# 前言

一天入门 STM32，仅一天的时间，是否有真的这么快。不同的人对入门的理解不一样，这篇一天入门 STM32 的教程，我们先对入门达成一个共识，如果你有异议，一天入门不了，请不要较真，不要骂街，保持一个工程师该有的修养，默默潜心学习，因为你还有很大的上升空间。

我眼中的入门：（前提是你学过 51 单片机和 C 语言）

1、知道参考官方的什么资料来学习，而不是陷入一大堆资料中无从下手。

2、知道如何参考官方的手册和官方的代码来独立写自己的程序，而不是一味的看到人家写的代码就觉得人家很牛逼。

3、消除对 STM32 的恐惧，消除对库开发的恐惧，学习是一个快乐而富有成就感的过程。

# 第 1 章　一天入门 STM32

本章参考资料：《STM32 中文参考手册》　《CM3 权威指南 CnR2》

学习本章时，配合《STM32 中文参考手册》GPIO 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

## 1.1 51 与 STM32 简介

51 是嵌入式学习中一款入门级的精典 MCU，因其结构简单，易于教学，且可以通过串口编程而不需要额外的仿真器，所以在教学时被大量采用，至今很多大学在嵌入式教学中用的还是 51。51 诞生于 70 年代，属于传统的 8 位单片机，如今，久经岁月的洗礼，既有其辉煌又有其不足。现在的市场产品竞争激烈，对成本极其敏感，相应地对 MCU 的要求也更苛刻：功能更多，功耗更低，易用界面和多任务。面对这些要求，51 现有的资源就显得得抓襟见肘了。所以无论是高校教学还是市场需求，都急需一款新的 MCU 来为这个领域注入新的活力。

基于这市场的需求， ARM 公司推出了其全新的基于 ARMv7 架构的 32 位 Cortex-M3 微控制器内核。紧随其后，ST（意法半导体）公司就推出了基于 Cortex-M3 内核的 MCU—STM32。STM32 凭借其产品线的多样化、极高的性价比、简单易用的库开发方式，迅速在众多 Cortex-M3 MCU 中脱颖而出，成为最闪亮的一颗新星。STM32 一上市就迅速占领了中低端 MCU 市场，受到了市场和工程师的无比青睐，颇有星火燎原之势。

作为一名合格的嵌入式工程师，面对新出现的技术，我们不是充耳不闻，而是要尽快吻合市场的需要，跟上技术的潮流。如今 STM32 的出现就是一种趋势，一种潮流，我们要做的就是搭上这趟快车，让自己的技术更有竞争力。

### 1.1.1 51 与 STM32 架构的区别

我们先普及一个概念，单片机（即 MCU）里面有什么。一个人最重要的是大脑，身体的各个部分都在大脑的指挥下工作。MCU 跟人体很像，简单来说是由一个最重要的内核加其他外设组成，内核就相当于人的大脑，外设就如人体的各个功能器官。

下面我们来简单介绍下 51 和 STM32 的结构。

#### 1.　51 系统结构

51 系统结构框图

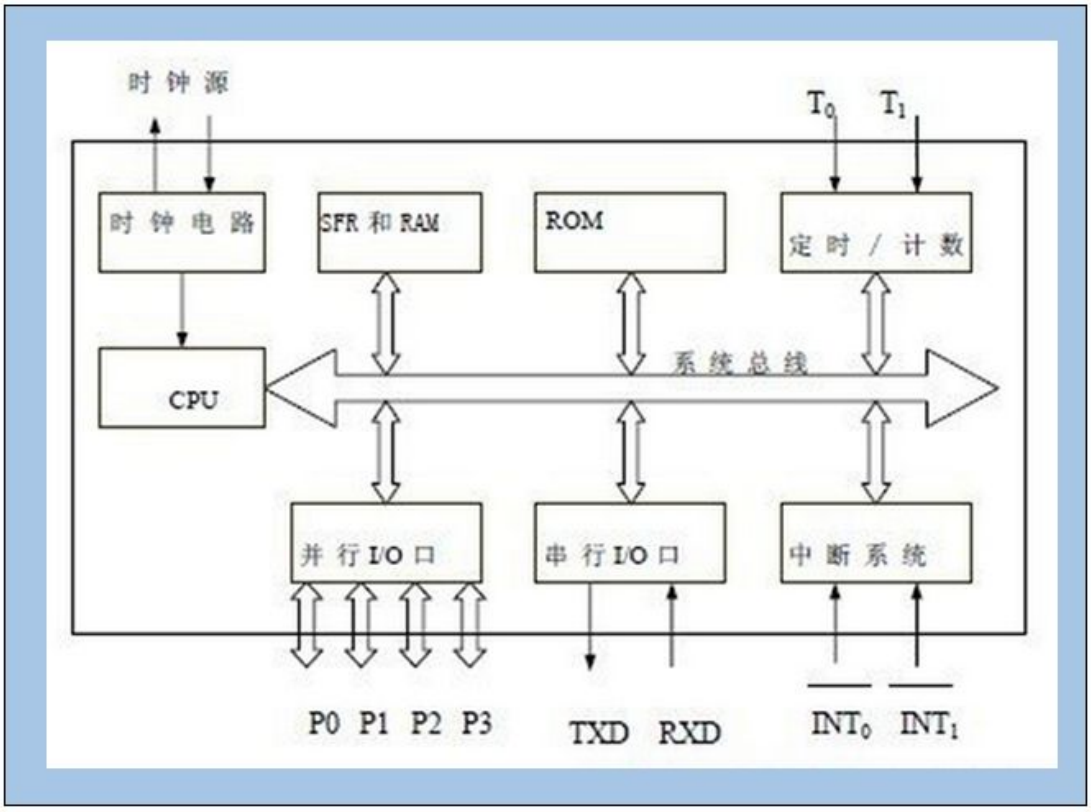



图 1 51 系统结构框图

我们说的 51 一般是指 51 系列的单片机，型号有很多，常见的有 STC89C51、AT89S51，其中国内用的最多的是 STC89C51/2，下面我们就以 STC89C51 来讲解，并以 51 简称。

### 内核

51 由一个 IP 核和片上外设组成，IP 核就是上图中的 CPU，片上外设就是上图中的：时钟电路、SFR 和 RAM、ROM、定时/计数器、并行 I/O 口、串行 I/O 口、中断系统。IP 核跟外设之间由系统总线连接，且是 8bit 的，速度有限。

51 内核是上个世纪 70 年代 intel 公司设计的，速度只有 12M，外设是 IC 厂商（STC）在内核的基础上添加的，不同的 IC 厂商会在内核上添加不同的外设，从而设计出各具特色的单片机。这里 intel 属于 IP 核厂商，STC 属于 IC 厂商。我们后面要讲的 STM32 也一样，ARM 属于 IP 核厂商，ARM 给 ST 授权，ST 公司在 Cortex-M3 内核的基础上设计出 STM32 单片机。

### 外设

我们在学习 51 的时候，关于内核部分接触的比较少，使用的最多的是片上外设，我们在编程的时候操作的也就是这些外设。

编程的时候操作的寄存器位于 SFR 和 RAM 这个部分，其中 SFR（特殊功能寄存器）占有 128 字节（实际上只用了 26 个字节，只有 26 个寄存器，其他都属于保留区），RAM 占有 128 字节，我们在程序中定义的变量就是放在 RAM 中。其中 SFR 和 RAM 在地址上是重合的，都是在 80~FF 这个地址区间，但在物理区间上是分开的，所以 51 的 RAM 是有 256 个字节。

编写好的程序是烧写到 ROM 区。剩下的外设都是我们非常熟悉的 IO 口，串口、定时器、中断这几个外设。

## 2. STM32 系统结构

STM32 系统结构框图

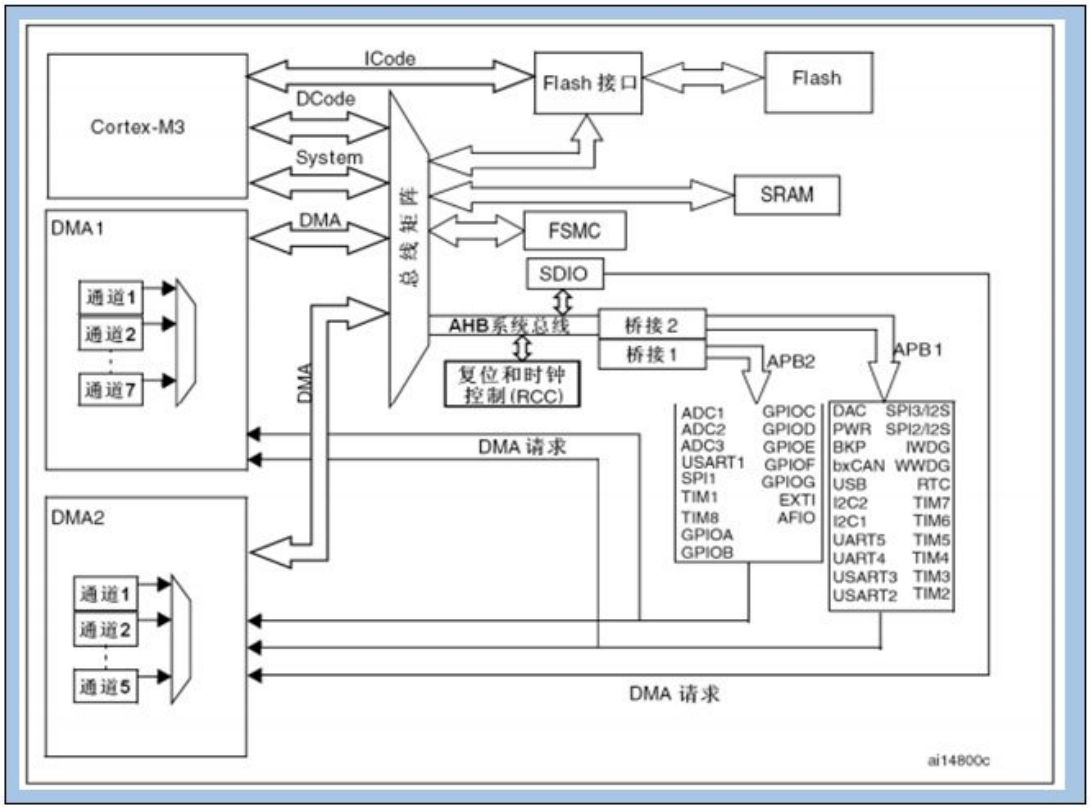



图 2 STM32 系统结构框图

### 内核

在系统结构上，STM32 和 51 都属于单片机，都是由内核和片上外设组成。只是 STM32 使用的 Cortex-M3 内核比 51 复杂得多，优秀得多，支持的外设也比 51 多得多，同时总线宽度也上升到 32bit，无论速度、功耗、外设都强与 51。

从结构框图上看，对比 51 内核只有一种总线，取指和取数共用。Cortex-M3 内部有若干个总线接口，以使 CM3 能同时取址和访内（访问内存），它们是：

指令存储区总线（两条）、系统总线、私有外设总线。有两条代码存储区总线负责对代码存储区（即 FLASH 外设）的访问，分别是 I-Code 总线和 D-Code 总线。

I-Code 用于取指，D-Code 用于查表等操作，它们按最佳执行速度进行优化。

系统总线（System）用于访问内存和外设，覆盖的区域包括 SRAM，片上外设，片外 RAM，片外扩展设备，以及系统级存储区的部分空间。

私有外设总线负责一部分私有外设的访问，主要就是访问调试组件。它们也在系统级存储区。

还有一个 MDA 总线，从字面上看，DMA 是 data memory access 的意思，是一种连接内核和外设的桥梁，它可以访问外设、内存，传输不受 CPU 的控制，并且是双向通信。简而言之，这个家伙就是一个速度很快的且不受老大控制的数据搬运工，这个在 51 里面是没有的。

#### 外设

从结构框图上看，STM32 比 51 的外设多得多，51 有的串口、定时器、IO 口等外设 STM32 都有。STM32 还多了很多特色外设：如 FSMC、SDIO、SPI、I2C 等，这些外设按照速度的不同，分别挂载到 AHB、APB2、APB1 这三条总线上。

### 3. 小结

从内核和外设这两大方面来比较，STM32 之于 51 就是一个升级版的单片机。它适应市场，引流潮流，在中低端的微控制器中流光溢彩。

## 1.2 学习方法的区别

学习 51 用寄存器，学习 STM32 用库。

以前我们在学习 51 的时候，用的是寄存器编程的方法，想要实现什么效果，直接往寄存器里面赋值，优点是直观，简单粗暴，知道自己具体干了啥，心里踏实。

直接操作寄存器之所以在 51 上可行，究其原因，我想有两点：

1、51 主频不高，资源有限，必须注重程序执行的效率，只能直接操作寄存器。关键的地方还得用汇编，不适合用固件库。

要知道当初我们学习 51 单片机的时候用的还是汇编，连现在的 C 编程都不是，就更别说什么库函数编程。

2、51 功能简单，寄存器不多。以国内普及最广的 STC89C52 为例，寄存器全部加起来不到 30 个。按照功能区分来记的话，可以把每个寄存器背的滚瓜烂熟，并且寄存器每一位的功能都可以记得住，在编程的时候做到了然于胸。

现在从 51 过度到 STM32 的学习，很多人还是喜欢沿用 51 的学习方法。接受不了库，在学习库的时候陷入迷糊之中，来回几个月下来，都不知道到底有没学会 STM32，因为在这一路的学习中都是在调用库函数，压根就没有操作过寄存器，心里面很不踏实。其实大家在调用库函数的时候心中难道就没有疑问，库的底层是怎么实现的？难道就没有勇气对库的底层一探究竟。可最后当我们开始跟踪库函数底层的时候，看到一堆的宏定义、结构体、指针、各种的文件包含，而且注释全部都是英文的，是不是又心生忌惮。鉴于此，我想用两个原因来总结下很多初学者畏惧库不愿意用库的原因。

1、C 语言知识点的欠缺

库在实现寄存器映像时使用的宏定义，强制类型转换，在定义寄存器时使用的结构体，在外设初始化函数时使用的指针，在组织头文件时使用的条件编译等 C 语言知识，在大学课程中很少涉及，大多数老师也基本是不讲。在一些简单的 51 单片机编程中又很少会用到这些知识。学单片机，做嵌入式开发其实 80%的工作都跟 C 语言编程相关，剩下的 20%的工作就是阅读各种数据手册，熟悉各种硬件外设。所以掌握这些基本的 C 语言知识，是嵌入式学习中一道迈不过去的坎，STM32 的库则给了我们一次提升 C 的机会。

凡是可以从书本中找到的，相信我们基本都可以学会，很多初学者并不是不够聪明或者勤奋，只是缺少方向性的指导罢了。对于这欠缺的知识点我们稍微花点时间就可以掌握，剩下的就是不断地实践调试。这里我为大家推荐一本 C 语言的书籍《C 和指针》。

2、程序架构设计思想的欠缺

这个比较难搞，很多 C 语言学习得挺好好的人，也比较难掌握。还好我们遇到了 STM32 的库，这给了我们一个学习和提升 C 语言绝佳的机会。库的整个架构是如何搭建起来的，代码上是如何如何一步一步写出来的：从寄存器映像开始，到寄存器的封装，然后到函数的编写，到每个外设函数对应的驱动文件，这里面涉及到了大量的条件编译，文件包含的思想，对应刚写过几行 51 单片机的初学者来说简直就是噩梦。但是，如果你把这一系列的关系弄明白了，那么对库的整个架构也了解的差不多了，以后你就不用嚷嚷着说要操作寄存器了。

如果你一开始不喜欢用库，对库开发很忌惮，那么请自问：是不是我的 C 语学得不够好。库是一种全新的学习方法，是一种潮流，我更把它看做是与 C 语言的又一次历练和提升。是否用库，只差你一个闪亮的回眸。

## 1.3 用寄存器点亮 LED

为了顺利过渡到库开发，在 STM32 编程的开始，我们对照 51 点亮一个 LED 的方法，给大家演示一下 STM32 如何用操作寄存器的方法点亮一个 LED，然后再慢慢讲解到底什么是库，让大家知道库跟寄存器的关系。

### 1.3.1 用 51 点亮一个 LED

在用 STM32 点亮一个 LED 之前，我们先来复习下用 51 如何点亮一个 LED。

硬件上我们假设 51 单片机的 P0 口的第 0 位接了一个 LED，负逻辑亮。如果我们要点亮这个 LED，代码上我们会这么写：

```
1 P0 = 0XFE;     // 总线操作点亮 LED
```

这时候我们就把 LED 点亮了，如果要关掉 LED，则是：

```
1 P0 = 0XFF;     // 总线操作关闭 LED
```

这里面我们用的是总线操作的方法，即是对 P0 口的 8 个 IO 同时操作，但起作用的只是 P0^0。

除了这种总线操作的方法，我们还学习过位操作，利用 51 编译器的关键字 sbit，我们可以定义一个位变量：

```
1 sbit LED = P0^0;
```

那么 LED = 0；就点亮了 LED，LED = 1;就关闭了 LED。为了让程序看起来见名知义，我们定义两个宏：

```
1 #define ON  0
2 #define OFF 1
```

点亮和关闭 LED 的代码就变成了：

```
1 LED = ON;      // 位操作点亮 LED
2 LED = OFF;     // 位操作关闭 LED
```

稍微整理下代码，整体的效果就是：

```
1 // 假设 51 单片机的 P0^0 口接 LED，负逻辑点亮
2
3 #define ON  0
4 #define OFF 1
5
6 sbit LED = P0^0;
7
8 void main(void)
```

```
 9 {
10     P0 = 0XFE;     // 总线操作点亮 LED
11     P0 = 0XFF;     // 总线操作关闭 LED
12
13     LED = ON;      // 位操作点亮 LED
14     LED = OFF;     // 位操作关闭 LED
15 }
```

