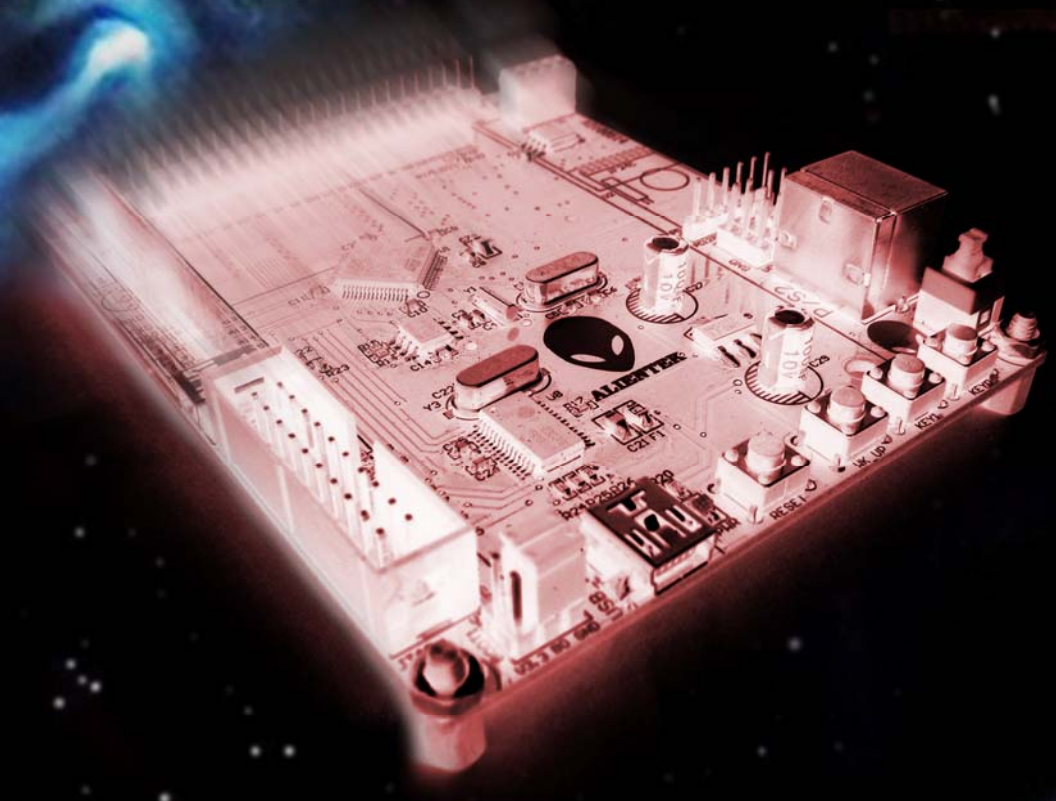
ALIENTEK

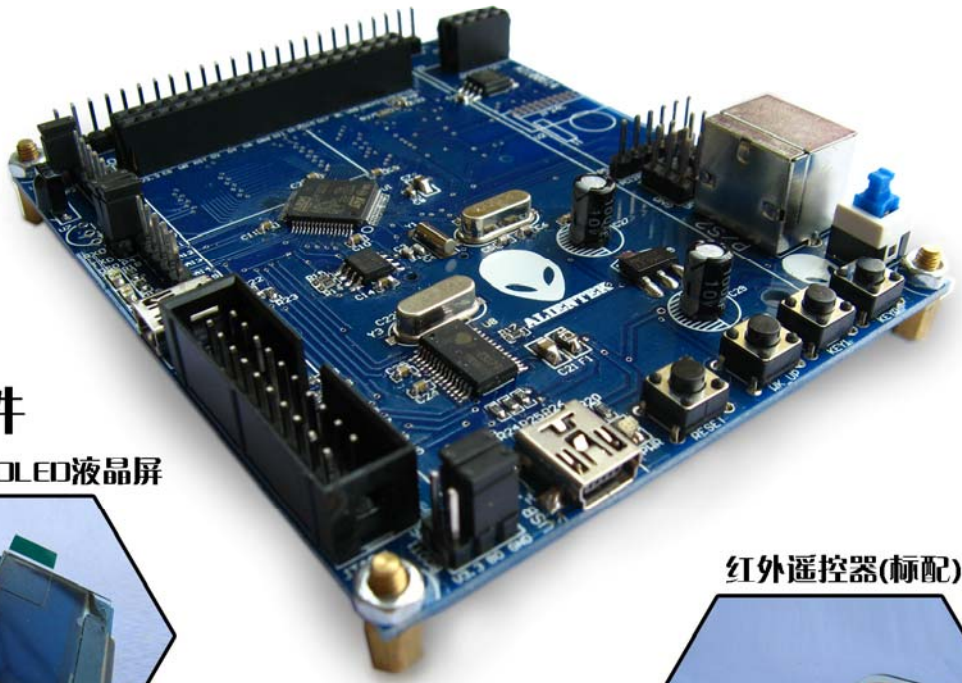
STM32不完全手册

ALIENTEK MiniSTM32开发板



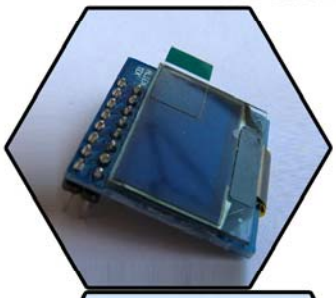
玩转STM32

ALIENTEK MiniSTM32开发板



可选配件

128x64OLED液晶屏



红外遥控器(标配)

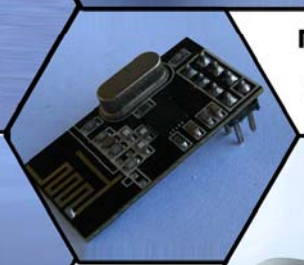
JF24C

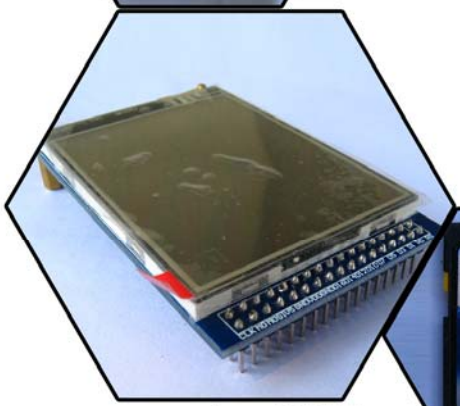
2.4G无线模块

SD卡

NRF2401

2.4G无线模块





2.4/2.8寸TFT液晶屏

PS2鼠标

JLINK

ALIENTEK MiniSTM32开发板的特点包括：

1） 小巧。整个板子尺寸为8cm*10cm*2cm（包括液晶，但不计算铜柱的高度）。

2） 灵活。板上除晶振外的所有的IO口全部引出，特别还有GPIOA和GPIOB的IO口是按顺序引出的，可以极大的方便大家扩展及使用。

3） 资源丰富。板载十多种外设及接口，让你畅游STM32。

4） 教程齐全。共计29个实例，各个实例代码均有详细注释（ucGUI实例除外）。并配有450多页，10多万字的教程，让你彻底玩转STM32。

目　　录

前言

STM32 是基于 ARM Cortex-M3 内核的 32 位处理器，具有杰出的功耗控制以及众多的外设，最重要的是其性价比。而且 STM32 官方在国内的宣传也是做得非常不错，而且针对 8 位机市场推出了 STM8。

本人在 08 年初开始接触 STM32，之前也用过 51，用过 AVR ，对这几款芯片还是比较了解，下面就来看看我们为什么要选择 STM32。

AVR 是很成功的一款芯片，功耗低，性能强。较之前的 51，性能提升了好几个档次。如果一个初学者，学完了 51，再学 AVR，肯定就会对 AVR 爱不释手。我也是这么过来的，AVR 对当时的我来说可谓是要啥有啥。所以从大二开始，一直用到毕业。PIC 据说也不错，但是很遗憾，我们学校，没看到几个搞 PIC 的，大概是因为这个东西价格太高了，对我们学生来说，基本上不考虑。当然，有钱人例外。

其次，AVR 的下载也是很方便的。和 51 的可以通用。这其中双龙电子对 avr 的支持，至少在国内来说，爱特梅尔是要感谢他们的。至于 STM8，我没有详细了解，我估计他存在的目的，就是要把 8 位市场给占领了。他最大的对手，估计就是 AVR 和 51 了。目前 AVR 的局势，岌岌可危，像目前这个情况，持续下去，很快就可以退出历史舞台了。

STM8 目前最低端的是 STM8S103F2 最少引脚数是 20 脚的，淘宝最低售价是 3.5 元，AVR 同样配置(仅仅指 SRAM 和 FLASH)的芯片，价格在 3 块左右。基本不分上下。其他功能方面也很相似。STM8 最高端的，STM8S208MB，淘宝价格在 15 元左右，而同样配置的 avr 芯片只有 MEGA128 了，还少了 2k 的 ram 和 CAN 控制器，不过多了总线控制器。但是 MEGA128 的价格，在 30 块钱左右，这就毫无竞争力了。这样的价格，STM32F103 都能买到很好的芯片了。15 块钱左右，基本只够买个 MEGE32，而 mega32 和 STM8S208MB 相比，显然差距很明显。以目前的局势（AVR 价格居高不下），我建议，没学 AVR 的就可以跳过了，学过的，就赶紧选择新的 MCU。不过 STM8 的下载，好像不如 AVR 那么方便，这方面，我没有了解过，这里就不评论了。

高端市场，ST 最近几年，对 STM32 的推广，可谓是不遗余力。效果也是很显著的。我是阴错阳差，在 08 年开始学 STM32，而且 STM32 的价格，现在也很便宜，当时，STM32F103RBT6 也就 30 块钱，外设功能是很强悍了：128K FLASH、20K SRAM、USB、CAN、12 位 ADC、SPI、IIC、TIMER、USART、RTC、DMA 等，基本上，你能想到的，它都有了。显然，此时的 MEGA128 已经毫无竞争力了。现在 STM32 低配置的芯片，STM32F101C4，16K FLASH，4K SRAM，价格在 10 块钱左右。F103 较低配置的 STM32F103C8，也卖到了 13 块钱一个，64K FLASH，20K SRAM，带 USB 和 CAN。单从这 2 个数据，就能说明很多问题了。

再说 LM3S，应该是和 STM32 一同推上市场的，至少不会比 STM32 晚，据说当时敢尝 CM3 螃蟹的就流名和 ST。周立功还选择了推流明，后面也不知道什么原因，一直没见流明起来，可能周立功和流明，都有错吧(脑子被驴踢了可能)。却见 STM32 是打得红红火火。如今流明(被 TI 收购了)已经没办法和 STM32 竞争了。估计老周也很郁闷吧，当时怎么就没推 STM32 呢?呵呵。

继续说 STM32，STM32 现在推出的型号，从最低的 10 块钱的，到最高端的 STM32F103ZET6，价格也不过 40 元不到。其中包括的型号，有 50 种之多。用户可以随便选择满足自己需求的产品。高端方面，STM32 还推了 F105/F107 系列。强化了 USB 和网络的功能。

这方面 AVR32，从淘宝上看到的最低价格是 30 元左右，具体配置没去看了。AVR32 同 STM32 最大的缺点就是下载程序不方便，人家得专门为你做个下载器，或者从你那里买，OURAVR 论坛上也有人搞了下载器出来，还真佩服这些哥们，挺厉害的。

不过 STM32 呢？支持 JTAG，支持串口下载。 这就把学习 STM32 的门槛一下降低了，加上 KEIL 对 STM32 的支持，比学习 AVR 的门槛还低了。这就很快的培养了大批使用者。其次，STM32 的中文支持，做的也很到位，中文的《参考手册》，中文的《CM3 权威指南》，给用户提供了很大的便利。反观其他，LPC 和爱特梅尔由于之前不敢吃螃蟹，到现在他们的 CM3 构架芯片，都还在襁褓之中，LM3S 虽然和 ST 一起吃了螃蟹，不过没搞好，算是玩完了。所以，现在就剩下 ST 独领风骚了。

总结下来，STM32 具有价格低、功能强、使用简单、开发方便等几个很有利的优势，所以对有志于 32 位市场应用开发的人，都应该学学。

在应用上，我认为 STM32 设计的比较方便的地方有：

1） 复用 IO 口重映射功能。

由于有些复用功能可以重映射，使得在 STM32 的 PCB 设计的时候，方便很多。

2） 全部引脚都可以作为中断输入。

全部 IO 口都可以作为中断输入，这点比很多 ARM 好，当要使用中断的时候，随便那个 IO 口都可以，而不需要接到特定的几个脚上，这样极大的方便了设计，不论原理图设计还是 PCB 设计。

3） SWD 调试支持。

STM32 支持 SWD 调试，只需要 2 跟 IO 线，就可以用来调试和下载代码，对引脚不多的型号尤其适用。

4） 串口下载程序。

串口下载代码很多 ARM 都具有这个功能，STM32 也保留了这一优秀设计，极大的降低了开发成本（不需要什么 JLINK、ULINK 之类的了，也不需要专门的下载器）。

本手册正是根据 STM32 的特点，结合实例逐一介绍其功能，取名为《STM32 不完全手册》有如下几个原因：

1， 本手册虽然逐一介绍了 STM32 的各个功能，但一些我不熟悉的介绍的比较简单，比如 USB 部分。

2， 对我不了解，没用过的 CAN 口，本手册并未提及。、

3， 本手册针对的是 STM32F103RBT6 的外设，对 STM32 其他系列的产品并不能完全兼顾，比如 ZE 系列有的 DAC 和 FSMC 部分，本手册并没有介绍。

鉴于以上三点，所以本手册取了这个名字。希望在以后的学习中，能不断丰富该手册，最终能给 STM32 学习者一点参考。本手册参考最多的是《STM32 参考手册》以及《CM3 权威指南》，很多地方摘自这两个文档，另外有很多资料及代码也都参考了网友的，有了他们的奉献及开源，才有了《STM32 不完全手册》。

此版本为 V1.0 修订版修改而来，修正了里面一些错误，并且针对新的板子 V1.8 版的新特性，做了相应修改。此版本定位 2.0 版本，相对于 V1.0 版本，更加翔实。

正点原子@ALIENTEKE

2011-1-11

QQ 群：32658778
电话：15902020353
Email：497610476@qq.com
开源电子网：www.openedv.com

第一章 硬件篇

实践出真知，要想学好 STM32，实验平台必不可少！本篇将详细介绍我们用来学习 STM32 的硬件平台：ALIENTEK MiniSTM32 开发板，通过该篇的介绍，您将了解到我们的学习平台 ALIENTEK MiniSTM32 开发板的功能及特点。

为了让读者更好的使用 ALIENTEK MiniSTM32 开发板，本篇还介绍了开发板的一些使用注意事项，请读者在使用开发板的时候一定要注意。

本章将分为如下几个部分：

1，ALIENTEK MiniSTM32 开发板简介；

2，ALIENTEK MiniSTM32 开发板硬件详解；

3，ALIENTEK MiniSTM32 开发板使用注意事项；

1.1 ALIENTEK MiniSTM32 开发板简介

ALIENTEK MiniSTM32 开发板是一款迷你型的开发板，小巧而不小气，简约而不简单。 她的外观尺寸只有 8cm*10cm 大小，如下图所示：

ALIENTEK MiniSTM32 开发板是的外观如图 1.1 所示：

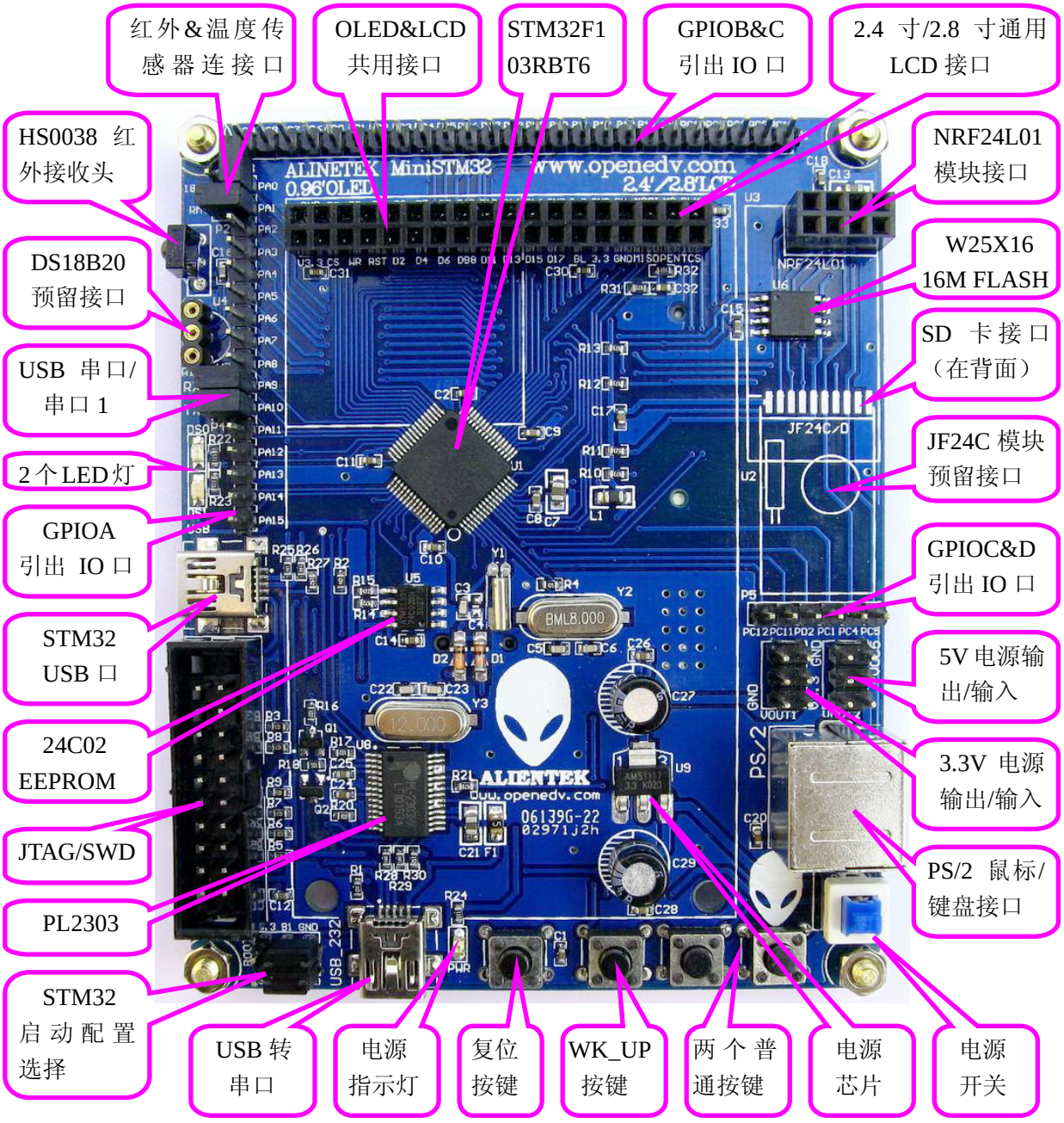


图 1.1 MiniSTM32 开发板外观图

这款 MiniSTM32 开发板，不求最全，但求最精！板子的设计充分考虑了成本与功能这两个矛盾面，再结合实际使用的经验及 STM32 的特点，最终确定了这样的设计。总体来说是该有的都有，不该有的坚决不要，可有可无的选择性价比最高的留下。

ALIENTEK MiniSTM32 开发板板载资源如下：

- CPU：STM32F103RBT6，LQFP64，FLASH:128K，SRAM：20K；
- 1 个标准的 JTAG/SWD 调试下载口
- 1 个电源指示灯（蓝色）
- 2 个状态指示灯（DS0：红色，DS1：绿色）
- 1 个红外接收头，配备一款小巧的红外遥控器
- 1 个 IIC 接口的 EEPROM 芯片，24C02，容量 256 字节
- 1 个 SPI FLASH 芯片，W25X16，容量为 2M 字节
- 1 个 DS18B20/DS1820 温度传感器预留接口
- 1 个标准的 2.4/2.8 寸 LCD 接口，支持触摸屏
- 1 个 OLED 模块接口
- 1 个 USB 串口，可用于程序下载和代码调试
- 1 个 USB SLAVE 接口，用于 USB 通信
- 1 个 SD 卡接口
- 1 个 PS/2 接口，可外接鼠标、键盘
- 1 组 5V 电源供应/接入口
- 1 组 3.3V 电源供应/接入口
- 1 个启动模式选择配置接口
- 2 个 2.4G 无线通信接口（24L01 和 JF24C）
- 1 个 RTC 后备电池座，并带电池
- 1 个复位按钮，可用于复位 MCU 和 LCD
- 3 个功能按钮，其中 WK_UP 兼具唤醒功能
- 1 个电源开关，控制整个板的电源
- 独创的一键下载功能
- 除晶振占用的 IO 口外，其余所有 IO 口全部引出，其中 GPIOA 和 GPIOB 按顺序引

从上面的板载资源可以看出，MiniSTM32 开发板的板载资源是很丰富的，加上灵活的设计，让您的开发变得更加简单。

ALIENTEK MiniSTM32 开发板的特点包括：

1）小巧。整个板子尺寸为 8cm*10cm*2cm（包括液晶，但不计算铜柱的高度）。

2）灵活。板上除晶振外的所有的 IO 口全部引出，特别还有 GPIOA 和 GPIOB 的 IO 口是按顺序引出的，可以极大的方便大家扩展及使用，另外板载独特的一键下载功能，避免了频繁设置 B0、B1 带来的麻烦，直接在电脑上一键下载。

3） 资源丰富。板载十多种外设及接口，可以充分挖掘 STM32 的潜质。

此开发板目前已经开始出售，大家可以在以下三个店铺买到：

淘宝店 1：http://shop62103354.taobao.com/

淘宝店 2：http://shop62057469.taobao.com/

接下来我们详细介绍 MiniSTM32 开发板的各个部分（上图中的标注部分），我们将按逆时针的顺序依次介绍。

1. HS0038 红外接收头

这是 ALIENTEK MiniSTM32 开发板板载的标准 38K 红外信号接收头，用于接收红外遥控器的信号，有了它，就可以用红外遥控器控制这款开发板了，也可以用来做红外解码等其他相关实验。ALIENTEK MiniSTM32 开发板是标配红外遥控器的，标配的红外遥控器外观如图 1.2 所示：

图 1.2 红外遥控器图片

关于该遥控器的使用，在 3.21 节会有详细介绍。

2. DS18B20 预留接口

这是 ALIENTEK MiniSTM32 开发板预留的数字温度传感器 DS18B20/DS1820 接口，采用的是镀金的圆孔母座。当要做 DS18B20 实验的时候，直接插到这个母座上即可，很方便。不过开发板不配 DS18B20，大家可以自行购买，插上就可以用的。同样 ALIENTEK 提供了 DS18B20 的相关例程。

3. USB 串口/串口 1

这是 USB 串口同 STM32F103RBT6 的串口 1 进行连接的地方，标号 RXD 和 TXD 是 USB 串口的 2 个数据口，而 PA9(TXD)和 PA10(RXD)则是 STM32 的串口 1 的两个数据口（复用功能下）。他们通过跳线帽对接，就可以和连接在一起了，从而实现 STM32 的程序下载以及串口通信。

设计成 USB 串口，是出于现在电脑上串口正在消失，尤其是笔记本，几乎清一色的没有串口。所以板载了 USB 串口可以方便大家下载代码和调试。而在板子上并没有直接连接在一起，则是出于实用方便的考虑。这样设计，您可以把 ALIENTEK MiniSTM32 开发板当成一个 USB 串口，来和其他板子通信，而其他板子的串口，也可以方便地接到 ALIENTEK MiniSTM32 开发板上。

4. LED 灯

这是 ALIENTEK MiniSTM32 开发板板载的两个 LED 灯，它们在开发板上的标号为：DS0 和 DS1。DS0 是红色的，DS1 是绿色的，主要是方便大家识别。一般的应用 2 个 LED 足够了，在调试代码的时候，使用 LED 来指示程序状态，是非常不错的一个辅助调试方法。ALIENTEK 开发板几乎每个实例都使用了 LED 来指示程序的运行状态。

这两个 LED 是直接连接在 IO 口上的，但是也可以被其他 IO 口使用，具体的使用实例请参考 3.1 节。

5. GPIOA 引出 IO 口

这是 ALIENTEK MiniSTM32 开发板 GPIOA 的引出排针，在开发板上的标号为 P3。ALIENTEK 开发板将所有的 IO 口（除了 2 个晶振占用的 4 个 IO 口）都用排针引出来了，而且 GPIOA 和 GPIOB 是按顺序引出的。按顺序引出，在很多时候能方便大家的实验和测试，比如外接带并行控制的器件，有了并行引出的排针，那么就可以很方便的通过这些排针连接到外部设备了。

将开发板的 IO 口全部引出，大家就可以用来外接其他模块等，不论调试还是功能扩展都是很方便的。

6. STM32 USB 口

这是 ALIENTEK MiniSTM32 开发板板载的一个 MiniUSB 头，用于 STM32 与电脑的 USB 通讯，此 MiniUSB 头在开发板上的标号为：USB，用于连接 STM32F103RBT6 自带的 USB，通过此 MiniUSB 头，开发板就可以和电脑进行 USB 通信了。开发板总共板载了 2 个 MiniUSB 头，一个用于接 USB 串口，连接 PL2303 芯片；另外一个用于 STM32 内带的 USB 连接。

开发板通过 MiniUSB 头供电，板载两个 MiniUSB 头（不共用），主要是考虑了使用的方便性，以及可以给板子提供更大的电流（两个 USB 都接上）这两个因素。

7. JTAG/SWD

这是 ALIENTEK MiniSTM32 开发板板载的 20 针标准 JTAG 调试口，在开发板上的标号为：JTAG。该 JTAG 口直接可以和 ULINK 或者 JLINK 或者 STLINK 等调试器（仿真器）连接，同时由于 STM32 支持 SWD 调试，这个 JTAG 口也可以用 SWD 模式来连接。

用标准的 JTAG 调试，需要占用 5 个 IO 口，很多时候，可能造成 IO 口不够用，而用 SWD 则只需要 2 个 IO 口，大大节约了 IO 数量，但他们达到的效果是一样的。所以在 ALIENTEK MiniSTM32 开发板上，调试下载，**强烈建议使用 SWD 模式!!!**

8. 24C02 EEPROM

这是 ALIENTEK MiniSTM32 开发板板载的 2Kbit（256 个字节）EEPROM ，型号为：24C02，用于掉电数据保存。因为 STM32 内部没有 EEPROM，所开发板外扩了 24C02，用于存储重要数据，也可以用来做 IIC 实验，及其他应用。该芯片直接挂在 STM32 的 IO 口上。

9. PL2303

这是 ALIENTEK MiniSTM32 开发板板载的 USB 串口芯片，型号为：PL2303。PL2303 是一颗 USB 转串口的芯片，用于 USB 串口。有了它我们就可以实现 USB 转串口，从而能 USB 下载代码，串口调试代码等。

10. STM32 启动配置选择

这是 ALIENTEK MiniSTM32 开发板板载的启动模式选择开关，在开发板上的标号为：BOOT1。STM32 有 BOOT0（B0）和 BOOT1（B1）两个启动选择引脚，用于选择复位后 STM32 的启动模式，作为开发板，这两个是必须的。在开发板上，我们通过跳线帽选择 STM32 的启动模式。关于启动模式的说明，请看 1.2.1 节。

11. USB 转串口

这是 ALIENTEK MiniSTM32 开发板板载的另外一个 MiniUSB 头，用于 USB 连接 PL2303 芯片，从而实现 USB 转串口，此接头在开发板上的丝印标号为：USB_232。此 MiniUSB 接头是开发板电源的主要提供口。

12. 电源指示灯

这是 ALIENTEK MiniSTM32 开发板板载的一颗蓝色的 LED，用于指示电源状态，在开发板上的标号为：PWR。在电源开启的时候（通过板上的电源开关控制），该灯会亮，否则不亮。通过这个 LED，可以判断开发板的上电情况。

13. 复位按键

这是 ALIENTEK MiniSTM32 开发板板载的复位按键，用于复位 STM32，还具有复位液晶的功能，因为液晶模块的复位引脚和 STM32 的复位引脚是连接在一起的，此按键在开发板上的标号为：RESET。当按下该键的时候，STM32 和液晶一并被复位。

14. WK_UP 按键

这是 ALIENTEK MiniSTM32 开发板板载的一个唤醒按键，该按键连接到 STM32 的 WAKE_UP（PA0）引脚，可用于待机模式下的唤醒，在不使用唤醒功能的时候，也可以做为普通按键输入使用，此按键在开发板上的标号为：WK_UP。

15. 两个普通按键

这是 ALIENTEK MiniSTM32 开发板板载的两个普通按键，可以用于人机交互的输入，这两个按键是直接连接在 STM32 的 IO 口上的，这两个按键在开发板上的标号分别为：KEY0、KEY1。

16. 电源芯片

这是 ALIENTEK MiniSTM32 开发板的电源芯片，型号为：AMS1117-3.3。因为 STM32 是 3.3V 供电的，所以我们需要将 USB 的 5V 电压转换为 3.3V，这个芯片就是将 5V 转换为 3.3V 的线性稳压芯片。

17. 电源开关

这是 ALIENTEK MiniSTM32 开发板板载的电源开关，此开关在开发板上的标号为：ON/OFF。该开关用于控制整个开发板的供电，如果切断，则整个开发板都将断电，电源指示灯（PWR）会随着此开关的状态而亮灭。

18. PS/2 鼠标/键盘接口

这是 ALIENTEK MiniSTM32 开发板板载的一个标准 PS/2 母头，用于连接电脑鼠标和键盘等 PS/2 设备，在开发板上的标号为：PS/2。

通过 PS/2 口，我们仅仅需要 2 个 IO 口，就可以扩展一个键盘，所以大家不必要对板上只有 3 个按键而感到担忧。ALIENTEK 提供了标准的鼠标驱动例程，方便大家学习 PS/2 协议。

19. 3.3V 电源输出/输入

这是 ALIENTEK MiniSTM32 开发板板载的一组 3.3V 电源输入输出排针（2*3），在开发板上的标号为：VOUT1。该排针用于给外部提供 3.3V 的电源，也可以用于从外部取 3.3V 的电源给板子供电。大家在实验的时候可能经常会为没有 3.3V 电源而苦恼不已，ALIENTEK 充分考虑到了大家需求，有了这组 3.3V 排针，您就可以很方便的拥有一个简单的 3.3V 电源（最大电流不能超过 500ma）。

20. 5V 电源输出/输入

这是 ALIENTEK MiniSTM32 开发板板载的一组 5V 电源输入输出排针（2*3），用于给外部提供 5V 的电源，也可以用于从外部取 5V 的电源给板子供电。同样大家在实验的时候可能经常会为没有 5V 电源而苦恼不已，有了 ALIENTEK MiniSTM32 开发板，您就可以很方便的拥有一个简单的 5V 电源（最大电流不能超过 500ma）。

21. GPIOC&D 引出 IO 口

这是 ALIENTEK MiniSTM32 开发板板载的 GPIOC 与 GPIOD 等 IO 口的引出排针，在开发板上的标号为：P5。我们可以用这些引出的 IO 口来连接外部模块，方便大家外接其他模块。

22 .JF24C/D 模块预留接口

这是 ALIENTEK MiniSTM32 开发板板载的 JF24C/D 预留接口。JF24C/D 是安阳新世纪开发的一款性价比很高的 2.4G 无线通信模块，该模块的价格在 10 元以内，1Mbps 的速率，以及 10db 的发射功率，可以应用在很多方面。通过我们预留的这个接口，大家买到模块直接焊上去

即可使用。

23. SD 卡接口

这是 ALIENTEK MiniSTM32 开发板板载的 SD 卡接口。SD 卡作最常见的存储设备，是很多数码设备的存储媒介，比如数码相框、数码相机、MP5 等。ALIENTEK MiniSTM32 开发板自带了 SD 卡接口，可以用于 SD 卡实验，方便大家学习 SD 卡，TF 卡通过转接座也可以很方便的接到我们的开发板上。

有了它，开发板就相当于拥有了一个大容量的外部存储器，不但可以用来提供数据，也可以用来存储数据，使得这款开发板可以完成更多的功能。

这里要特别说明一下：该 SD 卡卡座是在开发板的背面!

24. W25X16 16M FLASH

这是 ALIENTEK MiniSTM32 开发板板载的一颗 FLASH 芯片，型号为 W25X16。这颗芯片的容量为 16M bit，也就是 2M 字节，其容量和 AT45DB161 是一样的，但是价格实惠很多。这颗芯片非常适合我们存储一些不常修改的数据，比如字库等。关于该芯片的使用见 3.17 节。

25. NRF24L01 模块接口

这是 ALIENTEK MiniSTM32 开发板板载的 NRF24L01 模块接口，只要插入模块，我们便可以实现无线通信，从而使得我们板子具备了无线功能，这点 JF24C/D 也可以。但是它们由于价格，速率，功率等的不同，而有不同的应用方向，所以我们这两种接口都有预留，大家可以根据自己的喜好，选择不同的无线模块来使用，但是在开发板上，一次只能接有 1 个模块。

26. 2.4 寸/2.8 寸通用 LCD 接口

这是 ALIENTEK MiniSTM32 开发板板载的 LCD 接口，该接口是一个目前比较通用的 LCD 接口，支持 8 位或者 16 位总线或者 SPI 的液晶屏。

有了它，ALIENTEK MiniSTM32 开发板就可以显示图片及文字了。这在后续的实例部分会有很多使用。

27. GPIOB&C 引出 IO 口

这是 ALIENTEK MiniSTM32 开发板板载的 GPIOB 与 GPIOC 的引出口，该接口用于将 STM32 的 GPIOB 和部分的 GPIOC 引出，方便大家的使用，在开发板上的标号为：P1。这里 GPIOB 全部使用顺序引出的方式，尤其适合外部总线型器件的接入。

28. STM32F103RBT6

这是 ALIENTEK MiniSTM32 开发板的核心芯片，型号为：STM32F103RBT6。该芯片具有 20K SRAM、128K FLASH、3 个普通的 16 位定时器、一个 16 位的高级定时器、2 个 SPI、2 个 IIC、3 个串口、1 个 USB、1 个 CAN、2 个 12 位的 ADC、51 个通用 IO 口。

29. OLED&LCD 共用接口

这是 ALIENTEK MiniSTM32 开发板的特色设计，一个接口，兼容两种模块。在此部分，LCD 的部分 IO 和 OLED 的 IO 共用，具体请参看后面的开发板原理图。这样我们一个接口既可以接 LCD 模块，又可以接 OLED 模块。OLED 模块使用的是 ALIENTEK 的 OLED 模块，分辨率为 128*64，模块大小为 2.6cm*2.7cm。

OLED 模块的使用，我们在 3.9 节会有详细介绍。

30. 红外&温度传感器连接口

ALIENTEK MiniSTM32 开发板虽然自带了红外接收头和 DS18B20 的接口，但是并没有将这两个器件直接挂在 IO 口上，而是通过跳线帽来连接，以防止在不使用这两器件的时候，他们对 IO 口的干扰，当然我们也可以用跳线，把 DS18B20 和红外遥控接收模块接到其他电路上使用。

1.2 主流入门级STM32 开发平台对比

目前市面上常见的几款入门级开发板有：ALIENTEK MiniSTM32 开发板、奋斗板、芯达板、ST 三合一开发板等。下面我们简单对比一下这几款开发板，如表 1.1 所示：

外设 \ 品牌		ALIENTEK	奋斗板	芯达板	ST三合一板
主要外设	CPU	STM32F103RBT6	STM32F103VBT6	STM32F103VCT6	STM32F103C8T6
	FLASH	2M BYTE	2M BYTE	无	无
	EEPROM	2K BIT	无	无	无
	USB	支持	支持	支持	支持
	RTC	支持	支持	支持	支持
	JTAG	支持	支持	支持	支持
	SD 卡	大卡座	小卡座	小卡座	无
	液晶屏	2.8寸TFTLCD	2.4寸TFTLCD	2.4寸TFTLCD	无
	触摸屏	支持	支持	支持	无
	按键	3个	1个	2个	5维导向键
	LED	红绿蓝各1个	1个蓝色1个橙色	4个红色1个蓝色	4个红色
	无线	NRF24L01标准接口+JF24C/D无线接口	无	无	无
	红外	38K接收头+配送遥控器	无	无	无
	串口	1路USB串口	1路RS232串口	2路RS232串口	无
	引出IO	全部IO	剩余IO	11个	全部IO
其他外设	PS/2	支持	无	无	无
	DS18B20	支持	无	无	无
	OLED	支持	无	无	无
	电源输出	3.3V和5V 双电源输入输出,每组3路	无	3.3V和5V各一路	无
	STLINK+STM8小板	无	无	无	有
其他	价格(￥)	218	198	148	199
	特点	全部原创例程,内容丰富,高效率寄存器操作,注释详细,外设齐全,资料翔实.性价比极高	例程大都基于库操作,以ucGUI见长,外设较少,性价比适中	例程大都基于库,例程较少,外设较少,但价格便宜,性价比较高	ST原装开发板,外设几乎没有,但配了STLINK和STM8小板,性价比一般

表 1.1 入门级开发板对比表

由表 1.1 可以看出从外设资源方面来说 ALIENTEK 相比其他几款开发板，要多得多。但是 ALIENTEK MiniSTM32 开发板的 CPU 配置要比奋斗板和芯达板的要低一些，不过 ALIENTEK 开发板的 IO 口全部（时钟所占 IO 口不包括在内）都引出来了这样，总共引出的 IO 数有 41 个，在实验的时候，是绰绰有余了。

从表 1.1 中还可以看出，ALIENTEK 带的是 USB 串口，而其他的都是 RS232 串口或者没有，USB 串口的好处是不需要电脑自带串口，只需要一根 USB 线即可实现串口，从而给 STM32 下载代码。事实上，现在的电脑自带串口的越来越少了，尤其是笔记本，几乎都没有串口。ALIENTEK 开发板设计成 USB 串口，就是考虑到了这点，所以在 ALIENTEK MiniSTM32 开发板上只需要一根 USB 线，就能实现 STM32 开发（供电+下载+串口调试）。

另外，ST 的三合一套件，在 STM32 这方面，其实就只提供了一个 STM32F103C8T6 的核心板，这样在 STM32 这块，它比其他几款开发板都要逊色，只是带了 STLINK 和 STM8 小板，对于想同时学习 STM32 和 STM8 的朋友来说，还是一个不错的选择。

从例程方面来说，ALIENTEK 相比其他几款开发板，最大的区别就是 ALIENTEK 的例程，几乎都不是用库函数的，而其他几款开发板的例程，都是基于库函数的。基于库函数的例程，其实在 MDK 的安装目录下都有，只是那些例程是基于 ST 的另外两款开发板的，但只要稍作修改，大都能在其他开发板上运行。ALIENTEK 的这款开发板的例程，基本都是原创的，不基于库函数的，而且拥有翔实的注释，代码风格统一，循序渐进，非常适合初学者入门。而其他几款开发板的例程，大都是来自 ST 库函数的直接修改，注释也比较少，对初学者来说不那么容易入门。

综合看来，ALIENTEK 开发板虽然在 CPU 上处于劣势，但是其丰富的外设和例程，以及翔实的资料，都是 STM32 初学者不可多得的入门首选，所以在这几款入门级开发平台里面，ALIENTEK MiniSTM32 开发板是最适合初学者的。当然，对于想深入了解 STM32 内部资源的朋友，ALIENTEK MiniSTM32 开发板也绝对是一个不错的选择。

1.2 ALIENTEK MiniSTM32 开发板硬件详解

本节将向大家介绍 ALIENTEK MiniSTM32 开发板的各部分硬件，让大家对该开发板的各部分硬件原理有个理解。

1.2.1 MCU

ALIENTEK MiniSTM32 选择的是 STM32F103RBT6 作为 MCU，STM32F103 的型号众多，我们选择这款的原因是看重其性价比，作为一款低端开发板，选择 STM32F103RBT6 是最佳的选择。128K FLASH、20K SRAM、2 个 SPI、3 个串口、1 个 USB、1 个 CAN、2 个 12 位的 ADC、RTC、51 个可用 IO 脚…，这样的配置无论放到哪里都是很不错的了，更重要的是其价格，18 元左右的零售价，足以秒杀很多其他芯片了，所以我们选择了它作为我们的主芯片。MCU 部分原理图如下：

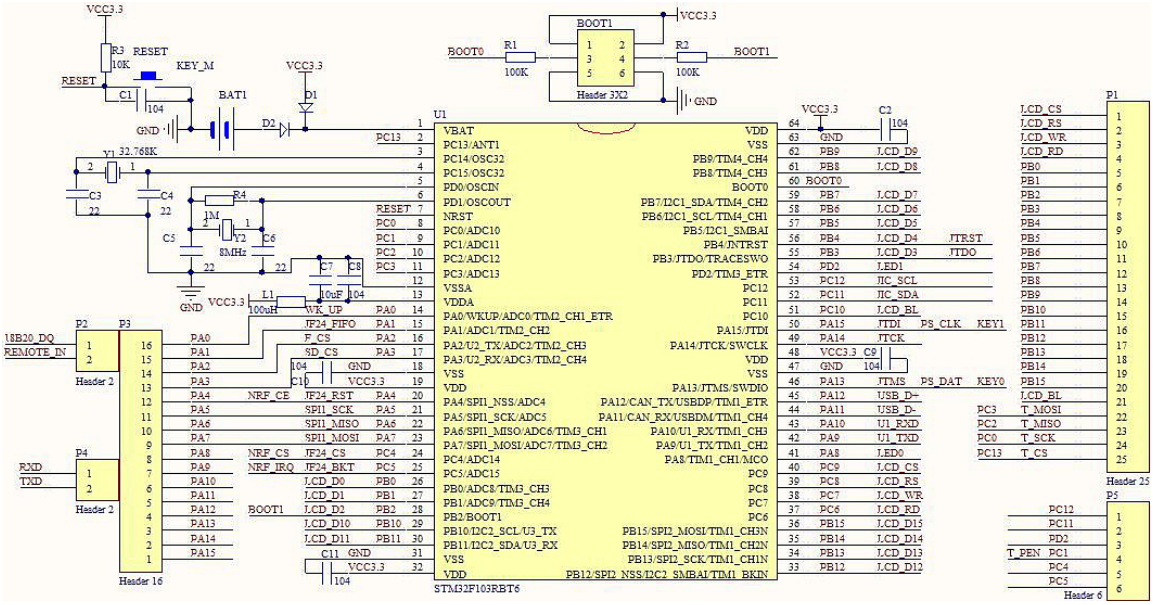


图 1.2.1.1 MCU 部分原理图

上图中中上部的 BOOT1 用于设置 STM32 的启动方式，其对应启动模式如下表所示：

BOOT0	BOOT1	启动模式	说明
0	X	用户闪存存储器	用户闪存存储器，也就是FLASH启动
1	0	系统存储器	系统存储器启动，用于串口下载
1	1	SRAM启动	SRAM启动，用于在SRAM中调试代码

表 1.2.1.1 BOOT0、BOOT1 启动模式表

按照表 1.2.1.1，一般情况下如果我们想用用串口下载代码，则必须配置 BOOT0 为 1，BOOT1 为 0，而如果想让 STM32 一按复位键就开始跑代码，则需要配置 BOOT0 为 0，BOOT1 随便设置都可以。这里 ALIENTEK 这款开发板专门设计了一键下载电路，通过串口的 DTR 和 RTS 信号，来自动配置 BOOT0 和 BOOT1，因此不需要用户来手动切换他们的状态，直接串口下载软件自动控制，可以非常方便的下载代码。

P3 和 P1 分别用于 PORTA 和 PORTB 的 IO 口引出，其中 P2 还有部分用于 PORTC 口的引出。PORTA 和 PORTB 都是按顺序排列的，这样设计的目的是为了让大家更方便地与外部设备连接。

P2 连接了 DS18B20 的数据口以及红外传感器的数据线，它们分别对应着 PA0 和 PA1，只需要通过跳线帽将 P2 和 P3 连接起来就可以使用了。这里不直接连在一起的原因有二：1，防止红外传感器和 DS18B20 对这两个 IO 口作为其他功能使用的时候的影响；2，DS18B20 和红外传感器还可以用来给其他板子提供输入，等于我们的板子为别的板子提供了红外接口和温度传感器，在调试的时候，还是蛮有用的。

P4 口连接了 PL2303 的串口输出，对应着 STM32 的串口 1（PA9/PA10），在使用的时候，也是通过跳线帽将这两处连接起来。这样设计有 2 个好处：1，使得 PA9 和 PA10 用作其他用途使用的时候，不受到 PL2303 的影响。2，USB 转串口可以用作他用，并不仅限这个板上的 STM32 使用，也可以连接到其他板子上，这样 ALIENEK MiniSTM32 就相当于一个 USB 串口。

P5 口是另外一个 IO 引出排阵，将 PORTC 和 PORTD 等的剩余 IO 口从这里引出。

在此部分原理图中，我们还可以看到 STM32F103RBT6 的各个 IO 口与外设的连接关系，这些将在后面给大家介绍。

这里 STM32 的 VBAT 采用 CR1220 纽扣电池和 VCC3.3 混合供电的方式，在有外部电源（VCC3.3）的时候，CR1220 不给 VBAT 供电，而在外部电源断开的时候，则由 CR1220 给 VBAT 供电。这样，VBAT 总是有电的，以保证 RTC 的走时以及后备寄存器的内容不丢失。

该部分还有 JTAG，JTAG 部分电路如下图：

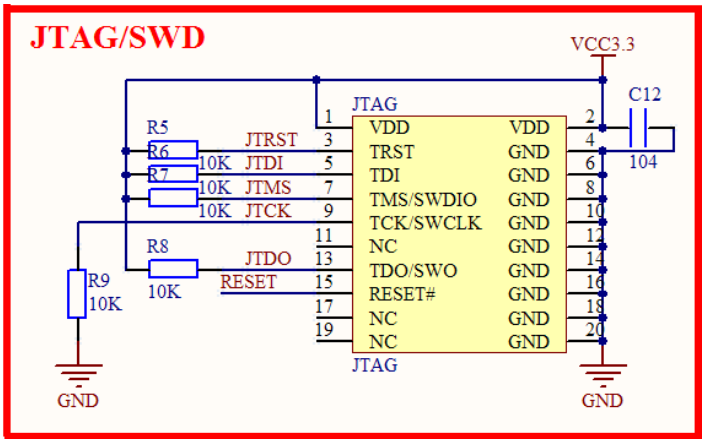


图 1.2.1.2　JTAG 原理图

这里采用的是标准的 JTAG 接法，但是 STM32 还有 SWD 接口，SWD 只需要最少 2 跟线（SWCLK 和 SWDIO）就可以下载并调试代码了，这同我们使用串口下载代码差不多，而且速度更快，能调试。所以建议大家在设计产品的时候，可以留出 SWD 来下载调试代码，而摒弃 JTAG。STM32 的 SWD 接口与 JTAG 是共用的，只要接上 JTAG，你就可以使用 SWD 模式了（其实并不需要 JTAG 这么多线），JLINKV8/JLINKV7 和 ULINK2 都支持 SWD。

1.2.2 EEPROM

ALIENTEK MiniSTM32 自带了 24C02 的 EEPROM 芯片，该芯片的容量为 2Kbit，也就是 256 个字节，对于我们普通应用来说是足够了的。你也可以选择换大的芯片，因为在原理上是兼容 24C02~24C512 全系列的 EEPROM 芯片的。其原理图如下：

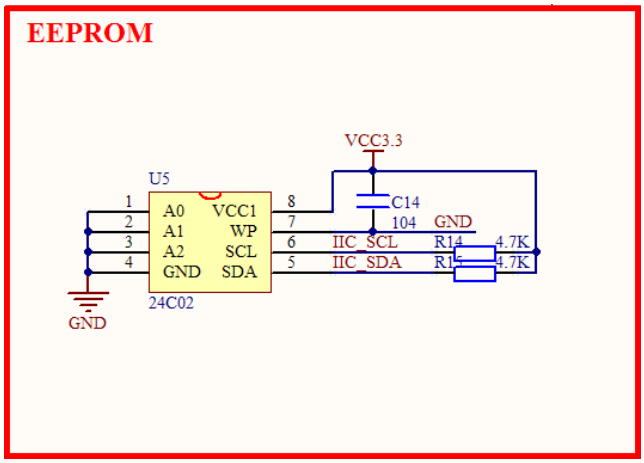


图 1.2.2.1　EEPROM 原理图

这里我们把 A0~A2 均接地，对 24C02 来说也就是把地址位设置成了 0 了，写程序的时候要注意这点。IIC_SCL 接在 MCU 的 PC12 上，IIC_SDA 接在 MCU 的 PC11 上，这里我们并没有接到 STM32 内部的 IIC 上，因为 STM32 的 IIC 是鸡肋！如果你想在 ALIENTEK MiniSTM32 开发板上使用硬件 IIC，那么也是可以的，你只需要设置 PC11 和 PC12 为浮空输入，然后把 PB10 和 PB11（IIC2）或者 PB6 和 PB7(IIC1)通过飞线连接到 PC11 和 PC12 上就可以使用硬件 IIC 了。

1.2.3 温度传感器

温度传感器我们使用的是 DS18B20，其原理图如下：

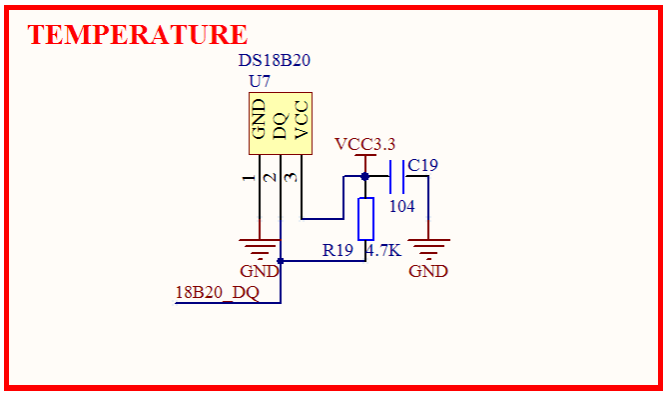


图 1.2.3.1　温度传感器原理图

DS18B20 的数据脚（18B20_DQ）接 P2 的第一脚，并没有直接连接到 MCU，至于为什么，前面已有介绍。要使用这里，我们用跳线帽把 PA0 和 P2-1 连接起来就可以了。

1.2.4 按键

ALIENTEK MiniSTM32 开发板总共有 3 个按键，其原理图如下：

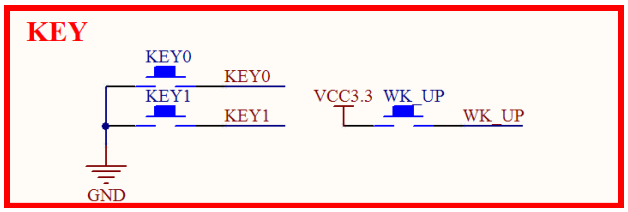


图 1.2.4.1　按键输入原理图

KEY0 和 KEY1 用作普通按键输入，分别连接在 PA13 和 PA15 上，他们都连接在了 JTAG 相关的引脚上（KEY0 还连接在 SWDIO 上），这样，在使用 KEY0 和 KEY1 的时候，就不能使用 JTAG 来调试了，这点在使用的时候要注意。KEY0 和 KEY1 还和 PS/2 的 DAT 和 CLK 线共用，他们都通过 JTAG 的上拉电阻来提供上拉。

WK_UP 按键连接到 PA0(STM32 的 WKUP 引脚)，它除了可以用作普通输入按键外，还可以用作 STM32 的唤醒输入。这个按键是高电平触发的。PA0 还是 DS18B20 的输入引脚，所以在使用的时候要注意哦。

1.2.5 液晶显示模块

ALIENTEK MiniSTM32 开发板载有目前比较通用的液晶显示模块接口，还有其比较有特色的兼容性接口，不仅支持 2.4、2.8 寸的 TFTLCD,还支持 OLED 显示器。其原理图如下：

