经过几天的学习，基本掌握了 STM32 的调试环境和一些基本知识。想拿出来与大家共享，笨教程本着最大限度简化删减 STM
入门的过程的思想，会把我的整个入门前的工作推荐给大家。就算是给网上的众多教程、笔记的一种补充吧，所以叫学前班
程。其中涉及产品一律隐去来源和品牌，以防广告之嫌。全部汉字内容为个人笔记。所有相关参考资料也全部列出。:lol

教程会分几篇，因为太长啦。今天先来说说为什么是它——我选择 STM32 的原因。

我对未来的规划是以功能性为主的，在功能和面积之间做以平衡是我的首要选择，而把运算放在第二位，这根我的专业有关
里面的运算其实并不复杂，在入门阶段想尽量减少所接触的东西。

不过说实话，对 DSP 的外设并和开发环境不满意，这是为什么 STM32 一出就转向的原因。下面是我自己做过的两块 DSP28 的
功能最小系统板，在做这两块板子的过程中发现要想尽力缩小 DSP 的面积实在不容易（目前只能达到 50mm×45mm，这还是
有其他器件的情况下），尤其是双电源的供电方式和 1.9V 的电源让人很头疼。

后来因为一个项目，接触了 LPC2148 并做了一块板子，发现小型的 ARM7 在外设够用的情况下其实很不错，于是开始搜集相
芯片资料，也同时对小面积的 AVR 和 51 都进行了大致的比较，这个时候发现了 CortexM3 的 STM32，比 2148 拥有更丰富和
活的外设，性能几乎是 2148 两倍（按照 MIPS 值计算）。正好 2148 我还没上手，就直接转了这款 STM32F103。

与 2811 相比较（核心 1.8V 供电情况下），135MHz×1MIPS。现在用 STM32F103，72MHz×1.25MIPS，性能是 DSP 的 66%，ST
2F103R 型（64 管脚）芯片面积只有 2811 的 51%，STM32F103C 型（48 管脚）面积是 2811 的 25%，最大功耗是 DSP 的 20%，
片价格是 DSP 的 30%。且有更多的串口，CAP 和 PWM，这是有用的。高端型号有 SDIO，理论上比 SPI 速度快。

由以上比较，准备将未来的拥有操作系统的高端应用交给 DSP 的新型浮点型单片机 28335，而将所有紧凑型小型、微型
用交给 STM32。

的 STM32 学前班教程：怎么开发

sw 笨笨的 STM32 学前班教程之二：怎么开发目前手头的入门阶段使用的开发器概述

该产品为简易 STM32 调试器和 DEMO 板一体化的调试学习设备，价格在一百多块。

2、硬件配置

仿真部分：USB 口，reset，指示灯，JTAG

DEMO 部分：4 按键（IO），4LED（IO），一个串口，启动方式跳线，所有引脚的焊盘（可自行焊接插针进行扩展）

DEMO 芯片：STM32F103C8T6（程序空间 64K）

参数和扩展：

注：学习的目标芯片是 STM32F103CBT6（7×7mm，128K flash，16K RAM）以及 STM32F103RET6（10×10mm，512K flash，64K

STM32-SK 的硬件连接方法（用板载调试器调试板载 DEMO）：

JP3、JP5 须全部短接

USB 通过电缆连接至 PC 的 USB

串口连接至 PC 的串口或者通过 USB 转串口电缆连接（力特 Z-TEC，USB2.0 与 RS232 转接电缆）

WindowsXP 自动安装驱动

安装完成后如果 DEMO 板里面有程序就会自动运行了。这是 ST-Link-II 的通用连接方法

以上是学习阶段比较方便的仿真器，进入工程阶段后准备换 J-Link V7 的仿真器进行开发。目前比较满意的产品：JLink v
USB 转串口：

购买后所需的改造：打开壳体，将 USB 的+5V 供电跟 JTAG20 针的第二脚 Vsupply 飞线，提供目标板 5V500mA 的供电。看
的特点：集成串口，拥有 20 针 JTAG 可以改造 Vspply 为供电接口，小巧好带，便宜。

常见的用于 STM32 单片机的仿真器分类

a) Ulink2：之前常用的仿真器。Keil 公司产品，之前专用于 ARM7，现扩展到 CortexM3，调试接口支持 JTAG 和 SWD，连接
PC 主机的 USB。现在这种调试器已经用的越来越少了。

b) ST-Link-II：ST 公司的仿真接口，支持 IAR EWARM，USB 1.1 全速，USB 电源供电，自适应目标系统 JTAG 电平 3.3V-5V
可向目标系统提供不大于 5V/200mA 电源。这种调试器不多见，但是许多调试器与目标板一体设计的学习板上常见。

c) J-Link V6/V7：SEGGER 公司产品，调试接口支持 JTAG 和 SWV（V7 速度是 V6 的 12 倍），USB 2.0 接口，通过 USB 供电，
载速度达到 720k byte/s， 与 IAR WEARM 无缝集成，宽目标板电压范围：1.2V-3.3V（V7 支持 5V），多核调试，给目标板
供 3.3V50mA 电源。这种调试器现在出现的越来越多，兼容性比较好（主要是指能够与 IAR WEARM 无缝集成这点），国内山
货和各种变种也很多。

6、目标板主要分为一体化设计（与调试器、供电整合）和单独设计两类，详细产品比较见豆皮的《如何选择 STM32 开发板

STM32 学前班教程之三：让 PC 工作

开发软件的选择

1、 软件与版本的选择

需求：支持 STLink2 或未来的 Jlink V7 调试接口（因为 STM32-SK 使用这个接口），能够找到去除软件限制的方法，最好具
中文版帮助和界面，最好带有纯软件仿真

2、 RealView MDK 3.23RPC（中国版）安装与去除限制

第一步：执行安装程序完成基本安装，最后选项选择加入虚拟硬件，便于纯软件调试。

第二步：执行软件，点击 File-->Licence Manager，复制 CID 的数据到破解器的 CID，其他选项如下图，然后点击 Generat

第三步：复制 LICO 的数据到软件的 LIC 框里面，点击 Add LIC。注意添加序列号后 Licence Manager 会算出这个号对应的
效期，如果到期会显示为红色，需要重新点击破解软件的 Generate，再算一个填进去就行了。

第四步：将 ST-LINKII-KEIL Driver 所需的文件（两个 DLL）拷贝到\Keil\ARM\BIN 下，替换原有文件。

第五步：打开 Keil 安装目录下的 TOOLS.INI 文件，在[ARM]、[ARMADS]、[KARM]项目下添加 TDRV7=BIN\ST-LINKII-KEIL.dll
ST LinkII Debugger")行，并保存修改。

第六步：打开 MDK，在项目的 options 设置的 Debug 选项中选择 ST LINKII Debugger，同时在 Utilities 的选项中选择 ST
NKII Debugger。

完成以上步骤，就完成了 ST-LINKII 的相关配置，可以作为调试器开始使用。注意：目前 ST-LINKII 不支持 Flash 菜单中的
ownload 和 Erase 命令，程序在使用 Start/Stop Debug Session 时自动载入 flash 中供调试。

3、 IAR EWARM 4.42A 安装与破解

第一步：开始/运行…/CMD 显示 DOS 界面，执行 iarid.exe>>ID.TXT 得到本机 ID 码，复制这个 ID 码，再执行 iarkg.exe I
码>>Lic422A.TXT，得到一组注册码。

第二步：使用 EWARM-EV-WEB-442A.exe（30 天限制版，其他版本无法使用第一步中的注册码），执行安装程序完成基本安
过程中需要添入第一步里面算出来的注册码，可以取消时间限制，但是那一组当中只有一个有效，需要实验。

4、 链接硬件调试程序

RealView MDK：找到一个 STM32-SK 的基础程序，最好是只关于 IO 的且与当前板子程序不同，这样在板上就可以看到结果，
击 Project/open project。例如 GPIO、TIMER（另两个例程是关于串口的，需要连接串口才能够看到运行结果）。

使用“Open Project”打开，然后设置 Option 里面的 linker 和 Utilities 里面的项目为“ST LinkII Debuger”。

编译程序，再使用“Start/Stop Debug Session”来写入程序。

IAR EWARM：与以上相同，找到一个符合条件的例程。打开一个 eww 工程文件，右键选取 Option，在 Debuger 里面选择“Th
d-Party Driver”,在“Third-Party Driver”里面添上“$PROJ_DIR$\..\ddl\STM32Driver.dll”。

使用“Make”或“Rebuild All”来编译程序，点“Debug”就烧写进 Flash。使用调试栏里面的“go”等等运行程序。

注：由于目前版本 MDK 与我手头的 ST-LINK-II 编程器不兼容，所以后面的所有工作均改用 IAR。

STM32 学前班教程之四：打好基础建立模板

1、 新建目录 Project_IAR4，按照自己的顺序重新组织 dll（驱动）；inc、src 函数库；settings，其他所有文件全部放个新建的目录下。

2、 双击打开 Project.eww，继续更改内部设置。

3、 需更改的内容列表：

位置和项目 目标 说明

Project\Edir confignations 新建基于 STM3210B 的配置 编译目标和过程文件存放

Project\Option\General Option\Target ST STM32F10x 选择芯片类型

Project\Option\ C/C++ Compiler\Preprocessor\Additional include directories $PROJ_DIR$\

$PROJ_DIR$\inc 头文件相对位置，需要包括“map/lib/type”的位置

Project\Option\ C/C++ Compiler\Preprocessor\Defined symbols 空 空白是在 Flash 里面调试程序,VECT_TAB_RAM 是在AM 里调试程序

Project\Option\ C/C++ Compiler\Optimizations\Size 最终编译一般选择 High

调试可选 None None,Low,Medium,High 是不同的代码优化等级

Project\Option\ Linker\Output 去掉 Overrride default 输出格式使用默认

Project\Option\ Linker\Extra Output 打开 General Extra Output 去掉 Overrride default 厂家要求

Project\Option\ Linker\Config 打开 Overrride default

$PROJ_DIR$\lnkarm_flash.xcl 使用 Flash 调试程序，如果需要使用 RAM 调试则改为 lnkarm_RAM.xcl

Project\Option\ Debugger\Setup\Driver Third-Party Driver 使用第三方驱动连接单片机

Project\Option\ Debugger\ Download Use flash loader 下载到 flash 所需的设置

Project\Option\ Debugger\ Third-Party Driver\ Third-Party Driver\IAR debugger driver $PROJ_DIR$\ddl\STM32Dr er.dll 驱动文件路径

注 1：所有跟路径相关的设置需要根据实际情况编写，相对路径的编写——“$PROJ_DIR$”代表 eww 文件所在文件夹，“. 代表向上一层。

4、 需要重新删除并重新添加 Project 下“FWLib”和“User”的所有文件，为了删减外设模块方便需要在“USER”额外添
“stm32f10x_conf.h”（不添加也可以，需要展开 main.c 找到它）。然后执行 Project\Rebuid All，通过则设置完毕。

5、 完成以上步骤，第一个自己习惯的程序库就建立完毕了，以后可以从“stm32f10x_conf.h”中删减各种库文件，从“s
32f10x_it.c”编辑中断，从“main.c”编写得到自己的程序。最后需要将这个库打包封存，每次解压缩并修改主目录名称
可。

6、 我的程序库特点：

a) 默认兼容 ST-LINK-II，IAR EWARM 4.42A，Flash 调试，其他有可能需要更改设置

b) 为操作方便减少了目录的层次

c) 为学习方便使用网友汉化版 2.0.2 固件，主要是库函数中 c 代码的注释。

后面随着学习深入将在我的模板里面加入如下内容：

d) 加入必用的 flash（读取优化），lib（debug），nvic（中断位置判断、开中断模板），rcc（时钟管理模板，开启外设
钟模板），gpio（管脚定义模板）的初始化代码，所有模板代码用到的时候只要去掉前面的注释“//”，根据需求填入相应
就可以了。

e) 因为自己记性不好，所以 main 函数中的代码做到每行注释，便于自己以后使用。

f) 集成 Print_U 函数简单串口收发函数代码，便于调试，改变使用 Printf 函数的调试习惯。

g) 集成使用 systick 的精确延时函数 delay。

h) 集成时钟故障处理代码。

i) 集成电压监控代码。

j) 集成片上温度检测代码。

k) 逐步加入所有外设的初始化模块

一、编写程序所需的步骤

1、解压缩，改目录名称，和 eww 文件名，以便跟其他程序区分。

2、更改设置：在“stm32f10x_conf.h”关闭不用的外设（在其声明函数前面加注释符号“//”）。并根据外部晶振速度更
其中“HSE_Value”的数值，其单位是 Hz。

3、完成各种头文件的包含（#include "xxx.h";），公共变量的声明（static 数据类型 变量名称;），子程序声明（void
数名称(参数);）……C 语言必须的前置工作。

4、改写我的程序库里面所预设的模板，再进行其他模块的初始化子程序代码的编写，并在程序代码的开始部分调用。注意

a) 开时钟 RCC（在 RCC 初始化中）；

b) 自身初始化；

c) 相关管脚配置（在 GPIO 初始化中）；

d) 是否使用中断（在 NVIC 初始化中）

5、编写 main.c 中的主要代码和各种子函数。

6、在“stm32f10x_it.c”填写各种中断所需的执行代码，如果用不到中断的简单程序则不用编写此文件。

7、编译生成 “bin”的方法：Project\Option\ Linker\Output\Format，里面选择“Other”，在下面的“Output”选 “w-binary”生成 bin。

8、编译生成“hex”的方法：Project\Option\ Linker\Output\Format，里面选择“Other”，在下面的“Output”选“irl-extended”，生成 a79 直接改名成为 hex 或者选中上面的“Output Flie”在“Overrride default”项目里面改扩展名为ex。

使用软件界面的 Debug 烧写并按钮调试程序。注意，ST-Link-II 是直接将程序烧写进 Flash 进行调试，而不是使用 RAM 的式。

STM32 学前班教程之五：给等待入门的人一点点建议

入门必须阅读的相关文档

1、 几个重要官方文档的功能：

a) Datasheet——芯片基本数据，功能参数封装管脚定义和性能规范。

b) 固件函数库用户手册——函数库功能，库函数的定义、功能和用法。

c) 参考手册——各种功能的具体描述，使用方法，原理，相关寄存器。

d) STM32F10xxx 硬件开发：使用入门——相关基础硬件设计

e) STM32F10XXX 的使用限制：芯片内部未解决的硬件设计 bug，开发需要注意绕开。

f) 一本简单的 C 语言书，相信我，不用太复杂。

2、 其他的有用文档，对初学帮助很大

b) 轻松进入 STM32+Cortex-M3 世界.ppt——开发板和最小系统设计需求。

c) 如何选择 STM32 开发板.pdf——各种开发板介绍和功能比较。

d) MXCHIP 的系列视频教程——全部芯片基础及其外设的教程，使用函数库编程的话就不用看每个视频后半段的关于寄存器介绍了。

e) STM32_Technical_Slide(常见问题)——一些优化设计方案。

3、 关于参考书，买了两本但是基本对学习没什么帮助，如果凑齐以上资料，建议慎重买书，不如留着那 n 个几十块钱，攒一起买开发板。

我自己的学习过程

1、 一共 24 个库，不可能都学，都学也没用。按照我的工作需求必须学的有 16 个，这 16 个也不是全学。主要学习来源是种例程代码、“固件函数库用户手册”和“参考手册”。

具体学习方法是通读不同来源的程序，在程序中找到相关的函数库的应用，然后再阅读相关文档，有条件的实验。对于内容选择方面，根据入门内容和未来应用，将所涉及的范围精简到最低，但是对所选择的部分的学习则力求明确。以下是我按照己的需求对程序库函数排列的学习顺序：

a) 绝大部分程序都要涉及到的库——flash，lib，nvic，rcc，只学基础的跟最简单应用相关必用的部分，其他部分后期再回头学。

b) 各种程序通用但不必用的库——exti，MDA，systic，只通读理解其作用。

c) DEMO 板拥有的外设库——gpio，usart，编写代码实验。

d) 未来需要用到的外设的库——tim，tim1，adc，i2c，spi，先理解等待有条件后实验。

e) 开发可靠性相关库——bkp，iwdg，wwdg，pwr，参考其他例程的做法。

f) 其他，根据兴趣来学。

STM32 学前班教程之六：这些代码大家都用得到

2、 阅读 flash： 芯片内部存储器 flash 操作函数

我的理解——对芯片内部 flash 进行操作的函数，包括读取，状态，擦除，写入等等，可以允许程序去操作 flash 上的数

—48MHz 时，取 Latency=1；48~72MHz 时，取 Latency=2。所有程序中必须的

用法：FLASH_SetLatency(FLASH_Latency_2);