上面总线和位操作的的方法，学过 51 的朋友是非常熟悉的，也很容易理解。那么我们再说一下大家容易忽略的几个知识点。

## 1. 什么是寄存器

在点亮 LED 的时候，我们都是用操作寄存器的方法来实现的，那大家是否想过，这个寄存器到底是什么？为什么我们可以直接操作 P0 口？

解答上面的问题之前，我们先简单介绍下 51 单片机的主要组成部分，这对我们学习其他单片机也有好处。

我们以国内的 STC89C51 为例，该单片机主要由 51 内核、外设 IP、和总线这三大部分组成。内核是由 Intel 公司生产的，外设 IP 就是 STC 公司在内核的基础上添加的诸如定时器、串口、IO 口等这些东西，总线就是用来连接内核和外设的接口单元。Intel 在这里属于 IP 核设计公司，STC 属于 IC 设计公司。世界上能设计 IP 核的公司屈指可数。我们非常熟悉的 ARM 公司就属于 IP 核设计公司，ARM 给其他公司授权，其他 IC 公司就在 ARM 内核上设计出各具特色的 MCU，我们后面要学习的 STM32 就是属于一中基于 ARM 内核的 MCU。

寄存器则是内置于各个 IP 外设中，是一种用于配置外设功能的存储器，就是一种内存，并且有想对应的地址。学过 C 语言我们就知道，要操作这些内存就可以使用 C 语言中的指针，通过寻址的方式来操作这些具有特殊功能的内存—寄存器。比如 P0 口对应的地址是 0X80，那么我们要修改 0X80 这个地址对应的内存的内容的话，按照常理可以这样操作：

```
1 *(*0X80) = 0XFE;    // 点亮 LED
```

可当我们编译的时候，编译器会报错，在 51 里面只能通过 SFR 和 SBIT 这两个关键字来实现寄存器映像，不能直接操作寄存器对应的地址，这是 51 相较于 STM32 不同的地方。

51 单片机的这些寄存器位于地址 80H~FFH 中，对应着 128 个地址，但不是每个地址都是有效的，51 系列的单片机有 21 个，52 系列的则有 26 个，其他的都是保留区。

表2 AT89C52 SFR 映象及复位状态

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0F8H | | | | | | | | 0FFH |
| 0F0H | B<br>00000000 | | | | | | | 0F7H |
| 0E8H | | | | | | | | 0EFH |
| 0E0H | ACC<br>00000000 | | | | | | | 0E7H |
| 0D8H | | | | | | | | 0DFH |
| 0D0H | PSW<br>00000000 | | | | | | | 0D7H |
| 0C8H | T2CON<br>00000000 | T2MOD<br>XXXXXX00 | RCAP2L<br>00000000 | RCAP2H<br>00000000 | TL2<br>00000000 | TH2<br>00000000 | | 0CFH |
| 0C0H | | | | | | | | 0C7H |
| 0B8H | IP<br>XX000000 | | | | | | | 0BFH |
| 0B0H | P3<br>11111111 | | | | | | | 0B7H |
| 0A8H | IE<br>0X000000 | | | | | | | 0AFH |
| 0A0H | P2<br>11111111 | | | | | | | 0A7H |
| 98H | SCON<br>00000000 | SBUF<br>XXXXXXXX | | | | | | 9FH |
| 90H | P1<br>11111111 | | | | | | | 97H |
| 88H | TCON<br>00000000 | TMOD<br>00000000 | TL0<br>00000000 | TL1<br>00000000 | TH0<br>00000000 | TH1<br>00000000 | | 8FH |
| 80H | P0<br>11111111 | SP<br>00000111 | DPL<br>00000000 | DPH<br>00000000 | | | | PCON<br>0XXX0000 87H |

图 3 51 寄存器映射

## 2. 寄存器映射

实际上我们在编程的时候并不是通过指针来操作寄存器的，而是直接给 P0、P1 这些端口寄存器赋值。那么这些外设资源是如何与地址建立一一对应的关系（寄存器映射定义），这得益与 51 特有的两个关键字：SFR 和 sbit，其他单片机没有，只能用其他的方式来实现寄存器映射。这两个关键字帮我们实现了所有寄存器的定义，所以我们才可以像操作普通变量一个来操作寄存器。其实我们一开始提到的点亮 LED 的代码，全貌应该是这样的：

```
1 sfr P0    = 0x80;   // 寄存器定义
2 P0 = 0XFE;          // 总线操作点亮 LED
```

为了方便起见，我们可以把寄存器映射全部写好封装在一个头文件里面，不用每用一个寄存器就定义一次。其实这方面的工作不用我们做，我们在编程的时候都会在开始的地方添加一个头文件：

```
1 #include <reg51.h>
```

这个头文件已经实现了全部寄存器的定义，该文件是 keil 自带，在安装目录：Keil\C51\INC 下可以找到。这个文件实现了字节寄存器和位寄存器的定义。

```
/*--------------------------------------------------------------
REG51.H

Header file for generic 80C51 and 80C31 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
--------------------------------------------------------------*/

#ifndef __REG51_H__
#define __REG51_H__

/*  BYTE Register  */
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;


/*  BIT Register  */
/*  PSW   */
sbit CY   = 0xD7;
sbit AC   = 0xD6;
sbit F0   = 0xD5;
sbit RS1  = 0xD4;
sbit RS0  = 0xD3;
sbit OV   = 0xD2;
sbit P    = 0xD0;

/*  TCON  */
sbit TF1  = 0x8F;
sbit TR1  = 0x8E;
sbit TF0  = 0x8D;
sbit TR0  = 0x8C;
sbit IE1  = 0x8B;
sbit IT1  = 0x8A;
sbit IE0  = 0x89;
sbit IT0  = 0x88;

/*  IE   */
sbit EA   = 0xAF;
sbit ES   = 0xAC;
sbit ET1  = 0xAB;
sbit EX1  = 0xAA;
sbit ET0  = 0xA9;
sbit EX0  = 0xA8;

```

```
64 /*  IP   */
65 sbit PS   = 0xBC;
66 sbit PT1  = 0xBB;
67 sbit PX1  = 0xBA;
68 sbit PT0  = 0xB9;
69 sbit PX0  = 0xB8;
70
71 /*  P3  */
72 sbit RD   = 0xB7;
73 sbit WR   = 0xB6;
74 sbit T1   = 0xB5;
75 sbit T0   = 0xB4;
76 sbit INT1 = 0xB3;
77 sbit INT0 = 0xB2;
78 sbit TXD  = 0xB1;
79 sbit RXD  = 0xB0;
80
81 /*  SCON  */
82 sbit SM0  = 0x9F;
83 sbit SM1  = 0x9E;
84 sbit SM2  = 0x9D;
85 sbit REN  = 0x9C;
86 sbit TB8  = 0x9B;
87 sbit RB8  = 0x9A;
88 sbit TI   = 0x99;
89 sbit RI   = 0x98;
90
91 #endif
92
```

## 3. 启动文件—STARTUP.A51

还有一个就是启动代码，这个也是很多初学者容易忽略的地方，对于这部分我们主要总结下它的功能，不详解讲解里面的代码。

单片机在上电复位后，首先执行的是启动文件—STARTUP.A51，而不是我们通常看到的 main 函数。我们新建 51 工程的时候会有一个提示：是否拷贝启动代码到当前的工程，我们一般选择是。

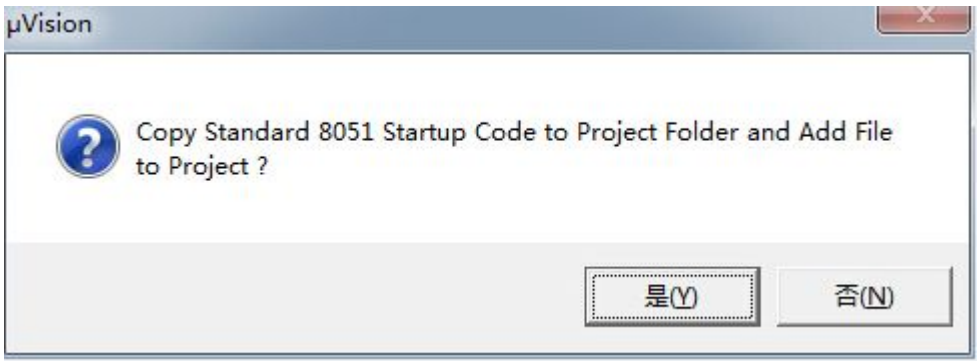



图 4 是否添加启动代码

启动代码用汇编语言编写，主要实现了以下功能：清除内部数据存储器、清除外部数据存储器、清除外部页储存器、初始化 small 模式下的可重入栈和指针、初始化 large 模式下可重入栈和指针、初始化 compact 模式下的可重入栈和指针、初始化 8051 硬件栈指针、

传递初始化全局变量的控制命令或者在没有初始化全局变量时给 main 函数传递命令。然后程序就跳转到 main 函数，来到我们熟知的 C 世界。

## 4. 总结

在讲解用 51 点亮 LED 的时候，我们补充了什么是寄存器、寄存器映射、启动代码这三部分的内容，这三部分内容本来是放到 STM32 里面讲解的，但考虑到大家已经有 51 的基础，并且对 51 比较熟悉，那我再添加点内容，大家自然没有那么抗拒，并且可以根据上面讲的内容亲自实践，学习得也会更深入。那当我再在 STM32 讲解这几个内容的时候，大家就会对比着学习，对 STM32 也就没有那么忌惮。

### 1.3.2 用 STM32 点亮一个 LED

对比着 51 点亮 LED 的方法，我们先用操作寄存器的方法用 STM32 点亮一个 LED，然后再一步步完善代码，构建最简单的库函数，让我们知道库是怎么建立起来的。

在写代码之前，我们先建一个工程。大家要注意的是，虽然 51 跟 STM32 用的都是 keil，但是针对的 MCU 是不一样，软件在安装的时候要安装在不同的目录且不能安装在英文目录，不然会起冲突。我们这里用的是 keil5，MDK5.15 版本。

## 1. 新建工程

用 KEIL5 新建一个工程，把工程放在一个事先建好的文件夹内，工程命名为 REG 后保存。然后在工程目录下添加启动文件：startup_stm32f10x_hd.s，该文件可以从 KEIL5 安装目录找到，也可以从 ST 库里面找到，然后把启动文件添加到工程里面。

## 2. 启动文件—startup_stm32f10x_hd.s

启动文件由汇编语言编写，具体功能跟 51 里面的启动文件：STARTUP.A51 差不多。

STM32 的启动文件主要实现了：1、设置初始 SP 。2、设置初始 PC=Reset_Handler 3、设置向量表入口地址，并初始化向量表。4、调用库函数 SystemInit，把系统时钟配置成 72M，SystemInit 在库文件 system_stm32f10.c 定义。5、跳转到标号_mian，最终来到 C 的世界。这里我们先去除繁枝细节，挑重点的讲，主要理解第四和第五点，在启动文件的 147~155 行，是复位处理函数，代码如下：

```
1 ;Reset handler
2 Reset_Handler    PROC
3 EXPORT  Reset_Handler             [WEAK]
4 IMPORT  __main
5 IMPORT  SystemInit
6 LDR     R0, =SystemInit
7 BLX     R0
```

```
 8 LDR      R0, =__main
 9 BX       R0
10 ENDP
```

这里我们简单介绍下这 10 行代码。

第一行是程序注释，在汇编里面注释用的是“;”，跟 C 语言不一样。

第二行是定义了一个子程序：Reset_Handler。PROC 是子程序定义伪指令。一般用法为：

```
1 子程序名 PROC NEAR ( 或 FAR )
2 ……
3 ret
4 子程序名 ENDP
```

其中 NEAR 和 FAR 是属性词。NEAR 属性(段内近调用): 调用程序和子程序在同一代码段中,只能被相同代码段的其他程序调用。FAR 属性(段间远调用): 调用程序和子程序不在同一代码段中,可以被相同或不同代码段的程序调用。

第三行 EXPORT 表示 Reset_Handler 这个子程序可供其他模块调用。

关键字[WEAK] 表示弱定义，如果编译器发现在别处定义了同名的函数，则在链接时用别处的地址进行链接，如果其它地方没有定义，编译器也不报错，以此处地址进行链接。

第四行和第五行 IMPORT 说明 SystemInit 和__main 这两个标号在其他文件，在链接的时候需要到其他文件去寻找。

SystemInit 在库文件 system_stm32f10x.c 实现，用来初始化 STM32 的一系列时钟，把系统时钟设置为 72MHZ。STM32 的时钟比 51 单片机复杂，需要经过一系列的配置才能达到稳定运行的状态。

__main 其实不是我们定义的，当编译器编译时，只要遇到这个标号就会定义这个函数，该函数的主要功能是：负责初始化栈、堆，配置系统环境，并在最后跳转到用户自定义的 main 函数，从此来到 C 的世界。

第六行把 SystemInit 的地址加载到寄存器 R0。

第七行程序跳转到 R0 中的地址执行程序，之后系统的时钟就被设置成 72MHZ。

第八行把_main 的地址加载到寄存器 R0。

第九行程序跳转到 R0 中的地址执行程序，执行完毕之后就去到我们熟知的 C 世界。

第十行表示子程序的结束。

总结下就是，Reset_Handler 这个函数执行了两个函数调用，一个是 SystemInit，把系统时钟设置成 72M，令一个是__main，初始化好系统环境，最终调用 C 的 main，从此去到 C 的世界。

等下我们点亮 LED 的时候采用最简单的方法，直接使用内部的 LSI 时钟（8MHZ）作为主时钟即可，不使用外部时钟 LSE。

__main 函数由编译器生成，负责初始化栈、堆等，并在最后跳转到用户自定义的 main()函数，来到 C 的世界。

## 3. 新建 main.c

用记事本新建一个 main.c 文件放到工程目录下，然后把 main.c 添加到工程中。

现在我们就可以开始编写程序了，我们先编写一个 main 函数，里面啥都没有，暂时为空。这时跟编写 51 程序时是不是很像。

```
1 int main(void)
2 {
3
4 }
```

现在我们可以编译看看，看看有啥现象。

这时候出现如下错误：

```
Undefined symbol SystemInit (referred from startup_stm32f10x_hd.o).
```

错误提示说 SystemInit 没有定义。从分析启动文件时我们知道，Reset_Handler 调用了该函数用来初始化系统时钟，而该函数是在库文件 system_stm32f10x.c 中实现的。我们重新写一个这样的函数也可以，把功能完整实现一遍，但是为了简单起见，我们在 main 文件里面定义一个 SystemInit 空函数，为的是骗过编译器，把这个错误去掉。关于配置系统时钟我们在后面再写简单的代码。

```
1 void SystemInit(void)
2 {
3
4 }
```

这时我们再编译就没有错了，完美解决。还有一个方法就是在启动文件中把有关 SystemInit 的代码注释掉也可以，代码如下所示：

```
1 ;Reset handler
2 Reset_Handler    PROC
3 EXPORT  Reset_Handler             [WEAK]
4 IMPORT   __main
5 ;IMPORT  SystemInit
6 ;LDR     R0, =SystemInit
7 BLX      R0
8 LDR      R0, =__main
9 BX       R0
10 ENDP
```

# 4. 控制 IO 口

下面我们从三个方面来讲解 STM32 的 IO 在控制 LED 时跟 51 的区别。有关 STM32 的 IO 的寄存器介绍，我们可以看《STM32 中文参考手册》的第八章即可，下面涉及到的 IO 寄存器均来自这一章的第二小节：8.2 GPIO 寄存器描述

### 电平控制

51 单片机的 IO 口如果要输出 1 和 0，可以直接赋值，不用控制其他寄存器。

而 STM32 的 IO 口比较复杂，如果要输出 1 和 0，则要通过控制：端口输出数据寄存器 ODR 来实现，ODR 是：Output data register 的简写，在 STM32 里面，其寄存器的命名名称都是英文的简写，很容易记住。从手册上我们知道 ODR 是一个 32 位的寄存器，低 16 位有效，高 16 位保留。低 16 位对应着 IO0~IO16，只要往相应的位置写入 0 或者 1 就可以输出低或者高电平。

PB0 输出低电平，代码如下：

```
1 GPIOB_ODR = 0<<0;
```

这时候编译，我们会发现有个错误，说 GPIOB_ODR 没有定义，不过我们确实没有定义。在 51 单片机中，我们可以直接往 P0 口赋值，那是因为在 reg51.h 这个头文件中实现了 P0 口这个寄存器的映像，用的是 51 特有的关键字 SFR 来定义的。

STM32 跟 51 不一样，没有 SFR，只能用其他的方式来实现寄存器映像。因为寄存器实际上就是具有特殊功能的内存，那么我们可以通过宏定义来实现寄存器映像，其实 ST 的库函数中用的也是这种方法。

从手册中我们看到 ODR 寄存器的地址偏移是：0CH，这个偏移地址是基于端口的起始地址而言的。在 STM32 中，每个外设都有一个起始地址，叫做外设基地址，外设的寄存器就以这个基地址为标准按照顺序排列，跟结构体里面的成员差不多。

