

STM32 学习笔记

从 51 开始，单片机玩了很长时间了，有 51，PIC，AVR 等等，早就想跟潮流玩玩 ARM，但一直没有开始，原因-----不知道玩了 ARM 可以做什么（对我自己而言）。如果为学习而学习，肯定学不好。然后 cortex-m3 出来了，据说，这东西可以替代单片机，于是马上开始关注。也在第一时间开始学习，可惜一开始就有点站错了队，选错了型（仍是对我自己而言）。我希望这种芯片应该是满大街都是，随便哪里都可以买得到，但我选的第一种显然做不到。为此，大概浪费了一年多时间吧，现在，回到对我来说是正确的道路上来啦，边学边写点东西。

这里写的是我的学习的过程，显然，很多时候会是不全面的，不系统的，感悟式的，甚至有时会是错误的，有些做法会是不专业的。那么，为什么我还要写呢？这是一个有趣的问题，它甚至涉及到博客为什么要存在的问题。显然，博客里面的写的东西，其正确性、权威性大多没法和书比，可为什么博客会存在呢？理由很多，我非专家，只说我的感慨。

我们读武侠小说，总会有一些创出独门功夫的宗师，功夫极高，然后他的弟子则基本上无法超越他。我在想，这位宗师在创造他自己的独门功夫时，必然会有很多的次的曲折、弯路、甚至失败，会浪费他的很多时间，而他教给弟子时，则已去掉了这些曲折和弯路，当然更不会把失败教给弟子，按理说，效率应该更高，可是没用，弟子大都不如师。为什么呢？也许知识本身并不是最重要的，获取知识的过程才是最重要的？也许所谓的知识，并不仅仅是一条条的结论，而是附着着很多说不清道不明的东西？如植物的根，一条主根上必带有大量的小小的触须？

闲话多了些，就权当前言了。下面准备开始。

一、条件的准备

我的习惯，第一步是先搭建一个学习的平台。原来学 51，PIC，AVR 时，都是想方设法自己做些工具，实验板之类，现在人懒了，直接购买成品了。

硬件电路板：火牛板

软件：有 keil 和 iar 可供选择。网上的口水仗不少，我选 keil，理由很简单，这个我熟。目前要学的知识中，软、硬件我都不熟，所以找一个我有点熟的东西就很重要。在我相当熟练之前，肯定不会用到 IAR，如果真的有一天不得不用 IAR，相信学起来也很容易，因为这个时候硬件部分我肯定很熟了，再加上有 keil 的基础，所以应该很容易学会了。

调试工具：JLINK V8。这个不多说了，价格便宜又好用，就是它了。

二、热身

网上选购的，付了款就是等了。拿到包裹，端详良久，起身。。。沐浴，更衣，焚香，，，

，

，

总得先吃晚饭，洗澡，再点个电蚊香什么的吧。

， 拆包

细细端详，做工精良，尤其那上面的 3.2 吋屏，越看越喜欢。接下来就是一阵折腾了，装 JLINK 软件，给板子通电，先试试 JLINK 能不能与电脑和板子通信上了。真顺，一点问题也没有。于是准备将附带的程序一个一个地写进去试一试。一检查，大部分例子的 HEX 文件并没有给出，这要下一步自己生成，但是几个大工程的例子都有 HEX 文件，如 MP3，如 UCCGI 测试等，写完以后观察程序运行的效果。因为之前也做过彩屏的东西，知道那玩艺代码量很大，要流畅地显示并不容，当时是用 AVR 做的，在 1.8 吋屏上显示一幅画要有一段时间。现在看起来，用 STM32 做的驱动显示出来的画面还是很快的，不过这里显示的大部分是自画图，并没有完整地显示一整幅的照片，所以到底快到什么程度还不好说，看来不久以后这可以作为一个学习点的。

一个晚上过去了，下一篇就是要开始 keil 软件的学习了。

STM32 学习笔记（2）

本想着偷点懒的，没想到竟被加了“精”，没办法啦，只能勤快点啦。。。硬件调通后，就要开始编程了。

编程的方法有两种，一种是用 st 提供的库，另一种是从最底层开始编程，网上关于使用哪种方法编程的讨论很多，据说用库的效率要低一些。但是用库编程非常方便，所以我还是从库开始啦。库是 ST 提供的，怎么说也不会差到哪里，再说了，用 32 位 ARM 的话，开发的观念也要随之改变一点了。

说说我怎么学的吧。

找个例子，如 GPIO，可以看到其结构如下：

SOURCE（文件夹）

- APP(文件夹)

-CMSIS（文件夹）

-STM32F10x_StdPeriph_Driver（文件夹）

Lis（文件夹）

OBJ（文件夹）

其中 SOURCE 中保存的是应用程序，其中又有好多子文件夹，而 CMSIS 文件夹中和 STM32F10x_StdPeriph_Driver 文件夹中是 ST 提供的库，这样，如果要做新的工程只要将这个文件夹整个复制过来就行，其中 APP 中保存自己的代码。

因为我们用 51 单片机时一般比较简单，有时就一个文件，所以通常不设置专门的输出文件夹，而这里做开发，通常会有很多个文件加入一个工程中，编译过程中会产生很多中间文件，因此设置专门的文件夹 LIS 和 OBJ 用来保存中间文件。