位置：RCC 初始化子函数里面，时钟起振之后。

基础应用 2，开启 FLASH 预读缓冲功能，加速 FLASH 的读取。所有程序中必须的

用法：FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

位置：RCC 初始化子函数里面，时钟起振之后。

3、 阅读 lib：调试所有外设初始化的函数。

我的理解——不理解，也不需要理解。只要知道所有外设在调试的时候，EWRAM 需要从这个函数里面获得调试所需信息的地或者指针之类的信息。

基础应用 1，只有一个函数 debug。所有程序中必须的。

用法： #ifdef DEBUG

```
    debug();
#endif
```

位置：main 函数开头，声明变量之后。

4、 阅读 nvic：系统中断管理。

我的理解——管理系统内部的中断，负责打开和关闭中断。

基础应用 1，中断的初始化函数，包括设置中断向量表位置，和开启所需的中断两部分。所有程序中必须的。

用法： void NVIC_Configuration(void)

```
{
NVIC_InitTypeDef NVIC_InitStructure;//中断管理恢复默认参数
#ifdef  VECT_TAB_RAM
//如果 C/C++ Compiler\Preprocessor\Defined symbols 中的定义了 VECT_TAB_RAM（见程序库更改内容的表格）
NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0); //则在 RAM 调试
#else        //如果没有定义 VECT_TAB_RAM
```

#endif　　//结束判断语句

//以下为中断的开启过程，不是所有程序必须的。

//NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

//设置 NVIC 优先级分组，方式。

//注：一共 16 个优先级，分为抢占式和响应式。两种优先级所占的数量由此代码确定，NVIC_PriorityGroup_x 可以是 0、2、3、4，分别代表抢占优先级有 1、2、4、8、16 个和响应优先级有 16、8、4、2、1 个。规定两种优先级的数量后，所有中断级别必须在其中选择，抢占级别高的会打断其他中断优先执行，而响应级别高的会在其他中断执行完优先执行。

//NVIC_InitStructure.NVIC_IRQChannel = 中断通道名;

//开中断，中断名称见函数库

//NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;

//抢占优先级

//NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;

//响应优先级

//NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//启动此通道的中断

//NVIC_Init(&NVIC_InitStructure);　　//中断初始化

}

5、 阅读 rcc：单片机时钟管理。

我的理解——管理外部、内部和外设的时钟，设置、打开和关闭这些时钟。

基础应用 1：时钟的初始化函数过程——

用法：void RCC_Configuration(void)　　//时钟初始化函数

{

　ErrorStatus HSEStartUpStatus;　　//等待时钟的稳定

　RCC_DeInit();　　//时钟管理重置

　RCC_HSEConfig(RCC_HSE_ON);　　//打开外部晶振

```
  if (HSEStartUpStatus == SUCCESS)
  {
FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
//flash 读取缓冲，加速
    FLASH_SetLatency(FLASH_Latency_2); //flash 操作的延时
    RCC_HCLKConfig(RCC_SYSCLK_Div1); //AHB 使用系统时钟
    RCC_PCLK2Config(RCC_HCLK_Div2);  //APB2（高速）为 HCLK 的一半
    RCC_PCLK1Config(RCC_HCLK_Div2);  //APB1（低速）为 HCLK 的一半
//注：AHB 主要负责外部存储器时钟。PB2 负责 AD，I/O，高级 TIM，串口 1。APB1 负责 DA，USB，SPI，I2C，CAN，串口 234
普通 TIM。


    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
//PLLCLK = 8MHz * 9 = 72 MHz
    RCC_PLLCmd(ENABLE);     //启动 PLL
while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){}
//等待 PLL 启动
RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
//将 PLL 设置为系统时钟源
while (RCC_GetSYSCLKSource() != 0x08){}
          //等待系统时钟源的启动
  }
  //RCC_AHBPeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE);
  //启动 AHP 设备
  //RCC_APB2PeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE);
```

```
  //启动 ABP2 设备

  //RCC_APB1PeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE);

  //启动 ABP1 设备

}
```

1、阅读 exti：外部设备中断函数

我的理解——外部设备通过引脚给出的硬件中断，也可以产生软件中断，19 个上升、下降或都触发。EXTI0～EXTI15 连接到脚，EXTI 线 16 连接到 PVD（VDD 监视），EXTI 线 17 连接到 RTC（闹钟），EXTI 线 18 连接到 USB（唤醒）。

基础应用 1，设定外部中断初始化函数。按需求，不是必须代码。

用法： void EXTI_Configuration(void)

```
{

EXTI_InitTypeDef EXTI_InitStructure; //外部设备中断恢复默认参数

EXTI_InitStructure.EXTI_Line = 通道 1|通道 2;

//设定所需产生外部中断的通道，一共 19 个。

EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //产生中断

EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;

//上升下降沿都触发

EXTI_InitStructure.EXTI_LineCmd = ENABLE; //启动中断的接收

EXTI_Init(&EXTI_InitStructure);            //外部设备中断启动

}
```

2、阅读 dma：通过总线而越过 CPU 读取外设数据

我的理解——通过 DMA 应用可以加速单片机外设、存储器之间的数据传输，并在传输期间不影响 CPU 进行其他事情。这对于门开发基本功能来说没有太大必要，这个内容先行跳过。

3、阅读 systic：系统定时器

我的理解——可以输出和利用系统时钟的计数、状态。

基础应用 1，精确计时的延时子函数。推荐使用的代码。

用法：

```
static vu32 TimingDelay;//全局变量声明

void SysTick_Config(void)//systick 初始化函数

{

    SysTick_CounterCmd(SysTick_Counter_Disable);//停止系统定时器

    SysTick_ITConfig(DISABLE);                //停止 systick 中断

    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);

    //systick 使用 HCLK 作为时钟源，频率值除以 8。

    SysTick_SetReload(9000);//重置时间 1 毫秒（以 72MHz 为基础计算）

    SysTick_ITConfig(ENABLE);//开启 systic 中断

}

void Delay (u32 nTime) //延迟一毫秒的函数

{

    SysTick_CounterCmd(SysTick_Counter_Enable);   //systic 开始计时


    TimingDelay = nTime;              //计时长度赋值给递减变量

    while(TimingDelay != 0);          //检测是否计时完成


SysTick_CounterCmd(SysTick_Counter_Disable);  //关闭计数器

    SysTick_CounterCmd(SysTick_Counter_Clear);    //清除计数值

}

void TimingDelay_Decrement(void)

//递减变量函数，函数名由“stm32f10x_it.c”中的中断响应函数定义好了。
```

```
{
    if (TimingDelay != 0x00)                //检测计数变量是否达到 0
      {
     TimingDelay--;                         //计数变量递减
      }
}
```

注：建议熟练后使用，所涉及知识和设备太多，新手出错的可能性比较大。新手可用简化的延时函数代替：

```
void Delay(vu32 nCount)//简单延时函数
{
  for(; nCount != 0; nCount--);（循环变量递减计数）
}
```

当延时较长，又不需要精确计时的时候可以使用嵌套循环：

```
void Delay(vu32 nCount)        //简单的长时间延时函数
{int i;                         //声明内部递减变量
  for(; nCount != 0; nCount--)  //递减变量计数
{for (i=0; i<0xffff; i++)}  //内部循环递减变量计数
}
```

4、阅读 gpio：I/O 设置函数

我的理解——所有输入输出管脚模式设置，可以是上下拉、浮空、开漏、模拟、推挽模式，频率特性为 2M，10M，50M。也以向该管脚直接写入数据和读取数据。

基础应用 1，gpio 初始化函数。所有程序必须。

用法：void GPIO_Configuration(void)

```
{
    GPIO_InitTypeDef GPIO_InitStructure;   //GPIO 状态恢复默认参数
```

//管脚位置定义，标号可以是 NONE、ALL、0 至 15。

GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;//输出速度 2MHz

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //模拟输入模式

GPIO_Init(GPIOC, &GPIO_InitStructure);　　//C 组 GPIO 初始化

//注：以上四行代码为一组，每组 GPIO 属性必须相同，默认的 GPIO 参数为：ALL，2MHz，FLATING。如果其中任意一行与前组相应设置相同，那么那一行可以省略，由此推论如果前面已经将此行参数设定为默认参数（包括使用 GPIO_InitTypeDef IO_InitStructure 代码），本组应用也是默认参数的话，那么也可以省略。以下重复这个过程直到所有应用的管脚全部被义完毕。

……

}

基础应用 2，向管脚写入 0 或 1

用法：GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01);//写入 1

STM32 笔记之七：让它跑起来，基本硬件功能的建立

0、　实验之前的准备

a)　接通串口转接器

b)　下载 IO 与串口的原厂程序，编译通过保证调试所需硬件正常。

1、　flash，lib，nvic，rcc 和 GPIO，基础程序库编写

a)　这几个库函数中有一些函数是关于芯片的初始化的，每个程序中必用。为保障程序品质，初学阶段要求严格遵守方习惯。注意，官方程序库例程中有个 platform_config.h 文件，是专门用来指定同类外设中第几号外设被使用，就是说在ain.c 里面所有外设序号用 x 代替，比如 USARTx，程序会到这个头文件中去查找到底是用那些外设，初学的时候参考例程别这个所迷惑住。

b)　全部必用代码取自库函数所带例程，并增加逐句注释。

c)　习惯顺序——Lib（debug），RCC（包括 Flash 优化），NVIC，GPIO

d)　必用模块初始化函数的定义：

void RCC_Configuration(void);　　//定义时钟初始化函数

void GPIO_Configuration(void);　　//定义管脚初始化函数

void NVIC_Configuration(void);　　//定义中断管理初始化函数

void Delay(vu32 nCount);　　//定义延迟函数

e)　　Main 中的初始化函数调用：

RCC_Configuration();　　//时钟初始化函数调用

NVIC_Configuration();　　//中断初始化函数调用

GPIO_Configuration();　　//管脚初始化函数调用

f)　　Lib 注意事项：

属于 Lib 的 Debug 函数的调用，应该放在 main 函数最开始，不要改变其位置。

g)　　RCC 注意事项：

Flash 优化处理可以不做，但是两句也不难也不用改参数……

根据需要开启设备时钟可以节省电能

时钟频率需要根据实际情况设置参数

h)　　NVIC 注意事项

注意理解占先优先级和响应优先级的分组的概念

i)　　GPIO 注意事项

注意以后的过程中收集不同管脚应用对应的频率和模式的设置。

作为高低电平的 I/O，所需设置：RCC 初始化里面打开 RCC_APB2

PeriphClockCmd(RCC_APB2Periph_GPIOA);GPIO 里面管脚设定：IO 输出（50MHz，Out_PP）；IO 输入（50MHz，IPU）；

j)　　GPIO 应用

GPIO_WriteBit(GPIOB, GPIO_Pin_2, Bit_RESET);//重置

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01);//写入 1

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x00);//写入 0

GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_6) ;//读入 IO

k)　　简单 Delay 函数

void Delay(vu32 nCount)//简单延时函数

{for(; nCount != 0; nCount--);}

实验步骤：

RCC 初始化函数里添加：RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA | RCC APB2Periph G
OB , ENABLE);

不用其他中断，NVIC 初始化函数不用改

GPIO 初始化代码：

//IO 输入，GPIOB 的 2、10、11 脚输出

```
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 ;//管脚号

  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;   //输出速度

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    //输入输出模式

     GPIO_Init(GPIOB, &GPIO_InitStructure);              //初始化
```

简单的延迟函数：

```
void Delay(vu32 nCount)                      //简单延时函数

{ for (; nCount != 0; nCount--);}            //循环计数延时
```

完成之后再在 main.c 的 while 里面写一段：

```
GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01);//写入 1

Delay（0xffff）;

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x00);//写入 0

Delay（0xffff）;
```

就可以看到连接在 PB2 脚上的 LED 闪烁了，单片机就跑起来了。

STM32 笔记之八：来跟 PC 打个招呼，基本串口通讯

a) 目的：在基础实验成功的基础上，对串口的调试方法进行实践。硬件代码顺利完成之后，对日后调试需要用到的 rintf 重定义进行调试，固定在自己的库函数中。

b) 初始化函数定义：

```
void USART_Configuration(void);        //定义串口初始化函数
```

c) 初始化函数调用：

```
void UART_Configuration(void);        //串口初始化函数调用
```

初始化代码：

```
void USART_Configuration(void)                          //串口初始化函数
{
//串口参数初始化
  USART_InitTypeDef USART_InitStructure;              //串口设置恢复默认参数
//初始化参数设置
  USART_InitStructure.USART_BaudRate = 9600;                                        //波特率 9600
   USART_InitStructure.USART_WordLength = USART_WordLength_8b;   //字长 8 位
  USART_InitStructure.USART_StopBits = USART_StopBits_1;                  //1 位停止字节
  USART_InitStructure.USART_Parity = USART_Parity_No;                       //无奇偶校验
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无流控制
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//打开 Rx 接收和 Tx 发送功能
    USART_Init(USART1, &USART_InitStructure);                                        //初始化
  USART_Cmd(USART1, ENABLE);                                                         //启动串口
}
```

RCC 中打开相应串口

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

GPIO 里面设定相应串口管脚模式

```
//串口 1 的管脚初始化

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;                        //管脚 9

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;       //复用推挽输出

  GPIO_Init(GPIOA, &GPIO_InitStructure);                              //TX 初始化

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;                     //管脚 10

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //浮空输入

  GPIO_Init(GPIOA, &GPIO_InitStructure);                             //RX 初始化
```

d)　　简单应用：

发送一位字符

```
USART_SendData(USART1, 数据);                //发送一位数据

while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET)
{}                                                                                    //等待发送完毕
```

接收一位字符

```
while(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET)
{}                                                                                    //等待接收完毕

变量= (USART_ReceiveData(USART1));        //接受一个字节
```

发送一个字符串

先定义字符串：char rx_data[250];

然后在需要发送的地方添加如下代码

```
  int i;                                                                    //定义循环变量

    while(rx_data!='\0')                                          //循环逐字输出，到结束字'\0'

    {USART_SendData(USART1, rx_data);            //发送字符

     while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){} //等待字符发送完毕

     i++;}
```

e) USART 注意事项：

发动和接受都需要配合标志等待。

只能对一个字节操作，对字符串等大量数据操作需要写函数

使用串口所需设置：RCC 初始化里面打开 RCC_APB2PeriphClockCmd

(RCC_APB2Periph_USARTx);GPIO 里面管脚设定：串口 RX（50Hz，IN_FLOATING）；串口 TX（50Hz，AF_PP）；

f) printf 函数重定义（不必理解，调试通过以备后用）

（1） 需要 c 标准函数：

```
#include "stdio.h"
```

（2） 粘贴函数定义代码

```
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)  //定义为 putchar 应用
```

（3） RCC 中打开相应串口

（4） GPIO 里面设定相应串口管脚模式

（6） 增加为 putchar 函数。

```
int putchar(int c)                                          //putchar 函数
{
  if (c == '\n'){putchar('\r');}                            //将 printf 的\n 变成\r
  USART_SendData(USART1, c);                                //发送字符
  while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){} //等待发送结束
  return c;                                                 //返回值
}
```