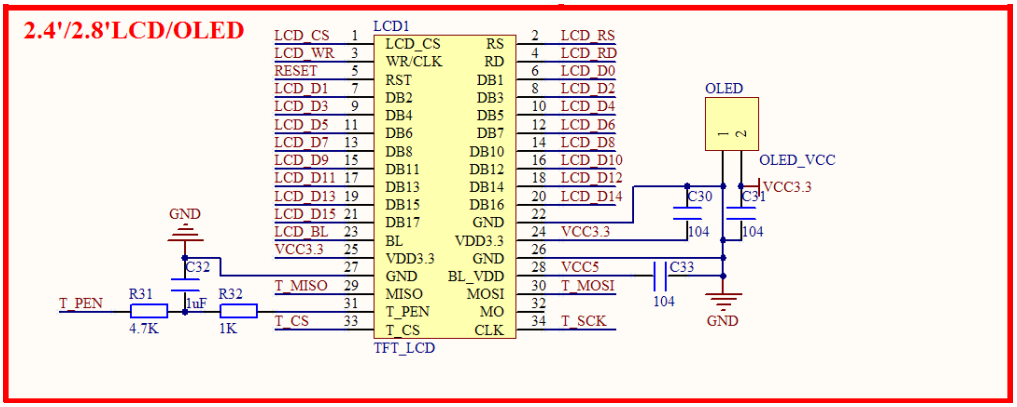


图 1.2.5.1　液晶显示模块原理图

LCD1 是一个通用的液晶模块接口。OLED 是一个给 OLED 显示模块供电的接口，它和 LCD1 拼接在一起。当使用 2.4’/2.8’ 的 LCD 时，我们接到 LCD1 上就可以了，而当我们使用 ALIENTEK 的 OLED 模块时，则接 OLED 排阵做电源，同时会连接到 LCD1 上的部分管脚，从而实现 OLED 与 MCU 的连接。ALIENTEK MiniSTM32 的 LCD 接口兼容：ALIENTEK 的 TFTLCD 模块、红牛开发板的液晶模块、CRE 开发板的液晶模块、STMSKY 开发板的液晶模块等。所以，如果有以上几款开发板的液晶模块，或者接口与上面原理图兼容的，都是可以在 ALIENTEK MiniSTM32 开发板上使用的。

这些引脚与 MCU 的连接关系我们在这里就不一一列出了，大家可以从 MCU 的原理图上找到。T_PEN 是触摸屏的 PEN 信号输出，我们在这里加了滤波电路，使得触摸屏读数更加准确。

1.2.6 红外接收头

ALIENTEK MiniSTM32 开发板载有红外接收传感器 HS0038，原理图如下：

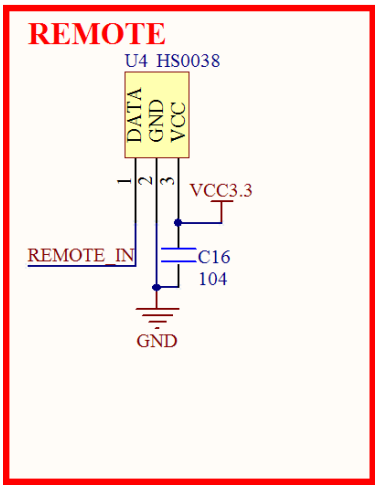


图 1.2.6.1　液晶显示模块原理图

REMOTE_IN 接到 P2 的第二脚，也没有直接接在 MCU 的 IO 口上，目的也是防止 IO 口在做其他功能使用的时候，收到红外信号的干扰。

1.2.7 PS/2

ALIENTEK MiniSTM32 开发板载有 PS/2 接口，有了该接口，我们就可以用来连接外部标准的 PS/2 鼠标键盘了，也就大大的扩展了 ALIENTEK MiniSTM32 的输入。原理图如下：

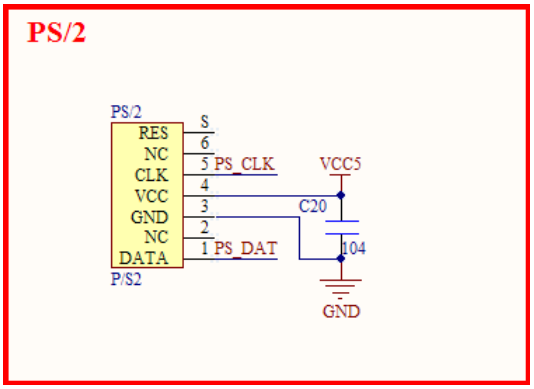


图 1.2.7.1　PS/2 接口原理图

PS_CLK 和 PS_DAT 分别接 PA15 和 PA13，PS/2 的信号线是需要外部提供上拉电阻的，这里我们和 JTAG 共用，使用 JTAG 的上拉电阻来提供，PS/2 的 CLK 和 DAT 还与两个按键共用。所以在使用这几部分的时候，要特别注意，别冲突，可以分时复用。在使用 PS/2 的时候，同样不能使用 JTAG 调试。

1.2.8 LED

ALIENTEK MiniSTM32 开发板上总共有 3 个 LED，其原理图如下：

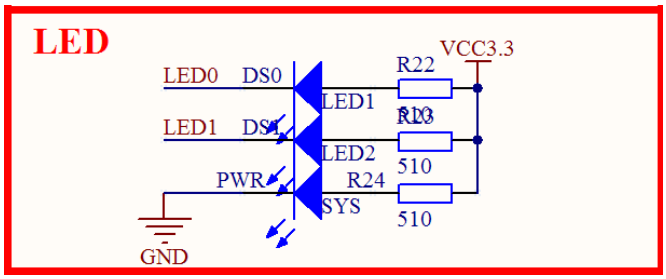


图 1.2.8.1　LED 原理图

其中 PWR 是系统电源指示灯，为蓝色。LED0 和 LED1 分别接在 PA8 和 PD2 上，PA8 还可以通过 TIM1 的通道 1 的 PWM 输出来控制 DS0 的亮度。为了方便大家判断，我们选择了 DS0 为红色，DS1 为绿色的 LED 灯。

1.2.9 SD 卡

ALIENTEK MiniSTM32 开发板载有标准的 SD 卡接口，有了这个接口，我们就可以外扩大容量存储设备，可以用来记录数据。其原理图如下：

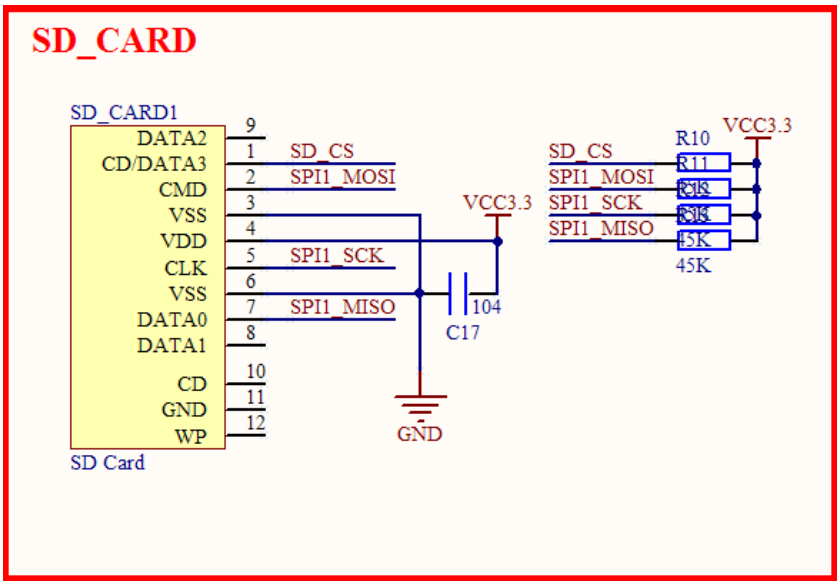


图 1.2.9.1　SD 卡接口原理图

SD 卡我们使用的是 SPI 模式通信，SD 卡的 SPI 接口连接到 STM32 的 SPI1 上，SD_CS 接在 PA3 上，ALIENTEK MiniSTM32 开发板上的 SPI1 总共由 4 个外设共用，他们分别是：SD 卡、NRF24L01 无线模块、JF24C 无线模块和 W25X16。他们可以通过不同的片选信号来分时复用。

1.2.10 无线模块

ALIENTEK MiniSTM32 开发板板载了 2 款无线模块的接口，NRF24L01 模块和 JF24C/D 模块，他们都属于 2.4G 通信的无线模块，并且都有性价比极高的特点。其中 NRF24L01 模块的最大通信速率为 2Mbps，JF24C/D 的为 1Mbps。有了这个两个接口，我们就可以做无线通信，以及其他很多的应用了。这部分原理图如下：

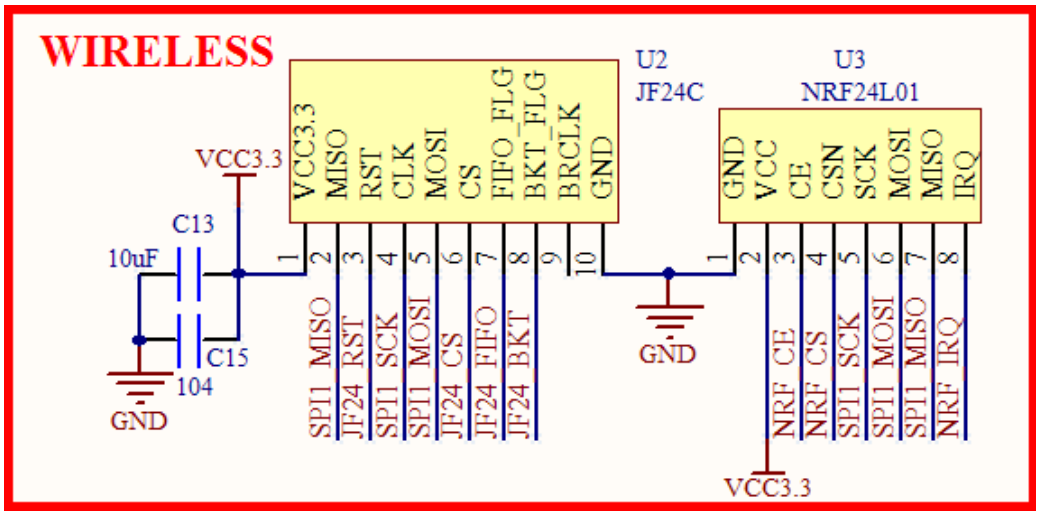


图 1.2.10.1　无线模块接口原理图

这两部分也是共用 SPI 接口，其余引脚与 MCU 的对应关系在 MCU 部分有，我们这里就不列出了。注意这两个无线模块是不能同时连接在板子上的！

1.2.11 SPI FLASH

ALIENTEK MiniSTM32 开发板载有 SPI FLASH 芯片 W25X16，该芯片的容量为 2M 字节，与 AT45DB161 属于同一级别，ATMEL 的东西价格近来很不稳定，因而我们选择了价格稳定，货源较好，而且通用性很强的 W25X16，其原理图如下：

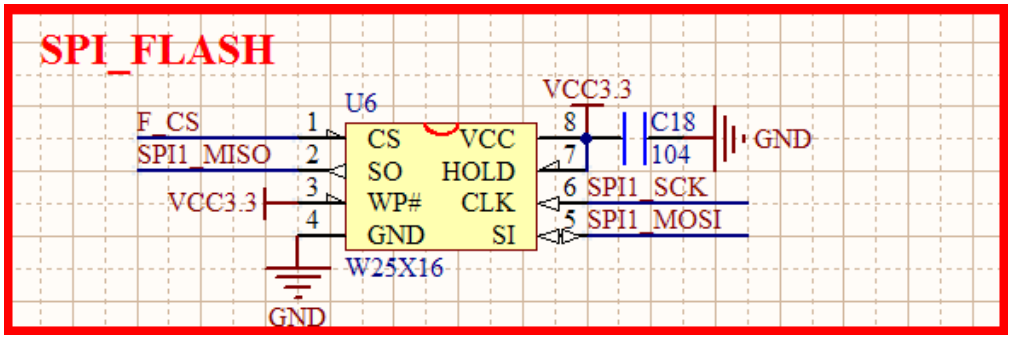


图 1.2.11.1　W25X16 原理图

W25X16 也是共用了 SPI1，F_CS 接在 PA2 上。至此，总共 SPI1 的四个器件都已介绍完毕，他们的 CS 都接在不同的 IO 口上（两个无线模块除外），所以在使用其中一个器件的时候，要记得禁止其他器件的 CS 脚，否则会有干扰。

1.2.12 USB 串口、USB、电源

这里三个部分一起介绍，ALIENTEK MiniSTM32 开发板板载了 USB 串口，并且由 USB 提供电源，使得我们只需要一根 USB 线就可以使用 ALIENTEK MiniSTM32 开发板了，包括下载、供电、调试 3 位一体。

ALIENTEK MiniSTM32 开发板的供电部分还引出了 5V 和 3.3V 的排阵，可以用来为外部设备提供电源或者从外部引入电源，这在很多时候是非常有用的，有时候你突然要一个 3.3V 的电源，但找半天就是没这样的电源，而我们的板子则可直接向外部提供 3.3V 电源，有了它，你就可以给外部设备提供 3.3V、5V 电源了。注意电流不能太大哦！

ALIENTEK MiniSTM32 开发板的 USB 接口通过独立的 Mini USB 头引出，不和 USB_232 共用，这样不但可以同时使用，还可以给系统提供更大的电流。

这几个部分的原理图如下：

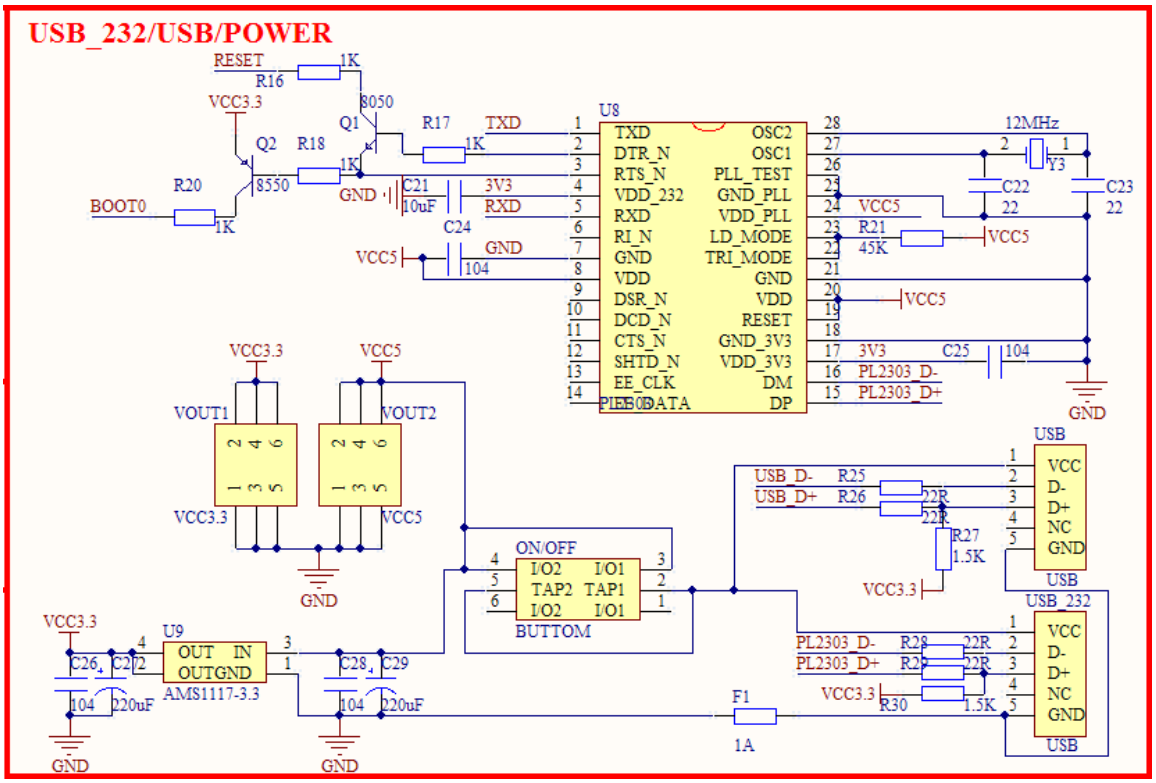


图 1.2.12.1　USB 串口、USB、电源部分原理图

图中的 Q1 和 Q2 外加几个电阻就构成了开发板的一键下载电路，此电路通过 RST 和 DTR 信号来控制 BOOT0 和 RESET 信号，从而实现一键下载的功能。

此部分还有一个开关 BUTTON（对应开发板的标号为 ON/OFF），用来控制整个系统的供电，如果断开则整个系统的 3.3V 部分都将断电。而 5V 部分的电源还是开启的。图中 F1 为可恢复保险丝，用于保护 USB。

1.3 ALIENTEK MiniSTM32 开发板使用注意事项

为了让大家更好的使用 ALIENTEK MiniSTM32 开发板，我们在这里总结该开发板使用的时候尤其要注意的一些问题，希望大家在使用的时候多多注意，以减少不必要的问题。

1，开发板供电由 USB 供电，当然也可以外部供电（接在 5V 的排针上），向外部供电的时候，最大可提供电流为 500mA。在给外部供电的时候不能提供给需要大于此电流的设备。虽然板载了可恢复保险丝，但还是请读者不要乱搞他（将电源短路），毕竟常在河边走，哪有不湿鞋。要是电脑被搞挂了，那就不太好了。

2，SPI1 被 4 个 SPI 器件共用，在使用的时候，一定要在软件里面注意同一时刻只有一个处于激活状态（CS 有效），以防止他们相互干扰。他们的 CS 脚都是直接接在 PA 口上的，所以 PA 口不适合用来做 PA 口的这几个部分不适宜接总线型设备，而比较适合接串行设备。

3，WK_UP 按键和 DS18B20 共用，所以这两个功能不能同时使用，否则有可能干扰，这也需要我们在使用的时候注意。不过您不用担心，他们是可以分时复用的。

4，JTAG 部分和很多的外设共用了 IO 口，LCD 部分和 JTAG 共用了 PB3 和 PB4，所以在使用的时候，我们不建议使用 JTAG 模式，而建议大家使用占用引脚数极少的 SWD 模式仿真调试。但是 SWD 和 KEY0 以及 PS_DAT 共用了 SWDIO，所以 KEY0 和 PS/2 是无论如何都不能通过 JTAG/SWD 调试的，而只能通过串口来调试（打印信息）。

5，当使用液晶模块（16 位模式）的时候，PB0~PB16 都被占用了，可以分时复用，但是在写程序的时候要注意，这里还有连接到触摸屏的 PC2/PC3/PC0/PC13 均会存在这样的问题，在使用的时候要格外注意，看是否会产生干扰。

6，由于开发板加入了串口自动下载电路，在 PL2303 与 USB 握手的时候，可能导致 STM32 被复位，此状态是不稳定的，所以在开发板刚刚与电脑连接的时候，一般可以看到 STM32 被多次复位了。另外，电脑其他 USB 的加载与卸载，也有可能导致 STM32 被异常复位。如果觉得这个很影响使用，你只需要把 R16 焊掉即可。

至此，ALIENTEK MiniSTM32 开发板的硬件部分就介绍完了，了解了整个硬件对我们后面的学习会有很大帮助，有助于理解后面的代码，在编写软件的时候，可以事半功倍，希望大家细读！另外ALIENTEK开发板的其他资料及教程更新，都可以在技术论坛www.openedv.com下载到，大家可以经常去这个论坛获取更新的信息。

第二章 软件篇

本章将详细介绍 STM32 的开发软件 RVMDK，通过该章，你将了解到：1、在 RVMDK 下新建工程；2、工程的编译；3、软件仿真；4、程序下载；5、在线调试； 6、RVMDK 的一些使用技巧；以上几个环节构成了一个完整的 STM32 开发流程。本章将图文并茂的为大家介绍以上几个方面，使你从一个 KEIL 生手变成熟手。

本章将分为如下几个部分：

1，摘要；

2，RVMDK3.80A 简介；

3，新建 RVMDK 工程；

4，软件仿真；

5，程序下载；

6，在线调试；

7，SYSTEM 文件介绍；

8，RVMDK 使用技巧；

2.1 摘要

本章将结合一个 STM32 的 KEIL 实例，图文并茂的给大家介绍 RVMDK 软件的使用。并简单介绍 RVMDK 的一些使用技巧，希望通过这章的内容，能让一个生手变成熟手。至少能自己利用 RVMDK 编写 STM32 的代码，并在 STM32 上跑起来。

2.2 RVMDK3.80A简介

RVMDK 源自德国的 KEIL 公司，是 RealView MDK 的简称。在全球 RVMDK 被超过 10 万的嵌入式开发工程师使用，RealView MDK 集成了业内最领先的技术，包括 μ Vision3 集成开发环境与 RealView 编译器。支持 ARM7、ARM9 和最新的 Cortex-M3 核处理器，自动配置启动代码，集成 Flash 烧写模块，强大的 Simulation 设备模拟，性能分析等功能。与 ARM 之前的工具包 ADS1.2 相比，RealView 编译器具有代更小、性能更高的优点，RealView 编译器与 ADS.2 的比较：

代码密度：比 ADS1.2 编译的代码尺寸小 10%；

代码性能：比 ADS1.2 编译的代码性能提高 20％；

现在 RVMDK 的最新版本是 RVMDK4.13a，4.0 以上的版本的 RVMDK 对 IDE 界面进行了很大改变，并且支持 Cortex-M0 内核的处理器。作者曾用过 RVMDK3.24/3.80A/4.10 等几个版本，并对他们进行了一些简单的对比，从对比情况来看：3.24 和 3.80a 在各方面的比较都差不多，当然有人说稳定性 3.80A 要好一些，这个我不做评论。但是 4.10 确实界面是好了，支持的器件也多了，但编译效率没有 3.24/3.80A 高，尤其在编译后的代码执行速度（FFT 运算），4.10 要对速度进行-O2 优化才能和 3.24/3.80A 的普通级别相比。另外，国内大都数单片机工程师都接触和使用过 KEIL，相信大家都知道 KEIL 的使用是非常简单的，而且很容易上手。RVMDK3.80A 的编译器界面和 KEIL 十分相似，对于使用过 KEIL 的朋友来说，更容易上手。基于以上几点，本手册将选择 RVMDK3.80A 版本的编译器作为学习 STM32 的软件。当然大家也可以根据自己的喜好换用 4.10 或以上版本的软件。不过请注意：MDK4.0 以上版本编译实验 28 的时候，触摸屏会失效！

2.3 新建RVMDK工程

首先，打开 MDK(以下简称 RVMDK 为 MDK)软件.再点击 Project->New uVision Project 如下图所示:

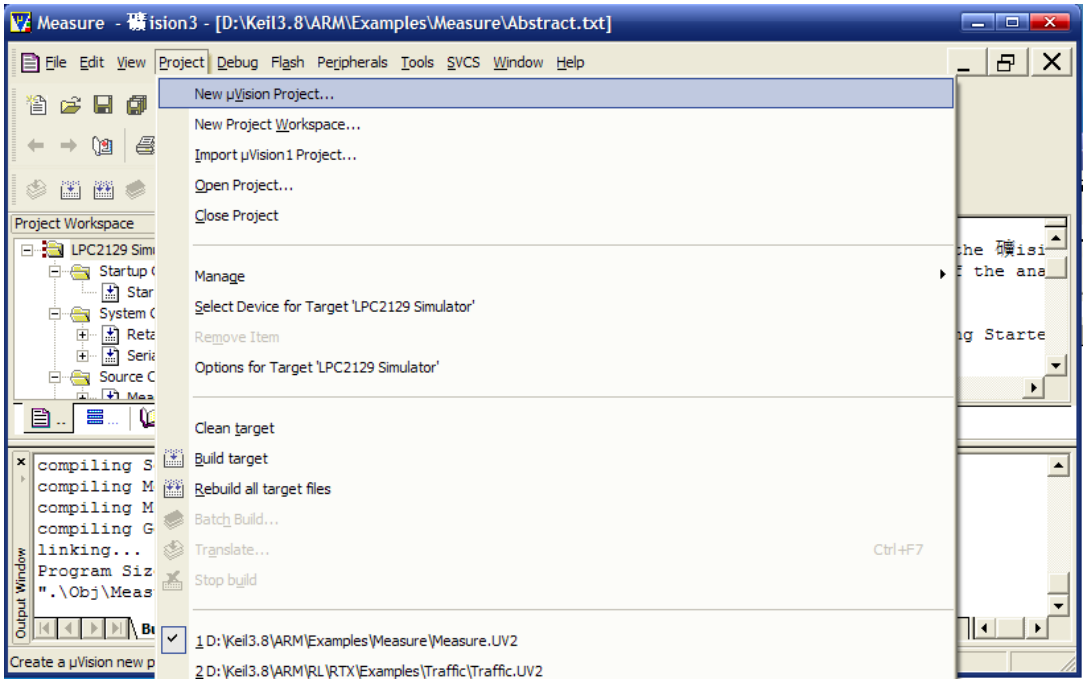


图 2.3.1 新建 MDK 工程

弹出如下对话框:

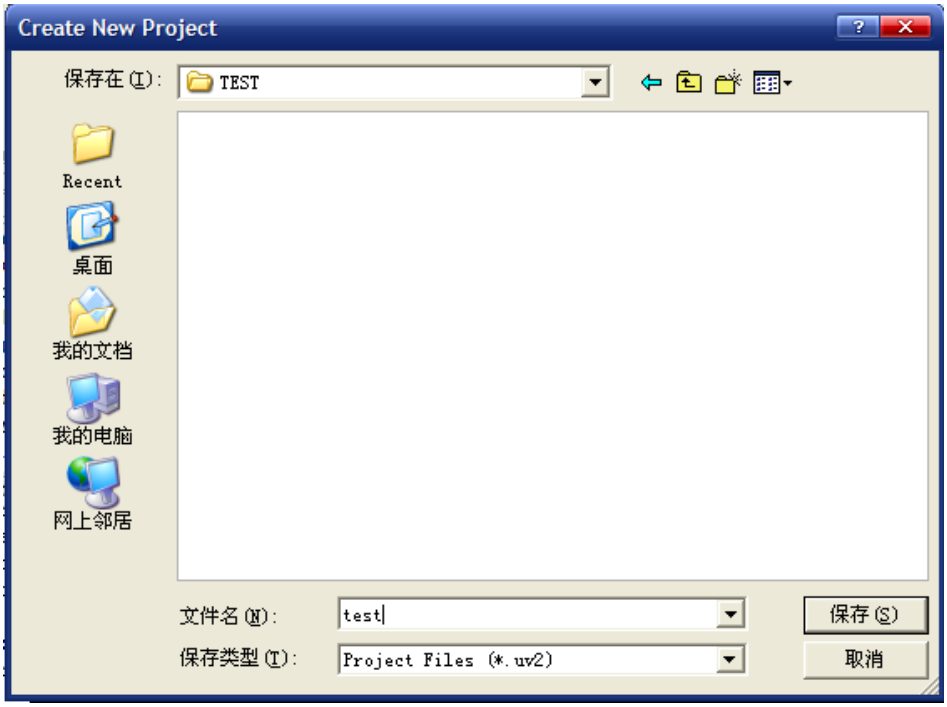


图 2.3.2 保存工程界面

新建一个文件夹 TEST，然后把工程名字设为 test.点击保存. 弹出选择器件的对话框，因为我们的开发板使用的是 STM32F103RBT6，所以在这里我们选择 STMicroelectronics 下面的 STM32F103RB(如果使用的是其他系列的芯片，选择相应的型号就可以了).如下图所示:

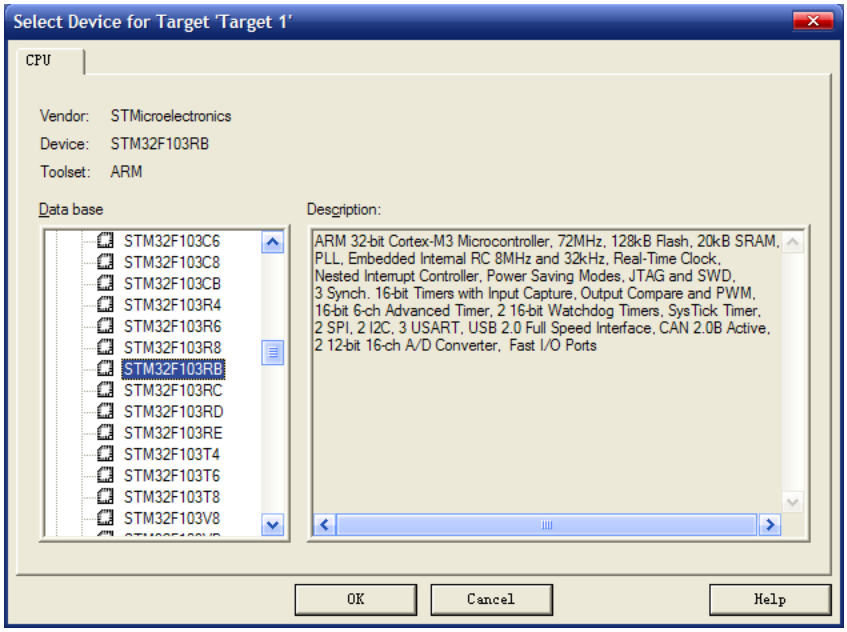


图 2.3.3 器件选择界面

点击 OK，MDK 会弹出一个对话框，问你是否加载启动代码到当前工程下面，这里我们选择是.如下图所示:

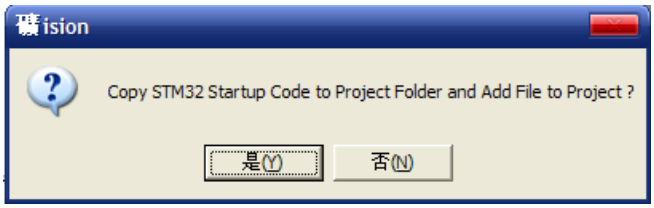


图 2.3.4 提示界面

启动代码是一段和硬件相关的汇编代码.是必不可少的!这代码具体如何工作的，这个我们不必太关心，感兴趣的朋友可以去研究下.在上面点击了是以后，MDK 就把启动代码 STM32F10x.s 加入到了我们的工程下面.如下图所示:

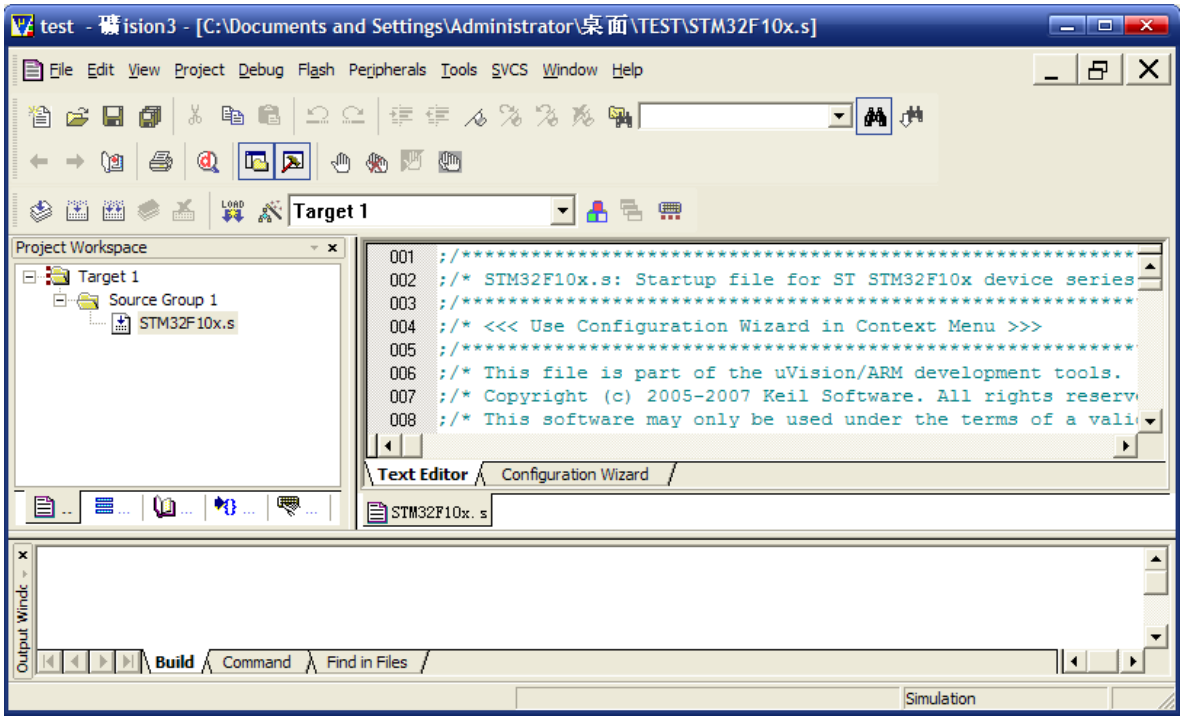


图 2.3.5 工程初步建立

到这里，我们就可以开始编写自己的代码了.在此之前，我们先做 2 件事:第一件，先编译一下，看看什么情况?编译后如下图所示:

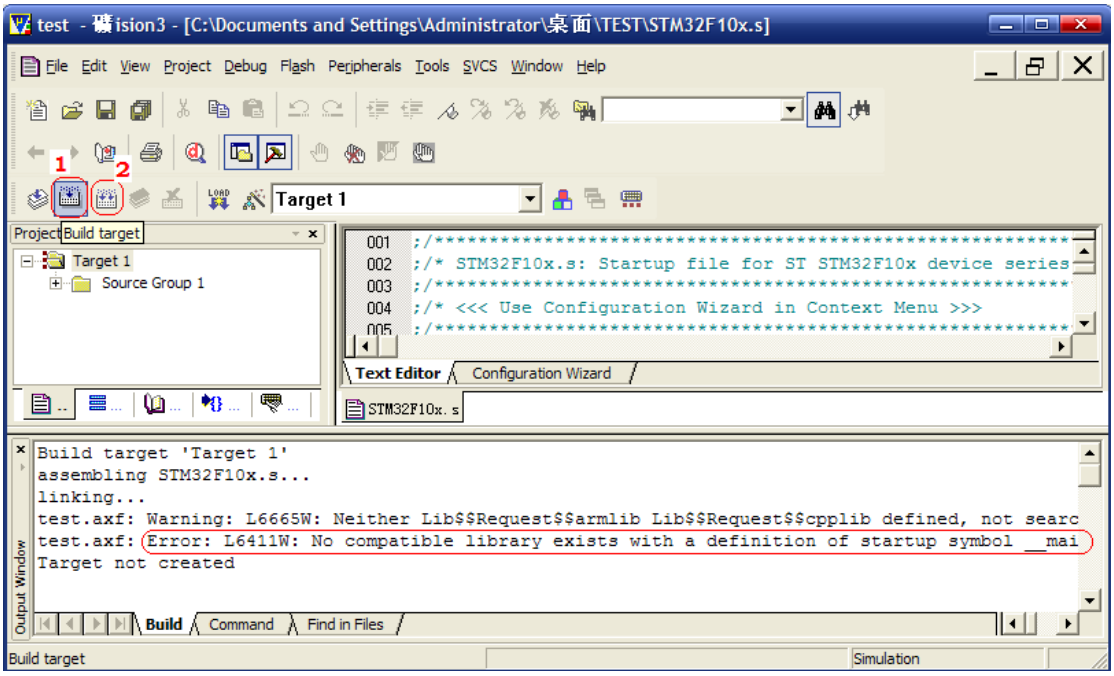


图 2.3.6 编译结果

上图中 1 处为编译当前目标按钮，2 为全部重新编译按钮(工程大时耗时较久，建议少用).出错和警告信息在下面的 Output Windows 对话框中提示出来了.因为工程中没有 main 函数，不报错才怪.

第二件，来看我们存放工程的文件夹有什么变化，如下图所示:

图 2.3.7 编译后工程文件夹的变化

看到文件夹下面有 10 个文件，STM32F10x.s 的启动代码也在该目录下，因为我们之前选择了自动加载，故该文件是由 MDK 自己加进去的.其他还有一些过程文件，总之这样看起来很乱.所以，我们在 TEST 目录下新建一个新文件夹 USER，然后把这些东西全部移到该文件夹下(当然要先关闭 MDK 软件).

由于上面我们还没有任何代码在工程里面，这里我们把系统代码 COPY 过来(整个 SYSTEM 文件夹，这些代码在任何 STM32F103 的芯片上都是通用的，可以用于快速构建自己的工程，后面会有详细介绍).完了之后，TEST 文件夹下的文件如下图所示:

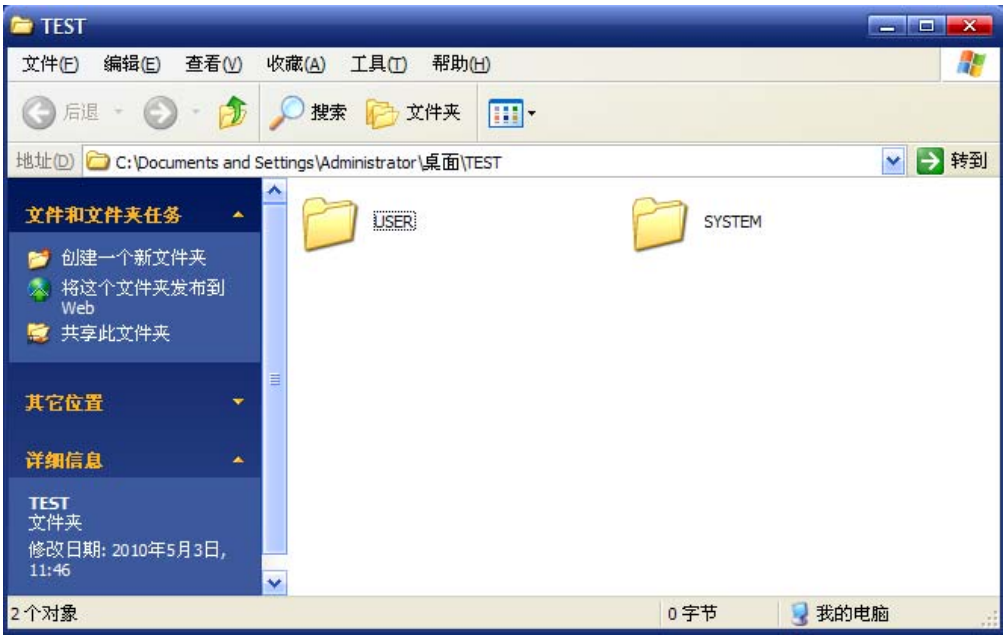


图 2.3.8 工程文件夹最终模样

然后我们在 USER 文件夹下面找到 test.Uv2，打开它.然后在 Target 目录树上右键->Manage Components.如下图所示:

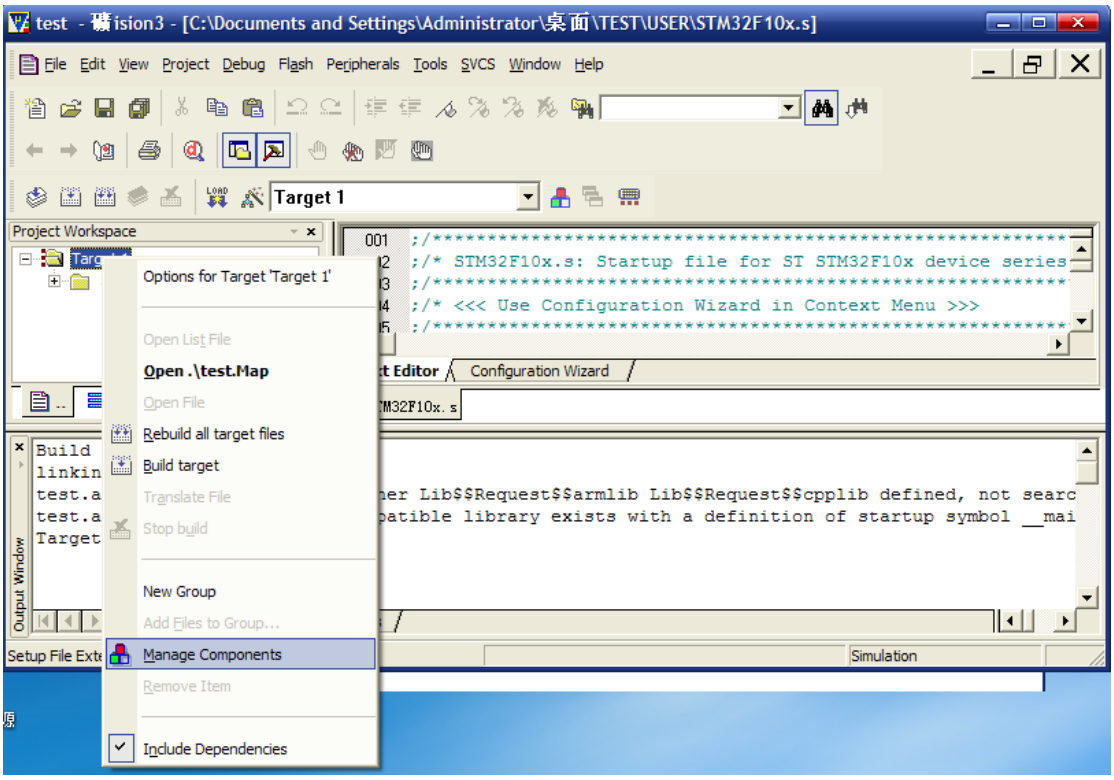


图 2.3.9 调出 Manage Components

在进入 Manage Components 界面之后，弹出如下对话框:

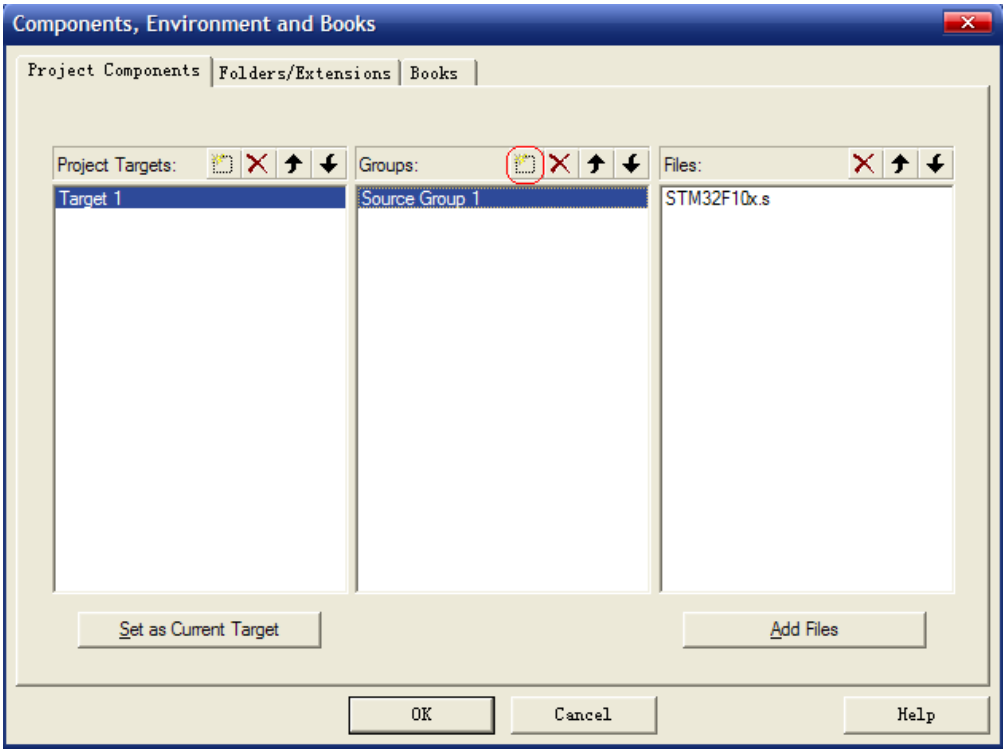


图 2.3.10 Components 选项卡

在上面对话框的中间栏，点新建(用红圈标出)按钮(也可以通过双击下面的空白处实现)，新建 USER 和 SYSTEM 两个组.然后点击 Add Files 按钮，把 sys.c，usart.c，delay.c 加入到 SYSTEM 组中.此时 USER 组下还是没有任何文件的.得到如下图所示的界面:

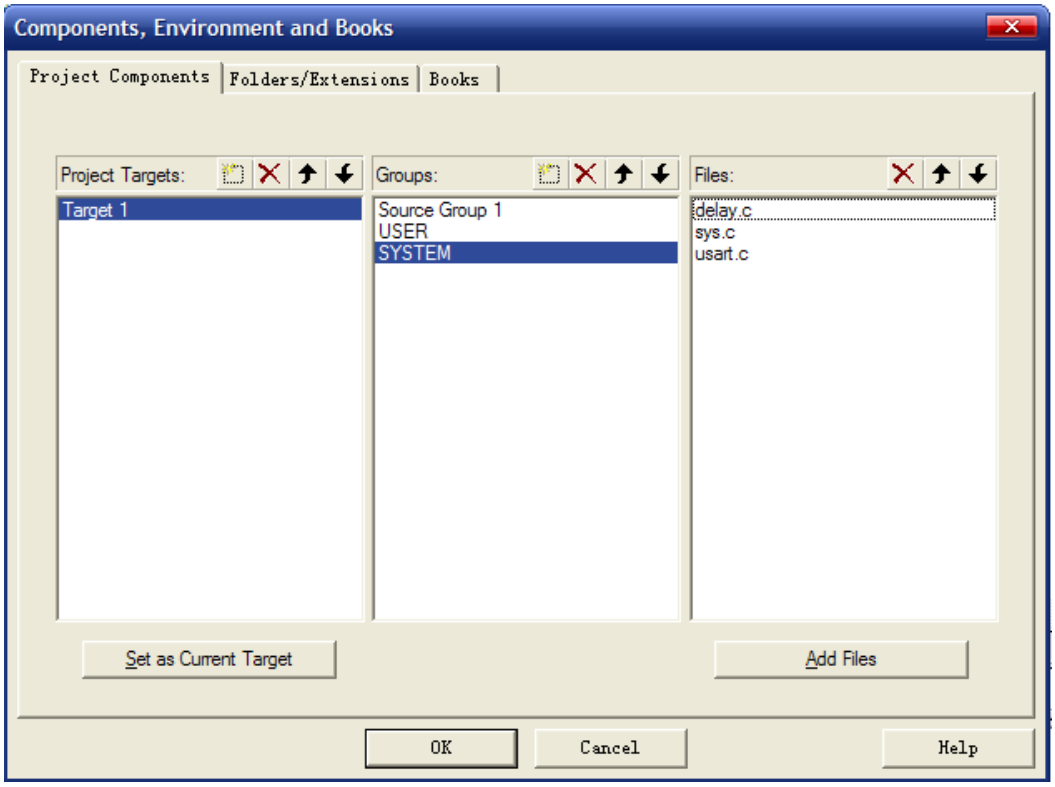


图 2.3.11 修改结果

点击 OK，退出该界面返回 IDE.这时，我们在 Target 树下发现多了 2 个组名，就是我们刚刚新建的 2 个组.如下图所示:

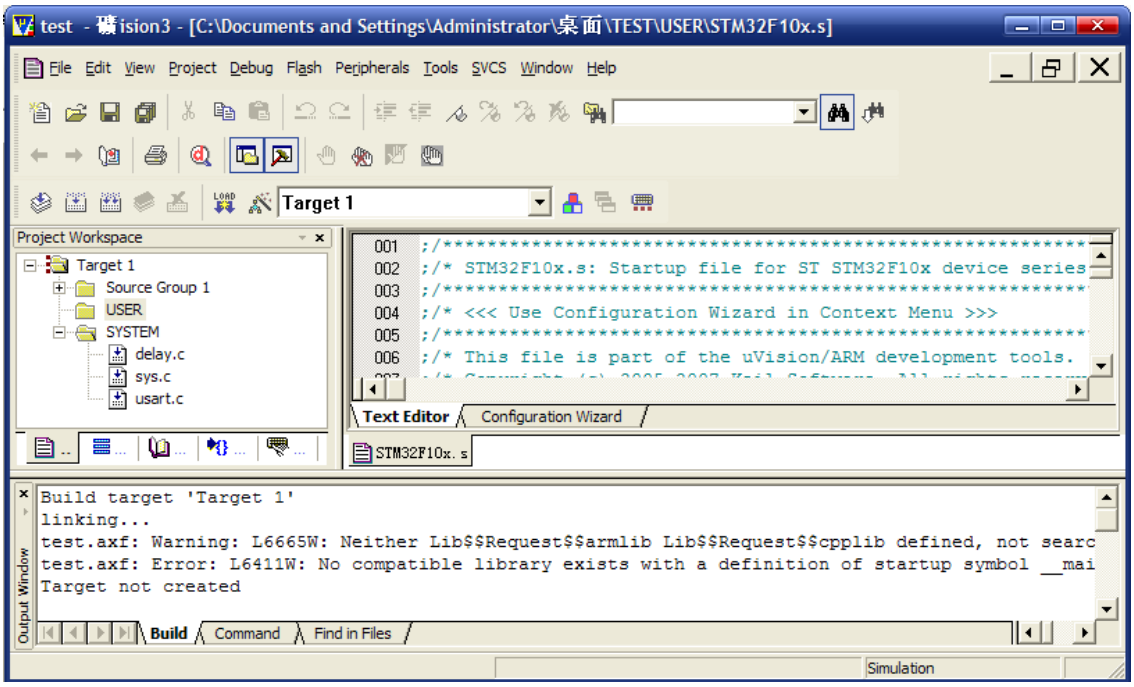


图 2.3.12 在编辑状态下的体现

接着，我们新建一个 test.c 文件，并保存在 USER 目录下.然后双击 USER 组，会弹出加载文件的对话框，此时我们在 USER 目录下选择 test.c 文件，加入到 USER 组下.得到如下界面:

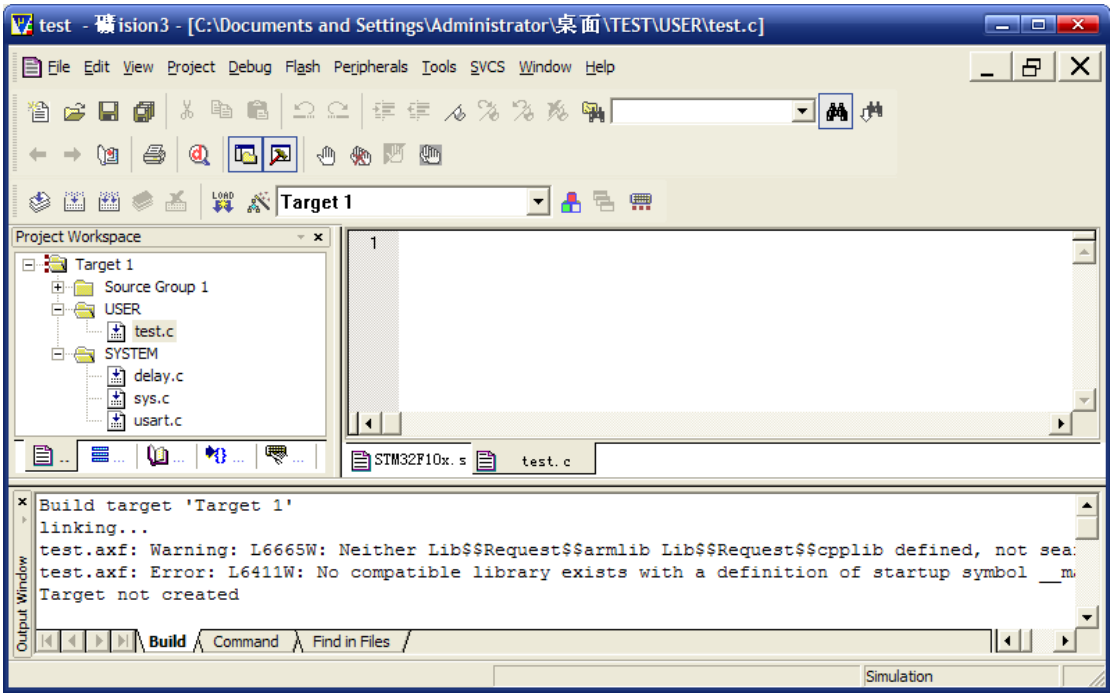


图 2.3.13 在 USER 组下加入 test.c 文件

接着我们在 test.c 文件里面输入如下代码：

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
int main(void)
{
    u8 t=0;
    Stm32_Clock_Init(9);   //72M
    delay_init(72);            //延时初始化
    uart_init(72，9600);       //设置串口 1 波特率
    while(1)
    {
        printf("t:%d\n"，t);
        delay_ms(500);
        t++;
    }
}
```

点击  （部分编译按钮）编译一下，会在 Output Windows 信息栏中发现如下报错信息：test.c(1): error: #5: cannot open source input file "sys.h": No such file or directory。如下图所示（图中红圈内信息）：

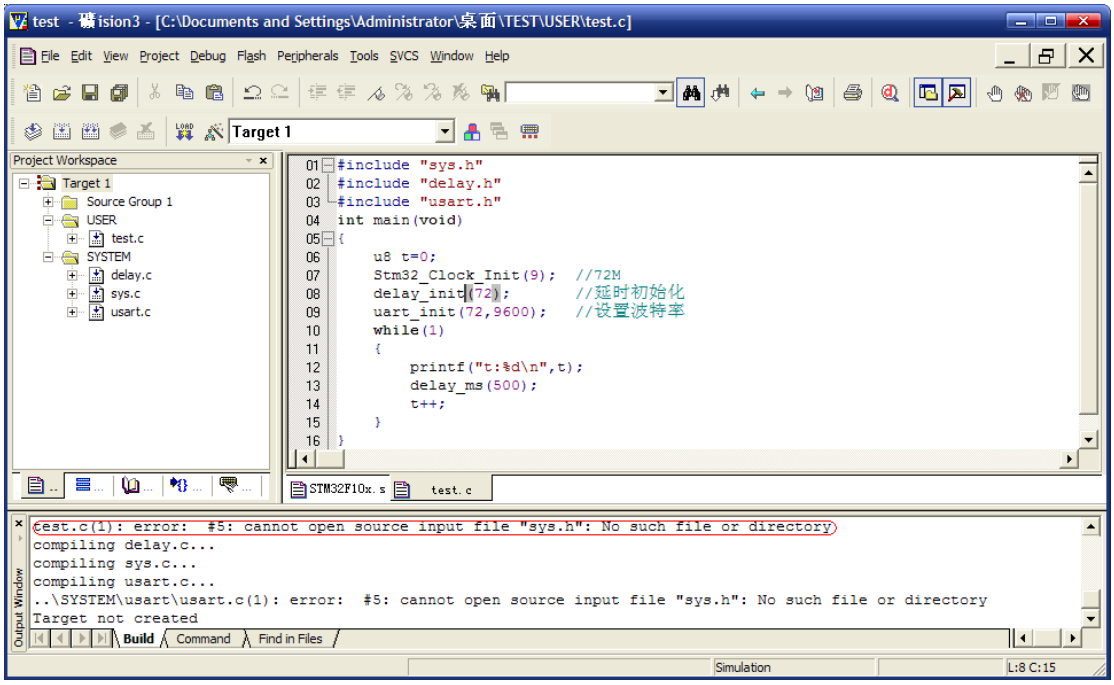


图 2.3.14 编译报错

意思是无法打开源输入文件 sys.h，没这个文件或目录。双击红圈内的内容，你会发现在 test.c 的 01 行出现了一个浅绿色的小箭头，说明错误是这个地方产生的。（这个功能很实用的哦！熟悉 C++的人就知道 C++也有这个功能，快速定位错误、警告产生的地方）。双击后浅绿色箭头出现位置如下：

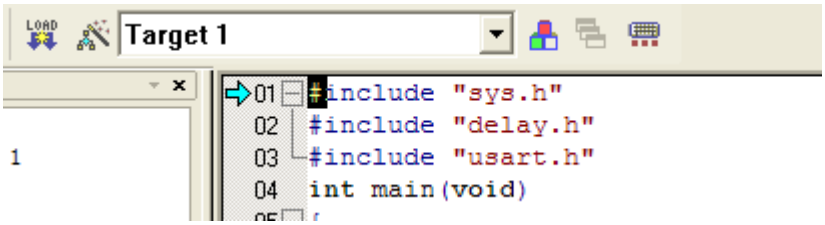


图 2.3.15 编译报错

其实通过下面错误的说明，就是 include 的路径没有加进去的缘故，而导致了这个错误。现在我们点击  （Options for Target 按钮），弹出 Options for Target’Target 1’对话框，选择 C/C++选项卡，如下图所示:

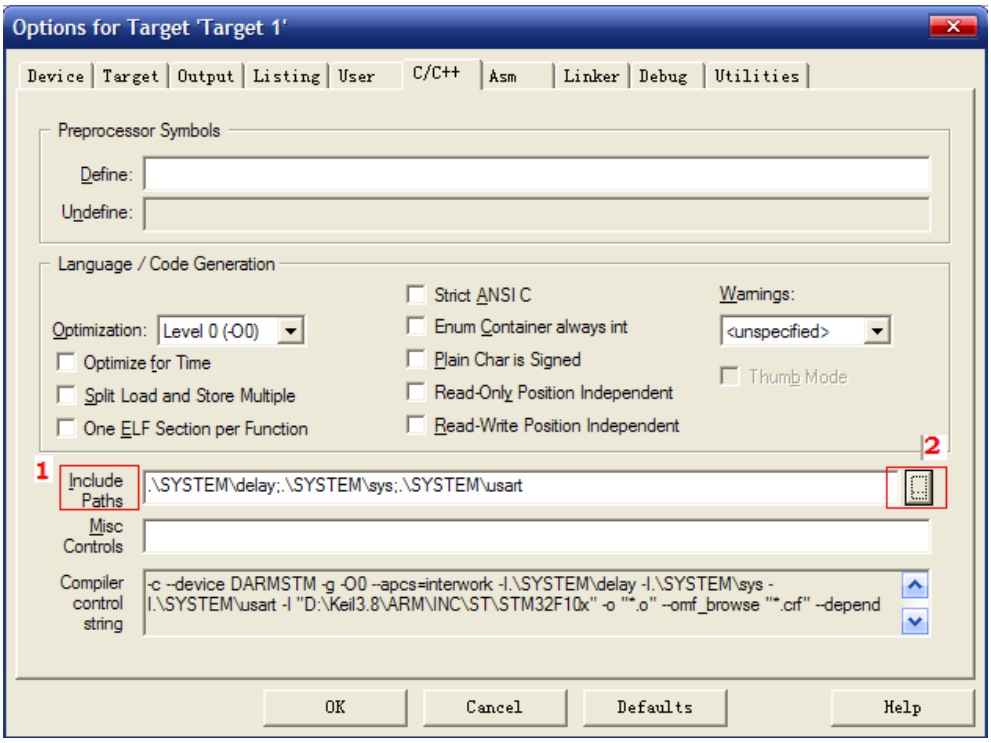


图 2.3.16 加入头文件包含路径

在 Include Paths 处，点击 2 出的按钮。在弹出的对话框中加入 SYSTEM 文件夹下的 3 个文件夹名字，把这几个路径都加进去（此操作即设定编译器的头文件包含路径，面会经常用到）。如下图所示:

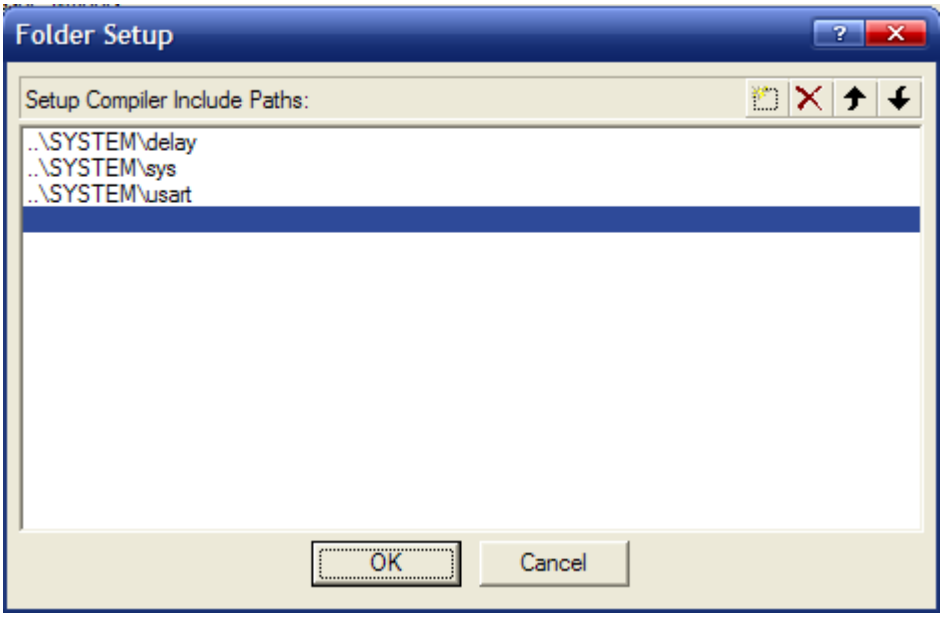


图 2.3.17 头文件包含路径设置

点击 OK 确认，回到 IDE，此时再点击按钮，再编译一次，发现没错误了，得到如下界面:

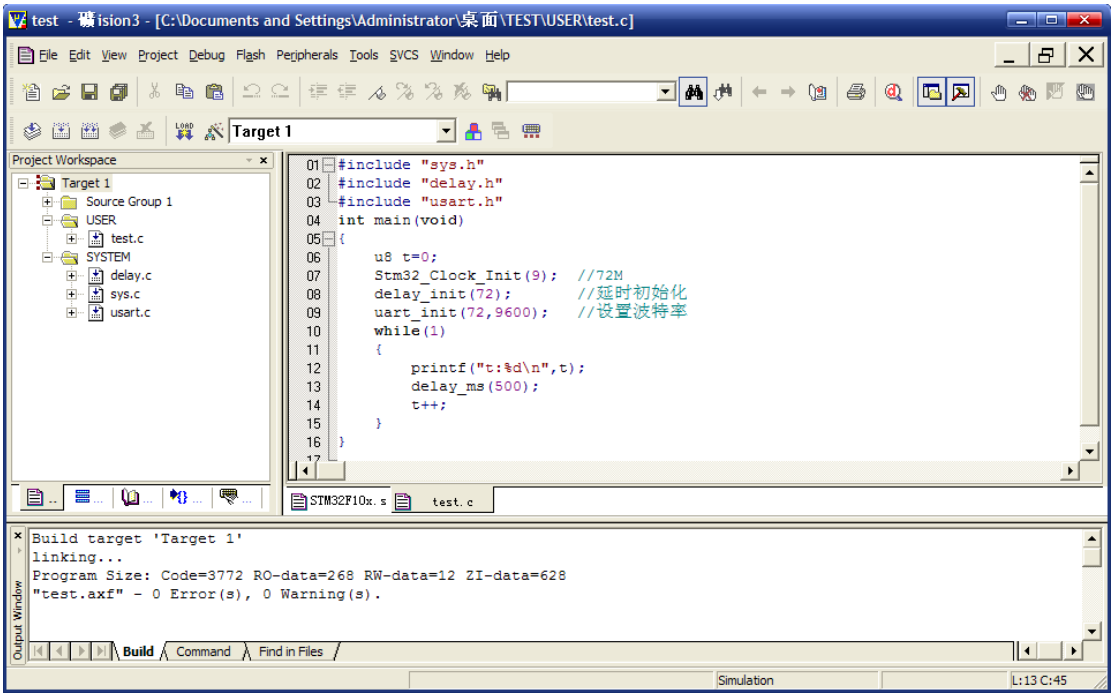


图 2.3.18 再次编译结果

至此，一个完整的 STM32 开发工程在 MDK 下建立了。接下来我们就是进行软件仿真了。是不是很简单?

2.4 软件仿真

MDK 的一个强大的功能就是提供软件仿真，通过软件仿真，我们可以发现很多将要出现的问题，避免了下载到 STM32 里面来查这些错误，这样最大的好处是能很方便的检查程序存在的问题，因为在 MDK 的仿真下面，你可以查看很多硬件相关的寄存器，通过观察这些寄存器，你可以知道代码是不是真正有效。另外一个优点是不必频繁的刷机，从而延长了 STM32 的 FLASH 寿命。当然，软件仿真不是万能的，很多问题还是要到在线调试才能发现。废话不多说了，接下来我们开始进行软件仿真。

在软件仿真之前，先检查一下配置是不是正确，点击  ，确定 Target 选项卡内容如下所示（主要检查芯片型号和晶振频率，其他的一般默认就可以）：

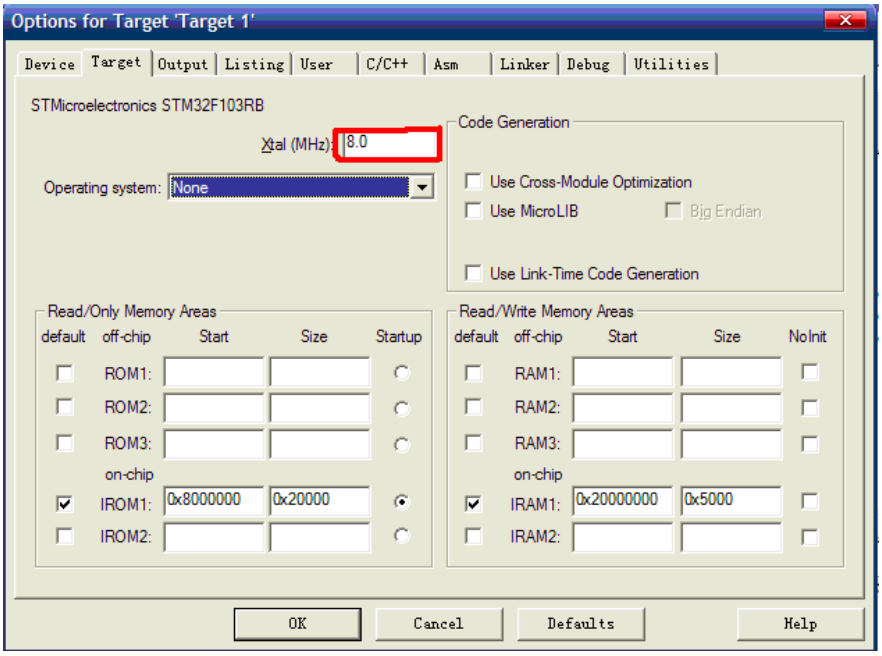


图 2.4.1 Target 选项卡

确认了芯片以及外部晶振频率（8.0M）之后，基本上就确定了硬件环境了，接下来，我们再看 Debug 选项卡，设置为入下图所示的设置：

图 2.4.2 Debug 选项卡

主要确认是 Use Simulator 是否选择（因为如果选择右边的 Use，那就是用 ULINK 进行硬件 Debug 了，这个将在下面介绍），其他的采用默认的就可以。确认了这项之后，我们便可以选择 OK，退出 Options for Target 对话框了。

接下来，我们点击  （开始/停止仿真按钮），开始仿真，出现如下界面:

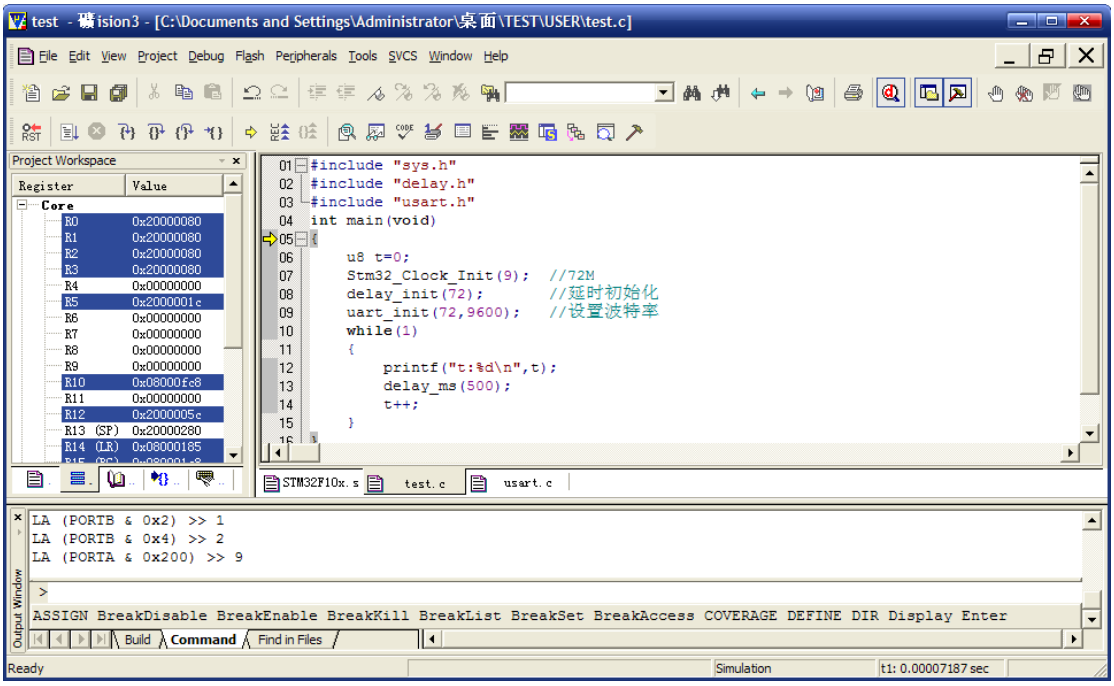


图 2.4.3 开始仿真

可以发现，多出了一个工具条，这个工具条对于我们仿真是非常有用的，下面简单介绍一下工具条相关按钮的功能，工具条部分按钮的功能如下图所示：

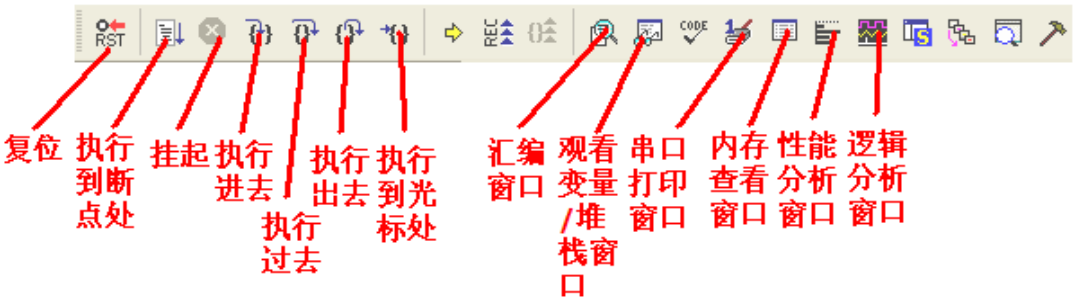


图 2.4.4 仿真工具条

复位：其功能等同于硬件上按复位按钮。相当于实现了一次硬复位。代码重新执行。

执行到断点处：该按钮用来快速执行到断点处，有时候你并不需要观看每步是怎么执行的，而是想快速的执行到程序的某个地方看结果，这个按钮就可以实现这样的功能。

挂起：此按钮在程序一直执行的时候会变为有效，通过按该按钮，就可以使程序停止下来，进入到单步调试状态。

执行进去：该按钮用来实现执行到某个函数里面去的功能，在没有函数的情况下，是等同于执行过去按钮的。

执行过去：在碰到有函数的地方，通过该按钮就可以单步执行过这个函数，而不进入这个函数单步执行。

执行出去：该按钮是在进入了函数单步调试的时候，有时候你可能不必再执行该函数的剩余部分了，通过该按钮就直接一步执行完函数余下的部分，并跳出函数，回到函数被调用的位置。

执行到光标处：该按钮可以迅速的使程序运行到光标处，其实是挺像执行到断点处按钮功能，但是两者是有区别的，断点可以有多个，但是光标所在处只有一个。

汇编窗口：通过该按钮，就可以查看汇编代码，这对分析程序很有用。

观看变量/堆栈窗口：该按钮按下，会弹出一个显示变量的窗口，在里面可以查看各种你想要看的变量值，也是很常用的一个调试窗口。

串口打印窗口：该按钮按下，会弹出一个串口调试助手界面的窗口，用来显示从串口打印出来的内容。

内存查看窗口：该按钮按下，会弹出一个内存查看窗口，可以在里面输入你要查看的内存地址，然后观察这一片内存的变化情况。是很常用的一个调试窗口

性能分析窗口：按下该按钮，会弹出一个观看各个函数执行时间和所占百分比的窗口，用来分析函数的性能是比较有用的。

逻辑分析窗口：按下该按钮会弹出一个逻辑分析窗口，通过 SETUP 按钮新建一些 IO 口，就可以观察这些 IO 口的电平变化情况，以多种形式显示出来，比较直观。

其他几个按钮用的比较少，以上是比较常用的，当然也不是每次都用得着这么多，具体看你程序调试的时候有没有必要观看这些东西，来决定要不要看。

这样，我们在上面的仿真界面里面选内存查看窗口、串口打印窗口。然后调节一下这两个窗口的位置，如下图所示：

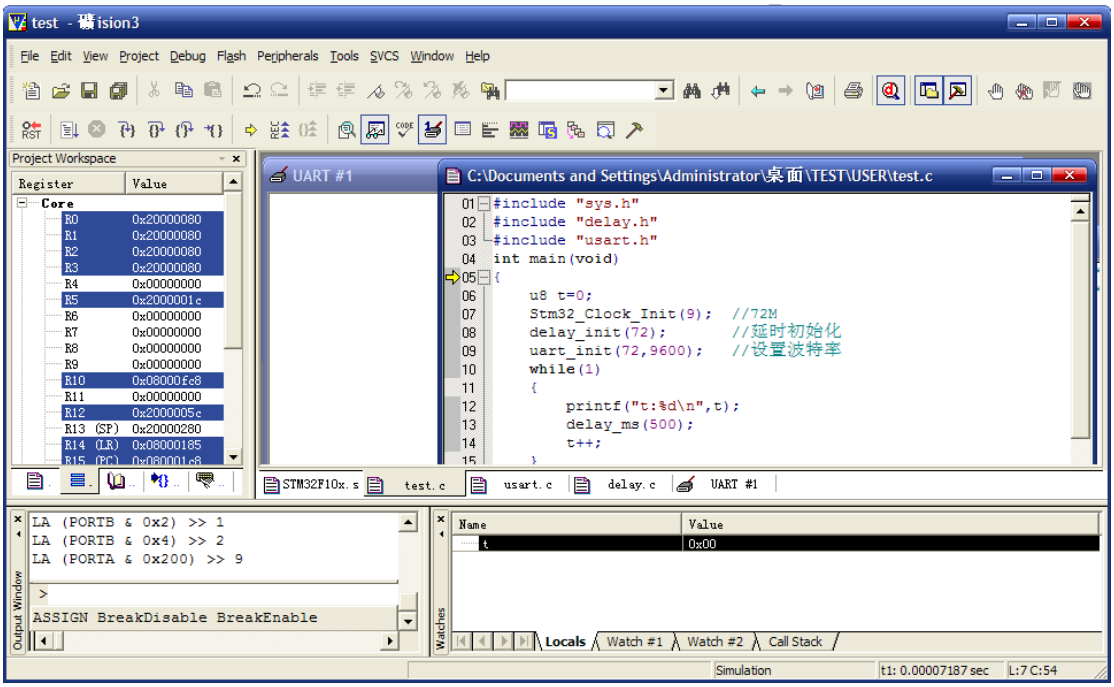


图 2.4.5 调出仿真串口打印窗口

我们把光标放到 test.c 的 09 行，然后双击鼠标左键，可以看到在 09 行的右边出现了一个红框，即表示设置了一个断点（也可以通过鼠标右键弹出菜单来加入，再次双击则取消）。 然后我们点击  ，执行到断点处，如下图所示：

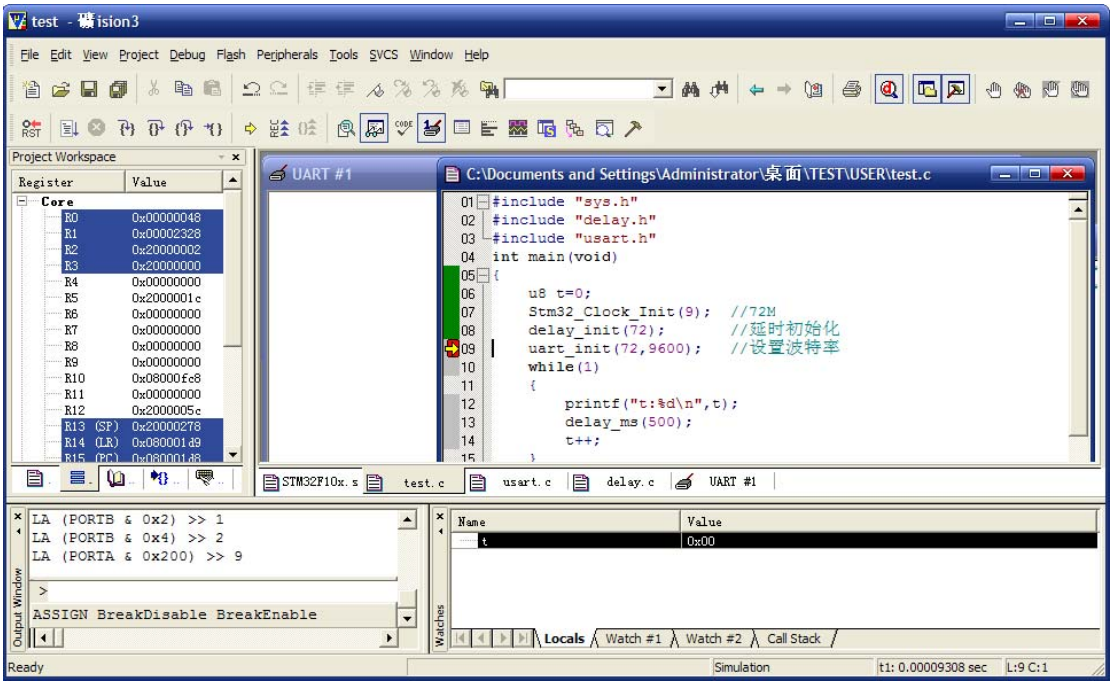


图 2.4.6 执行到断点处

我们不忙着往下执行，现在来点击菜单栏的 Peripherals->USARTs->USART 1。可以看到，有很多外设可以查看，这里我们查看的是串口 1 的情况。如下图所示：

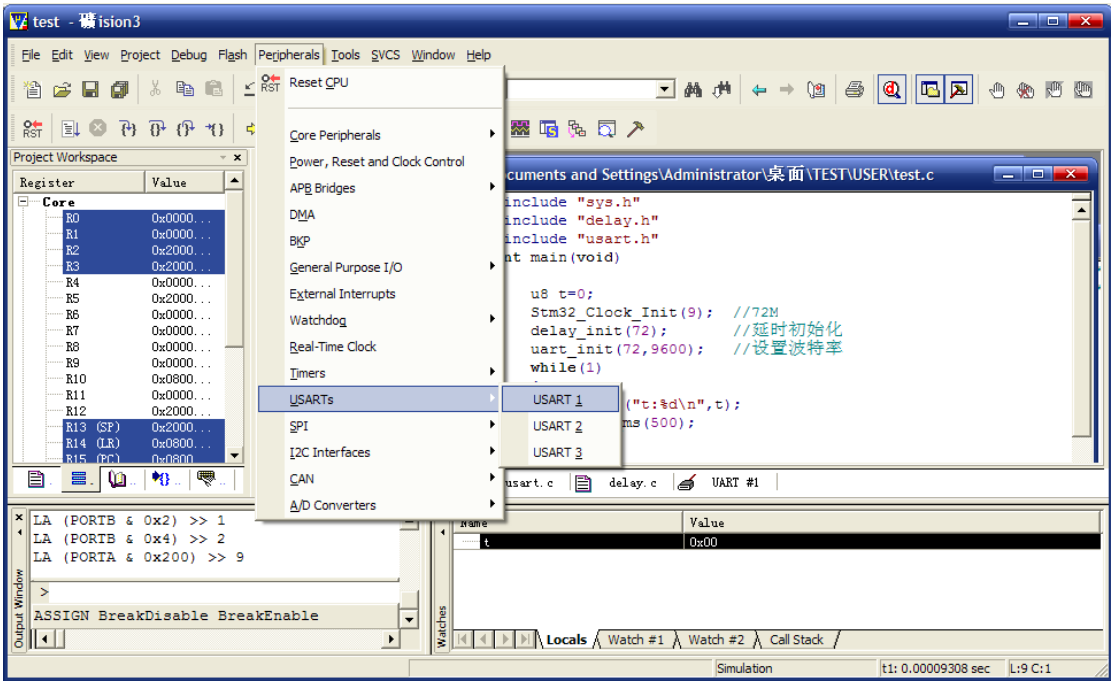


图 2.4.7 查看串口 1 相关寄存器

单击 USART1 后会在 IDE 之外出现一个如下界面:

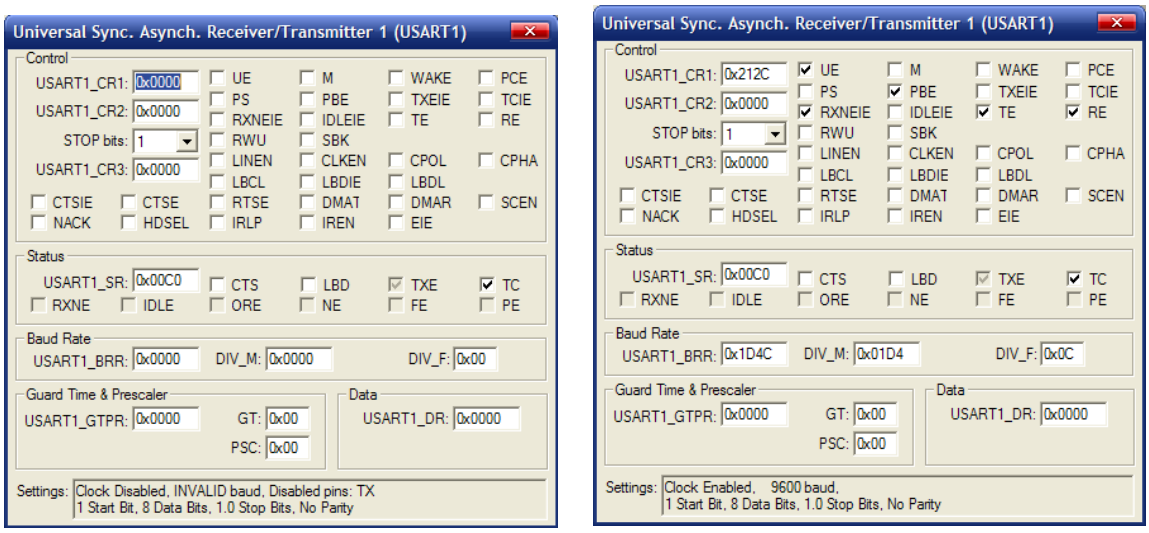


图 2.4.8 串口 1 各寄存器初始化前后对比

左边这个就是 STM32 默认时候，串口 1 的状态，从中可以看到所有与串口相关的寄存器全部在这上面表示出来了，而且有当前串口的波特率等信息的显示。我们接着单击一下 {}，执行完串口初始化函数，得到了如上面右边图片所示的串口信息。你可以对比一下这两个的区别，就知道在 uart_init(72，9600);这个函数里面大概执行了哪些操作。

这样可以很清楚的告诉你，当前的串口是否可用，你的设置是否正确，同样这样的方法也可以适用于很多其他外设，这个读着慢慢体会吧！总之这个是很有用的。

然后我们继续单击 {} 按钮，一步步执行，最后就会看到在 USART #1 中打印出相关的信息，如下图所示：

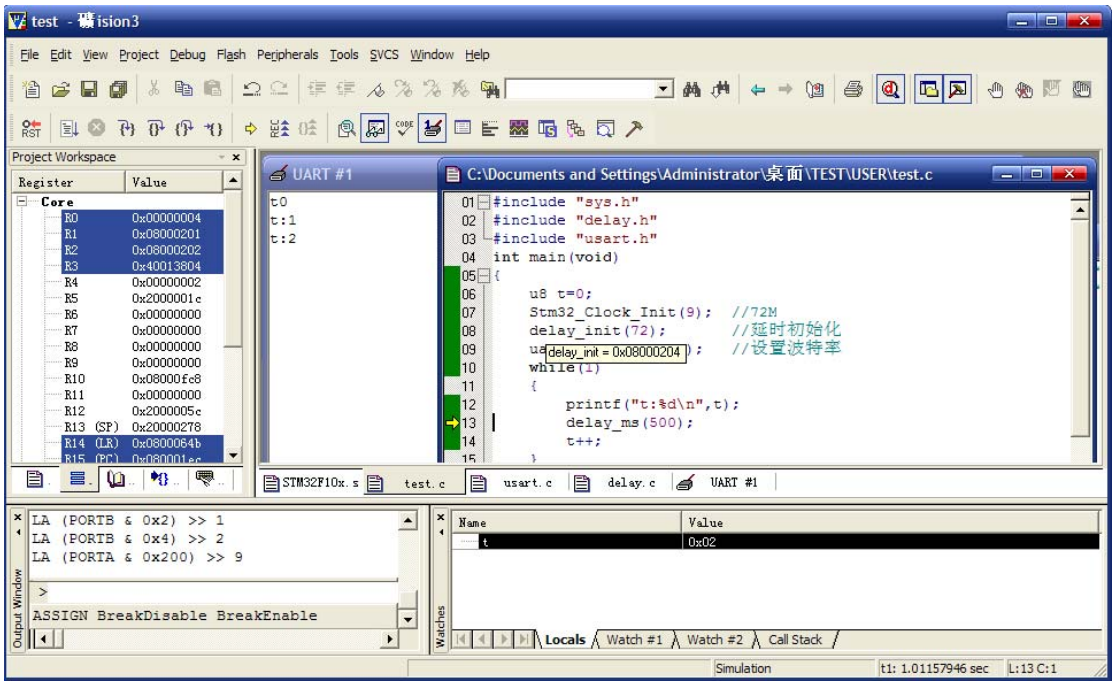


图 2.4.9 串口 1 输出信息

这样证明我们的仿真是通过的，代码运行会在串口 1 不停的输出 t 的值，每 0.5s 执行一次

（时间可以通过 IDE 的最下面，观看到，如下图所示），并且 t 自增。与我们预期的目地是一致的。再次按下  结束仿真。

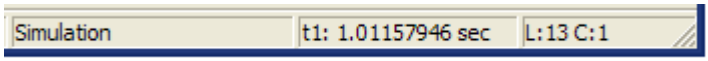


图 2.4.10 仿真持续时间

至此，我们软件仿真算是结束了，接下来我们下载代码到硬件上来真正验证一下我们的代码是否可行。

2.5 程序下载

STM32 的程序下载有多种方法：USB、串口、JTAG、SWD 等，这几种方式，都可以用来给 STM32 下载代码。不过，我们最常用的，最经济的，就是通过串口给 STM32 下载代码。本节，我们将向大家介绍，如何利用串口给 STM32 下载代码。

STM32 的串口下载一般是通过串口 1 下载的， 本手册的实验平台 ALIENTEK MiniSTM32 开发板，不是通过 RS232 串口下载的，而是通过自带的 USB 串口来下载。看起来像是 USB 下载（只有一根 USB 线，并没有串口线）的，实际上，是通过 USB 转成串口，然后再下载的。

下面，我们就一步步教大家如何在实验平台上利用 USB 串口来下载代码。

首先要在板子上设置一下，在板子上把 RXD 和 PA9（STM32 的 TXD），TXD 和 PA10(STM32 的 RXD)通过跳线帽连接起来，这样我们就把 PL2303 和 MCU 的串口 1 连接上了。这里由于 ALIENTEK 这款开发板（V1.8 及以后版本）自带了一键下载电路，所以我们并不需要去关心 BOOT0 和 BOOT1 的状态，但是为了让下下载完后可以按复位执行程序，我们建议大家把 BOOT1 和 BOOT0 都设置为 0。设置完成如下图所示：

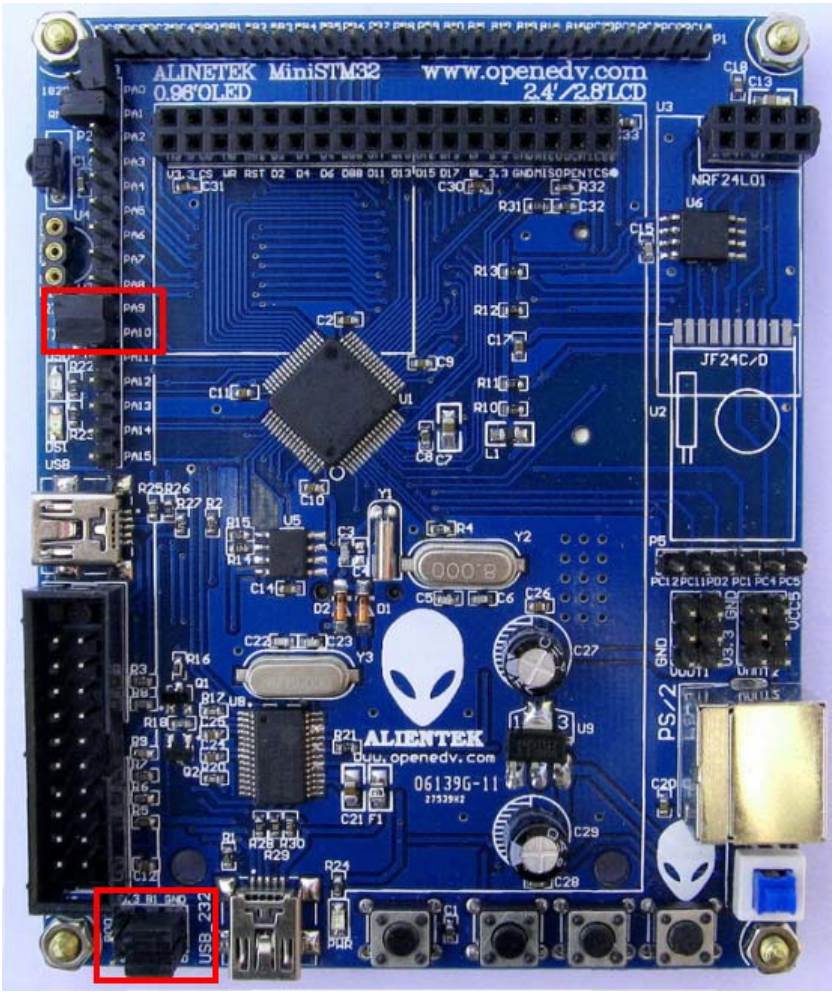


图 2.5.1 开发板串口下载跳线设置

接着我们在 USB_232 处插入 USB 线，并接上电脑，如果之前没有安装 PL2303 的驱动（如果已经安装过了驱动，则应该能在设备管理器里面看到 USB 串口，如果不能则要先卸载之前的驱动，卸载完后重启电脑（必要步骤），再重新安装我们提供的驱动），则电脑会提示找到新

硬件，如下图所示：

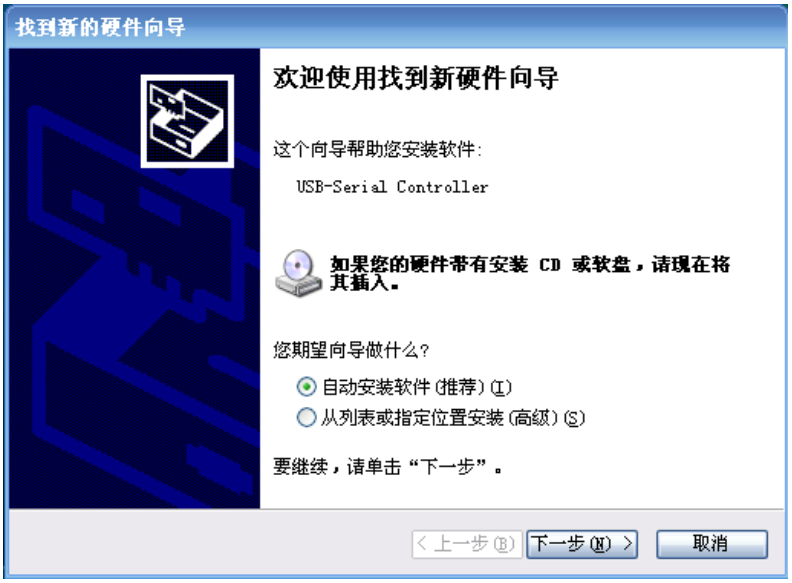


图 2.5.2 找到新硬件

我们不理会这个提示，直接找到光盘->软件文件夹下的 PL-2303 驱动，安装该驱动。在驱动安装完成之后，拔掉 USB 线，然后重新插入电脑，此时电脑就会自动给其安装驱动了。在安装完成之后，可以在电脑的设备管理器里面找到 USB 串口（如果找不到，则重启下电脑），如下图所示：

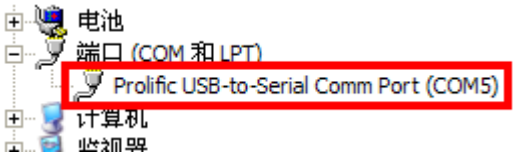


图 2.5.3 USB 串口驱动安装成功

上图中我们显示的 USB 串口为 COM5，不同电脑可能不一样，你的可能是 COM4、COM6 等，但是 Prolific USB-to-Serial Comm Port，这个一定是一样的，如果没找到，则有可能是你安装有误，或者系统不兼容。

在安装了USB串口驱动之后，我们就可以开始串口下载程序了，这里我们的串口下载软件选择的是mcuisp，该软件属于第三方软件，由单片机在线编程网提供，大家可以去 www.mcuisp.com 免费下载，我们光盘也附带了这个软件，目前最新版本为V0.993。该软件启动界面如下：

图 2.5. 4 mcuisp 启动界面

然后我们选择要下载的 HEX 文件，以前面我们新建的工程为例，因为我们前面的工程没有在 KEIL 里面设置生成.hex 文件，所以在 USER 文件夹下是找不到.hex 文件的。下面我们看看如何设置：先在工程里面点击  ，打开 Output 选项卡，勾选 Create HEX File 选项，如下图所示：

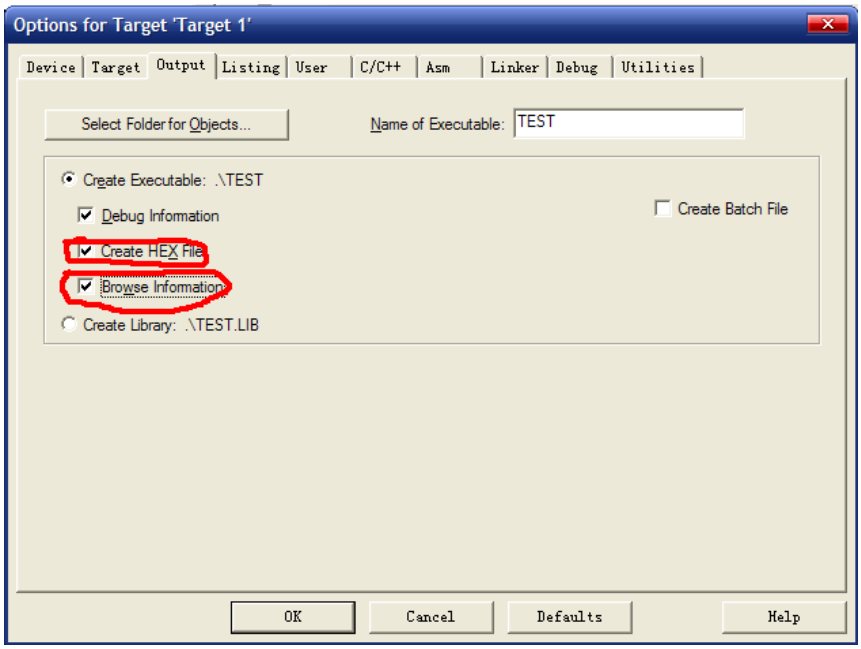


图 2.5.5 MDK 设置生成 HEX 文件

图中选中第二个红圈内的选项，可以使编译器产生浏览信息，方便快速查看函数和变量等，这点在后面会介绍。选中之后点击 OK，重新编译，编译结果如下图所示：

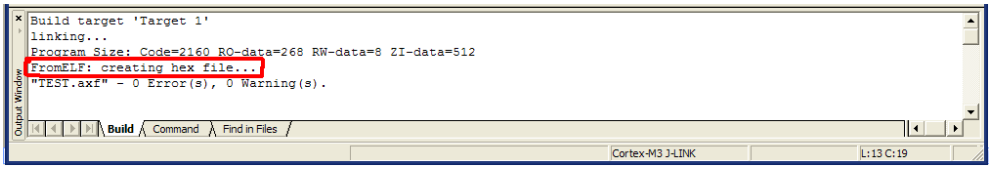


图 2.5.1.6 重新编译结果

从上图中可以看到，编译器已经产生了 hex 文件了。然后我们打开 USER 文件夹，看看里面发生了什么变化？如下图所示：

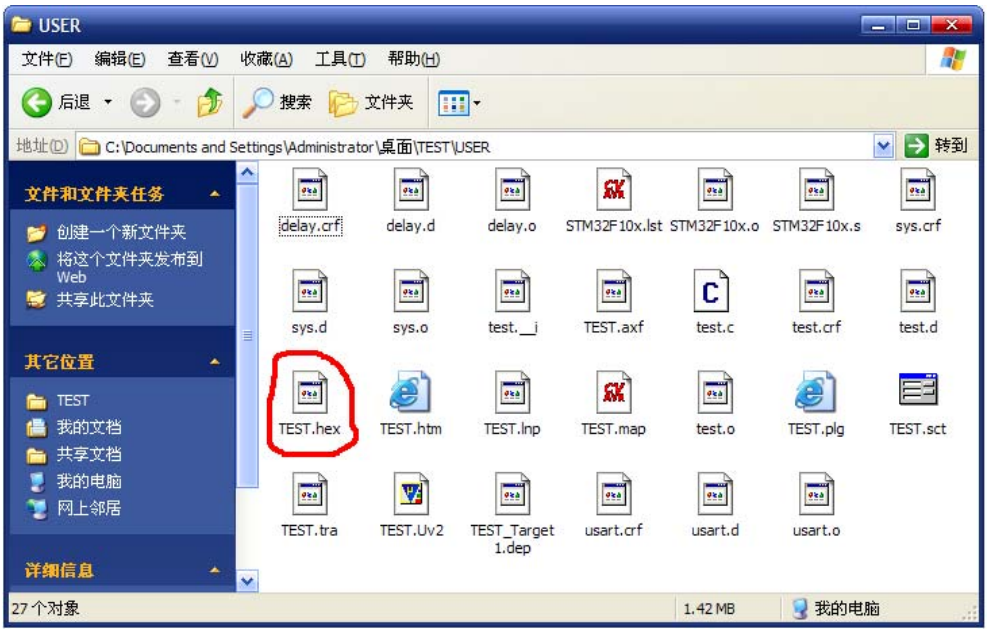


图 2.5.7 新编译生成文件

从上图可以看到，重新编译产生了很多文件，其中就有我们所需要的 HEX 文件（图中红圈圈中），至此，我们就可以开始下载了。

用 mcuisp 软件打开 USER 文件夹，找到 TEST.hex，打开并进行相应设置后，如下图所示：

图 2.5.8 mcuisp 设置

图 2.5.8 中红圈圈中的设置，是我们建议的设置。编程后执行，这个选项在无一键下载功能的条件下是很有用的，当选中该选项之后，可以在下载完程序之后自动运行代码。否则，下载

代码之后，必须先将 B0 连接 GND，再按复位键，才能开始运行刚刚下载的代码。ALIENTEK 开发板虽然自带了一键下载功能，但是还是建议选上这个设置。

编程前重装文件，该选项也比较有用，当选中该选项之后，mcuisp 会在每次编程之前，将 hex 文件重新装载一遍，这对于代码调试的时候是比较有用的。

最后，我们选择的 DTR 的低电平复位，RTS 高电平进 BootLoader，这个选择项选中，mcuisp 就会通过 DTR 和 RTS 信号来控制板载的一键下载功能电路，以实现一键下载功能。如果不选择，则无法实现一键下载功能。这个是必要的选项（在 BOOT0 接 GND 的条件下）。

在装载了 hex 文件之后，我们要下载代码还需要选择串口，这里 mcuisp 有智能串口搜索功能。每次打开 mcuisp 软件，软件会自动去搜索当前电脑上可用的串口，然后选中一个作为默认的串口（一般是您最后一次关闭时所选则的串口）。也可以通过点击菜单栏的搜索串口，来实现自动搜索当前可用串口。串口波特率则可以通过 bps 那里设置，对于 STM32，该波特率最大为 230400bps，这里我们一般选择最高的波特率：460800，让 mcuisp 自动去同步。搜索完串口之后界面如下图所示：

图 2.5.9 搜索串口

从之前 USB 串口的安装可知，开发板的串口被识别为 COM5 了（如果那您的是被识别为其他的串口，则选择相应的串口即可），所以我们选择 COM5。选择了相应串口之后，我们就可以通过按**开始编程（P）**这个按钮，一键下载代码到 STM32 上，下载成功后如下图所示：

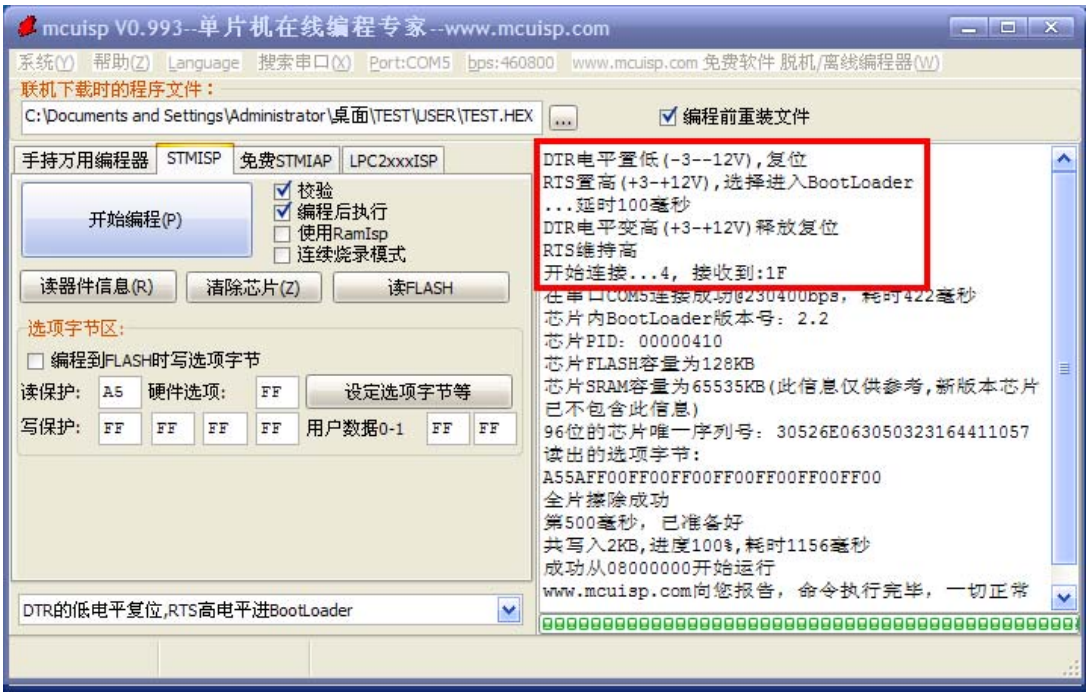


图 2.5.10 下载完成

图 2.5.10 中，我们用圈圈圈出了 mcuisp 对一键下载电路的控制过程，其实就是控制 DTR 和 RTS 电平的变化，控制 BOOT0 和 RESET，从而实现自动下载。另外界面提示已经下载完成（如果老提示：开始连接…，需要检查一下，开发板的设置是否正确，是否有其他因素干扰等），并且从 0X80000000 处开始运行了，我们打开串口调试助手选择 COM5，会发现从 ALIENTEK MiniSTM32 开发板发回来的信息，如下图所示：

图 2.5.11 程序开始运行了

接收到的数据和我们仿真的是一样的，证明程序没有问题。这里说一个小技巧，mcuisp 没有做复位 STM32 的按钮，不过我们可以通过按读取器件信息来执行一次硬件复位操作，这样

就可以在电脑上直接控制开发板的复位了，另外还不需要浪费 STM32 的 flash 寿命。至此，说明我们下载代码成功了，并且也从硬件上验证了我们代码的正确性。

2.6 在线调试

利用串口，我们只能下载程序，并不能实时跟踪，而利用调试工具，比如 JLINK、ULINK 等就可以实时跟踪程序，使你的开发事半功倍。这里我们以 JLINK V8 为例，说说如何在线调试。

JLINK V8 支持 JTAG 和 SWD，而 STM32 也支持 JTAG 和 SWD。所以，我们有 2 种方式可以用来调试，JTAG 调试的时候，占用的 IO 线比较多，而 SWD 调试的时候占用的 IO 线很少，只需要 2 跟即可。

JLINKV8 的安装我们这里就不说了（请参考光盘视频教材：《入门（KEIL 安装+JLINK 调试+串口下载+新建工程）.rmvb》）。在安装了 JLINK V8 之后，我们接上 JLINK-V8，并把 JTAG 口插到 ALIENTEK MiniSTM32 开发板上，打开之前 2.3 节新建的工程，点击  ，打开 Options for Target 选项卡，在 Debug 栏选择仿真工具为 Cortex-M3 J-LINK，如下图所示：

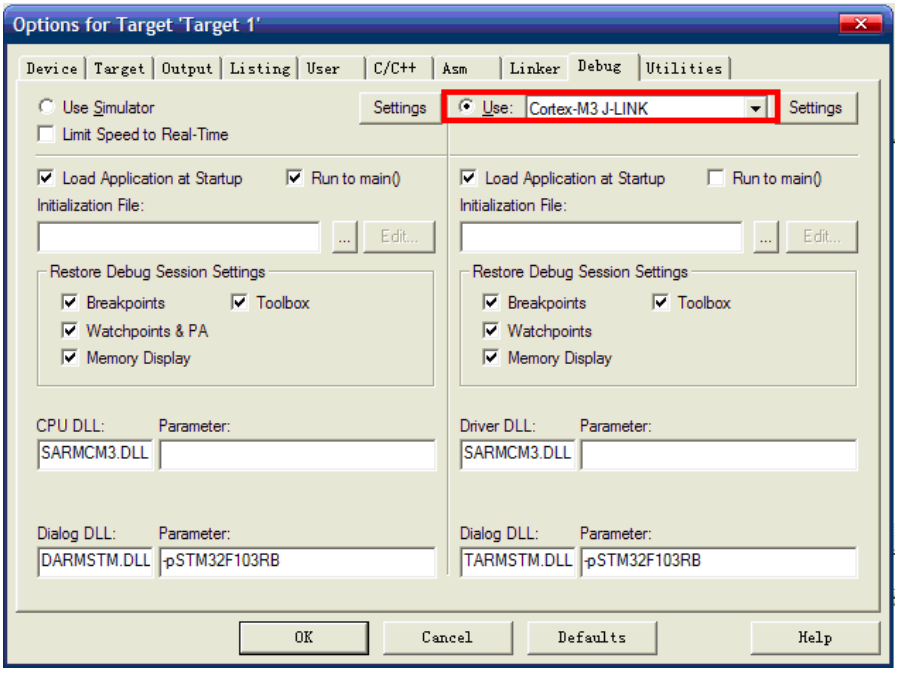


图 2.6.1 Debug 选项卡设置

然后我们点击 Settings，设置 J-LINK 的一些参数，如下图所示：

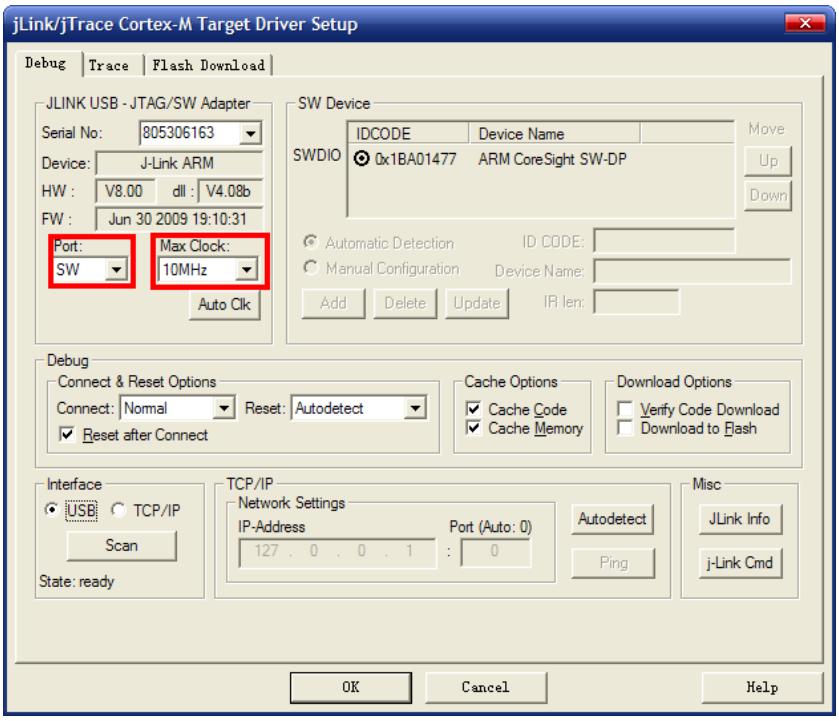


图 2.6.2 J-LINK 模式设置

上图中，我们使用 J-LINK V8 的 SW 模式调试，因为我们 JTAG 需要占用比 SW 模式多很多的 IO 口，而在 ALIENTEK MiniSTM32 开发板上这些 IO 口都是非常有用的，并造成部分外设无法使用。所以，我们建议大家在调试的时候，一定要选择 SW 模式。Max Clock，可以点击 Auto Clk 来自动设置，图 4.23 中我们设置 JLINK 的调试速度为 10Mhz，这里，如果您的 USB 数据线比较差，那么可能会出问题，此时，您可以通过降低这里的速率来试试。

单击 OK，完成此部分设置，接下来我们还需要在 Utilities 选项卡里面设置下载时的目标编程器，如下图所示：

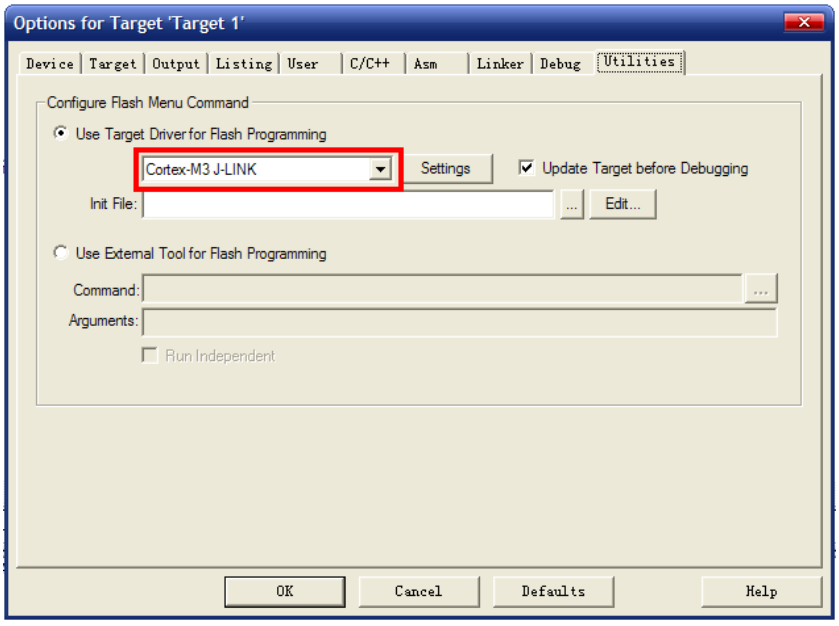


图 2.6.3 FLASH 编程器选择

上图中，我们选择 J-LINK 来调试 Cortex M3，然后点击 Settings，设置如下图所示：

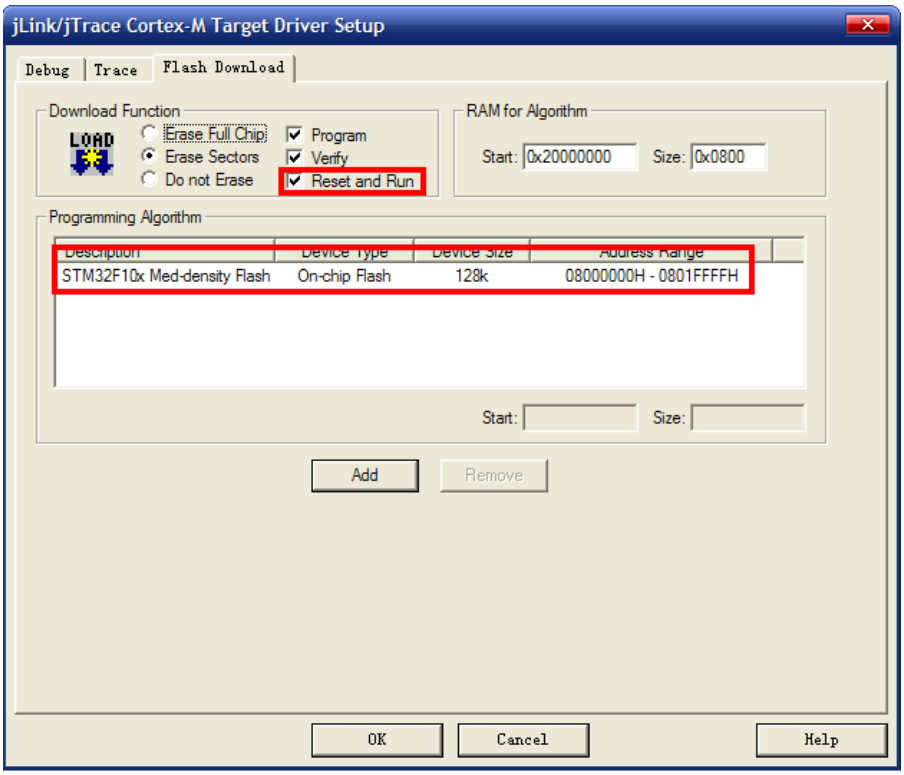


图 2.6.4　编程设置

这里要根据不同的 MCU 选择 FLASH 的大小，因为我们开发板使用的是 STM32F103RBT6，其 FLASH 大小为 128K，所以我们在 Programming Algorithm 里面选择 128K 型号的 STM32。然后选中 Reset and Run，以实现在编程后自动启动。其他默认设置即可。

在设置完之后，点击 OK，然后再点击 OK，回到 IDE 界面，编译一下工程。再点击，开始仿真（如果开发板的代码没被更新过，则会先更新代码，再仿真，你也可以通过按，只下载代码，而不进入仿真，特别注意：开发板上的 B0 要设置到 GND，否则代码下载后不会自动运行的！），如下图所示：

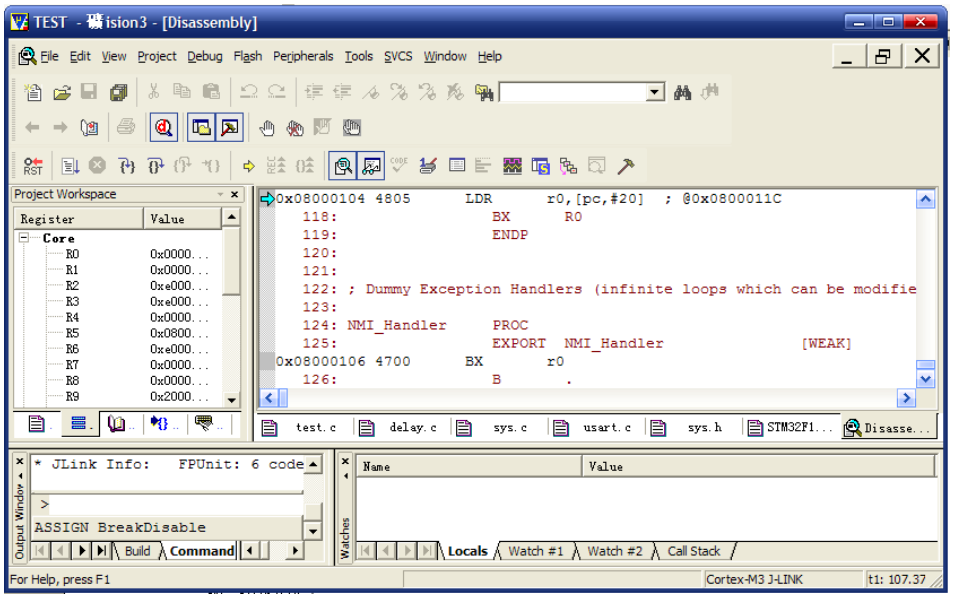


图 2.6.5　开始仿真

可以看到都是一些汇编码的查看，如果我们要快速运行到 main 函数，可以在 main 函数的第一句语句处放入断点，然后点击，来快速执行到该处。如下图所示：

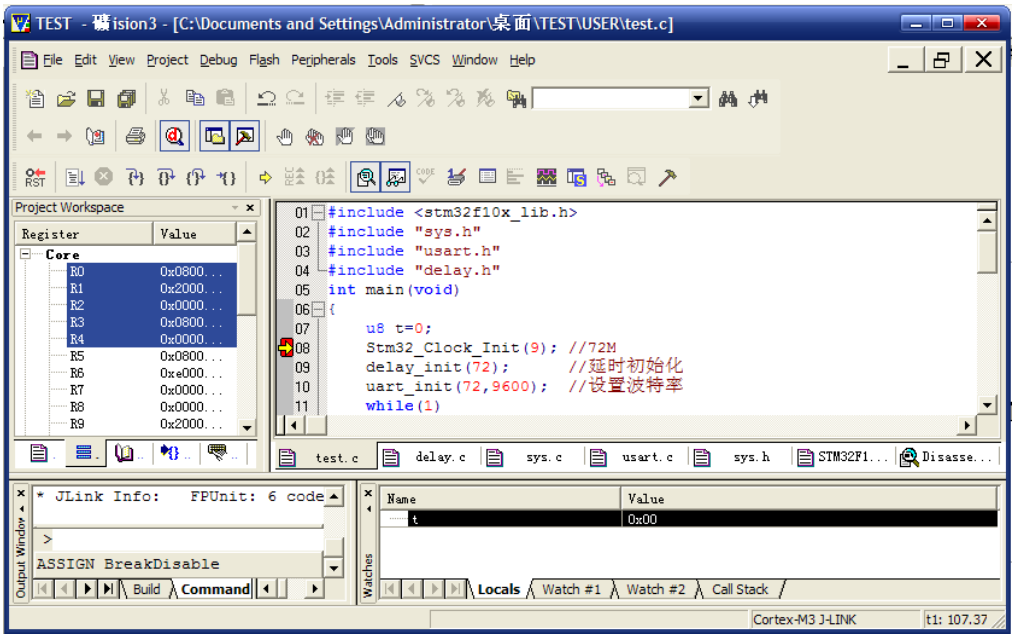


图 2.6.6　程序运行到断点处

接下来，我们就可以和软件仿真一样的开始仿真了，不过这是真正的在硬件上的仿真，其结果更可信。在线仿真就给大家介绍到这里。

2.7 SYSTEM文件夹介绍

前面，我们介绍了如何在 MDK3.80A 下建立 STM32 工程，在这个新建的工程之中，我们用到了一个 SYSTEM 文件夹里面的代码，此文件夹里面的代码由 ALIENTEK 提供，是 STM32F103 系列的底层核心驱动函数，可以用在 STM32F103 系列的各个型号上面，方便大家快速构建自己的工程。

SYSTEM 文件夹下包含了 delay、sys、usart 等三个文件夹。分别包含了 delay.c、sys.c、usart.c 及其头文件。通过这 3 个 c 文件，可以快速的给任何一款 STM32 构建最基本的框架。使用起来是很方便的。

本节，我们将向大家介绍这些代码，通过本节的学习，大家将了解到这些代码的由来，也希望大家可以灵活使用 SYSTEM 文件夹提供的函数，来快速构建工程，并实际应用到自己的项目中去。

2.7.1 delay 文件夹

delay 文件夹内包含了 delay.c 和 delay.h 两个文件，这两个文件用来实现系统的延时功能，其中包含 3 个函数：

void delay_init(u8 SYSCLK);

void delay_ms(u16 nms);

void delay_us(u32 Nus);

下面分别介绍这三个函数，在介绍之前，我们先了解一下编程思想：CM3 内核的处理器，内部包含了一个 SysTick 定时器，SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在 STM32 的参考手册里面介绍的很简单，其详细介绍，请参阅《Cortex-M3 权威指南》第 133 页。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

1）delay_init 函数

该函数用来初始化 2 个重要参数：fac_us 以及 fac_ms；同时吧 SysTick 的时钟源选择外部时钟。具体代码如下：

