零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

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

0、友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，于野火同行，乐意惬无边。

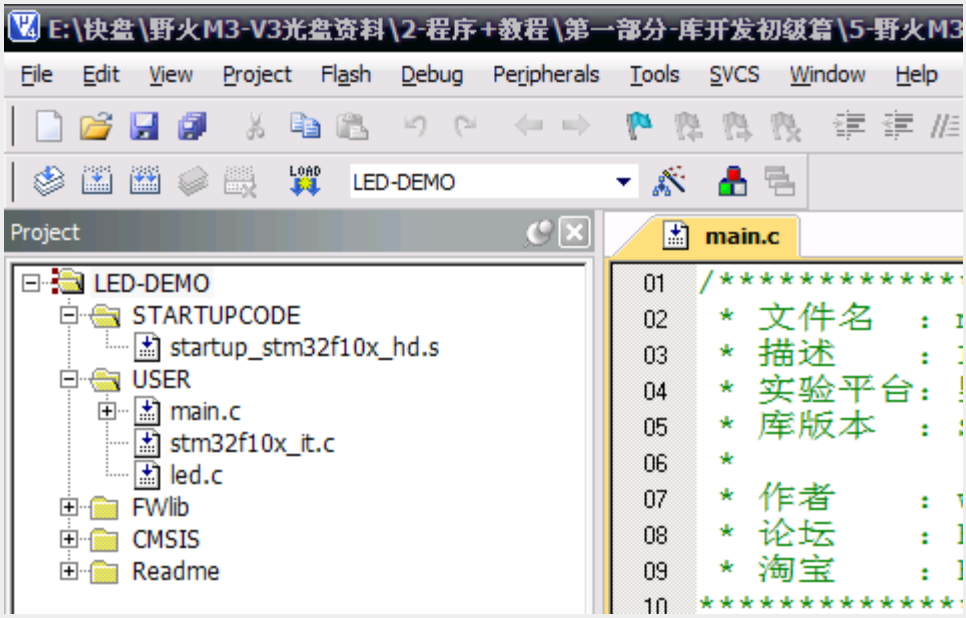
另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

1、如何编译和下载程序

在拿到开发板之后，很多朋友都想先尝尝鲜，想自己烧写个程序到开发板上，看看效果。下面将演示如何将光盘里面自带的程序烧写到野火 STM32 开发板上。前提是你的电脑上已经安装了 JLINK 驱动和 MDK 开发环境，如果这一部没有完成，请参考《2、**JLINK 驱动安装与 MDK 环境搭建**》。野火 STM32 开发板光盘上提供的代码都是已经编译好的，直接下载即可。

1.1 编译程序

首先打开一个 MDK 工程，在野火 STM32 开发板光盘目录下：*2-程序+教程|第一部分-库开发初级篇|5-野火 M3-流水灯|USER*，点击 STM32-DEMO.uvproj，打开流水灯这个工程。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。

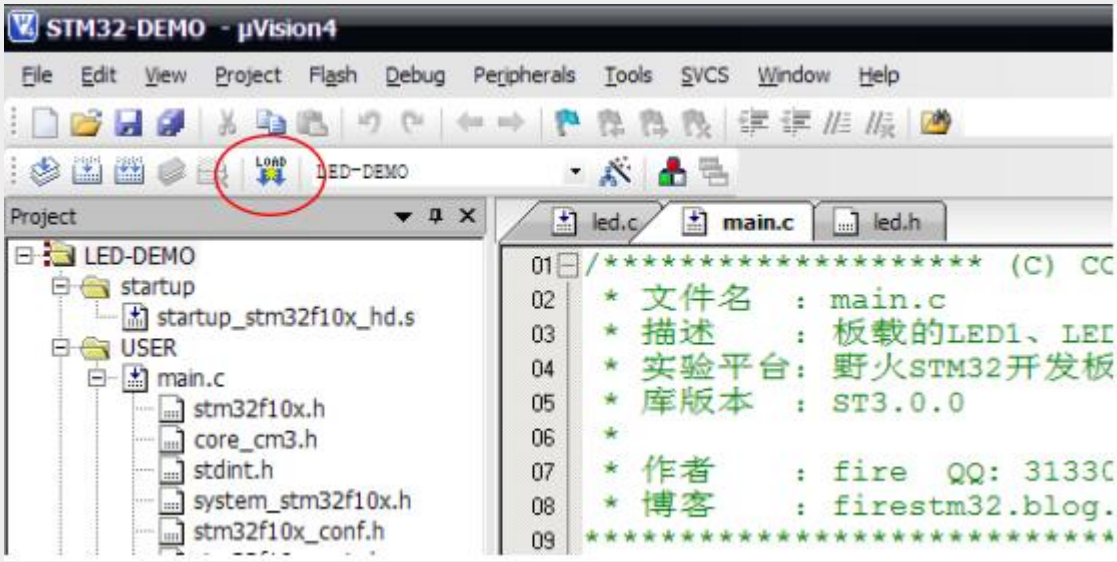
综上：当我们编辑好我们的程序之后，只需要用第二个 Build 按钮就可以，即方便又省时。第一个跟第三个按钮用的比较少。

1.2 下载程序

野火 STM32 开发板有两种下载方式，JLINKV8 下载和串口下载。要注意的是：1、JLINK 下载的时候，开发板中的拨动开关 BOOT0（在开发板边缘，靠近网口）即可以拨到 VCC 也可拨到 GND，但在 JLINK 下载完程序后，必须将 BOOT0 拨到 GND，好让程序从内部的存储器开始运行程序，所以在 JLINK 下载时最保险的方法就是将 BOOT0 拨到 GND 那端。2、在用串口下载程序的时候，必须将 BOOT0 开发拨到 VCC，在程序下载完后，然后将 BOOT0 开关拨到 GND。

1.2.1 JLINK 下载

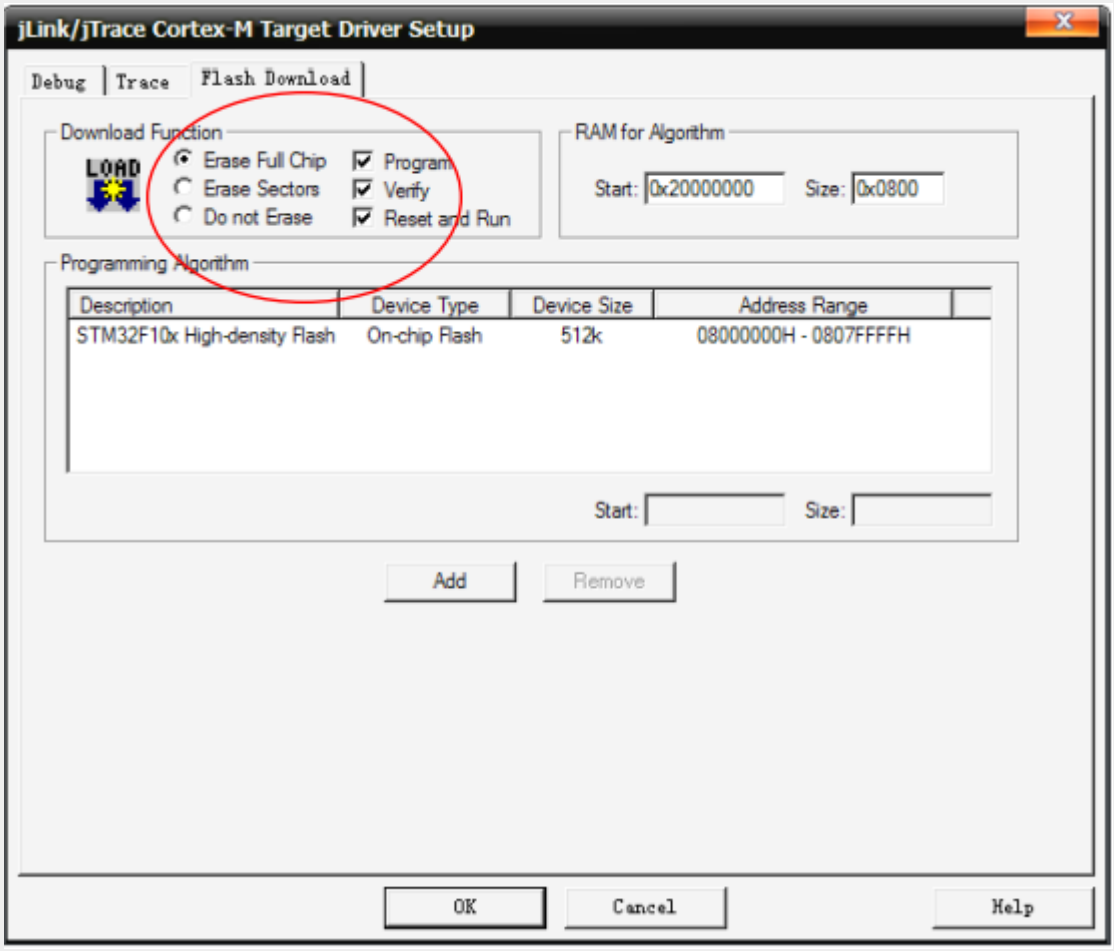
- 插上 DC-5V 电源给开发板供电，再插上 JLINK。
- 点击 MDK 工具栏中的 Load 按钮就可将编译好的程序下载到开发板中。



- 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。

```
Build Output
Programming Done.
Verify OK.
* JLink Info: TotalIRLen = 9, IRPrint = 0x0011
* JLink Info: Found Cortex-M3 r1p1, Little endian.
* JLink Info: TPIU fitted.
* JLink Info: ETM fitted.
* JLink Info:   FPUnit: 6 code (BP) slots and 2 literal slots
Application running ...
```

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->Falash DownLoad 中设置的：



如果没有设置为自动运行的话，我们需要在程序下载完毕之后进行手动复位，手动复位可以是按键复位和上电复位。

还有一点要注意的是：在程序下载到开发板之后，开发板要供电，JLINK 一端连开发板，另一端连 PC，这样程序才能运行。有些用户在下载程序之后，第二次用的时候只是给开发板供电，JLNK 的一端只连了开发板而没有连 PC，这样

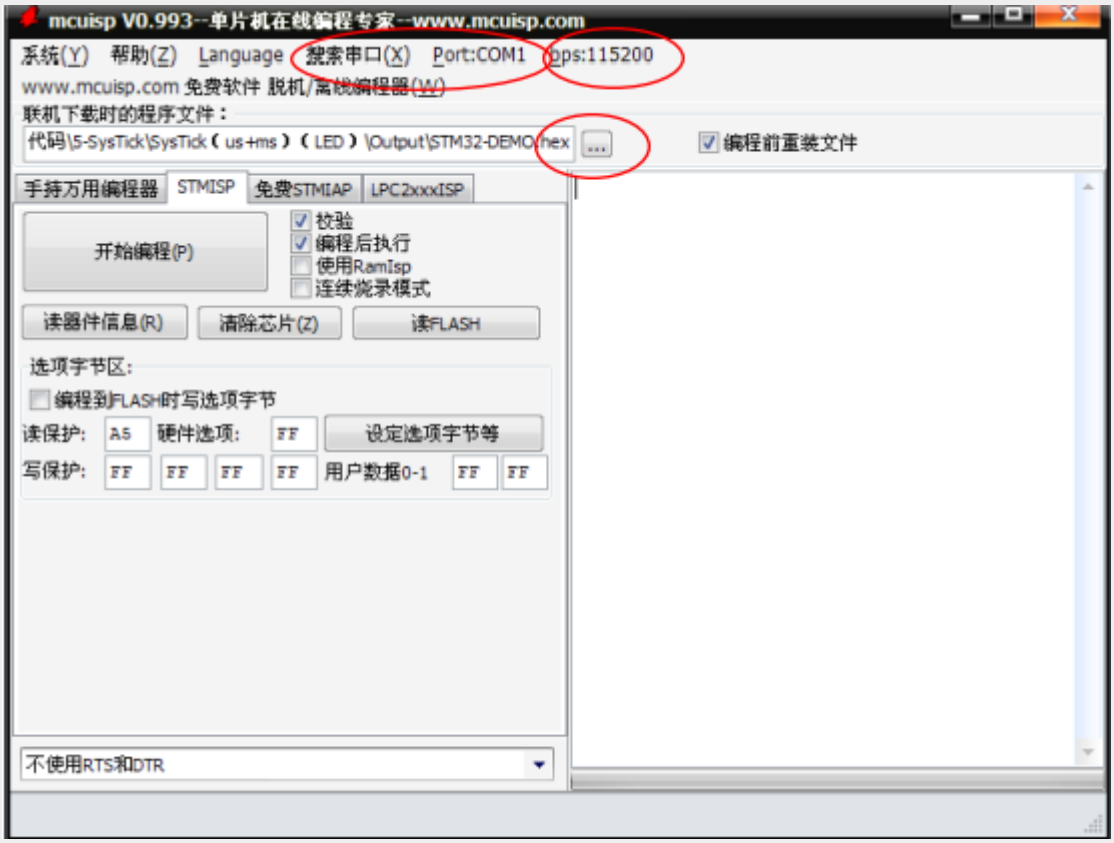
程序是不能工作的。要想只在供电的情况下要程序运行，只需把 JLNK 从开发板中拔掉即可，即只连电源，不接 JLINK 即可。

1.2.2 串口下载

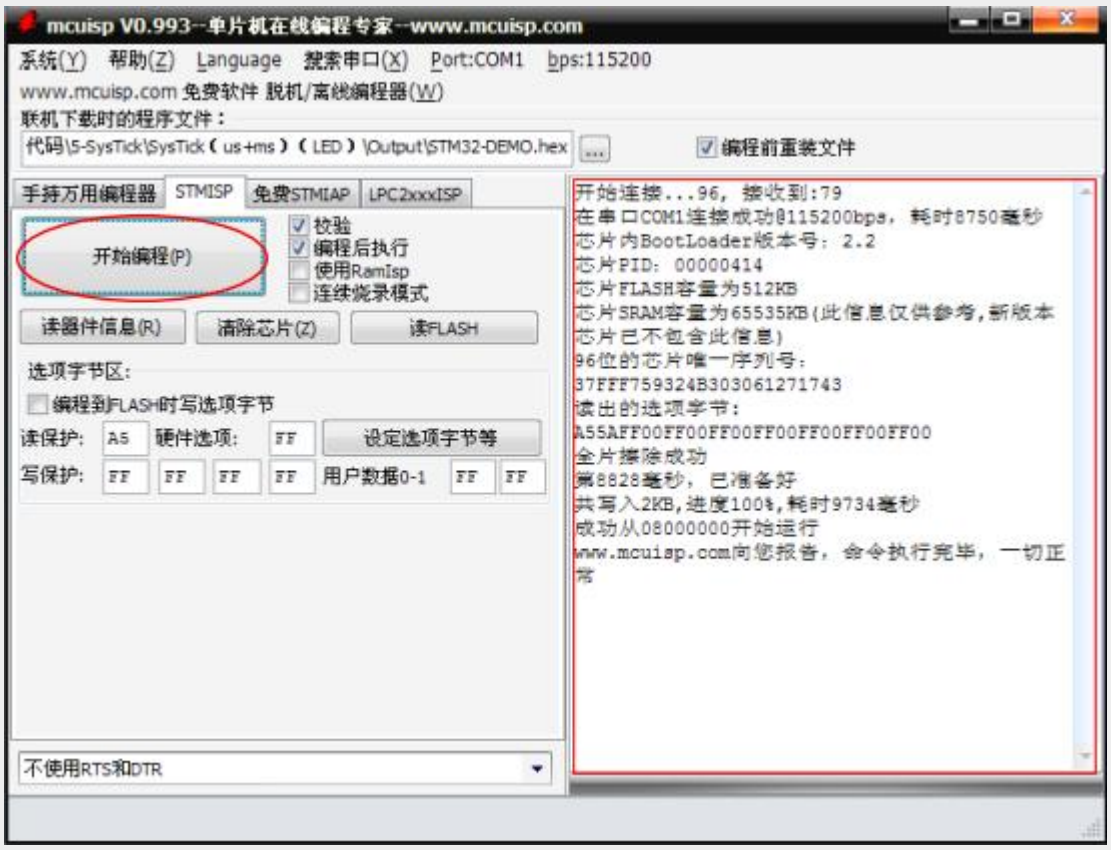
- 插上 DC-5V 电源给开发板供电，插上 JLINK，插上自带的串口线（**注意是两头都是母的交叉串口线）。**
- 将 BOOT0 开关拨到 VCC。

在这里我们用的串口下载软件是 mcuisp，这个一个绿色的软件，可从网上自由下载，野火 M3 光盘目录下：3-安装软件\3-串口下载软件找到。

- 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串口，它就会自动搜索串口，野火 STM32 开发板用的是电脑主板后面的串口，这个串口都会被默认为是串口 1。假如你是笔记本用户，用的是 USB 转串口，那么端口号可能就不是 COM1，需要到我的电脑\管理\设备管理器\端口中查找，然后再修改。
- 设置波特率为 115200，选择要下载的程序。在开发板自带的例程中，可执行文件(hex 文件)都在工程目录下的 Output 这个文件下。

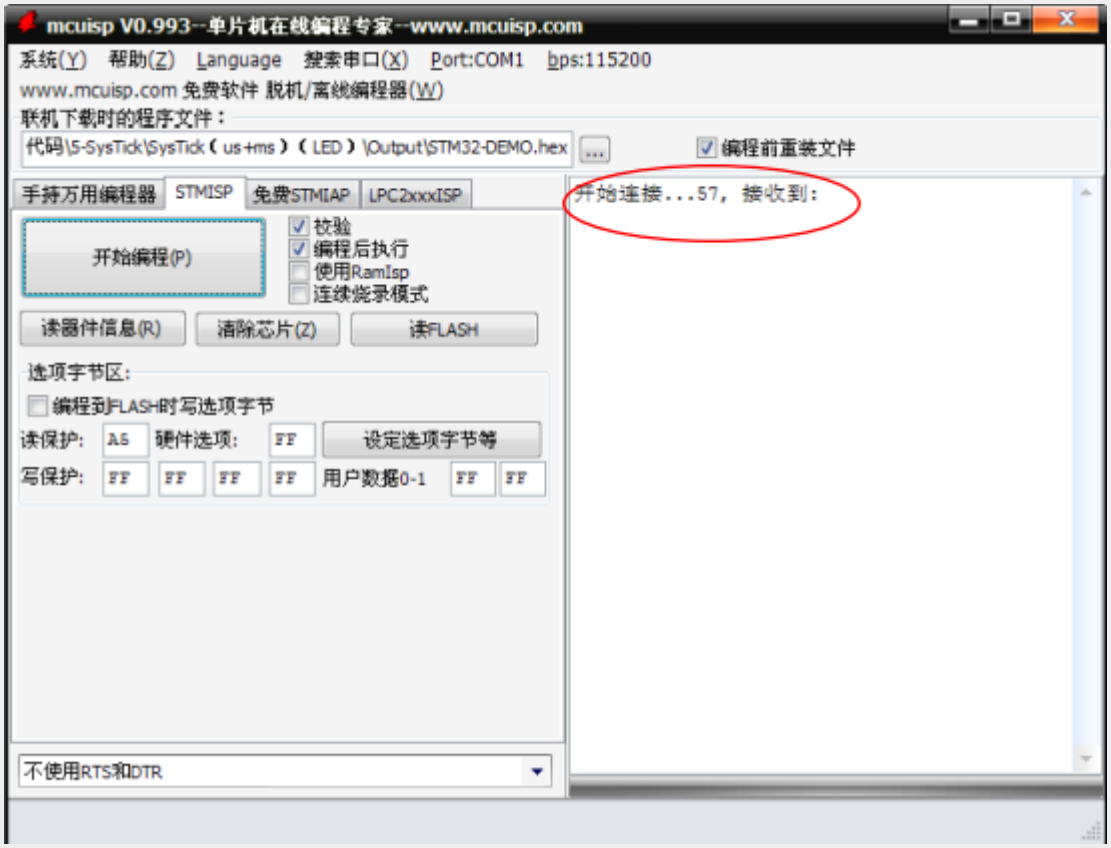

mcuisp V0.993--单片机在线编程专家--www.mcuisp.com
系统(Y) 帮助(Z) Language 搜索串口(X) Port:COM1 bps:115200
www.mcuisp.com 免费软件 脱机/离线编程器(W)
联机下载时的程序文件：
代码\5-SysTick\SysTick（us+ms）（LED）\Output\STM32-DEMO.hex
编程前重装文件
手持万用编程器 STMISP 免费STMIAP LPC2xxxISP
开始编程(P)
校验
编程后执行
使用RamIsp
连续烧录模式
读器件信息(R) 清除芯片(Z) 读FLASH
选项字节区：
编程到FLASH时写选项字节
读保护: A5 硬件选项: FF 设定选项字节等
写保护: FF FF FF FF 用户数据0-1 FF FF
不使用RTS和DTR

- 然后点击 开始编程 按钮，如果程序下载成功后则会打印出下面红色框中的信息。



- 程序下载成功之后，可是在开发板上看不到实验现象呀，怎么办？是不是出什么问题了呀？这是因为我们是通过串口将我们的程序烧写到 flash 里面去了，而我们想要从 flash 里面执行我们的程序的话，则需要将 BOOT0 开关拨到 GND，然后按下我们的复位按键就可以看到实验想象了。

- 在我们点击 开始编程按钮时，还会出现 mcuisp 一直处于连接的状态，导致程序下载不了，如下截图所示。解决的方法是只需我们按一下开发板中的复位按键即可。



1.2.3 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。如果你的板子用的的 MAX3232 是国产的毛片的话，则没有这个优点:(。国产的 MAX3232 价格是 0.3RMB，进口的是 3.8RMB。野火 STM32 开发板用的 MAX3232 是 3.8RMB 的，在波特率设置到 115200 时，仍可稳定下载:)。
  - 缺点：不能够在线调试。
- 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:)。

