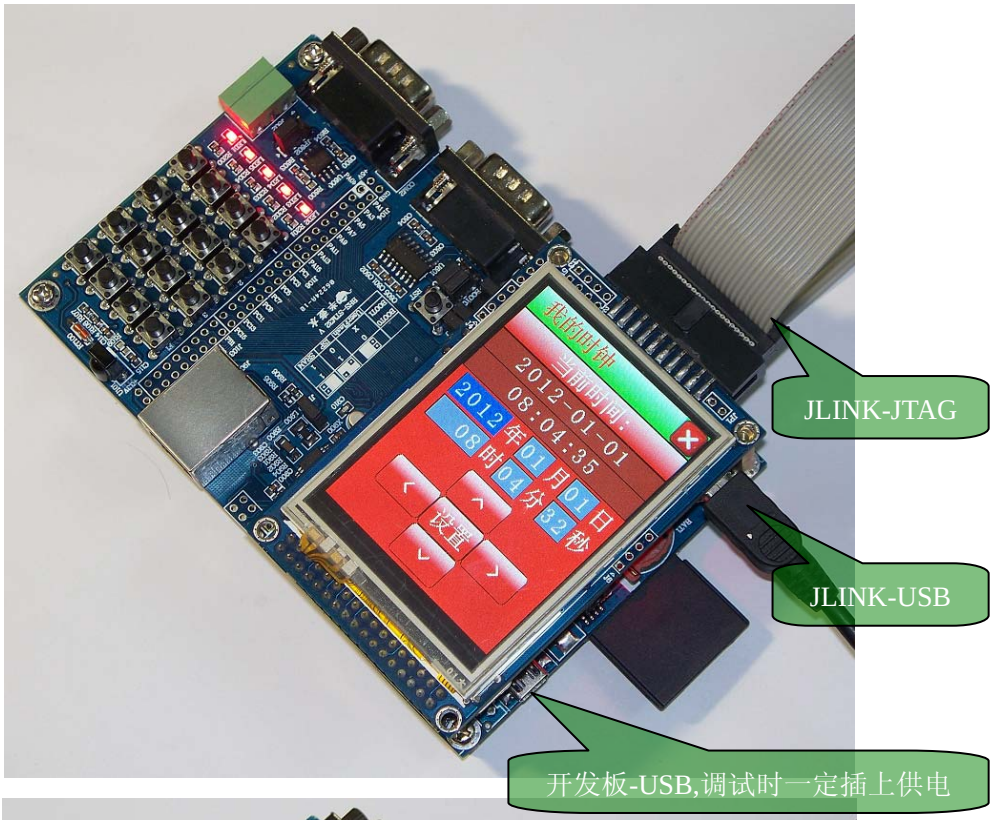
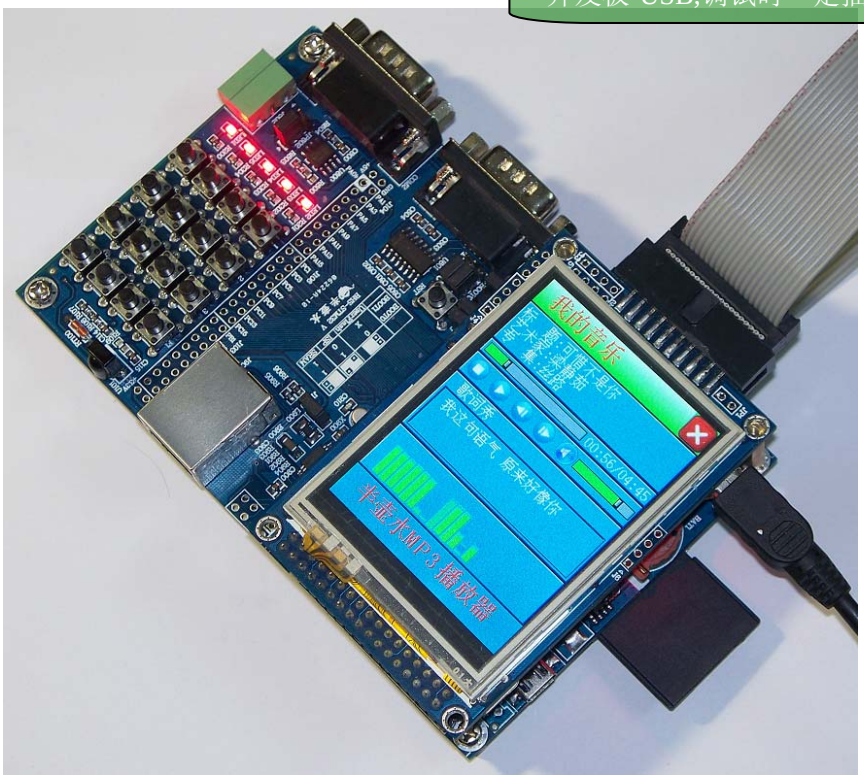
STM32 菜鸟学习手册-罗嗦版

lyw藏书





半壶水出品

序:

www藏书

阅读本文档请使用书签方便快速查找，PDF 阅读器推荐【Foxit Reader】，速度更快，书签显示更清楚

STM32 是 Cortex-M3 内核芯片，Cortex-M3 内核芯片还有很多，不管哪种核心都是一样的。所以《Cortex-M3 内核技术参考手册》是必须了解的，《Cortex-M3 内核技术参考手册》介绍所有不同厂家 Cortex-M3 芯片共有的东西。

《STM32F10x 微控制器参考手册》详细介绍了 STM32 系列 CPU 结构，组成，外设资源，做底层开发必须熟悉，即使不熟悉也应该知道当你使用某个外设，比如串口，ADC，在相应章节找到答案。如果你做的工作更倾向于应用开发那么《STM32F101xx 与 STM32F103xx 固件函数库用户手册》更适合你。建议使用 ST 官方提供的库函数，不管你是开发者还是老板，让产品快速上市才能在市场上赢的先机。ST 官方库将大大缩短你的开发时间。有人说使用固件库效率低，这个也没什么关系，只要你工作效率高，累的是 CPU，你管他干啥。☺

开发软件建议使用 MDK,也就是 KEIL。KEIL 简单易用，毕竟 KEIL 现在是 ARM 公司旗下产品，从 KEIL 软件更新速度就知道 ARM 公司对 KEIL 的重视程度。毕竟使用 KEIL 的人多最终受益的是 ARM 公司。在光盘里和 KEIL 安装目录有 MDK 手册《UV3.chm》，和 RTX,文件系统手册《rlarm.chm》

目前开发软件使用 MDK4.12,以前的版本使用 MDK3.5，实际上这 2 个版本没多大区别。提供的基于直接操作寄存器的例程和使用库函数的例程。两者功能一致，对照讲解，让你更容易理解。

MDK 软件仿真功能比较强大，这也是我选择 MDK 的原因之一

本文是入门教程，高手请飘过…

QQ: 958664258
21IC 用户名：banhushui
交流平台：http://blog.21ic.com/user1/5817/index.html
Email: banhushui@163.com
淘宝店铺：http://shop58559908.taobao.com

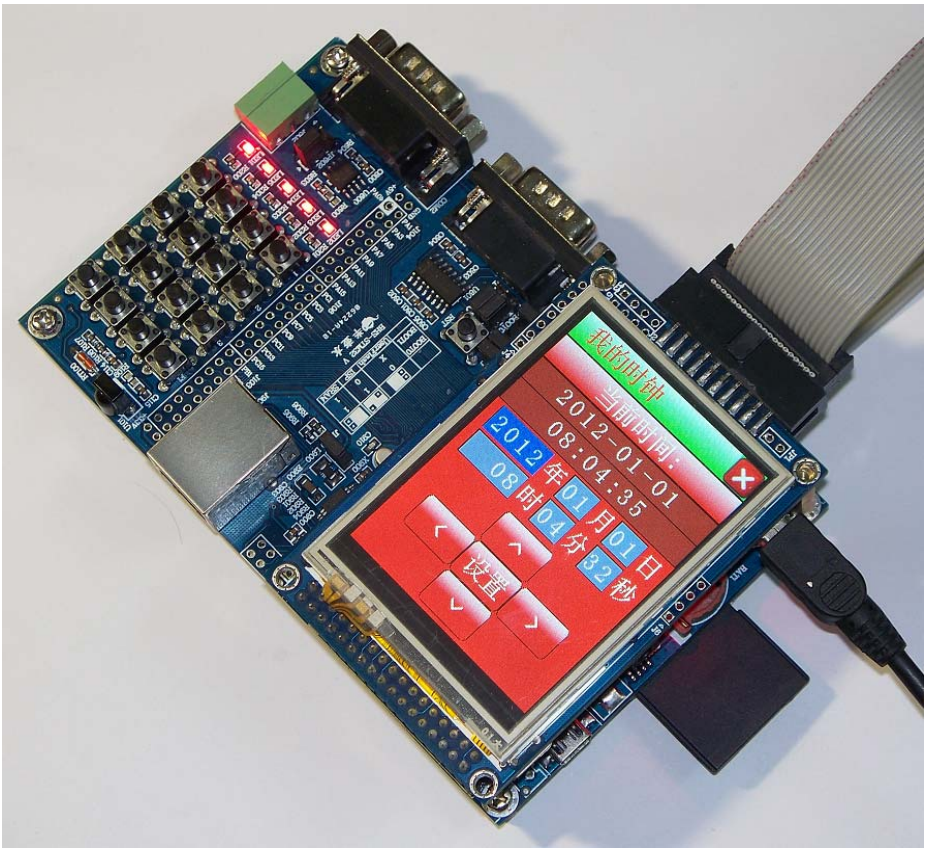
目录

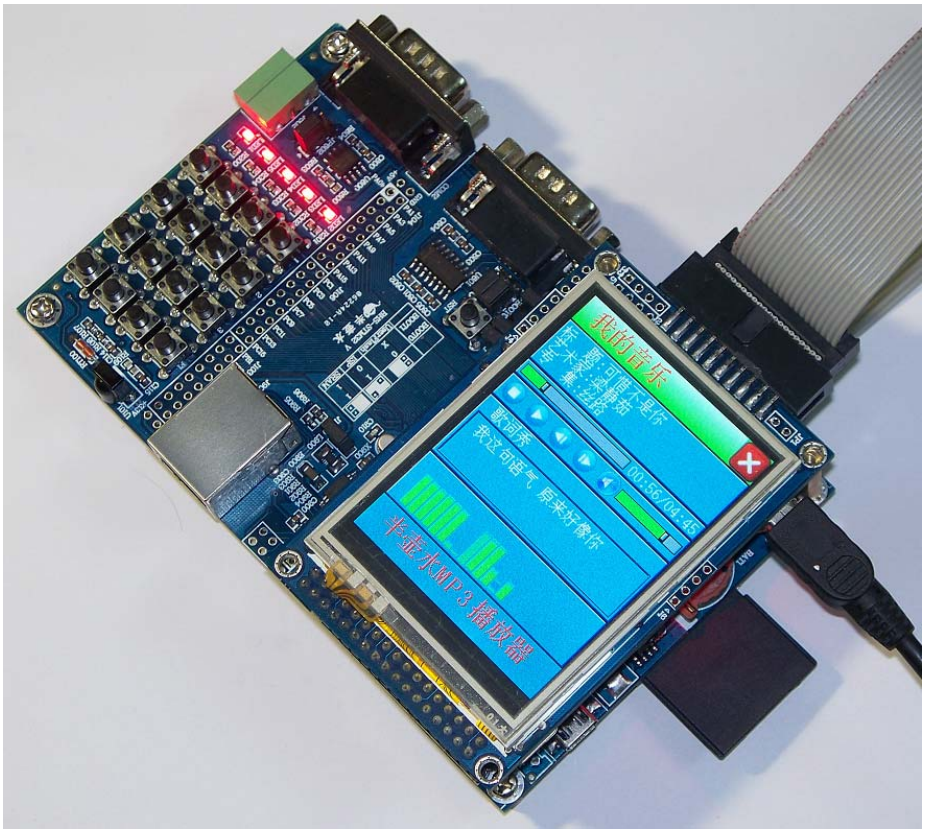
lyw藏书

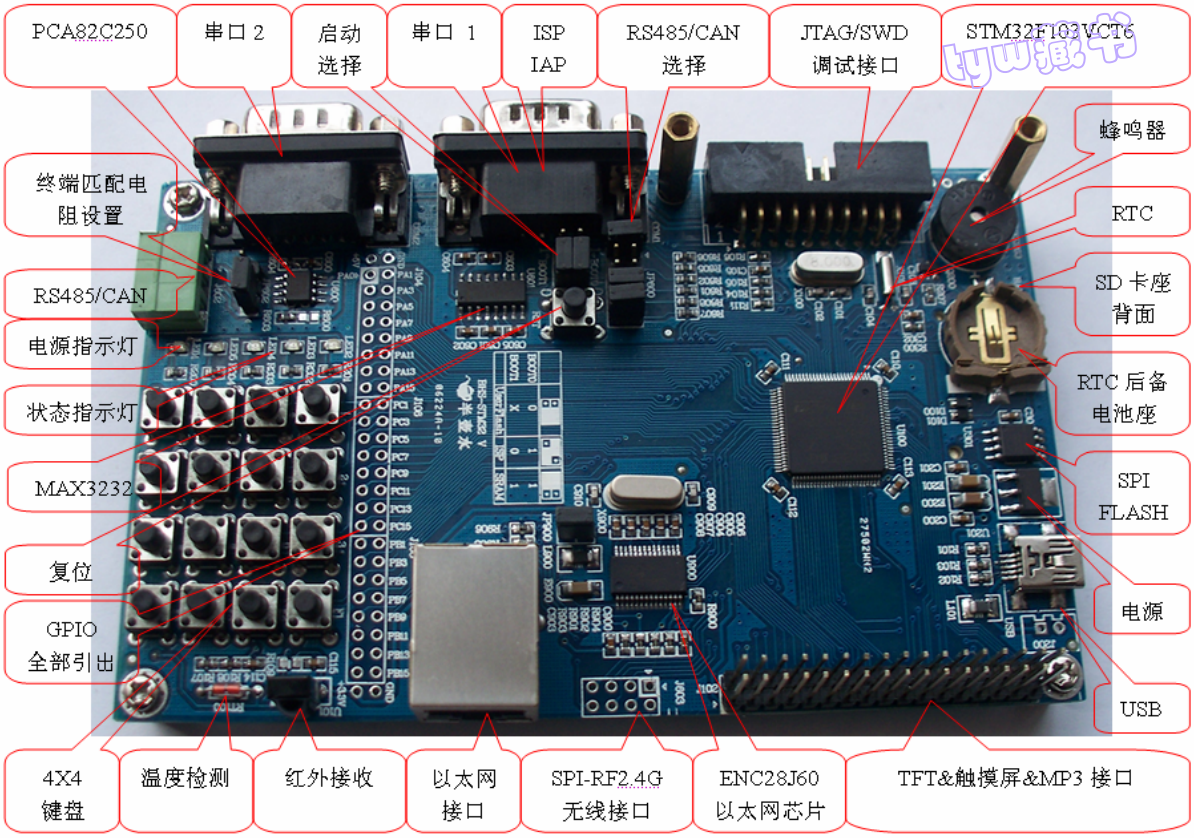
一 开发硬件选择

tyw藏书

1.1 BHS-STM32-V(+FSMC 总线 2.8TFT+MP3+以太网+CAN+RS485+JLINK V7)







BHS-STM32-V 硬件资源：

.CPU: STM32F103VC, TQFP100 脚；FLASH 容量：256KB,　SRAM 容量：48KB

.1 个 JTAG 调试接口

.1 个电源 LED，4 个状态 LED

.2 个 RS232,支持 3 线 ISP

.1 个 RS485

.1 个 CAN

.1 个 USB

.1 个 SD 卡插座

.1 个 TFT(带触摸屏)接口

.1 个 25F080(1M 字节)的串行 FLASH

.1 个 RTC 后备电池座

.1 个 28J60 网络接口

.1 个 SPI 2.4G RF 接口(NRF24L01 官方标准接口)

.1 个蜂鸣器接口

.1 个 NTC 温度传感器

.1 个红外接收

.1 个 4*4 键盘

.1 个 VS1003B MP3 硬件解码芯片(在 TFT 模块背面)

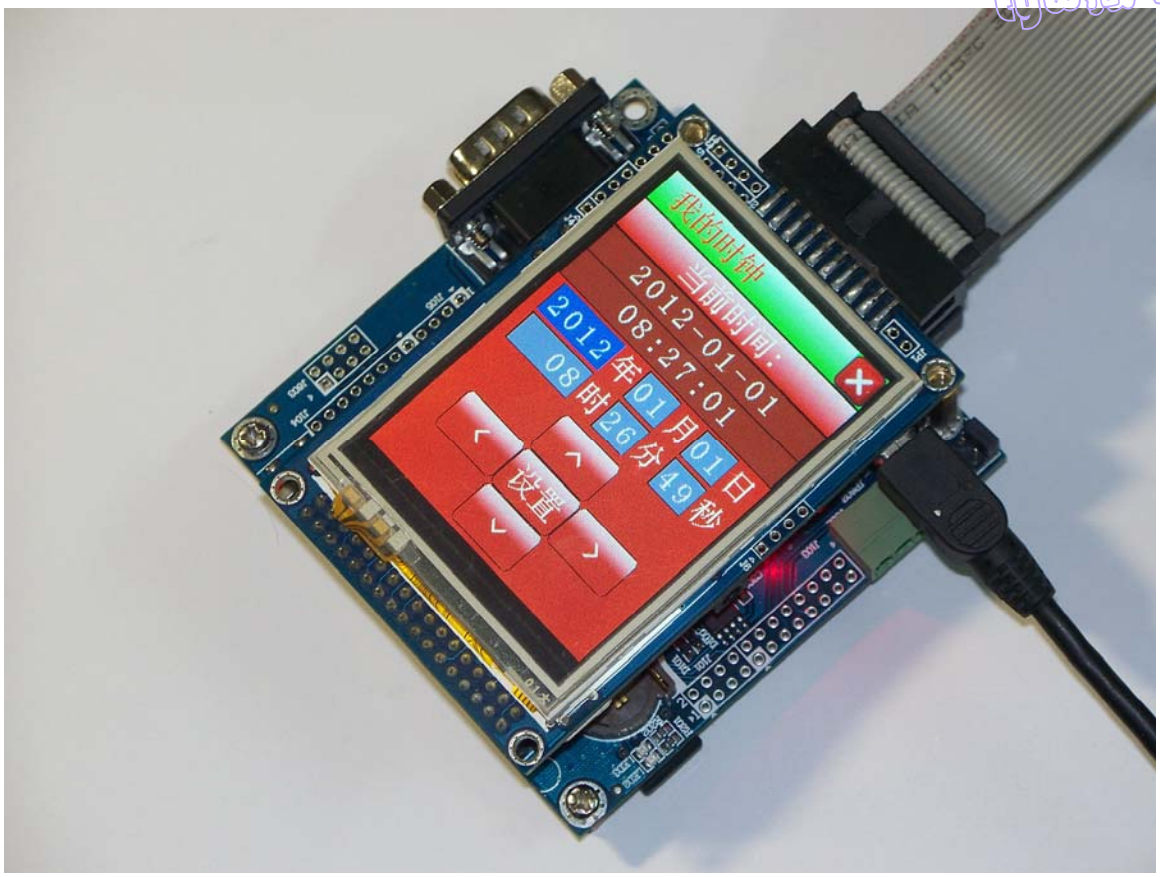
说明：本开发板是带 JLINK 硬件仿真器的哦。

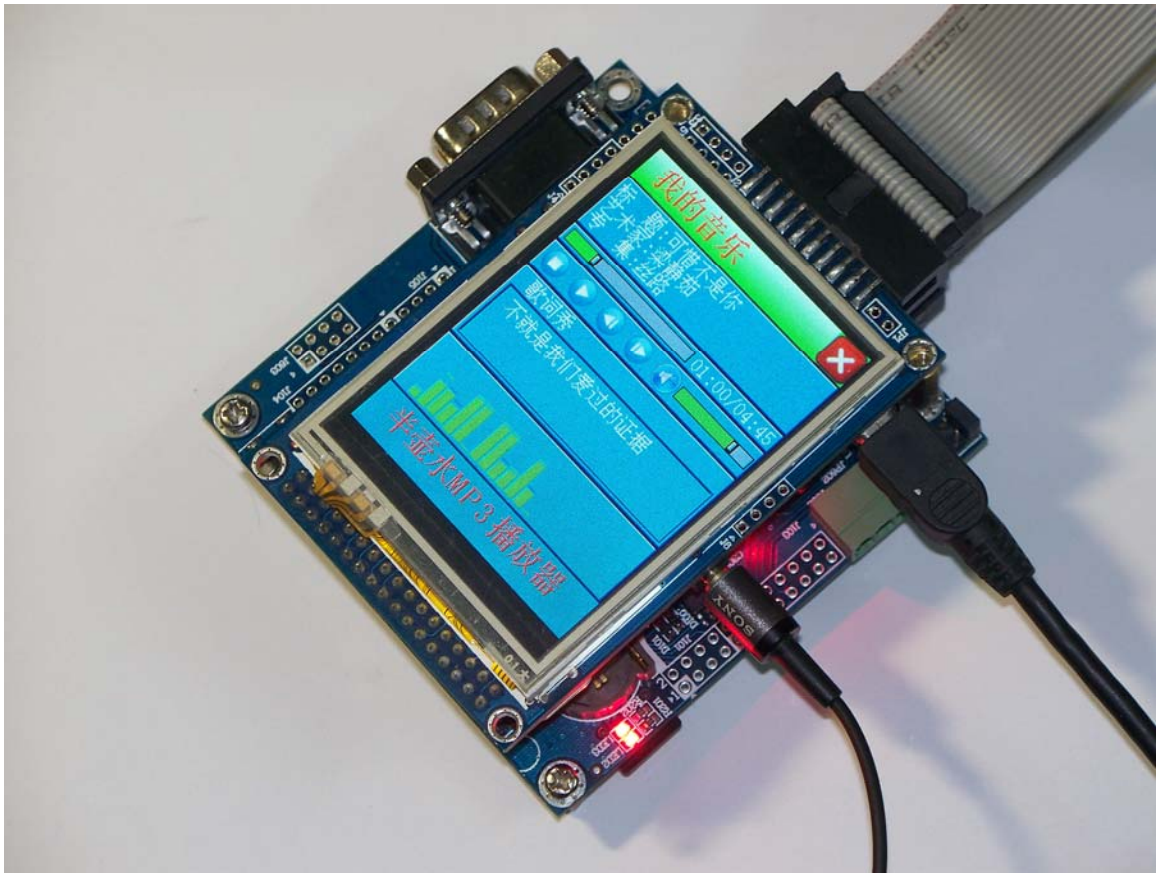
另外提供配套的 NRF24L01/CAN 调试套件

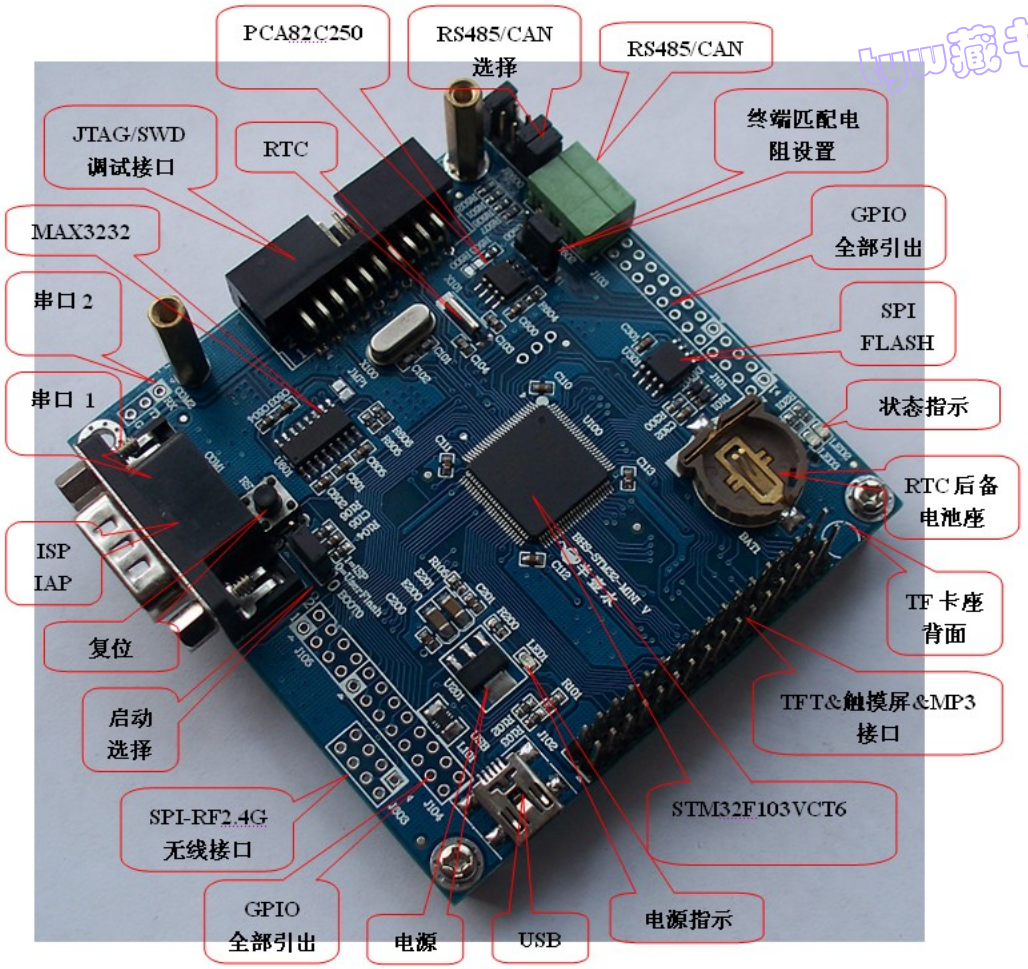
以上硬件表明 BSH-STM32 资源丰富，想玩腻不容易啊

软件方面提供独具特色的 BHS-GUI，原创作品。BHS-GUI 基于控件方式，使用简单，资源占用少。

1.2 BHS-STM32-V 精华版(+FSMC 总线 2.8TFT+MP3+CAN+RS485+JLINK V7)







BHS-STM32-V 开发板(精华版）资源：

.CPU: STM32F103VC, TQFP100 脚；FLASH 容量：256KB,　SRAM 容量：48KB

.1 个 JTAG 调试接口

.1 个电源 LED，2 个状态 LED

.1 个 RS232，支持 3 线 ISP

.1 个 RS485

.1 个 CAN

.1 个 USB

.1 个 TF 卡插座

.1 个 TFT(带触摸屏)接口

.1 个 25F080(1M 字节)的串行 FLASH

.1 个 RTC 后备电池座

.1 个 SPI 2.4G RF 接口(NRF24L01 官方标准接口)

.1 个 VS1003B MP3 硬件解码芯片

说明：本开发板是带 JLINK 硬件仿真器的哦。

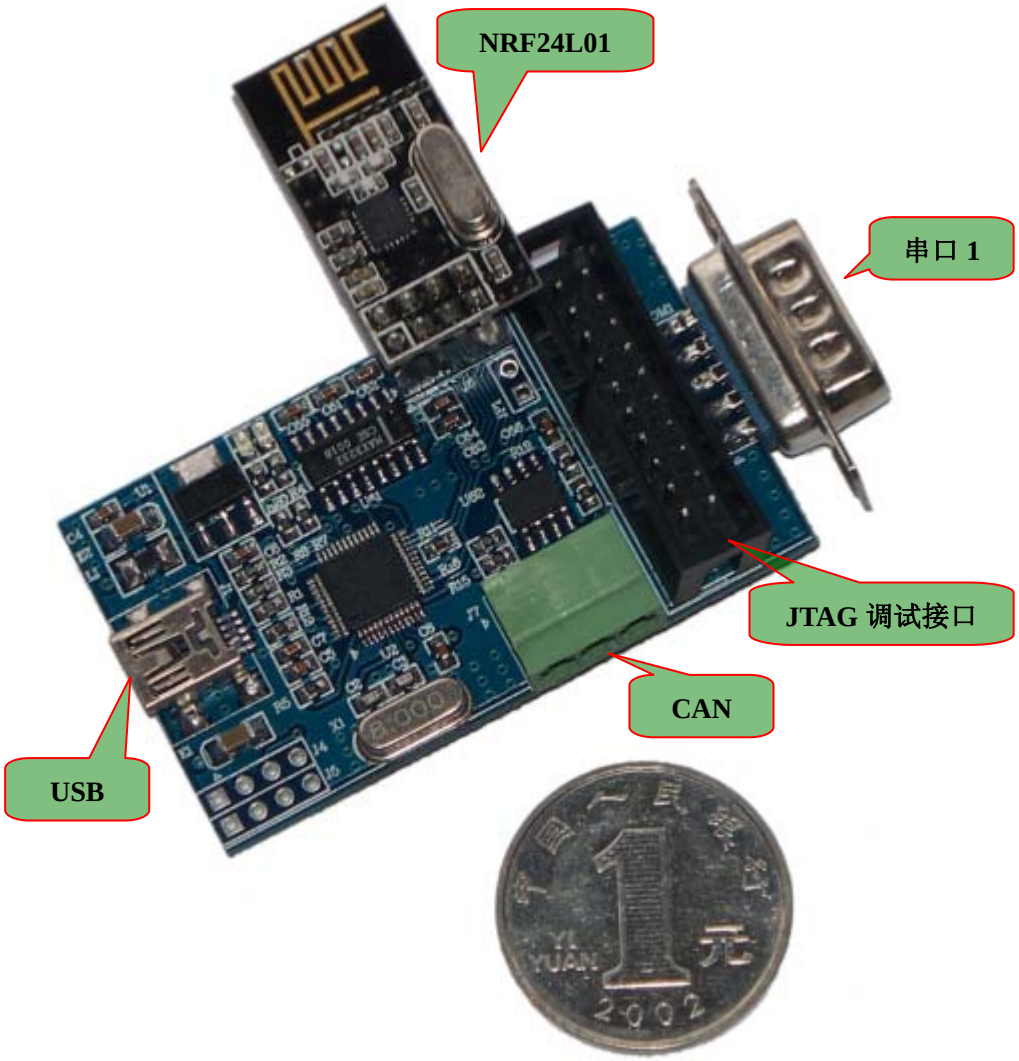
另外提供配套的 NRF24L01/CAN 调试套件

以上硬件表明 BSH-STM32 资源丰富，想玩腻不容易啊

软件方面提供独具特色的 BHS-GUI，原创作品。BHS-GUI 基于控件方式，使用简单，资源占用少。

BHS-STM32 配套多功能开发小板

tyw藏书



CPU:STM32F103C8T6　LQFP48 脚封装，FLASH=64K,RAM=20K

。支持标准 JTAG 调试接口

。支持 SWD 调试接口

。支持 NRF24L01 无线调试，需要 2 个开发小板，或者你自己有其他开发板

。支持 USB 转 RS232，为笔记本没串口的兄弟调试串口程序提供了方便。

。支持 CAN 调试。

USB 转 RS232 非常稳定，比 PL2303HX 系列便宜的 USB 转 232 稳定

1.3 IO资源分配表

BHS-STM32-V 系列开发硬件资源分配完全兼容，所以软件也兼容。

功能	IO	说明
串口 1	PA9	TXD1
	PA10	RXD1
	PB2	ISP 使能，使用该功能时，PB2 只能做输出使用
串口 2	PA2	TXD2
	PA3	RXD2
CAN	PB8	CAN-RXD
	PB9	CAN-TXD
USB	PA11	USB-D-
	PA12	USB-D+
	PA8	USB 使能
SPI-FLASH	PB13	SPI2-SCK
	PB14	SPI2-MISO
	PB15	SPI2-MOSI
	PB6	SPI2-FLASH CS1
SD 卡	PB7	SPI2-SD CS
	PB5	SPI2-SD 卡插入检测
网络	PB0	SPI2-NET CS
	PB1	SPI2-NET RESET
	PB2	SPI2-NET IRQ
SPI-RF	PB10	SPI2-RF CS
	PB11	SPI2-RF CSN
	PD15	SPI2-RF IRQ
TFT	PD14,PD15 PD0,PD1 PE7~PE15 PD9,PD10	TFT 数据总线
	PD7	TFT-CS
	PD11	TFT-RS
	PD5	TFT-WR
	PD4	TFT-RD
	PD2	TFT-RESET
	PD3	TFT 背光
触摸屏	PB13	SPI2-SCK
	PB14	SPI2-MISO
	PB15	SPI2-MOSI
	PD13	触摸屏-中断
	PE0	触摸屏-片选
键盘	PC0~PC7	4X4 矩阵键盘
LED	PC8~PC11	4 个 LED

温度检测	PA0	AD0
红外接收	PC12	
JTAG	PB3	TDO
	PB4	TRST
	PA13	TMS
	PA14	TCK
	PA15	TDI
RTC	PC14	OSC32-IN
	PC15	OSC32-OUT

说明：不同颜色块是不同的功能，另外：SPI2 被多个外设使用

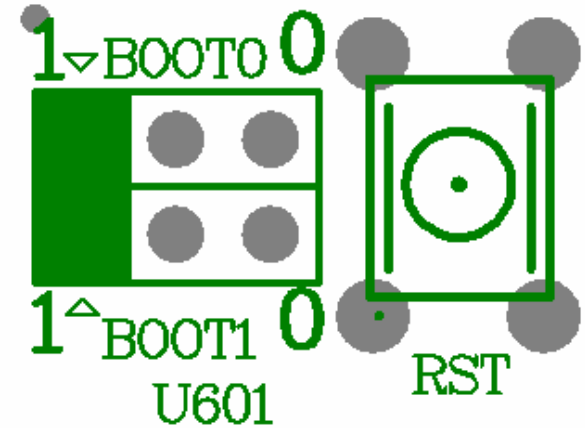
1.3 接口说明

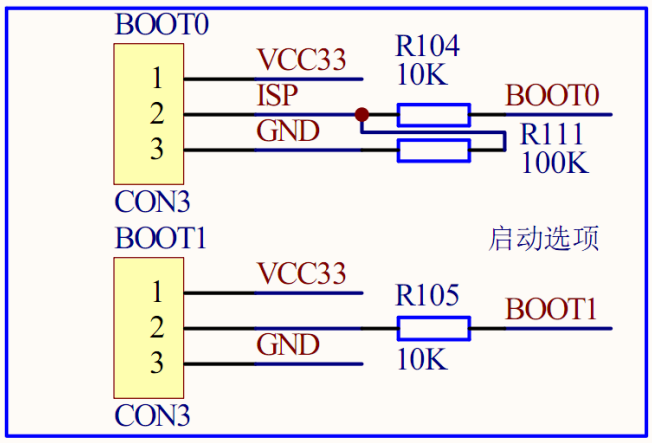
1.3.1 启动选择

BOOT1	BOOT0	启动模式	说明
X	0	用户闪存存储器	用户闪存存储器被选为启动区域
0	1	系统存储器	系统存储器被选为启动区域（即进入 ISP 模式）
1	1	内嵌 SRAM	内嵌 SRAM 被选为启动区域

说明：出场默认设置 BOOT0=0，BOOT1=0。注意 BOOT0 已经通过电阻设置为 0
由表可以看出要实现自动 ISP，设置 BOOT1=0 只需改变 BOOT0 状态就可以在用户模式和系统模式切换

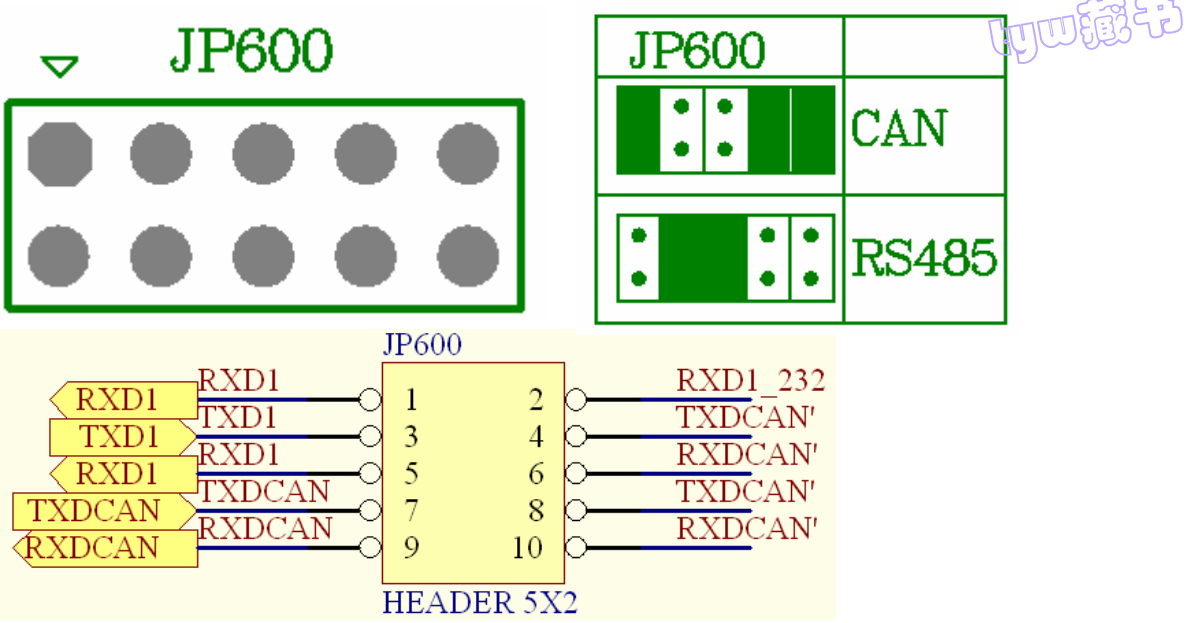
很多人问 RAM 启动有什么作用，当我们在 RAM 中调试程序时，如果启动模式不是设置为 RAM 启动，也可以调试，但是当你按软件复位时，由于启动模式不是 RAM，那么你将不能继续调试程序，必须退出调试状态再重新进入调试才可以。如果你设置是 RAM 启动那么按软件复位后才能继续调试程序。我一般是懒得动跳线





BOOT0	0	1	1
BOOT1	X	0	1
	UserFlash	ISP	SRAM

1.3.2 CAN/RS485/串口选择

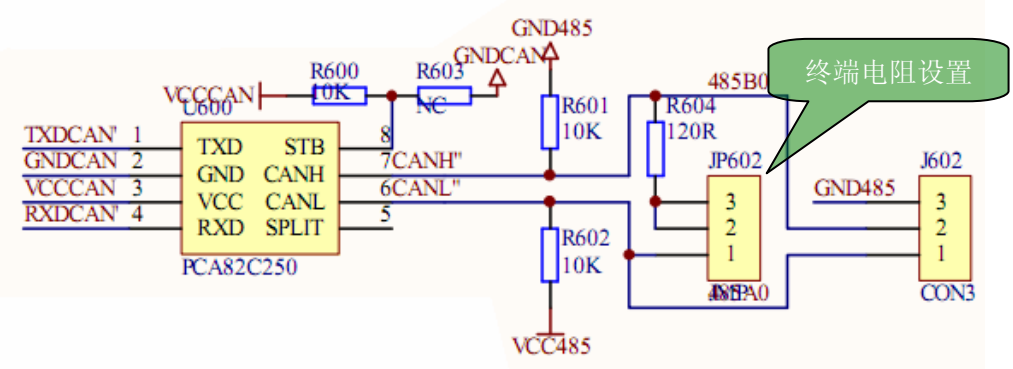


注意：

1. 使用 RS485，请将 JP600 的 1,2　7,8　9,10 断开，3,4　5,6 短接
2. 使用 RS232，请将 JP600 的 3,4　5,6 断开，1,2 短接
3. 使用 CAN，请将 JP600 的 1,2　3,4　5,6 断开，7,8　9,10 短接

4. 本板 RS485 是使用 CAN 芯片实现的，所以串口发送时，禁止接收（参考例子）

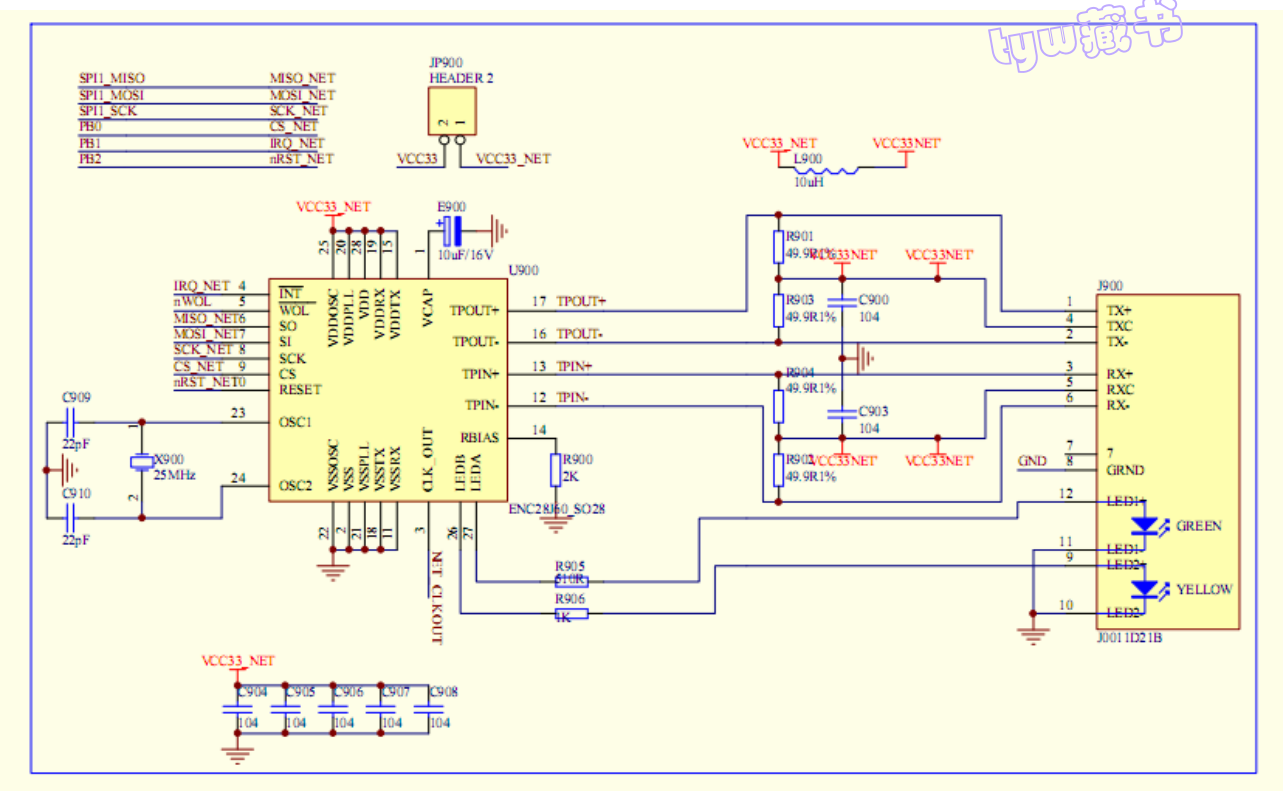
1.3.3 CAN/RS485 原理



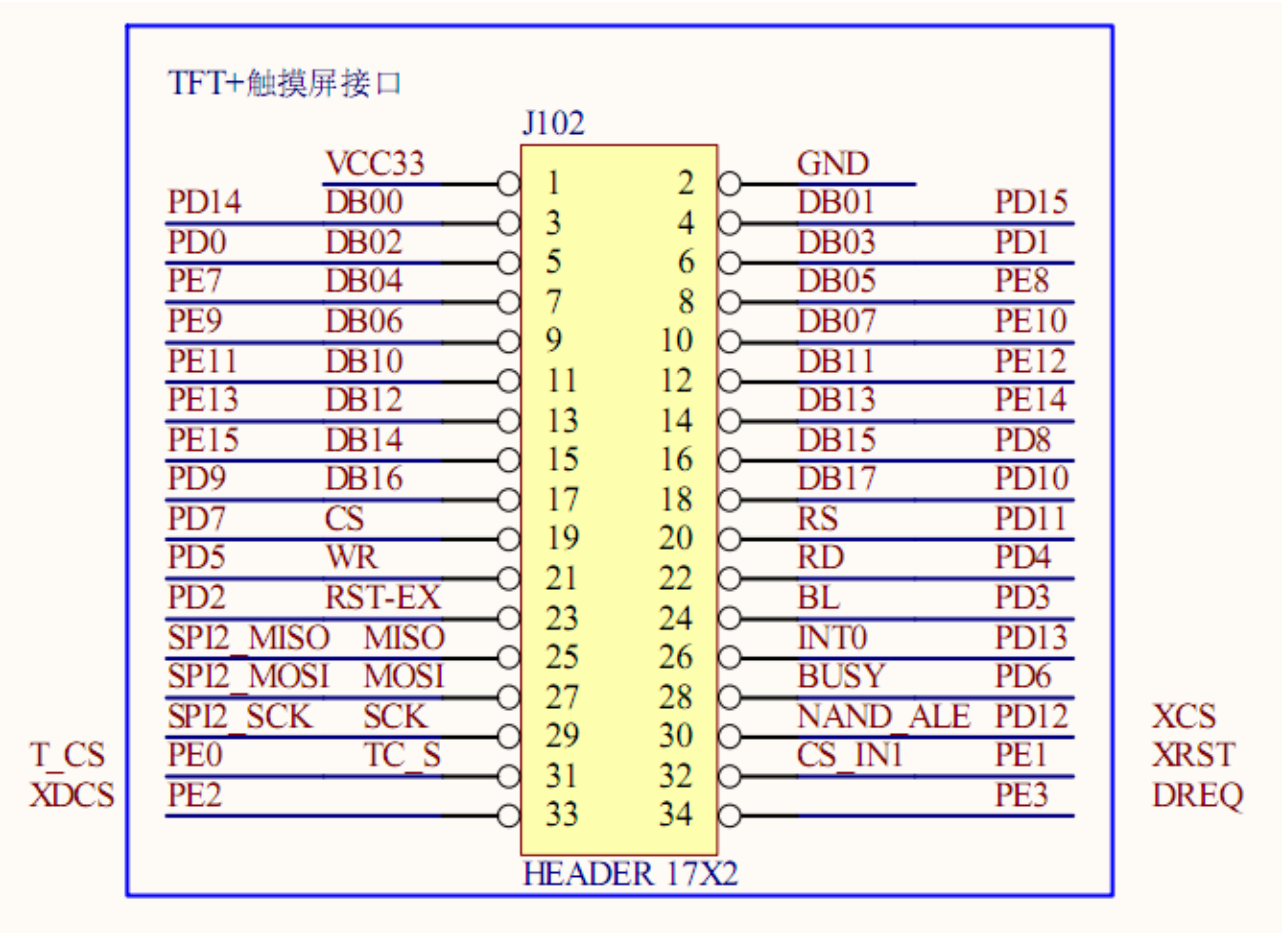
1.3.4 使用CAN芯片实现RS485 网络

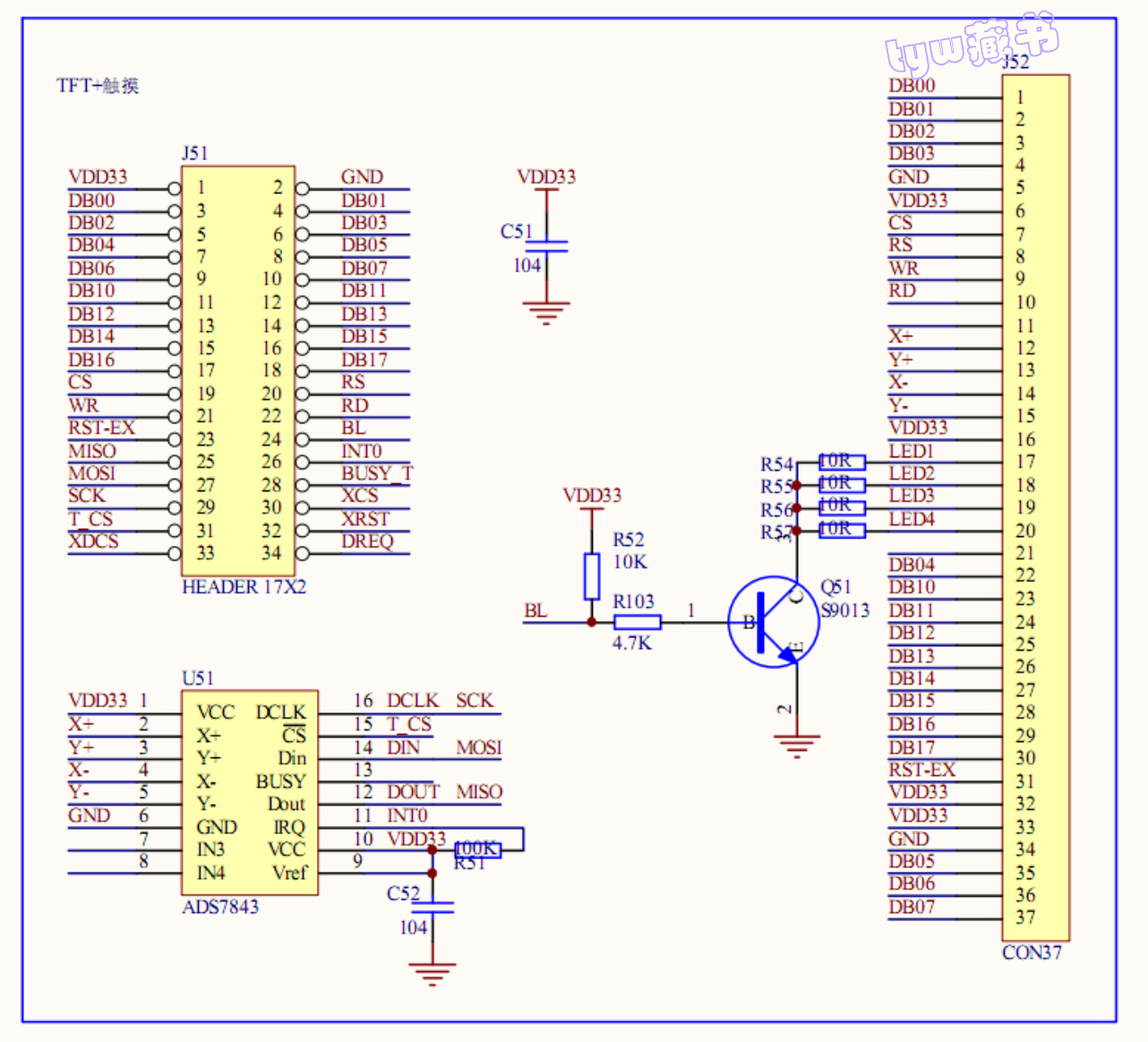
CAN 收发器和 RS485 收发器都是差分信号，CAN 发送时同时接收，所以做 485 使用时发送数据必须禁止数据接收。

1.3.5 网络接口选择（精华板无此功能）

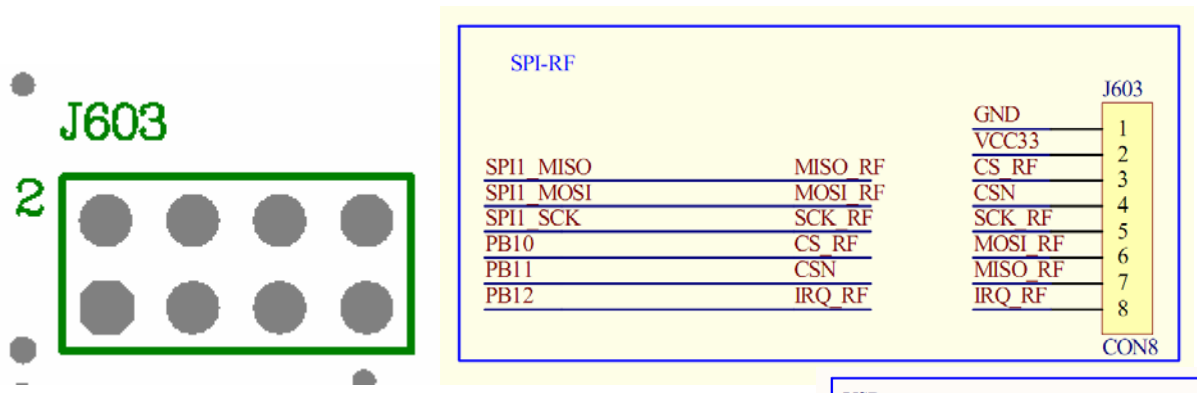


1.3.6 TFT&触摸屏接口&MP3 接口





1.3.7 SPI-RF接口

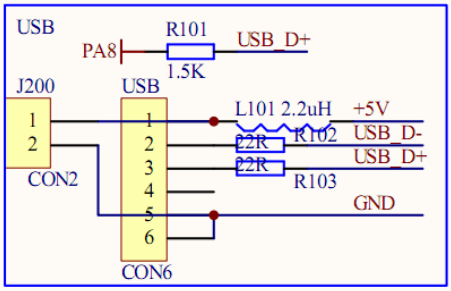


SPI-RF 接口是 NRF24L01 官方标准接口

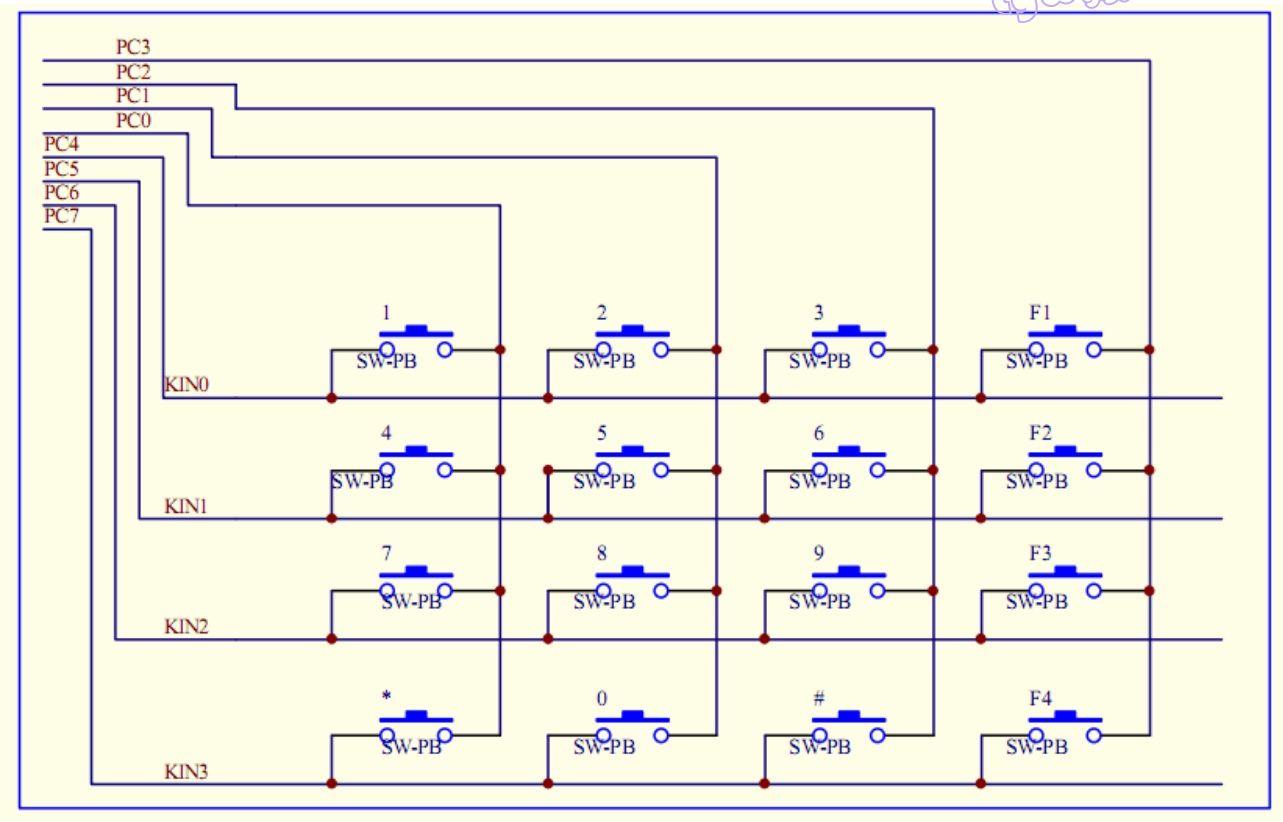
1.3.8 USB接口

USB 使能直接使用 PA8 控制，STM32 输出带推挽输出，

直接控制 USB 使能还是挺简单方便的，关键是实际项目还省成本

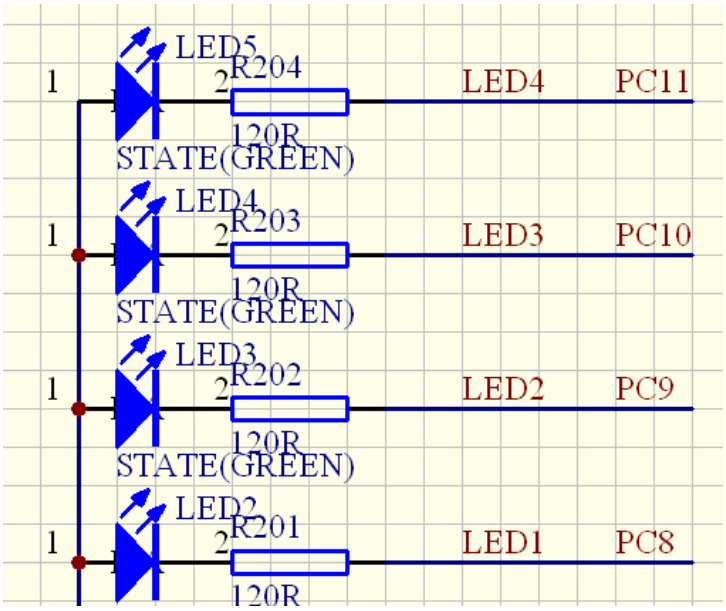


1.3.9 键盘接口（精华板无此功能）

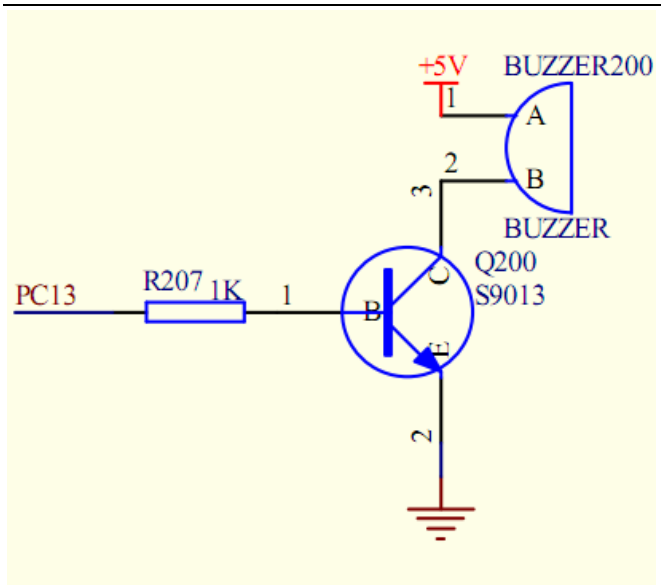


4X4 矩阵键盘，这部分不多说

1.3.10 LED状态灯（精华板只有LED2,LED3）

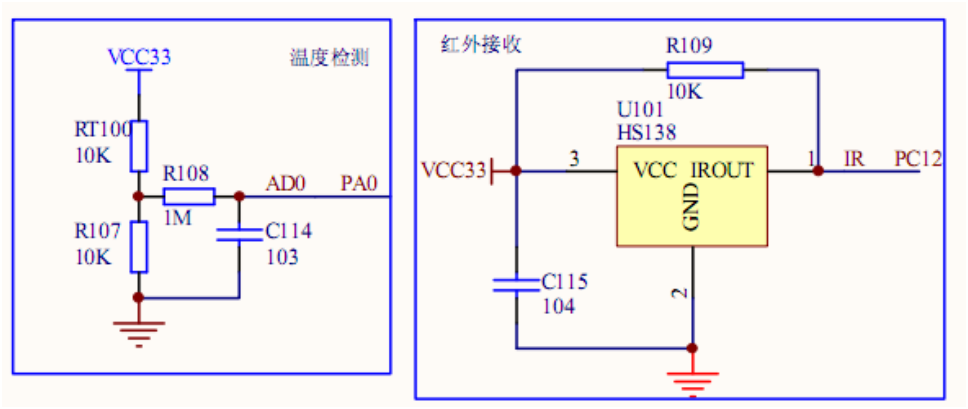


1.3.11 蜂鸣器接口（精华板无此功能）

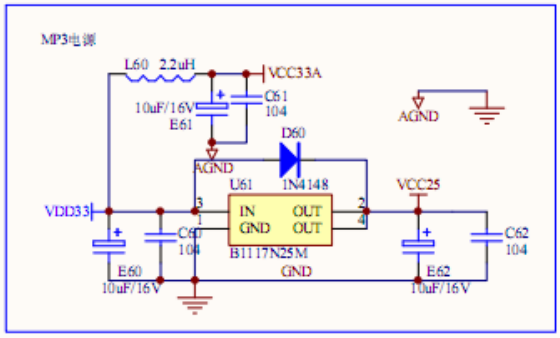


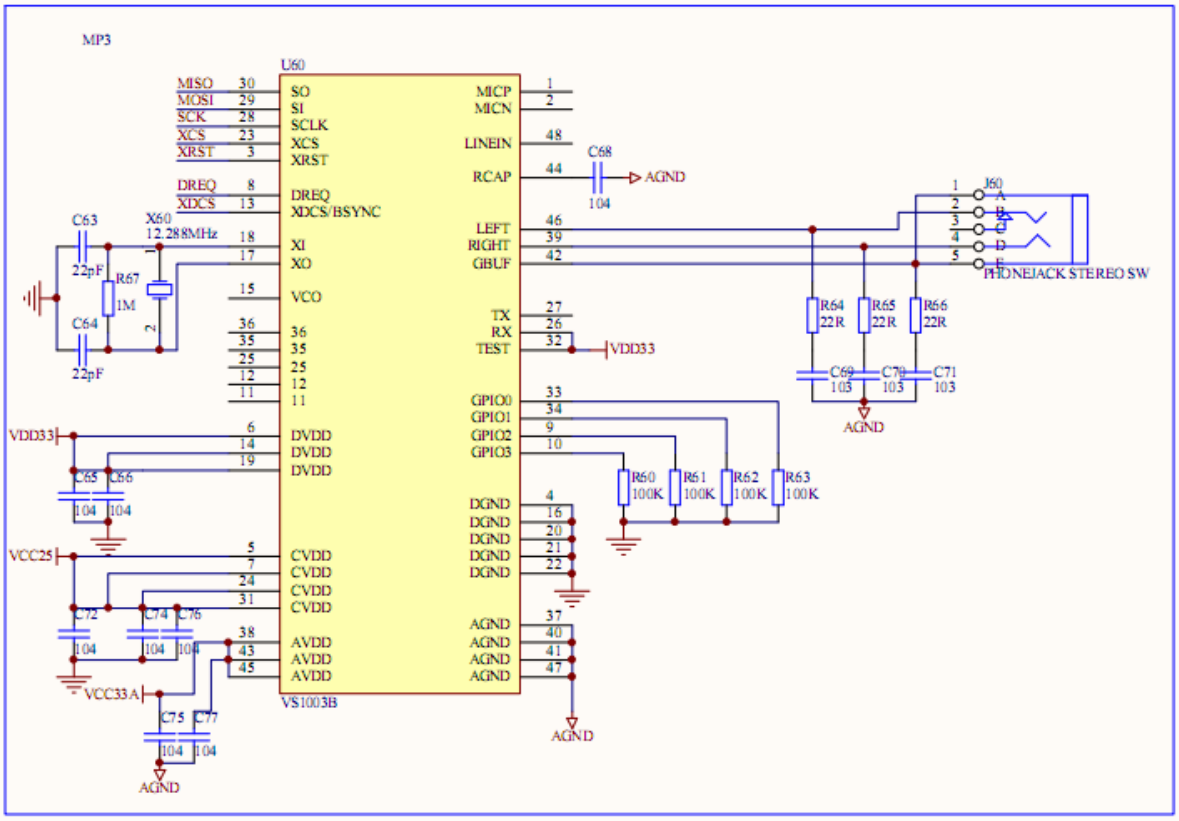
tyw藏书

1.3.12 温度检测、红外接收（精华板无此功能）



1.3.13 MP3(MP3 实际在TFT模块背面，没在开发底板上面的)

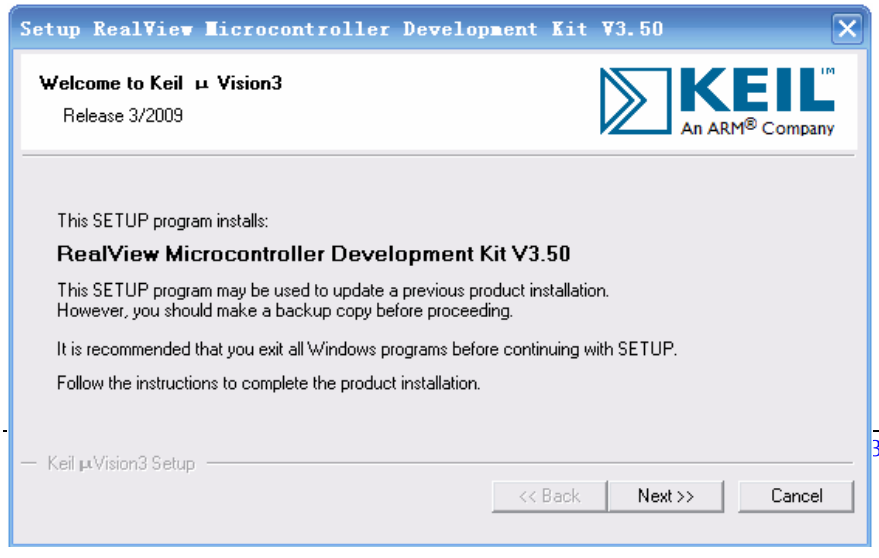




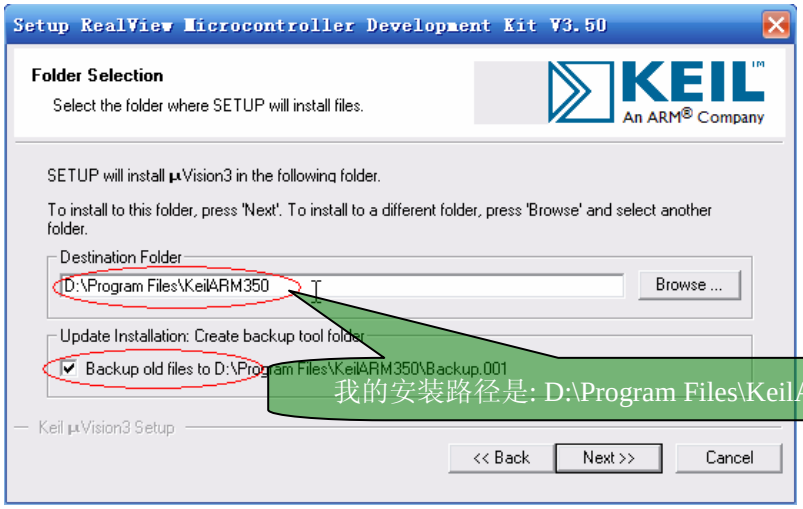
二、开发环境搭建

2.1 KEIL MDK3.5/4.12 安装

说明：目前例子使用 MDK4.12 了，大家安装时请安装 MDK4.12,安装方法与 MDK3.50 完全一样
下面只做简单介绍，更详细的请看视频教程
一般跟着提示选择 NEXT 就可以，下面介绍几个关键部分

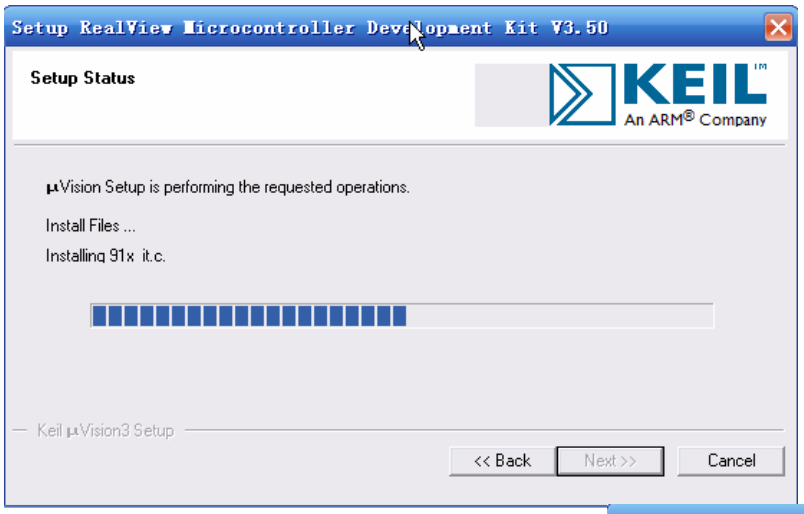


tyw藏书



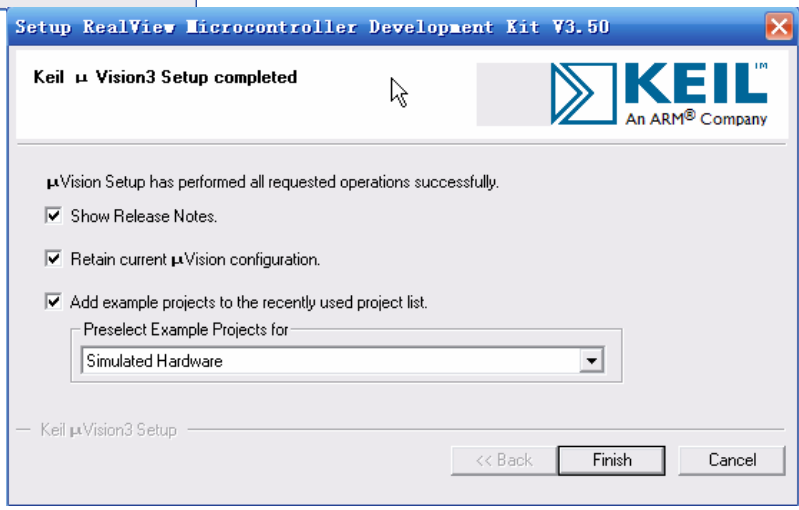
这里安装路径最好和我的保持一致，因为所有的使用了 ST 库函数例子都要设置库文件路径，这样以后方便些

因为我的电脑原来安装了 MDK 所以提示备份旧文件，也可以不备份



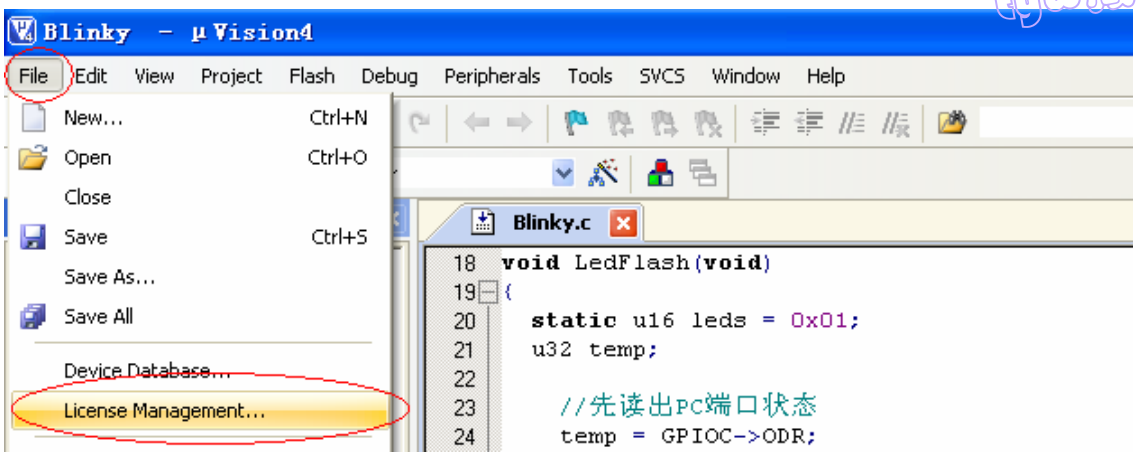
点【Next】后就开始安装了

不要一会就安装完了

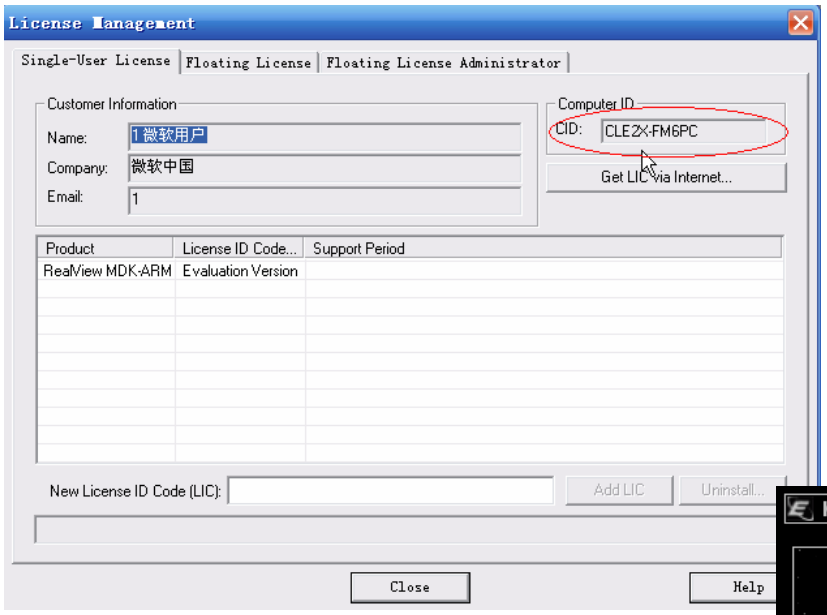


安装好 MDK 后就可以使用了，不过有 32K 代码限制，下面介绍如何破解

声明：此法仅仅提供学习之用，商业用途请购买正版软件

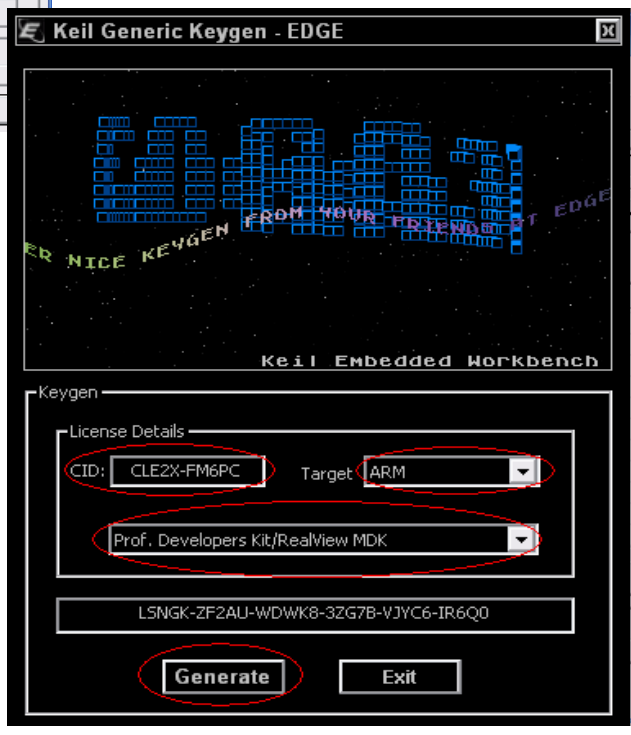


先打开 Keil uVision4 软件，
选择 File->License Management
出现下面对话框



复制[CID]

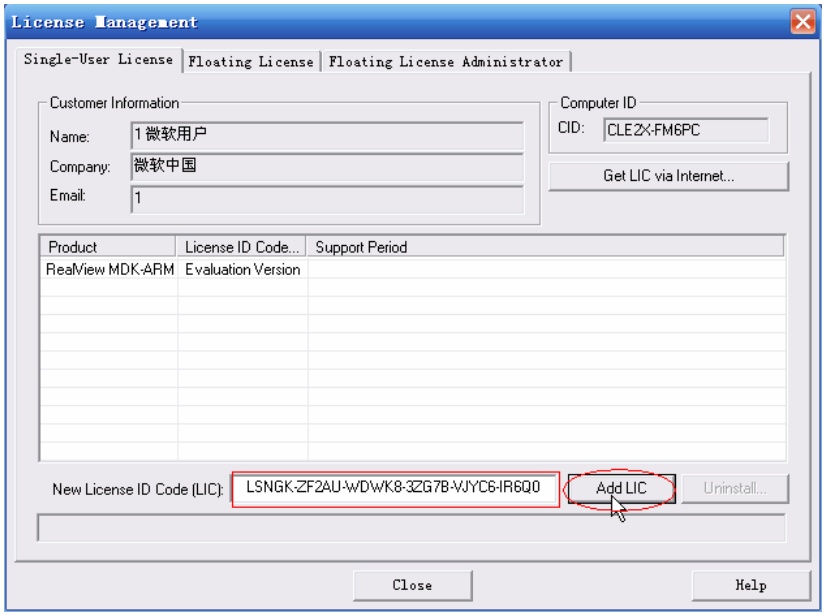
打开 MDK3.50 注册机
MDK3.50,MDK4.12 通用的



注意：
使用时一般要关闭杀毒软件，因为杀毒软件都认为是病毒的

将 CID 复制到注册机
选择 ARM
选择 MDK
点击【Generate】生成序列码，将生成的系列码复制到 KEIL 中

tyw藏书



点击【Add LIC】即可
KEIL MDK 安装完毕

2.2 JLINK仿真器驱动安装安装

JLINK 仿真器应该是开发 ARM 非常好的工具，驱动也非常简单，都是跟着点 NEXT 就可以了，这里就不做描述了。安装完 JLINK 驱动，将 JLINK 的 USB 电缆插入电脑，电脑立即提示找到 JLINK,说明一切正常到此为止开发环境已经搭建完成

三、开发环境熟悉

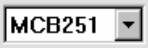
3.1 KEIL MDK介绍

也许很多人在 IAR 和 KEIL 两者之间徘徊，确实根据反应，两者都非常不错，如果你用过 KEIL C51,那么 KEIL MDK 用起来就很上手，个人觉得 KEIL 使用简单，调试方便。况且 KEIL 目前已经被 ARM 公司收购，如果关注 KEIL 的话，应该知道 KEIL MDK 软件版本升级非常之快，可见 ARM 公司对 KEIL 的支持是相当大的。如果有人问我选 KEIL 还是 IAR, 个人建议当然是用 KEIL 了。你如果对 IAR 情有独钟也没关系，IAR 也有很多忠实粉丝的，1 个开发工具而已，什么顺手用什么。
目前例程已经级为 4.12 版本

本文关于 MDK 的说明均来自 KEIL 官方文档《UV3.chm》
另外关于 RTX, 文件系统等参考《rlarm.chm》
这 2 个文件在安装路径的 HLP 文件下，另外光盘里也有中文版本

3.2 KEIL MDK常用工具及快捷方式

Project菜单和Project命令

Project 菜单	工具条	快捷键	功能描述
New Project...			创建一个新工程
Import µVision1 Project...			导入一个工程
Open Project...			打开一个工程
Close Project			关闭当前工程
Components, Environment, Books...			维护工程组件、配置工具环境及管理书
Select Device for Target			从设备库中选择CPU
Remove Item			从工程中移出组或文件
Options for Target			改变目标、组、文件的工具选项
		Alt+F7	改变当前目标的工具选项
			选择当前目标
Build target		F7	翻译已修改的文件及编译应用
Rebuild all target files			重新翻译所有的源文件并编译应用
Translate...		Ctrl+F7	翻译当前文件
Stop Build			停止编译当前程序
1 - 9			打开最近使用的工程文件

Debug菜单和Debug命令

Debug 菜单	工具条	快捷键	功能描述
Start/Stop Debug Session		Ctrl+F5	启动或停止µVision3调试模式
Go		F5	运行到下一个活动断点
Step		F11	单步运行进入一个函数
Step Over		F10	单步运行跳过一个函数
Step Out of current Function		Ctrl+F11	从当前函数跳出
Run to Cursor Line			运行到当前行
Stop Running		ESC	停止运行
Breakpoints...			打开断点对话框
Insert/Remove Breakpoint			在当前行设置断点
Enable/Disable Breakpoint		Alt+F7	Enable/disable当前行的断点
Disable All Breakpoints			使程序中的所有断点无效
Kill All Breakpoints		F7	去除程序中的所有断点
Show Next Statement			显示下一条要执行的指令
Enable/Disable Trace Recording		Ctrl+F7	使能跟踪刻录
View Trace Records			浏览前面执行的指令
Execution Profiling			记录执行时间
Setup Logic Analyzer			打开逻辑分析仪对话框
Memory Map...			打开存储器映射对话框
Performance Analyzer...			打开性能分析仪对话框
Inline Assembly...			打开在线汇编对话框
Function Editor (Open Ini File)...			编辑调试函数及调试初始化文件

Debug 菜单	工具条	快捷键	功能描述
Download			按照配置下载到FLASH中

说明：此按键是下载程序到 FLASH 中，在 RAM 中调试绝对不能点此按钮

Debug 菜单	工具条	快捷键	功能描述
Reset CPU	RST		重启CPU

tyw藏书

Edit菜单和Edit命令

Edit 菜单	工具条	快捷键	功能描述
		Home	将光标移到当前行的开始
		End	将光标移到当前行的结束
		Ctrl+Home	将光标移到当前文件的开始
		Ctrl+End	将光标移到当前文件的结束
		Ctrl+Left Arrow	将光标移到当前单词的左侧
		Ctrl+Right Arrow	将光标移到当前单词的右侧
		Ctrl+A	选中当前文件中的所有内容
			把光标返回到执行'find'或'go to line'命令前的位置
			把光标移到到执行'find'或'go to line'命令后的位置
Undo		Ctrl+Z	撤消键入
Redo		Ctrl+Y	恢复键入
Cut		Ctrl+X	剪切
Copy		Ctrl+C	复制
Paste		Ctrl+V	粘贴
Indent Selected Text			向右缩进选定的文本
Unindent Selected Text			I向左缩进选定的文本
Toggle Bookmark		Ctrl+F2	在当前行设置标签
Goto Next Bookmark		F2	将光标移到下一个标签
Goto Previous Bookmark		Shift+F2	将光标移到前一个标签
Clear All Bookmarks		Ctrl+Shift+F2	消除所有的标签
Find	command	Ctrl+F	查找
		F3	重复向前查找
		Shift+F3	重复向后查找
		Ctrl+F3	在光标之下查找
Replace		Ctrl+H	替换
Find in Files...		Shift+Ctrl+F	在几个文件内查找
Incremental Find		Ctrl+I	增量查找
Outlining			有关源代码的命令
Advanced			编辑器命令
Configuration			改变着色、字体、快捷键

在文件中查找

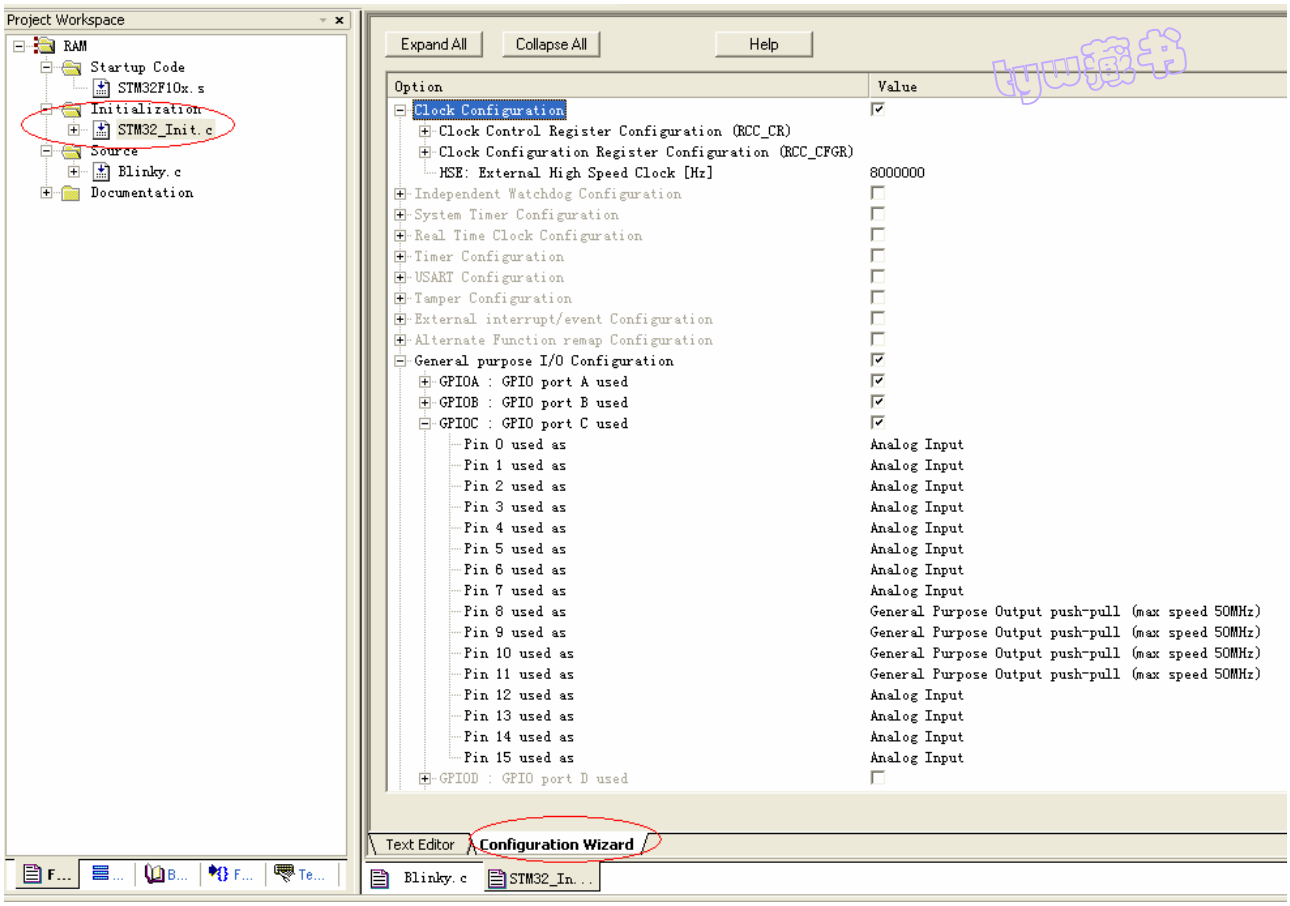
Find in Files...	Shift+Ctrl+F	在几个文件内查找

说明：有不少人哪个函数在哪个文件中，某个变量在哪里定义的，大家要学会用工具哈，KEIL 的这个文件查找工具是非常不错的

3.3 MDK配置向导

MDK 配置向导是 KEIL 提供的图形化硬件初始化工具，【\BHS-STM32 例程\基础例程-非库函数(入门篇)】里面的例程就是基于此工具初始化硬件。（后面将有介绍）

下图可以看到大概的面貌，一般初始化文件是 STM32_Init.c，注意只有切换到[Configuration Wizard]标签才能看到图形化配置方式，因为实际 STM32_Init.c 就是个文本文件，切换到[Text Editor]标签就能看到了



只要按照一定格式就能被 KEIL 软件识别，下面我们来详细分析其中奥妙。有了他，初始化硬件的工作变的简单很多了。

配置文件必须以下面的格式开始(必须的)，文件名称没限制

//-------- <<< Use Configuration Wizard in Context Menu >>> ----------

<<< Use Configuration Wizard in Context Menu >>>

配置向导以下面的格式结束（可以不要的）

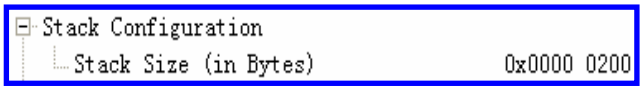
<<< end of configuration section >>>

关键字	说明
<h>	头标签：表示包含多个子项的一个组，无开关控制
<e> <e0>，<e1> <e1.4>	使能标签：表示包含多个子项的一个组，该组开关由这项控制（后面有个 ☑ ） 0，1， 表示操作对象序号，后面有解释 1 表示操作对象序号，4 表示 BIT4
</h> or </e>	头标签、使能标签结束
<i>	提示信息，可以连续写几条提示信息。当鼠标放在要操作项目上将出现提示信息
<q>§	位操作用一个 ☐ 控制， ☐ =0， ☑ =1
<o>§ <o0>，<o1> <o1.4..6>	数字标签 0，1， 表示操作对象序号 1 表示操作对象序号，4..6 操作对象的 BIT6～BIT4
<s>	字符串标签：操作的对象是字符串

<s1.30>	操作的对象是字符串，并且限制字符串长度（这里是 30）
<0-31>	数值范围，超过该范围无意义
<0-100:10>	10 进制数值范围
<0x40-0x1000:0x10>	16 进制数值范围
<0=> <1=> <2=>.....	数值选择，后面可以跟注释文字，如：<0=> SYSTICKCLK = HCLK/8
<#+1>　<#-1> <#*8>　<#/3>	数值做运算，分别表示+1，-1，　*8，　/3

<h>标签

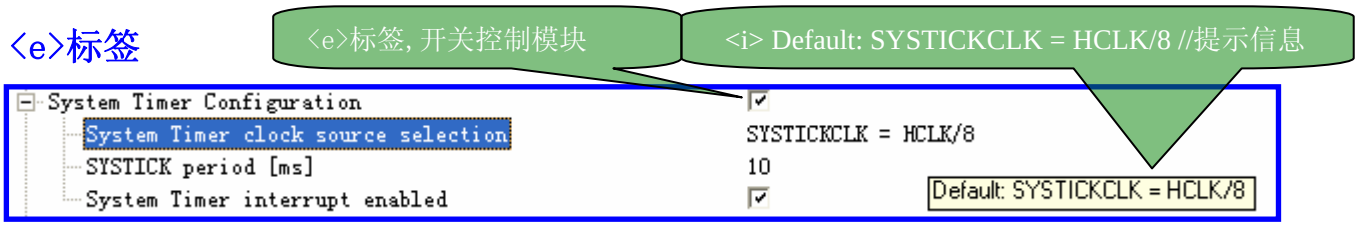
最简单的<h></h>例子是.s 文件中的堆栈配置



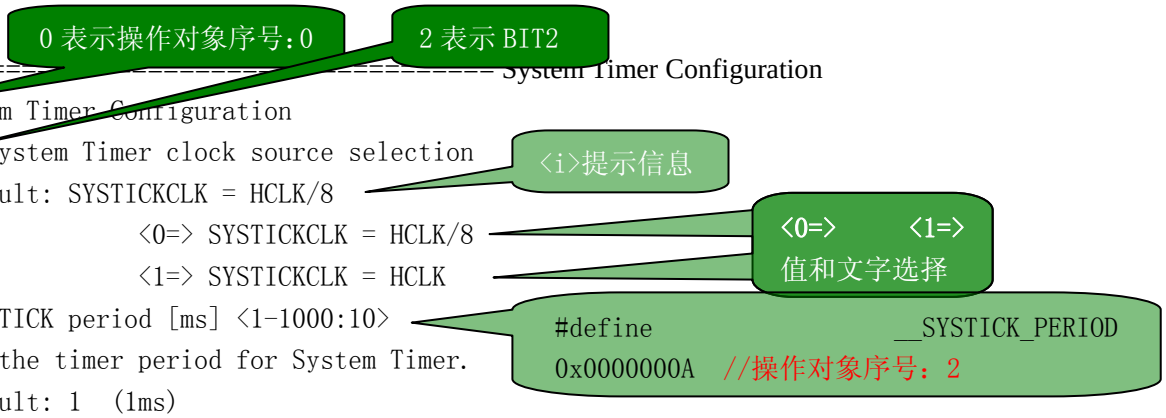
下面是代码

```
;// <h> Stack Configuration
;//   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
;// </h>
Stack_Size      EQU     0x00000200
```

<e>标签



系统时钟配置




```
//=============== ======== ========== System Timer Configuration
// <e0> System Timer Configuration
//   <o1.2> System Timer clock source selection
//   <i> Default: SYSTICKCLK = HCLK/8
//                    <0=> SYSTICKCLK = HCLK/8
//                    <1=> SYSTICKCLK = HCLK
//   <o2> SYSTICK period [ms] <1-1000:10>
//   <i> Set the timer period for System Timer.
//   <i> Default: 1  (1ms)
//   <o1.1> System Timer interrupt enabled
// </e>
#define __SYSTICK_SETUP          1                  //操作对象序号：0
#define __SYSTICK_CTRL_VAL       0x00000006         //操作对象序号：1
#define __SYSTICK_PERIOD          0x0000000A        //操作对象序号：2
```

解释：

<o>标签

<e0> <o1.2> <o2> <o1.1>

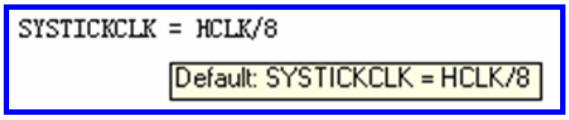
字母后的第 1 个数值表示要操作对象的序号，.1　.2　‘.’点后面的数字表示要操作对象的位

例：.1=BIT1，.2= BIT2

<o1.2>表示操作的第 1 个对象的 BIT2

<o1.2..5>表示操作的第 1 个对象的 BIT5，BIT4，BIT3，BIT2

<i> 前一个项目提示帮助，将鼠标放在要操作项目上将出现提示信息



<0=>，<1=>：值和文字的选择

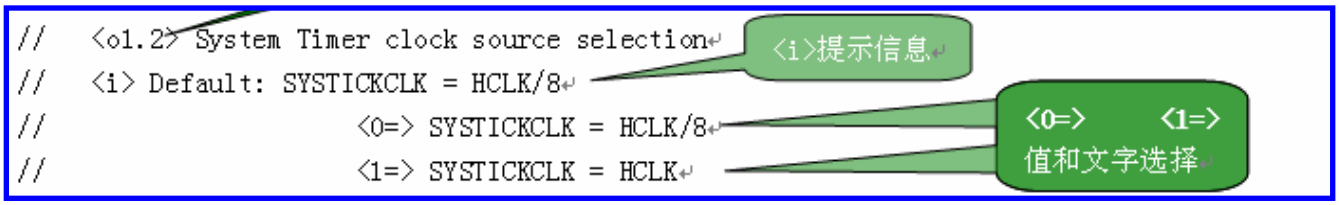
例：


```
//       <0=> SYSTICKCLK = HCLK/8
//       <1=> SYSTICKCLK = HCLK
```

<e> </e>是成对出现，包含一对<e> </e>里面的所以项通过该复选框启用/禁止

#define __SYSTICK_SETUP　　0　　//注意这里为 0 表示未禁止

下面 4 行整体意思表示：



系统定时器时钟频率由第 1 个操作数的 BIT2 决定

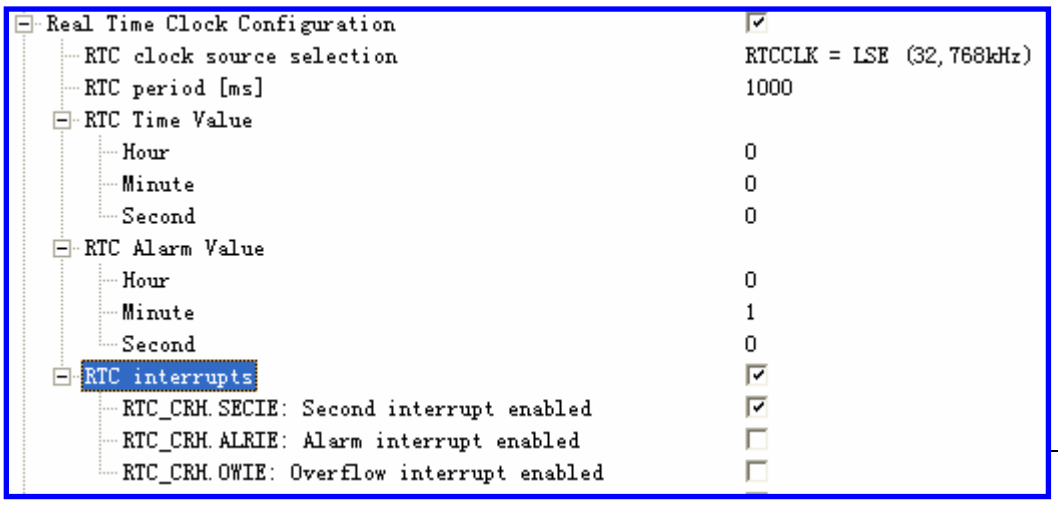
0：SYSTICKCLK = HCLK/8；　　1：SYSTICKCLK = HCLK

操作对象序号说明：

```
//   <e1.13> Enable Chip Select 0 (CSR0)
//   </e>
_WDWORD(0xFFE00000, 0x010024A9);    // EBI_CSR0: Flash
```

上面一行表示 2 个操作对象

例 2：RTC 配置



```
//========================================== Real Time Clock Configuration
// <e0> Real Time Clock Configuration
//   <o1.8..9> RTC clock source selection
//   <i> Default: No Clock
//                     <0=> No Clock
//                     <1=> RTCCLK = LSE (32,768kHz)
//                     <2=> RTCCLK = LSI (32 kHz)
//                     <3=> RTCCLK = HSE/128
//   <o2> RTC period [ms] <10-1000:10>
//   <i> Set the timer period for Real Time Clock.
//   <i> Default: 1000  (1s)
//   <h> RTC Time Value
//     <o3> Hour <0-23>
//     <o4> Minute <0-59>
//     <o5> Second <0-59>
//   </h>
//   <h> RTC Alarm Value
//     <o6> Hour <0-23>
//     <o7> Minute <0-59>
//     <o8> Second <0-59>
//   </h>
//   <e9> RTC interrupts
//     <o10.0> RTC_CRH.SECIE: Second interrupt enabled
//     <o10.1> RTC_CRH.ALRIE: Alarm interrupt enabled
//     <o10.2> RTC_CRH.OWIE: Overflow interrupt enabled
//   </e>
// </e>
#define __RTC_SETUP             0           //操作对象序号：0
#define __RTC_CLKSRC_VAL        0x00000100  //操作对象序号：1
#define __RTC_PERIOD            0x000003E8  //操作对象序号：2
#define __RTC_TIME_H            0x00        //操作对象序号：3
#define __RTC_TIME_M            0x00        //操作对象序号：4
#define __RTC_TIME_S            0x00        //操作对象序号：5
#define __RTC_ALARM_H           0x00        //操作对象序号：6
#define __RTC_ALARM_M           0x01        //操作对象序号：7
#define __RTC_ALARM_S           0x00        //操作对象序号：8
#define __RTC_INTERRUPTS        0x00000001  //操作对象序号：9
#define __RTC_CRH               0x00000001  //操作对象序号：10
```

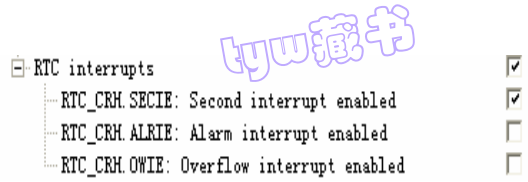
解释：

```
//   <h> RTC Time Value
//     <o3> Hour <0-23>
//     <o4> Minute <0-59>
//     <o5> Second <0-59>
//   </h>
```

<h></h>是成对出现，表示里面的子项都属于同一类，同一组，可以嵌套使用

<0-23> 数值范围,超过范围是无效的

```
//   <e9> RTC interrupts
//     <o10.0> RTC_CRH.SECIE: Second interrupt enabled
//     <o10.1> RTC_CRH.ALRIE: Alarm interrupt enabled
//     <o10.2> RTC_CRH.OWIE: Overflow interrupt enabled
//   </e>
```



注意：<h></h>和<e></e>区别是<h>是无参数的，只是表示包含子项是一个组，<e>是有参数的比如<e0>，<e1>，<e>项目后面还有个 ，这是<h>没有的。

<q>标签

<q>标签实际就是最简单的位操作标签，位操作用一个 控制， =0， =1


```
//   <q0.4>          q 标签测试
#define       QLABLE      0x00000000
```

BIT4 变化了

```
//   <q0.4>          q 标签测试
#define       QLABLE      0x00000010
```

<s>标签

<q>标签就是字符串标签啦，这个标签很少用到

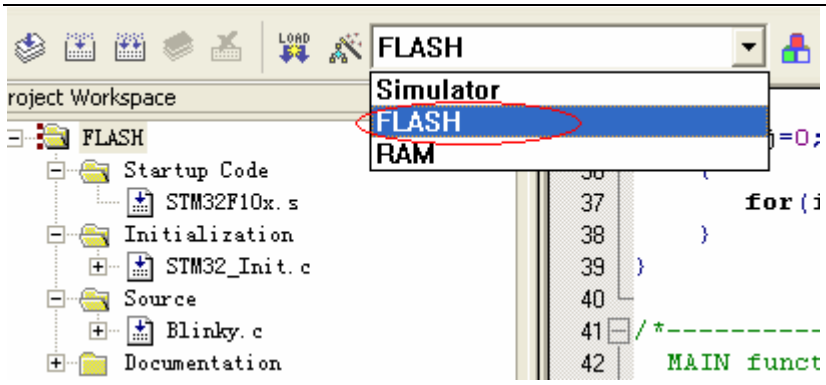
无限制的字符

限制的字符串长度为 30

```
// <s> Change ID
#define ID  "My User ID  8"

// <s0.30> Change Password String
char pw[] = "My Password1234567890123456789";
```

3.4 在FLASH中调试程序

下面以【\基础例程-非库函数(入门篇)\实验 1-GPIO-LED 闪灯(软件延时方式)】例子来做介绍，例子都用 MDK+JLINK 说明



tyw藏书

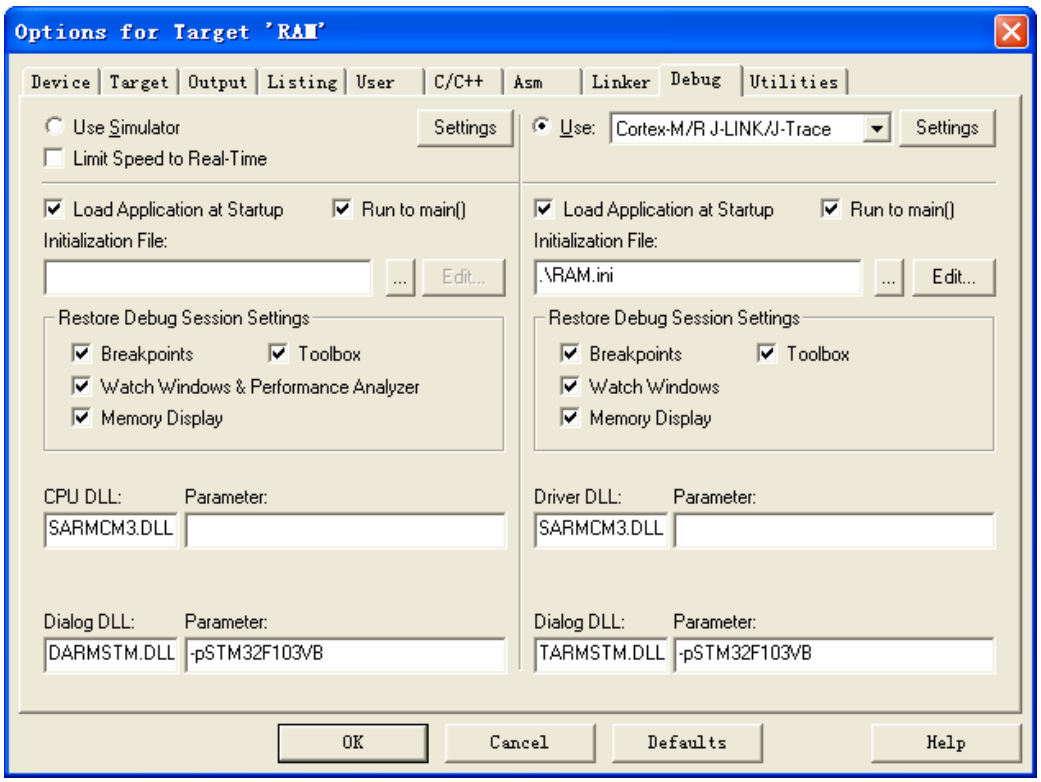
【Simulator】从 0x8000000 开始载入程序，该项是纯软件调试。例程部分外设是无法用软件调试的，请使用硬件仿真调试。

【FLASH】从 0x8000000 开始载入程序调试

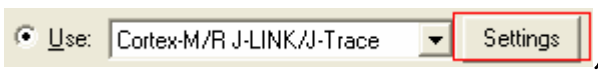
【RAM】从 0x20000000 开始载入程序调试

说明：STM32 可以从 FLASH/RAM 运行程序（这点与其他比如 51，AVR 是不同的），在 FLASH 中调试将减少 STM32 的使用寿命，STM32 正常是 10000 次擦写，一般小的程序建议在 RAM 中调试，RAM 中调试不影响 STM32 的寿命，不过 RAM 掉电数据就丢失的，所以产品最后正常运行还是将程序写入 FLASH 的

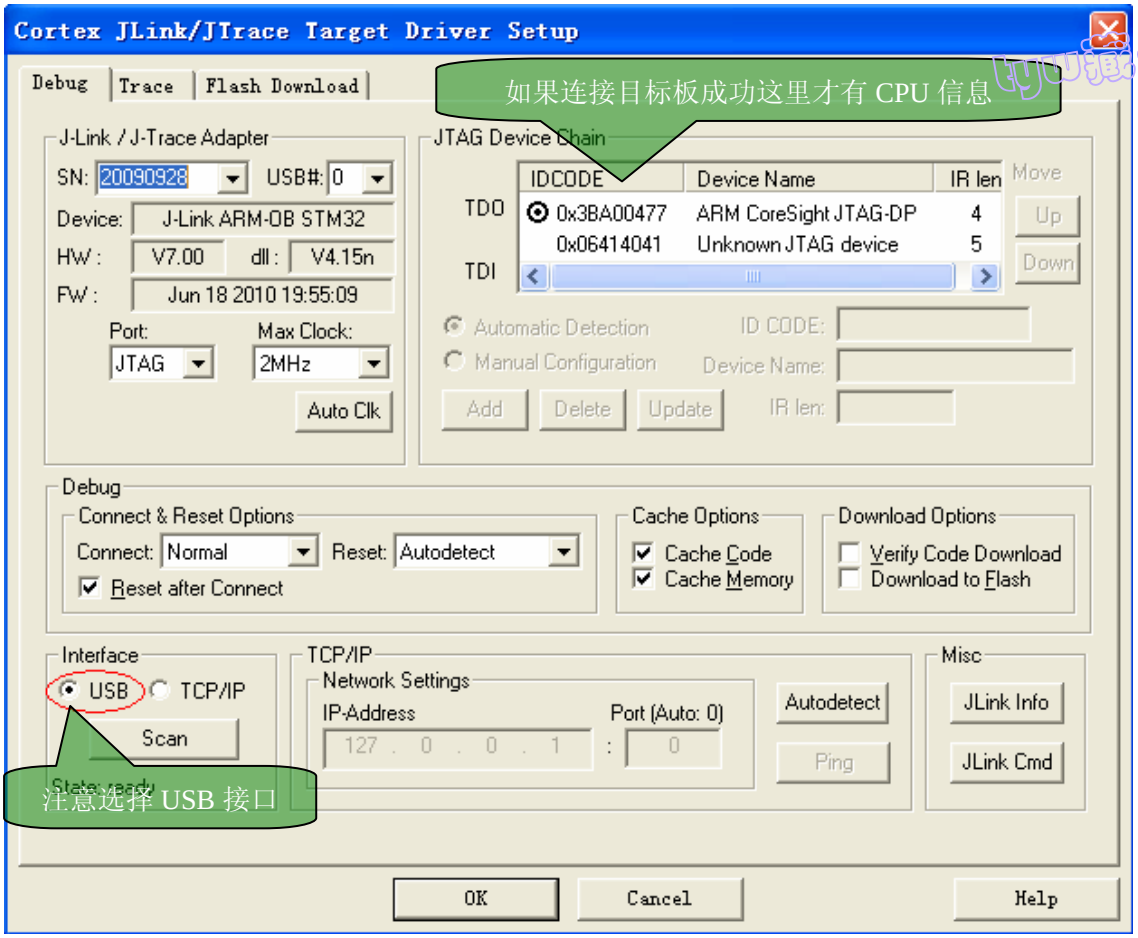
1.点击【项目配置】按钮弹出【项目配置】对话框，在[Debug]标签中选择【Cortex-M3 J-LINK】



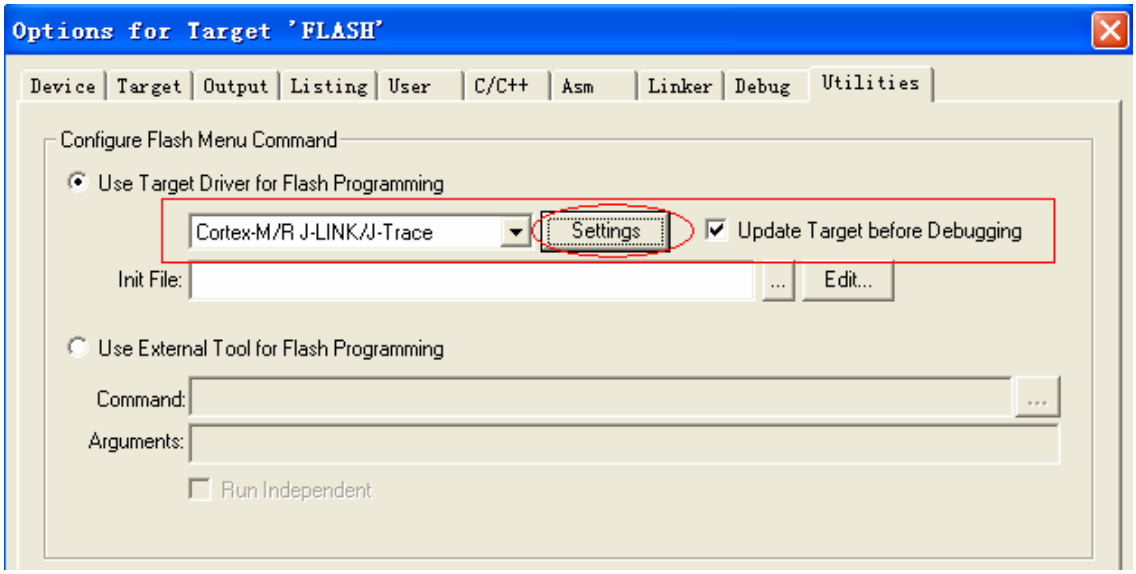
2.设置 JLINK

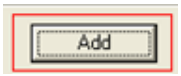


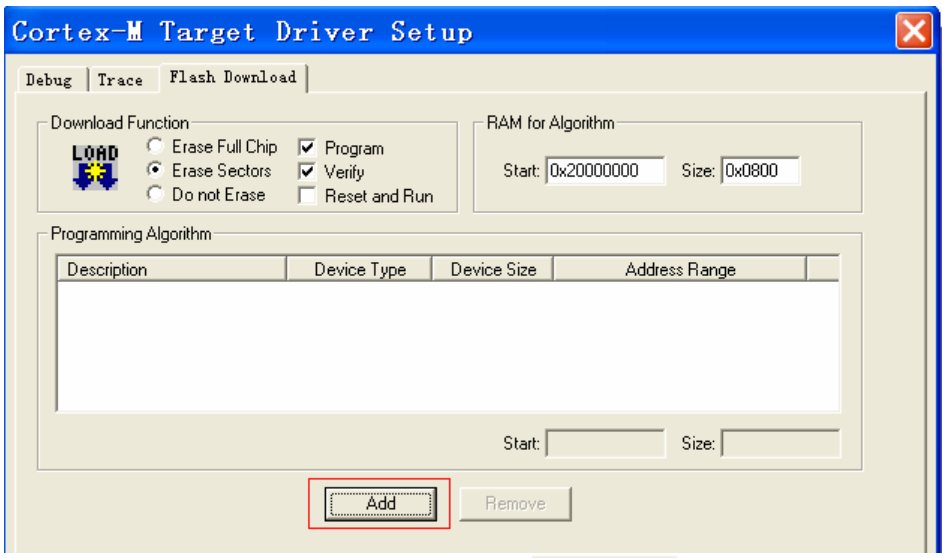
点击[settings]弹出设置对话框



继续选择[Utilities]标签，同样选择【Cortex-M3 J-LINK】点击[Settings] 

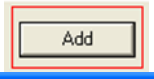


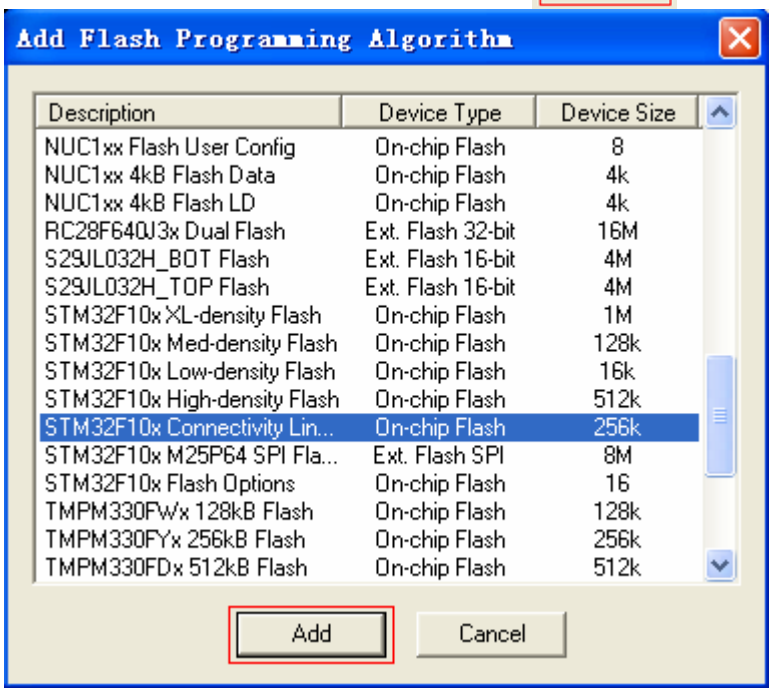
弹出目标板 CPU 加载对话框，点击[Add] 



tyw藏书

选择实际使用的 CPU 系列后，点击[Add]

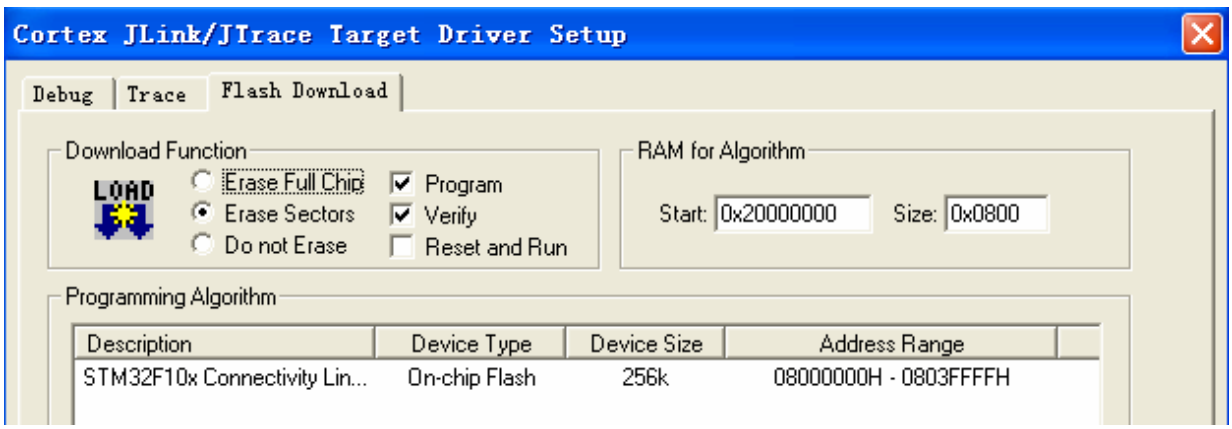




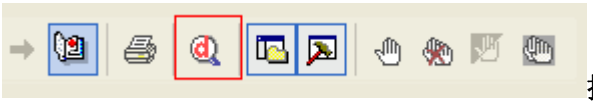
芯片加载成功，

说明：STM32 分小容量，中容量，大容量产品

STM32F103VBT6 为中容量，SSTM32F103VCT6 为大容量产品



进入调试状态

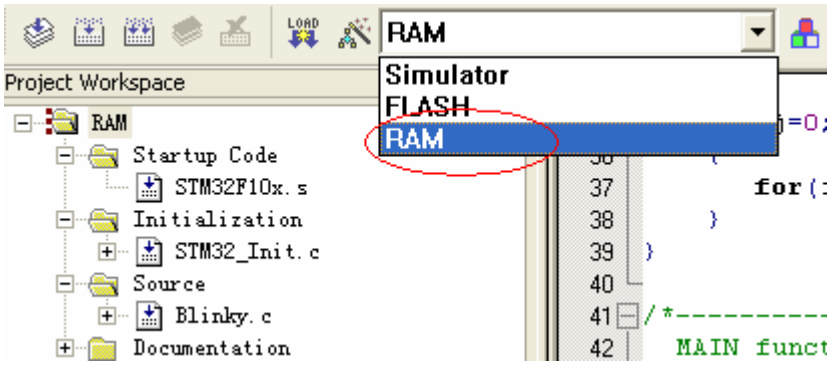


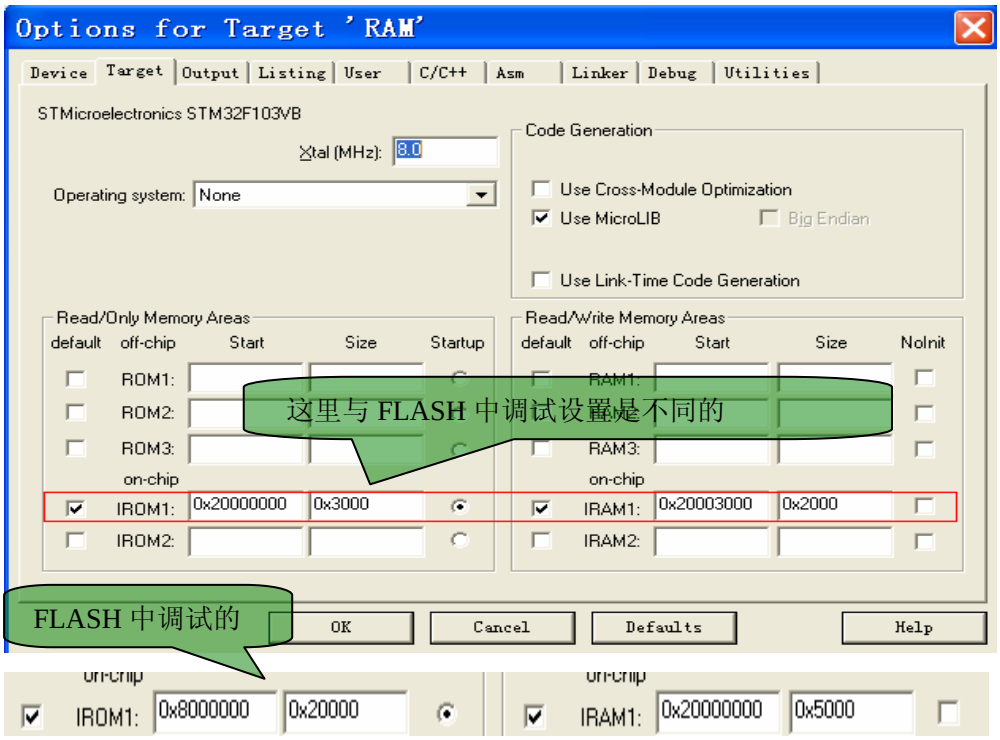
按下调试按钮进入调试状态

tyw藏书

3.5 在RAM中调试程序

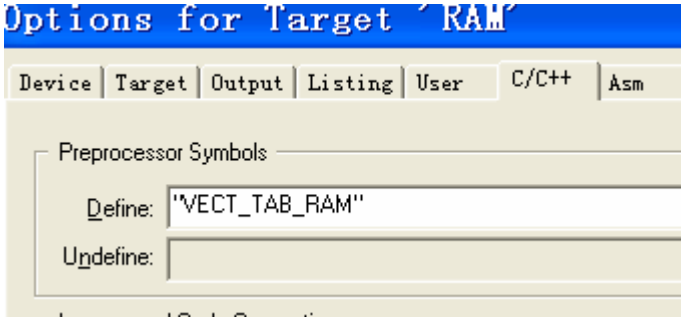
本开发板提供的绝大部分例程都是在 RAM 中调试完成的，在项目也有 RAM 调试模板。





