 USB 通讯模型为例，见图 32-4。PC 作为 USB 通讯中的主机，它拥有 USB 主控制器，在内部它与 USB 根集线器(Root hub)相连，向外给 USB 设备提供 USB 插座。USB 设备包括 USB 鼠标、键盘、音响等，还有用于扩展 USB 接口的特殊的 USB 设备——USB 集线器(USB hub)，如常见的“一拖五”。一个 USB 主控制器同时最多可支持 127 个设备，这是由于 USB 协议中规定 USB 设备接入后，主控制器给它分配一个 7 位的设备地址，以后就使用该地址进行通讯，所以使用 USB 集线器能扩展接口，但无法扩展主控制器可支持设备数目的上限。

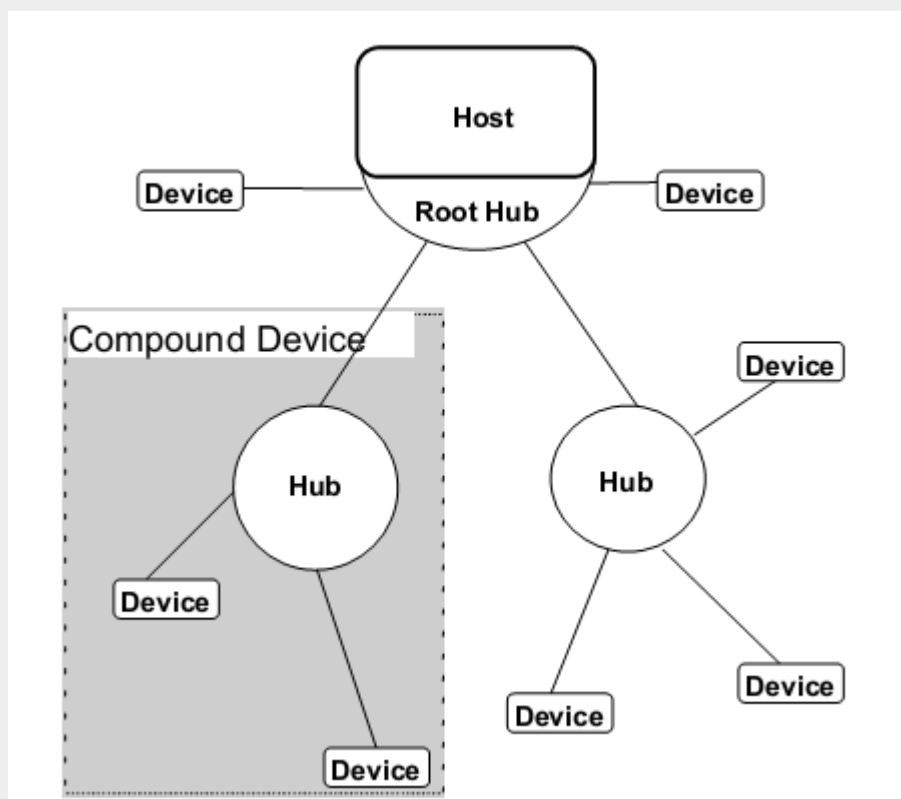


图 32-4 USB 主机、设备连接模型

大部分 USB 控制器只支持 USB 设备的功能，如 STM32F103 系列芯片的 USB 控制器外设仅可以作为 USB 设备的控制器，而不能用于 USB 主机控制器。在 STM32F105、STM32F107 系列芯片中的 USB 控制器称为“USB OTG 全速控制器(OTG FS)”，它带有 OTG (on the go) 功能。OTG 是 USB2.0 协议中规定的双重角色功能，这种控制器仅可作为 USB 的主机端，也可以作为 USB 通讯中的设备端，这在手机，平板电脑中应用十分广泛。在这些设备中使用的一般为 USB MINI 接口，虽然是 MINI，但它比 A、B 型 USB 接串口多出了一条用于区别主机和设备的 ID 线。

### ■ USB 端点

在 USB 通讯模型中，端点 (Endpoints) 是一个非常重要的概念，见图 32-5。

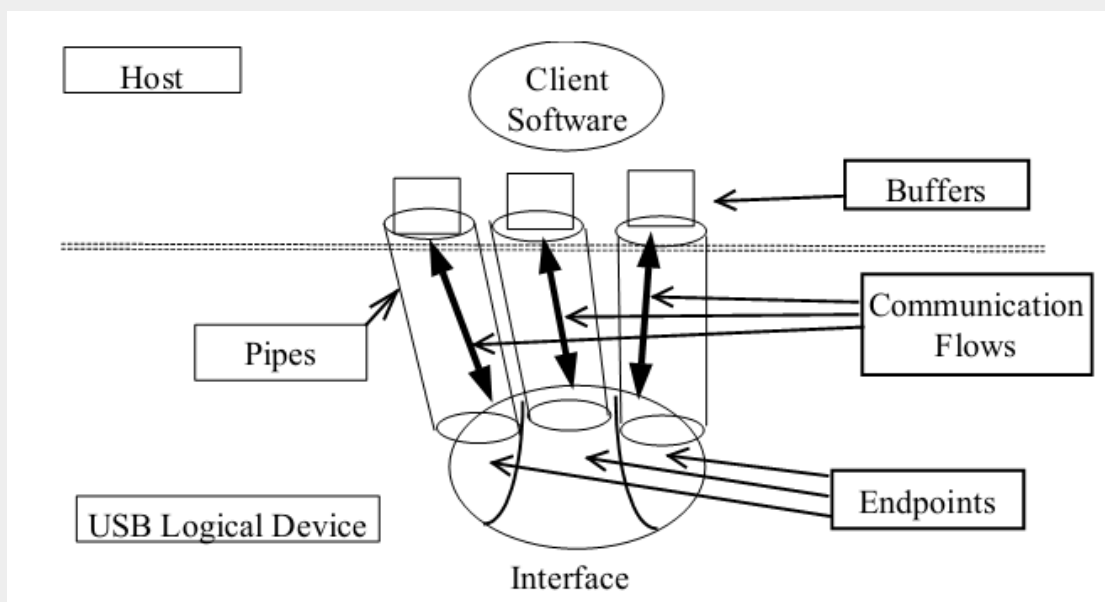


图 32-5 USB 端点、通道

在一个 USB 接口(Interface)中, 含有多个端点, 它是 USB 主机与设备通讯的节点。每个端点都有特定的编号、传输方向、最大数据包长度等属性。在主机和设备上的两个端点构成了一个数据通道(Pipe), 数据就在这通道间转移。通常在特定的端点中(即对应的通道), 传输指定的数据。如端点 0 是特殊端点, 默认用于接收和发送初始化控制信息, 利用它在通讯双方进行基本的协调, 之后才能使用其它的端点进行通讯。在 USB 存储器应用中, 端点 0 用于初始化, 建立 USB 通讯, USB 主机的端点 1 用于批量接收数据, 端点 2 用于批量输出数据。

通道、接口的关系可以与高速公路的收费站进行类比, 一个收费站(接口)拥有多个收费通道, 要注意 USB 虽然划分了多个端点和通道, 但由于只有一组差分数据线, 所以任一时刻至多只传输一个通道的数据(命令)。

### ■ USB 传输类型

USB 协议定义了四种传输类型, 它用于描述端点和通道的传输特性, 就如同公路规定的大车道、小车道一样, 这样规定有利于管理和提高效率。

□ 控制传输类型(Control Transfers): 突发、非周期的传输, 主要用于传输命令和状态消息。端点 0 一般就被配置为控制传输。



- 同步传输类型(Isochronous Transfers): 周期性、持续的传输, 常用于与时间相关的事件、信息的传送。如音频设备中的音频数据。
- 中断传输类型(Interrupt Transfers): 传输频率低、数据量小的传输。如鼠标、键盘设备的控制信息就是以中断传输方式传送的。
- 批量传输类型(Bulk Transfers): 非周期性的, 大数据量的传输。如 USB 存储器主要使用这种方式传输数据。

#### 32.1.4. USB 枚举

当 USB 主机检测到有新设备连接进来后, 主机会利用端点 0, 以控制传输的方式向设备发送各种请求, USB 设备对主机的这些请求进行应答, 以使主机识别该设备这个识别过程称为枚举。枚举成功, 才能建立起正式的 USB 通讯。枚举过程简介如下:

主机需要知道接入的是什么设备, 使用的协议版本、端点 0 最大数据包长度等信息, 这些信息由 USB 协议规定的设备描述符 来进行记录, 它是一串数据, 在程序中可以用结构体来表示。协议具体规定哪几个字节表示设备使用的 USB 协议版本、哪几个字节表示端点 0 的数最大数据包长度等等。记录设备其它的信息还有配置描述符、接口描述符、端点描述符等。

(1) 为了获得设备描述符, 主机首先使用地址 0, 向接入的设备发送 USB 标准请求: `Get_Device_Descriptor`(获取设备描述符)。正常时, 设备会给主机返回它自己的设备描述符, 但由于第一次通讯不知道端点 0 支持数据包的最大长度, 所以主机只能通过设备描述符中的第 8 个字节了解设备端点 0 的最大数据包长度, 这 8 个字节以外的信息还没法了解。

(2) 主机为设备分配一个新地址, 把这个地址存放到标准请求 `Set_Address`(设置地址)中, 发送这个请求给设备, 设备保存该地址, 以后的通讯就使用这个新地址。

(3) 主机重新向设备发送 `Get_Device_Descriptor`(获取设备描述符) 请求, 这次主机会完全读取设备返回的设备描述符, 了解设备的信息。



(4) 主机向循环设备发送 `Get_Device_Configuration`(获取配置描述符)请求, 获得设备的配置描述符、接口描述符、类特殊描述符、端点描述符。

(5) 主机发送 `Get_Device_String`(描述字符集) 获取厂商 ID、产品描述、型号等信息。

(6) 若 USB 能提供该设备的驱动, 主机向设备发送 `Set_Configuration`(选择设备配置)请求设备进入某个配置状态。

(7) 建立通讯。

### 32.2. STM32 的 USB 控制器

STM32F103 系列的 USB 控制器外设(不含 OTG 功能)可用于制作 USB 设备。其主要特性如下:

17. 符合 USB2.0 全速设备的技术规范, 即速度为 12Mb/s

18. 可配置 1 到 8 个 USB 端点, 其中的 8 个端点均为双向端点, 若配置的都是单向端点, 则可配置 16 个。

19. CRC(循环冗余校验)生成/ 校验, 反向不归零(NRZI)编码/ 解码和位填充

20. 支持同步传输

21. 支持批量/ 同步端点的双缓冲区机制, 为实现 USB 存储器提供了良好的支持。

22. 支持 USB 挂起/ 恢复操作, 这有利于降低功耗。

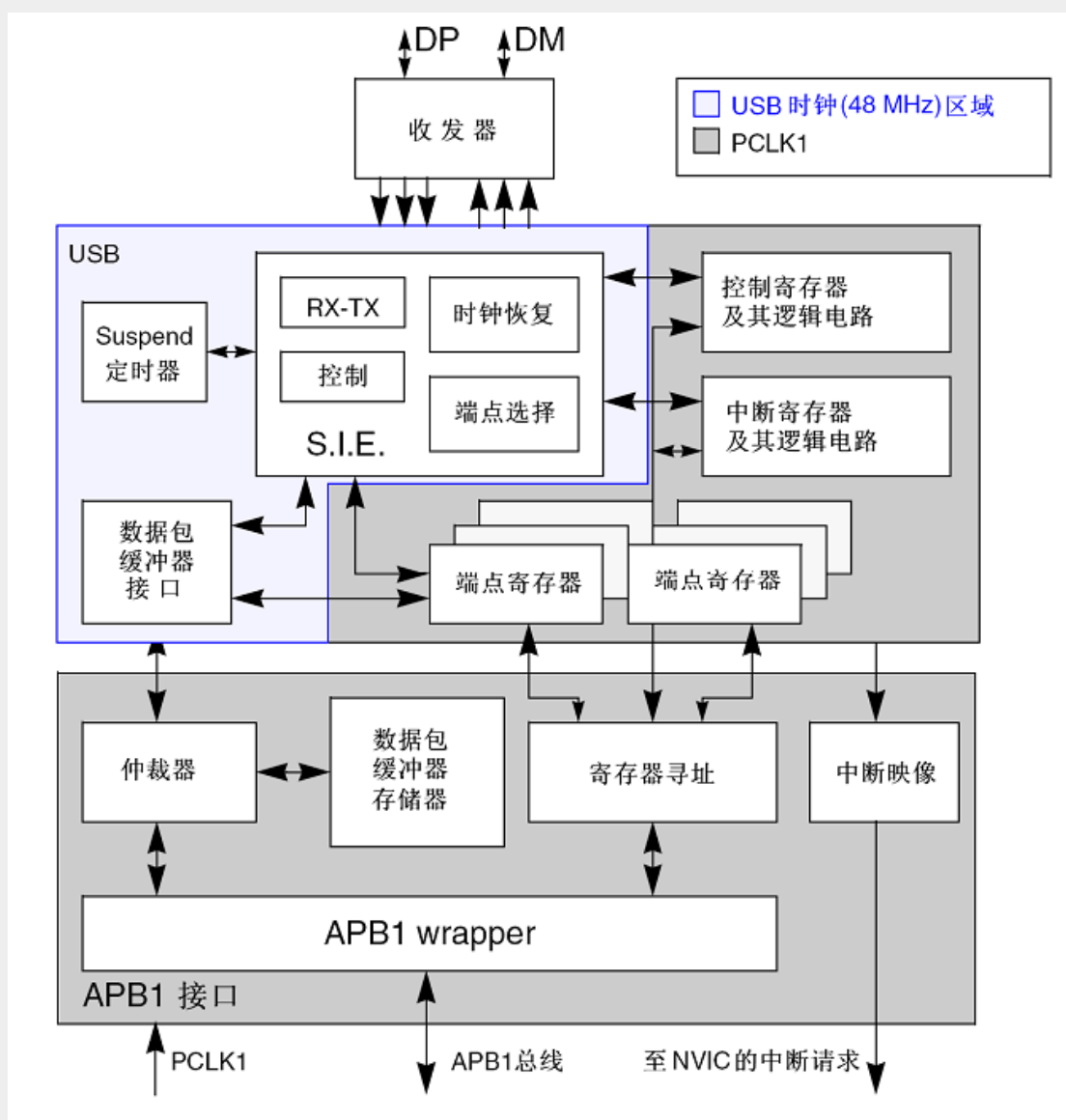


图 32-6 STM32 USB 控制器结构

见图 32-6，这是 STM32F103 系列的 USB 控制器的结构图。它主要具有如下结构：

**USB 收发器：**整个控制器通过 DP(D+)、DM(D-)与 USB 收发器相连，USB 收发器的作用跟 CAN 的收发器作用是类似的，主要负责一些协议相关的数模转换工作和设备接入检测。

**串行接口控制器(SIE):**主要负责识别 USB 帧的同步域识别、位填充、CRC 校验等。根据 USB 事件、传输结束、正确接收或端点相关的事件产生各种信号，这些信号可用于触发 USB 中断事件。

SIE 部件、数据包缓冲器接口、Suspend 定时器工作于 USB 48MHz 的时钟下，它由 PLLCLK 经 USB 预分频器得到。

通过配置和读取与 SIE 交互的控制寄存器、中断寄存器、端点寄存器，我们就可以利用 USB 外设进行 USB 通讯了。

USB 控制器还为 PC 主机和微控制器所实现的功能之间提供了符合 USB 规范的通信连接。PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区，最多可用于 16 个单向或 8 个双向端点。

开发 STM32 的 USB 驱动程序，主要是利用不同的中断信号，通过读状态寄存器 确定哪个端点需要得到服务、正在进行哪种类型的数据传输。

有关更多 USB 的介绍请参考《STM32 参考手册》。USB 是一个很复杂的设备，要想全面的学习 USB，光靠做完这个实验和看《STM32 参考手册》是远远不够的，还要大量阅读 ST 的官方说明和 USB 协议文档。还有网上有本关于 USB 方面的书《圈圈教你学 USB》很不错，大家可以去买来研究研究。总之，一句话，USB 耐学。

### 32.3. USB 读取 SD 卡——模拟 U 盘实验

#### 32.3.1. 实验描述及工程文件清单

##### 实验描述：

这是一个 USB Mass Storage (USB 存储器)实验，用 USB 线连接 PC 机与开发板，在电脑上就可以像操作普通 U 盘那样来操作开发板中的 MicroSD 卡，并在超级终端中打印出相应的调试信息。在这里野火用的 MicroSD 卡的容量是 1G，4G 的 SD 卡也已通过测试。

##### 硬件连接：

PE3-USB-MODE (PE3 为普通 I/O)

PA11-USBDM (D-)

PA12-USBDP (D+)

库文件：使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_usart.c

FWlib/stm32f10x\_dma.c

FWlib/stm32f10x\_sdio.c

FWlib/misc.c

**USB 库：**

usb\_core.c

usb\_init.c

usb\_mem.c

usb\_regs.c

usb\_desc.c

usb\_endp.c

usb\_bot.c

usb\_scsi.c

scsi\_data.c

**用户文件：**

USER/main.c

USER/stm32f10x\_it.c

USER/usart1.c

USER/sdio\_sdcard.c

USER/usb\_istr.c

USER/usb\_prop.c

USER/usb\_pwr.c

USER/hw\_config.c

USER/memory.c

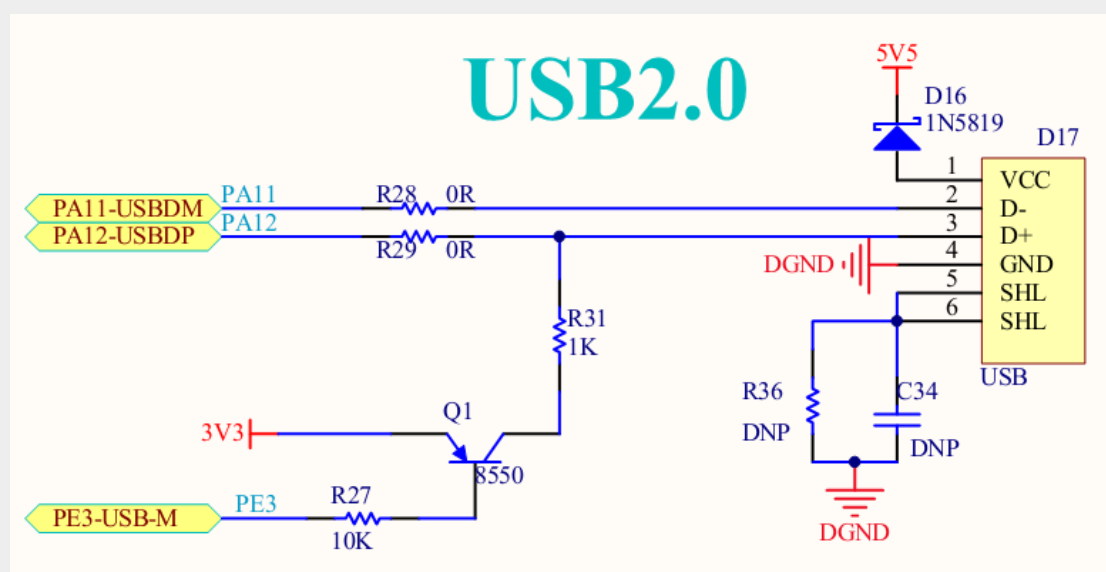


图 32-7 野火 STM32 USB 模块原理图

### 32.3.2. 配置工程环境

本 USB Mass Storage 实验中我们用到了 GPIO、RCC、USART、SDIO、DMA 外设，所以我们要把以下库文件添加到工程 `stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_usart.c`、`stm32f10x_sdio.c`、`stm32f10x_dma.c`。在 SDIO 及 USB 外设中还使用了中断，所以要把 `misc.c` 文件添加进工程。

使用串口来调试是必须的，添加用户文件 `usart1.c`。本实验是在 SDIO 驱动的基础上建立的，所以还需要把用户文件 `sdio_sdcard.c` 文件添加到工程之中。

咦？USB 相关的库文件呢？STM32 的 USB 外设和以太网外设的固件库都是独立出来的，不被包含在标准库中，所以需要到官网下载相应的固件库。

目前最新的 USB 相关固件库版本为 3.4 ，下载得的压缩文件名为：  
stm32\_usb-fs-device\_lib.zip ，解压后可以在 STM32\_USB-FS-Device\_Driver 文件夹中找到 USB 外设源文件和头文件。我们直接把 STM32\_USB-FS-Device\_Driver 文件夹复制到工程目录下，在编译选项中添加头文件包含路径，并添加 USB 库的源文件：usb\_core.c 、usb\_init.c 、usb\_int.c 、usb\_mem.c 、usb\_regs.c 、usb\_sil.c。其中 USB 固件库文件名中有的带“OTG”字符，这些文件只在 STM32F105 或 STM32F107 系列的芯片才用到，在本实验中不需要添加。

USB 固件库中还有一些 STM32 官方提供的 USB 应用例程，我们可以利用它来进行开发，这些例程在它的 project 文件夹下。

打开下载得来的 USB 库目录：STM32\_USB-FS-stm32\_usb-fs-device\_lib\STM32\_USB-FS-Device\_Lib\_V3.4.0\Project\Mass\_Storage

这是 USB Mass Storage 例程，找到该目录下的 src 和 inc 文件夹，复制这些文件到自己的工程，删除重复文件，最终添加了：hw\_config.c 、mass\_mal.c 、memory.c 、scsi\_data.c 、usb\_bot.c 、usb\_desc.c 、usb\_endp.c 、usb\_istr.c 、usb\_prop.c 、usb\_pwr.c 、usb\_scsi.c 文件(别忘记添加头文件及其路径呀)。

完成后野火的 USB 工程目录如图 32-8。

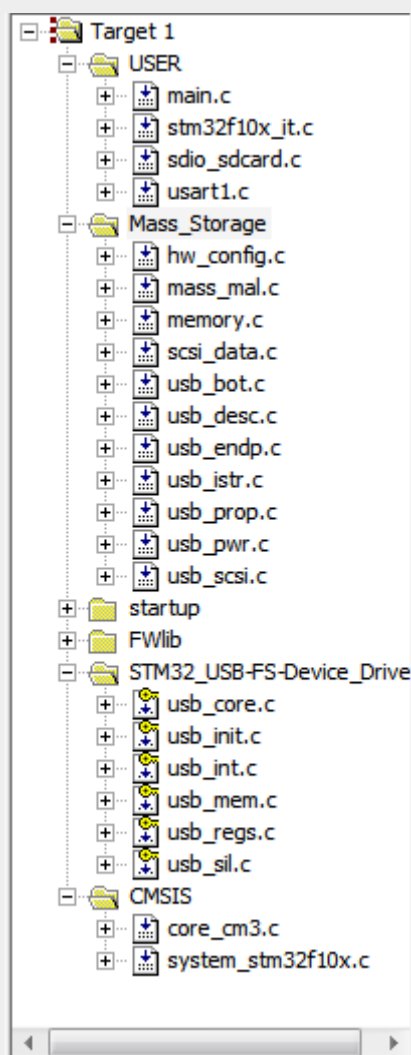


图 32-8 USB 工程目录

最后在 `stm32f10x_conf.h` 中把使用到的 ST 库的头文件注释去掉即可(USB 的固件库是不在 `stm32f10x_conf.h` 文件的管理范围的)。见代码清单 32-1。

代码清单 32-1: USB 例程的 `stm32f10x_conf.h` 文件配置

```
1.  /*****
2.   * @file      Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
3.   * @author    MCD Application Team
4.   * @version    V3.5.0
5.   * @date      08-April-2011
6.   * @brief     Library configuration file.
7.   *****/
8.  #include "stm32f10x_dma.h"
9.  #include "stm32f10x_gpio.h"
10. #include "stm32f10x_rcc.h"
11. #include "stm32f10x_sdio.h"
12. #include "stm32f10x_usart.h"
13. #include "misc.h"
```

## 32.3.3. USB 固件库说明

USB 固件库包含非常多的文件，它们的关系见图 32-9。

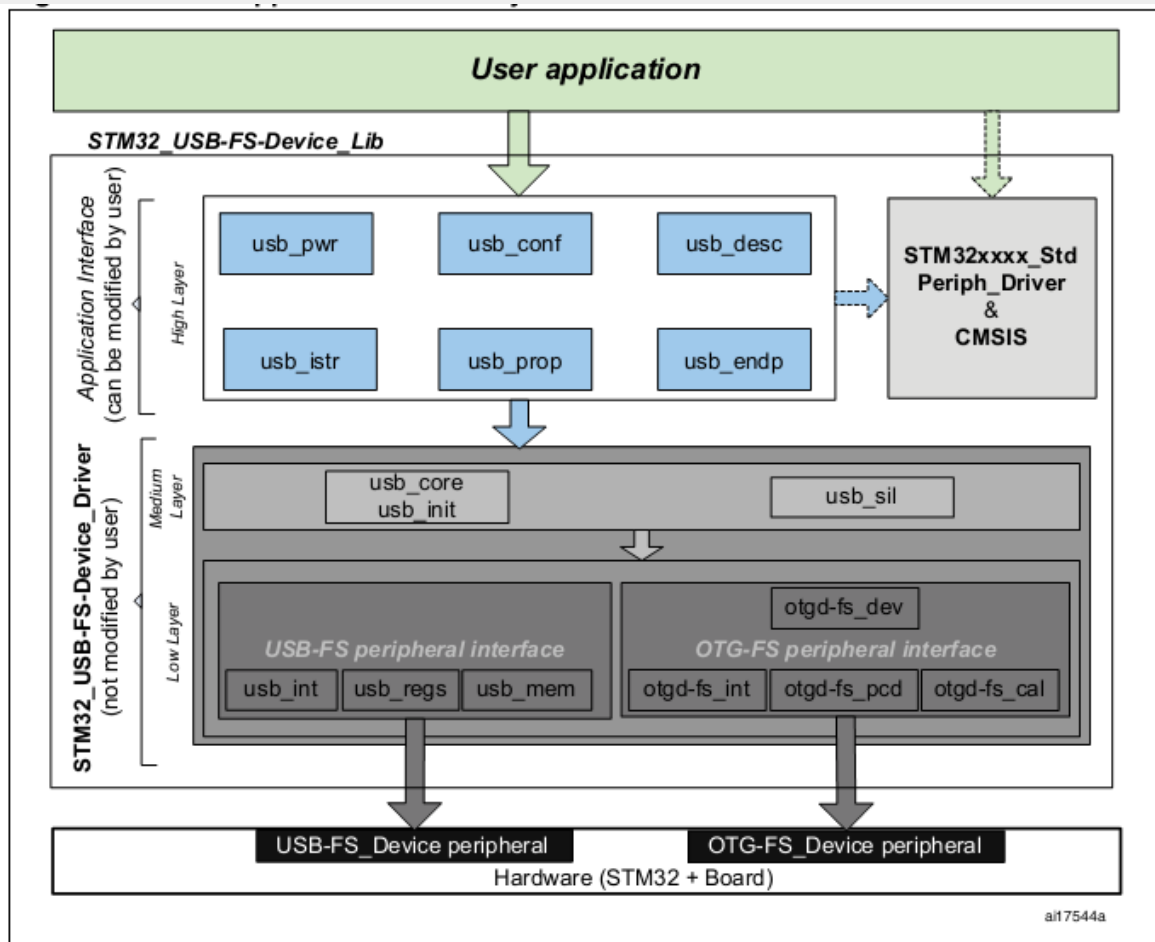


图 32-9 USB 固件库文件关系

该图清晰地说明了各文件所处的位置及作用。它把这些文件按三层整理如下：

□ 最底层：

(1) `usb_int.c`，本文件包含两个中断服务函数 `CTR_LP()` 和 `CTR_HP()`，在接收发送中断时使用这两个函数进行处理，这两个函数主要又调用各个 USB 端点的处理函数，端点处理函数由用户自定义。

(2) `usb_regs.c`，本文件是硬件抽象层。它含有各种用于读取或设置 USB 寄存器的函数，相当于对寄存器操作方法的封装。



(3) `usb_mem.c`，本文件主要负责进行数据传输处理。它的函数主要负责数据在用户区域(内存)或 USB 数据包缓冲区(参考 STM32 USB 结构图)的互相转移。

□ 中间层：

(1) `usb_core.c`，本文件主要负责 USB 协议管理，如 USB 标准请求、各种端点 0 的控制信息处理。

(2) `usb_init.c`，本文件主要包含了 USB 初始化的函数。

(3) `usb_sil.c`，本文件包含精简接口的初始化或向端点读写的操作函数。

□ 上层：

4. `usb_pwr.c`，该文件中包含处理上电、调电、挂起和恢复事件的函数。

5. `usb_istr.c`，该文件中只有一个函数，即 USB 中断的 `USB_Istr` 函数，该函数对各类引起 USB 中断的事件作轮询处理。

6. `hw_config.c (usb_conf.c)`，该文件中包含系统配置的函数，包含基本的时钟配置、中断配置和存储器初始化函数。

7. `usb_desc.c`，该文件包含一些与 USB 相关的设备描述符、配置描述符等，以数组形式存储，在 USB 主机请求的时候，这些信息会发送给主机。

8. `usb_prop.c`，该文件用于实现相关设备的 USB 协议，例如初始化、SETUP 包、IN 包、OUT 包等等。

9. `usb_endp.c`，本文件包含端点收、发送的处理函数，由用户根据不同的 USB 设备进行不同的定义。

USB 的应用软件在这三层之上，在不同的 USB 应用中，用户可能需要对 USB 固件库的上层部分文件配置进行修改(主要为 `usb_prop.c` 和 `usb_endp.c` 文件)。而中间层和底层封装得非常好，对这些文件用户是不需要进行改动的。由

于本实验的文件较多，采用 Source Insight 软件 来分析代码有利于帮助读者理清各函数的调用关系。

#### 32.3.4. main 文件

在配置好我们需要用到的库文件之后，我们就从 main 函数开始分析。见代码清单 32-2。

代码清单 32-2: USB 例程的 main 函数

```
1.  /* Includes -----*/
2.  #include "stm32f10x.h"
3.  #include "sdio_sdcard.h"
4.  #include "usart1.h"
5.  #include "hw_config.h"
6.  #include "usb_lib.h"
7.  #include "usb_pwr.h"
8.
9.
10.
11.
12. int main(void)
13. {
14.
15.     /*进入到 main 函数前，启动文件 startup (startup_stm32f10x_xx.s)
已经调用    了在 system_stm32f10x.c 中的 SystemInit()，配置好了系统时钟，在外部晶振 8M 的条件下，设置 HCLK = 72M */
16.
17.
18.     /* USART1 config */
19.     USART1_Config();
20.
21.     /*初始化 SD 卡*/
22.     Set_System();
23.
24.     /*设置 USB 时钟为 48M*/
25.     Set_USBClock();
26.
27.     /*配置 USB 中断(包括 SDIO 中断)*/
28.     USB_Interrupts_Config();
29.
30.     /*USB 初始化*/
31.     USB_Init();
32.
33.     while (bDeviceState != CONFIGURED); //等待配置完成
34.
35.     printf("野火 STM32 USB MASS STORAGE 实验\r\n");
36.
37.
38.     while (1)
39.     {}
40. }
```

在 main 函数中，调用 USART1\_Config()函数 初始化调试用的串口，调用 Set\_System() 函数 初始化 SD 卡。接下来调用 Set\_USBClock()、

USB\_Interrupts\_Config()、USB\_Init()分别用于设置 USB 时钟、配置 USB 的接收和发送中断及 SDIO 中断、对 USB 进行初始化。

在 main 函数初始化完成后，剩下任务都在中断服务函数中完成了，当使用 USB 线把开发板与 PC 连接起来后，作为 USB 主机的 PC 就会开始发送请求给作为 USB 设备的开发板，触发它的中断服务函数，开始 USB 数据传输。

### 32.3.5. 基本配置

#### ■ SDIO 初始化

Set\_System() 主要用于初始化系统中各个外设，在本实验中野火把它简化成只初始化 SD 卡，Set\_System()经过层层函数调用，最终调用了 MAL\_Init() 函数，见代码清单 32-3。

代码清单 32-3: MAL\_Init()函数

```
1.  /*****
***
2.  * Function Name   : MAL_Init
3.  * Description    : Initializes the Media on the STM32
4.  * Input          : None
5.  * Output         : None
6.  * Return         : None
7.  *****/
8.  uint16_t MAL_Init(uint8_t lun)
9.  {
10.     uint16_t status = MAL_OK;
11.
12.     switch (lun)
13.     {
14.         case 0:
15.             Status = SD_Init();
16.             break;
17.         default:
18.             return MAL_FAIL;
19.     }
20.     return status;
21. }
```

当输入参数 `lun = 0` 的时候，本函数调用 `SD_Init()` 进行 SD 卡的初始化，在原版官方的本函数中还有当 `lun =1` 时，进行 NAND FLASH 的初始化，但这需要硬件提供支持。从前面的例程知道 `SD_Init()`之后，我们就可以 SD 卡的读写了，本实验中的 STM32 不需要文件系统，这是因为在 USB Mass Storage 的应用中，文件系统的支持是由 USB 主机提供的，即 PC。

## ■ USB 时钟

接下来，在 main 函数调用的 Set\_USBClock() 函数定义见代码清单 32-4。

代码清单 32-4: Set\_USBClock() 函数

```
(8) /*****
(9) * Function Name : Set_USBClock
(10) * Description  : Configures USB Clock input (48MHz)
(11) * Input       : None.
(12) * Return      : None.
(13) *****/
(14) void Set_USBClock(void)
(15) {
(16)
(17)     RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
(18)
(19)     /* Enable the USB clock */
(20)     RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
(21)
(22) }
```

这个函数用于配置 USB 的时钟，调用库函数 RCC\_USBCLKConfig()把 USB 的时钟预分频设置为 1.5，USB 的时钟是由 PLLCLK 分频得来的，我们实验配置的 PLLCLK 为 72MHz，1.5 分频后正好为 48MHz 作为 USB 外设的时钟。

调用 RCC\_APB1PeriphClockCmd()函数使能 USB 外设的时钟，这时 USB 外设就运行起来了，不需要再配置 USB 外设使用到的管脚模式，在《STM32 参考手册》GPIO 复用功能配置小节说明见表 32-1 USB 复用引脚配置。

表 32-1 USB 复用引脚配置

| USB 引脚   | GPIO 配置                          |
|----------|----------------------------------|
| USB_DM / | 一旦使能了 USB 模块，这些引脚会自动连接到内部 USB 收发 |

## ■ 中断

在 main 中调用用于配置中断的 USB\_Interrupts\_Config()函数定义见代码清单 32-5。

代码清单 32-5: USB\_Interrupts\_Config()函数

```
1. /*****
2. * Function Name : USB_Interrupts_Config
```

```

3.  * Description      : Configures the USB interrupts
4.  * Input            : None.
5.  * Return           : None.
6.  *****/
7.  void USB_Interrupts_Config(void)
8.  {
9.      NVIC_InitTypeDef NVIC_InitStructure;
10.
11.     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
12.
13.
14.     NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
15.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
16.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
17.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
18.     NVIC_Init(&NVIC_InitStructure);
19.
20.     NVIC_InitStructure.NVIC_IRQChannel = USB_HP_CAN1_TX_IRQn;
21.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
22.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
23.     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
24.     NVIC_Init(&NVIC_InitStructure);
25.
26.
27.
28.     NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;
29.     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
30.     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
31.     NVIC_Init(&NVIC_InitStructure);
32.
33.
34.
35. }
```

这个函数配置了 SDIO 及 USB 的低等(LP)和高等(HP)中断的优先级，低优先级一般在控制、中断传输模式下使用，而高优先级中断在快速传输(同步、批量传输)中使用。

### 32.3.6. USB 初始化

在 main 函数中最后调用 USB\_Init()就完成了 USB 的初始化，它的定义见代码清单 32-6。

代码清单 32-6: USB\_Init()函数

```

51. /***/
52. * Function Name    : USB_Init
53. * Description      : USB system initialization
54. * Input            : None.
55. * Output           : None.
56. * Return           : None.
57. *****/
58. void USB_Init(void)
59. {
60.     pInformation = &Device_Info;
```

```
61. pInformation->ControlState = 2;
62. pProperty = &Device_Property;
63. pUser_Standard_Requests = &User_Standard_Requests;
64. /* Initialize devices one by one */
65. pProperty->Init();
66. }
```

本函数中有多个我们不认识的结构体指针，它们作为全局变量，定义见代码清单 32-7。

#### 代码清单 32-7：结构体指针定义

```
1. DEVICE_INFO *pInformation;
2. DEVICE_PROP *pProperty;
3. USER_STANDARD_REQUESTS *pUser_Standard_Requests;
```

这些结构体指针实际上指向各种用户接口的函数(用户根据不同的应用定义)，我们需要把自定义的接口函数指针赋值给以上几个变量，以供 `usb_core.c` 中的函数(可理解为 USB 内核函数)引用，在那些 USB 内核函数中会进行 USB 协议密切相关的处理。——这是一种回调函数的机制。

先来分析一下 `USB_Init()` 中第 12 行使用的结构体指针，它的类型为 `DEVICE_PROP`，这是一种由 `usb_core.h` 文件中定义的结构体，它的结构体成员大多为指向函数的指针变量，见代码清单 32-8。

#### 代码清单 32-8：DEVICE\_PROP 结构体类型指针

```
3) typedef struct _DEVICE_PROP
4) {
5)     void (*Init)(void);           /* Initialize the device */
6)     void (*Reset)(void);          /* Reset routine of this device */
7)
8)     void (*Process_Status_IN)(void);
9)     void (*Process_Status_OUT)(void);
10)
11)     RESULT (*Class_Data_Setup)(uint8_t RequestNo);
12)     RESULT (*Class_NoData_Setup)(uint8_t RequestNo);
13)
14)     RESULT (*Class_Get_Interface_Setting)(uint8_t Interface, uint8_t AlternateSetting);
15)
16)     uint8_t* (*GetDeviceDescriptor)(uint16_t Length);
17)     uint8_t* (*GetConfigDescriptor)(uint16_t Length);
18)     uint8_t* (*GetStringDescriptor)(uint16_t Length);
19)     void* RxEP_buffer;
20)     uint8_t MaxPacketSize;
21)
22) } DEVICE_PROP;
```

看第一个结构体成员 `void (*Init)(void)`，这是一个指向 USB 设备初始化函数的指针。在应用开始时需要被调用一次，回到 `USB_Init()` 函数中的最后一行：

```
1. pProperty->Init();
```

我们知道 `pProperty` 是一个 `DEVICE_PROP` 类型结构体，它的 `pProperty->Init()` 表示要引用它的结构体成员 `Init`，而这个成员是一个初始化函数指针，也就是说它以这样的方式调用了这个初始化函数，完成了 USB 设备的初始化，但是这个指针指向的函数具体是谁呢？回头看 `USB_Init()` 中第 12 行：

```
55. pProperty = &Device_Property;
```

这个语句使 `pProperty` 指向同类型的结构体变量 `Device_Property`，而这个结构体在 `usb_prop.c` 文件中赋值见代码清单 32-9。

代码清单 32-9：结构体变量 Device\_Property 赋值

```
1. DEVICE_PROP Device_Property =
2. {
3.     MASS_init,
4.     MASS_Reset,
5.     MASS_Status_In,
6.     MASS_Status_Out,
7.     MASS_Data_Setup,
8.     MASS_NoData_Setup,
9.     MASS_Get_Interface_Setting,
10.    MASS_GetDeviceDescriptor,
11.    MASS_GetConfigDescriptor,
12.    MASS_GetStringDescriptor,
13.    0,
14.    0x40 /*MAX PACKET SIZE*/
15. };
```

Device\_Property 的第一个结构体成员即是初始化函数指针，它的函数名为 MASS\_init，具体在 usb\_prop.c 文件定义见代码清单 32-10。

代码清单 32-10：MASS\_init 函数

```
1. void MASS_init()
2. {
3.     /* Update the serial number string descriptor with the data f
4.     rom the unique
5.     ID*/
6.     Get_SerialNum();
7.     pInformation->Current_Configuration = 0;
8.
9.     /* Connect the device */
10.    PowerOn();
11.
12.    /* Perform basic device initialization operations */
13.    USB_SIL_Init();
14.
15.    bDeviceState = UNCONNECTED;
16. }
```

就是这个函数，完成了 USB 的初始化，它调用了 Get\_SerialNum() 获取 USB 字符串描述符、调用 PowerOn() 给 USB 外设上电，调用 USB\_SIL\_Init() 进行基本初始化，提供基本的 USB 设备操作。

在 USB\_Init() 函数中的 pUser\_Standard\_Requests 指针指向关于 USB 标准请求的处理函数；在 DEVICE\_PROP 结构体中也有一些函数用于向主机返回设备描述符、配置描述符的函数，由读者自行分析。



### 32.3.7. 中断服务函数

STM32 USB 控制器的中断分为高优先级中断和低优先级中断，前面已介绍过它们的区别，由于低优先级中断包含控制传输(主要为端点 0)的处理，比较复杂，这里我们使用高优先级中断服务函数进行分析，它在 `stm32f10x_it.c` 文件定义见代码清单 32-11。

代码清单 32-11: USB 中断服务程序

```
3. /*
4.  * 函数名: USB_HP_CAN1_TX_IRQHandler
5.  * 描述   : USB 高优先级中断请求
6.  * 输入   : 无
7.  * 输出   : 无
8.  */
9. void USB_HP_CAN1_TX_IRQHandler(void)
10. {
11.     CTR_HP();
12. }
```

调用的函数 `CTR_HP()`见代码清单 32-12。

代码清单 32-12: `CTR_HP()`函数

```
91. /*****
92. * Function Name   : CTR_HP.
93. * Description     : High Priority Endpoint Correct Transfer interrupt's service
94. *                 routine.
95. * Input           : None.
96. * Output          : None.
97. * Return          : None.
98. *****/
99. void CTR_HP(void)
100. {
101.     uint32_t wEPVal = 0;
102.
103.     while (((wIstr = _GetISTR()) & ISTR_CTR) != 0)
104.     {
105.         _SetISTR((uint16_t)CLR_CTR); /* clear CTR flag */
106.
107.         /* extract highest priority endpoint number */
108.         EPindex = (uint8_t)(wIstr & ISTR_EP_ID);
109.
110.         /* process related endpoint register */
111.
112.         wEPVal = _GetENDPOINT(EPindex);
113.         if ((wEPVal & EP_CTR_RX) != 0)
114.         {
115.             /* clear int flag */
116.             _ClearEP_CTR_RX(EPindex);
117.
118.             /* call OUT service function */
119.             (*pEpInt_OUT[EPindex-1])();
120.
121.             /* if ((wEPVal & EP_CTR_RX) */
```

```

122.         else if ((wEPVal & EP_CTR_TX) != 0)
123.         {
124.             /* clear int flag */
125.             _ClearEP_CTR_TX(EPIndex);
126.
127.             /* call IN service function */
128.             (*pEpInt_IN[EPIndex-1]) ();
129.
130.
131.         } /* if((wEPVal & EP_CTR_TX) != 0) */
132.
133.     } /* while(...) */
134. }

```

分析如下：

- (1) 第 13 行，调用\_GetISTR()函数读取 ISTR 寄存器的内容到变量 wIstr 中。
- (2) 第 15 行，调用\_SetISTR()函数清除正确传输标志 CTR (正确传输会引起 USB 中断)。
- (3) 第 18 行，读取触发中断的端点号。
- (4) 第 22 行，读取相应端点号的端点寄存器内容，它包含了发送或接收信息。
- (5) 第 23 行，判断得端点接收到数据，在第 29 行调用回调函数(\*pEpInt\_OUT[EPIndex-1])()对接收到的数据进行处理。
- (6) 第 32 行，判断得端点发送数据成功，在第 38 行调用回调函数(\*pEpInt\_IN[EPIndex-1])()进行处理。

这两上个回调函数究竟是什么呢？感觉好奇怪，这真的是函数么？它们是在 usb\_istr.c 文件中定义的，见代码清单 32-13。

代码清单 32-13: pEpInt\_IN 和 pEpInt\_OUT 回调函数

```

1. /* function pointers to non-
2. control endpoints service routines */
3. void (*pEpInt_IN[7])(void) =
4. {
5.     EP1_IN_Callback,
6.     EP2_IN_Callback,
7.     EP3_IN_Callback,
8.     EP4_IN_Callback,
9.     EP5_IN_Callback,
10.    EP6_IN_Callback,

```

```
11.     EP7_IN_Callback,  
12. };  
13.  
14. void (*pEpInt_OUT[7])(void) =  
15. {  
16.     EP1_OUT_Callback,  
17.     EP2_OUT_Callback,  
18.     EP3_OUT_Callback,  
19.     EP4_OUT_Callback,  
20.     EP5_OUT_Callback,  
21.     EP6_OUT_Callback,  
22.     EP7_OUT_Callback,  
23. };
```

查看定义我们发现它们是函数指针数组，听这名字就不简单，这样定义是为了方便根据 USB 端点调用不同的端点处理函数。但在本实验中，并没有使用这么多的 USB 端点，以上函数指针数组中，仅有 EP1\_IN\_Callback 和 EP2\_OUT\_Callback 指针是指向实体函数的，分别被中断服务函数调用来处理 USB 主机输入数据 和 USB 主机输出数据 时的情况(函数名中的 IN 和 OUT 是针对 USB 主机来说的)。这两个函数定义见代码清单 32-14。

代码清单 32-14: EP1\_IN\_Callback 和 EP2\_OUT\_Callback 指针

```
3.  
4. void EP1_IN_Callback(void)  
5. {  
6.     Mass_Storage_In();  
7. }  
8.  
9. void EP2_OUT_Callback(void)  
10. {  
11.     Mass_Storage_Out();  
12. }
```

它们分别调用了 Mass\_Storage\_In() 和 Mass\_Storage\_Out() 函数。理解它们虽然先了解 BOT 和 SCSI 协议。

### 32.3.8. BOT 和 SCSI 协议

#### ■ BOT

BOT (Bulk Only Transport)，译为仅批量传输协议，它是 USB 的子类协议。它只使用 USB 传输模式中的批量传输通道进行命令、状态、数据的传输，没有中断和控制通道。建立了传输后(即枚举后)，默认的端点 0 只用于清除批量管道的状态和发送 Mass Storage 复位、Get Max Lun 这两个请求。

在本实验中就只配置了两个端点用于批量传输：端点 1 用于 USB 主机输入类的传输(BulkIn)，端点 2 用于 USB 主机输出类的传输(BulkOut)。

在 BOT 协议中使用 CBW 格式发送命令，用 CSW 格式发送命令状态(详细格式说明请参考 BOT 协议书)。

## ■ SCSI

SCSI 命令集为硬盘、磁带、大容量存储设备提供了高效的点对点操作，目前大多数 USB 存储设备都使用 SCSI 命令集。在 USB 设备中主要用到的 SCSI 命令如表 32-2。

表 32-2 SCSI 命令集

| Command name   | Op Code | Com mand | Description                                                           |
|----------------|---------|----------|-----------------------------------------------------------------------|
| Inquiry        | 0x12    | M        | Get device information                                                |
| Read Format    | 0x23    | M        | Report current media capacity and formattable capacities supported by |
| Mode Sense     | 0x1     | M        | Report parameters to the host                                         |
| Mode Sense     |         | M        | Report parameters to the host                                         |
| Prevent\ Allow | 0x1     | M        | Prevent or allow the removal of media from a removable media device   |
| Read (10)      | 0x28    | M        | Transfer binary data from the medium to the host                      |
| Read Capacity  | 0x25    | M        | Report current medium capacity                                        |
| Request        | 0x03    | O        | Transfer status sense data to the host                                |
| Start Stop     | 0x1     | M        | Enable or disable the Logical Unit for medium access operations and   |
| Test Unit      | 0x00    | M        | Request the device to report if it is                                 |
| Verify (10)    | 0x2     | M        | Verify data on the medium                                             |
| Write (10)     | 0x2     | M        | Transfer binary data from the host to the medium                      |