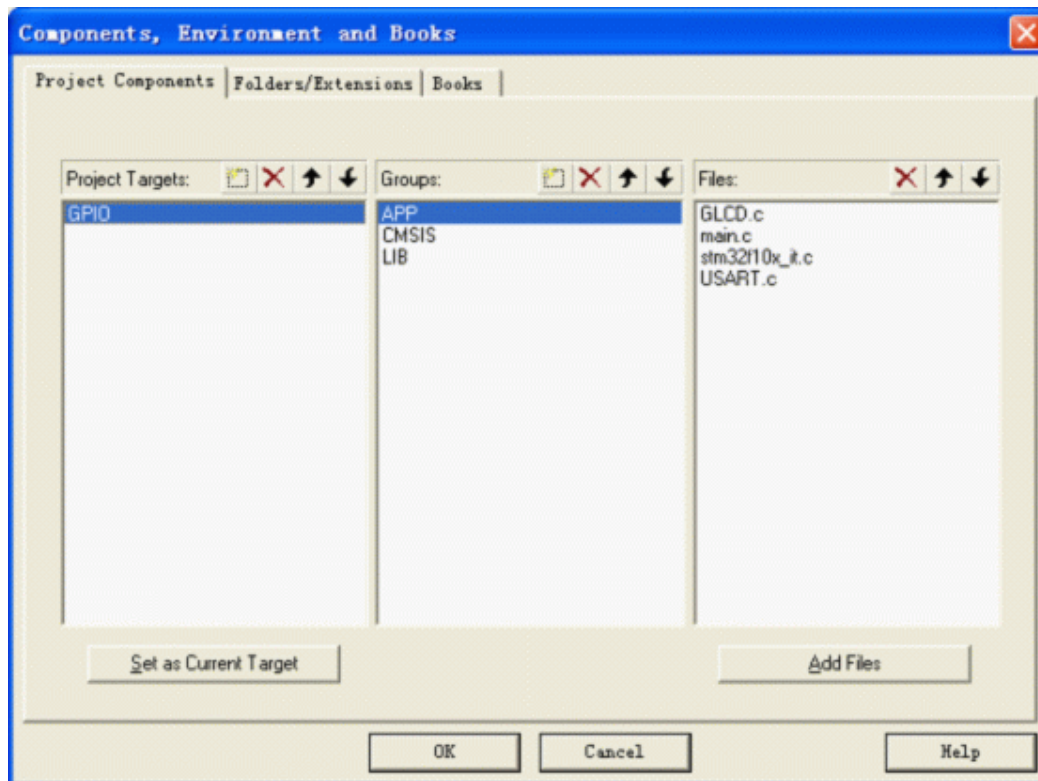
下面就将设置简单描述一下。

将复到过来的 GPIO 根目录下的所有文件删除，因为我们要学着自己建立工程。

用菜单 Project-->New uVision Project...建立新的工程，选择目标器件为 STM32103VC，这个过程与建立 51 单片机的工程没有什么区别，这里就偷点懒，不上图了。接下来看一看怎么设置。



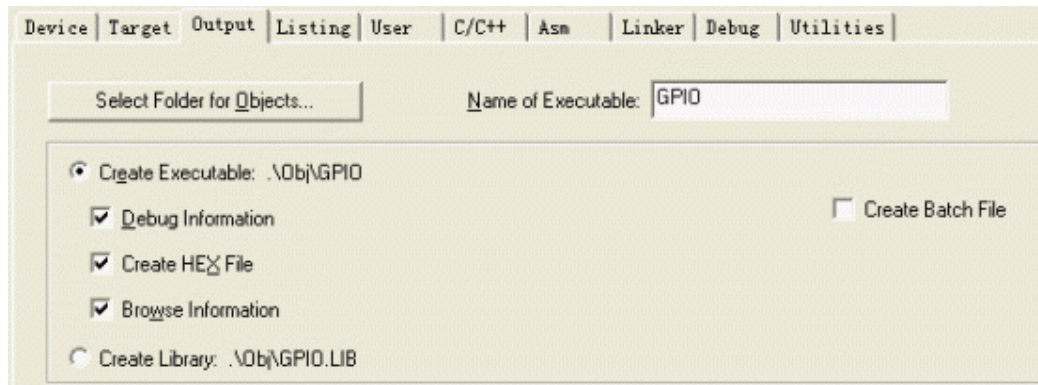
点那个品字形，打开对话框



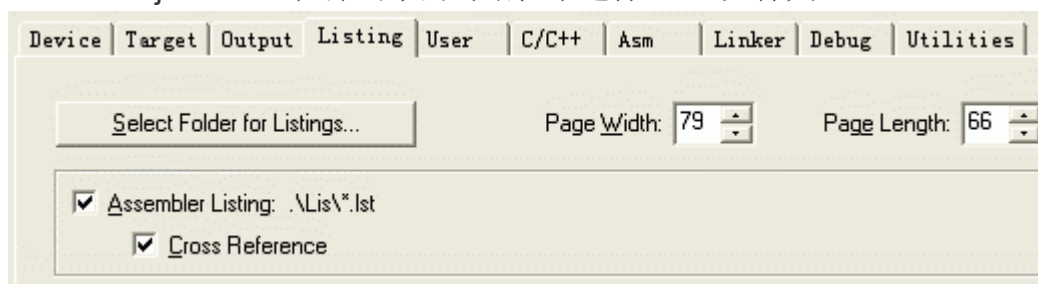
这里就给个图了，相信有一定操作基础的人应该会用。顺便提一下，原来用 VC 或者 IAR 时总觉得它们的一个功能：就是建立一个 Debug 组和 Release 组，这个功能挺好的，从这个图可在 Keil 里也是一样可以建的。

将刚才那个文件夹图中 CMSIS 中的文件加入 CMSIS 组，一共 3 个，其中 \Source\CMSIS\Core\CM3 有两个 C 语言源程序文件全部加入，另外还有一个在 \Source\CMSIS\Core\CM3\startup\arm 文件夹中，这个文件夹中有 4 个 .s 文件，我们选择其中的 startup_stm32f10x_hd.s 文件。这是根据项目所用 CPU 来选择的，我们用的 CPU 是 103VC 的，属于高密度的芯片，所以选这个。

至于 LIB 中的文件，就在这儿： \Source\STM32F10x_StdPeriph_Driver\src 啦。这里有很多个文件，把什么文件加进去呢？怕麻烦的话，把所有文件全部加进去，这并不会增加编译后的代码量，但会增加很多的编译时间。

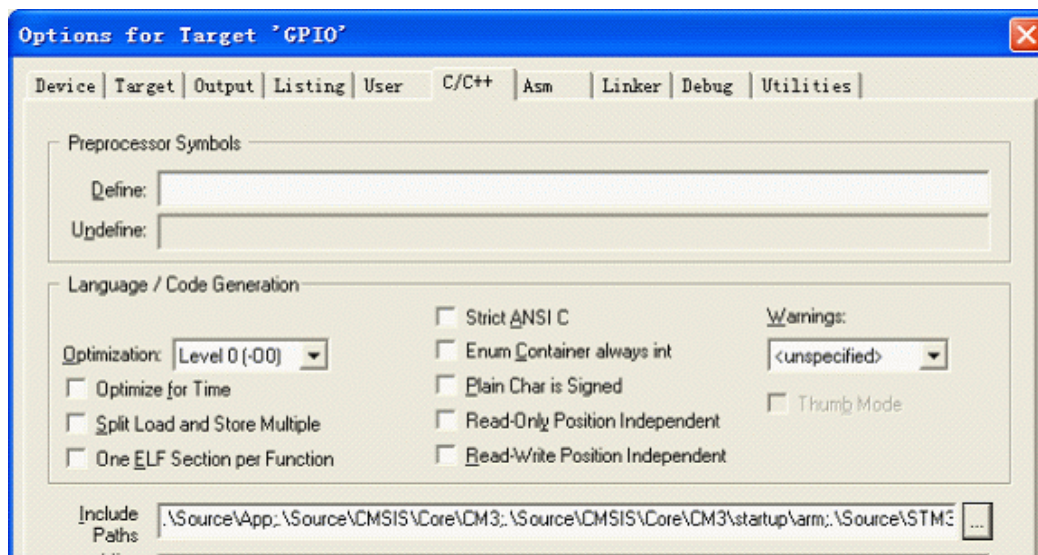


接下来设定目标输出文件夹。上面这个图怎么出来的就不说啦，单击“Select Folder for Objects...”，在弹出来的对话框中选择 OBJ 文件夹。