说明：IROM　IRAM 设置根据实际使用芯片 RAM 大小设置，本例是以 20K RAM 的芯片设置的其中 IROM 大小是你的程序容量

编译选项：该项配合代码重新定位中断向量表

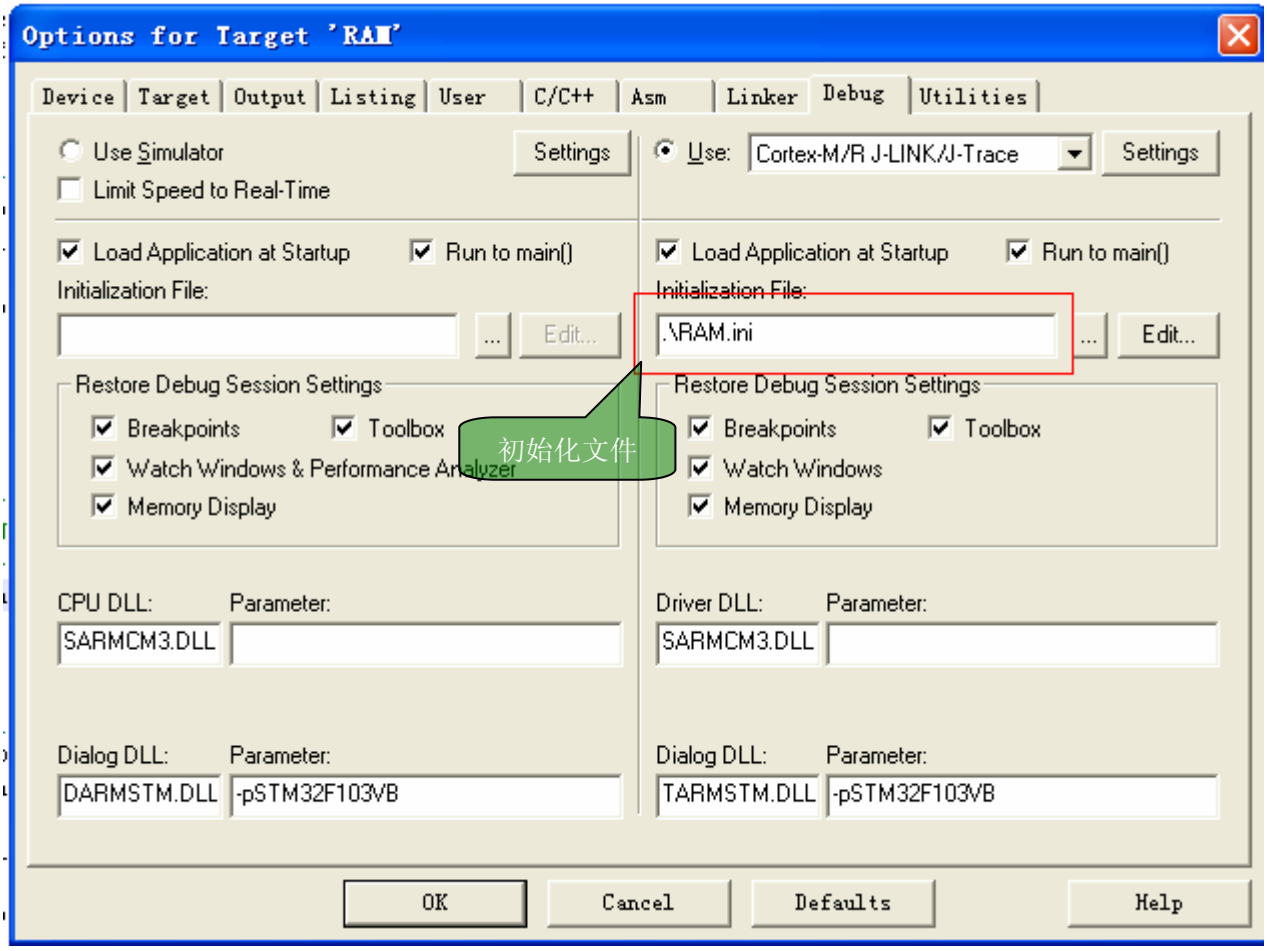


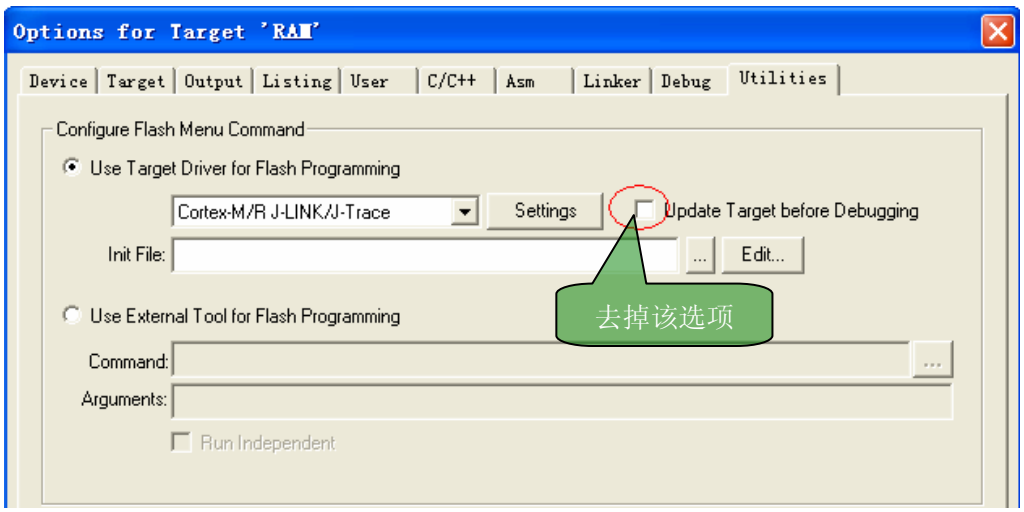
重定位中断向量表代码

```
#if defined (VECT_TAB_RAM)//RAM 中调试
  /* Set the Vector Table base location at 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#elif defined(VECT_TAB_FLASH_IAP)//生成 IAP 文件
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x2000);
#else  /* VECT_TAB_FLASH  */
  /* Set the Vector Table base location at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0); //FLASH 中调试
#endif
```

tyw藏书

载入初始化文件



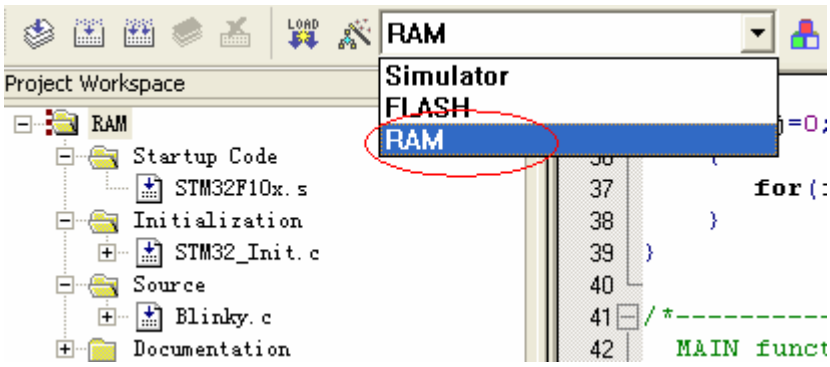


RAM 中调试大功告成，你再用 FLASH 中调试比较下，感觉 RAM 中调试速度是嗖嗖的，相当爽啊！让你都怀疑下载时到底有没完全下载下去。

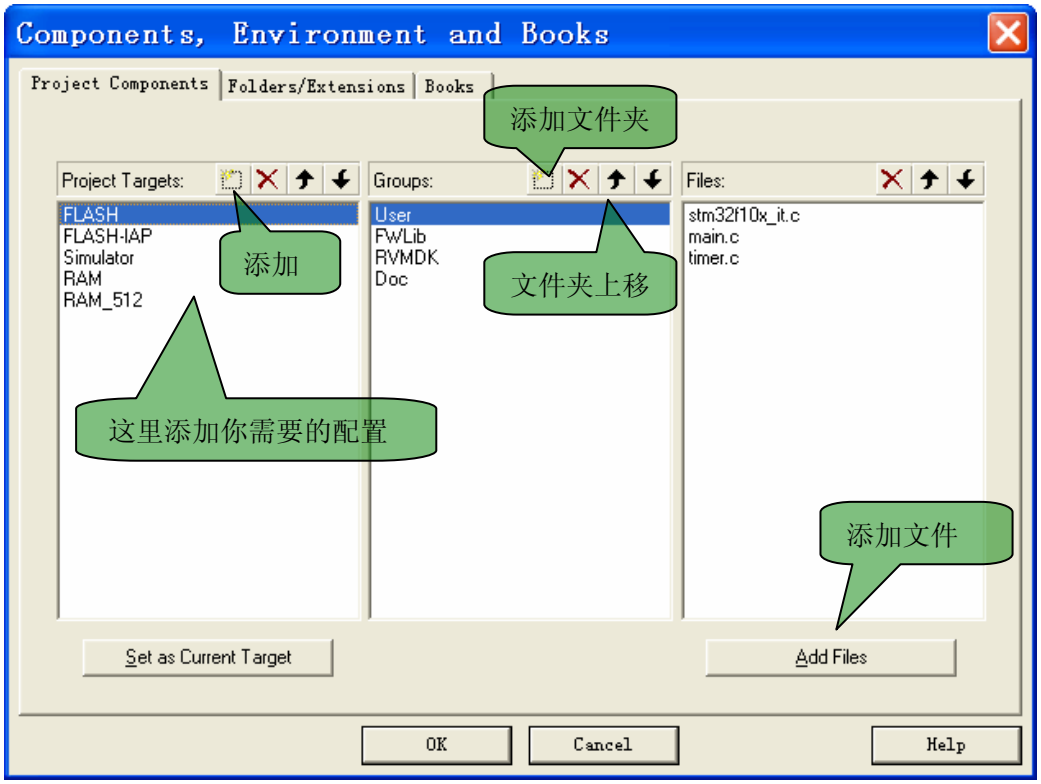
tyw藏书

3.6 项目配置说明

有人说自己新建的项目没有下边窗口选项，实际这是需要根据需求自己添加的，比如我提供的例子中一般包含在 FLASH，RAM 中调试，实际这些都是通过不同配置实现的



点击  ，弹出如下对话框



自己想添加多少个都可以，只要你有那么多特殊需求

3.7 使用JLINK下载程序

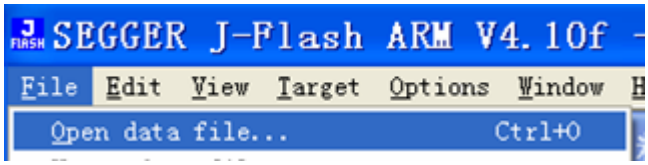
JLINK 官方文档《UM08001_JLinkARM.pdf》下载地址

http://www.segger.com/cms/admin/uploads/productDocs/UM08001_JLinkARM.pdf

JLINK 的软件工具使用可以参考《UM08001_JLinkARM.pdf》，我一般用的不是太多，偶尔用用 JFlash ARM

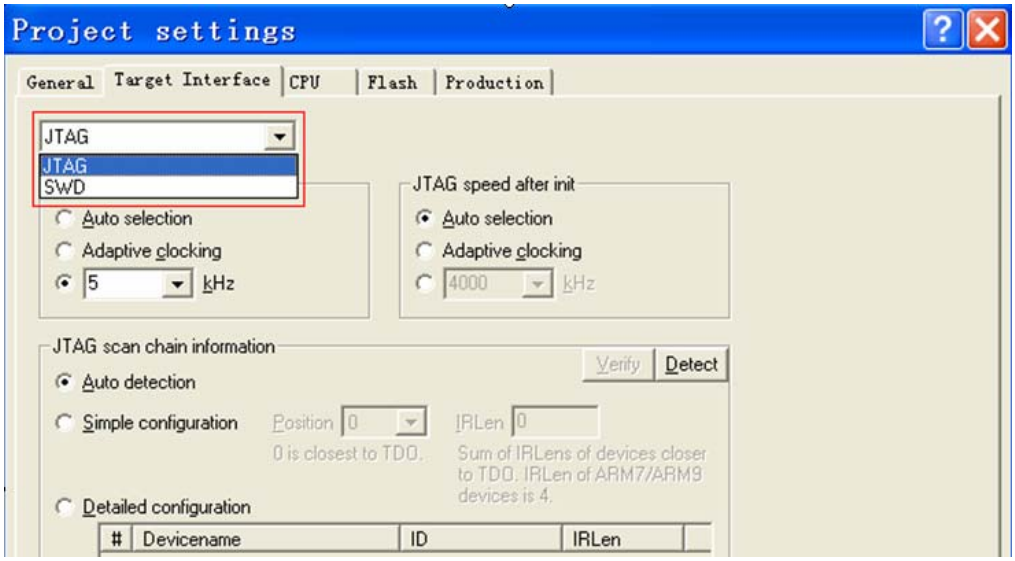
下面简单介绍用 JFlash ARM 编程 FLASH

a. 载入文件：选择菜单 File->Open data file…

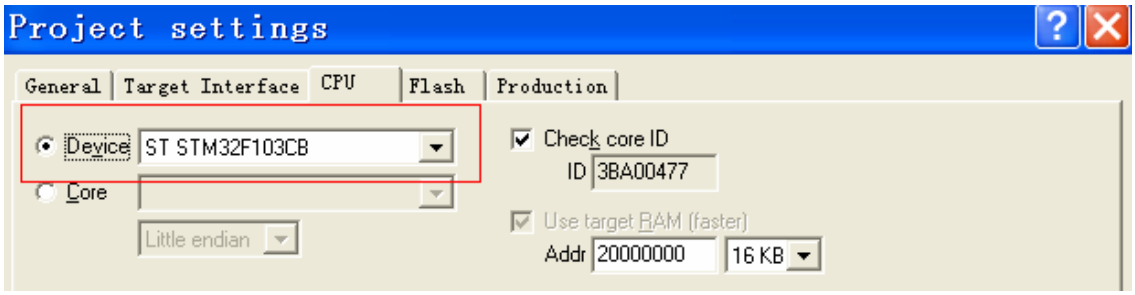


tyw藏书

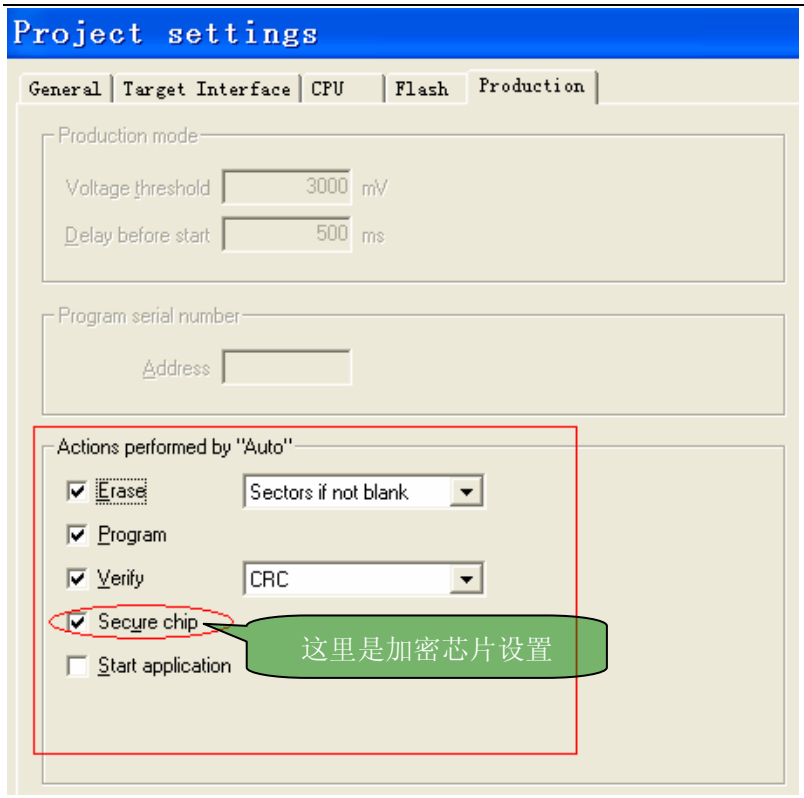
b. 项目设置：选择菜单 Options->Project settings…
[Target Interface]选择 JTAG 或者 SWD 均可，具体根据你的链接方式



c. 选择 CPU，一定要选择对应型号，要不连不上目标板



这里用什么芯片选什么芯片
d. 设置编程选项

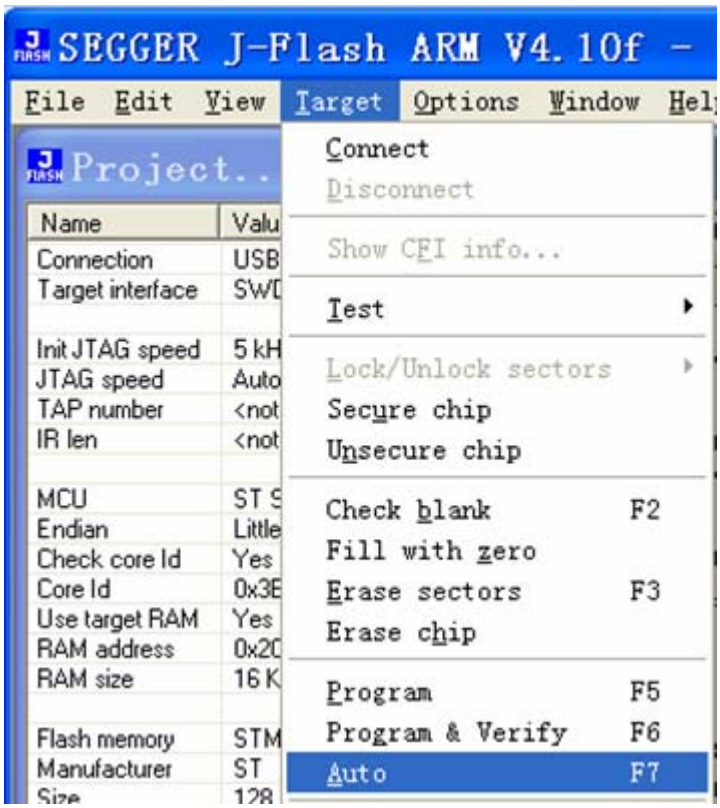


e. 连接目标板：选择菜单 Target->Connect

下面是连接信息

```
Connecting ...
- Connecting via USB to J-Link device 0
- J-Link firmware: V1.20 (J-Link ARM-OB STM32 compiled Dec  3 2009 11:40:54)
- JTAG speed: 5 kHz (Fixed)
- Initializing CPU core (Init sequence) ...
   - Initialized successfully
- JTAG speed: 2000 kHz (Auto)
- Connected successfully
```

f. 编程：选择菜单 Target->Auto



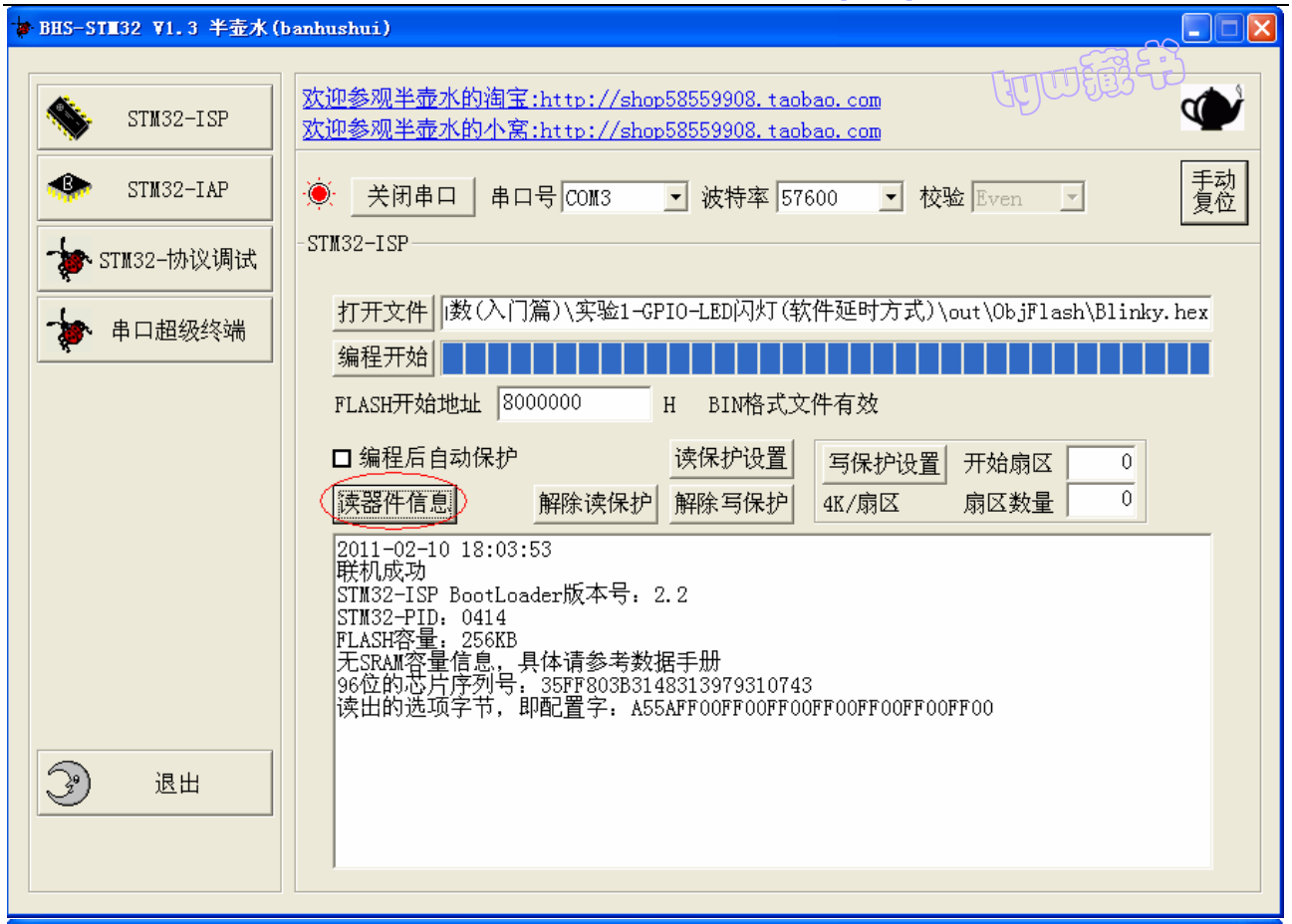
tyw藏书

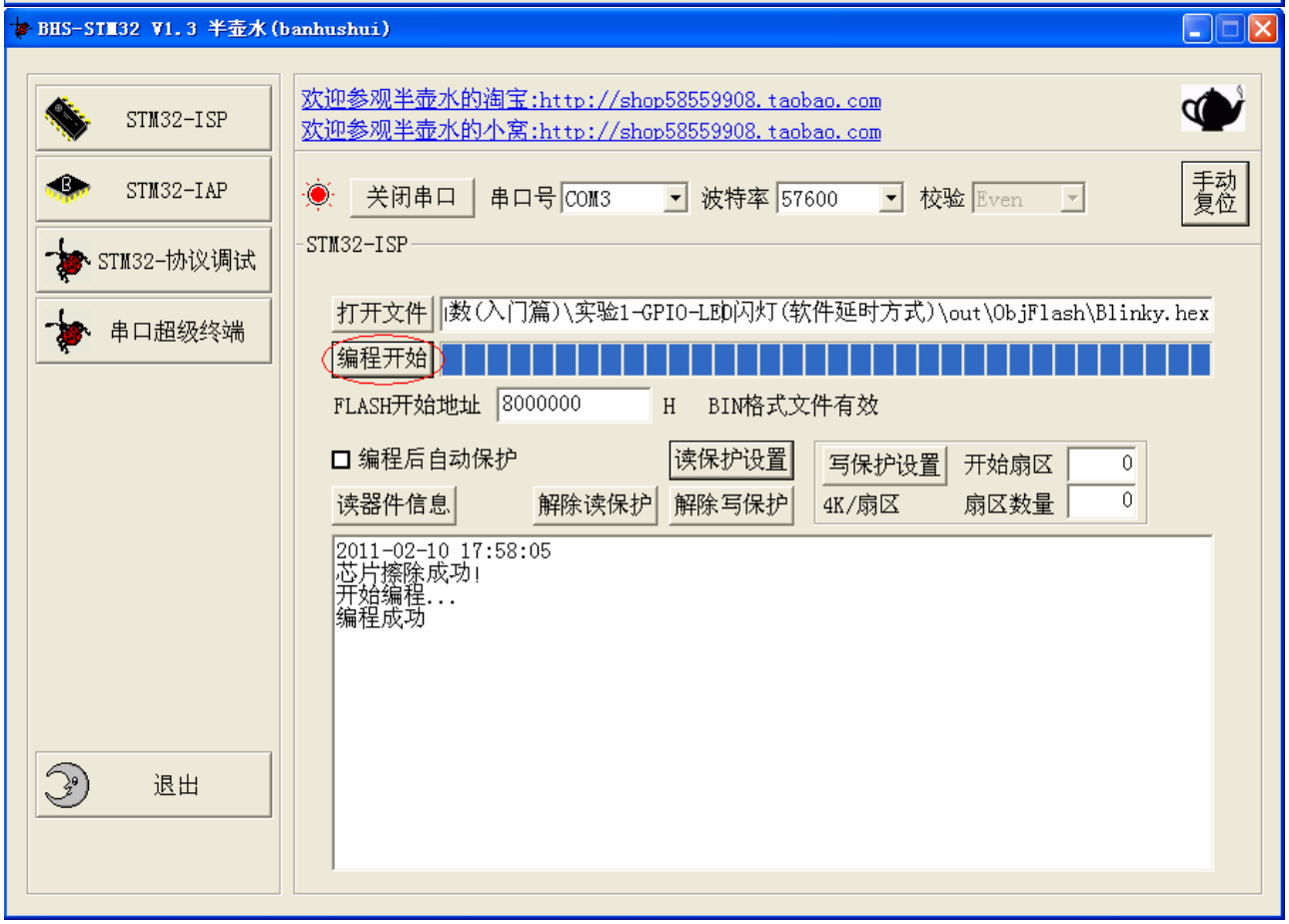
说明：如果在使用中不小心将芯片加密，使用上面的[Unsecure chip]菜单可以恢复。

3.8 ISP直接下载调试

用我配套的 BHS-STM32-ISP-IAP.exe 软件可以实现 ISP 下载运行功能。BHS-STM32-ISP-IAP.exe 使用串口的 DTR,RTS 控制信号自动完成 ISP 下载，无需手动操作。注意:本软件也可以用于其他 STM32 开发板，但是不一定能完成自动 ISP 功能，因为自动 ISP 功能也需要硬件配合。

对于没有自动 ISP 功能硬件的板子，先手动让板子进入 ISP 状态，使用该软件下载程序后重新设置为 FLASH 启动，断电重启。


BHS-STM32 V1.3 半壶水(banhushui)
STM32-ISP
STM32-IAP
STM32-协议调试
串口超级终端
退出
欢迎参观半壶水的淘宝:http://shop58559908.taobao.com
欢迎参观半壶水的小窝:http://shop58559908.taobao.com
关闭串口
串口号 COM3
波特率 57600
校验 Even
手动复位
STM32-ISP
打开文件
数(入门篇)\实验1-GPIO-LED闪灯(软件延时方式)\out\ObjFlash\Blinky.hex
编程开始
FLASH开始地址 8000000 H BIN格式文件有效
编程后自动保护
读保护设置
写保护设置
开始扇区 0
读器件信息
解除读保护
解除写保护
4K/扇区
扇区数量 0
2011-02-10 18:03:53
联机成功
STM32-ISP BootLoader版本号：2.2
STM32-PID：0414
FLASH容量：256KB
无SRAM容量信息，具体请参考数据手册
96位的芯片序列号：35FF803B3148313979310743
读出的选项字节，即配置字：A55AFF00FF00FF00FF00FF00FF00FF00


BHS-STM32 V1.3 半壶水(banhushui)
STM32-ISP
STM32-IAP
STM32-协议调试
串口超级终端
退出
欢迎参观半壶水的淘宝:http://shop58559908.taobao.com
欢迎参观半壶水的小窝:http://shop58559908.taobao.com
关闭串口
串口号 COM3
波特率 57600
校验 Even
手动复位
STM32-ISP
打开文件
数(入门篇)\实验1-GPIO-LED闪灯(软件延时方式)\out\ObjFlash\Blinky.hex
编程开始
FLASH开始地址 8000000 H BIN格式文件有效
编程后自动保护
读保护设置
写保护设置
开始扇区 0
读器件信息
解除读保护
解除写保护
4K/扇区
扇区数量 0
2011-02-10 17:58:05
芯片擦除成功!
开始编程...
编程成功

3.9 IAP直接下载调试

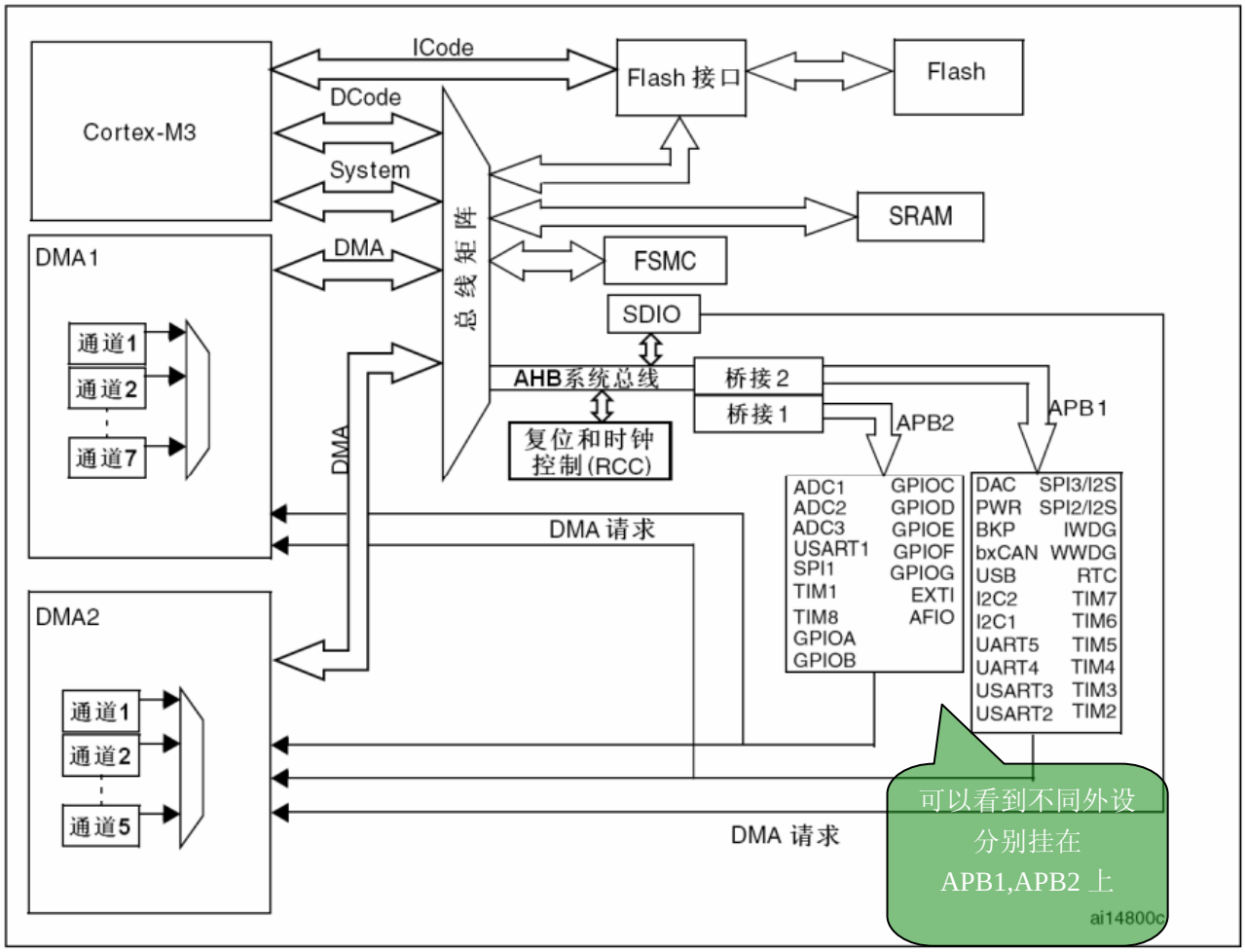
IAP 仍然使用上面的软件，由于 IAP 直接下载调试比较复杂，将单独对该方式做说明，这里略过。

具体参考：高级例程-实验 1- IAP 远程更新用户程序

四、STM32 系统结构

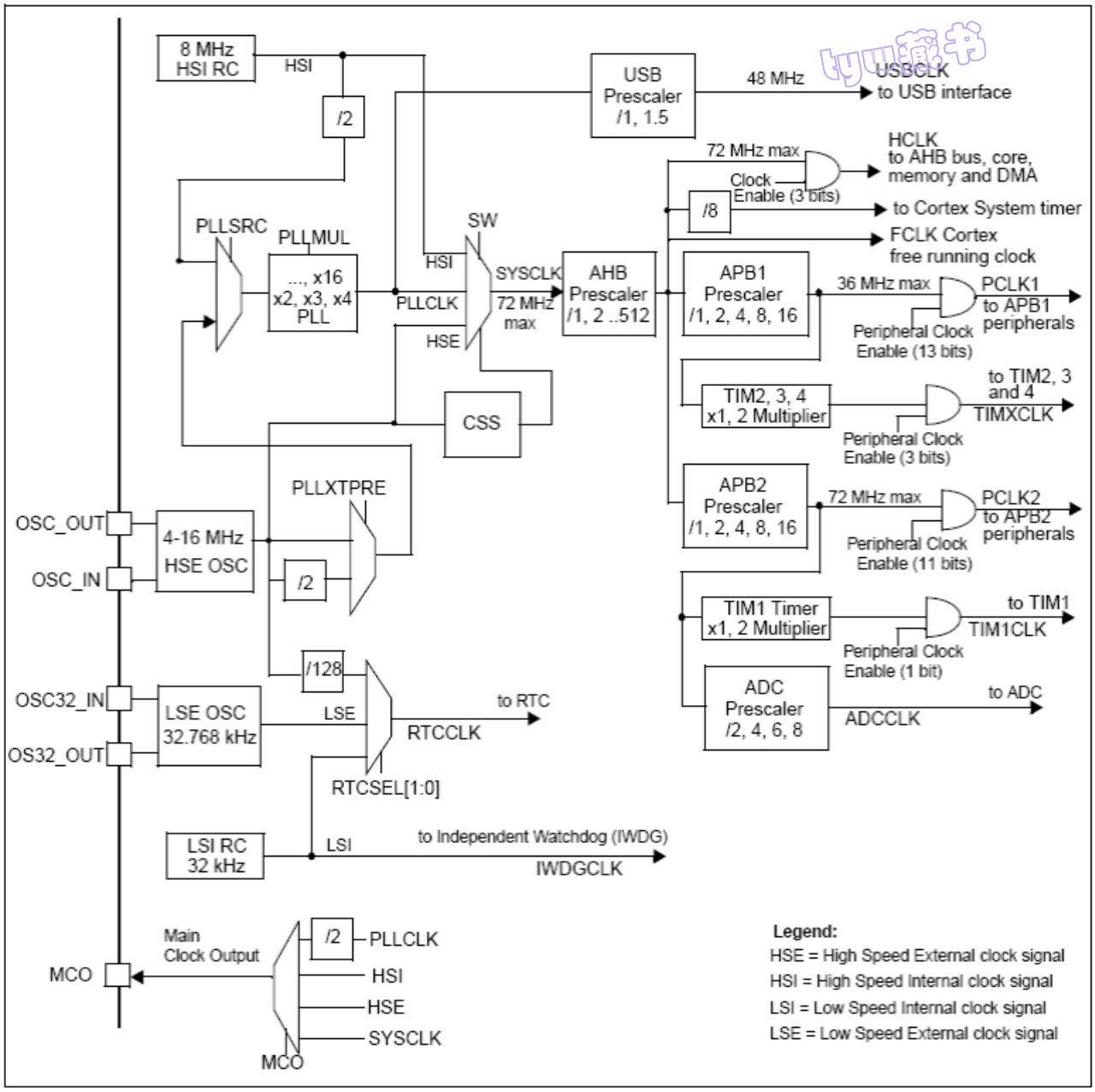
说明：使用 STM32 任何外设都要用到对应的 APB 时钟，只有开启 APB 时钟外设才能工作，很多初学者都忽略了这点，忘记开启外设 APB 时钟导致工作不正常找不到原因。

所以我们先来认识下 STM32 的 APB 时钟



APB1 操作速度限于 36MHz，APB2 操作于全速(最高 72MHz)

STM32 时钟树



特别提醒：STM32 初始化外设第一步是开启 APB 时钟

五、BHS-STM32 例程说明

在做实验之前请先熟悉 STM32 相关硬件，详细请参考《stm32f103 数据手册》《STM32F10x 微控制器参考手册》《Cortex-M3 内核技术参考手册》

基础例程-非库函数(入门篇)

基础例程-非库函数(入门篇)在 \BHS-STM32 例程\基础例程-非库函数(入门篇)文件夹里面，入门例程使用 MDK 配置文件初始化芯片外设。MDK 配置文件说明请参见《UV3.chm》或者前面第三章第 3 小节，所谓非库函数就是直接操作 STM32 相关寄存器，所以先要熟悉 STM32 才行

GPIO实验

实验目的：了解 STM32-GPIO 特性，掌握 GPIO 输入/输出的使用方法。让我们先来了解下 GPIO 介绍。

GPIO功能描述：

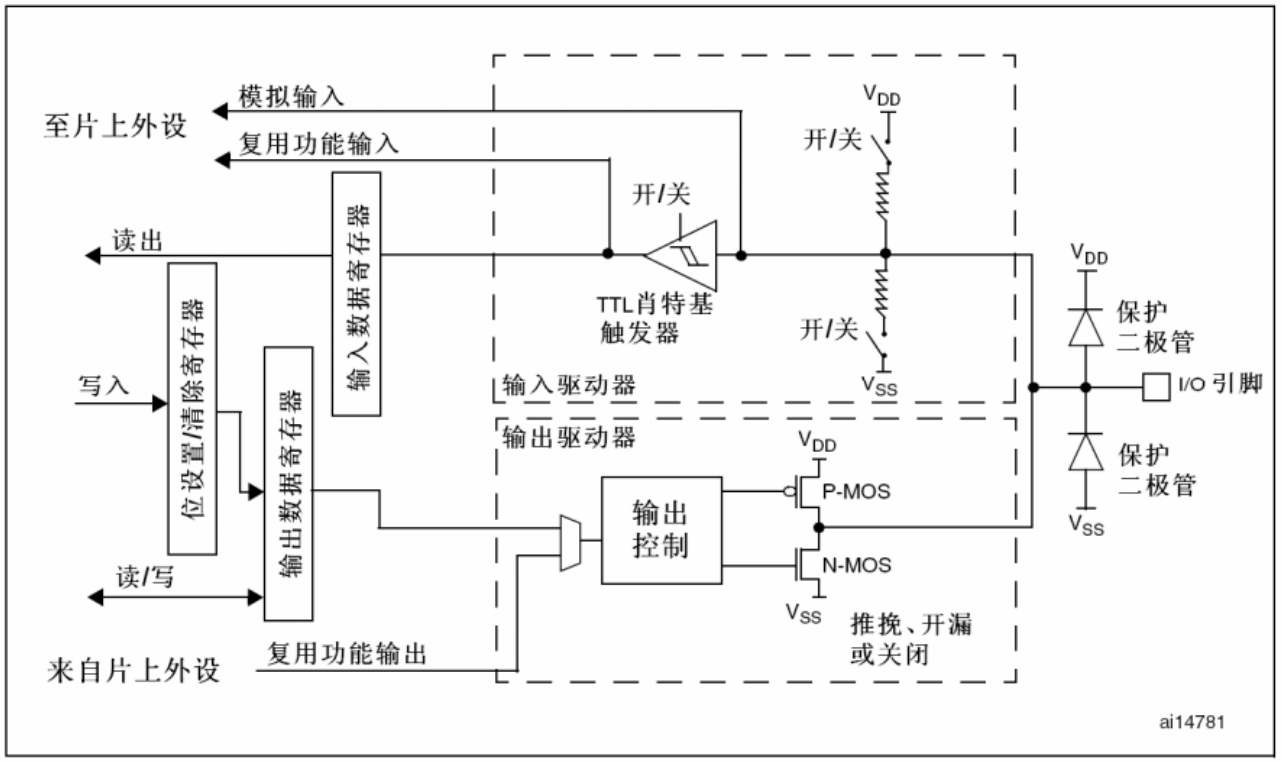
每个 GPI/O 端口有两个 32 位配置寄存器(GPIOx_CRL，GPIOx_CRH)，两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR)，一个 32 位置位/复位寄存器(GPIOx_BSRR)，一个 16 位复位寄存器(GPIOx_BRR)和一个 32 位锁定寄存器(GPIOx_LCKR)。

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

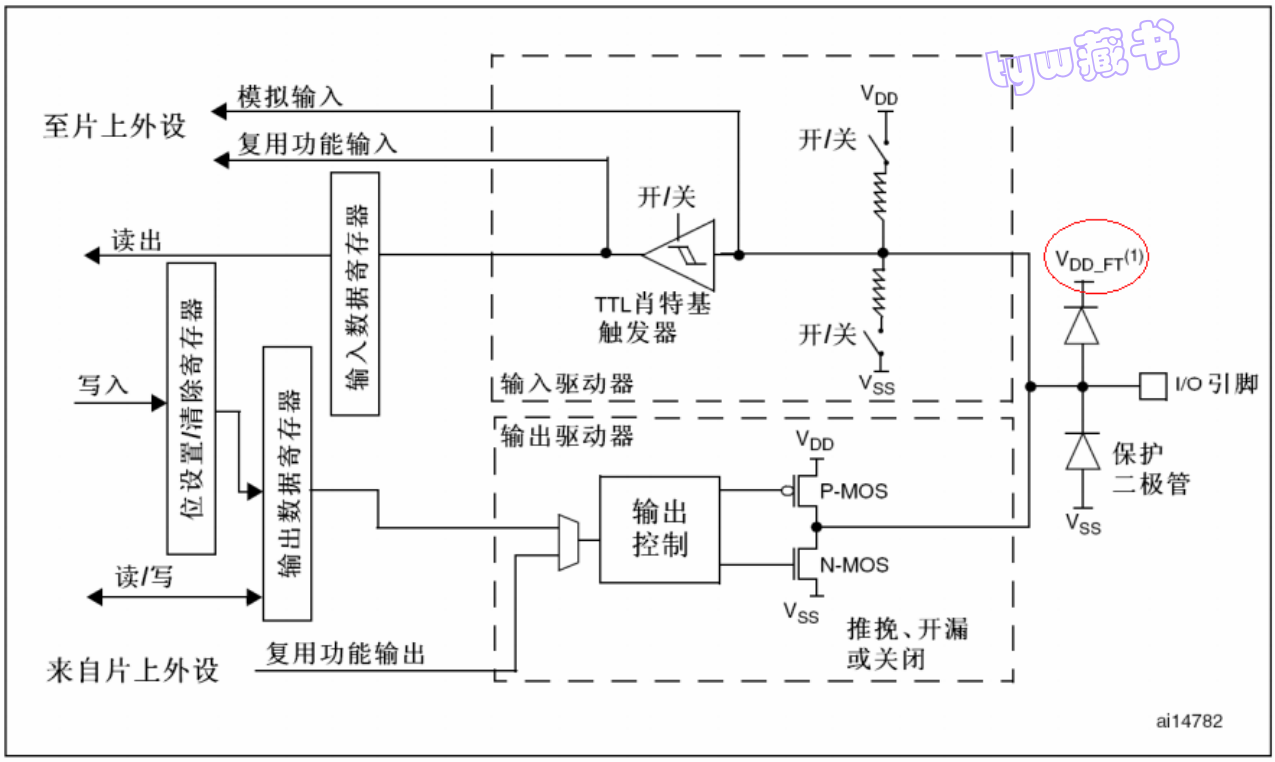
— 输入浮空
— 输入上拉
— 输入下拉
— 模拟输入
— 开漏输出
— 推挽式输出
— 推挽式复用功能
— 开漏复用功能

每个 I/O 端口位可以自由编程，然而 I/0 端口寄存器必须按 32 位字被访问(不允许半字或字节访问)。GPIOx_BSRR 和 GPIOx_BRR 寄存器允许对任何 GPIO 寄存器的读/更改的独立访问；这样，在读和更改访问之间产生 IRQ 时不会发生危险。

下图给出了一个 I/O 端口位的基本结构。



5 伏兼容 I/O 端口位的基本结构，与普通 IO 相比，红圈部分不同的，也就是箝位保护不同



(1) V_{DD_FT} 对5伏容忍I/O脚是特殊的，它与V_{DD}不同

GPIO 端口位配置表

配置模式		CNF1	CNF0	MODE1	MODE0	PxODR 寄存器
通用输出	推挽式(Push-Pull)	0	0	01		0 或 1
	开漏(Open-Drain)		1	10		0 或 1
复用功能输出	推挽(Push-Pull)	1	0	11		不使用
	开漏(Open-Drain)		1	见下表		不使用
输入	模拟输入	0	0	00		不使用
	浮空输入		1			不使用
	下拉输入	1	0			0
	上拉输入					1

输出模式位

MODE[1:0]	意义
00	保留
01	最大输出速度为 10MHz
10	最大输出速度为 2MHz
11	最大输出速度为 50M

特别说明：做为上拉输入时 PxODR 寄存器对应位必须写 1，下拉输入时 PxODR 寄存器对应位必须写 0
上拉输入等于接内部电阻到 VDD，下拉输入等于接内部电阻到 GND，浮空就是什么都不接

注意：每个 I/O 端口位可以自由编程，然而 I/0 端口寄存器必须按 32 位字被访问(不允许半字或字节访问)。GPIOx_BSRR 和 GPIOx_BRR 寄存器允许对任何 GPIO 寄存器的读/更改的独立访问；这样，在读和更改访

问之间产生 IRQ 时不会发生危险。

lyw藏书

每个端口包括以下寄存器，功能对应表

名称	寄存器	意义
端口配置寄存器	GPIOx_CRL GPIOx_CRH	配置 GPIO 工作模式
端口输入数据寄存器	GPIOx_IDR	读取 GPIO 输入状态
端口输出数据寄存器	GPIOx_ODR	控制 GPIO 输出状态
端口位设置/复位寄存器	GPIOx_BSRR	用于位操作 GPIO 的输出状态的：设置端口为 0 或 1
端口位复位寄存器	GPIOx_BRR	用于位操作 GPIO 的输出状态的：设置端口为 0
端口配置锁定寄存器	GPIOx_LCKR	端口锁定后下次系统复位之前将不能再更改端口位的配置

端口配置低寄存器(GPIOx_CRL) (x=A..E) ， 偏移地址：00h，复位值：4444 4444h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

端口配置高寄存器(GPIOx_CRH) (x=A..E)：偏移地址：04h，复位值，4444 4444h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

端口输入数据寄存器(GPIOx_IDR) (x=A..E)，地址偏移：08h，复位值：00000000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	保留，始终读为0。
位15:0	**IDRx[15:0]**：端口输入数据(x = 0…15) 这些位为只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。

端口输出数据寄存器(GPIOx_ODR) (x=A..E)，地址偏移：0Ch，复位值：00000000h

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

端口位设置/复位寄存器(GPIOx_BSRR) (x=A..E) ， 地址偏移：10h ， 复位值：00000000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	**BRx**: 清除位x (x = 0…15) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRx位不产生影响 1：清除对应的ODRx位为0 **注**：如果同时设置了BSx和BRx的对应位，BSx位起作用。
位15:0	**BRx**: 设置位x (x = 0…15) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRx位不产生影响 1：设置对应的ODRx位为1

端口位复位寄存器(GPIOx_BRR) (x=A..E) ， 地址偏移：14h ， 复位值：00000000h

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	保留。
位15:0	**BRx**: 清除位x (x = 0…15) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRx位不产生影响 1：清除对应的ODRx位为0 **注**：如果同时设置了BSx和BRx的对应位，BSx位起作用。

端口配置锁定寄存器(GPIOx_LCKR) (x=A..E) 地址偏移：18h 复位值：0000 0000h

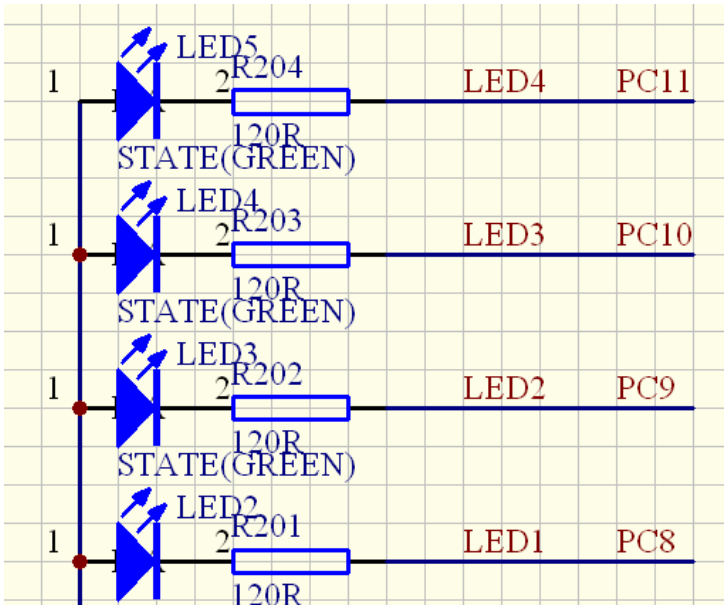
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															LCKK
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:17	保留。
位16	**LCKK[15:0]**：锁键 该位可随时读出，它只可通过锁键写入序列修改。 0：端口配置锁键位激活 1：端口配置锁键位被激活，下次系统复位前GPIOx_LCKR寄存器被锁住。 **锁键的写入序列**： 写1 -> 写0 -> 写1 -> 读0 -> 读1 最后一个读可省略，但可以用来确认锁键已被激活。 **注**：在操作锁键的写入序列时，不能改变LCK[15:0]的值。 操作锁键写入序列中的任何错误将不能激活锁键。
位15:0	**LCKx**: 锁位x (x = 0…15) 这些位可读可写但只能在LCKK位为0时写入。 0：不锁定端口的配置 1：锁定端口的配置

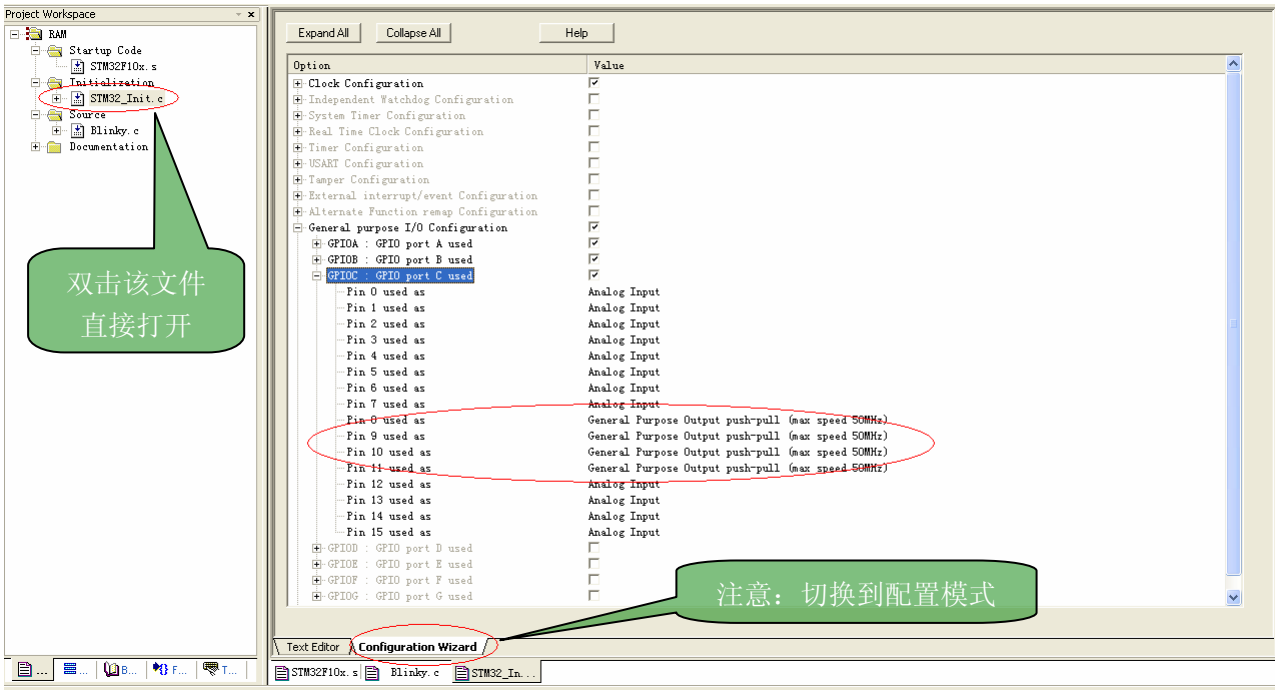
BHS-STM32 实验 1-GPIO输出-LED闪灯(软件延时方式)(直接操作寄存器)

使用的硬件

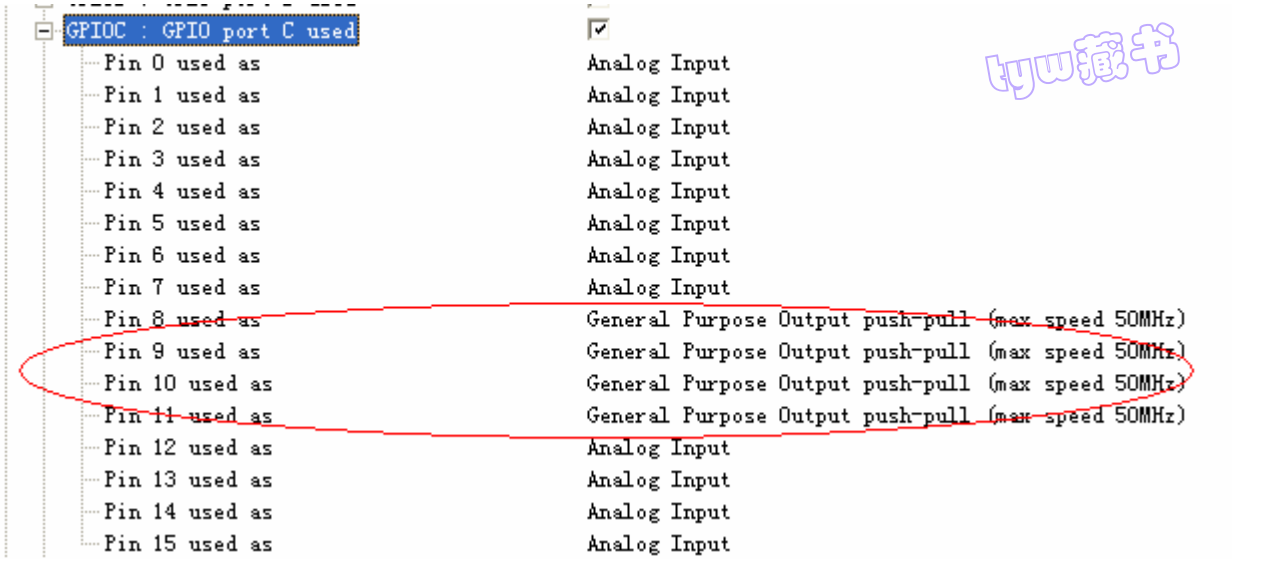


tyw藏书

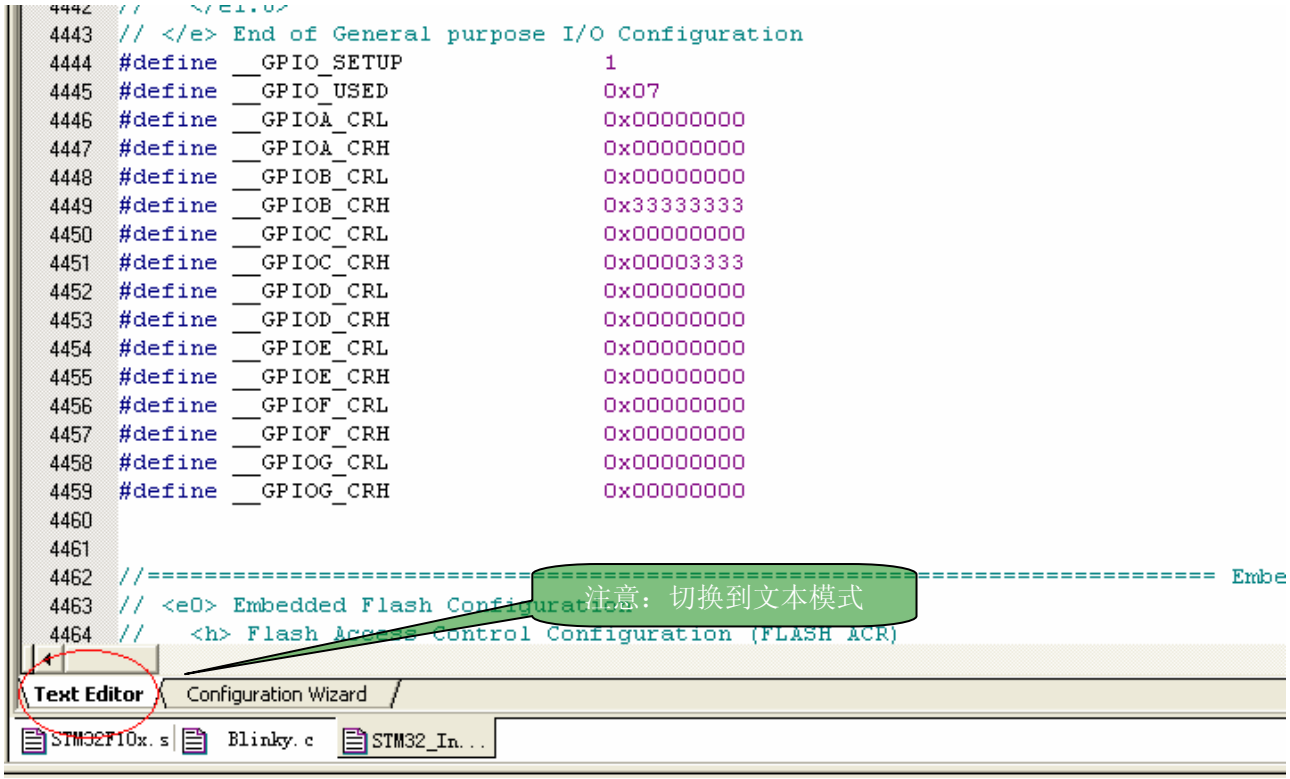
本例子是 GPIO 输出实验，通过 LED 观察端口变化，使用 PC8~PC11 端口，通过软件延时改变端口状态
所以配置 PC8~PC11 为推挽输出，直接驱动 LED
运行程序能看到 LED 循环闪烁
打开 STM32_Init.c，看到配置文件对话框，我们就可以选择需要的 GPIO 模式了



下面提供局部清晰图片



配置实际是定义了些常数，该常数就是 GPIO 寄存器的值



在这个文件里我们可以找到 GPIO 的初始化函数

```
4940  __inline static void stm32_GpioSetup (void) {
4941
4942    if (__GPIO_USED & 0x01) {                       // GPIO Port A used
4943      RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;           // enable clock for GPIOA
4944      GPIOA->CRL = __GPIOA_CRL;                     // set Port configuration register low
4945      GPIOA->CRH = __GPIOA_CRH;                     // set Port configuration register high
4946    }
4947
4948    if (__GPIO_USED & 0x02) {                       // GPIO Port B used
4949      RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;           // enable clock for GPIOB
4950      GPIOB->CRL = __GPIOB_CRL;                     // set Port configuration register low
4951      GPIOB->CRH = __GPIOB_CRH;                     // set Port configuration register high
4952    }
4953
4954    if (__GPIO_USED & 0x04) {                       // GPIO Port C used
4955      RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;           // enable clock for GPIOC
4956      GPIOC->CRL = __GPIOC_CRL;                     // set Port configuration register low
4957      GPIOC->CRH = __GPIOC_CRH;                     // set Port configuration register high
4958    }
4959
4960    if (__GPIO_USED & 0x08) {                       // GPIO Port D used
4961      RCC->APB2ENR |= RCC_APB2ENR_IOPDEN;           // enable clock for GPIOD
4962      GPIOD->CRL = __GPIOD_CRL;                     // set Port configuration register low
4963      GPIOD->CRH = __GPIOD_CRH;                     // set Port configuration register high
4964    }
4965
4966    if (__GPIO_USED & 0x10) {                       // GPIO Port E used
4967      RCC->APB2ENR |= RCC_APB2ENR_IOPEEN;           // enable clock for GPIOE
4968      GPIOE->CRL = __GPIOE_CRL;                     // set Port configuration register low
4969      GPIOE->CRH = __GPIOE_CRH;                     // set Port configuration register high
4970    }
```

Text Editor | Configuration Wizard

STM32F10x.s | Blinky.c | STM32_In...

下面我来解释下使用到的 PC 端口

#define __GPIOC_CRL　　0x00000000

#define __GPIOC_CRH　　0x00003333　　//配置 PC8~PC11 为推挽输出，最大速度 50MHz

实际就是：

```
Pin 7 used as                 Analog Input
Pin 8 used as                 General Purpose Output push-pull (max speed 50MHz)
Pin 9 used as                 General Purpose Output push-pull (max speed 50MHz)
Pin 10 used as                General Purpose Output push-pull (max speed 50MHz)
Pin 11 used as                General Purpose Output push-pull (max speed 50MHz)
Pin 12 used as                Analog Input
```

应该一下就明白了红框的提示信息就是：通用推挽输出，最大输出速度为 50MHz

KEIL 的配置向导是不是对我们了解寄存器的设置非常有帮助呢。

```
if (__GPIO_USED & 0x04) {                       // GPIO Port C used
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;         // enable clock for GPIOC，使能 GPIOC 的时钟
    //特别提醒：STM32 初始化外设第一步是开启 APB 时钟，根据前面的结构图知道 GPIOC 是挂在 APB2
    //下面的
    GPIOC->CRL = __GPIOC_CRL;       // set Port configuration register low 设置端口配置低寄存器
    GPIOC->CRH = __GPIOC_CRH;       // set Port configuration register high 设置端口配置高寄存器
  }
```

程序比较简单：

```
//LED 循环闪烁
void LedFlash(void)
{
  static u16 leds = 0x01;
  u32 temp;
```

tyw藏书

```
    //先读出 PC 端口状态
    temp = GPIOC->ODR;
    //先屏蔽掉 PC8~PC11
    temp |= 0x00000F00;
    //重新设置 PC8~PC11 输出状态，IO 输出低电平点亮 LED
    GPIOC->ODR = temp&(~(leds<<8));
    leds <<= 1;
    if ( (leds&0x0f) == 0)
        leds = 0x01;
}
//下面是主函数里面循环点亮 LED
while (1)
  {
      Delay(50);
      //循环显示 1 位 LED
      LedFlash();
      Delay(50);
      //关闭所有 LED
      GPIOC->ODR |= 0x00000F00;
  }
//LED 是从 PC8 开始的。高 16 位目前 STM32 是保留不用的。
```

不少人问 GPIOC->CRL 是啥意思？

其实这是 C 语言基础问题，->运算符实际是指针的运算符。

在<stm32f10x_map.h>中定义如下

```
typedef struct
{
  vu32 CRL;
  vu32 CRH;
  vu32 IDR;
  vu32 ODR;
  vu32 BSRR;
  vu32 BRR;
  vu32 LCKR;
} GPIO_TypeDef;
//根据名字可知道上面实际就是对应的 GPIO 端口的几个寄存器
#define PERIPH_BASE                ((u32)0x40000000)
/* Peripheral memory map */
#define APB1PERIPH_BASE            PERIPH_BASE
#define APB2PERIPH_BASE            (PERIPH_BASE + 0x10000)
#define GPIOC_BASE                 (APB2PERIPH_BASE + 0x1000)
#define GPIOC                      ((GPIO_TypeDef *) GPIOC_BASE)
```

0x4001 1800 - 0x4001 1BFF	GPIO端口E
0x4001 1400 - 0x4001 17FF	GPIO端口D
0x4001 1000 - 0x4001 13FF	GPIO端口C
0X4001 0C00 - 0x4001 0FFF	GPIO端口B
0x4001 0800 - 0x4001 0BFF	GPIO端口A

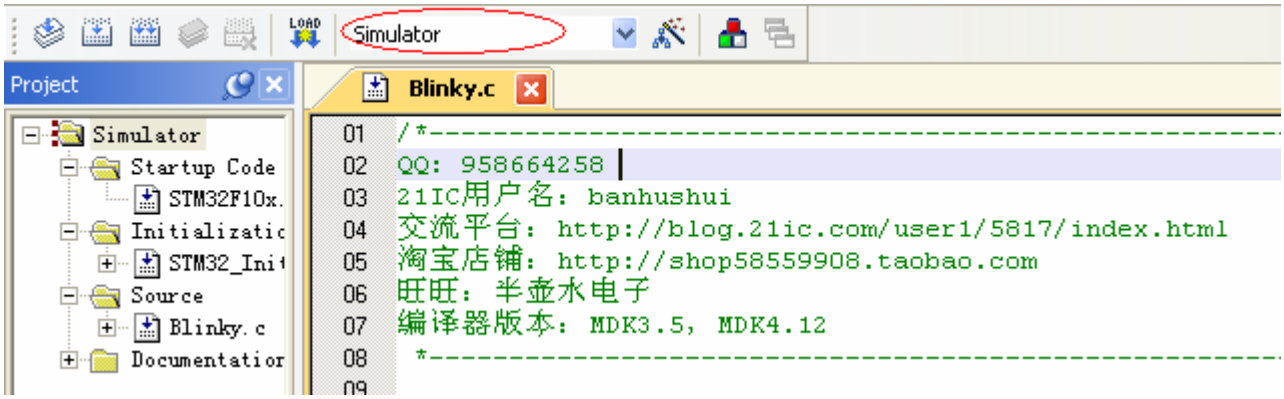
tyw藏书

GPIOC_BASE=0x40011000

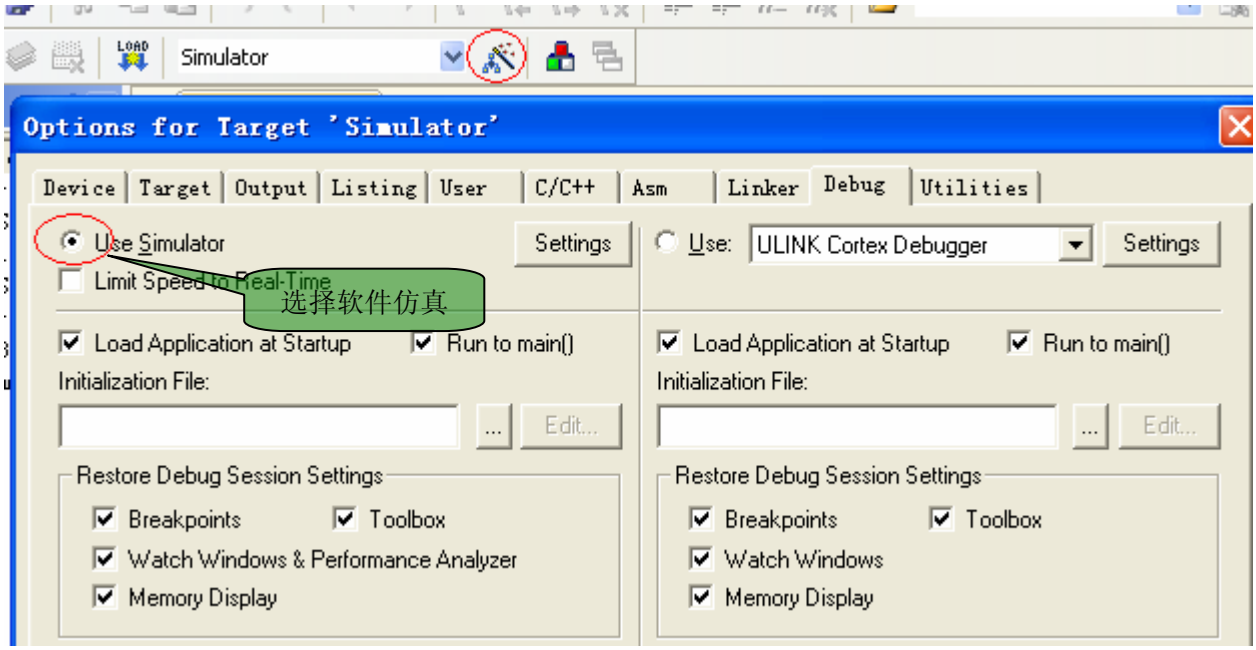
根据上面的寄存器组起始地址表可以知道，GPIOC 寄存器是从 0x40011000 开始的

软件仿真：

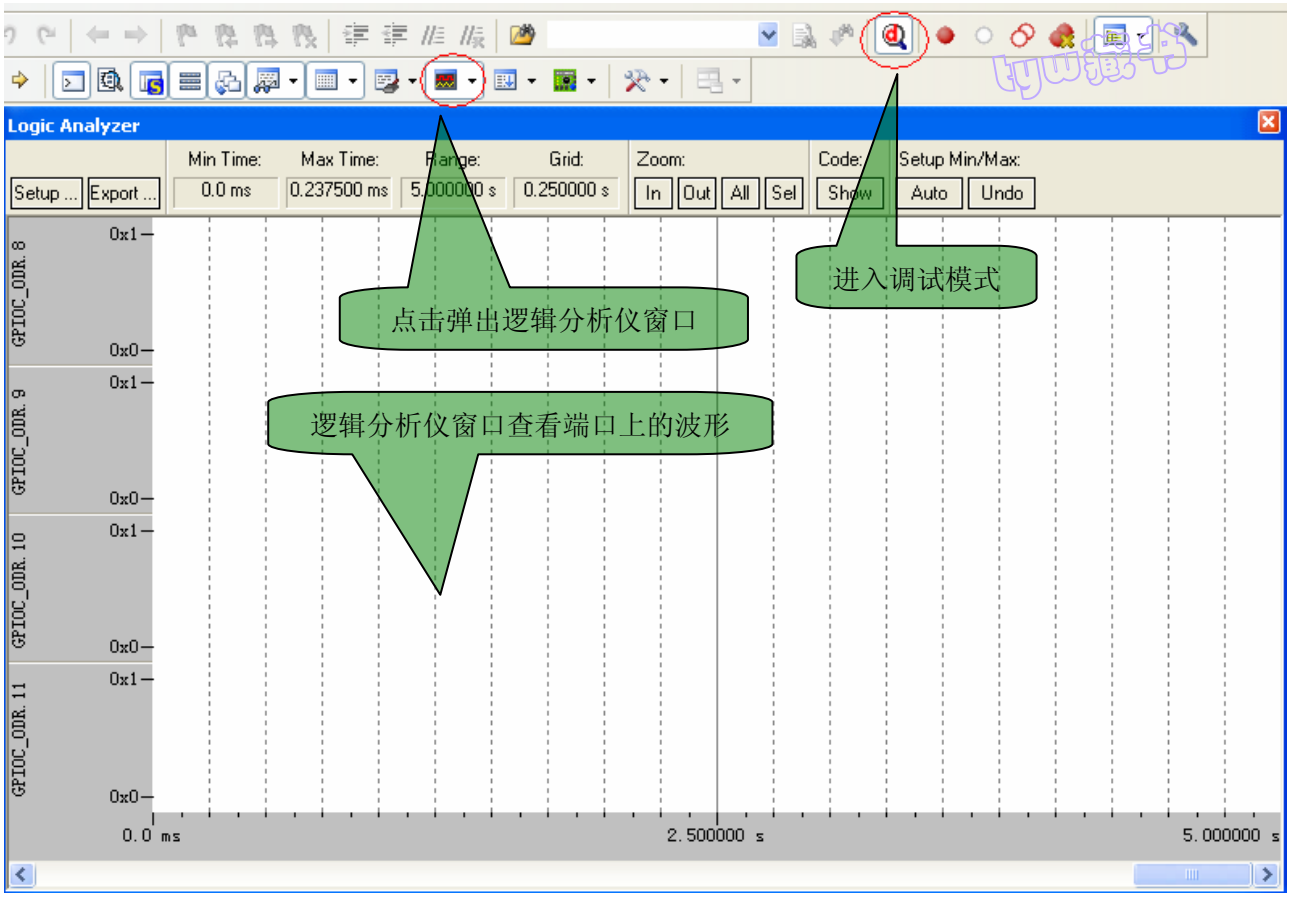
选择软件仿真



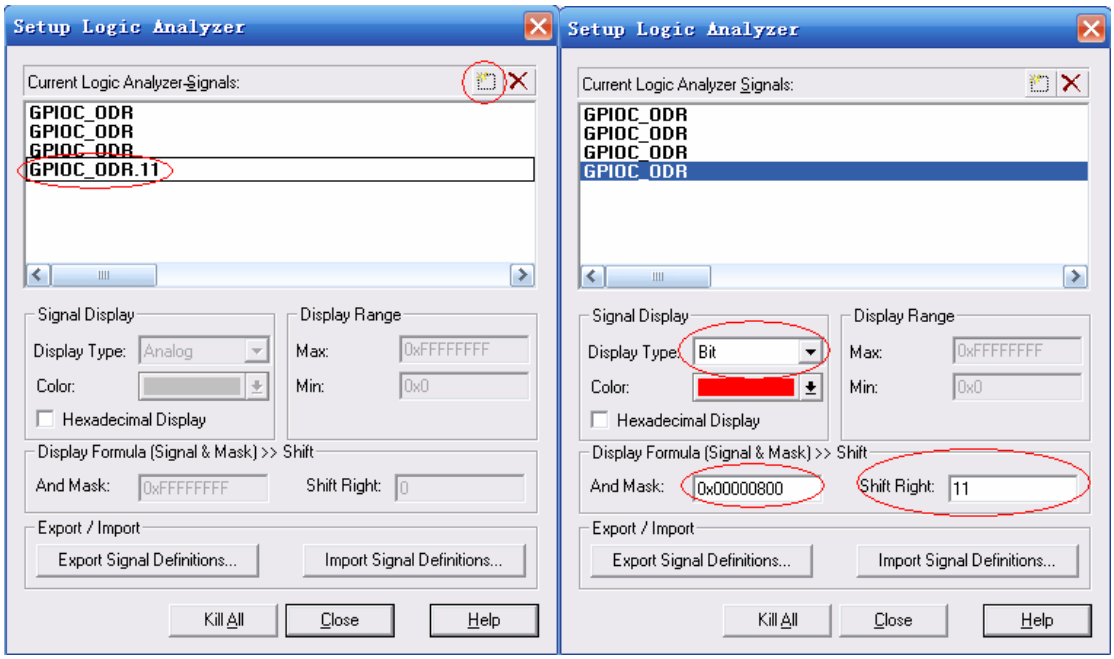
点击配置选项弹出下面的对话框



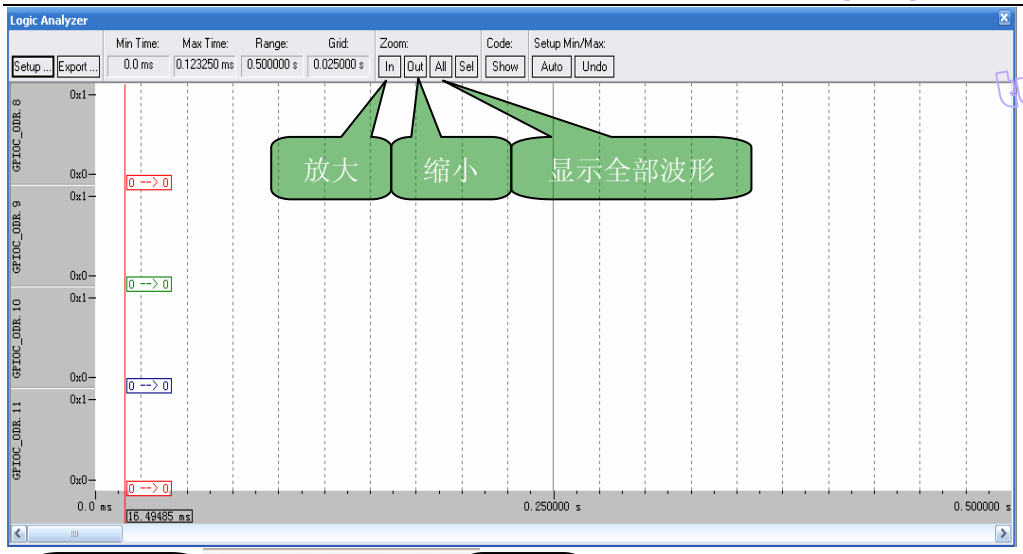
设置好后进入调试状态弹出下面对话框

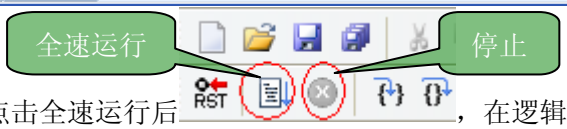


下面添加要查看的端口

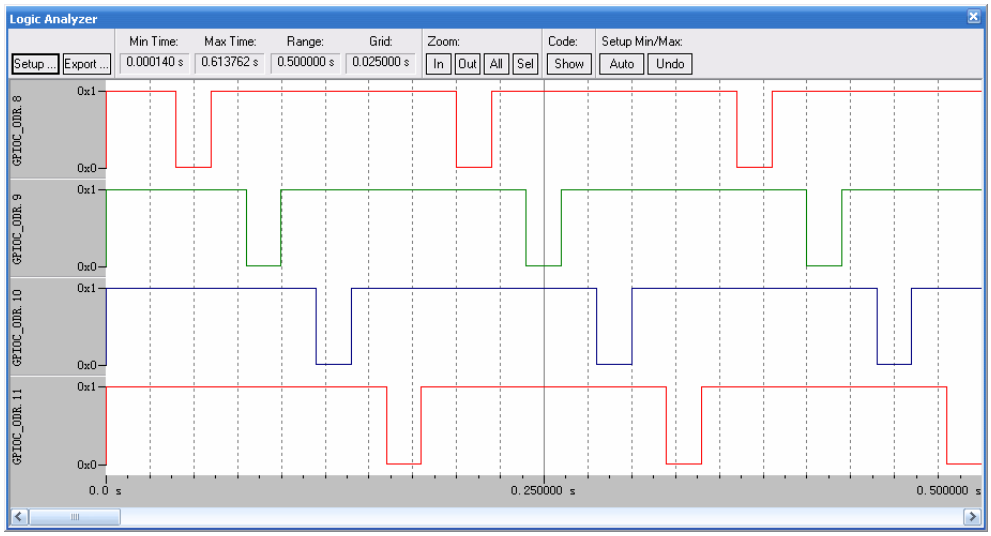


说明：【And Mask】 屏蔽掉不显示的位，【Shift Right】 将显示数据右移到最底位显示

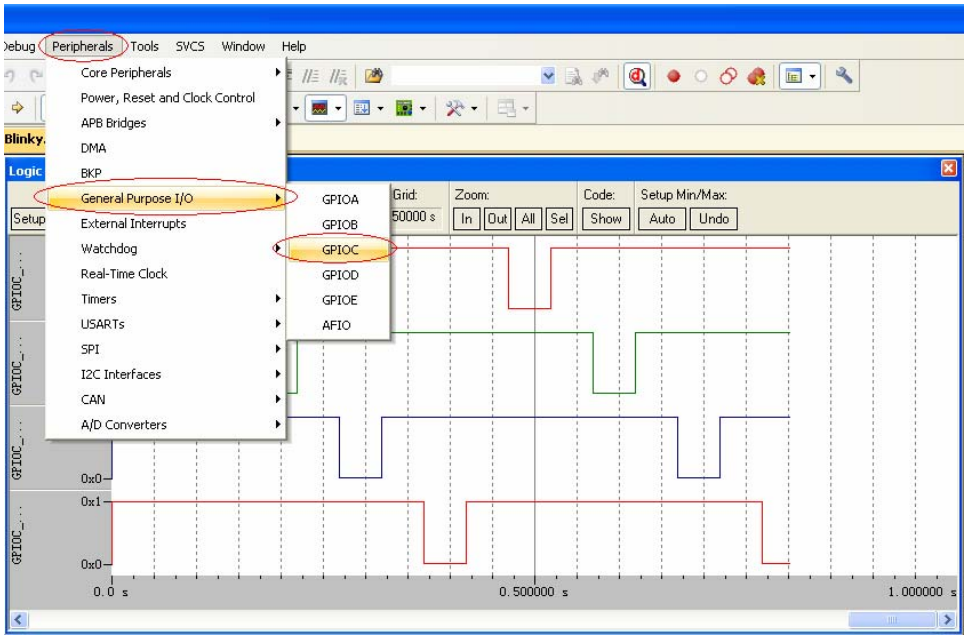




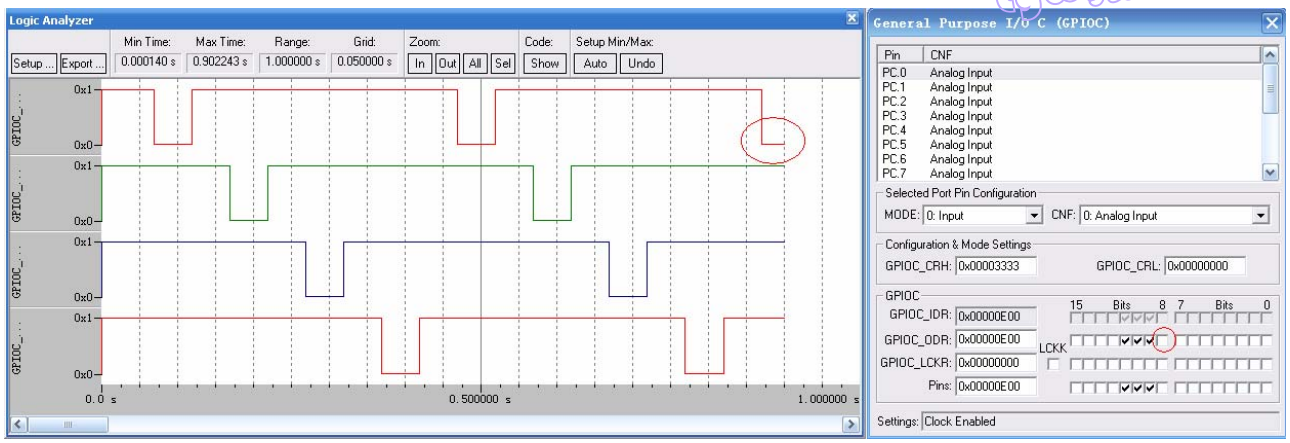
点击全速运行后，在逻辑分析仪里就能看到 PC8~PC11 的波形



再打开 GPIOC 窗口，看逻辑分析仪是否和 GPIOC 窗口同步。



GPIOC 窗口的 GPIOC_ODR 对应的位勾上表示端口是 1，没勾上表示端口是 0，运行可以看到勾勾在不停变化，因为是低电平点亮 LED，下图中 2 个红圈在同一时间都显示 GPIOC_ODR.8 是低电平。



从上面的波形，和勾勾变化可以看出跟我们在硬件上直观看到 LED 循环亮灭效果是一样的。

BHS-STM32 实验 2 STM32F10x库编译

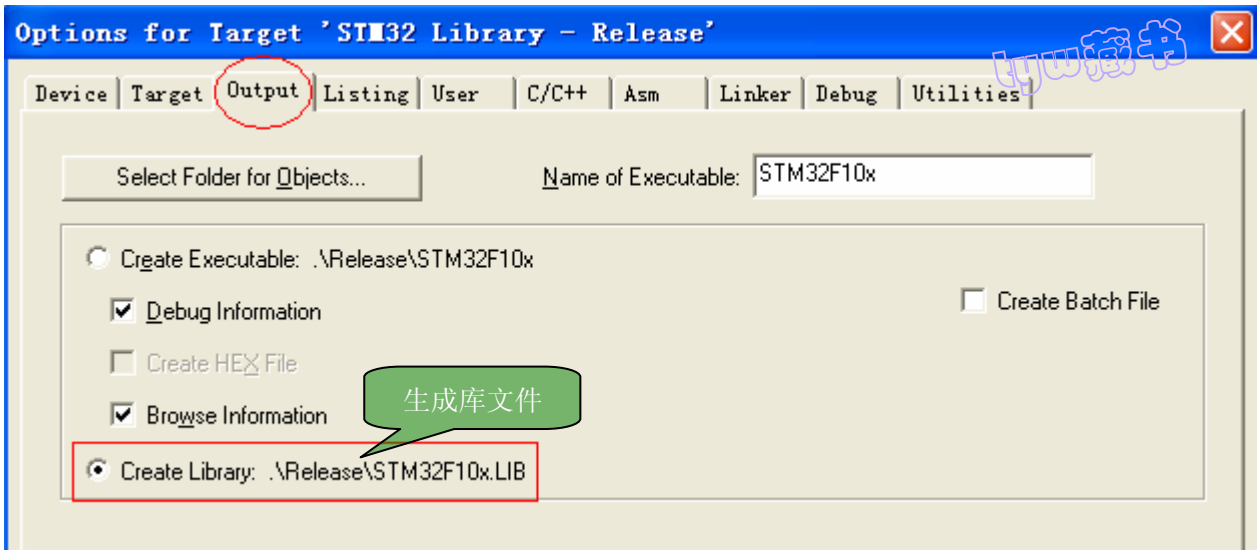
我们使用库的目的：

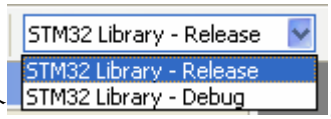
1. 开发变得简单，不用看一堆源文件。
2. 减少项目编译时间，库文件是已经编译好的，对于大型项目非常有用。
3. 防止不小心修改了源文件，文件太大，不小心文件就被修改了，生成库文件后无法修改。
4. 对于团队合作开发的项目，其他项目成员无需看到源文件可以提供库文件开发。

本实验是为后面用库函数的例子做铺垫，ST 库函数是 ST 官方为了方便用户快速应用 STM32 推出的底层函数库，该函数可以在不熟悉 STM32 芯片的情况下，通过阅读库函数手册，和参考例程快速应用 STM32. STM32 库函数在 MDK 安装目录下：

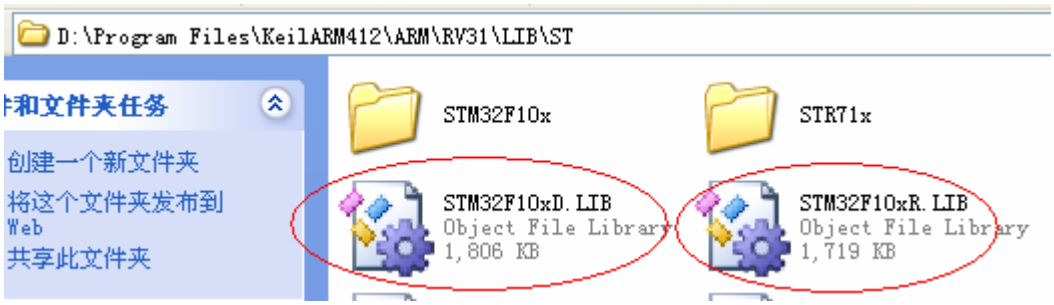
那么这个库怎么来的呢，实际上 STM32F10x 文件夹就是库的源文件，本例将改文件夹中所有文件复制过来

配置选项里配置如下即可产生 LIB 文件



细心的人一定能发现编译选项有 2 个 

同时在库文件路径下有两个库，分别是



实际上 STM32F10xD.LIB 是调试库，STM32F10xR.LIB 是发行库，两者有什么区别呢？
STM32F10xD.LIB 带了调试信息输出的，我们来看看库源文件
在 ST 官方的 STM32 的库函数里有很多 assert_param 函数
比如下面的
assert_param(IS_ADC_ALL_PERIPH(ADCx));
assert_param(IS_ADC_IT(ADC_IT));

assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
几乎是带参数的函数前面都有调用 assert_param
实际这是个调试函数，当你在调试程序时打开 DEBUG 参数 assert_param 才起作用。
assert_param 是反映参数你在调用库函数传递的参数是错误的。
assert_param 的原型定义在 stm32f10x_conf.h 文件里
定义如下：

```
/* Exported macro ------------------------------------------------------------*/
#ifdef  DEBUG
/*******************************************************************************
* Macro Name     : assert_param
```

```
* Description       : The assert_param macro is used for function's parameters check.
*                     It is used only if the library is compiled in DEBUG mode.
* Input             : - expr: If expr is false, it calls assert_failed function
*                       which reports the name of the source file and the source
*                       line number of the call that failed.
*                       If expr is true, it returns no value.
* Return            : None
*******************************************************************************/
  #define assert_param(expr) ((expr) ? (void)0 : assert_failed((u8 *)__FILE__, __LINE__))
/* Exported functions ------------------------------------------------------- */
  void assert_failed(u8* file, u32 line);
#else
  #define assert_param(expr) ((void)0)
#endif /* DEBUG */
#endif /* __STM32F10x_CONF_H */
```

可以看到 assert_param 实际在 DEBUG 打开时就是 assert_failed，关闭 DEBUG 时是空函数

assert_failed 函数如下

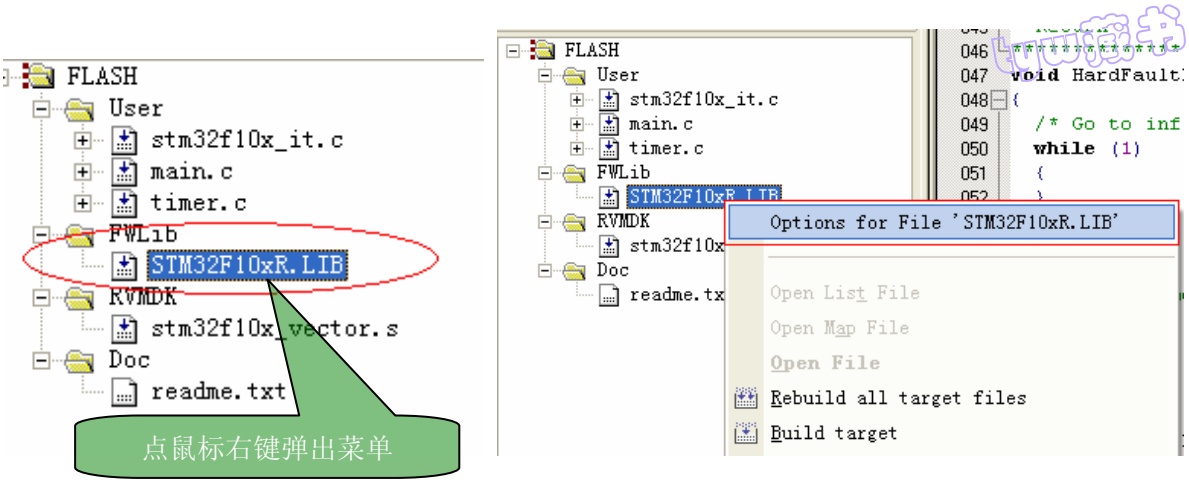
```
#ifdef  DEBUG
/*******************************************************************************
* Function Name  : assert_failed
* Description    : Reports the name of the source file and the source line number
*                  where the assert_param error has occurred.
* Input          : - file: pointer to the source file name
*                  - line: assert_param error line source number
* Output         : None
* Return         : None
*******************************************************************************/
void assert_failed(u8* file, u32 line)
{
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  //用户可以在这里添加错误信息:比如打印出出错的文件名和行号
  /* Infinite loop */
  while (1)
  {
  }
}
#endif
```

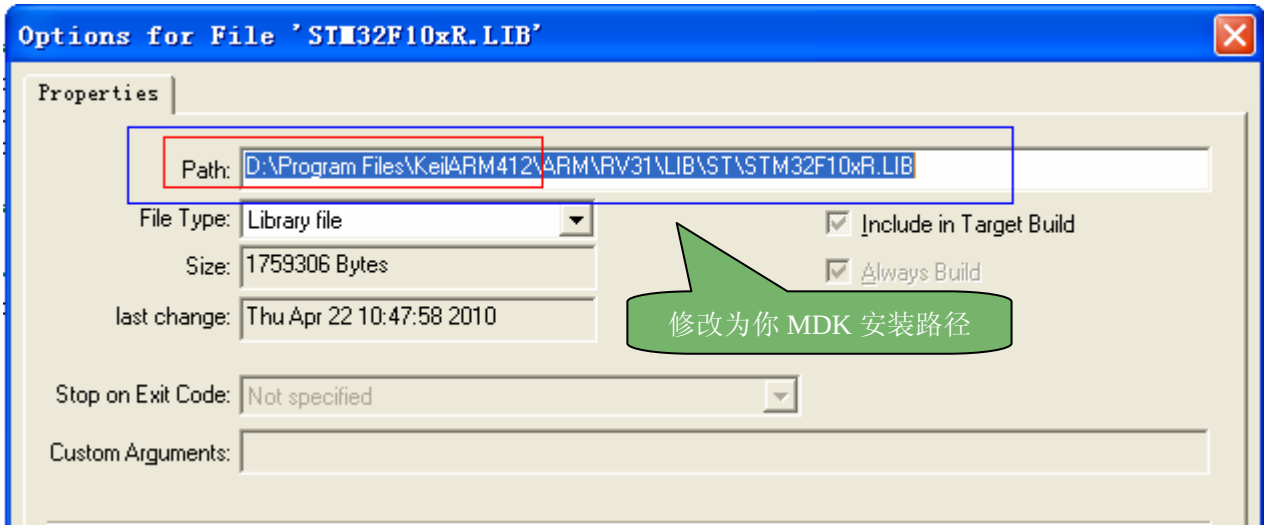
STM32F10xD.LIB 就是打开了 DEBUG 调试生成的库文件，STM32F10xR.LIB 就是关闭了 DEBUG 生成的发行库文件。如果你发现了库函数的 BUG，你可以修改库源文件重新编译生成库文件替换原来的库。

BHS-STM32 实验 3-GPIO输出-LED闪灯(软件延时方式)(库函数)

如果你的 MDK 安装路径与我的不同，那么使用库函数的例子需要修改库文件路径

我的 STM32F10xR.LIB 路径如下





注意：红框中路径填你的库文件 STM32F10xR.LIB 的路径，该路径是 KEIL MDK 的安装路径

STM32F10xR.LIB 详细说明可以参考【\资料文档\BHS-STM32 文档】文件夹《STM32F101xx 与 STM32F103xx 固件函数库用户手册》

本例子是使用 ST 官方函数库，函数库说明请参考《STM32F101xx 与 STM32F103xx 固件函数库用户手册》

GPIO 输出实验，通过 LED 观察端口变化，使用 PC8~PC11 端口，通过软件延时改变端口状态

下面介绍本例使用到的库函数

说明：使用 STM32 任何外设都要用到对应的 APB 时钟，只有开启 APB 时钟外设才能工作

函数 RCC_APB2PeriphClockCmd

函数名	RCC_APB2PeriphClockCmd
函数原形	void RCC_APB2PeriphClockCmd(u32 RCC_APB2Periph, FunctionalState NewState)
功能描述	使能或者失能 APB2 外设时钟
输入参数 1	RCC_APB2Periph：门控 APB2 外设时钟 参阅 Section：RCC_APB2Periph 查阅更多该参数允许取值范围
输入参数 2	NewState：指定外设时钟的新状态 这个参数可以取：ENABLE 或者 DISABLE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

RCC_APB2Periph

该参数被门控的 APB2 外设时钟，可以取下表的一个或者多个取值的组合作为该参数的值。

RCC_AHB2Periph	描述
RCC_APB2Periph_AFIO	功能复用 IO 时钟
RCC_APB2Periph_GPIOA	GPIOA 时钟
RCC_APB2Periph_GPIOB	GPIOB 时钟
RCC_APB2Periph_GPIOC	GPIOC 时钟
RCC_APB2Periph_GPIOD	GPIOD 时钟
RCC_APB2Periph_GPIOE	GPIOE 时钟
RCC_APB2Periph_ADC1	ADC1 时钟
RCC_APB2Periph_ADC2	ADC2 时钟
RCC_APB2Periph_TIM1	TIM1 时钟
RCC_APB2Periph_SPI1	SPI1 时钟
RCC_APB2Periph_USART1	USART1 时钟
RCC_APB2Periph_ALL	全部 APB2 外设时钟

函数 GPIO_Init

函数名	GPIO_Init
函数原形	void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
功能描述	根据 GPIO_InitStruct 中指定的参数初始化外设 GPIOx 寄存器
输入参数 1	GPIOx：x 可以是 A，B，C，D 或者 E，来选择 GPIO 外设
输入参数 2	GPIO_InitStruct：指向结构 GPIO_InitTypeDef 的指针，包含了外设 GPIO 的配置信息 参阅 Section：GPIO_InitTypeDef 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

GPIO_InitTypeDef structure

GPIO_InitTypeDef 定义于文件“stm32f10x_gpio.h”：

```
typedef struct
{
u16 GPIO_Pin;
GPIOSpeed_TypeDef GPIO_Speed;
GPIOMode_TypeDef GPIO_Mode;
```

} GPIO_InitTypeDef;

GPIO_Pin

该参数选择待设置的 GPIO 管脚，使用操作符“|”可以一次选中多个管脚。可以使用下表中的任意组合。

GPIO_Pin	描述
GPIO_Pin_None	无管脚被选中
GPIO_Pin_0	选中管脚 0
GPIO_Pin_1	选中管脚 1
GPIO_Pin_2	选中管脚 2
GPIO_Pin_3	选中管脚 3
GPIO_Pin_4	选中管脚 4
GPIO_Pin_5	选中管脚 5
GPIO_Pin_6	选中管脚 6
GPIO_Pin_7	选中管脚 7
GPIO_Pin_8	选中管脚 8
GPIO_Pin_9	选中管脚 9
GPIO_Pin_10	选中管脚 10
GPIO_Pin_11	选中管脚 11
GPIO_Pin_12	选中管脚 12
GPIO_Pin_13	选中管脚 13
GPIO_Pin_14	选中管脚 14
GPIO_Pin_15	选中管脚 15
GPIO_Pin_All	选中全部管脚

GPIO_Speed 用以设置选中管脚的速率

GPIO_Speed	描述
GPIO_Speed_10MHz	最高输出速率 10MHz
GPIO_Speed_2MHz	最高输出速率 2MHz
GPIO_Speed_50MHz	最高输出速率 50MHz

GPIO_Mode 用以设置选中管脚的工作状态

GPIO_Speed	描述
GPIO_Mode_AIN	模拟输入
GPIO_Mode_IN_FLOATING	浮空输入
GPIO_Mode_IPD	下拉输入
GPIO_Mode_IPU	上拉输入
GPIO_Mode_Out_OD	开漏输出
GPIO_Mode_Out_PP	推挽输出
GPIO_Mode_AF_OD	复用开漏输出
GPIO_Mode_AF_PP	复用推挽输出

下面介绍使用库函数操作 GPIO

函数 GPIO_SetBits

函数名	GPIO_SetBits
函数原形	void GPIO_SetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
功能描述	设置指定的数据端口位
输入参数 1	GPIOx：x 可以是 A，B，C，D 或者 E，来选择 GPIO 外设
输入参数 2	GPIO_Pin：待设置的端口位 该参数可以取 GPIO_Pin_x(x 可以是 0-15)的任意组合 参阅 Section：GPIO_Pin 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

例：PC8~PC11 设置为 1。

GPIO_SetBits(GPIOC, GPIO_Pin_8 | GPIO_Pin_9| GPIO_Pin_10| GPIO_Pin_11);

函数 GPIO_ResetBits

函数名	GPIO_ResetBits
函数原形	void GPIO_ResetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
功能描述	清除指定的数据端口位
输入参数 1	GPIOx：x 可以是 A，B，C，D 或者 E，来选择 GPIO 外设
输入参数 2	GPIO_Pin：待清除的端口位 该参数可以取 GPIO_Pin_x(x 可以是 0-15)的任意组合 参阅 Section：GPIO_Pin 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

例：PC8~PC11 设置为 0。

GPIO_ResetBits(GPIOC, GPIO_Pin_8 | GPIO_Pin_9| GPIO_Pin_10| GPIO_Pin_11);

函数 GPIO_ReadInputData

函数名	GPIO_ReadInputData
函数原形	u16 GPIO_ReadInputData(GPIO_TypeDef* GPIOx)
功能描述	读取指定的 GPIO 端口输入
输入参数	GPIOx：x 可以是 A，B，C，D 或者 E，来选择 GPIO 外设
输出参数	无
返回值	GPIO 输入数据端口值
先决条件	无
被调用函数	无

例：

```
//读出 PC 口输入状态
u16 ReadValue;
ReadValue = GPIO_ReadInputData (GPIOC);
```

函数 GPIO_Write

函数名	GPIO_Write
函数原形	void GPIO_Write(GPIO_TypeDef* GPIOx, u16 PortVal)
功能描述	向指定 GPIO 数据端口写入数据
输入参数 1	GPIOx：x 可以是 A，B，C，D 或者 E，来选择 GPIO 外设
输入参数 2	PortVal：待写入端口数据寄存器的值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

例：

```
//写数据到 PA 口
GPIO_Write(GPIOA, 0x1101);
```

上面介绍了比较常用的 GPIO 函数。
关于 GPIO 更多函数请参考《STM32F101xx 与 STM32F103xx 固件函数库用户手册》
或者在 KEIL 安装目录\ARM\RV31\LIB\ST\STM32F10x\stm32f10x_gpio.c 了解更多函数信息
stm32f10x_gpio.c 是 GPIO 库函数的源文件。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);//使能 GPIO 时钟
//特别提醒：STM32 初始化外设第一步是开启 APB 时钟，根据前面的结构图知道 GPIOC 是挂在 APB2
//下面的
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;//配置管脚
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //最大输出速度为 50MHz
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //通用推挽输出
GPIO_Init(GPIOC, &GPIO_InitStructure);                    //通用推挽输出，最大输出速度为 50MHz
```

在某些时候，比如用 GPIO 模拟时序时，建议直接操作寄存器，效率高些
但是理解了库函数编写的的代码可读行更强些。
下面我们来看看 stm32f10x_gpio.c 里面的源码

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  //前面 2 句是断言，调试排错用的
  GPIOx->BSRR = GPIO_Pin;//实际就是操作寄存器的，效率是一样的，多用的时间是函数调用的时间
}
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  GPIOx->BRR = GPIO_Pin; //实际就是操作寄存器的，效率是一样的，多用的时间是函数调用的时间
}
```

tyw藏书

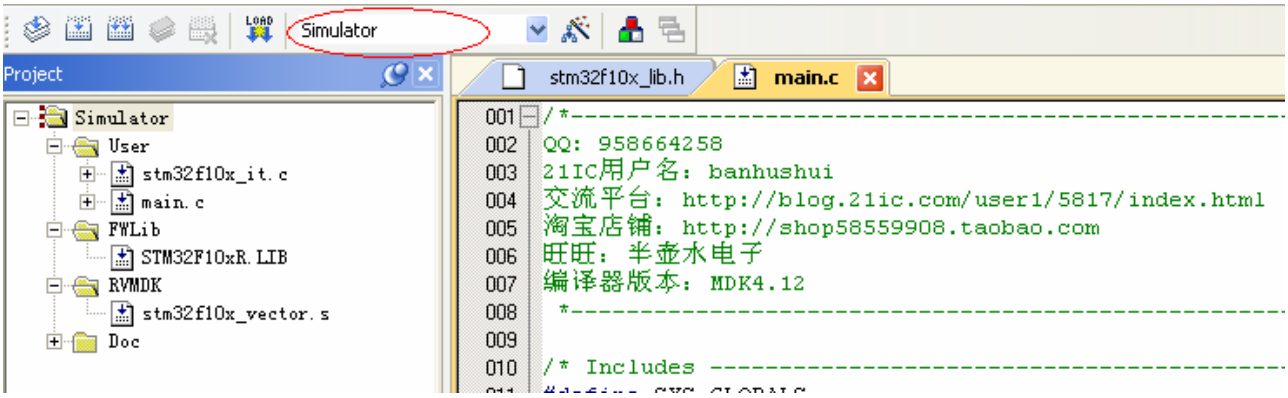
```
//LED 循环闪烁
void LedFlash(void)
{
   static u16 leds = 0x01;
   u16 temp;

      //先读出 PC 端口状态
      temp = GPIO_ReadInputData(GPIOC);

      //先屏蔽掉 PC8~PC11
      temp |= 0x0F00;

      //重新设置 PC8~PC11 输出状态，IO 输出低电平点亮 LED
      temp &= (~(leds<<8));
      GPIO_Write(GPIOC, temp);
      leds <<= 1;
      if ( (leds&0x0f) == 0)
          leds = 0x01;
}
//主函数
int main(void)
{
#ifdef DEBUG
   debug();
#endif

   /* System Clocks Configuration */
   RCC_Configuration();//配置系统时钟

   GPIO_Configuration();//配置 GPIO

   /* NVIC configuration */
   NVIC_Configuration();//配置中断

   //关闭所有 LED
   GPIO_Write(GPIOC, GPIO_ReadInputData(GPIOC)|0x00000F00);
   Delay(20);

   while (1)
   {
       Delay(50);

       //循环显示 1 位 LED
       LedFlash();
```

```
        Delay(50);

        //关闭所有 LED
        GPIO_Write(GPIOC, GPIO_ReadInputData(GPIOC)|0x00000F00);
    }
}
```

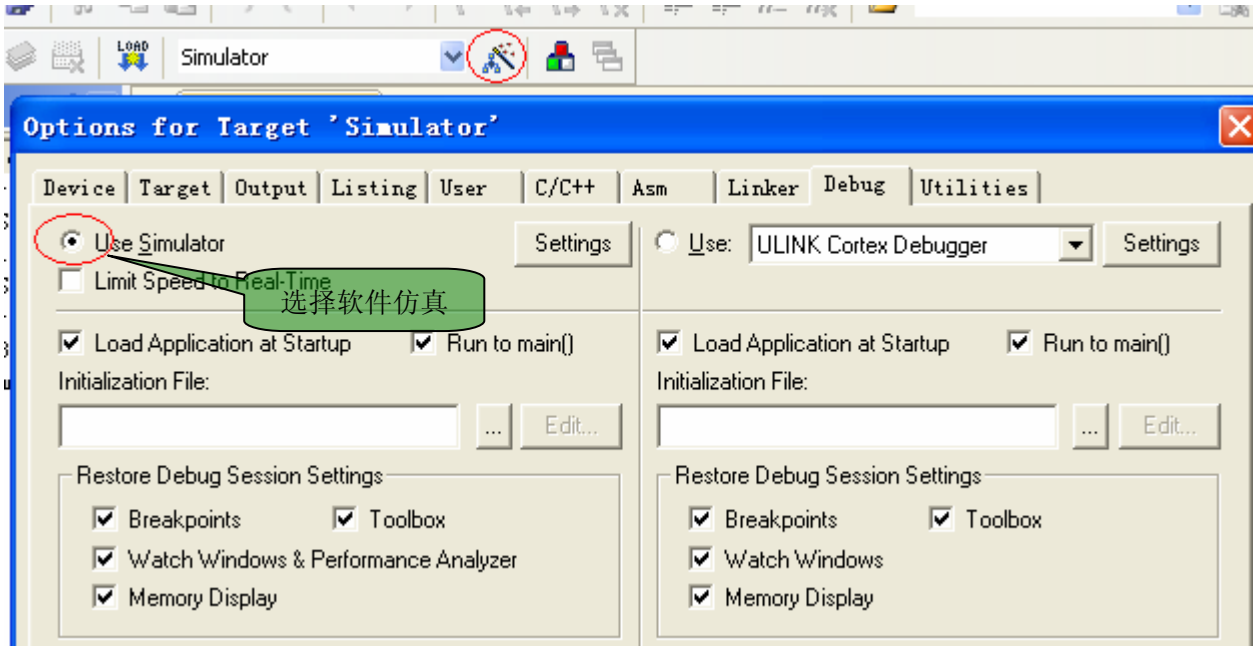
从上面可以知道，实际上库函数就是人家帮我们做了底层封装而已，没什么神秘的，这个完全是体力活。我们无需做上面重复的劳动。

软件仿真：

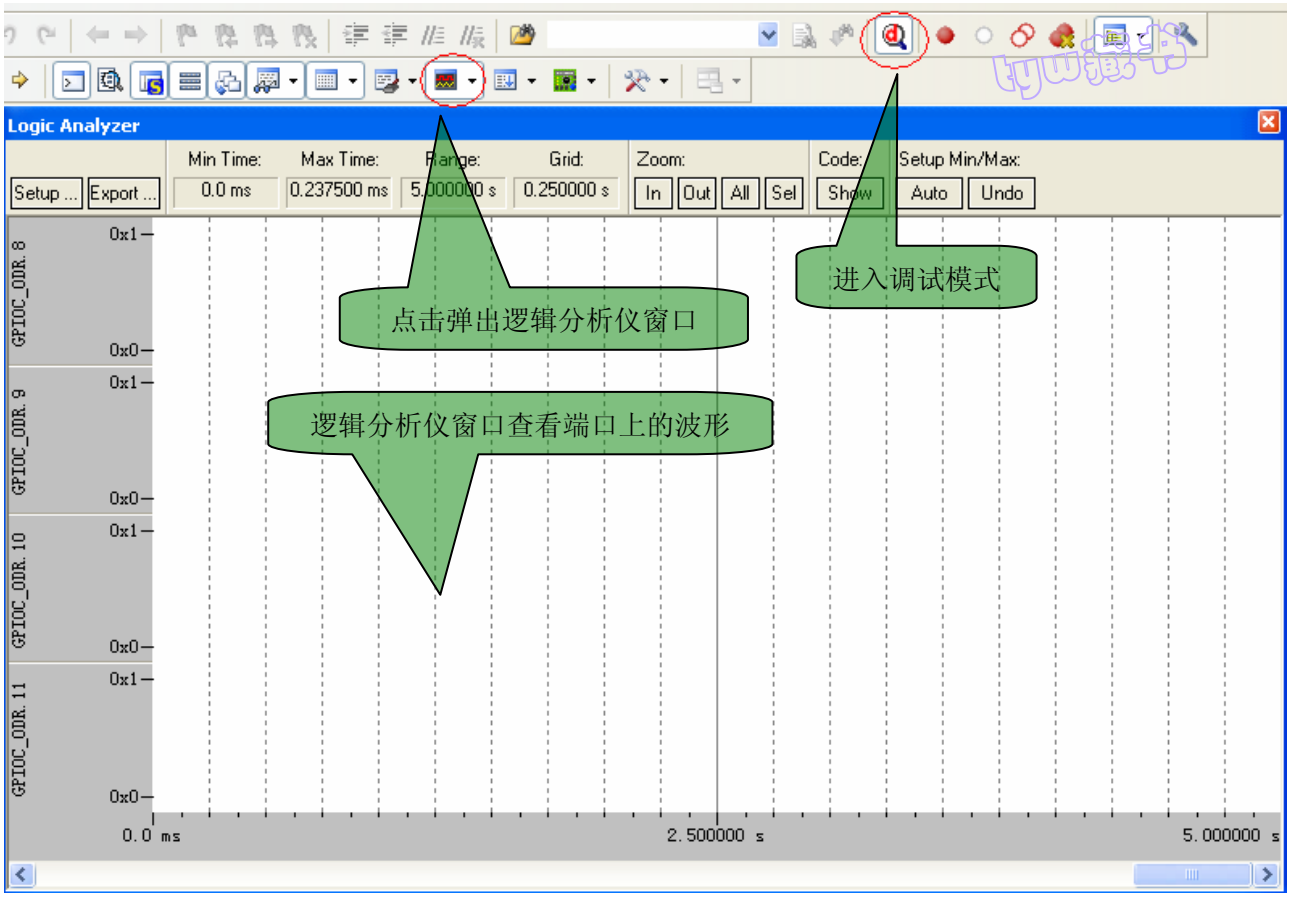
选择软件仿真



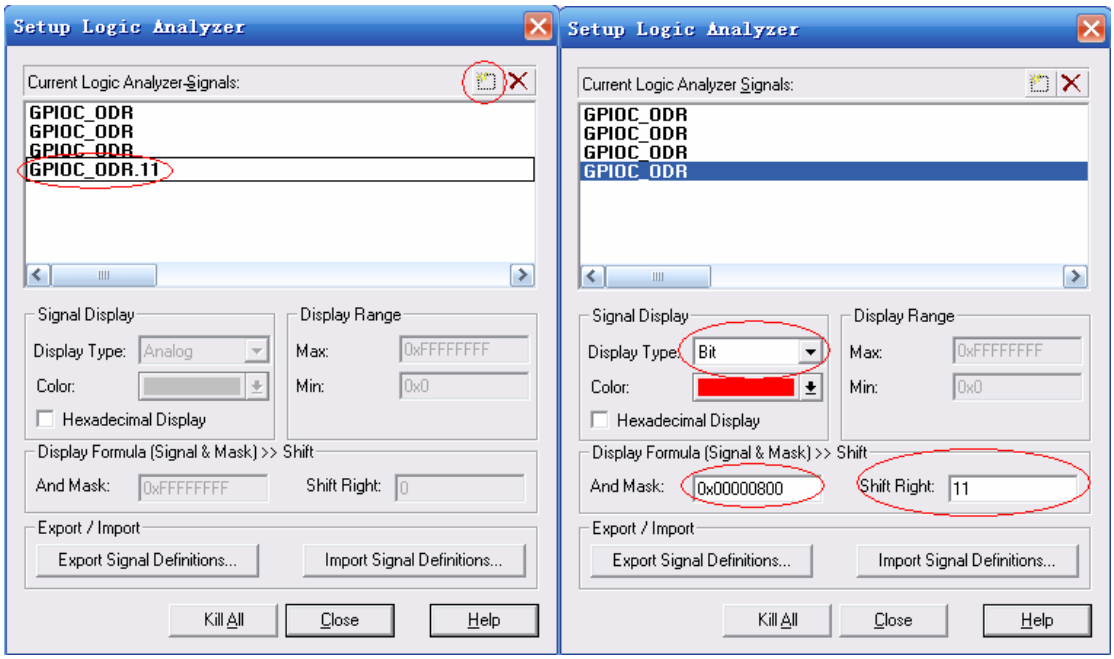
点击配置选项弹出下面的对话框



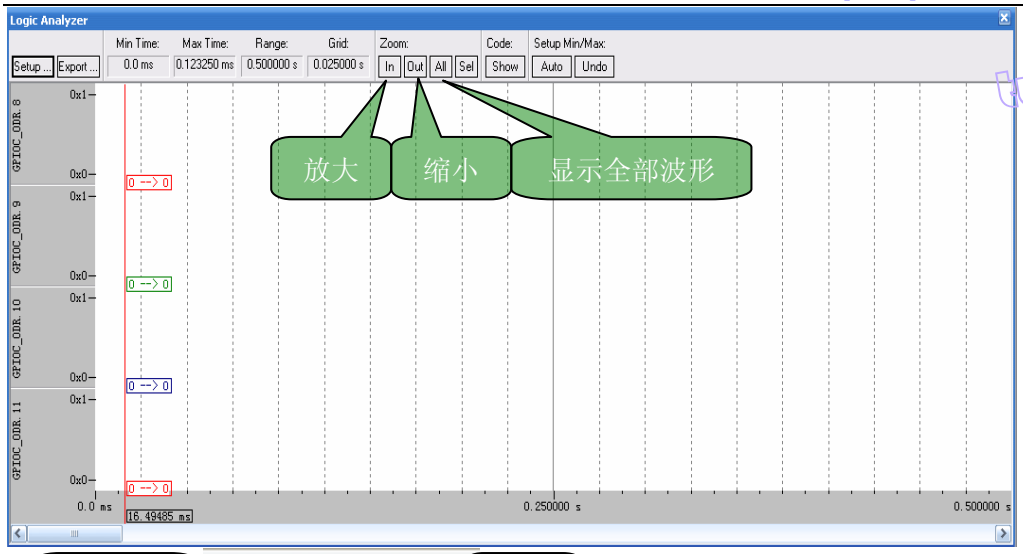
设置好后进入调试状态弹出下面对话框

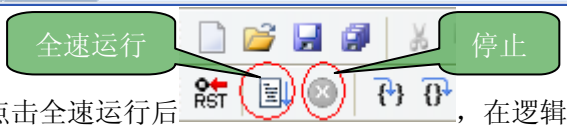


下面添加要查看的端口

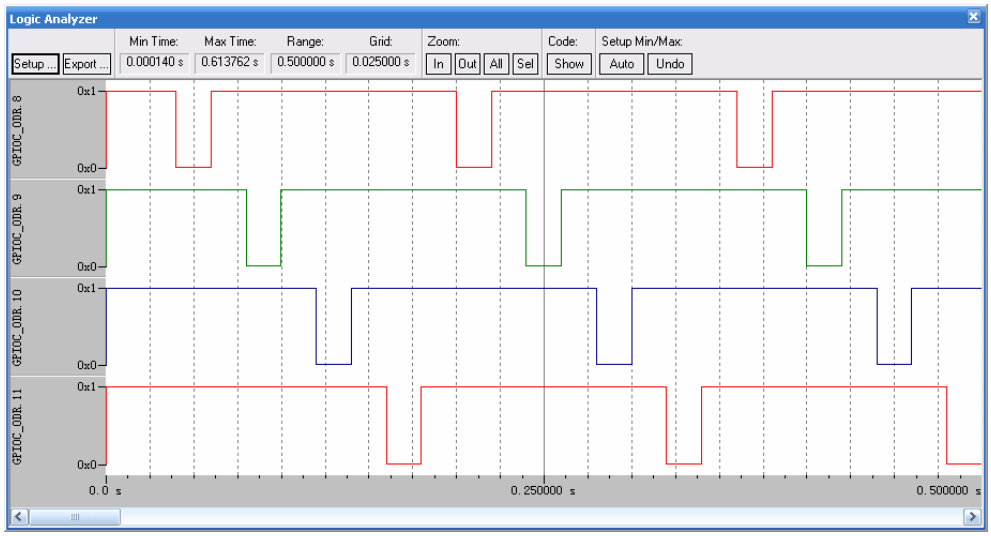


说明：【And Mask】 屏蔽掉不显示的位，【Shift Right】 将显示数据右移到最底位显示

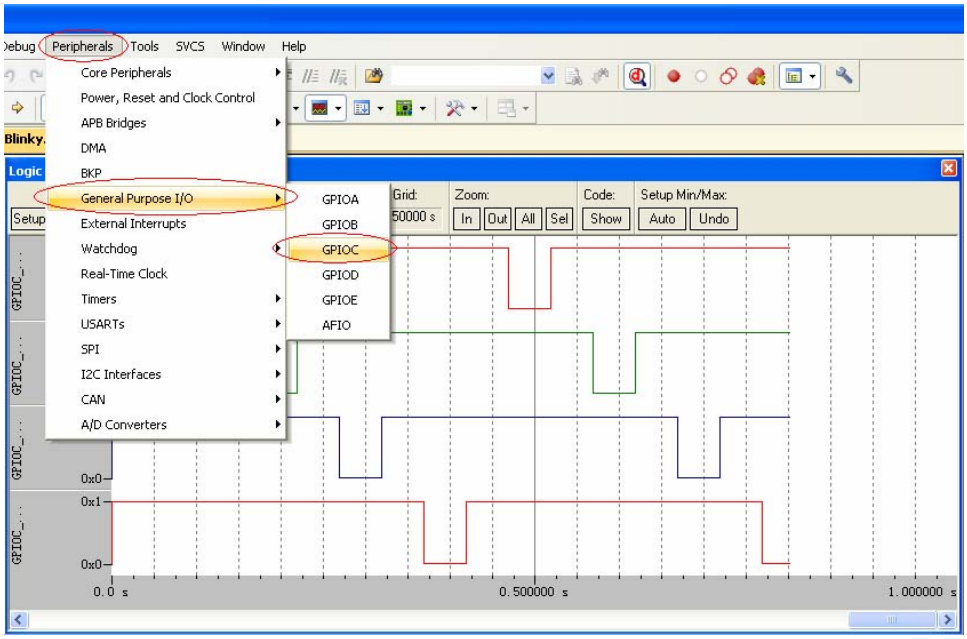




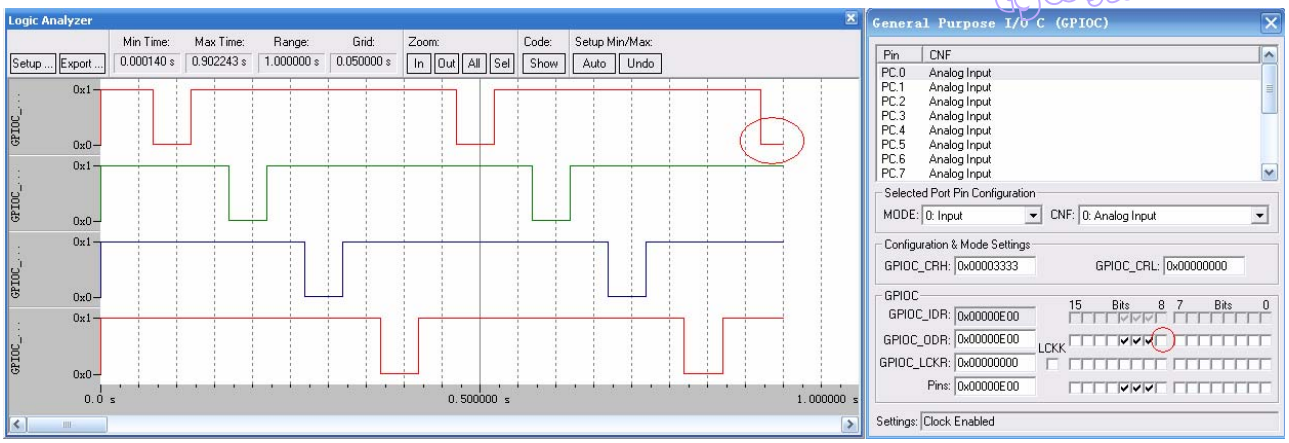
点击全速运行后，在逻辑分析仪里就能看到 PC8~PC11 的波形



再打开 GPIOC 窗口，看逻辑分析仪是否和 GPIOC 窗口同步。



GPIOC 窗口的 GPIOC_ODR 对应的位勾上表示端口是 1，没勾上表示端口是 0，运行可以看到勾勾在不停变化，因为是低电平点亮 LED，下图中 2 个红圈在同一时间都显示 GPIOC_ODR.8 是低电平。