（8） 通过，试验成功。printf 使用变量输出:%c 字符，%d 整数，%f 浮点数,%s 字符串，/n 或/r 为换行。注意：能用于 main.c 中。

3、 NVIC 串口中断的应用

a) 目的：利用前面调通的硬件基础，和几个函数的代码，进行串口的中断输入练习。因为在实际应用中，不使用中进行的输入是效率非常低的，这种用法很少见，大部分串口的输入都离不开中断。

b) 初始化函数定义及函数调用：不用添加和调用初始化函数，在指定调试地址的时候已经调用过，在那个 NVIC Configuration 里面添加相应开中断代码就行了。

c) 过程：

i. 在串口初始化中 USART_Cmd 之前加入中断设置：

USART_ITConfig(USART1, USART_IT_TXE, ENABLE);//TXE 发送中断，TC 传输完成中断，RXNE 接收中断，PE 奇偶错误中断，以是多个。

ii. RCC、GPIO 里面打开串口相应的基本时钟、管脚设置

iii. NVIC 里面加入串口中断打开代码：

```
NVIC_InitTypeDef NVIC_InitStructure;//中断默认参数

NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQChannel;//通道设置为串口 1 中断

NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;   //中断占先等级 0

NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;              //中断响应优先级 0

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //打开中断

NVIC_Init(&NVIC_InitStructure);                                          //初始化
```

iv. 在 stm32f10x_it.c 文件中找到 void USART1_IRQHandler 函数，在其中添入执行代码。一般最少三个步骤：先用 if 语句判断是发生那个中断，然后清除中断标志位，最后给字符串赋值，或做其他事情。

```
void USART1_IRQHandler(void)                              //串口 1 中断

{

char RX_dat;                                                          //定义字符变量



  if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)  //判断发生接收中断

  {USART_ClearITPendingBit(USART1,  USART_IT_RXNE);          //清除中断标志



   GPIO_WriteBit(GPIOB, GPIO_Pin_10, (BitAction)0x01);              //开始传输

   RX_dat=USART_ReceiveData(USART1) & 0x7F;                         //接收数据，整理除去前两位
```

```
  while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){}//等待发送结束

  }

}
```

d) 中断注意事项：

可以随时在程序中使用 USART_ITConfig(USART1, USART_IT_TXE, DISABLE);来关闭中断响应。

NVIC_InitTypeDef NVIC_InitStructure 定义一定要加在 NVIC 初始化模块的第一句。

全局变量与函数的定义：在任意.c 文件中定义的变量或函数，在其它.c 文件中使用 extern+定义代码再次定义就可以直接用了。

STM32 笔记之九：打断它来为我办事，EXIT (外部 I/O 中断)应用

a) 目的：跟串口输入类似，不使用中断进行的 IO 输入效率也很低，而且可以通过 EXTI 插入按钮事件，本节联系 EXTI 中断。

b) 初始化函数定义：

void EXTI_Configuration(void); //定义 IO 中断初始化函数

c) 初始化函数调用：

EXTI_Configuration();//IO 中断初始化函数调用简单应用：

d) 初始化函数：

```
void EXTI_Configuration(void)

{

  EXTI_InitTypeDef EXTI_InitStructure;        //EXTI 初始化结构定义



EXTI_ClearITPendingBit(EXTI_LINE_KEY_BUTTON);//清除中断标志

   GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource3);//管脚选择

   GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource4);
```

```
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource5);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource6);


   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;//事件选择

  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;//触发模式

  EXTI_InitStructure.EXTI_Line = EXTI_Line3 | EXTI_Line4; //线路选择

  EXTI_InitStructure.EXTI_LineCmd = ENABLE;//启动中断

  EXTI_Init(&EXTI_InitStructure);//初始化

}
```

e) RCC 初始化函数中开启 I/O 时钟

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA , ENABLE);
```

GPIO 初始化函数中定义输入 I/O 管脚。

```
//IO 输入，GPIOA 的 4 脚输入

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;

   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;        //上拉输入

  GPIO_Init(GPIOA, &GPIO_InitStructure);               //初始化
```

f) 在 NVIC 的初始化函数里面增加以下代码打开相关中断：

```
  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQChannel;       //通道

  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//占先级

  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;                        //响应级

  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                        //启动

  NVIC_Init(&NVIC_InitStructure);                                                                    //初始化
```

g) 在 stm32f10x_it.c 文件中找到 void USART1_IRQHandler 函数，在其中添入执行代码。一般最少三个步骤：先使 if 语句判断是发生那个中断，然后清除中断标志位，最后给字符串赋值，或做其他事情。

```
  if(EXTI_GetITStatus(EXTI_Line3) != RESET)                                    //判断中断发生来源

   { EXTI_ClearITPendingBit(EXTI_Line3);                                        //清除中断标志

    USART_SendData(USART1, 0x41);                                               //发送字符“a”

    GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)(1-GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_2)));//LED 发生明交替

}
```

h) 中断注意事项：

中断发生后必须清除中断位，否则会出现死循环不断发生这个中断。然后需要对中断类型进行判断再执行代码。

使用 EXTI 的 I/O 中断，在完成 RCC 与 GPIO 硬件设置之后需要做三件事：初始化 EXTI、NVIC 开中断、编写中断执行代码。

STM32 笔记之十：工作工作，PWM 输出

a) 目的：基础 PWM 输出，以及中断配合应用。输出选用 PB1，配置为 TIM3_CH4，是目标板的 LED6 控制脚。

b) 对于简单的 PWM 输出应用，暂时无需考虑 TIM1 的高级功能之区别。

c) 初始化函数定义：

```
void TIM_Configuration(void);  //定义 TIM 初始化函数
```

d) 初始化函数调用：

```
TIM_Configuration();  //TIM 初始化函数调用
```

e) 初始化函数，不同于前面模块，TIM 的初始化分为两部分——基本初始化和通道初始化：

```
void TIM_Configuration(void)//TIM 初始化函数

{

  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;//定时器初始化结构
```

```
//TIM3 初始化

  TIM_TimeBaseStructure.TIM_Period = 0xFFFF;        //周期 0～FFFF

  TIM_TimeBaseStructure.TIM_Prescaler = 5;          //时钟分频

  TIM_TimeBaseStructure.TIM_ClockDivision = 0;      //时钟分割

  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//模式

  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);   //基本初始化

  TIM_ITConfig(TIM3, TIM_IT_CC4, ENABLE);//打开中断，中断需要这行代码


//TIM3 通道初始化

  TIM_OCStructInit(& TIM_OCInitStructure);                                    //默认参数

  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;                    //工作状态

  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;         //设定为输出,需要 PWM 输出才需要这行
码

  TIM_OCInitStructure.TIM_Pulse = 0x2000;                              //占空长度

  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;             //高电平

  TIM_OC4Init(TIM3, &TIM_OCInitStructure);                             //通道初始化


  TIM_Cmd(TIM3, ENABLE);                                                      //启动 TIM3

}
```

f) RCC 初始化函数中加入 TIM 时钟开启：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM3, ENABLE);
```

g) GPIO 里面将输入和输出管脚模式进行设置。信号：AF_PP，50MHz。

h) 使用中断的话在 NVIC 里添加如下代码：

```
//打开 TIM2 中断
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;  //通道
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;//占先级
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;          //响应级
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //启动
  NVIC_Init(&NVIC_InitStructure);                                          //初始化
```

中断代码：

```
void TIM2_IRQHandler(void)
{
  if (TIM_GetITStatus(TIM2, TIM_IT_CC4) != RESET)       //判断中断来源
  {
    TIM_ClearITPendingBit(TIM2, TIM_IT_CC4);            //清除中断标志
    GPIO_WriteBit(GPIOB, GPIO_Pin_11, (BitAction)(1-GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_11)));//变换 LED
彩
    IC4value = TIM_GetCapture4(TIM2);                   //获取捕捉数值
  }
}
```

i) 简单应用：

```
//改变占空比
TIM_SetCompare4(TIM3, 变量);
```

j)　　注意事项：

管脚的 IO 输出模式是根据应用来定，比如如果用 PWM 输出驱动 LED 则应该将相应管脚设为 AF_PP，否则单片机没有输出。

STM32 笔记之十一：捕捉精彩瞬间，脉冲方波长度捕获

a)　　目的：基础 PWM 输入也叫捕获，以及中断配合应用。使用前一章的输出管脚 PB1（19 脚），直接使用跳线连接输的 PA3（13 脚），配置为 TIM2_CH4，进行实验。

b)　　对于简单的 PWM 输入应用，暂时无需考虑 TIM1 的高级功能之区别，按照目前我的应用目标其实只需要采集高电宽度，而不必知道周期，所以并不采用 PWM 输入模式，而是普通脉宽捕获模式。

c)　　初始化函数定义：

```
void TIM_Configuration(void);  //定义 TIM 初始化函数
```

d)　　初始化函数调用：

```
TIM_Configuration();  //TIM 初始化函数调用
```

e)　　初始化函数，不同于前面模块，TIM 的 CAP 初始化分为三部分——计时器基本初始化、通道初始化和时钟启动初化：

```
void TIM_Configuration(void)//TIM2 的 CAP 初始化函数
{
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;//定时器初始化结构
  TIM_ICInitTypeDef TIM_ICInitStructure;          //通道输入初始化结构

//TIM2 输出初始化
  TIM_TimeBaseStructure.TIM_Period = 0xFFFF;      //周期 0～FFFF
  TIM_TimeBaseStructure.TIM_Prescaler = 5;        //时钟分频
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;    //时钟分割
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//模式
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);//基本初始化
```

```
//TIM2 通道的捕捉初始化

  TIM_ICInitStructure.TIM_Channel = TIM_Channel_4;//通道选择

  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling;//下降沿

  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;//管脚与寄存器对应关系

  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;//分频器

  TIM_ICInitStructure.TIM_ICFilter = 0x4;        //滤波设置，经历几个周期跳变认定波形稳定 0x0～0xF

  TIM_ICInit(TIM2, &TIM_ICInitStructure);        //初始化


  TIM_SelectInputTrigger(TIM2, TIM_TS_TI2FP2);   //选择时钟触发源

  TIM_SelectSlaveMode(TIM2, TIM_SlaveMode_Reset);//触发方式

  TIM_SelectMasterSlaveMode(TIM2, TIM_MasterSlaveMode_Enable); //启动定时器的被动触发

  TIM_ITConfig(TIM2, TIM_IT_CC4, ENABLE);        //打开中断


  TIM_Cmd(TIM2, ENABLE);                         //启动 TIM2

}
```

f) RCC 初始化函数中加入 TIM 时钟开启：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM3, ENABLE);
```

g) GPIO 里面将输入和输出管脚模式进行设置。IN_FLOATING，50MHz。

h) 使用中断的话在 NVIC 里添加如下代码：

```
//打开 TIM 中断（与前一章相同）

  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
```

```
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;

NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
```

i) 简单应用：

```
变量 = TIM_GetCapture4(TIM2);
```

j) 注意事项：

i. 由于我的需求只跟高电平宽度有关，所以避免了使用 PWM 输入模式，这样可以每个管脚捕捉一路信号。如果使用
WM 模式，每一路需要占用两个寄存器，所以一个定时器只能同时使用两路 PWM 输入。

ii. 由于捕捉需要触发启动定时器，所以 PWM 输出与捕捉不容易在同一个 TIM 通道上实现。如果必须的话只能增加
数溢出的相关代码。

iii. 有些程序省略了捕捉通道的初始化代码，这是不对的

iv. 在基本计时器初始化代码里面注意选择适当的计数器长度，最好让波形长度不要长于一个计数周期，否则需要
加溢出代码很麻烦。一个计数周期的长度计算跟如下几个参数有关：

（1） RCC 初始化代码里面的 RCC_PCLKxConfig，这是 TIM 的基础时钟源与系统时钟的关系。

（2） TIM 初始化的 TIM_Period，这是计数周期的值

（3） TIM 初始化的 TIM_Prescaler，这是计数周期的倍频计数器，相当于调节计数周期，可以使 TIM_Period 尽量
提高计数精度。

STM32 笔记之十二：时钟不息工作不止，systic 时钟应用

a) 目的：使用系统时钟来进行两项实验——周期执行代码与精确定时延迟。

b) 初始化函数定义：

```
void SysTick_Configuration(void);
```

c) 初始化函数调用：

SysTick_Configuration();

d) 初始化函数：

```
void SysTick_Configuration(void)

{

  SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);//时钟除 8

  SysTick_SetReload(250000);                                    //计数周期长度

  SysTick_CounterCmd(SysTick_Counter_Enable);          //启动计时器

  SysTick_ITConfig(ENABLE);                                     //打开中断

}
```

e) 在 NVIC 的初始化函数里面增加以下代码打开相关中断：

```
NVIC_SystemHandlerPriorityConfig(SystemHandler_SysTick, 1, 0);//中断等级设置，一般设置的高一些会少受其他影
```

f) 在 stm32f10x_it.c 文件中找到 void SysTickHandler 函数

```
void SysTickHandler(void)

{

执行代码

}
```

g) 简单应用：精确延迟函数，因为 systic 中断往往被用来执行周期循环代码，所以一些例程中使用其中断的启动禁止来编写的精确延时函数实际上不实用，我自己编写了精确计时函数反而代码更精简，思路更简单。思路是调用后，变量零，然后使用时钟来的曾变量，不断比较变量与延迟的数值，相等则退出函数。代码和步骤如下：

i. 定义通用变量：u16 Tic_Val=0; //变量用于精确计时

ii. 在 stm32f10x_it.c 文件中相应定义：

```
extern u16 Tic_Val;//在本文件引用 MAIN.c 定义的精确计时变量
```

iii. 定义函数名称：void Tic_Delay(u16 Tic_Count);//精确延迟函数

iv. 精确延时函数：

```
void Tic_Delay(u16 Tic_Count)              //精确延时函数
```

```
{        Tic_Val=0;                          //变量清零

  while(Tic_Val != Tic_Count){printf("");}//计时

}
```

v. 在 stm32f10x_it.c 文件中 void SysTickHandler 函数里面添加

```
    Tic_Val++;//变量递增
```

vi. 调用代码：Tic_Delay(10); //精确延时

vii. 疑问：如果去掉计时行那个没用的 printf("");函数将停止工作，这个现象很奇怪

STM32 笔记之十三：恶搞，两只看门狗

a) 目的：

了解两种看门狗（我叫它：系统运行故障探测器和独立系统故障探测器，新手往往被这个并不形象的象形名称搞糊涂）之间
区别和基本用法。

b) 相同：

都是用来探测系统故障，通过编写代码定时发送故障清零信号（高手们都管这个代码叫做“喂狗”），告诉它系统运行正
一旦系统故障，程序清零代码（“喂狗”）无法执行，其计数器就会计数不止，直到记到零并发生故障中断（狗饿了开始叫唤
控制 CPU 重启整个系统（不行啦，开始咬人了，快跑……）。

c) 区别：

独立看门狗 Iwdg——我的理解是独立于系统之外，因为有独立时钟，所以不受系统影响的系统故障探测器。（这条狗是借
的，见谁偷懒它都咬！）主要用于监视硬件错误。

窗口看门狗 wwdg——我的理解是系统内部的故障探测器，时钟与系统相同。如果系统时钟不走了，这个狗也就失去作用了
（这条狗是老板娘养的，老板不干活儿他不管！）主要用于监视软件错误。

d) 初始化函数定义：鉴于两只狗作用差不多，使用过程也差不多初始化函数栓一起了，用的时候根据情况删减。

```
void WDG_Configuration(void);
```

e) 初始化函数调用：

```
WDG_Configuration();
```

f) 初始化函数

```
void WDG_Configuration()                //看门狗初始化
```

`

//软件看门狗初始化

```
  WWDG_SetPrescaler(WWDG_Prescaler_8); //时钟 8 分频 4ms
// (PCLK1/4096)/8= 244 Hz (~4 ms)
   WWDG_SetWindowValue(65);                    //计数器数值
  WWDG_Enable(127);                    //启动计数器，设置喂狗时间
// WWDG timeout = ~4 ms * 64 = 262 ms
   WWDG_ClearFlag();                    //清除标志位
  WWDG_EnableIT();                    //启动中断
//独立看门狗初始化
  IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);//启动寄存器读写
  IWDG_SetPrescaler(IWDG_Prescaler_32);//40K 时钟 32 分频
  IWDG_SetReload(349);                    //计数器数值
  IWDG_ReloadCounter();                    //重启计数器
  IWDG_Enable();                    //启动看门狗
}
```

g) RCC 初始化：只有软件看门狗需要时钟初始化，独立看门狗有自己的时钟不需要但是需要 systic 工作相关设置。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);
```

h) 独立看门狗使用 systic 的中断来喂狗，所以添加 systic 的中断打开代码就行了。软件看门狗需要在 NVIC 打开断添加如下代码：

```
  NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQChannel; //通道
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //占先中断等级
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;      //响应中断优先级
  NVIC_Init(&NVIC_InitStructure);                    //打开中断
```

i) 中断程序，软件看门狗在自己的中断中喂狗，独立看门狗需要使用 systic 的定时中断来喂狗。以下两个程序都

```
void WWDG_IRQHandler(void)

{

  WWDG_SetCounter(0x7F);          //更新计数值

WWDG_ClearFlag();               //清除标志位

}

void SysTickHandler(void)

{  IWDG_ReloadCounter();         //重启计数器（喂狗）

}
```

j) 注意事项：

i. 有狗平常没事情可以不理，但是千万别忘了喂它，否则死都不知道怎么死的！

ii. 初始化程序的调用一定要在 systic 的初始化之后。

iii. 独立看门狗需要 systic 中断来喂，但是 systic 做别的用处不能只做这件事，所以我写了如下几句代码，可以影响 systic 的其他应用，其他 systic 周期代码也可参考：