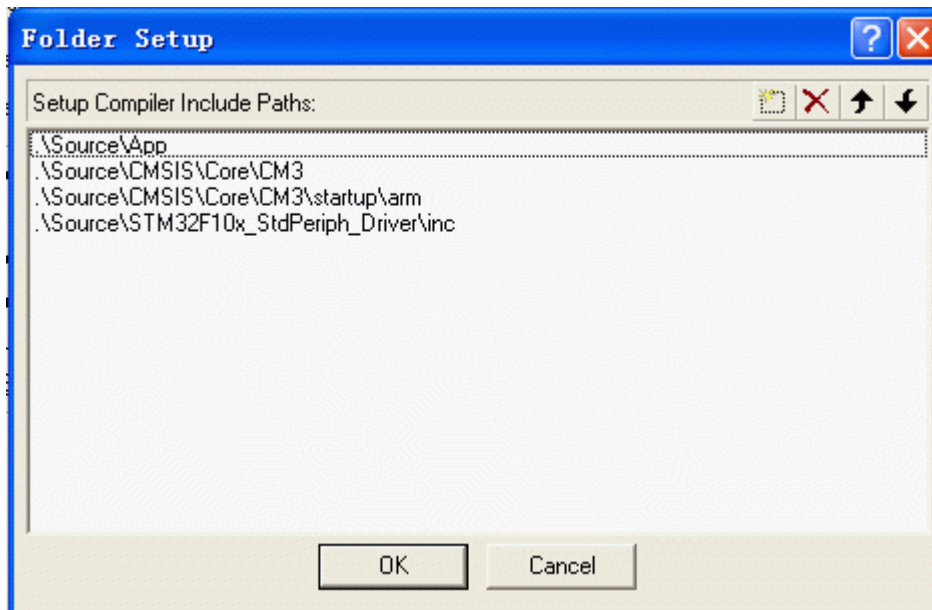


同样方法，选择 List 文件的输出文件夹。

设置好后，如果直接编译是不行的，会出错。还需要提供头文件所在位置。单击 c/C++ 标签页。



第一次进入时 Include Paths 文本框中是空白的，点击其后的按钮打开对话框，增加相应的路径



这样路径就设好了。单击 **OK**，回到上一界面，然后再单击 **OK**，退出设置，即可编译、链接。

下一会要试一试新的 3.12 版的库效果如何了。

STM32 学习笔记 (3)

升级库

光盘中所带的例子是 3.10 的，另外还有一个 3.12 的，我试着将 3.12 的库替代原来的库，还真有问题，下面就简述问题及解决方法。

(1) 将库文件解压

库文件名是：`stm32f10x_stdperiph_lib.zip`，解压后放在任意一个文件夹中。

(2) 由于原作者做了很好的规划，每一个项目中都分成三个文件夹，并且在 `source` 文件夹中又做了 3 个文件夹，其中 `APP` 文件夹是放自己写的文件的，其他的两个是从库中复制过来的，因此，想当然地把 3.1.2 版本中相同的两个文件夹：

`CMSIS` 和 `STM32F10x_StdPeriph_Driver` 直接复制过来，以为一切 **OK**，结果一编译，出来一堆错误。

其中有错误：

```
Source\App\main.c(7): error: #20: identifier "GPIO_InitTypeDef" is undefined
```

....

还有大量的警告：

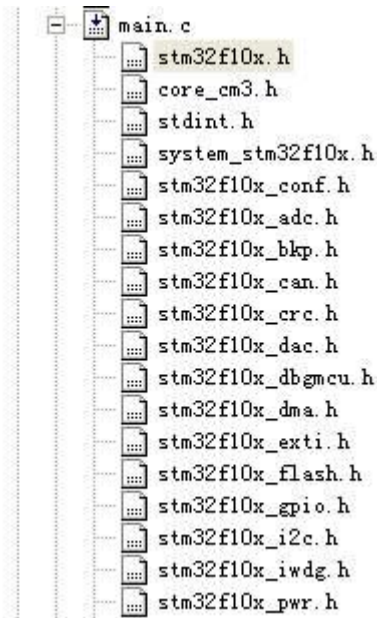
```
Source\STM32F10x_StdPeriph_Driver\src\stm32f10x_flash.c(130): warning: #223-D: function "assert_param" declared implicitly
```

看了看，在 `APP` 文件夹中还有一些不属于自己的东西：

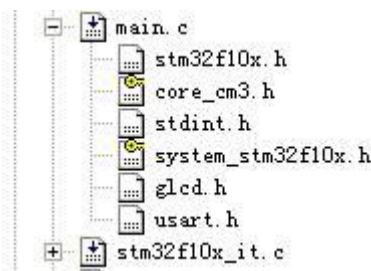
`stm32f10x_conf.h`, `stm32f10x_it.h`, `stm32f10x_it.c`，打开一看，果然是 3.10 版本的，没说的，换。。。 ，找到 `STM32F10x_StdPeriph_Lib_V3.1.2\Proje`

ctTemplate 文件夹，用里面的同样的文件替换掉这几个文件，这回应该万事大吉了吧。

再一看，依然如故，，没办法了，只好细细研究了。通过观察，发现原来可以编译通过的工程，在 main.c 下面挂满了.h 文件，而这个通不过的，则少得很。



这是编译能通过的工程



这是编译通不过的工程

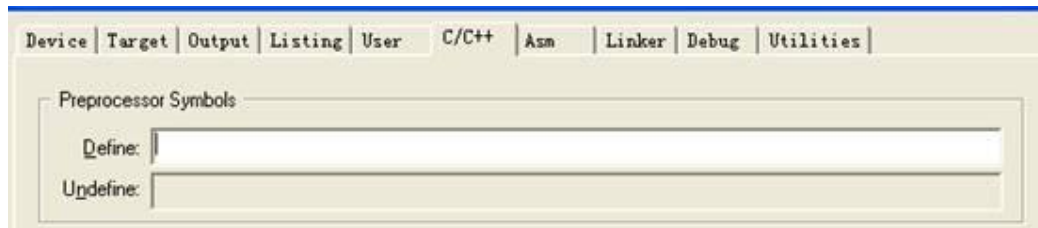
显然，有些文件没有被包含进来。一点一点跟踪，发现大部分的头文件都包含在 stm32f10x_conf.h 中，而这个文件又出现在 stm32f10x.h 中，其中有这样的一行：

```
#ifndef USE_STDPERIPH_DRIVER
    #include "stm32f10x_conf.h"
#endif
```