```
//初始化延迟函数
//SYSTICK 的时钟固定为 HCLK 时钟的 1/8
//SYSCLK:系统时钟
void delay_init(u8 SYSCLK)
{
    SysTick->CTRL&=0xfffffffb;//bit2 清空，选择外部时钟  HCLK/8
    fac_us=SYSCLK/8;
    fac_ms=(u16)fac_us*1000;
}
```

SysTick 是 MDK 定义了的一个结构体（在 stm32f10x_map.里面），里面包含 CTRL、LOAD、VAL、CALIB 等 4 个寄存器，

SysTick->CTRL 的各位定义如下图所示：

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经数到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

图 2.7.1.1 SysTick->CTRL 寄存器各位定义

SysTick-> LOAD 的定义如下图所示：

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时，将被重装载的值

图 2.7.1.2 SysTick->LOAD 寄存器各位定义

SysTick-> VAL 的定义如下图所示：

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

图 2.7.1.3 SysTick->VAL 寄存器各位定义

SysTick-> CALIB 不常用，在这里我们也用不到，故不介绍了。

SysTick->CTRL&=0xfffffffb;这一句把 SysTick 的时钟选择外部时钟，这里需要注意的是 SysTick 的时钟源自 HCLK 的 8 分频，假设我们外部晶振为 8M，然后倍频到 72M，那么 SysTick 的时钟即为 9Mhz。

fac_us，为 us 延时的基数，也就是延时 1us，SysTick->LOAD 所应设置的值。fac_ms 为 ms 延时的基数，也就是延时 1ms，SysTick->LOAD 所应设置的值。fac_us 为 8 位整形数据，fac_ms 为 16 位整形数据。正因为如此，系统时钟如果不是 8 的倍数，则会导致延时函数不准确，这也是我们推荐外部时钟选择 8M 的原因。这点大家要特别留意。

2）delay_us 函数

该函数用来延时指定的 us，其参数 nus 为要延时的微秒数。具体函数如下：

```
//延时 us
void delay_us(u32 nus)
{
    u32 temp;
    SysTick->LOAD=nus*fac_us; //时间加载
    SysTick->VAL=0x00;          //清空计数器
    SysTick->CTRL=0x01 ;        //开始倒数
    do
    {
        temp=SysTick->CTRL;
```

```
    }
    while(temp&0x01&&!(temp&(1<<16)));//等待时间到达
    SysTick->CTRL=0x00;          //关闭计数器
    SysTick->VAL =0X00;          //清空计数器
}
```

有了上面对 SysTick 寄存器的描述，这段代码不难理解。其实就是先把要延时的 us 数换算成 SysTick 的时钟数，然后写入 LOAD 寄存器。然后清空当前寄存器 VAL 的内容，再开启倒数功能。等到倒数结束，即延时了 nus。最后关闭 SysTick，清空 VAL 的值。实现一次延时 nus 的操作，但是这里要注意 nus 的值，不能太大，必须保证 nus<=（2^24）/fac_us，否则将导致延时时间不准确。

3）delay_ms 函数

该函数用来延时指定的 ms，其参数 nms 为要延时的微秒数。具体函数如下：

```
//延时 nms
//注意 nms 的范围
//SysTick->LOAD 为 24 位寄存器，所以，最大延时为:
//nms<=0xffffff*8*1000/SYSCLK
//SYSCLK 单位为 Hz，nms 单位为 ms
//对 72M 条件下，nms<=1864
void delay_ms(u16 nms)
{
    u32 temp;
    SysTick->LOAD=(u32)nms*fac_ms;//时间加载(SysTick->LOAD 为 24bit)
    SysTick->VAL =0x00;              //清空计数器
    SysTick->CTRL=0x01 ;             //开始倒数
    do
    {
        temp=SysTick->CTRL;
    }
    while(temp&0x01&&!(temp&(1<<16)));//等待时间到达
    SysTick->CTRL=0x00;          //关闭计数器
    SysTick->VAL =0X00;          //清空计数器
}
```

此部分代码和 7.2 节的 delay_us 大致一样，但是要注意因为 LOAD 仅仅是一个 24bit 的寄存器，延时的 ms 数不能太长。否则超出了 LOAD 的范围，高位会被舍去，导致延时不准。最大延迟 ms 数可以通过公式：nms<=0xffffff*8*1000/SYSCLK 计算。SYSCLK 单位为 Hz，nms 的单位为 ms。如果时钟为 72M，那么 nms 的最大值为 1864ms。超过这个值就会导致延时不准确。

2.7.2 sys 文件夹

sys 文件夹内包含了 sys.c 和 sys.h 两个文件。在 sys.h 里面定义了 STM32 的 IO 口输入读取宏定义和输出宏定义。sys.c 里面定义了很多与 STM32 底层硬件很相关的设置函数，包括系统时钟的配置，中断的配置等。下面我们分别介绍。

1）IO 口的位操作实现

该部分代码实现对 STM32 各个 IO 口的位操作，包括读入和输出。当然在这些函数调用之前，必须先进行 IO 口时钟的使能和 IO 口功能定义。此部分仅仅对 IO 口进行输入输出读取和控制。代码如下：

```
#define BITBAND(addr , bitnum) ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *)(addr))
#define BIT_ADDR(addr， bitnum) MEM_ADDR(BITBAND(addr， bitnum))
//IO 口地址映射
#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 GPIOF_ODR_Addr (GPIOF_BASE+12) //0x40011A0C
#define GPIOG_ODR_Addr (GPIOG_BASE+12) //0x40011E0C

#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 GPIOF_IDR_Addr (GPIOF_BASE+8) //0x40011A08
#define GPIOG_IDR_Addr (GPIOG_BASE+8) //0x40011E08

//IO 口操作，只对单一的 IO 口!
//确保 n 的值小于 16!
#define PAout(n) BIT_ADDR(GPIOA_ODR_Addr，n) //输出
#define PAin(n) BIT_ADDR(GPIOA_IDR_Addr，n) //输入

#define PBout(n) BIT_ADDR(GPIOB_ODR_Addr，n) //输出
#define PBin(n) BIT_ADDR(GPIOB_IDR_Addr，n) //输入

#define PCout(n) BIT_ADDR(GPIOC_ODR_Addr，n) //输出
#define PCin(n) BIT_ADDR(GPIOC_IDR_Addr，n) //输入

#define PDout(n) BIT_ADDR(GPIOD_ODR_Addr，n) //输出
#define PDin(n) BIT_ADDR(GPIOD_IDR_Addr，n) //输入
```

```
#define PEout(n)    BIT_ADDR(GPIOE_ODR_Addr，n)  //输出
#define PEin(n)     BIT_ADDR(GPIOE_IDR_Addr，n)  //输入

#define PFout(n)    BIT_ADDR(GPIOF_ODR_Addr，n)  //输出
#define PFin(n)     BIT_ADDR(GPIOF_IDR_Addr，n)  //输入

#define PGout(n)    BIT_ADDR(GPIOG_ODR_Addr，n)  //输出
#define PGin(n)     BIT_ADDR(GPIOG_IDR_Addr，n)  //输入
```

以上代码的实现得益于 CM3 的位带操作，具体的实现比较复杂，请参考<<CM3 权威指南>>第五章(87 页~92 页)。有了上面的代码，我们就可以像 51/AVR 一样操作 STM32 的 IO 口了。比如，我要 PORTA 的第七个 IO 口输出 1，则可以使用 PAout（6）=1；即可实现。我要判断 PORTA 的第 15 个位是否等于 1，则可以使用 if（PAin（14）==1）…；就可以了。

这里顺便说一下，在 sys.h 中的几个其他的全局宏定义，他们是：

```
//Ex_NVIC_Config 专用定义
#define GPIO_A 0
#define GPIO_B 1
#define GPIO_C 2
#define GPIO_D 3
#define GPIO_E 4
#define GPIO_F 5
#define GPIO_G 6
#define FTIR    1  //下降沿触发
#define RTIR    2  //上升沿触发
//JTAG 模式设置定义
#define JTAG_SWD_DISABLE    0X02
#define SWD_ENABLE          0X01
#define JTAG_SWD_ENABLE     0X00
```

这些宏定义在后面的使用中会不时的用到，他们分别是作为 Ex_NVIC_Config 函数和 JTAG_Set 函数的参数来使用的，具体见相关函数的说明。

2）Stm32_Clock_Init 函数

该函数的主要功能就是初始化 STM32 的时钟。其中还包括对向量表的配置，以及相关外设的复位及配置。其代码如下：

```
//系统时钟初始化函数
//pll:选择的倍频数，从 2 开始，最大值为 16
void Stm32_Clock_Init(u8 PLL)
{
    unsigned char temp=0;
    MYRCC_DeInit();          //复位并配置向量表
    RCC->CR|=0x00010000;  //外部高速时钟使能 HSEON
    while(!(RCC->CR>>17));//等待外部时钟就绪
    RCC->CFGR=0X00000400; //APB1/2=DIV2;AHB=DIV1;
    PLL-=2;//抵消 2 个单位
```

```
    RCC->CFGR|=PLL<<18;     //设置 PLL 值 2~16
    RCC->CFGR|=1<<16;       //PLLSRC ON
    FLASH->ACR|=0x32;       //FLASH 2 个延时周期

    RCC->CR|=0x01000000;    //PLLON
    while(!(RCC->CR>>25));//等待 PLL 锁定
    RCC->CFGR|=0x00000002;//PLL 作为系统时钟
    while(temp!=0x02)       //等待 PLL 作为系统时钟设置成功
    {
        temp=RCC->CFGR>>2;
        temp&=0x03;
    }
}
```

Stm32_Clock_Init 函数只有一个变量 PLL，就是用来配置时钟的倍频数的，比如当前所用的晶振为 8Mhz，PLL 的值设为 9，那么 STM32 将运行在 72M 的速度下。关于时钟的详细介绍，在《STM32 参考手册》第 6.2 节（55~60 页）有详细介绍。有不明白的地方，可以对照手册仔细研究。

MYRCC_DeInit 函数实现外设的复位，并关断所有中断，同时调用向量表配置函数 MY_NVIC_SetVectorTable，配置中断向量表。MYRCC_DeInit 函数如下：

```
//不能在这里执行所有外设复位!否则至少引起串口不工作.
//把所有时钟寄存器复位
void MYRCC_DeInit(void)
{
    RCC->APB1RSTR = 0x00000000;//复位结束
    RCC->APB2RSTR = 0x00000000;

    RCC->AHBENR = 0x00000014;  //睡眠模式闪存和 SRAM 时钟使能.其他关闭.
    RCC->APB2ENR = 0x00000000; //外设时钟关闭.
    RCC->APB1ENR = 0x00000000;
    RCC->CR |= 0x00000001;      //使能内部高速时钟 HSION
    RCC->CFGR &= 0xF8FF0000;
//复位 SW[1:0]，HPRE[3:0]，PPRE1[2:0]，PPRE2[2:0]，ADCPRE[1:0]，MCO[2:0]

    RCC->CR &= 0xFEF6FFFF;      //复位 HSEON，CSSON，PLLON
    RCC->CR &= 0xFFFBFFFF;      //复位 HSEBYP
    RCC->CFGR &= 0xFF80FFFF;    //复位 PLLSRC， PLLXTPRE， PLLMUL[3:0] and
USBPRE
    RCC->CIR = 0x00000000;      //关闭所有中断
    //配置向量表
#ifdef  VECT_TAB_RAM
    MY_NVIC_SetVectorTable(NVIC_VectTab_RAM， 0x0);
#else
    MY_NVIC_SetVectorTable(NVIC_VectTab_FLASH， 0x0);
```

```
#endif
}
```

RCC 也是 MDK 定义的一个结构体，包含 RCC 相关的寄存器组。其寄存器名与 STM32 参考手册里面定义的寄存器名字是一摸一样的，所以在您不明白某个寄存器干什么用的时候，可以到《STM32 参考手册》里面查找一下，您就可以迅速查到这个寄存器的作用以及每个位所代表的意思。

MY_NVIC_SetVectorTable 函数的代码如下：

```
//设置向量表偏移地址
//NVIC_VectTab:基址
//Offset:偏移量
void MY_NVIC_SetVectorTable(u32 NVIC_VectTab，  u32 Offset)
{
    //检查参数合法性
    assert_param(IS_NVIC_VECTTAB(NVIC_VectTab));
    assert_param(IS_NVIC_OFFSET(Offset));
    SCB->VTOR = NVIC_VectTab|(Offset & (u32)0x1FFFFF80);//设置 NVIC 的向量表偏移寄存器
    //用于标识向量表是在 CODE 区还是在 RAM 区
}
```

该函数是用来配置中断向量表基址和偏移量，决定是在那个区域。当在 RAM 中调试代码的时候，需要把中断向量表放到 RAM 里面这就需要通过这个函数来配置。关于向量表的详细介绍请参考<<CM3 权威指南>>第七章，第 113 页的向量表一章。

3）Sys_Soft_Reset 函数

该函数用来实现 STM32 的软复位。

```
//系统软复位
void Sys_Soft_Reset(void)
{
    SCB->AIRCR =0X05FA0000|(u32)0x04;
}
```

SCB 为 MDK 定义的一个寄存器组，里面包含了很多与系统相关的控制器，具体的定义如下所示：

```
typedef struct
{
  vuc32 CPUID;   //CM3 内核版本号寄存器
  vu32 ICSR;     //中断控制及状态控制寄存器
  vu32 VTOR;     //向量表便宜量寄存器
  vu32 AIRCR;    //应用程序中断及复位控制寄存器
  vu32 SCR;      //系统控制寄存器
  vu32 CCR;      //配置与控制寄存器
  vu32 SHPR[3];  //系统异常优先级寄存器组
  vu32 SHCSR;    //系统 Handler 控制及状态寄存器
  vu32 CFSR;     //MFSR+BFSR+UFSR
```

```
    vu32 HFSR;      //硬件 fault 状态寄存器
    vu32 DFSR;      //调试 fault 状态寄存器
    vu32 MMFAR;     //存储管理地址寄存器
    vu32 BFAR;      //硬件 fault 地址寄存器
    vu32 AFSR;      //辅助 fault 地址寄存器
} SCB_TypeDef;
```

在 Sys_Soft_Reset 函数里面，我们只是对 SCB-> AIRCR 进行了一次操作，即实现了 STM32 的软复位。AIRCR 寄存器的个位定义如下图所示：

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时把 0x05FA 写入此段，否则写操作被忽略。若读取此半字，则 0xFA05
15	ENDIANESS	R	-	指示端设置。1＝大端(BE8)，0＝小端。此值是在复位时确定的，不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核（调试逻辑除外），但是此复位不影响芯片上在内核以外的电路

图 2.7.2.1 AIRCR 寄存器各位定义

从上面的位定义可以看出，要实现 STM32 的软复位，只要置位 BIT2，这样就可以请求一次软复位。这里要注意 bit31~16 的访问钥匙，要将访问钥匙 0X05FA0000 与我们要进行的操作相或，然后写入 AIRCR，这样才被 CM3 接受。

4）Sys_SleepDeep 函数

STM32 提供了 3 种低功耗模式，以达到不同层次的降低功耗的目的，这三种模式如下：

睡眠模式（CM3 内核停止工作，外设仍在运行）；

停止模式（所有的时钟都停止）；

待机模式；

其中睡眠模式又分为有深度睡眠和睡眠之分。Sys_SleepDeep 函数用来使 STM32 进入待机模式，在该模式下，STM32 所消耗的功耗最低。下面是一个 STM32 的低功耗一览表：

模式	进入操作	唤醒	对1.8V区域时钟的影响	对VDD区域时钟的影响	电压调节器
睡眠(SLEEP-NOW或SLEEP-ON-EXIT)	WFI	任一中断	CPU时钟关，对其他时钟和ADC时钟无影响	无	开
	WFE	唤醒事件			
停机	PDDS和LPDS位+SLEEPDEEP位+WFI或WFE	任一外部中断(在外部中断寄存器中设置)	所有使用1.8V的区域的时钟都已关闭，HSI和HSE的振荡器关闭		在低功耗模式下可进行开/关设置(依据电源控制寄存器(PWR_CR)的设定)
待机	PDDS位+SLEEPDEEP位+WFI或WFE	WKUP引脚的上升沿、RTC警告事件、NRST引脚上的外部复位、IWDG复位			关

图 2.7.2.2　STM32 低功耗模式一览表

下表展示了如何进入和退出待机模式，关于待机模式的更详细介绍请参考《STM32 参考手册》第 4.3 节（40 页）。

待机模式	说明
进入	在以下条件下执行WFI或WFE指令： - 设置Cortex™-M3系统控制寄存器中的SLEEPDEEP位 - 设置电源控制寄存器(PWR_CR)中的PDDS位 - 清除电源控制/状态寄存器(PWR_CSR)中的WUF位被
退出	WKUP引脚的上升沿、RTC闹钟、NRST引脚上外部复位、IWDG复位。
唤醒延时	复位阶段时电压调节器的启动。

图 2.7.2.3　待机模式进入及退出方法

根据上面的了解，我们就可以写出进入待机模式的代码，Sys_Standby 的具体实现代码如下：

```
//进入待机模式
void Sys_Standby(void)
{
    SCB->SCR|=1<<2;//使能 SLEEPDEEP 位  (SYS->CTRL)
    RCC->APB1ENR|=1<<28;        //使能电源时钟
    PWR->CSR|=1<<8;             //设置 WKUP 用于唤醒
    PWR->CR|=1<<2;              //清除 Wake-up  标志
    PWR->CR|=1<<1;              //PDDS 置位
    WFI_SET();                  //执行 WFI 指令
}
```

这里用到了一个 WFI_SET();函数，该函数其实是在 C 语言里面嵌入一条汇编指令，因为 CM3 内核的 STM32 支持的 THUMB 指令，并不能内嵌汇编，所以需要通过这个方法来实现汇编代码的嵌入。该函数的代码如下：

```
//THUMB 指令不支持汇编内联
//采用如下方法实现执行汇编指令 WFI
__asm void WFI_SET(void)
{
    WFI;
```

```
}
```

在执行完 WFI 指令之后，STM32 就进入待机模式了，系统将停止工作，此时 JTAG 会失效，这点请大家在使用的时候要注意。

5）中断管理函数

CM3 内核支持 256 个中断，其中包含了 16 个内核中断和 240 个外部中断，并且具有 256 级的可编程中断设置。但 STM32 并没有使用 CM3 内核的全部东西，而是只用了它的一部分。STM32 有 76 个中断，包括 16 个内核中断和 60 个可屏蔽中断，具有 16 级可编程的中断优先级。而我们常用的就是这 60 个可屏蔽中断，所以我们就只针对这 60 个可屏蔽中断进行介绍。

在 MDK 内，与 NVIC 相关的寄存器，MDK 为其定义了如下的结构体：

```
typedef struct
{
  vu32 ISER[2];
  u32  RESERVED0[30];
  vu32 ICER[2];
  u32  RSERVED1[30];
  vu32 ISPR[2];
  u32  RESERVED2[30];
  vu32 ICPR[2];
  u32  RESERVED3[30];
  vu32 IABR[2];
  u32  RESERVED4[62];
  vu32 IPR[15];
} NVIC_TypeDef;
```

STM32 的中断在这些寄存器的控制下有序的执行的。了解这些中断寄存器，你才能方便的使用 STM32 的中断。下面重点介绍这几个寄存器：

ISER[2]：ISER 全称是：Interrupt Set-Enable Registers，这是一个中断使能寄存器组。上面说了 STM32 的可屏蔽中断只有 60 个，这里用了 2 个 32 位的寄存器，总共可以表示 64 个中断。而 STM32 只用了其中的前 60 位。ISER[0]的 bit0~bit31 分别对应中断 0~31。ISER[1]的 bit0~27 对应中断 32~59；这样总共 60 个中断就分别对应上了。你要使能某个中断，必须设置相应的 ISER 位为 1，使该中断被使能(这里仅仅是使能，还要配合中断分组、屏蔽、IO 口映射等设置才算是一个完整的中断设置)。具体每一位对应哪个中断，请参考 stm32f10x_nvic..h 里面的第 36 行处。

ICER[2]：全称是：Interrupt Clear-Enable Registers，是一个中断除能寄存器组。该寄存器组与 ISER 的作用恰好相反，是用来清除某个中断的使能的。其对应位的功能，也和 ICER 一样。这里要专门设置一个 ICER 来清除中断位，而不是向 ISER 写 0 来清除，是因为 NVIC 的这些寄存器都是写 1 有效的，写 0 是无效的。具体为什么这么设计，请看《CM3 权威指南》第 125 页，NVIC 概览一章。

ISPR[2]：全称是：Interrupt Set-Pending Registers，是一个中断挂起控制寄存器组。每个位对应的中断和 ISER 是一样的。通过置 1，可以将正在进行的中断挂起，而执行同级或更高级别的中断。写 0 是无效的。

ICPR[2]：全称是：Interrupt Clear-Pending Registers，是一个中断解挂控制寄存器组。其作用与 ISPR 相反，对应位也和 ISER 是一样的。通过设置 1，可以将挂起的中断接挂。写 0 无效。

IABR[2]：全称是：Active Bit Registers，是一个中断激活标志位寄存器组。对应位所代表

的中断和 ISER 一样，如果为 1，则表示该位所对应的中断正在被执行。这是一个只读寄存器，通过它可以知道当前在执行的中断是哪一个。在中断执行完了由硬件自动清零。

IPR[15]：全称是：Interrupt Priority Registers，是一个中断优先级控制的寄存器组。这个寄存器组相当重要！STM32 的中断分组与这个寄存器组密切相关。IPR 寄存器组由 15 个 32bit 的寄存器组成，每个可屏蔽中断占用 8bit，这样总共可以表示 15*4=60 个可屏蔽中断。刚好和 STM32 的可屏蔽中断数相等。IPR[0]的[31~24]，[23~16]，[15~8]，[7~0]分别对应中中断 3~0，依次类推，总共对应 60 个外部中断。而每个可屏蔽中断占用的 8bit 并没有全部使用，而是 只用了高 4 位。这 4 位，又分为抢占优先级和子优先级。抢占优先级在前，子优先级在后。而这两个优先级各占几个位又要根据 SCB->AIRCR 中中断分组的设置来决定。

这里简单介绍一下 STM32 的中断分组：STM32 将中断分为 5 个组，组 0~4。该分组的设置是由 SCB->AIRCR 寄存器的 bit10~8 来定义的。具体的分配关系如下表所示：

组	AIRCR[10:8]	bit[7:4]分配情况	分配结果
0	111	0:4	0 位抢占优先级，4 位响应优先级
1	110	1:3	1 位抢占优先级，3 位响应优先级
2	101	2:2	2 位抢占优先级，2 位响应优先级
3	100	3:1	3 位抢占优先级，1 位响应优先级
4	11	4:0	4 位抢占优先级，0 位响应优先级

表 2.7.2.1 AIRCR 中断分组设置表

通过这个表，我们就可以清楚的看到组 0~4 对应的配置关系，例如组设置为 3，那么此时所有的 60 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

结合实例说明一下：假定设置中断优先级组为 2，然后设置中断 3(RTC 中断)的抢占优先级为 3，响应优先级为 1。中断 6（外部中断 0）的抢占优先级为 4，响应优先级为 0。中断 7（外部中断 1）的抢占优先级为 3，响应优先级为 0。那么这 3 个中断的优先级顺序为：中断 7>中断 3>中断 6。

这里需要注意 2 点：

如果两个中断的响应优先级和响应优先级都是一样的话，则看哪个中断先发生就先执行。

高优先级的抢占优先级是可以打断正在进行的低抢占优先级中断的。而抢占优先级相同的中断，高优先级的响应优先级不可以打断低响应优先级的中断。上面例子中的中断 3 和中断 7 都可以打断中断 6 的中断。而中断 7 和中断 3 却不可以相互打断！

通过以上介绍，我们熟悉了 STM32 中断设置的大致过程。接下来我们介绍如何使用函数实现以上中断设置，使得我们以后的中断设置简单化。

第一个介绍的是 NVIC 的分组函数 MY_NVIC_PriorityGroupConfig，该函数的参数 NVIC_Group 为要设置的分组号，可选范围为 0~4，总共 5 组。如果参数非法，将可能导致不可预料的结果。MY_NVIC_PriorityGroupConfig 函数代码如下：

```
//设置 NVIC 分组
//NVIC_Group:NVIC 分组 0~4 总共 5 组
void MY_NVIC_PriorityGroupConfig(u8 NVIC_Group)
{
    u32 temp，temp1;
    temp1=(~NVIC_Group)&0x07;//取后三位
```

```
    temp1<<=8;
    temp=SCB->AIRCR;  //读取先前的设置
    temp&=0X0000F8FF; //清空先前分组
    temp|=0X05FA0000; //写入钥匙
    temp|=temp1;
    SCB->AIRCR=temp;  //设置分组
}
```

通过前面的介绍，我们知道 STM32 的 5 个分组是通过设置 SCB->AIRCR 的 BIT[10:8]来实现的，而通过 2.7.2.1 的介绍我们知道 SCB->AIRCR 的修改需要通过在高 16 位写入 0X05FA 这个密钥才能修改的，故在设置 AIRCR 之前，应该把密钥加入到要写入的内容的高 16 位，以保证能正常的写入 AIRCR。在修改 AIRCR 的时候，我们一般采用读->改->写的步骤，来实现不改变 AIRCR 原来的其他设置。以上就是 MY_NVIC_PriorityGroupConfig 函数设置中断优先级分组的思路。

第二个函数是 NVIC 设置函数 MY_NVIC_Init，该函数有 4 个参数，分别为：NVIC_PreemptionPriority、NVIC_SubPriority、NVIC_Channel、NVIC_Group。第一个参数 NVIC_PreemptionPriority 为中断抢占优先级数值，第二个参数 NVIC_SubPriority 为中断子优先级数值，前两个参数的值必须在规定范围内，否则也可能产生意想不到的错误。第三个参数 NVIC_Channel 为中断的编号（范围为 0~59），最后一个参数 NVIC_Group 为中断分组设置（范围为 0~4）。该函数代码如下：

```
//设置 NVIC
//NVIC_PreemptionPriority:抢占优先级
//NVIC_SubPriority        :响应优先级
//NVIC_Channel            :中断编号
//NVIC_Group              :中断分组 0~4
//注意优先级不能超过设定的组的范围!否则会有意想不到的错误
//组划分:
//组 0:0 位抢占优先级，4 位响应优先级
//组 1:1 位抢占优先级，3 位响应优先级
//组 2:2 位抢占优先级，2 位响应优先级
//组 3:3 位抢占优先级，1 位响应优先级
//组 4:4 位抢占优先级，0 位响应优先级
//NVIC_SubPriority 和 NVIC_PreemptionPriority 的原则是，数值越小，越优先
void MY_NVIC_Init(u8 NVIC_PreemptionPriority，u8 NVIC_SubPriority，u8 NVIC_Channel，u8 NVIC_Group)
{
    u32 temp;
    u8 IPRADDR=NVIC_Channel/4;  //每组只能存 4 个，得到组地址
    u8 IPROFFSET=NVIC_Channel%4;//在组内的偏移
    IPROFFSET=IPROFFSET*8+4;    //得到偏移的确切位置
    MY_NVIC_PriorityGroupConfig(NVIC_Group);//设置分组
    temp=NVIC_PreemptionPriority<<(4-NVIC_Group);
    temp|=NVIC_SubPriority&(0x0f>>NVIC_Group);
    temp&=0xf;//取低四位
```

```
    if(NVIC_Channel<32)NVIC->ISER[0]|=1<<NVIC_Channel;//使能中断位(要清除的话，相反操作就 OK)
    else NVIC->ISER[1]|=1<<(NVIC_Channel-32);
    NVIC->IPR[IPRADDR]|=temp<<IPROFFSET;//设置响应优先级和抢断优先级

}
```

通过前面的介绍，我们知道每个可屏蔽中断的优先级的设置是在 IPR 寄存器组里面的，每个中断占 8 位，但只用了其中的 4 个位，以上代码就是根据中断分组情况，来设置每个中断对应的高 4 位的数值的。当然在该函数里面还引用了 MY_NVIC_PriorityGroupConfig 这个函数来设置分组。其实这个分组函数在每个系统里面只要设置一次就够了，设置多次，则是以最后的那一次为准。但是只要多次设置的组号都是一样，就没事。否则前面设置的中断会因为后面组的变化优先级会发生改变，这点在使用的时候要特别注意！一个系统代码里面，所有的中断分组都要统一!!，以上代码对要配置的中断号默认是开启中断的。也就是 ISER 中的值设置为 1 了。

通过以上两个函数就实现了对 NVIC 的管理和配置。但是外部中断的设置，还需要配置相关寄存器才可以。下面就介绍外部中断的配置和使用。

STM32 的 EXTI 控制器支持 19 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。STM32 的 19 个外部中断为：

线 0~15：对应外部 IO 口的输入中断。

线 16：连接到 PVD 输出。

线 17：连接到 RTC 闹钟事件。

线 18：连接到 USB 唤醒事件。

对于外部中断 EXTI 控制 MDK 定义了如下结构体：

```
typedef struct
{
  vu32 IMR;
  vu32 EMR;
  vu32 RTSR;
  vu32 FTSR;
  vu32 SWIER;
  vu32 PR;
} EXTI_TypeDef;
```

通过这些寄存器的设置，就可以对外部中断进行详细设置了。下面我们就重点介绍这些寄存器的作用。

IMR：中断屏蔽寄存器。这是一个 32 寄存器。但是只有前 19 位有效。当位 x 设置为 1 时，则开启这个线上的中断，否则关闭该线上的中断。

EMR：事件屏蔽寄存器，同 IMR，只是该寄存器是针对事件的屏蔽和开启。

RTSR：上升沿触发选择寄存器。该寄存器同 IMR，也是一个 32 为的寄存器，只有前 19 位有效。位 x 对应线 x 上的上升沿触发，如果设置为 1，则是允许上升沿触发中断/事件。否则，不允许。

FTSR：下降沿触发选择寄存器。同 PTSR，不过这个寄存器是设置下降沿的。下降沿和上升沿可以被同时设置，这样就变成了任意电平触发了。

SWIER：软件中断事件寄存器。通过向该寄存器的位 x 写入 1，在未设置 IMR 和 EMR 的时候，将设置 PR 中相应位挂起。如果设置了 IMR 和 EMR 时将产生一次中断。被设置的 SWIER 位，将会在 PR 中的对应位清除后清除。

PR：挂起寄存器。当外部中断线上发生了选择的边沿事件，该寄存器的对应位会被置为 1。0，表示对应线上没有发生触发请求。通过向该寄存器的对应位写入 1 可以清除该位。在中断服务函数里面经常会要向该寄存器的对应位写 1 来清除中断请求。

以上就是与中断相关寄存器的介绍，更详细的介绍，请参考《STM32 参考手册》第 95 页，8.3 节 EXTI 寄存器描述这一章。

通过以上配置就可以正常设置外部中断了，但是外部 IO 口的中断，还需要一个寄存器配置，也就是 IO 复用里的外部中断配置寄存器 EXTICR。这是因为 STM32 任何一个 IO 口都可以配置成中断输入口，但是 IO 口的数目远大于中断线数（16 个）。于是 STM32 就这样设计，GPIOA~GPIOG 的[15:0]分别对应中断线 15~0。这样每个中断线对应了最多 7 个 IO 口，以线 0 为例：它对应了 GPIOA.0、PIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0、GPIOG.0。而中断线每次只能连接到 1 个 IO 口上，这样就需要 EXTICR 来决定对应的中断线配置到哪个 GPIO 上了。

EXTICR 在 AFIO 的结构体中定义，如下：

```
typedef struct
{
  vu32 EVCR;
  vu32 MAPR;
  vu32 EXTICR[4];
} AFIO_TypeDef;
```

EXTICR 寄存器组，总共有 4 个，因为编译器的寄存器组都是从 0 开始编号的，所以 EXTICR[0]~ EXTICR[3]，对应《STM32 参考手册》里的 EXTICR1~ EXTICR 4。每个 EXTICR 只用了其低 16 位。EXTICR[0]的分配如下：

位31:16	保留。
位15:0	**EXTIx[3:0]**：EXTIx配置(x = 0 … 3) 这些位可由软件读写，用于选择EXTIx外部中断的输入源。 0000：PA[x]管脚　0100：PE[x]管脚 0001：PB[x]管脚　0101：PF[x]管脚 0010：PC[x]管脚　0110：PG[x]管脚 0011：PD[x]管脚

图 2.7.2.4　寄存器 EXTICR[0]各位定义

比如如我要设置 GPIOB.1 映射到 1，则只要设置 EXTICR[0]的 bit[7:4]为 0001 即可。默认都是 0000 即映射到 GPIOA。从图 5.7 中可以看出，EXTICR[0]只管了 GPIO 的 0~3 端口，相应的其他端口由 EXTICR[1~3]管理。具体请参考《STM32 参考手册》第 126~128 页。

通过对上面的分析我们就可以完成对外部中断的配置了。该函数为 Ex_NVIC_Config，该

函数有 3 个参数：GPIOx 为 GPIOA~G（0~6），在 sys.h 里面有定义。代表要配置的 IO 口。BITx 则代表这个 IO 口的第几位。TRIM 为触发方式，低 2 位有效（0x01 代表下降触发；0x02 代表上升沿触发；0x03 代表任意电平触发）。其代码如下：

```
//外部中断配置函数
//只针对 GPIOA~G;不包括 PVD，RTC 和 USB 唤醒这三个
//参数:GPIOx:0~6，代表 GPIOA~G;BITx:需要使能的位;TRIM:触发模式，1，下升沿;2，上降沿;3，任意电平触发
//该函数一次只能配置 1 个 IO 口，多个 IO 口，需多次调用
//该函数会自动开启对应中断，以及屏蔽线
void Ex_NVIC_Config(u8 GPIOx，u8 BITx，u8 TRIM)
{
    u8 EXTADDR;
    u8 EXTOFFSET;
    EXTADDR=BITx/4;//得到中断寄存器组的编号
    EXTOFFSET=(BITx%4)*4;

    RCC->APB2ENR|=0x01;//使能 io 复用时钟
    AFIO->EXTICR[EXTADDR]|=GPIOx<<EXTOFFSET;//EXTI.BITx 映射到 GPIOx.BITx

    //自动设置
    EXTI->IMR|=1<<BITx;//  开启 line BITx 上的中断
    EXTI->EMR|=1<<BITx;//不屏蔽 line BITx 上的事件
    if(TRIM&0x01)EXTI->FTSR|=1<<BITx;//line BITx 上事件下降沿触发
    if(TRIM&0x02)EXTI->RTSR|=1<<BITx;//line BITx 上事件上升降沿触发
}
```

Ex_NVIC_Config 完全是按照我们之前的分析来编写的，首先根据 GPIOx 的位得到中断寄存器组的编号，即 EXTICR 的编号，在 EXTICR 里面配置中断线应该配置到 GPIOx 的哪个位。然后使能该位的中断及事件，最后配置触发方式。这样就完成了外部中断的的配置了。从代码中可以看到该函数默认是开启中断和事件的。其次还要注意的一点就是该函数一次只能配置一个 IO 口，如果你有多个 IO 口需要配置，则多次调用这个函数就可以了。

至此，我们对 STM32 的中断管理就介绍结束了。当然还有中断响应函数，我们这里没有介绍，这个在后面的实例中会向各位讲述的。

6）JTAG_Set 函数

STM32 支持 JTAG 和 SWD 两种仿真接口，他们和普通的 IO 口共用，当需要使用普通 IO 口的时候，则必须先禁止 JTAG/SWD。STM32 在默认状态下是开启 JTAG 的，所以那些和 JTAG 共用的 IO 口，在默认状态下是不能做普通 IO 口使用的。我们可以通过 AFIO_MAPR 寄存器的 24~26 位来修改 STM32 的 JTAG 配置，从而切换为普通 IO 口或者其他状态。AFIO_MAPR 寄存器的第 24~26 位描述如下图所示：

位26:24	**SWJ_CFG[2:0]**：串行线JTAG配置 这些位可由软件读写，用于配置SWJ和跟踪复用功能的I/O口。SWJ(串行线JTAG)支持JTAG或SWD访问Cortex的调试端口。系统复位后的默认状态是启用SWJ但没有跟踪功能，这种状态下可以通过JTMS/JTCK脚上的特定信号选择JTAG或SW(串行线)模式。 000：完全SWJ(JTAG-DP + SW-DP)：复位状态； 001：完全SWJ(JTAG-DP + SW-DP)但没有JNTRST； 010：关闭JTAG-DP，启用SW-DP； 100：关闭JTAG-DP，关闭SW-DP； 其它组合：禁用。

图 2.7.2.5　AFIO_MAPR 寄存器第 24~26 各位描述

有了上面这个图，我们就可以编写我们的 JTAG 模式配置函数了。JTAG 模式配置函数代码如下：

