一、教程 1

STM32 学习教程

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。主要学习来源是各种例程代码、“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。对于内容的选择方面，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。以下是我按照自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，其他部分后期再返回头学。

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

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……所以我把这些代码集中，进行了逐句注释，希望对新手们有价值。

1、 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 上的数据。

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；24—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址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 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
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);//则在 Flash 里调试
#endif
```
//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、1、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，所有的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();                                      //将时钟重置为缺省值
  RCC_HSEConfig(RCC_HSE_ON);                         //设置外部晶振
  HSEStartUpStatus = RCC_WaitForHSEStartUp();  //等待外部晶振就绪
if (HSEStartUpStatus == SUCCESS)
   {
FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable); //使能预取缓存
    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。APB2 负责 AD，I/O，高级 TIM，串口 1。APB1 负责 DA，USB，SPI，I2C，CAN，串口 2345，普通 TIM。
```

```
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);  //PLLCLK = 8MHz * 9 = 72 MH
                                                          //设置 PLL 时钟源及倍频系数
    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)
     {}                                                   //将 PLL 返回,并作为系统时钟源
     }
//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、**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 连接到管脚，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;      //设置中断模式
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
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、**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。这对于入门开发基本功能来说没有太大必要，这个内容先行跳过。

8、**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_ Configuration (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

}
```

9、 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。也可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_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一组相应设置相同，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 GPIO_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定义完毕。

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，**从管脚读取 0 或 1**

用法：GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_6)

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官方习惯。注意，官方程序库例程中有个 platform_conf.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 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被这个所迷惑住。

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_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,Enabule);启动时钟 GPIOA

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_GPIOB , 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 printf 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
{
  USART_InitTypeDef USART_InitStructure;
  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){}
    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');}              //若为\n，则发送\r
   USART_SendData(USART1, c);                          //发送字符
   while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){}
   return c;                                           //返回值
 }
```

（7）　　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 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，不使用中断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，可以是多个。

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;
  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 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)  //判断是否接收到中断
  {
     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; //接收数据并除去第一位
     USART_SendData(USART1, RX_dat);                    //发送数据
     while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){}
   }
}
```

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调用了。

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：（**嵌套中断向量控制器 NVIC**）

```
  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
发生明暗交替
 }
```

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
   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 输出才需要这行代码
    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;先占优先级
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;     //从优先级
  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**

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 是捕捉到的。

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了。不知道是为什么

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，变量清零，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**

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）之间的区别和基本用法。

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”），告诉它系统运行正常。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。（这条狗是借来的，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 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
}
```