看来，是这个 USE_STDPERIPH_DRIVER 没有被定义啊，于是，人为地去掉条件：

```
//#ifndef USE_STDPERIPH_DRIVER
    #include "stm32f10x_conf.h"
//#endif
```

再次编译，果然就 OK 了。可是，可是，也不能就这么去掉啊，怎么办呢？万能的网啊，一搜果然就有了。



到设置 C/C++ 页面

在那个 define 中加入“USE_STDPERIPH_DRIVER,STM32F10X_HD”

当然，去掉条件编译前面的注释，回到原样。

再次编译，一切顺利。可是，原来的工程例子也没有加这个啊，怎么回事呢？再次打开原来的例子，找到 stm32f10x.h，可以看到有这么一行：

```
#if !defined USE_STDPERIPH_DRIVER
/**
 * @brief Comment the line below if you wil
 * In this case, these drivers will not be
 * be based on direct access to peripherals
 */
#define USE_STDPERIPH_DRIVER
#endif
```

而新的 stm32f10x.h 中则是这样的：

```
#if !defined USE_STDPERIPH_DRIVER
/**
 * @brief Comment the line below if you will not use the
 * In this case, these drivers will not be included and
 * be based on direct access to peripherals registers
 */
/*#define USE_STDPERIPH_DRIVER*/
#endif
```

原来那个 3.1.0 版的 stm32f10x.h 被人为地修改了一下，所以，不在 define 中定义也不要紧，而新升级的 3.1.2 则不行了。

至此，简单的升级搞定。

本文见于好多地方，但查询后未能确定其原始出处及作者，故这里说明是转贴，但作者和原始出处信息就无法提供了，如果原作者看到请跟贴说明，知情者也请跟贴说明。

ARM 中的 RO、RW 和 ZI DATA

一直以来对于 ARM 体系中所描述的 RO，RW 和 ZI 数据存在似是而非的理解，这段时间对其仔细了解了一番，发现了一些规律，理解了一些以前书本上有的但是不理解的东西，我想应该有不少人也有和我同样的困惑，因此将我的一些关于 RO，RW 和 ZI 的理解写出来，希望能对大家有所帮助。

要了解 RO，RW 和 ZI 需要首先了解以下知识：

ARM 程序的组成

此处所说的“ARM 程序”是指在 ARM 系统中正在执行的程序，而非保存在 ROM

中的 bin 映像 (image) 文件, 这一点请注意区别。

一个 ARM 程序包含 3 部分: RO, RW 和 ZI

RO 是程序中的指令和常量

RW 是程序中的已初始化变量

ZI 是程序中的未初始化的变量

由以上 3 点说明可以理解为:

RO 就是 readonly,

RW 就是 read/write,

ZI 就是 zero

ARM 映像文件的组成

所谓 ARM 映像文件就是指烧录到 ROM 中的 bin 文件, 也称为 image 文件。以下用 Image 文件来称呼它。

Image 文件包含了 RO 和 RW 数据。

之所以 Image 文件不包含 ZI 数据, 是因为 ZI 数据都是 0, 没必要包含, 只要程序运行之前将 ZI 数据所在的区域一律清零即可。包含进去反而浪费存储空间。

Q: 为什么 Image 中必须包含 RO 和 RW?

A: 因为 RO 中的指令和常量以及 RW 中初始化过的变量是不能像 ZI 那样“无中生有”的。

ARM 程序的执行过程

从以上两点可以知道, 烧录到 ROM 中的 image 文件与实际运行时的 ARM 程序之间并不是完全一样的。因此就有必要了解 ARM 程序是如何从 ROM 中的 image 到达实际运行状态的。

实际上, RO 中的指令至少应该有这样的功能:

1. 将 RW 从 ROM 中搬到 RAM 中, 因为 RW 是变量, 变量不能存在 ROM 中。
2. 将 ZI 所在的 RAM 区域全部清零, 因为 ZI 区域并不在 Image 中, 所以需要程序根据编译器给出的 ZI 地址及大小来将相应得 RAM 区域清零。ZI 中也是变量, 同理: 变量不能存在 ROM 中

在程序运行的最初阶段, RO 中的指令完成了这两项工作后 C 程序才能正常访问变量。否则只能运行不含变量的代码。

说了上面的可能还是有些迷糊, RO, RW 和 ZI 到底是什么, 下面我将给出几个例子, 最直观的来说明 RO, RW, ZI 在 C 中是什么意思。

1; RO

看下面两段程序, 他们之间差了一条语句, 这条语句就是声明一个字符常量。因此按照我们之前说的, 他们之间应该只会在 RO 数据中相差一个字节 (字符常量为 1 字节)。

Prog1:

```
#include <stdio.h>
```

```
void main(void)
```



```

{
;
}
Prog2:
#include <stdio.h>
const char a = 5;
void main(void)
{
;
}

```

Prog1 编译出来后的信息如下:

```

=====
=====
Code   RO Data   RW Data   ZI Data   Debug
948    60         0         96        0         Grand Totals
=====
=====
Total RO   Size(Code + RO Data)           1008 ( 0.98kB)
Total RW   Size(RW Data + ZI Data)        96   ( 0.09kB)
Total ROM Size(Code + RO Data + RW Data)  1008 ( 0.98kB)
=====
=====

```

Prog2 编译出来后的信息如下:

```

=====
=====
Code   RO Data   RW Data   ZI Data   Debug
948    61         0         96        0         Grand Totals
=====
=====
Total  RO   Size(Code + RO Data)           1009 ( 0.99kB)
Total  RW   Size(RW Data + ZI Data)        96   ( 0.09kB)
Total  ROM Size(Code + RO Data + RW Data)  1009 ( 0.99kB)
=====
=====

```

以上两个程序编译出来后的信息可以看出:

Prog1 和 Prog2 的 RO 包含了 Code 和 RO Data 两类数据。他们的唯一区别就是 Prog2 的 RO Data 比 Prog1 多了 1 个字节。这正和之前的推测一致。如果增加的是一条指令而不是一个常量, 则结果应该是 Code 数据大小有差别。

2; RW

同样再看两个程序，他们之间只相差一个“已初始化的变量”，按照之前所讲的，已初始化的变量应该是算在 RW 中的，所以两个程序之间应该是 RW 大小有区别。

Prog3:

```
#include <stdio.h>
```

```
void main(void)
```

```
{
```

```
;
```

```
}
```