在手册中的第二章：存储器和总线构架 的 2.3：存储器映像 小节中可以查看到所有外设的基地址，如下：

表1 寄存器组起始地址

| 起始地址 | 外设 | 总线 | 寄存器映像 |
|---|---|---|---|
| 0x5000 0000 – 0x5003 FFFF | USB OTG 全速 | AHB | 参见26.14.6节 |
| 0x4003 0000 – 0x4FFF FFFF | 保留 | | |
| 0x4002 8000 – 0x4002 9FFF | 以太网 | | 参见27.8.5节 |
| 0x4002 3400 - 0x4002 3FFF | 保留 | AHB | |
| 0x4002 3000 - 0x4002 33FF | CRC | | 参见3.4.4节 |
| 0x4002 2000 - 0x4002 23FF | 闪存存储器接口 | | |
| 0x4002 1400 - 0x4002 1FFF | 保留 | | |
| 0x4002 1000 - 0x4002 13FF | 复位和时钟控制(RCC) | | 参见6.3.11节 |
| 0x4002 0800 - 0x4002 0FFF | 保留 | | |
| 0x4002 0400 - 0x4002 07FF | DMA2 | | 参见10.4.7节 |
| 0x4002 0000 - 0x4002 03FF | DMA1 | | 参见10.4.7节 |
| 0x4001 8400 - 0x4001 7FFF | 保留 | | |
| 0x4001 8000 - 0x4001 83FF | SDIO | | 参见20.9.16节 |
| 0x4001 4000 - 0x4001 7FFF | 保留 | APB2 | |
| 0x4001 3C00 - 0x4001 3FFF | ADC3 | | 参见11.12.15节 |
| 0x4001 3800 - 0x4001 3BFF | USART1 | | 参见25.6.8节 |
| 0x4001 3400 - 0x4001 37FF | TIM8定时器 | | 参见13.4.21节 |
| 0x4001 3000 - 0x4001 33FF | SPI1 | | 参见23.5节 |
| 0x4001 2C00 - 0x4001 2FFF | TIM1定时器 | | 参见13.4.21节 |
| 0x4001 2800 - 0x4001 2BFF | ADC2 | | 参见11.12.15节 |
| 0x4001 2400 - 0x4001 27FF | ADC1 | | 参见11.12.15节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口G | | 参见8.5节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口F | | 参见8.5节 |
| 0x4001 1800 - 0x4001 1BFF | GPIO端口E | | 参见8.5节 |
| 0x4001 1400 - 0x4001 17FF | GPIO端口D | | 参见8.5节 |
| 0x4001 1000 - 0x4001 13FF | GPIO端口C | | 参见8.5节 |
| 0X4001 0C00 - 0x4001 0FFF | GPIO端口B | | 参见8.5节 |
| 0x4001 0800 - 0x4001 0BFF | GPIO端口A | | 参见8.5节 |
| 0x4001 0400 - 0x4001 07FF | EXTI | | 参见9.3.7节 |
| 0x4001 0000 - 0x4001 03FF | AFIO | | 参见8.5节 |
| 0x4000 7800 - 0x4000FFFF | 保留 | APB1 | |
| 0x4000 7400 - 0x4000 77FF | DAC | | 参见12.5.14节 |
| 0x4000 7000 - 0x4000 73FF | 电源控制(PWR) | | 参见4.4.3节 |
| 0x4000 6C00 - 0x4000 6FFF | 后备寄存器(BKP) | | 参见5.4.5节 |
| 0x4000 6800 - 0x4000 6BFF | bxCAN2 | | 参见22.9.5节 |
| 0x4000 6400 - 0x4000 67FF | bxCAN1 | | 参见22.9.5节 |
| 0x4000 6000(1) - 0x4000 63FF | USB/CAN共享的512字节SRAM | | |

图 5 STM32 寄存器组起始地址

其中 GPIOB 的起始地址是：0X4001 0C00，这样就可以算出 GPIOB_ODR 寄存器的地址是：0X4001 0C00 + 0X0C = 0X4001 0C0C。现在我们就可以定义 GPIOB_ODR 这个寄存器了，代码如下：

```
1 #define GPIOB_ODR     *(volatile unsigned long *)0x40010C0C
```

有了这个寄存器定义，我们就可以直接操作 GPIOB_ODR 了。

### 方向控制

虽然配置了 ODR 寄存器，但是这个时候还不能点亮 LED，因为 STM32 的 IO 口还要配置方向，这个由端口配置寄存器来控制。端口配置寄存器分为高低两个，每 4bit 控制一个 IO 口，所以端口配置低寄存器：CRL 控制这 IO 口的低 8 位，端口配置高寄存器：CRH 控制这 IO 口的高 8bit。在 4 位一组的控制位中，CNFy[1:0] 用来控制端口的输入输出，MODEy[1:0]用来控制输出模式的速率，即输出时，IO 电平翻转的速度。

输入有三种模式，输出有 4 中模式，我们在控制 LED 的时候选择通用推挽输出。

输出速率有三种模式：2M、10M、50M，这里我们选择 2M。

同 GPIOB_ODR 一样，我们也可以算出 GPIO_CRL 的地址为：0x40010C00。那么设置 PB0 为通用推挽输出，输出速率为 2M 的代码则如下所示：

```
1 #define GPIOB_CRL     *(volatile unsigned long *)0x40010C00
2
3 // 配置 PB0 为通用推挽输出，输出速率为 2M
4 GPIOB_CRL = (2<<0) | (0<<2);
```

### 时钟控制

当我们设置了 IO 口的方向，并在相应的输出寄存器里面输入了值的时候，以为现在总算可以点亮 LED 了吧，其实还差最后一步。

STM32 外设很多，为了降低功耗，每个外设都对应着一个时钟，在系统复位的时候这些时钟都是被关闭的，如果想要外设工作，必须把相应的时钟打开。

STM32 的所有外设的时钟由一个专门的外设来管理，叫 RCC（reset and clock control），RCC 在 STM32 中文参考手册的第六章。

STM32 的外设因为速率的不同，分别挂载到三条总系上：AHB、APB2、APB1，APB 为高速总线，APB2 次之，APB1 再次之。所以的 IO 口都挂载到 APB2 总线上，属于高速外设。时钟由 APB2 外设时钟使能寄存器(RCC_APB2ENR)来控制，其中 PB 端口的时钟由该寄存器的位 3 写 1 使能。

同 ODR 和 CRL，我们可以算出 RCC_APB2ENR 的地址为：0x40021018。那么使能 PB 口的时钟代码则如下所示：

```
1 #define RCC_APB2ENR   *(volatile unsigned long *)0x40021018
2
3 // 开启端口 B 时钟
4 RCC_APB2ENR |= 1<<3;
```

如果你足够细心，你会发现我们虽然开了端口时钟，那这个时钟到底是多大？时钟到底是从哪里来的？

如果我们用的是库，那么有个库函数 SystemInit，会帮我们把系统时钟设置成 72M。现在我们没有使用库，那现在时钟是多少？答案是 8M，当外部 HSE 没有开启或者出现故障的时候，系统时钟由内部低速时钟 LSI 提供，现在我们是没有开启 HSE，所以系统默认的时钟是 LSI=8M。至于更深入的细节我们在后面的 RCC 时钟树中再详细分析。如果你想自己先尝鲜，那么看 RCC 外设中的：时钟控制寄存器(RCC_CR)和时钟配置寄存器(RCC_CFGR)这两个寄存器即可。

### 水到渠成

控制了电平，配置了方向，开启了时钟，经过这三步，我们总算可以控制一个 LED 了。比起 51 直接输出电平，控制 STM32 的 IO 多了两步：即配置方向可开启时钟。比起 AVR 和 PIC 这两种单片机则多了开启时钟这一步。

现在我们完整组织下用 STM32 控制一个 LED 的代码：

```
#define RCC_APB2ENR    *(volatile unsigned long *)0x40021018
#define GPIOB_CRL      *(volatile unsigned long *)0x40010C00
#define GPIOB_ODR      *(volatile unsigned long *)0x40010C0C

int main(void)
{
    // 开启端口 B 的时钟
    RCC_APB2ENR |= 1<<3;

    // 配置 PB0 为通用推挽输出模式，速率为 2M
    GPIOB_CRL = (2<<0) | (0<<2);

    // PB0 输出低电平，点亮 LED
    GPIOB_ODR = 0<<0;
}

void SystemInit(void)
{
}
```

很多人说学习 STM32 很难，一堆的寄存器，不知道怎么操作，特别是那些刚学习完 51 的朋友，不知道怎么过度。这里我们对比了 51 的编程方法，写了个简单的用 STM32 寄存器点亮 LED 的方法，希望可以起到抛砖引玉的作用。

## 1.4 再接再厉—构建库的雏形

学习 STM32 存在着一个用寄存器好还是用库好的争议点，就好比编程是用汇编好还是用 C 好一样。其实孰优孰劣，市场自有定论，用户群说明一切。

虽然我们上面用寄存器点亮了 LED，乍看一下好像代码也很简单，但是我们别侥幸以后就可以一直用寄存器开发。在用寄存器点亮 LED 的时候，我们是否发现 STM32 的寄存器都是 32 位的，在配置的时候非常容易出错，而且代码还很不好理解。所以学习 TM32 最好的方法是用库，然后在库的基础上了解底层，看遍所有寄存器。

但是很多人对库还是很忌惮，因为一开始用库的时候有很多代码，很多文件，不知道如何入手。不知道你是否认同这么一句话：一切的恐惧都来源于认知的空缺。我们对库忌惮那是因为我们不知道什么是库，不知道库是怎么实现的。

接下来，我们在寄存器点亮 LED 的代码上继续完善，把代码一层层封装，实现库的最初的雏形，相信经过这一步的学习后，你会对库的运用做到游刃有余。这里我们只讲关于 GPIO 库，其他外设的我们直接参考库学习即可，不必自己写。

### 1.4.1 定义外设寄存器结构体

上面我们在操作寄存器的时候，操作的是寄存器的绝对地址，如果每个寄存器都这样操作，那将非常麻烦。

我们考虑到外设寄存器的地址都是基于外设基地址的偏移地址，都是在外设基地址上逐个连续递增的，每个寄存器占 32 个或者 16 个字节，这种方式跟结构体里面的成员类似。所以我们可以定义一种外设结构体，结构体的地址等于外设的基地址，结构体的成员等于寄存器，成员的排列顺序跟寄存器的顺序一样。这样我们操作寄存器的时候就不用每次都找到绝对地址，只要知道外设的基地址就可以操作外设的全部寄存器，即操作结构体的成员即可。

下面我们先定义一个 GPIO 寄存器结构体，结构体里面的成员是 GPIO 的寄存器，成员的顺序按照寄存器的偏移地址从低到高排列，成员类型跟寄存器类型一样。

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

在《STM32 中文参考手册》8.2 寄存器描述章节，我们可以找到结构体里面的 7 个寄存器描述。在点亮 LED 的时候我们只用了 CRL 和 ODR 这两个寄存器，至于其他寄存器的功能大家可以自行看手册了解。

在 GPIO 结构体里面我们用了两个数据类型，一个是 uint32_t，表示无符号的 32 位整型，因为 GPIO 的寄存器都是 32 位的。这个类型声明在标准头文件 stdint.h 里面，我们在程序上只要包含这个头文件即可。

另外一个是__IO，这个是我们自己定义的，原型是 volatile，作用就是告诉编译器不要因优化而省略此指令，必须每次都直接读写其值，这样就能确保每次读或者写寄存器都真正执行到位。

关于这两个数据类型，我们添加如下代码：

```
1 #include <stdint.h>
2 #define     __IO    volatile
```

### 1.4.2 外设声明

现在 GPIO 寄存器结构体已经定义好了，STM32F1 系列的 GPIO 端口分 A~G，即 GPIOA、GPIOB。。。。。。GPIOG。每个端口都含有 GPIO_TypeDef 结构体里面的寄存器，我们可以根据各个端口的基地址把 GPIO 的各个端口定义成一个 GPIO_TypeDef 类型的指针，然后我们就可以根据端口名（实际上现在是结构体指针了）来操作各个端口的寄存器，代码实现如下：

```
1 #define GPIOA                ((GPIO_TypeDef *) 0X4001 0800)
2 #define GPIOB                ((GPIO_TypeDef *) 0X4001 0C00)
3 #define GPIOC                ((GPIO_TypeDef *) 0X4001 1000)
4 #define GPIOD                ((GPIO_TypeDef *) 0X4001 1400)
5 #define GPIOE                ((GPIO_TypeDef *) 0X4001 1800)
6 #define GPIOF                ((GPIO_TypeDef *) 0X4001 1C00)
7 #define GPIOG                ((GPIO_TypeDef *) 0X4001 2000)
```

对于其他外设我们也可以这样把外设的名字定义成一个外设寄存器结构体类型的指针，这里我们只讲 GPIO。

对于每个 GPIO 的基地址我们可以从《STM32 中文参考手册》2.3 小节：存储器映像中找到，如下所示：

| APB2总线 | | | |
|---|---|---|---|
| **起始地址** | **外设** | **总线** | **寄存器映像** |
| 0x4001 4000 - 0x4001 7FFF | 保留 | | |
| 0x4001 3C00 - 0x4001 3FFF | ADC3 | | 参见11.12.15节 |
| 0x4001 3800 - 0x4001 3BFF | USART1 | | 参见25.6.8节 |
| 0x4001 3400 - 0x4001 37FF | TIM8定时器 | | 参见13.4.21节 |
| 0x4001 3000 - 0x4001 33FF | SPI1 | | 参见23.5节 |
| 0x4001 2C00 - 0x4001 2FFF | TIM1定时器 | | 参见13.4.21节 |
| 0x4001 2800 - 0x4001 2BFF | ADC2 | | 参见11.12.15节 |
| 0x4001 2400 - 0x4001 27FF | ADC1 | | 参见11.12.15节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口G | APB2 | 参见8.5节 |
| 0x4001 2000 - 0x4001 23FF | GPIO端口F | | 参见8.5节 |
| 0x4001 1800 - 0x4001 1BFF | GPIO端口E | | 参见8.5节 |
| 0x4001 1400 - 0x4001 17FF | GPIO端口D | | 参见8.5节 |
| 0x4001 1000 - 0x4001 13FF | GPIO端口C | | 参见8.5节 |
| 0X4001 0C00 - 0x4001 0FFF | GPIO端口B | | 参见8.5节 |
| 0x4001 0800 - 0x4001 0BFF | GPIO端口A | | 参见8.5节 |
| 0x4001 0400 - 0x4001 07FF | EXTI | | 参见9.3.7节 |
| 0x4001 0000 - 0x4001 03FF | AFIO | | 参见8.5节 |

图 6 APB2 总线外设寄存器起始地址

### 1.4.3 外设内存映射

讲到基地址的时候我们再引人一个知识点：Cortex-M3 存储器系统，这个知识点在《Cortex-M3 权威指南》第 5 章里面讲到。CM3 的地址空间是 4GB，如下图所示：

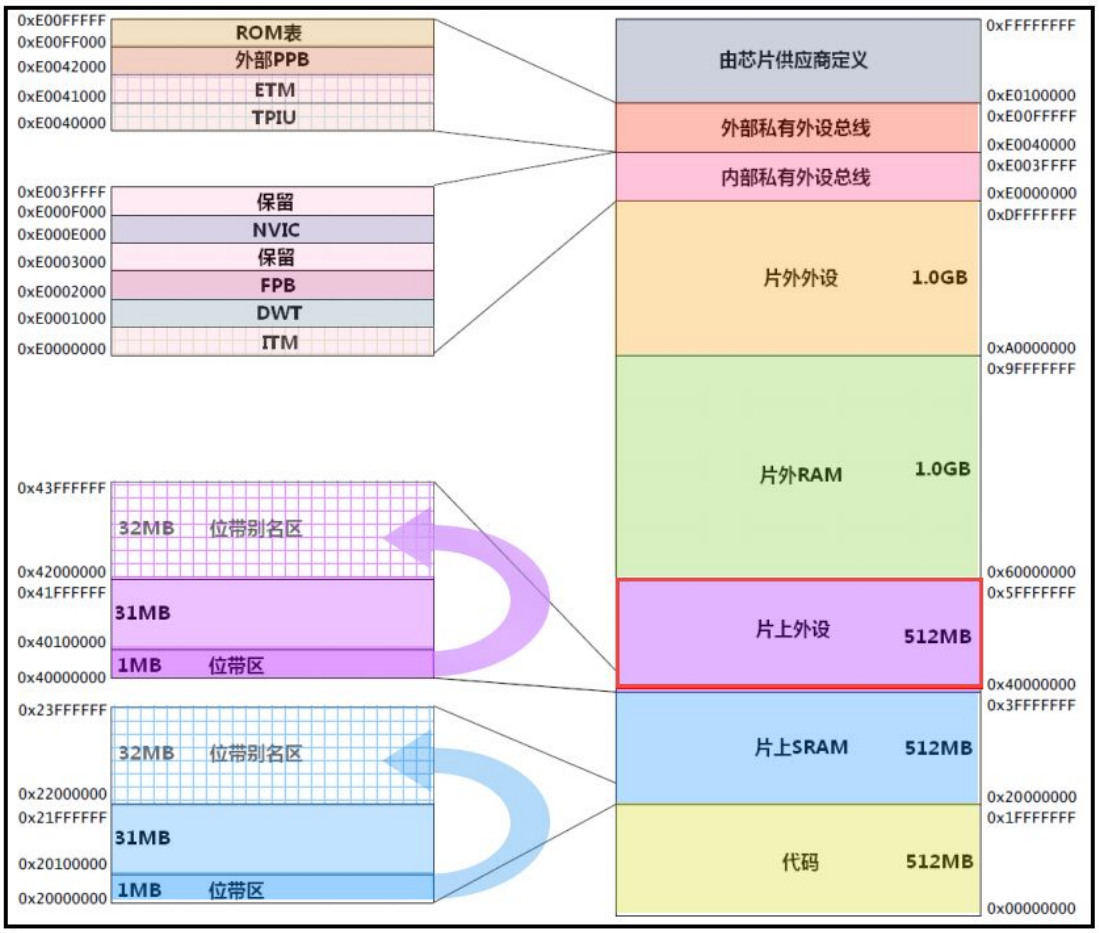



图 7 CM3 内存映射

我们这里要讲的是片上外设，就是我们所说的寄存器的根据地，其大小总共有 512MB，512MB 是其极限空间，并不是每个单片机都用得完，实际上各个 MCU 厂商都只是用了一部分而已。STM32F1 系列用到了：0x4000 0000 ~0x5003 FFFF。