缺点：下载大型程序时速度缓慢，还不稳定，非常蛋疼。但要注意的是调试的时候是不会出现这种情况的。

- 所以，建议大家在购买开发板的时候，也买一个 JLINK。

2、JLINK 驱动安装与 MDK 环境搭建

2.1 JLINK 驱动安装

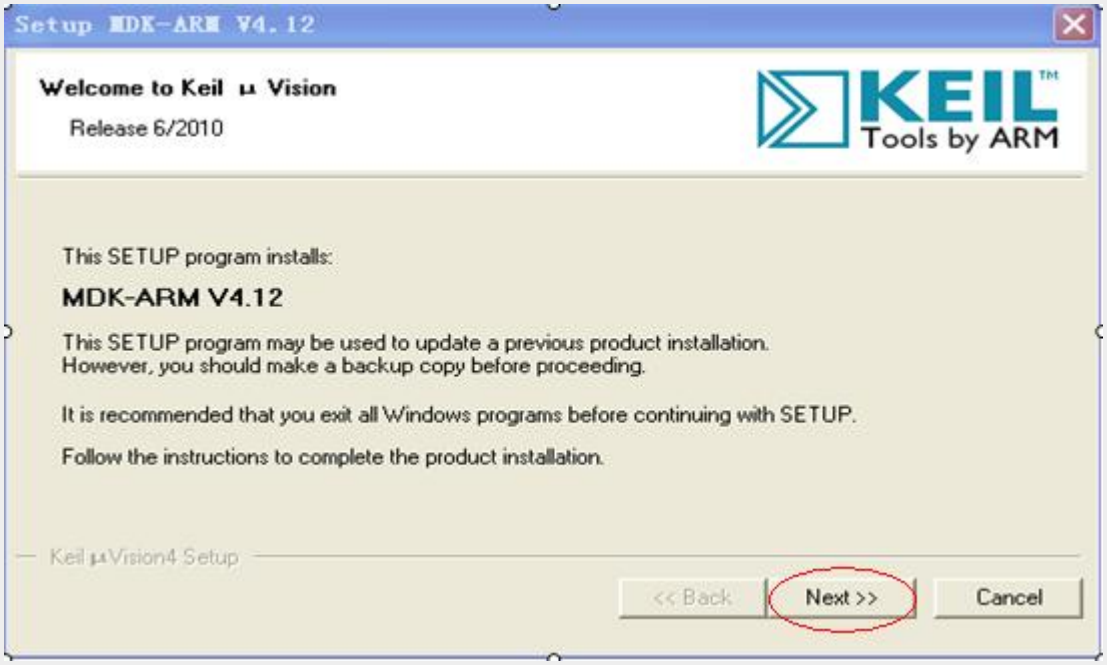
在用 JLINK 下载和调试程序之前，我们需要线在电脑上安装 JLINK 驱动，如果电脑上已经安装 JLINK 驱动，则可跳过这一步。在野火 M3 光盘目录下：3-安装软件\1-JLINKV8 驱动 点击 Setup_JLinkARM_V428c.exe ，完成 JLINK 驱动的安装。安装过程非常简单，这里将跳过。在安装完成后，我们将 JLINK 插接到电脑的 USB 口，即可在我的电脑\管理\设备管理器\通用串行总线控制器中看到一个 J-Link driver。要注意的是在安装完 JLINK 驱动后，一定要将 JLINK 插接到电脑的 USB 口，否则在电脑的设备管理器中是查看不到 J-Link driver 的。当你把 JLINK 拔出电脑的 USB 口时候，J-Link driver 就会消失。

2.2 MDK 环境搭建

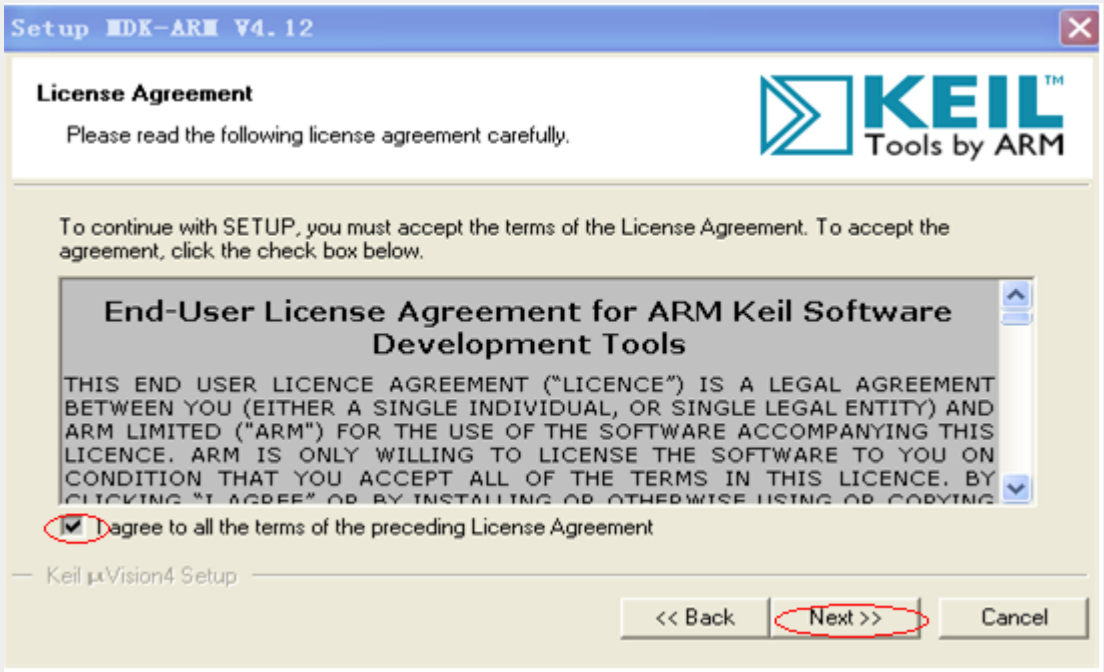
在我们学习编写代码之前需要先要把 MDK 这个软件安装好，野火用的版本是 V4.14，在安装完成之后可以在工具栏 help->about uVision 选项卡中查看到版本信息。MDK 是一个集代码编辑，编译，链接和下载于一体的集成开发环境（KDE）。MDK 这个名字我们可能不熟悉，但说到 KEIL，学过 51 的朋友就再熟悉不过了。后来 KEIL 被 ARM 公司收购之后就改名为 MDK 了，所以学过 51 的朋友是很快就可以熟悉这个开发环境的。

在野火 M3 光盘目录下：3-安装软件\2-MDK 找到 MDK414.exe，点击 MDK414.exe，在弹出 MDK 安装界面后，按照如下步骤操作即可。

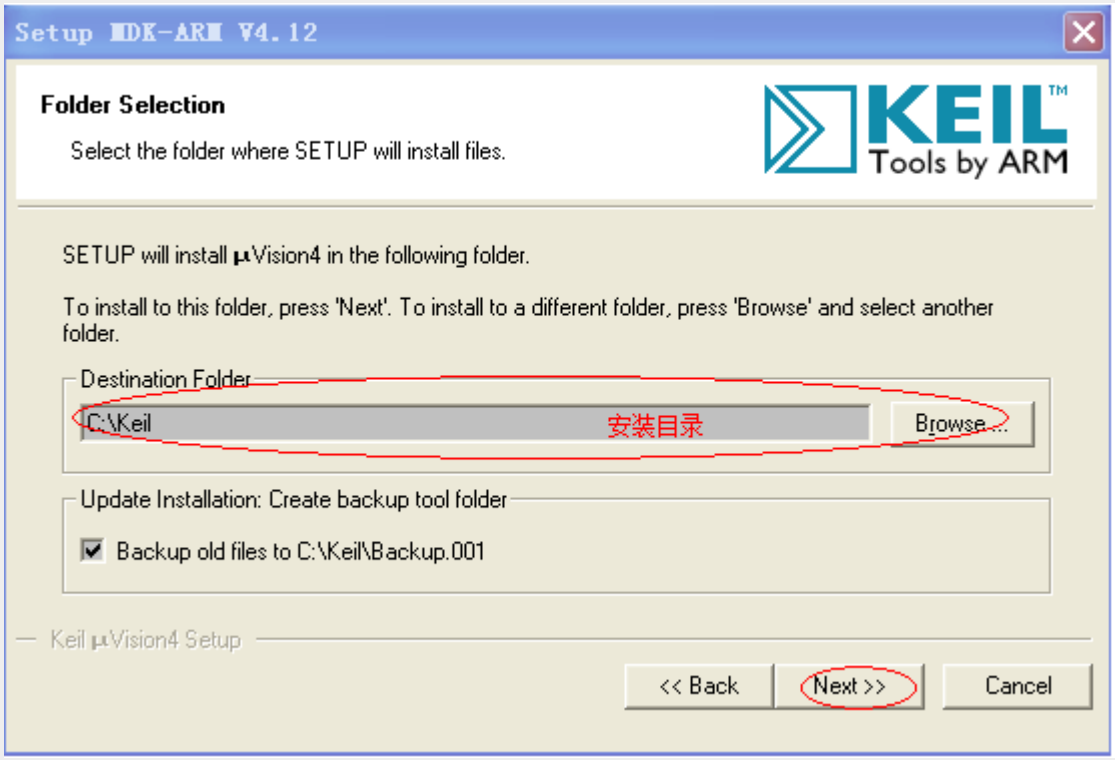
- 点击 Next。



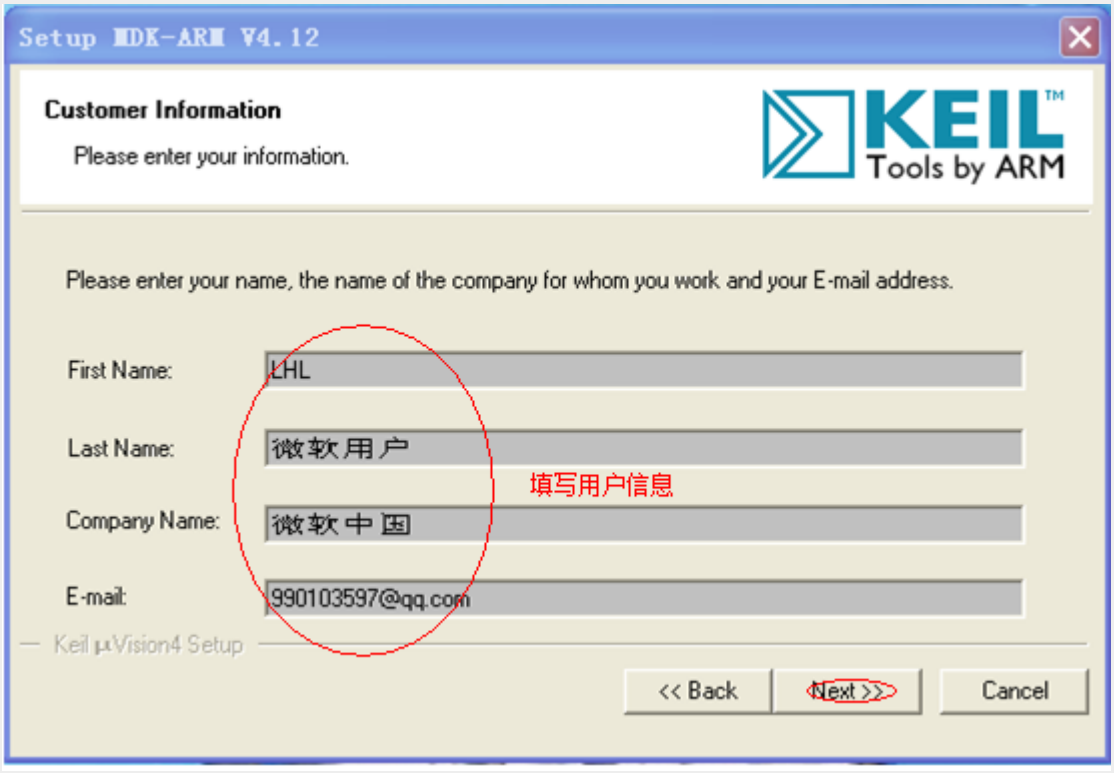
- 把勾勾上，点击 Next。



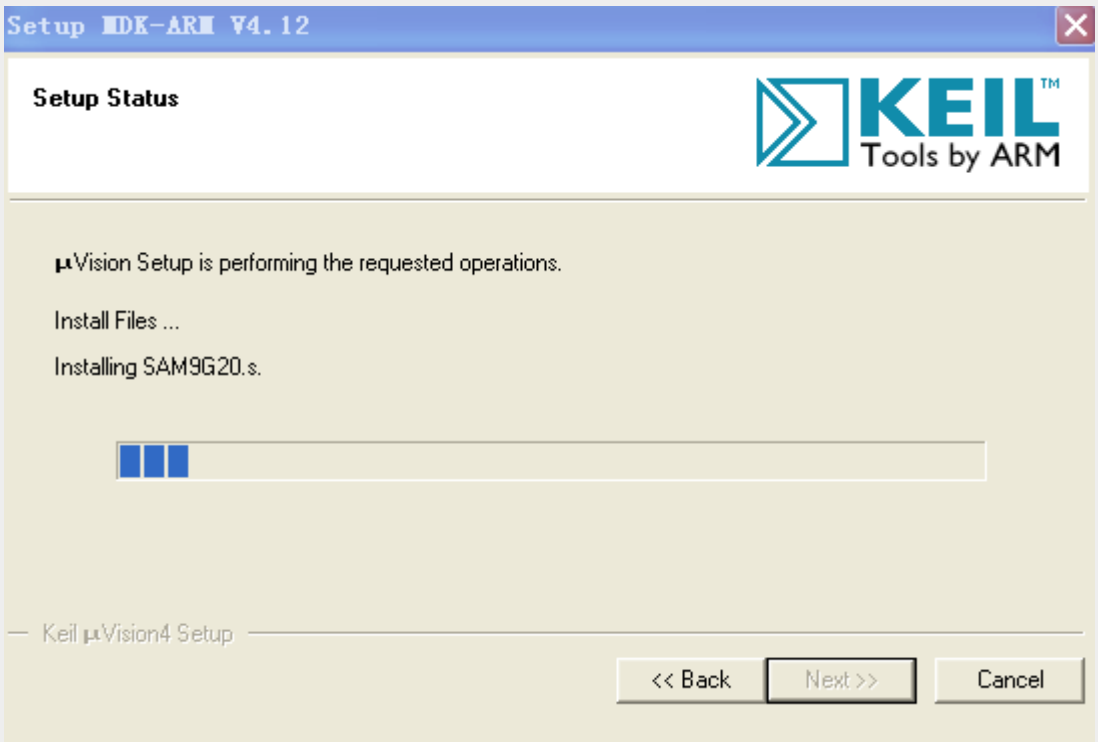
- 点击 Next，默认安装在 C:\keil 目录下。



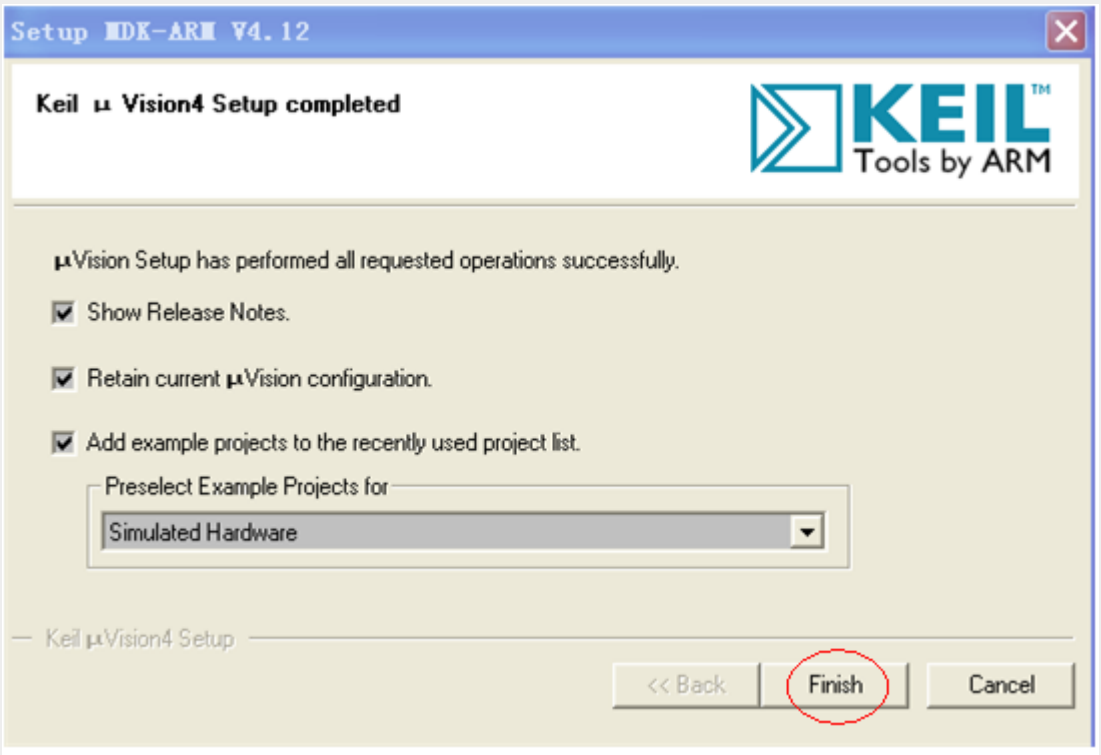
- 在用户名中填入名字（可随便写，可空格），在邮件地址那里填入邮件地址（可随便写，可空格），点击 Next。



- 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，如下所示：

2.3 和谐 MDK

安装完 MDK 开发环境后，在下载程序的时候会有 40K 的代码限制，我只需要和谐下即可搞定:)。在野火 M3 光盘目录下：3-安装软件\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（MDK 的 CID 在 MDK 开发环境中的菜单栏 File\License Managemant 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,大功告成:)。

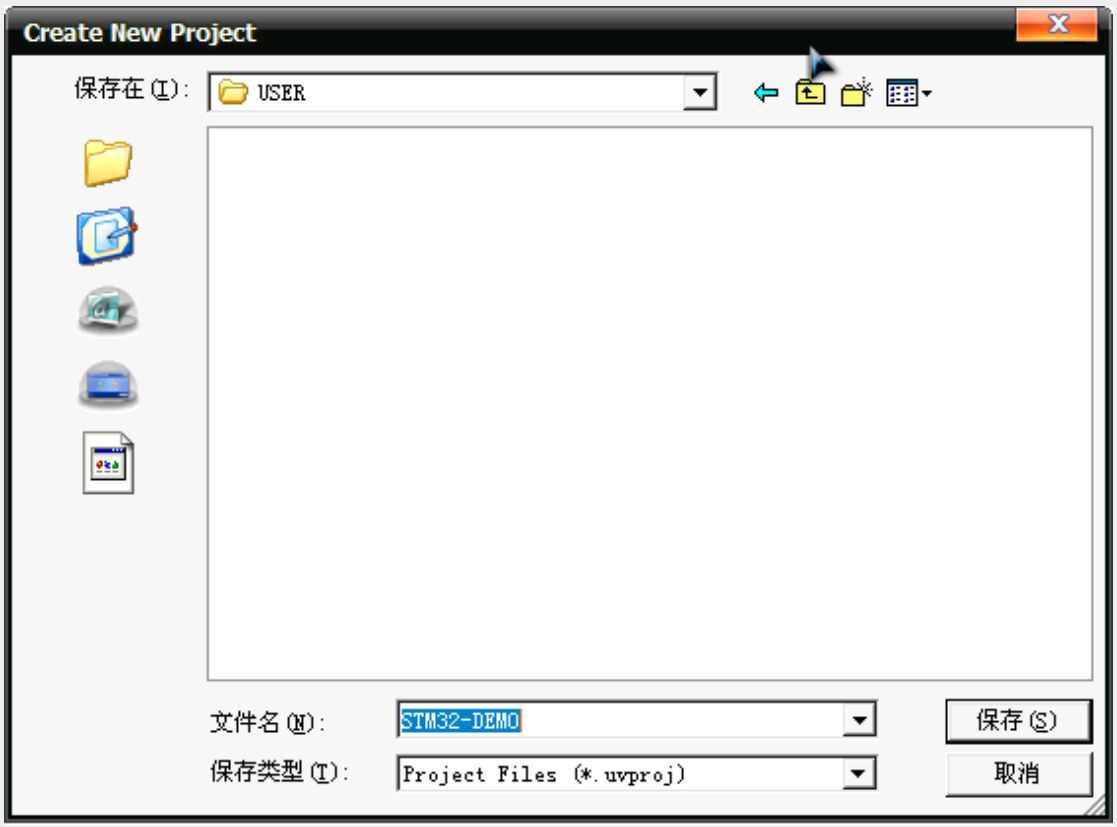
3、如何新建工程模板

3.1 获取 ST 库源码

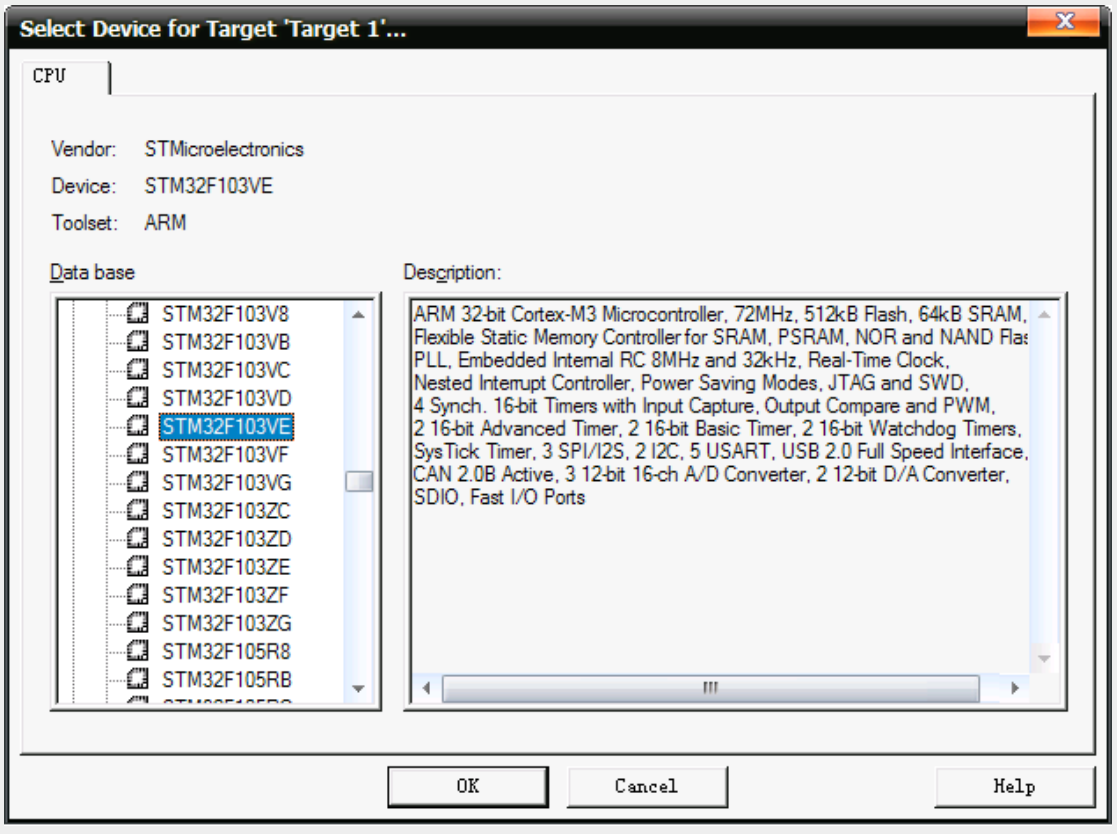
在新建工程模板之前，我们首先需要获取到 st 库的源码，源码可从 st 的官方网站下载到，也可在野火 M3 光盘目录下：\2-程序+教程\第一部分-库开发初级篇 找到，里面有 V3.0.0 和 V3.5.0 版本的库，这两个库的版本区别很小，几乎可以兼容。在这里我们以 V3.5.0 来新建我们的工程模板。