Prog4:

```
#include <stdio.h>
```

```
char a = 5;
```

```
void main(void)
```

```
{
```

```
;
```

```
}
```

Prog3 编译出来后的信息如下:

```
=====
=====
Code      RO Data    RW Data    ZI Data    Debug
948        60         0          96         0      Grand Totals
=====
=====
Total    RO    Size(Code + RO Data)          1008 ( 0.98kB)
Total    RW    Size(RW Data + ZI Data)       96   ( 0.09kB)
Total    ROM  Size(Code + RO Data + RW Data) 1008 ( 0.98kB)
=====
=====
```

Prog4 编译出来后的信息如下:

```
=====
=====
Code      RO Data    RW Data    ZI Data    Debug
948        60         1          96         0      Grand
Totals
=====
=====
Total    RO    Size(Code + RO Data)          1008 ( 0.98kB)
```

```
Total RW Size(RW Data + ZI Data) 97 ( 0.09kB)
Total ROM Size(Code + RO Data + RW Data) 1009 ( 0.99kB)
```

```
=====
=====
```

可以看出 Prog3 和 Prog4 之间确实只有 RW Data 之间相差了 1 个字节, 这个字节正是被初始化过的一个字符型变量“a”所引起的。

3; ZI

再看两个程序, 他们之间的差别是一个未初始化的变量“a”, 从之前的了解中, 应该可以推测, 这两个程序之间应该只有 ZI 大小有差别。

Prog3:

```
#include <stdio.h>
void main(void)
{
;
}
```

Prog4:

```
#include <stdio.h>
char a;
void main(void)
{
;
}
```

Prog3 编译出来后的信息如下:

```
=====
=====
Code RO Data RW Data ZI Data Debug Grand
948 60 0 96 0
Totals
```

```
=====
=====
Total RO Size(Code + RO Data) 1008 ( 0.98kB)
Total RW Size(RW Data + ZI Data) 96 ( 0.09kB)
Total ROM Size(Code + RO Data + RW Data) 1008 ( 0.98kB)
```

```
=====
=====
```

Prog4 编译出来后的信息如下:

```
=====
=====
```

Code	RO Data	RW Data	ZI Data	Debug	Grand
948	60	0	97	0	Grand
Totals					

```
=====
=====
Total  RO   Size(Code + RO Data)                1008 ( 0.98kB)
Total  RW   Size(RW Data + ZI Data)             97   ( 0.09kB)
Total  ROM  Size(Code + RO Data + RW Data)      1008 ( 0.98kB)
=====
=====
```

编译的结果完全符合推测，只有 ZI 数据相差了 1 个字节。这个字节正是未初始化的一个字符型变量“a”所引起的。

注意：如果一个变量被初始化为 0，则该变量的处理方法与未初始化变量一样放在 ZI 区域。

即：ARM C 程序中，所有的未初始化变量都会被自动初始化为 0。

总结：

- 1; C 中的指令以及常量被编译后是 RO 类型数据。
- 2; C 中的未被初始化或初始化为 0 的变量编译后是 ZI 类型数据。
- 3; C 中的已被初始化成非 0 值的变量编译后是 RW 类型数据。

附：

程序的编译命令（假定 C 程序名为 tst.c）：

```
armcc -c -o tst.o tst.c
```

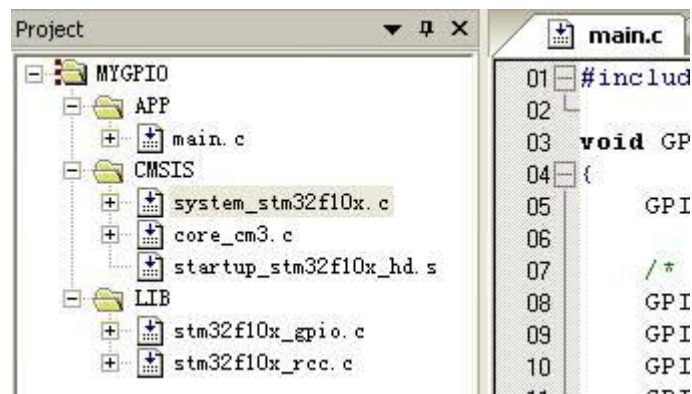
```
armlink -noremove -elf -nodebug -info totals -info sizes -map -list aa.map -o tst.elf tst.o
```

编译后的信息就在 aa.map 文件中。

ROM 主要指：NAND Flash，Nor Flash

RAM 主要指：PSRAM，SDRAM，SRAM，DDRAM

继续学习中，先把开发板自带一个例子做了些精简，以免看得吓人。。。。



就是这个，让 PORTD 上接的 4 个 LED 分别点亮。

开始研究代码

```
int main(void)
{
    Init_All_Periph();
    .....
```

看到这一行，开始跟踪，于是又看到了下面的内容

```
void Init_All_Periph(void)
{
    RCC_Configuration();
    .....
```

继续跟踪

```
void RCC_Configuration(void)
{
    SystemInit();
    .....
```

这行代码在 system_stm32f10x.c 中找到了。

```
void SystemInit (void)
{
    /* Reset the RCC clock configuration to the default reset state(for
    debug purpose) */
    /* Set HSION bit */
    RCC->CR |= (uint32_t)0x00000001;
    /* Reset SW, HPRE, PPRE1, PPRE2, ADCPRE and MCO bits */
#ifdef STM32F10X_CL
    RCC->CFGR &= (uint32_t)0xF8FF0000;
#else
    RCC->CFGR &= (uint32_t)0xF0FF0000;
#endif /* STM32F10X_CL */

    /* Reset HSEON, CSSON and PLLON bits */
    RCC->CR &= (uint32_t)0xFE6FFFFF;
    /* Reset HSEBYP bit */
    RCC->CR &= (uint32_t)0xFFBFFFFF;
    /* Reset PLLSRC, PLLXTPRE, PLLMUL and USBPRE/OTGFSPRE bits */
    RCC->CFGR &= (uint32_t)0xFF80FFFF;
#ifdef STM32F10X_CL
    /* Disable all interrupts and clear pending bits */

```

```

RCC->CIR = 0x009F0000;
#else
/* Reset PLL2ON and PLL3ON bits */
RCC->CR &= (uint32_t)0xEBFFFFFF;
/* Disable all interrupts and clear pending bits */
RCC->CIR = 0x00FF0000;
/* Reset CFGR2 register */
RCC->CFGR2 = 0x00000000;
#endif /* STM32F10X_CL */

/* Configure the System clock frequency, HCLK, PCLK2 and PCLK1 prescalers */
/* Configure the Flash Latency cycles and enable prefetch buffer */
SetSysClock();
}