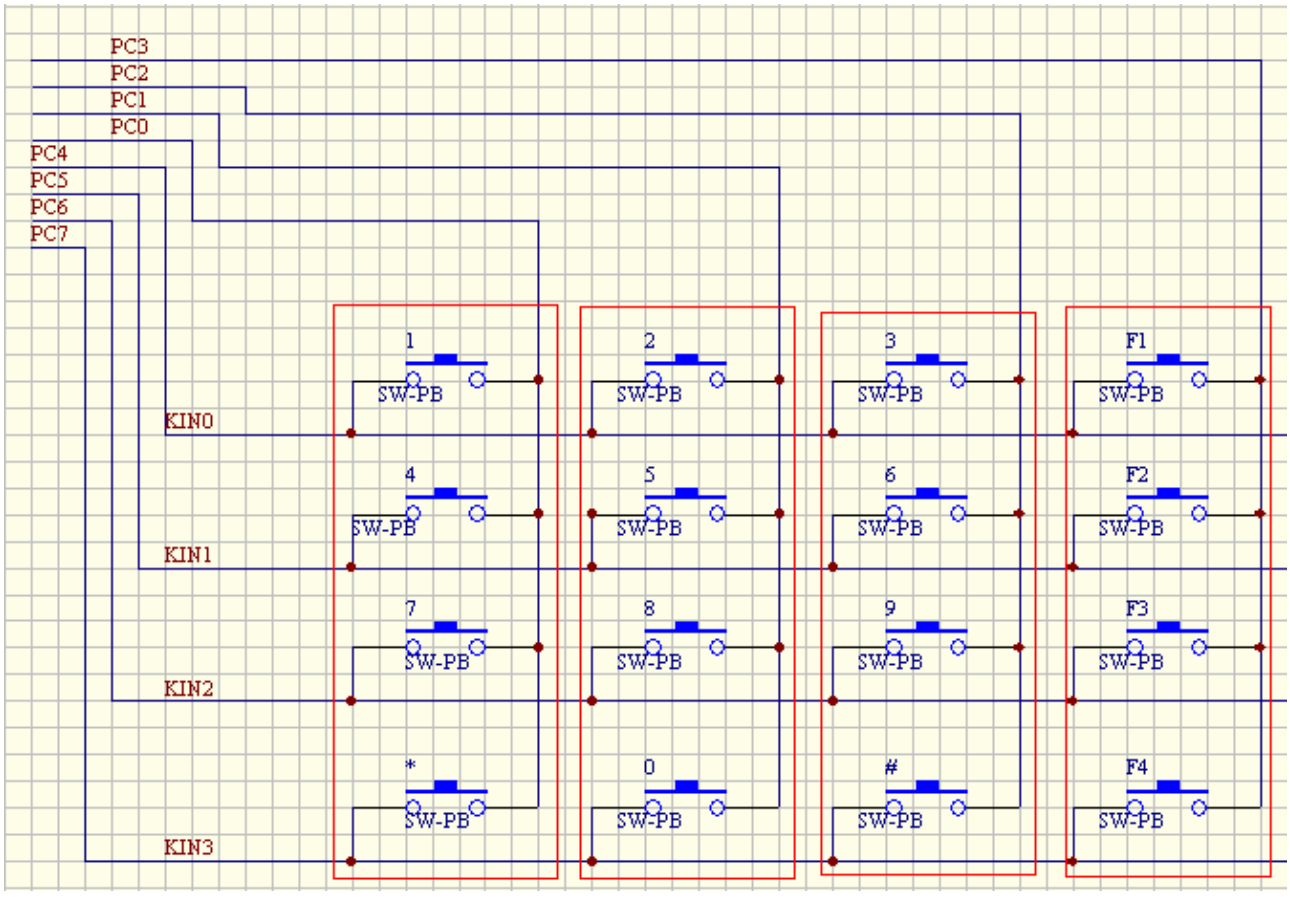


从上面的波形，和勾勾变化可以看出跟我们在硬件上直观看到 LED 循环亮灭效果是一样的。

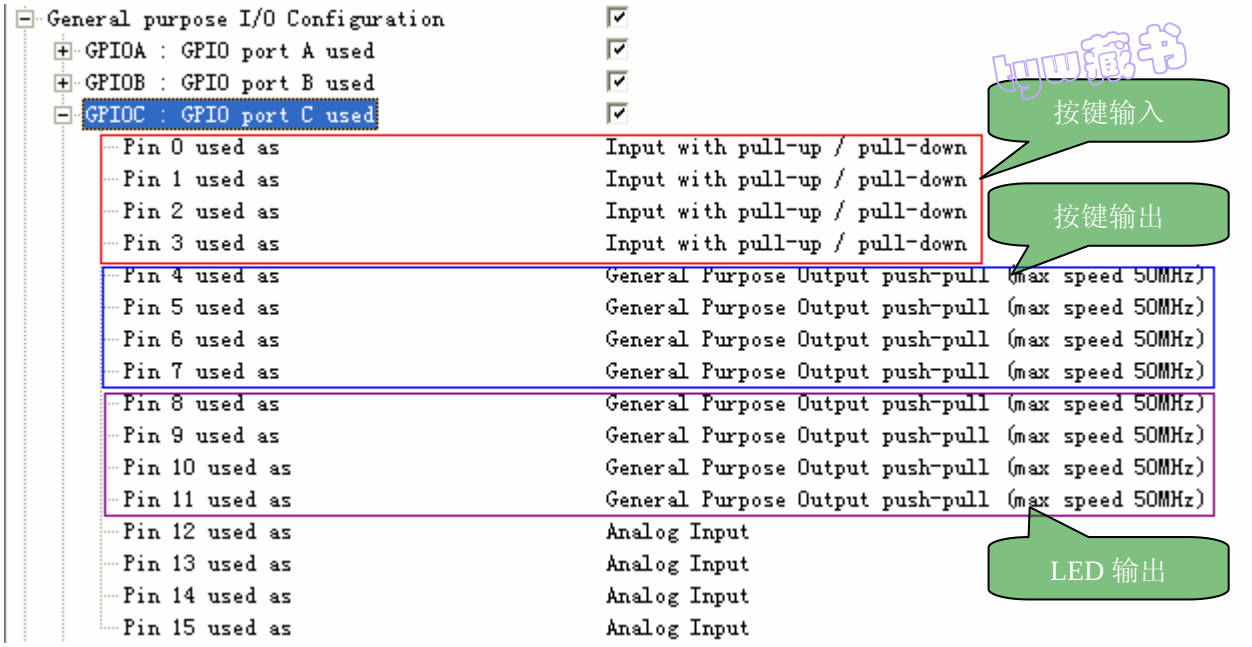
由上面的波形和 GPIOC 状态，我们可以看出本例与实验 1 是一样的。

BHS-STM32 实验 4-GPIO输入-(软件延时方式)(直接操作寄存器)

本例子是 GPIO 输入实验，把矩阵键盘的 4 个输出（PC4~PC7）设置为 0，4X4 键盘就变为 4 个独立按键了，这时红框中按键相当于一个按键，通过 LED1~LED4 状态判断键盘是按下还是弹起



下面是 GPIO 配置




```
//32 位数位操作某位
#define BIT32(n)              ( (u32)((u32)1UL<<n) )

//判断某位是否=1，=1 返回 1，=0 返回 0
#define   isBit32(dat, n)   ((dat&BIT32(n)) ? 1: 0)//0~31

//设置 32 位数据某位=1
#define SetBit32(Value,Bit)   ( Value |= (1UL<<Bit) )        //Bit(0~31)//一定要用()
//设置 32 位数据某位=0
#define ClrBit32(Value,Bit)   ( Value &= (~(1UL<<Bit)) )

//下面是主函数根据输入点亮/熄灭 LED：
int main (void)
{
  u32 port;

   stm32_Init ();// STM32  初始化

   GPIOC->ODR |= (0x000f<<8);//LED 先全灭
   GPIOC->ODR &= ~(0x000f<<4);//输出设置为 0
   GPIOC->ODR |= 0x000f;     //输输入设置为 1 表示上拉
   while (1)
   {
       //Delay(100);
          Delay(10);

          port=GPIOC->IDR;//读 PC 端口状态

          //根据 PC0 状态点亮/熄灭 LED1-PC8
```

tyw藏书

```
        if(isBit32(port, 0)==1)
            SetBit32(GPIOC->ODR, 8);
        else
            ClrBit32(GPIOC->ODR, 8);

        //根据 PC1 状态点亮/熄灭 LED2-PC9
        if(isBit32(port, 1)==1)
            SetBit32(GPIOC->ODR, 9);
        else
            ClrBit32(GPIOC->ODR, 9);

        //根据 PC2 状态点亮/熄灭 LED3-PC10
        if(isBit32(port, 2)==1)
            SetBit32(GPIOC->ODR, 10);
        else
            ClrBit32(GPIOC->ODR, 10);

        //根据 PC3 状态点亮/熄灭 LED4-PC11
        if(isBit32(port, 3)==1)
            SetBit32(GPIOC->ODR, 11);
        else
            ClrBit32(GPIOC->ODR, 11);
    }
}
```

BHS-STM32 实验 5-GPIO输入-(软件延时方式)(库函数)

本例子是 GPIO 输入实验，把矩阵键盘的 4 个输出（PC4~PC7）设置为 0，4X4 键盘就变为 4 个独立按键了，这时红框中按键相当于一个按键，通过 LED1~LED4 状态判断键盘是按下还是弹起

```
//下面是 GPIO 配置：
void GPIO_Configuration(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Enable GPIO_LED clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);//使能 GPIO 时钟
  //LED_init-------------------------------------------------------
  //配置端口 PC8~PC11  为 LED 输出
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//最大输出速度为 50MHz
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;////通用推挽输出
  GPIO_Init(GPIOC, &GPIO_InitStructure);                    //设置 GPIO 为输出

  //独立按键输入，把矩阵键盘的 4 个输出设置为 0，4X4 键盘就变为 4 个独立按键了
  //配置端口 PC4~PC7  为按键输出
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
```

```
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//最大输出速度为 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;////通用推挽输出
    GPIO_Init(GPIOC, &GPIO_InitStructure);                    //设置 GPIO 为输出

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//最大输出速度为 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;////通用推挽输出
    GPIO_Init(GPIOC, &GPIO_InitStructure);                    //设置 GPIO 为输出

    //配置端口 PC0~PC3  为按键输入
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;//上/下拉模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    //注意这里必须设置为 1 才是上拉输入
    GPIO_SetBits(GPIOC, GPIO_Pin_0);
    GPIO_SetBits(GPIOC, GPIO_Pin_1);
    GPIO_SetBits(GPIOC, GPIO_Pin_2);
    GPIO_SetBits(GPIOC, GPIO_Pin_3);
}
//下面是主函数根据输入点亮/熄灭 LED：
int main(void)
{
      u32 port;
#ifdef DEBUG
   debug();
#endif

   /* System Clocks Configuration */
   RCC_Configuration();//配置系统时钟

   GPIO_Configuration();//配置 GPIO

   /* NVIC configuration */
   NVIC_Configuration();//配置中断

   //关闭所有 LED
   GPIO_Write(GPIOC, GPIO_ReadInputData(GPIOC)|0x00000F00);
   Delay(20);

   while (1)
   {
       Delay(50);
          port=GPIO_ReadInputData(GPIOC);//读 PC 端口状态
```

```
        //根据 PC0 状态点亮/熄灭 LED1-PC8
        if(isBit32(port, 0)==1)
            GPIO_SetBits(GPIOC, GPIO_Pin_8);
        else
            GPIO_ResetBits(GPIOC, GPIO_Pin_8);

        //根据 PC1 状态点亮/熄灭 LED2-PC9
        if(isBit32(port, 1)==1)
            GPIO_SetBits(GPIOC, GPIO_Pin_9);
        else
            GPIO_ResetBits(GPIOC, GPIO_Pin_9);

        //根据 PC2 状态点亮/熄灭 LED3-PC10
        if(isBit32(port, 2)==1)
            GPIO_SetBits(GPIOC, GPIO_Pin_10);
        else
            GPIO_ResetBits(GPIOC, GPIO_Pin_10);

        //根据 PC3 状态点亮/熄灭 LED4-PC11
        if(isBit32(port, 3)==1)
            GPIO_SetBits(GPIOC, GPIO_Pin_11);
        else
            GPIO_ResetBits(GPIOC, GPIO_Pin_11);
    }
}
```

BHS-STM32 实验 6-像 51 单片机一样操作STM32 的GPIO

本例功能同实验 4，实验 5

本节原理来自《Cortex-M3 内核技术参考手册》的第 4.2 Bit-banding

Bit-banding

处理器存储器映射包括两个 bit-banding 区域。它们分别为 SRAM 和外设存储区域中的低的 1MB。这些 bit-band 区域将存储器别名区的一个字映射为 bit-band 区的一个位。Cortex-M3 存储器映射有 2 个 32MB 别名区，它们被映射为两个 1MB 的 bit-band 区。

- 对 32MB SRAM 别名区的访问映射为对 1MB SRAMbit-band 区的访问。
- 对 32MB 外设别名区的访问映射为对 1MB 外设 bit-band 区的访问。

映射公式显示如何将别名区中的字与 bit-band 区中的对应位或目标位关联。映射公式如：

bit_word_offset=(byte_offset×32)+(bit_number×4)

bit_word_addr=bit_band_base+bit_word_offset

这里：

Bit_word_offset 为 bit-band 存储区中的目标位的位置。

- Bit_word_addr 为别名存储区中映射为目标位的字的地址。
- Bit_band_base 是别名区的开始地址。
- Bit_offset 为 bit-band 区中包含目标位的字节的编号。
- Bit_number 为目标位的位位置（0-7）。

图 4-2 显示了 SRAM bit-band 别名区和 SRAMbit-band 区之间的 bit-band 映射的例子：

地址 0x23FFFFE0 的别名字映射为 0x200FFFFC 的 bit-band 字节的位 0：
0x23FFFFE0=0x22000000+(0xFFFFF*32)+0*4
地址 0x23FFFFEC 的别名字映射为 0x200FFFFC 的 bit-band 字节的位 7：
0x23FFFFEC=0x22000000+(0xFFFFF*32)+7*4
地址 0x22000000 的别名字映射为 0x20000000 的 bit-band 字节的位 0：
0x22000000=0x22000000+(0*32)+0*4
地址 0x220001C 的别名字映射为 0x20000000 的 bit-band 字节的位 0：
0x2200001C=0x22000000+(0*32)+7*4

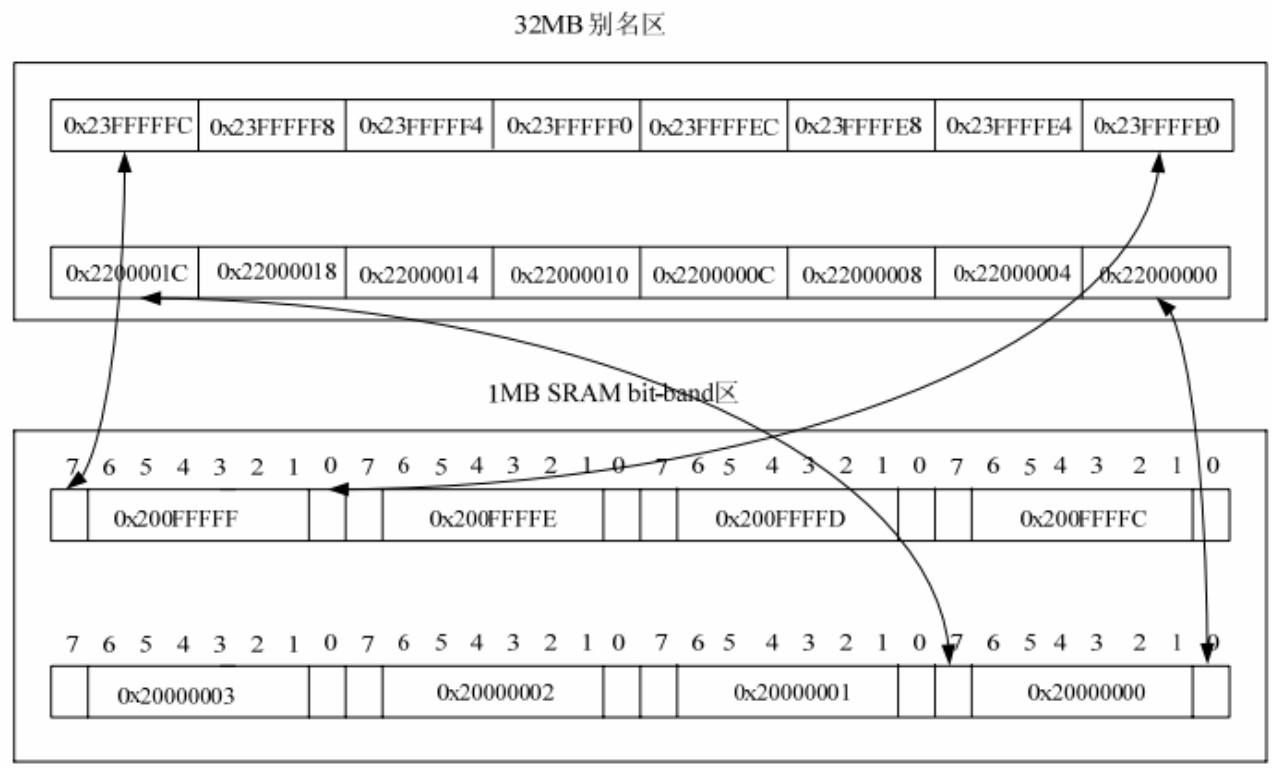


图 4-2 bit-band 映射

直接访问别名区

向别名区写入一个字与在 bit-band 区的目标位执行读-修改-写操作具有相同的作用。写入别名区的字的位 0 决定了写入 bit-band 区的目标位的值。将位 0 为 1 的值写入别名区表示向 bit-band 位写入 1，将位 0 为 0 的值写入别名区表示向 bit-band 位写入 0。别名字的位[31:1]在 bit-band 位上不起作用。写入 0x01 与写入 0xFF 的效果相同。写入 0x00 与写入 0x0E 的效果相同。

读别名区的一个字返回 0x01 或 0x00。0x01 表示 bit-band 区中的目标位置位。0x00 表示目标位清零。位[31:1]将为 0。

注：采用大端格式时，对 bit-band 别名区的访问必须以字节方式。否则访问值不可预知。

先声明宏定义如下：

```
//-----------------------------------------------------------------------------------------------
//别名区   ADDRESS=0x4200 0000 + (0x0001 100C*0x20) + (bitx*4) ;bitx:第 x 位
// 把“位段地址＋位序号”转换别名地址宏
#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))
```

```
//把该地址转换成一个指针
#define MEM_ADDR(addr)  *((volatile unsigned long  *)(addr))

#define BIT_ADDR(addr, bitnum)   MEM_ADDR( BITBAND(addr, bitnum)  )

#define GPIOA_ODR_Addr    (GPIOA_BASE+12) //0x4001080C
#define GPIOB_ODR_Addr    (GPIOB_BASE+12) //0x40010C0C
#define GPIOC_ODR_Addr    (GPIOC_BASE+12) //0x4001100C
#define GPIOD_ODR_Addr    (GPIOD_BASE+12) //0x4001140C
#define GPIOE_ODR_Addr    (GPIOE_BASE+12) //0x4001180C

#define GPIOA_IDR_Addr    (GPIOA_BASE+8) //0x40010808
#define GPIOB_IDR_Addr    (GPIOB_BASE+8) //0x40010C08
#define GPIOC_IDR_Addr    (GPIOC_BASE+8) //0x40011008
#define GPIOD_IDR_Addr    (GPIOD_BASE+8) //0x40011408
#define GPIOE_IDR_Addr    (GPIOE_BASE+8) //0x40011808

//------------------------------------------------
#define PA0   BIT_ADDR(GPIOA_ODR_Addr, 0)  //输出
#define PA1   BIT_ADDR(GPIOA_ODR_Addr, 1)  //输出
#define PA2   BIT_ADDR(GPIOA_ODR_Addr, 2)  //输出
#define PA3   BIT_ADDR(GPIOA_ODR_Addr, 3)  //输出
#define PA4   BIT_ADDR(GPIOA_ODR_Addr, 4)  //输出
#define PA5   BIT_ADDR(GPIOA_ODR_Addr, 5)  //输出
#define PA6   BIT_ADDR(GPIOA_ODR_Addr, 6)  //输出
#define PA7   BIT_ADDR(GPIOA_ODR_Addr, 7)  //输出
#define PA8   BIT_ADDR(GPIOA_ODR_Addr, 8)  //输出
#define PA9   BIT_ADDR(GPIOA_ODR_Addr, 9)  //输出
#define PA10  BIT_ADDR(GPIOA_ODR_Addr, 10) //输出
#define PA11  BIT_ADDR(GPIOA_ODR_Addr, 11) //输出
#define PA12  BIT_ADDR(GPIOA_ODR_Addr, 12) //输出
#define PA13  BIT_ADDR(GPIOA_ODR_Addr, 13) //输出
#define PA14  BIT_ADDR(GPIOA_ODR_Addr, 14) //输出
#define PA15  BIT_ADDR(GPIOA_ODR_Addr, 15) //输出

#define PA0in   BIT_ADDR(GPIOA_IDR_Addr, 0)  //输入
#define PA1in   BIT_ADDR(GPIOA_IDR_Addr, 1)  //输入
#define PA2in   BIT_ADDR(GPIOA_IDR_Addr, 2)  //输入
#define PA3in   BIT_ADDR(GPIOA_IDR_Addr, 3)  //输入
#define PA4in   BIT_ADDR(GPIOA_IDR_Addr, 4)  //输入
#define PA5in   BIT_ADDR(GPIOA_IDR_Addr, 5)  //输入
#define PA6in   BIT_ADDR(GPIOA_IDR_Addr, 6)  //输入
#define PA7in   BIT_ADDR(GPIOA_IDR_Addr, 7)  //输入
#define PA8in   BIT_ADDR(GPIOA_IDR_Addr, 8)  //输入
#define PA9in   BIT_ADDR(GPIOA_IDR_Addr, 9)  //输入
```

tyw藏书

```
#define PA10in  BIT_ADDR(GPIOA_IDR_Addr, 10) //输入
#define PA11in  BIT_ADDR(GPIOA_IDR_Addr, 11) //输入
#define PA12in  BIT_ADDR(GPIOA_IDR_Addr, 12) //输入
#define PA13in  BIT_ADDR(GPIOA_IDR_Addr, 13) //输入
#define PA14in  BIT_ADDR(GPIOA_IDR_Addr, 14) //输入
#define PA15in  BIT_ADDR(GPIOA_IDR_Addr, 15) //输入

//------------------------------------------------
#define PB0   BIT_ADDR(GPIOB_ODR_Addr, 0)  //输出
#define PB1   BIT_ADDR(GPIOB_ODR_Addr, 1)  //输出
#define PB2   BIT_ADDR(GPIOB_ODR_Addr, 2)  //输出
#define PB3   BIT_ADDR(GPIOB_ODR_Addr, 3)  //输出
#define PB4   BIT_ADDR(GPIOB_ODR_Addr, 4)  //输出
#define PB5   BIT_ADDR(GPIOB_ODR_Addr, 5)  //输出
#define PB6   BIT_ADDR(GPIOB_ODR_Addr, 6)  //输出
#define PB7   BIT_ADDR(GPIOB_ODR_Addr, 7)  //输出
#define PB8   BIT_ADDR(GPIOB_ODR_Addr, 8)  //输出
#define PB9   BIT_ADDR(GPIOB_ODR_Addr, 9)  //输出
#define PB10  BIT_ADDR(GPIOB_ODR_Addr, 10) //输出
#define PB11  BIT_ADDR(GPIOB_ODR_Addr, 11) //输出
#define PB12  BIT_ADDR(GPIOB_ODR_Addr, 12) //输出
#define PB13  BIT_ADDR(GPIOB_ODR_Addr, 13) //输出
#define PB14  BIT_ADDR(GPIOB_ODR_Addr, 14) //输出
#define PB15  BIT_ADDR(GPIOB_ODR_Addr, 15) //输出

#define PB0in   BIT_ADDR(GPIOB_IDR_Addr, 0)  //输入
#define PB1in   BIT_ADDR(GPIOB_IDR_Addr, 1)  //输入
#define PB2in   BIT_ADDR(GPIOB_IDR_Addr, 2)  //输入
#define PB3in   BIT_ADDR(GPIOB_IDR_Addr, 3)  //输入
#define PB4in   BIT_ADDR(GPIOB_IDR_Addr, 4)  //输入
#define PB5in   BIT_ADDR(GPIOB_IDR_Addr, 5)  //输入
#define PB6in   BIT_ADDR(GPIOB_IDR_Addr, 6)  //输入
#define PB7in   BIT_ADDR(GPIOB_IDR_Addr, 7)  //输入
#define PB8in   BIT_ADDR(GPIOB_IDR_Addr, 8)  //输入
#define PB9in   BIT_ADDR(GPIOB_IDR_Addr, 9)  //输入
#define PB10in  BIT_ADDR(GPIOB_IDR_Addr, 10) //输入
#define PB11in  BIT_ADDR(GPIOB_IDR_Addr, 11) //输入
#define PB12in  BIT_ADDR(GPIOB_IDR_Addr, 12) //输入
#define PB13in  BIT_ADDR(GPIOB_IDR_Addr, 13) //输入
#define PB14in  BIT_ADDR(GPIOB_IDR_Addr, 14) //输入
#define PB15in  BIT_ADDR(GPIOB_IDR_Addr, 15) //输入
//------------------------------------------------
#define PC0   BIT_ADDR(GPIOC_ODR_Addr, 0)  //输出
#define PC1   BIT_ADDR(GPIOC_ODR_Addr, 1)  //输出
#define PC2   BIT_ADDR(GPIOC_ODR_Addr, 2)  //输出
```

tyw藏书

```
#define PC3   BIT_ADDR(GPIOC_ODR_Addr, 3)  //输出
#define PC4   BIT_ADDR(GPIOC_ODR_Addr, 4)  //输出
#define PC5   BIT_ADDR(GPIOC_ODR_Addr, 5)  //输出
#define PC6   BIT_ADDR(GPIOC_ODR_Addr, 6)  //输出
#define PC7   BIT_ADDR(GPIOC_ODR_Addr, 7)  //输出
#define PC8   BIT_ADDR(GPIOC_ODR_Addr, 8)  //输出
#define PC9   BIT_ADDR(GPIOC_ODR_Addr, 9)  //输出
#define PC10  BIT_ADDR(GPIOC_ODR_Addr, 10) //输出
#define PC11  BIT_ADDR(GPIOC_ODR_Addr, 11) //输出
#define PC12  BIT_ADDR(GPIOC_ODR_Addr, 12) //输出
#define PC13  BIT_ADDR(GPIOC_ODR_Addr, 13) //输出
#define PC14  BIT_ADDR(GPIOC_ODR_Addr, 14) //输出
#define PC15  BIT_ADDR(GPIOC_ODR_Addr, 15) //输出

#define PC0in   BIT_ADDR(GPIOC_IDR_Addr, 0)  //输入
#define PC1in   BIT_ADDR(GPIOC_IDR_Addr, 1)  //输入
#define PC2in   BIT_ADDR(GPIOC_IDR_Addr, 2)  //输入
#define PC3in   BIT_ADDR(GPIOC_IDR_Addr, 3)  //输入
#define PC4in   BIT_ADDR(GPIOC_IDR_Addr, 4)  //输入
#define PC5in   BIT_ADDR(GPIOC_IDR_Addr, 5)  //输入
#define PC6in   BIT_ADDR(GPIOC_IDR_Addr, 6)  //输入
#define PC7in   BIT_ADDR(GPIOC_IDR_Addr, 7)  //输入
#define PC8in   BIT_ADDR(GPIOC_IDR_Addr, 8)  //输入
#define PC9in   BIT_ADDR(GPIOC_IDR_Addr, 9)  //输入
#define PC10in  BIT_ADDR(GPIOC_IDR_Addr, 10) //输入
#define PC11in  BIT_ADDR(GPIOC_IDR_Addr, 11) //输入
#define PC12in  BIT_ADDR(GPIOC_IDR_Addr, 12) //输入
#define PC13in  BIT_ADDR(GPIOC_IDR_Addr, 13) //输入
#define PC14in  BIT_ADDR(GPIOC_IDR_Addr, 14) //输入
#define PC15in  BIT_ADDR(GPIOC_IDR_Addr, 15) //输入

//------------------------------------------------
#define PD0   BIT_ADDR(GPIOD_ODR_Addr, 0)  //输出
#define PD1   BIT_ADDR(GPIOD_ODR_Addr, 1)  //输出
#define PD2   BIT_ADDR(GPIOD_ODR_Addr, 2)  //输出
#define PD3   BIT_ADDR(GPIOD_ODR_Addr, 3)  //输出
#define PD4   BIT_ADDR(GPIOD_ODR_Addr, 4)  //输出
#define PD5   BIT_ADDR(GPIOD_ODR_Addr, 5)  //输出
#define PD6   BIT_ADDR(GPIOD_ODR_Addr, 6)  //输出
#define PD7   BIT_ADDR(GPIOD_ODR_Addr, 7)  //输出
#define PD8   BIT_ADDR(GPIOD_ODR_Addr, 8)  //输出
#define PD9   BIT_ADDR(GPIOD_ODR_Addr, 9)  //输出
#define PD10  BIT_ADDR(GPIOD_ODR_Addr, 10) //输出
#define PD11  BIT_ADDR(GPIOD_ODR_Addr, 11) //输出
#define PD12  BIT_ADDR(GPIOD_ODR_Addr, 12) //输出
```

```
#define PD13  BIT_ADDR(GPIOD_ODR_Addr, 13) //输出
#define PD14  BIT_ADDR(GPIOD_ODR_Addr, 14) //输出
#define PD15  BIT_ADDR(GPIOD_ODR_Addr, 15) //输出

#define PD0in   BIT_ADDR(GPIOD_IDR_Addr, 0)  //输入
#define PD1in   BIT_ADDR(GPIOD_IDR_Addr, 1)  //输入
#define PD2in   BIT_ADDR(GPIOD_IDR_Addr, 2)  //输入
#define PD3in   BIT_ADDR(GPIOD_IDR_Addr, 3)  //输入
#define PD4in   BIT_ADDR(GPIOD_IDR_Addr, 4)  //输入
#define PD5in   BIT_ADDR(GPIOD_IDR_Addr, 5)  //输入
#define PD6in   BIT_ADDR(GPIOD_IDR_Addr, 6)  //输入
#define PD7in   BIT_ADDR(GPIOD_IDR_Addr, 7)  //输入
#define PD8in   BIT_ADDR(GPIOD_IDR_Addr, 8)  //输入
#define PD9in   BIT_ADDR(GPIOD_IDR_Addr, 9)  //输入
#define PD10in  BIT_ADDR(GPIOD_IDR_Addr, 10) //输入
#define PD11in  BIT_ADDR(GPIOD_IDR_Addr, 11) //输入
#define PD12in  BIT_ADDR(GPIOD_IDR_Addr, 12) //输入
#define PD13in  BIT_ADDR(GPIOD_IDR_Addr, 13) //输入
#define PD14in  BIT_ADDR(GPIOD_IDR_Addr, 14) //输入
#define PD15in  BIT_ADDR(GPIOD_IDR_Addr, 15) //输入

//------------------------------------------------
#define PE0   BIT_ADDR(GPIOE_ODR_Addr, 0)  //输出
#define PE1   BIT_ADDR(GPIOE_ODR_Addr, 1)  //输出
#define PE2   BIT_ADDR(GPIOE_ODR_Addr, 2)  //输出
#define PE3   BIT_ADDR(GPIOE_ODR_Addr, 3)  //输出
#define PE4   BIT_ADDR(GPIOE_ODR_Addr, 4)  //输出
#define PE5   BIT_ADDR(GPIOE_ODR_Addr, 5)  //输出
#define PE6   BIT_ADDR(GPIOE_ODR_Addr, 6)  //输出
#define PE7   BIT_ADDR(GPIOE_ODR_Addr, 7)  //输出
#define PE8   BIT_ADDR(GPIOE_ODR_Addr, 8)  //输出
#define PE9   BIT_ADDR(GPIOE_ODR_Addr, 9)  //输出
#define PE10  BIT_ADDR(GPIOE_ODR_Addr, 10) //输出
#define PE11  BIT_ADDR(GPIOE_ODR_Addr, 11) //输出
#define PE12  BIT_ADDR(GPIOE_ODR_Addr, 12) //输出
#define PE13  BIT_ADDR(GPIOE_ODR_Addr, 13) //输出
#define PE14  BIT_ADDR(GPIOE_ODR_Addr, 14) //输出
#define PE15  BIT_ADDR(GPIOE_ODR_Addr, 15) //输出

#define PE0in   BIT_ADDR(GPIOE_IDR_Addr, 0)  //输入
#define PE1in   BIT_ADDR(GPIOE_IDR_Addr, 1)  //输入
#define PE2in   BIT_ADDR(GPIOE_IDR_Addr, 2)  //输入
#define PE3in   BIT_ADDR(GPIOE_IDR_Addr, 3)  //输入
#define PE4in   BIT_ADDR(GPIOE_IDR_Addr, 4)  //输入
#define PE5in   BIT_ADDR(GPIOE_IDR_Addr, 5)  //输入
```

```
#define PE6in   BIT_ADDR(GPIOE_IDR_Addr, 6)  //输入
#define PE7in   BIT_ADDR(GPIOE_IDR_Addr, 7)  //输入
#define PE8in   BIT_ADDR(GPIOE_IDR_Addr, 8)  //输入
#define PE9in   BIT_ADDR(GPIOE_IDR_Addr, 9)  //输入
#define PE10in  BIT_ADDR(GPIOE_IDR_Addr, 10) //输入
#define PE11in  BIT_ADDR(GPIOE_IDR_Addr, 11) //输入
#define PE12in  BIT_ADDR(GPIOE_IDR_Addr, 12) //输入
#define PE13in  BIT_ADDR(GPIOE_IDR_Addr, 13) //输入
#define PE14in  BIT_ADDR(GPIOE_IDR_Addr, 14) //输入
#define PE15in  BIT_ADDR(GPIOE_IDR_Addr, 15) //输入

//举例:
//输出
PA0=0;
PA1=1;

//输入
if(PB0==0)
   ;//检测到低电平
//-------------------------------------------------------------------------------------------------
上面是不是有点 C51 的味道了
同样实现实验 3 的例子的代码如下：（初始化用一样的）
本例功能同实验 4，实验 5
//-------------------------------------------------------------------------------------------------
int main (void)
{
 u32 port;

  stm32_Init ();// STM32  初始化

  //LED 先全灭
  PC8=1;   PC9=1;   PC10=1;   PC11=1;

  //输出设置为 0
  PC4=0;   PC5=0;   PC6=0;     PC7=0;

  //输入设置为 1 表示上拉
  PC0=1; PC1=1; PC2=1; PC3=1;
  while (1)
  {
      //Delay(100);
        Delay(10);

        port=GPIOC->IDR;//读 PC 端口状态
```

```
        //根据 PC0 状态点亮/熄灭 LED1-PC8
        if(PC0in==1)
            PC8=1;
        else
            PC8=0;

        //根据 PC1 状态点亮/熄灭 LED2-PC9
        if(PC1in==1)
            PC9=1;
        else
            PC9=0;;

        //根据 PC2 状态点亮/熄灭 LED3-PC10
        if(PC2in==1)
            PC10=1;
        else
            PC10=0;

        //根据 PC3 状态点亮/熄灭 LED4-PC11
        if(PC3in==1)
            PC11=1;
        else
            PC11=0;
    }
}
```

lyw藏书

系统定时器(SysTick)实验

系统定时器功能描述：

系统定时器是 Cortex-M3 特殊的定时器，这是所有 Cortex-M3 内核 CPU 都有的。该定时器主要为操作系统设计的，用于操作系统产生系统节拍。系统嘀嗒校准值固定为 9000，当系统嘀嗒时钟设定为 9MHz(HCLK/8 的最大值)，产生 1ms 时间基准。《Cortex-M3 技术参考手册》里有详细说明，下面摘录与 SysTick 相关的寄存器描述如下：

系统时钟节拍控制与状态寄存器(SysTick_CTRL)

使用系统时钟节拍（SysTick）控制与状态寄存器来使能 SysTick 功能

地址: 0xE000E010 复位值: 0x00000000　访问类型: 读/写

域	名称	定义
[16]	COUNTFLAG	从上次读取定时器器开始，如果定时器计数到 0，则返回 1。读取时清零。
[2]	CLKSOURCE	0= 外部参考时钟 1= 内核时钟 如果没有提供参考时钟，那么该位保持为 1，并且因此赋予和内核时钟一样多的时间。内核时钟比参考时钟至少要快 2.5 倍。否则计数值将不可预测。
[1]	TICKINT	1= 向下计数至 0 会导致挂起 SysTick 处理器 0= 向下计数至 0 不会导致挂起 SysTick 处理器。软件可以使用 COUNTFLAG 来判断是否计数到 0。
[0]	ENABLE	1= 计数器工作在连拍模式（multi-shot）。即计数器装载重装值后接着开始往下计数。到计数到 0 时将 COUNTFLAG 设为 1，此时根据 TICKINT 的值可以选择是否挂起 SysTick 处理器。接着又再次装载重装值，并重新开始计数。 0= 禁能计数器

系统时钟节拍（SysTick）重装值寄存器(SysTick_LOAD)

在计数器到达 0 时，使用 SysTick 重装值寄存器来指定载入“当前值寄存器”的初始值。初始值可以是 1 到 0x00FFFFFF 之间的任何值。初始值也可以为 0，但因为从 1 计数到 0 时会将 SysTick 中断和 COUNTFLAG 激活，所以没有什么用处。

因此，作为一个连拍式（multi-shot）定时器，它每 N+1 个时钟脉冲就触发一次，周而复始，此处 N 为 1 到 0x00FFFFFF 之间的任意值。所以，如果每 100 个时钟脉冲就请求一次时钟中断（tick interrupt），那么必须向 RELOAD 载入 99。如果每次时钟中断后都写入一个新值，那么可以看作单拍（single shot）模式，因而必须写入实际的倒计数值。例如，如果在 400 个时钟脉冲后想请求一个时钟中断（tick），那么必须向 RELOAD 写入 400。

地址: 0xE000E014　复位状态: 不可预测　访问类型: 读/写

域	名称	定义
[23:0]	RELOAD	当计数器到达 0 时装载“当前值寄存器”的值

系统时钟节拍（SysTick）校准值寄存器

使用系统时钟节拍（SysTick）校准值寄存器通过乘法和除法运算可以将寄存器调节成任意所需的时钟速率。

地址: 0xE000E01C　复位状态: STCALIB　访问类型: 读

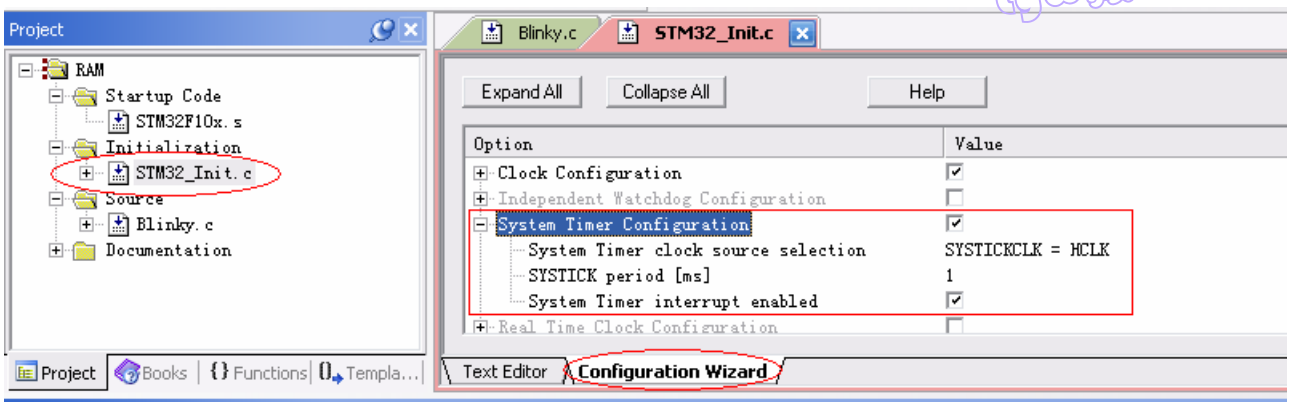
域	名称	定义
[31]	NOREF	1= 没有提供参考时钟
[30]	SKEW	1= 由于时钟频率的原因，校验值不是精确的 10ms。这会影响其作为一个软件实时时钟的适合性。
[23:0]	TENMS	该值是用于 10ms 定时的重装值。其值取决于 SKEW，它可以是精确的 10ms，也可以是最接近 10ms 的值。 如果为 0，那么检验值就未知。这很可能是因为参考时钟是系统的一个未知输入或者因为参考时钟可以动态调节。

BHS-STM32 实验 7-系统定时器(直接操作寄存器)

本实验功能：通过系统定时器定时，系统定时器中断产生 1ms 时钟节拍，循环点亮/熄灭 LED

本例子在实验 1 基础上修改，使用系统时钟定时，定时时间 1ms

下面是 SysTick 初始化配置如下




```
//下面是具体配置代码（切换到[Text Editor]模式才能看到）
#define __SYSTICK_SETUP              1
#define __SYSTICK_CTRL_VAL           0x00000006
#define __SYSTICK_PERIOD             0x00000001

#if __SYSTICK_SETUP
/*------------------------------------------------------------------------------
  STM32 System Timer setup.
  initializes the SysTick register
  *-----------------------------------------------------------------------------*/
__inline static void stm32_SysTickSetup (void) {

#if ((__SYSTICK_PERIOD*(__SYSTICKCLK/1000)-1) > 0xFFFFFF)          // 重载值太大了提示错误
    #error "Reload Value to large! Please use 'HCLK/8' as System Timer clock source or smaller period"
#else
   // set reload register 设置重载寄存器
   SysTick->LOAD  = __SYSTICK_PERIOD*(__SYSTICKCLK/1000)-1;
   // set clock source and Interrupt enable 设置时钟源和中断使能
   SysTick->CTRL  = __SYSTICK_CTRL_VAL;
   // clear the counter  清除计数值
   SysTick->VAL   =  0;
   // enable the counter  使能计数器
   SysTick->CTRL |= SYSTICK_CSR_ENABLE;
#endif
} // end of stm32_SysTickSetup
#endif
//在实验 1 基础上增加中断函数 SysTick_Handler，Delay 有点区别，其他完全一样
#define u16   unsigned short
#define u32   unsigned long

u32 volatile gTimer_1ms=0;

/*------------------------------------------------------------------------------
```

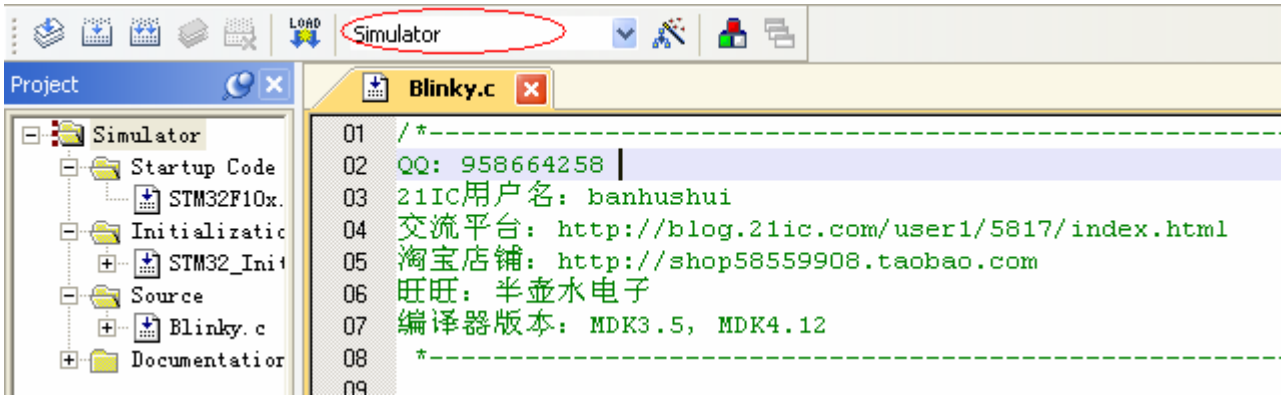
```
   Systick  中断函数
   SysTick 1 ms 中断 1 次
 *------------------------------------------------------------------------------*/
void SysTick_Handler (void)
{
      gTimer_1ms++;
} // end SysTick_Handler

//LED 循环闪烁
void LedFlash(void)
{
   static u16 leds = 0x01;
   u32 temp;

      //先读出 PC 端口状态
      temp = GPIOC->ODR;

      //先屏蔽掉 PC8~PC11
      temp |= 0x00000F00;

      //重新设置 PC8~PC11 输出状态，IO 输出低电平点亮 LED
      GPIOC->ODR = temp&(~(leds<<8));
      leds <<= 1;
      if ( (leds&0x0f) == 0)
            leds = 0x01;
}

//精确的延时
void Delay(u32 nTime)
{u32 counter;

   counter=gTimer_1ms;
   while( gTimer_1ms-counter < nTime);//定时时间到才退出

}
/*------------------------------------------------------------------------------
   MAIN function
 *------------------------------------------------------------------------------*/
int main (void)
{
   // STM32  初始化
   stm32_Init ();

   //关闭所有 LED
   GPIOC->ODR |= 0x00000F00;
```

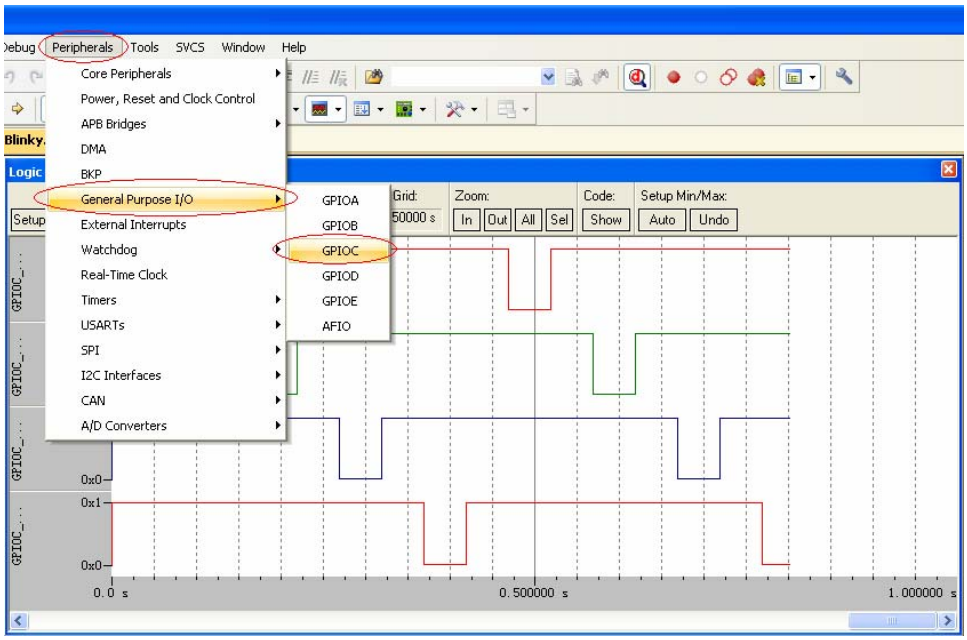
```
    Delay(20);

    while (1)
    {
        Delay(50);

        //循环显示 1 位 LED
        LedFlash();

        Delay(50);

        //关闭所有 LED
        GPIOC->ODR |= 0x00000F00;
    }
}
```

tyw藏书

软件仿真：

选择软件仿真





GPIOC 窗口的 GPIOC_ODR 对应的位勾上表示端口是 1，没勾上表示端口是 0，运行可以看到勾勾在不停

变化，因为是低电平点亮 LED，下图中 2 个红圈在同一时间都显示 GPIOC_ODR.8 是低电平。

从上面的波形，和勾勾变化可以看出跟我们在硬件上直观看到 LED 循环亮灭效果是一样的。

实验总结：

从代码看基本与实验 1 是一样的，功能也一样。但是本例由于使用的是硬件定时方式，所有时间更精确。从应用角度出发，如果系统对定时误差要求不高，用软件方式是可以的，如果对定时误差有严格要求那么使用硬件定时器是必须的。

Cortex-M3 内核的 CPU 的系统定时器是最简单的定时器。只要为操作系统应用设计的，与后面介绍的通用定时器区别非常大。通用定时器功能丰富，操作复杂，下面章节介绍。

BHS-STM32 实验 8-系统定时器(库函数)

本实验功能：通过系统定时器定时，系统定时器中断产生 1ms 时钟节拍，循环点亮/熄灭 LED
本实验使用库函数，功能与实验 6 相同，主要在初始化系统定时器时不同

函数 SysTick_SetReload

函数名	SysTick_SetReload
函数原形	void SysTick_SetReload(u32 Reload)
功能描述	设置 SysTick 重装载值
输入参数	Reload：重装载值 该参数取值必须在 1 和 0x00FFFFFF 之间
输出参数	无
返回值	无
先决条件	无
被调用函数	无

函数 SysTick_ITConfig

函数名	SysTick_ITConfig
函数原形	void SysTick_ITConfig(FunctionalState NewState)
功能描述	使能或者失能 SysTick 中断
输入参数	NewState：SysTick 中断的新状态 这个参数可以取：ENABLE 或者 DISABLE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

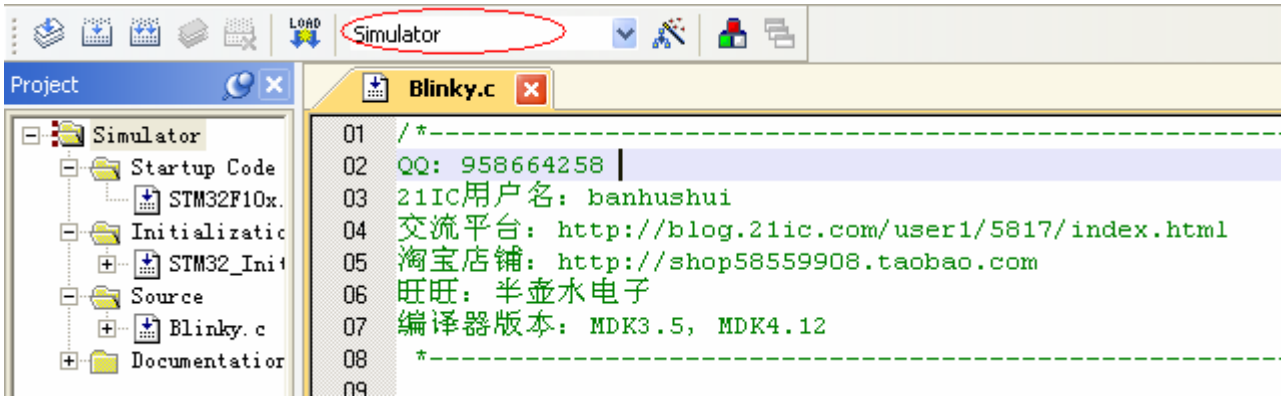
函数 SysTick_CounterCmd

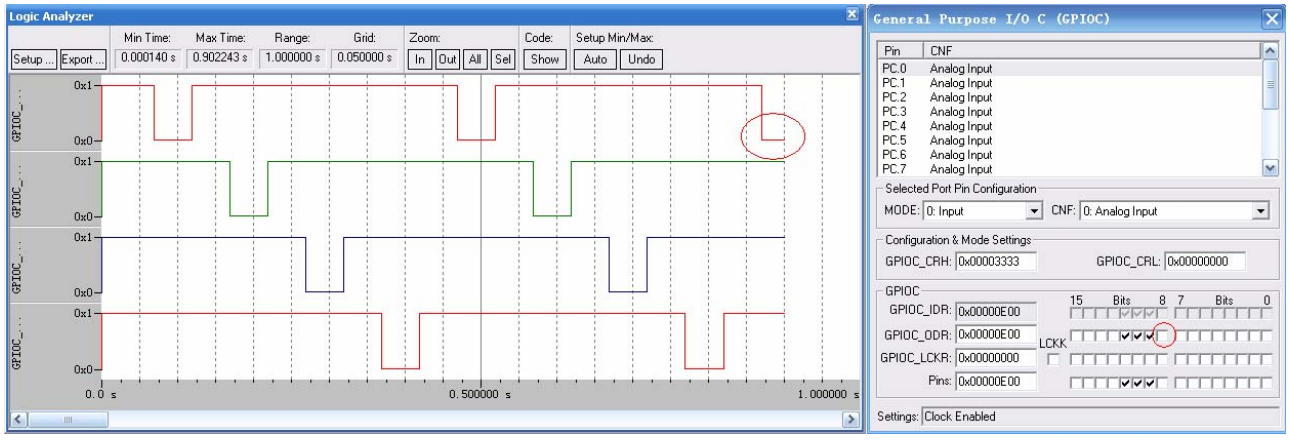
函数名	SysTick_CounterCmd
函数原形	void SysTick_CounterCmd(u32 SysTick_Counter)
功能描述	使能或者失能 SysTick 计数器
输入参数	SysTick_Counter：SysTick 计数器新状态 参阅 Section：SysTick_Counter 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

```
//系统定时器初始化
void SysTick_Init(void)
{
  /* SysTick end of count event each 1ms with input clock equal to 9MHz (HCLK/8, default) */
  //设置系统定时器中断时间为 1ms
  SysTick_SetReload(9000);
  /* Enable SysTick interrupt */
  //允许系统定时器中断
  SysTick_ITConfig(ENABLE);
  /* Enable the SysTick Counter */ //使能系统定时器
  SysTick_CounterCmd(SysTick_Counter_Enable);
}
```

软件仿真：

选择软件仿真





通用定时器Timer实验

tyw藏书

通用定时器功能描述

通用定时器功能复杂，本例只用到定时器的计算器模式,下面介绍相关内容

通用 TIMx (TIM2、TIM3、TIM4 和 TIM5)定时器功能包括：

- 16 位向上、向下、向上/向下自动装载计数器
- 16 位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为 1～65536 之间的任意数值
- 4 个独立通道：
  - 输入捕获
  - 输出比较
  - PWM 生成(边缘或中间对齐模式)
  - 单脉冲模式输出
- 使用外部信号控制定时器和定时器互连的同步电路
- 如下事件发生时产生中断/DMA：
  - 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)
  - 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)
  - 输入捕获
  - 输出比较
- 支持针对定位的增量(正交)编码器和霍尔传感器电路
- 触发输入作为外部时钟或者按周期的电流管理

时基单元

可编程通用定时器的主要部分是一个 16 位计数器和与其相关的自动装载寄存器。这个计数器可以向上计数、向下计数或者向上向下双向计数。此计数器时钟由预分频器分频得到。

计数器、自动装载寄存器和预分频器寄存器可以由软件读写，在计数器运行时仍可以读写。

时基单元包含：

- 计数器寄存器(TIMx_CNT)
- 预分频器寄存器 (TIMx_PSC)
- 自动装载寄存器 (TIMx_ARR)

自动装载寄存器是预先装载的，写或读自动重装载寄存器将访问预装载寄存器。根据在 TIMx_CR1 寄存器中的自动装载预装载使能位(ARPE)的设置，预装载寄存器的内容被立即或在每次的更新事件 UEV 时传送到影子寄存器。当计数器达到溢出条件(向下计数时的下溢条件)并当 TIMx_CR1 寄存器中的 UDIS 位等于’0’时，产生更新事件。更新事件也可以由软件产生。随后会详细描述每一种配置下更新事件的产生。

计数器由预分频器的时钟输出 CK_CNT 驱动，仅当设置了计数器 TIMx_CR1 寄存器中的计数器使能位(CEN)时，CK_CNT 才有效。(有关计数器使能的细节，请参见控制器的从模式描述)。

注：真正的计数器使能信号 CNT_EN 是在 CEN 的一个时钟周期后被设置。

预分频器描述

预分频器可以将计数器的时钟频率按 1 到 65536 之间的任意值分频。它是基于一个(在 TIMx_PSC 寄存器中的)16 位寄存器控制的 16 位计数器。这个控制寄存器带有缓冲器，它能够在工作时被改变。新的预分频器参数在下一次更新事件到来时被采用。

计数器模式

向上计数模式

在向上计数模式中，计数器从 0 计数到自动加载值(TIMx_ARR 计数器的内容)，然后重新从 0 开始

计数并且产生一个计数器溢出事件。

每次计数器溢出时可以产生更新事件，在 TIMx_EGR 寄存器中(通过软件方式或者使用从模式控制器)设置 UG 位也同样可以产生一个更新事件。

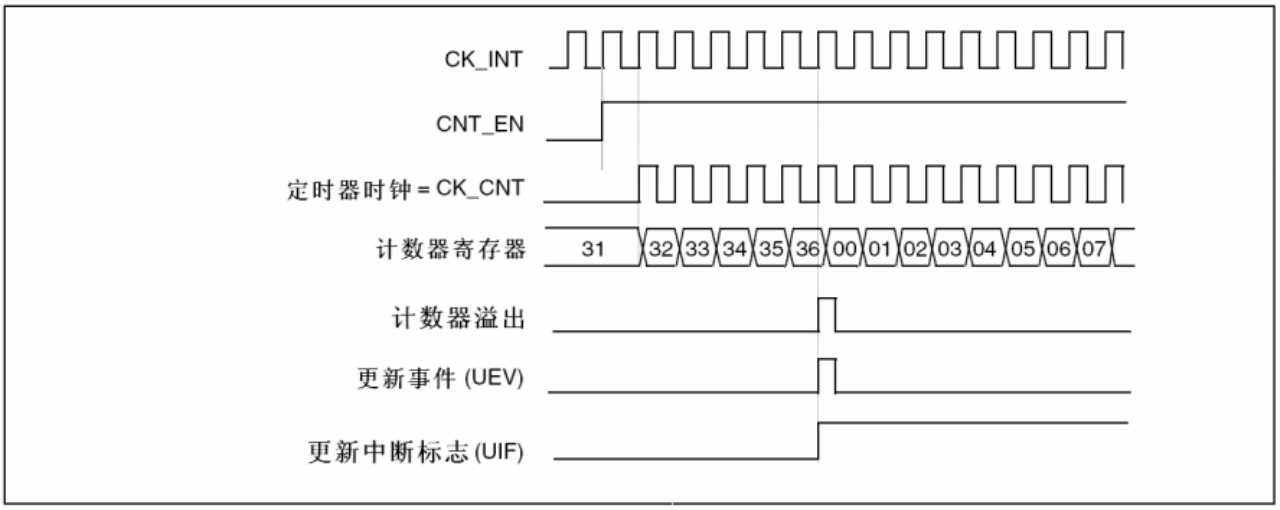
设置 TIMx_CR1 寄存器中的 UDIS 位，可以禁止更新事件；这样可以避免在向预装载寄存器中写入新值时更新影子寄存器。在 UDIS 位被清’0’之前，将不产生更新事件。但是在应该产生更新事件时，计数器仍会被清’0’，同时预分频器的计数也被请 0(但预分频系数不变)。此外，如果设置了 TIMx_CR1 寄存器中的 URS 位(选择更新请求)，设置 UG 位将产生一个更新事件 UEV，但硬件不设置 UIF 标志(即不产生中断或 DMA 请求)；这是为了避免在捕获模式下清除计数器时，同时产生更新和捕获中断。

当发生一个更新事件时，所有的寄存器都被更新，硬件同时(依据 URS 位)设置更新标志位(TIMx_SR 寄存器中的 UIF 位)。

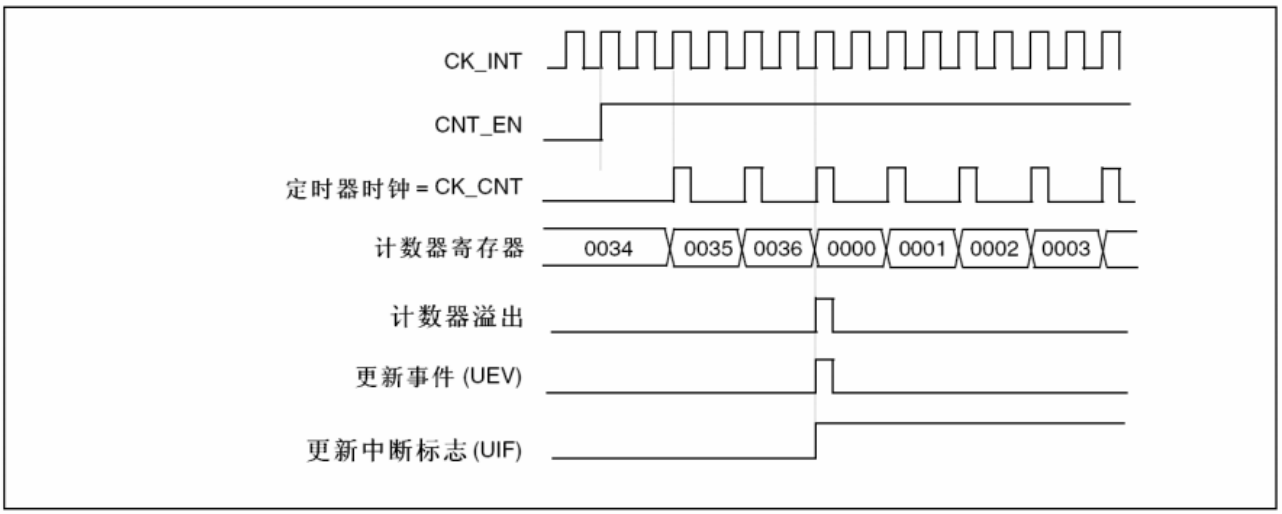
- 预分频器的缓冲区被置入预装载寄存器的值(TIMx_PSC 寄存器的内容)。
- 自动装载影子寄存器被重新置入预装载寄存器的值(TIMx_ARR)。

下图给出一些例子，当 TIMx_ARR=0x36 时计数器在不同时钟频率下的动作

计数器时序图，内部时钟分频因子为 1



计数器时序图，内部时钟分频因子为 2



向下计数模式

在向下模式中，计数器从自动装入的值(TIMx_ARR 计数器的值)开始向下计数到 0，然后从自动装入的值

重新开始并且产生一个计数器向下溢出事件。

每次计数器溢出时可以产生更新事件，在 TIMx_EGR 寄存器中(通过软件方式或者使用从模式控制器)设置 UG 位，也同样可以产生一个更新事件。

设置 TIMx_CR1 寄存器的 UDIS 位可以禁止 UEV 事件。这样可以避免向预装载寄存器中写入新值时更新影子寄存器。因此 UDIS 位被清为’0’之前不会产生更新事件。然而，计数器仍会从当前自动加载值重新开始计数，同时预分频器的计数器重新从 0 开始(但预分频系数不变)。

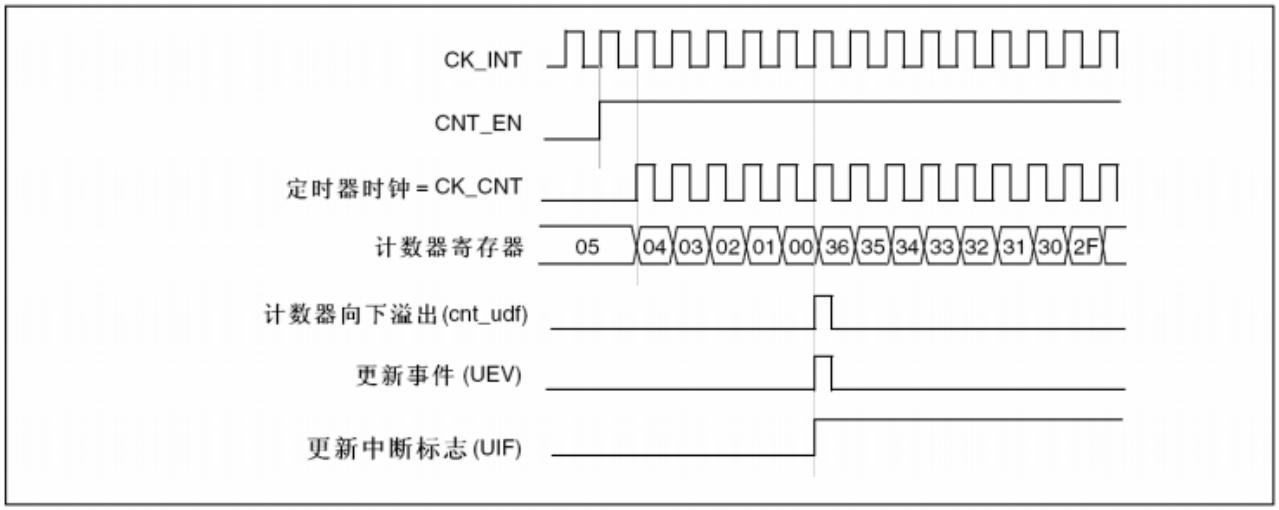
此外，如果设置了 TIMx_CR1 寄存器中的 URS 位(选择更新请求) ，设置 UG 位将产生一个更新事件 UEV 但不设置 UIF 标志(因此不产生中断和 DMA 请求)，这是为了避免在发生捕获事件并清除计数器时，同时产生更新和捕获中断。

当发生更新事件时，所有的寄存器都被更新，并且(根据 URS 位的设置)更新标志位(TIMx_SR 寄存器中的 UIF 位)也被设置。

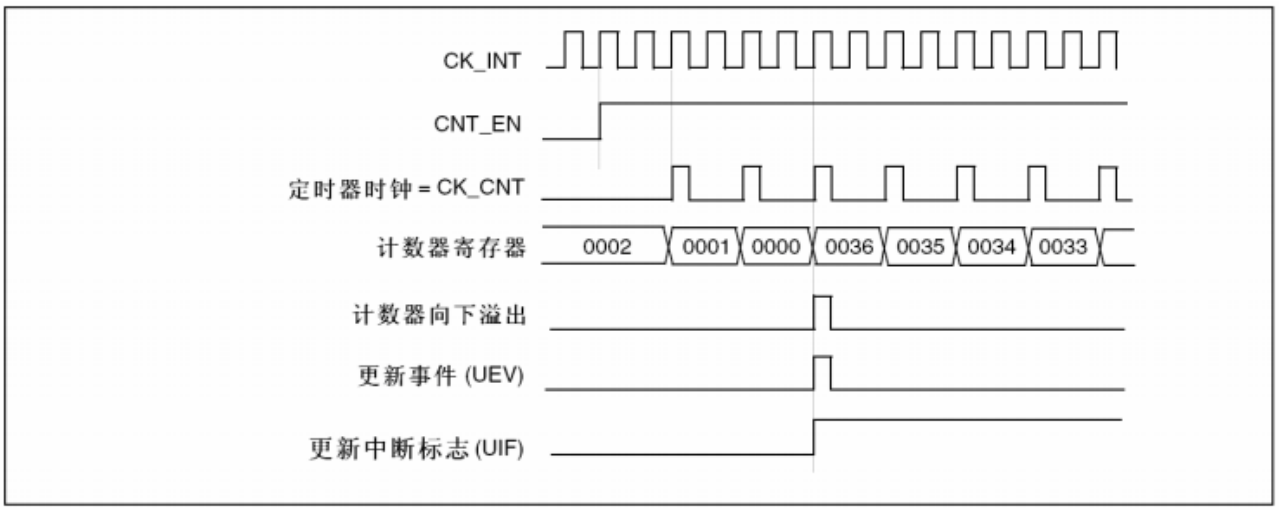
- 预分频器的缓存器被置入预装载寄存器的值(TIMx_PSC 寄存器的值)。
- 当前的自动加载寄存器被更新为预装载值(TIMx_ARR 寄存器中的内容)。注：自动装载在计数器重载入之前被更新，因此下一个周期将是预期的值。

以下是一些当 TIMx_ARR=0x36 时，计数器在不同时钟频率下的操作例子。

计数器时序图，内部时钟分频因子为 1



计数器时序图，内部时钟分频因子为 2



控制寄存器 1(TIMx_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:10	保留，始终读为0。
位9:8	**CKD[1:0]**: 时钟分频因子 (Clock division) 定义在定时器时钟(CK_INT)频率与数字滤波器(ETR，TIx)使用的采样频率之间的分频比例。 00：$t_{DTS} = t_{CK\_INT}$ 01：$t_{DTS} = 2 \times t_{CK\_INT}$ 10：$t_{DTS} = 4 \times t_{CK\_INT}$ 11：保留
位7	**ARPE**：自动重装载预装载允许位 (Auto-reload preload enable) 0：TIMx_ARR寄存器没有缓冲； 1：TIMx_ARR寄存器被装入缓冲器。
位6:5	**CMS[1:0]**：选择中央对齐模式 (Center-aligned mode selection) 00：边沿对齐模式。计数器依据方向位(DIR)向上或向下计数。 01：中央对齐模式1。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，只在计数器向下计数时被设置。 10：中央对齐模式2。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，只在计数器向上计数时被设置。 11：中央对齐模式3。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，在计数器向上和向下计数时均被设置。 注：在计数器开启时(CEN=1)，不允许从边沿对齐模式转换到中央对齐模式。
位4	**DIR**：方向 (Direction) 0：计数器向上计数； 1：计数器向下计数。 注：当计数器配置为中央对齐模式或编码器模式时，该位为只读。
位3	**OPM**：单脉冲模式 (One pulse mode) 0：在发生更新事件时，计数器不停止； 1：在发生下一次更新事件(清除CEN位)时，计数器停止。
位2	**URS**：更新请求源 (Update request source) 软件通过该位选择UEV事件的源 0：如果使能了更新中断或DMA请求，则下述任一事件产生更新中断或DMA请求： – 计数器溢出/下溢 – 设置UG位 – 从模式控制器产生的更新 1：如果使能了更新中断或DMA请求，则只有计数器溢出/下溢才产生更新中断或DMA请求。

控制寄存器 2(TIMx_CR2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								TI1S	MMS[2:0]			CCDS	保留		
								rw	rw	rw	rw	rw			

位15:8	保留，始终读为0。
位7	**TI1S**：TI1选择 (TI1 selection) 0：TIMx_CH1引脚连到TI1输入； 1：TIMx_CH1、TIMx_CH2和TIMx_CH3引脚经异或后连到TI1输入。 见上一章13.3.18的与霍尔传感器的接口一节。
位6:4	**MMS[2:0]**：主模式选择 (Master mode selection) 这3位用于选择在主模式下送到从定时器的同步信息(TRGO)。可能的组合如下： 000：**复位** – TIMx_EGR寄存器的UG位被用于作为触发输出(TRGO)。如果是触发输入产生的复位(从模式控制器处于复位模式)，则TRGO上的信号相对实际的复位会有一个延迟。 001：**使能** – 计数器使能信号CNT_EN被用于作为触发输出(TRGO)。有时需要在同一时间启动多个定时器或控制在一段时间内使能从定时器。计数器使能信号是通过CEN控制位和门控模式下的触发输入信号的逻辑或产生。 当计数器使能信号受控于触发输入时，TRGO上会有一个延迟，除非选择了主/从模式(见TIMx_SMCR寄存器中MSM位的描述)。 010：**更新** – 更新事件被选为触发输入(TRGO)。例如，一个主定时器的时钟可以被用作一个从定时器的预分频器。 011：**比较脉冲** – 在发生一次捕获或一次比较成功时，当要设置CC1IF标志时(即使它已经为高)，触发输出送出一个正脉冲(TRGO)。 100：**比较** – OC1REF信号被用于作为触发输出(TRGO)。 101：**比较** – OC2REF信号被用于作为触发输出(TRGO)。 110：**比较** – OC3REF信号被用于作为触发输出(TRGO)。 111：**比较** – OC4REF信号被用于作为触发输出(TRGO)。
位3	**CCDS**：捕获/比较的DMA选择 0：当发生CCx事件时，送出CCx的DMA请求； 1：当发生更新事件时，送出CCx的DMA请求。
位2:0	保留，始终读为0。

DMA/中断使能寄存器(TIMx_DIER)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	TDE	保留	CC4DE	CC3DE	CC2DE	CC1DE	UDE	保留	TIE	保留	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

位15	保留，始终读为0。
位14	**TDE**：允许触发DMA请求 (Trigger DMA request enable) 0：禁止触发DMA请求； 1：允许触发DMA请求。
位13	保留，始终读为0。
位12	**CC4DE**：允许捕获/比较4的DMA请求 (Capture/Compare 4 DMA request enable) 0：禁止捕获/比较4的DMA请求； 1：允许捕获/比较4的DMA请求。

位11	**CC3DE**：允许捕获/比较3的DMA请求 (Capture/Compare 3 DMA request enable) 0：禁止捕获/比较3的DMA请求； 1：允许捕获/比较3的DMA请求。
位10	**CC2DE**：允许捕获/比较2的DMA请求 (Capture/Compare 2 DMA request enable) 0：禁止捕获/比较2的DMA请求； 1：允许捕获/比较2的DMA请求。
位9	**CC1DE**：允许捕获/比较1的DMA请求 (Capture/Compare 1 DMA request enable) 0：禁止捕获/比较1的DMA请求； 1：允许捕获/比较1的DMA请求。
位8	**UDE**：允许更新的DMA请求 (Update DMA request enable) 0：禁止更新的DMA请求； 1：允许更新的DMA请求。
位7	保留，始终读为0。
位6	**TIE**：触发中断使能 (Trigger interrupt enable) 0：禁止触发中断； 1：使能触发中断。
位5	保留，始终读为0。
位4	**CC4IE**：允许捕获/比较4中断 (Capture/Compare 4 interrupt enable) 0：禁止捕获/比较4中断； 1：允许捕获/比较4中断。
位3	**CC3IE**：允许捕获/比较3中断 (Capture/Compare 3 interrupt enable) 0：禁止捕获/比较3中断； 1：允许捕获/比较3中断。
位2	**CC2IE**：允许捕获/比较2中断 (Capture/Compare 2 interrupt enable) 0：禁止捕获/比较2中断； 1：允许捕获/比较2中断。
位1	**CC1IE**：允许捕获/比较1中断 (Capture/Compare 1 interrupt enable) 0：禁止捕获/比较1中断； 1：允许捕获/比较1中断。
位0	**UIE**：允许更新中断 (Update interrupt enable) 0：禁止更新中断； 1：允许更新中断。

计数器(TIMx_CNT)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**CNT[15:0]**：计数器的值 (Counter value)

预分频器(TIMx_PSC)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**PSC[15:0]**：预分频器的值 (Prescaler value) 计数器的时钟频率CK_CNT等于f_{CK_PSC}/(PSC[15:0]+1)。 PSC包含了当更新事件产生时装入当前预分频器寄存器的值。

自动重装载寄存器(TIMx_ARR)

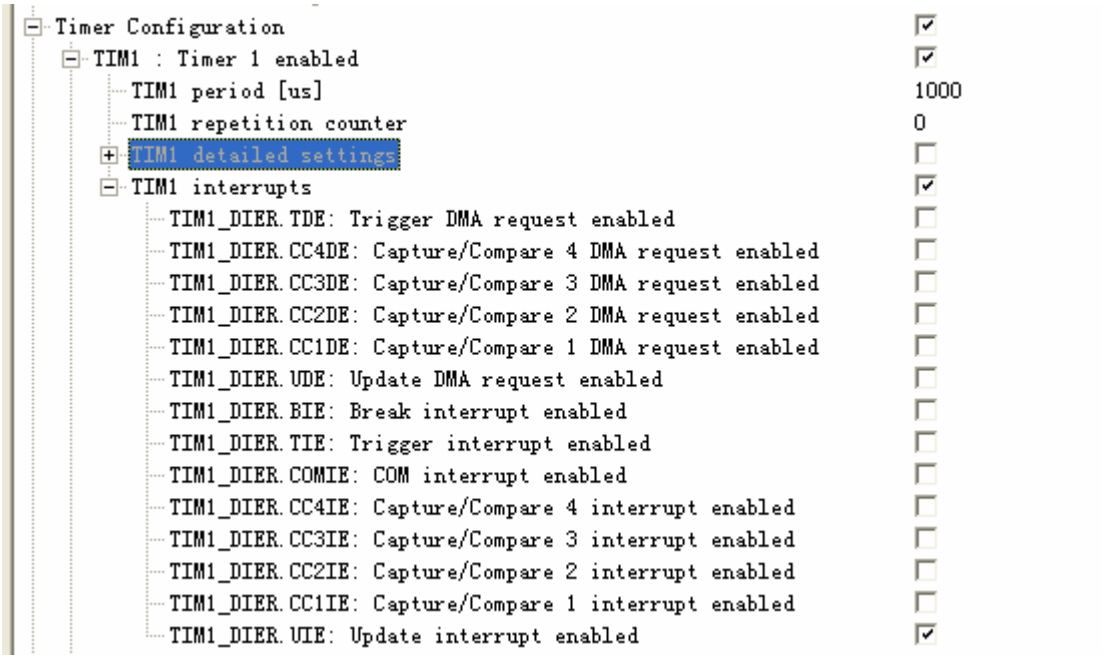
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**ARR[15:0]**: 自动重装载的值 (Auto reload value) ARR包含了将要传送至实际的自动重装载寄存器的数值。 详细参考14.3.1节：有关ARR的更新和动作。 当自动重装载的值为空时，计数器不工作。

因为通用寄存器比较多，功能复杂，更详细请参考《STM32F10x 微控制器参考手册(2009 年 12 月第 10 版)》

BHS-STM32 实验 9-通用定时器Timer(直接操作寄存器)

本例子在实验 1 基础上修改，使用通用定时器 1 定时，定时时间 1ms 使用硬件定时器延时循环点亮/熄灭 LED



在中断函数中实现延时功能，其他一样

```
//LED 循环闪烁
void LedFlash(void)
{
  static u16 leds = 0x01;
  u32 temp;

    //先读出 PC 端口状态
    temp = GPIOC->ODR;
```

tyw藏书

```
    //先屏蔽掉 PC8~PC11
    temp |= 0x00000F00;

    //重新设置 PC8~PC11 输出状态，IO 输出低电平点亮 LED
    GPIOC->ODR = temp&(~(leds<<8));
    leds <<= 1;
    if ( (leds&0x0f) == 0)
        leds = 0x01;
}

//精确的延时
void Delay(u32 nTime)
{u32 counter;

  counter=gTimer_1ms;
  while( gTimer_1ms-counter < nTime);//定时时间到才退出

}

/*---------------------------------------------------------------------------
  Timer1  中断函数
  1 ms 中断 1 次
 *---------------------------------------------------------------------------*/
void TIM1_UP_IRQHandler (void) {

  if ((TIM1->SR & 0x0001) != 0) {                  // check interrupt source

    gTimer_1ms++;
    // clear UIF flag   清除中断标志
    TIM1->SR &= ~(1<<0);
  }
} // end TIM1_UP_IRQHandler

int main (void)
{
  stm32_Init ();// STM32  初始化

  //关闭所有 LED
  GPIOC->ODR |= 0x00000F00;
  Delay(20);

  while (1)
  {
     Delay(50);
```

```
        //循环显示 1 位 LED
        LedFlash();

        Delay(50);

        //关闭所有 LED
        GPIOC->ODR |= 0x00000F00;
    }
}
```

BHS-STM32 实验 10-通用定时器Timer(库函数)

使用通用定时器 1 定时，定时时间 1ms,与上例功能一致，使用硬件定时器延时循环点亮/熄灭 LED

函数 TIM_DeInit

函数名	TIM_DeInit
函数原形	void TIM_DeInit(TIM_TypeDef* TIMx)
功能描述	将外设 TIMx 寄存器重设为缺省值
输入参数	TIMx：x 可以是 2，3 或者 4，来选择 TIM 外设
输出参数	无
返回值	无
先决条件	无
被调用函数	RCC_APB1PeriphClockCmd().

函数 TIM_TimeBaseInit

函数名	TIM_TimeBaseInit
函数原形	void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct)
功能描述	根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时间基数单位
输入参数 1	TIMx：x 可以是 2，3 或者 4，来选择 TIM 外设
输入参数 2	TIMTimeBase_InitStruct：指向结构 TIM_TimeBaseInitTypeDef 的指针，包含了 TIMx 时间基数单位的配置信息 参阅 Section：TIM_TimeBaseInitTypeDef 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

TIM_TimeBaseInitTypeDef structure

TIM_TimeBaseInitTypeDef 定义于文件“stm32f10x_tim.h”：

```
typedef struct
{
u16 TIM_Period;
u16 TIM_Prescaler;
u8 TIM_ClockDivision;
u16 TIM_CounterMode;
} TIM_TimeBaseInitTypeDef;
```

TIM_Period

TIM_Period 设置了在下一个更新事件装入活动的自动重装载寄存器周期的值。它的取值必须在 0x0000

和

0xFFFF 之间。

tyw藏书

TIM_Prescaler

TIM_Prescaler 设置了用来作为 TIMx 时钟频率除数的预分频值。它的取值必须在 0x0000 和 0xFFFF 之间。

TIM_ClockDivision

TIM_ClockDivision 设置了时钟分割。

该参数取值见下表 TIM_ClockDivision 值。

TIM_ClockDivision	描述
TIM_CKD_DIV1	TDTS = Tck_tim
TIM_CKD_DIV2	TDTS = 2Tck_tim
TIM_CKD_DIV4	TDTS = 4Tck_tim

TIM_CounterMode 值

TIM_CounterMode	描述
TIM_CounterMode_Up	TIM 向上计数模式
TIM_CounterMode_Down	TIM 向下计数模式
TIM_CounterMode_CenterAligned1	TIM 中央对齐模式 1 计数模式
TIM_CounterMode_CenterAligned2	TIM 中央对齐模式 2 计数模式
TIM_CounterMode_CenterAligned3	TIM 中央对齐模式 3 计数模式

函数 TIM_PrescalerConfig

函数名	TIM_PrescalerConfig
函数原形	void TIM_PrescalerConfig(TIM_TypeDef* TIMx, u16 Prescaler,u16 TIM_PSCReloadMode)
功能描述	设置 TIMx 预分频
输入参数 1	TIMx：x 可以是 2，3 或者 4，来选择 TIM 外设
输入参数 2	TIM_PSCReloadMode：预分频重载模式 参阅 Section：TIM_PSCReloadMode 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

TIM_PSCReloadMode 选择预分频重载模式

TIM_PSCReloadMode	描述
TIM_PSCReloadMode_Update	TIM 预分频值在更新事件装入
TIM_PSCReloadMode_Immediate	TIM 预分频值即时装入

函数 TIM _ITConfig

函数名	TIM_ITConfig
函数原形	void TIM_ITConfig(TIM_TypeDef* TIMx, u16 TIM_IT, FunctionalState NewState)
功能描述	使能或者失能指定的 TIM 中断
输入参数 1	TIMx：x 可以是 2，3 或者 4，来选择 TIM 外设
输入参数 2	TIM_IT：待使能或者失能的 TIM 中断源 参阅 Section：TIM_IT 查阅更多该参数允许取值范围
输入参数 3	NewState：TIMx 中断的新状态 这个参数可以取：ENABLE 或者 DISABLE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

TIM_IT

输入参数 TIM_IT 使能或者失能 TIM 的中断。可以取下表的一个或者多个取值的组合作为该参数的值。

TIM_IT	描述
TIM_IT_Update	TIM 中断源
TIM_IT_CC1	TIM 捕获/比较 1 中断源
TIM_IT_CC2	TIM 捕获/比较 2 中断源
TIM_IT_CC3	TIM 捕获/比较 3 中断源
TIM_IT_CC4	TIM 捕获/比较 4 中断源
TIM_IT_Trigger	TIM 触发中断源

```
/*------------------------------------------------------------------------
   中断函数
   1 ms 中断 1 次
 *------------------------------------------------------------------------*/
void TIM2_IRQHandler(void)
{

    if(TIM_GetITStatus(TIM2,TIM_IT_Update)!=RESET)
    {
        //清除中断标志
        TIM_ClearITPendingBit(TIM2,TIM_IT_Update);
        gTimer_1ms++;
    }
}

void Timer2_Configuration(void)
{
 TIM_TimeBaseInitTypeDef     TIM_TimeBaseStructure;
 NVIC_InitTypeDef NVIC_InitStructure;

   //-------------------------------------------------------------
   //打开定时器的时钟
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
```

```
//-------------------------------------------------------------

//TIMx 寄存器重设为缺省值
TIM_DeInit(TIM2);

TIM_TimeBaseStructure.TIM_Period=1;          //自动重装载寄存器周期的值
TIM_TimeBaseStructure.TIM_Prescaler=0;   //TIMx  时钟频率除数的预分频值
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1; //采样分频
TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

TIM_PrescalerConfig(TIM2,36000-1,TIM_PSCReloadMode_Immediate);//时钟分频系数，定时器 1ms
TIM_ARRPreloadConfig(TIM2, DISABLE);//禁止 ARR 预装载缓冲器
TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE);//定时器更新中断允许

TIM_Cmd(TIM2, ENABLE);   //开启时钟

//-------------------------------------------------------------
//配置 TIM2 中断
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
}
```

中断实验

中断功能描述

特性

- 68 个可屏蔽中断通道(不包含 16 个 Cortex™-M3 的中断线)；
- 16 个可编程的优先等级(使用了 4 位中断优先级)；
- 低延迟的异常和中断处理；
- 电源管理控制；
- 系统控制寄存器的实现；

嵌套向量中断控制器(NVIC)和处理器核的接口紧密相连，可以实现低延迟的中断处理和高效地处理晚到的中断。

嵌套向量中断控制器管理着包括内核异常等中断。更多关于异常和 NVIC 编程的说明请参考《STM32F10xxx Cortex-M3 编程手册》。

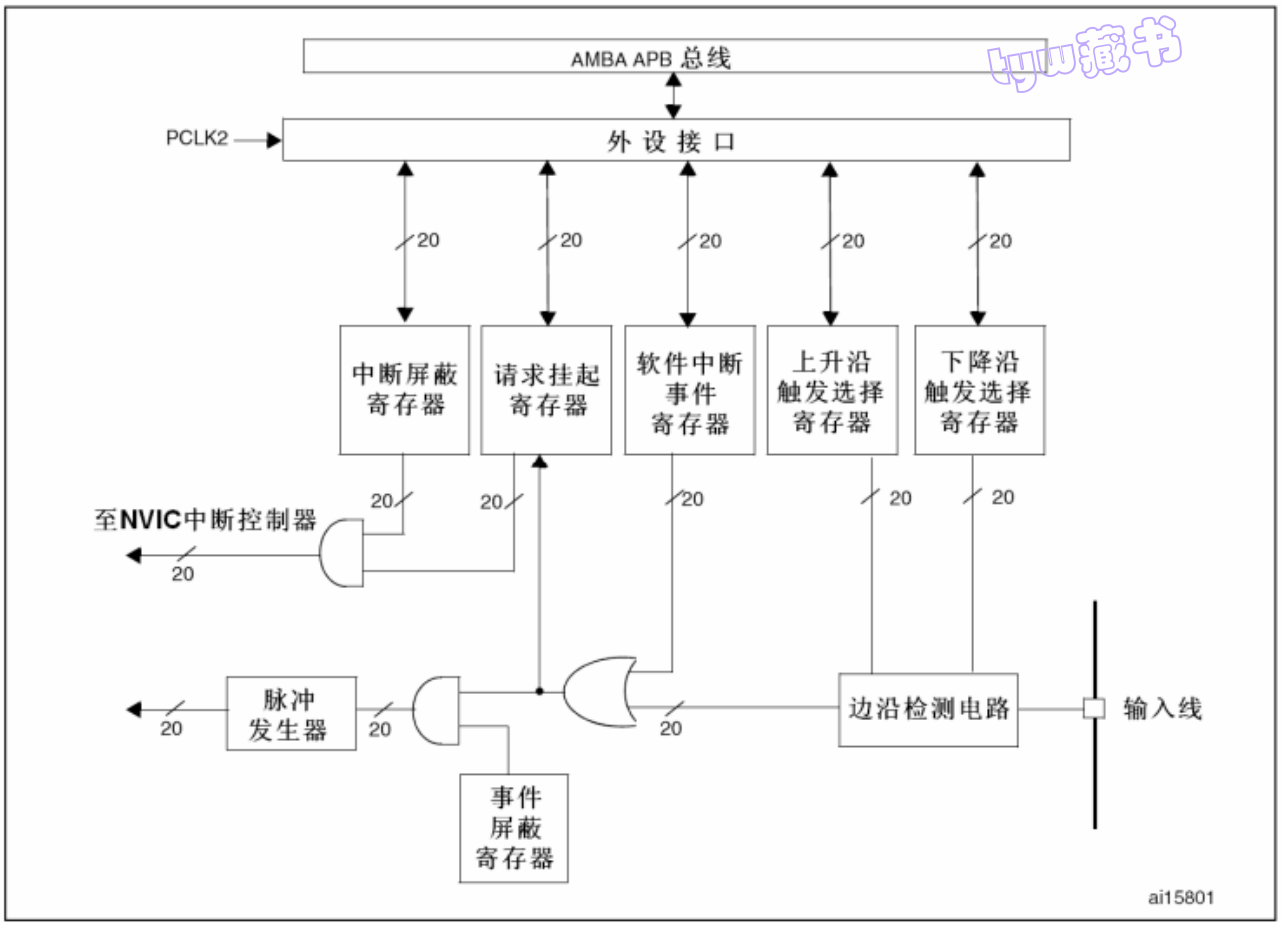
中断和异常向量

STM32F10xxx 产品(小容量、中容量和大容量)的向量表

位置	优先级	优先级类型	名称	说明	地址
	-	-	-	保留	0x0000_0000
	-3	固定	Reset	复位	0x0000_0004
	-2	固定	NMI	不可屏蔽中断 RCC时钟安全系统(CSS)联接到NMI向量	0x0000_0008
	-1	固定	硬件失效(HardFault)	所有类型的失效	0x0000_000C
	0	可设置	存储管理(MemManage)	存储器管理	0x0000_0010
	1	可设置	总线错误(BusFault)	预取指失败，存储器访问失败	0x0000_0014
	2	可设置	错误应用(UsageFault)	未定义的指令或非法状态	0x0000_0018
	-	-	-	保留	0x0000_001C ~0x0000_002B
	3	可设置	SVCall	通过SWI指令的系统服务调用	0x0000_002C
	4	可设置	调试监控(DebugMonitor)	调试监控器	0x0000_0030
	-	-	-	保留	0x0000_0034
	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C
0	7	可设置	WWDG	窗口定时器中断	0x0000_0040
1	8	可设置	PVD	连到EXTI的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050
5	12	可设置	RCC	复位和时钟控制(RCC)中断	0x0000_0054
6	13	可设置	EXTI0	EXTI线0中断	0x0000_0058
7	14	可设置	EXTI1	EXTI线1中断	0x0000_005C
8	15	可设置	EXTI2	EXTI线2中断	0x0000_0060
9	16	可设置	EXTI3	EXTI线3中断	0x0000_0064
10	17	可设置	EXTI4	EXTI线4中断	0x0000_0068
11	18	可设置	DMA1通道1	DMA1通道1全局中断	0x0000_006C
12	19	可设置	DMA1通道2	DMA1通道2全局中断	0x0000_0070
13	20	可设置	DMA1通道3	DMA1通道3全局中断	0x0000_0074
14	21	可设置	DMA1通道4	DMA1通道4全局中断	0x0000_0078
15	22	可设置	DMA1通道5	DMA1通道5全局中断	0x0000_007C
16	23	可设置	DMA1通道6	DMA1通道6全局中断	0x0000_0080
17	24	可设置	DMA1通道7	DMA1通道7全局中断	0x0000_0084
18	25	可设置	ADC1_2	ADC1和ADC2的全局中断	0x0000_0088
19	26	可设置	USB_HP_CAN_TX	USB高优先级或CAN发送中断	0x0000_008C
20	27	可设置	USB_LP_CAN_RX0	USB低优先级或CAN接收0中断	0x0000_0090

21	28	可设置	CAN_RX1	CAN接收1中断	0x0000_0094
22	29	可设置	CAN_SCE	CAN SCE中断	0x0000_0098
23	30	可设置	EXTI9_5	EXTI线[9:5]中断	0x0000_009C
24	31	可设置	TIM1_BRK	TIM1刹车中断	0x0000_00A0
25	32	可设置	TIM1_UP	TIM1更新中断	0x0000_00A4
26	33	可设置	TIM1_TRG_COM	TIM1触发和通信中断	0x0000_00A8
27	34	可设置	TIM1_CC	TIM1捕获比较中断	0x0000_00AC
28	35	可设置	TIM2	TIM2全局中断	0x0000_00B0
29	36	可设置	TIM3	TIM3全局中断	0x0000_00B4
30	37	可设置	TIM4	TIM4全局中断	0x0000_00B8
31	38	可设置	I2C1_EV	I^2C1事件中断	0x0000_00BC
32	39	可设置	I2C1_ER	I^2C1错误中断	0x0000_00C0
33	40	可设置	I2C2_EV	I^2C2事件中断	0x0000_00C4
34	41	可设置	I2C2_ER	I^2C2错误中断	0x0000_00C8
35	42	可设置	SPI1	SPI1全局中断	0x0000_00CC
36	43	可设置	SPI2	SPI2全局中断	0x0000_00D0
37	44	可设置	USART1	USART1全局中断	0x0000_00D4
38	45	可设置	USART2	USART2全局中断	0x0000_00D8
39	46	可设置	USART3	USART3全局中断	0x0000_00DC
40	47	可设置	EXTI15_10	EXTI线[15:10]中断	0x0000_00E0
41	48	可设置	RTCAlarm	连到EXTI的RTC闹钟中断	0x0000_00E4
42	49	可设置	USB唤醒	连到EXTI的从USB待机唤醒中断	0x0000_00E8
43	50	可设置	TIM8_BRK	TIM8刹车中断	0x0000_00EC
44	51	可设置	TIM8_UP	TIM8更新中断	0x0000_00F0
45	52	可设置	TIM8_TRG_COM	TIM8触发和通信中断	0x0000_00F4
46	53	可设置	TIM8_CC	TIM8捕获比较中断	0x0000_00F8
47	54	可设置	ADC3	ADC3全局中断	0x0000_00FC
48	55	可设置	FSMC	FSMC全局中断	0x0000_0100
50	57	可设置	TIM5	TIM5全局中断	0x0000_0108
51	58	可设置	SPI3	SPI3全局中断	0x0000_010C
52	59	可设置	UART4	UART4全局中断	0x0000_0110
53	60	可设置	UART5	UART5全局中断	0x0000_0114
54	61	可设置	TIM6	TIM6全局中断	0x0000_0118
55	62	可设置	TIM7	TIM7全局中断	0x0000_011C
56	63	可设置	DMA2通道1	DMA2通道1全局中断	0x0000_0120
57	64	可设置	DMA2通道2	DMA2通道2全局中断	0x0000_0124
58	65	可设置	DMA2通道3	DMA2通道3全局中断	0x0000_0128
59	66	可设置	DMA2通道4_5	DMA2通道4和DMA2通道5全局中断	0x0000_012C

外部中断/事件控制器框图



功能说明

要产生中断，必须先配置好并使能中断线。根据需要的边沿检测设置 2 个触发寄存器，同时在中断屏蔽寄存器的相应位写’1’允许中断请求。当外部中断线上发生了期待的边沿时，将产生一个中断请求，对应的挂起位也随之被置’1’。在挂起寄存器的对应位写’1’，将清除该中断请求。

如果需要产生事件，必须先配置好并使能事件线。根据需要的边沿检测通过设置 2 个触发寄存器，同时在事件屏蔽寄存器的相应位写’1’允许事件请求。当事件线上发生了需要的边沿时，将产生一个事件请求脉冲，对应的挂起位不被置’1’。

通过在软件中断/事件寄存器写’1’，也可以通过软件产生中断/事件请求。

硬件中断选择

通过下面的过程来配置 20 个线路做为中断源：

- 配置 20 个中断线的屏蔽位(EXTI_IMR)
- 配置所选中断线的触发选择位(EXTI_RTSR 和 EXTI_FTSR)；
- 配置对应到外部中断控制器(EXTI)的 NVIC 中断通道的使能和屏蔽位

请求可以被正确地响应。

硬件事件选择

通过下面的过程，可以配置 20 个线路为事件源

- 配置 20 个事件线的屏蔽位(EXTI_EMR)
- 配置事件线的触发选择位(EXTI_RTSR 和 EXTI_FTSR)

软件中断/事件的选择

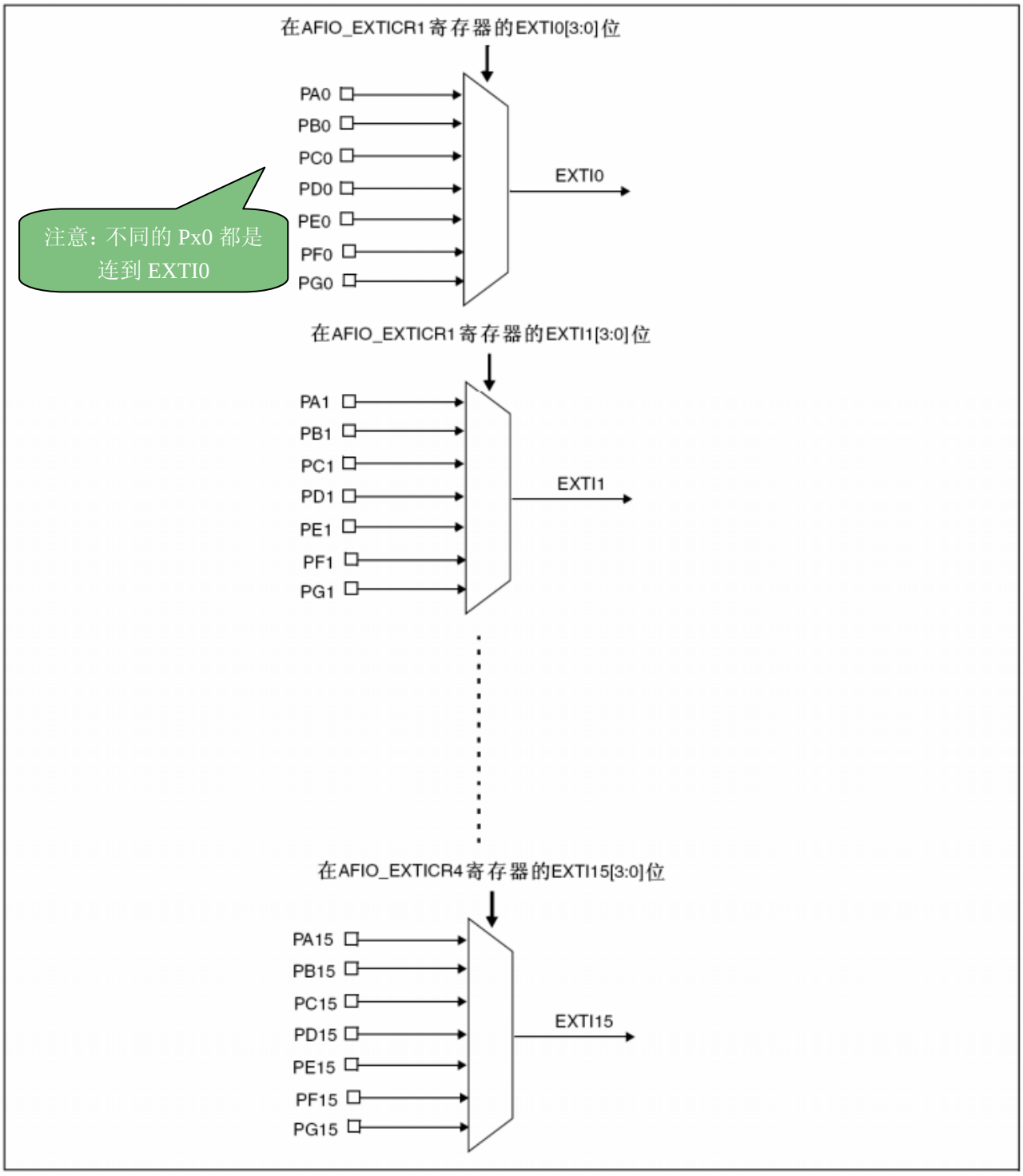
20 个线路可以被配置成软件中断/事件线。下面是产生软件中断的过程：

- 配置 20 个中断/事件线屏蔽位(EXTI_IMR, EXTI_EMR)
- 设置软件中断寄存器的请求位(EXTI_SWIER)

外部中断/事件线路映像

通用 I/O 端口以下图的方式连接到 16 个外部中断/事件线上：

外部中断通用 I/O 映像



另外四个 EXTI 线的连接方式如下：

- EXTI 线 16 连接到 PVD 输出
- EXTI 线 17 连接到 RTC 闹钟事件
- EXTI 线 18 连接到 USB 唤醒事件
- EXTI 线 19 连接到以太网唤醒事件(只适用于互联型产品)

EXTI 寄存器描述

中断屏蔽寄存器(EXTI_IMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												MR19	MR18	MR17	MR16
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:20	保留，必须始终保持为复位状态(0)。
位19:0	**MRx**: 线x上的中断屏蔽 (Interrupt Mask on line x) 0：屏蔽来自线x上的中断请求； 1：开放来自线x上的中断请求。 注：位19只适用于互联型产品，对于其它产品为保留位。

事件屏蔽寄存器(EXTI_EMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												MR19	MR18	MR17	MR16
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:20	保留，必须始终保持为复位状态(0)。
位19:0	**MRx**: 线x上的事件屏蔽 (Event Mask on line x) 0：屏蔽来自线x上的事件请求； 1：开放来自线x上的事件请求。 注：位19只适用于互联型产品，对于其它产品为保留位。

上升沿触发选择寄存器(EXTI_RTSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												TR19	TR18	TR17	TR16
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**TRx**: 线x上的上升沿触发事件配置位 (Rising trigger event configuration bit of line x) 0：禁止输入线x上的上升沿触发(中断和事件) 1：允许输入线x上的上升沿触发(中断和事件) 注：位19只适用于互联型产品，对于其它产品为保留位。

注意： 外部唤醒线是边沿触发的，这些线上不能出现毛刺信号。

在写 **EXTI_RTSR** 寄存器时，在外部中断线上的上升沿信号不能被识别，挂起位也不会被置位。

在同一中断线上，可以同时设置上升沿和下降沿触发。即任一边沿都可触发中断。

下降沿触发选择寄存器(EXTI_FTSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												TR19	TR18	TR17	TR16
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**TRx**: 线x上的下降沿触发事件配置位 (Falling trigger event configuration bit of line x) 0：禁止输入线x上的下降沿触发(中断和事件) 1：允许输入线x上的下降沿触发(中断和事件) 注：位19只适用于互联型产品，对于其它产品为保留位。

软件中断事件寄存器(EXTI_SWIER)

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

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

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**SWIERx**: 线x上的软件中断 (Software interrupt on line x) 当该位为'0'时，写'1'将设置EXTI_PR中相应的挂起位。如果在EXTI_IMR和EXTI_EMR中允许产生该中断，则此时将产生一个中断。 注：通过清除EXTI_PR的对应位(写入'1')，可以清除该位为'0'。 注：位19只适用于互联型产品，对于其它产品为保留位。

挂起寄存器(EXTI_PR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												PR19	PR18	PR17	PR16
												rc w1	rc w1	rc w1	rc w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1

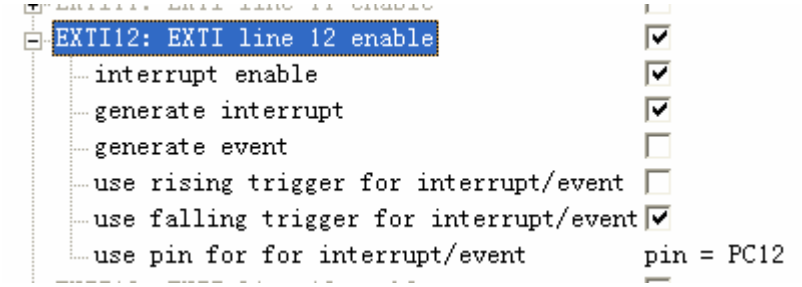
位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**PRx**: 挂起位 (Pending bit) 0：没有发生触发请求 1：发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件，该位被置'1'。在该位中写入'1'可以清除它，也可以通过改变边沿检测的极性清除。 注：位19只适用于互联型产品，对于其它产品为保留位。

BHS-STM32 实验 11-EXTI外部中断(直接操作寄存器)

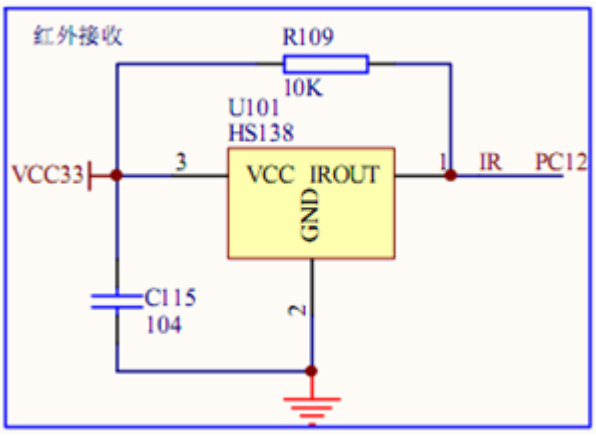
tyw藏书

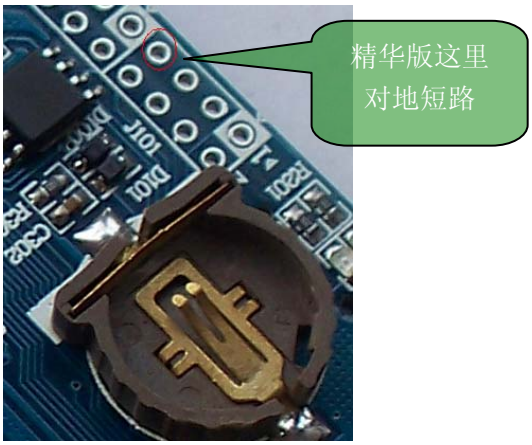
本例子实现外部中断功能

中断配置如下




```
/*---------------------------------------------------------------------------
   EXTI15..10 Interrupt Handler
   外部中断 15~10 使用共享中断向量
 *---------------------------------------------------------------------------*/
void EXTI15_10_IRQHandler(void)
{
   if (EXTI->PR & (1<<12)) {                          // EXTI12 interrupt pending?/是否是外部中断 12
      if ((ledExti ^=1) == 0)
         GPIOC->ODR &= ~(1 << (ledPosExti+8));         // switch on LED //点亮 LED
      else
         GPIOC->ODR |=   (1 << (ledPosExti+8));        // switch off LED    //熄灭 LED

      EXTI->PR |= (1<<12);                             // clear pending interrupt/清除中断标志
   }
}
```





例子使用 PC12,由于 PC12 也连接红外接收，所以用红外遥控器对着接收头按下按键将看到 LED2 闪烁，没有红外接收的，自己用手将 PC12 对地短路也行，没有红外接收的，自己用手将 PC12 对地短路也行,上图红圈中的就是 PC12（BHS-STM32-V 才有红外接收）

注意：不少人系统怎么知道 EXTI15_10_IRQHandler 就是中断函数，中断函数名可以任意修改吗？

在项目中大都有个.s 文件，这里是 STM32F10x.s，我们来看看他里面有上面内容：

我们可以在文件中看到如下内容

; Vector Table Mapped to Address 0 at Reset

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     DebugMon_Handler          ; Debug Monitor Handler
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler           ; Window Watchdog
                DCD     PVD_IRQHandler            ; PVD through EXTI Line detect
                DCD     TAMPER_IRQHandler         ; Tamper
                DCD     RTC_IRQHandler            ; RTC
                DCD     FLASH_IRQHandler          ; Flash
                DCD     RCC_IRQHandler            ; RCC
                DCD     EXTI0_IRQHandler          ; EXTI Line 0
                DCD     EXTI1_IRQHandler          ; EXTI Line 1
                DCD     EXTI2_IRQHandler          ; EXTI Line 2
                DCD     EXTI3_IRQHandler          ; EXTI Line 3
                DCD     EXTI4_IRQHandler          ; EXTI Line 4
                DCD     DMAChannel1_IRQHandler    ; DMA Channel 1
                DCD     DMAChannel2_IRQHandler    ; DMA Channel 2
                DCD     DMAChannel3_IRQHandler    ; DMA Channel 3
                DCD     DMAChannel4_IRQHandler    ; DMA Channel 4
                DCD     DMAChannel5_IRQHandler    ; DMA Channel 5
                DCD     DMAChannel6_IRQHandler    ; DMA Channel 6
                DCD     DMAChannel7_IRQHandler    ; DMA Channel 7
                DCD     ADC_IRQHandler            ; ADC
                DCD     USB_HP_CAN_TX_IRQHandler  ; USB High Priority or CAN TX
                DCD     USB_LP_CAN_RX0_IRQHandler ; USB Low  Priority or CAN RX0
                DCD     CAN_RX1_IRQHandler        ; CAN RX1
                DCD     CAN_SCE_IRQHandler        ; CAN SCE
                DCD     EXTI9_5_IRQHandler        ; EXTI Line 9..5
```

```
DCD     TIM1_BRK_IRQHandler          ; TIM1 Break
DCD     TIM1_UP_IRQHandler           ; TIM1 Update
DCD     TIM1_TRG_COM_IRQHandler      ; TIM1 Trigger and Commutation
DCD     TIM1_CC_IRQHandler           ; TIM1 Capture Compare
DCD     TIM2_IRQHandler              ; TIM2
DCD     TIM3_IRQHandler              ; TIM3
DCD     TIM4_IRQHandler              ; TIM4
DCD     I2C1_EV_IRQHandler           ; I2C1 Event
DCD     I2C1_ER_IRQHandler           ; I2C1 Error
DCD     I2C2_EV_IRQHandler           ; I2C2 Event
DCD     I2C2_ER_IRQHandler           ; I2C2 Error
DCD     SPI1_IRQHandler              ; SPI1
DCD     SPI2_IRQHandler              ; SPI2
DCD     USART1_IRQHandler            ; USART1
DCD     USART2_IRQHandler            ; USART2
DCD     USART3_IRQHandler            ; USART3
DCD     EXTI15_10_IRQHandler         ; EXTI Line 15..10
DCD     RTCAlarm_IRQHandler          ; RTC Alarm through EXTI Line
DCD     USBWakeUp_IRQHandler         ; USB Wakeup from suspend


AREA    |.text|, CODE, READONLY
```

对照前面的中断向量表，我们明白这里定义的就是中断向量表，这里的实际就是中断函数，只要你修改本文件中断函数名，你就可以使用自己中断函数名了，不过一般都用默认的就挺好的。

BHS-STM32 实验 12-EXTI外部中断(库函数)

本例子实现外部中断功能

函数 GPIO_EXTILineConfig

函数名	GPIO_EXTILineConfig
函数原形	void GPIO_EXTILineConfig(u8 GPIO_PortSource, u8 GPIO_PinSource)
功能描述	选择 GPIO 管脚用作外部中断线路
输入参数 1	GPIO_PortSource：选择用作外部中断线源的 GPIO 端口 参阅 Section：GPIO_PortSource 查阅更多该参数允许取值范围
输入参数 2	GPIO_PinSource：待设置的外部中断线路 该参数可以取 GPIO_PinSourcex(x 可以是 0-15)
输出参数	无
返回值	无
先决条件	无
被调用函数	无

外部中断配置

```
void Exit_Configuration(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  EXTI_InitTypeDef EXTI_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;
```

```
  //外部中断使用的 RCC 时钟
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC|RCC_APB2Periph_AFIO, ENABLE);//使能 GPIO 时钟

  //外部中断使用的 GPIO 配置
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  /* Connect EXTI Line12 to PC.12 */
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOC, GPIO_PinSource12);

  /* Configure EXTI Line12 to generate an interrupt on falling edge */
  //配置外部中断
  EXTI_InitStructure.EXTI_Line = EXTI_Line12;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  /* Generate software interrupt: simulate a falling edge applied on EXTI line 12 */
  //这里是软件模拟产生中断
  EXTI_GenerateSWInterrupt(EXTI_Line12);

  if(EXTI_GetITStatus(EXTI_Line12) != RESET)
  {
    /* Clear the EXTI line 12 pending bit */
    EXTI_ClearITPendingBit(EXTI_Line12);
  }

  //外部中断使用的 NVIC 配置
  /* Configure one bit for preemption priority */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

  /* Enable the EXTI15_10 Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
//中断函数
void EXTI15_10_IRQHandler(void)
{
```

```
  //判断中断管脚
  if(EXTI_GetITStatus(EXTI_Line12) != RESET)
  {
    /* Toggle PC8 pin */
    GPIO_WriteBit(GPIOC, GPIO_Pin_8, (BitAction)((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_8))));