```
//JTAG 模式设置,用于设置 JTAG 的模式
//mode:jtag,swd 模式设置;00,全使能;01,使能 SWD;10,全关闭;
void JTAG_Set(u8 mode)
{
    u32 temp;
    temp=mode;
    temp<<=25;
    RCC->APB2ENR|=1<<0;         //开启辅助时钟
    AFIO->MAPR&=0XF8FFFFFF; //清除 MAPR 的[26:24]
    AFIO->MAPR|=temp;           //设置 jtag 模式
}
```

通过该函数，我们就可以方便的设置 JTAG 的模式。这里顺便提一下，JTAG 和普通 IO 口，可以在仿真的时候，分时复用，不过在您禁止了 JTAG 的地方（先禁止，后开启），您必须一步跳过去，也就是这部分代码，您不要执行到里面去观察，直接跳过。这样，您的 JTAG 可以在不断线的情况下，继续调试下面的代码。

2.7.3 usart 文件夹介绍

usart 文件夹内包含了 usart.c 和 usart.h 两个文件。这两个文件用于串口的初始化和中断接收。这里只是针对串口 1，比如你要用串口 2 或者其他的串口，只要对代码稍作修改就可以了。usart.c 里面包含了 2 个函数一个是 void USART1_IRQHandler(void);另外一个是 void uart_init(u32 pclk2，u32 bound);里面还有一段对串口 printf 的支持代码，如果去掉，则会导致 printf 无法使用，虽然软件编译不会报错，但是硬件上 STM32 是无法启动的。这段代码不要去修该。

void USART1_IRQHandler(void)函数是一个串口 1 中断响应函数，当串口 1 发生了相应的中断后，就会跳到该函数执行。这里我们设计了一个小小的接收协议：通过这个函数，配合一个数组 USART_RX_BUF[64]，一个接收状态寄存器 USART_RX_STA 实现对串口数据的接收管理。USART_RX_BUF 的最大值为 64，也就是一次接收的数据最大不能超过 64 个字节。USART_RX_STA 是一个接收状态寄存器其各的定义如下表：

USART_RX_STA							
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

接收完成标志	接收到 0X0D 标志	接收到的有效数据个数

表 2.7.2.2 接收状态寄存器位定义表

设计思路如下：

当接收到从电脑发过来的数据，把接收到的数据保存在 USART_RX_BUF 中，同时在接收状态寄存器（USART_RX_STA）中计数接收到的有效数据个数，当收到回车（0X0D，0X0A）的第一个字节 0X0D 时，计数器将不再增加，等待 0X0A 的到来，而如果 0X0A 没有来到，则认为这次接收失败，重新开始下一次接收。如果顺利接收到 0X0A，则标记 USART_RX_STA 的第七位，这样完成一次接收，并等待该位被其他程序清除，从而开始下一次的接收，而如果迟迟没有收到 0X0D，那么在接收数据超过 64 个了，则会丢弃前面的数据，重新接收。函数代码如下：

```
/ #ifdef EN_USART1_RX     //如果使能了接收
//串口 1 中断服务程序
//注意，读取 USARTx->SR 能避免莫名其妙的错误
u8 USART_RX_BUF[64];        //接收缓冲，最大 64 个字节.
//接收状态
//bit7，接收完成标志
//bit6，接收到 0x0d
//bit5~0，接收到的有效字节数目
u8 USART_RX_STA=0;          //接收状态标记

void USART1_IRQHandler(void)
{
    u8 res;
    if(USART1->SR&(1<<5))//接收到数据
    {
        res=USART1->DR;
        if((USART_RX_STA&0x80)==0)//接收未完成
        {
            if(USART_RX_STA&0x40)//接收到了 0x0d
            {
                if(res!=0x0a)USART_RX_STA=0;//接收错误，重新开始
                else USART_RX_STA|=0x80;     //接收完成了
            }else //还没收到 0X0D
            {
                if(res==0x0d)USART_RX_STA|=0x40;
                else
                {
                    USART_RX_BUF[USART_RX_STA&0X3F]=res;
                    USART_RX_STA++;
                    if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错误，重新
开始接收
```

```
                }
            }
        }
    }
}
#endif
```

从上面的代码我们可以看使用了宏定义#ifdef，当需要使用串口接收的时候，我们只要在 usart.h 里面定义 EN_USART1_RX 就可以了。不使用的时候，注释掉就可，这样可以省出部分 sram 和 flash。

void uart_init(u32 pclk2，u32 bound)函数是串口 1 初始化函数。该函数有 2 个参数，第一个为 pclk2，是系统的时钟频率。第二个参数为需要设置的波特率，例如 9600，115200 等。而这个函数的重点就是在波特率的设置，由于 STM32 采用了分数波特率，所以 STM32 的串口波特率设置范围很宽，而且误差很小。

STM32 的每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以达到配置不同波特率的目的。其各位描述如下图所示：

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
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。

图 2.7.3.1 寄存器 USART_BRR 各位描述

前面提到 STM32 的分数波特率概念，其实就是在这个寄存器（USART_BRR）里面体现的。USART_BRR 的最低 4 位（位[3:0]）用来存放小数部分 DIV_Fraction，紧接着的 12 位（位[15：4]）用来存放整数部分 DIV_Mantissa，最高 16 位未使用。

这里，我们简单介绍一下波特率的计算，STM32 的串口波特率计算公式如下：

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

上式中，f_{PCLKx} 是给串口的时钟（PCLK1 用于 USART2、3、4、5，PCLK2 用于 USART1）；USARTDIV 是一个无符号定点数。我们只要得到 USARTDIV 的值，就可以得到串口波特率寄存器 USART1->BRR 的值，反过来，我们得到 USART1->BRR 的值，也可以推导出 USARTDIV 的值。但我们更关心的是如何从 USARTDIV 的值得到 USART_BRR 的值，因为一般我们知道的是波特率，和 PCLKx 的时钟，要求的就是 USART_BRR 的值。

下面我们来介绍如何通过 USARTDIV 得到串口 USART_BRR 寄存器的值。假设我们的串口 1 要设置为 9600 的波特率，而 PCLK2 的时钟为 72M。这样，我们根据上面的公式有：

USARTDIV=72000000/（9600*16）=468.75

那么得到：

DIV_Fraction=16*0.75=12=0X0C;

DIV_Mantissa= 468=0X1D4;

这样，我们就得到了 USART1->BRR 的值为 0X1D4C。只要设置串口 1 的 BRR 寄存器值为 0X1D4C 就可以得到 9600 的波特率。

当然，并不是任何条件下都可以随便设置串口波特率的，在某些波特率和 PCLK2 频率下，还是会存在误差的，具体可以参考《STM32 参考手册》的第 525 页的表 176。

接下来，我们就可以初始化串口了，需要注意的是这里初始化串口是按 8 位数据格式，1 位停止位，无奇偶校验位的。具体代码如下：

```
//初始化 IO 串口 1
//pclk2：PCLK2 时钟频率(Mhz)
//bound：波特率
void uart_init(u32 pclk2，u32 bound)
{
    float temp;
    u16 mantissa;
    u16 fraction;
    temp=(float)(pclk2*1000000)/(bound*16);//得到 USARTDIV
    mantissa=temp;                          //得到整数部分
    fraction=(temp-mantissa)*16; //得到小数部分
    mantissa<<=4;
    mantissa+=fraction;
    RCC->APB2ENR|=1<<2;     //使能 PORTA 口时钟
    RCC->APB2ENR|=1<<14;    //使能串口时钟
    GPIOA->CRH=0X444444B4;//IO 状态设置

    RCC->APB2RSTR|=1<<14;     //复位串口 1
    RCC->APB2RSTR&=~(1<<14);//停止复位
    //波特率设置
    USART1->BRR=mantissa; // 波特率设置
    USART1->CR1|=0X200C;  //1 位停止，无校验位.
#ifdef EN_USART1_RX             //如果使能了接收
    USART1->CR1|=1<<8;      //使能接收中断,PE 中断使能
    USART1->CR1|=1<<5;      //接收缓冲区非空中断使能
    MY_NVIC_Init(3，3，USART1_IRQChannel，2);//组 2，最低优先级
#endif
}
```

上面的代码，就实现了对串口 1 波特率的设置。通过该函数的初始化，我们就可以得到在当前频率（pclk2）下得到自己想要的波特率。

2.8 RVMDK 使用技巧

前面介绍了 RVMDK 的基本使用，接下来简单的介绍一下 RVMDK 的几个使用技巧。

1、文本美化。

2、代码编辑技巧。

3、调试技巧。

接下来我们就这几点技巧进行介绍，看看如何利用这些技巧使你的开发工作事倍功半。

2.8.1 文本美化

首先要介绍的是文本美化。前面我们在介绍 RVMDK 新建工程的时候看到如下界面：

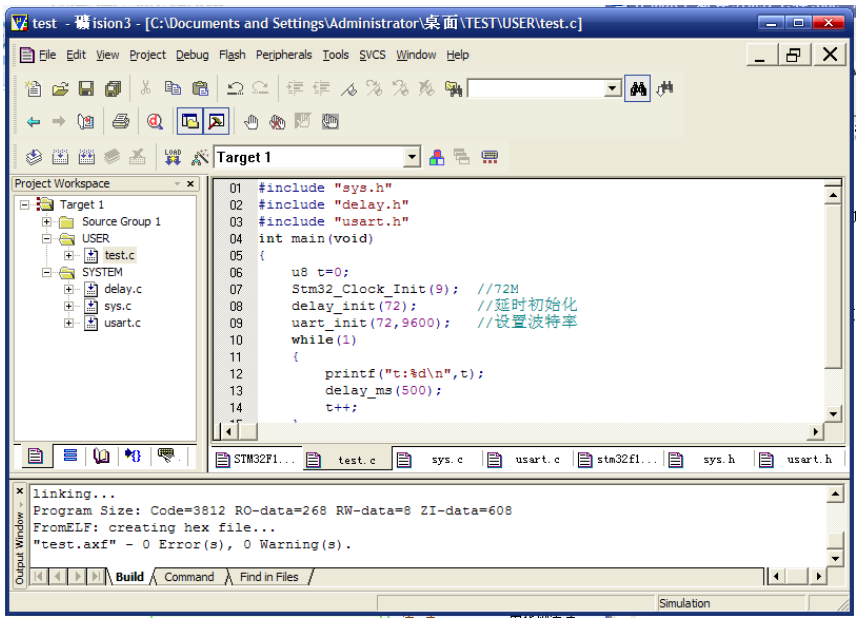


图 2.8.1.1 MDK 新建工程界面

可以看到其中的关键字和注释等字体的颜色不是很漂亮，而 keil 提供了我们自定义字体颜色的功能。我们可以在工具条上点击 (编辑配置对话框)弹出如下界面：

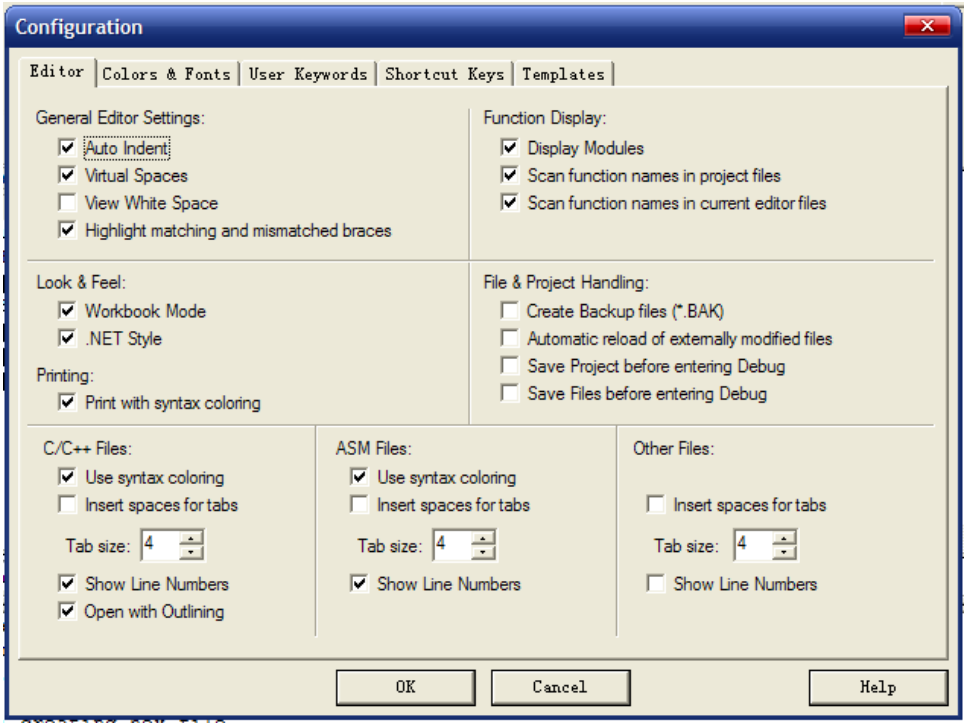


图 2.8.1.2　编辑配置对话框

在该对话框中我们选择 Colors&Fonts 选项卡，在该选项卡内，我们就可以设置自己的代码的子体和颜色了。由于我们使用的是 C 语言，故在 Text File Types 下面选择 ARM:Editor C Files 在右边就可以看到相应的元素了。入下图所示：

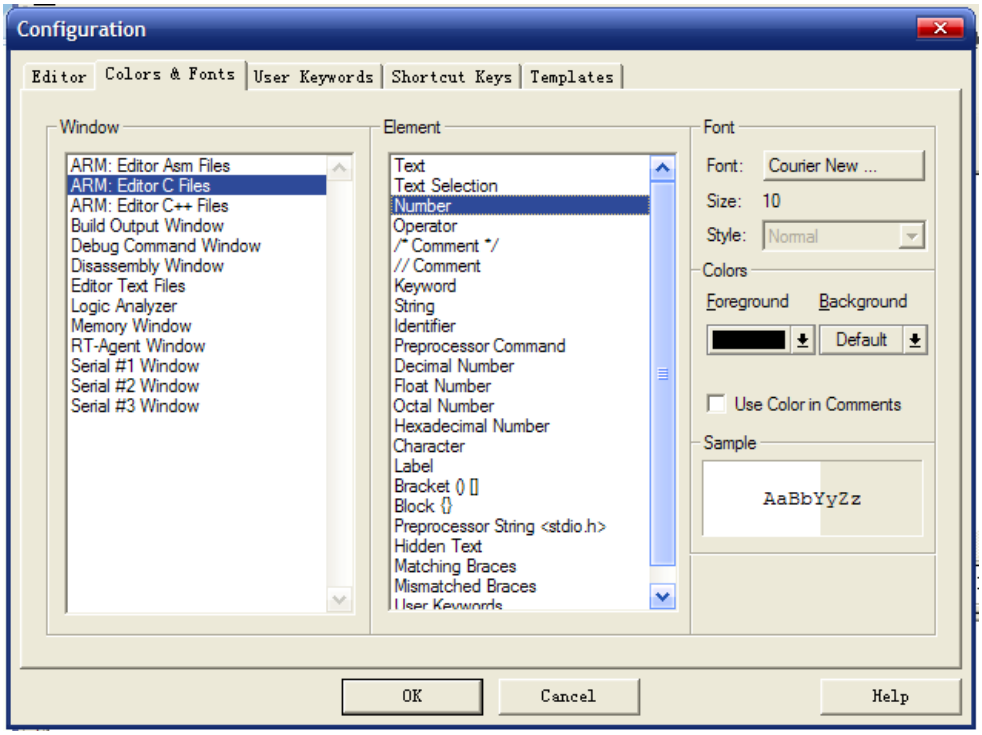


图 2.8.1.3　Colors&Fonts 选项卡

然后点击各个元素修改为你喜欢的颜色，当然也可以在 Font 栏设置你字体的类型，以及字体的大小等。，就可以在主界面看到你所修改后的结果，例如我修改后的代码显示效果如下：

```
01  #include "sys.h"
02  #include "delay.h"
03  #include "usart.h"
04  int main(void)
05  {
06      u8 t=0;
07      Stm32_Clock_Init(9);   //72M
08      delay_init(72);        //延时初始化
09      uart_init(72,9600);    //设置波特率
10      while(1)
11      {
12          printf("t:%d\n",t);
13          delay_ms(500);
14          t++;
15      }
16  }
```

图 2.8.1.4 设置完后显示效果

这就比开始的效果好看多了（个人觉得）。当然你觉得字体小了可以在刚刚的对话框 Font 栏设置大一点，觉得大了也可以设置小一点，总之设置到你认为可以为止。

细心的大家可能会发现，上面的代码里面有一个 u8，还是黑色的，这是一个用户自定义的关键字，为什么不显示蓝色（假定刚刚已经设置了用户自定义关键字颜色为蓝色）呢？这就又要回到我们刚刚的配置对话框了，单这次我们要选择 User Keywords 选项卡，同样选择 ARM:Editor C Files，在右边的 User Keywords 对话框下面输入你自己定义的关键字，入下图所示：

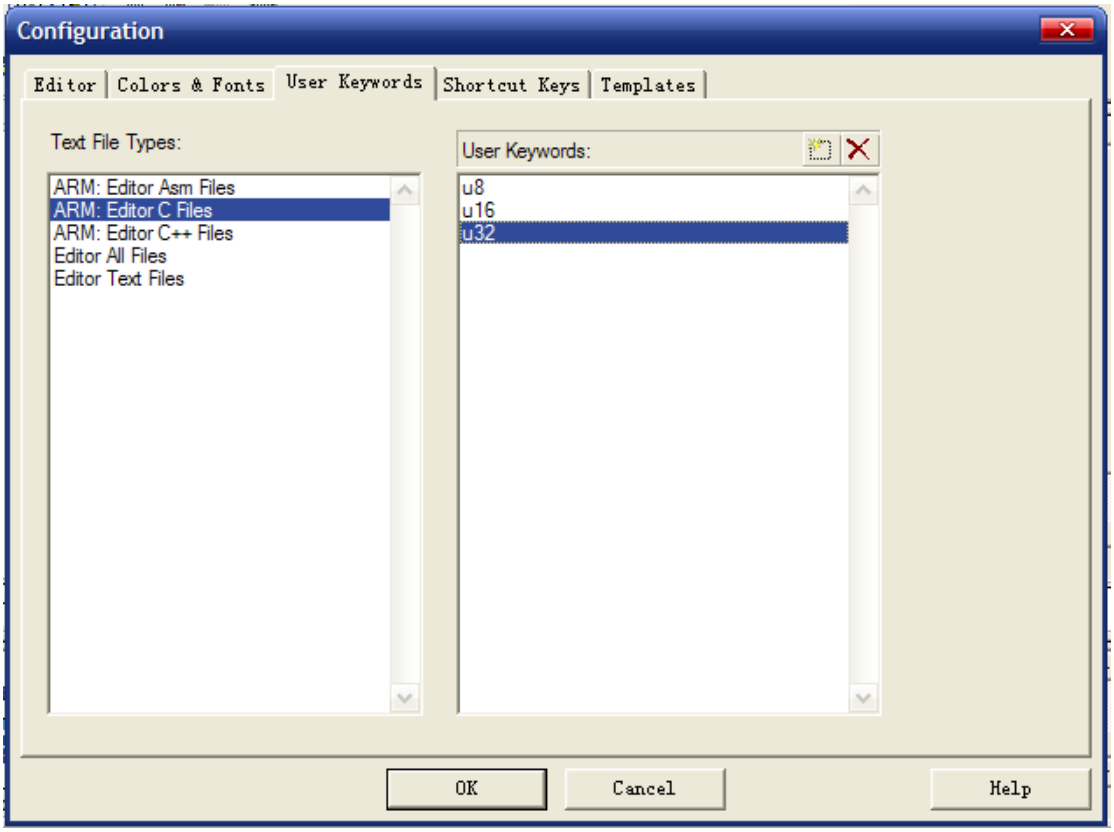


图 2.8.1.5 用户自定义关键字

上图中我定义了 u8、u16、u32 等 3 个关键字，这样在以后的代码编辑里面只要出现这三个

关键字，肯定就会变成蓝色。点击 OK，再回到主界面，可以看到 u8 变成了蓝色了，如下图所示：

```
01 #include "sys.h"
02 #include "delay.h"
03 #include "usart.h"
04 int main(void)
05 {
06     u8 t=0;
07     Stm32_Clock_Init(9);  //72M
08     delay_init(72);       //延时初始化
09     uart_init(72,9600);   //设置波特率
10     while(1)
11     {
12         printf("t:%d\n",t);
13         delay_ms(500);
14         t++;
15     }
16 }
```

图 2.8.1.6 设置完后显示效果

其实这个编辑配置对话框里面，还可以对其他很多功能进行设置，比如按 TAB 键右移多少位，快捷键修改等，有兴趣的大家可以自己摸索一下。文本美化的技巧就为大家介绍到这里，接下来我们为大家介绍 RVMDK 的代码编辑技巧。

2.8.2 代码编辑技巧

这里给大家介绍几个我常用的技巧，这些小技巧能给我们的代码编辑带来很大的方便，相信对你的代码编写一定会有所帮助。

1）TAB 键妙用

首先要介绍的就是 TAB 键的使用，这个键在很多编译器里面都是用来空位的，每按一下移空几个位。如果你是经常编写程序的对这个键一定再熟悉不过了。但是 MDK 的 TAB 键和一般编译器的 TAB 键有不同的地方，和 C++的 TAB 键差不多。MDK 的 TAB 键支持块操作。也就是可以一片代码整体右移固定的几个位，也可以通过 SHIFT+TAB 键整体左移固定的几个位。

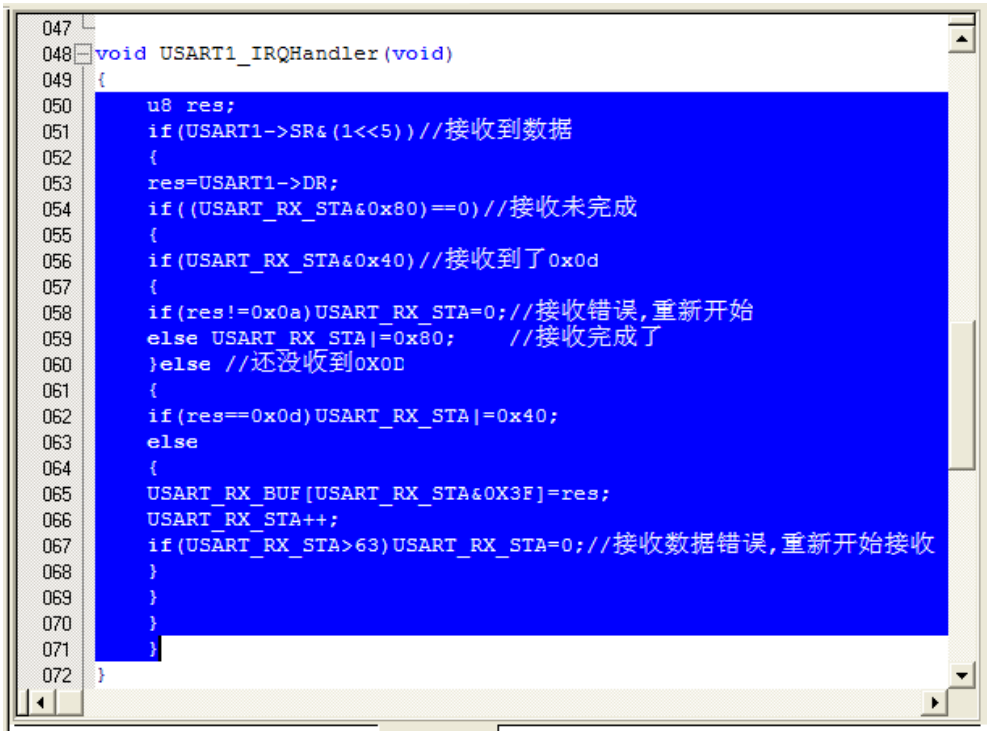
假设我们前面的串口 1 中断响应函数如下图所示：

```
047
048 void USART1_IRQHandler(void)
049 {
050 u8 res;
051 if(USART1->SR&(1<<5))//接收到数据
052 {
053 res=USART1->DR;
054 if((USART_RX_STA&0x80)==0)//接收未完成
055 {
056 if(USART_RX_STA&0x40)//接收到了0x0d
057 {
058 if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
059 else USART_RX_STA|=0x80;    //接收完成了
060 }else //还没收到0X0D
061 {
062 if(res==0x0d)USART_RX_STA|=0x40;
063 else
064 {
065 USART_RX_BUF[USART_RX_STA&0X3F]=res;
066 USART_RX_STA++;
067 if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错误,重新开始接收
068 }
069 }
070 }
071 }
072 }
```

图 2.8.2.1 头大的代码

这样代码你肯定不会喜欢，这还是短短的 20 来行代码，如果你的代码有几千行，全部是这个样子，你不头大才怪。看到这样的代码我们就可以通过 TAB 键的妙用来快速修改为比较规范的代码格式。

选中一块然后按 TAB 键，你可以看到整块代码都跟着右移了一定距离，如下图所示：



```
047
048 void USART1_IRQHandler(void)
049 {
050     u8 res;
051     if(USART1->SR&(1<<5))//接收到数据
052     {
053     res=USART1->DR;
054     if((USART_RX_STA&0x80)==0)//接收未完成
055     {
056     if(USART_RX_STA&0x40)//接收到了0x0d
057     {
058     if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
059     else USART_RX_STA|=0x80;    //接收完成了
060     }else //还没收到0X0D
061     {
062     if(res==0x0d)USART_RX_STA|=0x40;
063     else
064     {
065     USART_RX_BUF[USART_RX_STA&0X3F]=res;
066     USART_RX_STA++;
067     if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错误,重新开始接收
068     }
069     }
070     }
071     }
072 }
```

图 2.8.2.1 代码整体偏移

接下来我们就是要多选几次，然后多按几次 TAB 键就可以达到迅速使代码规范化的目的，最终效果如下图所示

```
047
048 void USART1_IRQHandler(void)
049 {
050     u8 res;
051     if(USART1->SR&(1<<5))//接收到数据
052     {
053         res=USART1->DR;
054         if((USART_RX_STA&0x80)==0)//接收未完成
055         {
056             if(USART_RX_STA&0x40)//接收到了0x0d
057             {
058                 if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
059                 else USART_RX_STA|=0x80;    //接收完成了
060             }else //还没收到0X0D
061             {
062                 if(res==0x0d)USART_RX_STA|=0x40;
063                 else
064                 {
065                     USART_RX_BUF[USART_RX_STA&0X3F]=res;
066                     USART_RX_STA++;
067                     if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错
068                 }
069             }
070         }
071     }
072 }
```

图 2.8.2.2 修改后的代码

从上面可以看到整个代码一下就变得有条理多了，看起来很舒服。

2）快速定位函数/变量被定义的地方

前面介绍了 TAB 键的功能，接下来我们介绍一下如何快速查看一个函数或者变量所定义的地方。

你在调试代码或编写代码的时候，一定有想看看某个函数是在那个地方定义的，具体里面的内容是怎么样的，也可能想看看某个变量或数组是在哪个地方定义的等。尤其在调试代码或者看别人代码的时候，如果编译器没有快速定位的功能的时候，你只能慢慢的自己找，代码量比较少还好，如果代码量一大，那就郁闷了，有时候要花很久的时间来找这个函数到底在哪里。型号 MDK 提供了这样的快速定位的功能（顺便说一下 CVAVR 的 2.0 以后的版本也有这个功能）。只要你把光标放到这个函数/变量（xxx）的上面（xxx 为你想要查看的函数或变量的名字），然后右键，如下图所示 ：

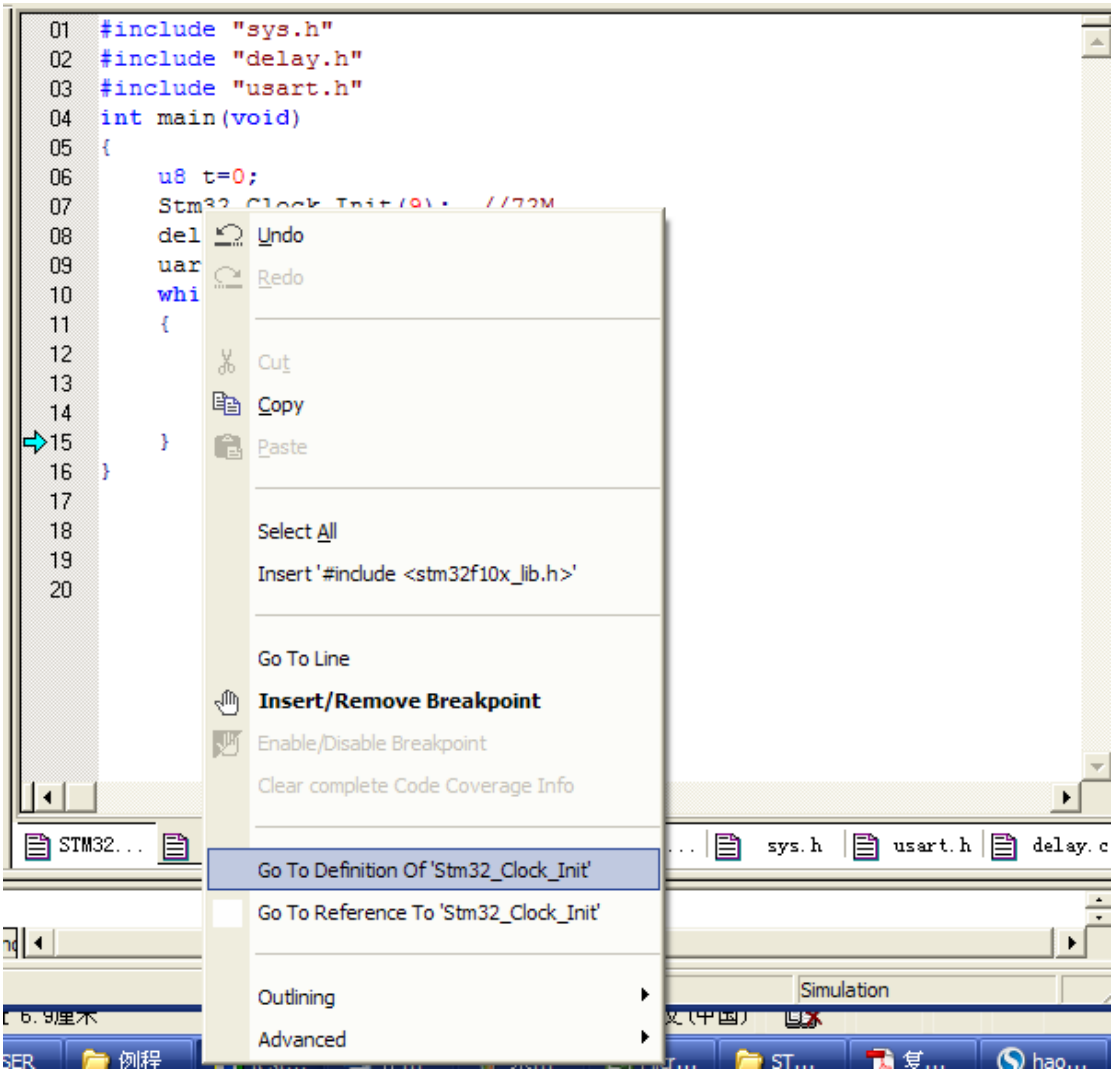


图 2.8.2.3 快速定位

在上图，我们找到 Go to Definition Of‘STM32_Clock_Init’ 这个地方，然后单击就可以快速跳到 STM32_Clock_Init 函数的定义处(注意要先在 Options for Target 的 Output 选项卡里面勾选 Browse Information 选项，再编译，再定位，否则无法定位!!!)。入下图所示：

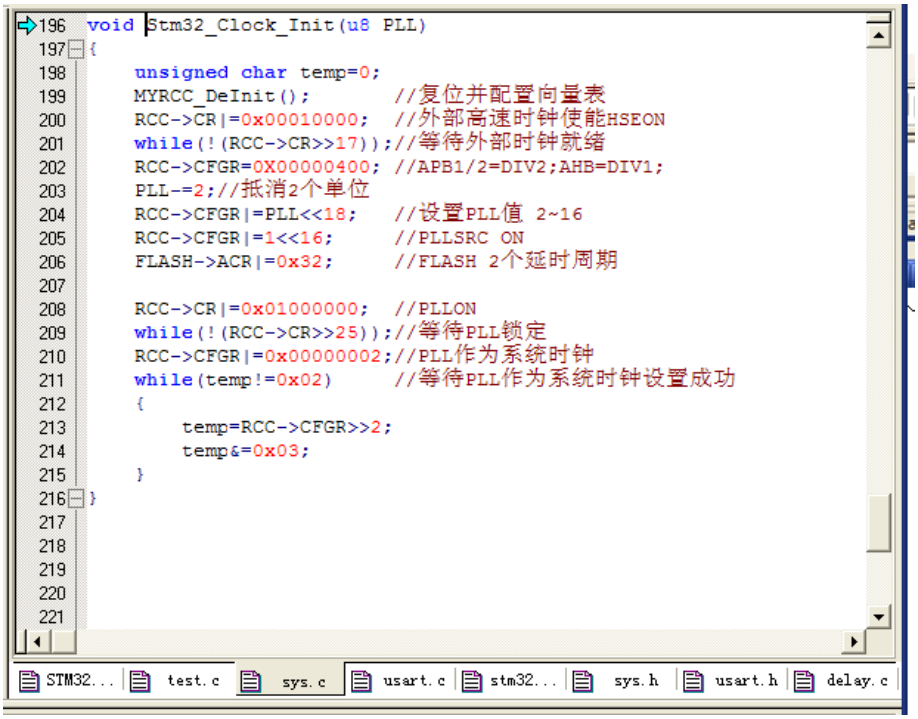


图 2.8.2.4 定位结果

对于变量，我们也可以按这样的操作快速来定位这个变量被定义的地方，大大缩短了你查找代码的时间。细心的大家会发现上面还有一个类似的选项，就是 Go to Reference To ‘STM32_Clock_Init’，这个是快速跳到该函数被声明的地方，有时候也会用到，但不如前者使用得多。

3）快速注释与快速消注释

接下来，我们介绍一下快速注释与快速消注释的方法。在调试代码的时候，你可能会想注释某一片的代码，来看看执行的情况，MDK 提供了这样的快速注释/消注释块代码的功能。也是通过右键实现的。这个操作比较简单，就是先选中你要注释的代码区，然后右键，选择 Advanced->Comment Selection 就可以了。

还是以 USART1_IRQHandler 函数为例，比如我要注释掉下图中所选中区域的代码，入下图所示：

```
047
048 void USART1_IRQHandler(void)
049 {
050     u8 res;
051     if(USART1->SR&(1<<5))//接收到数据
052     {
053         res=USART1->DR;
054         if((USART_RX_STA&0x80)==0)//接收未完成
055         {
056             if(USART_RX_STA&0x40)//接收到了0x0d
057             {
058                 if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
059                 else USART_RX_STA|=0x80;    //接收完成了
060             }else //还没收到0X0D
061             {
062                 if(res==0x0d)USART_RX_STA|=0x40;
063                 else
064                 {
065                     USART_RX_BUF[USART_RX_STA&0X3F]=res;
066                     USART_RX_STA++;
067                     if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错
068                 }
069             }
070         }
071     }
072 }
```

图 2.8.2.5 选中要注释的区域

我们只要在选中了之后，选择右键，再选择 Advanced->Comment Selection 就可以把这段代码注释掉了。执行这个操作以后的结果如下图所示：

```
047
048 void USART1_IRQHandler(void)
049 {
050     u8 res;
051     if(USART1->SR&(1<<5))//接收到数据
052     {
053         res=USART1->DR;
054 //      if((USART_RX_STA&0x80)==0)//接收未完成
055 //      {
056 //          if(USART_RX_STA&0x40)//接收到了0x0d
057 //          {
058 //              if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
059 //              else USART_RX_STA|=0x80;    //接收完成了
060 //      }else //还没收到0X0D
061 //          {
062 //              if(res==0x0d)USART_RX_STA|=0x40;
063 //              else
064 //              {
065 //                  USART_RX_BUF[USART_RX_STA&0X3F]=res;
066 //                  USART_RX_STA++;
067 //                  if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错
068 //              }
069 //          }
070 //      }
071     }
072 }
```

图 2.8.2.6 注释完毕

这样就快速的注释掉了一片代码，而在某些时候，我们又希望这段注释的代码能快速的取消注释，MDK 也提供了这个功能。与注释类似，先选中被注释掉的地方，然后通过右键->Advanced，不过这里选择的是 Uncomment Selection。

4）其他小技巧

除了前面介绍的几个比较常用的技巧，这里还介绍几个其他的小技巧，希望能让你的代码编写如虎添翼。

第一个是快速打开头文件。在将光标放到要打开的引用头文件上，然后右键选择 Open Document“XXX”，就可以快速打开这个文件了（XXX 是你要打开的头文件名字）。如下图所示：

图 2.8.2.7 快速打开头文件

第二个小技巧是查找替换功能。这个和 WORD 等很多文档操作的替换功能是差不多的，在 MDK 里面查找替换的快捷键是“CTRL+H”，只要你按下该按钮就会调出如下界面：

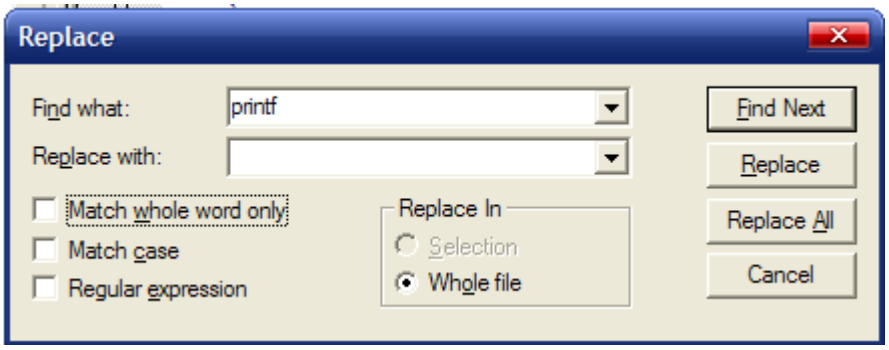


图 2.8.2.7 替换文本

这个替换的功能在有的时候是很有用的，它的用法与其他编辑工具或编译器的差不多，相信各位都不陌生了，这里就不在啰唆了。

2.8.3 调试技巧

之前的软件仿真已经介绍过了，在这里我们要介绍的是如何利用 MDK 自带的例程来快速编写代码。当我开始接触 STM32 的时候，基本上没有直接操作寄存器的例子，网上的所有例子几乎都是使用 ST 自带的例程。也就是 MDK 自带的使用库函数的版本。所以在开始的时候，我就根据 MDK 提供的例子，对照数据手册，再结合 MDK 提供的查看寄存器的功能，一步步把带库函数的例子改成了直接操作寄存器的。一开始会比较难，但是当你掌握规律了之后，就

可以很快的把 MDK 的使用库函数的代码改为直接操作寄存器的代码。这里总结几个要点：

1 个开发板。

这个是实验的基础，有时候软件仿真通过了，在板上并不一定能跑起来，而且有个开发板在手，什么东西都可以直观的看到，效果不是仿仿真能比的。但开发板不宜多，多了你自己都不知道该学哪个了，觉得这个也还可以，那个也不错，那就这个学半天，那个学半天，结果学个四不像。倒不如从一而终，学完一个在学另外一个。

2 本数据手册。即《STM32 参考手册》和《CM3 权威指南》。

《STM32 参考手册》是 ST 出的官方资料，有 STM32 的详细介绍，包括了 STM32 的各种寄存器定义以及功能等，是学习 STM32 的必备资料之一。而《CM3 权威指南》则是对《STM32 参考手册》的补充，后者一般认为使用 STM32 的人都对 CM3 有了较深的了解，所以 CM3 的很多东西它只是一笔带过，但前者对 CM3 有非常详细的说明，这样两者搭配，你就基本上任何问题都能得到解决了。

多做实验，多做笔记。

一个初学者，一开始对 STM32 一般是没有概念的，所以首先要做的就是多做实验，一定要相信实践出真知，结合上面 2 本手册，你很快就会熟悉 STM32，进而随心所欲。其次要多做笔记，在你不知道的时候，找 MDK 的例子，找 2 本手册，当你碰到新的知识点的时候，把它记下来，俗话说：好记心不如烂笔头。将你刚学到的东西用笔记下了，对以后没有坏处。

MDK 的例子分为 2 部分，一部分是与 USB 无关的，这部分代码存放在：D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Examples 目录下，而另外一部分与 USB 相关的例子则存放在：D:\KEIL3.80A\ARM\Examples\ST\STM32F10xUSBLib\Demos 目录下（D 盘是我 MDK 的安装盘，所以这里路径是这样的，如果你安装在其他位置，修改为相应的目录既可以）。

接下来我们用一个实例，来说明如何参考 MDK 的例子为自己所用。希望能起到抛砖引玉的作用。这里以一个 IO 口翻转为例，其实就是 LED 的闪烁，看看如何借用 MDK 的代码。首先打开 D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Examples 目录，可以看到很多例子，如下图所示：

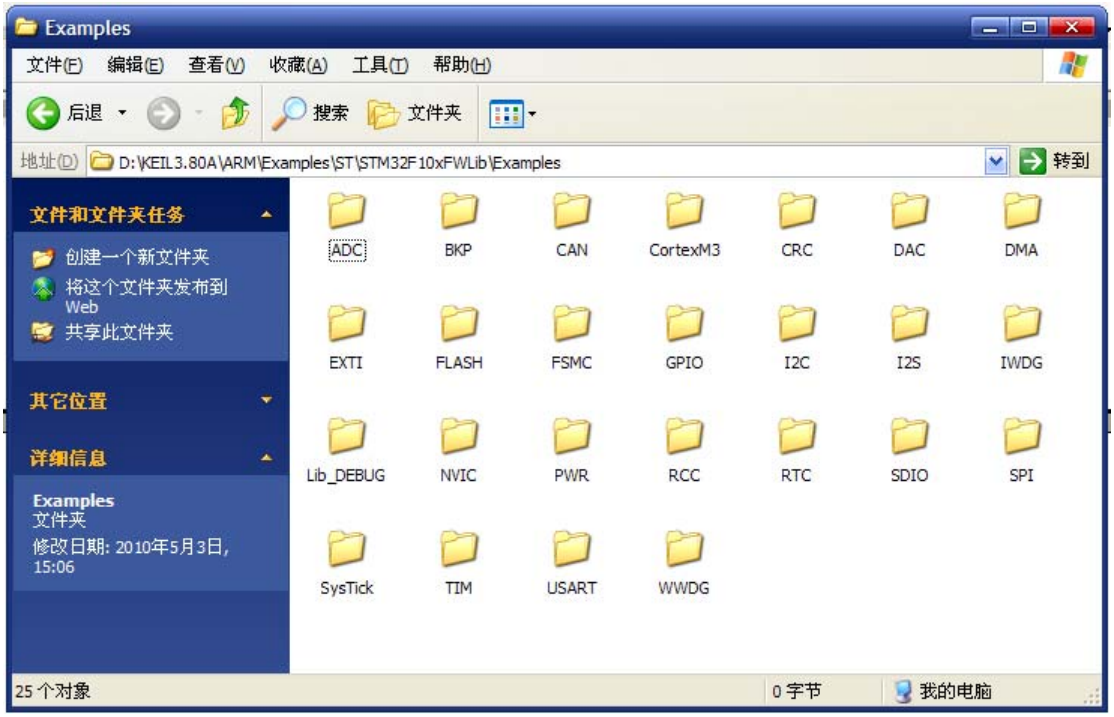


图 2.8.3.1 ST 提供的例程

IO 口翻转的例子在 GPIO 目录下的 IOToggle 下，我们将这个目录下面的所有文件拷贝到 D:\KEIL3.80A\ARM\Examples\ST\STM32F10xFWLib\Project 里面，这里会提示如下信息：

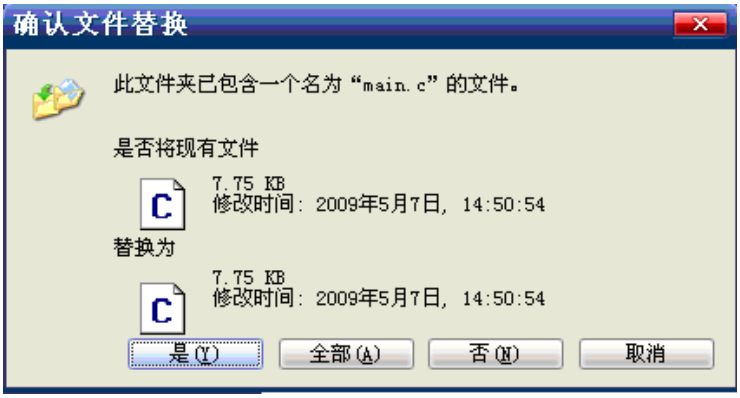


图 2.8.3.2 替换原有文件

我们选择全部就可以了。然后单击 Project.Uv2，打开工程，如下图所示：

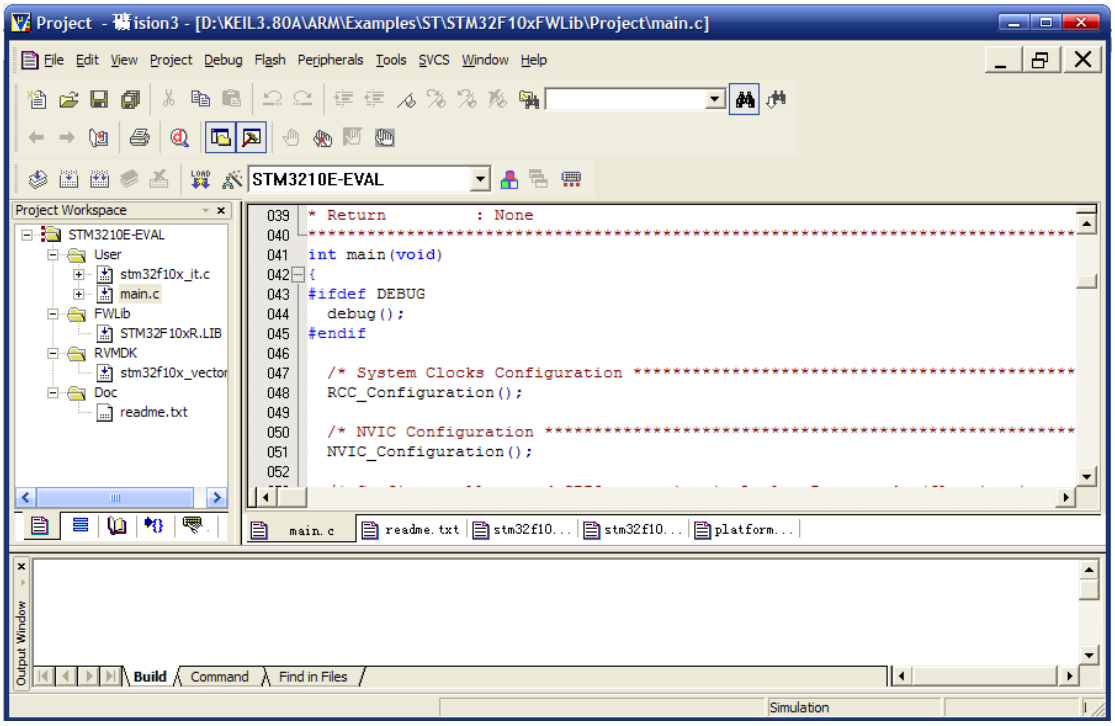


图 2.8.3.3 替换后的工程

然后点击，编译一遍。可以看到如下编译结果：

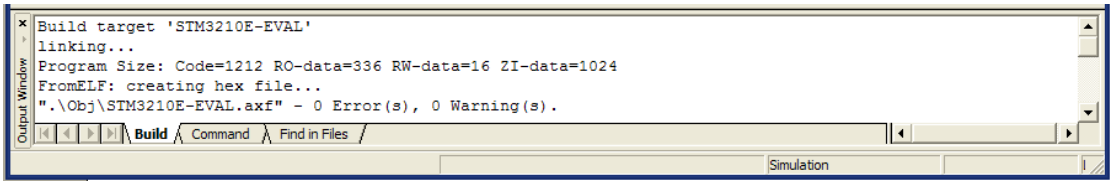


图 2.8.3.4 编译新工程

提示没有错误，没有警告。说明这个工程是可以用的。关于这个工程是如何使用的，在 readme.txt 里面是有详细说明的，在使用之前最好先看看这个说明。重点看看硬件环境的说明：

```
Hardware environment
====================
This example runs on STMicroelectronics STM3210B-EVAL and STM3210E-EVAL evaluation
boards and can be easily tailored to any other hardware.
To select the STMicroelectronics evaluation board used to run the example, uncomment
the corresponding line in platform_config.h file.