```

这一长串的又是什么，如何来用呢？看来，偷懒是不成的了，只能回过头去研究 STM32 的时钟构成了。

相当的复杂。

系统的时钟可以有 3 个来源：内部时钟 HSI，外部时钟 HSE，或者 PLL（锁相环模块）的输出。它们由 RCC_CFGR 寄存器中的 SW 来选择。

SW（1：0）：系统时钟切换

由软件置‘1’或清‘0’来选择系统时钟源。在从停止或待机模式中返回时或直接或间接作为系统时钟的 HSE 出现故障时，由硬件强制选择 HSI 作为系统时钟（如果时钟安全系统已经启动）

00：HSI 作为系统时钟；

01：HSE 作为系统时钟；

10：PLL 输出作为系统时钟；

11：不可用。

////////////////////////////////////

PLL 的输出直接送到 USB 模块，经过适当的分频后得到 48M 的频率供 USB 模块使用。

系统时钟的一路被直接送到 I2S 模块；另一路经过 AHB 分频后送出，送往各个系统，其中直接送往 SDI，FMSC，AHB 总线；8 分频后作为系统定时器时钟；经过 APB1 分频分别控制 PLK1、定时器 TIM2~TIM7；经过 APB2 分频分别控制 PLK2、定时器 TIM1~TIM8、再经分频控制 ADC；

由此可知，STM32F10x 芯片的时钟比之于 51、AVR、PIC 等 8 位机要复杂复多，因此，我们立足于对着芯片手册来解读程序，力求知道这些程序代码如何使用，

为何这么样使用，如果自己要改，可以修改哪些部分，以便自己使用时可以得心应手。

单步执行，看一看哪些代码被执行了。

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



这是 RCC_CR 寄存器，由图可见，HSION 是其 bit 0 位。

HSION: 内部高速时钟使能
由软件置' 1' 或清零。

当从待机和停止模式返回或用作系统时钟的外部 4-25MHz 时钟发生故障时，该位由硬件置' 1' 来启动内部 8MHz 的 RC 振荡器。当内部 8MHz 时钟被直接或间接地用作或被选择将要作为系统时钟时，该位不能被清零。

- 0: 内部 8MHz 时钟关闭;
- 1: 内部 8MHz 时钟开启。

```
////////////////////////////////////
//
```

```
/* Reset SW, HPRE, PPRE1, PPRE2, ADCPRE and MCO bits */
#ifndef STM32F10X_CL
RCC->CFGR &= (uint32_t)0xF8FF0000;
```



位31:27 保留，始终读为0。

这是 RCC_CFGR 寄存器

该程序清零了 MCO[2:0] 这三位，和 ADCPRE[1:0], ppre2[2:0], PPRE1 [2:0] , HPRE [3: 0] , SWS [1: 0] 和 SW [1: 0] 这 16 位。

/*

MCO: 微控制器时钟输出，由软件置' 1' 或清零。

0xx: 没有时钟输出;
100: 系统时钟(SYSCLK)输出;
101: 内部 8MHz 的 RC 振荡器时钟输出;
110: 外部 4-25MHz 振荡器时钟输出;
111: PLL 时钟 2 分频后输出。

*/

```
/* Reset HSEON, CSSON and PLLON bits */
```

```
RCC->CR &= (uint32_t)0xFE6FFFF;
```

清零了 PLLON, HSEBYP, HSERDY 这 3 位。

```
/* Reset HSEBYP bit */
```

```
RCC->CR &= (uint32_t)0xFFBFFFF;
```

清零了 HSEBYP 位 ///??? 为什么不一次写??

HSEBYP: 外部高速时钟旁路, 在调试模式下由软件置'1'或清零来旁路外部晶体振荡器。只有在外部 4-25MHz 振荡器关闭的情况下, 才能写入该位。

0: 外部 4-25MHz 振荡器没有旁路;

1: 外部 4-25MHz 外部晶体振荡器被旁路。

所以要先清 HSEON 位, 再清该位。

```
/* Reset PLLSRC, PLLXTPRE, PLLMUL and USBPRE/OTGFSPRE bits */
```

```
RCC->CFGR &= (uint32_t)0xFF80FFFF;
```

清零了: USBPRE, PLLMUL, PLLXTPR, PLLSRC 共 7 位

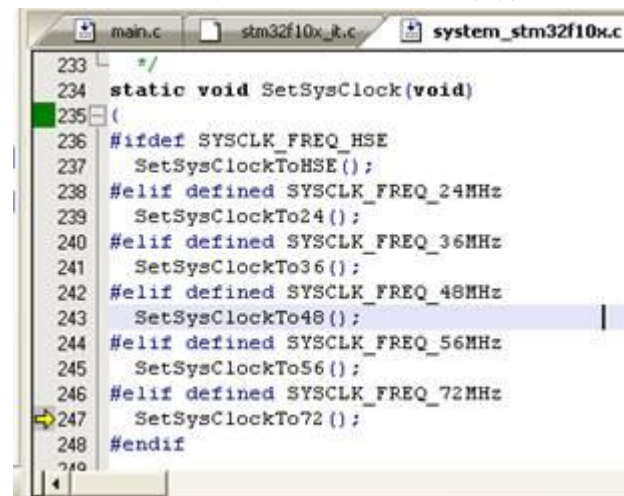
```
/* Disable all interrupts and clear pending bits */
```

```
RCC->CIR = 0x009F0000;
```

////这个暂不解读

```
SetSysClock();
```

跟踪进入该函数, 可见一连串的条件编译:



```
233 */
234 static void SetSysClock(void)
235 {
236     #ifdef SYSCLK_FREQ_HSE
237         SetSysClockToHSE();
238     #elif defined SYSCLK_FREQ_24MHz
239         SetSysClockTo24();
240     #elif defined SYSCLK_FREQ_36MHz
241         SetSysClockTo36();
242     #elif defined SYSCLK_FREQ_48MHz
243         SetSysClockTo48();
244     #elif defined SYSCLK_FREQ_56MHz
245         SetSysClockTo56();
246     #elif defined SYSCLK_FREQ_72MHz
247         SetSysClockTo72();
248     #endif
249 }
```