    /* Clear the EXTI line 12 pending bit */
    //清除中断标准
    EXTI_ClearITPendingBit(EXTI_Line12);
  }
}
```

例子使用 PC12,由于 PC12 也连接红外接收，所以用红外遥控器对着接收头按下按键将看到 LED2 闪烁，没有红外接收的，自己用手将 PC12 对地短路也行，没有红外接收的，自己用手将 PC12 对地短路也行,上图红圈中的就是 PC12（BHS-STM32-V 才有红外接收）

串口实验

串口功能描述

通用同步异步收发器(USART)提供了一种灵活的方法与使用工业标准 NRZ 异步串行数据格式的外部设备之间进行全双工数据交换。USART 利用分数波特率发生器提供宽范围的波特率选择。它支持同步单向通信和半双工单线通信，也支持 LIN(局部互连网)，智能卡协议和 IrDA(红外数据组织)SIR ENDEC 规范，以及调制解调器(CTS/RTS)操作。它还允许多处理器通信。
使用多缓冲器配置的 DMA 方式，可以实现高速数据通信。

接口通过三个引脚与其他设备连接在一起(见图 248)。任何 USART 双向通信至少需要两个脚：接收数据输入(RX)和发送数据输出(TX)。
RX：接收数据串行输。通过过采样技术来区别数据和噪音，从而恢复数据。
TX：发送数据输出。当发送器被禁止时，输出引脚恢复到它的 I/O 端口配置。当发送器被激活，并且不发送数据时，TX 引脚处于高电平。在单线和智能卡模式里，此 I/O 口被同时用于数据的发送和接收。

- 总线在发送或接收前应处于空闲状态
- 一个起始位
- 一个数据字(8 或 9 位)，最低有效位在前
- 0.5，1.5，2 个的停止位，由此表明数据帧的结束
- 使用分数波特率发生器 —— 12 位整数和 4 位小数的表示方法。
- 一个状态寄存器(USART_SR)
- 数据寄存器(USART_DR)
- 一个波特率寄存器(USART_BRR)，12 位的整数和 4 位小数
- 一个智能卡模式下的保护时间寄存器(USART_GTPR)

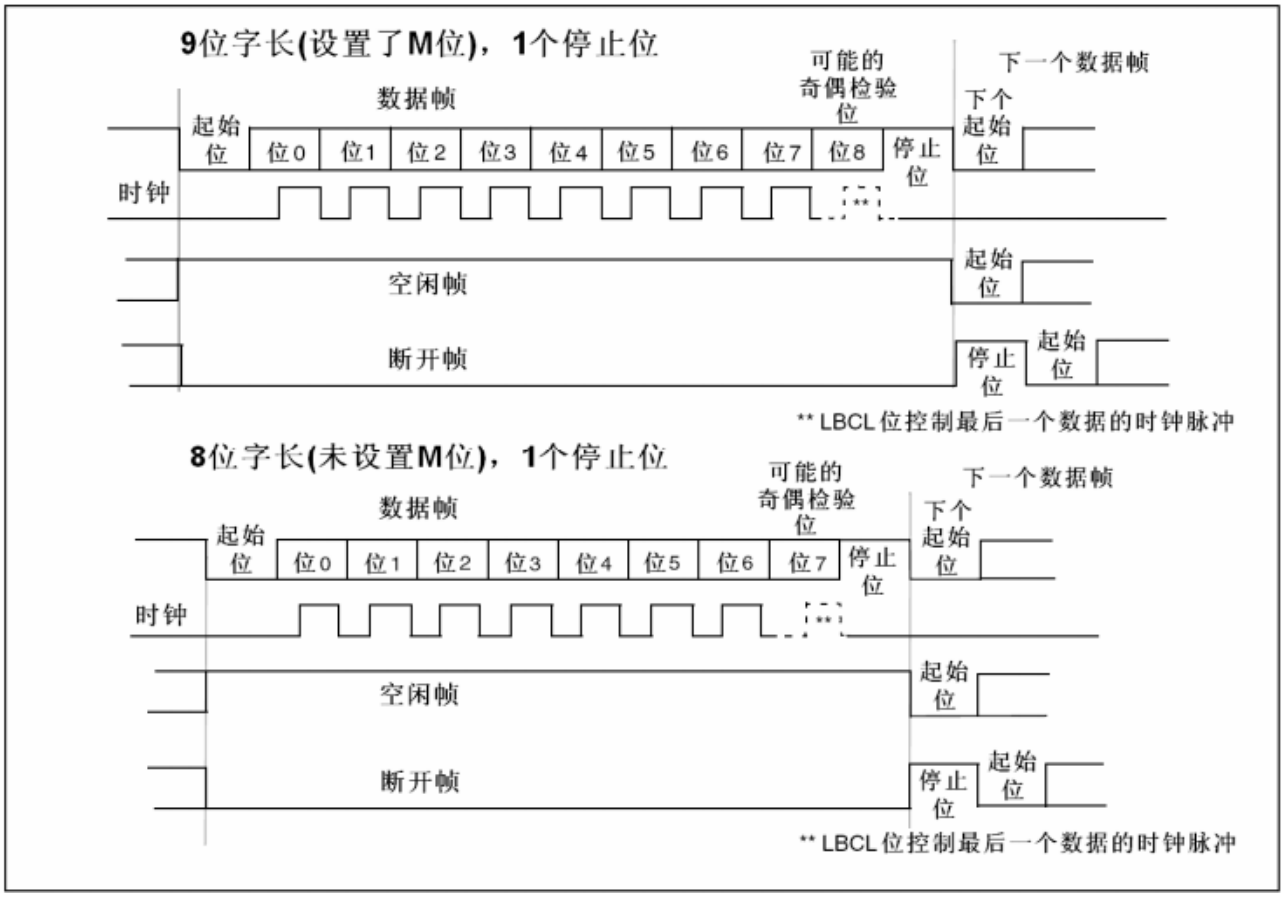
USART 特性描述

字长可以通过编程 USART_CR1 寄存器中的 M 位，选择成 8 或 9 位(见图 249)。在起始位期间，TX 脚处于低电平，在停止位期间处于高电平。
空闲符号被视为完全由’1’组成的一个完整的数据帧，后面跟着包含了数据的下一帧的开始位(‘1’的位数也包括了停止位的位数)。
断开符号 被视为在一个帧周期内全部收到’0’(包括停止位期间，也是’0’)。在断开帧结束时，发

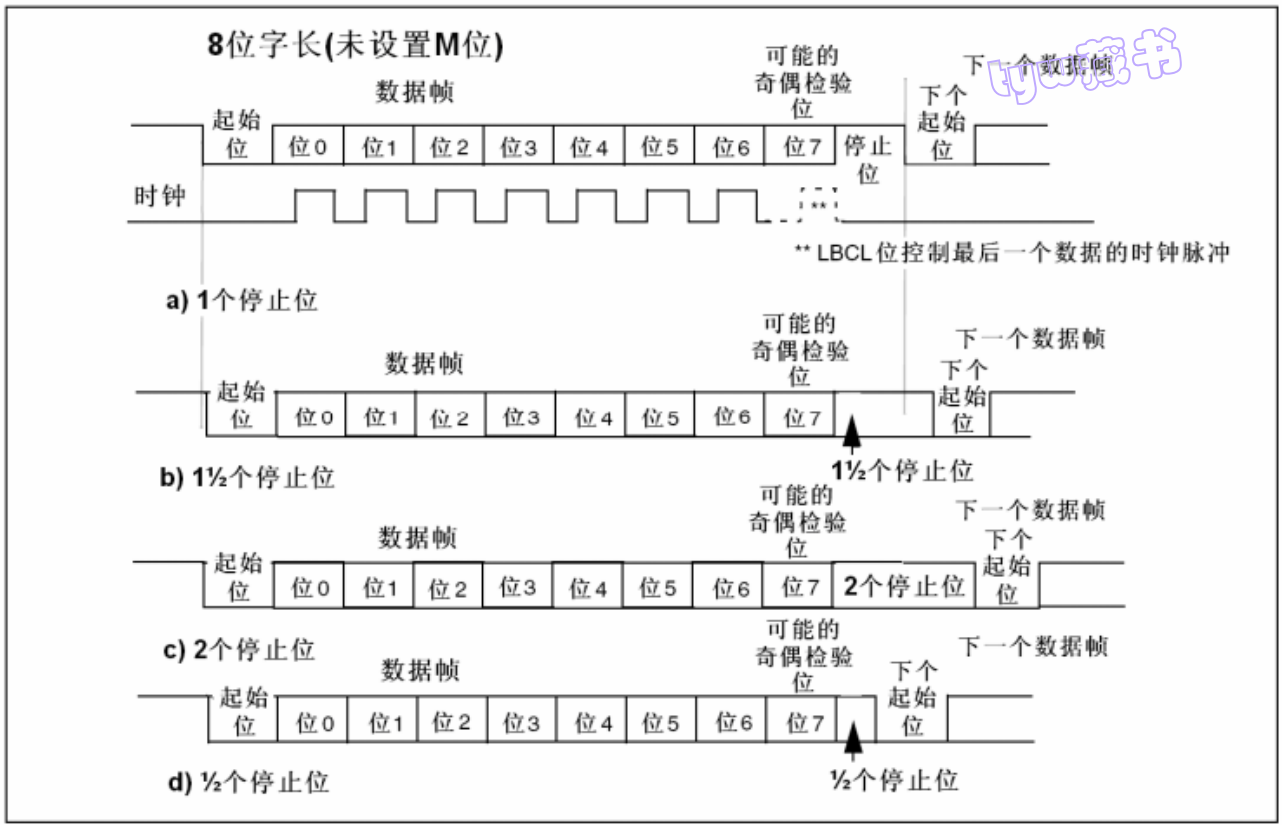
送器再插入 1 或 2 个停止位(‘1’)来应答起始位。

发送和接收由一共用的波特率发生器驱动，当发送器和接收器的使能位分别置位时，分别为其产生时钟。

字长设置



配置停止位



波特率的产生

接收器和发送器的波特率在 USARTDIV 的整数和小数寄存器中的值应设置成相同。

$$\text{Tx / Rx 波特率} = \frac{f_{CK}}{(16 * USARTDIV)}$$

这里的 fCK 是给外设的时钟(PCLK1 用于 USART2、3、4、5，PCLK2 用于 USART1)

USARTDIV 是一个无符号的定点数。这 12 位的值设置在 USART_BRR 寄存器。

状态寄存器(USART_SR)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

位31:10	保留位，硬件强制为0
位9	**CTS**: CTS 标志 (CTS flag) 如果设置了CTSE位，当nCTS输入变化状态时，该位被硬件置高。由软件将其清零。如果USART_CR3中的CTSIE为'1'，则产生中断。 0：nCTS状态线上没有变化； 1：nCTS状态线上发生变化。 注：UART4和UART5上不存在这一位。
位8	**LBD**: LIN断开检测标志 (LIN break detection flag) 当探测到LIN断开时，该位由硬件置'1'，由软件清'0'(向该位写0)。如果USART_CR3中的LBDIE = 1，则产生中断。 0：没有检测到LIN断开； 1：检测到LIN断开。 注意：若LBDIE=1，当LBD为'1'时要产生中断。
位7	**TXE**:发送数据寄存器空 (Transmit data register empty) 当TDR寄存器中的数据被硬件转移到移位寄存器的时候，该位被硬件置位。如果USART_CR1寄存器中的TXEIE为1，则产生中断。对USART_DR的写操作，将该位清零。 0：数据还没有被转移到移位寄存器； 1：数据已经被转移到移位寄存器。 注意：单缓冲器传输中使用该位。
位6	**TC**：发送完成 (Transmission complete) 当包含有数据的一帧发送完成后，并且TXE=1时，由硬件将该位置'1'。如果USART_CR1中的TCIE为'1'，则产生中断。由软件序列清除该位(先读USART_SR，然后写入USART_DR)。TC位也可以通过写入'0'来清除，只有在多缓存通讯中才推荐这种清除程序。 0：发送还未完成； 1：发送完成。
位5	**RXNE**：读数据寄存器非空 (Read data register not empty) 当RDR移位寄存器中的数据被转移到USART_DR寄存器中，该位被硬件置位。如果USART_CR1寄存器中的RXNEIE为1，则产生中断。对USART_DR的读操作可以将该位清零。RXNE位也可以通过写入0来清除，只有在多缓存通讯中才推荐这种清除程序。 0：数据没有收到； 1：收到数据，可以读出。

位4	**IDLE**：监测到总线空闲 (IDLE line detected) 当检测到总线空闲时，该位被硬件置位。如果USART_CR1中的IDLEIE为'1'，则产生中断。由软件序列清除该位(先读USART_SR，然后读USART_DR)。 0：没有检测到空闲总线； 1：检测到空闲总线。 注意：IDLE位不会再次被置高直到RXNE位被置起(即又检测到一次空闲总线)
位3	**ORE**：过载错误 (Overrun error) 当RXNE仍然是'1'的时候，当前被接收在移位寄存器中的数据，需要传送至RDR寄存器时，硬件将该位置位。如果USART_CR1中的RXNEIE为'1'的话，则产生中断。由软件序列将其清零(先读USART_SR，然后读USART_CR)。 0：没有过载错误； 1：检测到过载错误。 注意：该位被置位时，RDR寄存器中的值不会丢失，但是移位寄存器中的数据会被覆盖。如果设置了EIE位，在多缓冲器通信模式下，ORE标志置位会产生中断的。

位2	**NE**: 噪声错误标志 (Noise error flag) 在接收到的帧检测到噪音时，由硬件对该位置位。由软件序列对其清玲(先读USART_SR，再读USART_DR)。 0：没有检测到噪声； 1：检测到噪声。 注意：该位不会产生中断，因为它和RXNE一起出现，硬件会在设置RXNE标志时产生中断。在多缓冲区通信模式下，如果设置了EIE位，则设置NE标志时会产生中断。
位1	**FE**: 帧错误 (Framing error) 当检测到同步错位，过多的噪声或者检测到断开符，该位被硬件置位。由软件序列将其清零(先读USART_SR，再读USART_DR)。 0：没有检测到帧错误； 1：检测到帧错误或者break符。 注意：该位不会产生中断，因为它和RXNE一起出现，硬件会在设置RXNE标志时产生中断。如果当前传输的数据既产生了帧错误，又产生了过载错误，硬件还是会继续该数据的传输，并且只设置ORE标志位。 在多缓冲区通信模式下，如果设置了EIE位，则设置FE标志时会产生中断。
位0	**PE**: 校验错误 (Parity error) 在接收模式下，如果出现奇偶校验错误，硬件对该位置位。由软件序列对其清零(依次读USART_SR和USART_DR)。在清除PE位前，软件必须等待RXNE标志位被置'1'。如果USART_CR1中的PEIE为'1'，则产生中断。 0：没有奇偶校验错误； 1：奇偶校验错误。

数据寄存器(USART_DR)

位 31:9	保留位，硬件强制为 0
位 8:0	DR[8:0]：数据值 (Data value) 包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收 用(RDR)，该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的 并行接口(参见图 248)。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。 当使能校验位(USART_CR1 中 PCE 位被置位)进行发送时，写到 MSB 的值(根据数据的长度不 同，MSB 是第 7 位或者第 8 位)会被后来的校验位该取代。 当使能校验位进行接收时，读到的 MSB 位是接收到的校验位。

波特比率寄存器(USART_BRR)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留位，硬件强制为0
位15:4	**DIV_Mantissa[11:0]**：USARTDIV的整数部分 这12位定义了USART分频器除法因子(USARTDIV)的整数部分。
位3:0	**DIV_Fraction[3:0]**：USARTDIV的小数部分 这4位定义了USART分频器除法因子(USARTDIV)的小数部分。

控制寄存器 1(USART_CR1)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
res		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:14	保留位，硬件强制为 0。
位 13	UE：USART 使能 (USART enable) 当该位被清零，在当前字节传输完成后 USART 的分频器和输出停止工作，以减少功耗。该位由 软件设置和清零。 0：USART 分频器和输出被禁止； 1：USART 模块使能。
位 12	M：字长 (Word length) 该位定义了数据字的长度，由软件对其设置和清零 0：一个起始位，8 个数据位，n 个停止位； 1：一个起始位，9 个数据位，n 个停止位。 注意：在数据传输过程中(发送或者接收时)，不能修改这个位。
位 11	WAKE：唤醒的方法 (Wakeup method) 这位决定了把 USART 唤醒的方法，由软件对该位设置和清零。 0：被空闲总线唤醒； 1：被地址标记唤醒。
位 10	PCE：检验控制使能 (Parity control enable) 用该位选择是否进行硬件校验控制(对于发送来说就是校验位的产生；对于接收来说就是校验位 的检测)。当使能了该位，在发送数据的最高位(如果 M=1，最高位就是第 9 位；如果 M=0，最高 位就是第 8 位)插入校验位；对接收到的数据检查其校验位。软件对它置’1’或清’0’。一旦设置了 该位，当前字节传输完成后，校验控制才生效。 0：禁止校验控制； 1：使能校验控制。

位 9	PS：校验选择 (Parity selection) 当校验控制使能后，该位用来选择是采用偶校验还是奇校验。软件对它置’1’或清’0’。当前字节 传输完成后，该选择生效。 0：偶校验； 1：奇校验。
位 8	PEIE：PE 中断使能 (PE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 PE 为’1’时，产生 USART 中断。
位 7	TXEIE：发送缓冲区空中断使能 (TXE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 TXE 为’1’时，产生 USART 中断。
位 6	TCIE：发送完成中断使能 (Transmission complete interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 TC 为’1’时，产生 USART 中断。
位 5	RXNEIE：接收缓冲区非空中断使能 (RXNE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 ORE 或者 RXNE 为’1’时，产生 USART 中断。
位 4	IDLEIE：IDLE 中断使能 (IDLE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 IDLE 为’1’时，产生 USART 中断。
位 3	TE：发送使能 (Transmitter enable) 该位使能发送器。该位由软件设置或清除。 0：禁止发送； 1：使能发送。 注意：1．在数据传输过程中，除了在智能卡模式下，如果 TE 位上有个 0 脉冲(即设置为’0’之后 再设置为’1’)，会在当前数据字传输完成后，发送一个“前导符”(空闲总线)。 2．当 TE 被设置后，在真正发送开始之前，有一个比特时间的延迟。
位 2	RE：接收使能 (Receiver enable) 该位由软件设置或清除。 0：禁止接收； 1：使能接收，并开始搜寻 RX 引脚上的起始位。
位 1	RWU：接收唤醒 (Receiver wakeup) 该位用来决定是否把 USART 置于静默模式。该位由软件设置或清除。当唤醒序列到来时，硬件 也会将其清零。 0：接收器处于正常工作模式； 1：接收器处于静默模式。 注意：1．在把 USART 置于静默模式(设置 RWU 位)之前，USART 要已经先接收了一个数据

	字 节。否则在静默模式下，不能被空闲总线检测唤醒。 2．当配置成地址标记检测唤醒(WAKE 位=1)，在 RXNE 位被置位时，不能用软件修改 RWU 位。
位 0	SBK：发送断开帧 (Send break) 使用该位来发送断开字符。该位可以由软件设置或清除。操作过程应该是软件设置位它，然后 在断开帧的停止位时，由硬件将该位复位。 0：没有发送断开字符； 1：将要发送断开字符。

控制寄存器 2(USART_CR2)

位 31:15	保留位，硬件强制为 0。
位 14	LINEN：LIN 模式使能 (LIN mode enable) 该位由软件设置或清除。 0：禁止 LIN 模式； 1：使能 LIN 模式。 在 LIN 模式下，可以用 USART_CR1 寄存器中的 SBK 位发送 LIN 同步断开符(低 13 位)，以及检测 LIN 同步断开符。
位 13:12	STOP：停止位 (STOP bits) 这 2 位用来设置停止位的位数 00：1 个停止位； 01：0.5 个停止位； 10：2 个停止位； 11：1.5 个停止位； 注：UART4 和 UART5 不能用 0.5 停止位和 1.5 停止位。
位 11	CLKEN：时钟使能 (Clock enable) 该位用来使能 CK 引脚 0：禁止 CK 引脚； 1：使能 CK 引脚。 注：UART4 和 UART5 上不存在这一位。
位 10	CPOL：时钟极性 (Clock polarity) 在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的极性。和 CPHA 位一起配合来产生需要 的时钟/数据的采样关系 0：总线空闲时 CK 引脚上保持低电平； 1：总线空闲时 CK 引脚上保持高电平。 注：UART4 和 UART5 上不存在这一位。
位 9	CPHA： 时钟相位 (Clock phase)

	在同步模式下，可以用该位选择 SLCK 引脚上时钟输出的相位。和 CPOL 位一起配合来产生需要的时钟/数据的采样关系 0：在时钟的第一个边沿进行数据捕获； 1：在时钟的第二个边沿进行数据捕获。 注：UART4 和 UART5 上不存在这一位。
位 8	LBCL：最后一位时钟脉冲 (Last bit clock pulse) 在同步模式下，使用该位来控制是否在 CK 引脚上输出最后发送的那个数据字节(MSB)对应的时 钟脉冲 0：最后一位数据的时钟脉冲不从 CK 输出； 1：最后一位数据的时钟脉冲会从 CK 输出。 注意：1．最后一个数据位就是第 8 或者第 9 个发送的位(根据 USART_CR1 寄存器中的 M 位所定 义的 8 或者 9 位数据帧格式)。 2．UART4 和 UART5 上不存在这一位。
位 7	保留位，硬件强制为 0
位 6	LBDIE：LIN 断开符检测中断使能 (LIN break detection interrupt enable) 断开符中断屏蔽(使用断开分隔符来检测断开符) 0：禁止中断； 1：只要 USART_SR 寄存器中的 LBD 为’1’就产生中断。
位 5	LBDL：LIN 断开符检测长度 (LIN break detection length) 该位用来选择是 11 位还是 10 位的断开符检测 0：10 位的断开符检测； 1：11 位的断开符检测。
位 4	保留位，硬件强制为 0
位 3:0	ADD[3:0]：本设备的 USART 节点地址 该位域给出本设备 USART 节点的地址。 这是在多处理器通信下的静默模式中使用的，使用地址标记来唤醒某个 USART 设备。

控制寄存器 3(USART_CR3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留					CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:11	保留位，硬件强制为 0
位 10	CTSIE：CTS 中断使能 (CTS interrupt enable) 0：禁止中断； 1：USART_SR 寄存器中的 CTS 为’1’时产生中断。 注：UART4 和 UART5 上不存在这一位。
位 9	CTSE：CTS 使能 (CTS enable) 0：禁止 CTS 硬件流控制； 1：CTS 模式使能，只有 nCTS 输入信号有效(拉成低电平)时才能发送数据。如果在数据传输的 过程中，nCTS 信号变成无效，那么发完这个数据后，传输就停止下来。如果当 nCTS 为无效

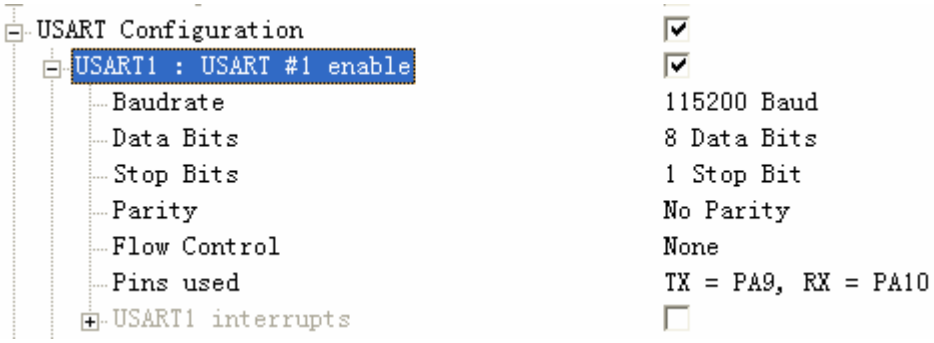
	时，往数据寄存器里写数据，则要等到 nCTS 有效时才会发送这个数据。 注：UART4 和 UART5 上不存在这一位。
位 8	RTSE：RTS 使能 (RTS enable) 0：禁止 RTS 硬件流控制； 1：RTS 中断使能，只有接收缓冲区内有空余的空间时才请求下一个数据。当前数据发送完成后，发送操作就需要暂停下来。如果可以接收数据了，将 nRTS 输出置为有效(拉至低电平)。 注：UART4 和 UART5 上不存在这一位。
位 7	DMAT：DMA 使能发送 (DMA enable transmitter) 该位由软件设置或清除。 0：禁止发送时的 DMA 模式。 1：使能发送时的 DMA 模式； 注：UART4 和 UART5 上不存在这一位。
位 6	DMAR: DMA 使能接收 (DMA enable receiver) 该位由软件设置或清除。 0：禁止接收时的 DMA 模式。 1：使能接收时的 DMA 模式； 注：UART4 和 UART5 上不存在这一位。
位 5	SCEN: 智能卡模式使能 (Smartcard mode enable) 该位用来使能智能卡模式 0：禁止智能卡模式； 1：使能智能卡模式。 注：UART4 和 UART5 上不存在这一位。
位 4	NACK：智能卡 NACK 使能 (Smartcard NACK enable) 0：校验错误出现时，不发送 NACK； 1：校验错误出现时，发送 NACK。 注：UART4 和 UART5 上不存在这一位。
位 3	HDSEL：半双工选择 (Half-duplex selection) 选择单线半双工模式 0：不选择半双工模式； 1：选择半双工模式。
位 2	IRLP：红外低功耗 (IrDA low-power) 该位用来选择普通模式还是低功耗红外模式 0：通常模式； 1：低功耗模式。
位 1	IREN：红外模式使能 (IrDA mode enable) 该位由软件设置或清除。 0：不使能红外模式； 1：使能红外模式。
位 0	EIE：错误中断使能 (Error interrupt enable) 在多缓冲区通信模式下，当有帧错误、过载或者噪声错误时(USART_SR 中的 FE=1，或者 ORE=1，或者 NE=1)产生中断。 0：禁止中断； 1：只要 USART_CR3 中的 DMAR=1，并且 USART_SR 中的 FE=1，或者 ORE=1，或者 NE=1，则产生中断

保护时间和预分频寄存器(USART_GTPR)

位 31:16	保留位，硬件强制为 0
位 15:8	GT[7:0]：保护时间值 (Guard time value) 该位域规定了以波特时钟为单位的保护时间。在智能卡模式下，需要这个功能。当保护时间过 去后，才会设置发送完成标志。 注：UART4 和 UART5 上不存在这一位。
位 7:0	PSC[7:0]：预分频器值 (Prescaler value) - 在红外(IrDA)低功耗模式下： PSC[7:0]=红外低功耗波特率 对系统时钟分频以获得低功耗模式下的频率： 源时钟被寄存器中的值(仅有 8 位有效)分频 00000000：保留 – 不要写入该值； 00000001：对源时钟 1 分频； 00000010：对源时钟 2 分频； …… - 在红外(IrDA)的正常模式下：PSC 只能设置为 00000001 - 在智能卡模式下： PSC[4:0]：预分频值 对系统时钟进行分频，给智能卡提供时钟。 寄存器中给出的值(低 5 位有效)乘以 2 后，作为对源时钟的分频因子 00000：保留 – 不要写入该值； 00001：对源时钟进行 2 分频； 00010：对源时钟进行 4 分频； 00011：对源时钟进行 6 分频； …… 注意：1. 位[7:5]在智能卡模式下没有意义。 2. UART4 和 UART5 上不存在这一位。

BHS-STM32 实验 13-USART串口查询方式(直接操作寄存器)

功能：不断查询串口状态，看串口是否收到数据，接收到什么字符就返回什么字符
串口 1 配置如下



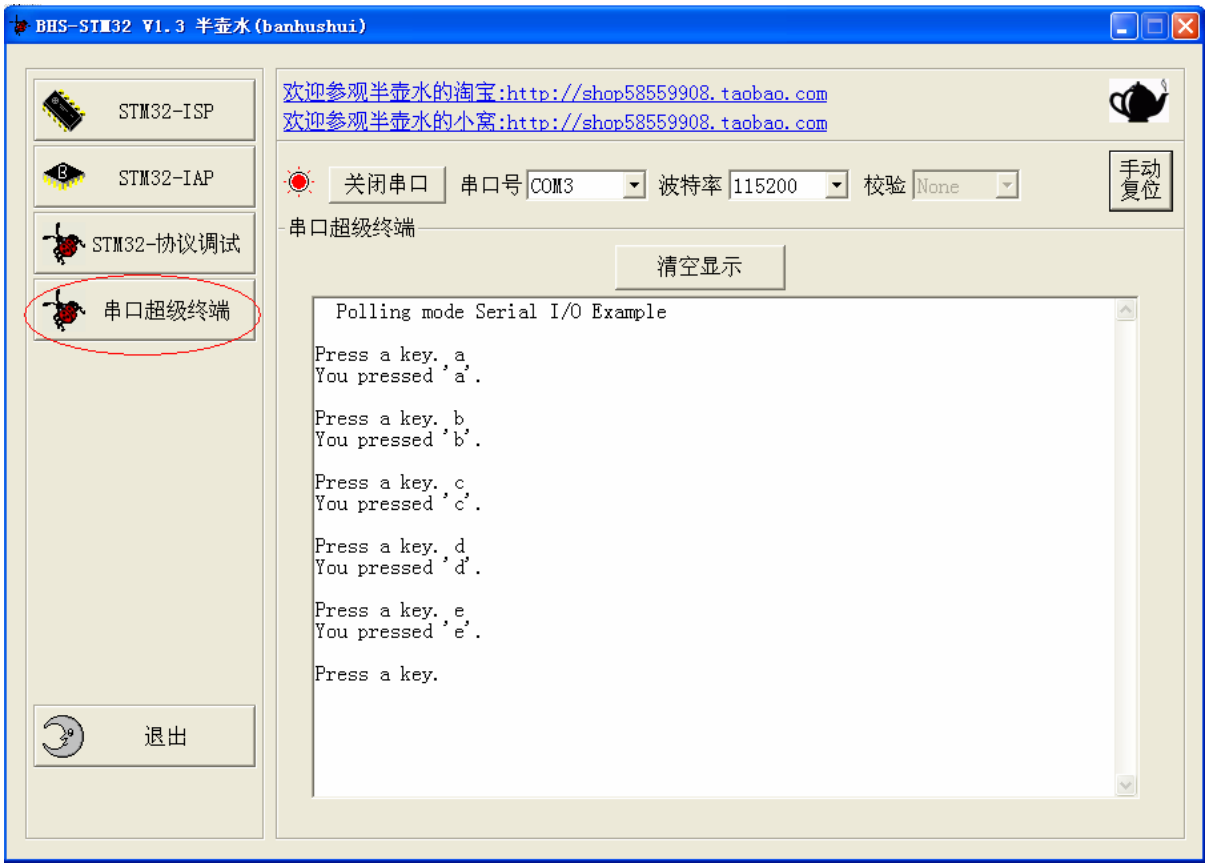
tyw藏书

```
/*---------------------------------------------------------------------
   从串口发送一个字节数据
   Write character to Serial Port.
  *---------------------------------------------------------------------*/
int SendChar (int ch)  {

   //等待发送结束
   while (!(USART1->SR & USART_FLAG_TXE));
   //将数据放入发送寄存器
   USART1->DR = (ch & 0x1FF);

   return (ch);
}


/*---------------------------------------------------------------------
   从串口读取一个字节数据，直到读到数据才返回
   Read character to Serial Port.
  *---------------------------------------------------------------------*/
int GetKey (void)  {

   //等待接收结束
   while (!(USART1->SR & USART_FLAG_RXNE));
   //从接受寄存器读取数据并返回
   return ((int)(USART1->DR & 0x1FF));
}


/*---------------------------------------------------------------------
   MAIN function
  *---------------------------------------------------------------------*/
int main (void) {
   // STM32 setup  初始化串口
   stm32_Init ();
   //串口实验开始
   printf ("Polling mode Serial I/O Example\r\n\r\n");
```

```
  while (1) {                                          // Loop forever
    unsigned char c;
    //提示从串口输入 1 个字符
    printf ("Press a key. ");
    //从串口接收 1 个字符
      c = getchar ();
      //从串口输出回车换行
    printf ("\r\n");
    //从串口输出接收到的字符
    printf ("You pressed '%c'.\r\n\r\n", c);
  } // end while
} // end main
```

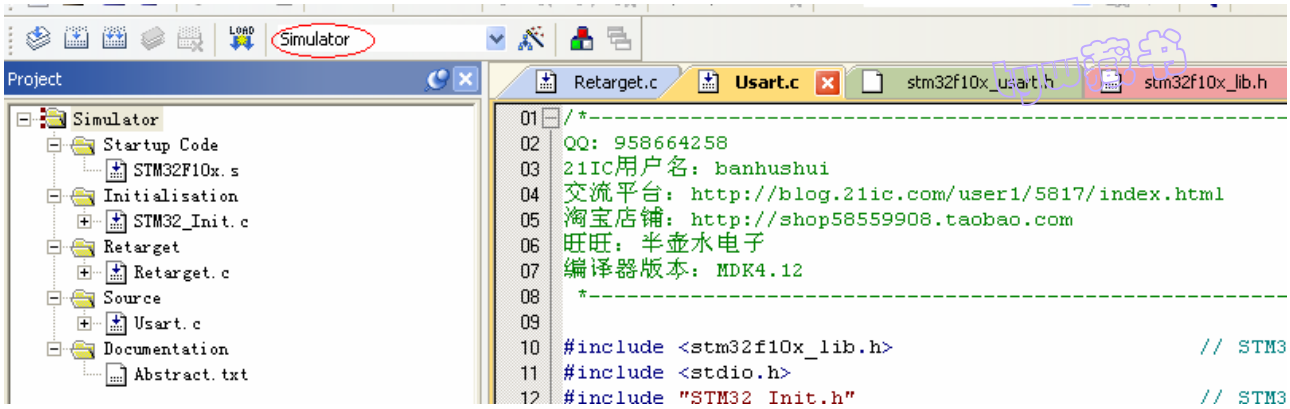
本串口程序是接收到什么字符就返回什么字符

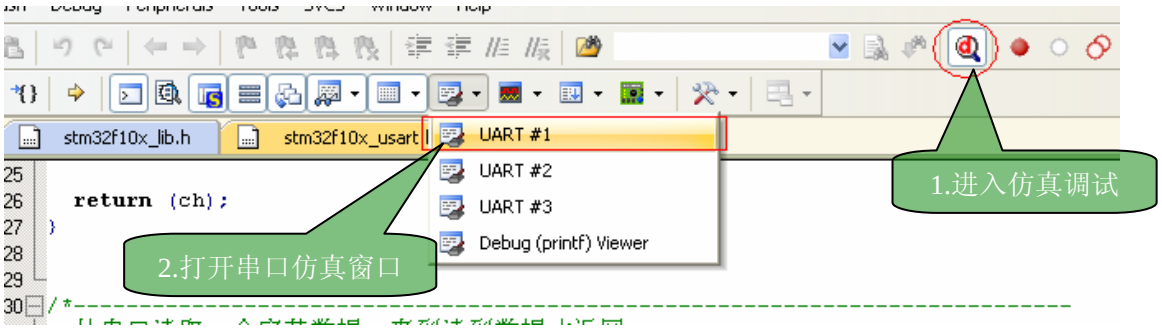
使用我提供的串口调试工具，选择【串口超级终端】波特率设置 115200



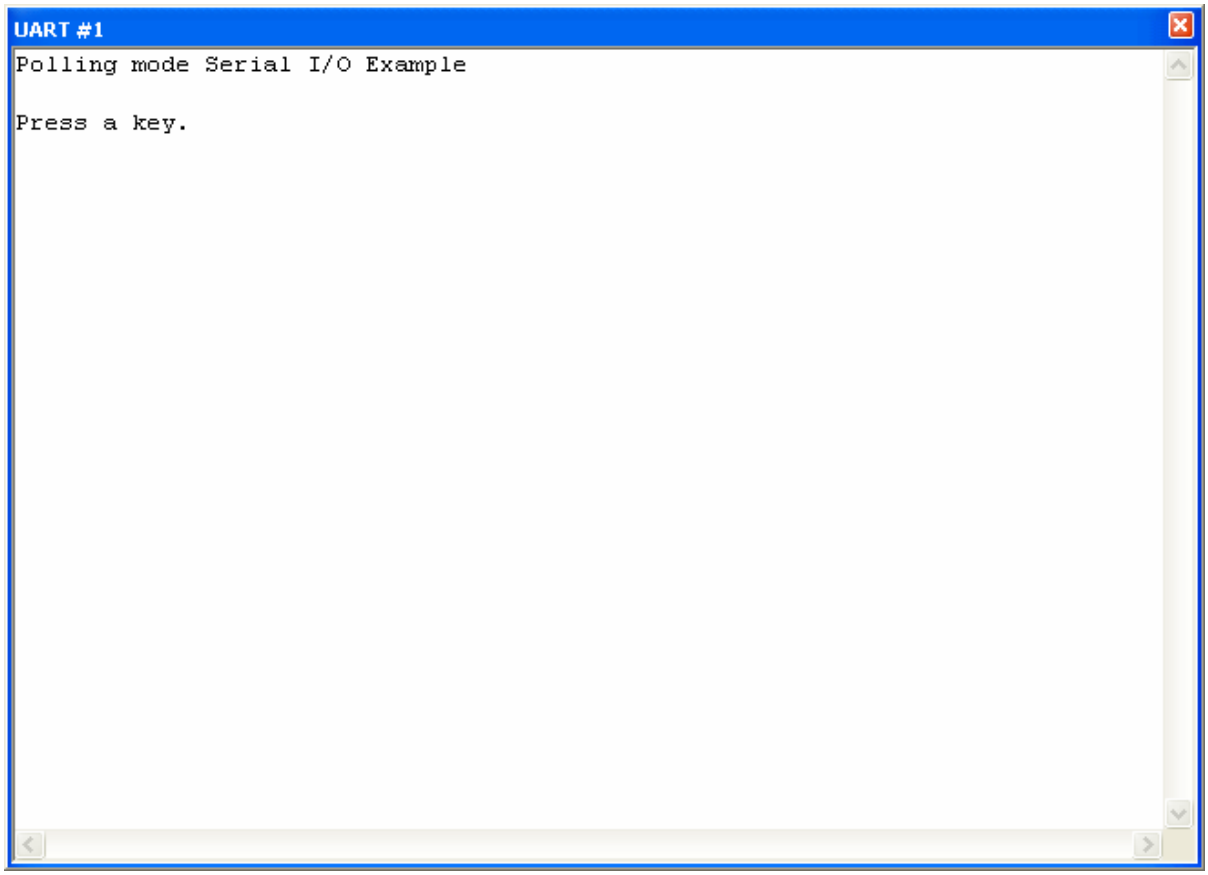
软件仿真：

选择软件仿真

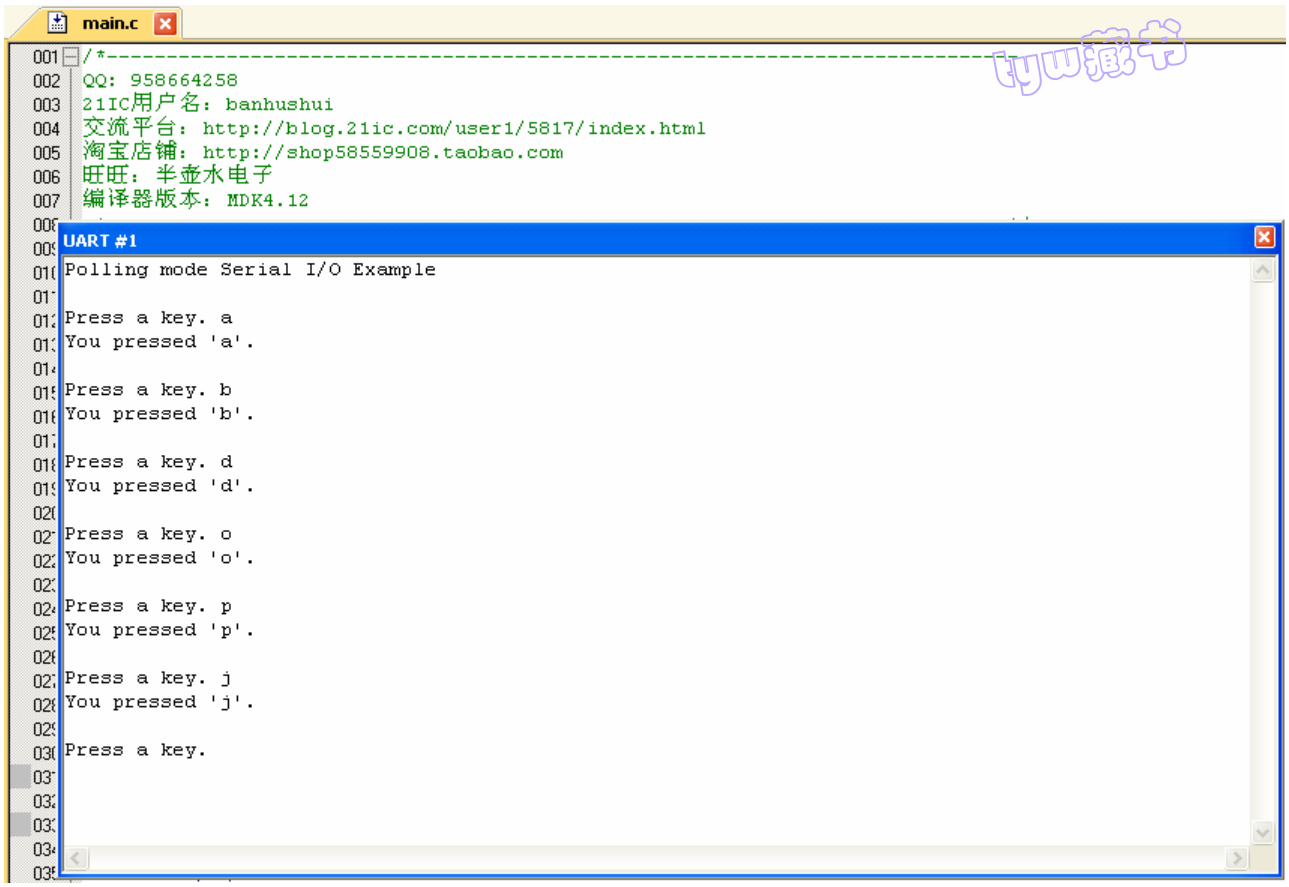




进入仿真调试后，打开串口仿真窗口，允许程序弹出下面仿真窗口



使用键盘输入字符，同样可以看到返回数据



BHS-STM32 实验 14-USART串口查询方式(库函数)

功能：不断查询串口状态，看串口是否收到数据，接收到什么字符就返回什么字符

//考虑到效率问题，串口初始化用库函数

//串口收发函数仍然直接操作寄存器

函数 USART_Init

函数名	USART_Init
函数原形	void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)
功能描述	根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器
输入参数 1	USARTx：x 可以是 1，2 或者 3，来选择 USART 外设
输入参数 2	USART_InitStruct：指向结构 USART_InitTypeDef 的指针，包含了外设 USART 的配置信息。参阅 Section：USART_InitTypeDef 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

USART_InitTypeDef structure

USART_InitTypeDef 定义于文件“stm32f10x_usart.h”：

typedef struct

{

u32 USART_BaudRate;

u16 USART_WordLength;

u16 USART_StopBits;

u16 USART_Parity;
u16 USART_HardwareFlowControl;
u16 USART_Mode;
u16 USART_Clock;
u16 USART_CPOL;
u16 USART_CPHA;
u16 USART_LastBit;
} USART_InitTypeDef;

tyw藏书

USART_InitTypeDef 成员 USART 模式对比

成员	异步模式	同步模式
USART_BaudRate	X	X
USART_WordLength	X	X
USART_StopBits	X	X
USART_Parity	X	X
USART_HardwareFlowControl	X	X
USART_Mode	X	X
USART_Clock		X
USART_CPOL		X
USART_CPHA		X
USART_LastBit		X

USART_BaudRate

该成员设置了 USART 传输的波特率，波特率可以由以下公式计算：

IntegerDivider = ((APBClock) / (16 * (USART_InitStruct->USART_BaudRate)))

FractionalDivider = ((IntegerDivider - ((u32) IntegerDivider)) * 16) + 0.5

USART_WordLength

USART_WordLength 提示了在一个帧中传输或者接收到的数据位数。

USART_WordLength	描述
USART_WordLength_8b	8 位数据
USART_WordLength_9b	9 位数据

USART_StopBits

USART_StopBits 定义了发送的停止位数目。

USART_StopBits	描述
USART_StopBits_1	在帧结尾传输 1 个停止位
USART_StopBits_0.5	在帧结尾传输 0.5 个停止位
USART_StopBits_2	在帧结尾传输 2 个停止位
USART_StopBits_1.5	在帧结尾传输 1.5 个停止位

USART_Parity

USART_Parity 定义了奇偶模式。

USART_Parity	描述
USART_Parity_No	奇偶失能
USART_Parity_Even	偶模式
USART_Parity_Odd	奇模式

注意：奇偶校验一旦使能，在发送数据的 MSB 位插入经计算的奇偶位（字长 9 位时的第 9 位，字长 8 位时的第 8 位）。

lyw藏书

USART_HardwareFlowControl

USART_HardwareFlowControl 指定了硬件流控制模式使能还是失能。

USART_HardwareFlowControl	描述
USART_HardwareFlowControl_None	硬件流控制失能
USART_HardwareFlowControl_RTS	发送请求 RTS 使能
USART_HardwareFlowControl_CTS	清除发送 CTS 使能
USART_HardwareFlowControl_RTS_CTS	RTS 和 CTS 使能

USART_Mode

USART_Mode 指定了使能或者失能发送和接收模式。

USART_Mode	描述
USART_Mode_Tx	发送使能
USART_Mode_Rx	接收使能

USART_CLOCK

USART_CLOCK 提示了 USART 时钟使能还是失能。

USART_CLOCK	描述
USART_Clock_Enable	时钟高电平活动
USART_Clock_Disable	时钟低电平活动

USART_CPOL

USART_CPOL 指定了下 SLCK 引脚上时钟输出的极性。

USART_CPOL	描述
USART_CPOL_High	时钟高电平
USART_CPOL_Low	时钟低电平

USART_CPHA

USART_CPHA 指定了下 SLCK 引脚上时钟输出的相位，和 CPOL 位一起配合来产生用户希望的时钟/数据的采样关系。

USART_CPHA	描述
USART_CPHA_1Edge	时钟第一个边沿进行数据捕获
USART_CPHA_2Edge	时钟第二个边沿进行数据捕获

USART_LastBit

USART_LastBit 来控制是否在同步模式下，在 SCLK 引脚上输出最后发送的那个数据字 (MSB)对应的时钟脉冲。

USART_LastBit	描述
USART_LastBit_Disable	最后一位数据的时钟脉冲不从 SCLK 输出
USART_LastBit_Enable	最后一位数据的时钟脉冲从 SCLK 输出

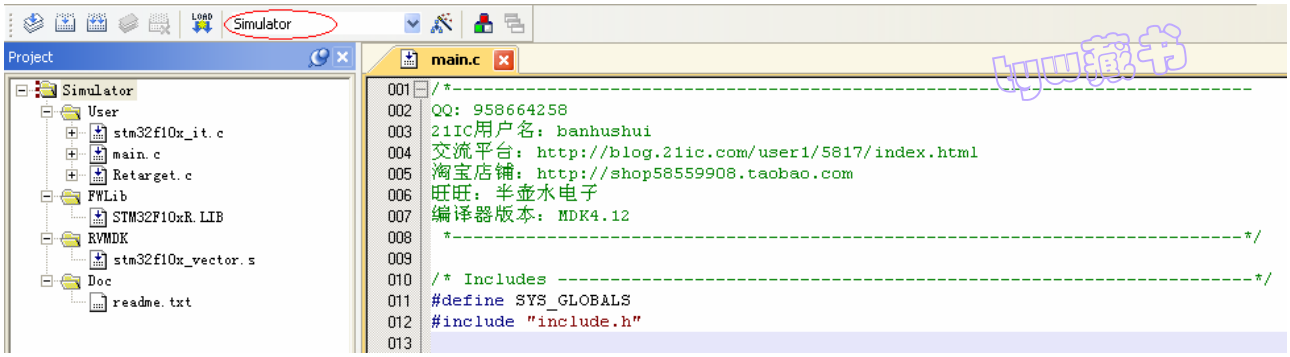
与上例最主要的区别：是串口初始化函数使用 ST 库函数

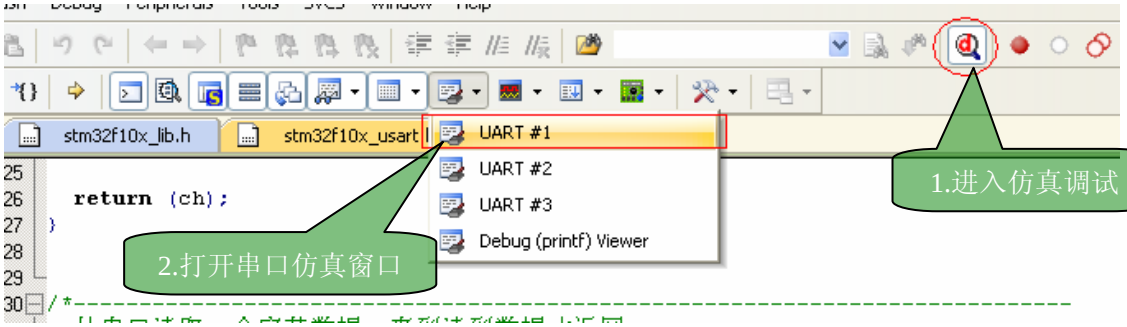
```
//串口初始化函数
void USART1_InitConfig(uint32 BaudRate)
```

```
{USART_InitTypeDef USART_InitStructure;
 GPIO_InitTypeDef GPIO_InitStructure;

  //使能串口的 RCC 时钟
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);

  //串口使用的 GPIO 口配置
  //配置串口接收脚
  /* Configure USART1 Rx (PA.10) as input floating */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* Configure USART1 Tx (PA.09) as alternate function push-pull */
  //配置串口发送脚
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  //配置串口
  USART_InitStructure.USART_BaudRate = BaudRate;
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure.USART_StopBits = USART_StopBits_1;
  USART_InitStructure.USART_Parity = USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

  /* Configure USART1 */
  USART_Init(USART1, &USART_InitStructure);//配置串口 1

  /* Enable the USART1 */
  USART_Cmd(USART1, ENABLE);//使能串口 1
}
```

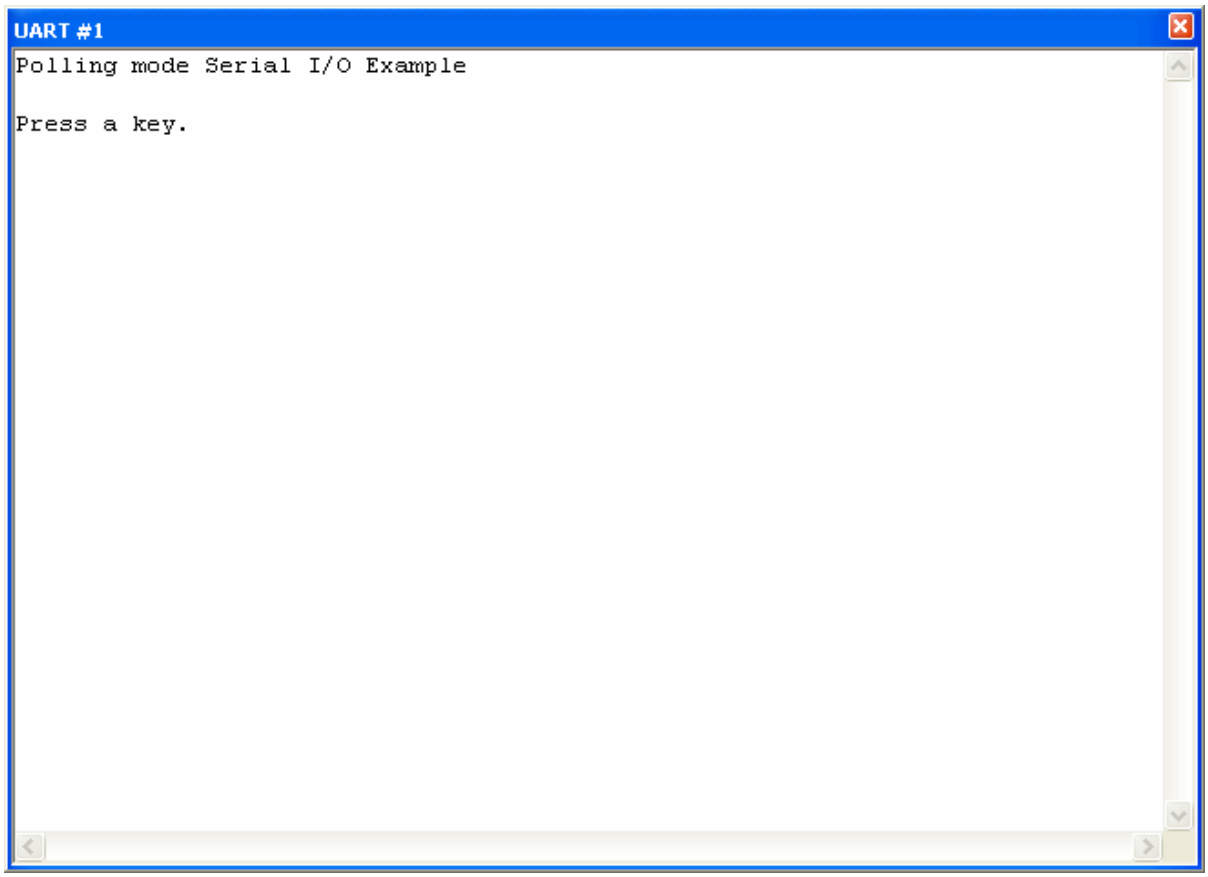
软件仿真：

选择软件仿真

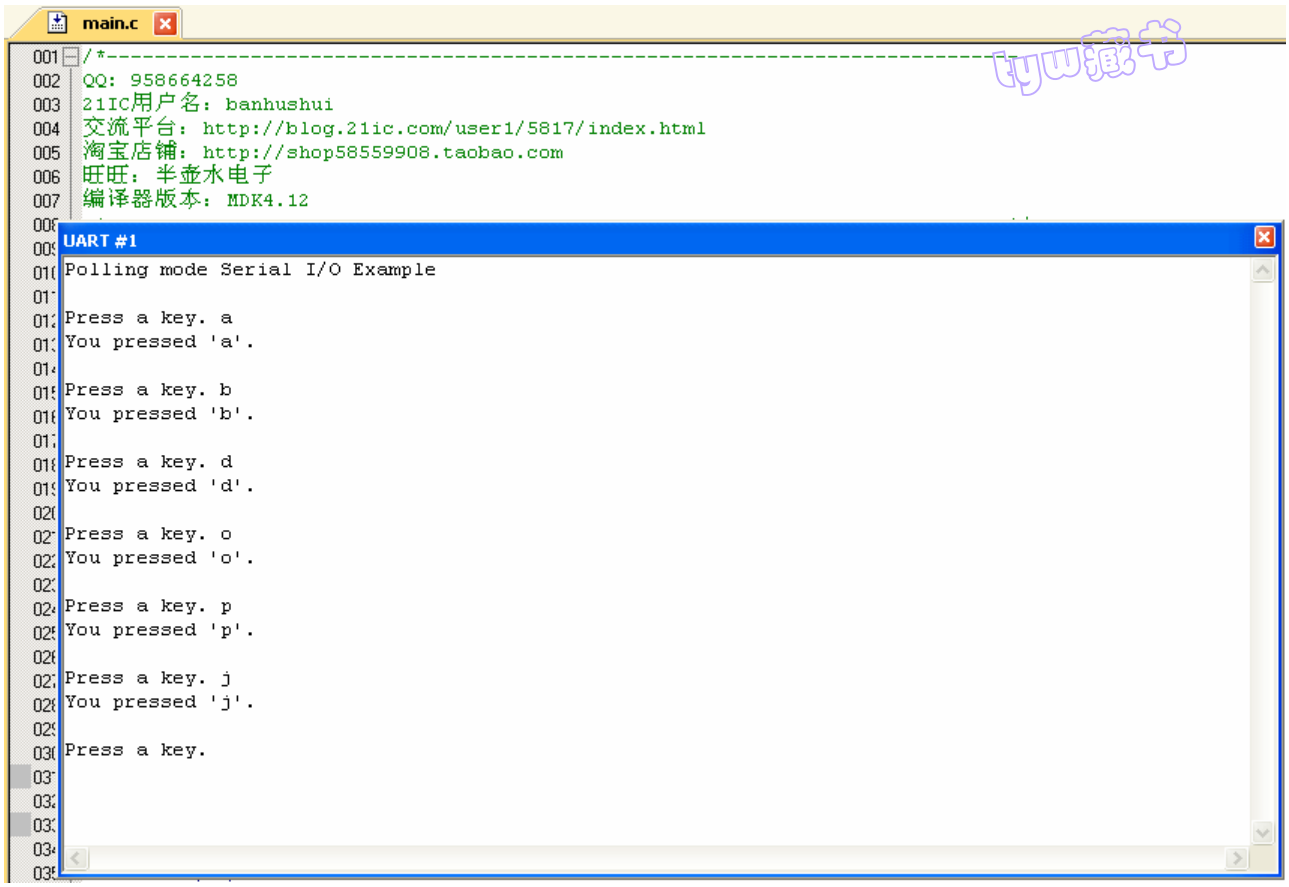




进入仿真调试后，打开串口仿真窗口，允许程序弹出下面仿真窗口



使用键盘输入字符，同样可以看到返回数据

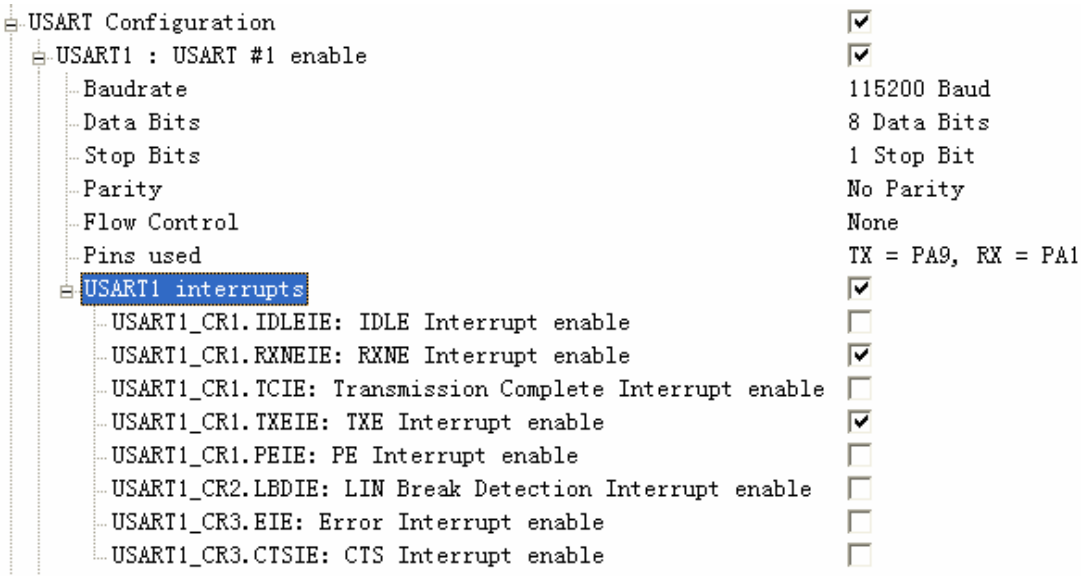


BHS-STM32 实验 15-USART串口中断方式(直接操作寄存器)

功能：不断查询串口状态，看串口是否收到数据，接收到什么字符就返回什么字符

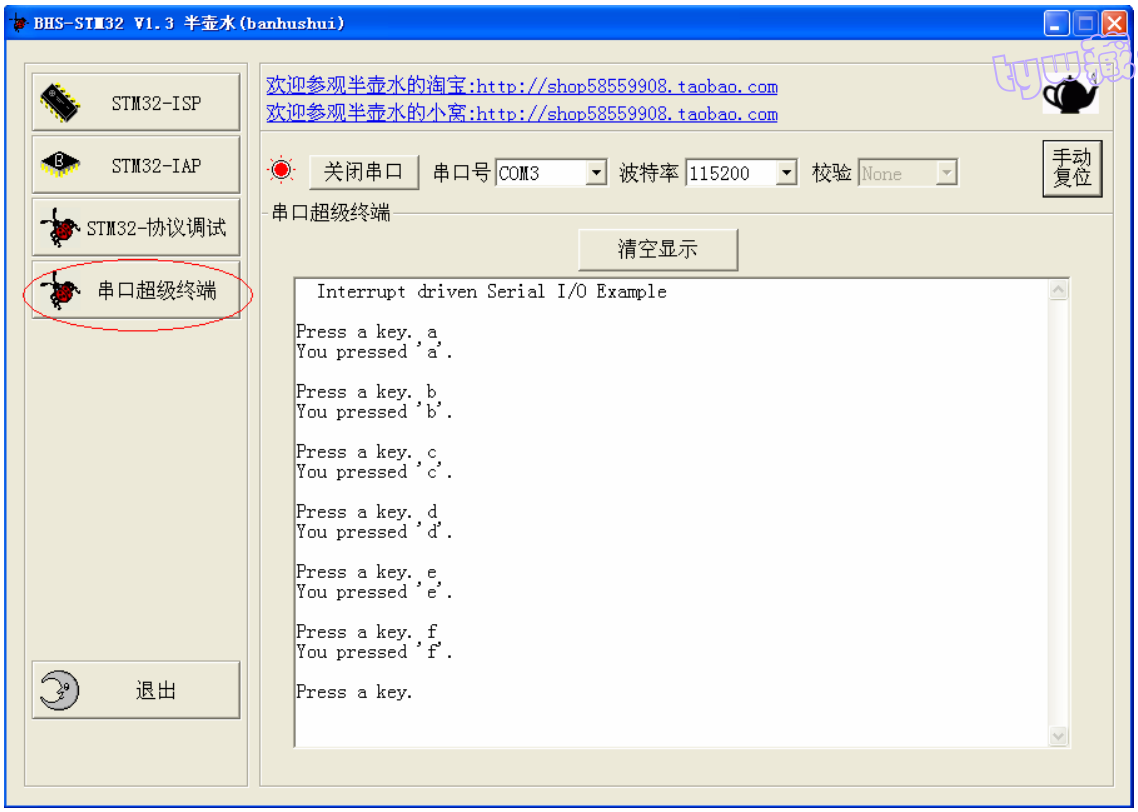
中断方式不再查询串口状态，这样 CPU 效率将显著提高

串口 1 配置如下



该程序功能与查询方式例子一样，只是实现方式不同而已

用我提供的串口调试工具，选择【串口超级终端】波特率设置 115200



串口数据处理使用队列缓冲，数据先进先出，缓冲满，不再处理

这样串口收发数据效率提高很多

```
/*-------------------------------------------------------------------------
  *-------------------------------------------------------------------------*/
//数据缓冲队列
struct buf_st
{
    unsigned int in;// Next In Index       下一个数据输入指针
    unsigned int out;// Next Out Index    下一个数据输出指针
    char buf[RBUF_SIZE];// Buffer    数据缓冲队列
};

//声明 1 个接收数据缓冲队列
static struct buf_st rbuf =
{
    0, 0,
};
#define SIO_RBUFLEN ((unsigned short)(rbuf.in - rbuf.out))

//声明 1 个发送数据缓冲队列
static struct buf_st tbuf =
{
    0, 0,
};
#define SIO_TBUFLEN ((unsigned short)(tbuf.in - tbuf.out))
```

```
static unsigned int tx_restart = 1;                    // NZ if TX restart is required

/*------------------------------------------------------------------------------
   USART1_IRQHandler
   Handles USART1 global interrupt request.
   串口中断函数
 *------------------------------------------------------------------------------*/
void USART1_IRQHandler(void)
{
    volatile unsigned int IIR;
    struct buf_st* p;

    //读取串口状态
    IIR = USART1->SR;
    //串口接收中断
    if (IIR & USART_FLAG_RXNE) // read interrupt
    {
        //必须清除中断标志
        USART1->SR &= ~USART_FLAG_RXNE;                    // clear interrupt

        p = &rbuf;
        //接收数据缓冲未满继续放数据进缓冲区
        if (((p->in - p->out) & ~(RBUF_SIZE - 1)) == 0)
        {
            p->buf[p->in & (RBUF_SIZE - 1)] = (USART1->DR & 0x1FF);
            p->in++;
        }
    }
    //串口发送中断
    if (IIR & USART_FLAG_TXE)
    {
        //必须清除中断标志
        USART1->SR &= ~USART_FLAG_TXE;                     // clear interrupt

        p = &tbuf;
        //发送数据缓冲还有数据继续从发送缓冲区取数据
        if (p->in != p->out)
        {
            USART1->DR = (p->buf[p->out & (TBUF_SIZE - 1)] & 0x1FF);
            p->out++;
            tx_restart = 0;
        }
        else//发送数据缓冲空表示数据发送结束
        {
```

```
            tx_restart = 1;
            USART1->CR1 &= ~USART_FLAG_TXE;          // disable TX interrupt if nothing to send
        }
    }
}

/*-------------------------------------------------------------------------
   buffer_Init
   initialize the buffers
   初始化接收缓冲、发送缓冲
 *-------------------------------------------------------------------------*/
void buffer_Init(void)
{
    tbuf.in = 0;                                    // Clear com buffer indexes
    tbuf.out = 0;
    tx_restart = 1;

    rbuf.in = 0;
    rbuf.out = 0;
}

/*-------------------------------------------------------------------------
   SenChar
   transmit a character
   发送 1 个字节数据
 *-------------------------------------------------------------------------*/
int SendChar(int c)
{
    struct buf_st* p = &tbuf;

    // If the buffer is full, return an error value
    //如果发送缓冲满，直接返回
    if (SIO_TBUFLEN >= TBUF_SIZE)
        return (-1);
    // Add data to the transmit buffer.
    //向发送缓冲填充数据
    p->buf[p->in & (TBUF_SIZE - 1)] = c;
    p->in++;
    //If transmit interrupt is disabled, enable it
    //如果发送中断禁止，那么开启发送中断
    if (tx_restart)
    {
        tx_restart = 0;
        // enable TX interrupt
        USART1->CR1 |= USART_FLAG_TXE;
```

tyw藏书

```
    }

    return (0);
}

/*---------------------------------------------------------------------------
  GetKey
  receive a character
  接收 1 个字节
 *---------------------------------------------------------------------------*/
int GetKey(void)
{
    struct buf_st* p = &rbuf;

    if (SIO_RBUFLEN == 0)
        return (-1);

    return (p->buf[(p->out++) & (RBUF_SIZE - 1)]);
}


/*---------------------------------------------------------------------------
  MAIN function
 *---------------------------------------------------------------------------*/
int main(void)
{
    // init RX / TX buffers
    //初始化接收发送缓冲
    buffer_Init();
    // STM32 setup  初始化串口
    stm32_Init();
    //串口实验开始
    printf("Interrupt driven Serial I/O Example\r\n\r\n");

    while (1)
    {
        // Loop forever
        unsigned char c;
        //提示从串口输入 1 个字符
        printf("Press a key. ");
        //从串口接收 1 个字符
        c = getchar();
        //从串口输出回车换行
        printf("\r\n");
        //从串口输出接收到的字符
```

```
        printf("You pressed '%c'.\r\n\r\n", c);
    } // end while
} // end main
```

tyw藏书

BHS-STM32 实验 16-USART串口中断方式(库函数)

功能：不断查询串口状态，看串口是否收到数据，接收到什么字符就返回什么字符
中断方式不再查询串口状态，这样 CPU 效率将显著提高
与上例最主要的区别：是串口初始化函数使用 ST 库函数

函数 USART_ITConfig

函数名	USART_ITConfig
函数原形	void USART_ITConfig(USART_TypeDef* USARTx, u16 USART_IT, FunctionalState NewState)
功能描述	使能或者失能指定的 USART 中断
输入参数 1	USARTx：x 可以是 1，2 或者 3，来选择 USART 外设
输入参数 2	USART_IT：待使能或者失能的 USART 中断源 参阅 Section：USART_IT 查阅更多该参数允许取值范围
输入参数 3	NewState：USARTx 中断的新状态 这个参数可以取：ENABLE 或者 DISABLE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

USART_IT

输入参数 USART_IT 使能或者失能 USART 的中断。可以取下表的一个或者多个取值的组合作为该参数的值。

USART_IT	描述
USART_IT_PE	奇偶错误中断
USART_IT_TXE	发送中断
USART_IT_TC	传输完成中断
USART_IT_RXNE	接收中断
USART_IT_IDLE	空闲总线中断
USART_IT_LBD	LIN 中断检测中断
USART_IT_CTS	CTS 中断
USART_IT_ERR	错误中断

```
//串口初始化函数
void USART1_InitConfig(uint32 BaudRate)
{
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    //使能串口的 RCC 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);

    //串口使用的 GPIO 口配置
```

```
    /* Configure USART1 Rx (PA.10) as input floating */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Configure USART1 Tx (PA.09) as alternate function push-pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    //配置串口
    USART_InitStructure.USART_BaudRate = BaudRate;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

    /* Configure USART1 */
    USART_Init(USART1, &USART_InitStructure);//配置串口 1

    /* Enable USART1 Receive interrupts  使能串口接收中断*/
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
    //串口发送中断在发送数据时开启
    //USART_ITConfig(USART1, USART_IT_TXE, ENABLE);

    /* Enable the USART1 */
    USART_Cmd(USART1, ENABLE);//使能串口 1

    //串口中断配置
    /* Configure the NVIC Preemption Priority Bits */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);

    /* Enable the USART1 Interrupt */
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

IWDG看门狗实验

IWDG看门狗功能描述

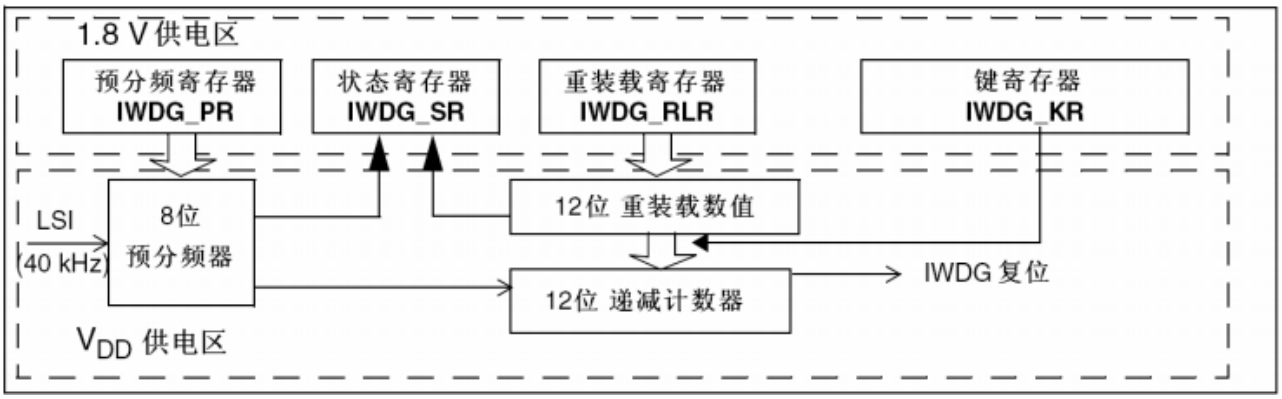
STM32F10xxx 内置两个看门狗，提供了更高的安全性、时间的精确性和使用的灵活性。两个看

门狗设备(独立看门狗和窗口看门狗)可用来检测和解决由软件错误引起的故障；当计数器达到给定的超时值时，触发一个中断(仅适用于窗口型看门狗)或产生系统复位。

独立看门狗(IWDG)由专用的低速时钟(LSI)驱动，即使主时钟发生故障它也仍然有效。窗口看门狗由从 APB1 时钟分频后得到的时钟驱动，通过可配置的时间窗口来检测应用程序非正常的过迟或过早的操作。

IWDG 最适合应用于那些需要看门狗作为一个在主程序之外，能够完全独立工作，并且对时间精度要求较低的场合。WWDG 最适合那些要求看门狗在精确计时窗口起作用的应用程序。

独立看门狗框图



看门狗超时时间(40kHz 的输入时钟(LSI))

预分频系数	PR[2:0]位	最短时间(ms) RL[11:0] = 0x000	最长时间(ms) RL[11:0] = 0xFFF
/4	0	0.1	409.6
/8	1	0.2	819.2
/16	2	0.4	1638.4
/32	3	0.8	3276.8
/64	4	1.6	6553.6
/128	5	3.2	13107.2
/256	(6或7)	6.4	26214.4

这些时间是按照 40kHz 时钟给出。实际上，MCU 内部的 RC 频率会在 30kHz 到 60kHz 之间变化。此外，即使 RC 振荡器的频率是精确的，确切的时序仍然依赖于 APB 接口时钟与 RC 振荡器时钟之间的相位差，因此总会有一个完整的 RC 周期是不确定的

IWDG 寄存器描述

键寄存器(IWDG_KR)

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

位31:16	保留，始终读为0。
位15:0	**KEY[15:0]**: 键值(只写寄存器，读出值为0x0000) (Key value) 软件必须以一定的间隔写入0xAAAA，否则，当计数器为0时，看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。(见17.3.2节) 写入0xCCCC，启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。

预分频寄存器(IWDG_PR)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留													PR[2:0]		
													rw	rw	rw

位31:3	保留，始终读为0。
位2:0	**PR[2:0]**: 预分频因子 (Prescaler divider) 这些位具有写保护设置，参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子，IWDG_SR寄存器的PVU位必须为0。 000: 预分频因子=4　　100: 预分频因子=64 001: 预分频因子=8　　101: 预分频因子=128 010: 预分频因子=16　　110: 预分频因子=256 011: 预分频因子=32　　111: 预分频因子=256 注意：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的PVU位为0时，读出的值才有效。

重装载寄存器(IWDG_RLR)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留				RL[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:12	保留，始终读为0。
位11:0	**RL[11:0]**: 看门狗计数器重装载值 (Watchdog counter reload value) 这些位具有写保护功能，参看17.3.2节。用于定义看门狗计数器的重装载值，每当向IWDG_KR寄存器写入0xAAAA时，重装载值会被传送到计数器中。随后计数器从这个值开始递减计数。看门狗超时周期可通过此重装载值和时钟预分频值来计算，参照表83。 只有当IWDG_SR寄存器中的RVU位为0时，才能对此寄存器进行修改。 注：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的RVU位为0时，读出的值才有效。

状态寄存器(IWDG_SR)

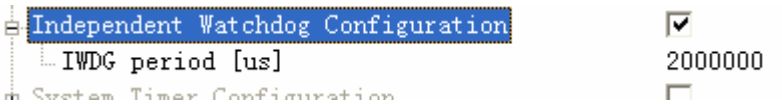
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留														RVU	PVU
														r	r

位31:2	保留。
位1	**RVU**: 看门狗计数器重装载值更新 (Watchdog counter reload value update) 此位由硬件置'1'用来指示重装载值的更新正在进行中。当在VDD域中的重装载更新结束后，此位由硬件清'0'(最多需5个40kHz的RC周期)。重装载值只有在RVU位被清'0'后才可更新。
位0	**PVU**: 看门狗预分频值更新 (Watchdog prescaler value update) 此位由硬件置'1'用来指示预分频值的更新正在进行中。当在VDD域中的预分频值更新结束后，此位由硬件清'0'(最多需5个40kHz的RC周期)。预分频值只有在PVU位被清'0'后才可更新。

如果在应用程序中使用了多个重装载值或预分频值，则必须在 RVU 位被清除后才能重新改变预装载值，在 PVU 位被清除后才能重新改变预分频值。然而，在预分频和/或重装值更新后，不必等待 RVU 或 PVU 复位，可继续执行下面的代码。(即是在低功耗模式下，此写操作仍会被继续执行完成。)

BHS-STM32 实验 17-IWDG看门狗(直接操作寄存器)

本例子实现看门狗功能，程序不喂狗后自动复位
看门狗配置如下：



1.该例子必须下载到 FLASH 中运行才能看到效果
2.运行程序时拔掉 JTAG 调试接口
3.main 函数执行到 while (1)这里，不喂狗，那么系统复位，如果系统不复位，LED 不再变化。
可以看到 LED 停止闪烁一段时间又开始闪烁说明看门狗复位了
演示代码如下

```
int main(void)
{
    int i;

    // STM32 setup
    //STM32 初始化
    stm32_Init();
    // IWDG Reset Flag set  检查看门狗复位标志
    if (RCC->CSR & (1 << 29))
    {
        // Clear Reset Flags  清除看门狗复位标志
        RCC->CSR |= (1 << 24);
        //LED on LED 点亮
        GPIOC->BRR = 1 << (ledPosIwdg + 8);
    }
    else
```

```
    {
        //LED off LED 熄灭
        GPIOC->BSRR = 1 << (ledPosIwdg + 8);
    }

    //循环 10 次重复喂狗
    //10 次之后不喂狗，看门狗将复位
    for (i = 0; i < 10; i++)
    {
        delay(1000000);           //  短延时
        IWDG->KR = 0xAAAA;       // reload the watchdog  重装载看门狗定时器

        GPIOC->BRR = 1 << (1 + 8);

        delay(1000000);           //短延时
        IWDG->KR = 0xAAAA;       // reload the watchdog  重装载看门狗定时器
        //LED off LED 熄灭
        GPIOC->BSRR = 1 << (1 + 8);
    }
    //LED on LED 点亮
    GPIOC->BRR = 1 << (ledPosEnd + 1 + 8);

    //不喂狗，等待看门狗复位
    //执行到这里，不喂狗，那么系统复位，如果系统不复位，LED 不再变化，
    //可以看到 LED 停止闪烁一段时间又开始闪烁说明看门狗复位了
    while (1)
    {
        // Loop forever
    } // end while
} // end main
```

BHS-STM32 实验 18-IWDG看门狗(库函数)

本例子实现看门狗功能，程序不喂狗后自动复位

1.该例子必须下载到 FLASH 中运行才能看到效果

2.运行程序时拔掉 JTAG 调试接口

3.main 函数执行到 while (1)这里，不喂狗，那么系统复位，如果系统不复位，LED 不再变化。可以看到 LED 停止闪烁一段时间又开始闪烁说明看门狗复位了

函数 IWDG_WriteAccessCmd

函数名	IWDG_WriteAccessCmd
函数原形	void IWDG_WriteAccessCmd(u16 IWDG_WriteAccess)
功能描述	使能或者失能对寄存器 IWDG_PR 和 IWDG_RLR 的写操作
输入参数	IWDG_WriteAccess：对寄存器 IWDG_PR 和 IWDG_RLR 的写操作的新状态 参阅 Section：IWDG_WriteAccess 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

IWDG_WriteAccess

该参数使能或者失能对寄存器 IWDG_PR 和 IWDG_RLR 的写操作

IWDG_WriteAccess	描述
IWDG_WriteAccess_Enable	使能对寄存器 IWDG_PR 和 IWDG_RLR 的写操作
IWDG_WriteAccess_Disable	失能对寄存器 IWDG_PR 和 IWDG_RLR 的写操作

函数 IWDG_SetPrescaler

函数名	IWDG_SetPrescaler
函数原形	void IWDG_SetPrescaler(u8 IWDG_Prescaler)
功能描述	设置 IWDG 预分频值
输入参数	IWDG_Prescaler：IWDG 预分频值 参阅 Section：IWDG_Prescaler 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

IWDG_Prescaler

该参数设置 IWDG 预分频值

IWDG_Prescaler	描述
IWDG_Prescaler_4	设置 IWDG 预分频值为 4
IWDG_Prescaler_8	设置 IWDG 预分频值为 8
IWDG_Prescaler_16	设置 IWDG 预分频值为 16
IWDG_Prescaler_32	设置 IWDG 预分频值为 32
IWDG_Prescaler_64	设置 IWDG 预分频值为 64
IWDG_Prescaler_128	设置 IWDG 预分频值为 128
IWDG_Prescaler_256	设置 IWDG 预分频值为 256

函数 IWDG_SetReload

函数名	IWDG_ SetReload
函数原形	void IWDG_SetReload(u16 Reload)
功能描述	设置 IWDG 重装载值
输入参数	IWDG_Reload：IWDG 重装载值 该参数允许取值范围为 0 – 0x0FFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

```
int main(void)
{
#ifdef DEBUG
    debug();
#endif

    int i;

    /* System Clocks Configuration */
    RCC_Configuration();//配置系统时钟

    GPIO_Configuration();//配置 GPIO

    /* NVIC configuration */
    NVIC_Configuration();//配置中断

    /* Check if the system has resumed from IWDG reset ----------------------*/
    //检查看门狗复位标志
    if (RCC_GetFlagStatus(RCC_FLAG_IWDGRST) != RESET)
    {
        /* IWDGRST flag set */
        /* Turn on led connected to PC.08 */
        //点亮 LED
        GPIO_WriteBit(GPIOC, GPIO_Pin_8, Bit_RESET);

        /* Clear reset flags */
        //清除复位标志
        RCC_ClearFlag();
    }
    else
    {
        /* IWDGRST flag is not set */
        /* Turn off led connected to PC.08 */
        //熄灭 LED
        GPIO_WriteBit(GPIOC, GPIO_Pin_8, Bit_SET);
```

```
}

/* IWDG timeout equal to 280 ms (the timeout may varies due to LSI frequency
    dispersion) ----------------------------------------------------------*/
/* Enable write access to IWDG_PR and IWDG_RLR registers */
//允许 IWDG 被修改
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);

/* IWDG counter clock: 40KHz(LSI) / 32 = 1.25 KHz */
//设置 IWDG 时钟
IWDG_SetPrescaler(IWDG_Prescaler_32);

/* Set counter reload value to 349 */
//设置定时器重载值
IWDG_SetReload(349);

/* Reload IWDG counter */
//重载 IWDG 计数器
IWDG_ReloadCounter();

/* Enable IWDG (the LSI oscillator will be enabled by hardware) */
//使能 IWDG
IWDG_Enable();

//循环 10 次重复喂狗
//10 次之后不喂狗，看门狗将复位
for (i = 0; i < 10; i++)
{
    delay(1000000);   // 短延时
    IWDG_ReloadCounter();   // reload the watchdog  重装载看门狗定时器

    GPIO_WriteBit(GPIOC, GPIO_Pin_9, Bit_RESET);

    delay(1000000);         // 短延时
    IWDG_ReloadCounter(); //reload the watchdog  重装载看门狗定时器
    //LED off LED 灭
    GPIO_WriteBit(GPIOC, GPIO_Pin_9, Bit_SET);
}
//LED on    LED 亮
GPIO_WriteBit(GPIOC, GPIO_Pin_9, Bit_RESET);

//不喂狗，等待看门狗复位
//执行到这里，不喂狗，那么系统复位，如果系统不复位，LED 不再变化，
//可以看到 LED 停止闪烁一段时间又开始闪烁说明看门狗复位了
while (1)
```

```
    {
        // Loop forever
         ;} // end while
}
```

tyw藏书

RTC实时时钟实验

RTC实时时钟功能描述

实时时钟是一个独立的定时器。RTC 模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。

RTC 模块和时钟配置系统(RCC_BDCR 寄存器)处于后备区域，即在系统复位或从待机模式唤醒后，RTC 的设置和时间维持不变。

系统复位后，对后备寄存器和 RTC 的访问被禁止，这是为了防止对后备区域(BKP)的意外写操作。执行以下操作将使能对后备寄存器和 RTC 的访问：

- 设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位，使能电源和后备接口时钟
- 设置寄存器 PWR_CR 的 DBP 位，使能对后备寄存器和 RTC 的访问。

主要特性

- 可编程的预分频系数：分频系数最高为 220

。

- 32 位的可编程计数器，可用于较长时间段的测量。
- 2 个分离的时钟：用于 APB1 接口的 PCLK1 和 RTC 时钟(RTC 时钟的频率必须小于 PCLK1 时钟频率的四分之一以上)。
- 可以选择以下三种 RTC 的时钟源：

— HSE 时钟除以 128；
— LSE 振荡器时钟；
— LSI 振荡器时钟(详见 6.2.8

308/754

节 RTC 时钟)。

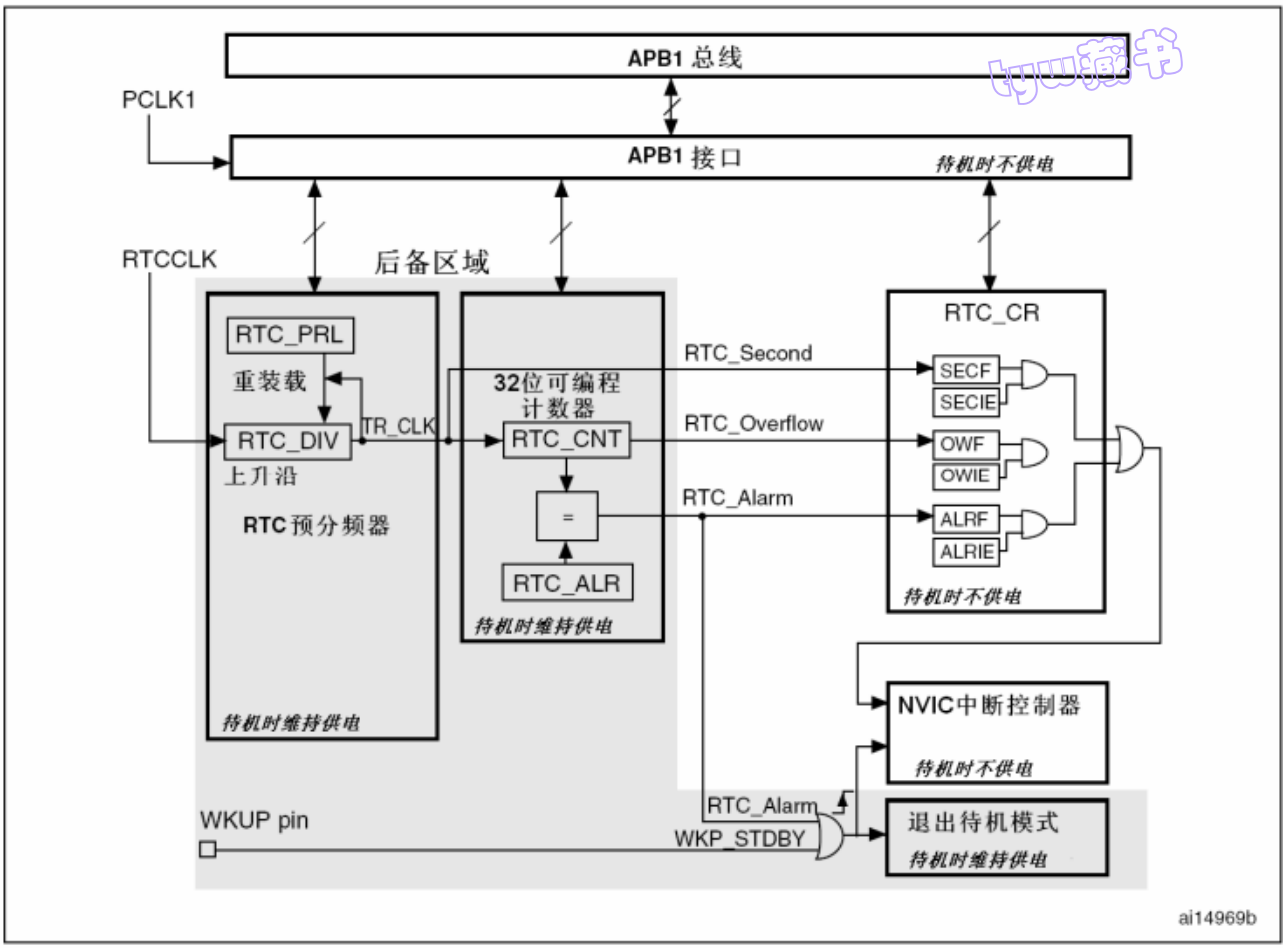
- 2 个独立的复位类型：

— APB1 接口由系统复位；
— RTC 核心(预分频器、闹钟、计数器和分频器)只能由后备域复位(详见 6.1.3 节)。

- 3 个专门的可屏蔽中断：

— 闹钟中断，用来产生一个软件可编程的闹钟中断。
— 秒中断，用来产生一个可编程的周期性中断信号(最长可达 1 秒)。
— 溢出中断，指示内部可编程计数器溢出并回转为 0 的状态

简化的 RTC 框图



RTC 寄存器描述

RTC 控制寄存器高位(RTC_CRH)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留													OWIE	ALRIE	SECIE
													rw	rw	rw

位 15:3	保留，被硬件强制为 0。
位 2	OWIE：允许溢出中断位 (Overflow interrupt enable) 0：屏蔽(不允许)溢出中断 1：允许溢出中断
位 1	ALRIE：允许闹钟中断 (Alarm interrupt enable) 0：屏蔽(不允许)闹钟中断 1：允许闹钟中断
位 0	SECIE：允许秒中断 (Second interrupt enable) 0：屏蔽(不允许)秒中断 1：允许秒中断

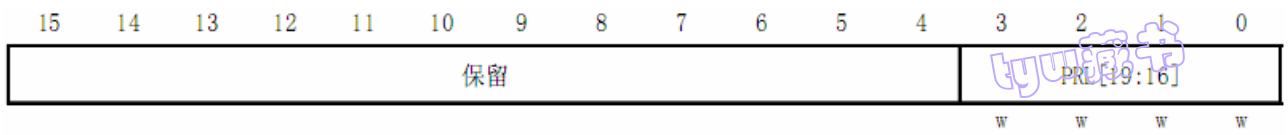
RTC 控制寄存器低位(RTC_CRL)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										RTOFF	CNF	RSF	OWF	ALRF	SECF
										r	rw	rc w0	rc w0	rc w0	rc w0

位 15:6	保留，被硬件强制为 0。

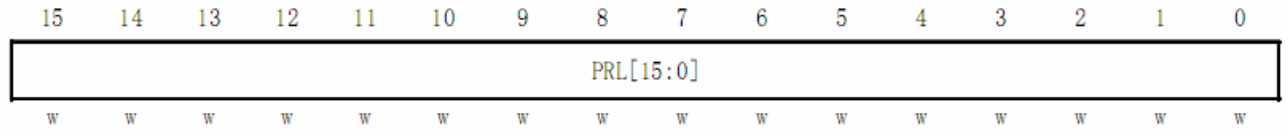
位 5	RTOFF：RTC 操作关闭 (RTC operation OFF) RTC 模块利用这位来指示对其寄存器进行的最后一次操作的状态，指示操作是否完成。若此位 为’0’，则表示无法对任何的 RTC 寄存器进行写操作。此位为只读位。 0：上一次对 RTC 寄存器的写操作仍在进行; 1：上一次对 RTC 寄存器的写操作已经完成。
位 4	CNF：配置标志 (Configuration flag) 此位必须由软件置’1’以进入配置模式，从而允许向 RTC_CNT、RTC_ALR 或 RTC_PRL 寄存器 写入数据。只有当此位在被置’1’并重新由软件清’0’后，才会执行写操作。 0：退出配置模式(开始更新 RTC 寄存器); 1：进入配置模式。
位 3	RSF：寄存器同步标志 (Registers synchronized flag) 每当 RTC_CNT 寄存器和 RTC_DIV 寄存器由软件更新或清’0’时，此位由硬件置’1’。在 APB1 复位 后，或 APB1 时钟停止后，此位必须由软件清’0’。要进行任何的读操作之前，用户程序必须等待 这位被硬件置’1’，以确保 RTC_CNT、RTC_ALR 或 RTC_PRL 已经被同步。 0：寄存器尚未被同步; 1：寄存器已经被同步。
位 2	OWF：溢出标志 (Overflow flag) 当 32 位可编程计数器溢出时，此位由硬件置’1’。如果 RTC_CRH 寄存器中 OWIE=1，则产生中 断。此位只能由软件清’0’。对此位写’1’是无效的。 0：无溢出; 1：32 位可编程计数器溢出。
位 1	ALRF：闹钟标志 (Alarm flag) 当 32 位可编程计数器达到 RTC_ALR 寄存器所设置的预定值，此位由硬件置’1’。如果 RTC_CRH 寄存器中 ALRIE=1，则产生中断。此位只能由软件清’0’。对此位写’1’是无效的。 0：无闹钟; 1：有闹钟。
位 0	SECF：秒标志 (Second flag) 当 32 位可编程预分频器溢出时，此位由硬件置’1’同时 RTC 计数器加 1。因此，此标志为分辨率可 编程的 RTC 计数器提供一个周期性的信号(通常为 1 秒)。如果 RTC_CRH 寄存器中 SECIE=1，则 产生中断。此位只能由软件清除。对此位写’1’是无效的。 0：秒标志条件不成立; 1：秒标志条件成立。

RTC 预分频装载寄存器高位(RTC_PRLH)



位 15:4	保留，被硬件强制为 0。
位 3:0	PRL[19:16]：RTC 预分频装载值高位 (RTC prescaler reload value high) 根据以下公式，这些位用来定义计数器的时钟频率： fTR_CLK = fRTCCLK/(PRL[19:0]+1) 注：不推荐使用 0 值，否则无法正确的产生 RTC 中断和标志位。

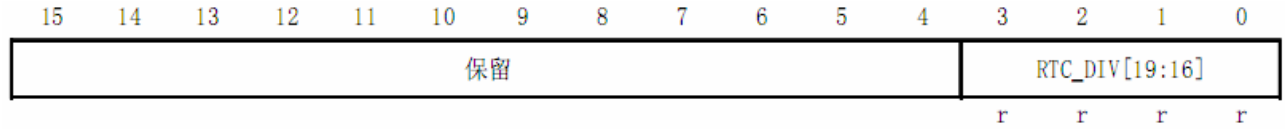
RTC 预分频装载寄存器低位(RTC_PRLL)



位 15:0	PRL[15:0]：RTC 预分频装载值低位 根据以下公式，这些位用来定义计数器的时钟频率： fTR_CLK = fRTCCLK/(PRL[19:0]+1)

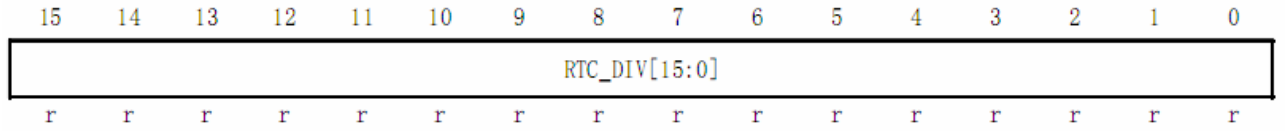
注：如果输入时钟频率是 32.768kHz(fRTCCLK)，这个寄存器中写入 7FFFh 可获得周期为 1 秒钟的信号。

RTC 预分频器余数寄存器高位(RTC_DIVH)



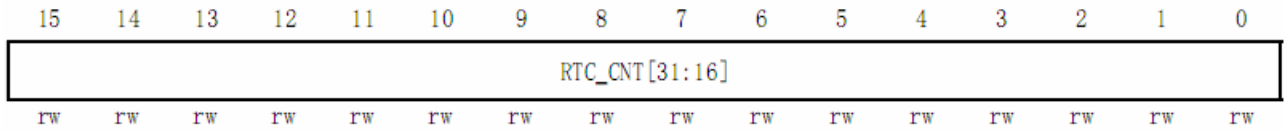
位15:4	保留
位3:0	**RTC_DIV[19:16]**：RTC时钟分频器余数高位 (RTC clock divider high)

RTC 预分频器余数寄存器低位(RTC_DIVL)



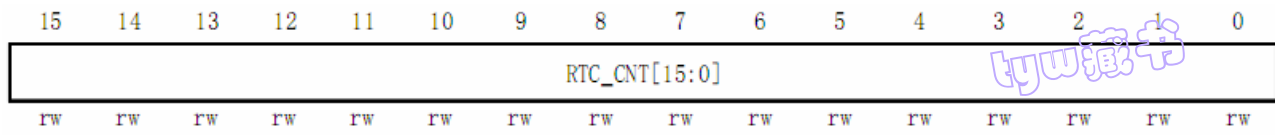
位15:0	**RTC_DIV[15:0]**：RTC时钟器余数低位 (RTC clock divider low)

RTC 计数器寄存器高位(RTC_CNTH)



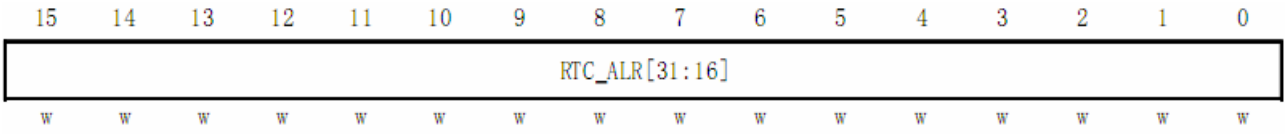
位15:0	**RTC_CNT[31:16]**：RTC计数器高位 (RTC counter high) 可通过读RTC_CNTH寄存器来获得RTC计数器当前值的高位部分。要对此寄存器进行写操作前，必须先进入配置模式(参见16.3.4节)。

RTC 计数器寄存器低位(RTC_CNTL)



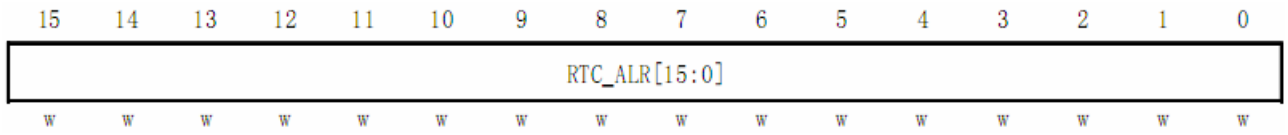
位15:0	**RTC_CNT[15:0]**：RTC计数器低位。 可通过读RTC_CNTL寄存器来获得RTC计数器当前值的低位部分。要对此寄存器进行写操作，必须先进入配置模式(参见16.3.4节)。

RTC 闹钟寄存器高位(RTC_ALRH)



位15:0	**RTC_ALR[31:16]**：RTC闹钟值高位 (RTC alarm high) 此寄存器用来保存由软件写入的闹钟时间的高位部分。要对此寄存器进行写操作，必须先进入配置模式(参见16.3.4节)。

RTC 闹钟寄存器低位(RTC_ALRL)



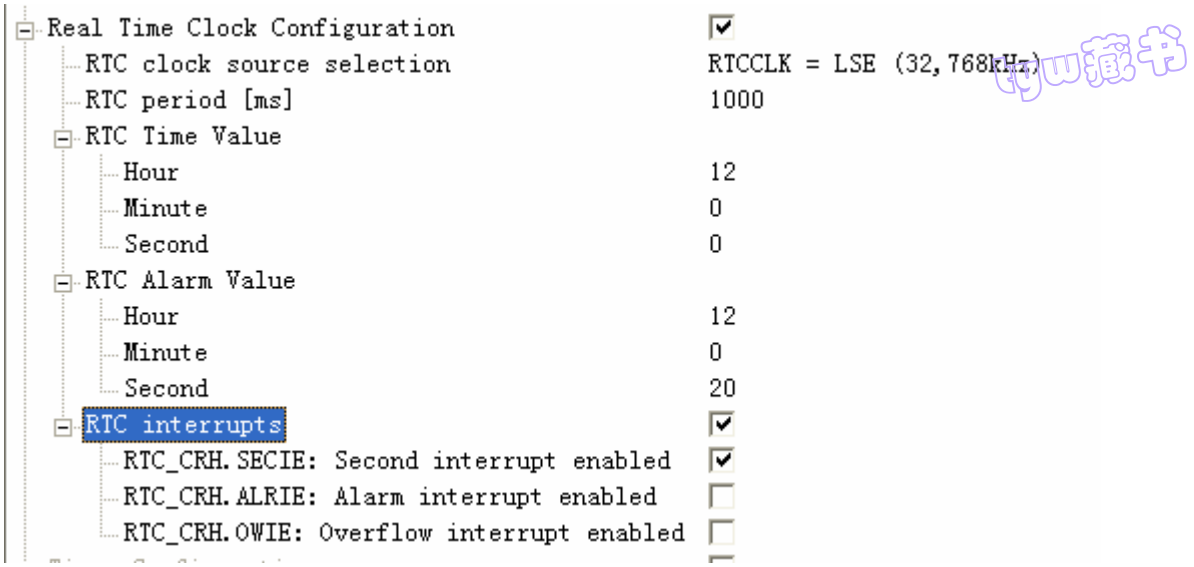
位15:0	**RTC_ALR[15:0]**：RTC闹钟值低位 (RTC alarm low) 此寄存器用来保存由软件写入的闹钟时间的低位部分。要对此寄存器进行写操作，必须先进入配置模式(参见16.3.4节)。

BHS-STM32 实验 19-RTC实时时钟(直接操作寄存器)

该例子用 RTC 产生 1 秒中断，驱动 LED 闪烁

STM32 的 RTC 实际是一个 32 位的计数器，要得到时分秒信号需要自己转换，可以参考我 BHS-GUI 里面的例子.

RTC 实时时钟配置




```
/*-------------------------------------------------------------------------
   RTC Interrupt Handler
   RTC 中断函数
  *-------------------------------------------------------------------------*/
void RTC_IRQHandler(void)
{
    if (RTC->CRL & (1 << 0))
    {
        // check second flag/检查是否是秒中断
        RTC->CRL &= ~(1 << 0);
        if ((ledRtcSec ^= 1) == 0)
            GPIOC->BSRR = 1 << (ledPosSec + 8);//LED off LED 熄灭
        else
            GPIOC->BRR = 1 << (ledPosSec + 8);//LED on LED 点亮

    }
} // end RTC_IRQHandler


/*-------------------------------------------------------------------------
   MAIN function
  *-------------------------------------------------------------------------*/
int main(void)
{
    // STM32 setup
    //STM32 初始化
    stm32_Init();

    while (1)
    {
        // Loop forever
```

```
    ;} // end while
} // end main
```

例程是在中断中点亮/熄灭 LED 的

tyw藏书

BHS-STM32 实验 20-RTC实时时钟(库函数)

该例子用 RTC 产生 1 秒中断，驱动 LED 闪烁

STM32 的 RTC 实际是一个 32 位的计数器，要得到时分秒信号需要自己转换，可以参考我 BHS-GUI 里面的例子.

```
//RTC 初始化
void RTC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

   //开启 GPIO 时钟
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC , ENABLE);

   /* Enable PWR and BKP clocks */
   //允许 RTC 使用到的 PWR,BKP 时钟
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

   /* Allow access to BKP Domain */
   //允许修改备份寄存器
   PWR_BackupAccessCmd(ENABLE);

   /* Reset Backup Domain */
   //复位备份域
   BKP_DeInit();

   /* Enable LSE */
   //使能外部 LSE(32768)时钟
   RCC_LSEConfig(RCC_LSE_ON);
   /* Wait till LSE is ready */
   //等待 LSE(32768)时钟稳定
   while(RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
   {
   }

   /* Select LSE as RTC Clock Source */
   //选择 LSE(32768)做 RTC 时钟源
   RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

   /* Enable RTC Clock */
   //使能 RTC
   RCC_RTCCLKCmd(ENABLE);
```

tyw藏书

```
    /* Wait for RTC registers synchronization */
    //等待 RTC 寄存器同步
    RTC_WaitForSynchro();

    /* Wait until last write operation on RTC registers has finished */
    //等待 RTC 寄存器最后一次写操作已完成
    RTC_WaitForLastTask();

    /* Enable the RTC Second */
    //使能 RTC 秒信号中断发生
    RTC_ITConfig(RTC_IT_SEC, ENABLE);

    /* Wait until last write operation on RTC registers has finished */
    //等待 RTC 寄存器最后一次写操作已完成
    RTC_WaitForLastTask();

    /* Set RTC prescaler: set RTC period to 1sec */
    //设置 RTC 分频
    RTC_SetPrescaler(32767); /* RTC period = RTCCLK/RTC_PR = (32.768 KHz)/(32767+1) */

    /* Wait until last write operation on RTC registers has finished */
    //等待 RTC 寄存器最后一次写操作已完成
    RTC_WaitForLastTask();

    //RTC 中断配置
    /* Configure one bit for preemption priority */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    /* Enable the RTC Interrupt */
    //配置 RTC 中断
    NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
/*******************************************************************************
RTC_IRQHandler
RTC 中断函数
*******************************************************************************/
void RTC_IRQHandler(void)
{
    // 判断是否是 RTC 秒中断
    if (RTC_GetITStatus(RTC_IT_SEC) != RESET)
```

```
    {
        /* Clear the RTC Second interrupt */
        //清除秒中断标志
        RTC_ClearITPendingBit(RTC_IT_SEC);

        /* Toggle GPIO_LED pin 8 each 1s */
        //GPIO 状态翻转，循环点亮/熄灭 LED
        GPIO_WriteBit(GPIOC, GPIO_Pin_8, (BitAction)(1 - GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_8)));

        /* Wait until last write operation on RTC registers has finished */
        //等待 RTC 寄存器最后一次写操作已完成
        RTC_WaitForLastTask();

        /* Reset RTC Counter when Time is 23:59:59 */
        //秒计数器到 23:59:59(86400 秒)归 0
        if (RTC_GetCounter() == 0x00015180)
        {
            RTC_SetCounter(0x0);
            /* Wait until last write operation on RTC registers has finished */
            //等待 RTC 寄存器最后一次写操作已完成
            RTC_WaitForLastTask();
        }
    }
}
```

Tamper侵入检测实验

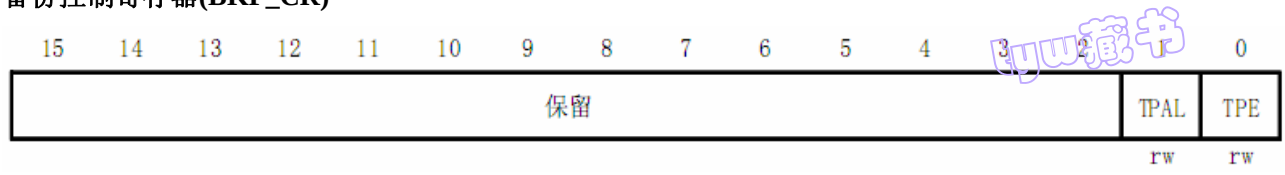
Tamper侵入检测功能描述

PC13 可以作为通用 I/O 口、TAMPER 引脚、RTC 校准时钟、RTC 闹钟或秒输出
当 TAMPER 引脚上的信号从 0 变成 1 或者从 1 变成 0(取决于备份控制寄存器 BKP_CR 的 TPAL 位)，会产生一个侵入检测事件。侵入检测事件将所有数据备份寄存器内容清除。
然而为了避免丢失侵入事件，侵入检测信号是边沿检测的信号与侵入检测允许位的逻辑与，从而在侵入检测引脚被允许前发生的侵入事件也可以被检测到。

- 当 TPAL=0 时：如果在启动侵入检测 TAMPER 引脚前(通过设置 TPE 位)该引脚已经为高电平，一旦启动侵入检测功能，则会产生一个额外的侵入事件(尽管在 TPE 位置’1’后并没有出现上升沿)。
- 当 TPAL=1 时：如果在启动侵入检测引脚 TAMPER 前(通过设置 TPE 位)该引脚已经为低电平，一旦启动侵入检测功能，则会产生一个额外的侵入事件(尽管在 TPE 位置’1’后并没有出现下降沿)。

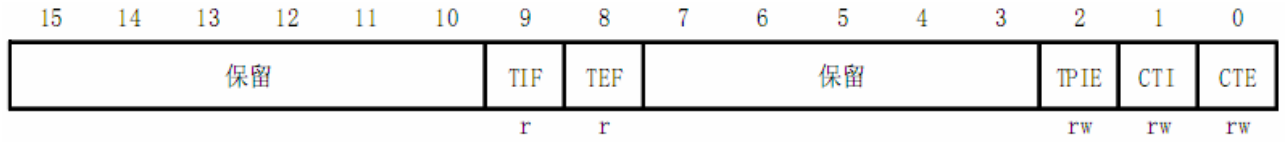
设置 BKP_CSR 寄存器的 TPIE 位为’1’，当检测到侵入事件时就会产生一个中断。
在一个侵入事件被检测到并被清除后，侵入检测引脚 TAMPER 应该被禁止。然后，在再次写入备份数据寄存器前重新用 TPE 位启动侵入检测功能。这样，可以阻止软件在侵入检测引脚上仍然有侵入事件时对备份数据寄存器进行写操作。这相当于对侵入引脚 TAMPER 进行电平检测。
注：当 VDD 电源断开时，侵入检测功能仍然有效。为了避免不必要的复位数据备份寄存器，TAMPER 引脚应该在片外连接到正确的电平。

备份控制寄存器(BKP_CR)



位15:2	保留，始终读为0。
位1	**TPAL**：侵入检测TAMPER引脚有效电平(TAMPER pin active level) 0：侵入检测TAMPER引脚上的高电平会清除所有数据备份寄存器(如果TPE位为1) 1：侵入检测TAMPER引脚上的低电平会清除所有数据备份寄存器(如果TPE位为1)
位0	**TPE**：启动侵入检测TAMPER引脚(TAMPER pin enable) 0：侵入检测TAMPER引脚作为通用IO口使用 1：开启侵入检测引脚作为侵入检测使用

备份控制/状态寄存器(BKP_CSR)



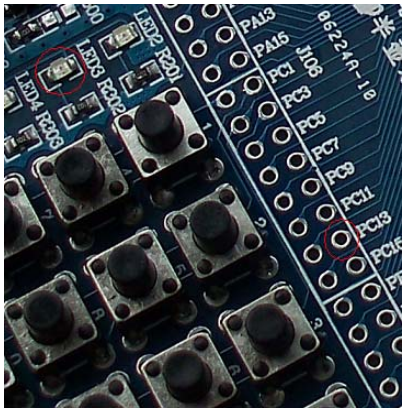
位 15:10	保留，始终读为 0。
位 9	TIF：侵入中断标志(Tamper interrupt flag) 当检测到有侵入事件且 TPIE 位为 1 时，此位由硬件置 1。通过向 CTI 位写 1 来清除此标志位(同时 也清除了中断)。如果 TPIE 位被清除，则此位也会被清除。 0：无侵入中断 1：产生侵入中断 注意：仅当系统复位或由待机模式唤醒后才复位该位
位 8	TEF：侵入事件标志(Tamper event flag) 当检测到侵入事件时此位由硬件置 1。通过向 CTE 位写 1 可清除此标志位 0：无侵入事件 1：检测到侵入事件 注：侵入事件会复位所有的 BKP_DRx 寄存器。只要 TEF 为 1，所有的 BKP_DRx 寄存器就一直保 持复位状态。当此位被置 1 时，若对 BKP_DRx 进行写操作，写入的值不会被保存。
位 7:3	保留，始终读为 0。
位 2	TPIE：允许侵入 TAMPER 引脚中断(TAMPER pin interrupt enable) 0：禁止侵入检测中断 1：允许侵入检测中断(BKP_CR 寄存器的 TPE 位也必须被置 1) 注 1：侵入中断无法将系统内核从低功耗模式唤醒。 注 2：仅当系统复位或由待机模式唤醒后才复位该位。
位 1	CTI：清除侵入检测中断(Clear tamper interrupt) 此位只能写入，读出值为 0。 0：无效 1：清除侵入检测中断和 TIF 侵入检测中断标志
位 0	CTE：清除侵入检测事件(Clear tamper event) 此位只能写入，读出值为 0。

	0：无效 1：清除 TEF 侵入检测事件标志(并复位侵入检测器)。

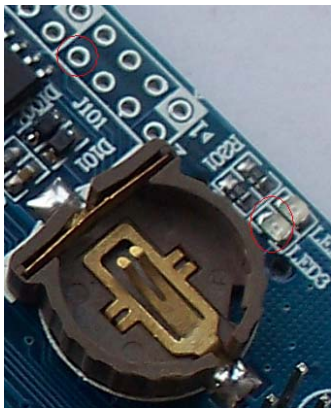
lyw藏书

BHS-STM32 实验 21-Tamper侵入检测(直接操作寄存器)

本例子实现侵入检测功能，该功能脚位于 PC13 上，可设置为低电平/高电平触发
程序正常运行时状态 LED 全灭，当检测到侵入时(图中红圈脚 PC13)LED3(红框内) 点亮
手拿镊子碰触 PC13，LED3 点亮



BSH-STM32-V



BSH-STM32-V 精华版

侵入检测配置如下：

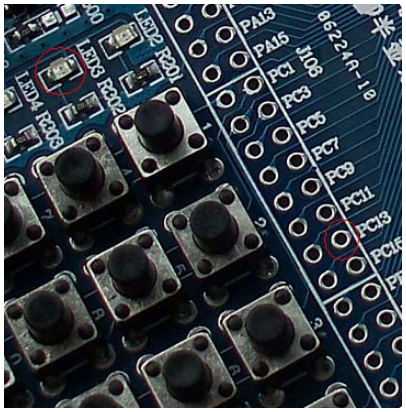
详细代码

```
/*-------------------------------------------------------------------------
   TAMPER Interrupt Handler
 *-------------------------------------------------------------------------*/
void TAMPER_IRQHandler(void)
{
    if (BKP->CSR & (1 << 9))
    {
        // enable access to RTC, BDC registers
        //允许修改 RTC，备份寄存器
        PWR->CR |= (1 << 8);
        //清除中断标志
        BKP->CSR |= (1 << 1);                    // clear Tamper Interrupt
        //清除 RTC 事件
        BKP->CSR |= (1 << 0);                    // clear tamper Event
        // disable access to RTC, BDC registers
        //禁止修改 RTC，备份寄存器
        PWR->CR &= ~(1 << 8);
        GPIOC->BRR = (1 << 9); //LED off LED 点亮
    }
```

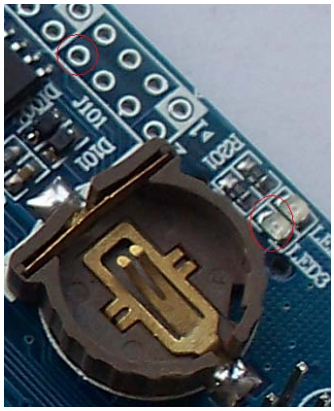
```
}
/*---------------------------------------------------------------------------
   MAIN function
 *---------------------------------------------------------------------------*/
int main(void)
{
    // STM32 setup
    //STM32 初始化
    stm32_Init();

    //关闭所有 LED
       GPIOC->ODR |= 0x00000F00;
    // enable access to RTC, BDC registers
    //允许修改 RTC，备份寄存器
    PWR->CR |= (1 << 8);
    // fill BKP_DR1 register
    //修改备份寄存器的值
    BKP->DR1 = 0x55AA;
    // fill BKP_DR2 register
    //修改备份寄存器的值
    BKP->DR2 = 0x33CC;
    // disable access to RTC, BDC registers
    //禁止修改 RTC，备份寄存器
    PWR->CR &= ~(1 << 8);
    while (1)
    { // Loop forever
         ;} // end while
} // end main
```

BHS-STM32 实验 22-Tamper侵入检测(库函数)

本例子实现侵入检测功能，该功能脚位于 PC13 上，可设置为低电平/高电平触发
程序正常运行时状态 LED 全灭，当检测到侵入时(图中红圈脚 PC13)LED3(红框内) 点亮
手拿镊子碰触 PC13，LED3 点亮



BSH-STM32-V



BSH-STM32-V 精华版

tyw藏书

```
//侵入检测初始化
void Tamper_Configuration(void)
{
NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable PWR and BKP clock  使能备份寄存器时钟*/
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

  /* Enable write access to Backup domain */
  //允许修改备份寄存器
  PWR_BackupAccessCmd(ENABLE);

  /* Clear Tamper pin Event(TE) pending flag  清除标志*/
  BKP_ClearFlag();

  /* Tamper pin active on low level  引脚低电平触发*/
  BKP_TamperPinLevelConfig(BKP_TamperPinLevel_Low);

  /* Enable Tamper interrupt  使能中断*/
  BKP_ITConfig(ENABLE);

  /* Enable Tamper pin  使能管脚*/
  BKP_TamperPinCmd(ENABLE);

  //配置中断
  /* Enable TAMPER IRQChannel */
  NVIC_InitStructure.NVIC_IRQChannel = TAMPER_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
/*******************************************************************************
侵入检测中断函数
*******************************************************************************/
void TAMPER_IRQHandler(void)
{
  //判断中断标志
  if(BKP_GetITStatus() != RESET)
  { /* Tamper detection event occured */
    //
    GPIO_ResetBits(GPIOC, GPIO_Pin_9);

    /* Clear Tamper pin interrupt pending bit */
```

```
    //清除中断标志
    BKP_ClearITPendingBit();

    /* Clear Tamper pin Event(TE) pending flag */
    //清除事件标志
    BKP_ClearFlag();
  }
}
```

lyw藏书

PWM实验

PWM功能描述

脉冲宽度调制模式可以产生一个由 TIMx_ARR 寄存器确定频率、由 TIMx_CCRx 寄存器确定占空比的信号。

在 TIMx_CCMRx 寄存器中的 OCxM 位写入’110’(PWM 模式 1)或’111’(PWM 模式 2)，能够独立地设置每个 OCx 输出通道产生一路 PWM。必须通过设置 TIMx_CCMRx 寄存器的 OCxPE 位使能相应的预装载寄存器，最后还要设置 TIMx_CR1 寄存器的 ARPE 位，(在向上计数或中心对称模式中)使能自动重装载的预装载寄存器。

仅当发生一个更新事件的时候，预装载寄存器才能被传送到影子寄存器，因此在计数器开始计数之前，必须通过设置 TIMx_EGR 寄存器中的 UG 位来初始化所有的寄存器。

OCx 的极性可以通过软件在 TIMx_CCER 寄存器中的 CCxP 位设置，它可以设置为高电平有效或低电平有效。OCx 的输出使能通过(TIMx_CCER 和 TIMx_BDTR 寄存器中)CCxE、CCxNE、MOE、OSSI 和 OSSR 位的组合控制。详见 TIMx_CCER 寄存器的描述。

在 PWM 模式(模式 1 或模式 2)下，TIMx_CNT 和 TIMx_CCRx 始终在进行比较，(依据计数器的计数方向)以确定是否符合 TIMx_CCRx≤TIMx_CNT 或者 TIMx_CNT≤TIMx_CCRx。

根据 TIMx_CR1 寄存器中 CMS 位的状态，定时器能够产生边沿对齐的 PWM 信号或中央对齐的 PWM 信号。

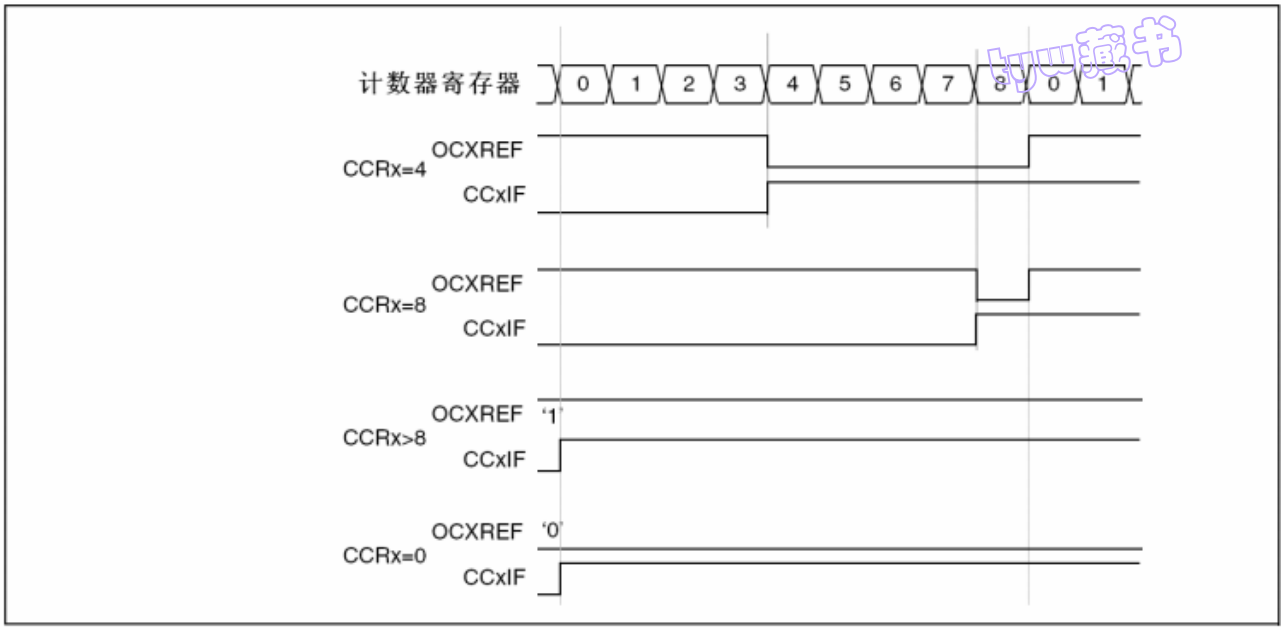
PWM 边沿对齐模式

- **向上计数配置**

当 TIMx_CR1 寄存器中的 DIR 位为低的时候执行向上计数。参看

下面是一个 PWM 模式 1 的例子。当 TIMx_CNT<TIMx_CCRx 时，PWM 参考信号 OCxREF 为高，否则为低。如果 TIMx_CCRx 中的比较值大于自动重装载值(TIMx_ARR)，则 OCxREF 保持为’1’。如果比较值为 0，则 OCxREF 保持为’0’。