野火主要给大家讲解的是 Read(10)和 write(10)命令，这两个命令分别用于从媒介(存储器)向主机传输二进制数据 和 主机向媒介传输数据。

## ■ 传输状态机

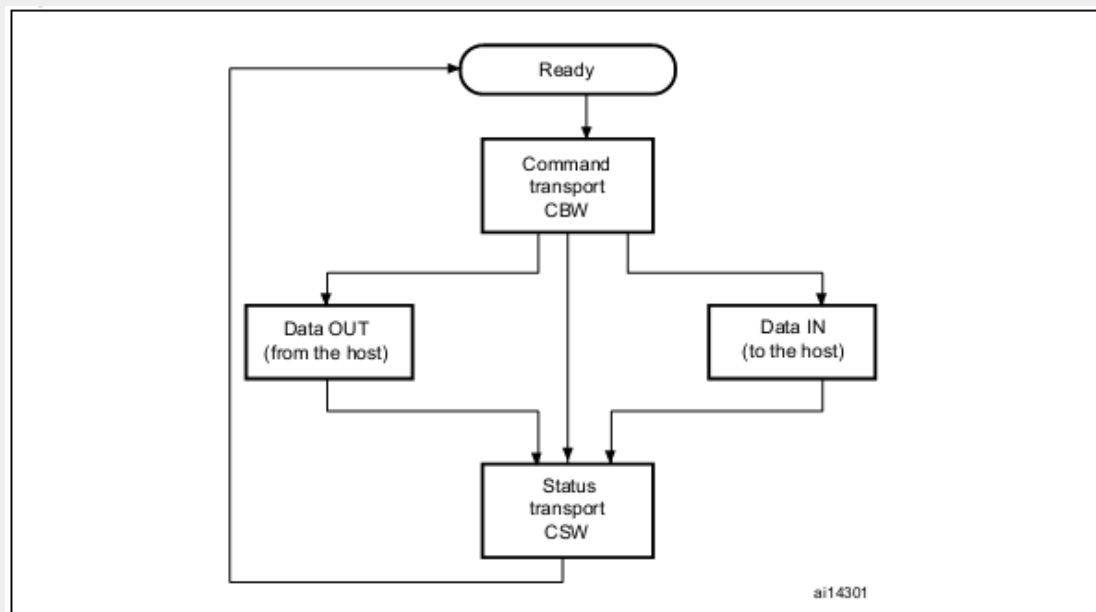


图 32-10 传输状态机

以图 32-10 分析，结合 BOT 协议和 SCSI 协议。当 USB 接口处于准备状态时，主机使用以 BOT 中的 CBW 格式向设备发送包含 write(10)的 SCSI 命令，USB 设备接收到该命令后，使用 CSW 格式向主机说明命令通过，接着主机使用端点 2 BulkOut 传输大量要被写入到 USB 存储器的数据(实际上，发送 CBW 时也是使用端点 2)。当要从 USB 存储器读取数据时的过程类似。

## ■ 代码实现

以端点 2 调用的 Mass\_Storage\_Out() 函数为例，作为 USB 设备，端点 2 是用于接收数据和命令的。见代码清单 32-15。

代码清单 32-15: Mass\_Storage\_Out() 函数

```
1.  /*****
2.  * Function Name   : Mass_Storage_Out
3.  * Description    : Mass Storage OUT transfer.
4.  * Input          : None.
5.  * Output         : None.
6.  * Return         : None.
7.  *****/
8.  void Mass_Storage_Out (void)
9.  {
10.     uint8_t CMD;
```

```

11.  CMD = CBW.CB[0];
12.
13.  Data_Len = USB_SIL_Read(EP2_OUT, Bulk_Data_Buff);
14.
15.  switch (Bot_State)
16.  {
17.      case BOT_IDLE:
18.          CBW_Decode();
19.          break;
20.      case BOT_DATA_OUT:
21.          if (CMD == SCSI_WRITE10)
22.          {
23.              SCSI_Write10_Cmd(CBW.bLUN , SCSI_LBA , SCSI_BlklLen);
24.              break;
25.          }
26.          Bot_Abort(DIR_OUT);
27.          Set_Scsi_Sense_Data(CBW.bLUN, ILLEGAL_REQUEST, INVALID_FI
ELED_IN_COMMAND);
28.          Set_CSW (CSW_PHASE_ERROR, SEND_CSW_DISABLE);
29.          break;
30.      default:
31.          Bot_Abort(BOTH_DIR);
32.          Set_Scsi_Sense_Data(CBW.bLUN, ILLEGAL_REQUEST, INVALID_FI
ELED_IN_COMMAND);
33.          Set_CSW (CSW_PHASE_ERROR, SEND_CSW_DISABLE);
34.          break;
35.  }
36. }

```

函数的第 11 行把接收到的 CBW 格式块中的 CB 成员 赋值给 CMD 变量，CMD 变量中的内容即为接收到的 SCSI 命令编码。在第 21 行，判断得 CMD 内容与宏 SCSI write10 相同时，表明接收到的是写入命令，于是调用 SCSI\_Write10\_Cmd() 函数进行写入数据，它进行一些处理后调用 Write\_Memory() 函数，再调用 MAL\_Write()函数，见代码清单 32-16。

代码清单 32-16: MAL\_Write()函数

```

1.  uint16_t MAL_Write(uint8_t lun, uint32_t Memory_Offset, uint32_
t *Writebuff, uint16_t Transfer_Length)
2.  {
3.
4.      switch (lun)
5.      {
6.          case 0:
7.              Status = SD_WriteBlock((uint8_t*)Writebuff, Memory_Offset, T
ransfer_Length);
8.              Status = SD_WaitWriteOperation();
9.              while(SD_GetStatus() != SD_TRANSFER_OK);
10.             if ( Status != SD_OK )
11.             {
12.                 return MAL_FAIL;
13.             }
14.             break;
15.             default:
16.                 return MAL_FAIL;
17.         }
18.         return MAL_OK;
19.     }

```

我们看到在 `MAL_Write()` 中最终调用了我们熟悉的 `SD_WriteBlock()` 函数，进行 SD 卡的块写入，到这里，我们终于把 USB 通讯与 SD 卡的读写联系起来了。

在本章的讲解中，舍去了大部分细节，这些细节需要读者在亲自实验中去理解、研究。

### 32.3.9. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上 MicroSD 卡，插上方口的 USB 线，将编译好的程序下载到开发板，如果程序运行成功，则可在电脑上看到开发板上的 U 盘(野火用的是 1G 的卡，4G 的也测试通过)，如图 32-11。



图 32-11 模拟 U 盘实验现象

打开 U 盘，可看到里面的文件。还可以进行新建、复制、粘贴、格式化等操作。与操作我们的普通 U 盘没什么区别。





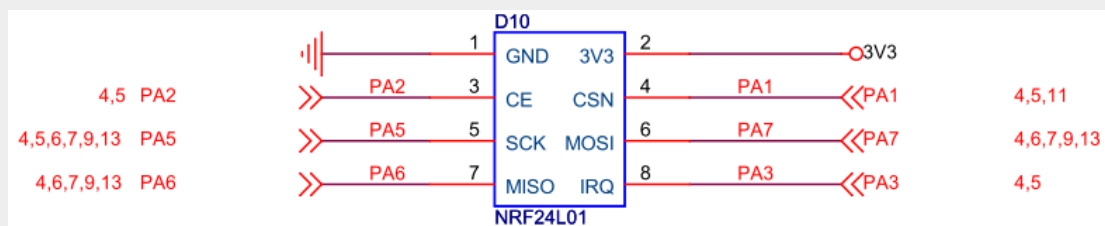
### 33. FM 收音机

## 34. 2.4g 无线传输

### 34.1. 9.1 实验描述及工程文件清单

|         |                                                                                                                                                                                     |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | 利用 NRF24L01+无线模块，使两块 STM32 开发板实现无线传输数据。用串口输出实验结果到 pc。                                                                                                                               |
| 硬件连接    | PA4-SPI1-NSS : W25X16-CS<br>PA5-SPI1-SCK : W25X16-CLK<br>PA6-SPI1-MISO : W25X16-DO<br>PA7-SPI1-MOSI : W25X16-DIO                                                                    |
| 用到的库文件  | startup/start_stm32f10x_hd.c<br>CMSIS/core_cm3.c<br>CMSIS/system_stm32f10x.c<br>FWlib/stm32f10x_gpio.c<br>FWlib/stm32f10x_rcc.c<br>FWlib/stm32f10x_usart.c<br>FWlib/stm32f10x_spi.c |
| 用户编写的文件 | USER/main.c<br>USER/stm32f10x_it.c<br>USER/usart1.c<br>SPI_NRF.c                                                                                                                    |

野火 STM32 开发板 2.4G 无线模块接口图:

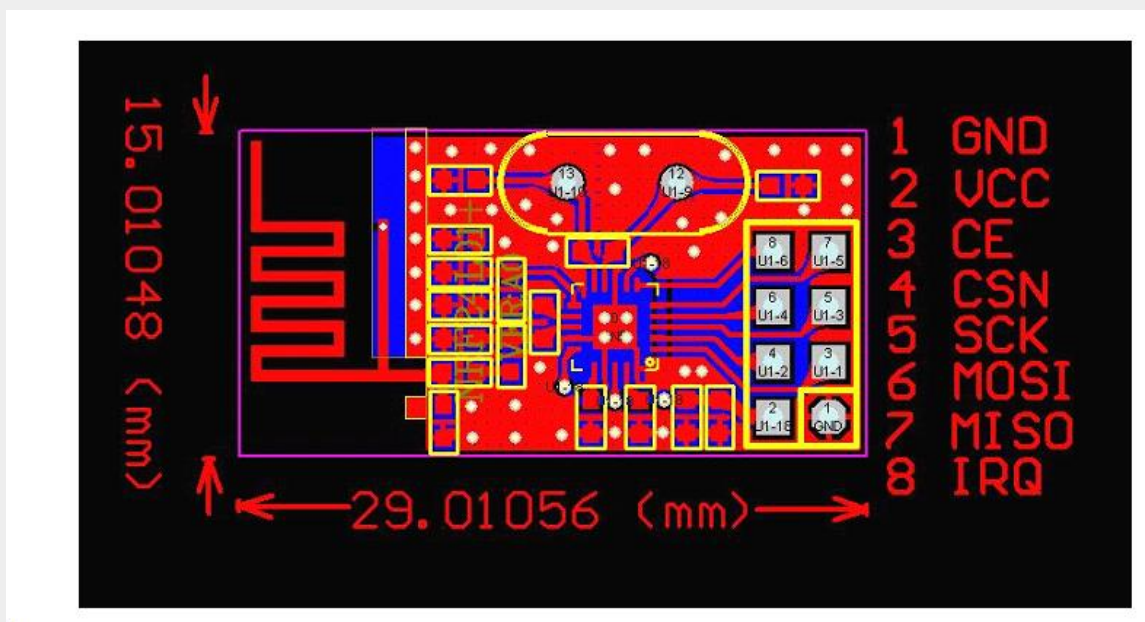


## 34.2. 9.2 NRF24L01 模块简介

本实验采用的无线模块芯片型号为 **NRF24L01+**，是工作在 2.4~2.5GHz 频段的，具备自动重发功能，6 个数据传输通道，最大无线传输速率为 **2Mbits**。MCU 可与该芯片通过 **SPI** 接口访问芯片的寄存器进行配置。

以下是该模块的硬件电路：

截图来自《NRF24L01 模块说明书.pdf》，page3



注意：这个模块的工作电压为 3.3V，实验时请把 vcc 接到板上的 3v3 接口，超过 3.6v 该模块会烧坏！

引脚说明及本实验中与开发板的连接：

| Pin | Name | Description                                  | 与开发板相连      |
|-----|------|----------------------------------------------|-------------|
| 1   | CE   | Chip Enable Activates RX or TX mode          | 排针 P5 的 PA2 |
| 2   | CSN  | SPI Chip Select                              | 排针 P3 的 PA1 |
| 3   | SCK  | SPI Clock                                    | 排针 P5 的 PA5 |
| 4   | MOSI | SPI Slave Data Input                         | 排针 P5 的 PA7 |
| 5   | MISO | SPI Slave Data Output, with tri-state option | 排针 P5 的 PA6 |
| 6   | IRQ  | Maskable interrupt pin. Active low           | 排针 P5 的 PA3 |

|   |     |                                 |           |
|---|-----|---------------------------------|-----------|
| 7 | VDD | Power Supply (+1.9V - +3.6V DC) | 电源 3v3    |
| 8 | VSS | Ground (0V)                     | 电源 GND 接口 |

这个例程采用的是 STM32 的 SPI1 接口，但其中的硬件 SPI1-CSN 端口（用于片选）已经在 2M-FLASH 上采用，所以本实验用一个空闲端口 PA1 用作无线模块的片选，由软件产生片选信号。

请注意区分这个模块的 CSN 片选信号与 CE 使能信号的功能。

CSN 端口是 SPI 通讯协议中的片选端。多个 SPI 设备可以共用 STM32 的 SCK, MISO, MOSI 端口，不同的设备间就是用 CSN 来区分。

CE 实际是 NRF24L01 的芯片使能端，通过配置 CE 可以使 NRF24L01 进入不同的状态。如下图示：

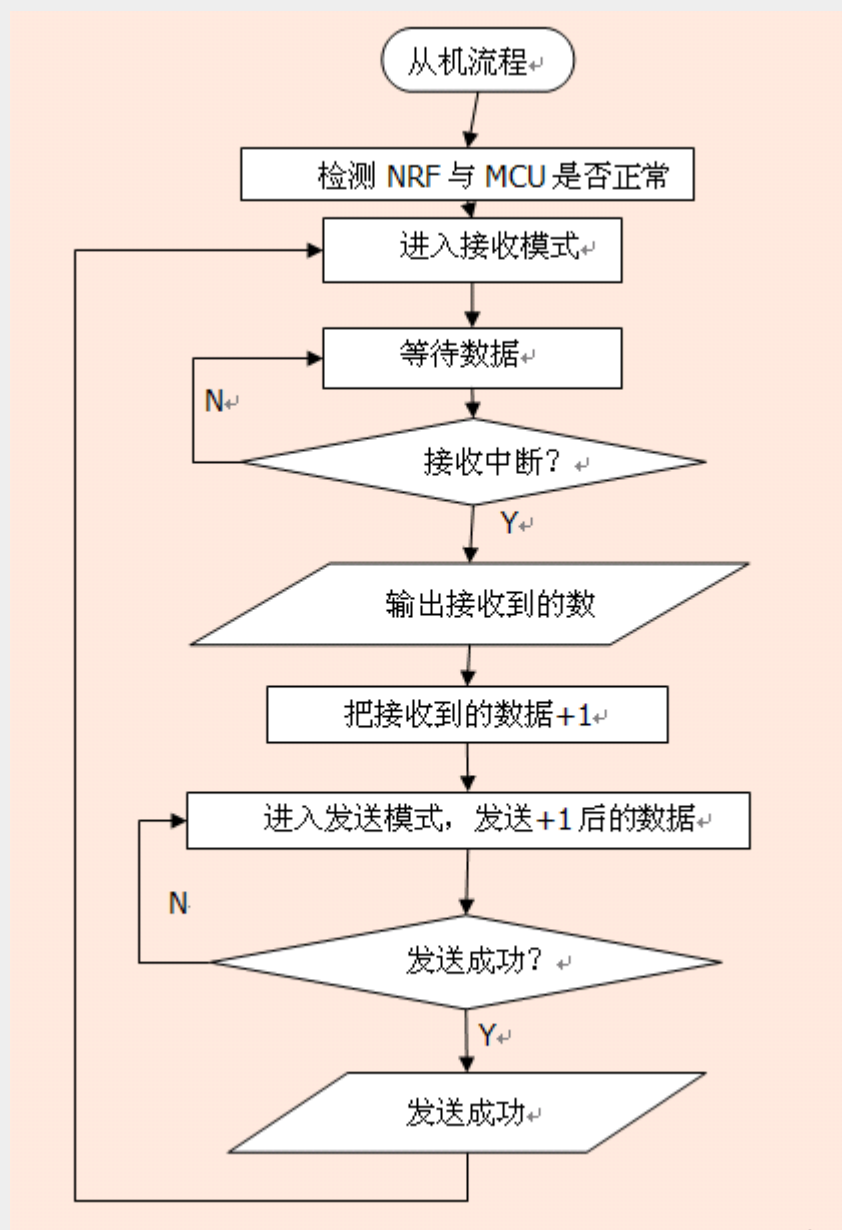
截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page24.

| Mode       | PWR_UP register | PRIM_RX register | CE input pin            | FIFO state                                                         |
|------------|-----------------|------------------|-------------------------|--------------------------------------------------------------------|
| RX mode    | 1               | 1                | 1                       | -                                                                  |
| TX mode    | 1               | 0                | 1                       | Data in TX FIFOs. Will empty all levels in TX FIFOs <sup>a</sup> . |
| TX mode    | 1               | 0                | Minimum 10μs high pulse | Data in TX FIFOs. Will empty one level in TX FIFOs <sup>b</sup> .  |
| Standby-II | 1               | 0                | 1                       | TX FIFO empty.                                                     |
| Standby-I  | 1               | -                | 0                       | No ongoing packet transmission.                                    |
| Power Down | 0               | -                | -                       | -                                                                  |

### 34.3. 9.3 代码分析

这个实验用到两个代码，主机和从机的代码驱动是一样的，区别只是 main 中函数调用的流程不一样，从机接收模式的时候，相应的主机在发送模式。

附从机流程图：



下面以的从机的代码为例进行分析。

首先要添加用的库文件，在工程文件夹下 **Fwlib** 下我们需添加以下库文件：

```

1. stm32f10x_gpio.c
2. stm32f10x_rcc.c
3. stm32f10x_usart.c
4. stm32f10x_spi.c
  
```

还要在 **stm32f10x\_conf.h** 中把相应的头文件添加进来：

```

1. #include "stm32f10x_gpio.h"
2. #include "stm32f10x_spi.h"
  
```

```
3. #include "stm32f10x_rcc.h"
4. #include "stm32f10x_usart.h"
```

进入 main 函数，边看代码边了解程序的流程：

```
1. int main(void)
2. {
3.     /* 串口1 初始化 */
4.     USART1_Config();
5.
6.     /*SPI 接口初始化*/
7.     SPI_NRF_Init();
8.
9.     printf("\r\n 这是一个 NRF24L01 无线传输实验 \r\n");
10.    printf("\r\n 这是无线传输 从机端 的反馈信息\r\n");
11.    printf("\r\n    正在检测 NRF 与 MCU 是否正常连接。。。 \r\n");
12.
13.    /*检测 NRF 模块与 MCU 的连接*/
14.    status = NRF_Check();
15.    if(status == SUCCESS)
16.        printf("\r\n        NRF 与 MCU 连接成功\r\n");
17.    else
18.        printf("\r\n    正在检测 NRF 与 MCU 是否正常连接。。。 \r\n");
19.
20.    while(1)
21.    {
22.        printf("\r\n 从机端 进入接收模式\r\n");
23.        NRF_RX_Mode();
24.
25.        /*等待接收数据*/
26.        status = NRF_Rx_Dat(rxbuf);
27.
28.        /*判断接收状态*/
29.        if(status == RX_DR)
30.        {
31.            for(i=0;i<4;i++)
32.            {
33.                printf("\r\n 从机端 接收到 主机端 发送的数据
为: %d \r\n",rxbuf[i]);
34.                /*把接收的数据+1 后发送给主机*/
35.                rxbuf[i]+=1;
36.                txbuf[i] = rxbuf[i];
37.            }
38.        }
39.        printf("\r\n 从机端 进入自应答发送模式\r\n");
40.        NRF_TX_Mode();
41.
42.        /*不断重发，直至发送成功*/
43.        do
44.        {
45.            status = NRF_Tx_Dat(txbuf);
46.        }while(status == MAX_RT);
47.    }
48. }
```

报告野火，这个代码错了，没有调用 `SystemInit()` 函数来设置时钟！是的，大家熟悉的 `SystemInit()` 函数不见了，但这样并没有出错，原因是这个例程的库是 **3.5 版本**的！在 **3.5 版本**的库中 `SystemInit()` 函数在启动文件

startup\_stm32f10x\_hd.d 中已用汇编语句调用了, 设置的时钟为默认的 72M。

所以在 main 函数就不需要再调用啦, 当然, 再调用一次也是没问题的。

关于 USART1\_Config() 函数, 是用来配置串口的, 关于这两个函数的具体讲解可以参考前面的教程, 这里不再详述。

接着进入 SPI\_NRF\_Init() 函数是怎样配置 STM32 的 SPI 接口的:

```
1. void SPI_NRF_Init(void)
2. {
3.     SPI_InitTypeDef SPI_InitStructure;
4.     GPIO_InitTypeDef GPIO_InitStructure;
5.
6.     /*使能 GPIOB, GPIOD, 复用功能时钟*/
7.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOE|RCC_APB2Periph_AFIO, ENABLE);
8.
9.     /*使能 SPI1 时钟*/
10.    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
11.
12.    /*配置 SPI_NRF_SPI 的 SCK, MISO, MOSI 引脚, GPIOA^5, GPIOA^6, GPIOA^7 */
13.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
14.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
15.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用功能
16.    GPIO_Init(GPIOA, &GPIO_InitStructure);
17.
18.    /*配置 SPI_NRF_SPI 的 CE 引脚, GPIOA^2 和 SPI_NRF_SPI 的 CSN 引脚: NSS GPIOA^1*/
19.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_1;
20.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
21.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
22.    GPIO_Init(GPIOA, &GPIO_InitStructure);
23.
24.    /*配置 SPI_NRF_SPI 的 IRQ 引脚, GPIOA^3*/
25.    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
26.    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
27.    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //上拉输入
28.    GPIO_Init(GPIOA, &GPIO_InitStructure);
29.
30.    /* 这是自定义的宏, 用于拉高 csn 引脚, NRF 进入空闲状态 */
31.    NRF_CSN_HIGH();
32.
33.    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //
    /*双线全双工
34.    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //
    主模式
35.    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //
    数据大小 8 位
36.    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //
    时钟极性, 空闲时为低
37.    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge; //
    第 1 个边沿有效, 上升沿为采样时刻
38.    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //
    NSS 信号由软件产生
39.    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
    //8 分频, 9MHz
40.    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //
    高位在前
```

```
41. SPI_InitStructure.SPI_CRCPolynomial = 7;
42. SPI_Init(SPI1, &SPI_InitStructure);
43.
44. /* Enable SPI1 */
45. SPI_Cmd(SPI1, ENABLE);
46. }
```

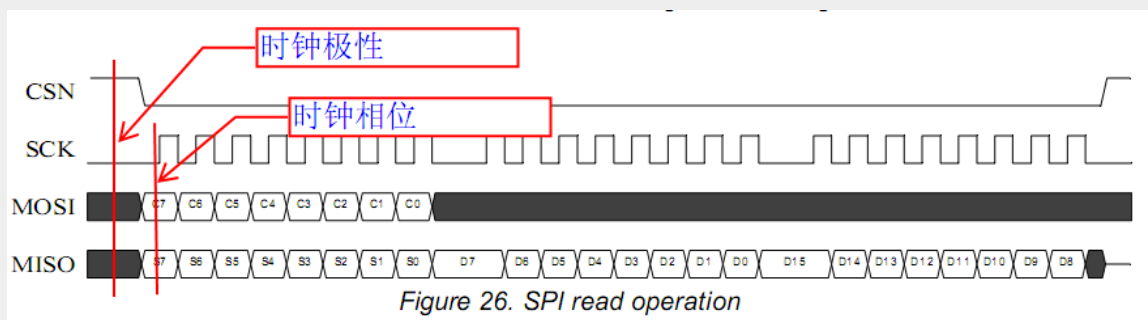
基本流程就是：

1. 开启所用到的端口的时钟 **GPIOA**，复用时钟 **AFIO**，**SPI1** 时钟。  
其中 **AFIO** 时钟是因为 **GPIOA** 是用作 **SPI** 方式的，所以别忘记开启它。
2. 配置 **SPI** 的模式，配置 **SPI** 模式的关键是时钟极性（**CPOL**）第 36 行，和时钟相位（**CPHA**）第 37 行。这是根据 **NRF24L01** 的时序图来设置的。

所谓**时钟极性**，就是 **SPI** 端口在空闲的时候，**SCK** 的电平，例程中为低电平。

而**时钟相位**，则是 **SPI** 采集数据的时钟边沿，例程中为第一个时钟边沿(上升沿)。

截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page52



继续分析 **main** 函数，在第 14 行调用了 **NRF\_Check()** 函数，分析一下它：

```
12. u8 NRF_Check(void)
13. {
14.     u8 buf[5]={0xC2,0xC2,0xC2,0xC2,0xC2};
15.     u8 buf1[5];
16.     u8 i;
17.
18.     /*写入 5 个字节的地址. */
19.     SPI_NRF_WriteBuf(NRF_WRITE_REG+TX_ADDR,buf,5);
20.
21.     /*读出写入的地址 */
22.     SPI_NRF_ReadBuf(TX_ADDR,buf1,5);
23.
24.     /*比较*/
```



```

25.     for(i=0;i<5;i++)
26.     {
27.         if(buf1[i]!=0xC2)
28.             break;
29.     }
30.
31.     if(i==5)
32.         return SUCCESS ;           //MCU 与 NRF 成功连接
33.     else
34.         return ERROR ;             //MCU 与 NRF 不正常连接
35. }

```

这个函数是通过调用 `SPI_NRF_WriteBuf()` 和 `SPI_NRF_ReadBuf()` 函数，对 NRF 的地址寄存器进行读写，先向寄存器写入数据，再读取出来进行比较，以此来检验 NRF24L01 是否与 MCU 正常连接的。

`SPI_NRF_WriteBuf()` 和 `SPI_NRF_ReadBuf()` 函数很类似，前者实现向寄存器写入一串数据，后者实现读取数据功能。下面以 `SPI_NRF_WriteBuf()` 为例：

```

14. u8 SPI_NRF_WriteBuf(u8 reg ,u8 *pBuf,u8 bytes)
15. {
16.     u8 status,byte_cnt;
17.     NRF_CE_LOW();
18.     /*置低 CSN, 使能 SPI 传输*/
19.     NRF_CSN_LOW();
20.
21.     /*发送寄存器号*/
22.     status = SPI_NRF_RW(reg);
23.
24.     /*向缓冲区写入数据*/
25.     for(byte_cnt=0;byte_cnt<bytes;byte_cnt++)
26.         SPI_NRF_RW(*pBuf++);      //写数据到缓冲区
27.
28.     /*CSN 拉高, 完成*/
29.     NRF_CSN_HIGH();
30.
31.     return (status);              //返回 NRF24L01 的状态
32. }

```

`SPI_NRF_WriteBuf()` 调用了 `SPI_NRF_RW()` 函数：

```

1. u8 SPI_NRF_RW(u8 dat)
2. {
3.     /* 当 SPI 发送缓冲器非空时等待 */
4.     while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
5.
6.     /* 通过 SPI2 发送一字节数据 */
7.     SPI_I2S_SendData(SPI1, dat);
8.
9.     /* 当 SPI 接收缓冲器为空时等待 */
10.    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
11.
12.    /* Return the byte read from the SPI bus */
13.    return SPI_I2S_ReceiveData(SPI1);
14. }

```

SPI\_NRF\_WriteBuf()流程:

1. 使 **CE** 端口置低, **SPI** 通讯时先进入**待机模式**。使 **CSN** 端口置低, 开启 **SPI** 通讯。
2. 调用 `SPI_NRF_RW()` 向 **NRF** 发送将要写入的**寄存器地址**, `SPI_NRF_RW()` 返回的是 **STATUS** 寄存器的数据 (不是将要操作的寄存器的值哦!)
3. 连续写入数据, 最后拉高 **CSN** 端口, 结束 **SPI** 传输。

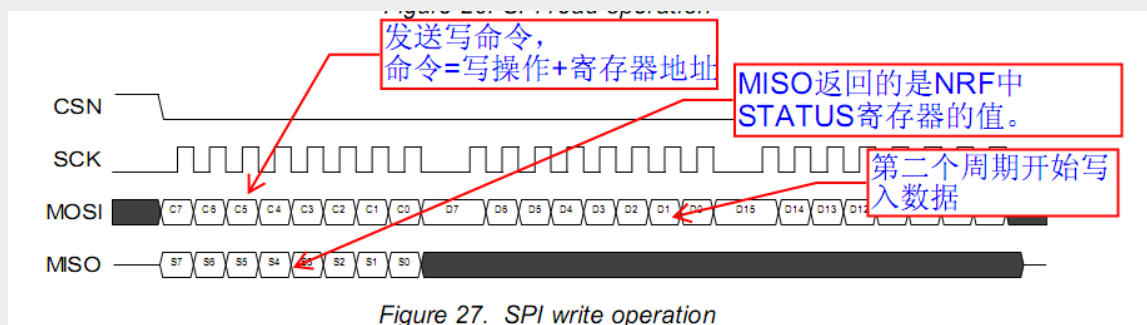
命令的组织形式: 完整的写命令 = 写命令+寄存器地址。

截图来自: 《nRF24L01P(新版无线模块控制 IC).PDF》, page51

| Command name | Command word (binary) | # Data bytes           | Operation                                                                                                                 |
|--------------|-----------------------|------------------------|---------------------------------------------------------------------------------------------------------------------------|
| R_REGISTER   | 000A AAAA             | 1 to 5<br>LSByte first | Read command and status registers. AAAAA = 5 bit Register Map Address                                                     |
| W_REGISTER   | 001A AAAA             | 1 to 5<br>LSByte first | Write command and status registers. AAAAA = 5 bit Register Map Address<br>Executable in power down or standby modes only. |

写时序:

截图来自: 《nRF24L01P(新版无线模块控制 IC).PDF》, page52



回到 **main** 函数, 检查完 **NRF** 与 **MCU** 的连接, 从机开始进入接收模

式 `NRF_RX_Mode()` 函数:

```
(5) void NRF_RX_Mode(void)
(6)
(7) {
(8)     NRF_CE_LOW();
(9)
(10)     SPI_NRF_WriteBuf(NRF_WRITE_REG+RX_ADDR_P0, RX_ADDRESS, RX_ADDR_
        WIDTH); //写 RX 节点地址
(11)
```

```
(12)      SPI_NRF_WriteReg(NRF_WRITE_REG+EN_AA,0x01);    //使能通道 0 的
    自动应答
(13)
(14)      SPI_NRF_WriteReg(NRF_WRITE_REG+EN_RXADDR,0x01); //使能通道 0 的
    接收地址
(15)
(16)      SPI_NRF_WriteReg(NRF_WRITE_REG+RF_CH,CHANAL);    //设置 RF
    通信频率
(17)
(18)      SPI_NRF_WriteReg(NRF_WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH); //选
    择通道 0 的有效数据宽度
(19)
(20)      SPI_NRF_WriteReg(NRF_WRITE_REG+RF_SETUP,0x0f); //设置 TX 发射参
    数,0db 增益,2Mbps,低噪声增益开启
(21)
(22)      SPI_NRF_WriteReg(NRF_WRITE_REG+CONFIG, 0x0f); //配置基本工作
    模式的参数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
(23)
(24)      /*CE 拉高, 进入接收模式*/
(25)      NRF_CE_HIGH();
(26)      }
```

配置接收模式和发送模式都是向 NRF 寄存器写入配置参数, 这些参数具体意义在

《nRF24L01P(新版无线模块控制 IC).PDF》, page57

要注意的是从机和主机的参数要一致。

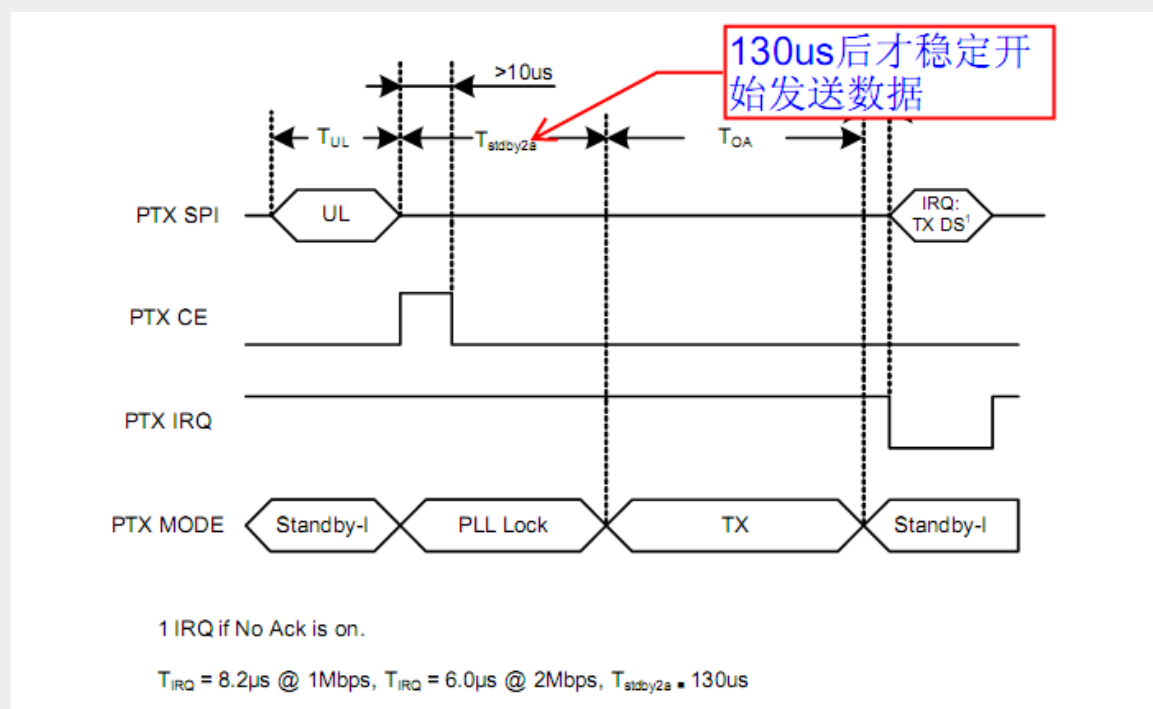
下面是发送模式的配置:

```
8. void NRF_TX_Mode(void)
9. {
10.     NRF_CE_LOW();
11.
12.     SPI_NRF_WriteBuf(NRF_WRITE_REG+TX_ADDR,TX_ADDRESS,TX_ADR_WIDTH);
    //写 TX 节点地址
13.
14.     SPI_NRF_WriteBuf(NRF_WRITE_REG+RX_ADDR_P0,RX_ADDRESS,RX_ADR_WIDTH);
    //设置 TX 节点地址, 主要为了使能 ACK
15.
16.     SPI_NRF_WriteReg(NRF_WRITE_REG+EN_AA,0x01);    //使能通道 0 的自动应
    答
17.
18.     SPI_NRF_WriteReg(NRF_WRITE_REG+EN_RXADDR,0x01); //使能通道 0 的接收地
    址
19.
20.     SPI_NRF_WriteReg(NRF_WRITE_REG+SETUP_RETR,0x1a); //设置自动重发间隔时
    间:500us + 86us;最大自动重发次数:10 次
21.
22.     SPI_NRF_WriteReg(NRF_WRITE_REG+RF_CH,CHANAL);    //设置 RF 通道为
    CHANAL
23.
24.     SPI_NRF_WriteReg(NRF_WRITE_REG+RF_SETUP,0x0f); //设置 TX 发射参数,0db
    增益,2Mbps,低噪声增益开启
25. }
```

```
26. SPI_NRF_WriteReg(NRF_WRITE_REG+CONFIG, 0x0e); //配置基本工作模式的参
    数;PWR_UP, EN_CRC, 16BIT_CRC, 发射模式, 开启所有中断
27.
28. /*CE 拉高, 进入发送模式*/
29. NRF_CE_HIGH();
30. Delay(0xffff); //CE 要拉高一段时间才进入发送模式
31. }
```

在发送模式配置要特别注意一点，第 23 行，延时，STM32 运行频率比 NRF 模块快得多，不加延时直接发送数据的话很容易导致发送出错。

截图来自：《nRF24L01P(新版无线模块控制 IC).PDF》，page42



回到 main 函数，配置完接收模式后就开始等待 NRF 模块传来中断接收数据。

由 NRF\_Rx\_Dat() 函数来实现：

```
3. u8 NRF_Rx_Dat(u8 *rxbuf)
4. {
5.     u8 state;
6.     NRF_CE_HIGH(); //进入接收状态
7.     /*等待接收中断*/
8.     while(NRF_Read_IRQ() != 0);
9.
10.    NRF_CE_LOW(); //进入待机状态
11.    /*读取 status 寄存器的值 */
12.    state = SPI_NRF_ReadReg(STATUS);
13.
14.    /* 清除中断标志 */
15.    SPI_NRF_WriteReg(NRF_WRITE_REG+STATUS, state);
16.
17.    /*判断是否接收到数据*/
18.    if(state & RX_DR) //接收到数据
```

```

19.     {
20.         SPI_NRF_ReadBuf(RD_RX_PLOAD, rxbuf, RX_PLOAD_WIDTH); //读取数据
21.         SPI_NRF_WriteReg(FLUSH_RX, NOP); //清除 RX FIFO 寄存
器
22.         return RX_DR;
23.     }
24.     else
25.         return ERROR; //没收到任何数据
26. }

```

接收流程:

1. 等待 IRQ 引脚的信号，NRF 模块在接收到数据，发送完成数据，或重发超过次数都会在 IRQ 引脚进行标志。（这个实验中采用的是 STM32 循环读取 IRQ 引脚信号，实际上可以把这个引脚配置成外部中断来减轻 MCU 的负担，读者可以一试。）

关于 NRF 整个无线传输过程可以参照《nRF24L01P(新版无线模块控制 IC).PDF》的 page36~37 页的流程图。

2. 读取状态寄存器的值来判断是否接收正常，利用 SPI\_NRF\_ReadBuf() 来从接收缓冲区读取数据到 STM32。
3. 清中断（通过向 STATUS 寄存器写 1 可以清除相应的中断位），清空接收缓冲区。

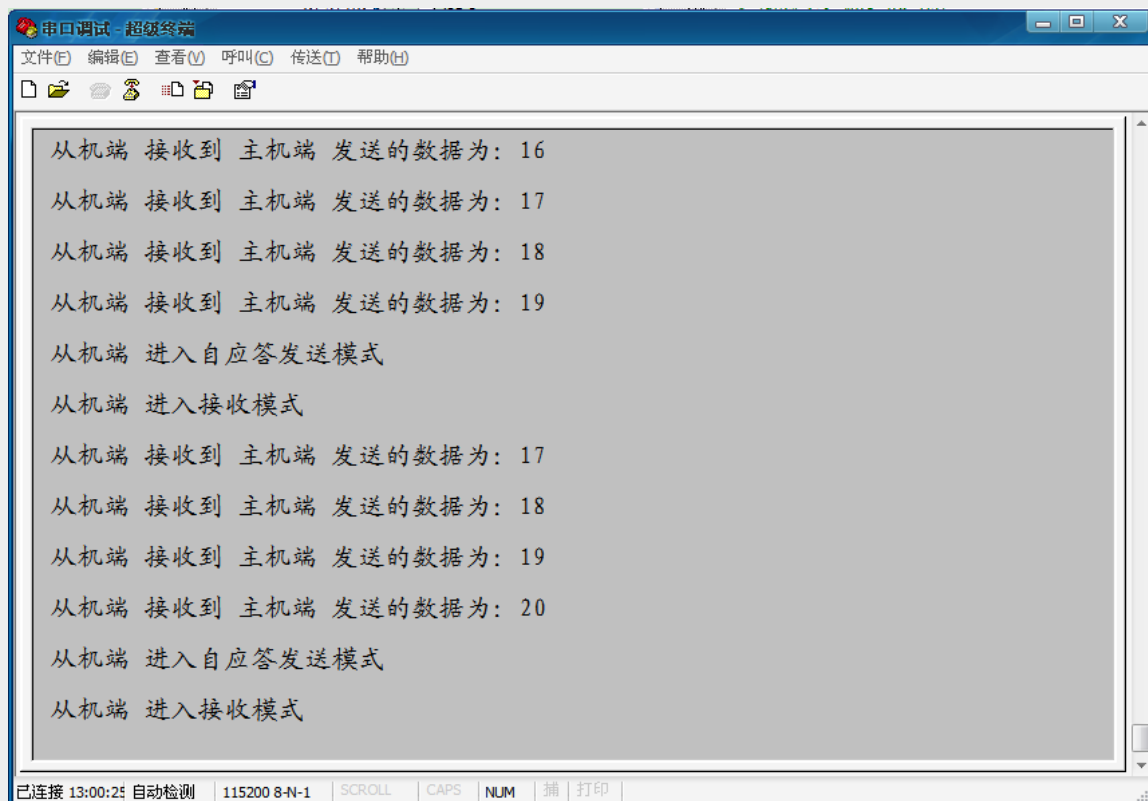
到这里就把所有的 NRF 驱动应用函数解说完啦! ^\_^

回到 main 函数，从机把接收到的数据都加 1 之后再发送给主机，主机又把数据发送回从机。。。如此循环，就是整个实验的流程。

#### 34.4. 9.4 实验想象

实验时请先开启从机的电源，再开启主机的电源，这是代码的流程决定的。

### 实验效果：从机的反馈：



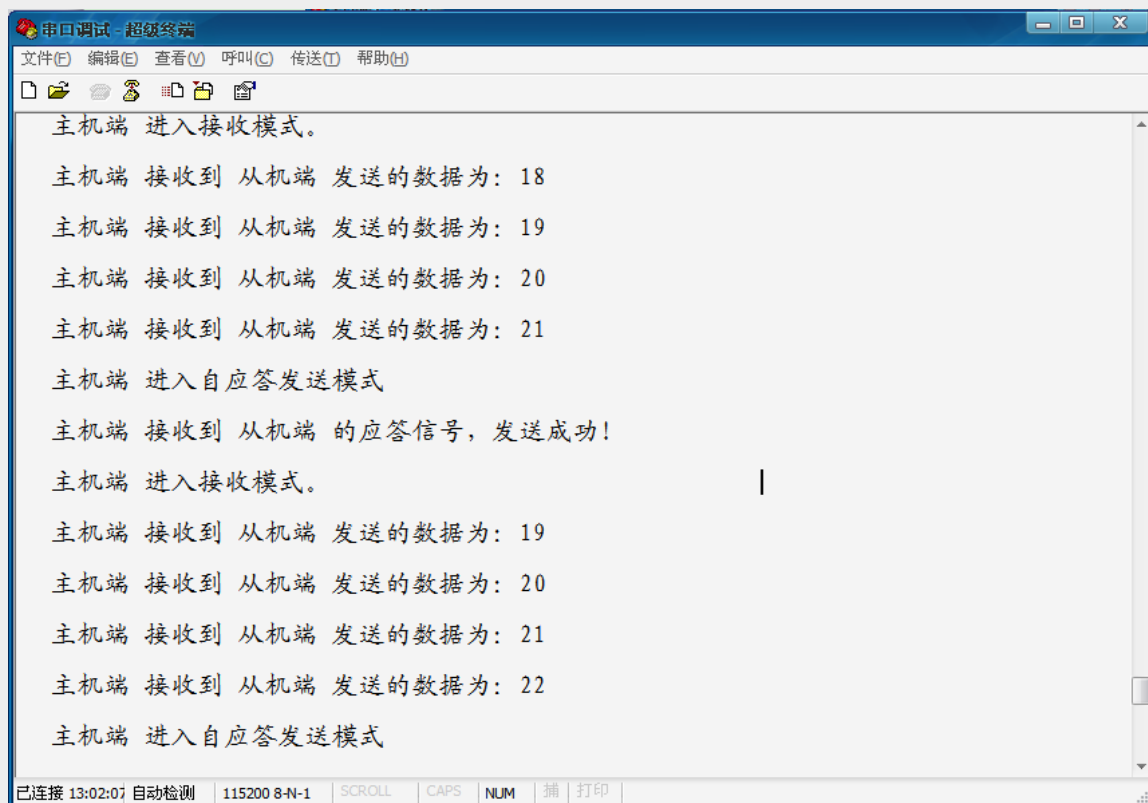
串口调试 - 超级终端

文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

从机端 接收到 主机端 发送的数据为：16  
 从机端 接收到 主机端 发送的数据为：17  
 从机端 接收到 主机端 发送的数据为：18  
 从机端 接收到 主机端 发送的数据为：19  
 从机端 进入自应答发送模式  
 从机端 进入接收模式  
 从机端 接收到 主机端 发送的数据为：17  
 从机端 接收到 主机端 发送的数据为：18  
 从机端 接收到 主机端 发送的数据为：19  
 从机端 接收到 主机端 发送的数据为：20  
 从机端 进入自应答发送模式  
 从机端 进入接收模式

已连接 13:00:25 自动检测 | 115200 8-N-1 | SCROLL | CAPS | NUM | 捕 | 打印

### 主机的反馈：



串口调试 - 超级终端

文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

主机端 进入接收模式。  
 主机端 接收到 从机端 发送的数据为：18  
 主机端 接收到 从机端 发送的数据为：19  
 主机端 接收到 从机端 发送的数据为：20  
 主机端 接收到 从机端 发送的数据为：21  
 主机端 进入自应答发送模式  
 主机端 接收到 从机端 的应答信号，发送成功！  
 主机端 进入接收模式。  
 主机端 接收到 从机端 发送的数据为：19  
 主机端 接收到 从机端 发送的数据为：20  
 主机端 接收到 从机端 发送的数据为：21  
 主机端 接收到 从机端 发送的数据为：22  
 主机端 进入自应答发送模式

已连接 13:02:07 自动检测 | 115200 8-N-1 | SCROLL | CAPS | NUM | 捕 | 打印



## 35. 重力感应/三轴加速 (MMA7455)

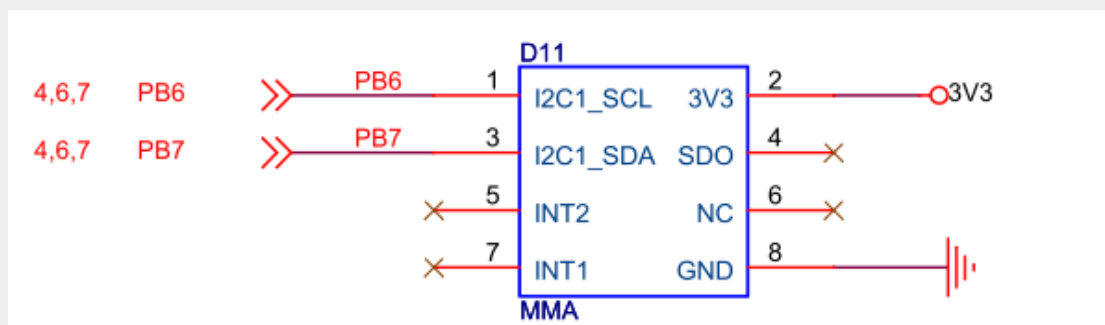
## 35.1. 10.1 实验描述及工程文件清单

|         |                                                                                                                                                                                     |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 实验描述    | 控制倾角传感器 (MMA7455) 测量加速度和倾角，通过串口打印测量得的数据。                                                                                                                                            |
| 硬件连接    | PB6-I2C1_SCL<br>PB7-I2C1_SDA                                                                                                                                                        |
| 用到的库文件  | startup/start_stm32f10x_hd.c<br>CMSIS/core_cm3.c<br>CMSIS/system_stm32f10x.c<br>FWlib/stm32f10x_gpio.c<br>FWlib/stm32f10x_rcc.c<br>FWlib/stm32f10x_usart.c<br>FWlib/stm32f10x_i2c.c |
| 用户编写的文件 | USER/main.c<br>USER/stm32f10x_it.c<br>USER/usart1.c<br>USER/I2C_MMA.c                                                                                                               |