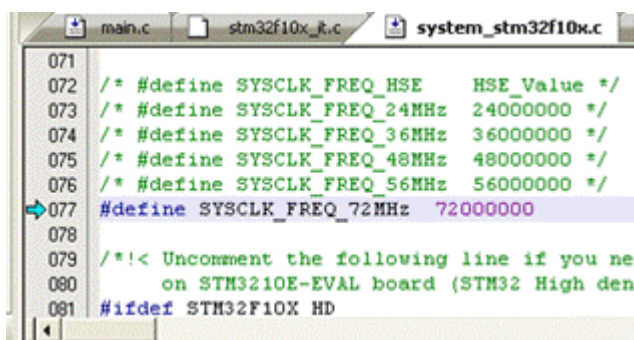
单步运行, 执行的是:

```
#elif defined SYSCLK_FREQ_72MHz
```



```
SetSysClockTo72();
```

为何执行该行呢，找到 SYSCLK_FREQ_**的相关定义，如下图所示。



```
071
072 /* #define SYSCLK_FREQ_HSE    HSE_Value */
073 /* #define SYSCLK_FREQ_24MHz  24000000 */
074 /* #define SYSCLK_FREQ_36MHz  36000000 */
075 /* #define SYSCLK_FREQ_48MHz  48000000 */
076 /* #define SYSCLK_FREQ_56MHz  56000000 */
077 #define SYSCLK_FREQ_72MHz    72000000
078
079 /*!< Uncomment the following line if you need
080    on STM3210E-EVAL board (STM32 High den
081    #ifdef STM32F10X HD
```

这样就得到了我们所要的一个结论:如果要更改系统工作频率,只需要在这里更改就可以了。

可以继续跟踪进入这个函数来观察如何将工作频率设定为 72MHz 的。

```
static void SetSysClockTo72(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;

    /* SYSCLK, HCLK, PCLK2 and PCLK1 configuration -----
    -----*/
    /* Enable HSE */
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);
    //开启 HSE
    /* Wait till HSE is ready and if Time out is reached exit */
    do
    {
        HSEStatus = RCC->CR & RCC_CR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSEStartUp_TimeOut));
    //等待 HSE 确实可用，这有个标志，即 RCC_CR 寄存器中的 HSERDY 位 (bit 17)，
    这个等待不会无限长，有个超时策略，即每循环一次计数器加 1，如果计数的次数超过 HSEStartUp_TimeOut，就退出循环，而这个 HSEStartUp_TimeOut 在 stm32f10x.h 中定义，
    #define HSEStartUp_TimeOut    ((uint16_t)0x0500) /*!< Time out for HSE
    start up */
    ////////////////////////////////////////////
    ////////////////////////////////////////////
    if ((RCC->CR & RCC_CR_HSERDY) != RESET)
```

```

{
    HSEStatus = (uint32_t)0x01;
}
else
{
    HSEStatus = (uint32_t)0x00;
}
///再次判断 HSERDY 标志位，并据此给 HSEStatus 变量赋值。
if (HSEStatus == (uint32_t)0x01)
{
    /* Enable Prefetch Buffer */
    FLASH->ACR |= FLASH_ACR_PRFTBE;

    /* Flash 2 wait state */
    FLASH->ACR &= (uint32_t)((uint32_t)~FLASH_ACR_LATENCY);
    FLASH->ACR |= (uint32_t)FLASH_ACR_LATENCY_2;

    /* HCLK = SYSCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;
    //找到定义: #define RCC_CFGR_HPRE_DIV1 ((uint32
_t)0x00000000) /*!< SYSCLK not divided */

    /* PCLK2 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;
    //找到定义: #define RCC_CFGR_PPRE2_DIV1 ((uint32_t)
0x00000000) /*!< HCLK not divided */

    /* PCLK1 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV2;
    //找到定义: #define RCC_CFGR_PPRE1_DIV2 ((uint32_t)0
x00000400) /*!< HCLK divided by 2 */

#ifdef STM32F10X_CL
    .....
#else
    /* PLL configuration: PLLCLK = HSE * 9 = 72 MHz */

```

```

RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_PLLSRC | RCC_CFGR_PL
LXTPRE |
                                RCC_CFGR_PLLMULL));
RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLMULL9);
#endif /* STM32F10X_CL */
//以上是设定 PLL 的倍频系数为 9，也就是说，这个 72M 是在外部晶振为 8M 时
得到的。
/* Enable PLL */
RCC->CR |= RCC_CR_PLLON;

/* Wait till PLL is ready */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Select PLL as system clock source */
RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;
/* Wait till PLL is used as system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) != (uint32_t)0x08)
{
}
}
else
{ /* If HSE fails to start-up, the application will have wrong cloc
k
configuration. User can add here some code to deal with this
error */
/* Go to infinite loop */
while (1)
{
}
}
}

```

至此，我们可以归纳几条：

- (1) 时钟源有 3 个
- (2) 开机时默认是 HSI 起作用，可以配置为所要求的任意一个时钟
- (3) 配置时必须按一定的顺序来打开或都关闭一些位，并且各时钟起作用有一定的时间，因此要利用芯片内部的标志位来判断是否可以执行下一步。

- (4) 如果外部时钟、PLL 输出失效，系统可以自动回复到 HSI（开启时钟安全系统）
- (5) HSI 的频率准确度可以达到 $\pm 1\%$ ，如果有必要时，还可以用程序来调整这个频率，可调的范围大致在 200KHz 左右。

最后让我们来感受一下劳动的果实吧--试着改改频率看有何反应。

为查看更改后的效果，先记录更改前的数据。将调试切换到仿真，在第一条：

```
Delay(0xAFFFF);
```

指令执行前后，分别记录下 Status 和 Sec

```
Status:2507 3606995
```

```
Sec:0.00022749 0.05028982
```

将振荡频率更改为 36MHz，即

...

```
#define SYSCLK_FREQ_36MHz 36000000 //去掉该行的注释
```

```
/* #define SYSCLK_FREQ_48MHz 48000000 */
```

```
/* #define SYSCLK_FREQ_56MHz 56000000 */
```

```
/*#define SYSCLK_FREQ_72MHz 72000000*/ //将该行加上注释
```

再次运行，结果如下：

```
Status:2506 3606994
```

```
Sec:0.00008478 0.10036276
```

基本上是延时时间长了一倍。改成硬件仿真，将代码写入板子，可以看到 LED 闪烁的频率明显变慢了。

STM32 学习笔记（6）-I/O 的简单研究

前面的例子研究了时钟，接下来就来了解一下引脚的情况

Main.c 中，有关 I/O 口的配置代码如下：

```
void GPIO_Configuration(void)
```

```
{
```

```
    GPIO_InitTypeDef GPIO_InitStructure;
```

```
    /* Configure IO connected to LD1, LD2, LD3 and LD4 leds *****  
    ******/
```

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_  
10 | GPIO_Pin_11;
```

```
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
```

```
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

```
    GPIO_Init(GPIOD, &GPIO_InitStructure);
```