## 1. APB1、APB2、AHB 总线基地址

现在我们说的 STM32 的寄存器就是位于这个区域，这里面 ST 设计了三条总线：AHB、APB2 和 APB1，其中 AHB 和 APB2 是高速总线，APB1 是低速总线。不同的外设根据速度不同分别挂载到这三条总线上。从下往上依次是：APB1、APB2、AHB，每个总线对应的地址分别是：APB1：0x40000000，APB2：0x4001 0000，AHB：0x4001 8000。

这三条总线的基地址我们是从《STM32 中文参考手册》2.3 小节—存储器映像 得到的：APB1 的基地址是 TIM2 定时器的起始地址，APB2 的基地址是 AFIO 的起始地址，AHB 的基地址是 SDIO 的起始地址。

其中 APB1 地址又叫做外设基地址，是所有外设的基地址，叫做 PERIPH_BASE。

现在我们把这三条总线地址用宏定义出来，以后我们在定义其他外设基地址的时候，只需要在这三条总线的基址上加上偏移地址即可，代码如下：

```
1 #define PERIPH_BASE           ((uint32_t)0x40000000)
2 #define APB1PERIPH_BASE       PERIPH_BASE
3 #define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
4 #define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
```

## 2. GPIO 端口基地址

因为 GPIO 挂载到 APB2 总线上，那么现在我们就可以根据 APB2 的基址算出各个 GPIO 端口的基地址，用宏定义实现代码如下：

```
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)
5 #define GPIOE_BASE            (APB2PERIPH_BASE + 0x1800)
6 #define GPIOF_BASE            (APB2PERIPH_BASE + 0x1C00)
7 #define GPIOG_BASE            (APB2PERIPH_BASE + 0x2000)
```

现在我们把上面的代码稍微整理下，如下：

```
1 typedef struct {
2     __IO uint32_t CRL;
3     __IO uint32_t CRH;
4     __IO uint32_t IDR;
5     __IO uint32_t ODR;
6     __IO uint32_t BSRR;
7     __IO uint32_t BRR;
8     __IO uint32_t LCKR;
9 } GPIO_TypeDef;
10
11 #define PERIPH_BASE           ((uint32_t)0x40000000)
12 #define APB1PERIPH_BASE       PERIPH_BASE
13 #define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
14 #define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
15
16 #define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
17 #define GPIOB_BASE            (APB2PERIPH_BASE + 0x0C00)
18 #define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)
19 #define GPIOD_BASE            (APB2PERIPH_BASE + 0x1400)
20 #define GPIOE_BASE            (APB2PERIPH_BASE + 0x1800)
21 #define GPIOF_BASE            (APB2PERIPH_BASE + 0x1C00)
22 #define GPIOG_BASE            (APB2PERIPH_BASE + 0x2000)
23
24 #define GPIOA               ((GPIO_TypeDef *) GPIOA_BASE)
25 #define GPIOB               ((GPIO_TypeDef *) GPIOB_BASE)
26 #define GPIOC               ((GPIO_TypeDef *) GPIOC_BASE)
27 #define GPIOD               ((GPIO_TypeDef *) GPIOD_BASE)
28 #define GPIOE               ((GPIO_TypeDef *) GPIOE_BASE)
29 #define GPIOF               ((GPIO_TypeDef *) GPIOF_BASE)
30 #define GPIOG               ((GPIO_TypeDef *) GPIOG_BASE)
```

在点亮 LED 的时候，我们还开了 GPIO 的时钟，用到了 RCC 这个外设，现在我们也定义一个 RCC 寄存器结构体，加上那些地址定义，总体代码如下：

```
1 #define RCC_BASE              (AHBPERIPH_BASE + 0x1000)
```

```
#define RCC                 ((RCC_TypeDef *) RCC_BASE)

typedef struct {
     __IO uint32_t CR;
     __IO uint32_t CFGR;
     __IO uint32_t CIR;
     __IO uint32_t APB2RSTR;
     __IO uint32_t APB1RSTR;
     __IO uint32_t AHBENR;
     __IO uint32_t APB2ENR;
     __IO uint32_t APB1ENR;
     __IO uint32_t BDCR;
     __IO uint32_t CSR;
} RCC_TypeDef;

```

跟 GPIO 不同的是，RCC 这个外设是挂载到 AHB 总线上。

现在我们点亮 LED 的函数就变成了

```
// 开启端口 B 的时钟
RCC->APB2ENR |= 1<<3;

// 配置 PB0 为通用推挽输出模式，速率为 2M
GPIOB->CRL = (2<<0) | (0<<2);

// PB0 输出低电平，点亮 LED
GPIOB->ODR = 0<<0;
```

对比之前的代码

```
// 开启端口 B 的时钟
RCC_APB2ENR |= 1<<3;

// 配置 PB0 为通用推挽输出模式，速率为 2M
GPIOB_CRL = (2<<0) | (0<<2);

// PB0 输出低电平，点亮 LED
GPIOB_ODR = 0<<0;
```

一个用的是结构体，一个用的是宏，仅仅从这三行代码看不出有啥区别，但是如果要操作其他寄存器的时候，用结构体就可以直接操作，用宏就还要一个个找到寄存器的绝对地址重新定义。

比如我们要操作 GPIOB 的 BSRR（bit reset register）的时候，用结构体时我们就可以这样操作：

```
GPIOB->BRR = 0X01;
```

这时候 PB0 就输出低电平，LED 被点亮。注意：BRR 低 16 位有效，只能以字的形式操作，功能是复位相应的 IO 口，写 1 清 0，写 0 没有影响。

### 8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)

地址偏移：0x14

复位值：0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 保留 | | | | | | | | | | | | | | | |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

通用和复用功能I/O STM32F10xxx参考手册

| | |
|---|---|
| 位31:16 | 保留。 |
| 位15:0 | **BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y)<br>这些位只能写入并只能以字(16位)的形式操作。<br>0：对对应的ODRy位不产生影响<br>1：清除对应的ODRy位为0 |

图 8 GPIO 端口位清除寄存器

现在我们再整理下代码，如下所示：

```
#include <stdint.h>
#define     __IO    volatile

typedef struct {
    __IO uint32_t CRL;
    __IO uint32_t CRH;
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t BRR;
    __IO uint32_t LCKR;
} GPIO_TypeDef;

typedef struct {
    __IO uint32_t CR;
    __IO uint32_t CFGR;
    __IO uint32_t CIR;
    __IO uint32_t APB2RSTR;
    __IO uint32_t APB1RSTR;
    __IO uint32_t AHBENR;
    __IO uint32_t APB2ENR;
    __IO uint32_t APB1ENR;
    __IO uint32_t BDCR;
    __IO uint32_t CSR;
} RCC_TypeDef;

#define PERIPH_BASE           ((uint32_t)0x40000000)

#define APB1PERIPH_BASE       PERIPH_BASE
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)

#define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
#define GPIOB_BASE            (APB2PERIPH_BASE + 0x0C00)
#define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)
#define GPIOD_BASE            (APB2PERIPH_BASE + 0x1400)
#define GPIOE_BASE            (APB2PERIPH_BASE + 0x1800)
#define GPIOF_BASE            (APB2PERIPH_BASE + 0x1C00)
#define GPIOG_BASE            (APB2PERIPH_BASE + 0x2000)
```

```
40 #define RCC_BASE              (AHBPERIPH_BASE + 0x1000)
41
42 #define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
43 #define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
44 #define GPIOC                ((GPIO_TypeDef *) GPIOC_BASE)
45 #define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
46 #define GPIOE                ((GPIO_TypeDef *) GPIOE_BASE)
47 #define GPIOF                ((GPIO_TypeDef *) GPIOF_BASE)
48 #define GPIOG                ((GPIO_TypeDef *) GPIOG_BASE)
49 #define RCC                  ((RCC_TypeDef *) RCC_BASE)
50
51
52 #define RCC_APB2ENR    *(volatile unsigned long *)0x40021018
53 #define GPIOB_CRL      *(volatile unsigned long *)0x40010C00
54 #define GPIOB_ODR      *(volatile unsigned long *)0x40010C0C
55
56 int main(void)
57 {
58     // 开启端口 B 的时钟
59     RCC->APB2ENR |= 1<<3;
60
61     // 配置 PB0 为通用推挽输出模式，速率为 2M
62     GPIOB->CRL = (2<<0) | (0<<2);
63
64     // PB0 输出低电平，点亮 LED
65     GPIOB->ODR = 0<<0;
66
67 }
68
69 void SystemInit(void)
70 {
71 }
```

### 1.4.4 小结流程

现在我们来总结下上面代码实现的过程，这个过程也是我们从零开始点亮 LED 的过程，代码全部由我们自己编写（除了启动代码），每一行都有根有据，都可以从《STM32 中文参考手册》查到。

①、定义一个外设（GPIO）寄存器结构体，结构体的成员包含该外设的所有寄存器，成员的排列顺序跟寄存器偏移地址一样，成员的数据类型跟寄存器的一样。

②外设内存映射，即把地址跟外设建立起一一对应的关系。51 单片机中用 SFR 实现，STM32 中用宏定义实现。

③外设声明，即把外设的名字定义成一个外设寄存器结构体类型的指针。

④操作寄存器，实现点亮 LED。

### 1.4.5 新建头文件 stm32f10x.h

为了使代码看起来不那么臃肿，我们这里引入文件的概念，让不同功能的代码放在不同的文件里面。在 main.c 里面我们只保留 main 函数和一些头文件，把其他的宏定义放到一个单独的文件。

新建一个 stm32f10x.h，跟寄存器相关的代码都放在这里，主要是寄存器映像，跟 51 单片机里面的 reg51.h 这个头文件差不多。然后我们在 main.c 里面包含这个头文件即可，现在我们的主函数就变成这样：

```
1  #include "stm32f10x.h"
2
3  int main(void)
4  {
5      // 开启端口 B 的时钟
6      RCC->APB2ENR |= 1<<3;
7
8      // 配置 PB0 为通用推挽输出模式，速率为 2M
9      GPIOB->CRL = (2<<0) | (0<<2);
10
11     // PB0 输出低电平，点亮 LED
12     GPIOB->ODR = 0<<0;
13
14 }
15
16 void SystemInit(void)
17 {
18 }
```

### 1.4.6 新建 tm32f10x_gpio.h

上面我们在控制 GPIO 输出内容的时候控制的是 ODR（Output data register）寄存器，ODR 是一个 16 位的寄存器，必须以字的形式控制，相当于 51 里面的总线操作。

其实我们还可以控制 BSRR 和 BRR 这两个寄存器来控制 IO 的电平，下面我们简单介绍下 BRR 寄存器的功能，BSRR 自行看手册研究。

**BRR:bit reset register**

**8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)**

地址偏移：0x14

复位值：0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 保留 | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 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 |

通用和复用功能I/O　　STM32F10xxx参考手册

| | |
|---|---|
| 位31:16 | 保留。 |
| 位15:0 | **BRy**: 清除端口x的位y (y = 0...15) (Port x Reset bit y)<br>这些位只能写入并只能以字(16位)的形式操作。<br>0：对对应的ODRy位不产生影响<br>1：清除对应的ODRy位为0 |

图 9 GPIO 端口位清除寄存器

位清除寄存器 BRR 只能实现位清 0 操作，是一个 32 位寄存器，低 16 位有效，写 0 没影响，写 1 清 0。

现在我们要使 PB0 输出低电平，点亮 LED，则只要往 BRR 的 BR0 位写 1 即可，其他位为 0，代码如下：

```
1 GPIOB->BRR = 0X0001;
```

这时 PB0 就输出了低电平，LED 就被点亮了。

如果要 PB2 输出低电平，则是：

```
1 GPIOB->BRR = 0X0004;
```

如果要 PB3/4/5/6。。。。。。这些 IO 输出低电平呢？道理是一样的，只要往 BRR 的相应位置赋不同的值即可。因为 BRR 是一个 16 位的寄存器，位数比较多，赋值的时候容易出错，而且从赋值的 16 进制数字我们很难清楚的知道控制的是哪个 IO。这时，我们是否可以把 BRR 的每个位置 1 都用宏定义来实现，如 GPIO_Pin_0 就表示 0X0001，GPIO_Pin_2 就表示 0X0004。只要我们定义一次，以后都可以使用，而且还见名知意。

GPIO_pins_define 代码如下：

```
 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 */
 5 #define GPIO_Pin_4      ((uint16_t)0x0010)  /*!< Pin 4 selected */
 6 #define GPIO_Pin_5      ((uint16_t)0x0020)  /*!< Pin 5 selected */
 7 #define GPIO_Pin_6      ((uint16_t)0x0040)  /*!< Pin 6 selected */
 8 #define GPIO_Pin_7      ((uint16_t)0x0080)  /*!< Pin 7 selected */
 9 #define GPIO_Pin_8      ((uint16_t)0x0100)  /*!< Pin 8 selected */
10 #define GPIO_Pin_9      ((uint16_t)0x0200)  /*!< Pin 9 selected */
11 #define GPIO_Pin_10     ((uint16_t)0x0400)  /*!< Pin 10 selected */
12 #define GPIO_Pin_11     ((uint16_t)0x0800)  /*!< Pin 11 selected */
13 #define GPIO_Pin_12     ((uint16_t)0x1000)  /*!< Pin 12 selected */
14 #define GPIO_Pin_13     ((uint16_t)0x2000)  /*!< Pin 13 selected */
15 #define GPIO_Pin_14     ((uint16_t)0x4000)  /*!< Pin 14 selected */
16 #define GPIO_Pin_15     ((uint16_t)0x8000)  /*!< Pin 15 selected */
17 #define GPIO_Pin_All    ((uint16_t)0xFFFF)  /*!< All pins selected */
```

这时 PB0 就输出了低电平的代码就变成了：

```
1 GPIOB->BRR = GPIO_Pin_0;
```

为了不使 main 函数看起来冗余，GPIO_pins_define 的代码不应该放在 main 里面，因为其是跟 GPIO 相关的，我们可以把这些宏放在一个单独的头文件里面。

在工程目录下新建 stm32f10x_gpio.h，把 GPIO_pins_define 代码放里面，然后把这个文件添加到工程里面。这时我们只需要在 main.c 里面包含这个头文件即可。

### 1.4.7 新建 stm32f10x_gpio.c

我们点亮 LED 的时候，控制的是 PB0 这个 IO，如果 LED 接到的是其他 IO，我们就需要把 GPIOB 修改成其他的端口，其实这样修改起来也很快很方便。但是为了提高程序的可读性和可移植性，我们是否可以编写一个专门的函数用来复位 GPIO 的某个位，这个函数有两个形参，一个是 GPIOX（X=A...G），另外一个是 GPIO_Pin（0...15），函数的主体则是根据形参 GPIOX 和 GPIO_Pin 来控制 BRR 寄存器，代码如下：

```
1 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
2 {
3     GPIOx->BRR = GPIO_Pin;
4 }
```

这时，PB0 输出低电平，点亮 LED 的代码就变成了：

```
1 GPIO_ResetBits(GPIOB,GPIO_Pin_0);
```

同样，因为这个函数是控制 GPIO 的函数，我们可以新建一个专门的文件来放跟 gpio 有关的函数。

在工程目录下新建 stm32f10x_gpio.c，把 GPIO 相关的函数放里面。

这时我们是否发现刚刚新建了一个头文件 stm32f10x_gpio.h，这两个文件存放的都是跟外设 GPIO 相关的。C 文件里面的函数会用到 h 头文件里面的定义，这两个文件是相辅相成的，故我们在 stm32f10x_gpio.c 文件中也包含 stm32f10x_gpio.h 这个头文件。别忘了把 stm32f10x.h 这个头文件也包含进去，因为有关寄存器的所有定义都在这个头文件里面。

如果我们写其他外设的函数，我们也应该跟 GPIO 一样，新建两个文件专门来存函数，比如 RCC 这个外设我们可以新建 stm32f10x_rcc.c 和 stm32f10x_rcc.h。其他外依葫芦画瓢即可。

stm32f10x_gpio.c 文件代码如下：

```
1 #include "stm32f10x.h"
2 #include"stm32f10x_gpio.h"
3
4 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
5 {
6     GPIOx->BRR = GPIO_Pin;
7 }
```

我们还要记得把 void GPIO_ResetBits()在 stm32f10x_gpio.h 里面声明下，这样其他文件只要包含 stm32f10x_gpio.h 这个头文件就可以使用 GPIO_ResetBits()这个函数了。以后不论新增加了什么函数都应该在自己的头文件下声明，这是个 C 语言的常识问题。

点亮 LED 会了，那关闭 LED 怎么办，我们可以控制 BSRR 这个寄存器来实现，这里我就直接写代码了：

先写一个 GPIO 端口置位函数，放到 stm32f10x_gpio.c 文件中，同样在 stm32f10x_gpio.h 头文件声明。

```
// GPIO 端口置位函数
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    GPIOx->BSRR = GPIO_Pin;
}
```