第一步：在 stm32f10x_it.c 中定义变量

```
int Tic_IWDG;            //喂狗循环程序的频率判断变量
```

第二步：将 SysTickHandler 中喂狗代码改为下面：

```
Tic_IWDG++;              //变量递增

if(Tic_IWDG>=100)        //每 100 个 systic 周期喂狗

{  IWDG_ReloadCounter();//重启计数器（喂狗）

  Tic_IWDG=0;           //变量清零

}
```

STM32 笔记之十四：基本问题，来讨论一下软件架构

网上大家都在讨论和学习，但是对于架构这个基本问题却没几个人讨论。个人认为有个好的架构是写好代码的基础，可以使期的调式工作事半功倍！！

a) 顺序执行代码

定义：按照顺序逐行执行的代码

优点：是思路简单，代码可靠不易被干扰。

缺点：占用资源

用途：只用来各种变量、函数的定义，硬件的初始化程序

位置：main.c 的开始一直到 main 函数的 while 函数之前

b) 空闲执行代码

定义：在 CPU 空闲的时候才执行的代码

优点：不占用资源

缺点：执行的实时性不能保证

用途：非实时任务，调试信息输出，不重要的状态指示

位置：main.c 的 main 函数的 while 函数内部

c) 中断执行代码

定义：由软硬件事件打断整个程序而执行的代码

优点：可以保证实时性，各种中断可以安排优先顺序

缺点：关系相对复杂，互相之间容易干扰

用途：触发性的代码，比如接收数据，响应外部设备，按钮的相应

位置：stm32f10x_it.c 内部

d) 循环执行代码

定义：按照时间周期执行的代码

优点：定期自动执行

缺点：实时性不好

用途：需要周期执行的任务，状态检查及相关信息输出，数据记录

e) DMA 自动代码

定义：不需要主程序干预，外设自动执行

优点：自动刷新数据，不占用 CPU 资源

缺点：不能控制

用途：数据通信存储，AD 转换

位置：不需要

2、架构执行顺序图

下载 (51.33 KB)

2009-4-22 15:39

STM32 学习笔记之十五——IAR4 的最后疯狂，笨笨的开发模板下载

准备大换血了，包括开发环境升级和固件升级，那个需要一定过程，吧之前完成的模板跟大家共享一下。

我的程序库特点：

a) 默认兼容 ST-LINK-II，IAR EWARM 4.42A，Flash 调试，使用串口 1，GPIOA 的 3、4、5、6 脚输入，GPIOB 的 1、2、10、11 脚输出，其他有可能需要更改设置

b) 为操作方便减少了目录的层次

c) 为学习方便使用网友汉化版 2.0.2 固件，主要是库函数中 c 代码的注释。

d) 加入必用的 flash（读取优化），nvic（RAM 与 Flash 调试选择），rcc（时钟管理模板，开启外设时钟模板），gpio（管脚定义模板）的初始化代码，所有模板代码用到的时候只要去掉前面的注释"//"，根据需求填入相应值就可以了。

e) 因为自己记性不好，所以 main 函数中的代码做到每行注释，便于自己以后使用。

f) 列出常见应用代码模板与 ASCII 常用列表。

g) [illegible]

h) 集成 NVIC 中断管理模板，EXTI 外部 I/O 中断模板

i) 针对自己情况集成 PWM 输出模板和 CAP 脉宽捕捉模板，并全部注释。

j) 集成系统循环时钟的初始化函数模板

k) 集成自己编写精确延时代码，不会影响 systic 的周期代码的执行。

l) 集成两种看门狗的使用代码，小心使用

M) 集成 hex 生成设置命令，位置在编译目录（STM32F103C8）的 Exe 下，集成 ISP 软件便

于脱离仿真器的串口调试 STMISP.exe。

由于注释写的太多，还加入自己编程以来的许多格式习惯，所以许多人会觉得混乱不堪，在

此声明，此程序库仅仅为个人学习之用！

STM32 学习笔记之十六——题外话，自己做块工程板

一、我的学习计划将 STM32 单片机的硬件设计工作：

第一步——用 STM32F103CBT6 的 48 脚芯片，为光电平台的简单控制为目标，实现基本外围硬件、PWM、串口、I/O。将 SPI、
2C 留插针。

第二步——为集成传感器应用为目标，在第一步硬件基础上制作功能性的套版，两板连接实现 AD、SPI、I2C、RTC 等等功能

二、硬件规划

选用 STM32F103CBT6，面积 7×7mm，128K flash，16K RAM，4 个 16bit PWM，12 个 12bit PWM 或 CAP，2 个 SPI，2 个 I2C，
个串口，1 个 CAN，1 个 USB，），12ADC。

管脚分配目标 1 如图，之后的功能包括：4 个 AD，3 个串口（1 个与 I2C 复用），1 个 SPI，8 个（两组）PWM 输入输出，1 个
SB，1 个 I/O，1 个 MCO。

三、管脚分配：

下载 (32.83 KB)

2009-4-26 16:14

之所以选择这个软件三个理由：1、界面习惯兼容 Protel。2、操作习惯于 Windows 类似方便。3、可输出 igs 用于结构设计

软件使用笔记如下：

a) 流程：新建工程，添加原理图，添加 SCH 库，画原理图，添加 PCB 库，设定封装，添加 PCB，布线，检查，导出
产文件。

b) 新建工程：最好使用自己以前的同版本文档设置，会包含各种库省去大量工作

c) 添加器件到 SCH，可使用复制粘贴的办法，注意管脚，有些需要外壳接地的器件把外壳的焊点画出来。完成后点
放置，改动后再器件名称点击右键更改。

d) 画原理图：操作类似其他 windows 软件，会自动检查错误连接和重复硬件。

e) 添加器件到 PCB 库，最好使用拷贝粘贴的办法，最好有官方的焊点图。没有的话可以按照封装的型号直接去 htt
//www.***search.com/搜索封装型号（不是器件型号），也有封装的相关尺寸和焊点图。

f) 双击原理图的器件，在右下角改封装名称。

g) 添加新的 PCB 到工程：

“设计/规则”改线宽、线距、器件距离……；

“设计/板子形状/重新定义板子形状”改工作区域大小，然后左键点击前置 Keepout 层，画电路板外形；

“设计/板参数选项”改网格大小，器件和走线中鼠标捕捉的间隔大小……；

“设计/Import changes From……”引入原理图的器件和连接方式，包括改动（出现对 match 提示选择继续就可以了）；

“查看/切换单位”改公制和英制；

“工具/取消布线”取消已经布好的线；

“自动布线”计算机自动布线，功能比 Protel 增强不少；

“报告/测量距离”测量实际距离；

在层标签单击，前置这个层。右键有隐藏层和显示层比较常用。

屏幕中点击右键菜单中“设计/规则”、“选项/板参数选项”、 “选项/层叠管理”（添加和删除层）、“选项/显示掩藏
（针对各种类型进行显示和隐藏，查找未布的线就使用此功能后在操作框中点击“所有最终”然后点击“Apply”，再手工
击所有的选项为“隐藏的”再点“Apply”就能看到未布线的连线了）

快捷键：空格键旋转器件，TAB 键切换线宽和放置过孔。左键单击选择，左键按住移动器件（多个重叠会有列表选择，未松
时右键取消操作），左键双击改器件属性（所在层、位置……），右键按住移动鼠标平移视野，滚轮上下移动，滚轮按住移

五、基本电路原理设计

抛弃复杂设计，专注于可独立调式的 CPU 板设计。计划设计模块包括：供电、JTAG、晶振、RTC（电池引出）。

注：未使用标准 JTAG 设计，原因有三：

1、原设计太占管脚，这个尺寸实在难实现

2、这只是 CPU 板具体应用会再做功能套版，上面可以连接标准 JTAG

3、有可能向 USB 烧写和 SW 双线调式方向转变，所以以后不一定会使用标准 JTAG

本文来自 CSDN 博客，转载请标明出处：http://blog.csdn.net/feiniao_lql/archive/2010/06/21/5684074.aspx

STM32 GPIO 应用笔记

1 STM32 的输入输出管脚有下面 8 种可能的配置：（4 输入+2 输出+2 复用输出）

① 浮空输入_IN_FLOATING

② 带上拉输入_IPU

③ 带下拉输入_IPD

④ 模拟输入_AIN
⑤ 开漏输出_OUT_OD

⑥ 推挽输出_OUT_PP

⑦ 复用功能的推挽输出_AF_PP

⑧ 复用功能的开漏输出_AF_OD

1.1 I/O 口的输出模式下，有 3 种输出速度可选(2MHz、10MHz 和 50MHz)，这个速度是指 I/O 口驱动电路的响应速
而不是输出信号的速度,输出信号的速度与程序有关(芯片内部在 I/O 口的输出部分安排了多个响应速度不同的输出驱动电
用户可以根据自己的需要选择合适的驱动电路）。通过选择速度来选择不同的输出驱动模块，达到最佳的噪声控制和降低功
的目的。高频的驱动电路，噪声也高，当不需要高的输出频率时，请选用低频驱动电路，这样非常有利于提高系统的 EMI 性
当然如果要输出较高频率的信号，但却选用了较低频率的驱动模块，很可能会得到失真的输出信号。

关键是 GPIO 的引脚速度跟应用匹配（推荐 10 倍以上？）。比如：

1.1.1 对于串口，假如最大波特率只需 115.2k，那么用 2M 的 GPIO 的引脚速度就够了，既省电也噪声小。

1.1.2　　对于 I2C 接口，假如使用 400k 波特率，若想把余量留大些，那么用 2M 的 GPIO 的引脚速度或许不够，这时可选用 10M 的 GPIO 引脚速度。

1.1.3　　对于 SPI 接口，假如使用 18M 或 9M 波特率，用 10M 的 GPIO 的引脚速度显然不够了，需要选用 50M 的 GPIO 的脚速度。

1.2　　GPIO 口设为输入时，输出驱动电路与端口是断开，所以输出速度配置无意义。

1.3　　在复位期间和刚复位后，复用功能未开启，I/O 端口被配置成浮空输入模式。

1.4　　所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。

1.5　　GPIO 口的配置具有上锁功能，当配置好 GPIO 口后，可以通过程序锁住配置组合，直到下次芯片复位才能解锁

2　　在 STM32 中如何配置片内外设使用的 IO 端口

首先，一个外设经过 ①配置输入的时钟和 ②初始化后即被激活(开启)；③如果使用该外设的输入输出管脚，则需要配置相的 GPIO 端口（否则该外设对应的输入输出管脚可以做普通 GPIO 管脚使用）；④再对外设进行详细配置。

对应到外设的输入输出功能有下述三种情况：
一、外设对应的管脚为输出：需要根据外围电路的配置选择对应的管脚为复用功能的推挽输出或复用功能的开漏输出。
二、外设对应的管脚为输入：则根据外围电路的配置可以选择浮空输入、带上拉输入或带下拉输入。
三、ADC 对应的管脚：配置管脚为模拟输入。

如果把端口配置成复用输出功能，则引脚和输出寄存器断开，并和片上外设的输出信号连接。将管脚配置成复用输出功能/
如果外设没有被激活，那么它的输出将不确定。

3　　通用 IO 端口（GPIO）初始化：

3.1　　GPIO 初始化

3.1.1　　RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | B | C, ENABLE)：使能 APB2 总线外设时钟

3.1.2　　RCC_ APB2PeriphResetCmd (RCC_APB2Periph_GPIOA | B | C, DISABLE)：释放 GPIO 复位

3.2　　配置各个 PIN 端口（模拟输入_AIN、输入浮空_IN_FLOATING、输入上拉_IPU、输入下拉_IPD、开漏输出_O_OD、推挽式输出_OUT_PP、推挽式复用输出_AF_PP、开漏复用输出_AF_OD）

3.3　　GPIO 初始化完成

下面我就在做个抛砖引玉，根据 ST 手册上的内容，简单地综述一下 GPIO 的功能：
一、共有 8 种模式，可以通过编程选择：
1. 浮空输入
2. 带上拉输入
3. 带下拉输入
4. 模拟输入
5. 开漏输出——(此模式可实现 hotpower 说的真双向 IO)
6. 推挽输出
7. 复用功能的推挽输出
8. 复用功能的开漏输出

模式 7 和模式 8 需根据具体的复用功能决定。

二、专门的寄存器(GPIOx_BSRR 和 GPIOx_BRR)实现对 GPIO 口的原子操作，即回避了设置或清除 I/O 端口时的“读-修改-写
操作，使得设置或清除 I/O 端口的操作不会被中断处理打断而造成误动作。

三、每个 GPIO 口都可以作为外部中断的输入，便于系统灵活设计。

四、I/O 口的输出模式下，有 3 种输出速度可选(2MHz、10MHz 和 50MHz)，这有利于噪声控制。

五、所有 I/O 口兼容 CMOS 和 TTL，多数 I/O 口兼容 5V 电平。

六、大电流驱动能力：GPIO 口在高低电平分别为 0.4V 和 VDD-0.4V 时，可以提供或吸收 8mA 电流；如果把输入输出电平分
放宽到 1.3V 和 VDD-1.3V 时，可以提供或吸收 20mA 电流。

七、具有独立的唤醒 I/O 口。

八、很多 I/O 口的复用功能可以重新映射。

九、GPIO 口的配置具有上锁功能，当配置好 GPIO 口后，可以通过程序锁住配置组合，直到下次芯片复位才能解锁。此功能
常有利于在程序跑飞的情况下保护系统中其他的设备,不会因为某些 I/O 口的配置被改变而损坏——如一个输入口变成输出
并输出电流。

STM32 第一个例子

```
//*********************************************************************
// 作者：YYYtech
// 时间：2007/12/14
//*********************************************************************
/*********************************************************************
main 文件，GPIO 操作，完成最简单的 IO 操作实验，就是控制 LED 灯
4 个 LED 分别对应 PC 的 6、7、8、9 引脚。4 个 LED 流水显示
*********************************************************************/
#include "stm32f10x_lib.h"
```

```
void LED_Init(void)
{
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);

}

void LED_TurnOn(u8 led)
{

}

void Delay(vu32 nCount)
{
for(; nCount != 0; nCount--);
}

main()
{

//RCC_Configuration();
LED_Init();

while(1)
{
   GPIO_SetBits(GPIOC, GPIO_Pin_9);
   Delay(0x8ffff);
   GPIO_ResetBits(GPIOC, GPIO_Pin_9);
   Delay(0x8ffff);
   GPIO_SetBits(GPIOC, GPIO_Pin_10);
   Delay(0x8ffff);
   GPIO_ResetBits(GPIOC, GPIO_Pin_10);
   Delay(0x8ffff);
   GPIO_SetBits(GPIOC, GPIO_Pin_11);
   Delay(0x8ffff);
   GPIO_ResetBits(GPIOC, GPIO_Pin_11);
   Delay(0x8ffff);
   GPIO_SetBits(GPIOC, GPIO_Pin_12);
   Delay(0x8ffff);
   GPIO_ResetBits(GPIOC, GPIO_Pin_12);
   Delay(0x8ffff);
}
```

注意：在这里用到了 RCC 和 GPIO 的库函数，所以必须把这两个函数加入工程。

关于固件库函数在文件夹：C:\Keil\ARM\RV31\LIB\ST\STM32F10x
为了不在操作过程中避免改变 KEIL 文件夹下的库函数，可以固件函数库放到其他文件夹下，如：E:\jy\work\STM\WxlStm3
LAB\library
其中 stm32f10x_lib.c 文件是整个库的一些定义，是必须要的。
加入后的工程为：