下图为 TIMx_ARR=8 时边沿对齐的 PWM 波形。



- **向下计数的配置**

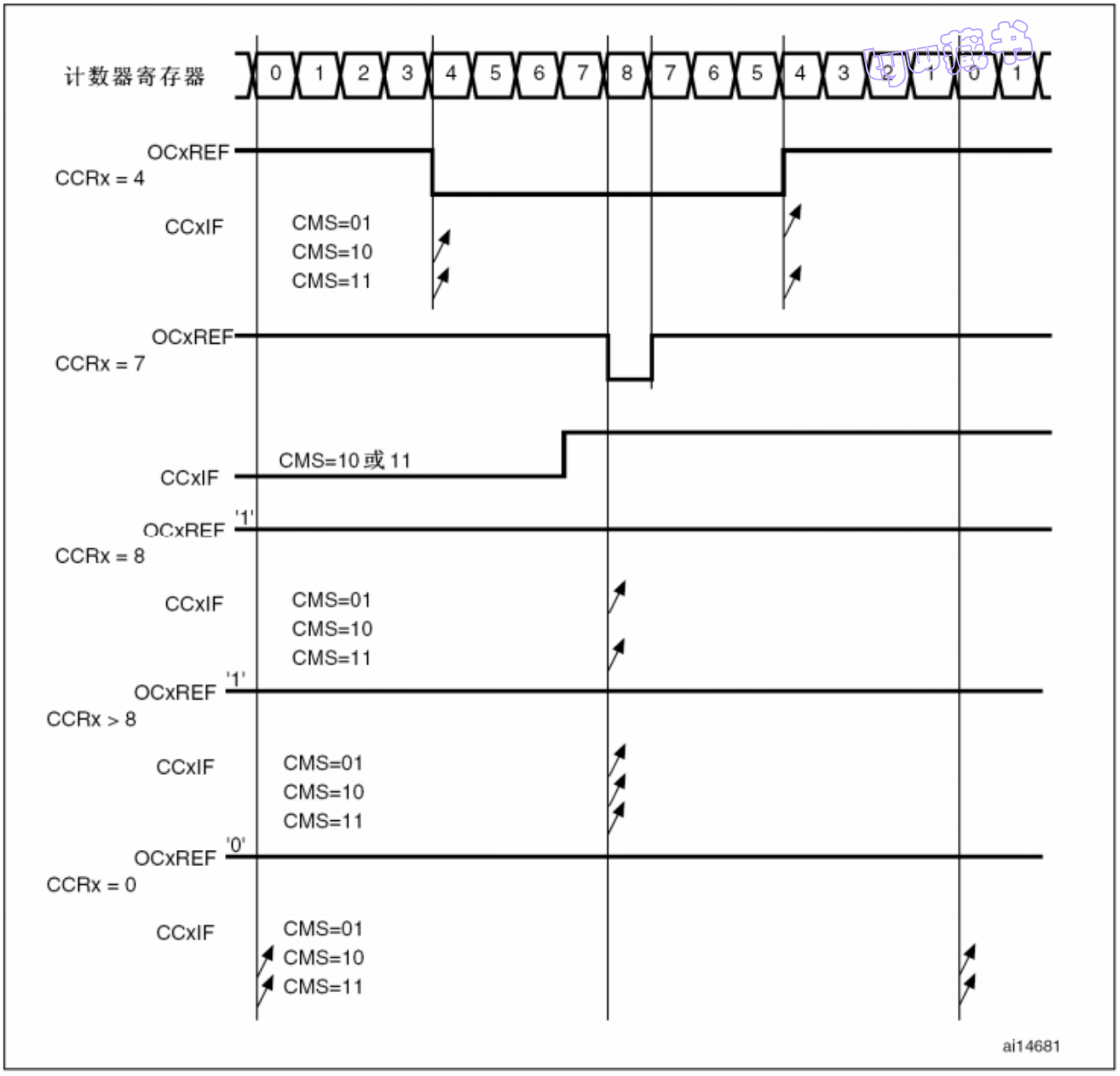
当 TIMx_CR1 寄存器的 DIR 位为高时执行向下计数。参看
在 PWM 模式 1，当 TIMx_CNT>TIMx_CCRx 时参考信号 OCxREF 为低，否则为高。如果 TIMx_CCRx 中的比较值大于 TIMx_ARR 中的自动重装载值，则 OCxREF 保持为’1’。该模式下不能产生 0%的 PWM 波形。

PWM 中央对齐模式

当 TIMx_CR1 寄存器中的 CMS 位不为’00’时为中央对齐模式(所有其他的配置对 OCxREF/OCx 信号都有相同的作用)。根据不同的 CMS 位设置，比较标志可以在计数器向上计数时被置 1、在计数器向下计数时被置 1、或在计数器向上和向下计数时被置 1。TIMx_CR1 寄存器中的计数方向位(DIR)由硬件更新，不要用软件修改它。参看 13.3.2 节的中央对齐模式。
下图给出了一些中央对齐的 PWM 波形的例子

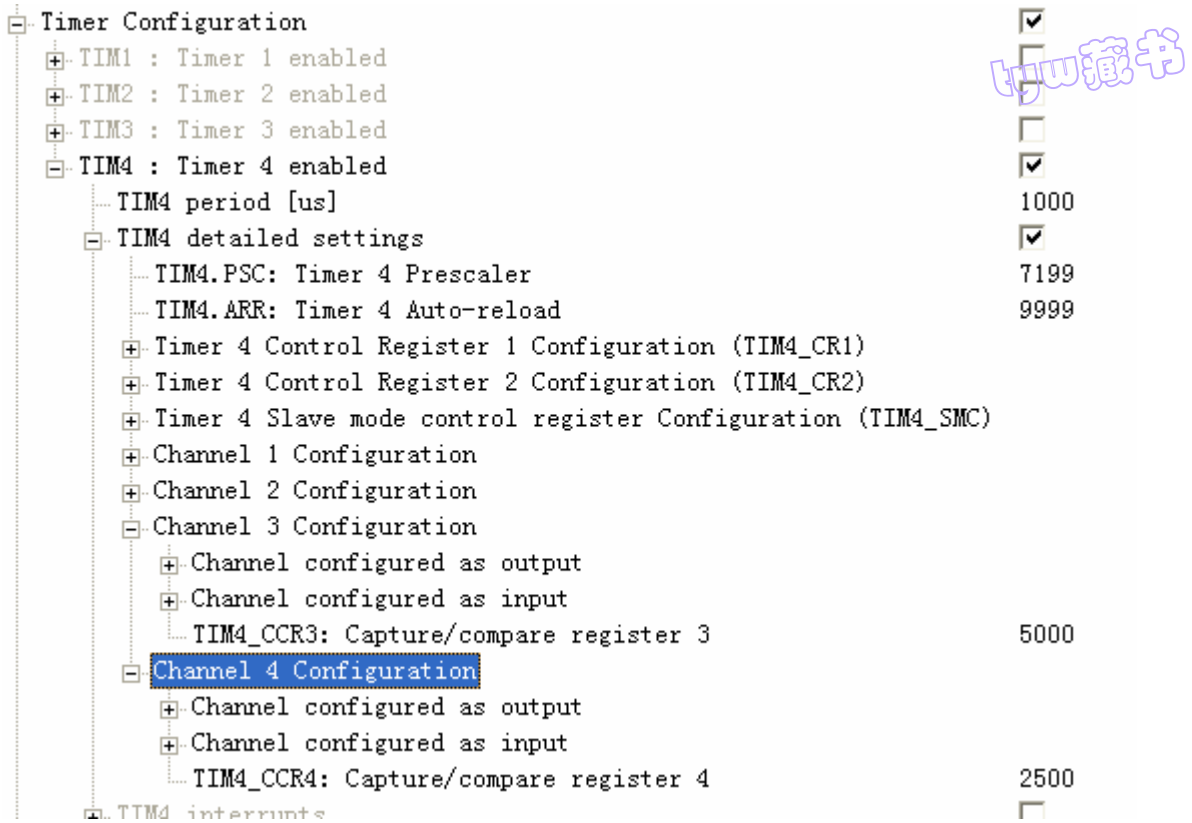
- TIMx_ARR=8
- PWM 模式 1
- TIMx_CR1 寄存器的 CMS=01，在中央对齐模式 1 下，当计数器向下计数时设置比较标志。

中央对齐的 PWM 波形(APR=8)



BHS-STM32 实验 23-PWM_1 固定占空比(直接操作寄存器)

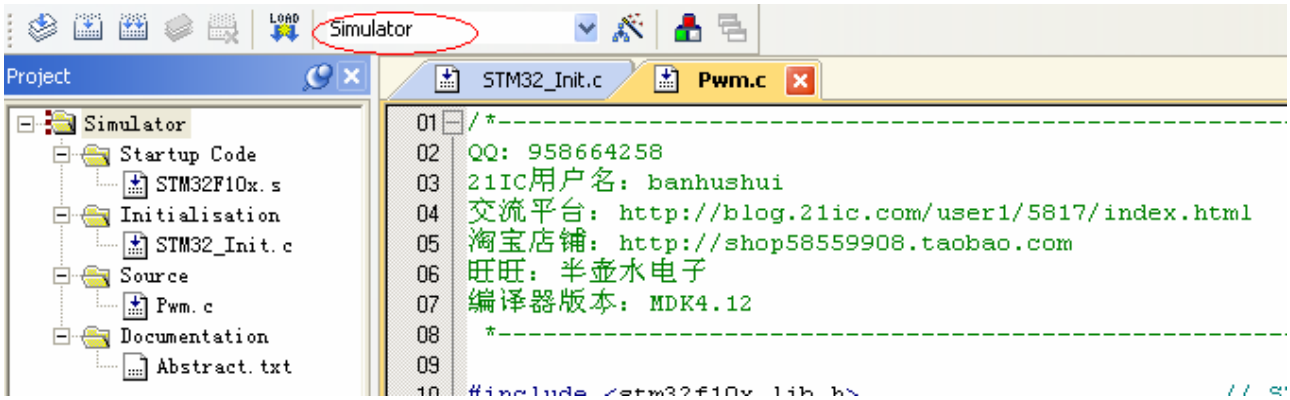
例子使用 TIM4,的通道 3，通道 4 产生 2 个固定占空比的 PWM 脉冲，其中
通道 3（PB8）产生占空比 50% 的 PWM 脉冲
通道 4（PB9）产生占空比 25% 的 PWM 脉冲
本例最好使用示波器观察不同的占空比波形，也可以软件仿真看结果
配置如下

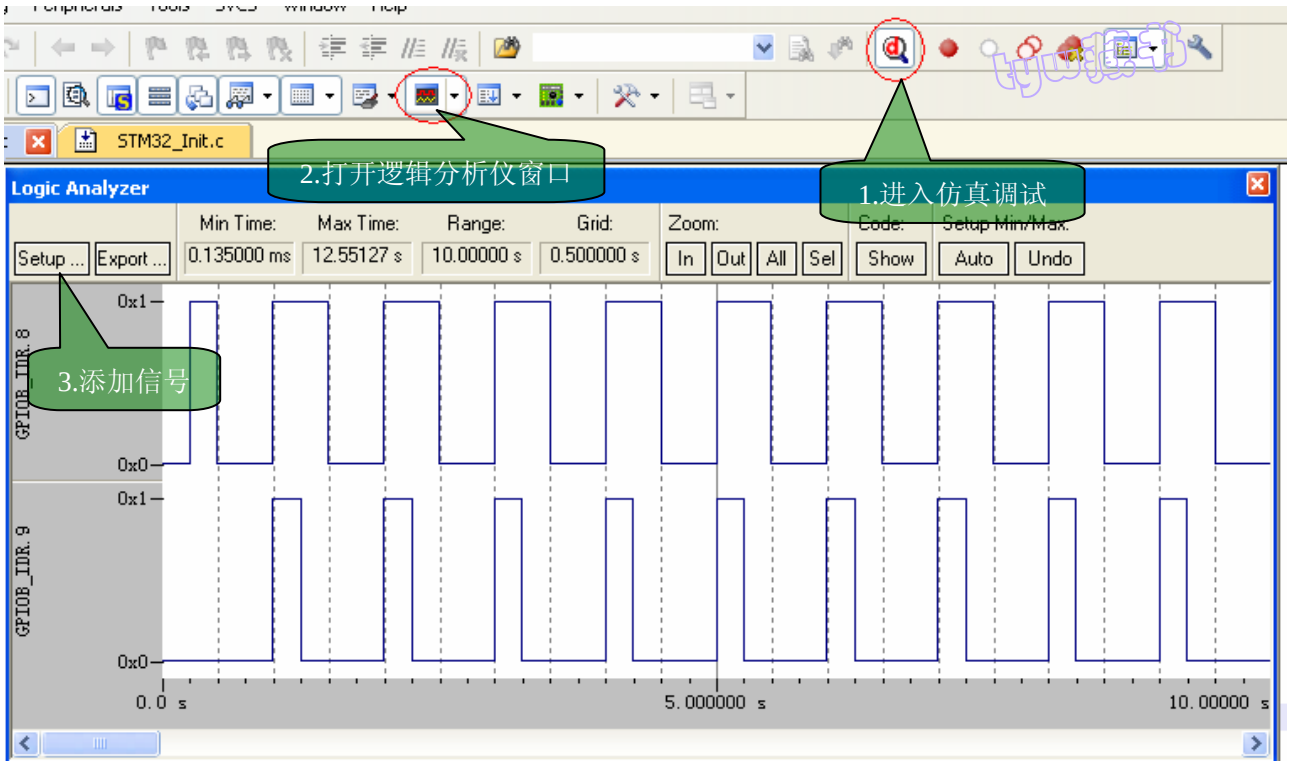


软件仿真：

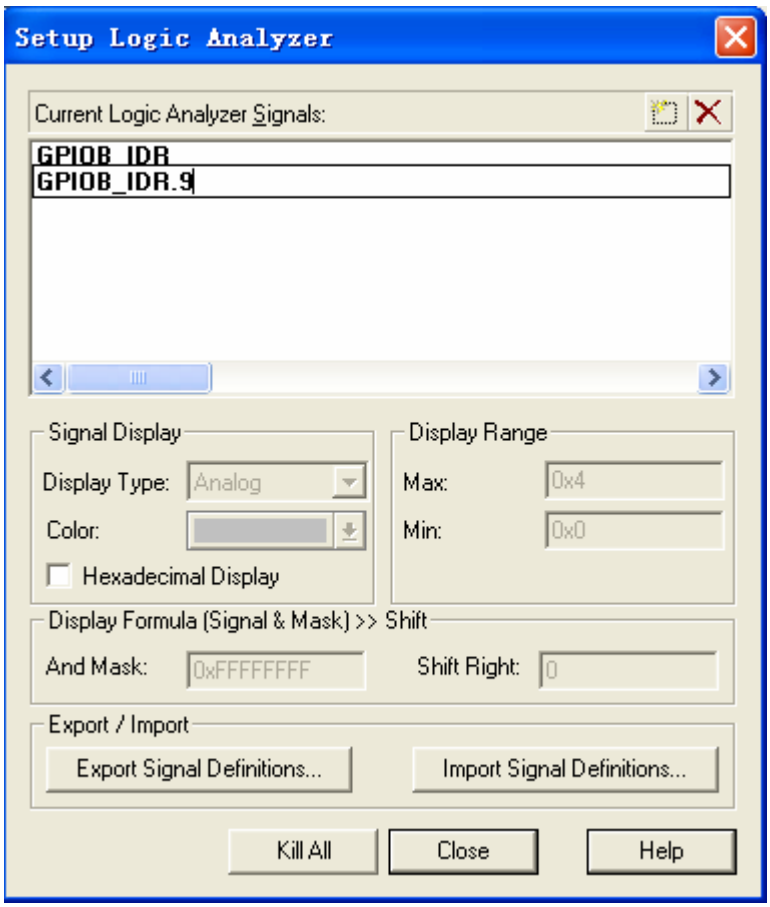
PWM 实验也是可以使用软件仿真在逻辑分析仪窗口中查看波形，选择软件仿真

选择软件仿真

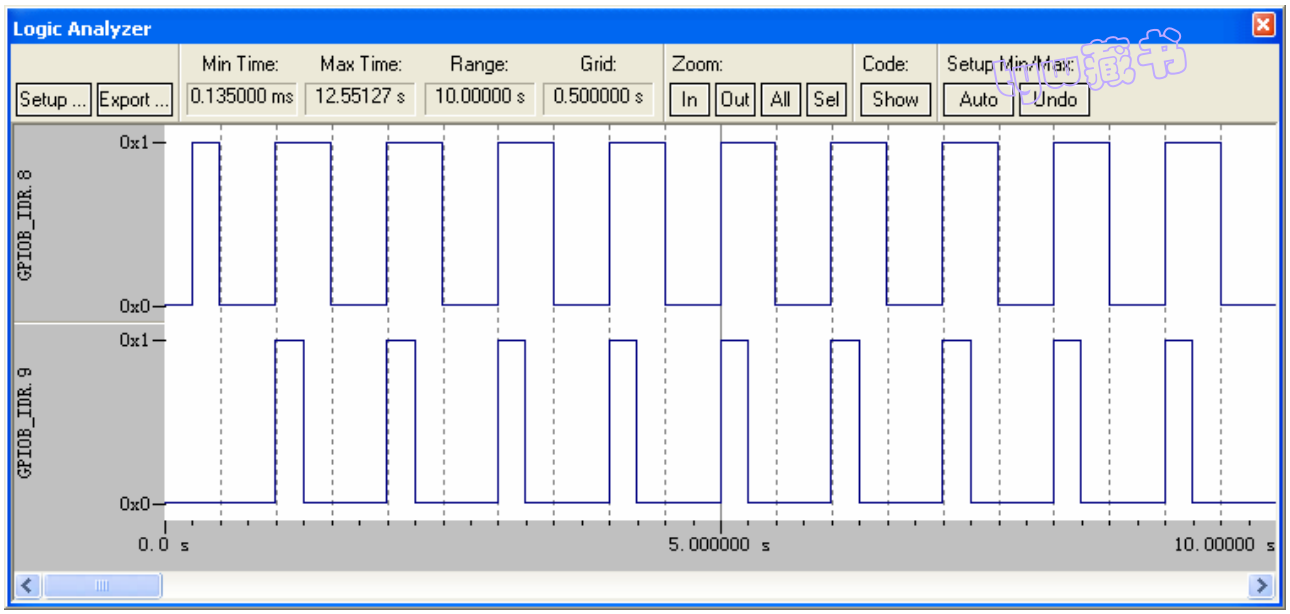




添加查看的信号



全速运行看到下面仿真波形



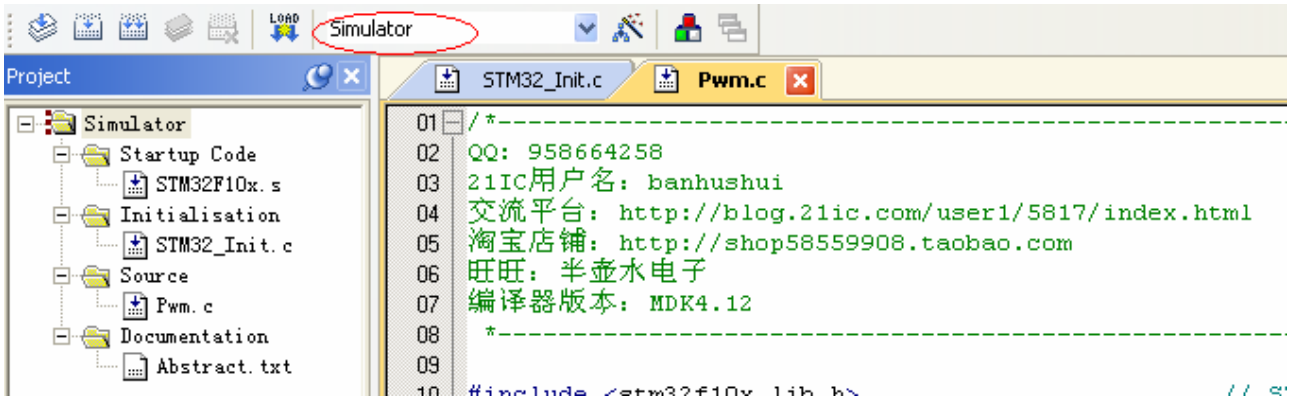
BHS-STM32 实验 24-PWM_1 固定占空比(库函数)

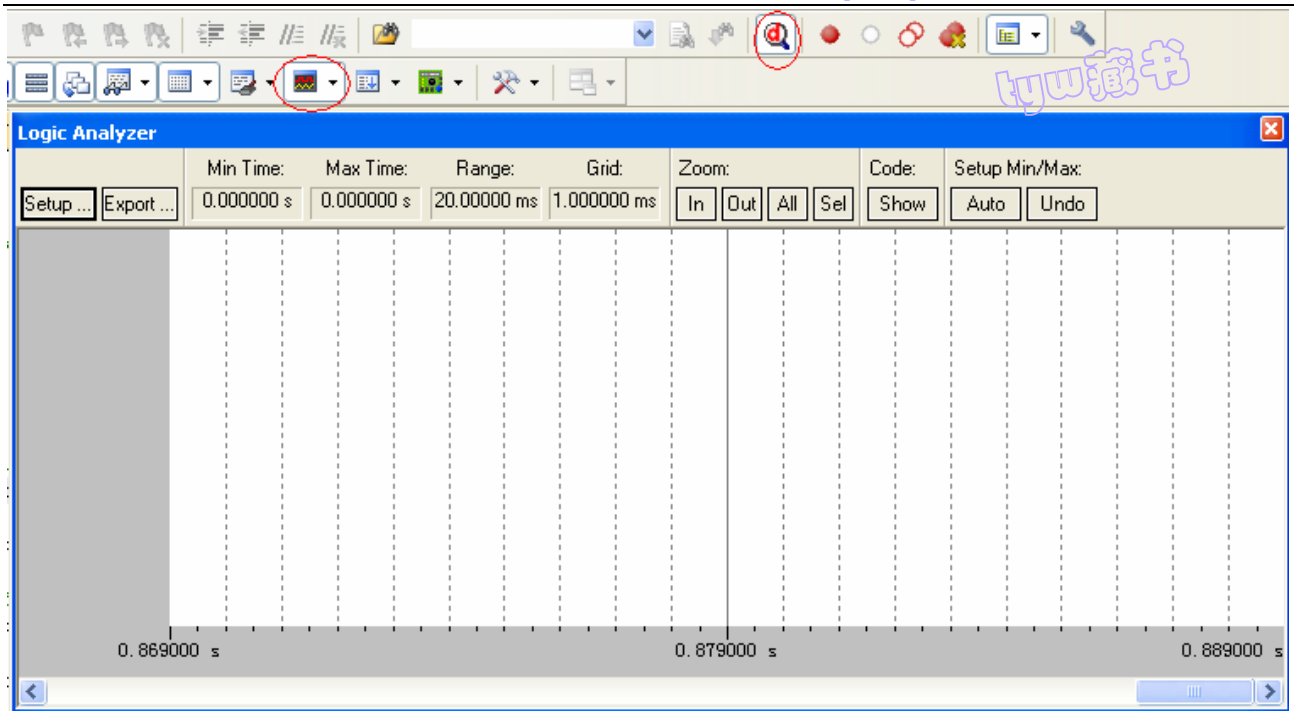
本例子在 PA8 上输出固定占空比的 PWM 波形，本例最好使用示波器观察不同的占空比波形

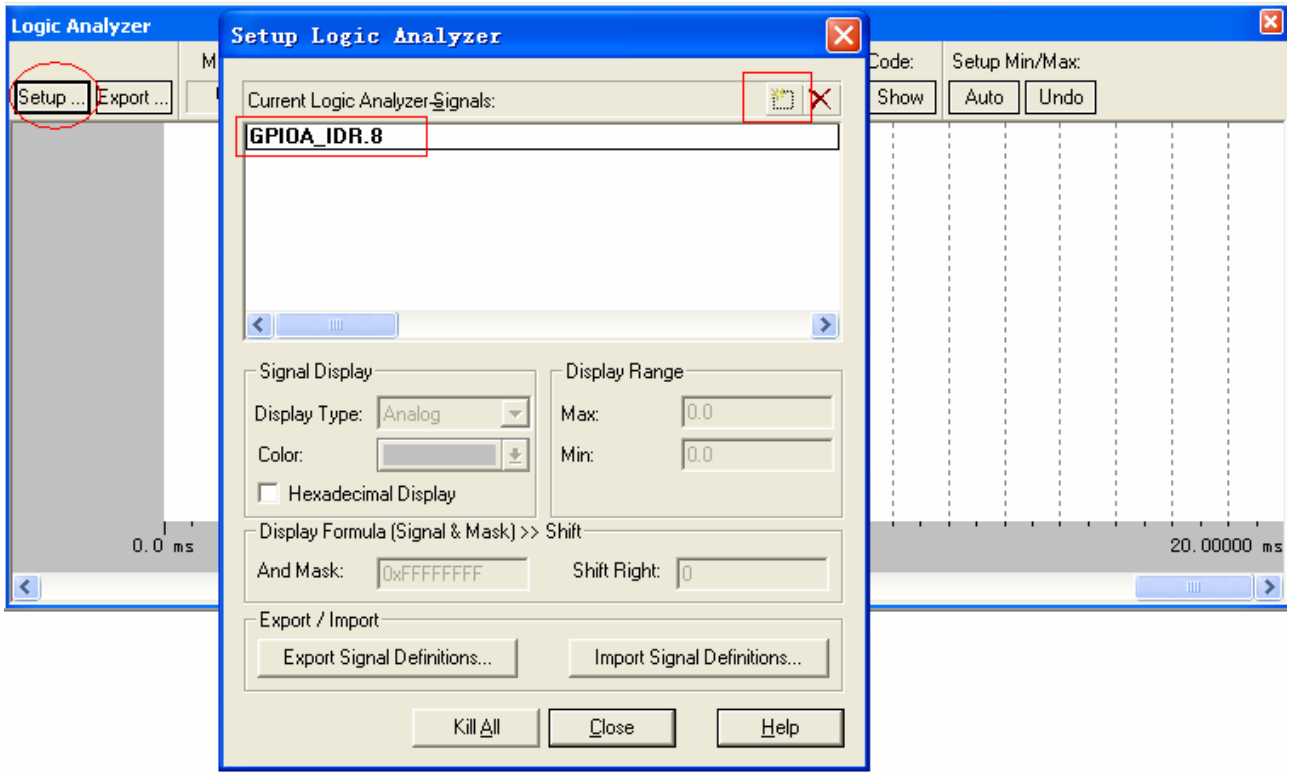
软件仿真：

PWM 实验也是可以使用软件仿真在逻辑分析仪窗口中查看波形，选择软件仿真

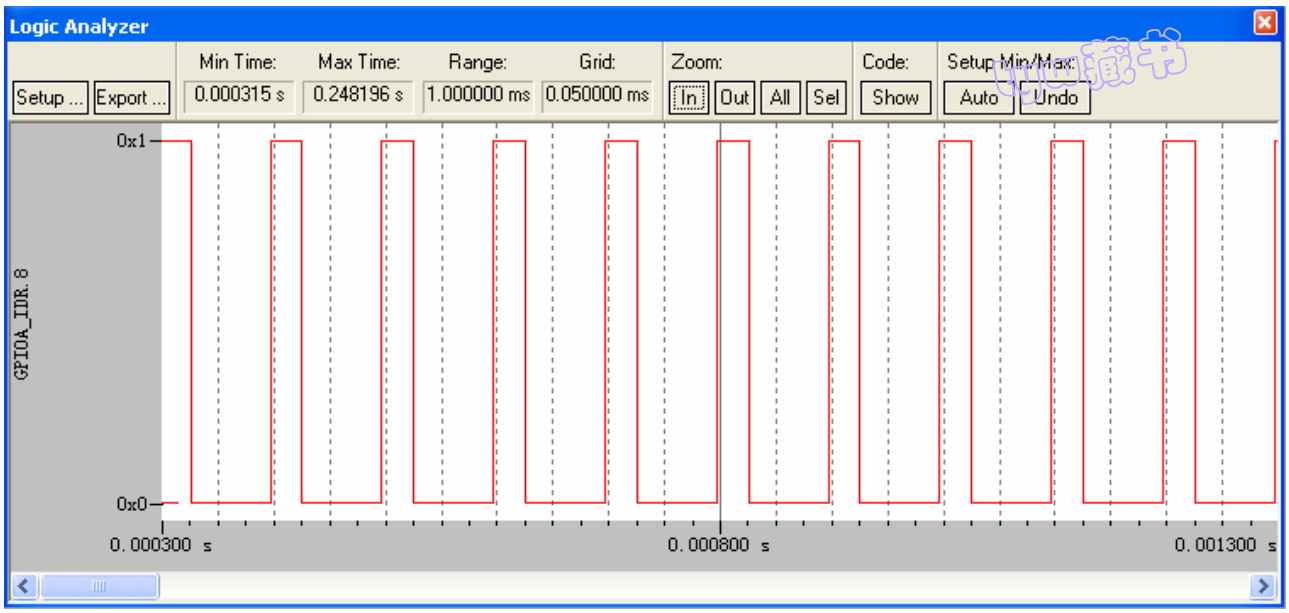
选择软件仿真







全速运行看到下面仿真波形



BHS-STM32 实验 25-PWM_2 可变占空比(直接操作寄存器)

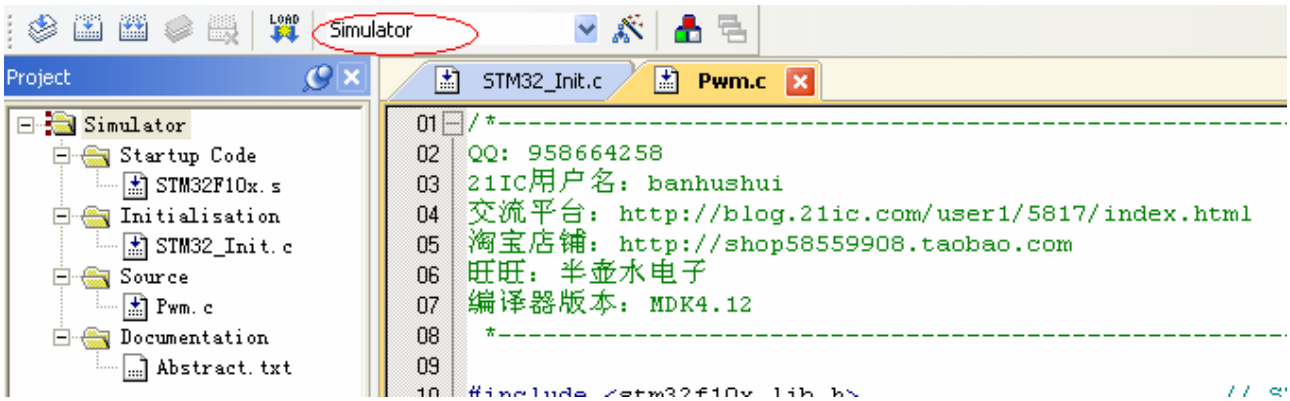
例子使用 TIM4,的通道 3，通道 4 产生 2 个可变占空比的 PWM 脉冲，其中
通道 3（PB8）通道 4（PB9）
本例最好使用示波器观察不同的占空比波形，也可以软件仿真
本例可以看到与上例不同是占空比在变化。
配置如下

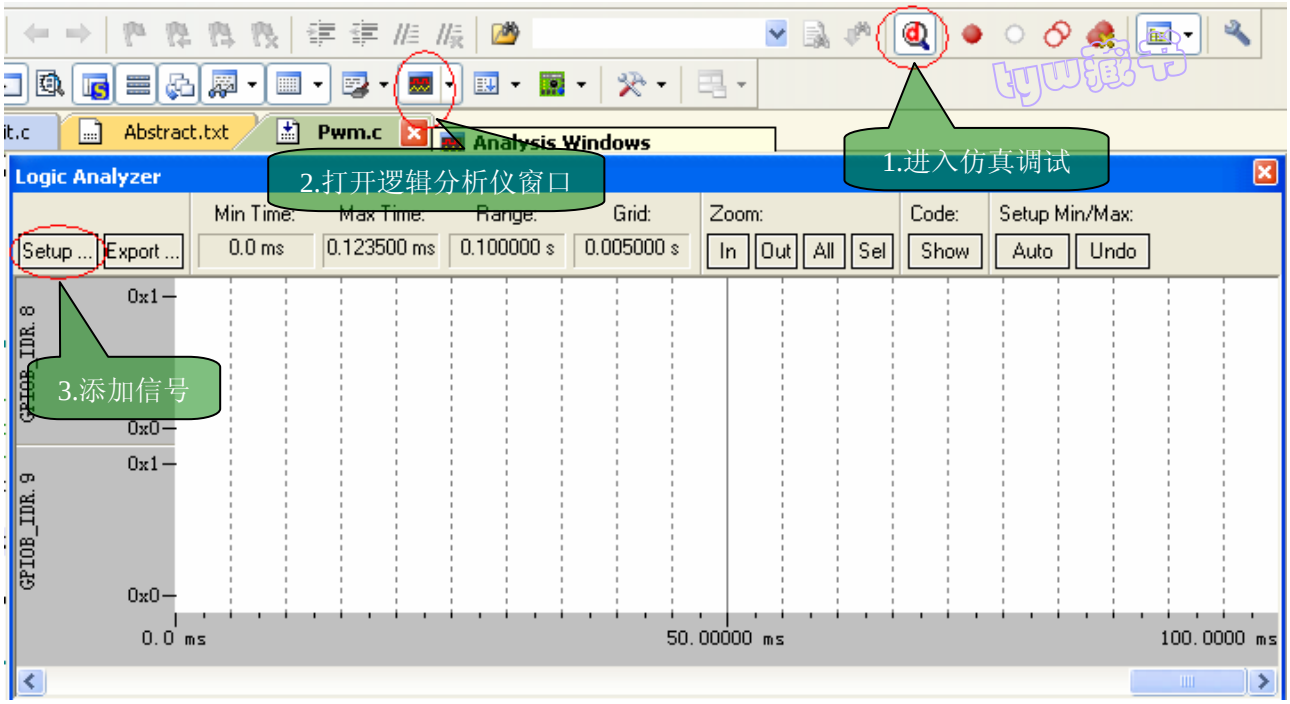
```
TIM4 : Timer 4 enabled                                                    ☑
    TIM4 period [us]                                                      1000
    TIM4 detailed settings                                                ☑
        TIM4.PSC: Timer 4 Prescaler                                       7199
        TIM4.ARR: Timer 4 Auto-reload                                     99
      ⊞ Timer 4 Control Register 1 Configuration (TIM4_CR1)
      ⊞ Timer 4 Control Register 2 Configuration (TIM4_CR2)
      ⊞ Timer 4 Slave mode control register Configuration (TIM4_SMC)
      ⊞ Channel 1 Configuration
      ⊞ Channel 2 Configuration
      ⊟ Channel 3 Configuration
        ⊞ Channel configured as output
        ⊞ Channel configured as input
          TIM4_CCR3: Capture/compare register 3                           5000
      ⊟ Channel 4 Configuration
        ⊞ Channel configured as output
        ⊞ Channel configured as input
          TIM4_CCR4: Capture/compare register 4                           2500
    TIM4 interrupts                                                       ☑
        TIM4_DIER.TDE: Trigger DMA request enabled                        ☐
        TIM4_DIER.CC4DE: Capture/Compare 4 DMA request enabled            ☐
        TIM4_DIER.CC3DE: Capture/Compare 3 DMA request enabled            ☐
        TIM4_DIER.CC2DE: Capture/Compare 2 DMA request enabled            ☐
        TIM4_DIER.CC1DE: Capture/Compare 1 DMA request enabled            ☐
        TIM4_DIER.UDE: Update DMA request enabled                         ☐
        TIM4_DIER.TIE: Trigger interrupt enabled                          ☐
        TIM4_DIER.CC4IE: Capture/Compare 4 interrupt enabled              ☐
        TIM4_DIER.CC3IE: Capture/Compare 3 interrupt enabled              ☐
        TIM4_DIER.CC2IE: Capture/Compare 2 interrupt enabled              ☐
        TIM4_DIER.CC1IE: Capture/Compare 1 interrupt enabled              ☐
        TIM4_DIER.UIE: Update interrupt enabled                           ☑
```

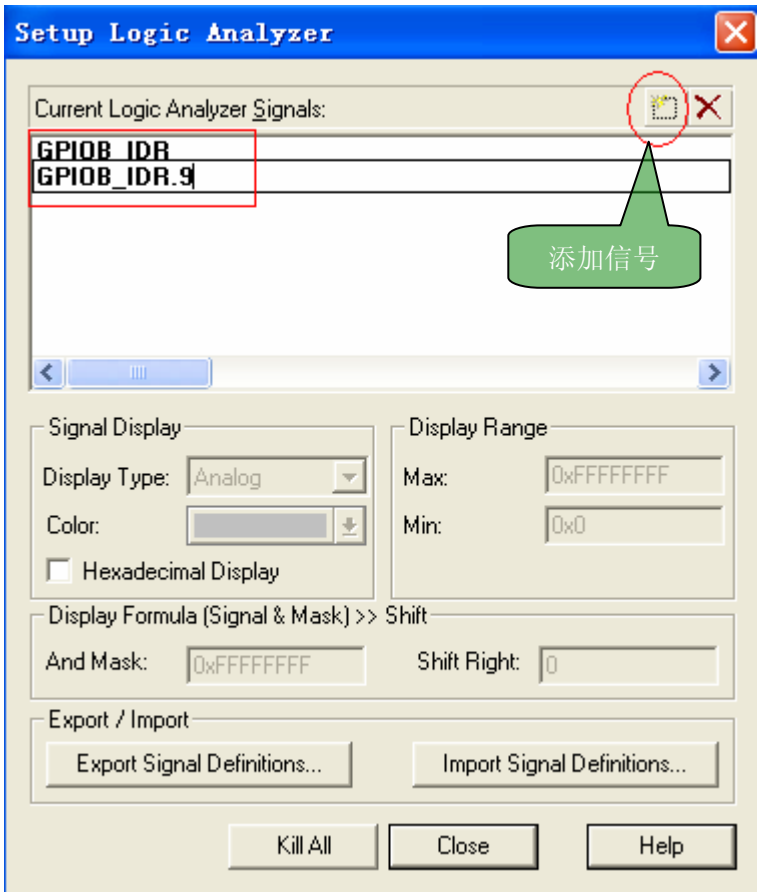
软件仿真：

PWM 实验也是可以使用软件仿真在逻辑分析仪窗口中查看波形，选择软件仿真

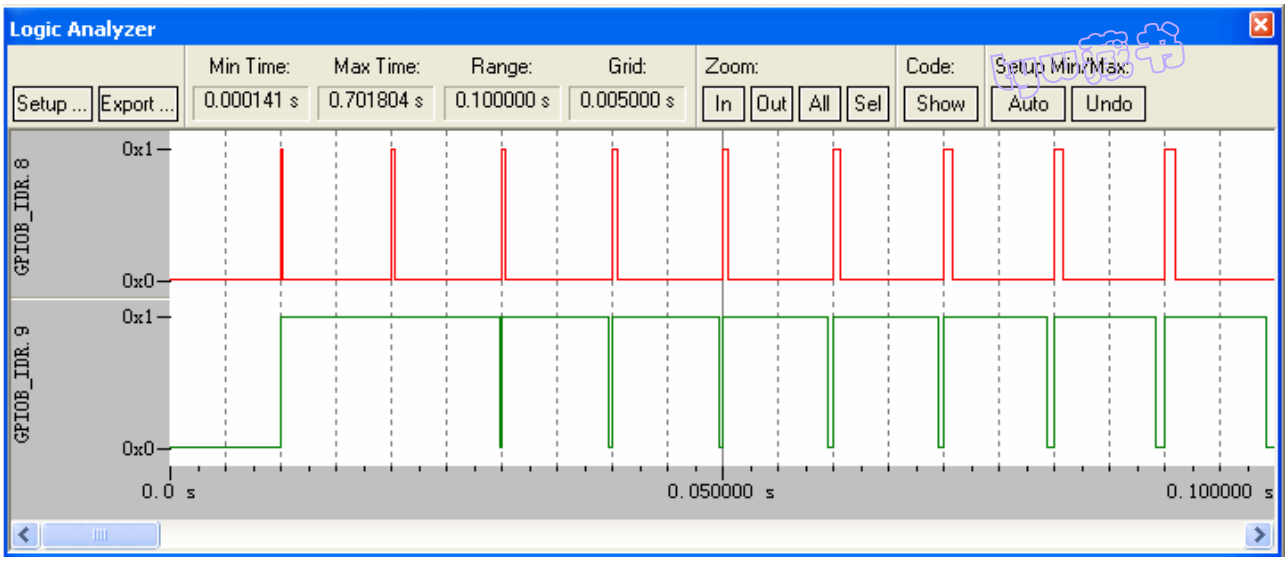
选择软件仿真







全速运行看到下面仿真波形



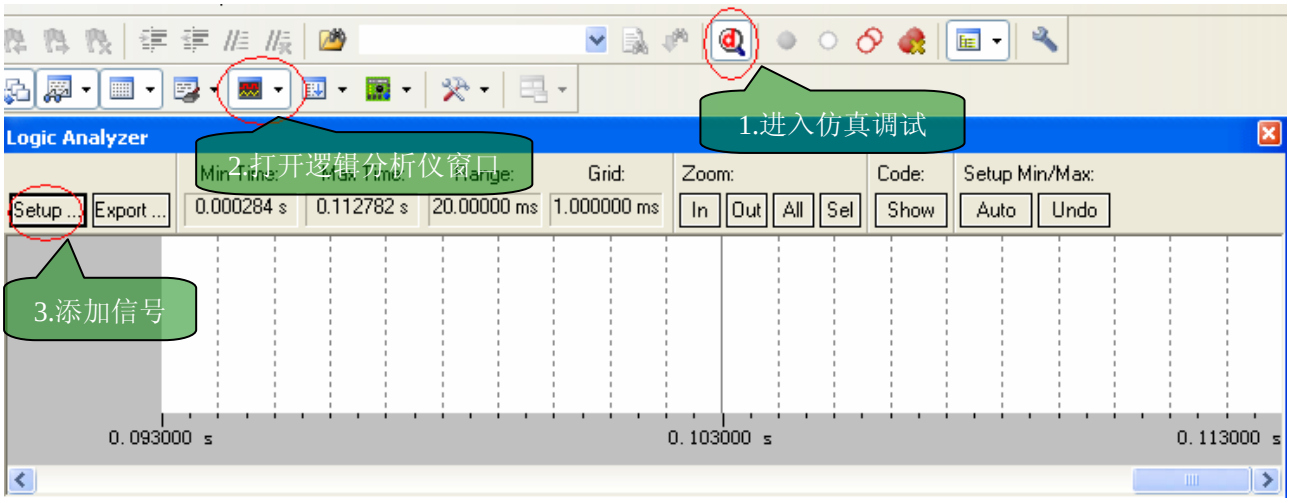
BHS-STM32 实验 26-PWM_2 可变占空比(库函数)

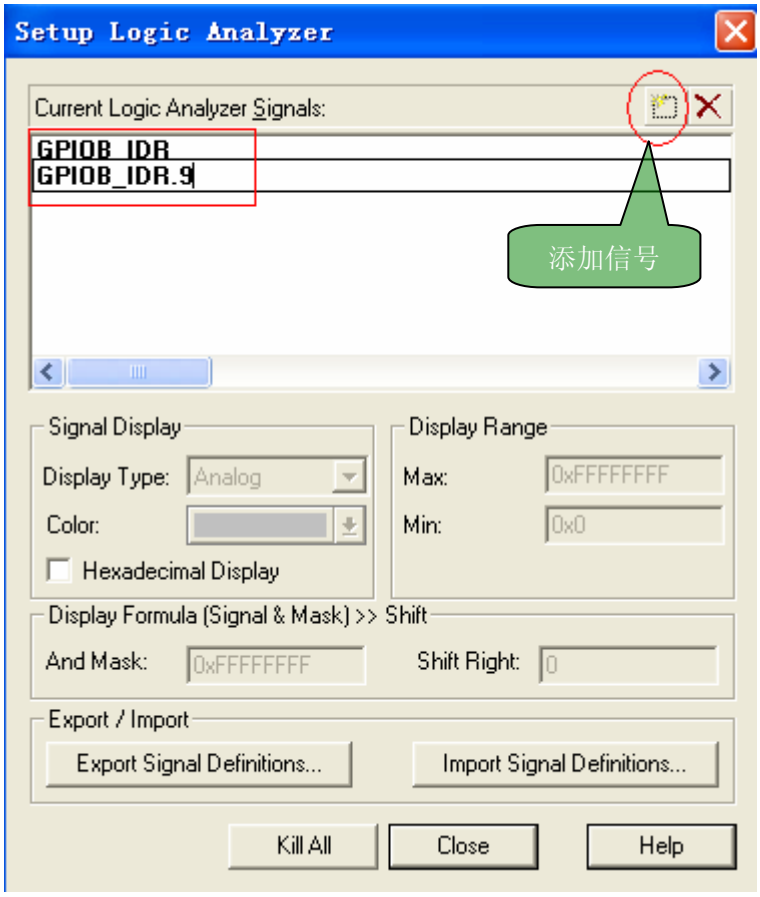
例子使用 TIM4,的通道 3，通道 4 产生 2 个可变占空比的 PWM 脉冲，其中
通道 3（PB8）通道 4（PB9）
本例最好使用示波器观察不同的占空比波形，也可以软件仿真

```
//PWM 初始化
void PWM_Configuration(void)
{
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef NVIC_InitStructure;


  /* GPIOB clock enable */
  //GPIOB 使用的 RCC 时钟使能
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|RCC_APB2Periph_AFIO, ENABLE);

  //配置使用的 GPIO
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;//GPIO_Mode_Out_PP;//;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

  GPIO_Init(GPIOB, &GPIO_InitStructure);

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

  /* Time base configuration */
    TIM_TimeBaseStructure.TIM_Period = 100-1;//1000; //设置在下一个更新事件装入活动的自动重装载寄存器周期的值　　80K
    TIM_TimeBaseStructure.TIM_Prescaler =7200-1; //设置用来作为 TIMx 时钟频率除数的预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDTS = Tck_tim
```

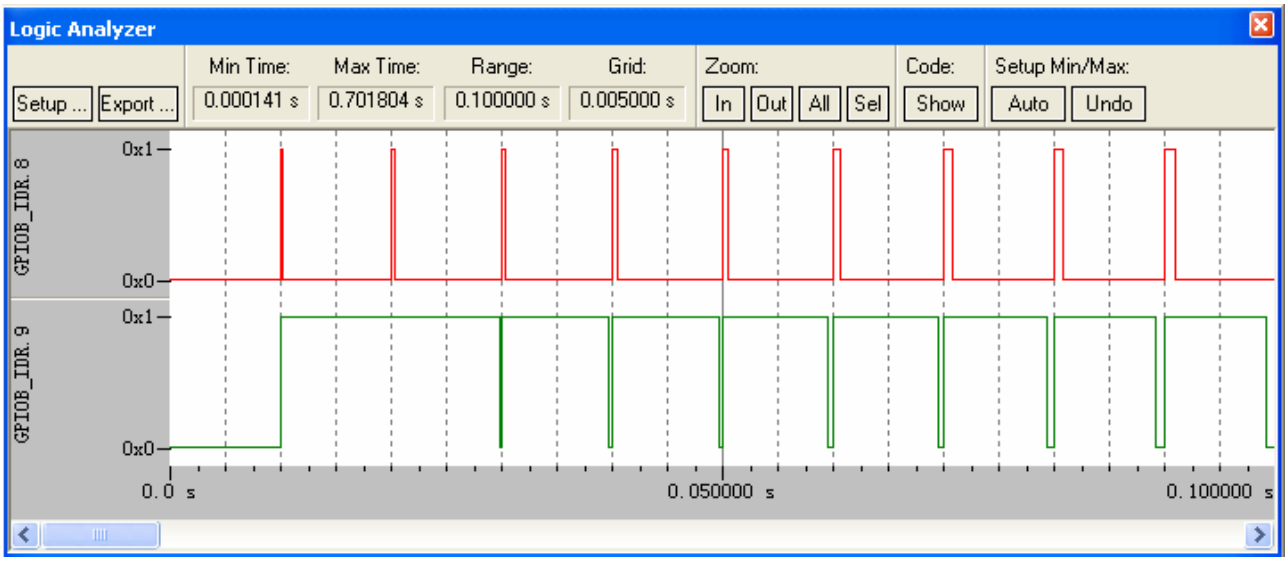
```
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  //TIM 向上计数模式
    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure); //根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时间基数单位
    /* Output Compare Active Mode configuration: Channel1 */
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2; //选择定时器模式:TIM 脉冲宽度调制模式 2
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
    TIM_OCInitStructure.TIM_Pulse = 0; //设置待装入捕获比较寄存器的脉冲值
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; //输出极性:TIM 输出比较极性高
    TIM_OC3Init(TIM4, &TIM_OCInitStructure);  //根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
    TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable);  //使能 TIMx 在 CCR3 上的预装载寄存器

    TIM_OC4Init(TIM4, &TIM_OCInitStructure);  //根据 TIM_OCInitStruct 中指定的参数初始化外设 TIMx
    TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);  //使能 TIMx 在 CCR4 上的预装载寄存器

    TIM_ARRPreloadConfig(TIM4, ENABLE); //使能 TIMx 在 ARR 上的预装载寄存器

    /* TIM4 enable counter */
    TIM_Cmd(TIM4, ENABLE);  //使能 TIMx 外设

    //TIM4->DIER = 0x0001;                                  // enable interrupt
  //NVIC->ISER[0] |= (1 << (TIM4_IRQChannel & 0x1F));       // enable interrupt

    //使能更新中断
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
    //配置中断
    NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

软件仿真





tyw藏书



ADC模数转换实验

ADC模数转换功能描述

12 位 ADC 是一种逐次逼近型模拟数字转换器。它有多达 18 个通道，可测量 16 个外部和 2 个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。

模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阀值。

ADC 的输入时钟不得超过 14MHz，它是由 PCLK2 经分频产生。

ADC 主要特征

tyw藏书

- 12 位分辨率
- 转换结束、注入转换结束和发生模拟看门狗事件时产生中断
- 单次和连续转换模式
- 从通道 0 到通道 n 的自动扫描模式
- 自校准
- 带内嵌数据一致性的数据对齐
- 采样间隔可以按通道分别编程
- 规则转换和注入转换均有外部触发选项
- 间断模式
- 双重模式(带 2 个或以上 ADC 的器件)
- ADC 转换时间：

— STM32F103xx 增强型产品：时钟为 56MHz 时为 1μs(时钟为 72MHz 为 1.17μs)
— STM32F101xx 基本型产品：时钟为 28MHz 时为 1μs(时钟为 36MHz 为 1.55μs)
— STM32F102xxUSB 型产品：时钟为 48MHz 时为 1.2μs
— STM32F105xx 和 STM32F107xx 产品：时钟为 56MHz 时为 1μs(时钟为 72MHz 为 1.17μs)

- ADC 供电要求：2.4V 到 3.6V
- ADC 输入范围：VREF- ≤ VIN ≤ VREF+
- 规则通道转换期间有 DMA 请求产生。

通道选择

有 16 个多路通道。可以把转换组织成两组：规则组和注入组。在任意多个通道上以任意顺序进行的一系列转换构成成组转换。例如，可以如下顺序完成转换：通道 3、通道 8、通道 2、通道 2、通道 0、通道 2、通道 2、通道 15。

- 规则组由多达 16 个转换组成。规则通道和它们的转换顺序在 ADC_SQRx 寄存器中选择。规则组中转换的总数应写入 ADC_SQR1 寄存器的 L[3:0]位中。
- 注入组由多达 4 个转换组成。注入通道和它们的转换顺序在 ADC_JSQR 寄存器中选择。注入组里的转换总数目应写入 ADC_JSQR 寄存器的 L[1:0]位中。

如果 ADC_SQRx 或 ADC_JSQR 寄存器在转换期间被更改，当前的转换被清除，一个新的启动脉冲将发送到 ADC 以转换新选择的组。

温度传感器/ VREFINT 内部通道

温度传感器和通道 ADC1_IN16 相连接，内部参照电压 VREFINT 和 ADC1_IN17 相连接。可以按注入或规则通道对这两个内部通道进行转换。

单次转换模式

单次转换模式下，ADC 只执行一次转换。该模式既可通过设置 ADC_CR2 寄存器的 ADON 位(只适用于规则通道)启动也可通过外部触发启动(适用于规则通道或注入通道)，这时 CONT 位为 0。

一旦选择通道的转换完成：

- 如果一个规则通道被转换：

— 转换数据被储存在 16 位 ADC_DR 寄存器中
— EOC(转换结束)标志被设置
— 如果设置了 EOCIE，则产生中断。

- 如果一个注入通道被转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中
— JEOC(注入转换结束)标志被设置
— 如果设置了 JEOCIE 位，则产生中断。

然后 ADC 停止。

连续转换模式

在连续转换模式中，当前面 ADC 转换一结束马上就启动另一次转换。此模式可通过外部触发启动或通过设置 ADC_CR2 寄存器上的 ADON 位启动，此时 CONT 位是 1。

每个转换后：

● 如果一个规则通道被转换：

— 转换数据被储存在 16 位的 ADC_DR 寄存器中

— EOC(转换结束)标志被设置

— 如果设置了 EOCIE，则产生中断。

● 如果一个注入通道被转换：

— 转换数据被储存在 16 位的 ADC_DRJ1 寄存器中

— JEOC(注入转换结束)标志被设置

— 如果设置了 JEOCIE 位，则产生中断。

扫描模式

此模式用来扫描一组模拟通道。

扫描模式可通过设置 ADC_CR1 寄存器的 SCAN 位来选择。一旦这个位被设置，ADC 扫描所有被 ADC_SQRX 寄存器(对规则通道)或 ADC_JSQR(对注入通道)选中的所有通道。在每个组的每个通道上执行单次转换。在每个转换结束时，同一组的下一个通道被自动转换。如果设置了 CONT 位，转换不会在选择组的最后一个通道上停止，而是再次从选择组的第一个通道继续转换。

如果设置了 DMA 位，在每次 EOC 后，DMA 控制器把规则组通道的转换数据传输到 SRAM 中。而注入通道转换的数据总是存储在 ADC_JDRx 寄存器中。

注入通道管理

触发注入

清除 ADC_CR1 寄存器的 JAUTO 位，并且设置 SCAN 位，即可使用触发注入功能。

1. 利用外部触发或通过设置 ADC_CR2 寄存器的 ADON 位，启动一组规则通道的转换。
2. 如果在规则通道转换期间产生一外部注入触发，当前转换被复位，注入通道序列被以单次扫描方式进行转换。
3. 然后，恢复上次被中断的规则组通道转换。如果在注入转换期间产生一规则事件，注入转换不会被中断，但是规则序列将在注入序列结束后被执行。

注：　当使用触发的注入转换时，必须保证触发事件的间隔长于注入序列。例如：序列长度为 28 个 ADC 时钟周期(即 2 个具有 1.5 个时钟间隔采样时间的转换)，触发之间最小的间隔必须是 29 个 ADC 时钟周期。

自动注入

如果设置了 JAUTO 位，在规则组通道之后，注入组通道被自动转换。这可以用来转换在 ADC_SQRx 和 ADC_JSQR 寄存器中设置的多至 20 个转换序列。

在此模式里，必须禁止注入通道的外部触发。

如果除 JAUTO 位外还设置了 CONT 位，规则通道至注入通道的转换序列被连续执行。

对于 ADC 时钟预分频系数为 4 至 8 时，当从规则转换切换到注入序列或从注入转换切换到规则序列时，会自动插入 1 个 ADC 时钟间隔；当 ADC 时钟预分频系数为 2 时，则有 2 个 ADC 时钟间隔的延迟。

注意：　不可能同时使用自动注入和间断模式。

间断模式

规则组

此模式通过设置 ADC_CR1 寄存器上的 DISCEN 位激活。它可以用来执行一个短序列的 n 次转换(n<=8)，此转换是 ADC_SQRx 寄存器所选择的转换序列的一部分。数值 n 由 ADC_CR1 寄存器的 DISCNUM[2:0]位给出。

一个外部触发信号可以启动 ADC_SQRx 寄存器中描述的下一轮 n 次转换，直到此序列所有的转换完成为止。总的序列长度由 ADC_SQR1 寄存器的 L[3:0]定义。

举例：

n=3，被转换的通道 = 0、1、2、3、6、7、9、10

第一次触发：转换的序列为 0、1、2

第二次触发：转换的序列为 3、6、7

第三次触发：转换的序列为 9、10，并产生 EOC 事件

第四次触发：转换的序列 0、1、2

注意：　当以间断模式转换一个规则组时，转换序列结束后不自动从头开始。

当所有子组被转换完成，下一次触发启动第一个子组的转换。在上面的例子中，第四次触发重新转换第一子组的通道 0、1 和 2。

注入组

此模式通过设置 ADC_CR1 寄存器的 JDISCEN 位激活。在一个外部触发事件后，该模式按通道顺序逐个转换 ADC_JSQR 寄存器中选择的序列。

一个外部触发信号可以启动 ADC_JSQR 寄存器选择的下一个通道序列的转换，直到序列中所有的转换完成为止。总的序列长度由 ADC_JSQR 寄存器的 JL[1:0]位定义。

例子：

n=1，被转换的通道 = 1、2、3

第一次触发：通道 1 被转换

第二次触发：通道 2 被转换

第三次触发：通道 3 被转换，并且产生 EOC 和 JEOC 事件

第四次触发：通道 1 被转换

注意：

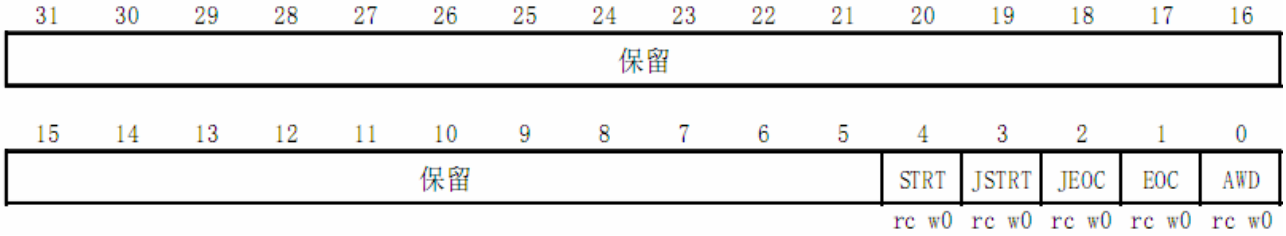
1　当完成所有注入通道转换，下个触发启动第 1 个注入通道的转换。在上述例子中，第四个触发重新转换第 1 个注入通道 1。

2　不能同时使用自动注入和间断模式。

3　必须避免同时为规则和注入组设置间断模式。间断模式只能作用于一组转换。

ADC 寄存器

ADC 状态寄存器(ADC_SR)



位31:15	保留。必须保持为0。
位4	**STRT**：规则通道开始位 (Regular channel Start flag) 该位由硬件在规则通道转换开始时设置，由软件清除。 0：规则通道转换未开始； 1：规则通道转换已开始。
位3	**JSTRT**：注入通道开始位 (Injected channel Start flag) 该位由硬件在注入通道组转换开始时设置，由软件清除。 0：注入通道组转换未开始； 1：注入通道组转换已开始。
位2	**JEOC**：注入通道转换结束位 (Injected channel end of conversion) 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0：转换未完成； 1：转换完成。
位1	**EOC**：转换结束位 (End of conversion) 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0：转换未完成； 1：转换完成。
位0	**AWD**：模拟看门狗标志位 (Analog watchdog flag) 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0：没有发生模拟看门狗事件； 1：发生模拟看门狗事件。

ADC 控制寄存器 1(ADC_CR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								AWDEN	JAWDEN	保留		DUALMOD[3:0]			
								rw	rw			rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:24	保留。必须保持为 0。
位 23	AWDEN：在规则通道上开启模拟看门狗 (Analog watchdog enable on regular channels) 该位由软件设置和清除。 0：在规则通道上禁用模拟看门狗； 1：在规则通道上使用模拟看门狗。
位 22	JAWDEN：在注入通道上开启模拟看门狗 (Analog watchdog enable on injected channels) 该位由软件设置和清除。 0：在注入通道上禁用模拟看门狗； 1：在注入通道上使用模拟看门狗。
位 21:20	保留。必须保持为 0。
位 19:16	DUALMOD[3:0]：双模式选择 (Dual mode selection) 软件使用这些位选择操作模式。 0000：独立模式 0001：混合的同步规则+注入同步模式

	0010：混合的同步规则+交替触发模式 0011：混合同步注入+快速交叉模式 0100：混合同步注入+慢速交叉模式 0101：注入同步模式 0110：规则同步模式 0111：快速交叉模式 1000：慢速交叉模式 1001：交替触发模式 注： 在 ADC2 和 ADC3 中这些位为保留位 在双模式中，改变通道的配置会产生一个重新开始的条件，这将导致同步丢失。建议在进行任 何配置改变前关闭双模式。
位 15:13	DISCNUM[2:0]：间断模式通道计数 (Discontinuous mode channel count) 软件通过这些位定义在间断模式下，收到外部触发后转换规则通道的数目 000：1 个通道 001：2 个通道 …… 111：8 个通道
位 12	JDISCEN：在注入通道上的间断模式 (Discontinuous mode on injected channels) 该位由软件设置和清除，用于开启或关闭注入通道组上的间断模式 0：注入通道组上禁用间断模式； 1：注入通道组上使用间断模式。
位 11	DISCEN：在规则通道上的间断模式 (Discontinuous mode on regular channels) 该位由软件设置和清除，用于开启或关闭规则通道组上的间断模式 0：规则通道组上禁用间断模式； 1：规则通道组上使用间断模式。
位 10	AUTO：自动的注入通道组转换 (Automatic Injected Group conversion) 该位由软件设置和清除，用于开启或关闭规则通道组转换结束后自动的注入通道组转换 0：关闭自动的注入通道组转换； 1：开启自动的注入通道组转换。
位 9	AWDSGL：扫描模式中在一个单一的通道上使用看门狗 (Enable the watchdog on a single channel in scan mode) 该位由软件设置和清除，用于开启或关闭由 AWDCH[4:0]位指定的通道上的模拟看门狗功能 0：在所有的通道上使用模拟看门狗； 1：在单一通道上使用模拟看门狗。
位 8	SCAN：扫描模式 (Scan mode) 该位由软件设置和清除，用于开启或关闭扫描模式。在扫描模式中，转换由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道。 0：关闭扫描模式； 1：使用扫描模式。 注：如果分别设置了 EOCIE 或 JEOCIE 位，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。
位 7	JEOCIE：允许产生注入通道转换结束中断 (Interrupt enable for injected channels) 该位由软件设置和清除，用于禁止或允许所有注入通道转换结束后产生中断。 0：禁止 JEOC 中断；

	1：允许 JEOC 中断。当硬件设置 JEOC 位时产生中断。
位 6	AWDIE：允许产生模拟看门狗中断 (Analog watchdog interrupt enable) 该位由软件设置和清除，用于禁止或允许模拟看门狗产生中断。在扫描模式下，如果看门狗检 测到超范围的数值时，只有在设置了该位时扫描才会中止。 0：禁止模拟看门狗中断； 1：允许模拟看门狗中断。
位 5	EOCIE：允许产生 EOC 中断 (Interrupt enable for EOC) 该位由软件设置和清除，用于禁止或允许转换结束后产生中断。 0：禁止 EOC 中断； 1：允许 EOC 中断。当硬件设置 EOC 位时产生中断。
位 4:0	AWDCH[4:0]：模拟看门狗通道选择位 (Analog watchdog channel select bits) 这些位由软件设置和清除，用于选择模拟看门狗保护的输入通道。 00000：ADC 模拟输入通道 0 00001：ADC 模拟输入通道 1 …… 01111：ADC 模拟输入通道 15 10000：ADC 模拟输入通道 16 10001：ADC 模拟输入通道 17 保留所有其他数值。 注：ADC1 的模拟输入通道 16 和通道 17 在芯片内部分别连到了温度传感器和 VREFINT。 ADC2 的模拟输入通道 16 和通道 17 在芯片内部连到了 VSS。 ADC3 模拟输入通道 9、14、15、16、17 与 Vss 相连。

ADC 控制寄存器 2(ADC_CR2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								TS VREFE	SW START	JSW START	EXT TRIG	EXTSEL[2:0]			保留
								rw	rw	rw	rw	rw	rw	rw	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JEXT TRIG	JEXTSEL[2:0]			ALIGN	保留		DMA	保留				RST CAL	CAL	CONT	ADON
rw	rw	rw	rw	rw			rw					rw	rw	rw	rw

位 31:24	保留。必须保持为 0。
位 23	TSVREFE：温度传感器和 VREFINT 使能 (Temperature sensor and VREFINT enable) 该位由软件设置和清除，用于开启或禁止温度传感器和 VREFINT 通道。在多于 1 个 ADC 的器件 中，该位仅出现在 ADC1 中。 0：禁止温度传感器和 VREFINT； 1：启用温度传感器和 VREFINT。
位 22	SWSTART：开始转换规则通道 (Start conversion of regular channels) 由软件设置该位以启动转换，转换开始后硬件马上清除此位。如果在 EXTSEL[2:0]位中选择了 SWSTART 为触发事件，该位用于启动一组规则通道的转换， 0：复位状态； 1：开始转换规则通道。
位 21	JSWSTART：开始转换注入通道 (Start conversion of injected channels)

	由软件设置该位以启动转换，软件可清除此位或在转换开始后硬件马上清除此位。如果在 JEXTSEL[2:0]位中选择了 JSWSTART 为触发事件，该位用于启动一组注入通道的转换， 0：复位状态； 1：开始转换注入通道。
位 20	EXTTRIG：规则通道的外部触发转换模式 (External trigger conversion mode for regular channels) 该位由软件设置和清除，用于开启或禁止可以启动规则通道组转换的外部触发事件。 0：不用外部事件启动转换； 1：使用外部事件启动转换。
位 19:17	EXTSEL[2:0]：选择启动规则通道组转换的外部事件 (External event select for regular group) 这些位选择用于启动规则通道组转换的外部事件 ADC1 和 ADC2 的触发配置如下 000：定时器 1 的 CC1 事件　100：定时器 3 的 TRGO 事件 001：定时器 1 的 CC2 事件　101：定时器 4 的 CC4 事件 110：EXTI 线 11/ TIM8_TRGO 事件，仅大容量产 品具有 TIM8_TRGO 功能 010：定时器 1 的 CC3 事件 011：定时器 2 的 CC2 事件　111：SWSTART ADC3 的触发配置如下 000：定时器 3 的 CC1 事件　100：定时器 8 的 TRGO 事件 001：定时器 2 的 CC3 事件　101：定时器 5 的 CC1 事件 010：定时器 1 的 CC3 事件　110：定时器 5 的 CC3 事件 011：定时器 8 的 CC1 事件　111：SWSTART
位 16	保留。必须保持为 0。
位 15	JEXTTRIG：注入通道的外部触发转换模式 (External trigger conversion mode for injected channels) 该位由软件设置和清除，用于开启或禁止可以启动注入通道组转换的外部触发事件。 0：不用外部事件启动转换； 1：使用外部事件启动转换。
位 14:12	JEXTSEL[2:0]：选择启动注入通道组转换的外部事件 (External event select for injected group) 这些位选择用于启动注入通道组转换的外部事件。 ADC1 和 ADC2 的触发配置如下 000：定时器 1 的 TRGO 事件　100：定时器 3 的 CC4 事件 001：定时器 1 的 CC4 事件　101：定时器 4 的 TRGO 事件 110：EXTI 线 15/TIM8_CC4 事件(仅大容量产品具 有 TIM8_CC4) 010：定时器 2 的 TRGO 事件 011：定时器 2 的 CC1 事件　111：JSWSTART ADC3 的触发配置如下 000：定时器 1 的 TRGO 事件　100：定时器 8 的 CC4 事件 001：定时器 1 的 CC4 事件　101：定时器 5 的 TRGO 事件 010：定时器 4 的 CC3 事件　110：定时器 5 的 CC4 事件 011：定时器 8 的 CC2 事件　111：JSWSTART
位 11	ALIGN：数据对齐 (Data alignment)

	174/754 该位由软件设置和清除。参考图 29 和图 30。 0：右对齐； 1：左对齐
位 10:9	保留。必须保持为 0。
位 8	DMA：直接存储器访问模式 (Direct memory access mode) 该位由软件设置和清除。详见 DMA 控制器章节。 0：不使用 DMA 模式； 1：使用 DMA 模式。 注：只有 ADC1 和 ADC3 能产生 DMA 请求。
位 7:4	保留。必须保持为 0。
位 3	RSTCAL：复位校准 (Reset calibration) 该位由软件设置并由硬件清除。在校准寄存器被初始化后该位将被清除。 0：校准寄存器已初始化； 1：初始化校准寄存器。 注：如果正在进行转换时设置 RSTCAL，清除校准寄存器需要额外的周期。
位 2	CAL：A/D 校准 (A/D Calibration) 该位由软件设置以开始校准，并在校准结束时由硬件清除。 0：校准完成； 1：开始校准。
位 1	CONT：连续转换 (Continuous conversion) 该位由软件设置和清除。如果设置了此位，则转换将连续进行直到该位被清除。 0：单次转换模式； 1：连续转换模式。
位 0	ADON：开/关 A/D 转换器 (A/D converter ON / OFF) 该位由软件设置和清除。当该位为’0’时，写入’1’将把 ADC 从断电模式下唤醒。 当该位为’1’时，写入’1’将启动转换。应用程序需注意，在转换器上电至转换开始有一个延迟 tSTAB。 0：关闭 ADC 转换/校准，并进入断电模式； 1：开启 ADC 并启动转换。 注：如果在这个寄存器中与 ADON 一起还有其他位被改变，则转换不被触发。这是为了防止触发错误的转换。

ADC 采样时间寄存器 1(ADC_SMPR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:24	保留。必须保持为0。
位23:0	**SMPx[2:0]**：选择通道x的采样时间 (Channel x Sample time selection) 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000：1.5周期　　100：41.5周期 001：7.5周期　　101：55.5周期 010：13.5周期　　110：71.5周期 011：28.5周期　　111：239.5周期 注：ADC1的模拟输入通道16和通道17在芯片内部分别连到了温度传感器和V_{REFINT}。 ADC2的模拟输入通道16和通道17在芯片内部连到了Vss。 ADC3模拟输入通道14、15、16、17与Vss相连

ADC 采样时间寄存器 2(ADC_SMPR2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0		SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30	保留。必须保持为0。
位29:0	**SMPx[2:0]**：选择通道x的采样时间 (Channel x Sample time selection) 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000：1.5周期　　100：41.5周期 001：7.5周期　　101：55.5周期 010：13.5周期　　110：71.5周期 011：28.5周期　　111：239.5周期 注：ADC3模拟输入通道9与Vss相连

ADC 注入通道数据偏移寄存器 x (ADC_JOFRx)(x=1..4)

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

位31:12	保留。必须保持为0。
位11:0	**JOFFSETx[11:0]**：注入通道x的数据偏移 (Data offset for injected channel x) 当转换注入通道时，这些位定义了用于从原始转换数据中减去的数值。转换的结果可以在ADC_JDRx寄存器中读出。

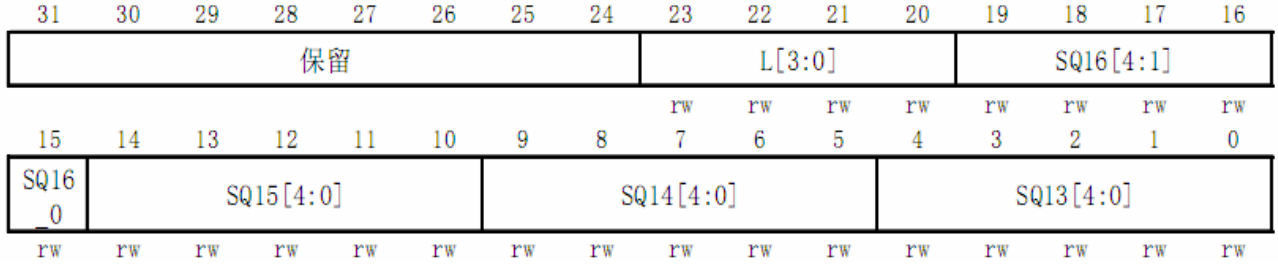
ADC 看门狗高阀值寄存器(ADC_HTR)

位31:12	保留。必须保持为0。
位11:0	**HT[11:0]**：模拟看门狗高阀值 (Analog watchdog high threshold) 这些位定义了模拟看门狗的阀值高限。

ADC 看门狗低阀值寄存器(ADC_LRT)

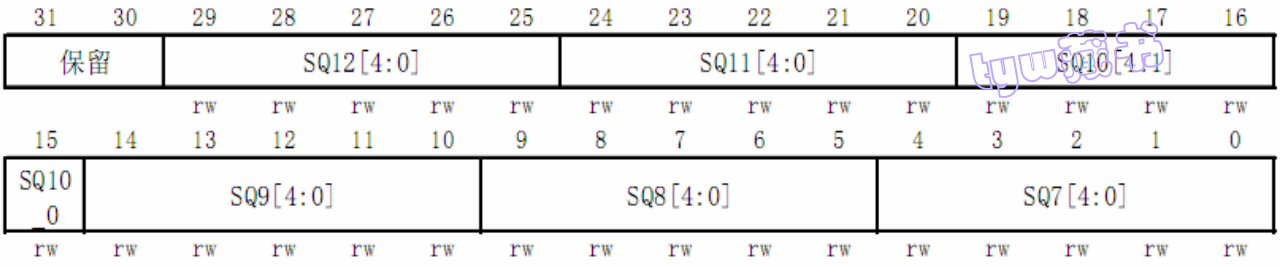
位31:12	保留。必须保持为0。
位11:0	**LT[11:0]**：模拟看门狗低阀值 (Analog watchdog low threshold) 这些位定义了模拟看门狗的阀值低限。

ADC 规则序列寄存器 1(ADC_SQR1)



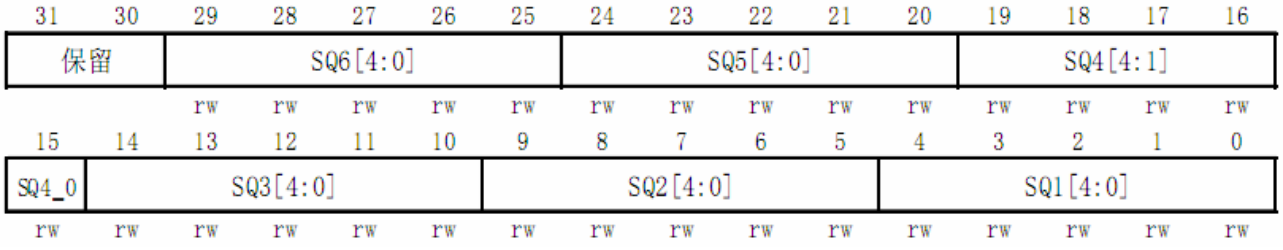
位31:24	保留。必须保持为0。
位23:20	**L[3:0]**：规则通道序列长度 (Regular channel sequence length) 这些位由软件定义在规则通道转换序列中的通道数目。 0000：1个转换 0001：2个转换 …… 1111：16个转换
位19:15	**SQ16[4:0]**：规则序列中的第16个转换 (16th conversion in regular sequence) 这些位由软件定义转换序列中的第16个转换通道的编号(0~17)。
位14:10	**SQ15[4:0]**：规则序列中的第15个转换 (15th conversion in regular sequence)
位9:5	**SQ14[4:0]**：规则序列中的第14个转换 (14th conversion in regular sequence)
位4:0	**SQ13[4:0]**：规则序列中的第13个转换 (13th conversion in regular sequence)

ADC 规则序列寄存器 2(ADC_SQR2)



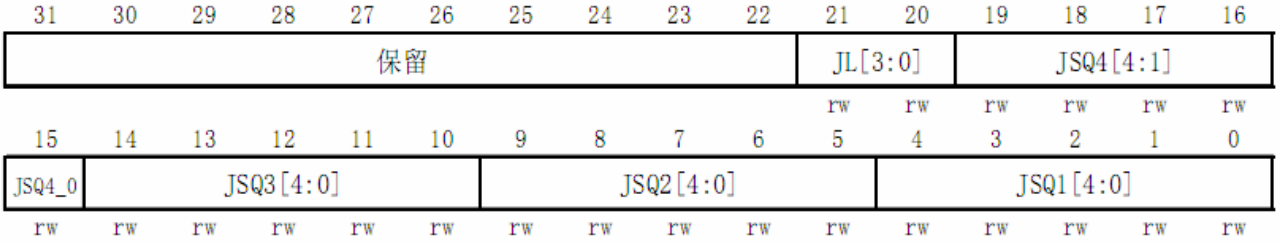
位31:30	保留。必须保持为0。
位29:25	**SQ12[4:0]**：规则序列中的第12个转换 (12th conversion in regular sequence) 这些位由软件定义转换序列中的第12个转换通道的编号(0~17)。
位24:20	**SQ11[4:0]**：规则序列中的第11个转换 (11th conversion in regular sequence)
位19:15	**SQ10[4:0]**：规则序列中的第10个转换 (10th conversion in regular sequence)
位14:10	**SQ9[4:0]**：规则序列中的第9个转换 (9th conversion in regular sequence)
位9:5	**SQ8[4:0]**：规则序列中的第8个转换 (82th conversion in regular sequence)
位4:0	**SQ7[4:0]**：规则序列中的第7个转换 (7th conversion in regular sequence)

ADC 规则序列寄存器 3(ADC_SQR3)



位31:30	保留。必须保持为0。
位29:25	**SQ6[4:0]**：规则序列中的第6个转换 (6th conversion in regular sequence) 这些位由软件定义转换序列中的第6个转换通道的编号(0~17)。
位24:20	**SQ5[4:0]**：规则序列中的第5个转换 (5th conversion in regular sequence)
位19:15	**SQ4[4:0]**：规则序列中的第4个转换 (4th conversion in regular sequence)
位14:10	**SQ3[4:0]**：规则序列中的第3个转换 (3rd conversion in regular sequence)
位9:5	**SQ2[4:0]**：规则序列中的第2个转换 (2nd conversion in regular sequence)
位4:0	**SQ1[4:0]**：规则序列中的第1个转换 (1st conversion in regular sequence)

ADC 注入序列寄存器(ADC_JSQR)



位31:22	保留。必须保持为0。
位21:20	**JL[1:0]**：注入通道序列长度 (Injected sequence length) 这些位由软件定义在规则通道转换序列中的通道数目。 00：1个转换 01：2个转换 10：3个转换 11：4个转换
位19:15	**JSQ4[4:0]**：注入序列中的第4个转换 (4th conversion in injected sequence) 这些位由软件定义转换序列中的第4个转换通道的编号(0~17)。 注：不同于规则转换序列，如果JL[1:0]的长度小于4，则转换的序列顺序是从(4-JL)开始。例如：ADC_JSQR[21:0] = 10 00011 00011 00111 00010，意味着扫描转换将按下列通道顺序转换：7、3、3，而不是2、7、3。
位14:10	**JSQ3[4:0]**：注入序列中的第3个转换 (3rd conversion in injected sequence)
位9:5	**JSQ2[4:0]**：注入序列中的第2个转换 (2nd conversion in injected sequence)
位4:0	**JSQ1[4:0]**：注入序列中的第1个转换 (1st conversion in injected sequence)

ADC 注入数据寄存器 x (ADC_JDRx) (x= 1..4)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JDATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	保留。必须保持为0。
位21:20	**JDATA[15:0]**：注入转换的数据 (Injected data) 这些位为只读，包含了注入通道的转换结果。数据是左对齐或右对齐，如图29和图30所示。

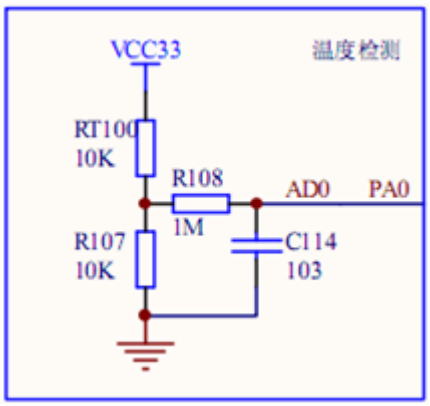
ADC 规则数据寄存器(ADC_DR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADC2DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	**ADC2DATA[15:0]**：ADC2转换的数据 (ADC2 data) - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见11.9：双ADC模式 - 在ADC2和ADC3中：不使用这些位。
位15:0	**DATA[15:0]**：规则转换的数据 (Regular data) 这些位为只读，包含了规则通道的转换结果。数据是左对齐或右对齐，如图29和图30所示。

BHS-STM32 实验 27-ADC模数转换(直接操作寄存器)

本例子通过串口发送 AD0 的值，AD0 连接一个 NTC 温度电阻，本例 ADC 使用了 DMA 方式
使用 PA0-ADC1.0，用手触摸 NTC 温度电阻,输出数字将变化。

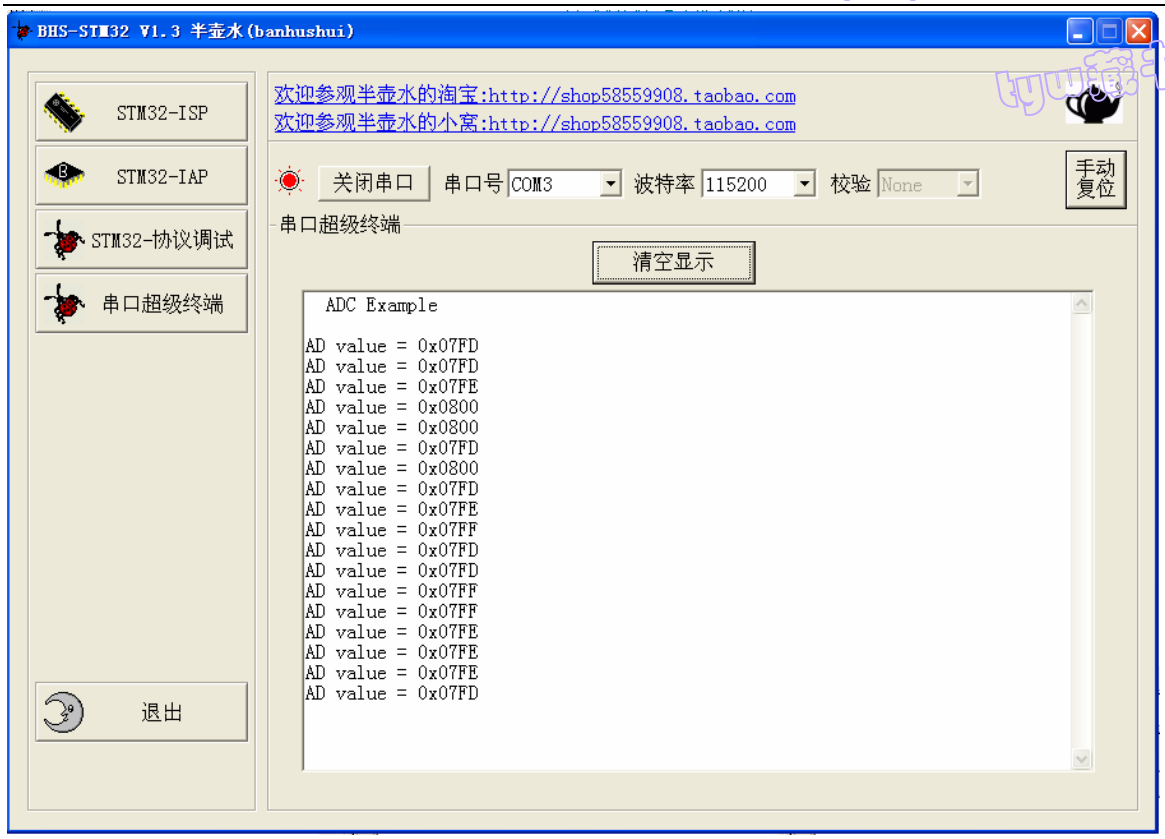


由于 BHS-STM32-V 精华版没有 NTC 温度电阻，后面我们将使用 MDK 的软件仿真看 ADC

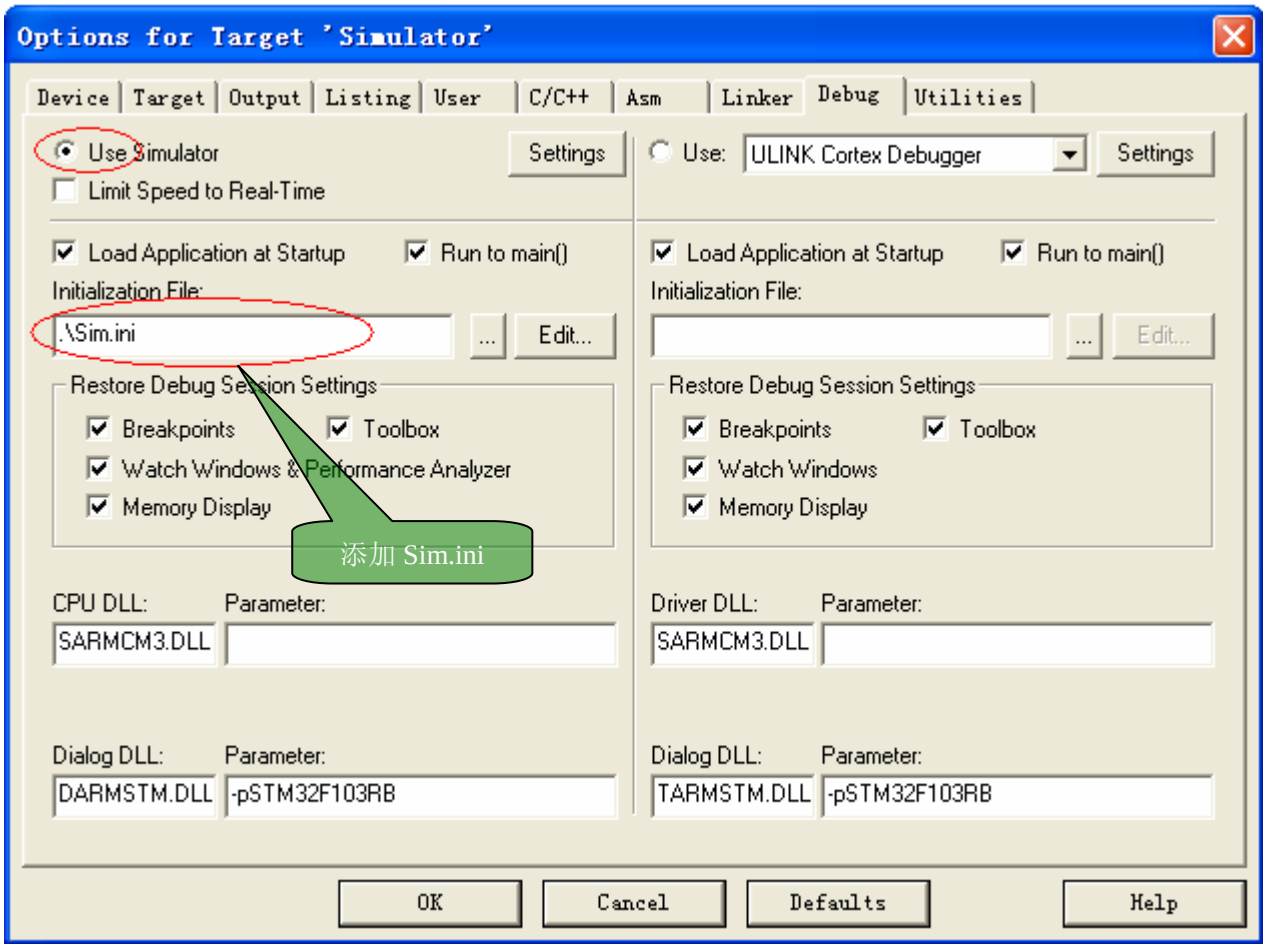
```
//ADC 初始化
void adc_Init (void) {

  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;                    //使能 GPIO 时钟
  //GPIO 设置为模拟输入
  GPIOA->CRL &= ~0x00000000;
  GPIOA->CRH &= ~0x00000000;

  // enable peripheral clock for DMA
  //使能 DMA 时钟
  RCC->AHBENR |= (1<<0);
  // set channel1 memory address
  //设置 DMA 内存地址，ADC 转换结果直接放入该地址
  DMA1_Channel1->CMAR  = (u32)&ADC_ConvertedValue;
  // set channel0 peripheral address
  //设置通道 1 外设地址
  DMA1_Channel1->CPAR  = (u32)&(ADC1->DR);
  // transmit 1 word
  //DMA 传送 1 个字
  DMA1_Channel1->CNDTR = 1;
  // configure DMA channel
  DMA1_Channel1->CCR   = 0x00002520;
  // DMA Channel 1 enable      //使能 DMA 通道
  DMA1_Channel1->CCR  |= (1 << 0);

  // enable peripheral clock for ADC1
  //使能 ADC 时钟
  RCC->APB2ENR |= (1<<9);
  // only one conversion    //  只有 1 个转换通道
  ADC1->SQR1  = 0x00000000;
  // set sample time channel0 (55,5 cycles)// (3bit)
  //通道 1 采样周期 55.5 个时钟周期
  ADC1->SMPR2 = 0x00000028;
  // set channel1 as 1st conversion// (5bit)
  //第 1 个转换通道是 1 通道
```

```
    ADC1->SQR3    = 0x00000000;
    // use independant mode, SCAN mode
    //使用独立模式，扫描模式
    ADC1->CR1     = 0x00000100;
    // use data align right,continuous conversion
    //使用数据右对齐，连续转换
    ADC1->CR2     = 0x000E0103;
    // EXTSEL = SWSTART
    // enable ADC, DMA mode, no external Trigger
    // start SW conversion
    //允许 ADC，DMA 模式，无需外接触发器
    //开始转换
    ADC1->CR2    |=    0x00500000;
}
//主函数
int main (void)
{
int AD_value;

    // STM32 setup
    //STM32 初始化
    stm32_Init ();
    //ADC 初始化
    adc_Init();
    printf ("ADC Example\r\n\r\n");

    while (1)
    {
            //Delay(1000);
            Delay(50);
            //因为使用 DMA 操作，所以每次 ADC 转换结束，ADC 的值都被保存到 ADC_ConvertedValue 里
            AD_value    = ADC_ConvertedValue;
            //这里可以通过串口看到 ADC 的值
            printf("AD value = 0x%04X\r\n", AD_value);
    } // end while
} // end main
```

使用我提供的串口调试工具，选择【串口超级终端】波特率设置 115200



由于 BHS-STM32-V 精华版没有 NTC 温度电阻，下面我们来使用 MDK 的软件仿真看 ADC



MDK 软件仿真有很强的功能，可以在 GPIO 上模拟输入信号，比如可以输入方波，三角波，

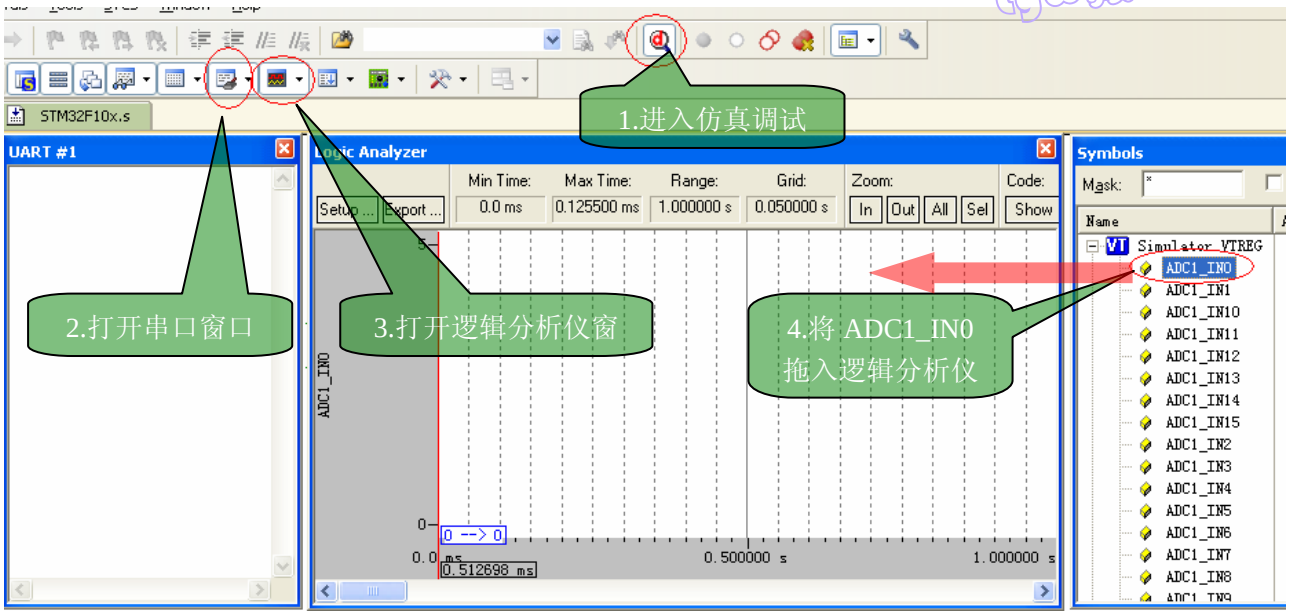
正玄波，模拟噪声信号，混合信号等。

下面我们来看 Sim.ini 是啥

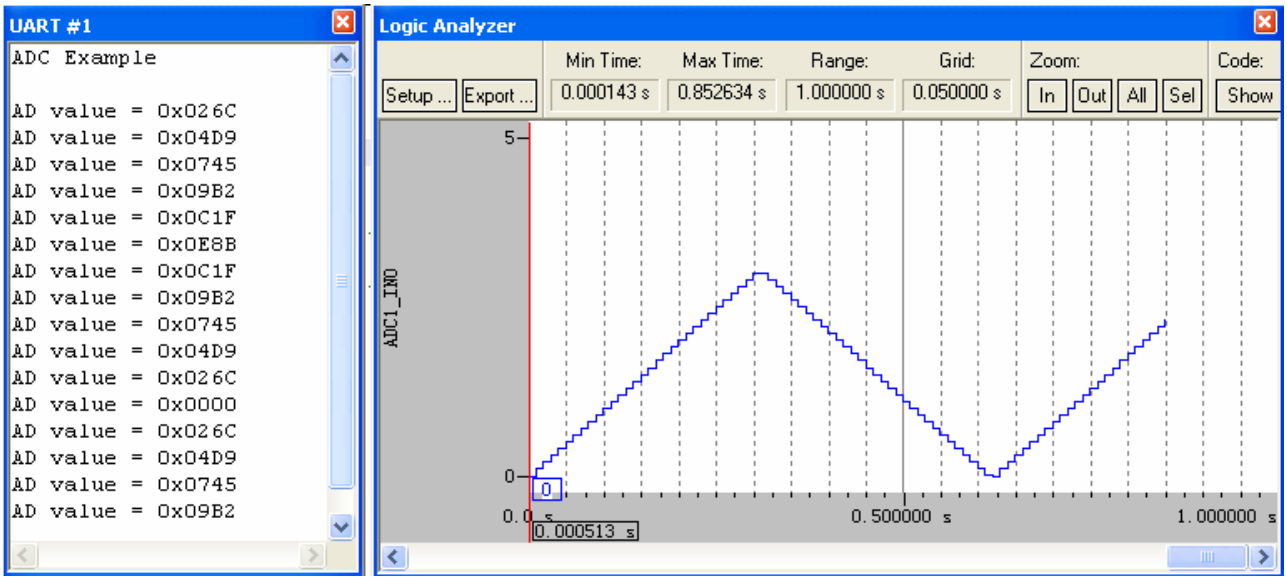
lyw藏书

```
/*----------------------------------------------------------------------------
   Analog() simulates analog input values given to channel-1 (ADC1.0)
   //下面实际是模拟一个三角波
 *----------------------------------------------------------------------------*/
Signal void Analog (float limit)   {
   float volts;

   printf ("Analog (%f) entered.\n", limit);
   while (1)
   {                                    // forever
      volts = 0;
      while (volts <= limit)
      {
         // analog input-1
         //ADC1.0 输入模拟电压
         ADC1_IN0 = volts;

         // wait 0.01 seconds
         //延时 0.01 秒
         swatch (0.01);

         // increase voltage
         //模拟电压步进增加 0.1
         volts += 0.1;
      }
      volts = limit;
      while (volts >= 0.0)
      {
         // analog input-1
         //ADC1.0 输入模拟电压
         ADC1_IN0 = volts;

         // wait 0.01 seconds
         //延时 0.01 秒
         swatch (0.01);

         // decrease voltage
         //模拟电压步进减少 0.1
         volts -= 0.1;
      }
   }
}
```

//模拟输入电压，最大值是 3
Analog(3)

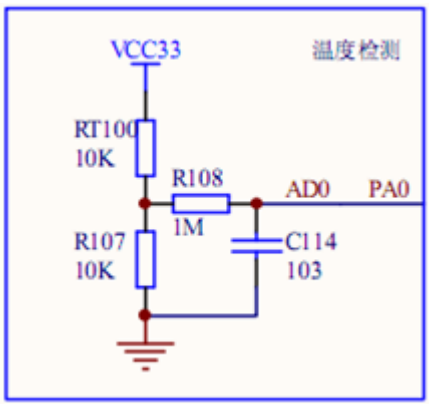


全速运行将看到下面的结果，其中[Logic Analyzer]窗口是 KEIL 模拟输入的三角波电压波形，[UART #1]窗口就是我们测定的 AD 值发送到串口，与硬件仿真是一样的效果呢
运行中我们看到电压增加 AD 值就增加，电压减少 AD 值就减少。



BHS-STM32 实验 28-ADC模数转换(库函数)

本例子通过串口发送 AD0 的值，AD0 连接一个 NTC 温度电阻，本例 ADC 使用了 DMA 方式
使用 PA0-ADC1.0，用手触摸 NTC 温度电阻,输出数字将变化。



由于 BHS-STM32-V 精华版没有 NTC 温度电阻，后面我们将使用 MDK 的软件仿真看 ADC

```
//ADC 初始化
void adc_Init (void) {
ADC_InitTypeDef ADC_InitStructure;
DMA_InitTypeDef DMA_InitStructure;
GPIO_InitTypeDef GPIO_InitStructure;


   //使能 GPIO 时钟
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
   /* Configure PC.04 (ADC Channel14) as analog input -----------------------*/
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;              //GPIO 设置为模拟输入
   GPIO_Init(GPIOA, &GPIO_InitStructure);
   // enable peripheral clock for DMA
   //使能 DMA 时钟
//   RCC->AHBENR |= (1<<0);
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
//   DMA1_Channel1->CMAR    = (u32)&ADC_ConvertedValue;// set channel1 memory address
//   DMA1_Channel1->CPAR    = (u32)&(ADC1->DR);          // set channel1 peripheral address
//   DMA1_Channel1->CNDTR = 1;                           // transmit 1 word
//   DMA1_Channel1->CCR     = 0x00002520;                // configure DMA channel
//   DMA1_Channel1->CCR    |= (1 << 0);                  // DMA Channel 1 enable

/* DMA1 channel1 configuration  配置 DMA 通道----------------------------------------------*/
   DMA_DeInit(DMA1_Channel1);
   //设置通道 1 外设地址
   DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
   //设置 DMA 内存地址，ADC 转换结果直接放入该地址
   DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
   DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
   //DMA 传送 1 个字
   DMA_InitStructure.DMA_BufferSize = 1;
   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
```

```
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);

    /* Enable DMA1 channel1 */
    DMA_Cmd(DMA1_Channel1, ENABLE);                          //使能 DMA 通道

//    RCC->APB2ENR |= (1<<9);                                 // enable peripheral clock for ADC1
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
//    ADC1->SQR1    = 0x00000000;                             // only one conversion
//    ADC1->SMPR2 = 0x00000028;                               // set sample time channel1 (55,5 cycles)
//    ADC1->SQR3    = 0x00000001;                             // set channel1 as 1st conversion

//    ADC1->CR1     = 0x00000100;                             // use independant mode, SCAN mode
//    ADC1->CR2     = 0x000E0103;                             // use data align right,continuous conversion
                                                            // EXTSEL = SWSTART
                                                            // enable ADC, DMA mode, no external Trigger
//    ADC1->CR2    |=    0x00500000;                          // start SW conversion
    /* ADC1 configuration  配置 ADC----------------------------------------------------*/
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
     //使用独立模式，扫描模式
    ADC_InitStructure.ADC_ScanConvMode = ENABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    //无需外接触发器
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    //使用数据右对齐
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    //  只有 1 个转换通道
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init(ADC1, &ADC_InitStructure);

    /* ADC1 regular channel-0 configuration */
    //通道 1 采样周期 55.5 个时钟周期
    ADC_RegularChannelConfig(ADC1, ADC_Channel_0, 1, ADC_SampleTime_55Cycles5);

    /* Enable ADC1 DMA */
    //使能 ADC 的 DMA
    ADC_DMACmd(ADC1, ENABLE);

    /* Enable ADC1 */
    //使能 ADC1
    ADC_Cmd(ADC1, ENABLE);
```

```
   /* Enable ADC1 reset calibaration register */
   ADC_ResetCalibration(ADC1);
   /* Check the end of ADC1 reset calibration register */
   while(ADC_GetResetCalibrationStatus(ADC1));

   /* Start ADC1 calibaration */
   ADC_StartCalibration(ADC1);
   /* Check the end of ADC1 calibration */
   while(ADC_GetCalibrationStatus(ADC1));

   /* Start ADC1 Software Conversion */
   //开始转换
   ADC_SoftwareStartConvCmd(ADC1, ENABLE);

}

//主函数
int main(void)
{
#ifdef DEBUG
   debug();
#endif
int AD_value;
   /* System Clocks Configuration */
   RCC_Configuration();//配置系统时钟

   GPIO_Configuration();//配置 GPIO


   /* NVIC configuration */
   NVIC_Configuration();//配置中断

   USART1_InitConfig(115200);
   adc_Init();
   printf ("ADC Example\r\n\r\n");

   while (1)
   {                                                    // Loop forever
       //Delay(1000);