在这个加速度传感器例程中野火采用使用了集成模块，只需要把模块的电源线和 SDA，SCL 端口连接到开发板上的 I2C 总线即可。EEPROM 的也使用了 I2C 总线，所以可以参照 EEPROM 的件原理图。



野火 STM32 开发板 2.4G 无线模块接口图:



### 35.2. 10.2 MMA7455 简介

MMA7455 是一款数字三轴加速度传感器，可以利用测量出来的重力加速度在某方向上的分量来计算器件与水平面间的夹角，测量倾角是很多有趣应用的基础。

本实验采用 I2C 方式与传感器通讯，控制传感器用 2g 的量程。

在测量前对传感器进行校准。

### 35.3. 10.3 代码分析

首先要添加用的库文件，在工程文件夹下 Fwlib 下我们需添加以下库文件：

```
1. stm32f10x_gpio.c
2. stm32f10x_rcc.c
3. stm32f10x_usart.c
4. stm32f10x_i2c.c
```

还要在 `stm32f10x_conf.h` 中把相应的头文件添加进来：

```
1. #include "stm32f10x_gpio.h"
2. #include "stm32f10x_i2c.h"
3. #include "stm32f10x_rcc.h"
4. #include "stm32f10x_usart.h"
```

配置好所需的库文件之后，我们就从 `main` 函数开始分析：

```
1. *
```

```
2.  * 函数名: main
3.  * 描述   : 主函数
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 返回   : 无
7.  */
8.  int main(void)
9.  {
10.     /* 配置系统时钟为 72M */
11.     SystemInit();
12.
13.     /* 串口1初始化 */
14.     USART1_Config();
15.
16.     /*重力传感器初始化*/
17.     I2C_MMA_Init();
18.
19.     /*重力传感器校准*/
20.     I2C_MMA_Cal();
21.
22.     printf("\r\n-----这是一个重力传感器测试程序-----\r\n");
23.
24.
25.     /* 检测倾角*/
26.     I2C_MMA_Test(&X_Value);
27.     I2C_MMA_Test(&Y_Value);
28.     I2C_MMA_Test(&Z_Value);
29.
30.     printf("\r\n-----X 方向的数据-----\r\n");
31.     I2C_MMA_Printf(&X_Value);
32.
33.     printf("\r\n-----Y 方向的数据-----\r\n");
34.     I2C_MMA_Printf(&Y_Value);
35.
36.     printf("\r\n-----Z 方向的数据-----\r\n");
37.     I2C_MMA_Printf(&Z_Value);
38.
39.     /*进入省电模式*/
40.     if(I2C_MMA_Standby() == SUCCESS )
41.
42.         printf("\r\n Acceleration enter standby mode! \r\n");
43.     else
44.         printf("\r\n Standby mode ERROR! \r\n");
45.
46. }
```

系统库函数 `SystemInit()`；将系统时钟设置为 72M，`USART1_Config()`；配置串口，关于这两个函数的具体讲解可以参考前面的教程，这里不再详述。

```
1.  /*
2.  * 函数名: I2C_MMA_Init
3.  * 描述   : I2C 外设(MMA7455)初始化
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8.  void I2C_MMA_Init(void)
9.  {
10.     I2C_GPIO_Config();
11.     I2C_Mode_Config();
12. }
```

`I2C_MMA_Init()`；是用户编写的函数，其中调用了 `I2C_GPIO_Config()`；配置好 I2C 所用的 I/O 端口，调用 `I2C_Mode_Configu()`；设置 I2C 的工作模式，并使能相关外设的时钟。

`I2C_MMA_Cal()` 函数是用来校正传感器的，校正需要先测量原始数据，我们先看

`I2C_MMA_Test()` 函数

```
1. /*
2.  * 函数名: I2C_MMA_Test
3.  * 描述   : 测量倾角 和 加速度 (量程 0-2g)
4.  * 输入   : 数据结构体的指针
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void I2C_MMA_Test(MMA_Dat* MMA_Value)
9. {
10.     u8 temp;
11.
12.     /*MMA 进入 2g 量程测试模式*/
13.     I2C_MMA_ByteWrite(0x05,MMA_MCTL_Addr);
14.
15.     /*DRDY 标置位,等待测试完毕*/
16.     while(!(I2C_MMA_ByteRead(MMA_STATUS_Addr)&0x01));
17.
18.     /*读取测得的数据*/
19.     MMA_Value->Out = I2C_MMA_ByteRead(MMA_Value->Addr);
20.
21.     if((MMA_Value->Out&0x80) ==0x00) /*读出的原始值为正数 */
22.     {
23.         temp = MMA_Value->Out;
24.
25.         /*将原始值转换为加速度,乘以 -1 为方向处理*/
26.         MMA_Value->Acc = (float) (-1)*temp *ACC_Gravity/64;
27.
28.         /*将原始值转换为角度*/
29.         if(temp >=64)
30.             /*加速度值大于 1g */
31.             MMA_Value->Angle = 90.0;
32.
33.         else
34.             /*加速度小于 1 g, Angle = asin(Acc/9.8)*57.32; 弧度制转换
35.             57.32 = 180/3.14*/
36.             MMA_Value->Angle = asin((float)temp/64)*57.32;
37.
38.         /*读出的原始值为负数 */
39.     else
40.     {
41.         temp = MMA_Value->Out;
42.
43.         /*二补码转换*/
44.         temp -= 1;
45.         temp = ~temp;
46.
47.         /*将原始值转换为加速度*/
48.         MMA_Value->Acc = (float) temp *ACC_Gravity/64;
49.
50.         /*将原始值转换为角度,乘以 -1 为方向处理*/
51.         if(temp>=64)
52.             MMA_Value->Angle = -90.0;
53.     else
```

```
54.         /* Angle = asin(Acc/9.8)*57.32 */
55.         MMA_Value->Angle = (-1)*asin((float)temp/64)*57.32;
56.     }
57. }
```

这个函数调用了 `I2C_MMA_ByteWrite()`，以下是函数原型：

```
1.  /*
2.   * 函数名: I2C_MMA_ByteWrite
3.   * 描述   : 写一个字节到 I2C MMA 寄存器中
4.   * 输入   : -pBuffer 缓冲区指针
5.   *         -WriteAddr 接收数据的 MMA 寄存器的地址
6.   * 输出   : 无
7.   * 返回   : 无
8.   * 调用   : 内部调用
9.   */
10. static void I2C_MMA_ByteWrite(u8 pBuffer, u8 WriteAddr)
11. {
12.     /*wait until I2C bus is not busy*/
13.     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
14.
15.     /* Send START condition */
16.     I2C_GenerateSTART(I2C1, ENABLE);
17.
18.     /* Test on EV5 and clear it */
19.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
20.
21.     /* Send MMA address for write */
22.     I2C_Send7bitAddress(I2C1, MMA_ADRESS, I2C_Direction_Transmitter);
23.
24.     /* Test on EV6 and clear it */
25.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECT
26. ED));
27.
28.     /* Send the MMA's Register address to write to */
29.     I2C_SendData(I2C1, WriteAddr);
30.
31.     /* Test on EV8 and clear it */
32.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
33.
34.     /* Send the byte to be written */
35.     I2C_SendData(I2C1, pBuffer);
36.
37.     /* Test on EV8 and clear it */
38.     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));
39.
40.     /* Send STOP condition */
41.     I2C_GenerateSTOP(I2C1, ENABLE);
42. }
```

这与 EEPROM 的 `I2C_EE_PageWrite()` 函数很类似，第一个参数为要写入的数值，第二个参数为待写入寄存器的地址。这些寄存器的地址在 `I2C_MMA.h` 文件中都有宏定义。

其中要强调的地方是 MMA7455 的器件地址（分清器件地址与寄存器地址的区别哦！），在 MMA7455 的 DataSheet 中查到的器件地址为 `0X1D`，但这个地址是不正确的，与野火使用的这块芯片有区别，市面上的传感器一般这个器件地址也为 `0x3A`。

在用 Jlink 调试程序的时候，程序循环运行在第 25 行就可以知道是器件地址出了问题，其它通讯的错误也可以用这样的方式查找出来，省了示波器 ^\_^。

I2C\_MMA\_ByteRead() 类似，功能是读取寄存器的值，就不分析了。

回到 I2C\_MMA\_Test() 函数中，I2C\_MMA\_Test() 的第 13 行代码是向 MMA7455 写入命令，开启 2g 转换模式，开启了检测模式后，通过查询 MMA7455 的 DRDY 位，等待 MMA7455 的加速度转化完成。以下为 DataSheet 的说明，附上野火的注释：

截图来自《MMA7455L》

\$16: Mode Control Register (Read/Write)

| D7 | D6   | D5    | D4   | D3      | D2      | D1      | D0      | Bit      |
|----|------|-------|------|---------|---------|---------|---------|----------|
| -- | DRPD | SPI3W | STON | GLVL[1] | GLVL[0] | MODE[1] | MODE[0] | Function |
| 0  | 0    | 0     | 0    | 0       | 0       | 0       | 0       | Default  |

Table 5. Configuring the g-Select for 8-bit output using Register \$16 with GLVL[1:0] bits

| GLVL [1:0] | g-Range | Sensitivity |
|------------|---------|-------------|
| 00         | 8g      | 16 LSB/g    |
| 01         | 2g      | 64 LSB/g    |
| 10         | 4g      | 32 LSB/g    |

2g模式最小量化单位

#### Standby Mode

This digital output 3-axis accelerometer provides a standby mode that is ideal for battery operated products. When standby mode is active, the device outputs are turned off, providing significant reduction of operating current. When the device is in standby mode the current will be reduced to 2.5  $\mu$ A typical. In standby mode the device can read and write to the registers with the I<sup>2</sup>C/ SPI available, but no new measurements can be taken in this mode as all current consuming parts are off. The mode of the device is controlled through the mode control register by accessing the two mode bits as shown in Table 6.

Table 6. Configuring the Mode using Register \$16 with MODE[1:0] bits

| MODE [1:0] | Function             |
|------------|----------------------|
| 00         | Standby Mode         |
| 01         | Measurement Mode     |
| 10         | Level Detection Mode |
| 11         | Pulse Detection Mode |

使用检测模式

#### Measurement Mode

The device can read XYZ measurements in this mode. The pulse and threshold interrupt measurement data and the g-range of 8g is selectable with 10-bit data. The sample rate during measurement mode is 250 Hz with the 125 Hz filter selected. Therefore, when a conversion is complete (signaled by the DRDY flag), the next measurement will be ready.

读取数据前检查 DRDY寄存器

When measurements on all three axes are completed, a logic high level is output to the DRDY pin, indicating "measurement data is ready." The DRDY status can be monitored by the DRDY bit in Status Register (Address: \$09). The DRDY pin is kept high until one of the three Output Value Registers are read. If the next measurement data is written before the previous data is read, the DOVR bit in the Status Register will be set. Also note that in measurement mode, level detection mode and pulse detection mode are not available.

转化完成后根据传入的参数，把相应的 MMA7455 数据寄存器中的数据读取出来，各个方向的原始数据储存在相应的结构体的 .Out 变量中：

```
1. typedef struct
2. {
3.     uc8 Addr;           //寄存器的地址
4.     uc8 Name;           //数据的方向，X，Y 或 Z
5.     s8 Out;             //寄存器的值
6.     float Acc;          //加速度值
7.     float Angle;        //角度值
8. }MMA_Dat;
```

在结构体初始化的时候把寄存器的地址和名称定义好了：

```
1.  /*****/
2.    MMA_Dat X_Value={MMA_XOUT8_Addr,'X'};
3.    MMA_Dat Y_Value={MMA_YOUT8_Addr,'Y'};
4.    MMA_Dat Z_Value={MMA_ZOUT8_Addr,'Z'};
```

要注意的是 MMA7455 中的数据以二补数的形式来储存：

截图来自《MMA7455L》

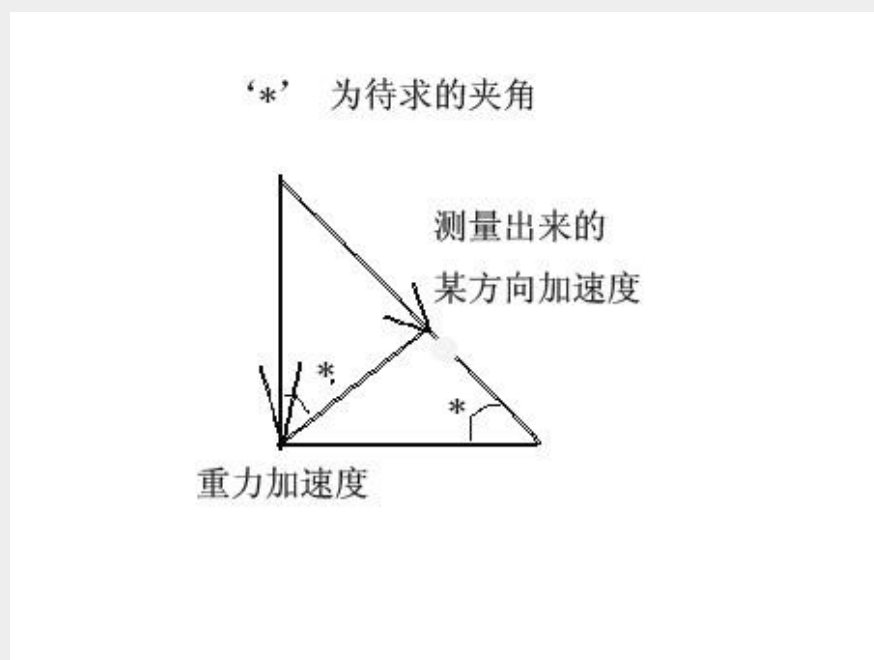
\$00: 10bits Output Value X LSB (Read only)

| D7       | D6       | D5       | D4       | D3       | D2       | D1       | D0       | Bit      |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| XOUT [7] | XOUT [6] | XOUT [5] | XOUT [4] | XOUT [3] | XOUT [2] | XOUT [1] | XOUT [0] | Function |
| 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | Default  |

Signed byte data (2's complement): 0g = 10'h000

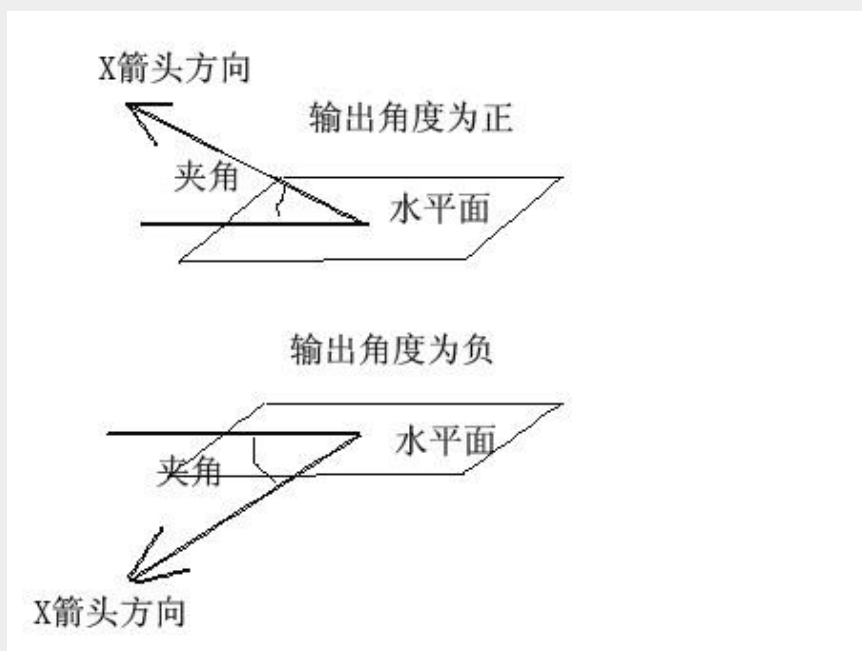
把二补数转换为原码后就可以像处理 ADC 的数据一样了，我们选择的是 2g 量程，灵敏度为：64LSB/g。

至于角度的计算公式，把重力加速度按各方向的分解，可以推导出来。



计算出来的 加速度或角度 乘以 -1 是野火根据传感器上标注的箭头的方向，并以此为正方向进行调整的，不同的厂家生产的传感器稍有区别。

下图是以 X 方向输出的角度为例：



输出的角度为各箭头方向的直线与水平面间的夹角，在水平面上方为正，下方为负。

接下来分析一下 校准函数：

```
1. /*
2.  * 函数名: I2C_MMA_Cal
3.  * 描述 : MMA7455 校准
4.  * 输入 : 无
5.  * 输出 : 无
6.  * 调用 : 外部调用，在初始化后调用
7.  */
8. void I2C_MMA_Cal(void)
9. {
10.     I2C_MMA_ByteWrite(0x00,MMA_YOFFL_Addr); /*校正 x 值 00 */
11.     I2C_MMA_ByteWrite(0x30,MMA_YOFFL_Addr); /*校正 y 值 48*/
12.     I2C_MMA_ByteWrite(0xE2,MMA_ZOFFL_Addr); /*校正 z 值 -30 的补码 */
13.     I2C_MMA_ByteWrite(0xFF,MMA_ZOFFH_Addr); /*校正 z 值,校正值为负数,要把
        高位写 1;*/
14. }
```

`I2C_MMA_Cal()` 函数是用来校正传感器的，称为 **0g 校准**。这个函数在第一次测量前必须调用，而且每个传感器的校正值都有不同，其中的校正参数就要大家亲手去调试出来啦。

参照 DataSheet 《AN3745》按以下步骤校准：

1. 把传感器按水平方式放置，读取各方向寄存器输出值。

这个情况下，Z 轴方向标准输出应为 **1g**，X 轴和 Y 轴均为 **0**。对应到各个寄存器的原始数据就应是 **ZOUT8 = 64**，**XOUT8 = 0**，**YOUT8 = 0**。但是未校准前，各个寄存器的输出会有一定的偏差。



以下为传感器水平放置，野火的例程中未校准得出的数据

```

-----这是一个重力传感器测试程序-----

-----X方向的数据-----

寄存器的原始数据是: XOUT8 = 0
以 箭头标注 方向为 正 方向, X方向加速度是: 0.00 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: 0.00 度

-----Y方向的数据-----

寄存器的原始数据是: YOUT8 = -19
以 箭头标注 方向为 正 方向, Y方向加速度是: 2.91 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -17.28 度

-----Z方向的数据-----

寄存器的原始数据是: ZOUT8 = 75
以 箭头标注 方向为 正 方向, Z方向加速度是: -11.48 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: 90.00 度

Acceleration enter standby mode!

```

## 2. 向相应的 OffSet 寄存器写入校准值:

要注意两个问题。

一是校准寄存器中的值为 **1/2 LSB**，所以我们写入的值要相应地**乘以 2**倍:

**XOUT8** 很标准，不用写入校准值，或向 **XOFFL** 写入 0;

**YOUT8** 输出为-19，所以应 **YOFFL** 写入  $38 = 2 \times 19$ ;

**ZOUT8** 输出为 75，所以应写入  $-22 = 2 \times (64 - 75)$ ;



截图来自《MMA7455L》

**\$10: Offset Drift X LSB (Read/Write)**

The following Offset Drift Registers are used for setting and storing the offset calibrations to eliminate the 0g offset. Please refer to Freescale application note AN3745 for detailed instructions on the process to set and store the calibration values.

| D7      | D6      | D5      | D4      | D3      | D2      | D1      | D0      | Bit      |
|---------|---------|---------|---------|---------|---------|---------|---------|----------|
| XOFF[7] | XOFF[6] | XOFF[5] | XOFF[4] | XOFF[3] | XOFF[2] | XOFF[1] | XOFF[0] | Function |
| 0       | 0       | 0       | 0       | 0       | 0       | 0       | 0       | Default  |

Signed byte data (2's complement): User level offset trim value for X-axis

| Bit     | XOFF[7] | XOFF[6] | XOFF[5] | XOFF[4] | XOFF[3] | XOFF[2] | XOFF[1] | XOFF[0] |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| Weight* | 64 LSB  | 32 LSB  | 16 LSB  | 8 LSB   | 4 LSB   | 2 LSB   | 1 LSB   | 0.5 LSB |

\*Bit weight is for 8g 10-bit data output. Typical value for reference only. Variation is specified in "Electrical Characteristics" section.

**\$11: Offset Drift X MSB (Read/Write)**

| D7 | D6 | D5 | D4 | D3 | D2       | D1      | D0      | Bit      |
|----|----|----|----|----|----------|---------|---------|----------|
| -- | -- | -- | -- | -- | XOFF[10] | XOFF[9] | XOFF[8] | Function |
| 0  | 0  | 0  | 0  | 0  | 0        | 0       | 0       | Default  |

Signed byte data (2's complement): User level offset trim value for X-axis

当校准值要写入负数时，要注意第二个问题：

就是在 MMA7455 寄存器中数值是以补码的形式储存的，所以实际写入的数据要经过转化：-22 的补码为 0xEA。但是，向 ZOFFL 写入 0xEA 还是未能校准，因为校准寄存器的还有高 8 位，高 8 位必须全写入 1 才是 -22 的补码，所以还要向 ZOFFH 写入 0xff。

选取恰当的校准值往往要经过多次的检测，为了提高准确度，就辛苦一点吧。

最后讲解一下 Standby 模式，只要向 MCTL 寄存器写入 0x04 命令就可进入 Standby 模式，这时传感器不工作，功耗大大降低：

```

1. *
2. * 函数名: I2C_MMA_Standby
3. * 描述   : 重力传感器进入 Standby 模式，省电
4. * 输入   : 无
5. * 输出   : 无
6. * 返回   : 是否成功进入 Standby 模式
7. * 调用   : 外部调用
8. */
9. u8 I2C_MMA_Standby(void)
10. {
11.     u8 MMA_Test;
12.
13.     /*MMA Standby Mode*/
14.     I2C_MMA_ByteWrite(0x04, MMA_MCTL_Addr);
15.
16.     /*MMA_Test*/
17.     MMA_Test = I2C_MMA_ByteRead(MMA_MCTL_Addr);

```

```

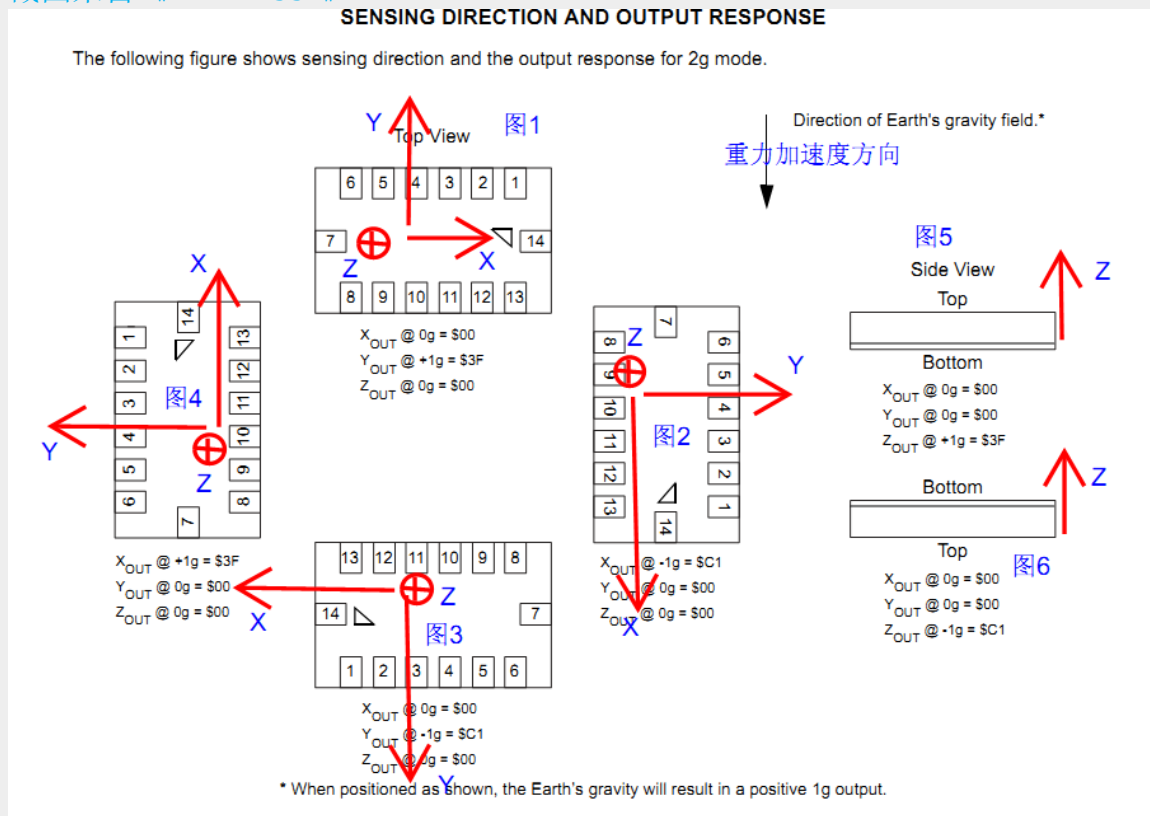
18.
19.     if(MMA_Test == 0x04)
20.         return SUCCESS ;
21.     else
22.         return ERROR;
23. }
    
```

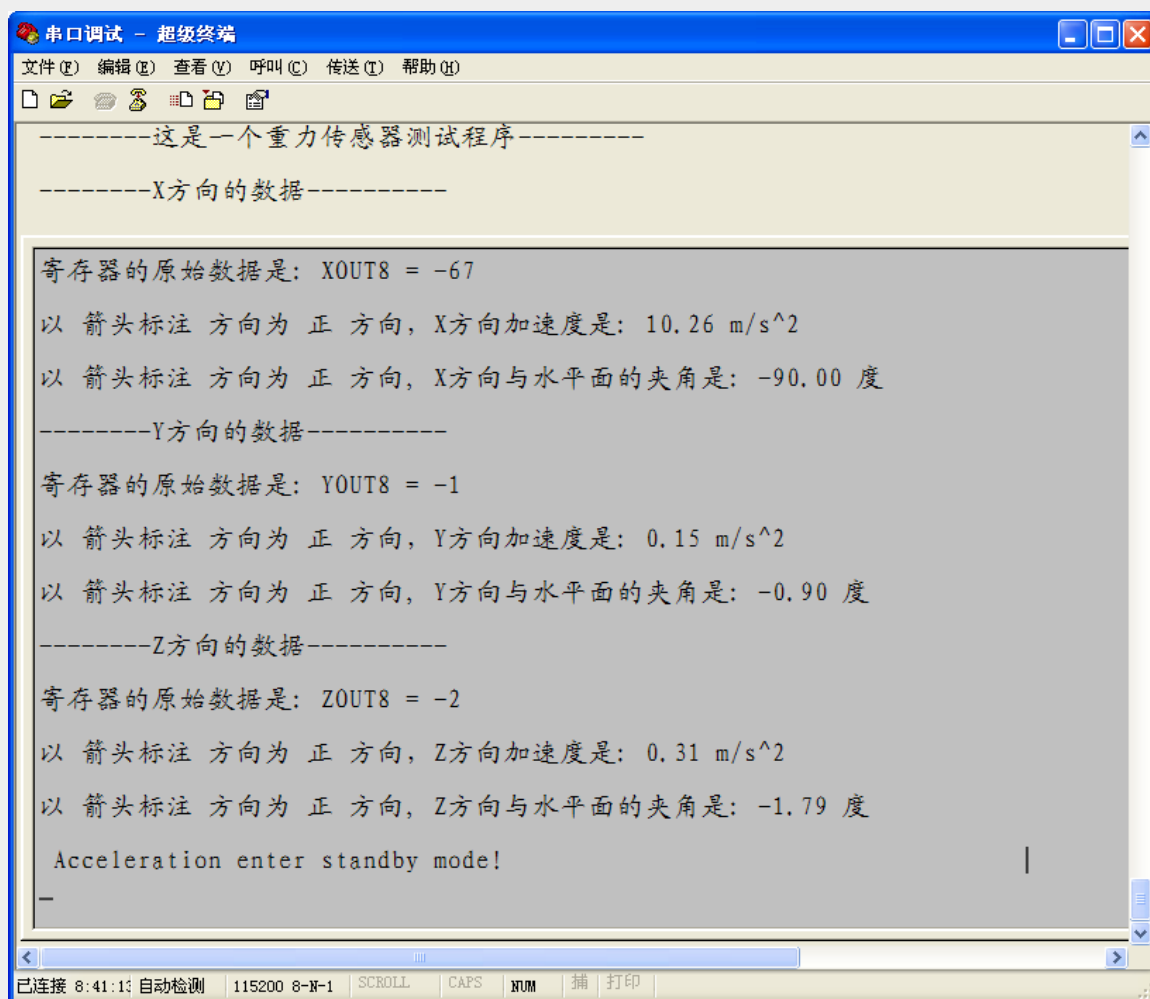
### 35. 4. 10.4 实验现象

很多第一次使用加速度传感器的人都会被它的方向弄糊涂，以下为传感器分别按 DataSheet 的这图 3，图 4，图 5，三个个方向来放置时没得的数据。

注释中的各个 XYZ 轴的方向为野火使用的传感器上标注的箭头方向，设为正方向。将野火 STM32 开发板供电(DC5V)，插上 JLINK。

截图来自《MMA7455L》



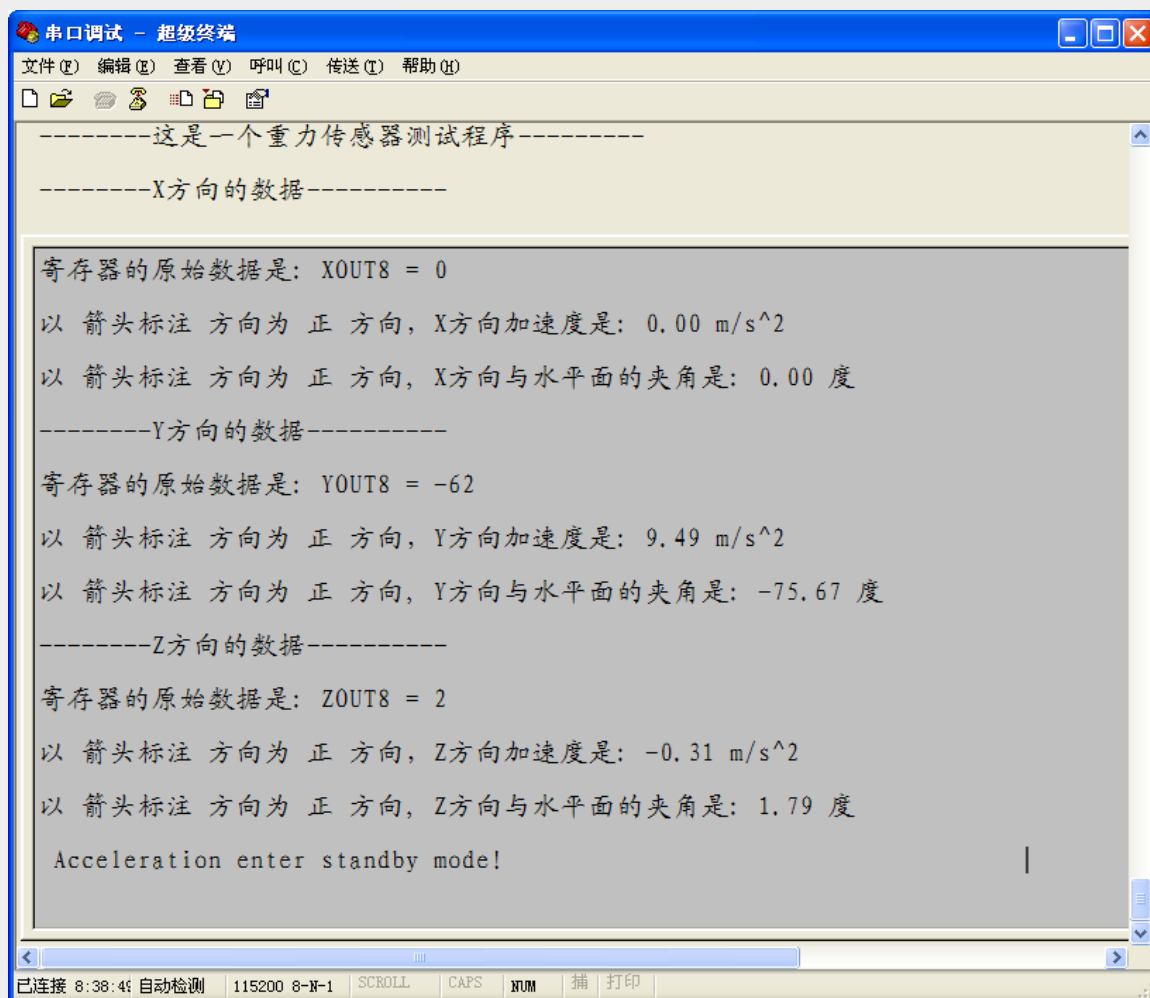


```

串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = -67
以 箭头标注 方向为 正 方向, X方向加速度是: 10.26 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: -90.00 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -1
以 箭头标注 方向为 正 方向, Y方向加速度是: 0.15 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -0.90 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = -2
以 箭头标注 方向为 正 方向, Z方向加速度是: 0.31 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: -1.79 度
Acceleration enter standby mode!
-
已连接 8:41:13 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

```

按图 2 方向放置

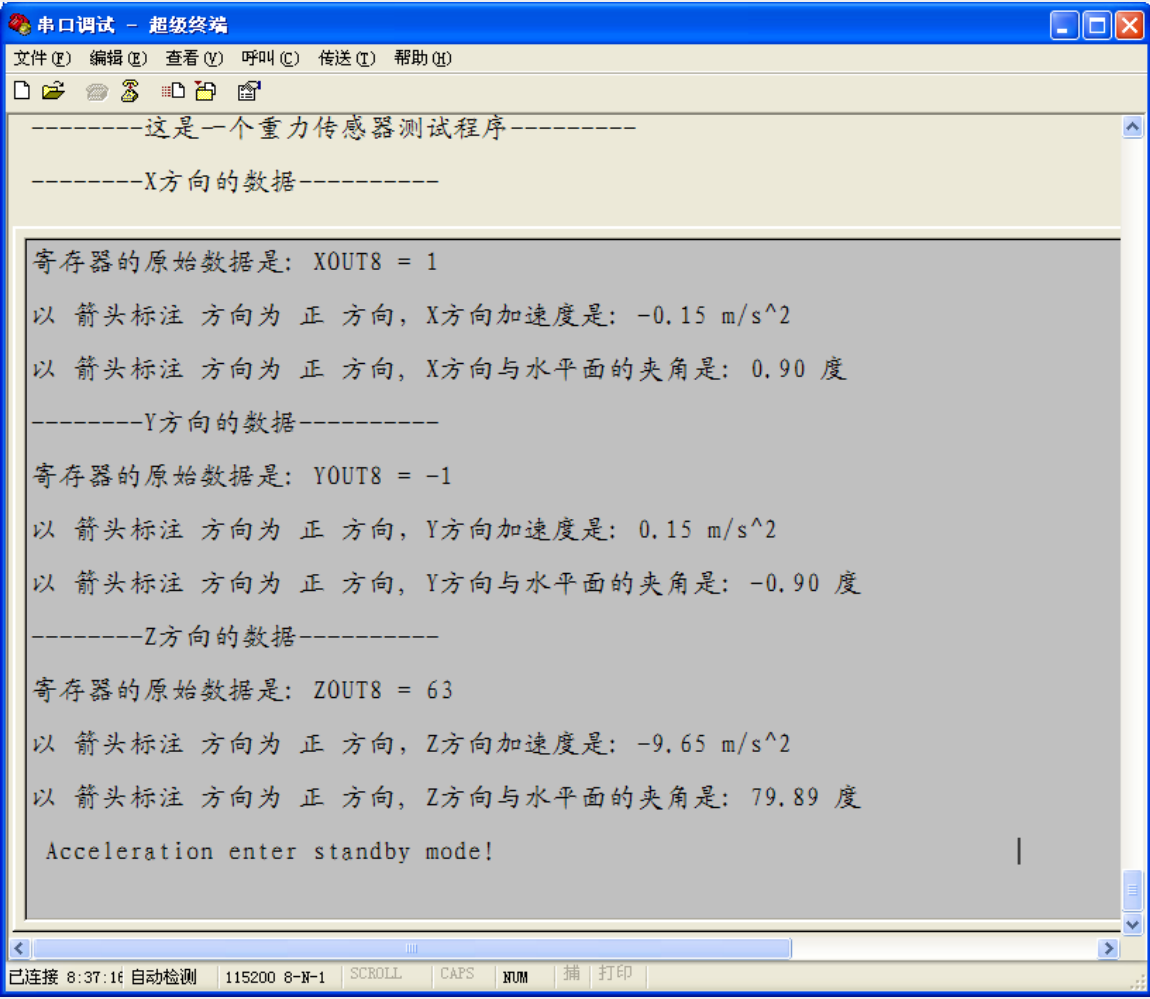


```

串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = 0
以 箭头标注 方向为 正 方向, X方向加速度是: 0.00 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: 0.00 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -62
以 箭头标注 方向为 正 方向, Y方向加速度是: 9.49 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -75.67 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = 2
以 箭头标注 方向为 正 方向, Z方向加速度是: -0.31 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: 1.79 度
Acceleration enter standby mode!
已连接 8:38:44 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