3.2 开始新建工程

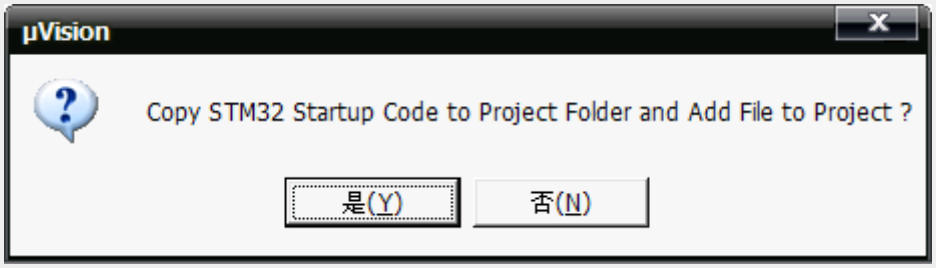
- 点击桌面 UVision4 图标，启动软件。如果是第一次使用的话会打开一个自带的工程文件，我们可以通过工具栏 Project->Close Project 选项把它关掉。
- 在工具栏 Project->New μVision Project...新建我们的工程文件，我们将新建的工程文件保存在桌面的 STM32-Template\USER 文件夹下（先在电脑桌面上新建一个 STM32-Template 文件夹，在 STM32-Template 里面新建一个 USER 文件夹），文件名取为 STM32-DEMO（英文 DEMO 的意思是例子），名字可以随便取，点击保存。



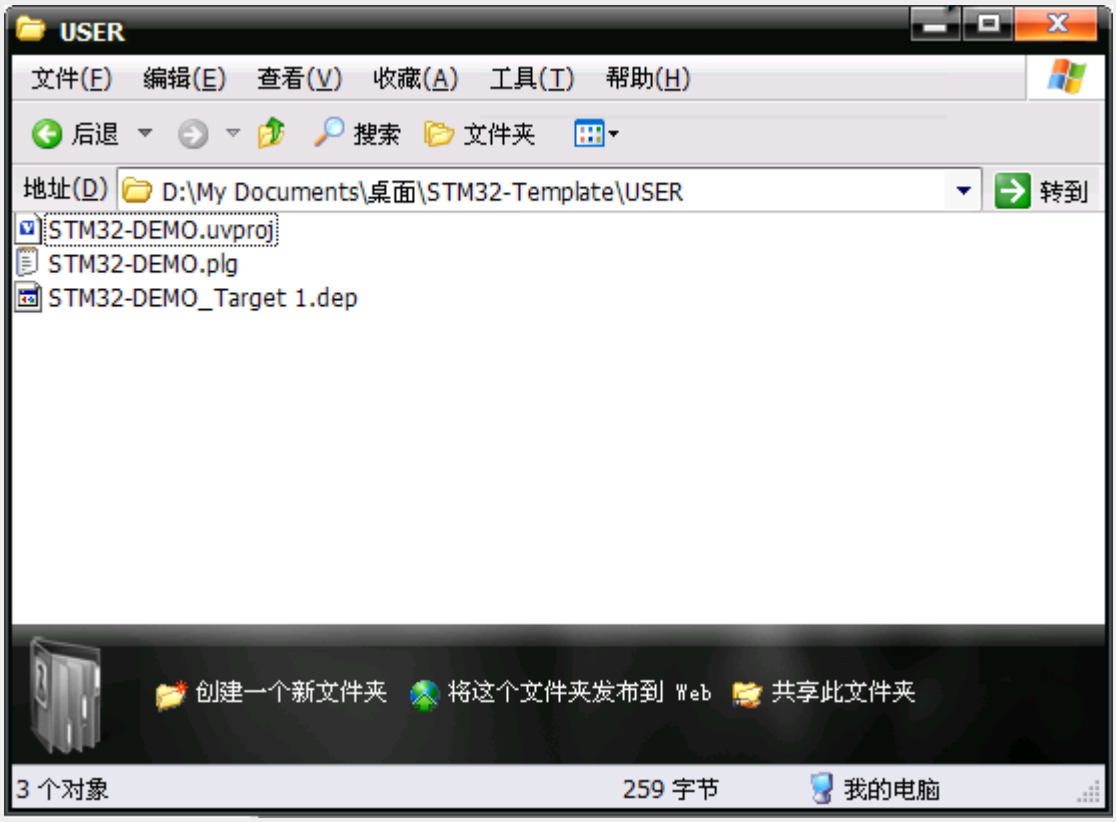
- 接下来的窗口是让我们选择公司跟芯片的型号，我们用的芯片是 ST 公司的 STM32F103VET6，有 64K SRAM,512K Flash，属于高集成度的芯片。按如下选择即可。



- 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。

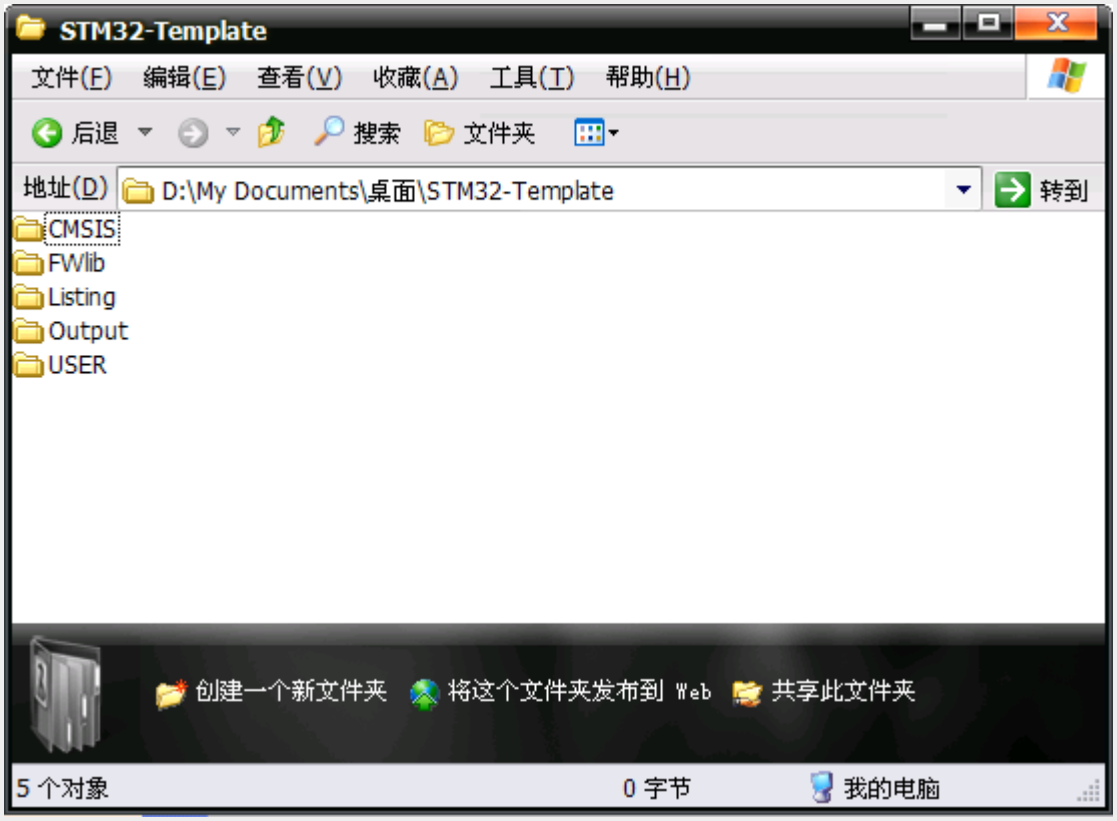


- 此时我们的工程新建成功，如下图所示。但我们的工程中还没有任何文件，接下来我们需要在我们的工程中添加所需文件。



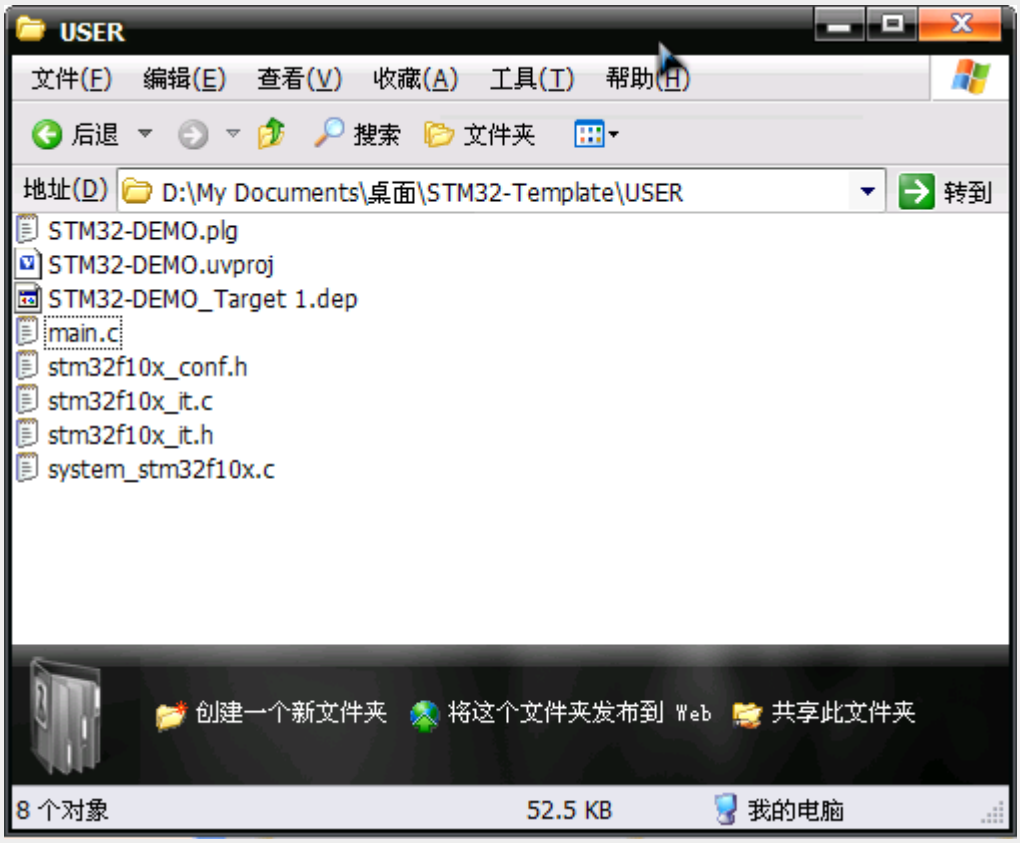
- 在 STM32-Template 文件夹下，我们新建四个文件夹，分别为 FWlib、CMSIS、Uotput、Listing。原先新建的 USER 用来存放工程文件和用户代码，包括主函数 main.c。 FWlib 用来存放 STM32 库里面的 inc 和 src 这两个文件，这两个文件包含了芯片上的所有驱动。CMSIS 用来存放库为我们自带的启动文件和一些 M3

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。Uotput 用来保存软件编译后输出的文件，Listing 用来存放一些编译过程中产生的文件，具体可不用了解。



- 把野火 M3 光盘目录下：\3-ST 库 3.5.0 源码\3.5.0\3.5.0\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\STM32F10x_StdPeriph_Driver 的 inc 跟 src 这两个文件夹拷贝到 STM32-Template\FWlib 文件夹中。

- 把野火 M3 光盘目录下：\2-程序+教程\第一部分-库开发初级篇\3-ST 库 3.5.0 源码 \3.5.0\3.5.0\STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Template 下的 main.c、stm32f10x_conf.h、stm32f10x_it.h、 stm32f10x_it.c 、system_stm32f10x.c 拷贝到 STM32-Template\USER 目录下。 stm32f10x_it.h、和 stm32f10x_it.c 这两个文件里面是中断函数，里面为空，并没有写任何的中断服务程序。stm32f10x_conf.h 是用户需要配置的头文件，当我们需要用到芯片中的某部分外设的驱动时，我们只需要在该文件下将该驱动的头文件包含进来即可，片上外设的驱动在 src 文件夹中，inc 文件夹里面是它们的头文件。这三个文件是用户在编程时需要修改的文件，其他库文件一般不需要修改。system_stm32f10x.c 是 ARM 公司提供的符合 CMSIS 标准的库文件，等下我们把这个文件移动到 STM32-Template\CMSIS 这个文件夹中。



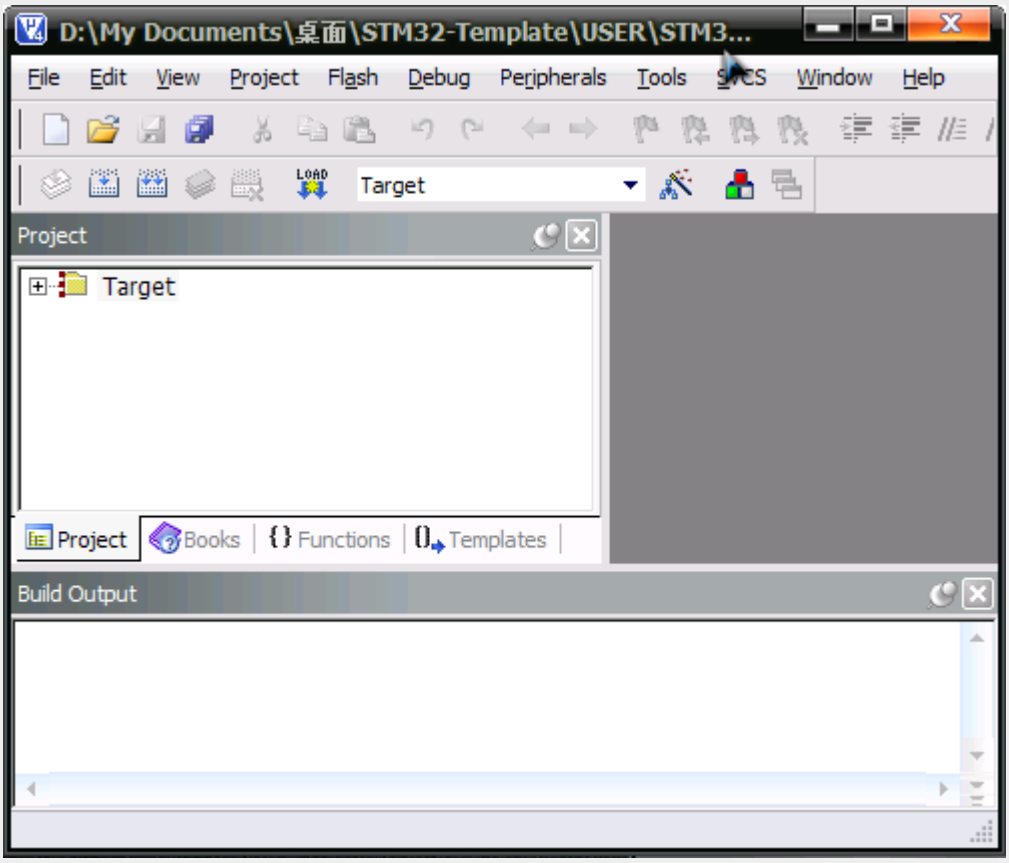
- （1）把野火 M3 光盘目录下：\2-程序+教程\第一部分-库开发初级篇\3-ST 库 3.5.0 源码 \3.5.0\3.5.0\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm 的全部文件拷贝到 STM32-Template\CMSIS\startup（需先在 CMSIS 新建好 startup 文件夹）文件夹下。这些是用汇编写的启动文件。野火 M3 开发板用的 CPU 是 STM32F103VET6，有 512K Flash，属于大容量的，所以等下我们把 startup_stm32f10x_hd.s 添加到我们的工程中。根据 ST 的官方资料：Flash 在 16 ~32 Kbytes 为小容量， 64 ~128 Kbytes 为中容量，256 ~512 Kbytes 为大容量，不同大小的 Flash 对应的启动文件不一样，这点要注意。（2）把野火 M3 光盘目录下：\2-程序+教程\第一部分-库开发初级篇\3-ST 库 3.5.0 源码 \3.5.0\3.5.0\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\CoreSupport 的 core_cm3.c 和 core_cm3.h 也拷贝到 STM32-Template\CMSIS 文件夹下。

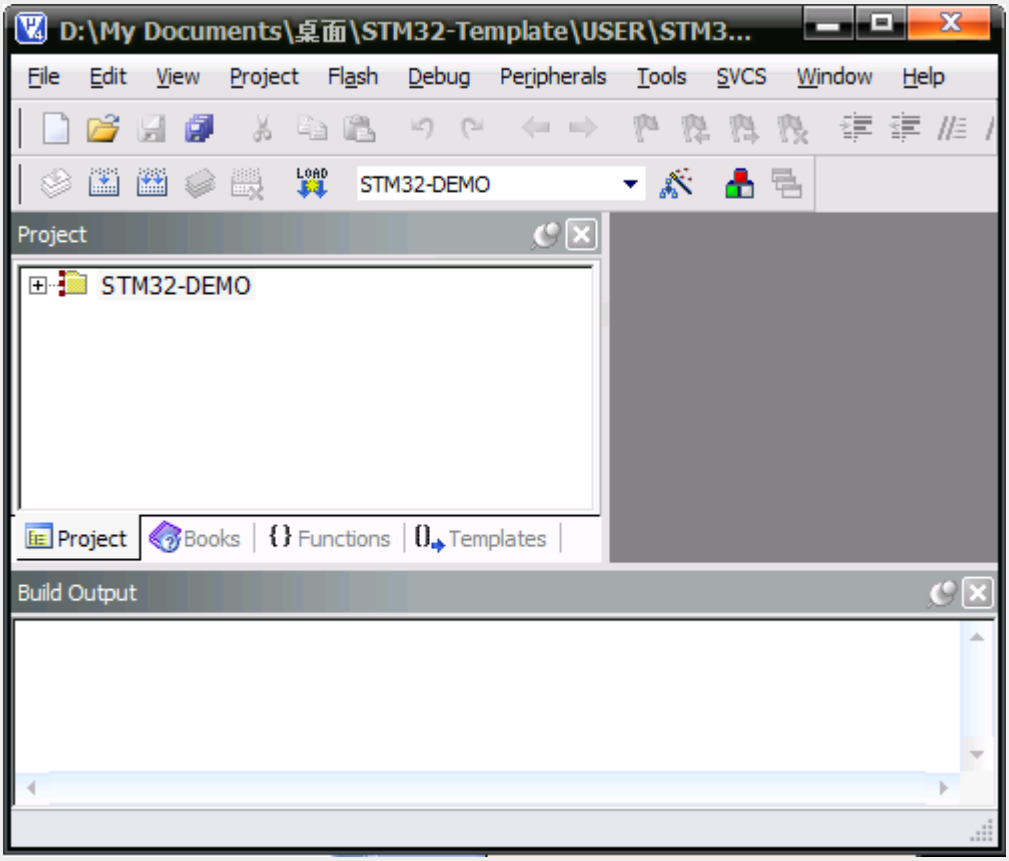
  （3）把野火 M3 光盘目录下：\2-程序+教程\第一部分-库开发初级篇\3-ST 库 3.5.0 源码

\3.5.0\3.5.0\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x *的* stm32f10x.h、system_stm32f10x.c、system_stm32f10x.h *拷贝到* STM32-Template\CMSIS 文件夹下。