GPIO 库函数简单说明：
函数名称 功能描述
GPIO_DeInit 重新初始化外围设备 GPIOx 相关寄存器到它的默认复位值
GPIO_AFIODeInit 初始化交错功能(remap, event control 和 EXTI 配置) 寄存器
GPIO_Init 根据 GPIO_初始化结构指定的元素初始化外围设备 GPIOx
GPIO_StructInit 填充 GPIO_初始化结构（GPIO_InitStruct）内的元素为复位值
GPIO_ReadInputDataBit 读指定端口引脚输入数据
GPIO_ReadInputData 读指定端口输入数据
GPIO_ReadOtputDataBit 读指定端口引脚输出数据
GPIO_ReadOtputData 读指定端口输出数据
GPIO_SetBits 置 1 指定的端口引脚
GPIO_ResetBits 清 0 指定的端口引脚
GPIO_WriteBit 设置或清除选择的数据端口引脚
GPIO_Write 写指定数据到 GPIOx 端口寄存器
GPIO_ANAPinConfig 允许或禁止 GPIO 4 模拟输入模式
GPIO_PinLockConfig 锁定 GPIO 引脚寄存器
GPIO_EventOutputConfig 选择 GPIO 引脚作为事件输出
GPIO_EventOutputCmd 允许或禁止事件输出
GPIO_PinRemapConfig 改变指定引脚的影射
GPIO_EMIConfig 允许或禁止 GPIO 8 和 9 的 EMI 模式
拓展实验：
在上面 LED 灯流水显示的基础之上加上按键程序，首先来看看按键的原理图：

当然这个原理图也是相当之简单的，不用读解释了，唯一注意的是 OK 键与其他三个键的区别是按下为高电平，其余三个按
为低电平。
加入后的完整清单如下：

```
//*********************************************************************
// 作者：JingYong
// 时间：2008/4/24
//*********************************************************************
/*********************************************************************
GPIO 操作，完成最简单的 IO 操作实验，用按键控制 LED 灯闪烁
*********************************************************************/
#include "stm32f10x_lib.h"

GPIO_InitTypeDef GPIO_InitStructure;
//键盘定义
#define    KEY_OK     GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)
#define    KEY_DOWN    GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_1)
```

```
#define    KEY_ESC    GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_3)
//LED 初始化
void LED_Init(void)
{
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
}
//按键初始化
void KEY_Init (void)
{
    GPIO_InitTypeDef gpio_init;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    gpio_init.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3;
    gpio_init.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &gpio_init);
}
//延迟函数
void Delay(vu32 nCount)
{
for(; nCount != 0; nCount--);
}
//主函数
main()
{
//RCC_Configuration();
LED_Init();
KEY_Init ();

while(1)
{
   if(!KEY_ESC)
   {
    while(!KEY_ESC) ;
    GPIO_SetBits(GPIOC, GPIO_Pin_9);
    Delay(0x8ffff);
    GPIO_ResetBits(GPIOC, GPIO_Pin_9);
    Delay(0x8ffff);
   }
   else if(!KEY_UP)
   {
    while(!KEY_UP) ;
    GPIO_SetBits(GPIOC, GPIO_Pin_10);
    Delay(0x8ffff);
    GPIO_ResetBits(GPIOC, GPIO_Pin_10);
```

```
   }
   else if(!KEY_DOWN)
   {
    while(!KEY_DOWN) ;
    GPIO_SetBits(GPIOC, GPIO_Pin_11);
    Delay(0x8ffff);
    GPIO_ResetBits(GPIOC, GPIO_Pin_11);
    Delay(0x8ffff);
   }
   else if(KEY_OK)
   {
    while(KEY_OK) ;
    GPIO_SetBits(GPIOC, GPIO_Pin_12);
    Delay(0x8ffff);
    GPIO_ResetBits(GPIOC, GPIO_Pin_12);
    Delay(0x8ffff);
   }
}
}
```

该例子是按下不同的按键，闪烁对应的LED灯。

STM32的GPIO口的输出:开漏输出和推挽输出 收藏

推挽输出与开漏输出的区别:

>>推挽输出:可以输出高,低电平,连接数字器件

>>开漏输出:输出端相当于三极管的集电极. 要得到高电平状态需要上拉电阻才行. 适合于做电流型的驱动,其吸收电流的力相对强(一般20ma以内).

//
/////////////////////////////////// 三极管的开漏输出有什么特性，和推挽是不是一回事，

问题：

很多芯片的供电电压不一样，有3.3v和5.0v，需要把几种IC的不同口连接在一起，是不是直接连接就可以了？实际系统是应用在I2C上面。

简答：

1、部分3.3V器件有5V兼容性，可以利用这种容性直接连接

2、应用电压转换器件，如TPS76733就是5V输入，转换成3.3V、1A输出。

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

开漏电路特点及应用

在电路设计时我们常常遇到开漏（open drain）和开集（open collector）的概念。所谓开漏电路概念中提到的“漏”是指MOSFET的漏极。同理，开集电路中的“集”就是指三极管的集电极。开漏电路就是指以MOSFET的漏极为输出的电路。般的用法是会在漏极外部的电路添加上拉电阻。完整的开漏电路应该由开漏器件和开漏上拉电阻组成。

组成开漏形式的电路有以下几个特点：

1. 利用 外部电路的驱动能力，减少IC内部的驱动。当IC内部MOSFET导通时，驱动电流是从外部的VCC流经R pull-up

2．可以将多个开漏输出的 Pin，连接到一条线上。形成 “与逻辑” 关系。如图 1，当 PIN_A、PIN_B、PIN_C 任意一个变低，
开漏线上的逻辑就为 0 了。这也是 I2C，SMBus 等总线判断总线占用状态的原理。
3．可以利用改变上拉电源的电压，改变传输电平。如图 2，IC 的逻辑电平由电源 Vcc1 决定，而输出高电平则由 Vcc2 决定
这样我们就可以用低电平逻辑控制输出高电平逻辑了。
4．开漏 Pin 不连接外部的上拉电阻，则只能输出低电平(因此对于经典的 51 单片机的 P0 口而言，要想做输入输出功能必须
外部上拉电阻，否则无法输出高电平逻辑)。
5．标准的开漏脚一般只有输出的能力。添加其它的判断电路，才能具备双向输入、输出的能力。

应用中需注意：
1． 开漏和开集的原理类似，在许多应用中我们利用开集电路代替开漏电路。例如，某输入 Pin 要求由开漏电路驱动。则
们常见的驱动方式是利用一个三极管组成开集电路来驱动它，即方便又节省成本。如图 3。
2．上拉电阻 R pull-up 的 阻值 决定了 逻辑电平转换的沿的速度 。阻值越大，速度越低功耗越小。反之亦然。

Push-Pull 输出就是一般所说的推挽输出，在 CMOS 电路里面应该较 CMOS 输出更合适，应为在 CMOS 里面的 push－pull
出能力不可能做得双极那么大。输出能力看 IC 内部输出极 N 管 P 管的面积。和开漏输出相比，push－pull 的高低电平由 I
的电源低定，不能简单的做逻辑操作等。 push－pull 是现在 CMOS 电路里面用得最多的输出级设计方式。
at91rm9200 GPIO 模拟 I2C 接口时注意！！

一.什么是 OC、OD

集电极开路门(集电极开路 OC 或源极开路 OD)
open-drain 是漏极开路输出的意思，相当于集电极开路(open-collector)输出，即 ttl 中的集电极开路（oc）输出。一般
于线或、线与，也有的用于电流驱动。
open-drain 是对 mos 管而言，open-collector 是对双极型管而言，在用法上没啥区别。
开漏形式的电路有以下几个特点：
1.利用外部电路的驱动能力，减少 IC 内部的驱动。 或驱动比芯片电源电压高的负载.
2．可以将多个开漏输出的 Pin，连接到一条线上。通过一只上拉电阻，在不增加任何器件的情况下，形成“与逻辑”关系
这也是 I2C，SMBus 等总线判断总线占用状态的原理。如果作为图腾输出必须接上拉电阻。接容性负载时，下降延是芯片内
晶体管，是有源驱动，速度较快；上升延是无源的外接电阻，速度慢。如果要求速度高电阻选择要小，功耗会大。所以负载
阻的选择要兼顾功耗和速度。
3.可以利用改变上拉电源的电压，改变传输电平。例如加上上拉电阻就可以提供 TTL/CMOS 电平输出等。
4.开漏 Pin 不连接外部的上拉电阻，则只能输出低电平。一般来说，开漏是用来连接不同电平的器件，匹配电平用的。
5.正常的 CMOS 输出级是上、下两个管子，把上面的管子去掉就是 OPEN-DRAIN 了。这种输出的主要目的有两个：电平转换和
与。

6.由于漏级开路，所以后级电路必须接一上拉电阻，上拉电阻的电源电压就可以决定输出电平。这样你就可以进行任意电平
转换了。
7.线与功能主要用于有多个电路对同一信号进行拉低操作的场合，如果本电路不想拉低，就输出高电平，因为 OPEN-DRAIN
面的管子被拿掉，高电平是靠外接的上拉电阻实现的。（而正常的 CMOS 输出级，如果出现一个输出为高另外一个为低时，
于电源短路。）

8.OPEN-DRAIN 提供了灵活的输出方式，但是也有其弱点，就是带来上升沿的延时。因为上升沿是通过外接上拉无源电阻对
载充电，所以当电阻选择小时延时就小，但功耗大；反之延时大功耗小。所以如果对延时有要求，则建议用下降沿输出。

二.什么是线或逻辑与线与逻辑?

在一个结点(线)上，连接一个上拉电阻到电源 VCC 或 VDD 和 n 个 NPN 或 NMOS 晶体管的集电极 C 或漏极 D，这些

因为这些晶体管的基极注入电流(NPN)或栅极加上高电平(NMOS)，晶体管就会饱和，所以这些基极或栅极对这个结点(线)的
系是或非 NOR 逻辑．如果这个结点后面加一个反相器，就是或 OR 逻辑．

注：个人理解：线与，接上拉电阻至电源。(~A)&(~B)=~(A+B)，由公式较容易理解线与此概念的由来 ；

如果用下拉电阻和 PNP 或 PMOS 管就可以构成与非 NAND 逻辑，或用负逻辑关系转换与/或逻辑．

注：线或，接下拉电阻至地。(~A)+(~B)=~(AB)；
这些晶体管常常是一些逻辑电路的集电极开路 OC 或源极开路 OD 输出端．这种逻辑通常称为线与/线或逻辑，当你看到一
芯片的 OC 或 OD 输出端连在一起，而有一个上拉电阻时，这就是线或/线与了，但有时上拉电阻做在芯片的输入端内．
顺便提示如果不是 OC 或 OD 芯片的输出端是不可以连在一起的，总线 BUS 上的双向输出端连在一起是有管理的，同时只
有一个作输出，而其他是高阻态只能输入．

三.什么是推挽结构
一般是指两个三极管分别受两互补信号的控制,总是在一个三极管导通的时候另一个截止.要实现线与需要用 OC(open collt or)门电路 .如果输出级的有两个三极管，始终处于一个导通、一个截止的状态，也就是两个三级管推挽相连，这样的电路
构称为推拉式电路或图腾柱（Totem- pole）输出电路（可惜，图无法贴上）。当输出低电平时，也就是下级负载门输入低
平时，输出端的电流将是下级门灌入 T4；当输出高电平时，也就是下级负载门输入高电平时，输出端的电流将是下级门从
级电源经 T3、D1 拉出。这样一来，输出高低电平时，T3 一路和 T4 一路将交替工作，从而减低了功耗，提高了每个管的
受能力。又由于不论走哪一路，管子导通电阻都很小，使 RC 常数很小，转变速度很快。因此，推拉式输出级既提高电路的
载能力，又提高开关速度。供你参考。
推挽电路是两个参数相同的三极管或 MOSFET,以推挽方式存在于电路中,各负责正负半周的波形放大任务,电路工作时，两只
称的功率开关管每次只有一个导通，所以导通损耗小效率高。

输出既可以向负载灌电流，也可以从负载抽取电流

本文来自 CSDN 博客，转载请标明出处：http://blog.csdn.net/feiniao_lql/archive/2010/06/13/5668585.aspx

STM32 时钟控制 RCC

对于单片机系统来说，CPU 和总线以及外设的时钟设置是非常重要的，因为没有时钟就没有时序，组合电路能干什么想必各
心里都清楚。其实时钟的学习这部分应该提前一些，但由于一开始时间比较短，有些急于求成，所以直接使用了万利给的例
姑且跳过了这一步。介于下面我计划要学习的任务都涉及到兆级的高速传输，例如全速 USB，DMA 等等，所以不能再忽略时
啦，必须要仔细研究一下。

我学习 RCC 的参考资料：
技术文档 0427 及其中文翻译版 STM32F10xxx_Library_Manual_ChineseV2 的第十五章和 RM0008_CH 参考手册。

片上总线标准种类繁多，而由 ARM 公司推出的 AMBA 片上总线受到了广大 IP 开发商和 SoC 系统集成者的青睐，已成为一种流的工业标准片上结构。AMBA 规范主要包括了 AHB(Advanced High performance Bus)系统总线和 APB(Advanced Peripherals)外围总线。二者分别适用于高速与相对低速设备的连接。

由于时钟是一个由内而外的东西，具体设置要从寄存器开始。

RCC 寄存器结构，RCC_TypeDeff，在文件“stm32f10x_map.h”中定义如下：

```
typedef struct
{
vu32 CR;
vu32 CFGR;
vu32 CIR;
vu32 APB2RSTR;
vu32 APB1RSTR;
vu32 AHBENR;
vu32 APB2ENR;
vu32 APB1ENR;
vu32 BDCR;
vu32 CSR;
} RCC_TypeDef;
```

这些寄存器的具体定义和使用方式参见芯片手册，在此不赘述，因为 C 语言的开发可以不和他们直接打交道，当然如果能够以理解和记忆，无疑是百利而无一害。

相信细心的朋友早就发现板子上只有 8Mhz 的晶振，而增强型最高工作频率为 72Mhz，显然需要用 PLL 倍频 9 倍，这些设置需要在初始化阶段完成。为了方便说明，我借用一下例程的 RCC 设置函数，并用中文注释的形式加以说明：