PB0 输出高电平，关闭 LED，代码如下：

```
GPIO_SetBits(GPIOB,GPIO_Pin_0);
```

现在我们再来看看 main 函数，看看点亮 LED 的代码是如何一步一步进化的：

```
#include "stm32f10x.h"
#include"stm32f10x_gpio.h"

int main(void)
{
    // 开启端口 B 的时钟
    RCC->APB2ENR |= 1<<3;

    // 配置 PB0 为通用推挽输出模式，速率为 2M
    GPIOB->CRL = (2<<0) | (0<<2);

    // PB0 输出低电平，点亮 LED
    //GPIOB->ODR = 0<<0;

    // PB0 输出低电平，点亮 LED
    //GPIOB->BRR = 0X0001;

    // PB0 输出低电平，点亮 LED
    //GPIOB->BRR = GPIO_Pin_0;

    // PB0 输出低电平，点亮 LED
    GPIO_ResetBits(GPIOB,GPIO_Pin_0);

    // PB0 输出高电平，关闭 LED
    GPIO_SetBits(GPIOB,GPIO_Pin_0);
}

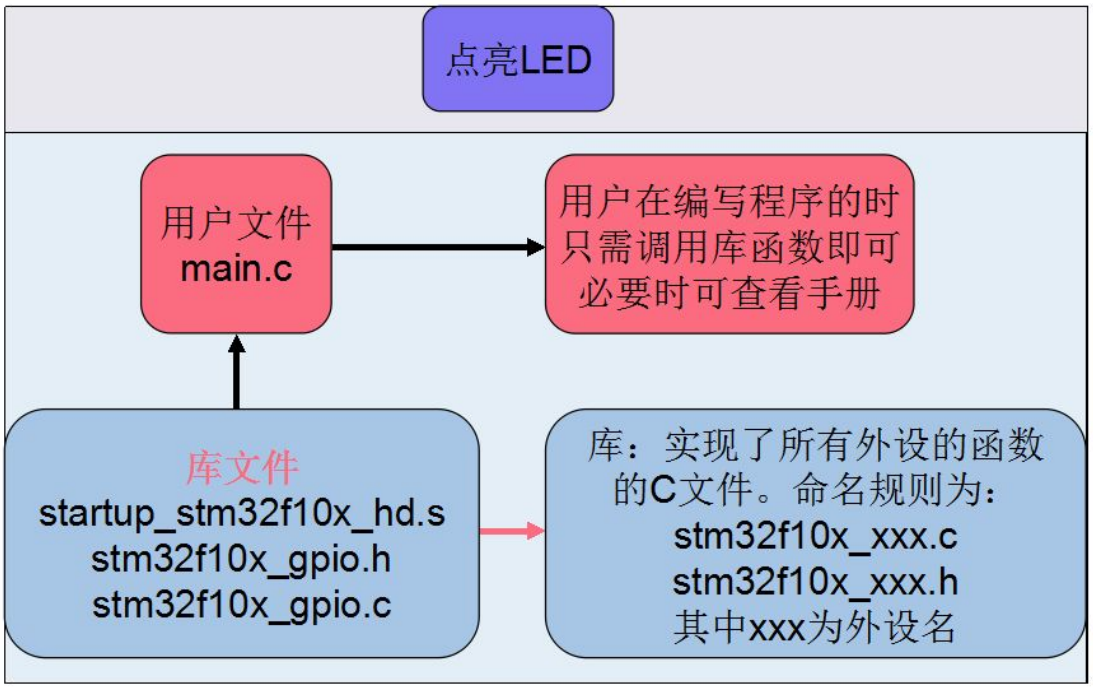
void SystemInit(void)
{
}
```

### 1.4.8 小结

我们从寄存器映像开始，把内存跟寄存器建立起一一对应的关系，然后操作寄存器点亮 LED，再到把寄存器操作封装成一个个函数。为了把不同外设的函数归类，我们引入了相应的文件来放这些函数，这一步一步走来，我们实现了库最简单的雏形，知道库是怎么来的。后面的工作就是不断的增加操作外设的函数，并且把所有的外设都写完，这样一个完整的库就实现了。

什么是库，这就是库。

下面我们用一张图来描述下我们刚刚的代码，让大家有一个整体的把握。




## 1.5 新的尝试—用库函数点亮 LED

### 1.5.1 新建工程

1. 新建本地工程文件夹

为了工程目录更加清晰，我们在本地电脑上新建 6 个文件夹，具体如下：

表格 1 工程目录文件夹清单

| 名称 | 作用 |
|---|---|
| Doc | 用来放对程序说明的文件，由写程序的人添加 |
| Libraries | 存放是库文件 |
| Listing | 存放编译器编译时候产生的 c/汇编/链接的列表清单 |
| Output | 存放编译产生的调试信息、hex 文件、预览信息、封装库等 |
| Project | 用来存放工程 |
| User | 用户编写的驱动文件 |

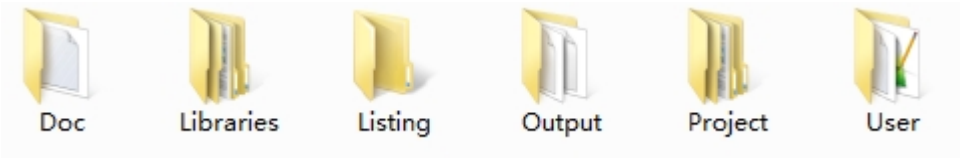



图 10 工程文件夹目录

在本地新建好文件夹后，把准备好的库文件添加到相应的文件夹下：

表格 2 工程目录文件夹内容清单

| 名称 | 作用 |
| --- | --- |
| Doc | readme.txt |
| Libraries | CMSIS：里面放着跟 CM3 内核有关的库文件 |
| | FWlib：STM32 外设库文件 |
| Listing | 暂时为空 |
| Output | 暂时为空 |
| Project | 暂时为空 |
| User | stm32f10x_conf.h：用来配置库的头文件 |
| | stm32f10x_it.h<br>stm32f10x_it.c：中断相关的函数都在这个文件编写，暂时为空 |
| | main.c：main 函数文件 |

## 2. 新建工程

打开 KEIL5，新建一个工程，工程名根据喜好命名，我这里取 LED-LIB，保存在 Project\RVMDK（uv4）文件夹下。

### 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，MINI 选 STM32F103VE，ISO 选 STM32F103ZE。

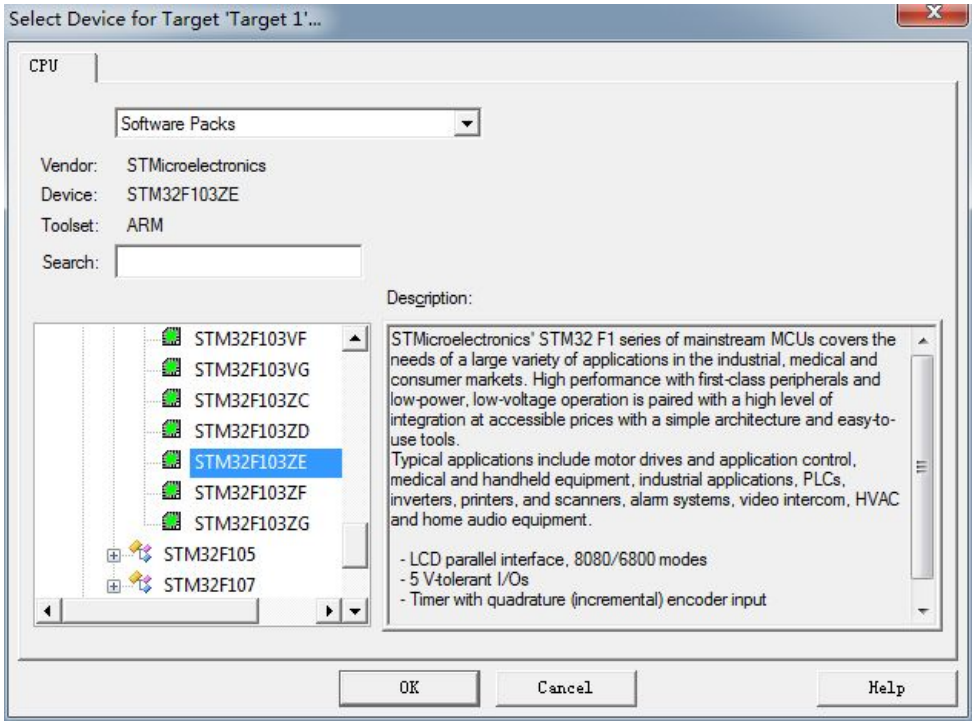



图 11 选择具体的 CPU 型号

### 在线添加库文件

等下我们手动添加库文件，这里我们点击关掉。

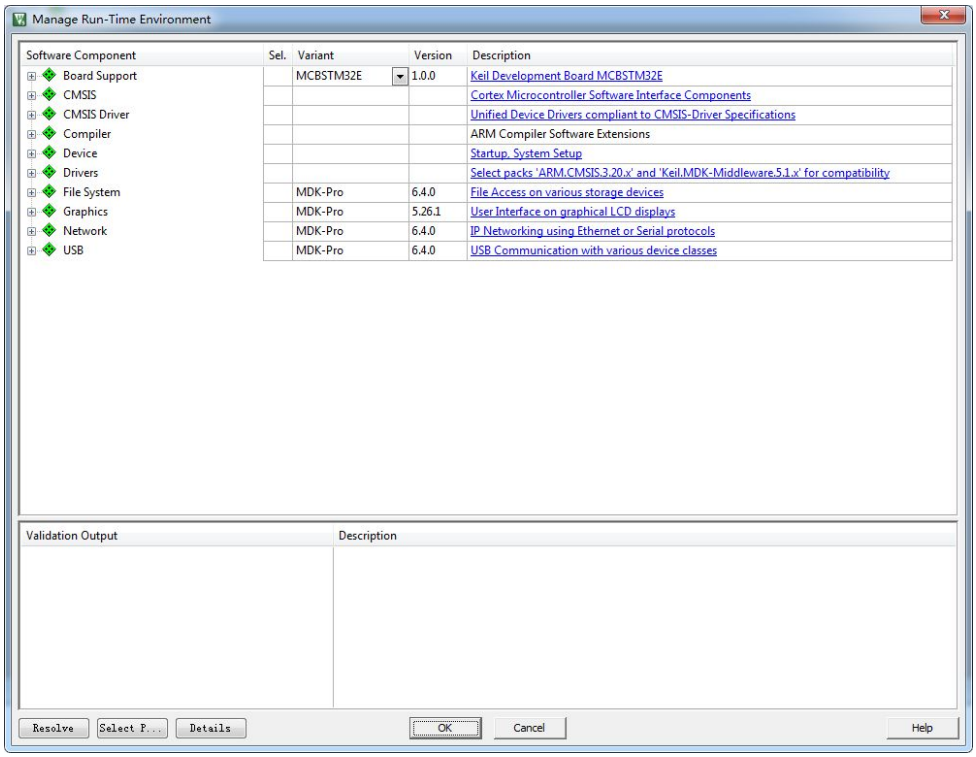



图 12 库文件管理

### 添加组文件夹

在新建的工程中添加 5 个组文件夹，用来存放各种不同的文件，文件从本地建好的工程文件夹下获取：

表格 3 工程内组文件夹内容清掉

| 名称 | 作用 |
| --- | --- |
| STARTUP | 存放汇编的启动文件：startup_stm32f10x_hd.s |
| CMSIS | 与 CM3 内核有关的库文件：<br>core_cm3.c<br>system_stm32f10x.c |
| FWLB | 与 STM32 外设相关的库文件<br>misc.c<br>stm32f10x_xxx.c（总共 22 个，xxx 代表外设名称） |
| USER | 用户编写的文件：<br>main.c：main 函数文件，暂时为空<br>Stm32f10x_it.c：跟中断有关的函数都放这个文件，暂时为空 |
| DOC | readme.txt：程序说明文件，用于说明程序的功能和注意事项等 |

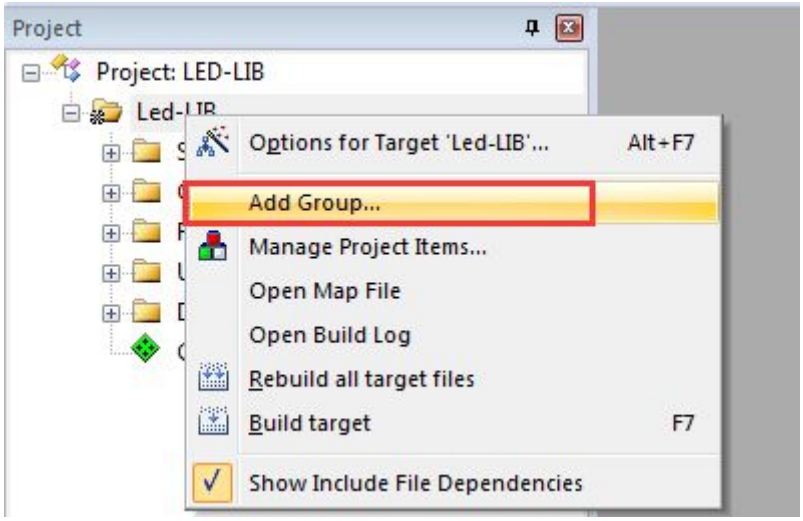



图 13 如何在工程中添加文件夹

### 配置魔术棒选项卡

这一步的配置工作很重要，很多人串口用不了 printf 函数，编译有问题，下载有问题，都是这个步骤的配置出了错。

①在 Target 中选中微库，为的是在日后编写串口驱动的时候可以使用 printf 函数

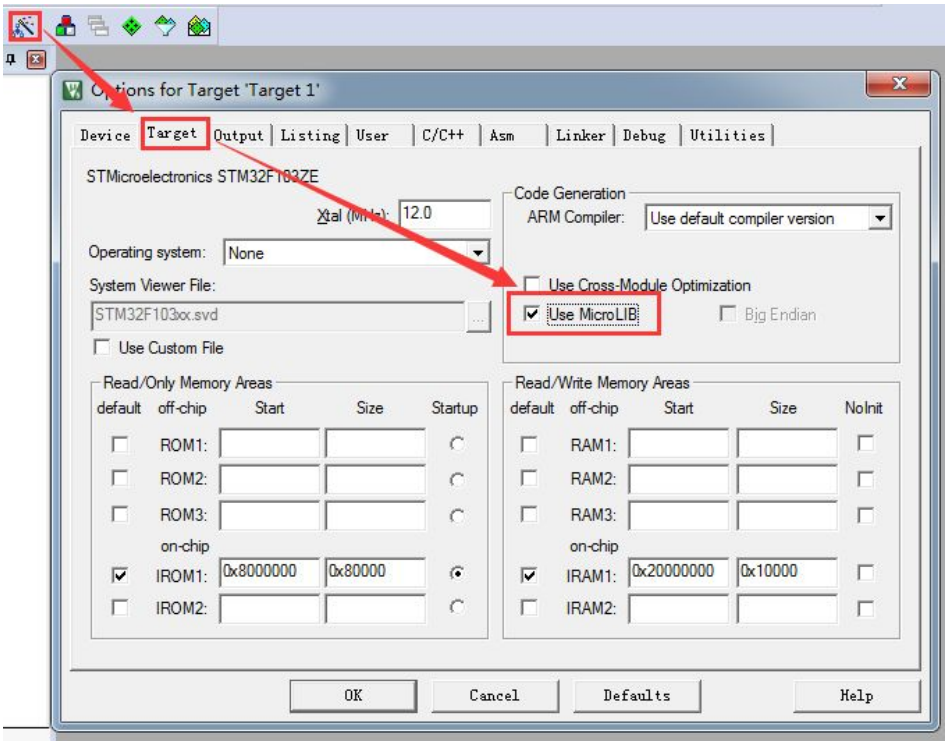



图 14 添加微库

②在 Output 选项卡中把输出文件夹定位到我们工程目录下的 output 文件夹，如果想在编译的过程中生成 hex 文件，那么那 Create HEX File 选项勾上。