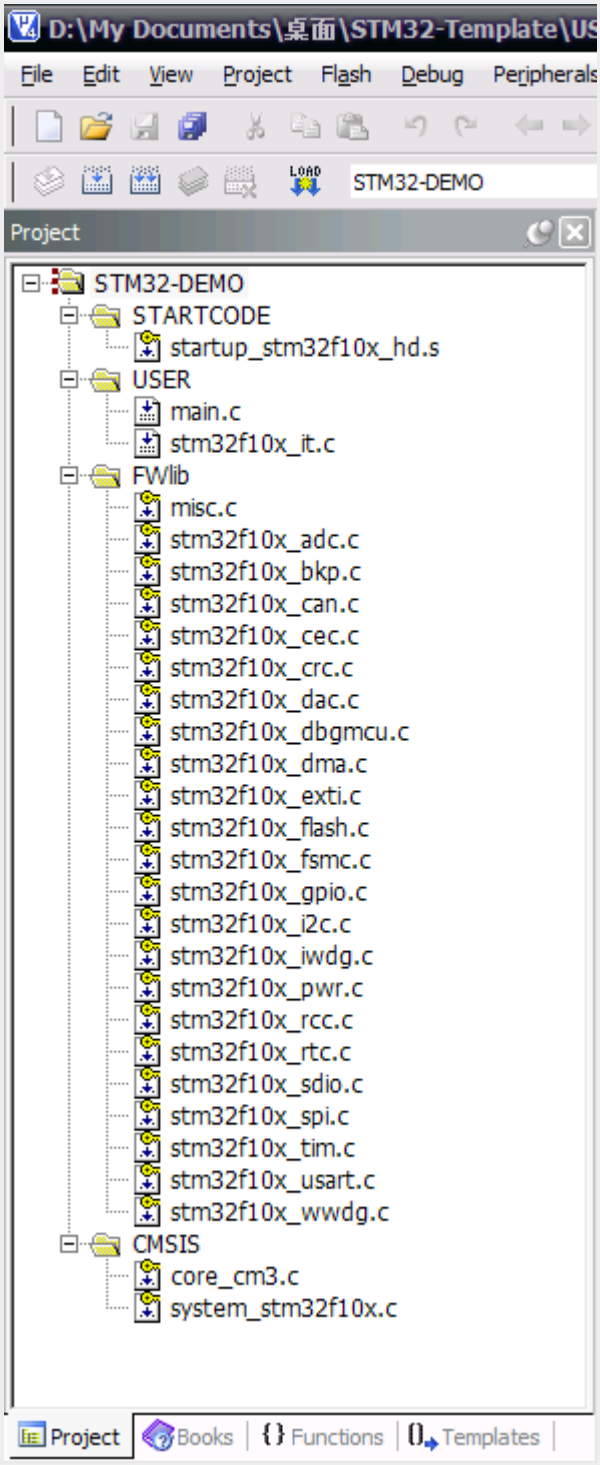
- 此时我们新进的工程目录如下所示

- 回到我们刚刚新建的 MDK 工程中，将 Target 改为 STM32-DEMO（不改也行）



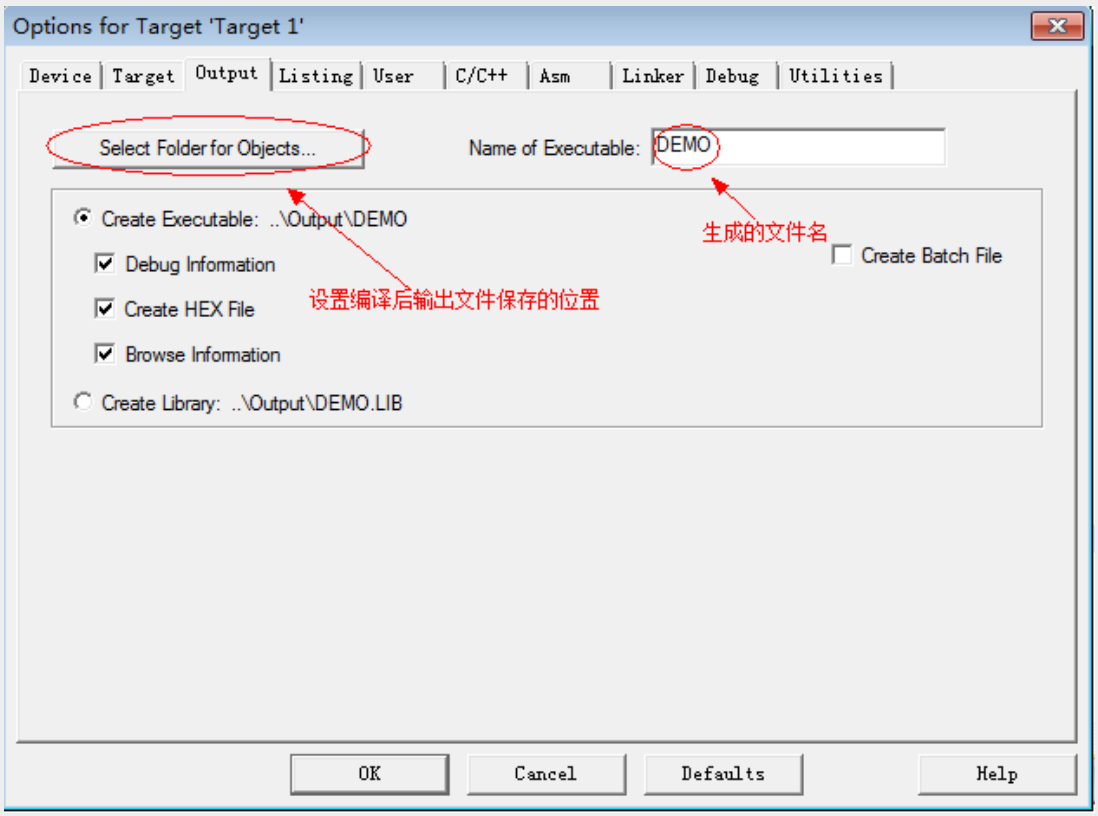


- 在 STM32-DEMO 上右键选中 Add Group...选项，新建四个组，分别命名为 STARTCODE、USER、FWlib、CMSIS。STARTCODE 从名字就可以看得出我们是用它来放我们的启动代码的，USER 用来存放用户自定义的应用程序，FWlib 用来存放库文件，CMSIS 用来存放 M3 系列单片机通用的文件。
- 接下来我们往我们这些新建的组中添加文件，双击哪个组就可以往哪个组里面添加文件。我们在 STARTCOKE 里面添加 startup_stm32f10x_hd.s，在 USER 组里面添加 main.c 和 stm32f10x_it.c 这两个文件，在 FWlib 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。在 CMSIS 里面添加 core_cm3.c 和 system_stm32f10x.c 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。最终效果如下图：

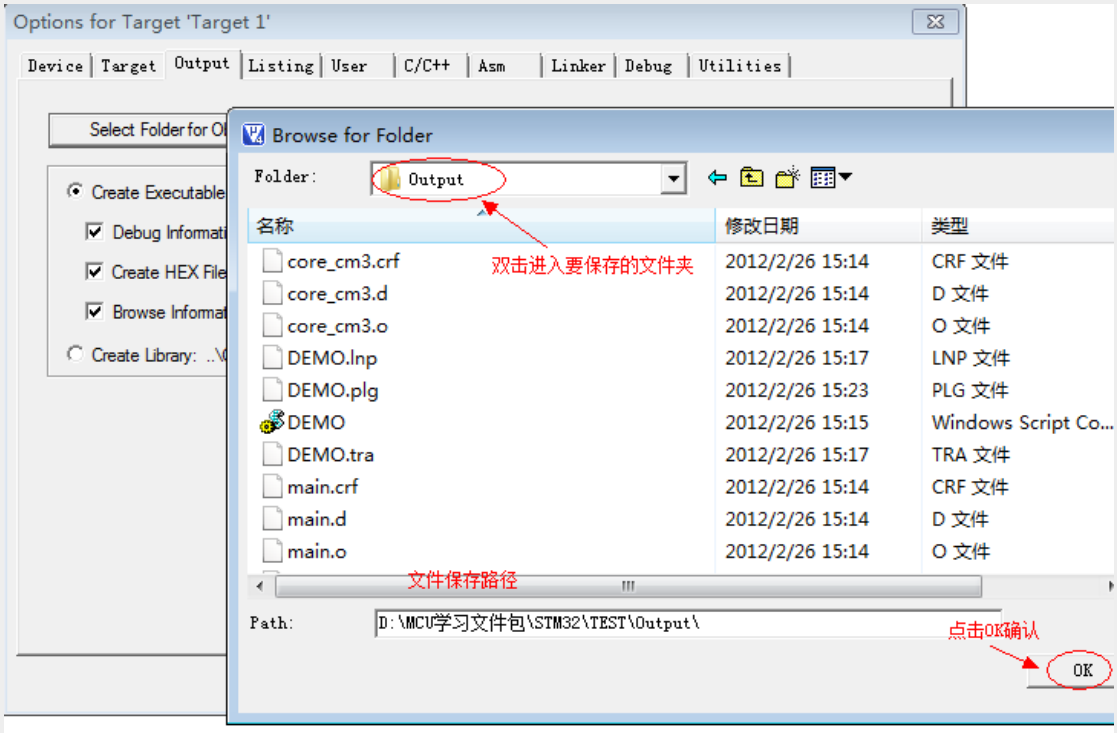


至于有些文件有个锁的图标，是因为这些都是库文件，不需要我们修改，属性为只读。

- 至此，我们的工程已经基本建好，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 ，在弹出来的窗口中选中 Output

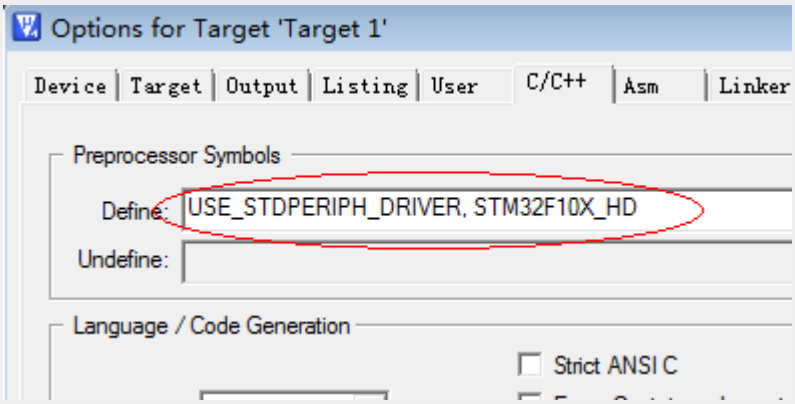


点击 Select Folder for Objects... 设置编译后输出文件保存的位置。同时把 Create HEX File 和 Browse information 这两个选项框也选上。



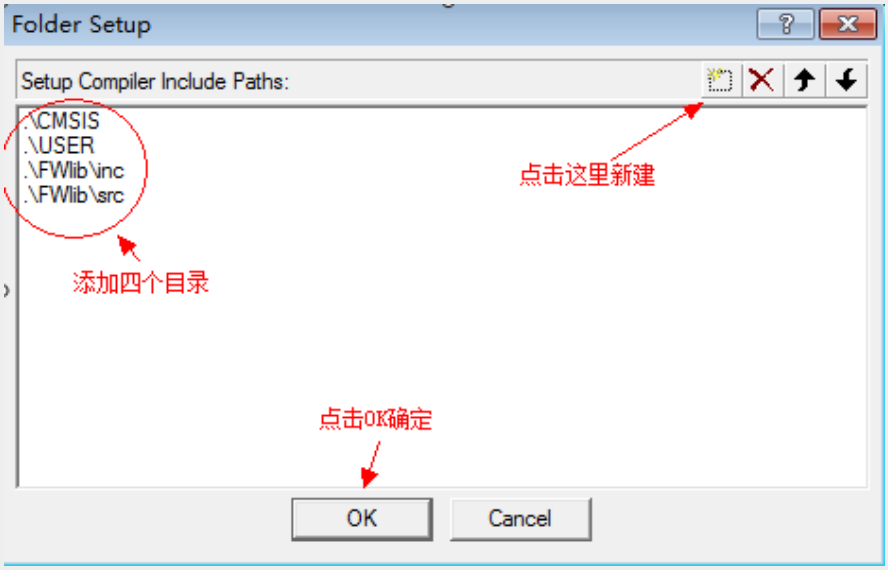
同样在 Listing 这个选项卡中，我们也点击 Select Folder listings...定位到模板中的 Listing 文件夹。

- 选中 C/C++ 选项卡，在 Define 里面输入添加 USE_STDPERIPH_DRIVER, STM32F10X_HD。



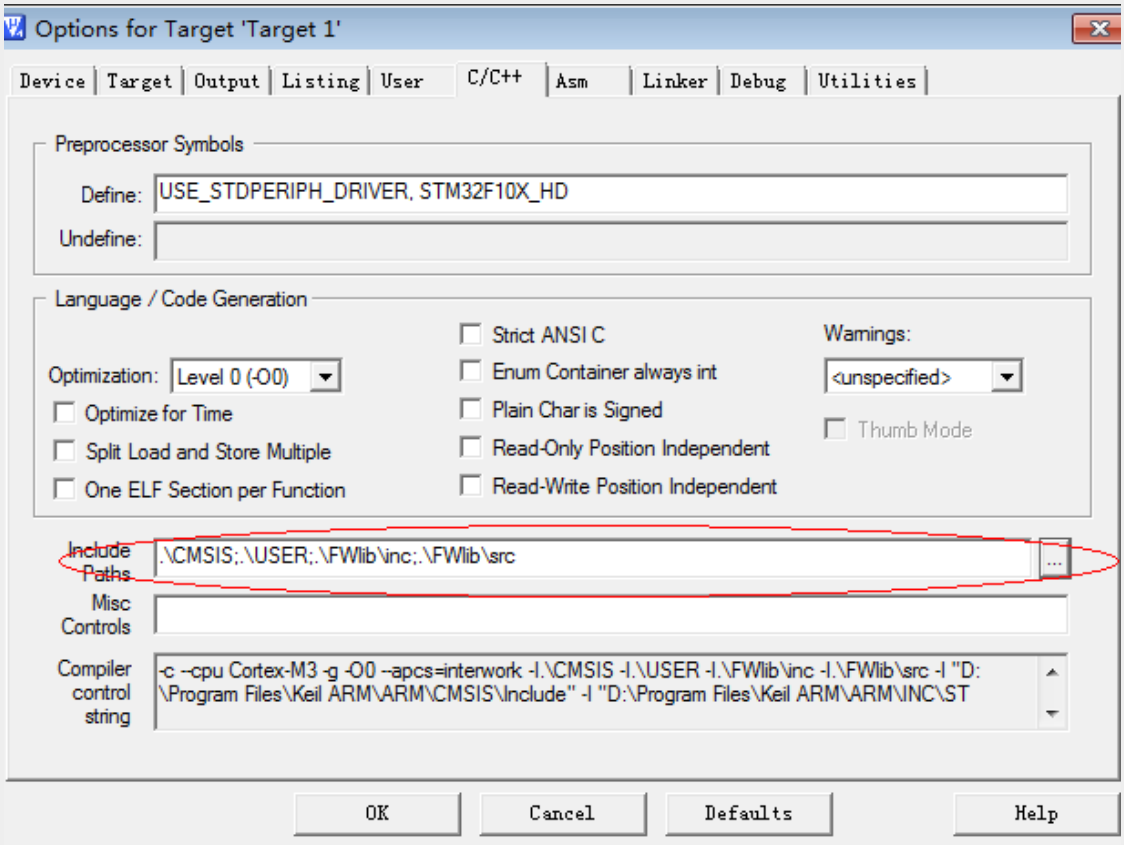
添加 USE_STDPERIPH_DRIVER 是为了**屏蔽编译器的默认搜索路径，转而**使用我们添加到工程中的 ST 的库，添加 STM32F10X_HD 是因为我们用的芯片是大容量的，添加了 STM32F10X_HD 这个宏之后，库文件里面为大容量定义的寄存器我们就可以用了。芯片是小或中容量的时候宏要换成 STM32F10X_LD 或者 STM32F10X_MD。其实不管是什么容量的，我们只要添加上 STM32F10X_HD 这个宏即可，当你用小或者中容量的芯片时，那些为大容量定义的寄存器我不去访问就是了，反正也访问不了。

在 Include Paths 栏点击  ，在这里添加库文件的搜索路径，这样就可以屏蔽掉默认的搜索路径。



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 ANSIC 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：



- 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，里面有许多的东西我们是不需要的，为了简化 main.c 文件，我们将修改如下。

```
/******************** (C) COPYRIGHT 2012 WildFire Team **************************
 * 文件名  ：main.c
 * 描述    ：用 3.5.0 版本建的工程模板。
 * 实验平台：野火 STM32 开发板
 * 库版本  ：ST3.5.0
 *
 * 作者    ：wildfire team
 * 论坛    ：http://www.amobbs.com/forum-1008-1.html
 * 淘宝    ：http://firestm32.taobao.com
*******************************************************************************/
#include "stm32f10x.h"



/*
 * 函数名：main
```

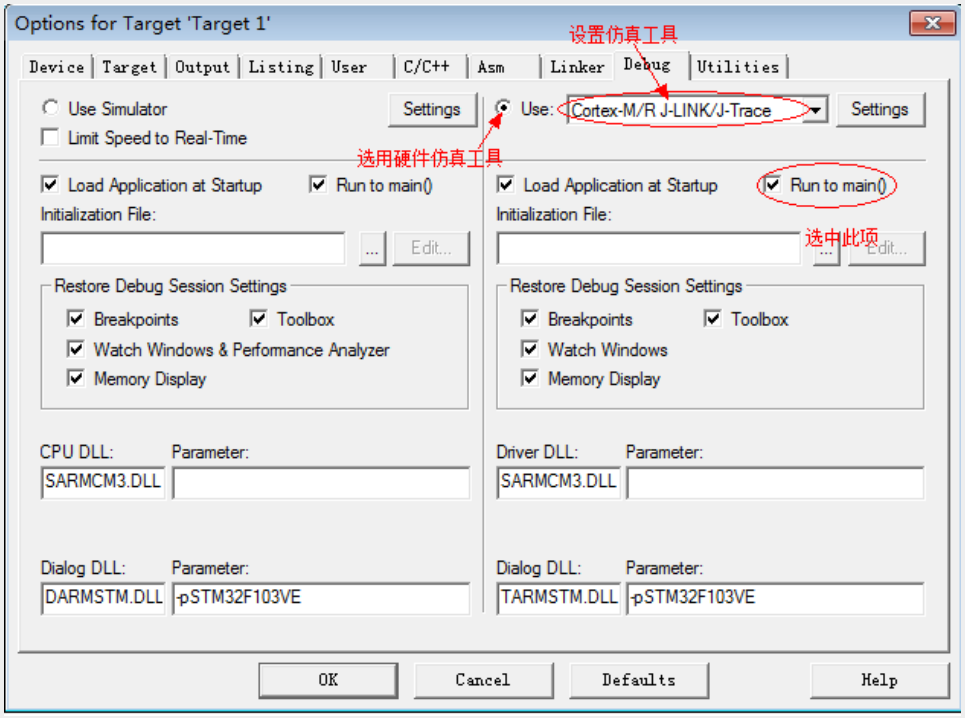
```
 * 描述    : 主函数
 * 输入    : 无
 * 输出    : 无
 */
int main(void)
{
      while(1);
      // add your code here ^_^。
}
/****************** (C) COPYRIGHT 2012 WildFire Team *****END OF FILE************/
```

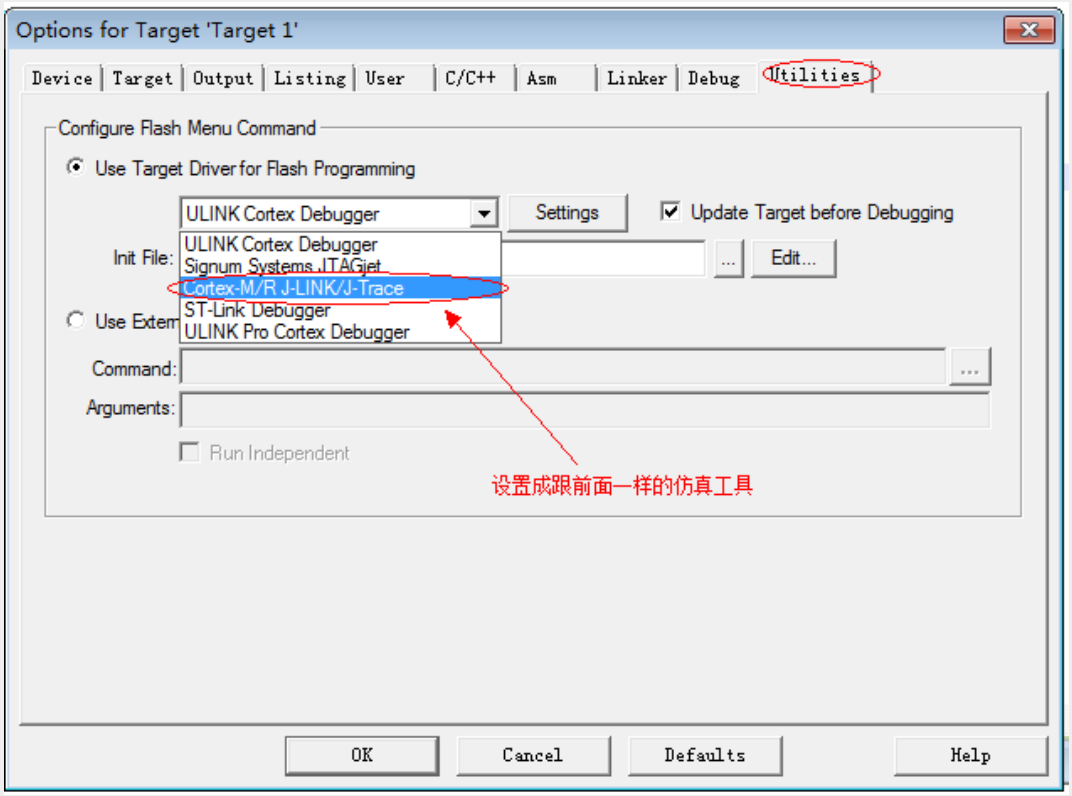
- 至此，我们的工程模板就建成了。学会新建工程，是学习 stm32 的第一步。

3.3 硬件调试配置

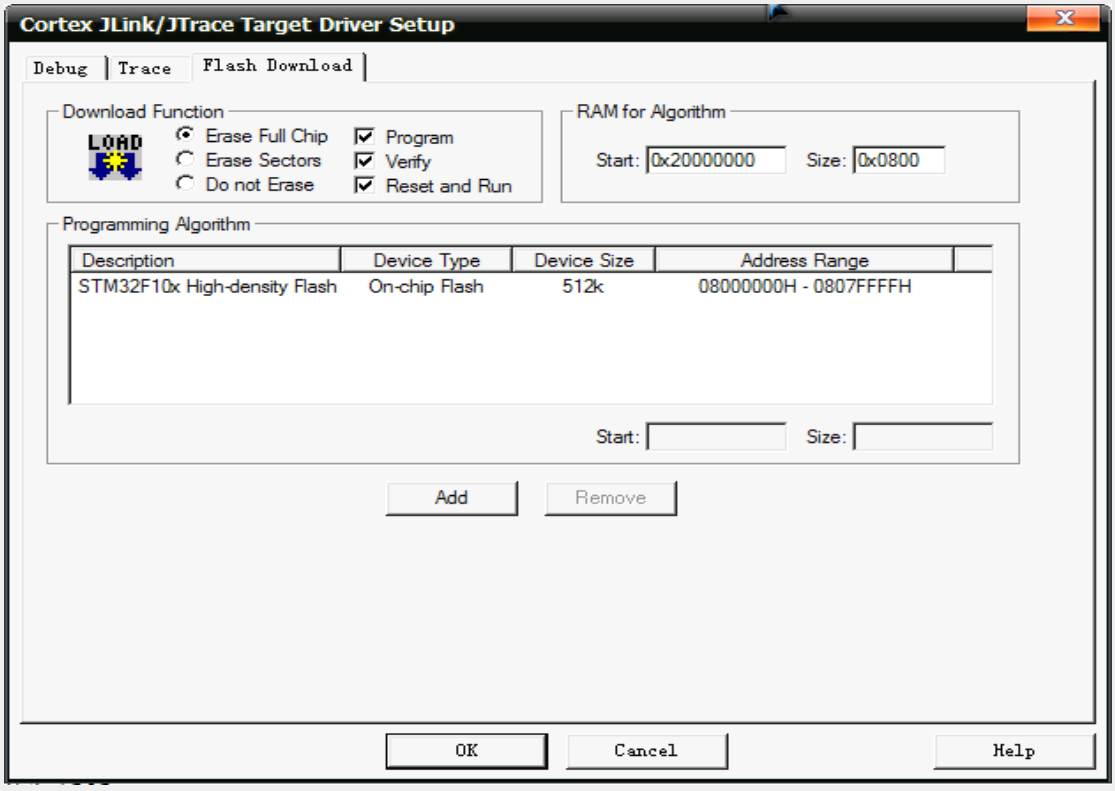
这个工程默认的是软件仿真，如果开发板要用 J-LINK 调试的话，还需要在开发环境中做如下修改。实际上，我们开发程序的时候 80%都是在硬件上调试的。