 + STM3210B-EVAL
    - Use LD1, LD2, LD3 and LD4 leds connected respectively to PC.06, PC.07, PC.08
      and PC.09 pins

 + STM3210E-EVAL
    - Use LD1, LD2, LD3 and LD4 leds connected respectively to PF.06, PF0.7, PF.08
      and PF.09 pins
```

图 2.8.3.5 查看说明

从上面的说明可以知道，这个 LED 的翻转程序，对 2 套板子分别是连在哪几个 IO 口上的，我们这个是在 USE_STM3210E_EVAL 板上运行的，所以使用的是 PF.6~9。

接下来我们要做的就是一步步跟踪代码，然后针对你的疑问点，打开 Peripherals 里面的相关外设，查看寄存器，看看 MDK 的示例代码是如何一步步修改里面的寄存器来实现的。对与外设的配置，MDK 一般都是调用库函数实现的，无法直接查看，这就需要你对照手册，慢慢摸索了，根据从寄存器看到的结果，大概也就能推出 MDK 是如何实现这样的操作了。其次一个重要的方法是通过查看汇编代码，来看到底是如何操作的，由于鄙人对汇编不熟悉，这里就不废话了，免得误导大家。

这样对照着 MDK 的例子，看看自己的代码在哪些地方和它有不一样的地方，如果出了问题，很可能就在这些不同的地方，只要根据 MDK 的示例来修改，一般你的问题就能得到解决。当然，这过程中需要多多查看手册，看看手册里怎么说的，MDK 又是怎么做的。

第三章 实战篇

经过前两章的学习，我们对 ALIENTEK MiniSTM32 开发板的硬件，以及其开发环境有了个比较深入的了解了，接下来就是通过实战，来真正开始 STM32 的开发。通过本章的学习，你将学会 STM32 的大部分外设的使用。本章将从浅入深向大家介绍如何一步步开发 STM32，使你真正能独立用 STM32 开发自己的东西。本章将通过二十八个实例，向大家介绍 STM32 的强大功能，及开发实例。

3.1 跑马灯实验

通过本节的学习，你将了解到 STM32 的 IO 口作为输出使用的方法。本节分为如下几个小节：

3.1.1 STM32 IO 口简介
3.1.2 硬件设计
3.1.3 软件设计
3.1.4 仿真与下载

3.1.1 STM32 IO 简介

作为所有开发板的经典入门实验，莫过于跑马灯了。ALIENTEK MiniSTM32 开发板板载了 2 个 LED，DS0 和 DS1，本实验将通过教你如何控制这两个灯实现交替闪烁的类跑马灯效果。

该实验的关键在于如何控制 STM32 的 IO 口输出。了解了 STM32 的 IO 口如何输出的，就可以实现跑马灯了。通过这一节的学习，你将初步掌握 STM32 基本 IO 口的使用，而这是迈向 STM32 的第一步。

STM32 的 IO 口可以由软件配置成 8 种模式：

1、输入浮空
2、输入上拉
3、输入下拉
4、模拟输入
5、开漏输出
6、推挽输出
7、推挽式复用功能
8、开漏复用功能

每个 IO 口可以自由编程，单 IO 口寄存器必须要按 32 位字被访问。STM32 的很多 IO 口都是 5V 兼容的，这些 IO 口在与 5V 电平的外设连接的时候很有优势，具体哪些 IO 口是 5V 兼容的，可以从该芯片的数据手册管脚描述章节查到（I/O Level 标 FT 的就是 5V 电平兼容的）。

STM32 的每个 IO 端口都有 7 个寄存器来控制。他们分别是：配置模式的 2 个 32 位的端口配置寄存器 CRL 和 CRH；2 个 32 位的数据寄存器 IDR 和 ODR；1 个 32 位的置位/复位寄存器 BSRR；一个 16 位的复位寄存器 BRR；1 个 32 位的锁存寄存器 LCKR；这里我们仅介绍常用 的几个寄存器，我们常用的 IO 端口寄存器只有 4 个：CRL、CRH、IDR、ODR。

CRL 和 CRH 控制着每个 IO 口的模式及输出速率。

STM32 的 IO 口位配置表如表 3.1.1.1 所示：

<table>
<tr><th colspan="2">配置模式</th><th>CNF1</th><th>CNF0</th><th>MODE1</th><th>MODE0</th><th>PxODR寄存器</th></tr>
<tr><td rowspan="2">通用输出</td><td>推挽式(Push-Pull)</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">01
10
11
见表3.1.2</td><td>0 或 1</td></tr>
<tr><td>开漏(Open-Drain)</td><td>1</td><td>0 或 1</td></tr>
<tr><td rowspan="2">复用功能输出</td><td>推挽式(Push-Pull)</td><td rowspan="2">1</td><td>0</td><td>不使用</td></tr>
<tr><td>开漏(Open-Drain)</td><td>1</td><td>不使用</td></tr>
<tr><td rowspan="4">输入</td><td>模拟输入</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">00</td><td>不使用</td></tr>
<tr><td>浮空输入</td><td>1</td><td>不使用</td></tr>
<tr><td>下拉输入</td><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>上拉输入</td><td>1</td></tr>
</table>

表 3.1.1.1 STM32 的 IO 口位配置表

STM32 输出模式配置如表 3.1.1.2 所示：

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50MHz

表 3.1.1.2 STM32 输出模式配置表

接下来我们看看端口低配置寄存器 CRL 的描述，如下图所示：

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	**CNFy[1:0]**：端口x配置位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15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29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) 软件通过这些位配置相应的I/O端口，请参考表15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

图 3.1.1.1 端口低配置寄存器 CRL 各位描述

该寄存器的复位值为 0X4444 4444，从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每个 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X4 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能，50M 速率）。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口。这里我们对 CRH 就不做详细介绍了。

给个实例，比如我们要设置 PORTC 的 11 位为上拉输入，12 位为推挽输出。代码如下：

```
GPIOC->CRH&=0XFFF00FFF;//清掉这 2 个位原来的设置，同时也不影响其他位的设置
GPIOC->CRH|=0X00038000;  //PC11 输入，PC12 输出
GPIOC->ODR=1<<11;//PC11 上拉
```

通过这 3 句话的配置，我们就设置了 PC11 为上拉输入，PC12 为推挽输出。

IDR 是一个端口输入数据寄存器，只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。该寄存器各位的描述如下图所示：

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	**IDRy[15:0]**：端口输入数据(y = 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。

图 3.1.1.2 端口输入数据寄存器 IDR 各位描述

要想知道某个 IO 口的状态，你只要读这个寄存器，再看某个位的状态就可以了。使用起来是比较简单的。

ODR 是一个端口输出数据寄存器，也只用了低 16 位。该寄存器为可读写，从该寄存器读出来的数据可以用于判断当前 IO 口的输出状态。而向该寄存器写数据，则可以控制某个 IO 口的输出电平。该寄存器的各位描述如下图所示：

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。

图 3.1.1.3 端口输出数据寄存器 ODR 各位描述

了解了这几个寄存器，我们就可以开始跑马灯实验的真正设计了。关于 IO 口更详细的介绍，请参考《STM32 参考手册》第 105 页 8.1 节。

在此，我们可以总结一下，对于学过 AVR 的人来说，我们都知道 AVR 的 IO 口由 3 个寄存器控制：DDR、 PORT、PIN。这里我们可以拿 STM32 的 IO 控制寄存器和 AVR 的来个类比：

1， STM32 的 CRL 和 CRH 就相当于 AVR 的 DDR 寄存器，用来控制 IO 口的方向，只不过 STM32 的 CRL 和 CRH 功能更强大一点罢了。

2， STM32 的 ODR 就相当于 AVR 的 PORT，都是用来控制 IO 口的输出电平或者上下拉电阻的。

3， STM32 的 IDR 就相当于 AVR 的 PIN，都是用来存储 IO 口当前的输入状态（高低电平）的。

除此之外，STM32 还有 BSRR、BRR、LCKR 等几个寄存器用于控制 IO 口，这点是 AVR 所没有的。

3.1.2 硬件设计

该实验的硬件电路在 ALIENTEM Mini STM32 开发板上默认是已经连接好了的。DS0 接

PA8，DS1 接 PD2。所以在硬件上不需要动任何东西。其连接原理图如下：

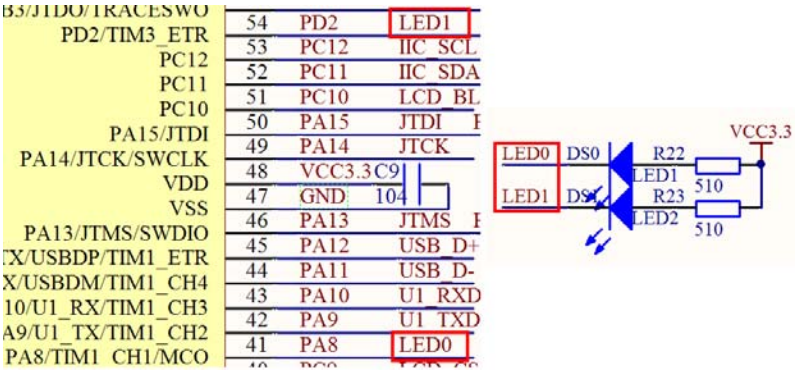


图 3.1.2.1 LED 与 STM32 连接原理图

3.1.3 软件设计

首先，找到之前新建的 TEST 工程，在该文件夹下面新建一个 HARDWARE 的文件夹，用来存储以后与硬件相关的代码。然后在 HARDWARE 文件夹下新建一个 LED 文件夹，用来存放与 LED 相关的代码。如下图所示：

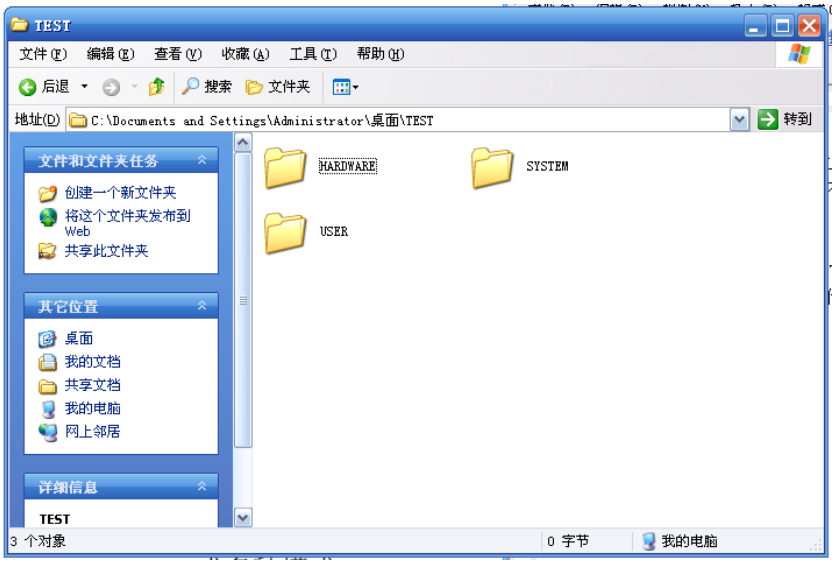


图 3.1.3.1 新建 HARDWARE 文件夹

然后我们打开 USER 文件夹下的 TEST.Uv2 工程，按  按钮新建一个文件，然后保存在 HARDWARE->LED 文件夹下面，保存为 led.c。在该文件中输入如下代码：

```
#include <stm32f10x_lib.h>
#include "led.h"
//Mini STM32 开发板
//LED 驱动代码
//正点原子@ALIENTEK
//2010/5/27
```

```
// V1.0
//初始化 PA8 和 PD2 为输出口.并使能这两个口的时钟

//LED IO 初始化
void LED_Init(void)
{
    RCC->APB2ENR|=1<<2;     //使能 PORTA 时钟
    RCC->APB2ENR|=1<<5;     //使能 PORTD 时钟
    GPIOA->CRH&=0XFFFFFFF0;
    GPIOA->CRH|=0X00000003;//PA8 推挽输出
    GPIOA->ODR|=1<<8;       //PA8 输出高
    GPIOD->CRL&=0XFFFFF0FF;
    GPIOD->CRL|=0X00000300;//PD.2 推挽输出
    GPIOD->ODR|=1<<2;       //PD.2 输出高
}
```

该代码里面就包含了一个函数 void LED_Init(void)，该函数的功能就是用来实现配置 PA8 和 PD2 为推挽输出。在配置 STM32 外设的时候，任何时候都要先使能该外设的时钟！APB2ENR 是 APB2 总线上的外设时钟使能寄存器，其各位的描述如下：

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
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART1 EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	保留	AFIO EN

图 3.1.3.2 寄存器 APB2ENR 各位描述

我们要使能的 PORTA 和 PORTD 的时钟使能位，分别在 bit2 和 bit5，只要将这两位置 1 就可以使能 PORTA 和 PORTD 的时钟了。该寄存器还包括了很多其他外设的时钟使能。大家在以后会慢慢使用到的。关于这个寄存器的详细说明在《STM32 参考手册》的第 70 页。

在设置完时钟之后就是配置完时钟之后，LED_Init 配置了 PA8 和 PD2 的模式为推挽输出，并且默认输出 1。这样就完成了对这两个 IO 口的初始化。

保存 led.c 代码，然后我们按同样的方法，新建一个 led.h 文件，也保存在 LED 文件夹下面。在 led.h 中输入如下代码：

```
#ifndef __LED_H
#define __LED_H
#include "sys.h"
//Mini STM32 开发板
//LED 驱动代码
//正点原子@ALIENTEK
//2010/5/27
//LED 端口定义
#define LED0 PAout(8)// PA8
#define LED1 PDout(2)// PD2
void LED_Init(void);//初始化
```

#endif

这段代码里面最关键就是 2 个宏定义:

#define LED0 PAout(8)// PA8

#define LED1 PDout(2)// PD2

这里使用的是位带操作来实现操作某个 IO 口的 1 个位的，关于位带操作前面已经有介绍，这里不再多说。需要说明的是，这里可以使用另外一种操作方式实现。如下：

```
#define    LED0 (1<<8)    //led0 PA8
#define    LED1 (1<<2)    //led1    PD2
#define LED0_SET(x) GPIOA->ODR=(GPIOA->ODR&~LED0)|(x ? LED0:0)
#define LED1_SET(x) GPIOD->ODR=(GPIOD->ODR&~LED1)|(x ? LED1:0)
```

后者通过 LED0_SET(0)和 LED0_SET(1)来控制 PA8 的输出 0 和 1。而前者的类似操作为：LED0=0 和 LED0=1。显然前者简单很多，从而可以看出位带操作带来的好处。以后像这样的 IO 口操作，我们都使用位带操作来实现，而不使用第二种方法。

将 led.h 也保存一下。接着，我们在 Manage Components 管理里面新建一个 HARDWARE 的组，并把 led.c 加入到这个组里面，如下图所示：

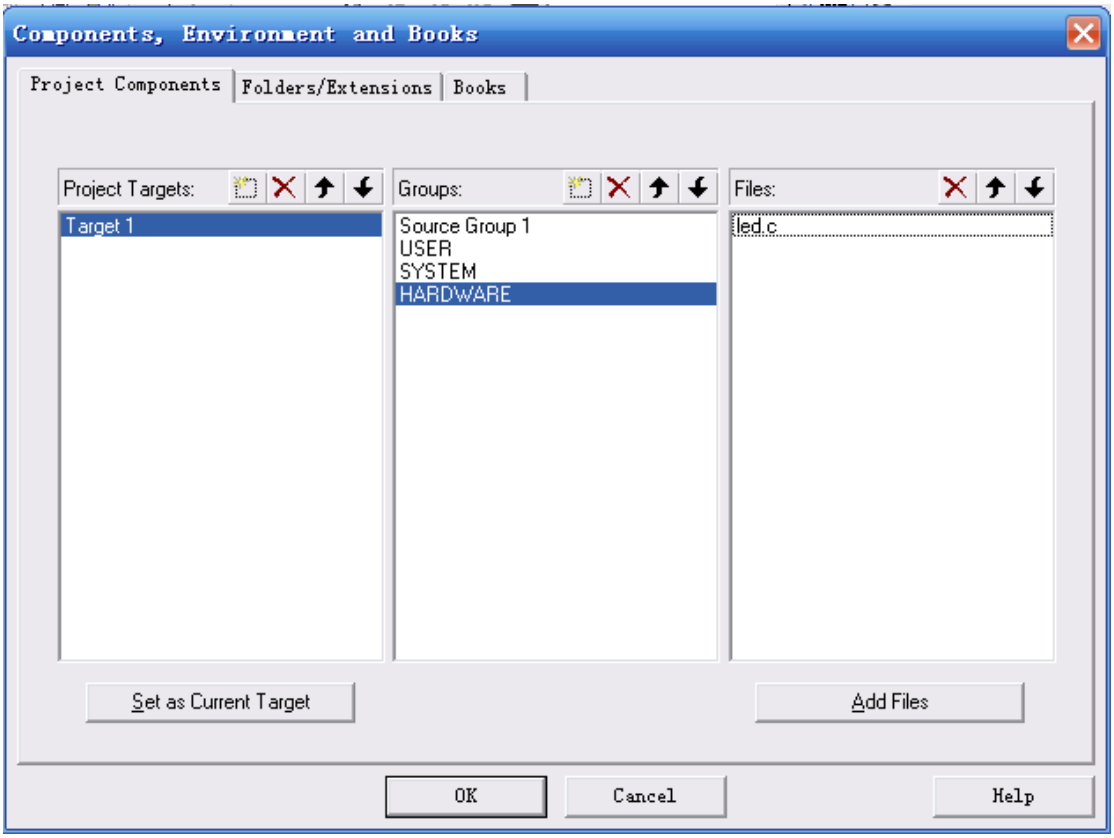


图 3.1.3.3 给工程新增 HARDWARE 组

单击 OK，回到工程，然后你会发现在 Project Workspace 里面多了一个 HARDWARE 的组，在改组下面有一个 led.c 的文件。如下图所示：

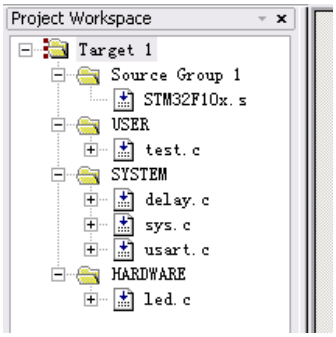


图 3.1.3.4 给工程新增 HARDWARE 组

然后用之前介绍的方法将 led.h 头文件的路径加入到工程里面（参照本文第 36 页）。回到主界面，在 main 函数里面编写如下代码：

```
#include <stm32f10x_lib.h>
#include "sys.h"
#include "usart.h"
#include "delay.h"
#include "led.h"
//Mini STM32 开发板范例代码 1
//跑马灯实验
//正点原子@ALIENTEK
//2010.5.27
int main(void)
{
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);      //延时初始化
    LED_Init();          //初始化与 LED 连接的硬件接口
    while(1)
    {
        LED0=0;
        LED1=1;
        delay_ms(300);
        LED0=1;
        LED1=0;
        delay_ms(300);
    }
}
```

代码先包含了#include "led.h"这句，使得 LED0、LED1、LED_Init 等能在 main 函数里被调用。接下来，main 函数先配置系统时钟为 72M，然后把延时函数初始化一下。接着就是调用 LED_Init 来初始化 PA8 和 PD2 为输出。最后在死循环里面实现 LED0 和 LED1 交替闪烁，间隔为 300ms。

然后按  ，编译工程，得到结果如下图所示：

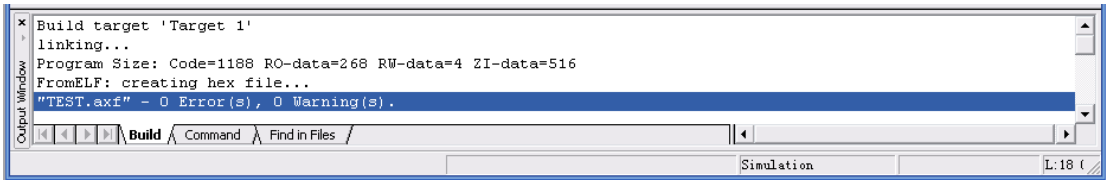


图 3.1.3.5 编译结果

可以看到没有错误，也没有警告。接下来，我们就先进行软件仿真，验证一下是否有错误的地方，然后下载到 Mini STM32 看看实际运行的结果。

3.1.4 仿真与下载

我们可以先用软件仿真，看看结果对不对，根据软件仿真的结果，然后再下载到 MINI STM32 板子上面看运行是否正确。

首先，我们进行软件仿真（注意在 MDK 的 Debug 选项卡里面设置为：Use Simulator）。先按  开始仿真，接着按  ，显示逻辑分析窗口，点击 Setup，新建两个信号 PORTA.8 和 PORTD.2，如下图所示：

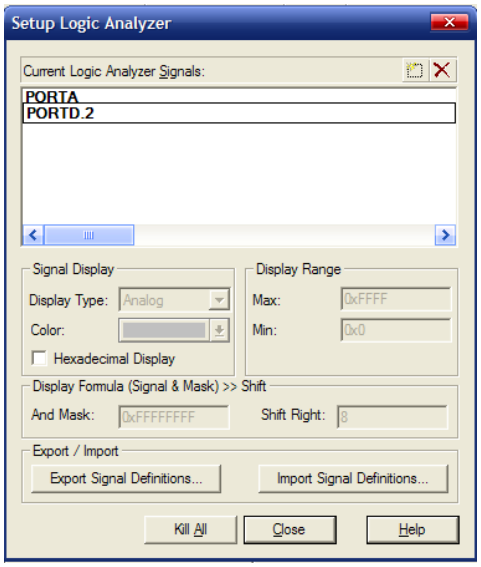


图 3.1.4.1 逻辑分析设置

Display Type 选择 bit，然后单击 Close 关闭该对话框，可以看到逻辑分析窗口出来了 2 个信号，如下图所示：

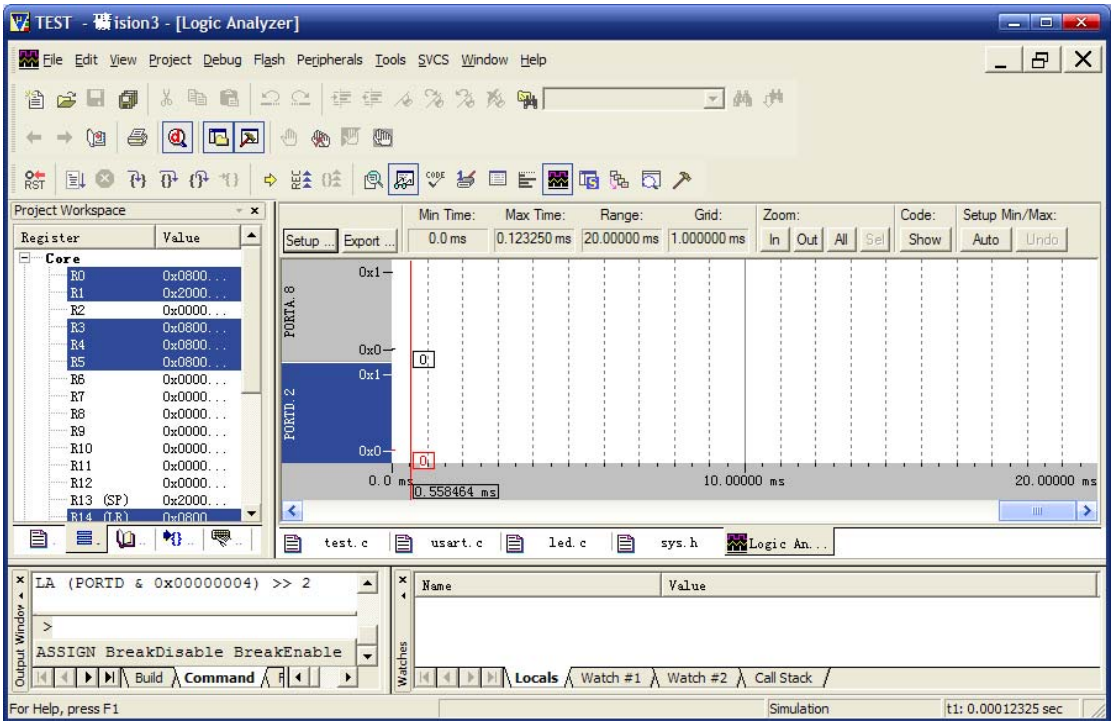


图 3.1.4.2 设置后的逻辑分析窗口

接着，点击  ，开始运行。运行一段时间之后，按  按钮，暂停仿真回到逻辑分析窗口，可以看到如下波形：

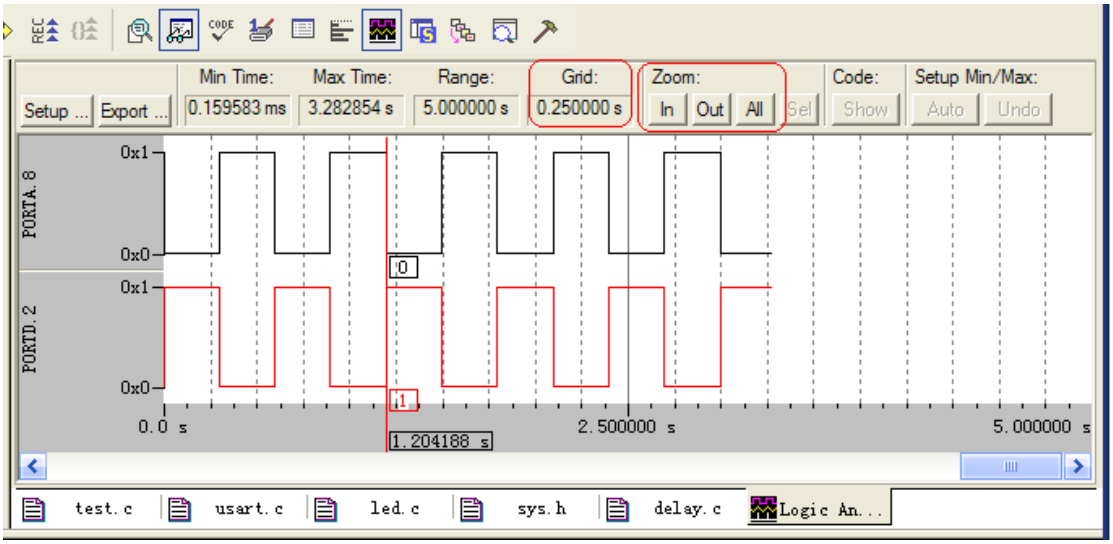


图 3.1.4.3 仿真波形

这里注意 Gird 要调节到 0.25s 左右比较合适，可以通过 Zoom 里面的 In 按钮来放大波形，通过 Out 按钮来缩小波形，或者按 All 显示全部波形。从上图中可以看到 PORTD.2 和 PORTA.8 交替输出，周期可以通过中间那根红线来测量。至此，我们的软件仿真已经顺利通过。

在软件仿真没有问题了之后，我们就可以把代码下载到 Mini STM32 观看运行结果是否与我们仿真的一致。运行结果如下图所示：

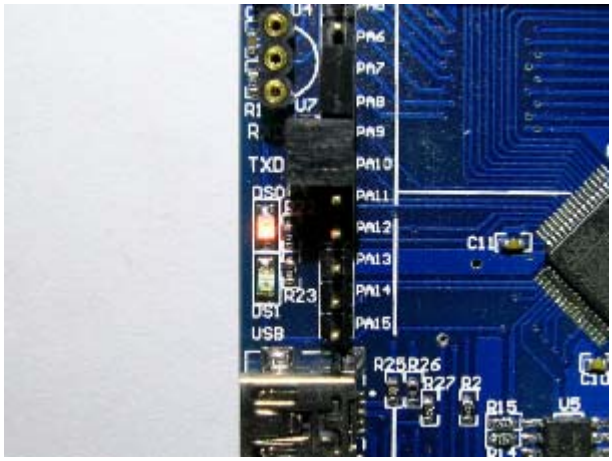


图 3.1.4.4 执行结果

至此，我们的第一节学习就结束了，这一节，作为 STM32 的入门第一个例子，详细介绍了 STM32 的 IO 口操作，同时巩固了前面的学习，并进一步介绍了 MDK 的软件仿真功能。希望大家好好理解一下。

3.2 按键输入实验

上一节介绍了 STM32 的 IO 口输出，这一节，我们将向大家介绍如何使用 STM32 的 IO 口作为输入用。通过本节的学习，你将了解到 STM32 的 IO 口作为输入使用的方法。本节分为如下几个小节：

3.2.1 STM32 IO 口简介
3.2.2 硬件设计
3.2.3 软件设计
3.2.4 仿真与下载

3.2.1 STM32 IO 口简介

STM32 的 IO 口在上一节已经有了详细的介绍，这里我们不再多说。STM32 的 IO 口做输入使用的时候，是通过读取 IDR 的内容来读取 IO 口的状态的。了解了这点，就可以开始我们的代码编写了。

这一节，我们将通过 MiniSTM32 板上载有的 3 个按钮，来控制板上的 2 个 LED，其中 KEY0 控制 DS0，按一次亮，再按一次，就灭。KEY1 控制 DS1，效果同 KEY0。KEY_2（KEY_UP），同时控制 DS0 和 DS1，按一次，他们的状态就翻转一次。

3.2.2 硬件设计

该实验所需要的硬件电路在 MiniSTM32 开发板上都已经连接好了，不需要经过任何设置，直接编写代码就可。LED 的连接在上一节已经介绍过了，在 MiniSTM32 开发板上的按键 KEY0 是接在 PA13 上，KEY1 是接在 PA15 上的，WK_UP(KEY2)接在 PA0 上。如下图所示：

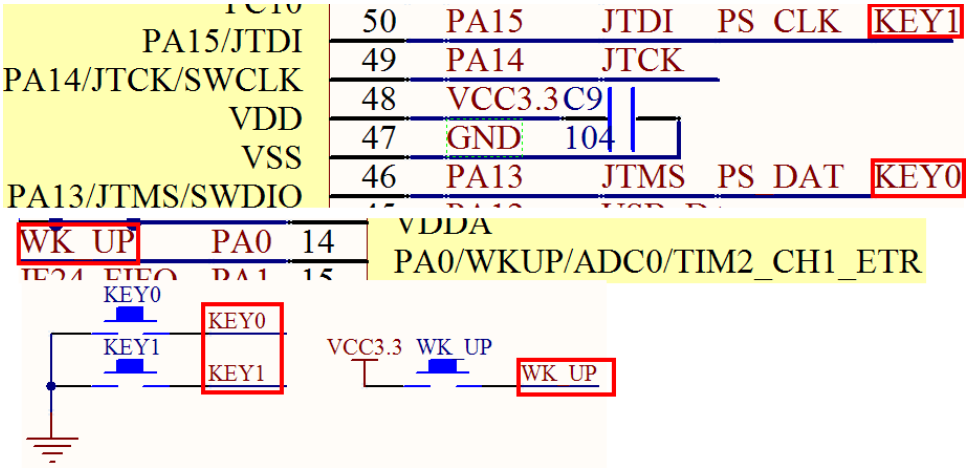


图 3.2.2.1 按键与 STM32 连接原理图

这里需要注意的是 KEY0 和 KEY1 是低电平有效的，而 WK_UP 是高电平有效的，而且要确认 WK_UP 按钮与 DS18B20 的连接是否已经断开，要先断开，否则 DS18B20 会干扰 WK_UP 按键！并且 KEY0 和 KEY1 连接在与 JTAG 相关的 IO 口上，所以在软件编写的时候要先禁用 JTAG 功能，才能把这两个 IO 口当成普通 IO 口使用。

3.2.3 软件设计

这里的代码设计，我们还是在之前的基础上继续编写，打开 3.1 节的 TEST 工程，然后在 HARDWARE 文件夹下新建一个 KEY 文件夹，用来存放与 KEY 相关的代码。如下图所示：

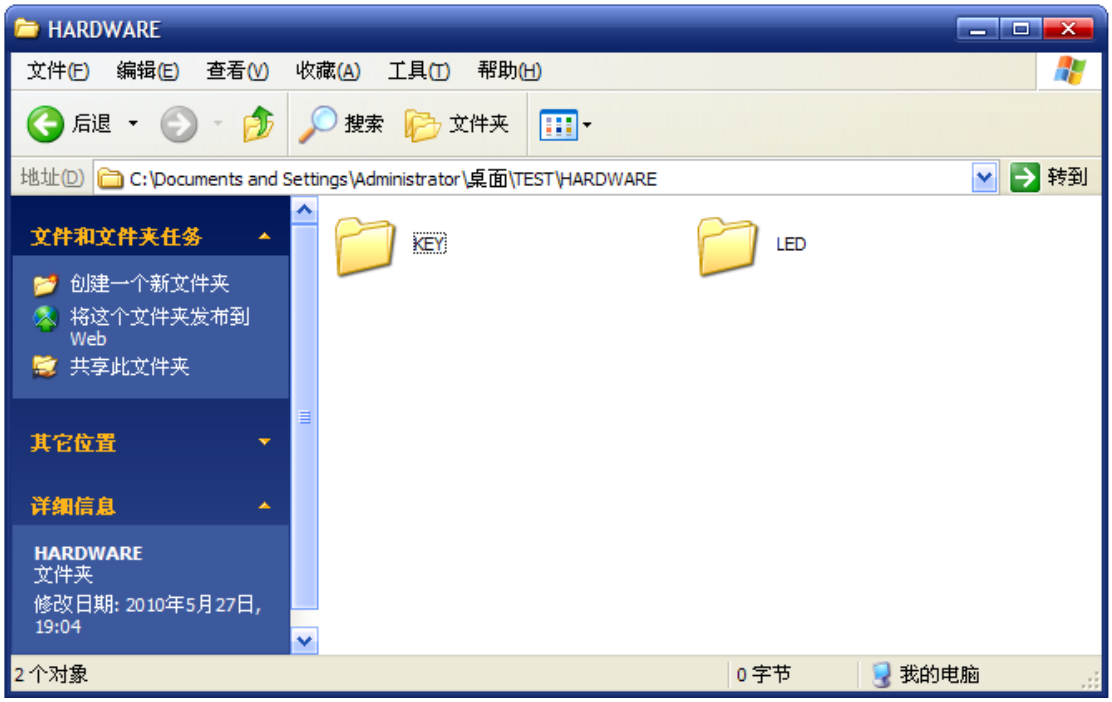


图 3.2.3.1 在 HARDWARE 下新增 KEY 文件夹

然后我们打开 USER 文件夹下的 TEST.Uv2 工程，按  按钮新建一个文件，然后保存在 HARDWARE->KEY 文件夹下面，保存为 key.c。在该文件中输入如下代码：

```
#include <stm32f10x_lib.h>
#include "key.h"
#include "delay.h"
//ALIENTEK Mini STM32 开发板
//按键输入 驱动代码
//按键初始化函数
//PA0.13.15 设置成输入
void KEY_Init(void)
{
    RCC->APB2ENR|=1<<2;       //使能 PORTA 时钟
    GPIOA->CRL&=0XFFFFFFF0;//PA0 设置成输入
    GPIOA->CRL|=0X00000008;
    GPIOA->CRH&=0X0F0FFFFF;//PA13,15 设置成输入
    GPIOA->CRH|=0X80800000;
    GPIOA->ODR|=1<<13;     //PA13 上拉,PA0 默认下拉
    GPIOA->ODR|=1<<15;     //PA15 上拉
}
//按键处理函数
//返回按键值
//0，没有任何按键按下
//1，KEY0 按下
//2，KEY1 按下
```

```
//3，KEY2 按下 WK_UP
//注意此函数有响应优先级,KEY0>KEY1>KEY2!!
u8 KEY_Scan(void)
{
    static u8 key_up=1;//按键按松开标志
    JTAG_Set(JTAG_SWD_DISABLE);
    if(key_up&&(KEY0==0||KEY1==0||KEY2==1))
    {
        delay_ms(10);//去抖动
        key_up=0;
        if(KEY0==0)
        {
            JTAG_Set(SWD_ENABLE);
            return 1;
        }else if(KEY1==0)
        {
            JTAG_Set(SWD_ENABLE);
            return 2;
        }else if(KEY2==1)
        {
            JTAG_Set (SWD_ENABLE);
            return 3;
        }
    }else if(KEY0==1&&KEY1==1&&KEY2==0)key_up=1;
    JTAG_Set(SWD_ENABLE);
    return 0;// 无按键按下
}
```

这段代码包含 2 个函数，void KEY_Init(void)和 u8 KEY_Scan(void)，KEY_Init 是用来初始化按键输入的 IO 口的。实现 PA0、PA13、PA15 的输入设置，这里和上一节的输出配置差不多，只是这里用来设置成的是输入而上一节是输出。

KEY_Scan 函数，则是用来扫描这 3 个 IO 口是否有按键按下。这个 KEY_Scan 函数，扫描某个按键，该按键按下之后必须要松开，才能第二次触发，否则不会再响应这个按键，这样的好处就是可以防止按一次多次触发，而坏处就是在需要长按的时候比较不合适。同时还有一点要注意的就是，该函数的按键扫描是有优先级的，最优先的是 KEY0，第二优先的是 KEY1，最后是 KEY2（KEY2 对应 WK_UP 按键）。该函数有返回值，如果有按键按下，则返回非 0 值，如果没有或者按键不正确，则返回 0。具体怎么实现请参考 KEY_Scan 的代码。

此外，KEY_Scan 函数里面频繁的调用了 JTAG_Set 函数，用于 JTAG 的开启和关闭，JTAG_Set 函数在 2.7.2 节已经详细向大家介绍过了。这里需要特别说明一下的是：在 KEY_Scan 函数里面，每次开始按键扫描的时候，我们都禁用了 JTAG 和 SWD，这样做的目的是使得 PA13 和 PA15 得以用作普通 IO 口，从而检测 KEY0 和 KEY1 的状态。在按键扫描完成之后，我们又开启了 SWD（JTAG 一直是禁用的），这样让 JLINK 能继续后面的跟踪。这样的好处就是可以在 JLINK 不拔掉的情况下，下载代码，并仿真。不过，由于 JLINK 的存在，会干扰 KEY1 和 KEY0 的按键状态，所以在 JLINK 不拔掉的情况下，是得不到正确的结果的。

另外还要注意，使用 SWD 模式的 JLINK 调试这个代码的时候，不要去调试 KEY_Scan 函数，否则会引起 JLINK 的追踪失效，从而导致调试终止。

保存 key.c 代码，然后我们按同样的方法，新建一个 key.h 文件，也保存在 KEY 文件夹下面。在 key.h 中输入如下代码：

```
#ifndef __KEY_H
#define __KEY_H
#include "sys.h"
//Mini STM32 开发板
//按键输入  驱动代码
//正点原子@ALIENTEK
//2010/5/27
#define KEY0 PAin(13)     //PA13
#define KEY1 PAin(15) //PA15
#define KEY2 PAin(0)   //PA0    WK_UP
void KEY_Init(void);//IO 初始化
u8 KEY_Scan(void);    //按键扫描函数
#endif
```

这段代码里面最关键就是 3 个宏定义:

```
#define KEY0 PAin(13)     //PA13
#define KEY1 PAin(15) //PA15
#define KEY2 PAin(0)   //PA0    WK_UP
```

这里使用的是位带操作来实现读取某个 IO 口的 1 个位的。同输出一样，我们也有另外一种方法可以实现上面代码的功能，如下：

```
#define    KEY0 (1<<13)    //KEY0    PA13
#define    KEY1 (1<<15)    //KEY1    PA15
#define    KEY2 (1<<0)     //KEY2    PA0
#define KEY0_GET() ((GPIOA->IDR&(KEY0))?1:0)//读取按键 0
#define KEY1_GET() ((GPIOA->IDR&(KEY1))?1:0)//读取按键 1
#define KEY2_GET() ((GPIOA->IDR&(KEY2))?1:0)//读取按键 2
```

通输出一样，我们使用第一种方法，比较简单，看起来也清晰明了，最重要的是修改起来比较方便，后续实例，我们一般都使用第一种方法来实现输入口的读取。而第二种方法则适合在不同编译器之间移植，因为他不依靠其他代码。具体选择哪种，大家可以根据自己的喜好来决定。

将 key.h 也保存一下。接着，我们把 key.c 加入到 HARDWARE 这个组里面，这一次我们通过双击的方式来增加新的.c 文件，双击 HARDWARE，找到 key.c，加入到 HARDWARE 里面，如下图所示：

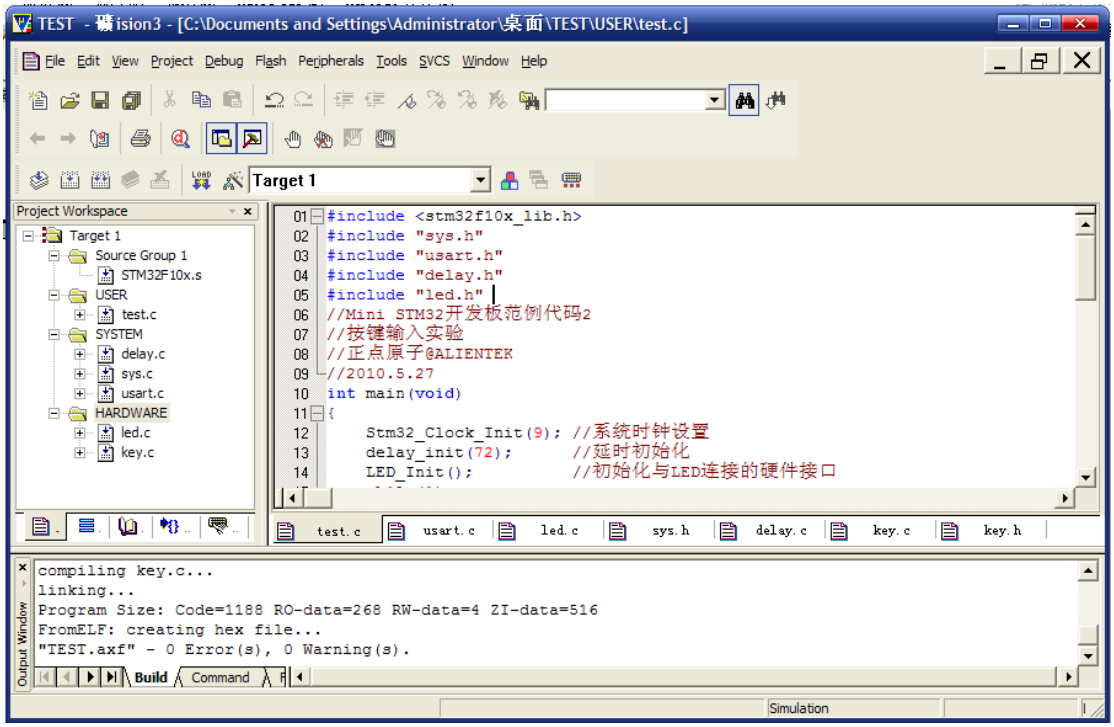


图 3.2.3.2　将 key.c 加入 HARDWARE 组下

可以看到 HARDWARE 文件夹里面多了一个 key.c 的文件， 然后还是用老办法把 key.h 头文件所在的的路径加入到工程里面。回到主界面，在 test.c 里面编写如下代码：

```
#include <stm32f10x_lib.h>
#include "sys.h"
#include "usart.h"
#include "delay.h"
#include "led.h"
#include "key.h"
//Mini STM32 开发板范例代码 2
//按键输入实验
//正点原子@ALIENTEK
//2010.5.27
int main(void)
{
    u8 t;
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);      //延时初始化
    LED_Init();          //初始化与 LED 连接的硬件接口
    KEY_Init();          //初始化与按键连接的硬件接口
    while(1)
    {
        t=KEY_Scan();//得到键值
        if(t)
        {
```

```
                switch(t)
                {
                    case 1:
                        LED0=!LED0;
                        break;
                    case 2:
                        LED1=!LED1;
                        break;
                    case 3:
                        LED0=!LED0;
                        LED1=!LED1;
                        break;
                }
            }
        }
    }
```

注意要将 KEY 文件夹加入头文件包含路径，不能少，否则编译的时候会报错的哦，呵呵。这段实现代码比较简单，就是实现前面简介所阐述的功能。

然后按  ，编译工程，得到结果如下图所示：

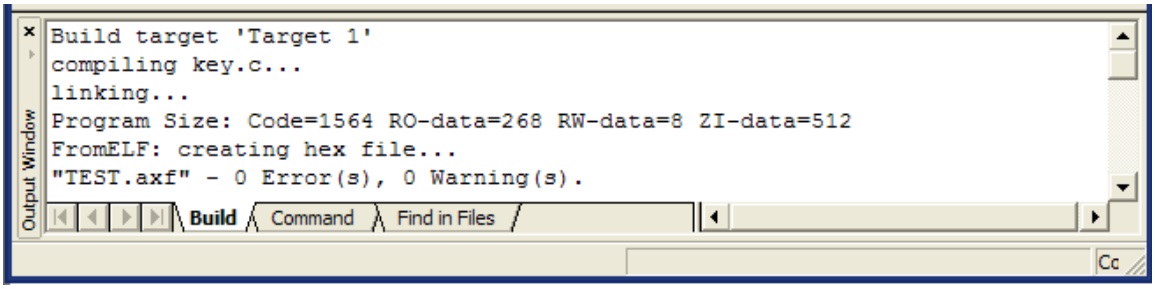


图 3.2.3.3　编译结果

可以看到没有错误，也没有警告。从编译信息可以看出，我们的代码占用 FLASH 大小为：1792 字节（1524+268），所用的 SRAM 大小为：520 个字节。

这里我们解释一下，编译结果里面的几个数据的意义：

Code：表示程序所占用 FLASH 的大小（FLASH）。

RO-data：即 Read Only-data，表示程序定义的常量（FLASH）。

RW-data：即 Read Write-data，表示已可以读写的变量（SRAM）

ZI-data：即 Zero Init-data，表示已被初始化为 0 的变量(SRAM)

有了这个就可以知道您当前使用的 flash 和 sram 大小了，所以，一定要注意的是程序的大小不是.hex 文件的大小，而是编译后的 Code 和 RO-data 之和。

接下来，我们还是先进行软件仿真，验证一下是否有错误的地方，然后才下载到 Mini STM32 看看实际运行的结果。

3.2.4 仿真与下载

我们可以先用软件仿真，看看结果对不对，根据软件仿真的结果，然后再下载到 MINI STM32 板子上面看运行是否正确。

首先，我们进行软件仿真。先按  开始仿真，接着按  ，显示逻辑分析窗口，点击 Setup，新建 5 个信号 PORTA.8、PORTD.2、PORTA.0、PORTA.13、PORTA.15，如下图所示：

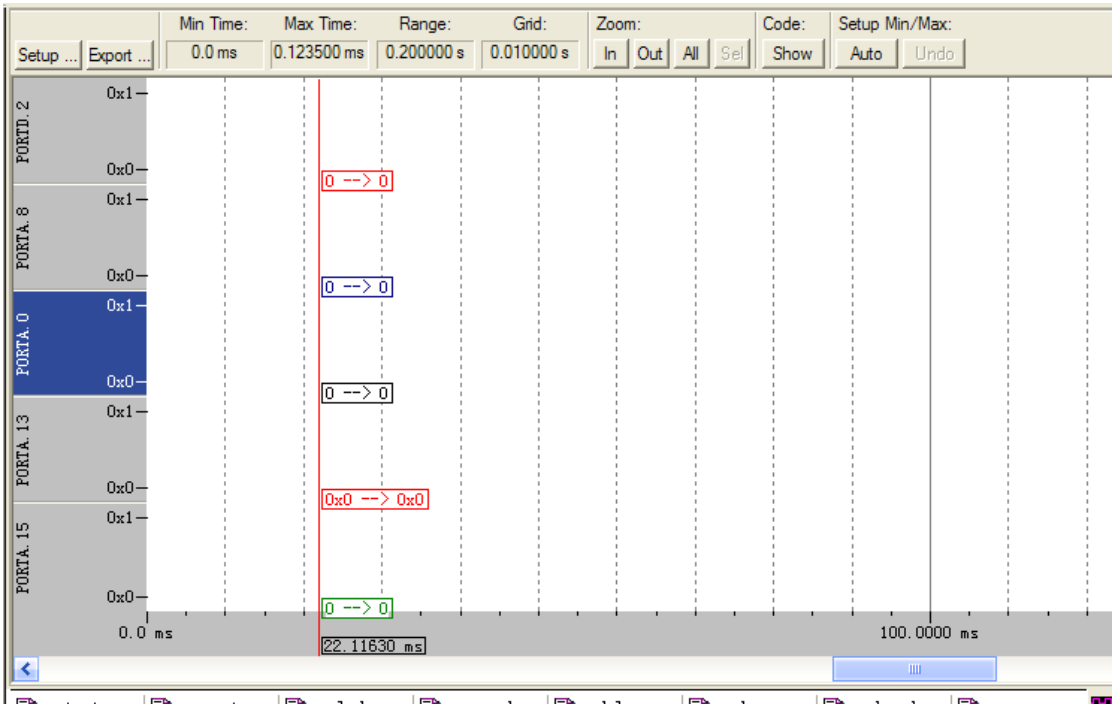


图 3.2.4.1　新建仿真信号

然后再点击 Peripherals->General Purpose I/O->GPIOA，弹出 GPIOA 的查看窗口，如下图所示：

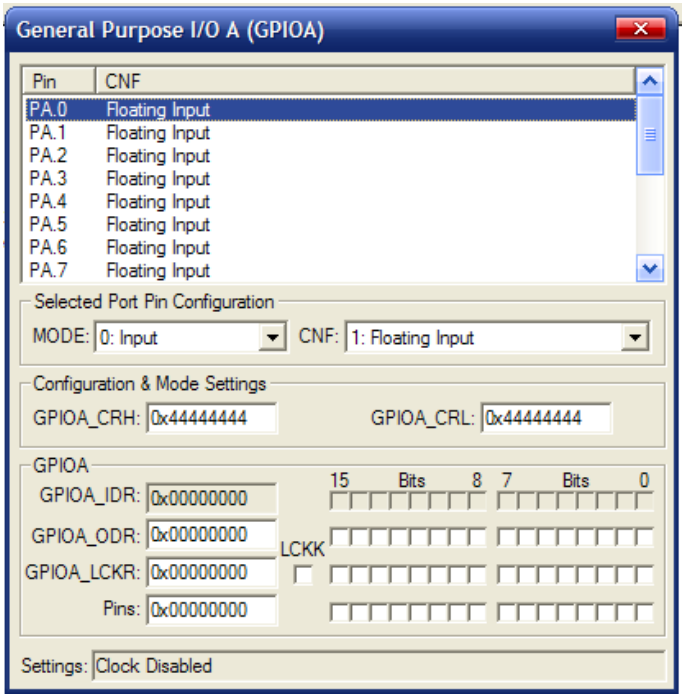


图 3.2.4.2　单看 GPIOA 寄存器

然后在 t=KEY_Scan();这里设置一个断点，按直接执行到这里，然后在 General Purpose I/O A 窗口内设置 GPIOA.0 为 0，GPIOA.13 和 15 为 1，这是因为我们已经设置了这几个 IO 口的状态就是这个样子，而 MDK 不会考虑 STM32 自带的上拉和下拉，所以我们得自己设置一下，来使得其初始状态和外部硬件的状态一摸一样。如下图所示：

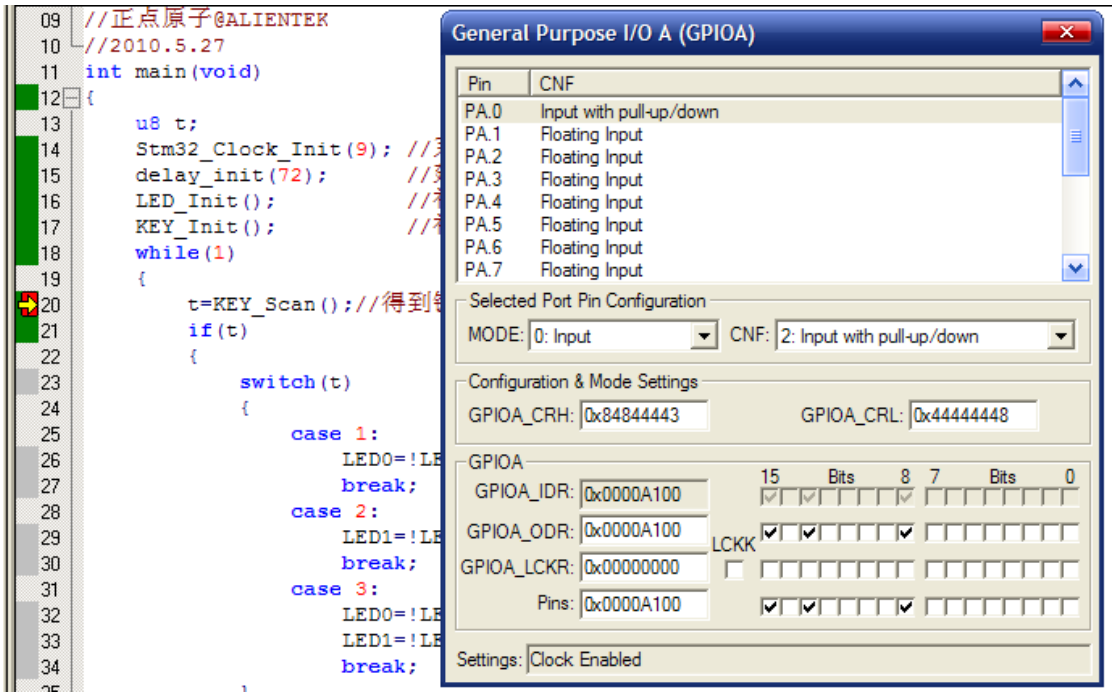


图 3.2.4.3　执行到断点处

接着我们执行过这句，可以看到 t 的值依旧为 0，也就是没有任何按键按下。接着我们再

按，再次执行到 t=KEY_Scan();我们此次把 Pins 的 PA0 勾上，再次执行过这句，得到 t 的值为 3，如下图所示：

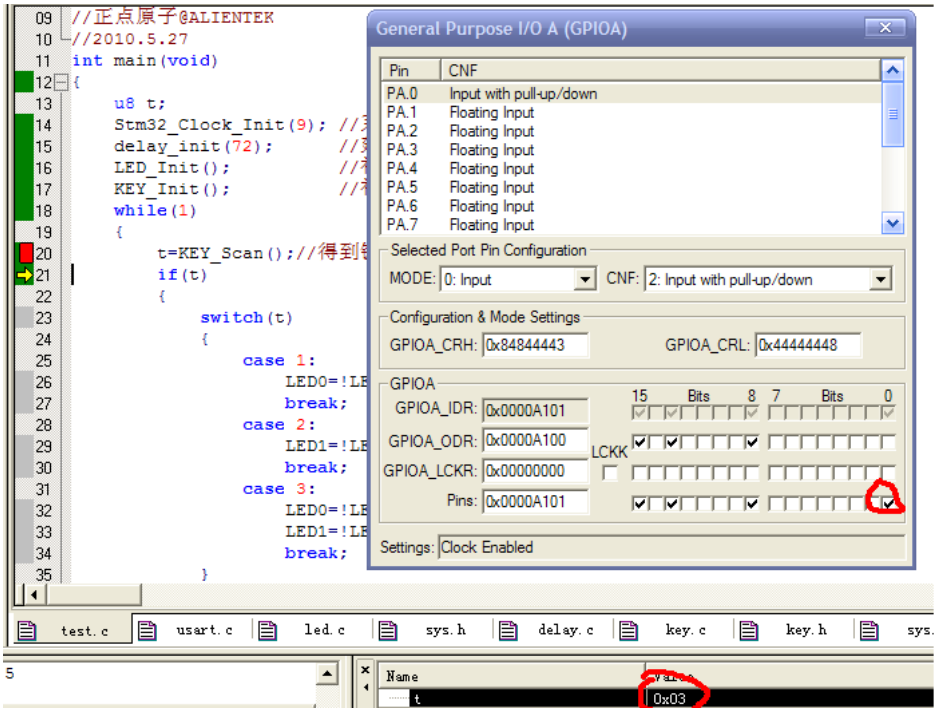


图 3.2.4.4　按键扫描结果

然后按相似的方法，分别勾选 PA13 和 PA15，然后再把它们还原，可以看到逻辑分析窗口的波形如下：

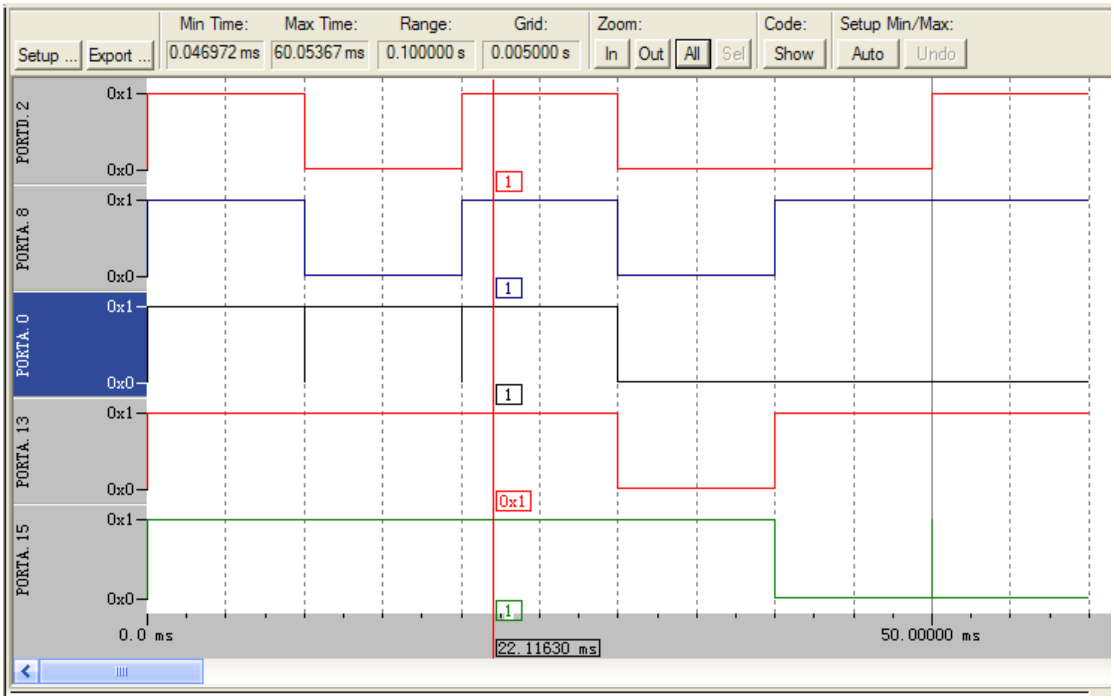


图 3.2.4.5　仿真波形

从上图可以看出，每当 PA0 有按键按下的时候，DS0，DS1 都会翻转一次电平。而 PA13 有按键按下的时候 PA8 就翻转，PA15 按下的时候，PD2 就翻转。和我们之前的设计一摸一样。

因此，可以确定软件仿真基本没有问题了。接下来可以把代码下载到 Mini STM32 开发板上看看运行结果是否正确。

下载完代码之后，如果您想用 JLINK 来调试，那就是不行的了，至于为什么，前面已经说过了，这里我们在下载完代码之后，一定要拔掉 JLINK（如果是用 JLINK 下载的话）。在下载完之后，我们可以按 KEY0、KEY1 和 KEY_UP 来看看 DS0 和 DS1 的变化，是否和我们仿真的结果一致（结果肯定是一致的）。

至此，我们的本实例的学习就结束了。本节，作为 STM32 的入门第二个例子，介绍了 STM32 的 IO 作为输入的使用方法，同时巩固了前面的学习。希望大家在开发板上实际验证一下，从而加深印象。

3.3 串口实验

前面两节介绍了 STM32 的 IO 口操作。这一节我们将学习 STM32 的串口。通过本节的学习，你将了解到 STM32 串口的基本使用方法。本节分为如下几个小节：

3.3.1 STM32 串口简介
3.3.2 硬件设计
3.3.3 软件设计
3.3.4 仿真与下载

3.3.1 STM32 串口简介

前面两节介绍了 STM32 的 IO 口操作。这一节我们将学习 STM32 的串口。作为软件开发重要的调试手段，串口的作用是很大的。在调试的时候可以用来查看和输入相关的信息。在使用的时候，串口也是一个和外设(比如 GPS，GPRS 模块等)通信的重要渠道。

STM32 的串口资源相当丰富的，功能也相当强劲。STM32 最多可提供 5 路串口（ALIENTEK Mini STM32 使用的是 STM32F103RBT6，只有 3 个串口），有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范（仅串口 3 支持）、具有 DMA 等。

在 2.7.3 节对串口有过简单的介绍，接下来我们将从寄存器层面，告诉您如何设置串口，以达到我们最基本的通信功能。本实例中，我们将实现利用串口 1 不停的打印一个信息到电脑上，同时接收从串口发过来的数据，把发送过来的数据直接送回给电脑。

串口最基本的设置，就是波特率的设置。STM32 的串口使用起来还是蛮简单的，只要您开启了串口时钟，并设置相应 IO 口的模式，然后配置一下波特率，数据位长度，奇偶校验位等信息，就可以使用了，详见 2.7.3 节。下面，我们就简单介绍下这几个与串口基本配置直接相关的寄存器。

1，串口时钟使能。串口作为 STM32 的一个外设，其时钟由外设时钟使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。APB2ENR 寄存器在之前已经介绍过了，这里不再介绍。只是说明一点，就是除了串口 1 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR 寄存器。

2，串口复位。当外设出现异常的时候可以通过复位寄存器里面的对应位设置，实现该外设的复位，然后重新配置这个外设达到让其重新工作的目的。一般在系统刚开始配置外设的时候，都会先执行复位该外设的操作。串口 1 的复位是通过配置 APB2RSTR 寄存器的第 14 位来实现的。APB2RSTR 寄存器的各位描述如下图所示：

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
ADC3 RST	USART1 RST	TIM8 RST	SPI1 RST	TIM1 RST	ADC2 RST	ADC1 RST	IOPG RST	IOPF RST	IOPE RST	IOPD RST	IOPC RST	IOPB RST	IOPA RST	保留	AFIO RST
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	res	rw

图 3.3.1.1 APB2RSTR 寄存器各位描述

从图 3.3.1.1 可知串口 1 的复位设置位在 APB2RSTR 的第 14 位。通过向该位写 1 复位串口 1，写 0 结束复位。其他串口的复位位在 APB1RSTR 里面。

3，串口波特率设置。在 2.7.3 节，我们已经介绍过了，每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以达到配置不同波特率的目的。具体实现方法，请参考 5.3.2 节。

4，串口控制。STM32 的每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3 个寄存器来设置的。这里我们只要用到 USART_CR1 就可以实现我们的功能了，该寄存器的各位描述如下图所示：

图 3.3.1.2　USART_CR 寄存器各位描述

该寄存器的高 18 位没有用到，低 14 位用于串口的功能设置。UE 为串口使能位，通过该位置 1，以使能串口。M 为字长选择位，当该位为 0 的时候设置串口为 8 个字长外加 n 个停止位，停止位的个数（n）是根据 USART_CR2 的[13:12]位设置来决定的，默认为 0。PCE 为校验使能位，设置为 0，则禁止校验，否则使能校验。PS 为校验位选择，设置为 0 则为偶校验，否则为奇校验。TXIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_SR 中的 TXE 位为 1 时，将产生串口中断。TCIE 为发送完成中断使能位，设置该位为 1，当 USART_SR 中的 TC 位为 1 时，将产生串口中断。RXNEIE 为接收缓冲区非空中断使能，设置该位为 1，当 USART_SR 中的 ORE 或者 RXNE 位为 1 时，将产生串口中断。TE 为发送使能位，设置为 1，将开启串口的发送功能。RE 为接收使能位，用法同 TE。

其他位的设置，这里就不一一列出来了，大家可以参考《STM32 参考手册》第 542 页有详细介绍，在这里我们就不列出来了。

5，数据发送与接收。STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到收据的时候，也是存在该寄存器内。该寄存器的各位描述如下图所示：

图 3.3.1.3 USART_DR 寄存器各位描述

可以看出，虽然是一个 32 位寄存器，但是只用了低 9 位（DR[8：0]），其他都是保留。

DR[8：0]为串口数据，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。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 种 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。

6，串口状态。串口的状态可以通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如下图所示：

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

图 3.3.1.4　USART_SR 寄存器各位描述

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。

RXNE（读数据寄存器非空），当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TC（发送完成），当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如果设置了这个位的中断，则会产生中断。该位也有两种清零方式：1）读 USART_SR，写 USART_DR。2）直接向该位写 0。

通过以上一些寄存器的操作外加一下 IO 口的配置，我们就可以达到串口最基本的配置了，关于串口更详细的介绍，请参考《STM32 参考手册》第 516 页至 548 页，通用同步异步收发器一章。

3.3.2 硬件设计

该实验的硬件配置不同于前两个实验，串口 1 与 USB 串口默认是分开的，并没有在 PCB 上连接在一起，需要通过跳线帽来连接一下。这里我们把 P4 的 RXD 和 TXD 用跳线帽与 P3 的 PA9 和 PA10 连接起来。如下图所示：

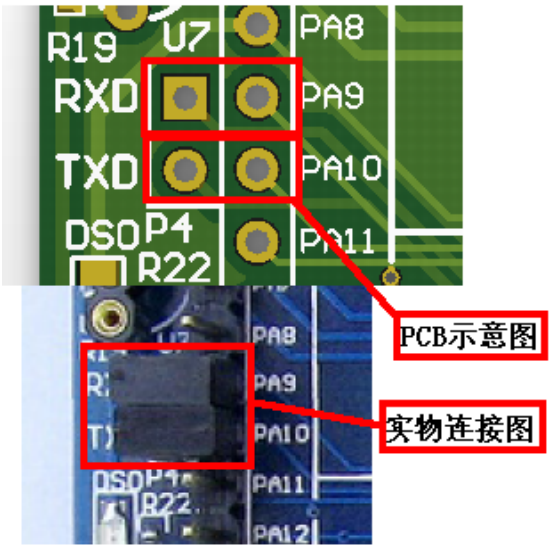


图 3.3.2.1 硬件连接图

连接上这里之后，我们在硬件上就设置完成了，可以开始软件设计了。

3.3.3 软件设计

这里的代码设计，比前两节简单很多，因为我们的串口初始化代码和接收代码就是用我们之前介绍的 SYSTEM 文件夹下的串口部分内容。这里我们对代码部分稍作讲解。

打开 3.2 节的 TEST 工程，然后在 SYSTEM 组下双击 usart.c，我们就可以看到该文件里面的代码，先介绍 uart_init 函数，该函数代码如下：