```

按图 3 方向放置



```
串口调试 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
-----这是一个重力传感器测试程序-----
-----X方向的数据-----
寄存器的原始数据是: XOUT8 = 1
以 箭头标注 方向为 正 方向, X方向加速度是: -0.15 m/s^2
以 箭头标注 方向为 正 方向, X方向与水平面的夹角是: 0.90 度
-----Y方向的数据-----
寄存器的原始数据是: YOUT8 = -1
以 箭头标注 方向为 正 方向, Y方向加速度是: 0.15 m/s^2
以 箭头标注 方向为 正 方向, Y方向与水平面的夹角是: -0.90 度
-----Z方向的数据-----
寄存器的原始数据是: ZOUT8 = 63
以 箭头标注 方向为 正 方向, Z方向加速度是: -9.65 m/s^2
以 箭头标注 方向为 正 方向, Z方向与水平面的夹角是: 79.89 度
Acceleration enter standby mode!
已连接 8:37:16 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

## 36. 以太网-Lwip

互联网技术对人类社会的影响不言而喻。当今大部分电子设备都能以不同的方式接入互联网(Internet)，在家庭中 PC 常见的互联网接入方式是使用路由器(Router)组建小型局域网(LAN)，利用调制调解器(modem)经过电话线网络，连接到互联网服务提供商(ISP)，由互联网服务提供商把用户的局域网接入互联网。而企业或学校的局域网规模较大，常使用交换机组成局域网，经过路由以不同的方式接入到互联网中。

### 36.1. 互联网模型

互联网技术的发展过程中，曾出现过 OSI 七层模型和 TCP/IP 四层模型，两种模型分别有对应的协议，这两种模型和协议都有各自的优缺点。而最终 TCP/IP 改进模型(混合模型)及其协议被广泛使用。

设计网络时，为了降低网络设计的复杂性，对组成网络的硬件、软件进行封装、分层，这些分层即构成了网络体系模型。在两个设备相同层之间的对话、通信约定，构成了层级协议。设备中使用的所有协议加起来统称协议栈。这三个概念 是互联网技术的核心。

虽然当前互联网的模型没有被明确规定，但我们一般可以使用五层的混合参考模型来理解。见图 36-1。

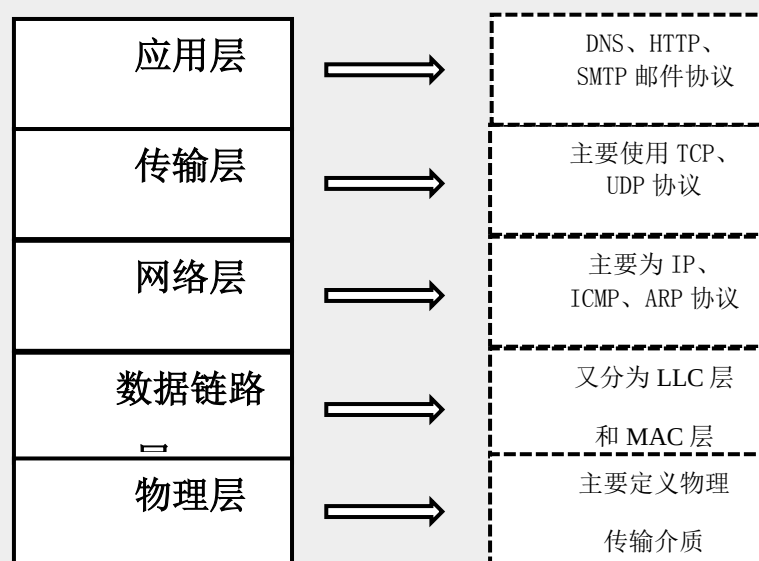


图 36-1 混合参考模型

本书开篇便强调了分层在解决计算机科学问题的强大作用，在各个章节中都有不同程度的体现，网络体系模型对网络传输问题的分层解决则再次印证了分层思想的作用。在这个网络模型中，每一层完成不同的任务，都提供接口供上一层访问。而在每层的内部，可以使用不同的方式来实现接口，因而内部的变化不会影响其它层。

在本参考模型中，数据链路层又被分为 LLC 层(逻辑链路层)和 MAC 层(媒体介质访问层)。目前，对于普通的接入网络终端的设备，LLC 层和 MAC 层是软、硬件的分界线。如 PC 的网卡主要负责实现参考模型中的 MAC 子层和物理层，在 PC 的软件系统中则有一套庞大程序实现了 LLC 层及以上的所有网络层次的协议。

由硬件实现的物理层和 MAC 子层在不同的网络形式有很大的区别，如以太网和 Wi-Fi，这是由物理传输方式决定的。但由软件实现的其它网络层次通常不会有太大区别，在 PC 上也许能实现完整的功能，支持所有协议，而在嵌入式领域则按需要进行裁剪。本章的网络模型的硬件部分用以太网为例讲解，软件部分以 LwIP 协议栈讲解。

## 36.2. 以太网

以太网(Ethernet)是互联网技术的一种, 由于它是在组网技术中占的比例最高, 很多人直接把以太网理解为互联网。

以太网是指遵守 IEEE 802.3 标准组成的局域网, 由 IEEE 802.3 标准规定的主要是位于参考模型的物理层(PHY)和数据链路层中的媒体接入控制子层(MAC)。在家庭、企业和学校所组建的 PC 局域网形式一般也是以太网, 其标志是使用水晶头网线来连接(当然还有其它形式)。IEEE 还有其它局域网标准, 如 IEEE 802.11 是无线局域网, 俗称 Wi-Fi。IEEE 802.15 是个人域网, 即蓝牙技术, 其中的 802.15.4 标准则是当前非常热门的 ZigBee 技术。

渗透到工业控制、环境监测、智能家居的嵌入式设备产生了接入互联网的需求, 利用以太网技术, 嵌入式设备可以非常容易地接入到现有的计算机网络中。

### 36.2.1. PHY 层

在物理层, 由 IEEE 802.3 标准规定了以太网使用的传输介质、传输速度、数据编码方式和冲突检测机制。

#### ■ 传输介质

传输介质包括同轴电缆、双绞线(水晶头网线是一种双绞线)、光纤。根据不同的传输速度和距离要求, 基于这三类介质的信号线又衍生出很多不同的种类。最常用的是“五类线”适用于 100BASE-T 和 10BASE-T 的网络, 它们的网络速率分别为 100Mbps 和 10Mbps。

#### ■ 编码

为了让接收方在没有外部时钟参考的情况也能确定每一位的起始、结束和中间位置, 在传输信号时不直接采用二进制编码。在 10BASE-T 的传输方式中采用曼彻斯特编码, 在 100BASE-T 中则采用 4B/5B 编码。



曼彻斯特编码把每一个二进制位的周期分为两个间隔，在表示“1”时，以前半个周期为高电平，后半个周期为低电平。表示“0”时则相反，见图36-2。

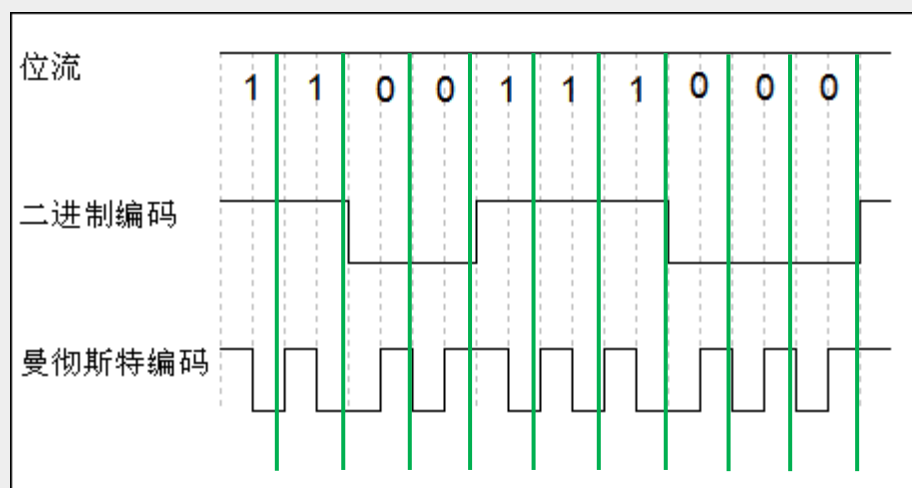


图 36-2 曼彻斯特编码

采用曼彻斯特码在每个位周期都有电压变化，便于同步。但这样的编码方式效率太低，只有 50%。

在 100BASE-T 采用的 4B/5B 编码是把待发送数据位流的每 4 位分为一组，以特定的 5 位编码来表示，这些特定的 5 位编码能使数据流有足够多的跳变，达到同步的目的，而且效率也从曼彻斯特编码的 50%提高到了 80%。

### ■ CSMA/CD 冲突检测

早期的以太网大多是多个节点连接到同一条网络总线上(总线型网络)，存在信道竞争问题，因而每个连接到以太网上的节点都必须具备冲突检测功能。以太网具备 CSMA/CD 冲突检测机制，如果多个节点同时利用同一条总线发送数据，则会产生冲突，总线上的节点可通过接收到的信号与原始发送的信号的比较检测是否存在冲突，若存在冲突则停止发送数据，随机等待一段时间再重传。

现在大多数局域网组建的时候很少采用总线型网络，大多是一个设备接入到一个独立的路由或交换机接口，组成星型网络，不会产生冲突。但为了兼容，新出的产品还是带有冲突检测机制。

### 36.2.2. MAC 子层

#### ■ MAC 的功能

MAC 子层是属于数据链路层的下半部分，它主要负责与物理层进行数据交接，如是否可以发送数据，发送的数据是否正确，对数据流进行控制等。它自动对来自上层的数据包加上一些控制信号，交给物理层。接收方得到正常数据时，自动去除 MAC 控制信号，把该数据包交给上层。

#### ■ MAC 数据包

IEEE 对以太网上传输的数据包格式也进行了统一规定，见图 36-3。该数据包被称为 MAC 数据包。

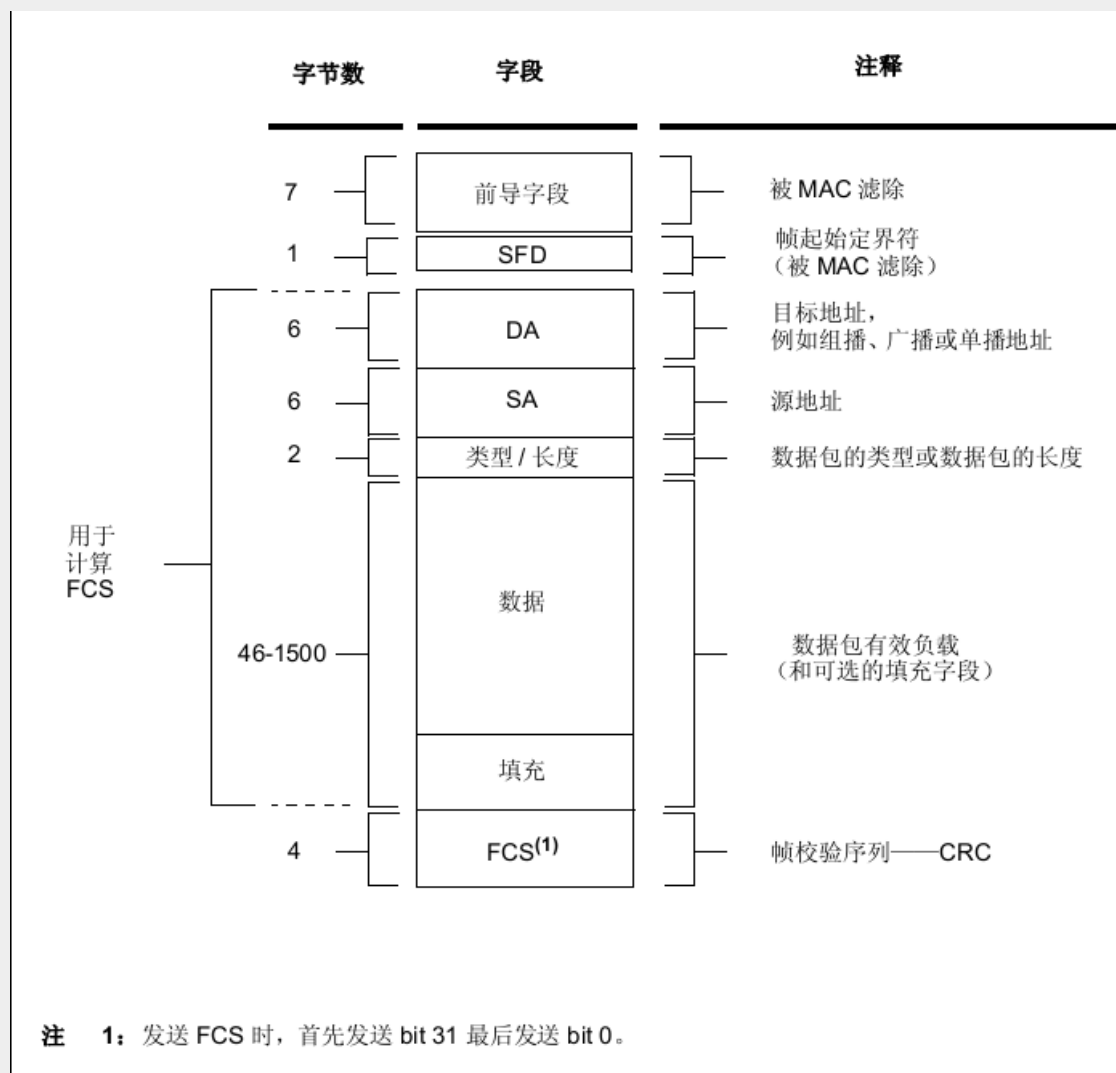


图 36-3 MAC 数据包格式

MAC 数据包由前导字段、帧起始定界符、目标地址、源地址、数据包类型、数据域、填充域、校验和域组成。

❑ 前导字段，这是一段方波，用于使收发节点的时钟同步。而帧起始定界符则用于区分前导段与数据段的。字段和帧起始定界符在 MAC 收到数据包后会自动过滤掉。

❑ MAC 地址：MAC 地址由 48 位数字组成，它是网卡的物理地址，在以太网传输的最底层，就是根据 MAC 地址来收发数据的。部分 MAC 地址用于广播和多播，在同一个网络里不能有两个相同的 MAC 地址。PC 的网卡在出厂时已经设置好了 MAC 地址，但也可以通过一些软件来进行修改，

在嵌入式的以太网控制器中可由程序进行配置。数据包中的 DA 是目标地址，SA 是源地址。

□ 数据包类型：本区域可以用来描述本 MAC 数据包是属于 TCP/IP 协议层的 IP 包、ARP 包还是 SNMP 包，也可以用来描述本 MAC 数据包数据段的长度。

□ 数据段：数据段是 MAC 包的核心内容，它包含的数据来自 MAC 的上层。其长度可以从 0~1500 字节间变化。

□ 填充域：由于协议要求整个 MAC 数据包的长度至少为 64 字节(接收到的数据包如果少于 64 字节会被认为发生冲突，数据包被自动丢弃)，当数据段的字节少于 46 字节时，在填充域会自动填上无效数据，以使数据包符合长度要求。

□ 校验和域：MAC 数据包的尾部是校验和域，它保存了 CRC 校验序列，用于检错。

### 36.2.3. 以太网控制器

#### ■ 集成 MAC 控制器的 MCU 接入方案

PHY 层和 MAC 层的实现是由以太网控制器实现的。部分 MCU 如 STM32F107 型号的芯片，集成了 MAC 控制器外设，在芯片外再外接一个 PHY 控制器 和以太网变压器接口即可实现以太网功能。它们的关系可以类比 STM32 的 CAN 通信模型，STM32 集成了 CAN 控制器，但要实现 CAN 通信还需要外接 CAN 收发器芯片。以太网变压器的功能是增强、隔离信号及与 RJ45 水晶头进行连接。

STM32 的 MAC 控制器外设可通过 IEEE 协议规定的 MII 或 RMII、SMI 与 PHY 芯片进行通讯。STM32F107 型号芯片使用 MII 接口与 PHY 构成的以太网接口见图 36-4。(省略了以太网变压器)。

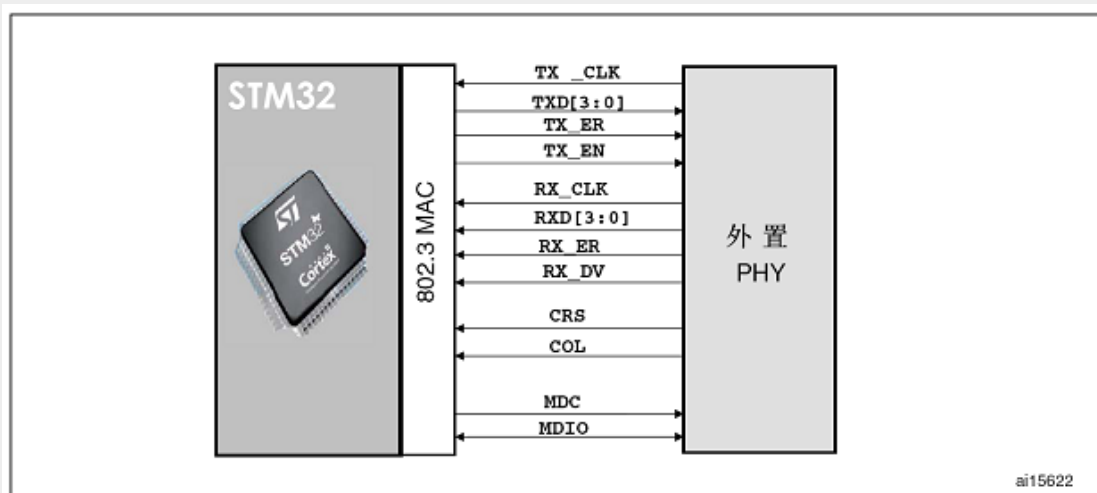


图 36-4 STM32F107 与 PHY 芯片构成以太网接口

### ■ 外接以太网控制器方案

对于没有集成以太网控制器的 MCU，可通过外接以太网控制器芯片接入以太网。本书使用的 STM32F103VET6 型号的 MCU 没有集成以太网控制器，所以采用 STM32 外接常用的嵌入式以太网控制器 ENC28J60 接入网络。

ENC28J60 芯片是兼容 IEEE 802.3 的以太网控制器，集成 MAC 控制器和 10 BASE-T PHY 控制器，自带缓冲区、DMA，使用 SPI 接口与 MCU 进行通讯。MCU 使用 SPI 对 ENC28J60 芯片的寄存器写入控制参数和接收数据，实现以太网的功能。它与 MCU 连接组成的以太网接口见图 36-5。

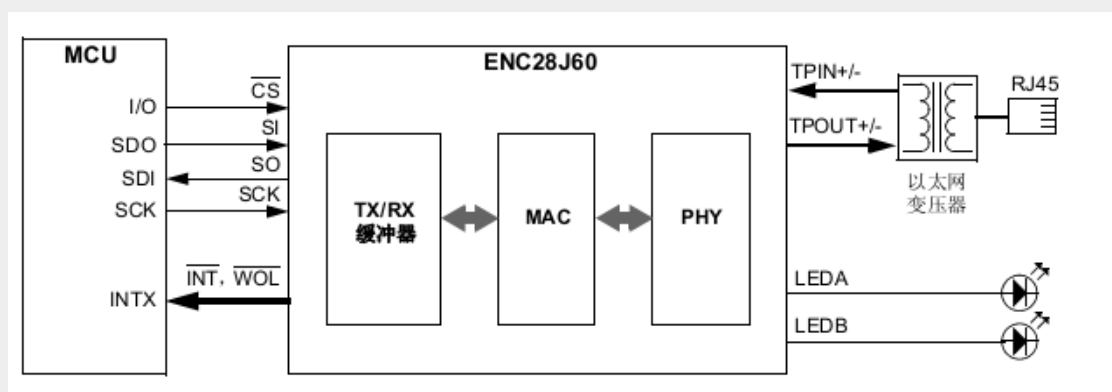


图 36-5 MCU 与 ENC28J60 组成以太网接口

使用以太网控制器，实现了混合模型的 MAC 子层和物理层，就可以利用它进行基本的收发 MAC 数据包了。

### 36.3. MAC 之上的网络层

#### 36.3.1. 为什么在 MAC 之上还有分层？

如果两个 STM32 开发板，都实现了以太网接口的 MAC 子层和物理层，我们使用网线把这两个板子连接起来，针对接收端和发送端写两个代码，使得在 MAC 控制器收取到 MAC 数据包时，把位于数据段的内容提取出来，毫无疑问这是可以实现数据传输的，如果读者想尝试这样的功能，可以参考配本书的 NRF24L01 无线传输例程。

但是如果使用以太网仅仅是为了实现这样的功能真是大材小用了，而且还是有线传输，实在是鸡肋。想实现用 STM32 开发板与 PC 通讯，如果不同的 PC 使用不同的系统，或者设备使用嵌入式系统，针对每种差异，都需要分别写一套软件，这样会过于复杂。

使用以太网接口的目的就是为了方便与其它设备互联，如果所有设备都约定使用一种互联方式，在软件上加一些层次来封装，这样不同系统、不同的设备通讯就变得相对容易了。而且只要新加入的设备也使用同一种方式，就可以直接与之前存在于网络上的其它设备通讯。这就是为什么产生了在 MAC 之上的其它层次的网络协议及为什么要使用协议栈的原因。又由于在各种协议栈中 TCP/IP 协议栈 得到了最广泛使用，所有接入互联网的设备都遵守 TCP/IP 协议。所以，想方便地与其它设备互联通信，需要提供对 TCP/IP 协议的支持。

#### 36.3.2. 各网络层的功能

用以太网和 Wi-Fi 作例子，它们的 MAC 子层和物理层有较大的区别，但在 MAC 之上的 LLC 层、网络层、传输层和应用层的协议，是基本上同的，这几层协议由软件实现，并对各层进行封装。根据 TCP/IP 协议，各层的要实现的功能如下：

LLC 层：处理传输错误；调节数据流，协调收发数据双方速度，防止发送方发送得太快而接收方丢失数据。主要使用数据链路协议。

网络层：本层也被称为 IP 层。LLC 层负责把数据从线的一端传输到另一端，但很多时候不同的设备位于不同的网络中(并不是简单的网线的两头)。此时就需要网络层来解决子网路由拓扑问题、路径选择问题。在这一层主要有 IP 协议、ICMP 协议。

传输层：由网络层处理好了网络传输的路径问题后，端到端的路径就建立起来了。传输层就负责处理端到端的通讯。在这一层中主要有 TCP、UDP 协议

应用层：经过前面三层的处理，通讯完全建立。应用层可以通过调用传输层的接口来编写特定的应用程序。而 TCP/IP 协议一般也会包含一些简单的应用程序如 Telnet 远程登录、FTP 文件传输、SMTP 邮件传输协议。

实际上，在发送数据时，经过网络协议栈的每一层，都会给来自上层的数据添加上一个数据包的头，再传递给下一层。在接收方收到数据时，一层层地把所在层的数据包的头去掉，向上层递交数据。见图 36-6。

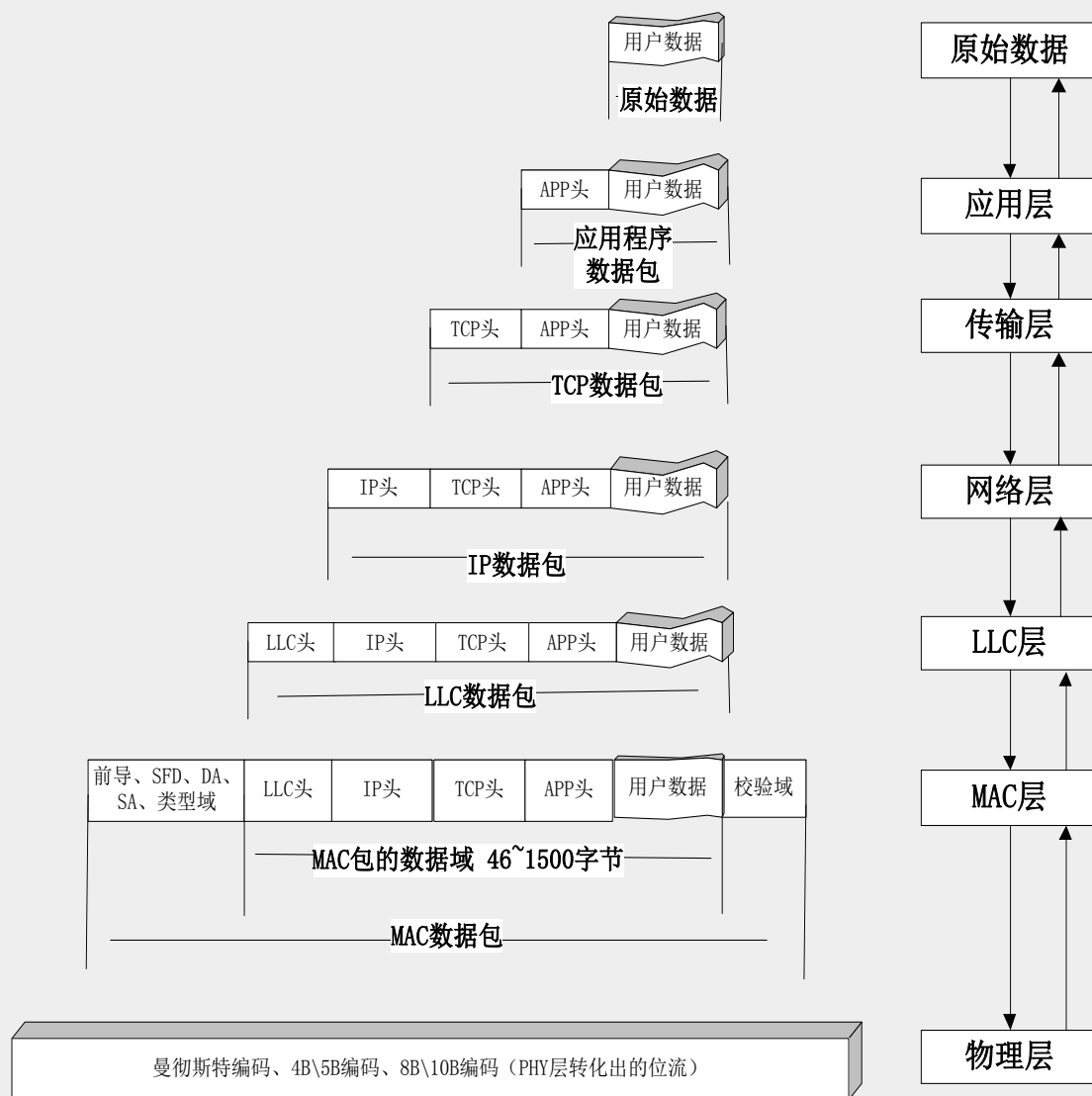


图 36-6 数据经过每一层的封装和还原

### 36.3.3. LwIP 协议栈

#### ■ LwIP 协议栈简介

要实现 TCP/IP 协议栈的，把每一层都处理好，这样的代码当然可以自己亲自写，但我们一般会移植更加稳定和性能优良的代码。最著名的实现了 TCP/IP 协议栈的代码是 BSD TCP/IP 协议栈，大多数专业的协议栈都是由它派生出来的，由它提供的接口 **socket** 连接模式也几乎成为了通用标准。但 BSD 协议栈比



庞大，大多数小型的嵌入式设备不宜使用，所以人们还开发出了 uc/IP、LwIP、uIP 等适用于嵌入式设备使用的协议栈。

LwIP 是 Light Weight Internet Protocol 的缩写，是由瑞士计算机科学院 Adam Dunkels 等开发的适用于嵌入式领域的轻量级 TCP/IP 协议栈。它可以移植到含有操作系统的平台中，也可以在没有操作系统的平台下运行。由于它开源、占用的 RAM 和 ROM 比较少、支持较为完整的 TCP/IP 协议、且十分便于裁剪、调试，被广泛应用在中低端的 32 位控制器平台。

### ■ 获取 LwIP 协议栈

LwIP 协议栈是开源的，可以在网址 <http://download.savannah.gnu.org/releases/lwip/> 下载得到，目前最新版本 1.4.0 版刚推出不久，尚不清楚其稳定性如何，本章采用 LwIP 的 1.3.2 版本进行讲解。从该网站中可以下载到 lwip-1.3.2.zip 文件和相应版本的 contrib-1.3.0.zip 文件，它们包含了 LwIP 协议栈的核心源代码及应用、移植例程。

解压 lwip-1.3.2.zip 文件后，其目录\lwip-1.3.2\src 的内容如图 36-7。

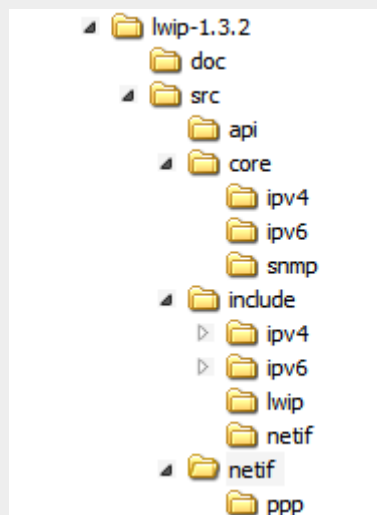


图 36-7 lwip 的 src 目录

在 src 目录下的 api 文件夹保存的文件中，包含了适用于具有操作系统平台调用的应用层接口函数。core 文件夹下的文件内容为 LwIP 协议栈对于各种协

议的实现。`netif` 下的文件则保存了与硬件底层关系比较紧密的函数。这三个文件夹下的都是 c 源文件，它们的头文件都被保存到 `include` 文件夹中。在实际应用中，可根据需要进行裁剪。

在移植的时候，我们也常常需要利用 `contrib-1.3.0.zip` 中的文件。解压后，在 `\contrib-1.3.0_\contrib\ports` 目录下有一些针对特定平台移植时使用的文件，选择进入其中一个目录，如 `\contrib-1.3.0_\contrib\ports\old\6502\include\arch`，其内容如图 36-8。

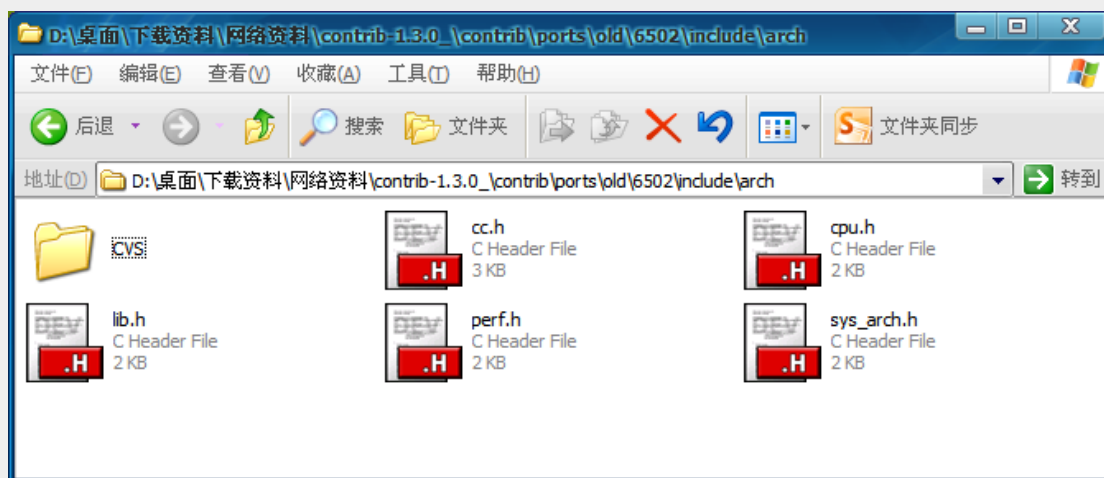


图 36-8 特定头文件

我们需要用到的是 `cc.h`、`perf.h` 和 `sys_arch.h` 文件，通常把它复制出来存放到自己工程中的 `arch` 文件夹中。`cc.h` 包含了 LwIP 对于基本数据类型的定义。`sys_arch.h` 定义了与系统有关的信号量、邮箱及线程。

## 36.4. ENC28J60+LwIP 以太网实验

### 36.4.1. 实验描述及工程文件清单

#### 实验描述：

使用野火开发板利用 ENC28J60 以太网控制器接入局域网，移植 LwIP 协议栈，并利用协议栈提供的函数在 STM32 上建立服务器和创建 Telnet 应用。

建立服务器后可以使用 PC 的浏览器访问网页，通过鼠标点击网页按钮，控制开发板上的 LED 灯。Telnet 应用，利用 PC 上的 Telnet 程序向 STM32 输入命令，控制 LED 灯。

本实验分析十分复杂，对网络应用不熟悉的读者，务必先阅读本章的实验步骤和实验现象，这样既能有个大概了解，也能激起学习的兴趣。

#### 硬件连接：

PB13 : ENC28J60-INT

PA6-SPI1-MISO : ENC28J60-SO

PA7-SPI1-MOSI : ENC28J60-SI

PA5-SPI1-SCK : ENC28J60-SCK

PA4-SPI1-NSS : ENC28J60-CS

PE1 : ENC28J60-RST

#### 库文件：

使用 3.5 版本固件库

startup/start\_stm32f10x\_hd.c

CMSIS/core\_cm3.c

CMSIS/system\_stm32f10x.c

FWlib/stm32f10x\_spi.c

FWlib/stm32f10x\_rcc.c

FWlib/stm32f10x\_gpio.c

FWlib/stm32f10x\_usart.c

#### 用户文件：

USER/main.c

USER/stm32f10x\_it.c

USER/SysTick.c

USER/usart1.c

USER/led.c

USER/ENC28J60.c

USER/SPI.c

USER/httpd.c

USER/netconfig.c

USER/cmd.c

## LwIP 文件:

## 使用 1.3.2 版本 LwIP 协议栈

lwip-1.3.2.zip 文件解压后 src 中的源文件，具体见工程环境配置。

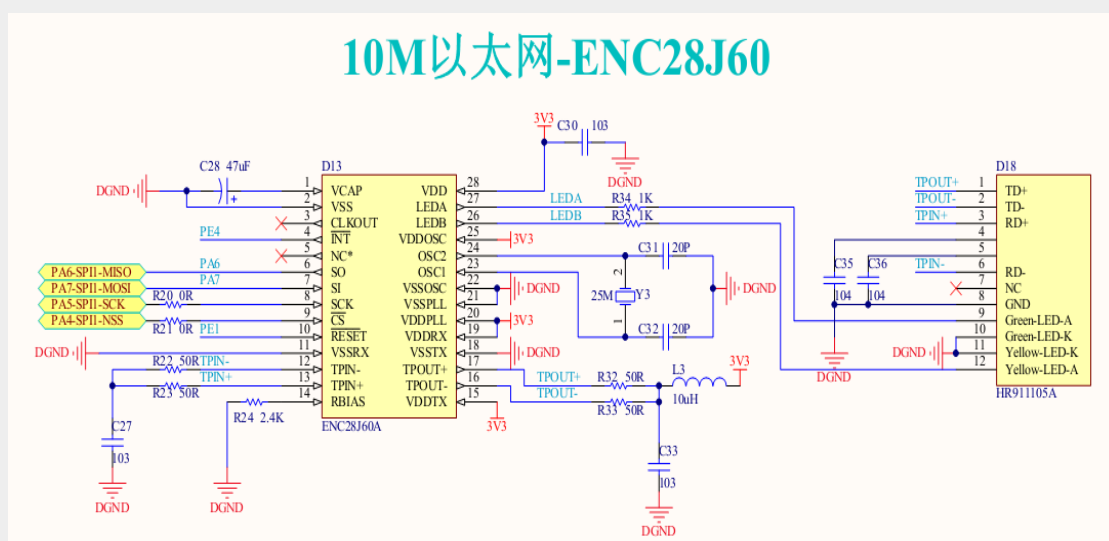


图 36-9 ENC28J60 与 STM32 硬件连接图

### 36.4.2. 配置工程环境

以太网实验中我们用到了 GPIO、RCC、SPI、USART 外设，所以我们要先把以下库文件添加到工程：`stm32f10x_gpio.c`、`stm32f10x_rcc.c`、`stm32f10x_spi.c`、`stm32f10x_usart.c`。实验中还使用了旧工程中的 `led.c` 和 `systick.c` 文件用于网络控制 LED 和给 LwIP 协议栈提供定时功能。

接下来就要新建文件编写 **SPI** 初始化函数和编写 **ENC28J60** 的驱动文件了，由于本次实验使用的文件十分多，所以把 **SPI.C** 文件和 **ENC28J60.C** 文件放到新的分组中管理。

在应用层则创建了三个文件用来配置网卡、服务器和 **telnet** 应用程序，这三个文件分别为 **httpd.c**、**netconfig.c** 和 **cmd.c**。

最后，LwIP 协议栈整个文件夹 lwip-1.3.2 复制到目录下，从 contrib-1.3.0.zip 中复制 cc.h 等头文件保存到新建的 ARCH 文件夹下。把使用到的 LwIP 协议栈的源文件都按它在 scr 的分组添加到工程，添加后的结果见图 36-12。

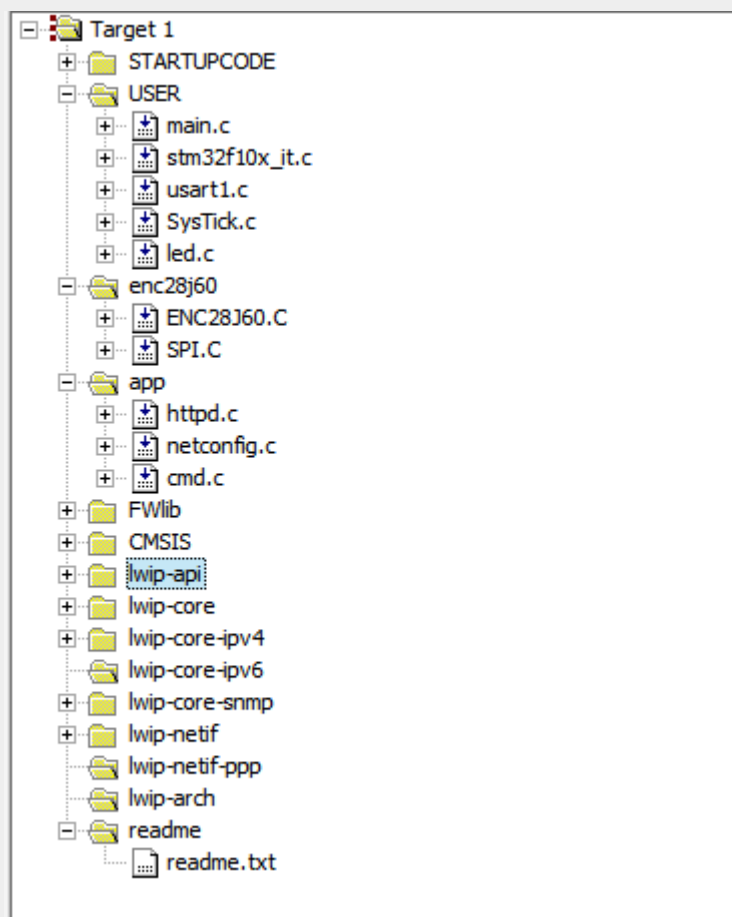


图 36-10 添加工程文件

其中带有“+”号的 LwIP 组下，都包含 LwIP 协议文件包中相应文件夹的所有源文件(文件太多，无法一一列出，请参照例程)，没有“+”号的 ipv6 和 ppp 工程组相应的源文件则因为没有被使用到而未被添加进工程。

添加完工程后，还要在 c/c++编译器选项中添加头文件路径，主要为 LwIP 的头文件，见图 36-11。

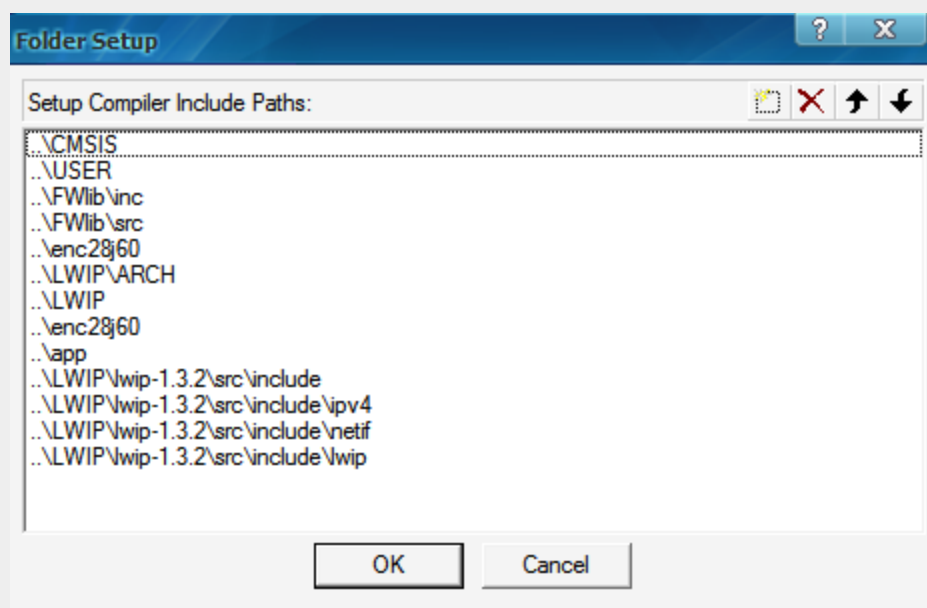


图 36-11 添加头文件路径

### 36.4.3. main 文件

无论工程多么复杂，坚持从 main 文件理解不动摇。本工程中的 main 文件主要内容见代码清单 36-1。

代码清单 36-1：以太网例程的 main 函数

```
46. /*****
*****/
47.
48. __IO uint32_t LocalTime = 0;
49. /* 该变量每 10ms 改变一次，增量为 10*/
50.
51. /*
52. * 函数名: main
53. * 描述   : 主函数
54. * 输入   : 无
55. * 输出   : 无
56. * 调用   :
57. */
58. int main(void)
59. {
60.     /*初始化串口*/
61.     USART1_Config();
62.
63.     /*初始化 以太网 SPI 接口*/
64.     ENC_SPI_Init();
65.
66.     /*初始化 systick, 用于定时轮询输入或给 LWIP 提供定时*/
67.     SysTick_Init();
68.
69.     /*初始化 LED 使用的端口*/
70.     LED_GPIO_Config();
71.
```

```
72.     printf("\r\n*****野火 STM32_enc8j60+lwip 移植实验\n");
73.
74.     /* 初始化 LWIP 协议栈 */
75.     LwIP_Init();
76.
77.     /* 初始化 web server 显示网页程序 */
78.     httpd_init();
79.
80.     /* 初始化 telnet 远程控制程序 */
81.     CMD_init();
82.
83.     /* Infinite loop */
84.     while ( 1 )
85.     {
86.         /* 轮询 */
87.         LwIP_Periodic_Handle(LocalTime);
88.     }
89. }
90.
```

在 main 函数中，使用 USART1\_Config() 配置了串口驱动；用 ENC\_SPI\_Init() 配置好了与 ENC28J60 进行通讯的 SPI 接口；SysTick\_Init() 配置了用于给 LwIP 定时的 systick；LED\_GPIO\_Config() 则对控制 LED 的 GPIO 进行了初始化。

LwIP\_Init() 初始化了 LwIP 协议栈，它设置了网络接口的 IP、子网掩码、网关，并使能了 ENC28J60。httpd\_init() 完成了建立服务器所需的状态，如监听网络端口，配置连接后调用的各种函数，以完成网页控制 LED 的功能。类似地，CMD\_init() 完成了 telnet 控制 LED 的准备。

最后，就在 while 死循环中调用 LwIP\_Periodic\_Handle() 根据 LocalTime 变量的计时值，轮询网络接口和给 LwIP 协议栈系统提供定时。

#### 36.4.4. LwIP 对底层数据结构的封装

了解了 main 函数的执行流程后，读者可能会觉得奇怪为何没有关于 ENC28J60 接口驱动的初始化呢？(ENC\_SPI\_Init() 函数仅仅完成了 SPI 接口的 GPIO、SPI 模式配置)，实际上它是在 LwIP\_Init() 函数完成的，但由于封装的层次比较多，所以不仔细分析很难了解到它是如何完成对底层的 ENC28J60 进行初始化的。

由于硬件底层使用的设备十分容易发生变化，同时也为了使 LwIP 的适用范围更广，就像文件系统的底层驱动是留给移植者完成一样，LwIP 协议栈给底层的驱动提供了接口，如初始化网卡、发送数据包、接收数据包等最底层的操作。这些接口定义位于\lwip-1.3.2\src\netif 目录下的 ethernetif.c 文件，该文件已经被添加进了工程中，接口的具体实现因使用的网络接口不同而有所区别，需要我们移植者来完成。

在编写这些接口相关的内容时，涉及到了两个关系密切的 LwIP 的底层数据结构 pbuf、netif 结构体。

### ■ pbuf 结构体

LwIP 的底层接口涉及到了大量对 pbuf、netif 结构体赋值的内容，我们先来分析它们的结构体成员。

当 MAC 层收到 MAC 数据包的时候，它会把 MAC 包中的前导字段和帧起始定界符过滤掉，把剩余的数据段都保存起来，交给 LwIP 的 pbuf 结构体类型的变量，由 LwIP 协议栈进行处理。pbuf 结构体在 pbuf.h 文件中定义见代码清单 36-2。

代码清单 36-2: pbuf 结构体定义

```
23. struct pbuf {
24.     /** next pbuf in singly linked pbuf chain */
25.     struct pbuf *next;
26.
27.     /** pointer to the actual data in the buffer */
28.     void *payload;
29.
30.     /**
31.      * total length of this buffer and all next buffers in chain
32.      * belonging to the same packet. */
33.     u16_t tot_len;
34.
35.     /** length of this buffer */
36.     u16_t len;
37.
38.     /** pbuf_type as u8_t instead of enum to save space */
39.     u8_t type; /*pbuf_type*/
40.
41.     /** misc flags */
42.     u8_t flags;
43.
44.     /**
45.      * the reference count always equals the number of pointers
```



```
46.     * that refer to this pbuf. This can be pointers from an appl  
ication,     * the stack itself, or pbuf->next pointers from a chain.  
47.     */  
48.     u16_t ref;  
49.  
50.};
```

其成员作用如下：

### (1) next

这是一个指向 `pbuf` 类型的指针，它用于指向下一个 `pbuf` 结构体。由于每个 MAC 包的数据段最大只是 1500 字节，而来自上层的数据包往往很大，LwIP 把大数据包分装到多个 `pbuf` 结构体，并把这些 `pbuf` 组成链表，通过 `next` 指针来索引。

### (2) payload

这是一个指向实际数据的指针。当 MAC 控制器接收到 MAC 数据包时，整个 MAC 包(包括 MAC 地址)直接被存储到 `payload` 指向的存储区。

当需要发送数据时，LwIP 一层层地给原始数据加上数据头，最终组装成 MAC 包的格式，也保存到 `payload` 指向的存储区，然后把这些数据交给底层的 MAC 控制器发送出去。

### (3) tot\_len 和 len

`len` 成员表示本 `pbuf` 结构体的长度，`tot_len` 表示本 `pbuf` 及下一个 `pbuf` (`next` 指向的 `pbuf`)长度的和。

例如 `pbuf A` 的 `next` 成员指向 `pbuf B`，若 `pbuf A` 的 `len` 值为 500，`pbuf B` 的 `len` 值为 600，那么 `pbuf A` 的 `tot_len` 长度就等于 500+600，而 `pbuf B` 的 `tot_len` 长度就等于 600+下一个 `pbuf` 的 `len`。若 `pbuf C` 是 `pbuf` 链表中的最后一个，那么 `pbuf C` 的 `len` 就等于它的 `tot_len`，从而就可以判断链表的结束。

### (4) type 和 flags

`type` 表示 `pbuf` 的存储类型，共有四种，分别为 `PBUF_RAM`、`PBUF_ROM`、`PBUF_REF` 和 `PBUF_POOL`，针对不同的应用而使用不同的存储方式。`flags` 在存储分配时被赋值为 0，作用不明，可能是作者为了兼容而保留的。

### (5) ref

ref 用于记录 pbuf 的访问次数，在 LwIP 中存在一根据访问次数来决定内存释放的机制

从上面的分析可以了解到 LwIP 利用 pbuf 结构体来进行与底层硬件的交接，用 payload 存储数据包的指针，为了便于管理，使用 next、len、tot\_len、type、ref 成员来定义 pbuf 的属性。

最后，参考图 36-12 可以让我们更加清晰地了解 pbuf 的结构，该图是存储形式为 PBUF\_POOL 类型的结构体存储分布，其它类型的 pbuf 也是类似的。

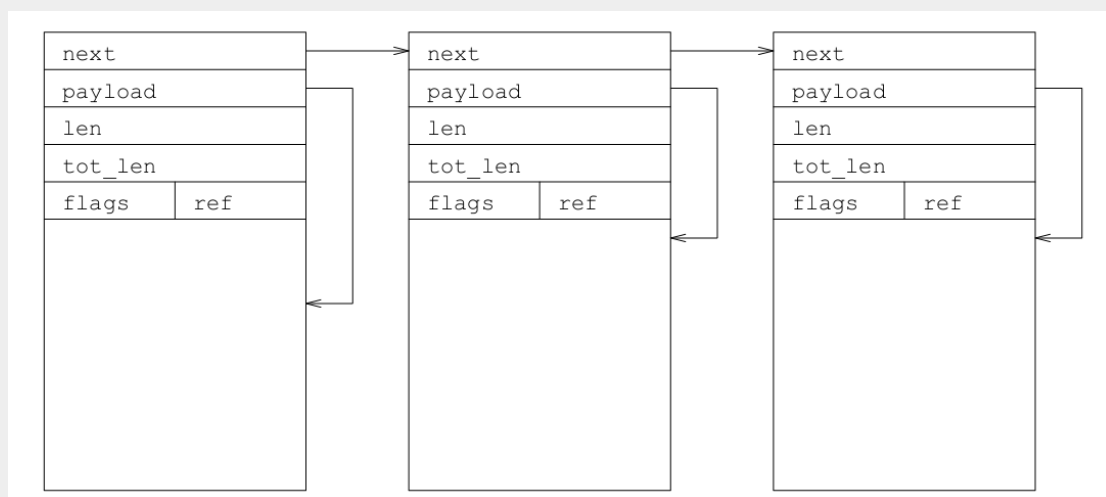


图 36-12 PBUF\_POOL 类型的存储方式

### ■ netif 结构体

LwIP 通过 pbuf 建立了与底层硬件收发数据包的数据结构，可以实现数据的管理。但很多时候，收发数据还需要清楚网卡的状态，特别是网卡地址、IP 地址、网关等设置，有的设备上还可能有多网卡，这就需要有一个结构体来保存这些信息，让 LwIP 进行管理和使用。这些信息被保存在 netif 结构体中，它的定义在 netif.h 文件定义见代码清单 36-3。(为了便于讲解，在初学时，只分析重点代码，野火只保留了重点与网卡相关的结构体成员，条件编译部分都被删除了，想了解完整版的读者可参考源码)。

代码清单 36-3: netif 结构体

```
(10) struct netif {
(11)     /** pointer to next in linked list */
(12)     struct netif *next;
(13)
(14)     /** IP address configuration in network byte order */
(15)     struct ip_addr ip_addr;
(16)     struct ip_addr netmask;
(17)     struct ip_addr gw;
(18)
(19)     /** This function is called by the network device driver
(20)      * to pass a packet up the TCP/IP stack. */
(21)     err_t (* input)(struct pbuf *p, struct netif *inp);
(22)
(23)     /** This function is called by the IP module when it wants
(24)      * to send a packet on the interface. This function typically
y
(25)      * first resolves the hardware address, then sends the packet. */
t.
(26)
(27)     err_t (* output)(struct netif *netif, struct pbuf *p,
(28)                     struct ip_addr *ipaddr);
(29)
(30)     /** This function is called by the ARP module when it wants
(31)      * to send a packet on the interface. This function outputs
(32)      * the pbuf as-is on the link medium. */
(33)     err_t (* linkoutput)(struct netif *netif, struct pbuf *p);
(34)
(35)     /** This field can be set by the device driver and could point
t
(36)      * to state information for the device. */
(37)
(38)     void *state;
(39)
(40)     /** maximum transfer unit (in bytes) */
(41)     u16_t mtu;
(42)
(43)     /** number of bytes used in hwaddr */
(44)     u8_t hwaddr_len;
(45)
(46)     /** link level hardware address of this interface */
(47)
(48)     u8_t hwaddr[NETIF_MAX_HWADDR_LEN];
(49)     /** flags (see NETIF_FLAG_ above) */
(50)
(51)     u8_t flags;
(52)     /** descriptive abbreviation */
(53)
(54)     char name[2];
(55)     /** number of this interface */
(56)
(57)     u8_t num;
(58)
(59) };
```

各成员的作用如下：

## 27. next

这个 `next` 指针是 `netif` 结构体类型，它与 `pbuf` 的 `next` 指针作用类似，也是用作链表。`netif` 结构体是用来存储网卡属性的，由 `netif` 结构体构成的链表即表示同一个设备上不同网卡的属性，LwIP 就通过本 `next` 指针访问这个链表。

## 28. `ip_addr`、`netmask` 和 `gw`

这三个结构体成员分别用来存储本网卡的 IP 地址、子网掩码和网关。

## 29. `err_t (*input)(struct pbuf *p, struct netif *inp)`

这个结构体成员第一眼看上去有点奇怪，与以往碰到的都不一样，实际上这是一个函数指针。这个函数是用来接收底层硬件输入的数据包的。函数的返回值是 `err_t` 类型，它是由 LwIP 定义的一种用于表示操作成功、失败等参数的变量。函数的输入参数为 `pbuf` 类型的 `p` 指针和 `netif` 类型的 `inp` 指针，表示我们要接收 `inp` 网卡通过底层硬件接收到的数据包，并把数据包保存到结构体 `p` 中。

## 30. `err_t (*output)(struct netif *netif, struct pbuf *p, struct ip_addr *ipaddr)`

本成员也是一个函数指针，它指向一个用于 IP 层输出的函数。在 IP 层有数据包需要发送时，就通过该指针调用 `output` 函数。本函数输入参数有 `netif`、`pbuf` 和 `ip_addr` 三个参数，表示使用 `netif` 网卡、把 `pbuf` 结构体的内容发送到 IP 地址为 `ip_addr` 的设备中。但由于在硬件底层收发数据时是使用 MAC 地址的，所以本函数把 IP 地址经过转化得到 MAC 地址后，调用下一个成员——`linkoutput` 函数。

## 31. `err_t (*linkoutput)(struct netif *netif, struct pbuf *p)`

这个函数指针是指向最底层的网卡发送数据包函数。它是被上面的 `output` 函数调用的。在 `output` 函数中的 `pbuf` 数据包实际上缺少了 MAC 包中的 DA 段（目标 MAC 地址），`output` 函数把它的输入参数 IP 地址转化成 MAC 地址后，添加到 `pbuf` 的 `payload` 成员，组成完整 MAC 包，再由本函数 `linkoutput` 输出到网络中。

## 32. `state`

`state` 指针指向一用户感兴趣的信息，这部分可以由用户自行配置，也可以不使用。

### 33. `mtu`

本成员记录了最大传输单元，我们知道 MAC 数据段的最大长度为 1500 字节，所以本成员我们一般会直接赋值为 1500。

### 34. `hwaddr_len` 和 `hwaddr[NETIF_MAX_HWADDR_LEN]`

`hwaddr_len` 存储的是 MAC 地址长度，`hwaddr[]` 数组则存储了本网卡的 MAC 完整地址。

### 35. `flags`

本成员保存了网卡允许使用的功能，如使用广播地址、ARP 功能等。

### 36. `name` 和 `num`

`name` 用于保存网络接口的名字，可以自由设置，名字是两个字节的 ASCII 编码。当网络接口名字出现相同的时候，使用 `num` 的数值进行区分。

## 36.4.5. 初始化协议栈

了解了 `pbuf`、`netif` 结构体后，我们进入被 `main` 函数调用的 `LwIP_Init()` 的具体代码中(删减了部分扩展功能的代码)，该函数位于自定义的 `netconfig.c` 文件，见代码清单 36-4。

代码清单 36-4: `LwIP_Init()` 函数