具体配置如下图所示：点击  ，在 Debug 选项里





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 M3 光盘目录下提供的已经新建好的工程模板。

4、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。

4.1 STM32 神器之库开发

4.1.1 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 prinft()函数时只是学习它的使用格式，并没有去研究它的源码实现，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见**错误！未找到引用源。**4-1。

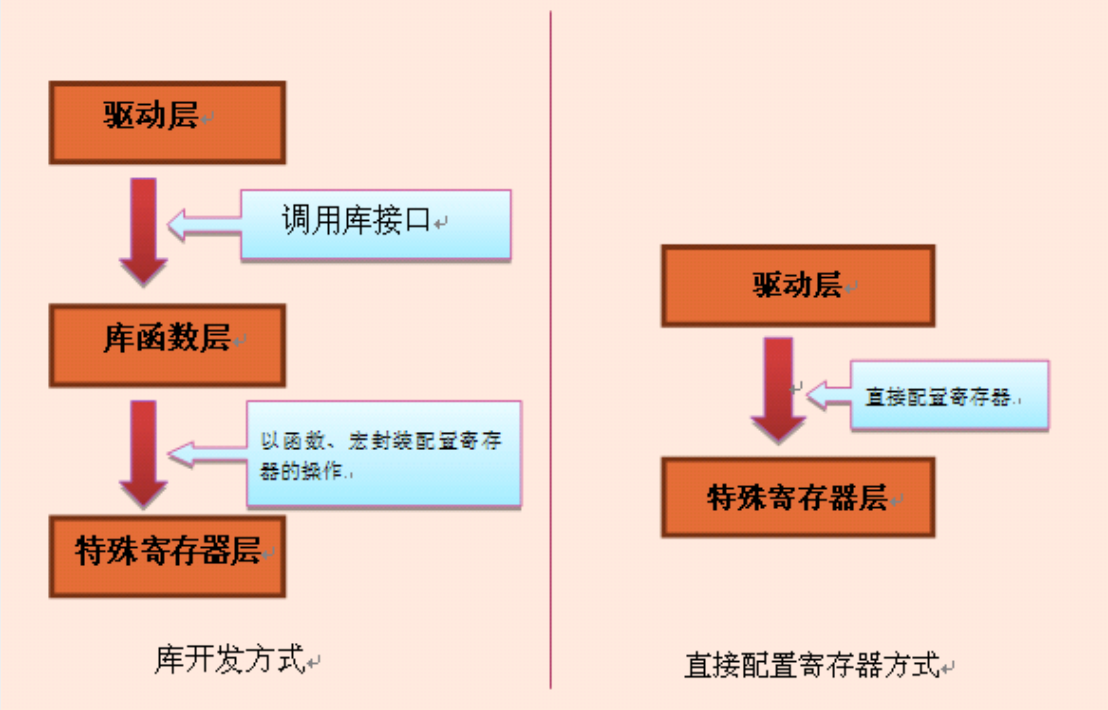


图 4-1

4.1.2 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 **错误！未找到引用源。**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会么采用 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（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（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 2)用宏表示参数，意义明确 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（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

4.2 STM32 结构及库层次关系

4.2.1 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。见**错误！未找到引用源。**。

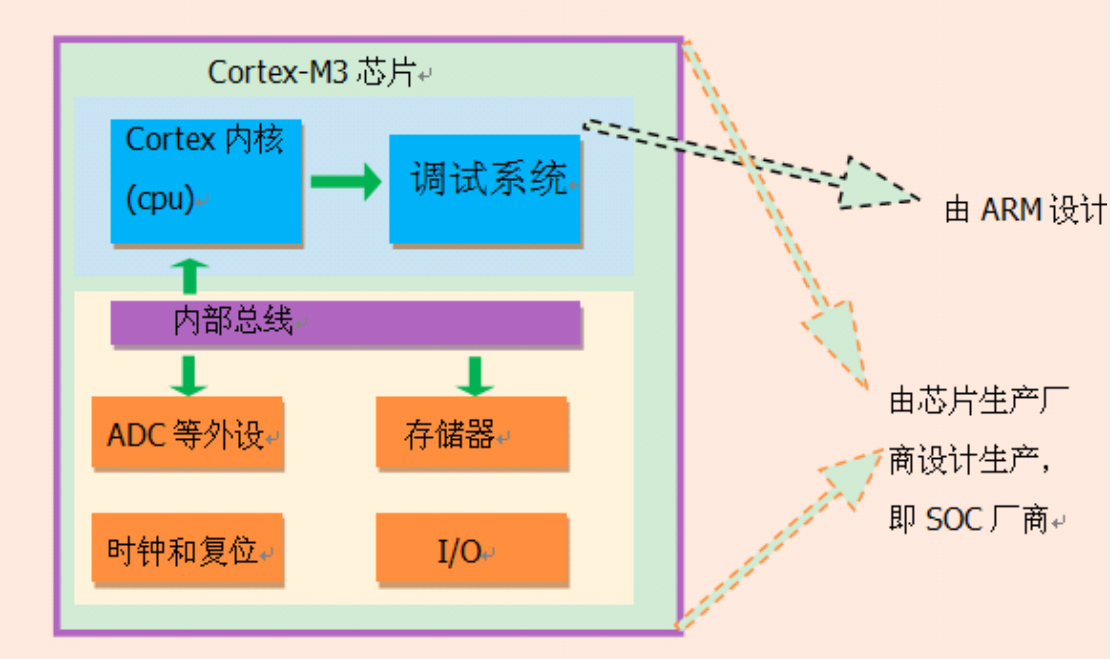
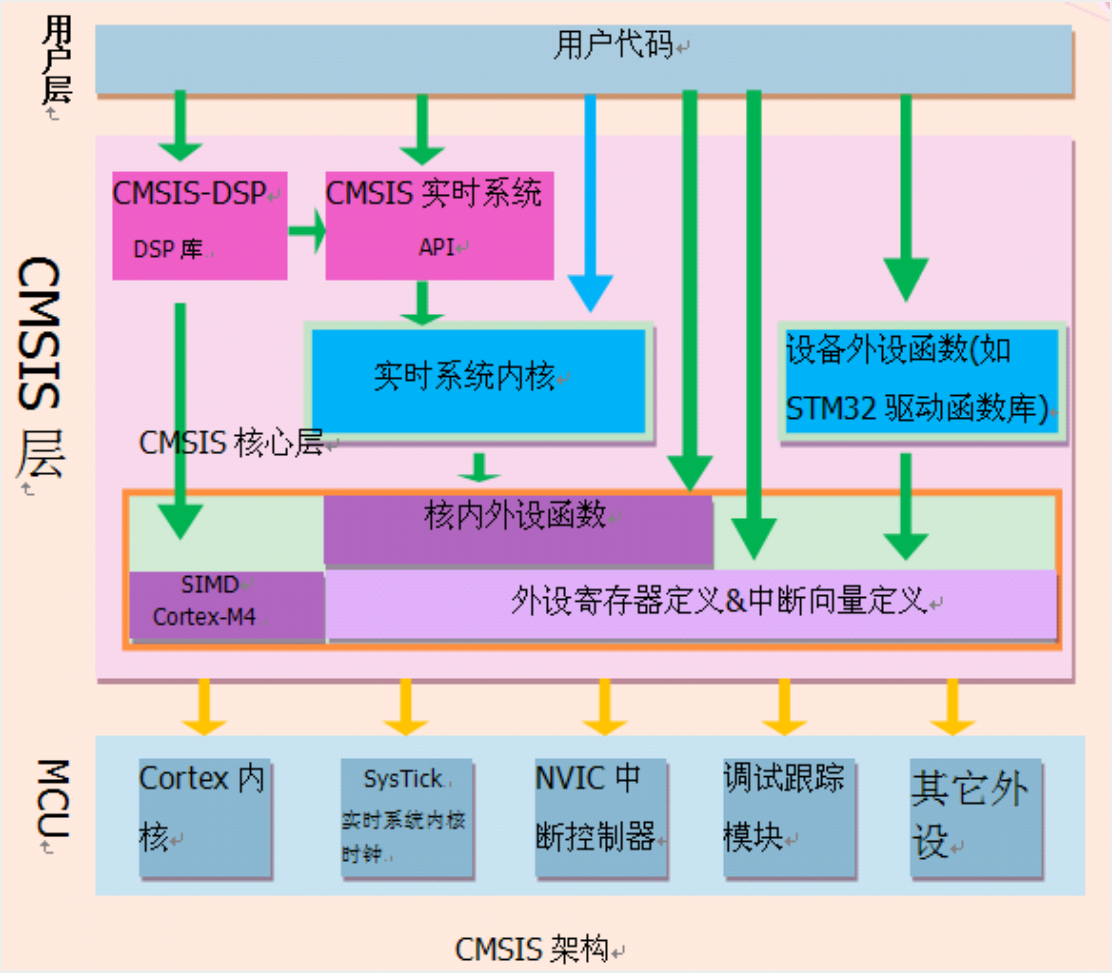


图 4-2

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。见**错误！未找到引用源。**。



错误！未找到引用源。

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。

4.2.2 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

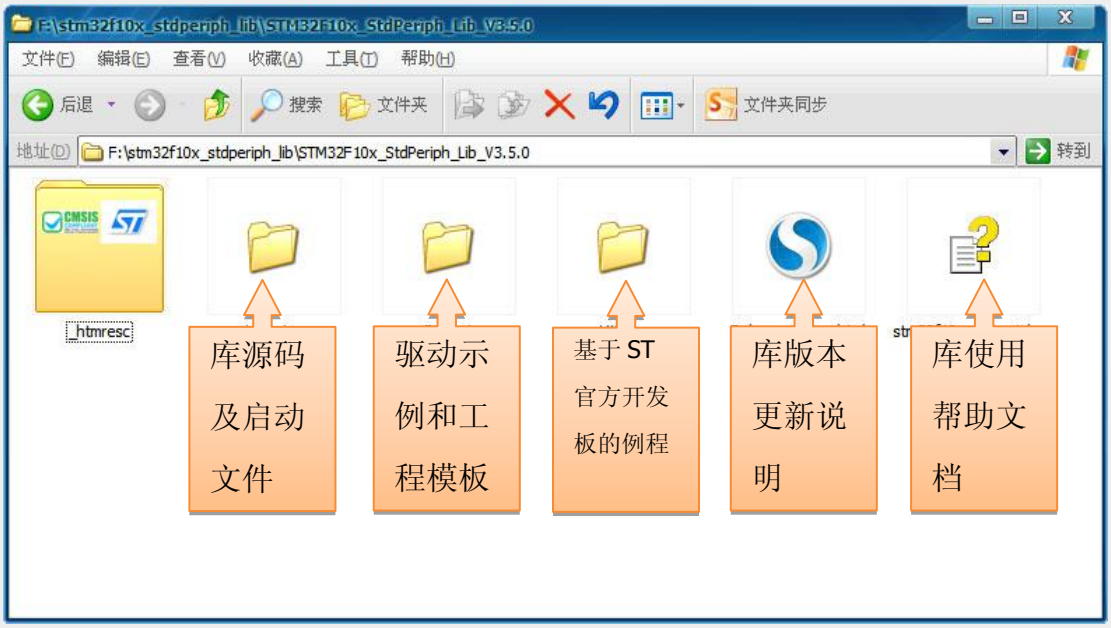


图 4-1

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。

4.2.2.1 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__  )
10.   #define __ASM            __asm
      #define __INLINE         inline
11.
12. #elif defined   (  __TASKING__  )
13.   #define __ASM            __asm
```

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

```
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。

4.2.2.2 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。

4.2.2.3 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。

4.2.2.4 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

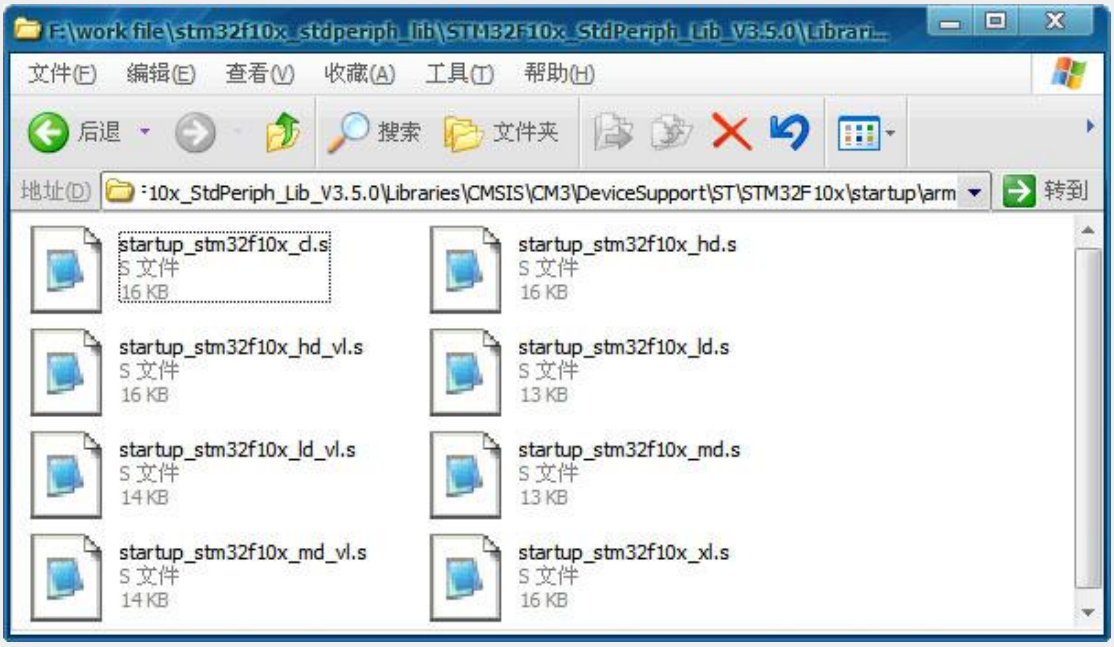


图 4-3

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点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。

4.2.2.5 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。

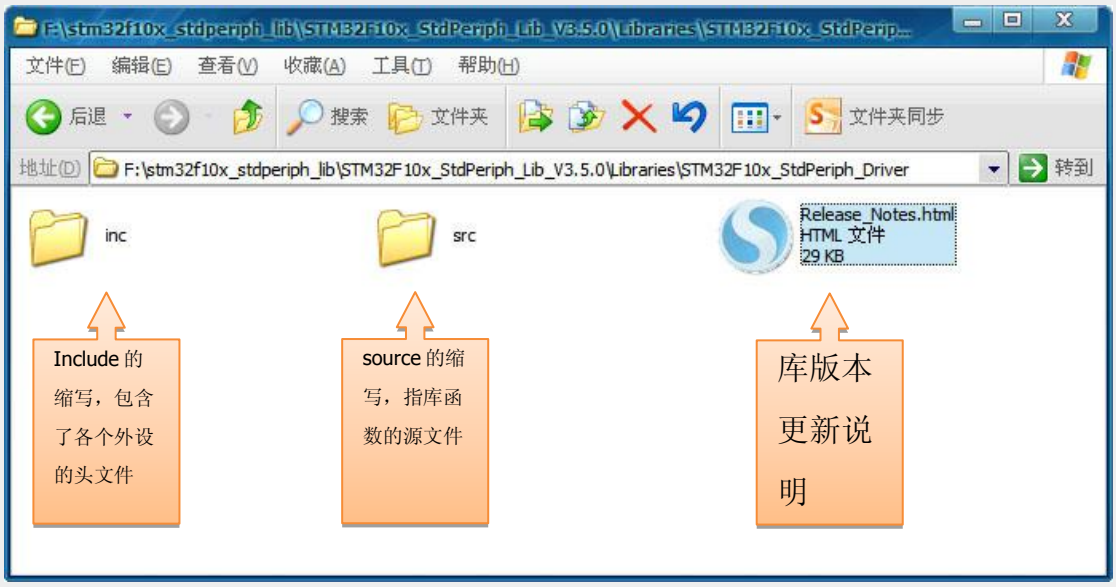


图 4-4

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。

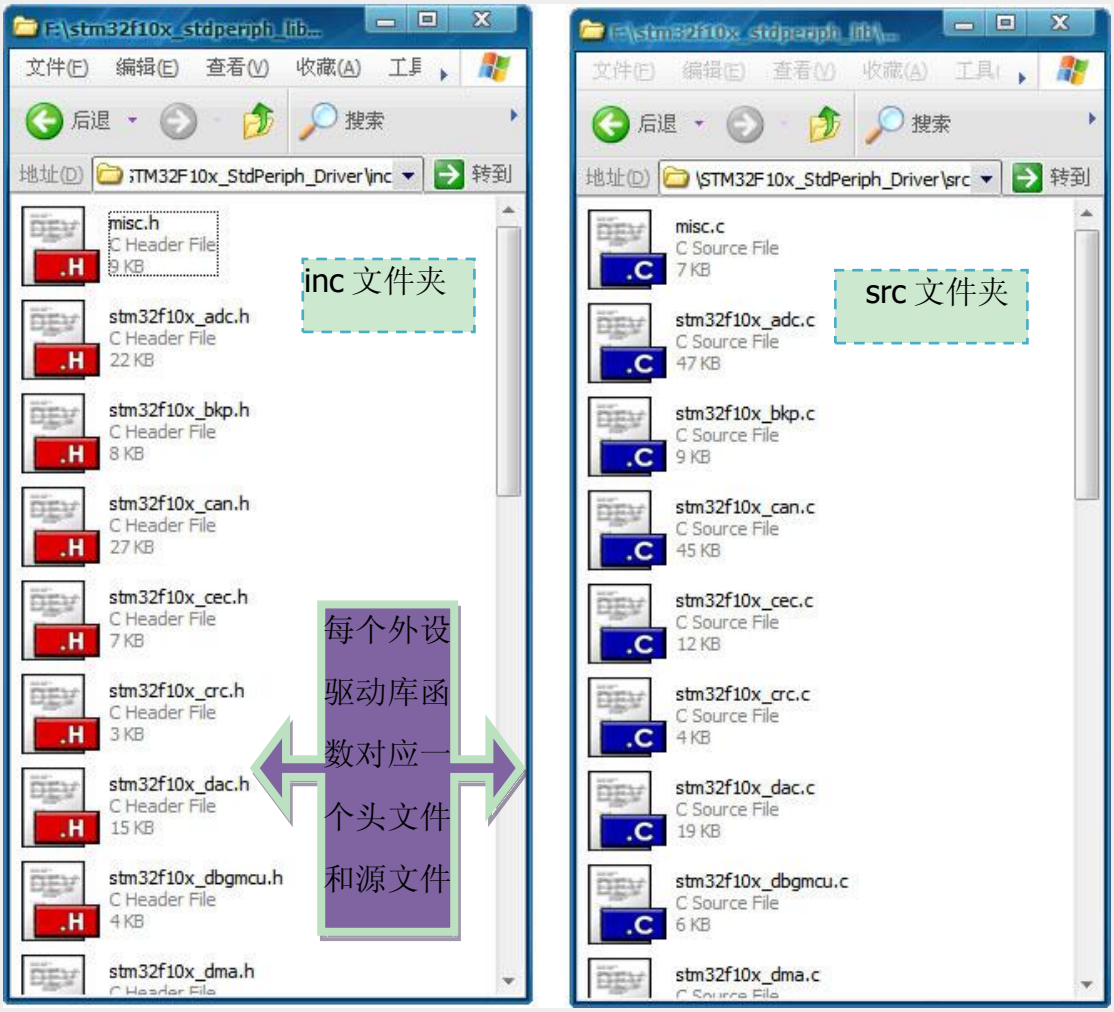


图 4-5

这两个文件夹中，还有一个很特别的 ***misc.c*** 文件，这个文件提供了外设对内核中的 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。

4.2.2.6 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 heade
   r 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。