           //为了方便观察软件仿真好观察采用下面的延时
           Delay(50);
           AD_value    = ADC_ConvertedValue;
           printf("AD value = 0x%04X\r\n", AD_value);
   } // end while
```

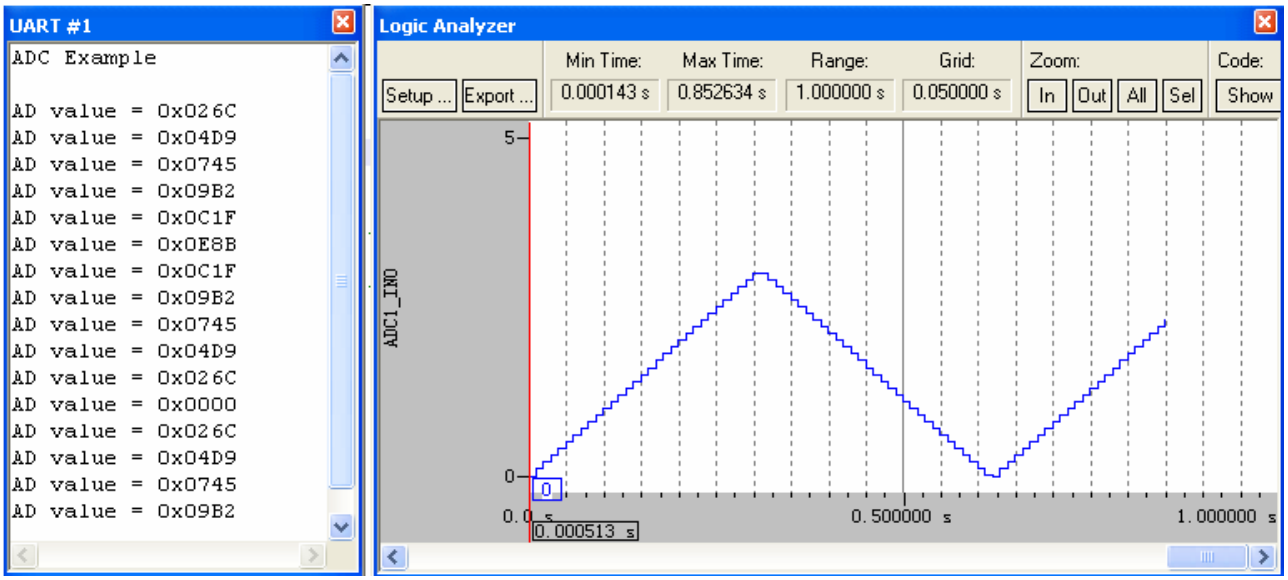
}

由于 BHS-STM32-V 精华版没有 NTC 温度电阻，下面我们来使用 MDK 的软件仿真看 ADC

添加仿真同上一例相同

全速运行将看到下面的结果，其中[Logic Analyzer]窗口是 KEIL 模拟输入的三角波电压波形，[UART #1]窗口就是我们测定的 AD 值发送到串口，与硬件仿真是一样的效果呢

运行中我们看到电压增加 AD 值就增加，电压减少 AD 值就减少。



CAN通信实验

CAN功能描述

STM32 的 CAN 控制器是 bxCAN, bxCAN 是基本扩展 CAN(Basic Extended CAN)的缩写，它支持 CAN 协议 2.0A 和 2.0B。它的设计目标是，以最小的 CPU 负荷来高效处理大量收到的报文。它也支持报文发送的优先级要求(优先级特性可软件配置)。

对于安全紧要的应用，bxCAN 提供所有支持时间触发通信模式所需的硬件功能。

bxCAN 主要特点

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高可达 1 兆位/秒
- 支持时间触发通信功能

发送

- 3 个发送邮箱
- 发送报文的优先级特性可软件配置
- 记录发送 SOF 时刻的时间戳

接收

- 3 级深度的 2 个接收 FIFO
- 可变的过滤器组：

— 在互联型产品中，CAN1 和 CAN2 分享 28 个过滤器组

— 其它 STM32F103xx 系列产品中有 14 个过滤器组

- 标识符列表
- FIFO 溢出处理方式可配置
- 记录接收 SOF 时刻的时间戳

时间触发通信模式

- 禁止自动重传模式
- 16 位自由运行定时器
- 可在最后 2 个数据字节发送时间戳

管理

- 中断可屏蔽
- 邮箱占用单独 1 块地址空间，便于提高软件效率

注：　在中容量和大容量产品中，USB 和 CAN 共用一个专用的 512 字节的 SRAM 存储器用于数据的发送和接收，因此不能同时使用 USB 和 CAN(共享的 SRAM 被 USB 和 CAN 模块互斥地访问)。USB 和 CAN 可以同时用于一个应用中但不能在同一个时间使用。

bxCAN 工作模式

bxCAN 有 3 个主要的工作模式：初始化、正常和睡眠模式。在硬件复位后，bxCAN 工作在睡眠模式以节省电能，同时 CANTX 引脚的内部上拉电阻被激活。软件通过对 CAN_MCR 寄存器的 INRQ 或 SLEEP 位置’1’，可以请求 bxCAN 进入初始化或睡眠模式。一旦进入了初始化或睡眠模式，bxCAN 就对 CAN_MSR 寄存器的 INAK 或 SLAK 位置’1’来进行确认，同时内部上拉电阻被禁用。当 INAK 和 SLAK 位都为’0’时，bxCAN 就处于正常模式。在进入正常模式前，bxCAN 必须跟 CAN 总线取得同步；为取得同步，bxCAN 要等待 CAN 总线达到空闲状态，即在 CANRX 引脚上监测到 11 个连续的隐性位。

初始化模式

在初始化完成后，软件应该让硬件进入正常模式，以便正常接收和发送报文。软件可以通过对 CAN_MCR 寄存器的 INRQ 位清’0’，来请求从初始化模式进入正常模式，然后要等待硬件对 CAN_MSR 寄存器的 INAK 位置’1’的确认。在跟 CAN 总线取得同步，即在 CANRX 引脚上监测到 11 个连续的隐性位(等效于总线空闲)后，bxCAN 才能正常接收和发送报文。 不需要在初始化模式下进行过滤器初值的设置，但必须在它处在非激活状态下完成(相应的 FACT 位为 0)。而过滤器的位宽和模式的设置，则必须在初始化模式中进入正常模式前完成。

睡眠模式(低功耗)

bxCAN 可工作在低功耗的睡眠模式。软件通过对 CAN_MCR 寄存器的 SLEEP 位置’1’，来请求进入这一模式。在该模式下，bxCAN 的时钟停止了，但软件仍然可以访问邮箱寄存器。

当 bxCAN 处于睡眠模式，软件必须对 CAN_MCR 寄存器的 INRQ 位置’1’并且同时对 SLEEP 位清’0’，才能进入初始化模式。

有 2 种方式可以唤醒(退出睡眠模式)bxCAN：通过软件对 SLEEP 位清’1’，或硬件检测到 CAN 总线的活动。

如果 CAN_MCR 寄存器的 AWUM 位为’1’，一旦检测到 CAN 总线的活动，硬件就自动对 SLEEP 位清’0’来唤醒 bxCAN。如果 CAN_MCR 寄存器的 AWUM 位为’0’，软件必须在唤醒中断里对 SLEEP 位清’0’才能退出睡眠状态。

注：　如果唤醒中断被允许(CAN_IER 寄存器的 WKUIE 位为’1’)，那么一旦检测到 CAN 总线活动就会产生唤醒中断，而不管硬件是否会自动唤醒 bxCAN。

测试模式

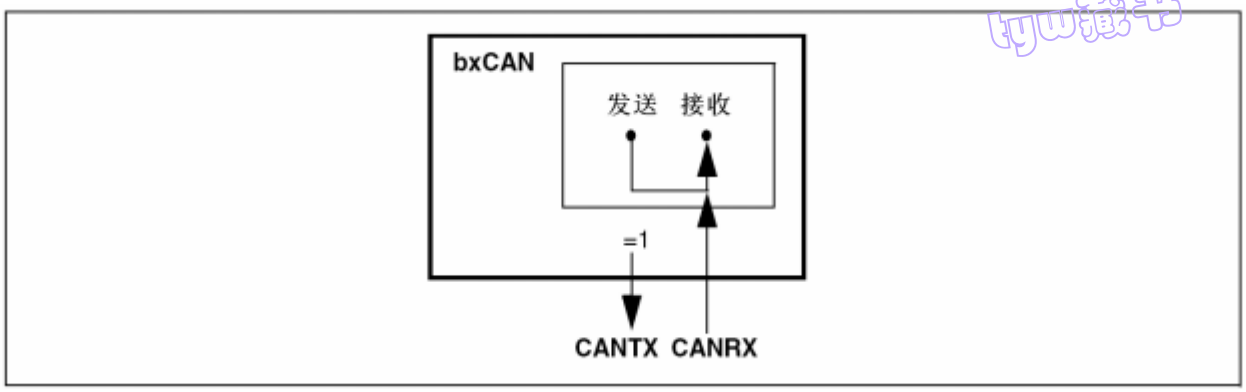
通过对 CAN_BTR 寄存器的 SILM 和/或 LBKM 位置’1’，来选择一种测试模式。只能在初始化模式下，修改这 2 位。在选择了一种测试模式后，软件需要对 CAN_MCR 寄存器的 INRQ 位清’0’，来真正进入测试模式。

静默模式

通过对 CAN_BTR 寄存器的 SILM 位置’1’，来选择静默模式。

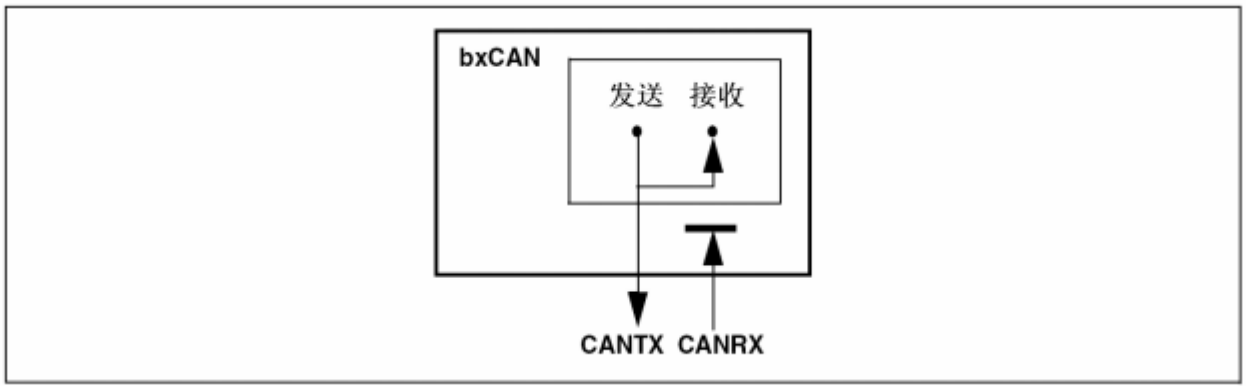
在静默模式下，bxCAN 可以正常地接收数据帧和远程帧，但只能发出隐性位，而不能真正发送报文。如果 bxCAN 需要发出显性位(确认位、过载标志、主动错误标志)，那么这样的显性位在内部被接回来从而可以被 CAN 内核检测到，同时 CAN 总线不会受到影响而仍然维持在隐性位状态。因此，静默模式通常用于

分析 CAN 总线的活动，而不会对总线造成影响－显性位(确认位、错误帧)不会真正发送到总线上。



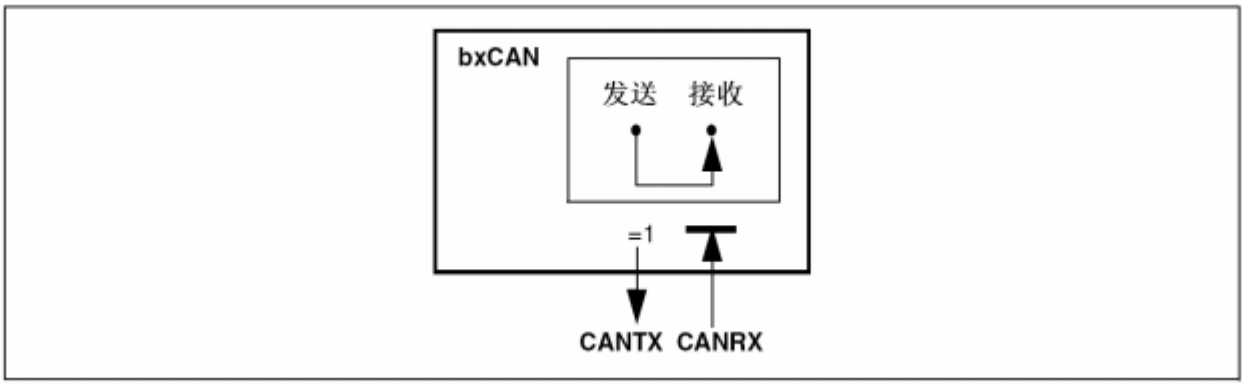
环回模式

通过对 CAN_BTR 寄存器的 LBKM 位置’1’，来选择环回模式。在环回模式下，bxCAN 把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。



环回静默模式

通过对 CAN_BTR 寄存器的 LBKM 和 SILM 位同时置’1’，可以选择环回静默模式。该模式可用于“热自测试”，即可以像环回模式那样测试 bxCAN，但却不会影响 CANTX 和 CANRX 所连接的整个 CAN 系统。在环回静默模式下，CANRX 引脚与 CAN 总线断开，同时 CANTX 引脚被驱动到隐性位状态。



CAN 寄存器描述

CAN 主控制寄存器 (CAN_MCR)

位 31:15	保留，硬件强制为 0。
位 16	BF：调试冻结 (Debug freeze) 0：在调试时，CAN 照常工作 1：在调试时，冻结 CAN 的接收/发送。仍然可以正常地读写和控制接收 FIFO。
位 15	RESET: bxCAN 软件复位 (bxCAN software master reset) 0：本外设正常工作； 1：对 bxCAN 进行强行复位，复位后 bxCAN 进入睡眠模式(FMP 位和 CAN_MCR 寄存器被初始化为其复位值)。此后硬件自动对该位清’0’。
位 14:8	保留，硬件强制为 0。
位 7	TTCM: 时间触发通信模式 (Time triggered communication mode) 0：禁止时间触发通信模式； 1：允许时间触发通信模式。 注: 要想了解关于时间触发通信模式的更多信息，请参考 22.7.2：时间触发通信模式。
位 6	ABOM: 自动离线(Bus-Off)管理 (Automatic bus-off management) 该位决定 CAN 硬件在什么条件下可以退出离线状态。 0：离线状态的退出过程是，软件对 CAN_MCR 寄存器的 INRQ 位进行置’1’随后清’0’后，一旦硬件检测到 128 次 11 位连续的隐性位，则退出离线状态； 1：一旦硬件检测到 128 次 11 位连续的隐性位，则自动退出离线状态。 注: 关于离线状态的更多信息，请参考 22.7.6：出错管理。
位 5	AWUM: 自动唤醒模式 (Automatic wakeup mode) 该位决定 CAN 处在睡眠模式时由硬件还是软件唤醒 0：睡眠模式通过清除 CAN_MCR 寄存器的 SLEEP 位，由软件唤醒； 1：睡眠模式通过检测 CAN 报文，由硬件自动唤醒。唤醒的同时，硬件自动对 CAN_MSR 寄存器的 SLEEP 和 SLAK 位清’0’ 。
位 4	NART: 禁止报文自动重传 (No automatic retransmission) 0：按照 CAN 标准，CAN 硬件在发送报文失败时会一直自动重传直到发送成功； 1：CAN 报文只被发送 1 次，不管发送的结果如何(成功、出错或仲裁丢失)。
位 3	RFLM: 接收 FIFO 锁定模式 (Receive FIFO locked mode) 0：在接收溢出时 FIFO 未被锁定，当接收 FIFO 的报文未被读出，下一个收到的报文会覆盖原有的报文； 1：在接收溢出时 FIFO 被锁定，当接收 FIFO 的报文未被读出，下一个收到的报文会被丢弃。
位 2	TXFP: 发送 FIFO 优先级 (Transmit FIFO priority) 当有多个报文同时在等待发送时，该位决定这些报文的发送顺序 0：优先级由报文的标识符来决定； 1：优先级由发送请求的顺序来决定。
位 1	SLEEP: 睡眠模式请求 (Sleep mode request) 软件对该位置’1’可以请求 CAN 进入睡眠模式，一旦当前的 CAN 活动(发送或接收报文)结束，CAN 就进入睡眠。

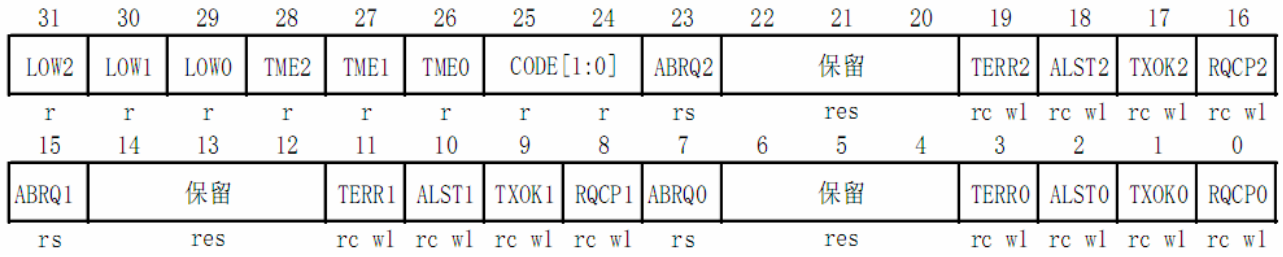
	软件对该位清’0’使 CAN 退出睡眠模式。 当设置了 AWUM 位且在 CAN Rx 信号中检测出 SOF 位时，硬件对该位清’0’。 在复位后该位被置’1’，即 CAN 在复位后处于睡眠模式。
位 0	INRQ: 初始化请求 (Initialization request) 软件对该位清’0’可使 CAN 从初始化模式进入正常工作模式：当 CAN 在接收引脚检测到连续的 11 个隐性位后，CAN 就达到同步，并为接收和发送数据作好准备了。为此，硬件相应地对 CAN_MSR 寄存器的 INAK 位清’0’。 软件对该位置 1 可使 CAN 从正常工作模式进入初始化模式：一旦当前的 CAN 活动(发送或接收)结束，CAN 就进入初始化模式。相应地，硬件对 CAN_MSR 寄存器的 INAK 位置’1’。

CAN 主状态寄存器 (CAN_MSR)

位 31:12	保留位，硬件强制为 0
位 11	RX：CAN 接收电平 (CAN Rx signal) 该位反映 CAN 接收引脚(CAN_RX)的实际电平。
位 10	SAMP：上次采样值 (Last sample point) CAN 接收引脚的上次采样值(对应于当前接收位的值)。
位 9	RXM：接收模式 (Receive mode) 该位为’1’表示 CAN 当前为接收器。
位 8	TXM：发送模式 (Transmit mode) 该位为’1’表示 CAN 当前为发送器。
位 7:5	保留位，硬件强制为 0。
位 4	SLAKI：睡眠确认中断 (Sleep acknowledge interrupt) 当 SLKIE=1，一旦 CAN 进入睡眠模式硬件就对该位置’1，紧接着相应的中断被触发。当设置该位为’1’时，如果设置了 CAN_IER 寄存器中的 SLKIE 位，将产生一个状态改变中断。 软件可对该位清’0’，当 SLAK 位被清’0’时硬件也对该位清’0’。 注: 当 SLKIE=0, 不应该查询该位，而应该查询 SLAK 位来获知睡眠状态。
位 3	WKUI：唤醒中断挂号 (Wakeup interrupt) 当 CAN 处于睡眠状态，一旦检测到帧起始位(SOF)，硬件就置该位为’1’；并且如果 CAN_IER 寄存器的 WKUIE 位为’1’，则产生一个状态改变中断。 该位由软件清’0’。
位 2	ERRI：出错中断挂号 (Error interrupt) 当检测到错误时，CAN_ESR 寄存器的某位被置’1’，如果 CAN_IER 寄存器的相应中断使能位也被置’1’时，则硬件对该位置’1’；如果 CAN_IER 寄存器的 ERRIE 位为’1’，则产生状态改变中断。 该位由软件清’0’。
位 1	SLAK：睡眠模式确认 该位由硬件置’1’，指示软件 CAN 模块正处于睡眠模式。 该位是对软件请求进入睡眠模式的确认 (对 CAN_MCR 寄存器的 SLEEP 位置’1’)。 当 CAN 退出睡眠模式时硬件对该位清’0’ (需要跟 CAN 总线同步)。 这里跟 CAN 总线同步是指，硬件需要在 CAN 的 RX 引脚上检测到连续的 11 位隐性位。 注: 通过软件或硬件对 CAN_MCR 的 SLEEP 位清’0’，将启动退出睡眠模式的过程。有关清除 SLEEP 位的详细信息，参见 CAN_MCR 寄存器的 AWUM 位的描述。
位 0	INAK：初始化确认 位 0 该位由硬件置’1’，指示软件 CAN 模块正处于初始化模式。 该位是对软件请求进入初始化模式的确认(对 CAN_MCR 寄存器的 INRQ 位置’1’)。

	当 CAN 退出初始化模式时硬件对该位清’0’ (需要跟 CAN 总线同步)。这里跟 CAN 总线同步是指，硬件需要在 CAN 的 RX 引脚上检测到连续的 11 位隐性位。

CAN 发送状态寄存器 (CAN_TSR)



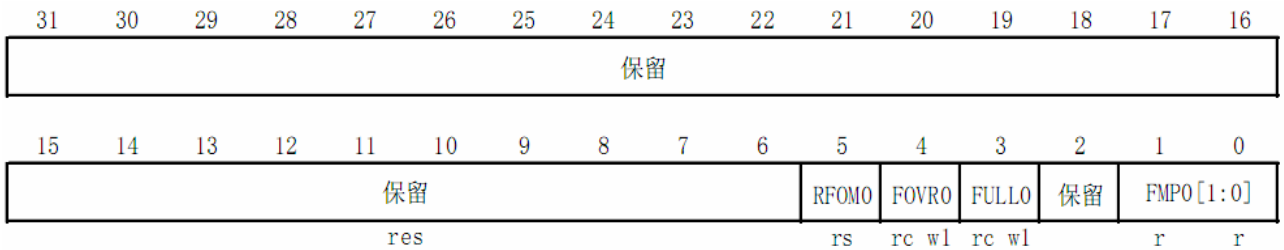
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOW2	LOW1	LOW0	TME2	TME1	TME0	CODE[1:0]		ABRQ2	保留			TERR2	ALST2	TXOK2	RQCP2
r	r	r	r	r	r	r	r	rs	res			rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABRQ1	保留			TERR1	ALST1	TXOK1	RQCP1	ABRQ0	保留			TERR0	ALST0	TXOK0	RQCP0
rs	res			rc_w1	rc_w1	rc_w1	rc_w1	rs	res			rc_w1	rc_w1	rc_w1	rc_w1

位 31	LOW2: 邮箱 2 最低优先级标志 (Lowest priority flag for mailbox 2) 当多个邮箱在等待发送报文，且邮箱 2 的优先级最低时，硬件对该位置’1’。
位 30	LOW1: 邮箱 1 最低优先级标志 (Lowest priority flag for mailbox 1) 当多个邮箱在等待发送报文，且邮箱 1 的优先级最低时，硬件对该位置’1’。
位 29	LOW0: 邮箱 0 最低优先级标志 (Lowest priority flag for mailbox 0) 当多个邮箱在等待发送报文，且邮箱 0 的优先级最低时，硬件对该位置’1’。 注：如果只有 1 个邮箱在等待，则 LOW[2:0]被清’0’。
位 28	TME2: 发送邮箱 2 空 (Transmit mailbox 2 empty) 当邮箱 2 中没有等待发送的报文时，硬件对该位置’1’。
位 27	TME1: 发送邮箱 1 空 (Transmit mailbox 1 empty) 当邮箱 1 中没有等待发送的报文时，硬件对该位置’1’。
位 26	TME0: 发送邮箱 0 空 (Transmit mailbox 0 empty) 当邮箱 0 中没有等待发送的报文时，硬件对该位置’1’。
位 25:24	CODE[1:0]: 邮箱号 (Mailbox code) 当有至少 1 个发送邮箱为空时，这 2 位表示下一个空的发送邮箱号。 当所有的发送邮箱都为空时，这 2 位表示优先级最低的那个发送邮箱号。
位 23	ABRQ2: 邮箱 2 中止发送 (Abort request for mailbox 2) 软件对该位置’1’，可以中止邮箱 2 的发送请求，当邮箱 2 的发送报文被清除时硬件对该位清’0’。如果邮箱 2 中没有等待发送的报文，则对该位置’1’没有任何效果。
位 22:20	保留位，硬件强制其值为 0
位 19	TERR2: 邮箱 2 发送失败 (Transmission error of mailbox 2) 当邮箱 2 因为出错而导致发送失败时，对该位置’1’。
位 18	ALST2: 邮箱 2 仲裁丢失 (Arbitration lost for mailbox 2) 当邮箱 2 因为仲裁丢失而导致发送失败时，对该位置’1’。
位 17	TXOK2: 邮箱 2 发送成功 (Transmission OK of mailbox 2) 每次在邮箱 2 进行发送尝试后，硬件对该位进行更新： 0：上次发送尝试失败； 1：上次发送尝试成功。 当邮箱 2 的发送请求被成功完成后，硬件对该位置’1’。
位 16	RQCP2: 邮箱 2 请求完成 (Request completed mailbox 2) 当上次对邮箱 2 的请求(发送或中止)完成后，硬件对该位置’1’。 软件对该位写’1’可以对其清’0’；当硬件接收到发送请求时也对该位清’0’(CAN_TI2R 寄存器的 TXRQ 位被置’1’)。 该位被清’0’时，邮箱 2 的其它发送状态位(TXOK2, ALST2 和 TERR2)也被清’0’。

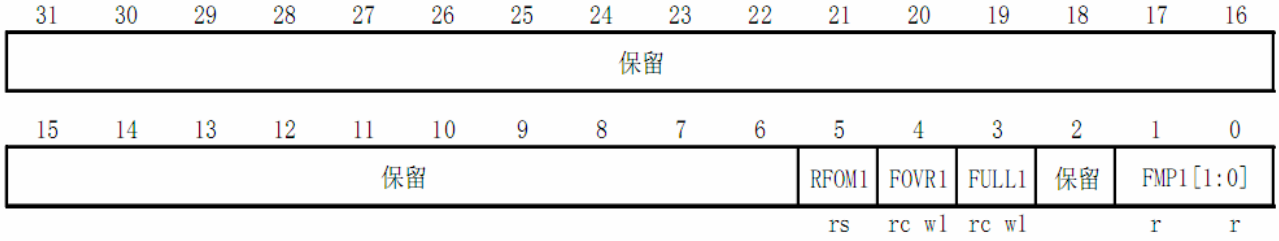
位 15	ABRQ1: 邮箱 1 中止发送 (Abort request for mailbox 1) 软件对该位置’1’，可以中止邮箱 1 的发送请求，当邮箱 1 的发送报文被清除时硬件对该位清’0’。 如果邮箱 1 中没有等待发送的报文，则对该位置’1’没有任何效果。
位 14:12	保留位，硬件强制其值为 0
位 11	TERR1: 邮箱 1 发送失败 (Transmission error of mailbox 1) 当邮箱 1 因为出错而导致发送失败时，对该位置’1’。
位 10	ALST1: 邮箱 1 仲裁丢失 (Arbitration lost for mailbox 1) 当邮箱 1 因为仲裁丢失而导致发送失败时，对该位置’1’。
位 9	TXOK1: 邮箱 1 发送成功 (Transmission OK of mailbox 1) 每次在邮箱 1 进行发送尝试后，硬件对该位进行更新： 0：上次发送尝试失败； 1：上次发送尝试成功。 当邮箱 1 的发送请求被成功完成后，硬件对该位置’1’。
位 8	RQCP1: 邮箱 1 请求完成 (Request completed mailbox 1) 当上次对邮箱 1 的请求(发送或中止)完成后，硬件对该位置’1’。 软件对该位写’1’可以对其清’0’；当硬件接收到发送请求时也对该位清’0’(CAN_TI1R 寄存器的 TXRQ 位被置’1’)。 该位被清’0’时，邮箱 1 的其它发送状态位(TXOK1, ALST1 和 TERR1)也被清’0’。
位 7	ABRQ0: 邮箱 0 中止发送 (Abort request for mailbox 0) 软件对该位置’1’可以中止邮箱 0 的发送请求，当邮箱 0 的发送报文被清除时硬件对该位清’0’。 如果邮箱 0 中没有等待发送的报文，则对该位置 1 没有任何效果。
位 6:4	保留位，硬件强制其值为 0
位 3	TERR0: 邮箱 0 发送失败 (Transmission error of mailbox 0) 当邮箱 0 因为出错而导致发送失败时，对该位置’1’。
位 2	ALST0: 邮箱 0 仲裁丢失 (Arbitration lost for mailbox 0) 当邮箱 0 因为仲裁丢失而导致发送失败时，对该位置’1’。
位 1	TXOK0: 邮箱 0 发送成功 (Transmission OK of mailbox 0) 每次在邮箱 0 进行发送尝试后，硬件对该位进行更新： 0：上次发送尝试失败； 1：上次发送尝试成功。 当邮箱 0 的发送请求被成功完成后，硬件对该位置’1’。
位 0	RQCP0: 邮箱 0 请求完成 (Request completed mailbox 0) 当上次对邮箱 0 的请求(发送或中止)完成后，硬件对该位置’1’。 软件对该位写’1’可以对其清’0’；当硬件接收到发送请求时也对该位清’0’(CAN_TI0R 寄存器的 TXRQ 位被置’1’)。 该位被清’0’时，邮箱 0 的其它发送状态位(TXOK0, ALST0 和 TERR0)也被清’0’。

CAN 接收 FIFO 0 寄存器 (CAN_RF0R)



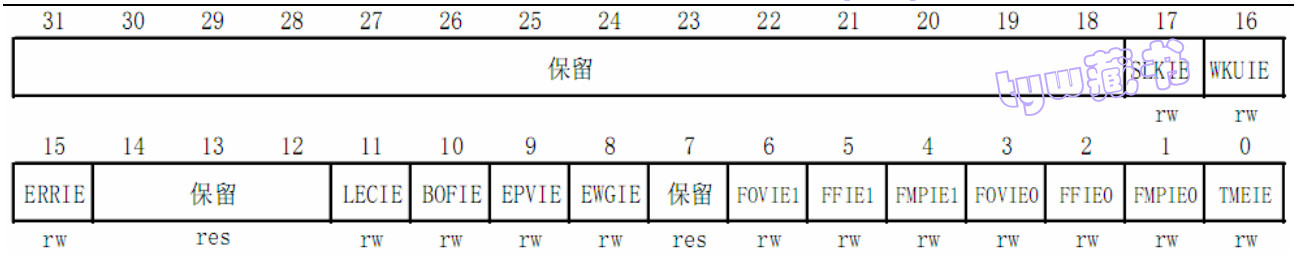
位31:6	保留位，硬件强制为0
位5	**RFOM0:** 释放接收FIFO 0输出邮箱 (Release FIFO 0 output mailbox) 软件通过对该位置'1'来释放接收FIFO的输出邮箱。如果接收FIFO为空，那么对该位置'1'没有任何效果，即只有当FIFO中有报文时对该位置'1'才有意义。如果FIFO中有2个以上的报文，由于FIFO的特点，软件需要释放输出邮箱才能访问第2个报文。 当输出邮箱被释放时，硬件对该位清'0'。
位4	**FOVR0:** FIFO 0溢出 (FIFO 0 overrun) 当FIFO 0已满，又收到新的报文且报文符合过滤条件，硬件对该位置'1'。 该位由软件清'0'。
位3	**FULL0:** FIFO 0满 (FIFO 0 full) 当FIFO 0中有3个报文时，硬件对该位置'1'。 该位由软件清'0'。
位2	保留位，硬件强制其值为0
位1:0	**FMP0[1:0]:** FIFO 0 报文数目 (FIFO 0 message pending) FIFO 0报文数目这2位反映了当前接收FIFO 0中存放的报文数目。 每当1个新的报文被存入接收FIFO 0，硬件就对FMP0加1。 每当软件对RFOM0位写'1'来释放输出邮箱，FMP0就被减1，直到其为0。

CAN 接收 FIFO 1 寄存器(CAN_RF1R)



位 31:6	保留位，硬件强制为 0
位 5	RFOM1: 释放接收 FIFO 1 输出邮箱 (Release FIFO 1 output mailbox) 软件通过对该位置’1’来释放接收 FIFO 的输出邮箱。如果接收 FIFO 为空，那么对该位置’1’没有任何效果，即只有当 FIFO 中有报文时对该位置’1’才有意义。如果 FIFO 中有 2 个以上的报文，由于 FIFO 的特点，软件需要释放输出邮箱才能访问第 2 个报文。 当输出邮箱被释放时，硬件对该位清’0’。
位 4	FOVR1: FIFO 1 溢出 (FIFO 1 overrun) 当 FIFO 1 已满，又收到新的报文且报文符合过滤条件，硬件对该位置’1’。 该位由软件清’0’。
位 3	FULL1: FIFO 1 满 (FIFO 1 full) 当 FIFO 1 中有 3 个报文时，硬件对该位置’1’。 该位由软件清’0’。
位 2	保留位，硬件强制其值为 0
位 1:0	FMP1[1:0]: FIFO 1 报文数目 (FIFO 1 message pending) FIFO 1 报文数目这 2 位反映了当前接收 FIFO 1 中存放的报文数目。 每当 1 个新的报文被存入接收 FIFO 1，硬件就对 FMP1 加 1。 每当软件对 RFOM1 位写 1 来释放输出邮箱，FMP1 就被减 1，直到其为 0。

CAN 中断使能寄存器 (CAN_IER)



位 31:18	保留位，硬件强制为 0
位 17	SLKIE：睡眠中断使能 (Sleep interrupt enable) 0：当 SLAKI 位被置’1’时，不产生中断； 1：当 SLAKI 位被置’1’时，产生中断。
位 16	WKUIE: 唤醒中断使能 (Wakeup interrupt enable) 0：当 WKUI 位被置’1’时，不产生中断； 1：当 WKUI 位被置’1’时，产生中断。
位 15	ERRIE: 错误中断使能 (Error interrupt enable) 0：当 CAN_ESR 寄存器有错误挂号时，不产生中断； 1：当 CAN_ESR 寄存器有错误挂号时，产生中断。
位 14:12	保留位，硬件强制为 0。
位 11	LECIE: 上次错误号中断使能 (Last error code interrupt enable) 0：当检测到错误，硬件设置 LEC[2:0]时，不设置 ERRI 位； 1：当检测到错误，硬件设置 LEC[2:0]时，设置 ERRI 位为’1’。
位 10	BOFIE: 离线中断使能 (Bus-off interrupt enable) 0：当 BOFF 位被置’1’时，不设置 ERRI 位； 1：当 BOFF 位被置’1’时，设置 ERRI 位为’1’。
位 9	EPVIE: 错误被动中断使能 (Error Passive Interrupt Enable) 0：当 EPVF 位被置’1’时，不设置 ERRI 位； 1：当 EPVF 位被置’1’时，设置 ERRI 位为’1’。
位 8	EWGIE: 错误警告中断使能 (Error warning interrupt enable) 0：当 EWGF 位被置’1’时，不设置 ERRI 位； 1：当 EWGF 位被置’1’时，设置 ERRI 位为’1’。
位 7	保留位，硬件强制为 0
位 6	FOVIE1: FIFO 1 溢出中断使能 (FIFO overrun interrupt enable) 0：当 FIFO 1 的 FOVR 位被置’1’时，不产生中断； 1：当 FIFO 1 的 FOVR 位被置’1’时，产生中断。
位 5	FFIE1: FIFO 1 满中断使能 (FIFO full interrupt enable) 0：当 FIFO 1 的 FULL 位被置’1’时，不产生中断； 1：当 FIFO 1 的 FULL 位被置’1’时，产生中断。
位 4	FMPIE1: FIFO 1 消息挂号中断使能 (FIFO message pending interrupt enable) 0：当 FIFO 1 的 FMP[1:0]位为非 0 时，不产生中断； 1：当 FIFO 1 的 FMP[1:0]位为非 0 时，产生中断。
位 3	FOVIE0: FIFO 0 溢出中断使能 (FIFO overrun interrupt enable) 0：当 FIFO 0 的 FOVR 位被置’1’时，不产生中断； 1：当 FIFO 0 的 FOVR 位被置’1’时，产生中断。
位 2	FFIE0: FIFO 0 满中断使能 (FIFO full interrupt enable) 0：当 FIFO 0 的 FULL 位被置’1’时，不产生中断； 1：当 FIFO 0 的 FULL 位被置’1’时，产生中断。

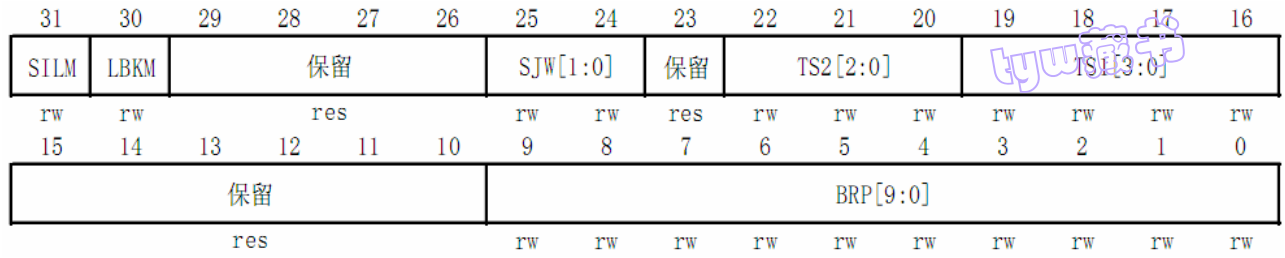
位 1	FMPIE0: FIFO 0 消息挂号中断使能 (FIFO message pending interrupt enable) 0：当 FIFO 0 的 FMP[1:0]位为非 0 时，不产生中断； 1：当 FIFO 0 的 FMP[1:0]位为非 0 时，产生中断。
位 0	TMEIE: 发送邮箱空中断使能 (Transmit mailbox empty interrupt enable) 0：当 RQCPx 位被置’1’时，不产生中断； 1：当 RQCPx 位被置’1’时，产生中断。

CAN 错误状态寄存器 (CAN_ESR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
REC[7:0]								TEC[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留									LEC[2:0]			保留	BOFF	EPVF	WEGF
									rw	rw	rw		r	r	r

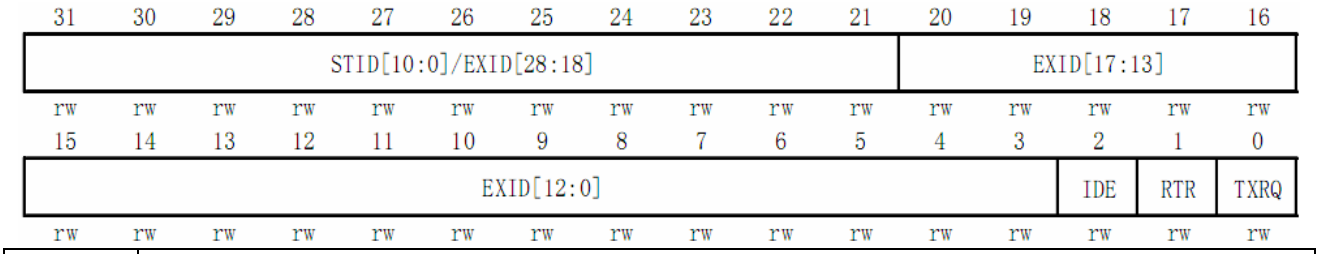
位 31:24	REC[7:0]: 接收错误计数器 (Receive error counter) 这个计数器按照 CAN 协议的故障界定机制的接收部分实现。按照 CAN 的标准，当接收出错时，根据出错的条件，该计数器加 1 或加 8；而在每次接收成功后，该计数器减 1，或当该计数器的值大于 127 时，设置它的值为 120。当该计数器的值超过 127 时，CAN 进入错误被动状态。
位 23:16	TEC[7:0]: 9 位发送错误计数器的低 8 位 (Least significant byte of the 9-bit transmit error counter) 与上面相似，这个计数器按照 CAN 协议的故障界定机制的发送部分实现。
位 15:7	保留位，硬件强制为 0。
位 6:4	LEC[2:0]: 上次错误代码 (Last error code) 在检测到 CAN 总线上发生错误时，硬件根据出错情况设置。当报文被正确发送或接收后，硬件清除其值为’0’。 硬件没有使用错误代码 7，软件可以设置该值，从而可以检测代码的更新。 000: 没有错误； 001: 位填充错； 010: 格式(Form)错； 011: 确认(ACK)错； 100: 隐性位错； 101: 显性位错； 110: CRC 错； 111: 由软件设置。
位 3	保留位，硬件强制为 0。
位 2	BOFF: 离线标志 (Bus-off flag) 当进入离线状态时，硬件对该位置’1’。当发送错误计数器 TEC 溢出，即大于 255 时，CAN 进入离线状态。请参考 22.7.6。
位 1	EPVF: 错误被动标志 (Error passive flag) 当出错次数达到错误被动的阈值时，硬件对该位置’1’。 (接收错误计数器或发送错误计数器的值>127)。
位 0	EWGF: 错误警告标志 (Error warning flag) 当出错次数达到警告的阈值时，硬件对该位置’1’。 (接收错误计数器或发送错误计数器的值≥96)。

CAN 位时序寄存器 (CAN_BTR)



位 31	SILM: 静默模式(用于调试) (Silent mode (debug)) 0: 正常状态； 1: 静默模式。
位 30	LBKM: 环回模式(用于调试) (Loop back mode (debug)) 0: 禁止环回模式； 1: 允许环回模式。
位 29:26	保留位，硬件强制为 0。
位 25:24	SJW[1:0]: 重新同步跳跃宽度 (Resynchronization jump width) 为了重新同步，该位域定义了 CAN 硬件在每位中可以延长或缩短多少个时间单元的上限。 tRJW = tCAN x (SJW[1:0] + 1)。
位 23	保留位，硬件强制为 0。
位 22:20	TS2[2:0]: 时间段 2 (Time segment 2) 该位域定义了时间段 2 占用了多少个时间单元 tBS2 = tCAN x (TS2[2:0] + 1)。
位 19:16	TS1[3:0]: 时间段 1 (Time segment 1) 该位域定义了时间段 1 占用了多少个时间单元 tBS1 = tCAN x (TS1[3:0] + 1) 22.7.7 关于位时间特性的详细信息，请参考 节位时间特性。
位 15:10	保留位，硬件强制为 0。
位 9:0	BRP[9:0]: 波特率分频器 (Baud rate prescaler) 该位域定义了时间单元(tq)的时间长度 tq = (BRP[9:0]+1) x tPCLK

发送邮箱标识符寄存器 (CAN_TIxR) (x=0..2)



位 31:21	STID[10:0]/EXID[28:18]: 标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据 IDE 位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。
位 20:3	EXID[17:0]: 扩展标识符 (Extended identifier) 扩展身份标识的低字节。
位 2	IDE: 标识符选择 (Identifier extension) 该位决定发送邮箱中报文使用的标识符类型 0：使用标准标识符； 1：使用扩展标识符。

位 1	RTR：远程发送请求 (Remote transmission request) 0：数据帧； 1：远程帧。
位 0	TXRQ：发送数据请求 (Transmit mailbox request) 由软件对其置’1’，来请求发送邮箱的数据。当数据发送完成，邮箱为空时，硬件对其清’0’。

lyw藏书

发送邮箱数据长度和时间戳寄存器 (CAN_TDTxR) (x=0..2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TIME[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留							TGT	保留				DLC[3:0]			
res							rw	res				rw	rw	rw	rw

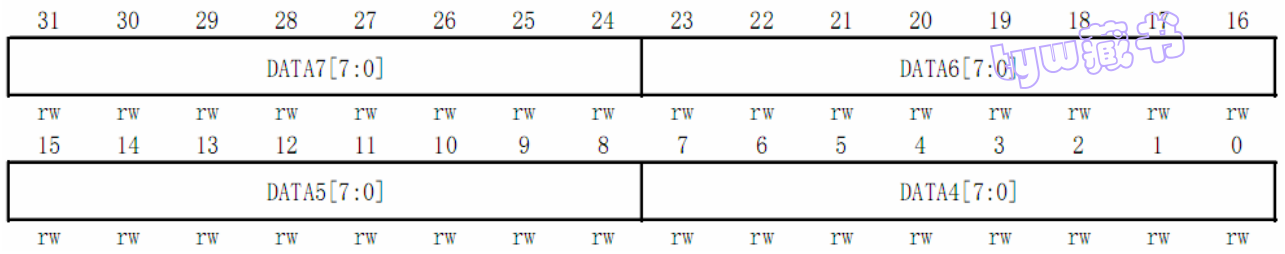
位31:16	**TIME[15:0]:** 报文时间戳 (Message time stamp) 该域包含了，在发送该报文SOF的时刻，16位定时器的值。
位15:9	保留位
位8	**TGT:** 发送时间戳 (Transmit global time) 只有在CAN处于时间触发通信模式，即CAN_MCR寄存器的TTCM位为'1'时，该位才有效。 0：不发送时间戳TIME[15:0]； 1：发送时间戳TIME[15:0]。在长度为8的报文中，时间戳TIME[15:0]是最后2个发送的字节：TIME[7:0]作为第7个字节，TIME[15:8]为第8个字节，它们替换了写入CAN_TDHxR[31:16]的数据(DATA6[7:0]和DATA7[7:0])。为了把时间戳的2个字节发送出去，DLC必须编程为8。
位7:4	保留位。
位3:0	**DLC[15:0]:** 发送数据长度 (Data length code) 该域指定了数据报文的数据长度或者远程帧请求的数据长度。1个报文包含0到8个字节数据，而这由DLC决定。

发送邮箱低字节数据寄存器 (CAN_TDLxR) (x=0..2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

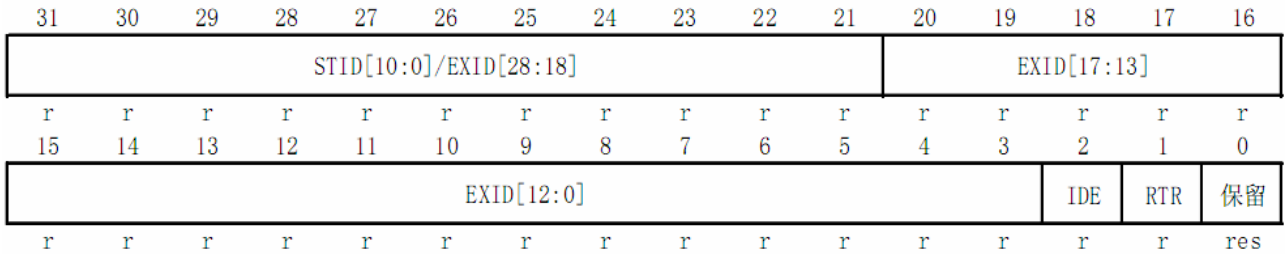
位 31:24	DATA3[7:0]：数据字节 3 (Data byte 3) 报文的数据字节 3。
位 23:16	DATA2[7:0]：数据字节 2 (Data byte 2) 报文的数据字节 2。
位 15:8	DATA1[7:0]：数据字节 1 (Data byte 1) 报文的数据字节 1。
位 7:0	DATA0[7:0]：数据字节 0 (Data byte 0) 报文的数据字节 0。 报文包含 0 到 8 个字节数据，且从字节 0 开始。

发送邮箱高字节数据寄存器 (CAN_TDHxR) (x=0..2)



位31:24	**DATA7[7:0]**：数据字节7 (Data byte 7) 报文的数据字节7 **注**：如果CAN_MCR寄存器的TTCM位为'1'，且该邮箱的TGT位也为'1'，那么DATA7和DATA6将被TIME时间戳代替。
位23:16	**DATA6[7:0]**：数据字节6 (Data byte 6) 报文的数据字节6。
位15:8	**DATA5[7:0]**：数据字节5 (Data byte 5) 报文的数据字节5。
位7:0	**DATA4[7:0]**：数据字节4 (Data byte 4) 报文的数据字节4。

接收 FIFO 邮箱标识符寄存器 (CAN_RIxR) (x=0..1)



位 31:21	STID[10:0]/EXID[28:18]:　标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据 IDE 位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。
位 20:3	EXID[17:0]: 扩展标识符 (Extended identifier) 扩展标识符的低字节。
位 2	IDE: 标识符选择 (Identifier extension) 该位决定接收邮箱中报文使用的标识符类型 0：使用标准标识符； 1：使用扩展标识符。
位 1	RTR: 远程发送请求 (Remote transmission request) 0：数据帧； 1：远程帧。
位 0	保留位。

接收 FIFO 邮箱数据长度和时间戳寄存器 (CAN_RDTxR) (x=0..1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TIME[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMI[7:0]								保留				DLC[3:0]			
r	r	r	r	r	r	r	r	res				r	r	r	r

位31:16	**TIME[15:0]:** 报文时间戳 (Message time stamp) 该域包含了，在接收该报文SOF的时刻，16位定时器的值。
位15:8	**FMI[15:0]:** 过滤器匹配序号 (Filter match index) 这里是存在邮箱中的信息传送的过滤器序号。关于标识符过滤的细节，请参考22.7.4中有关过滤器匹配序号。
位7:4	保留位，硬件强制为0。
位3:0	**DLC[15:0]:** 接收数据长度 (Data length code) 该域表明接收数据帧的数据长度(0～8)。对于远程帧请求，数据长度DLC恒为0。

接收 FIFO 邮箱低字节数据寄存器 (CAN_RDLxR) (x=0..1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:24	**DATA3[7:0]：**数据字节3 (Data byte 3) 报文的数据字节3。
位23:16	**DATA2[7:0]：**数据字节2 (Data byte 2) 报文的数据字节2。
位15:8	**DATA1[7:0]：**数据字节1 (Data byte 1) 报文的数据字节1。
位7:0	**DATA0[7:0]：**数据字节0 (Data byte 0) 报文的数据字节0。 报文包含0到8个字节数据，且从字节0开始。

接收 FIFO 邮箱高字节数据寄存器 (CAN_RDHxR) (x=0..1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA7[7:0]								DATA6[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA5[7:0]								DATA4[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:24	**DATA7[7:0]： 数据字节 7 (Data byte 7)** **报文的数据字节 7**
位 23:16	**DATA6[7:0]： 数据字节 6 (Data byte 6)**

	报文的数据字节 6。
位 15:8	**DATA5[7:0]：数据字节 5 (Data byte 5)** **报文的数据字节 5。**
位 7:0	**DATA4[7:0]：数据字节 4 (Data byte 4)** **报文的数据字节 4。**

tyw藏书

CAN 过滤器寄存器

CAN 过滤器主控寄存器 (CAN_FMR)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留															FINIT
															rw

保留	CAN2SB[5:0]	保留	FINIT
res	rw	res	rw

位31:14	保留位，强制为复位值。
位13:8	**CAN2SB[5:0]**：CAN2开始组 (CAN2 start bank) 这些位由软件置'1'、清'0'。它们定义了CAN2(从)接口的开始组，范围是1~27。 注：这些位只出现在互联型产品中，其它产品中为保留位。
位7:1	保留位，强制为复位值。
位0	**FINIT**：过滤器初始化模式 (Filter init mode) 针对所有过滤器组的初始化模式设置。 0: 过滤器组工作在正常模式； 1: 过滤器组工作在初始化模式。

CAN 过滤器模式寄存器 (CAN_FM1R)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留位，硬件强制为0
位13:0	**FBMx**：过滤器模式 (Filter mode) 过滤器组x的工作模式。 0: 过滤器组x的2个32位寄存器工作在标识符屏蔽位模式； 1: 过滤器组x的2个32位寄存器工作在标识符列表模式。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

CAN 过滤器位宽寄存器 (CAN_FS1R)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留位，硬件强制为0
位13:0	**FSCx**：过滤器位宽设置 (Filter scale configuration) 过滤器组x(13～0)的位宽。 0：过滤器位宽为2个16位； 1：过滤器位宽为单个32位。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

CAN 过滤器 FIFO 关联寄存器 (CAN_FFA1R)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:14	**保留位，硬件强制为 0。**
位 13:0	**FFAx：过滤器位宽设置 (Filter FIFO assignment for filter x)** **报文在通过了某过滤器的过滤后，将被存放到其关联的 FIFO 中。** **0：过滤器被关联到 FIFO0；** **1：过滤器被关联到 FIFO1。** **注：位 27:14 只出现在互联型产品中，其它产品为保留位。**

CAN 过滤器激活寄存器 (CAN_FA1R)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FACT27	FACT26	FACT25	FACT24	FACT23	FACT22	FACT21	FACT20	FACT19	FACT18	FACT17	FACT16
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FACT15	FACT14	FACT13	FACT12	FACT11	FACT10	FACT9	FACT8	FACT7	FACT6	FACT5	FACT4	FACT3	FACT2	FACT1	FACT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:14	保留位，硬件强制为0。
位13:0	**FACTx**：过滤器激活 (Filter active) 软件对某位设置'1'来激活相应的过滤器。只有对FACTx位清'0'，或对CAN_FMR寄存器的FINIT位设置'1'后，才能修改相应的过滤器寄存器x(CAN_FxR[0:1])。 0：过滤器被禁用； 1：过滤器被激活。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

CAN 过滤器组 i 的寄存器 x (CAN_FiRx) (互联产品中 i=0..27，其它产品中 i＝0..13；x=1..2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FB31	FB30	FB29	FB28	FB27	FB26	FB25	FB24	FB23	FB22	FB21	FB20	FB19	FB18	FB17	FB16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FB15	FB14	FB13	FB12	FB11	FB10	FB9	FB8	FB7	FB6	FB5	FB4	FB3	FB2	FB1	FB0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:0	**FB[31:0]**：过滤器位 (Filter bits) **标识符模式** 寄存器的每位对应于所期望的标识符的相应位的电平。 0: 期望相应位为显性位； 1: 期望相应位为隐性位。 **屏蔽位模式** 寄存器的每位指示是否对应的标识符寄存器位一定要与期望的标识符的相应位一致。 0: 不关心，该位不用于比较； 1: 必须匹配，到来的标识符位必须与滤波器对应的标识符寄存器位相一致。

CAN相关知识

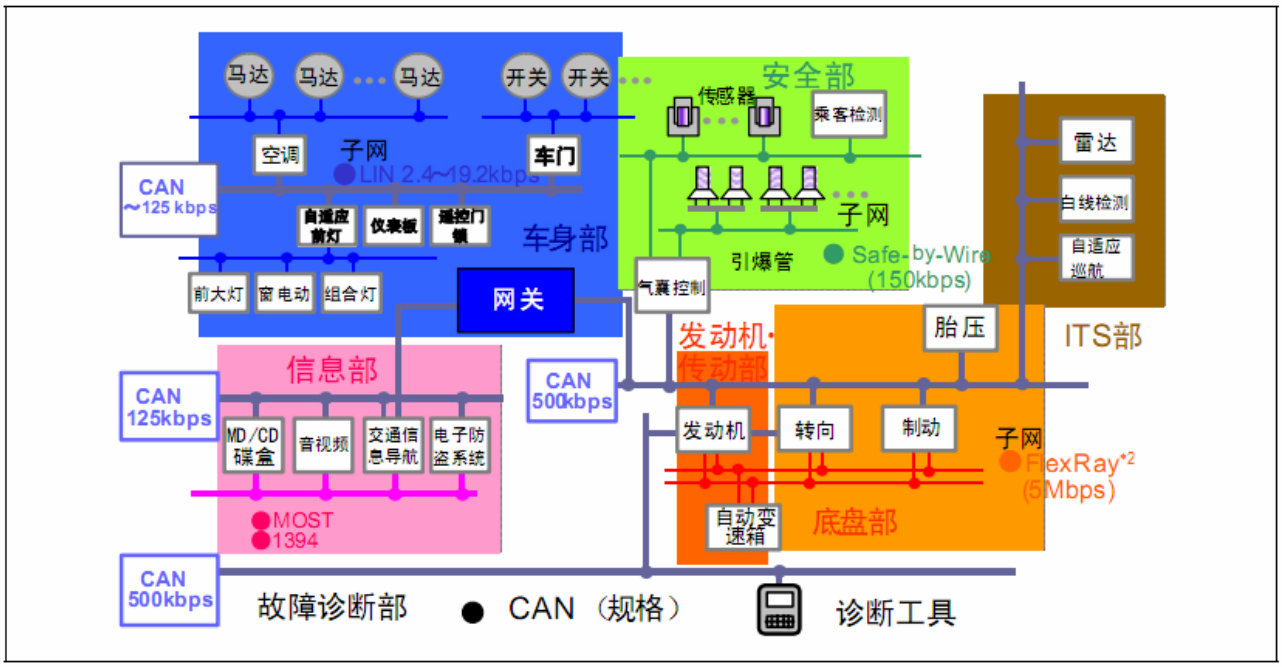
CAN介绍

CAN 是 Controller Area Network 的缩写（以下称为 CAN），是 ISO 国际标准化的串行通信协议。

在当前的汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，现在在欧洲已是汽车网络的标准协议。

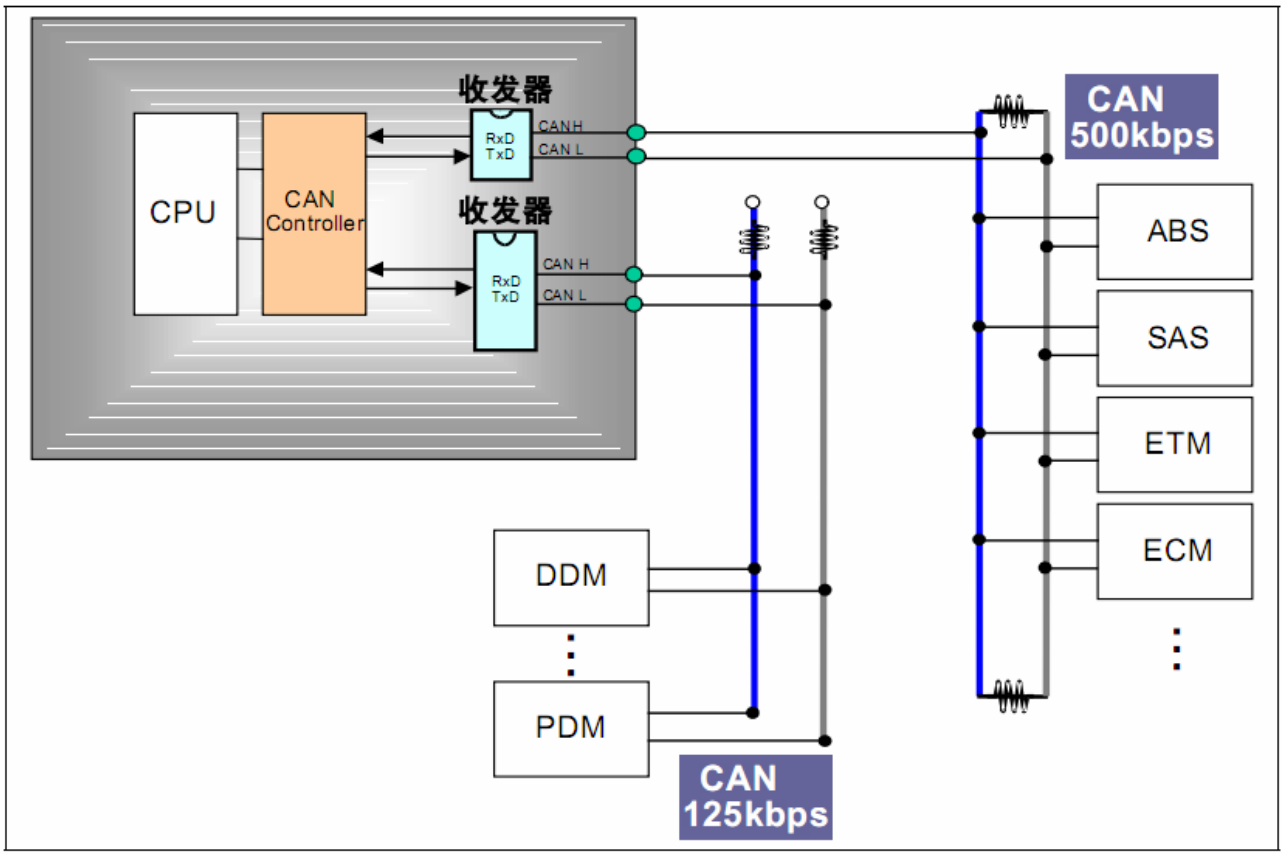
现在，CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。

图 1 是车载网络的构想示意图。



CAN总线拓扑图

CAN 控制器根据两根线上的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。发送方通过使总线电平发生变化，将消息发送给接收方。



CAN的特点

(1)　多主控制

在总线空闲时，所有的单元都可开始发送消息（多主控制）。

最先访问总线的单元可获得发送权（CSMA/CA 方式）。

多个单元同时开始发送时，发送高优先级 ID 消息的单元可获得发送权。

(2)　消息的发送

在 CAN 协议中，所有的消息都以固定的格式发送。总线空闲时，所有与总线相连的单元都可以开始发送新消息。两个以上的单元同时开始发送消息时，根据标识符（Identifier　以下称为 ID）决定优先级。ID 并不是表示发送的目的地址，而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时，对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜（被判定为优先级最高）的单元可继续发送消息，仲裁失利的单元则立刻停止发送而进行接收工作。

(3)　系统的柔软性

与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时，连接在总线上的其它单元的软硬件及应用层都不需要改变。

(4)　通信速度

根据整个网络的规模，可设定适合的通信速度。

在同一网络中，所有单元必须设定成统一的通信速度。即使有一个单元的通信速度与其它的不一样，此单元也会输出错误信号，妨碍整个网络的通信。不同网络间则可以有不同的通信速度。

(5)　远程数据请求

可通过发送“遥控帧”　请求其他单元发送数据。

(6)　错误检测功能・错误通知功能・错误恢复功能

所有的单元都可以检测错误（错误检测功能）。

检测出错误的单元会立即同时通知其他所有单元（错误通知功能）。

正在发送消息的单元一旦检测出错误，会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止（错误恢复功能）。

(7)　故障封闭

CAN 可以判断出错误的类型是总线上暂时的数据错误（如外部噪声等）还是持续的数据错误（如单元内部故障、驱动器故障、断线等）。由此功能，当总线上发生持续数据错误时，可将引起此故障的单元从总线上隔离出去。

(8)　连接

CAN 总线是可同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受总线上的时间延迟及电气负载的限制。降低通信速度，可连接的单元数增加；提高通信速度，则可连接的单元数减少。

CAN协议及标准规格

CAN 协议经 ISO 标准化后有 ISO11898 标准和 ISO11519-2 标准两种。ISO11898 和 ISO11519-2 标准对于数据链路层的定义相同，但物理层不同。

(1)　关于 ISO11898

ISO11898 是通信速度为 125kbps-1Mbps 的 CAN 高速通信标准。

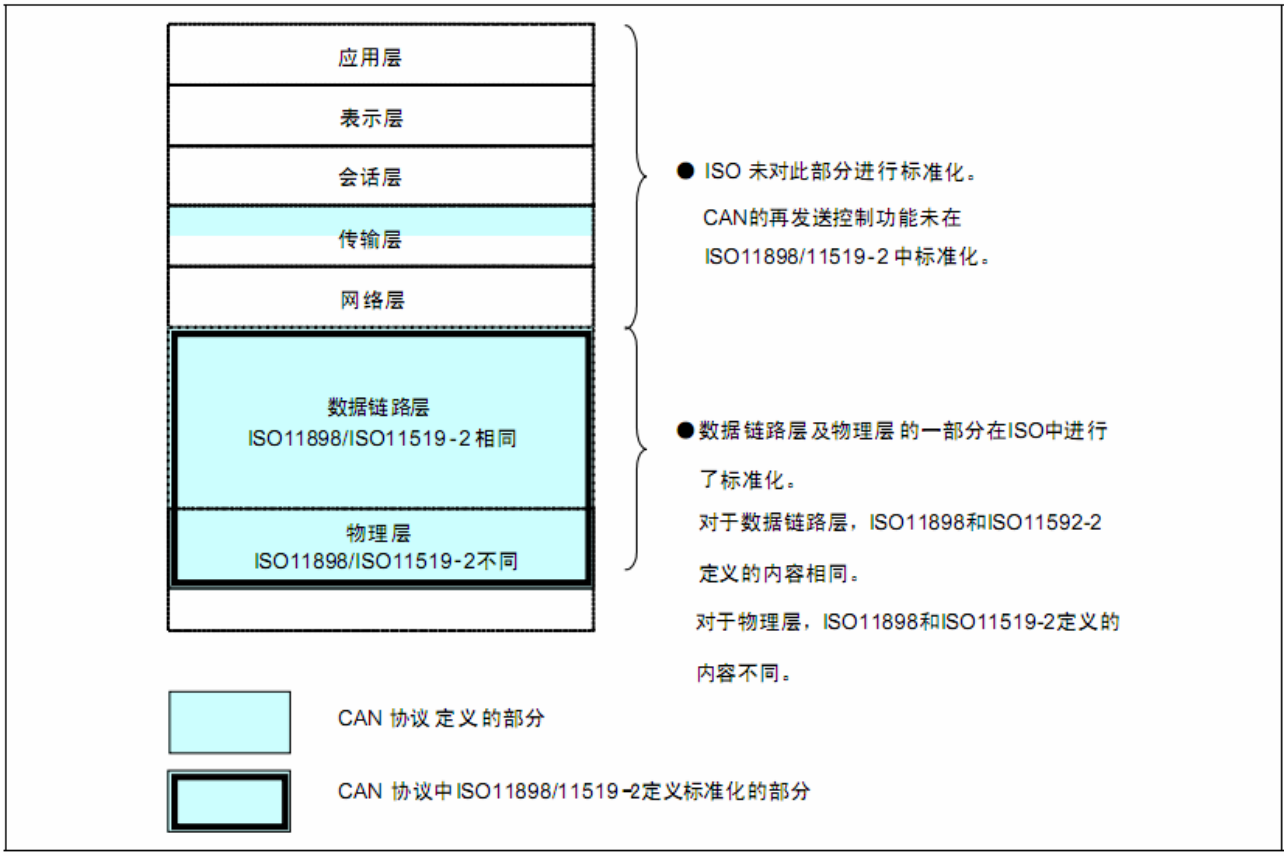
目前，ISO11898 追加新规约后，成为 ISO11898-1 新标准。

(2)　关于 ISO11519

ISO11519 是通信速度为 125kbps 以下的 CAN 低速通信标准。

ISO11519-2 是 ISO11519-1 追加新规约后的版本。

下图为 CAN 协议和 ISO11898 及 ISO11519-2 标准的范围



CAN2.0B 标准帧

CAN 标准帧信息为 11 个字节，包括两部分：信息和数据部分。前 3 个字节为信息部分。

	7	6	5	4	3	2	1	0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码）				ID.10-ID.3			
字节 3	ID.2-ID.0			X	X	X	X	X
字节 4	数据 1							
字节 5	数据 2							
字节 6	数据 3							
字节 7	数据 4							
字节 8	数据 5							
字节 9	数据 6							
字节 10	数据 7							
字节 11	数据 8							

- **字节 1 为帧信息。第 7 位（FF）表示帧格式，在标准帧中，FF＝0；第 6 位（RTR）表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示在数据帧时实际的数据长度。**
- **字节 2、3 为报文识别码，11 位有效。**
- **字节 4～11 为数据帧的实际数据，远程帧时无效。**

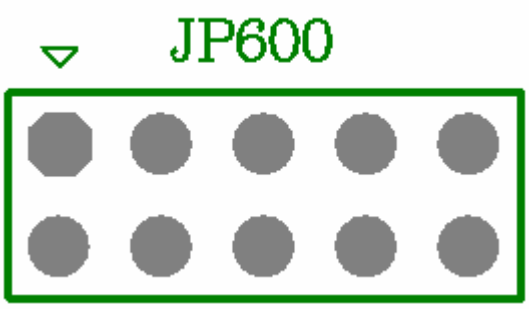
CAN2.0B 扩展帧

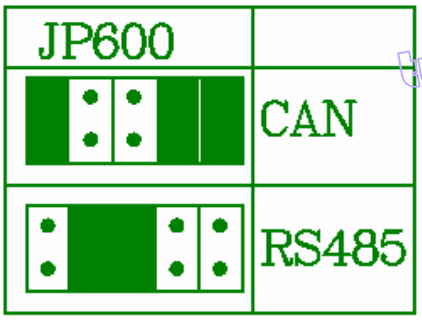
CAN 扩展帧信息为 13 个字节，包括两部分，信息和数据部分。前 5 个字节为信息部分。

	7	6	5	4	3	2	1	0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码）				ID.28-ID.21			
字节 3	ID.20-ID.13							
字节 4	ID.12-ID.5							
字节 5	ID.4-ID.0					X	X	X
字节 6	数据 1							
字节 7	数据 2							
字节 8	数据 3							
字节 9	数据 4							
字节 10	数据 5							
字节 11	数据 6							
字节 12	数据 7							
字节 13	数据 8							

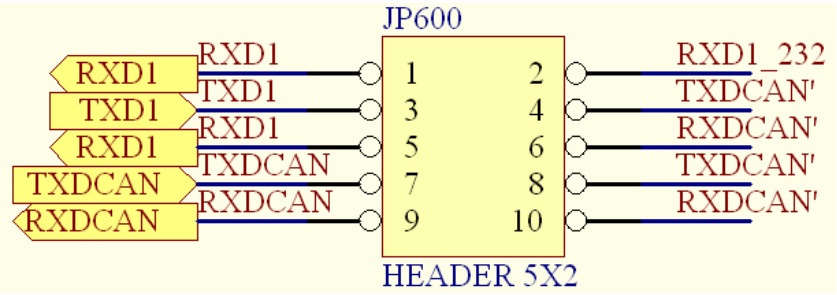
BHS-STM32 实验 29-CAN通讯(直接操作寄存器)

跳线设置



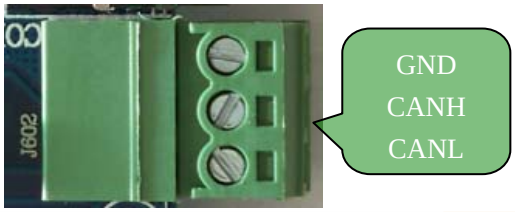


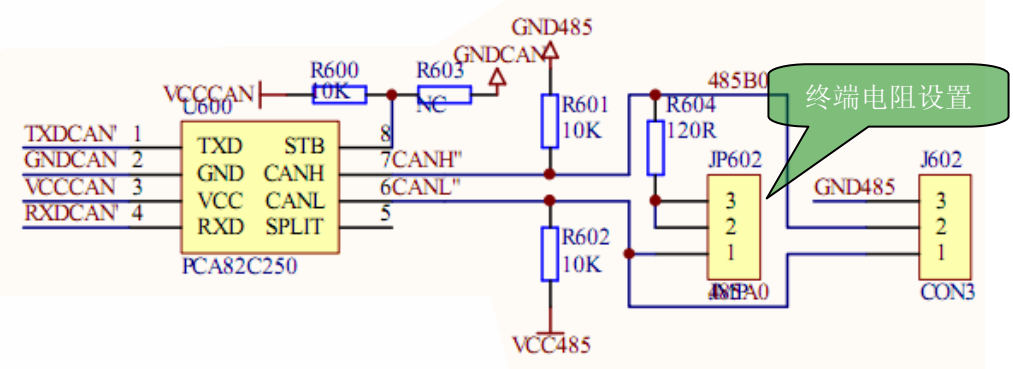
lyw藏书



注意：

1. 使用 RS485，请将 JP600 的 1,2　7,8　9,10 断开，3,4　5,6 短接
2. 使用 RS232，请将 JP600 的 3,4　5,6 断开，1,2 短接
3. 使用 CAN，请将 JP600 的 1,2　3,4　5,6 断开，7,8　9,10 短接





CAN 实验需要两个带 CAN 接口的板子

CAN 实验太复杂了，所以不是简单的配置文件能搞定的事，建议熟悉了 STM32 后再来做此实验。

准备工作，准备两个 CAN 板子，连接两个 CAN 接口，两个板子都下载本程序。

本实验通过 CAN 总线控制另外一个板子的 LED2~LED5 闪烁的例子，如果通信正常可以看到两个板子的 LED 都再闪烁，如果通信失败或者断开通信线，LED 不再闪烁

下面是详细代码

```
// CAN messge for sending
//CAN 发送消息邮箱
CAN_msg        CAN_TxMsg;
// CAN message for receiving
//CAN 接收消息邮箱
```

```
CAN_msg         CAN_RxMsg;
// CAN HW ready to transmit a message
//发送就绪标志
unsigned int    CAN_TxRdy = 0;
// CAN HW received a message
//接收就绪标志
unsigned int    CAN_RxRdy = 0;

/*----------------------------------------------------------------------------
   setup CAN interface
 *----------------------------------------------------------------------------*/
void CAN_setup (void)   {
   unsigned int brp = stm32_GetPCLK1();

   RCC->APB1ENR |= RCC_APB1ENR_CANEN;                    // enable clock for CAN

   // Note: uses PB8 and PB9 for CAN
   // enable clock for Alternate Function
   //启用复用功能的时钟
   RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;
   // reset CAN remap
   //复位 CAN 重新映射
   AFIO->MAPR     &= 0xFFFF9FFF;
   //     set CAN remap, use PB8, PB9
   //设置 CAN 重新映射，使用 PB8，PB9
   AFIO->MAPR     |= 0x00004000;
     // enable clock for GPIO B
     //使能 GPIOB 使用的 RCC 时钟
   RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;
   // CAN RX pin PB.8 input push pull
   //PB.8 推挽输入
   GPIOB->CRH &= ~(0x0F<<0);
   GPIOB->CRH |=   (0x08<<0);
   // CAN TX pin PB.9 alternate output push pull
   //PB.9 复用推挽输出
   GPIOB->CRH &= ~(0x0F<<4);
   GPIOB->CRH |=   (0x0B<<4);
   // enable interrupt
   //发送中断使能
   NVIC->ISER[0] |= (1 << (USB_HP_CAN_TX_IRQChannel    & 0x1F));
   // enable interrupt
   //接收中断使能
   NVIC->ISER[0] |= (1 << (USB_LP_CAN_RX0_IRQChannel & 0x1F));
   //初始化模式，  禁止报文自动重传
   CAN->MCR  =  (CAN_MCR_NART | CAN_MCR_INRQ);           // init mode, disable auto.
```

```
retransmission
   // Note: only FIFO 0, transmit mailbox 0 used
   // FIFO 0 msg pending, Transmit mbx empty
   //FIFO0 发生溢出的情况,FIFO 0 的 FOVR 位被置’1’时，产生中断。
   //发送邮箱 0 变为空，RQCPx 位被置’1’时，产生中断。
   CAN->IER = (CAN_IER_FMPIE0 | CAN_IER_TMEIE);

   /* Note: this calculations fit for PCLK1 = 36MHz */
   //设置波特率
   brp    = (brp / 18) / 500000;                          // baudrate is set to 500k bit/s

   /* set BTR register so that sample point is at about 72% bit time from bit start */
   /* TSEG1 = 12, TSEG2 = 5, SJW = 4 => 1 CAN bit = 18 TQ, sample at 72%       */
   CAN->BTR &= ~(((        0x03) << 24) | ((        0x07) << 20) | ((        0x0F) <<
16) | (        0x1FF));
   CAN->BTR |=    ((((4-1) & 0x03) << 24) | (((5-1) & 0x07) << 20) | (((12-1) & 0x0F) << 16) |
((brp-1) & 0x1FF));
}


/*----------------------------------------------------------------------------
   leave initialisation mode
 *----------------------------------------------------------------------------*/
void CAN_start (void)    {

   // normal operating mode, reset INRQ
   //使 CAN 从初始化模式进入正常工作模式
   CAN->MCR &= ~CAN_MCR_INRQ;
   while (CAN->MSR & CAN_MCR_INRQ);

}

/*----------------------------------------------------------------------------
   set the testmode
 *----------------------------------------------------------------------------*/
void CAN_testmode (unsigned int testmode) {

   // set testmode
   //设置 CAN 工作模式
   CAN->BTR &= ~(CAN_BTR_SILM | CAN_BTR_LBKM);
   CAN->BTR |=    (testmode & (CAN_BTR_SILM | CAN_BTR_LBKM));
}

/*----------------------------------------------------------------------------
   check if transmit mailbox is empty
```

```
 *---------------------------------------------------------------------------*/
void CAN_waitReady (void)  {

  // Transmit mailbox 0 is empty
  //发送邮箱空？
  while ((CAN->TSR & CAN_TSR_TME0) == 0);
  CAN_TxRdy = 1;

}

/*----------------------------------------------------------------------------
  wite a message to CAN peripheral and transmit it
 *---------------------------------------------------------------------------*/
void CAN_wrMsg (CAN_msg *msg)  {

  // Reset TIR register
  //发送邮箱标识符寄存器复位
  CAN->sTxMailBox[0].TIR  = (unsigned int)0;
  // Setup identifier information
  //如果是标准帧，标准帧是 11 位 ID(报文识别码)
  if (msg->format == STANDARD_FORMAT)
  {               //     Standard ID
    CAN->sTxMailBox[0].TIR |= (unsigned int)(msg->id << 21) | CAN_ID_STD;
  }
  else//如果是扩展帧，扩展帧是 29 位 ID(报文识别码)
  {                                                 // Extended ID
    CAN->sTxMailBox[0].TIR |= (unsigned int)(msg->id <<  3) | CAN_ID_EXT;
  }
  // Setup type information
  //如果消息为数据帧
  if (msg->type == DATA_FRAME)
  {                       // DATA FRAME
    CAN->sTxMailBox[0].TIR |= CAN_RTR_DATA;
  }
  else//数据为远程帧
  {                                                 // REMOTE FRAME
    CAN->sTxMailBox[0].TIR |= CAN_RTR_REMOTE;
  }
  //发送邮箱填充数据                                                  // Setup data bytes
  CAN->sTxMailBox[0].TDLR = (((unsigned int)msg->data[3] << 24) |
                             ((unsigned int)msg->data[2] << 16) |
                             ((unsigned int)msg->data[1] <<  8) |
                             ((unsigned int)msg->data[0])       );
  CAN->sTxMailBox[0].TDHR = (((unsigned int)msg->data[7] << 24) |
                             ((unsigned int)msg->data[6] << 16) |
```

```
                                    ((unsigned int)msg->data[5] <<    8) |
                                    ((unsigned int)msg->data[4])            );
  // Setup length
  //设置消息数据长度
  CAN->sTxMailBox[0].TDTR &= ~CAN_TDTxR_DLC;
  CAN->sTxMailBox[0].TDTR |=    (msg->len & CAN_TDTxR_DLC);
  // enable    TME interrupt
  //发送邮箱空中断使能
  CAN->IER |= CAN_IER_TMEIE;
  // transmit message
  //发送消息
  CAN->sTxMailBox[0].TIR |=    CAN_TIxR_TXRQ;
}

/*----------------------------------------------------------------------------
  read a message from CAN peripheral and release it
 *----------------------------------------------------------------------------*/
void CAN_rdMsg (CAN_msg *msg)    {
  // Read identifier information
  //标识信息
  //如果是标准帧，标准帧是 11 位 ID(报文识别码)
  if ((CAN->sFIFOMailBox[0].RIR & CAN_ID_EXT) == 0)
  { // Standard ID
    msg->format = STANDARD_FORMAT;
    msg->id     = (u32)0x000007FF & (CAN->sFIFOMailBox[0].RIR >> 21);
  }
  else//如果是扩展帧，扩展帧是 29 位 ID(报文识别码)
  {                                                    // Extended ID
    msg->format = EXTENDED_FORMAT;
    msg->id     = (u32)0x0003FFFF & (CAN->sFIFOMailBox[0].RIR >> 3);
  }
  // Read type information
  //如果消息为数据帧
  if ((CAN->sFIFOMailBox[0].RIR & CAN_RTR_REMOTE) == 0)
  {
    msg->type =    DATA_FRAME;                          // DATA    FRAME
  }
  else//数据为远程帧
  {
    msg->type = REMOTE_FRAME;                           // REMOTE FRAME
  }
  // Read length (number of received bytes)
  //消息数据长度
  msg->len = (unsigned char)0x0000000F & CAN->sFIFOMailBox[0].RDTR;
  // Read data bytes
```

```
   //从接收邮箱读数据
   msg->data[0] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDLR);
   msg->data[1] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDLR >> 8);
   msg->data[2] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDLR >> 16);
   msg->data[3] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDLR >> 24);

   msg->data[4] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDHR);
   msg->data[5] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDHR >> 8);
   msg->data[6] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDHR >> 16);
   msg->data[7] = (unsigned int)0x000000FF & (CAN->sFIFOMailBox[0].RDHR >> 24);

   // Release FIFO 0 output mailbox
   //释放接收 FIFO  0 输出邮箱，由于 FIFO 的特点，软件需要释放输出邮箱才能访问第 2 个报文
   CAN->RF0R |= CAN_RF0R_RFOM0;
}


void CAN_wrFilter (unsigned int id, unsigned char format)    {
   static unsigned short CAN_filterIdx = 0;
            unsigned int      CAN_msgId         = 0;
   // check if Filter Memory is full
   //检查 CAN 过滤器是否已满
   if (CAN_filterIdx > 13)
   {
      return;
   }
   // Setup identifier information
   //如果是标准帧，标准帧是 11 位 ID(报文识别码)
   if (format == STANDARD_FORMAT)
   {                        // Standard ID
        CAN_msgId    |= (unsigned int)(id << 21) | CAN_ID_STD;
   }
   else//如果是扩展帧，扩展帧是 29 位 ID(报文识别码)
   {                                                    // Extended ID
        CAN_msgId    |= (unsigned int)(id <<    3) | CAN_ID_EXT;
   }
   // set Initialisation mode for filter banks
   //CAN 过滤器组工作在初始化模式
   CAN->FMR    |=    CAN_FMR_FINIT;
   // deactivate filter
   CAN->FA1R &=    ~(unsigned int)(1 << CAN_filterIdx);

   // initialize filter
   // set 32-bit scale configuration
```

```
   //初始化过滤器
   //过滤器位宽为单个 32 位
   CAN->FS1R |= (unsigned int)(1 << CAN_filterIdx);
   // set 2 32-bit identifier list mode
   //过滤器组 x 的 2 个 32 位寄存器工作在标识符列表模式
   CAN->FM1R |= (unsigned int)(1 << CAN_filterIdx);
   //    32-bit identifier
   //32 位标识符
   CAN->sFilterRegister[CAN_filterIdx].FR1 = CAN_msgId;
   //    32-bit identifier
   //32 位标识符
   CAN->sFilterRegister[CAN_filterIdx].FR2 = CAN_msgId;
   // assign filter to FIFO 0
   //过滤器被关联到 FIFO0
   CAN->FFA1R &= ~(unsigned int)(1 << CAN_filterIdx);
   // activate filter
   //过滤器被激活
   CAN->FA1R    |=    (unsigned int)(1 << CAN_filterIdx);
   // reset Initialisation mode for filter banks
   //CAN 过滤器组工作在正常模式
   CAN->FMR &= ~CAN_FMR_FINIT;
   // increase filter index
   //设置的过滤器数量增加
   CAN_filterIdx += 1;
}

/*---------------------------------------------------------------------------
   CAN transmit interrupt handler
   CAN 发送中断
  *---------------------------------------------------------------------------*/
void USB_HP_CAN_TX_IRQHandler (void) {
   // request completed mbx 0
   //邮箱 0 请求完成
   if (CAN->TSR & CAN_TSR_RQCP0)
   {
     // reset request complete mbx 0
     //复位邮箱 0 请求
      CAN->TSR |= CAN_TSR_RQCP0;
      // disable    TME interrupt
      //禁止发送邮箱空中断
      CAN->IER &= ~CAN_IER_TMEIE;

        CAN_TxRdy = 1;
   }
}
```

tyw藏书

```
/*---------------------------------------------------------------------------
   CAN receive interrupt handler
   CAN 接收中断
 *---------------------------------------------------------------------------*/
void USB_LP_CAN_RX0_IRQHandler (void) {
   // message pending ?
   //接收到 CAN 报文，通过判断报文数目判断是否有报文
   if (CAN->RF0R & CAN_RF0R_FMP0)
   {
        // read the message
        //接收报文
        CAN_rdMsg (&CAN_RxMsg);

      CAN_RxRdy = 1;// set receive flag
   }
}

/*---------------------------------------------------------------------------
   initialize CAN interface
 *---------------------------------------------------------------------------*/
void can_Init (void) {

   CAN_setup ();                                    // setup CAN interface
   CAN_wrFilter (33, STANDARD_FORMAT);              // Enable reception of messages

   /* COMMENT THE LINE BELOW TO ENABLE DEVICE TO PARTICIPATE IN CAN NETWORK     */
   //CAN_testmode(CAN_BTR_SILM | CAN_BTR_LBKM);          // Loopback, Silent Mode (self-test)

   //软件仿真使用环回模式
    //CAN_testmode(CAN_BTR_SILM | CAN_BTR_LBKM); /* Loopback and                */
    //硬件仿真使用正常模式
    CAN_testmode(0); //正常模式
    // leave init mode
    //进入正常模式
    CAN_start ();
    // wait til mbx is empty
    //等待 CAN 就绪
    CAN_waitReady ();
}
```

```
/*---------------------------------------------------------------------------
   MAIN function
 *---------------------------------------------------------------------------*/
int main (void)
{
  uint32 i;
  //uint8 flag=0;

   stm32_Init ();
   can_Init ();                                          // initialise CAN interface


   CAN_TxMsg.id = 33;                                    // initialise message to send
   for (i = 0; i < 8; i++) CAN_TxMsg.data[i] = 0;
   CAN_TxMsg.len = 4;
   CAN_TxMsg.format = STANDARD_FORMAT;//使用标准帧
   CAN_TxMsg.type = DATA_FRAME;//数据帧

   i=0;
   while (1)
   {
      if (CAN_TxRdy)
      {
          if(++i==60000)
          {
              //i=0;
              //下面是数据报文
              CAN_TxMsg.data[0] = 0;
              CAN_TxMsg.data[1] = 0;
              CAN_TxMsg.data[2] = 0;
              CAN_TxMsg.data[3] = 0;

              //发送 CAN 报文
              CAN_TxRdy = 0;
              CAN_wrMsg (&CAN_TxMsg);
          }
          else if(i==120000)
          {
              i=0;
              //下面是数据报文
              CAN_TxMsg.data[0] = 1;
              CAN_TxMsg.data[1] = 1;
              CAN_TxMsg.data[2] = 1;
              CAN_TxMsg.data[3] = 1;
```

```
            //发送 CAN 报文
            CAN_TxRdy = 0;
            CAN_wrMsg (&CAN_TxMsg);
        }

    }


    if (CAN_RxRdy)
    {
        CAN_RxRdy = 0;

        if(CAN_RxMsg.data[0]==0)////PC8 状态
              PC8=0;
        else
            PC8=1;

        if(CAN_RxMsg.data[1]==0)////PC9 状态
            PC9=0;
        else
            PC9=1;

        if(CAN_RxMsg.data[2]==0)////PC10 状态
            PC10=0;
        else
            PC10=1;

        if(CAN_RxMsg.data[3]==0)////PC11 状态
            PC11=0;
        else
            PC11=1;

    }

  } // end while

} // end main
```

BHS-STM32 实验 30-CAN通讯(库函数)

CAN 实验需要两个带 CAN 接口的板子
CAN 实验太复杂了，所以不是简单的配置文件能搞定的事，建议熟悉了 STM32 后再来做此实验。
准备工作，准备两个 CAN 板子，连接两个 CAN 接口，两个板子都下载本程序。
本实验通过 CAN 总线控制另外一个板子的 LED2~LED5 闪烁的例子，如果通信正常可以看到两个板子的 LED 都再闪烁，如果通信失败或者断开通信线，LED 不再闪烁
由于考虑效率问题，所以 CAN 通信代码只有初始化使用库函数，其他仍然是直接操作寄存

器，下面是与上例不同的部分

tyw藏书

```
void CAN_setup(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    CAN_InitTypeDef CAN_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    unsigned int brp = stm32_GetPCLK1();


    //配置 CAN 使用的时钟
    /* CAN Periph clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN, ENABLE);
    // enable clock for Alternate Function
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB,
ENABLE);


    //CAN 功能重新映射
    //  AFIO->MAPR &= 0xFFFF9FFF;                    // reset CAN remap
    //  AFIO->MAPR |= 0x00004000;                    //  set CAN remap, use PB8,
PB9
    GPIO_PinRemapConfig(GPIO_Remap1_CAN, ENABLE);

    //配置 CAN 使用的 IO 口
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    /* Configure CAN pin: RX */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* Configure CAN pin: TX */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    //  NVIC->ISER[0] |= (1 << (USB_HP_CAN_TX_IRQChannel  & 0x1F));// enable interrupt
    //  NVIC->ISER[0] |= (1 << (USB_LP_CAN_RX0_IRQChannel & 0x1F));// enable interrupt
    /* Enable CAN RX0 interrupt IRQ channel */
    //配置 CAN 收发中断
    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN_RX0_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
```

```
    NVIC_InitStructure.NVIC_IRQChannel = USB_HP_CAN_TX_IRQChannel;
    NVIC_Init(&NVIC_InitStructure);

    //  CAN->MCR = (CAN_MCR_NART | CAN_MCR_INRQ);          // init mode, disable auto. retransmission
    // Note: only FIFO 0, transmit mailbox 0 used
    //*

    //  CAN_DeInit();
    //配置 CAN 工作模式，波特率
    CAN_StructInit(&CAN_InitStructure);

    // CAN cell init
    //初始化 CAN
    CAN_InitStructure.CAN_TTCM = DISABLE;//禁止时间触发通讯模式
    CAN_InitStructure.CAN_ABOM = DISABLE;//禁止自动离线管理
    CAN_InitStructure.CAN_AWUM = DISABLE;//禁止自动唤醒模式
    CAN_InitStructure.CAN_NART = ENABLE;//允许非自动重传输模式
    CAN_InitStructure.CAN_RFLM = DISABLE;//禁止接收  FIFO 锁定模式
    CAN_InitStructure.CAN_TXFP = DISABLE;//禁止发送  FIFO 优先级
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;//CAN 工作在正常模式
    CAN_InitStructure.CAN_SJW = CAN_SJW_4tq;//重新同步跳跃宽度  4  个时间单位
    CAN_InitStructure.CAN_BS1 = CAN_BS1_12tq;//时间段 1  为 12  个时间单位
    CAN_InitStructure.CAN_BS2 = CAN_BS2_5tq;//时间段 2  为 5  个时间单位
    CAN_InitStructure.CAN_Prescaler = (brp / 18) / 500000; //设置波特率
    CAN_Init(&CAN_InitStructure);
    //*/

    // Note: only FIFO 0, transmit mailbox 0 used
    // FIFO 0 msg pending, Transmit mbx empty
    //FIFO0 发生溢出的情况,FIFO 0 的 FOVR 位被置’1’时，产生中断。
    //发送邮箱 0 变为空，RQCPx 位被置’1’时，产生中断。
    CAN_ITConfig(CAN_IT_FMP0 | CAN_IT_TME, ENABLE);

}
```

后面的例子可能是直接操作寄存器，也可能使用库函数，也可能两者都有

中级例程-(应用篇)

初级例程是熟悉 STM32 芯片硬件的例程，中级例程是比较实用的例程，里面介绍了一些 SPI-FLASH, TFT,等外设驱动

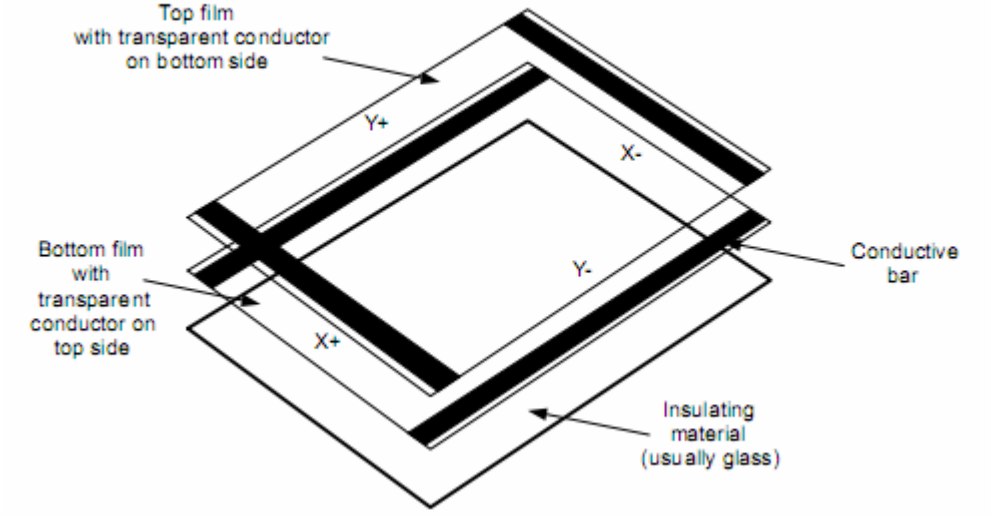
BHS-STM32 实验 31-3 点触摸校正

开发板使用的触摸屏是 4 线电阻屏。

四线触摸屏的结构如图 1 所示，由两个透明层构成，透明层的内表面均涂了薄薄一层导电材料。当触摸屏表面受到的压力（如通过触笔或手指进行按压）足够大时，顶层与底层之间会产生接触，从而使电阻层

发生接触。

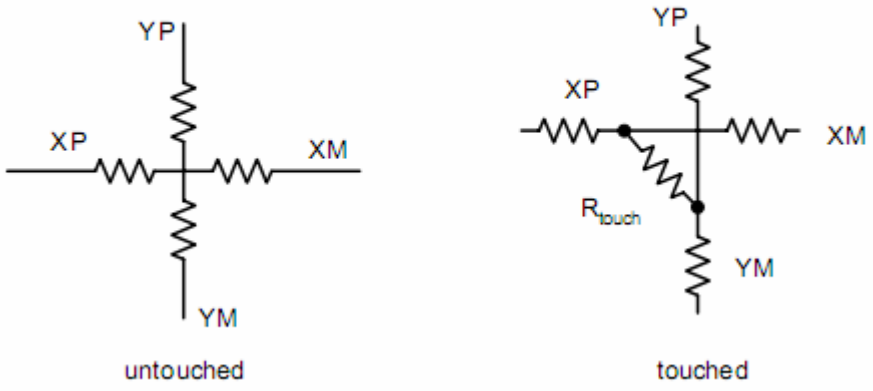
图 1. 四线触摸屏结构



第一种测量方法

触摸可以由三个参数界定。第一和第二参数分别是 X 触摸点位置和 Y 触摸点位置。第三参数即为“触摸压力”，可使触摸屏区分手指接触和触笔接触。图 2 说明的是被触摸和未被触摸的屏幕的等效电路。每个测量周期均采用不同的电压组合。在本项目中，触摸屏接 Vdd 是指切换端口驱动模式为 “strong” 并把逻辑值设为 “high”。接地是指将端口驱动模式设为 “strong” 并把逻辑值设为 “low”。输出级漏电阻较低，其感应也小。校准算法可以对这种差异进行补偿。

图 2. 触摸屏等效电路



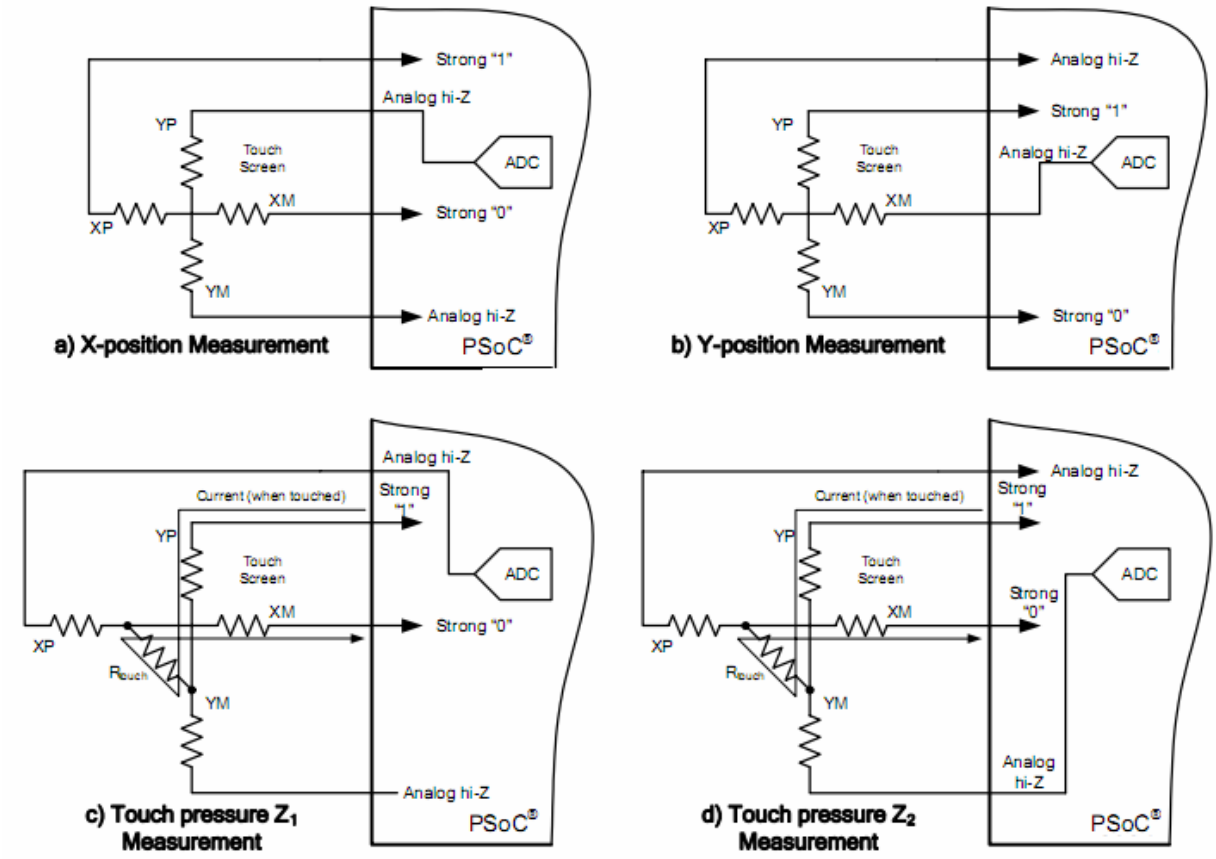
所有测量都可以通过可编程增益放大器 (PGA) 和增强型模数转换器 (ADC) 来进行。触摸屏与 PSoC® 使用同一电源。ADC 的配置是用于测量 GND-Vdd 的范围。因此，测量并不取决于电源的电压。

图 3 说明的是触摸屏测量周期的各种方式。通过将 XP 连接到 Vdd 且 XM 接地可以测定 X 触点在 X 轴平面的位置。从 YP 或 YM 触摸屏连接器上测得的电压与触点 X 坐标成比例。

通过将 YP 连接到 Vdd 且 YM 接地可以测定 Y 触点在 Y 轴平面的位置。从 YP 或 YM 触摸屏连接器上测得的电压与触点 Y 坐标成比例。

图 3. 各种参数测量

tyw藏书



触摸屏校准

在很多情况下，触摸屏安装在 LCD 或另一个显示器上。在此情况下，触摸屏测量的数据必须转换为真实的屏幕坐标点。本手册中提出的校准算法可以消除缩放比例因素以及触摸屏的机械不同轴性 (mechanical misalignment)。

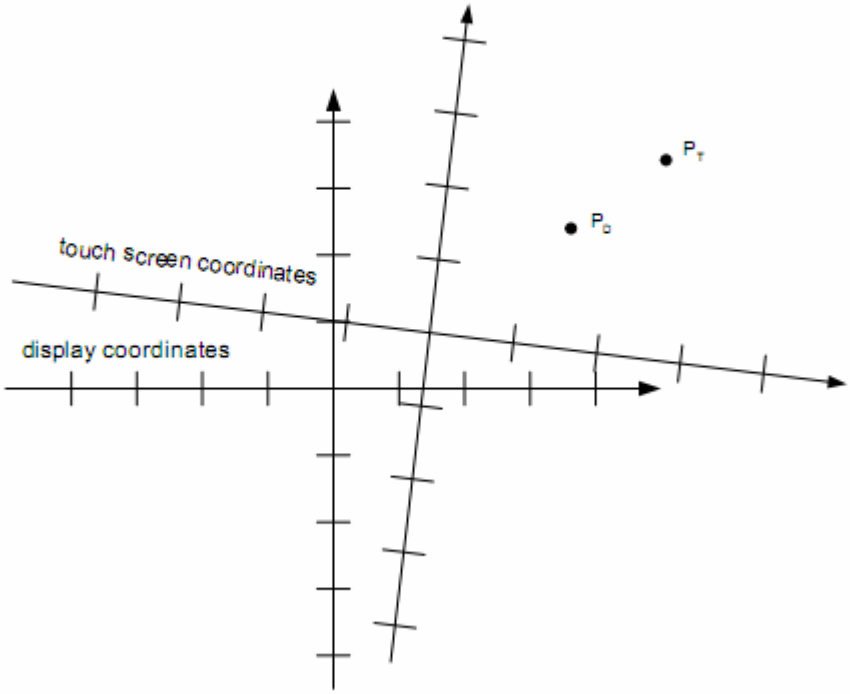
校准算法面临的挑战是必须将触摸屏测量出来的坐标系转换成准确描绘显示器上某一点的坐标系。

$$[X_D, Y_D] = \overline{f}([X_T, Y_T])$$

图 6 表示的是某些未对准触摸屏和显示器的坐标。该图也表明，可以将显示器的每个点表示为 PD = [XD,YD]，并可以将触摸屏上的每个点表示为 PT = [XT,YT]。

图 6. 显示器与触摸屏未对准。显示器与触摸屏坐标

tyw藏书



但可能会有如下三个因素导致结果错误：

- 相对于显示器坐标，触摸屏坐标发生旋转。
- 坐标的线性移动。
- 缩放比例因素。

下列表达式考虑了所有这三个因素：

$$X_D = A(X_T) + B(T_T) + C$$
$$Y_D = D(X_T) + E(T_T) + F$$

Equation 7

这些关系式提出了六个校准系数。因此需要三个取样点来求得线性方程组 (8)的解，并得到校准系数的值。

$$X_{D0} = A(X_{T0}) + B(T_{T0}) + C$$
$$X_{D1} = A(X_{T1}) + B(T_{T1}) + C$$
$$X_{D2} = A(X_{T2}) + B(T_{T2}) + C$$
$$Y_{D0} = D(X_{T0}) + E(T_{T0}) + F$$
$$Y_{D1} = D(X_{T1}) + E(T_{T1}) + F$$
$$Y_{D2} = D(X_{T2}) + E(T_{T2}) + F$$

Equation 8

未知数可以在方程式 (9) 中求解：

lyw藏书

$$A=\frac{(X_{D0}-X_{D2})(Y_{T1}-Y_{T2})-(X_{D1}-X_{D2})(Y_{T0}-Y_{T2})}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})}$$

$$B=\frac{(X_{T0}-X_{T2})(Y_{D1}-Y_{D2})-(X_{D1}-X_{D2})(Y_{T0}-Y_{T2})}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})}$$

$$C=\frac{Y_{T0}[X_{T2}(X_{D1})-X_{T1}(X_{D2})]+Y_{T1}[X_{T0}(X_{D2})-X_{T2}(X_{D0})]+Y_{T2}[X_{T1}(X_{D0})-X_{T0}(X_{D1})]}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2}(Y_{T0}-Y_{T2})}$$

$$D=\frac{(Y_{D0}-Y_{D2})(Y_{T1}-Y_{T2})-(Y_{D1}-Y_{D2})(Y_{T0}-Y_{T2})}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})}$$

$$E=\frac{(X_{T0}-X_{T2})(Y_{D1}-Y_{D2})-(Y_{D0}-Y_{D2})(X_{T1}-X_{T2})}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})}$$

$$F=\frac{Y_{T0}[X_{T2}(Y_{D1})-X_{T1}(Y_{D2})]+Y_{T1}[X_{T0}(Y_{D2})-X_{T2}(Y_{D0})]+Y_{T2}[X_{T1}(Y_{D0})-X_{T0}(Y_{D1})]}{(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})}$$

Equation 9

所有表达式的右边包含同样的分母。这对整数算法非常有用。假设分母为 K，即可得到方程式 (10) 。将 K 代入方程式 (8)，即可得到方程式 (11) 。

方程式 (8) 所要求的所有运算都可以整数计算进行。

$$K=(X_{T0}-X_{T2})(Y_{T1}-Y_{T2})-(X_{T1}-X_{T2})(Y_{T0}-Y_{T2})$$

$$A'=(X_{D0}-X_{D2})(Y_{T1}-Y_{T2})-(X_{D1}-X_{D2})(Y_{T0}-Y_{T2})$$

$$B'=(X_{T0}-X_{T2})(X_{D1}-X_{D2})-(X_{D0}-X_{D2})(X_{T1}-X_{T2})$$

$$C'=Y_{T0}\big((X_{T2})(X_{D1})-(X_{T1})(X_{D2})\big)+Y_{T1}\big((X_{T0})(X_{D2})-(X_{T2})(X_{D0})\big)+Y_{T2}\big((X_{T1})(X_{D0})-(X_{T0})(X_{D1})\big)$$

$$D'=(Y_{D0}-Y_{D2})(Y_{T1}-Y_{T2})-(Y_{D1}-Y_{D2})(Y_{T0}-Y_{T2})$$

$$E'=(X_{T0}-X_{T2})(Y_{D1}-Y_{D2})-(Y_{D0}-Y_{D2})(X_{T1}-X_{T2})$$

$$F'=Y_{T0}\big((X_{T2})(Y_{D1})-(X_{T1})(Y_{D2})\big)+Y_{T1}\big((X_{T0})(Y_{D2})-(X_{T2})(Y_{D0})\big)+Y_{T2}\big((X_{T1})(Y_{D0})-(X_{T0})(Y_{D1})\big)$$

Equation 10

$$X_D=\frac{A'(X_T)+B'(T_T)+C'}{K}$$

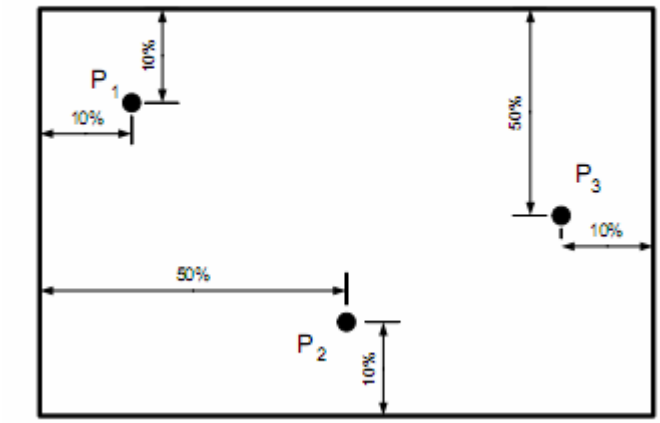
$$Y_D=\frac{D'(Y_T)+E'(T_T)+F'}{K}$$

Equation 11

为得到最佳结果，第一个取样点必须位于距离屏幕左上方大约 10% 处。第 2、第 3 点分别位于距离中心和边缘大约 10% 处（参见图 7）。在校准过程中，点 1、2、3 必须按升序触摸。

在使用前，必须对每个含有触摸屏的器件进行校准。没有必要每次在器件上电后都进行校准。但是，为谨慎起见，在每个独立内存中都应保存校准参数。在每次校准后器件被使用时，器件都会读取这些系数并用于计算真实屏幕坐标。

图 7. 屏幕上校准点的布置



本例子使用 3 点法校正触摸屏，一般情况下触摸屏是非线性的，都需要校正才能使用。
当屏上显示红点时，用手指触摸该点，直到该点变为绿色抬起手指，屏上依次出现 3 个点，3 个点都变绿时校正完成，这时用手触摸屏任意位置，观察光标是否跟手指位置对应

说明：
1.触摸是不可太用力，否则触摸屏易碎
2.为了简化程序，例程不一定有中文提示了。只要出现红点就开始触摸校正。
下面是部分程序具体实现
//定义采样参数 ReadLoop 采集次数， LOSS_DATA 丢掉最大值，最小值个数

#define ReadLoop 13 //连续采集次数必须>3

```
#define LOSS_DATA 5 //丢掉最大最小数据个数
/*
功能：采集 TP 坐标数据

输出：
  xValue X 坐标采集到的值,  yValue Y 坐标采集到的值
返回：
  0=无效数据，1=有效数据

说明：采集 13 次，然后排序，保留中间 3 次取平均值
*/
u8 Read_XY(u16 *xValue, u16 *yValue)
{
 u16 i, j;
 u16 bufX[ReadLoop];
 u16 bufY[ReadLoop];
 uint32 sumX;
 uint32 sumY;
 u16 val;

    //连续采集 X 坐标，Y 坐标数据
    for(i=0; i<ReadLoop; i++)
    {
        //如果采集还没完，触摸已经抬起，那么放弃该次采集
        if(TP_IRQ)
            return(0);

        //while(TP_BUSY);
        //Delayus(5);
        bufX[i]=RD_AD(CHY);

        //while(TP_BUSY);
        //Delayus(5);
        bufY[i]=RD_AD(CHX);

        Delayus(5);
    }


    //对采集数据排序
    for(i=0; i<ReadLoop-1; i++)
    {
        for(j=i+1; j<ReadLoop; j++)
        {
            if(bufX[i]>bufX[j])
```

tyw藏书

```
            {
                val=bufX[i];
                bufX[i]=bufX[j];
                bufX[j]=val;
            }

            if(bufY[i]>bufY[j])
            {
                val=bufY[i];
                bufY[i]=bufY[j];
                bufY[j]=val;
            }
        }
    }

    //丢掉最大最小数据后求和
    sumX=0;sumY=0;
    for(i=LOSS_DATA; i<ReadLoop-LOSS_DATA; i++)
    {
        sumX += bufX[i];
        sumY += bufY[i];
    }

    //取平均值
    *xValue=sumX/(ReadLoop-2*LOSS_DATA);
    *yValue=sumY/(ReadLoop-2*LOSS_DATA);

    return (1);
}
/*
功能：采集 TP 坐标数据
输出：
  x：x 坐标
  y：y 坐标
返回：
  0=无效数据，1=有效数据
*/
uint8 TP_GetAdXY(u16 *x, u16 *y)
{
 u8 flag;

    if(TP_IRQ)
        return(0);
    flag=Read_XY(x, y);
```

```
    if(flag==0)
        return(0);


    //一般情况，X,Y 都比 0 大，具体你根据实际情况定，小于一定范围的可以认为是无效数据
    if(*x<100 || *y<100)
        return(0);
    else
        return(1);
}
/*
功能：本函数连续采样 2 次，间隔时间由 delay 参数确定,2 次采样结果+-50 范围内才算有效
2 次是指连续采集到 2 个处理后的有效数据，本例实际是采集 2*13=26 次数据
输出：
  x：x 坐标
  y：y 坐标
输入：
  delay：采样间隔时间，单位 ms

返回：
  0=无效数据，1=有效数据
*/
uint8 TP_GetAdXY2(u16 *x, u16 *y, uint32 delay)
{u16 x1,y1;
 u16 x2,y2;
 u8 flag;
 //u16 adx,ady;


    flag=TP_GetAdXY(&x1, &y1);

    if(flag==0)
        return(0);

    //os_dly_wait(delay/OS_TIME);//300ms 后再采样 1 次
    Delay(delay);

    flag=TP_GetAdXY(&x2, &y2);

    if(flag==0)
        return(0);

    //两次采样结果必须保持在一定范围内，该数值根据你实际情况定
    if( ( (x2<=x1 && x1<x2+50) || (x1<=x2 && x2<x1+50) )//前后两次采样在+-50 内
     && ( (y2<=y1 && y1<y2+50) || (y1<=y2 && y2<y1+50) ) )
```

tyw藏书

```
    {
        *x=(x1+x2)/2;
        *y=(y1+y2)/2;
        return(1);
    }
    else
        return(0);
}
//校正参数
//如果不想每次开机都校正触摸屏那么就应该把该校正后的参数保存到非易失存储器里面，比如FLASH,EEPROM 等。
MATRIX    matrix=
{0x0000F900, 0xFFFFFDC0, 0xFFE714B0, 0x00000280, 0x00017700, 0xFFC219C0, 0x0002887A };
POINT     display;

void Dlg_Main(void)
{uint16 xt,yt;
// uint8 buf[32];
  uint16 i;
  uint8 flag;

    //填充色块
    FillSolidRect(0, 0, 240, 320, BLUE);
    //os_tsk_delete_self ();
    //显示校正点
    i=0;
    DrawIcon(LCDSample[i].x, LCDSample[i].y, RED, CURSOR_CROSS);
    do
    {
        //os_dly_wait(100/OS_TIME);//100ms
        Delay(100);
        flag=TP_GetAdXY2(&xt, &yt, 300);
        if(flag==0)//无触摸
            ;
        else
        {
            //如果有触摸按下，校正点变色
            DrawIcon(LCDSample[i].x, LCDSample[i].y, GREEN, CURSOR_CROSS);
            //保存读回的坐标数据
            TouchSample[i].x = xt/4;
            TouchSample[i].y = yt/4;

            //等待抬起
            //os_dly_wait(500/OS_TIME);//1000ms
                Delay(500);
```

```
            while( flag )
            {
                flag=TP_GetAdXY(&xt, &yt);
            }
            //继续显示下一个校正点
            i++;
            DrawIcon(LCDSample[i].x, LCDSample[i].y, RED, CURSOR_CROSS);
        }

    }while(i<3);
    //填充色块
    FillSolidRect(0, 0, 240, 320, BLUE);
    //设置校正后的参数
    setCalibrationMatrix( &LCDSample[0], &TouchSample[0], &matrix );
    //初始化光标
    InitCursor(CURSOR_ARROW, WHITE, BLUE);
    SetCursor(240/2-1, 320/2-1, 1);
    //下面测试校正后是否有误差，正常时手指点哪里，光标就显示哪里
    while(1)
    {

        flag=TP_GetAdXY2(&xt, &yt, 0);
        if(flag)
        {
            TouchSample[0].x=xt/4; TouchSample[0].y=yt/4;
            getDisplayPoint( &display, &TouchSample[0], &matrix );

            SetCursor(display.x, display.y, 1);
        }
    }
}
```

BHS-STM32 实验 32-SPI-Flash

本实验学习 SPI-FLASH 读、写、擦除，SPI-FLASH 一般容量大，价格相对较便宜，是存储字库，图片，数据不错的选择。一般 CPU 自带的 FLASH 容量有限，外扩 SPI 接口简单，占用资源少。对于并行 FLASH 来说价格贵，扩展还占不少 IO 资源。所以 SPI-FLASH 存储器比并行存储器在不少应用更有优势。
SPI-FLASH 一般寿命是 10 万次。

```
/*
功能：SPI-FLASH 读写测试
此函数对固定地址 0xff000 读出写入 8 个字节数据
在硬件仿真状态下观察 buf[]里面的数据，正常情况下能观察到有所变化的
*/
void TestSPI_Flash(void)
{
```

```
uint8 buf[8];
uint32 address;

    UnProtectSST25VF080();//解除保护才能对 FLASH 写

    address=0xff000;

    //为了保证数据正确写入，应该先擦除该扇区，
    Sector_Erase(address);

    //读取 SPI-FLSH 数据，address=地址，16=数据个数，buf=读取的数据存放缓冲
    Read_Flash_Page ( address, 8, buf );

    //写数据到 SPI-FLSH
    Write_Flash_Page(address, 8, "\xaa\xbb\xcc\xdd\x11\x22\x33\x44" );//写数据到 FLASH

    //读取 SPI-FLSH 数据，address=地址，16=数据个数，buf=读取的数据存放缓冲
    Read_Flash_Page ( address, 8, buf );

    //为了保证数据正确写入，应该先擦除该扇区
    Sector_Erase(address);

    //写数据到 SPI-FLSH
    Write_Flash_Page(address, 8, "\x12\x34\x56\x78\x90\xab\xcd\xef" );//写数据到 FLASH

    //读取 SPI-FLSH 数据，address=地址，16=数据个数，buf=读取的数据存放缓冲
    Read_Flash_Page ( address, 8, buf );
}
```

用仿真器单步仿真，观察 buf 变化

注意：很多人忽略了 SPI-FLASH 上电都是写保护的，必须解除写保护才能对 FLASH 执行擦除，编程操作。那么系统上掉调用解除写保护就非常重要的：UnProtectSST25VF080();//解除保护才能对 FLASH 写

BHS-STM32 实验 33-TFT测试+汉字显示

TFT 测试一般显示几种单色就可以判断是否正常，比如分别显示：红，绿，蓝

汉字字库相关知识

1 汉字机内码 国标码和区位码

在 PC 机的文本文件中 汉字是以机内码的形式存储的 每个汉字占用两个字节长度 为了和 ASCII 码区别 范围从十六进制的 0A1H 开始 小于 80H 的为 ASCII 码 将机内码每个字节的最高位屏蔽掉 再以十六进制的形式显示出来 则为国标码 将机内码的每个字节各减去 0A0H 再以十进制显示出来 即为该汉字的区位码 例如 国 字的机内码 国标码和区位码如表 1 所示

表 1 “国”字的机内码、国标码和区位码

类别	数值	高位字节								低位字节							
机内码	B9FAH	1	0	1	1	1	0	0	1	1	1	1	1	1	0	1	0
国标码	397AH	0	0	1	1	1	0	0	1	0	1	1	1	1	0	1	0
区位码	195AH	0	0	0	1	1	0	0	1	0	1	0	1	1	0	1	0

即区位码 机内码 0A0AOH 就 国 字而言 其区位码和机内码的关系为
195AH 区位码 0B9FAH 机内码 0A0A0H
记住这个关系 是我们理解下面有关程序的关键
2 国标汉字字符集与区位码
根据对汉字使用频率程度的研究 可把汉字分成高频字 约 100 个 常用字 约 3000 个 次常用字 约 4000 字 罕见字 约 8000 个 和死字 约 45000 个 即正常使用的汉字达 15000 个 我国 1981 年公布了 通讯用汉字字符集 基本集 及其交换码标准 GB2312-80 方案 把高频字 常用字和次常用字集合成汉字基本字符集 共 6763 个 在该字符集中按汉字使用的频度 又将其分为一级汉字 3755 个 按拼音排序 二级汉字 3008 个 按部首排序 再加上西文字母 数字 图形符号等 700 个 国家标准的汉广州周立功单片机发展有限公司 Tel: (020)38730976 38730977 Fax: 38730925 http://www.zlgmcu.com
- 2 -
字字符集 GB2312-80 在汉字操作系统中是以汉字库的形式提供的 汉字库结构作了统一规定 如图 1 所

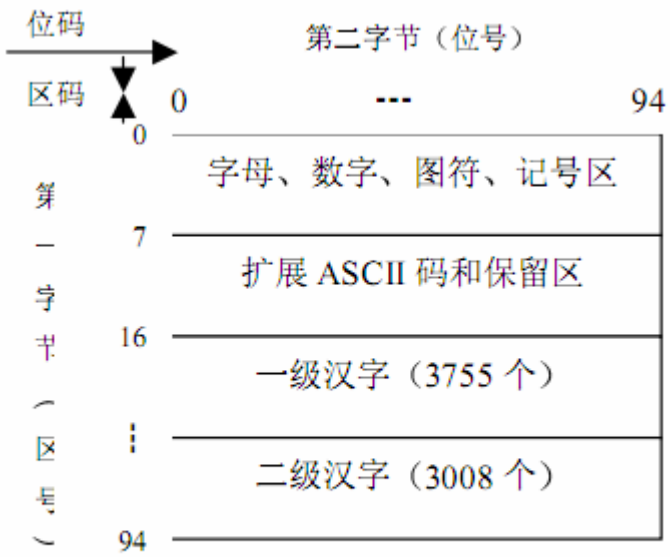
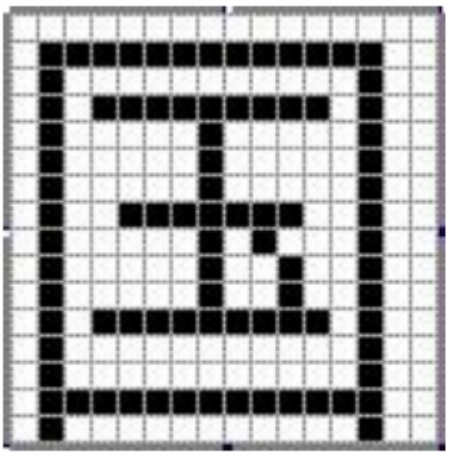


图 1 国标（GB2312-80）汉字字符集

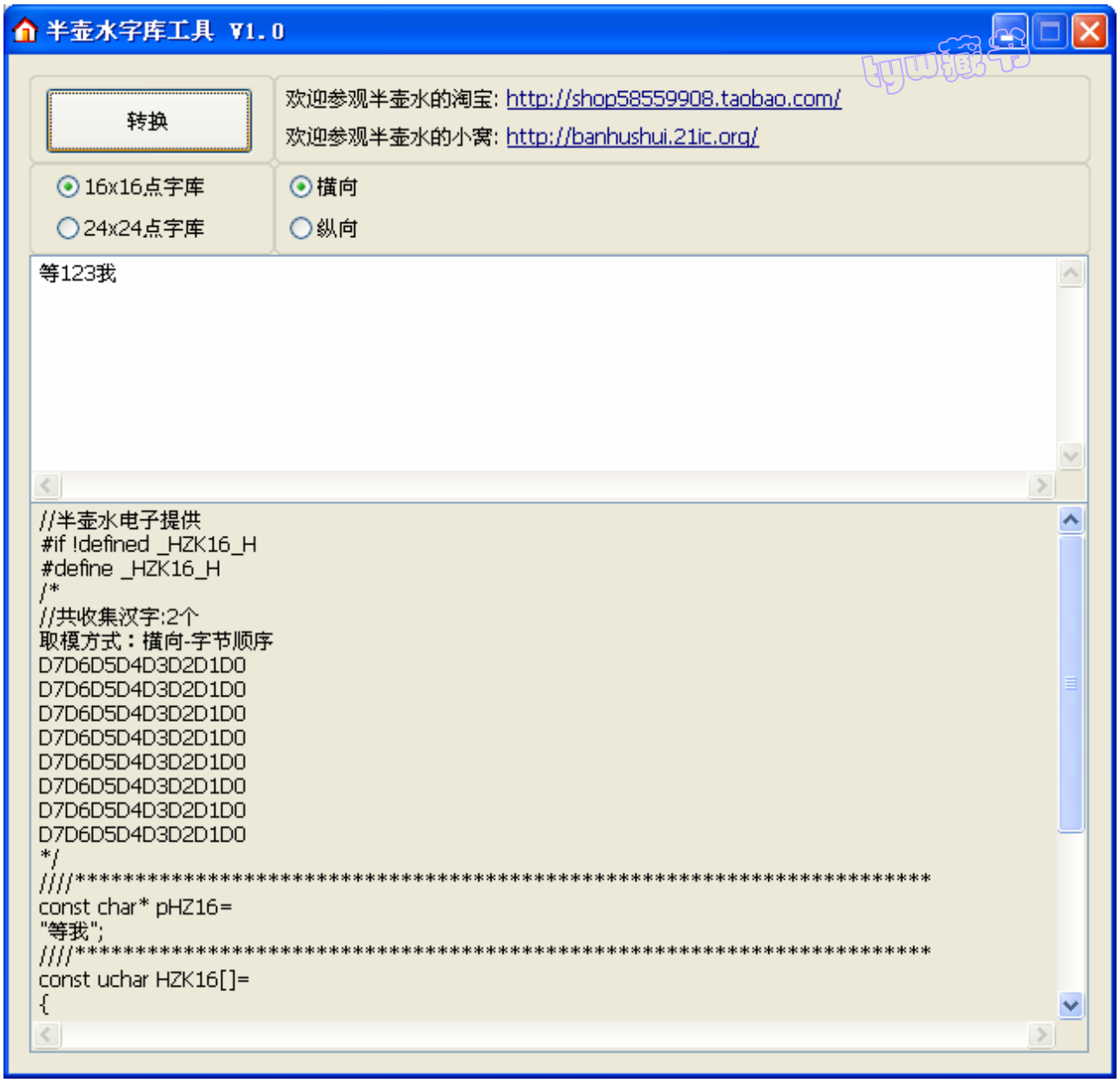
示
即将字库分成 94 个区 每个区有 94 个汉字 以位作区别 每一个汉字在汉字库中有确定的区和位编号 用两个字节 这就是所谓的区位码 区位码的第一个字节表示区号 第二个字节表示位号 因而只要知道了区位码 就可知道该汉字在字库中的地址 每个汉字在字库中是以点阵字模形式存储的 如一般采用 16 16 点阵形式 每个点用一个二进制位表示 存 1 的点 当显示时 可以在屏上显示一个亮点 存 0 的点 则在屏上不显示 这样就把存某字的 16 16 点阵信息直接用来在显示器上按上述原则显示 则将出现对应的汉字 如一个 国 字的 16 16 点阵字模如图 2 所示 当用存储单元存储该字模信息时 将需 32 个字节地址 在图 2 的右边写出了该字模对应的字节值



字节		字节	
0	00H	1	00H
2	7FH	3	FCH
4	40H	5	04H
6	5FH	7	F4H
8	41H	9	04H
10	41H	11	04H
12	41H	13	04H
14	4FH	15	E4H
16	41H	17	44H
18	41H	19	24H
20	41H	21	24H
22	5FH	23	F4H
24	40H	25	04H
26	40H	27	04H
28	7FH	29	FCH
30	40H	31	04H

图 2 “国”字的 16×16 点阵字模

上面是横向取模得出的数据
使用我提供的汉字字库工具可以方便得到想要的汉字字库



软件说明：

1.支持 16×16 点阵字库，24×24 点阵字库支持横向、纵向取模，一般彩色 TFT 使用横向方式

2.支持过滤功能，只搜索编辑框中的汉字，自动过滤重复的汉字

3.排序功能，对汉字排序，方便用 2 分法快速查找汉字位置

4.自动生成头文件，将文件考入自己的头文件就可以了

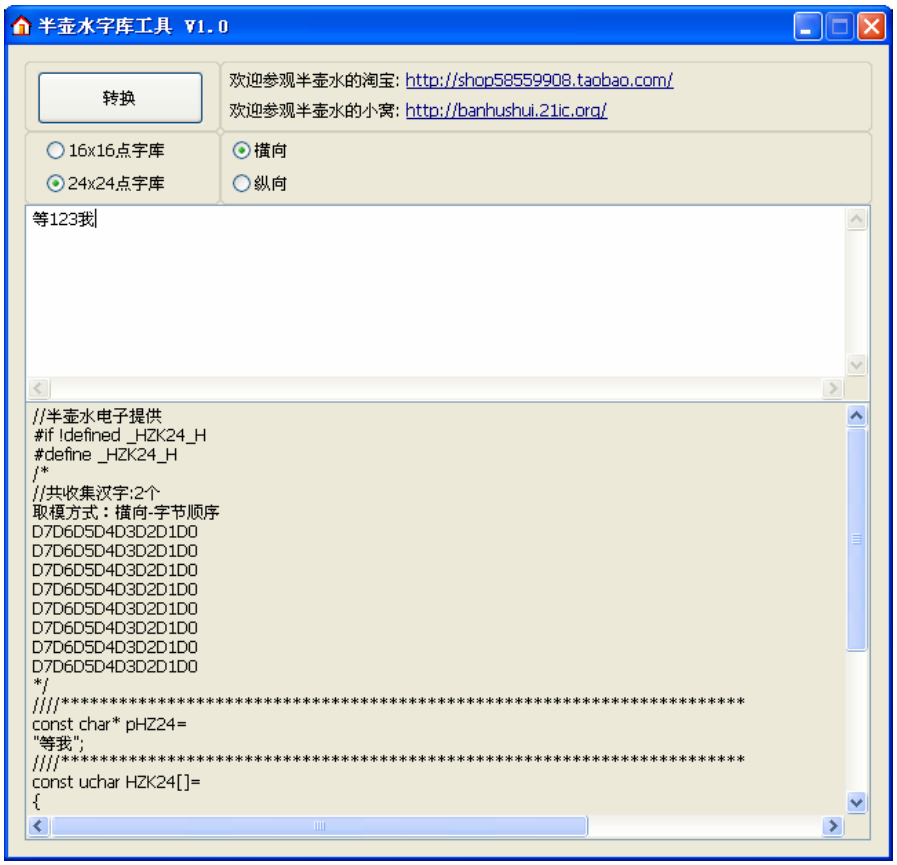
CTRL+A 全选，CTRL+C 复制

下面是输入：等 123 我得到的文件

```
//半壶水电子提供
#if !defined _HZK16_H
#define _HZK16_H
/*
//共收集汉字:2 个
取模方式：横向-字节顺序
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
```

D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
*/
////**
const char* pHZ16=
"等我";
////**
const uchar HZK16[]=
{
//等
0x10, 0x40, 0x10, 0x48, 0x3E, 0xFC, 0x49, 0x20, 0x01, 0x00, 0x3F, 0xF8, 0x01, 0x00, 0x01, 0x00,
0xFF, 0xFE, 0x00, 0x40, 0x7F, 0xFC, 0x08, 0x40, 0x04, 0x40, 0x04, 0x40, 0x01, 0x40, 0x00, 0x80,
//我
0x04, 0x80, 0x0E, 0xA0, 0x78, 0x90, 0x08, 0x90, 0x08, 0x84, 0xFF, 0xFE, 0x08, 0x80, 0x08, 0x90,
0x0A, 0x90, 0x0C, 0x60, 0x18, 0x40, 0x68, 0xA0, 0x09, 0x20, 0x0A, 0x14, 0x28, 0x14, 0x10, 0x0C
};
#endif

详细请看 hzk16.h 中的内容
下面是生成 24×24 点字库



说明：此方法适用与字库量少的应用，如果要显示很多汉字，建议将完整字库保存到 SPI-FLASH 中，BHS-GUI 例程就是这样做的。

将上面工具生成的字库文件保存为.h 文件加入到项目中，例子中 16X16 点阵保存为 hzk16.h

```
#if !defined _HZK16_H
#define _HZK16_H
/*
//共收集汉字:7 个
取模方式：横向-字节顺序
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
*/
////**************************************************************************
const char* pHZ16=
"备待等好设外准";
////**************************************************************************
const uchar HZK16[]=
{
//备
0x04, 0x20, 0x07, 0xF0, 0x08, 0x20, 0x14, 0x40, 0x23, 0x80, 0x02, 0x80, 0x0C, 0x60, 0x30, 0x18,
0xDF, 0xF6, 0x11, 0x10, 0x11, 0x10, 0x1F, 0xF0, 0x11, 0x10, 0x11, 0x10, 0x1F, 0xF0, 0x10, 0x10,
//待
0x08, 0x40, 0x08, 0x48, 0x13, 0xFC, 0x20, 0x40, 0x48, 0x40, 0x08, 0x44, 0x17, 0xFE, 0x30, 0x10,
0x50, 0x10, 0x97, 0xFE, 0x10, 0x10, 0x11, 0x10, 0x10, 0x90, 0x10, 0x90, 0x10, 0x10, 0x10, 0x60,
//等
0x10, 0x40, 0x10, 0x48, 0x3E, 0xFC, 0x49, 0x20, 0x01, 0x00, 0x3F, 0xF8, 0x01, 0x00, 0x01, 0x00,
0xFF, 0xFE, 0x00, 0x40, 0x7F, 0xFC, 0x08, 0x40, 0x04, 0x40, 0x04, 0x40, 0x01, 0x40, 0x00, 0x80,
//好
0x10, 0x00, 0x11, 0xFC, 0x10, 0x04, 0x10, 0x08, 0xFC, 0x10, 0x24, 0x20, 0x24, 0x24, 0x27, 0xFE,
0x24, 0x20, 0x44, 0x20, 0x28, 0x20, 0x10, 0x20, 0x28, 0x20, 0x44, 0x20, 0x84, 0xA0, 0x00, 0x40,
//设
0x01, 0xF0, 0x21, 0x10, 0x11, 0x10, 0x11, 0x10, 0x01, 0x10, 0x02, 0x0E, 0xF4, 0x00, 0x13, 0xF8,
0x11, 0x08, 0x11, 0x10, 0x10, 0x90, 0x10, 0xA0, 0x14, 0x40, 0x18, 0xB0, 0x13, 0x0E, 0x0C, 0x04,
//外
0x08, 0x40, 0x08, 0x40, 0x08, 0x40, 0x10, 0x40, 0x1F, 0x40, 0x21, 0x60, 0x21, 0x50, 0x62, 0x48,
0x92, 0x46, 0x0A, 0x42, 0x04, 0x40, 0x08, 0x40, 0x10, 0x40, 0x20, 0x40, 0x40, 0x40, 0x00, 0x40,
//准
0x01, 0x40, 0x41, 0x20, 0x31, 0x24, 0x13, 0xFE, 0x02, 0x20, 0x16, 0x20, 0x1B, 0xFC, 0x12, 0x20,
0x22, 0x20, 0xE3, 0xFC, 0x22, 0x20, 0x22, 0x20, 0x22, 0x24, 0x23, 0xFE, 0x22, 0x00, 0x02, 0x00
};
```

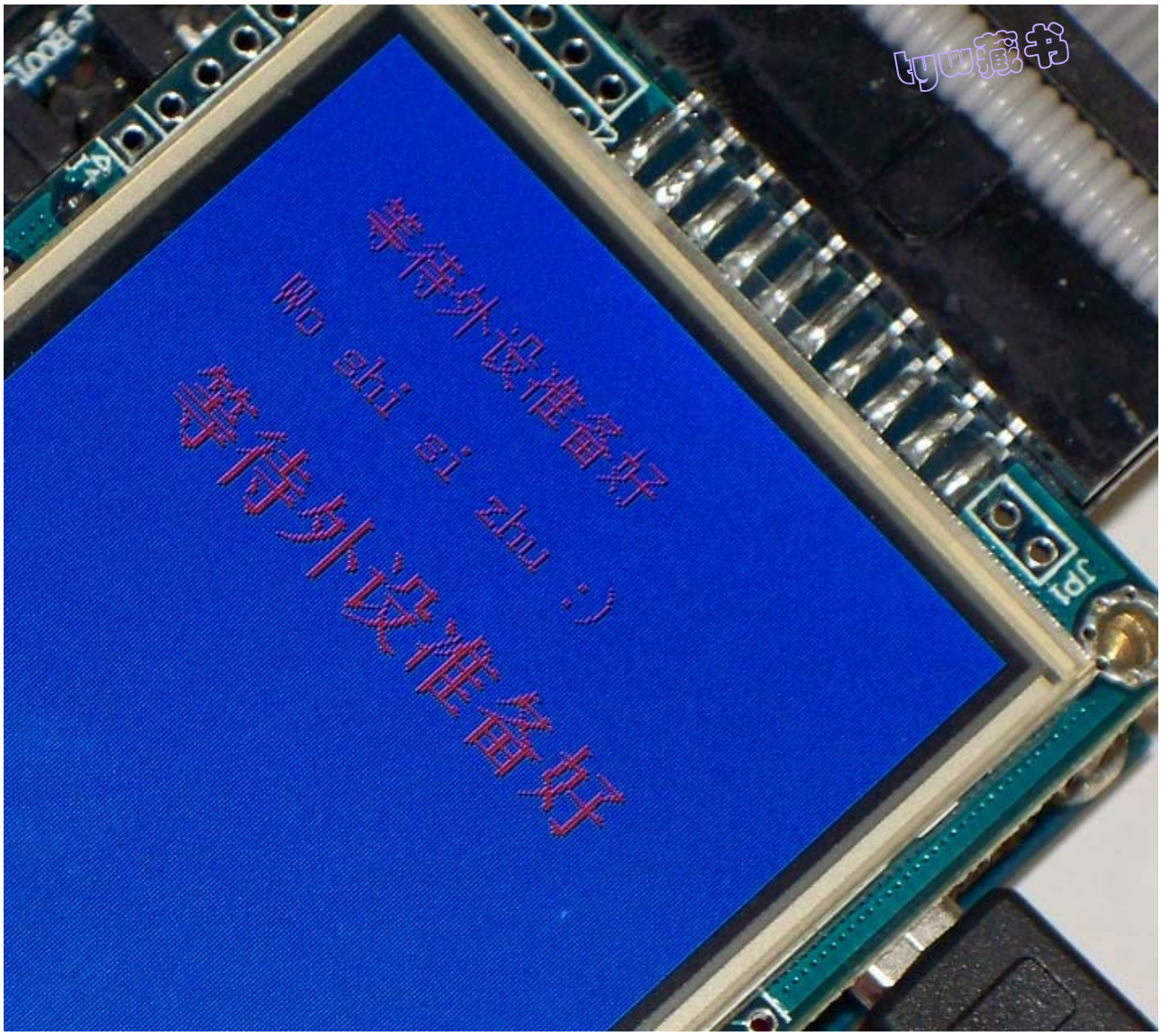
tyw藏书