```
19. /*
20. * 函数名: LwIP_Init
21. * 描述   : 初始化 LWIP 协议栈，主要是把 enc28j60 与 LWIP 连接起来。
22.           包括 IP、MAC 地址，接口函数
23. * 输入   : 无
24. * 输出   : 无
25. * 调用   : 外部调用
26. */
27. void LwIP_Init( void )
28. {
29.     struct ip_addr ipaddr;
30.     struct ip_addr netmask;
31.     struct ip_addr gw;
32.
33.     /*调用 LWIP 初始化函数，
34.     初始化网络接口结构体链表、内存池、pbuf 结构体*/
35.     lwip_init();
36. }
```

```

37. IP4_ADDR(&ipaddr, 192, 168, 1, 18);           //设置网络接口的 ip 地
址
38. IP4_ADDR(&netmask, 255, 255, 255, 0);        //子网掩码
39. IP4_ADDR(&gw, 192, 168, 1, 1);              //网关
40.
41. /*初始化 enc28j60 与 LWIP 的接口, 参数为网络接口结构体、ip 地址、
42. 子网掩码、网关、网卡信息指针、初始化函数、输入函数*/
43. netif_add(&enc28j60, &ipaddr, &netmask, &gw, NULL, &etherneti
f_init, &ethernet_input);
44.
45. /*把 enc28j60 设置为默认网卡 */
46. netif_set_default(&enc28j60);
47.
48. /* When the netif is fully configured this function must be
called.*/
49. netif_set_up(&enc28j60); //使能 enc28j60 接口
50. }

```

函数执行流程如下：

18. 第 11~13 行，定义了三个 `ip_addr` 类型的变量用于存储 IP 地址、子网掩码、网关，这些变量在第 19~21 行被赋值，本实验的网卡这三个属性就是由它们配置的。

19. 第 17 行，调用 `lwip_init()` 函数(名字跟 `LwIP_Init()` 很相似，注意区分)，`lwip_init()` 是由 LwIP 协议栈定义的函数，在使用 LwIP 协议栈相关的内容前，要先调用它对网卡结构体链表、内存池、`pbuf` 进行初始化。

20. 第 25 行，调用 `netif_add()` 函数，它把我们配置好的网卡属性赋值到 `netif` 类型的 `enc28j60` 变量中，它是在本文件的开头定义的(被省略了)。如 `ip` 地址、子网掩码、网关地址、网卡初始化函数 `ethernetif_init()`、网卡输入函数 `ethernet_input()`。在本函数之后的 LwIP 协议栈操作，只要使用到了相关的底层服务，都通过这个 `netif` 类型的 `enc28j60` 变量来访问。代码中的 28~31 行调用两个函数的作用分别是设置 `enc28j60` 结构体指向的网卡为默认网卡，并使能。

### ■ `netif_add()` 函数

`netif_add()` 是上面提到的重点函数，它是由 LwIP 协议栈定义的，它的使用方法和我们使用 STM32 固件库初始化外设时类似，先把初始化结构体配置好，

然后调用 `xxx_init` 函数把这些配置写入到寄存器。`netif_add()`函数则把我们对网卡属性的配置写入到该函数的输入参数 `netif` 中，本实验调用时该参数是变量 `enc28j60`。`netif_add()`在 `netif.c` 文件的定义见代码清单 36-5(有删减)。

代码清单 36-5: `netif_add()`函数

```
40. struct netif *
41. netif_add(struct netif *netif, struct ip_addr *ipaddr, struct i
p_addr *netmask,
42.   struct ip_addr *gw,
43.   void *state,
44.   err_t (* init)(struct netif *netif),
45.   err_t (* input)(struct pbuf *p, struct netif *netif))
46. {
47.     static u8_t netifnum = 0;
48.
49.     /* reset new interface configuration state */
50.     netif->ip_addr.addr = 0;    //复位结构体成员
51.     netif->netmask.addr = 0;
52.     netif->gw.addr = 0;
53.     netif->flags = 0;
54.
55.     /* remember netif specific state information data */
56.     netif->state = state;
57.     netif->num = netifnum++;
58.     netif->input = input;    //输入函数
59.
60.     netif_set_addr(netif, ipaddr, netmask, gw);    //设置三个地址
61.
62.     /* call user specified initialization function for netif */
63.     if (init(netif) != ERR_OK) {
64.         return NULL;
65.     }    //初始化函数
66.
67.     /* add this netif to the list */
68.     netif->next = netif_list;    //加入链表
69.     netif_list = netif;    //
70.     snmp_inc_iflist();
71.
72.     return netif;
73. }
```

从它的源码中可以看出这个函数就是完成了对 `netif` 结构体的赋值，在这里有一个问题，调用 `netif_add()`函数时(见 `LwIP_Init()`中的调用)，输入的参数 `ethernetif_init` 和 `ethernet_input` 分别是网卡初始化函数和输入函数的指针，这两个指针指向的函数具体是如何定义的呢？

#### 36.4.6. LwIP 对底层操作的封装

要解决上一小节的问题，需要对了解 `LwIP` 对底层操作的封装。对这些操作封装后给上层提供的接口呈现在 `LwIP` 协议栈中就是上述的函数指针形式，

LwIP 协议栈内部的函数通过这些指针来调用函数。这些接口都保存在 \lwip-1.3.2\src\netif 文件夹下的 ethernetif.c 文件中。它已被添加至本实验工程的 lwip-netif 组中。

### ■ ethernetif\_init 指针

首先看看网卡初始化函数 ethernetif\_init(), 它并不是最底层的, 该函数的具体定义都是由 LwIP 协议栈已经完全编写好。其定义见代码清单 36-6。

代码清单 36-6: ethernetif\_init()函数

```
33.err_t ethernetif_init( struct netif *netif )
34.{
35. struct ethernetif *ethernetif;
36.
37. ethernetif = mem_malloc( sizeof(struct ethernetif) ); //分
配内存
38.
39. if( ethernetif == NULL ) //debug 时使用的输出
40. {
41. LWIP_DEBUGF( NETIF_DEBUG, ("ethernetif_init: out of mem
ory\n\r") );
42. return ERR_MEM;
43. }
44.
45. netif->state = ethernetif;
46. netif->name[0] = IFNAME0; //网卡名字
47. netif->name[1] = IFNAME1;
48. netif->output = etharp_output; //IP 层输出函数
49. netif->linkoutput = low_level_output; //底层硬件输出函数
50.
51. ethernetif->ethaddr = ( struct eth_addr * ) &( netif-
>hwaddr[0] ); //MAC 地址
52.
53. low_level_init( netif ); //最底层初始化函
数
54.
55. return ERR_OK;
56. }
```

在上一小节的 netif\_add()只是对 netif 部分的结构体成员进行了赋值, 还有一些成员如 state、name、output 函数指针、linkoutput 函数指针及 mac 地址成员, 就是在本函数中进行赋值的, 见第 13~21 行。来到这个函数, 我们终于发现了最底层的输出函数指针 low\_level\_output 和最底层的初始化函数指针 low\_level\_init。



## ■ 底层输出函数

LwIP 协议栈的原文档只是提供了 `low_level_output` 函数的模型，野火对该函数的具体实现见代码清单 36-7。

代码清单 36-7: `low_level_output()`函数

```
1.  /*
2.   * low_level_output(): Should do the actual transmission of the
3.   * packet. The
4.   * packet is contained in the pbuf that is passed to the functi
5.   * on. This pbuf
6.   * might be chained.
7.   */
8.  static err_t low_level_output( struct netif *netif, struct pbuf
9.  *p ) /*底层发送数据函数*/
10. {
11.     struct pbuf *q;
12.     int i = 0;
13.     err_t xReturn = ERR_OK;
14.     /* Parameter not used. */
15.     for(q = p; q != NULL; q = q->next)
16.     {
17.         //复制 pbuf 内容
18.         memcpy(&Tx_Data_Buf[i], (u8_t*)q->payload, q->len);
19.         i = i + q->len;
20.     }
21.     enc28j60PacketSend(i, Tx_Data_Buf); //发送数据包
22.     return xReturn;
23. }
```

本函数看上去十分简单，在上层需要发送数据的时候，以 `pbuf` 作为输入参数调用 `low_level_output()` 函数。而 `pbuf` 链表的所有 `payload` 成员就是需要发送的 MAC 数据包，每个数据包的长度为 `len` 成员的值。函数中利用 `for` 循环调用 `memcpy()` 把 `q->payload` 链表的数据都取出来存放在 `Tx_Data_Buf` 数组中，最后调用 `enc28j60PacketSend()` 函数把 `Tx_Data_Buf` 数组发送出去。`enc28j60PacketSend()` 是在 `ENC28J60.C` 文件编写好的 `enc28j60` 驱动，它使用 `SPI` 接口控制 `enc28j60` 芯片发送数据。

## ■ 底层初始化函数

最底层的网卡初始化函数 `low_level_init()` 的定义见代码清单 36-8。

代码清单 36-8: `low_level_init()`函数

```

2. static void low_level_init( struct netif *netif )
3. {
4.     /* set MAC hardware address length */
5.     netif->hwaddr_len = 6;
6.
7.     /* set MAC hardware address */
8.     /* MAC 地址 */
9.     netif->hwaddr[0] = macaddress[0];
10.    netif->hwaddr[1] = macaddress[1];
11.    netif->hwaddr[2] = macaddress[2];
12.    netif->hwaddr[3] = macaddress[3];
13.    netif->hwaddr[4] = macaddress[4];
14.    netif->hwaddr[5] = macaddress[5];
15.
16.    /* maximum transfer unit */
17.    /* 最大传输单元 */
18.    netif->mtu = netifMTU;
19.
20.    /* broadcast capability */
21.    netif->flags =
22.    NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP | NETIF_FLAG_LINK_UP;
23.
24.    enc28j60Init(netif->hwaddr);          //初始化 enc28j60
25.    enc28j60clkout(1);    // change clkout from 6.25MHz to 12.5
MHz
26.
27. }

```

本初始化函数中先对 netif 结构体的 MAC 地址成员 netif->hwaddr 赋值为 macaddress 数组的值，macaddress 数组是在 netconfig.c 文件中定义的，定义的时候已经设置了它的初始值，在本函数中复制到 netif->hwaddr 后作为网卡的 MAC 地址。本函数中对最大传输单元结构体成员 netif->mtu 赋值为 1500，对 netif->flags 赋值使其和允许使用广播地址、使能 ARP 功能。函数的最后，像上面的 low\_level\_output 函数一样，调用了在 ENC28J60.C 文件中编写的驱动。分别是初始化函数 enc28j60Init()和时钟输出函数 enc28j60clkout()。

## ■ ethernet\_input 指针

分析完 netif\_add()输入参数 ehernetif\_init 指针，再来看看另一个作为它的输入参数的函数指针 ethernet\_input，它是由 LwIP 协议栈自带的完整函数，一般不需要我们修改，见代码清单 36-9。

代码清单 36-9: ethernet\_input()函数

```

1. /*
2.  * ethernet_input(): This function should be called when a pa
cket is ready to

```

```

3.  * be read from the interface. It uses the function low_level_i
nput() that
4.  * should handle the actual reception of bytes from the network
interface.
5.  */
6. err_t ethernetif_input(struct netif *netif)
7. {
8.     err_t err = ERR_OK;
9.     struct pbuf *p = NULL;
10.
11.     /* move received packet into a new pbuf */
12.     p = low_level_input(netif);
13.
14.     if (p == NULL) return ERR_MEM;
15.
16.     err = netif->input(p, netif);
17.     if (err != ERR_OK)
18.     {
19.         LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error
\n"));
20.         pbuf_free(p);
21.         p = NULL;
22.     }
23.
24.     return err;
25. }

```

由于本实验中的 enc28j60 驱动没有使用中断接收数据包，所以 ethernetif\_input() 需要被 main 函数中 while 循环的 LwIP\_Periodic\_Handle() 函数不停地调用，通过 ethernetif\_input() 函数的返回值来判断是否接收到数据包。本函数执行时，先调用了最底层的输入函数 low\_level\_input()，它在底层又调用了 enc28j60 接收数据包的驱动，若接收到数据包，返回值赋值到 pbuf 类型指针 pbuf 非空，然后在本函数的第 16 行，把 pbuf 输入到 netif 结构体的 input 函数指针，通过 input 指针把数据包传递给上层的 LwIP 协议栈代码进行处理，完成数据包的收取。（读者也可以修改 enc28j60 的驱动，使用中断方式接收数据包，就不需要采用轮询的方式了）

### ■ 底层输入函数

上面的 ethernetif\_input() 调用了最底层的输入函数 low\_level\_input()，我们需要在这个函数中实验 LwIP 接收数据包的功能，主要是通过调用 enc28j60 来实现的。见代码清单 36-10。

代码清单 36-10: low\_level\_input() 函数

```
18. /*
```

```

19. * low_level_input(): Should allocate a pbuf and transfer the b
ytes of the
20. * incoming packet from the interface into the pbuf.
21. */
22.
23. static struct pbuf *low_level_input( struct netif *netif )
24. {
25.
26.     struct pbuf *q,*p = NULL;
27.     u16 Len = 0;
28.
29.     int i =0;
30.
31.     //接收数据包, 并返回接收到的数据包长度
32.     Len = enc28j60PacketReceive(1520 *4, Data_Buf);
33.
34.     if ( Len == 0 ) return 0;
35.
36.     p = pbuf_alloc(PBUF_RAW, Len, PBUF_POOL);
37.
38.     if (p != NULL)
39.     {
40.
41.         for (q = p; q != NULL; q = q->next)
42.         {
43.             memcpy((u8_t*)q->payload, (u8_t*)&Data_Buf[i], q-
>len);
44.
45.             i = i + q->len;
46.         }
47.         if( i != p->tot_len ){ return 0;} //相等时表明了数据
尾
48.     }
49.
50.     return p;
51. }

```

在本函数的第 15 行，就是调用 ENC28J60.C 文件中的 enc28j60 接收数据包的驱动，enc28j60PacketReceive()函数，它把接收到的数据存放在它的输入参数 Data\_Buf 数组中，并把该数据包的长度以返回值的形式赋值给变量 Len，若 Len 的长度非 0 时，即接收到数据包，就调用 LwIP 的内存管理函数 pbuf\_alloc 申请一个长度为 Len 的 pbuf 存储空间，在 for 循环中把数组 Data\_Buf 的内容转移到 pbuf 的 q->payload 成员中，最后以返回值的形式把 pbuf 交给调用它的函数。

对于最底层 enc28j60 驱动函数的具体实现，请读者利用源码进行分析，它们都是在 ENC28J60.C 文件中定义的。编写的思想就是利用 SPI 接口根据 datasheet 发送不同的命令，控制 enc28j60 芯片，收发数据包时则往它的缓冲区读取或写入数据。

### 36.4.7. 轮询和计时

确保底层驱动正确，并且把 LwIP 接口与驱动关联起来后，还需要在循环中轮询输入函数，给 LwIP 协议栈提供计时。这是在 main 函数中循环调用的 LwIP\_Periodic\_Handle()函数实现的。调用它时，它的输入参数 LocalTime 变量由 systick 定时器每 10ms 更新一次，每次加 10。其具体定义见代码清单 36-11。

代码清单 36-11: LwIP\_Periodic\_Handle()函数

```
1.  /*
2.   * 函数名: LwIP_Periodic_Handle
3.   * 描述   : lwip 协议栈要求周期调用一些函数
4.             tcp_tmr   etharp_tmr   dhcp_fine_tmr dhcp_coarse_tmr
5.   * 输入   : 无
6.   * 输出   : 无
7.   * 调用   : 外部调用
8.   */
9. void LwIP_Periodic_Handle(__IO uint32_t localtime)
10. {
11.     //err_t err;
12.
13.         /* 接收数据包 */
14.     ethernetif_input(&enc28j60);    //轮询是否接收到数
据
15.
16.
17.     /* TCP periodic process every 250 ms */
18.     if (localtime - TCPTimer >= TCP_TMR_INTERVAL)
19.     {
20.         TCPTimer = localtime;
21.         tcp_tmr();    //每 250ms 调用一次
22.     }
23.     /* ARP periodic process every 5s */
24.     if (localtime - ARPTimer >= ARP_TMR_INTERVAL)
25.     {
26.         ARPTimer = localtime;
27.         etharp_tmr();    //每 5s 调用一次
28.     }
29.
30. }
```

本函数在第 14 行调用了 ethernetif\_input()函数，用于查询是否接收到数据包，若接收到数据包则通过内部的机制把它传递给 LwIP 的上层。接下来检查当前 LocalTime 变量与上一次调用了 tcp\_tmr()或 etharp\_tmr()的差值，若时间差达到 250ms 或 5s 时，则调用一次 tcp\_tmr()或 etharp\_tmr()一次。这两个函数分别用于 TCP 协议层和 ARP 模块，这些模块涉及到一些超时重传或其它与时间有关的操作。

## 36.4.8. opt.h 文件和 debug

## ■ 裁剪和配置 LwIP

正确完成了 LwIP 与底层的接口，轮询也配置完毕，还需要配置 LwIP 的 opt.h 文件，它如同 fatfs 文件系统中的 ffconf.h 文件。opt.h 用于裁剪、配置 LwIP 协议栈，如是否使用操作系统，上层的应用是使用 API 或是 RAW 函数等等。部分关于 TCP 配置的一些宏见代码清单 36-12。

代码清单 36-12: TCP 配置宏

```
10. /*
11.  -----
12.  ----- TCP options -----
13.  -----
14. */
15. /**
16.  * LWIP_TCP==1: Turn on TCP.
17.  */
18. #ifndef LWIP_TCP
19. #define LWIP_TCP 1
20. #endif
21.
22. /**
23.  * TCP_TTL: Default Time-To-Live value.
24.  */
25. #ifndef TCP_TTL
26. #define TCP_TTL (IP_DEFAULT_TTL)
27. #endif
28.
29. /**
30.  * TCP_WND: The size of a TCP window. This must be at least
31.  * (2 * TCP_MSS) for things to work well
32.  */
33. #ifndef TCP_WND
34. #define TCP_WND (4 * TCP_MSS)
35. #endif
```

由于 opt.h 文件中这些宏已经配置了默认值，所以大部分是不需要修改的。又由于这里的每个宏都包含条件编译的设置，所以我们在移植 LwIP 协议栈时，通常新建一个名为 lwipopts.h 的头文件，在该文件中修改的需要的配置。

如上面 的 LWIP\_TCP 宏用于使能 TCP 协议或关闭 TCP 协议，它的默认值为 1，表示使能。若我们不需要使用 LwIP 的 TCP 协议时，我们就在 lwipopts.h 文件中添加一个宏，把 LWIP\_TCP 宏定义为 0。

由于 opt.h 文件中的 LWIP\_TCP 宏是在条件编译#ifndef 中，若在其它地方定义了相同的宏，如 lwipopts.h 文件中定义了 LWIP\_TCP 宏，则该宏的参数以 lwipopts.h 文件的定义为准，若在其它地方没有定义相同名字的宏，则以 opt.h 文件的为准。野火在本工程实验中的 lwipopts.h 文件对 LwIP 配置见代码清单 36-13。

代码清单 36-13: lwipopts.h 文件配置

```

1.  /*****
2.  **-----Socket 参数配置: Socket options-----
3.  *****/
4.  #define LWIP_SOCKET 0
5.  // #define LWIP_COMPAT_SOCKETS 0
6.  #define NO_SYS 1 //不使用操作系统
7.
8.  /*****
9.  -----Sequential layer options-----
10. *****/
11. #define LWIP_NETCONN 0
12.
13. #define NO_SYS_NO_TIMERS 1
14.
15. #define LWIP_DHCP 0
16.
17. /*****
18. #define ETHARP_TMR_INTERVAL 5000 /* Time in millis
19. econds to perform ARP processing */
20. *****/
21.
22. /* ----- Memory options ----- */
23. /* MEM_ALIGNMENT: should be set to the alignment of the CPU for
24. which lwIP is compiled. 4 byte alignment -
25. > define MEM_ALIGNMENT to 4, 2
26. byte alignment -> define MEM_ALIGNMENT to 2. */
27. #define MEM_ALIGNMENT 4
28. /* Align memory on 4 byte boundery (32-bit) */
29.
30. /* MEM_SIZE: the size of the heap memory. If the application wi
31. ll send
32. a lot of data that needs to be copied, this should be set high.
33. */
34. #define MEM_SIZE (4*1024)

```

```
35. /* MEMP_NUM_PBUF: the number of memp struct pbufs. If the appli
cation
36.   sends a lot of data out of ROM (or other static memory), thi
s
37.   should be set high. */
38.
39. #define MEMP_NUM_PBUF          10
40.
41. /* MEMP_NUM_UDP_PCB: the number of UDP protocol control blocks.
One
42.   per active UDP "connection". */
43.
44. #define MEMP_NUM_UDP_PCB       6
45.
46. /* MEMP_NUM_TCP_PCB: the number of simulatenously active TCP
47.   connections. */
48.
49. #define MEMP_NUM_TCP_PCB       10
50. /* MEMP_NUM_TCP_PCB_LISTEN: the number of listening TCP
51.   connections. */
52.
53. #define MEMP_NUM_TCP_PCB_LISTEN 6
54.
55. /* MEMP_NUM_TCP_SEG: the number of simultaneously queued TCP
56.   segments. */
57.
58. #define MEMP_NUM_TCP_SEG       12
59.
60. /* MEMP_NUM_SYS_TIMEOUT: the number of simulateously active
61.   timeouts. */
62.
63. #define MEMP_NUM_SYS_TIMEOUT   3
64.
65.
66. /* ----- Pbuf options ----- */
67. /* PBUF_POOL_SIZE: the number of buffers in the pbuf pool. */
68.
69. #define PBUF_POOL_SIZE         10
70.
71. /* PBUF_POOL_BUFSIZE: the size of each pbuf in the pbuf pool. */
/
72.
73. #define PBUF_POOL_BUFSIZE      1500
74. /*****
75. **-----TCP 参数配置: TCP options-----
76. *****/
77.
78. #define MEMP_NUM_REASSDATA      5
79. /* ----- TCP options ----- */
80. #define LWIP_TCP                1
81. #define TCP_TTL                 255
82.
83. /* Controls if TCP should queue segments that arrive out of
84.   order. Define to 0 if your device is low on memory. */
85.
86. #define TCP_QUEUE_OOSEQ        0
87.
88. /* TCP Maximum segment size. */
89.
90. #define TCP_MSS                 (1500 -
40) /* TCP_MSS = (Ethernet MTU - IP header size -
TCP header size) */
91.
92. /* TCP sender buffer space (bytes). */
93.
```



```

94. #define TCP_SND_BUF          (2*TCP_MSS)
95.
96. /* TCP sender buffer space (pbufs). This must be at least = 2 *
97.    TCP_SND_BUF/TCP_MSS for things to work. */
98.
99. #define TCP_SND_QUEUELEN      (6 * TCP_SND_BUF)/TCP_MSS
100.
101.    /* TCP receive window. */
102.
103.    #define TCP_WND              (2*TCP_MSS)
104.
105.
106.    /* ----- ICMP options ----- */
107.    #define LWIP_ICMP            1
108.

```

完成了这部分后(不需要例程 main 中的 `httpd_init()`和 `CMD_init()`函数), 下载程序, 接上网线就可以 ping 通了, 表示 LwIP 移植初步成功。但由于没有建立服务器和 telnet 响应的程序, 这两个功能还不能实现。

## ■ LwIP 的 debug 功能

大部分初始学者, 不熟悉 TCP/IP 协议, 移植 LwIP 十分不容易, 而 LwIP 协议栈有一个十分贴心的输出调试信息的功能。

如在前面提到的 `ethernet_input()`函数的第 19 行有一句宏:

```

1. LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n")
);

```

如果我们设置了输出调试信息使能, 若 `ethernet_input()`中出现错误时就会输出调试信息。这样类似的宏由 LwIP 的设计者精心安排在各种容易出错场合, 以便于移植和调试。宏 `LWIP_DEBUGF` 是在 LwIP 的 `debug.h` 文件定义的, 见代码清单 36-14。

代码清单 36-14: `LWIP_DEBUGF` 宏

```

1.    #define LWIP_DEBUGF(debug, message) do { \
2.        if ( \
3.            ((debug) & LWIP_DBG_ON) && \
4.            ((debug) & LWIP_DBG_TYPES_ON
) && \
5.            ((s16_t)((debug) & LWIP_DBG_
MASK_LEVEL) >= LWIP_DBG_MIN_LEVEL)) { \

```

```
6.             LWIP_PLATFORM_DIAG(message); \
7.             if ((debug) & LWIP_DBG_HALT) {
\
8.                 while(1); \
9.             } \
10.         } \
11.     } while(0)
```

这个宏看似十分复杂，其功能就是当符合某些条件的时候，调用第 6 行中的宏 `LWIP_PLATFORM_DIAG` 输出信息。具体如何输出信息需要我们去实现，本工程在 `cc.h` 文件对该宏设置如下：

```
18. #define LWIP_PLATFORM_DIAG(x) do {printf x;} while(0)
```

即调用 `printf` 函数来输出这些调试信息，而 `printf` 函数已经被我们重定义到串口，所以 PC 的终端就可以接收调试信息了。

在默认的情况下，是不输出调试信息的，详见 `opt.h` 文件相关的 `debug` 宏设置，可以配置不输出调试信息，输出部分调试信息。野火的本实验工程已调试完毕，所以野火把大部分输出调试信息的功能都关了。本例程中在 `lwipopts.h` 文件中设置的与调试信息相关的宏，见代码清单 36-15。

代码清单 36-15：与调试信息相关的宏

```
1.  /*****
2.  #define LWIP_DEBUG                1
3.  #define LWIP_DBG_TYPES_ON        LWIP_DBG_OFF
4.
5.  /**
6.   * ETHARP_DEBUG: Enable debugging in etharp.c.
7.   */
8.  #define ETHARP_DEBUG                LWIP_DBG_OFF
9.
10. /**
11.  * NETIF_DEBUG: Enable debugging in netif.c.
12.  */
13. #define NETIF_DEBUG                LWIP_DBG_OFF
14.
15.
16. /**
17.  * PBUF_DEBUG: Enable debugging in pbuf.c.
18.  */
19. #define PBUF_DEBUG                LWIP_DBG_OFF
```

若需要输出调试信息，必须把宏 `LWIP_DEBUG` 设置为 1，其它的如 `NETIF_DEBUG`、`ETHARP_DEBUG` 可根据调试的情况决定是否开启。

### 36.4.9. LwIP 应用

#### ■ 网络应用软件架构

构成网络应用的软件有不同的结构，有 B/S 结构(浏览器/服务器) 和 C/S 结构(客户端/服务器)。

基于 B/S 结构的应用程序以网页形式存放于服务器上，用户运行应用程序时通过 web 调用服务器的应用程序，并通过浏览器把结果显示给用户，用户只需要浏览器即可运行所有服务器中提供的应用模块。

而 C/S 结构的软件需要针对不同的应用而对客户端进行更改。本实验工程中的 http 服务器是基于 B/S 结构的，而 telnet 程序是基于 C/S 结构的。

#### ■ TCP 网络应用

基于 TCP 协议的网络应用十分常见。TCP 应用的服务器流程如图 36-13，例如要把 STM32 作为网页服务器，其程序就是根据该流程编写的。

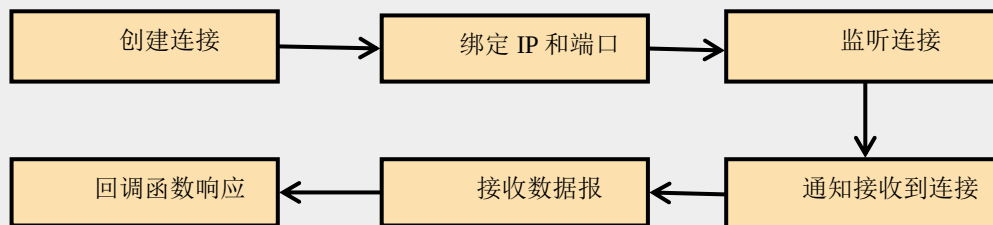


图 36-13 建立 TCP 应用流程

当服务器开始监听连接时，客户端或浏览器就可以向服务器提出连接请求，然后服务器作出响应。

## ■ LwIP 的应用函数

移植好 LwIP 后，就可以利用它来编写应用层的程序了。LwIP 协议栈支持类似 BSD 协议栈的 **socket** 应用程序接口，这种被称为 LwIP 的 API 函数，API 函数更适合用在具有操作系统的平台上。

另一种方式是直接调用 LwIP 协议栈内部的函数，这些函数被称为 **RAW** 函数，RAW 函数适合于用在没有操作系统的平台，效率高，但开发起来较为复杂。由于本例程中没有使用操作系统，所以我们利用 LwIP 的 RAW 函数进行应用层编程。

利用 LwIP 编写的 TCP 类型应用程序，它在各个层次的处理过程如图 36-14。

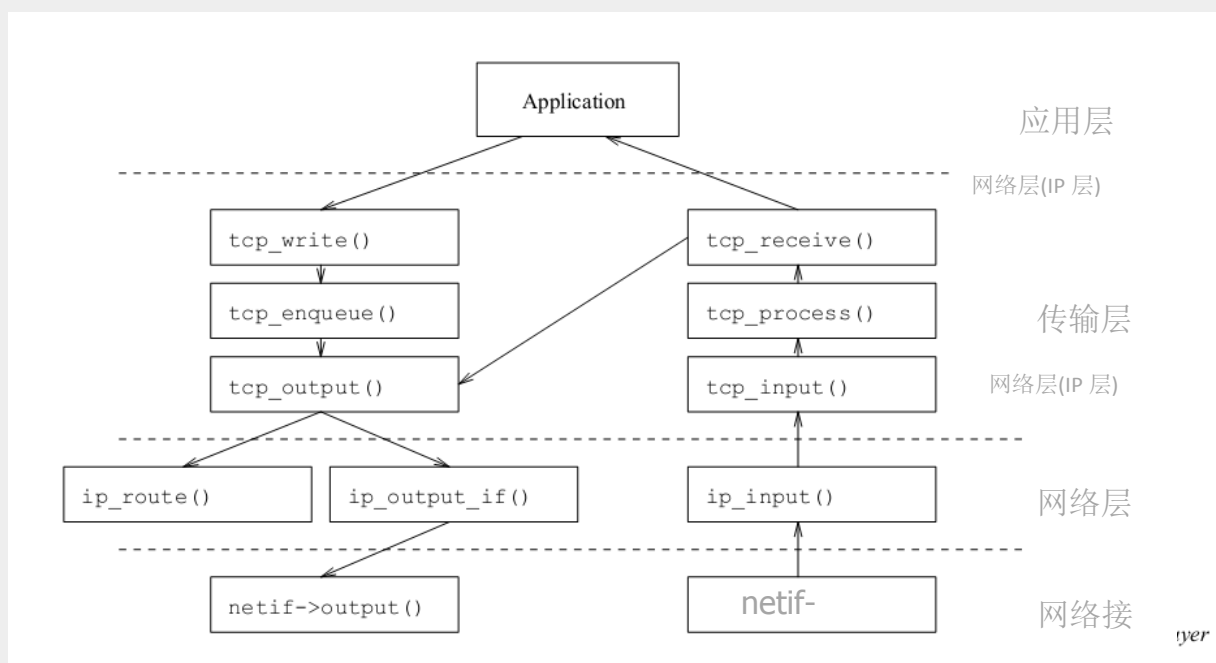


图 36-14 TCP 处理过程

在移植的时候，我们主要完成的就是图中网络接口层的 **netif->output** 和 **netif->input** 函数。移植完成后，即可利用 LwIP 的应用层接口编写应用程序，而网络层和传输层则由 LwIP 协议栈完成了。不过，大部分 RAW 函数都是位于传输层的。

### 36.4.10. 网页服务器

利用 LwIP 编写网页服务器是在 main 中调用的 httpd\_init() 函数实现的。该函数的源码见代码清单 36-16。

代码清单 36-16: httpd\_init() 函数

```
1.  /*
2.  * 函数名: httpd_init
3.  * 描述   : 初始化 web server, 初始化后才能显示网页
4.  * 输入   : 无
5.  * 输出   : 无
6.  * 调用   : 外部调用
7.  */
8. void httpd_init(void)
9. {
10.     struct tcp_pcb *pcb;
11.
12.     pcb = tcp_new();           /* 建立通信的 TCP 控制块
(pcb) */
13.
14.     tcp_bind(pcb, IP_ADDR_ANY, 80); /* 绑定本地 IP 地址和端口号 */
15.
16.     pcb = tcp_listen(pcb);      /* 进入监听状态 */
17.
18.     tcp_accept(pcb, http_accept); /* 设置有连接请求时的回调函
数 */
19.
20. }
```

httpd\_init() 函数只有几行，它每调用一个函数，就完成了 TCP 服务器流程图中的一个步骤。分析如下：

#### (1) tcp\_new()

本函数的原型为：

```
1. struct tcp_pcb * tcp_new(void)
```

它用于创建一个 TCP 连接。它向 LwIP 协议栈申请分配一个 TCP 控制块，若分配成功，则返回控制块的指针，否则返回 NULL。TCP 控制块结构体用 tcp\_pcb 进行定义，它是用于描述 TCP 连接的，包含了非常丰富的信息。

本实验中调用 tcp\_new() 后，把它的返回值赋给 TCP 控制块类型的指针 pcb。

#### (2) tcp\_bind()

本函数原型为：

```
err_t  
tcp_bind(struct tcp_pcb *pcb, struct ip_addr *ipaddr, u16_t port)
```

它用于把一个本地的 IP 和端口进行绑定，它的输入参数分别为要进行绑定的连接、绑定的 IP 地址、端口号。

本实验中把 `tcp_new()` 新建的连接 `pcb` 进行绑定，绑定的 IP 地址为 `IP_ADDR_ANY`，这个宏展开为 `0x00000000`，绑定的端口为 `80`，表示把本地的任意 IP 地址都与 `80` 端口绑定。网页浏览器默认使用 `80` 端口。

### (3) tcp\_listen()

它实际是一个宏，宏展开是一个函数

```
1. #define tcp_listen(pcb) tcp_listen_with_backlog(pcb, TCP_DEFAULT_LISTEN_BACKLOG)
```

函数的原型如下：

```
(7) struct tcp_pcb *  
(8) tcp_listen_with_backlog(struct tcp_pcb *pcb, u8_t backlog)
```

它用于监听端口，若接收到连接，则会调用下面 `tcp_accept()` 函数中指定中的回调函数进行响应，返回值为新的 TCP 控制块，用于回调函数的输入参数。

本实验调用 `tcp_listen()` 对以上创建的 `80` 端口进行监听。

### (4) tcp\_accept()

函数的原型为：

```
1. void  
2. tcp_accept(struct tcp_pcb *pcb, err_t (* accept)(void *arg, struct tcp_pcb *newpcb, err_t err))
```

它的作用是用于指定连接建立后调用的回调函数。第一个输入参数为 TCP 控制块，用于指定是哪个连接，第二个参数是回调函数的指针。

本实验中调用 `tcp_accept()` 指定回调函数为 `http_accept`。

当我们在 PC 的浏览器的地址栏输入：<http://192.168.1.18:80>（:80 表示 80 端口，可省略），由于服务器 STM32 的监听设置，连接成功，就会调用 tcp\_accept() 指定的 http\_accept 函数。http\_accept() 函数定义见代码清单 36-17。

代码清单 36-17: http\_accept() 函数

```

86. /*
87. * 函数名: http_accept
88. * 描述   : http web server 的回调函数，建立连接后调用
89. * 输入   : tcp_arg 设置的参数、pcb、err
90. * 输出   : err
91. * 调用   : 内部调用
92. */
93. static err_t http_accept(void *arg, struct tcp_pcb *pcb, err_t err)
94. {
95. /* 设置回调函数优先级，当存在几个连接时特别重要，此函数必须调用 */
96. tcp_setprio(pcb, TCP_PRIO_MIN);
97.
98. tcp_recv(pcb, http_recv); /* 设置 TCP 段到时的回调函数 */
99.
100. return ERR_OK;
101. }

```

http\_accept() 函数又调用了 tcp\_setprio 进行优先级设置，并调用 tcp\_recv 指定当接收到新的数据的时候调用 http\_recv 函数。在本实验中我们使用 http\_recv() 函数向浏览器输出网页，它的定义见代码清单 36-18。

代码清单 36-18: http\_recv() 函数

```

1. /*
2. * 函数名: http_recv
3. * 描述   : http 接收到数据后的回调函数
4. * 输入   : tcp_arg 设置的参数、tcp_pcb、pcb、err
5. * 输出   : err
6. * 调用   : 内部调用
7. */
8.
9. static err_t http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err)
10. {
11.     char *data = NULL;
12.     char *UserName = NULL;
13.     char *PassWord = NULL;
14.     char *LED_CMD = NULL;
15.     char *ch = NULL;
16.
17.     data = p->payload; //把接收到的数据指针交给 data
18.
19.     if (err == ERR_OK && p != NULL) //判断是否非空

```

```
20.     {
21.         if(strncmp(data,"GET",3)==0)
22.         {
23.
24.             /*第一次与服务器连接, 打开登录页面, 未输入登录信息*/
25.             tcp_write(pcb,http_html_hdr,sizeof(http_html_hdr),0);
26.             /* 发送 http 协议头部信息 */
27.             tcp_write(pcb,login,sizeof(login),0);
28.             /* 发送登录网页信息 */
29.         }
30.         else if(strncmp(data,"POST",4)==0)
31.         {
32.             LED_CMD = strstr(data,"LED_CTRL=");
33.             UserName = strstr(data,"UserName=");
34.             //截取 UserName=
35.             PassWord = strstr(UserName,"PassWord=");
36.             //截取 PassWord=
37.             /*输入了用户名和密码*/
38.             if(UserName!=NULL && PassWord!=NULL)
39.             {
40.                 ch = strchr(UserName,'&');
41.                 //把&替换为'\0'
42.                 *ch = '\0';
43.                 UserName +=sizeof("UserName=")-1;
44.                 //提取用户名。sizeof 字符串包括终止符'\0'
45.                 PassWord += sizeof("PassWord=")-1;
46.                 //提取密码
47.
48.                 if(strcmp(UserName,"wildfire")==0 && strncmp(PassWord,"123456",6)==0) /*输入的用户名和密码正确*/
49.                 {
50.                     LED1(ON);
51.                     // printf("\r\n 提取出的用户名 =%s\r\n 提取出的密码 =%s",UserName,PassWord);
52.
53.                     tcp_write(pcb,http_html_hdr,sizeof(http_html_hdr),0); /*
54.                     发送 http 协议头部信息 */
55.                     tcp_write(pcb,led_ctrl_on,sizeof(led_ctrl_on),0);
56.                     /* 发送 led 控制网页信息 */
57.                 }
58.                 else /*输入的用户名和密码错误*/
59.                 {
60.                     tcp_write(pcb,http_html_hdr,sizeof(http_html_hdr),0); /* 发送 http 协议头部信息 */
61.                     tcp_write(pcb,login,sizeof(login),0); /* 发送登录网页信息 */
62.                 }
63.                 else if(LED_CMD !=NULL) /*接收到 LED 控制命令*/
64.                 {
65.                     if(strstr(LED_CMD,"LED_CTRL=ON")) /*
66.                     检测是哪个命令: 开\关*/
67.                     {
68.                         LED1(ON);
```



```

68.         tcp_write(pcb,http_html_hdr,sizeof(
http_html_hdr),0);    /* 发送http 协议头部信息 */
69.         tcp_write(pcb,led_ctrl_on,sizeof(led
d_ctrl_on),0);    /* 发送led 控制网页信息 */
70.     }
71.     else if(strstr(LED_CMD,"LED_CTRL=OFF"))

72.     {
73.         LED1(OFF);
74.         tcp_write(pcb,http_html_hdr,sizeof(
http_html_hdr),0);    /* 发送http 协议头部信息 */
75.         tcp_write(pcb,led_ctrl_off,sizeof(led
ctrl_off),0);    /* 发送led 控制网页信息 */
76.     }
77. }
78. }
79.
80.     pbuf_free(p);    /* 释放该 pbuf 段 */
81. }
82. tcp_close(pcb);
    /* 关闭这个连接 */
83. err = ERR_OK;
84.
85. return err;
86. }

```

该函数看似十分复杂，其实只是重复判断接收到什么数据，然后根据数据的内容调用 `tcp_write()` 函数发送不同的网页。其流程如下：

(1) 第 17 行，把接收到的数据包 `p->payload` 的指针赋值给 `data`，在后面使用 `data` 来判断接收到的数据。

(2) 第 19 行，若数据包非空，则开始判断数据包的内容。

(3) 当我们在浏览器输入服务器的 IP 地址并访问时，浏览器会向服务器发送一些数据，我们可以使用一些网络抓包工具查看这些数据包。其数据内容见图 36-15 中的箭头所指的区域。数据包中开头的内容为“GET”。在服务器中我们以这三个字符作为第一次与服务器连接的标志，当确定为第一次连接时，调用 LwIP 的 `tcp_write()` 函数输出网页。

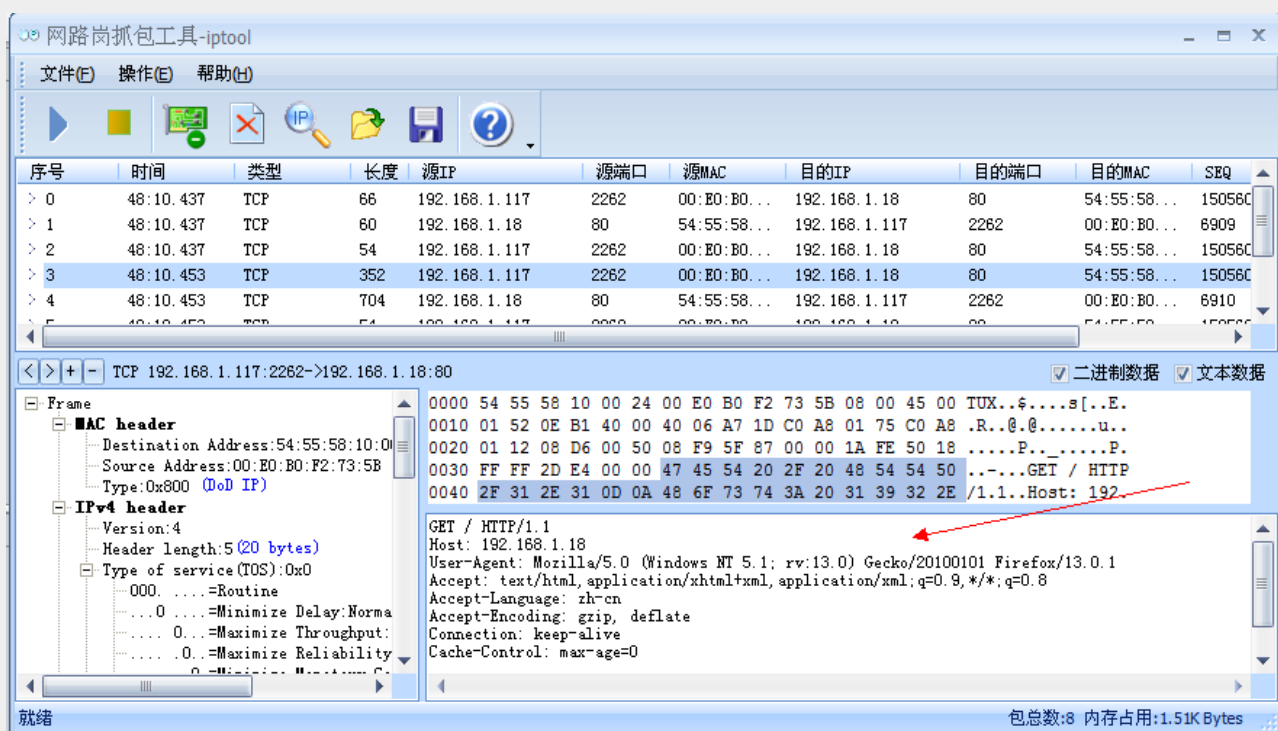


图 36-15 浏览器发送的数据

(4) 第 25 行，调用 `tcp_write()` 函数，输出参数 `http_html_hdr` 数组的内容，这部分内容的大小为 `sizeof(http_html_hdr)`。`http_html_hdr` 数组是 `http` 网页的文件头部信息，在输出网页的时候我们需要先发送这部分内容。见代码清单 36-19。

代码清单 36-19: `http` 文件头

```
/*http 文件头 */  
  
#define  
  
HTTP_HEAD "HTTP/1.1 200 OK\r\nContent-type: text/html\r\n\r\n"
```

(5) 第 26 行，调用 `tcp_write()` 函数，输出 `login` 数组内容 `login` 数组存储的就是登录页面。由于没有使用文件系统，所以在本实验中 `html` 格式的网页文件都是以 `c` 语言的数组存储起来的。

(6) 登录页面后的代码作用是类似的，只是浏览器向服务器发送的数据不同，而服务器响应的网页也有所区别而已。服务器接收到特定的 `LED` 控制的信息后，`STM32` 对 `LED` 进行控制。建议读者了解一下 `html` 语言，

本实验中重点用到的是 html 语言中用于提交表单的 GET 方法和 POST 方法。

经过以上步骤，我们就成功建立了一个网页服务器，实验中还有一个针对 telnet 程序的服务，它是在 CMD\_init()函数中完成的，编写的思路和搭建网页服务器十分相似。

#### 36.4.11. 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，利用网线把 STM32 开发板接入与 PC 相同的路由，也可以直接利用网线把开发板和 PC 相连，其实验的操作是相同的(这样可以排除路由的问题)，但在进行浏览网页实验时，图片可能无法正常显示。把本工程文件编译后烧录到开发板上，在程序运行框输入 cmd 命令进入 dos 模式。

a)ping 实验，见图 36-16 ping 192.168.1.18。

在命令提示符窗口输入命令并回车： ping 192.168.1.18

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\flyleaf>ping 192.168.1.18

Pinging 192.168.1.18 with 32 bytes of data:

Reply from 192.168.1.18: bytes=32 time=1ms TTL=255
Reply from 192.168.1.18: bytes=32 time<1ms TTL=255
Reply from 192.168.1.18: bytes=32 time=1ms TTL=255
Reply from 192.168.1.18: bytes=32 time<1ms TTL=255

Ping statistics for 192.168.1.18:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms

C:\Documents and Settings\flyleaf>
```

图 36-16 ping 192.168.1.18

## (2) telnet 实验。

1) 如果使用 windows 7 系统，系统没有 telnet 程序，需要自行下载安装。使用 xp 系统的用户，在命令提示符窗口输入命令并回车：

telnet 192.168.1.18

输入命令后弹出如下窗口：



图 36-17 进入 telnet 程序

2) 见图 36-18，在弹出的窗口下输入用户名并回车： wildfire

3) 若用户名正确，程序提示输入密码，键入密码并回车：  
123456

4) 若密码正确，提示输入命令，本工程只允许两条命令，分别为 LED1\_ON 和 LED1\_OFF，用于控制 LED1 的亮和灭。

输入命令： LED1\_ON

板上的 LED1 灯会被点亮，窗口会弹出控制成功的信息，并且提示输入命令。

输入命令： LED1\_OFF

板上的 LED1 会被关灭，窗口弹出控制成功信息，再次提示输入命令。



图 36-18 telnet 控制流程

若用户输入的用户名、密码不正确或不存在的命令，会出现各种提示，并可以重新输入。

(5) 网页浏览实验。(若 PC 没有接入互联网，图片可能没法正常显示。)