4.2.3 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 0-6

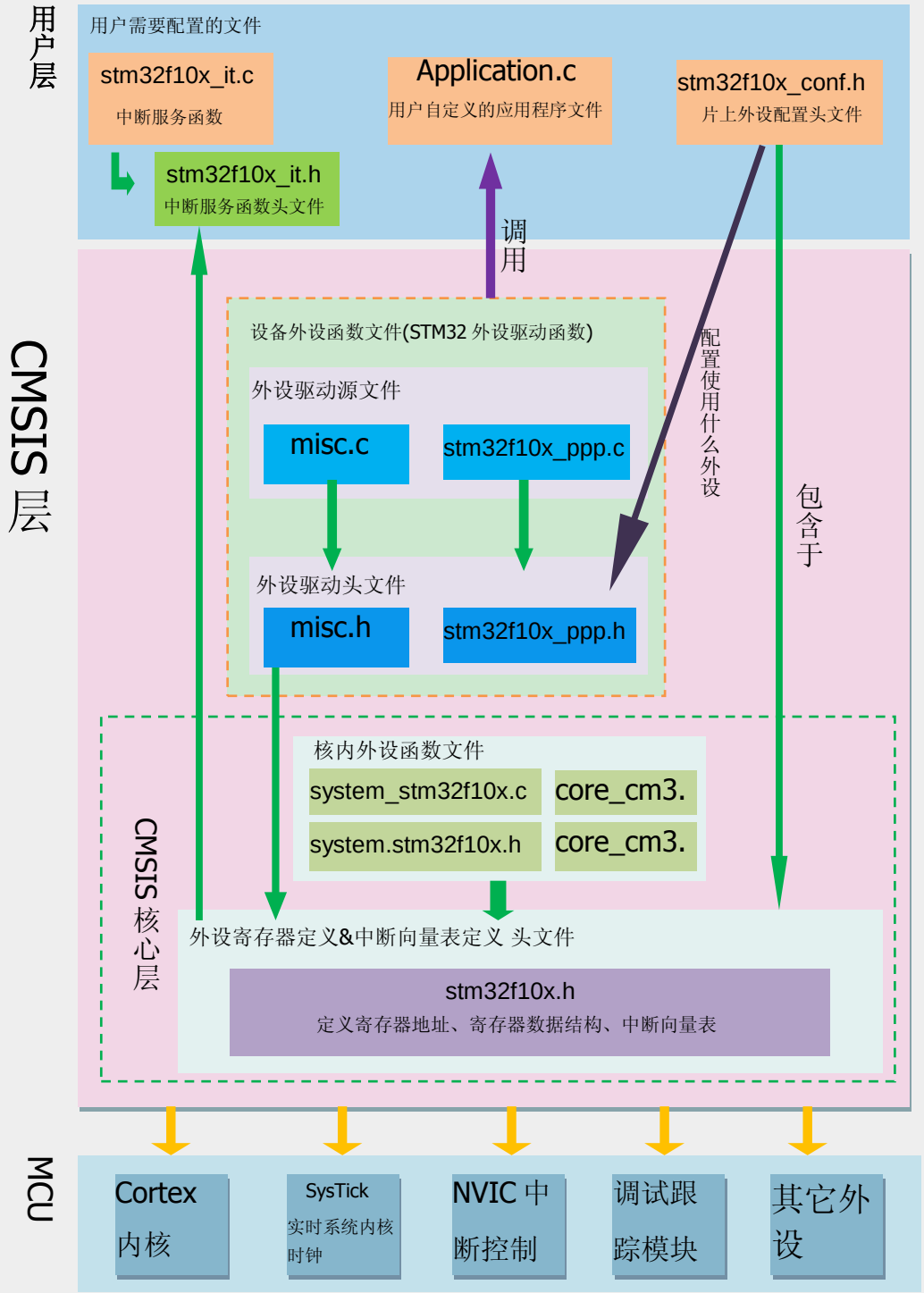


图 0-6（STM32 库文件结构）

图 0-6 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。

4.2.4 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。

4.2.4.1 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……

4.4.2.2 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 0-7

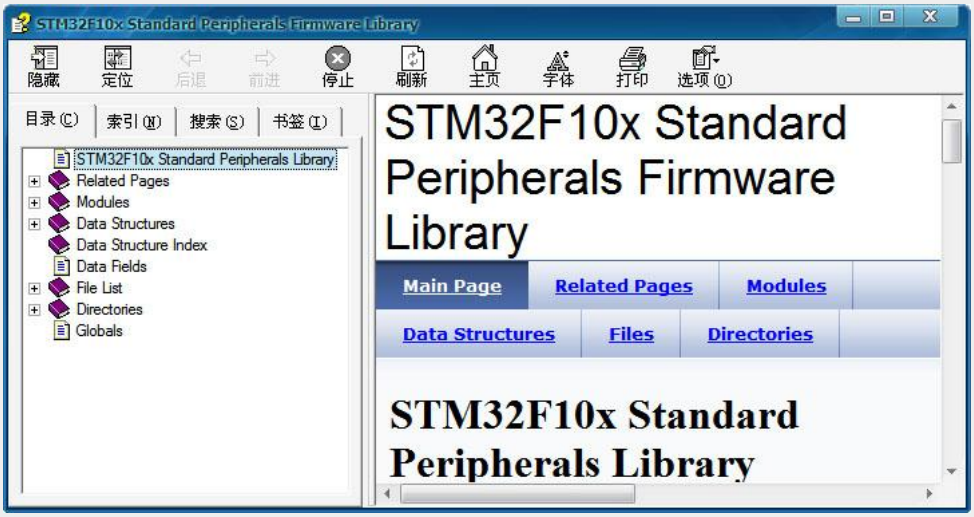


图 0-7

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

0-8

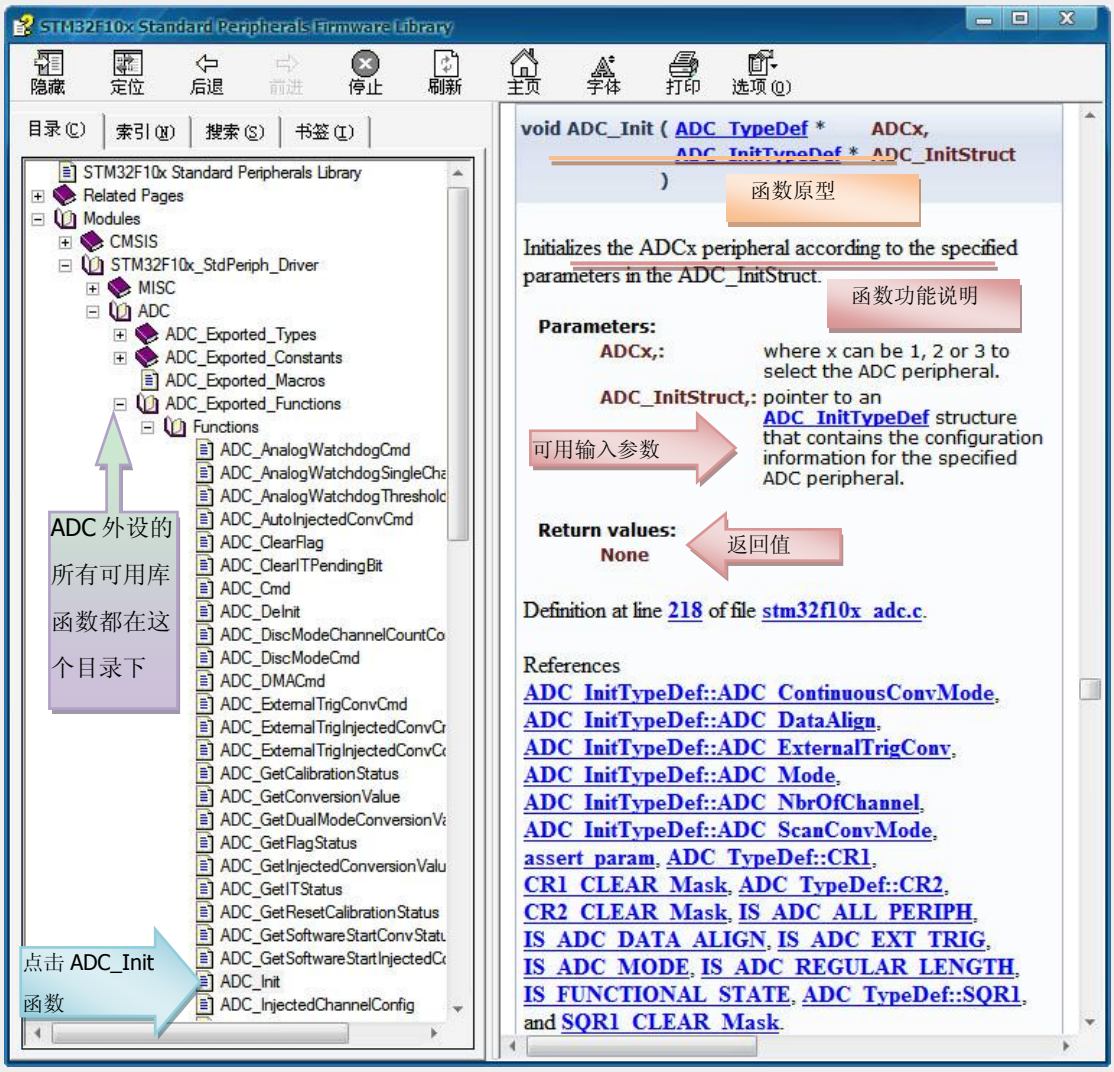


图 0-8

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_Init_TypeDef*** * ADC_InitStruct)

其中输入的参数 ADCx 和 ADC_InitSturct 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。

5、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。

5.1 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*，对于不同型号的芯片，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，见图 5-2 及图 5-3 **错误！未找到引用源。**

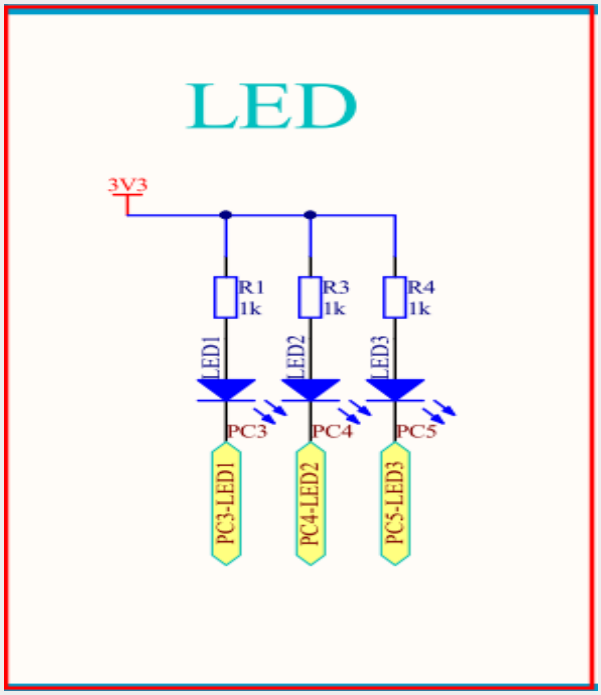


图 5-2

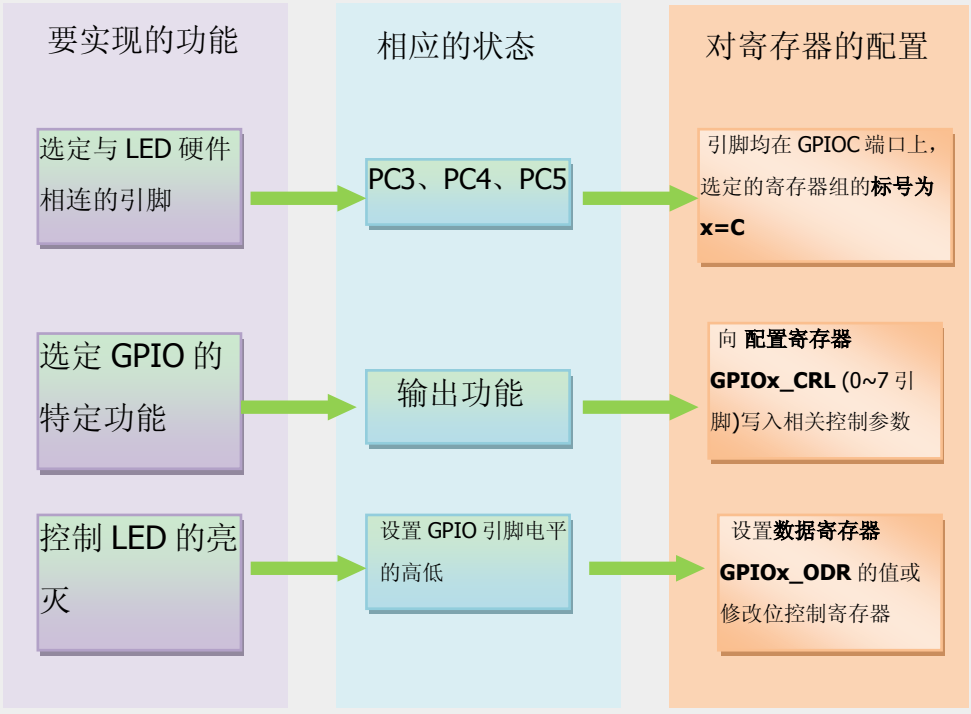


图 5-3

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。

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

每 4 个寄存器位配置一个引脚　　这 4 位控制 pin12

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

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

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。

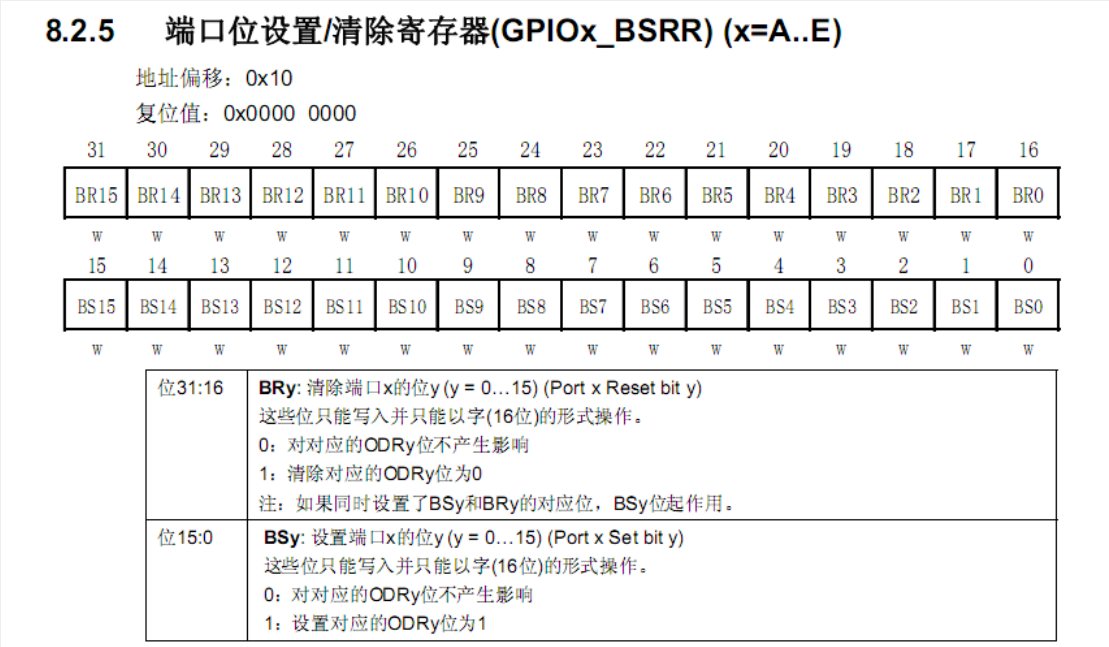
MODEy:向这两个位写入不同的值，设置引脚为不同的最大输出速率，或设置为输入模式。

图 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。



8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移：0x10

复位值：0x0000 0000

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

位31:16	**BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0 注：如果同时设置了BSy和IBRy的对应位，BSy位起作用。
位15:0	**BSy**: 设置端口x的位y (y = 0…15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：设置对应的ODRy位为1

图 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。

5.2 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.  {
```

```
    P0=0;
  while(1);
}
```

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：

```
/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
```

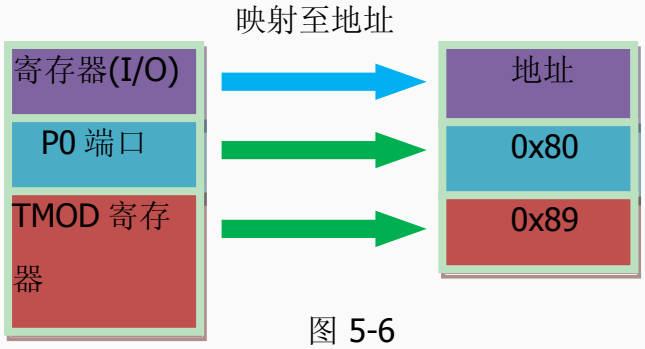


图 5-6

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} bit=4GB。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

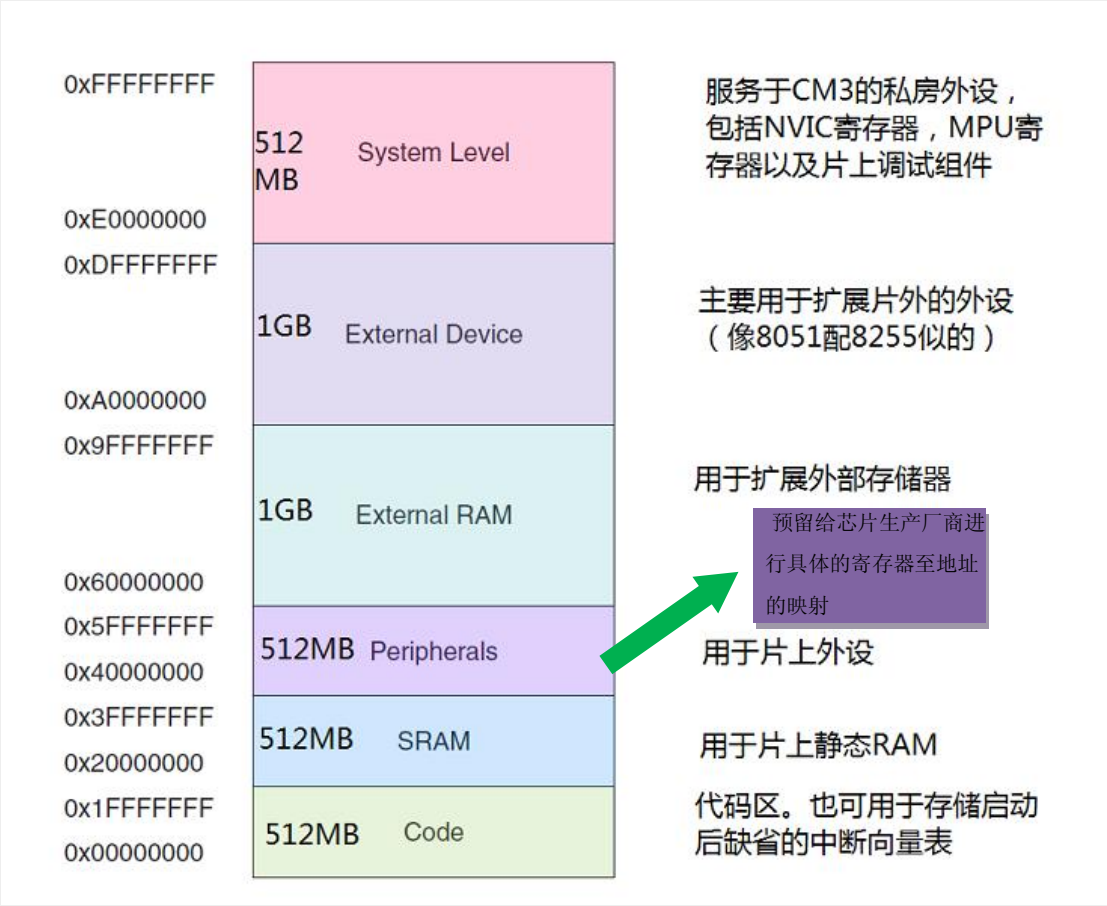


图 5-7

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

的外设有特定的地址范围。

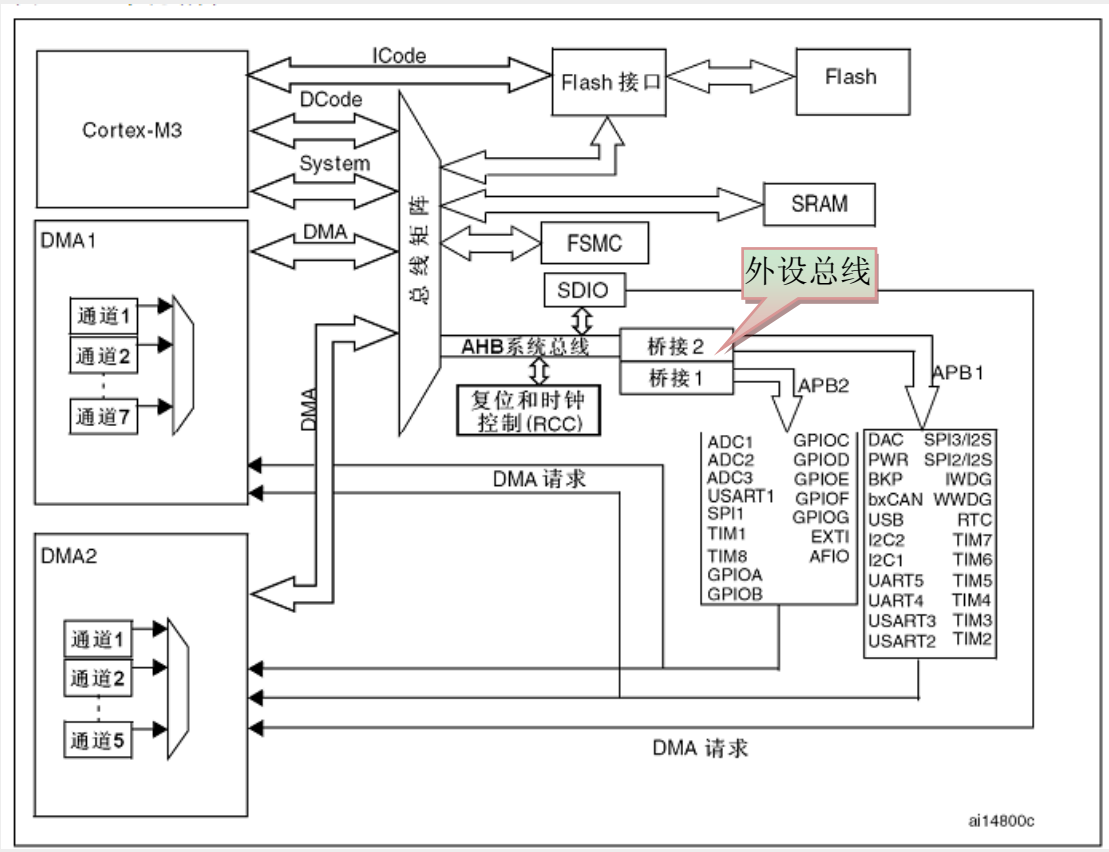


图 5-8

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（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

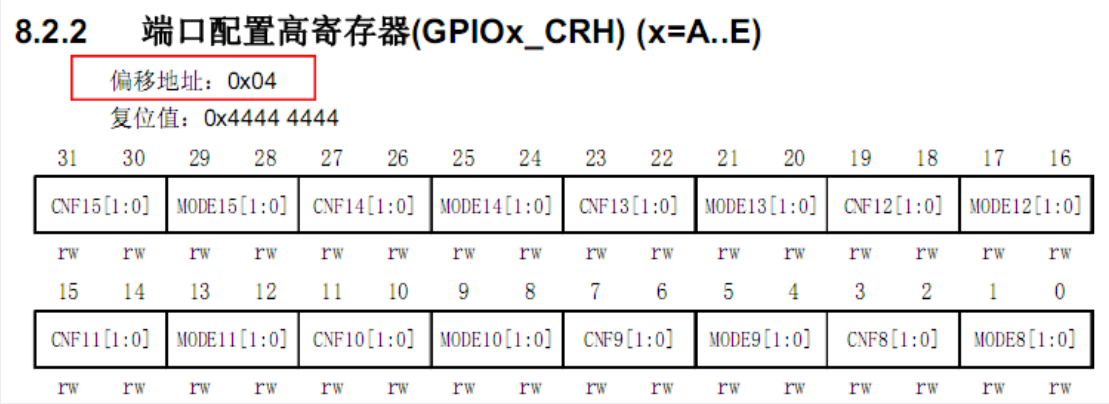
最后到了宏 ***GPIOC_BASE，宏展开为 APB2PERIPH_BASE*** (APB2 总线外设的基地址)加上相对 APB2 总线基地址的偏移量 ***0x1000*** 得到了 GPIOC 端口的寄存器组的基地址。这个所谓的寄存器组又是什么呢？它包括什么寄存器？

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。

5.3 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 */
```

volatitle 是 c 语言的一个关键字，有关 volatitle 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。