这几行代码是将 GPIOD 的第 8，9，10 和 11 引脚配置成输出，并且还可以设定输出引脚的速度（驱动能力？），这里设定为 50MHz，这应该是常用的，还有

可以设置为 2MHz 的。那么如何将引脚设置成输入呢？查看电路原理图，GPIO D.0~GPIO.4 是接一个摇杆的 5 个按钮的，因此，下面尝试着将它们设置成为输入端。

```
GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4;
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
```

```
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

第 1 行和第 3 行完全是照抄，第 2 行那个 GPIO_Mode_IN_FLOATING 是在 stm32f10x_gpio.h 中找到的。

```
044 /* Configuration Mode enumeration -
045 typedef enum
046 { GPIO_Mode_AIN = 0x0,
047   GPIO_Mode_IN_FLOATING = 0x04,
048   GPIO_Mode_IPD = 0x28,
049   GPIO_Mode_IPU = 0x48,
050   GPIO_Mode_Out_OD = 0x14,
051   GPIO_Mode_Out_PP = 0x10,
052   GPIO_Mode_AF_OD = 0x1C,
053   GPIO_Mode_AF_PP = 0x18
054 }GPIO_Mode_TypeDef;
055
```

当然是因为这里还有 GPIO_Mode_Out_PP，所以猜测应该是它了。至于还有其他那么多的符号就不管了。

定义完成，编译完全通过，那就接下来准备完成下面的代码了。

```
int main(void)
{
    Init_All_Periph();
    while(1)
    {
        if( GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_0)) //1
        {
            GPIO_ResetBits(GPIOD, GPIO_Pin_8);
        }
        else
        {
            /* Turn on LD1 */
            GPIO_SetBits(GPIOD, GPIO_Pin_8);
            /* Insert delay */
        }
    }
    .....
}
```

标号为 1 的行显然其作用是判断 GPIOD.0 引脚是 0 还是 1。这个函数是在 stm32f10x_gpio.c 中找到的。

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

```

{
    uint8_t bitstatus = 0x00;

    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GET_GPIO_PIN(GPIO_Pin));

    if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)Bit_RESET)
    {
        bitstatus = (uint8_t)Bit_SET;
    }
    else
    {
        bitstatus = (uint8_t)Bit_RESET;
    }
    return bitstatus;
}

```

虽然程序还有很多符号看不懂（没有去查），但凭感觉它应该是对某一个引脚的状态进行判断，因为这个函数的类型是 `uint8_t`，估计 `stm32` 没有 `bit` 型函数（需要验证），所以就用了 `uint8_t` 型了），如果是读的端口的值，应该用 `uint16_t` 型。这一点在下面也可以得到部分的验证：

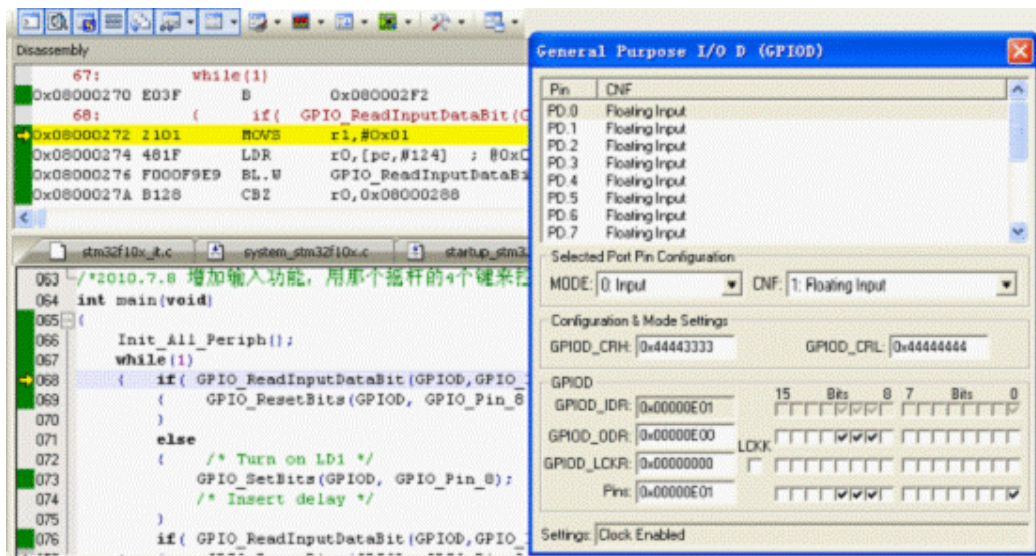
```

uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx)
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx)

```

这些函数是读引脚及输出寄存器的数据的。

再次编译，也是顺利通过，依法炮制，将其他三个引脚输入控制 `LED` 的代码也写上，为保险起见，先用软件仿真，免得反复擦写 `FLASH`（顺便说一句，目前还没有搞定将代码写入 `RAM` 及从 `RAM` 中执行，惭愧）



进入仿真后打开外围部件接口，单步执行，果然如同设想那样运作了，单击 Pins 0 后面的勾，再次运行，果然 PIN8 后面的勾没了。做到这里，就感觉到用 keil 的好处了，这块熟啊，几乎没有花时间在上面，一用就成了。

至此，按我的习惯，要翻开 STM32F 的数据手册，研究一下其 IO 端口了。下面是数据手册中的一段话：

每个 GPIO 端口有两个 32 位配置寄存器(GPIOx_CRH, GPIOx_CRL)，两个 32 位数据寄存器(GPIOx_IDR, GPIOx_ODR)，一个 32 位置位/复位寄存器(GPIOx_BSRR)，一个 16 位复位寄存器(GPIOx_BRR)和一个 32 位锁定寄存器(GPIOx_LCKR)。

根据数据手册中列出的每个 I/O 端口的特定硬件特征，GPIO 端口的每个位可以由软件分别配置成多种模式。

- 输入浮空
- 输入上拉
- 输入下拉
- 模拟输入
- 开漏输出
- 推挽式输出
- 推挽式复用功能
- 开漏复用功能

当然，数据手册上关于 IO 端口的描述是很多很多的，我也只是大概地了解了一下，真正要设计产品时，肯定还要细看。但至少，知道了 IO 端口复位后处于浮空状态，也就是其电平状态由外围电路决定，这很重要，如果设计工业品的话，这是必须要确定的；知道了 IO 引脚可以兼容 5V 电源；知道了在什么地方可以找到这些引脚在库中的定义而不必看着数据手册去控制那些位；也知道了这

些引脚的一些基本操作函数（连猜带蒙带测试应该可以搞定大部分功能），那么我心里基本就有底啦。