```
#endif
例子中 24X24 点阵保存为 hzk24.h
#if !defined _HZK24_H
#define _HZK24_H
/*
//共收集汉字:7 个
取模方式：横向-字节顺序
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
D7D6D5D4D3D2D1D0
*/
////**************************************************************************
const char* pHZ24=
"备待等好设外准";
////**************************************************************************
const uchar HZK24[]=
{
//备
0x00, 0x80, 0x00, 0x00, 0xE0, 0x00, 0x01, 0x80, 0x00, 0x01, 0x81, 0x80, 0x03, 0xFF, 0xE0, 0x03, 0x81, 0x80,
0x06, 0x43, 0x00, 0x04, 0x26, 0x00,
0x08, 0x1C, 0x00, 0x10, 0x3C, 0x00, 0x00, 0x67, 0x00, 0x01, 0x81, 0xC0, 0x06, 0x00, 0x7E, 0x1F, 0xFF, 0xF8,
0x66, 0x18, 0x60, 0x06, 0x18, 0x60,
0x06, 0x18, 0x60, 0x07, 0xFF, 0xE0, 0x06, 0x18, 0x60, 0x06, 0x18, 0x60, 0x06, 0x18, 0x60, 0x07, 0xFF, 0xE0,
0x06, 0x00, 0x60, 0x04, 0x00, 0x40,
//待
0x00, 0x04, 0x00, 0x02, 0x07, 0x00, 0x07, 0x06, 0x00, 0x0C, 0x06, 0x00, 0x18, 0x06, 0x30, 0x32, 0x7F, 0xF8,
0x43, 0x06, 0x00, 0x06, 0x06, 0x00,
0x06, 0x06, 0x0C, 0x0D, 0xFF, 0xFE, 0x0C, 0x00, 0xC0, 0x1C, 0x00, 0xC0, 0x2C, 0x00, 0xCC, 0x4C, 0xFF,
0xFE, 0x0C, 0x00, 0xC0, 0x0C, 0x20, 0xC0,
0x0C, 0x30, 0xC0, 0x0C, 0x18, 0xC0, 0x0C, 0x18, 0xC0, 0x0C, 0x10, 0xC0, 0x0C, 0x00, 0xC0, 0x0C, 0x0F,
0xC0, 0x0C, 0x03, 0x80, 0x08, 0x01, 0x00,
//等
0x04, 0x02, 0x00, 0x07, 0x03, 0x80, 0x0C, 0x66, 0x18, 0x0F, 0xF7, 0xFC, 0x19, 0x0C, 0x80, 0x11, 0x88, 0xC0,
0x21, 0xA0, 0xC0, 0x01, 0x38, 0x80,
0x00, 0x30, 0x60, 0x0F, 0xFF, 0xF0, 0x00, 0x30, 0x00, 0x00, 0x30, 0x0C, 0x7F, 0xFF, 0xFE, 0x00, 0x01, 0x00,
0x00, 0x01, 0x98, 0x3F, 0xFF, 0xFC,
0x01, 0x01, 0x80, 0x00, 0xC1, 0x80, 0x00, 0x61, 0x80, 0x00, 0x61, 0x80, 0x00, 0x01, 0x80, 0x00, 0x1F, 0x80,
0x00, 0x07, 0x00, 0x00, 0x02, 0x00,
//好
```

0x08, 0x00, 0x00, 0x0E, 0x00, 0x00, 0x0C, 0x00, 0x08, 0x0C, 0x0F, 0xFC, 0x0C, 0x00, 0x18, 0x0C, 0xC0, 0x30, 0x7F, 0xE0, 0xA0, 0x0C, 0xC0, 0xC0,
0x18, 0xC0, 0xC0, 0x18, 0xC0, 0xC0, 0x18, 0xC0, 0xC0, 0x18, 0xC0, 0xCC, 0x31, 0xBF, 0xFE, 0x31, 0x80, 0xC0, 0x39, 0x80, 0xC0, 0x0D, 0x00, 0xC0,
0x07, 0x00, 0xC0, 0x07, 0x80, 0xC0, 0x0D, 0xC0, 0xC0, 0x08, 0xC0, 0xC0, 0x10, 0xC0, 0xC0, 0x20, 0x0F, 0xC0, 0x40, 0x03, 0x80, 0x00, 0x01, 0x00,
//设
0x00, 0x00, 0x00, 0x18, 0x10, 0x60, 0x0C, 0x1F, 0xF0, 0x06, 0x18, 0x60, 0x06, 0x18, 0x60, 0x02, 0x18, 0x60, 0x00, 0x18, 0x64, 0x00, 0x30, 0x7E,
0x0C, 0x60, 0x3C, 0x7E, 0x80, 0x00, 0x0C, 0x00, 0x30, 0x0C, 0x7F, 0xF8, 0x0C, 0x10, 0x60, 0x0C, 0x10, 0x60, 0x0C, 0x08, 0xC0, 0x0C, 0x28, 0xC0,
0x0C, 0x4D, 0x80, 0x0C, 0x87, 0x00, 0x0F, 0x07, 0x00, 0x0E, 0x0D, 0x80, 0x1C, 0x18, 0xE0, 0x08, 0x60, 0x7E, 0x01, 0x80, 0x18, 0x06, 0x00, 0x00,
//外
0x00, 0x01, 0x00, 0x04, 0x01, 0xC0, 0x07, 0x01, 0x80, 0x06, 0x01, 0x80, 0x06, 0x01, 0x80, 0x06, 0x01, 0x80, 0x0C, 0x31, 0x80, 0x0F, 0xF9, 0x80,
0x08, 0x31, 0x80, 0x18, 0x31, 0xC0, 0x16, 0x31, 0xB0, 0x33, 0x31, 0x9C, 0x23, 0x61, 0x8E, 0x41, 0x61, 0x86, 0x00, 0xC1, 0x84, 0x01, 0x81, 0x80,
0x01, 0x81, 0x80, 0x03, 0x01, 0x80, 0x06, 0x01, 0x80, 0x0C, 0x01, 0x80, 0x30, 0x01, 0x80, 0x40, 0x01, 0x80, 0x00, 0x01, 0x80, 0x00, 0x01, 0x00,
//准
0x00, 0x10, 0x00, 0x00, 0x1D, 0x00, 0x00, 0x19, 0x80, 0x30, 0x18, 0xC0, 0x19, 0x30, 0x8C, 0x1D, 0x3F, 0xFE, 0x09, 0x71, 0x80, 0x02, 0x71, 0x80,
0x02, 0x71, 0x80, 0x04, 0xB1, 0x98, 0x04, 0xBF, 0xFC, 0x09, 0x31, 0x80, 0x0A, 0x31, 0x80, 0x18, 0x31, 0x80, 0x78, 0x31, 0x98, 0x18, 0x3F, 0xFC,
0x18, 0x31, 0x80, 0x18, 0x31, 0x80, 0x18, 0x31, 0x80, 0x18, 0x31, 0x8C, 0x18, 0x3F, 0xFE, 0x08, 0x30, 0x00, 0x00, 0x30, 0x00, 0x00, 0x20, 0x00
};

#endif

下面是 16X16,24X24 汉字显示效果



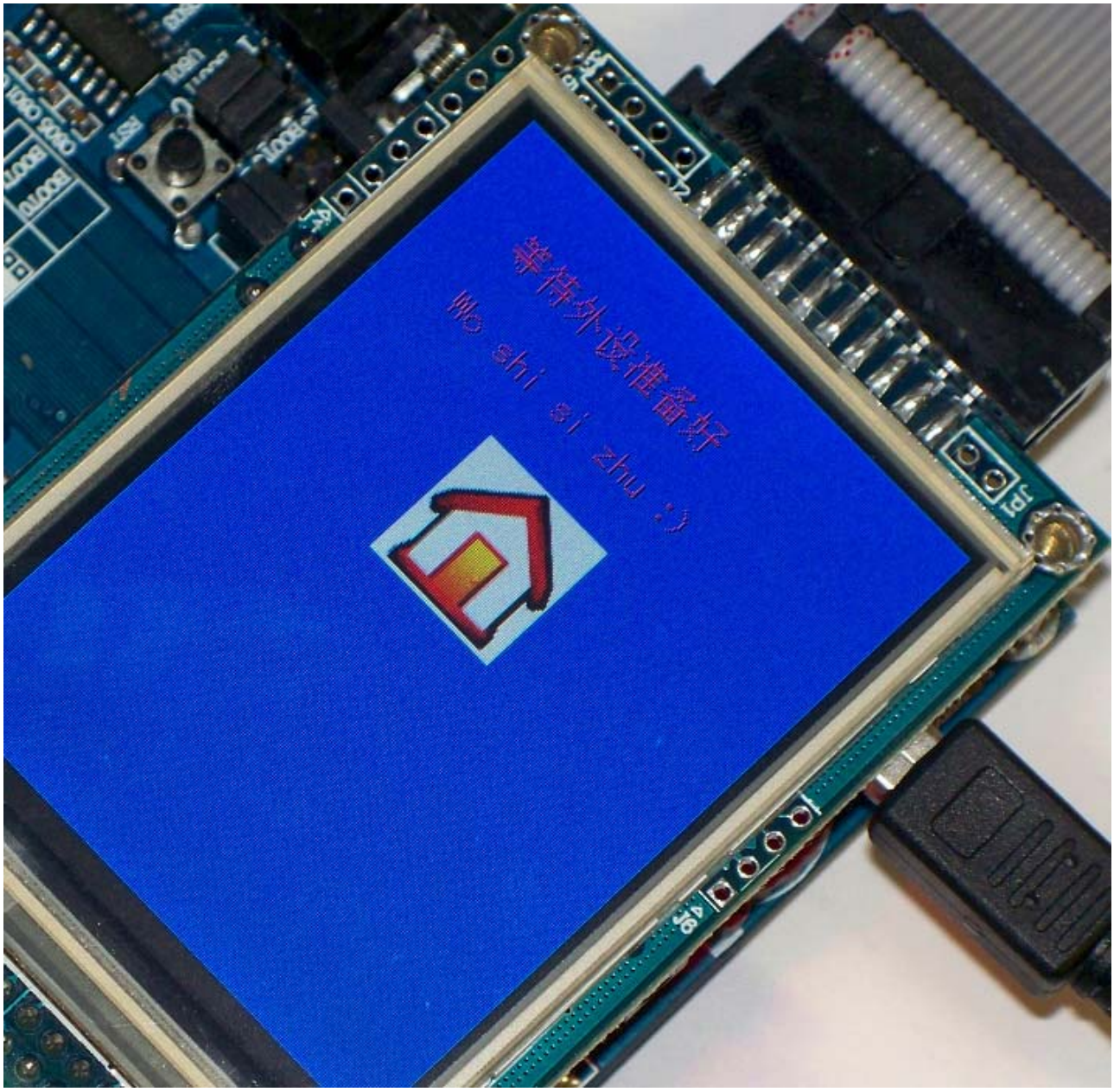
BHS-STM32 实验 34-TFT测试+汉字+图片显示

本例子在上面例子基础上增加了图片显示功能，
本例子使用的图片数据是通过我提供的图片转换工具将 JPG,BMP 转换为 BIN/.H 格式显示

软件说明：
1.可以转换 JPG, BMP 图片格式
2.将图片直接转换为 RGB565 格式输出，这样 CPU 就直接读出数据显示就可以了，不用解码。
3.转换 JPG 文件时，分别输出.bmp 文件，.bin 文件，.h 文件
其中.h 文件可以直接加入程序中使用，方便一些程序只使用些小的图标
对应大的图片建议使用.bin 文件放入 SPI-FLASH 或者 SD 卡中

tyw藏书

本例将图片 house.jpg 转换生成 houseRGB565.h 文件，将 houseRGB565.h 添加到项目中去编译。houseRGB565.h 文件是 RGB565 格式文件，RGB565 格式的数据可以直接送 16 位模式下 TFT 显示。下面是显示效果



BHS-STM32 实验 35-USART一个完整通信协议

通信协议【资料文档\BHS-STM32 文档】《BHS-STM32 IAP 通讯协议 V1.0.pdf》
通讯命令格式描述如表格：

序号	1	2	3	4	5	6
内容	起始符	校验	数据长度	从机地址	功能码	参数/数据
字节	1	2	2	2	2	N
说明	***0x02***	xxxx	xxxx	xxxx	xxxx	xxxx

lyw藏书

[起始符]

1Byte，固定的 ***0x02***

[数据长度]

2Byte，整个协议包长度，从[校验]到[数据]最后字节（包含[长度]本身字节）

[校验]

2byte，是从[数据长度]到[数据]最后字节所有字节的累加和（16 bit 字的二进制反码和）。

整个数据报的长度可能为奇数字节，但是检验和算法是把若干个16 bit字相加。解决方法是在最后增加填充字节0，这只是为了检验和的计算（也就是说，可能增加的填充字节不被传送）。

[目的地址]

2Byte，数据要到达的地址

[功能码]

2Byte

[数据]

N Byte，可以为空

说明：下面具体命令只对【参数/数据】解释

2 命令说明

■ (0x0001) 联机测试

请求→

内容	字节	说明
数据	0	

应答→

内容	字节	说明
数据	0	

■ (0x0007) 读设备时间

请求→

内容	字节	说明
数据	0	

应答→

内容	字节	说明
时间	0	HEX，高位在前 从 1970 年 1 月 1 日 00：00：00 到当前的秒数，即 UTC 时间

说明：

■ (0x0008) 写设备时间

tyw藏书

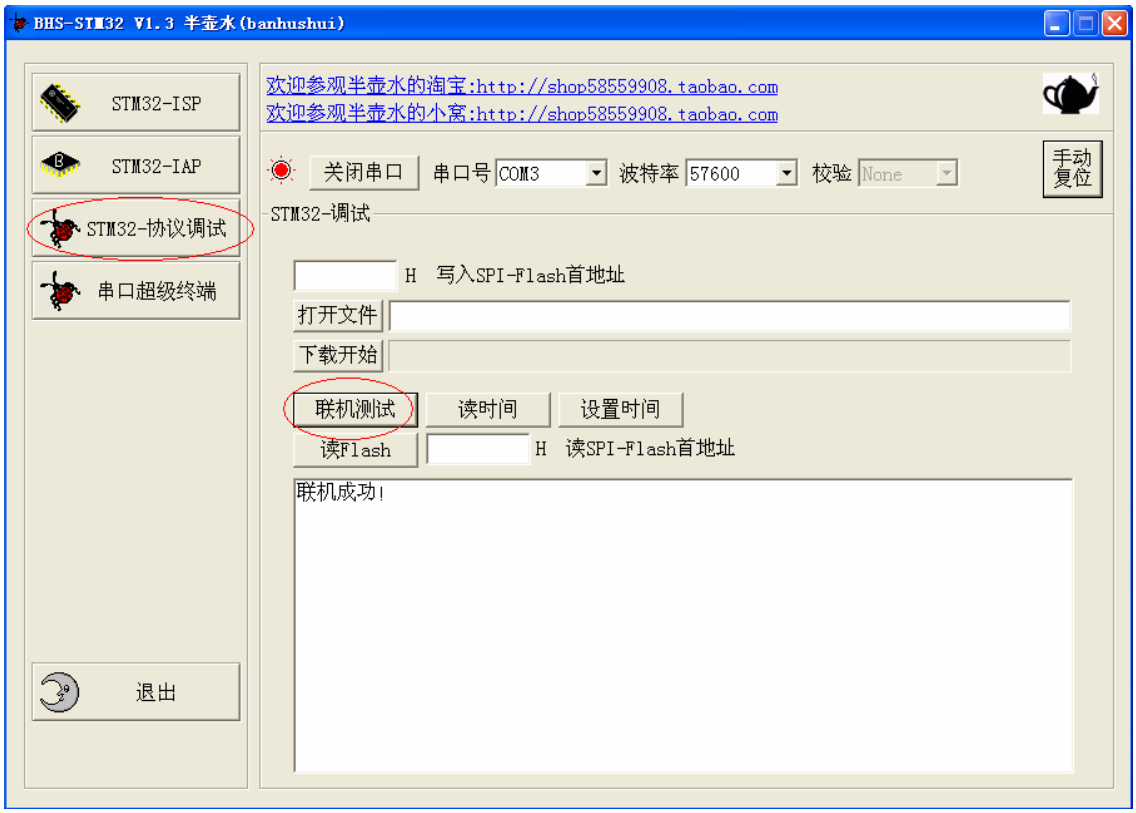
请求→

内容	字节	说明
时间	4	HEX，高位在前 从 1970 年 1 月 1 日 00：00：00 到当前的秒数，即 UTC 时间

应答→

内容	字节	说明
数据	0	

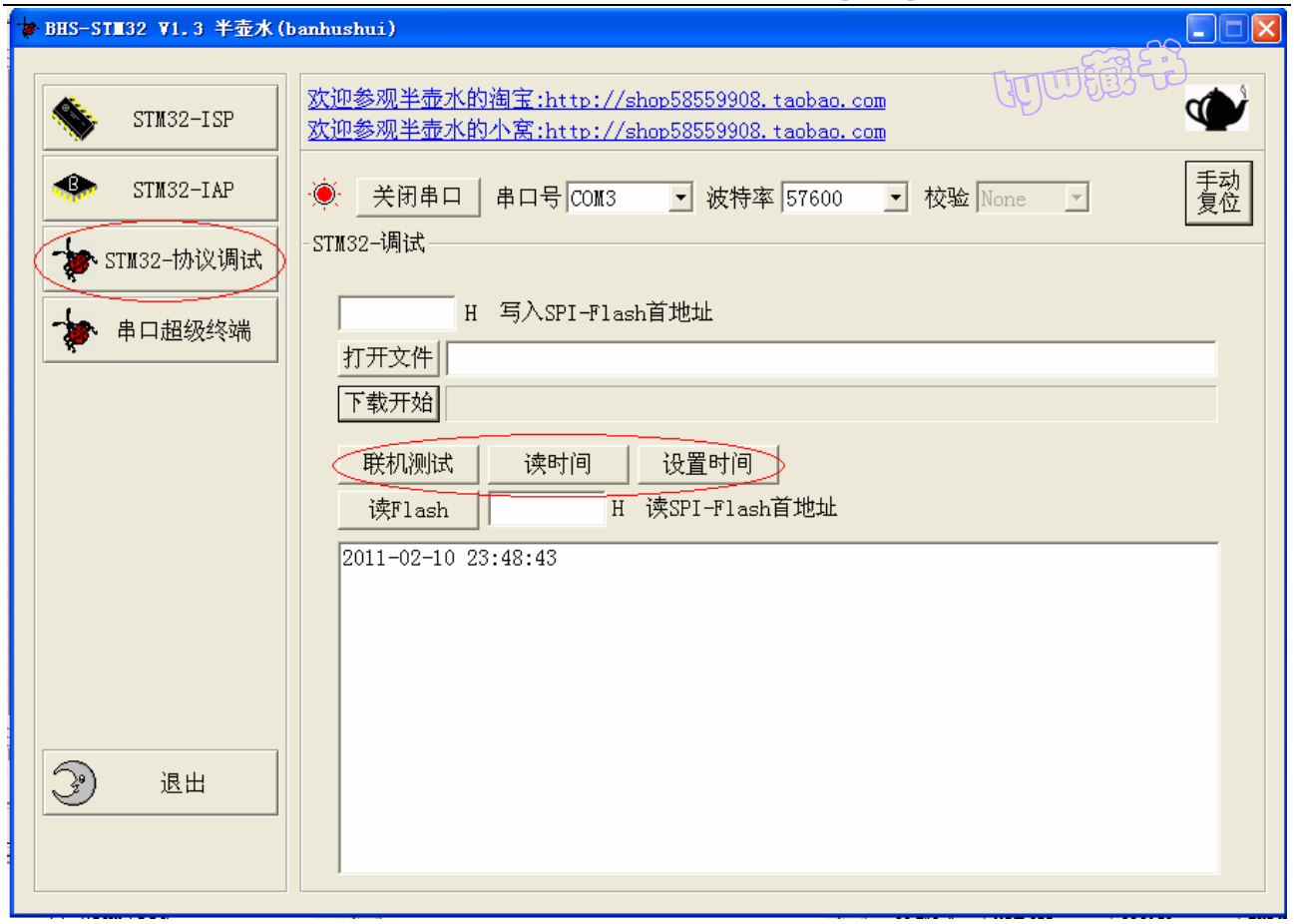
PC 软件使用前面提到的调试工具



选择 STM32-协议调试，波特率 57600，这个例子只支持一个命令：【联机测试】

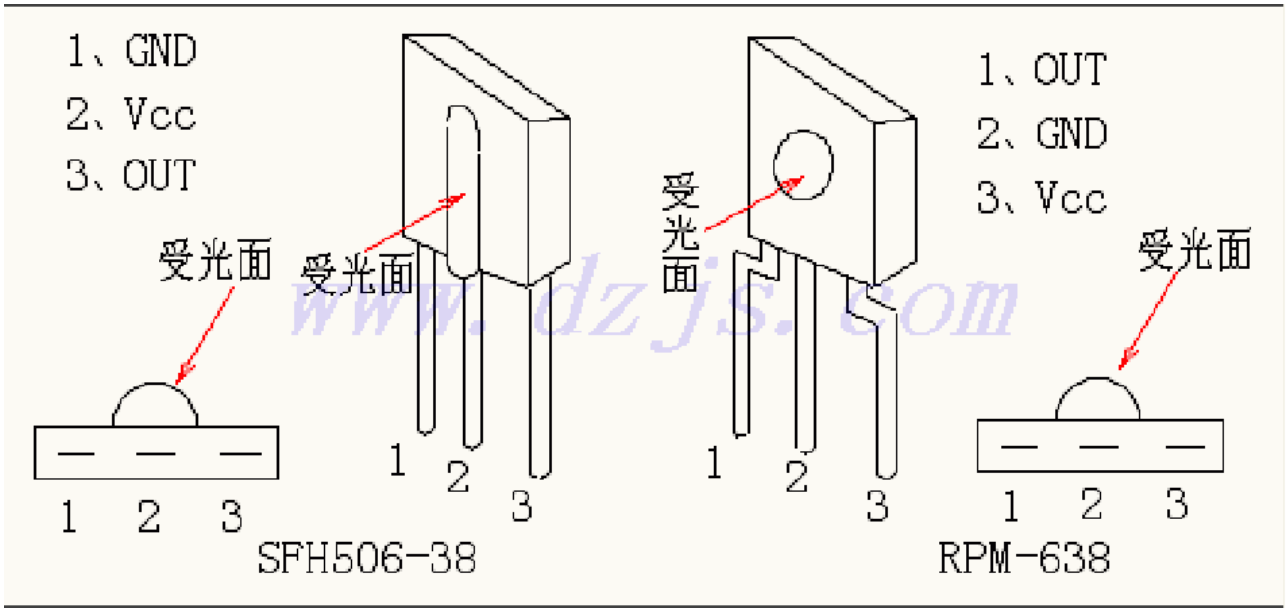
BHS-STM32 实验 36-USART—个完整通信协议+RTC实时时钟

通信协议【资料文档\BHS-STM32 文档】《BHS-STM32 IAP 通讯协议 V1.0.pdf》
选择 STM32-协议调试，波特率 57600，本例子在上例基础上增加【读时间】【设置时间】

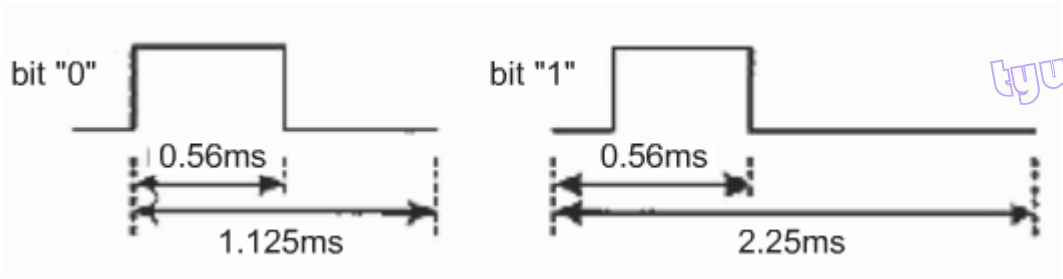


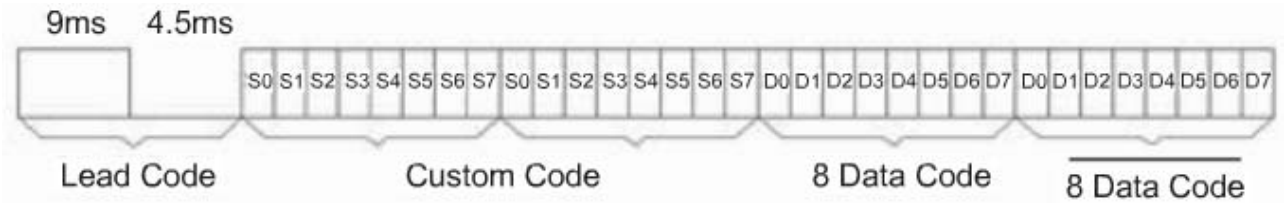
BHS-STM32 实验 37-红外接收

目前一般都采用一体化红外接收头：

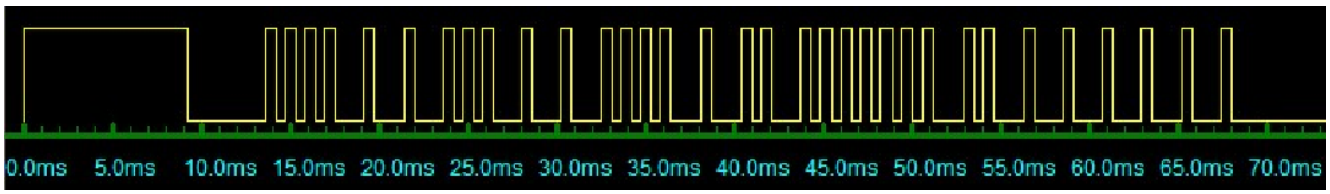


目前家电使用最多的是 NEC 的红外编码格式：





红外一般是 38K 的调制脉冲信号，实际我们使用的一体化红外接收头接收调制后的脉冲，结果解调后输出给我们的是电平信号。如下图：



说明：图中波形是反向的
//红外数据格式
同步头 + 8bit 用户码 + 8bit 用户码反码 + 8bit 数据 + 8bit 数据反码

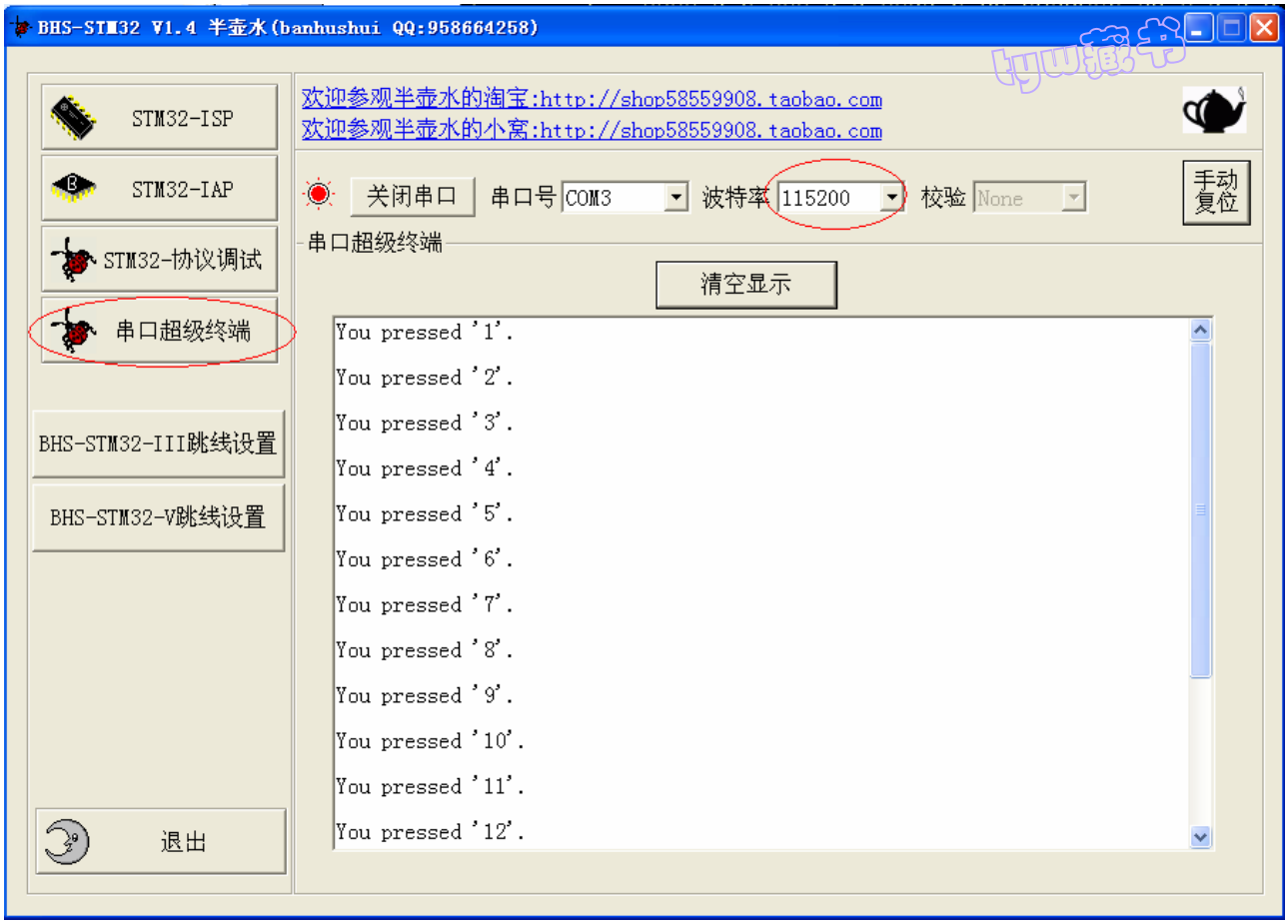
//同步头： 9ms 低电平，4.5ms 高电平
数据：
//0.5ms 低电平,0.5ms 高电平 ==BIT 0
//0.5ms 低电平,1ms 高电平 ==BIT 1

连续码：
9ms 低电平+2.2ms 高电平+0.6ms 低电平

上面的时间是个大概值，不一定精确
本例实验接收到红外信号 LED 将交替亮灭

BHS-STM32 实验 38-按键蜂鸣器测试

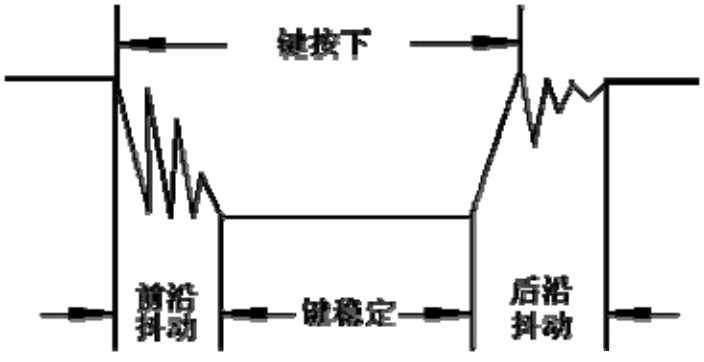
本例功能，按下按键蜂鸣器响（对于有蜂鸣器的板子）并且通过串口可以看到按下的键值：如下图



前面实际我们已经介绍了 GPIO 输入，输出，串口
这个例子就是 GPIO 输入输出串口综合运用

按键消抖：

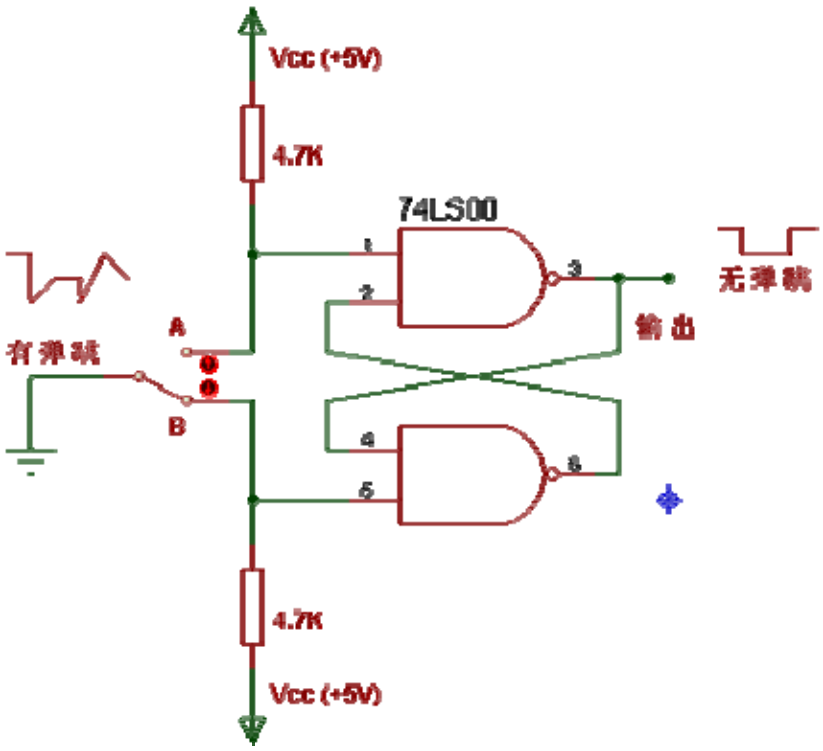
通常的按键所用开关为机械弹性开关,当机械触点断开、闭合时,电压信号小型如下图。由于机械触点的弹性作用,一个按键开关在闭合时不会马上稳定地接通,在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动,如下图。抖动时间的长短由按键的机械特性决定,一般为 5ms～10ms。这是一个很重要的时间参数,在很多场合都要用到。



按键稳定闭合时间的长短则是由操作人员的按键动作决定的,一般为零点几秒至数秒。键抖动会引起一次按键被误读多次。为确保 CPU 对键的一次闭合仅作一次处理,必须去除键抖动。在键闭合稳定时读取键的状态,并且必须判别到键释放稳定后再作处理。按键的抖动,可用硬件或软件两种方法。

〈1〉硬件消抖：在键数较少时可用硬件方法消除键抖动。下图所示的 RS 触发器为常用的硬件去抖。

tyw藏书



图中两个“与非”门构成一个 RS 触发器。当按键未按下时,输出为 1;当键按下时,输出为 0。此时即使用按键的机械性能,使按键因弹性抖动而产生瞬时断开（抖动跳开 B）,中要按键不返回原始状态 A,双稳态电路的状态不改变,输出保持为 0,不会产生抖动的波形。也就是说,即使 B 点的电压波形是抖动的,但经双稳态电路之后,其输出为正规的矩形波。这一点通过分析 RS 触发器的工作过程很容易得到验证。

〈2〉 软件消抖：如果按键较多,常用软件方法去抖,即检测出键闭合后执行一个延时程序,产生 5ms～10ms 的延时,让前沿抖动消失后再一次检测键的状态,如果仍保持闭合状态电平,则确认为真正有键按下。当检测到按键释放后,也要给 5ms～10ms 的延时,待后沿抖动消失后才能转入该键的处理程序。

高级例程-(应用篇)

高级例程主要涉及多种操作系统，GUI 图形界面，TCP，USB，文件系统，IAP 远程更新用户程序。

这里面的例程都需要比较扎实过硬的基础了,学习操作系统的首先要了解操作系统原理,这里推荐《uCOS-II 中文书》邵贝贝翻译，此书详细介绍了 uCOS 操作系统原理，当然其他操作系统原理也差不太多。另外文件系统，TCP，USB 也是相当复杂的技术，光是介绍协议的书籍就厚厚的，所以学习不是一两月的事。本人时间精力有限，也是略知皮毛，谈不上精通。希望和大家一起交流学习。

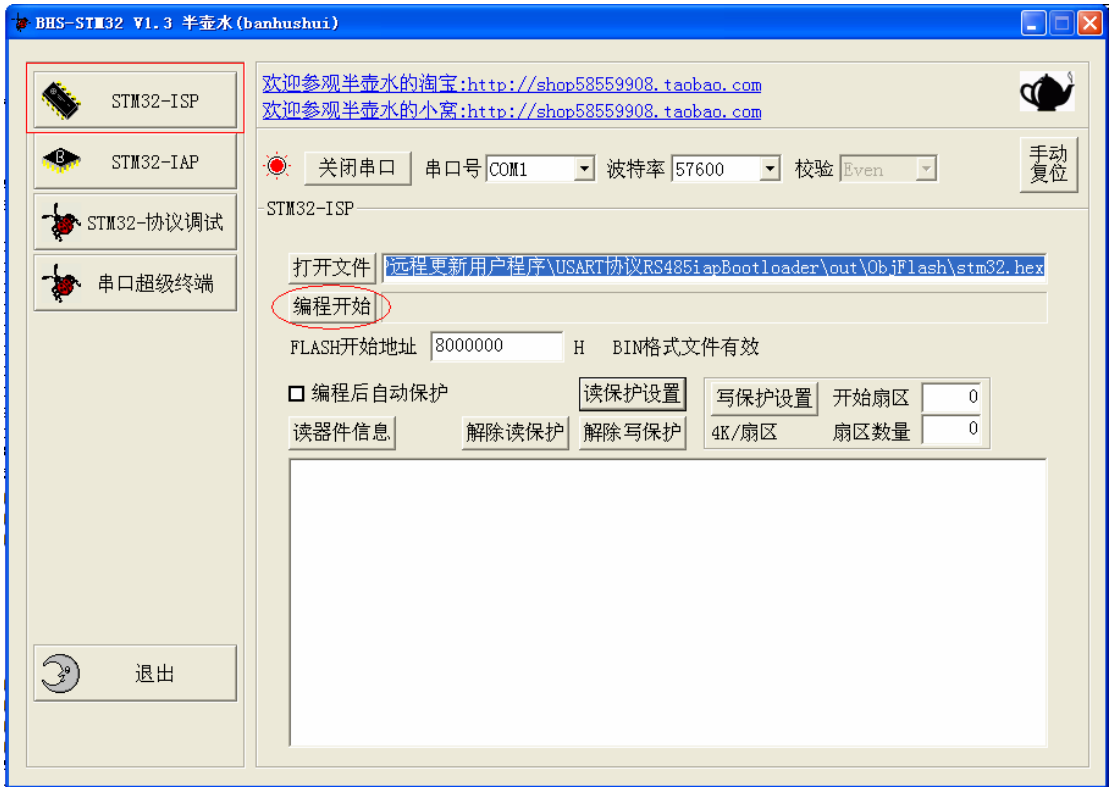
BHS-STM32 实验 39-IAP远程更新用户程序

本实验使用的通信协议在【资料文档\BHS-STM32 文档】《BHS-STM32 IAP 通讯协议 V1.0.pdf》
很多应用需要更新用户程序，比如：你的设备已经在用户手里使用，需要增加新功能，修正一下小错误。设备招回几乎是不可能的时，即使可能也要耗费不少时间和金钱。所以远程更新用户程序是非常必要的。
远程更新用户程序原理：
本例程是通过串口更新用户程序。当然如果你有精力和能力完全可以修改为 USB,TCP,GPRS,等其他方式。

首先在 FLASH 开始区域划分 8K 空间做引导区域，系统上电先运行引导区域程序，如果一段时间内没有程序更新请求命令那么就进入用户程序。用户程序也可以在收到更新程序命令后执行系统复位进入引导程序，那么这样就不需要任何干预就可以更新用户程序了。

【\BHS-STM32 例程\高级例程-(实战篇)\IAP 远程更新用户程序\USART 协议 RS485iapBootloader】就是引导程序。引导程序主要就是编程用户 FLASH 的。

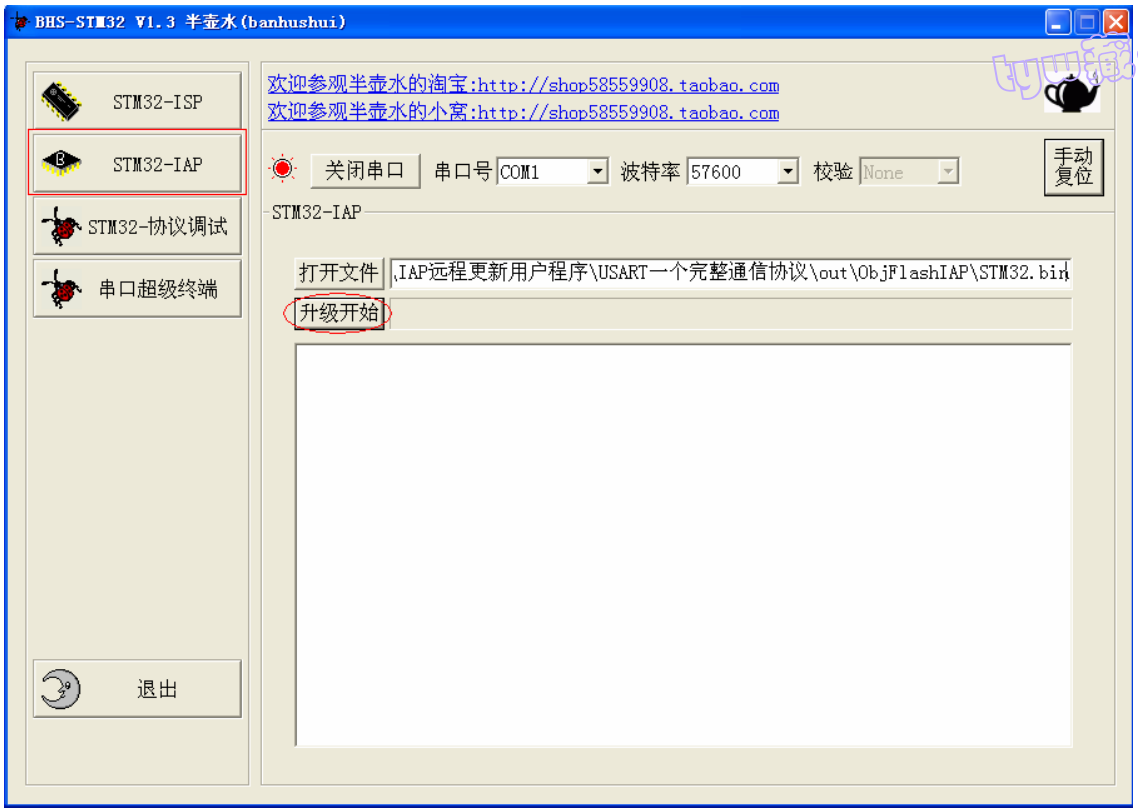
首先选择 FLASH 编译该例程，在\out\ObjFlash\生成 stm32.hex，开发板启动模式设置为 ISP 模式，使用我提供的 ISP 工具先将引导写入 CPU 中,



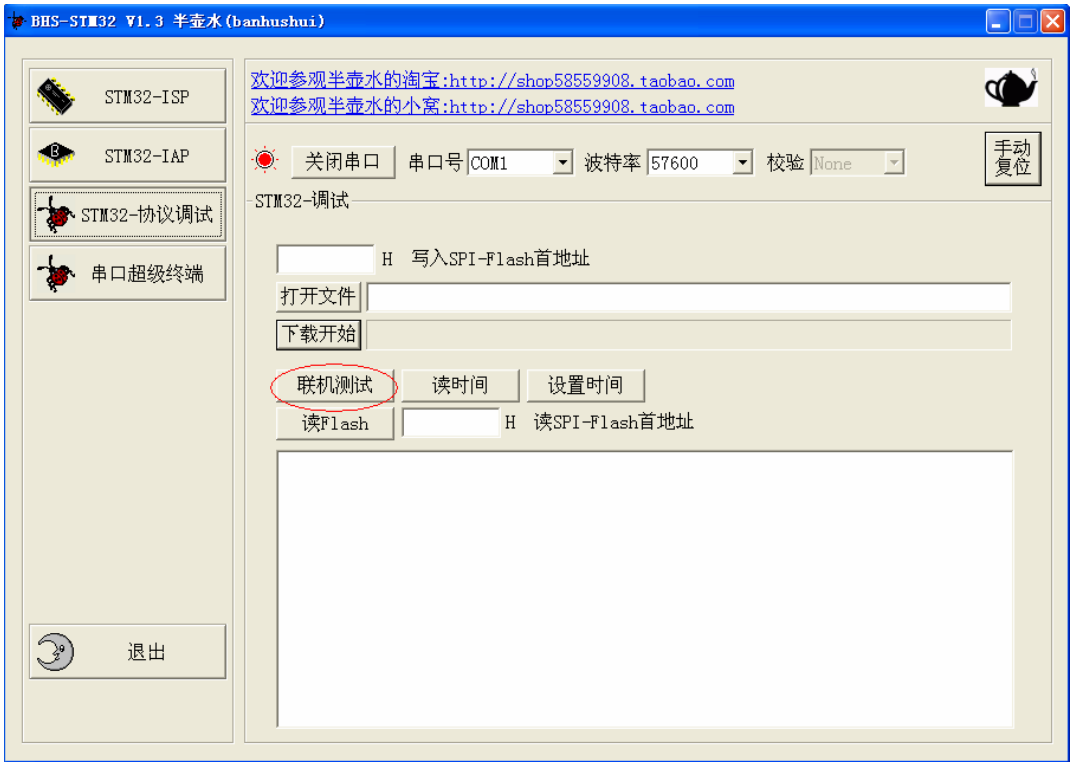
设置启动模式为用户模式为复位后可以看到 LED 快速闪烁，那么现在系统就用更新用户程序功能了，同样使用我提供的工具，软件切换到 STM32-IAP 模式，用户程序使用【\BHS-STM32 例程\高级例程-(实战篇)\IAP 远程更新用户程序\USART 一个完整通信协议】，选择 FLASH-IAP 编译该例子在\out\ObjFlashIAP 生成 STM32.bin

原理：

将 CPU 的 FLASH 分为 2 部分，1 部分是引导程序，1 部分是用户程序。系统上电时进入引导程序，如果一段时间内没有检测到串口升级命令，那么引导程序自动退出，跳转到用户程序执行。如果有升级命令，那么该引导程序开始升级用户程序。

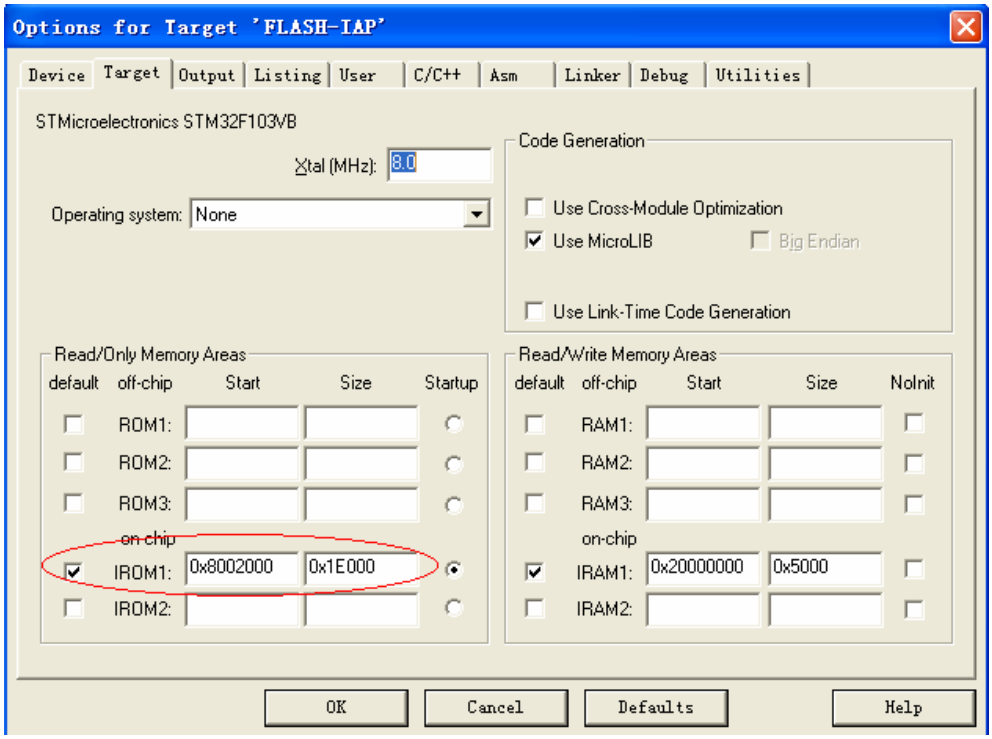


更新完程序后可以切换到【STM32-协议调试】模式，这个例子只支持【联机测试】功能，另外【USART 一个完整通信协议+RTC 实时时钟】用户程序增加了设置/读取时间功能。

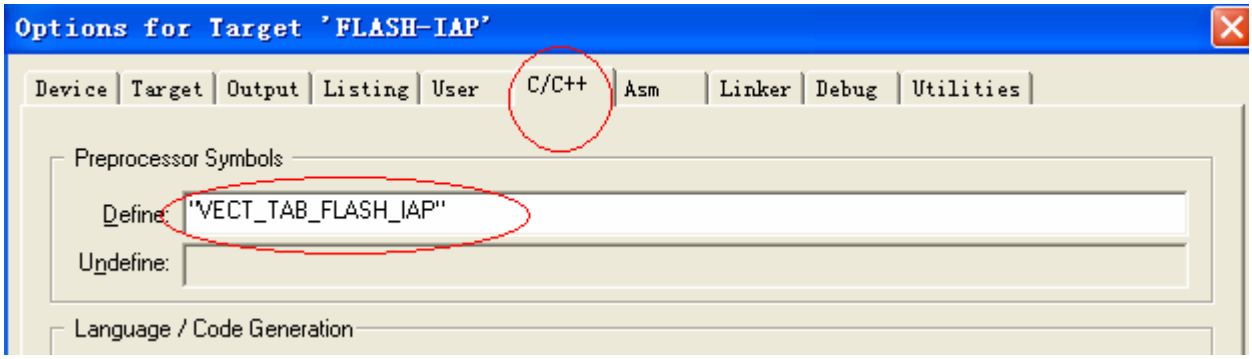


用户程序实际是从 0x8002000 开始运行的，那么用户程序需要注意下面几点：

1.设置用户程序开始地址（注意是 0x8002000）



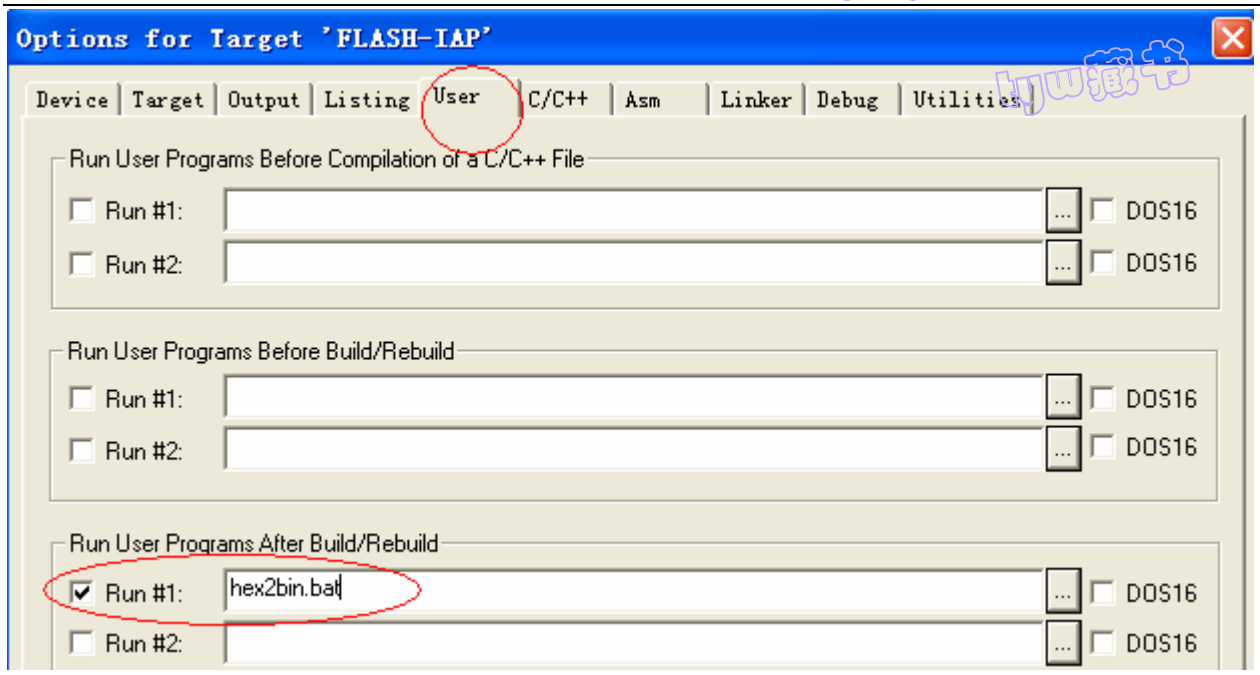
2. 重定位中断向量表,设置编译预处理定义



重定位中断向量表代码（在 bsp.c 文件中）

```
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

//#ifdef  VECT_TAB_RAM
#if defined (VECT_TAB_RAM)
  /* Set the Vector Table base location at 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#elif defined(VECT_TAB_FLASH_IAP)
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x2000);
#else  /* VECT_TAB_FLASH  */
  /* Set the Vector Table base location at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif
```

3.设置生成 BIN 文件



编译后自动运行 hex2bin.bat 批处理文件，该文件内容如下：

hex2bin.exe　..\out\ObjFlashIAP\STM32.hex

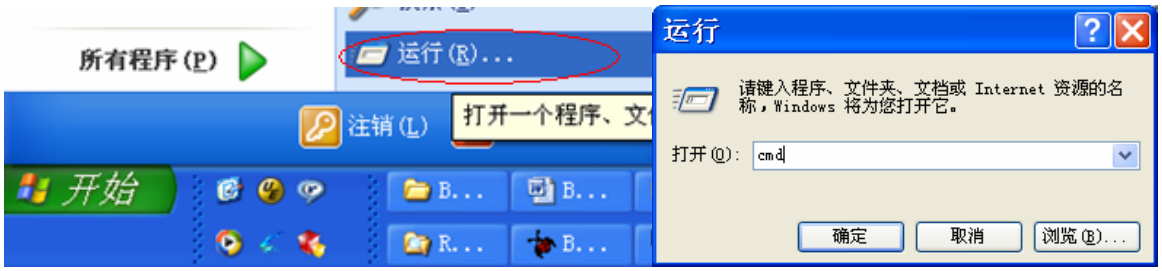
实际上 hex2bin.exe 是一个将 HEX 格式文件转换为 BIN 格式文件的小工具

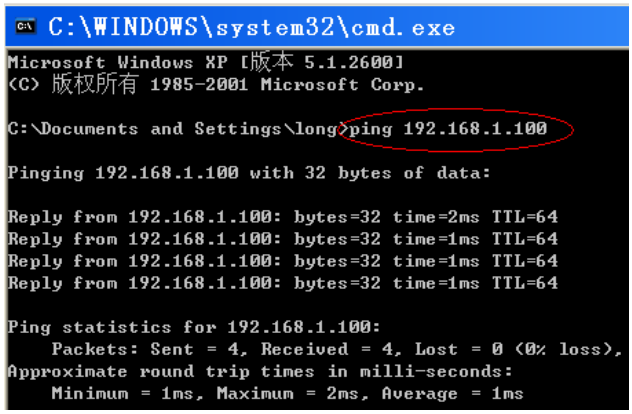
BHS-STM32 实验 40-网页控制LED

本例是一个简单的 WEB 程序，通过网页控制 LED 的亮灭，程序默认 IP 是 192.168.1.100

要使用该例程需要将你电脑 IP 设置为同一个网段：192.168.1.xxx

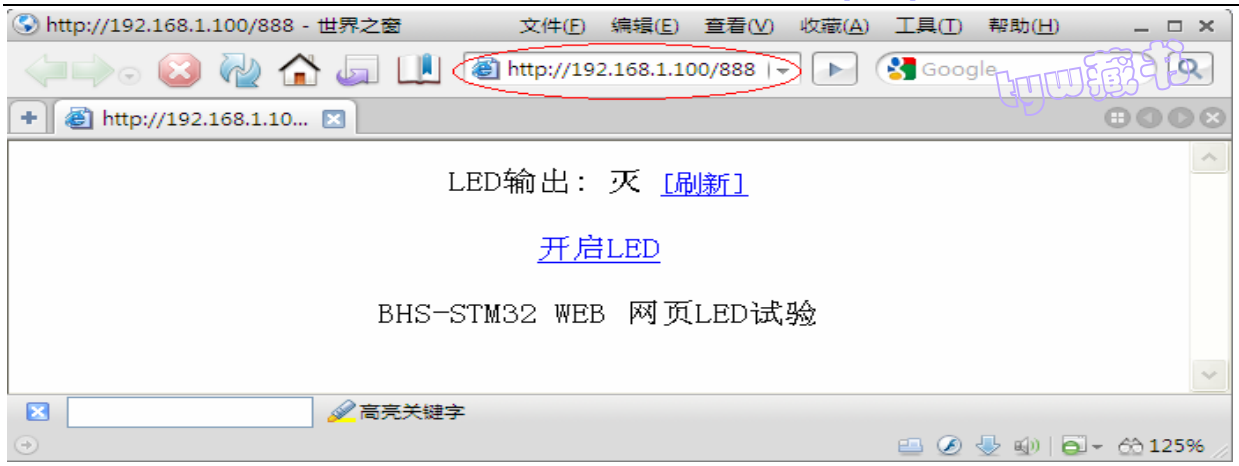
使用提供的交叉网线连接好开发板和电脑的网口





在命令行输入：ping 192.168.1.100 能看到已经连接上网络了

在IE地址蓝输入：http://192.168.1.100/888　　将打开如下网页，现在可以同网页控制LED了



如果网络不通可能的原因如下：

a. 你调试电脑的 IP 地址与开发板不是同一个网段
b. 部分防火墙程序禁止了访问，所以做网络实验最好关掉网络防火墙软件
c. 如果你的电脑有 2 个或以上网卡，最好禁止其他网卡，只保留 1 个网卡调试用
d. 曾经遇到部分 GHOST XP 系统的电脑不能 PING 通网络，具体原因是什么也不是太清楚，最好安装纯净版系统或者在其他电脑上测试下
e. 配套的网线是交叉网线，可以直接连电脑网卡调试，如果你使用直通网线直接连开发板是无法调试的

BHS-STM32 实验 41-VirtualCOMPort(USB虚拟串口)

该例子实现 USB 转串口功能
编译该例子生成 HEX，下载程序到开发板，将开发板 USB 插入电脑 USB，可见到找到新硬件提示，这是需要安装驱动。驱动程序在
\BHS-STM32 例程\高级例程-(实战篇)\VirtualCOMPort-BHS-STM32(虚拟串口)\虚拟串口驱动.inf
驱动安装好后在[设备管理器]里可以看到多了个串口
说明：开发板上的串口是串口 1

BHS-STM32 实验 42-BHS-STM32+FATFS R0.07C文件系统+BMP显示

此例子演示从 SD 卡读取 BMP 图片文件显示到 TFT 上，SD 卡使用的文件系统是 FATFS 开源文件系统，改文件系统网络上可找到资源比较丰富，也比较成熟。
首先将【\BHS-STM32 例程\高级例程-(实战篇)\BHS-STM32+FATFS R0.07C 文件系统+BMP 显示】文件夹里的 BMP 图片复制到 SD 卡根目录，将 SD 卡插入开发板。运行该例子，TFT 上将显示刚才复制的图片

FatFS相关知识

FatFS简介：

FatFs是一个通用的文件系统模块，用于在小型嵌入式系统中实现FAT文件系统。 FatFs 的编写遵循ANSI C，因此不依赖于硬件平台。它可以嵌入到便宜的微控制器中，如 8051, PIC, AVR, SH, Z80, H8, ARM 等等，不需要做任何修改。
官方网站：http://elm-chan.org/fsw/ff/00index_e.html

特点：

- 支持 FAT12, FAT16 与 FAT32.
- 支持多个卷(物理驱动器与分区).
- 两种分区规则： FDISK 与 Super-floppy.
- 多种配置选项:
- 长文件名支持。

- 可选的编码页，包括 DBCS（DBCS 为双位元组字元系统 Double Byte Char Systems)
- 多任务支持
- 只读，最小化 API，缓冲区配置等等

tyw藏书

应用程序接口

FatFs 提供下面的函数：

※ f_mount - 注册/注销一个工作区域（Work Area）
※ f_open - 打开/创建一个文件
※ f_close - 关闭一个文件
※ f_read - 读文件
※ f_write - 写文件
※ f_lseek - 移动文件读/写指针
※ f_truncate - 截断文件
※ f_sync - 冲洗缓冲数据 Flush Cached Data
※ f_opendir - 打开一个目录
※ f_readdir - 读取目录条目
※ f_getfree - 获取空闲簇 Get Free Clusters
※ f_stat - 获取文件状态
※ f_mkdir - 创建一个目录
※ f_unlink - 删除一个文件或目录
※ f_chmod - 改变属性（Attribute）
※ f_utime - 改变时间戳（Timestamp）
※ f_rename - 重命名/移动一个文件或文件夹
※ f_mkfs - 在驱动器上创建一个文件系统
※ f_forward - 直接转移文件数据到一个数据流 Forward file data to the stream directly
※ f_gets - 读一个字符串
※ f_putc - 写一个字符
※ f_puts - 写一个字符传
※ f_printf - 写一个格式化的字符

磁盘I/O接口

因为 FatFs 模块完全与磁盘 I/O 层分开，因此需要下面的函数来实现底层物理磁盘的读写与获取当前时间。底层磁盘 I/O 模块并不是 FatFs 的一部分，并且必须由用户提供。资源文件中也包含有范例驱动。

※ disk_initialize - Initialize disk drive 初始化磁盘驱动器
※ disk_status - Get disk status 获取磁盘状态
※ disk_read - Read sector(s) 读扇区
※ disk_write - Write sector(s) 写扇区
※ disk_ioctl - Control device dependent features 设备相关的控制特性
※ get_fattime - Get current time 获取当前时间

FatFs 使用说明

FatFs 模块是用 ANSI C 编写的中间件，只要编译器遵循 ANSI C，它都是平台无关的。

FatFs 假定 char/short/long 的长度为 8/16/32 位，而 int 为 16 位或 32 位，这些相应的定义位于 integer.h 文件。这在大多数的编译器上都不会是问题，但是当与预定义的内容发生冲突时，你必须小心地解决。

内存使用(R0.07)

	AVR	H8/300H	PIC	TLCS-870/C	V850ES	SH2	ARM7TDMI	IA-32
编译器	gcc(WinAVR)	CH38	gcc(C30)	CC870C	CA850	SHC	gcc(WinARM)	MSC
_WORD_ACCESS	1	0	0	1	1	0	0	1
ROM (Full, R/W)	11136	10356	10838	15167	7682	8654	10628	7232
ROM (Min, R/W)	7072	6696	7007	9800	4634	5570	6564	4647
ROM (Full, R/O)	5218	4626	4949	6786	3528	3826	4676	3267
ROM (Min, R/O)	3626	3418	3536	4941	2558	2874	3272	2397
RAM (Static)	D*2 + 2	D*4 + 2	D*2 + 2	D*2 + 2	D*4 + 2	D*4 + 2	D*4 + 2	D*4 + 2
RAM (Dynamic) (_FS_TINY == 0)	D*560 + F*544	D*560 + F*550	D*560 + F*544		D*560 + F*550	D*560 + F*550	D*560 + F*550	D*560 + F*550
RAM (Dynamic) (_FS_TINY == 1)	D*560 + F*32	D*560 + F*36	D*560 + F*32	D*560 + F*32	D*560 + F*36	D*560 + F*36	D*560 + F*36	D*560 + F*36

这是在以下情形时一些目标系统的内存用量，内存大小以字节计算，D 代表卷数量，F 代表打开文件的数量。所有例子都是已优化代码大小的。

减小模块大小

下面的表显示了通过设置配置选项哪些函数将会移除，进而减小模块大小

Function	_FS_MINIMIZE			_FS_READONLY	_USE_STRFUNC	_USE_MKFS	_USE_FORWARD
	1	2	3	1	0	0	0
f_mount							
f_open							
f_close							
f_read							
f_write				x			
f_sync				x			
f_lseek			x				
f_opendir		x	x				
f_readdir		x	x				
f_stat	x	x	x				
f_getfree	x	x	x	x			
f_truncate	x	x	x	x			

f_unlink	x	x	x	x					
f_mkdir	x	x	x	x					
f_chmod	x	x	x	x					
f_utime	x	x	x	x					
f_rename	x	x	x	x					
f_mkfs				x			x		
f_forward								x	
f_putc				x		x			
f_puts				x		x			
f_printf				x		x			
f_gets						x			

tyw藏书

长文件名支持

FatFs 从 0.07 版本开始支持长文件名（LFN）。在调用文件函数时，一个文件的两个文件名（SFN 与 LFN）是通用的，除了 f_readdir 函数。支持长文件特性将需要一个额外的工作缓冲区，此缓冲区的大小可以通过设置_MAX_LFN 来以可用的内存大小相符。因为长文件名 长达 255 个字符，因此_MAX_LFN 应该设置为 255 来支持全特性的 LFN 选项。当工作缓冲区的大小容不下给出的文件名时文件函数就会因为 FR_INVALID_NAME 而调用失败。

当使能 LFN，将需要一个巨大的 OEM-Unicode 双向转换表，模块的大小将大大的增大

重入

对不同卷的文件操作总是可以同时地工作，而与重入设置无关。而对于同一个卷的重入访问可以通过使能 _FS_REENTRANT 选项。此 时，在 ff.c 中的与平台相关的锁定函数必须为每个 RTOS 重新编写。如果一个文件函数调用时其访问的卷正被另一个线程使用，则此访问将阻塞直到该卷解 锁。如果等待时间超过了_TIMEOUT 毫秒，则函数将因 FR_TIMEOUT 而终止。某些 RTOS 可能不支持超时操作。

临界段

当对 FAT 文件系统的写操作由于默写意外而中断，如突然断电，不正确的磁盘移除或不可恢复的磁盘错误，FAT 结构可以被毁坏。下面的图片显示了 FatFs 的临界段。

Figure 4. Long critical section

Figure 5. Minimized critical section

```
f_mount(...);

f_open(...);          //Create  file

// any procedure

do {
    t = get_adc(...);

    // any procedure

    f_write(...);     // write file

    delay_second(1);

} while (...);

// any procedure

f_close(...);         // close file
```

```
f_mount(...);

f_open(...);          //Create  file
f_sync(...);
// any procedure

do {
    t = get_adc(...);

    // any procedure

    f_write(...);     // write file
    f_sync(...);
    delay_second(1);

} while (...);

// any procedure

f_close(...);         // close file
```

```
f_mkdir(...);

f_rename(...);

f_unlink(...);
```

```
f_mkdir(...);

f_rename(...);

f_unlink(...);
```

红色区域的中断会导致一个交叉链接，结果，正在修改的文件/目录可能会丢失。而黄色区域中断可能导致的效果在下面列出：

※ 正在重写的文件数据被毁坏
※ 正在添加内容的文件回到初始状态
※ 丢失新建的文件
※ 一个新建或覆盖的文件保持长度为 0
※ 因为丢失关联，磁盘的使用效率变坏。

在文件不是用写模式打开时，这些情况不会发生。为了最小化磁盘数据的丢失，临界段可以像图表 5 显示的那样最小化，通过最小化文件处于写模式打开的时间或者适当的使用 f_sync 函数。

以上信息来自：http://www.openrtos.cn，http://elm-chan.org/fsw/ff/00index_e.html

BMP知识

BMP 是一种与硬件设备无关的图像文件格式，也是我们最常在 PC 机上的 Windows 系统下见到的标准位图格式，使用范围很广泛。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，BMP 文件所占用的空间很大。它最大的好处就是能被大多数软件“接受”，可称为通用格式。

BMP 在过去是比较普及的图像格式，现在 BMP（Window 位图）图像主要被用在 PC 机运行 Window 时的墙纸。BMP 可以提供无损压缩，压缩方式叫 RLE（游程长度编码的编写），在创建墙纸图像文件时是一个极好的选项。Window 有时在查找以 RLE 压缩文件方式保存的墙纸图像时也会出现识别错误。，因此使用时最好先关闭 RLE 压缩功能。

BMP 文件由文件头、位图信息头、颜色信息和图形数据四部分组成。

1、 BMP 文件头：BMP 文件头数据结构含有 BMP 文件的类型、文件大小和位图起始位置等信息。

```
typedef struct tagBITMAPFILEHEADER{
WORD bfType; // 位图文件的类型，必须为 BM
DWORD bfSize; // 位图文件的大小，以字节为单位
```

WORD bfReserved1; // 位图文件保留字，必须为 0
WORD bfReserved2; // 位图文件保留字，必须为 0
DWORD bfOffBits; // 位图数据的起始位置，以相对于位图文件头的偏移量表示，以字节为单位
} BITMAPFILEHEADER;

2、 位图信息头：BMP 位图信息头数据用于说明位图的尺寸等信息。

typedef struct tagBITMAPINFOHEADER{
DWORD biSize; // 本结构所占用字节数
LONGbiWidth; // 位图的宽度，以像素为单位
LONGbiHeight; // 位图的高度，以像素为单位
WORD biPlanes; // 目标设备的级别，必须为 1
WORD biBitCount// 每个像素所需的位数，必须是 1(双色),4(16 色)，8(256 色)或 24(真彩色)之一
DWORD biCompression; // 位图压缩类型，必须是 0(不压缩),1(BI_RLE8 压缩类型)或 2(BI_RLE4 压缩类型)之一
DWORD biSizeImage; // 位图的大小，以字节为单位
LONG biXPelsPerMeter; // 位图水平分辨率，每米像素数
LONG biYPelsPerMeter; // 位图垂直分辨率，每米像素数
DWORD biClrUsed;// 位图实际使用的颜色表中的颜色数
DWORD biClrImportant;// 位图显示过程中重要的颜色数
} BITMAPINFOHEADER;

3、 颜色表：颜色表用于说明位图中的颜色，它有若干个表项，每一个表项是一个 RGBQUAD 类型的结构，定义一种颜色。

typedef struct tagRGBQUAD {
BYTE rgbBlue;// 蓝色的亮度(值范围为 0-255)
BYTE rgbGreen; // 绿色的亮度(值范围为 0-255)
BYTE rgbRed; // 红色的亮度(值范围为 0-255)
BYTE rgbReserved;// 保留，必须为 0
} RGBQUAD;

颜色表中 RGBQUAD 结构数据的个数有 biBitCount 来确定:
当 biBitCount=1,4,8 时，分别有 2,16,256 个表项;
当 biBitCount=24 时，没有颜色表项。
位图信息头和颜色表组成位图信息，BITMAPINFO 结构定义如下:

typedef struct tagBITMAPINFO {
BITMAPINFOHEADER bmiHeader; // 位图信息头
RGBQUAD bmiColors[1]; // 颜色表
} BITMAPINFO;

4、 位图数据：位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右,扫描行之间是从下到上。位图的一个像素值所占的字节数:
当 biBitCount=1 时，8 个像素占 1 个字节;
当 biBitCount=4 时，2 个像素占 1 个字节;
当 biBitCount=8 时，1 个像素占 1 个字节;
当 biBitCount=24 时,1 个像素占 3 个字节;
Windows 规定一个扫描行所占的字节数必须是 4 的倍数(即以 long 为单位),不足的以 0 填充,
一个扫描行所占的字节数计算方法:

DataSizePerLine= (biWidth* biBitCount+31)/8; // 一个扫描行所占的字节数
DataSizePerLine= DataSizePerLine/4*4; // 字节数必须是 4 的倍数

位图数据的大小(不压缩情况下):

DataSize= DataSizePerLine* biHeight;

tyw藏书

二、BMP 文件分析

1、 工具软件：Hex Workshop 或 UltraEdit

2、 分析：首先请注意所有的数值在存储上都是按"高位放高位、低位放低位的原则"，如 12345678h 放在存储器中就是 7856 3412）。下图是一张图 16 进制数据，以此为例进行分析。在分析中为了简化叙述，以一个字（两个字节为单位，如 424D 就是一个字）为序号单位进行，"h"表示是 16 进制数。

424D 4690 0000 0000 0000 4600 0000 2800
0000 8000 0000 9000 0000 0100 1000 0300
0000 0090 0000 A00F 0000 A00F 0000 0000
0000 0000 0000 00F8 0000 E007 0000 1F00
0000 0000 0000 02F1 84F1 04F1 84F1 84F1
06F2 84F1 06F2 04F2 86F2 06F2 86F2 86F2

1：图像文件头。424Dh='BM'，表示是 Windows 支持的 BMP 格式。

2-3：整个文件大小。4690 0000，为 00009046h=36934。

4-5：保留，必须设置为 0。

6-7：从文件开始到位图数据之间的偏移量。4600 0000，为 00000046h=70，上面的文件头就是 35 字=70 字节。

8-9：位图图信息头长度。

10-11：位图宽度，以像素为单位。8000 0000，为 00000080h=128。

12-13：位图高度，以像素为单位。9000 0000，为 00000090h=144。

14：位图的位面数，该值总是 1。0100，为 0001h=1。

15：每个像素的位数。有 1（单色），4（16 色），8（256 色），16（64K 色，高彩色），24（16M 色，真彩色），32（4096M 色，增强型真彩色）。T408 支持的是 16 位格式。1000 为 0010h=16。

16-17：压缩说明：有 0（不压缩），1（RLE 8，8 位 RLE 压缩），2（RLE 4，4 位 RLE 压缩），3（Bitfields，位域存放）。RLE 简单地说是采用像素数+像素值的方式进行压缩。T408 采用的是位域存放方式，用两个字节表示一个像素，位域分配为 r5b6g5。图中 0300 0000 为 00000003h=3。

18-19：用字节数表示的位图数据的大小，该数必须是 4 的倍数，数值上等于位图宽度×位图高度×每个像素位数。0090 0000 为 00009000h=80×90×2h=36864。

20-21：用象素/米表示的水平分辨率。A00F 0000 为 0000 0FA0h=4000。

22-23：用象素/米表示的垂直分辨率。A00F 0000 为 0000 0FA0h=4000。

2：位图使用的颜色索引数。设为 0 的话，则说明使用所有调色板项。

26-27：对图象显示有重要影响的颜色索引的数目。如果是 0，表示都重要。

28-35：彩色板规范。对于调色板中的每个表项，用下述方法来描述 RGB 的值：

1 字节用于蓝色分量

1 字节用于绿色分量

1 字节用于红色分量

1 字节用于填充符(设置为 0)

对于 24-位真彩色图像就不使用彩色表，因为位图中的 RGB 值就代表了每个象素的颜色。但是 16 位 r5g6b5 位域彩色图像需要彩色表，看前面的图，与上面的解释不太对得上，应以下面的解释为准。

图中彩色板为 00F8 0000 E007 0000 1F00 0000 0000 0000，其中：

00FB 0000 为 FB00h=1111100000000000（二进制），是红色分量的掩码。

E007 0000 为 07E0h=0000011111100000（二进制），是绿色分量的掩码。

1F00 0000 为 001Fh=0000000000011111（二进制），是红色分量的掩码。

0000 0000 总设置为 0。

将掩码跟像素值进行"与"运算再进行移位操作就可以得到各色分量值。看看掩码，就可以明白事实上在每个像素值的两个字节 16 位中，按从高到低取 5、6、5 位分别就是 r、g、b 分量值。取出分量值后把 r、g、b 值分别乘以 8、4、8 就可以补齐第个分量为一个字节，再把这三个字节按 rgb 组合，放入存储器（同样要反序），就可以转换为 24 位标准 BMP 格式了

RTX操作系统实验

RTX例程在 \BHS-STM32 例程\高级例程-(实战篇)\RTX操作系统文件夹里，这个文件夹的例程是基于MDK自带的操作系统的应用，RTX官方文档在安装路径的HLP文件下《rlarm.chm》做了详细介绍，光盘里也有个中文版的，要使用RTX的朋友请先先阅读该文档。

RTX基本知识

RTX简介：

RTX 内核是一个实时操作系统(RTOS)，可以同时运行多函数或是任务。在嵌入式运用中这是非常有用的。当然也可以不用 RTOS 开发实时程序不需要，例如通过循环执行一个或多个任务。但有像 RTX 这样的实时操作系统，可以解决众多的调度、维护、定时等问题。

RTOS 可以自由地调度系统资源，比如 CPU 和内存，并且提供一种任务间通信机制。 RTX 内核是一个强大的实时操作系统，可以很容易地使用和运行基于 ARM7TDMI、ARM9 或是 Cortex-M3 CPU 内核的微控制器。

RTX 程序使用标准的 C 结构编写，运用 RealView® 编译器进行编译。RTX.H 头文件定义了 RTX 函数以及宏，可以让轻松地声明任务并达到 RTOS 所有特性。

技术规范：

描述	RX Kernel
支持的进程数	最多 256
支持的信箱数	无限制
支持的信号量数	无限制
支持的互斥量数	无限制
支持的信号量数	每个进程 16 个
支持的用户定时器数	无限制
RAM 要求	最少 500 字节
代码要求	小于 5 K 字节
硬件要求	一个或多个片上时钟可用

用户进程优先级	1 - 255
进程切换的时间	小于 5µsec @60MHz, 0 ws.
中断停止时间	小于 1.8 µsec @60MHz, 0 ws

注意：

无限制是说不受 RTX 核操作系统的限制，但是要受到系统内存资源的限制；

RTX 核的默认配置是：10 个任务、10 个用户定时器、禁止栈的检查；

这里的“信号”就是事件的意思；

RAM 的限制是由并发执行的进程数来决定的。

时序规格

函数	时间性能
初始化系统(os_sys_init)，启动进程	36.3
创建定义的进程，没有进程切换	12.9
创建定义的进程，切换进程	14.6
撤消进程(通过 os_tsk_delete)	5.9
进程切换(通过 os_tsk_delete_self)	8.8
进程切换(通过 os_tsk_pass)	4.6
进程切换(upon set event)	7.3
进程切换(upon sent semaphore)	5.5
进程切换(upon sent message)	6.1
设置时间(没有进程切换)	2.5
发送信号量 (没有进程切换)	1.9
发送消息(没有进程切换)	2.8
获得进程标识符 (os_tsk_self)	1.0
IRQ 中断服务子程序的中断响应时间	0.4
IRQ 中断服务子程序的最大等待时间(lockout)	2.2
IRQ 中断服务子程序的最大中断延迟 (response + lockout)	2.6

注意：

这里的时间数据是在 LPC21xx 上执行的结果，系统时钟为 60MHz，单位是 µs，代码在处理器的内部 Flash 中执行；

这些时间数据由 µVision3 调试器和 SARM.DLL 软件仿真器(1.50e 版本)测得的。

中断服务子程序及调用中断服务的程序都由 RealView 编译器编译。在不使用 RTX 核时，RVCT 快速中断程序也有同样的中断响应时间。

进程通信

RTX 提供了几种不同的进程通信方法：

事件标志

事件标志是实现进程同步的主要方法，每个进程有 16 个事件标识可供使用，所以最多能等待 16 个不同的

事件。也可以同时等待多个事件标志，这种情况下，如果这些事件标志是与的关系，那么这些事件标志必须都被置位后该进程才能继续运行；如果这些事件标志是或的关系，那么这些事件标志中的一个或几个被置位后该进程就可以继续运行。

事件标志也可被 ARM 中断功能置位。在这种机制下，通过使用 ARM 中断函数设置任务等待的标志，可以使异步的外部事件和 RTX 核的任务同步。

信号量

在多任务实时操作系统中，需要特别的方法访问共享资源。否则，这些任务对共享资源的同时访问可能会导致数据的不一致或外设的错误操作。

解决访问临界资源问题的主要方法是信号量。信号量是包含了虚拟标志的软件对象。内核将标志给第一个请求的任务。在任务将其返回给信号量之前，没有其他的任务可以获取这个标志。只有拥有标志的任务才能访问公共资源，这就阻止了其他的任务访问和扰乱公共资源。

当信号量的标志不可用时，访问它的进程将被挂起，一旦标志被返回，这个进程就会被唤醒。为了解决错误的等待条件，必须引入超时机制。

互斥量

互斥量是解决进程同步问题的另一种方法。它们用作对临界区的访问控制，只有拥有互斥量的进程才能访问临界区，其他试图访问临界区的进程将被阻塞。

信箱

有时进程之间需要交换消息，这在网络中是很常见的，例如 TCP-IP、UDP、ISDN 等。

消息就是包含协议消息或帧的内存块的指针，这样的内存块可以动态的分配和提供给用户。为了防止内存泄漏，用户有责任正确地分配和回收内存块。

如果接收进程访问信箱中的消息不存在，它将被挂起，直到该消息被发送进程发送到信箱中，该被挂起的接收进程才会被唤醒。

RTX基础配置

在使用 RTX 的嵌入式应用程序中，必须对 RTX 内核进行基础配置。在文件夹\Keil\ARM\Startup 中可以找到 RTX_Config.c ，它包含了所有的配置设置，并且可随着不同的 ARM 设备而不同。在 RTX_Config.c 中的配置选项可以：

指定当前运行任务的数目；

指定使用用户自定堆栈任务的数目；

指定为每个任务分配堆栈的大小；

开启或是禁止堆栈校核；

指定 CPU 定时器作为系统定时器；

为选中的定时器指定输入的时钟频率；；

指定定时器节拍间隔；

开启或是禁止轮转任务调度；

为轮转任务调度指定时间片；

定义空闲任务操作；

指定用户定时器的数目；

为用户定时器回调函数指定代码；

在 RL-RTX 库中没有默认的配置，因此，必须为每一个工程添加 RTX_Config.c 配置文件。

为了适应 RTX 内核的特性，必须修改 RTX_Config.c 中的配置。

RTX详细配置

任务数量定义

OS_TASKCNT 标识同时处于活跃状态任务的最大数目，这包括了除了停止外的所有状态（运行，等待或是就绪）。

RTX内核利用该信息来为任务控制的变量预留存储池。在确保创建和运行的任务数不会超过 OS_TASKCNT 的前提下，数目可以高于或是低于运用程序中定义的任务数目（一个任务可在多个多个实例中运行）。

lyw藏书

#define OS_TASKCNT　　6

OS_PRIVCNT 标识带有用户提供栈的任务数目。

默认情况下，RTX 内核给每个任务分配一个固定大小的堆栈。然而，任务对堆栈的需求是非常广泛的。例如，如果一个任务的局部变量包含大块的缓冲区，队列或是复杂的结构，那么任务就需要更多的堆栈。如果有像这样的任务，需要的比分配的更多堆栈，就有可能造成越界写入到相邻的任务了。这是因为任务的固定大小堆栈是公共系统堆的一部分，而且是相互比邻的。这样就导致 RTX 内核的故障，并且很可能导致系统瘫痪。显而易见的解决办法是增加固定的堆栈大小。然而，这样就给其他每一个任务都增加了的大小，也许有些并不需要。为了避免这样的资源浪费，一个好的解决办法是，给需要额外多堆栈的任务分配一个单独的用户设定堆栈。

在这种情况下，用户设定就意味着任务创建时，用户自己设定任务需要的堆栈存储空间，而不是由内核自动分配。RTX 内核采用 OS_PRIVCNT 使存储空间的利用最优化。内核不会保留堆栈空间给用户设定堆栈大小的任务。

#define OS_PRIVCNT　　0

注意:

加上 OS_TASKCNT 用户任务,系统创建了两个系统任务 os_clock_demon 和 os_idle_demon. 这两个任务经常是RTX内核所必需的。当前运行的任务总数是OS_TASKCNT+2 (用户任务的数量加上两个系统任务)。

堆栈大小

一个特殊任务的堆栈用法 依赖于局部自动变量数量和子程序的数量。中断函数不使用中断任务的堆栈。OS_STKSIZE 标识分配给每个任务的 RAM 数量。堆栈大小用 U32（无符号整型）定义。同时，系统转换定义的大小，并以字节为单位显示。以下的堆栈大小定义为 400 字节。

#define OS_STKSIZE　　100

在所有的上下任务的转换中，RTX 内核在堆栈中存储所有的 ARM 寄存器。所有任务内容的保存需要 64K 的堆栈空间。

堆栈检查

由于很多嵌套子程序的调用或是大量的自动变量的广泛使用，有可能导致堆栈资源耗尽。

如果堆栈检查激活，内核可以发现堆栈耗尽问题并且运行os_stk_overflow() 堆栈出错函数。运用程序将会在堆栈出错函数内挂起一个无止境的循环。变量 task_id 将会保存出现堆栈问题的任务ID。可以通过活动任务调试对话框来查询任务名字。

解决这个问题的办法是通过在 配置文件中为所有的任务增加堆栈大小。如果只有一个任务需要大堆栈并且RAM有限，可以通过 创建带有用户设定堆栈空间的任务。

OS_STKCHECK 激活堆栈检查算法，1 表示激活，0 表示禁止，默认是激活。

#define OS_STKCHECK　　1

激活堆栈检查会轻微地降低内核的性能，因为当每进行任务切换时，内核需要运行额外的代码进行堆栈检查。

硬件时钟

以下的 #defines 说明 RTX 内核的硬件时钟是如何被设置的：

S_TIMER 指定芯片时钟作为实时系统的基本时钟。其发送了一个周期的中断来唤醒时间纪录系统任务。用户可以选择使用哪个计时器来实现这个目的，0 表示计时器 0，而 1 则表示选择计时器 1。

#define OS_TIMER　　1

OS_CLOCK 为选中的时钟指定输入时钟频率。值的计算是： f(xtal) / VPBDIV.在 CPU 时钟为 60 MHz 并且 VPBDIV = 4 时是 15 MHz。

#define OS_CLOCK　　15000000

OS_TICK 指定时钟脉冲间隔，单位是 μsec。建议值是 1000 到 100000。产生的间隔为 1 ms 到 100 ms，默认的设置为 10ms。.

#define OS_CLOCK　　10000

多任务轮转

以下的 #define 指定RTX内核的 多任务轮转如何被设置：

OS_ROBIN 激活多任务轮转，1 表示运行，0 表示禁止，默认是激活。

#define OS_ROBIN　　1

OS_ROBINTOUT 标识轮转时间片。这是分配给当前运行任务的时间片。当时间片用完，当前运行任务被中止，下一个就绪任务被重新开始。OS_ROBINTOUT 被用来表示系统的节拍数。.

#define OS_ROBINTOUT　5

空闲任务

当没有任务就绪，RTX内核执行 idle task 。空闲任务是一个简单的循环，不执行任何操作，只等待时钟中断来选择就绪的任务。

os_idle_demon() 标识 IDLE 指令是否在空闲任务中运行，默认设置是 OFF，禁止 CPU 进入空闲任务。可以在这儿添加代码，当没有任务就绪时就可以执行这段代码。

以下是包含在库中的 os_idle_demon() 默认函数代码的一个示例：

```
/*--------------------------- os_idle_demon -------------------------------*/

void os_idle_demon (void) __task {
  /* The idle demon is a system task. It is running when no other task is    */
  /* ready to run (idle situation). It must not terminate. Therefore it      */
  /* should contain at least an endless loop.                                */

  for (;;) {
  /* HERE: include here optional user code to be executed when no task runs.*/
  }
} /* end of os_idle_demon */
```

注意

如果使用 ULINK/JLINK 进行调试，就不能使用 IDLE 模式。在一些 ARM 设备中，空闲模式可以阻塞 JTAG 接口。

用户定时器

在运行时间时，可以创建或是结束 用户定时器 ，但必须指定运行的用户定时器的最大值以及 os_tmr_call() 函数的代码。

OS_TIMERCNT 创建用户定时器时将指定定时数目。如果用户定时器没有被采用，设置其值为 0。这个消息被 RTX 内核用来为时钟控制块保留存储空间。

#define OS_TIMERCNT　　5

当用户定时器终止时，回调函数os_tmr_call()被调用，其作为一个空函数由 RTX_Config.c 配置文件提供，使用时必须依据需要进行修改。

当定时器被创建时，info 作为参数传递给 os_tmr_create()函数。

```
/*--------------------------- os_tmr_call ---------------------------------*/

void os_tmr_call (U16 info) {
  /* This function is called when the user timer has expired.                */
  /* Parameter "info" is the parameter defined when the timer was created.   */
```

```
    /* HERE: include here optional user code to be executed on timeout.          */
    info = info;
} /* end of os_tmr_call */
```

tyw藏书

配置宏

所有硬件相关的配置选项都被用配置宏来描述。其中，一些仅被用于简化代码(如：OS_TID_和 OS_TIM_)。这些宏并不都在所有的配置文件中使用。使用配置宏可以轻松地定制 ARM 设备提供的不同外围设备定时器的配置文件。

以下的配置宏包括：(以 Philips LPC21xx 设备定时器 0 为例)

OS_TRV： 该宏定义了外围计时器的重装值。外围计时器计算出一个重装值，当达到 0 时，就产生一个中断。重装值被用于计算一个要求的间隔长度（比如 10ms）。

```
#define OS_TRV          ((U32)(((double)OS_CLOCK*(double)OS_TICK)/1E6)-1)
```

OS_TVAL：该宏被用于为一个 count-up 计时器读取当前的计时器值。RTX 内核用其来确定一个中断是否是周期计时中断，还是软件强制中断。.

```
#define OS_TVAL         T0TC                    /* Timer Value */
```

对于一个减法数计时器，用户必须转换其返回值。以下是一个 16 位减法数计时器的例子：:

```
#define OS_TVAL         (0xFFFF - T0VAL)    /* Timer Value */
```

OS_TOVF ：定于一个计时器溢出标志。RTX 内核将其同 OS_TVAL 宏一起使用，用来区别周期计时中断和强制计时中断。

```
#define OS_TOVF         (T0IR & 1)              /* Overflow Flag */
```

OS_TREL() ：当计时器溢出的 时候，该宏定义一个代码系列来重装周期计时器。而当一个周期计时器有自动重装功能的时候，该宏不起作用。

```
#define OS_TREL()   ;                           /* Timer Reload */
```

OS_TFIRQ() ：定义一个代码系列来执行强制计时中断。如果周期计时器不允许手动设置溢出标志时，则必须是个软件触发中断。如果可以进行手动设置，该宏必须设定一个外围计时器溢出标志，用来产生计时中断。

```
#define OS_TFIRQ()    VICSoftInt     |= OS_TIM_;    /* Force Interrupt */
```

OS_TIACK() ：应答来自计时中断函数产生的中断，释放计时器中断逻辑。.

```
#define OS_TIACK()    T0IR = 1;                 /* Interrupt Ack */   \
                      VICSoftIntClr = OS_TIM_;                        \
                      VICVectAddr     = 0;
```

OS_TINIT() ：该宏用于初始化外围计时器/计数器，设置计时模式并设定计时器重装功能。时钟中断也可被外设时钟中断激活。代码在 os_sys_init() 函数中执行。

```
#define OS_TINIT()    T0MR0 = OS_TRV;         /* Initialization */   \
                      T0MCR = 3;                                      \
                      T0TCR = 1;                                      \
                      VICDefVectAddr = (U32)os_def_interrupt;     \
                      VICVectAddr15    = (U32)os_clock_interrupt; \
                      VICVectCntl15    = 0x20 | OS_TID_;
```

OS_LOCK() ：该宏禁止计时中断，被用于避免中断系统任务调度，包括周期计时中断和强制中断都被屏蔽。代码在 tsk_lock() 函数中执行。

```
#define OS_LOCK()     VICIntEnClr = OS_TIM_;      /* Task Lock */
```

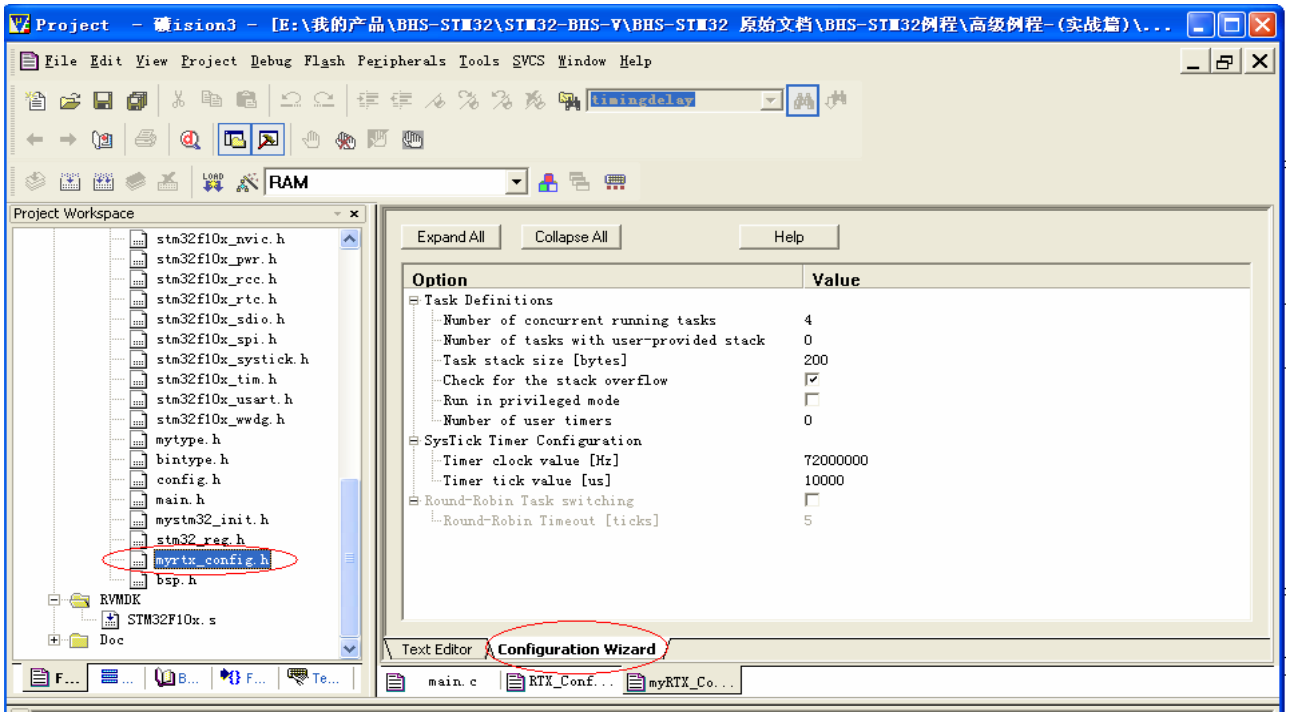
OS_UNLOCK() ：该宏允许计时中断，包括周期计时中断和强制中断。代码在 tsk_unlock() 函数中执行。

```
#define OS_UNLOCK() VICIntEnable |= OS_TIM_;    /* Task Unlock */
```

tyw藏书

BHS-STM32 实验 43-RTX最简单点灯

此例子是 RTX 操作系统简单的演示例程，首先打开项目文件，打开 RTX 的配置文件 myrtx_config.h



Number of concurrent running tasks　　任务数量

Number of tasks with user-provided stack　　使用自定义栈任务数量

Task stack size　　默认的任务栈空间大小

下面介绍使用的 RTX 函数：

使用默认任务栈空间的任务创建函数 `os_tsk_create`

```
OS_TID os_tsk_create (
  void (*task)(void),  /* Task to create 建立任务 */
  U8  priority );    /* Task priority (1-254)任务优先权*/
```

描述：

os_tsk_creat 函数创建由参数任务函数指针*task* 指定的任务，并将任务添加准备好队列中，新的任务会被动态分配一个任务识别号(TID)。

参数 *priority* 指定任务的优先级，默认的任务优先权是 1。0 为闲置的任务保留的，如果指定一个任务的优先权为 0，则自动用 1 代替，值 255 也保留。如果新任务比当前任务具有更高的优先权，任务跳立即切换执行新的任务。

返回值：

os_tsk_creat 函数返回新任务的任务识别号(TID)。如果函数失效，例如无效参数，它返回 0。

注意：

RTX 核使用默认的栈的大小，在 **rtx_config.c** 中已定义。

优先权值 255 代表最重要的任务。

使用自定义栈空间务的任务创建函数 os_tsk_create_user

```
OS_TID os_tsk_create_user(
  void (*task)(void),  /* Task to create */
  U8  priority,      /* Task priority (1-254) */
  void* stk,           /* Pointer to the task's stack*/
  U16   size );       /* Number of bytes in the stack*/
```

tyw藏书

描述：

os_tsk_create_user 函数建立由任务函数指针 *task* 指定的任务，并将其添加到就绪队列中。函数动态地给该任务分配一个新的 TID。该函数可为任务提供单独的栈，当任务需要一个更大的栈来存储局部变量时，这非常有用。

参数 *priority* 指定了任务的优先级，默认的任务优先权是 1。0 为闲置任务的优先级，保留。如果设置任务的优先级为 0，则会自动被 1 代替，优先级 255 也保留。如果新任务比当前任务具有更高的优先权，立即切换执行新的任务。

参数 *Stk* 是为任务的栈的内存块的指针，参数 *size* 指定了栈的字节数。

返回值：

os_tsk_create_user 函数返回新任务的任务识别号(TID)；如果函数失败，例如无效的参数，它就返回 0。

注意：

栈必须以 8 字节为边界对齐，以 U64 类型声明(无符号的长整型)。

默认栈的大小在 rtx_config.c 中定义。

系统延时 os_dly_wait

```
void os_dly_wait (
   U16 delay_time );    /* Length of time to pause 预约时间长度*/
```

描述：

os_dly_wait 函数暂停调用任务。 delay_time 具体规定停顿的时间长度，它由 system_ticks 加以衡量。可以设置 delay_time 从 1 至 0xfffe 的任何值。

返回值：

os_dly_wait 函数不返回任何值

注意：不能单一任务混杂调用函数 os_itv_wait ( )和 os_dly_wait ( )

停止并删除当前任务 os_tsk_delete_self

```
void os_tsk_delete_self (void);
```

描述：

os _tsk_delete_self 函数停止并删除当前任务。程序继续执行下一个在就绪队列中具有最高优先权的任务。

返回值：

os_tsk_delete_self 函数不返回。程序继续执行下一个在就绪队列中具有最高优先权的任务。

注意：

删除一个任务就释放了所有分配给该任务的动态资源。

详细代码如下：

```
//LED 任务
__task void Led1(void)
{

    while(1)
```

```
    {
        //LED1 点亮
        PC8=0;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED1 熄灭
        PC8=1;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED2 点亮
        PC9=0;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED2 熄灭
        PC9=1;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED3 点亮
        PC10=0;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED3 熄灭
        PC10=1;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED4 点亮
        PC11=0;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms

        //LED4 熄灭
        PC11=1;
        //延时 100ms
        os_dly_wait(100/OS_TIME);//100ms
    }
}

__task void init (void)
{
```

tyw藏书

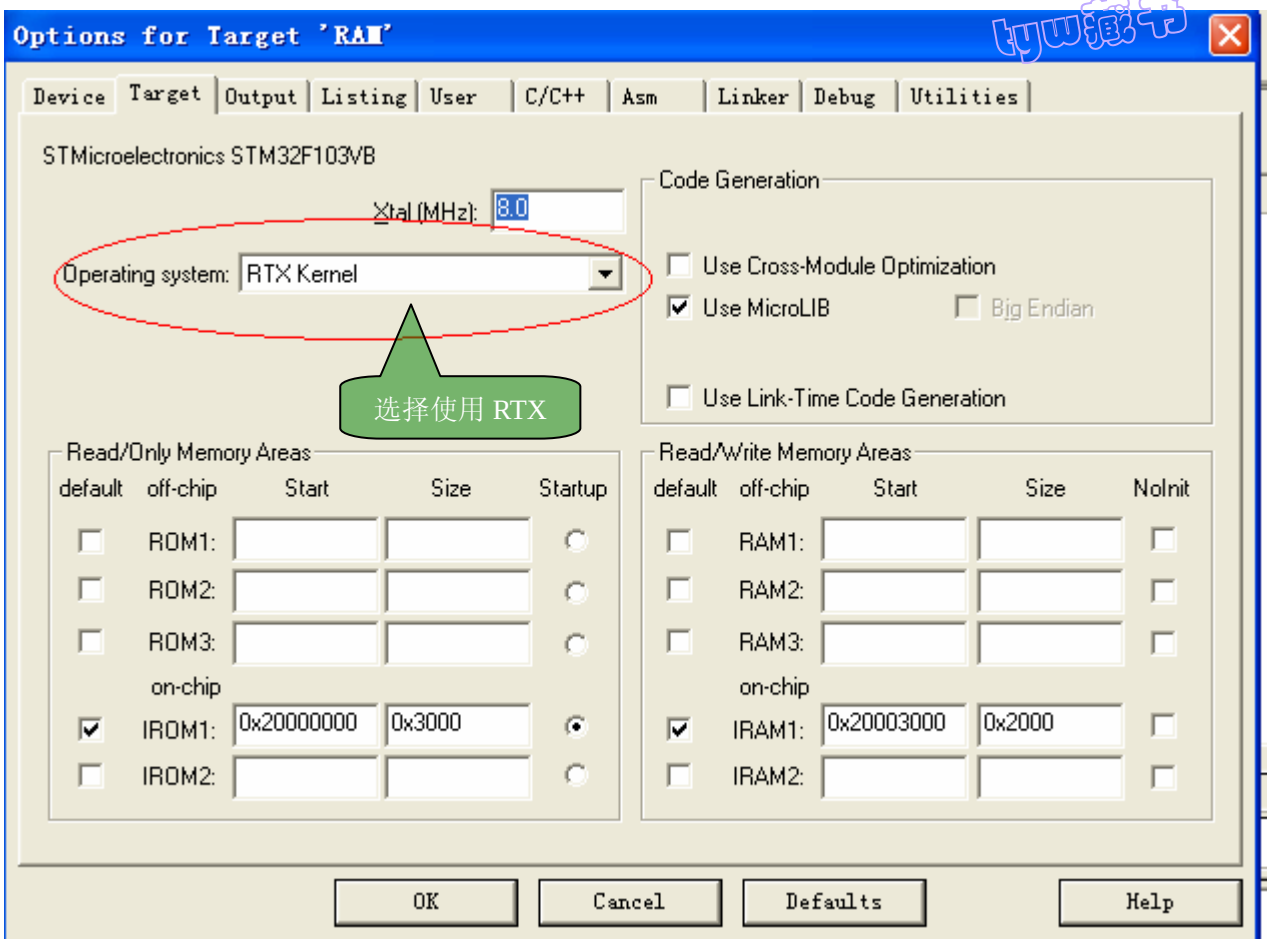
```
    //创建 LED 任务
    os_tsk_create(Led1, 0);

    //初始化任务删除
    os_tsk_delete_self ();
}

/*******************************************************************************
* Function Name   : main
* Description     : Main program.
* Input           : None
* Output          : None
* Return          : None
*******************************************************************************/
int main(void)
{//uint8 i=0;
#ifdef DEBUG
    debug();
#endif

//ReStart:
    /* System Clocks Configuration */
    //初始化 RCC 时钟
    RCC_Configuration();

    //初始化 GPIO
    GPIO_Configuration();

    /* NVIC configuration */
    //初始化中断
    NVIC_Configuration();

    //RTX 操作系统初始化
    os_sys_init (init);

}
```

总结：

本例中实际只创建了一个用户任务 Led1

从这个简单的例子来看，我们是不是觉得对于实际应用来讲，操作系统也不是很复杂呢

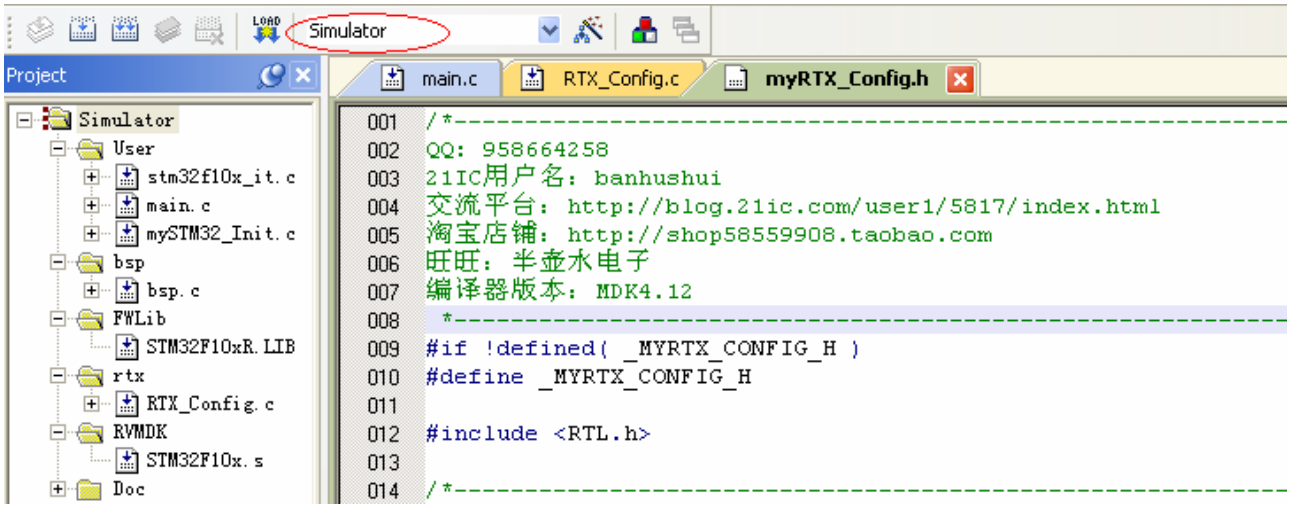
使用 RTX 的程序必须设置选择 RTX Kernel 选项才能编译通过。

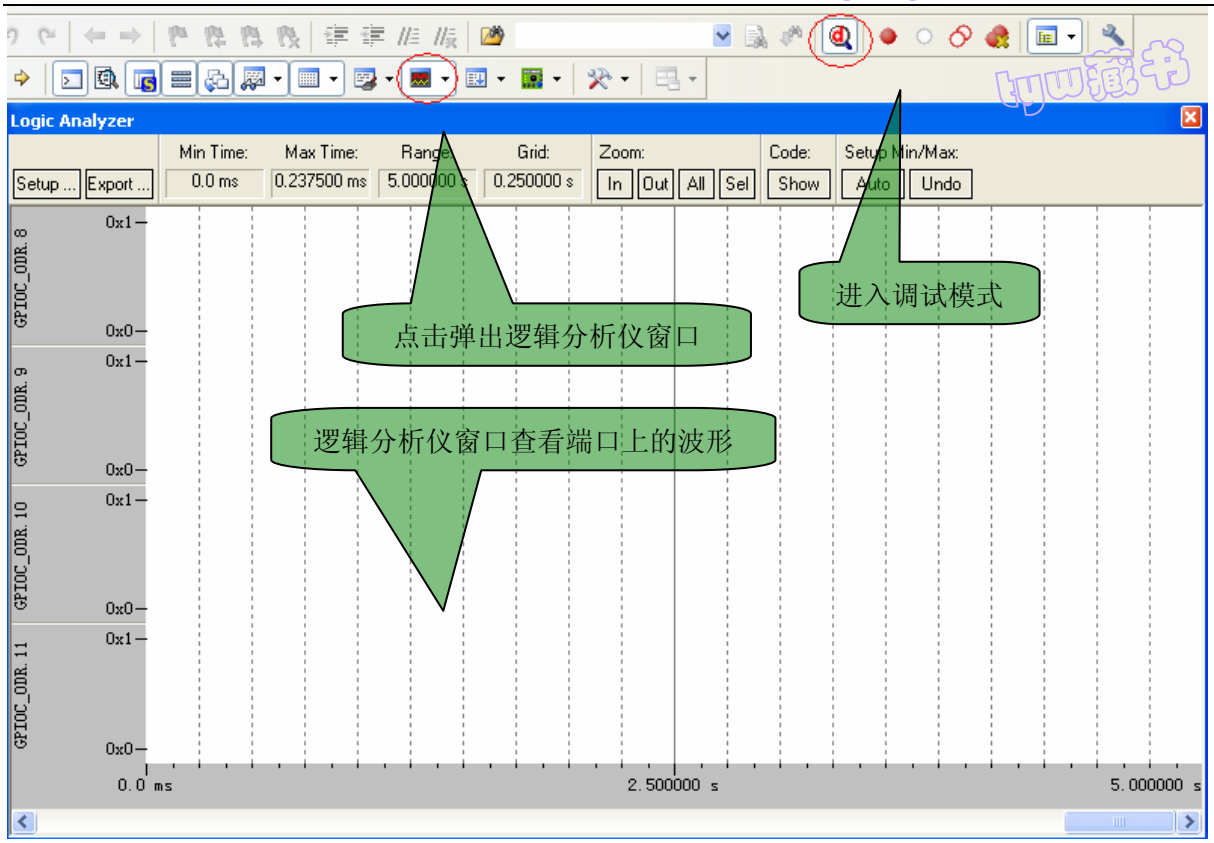


RTX 操作系统详细说明请参考【\资料文档\MDK 及 RTX 操作系统资料\rlarm.chm】,该文档是官方文档

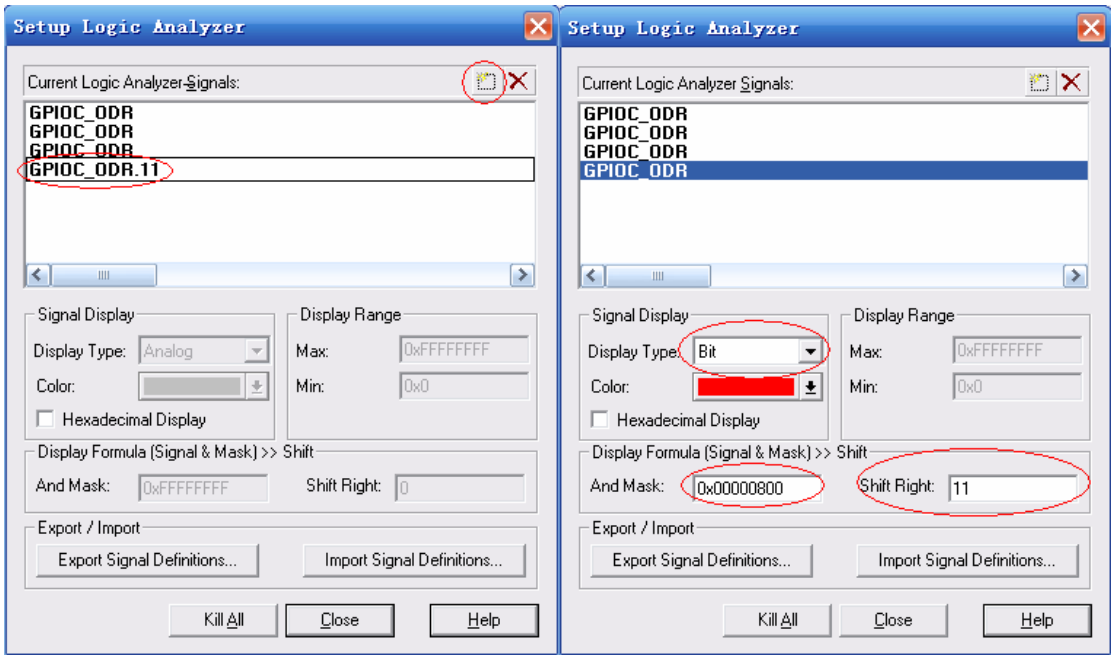
软件仿真：

选择软件仿真

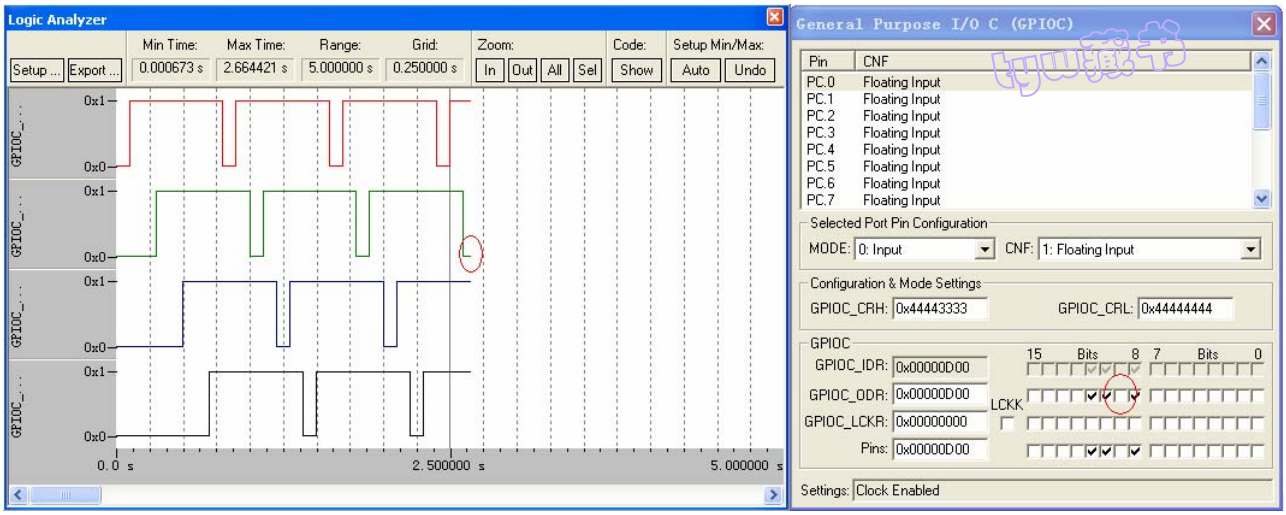




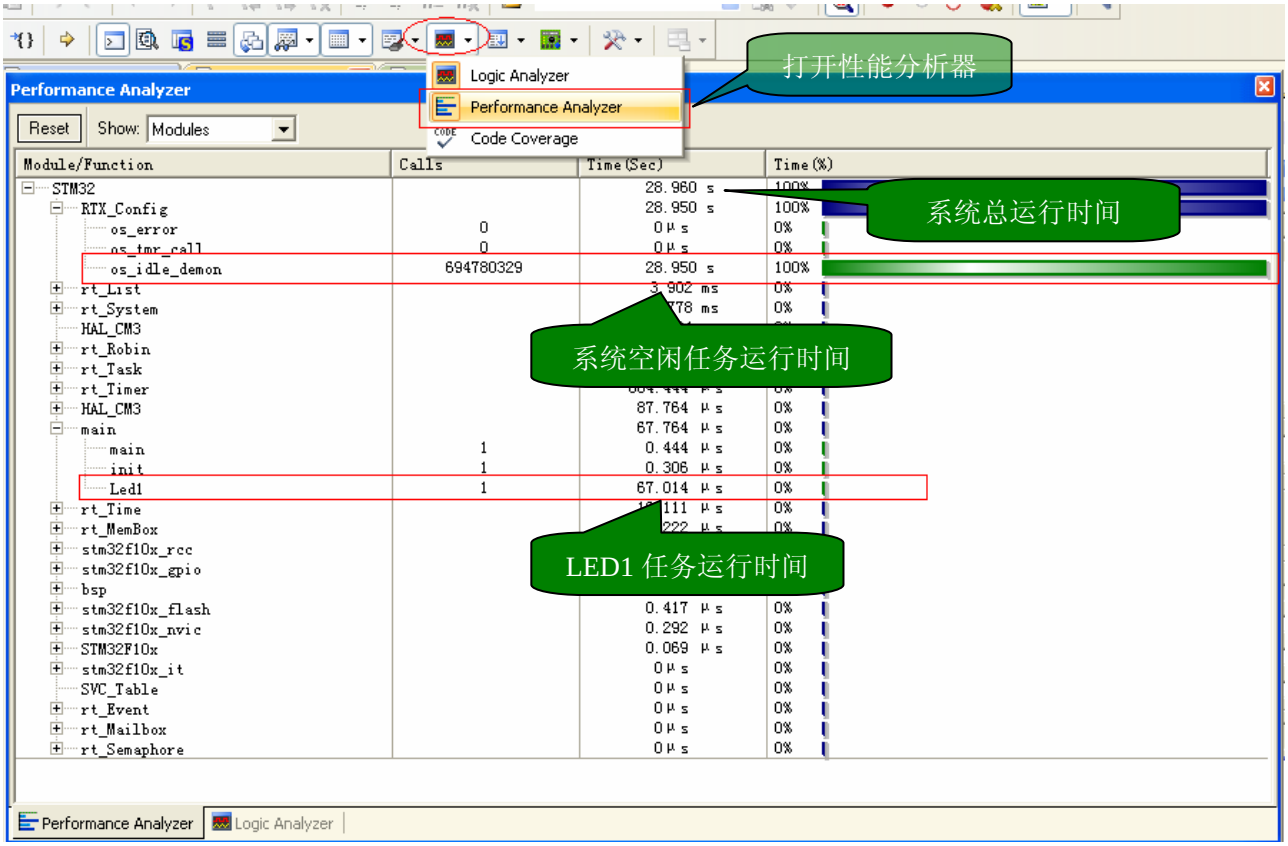
下面添加要查看的端口



软件仿真波形

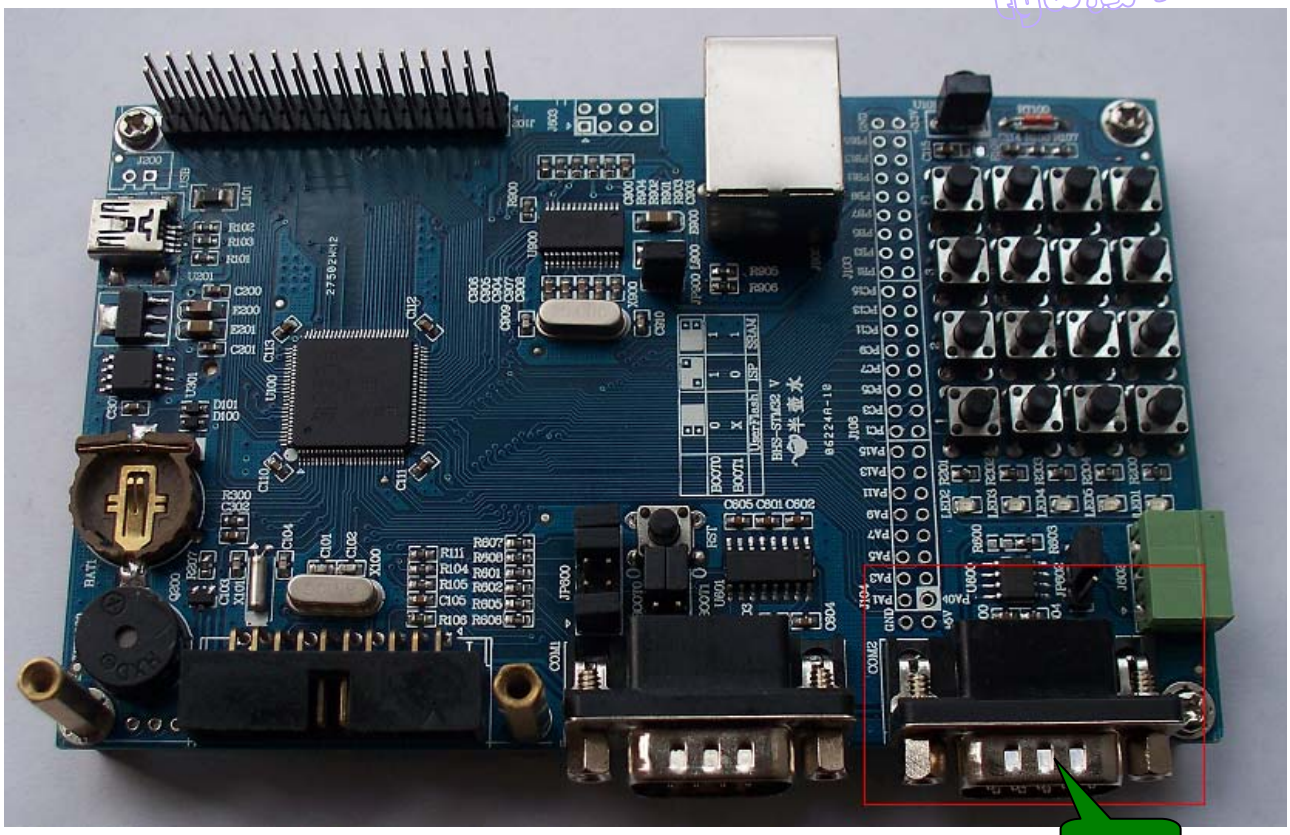


下面我们打开【性能分析器】，看看使用 RTX 操作系统的效率



这个例子中我们只创建了一个用户任务，由上面的性能分析器可以看出我们自己的用户任务占用 CPU 时间是非常低的。

BHS-STM32 实验 44-USART一个完整通信协议(串口 2)



通信协议【资料文档\BHS-STM32 文档】《BHS-STM32 IAP通讯协议V1.0.pdf》
本实验通信串口是使用的串口 2，功能同实验 35 完全一样，只是串口不同而已。

```
static U64 COM_stack[800/8];



//LED任务
__task void Led1(void)
{
    while(1)
    {
        PC8=0; PC9=0; PC10=0; PC11=0;
        os_dly_wait(100/OS_TIME);//100ms
        PC8=1; PC9=1; PC10=1; PC11=1;
        os_dly_wait(300/OS_TIME);//100ms
    }
}

//串口初始化
void Init_PCCOM(void)
{
    //本机地址
    SetLocalAddr(1);
```

tyw藏书

```
    //串口协议初始化
    InitCom();

    //初始化串口 2
    USART2_InitConfig(57600);


    PC_RS485Receive_Enable();
}

/*********************************************************************************************
*  函数原型:PC_COMTask
*  函数功能:串口任务
*  输入参数:
*  输出参数:
*  函数说明:
*********************************************************************************************/
__task void PC_COM_Task(void) //
{uint16     cmd;


        while(1)
        {
            //等待事件
            os_evt_wait_or(0x0001, 0xffff);

            if( testReceiveOver()==0 ) //有数据
            {

                cmd=getCmd();
                if( cmd !=0 )
                    PC_ComdCpp(cmd);

                init_Receive();//处理完后清空接受缓冲重新开始接收数据
            }
        }
}

__task void init (void)
{
   Init_PCCOM();

   //创建LED任务
   os_tsk_create(Led1, 0);
```

```
    //创建串口任务
    rd_task=os_tsk_create_user (PC_COM_Task, 0, &COM_stack, sizeof(COM_stack));

    os_tsk_delete_self ();
}



/*******************************************************************************
* Function Name  : main
* Description    : Main program.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
int main(void)
{//uint8 i=0;
#ifdef DEBUG
  debug();
#endif

//ReStart:
  /* System Clocks Configuration */
  //初始化RCC时钟
  RCC_Configuration();

  //初始化GPIO
  GPIO_Configuration();

  /* NVIC configuration */
  //初始化中断
  NVIC_Configuration();

  //RTX操作系统初始化
  os_sys_init (init);

}
```

实验总结：

本例创建了 2 个用户任务

其中LED任务是默认堆栈任务，所以创建函数使用os_tsk_create(Led1, 0);

串口任务是自定义堆栈任务，所以创建函数使用os_tsk_create_user

实验 43 已经介绍了这 2 个函数，这就不做介绍了。

下面我们介绍本例使用的 2 个RTX新函数os_evt_wait_or，isr_evt_set

事件等待函数os_evt_wait_or

```
OS_RESULT os_evt_wait_or (
   U16 wait_flags,    /* Bit pattern of events to wait for 事件等待的位模式 */
   U16 timeout );     /* Length of time to wait for event 事件等待的时间长度 */
```

描述

os_evt_wait_or函数能等待在参数 wait_flags 中被指定发生的所有的事件。函数等等在参数wait_flags 中相应位为 1 的事件。函数能访问多达 16 个不同的事件。

能用*timeout*设置预约时间， 预约时间之后即使没有一个事件发生，函数也必须返回。可使用除了 0xFFFF 之外的预约时间，如果设置*timeout*为 0xFFFF，则表示一个不确定的预约时间。 预约时间由系统时间衡量。当至少一个列在*wait_flags* 的事件发生或预约时间到时，**os_evt_wait_or函数**返回。

返回值

OS_R_EVT 至少有一个列在 **wait_flags** 中的标志已被设置。

OS_R_TMO预约时间到。

注意

每一事件都有其自己的 16 位的等待标志。

事件设置函数：isr_evt_set

```
void isr_evt_set (
   U16    event_flags,    /* Bit pattern of event flags to set */
   OS_TID task );         /* The task that the events apply to */
```

描述

isr_evt_set通过函数参数为任务设置事件标志。该函数只设置在参数*event_flags*上 对应位为 1 的事件标志。

返回值

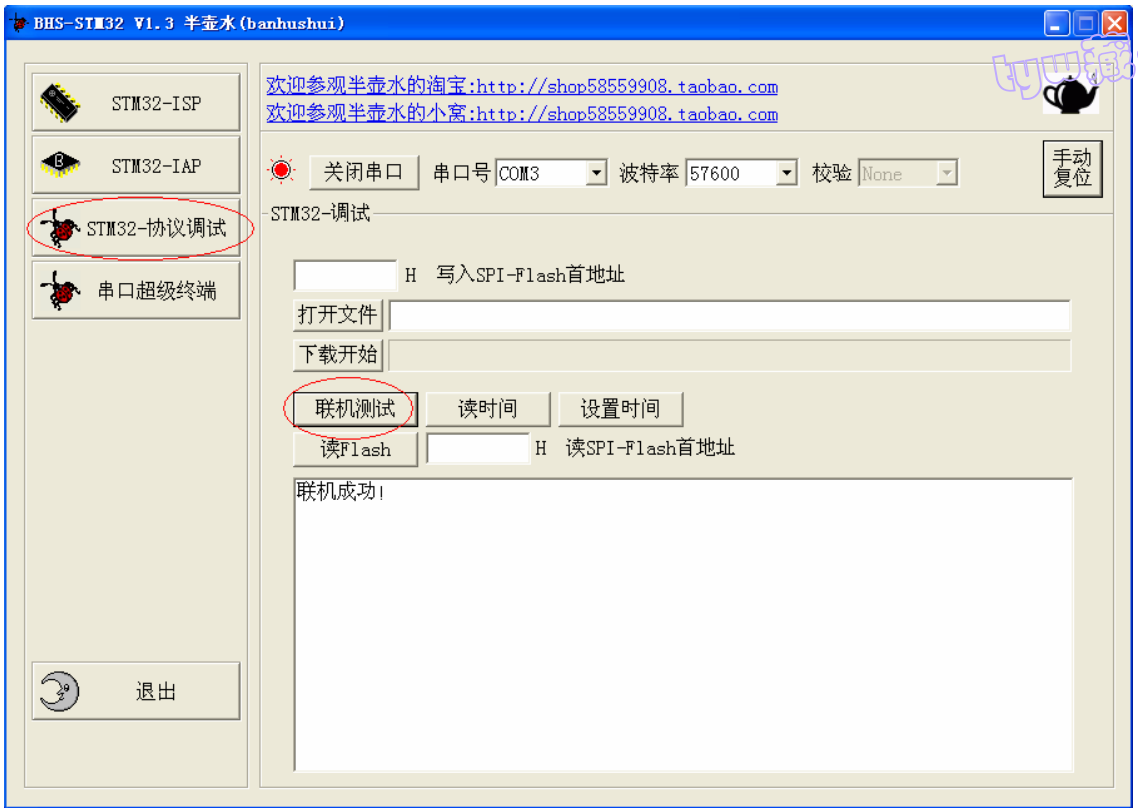
isr_evt_set 函数没有返回值

注意：

只能从 IRQ 中断函数调用 isr_sem_set，而且不能通过 FIQ 函数调用。

当 isr_evt_set 被频繁调用的时候，导致了太多的计时中断，并且 os_clock_demon 任务调度执行频繁。这就造成了任务还没运行 s_evt_wait_o，另外的一个 isr_evt_set 就被调度，即同一个任务有两个 isr_evt_set 函数。当然，这样一个事件就丢失了，因为事件标志没有被加入对象。

PC 软件使用前面提到的调试工具



选择 STM32-协议调试，波特率 57600，这个例子只支持一个命令：【联机测试】

BHS-STM32 实验 45-RTX之TCP uIP 1.0

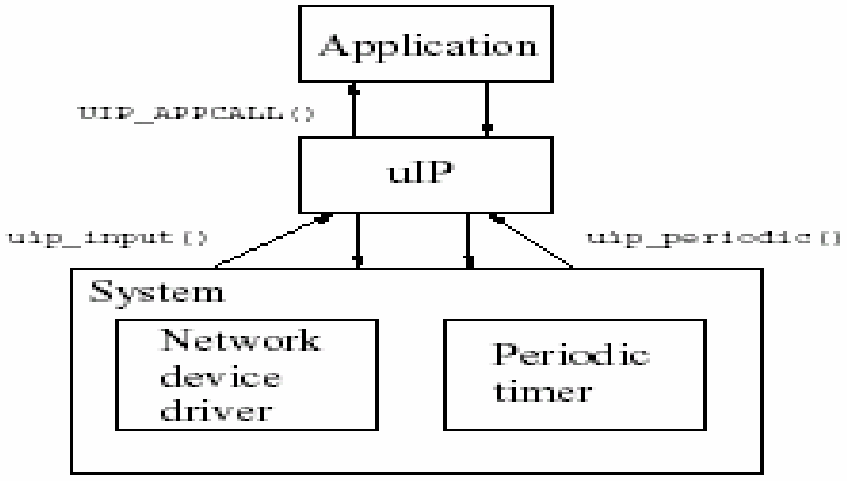
uIP相关知识：

uIP是一个免费的TCP/IP栈，uIP TCP/IP栈是适用于低至8位/16/32位微处理器的嵌入式系统的一个可实现的极小的TCP/IP协议栈。现时，uIP代码的大小和RAM的需求比其它一般的TCP/IP栈要小。

uIP的接口技术

uIP可以看作是一个代码库为系统提供确定的函数。图 1 展示了uIP，系统底层和应用程序之间的关系。uIP提供三个函数到系统底层， uip_init()， uip_input()，和uip_periodic()。应用程序必须提供一个回应函数给uIP。当网络或定时事件发生时，调用回应函数。 uIP提供许多函数和堆栈交互。

要注意的就是 uIP 提供的大部分函数是作为 C 的宏命令实现的，主要是为了速度，代码大小，效率和堆栈的使用。



uIP应用接口

BSD套节字接口使用于大部分的操作系统，它不适合微系统，因为在应用设计里，它逼使一个线程基于编程模块。一个多线程环境代价重大，因为，不但在线程管理里涉及增加代码的复杂性，而且保存每线程堆栈需要额外的储存器，还有执行任务切换的时间开销也摊派在这里。微型系统不会有足够的资源去实现一个多线程环境，因此需要这个环境的应用接口不适合uIP。

相反，uIP使用一个基于编程模块的事件，模块是实现应用程序作为一个C函数被uIP调用的地方，uIP响应一定的事件。uIP调用应用在，当接收数据时，当数据成功送达另一方中止连接时，当一个新的连接建立时，或者当数据需要重发时。 应用程序也周期性地循环等待新数据。应用程序只提供一个回应函数；它提升了应用程序处理不同的网络服务的不同的端口和连接的映射

uIP 与其它 TCP/IP 栈不同的是，当正在重发工作，它需要应用程序的帮助。其它 TCP/IP 栈缓存传输数据在储存器里，直到在连接的最后数据确应成功发送。如果数据需要重传，堆栈在没有通知应用程序下监视着重传工作。通过这种方法，当要等待一个确应，数据必须缓存在储存器里，如果产生一个重发，应用程序可以快速重新生成数据。为了减少储存器的使用量，uIP 利用的论据是应用程序可以重新生成发送的数据和让应用程序参加重发。

uIP应用事件

应用程序必须作为C函数去实现，uIP在任何一个事件发生时调用UIP_APPCALL()。表 1 列出可能的事件和每个事件的对应测试函数。测试函数用于区别不同的事件。函数是作为C宏命令实现的，将会是零值或非零值。注意的是某些函数可以在互相连接时发生(也就是新数据可以在数据确应的同时到达)。

表 1: uIP应用事件和对应的测试参数

一个数据包到达，确应先前发送到数据	uip_acked()
应用程序的新数据包已经到达	uip_newdata()
一个远程主机连接到监听端口	uip_connected()
一个到达远程主机的连接成功建立	uip_connected()
计时时间满重发	uip_rexmit()
计时时间满周期性轮询	uip_poll()
远程主机关闭连接	uip_closed()
远程主机中断连接	uip_aborted()
由于太多重传，连接中断	uip_timedout()

当应用程序调用时，uIP设置全局变量uip_conn去指向当前连接的uip_conn结构 (图 5) 。这可以用来区别不同的服务。一个典型的应用是检查uip_conn->lport (当地TCP端口号)去决定那个服务连接应该提供。例如，如果值uip_conn->lport等于80，应用程序可以决定启动一个HTTP服务，值是23是启动TELNET服务。

※ 接收数据

如果 uIP 测试函数 uip_newdata()值为 1,远程连接的主机有发送新数据。uip_appdata 指针指向实际数据。数据的大小通过 uIP 函数 uip_datalen()获得。在数据不是被缓冲后，应用程序必须立刻启动。

※ 发送数据

应用程序通过使用 uIP 函数 uip_send()发送数据。uip_send()函数采用两个参数；一个指针指向发送数据和数据的长度。如果应用程序为了产生要发送的实际数据需要 RAM 空间，包缓存(通过 uip_appdata 指针指向)可以用于这方面。

在一个时间里应用程序只能在连接中发送一块数据。因此不可以在每个应用程序启用中调用 uip_send()超过一次；只有上一次调用的数据将会发出后才可以。注意，调用 uip_send()以后会改变某些全局变量，在应用函数返回前它不能被调用。

※ 重发数据

如果数据在网络中丢失，应用程序必须重发数据。无论数据收到或没有收到，uIP 保持跟踪，和通知应用程序什么时候察觉出数据是丢失了。如果测试函数 uip_rexmit()为真，应用程序要重发上一次发出的数据。重发就好像原来那样发送，也就是通过 uip_send()。

※ 关闭连接

应用程序通过调用 uip_close()关闭当前连接。这会导致连接干净地关闭。为了指出致命的错误，应用程序可以中止连接和调用 uip_abort()函数完成这个工作。

如果连接已经被远端关闭，测试函数 uip_closed()为真。应用程序接着可以做一些必要的清理工作。

※ 报告错误

有两个致命的错误可以发生在连接中，不是连接由远程主机中止，就是连接多次重发上一数据和被中止。uIP 通过调用函数报告这些问题。应用程序使用两个测试函数 uip_aborted()和 uip _timedout() 去测试那些错误情况。

※ 轮询

当连接空闲时，uIP 在每一个时候周期性地轮询应用程序。应用程序使用测试函数 uip_poll()去检查它是否被轮询过。

※ 监听端口

uIP 维持一个监听 TCP 端口列表。通过 uip_listen()函数，一个新的监听端口打开。当一个连接请求在一个监听端口到达，uIP 产生一个新的连接和调用应用程序函数。如果一个新连接产生，应用程序被调用，测试函数 uip_connected()为真。

※ 打开连接

作为uIP的0.6版，在uIP里面通过使用uip_connect()函数打开一个新连接。这个函数打开一个新连接到指定的IP地址和端口，返回一个新连接的指针到uip_conn结构。如果没有空余的连接槽，函数返回空值。为了方便，函数uip_ipaddr()可以用于将IP地址打包进两个单元16位数组里，通过uIP去代表IP地址。

第一个例子展示了怎样打开一个连接去远端 TCP 端口 8080。如果没有足够的 TCP 连接插槽去允许一个新连接打开，uip_connect()函数返回 NULL 和通过 uip_abort()中止当前连接。第二个例子展示怎样打开一个新连接去指定的 IP 地址。这例子里没有错误检查。

打开一个连接去当前连接的远端的端口8080

```
void connect_example1_app(void) {
    if(uip_connect(uip_conn->ripaddr, 8080) == NULL) {
      uip_abort();
  }
}
```

打开一个到主机192.168.0.1上端口8080的连接

```
void connect_example2(void) {
    u16_t   ipaddr[2];
    uip_ipaddr(ipaddr, 192,168,0,1);
uip_connect(ipaddr, 8080);
}
```

※ 数据流控制

通过函数 uip_stop()和 uip_restart()，uIP 提供存取 TCP 数据流的控制途径。设想一个应用程序下载数据到一个慢速设备，例如磁盘驱动器。如果磁盘驱动器的作业队列满了，应用程序不会准备从服务器接收更多的数据，直到队列排出空位。函数 uip_stop()可以用于维护流控制和停止远程主机发送数据。当应用程序准备好接收更多数据，函数 uip_restart()用于告知远程终端再次发送数据。函数 uip_stopped()可以用于检查当前连接是否停止。

uIP/系统接口

从系统的立场看， uIP由3个C函数 uip_init(),uip_input(), 和 uip_periodic()。uip_init()函数用于初始化uIP堆栈和在系统启动期间调用。当网络设备驱动器读一个IP包到包缓存时，调用函数uip_input()。周期性

运行是调用uip_periodic()，代表的是一秒一次。调用uIP函数是系统的职责。

※ uIP/设备驱动接口

当设备驱动放一个输入包在包缓存里(uip_buf)，系统应该调用 uip_input()函数。函数将会处理这个包和需要时调用应用程序。当 uip_input()返回，一个输出包放在包缓存里。包的大小由全局变量 uip_len 约束。如果 uip_len 是 0，没有包要发送。

※ uIP/周期计时接口

周期计时是用于驱动所有 uIP 内部时钟事件，例如包重发。当周期计时激发，每一个 TCP 连接应该调用 uIP 函数 uip_periodic()。连接编号传递是作为自变量给 uip_periodic()函数的。类似于 uip_input()函数，当 uip_periodic()函数返回，输出的 IP 包要放在包缓存里。下面展示了调用 uip_periodic()函数和监视输出包的一小段代码。在这个特别的例子，函数 netdev_send()是网络驱动的部分，将 uip_buf 数组的目录发出到网上。

周期计时和uIP的接口的例子代码.