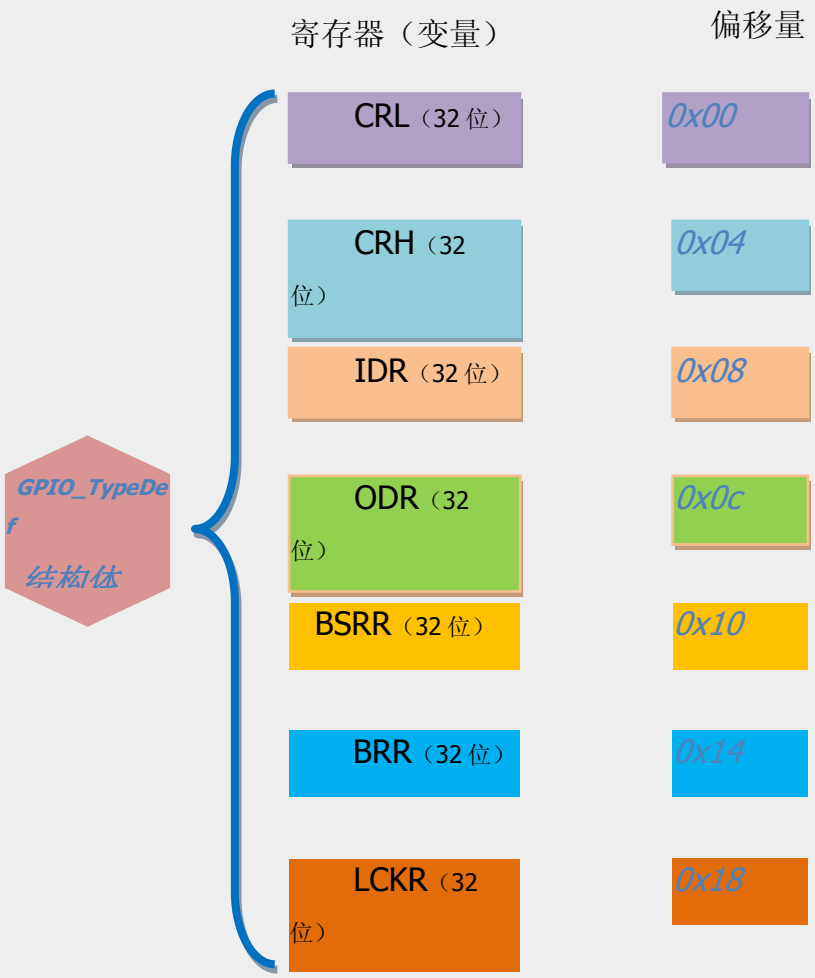


图 0-10

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 组寄存器的基地址。***(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。

5.4 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。

5.4.1 时钟树&时钟源

首先，从整体上了解 STM32 的时钟系统。见图 0-11

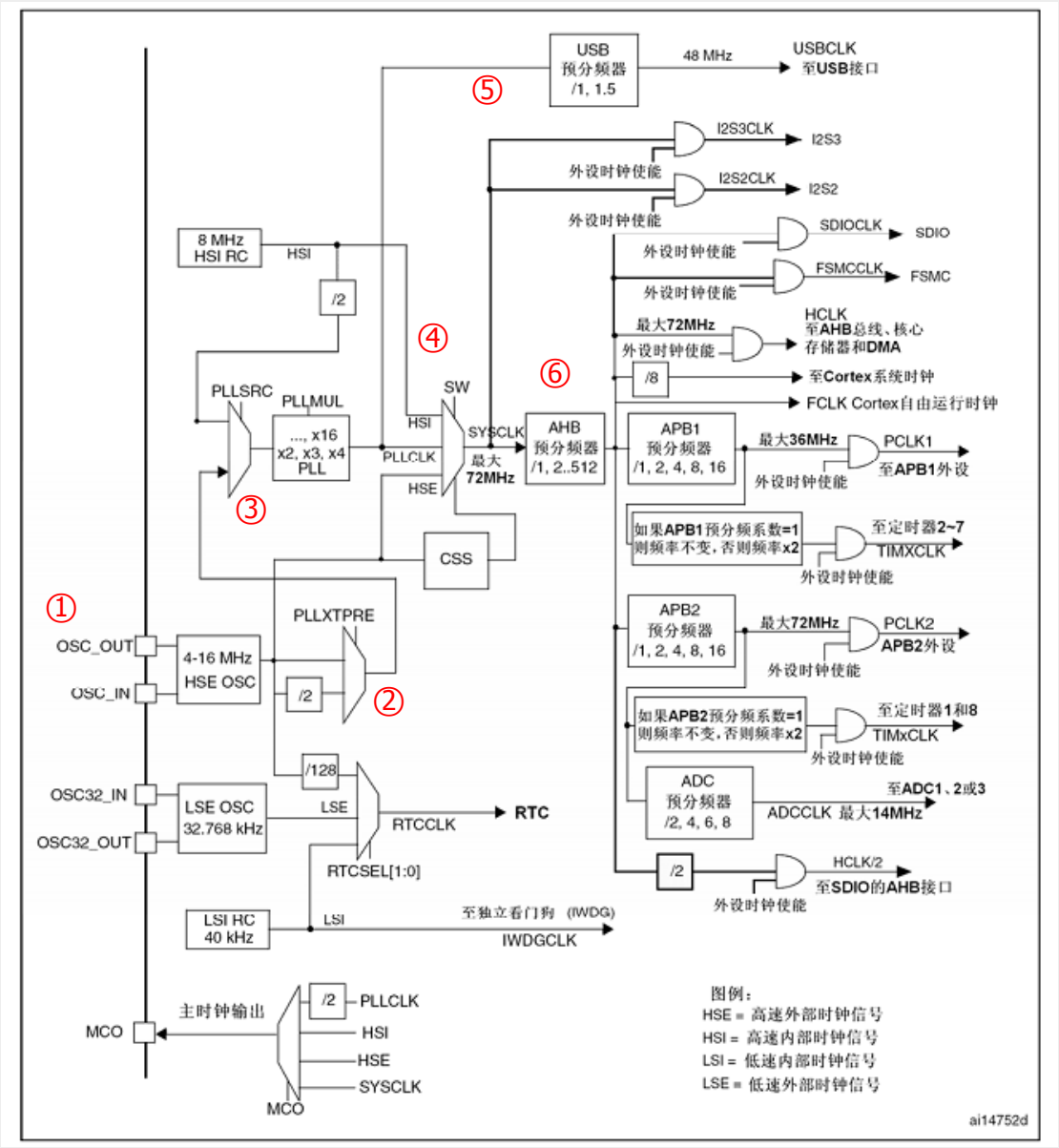


图 0-11

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。

5.4.2 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。

5.4.3 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。

5.5 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。

5.5.1 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	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c*
用户编写的文件	*USER/main.c*

	USER/stm32f10x_it.c *USER/led.c* *USER/led.h*

5.5.2 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。

5.5.3 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.     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.
40.     /*调用库函数，初始化 GPIOC*/
41.     GPIO_Init(GPIOC, &GPIO_InitStructure);
42.
43.     /* 关闭所有 led 灯 */
44.     GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
45. }
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。

5.5.4 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，GPIOMode_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. }GPIOMode_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 引脚的模式的，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。

5.5.5 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 0 使用库帮助文档小节。查询结果见图 0-12。

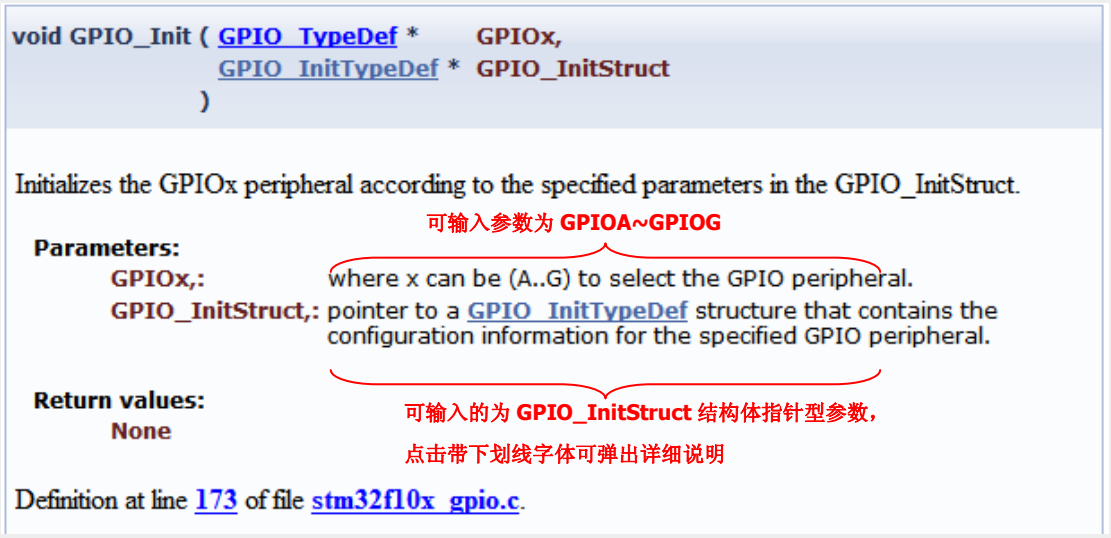


图 0-12 GPIO_Init 函数

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，说明它将要对 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 0 小节进行详细分析。

5.5.6 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， 0 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。

5.5.6.1 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 0 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。

5.5.6.2 开启外设时钟

SYSCLK 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 0-13

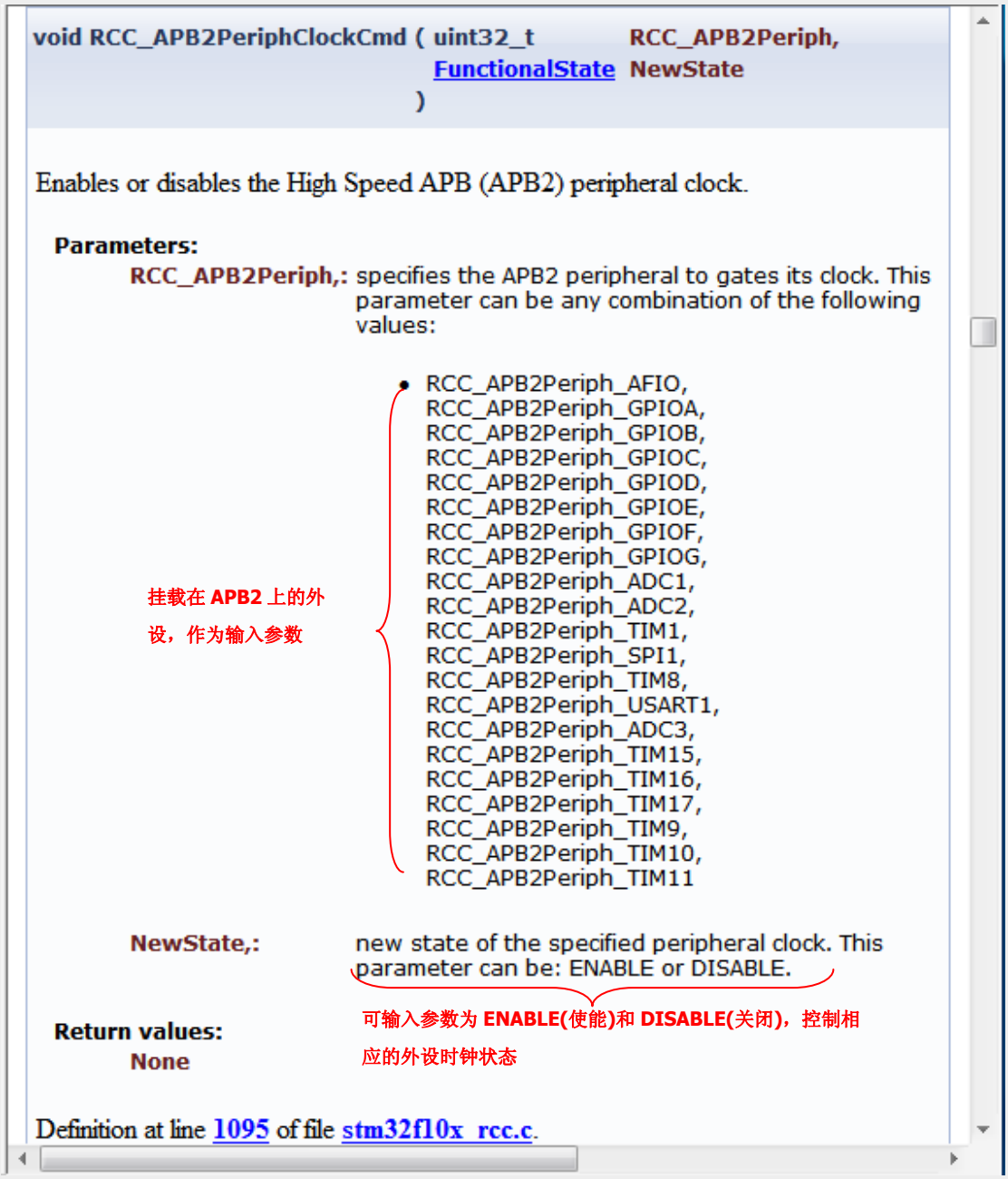


图 0-13 APB2 时钟使能函数

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。

5.5.7 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 0-14 及图 0-15

```
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 0-14 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 0-15 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。

5.5.8 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。

5.5.9 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(0x0FFFEF);
31.         LED1( OFF );           // 灭
```

```
32.
33.         LED2( ON );
34.         Delay(0x0FFFEF);
35.         LED2( OFF );
36.
37.         LED3( ON );
38.         Delay(0x0FFFEF);
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。

5.5.10 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。

5.6 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。

5.6.1 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
```

5.6.2 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 = 0x
   00;
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. /*判断是否为下拉输入模式*/
46.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
47.         {
48. /*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*/
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 0-16，及**错误！未找到引用源。**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]	
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	rw	rw	rw	rw	rw	rw	rw	rw

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

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 0-16 GPIOx_CRL 寄存器

图 0-17 GPIO_Init 分析

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。

5.6.3 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，确定这样的一个值，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：

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

6、Sysstick（系统滴答定时器）

6.1 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。

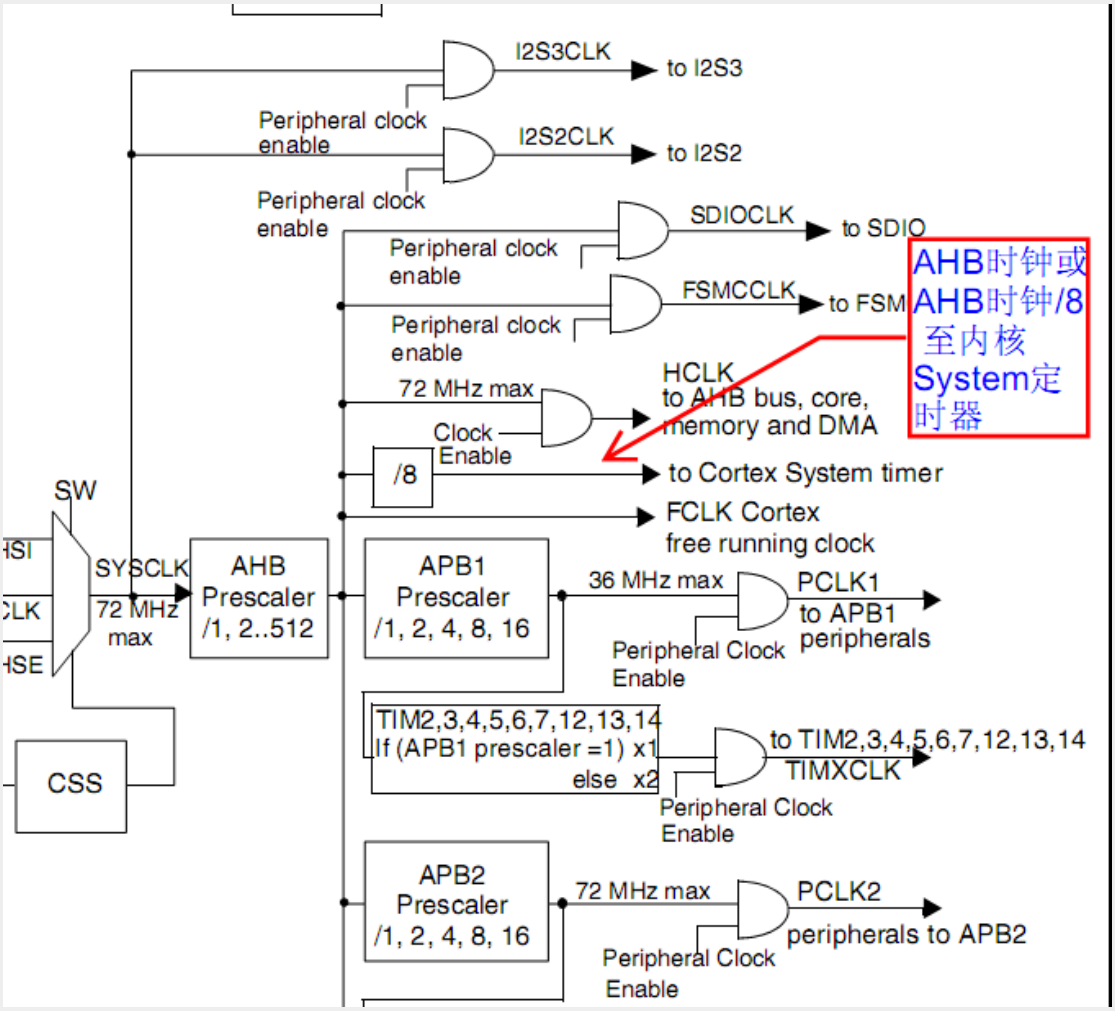


图 6-1 时钟树（部分）-SysTick timer 时钟来源

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

6.2 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-。

The SysTick CTRL register enables the SysTick features.

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

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 0-2，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	Reserved															COUNTFLAG	Reserved													CLKSOURCE	TICK INT	EN ABLE
	Reset Value																0														1	0	0
0x04	STK_LOAD	Reserved								RELOAD[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	STK_VAL	Reserved								CURRENT[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	STK_CALIB	Reserved								TENMS[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 0-2 SysTick 寄存器映像

6.3 SysTick 精确延时实例精讲

前面的的实验例程中，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。

6.3.1 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	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c*
用户编写的文件	*USER/main.c* *USER/stm32f10x_it.c* *USER/led.c* *USER/led.h* USER/ SysTick.c

6.3.2 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。

```
/**
  ******************************************************
  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
  * @author  MCD Application Team
  * @version V3.5.0
  * @date    08-April-2011
  * @brief   Library configuration file.
  ******************************************/

#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
```

6.3.3 main 文件

我们从看 main 函数看起：

```
/*
 * 函数名：main
 * 描述  ：主函数
 * 输入  ：无
 * 输出  ：无
 */
int main(void)
{
    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 配置 SysTick 为 10us 中断一次 */
    SysTick_Init();

    for(;;)
    {

        LED1( 0 );
        Delay_us(50000);        // 50000 * 10us = 500ms
        LED1( 1 );

        LED2( 0 );
        Delay_us(50000);        // 50000 * 10us = 500ms
        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。

6.3.4 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
    le SysTick IRQ and SysTick Timer */
28.   return (0);                                                  /* Func
    tion successful */
29. }
```

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，出错进返回 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 0 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            (0xFFFFFFul << 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            (0xFFFFFFul << 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             (0xFFFFFFul << 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;
```

6.3.5 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（SYSCLK）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：

$$T=ticks*(1/f)$$

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 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
```

6.3.6 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 $T_{延时} = T_{中断周期} * TimingDelay$ 。

至此，SysTick 的精确延时功能讲解完毕。

6.3.7 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。

6.3.8 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。

7、KEY（Polling）

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。

7.1 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：

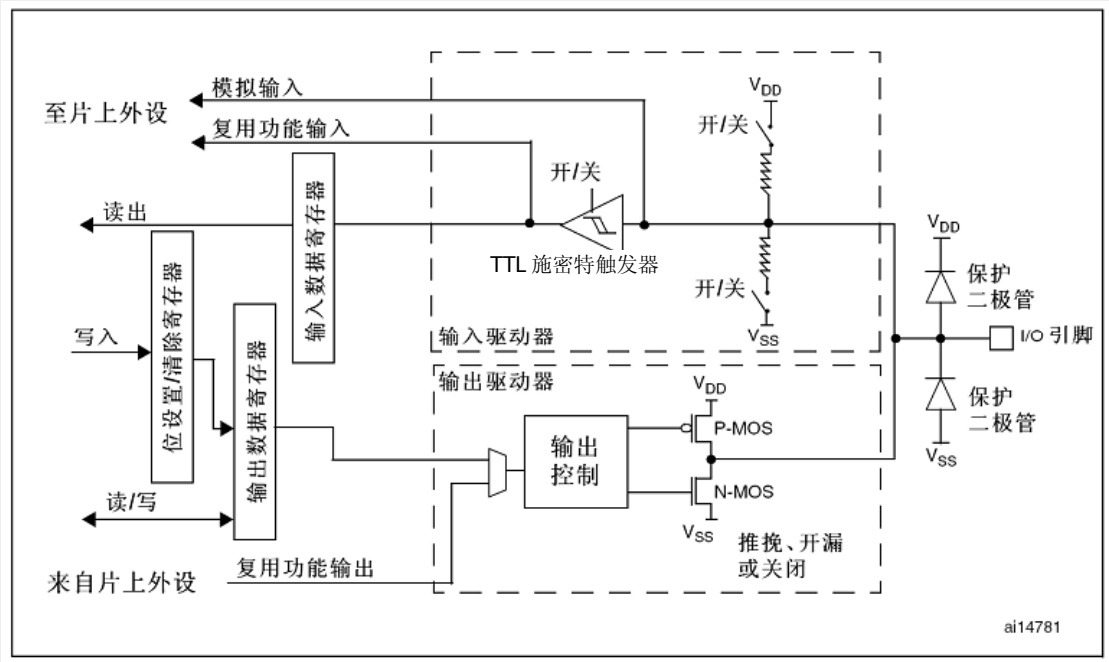


图 7-1 GPIO 结构图

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。

7.1.1 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。*

7.1.2 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“*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。