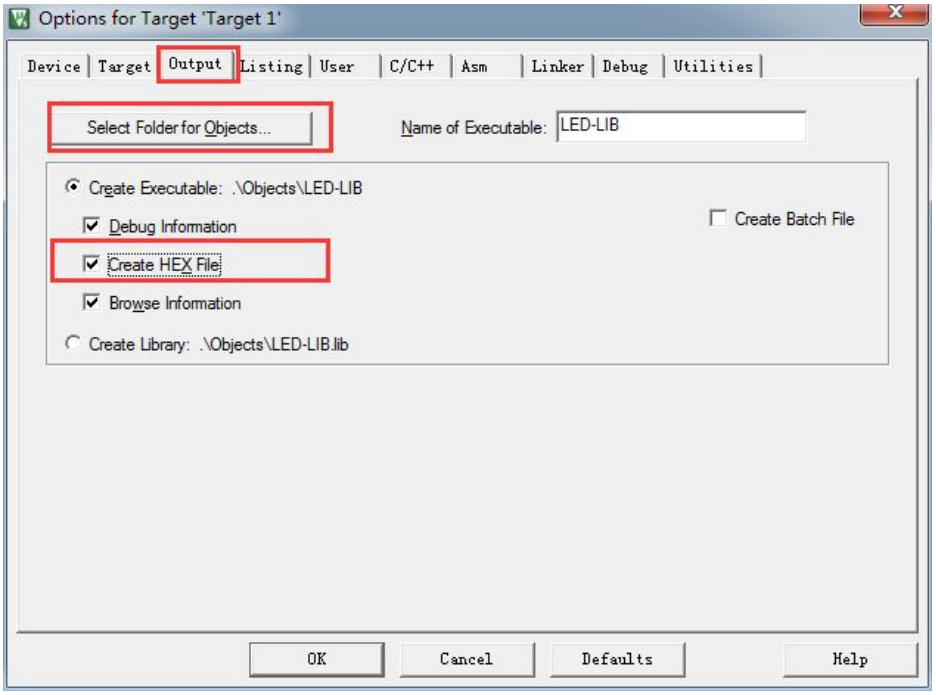



图 15 配置 Output 选项卡

③在 Listing 选项卡中把输出文件夹定位到我们工程目录下的 Listing 文件夹。

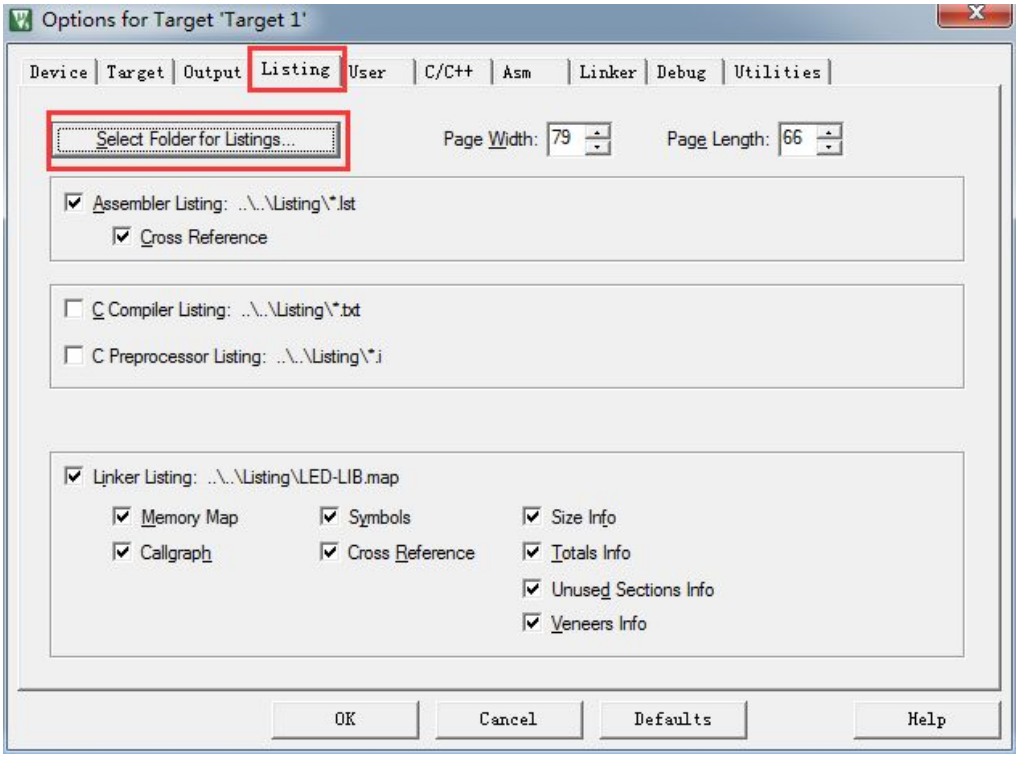



图 16 配置 Listing 选项卡

④在 C/C++选项卡中添加处理宏，和编译器编译的时候查找的头文件路径。

STM32F10X_HD：这个宏是为了区分使用 STM32F103 系列中不同容量型号的单片机库。我们用的单片机的 FLASH 的容量都是 512K，属于大容量

STM32F10X_HD：FLASH 大小在 256K~512K 之间的 STM32F101xx 和 STM32F103xx 控制器。STM32F10X_MD：FLASH 大小在 64K~128K 之间的 STM32F101xx 和 STM32F103xx 控制器。STM32F10X_LD：FLASH 大小在 16K~32K 之间的 STM32F101xx 和 STM32F103xx 控制器。

USE_STDPERIPH_DRIVER：为了包含 stm32f10x_conf.h 这个头文件。

在编译器中添加宏的好处就是，只要用了这个模版，就不用源文件中修改代码或者添加头文件。

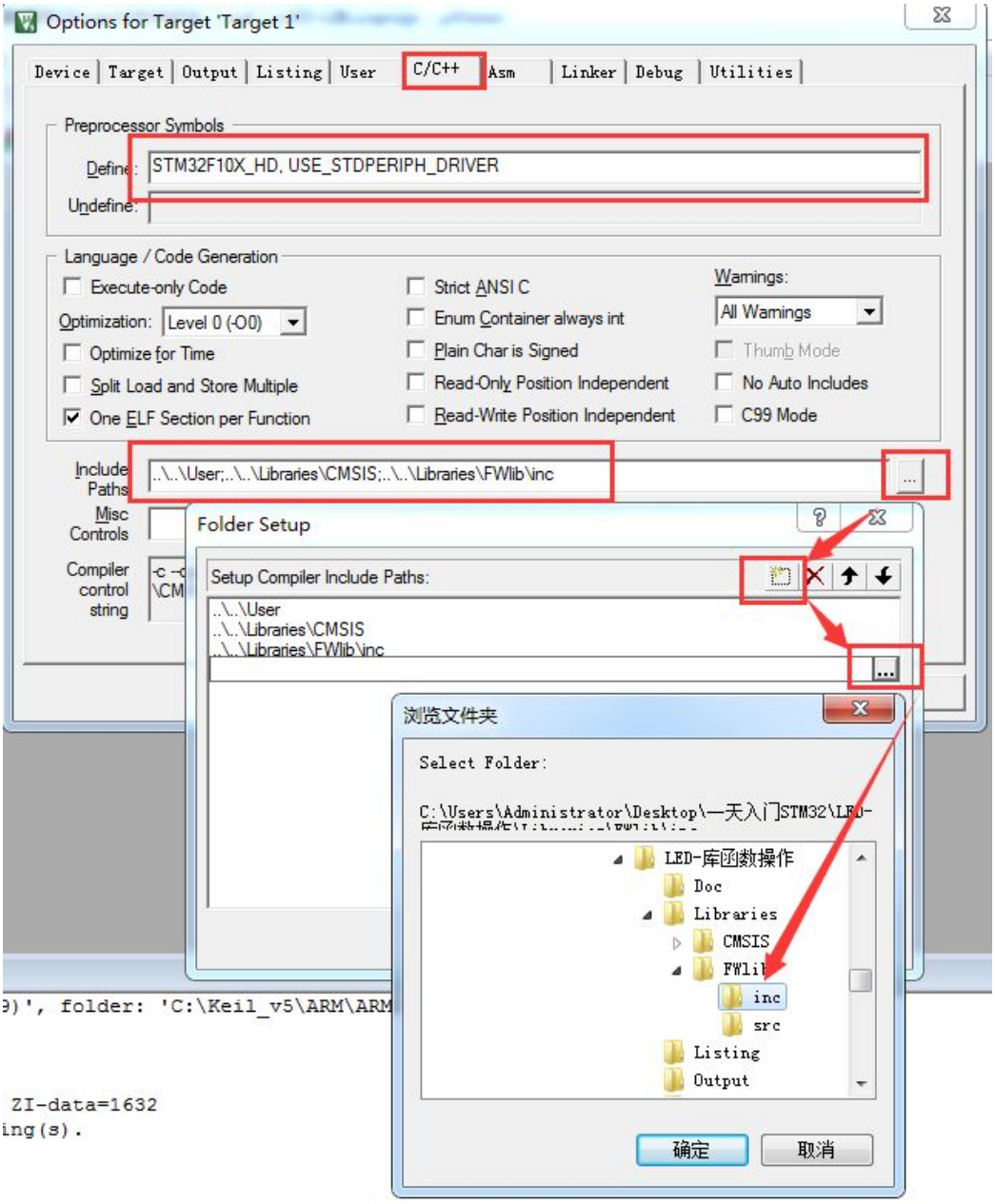



图 17 配置 C/C++ 选项卡

Include Paths 这里添加的是头文件的路径，如果编译的时候提示说找不到头文件，一般就是这里配置出了问题。你把头文件放到了哪个文件夹，就把该文件夹添加到这里即可。

### 下载器配置

这部分的配置最好是在安装好下载器驱动，下载器连接了电脑和开发板，且开发板上电后来配置。

这里面需要根据你使用了什么仿真器来配置，常用的有三种仿真器：JLINK/ARM-OB，ST-LINK，ULINK2，而且这个配置不是配置完一次之后以后就不会改变，当你换了芯片型号，或者其他操作（具体原因不明）都会改变下载器的配置。

①JLINK/ARM-OB 配置

要先安装了 JLINK 驱动之后，该配置才能下载，两者缺一不可。

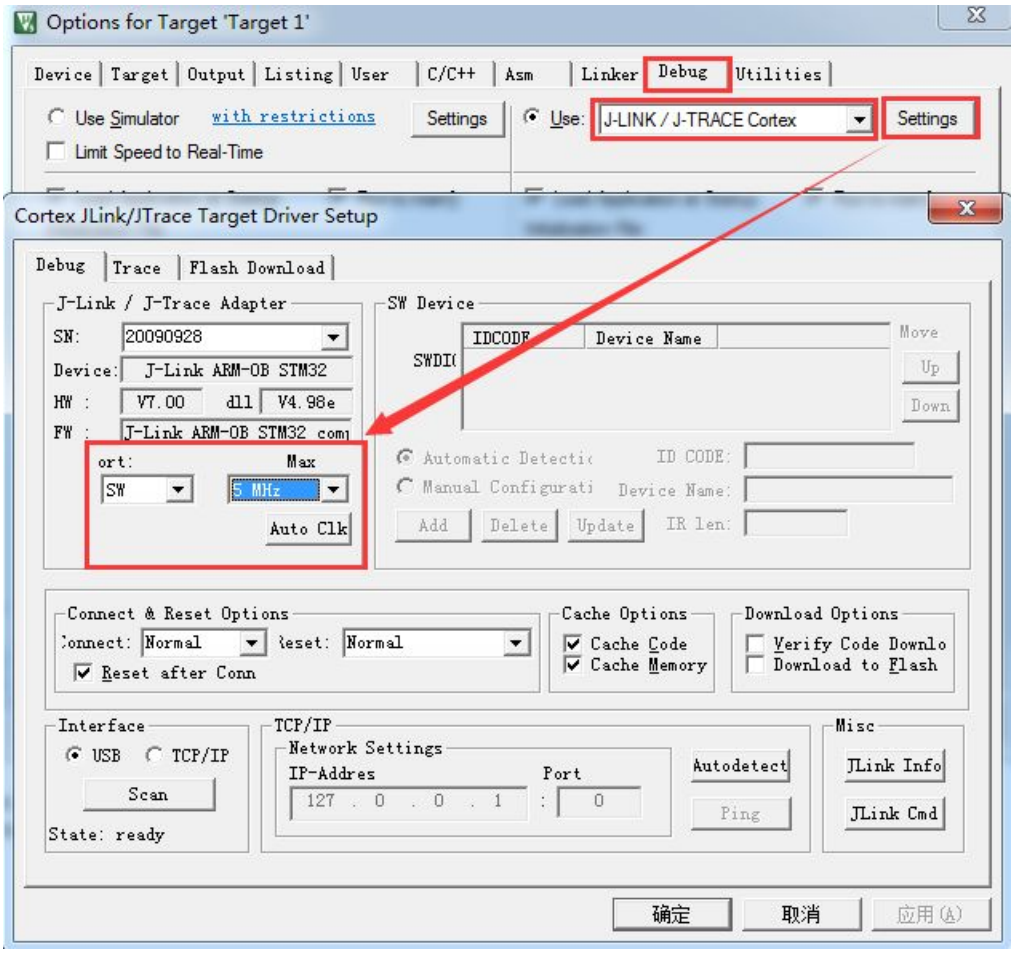



图 18 JLINK/ARM-OB 下载配置

②ST-LINK 配置

要先安装了 ST-LINK 驱动之后，该配置才能下载，两者缺一不可。

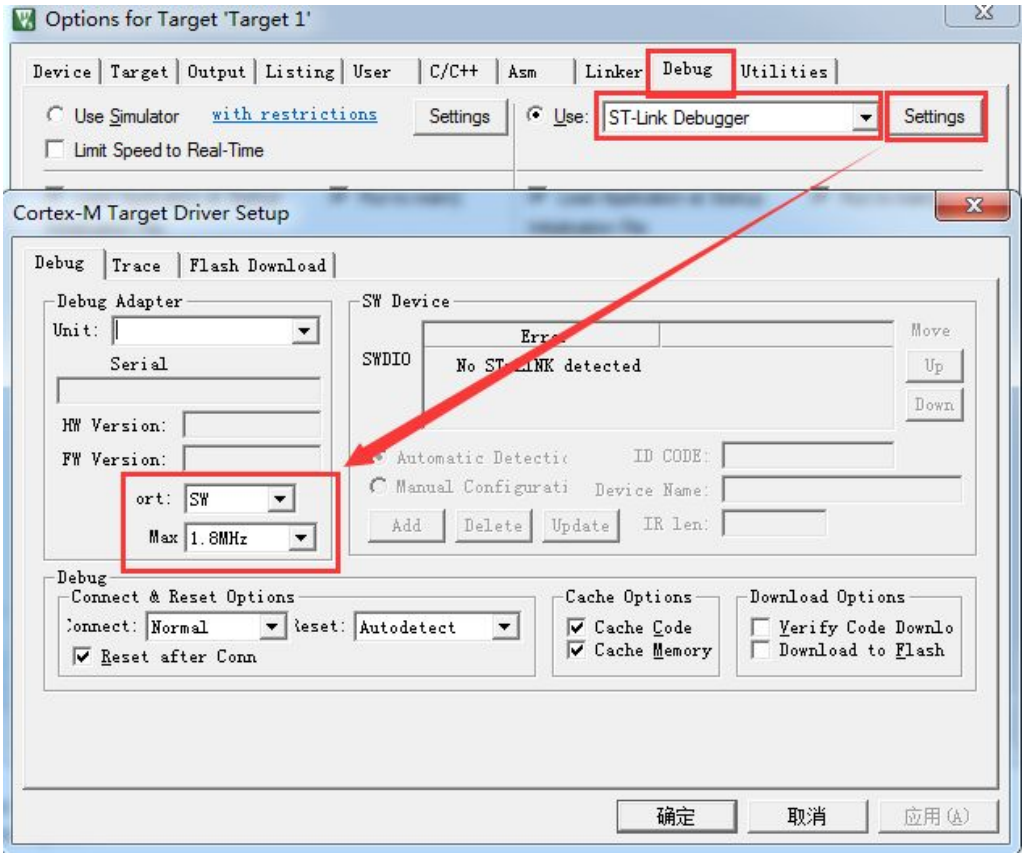



图 19 ST-LINK 下载配置

③ULINK2 配置

要先安装了 ULINK2 驱动之后，该配置才能下载，两者缺一不可。要注意的是设置成 ULINK2，而不是 ULINK。

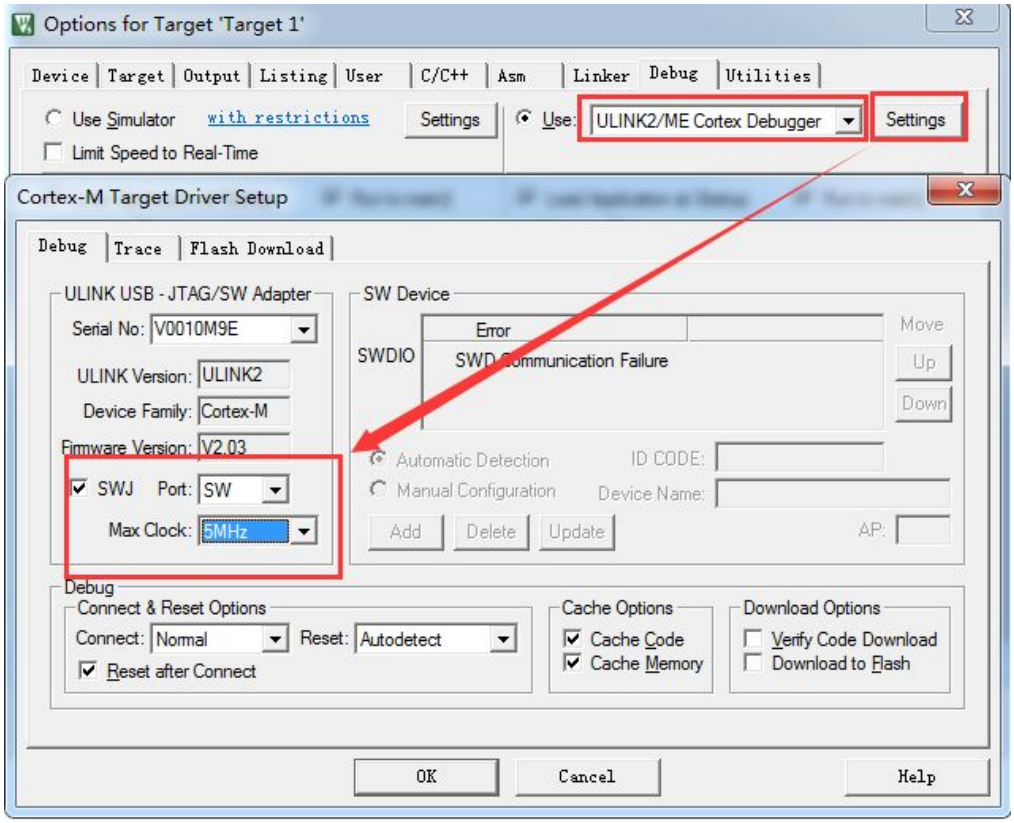
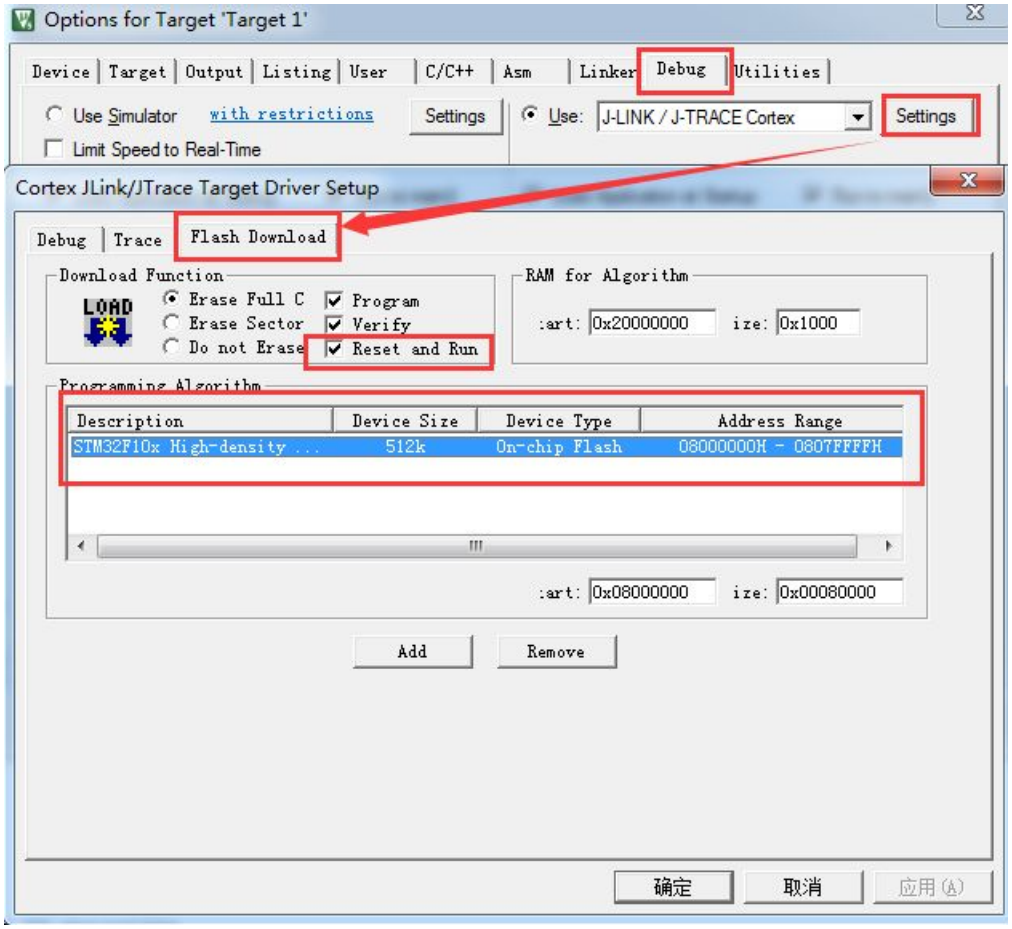