```
//初始化 IO 串口 1
//pclk2:PCLK2 时钟频率(Mhz)
//bound:波特率
void uart_init(u32 pclk2，u32 bound)
{
    float temp;
    u16 mantissa;
    u16 fraction;
    temp=(float)(pclk2*1000000)/(bound*16);//得到 USARTDIV
    mantissa=temp;                          //得到整数部分
    fraction=(temp-mantissa)*16; //得到小数部分
    mantissa<<=4;
    mantissa+=fraction;
    RCC->APB2ENR|=1<<2;     //使能 PORTA 口时钟
    RCC->APB2ENR|=1<<14;    //使能串口时钟
    GPIOA->CRH=0X444444B4;//IO 状态设置

    RCC->APB2RSTR|=1<<14;     //复位串口 1
    RCC->APB2RSTR&=~(1<<14);//停止复位
    //波特率设置
    USART1->BRR=mantissa; // 波特率设置
    USART1->CR1|=0X200C;  //1 位停止，无校验位.
#ifdef EN_USART1_RX              //如果使能了接收
    //使能接收中断
    USART1->CR1|=1<<8;       //PE 中断使能
    USART1->CR1|=1<<5;       //接收缓冲区非空中断使能
    MY_NVIC_Init(3，3，USART1_IRQChannel，2);//组 2，最低优先级
#endif
}
```

从该代码可以看出，其初始化串口的过程，和我们前面介绍的一致先计算得到 USART1->BRR 的内容。然后开始初始化串口引脚，接着把 USART1 复位，然之后设置波特率和奇偶校验等。

这里需要注意一点，因为我们使用到了串口的中断接收，必须在 usart.h 里面定义 EN_USART1_RX 。该函数才会配置中断使能，以及开启串口 1 的 NVIC 中断。这里我们把串口 1 中断放在组 2，优先级设置为组 2 里面的最低。

再介绍一下串口 1 的中断服务函数 USART1_IRQHandler，该函数的名字不能自己定义了，

MDK 已经给每个中断都分配了一个固定的函数名，我们直接用就可以了。具体这些函数的名字是什么，我们可以在 MDK 提供的例子里面，找到 stm32f10x_it.c，该文件里面包含了 STM32 所有的中断服务函数。USART1_IRQHandler 的代码如下：

```
void USART1_IRQHandler(void)
{
    u8 res;
    if(USART1->SR&(1<<5))//接收到数据
    {
        res=USART1->DR;
        if((USART_RX_STA&0x80)==0)//接收未完成
        {
            if(USART_RX_STA&0x40)//接收到了 0x0d
            {
                if(res!=0x0a)USART_RX_STA=0;//接收错误，重新开始
                else USART_RX_STA|=0x80;    //接收完成了
            }else //还没收到 0X0D
            {
                if(res==0x0d)USART_RX_STA|=0x40;
                else
                {
                    USART_RX_BUF[USART_RX_STA&0X3F]=res;
                    USART_RX_STA++;
                    if(USART_RX_STA>63)USART_RX_STA=0;//接收数据错误，重新开始接收
                }
            }
        }
    }
}
```

该函数的重点就是判断接收是否完成，通过检测是否收到 0X0D、0X0A 的连续 2 个字节（0X0D 后跟 0X0A 表示回车键）来检测是否结束。当检测到这个结束序列之后，就会置位 USART_RX_STA 的最高为来标记已经收到了一次数据。之后等待外部函数清空该位之后才开始第二次接收。所接收的数据全部存放在 USART_RX_BUF 里面，一次接收数据不能超过 64 个字节，否则被丢弃。

介绍完了这两个函数，我们回到 test.c，在 test.c 里面编写如下代码：

```
#include <stm32f10x_lib.h>
#include "sys.h"
#include "usart.h"
#include "delay.h"
#include "led.h"
#include "key.h"
//Mini STM32 开发板范例代码 3
//串口实验
```

```
//正点原子@ALIENTEK
//2010.5.28
int main(void)
{
    u8 t;
    u8 len;
    u16 times=0;
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72，9600);  //串口初始化为 9600
    LED_Init();                 //初始化与 LED 连接的硬件接口
    while(1)
    {
        if(USART_RX_STA&0x80)
        {
            len=USART_RX_STA&0x3f;//得到此次接收到的数据长度
            printf("\n 您发送的消息为:\n");
            for(t=0;t<len;t++)
            {
                USART1->DR=USART_RX_BUF[t];
                while((USART1->SR&0X40)==0);//等待发送结束
            }
            printf("\n\n");//插入换行
            USART_RX_STA=0;
        }else
        {
            times++;
            if(times%5000==0)
            {
                printf("\nMiniSTM32 开发板  串口实验\n");
                printf("正点原子@ALIENTEK\n\n\n");
            }
            if(times%200==0)printf("请输入数据，以回车键结束\n");
            if(times%30==0)LED0=!LED0;//闪烁 LED，提示系统正在运行.
            delay_ms(10);
        }
    }
}
```

这段代码比较简单，重点看下以下两句：

```
USART1->DR=USART_RX_BUF[t];
while((USART1->SR&0X40)==0);//等待发送结束
```

第一句，其实就是发送一个字节到串口，通过直接操作寄存器来实现的。第二句呢，就是我们在写了一个字节在 USART1->DR 之后，要检测这个数据是否已经被发送完成了，通过检测

USART1->SR 的第 6 位，是否为 1 来决定是否可以开始第二个字节的发送。

其他的代码比较简单，我们执行编译之后看看有没有错误，没有错误就可以开始仿真与调试了。整个工程的编译结果如下：

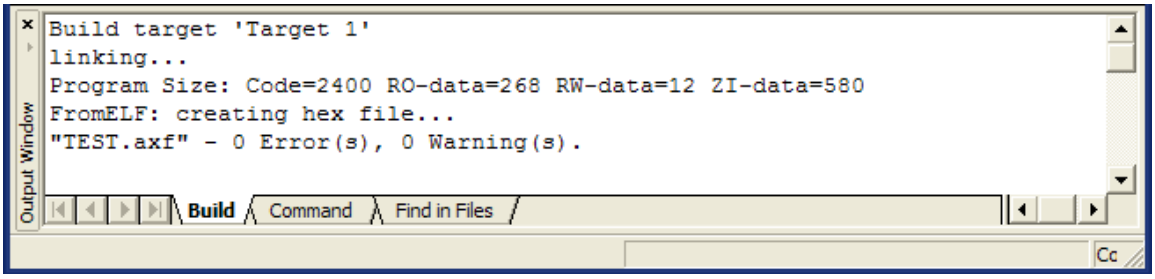


图 3.3.3.1 编译结果

可以看到，这里我们的代码比上一节的多了 800 多字节，其实这里主要是使用了 printf 函数的缘故，所以如果有时候你程序太大了，可以通过不使用 printf 函数，来给其他代码腾出一部分空间。

3.3.4 仿真与下载

前面 2 接已经重点介绍了仿真，仿真的基本技巧也差不多介绍完了，接下来我们将淡化这部分，因为代码都是经过作者实际检测，并且在开发板上验证了的，有兴趣的大家可以自己仿真看看。但是这里要说明几点：

对于串口仿真，MDK3.80 的仿真串口窗口的显示貌似有 BUG，因为实际硬件上是没问题，但是仿真的时候很少有机会能打印出来（有时候又可以有时候不行，希望 MDK 的后续版本能得到改正）。

IO 口复用的，信号在逻辑分析窗口是不能显示出来的，这一点也请大家注意。比如串口的输出，SPI，USB，CAN 等。你在仿真的时候在该窗口看不到任何信息。遇到这样的情况，你就不得不准备一个逻辑分析仪，外加一个 ULINK 或者 JLINK 来做在线调试。但一般情况，这些都是有现成的例子，不用这几个东西一般也能编出来。

仿真并不能代表实际情况。只能从某些方面给你一些启示，告诉你大方向，不能尽信仿真，当然也不能完全没有仿真。比如上面 IO 口的输出，仿真的时候，其翻转速度可以达到很快，但是实际上 STM32 的 IO 输出就达不到这个速度。

总之，我们要合理的利用仿真，也不能过于依赖仿真。当仿真解决不了了，可以试试在线调试，在线调试一般都可以知道问题在哪个地方，但是问题要怎么解决还是得各位自己动脑筋、找资料了。

我们把代码下载到 MiniSTM32 开发板，可以看到板子上的 DS0 开始闪烁，说明程序已经在跑了。接着我们打开串口调试助手，看到如下信息：

图 3.3.4.1 串口调试助手收到的信息

证明串口数据发送没问题。接着，我们在发送区输入上面的文字，输入完后按回车键。然后单击发送，可以得到如下结果：

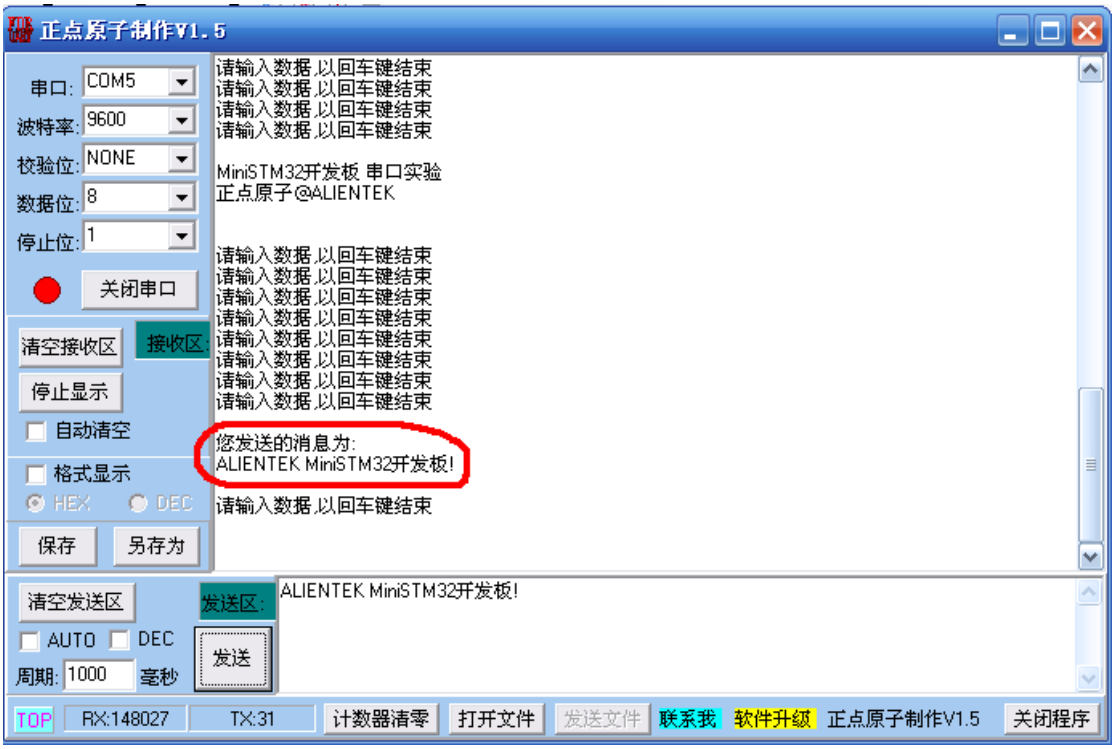


图 3.3.4.2 发送数据后收到的数据

可以看到，我们发送的消息被发送回来了（图中红圈内）。各位可以试试，如果不输入回车键，直接按发送是什么结果。

3.4 外部中断实验

这一节，我们将向大家介绍如何使用 STM32 的外部输入中断。通过第 1、2 节的学习，我们掌握了 STM32 的 IO 口最基本的操作。这节我们将介绍作为外部中断输入口，STM32 的需要做哪些设置。本节分为如下几个部分：

3.4.1 STM32 外部中断简介

3.4.2 硬件设计

3.4.3 软件设计

3.4.4 下载与测试

3.4.1 STM32 外部中断简介

STM32 的 IO 口在本章第一节有详细介绍，而外部中断在第二章也有详细的阐述。这里我们将介绍如何将这两者结合起来，实现外部中断输入。

STM32 的每个 IO 口都可以作为中断输入，这点很好用。要把 IO 口作为外部中断输入，有以下几个步骤：

1）初始化 IO 口为输入。

这一步设置你要作为外部中断输入的 IO 口的状态，可以设置为上拉/下拉输入，也可以设置为浮空输入，但浮空的时候外部一定要带上拉，或者下拉电阻。否则可能导致中断不停的触发。在干扰较大的地方，就算使用了上拉/下拉，也建议使用外部上拉/下拉电阻，这样可以一定程度防止外部干扰带来的影响。

2）开启 IO 口复用时钟，设置 IO 口与中断线的映射关系。

STM32 的 IO 口与中断线的对应关系需要配置外部中断配置寄存器 EXTICR，这样我们要先开启复用时钟，然后配置 IO 口与中断线的对应关系。才能把外部中断与中断线连接起来。

3）开启与该 IO 口相对的线上中断/事件，设置触发条件。

这一步，我们要配置中断产生的条件，STM32 可以配置成上升沿触发，下降沿触发，或者任意电平变化触发，但是不能配置成高电平触发和低电平触发。这里根据自己的实际情况来配置。同时要开启中断线上的中断，这里需要注意的是：如果使用外部中断，并设置该中断的 EMR 位的话，会引起软件仿真不能跳到中断，而硬件上是可以的。而不设置 EMR，软件仿真就可以进入中断服务函数，并且硬件上也是可以的。建议不要配置 EMR 位。

4）配置中断分组（NVIC），并使能中断。

这一步，我们就是配置中断的分组，以及使能，对 STM32 的中断来说，只有配置了 NVIC 的设置，并开启才能被执行，否则是不会执行到中断服务函数里面去的。关于 NVIC 的详细介绍，请参考前面章节。

5）编写中断服务函数。

这是中断设置的最后一步，中断服务函数，是必不可少的，如果在代码里面开启了中断，但是没编写中断服务函数，就可能引起硬件错误，从而导致程序崩溃！所以在开启了某个中断后，一定要记得为该中断编写服务函数。在中断服务函数里面编写你要执行的中断后的操作。

通过以上几个步骤的设置，我们就可以正常使用外部中断了。

这一节，我们将实现同第二节差不多的功能，但是这里我们使用的是中断来检测按键，还是通过 WK_UP 按键实现按一次 DS0 和 DS1 同时翻转，按 KEY0 翻转 DS0，按 KEY1 翻转 DS1。

3.4.2 硬件设计

这里的硬件电路和第二节一模一样，不再多做介绍了。

3.4.3 软件设计

软件设计我们还是在之前的工程上面增加，首先在 HARDWARE 文件夹下新建 EXTI 的文

件夹。然后打开 USER 文件夹下的工程，新建一个 exti.c 的文件和 exti.h 的头文件，保存在 EXTI 文件夹下，并将 EXTI 文件夹加入头文件包含路径（即设定编译器包含路径，第二章已有介绍。以下类似）。

我们在 exit.c 里输入如下代码：

```
#include "exti.h"
#include "led.h"
#include "key.h"
#include "delay.h"
#include "usart.h"
//ALIENTEK Mini STM32 开发板
//外部中断  驱动代码
//外部中断 0 服务程序
void EXTI0_IRQHandler(void)
{
    delay_ms(10);//消抖
    if(KEY2==1)    //按键 2
    {
        LED0=!LED0;
        LED1=!LED1;
    }
    EXTI->PR=1<<0;    //清除 LINE0 上的中断标志位
}

//外部中断 15~10 服务程序
void EXTI15_10_IRQHandler(void)
{
    delay_ms(10);       //消抖
    if(KEY0==0)          //按键 0
    {
        LED0=!LED0;
    }else if(KEY1==0)//按键 1
    {
        LED1=!LED1;
    }
    EXTI->PR=1<<13;         //清除 LINE13 上的中断标志位
    EXTI->PR=1<<15;         //清除 LINE15 上的中断标志位
}
//外部中断初始化程序
//初始化 PA0,PA13,PA15 为中断输入.
void EXTIX_Init(void)
{
    RCC->APB2ENR|=1<<2;         //使能 PORTA 时钟
    JTAG_Set(JTAG_SWD_DISABLE);//关闭 JTAG 和 SWD
```

```
    GPIOA->CRL&=0XFFFFFFF0;//PA0 设置成输入
    GPIOA->CRL|=0X00000008;
    GPIOA->CRH&=0X0F0FFFFF;//PA13,15 设置成输入
    GPIOA->CRH|=0X80800000;
    GPIOA->ODR|=1<<13;      //PA13 上拉,PA0 默认下拉
    GPIOA->ODR|=1<<15;      //PA15 上拉

    Ex_NVIC_Config(GPIO_A,0,RTIR); //上升沿触发
    Ex_NVIC_Config(GPIO_A,13,FTIR);//下降沿触发
    Ex_NVIC_Config(GPIO_A,15,FTIR);//下降沿触发

    MY_NVIC_Init(2,2,EXTI0_IRQChannel,2);      //抢占 2，子优先级 2，组 2
    MY_NVIC_Init(2,1,EXTI15_10_IRQChannel,2);//抢占 2，子优先级 1，组 2
}
```

exit.c 文件总共包含 3 个函数。一个是外部中断初始化函数 void EXTIX_Init(void)，另外两个都是中断服务函数。void EXTI0_IRQHandler(void)是外部中断 0 的服务函数，负责 WK_UP 按键的中断检测。void EXTI15_10_IRQHandler(void)是外部中断 10~15 的中断服务函数，这里我们是用了中断 13 和 15，这两个中断共用一个中断服务函数。下面我们分别介绍这几个函数。

首先是外部中断初始化函数 void EXTIX_Init(void)，该函数严格按照我们之前的步骤来初始化外部中断，这里有个关闭 JTAG 和 SWD 的操作，和 3.2 节的不完全一样，这里因为用的是中断，而这几个中断和 JLINK 的 SWD 或 JTAG 接口是复用的，所以不能开启 SWD 了。在下载代码之后，必须拔掉 JLINK 才能得到正确的结果，而且，此程序只有第一次能用 JTAG 下载，此后除非设置为 ISP 模式，否则就得用串口下载该程序了。

这里面调用了两个函数 Ex_NVIC_Config 和 MY_NVIC_Init，其作用在第二章已经介绍了，有不明白的可以翻到前面看看，这里不再多说。需要说明的是因为我们的 WK_UP 按键是高电平有效的，KEY0 和 KEY1 是低电平有效的，所以我们设置 WK_UP 按键下拉输入，而 KEY0 和 KEY1 设置成上拉输入。当中断触发的时候，在 WK_UP 上会检测到上升沿，而 KEY0 和 KEY1 会产生下降沿。这里我们把所有中断都分配到第二组，把按键的抢占优先级设置成一样，而子优先级不同，KEY0 和 KEY1 的子优先级大于 WK_UP。

接下来我们介绍两个中断服务函数。先看 WK_UP 的中断服务函数 void EXTI0_IRQHandler(void)，该函数代码比较简单，先延时 10ms 以消抖，再检测 KEY2(WK_UP)是否还是为高电平，如果是，则执行此次操作（翻转 DS0 和 DS1），如果不是，则直接跳过，在最后有一句 EXTI->PR=1<<0;通过该句清除已经发生的中断请求。再看 KEY0 和 KEY1 的中断服务函数 void EXTI15_10_IRQHandler(void)，该函数和 KEY2 的中断服务函数有点区别，从函数名就可以看出是给中断线 10~15 服务的，也就是多个中断线上的中断共用一个中断服务函数。在该函数里面我们先对进入中断的信号进行区分（通过中断输入 IO 口上的电平判断），再分别进行处理。最后也是通过向 EXTI->PR 的对应位写 1 清除中断线上的中断请求。

我们将 exti.c 文件保存，然后加入到 HARDWARE 组下。在 exti.h 文件里面，我们输入如下代码：

```
#ifndef __EXTI_H
#define __EXIT_H
void EXTIX_Init(void);//IO 初始化
```

```
#endif
```

这部分代码就很简单了，我们这里不多废话，保存就可以了。接着我们在 test.c 里面写入如下内容：

```
#include <stm32f10x_lib.h>
#include "sys.h"
#include "usart.h"
#include "delay.h"
#include "led.h"
#include "key.h"
#include "exti.h"
//Mini STM32 开发板范例代码 4
//外部中断实验
//正点原子@ALIENTEK
//2010.5.27
int main(void)
{
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);         //延时初始化
    uart_init(72，9600);   //串口初始化
    LED_Init();                  //初始化与 LED 连接的硬件接口
    EXTIX_Init();         //外部中断初始化
    while(1)
    {
        printf("OK\n");
        delay_ms(1000);
    }
}
```

该部分代码很简单，在初始化完中断后，就进入死循环等待了，这里死循环里面通过一个 printf 函数来告诉我们系统正在运行。其他在在这里不再多说。

3.4.4 下载与测试

在编译成功之后，我们就可以下载代码到 MiniSTM32 开发板上，实际验证一下，我们的程序是否正确。下载代码后，可以通过按下 KEY0、KEY1、KEY2 来观看 DS0、DS1 是否跟着按键的变化而变化。

3.5 独立看门狗（IWDG）实验

这一节，我们将向大家介绍如何使用 STM32 的独立看门狗（以下简称 IWDG）。STM32 内部自带了 2 个看门狗：独立看门狗（IWDG）和窗口看门狗（WWDG）。这一节我们只介绍独立看门狗，窗口看门狗将在下一节介绍。本节分为如下几个部分：

3.5.1 STM32 独立看门狗简介

3.5.2 硬件设计

3.5.3 软件设计

3.5.4 下载与测试

3.5.1 STM32 独立看门狗简介

STM32 的独立看门狗由内部专门的 40Khz 低速时钟驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟并不是准确的 40Khz，而是在 30~60Khz 之间变化的一个时钟，只是我们在估算的时候，以 40Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

独立看门狗有几个寄存器与我们这节相关，我们分别介绍这几个寄存器，首先是键值寄存器 IWDG_KR，该寄存器的各位描述如下：

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

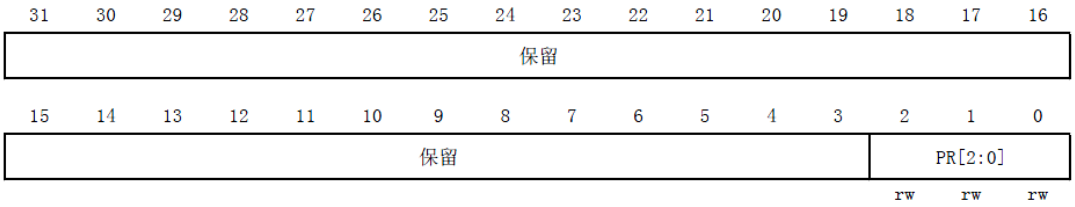
位31:16	保留，始终读为0。
位15:0	**KEY[15:0]**: 键值(只写寄存器，读出值为0x0000) 软件必须以一定的间隔写入0xAAAA，否则，当计数器为0时，看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。 写入0xCCCC，启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。

图 3.5.1.1 寄存器 IWDG_KR 各位描述

在键寄存器(IWDG_KR)中写入 0xCCCC，开始启用独立看门狗；此时计数器开始从其复位值 0xFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号(IWDG_RESET)。 无论何时，只要键寄存器 IWDG_KR 中被写入 0xAAAA， IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位 。

IWDG_PR 和 IWDG_RLR 寄存器具有写保护功能。 要修改这两个寄存器的值，必须先向 IWDG_KR 寄存器中写入 0x5555。以不同的值写入这个寄存器将会打乱操作顺序，寄存器将重新被保护。重装载操作(即写入 0xAAAA)也会启动写保护功能。

接下来，我们介绍预分频寄存器（IWDG_PR），该寄存器用来设置看门狗时钟的分频系数，最低为 4，最高位 256，该寄存器是一个 32 位的寄存器，但是我们只用了最低 3 位，其他都是保留位。预分频寄存器各位定义如下：



位31:3	保留，始终读为0。
位2:0	**PR[2:0]**: 预分频因子 这些位具有写保护设置，上面已有介绍。 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。

图 3.5.1.2 寄存器 IWDG_ PR 各位描述

在介绍完 IWDG_PR 之后，我们介绍一下重装载寄存器。该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器，但是只有低 12 位是有效的，该寄存器的各位描述如下：

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 这些位具有写保护功能，前面已有介绍。 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。 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。

图 3.5.1.3 重装载寄存器各位描述

只要对以上三个寄存器进行相应的设置，我们就可以启动 STM32 的独立看门狗，启动过程可以按如下步骤实现：

1）向 IWDG_KR 写入 0X5555。

通过这步，我们取消 IWDG_PR 和 IWDG_RLR 的写保护，使后面可以操作这两个寄存器。

设置 IWDG_PR 和 IWDG_RLR 的值。

这两步设置看门狗的分频系数，和重装载的值。由此，就可以知道看门狗的喂狗时间（也就是看门狗溢出时间），该时间的计算方式为：

$$Tout=((4\times 2^{prer})\times rlr)/40$$

其中 Tout 为看门狗溢出时间（单位为 ms）；prer 为看门狗时钟预分频值（IWDG_PR 值），范围为 0~7；rlr 为看门狗的重装载值（IWDG_RLR 的值）；

比如我们设定 prer 值为 4，rlr 值为 625，那么就可以得到 Tout=64×625/40=1000ms，这样，看门狗的溢出时间就是 1s，只要你在一秒钟之内，有一次写入 0XAAAA 到 IWDG_KR，就不

会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 40Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

2）向 IWDG_KR 写入 0XAAAA。

通过这句，将使 STM32 重新加载 IWDG_RLR 的值到看门狗计数器里面。也可以用该命令来喂狗。

3）向 IWDG_KR 写入 0XCCCC。

通过这句，来启动 STM32 的看门狗。注意 IWDG 在一旦启用，就不能再被关闭！想要关闭，只能重启，并且重启之后不能打开 IWDG，否则问题依旧，所以在这里提醒大家，如果不用 IWDG 的话，就不要去打开它，免得麻烦。

通过上面 3 个步骤，我们就可以启动 STM32 的看门狗了，使能了看门狗，在程序里面就必须间隔一定时间喂狗，否则将导致程序复位。利用这一点，我们本章将通过一个 LED 灯来指示程序是否重启，来验证 STM32 的独立看门狗。

在配置看门狗后，DS0 将常亮，如果 WK_UP 按键按下，就喂狗，只要 WK_UP 不停的按，看门狗就一直不会产生复位，保持 DS0 的常亮，一旦超过看门狗定溢出时间（Tout）还没按，那么将会导致程序重启，这将导致 DS0 熄灭一次。

3.5.2 硬件设计

这里的核心是在 STM32 内部进行，并不需要外部电路。但是考虑到指示当前状态和喂狗等操作，我们需要 2 个 IO 口，一个用来输入喂狗信号，另外一个用来指示程序是否重启。喂狗我们采用板上的 WK_UP 键来操作，而程序重启，则是通过 DS0 来指示的。DS0 和 WK_UP 的连接在前面已经介绍了，这里我们不再多说。

3.5.3 软件设计

软件设计我们依旧是在前面的代码基础上往上加，先在 HARDWARE 文件夹下面新建一个 WDG 的文件夹，用来保存与看门狗相关的代码。再打开工程，新建 wdg.c 和 wdg.h 两个文件，并保存在 WDG 文件夹下，并将 WDG 文件夹加入头文件包含路径。

在 wdg.c 里面输入如下代码：

```
#include "wdg.h"
//ALIENTEK Mini STM32 开发板
//看门狗 驱动代码
//初始化独立看门狗
//prer:分频数:0~7(只有低 3 位有效!)
//分频因子=4*2^prer.但最大值只能是 256!
//rlr:重装载寄存器值:低 11 位有效.
//时间计算(大概):Tout=((4×2^prer) ×rlr)/40 (ms).
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG->KR=0X5555;//使能对 IWDG->PR 和 IWDG->RLR 的写
```

```
    IWDG->PR=prer;     //设置分频系数
    IWDG->RLR=rlr;   //从加载寄存器 IWDG->RLR
    IWDG->KR=0XAAAA;//reload
    IWDG->KR=0XCCCC;//使能看门狗
}//喂独立看门狗
void IWDG_Feed(void)
{
    IWDG->KR=0XAAAA;//reload
}
```

该代码就 2 个函数，void IWDG_Init(u8 prer，u16 rlr)是独立看门狗初始化函数，就是按照上面介绍的步骤来初始化独立看门狗的。该函数有 2 个参数，分别用来设置与预分频数与重装寄存器的值的。通过这两个参数，就可以大概知道看门狗复位的时间周期为多少了。其计算方式上面有详细的介绍，这里不再多说了。

void IWDG_Feed(void)函数，该函数用来喂狗，因为 STM32 的喂狗只需要向键值寄存器写入 0XAAAA 即可，所以，我们这个函数也是简单的很。保存 wdg.c，然后把该文件加入到 HARDWARE 组下。

在 wdg.h 里面，我们输入如下内容：

```
#ifndef __WDG_H
#define __WDG_H
#include "sys.h"
//Mini STM32 开发板
//看门狗 驱动代码
//正点原子@ALIENTEK
//2010/5/30

void IWDG_Init(u8 prer，u16 rlr);
void IWDG_Feed(void);
#endif
```

这个很简单，估计大家都能理解，不再多说了。我们保存下这两个文件，来看主程序该如何写，在主程序里面我们先初始化一下系统代码，然后启动按键输入和看门狗，在看门狗开启后马山点亮 LED0，并进入死循环等待按键的输入，一旦 WK_UP 有按键，则喂狗，佛否则等待 IWDG 复位的到来。该部分代码如下：

```
int main(void)
{
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);        //延时初始化
    uart_init(72，9600);   //串口初始化
    LED_Init();                  //初始化与 LED 连接的硬件接口
    KEY_Init();                 //按键初始化
    delay_ms(300);        //让人看得到灭
    IWDG_Init(4，625);      //与分频数为 64，重载值为 625，溢出时间为 1s
    LED0=0;                     //点亮 LED0
    while(1)
```

```
    {
        if(KEY_Scan()==3)IWDG_Feed();//如果 WK_UP 按下，则喂狗
    };
}
```

至此，独立看门狗的实验代码，我们就全部编写完了，接着要做的就是下载与测试了，看看我们的代码是否真的正确，当然在下载之前可以通过软件仿真看看是否可行。

3.5.4 下载与测试

在编译成功之后，我们就可以下载代码到 MiniSTM32 开发板上，实际验证一下，我们的程序是否正确。下载代码后，看到 DS0 不停的闪烁，证明程序在不停的复位，否则只会 DS0 常量。这时我们试试不停的按 WK_UP 按键，可以看到 DS0 就常亮了，不会在闪烁。说明我们的实验是成功的。

3.6 窗口门狗（WWDG）实验

这一节，我们将向大家介绍如何使用 STM32 的另外一个看门狗，窗口看门狗（以下简称 WWDG）。上一节，已经介绍了独立看门狗，这一节我们将介绍另外一个 STM32 的看门狗，窗口看门狗。本节分为如下几个部分：

3.6.1 STM32 窗口看门狗简介

3.6.2 硬件设计

3.6.3 软件设计

3.6.4 下载与测试

3.6.1 STM32 窗口看门狗简介

窗口看门狗（WWDG）通常被用来监测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位（WWDG->CR 的第六位）变成 0 前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口配置寄存器(WWDG->CFR)数值之前，如果 7 位的递减计数器数值(在控制寄存器中)被刷新， 那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。他们的关系可以用图 11.1 来说明：

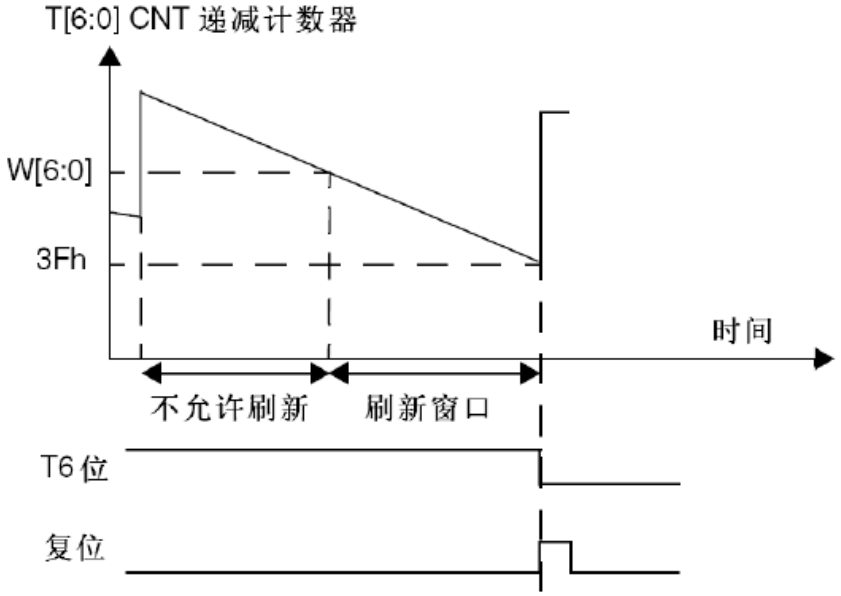


图 3.6.1.1 窗口看门狗工作示意图

图 3.6.1.1 中，T[6:0]就是 WWDG_CR 的低七位，W[6:0]即是 WWDG->CFR 的低七位。T[6:0]就是窗口看门狗的计数器，而 W[6:0]则是窗口看门狗的上窗口，下窗口值是固定的（0X40）。当窗口看门狗的计数器在上窗口值之外被刷新，或者低于下窗口值都会产生复位。

上窗口值（W[6:0]）是由用户自己设定的，根据实际要求来设计窗口值，但是一定要确保窗口值大于 0X40，否则窗口就不存在了。

窗口看门狗的超时公式如下：

Twwdg=(4096×2^WDGTB×(T[5：0]+1)) /Fpclk1;

其中：

Twwdg：WWDG 超时时间（单位为 ms）

Fpclk1：APB1 的时钟频率（单位为 Khz）

WDGTB：WWDG 的预分频系数

T[5:0]：窗口看门狗的计数器低 6 位

根据上面的公式，假设 Fpclk1=36Mhz，那么可以得到最小-最大超时时间表如表 3.6.1.1 所示：

WDGTB	最小超时值	最大超时值
0	113μs	7.28ms
1	227μs	14.56ms
2	455μs	29.12ms
3	910μs	58.25ms

表 3.6.1.1 36M 时钟下窗口看门狗的最小最大超时表

接下来，我们介绍窗口看门狗的 3 个寄存器。首先介绍控制寄存器（WWDG_CR），该寄存器的各位描述如下：

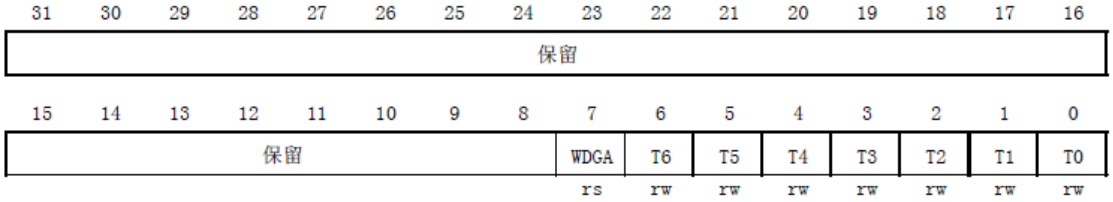
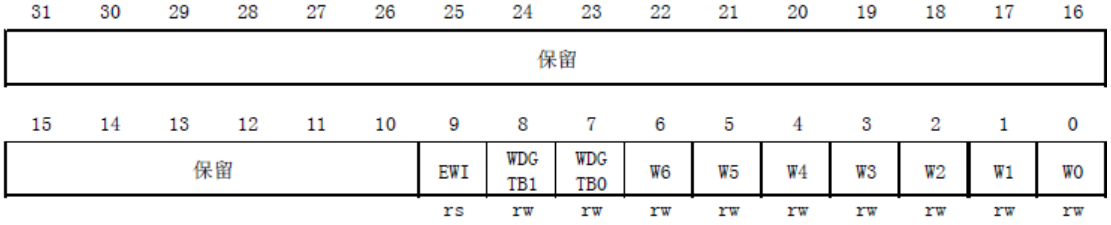


图 3.6.1.2 寄存器 WWDG_CR 各位描述

可以看出，这里我们的 WWDG_CR 只有低八位有效，T[6:0]用来存储看门狗的计数器值，随时更新的，每个 PCLK1 周期（4096×2^wdgtb）减 1。当该计数器的值从 0X40 变为 0X3F 的时候，将产生看门狗复位。

WDGA 位则是看门狗的激活位，该位由软件置 1，以启动看门狗，并且一定要注意的是该位一旦设置，就只能在硬件复位后才能清零了。

窗口看门狗的第二个寄存器是配置寄存器（WWDG_CFR），该寄存器的各位及其描述如下：



位31:8	保留。
位9	**EWI**: 提前唤醒中断 此位若置1，则当计数器值达到40h，即产生中断。 此中断只能由硬件在复位后清除。
位8:7	**WDGTB[1:0]**: 时基 预分频器的时基可根据如下修改： 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8
位6:0	**W[6:0]**: 7位窗口值 这些位包含了用来与递减计数器进行比较用的窗口值。

图 3.6.1.3 寄存器 WWDG_ CFR 各位描述

该位中的 EWI 是提前唤醒中断，也就是在快要产生复位的前一段时间来提醒我们，需要进行喂狗了，否则将复位！因此，我们一般用该位来设置中断，当窗口看门狗的计数器值减到 0X40 的时候，如果该位设置，并开启了中断，则会产生中断，我们可以在中断里面想 WWDG_CR 重新写入计数器的值，来达到喂狗的目的。注意这里你在进入中断后，你得在不大于 113us 的时间（PCLK1 为 36M 的条件下）内重新写 WWDG_CR，否则，看门狗将产生复位！

最后我们要介绍的是状态寄存器（WWDG_SR），该寄存器用来记录当前是否有提前唤醒的标志。该寄存器仅有位 0 有效，其他都是保留位。当计数器值达到 40h 时，此位由硬件置 1。它必须通过软件写’0’来清除。对此位写’1’无效。若中断未被使能，此位也会被置’1’。

在介绍完了窗口看门狗的寄存器之后，我们介绍要如何启用 STM32 的窗口看门狗。这里我们介绍的方法是用中断的方式来喂狗的。采取的步骤如下：

1）使能 WWDG 时钟

WWDG 不同于 IWDG，IWDG 有自己独立的 40Khz 时钟，不存在使能问题。而 WWDG 使用的是 PCLK1 的时钟，需要先使能时钟。

2）设置 WWDG_CFR 和 WWDG_CR 两个寄存器

在时钟使能完后，我们设置 WWDG 的 CFR 和 CR 两个寄存器，对 WWDG 进行配置。包括使能窗口看门狗、开启中断、设置计数器的初始值、设置窗口值并设置分频数 WDGTB 等。

3）开启 WWDG 中断并分组

在设置完了 WWDG 后，需要配置该中断的分组及使能。这点通过我们之前所编写的 MY_NVIC_Init 函数实现就可以了。

4）编写中断服务函数

在最后，还是要编写窗口看门狗的中断服务函数，通过该函数来喂狗，喂狗要快，否则当窗口看门狗计数器值减到 0X3F 的时候，就会引起软复位了。在终端服务函数里面也要将状态寄存器的 EWIF 位清空。

完成了以上 4 个步骤之后，我们就可以使用 STM32 的窗口看门狗了。这一节的实验，我们将通过 LED 来指示 STM32 是否被复位了。以及中断喂狗操作有没有进行。这里我们用 DS0 来指示 STM32 是否被复位了，如果被复位了就会点亮 300ms。DS1 用来指示中断喂狗，每次中断喂狗翻转一次。

3.6.2 硬件设计

窗口看门狗的操作本身和外部没什么关系，但是考虑到指示 STM32 的状态，使用了 2 个外部的 LED，DS0 和 DS1。其他的没有什么电路。

3.6.3 软件设计

这里，我们在之前的 IWDG 看门狗实例内增添部分代码来实现我们此实验。首先打开上次的工程，然后在 wdg.c 加入如下代码（之前代码保留）：

```
//保存 WWDG 计数器的设置值，默认为最大.
u8 WWDG_CNT=0x7f;
//初始化窗口看门狗
//tr   ：T[6：0]，用于存储计数器的值
//wr   ：W[6：0]，用于存储窗口值
//fprer：窗口看门狗的实际设置
//低 2 位有效.Fwwdg=PCLK1/4096/2^fprer.
void WWDG_Init(u8 tr，u8 wr，u8 fprer)
```

```
{
    RCC->APB1ENR|=1<<11;   //使能 wwdg 时钟
    WWDG_CNT=tr&WWDG_CNT;    //初始化 WWDG_CNT.
    WWDG->CFR|=fprer<<7;      //PCLK1/4096 再除 2^fprer
    WWDG->CFR|=1<<9;          //使能中断
    WWDG->CFR&=0XFF80;
    WWDG->CFR|=wr;                  //设定窗口值
    WWDG->CR|=WWDG_CNT|(1<<7); //开启看门狗，设置 7 位计数器
    MY_NVIC_Init(2，3，WWDG_IRQChannel，2);//抢占 2，子优先级 3，组 2
}
//重设置 WWDG 计数器的值
void WWDG_Set_Counter(u8 cnt)
{
    WWDG->CR|=(cnt&0x7F);//重设置 7 位计数器
}
//窗口看门狗中断服务程序
void WWDG_IRQHandler(void)
{
    u8 wr，tr;
    wr=WWDG->CFR&0X7F;
    tr=WWDG->CR&0X7F;
    if(tr<wr)WWDG_Set_Counter(WWDG_CNT);//只有当计数器的值，小于窗口寄存器的值才能写 CR!!
    WWDG->SR=0X00;//清除提前唤醒中断标志位
    LED1=!LED1;
}
```

新增的这三个函数都比较简单，第一个函数 void WWDG_Init(u8 tr，u8 wr，u8 fprer)用来设置 WWDG 的初始化值。包括看门狗计数器的值和看门狗比较值等。该函数就是按照我们上面的 4 个思路设计出来的代码。注意到这里有个全局变量 WWDG_CNT，该变量用来保存最初设置 WWDG_CR 计数器的值。在后续的中断服务函数里面，就又把该数值放回到 WWDG_CR 上。

WWDG_Set_Counter 函数比较简单，就是用来复位窗口看门狗的计数器值的。该函数很简单，我们就不多说了。

最后中断服务函数里面，一定要先比较窗口计数器的值是否小于看门狗的窗口值，如果不小于，则不要修改，如果小于，才能进行修改。我们通过 DS0 不停的取反，来观测中断服务函数的执行了状况。我们再把这几个函数名加入到头函数里面去，以方便其他文件调用。

在完成了以上部分之后，我们就回到主函数，输入如下代码：