26. 打开浏览器，在地址栏输入 IP 并回车：192.168.1.18

在弹出的网页中输入用户名和密码：wildfire 123456



图 36-19 网页登录

27. 点击登录后，出现如下界面，且开发板上的 LED 被点亮。



图 36-20 登录后的页面

28. 点选关闭 LED1，并点击控制按钮，网页显示的 LED 状态改变，板上的 LED1 也被关灭。



图 36-21 关闭 LED



## 37. SRM

## 38. NANDFLASH

## 39. NORFLASH

## 40. 485 通信

## 41. DHT11 温湿度传感器

## 42. DS1302 时钟实验

## 43. PS2 接口

## 野火 ISO 系统进阶篇



## 44. uCOS

## 45. 前言

uC/OS 是一个微型的实时操作系统，包括了一个操作系统最基本的一些特性，如任务调度、任务通信、内存管理、中断管理、定时管理等。而且这是一个代码完全开放的实时操作系统，简单明了的结构和严谨的代码风格，非常适合初涉嵌入式操作系统的人士学习。

很多人在学习 STM32 中，都想亲自移植一下 uC/OS，而不是总是用别人已经移植好的。在我学习 uC/OS 的过程中，查找了很多资料，也看过很多关于如何移植 uC/OS 到 STM32 处理器上的教程，但都不尽人意，主要是写得太随意了，思路很乱，读者看到最后还是不确定该怎样移植。为此，我决定写这个教程，让广大读者真正了解怎样移植。

学前建议：C 语言 + 数据结构

Wildfire Team

2011 年 11 月 3 日

## 46. 1、官方源代码介绍

首先我们下载源代码，官方下载地址：

<http://micrium.com/page/downloads/ports/st/stm32> （下载资料需要注册帐号）

或者网盘下载：<http://dl.dbank.com/c0jnhmfxcp>

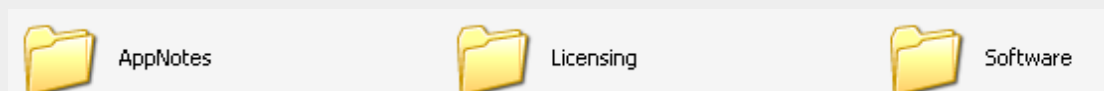
我们需要下载的就是下面这个，因为我用到的开发板芯片是 STM32F103VET6

| Download                                                                                                                                                             | Processor                | OS version | Compiler                           | Contributor    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|------------|------------------------------------|----------------|
| <a href="#">Download</a> <br>see STM3210B-EVAL<br>see STM3210E-EVAL<br>see STM32-SK | STM32 (Cortex-M3)        | V2.86      | <a href="#">IAR &amp; ARM/Keil</a> | <b>Micrium</b> |
| <a href="#">Download</a> <br>see STM32F103ZE-SK                                   | <u>STM32 (Cortex-M3)</u> | V2.86      | <u>IAR</u>                         | <b>Micrium</b> |
| <a href="#">Download</a> <br>see uC-Eval-STM32F107                                | STM32F107                | v2.92      | IAR V6.10.5                        | <b>Micrium</b> |

注意：下载的源代码开发环境是 IAR 编译器的。

我们使用的 uCOS 是 2.86 版本。

下载解压后可以看到 Micrium 含有三个文件夹：



| 文件名              | 说明                                                                                                                                           |                  |                                                        |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------|------------------|--------------------------------------------------------|
| <b>AppNotes</b>  | 包含 uCOS-II 的说明文件，其中文件 <b>Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf</b> 是很重要的。这个文件对 uC/OS 在 M3 内核移植过程中需要修改的代码做了详细的说明。 |                  |                                                        |
| <b>Licensing</b> | 包含了 uCOS-II 使用许可证                                                                                                                            |                  |                                                        |
| <b>Software</b>  | 应用软件，我们这里用到的就是 uCOS-II 文件夹。在整个移植过程中我们只需用到 uCOS-II 下的两个文件，分别是 Ports 和 Source.                                                                 |                  |                                                        |
|                  | <b>uCOS-II</b>                                                                                                                               | <b>Doc</b>       | uC/OS 官方自带说明文档和教程                                      |
|                  |                                                                                                                                              |                  | 官方移植到 M3 的移植文件（IAR 工程）                                 |
|                  |                                                                                                                                              | <b>cpu.h</b>     | 定义数据类型、处理器相关代码、声明函数原型                                  |
|                  |                                                                                                                                              | <b>Ports</b>     |                                                        |
|                  |                                                                                                                                              | <b>cpu_c.c</b>   | 定义用户钩子函数，提供扩充软件功能的入口点。（所谓钩子函数，就是指那些插入到某函数中拓展这些函数功能的函数） |
|                  |                                                                                                                                              | <b>cpu_a.asm</b> | 与处理器相关汇编函数，主要是任务切换函数                                   |
|                  |                                                                                                                                              | <b>os_dbg.c</b>  | 内核调试数据和函数                                              |
|                  |                                                                                                                                              |                  | uC/OS 的源代码文件                                           |
|                  | <b>Source</b>                                                                                                                                | <b>ucos_ii.h</b> | 内部函数参数设置                                               |
|                  |                                                                                                                                              | <b>os_core.c</b> | 内核结构管理，uC/OS 的核心，包含了内核初始化，任务切换，事件块管理、事件标志组管理等功能。       |

|                   |                                                |                 |                   |                               |
|-------------------|------------------------------------------------|-----------------|-------------------|-------------------------------|
|                   |                                                |                 | <b>os_time.c</b>  | 时间管理，主要是延时                    |
|                   |                                                |                 | <b>os_tmr.c</b>   | 定时器管理，设置定时时间，时间到了就进行一次回调函数处理。 |
|                   |                                                |                 | <b>os_task.c</b>  | 任务管理                          |
|                   |                                                |                 | <b>os_mem.c</b>   | 内存管理                          |
|                   |                                                |                 | <b>os_sem.c</b>   | 信号量                           |
|                   |                                                |                 | <b>os_mutex.c</b> | 互斥信号量                         |
|                   |                                                |                 | <b>os_mbox.c</b>  | 消息邮箱                          |
|                   |                                                |                 | <b>os_q.c</b>     | 队列                            |
|                   |                                                |                 | <b>os_flag.c</b>  | 事件标志组                         |
| <b>CPU</b>        | STM32 标准外设库                                    |                 |                   |                               |
| <b>EvalBoards</b> | micrium 官方评估板的代码                               |                 |                   |                               |
|                   | <b>OS-Probe-LCD</b>                            | <b>os_cfg.h</b> | 内核配置              |                               |
| <b>uC-CPU</b>     | 基于 micrium 官方评估板的 CPU 移植代码                     |                 |                   |                               |
| <b>uC-LIB</b>     | micrium 官方的一个库代码                               |                 |                   |                               |
| <b>uC-Probe</b>   | uC-Probe 有关的代码，是一个通用工具，能让嵌入式开发人员在实时环境中监测嵌入式系统。 |                 |                   |                               |

以上这些都是下载下来的官方资源。有没有发现，uC/OS 的代码文件都被分开放到不同的文件夹里了？呵呵，这个是官方移植好到 STM32 的 uC/OS 系统，他已经帮我们对 uC/OS 的文件进行分类存放。如果你不想要移植好的，也可以下载没有移植的，那样就所以文件都放在一个文件夹里。

下载地址：<http://micrium.com/download/Micrium-uCOS-II-V290.ZIP>

提示一下，如果是没移植好的，是找不到 `main` 函数的哦！初学者，相信很多都下载没移植好的，然后直接看它的源代码，然后看到头晕也找不到工程的入口。其实，`uC/OS` 就是一个库而已，熟悉它的运行流程和函数接口，就可以基本跑起来。

在自己亲自移植之前，总是看到移植好的例程包含有 `CPU`、`uC-CPU`、`uC-LIB`、`uCOS-II` 四个文件夹下的代码。`uCOS-II` 文件夹下的是源代码，这个好理解；但是前面三个有什么用啊？

通常看其他移植教程时，一般都说只需改 `os_cpu.h`，`os_cpu_a.asm` 和 `os_cpu_c.c` 就可以了，就没听说过有 `CPU`、`uC-CPU`、`uC-LIB` 这些的。心中一直很纳闷，难道后三个都要自己编写的吗？后来在上面网址把源代码下载后，才知道 `CPU`、`uC-CPU`、`uC-LIB` 这三个文件是官方自己写的移植文件，而我们使用了标准外设库 `CMSIS` 中提供的启动文件及固件库了，因此可以不用这三个文件，哈哈，心中的疑团解决了！

先看一下开发板与 `uC/OS-II` 的框架图（注意 `APP.C` 就是 `main` 文件，我们下面移植的文件并没有 `APP_VECT.C` 这个文件，应用文件可以灵活处理的）

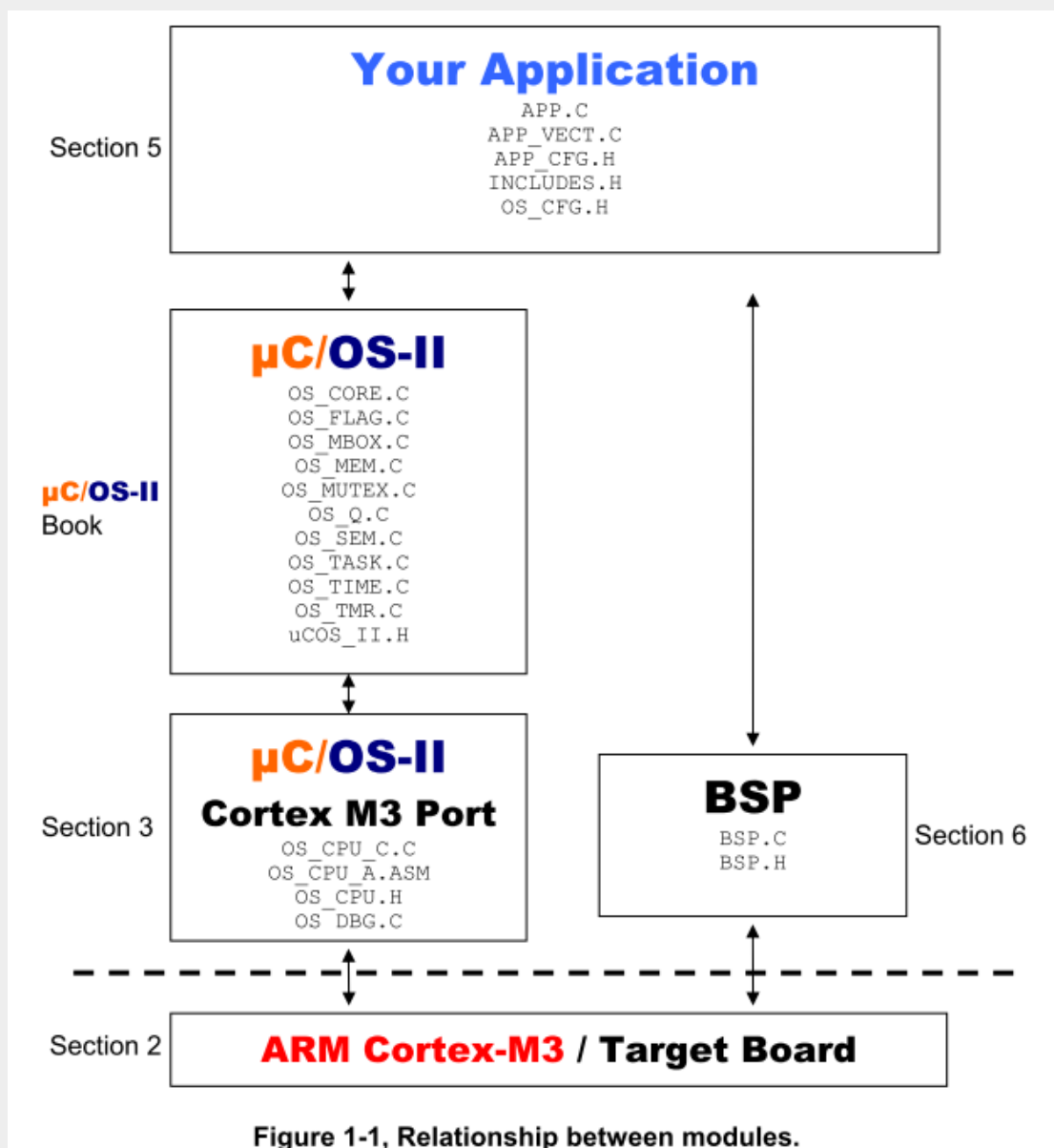


Figure 1-1, Relationship between modules.

## 47. 2、重要文件代码详解

移植前，我们需要先了解一下 uC/OS 的重要文件代码。

对于从没接触过 uC/OS 或者其他嵌入式系统的朋友们，你们需要先了解 uC/OS 的工作原理和各模块功能，不然就不知道为啥这样移植。

### 推荐教程

| 作者                   | 书名                                           | 推荐理由                                                                 |
|----------------------|----------------------------------------------|----------------------------------------------------------------------|
| 野火团队                 | 初探 uCOS-II                                   | 清晰简单地讲解了 uC/OS 的运行流程，方便初学者学习。                                        |
| 任哲                   | 嵌入式实时操作系统<br>uC/OS-II 原理及应用<br>(北京航空航天大学出版社) | 通俗易懂的一本 uC/OS 教程，非常适合初学者学习。<br><br>不过教程没得到更新，不能适应 uC/OS 的发展，但还是值得推荐。 |
| Joseph Yiu 著<br>宋岩 译 | Cortex-M3 权威指南                               | 呵呵，不用说吧？移植 uC/OS 到 M3 内核中，怎么能不了解内核呢？                                 |

下面的内容主要来自于刚才下载的文件里面的 [Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf](#) 文件来讲的，因为这文件是 uC/OS 作者移植 uC/OS 到 STM32 的移植手册，里面谈到很多移植说需要注意的事项和相关知识。我在这里添加也按照作者的思路来讲解，并加入个人理解，如果有误，欢迎指出错误。

## 47.1. 2.1 os\_cpu.h

定义数据类型、处理器相关代码、声明函数原型

### 47.1.1. 全局变量

OS\_CPU\_GLOBALS 和 OS\_CPU\_EXT 允许我们是否使用全局变量。

```
6.  #ifndef OS_CPU_GLOBALS
7.  #define OS_CPU_EXT
8.  #else                                     //如果没有定义 OS_CPU_GLOBALS
9.  #define OS_CPU_EXT extern                //则用 OS_CPU_EXT 声明变量已经外部定义了。
10. #endif
```

### 47.1.2. 数据类型

```
11. typedef unsigned char  BOOLEAN;
12. typedef unsigned char  INT8U;
13. typedef signed  char   INT8S;
14. typedef unsigned short INT16U;           //大多数 Cortex-M3 编译器, short 是 16
    位,int 是 32 位
15. typedef signed  short  INT16S;
16. typedef unsigned int   INT32U;
17. typedef signed  int    INT32S;
18. typedef float          FP32;             //尽管包含了浮点数, 但 uC/OS-II 中并没用
    到
19. typedef double         FP64;
20.
21. typedef unsigned int    OS_STK;           //M3 是 32 位, 所以堆栈的数据类型 OS_STK
    设置 32 位
22. typedef unsigned int    OS_CPU_SR;       //M3 的状态寄存器 (xPSR) 是 32 位
```

### 47.1.3. 临界段

临界段, 就是不可被中断的代码段, 例如常见的入栈出栈等操作就不可被中断。

uC/OS-II 是一个实时内核, 需要关闭中断进入和开中断退出临界段。为此, uC/OS-II 定义了两个宏定义来关中断 OS\_ENTER\_CRITICAL()和开中断 OS\_EXIT\_CRITICAL()。

```
23. #define OS_CRITICAL_METHOD    3          //进入临界段的三种模式, 一般选择第 3 种, 即这里
    设置为 3
24.
25.
26. #define OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();} //进入临界段
27. #define OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);} //退出临界段
```



事实上，有 3 种开关中断的方法，根据不同的处理器选用不同的方法。大部分情况下，选用第 3 种方法。

另外，关于汇编函数 `OS_CPU_SR_Save()` 和 `OS_CPU_SR_Restore()`，在后面谈到 `os_cpu_a.asm` 文件时会再说。

#### 47.1.4. 栈生长方向

M3 的栈生长方向是由高地址向低地址增长的，因此 `OS_STK_GROWTH` 定义为 1。

```
28.  #define OS_STK_GROWTH 1
```

#### 47.1.5. 任务切换宏

定义任务切换宏，关于汇编函数 `OSCtxSw()`，在后面谈到 `os_cpu_a.asm` 文件时会再说。

```
29.  #define OS_TASK_SW() OSCtxSw()
```

#### 47.1.6. 函数原型

##### ■ 开中断和关中断

如果定义了进入临界段的模式为 3，就声明开中断和关中断函数

```
30.  #if OS_CRITICAL_METHOD == 3
    31. OS_CPU_SR OS_CPU_SR_Save(void);
32.  void OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
33.  #endif
```

##### ■ 任务管理函数

```
34. /*****任务切换的函数*****/  
35. void OSCtxSw(void); //用户任务切换  
36. void OSIntCtxSw(void); //中断任务切换函数  
37. void OSStartHighRdy(void); //在操作系统第一次启动的时候调用  
    的任务切换  
38.  
39. void OS_CPU_PendSVHandler(void); //用户中断处理函数，旧版本为  
    OSPendSV  
40.  
41. void OS_CPU_SysTickHandler(void); //系统定时中断处理函数，时钟  
    节拍函数  
42. void OS_CPU_SysTickInit(void); //系统 SysTick 定时器初始化  
43.  
44. INT32U OS_CPU_SysTickClkFreq(void); //返回 SysTick 定时器的时钟频率
```

这三个函数是为  
SysTick 定时器服务的

关于任务切换，利用到异常处理知识，可以看《Cortex-M3 权威指南》（Joseph Yiu 著 宋岩译）中第 3.4 小节。

关于 PendSV，有不懂的朋友，可以看《Cortex-M3 权威指南》中第 7.6 小节 SVC 和 PendSV：

SVC（系统服务调用，亦简称系统调用）和 PendSV（可悬起系统调用），它们多用在上了操作系统的软件开发中。

SVC 用于产生系统函数的调用请求，SVC 异常是必须在执行 SVC 指令后立即得到响应的。PendSV（可悬起的系统调用）则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行。

PendSV 的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：

11. 执行一个系统调用

12. 系统滴答定时器（SysTick）中断。（轮转调度中需要）

注：此部分内容出自《Cortex-M3 权威指南》

关于 SysTick 定时器的三个函数，为了便于理解，我们把它注释掉，不采用官方的，自己编写：

#### 需要注释的函数

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| OS_CPU_SysTickHandler() | 在 os_cpu.c 中定义，是 SysTick 中断的中断处理函数，而在 stm32f10x_it.c 中已经有该中断函数的定义 SysTick_Handler()，这里也就不需要了。 |
| OS_CPU_SysTickClkFreq() | 定义在 BSP.C 中，此函数我们自己会编写，把它注释掉。                                                                 |
| OS_CPU_SysTickInit()    | 定义在 os_cpu.c 中，用于初始化 SysTick 定时器，它依赖于 OS_CPU_SysTickClkFreq()，也要注释掉。                          |

## 47.2. 2.2 os\_cpu.c

移植 uC/OS 时，我们需要写 10 个相当简单的 C 函数：9 个钩子函数和 1 个任务堆栈结构初始化函数。

### 47.2.1. 钩子函数

所谓钩子函数，指那些插入到某些函数中为扩展这些函数功能的函数。一般地，钩子函数为第三方软件开发人员提供扩充软件功能的入口点。为了拓展系统功能，uC/OS-II 中提供有大量的钩子函数，用户不需要修改 uC/OS-II 内核代码程序，而只需要向钩子函数添加代码就可以扩充 uC/OS-II 的功能。

注：此部分内容出自 张勇的《嵌入式操作系统原理与面向任务程序设计——基于 uC/OS-II v2.86 和 ARM920T》

尽管 uC/OS-II 中提供了大量的钩子函数，但实际上，移植时我们需要编写的也就 9 个钩子函数：

```
45.OSInitHookBegin()           //OSInit()           系统初始化函数开头的钩子函数
46.OSInitHookEnd()            //OSInit()           系统初始化函数结尾的钩子函数
```

```

47. OSTaskCreateHook()           //OSTaskCreate() 或 OSTaskCreateExt() 创建任务
钩子函数
48. OSTaskDelHook()             //OSTaskDel()           删除任务钩子函数
49. OSTaskIdleHook()            //OS_TaskIdle() 空闲任务钩子函数
50. OSTaskStatHook()            //OSTaskStat()          统计任务钩子函数
51. OSTaskSwHook()              //OSTaskSW()           任务切换钩子函数
52. OSTCBInitHook()             //OS_TCBInit()        任务控制块初始化钩子函
数
53. OSTimeTickHook()            //OSTaskTick()          时钟节拍钩子函数

```

这些函数都是一些钩子函数，一般由用户拓展。如果要用到这些钩子函

数，需要在 OS\_CFG.H 中定义 OS\_CPU\_HOOKS\_EN 为 1，即：

```

54. #define OS_CPU_HOOKS_EN 1 //在 OS_CFG.H 中定义

```

钩子函数的编写，例如：

```

55. /*** 系统初始化函数 OSInit() 开头调用 ***/
56. void OSInitHookBegin (void)
57. {
58.     #if OS_TMR_EN > 0           //当使用 OS_TMR.C 定时器管理模块
59.         OSTmrCtr = 0;          //初始化系统节拍计数变量 OSTmrCtr 为 0
60.                                 //每个时钟节拍 OSTmrCtr（全局变量，初始
值为 0）增 1
61.     #endif
62. }

```

```

63. /*** 创建任务 OSTaskCreate() 或 OSTaskCreateExt() 中调用 ***/
64. void OSTaskCreateHook (OS_TCB *ptcb)
65. {
66.     #if OS_APP_HOOKS_EN > 0    //如果有定义应用任务
67.         App_TaskCreateHook(ptcb); //调用应用任务创建钩子函数
68.     #else                      //否则
69.         (void)ptcb;             //告诉编译器 ptcb 没用到
70.     #endif
71. }

```

```

72. /*** 切换任务时被调用 ***/
73. void OSTaskSwHook (void)
74. {
75.     #if OS_APP_HOOKS_EN > 0
76.         App_TaskSwHook();       //应用任务切换时调用的钩子函数
77.     #endif
78. }

```

```

79. /*** 每个系统节拍到了 ***/
80. void OSTimeTickHook (void)
81. {
82.     #if OS_APP_HOOKS_EN > 0
83.         App_TimeTickHook();     //应用软件的时钟节拍钩子
84.     #endif
85.
86.     #if OS_TMR_EN > 0           //如果有启动定时器管理
87.         OSTmrCtr++;            //计时变量 OSTmrCtr 加 1
88.         if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC))
{//如果时间到了
89.             OSTmrCtr = 0;      //计时清 0

```

```
90.         OSTmrSignal();           //发送信号量 OSTmrSemSignal (初始值为0)
91.         //以便软件定时器扫描任务 OSTmr_Task 能请求到信
号量而继续运行下去
92.     }
93. #endif
94. }
```

这些钩子函数是必须声明的，但不是必须定义的，只是为了拓展你的系统功能而已。

### 47.2.2. 任务堆栈结构初始化函数

```
95.OSTaskStkInit()           //任务堆栈结构初始化函数
```

通常，我们的任务定义都是这样的：

```
96. void MyTask (void *p_arg)
97. {
98.     /* 可选，例如处理 'p_arg' 变量 */
99.     while (1) {
100.         /* 任务主体 */
101.     }
102. }
```

典型的 ARM 编译器（Cortex-M3 也是这样）都会把这个函数的第一个参量传递到 R0 寄存器中。

对于像 ARM 内核一般都比较多寄存器的单片机，我们可以把函数中断的局部变量保存在寄存器中，以加快速度。

```
103. OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg,
104. OS_STK *ptos, INT16U opt)
105. {
106.     OS_STK *stk;
107.
108.
109.     (void)opt;           // 'opt' 并没有用到，防止编译器提示警告
110.     stk = ptos;          // 加载栈指针
111.
112.     /* 中断后 xPSR, PC, LR, R12, R3-R0 被自动保存到栈中*/
113.     *(stk) = (INT32U)0x01000000L; // xPSR
114.     *--stk = (INT32U)task;        // 任务入口 (PC)
115.     *--stk = (INT32U)0xFFFFFFFFL; // R14 (LR)
116.     *--stk = (INT32U)0x12121212L; // R12
117.     *--stk = (INT32U)0x03030303L; // R3
118.     *--stk = (INT32U)0x02020202L; // R2
119.     *--stk = (INT32U)0x01010101L; // R1
120.     *--stk = (INT32U)p_arg;       // R0 : 变量
121.
122.     /* 剩下的寄存器需要手动保存在堆栈 */
123.     *--stk = (INT32U)0x11111111L; // R11
124.     *--stk = (INT32U)0x10101010L; // R10
125.     *--stk = (INT32U)0x09090909L; // R9
126.     *--stk = (INT32U)0x08080808L; // R8
127.     *--stk = (INT32U)0x07070707L; // R7
128.     *--stk = (INT32U)0x06060606L; // R6
```

```
129.          * (--stk) = (INT32U) 0x05050505L; // R5
130.          * (--stk) = (INT32U) 0x04040404L; // R4
131.
132.          return (stk);
133.      }
```

这是初始化任务堆栈函数。OSTaskStkInit()被任务创建函数调用，所以在开始时，在栈中作出该任务好像刚被中断一样的假象。

在 ARM 内核中，函数中断后，xPSR, PC, LR, R12, R3-R0 被自动保存到栈中的，R11-R4 如果需要保存，只能手工保存。为了模拟被中断后的假象，OSTaskStkInit()的工作就是在任务自己的栈中保存 cpu 的所有寄存器。这些值里 R1-R12 都没什么意义，这里用相应的数字代号（如 R1 0x01010101）主要是方便调试。

问大家两个问题，以便大家知道是否掌握了这个知识点：

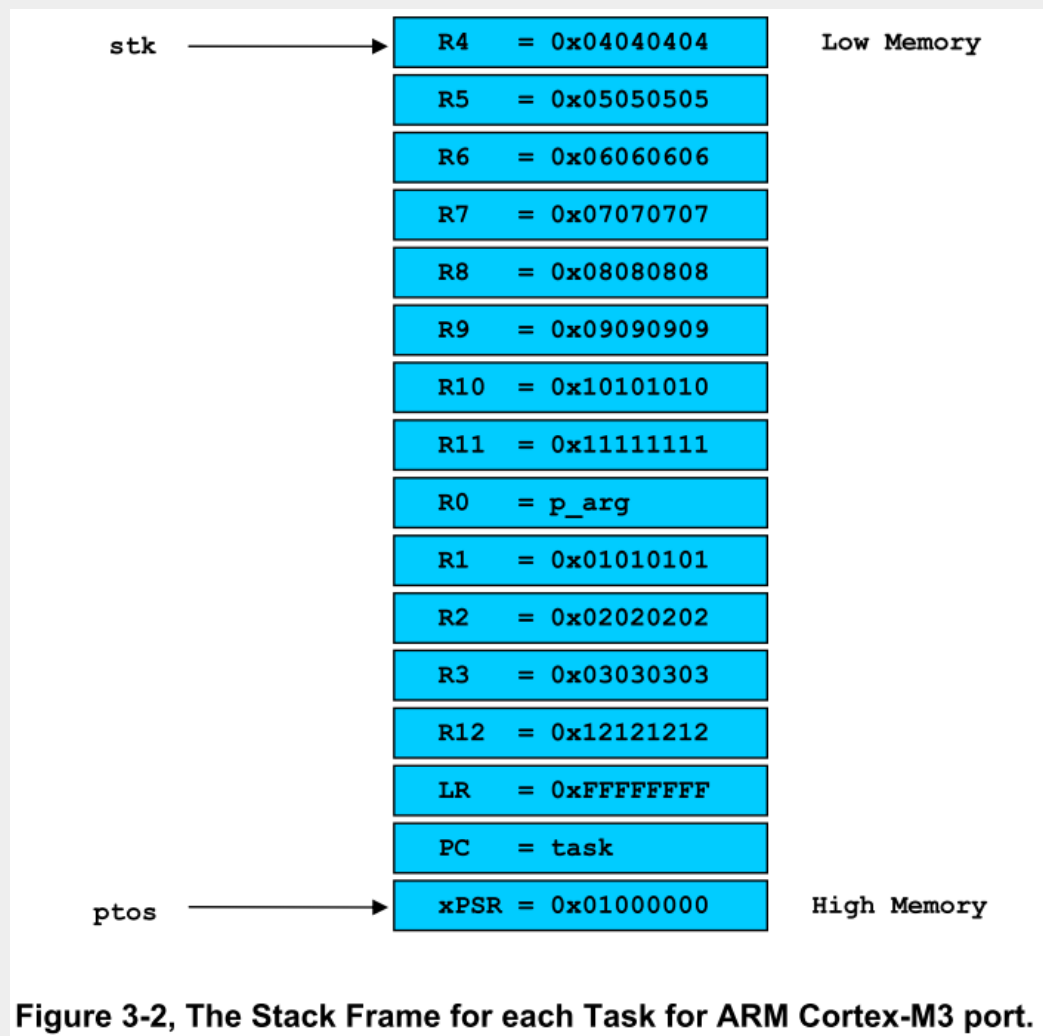
为什么程序是 `*(--stk) = (INT32U)*****;` 而不是保存寄存器的值：`*(--stk) = *(INT32U)*****` 呢？

答案很简单，就是上面说的，任务还没开始运行，栈里保存的 R1-R12 值都没什么意义的，这里仅仅是模拟中断那样的假象，R1-R12 可以是其他任意义的值。

为什么程序是 `*(--stk) = (INT32U)*****;` 而不是 `*(++stk) = (INT32U)*****`

前面已经讲过，M3 的栈生长方向是由高地址向低地址增长的。

栈初始化后，各寄存器的初始值如下：



xPSR = 0x01000000L，**xPSR T 位（第 24 位）置 1**，否则第一次执行任务时 Fault，**PC 必须指向任务入口**，**R14 = 0xFFFFFFFFEL**，最低 4 位为 E，是一个非法值，主要目的是不让使用 R14，即任务是不能返回的。**R0** 用于传递任务函数的参数，因此等于 **p\_arg**。。

### 47.2.3. SysTick 时钟初始化

OS\_CPU\_SysTickInit()会被第一个任务调用，以便初始化 SysTick 定时器。

OS\_CPU\_SysTickInit() 将会调用 OS\_CPU\_SysTickClkFreq() 获取系统时钟频率，用户需要为自己的开发板编写此函数获取时钟频率。

```
134. void OS_CPU_SysTickInit (void)
135. {
136.     INT32U cnts;
```

```

137.
138.
139.         cnts = OS_CPU_SysTickClkFreq() / OS_TICKS_PER_SEC;

140.         //OS_CPU_SysTickClkFreq() 获取时钟频率
141.         //OS_TICKS_PER_SEC 定义每秒时钟节拍中断的次数，即时钟节拍时间
        为 1/OS_TICKS_PER_SEC
142.

143.         /* 使能 SysTick 定时器 */
144.         OS_CPU_CM3_NVIC_ST_RELOAD = (cnts - 1);
145.
146.         /* 使能 SysTick 定时器中断 */
147.         OS_CPU_CM3_NVIC_ST_CTRL |= OS_CPU_CM3_NVIC_ST_CTRL_CLK_S
        RC
148.         | OS_CPU_CM3_NVIC_ST_CTRL_ENABL
        E;
149.
150.         OS_CPU_CM3_NVIC_ST_CTRL |= OS_CPU_CM3_NVIC_ST_CTRL_INTEN
        ;
151.     }

```

但在这里，为了便于理解，我们需要手动修改成自己的，不用这些函数（看上面任务管理函数中需要注释掉的函数）。

除了注释刚才上面说的三个函数外，我们还要注释掉这些宏定义：

```

152.     /*****
        *****/
153.     *                               SYS TICK DEFINES
154.     *****/
        *****/
155. #define OS_CPU_CM3_NVIC_ST_CTRL      (*((volatile INT32U *)0xE000E010))
156.
        /* SysTick Ctrl & Status Reg. */
157. #define OS_CPU_CM3_NVIC_ST_RELOAD    (*((volatile INT32U *)0xE000E014))
158.
        /* SysTick Reload Value Reg. */
159. #define OS_CPU_CM3_NVIC_ST_CURRENT   (*((volatile INT32U *)0xE000E018))
160.
        /* SysTick Current Value Reg. */
161. #define OS_CPU_CM3_NVIC_ST_CAL        (*((volatile INT32U *)0xE000E01C))
162.
        /* SysTick Cal Value Reg. */
163. #define OS_CPU_CM3_NVIC_PRIO_ST      (*((volatile INT8U *)0xE000ED23))
164.
        /* SysTick Handler Prio Reg. */
165.
166. #define OS_CPU_CM3_NVIC_ST_CTRL_COUNT      0x00010000
167.
        /* Count flag. */
168. #define OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC    0x00000004
169.
        /* Clock Source. */
170. #define OS_CPU_CM3_NVIC_ST_CTRL_INTEN      0x00000002
171.
        /* Interrupt enable. */
172. #define OS_CPU_CM3_NVIC_ST_CTRL_ENABLE     0x00000001
173.
        /* Counter mode. */
174. #define OS_CPU_CM3_NVIC_PRIO_MIN           0xFF
175.
        /* Min handler prio. */

```



因为它们为 SysTick 定时器服务的，即需要把所有与 SysTick 有关的代码都要去掉。

### 47.3. 2.3 os\_cpu\_a.asm

这个文件包含了需要用汇编编写的代码。

#### 47.3.1. 声明外部定义

```
176.  EXTERN  OSRunning                ; 声明外部定义，相当于 c 语言的  
      extern  
177.  EXTERN  OSPrioCur  
178.  EXTERN  OSPrioHighRdy  
179.  EXTERN  OSTCBCur  
180.  EXTERN  OSTCBHighRdy  
181.  EXTERN  OSIntNesting  
182.  EXTERN  OSIntExit  
183.  EXTERN  OSTaskSwHook
```

申明这些变量是在其他文件定义的。

#### 47.3.2. 声明全局变量

由于编译器的原因，我们需要将下面的 PUBLIC 改为 EXPORT。（如果下载的源代码是用 RealView 编译的，则此处就不用改了，因为代码本来就是用 EXPORT）

```
184.  PUBLIC  OS_CPU_SR_Save          ; 声明函数在此文件定  
      义  
185.  PUBLIC  OS_CPU_SR_Restore  
186.  PUBLIC  OSStartHighRdy  
187.  PUBLIC  OSCtxSw  
188.  PUBLIC  OSIntCtxSw  
189.  PUBLIC  OS_CPU_PendSVHandler
```

修改后

```
190.  EXPORT  OS_CPU_SR_Save          ; 声明函数在此文件定义  
191.  EXPORT  OS_CPU_SR_Restore  
192.  EXPORT  OSStartHighRdy  
193.  EXPORT  OSCtxSw  
194.  EXPORT  OSIntCtxSw  
195.  EXPORT  OS_CPU_PendSVHandler
```

关于 EXPORT 的用法和意义，可以参考 [RealView 编译工具 4.0 版《汇编器指南》](#) 第 7.8.7 小节 EXPORT 或 GLOBAL：

EXPORT 指令声明一个符号，链接器可以使用该符号解析不同对象和库文件中的符号引用。GLOBAL 是 EXPORT 的同义词。

使用 EXPORT 可使其他文件中的代码能够访问当前文件中的符号。

与 EXPORT 相对应的是 IMPORT，可以参考 RealView 编译工具 4.0 版《汇编器指南》第 7.8.10 小节 IMPORT 和 EXTERN：

这些指令为汇编器提供一个未在当前汇编中定义的名称。在链接时，名称被解析为在其他对象文件中定义的符号。该符号被当作程序地址。如果未指定 [WEAK] 且在链接时没有找到相应的符号，则链接器会产生错误。

### 47.3.3. 段

由于编译器的原因，也要将下面的内容替换一下：

```
196.  RSEG CODE:CODE:NOROOT(2) ; RSEG CODE: 选择段 code。第二个 CODE 表示代码段的意思，只读。  
197.                                ; NOROOT 表示：如果这段中的代码没调用，则允许连接器丢弃这段  
198.                                ; (2)表示：4 字节对齐。假如是 (n)，则表示 2^n 对齐
```

替换为：

```
199.  AREA |.text|, CODE, READONLY, ALIGN=2 ;AREA |.text| 表示：选择段 |.text|。  
200.                                ;CODE 表示代码段，READONLY 表示只读（缺省）  
201.                                ;ALIGN=2 表示 4 字节对齐。若 ALIGN=n，这 2^n 对齐  
202.  THUMB ;Thumb 代码  
203.  REQUIRE8 ;指定当前文件要求堆栈八字节对齐  
204.  PRESERVE8 ;令指定当前文件保持堆栈八字节对齐
```

对于汇编命令，想了解更多，请看 RealView 编译工具 4.0 版《汇编器指南》

关于段的补充：段可以分为代码段和数据段，其中代码段的内容就是可执行代码。

用 keil 编译时，经常会出现这样的提示：

```
linking...  
Program Size: Code=3732 RO-data=336 RW-data=24 ZI-data=512  
FromELF: creating hex file...
```

Code 是代码占用的空间，RO-data 是 Read Only 只读常量的大小，如 const 型，RW-data 是（Read Write）初始化了的可读写变量的大小，ZI-data 是（Zero Initialize）没有初始化的可读写变量的大小。ZI-data 不会被算做代码里因为不会被初始化。

简单的说就是在烧写的时候是 FLASH 中的被占用的空间为：Code+RO Data+RW Data

程序运行的时候，芯片内部 RAM 使用的空间为：RW Data + ZI Data

#### 47.3.4. 向量中断控制器 NVIC

前面讲过，关于 PendSV，可以看《Cortex-M3 权威指南》中第 7.6 小节 SVC 和 PendSV。不知道有多少位朋友看过呢？呵呵，如果看过，那下面的内容，就容易理解很多，不然，像看天书那样。

```
205.      NVIC_INT_CTRL    EQU    0xE000ED04      ;中断控制及状态寄存器
ICSRR 的地址
206.                                          ;见《Cortex-M3
权威指南》第 8.4.5 小节 表 8.5)
207.      NVIC_SYSPRI14   EQU    0xE000ED22      ;系统异常优先级寄存器
PRI_14
208.                                          ;
即设置 PendSV 的优先级
209.                                          ;见
《Cortex-M3 权威指南》第 8.4.2 小节 表 8.3B
210.      NVIC_PENDSV_PRI EQU    0xFF           ;定义 PendSV 的可编程优
优先级为 255，即最低
211.                                          ;
为啥是最低呢？大家思考一下
212.      NVIC_PENDSVSET   EQU    0x10000000     ;中断控制及状态寄存器 ICSR 的位
28
213.                                          ;写 1 以悬起 PendSV 中
断。读取它则返回 PendSV 的状态
```

关于向量中断控制器 NVIC，推荐大家看《Cortex-M3 权威指南》的第 7、第 8 章，里面有很详细的说明，我这里就不做太多的解释。

回答一下刚才提出的问题：**为啥要把 PendSV 的可编程优先级设为最低？**

与 SVC 异常必须在执行 SVC 指令后立即得到响应的不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。

悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行（这里是为什么需要定义 NVIC\_PENDSVSET 的原因）。

PendSV 的典型使用是用在任务切换上。

假如系统使用 SysTick 异常进行任务切换，则正常情况下：

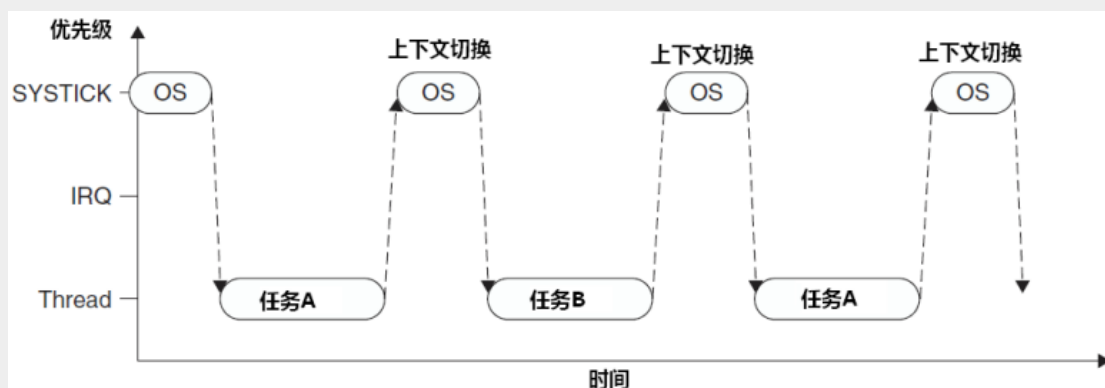
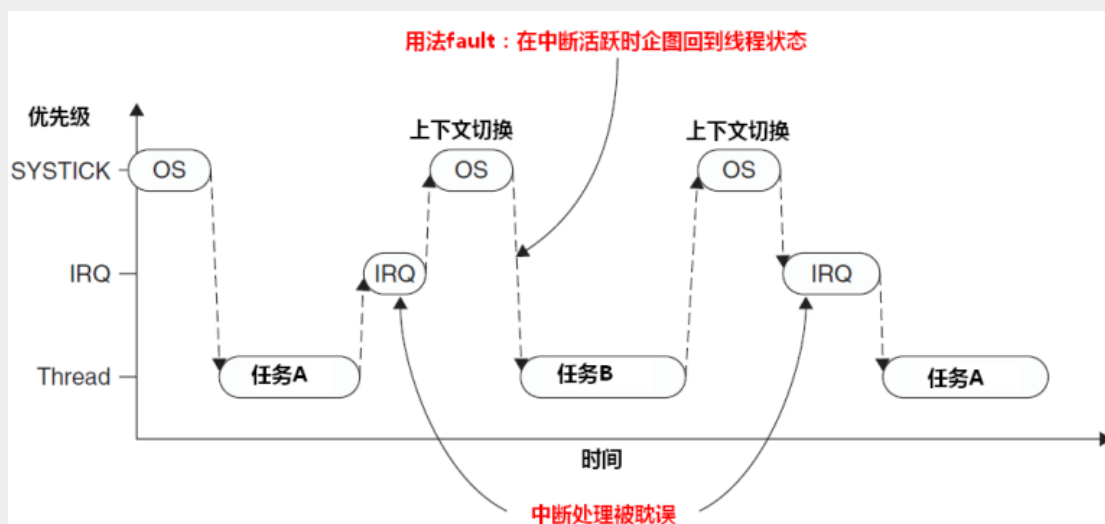
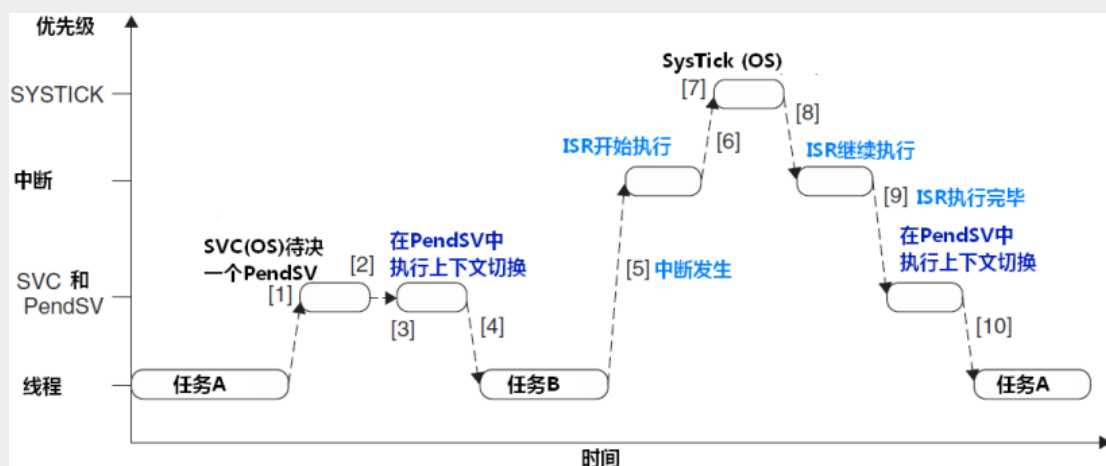


图 7.15 两个任务间通过 SysTick 进行轮转调度的简单模式

但实际上，有时候单片机会进入中断状态响应其他中断，这时如果再产生滴答定时器中断，进行任务切换，打断了原来的中断服务，则运行流程为：



显然，中断服务被打断了，间距的时间比较长，这是实时系统所无法忍受的。为此，引入了 PendSV 来完美解决这个问题了：



PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。

注：此部分内容出自《Cortex-M3 权威指南》

### 47.3.5. 中断

与中断方式 3 相关的有两个汇编函数：

```
214.      ; OS_ENTER_CRITICAL() 里进入临界段调用，保存现场环境
215.      OS_CPU_SR_Save
216.      MRS      R0, PRIMASK      ; 读取 PRIMASK 到 R0 (保存全局
中断标记，除了故障中断)
217.      CPSID   I                  ; PRIMASK=1，关中断
218.      BX      LR                  ; 返回，返回值保存在 R0
219.
220.
221.      ; OS_EXIT_CRITICAL() 里退出临界段调用，恢复现场环境
222.      OS_CPU_SR_Restore
223.      MSR      PRIMASK, R0      ; 读取 R0 到 PRIMASK 中 (恢复全局中
断标记)，通过 R0 传递参数
224.      BX      LR
```

功能：关全局中断前，保存全局中断标志，进入临界段。退出临界段后恢复中断标记。

汇编命令讲解：

| 功能                   | 作用                                                                                                                                                                                           |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>CPS</b> （更改处理器状态） | 会更改 CPSR 中的一个或多个模式以及 A、I 和 F 位，但不更改其他 CPSR 位。<br>CPSID 就是中断禁止，CPSIE 中断允许。<br><br>A      表示 启用或禁用不精确的中止。<br>I      表示 启用或禁用 IRQ 中断。<br>F      表示 启用或禁用 FIQ 中断。<br><br>此处 CPSID I 就表示禁止 IRQ 中断 |
| <b>MRS</b>           | 将 CPSR 或 SPSR 的内容移到一个通用寄存器中。                                                                                                                                                                 |
| <b>MSR</b>           | 将立即数或通用寄存器的内容加载到 CPSR 或 SPSR 的指定字段中。                                                                                                                                                         |
| <b>BL</b>            | 跳转指令，可将下一个指令的地址复制到 LR（R14，链接寄存器）中。                                                                                                                                                           |

注：此部分内容出自 RealView 编译工具 4.0 版《汇编器指南》

#### 47.3.6. 启动最高优先级任务

OSStartHighRdy() 启动最高优先级任务，由 OSStart() 里调用，调用前必须先调用 OSTaskCreate 创建至少一个用户任务，否则系统会发生崩毁。

```
225.      OSStartHighRdy
226.          LDR    R0, =NVIC_SYSPRI14      ; 装载 系统异常优先级寄
寄存器 PRI_14
227.                                          ; 即设置 PendSV
中断优先级的寄存器
228.          LDR    R1, =NVIC_PENDSV_PRI    ; 装载 PendSV 的可编程
优先级 (255)
```

```

229.          STRB    R1, [R0]                ; 无符号字节寄存器存储。
R1 是要存储的寄存器
230.                ; 存储到内存地址所基于的
寄存器
231.                ;
即设置 PendSV 中断优先级为 255
232.
233.          MOV     R0, #0                  ; 把数值 0 复制到 R0 寄存
器
234.          MSR     PSP, R0                 ; 将 R0 的内容加载到程序
状态寄存器 PSR 的指定字段中。
235.
236.          LDR     R0, __OS_Running
; OSRunning = TRUE
237.          MOV     R1, #1
238.          STRB    R1, [R0]
239.
240.          LDR     R0, =NVIC_INT_CTRL      ; 装载 中断控制及状态寄
寄存器 ICSR 的地址
241.          LDR     R1, =NVIC_PENDSVSET    ; 中断控制及状态寄存器
ICSR 的位 28
242.          STR     R1, [R0]                ; 设置 中断控制及状态寄
寄存器 ICSR 位 28 为 1
243.                ;
以悬起 (允许) PendSV 中断
244.
245.          CPSIE   I                        ; 开中断 (前面已经讲解
过)