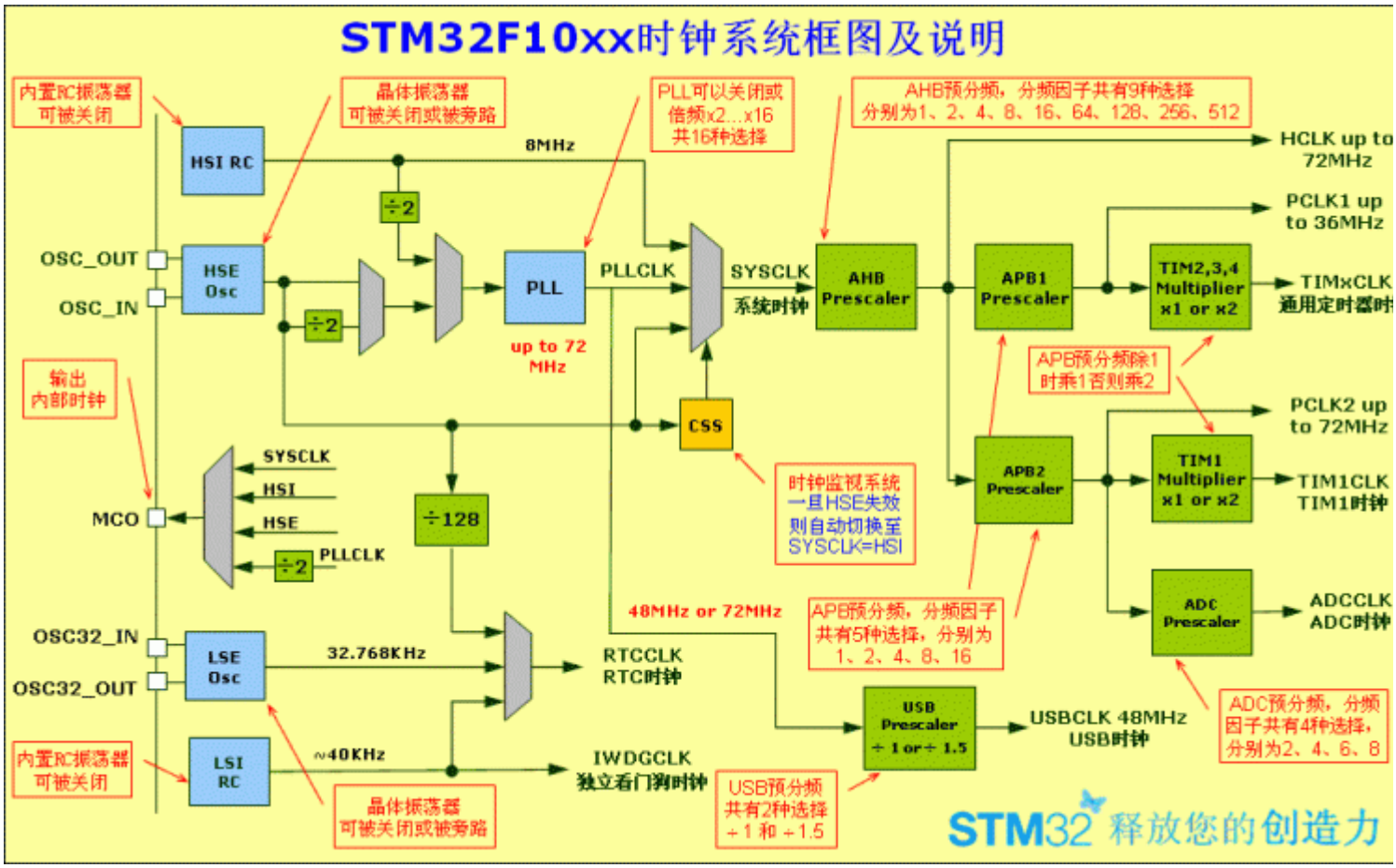
```
/*******************************************************************************
* Function Name : Set_System
* Description    : Configures Main system clocks & power
* Input          : None.
* Return         : None.
*******************************************************************************/
//在此指出上面的注释头应该是复制过来的，写错了...不过没关系，反正没参数需要说明，重要的是函数体。
static void RCC_Config(void)
{

/* 这里是重置了 RCC 的设置，类似寄存器复位 */
RCC_DeInit();

/* 使能外部高速晶振 */
RCC_HSEConfig(RCC_HSE_ON);

/* 等待高速晶振稳定 */
HSEStartUpStatus = RCC_WaitForHSEStartUp();
```

```
{
    /* 使能 flash 预读取缓冲区 */
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

    /* 令 Flash 处于等待状态，2 是针对高频时钟的，这两句跟 RCC 没直接关系，可以暂且略过 */
    FLASH_SetLatency(FLASH_Latency_2);

    /* HCLK = SYSCLK 设置高速总线时钟=系统时钟*/
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK 设置低速总线 2 时钟=高速总线时钟*/
    RCC_PCLK2Config(RCC_HCLK_Div1);

    /* PCLK1 = HCLK/2 设置低速总线 1 的时钟=高速时钟的二分频*/
    RCC_PCLK1Config(RCC_HCLK_Div2);

    /* ADCCLK = PCLK2/6 设置 ADC 外设时钟=低速总线 2 时钟的六分频*/
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    /* Set PLL clock output to 72MHz using HSE (8MHz) as entry clock */
    //上面这句例程中缺失了，但却很关键
    /* 利用锁相环讲外部 8Mhz 晶振 9 倍频到 72Mhz */
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);

    /* Enable PLL 使能锁相环*/
    RCC_PLLCmd(ENABLE);


    /* Wait till PLL is ready 等待锁相环输出稳定*/
    while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
    {}

    /* Select PLL as system clock source 将锁相环输出设置为系统时钟 */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

    /* Wait till PLL is used as system clock source 等待校验成功*/
    while (RCC_GetSYSCLKSource() != 0x08)
    {}
}

    /* Enable FSMC, GPIOD, GPIOE, GPIOF, GPIOG and AFIO clocks */
    //使能外围接口总线时钟，注意各外设的隶属情况，不同芯片的分配不同，到时候查手册就可以
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC, ENABLE);

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE |
                       RCC_APB2Periph_GPIOF | RCC_APB2Periph_GPIOG |
                       RCC_APB2Periph_AFIO, ENABLE);
```

由上述程序可以看出系统时钟的设定是比较复杂的，外设越多，需要考虑的因素就越多。同时这种设定也是有规律可循的，定参数也是有顺序规范的，这是应用中应当注意的，例如 PLL 的设定需要在使能之前，一旦 PLL 使能后参数不可更改。

经过此番设置后，由于我的电路板上是 8Mhz 晶振，所以系统时钟为 72Mhz，高速总线和低速总线 2 都为 72Mhz，低速总线为 36Mhz，ADC 时钟为 12Mhz，USB 时钟经过 1.5 分频设置就可以实现 48Mhz 的数据传输。

一般性的时钟设置需要先考虑系统时钟的来源，是内部 RC 还是外部晶振还是外部的振荡器，是否需要 PLL。然后考虑内部线和外部总线，最后考虑外设的时钟信号。遵从先倍频作为 CPU 时钟，然后在由内向外分频，下级迁就上级的原则有点儿类 PCB 制图的规范化要求，在这里也一样。



(原文件名:STM32 系统时钟框图.gif)
引用图片

STM32 的时钟系统分析

在 STM32 中，有五个时钟源，为 HSI、HSE、LSI、LSE、PLL。

①、HSI 是高速内部时钟，RC 振荡器，频率为 8MHz。

③、LSI 是低速内部时钟，RC 振荡器，频率为 40kHz。

④、LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体。

⑤、PLL 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率大不得超过 72MHz。

其中 40kHz 的 LSI 供独立看门狗 IWDG 使用，另外它还可以被选择为实时时钟 RTC 的时钟源。另外，实时时钟 RTC 的时源还可以选择 LSE，或者是 HSE 的 128 分频。RTC 的时钟源通过 RTCSEL[1:0]来选择。

STM32 中有一个全速功能的 USB 模块，其串行接口引擎需要一个频率为 48MHz 的时钟源。该时钟源只能从 PLL 输出端获取可以选择为 1.5 分频或者 1 分频，也就是，当需要使用 USB 模块时，PLL 必须使能，并且时钟频率配置为 48MHz 或 72MHz。

另外，STM32 还可以选择一个时钟信号输出到 MCO 脚(PA8)上，可以选择为 PLL 输出的 2 分频、HSI、HSE、或者系统时钟

系统时钟 SYSCLK，它是供 STM32 中绝大部分部件工作的时钟源。系统时钟可选择为 PLL 输出、HSI 或者 HSE。系统时钟大频率为 72MHz，它通过 AHB 分频器分频后送给各模块使用，AHB 分频器可选择 1、2、4、8、16、64、128、256、512 分频其中 AHB 分频器输出的时钟送给 5 大模块使用：

①、送给 AHB 总线、内核、内存和 DMA 使用的 HCLK 时钟。

②、通过 8 分频后送给 Cortex 的系统定时器时钟。

③、直接送给 Cortex 的空闲运行时钟 FCLK。

④、送给 APB1 分频器。APB1 分频器可选择 1、2、4、8、16 分频，其输出一路供 APB1 外设使用(PCLK1，最大频率 36MHz另一路送给定时器(Timer)2、3、4 倍频器使用。该倍频器可选择 1 或者 2 倍频，时钟输出供定时器 2、3、4 使用。

⑤、送给 APB2 分频器。APB2 分频器可选择 1、2、4、8、16 分频，其输出一路供 APB2 外设使用(PCLK2，最大频率 72MHz另一路送给定时器(Timer)1 倍频器使用。该倍频器可选择 1 或者 2 倍频，时钟输出供定时器 1 使用。另外，APB2 分频器还一路输出供 ADC 分频器使用，分频后送给 ADC 模块使用。ADC 分频器可选择为 2、4、6、8 分频。

在以上的时钟输出中，有很多是带使能控制的，例如 AHB 总线时钟、内核时钟、各种 APB1 外设、APB2 外设等等。当需使用某模块时，记得一定要先使能对应的时钟。

需要注意的是定时器的倍频器，当 APB 的分频为 1 时，它的倍频值为 1，否则它的倍频值就为 2。

连接在 APB1(低速外设)上的设备有：电源接口、备份接口、CAN、USB、I2C1、I2C2、UART2、UART3、SPI2、窗口看门狗Timer2、Timer3、Timer4。注意 USB 模块虽然需要一个单独的 48MHz 时钟信号，但它应该不是供 USB 模块工作的时钟，而只提供给串行接口引擎(SIE)使用的时钟。USB 模块工作的时钟应该是由 APB1 提供的。

连接在 APB2(高速外设)上的设备有：UART1、SPI1、Timer1、ADC1、ADC2、所有普通 IO 口(PA~PE)、第二功能 IO 口。

```
/*******************************************************************************
* Function Name : RCC_Configuration
```

```
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void RCC_Configuration(void)
{
ErrorStatus HSEStartUpStatus;

/* RCC system reset(for debug purpose) */
// RCC_DeInit();

/* Enable HSE */
RCC_HSEConfig(RCC_HSE_ON);

/* Wait till HSE is ready */
HSEStartUpStatus = RCC_WaitForHSEStartUp();

if(HSEStartUpStatus == SUCCESS)
{
    /* HCLK = SYSCLK */
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK */
    RCC_PCLK2Config(RCC_HCLK_Div1);

    /* PCLK1 = HCLK/2 */
    RCC_PCLK1Config(RCC_HCLK_Div2);

    /* ADCCLK = PCLK2/6 */
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    /* Flash 2 wait state */
    FLASH_SetLatency(FLASH_Latency_2);

    /* Enable Prefetch Buffer */
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

    /* PLLCLK = 8MHz * 9 = 72 MHz */
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);//Pll在最后设置

    /* Enable PLL */
    RCC_PLLCmd(ENABLE);

    /* Wait till PLL is ready */
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
    {
    }
```

```
    /* Select PLL as system clock source */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

    /* Wait till PLL is used as system clock source */
    while(RCC_GetSYSCLKSource() != 0x08)
    {
    }
}

/* Enable GPIOA, GPIOB, GPIOC, GPIOD, GPIOE and AFIO clocks */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |RCC_APB2Periph_GPIOC
         | RCC_APB2Periph_GPIOD | RCC_APB2Periph_GPIOE | RCC_APB2Periph_AFIO, ENABLE);

/* TIM2 clocks enable */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

/* CAN Periph clock enable */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN, ENABLE);
}
```

STM32 的学习心得之 TIMx（通用定时器）

基本的配置定时器的基本设置

1、TIM_TimeBaseStructure.TIM_Prescaler = 0x0;//时钟预分频数 例如：时钟频率=72/(时钟预分频+1)

2、TIM_TimeBaseStructure.TIM_CounterMode = TIM1_CounterMode_Up; //定时器模式 向上计数

3、TIM_TimeBaseStructure.TIM_Period = 0xFFFF; // 自动重装载寄存器周期的值(定时时间) 累计 0xFFFF 个频率后产生更新或者中断(也是说定时时间到)

4、TIM_TimeBaseStructure.TIM_ClockDivision = 0x0; //时间分割值

5、TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);//初始化定时器 2

6、TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE); //打开中断 溢出中断

7、TIM_Cmd(TIM2, ENABLE);//打开定时器

此外要记住一定要打开定时器的时钟(RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);),定时器的频率的可以编的，有对应的模式设置和中断处理。

STM32 学习笔记之 串口通讯

主要功能是把 PC 机发送的数据接收后再返回给 PC 机参数 9600,8,1,N。

```
/**********************************************************************
     Copyright (c) 2008 wormchen
```

```
文 件 名: main.c
说     明: 串口发送接收数据 将 PC 端发来的数据返回给 PC
主要硬件: EMSTM32V1+miniSTMV100(外部 8MRC)
编译环境: MDK3.10
当前版本: 1.0
*********************************************************************/
#include



void RCC_Config(void);
void GPIO_Config(void);
void USART_Config(void);
void Put_String(u8 *p);

int main(void)
{
RCC_Config();
GPIO_Config();
USART_Config();
Put_String("\r\n 请发送数据_\r\n");
while(1)
{
   while(1)
   {
    if(USART_GetFlagStatus(USART2, USART_FLAG_RXNE) == SET)
    {

      USART_SendData(USART2, USART_ReceiveData(USART2));
    }
   }
}
}
/**************************************************
函数: void RCC_Config(void)
功能: 配置系统时钟
参数: 无
返回: 无
**************************************************/
void RCC_Config(void)
{
ErrorStatus HSEStartUpStatus;//定义外部高速晶体启动状态枚举变量
RCC_DeInit();//复位 RCC 外部设备寄存器到默认值
RCC_HSEConfig(RCC_HSE_ON); //打开外部高速晶振
HSEStartUpStatus = RCC_WaitForHSEStartUp();//等待外部高速时钟准备好
if(HSEStartUpStatus == SUCCESS)//外部高速时钟已经准别好
   {
```

```
    RCC_PCLK2Config(RCC_HCLK_Div1); //配置 APB2(PCLK2)钟==AHB 时钟
    RCC_PCLK1Config(RCC_HCLK_Div2);//配置 APB1(PCLK1)钟==AHB1/2 时钟


    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
    //配置 PLL 时钟 == 外部高速晶体时钟*9

      RCC_PLLCmd(ENABLE);//使能 PLL 时钟
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET) //等待 PLL 时钟就绪
       {
       }
      RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);//配置系统时钟 = PLL 时钟

      while(RCC_GetSYSCLKSource() != 0x08) //检查 PLL 时钟是否作为系统时钟
       {
       }
   }
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE);
   //打开 GPIOD 和 AFIO 时钟
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);//使能串口 2 时钟
}
/**************************************************
函数: void GPIO_Config(void)
功能: GPIO 配置
参数: 无
返回: 无
**************************************************/
void GPIO_Config(void)
{
//设置 RTS(PD.04),Tx(PD.05)为推拉输出模式
GPIO_InitTypeDef GPIO_InitStructure; //定义 GPIO 初始化结构体
GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);//使能 GPIO 端口映射 USART2
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;//选择 PIN4 PIN5
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //引脚频率 50M
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;//引脚设置推拉输出
GPIO_Init(GPIOD, &GPIO_InitStructure);//初始化 GPIOD

//配置 CTS (PD.03),USART2 Rx (PD.06)为浮点输入模式
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOD, &GPIO_InitStructure);
}
/**************************************************
函数: void USART_Config(void)
功能: USART 配置
参数: 无
返回: 无
```

```
void USART_Config(void)
{
USART_InitTypeDef USART_InitStructure; //定义串口初始化结构体
USART_InitStructure.USART_BaudRate = 9600;//波特率 9600
USART_InitStructure.USART_WordLength = USART_WordLength_8b;//8 位数据
USART_InitStructure.USART_StopBits = USART_StopBits_1;//1 个停止位
USART_InitStructure.USART_Parity = USART_Parity_No ; //无校验位
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
//禁用 RTSCTS 硬件流控制
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//使能发送接收
USART_InitStructure.USART_Clock = USART_Clock_Disable; //串口时钟禁止
USART_InitStructure.USART_CPOL = USART_CPOL_Low; //时钟下降沿有效
USART_InitStructure.USART_CPHA = USART_CPHA_2Edge;//数据在第二个时钟沿捕捉
USART_InitStructure.USART_LastBit = USART_LastBit_Disable;
//最后数据位的时钟脉冲不输出到 SCLK 引脚
USART_Init(USART2, &USART_InitStructure);//初始化串口 2
USART_Cmd(USART2, ENABLE);//串口 2 使能
}
/***********************************************
函数: void Put_String(void)
功能: 向串口输出字符串
参数: 无
返回: 无
***********************************************/
void Put_String(u8 *p)
{
while(*p)
{
   USART_SendData(USART2, *p++);
   while(USART_GetFlagStatus(USART2, USART_FLAG_TXE) == RESET)
   {

   }
}
}
```

基于 STM32 的 PWM 输出

) 初始化函数定义：

```
void TIM_Configuration(void);   //定义 TIM 初始化函数
```

d) 初始化函数调用：

```
TIM_Configuration();   //TIM 初始化函数调用
```

```
void TIM_Configuration(void)//TIM 初始化函数
{
   TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;//定时器初始化结构
   TIM_OCInitTypeDef   TIM_OCInitStructure;//通道输出初始化结构

//TIM3 初始化
   TIM_TimeBaseStructure.TIM_Period = 0xFFFF;        //周期 0～FFFF
   TIM_TimeBaseStructure.TIM_Prescaler = 5;       //时钟分频
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;    //时钟分割
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//模式
   TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //基本初始化
   TIM_ITConfig(TIM3, TIM_IT_CC4, ENABLE);//打开中断，中断需要这行代码

//TIM3 通道初始化
   TIM_OCStructInit(& TIM_OCInitStructure);                                          //默认参数
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;                    //工作状态
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;       //设定为输出，需要 PWM 输出才需要这行
码
   TIM_OCInitStructure.TIM_Pulse = 0x2000;                   //占空长度
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;                //高电平
   TIM_OC4Init(TIM3, &TIM_OCInitStructure);                   //通道初始化

   TIM_Cmd(TIM3, ENABLE);                                                            //启动 TIM3
}
```

f) RCC 初始化函数中加入 TIM 时钟开启：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM3, ENABLE);
```

g) GPIO 里面将输入和输出管脚模式进行设置。信号：AF_PP，50MHz。

h) 使用中断的话在 NVIC 里添加如下代码：