图 20 ULINK2 下载配置

### 选择 CPU 型号

这一步的配置也不是配置一次之后完事，常常会因为各种原因需要重新选择，当你下载的时候，提示说找不到 Device 的时候，请确保该配置是否正确。有时候下载程序之后，不会自动运行，要手动复位的时候，也回来看看这里的 Reset and Run 配置是否失效。MINI 和 ISO 用的 STM32 的 FLASH 都是 512K，所以选择 512K 大容量，如果使用的是其他型号的，要根据实际情况选择。




### 1.5.2 固件库分析

在写代码之前，我们先来分析下固件库，看看每个文件的作用是什么，这对我们能否清晰的调用库函数编程非常重要。

STM32 由 Cortex-M3 内核和内核之外的各种外设组成，库在编写的时候也遵循这中组成结构，把代码分成两大部分，一种是操作内核外设的，另外一种是内核之外的外设，为了听起来不那么绕，下面我们把内核之外的外设用处理器外设来代替。

| 内核相关 | 处理器相关 |
| --- | --- |
| cor_cm.h | startup_stm32f10x_hd.s |
| misc.h 、misc.c | stm32f10x.h |
| | stm32f10x_xxx.h、stm32f10x_xxx.c （XXX 表示外设） |

下面我们大概分析下每个文件的作用。

# 1. 处理器相关

### startup_stm32f10x_hd.s

这个是由汇编编写的启动文件，是 STM32 上电启动的第一个程序，启动文件主要实现了：1、初始化堆栈指针 SP；2、设置 PC 指针=Reset_Handler；3、设置向量表的地址，并初始化向量表，向量表里面放的是 STM32 所有中断函数的入口地址 4、调用库函数 SystemInit，把系统时钟配置成 72M，SystemInit 在库文件 stytem_stm32f10x.c 中定义；5、跳转到标号_main，最终去到 C 的世界。

### system_stm32f10x.c

这个文件的作用是里面实现了各种常用的系统时钟设置函数，有 72M，56M，48，36，24，8M，我们使用的是是把系统时钟设置成 72M。

### Stm32f10x.h

这个头文件非常重要，可以说是上帝之手。这个头文件实现了：1、处理器外设寄存器的结构体定义 2、处理器外设的内存映射 3、处理器外设寄存器的位定义。

关于 1 和 2 我们在用寄存器点亮 LED 的时候有讲解。其中 3：处理器外设寄存器的位定义，这个非常重要，具体是什么意思？我们知道一个寄存器有很多个位，每个位写 1 或者写 0 的功能都是不一样的，处理器外设寄存器的位定义就是把外设的每个寄存器的每一个位写 1 的 16 进制数定义成一个宏，宏名即用该位的名称表示，如果我们操作寄存器要开启某一个功能的话，就不用自己亲自去算这个值是多少，可以直接到这个头文件里面找。

我们以片上外设 ADC 为例，假设我们要启动 ADC 开始转换，根据手册我们知道是要控制 ADC_CR2 寄存器的位 0：ADON，即往位 0 写 1，即：ADC->CR2=0x00000001;这是一般的操作方法。现在这个头文件里面有关于 ADON 位的位定义：
#define ADC_CR2_ADON ((uint32_t)0x00000001)，有了这个位定义，我们刚刚的代码就变成了：ADC->CR2=ADC_CR2_ADON。这对于我们编程是何其方便，简直就是天降救星，感激之情无以言表。

无论是寄存器编程还是固件库编程，都必须包含这个头文件，有关外设寄存器的说明都在这里面。

### stm32f10x_xxx.h

stm32f10x_xxx.h：外设 xxx 应用函数库头文件，这里面主要定义了实现外设某一功能的结构体，比如通用定时器有很多功能，有定时功能，有输出比较功能，有输入捕捉功能，而通用定时器有非常多的寄存器要实现某一个功能，比如定时功能，我们根本不知道具体要操作哪些寄存器，这个头文件就为我们打包好了要实现某一个功能的寄存器，是以

机构体的形式定义的，比如通用定时器要实现一个定时的功能，我们只需要初始化 TIM_TimeBaseInitTypeDef 这个结构体里面的成员即可，里面的成员就是定时所需要操作的寄存器。 有了这个头文件，我们就知道要实现某个功能需要操作哪些寄存器，然后再回手册中精度这些寄存器的说明即可。

#### stm32f10x_xxx.c

stm32f10x_xxx.c：外设 xxx 应用函数库，这里面写好了操作 xxx 外设的所有常用的函数，我们使用库编程的时候，使用的最多的就是这里的函数。

## 2. 内核相关

#### cor_cm3.h

这个头文件实现了：1、内核结构体寄存器定义 2、内核寄存器内存映射 3、内存寄存器位定义。跟处理器相关的头文件 stm32f10x.h 实现的功能一样，一个是针对内核的寄存器，一个是针对内核之外，即处理器的寄存器。

#### misc.h

内核应用函数库头文件，对应 stm32f10x_xxx.h。

#### misc.c

内核应用函数库文件，对应 stm32f10x_xxx.c。在 CM3 这个内核里面还有一些功能组件，如 NVIC、SCB、ITM、MPU、CoreDebug，CM3 带有非常丰富的功能组件，但是芯片厂商在设计 MCU 的时候有一些并不是非要不可的，是可裁剪的，比如 MPU、ITM 等在 STM32 里面就没有。其中 NVIC 在每一个 CM3 内核的单片机中都会有，但都会被裁剪，只能是 CM3 NVIC 的一个子集。在 NVIC 里面还有一个 SysTick，是一个系统定时器，可以提供时基，一般为操作系统定时器所用。

misc.h 和 mics.c 这两个文件提供了操作这些组件的函数，并可以在 CM3 内核单片机直接移植。

### 1.5.3 开始写代码

## 1. 如何管理库的头文件

这么多的库文件，如何调用，如何管理？当我们开始调用库函数写代码的时候，有些库我们不需要，在编译的时候可以不编译，可以通过一个总的头文件 stm32f10x_conf.h 来控制，该头文件主要代码如下：

代码 1 stm32f10x_conf.h 头文件代码

```
 1 //#include "stm32f10x_adc.h"
 2 //#include "stm32f10x_bkp.h"
 3 //#include "stm32f10x_can.h"
 4 //#include "stm32f10x_cec.h"
 5 //#include "stm32f10x_crc.h"
 6 //#include "stm32f10x_dac.h"
 7 //#include "stm32f10x_dbgmcu.h"
 8 //#include "stm32f10x_dma.h"
 9 //#include "stm32f10x_exti.h"
10 //#include "stm32f10x_flash.h"
11 //#include "stm32f10x_fsmc.h"
12 #include "stm32f10x_gpio.h"
13 //#include "stm32f10x_i2c.h"
14 //#include "stm32f10x_iwdg.h"
15 //#include "stm32f10x_pwr.h"
16 #include "stm32f10x_rcc.h"
17 //#include "stm32f10x_rtc.h"
18 //#include "stm32f10x_sdio.h"
19 //#include "stm32f10x_spi.h"
20 //#include "stm32f10x_tim.h"
21 //#include "stm32f10x_usart.h"
22 //#include "stm32f10x_wwdg.h"
23 //#include "misc.h"
```

这里面包含了全部外设的头文件，点亮一个 LED 我们只需要 RCC 和 GPIO 这两个外设的库函数即可，其中 RCC 控制的是时钟，GPIO 控制的具体的 IO 口。所以其他外设库函数的头文件我们注释掉，当我们需要的时候就把相应头文件的注释去掉即可。

stm32f10x_conf.h 这个头文件在 stm32f10x.h 这个头文件的最后面被包含，在第 8296 行：

```
1 #ifdef USE_STDPERIPH_DRIVER
2 #include "stm32f10x_conf.h"
3 #endif
```

代码的意思是，如果定义了 USE_STDPERIPH_DRIVER 这个宏的话，就包含 stm32f10x_conf.h 这个头文件。我们在新建工程的时候，在魔术棒选项卡 C/C++中，我们定义了 USE_STDPERIPH_DRIVER 这个宏，所以 stm32f10x_conf.h 这个头文件就被 stm32f10x.h 包含了，我们在写程序的时候只需要调用一个头文件：stm32f10x.h 即可。

## 2. 编写 LED 初始化函数

经过寄存器点亮 LED 的操作，我们知道操作一个 GPIO 输出的编程要点大概如下：

1、开启 GPIO 的端口时钟

2、选择要具体控制的 IO 口，即 pin

3、选择 IO 口输出的速率，即 speed

4、选择 IO 口输出的模式，即 mode

5、输出高/低电平

STM32 的时钟功能非常丰富，配置灵活，为了降低功耗，每个外设的时钟都可以独自的关闭和开启。STM32 中跟时钟有关的功能都由 RCC 这个外设控制，RCC 中有三个寄存器控制着所以外设时钟的开启和关闭：RCC_APHENR、RCC_APB2ENR 和 RCC_APB1ENR，AHB、APB2 和 APB1 代表着三条总线，所有的外设都是挂载到这三条总线上，GPIO 属于高速的外设，挂载到 APB2 总线上，所以其时钟有 RCC_APB2ENR 控制。

### GPIO 时钟控制

寄存器方式

```
1 // 开启端口 B 的时钟
2 RCC->APB2ENR |= 1<<3;
```

固件库方式

```
1 // 开启 GPIOB 的时钟
2 RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOB, ENABLE);
```

固件库函数：RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOB, ENABLE)函数的原型为：

```
1  void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph,
                               FunctionalState NewState)
2  {
3      /* Check the parameters */
4      assert_param(IS_RCC_APB2_PERIPH(RCC_APB2Periph));
5      assert_param(IS_FUNCTIONAL_STATE(NewState));
6      if (NewState != DISABLE) {
7          RCC->APB2ENR |= RCC_APB2Periph;
8      } else {
9          RCC->APB2ENR &= ~RCC_APB2Periph;
10     }
11 }
```

当程序编译一次之后，把光标定位到函数/变量/宏定义 处，按键盘的 F12 或鼠标右键的 Go to definition of，就可以找到原型。固件库的底层操作的就是 RCC 外设的 APB2ENR 这个寄存器，宏 RCC_APB2Periph_GPIOB 的原型是：0x00000008，即（1<<3），还原成

寄存器操作就是：RCC->APB2ENR |= 1<<<3。相比固件库操作，寄存器操作的代码可读性就很差，只有才查阅寄存器配置才知道具体代码的功能，而固件库操作恰好相反，见名知意。

### GPIO 端口配置

GPIO 的 pin，速度，模式，都由 GPIO 的端口配置寄存器来控制，其中 IO0~IO7 由端口配置低寄存器 CRL 控制，IO8~IO15 由端口配置高寄存器 CRH 配置。

寄存器方式

```
1 // 配置 PB0 为通用推挽输出模式，速率为 2M
2 GPIOB->CRL = (2<<0) | (0<<2);
```

相比寄存器一句话的代码，固件库的操作就显得有些复杂，但换来的是简单明了。固件库把端口配置的 pin，速度和模式封装成一个结构体：

```
1 typedef struct {
2     uint16_t GPIO_Pin;
3     GPIOSpeed_TypeDef GPIO_Speed;
4     GPIOMode_TypeDef GPIO_Mode;
5 } GPIO_InitTypeDef;
```

pin 可以是 GPIO_Pin_0~GPIO_Pin_15 或者是 GPIO_Pin_All，这些都是库预先定义好的宏。

speed 也被封装成一个结构体：

```
1 typedef enum {
2     GPIO_Speed_10MHz = 1,
3     GPIO_Speed_2MHz,
4     GPIO_Speed_50MHz
5 } GPIOSpeed_TypeDef;
```

速度可以是 10M，2M 或者 50M，这个由端口配置寄存器的 MODE 位控制，速度是针对 IO 口输出的时候而言，在输入的时候可以不用设置。

mode 也被封装成一个结构体：

```
1  typedef enum {
2      GPIO_Mode_AIN = 0x0,            // 模拟输入
3      GPIO_Mode_IN_FLOATING = 0x04,   // 浮空输入（复位后的状态）
4      GPIO_Mode_IPD = 0x28,           // 下拉输入
5      GPIO_Mode_IPU = 0x48,           // 上拉输入
6      GPIO_Mode_Out_OD = 0x14,        // 通用开漏输出
7      GPIO_Mode_Out_PP = 0x10,        // 通用推挽输出
8      GPIO_Mode_AF_OD = 0x1C,         // 复用开漏输出
9      GPIO_Mode_AF_PP = 0x18          // 复用推挽输出
10 } GPIOMode_TypeDef;
```

IO 口的模式有 8 种，输入输出各 4 种，由端口配置寄存器的 CNF 配置。平时用的最多的就是通用推挽输出，可以输出高低电平，驱动能力大，一般用于接数字器件。至于剩下的七种模式的用法和电路原理，我们在后面的 GPIO 章节再详细讲解。

所以 GPIO 端口的配置，最终用固件库实现就变成这样：

```
// 定义一个 GPIO_InitTypeDef 类型的结构体
GPIO_InitTypeDef GPIO_InitStructure;

// 选择要控制的 IO 口
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;

// 设置引脚为推挽输出
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;

// 设置引脚速率为 50MHz
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

/*调用库函数，初始化 GPIOB0*/
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

配置好 pin，speed，mode 之后，我们最后调用库函数 GPIO_Init()把刚刚的参数写到 CRL 或者 CRH 这两个寄存器中。

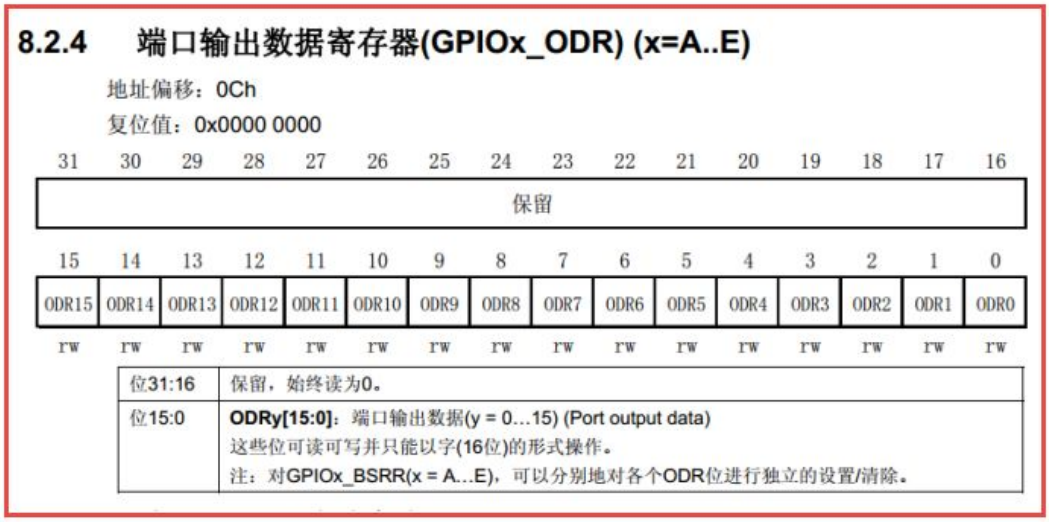
### GPIO 输出控制

GPIO 输出控制，可以通过端口数据输出寄存器 ODR、端口位设置/清除寄存器 BSRR 和端口位清除寄存器 BRR 这三个来控制。

端口输出寄存器 ODR 是一个 32 位的寄存器，低 16 位有效，对应着 IO0~IO15，只能以字的形式操作，不能单独对某一个位置位/清除。

**代码 2 寄存器操作 ODR**