7.2 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，采且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）：

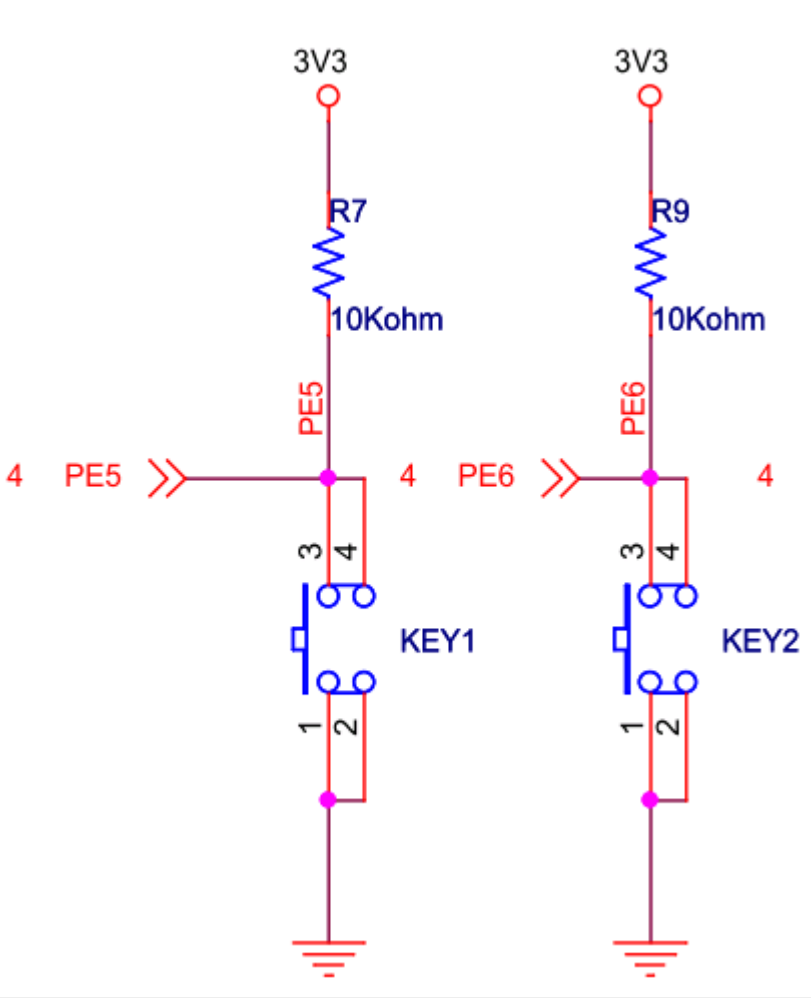


图 7-2 野火 STM32 开发板按键硬件图

7.3 按键代码分析

7.3.1 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	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c*

	FWlib/stm32f10x_gpio.c *FWlib/stm32f10x_rcc.c*
用户编写的文件	*USER/main.c* *USER/stm32f10x_it.c* *USER/led.c* USER/key.c

7.3.2 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。

```
/**
  ******************************************************
  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
  * @author  MCD Application Team
  * @version V3.5.0
  * @date    08-April-2011
  * @brief   Library configuration file.
  ******************************************************/

#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
```

7.3.3 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。

```
/*
 * 函数名：main
 * 描述  ：主函数
 * 输入  ：无
 * 输出  ：无
 */
int main(void)
{
    /* config the led */
    LED_GPIO_Config();
    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-
    GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_3))));
23.         }
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。

7.3.4 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。

7.3.5 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：

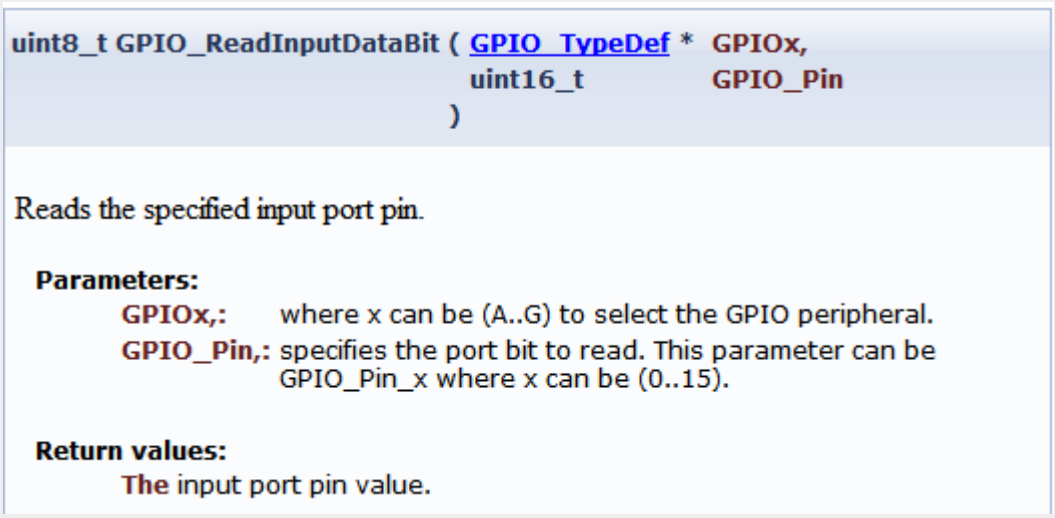
```
1.  /*
2.   * 函数名：Key_Scan(GPIO_TypeDef* GPIOx,u16 GPIO_Pin)
3.   * 描述  ：检测是否有按键按下
4.   * 输入  ：GPIOx：x 可以是 A，B，C，D 或者 E
5.              GPIO_Pin：待读取的端口位
6.   * 输出  ：KEY_OFF(没按下按键)、KEY_ON（按下按键）
7.   */
8.  u8 Key_Scan(GPIO_TypeDef* GPIOx,u16 GPIO_Pin)
9.  {
10.          /*检测是否有按键按下 */
11.     if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == 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，若从相应引脚读取得的数据等于 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；



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

Reads the specified input port pin.

Parameters:

GPIOx,: where x can be (A..G) to select the GPIO peripheral.

GPIO_Pin,: specifies the port bit to read. This parameter can be GPIO_Pin_x where x can be (0..15).

Return values:

The input port pin value.

图 0-3 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,u16 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！

7.3.6 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，相当于获取一个也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，我们只不过是要取反输出，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。

7.3.7 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 灭，如此循环。

8、EXTI 之按键中断实验

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。

8.1 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 RCC时钟安全系统(CSS)联接到NMI向量	0x0000_0008
	-1	固定	硬件失效(HardFault)	所有类型的失效	0x0000_000C
	0	可设置	存储管理(MemManage)	存储器管理	0x0000_0010
	1	可设置	总线错误(BusFault)	预取指失败，存储器访问失败	0x0000_0014
	2	可设置	错误应用(UsageFault)	未定义的指令或非法状态	0x0000_0018
	-	-	-	保留	0x0000_001C ~0x0000_002B
	3	可设置	SVCall	通过SWI指令的系统服务调用	0x0000_002C
	4	可设置	调试监控(DebugMonitor)	调试监控器	0x0000_0030
	-	-	-	保留	0x0000_0034
	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C
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^2C1事件中断	0x0000_00BC
32	39	可设置	I2C1_ER	I^2C1错误中断	0x0000_00C0
33	40	可设置	I2C2_EV	I^2C2事件中断	0x0000_00C4
34	41	可设置	I2C2_ER	I^2C2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。

8.2 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。

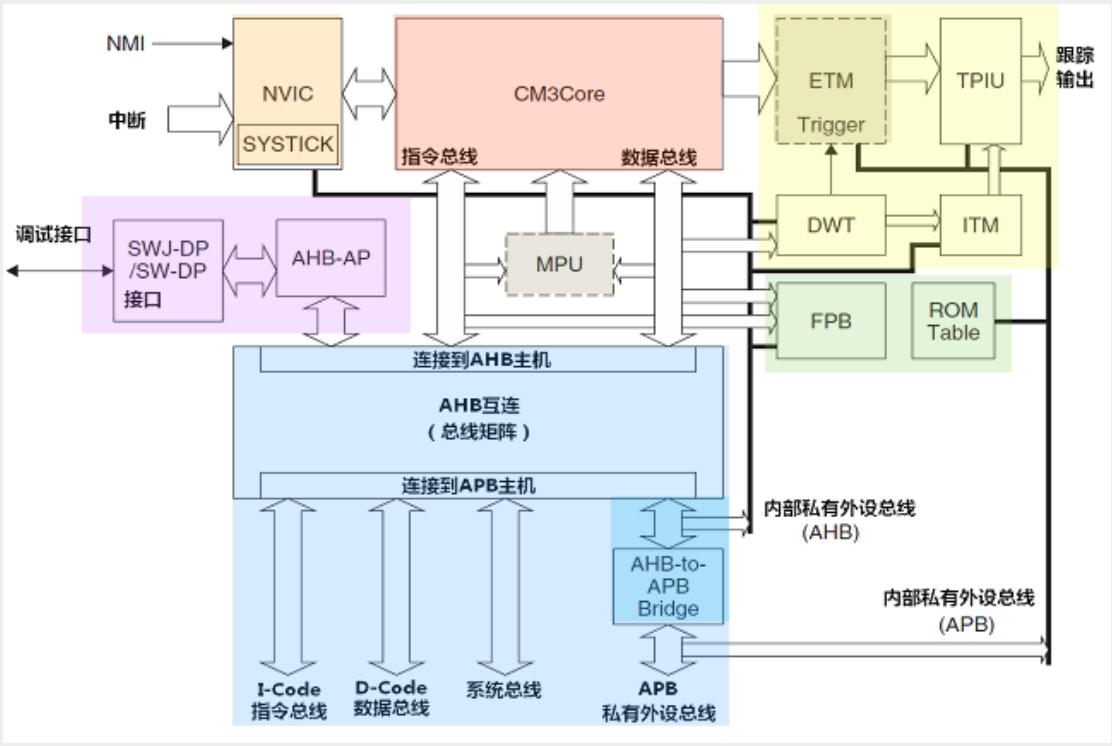


图 8-2 NVIC 在内核中的位置

8.2.1 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

NVIC_IRQChannel	需要配置的中断向量
NVIC_IRQChannelCmd	使能或关闭相应中断向量的中断响应
NVIC_IRQChannelPreemptionPriority	配置相应中断向量抢占优先级
NVIC_IRQChannelSubPriority	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？

8.2.2 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。

8.2.3 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。

STM2 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

8.3 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 0-3

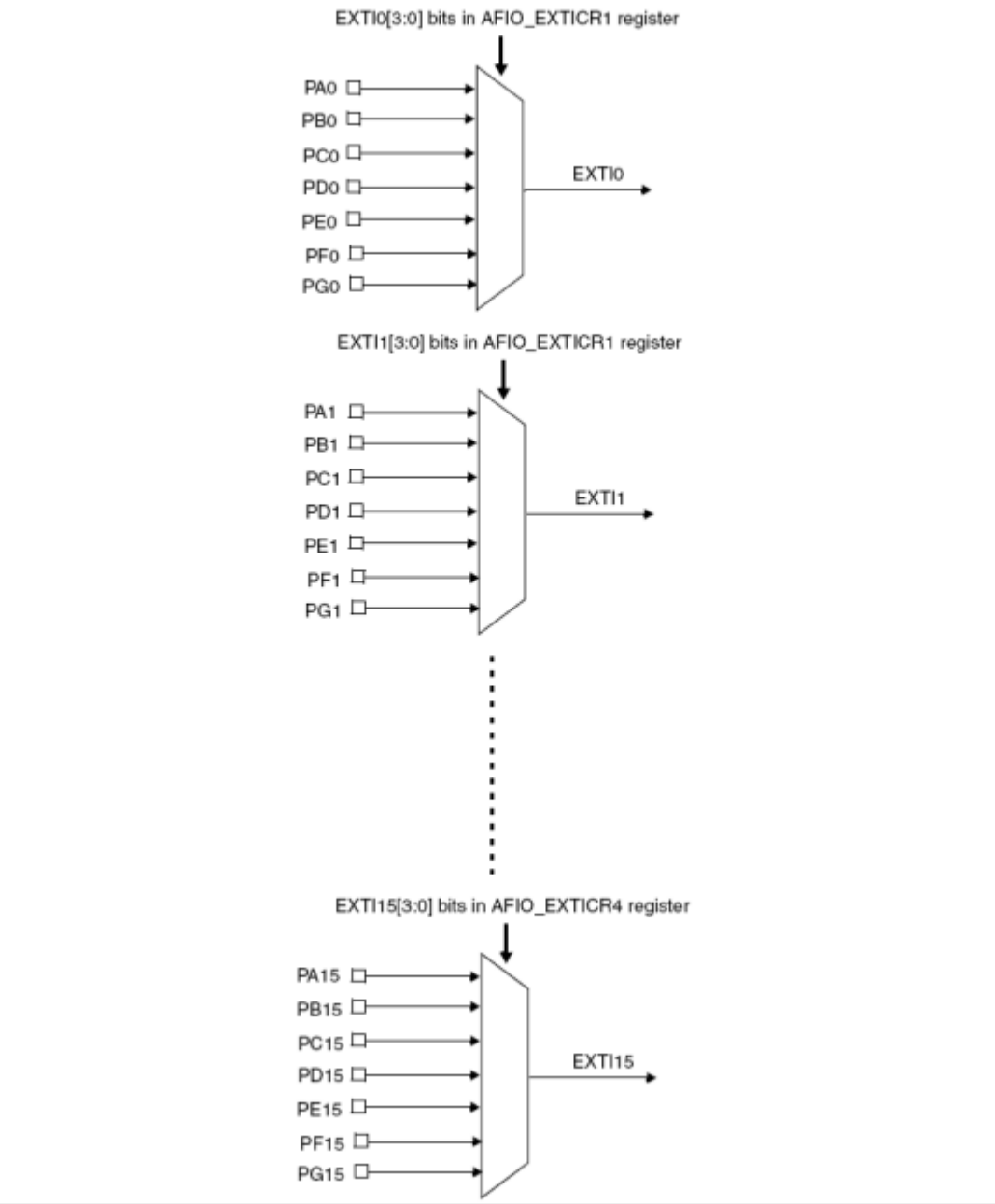


图 0-3 EXTI 与 GPIO 连接图

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。

8.4 中断检测按键实验分析

8.4.1 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	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c* *FWlib/stm32f10x_exti.c* *FWlib/misc.c*
用户编写的文件	*USER/main.c* *USER/stm32f10x_it.c* *USER/led.c* *USER/exti.c*

8.4.2 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。

```
/**
  ******************************************************
  * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
  * @author  MCD Application Team
  * @version V3.5.0
  * @date    08-April-2011
  * @brief   Library configuration file.
  ******************************************************/
#include "stm32f10x_exti.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "misc.h" /
```

8.4.5 main 文件

我们从 main 函数开始分析：

```
/*
 * 函数名：main
 * 描述  ：主函数
 * 输入  ：无
 * 输出  ：无
 */
int main(void)
{
    /* config the led */
    LED_GPIO_Config();
    LED1( ON );

    /* exti line config */
    EXTI_PE5_Config();

    /* wait interrupt */
    while(1)
    {
    }
}
```

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。

8.4.6 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,
  ENABLE);
15.
16.        /* config the NVIC(PE5) */
17.        NVIC_Configuration();
18.
19.        /* EXTI line gpio config(PE5) */
20.      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
21.      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;  // 上拉输入
22.      GPIO_Init(GPIOE, &GPIO_InitStructure);
23.
24.        /* EXTI line(PE5) mode config */
25.        GPIO_EXTILineConfig(GPIO_PortSourceGPIOE, GPIO_PinSource5);
26.      EXTI_InitStructure.EXTI_Line = EXTI_Line5;
27.      EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
28.      EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿中
  断
29.      EXTI_InitStructure.EXTI_LineCmd = ENABLE;
30.      EXTI_Init(&EXTI_InitStructure);
31.    }
```

8.4.7 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 0-4。

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(1)	I/O电平(2)	主功能(3)(复位后)	可选的复用功能	
BGA144	BGA100	WLCSP64	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 0-4 GPIO 引脚功能说明

8.4.8 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。

8.4.9 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。

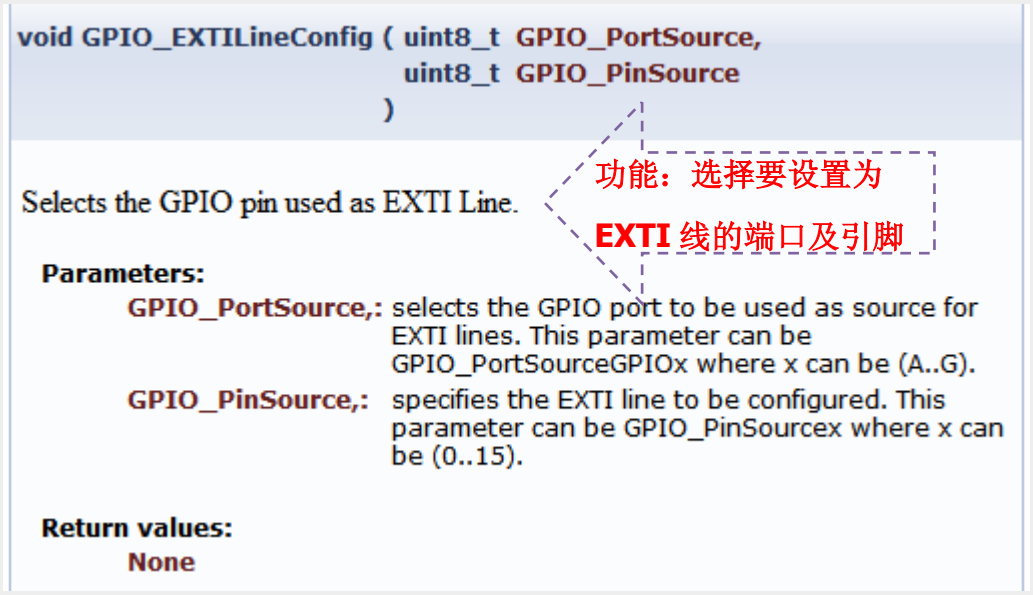


图 0-5 EXTI 中断源配置函数

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。

8.4.10 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_5IRQHandler*，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()* 清除中断标置位再退出中断服务函数。这两个函数的解释见图 0-6 及图 0-7。

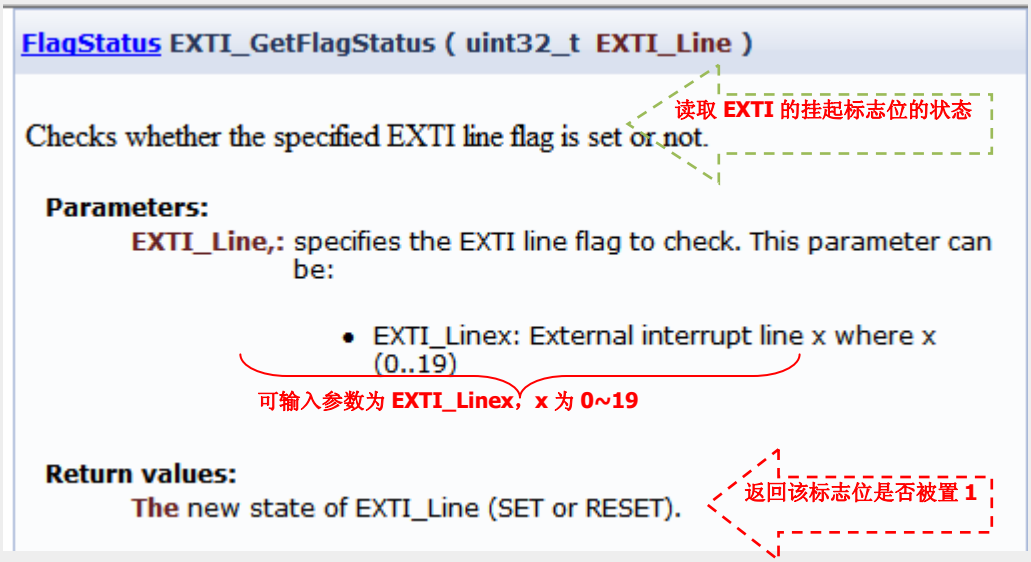


图 0-6EXTI 状态检查函数

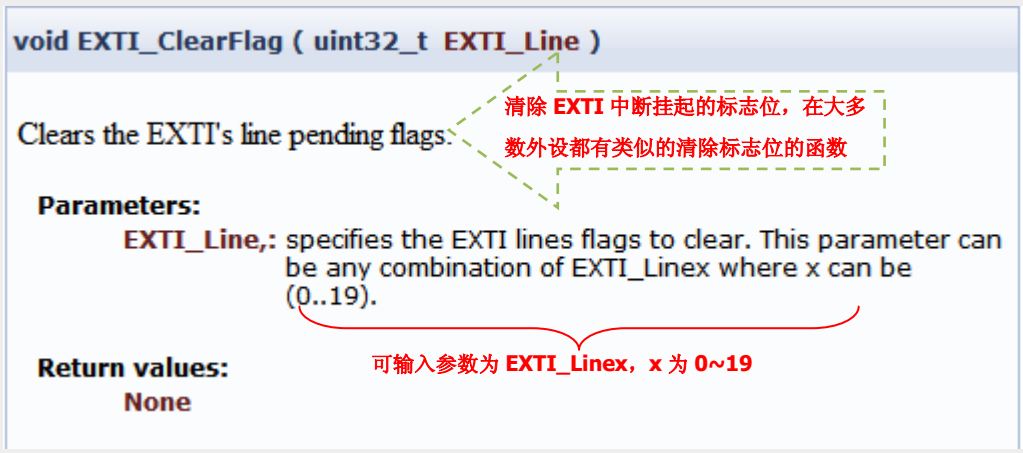


图 0-7 EXTI 清除标志位函数

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* 这样子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。

8.4.11 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。