```
//打开 TIM2 中断
   NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;   //通道
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;//占先级
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;           //响应级
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;       //启动
   NVIC_Init(&NVIC_InitStructure);                                          //初始化
```

中断代码：

```
void TIM2_IRQHandler(void)
{
   if (TIM_GetITStatus(TIM2, TIM_IT_CC4) != RESET)    //判断中断来源
   {
TIM_ClearITPendingBit(TIM2, TIM_IT_CC4);          //清除中断标志
GPIO_WriteBit(GPIOB, GPIO_Pin_11, (BitAction)(1-GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_11)));//变换 LED 色彩
IC4value = TIM_GetCapture4(TIM2);                //获取捕捉数值
   }
}
```

i) 简单应用：
//改变占空比
TIM_SetCompare4(TIM3, 变量);

j) 注意事项：
管脚的 IO 输出模式是根据应用来定，比如如果用 PWM 输出驱动 LED 则应该将相应管脚设为 AF_PP，否则单片机没有输出。

STM32 资料一（转载）

注：下面是一些常用的代码，网上很多但是大多注释不全。高手看没问题，对于我们这些新手就费劲了……所以我把这些代
集中，进行了逐句注释，希望对新手们有价值。

flash： 芯片内部存储器 flash 操作函数

我的理解——对芯片内部 flash 进行操作的函数，包括读取，状态，擦除，写入等等，可以允许程序去操作 flash 上的数

基础应用 1，FLASH 时序延迟几个周期，等待总线同步操作。推荐按照单片机系统运行频率，0—24MHz 时，取 Latency=0；
—48MHz 时，取 Latency=1；48~72MHz 时，取 Latency=2。所有程序中必须的

用法：FLASH_SetLatency(FLASH_Latency_2);

位置：RCC 初始化子函数里面，时钟起振之后。

基础应用 2，开启 FLASH 预读缓冲功能，加速 FLASH 的读取。所有程序中必须的

用法：FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

位置：RCC 初始化子函数里面，时钟起振之后。

3、 lib：调试所有外设初始化的函数。

我的理解——不理解，也不需要理解。只要知道所有外设在调试的时候，EWRAM 需要从这个函数里面获得调试所需信息的地
或者指针之类的信息。

基础应用 1，只有一个函数 debug。所有程序中必须的。

用法：　　#ifdef DEBUG

　　debug();

#endif

　　位置：main 函数开头，声明变量之后。

4、　　nvic：系统中断管理。

我的理解——管理系统内部的中断，负责打开和关闭中断。

基础应用 1，中断的初始化函数，包括设置中断向量表位置，和开启所需的中断两部分。所有程序中必须的。

用法：　　void NVIC_Configuration(void)

{

NVIC_InitTypeDef NVIC_InitStructure;　　//中断管理恢复默认参数

#ifdef　VECT_TAB_RAM　//如果 C/C++ Compiler\Preprocessor\Defined symbols 中的定义了 VECT_TAB_RAM（见程序库改内容的表格）

NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0); //则在 RAM 调试

#else　　//如果没有定义 VECT_TAB_RAM

NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);//则在 Flash 里调试

#endif　　//结束判断语句

//以下为中断的开启过程，不是所有程序必须的。

//NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

//设置 NVIC 优先级分组，方式。

//注：一共 16 个优先级，分为抢占式和响应式。两种优先级所占的数量由此代码确定，NVIC_PriorityGroup_x 可以是 0、2、3、4，分别代表抢占优先级有 1、2、4、8、16 个和响应优先级有 16、8、4、2、1 个。规定两种优先级的数量后，所有中断级别必须在其中选择，抢占级别高的会打断其他中断优先执行，而响应级别高的会在其他中断执行完优先执行。

//NVIC_InitStructure.NVIC_IRQChannel = 中断通道名; //开中断，中断名称见函数库

//NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //抢占优先级

//NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;　　//响应优先级

```
//NVIC_Init(&NVIC_InitStructure);                                    //中断初始化
}
```

5、 rcc：单片机时钟管理。

我的理解——管理外部、内部和外设的时钟，设置、打开和关闭这些时钟。

基础应用 1：时钟的初始化函数过程——

```
用法：void RCC_Configuration(void)                                  //时钟初始化函数
{
   ErrorStatus HSEStartUpStatus;                                    //等待时钟的稳定
   RCC_DeInit();                                                    //时钟管理重置
   RCC_HSEConfig(RCC_HSE_ON);                                       //打开外部晶振
   HSEStartUpStatus = RCC_WaitForHSEStartUp();          //等待外部晶振就绪
if (HSEStartUpStatus == SUCCESS)
   {
FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
//flash 读取缓冲，加速
FLASH_SetLatency(FLASH_Latency_2);                        //flash 操作的延时
RCC_HCLKConfig(RCC_SYSCLK_Div1);                         //AHB 使用系统时钟
RCC_PCLK2Config(RCC_HCLK_Div2);                           //APB2（高速）为 HCLK 的一半
RCC_PCLK1Config(RCC_HCLK_Div2);                           //APB1（低速）为 HCLK 的一半
//注：AHB 主要负责外部存储器时钟。PB2 负责 AD，I/O，高级 TIM，串口 1。APB1 负责 DA，USB，SPI，I2C，CAN，串口 234
普通 TIM。
RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9); //PLLCLK = 8MHz * 9 = 72 MH
RCC_PLLCmd(ENABLE);                                              //启动 PLL
while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){}      //等待 PLL 启动
```

```
RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK); //将 PLL 设置为系统时钟源

while (RCC_GetSYSCLKSource() != 0x08){}     //等待系统时钟源的启动

   }

//RCC_AHBPeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE); //启动 AHP 设备

//RCC_APB2PeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE);//启动 ABP2 设备

   //RCC_APB1PeriphClockCmd(ABP2 设备 1 | ABP2 设备 2 |, ENABLE); //启动 ABP1 设备

}
```

6、 exti：外部设备中断函数

我的理解——外部设备通过引脚给出的硬件中断，也可以产生软件中断，19 个上升、下降或都触发。EXTI0～EXTI15 连接到
脚，EXTI 线 16 连接到 PVD（VDD 监视），EXTI 线 17 连接到 RTC（闹钟），EXTI 线 18 连接到 USB（唤醒）。

基础应用 1，设定外部中断初始化函数。按需求，不是必须代码。

用法：

```
void EXTI_Configuration(void)

{

EXTI_InitTypeDef EXTI_InitStructure;                                //外部设备中断恢复默认参数

EXTI_InitStructure.EXTI_Line = 通道 1|通道 2; //设定所需产生外部中断的通道，一共 19 个。

EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;     //产生中断

EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //上升下降沿都触发

EXTI_InitStructure.EXTI_LineCmd = ENABLE;             //启动中断的接收

EXTI_Init(&EXTI_InitStructure);             //外部设备中断启动

}
```

7、 dma：通过总线而越过 CPU 读取外设数据

我的理解——通过 DMA 应用可以加速单片机外设、存储器之间的数据传输，并在传输期间不影响 CPU 进行其他事情。这对于
门开发基本功能来说没有太大必要，这个内容先行跳过。

8、 systic：系统定时器

我的理解——可以输出和利用系统时钟的计数、状态。

基础应用 1，精确计时的延时子函数。推荐使用的代码。

用法：

```
static vu32 TimingDelay;                                                    //全
变量声明

void SysTick_Config(void)                                                    //syst
k 初始化函数

{

SysTick_CounterCmd(SysTick_Counter_Disable);                    //停止系统定时器

SysTick_ITConfig(DISABLE); //停止 systick 中断

    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //systick 使用 HCLK 作为时钟源，频率值除以 8。

SysTick_SetReload(9000);                                                    //重置时间 1
秒（以 72MHz 为基础计算）

SysTick_ITConfig(ENABLE);                                                   //开启 systic 中


}

void Delay (u32 nTime)                                                      //
迟一毫秒的函数

{

SysTick_CounterCmd(SysTick_Counter_Enable);                     //systic 开始计时

   TimingDelay = nTime;                        //计时长度赋值给递减变量

while(TimingDelay != 0); //检测是否计时完成

SysTick_CounterCmd(SysTick_Counter_Disable);                    //关闭计数器

SysTick_CounterCmd(SysTick_Counter_Clear);                      //清除计数值

}

void TimingDelay_Decrement(void) //递减变量函数，函数名由“stm32f10x_it.c”中的中断响应函数定义好了。

{
```

{ TimingDelay--; //计数变量递减

}

}

注：建议熟练后使用，所涉及知识和设备太多，新手出错的可能性比较大。新手可用简化的延时函数代替：

void Delay(vu32 nCount) //简
延时函数

{

for(; nCount != 0; nCount--); /
循环变量递减计数

}

当延时较长，又不需要精确计时的时候可以使用嵌套循环：

void Delay(vu32 nCount) //简单的长时间延时函数

{int i; //声明内部递减变量

for(; nCount != 0; nCount--) //递减变量计数

{for (i=0; i<0xffff; i++)}
/内部循环递减变量计数

}

9、 gpio：I/O 设置函数

我的理解——所有输入输出管脚模式设置，可以是上下拉、浮空、开漏、模拟、推挽模式，频率特性为 2M，10M，50M。也以向该管脚直接写入数据和读取数据。

基础应用 1，gpio 初始化函数。所有程序必须。

用法：void GPIO_Configuration(void)

{

GPIO_InitTypeDef GPIO_InitStructure; //GPIO 状态恢复默认参数

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_标号 | GPIO_Pin_标号 ; //管脚位置定义，标号可以是 NONE、ALL、0 至 15。

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN; //模拟输入模式

GPIO_Init(GPIOC, &GPIO_InitStructure); //C 组 GPIO 初始化

//注：以上四行代码为一组，每组 GPIO 属性必须相同，默认的 GPIO 参数为：ALL，2MHz，FLATING。如果其中任意一行与前组相应设置相同，那么那一行可以省略，由此推论如果前面已经将此行参数设定为默认参数（包括使用 GPIO_InitTypeDef IO_InitStructure 代码），本组应用也是默认参数的话，那么也可以省略。以下重复这个过程直到所有应用的管脚全部被义完毕。

……

}

基础应用 2，向管脚写入 0 或 1

用法：GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01); //写入 1

基础应用 3，从管脚读入 0 或 1

用法：GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_6)

它跑起来，基本硬件功能的建立

0、 实验之前的准备

a) 接通串口转接器

b) 下载 IO 与串口的原厂程序，编译通过保证调试所需硬件正常。

1、 flash，lib，nvic，rcc 和 GPIO，基础程序库编写

a) 这几个库函数中有一些函数是关于芯片的初始化的，每个程序中必用。为保障程序品质，初学阶段要求严格遵守官方习惯注意，官方程序库例程中有个 platform_config.h 文件，是专门用来指定同类外设中第几号外设被使用，就是说在 main.c 面所有外设序号用 x 代替，比如 USARTx，程序会到这个头文件中去查找到底是用那些外设，初学的时候参考例程别被这个迷惑住。

b) 全部必用代码取自库函数所带例程，并增加逐句注释。

c) 习惯顺序——Lib（debug），RCC（包括 Flash 优化），NVIC，GPIO

d) 必用模块初始化函数的定义：

void RCC_Configuration(void); //定义时钟初始化函数

void NVIC_Configuration(void); //定义中断管理初始化函数

void Delay(vu32 nCount); //定义延迟函数

e) Main 中的初始化函数调用：

RCC_Configuration(); //时钟初始化函数调用

NVIC_Configuration(); //中断初始化函数调用

GPIO_Configuration(); //管脚初始化函数调用

f) Lib 注意事项：

属于 Lib 的 Debug 函数的调用，应该放在 main 函数最开始，不要改变其位置。

g) RCC 注意事项：

Flash 优化处理可以不做，但是两句也不难也不用改参数……

根据需要开启设备时钟可以节省电能

时钟频率需要根据实际情况设置参数

h) NVIC 注意事项

注意理解占先优先级和响应优先级的分组的概念

i) GPIO 注意事项

注意以后的过程中收集不同管脚应用对应的频率和模式的设置。

作为高低电平的 I/O，所需设置：RCC 初始化里面打开 RCC_APB2

PeriphClockCmd(RCC_APB2Periph_GPIOA);GPIO 里面管脚设定：IO 输出（50MHz，Out_PP）；IO 输入（50MHz，IPU）；

j) GPIO 应用

GPIO_WriteBit(GPIOB, GPIO_Pin_2, Bit_RESET);//重置

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01);//写入 1

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x00);//写入 0

GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_6) ;//读入 IO

void Delay(vu32 nCount)//简单延时函数

{for(; nCount != 0; nCount--);}

实验步骤：

RCC 初始化函数里添加：RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA | RCC APB2Periph G
OB , ENABLE);

不用其他中断，NVIC 初始化函数不用改

GPIO 初始化代码：

//IO 输入，GPIOB 的 2、10、11 脚输出

```
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 ;//管脚号

   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //输出速度

   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //输入输出模式

    GPIO_Init(GPIOB, &GPIO_InitStructure);              //初始化
```

简单的延迟函数：

```
void Delay(vu32 nCount)                    //简单延时函数

{ for (; nCount != 0; nCount--);}           //循环计数延时
```

完成之后再在 main.c 的 while 里面写一段：

```
GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x01);//写入 1

Delay（0xffff）;

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)0x00);//写入 0

Delay（0xffff）;
```

就可以看到连接在 PB2 脚上的 LED 闪烁了，单片机就跑起来了。

STM32 笔记之八：来跟 PC 打个招呼，基本串口通讯

a) 目的：在基础实验成功的基础上，对串口的调试方法进行实践。硬件代码顺利完成之后，对日后调试需要用到的 print 重定义进行调试，固定在自己的库函数中。

b) 定义初始化函数

```
void USART_Configuration(void);        //定义串口初始化函数
```

c) 初始化函数调用：

```
void UART_Configuration(void); //串口初始化函数调用
```

初始化代码：

```
void USART_Configuration(void)                        //串口初始化函数
{
//串口参数初始化
   USART_InitTypeDef USART_InitStructure;              //串口设置恢复默认参数
//初始化参数设置
   USART_InitStructure.USART_BaudRate = 9600;                                  //波特率 9600
USART_InitStructure.USART_WordLength = USART_WordLength_8b; //字长 8 位
   USART_InitStructure.USART_StopBits = USART_StopBits_1;                  //1 位停止字节
   USART_InitStructure.USART_Parity = USART_Parity_No;                    //无奇偶校验
   USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无流控制
   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//打开 Rx 接收和 Tx 发送功能
USART_Init(USART1, &USART_InitStructure);                                      //初始化
   USART_Cmd(USART1, ENABLE);                                                    //启动串口
}
```

RCC 中打开相应串口

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

GPIO 里面设定相应串口管脚模式

```
//串口 1 的管脚初始化
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;                        //管脚 9
```

GPIO_Init(GPIOA, &GPIO_InitStructure); //TX 初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10; //管脚 10

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; //浮空输入

GPIO_Init(GPIOA, &GPIO_InitStructure); //RX 初始化

d) 简单应用：

发送一位字符

USART_SendData(USART1, 数据); //发送一位数据

while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){} //等待发送完毕

接收一位字符

while(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET){} //等待接收完毕

变量= (USART_ReceiveData(USART1)); //接受一个字节

发送一个字符串

先定义字符串：char rx_data[250];

然后在需要发送的地方添加如下代码

int i; //定义循环变量

while(rx_data!='\0') //循环逐字输出，到结束字'\0'

{USART_SendData(USART1, rx_data); //发送字符

while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){} //等待字符发送完毕

i++;}

e) USART 注意事项：

发动和接受都需要配合标志等待。

只能对一个字节操作，对字符串等大量数据操作需要写函数

使用串口所需设置：RCC 初始化里面打开 RCC_APB2PeriphClockCmd

f) printf 函数重定义（不必理解，调试通过以备后用）

（1） 需要 c 标准函数：