```

### 47.3.7. 任务切换

当任务放弃 CPU 的使用权时，就会调用 OS\_TASK\_SW()

一般情况下，OS\_TASK\_SW() 是做任务切换。但在 M3 中，任务切换的工作都被放到 PendSV 的中断处理服务中去做以加快处理速度，因此 OS\_TASK\_SW() 只需简单的悬起 (允许) PendSV 中断即可。当然，这样就只有当再次开中断的时候，PendSV 中断处理函数才能执行。

OS\_TASK\_SW() 是由 OS\_Sched() (此函数在 OS\_CORE.C) 调用。

```

246.          /*****任务级调度器*****/
247. void OS_Sched (void)
248. {
249. #if OS_CRITICAL_METHOD == 3
250.     OS_CPU_SR cpu_sr = 0;
251. #endif
252.
253.     OS_ENTER_CRITICAL();
254.     if (OSIntNesting == 0) {
255.         if (OSLockNesting == 0) {
256.             OS_SchedNew();
257.         }
258.     }
259. }

```

```
257. // 见 os_core.c , 会修改 OSPrioHighRdy
258. if (OSPrioHighRdy != OSPrioCur) { //如果得到的最高
259. //注：当前运行的任务也在就绪表里
260. OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; //
//得到任务控制块指针
261. #if OS_TASK_PROFILE_EN > 0
262. OSTCBHighRdy->OSTCBCtxSwCtr++; //统计任务切换到
次任务的计数器加 1
263. #endif
264. OSCtxSwCtr++; //统计任务切换次
数的计数器加 1
265. OS_TASK_SW(); //进行任务切
换
266. }
267. }
268. }
269. OS_EXIT_CRITICAL(); //退出临界段，开
中断
270. }
```

悬起(允许)PendSV

开中断，执行 PendSV 中断

OS\_TASK\_SW() 就是用宏定义包装的 OSCtxSw() (见 OS\_CPU.H):

```
271. #define OS_TASK_SW() OSCtxSw()
```

前面已经说了，OS\_TASK\_SW() 只需简单的悬起(允许)PendSV 中断即可。

```
272. OSCtxSw
273. ; 悬起(允许)PendSV 中断 (看不懂这段代码的，可参考前面见过的 OSStartHighRdy )
274. LDR R0, =NVIC_INT_CTRL ; 装载 中断控制及状态
寄存器 ICSR 的地址
275. LDR R1, =NVIC_PENDSVSET ; 中断控制及状态寄存器
ICSR 的位 28
276. STR R1, [R0] ; 设置 中断控制及状态
寄存器 ICSR 位 28 为 1
277. ; 以悬起(允许)PendSV 中断
278. BX LR ; 返回
```

#### 47.3.8. 中断退出处理

当中断处理函数退出时，就会调用 OSIntExit()来决定是否有优先级更高的任务需要执行。如果有，OSIntExit()会调用 OSIntCtxSw() 做任务切换。

在 M3 里，与 OSCtxSw 一样，任务切换时，OSIntCtxSw 都只需简单的悬起(允许)PendSV 中断即可，真正的任务切换工作放在 PendSV 中断服务程序里，等待开中断时才正在执行任务切换。



在这里，OSCtxSw 的代码是与 OSIntCtxSw 完全相同的：

```
279.      OSIntCtxSw
280.          LDR      R0, =NVIC_INT_CTRL      ; trigger the PendSV
exception
281.          LDR      R1, =NVIC_PENDSVSET
282.          STR      R1, [R0]
283.          BX      LR
```

尽管这里的 SCtxSw()和 OSIntCtxSw()代码是完全一样的，但事实上，这两个函数的意义是不一样的。

OSCtxSw() 做的是任务之间的切换。例如任务因为等待某个资源或做延时，就会调用这个函数来进行任务调度，有任务调度进行任务切换。

OSIntCtxSw()则是中断退出时，如果最高优先级就绪任务并不是被中断的任务就会被调用，由中断状态切换到最高优先级就绪任务中，所以 OSIntCtxSw()又称中断级的中断任务。

由于调用 OSIntCtxSw()之前肯定发生了中断，所以无需保存 CPU 寄存器的值了。这里只不过由于 CM3 的特殊机制导致了在这两个函数中只要做触发 PendSV 中断即可，具体切换由 PendSV 中断服务来处理。

#### 47.3.9. PendSV 中断服务

前面已经讲解过很多次 PendSV 的作用了，这里就不啰嗦了，先来 PendSV 中断服务的伪代码吧，方便理解：

```
284.      //OS_CPU_PendSVHandler 伪代码思路
285.      OS_CPU_PendSVHandler:
286.          if (PSP != NULL) {          //当调用 OS_CPU_PendSVHandler() 时，
287.                                          //CPU 就会自动保
存 xPSR、PC、LR、R12、R0-R3 寄存器到堆栈
288.                                          //保存后，CUP 的栈 SP 指针会切换到使用主
堆栈指针 MSP 上
289.                                          //我们只需检测 进入栈指针 PSP 是否为
NULL 就知道是否进行任务切换
290.                                          //因此当我们第一次启动任务是，
OSStartHighRdy() 就把 PSP 设为 NULL，
291.                                          //避免系统以为已经进行任
务切换
292.          Save R4-R11 onto task stack; //手动保存 R4-R11
293.          OSTCBCur->OSTCBStkPtr = SP; //保存进入栈指针 PSP 到任务控制块
294.
//以便下次继续任务运行时继续使用原来的栈
```

```

295.    }
296.    OSTaskSwHook();           //此处便于我们使用钩子函数来拓展
功能
297.    OSPrioCur = OSPrioHighRdy;           //获取最高优先级就绪任务
的优先级
298.    OSTCBCur  = OSTCBHighRdy;           //获取最高优先级就绪任务的任务控
制块指针
299.    PSP        = OSTCBHighRdy->OSTCBStkPtr;           //保存进入栈指
针
300.    Restore R4-R11 from new task stack;           //从新的栈恢复
R4-R11 寄存器
301.    Return from exception;           //返回

```

具体的汇编代码:

```

302.    OS_CPU_PendSVHandler           ;CPU 会自动保
存 xPSR, PC, LR, R12, R0-R3
303.    CPSID    I           ;关中断
304.    MRS      R0, PSP           ;PSP 就是栈指针,
R0=PSP
305.    CBZ      R0, OSPendSV_nosave           ;当 PSP==0, 执
行 OSPendSV_nosave 函数
306.
307.    SUB      R0, R0, #0x20           ;装载 r4-11 到栈, 共 8 个
寄存器, 32 位, 4 个字节
308.
    ;即 8*4=32=0x20
309.    STM      R0, {R4-R11}           ;
310.
311.    LDR      R1, __OS_TCBCur           ;R1=&OSTCBCur
312.    LDR      R1, [R1]           ;R1=*R1 (R1=OSTCBCur)
313.    STR      R0, [R1]           ;*R1=R0 (*OSTCBCur=SP)
314.
315.    OSPendSV_nosave
316.    PUSH     {R14}           ;保存 R14
317.    LDR      R0, __OS_TaskSwHook           ;调用钩子函数
OSTaskSwHook()
318.    BLX      R0
319.    POP      {R14}           ;恢复 R14
320.
321.    LDR      R0, __OS_PrioCur           ;设置当前优先级为最高优先级
就绪任务的优先级
322.
    ;OSPrioCur = OSPrioHighRdy
323.    LDR      R1, __OS_PrioHighRdy
324.    LDRB     R2, [R1]
325.    STRB     R2, [R0]
326.
327.    LDR      R0, __OS_TCBCur           ;设置当前任务控制块指针
328.    LDR      R1, __OS_TCBHighRdy           ;OSTCBCur = OSTCBHigh
Rdy
329.    LDR      R2, [R1]
330.    STR      R2, [R0]
331.
332.    LDR      R0, [R2]           ;R0 是新的 SP
333.    ;SP = OSTCBHighRdy-
>OSTCBStkPtr;
334.

```

```

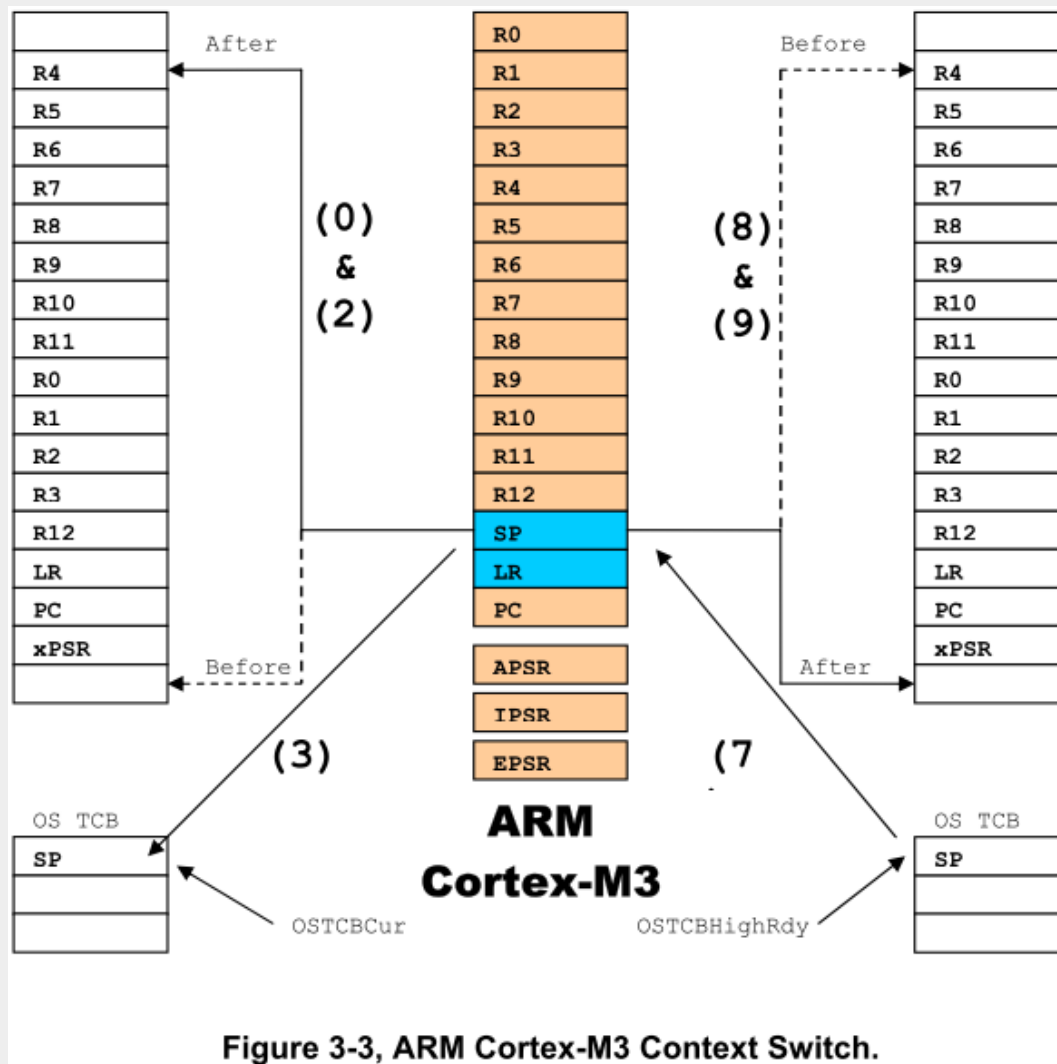
335.          LDM      R0, {R4-R11}          ;从新的栈恢复 R4-R11
336.          ADD     R0, R0, #0x20
337.          MSR     PSP, R0                ;PSP=R0,用新的栈 SP 加载
PSP
338.          ORR     LR, LR, #0x04          ;确保 LR 位 2 为 1, 返回到
使用进程堆栈
339.          CPSIE   I                    ;开中断
340.          BX      LR                    ;返回

```

当第一次开始任务切换时时，而任务刚创建时 R4-R11 已经保存在堆栈中，此时不用再保存，就会跳到 OS\_CPU\_PendSVHandler\_nosave 执行。

前面已经说过真正的任务切换是在 PendSV 中断处理函数里做的，由于 M3 在中断时会有一些的寄存器自动保存到任务堆栈里，所以在 PendSV 中断处理函数中只需保存 R4-R11 并调节堆栈指针即可。其中 xPSR, PC, LR, R12, R0-R3 已自动保存，不用我们管了。

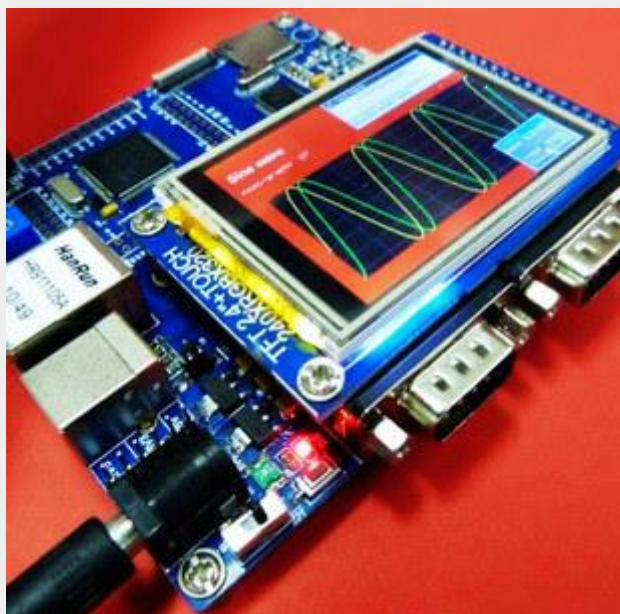
下面是一个任务切换时寄存器的情况：



到此，重要的代码知识点就讲解完毕了，呵呵，初学者看起来会有点困难，不过要加油哦！多看几次就弄懂的！限于个人能力，欢迎各位高手指出错误，在此先表达感谢！

### 48. 3、uC/OS-II 移植到 STM32 处理器的步骤

下面，我们将讲解移植 uC/OS-II 到野火开发板的示范实验，先来一张野火 STM32 开发板的图片：

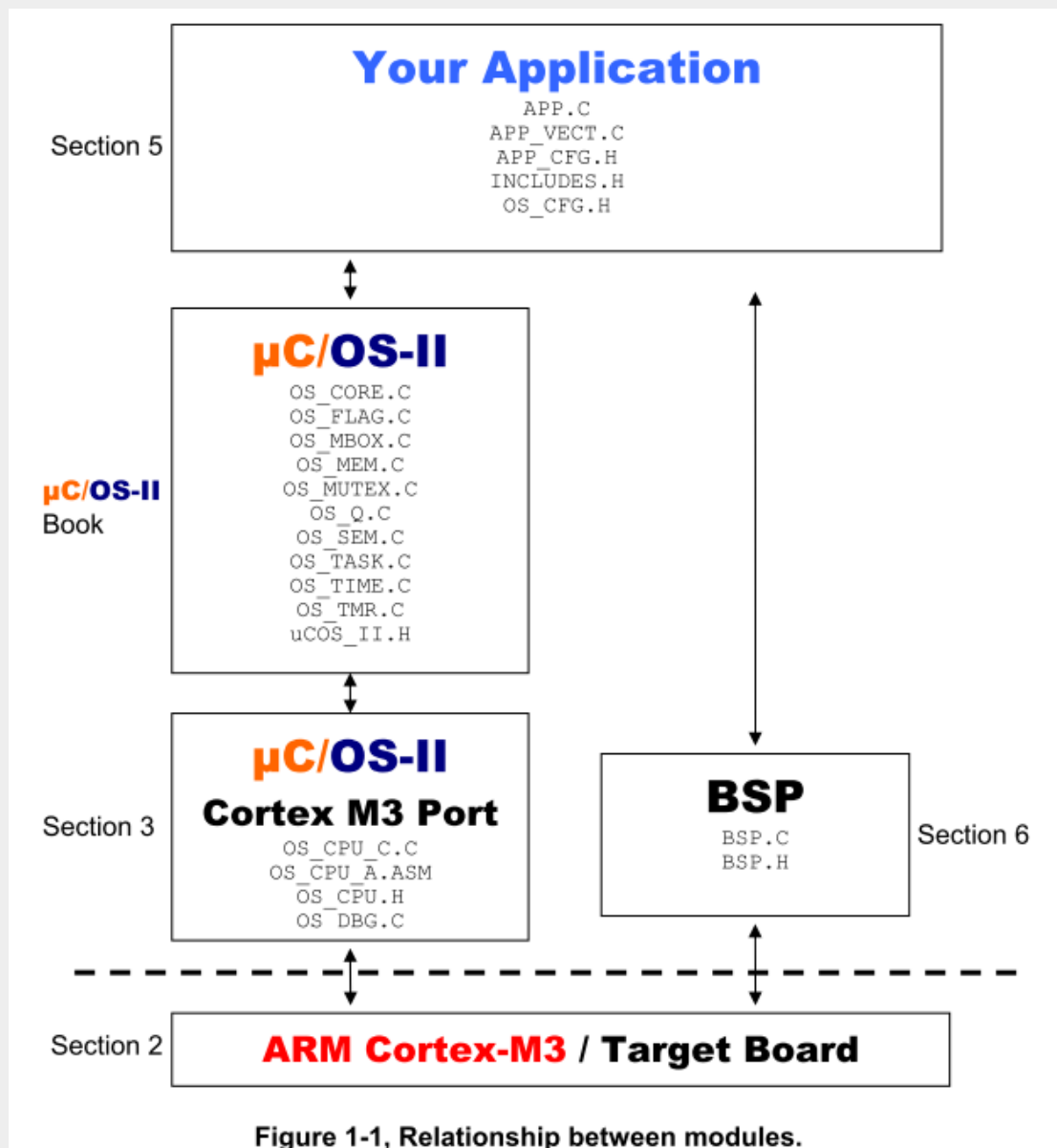


我们的 uC/OS-II 移植实验是在野火 STM32 开发板附带的 LED 实验基础上来讲的，所用的工程文件也是野火 STM32 开发板所带的 LED 例程。

对于没接触过野火 STM32 开发板实验教程的朋友，建议你们还是看下野火的 LED 教程。

好了，转到正题上。看完前面的内容，不知道各位是否对 uC/OS-II 的移植有了整体的把握了？对于 uC/OS-II 的工程文件结构，又是否了解呢？

我们先来回顾一下一个 uC/OS-II 的开发板工程的文件结构吧：



很明显，为了让开发板硬件驱动程序与 uC/OS-II 系统的文件系统分开，好让我们开发工程时不必太乱，我们需要按照一定的规则建立分类文件夹。

好了，下面开始正式移植 uC/OS-II 了：

在这里，我们直接采用野火 STM32 的 LED 工程来作为基础，进行 uC/OS-II 移植的讲解（如果不知道 LED 工程如何建立，请看野火 STM32 的 LED 教程，这里就不再重复）：

#### 48. 1. 3.1 打开 LED 工程模版

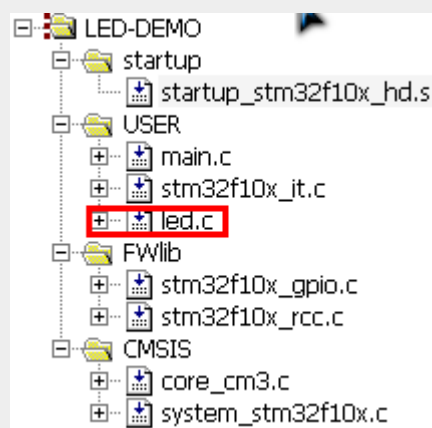
首先，我们从野火 STM32 光盘资料那里提取 LED 实验：



LED 工程文件在光盘目录下:\实验代码+PDF 教程\野火 Stm32-实验代码\  
2-LED.rar



解压打开工程后，就会看到 LED 工程的文件结构为：



这个是我们过往开发裸机单片机程序时写的工程文件结构，但对于 uC/OS-II，或者其他大型点的软件工程，这样的文件结构就会很乱的。

## 48.2. 3.2 搭建 uC/OS-II 工程文件结构

我们需要建立的文件结构为(其他没显示出来的文件，按照原来位置那样不改变)：



## STM32+UCOS+LED

```
|
|
|—USER
|   main.c
|   includes.h           //新建
|—uCOS-II
|   |—Source             //这文件夹来自于下载附件的 Micrium\Software\ucOS-II
|   |   os_core.c
|   |   os_flag.c
|   |   os_mbox.c
|   |   os_mem.c
|   |   os_mutex.c
|   |   os_q.c
|   |   os_sem.c
|   |   os_task.c
|   |   os_time.c
|   |   os_tmr.c
|   |   ucos_ii.h
|   |—Ports              //里面文件来自于\Micrium\Software\ucOS-II\Ports\arm-cortex-
m3\Generic\IAR
|       os_cpu.h
|       os_cpu_a.asm
|       os_cpu_c.c
|       os_dbg.c
|—BSP                     //这文件夹新建，里面文件来自 USER 文件夹
|   led.c
```

```
|   led.h
|   BSP.c
|   BSP.h
└─APP                                //这文件及里面的文件（除 os_cfg.h）都是新建
    app.c
    app.h
    app_cfg.h //是用来配置应用软件，主要是任务的优先级和堆栈大小，中断优先级等
    os_cfg.h  //拷贝自 Micrium\Software\EvalBoards\ST\S.\I.\OS.\os_cfg.h
```

为了方便初学者，下面的为具体的详细步骤，如果会自行搭建文件结构，可跳过这一小节：

- ① 把 LED 工程所在的文件夹先改名为：STM32+UCOS+LED (建议这样做，避免与原来 LED 工程混乱)
- ② 在 USER 文件夹下新建 includes.h 头文件。
- ③ 按照之前给的 uC/OS-II 文件结构图，我们在工程的根目录下建立 BSP 文件夹、APP 文件夹和 uCOS-II 文件夹。

BSP 文件夹                      存放外设硬件驱动程序。

APP 文件夹                      存放应用软件任务

uCOS-II 文件夹 uC/OS-II 的相关代码

- ④ 把 USER 文件夹下的 led.h 和 led.c 文件剪切到 BSP 文件夹里。

在 BSP 文件夹里新建 BSP.c 和 BSP.h 文件。

- ⑤ 在 APP 文件夹下建立 app.h、app.c 和 app\_cfg.h 文件。

拷贝 uC/OS-II 源代码附件那里的 Micrium\Software\EvalBoards\ST\STM32F103ZE-SK\IAR\OS-Probe-LCD\os\_cfg.h 到此目录。

- ⑥ 把 uC/OS-II 源代码附件那里的\Micrium\Software\uCOS-II 下的 Source 文件夹复制到工程里刚才新建的 uCOS-II 文件夹里。

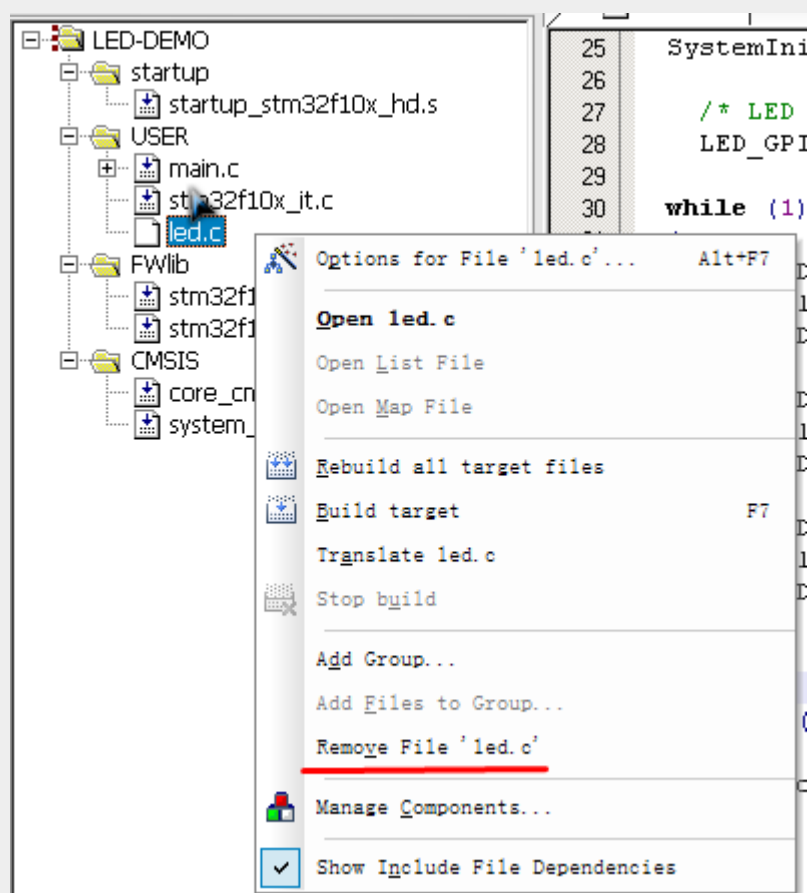
把 Micrium\Software\uCOS-II\Ports\arm-cortex-m3\Generic\IAR 下的文件复制到工程 uCOS-II 文件夹中新建的 Ports 文件夹里。复制后，选中全部文件，右键——属性——去掉只读属性——确定。

到此，工程的目录结构就建立好了，需要修改工程设置。

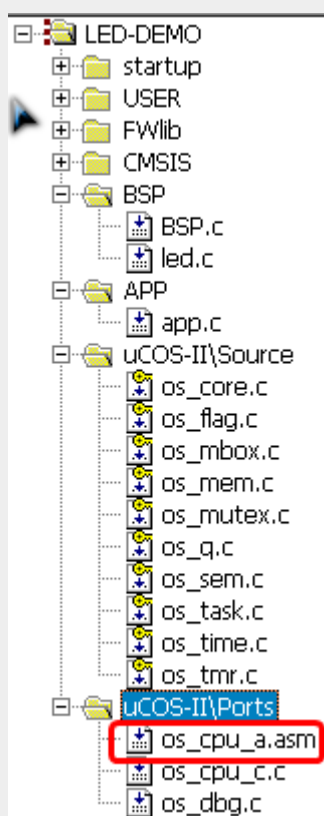
- ⑦ 打开工程文件，会发现提示出错，不需要管他，直接点击确定就可以了。



原因是我们修改了 led.h 和 led.c 的路径。所以我们需要在项目里手动删掉原来的 led.c :

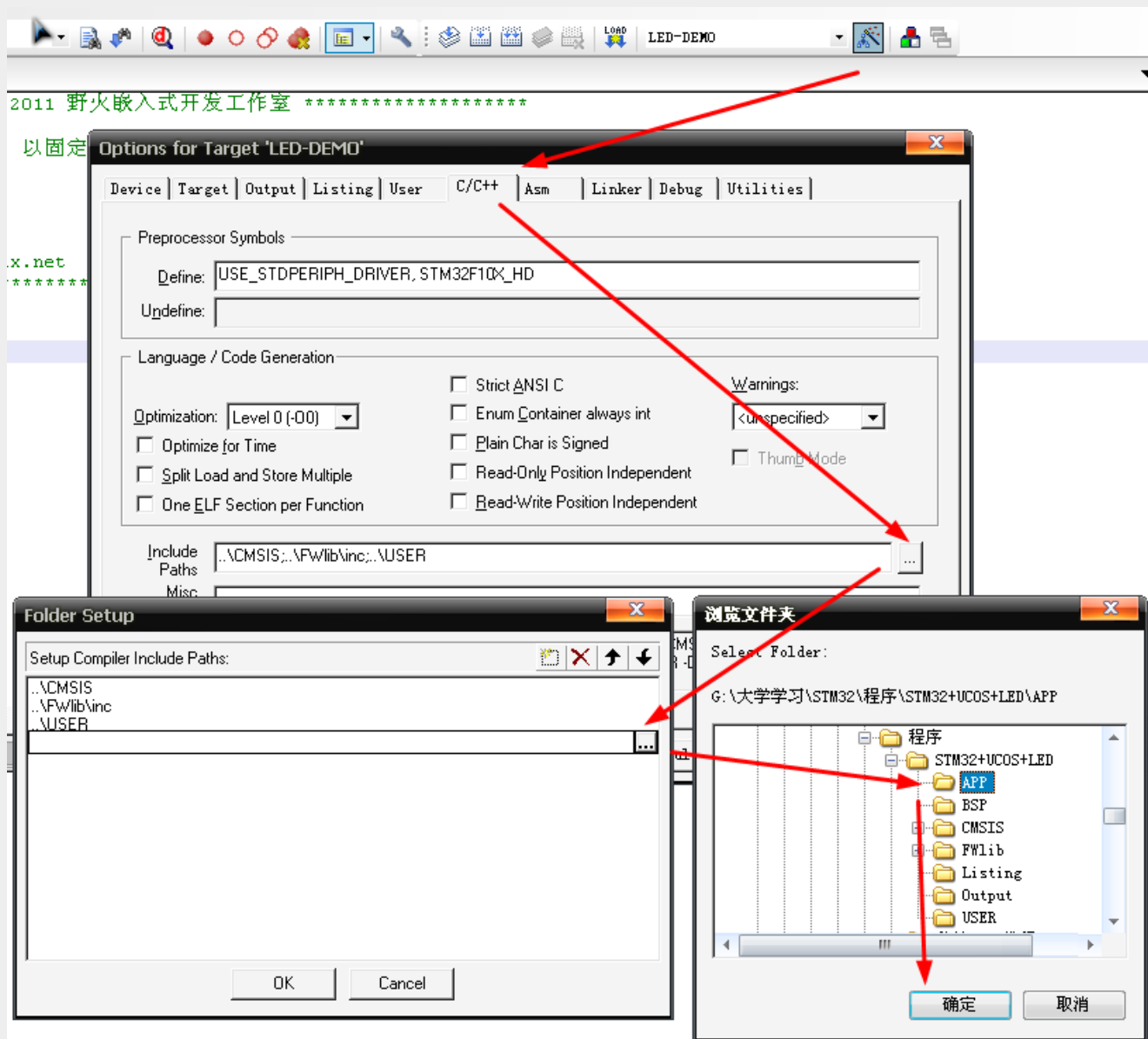


建立 BSP、APP 和 uCOS-II 下两个文件夹，即共四个文件夹的组，并添加进相应的文件：



注意：别漏了在 **uCOS-II\Ports** 中添加汇编文件 **os\_cpu\_a.asm** !!!

也要添加这四个文件进去编译路径：



即 include paths 设置为:

..\CMSIS;..\FWlib\inc;..\USER;..\APP;..\BSP;..\uCOS-II\Ports;..\uCOS-II\Source

至此，完成全部工程的设置，需要开始移植修改代码了！

## 48.3. 3.3 配置 uC/OS-II

首先，修改代码，当然是从配置 uC/OS-II 开始啦。因为我们做的是简单的实验，为此，我们需要把多余的模块剪裁掉，等需要用到再启用，以减少内核体积。

## 48.3.1. os\_cfg.h

os\_cfg.h 是用来配置系统功能的，我们需要通过修改它来达到剪裁系统功能的目的。

在做实际项目时，我们通常也不会用完全部的 uC/OS-II 功能，我们需要通过裁剪内核以避免浪费系统的宝贵资源。

配置 os\_cfg.h，是每个入门移植 uC/OS-II 的初学者都应该需要学会的，尽管非常枯燥无味。

其实，os\_cfg.h 的配置也是有规律的。

## ■ os\_cfg.h 配置表格

| 文件名      | 分类   |    | 配置宏                    | 注解        |
|----------|------|----|------------------------|-----------|
| os_cfg.h | 功能剪裁 | 任务 | OS_TASK_CHANGE_PRIO_EN | 改变任务优先级   |
|          |      |    | OS_TASK_CREATE_EN      |           |
|          |      |    | OS_TASK_CREATE_EXT_EN  |           |
|          |      |    | OS_TASK_DEL_EN         |           |
|          |      |    | OS_TASK_NAME_SIZE      |           |
|          |      |    | OS_TASK_PROFILE_EN     |           |
|          |      |    | OS_TASK_QUERY_EN       | 获得有关任务的信息 |
|          |      |    | OS_TASK_STAT_EN        | 使用统计任务    |
|          |      |    | OS_TASK_STAT_STK_CHK_  | 检测任务堆栈    |

|  |  |       |                       |  |
|--|--|-------|-----------------------|--|
|  |  |       | EN                    |  |
|  |  |       | OS_TASK_SUSPEND_EN    |  |
|  |  |       | OS_TASK_SW_HOOK_EN    |  |
|  |  | 信号量集  | OS_FLAG_EN            |  |
|  |  |       | OS_FLAG_ACCEPT_EN     |  |
|  |  |       | OS_FLAG_DEL_EN        |  |
|  |  |       | OS_FLAG_QUERY_EN      |  |
|  |  |       | OS_FLAG_WAIT_CLR_EN   |  |
|  |  | 消息邮箱  | OS_MBOX_EN            |  |
|  |  |       | OS_MBOX_ACCEPT_EN     |  |
|  |  |       | OS_MBOX_DEL_EN        |  |
|  |  |       | OS_MBOX PEND_ABORT_EN |  |
|  |  |       | OS_MBOX_POST_EN       |  |
|  |  |       | OS_MBOX_POST_OPT_EN   |  |
|  |  |       | OS_MBOX_QUERY_EN      |  |
|  |  | 内存管理  | OS_MEM_EN             |  |
|  |  |       | OS_MEM_QUERY_EN       |  |
|  |  | 互斥信号量 | OS_MUTEX_EN           |  |
|  |  |       | OS_MUTEX_ACCEPT_EN    |  |
|  |  |       | OS_MUTEX_DEL_EN       |  |
|  |  |       | OS_MUTEX_QUERY_EN     |  |
|  |  | 队列    | OS_Q_EN               |  |
|  |  |       | OS_Q_ACCEPT_EN        |  |
|  |  |       | OS_Q_DEL_EN           |  |
|  |  |       | OS_Q_FLUSH_EN         |  |
|  |  |       | OS_Q PEND_ABORT_EN    |  |



|  |     |       |                       |          |
|--|-----|-------|-----------------------|----------|
|  |     |       | OS_Q_POST_EN          |          |
|  |     |       | OS_Q_POST_FRONT_EN    |          |
|  |     |       | OS_Q_POST_OPT_EN      |          |
|  |     |       | OS_Q_QUERY_EN         |          |
|  |     | 信号量   | OS_SEM_EN             |          |
|  |     |       | OS_SEM_ACCEPT_EN      |          |
|  |     |       | OS_SEM_DEL_EN         |          |
|  |     |       | OS_SEM_PEND_ABORT_EN  |          |
|  |     |       | OS_SEM_QUERY_EN       |          |
|  |     |       | OS_SEM_SET_EN         |          |
|  |     | 时间管理  | OS_TIME_DLY_HMSM_EN   |          |
|  |     |       | OS_TIME_DLY_RESUME_EN |          |
|  |     |       | OS_TIME_GET_SET_EN    |          |
|  |     |       | OS_TIME_TICK_HOOK_EN  |          |
|  |     | 定时器管理 | OS_TMR_EN             |          |
|  |     | 其他    | OS_APP_HOOKS_EN       | 应用函数钩子函数 |
|  |     |       | OS_CPU_HOOKS_EN       | CPU 钩子函数 |
|  |     |       | OS_ARG_CHK_EN         |          |
|  |     |       | OS_DEBUG_EN           | 调试       |
|  |     |       | OS_EVENT_MULTI_EN     | 使能多重事件控制 |
|  |     |       | OS_TICK_STEP_EN       | 使能节拍定时   |
|  |     |       | OS_SCHED_LOCK_EN      | 使能调度锁    |
|  | 数据结 | 任务    | OS_MAX_TASKS          |          |

|  |   |       |                          |              |
|--|---|-------|--------------------------|--------------|
|  | 构 |       | OS_TASK_TMR_STK_SIZE     |              |
|  |   |       | OS_TASK_STAT_STK_SIZE    | 统计任务堆栈容量     |
|  |   |       | OS_TASK_IDLE_STK_SIZE    |              |
|  |   | 信号量集  | OS_MAX_FLAGS             |              |
|  |   |       | OS_FLAG_NAME_SIZE        |              |
|  |   |       | OS_FLAGS_NBITS           |              |
|  |   | 内存管理  | OS_MAX_MEM_PART          | 内存块的最大数目     |
|  |   |       | OS_MEM_NAME_SIZE         |              |
|  |   | 队列    | OS_MAX_QS                | 消息队列的最大数目    |
|  |   | 定时器管理 | OS_TMR_CFG_MAX           |              |
|  |   |       | OS_TMR_CFG_NAME_SIZE     |              |
|  |   |       | OS_TMR_CFG_WHEEL_SIZE    |              |
|  |   |       | OS_TMR_CFG_TICKS_PER_SEC |              |
|  |   | 其他    | OS_EVENT_NAME_SIZE       |              |
|  |   |       | OS_LOWEST_PRIO           | 最低优先级        |
|  |   |       | OS_MAX_EVENTS            | 事件控制块的最大数量   |
|  |   |       | OS_TICKS_PER_SEC         | 节拍定时器每1s定时次数 |

我们需要对 os\_cfg.h 进行如下修改：

- ① 首先肯定是禁用信号量、互斥信号量、邮箱、队列、信号量集、定时器、内存管理，关闭调试模式：

```
341. #define OS_FLAG_EN          0 //禁用信号量集
342. #define OS_MBOX_EN          0 //禁用邮箱
343. #define OS_MEM_EN           0 //禁用内存管理
344. #define OS_MUTEX_EN         0 //禁用互斥信号量
345. #define OS_Q_EN             0 //禁用队列
346. #define OS_SEM_EN           0 //禁用信号量
347. #define OS_TMR_EN           0 //禁用定时器
348. #define OS_DEBUG_EN         0 //禁用调试
```

- ② 现在也用不着应用软件的钩子函数，也禁掉；多重事件控制也禁掉

```
349. #define OS_APP_HOOKS_EN      0
350. #define OS_EVENT_MULTI_EN    0
```

这些所做的修改主要是把一些功能给去掉，减少内核大小，也利于调试。等用到的时候，再开启相应的功能。

注意，有时候，配置时，会出现无法通过编译，例如提示某个变量没声明。一方面有可能是你自己配置问题，另外一方面，也有可能是作者代码不够完善。

做完这个移植实验后，你们可以来试验一下。

把 OS\_Q\_EN 和 OS\_MBOX\_EN 都设为 0，OS\_EVENT\_MULTI\_EN 为 1，编译时会提示：

```
..\uCOS-II\Source\os_core.c(535): error: #136: struct "os_tcb" has no field "OSTCBMsg"
```

意思是在 os\_core.c 第 535 行 结构体 os\_tcb 没有 OSTCBMsg 这个成员。当然，解决方法也很简单。

#### 48.4. 3.4 修改 os\_cpu.h

前面我们已经介绍了移植过程中要修改的三个文件，首先我们来看 os\_cpu.h：

```
void      OS_CPU_SysTickHandler(void);
void      OS_CPU_SysTickInit(void);
INT32U    OS_CPU_SysTickClkFreq(void);
```

将以上三个文件注释掉即可。

#### 48.5. 3.5 修改 os\_cpu.c

把 OS\_CPU\_SysTickHandler(), OS\_CPU\_SysTickInit() 及

```
#define OS_CPU_CM3_NVIC_ST_CTRL      (*(volatile INT32U *)0xE000E010)
#define OS_CPU_CM3_NVIC_ST_RELOAD   (*(volatile INT32U *)0xE000E014)
#define OS_CPU_CM3_NVIC_ST_CURRENT  (*(volatile INT32U *)0xE000E018)
#define OS_CPU_CM3_NVIC_ST_CAL      (*(volatile INT32U *)0xE000E01C)
#define OS_CPU_CM3_NVIC_PRIO_ST     (*(volatile INT8U *)0xE000ED23)

#define OS_CPU_CM3_NVIC_ST_CTRL_COUNT      0x00010000
#define OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC   0x00000004
#define OS_CPU_CM3_NVIC_ST_CTRL_INTEN     0x00000002
#define OS_CPU_CM3_NVIC_ST_CTRL_ENABLE    0x00000001
#define OS_CPU_CM3_NVIC_PRIO_MIN           0xFF
```

注释掉（前面加#if 0，后面加#endif 就能注释掉）

### 48.6. 3.6 修改 os\_cpu\_a.asm

由于编译器的原因：

要将下面的 PUBLIC 改为 EXPORT

即：

```
351.          PUBLIC  OS_CPU_SR_Save                ; Functions declared in this file
352.          PUBLIC  OS_CPU_SR_Restore
353.          PUBLIC  OSStartHighRdy
354.          PUBLIC  OSCtxSw
355.          PUBLIC  OSIntCtxSw
356.          PUBLIC  OS_CPU_PendSVHandler
```

改为：

```
357.          EXPORT  OS_CPU_SR_Save                ; Functions declared in this file
358.          EXPORT  OS_CPU_SR_Restore
359.          EXPORT  OSStartHighRdy
360.          EXPORT  OSCtxSw
361.          EXPORT  OSIntCtxSw
362.          EXPORT  OS_CPU_PendSVHandler
```

下面这个也要修改下

原来的：

```
363.          RSEG CODE:CODE:NOROOT(2)
```

修改后：

```
364.          AREA |.text|, CODE, READONLY, ALIGN=2 ;AREA |.text| 选  
择段 |.text|。  
365.                                     ;CODE 表示代码段，  
READONLY 表示只读（缺省）  
366.                                     ;ALIGN=2 表示 4 字节对齐。  
若 ALIGN=n，这 2^n 对齐  
367.          THUMB ;Thumb 代码  
368.          REQUIRE8 ;指定当前文件要求堆  
栈八字节对齐  
369.          PRESERVE8 ;令指定当前文件保持  
堆栈八字节对齐
```

### 48.7. 3.7 修改 os\_dbg.c

将 os\_dbg.c 中

```
370.  #define OS_COMPILER_OPT __root
```

修改为：

```
371.  #define OS_COMPILER_OPT // __root
```

这个问题也是由编译器不同而产生的。

### 48.8. 3.8 修改 startup\_stm32f10x\_hd.s

修改完了这几个必要的部分后，有一处我们也必须要注意的。因为我们的移植是使用标准外设库 CMSIS 中 startup\_stm32f10x\_hd.s 作为启动文件的，还没有设置 OS\_CPU\_SysTickHandler。而 startup\_stm32f10x\_hd.s 文件中，PendSV 中断向量名为 PendSV\_Handler，因此只需把所有出现 PendSV\_Handler 的地方替换成 OS\_CPU\_PendSVHandler 即可。

至此，修改 uC/OS-II 代码就差不多结束，剩下的，就是编写我们自己的代码。

### 48.9. 3.9 编写 includes.h

includes.h 是保存全部头文件的头文件，方便我们理清工程函数思路。先给大家看我们用到的头文件，以便让大家知道我们的工程是怎样的一个架构。

```
372.  #ifndef __INCLUDES_H__
373.  #define __INCLUDES_H__
374.
375.  #include "stm32f10x.h"
376.  #include "stm32f10x_rcc.h" //SysTick 定时器相关
377.
378.  #include "ucos_ii.h" //uC/OS-II 系统函数头文件
379.
380.  #include "BSP.h" //与开发板相关的函数
381.  #include "app.h" //用户任务函数
382.  #include "led.h" //LED 驱动函数
383.
384.  #endif //__INCLUDES_H__
```

### 48.10. 3.10 编写 BSP

在前面我们讲到 SysTick 定时器我们自己定义，因此在 BSP.c 中我们加入我们自己的定义并在 BSP.h 中声明这个函数。这个函数需要添一个头文件 stm32f10x\_rcc.h

另外，我们也需要编写一个开发板初始化启动函数 BSP\_Init()，包含设置系统时钟，初始化硬件。

#### 48.10.1. BSP.C 文件代码

```
385.  #include "includes.h"
386.
387.  /*
388.   * 函数名: BSP_Init
389.   * 描述   : 时钟初始化、硬件初始化
390.   * 输入   : 无
391.   * 输出   : 无
392.   */
393.  void BSP_Init(void)
394.  {
```

```

395.         SystemInit();           /* 配置系统时钟为 72M */
396.         SysTick_init();          /* 初始化并使能 SysTick 定时
器 */
397.         LED_GPIO_Config();       /* LED 端口初始化 */
398.     }
399.
400.     /*
401.     * 函数名: SysTick_init
402.     * 描述   : 配置 SysTick 定时器
403.     * 输入   : 无
404.     * 输出   : 无
405.     */
406. void SysTick_init(void)
407. {
408.     SysTick_Config(SystemFrequency/OS_TICKS_PER_SEC); //初始化
并使能 SysTick 定时器
409. }

```

## 48.10.2. BSP.h 头文件

```

410.
411. #ifndef __BSP_H
412. #define __BSP_H
413.
414. void SysTick_init(void);
415. void BSP_Init(void);
416.
417. #endif // __BSP_H

```

## 48.11. 3.11 编写 stm32f10x\_it.c

需要在 stm32f10x\_it.c 添加 SysTick 中断的处理代码:

```

418. void SysTick_Handler(void)
419. {
420.     OSIntEnter();
421.     OSTimeTick();
422.     OSIntExit();
423. }

```

因为调用 uC/OS-II 的函数, 所以这样之后, 时钟也配置好了。下面我们可以创建任务了。

## 48.12.3.12 创建任务

### 48.12.1. 编写 app\_cfg.h

用来设置任务的优先级和栈大小

```
424.  #ifndef __APP_CFG_H__
425.  #define __APP_CFG_H__
426.
427.      /*****设置任务优先级*****/
428.  #define STARTUP_TASK_PRIO      4
429.
430.      /*****设置栈大小（单位为 OS_STK ）*****/
431.  #define STARTUP_TASK_STK_SIZE  80
432.
433.  #endif
```

### 48.12.2. 编写 app.c

这个是创建 LED 显示任务

```
434.  #include "includes.h"
435.
436.  void Task_LED(void *p_arg)
437.  {
438.      (void)p_arg;    // 'p_arg' 并没有用到，防止编译器提示警告
439.      while (1)
440.      {
441.          LED1( ON );
442.          OSTimeDlyHMSM(0, 0,0,500);
443.          LED1( OFF);
444.
445.          LED2( ON );
446.          OSTimeDlyHMSM(0, 0, 0,500);
447.          LED2( OFF);
448.
449.          LED3( ON );
450.          OSTimeDlyHMSM(0, 0,0,500);
451.          LED3( OFF);
452.      }
453.  }
```

### 48.12.3. 编写 app.h 头文件

```
454.  #ifndef __APP_H__
455.  #define __APP_H__
456.
457.      /***** 用户任务声明 *****/
458.  void Task_LED(void *p_arg);
459.
```



```
460. #endif // _APP_H_
```

### 48.13. 3.13 main 函数

```
461. #include "includes.h"
462.
463. static OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE]; //定义栈
464.
465. int main(void)
466. {
467.     BSP_Init();
468.     OSInit();
469.     OSTaskCreate(Task_LED, (void *)0,
470.         &startup_task_stk[STARTUP_TASK_STK_SIZE-1], STARTUP_TASK_PRIO);
471.
472.     OSStart();
473.     return 0;
474. }
```

编译之后，发现没错误了，下载下去看下灯闪了，哈哈，成功了。

简单的 uC/OS 移植就这样完成了，难不？

## 49. 4. 运行多任务

移植 uC/OS-II 弄好了，那运行多任务更简单。

注意哦，这里说的单任务、多任务是对于用户任务而言的。刚才那个实验，实际上也有两个任务，一个我们创建的 LED 任务，另外一个系统是自带的空闲任务。

现在，我们要做的实验就是：主任务 Task\_Start 先创建，再在主任务运行时创建两个任务 Task\_LED2 和 Task\_LED3。三个任务都分别控制 3 个 LED 灯。

这次，我们只需修改 4 个文件即可完成这次实验：main.c、app.c、app.h、app\_cfg.h，其他的都是跟原来工程一样的。

这次都是依葫芦画瓢，就不讲解了，直接上代码：

### 49.1. 4.1 修改 app.c

```
475.  #include "includes.h"
476.
477.      OS_STK task_led2_stk[TASK_LED2_STK_SIZE];           //定义栈
478.      OS_STK task_led3_stk[TASK_LED3_STK_SIZE];           //定义栈
479.
480.      //主任务
481.  void Task_Start(void *p_arg)
482.  {
483.      (void)p_arg;                                           // 'p_arg' 并没有用
484.      //到，防止编译器提示警告
485.      OSTaskCreate(Task_LED2, (void *)0,                   //创建任务 2
486.                  &task_led2_stk[TASK_LED2_STK_SIZE-1], TASK_LED2_PRIO);
487.
488.      OSTaskCreate(Task_LED3, (void *)0,                   //创建任务 3
```

```

489.                                     &task_led3_stk[TASK_LED3_STK_SIZE-
490. 1], TASK_LED3_PRIO);
491.     while (1)
492.     {
493.         LED1( ON );
494.         OSTimeDlyHMSM(0, 0,0,100);
495.         LED1( OFF);
496.         OSTimeDlyHMSM(0, 0,0,100);
497.     }
498. }
499.
500.     //任务 2
501. void Task_LED2(void *p_arg)
502. {
503.     (void)p_arg;
504.     while (1)
505.     {
506.         LED2( ON );
507.         OSTimeDlyHMSM(0, 0,0,200);
508.         LED2( OFF);
509.         OSTimeDlyHMSM(0, 0,0,200);
510.     }
511. }
512.
513.     //任务 3
514. void Task_LED3(void *p_arg)
515. {
516.     (void)p_arg;
517.     while (1)
518.     {
519.         LED3( ON );
520.         OSTimeDlyHMSM(0, 0,0,300);
521.         LED3( OFF);
522.         OSTimeDlyHMSM(0, 0,0,300);
523.     }
524. }
525.
526.