```
for(i = 0; i < UIP_CONNS; ++i) {
    uip_periodic(i);
    if(uip_len > 0)
    netdev_send();
}
```

uIP 函数总结

下表包含了所有uIP提供的函数

系统接口	
uip_init() uip_input() uip_periodic()	初始化uIP 处理输入包 处理周期计时事件
应用程序接口	
uip_listen() uip_connect() uip_send() uip_datalen() uip_close() uip_abort() uip_stop() uip_stopped() uip_restart()	开始监听端口 连接到远程主机 在当前连接发送数据 输入数据的大小 关闭当前连接 中止当前连接 停止当前连接 查找连接是否停止 重新启动当前连接
测试函数	
uip_newdata() uip_acked() uip_connected() uip_closed() uip_aborted() uip_timeou() uip_rexmit uip_poll()	远程主机已经发出数据 确应发出的数据 当前连接刚连上 当前连接刚关闭 当前连接刚中止 当前连接刚超时 数据重发 应用程序循环运行
其它	
uip_mss() uip_ipaddr()	获得当前连接的最大的段大小 将IP地址结构打包

htons(),ntohs()	在主机和网络之间转换字节次序

tyw藏书

实现协议

uIP 实现了 TCP/IP 协议组的四个基本协议； ARP [Plu82],IP [Pos81b], ICMP [Pos81a] 和 TCP [Pos81c]。链路层协议例如 PPP 可以实现作为 uIP 下面的设备驱动。应用层协议例如 HTTP, FTP 或 SMTP 可以实现为 uIP 之上的应用程序。

地址解析协议 | ARP

ARP 协议映射了 IP 地址和以太网 MAC 物理地址，它在以太网上的 TCP/IP 操作是需要的。ARP 在 uIP 里实现的是包含一个 IP 到 MAC 地址的映射。当一个 IP 包要在以太网上发出，查询 ARP 表，去找出包要发送去的 MAC 地址。如果在表里找不到 IP 地址，ARP 请求包就会发出。请求包在网络里广播和请求给出 IP 地址的 MAC 地址。主机通过发出一个 ARP 回应，响应请求 IP 地址。当 uIP 给出一个 ARP 回应，更新 ARP 表。

为了节省储存器，一个 IP 地址的 ARP 请求覆盖发出的请求输出 IP 包。它是假定上层将重新发送那些被覆盖了的数据。

每十秒表更新一次，旧的条目会被丢弃。默认的 ARP 表条目生存时间是 20 分钟。

※　网际协议 | IP

uIP 的 IP 层代码有两个职责：验证输入包的 IP 头的正确性和 ICMP 和 TCP 协议之间多路复用。 IP 层代码是非常简单的，由 9 条语句组成。事实上，uIP 的 IP 层极大地简化了，它没有实现碎片和重组。

※　因特网信息控制协议 | ICMP

在 uIP 里，只有一种 ICMP 信息实现了： ICMP 回响信息。ICMP 回响信息常常用于 ping 程序里的检查主机是否在线。在 uIP 里，ICMP 回响处理在一个非常简单的方式。ICMP 类型字段的改变是从 \echo"类型到 \echo reply"类型，从而 ICMP 调整校验和。其次，IP 地址里的 IP 头交换，包发回到原先的发送者。

※　传输控制协议| TCP

为了减少储存器的使用，uIP 里的 TCP 没有实现发送和接收数据的调整窗口。输入的 TCP 段不会通过 uIP 缓存，但必须立即由应用程序处理。注意这不能避免应用程序自己缓冲数据。输出数据时，uIP 不能在每个连接有超过一个未解决的 TCP 段。

※　连接状态

在 uIP，每个 TCP 连接的完全态包含当地和远端的 TCP 端口编号，远程主机的 IP 地址，重发时间值，上一段重发的编号，和连接的段的最大尺寸。除此之外，每个连接也可以保持一些应用状态。三个序列号是，期望接收的下一个字节的序列号，上一发送段第一字节的序列号，下一发送字节的序列号。连接的状态由 uip_conn 结构表现，可以在图 5 看到。一个 uip_conn 结构数组用于在 uIP 里保持所有的连接。数组的大小等于同时的最大数量的连接，它在编译时间里设置(看第 4 节)。

uip_conn 结构

```
struct uip_conn {
u8_t    tcpstateflags; /* TCP 状态和标志. */
u16_t    lport, rport; /*当地和远端端口. */
u16_t    ripaddr[2]; /*同等远端的 IP 地址. */
u8_t    rcv_nxt[4]; /*  我们期待接收的下一个序列号. */
u8_t    snd_nxt[4]; /*上一个发送的序列号. */
u8_t    ack_nxt[4]; /*  通过从远端的下一个应答去应答序列号. */
u8_t    timer; /*  重发时间 r. */
u8_t    nrtx; /*计算特殊段的重发数量. */
u8_t    mss; /*  连接的最大段大小. */
u8_t    appstate[UIP_APPSTATE_SIZE];
};
```

※ 输入处理

TCP 输入处理和检验 TCP 校验和一起开始。如果校验和是对的，在当前活动的 TCP 连接之间源、目的端口号和 IP 地址复用包。没有活动的连接符合输入包时，如果包不是一个监听端口的连接请求，包丢弃。如果包是一个关闭端口的请求，uIP 发一个 RST 包回应。

如果发现了一个监听端口，uip_conn 结构数组扫描任何一个非活动连接。如果发现一个，数组由新连接的端口号和 IP 地址填充。 如果连接请求携带一个 TCP MSS (最大段大小)选择，它会分析，再次检查当前最大段大小 MSS 去决定当前连接的 MSS，前后者的最小值会被选择。最后，一个回应包发去确应开启连接。应该将输入包送去一个已经活跃的连接，包的序列号和从远端主机来的期望的下一个序列号一起被检查(uip_conn 结构里的 rcv_nxt 变量显示于图 5)。如果序列号不是期望得到的下一个，包会被丢掉和发一个 ACK 去指出期望得到的下一个序列号。紧接着，检查输入包里的确应号，看看是否确应连接的所有输出数据。它做了后，应用程序会知道这个事实的。

当序列号和确应号被检测过，依靠当前 TCP 状态，包将会被不同地处理。如果连接在 SYN-RCVD 状态和输入包确应先前发送的 SYNACK 包，连接将会输入 ESTABLISHED 状态，调用应用函数去通知已经完全连接。在连接的建立状态，如果有新数据由远端主机发送或者远端主机确应之前发送的数据，就调用应用函数。

当应用函数返回，TCP 检查应用程序是否还有数据要发。如果有，一个 TCP/IP 包会形成在包缓存里。

※ 输出处理

输出处理过程比输入处理直接和简单得多。 基本上，所有 TCP 和 IP 头字段由 uip_conn 结构里的值充满，计算 TCP 和 IP 的校验和。当 uip_process()函数返回，包通过网络设备驱动发出去。

※ 重发

当 uIP 通过 periodic_timer 被调用时，重发就进入运作(看段 2.2.2)。连接里有些特殊的数据(也就是数据发出了去但仍没有确应的)通过 UIP_OUTSTANDING 位在 uip_conn 结构里的 TCP 状态标志变量标记(图 5)。那个连接，时间变量减少。当时间到达零，上一段必须重发和调用应用函数去做真正的重发。如果一个特殊段重发编号超出一个可设置的界限，连接会结束和发一个 RST 段到远端连接结束，调用应用函数去通知它连接超时。

※ 重置 TCP

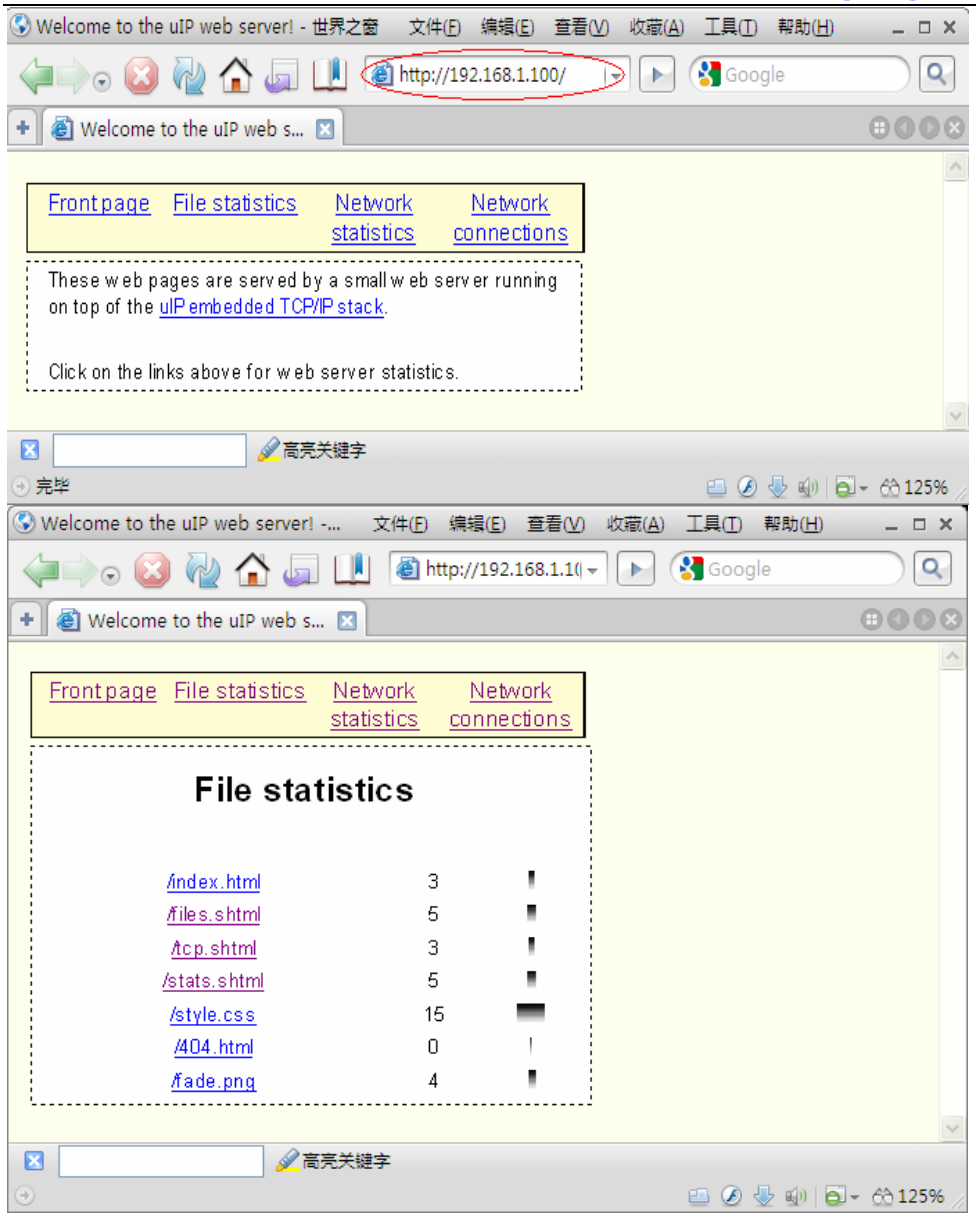
TCP 规格要求如果 TCP 头里的序列号和确应号在当前连接的接收窗口失去了，有 RST (复位)标志设置的包必须断开连接。为了减少代码的大小，uIP 不严格遵守这个规定。相反，如果一个有 RST 标志设置的包在连接里到达，连接会消灭那些不重要的序列号和确应号值。这个行为将会在将来的 uIP 版本修订。

本例是移植的开源的 uIP1.0

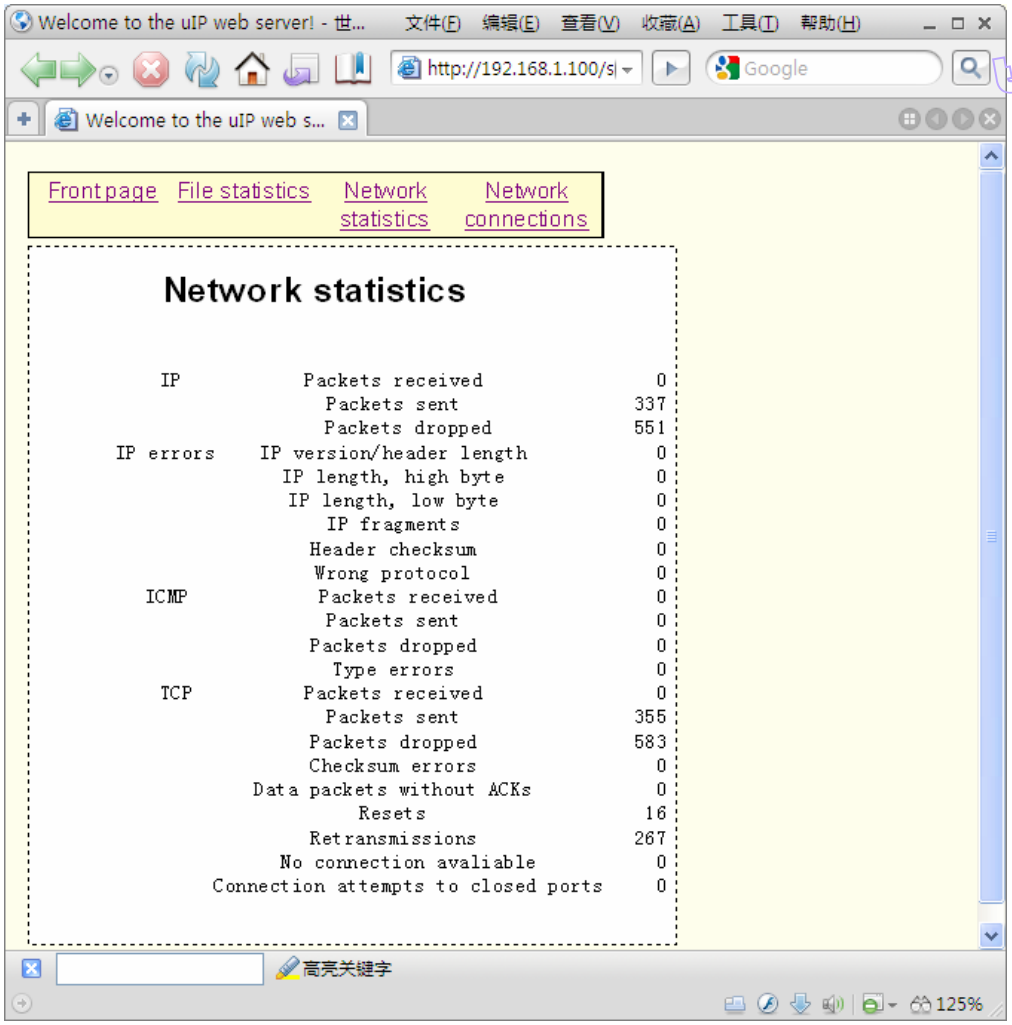
运行改例程，在命令行输入：ping 192.168.1.100 能看到已经连接上网络了

```
C:\Documents and Settings\long>ping 192.168.1.100

Pinging 192.168.1.100 with 32 bytes of data:

Reply from 192.168.1.100: bytes=32 time=2ms TTL=128
Reply from 192.168.1.100: bytes=32 time=1ms TTL=128
Reply from 192.168.1.100: bytes=32 time=1ms TTL=128
Reply from 192.168.1.100: bytes=32 time=1ms TTL=128

Ping statistics for 192.168.1.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 2ms, Average = 1ms
```

在IE地址栏输入：http://192.168.1.100 可以查看网页


Welcome to the uIP web server! - 世界之窗
http://192.168.1.100/
Welcome to the uIP web s...
Front page
File statistics
Network statistics
Network connections
These web pages are served by a small web server running on top of the uIP embedded TCP/IP stack.
Click on the links above for web server statistics.
完毕
125%
Welcome to the uIP web server! -...
http://192.168.1.1(
Welcome to the uIP web s...
Front page
File statistics
Network statistics
Network connections
File statistics
/index.html 3
/files.shtml 5
/tcp.shtml 3
/stats.shtml 5
/style.css 15
/404.html 0
/fade.png 4
125%

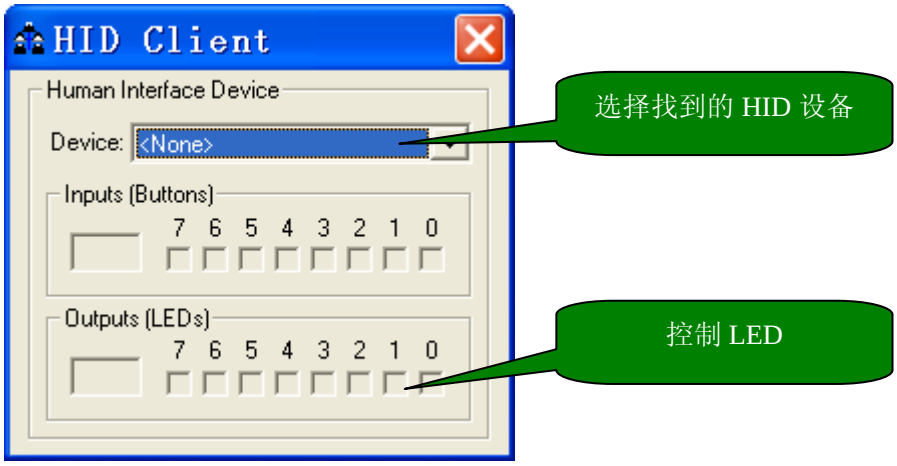
tyw藏书



关于设计嵌入式 WEB 服务器，可以参考\资料文档\TCP-IP\嵌入式 WEB 服务器及远程测控应用详解

BHS-STM32 实验 46-RTX_USB_HID

本例是 USB-HID 的应用，HID 无需安装驱动，因为操作系统一般都带了 HID 的驱动了，该例子需要 PC 软件 HIDClient.exe 配合使用，通过该软件可以控制板上的 LED 亮灭



BHS-STM32 实验 47-RTX-CAN

本例是与基础实验里面的 CAN 实验实现完全相同的功能，并且可以和前面 2 个 CAN 实验通信，正常通信能看到板上的 LED 闪烁，这个实验使用了 RTX 操作系统，实验方法与前面的 CAN 实验完全相同

BHS-STM32 实验 48-RTX-3 点触摸校正

ww藏书

本例子使用 3 点法校正触摸屏，一般情况下触摸屏是非线性的，都需要校正才能使用。
校正原理参考《四线电阻式触摸屏控制与校准.pdf》，

当屏上显示红点时，用手指触摸该点，直到该点变为绿色抬起手指，屏上依次出现 3 个点，3 个点都变绿时校正完成，这时用手触摸屏任意位置，观察光标是否跟手指位置对应

说明：
1.触摸是不可太用力，否则触摸屏易碎
2.为了简化程序，例程不一定有中文提示了。只要出现红点就开始触摸校正。
(说明：本例与前面的触摸校正例子功能完全相同，只是使用 RTX 操作系统而已)

BHS-STM32 实验 49-BHS-GUI-DEMO

简介：

本例子是一个综合实验，包含了触摸校正，SPI-FLASH 编程，串口通信，RTC 实时时钟，原创 BHS-GUI
最好在已经比较熟悉了 STM32 的情况下使用该例子。
系统开机首先进入触摸屏校准界面，校准后立即进入主界面

BHS-GUI使用的资源

所有资源在【\BHS-STM32 例程\高级例程-(实战篇)\RTX 操作系统\BHS-GUI-DEMO 发布 20100419(V1.0.1)\字库及自定义图片（GUI 使用到的资源）】文件夹里，所有资源合并成最终资源文件【所有字库图片文件.bin】

中文显示支持 8*16 点 ASCII，16 *16 点汉字(GB2312)，24*24 点汉字
字库文件写入 SST25VF080 SPI 接口 Flash 中
字库文件地址如下：
16 *16 点汉字：　0H~3FFFFH(256K)
8 *16 点 ASCII：40000H~40FFFH(4K)，实际 95 个字符占 1520 字节，Flash1 扇区 4K
24*24 点汉字：　41000H~BAFFFH(488K)，实际 41000H~BAB60
16*24 点 ASCII：BB000H~ BCFFFH(8K)实际 95 个字符占 4560 字节

32×32 图标：32×32×2=2048
1 个扇区保存 2 个图标
BD000H~C6FFFH：10 扇区保存 20 个图标


“窗口关闭”　0xBD000

64×64 图标：64×64×2=8192
C7000H~E6FFFH：32 扇区保存 16 个图标


"画图", 0xC7000


"串口", 0xD3000


"关机", 0xCD000


"时钟", 0xCF000


"复制", 0xD1000

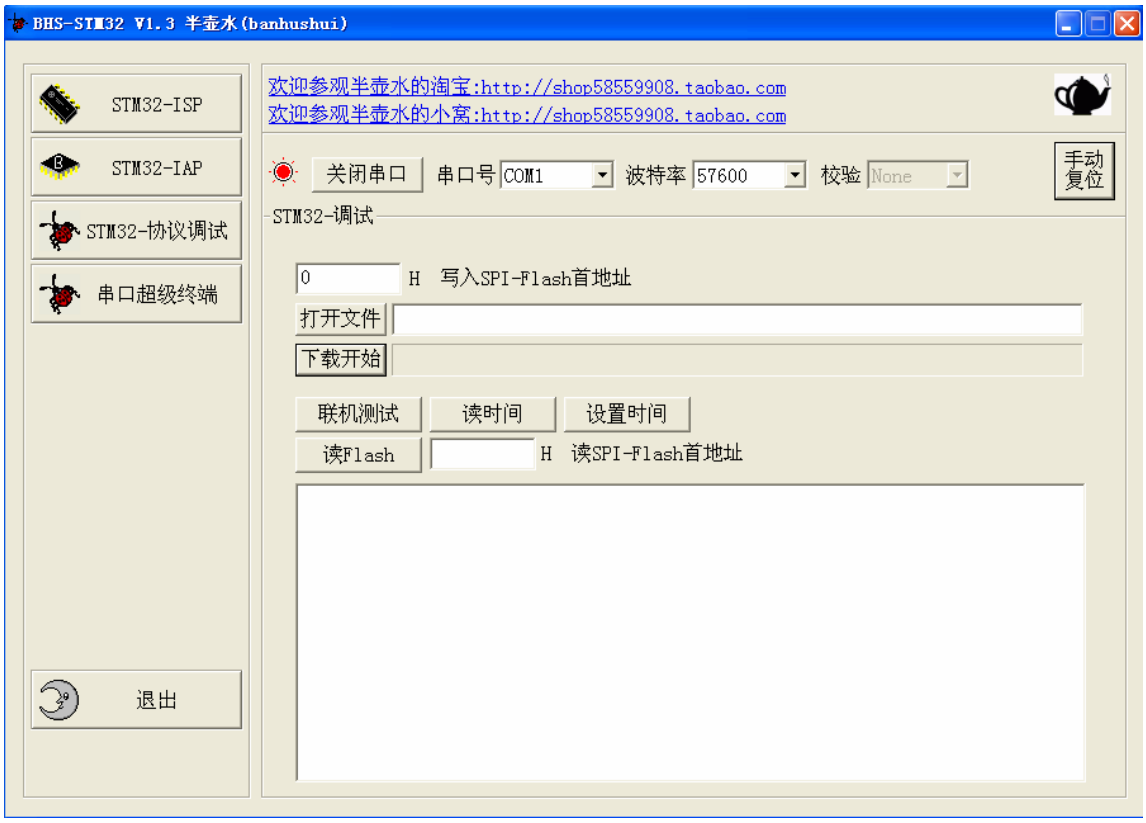
tyw藏书

E7000H~EAFFFH:　4 扇区保存 1 个 logo 图标

LOGO 地址：0xE7000

资源文件【所有字库图片文件.bin】使用调试工具通过开发板的串口 1 下载到 SPI-FLASH 中去。

注意：下载前开发板要先写入【BHS-GUI-DEMO】程序。



常用GUI函数介绍

创建按钮

```
CButton *Button(u16 X, u16 Y, u16 Width, u16 Height, COLOR Color,
                u8 Halign, u8 Font, u16 FontColor, char *text, u32 picAddr )
//    uint16 X;          //按键 X 坐标
//    uint16 Y;          //按键 Y 坐标
//    uint16 Width;      //按键长度
//    uint16 Height;     //按键高度
//    uint16 Color;      //按键颜色
//    uint8 Halign;      //文字水平对齐方式 Left,Right,Center
//    uint8 Font;        //字体，支持 16x16,24x24 点阵，16=16x16 点阵，24=24x24 点阵
//    uint16 FontColor; //字体颜色
//    char *text;        //文字
//    uint32 picAddr;    //图片地址，0 表示无图片，!=0 表示有图片
```

tyw藏书

创建文本编辑框

```
CEdit *EditBasic(u16 X, u16 Y, u16 Width, u16 Height, COLOR Color,
                 u8 Halign, u8 Valign, u8 Font, u16 FontColor, char *text )
//     uint16 X;          // X 坐标
//     uint16 Y;          //Y 坐标
//     uint16 Width;      //长度
//     uint16 Height;     //高度
//     uint16 Color;      //颜色
//     uint8 Halign;      //文字水平对齐方式 Left,Right,Center
//     uint8 Font;        //字体，支持 16x16,24x24 点阵，16=16x16 点阵，24=24x24 点阵
//     uint16 FontColor; //字体颜色
//     char *text;        //文字
```

重新设置编辑框文字

```
void EditSetWindowText(CWidgets *Widgets, char *text)
// CWidgets *Widgets   //编辑框指针
// char *text          //文字
```

创建静态文本

```
CStatic *StaticBasic(u16 X, u16 Y, u16 Width, u16 Height, COLOR Color,
                     u8 Halign, u8 Valign, u8 Font, u16 FontColor, char *text )
//     uint16 X;          // X 坐标
//     uint16 Y;          // Y 坐标
//     uint16 Width;      //长度
//     uint16 Height;     //高度
//     uint16 Color;      //颜色
//     uint8 Halign;      //文字水平对齐方式 Left,Right,Center
//     uint8 Font;        //字体，支持 16x16,24x24 点阵，16=16x16 点阵，24=24x24 点阵
//     uint16 FontColor; //字体颜色
//     char *text;        //文字
```

创建进度条

```
CProgressCtrl *ProgressCtrl (u16 X, u16 Y, u16 Width, u16 Height, COLOR Color,
                             COLOR FontColor, u16 range, u16 pos, u16 step )
//     uint16 X;          // X 坐标
//     uint16 Y;          // Y 坐标
//     uint16 Width;      //长度
//     uint16 Height;     //高度
//     uint16 Color;      //背景颜色
//     uint16 FontColor;  //前景颜色
//     uint16 range;      //进度条范围
//     uint16 pos;        //当前进度
//     u16 step;          //步进值
```

//进度条前进一步

void ProgressStepIt(CProgressCtrl *pProgressCtrl)

//　　CProgressCtrl *pProgressCtrl　　进度条指针

tyw藏书

创建弹出式对话框

uint8 MessageBox(char *text, char *B1text, char *B2text, char *B3text, u16 delay)

char *text=消息内容

char *B1text=按键 1 文字

char *B1text=按键 1 文字

char *B2text=按键 2 文字

u16 delay 延时(毫秒)自动关闭, 0xffff，不自动关闭

返回值：0，1，2：对应按键序号

可以创建带 0～3 个按键的对话框，没有按键的对话框必须延时关闭。

消息框：消息框默认居中显示，消息默认字体都是 16 点阵

限制最多显示 100 个 ASCII 字符宽度,超过 100 个将丢弃

初始化光标

void InitCursor(u16 type, COLOR Color)

//u16 type　　光标类型 CURSOR_CROSS=十字光标.；CURSOR_ARROW=箭头光标

//COLOR Color　　光标颜色

开启/关闭光标

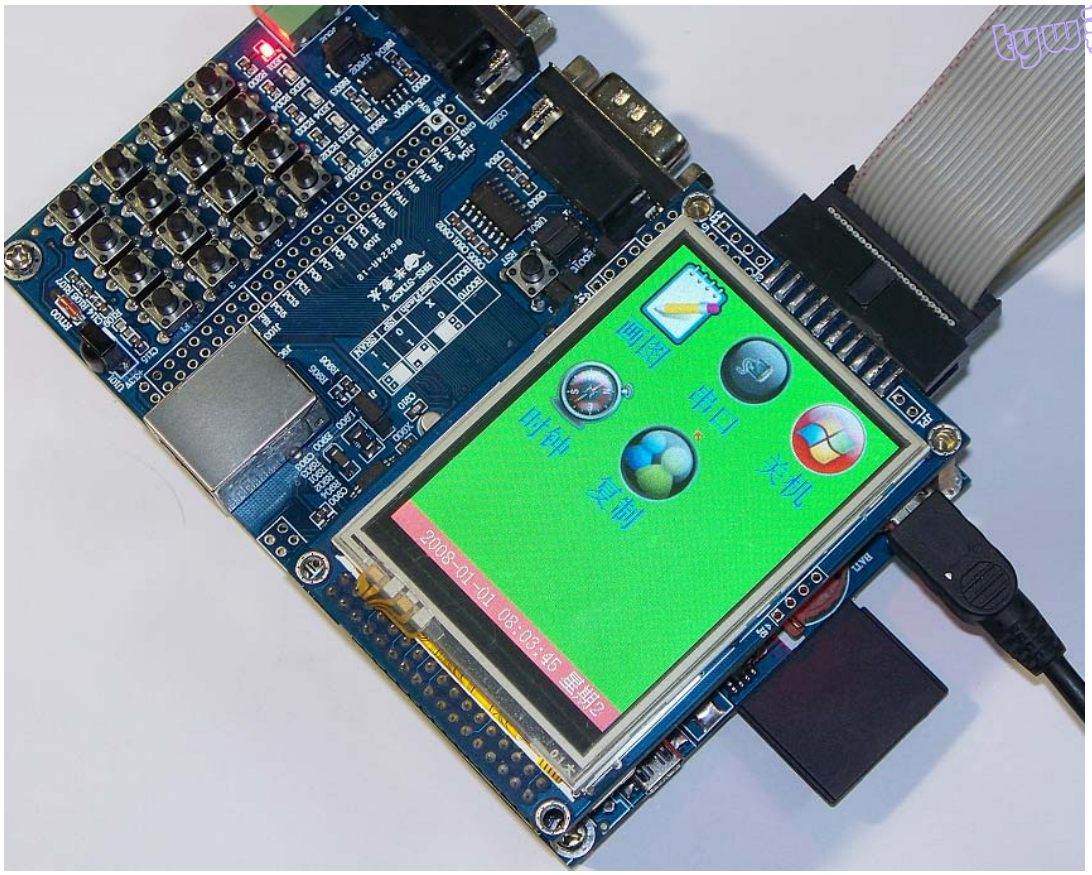
void SetCursor(u16 x, u16 y, uint8 flag)

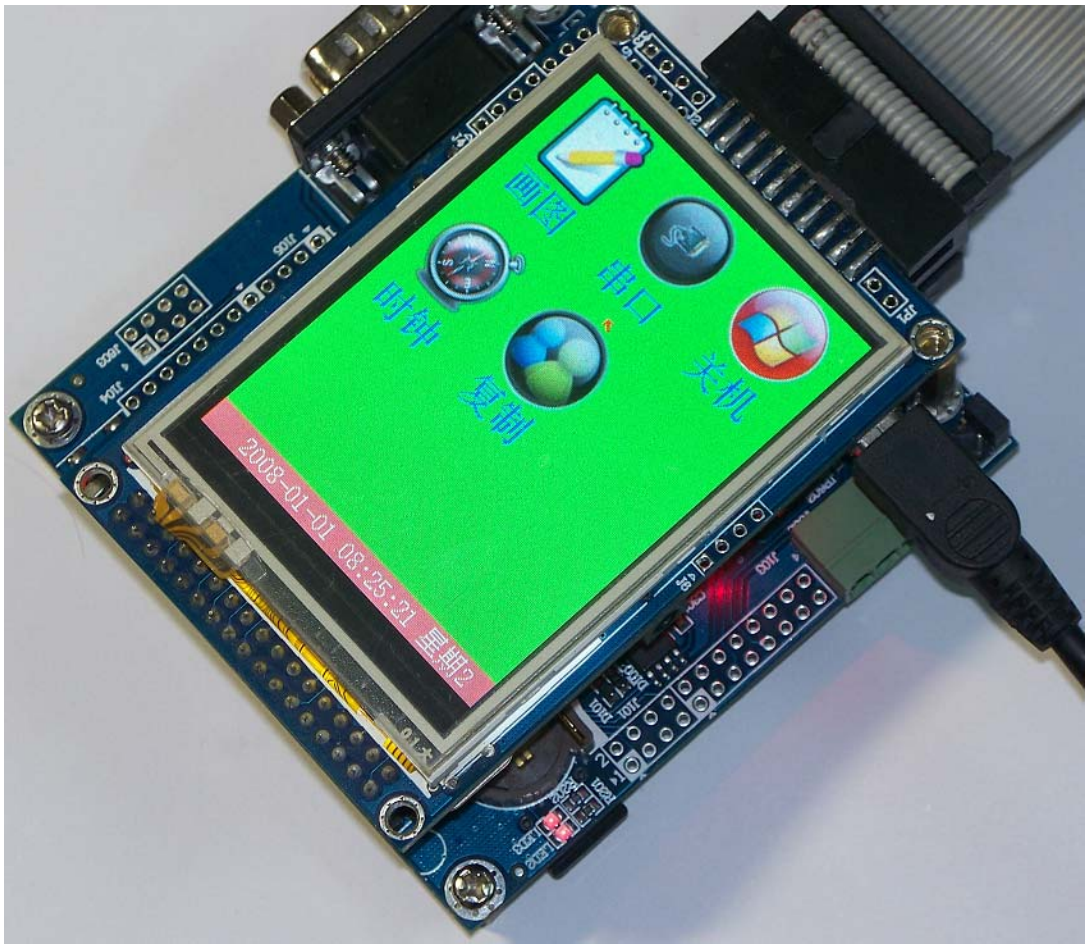
//u16 x　　X 坐标

//u16 y　　Y 坐标

//uint8 flag　　0=关闭光标，1=开启光标

主窗口界面


画图
串口
时钟
关机
复制
2008-01-01 08:03:45 星期2


画图
串口
时钟
关机
复制
2008-01-01 08:25:21 星期2

本窗口下，单击图标进入相应对话窗口

tyw藏书

```
void Dlg_Main(void)
{
 pFUN    event;
 uint16 iy;

 CWidgetsEvent *pWidgetsEvent;//[9]={0};//={ButtonExit, NULL};
     //清空窗口控件内存
     memset(CurWindowsWidgetsEvent, 0, sizeof(CurWindowsWidgetsEvent));

     pWidgetsEvent=CurWindowsWidgetsEvent;
     //绘制背景色
     FillSolidRect(0, 0, 240, 320, GREEN);
     //有消息框的对话框一定要设置背景颜色
     //弹出式对话框关闭后使用该颜色重绘窗口
     //设置背景色
     SetDlgBackColor(GREEN);

     iy=2;

     //下面是按键输入参数：
     //按键:  X 坐标, Y 坐标，宽度，高度，背景色
     //文字中对齐；ALIGN_LEFT=左对齐;ALIGN_CENTER=中对齐;ALIGN_RIGHT=右对齐,
     //文字字体，目前支持 16，24 点阵，文字颜色，文字内容
     //字体==0 将不显示文字
     //背景图片地址，0=不使用背景图片，其他=背景图片在 FLASH 中的物理地址

     //添加【画图】按键
     pWidgetsEvent ->widgets = Button(4, iy, 64, 64, BLUE,      ALIGN_CENTER, 24, BLUE, "画图",
0xC7000);//WHITE
     //如果该按键不响应事件，那么事件应该赋空值
     //pWidgetsEvent++->event=NULL;
     //添加单击【画图】按键执行的事件
     pWidgetsEvent++->event=ExitDlg_Main_EnterDlg_Paint;

     //添加【串口】按键
     pWidgetsEvent ->widgets= Button(84, iy, 64, 64, BLUE,      ALIGN_CENTER, 24, BLUE, "串口",
0xD3000 );
     //如果该按键不响应事件，那么事件应该赋空值
     //pWidgetsEvent++ ->event=NULL;
     //添加单击【串口】按键执行的事件
     pWidgetsEvent++ ->event=ExitDlg_Main_EnterDlg_Com;
     //添加【关机】按键
     pWidgetsEvent ->widgets= Button(164, iy, 64, 64, BLUE,      ALIGN_CENTER, 24, BLUE, "关机",
0xCD000);
```

```
    //如果该按键不响应事件，那么事件应该赋空值
    //pWidgetsEvent++ ->event=NULL;
    //添加单击【关机】按键执行的事件
    pWidgetsEvent++ ->event=Close;

    iy+=79+28;
    //添加【时钟】按键
    pWidgetsEvent ->widgets=(CWidgets*)Button(4, iy, 64, 64, BLUE,      ALIGN_CENTER, 24, BLUE, "时钟", 0xCF000 );
    //如果该按键不响应事件，那么事件应该赋空值
    //pWidgetsEvent++ ->event=NULL;
    //添加单击【时钟】按键执行的事件
    pWidgetsEvent++ ->event=ExitDlg_Main_EnterDlg_Clock;

    //添加【复制】按键
    pWidgetsEvent ->widgets=(CWidgets*)Button(84, iy, 64, 64, BLUE,      ALIGN_CENTER, 24, BLUE, "复制", 0xD1000 );
    //如果该按键不响应事件，那么事件应该赋空值
    //pWidgetsEvent++ ->event=NULL;
    //添加单击【复制】按键执行的事件
    pWidgetsEvent++->event=ExitDlg_Main_EnterDlg_FlashCopy;

    //初始化光标
    InitCursor(CURSOR_ARROW, RED);//WHITE
    //开启光标，并且第一次设置光标出现位置
    SetCursor(240/2-1, 100, 1);

    //--------------------------------
    RTC_ITConfig(RTC_IT_SEC, ENABLE); //RTC 秒中断在主窗口建立后才开启
    while(1)
    {
        //下面是消息循环处理。
        //等待消息，该消息来自于触摸屏事件
        //触摸屏检测到单击按键事件后将按键对应的事件发送到该任务
        if( os_mbx_wait (EventMailbox, (void **)&event, 0)==OS_R_OK )
        {
            event();

            if(CloseFlag)
            {
                CloseFlag=0;
                break;
            }
        }
        //底部任务栏显示日期时间
```

```
        displayTime();

        os_dly_wait(50/OS_TIME);
    }
}
```

tyw藏书

//----------

下面介绍使用到的有关 RTX 相关函数：

从 *mailbox* 中得到一个消息指针 os_mbx_wait

```
OS_RESULT os_mbx_wait (
   OS_ID  mailbox,      /* The mailbox to get message from 邮箱得到消息*/
   void** message,      /* Location to store the message pointer 放置消息指针区 */
   U16    0xFFF );    /* Wait time for message to become available 消息可得等待时间
*/
```

描述

如果邮箱不空，os_mbx_wait 函数从 mailbox 中得到一个消息指针，函数把从邮箱中获得消息指针放入参数 message 所指的地方。

如果邮箱为空，RTX 核休眠任务。*timeout* 规定了任务预约时间的长度。当预约时间（timeout）已到时或者邮箱中的消息变为有效时，RTX 核唤醒任务。

可以给预约时间（timeout）设置任何从 0 到 0xFFFE 的值，也可将其设置为 0xFFFF 获得一个不确定值。如果指定预约时间为 0，任务立即继续，即使有高优先的任务要执行，也无论邮箱当前是否有消息。

os_mbx_wait 函数在 RL-RTX 库中。其原型在 rtl.h.函数库中。

返回值

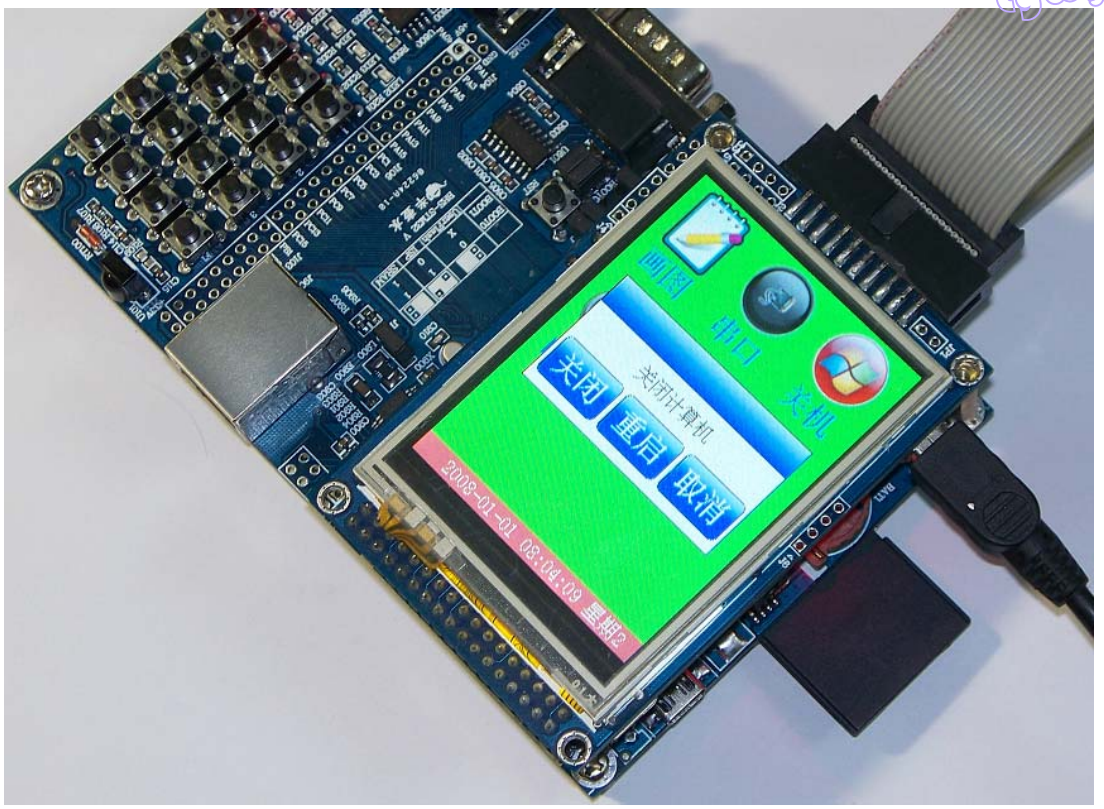
OS_R_MBX　　任务等到消息被放入邮箱中。
OS_R_TMO　　邮箱中在等到可用消息之前，预约期满。
OS_R_OK　　邮箱中的消息可用，任务继续不等待。

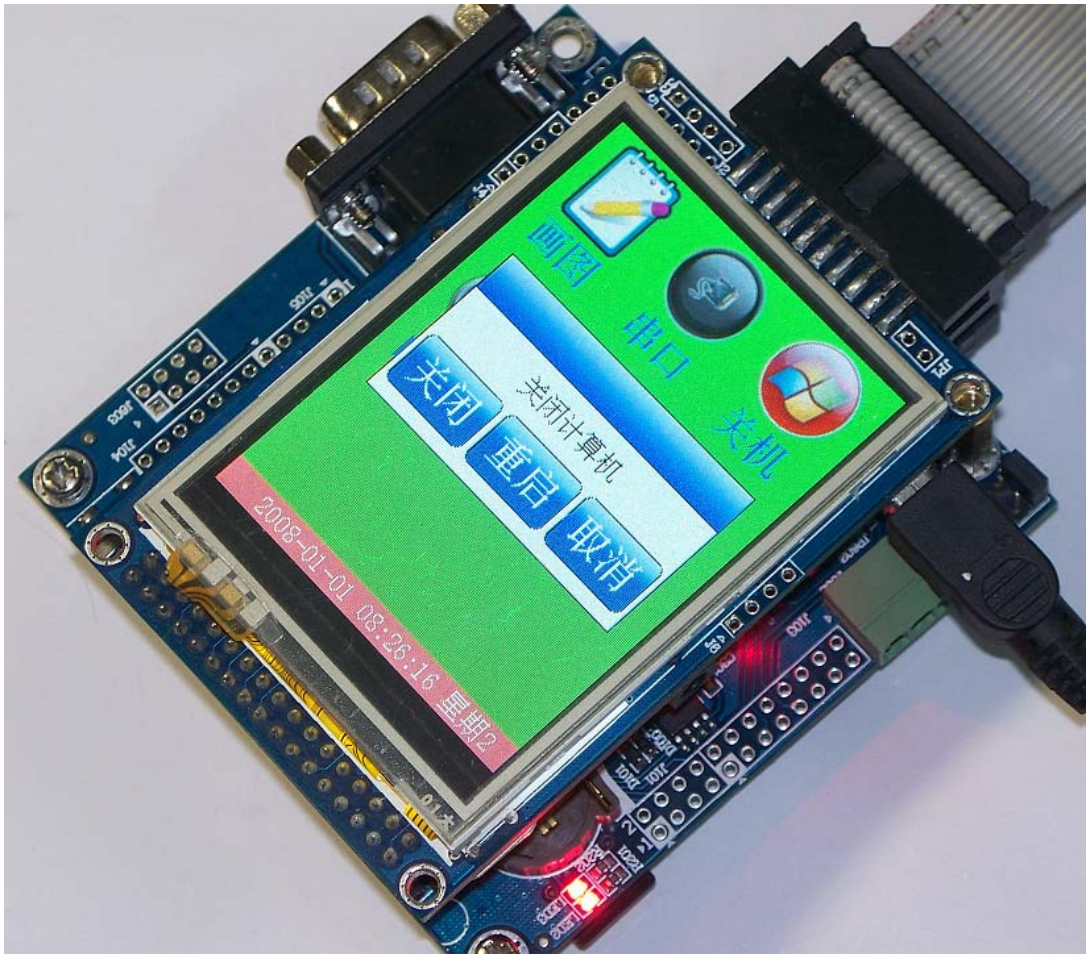
注意：

- 在对邮箱进行任何操作之前，必须声明和初始化邮箱对象；
- 预约时间是通过系统时间衡量；
- 当从邮箱中获得消息时，必须释放包含消息的内存模块，避免溢出存储区；
- 当从邮箱中得到消息时，为新消息产生的新的内存空间。

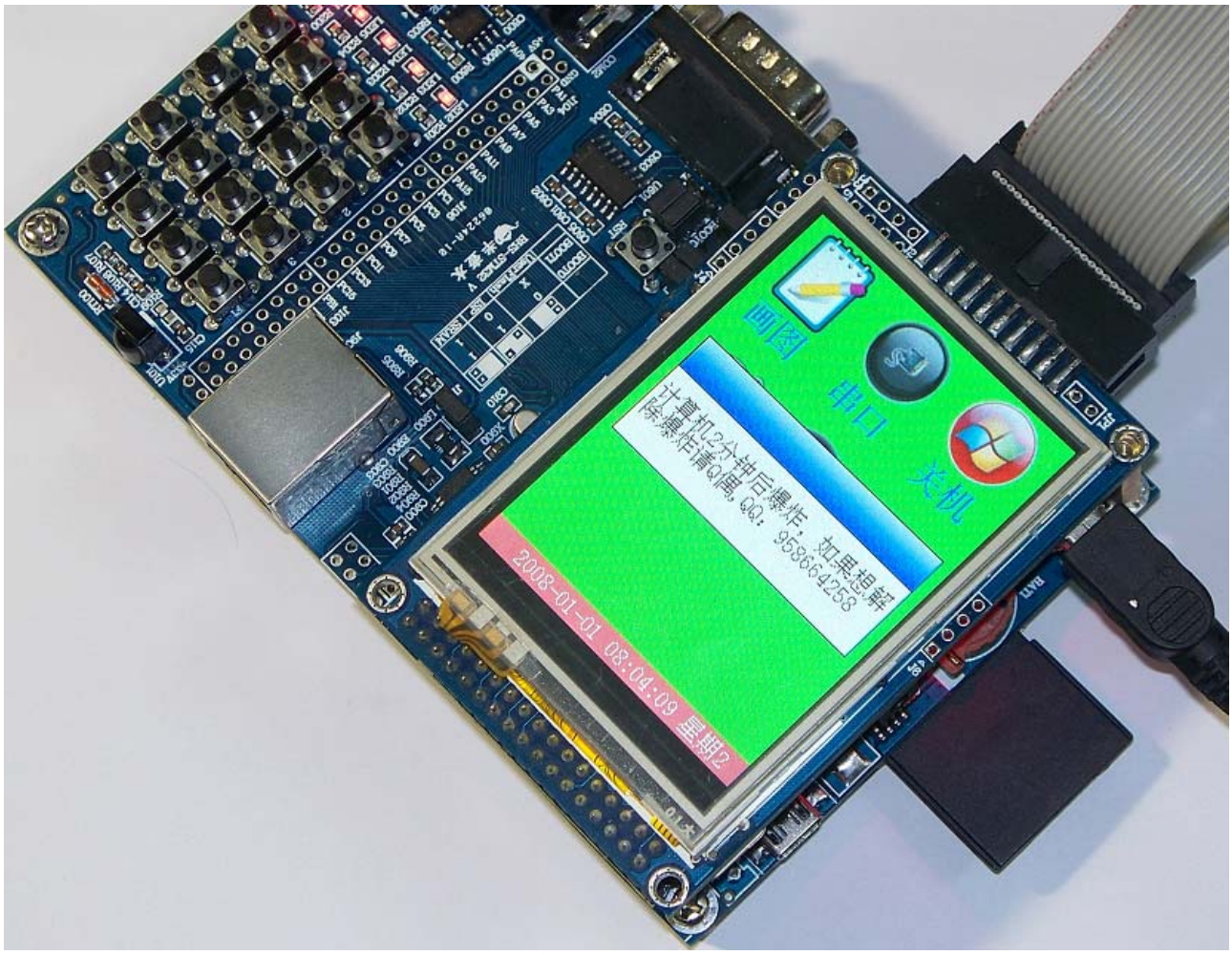
弹出式消息窗口界面

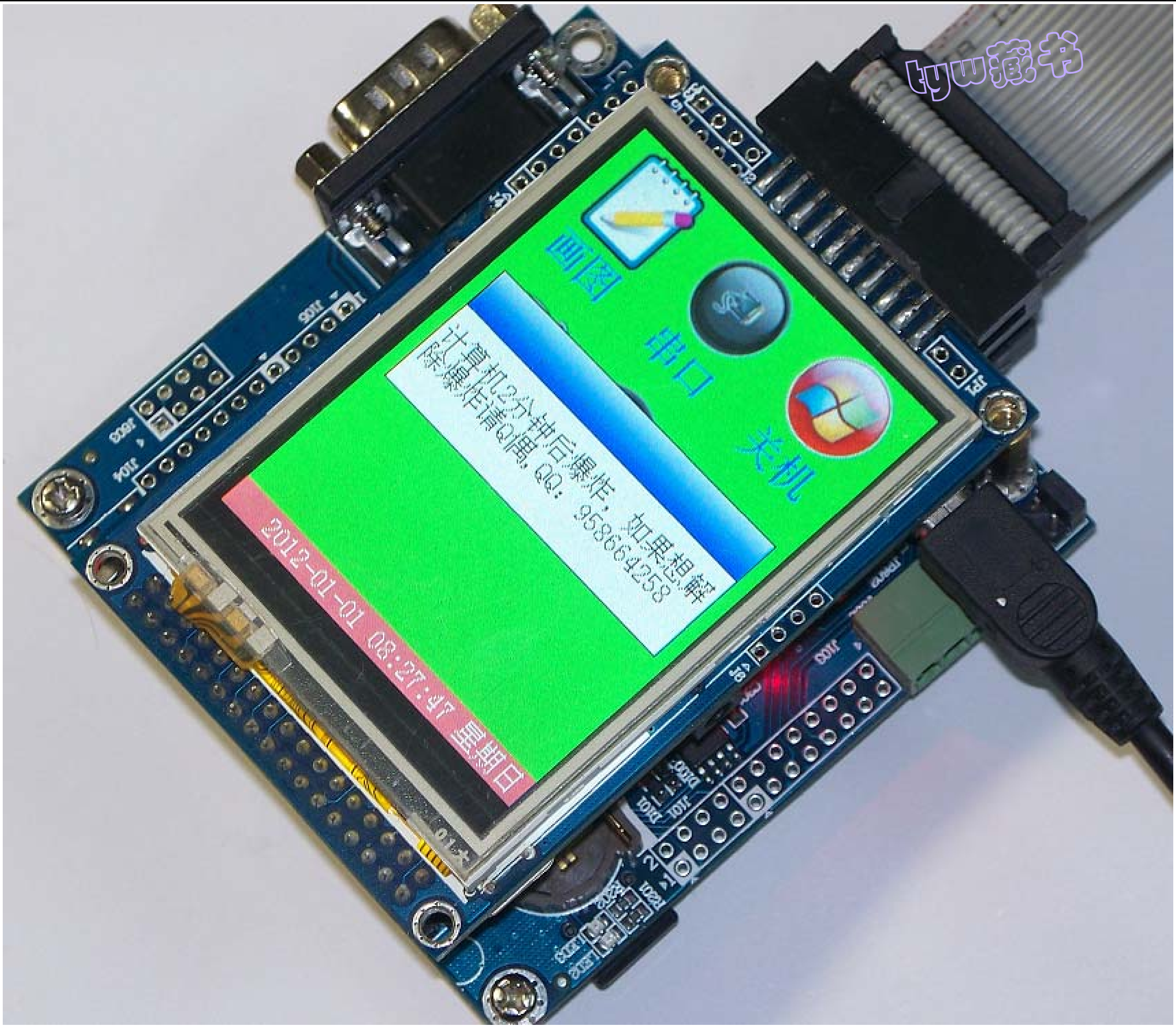
lyw藏书



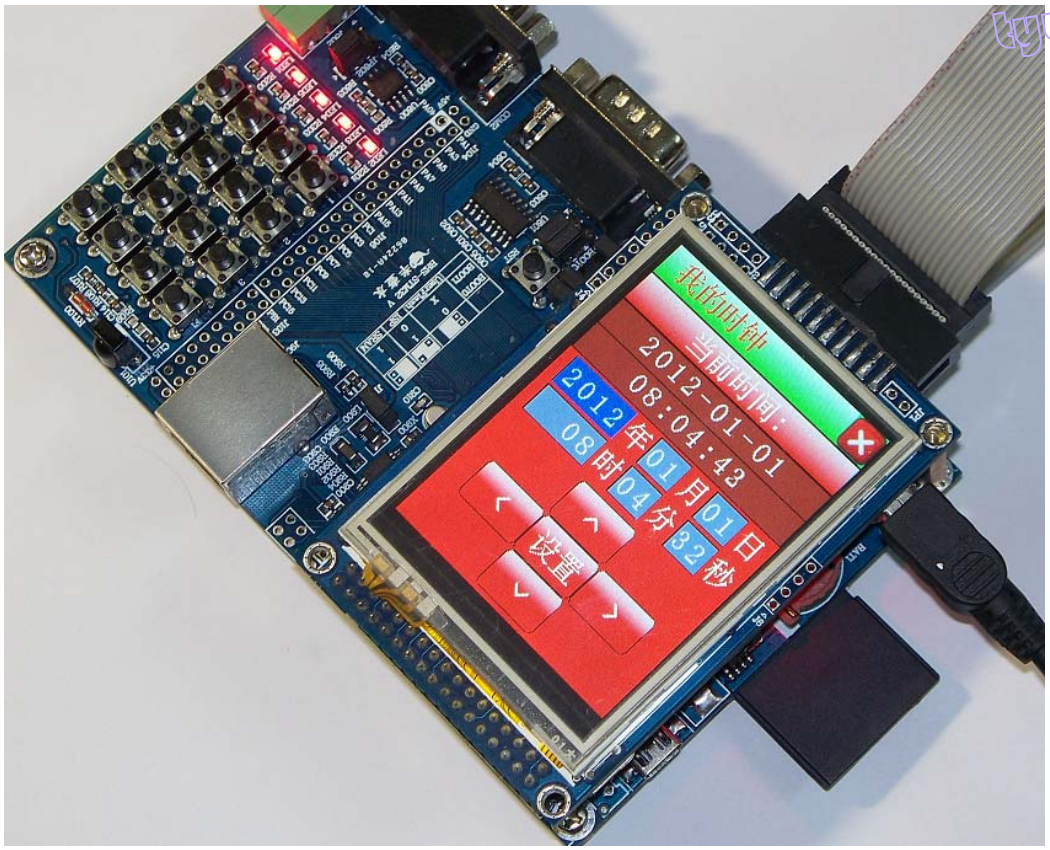


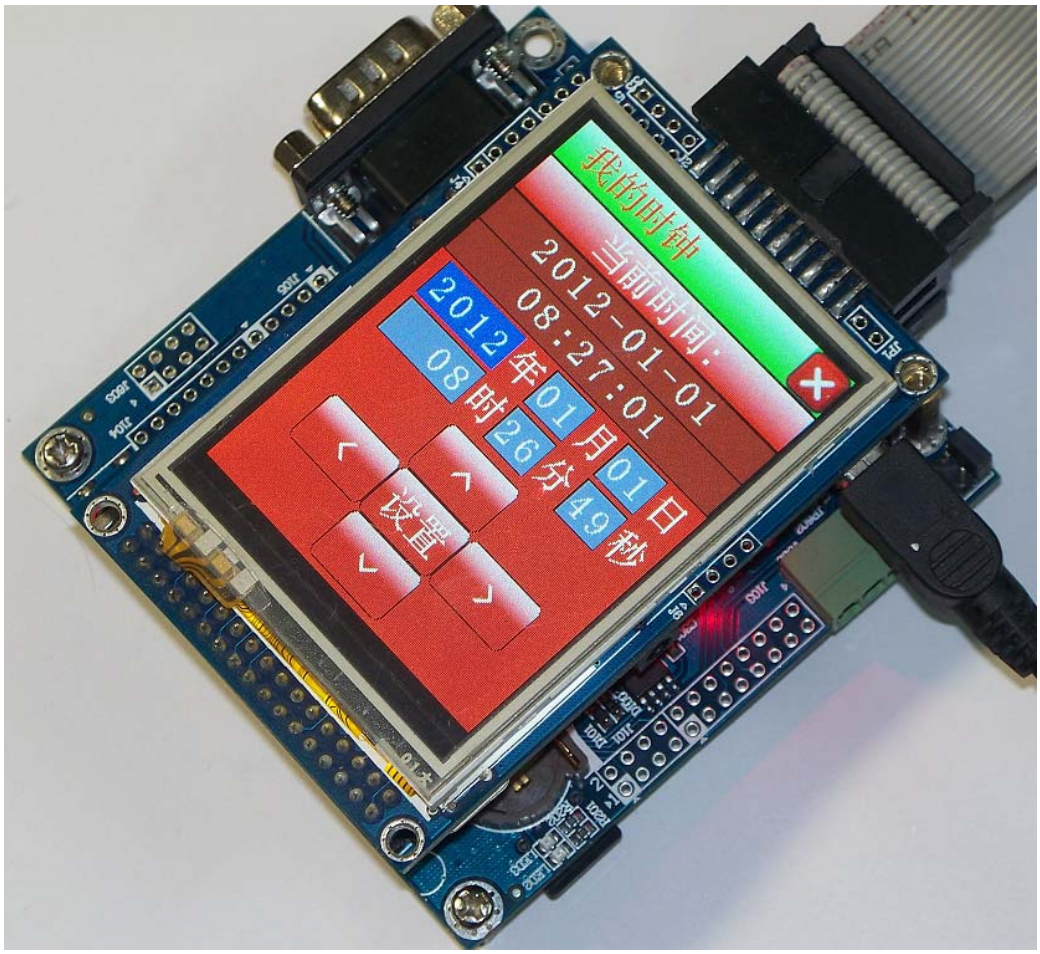
```
//下面对话框不自动关闭
    i=MessageBox("关闭计算机", "关闭", "重启", "取消", 0xffff);//按下按键才关闭消息框
   //返回值：0，1，2：对应按键序号
    if(i==0)
    {
        //MessageBox("计算机关闭中,请等待...", NULL, NULL, NULL, 2000);
        //下面对话框延时 3 秒自动关闭
        MessageBox("定时爆炸程序已启动,赶紧离开现场...如果两腿发软，请双手抱头立刻趴下", NULL, NULL, NULL, 3000);//对话框 3 秒自动关闭
    }
    else if(i==1)
    {
        MessageBox("计算机 2 分钟后爆炸，如果想解除爆炸请 Q 偶,QQ：958664258", NULL, NULL, NULL, 3000);//消息框 3 秒自动关闭
    }
    else if(i==2)
    {
        MessageBox("恭喜发财,红包拿来", NULL, NULL, NULL, 2000);//消息框 2 秒自动关闭
    }
//下面的消息框 3 秒后会自动关闭
```




画图
串口
关机
计算机2分钟后爆炸，如果想解除爆炸请Q偶. QQ: 958664258
2012-01-01 08:27:47 星期日

时钟窗口界面


我的时钟
当前时间：
2012-01-01
08:04:43
2012
年
01
月
01
日
08
时
04
分
32
秒
设置


我的时钟
当前时间：
2012-01-01
08:27:01
2012
年
01
月
01
日
08
时
26
分
49
秒
设置

本窗口界面下，左右移动选择要设置的日期时间，上下按键调整日期时间数值，中间【设置】按键保存日期时间，标题栏关闭按键返回到主窗口

tyw藏书

```
void Dlg_Clock(void)
{
 uint16 x,y;

 CWidgetsEvent *pWidgetsEvent;
 pFUN    event;
 u32 sec;

      //读取系统时间
      time(&sec);
      //保存系统当前时间方便调整
      memcpy(&setTime, mylocaltime(&sec), sizeof(setTime));
      memcpy(&dispTime, mylocaltime(&sec), sizeof(dispTime));

      GetCurWindows()->cur_widgets=0xffff;
      //清空窗口控件内存
      memset(CurWindowsWidgetsEvent, 0, sizeof(CurWindowsWidgetsEvent));
      pWidgetsEvent=CurWindowsWidgetsEvent;

      //FillSolidRect(0, 0, 240, 320, RGB565(16, 32, 22));

      //绘制标题栏
      //FillSolidRectChangeX(0, 0, 240, 32, GREEN, WHITE, 40, 2, 1);
      FillSolidRectChangeY(0, 0, 240, 34, GREEN, WHITE, 20, 8, 1);

      //标题名称
      LCD_WriteString24(16, (32-24)/2, RED, "我的时钟");

      //标题栏.退出键
      pWidgetsEvent->widgets= Button(240-1-32, 1, 32, 32, RED,      ALIGN_CENTER, 0, WHITE, "",
0xBD000);//WHITE
      //添加退出按键单击执行事件
      pWidgetsEvent->event=ExitDlg_Clock;
      pWidgetsEvent++;

      y=34;//36;
      //FillSolidRect(2, 36, 240-2*2, 100, BLUE);
      //FillSolidRectChangeY(2, 36, 240-2*2, 100, BLUE, WHITE, 20, 16, 2);
      //FillSolidRectChangeY(2, 36, 240-2*2, 92, RED, WHITE, 255, 0, 1000);
      FillSolidRectChangeY(0, y, 240, 32, RED, WHITE, 20, 8, 1);

      LCD_WriteString24(64, y+4, WHITE, "当前时间:");//WHITE
```

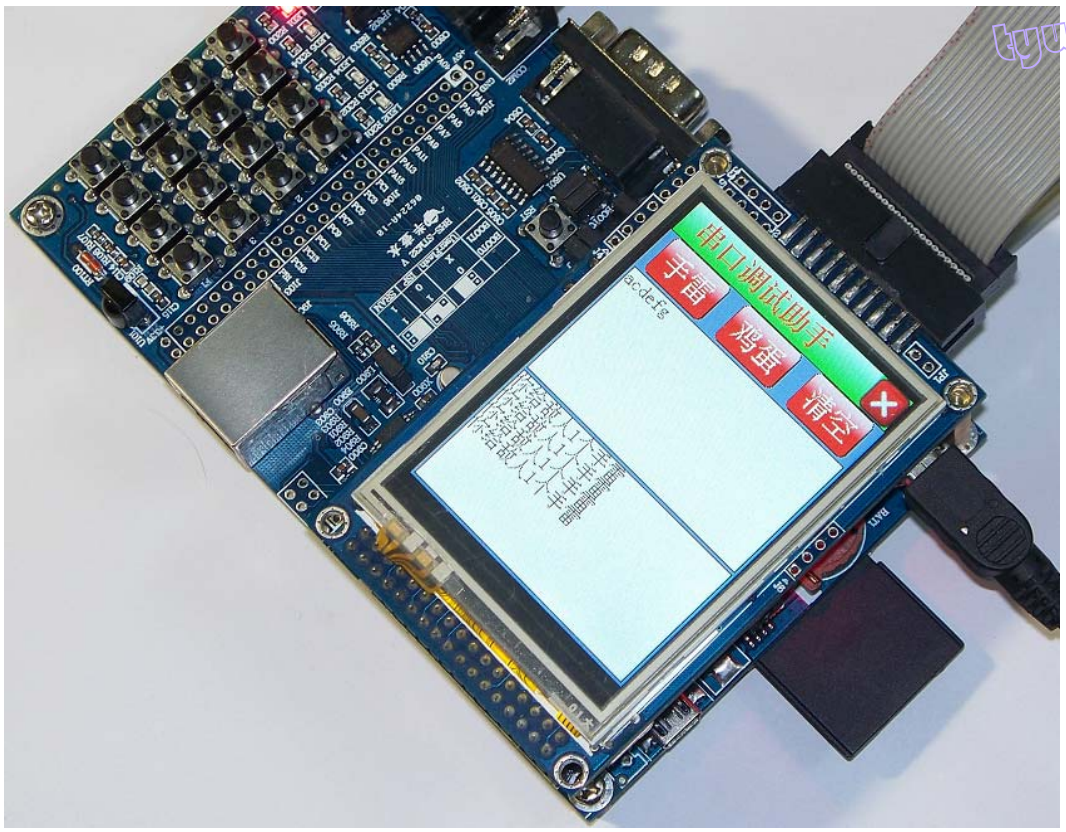
```
    y+=32;
    //年月日时分秒    ALIGN_LEFT, ALIGN_CENTER, ALIGN_RIGHT //RGB565(16, 32, 22)//RGB565(20, 8, 2)
    //pWidgetsEvent ->widgets= Edit(40, y, 160+2, 30, RED, ALIGN_LEFT, 24, WHITE, "2009-10-02");//WHITE
    //下面是编辑框控件输入参数：
    //按键:  X 坐标, Y 坐标，宽度，高度，背景色
    //文字中对齐；ALIGN_LEFT=左对齐;ALIGN_CENTER=中对齐;ALIGN_RIGHT=右对齐,
    //文字字体，目前支持 16，24 点阵，文字颜色，文字内容
    //文字内容

    //添加编辑框控件，该编辑框显示日期
    pWidgetsEvent->widgets= EditMul(0, y, 240, 30, RGB565(20, 6, 1), ALIGN_CENTER, 24, WHITE, "");//"2008-10-02");//WHITE
    pWidgetsEvent->event=NULL;
    pWidgetsEvent++;

    //添加编辑框控件，该编辑框显示时间
    y+=30-1;
    //pWidgetsEvent ->widgets= Edit(40, y, 160+2, 30, RED, ALIGN_LEFT, 24, WHITE, "<22:10:12>");//WHITE
    pWidgetsEvent ->widgets= EditMul(0, y, 240, 30, RGB565(20, 6, 1), ALIGN_CENTER, 24, WHITE, "");//"22:10:12");//WHITE
    pWidgetsEvent->event=NULL;
    pWidgetsEvent++;

    y+=30;
    //-------------------------------------------------------------------------------
//      y=36+100;
//      //FillSolidRectChangeY(2, 36+100, 240-2*2, 320-36-100-2, RED, WHITE, 20, 16, 4);
//      FillSolidRectChangeY(2, y, 240-2*2, 320-36-100-2, RED, WHITE, 255, 0, 1000);
//      FillSolidRectChangeY(2, y, 240-2*2, 32, GREEN, WHITE, 20, 8, 1);
//
//      LCD_WriteString24(40, y+2, BLUE, "重新设置时间:");//RED

    //-------------------------------------------------------------------------------
    //y+=34+92;
    //FillSolidRectChangeY(0, y, 240, 320-y-2, RED, WHITE, 255, 0, 1000);
    FillSolidRect(0, y, 240, 320-y, RED);

    x=4;
    //y+=32+4;

    //FillSolidRectChangeY(4, y, 240-4*2, 34, BLUE, WHITE, 255, 0, 1000);
```

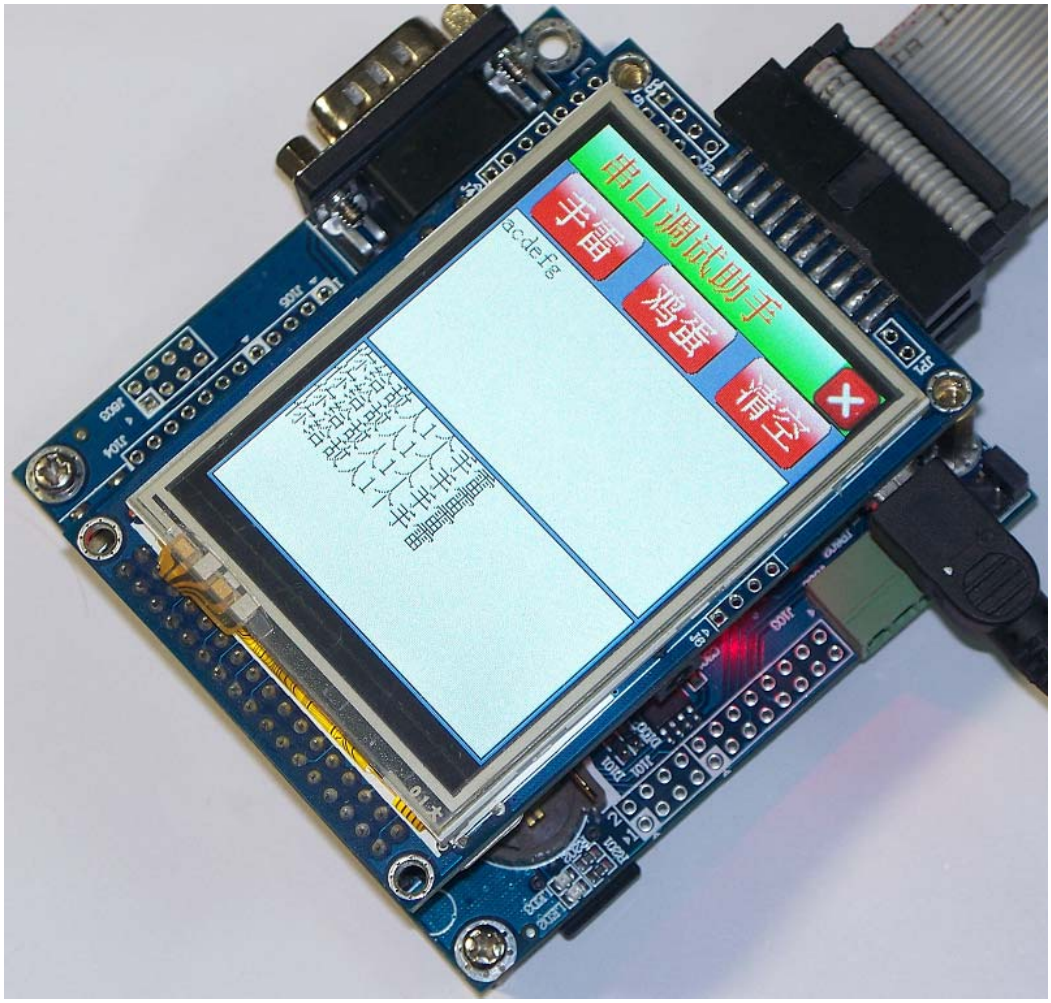
```
x=10;
y+=2;
//添加年编辑框,ascii 24 点阵=16x24
sprintf(bufy, "%4u", setTime.tm_year+1900);
pWidgetsEvent ->widgets= Edit(x, y, 70, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufy);
pWidgetsEvent ->widgets->tab=1;
pWidgetsEvent++ ->event=NULL;

x+=70+2;

LCD_WriteString24(x, y+2, WHITE, "年");//RED
x+=24+2;

//添加月编辑框
sprintf(bufM, "%02u", setTime.tm_mon+1);
pWidgetsEvent ->widgets= Edit(x, y, 36, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufM);
pWidgetsEvent ->widgets->tab=1;
pWidgetsEvent++ ->event=NULL;
x+=36+2;

LCD_WriteString24(x, y+2, WHITE, "月");//RED
x+=24+2;

//添加日编辑框
sprintf(bufd, "%02u", setTime.tm_mday);
pWidgetsEvent ->widgets= Edit(x, y, 36, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufd);
pWidgetsEvent ->widgets->tab=1;
pWidgetsEvent++ ->event=NULL;
x+=36+2;

LCD_WriteString24(x, y+2, WHITE, "日");//RED
//-------------------------------------------------------------------------------------

x=4;
y+=34-2-1;

//FillSolidRectChangeY(4, y, 240-4*2, 34, BLUE, WHITE, 255, 0, 1000);

x=10;//x=10+34;
y+=2;

//添加时编辑框,ascii 24 点阵=16x24
sprintf(bufh, "%02u", setTime.tm_hour);
pWidgetsEvent ->widgets= Edit(x, y, 36+34, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufh);
pWidgetsEvent ->widgets->tab=1;
```

```
    pWidgetsEvent++ ->event=NULL;
    x+=36+34+2;

    LCD_WriteString24(x, y+2, WHITE, "时");//RED
    x+=24+2;

    //添加分编辑框
    sprintf(bufm, "%02u", setTime.tm_min);
    pWidgetsEvent ->widgets= Edit(x, y, 36, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufm);
    pWidgetsEvent ->widgets->tab=1;
    pWidgetsEvent++ ->event=NULL;
    x+=36+2;

    LCD_WriteString24(x, y+2, WHITE, "分");//RED
    x+=24+2;

    //添加秒编辑框
    sprintf(bufs, "%02u", setTime.tm_sec);
    pWidgetsEvent ->widgets= Edit(x, y, 36, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE, bufs);
    pWidgetsEvent ->widgets->tab=1;
    pWidgetsEvent++ ->event=NULL;
    x+=36+2;

    LCD_WriteString24(x, y+2, WHITE, "秒");//RED
    x+=24+2;
    //-------------------------------------------------------------------------------------------
    //设置按键
    //添加向上按键
    y=196;
    x=4+16;
    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\1", 0);//
上
    //添加日期时间增加事件
    pWidgetsEvent->event=AddTime; //NULL;
    pWidgetsEvent++;

    //添加向左按键
    y+=40;
    pWidgetsEvent ->widgets= Button(x+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\3", 0);//左
    //添加选择上一个参数事件
    pWidgetsEvent++ ->event=PreviousWidgets;

    //添加【设置】按键
    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "设置",
0);//
```

```
    //设置并保存时间
    pWidgetsEvent->event=UpdataTime; //NULL;
    pWidgetsEvent++;

    //添加向右按键
    pWidgetsEvent ->widgets= Button(x+64+4+64+4-4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE,
"\4", 0);//右
    //添加选择下一个参数事件
    pWidgetsEvent++ ->event=NextWidgets;

    //添加向下按键
    y+=40;
    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\2", 0);//
下
    //添加日期时间减少事件
    pWidgetsEvent->event=DecTime; //NULL;
    pWidgetsEvent++;
    while(1)
    {
        //下面是消息循环处理。
        //等待消息，该消息来自于触摸屏事件
        //触摸屏检测到单击按键事件后将按键对应的事件发送到该任务
        if( os_mbx_wait (EventMailbox, (void **)&event, 0)==OS_R_OK )
        {
            event();
            if(CloseFlag)
            {
                CloseFlag=0;
                break;
            }
        }
        //更新显示时间
        displayTime();

        //os_tsk_pass();
        os_dly_wait(50/OS_TIME);
    }
}
```

串口调试助手串口界面





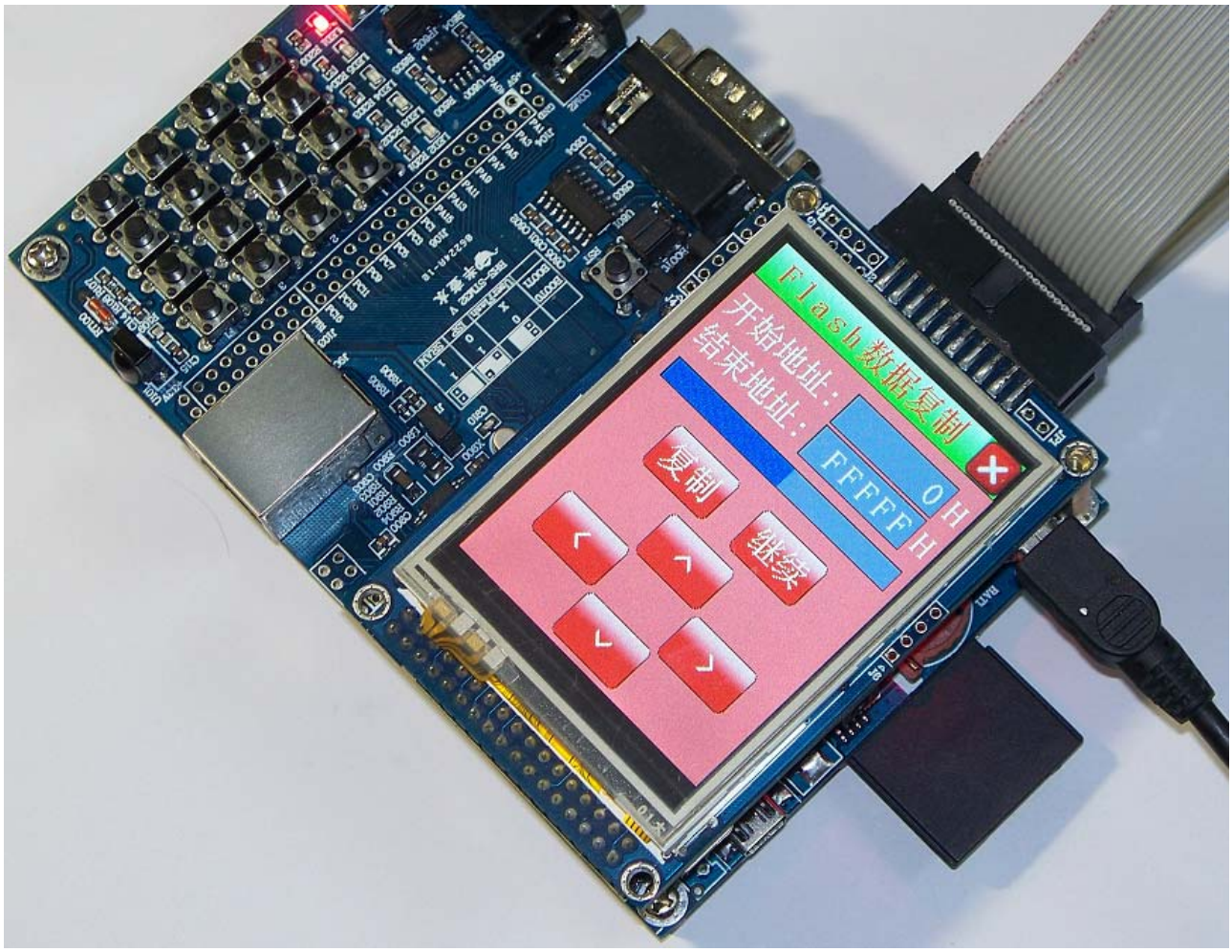
```
void Dlg_Com(void)
{
// uint16 x,y;

  CWidgetsEvent *pWidgetsEvent;
  pFUN    event;
  char *receive;

      //清空显示区域
      //清空接收窗口
      strcpy((char*)textReceive, "");
      //清空发送窗口
      strcpy(textSend, "");

      //清空窗口控件内存
      memset(CurWindowsWidgetsEvent, 0, sizeof(CurWindowsWidgetsEvent));
      pWidgetsEvent=CurWindowsWidgetsEvent;

      //绘制背景
      FillSolidRect(0, 0, 240, 320, RGB565(16, 32, 22));

      //绘制顶部标题栏
      //FillSolidRectChangeX(0, 0, 240, 32, GREEN, WHITE, 40, 2, 1);
      FillSolidRectChangeY(0, 0, 240, 34, GREEN, WHITE, 20, 8, 1);
      //标题名称
      LCD_WriteString24(16, (32-24)/2, RED, "串口调试助手");

      //下面是按键输入参数：
      //按键:   X 坐标, Y 坐标，宽度，高度，背景色
      //文字中对齐；ALIGN_LEFT=左对齐;ALIGN_CENTER=中对齐;ALIGN_RIGHT=右对齐,
      //文字字体，目前支持 16，24 点阵，文字颜色，文字内容
      //字体==0 将不显示文字
      //背景图片地址，0=不使用背景图片，其他=背景图片在 FLASH 中的物理地址

      //标题栏.退出键
      pWidgetsEvent ->widgets= Button(240-1-32, 1, 32, 32, RED,      ALIGN_CENTER, 0, WHITE, "",
0xBD000);//WHITE
      //添加退出按键单击执行事件
      pWidgetsEvent++ ->event=ExitDlg_Com;

      //添加按键
      pWidgetsEvent ->widgets= Button(8, 34, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "手雷", 0);//发
送
      //添加事件
      pWidgetsEvent++ ->event=ComSend;
```

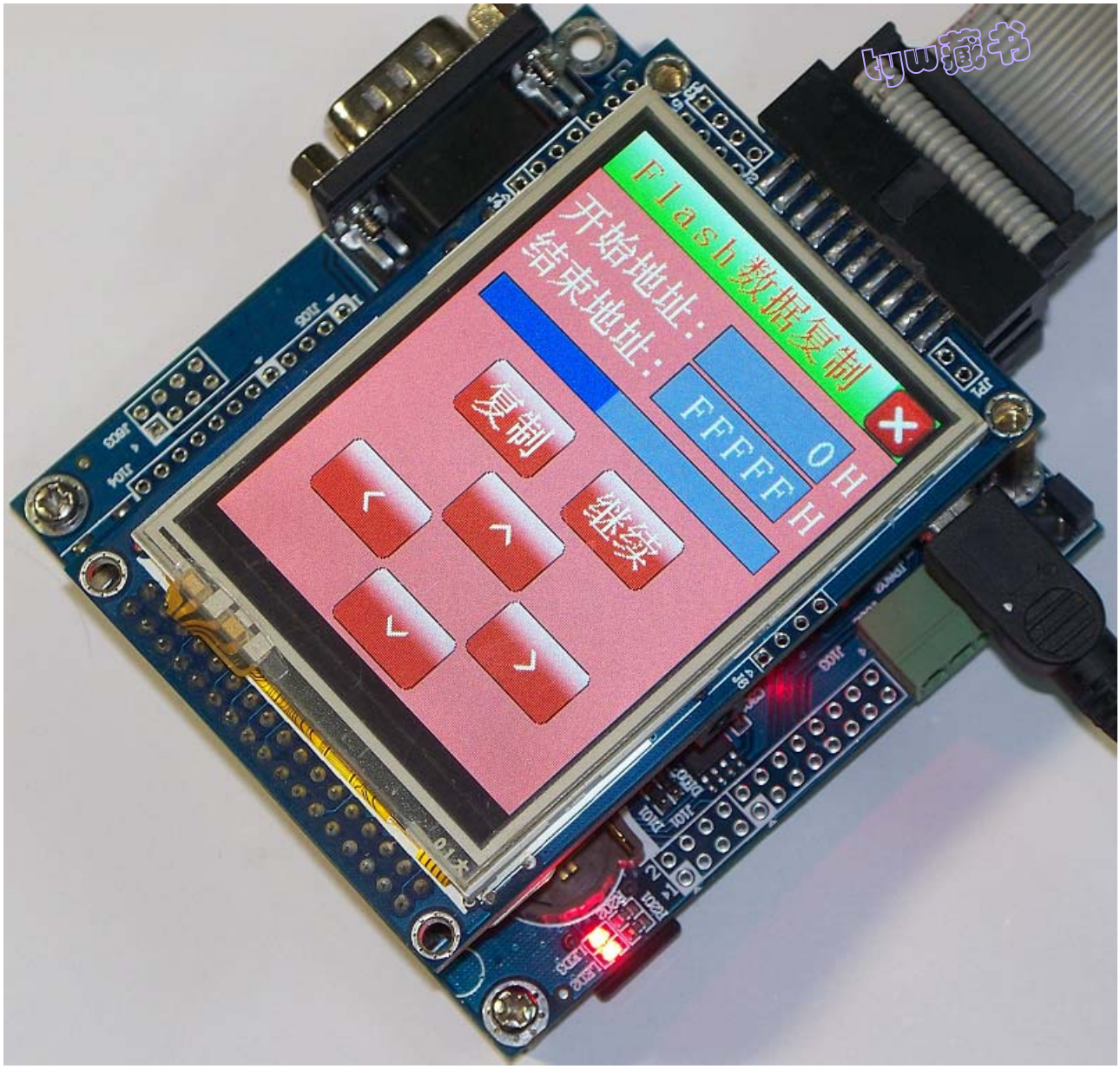
```
    //添加按键
    pWidgetsEvent ->widgets= Button(8+64+16, 34, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "鸡蛋", 0);//测试
    //添加事件
    pWidgetsEvent++ ->event=ComDebug;

    //添加按键
    pWidgetsEvent ->widgets= Button(8+64+16+64+16, 34, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "清空", 0);
    //添加事件
    pWidgetsEvent++ ->event=ClrDisplay;


    //FillSolidRect(2, 76, 240-2*2, 320-76-2, WHITE);

    //pWidgetsEvent ->widgets= Edit(2, 76, 240-2*2, 320-76-2, WHITE, ALIGN_LEFT, 16, BLACK, "acdefg");
    //pWidgetsEvent++ ->event=NULL;
    //每行能显示 16 点阵 29 个 ascii,14 个汉字，总共 7 行　；　7*29=203 ascii, 7*14=98 汉字
    //pWidgetsEvent ->widgets= EditMul(2, 76, 240-2*2, (320-76-2-2)/2, WHITE, ALIGN_LEFT, 16, BLACK, "acdefg 串口调试助手串口调试助手串口调试助手串口调试助手串口调试助手串口调试助手");

    //添加多行编辑框控件
    pWidgetsEvent ->widgets= EditMul(2, 76, 240-2*2, (320-76-2-2)/2, WHITE, ALIGN_LEFT, 16, BLACK, "acdefg");
    editReceive=pWidgetsEvent ->widgets;
    pWidgetsEvent++ ->event=NULL;

    //添加多行编辑框控件
    pWidgetsEvent ->widgets= EditMul(2, 76+2+(320-76-2-2)/2, 240-2*2, (320-76-2-2)/2, WHITE, ALIGN_LEFT, 16, BLACK, "1234");
    editSend=pWidgetsEvent ->widgets;
    pWidgetsEvent++ ->event=NULL;

//      SetTouchWidgetsEvent(CurWindowsWidgetsEvent, 6 );

    SetCOM_Debug(1);//进入串口调试模式
    while(1)
    {
        //os_mbx_wait (EventMailbox, (void **)&event, 0xffff);
        //event();
        //下面是消息循环处理。
        //等待消息，该消息来自于触摸屏事件
        //触摸屏检测到单击按键事件后将按键对应的事件发送到该任务
```

```
        if( os_mbx_wait (EventMailbox, (void **)&event, 0)==OS_R_OK )
        {
            //执行对应事件
            event();
            if(CloseFlag)
            {
                CloseFlag=0;
                break;
            }
        }
        //处理串口接收数据，将接收到的数据显示在串口中
        if( os_mbx_wait (COMReceiveMailbox, (void **)&receive, 0)==OS_R_OK )
        {
            //设置编辑框文字
            EditSetWindowText(editReceive, (char*)textReceive);
        }
        os_tsk_pass();
    }
}
```

lyw藏书

FLASH数据复制窗口






```
void Dlg_FlashCopy(void)
{
  uint16 x,y;

  CWidgetsEvent *pWidgetsEvent;
  pFUN    event;

      //当前编辑参数不选中
      GetCurWindows()->cur_widgets=0xffff;

      //清空窗口控件内存
      memset(CurWindowsWidgetsEvent, 0, sizeof(CurWindowsWidgetsEvent));
      pWidgetsEvent=CurWindowsWidgetsEvent;
```

```
    //绘制顶部标题栏
    //FillSolidRect(0, 0, 240, 320, RGB565(16, 32, 22));
    FillSolidRectChangeY(0, 0, 240, 34, GREEN, WHITE, 20, 8, 1);
    //标题名称
    LCD_WriteString24(16, (32-24)/2, RED, "Flash 数据复制");

    //FillSolidRect(0, 34, 240, 320-34, 0xfb2c);
    //SetDlgBackColor(0xfb2c);//有消息框的对话框一定要设置背景颜色

    //绘制背景颜色
    FillSolidRect(0, 34, 240, 320-34, RED);
    //有消息框的对话框一定要设置背景颜色
    SetDlgBackColor(RED);

    //下面是按键输入参数：
    //按键:  X 坐标, Y 坐标，宽度，高度，背景色
    //文字中对齐；ALIGN_LEFT=左对齐;ALIGN_CENTER=中对齐;ALIGN_RIGHT=右对齐,
    //文字字体，目前支持 16，24 点阵，文字颜色，文字内容
    //字体==0 将不显示文字
    //背景图片地址，0=不使用背景图片，其他=背景图片在 FLASH 中的物理地址

    //标题栏.退出键
    pWidgetsEvent ->widgets= Button(240-1-32, 1, 32, 32, RED,      ALIGN_CENTER, 0, WHITE, "",
0xBD000);//WHITE
    pWidgetsEvent++ ->event=ExitDlg_FlashCopy;

    y=36+8;
    LCD_WriteString24(4, y+2, WHITE, "开始地址:"); //LCD_WriteString24(13, y+2, WHITE, "开始地址:");
    LCD_WriteString24(240-18, y+2, WHITE, "H");
    sprintf(bufStart, "%6X", StartFlashAddress);
    //添加单行编辑框
    pWidgetsEvent ->widgets= Edit(13+2+112-9, y, 100, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE,
bufStart);//"000000");
    pWidgetsEvent ->widgets->tab=1;
    pWidgetsEvent++ ->event=NULL;

    y+=30+2;
    LCD_WriteString24(4, y+2, WHITE, "结束地址:"); //LCD_WriteString24(13, y+2, WHITE, "结束地址:");
    LCD_WriteString24(240-18, y+2, WHITE, "H");
    sprintf(bufEnd, "%6X", EndFlashAddress);
    //添加单行编辑框
    pWidgetsEvent ->widgets= Edit(13+2+112-9, y, 100, 30, RGB565(16, 32, 22), ALIGN_RIGHT, 24, WHITE,
bufEnd);//"100000");
    pWidgetsEvent ->widgets->tab=1;
    pWidgetsEvent++ ->event=NULL;
```

tyw藏书

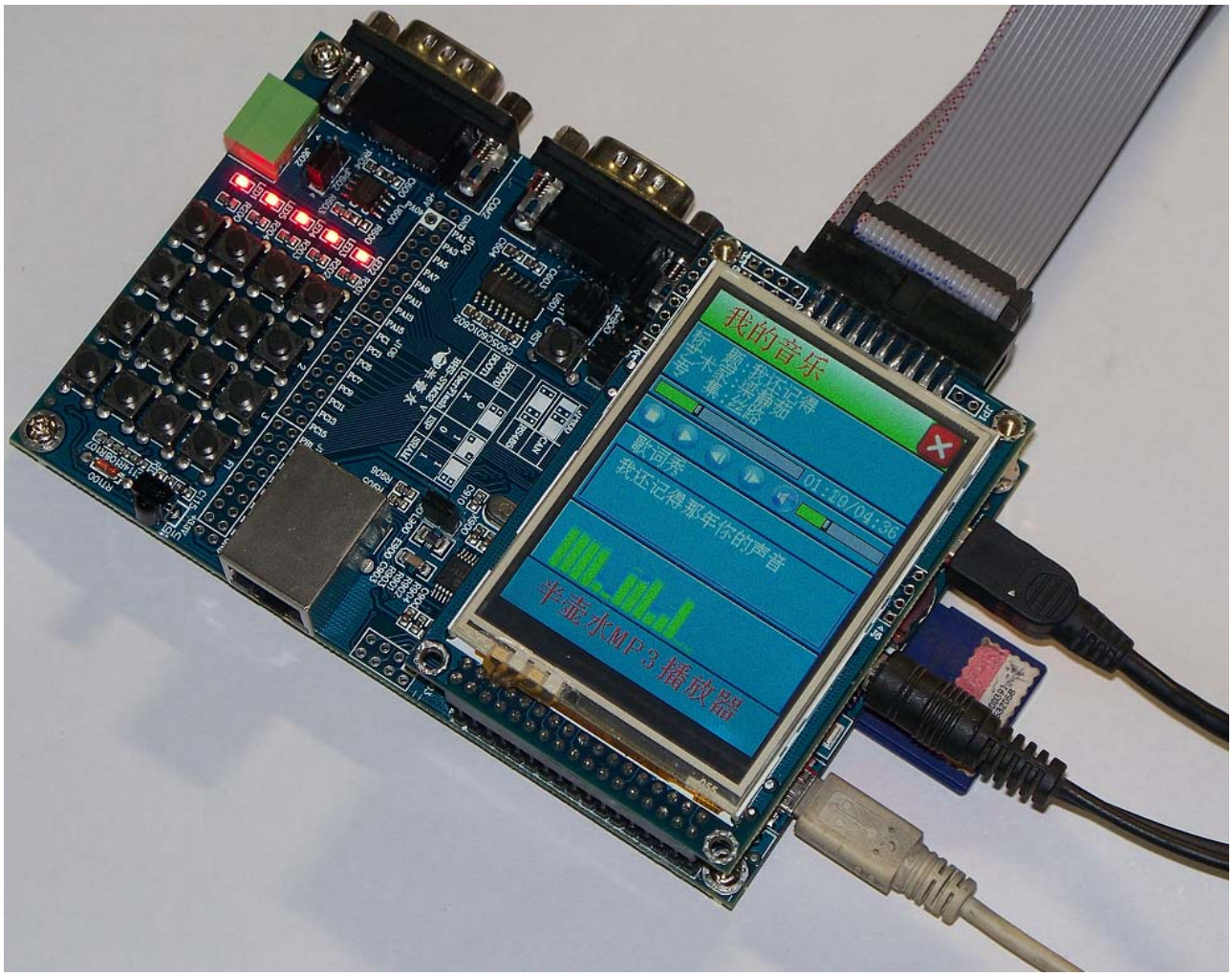
```
    y+=30+8;

    //下面是进度条参数
    //    uint16 X;          // X 坐标
    //    uint16 Y;          // Y 坐标
    //    uint16 Width;      //长度
    //    uint16 Height;     //高度
    //    uint16 Color;      //背景颜色
    //    uint16 FontColor;  //前景颜色
    //    uint16 range;        //进度条范围
    //    uint16 pos;        //当前进度
    //    u16 step;          //步进值


    //添加进度条控件
    //进度条
    pWidgetsEvent ->widgets= (CWidgets*)ProgressCtrl(8, y, 240-8*2, 20, RGB565(16, 32, 22), BLUE, 100, 0,
1);
    pProgressCtrl_FlashCopy=(CProgressCtrl*)pWidgetsEvent ->widgets;
    pWidgetsEvent++ ->event=NULL;

    x=4+16;
    y=196-40-8;
    //添加按键
    pWidgetsEvent ->widgets= Button(20+x+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "复制", 0);
    //添加事件
    pWidgetsEvent ->event=StartFlashCopy; //NULL;
    pWidgetsEvent++;

//    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "暂停",
0);
//    pWidgetsEvent->event=PauseFlashCopy;
//    pWidgetsEvent++;

    //添加按键
    pWidgetsEvent ->widgets= Button(x+64+4+64+4-4-20, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE,
"暂停", 0);//继续
    pButton_PauseContinue=pWidgetsEvent->widgets;
    //添加事件
    pWidgetsEvent->event=PauseContinue;
    pWidgetsEvent++;

    //-------------------------------------------------------------------------------------------
    //设置按键
```

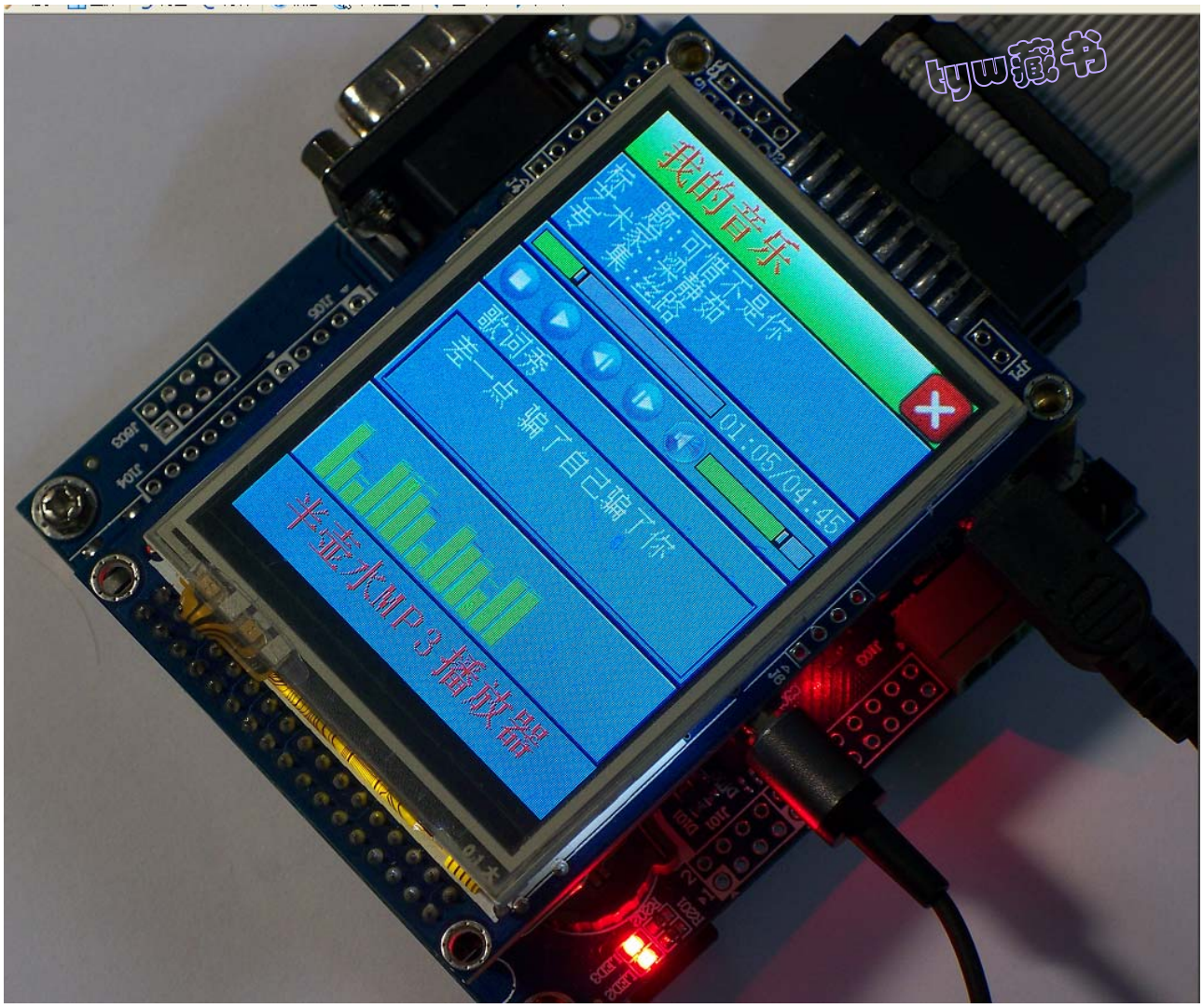
```
    x=4+16;
    y=196;
    //添加按键
    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\1", 0);//
上
    //添加事件
    pWidgetsEvent->event=AddAddress; //NULL;
    pWidgetsEvent++;

    y+=40;
    //添加按键
    pWidgetsEvent ->widgets= Button(x+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\3", 0);//左
    //添加事件
    pWidgetsEvent->event=PreviousWidgets;
    pWidgetsEvent++;

    //添加按键
    pWidgetsEvent ->widgets= Button(x+64+4+64+4-4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE,
"\4", 0);//右
    //添加事件
    pWidgetsEvent->event=NextWidgets;
    pWidgetsEvent++;

    y+=40;
    //添加按键
    pWidgetsEvent ->widgets= Button(x+64+4, y, 64, 40, RED,      ALIGN_CENTER, 24, WHITE, "\2", 0);//
下
    //添加事件
    pWidgetsEvent->event=DecAddress; //NULL;
    pWidgetsEvent++;

    while(1)
    {
        //下面是消息循环处理。
        //等待消息，该消息来自于触摸屏事件
        //触摸屏检测到单击按键事件后将按键对应的事件发送到该任务
        os_mbx_wait (EventMailbox, (void **)&event, 0xffff);
        event();
        if(CloseFlag)
        {
            CloseFlag=0;
            break;
        }
    }
}
```

BHS-STM32 实验 50-BHS-GUI-FATFS-MP3

lyw藏书

本例子是基于半壶水原创的 BHS-GUI+RTX+FATFS 文件系统的 MP3 播放器，也是 BHS-GUI 的一个应用例子。将资源文件夹的所有文件拷贝到 SD 卡根目录，播放器功能：
播放、停止、上一首、下一首、音量调节、静音、播放进度显示、歌词显示（部分从网络下载的歌词需要使用我提供的格式化工具格式化下才同步显示）
资源说明：sys-系统文件夹，存放系统使用的图片
mp3-歌曲文件夹，lrc-歌词文件夹，歌词名必须与歌曲同名才能显示





说明：其他操作系统和 GUI 的例子本手册不做说明，详细信息请参考例程