```
// PB0 输出低电平，点亮 LED
GPIOB->ODR = 0<<0;
// PB0 输出高电平，点亮 LED
GPIOB->ODR = 1<<0;
```



8.2.4 端口输出数据寄存器(GPIOx_ODR) (x=A..E)

地址偏移：0Ch

复位值：0x0000 0000

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

| | |
|---|---|
| 位31:16 | 保留，始终读为0。 |
| 位15:0 | **ODRy[15:0]**：端口输出数据(y = 0...15) (Port output data)<br>这些位可读可写并只能以字(16位)的形式操作。<br>注：对GPIOx_BSRR(x = A...E)，可以分别地对各个ODR位进行独立的设置/清除。 |

图 21 ODR 寄存器

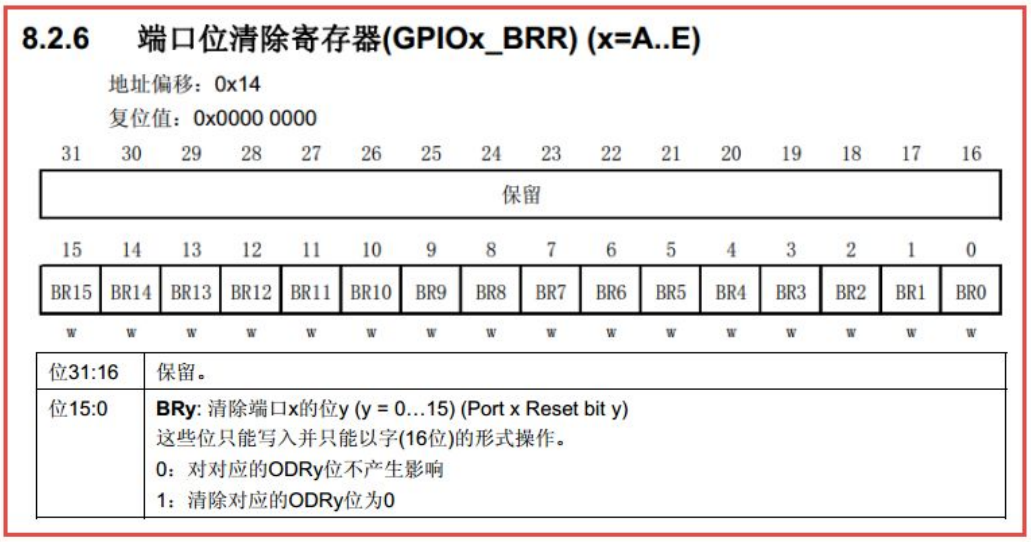
端口位清除寄存器 BRR 是一个 32 位的寄存器，低十六位有效，对应着 IO0~IO15，只能以字的形式操作，可以单独对某一个位操作，写 1 清 0。

代码 3 寄存器操作 BRR

```
1 // PB0 输出低电平，点亮 LED
2 GPIOB->BRR = 0X0001;
```

代码 4 固件库操作 BRR

```
1 // PB0 输出低电平，点亮 LED
2 GPIO_ResetBits(GPIOB, GPIO_Pin_0);
```



8.2.6 端口位清除寄存器(GPIOx_BRR) (x=A..E)

地址偏移：0x14

复位值：0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 保留 | | | | | | | | | | | | | | | |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

| | |
|---|---|
| 位31:16 | 保留。 |
| 位15:0 | **BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y)<br>这些位只能写入并只能以字(16位)的形式操作。<br>0：对对应的ODRy位不产生影响<br>1：清除对应的ODRy位为0 |

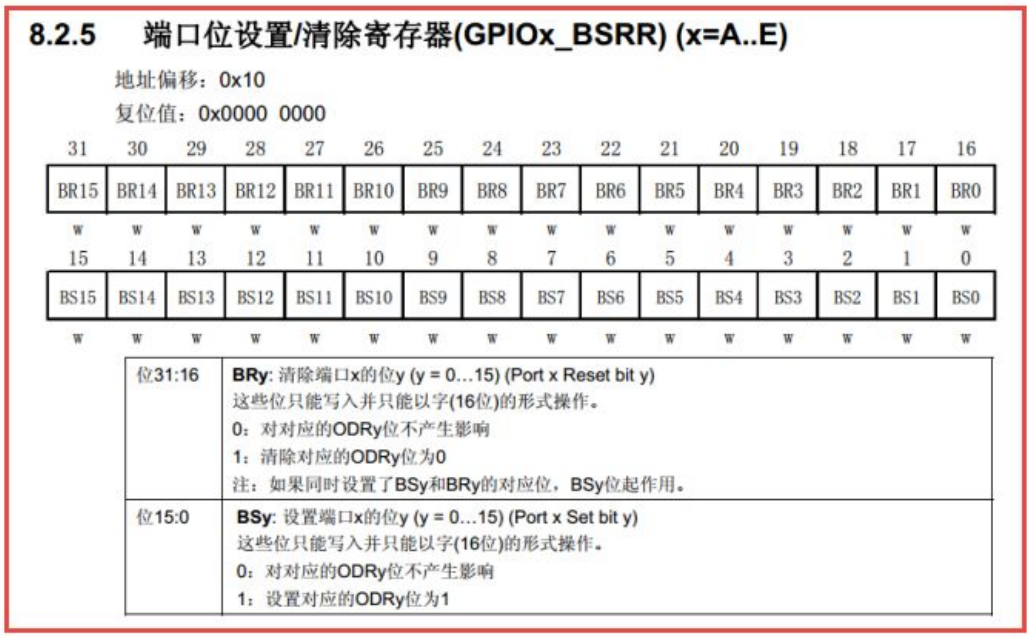
图 22 BRR 寄存器

BSRR 是一个 32 位的寄存器，低 16 位用于置位，写 1 有效，高 16 位用于复位，写 1 有效，相当于 BRR 寄存器。高 16 位我们一般不用，而是操作 BRR 这个寄存器，所以 BSRR 这个寄存器一般用来置位操作。

```
1 // PB0 输出高电平，熄灭 LED
2 GPIOB->BSRR = 0X0001;
```

代码 5 固件库操作 BSRR

```
1 // PB0 输出高电平，熄灭 LED
2 GPIO_SetBits(GPIOB, GPIO_Pin_0);
```



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

图 23 BSRR 寄存器

### LED GPIO 初始化函数

代码 6 寄存器 LED GPIO 初始化函数

```
// 开启端口 B 的时钟
RCC->APB2ENR |= 1<<3;

// 配置 PB0 为通用推挽输出模式，速率为 2M
GPIOB->CRL = (2<<0) | (0<<2);

// PB0 输出低电平，点亮 LED
GPIOB->ODR = 0<<0;

// PB0 输出高电平，熄灭 LED
GPIOB->ODR = 1<<0;
```

代码 7 固件库 LED GPIO 初始化函数

```
void LED_GPIO_Config(void)
{
    // 定义一个 GPIO_InitTypeDef 类型的结构体
    GPIO_InitTypeDef GPIO_InitStructure;

    // 开启 GPIOB 的时钟
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOB, ENABLE);

    // 选择要控制的 IO 口
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;

    // 设置引脚为推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;

    // 设置引脚速率为 50MHz
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

    /*调用库函数，初始化 GPIOB0*/
```

```
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    // 关闭 LED
    GPIO_SetBits(GPIOB, GPIO_Pin_0);
}
```

### 软件延时

```
// 简陋的软件延时函数
void SOFT_Delay(__IO uint32_t nCount)
{
    for (; nCount != 0; nCount--);
}
```

简单的通过软件来延时，具体时间不确定，并不能像 51 那么通过计算每条指令执行的时间来确切的计算延时时间。要想精确延时，必须通过定时器实现。

### 主函数

```
#include "stm32f10x.h"

void SOFT_Delay(__IO uint32_t nCount);
void LED_GPIO_Config(void);

int main(void)
{
    // 程序来到 main 函数之前，启动文件：statup_stm32f10x_hd.s 已经调用
    // SystemInit()函数把系统时钟初始化成 72MHZ
    // SystemInit()在 system_stm32f10x.c 中定义
    // 如果用户想修改系统时钟，可自行编写程序修改

    LED_GPIO_Config();

    while ( 1 ) {
        // 点亮 LED
        GPIO_ResetBits(GPIOB, GPIO_Pin_0);
        SOFT_Delay(0x0FFFFF);

        // 熄灭 LED
        GPIO_SetBits(GPIOB, GPIO_Pin_0);
        SOFT_Delay(0x0FFFFF);
    }
}
```

初始化 LED 用到的 GPIO，在 while 死循环中让 LED 闪烁。在程序来到 main 函数前，系统时钟已经初始化成了 72M，有关时钟部分我们在 RCC 这个章节中会详细讲解，这里不是重点。

### GPIO 其他库函数

有关 GPIO 的其他库函数，我们可以在 stm32f10x_gpio.h 中找到声明，然后在 stm32f10x_gpio.c 中找到函数的原型，根据函数的注释，可以知道每个函数的作用。阅读这些库函数的时候，最好配合《STM32 中文参考手册》寄存器描述部分一起看，这样学习的效果会非常好。

## 1.6 如果用固件库写自己的程序

### 力量之源—《STM32 中文参考手册》

在点亮 LED 的时候，无论是使用寄存器还是固件库，我们看的资料只有官方的《STM32 中文参考手册》，这个手册里面有每个外设的详细描述，我们应该花最多的时间去研读，网络上很多资料，包括 STM32 的很多书籍都是从这里衍生而来。这手册好比《易经》，浓缩了全部精华，世人根据易经原著写的书就好比网上各种各样的 STM32 书籍。

### 如虎添翼—固件库

那我们写其他驱动的时候是否也只需要这个手册就可以了？按道理上来说，是的，但这会花费我们非常多的时间，我们应该站在巨人的肩膀上来学习，参考他人的成果为自己所用。刚好这个巨人就在我们身边，固件库里面不仅提供了所有外设的驱动，还提供了驱动每个外设的例程，我们写程序首先就是参考官方提供的这些例程，然后再慢慢修改用到自己的项目中。这些官方例程在固件库的下面路径中可找到：
STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples

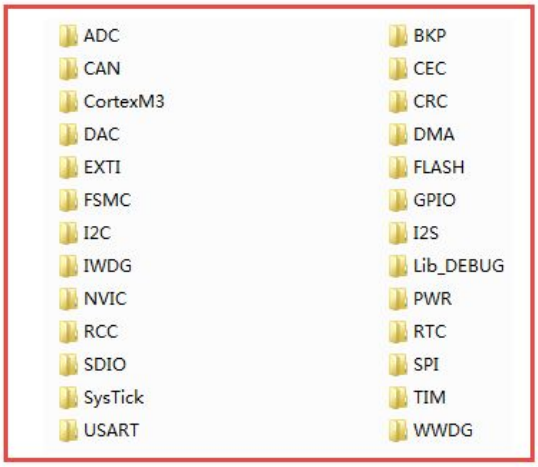
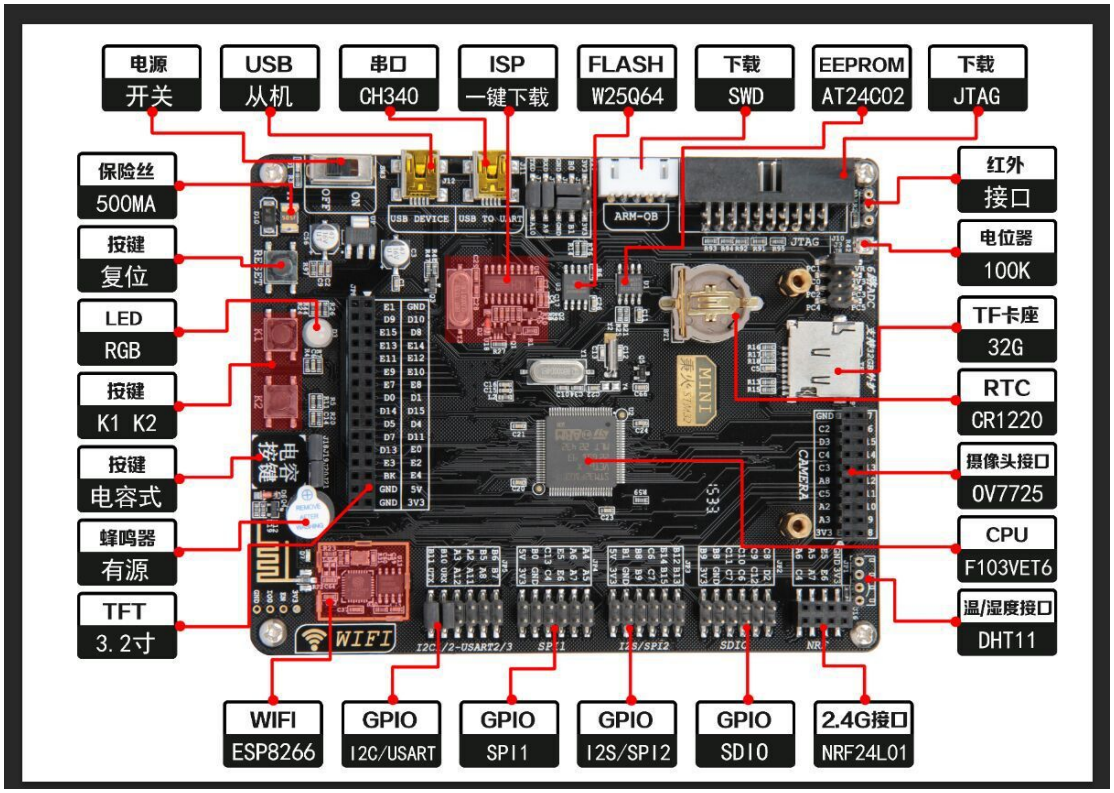



图 24 固件库例程

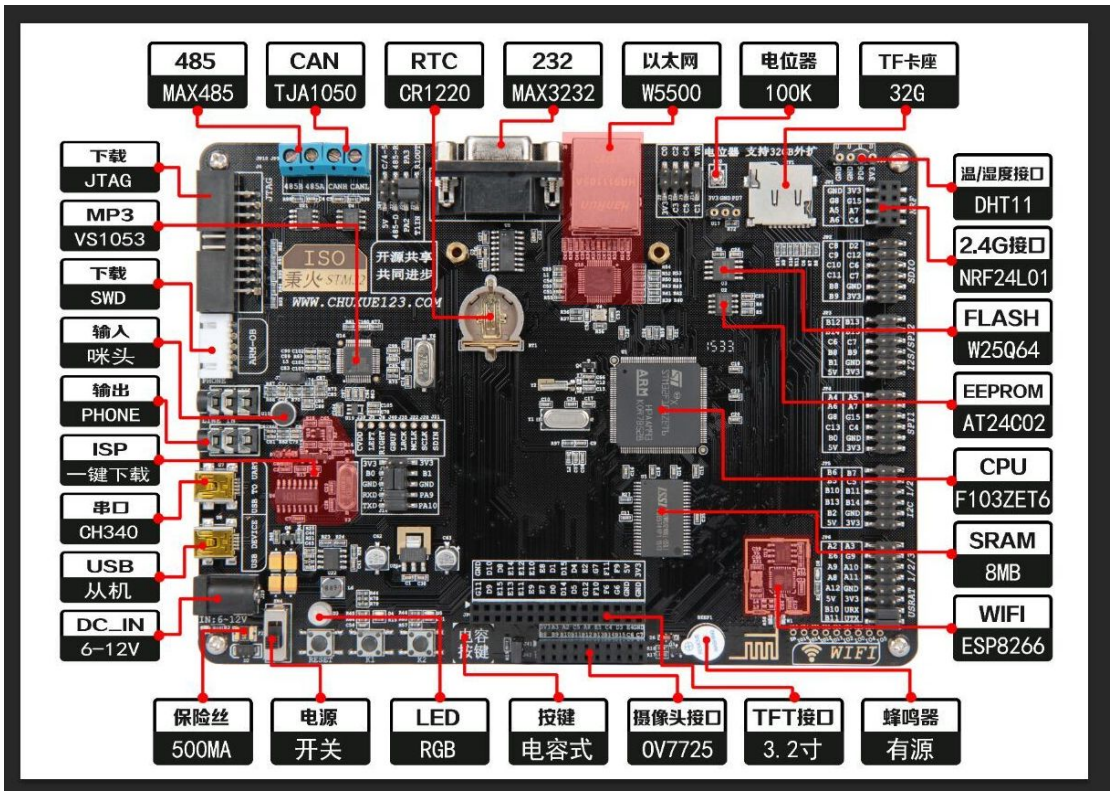
学习的时候，资料不需要多，特别是初学者，在最短的时间内点亮一个 LED 才是王道。STM32 入门首选《STM32 中文参考手册》+固件库，当入门之后，学得差不多了，你还需要一本《CM3 权威指南 CnR2》，有了这三板斧，搞定 STM32 就差不多了。

# 1.7 配套开发板

**MINI-V3 助学版（F103 系列）**



ISO-V3 旗舰版（F103 系列）




F429—至尊版（F4 系列）