```
#include "stdio.h"
```

（2） 粘贴函数定义代码

```
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)   //定义为 putchar 应用
```

（3） RCC 中打开相应串口

（4） GPIO 里面设定相应串口管脚模式

（6） 增加为 putchar 函数。

```
int putchar(int c)                                          //putchar 函数
{
   if (c == '\n'){putchar('\r');}                           //将 printf 的\n 变成\r
   USART_SendData(USART1, c);                               //发送字符
   while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){} //等待发送结束
   return c;                                                //返回值
}
```

（8） 通过，试验成功。printf 使用变量输出:%c 字符，%d 整数，%f 浮点数,%s 字符串，/n 或/r 为换行。注意：只能用于 ain.c 中。

3、 NVIC 串口中断的应用

a) 目的：利用前面调通的硬件基础，和几个函数的代码，进行串口的中断输入练习。因为在实际应用中，不使用中断进行输入是效率非常低的，这种用法很少见，大部分串口的输入都离不开中断。

b) 初始化函数定义及函数调用：不用添加和调用初始化函数，在指定调试地址的时候已经调用过，在那个 NVIC Configuration 里面添加相应开中断代码就行了。

c) 过程：

i. 在串口初始化中 USART_Cmd 之前加入中断设置：

```
USART_ITConfig(USART1, USART_IT_TXE, ENABLE);//TXE 发送中断，TC 传输完成中断，RXNE 接收中断，PE 奇偶错误中断，
```

ii. RCC、GPIO 里面打开串口相应的基本时钟、管脚设置

iii. NVIC 里面加入串口中断打开代码：

```
NVIC_InitTypeDef NVIC_InitStructure;//中断默认参数

NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQChannel;//通道设置为串口 1 中断

NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //中断占先等级 0

NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;              //中断响应优先级 0

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;          //打开中断

NVIC_Init(&NVIC_InitStructure);                                          //初始化
```

iv. 在 stm32f10x_it.c 文件中找到 void USART1_IRQHandler 函数，在其中添入执行代码。一般最少三个步骤：先使用 if 句判断是发生那个中断，然后清除中断标志位，最后给字符串赋值，或做其他事情。

```
void USART1_IRQHandler(void)                              //串口 1 中断

{

char RX_dat;                                                    //定义字符变量

   if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)   //判断发生接收中断

   {USART_ClearITPendingBit(USART1,   USART_IT_RXNE);       //清除中断标志

    GPIO_WriteBit(GPIOB, GPIO_Pin_10, (BitAction)0x01);          //开始传输

RX_dat=USART_ReceiveData(USART1) & 0x7F;                        //接收数据，整理除去前两位

USART_SendData(USART1, RX_dat);                                      //发送数据

while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){}//等待发送结束

   }

}
```

d) 中断注意事项：

可以随时在程序中使用 USART_ITConfig(USART1, USART_IT_TXE, DISABLE);来关闭中断响应。

NVIC_InitTypeDef NVIC_InitStructure 定义一定要加在 NVIC 初始化模块的第一句。

全局变量与函数的定义：在任意.c 文件中定义的变量或函数，在其它.c 文件中使用 extern+定义代码再次定义就可以直接用了。

STM32 笔记之九：打断它来为我办事，EXIT (外部 I/O 中断)应用

a) 目的：跟串口输入类似，不使用中断进行的 IO 输入效率也很低，而且可以通过 EXTI 插入按钮事件，本节联系 EXTI 中

b) 初始化函数定义：

```
void EXTI_Configuration(void); //定义 IO 中断初始化函数
```

c) 初始化函数调用：

```
EXTI_Configuration();//IO 中断初始化函数调用简单应用：
```

d) 初始化函数：

```
void EXTI_Configuration(void)

{ EXTI_InitTypeDef EXTI_InitStructure; //EXTI 初始化结构定义

EXTI_ClearITPendingBit(EXTI_LINE_KEY_BUTTON);//清除中断标志

GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource3);//管脚选择

GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource4);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource5);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource6);

EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;//事件选择

   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;//触发模式

   EXTI_InitStructure.EXTI_Line = EXTI_Line3 | EXTI_Line4; //线路选择

   EXTI_InitStructure.EXTI_LineCmd = ENABLE;//启动中断

   EXTI_Init(&EXTI_InitStructure);//初始化

}
```

e) RCC 初始化函数中开启 I/O 时钟

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA , ENABLE);
```

GPIO 初始化函数中定义输入 I/O 管脚。

//IO 输入，GPIOA 的 4 脚输入

```
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;       //上拉输入

   GPIO_Init(GPIOA, &GPIO_InitStructure);              //初始化
```

f) 在 NVIC 的初始化函数里面增加以下代码打开相关中断：

```
   NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQChannel; //通道

   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//占先级

   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //响应级

   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //启动

   NVIC_Init(&NVIC_InitStructure); //初始化
```

g) 在 stm32f10x_it.c 文件中找到 void USART1_IRQHandler 函数，在其中添入执行代码。一般最少三个步骤：先使用 if 语判断是发生那个中断，然后清除中断标志位，最后给字符串赋值，或做其他事情。

```
   if(EXTI_GetITStatus(EXTI_Line3) != RESET)                                    //判断中断发生来源

{ EXTI_ClearITPendingBit(EXTI_Line3);                                            //清除中断标志

USART_SendData(USART1, 0x41);                                                  //发送字符“a”

GPIO_WriteBit(GPIOB, GPIO_Pin_2, (BitAction)(1-GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_2)));//LED 发生明暗交


}
```

h) 中断注意事项：

中断发生后必须清除中断位，否则会出现死循环不断发生这个中断。然后需要对中断类型进行判断再执行代码。

使用 EXTI 的 I/O 中断，在完成 RCC 与 GPIO 硬件设置之后需要做三件事：初始化 EXTI、NVIC 开中断、编写中断执行代码。

STM32 笔记之十：工作工作，PWM 输出

a) 目的：基础 PWM 输出，以及中断配合应用。输出选用 PB1，配置为 TIM3_CH4，是目标板的 LED6 控制脚。

b) 对于简单的 PWM 输出应用，暂时无需考虑 TIM1 的高级功能之区别。

c) 初始化函数定义：

```
void TIM_Configuration(void);   //定义TIM初始化函数
```

d) 初始化函数调用：

```
TIM_Configuration();   //TIM初始化函数调用
```

e) 初始化函数，不同于前面模块，TIM的初始化分为两部分——基本初始化和通道初始化：

```
void TIM_Configuration(void)//TIM初始化函数
{
   TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;//定时器初始化结构
   TIM_OCInitTypeDef   TIM_OCInitStructure;//通道输出初始化结构
//TIM3初始化
   TIM_TimeBaseStructure.TIM_Period = 0xFFFF;        //周期0～FFFF
   TIM_TimeBaseStructure.TIM_Prescaler = 5;       //时钟分频
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;    //时钟分割
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//模式
   TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //基本初始化
   TIM_ITConfig(TIM3, TIM_IT_CC4, ENABLE);//打开中断，中断需要这行代码
   //TIM3通道初始化
   TIM_OCStructInit(& TIM_OCInitStructure);                                    //默认参数
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;                    //工作状态
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;       //设定为输出，需要PWM输出才需要这行码
   TIM_OCInitStructure.TIM_Pulse = 0x2000;                    //占空长度
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;                //高电平
   TIM_OC4Init(TIM3, &TIM_OCInitStructure);                    //通道初始化
```

```
}
```

f) RCC 初始化函数中加入 TIM 时钟开启：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM3, ENABLE);
```

g) GPIO 里面将输入和输出管脚模式进行设置。信号：AF_PP，50MHz。

h) 使用中断的话在 NVIC 里添加如下代码：

```
//打开 TIM2 中断
   NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;   //通道
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;//占先级
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;           //响应级
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;       //启动
   NVIC_Init(&NVIC_InitStructure);                                          //初始化
```

中断代码：

```
void TIM2_IRQHandler(void)
{
   if (TIM_GetITStatus(TIM2, TIM_IT_CC4) != RESET)    //判断中断来源
   {
TIM_ClearITPendingBit(TIM2, TIM_IT_CC4);          //清除中断标志
GPIO_WriteBit(GPIOB, GPIO_Pin_11, (BitAction)(1-GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_11)));//变换 LED 色彩
IC4value = TIM_GetCapture4(TIM2);                 //获取捕捉数值
   }
}
```

i) 简单应用：

```
//改变占空比
```

j) 注意事项：

管脚的 IO 输出模式是根据应用来定，比如如果用 PWM 输出驱动 LED 则应该将相应管脚设为 AF_PP，否则单片机没有输出

我的测试程序可以发出不断循环三种波长并捕获，对比结果如下：

捕捉的稳定性很好，也就是说，同样的方波捕捉到数值相差在一两个数值。

捕捉的精度跟你设置的滤波器长度有关，在这里

TIM_ICInitStructure.TIM_ICFilter = 0x4;　　//滤波设置，经历几个周期跳变认定波形稳定 0x0～0xF

这个越长就会捕捉数值越小，但是偏差几十个数值，下面是 0、4、16 个周期滤波的比较，out 是输出的数值，in 是捕捉到

现在有两个疑问：

1、在 TIM2 的捕捉输入通道初始化里面这句

TIM_SelectInputTrigger(TIM2, TIM_TS_TI2FP2); //选择时钟触发源

按照硬件框图，4 通道应该对应 TI4FP4。可是实际使用 TI1FP1，TI2FP2 都行，其他均编译错误未注册。这是为什么？

2、关闭调试器和 IAR 程序，直接供电跑出来的结果第一个周期很正常，当输出脉宽第二次循环变小后捕捉的数值就差的远
不知道是为什么

时钟不息工作不止，systic 时钟应用

a) 目的：使用系统时钟来进行两项实验——周期执行代码与精确定时延迟。

b) 初始化函数定义：

void SysTick_Configuration(void);

c) 初始化函数调用：

SysTick_Configuration();

d) 初始化函数：

void SysTick_Config(void)

{

SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);//时钟除 8

SysTick_SetReload(250000);　　//计数周期长度

SysTick_CounterCmd(SysTick_Counter_Enable);　　//启动计时器

SysTick_ITConfig(ENABLE);　　//打开中断

}

e) 在 NVIC 的初始化函数里面增加以下代码打开相关中断：

NVIC_SystemHandlerPriorityConfig(SystemHandler_SysTick, 1, 0);//中断等级设置，一般设置的高一些会少受其他影响

f) 在 stm32f10x_it.c 文件中找到 void SysTickHandler 函数

void SysTickHandler(void)

{

执行代码

}

g) 简单应用：精确延迟函数，因为 systic 中断往往被用来执行周期循环代码，所以一些例程中使用其中断的启动和禁止来写的精确延时函数实际上不实用，我自己编写了精确计时函数反而代码更精简，思路更简单。思路是调用后，变量清零，然使用时钟来的曾变量，不断比较变量与延迟的数值，相等则退出函数。代码和步骤如下：

i. 定义通用变量：u16 Tic_Val=0; //变量用于精确计时

ii. 在 stm32f10x_it.c 文件中相应定义：

extern u16 Tic_Val;//在本文件引用 MAIN.c 定义的精确计时变量

iii. 定义函数名称：void Tic_Delay(u16 Tic_Count);//精确延迟函数

iv. 精确延时函数：

void Tic_Delay(u16 Tic_Count)　　//精确延时函数

{ Tic_Val=0;　　//变量清零

while(Tic_Val != Tic_Count){printf("");}//计时

v. 在 stm32f10x_it.c 文件中 void SysTickHandler 函数里面添加

```
Tic_Val++;//变量递增
```

vi. 调用代码：Tic_Delay(10); //精确延时

vii. 疑问：如果去掉计时行那个没用的 printf("");函数将停止工作，这个现象很奇怪

C 语言功底问题。是的，那个“注意事项”最后的疑问的原因就是这个

Tic_Val 应该改为 vu16

```
while(Tic_Val != Tic_Count){printf("");}//计时
```

就可以改为：

```
while(Tic_Val != Tic_Count);                //检查变量是否计数到位
```

STM32 笔记之十三：恶搞，两只看门狗

a) 目的：

了解两种看门狗（我叫它：系统运行故障探测器和独立系统故障探测器，新手往往被这个并不形象的象形名称搞糊涂）之间
区别和基本用法。

b) 相同：

都是用来探测系统故障，通过编写代码定时发送故障清零信号（高手们都管这个代码叫做“喂狗”），告诉它系统运行正
一旦系统故障，程序清零代码（“喂狗”）无法执行，其计数器就会计数不止，直到记到零并发生故障中断（狗饿了开始叫唤
控制 CPU 重启整个系统（不行啦，开始咬人了，快跑……）。

c) 区别：

独立看门狗 Iwdg——我的理解是独立于系统之外，因为有独立时钟，所以不受系统影响的系统故障探测器。（这条狗是借
的，见谁偷懒它都咬！）主要用于监视硬件错误。

窗口看门狗 wwdg——我的理解是系统内部的故障探测器，时钟与系统相同。如果系统时钟不走了，这个狗也就失去作用了
（这条狗是老板娘养的，老板不干活儿他不管！）主要用于监视软件错误。

d) 初始化函数定义：鉴于两只狗作用差不多，使用过程也差不多初始化函数栓一起了，用的时候根据情况删减。

```
void WDG_Configuration(void);
```

e) 初始化函数调用：

```
WDG_Configuration();
```

f) 初始化函数

```
void WDG_Configuration()              //看门狗初始化

{

//软件看门狗初始化

   WWDG_SetPrescaler(WWDG_Prescaler_8); //时钟 8 分频 4ms

// (PCLK1/4096)/8= 244 Hz (~4 ms)

WWDG_SetWindowValue(65);              //计数器数值

   WWDG_Enable(127);                 //启动计数器，设置喂狗时间

// WWDG timeout = ~4 ms * 64 = 262 ms

WWDG_ClearFlag();                 //清除标志位

   WWDG_EnableIT();                     //启动中断

//独立看门狗初始化

   IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);//启动寄存器读写

   IWDG_SetPrescaler(IWDG_Prescaler_32);//40K 时钟 32 分频

   IWDG_SetReload(349);                  //计数器数值

   IWDG_ReloadCounter();              //重启计数器

   IWDG_Enable();                        //启动看门狗

}
```

g) RCC 初始化：只有软件看门狗需要时钟初始化，独立看门狗有自己的时钟不需要但是需要 systic 工作相关设置。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);
```

h) 独立看门狗使用 systic 的中断来喂狗，所以添加 systic 的中断打开代码就行了。软件看门狗需要在 NVIC 打开中断添加下代码：

```
   NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQChannel; //通道

   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //占先中断等级
```

```
   NVIC_Init(&NVIC_InitStructure);                        //打开中断
```

i) 中断程序，软件看门狗在自己的中断中喂狗，独立看门狗需要使用 systic 的定时中断来喂狗。以下两个程序都在 stm32f10x_it.c 文件中。

```
void WWDG_IRQHandler(void)

{

   WWDG_SetCounter(0x7F);        //更新计数值

WWDG_ClearFlag();              //清除标志位

}

void SysTickHandler(void)

{   IWDG_ReloadCounter();        //重启计数器（喂狗）

}
```

j) 注意事项：

i. 有狗平常没事情可以不理，但是千万别忘了喂它，否则死都不知道怎么死的！

ii. 初始化程序的调用一定要在 systic 的初始化之后。

iii. 独立看门狗需要 systic 中断来喂，但是 systic 做别的用处不能只做这件事，所以我写了如下几句代码，可以不影响 stic 的其他应用，其他 systic 周期代码也可参考：

第一步：在 stm32f10x_it.c 中定义变量

```
int Tic_IWDG;            //喂狗循环程序的频率判断变量
```

第二步：将 SysTickHandler 中喂狗代码改为下面：

```
Tic_IWDG++;           //变量递增

if(Tic_IWDG>=100)     //每 100 个 systic 周期喂狗

{   IWDG_ReloadCounter();//重启计数器（喂狗）

   Tic_IWDG=0;        //变量清零

}
```

一切来自网络

继续更新.....

本贴被 tangwei039 编辑过,最后修改时间：2010-08-08,20:06:30.

_____ http://www.ourdev.cn/guide_index.html