```

#### 49.2. 4.2 编写 app.h

```

527. #ifndef _APP_H_
528. #define _APP_H_
529.
530.     /***** 用户任务声明 *****/
531. void Task_Start(void *p_arg);
532. void Task_LED2(void *p_arg);
533. void Task_LED3(void *p_arg);
534. #endif // _APP_H_

```

#### 49.3. 4.3 编写 app\_cfg.h

```

535. #ifndef _APP_CFG_H_
536. #define _APP_CFG_H_
537.

```

```

538.
539.      /*****设置任务优先级*****/
540.  #define STARTUP_TASK_PRIO      4
541.  #define TASK_LED2_PRIO        5
542.  #define TASK_LED3_PRIO        6
543.
544.      /*****设置栈大小（单位为 OS_STK ）*****/
545.  #define STARTUP_TASK_STK_SIZE  80
546.  #define TASK_LED2_STK_SIZE     80
547.  #define TASK_LED3_STK_SIZE     80
548.
549.  #endif

```

#### 49.4. 4.4 编写 main.c

```

550.      /***** (C) COPYRIGHT 2011 野火嵌入式开发工作
室 *****/
551.      * 文件名   : main.c
552.      * 描述     : 建立 3 个任务，每个任务控制一个 LED，以固定的频率轮流闪烁
（频率可调）。
553.      * 实验平台：野火 STM32 开发板
554.      * 库版本   : ST3.0.0
555.      *
556.      * 作者     : fire   QQ: 313303034
557.      * 博客     : firestm32.blog.chinaunix.net
558.      *****/
559.  #include "includes.h"
560.
561.      OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE];      //定义
栈
562.
563.  int main(void)
564.  {
565.      BSP_Init();
566.      OSInit();
567.      OSTaskCreate(Task_Start, (void *)0,
568.                  &startup_task_stk[STARTUP_TASK_STK_SIZE-
1], STARTUP_TASK_PRIO);
569.
570.      OSStart();
571.      return 0;
572.  }
573.
574.      /***** (C) COPYRIGHT 2011 野火嵌入式开发工作
室 *****END OF FILE*****/

```

编译一下，下载到开发板，就可以看到 3 个 LED 以不同频率闪烁

这个实验是按葫芦画瓢的，应该容易明白吧？呵呵，这次的教程就到这里结束。

## 50. 5。升级到最新版本 V2.90

### 50.1. 5.1 不得不说的这些话儿

写这小节之前，首先非常感谢网友：i55x 提出的批评。

详情请看：批判一下“野火开源《从 0 开始移植 uc0s 到野火 stm32 开发板》pdf 教程”

[http://www.ourdev.cn/bbs/bbs\\_content.jsp?bbs\\_sn=5252775&bbs\\_page\\_no=1&bbs\\_id=1008](http://www.ourdev.cn/bbs/bbs_content.jsp?bbs_sn=5252775&bbs_page_no=1&bbs_id=1008)

呵呵，网友提出的批评确实非常有道理的。

5. 我们的教程选用的 uC/OS-II 版本为 V2.86，这个版本确实是有 Debug 的，所以网友提出的批评是正确的。

不过，我也解释一下为啥要选用 V2.86 这个版本：因为我们这个是教程，官方已经提供了这个版本给我们参考，我们就可以轻松讲解给初学者，以便初学者容易入门。作为初学者，不必追求完美的版本，而是追求通俗易懂的版本入门。事实上，我们本来就有打算写更新到最新版本的教程，只是限于时间的关系，没来得及写。

6. 我们使用的工作是 MDK，这个确实在调试 uC/OS-II 的时候非常痛苦，尤其是面对栈溢出问题。另外，我们在使用 MDK 优化的时候，也发现 MDK 的优化有问题，导致函数结果与不开优化时不一样。

我们也知道 IAR 有 uC/OS plug-in，正在考虑是否换成 IAR。因为我们之前的教程一直围绕着 MDK 来讲的，如果换成 IAR，相信不少的朋友会不习惯。

真的，非常感谢网友：i55x 提出的批评。也欢迎各位高手提出批评和建议，没有你们的批评，我们就不会有进步！！！！

好了，现在回归正题。

### 50.2. 5.2 从 V2.86 升级到 V2.90 说明

前面已经讲过，移植的时候，我们只需修改几个文件即可。升级其实也很简单，对于原来不需要修改的文件，我们可以直接通过简单的替换就可以；对于需要修改的文件，依葫芦画瓢就行。

首先，我们需要从官网下载 uC/OS-II 的最新版本。

下载地址为：[http://micrium.com/page/downloads/source\\_code](http://micrium.com/page/downloads/source_code)

目前最新的版本为 V2.90，而之前我们提供的是 V2.86，前面已经说到 V2.86 是一个有 Debug 的版本，所以有必要进行升级。

下载最新版本 V2.90 后，首先当然先看下更新文档：

[Micrium-uCOS-II-V290\Micrium\Software\uCOS-II\Doc\ReleaseNotes.PDF](#)

我们的版本是 V2.86，所以从 V2.87 开始看起。事实上，我们只需看我们移植过程中需要更改的那两个文件就可以了：os\_cfg.h 和 os\_dbg\_r.c。

os\_dbg\_r.c，我们暂时还没讲到，所以也可以不去看，**也就是说，我们只需看 os\_cfg.h 的更新就好了**。当然，如果你要了解 uC/OS 的更新细节，其他的内容也最好看下。

### 50.2.1. v2.87 更新日志

在 V2.87 里，更新的文件有：

OS\_CFG.H、OS\_TIME.C、OS\_CORE.C、OS\_CORE.C、OS\_MBOX.C、  
OS\_MUTEX.C、OS\_Q.C、OS\_SEM.C、OS\_TASK.C、OS\_TMR.C

在 OS\_CFG.H 文件中，谈到了将几个 size 配置改为 EN 使能选项，可以减少内存浪费。呵呵，官方这个更新，可以为我们减少不少配置烦恼，使得配置更容易。既然更新了 OS\_CFG.H，那我们就得重新配置内核。

### 50.2.2. v2.88 更新日志

在 V2.88 里，更新的文件有：

OS\_CORE.C、uCOS\_II.H

没修改 OS\_CFG.H，跳过不看。

### 50.2.3. v2.89 更新日志

在 V2.89 里，更新的文件有：

All Files、OS\_CORE.C、OS\_MEM.C、OS\_MUTEX.C、OS\_TASK.C

All Files 中谈到，对于每个常量，后面都添加‘u’，为啥呢？

例如：

```
575.  #define num      128
```



那么编译器编译的时候，默认会把 128 当成 signed 型来看待，这时，会用 int 型来保存 128。

而添加 'u' 后，即

```
576. #define num 128u
```

告诉了编译器，这个是 unsigned 型，所以编译器就会用 unsigned char 来保存 128。

显然，这个更新无关重要，不影响我们更新 uC/OS-II 。

#### 50.2.4. v2.90 更新日志

在 V2.90 里，更新的文件有：

All Files 、 OS\_CORE.C 、 uCOS\_II.H 、 OS\_FLAG.C 、 OS\_MBOX.C 、 OS\_MEM.C 、 OS\_MUTEX.C 、 OS\_Q.C 、 OS\_SEM.C 、 OS\_TASK.C 、 OS\_TMR.C

All Files 中谈到，根据 MISRA-C 2004 的规则来修改所有的文件：

8.5 –不得在头文件中定义函数和变量

14.7–函数都应该有一个单点退出，即 return。

15.2–switch 中，每个非空的 case 都应该有个 break 来中止。

15.4–只有一个 case——switch 的表达不应该是一个布尔值。（只有一个，那应该用 if）

17.4–数组索引只适用于对象定义为一个数组类型

17.4 – 不应该使用指针运算

不影响 OS\_CFG.H ， 原来的 OS\_CFG.H 中并没有定义变量和函数。

所以，上面那么多更新，就 V2.87 更新的影响最大，但这个配置文件，我们可以直接替换后重新配置就行。

好了，该谈更新步骤了。

### 50.3. 5.3 更新步骤

这里，我们利用前面已经做好的 STM32+UCOS+LED(单任务) 的程序来讲解如何更新到最新版本：

34. 先把 STM32+UCOS+LED(单任务) 复制一份出来，命名为 STM32+UCOS+LED(V2.90 单任务)，这样避免我们以后的版本混乱。
35. 复制下载的 uC/OS-II 新版本附件里 Micrium-uCOS-II-V290\Micrium\Software\uCOS-II\Source 文件夹下除 os\_cfg\_r.h 和 os\_dbg\_r.c 外的其他文件到我们的项目 uCOS-II\Source 文件夹下，直接替换原来已有的文件。
36. 复制刚才提到的 os\_cfg\_r.h 文件到我们的项目 APP 文件夹下
37. 打开 os\_cfg\_r.h，由到了配置、裁剪 uC/OS-II 的时候了。这里，我们仅仅是简单的 LED 显示，不需要使用通信等模块，可以这样配置：

```
577.  #define OS_APP_HOOKS_EN           0u
578.  #define OS_DEBUG_EN               0u
579.  #define OS_EVENT_MULTI_EN         0u
580.  #define OS_EVENT_NAME_EN          0u
581.
582.  #define OS_TICKS_PER_SEC           1000u
583.                                     // 设置每秒中断次数，我们设置为每 1ms 中断一
                                     次比较合适
584.
585.                                     /*裁剪其他模块*/
586.  #define OS_FLAG_EN                 0u
587.  #define OS_MBOX_EN                 0u
588.  #define OS_MEM_EN                  0u
589.  #define OS_Q_EN                    0u
590.  #define OS_TMR_EN                  0u
```

保存为 os\_cfg.h，本来就已经存在这样的一个文件，所以可以直接覆盖保存。

38.用 MDK 打开工程，编译一下，提示出错：

..\Output\STM32-DEMO.axf: Error: L6218E: Undefined symbol

OSTaskReturnHook (referred from os\_task.o).

这里说我们没有定义 OSTaskReturnHook，搜索一下：Edit——Find in Files——填入“OSTaskReturnHook”，找到两个地方有 OSTaskReturnHook。

一个在 ucos\_ii.h 中声明（第 1313 行）：

```
591. void OSTaskReturnHook (OS_TCB *ptcb);
```

另外一个在 os\_task.c 中调用（第 1206 行）

```
592. void OS_TaskReturn (void)
593. {
594.     OSTaskReturnHook (OSTCBCur);
595.
596. #if OS_TASK_DEL_EN > 0u
597.     (void)OSTaskDel (OS_PRIO_SELF);
598. #else
599.     for (;;) {
600.         OSTimeDly (OS_TICKS_PER_SEC);
601.     }
602. #endif
603. }
```

只有声明，没有定义。呵呵，又是一个钩子函数，显然，需要我们去定义。（这个前面讲移植的时候说过的哦，有些钩子函数是需要我们去定义！）

可以在 os\_cpu\_c.c 中添加这段代码：

```
604. #if OS_CPU_HOOKS_EN > 0u && OS_VERSION > 290u
605. void OSTaskReturnHook (OS_TCB *ptcb)
606. {
607.     (void)ptcb;
608. }
609. #endif
```

编译一下，没错了，下载到板子上验证，LED 闪烁了，呵呵，升级完毕。

说一下，关于 `os_dbg_r.c`，我们这里还没用到，不过也先复制到工程 `uCOS-II\Ports` 文件夹下，以便以后用得着的时候再去重新网上下载附件就麻烦了。

#### 50.4. 5.4 更新其他工程

呵呵，前面都已经讲到，升级的步骤，其实就是复制替换到原来旧版本的文件，然后重新配置 `os_cfg.h`，可能需要在 `os_cpu_c.c` 里创建某些钩子函数。

所以，这里我们可以直接把我们升级好的 `uC/OS-II` 工程里的文件复制到还没升级好的工程里。需要注意的是 `os_cfg.h` 和 `os_cpu_c.c`，这部分的代码，还是需要根据工程的需要进行某些简单的修改。

##### 50.4.1. 更新 STM32+UCOS+LED 多任务 的例子

12. 先把 `STM32+UCOS+LED(多任务)` 复制一份出来，命名为 `STM32+UCOS+LED(V2.90 多任务)`，这样避免我们以后的版本混乱。
13. 把 `STM32+UCOS+LED(V2.90 单任务)` 里的 `uCOS-II` 文件夹整个复制到 `STM32+UCOS+LED(V2.90 多任务)` 里，覆盖同名文件。
14. 打开 `STM32+UCOS+LED(V2.90 多任务)\APP` 下的 `os_cfg.h` 命名为 `os_cfg_V2.86.h`。这样子是为了以后方便了解原先如何配置的。当然，如果你熟悉 `uC/OS-II` 的配置，也可以把这步删掉。
15. 把 `STM32+UCOS+LED(V2.90 单任务)\APP` 下的 `os_cfg.h` 和 `os_cfg_r.h` 复制到 `STM32+UCOS+LED(V2.90 多任务)\APP` 下。

16. 然后参考 `os_cfg_V2.86.h` 文件，配置新的 `os_cfg.h`。因为这个多任务跟单任务都没用到通信这些模块，所以这里第四步直接复制过来的 `os_cfg.h` 就可以直接用。

打开工程，编译一下没问题，下载到板子上，呵呵，没问题了……更新就这样完成了，简单吧？

## 51. 6、移植计算器

### 51. 1. 6.1 处理外部中断

上面都是简单的点亮 LED，相信不少朋友看到后，对 uC/OS-II 下编程还是会有所不解的，尤其是在 uC/OS-II 里如何处理外部中断。

其实，uC/OS-II 里的中断 跟 裸机的中断 没啥区别，唯一的区别就是 uC/OS-II 中断服务中需要调用中断服务函数 OSIntEnter 和 OSIntExit。

OSIntEnter 用来统计中断嵌套层数

OSIntExit 当中断嵌套层数为 0，OSLockNesting 为 0，有最高优先级进入就绪表时，就会进行任务切换。

M3 移植过程中，把 uC/OS-II 的任务切换编程为最低中断优先级，即在中断函数里，不会发生任务切换这类的事情，等中断结束后调用中断服务函数 OSIntExit() 进行任务调度（中断嵌套层数为 0 且 OSLockNesting 为 0，有最高优先级进入就绪表）。而裸机的时候，中断返回到我们被中断的函数，两者的差别就在开头和结尾一步。

参考在 stm32f10x\_it.c 的 SysTick 中断的处理代码：

```
610. void SysTick_Handler(void)
611. {
612.     OSIntEnter();           //开头添加 OSIntEnter
613.     OSTimeTick();           //中间就是中断函数需要处理的代码
614.     OSIntExit();            //结尾添加 OSIntExit
615. }
```

### 51.2. 6.2 移植步骤

这次我们在《30-计算器-20111214》这个版本的基础上进行移植，这版本重新整理了 LCD 接口，使用起来会方便很多，也修正了一个计算 Debug，详情可以看里面的 USER\提示.txt 文件。

uC/OS-II 工程就用 STM32+UCOS+LED(V2.90 多任务) 为基础，进行移植。

思路就是把计算器工程的代码复制到 uC/OS-II 工程里，设置好中断服务函数，编译出错后，再修改代码。为了便于理解，这次的计算器，我们同样不会用通信功能。

当然，也可以往计算器里面移植 uC/OS-II，但那样就不利于了解 uC/OS-II 里如何处理外部中断。

#### 51.2.1. 具体步骤

11. 把 STM32+UCOS+LED(V2.90 多任务) 工程文件夹命名为“STM32+UCOS+计算器(V290 不通信)”。
12. 把 30-计算器-20111214\USER 文件夹下涉及硬件功能的文件都 copy 到 STM32+UCOS+计算器(V290 不通信)\BSP 文件夹下。

这些文件为：

ascii.h

asc\_font.h

diskio.c

diskio.h

exti.c

exti.h

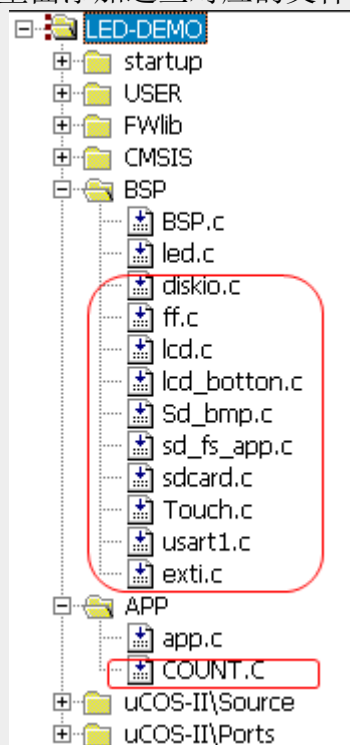
ff.c

ff.h

|              |              |
|--------------|--------------|
| ff_config.h  |              |
| integer.h    |              |
| lcd.c        | lcd.h        |
| lcd_botton.c | lcd_botton.h |
| sdcard.c     | sdcard.h     |
| Sd_bmp.c     | Sd_bmp.h     |
| sd_fs_app.c  | sd_fs_app.h  |
| Touch.c      | Touch.h      |
| usart1.c     | usart1.h     |

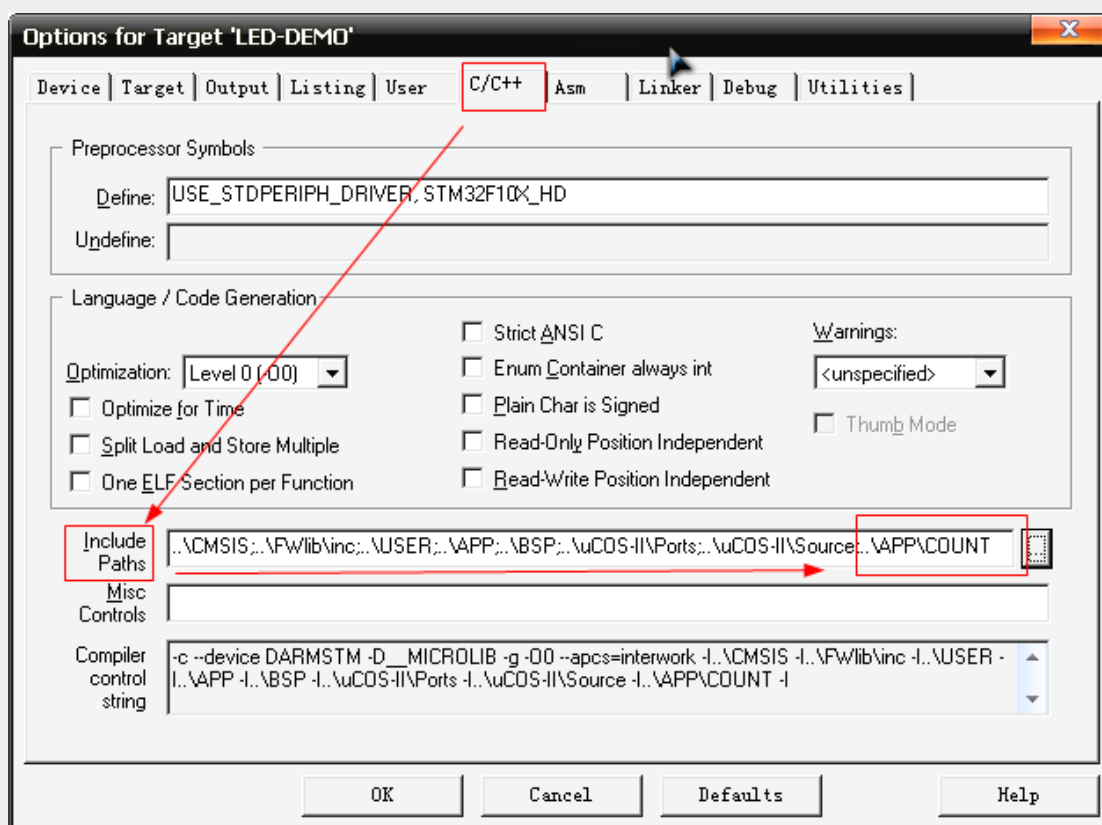
13. 把 30-计算器-20111214\USER 文件夹下 COUNT 文件夹复制到 STM32+UCOS+计算器(V290 不通信)\APP 文件夹下

14. 打开工程，也往工程里面添加这些对应的文件。



记得别忘了在 Include Path 里添加 ..\APP\COUNT





15. 编译一下，出现各类异常的错误：

40. ..\BSP\Touch.c(11): error: #5: cannot open source input file "systick.h": No such file or directory

原因： 找不到"systick.h"文件

解决方法： 这个是延时头文件，我们这里用 uC/OS-II 的延时函数，而不用之前的延时函数，所以，直接把这个注释掉就可以。

41. ..\BSP\usart1.c(28): error: #20: identifier "USART\_InitTypeDef" is undefined  
还有几个与之类似的……

原因： "USART\_InitTypeDef"没定义。

解决方法：在原来计算器工程那里跳到定义处，发现这个是在 "stm32f10x\_usart.h" 里定义的。

呵呵，还没配置好库文件，打开 STM32+UCOS+计算器(V290 不通信)\USER\stm32f10x\_conf.h。取消 #include "stm32f10x\_usart.h" 的注释。

顺便也取消 #include "stm32f10x\_exti.h" 的注释，因为触摸中断用到。

42. ..\APP\COUNT\COUNT\_CFG.H(81): error: #5: cannot open source input file "systick.h": No such file or directory

这个，是计算器的移植配置文件，为了便于下次移植方便，我们把

```
616.          /*****定义延时函数
          *****/
617.  #ifndef DELAY
618.      #include "systick.h"
619.      #define DELAY delay_ms(300) //延时函数，避免触摸屏按的速度太
        快。
620.      #define DELAY_MS(x) delay_ms(x)
621.  #endif
```

替换为：

```
622.  #ifndef DELAY
623.      #if 0
624.          #include "systick.h"
625.          #define DELAY delay_ms(300) //延时函数，避免触摸屏按的
        速度太快。
626.          #define DELAY_MS(x) delay_ms(x)
627.      #else
628.          #include <ucos_ii.h>
629.          #define DELAY OSTimeDlyHMSM(0, 0, 0, 300);
630.          #define DELAY_MS(x) OSTimeDlyHMSM(0, 0, 0, x);
631.      #endif
632.  #endif //DELAY
```

16. 继续重新编译一下，出现其他问题：

12. ..\BSP\Touch.c(729): warning: #223-D: function "delay\_ms" declared implicitly

原因：没有定义"delay\_ms"

解决方法：uC/OS-II 里用的是 uC/OS 自带的函数，我们在 Touch.c 文件上面添加：

```
633. #ifndef DELAY
634. #if 0
635. #include "systick.h"
636. #define DELAY delay_ms(300) //延时函数，避免触摸屏按的
    速度太快。
637. #define DELAY_MS(x) delay_ms(x)
638. #else
639. #include <ucos_ii.h>
640. #define DELAY OSTimeDlyHMSM(0, 0, 0, 300);
641. #define DELAY_MS(x) OSTimeDlyHMSM(0, 0, 0, x);
642. #endif
643. #endif //DELAY
```

把所有的 delay\_ms 函数替换为 DELAY\_MS 宏定义。

13. ..\BSP\Touch.c(794): warning: #223-D: function "delay\_init" declared implicitly

原因：没有定义"delay\_init"延时初始化函数。

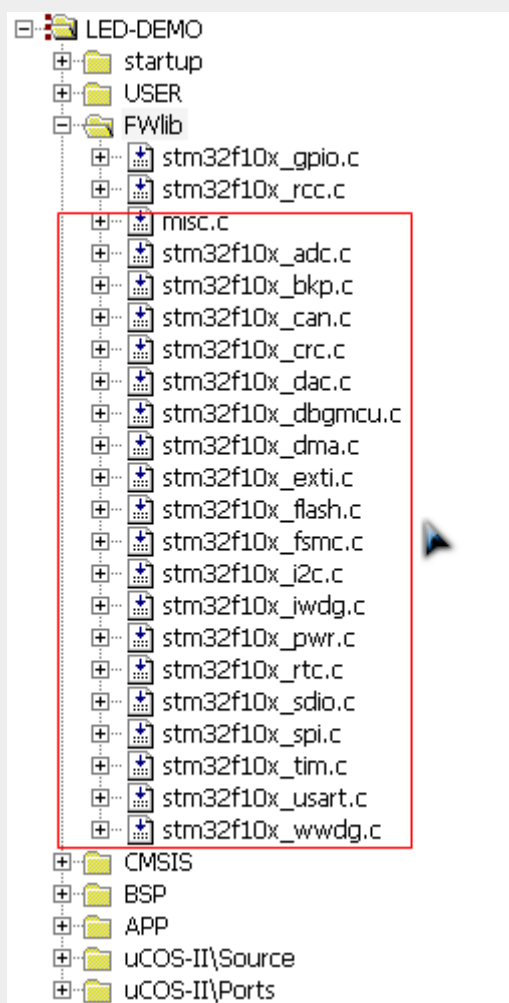
解决方法：晕，估计队友不小心在 TouchI\_Calibrate 函数里添加了延时初始化函数，这个应该在 main 函数了实现的，所以直接删掉  
^\_^

14. ..\Output\STM32-DEMO.axf: Error: L6218E: Undefined symbol EXTI\_Init (referred from exti.o).

原因：没定义 EXTI\_Init

解决方法：在原来工程那里跳到定义处，发现这个是在"  
stm32f10x\_exti.c"里定义的。

呵呵，还没给工程添加好库文件。把 STM32+UCOS+计算器(V290  
不通信)\FWlib\src 里的 C 文件都添加到工程里面。



其他类似的错误，也是添加库后就解决。

17. 再次编译一下，也出现了错误：

4. ..\APP\COUNT\COUNT.C(374): warning: #223-D: function "ScanTouch" declared implicitly

原因：没定义 ScanTouch

解决方法：呵呵，这个是读取触摸按键的，计算器工程那里是放在 main.c 里的，我们就在 app.c 里添加这个函数功能：

```

644.      /*****
645.      * 名    称 ScanTouch
646.      * 功    能：扫描触摸屏，识别按下的键
647.      * 入口参数：key 指针，用于保存识别到按下的键。
648.      * 出口参数：无
649.      * 说    明：函数，在 CountMain 函数里被调用。
650.      * 调用方法：ScanTouch(&Key) ,注：此处 Key 定义为 char 型
651.      * 完成程度：测试成功，by Song 2011.12.15
652.      *****/
653.      //触摸读取按键
654.      //设置触摸扫描键盘设置
655.  #define stx      94
656.  #define sty      8
657.  #define tx1     120
658.  #define txc      36
659.  #define ty1      40
660.  #define tyc      65
661.  char ScanTouch(char *key)                //测试成功
662.  {
663.      u16 x,y;
664.      if(Get_touch_place(&x, &y)==0)      /*如果触笔有按下
665.      {
666.
667.          x = 240-x;
668.          if(x>stx          && x<=tx1 )      //x=0~40
669.          {
670.              if          (y>=sty          && y<=ty1          ) *key='7
671.              ';
672.              else if (y>=sty+tyc          && y<=ty1+tyc          ) *key='8
673.              ';
674.              else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='9
675.              ';
676.              else if (y>=sty+tyc*3 && y<=ty1+tyc*3 ) *key='+'
677.              ';
678.              else if (y>=sty+tyc*4 && y<=ty1+tyc*4 ) *key='('
679.              ';
680.              else return 0;
681.          }
682.          else if(x>=stx+txc          && x<=tx1+txc )      //x=70~11
683.          0
684.          {
685.              if          (y>=sty          && y<=ty1          ) *key='4
686.              ';
687.              else if (y>=sty+tyc          && y<=ty1+tyc          ) *key='5
688.              ';
689.              else if (y>=sty+tyc*2 && y<=ty1+tyc*2 ) *key='6
690.              ';

```

```

682.         else if (y>=sty+tyc*3 && y<=tyl+tyc*3 ) *key='-
';
683.         else if (y>=sty+tyc*4 && y<=tyl+tyc*4 ) *key=')
';
684.         else return 0;
685.     }
686.     else if (x>=stx+txc*2 && x<=txl+txc*2 ) //x=140~1
80
687.     {
688.         if (y>=sty && y<=tyl ) *key='1
';
689.         else if (y>=sty+tyc && y<=tyl+tyc ) *key='2
';
690.         else if (y>=sty+tyc*2 && y<=tyl+tyc*2 ) *key='3
';
691.         else if (y>=sty+tyc*3 && y<=tyl+tyc*3 ) *key='*
';
692.         else if (y>=sty+tyc*4 && y<=tyl+tyc*4 ) *key='D
';
693.         else return 0;
694.     }
695.     else if (x>=stx+txc*3 && x<=txl+txc*3 ) //x=210~2
50
696.     {
697.         if (y>=sty && y<=tyl ) *key='0
';
698.         else if (y>=sty+tyc && y<=tyl+tyc ) *key='.'
';
699.         else if (y>=sty+tyc*2 && y<=tyl+tyc*2 ) *key='=
';
700.         else if (y>=sty+tyc*3 && y<=tyl+tyc*3 ) *key='/'
';
701.         else if (y>=sty+tyc*4 && y<=tyl+tyc*4 ) *key='C
';
702.         else return 0;
703.     }
704.     else return 0;
705.     return 1;
706. }
707. else return 0;
708. }

```

在 app.h 里声明:

```

709. extern char ScanTouch(char *key);

```

18. 编译一下, 又出现问题:

13. ..\APP\app.c(74): warning: #223-D: function "Get\_touch\_place" declared implicitly

原因: 没定义 Get\_touch\_place, 其实是没有包含头文件

解决方法：在 `includes.h` 里添加对应的头文件。为了方便，把其他用到的头文件也一起加进来：

```
710. #include "lcd.h" //LCD 接口
711. #include "sd_fs_app.h" //SD 卡文件系统接口
712. #include "Sd_bmp.h" //SD 卡解码 BMP 接口
713. #include "usart1.h" //串口 1
714. #include "Touch.h" //触摸
715. #include "COUNT.H" //计算器头文件
```

19. 编译一下，没报错了。但我们还没有编写计算任务，目前仅仅还是 LED 的任务。

在 `app.c` 里，先删掉任务 `Task_LED2` 和 `Task_LED3`，再创建计算任务 `Task_Count`，修改启动任务 `Task_START`：

```
716. static OS_STK Count_task_stk[COUNT_TASK_STK_SIZE]; //定义 Count 任务的
    栈
717.
718. /*****
    *****/
719. * 名 称 Task_Start
720. * 功 能：启动任务，创建另外一个任务，然后进入死循环闪烁 LED1
721. * 入口参数：无
722. * 出口参数：无
723. * 说 明：任务
724. * 调用方法：必须作为 OSTaskCreate 函数参数来调用
725. * 完成程度：测试成功，by Song 2011.12.15
726. *****/
727. void Task_Start(void *p_arg)
728. {
729.     (void)p_arg; // 'p_arg' 并没有用到，
    防止编译器提示警告
730.
731.     OSTaskCreate(Task_Count, (void *)0,
732.                 &Count_task_stk[COUNT_TASK_STK_SIZE-
733. 1], COUNT_TASK_PRIO);
734.
735.     while (1)
736.     {
737.         LED1( ON );
738.         OSTimeDlyHMSM(0, 0,0,500);
739.         LED1( OFF);
740.         OSTimeDlyHMSM(0, 0,0,500);
741.     }
742. /*****
    *****/
743. * 名 称 Task_Count
```

```

744.      * 功    能：计算任务，包括扫描触摸屏，识别按键，对输入的按键进行处
理。
745.      * 入口参数：无
746.      * 出口参数：无
747.      * 说    明：任务
748.      * 调用方法：必须作为 OSTaskCreate 函数参数来调用
749.      * 完成程度：测试成功，by Song 2011.12.15
750.      *****/
751.      void Task_Count(void *p_arg)
752.      {
753.          (void)p_arg;                // 'p_arg' 并没有用到，
防止编译器提示警告
754.          while (1)
755.          {
756.              CountMain();            //计算器函数
757.          }
758.      }

```

在 **app.h** 里声明 Task\_Count，并删掉任务 Task\_LED2 和 Task\_LED3 的声明：

```

759.      void Task_Count(void *p_arg);

```

在 **app\_cfg.h** 里设置优先级和栈：

```

760.      #ifndef __APP_CFG_H__
761.      #define __APP_CFG_H__
762.
763.          /*****设置任务优先级*****/
764.      #define STARTUP_TASK_PRIO      0                //开始启动的任务的优先
级
765.      #define COUNT_TASK_PRIO        5                //计算任务的优先级
766.
767.          /*****设置栈大小（单位为 OS_STK ）*****/
768.      #define STARTUP_TASK_STK_SIZE  80
769.      #define COUNT_TASK_STK_SIZE    100
770.
771.      #endif

```

## 20. 添加硬件初始化

在 BSP.c 里，修改 BSP\_Init 函数：

```

772.      void BSP_Init(void)
773.      {
774.          SystemInit();                /* 配置系统时钟为
72M */
775.          SysTick_init();              /* 初始化并使能 SysTick 定时器 */
776.          LED_GPIO_Config();           /* LED 端口初始化 */
777.          LCD_Init();
778.          USART1_Config();
779.          sd_fs_init();                /*文件系统初始化*/
780.          Touch_init();                /*触摸初始化*/
781.          while(Touch1_Calibrate() !=0); /*触摸品校准*/

```



```
782.         Lcd_show_bmp(0, 0, "/CAL.bmp");          /*在 LCD 上显示 SD 卡上存
放的 test.bmp 图片*/
783.         LCD_Str_O(29, 2, "10", BLACK);
784.     }
```

## 21. 添加中断函数

注意了，这里就告诉你怎么处理外部中断。

在 `stm32f10x_it.c` 里添加下面两个函数：

```
785.     /*
786.     * 函数名: SDIO_IRQHandler
787.     * 描述   : SDIO 全局中断请求服务程序
788.     * 输入   : 无
789.     * 输出   : 无
790.     */
791. void SDIO_IRQHandler(void)
792. {
793.     OSIntEnter();
794.     SD_ProcessIRQSrc(); /* 处理 SDIO 全部中断 */
795.     OSIntExit();       //调用中断服务函数
796. }
797.
798. /* XPT2046 中断处理函数 */
799. void EXTI9_5_IRQHandler(void)
800. {
801.     OSIntEnter();
802.     if(EXTI_GetITStatus(EXTI_Line6) != RESET)
803.     {
804.         touch_flag = 1; /* 触摸按下 */
805.         EXTI_ClearITPendingBit(EXTI_Line6);
806.     }
807.     OSIntExit();       //调用中断服务函数
808. }
```

其实就是开头和结尾分别多了个：`OSIntEnter()`和`OSIntExit()`。可以打开计算器工程里的 `stm32f10x_it.c` 看看就知道。

## 22. 编译一下，出现错误了：stm32f10x\_it.c(65): error: #20: identifier "touch\_flag" is undefined

原因：没声明 `touch_flag`

解决方法：在 **Touch.h** 里添加声明：

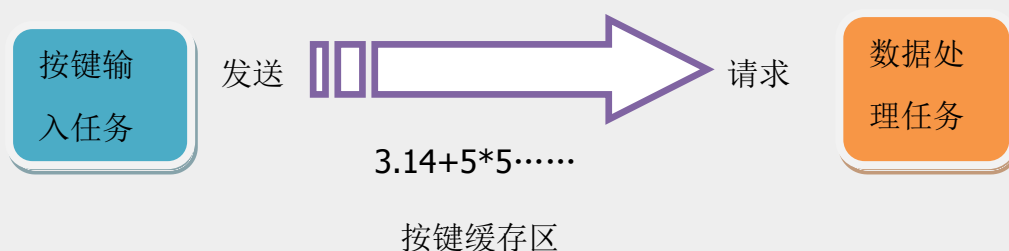
```
809. volatile extern unsigned char touch_flag;
```

呵呵，编译一下，就会发现没错的。烧到单片机上试试，记得要拷贝计算器模块用到的资源到 SD 卡哦。

辛苦了那么久，这部分终于讲完。其实修改的内容也不多，不过你们看着教程来修改，肯定也得花不少时间。

### 51.3. 6.3 增加通信功能——队列

这一小节，我们给计算器实验添加通信功能。把原来的计算器任务分为按键输入和数据处理并显示出来两个部分：



那样对于用户而已，或许他的输入速度较快，单片机还没来得及处理完数据，则队列就提供一个缓冲区来缓存数据，等单片机处理完一个数据时就会请求另外一个数据，继续处理。

下面，我们来在原来的《STM32+UCOS+计算器(V290 不通信)》修改步骤：

18. 复制《STM32+UCOS+计算器(V290 不通信)》出来，把文件夹命名为《STM32+UCOS+计算器(V290 队列)》

19. 修改 os\_cfg.h，把里面的 OS\_Q\_EN 宏定义定义为 1u:

```
810. #define OS_Q_EN 1u //允许使用消息队列
```

## 20. 修改 app.c 的各个任务:

```

811.  #include "includes.h"
      812.
      813.      /*-----设置任务堆栈-----*/
814.  static OS_STK Count_task_stk[COUNT_TASK_STK_SIZE];          //定义 Count 任务
      的栈
815.  static OS_STK Touch_task_stk[TOUCH_TASK_STK_SIZE];          //定义 Touch 任务
      的栈
      816.
      817.      /*-----设置队列参量-----*/
818.  #define N_MESSAGES 10u          //定义消息队列长度
819.  void *MsgGrp[N_MESSAGES];        //定义消息指针数
      组
      820.      OS_EVENT *Touch_Q;          //
      定义事件控制块
      821.
822.  char touchkey[N_MESSAGES];          //定义按键缓冲
      区
      823.      unsigned char err;          //
      队列返回错误类型
      824.
      825.
      826.      /******
      *****
      827.      * 名 称 Task_START
      828.      * 功 能: 启动任务, 创建其他两个任务, 然后进入死循环闪烁 LED1
      829.      * 入口参数: 无
      830.      * 出口参数: 无
      831.      * 说 明: 任务
      832.      * 调用方法: 必须作为 OSTaskCreate 函数参数来调用
      833.      * 完成程度: 测试成功, by Song 2011.12.15
      834.      * *****/
      835.  void Task_Start(void *p_arg)
      836.      {
      837.          (void)p_arg;          // 'p_arg' 并没有用到,
      防止编译器提示警告
      838.          SysTick_init();
      839.
      840.          Touch_Q=OSQCreate(&MsgGrp[0],N_MESSAGES);
      //创建消息队列
      841.
      842.          OSTaskCreate(Task_Touch,(void *)0,
      843.              &Touch_task_stk[TOUCH_TASK_STK_SIZE-
      1],TOUCH_TASK_PRIO); //创建触摸任务
      844.
      845.          OSTaskCreate(Task_Count,(void *)0,
      //创建计算任务
      846.              &Count_task_stk[COUNT_TASK_STK_SIZE-
      1],COUNT_TASK_PRIO);
      847.
      848.          while (1)
      849.          {
      850.              LED1( ON );
      851.              OSTimeDlyHMSM(0, 0,0,500);
      852.              LED1( OFF);
      853.              OSTimeDlyHMSM(0, 0,0,500);
      854.          }
      855.      }

```

```

856.
857.
858.      /*****
*****
859.      * 名    称 Task_Touch
860.      * 功    能: 触摸任务, 不断检测是否有发送触摸, 把按下的键发送到队列
里。
861.      * 入口参数: 无
862.      * 出口参数: 无
863.      * 说    明: 任务
864.      * 调用方法: 必须作为 OSTaskCreate 函数参数来调用
865.      * 完成程度: 测试成功, by Song 2011.12.15
866.      *****/
867. void Task_Touch(void *p_arg)
868. {
869.     char    i=0;                                //用于计数
870.     (void)p_arg;                                // 'p_arg' 并没有
用到, 防止编译器提示警告
871.     while (1)
872.     {
873.         if(ScanTouch(&touchkey[i]))            //按下键了
874.         {
875.             OSQPost(Touch_Q, &touchkey[i]);    //发送消息
876.             i++;                                //执行下个缓冲
区, 避免覆盖原来的按键数据
877.             if(i==N_MESSAGES) i=0;
878.             OSTimeDlyHMSM(0, 0, 0, 200);        //延时 200ms,
避免太灵敏
879.         }
880.         OSTimeDlyHMSM(0, 0, 0, 10);            //延时, 避免低
优先级无法执行
881.     }
882. }
883.
884.
885.      /*****
*****
886.      * 名    称 Task_Count
887.      * 功    能: 计算任务, 不断请求队列, 请求到数据后, 处理数据, 并在 LCD 显
示结果
888.      * 入口参数: 无
889.      * 出口参数: 无
890.      * 说    明: 任务
891.      * 调用方法: 必须作为 OSTaskCreate 函数参数来调用
892.      * 完成程度: 测试成功, by Song 2011.12.15
893.      *****/
894. void Task_Count(void *p_arg)
895. {
896.     char    key;
897.     char    CountStatus=calnot;                /
/计算状态
898.     struct  num_point tmp;                      /
/计算结果
899.     struct  str_point sp;                      /
/计算式子
900.     u8      Result[RN];                        //计算结果字符
串, 把计算结果转换为字符串
901.

```

```

902.         (void)p_arg;                                // 'p_arg' 并没有用
到,防止编译器提示警告
903.
904.         ClearSP(&sp);
905.         while (1)
906.         {
907.             key=((char *)OSQPend(Touch_Q,0,&err));
908.                                     //向 Touch_Q 队列请求消息。等待时限为
0,即无限等待下去,直到获取消息
909.
910.             /*****进入计算部分
******/
911.             switch(key)
912.             {
913.                 case '=': //-----计算式子-----
-----
914.                     if(CountStatus==caled)break;
915.                     CountStatus=caled;                /
/计算完毕
916.
917.                     tmp=calculate(sp.str,10);          /
/计算字符串结果
918.                     if(tmp.point==0)
919.                     {
920.                         PERROR;                          /
/输入式子有误
921.                         break;
922.                     }
923.                     CLEAN_R;                            /
/清屏
924.                     float2stre(tmp.result,Result);    /
/把计算结果转为字符串
925.                     PRINT_R(Result);                  /
/显示计算结果
926.                     break;

927.                 case 'C': //-----清空式子-----
-----
928.                     CountStatus=caling;
929.                     ClearSP(&sp);
930.                     CLEAN_R;                            /
/清屏
931.                     CLEAN_S;
932.                     break;

933.                 case 'D': //-----退格-----
-----
934.                     CountStatus=caling;
935.                     DelSP(&sp);
936.                     CLEAN_R;                            /
/清屏
937.                     CLEAN_S;
938.                     PRINT_S(sp.str);                  /
/更新屏幕字符串
939.                     break;

940.                 default: //-----输入式子-----
-----
- (剩下的就是运算表达式)
941.                     if(CountStatus==caled)            /
/如果计算完成后,就清空
942.                     {
943.                         CountStatus=caling;
944.                         ClearSP(&sp);
945.                         CLEAN_R;                      /
/清屏

```

```

946.                                CLEAN_S;
947.                                }
948.                                if (AddCheckSP (&sp, key))           /
/输入 key 有效
949.                                {
950.                                PRINT_S (sp.str);                     /
/更新屏幕字符串
951.                                }
952.                                } //switch
953.                                }
954.                                }

```

## 21. 修改 app.h (添加声明) :

```

955. #ifndef _APP_H_
956. #define _APP_H_
957.
958.     /***** 用户任务声明 *****/
959. void Task_Start(void *p_arg);
960. void Task_Count(void *p_arg);
961. void Task_Touch(void *p_arg);
962.
963. extern char ScanTouch(char *key); //触摸扫描程序
964.
965. #endif // _APP_H_

```

## 22. 修改 app\_cfg.h (设置优先级和栈大小) :

```

966. #ifndef _APP_CFG_H_
967. #define _APP_CFG_H_
968.
969.
970.     /***** 设置任务优先级 *****/
971. #define STARTUP_TASK_PRIO    0           //开始启动的任务的优先
级
972. #define TOUCH_TASK_PRIO      1           //触摸优先级
973. #define COUNT_TASK_PRIO      6           //计算优先级
974.
975.     /***** 设置栈大小 (单位为 OS_STK ) *****/
976. #define STARTUP_TASK_STK_SIZE 100
977. #define COUNT_TASK_STK_SIZE   100
978. #define TOUCH_TASK_STK_SIZE   100
979.
980. #endif

```

其实就是把 CountMain()函数的功能拆分成两个部分，用两个任务来实现，然后通过队列来通信。修改后，编译一下，再烧写到单片机上，计算器的功能能够正常使用，与没通信的看不出有啥区别，就这样完成了。