```
int main(void)
{
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);         //延时初始化
    uart_init(72，9600);   //串口初始化
    LED_Init();                  //初始化与 LED 连接的硬件接口
```

```
        LED0=0;
        delay_ms(300);
        WWDG_Init(0X7F，0X5F，3);//计数器值为 7f，窗口寄存器为 5f，分频数为 8
        while(1)
        {
            LED0=1;
        }
    }
```

该函数通过 LED0 来指示是否正在初始化。而 LED1 用来指示是否发生了中断。我们先让 LED0 亮 300ms，然后关闭以提示有一次复位发生了。在初始化 WWDG 之后，我们回到死循环，关闭 LED1，并等待看门狗中断的触发/复位。

编译该代码，直到编译通过为止。在没有错误了之后，我们就可以下载到 MiniSTM32 开发板上，看看是否结果与预期一致。

3.6.4 下载与测试

将代码下载到 MiniSTM32 后，可以看到 DS0 亮一下之后熄灭，紧接着 DS1 开始不停的闪烁。每秒钟闪烁 5 次左右，和我们预期的一致。不过注意，下载完后记得把 B0 设置为 0！

有兴趣的大家，可以在窗口看门狗的中断服务函数里面把下面这句注释掉：

```
if(tr<wr)WWDG_Set_Counter(WWDG_CNT);
```

之后再次编译并下载代码到 MiniSTM32 开发板上，可以看看是什么现象？

3.7 定时器中断实验

这一节，我们将向大家介绍如何使用 STM32 的通用定时器，STM32 的定时器功能十分强大，有 TIME1 和 TIME8 等高级定时器，也有 TIME2~TIME5 等通用定时器，还有 TIME6 和 TIME7 等基本定时器。在《STM32 参考手册》里面，定时器的介绍占了 1/5 的篇幅，足见其重要性。这一节，我们选择难度适中的通用定时器来介绍。本节分为如下几个部分：

3.7.1 STM32 通用定时器简介

3.7.2 硬件设计

3.7.3 软件设计

3.7.4 下载与测试

3.7.1 STM32 通用定时器简介

STM32 的通用定时器是一个通过可编程预分频器（PSC）驱动的 16 位自动装载计数器（CNT）构成。STM32 的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和 PWM)等。 使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32 的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM3 的通用 TIMx (TIM2、TIM3、TIM4 和 TIM5)定时器功能包括：

1)16 位向上、向下、向上/向下自动装载计数器（TIMx_CNT）。

2)16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为 1～65535 之间的任意数值。

2）4 个独立通道（TIMx_CH1~4），这些通道可以用来作为：

A．输入捕获

B．输出比较

C．PWM 生成(边缘或中间对齐模式)

D．单脉冲模式输出

3）可使用外部信号（TIMx_ETR）控制定时器和定时器互连（可以用 1 个定时器控制另外一个定时器）的同步电路。

4）如下事件发生时产生中断/DMA：

A．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)

B．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)

C．输入捕获

D．输出比较

E．支持针对定位的增量(正交)编码器和霍尔传感器电路

F．触发输入作为外部时钟或者按周期的电流管理

由于 STM32 通用定时器比较复杂，这里我们不再多介绍，请大家直接参考《STM32 参考手册》第 253 页，通用定时器一章。下面我们介绍一下与我们这章的实验密切相关的几个通用定时器的寄存器。

首先是控制寄存器 1（TIMx_CR1），该寄存器的各位描述如下：

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 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 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 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。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。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 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 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 软件通过该位选择UEV事件的源 0：如果允许产生更新中断或DMA请求，则下述任一事件产生一个更新中断或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：如果允许产生更新中断或DMA请求，则只有计数器溢出/下溢才产生一个更新中断或DMA请求。
位1	**UDIS**：禁止更新 软件通过该位允许/禁止UEV事件的产生 0：允许UEV。更新(UEV)事件由下述任一事件产生： – 计数器溢出/下溢 – 设置UG位 – 从模式控制器产生的更新 被缓存的寄存器被装入它们的预装载值。 1：禁止UEV。不产生更新事件，影子寄存器(ARR、PSC、CCRx)保持它们的值。如果设置了UG位或从模式控制器发出了一个硬件复位，则计数器和预分频器被重新初始化。
位0	**CEN**：使能计数器 0：禁止计数器； 1：使能计数器。 注：在软件设置了CEN位后，外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 在单脉冲模式下，当发生更新事件时，CEN被自动清除。

图 3.7.1.1 寄存器 TIMx_CR1 各位描述

接下来介绍第二个与我们这节密切相关的寄存器：DMA/中断使能寄存器（TIMx_DIER）。该寄存器是一个 16 位的寄存器，其各位描述如下：

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

图 3.7.1.2 寄存器 TIMx_ DIER 各位描述

这里我们仅关心它的第 6 位和第 0 位，第 6 位 TIE 为触发中断使能位，通过将该位置 1 使能 TIMx 的中断触发，注意只要是 TIMx 需要使用中断，该位必须为 1。而第 0 位，则为允许更新中断位，通过置 1，来允许由于更新事件而产生的中断。

接下来我们看第三个与我们这节有关的寄存器：预分频寄存器（TIMx_PSC）。该寄存器用设置对时钟进行分频，然后提供给计数器，作为计数器的时钟。该寄存器的各位描述如下：

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 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。

图 3.7.1.3 寄存器 TIMx_ PSC 各位描述

这里，我们的时钟来源有 4 个：

1）内部时钟（CK_INT）

2）外部时钟模式 1：外部输入脚（TIx）

3）外部时钟模式 2：外部触发输入（ETR）

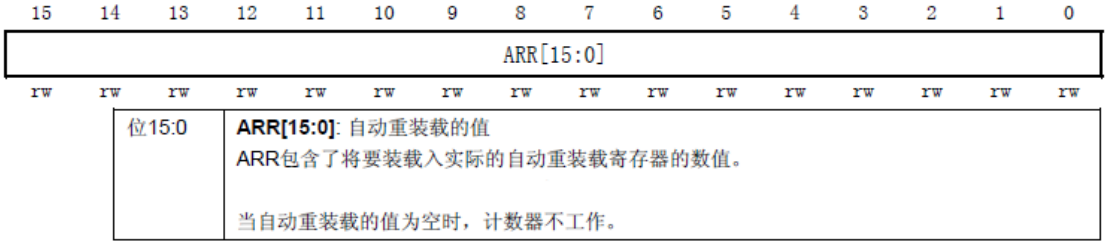
4）内部触发输入（ITRx）：使用 A 定时器作为 B 定时器的预分频器（A 为 B 提供时钟）。

这些时钟，具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟是从 APB1 倍频的来的，除非 APB1 的时钟分频数设置为 1，否则通用定时器 TIMx 的时钟是 APB1 时钟的 2 倍，当 APB1 的时钟不分频的时候，通用定时器 TIMx 的时钟就等于 APB1 的时钟。这里还要注意的就是高级定时器的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。

接着我们介绍自动重装载寄存器（TIMx_ARR），该寄存器在物理上实际对应着 2 个寄存器。一个是程序员可以直接操作的，另外一个是程序员看不到的，这个看不到的寄存器在《STM32 参考手册》里面被叫做影子寄存器。事实上真正起作用的是影子寄存器。根据 TIMx_CR1 寄存器中 APRE 位的设置：APRE=0 时，预装载寄存器的内容可以随时传送到影子寄存器，此时 2 者是连通的；而 APRE=1 时，在每一次更新事件（UEV）时，才把预装在寄存器的内容传送到影子寄存器。

自动重装载寄存器的各位描述如下：



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 ARR包含了将要装载入实际的自动重装载寄存器的数值。 当自动重装载的值为空时，计数器不工作。

图 3.7.1.4 寄存器 TIMx_ ARR 各位描述

最后，我们要介绍的寄存器是：状态寄存器（TIMx_SR）。该寄存器用来标记当前与定时器相关的各种事件/中断是否发生。该寄存器的各位描述如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			CC4OF	CC3OF	CC2OF	CC1OF	保留		TIF	保留	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc w0	rc w0	rc w0	rc w0			rc w0		rc w0	rc w0	rc w0	rc w0	rc w0

图 3.7.1.5 寄存器 TIMx_ SR 各位描述

关于这些位的详细描述，请参考《STM32 参考手册》第 245 页。

只要对以上几个寄存器进行简单的设置，我们就可以使用通用定时器了，并且可以产生中断。

这一节，我们将使用定时器产生中断，然后在中断服务函数里面翻转 LED1 上的电瓶，来指示定时器中断的产生。接下来我们以通用定时器 TIM3 为实例，来说明要经过哪些步骤，才能达到这个要求，并产生中断。

1）TIM3 时钟使能。

这里我们通过 APB1ENR 的第 1 位来设置 TIM3 的时钟，因为 Stm32_Clock_Init 函数里面把 APB1 的分频设置为 2 了，所以我们的 TIM3 时钟就是 APB1 时钟的 2 被，等于系统时钟。

2）设置 TIM3_ARR 和 TIM3_PSC 的值。

通过这两个寄存器，我们来设置自动重装的值，以及分频系数。这两个参数加上时钟频率就决定了定时器的溢出时间。

3）设置 TIM3_DIER 允许更新中断。

因为我们要使用 TIM3 的更新中断，所以设置 DIER 的 UIE 位，并使能触发中断。

4）允许 TIM3 工作。

光配置好定时器还不行，没有开启定时器，照样不能用。我们在配置完后要开启定时器，通过 TIM3_CR1 的 CEN 位来设置。

5）TIM3 中断分组设置。

在定时器配置完了之后，因为要产生中断，必不可少的要设置 NVIC 相关寄存器，以使能 TIM3 中断。

6）编写中断服务函数。

在最后，还是要编写定时器中断服务函数，通过该函数来处理定时器产生的相关中断。在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，我们这里使用的是更新（溢出）中断，所以在状态寄存器 SR 的最低位。在处理完中断之后应该向 TIM3_SR 的最低位写 0，来清除该中断标志。

通过以上几个步骤，我们就可以达到我们的目的了，使用通用定时器的的更新中断，来控制外部 LED1 的翻转。

3.7.2 硬件设计

本节将通过 TIM3 的中断来控制 DS1 的亮灭，DS1 是直接连接到 PD2 上的，所以电路上不需要任何改动。

3.7.3 软件设计

软件设计我们在之前的工程上面增加，首先在 HARDWARE 文件夹下新建 TIMER 的文件夹。然后打开 USER 文件夹下的工程，新建一个 timer.c 的文件和 timer.h 的头文件，保存在 TIMER 文件夹下，并将 TIMER 文件夹加入头文件包含路径。

我们在 timer.c 里输入如下代码：

```
#include "timer.h"
#include "led.h"
//Mini STM32 开发板
//通用定时器 驱动代码
//正点原子@ALIENTEK

//定时器 3 中断服务程序
//2ms 中断 1 次
void TIM3_IRQHandler(void)
{
    if(TIM3->SR&0X0001)//溢出中断
    {
        LED1=!LED1;
    }
    TIM3->SR&=~(1<<0);//清除中断标志位
}
//通用定时器中断初始化
//这里始终选择为 APB1 的 2 倍，而 APB1 为 36M
//arr：自动重装值。
//psc：时钟预分频数
//这里使用的是定时器 3!
void Timerx_Init(u16 arr，u16 psc)
{
    RCC->APB1ENR|=1<<1;//TIM3 时钟使能
    TIM3->ARR=arr;  //设定计数器自动重装值//刚好 1ms
    TIM3->PSC=psc;  //预分频器 7200，得到 10Khz 的计数时钟
    //这两个东东要同时设置才可以使用中断
    TIM3->DIER|=1<<0;    //允许更新中断
    TIM3->DIER|=1<<6;    //允许触发中断
```

```
    TIM3->CR1|=0x01;      //使能定时器 3
    MY_NVIC_Init(1，3，TIM3_IRQChannel，2);//抢占 1，子优先级 3，组 2

}
```

该文件下包含一个中断服务函数和一个定时器初始化函数，中断服务函数比较简单，在每次中断后，判断 TIM3 的中断类型，如果中断类型正确，则执行 LED1（DS1）的取反。

Timerx_Init 函数就是执行我们上面介绍的那 5 个步骤，使得 TIM3 开始工作，并开启中断。该函数的 2 个参数用来设置 TIM3 的溢出时间。因为我们在 Stm32_Clock_Init 函数里面已经初始化 APB1 的时钟为 2 分频，所以，TIM3 的时钟为 36M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

Tout= (arr*（psc+1））/Tclk；

其中：

Tclk：TIM3 的输入时钟频率（单位为 Khz）。

Tout：TIM3 溢出时间（单位为 ms）。

我们将 timer.c 文件保存，然后加入到 HARDWARE 组下。接下来，在 timer.h 文件里，我们输入如下代码：

```
#ifndef __TIMER_H
#define __TIMER_H
#include "sys.h"
//Mini STM32 开发板
//定时器 驱动代码
//正点原子@ALIENTEK
void Timerx_Init(u16 arr，u16 psc);
#endif
```

关于这部分代码，我们不多说了。

最后，我们修改 main 函数如下：

```
int main(void)
{
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72，9600);  //串口初始化
    LED_Init();                 //初始化与 LED 连接的硬件接口
    Timerx_Init(5000，7199);//10Khz 的计数频率，计数到 5000 为 500ms
    while(1)
    {
        LED0=!LED0;
        delay_ms(200);
    }
}
```

这里的代码和之前大同小异，此段代码对 TIM3 进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3_CNT 的值等于 TIM3_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反 LED1，TIM3_CNT 再从 0 开始计数。

3.7.4 下载与测试

在完成软件设计之后，我们将编译好的文件下载到 MiniSTM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停闪烁（每 400ms 闪烁一次），而 DS1 也是不停的闪烁，但是闪烁时间较 DS0 慢（1s 一次）。

3.8 PWM输出实验

上一节，我们介绍了 STM32 的通用定时器 TIM3，用该定时器的中断来控制 DS1 的闪烁，这一节，我们将向大家介绍如何使用 STM32 的 TIM3 来产生 PWM 输出。本节分为如下几个部分：

3.8.1 PWM 简介
3.8.2 硬件设计
3.8.3 软件设计
3.8.4 下载与测试

3.8.1 PWM简介

脉冲宽度调制(PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。简单一点，就是对脉冲宽度的控制。

STM32 的定时器除了 TIM6 和 7。其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出，这样，STM32 最多可以同时产生 30 路 PWM 输出！这里我们仅利用 TIM3 的 CH2 产生一路 PWM 输出。如果要产生多路输出，大家可以根据我们的代码稍作修改即可。

要使 STM32 的通用定时器 TIMx 产生 PWM 输出，除了上一节介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM 的。这三个寄存器分别是：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）。接下来我们简单介绍一下这三个寄存器。

首先是捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器总共有 2 个，TIMx _CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR1 控制 CH3 和 4。该寄存器的各位描述如下：

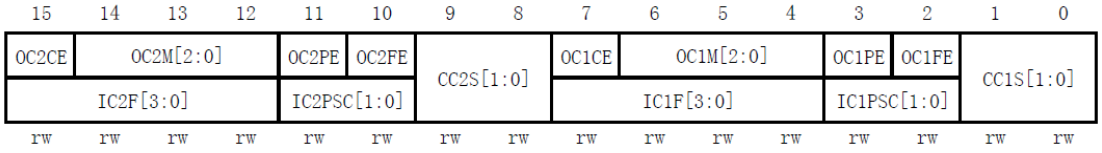


图 3.8.1.1 寄存器 TIMx_CCMR1 各位描述

该寄存器的有些位在不同模式下，功能不一样，所以上图把寄存器分了 2 层，上面一层对应输出而下面的则对应输入。关于该寄存器的详细说明，请参考《STM32 参考手册》第 288 页，14.4.7 一节。这里我们需要说明的是模式设置位 OCxM，此部分由 3 位组成。总共可以配置成 7 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110/111。这两种 PWM 模式的区别就是输出电平的极性相反。

接下来，我们介绍捕获/比较使能寄存器（TIMx_CCER），该寄存器控制着各个输入输出通道的开关。该寄存器的各位描述如下：

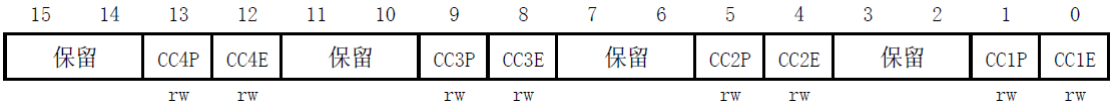


图 3.8.1.2 寄存器 TIMx_ CCER 各位描述

该寄存器比较简单，我们这里不多说了，有不明白的地方，请请参考《STM32 参考手册》第 292 页，14.4.9 这一节。

最后，我们介绍一下捕获/比较寄存器（TIMx_CCR1~4），该寄存器总共有 4 个，对应 4 个输通道 CH1~4。因为这 4 个寄存器都差不多，我们仅以 TIMx_CCR1 为例介绍，该寄存器的各位描述如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**CCR1[15:0]**: 捕获/比较1的值 **若CC1通道配置为输出：** CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性，写入的数值会立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 **若CC1通道配置为输入：** CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。

图 3.8.1.3 寄存器 TIMx_ CCR1 各位描述

在输出模式下，该寄存器的值与 CNT 的值比较，根据比较结果产生相应动作。利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的输出脉宽了。

至此，我们把这一节要用的几个 TIMx 相关寄存器都介绍完了，解析来我们就说说这一节要实现的功能。我们要利用 TIM3 的 CH2（对应 MiniSTM32 开发板的 PA7）输出 PWM 来控制 DS0 的亮度。所以我们在软件上要做的就是控制 TIM3_CH2 的 PWM 输出。接下来我们将介绍通过哪几个步骤，就可以达到这个目的：

1）开启 TIM3 时钟，配置 PA7 为复用输出。

要使用 TIM3，我们必须先开启 TIM3 的时钟(通过 APB1ENR 设置)，这点相信大家看了这么多代码，应该明白了。这里我们还要配置 PA7 为复用输出，这是因为 TIM3_CH2 通道是以 IO 复用的形式连接到 PA7 上的，这里我们要使用复用输出功能。

2）设置 TIM3 的 ARR 和 PSC。

在开启了 TIM3 的时钟之后，我们要设置 ARR 和 PSC 两个寄存器的值来控制输出 PWM 的周期。当 PWM 周期太慢（低于 50Hz）的时候，我们就会明显感觉到闪烁了。因此，PWM 周期在这里不宜设置的太小。

3）设置 TIM3_CH2 的 PWM 模式。

接下来，我们要设置 TIM3_CH2 为 PMW 模式（默认是冻结的），因为我们的 DS0 是低电平亮，而我们希望当 CCR2 的值小的时候，DS0 就暗，CCR2 值大的时候，DS0 就亮，所以我们要通过配置 TIM3_CCMR1 的相关位来控制 TIM3_CH2 的模式。

4）使能 TIM3 的 CH2 输出，使能 TIM3。

在完成以上设置了之后，我们需要开启 TIM3 的通道 2 输出以及 TIM3。前者通过 TIM3_CCER1 来设置，是单个通道的开关，而后者则通过 TIM3_CR1 来设置，是整个 TIM3 的总开关。只有设置了这两个寄存器，这样我们才能在 TIM3 的通道 2 上看到 PWM 波输出。

5）修改 TIM3_CCR2 来控制占空比。

最后，在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而我们通过修改 TIM3_CCR2 则可以控制 CH2 的输出占空比。继而控制 DS0 的亮度。

通过以上 5 个步骤，我们就可以控制 TIM3 的 CH2 输出 PWM 波了。

3.8.2 硬件设计

该部分，因为我们 DS0 是连接在 PA8 上的，而我们的 PWM 输出是在 PA7，所以，硬件上应该把 PA7 和 PA8 通过跳线帽短接起来，然后配置 PA8 为浮空输入（IO 口复位后的状态），以免干扰 PA7 的信号。因此，这一节的电路，除了在 PA7 和 PA8 之间放一个跳线帽，其他的都不需要改动。

图 3.8.2.1 硬件连接图

将上图中的 PA7 和 PA8 用跳线帽短接，图中红色圈出部分。

3.8.3 软件设计

这里，我们在之前的定时器中断实验的基础上修改，先打开之前的工程，然后我们在 timer.c 里面加入如下代码：

```
//PWM 输出初始化
//arr：自动重装值
//psc：时钟预分频数
void PWM_Init(u16 arr，u16 psc)
{
    //此部分需手动修改 IO 口设置
    RCC->APB1ENR|=1<<1;          //TIM3 时钟使能
    GPIOA->CRH&=0XFFFFFFF0;//PA8 输出
    GPIOA->CRH|=0X00000004;//浮空输入
    GPIOA->CRL&=0X0FFFFFFF;//PA7 输出
    GPIOA->CRL|=0XB0000000;//复用功能输出
    GPIOA->ODR|=1<<7;//PA7 上拉

    TIM3->ARR=arr;//设定计数器自动重装值
    TIM3->PSC=psc;//预分频器不分频
    TIM3->CCMR1|=7<<12;  //CH2 PWM2 模式
    TIM3->CCMR1|=1<<11; //CH2 预装载使能
    TIM3->CCER|=1<<4;    //OC2  输出使能
    TIM3->CR1=0x8000;    //ARPE 使能
    TIM3->CR1|=0x01;     //使能定时器 3
}
```

此部分代码包含了上面介绍的 PWM 输出设置的前 4 个步骤。这里我们关于 TIM3 的设置就不再说了，要说的是里面对于 PA7 和 PA8 的设置，此函数刚开始就设置了 PA8 为浮空输入，这是因为我们把 PA7 和 PA8 端接起来了，而在 LED_Init 函数里面有把 PA8 设置成推挽输出，因为这里我们不需要用 PA8 的输出，而是使用 PA7 的复用输出，所以必须禁止 PA8，否则就会干扰 PA7 的输出，甚至出现 IO 口自短路！而 PA7 设置成复用输出，则比较好理解了，因为我们使用的是 IO 口的第二功能。

接着我们修改 timer.h 如下：

```
#ifndef __TIMER_H
#define __TIMER_H
#include "sys.h"
//通过改变 TIM3->CCR2 的值来改变占空比，从而控制 LED0 的亮度
#define LED0_PWM_VAL TIM3->CCR2
void Timerx_Init(u16 arr，u16 psc);
void PWM_Init(u16 arr，u16 psc);
#endif
```

这里头文件与上一节的不同是加入了 PWM_Init 的声明以及宏定义了 TIM3 通道 2 的输入/捕获寄存器。通过这个宏定义，我们可以在其他文件里面修改 LED0_PWM_VAL 的值，就可以达到控制 LED0 的亮度的目的了。也就是实现了前面介绍的最后一个步骤。

接下来，我们修改 main 函数如下：

```
int main(void)
{
    u16 led0pwmval=0;
    u8 dir=1;
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);         //延时初始化
    uart_init(72，9600);   //串口初始化
    LED_Init();                  //初始化与 LED 连接的硬件接口
    PWM_Init(900，0);       //不分频。PWM 频率=72000/900=8Khz
    while(1)
    {
        delay_ms(10);
        if(dir)led0pwmval++;
        else led0pwmval--;

        if(led0pwmval>300)dir=0;
        if(led0pwmval==0)dir=1;
        LED0_PWM_VAL=led0pwmval;
    }
}
```

这里，我们从死循环函数可以看出，我们控制 LED0_PWM_VAL 的值从 0 变到 300，然后又从 300 变到 0，如此循环，因此 DS0 的亮度也会跟着从按变到亮，然后又从亮变到暗。至于这里的值，我们为什么取 300，是因为 PWM 的输出占空比达到这个值的时候，我们的 LED 亮度变化就不大了（虽然最大值可以设置到 900），因此设计过大的值在这里是没必要的。至此，

我们的软件设计就完成了。

3.8.4 下载与测试

在完成软件设计之后，将我们将编译好的文件下载到 MiniSTM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停的由暗变到亮，然后又从亮变到暗。每个过程持续时间大概为 3 秒钟左右。

实际运行结果如下图所示：

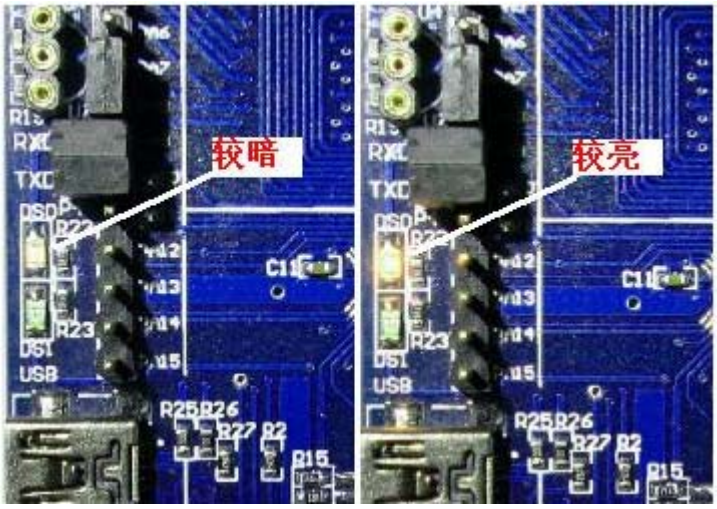


图 3.8.4.1 PWM 控制 DS0 亮度

3.9 OLED显示实验

前面所有的介绍都没有涉及到液晶显示，从这一节开始，我们将陆续向大家介绍几款液晶显示模块。本节我们将向大家介绍相对简单的 OLED。本节分为如下几个部分：

3.9.1 OLED 简介
3.9.2 硬件设计
3.9.3 软件设计
3.9.4 下载与测试

3.9.1 OLED 简介

OLED，即有机发光二极管（Organic Light-Emitting Diode），又称为有机电激光显示（Organic Electroluminesence Display， OELD）。OLED 由于同时具备自发光，不需背光源、对比度高、厚度薄、视角广、反应速度快、可用于挠曲性面板、使用温度范围广、构造及制程较简单等优异之特性，被认为是下一代的平面显示器新兴应用技术。

LCD 都需要背光，而 OLED 不需要，因为它是自发光的。这样同样的显示，OLED 效果要来得好一些。OLED 的尺寸难以大型化，但是分辨率确可以做到很高。这一节，我们使用的是 ALINETEK 的 OLED 显示模块，该模块有以下特点：

1）模块有单色和双色两种可选，单色为纯白色，而双色则为黄蓝双色。

2）尺寸小，显示尺寸为 0.96 寸，而模块的尺寸仅为 27mm*26mm 大小。

3）高分辨率，该模块的分辨率为 128*64。

4）多种接口方式，该模块提供了总共 5 种接口包括：6800、8080 两种并行接口方式、3 线或 4 线的穿行 SPI 接口方式，、IIC 接口方式（只需要 2 根线就可以控制 OLED 了！）。

5）不需要高压，直接接 3.3V 就可以工作了。

这里要提醒大家的是，该模块不和 5.0V 接口兼容，所以请大家在使用的时候一定要小心，别接到 5V 的系统上去，否则可能烧坏模块。以上 5 种模式通过模块的 BS0~2 设置，BS0~2 的设置与模块接口模式的关系如下表：

模块跳线口	IIC 接口	6800并行接口	8080并行接口	4线串行接口	3线串行接口
BS0	0	0	0	0	1
BS1	1	0	1	0	0
BS2	0	1	1	0	0

表 3.9.1.1 OLED 模块接口方式设置表

上表中：“1”代表接 VCC，而“0”代表接 GND。

该模块的外观图如下：

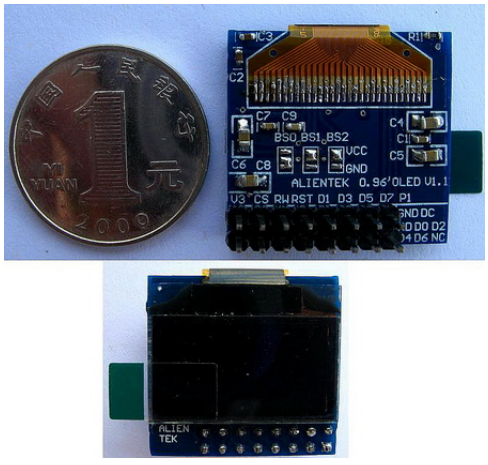


图 3.9.1.1 ALIENTEK OLED 模块外观图

模块的原理图如下：

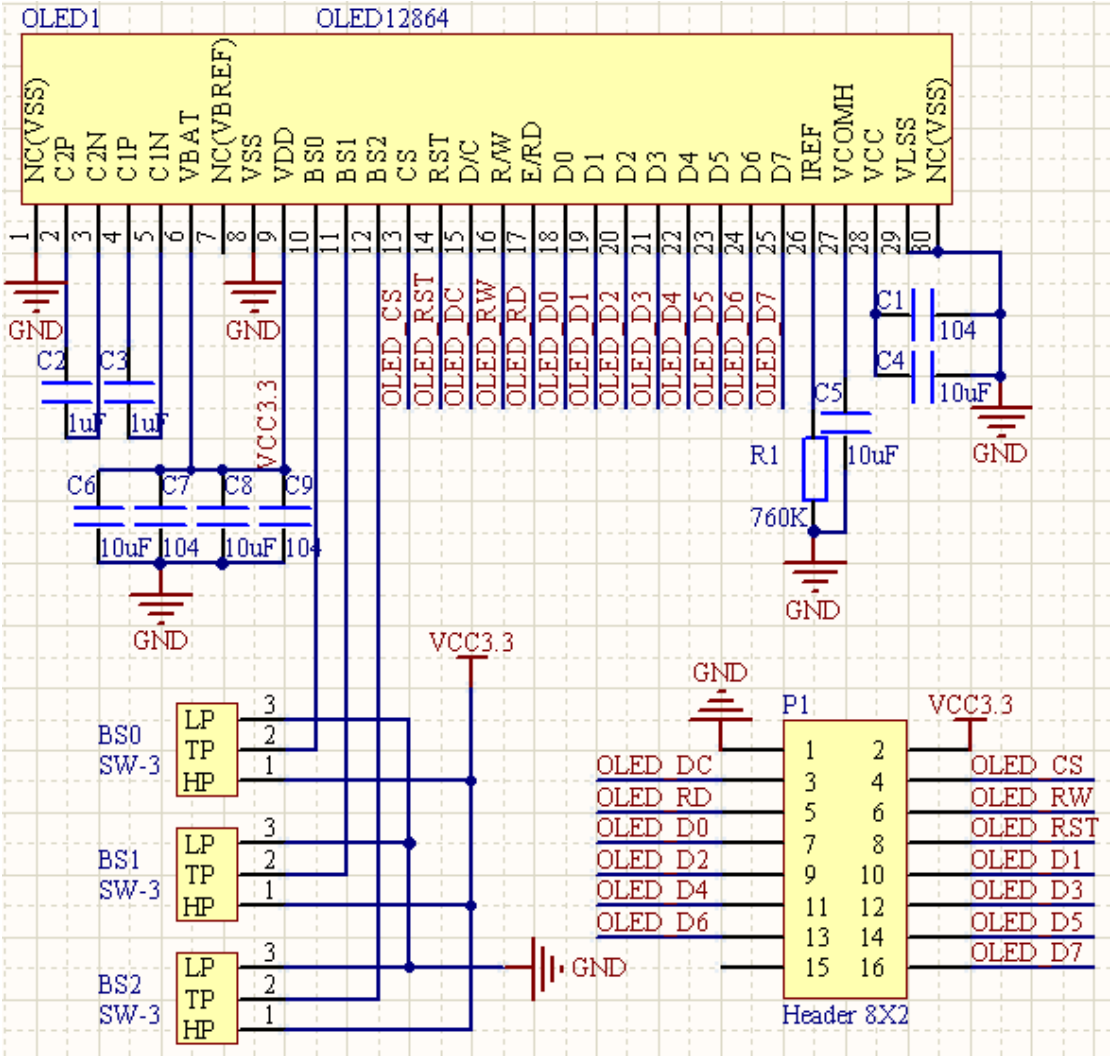


图 3.9.1.2 ALIENTEK OLED 模块原理图

该模块采用 8*2 的 2.54 排针与外部连接，其引线图如上图所示，总共有 16 个管脚，在 16 条线中，我们只用了 15 条，有一个是悬空的。15 条线中，电源和地线占了 2 条，还剩下 13 条信号线。在不同模式下，我们需要的信号线数量是不同的，在 8080 模式下，需要全部 13 条，而在 IIC 模式下，仅需要 2 条线就够了！这其中有一条是共同的，那就是复位线 RST（RES），该线我们可以直接接在 MCU 的复位上（要先确认复位方式一样），这样可以省掉一条线。

ALIENTEK OLED 模块的控制器是 SSD1306，这一节，我们将学习如何通过 STM32 来控制该模块显示字符和数字，本节实例将可以支持 2 种方式与 OLED 模块连接，一种是 8080 的并口方式，另外一种是 4 线 SPI 方式。

首先我们介绍一下模块的 8080 并行接口，8080 并行接口的发明者是 INTEL，该总线也被广泛应用于各类液晶显示器，ALIENTEK OLED 模块也提供了这种接口，使得 MCU 可以快速的访问 OLED。ALIENTEK OLED 模块的 8080 接口方式需要如下一些信号线：

CS：OLED 片选信号。

WR：向 OLED 写入数据。

RD：从 OLED 读取数据。

D[7:0]：8 位双向数据线。

RST(RES)：硬复位 OLED。

DC：命令/数据标志（0，读写命令；1，读写数据）。

模块的 8080 并口读/写的过程为：先根据要写入/读取的数据的类型，设置 DC 为高（数据）/低（命令），然后拉低片选，选中 SSD1306，接着我们根据是读数据，还是要写数据置 RD/WR 为低，然后：

在 RD 的上升沿， 使数据锁存到数据线（D[7:0]）上；

在 WR 的上升沿，使数据写入到 SSD1306 里面；

SSD1306 的 8080 并口写时序图如下：

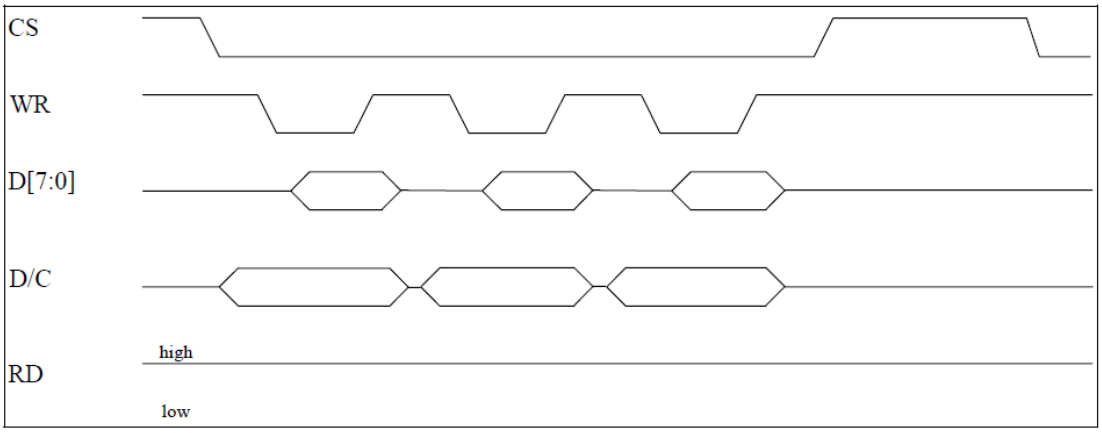


图 3.9.1.3　8080 并口写时序图

SSD1306 的 8080 并口读时序图如下：

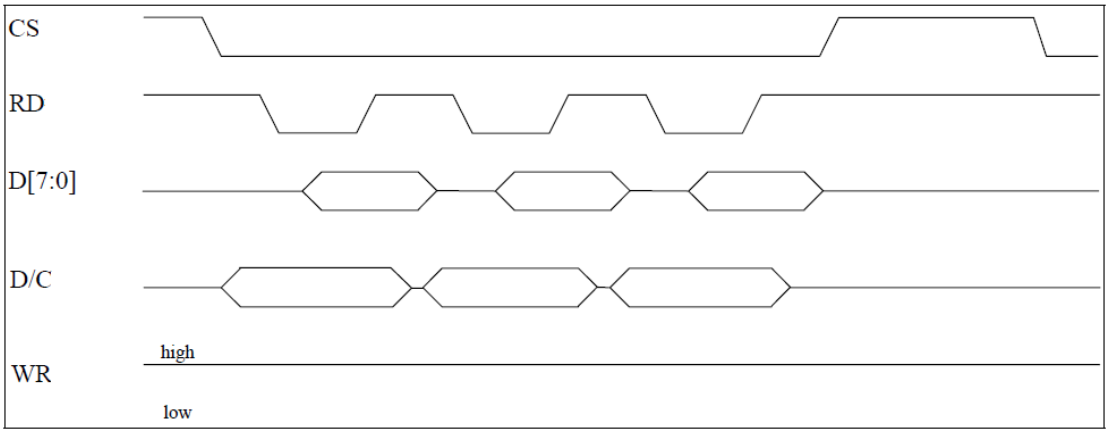


图 3.9.1.4　8080 并口读时序图

SSD1306 的 8080 接口方式下，控制脚的信号状态所对应的功能如下表：

功能	RD	WR	CS	DC
写命令	H	↑	L	L
读状态	↑	H	L	L
写数据	H	↑	L	H
读数据	↑	H	L	H

表 3.9.1.2　控制脚信号状态功能表

在 8080 方式下读数据操作的时候，我们有时候（例如读显存的时候）需要一个假读命（Dummy Read），以使得微控制器的操作频率和显存的操作频率相匹配。在读取真正的数据之前，由一个的假读的过程。这里的假读，其实就是第一个读到的字节丢弃不要，从第二个开始，才是我们真正要读的数据。

一个典型的读显存的时序图，如下图所示：

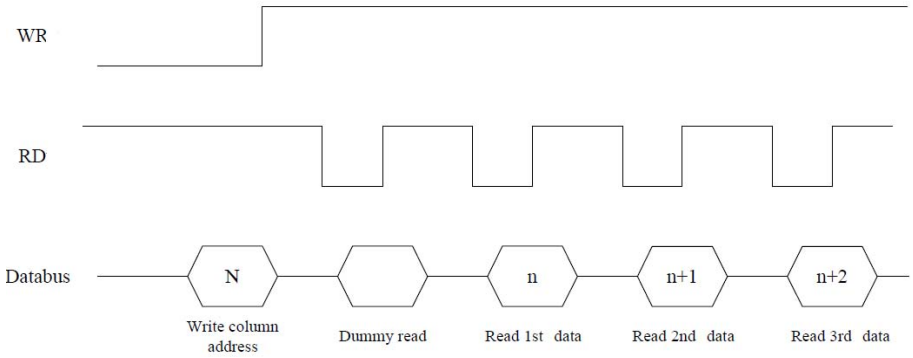


图 3.9.1.5　读显存时序图

可以看到，在发送了列地址之后，开始读数据，第一个是 Dummy Read，也就是假读，我们从第二个开始，才算是真正有效的数据。

并行接口模式就介绍到这里，我们接下来介绍一下 4 线串行（SPI）方式，4 先串口模式使用的信号线有如下几条：

CS：OLED 片选信号。

RST(RES)：硬复位 OLED。

DC：命令/数据标志（0，读写命令；1，读写数据）。

SCLK：串行时钟线。在 4 线串行模式下，D0 信号线作为串行时钟线 SCLK。

SDIN：串行数据线。在 4 线串行模式下，D1 信号线作为串行数据线 SDIN。

模块的 D2 需要悬空，其他引脚可以接到 GND。在 4 线串行模式下，只能往模块写数据而不能读数据。

在 4 线 SPI 模式下，每个数据长度均为 8 位，在 SCLK 的上升沿，数据从 SDIN 移入到 SSD1306，并且是高位在前的。DC 线还是用作命令/数据的标志线。在 4 线 SPI 模式下，写操作的时序如下：

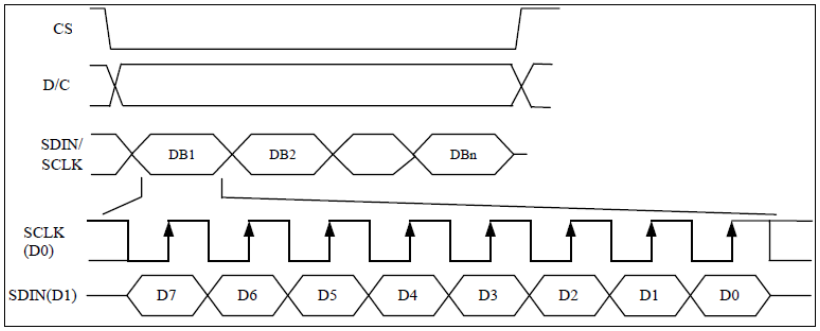


图 3.9.1.6　4 线 SPI 写操作时序图

4 线串行模式就为大家介绍到这里。其他还有几种模式，在 SSD1306 的数据手册上都有详细的介绍，如果要使用这些方式，请大家参考该手册。

接下来，我们介绍一下模块的显存，SSD1306 的显存总共为 128*64bit 大小，SSD1306 将这些显存分为了 8 页，其对应关系如下：

	行（COL0~127）						
列（COM0~63）	SEG0	SEG1	SEG2	……	SEG125	SEG126	SEG127
	PAGE0						
	PAGE1						
	PAGE2						
	PAGE3						
	PAGE4						
	PAGE5						
	PAGE6						
	PAGE7						

表 3.9.1.3　SSD1306 显存与屏幕对应关系表

可以看出，SSD1306 的每页包含了 128 个字节，总共 8 页，这样刚好是 128*64 的点阵大小。因为每次写入都是按字节写入的，这就存在一个问题，如果我们使用只写方式操作模块，那么，每次要写 8 个点，这样，我们在画点的时候，就必须把要设置的点所在的字节的每个位都搞清楚当前的状态（0/1？），否则写入的数据就会覆盖掉之前的状态，结果就是有些不需要显示的点，显示出来了，或者该显示的没有显示了。这个问题在能读的模式下，我们可以先读出来要写入的那个字节，得到当前状况，在修改了要改写的位之后再写进 GRAM，这样就不会影响到之前的状况了。但是这样需要能读 GRAM，对于 3 线或 4 线 SPI 模式，模块是不支持读的，而且读->改->写的方式速度也比较慢。

所以我们采用的办法是在 STM32 的内部建立一个 OLED 的 GRAM（共 128 个字节），在每次修改的时候，只是修改 STM32 上的 GRAM（实际上就是 SRAM），在修改完了之后，一次性把 STM32 上的 GRAM 写入到 OLED 的 GRAM。当然这个方法也有坏处，就是对于那些 SRAM 很小的单片机（比如 51 系列）就比较麻烦了。

SSD1306 的命令比较多，这里我们仅介绍几个比较常用的命令，这些命令如下表：

序号	指令	各位描述								命令	说明
	HEX	D7	D6	D5	D4	D3	D2	D1	D0		
0	81	1	0	0	0	0	0	0	1	设置对比度	A的值越大屏幕越亮，A的范围从0X00~0XFF
	A[7:0]	A7	A6	A5	A4	A3	A2	A1	A0		
1	AE/AF	1	0	1	0	1	1	1	X0	设置显示开关	X0=0，关闭显示；X0=1，开启显示；
2	8D	1	0	0	0	1	1	0	1	电荷泵设置	A2=0，关闭电荷泵
	A[7:0]	*	*	0	1	0	A2	0	0		A2=1,开启电荷泵
3	B0~B7	1	0	1	1	0	X2	X1	X0	设置页地址	X[2:0]=0~7对应页0~7
4	00~0F	0	0	0	0	X3	X2	X1	X0	设置列地址低四位	设置8位起始列地址的低四位
5	10~1F	0	0	0	0	X3	X2	X1	X0	设置列地址高四位	设置8位起始列地址的高四位

表 3.9.1.4　SSD1306 常用命令表

第一个命令为 0X81，用于设置对比度的，这个命令包含了两个字节，第一个 0X81 为命令，随后发送的一个字节为要设置的对比度的值。这个值设置得越大屏幕就越亮。

第二个命令为 0XAE/0XAF。0XAE 为关闭显示命令；0XAF 为开启显示命令。

第三个命令为 0X8D，该指令也包含 2 个字节，第一个为命令字，第二个为设置值，第二个字节的 BIT2 表示电荷泵的开关状态，该位为 1，则开启电荷泵，为 0 则关闭。在模块初始化的时候，这个必须要开启，否则是看不到屏幕显示的。

第四个命令为 0XB0~B7，该命令用于设置页地址，其低三位的值对应着 GRAM 的页地址。

第五个指令为 0X00~0X0F，该指令用于设置显示时的起始列地址低四位。

第六个指令为 0X10~0X1F，该指令用于设置显示时的起始列地址高四位。

其他命令，我们就不在这里一一介绍了，大家可以参考 SSD1306 datasheet 的第 28 页。从这页开始，对 SSD1306 的指令有详细的介绍。

最后，我们再来介绍一下 OLED 模块的初始化过程，SSD1306 的典型初始化框图如下图所示：

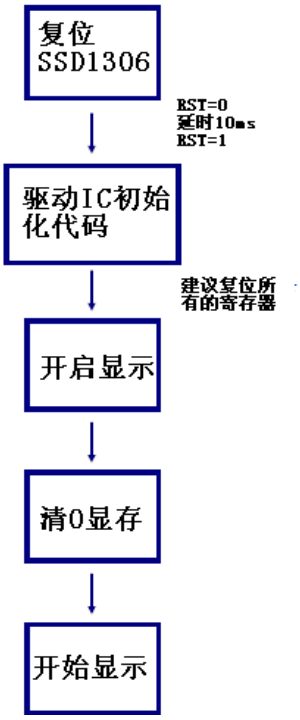


图 3.9.1.7　SSD1306 初始化框图

驱动 IC 的初始化代码，我们直接使用厂家推荐的设置就可以了，只要对细节部分进行一些修改，使其满足我们自己的要求即可，其他不需要变动。

OLED 的介绍就到此为止，我们重点向大家介绍了 ALIENTEK OLED 模块的相关知识，接下来我们将使用这个模块来显示字符和数字。通过以上介绍，我们可以得出 OLED 显示需要的相关设置步骤如下：

1）设置 STM32 与 OLED 模块相连接的 IO。

这一步，先将我们与 OLED 模块相连的 IO 口设置为输出，具体使用哪些 IO 口，这里需要根据连接电路以及 OLED 模块所设置的通讯模式来确定。这些将在硬件设计部分向大家介绍。

2）初始化 OLED 模块。

其实这里就是上面的初始化框图的内容，通过对 OLED 相关寄存器的初始化，来启动 OLED 的显示。为后续显示字符和数字做准备。

3）通过函数将字符和数字显示到 OLED 模块上。

这里就是通过我们设计的程序，将要显示的字符送到 OLED 模块就可以了，这些函数将在软件设计部分向大家介绍。

通过以上三步，我们就可以使用 ALIENTEK OLED 模块来显示字符和数字了，在后面我们还将会给大家介绍显示汉字的方法。这一部分就先介绍到这里。

3.9.2 硬件设计

OLED 模块的电路在上一节已有详细说明了，这里我们介绍 OLED 模块与 ALIETEK MiniSTM32 开发板的连接，MiniSTM32 开发板地板的 LCD 接口和 ALIENTEK OLED 模块直接可以对插，连接如下图：

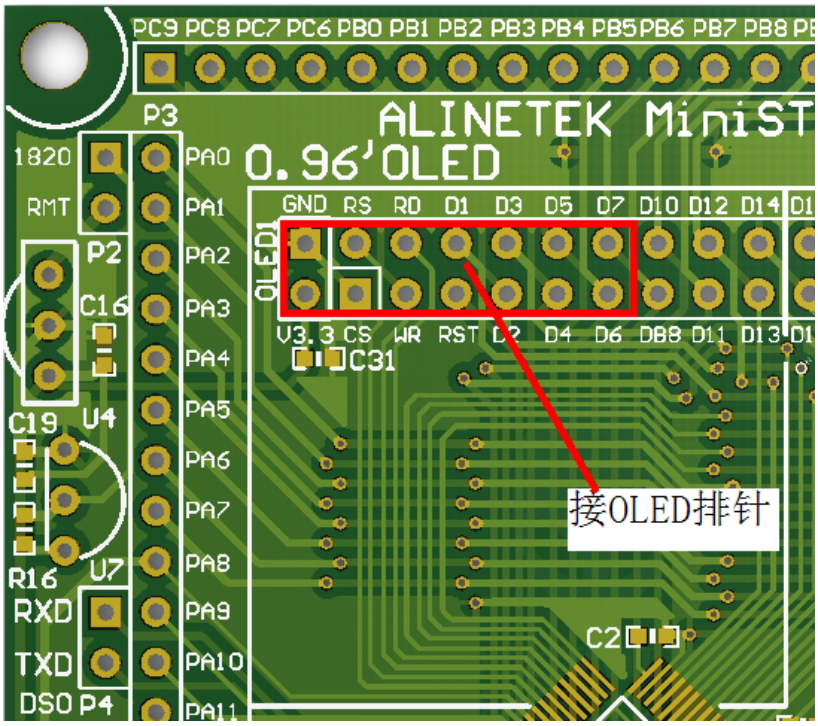


图 3.9.2.1　OLED 模块与开发板连接示意图

图中红色线圈出来的部分就是连接 OLED 的接口，这里在硬件上，OLED 与 MiniSTM32 开发板的 IO 口对应关系如下：

OLED_CS 对应 PC9;

OLED_RS 对应 PC8;

OLED_WR 对应 PC7;

OLED_RD 对应 PC6;

OLED_D[7:0]对应 PB[7:0];

这些线的连接，MiniSTM32 的内部已经连接好了，我们只需要将 OLED 模块插上去就好了。实物连接如下图所示：

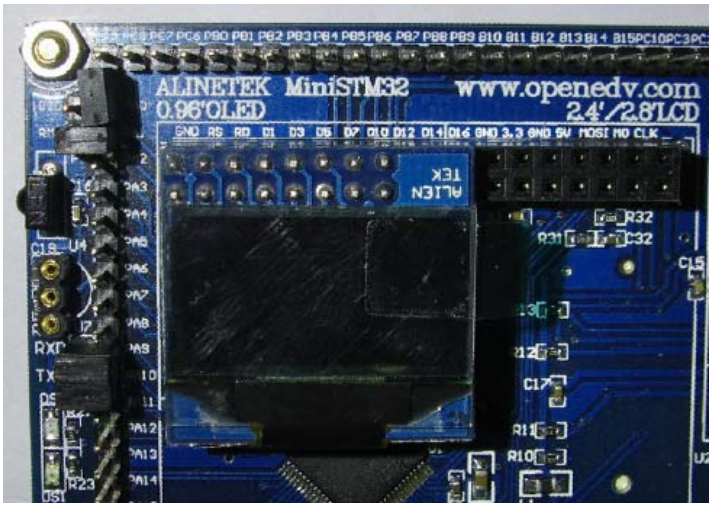


图 3.9.2.2　OLED 模块与开发板连接实物图

3.9.3 软件设计

软件设计我们依旧在之前的工程上面增加，首先在 HARDWARE 文件夹下新建一个 OLED 的文件夹。然后打开 USER 文件夹下的工程，新建一个 oled.c 的文件和 oled.h 的头文件，保存在 OLED 文件夹下，并将 OLED 文件夹加入头文件包含路径。

打开 oled.c，输入如下代码：

```
#include "oled.h"
#include "stdlib.h"
#include "font.h"
#include "delay.h"
//OLED 的显存
//存放格式如下.
//[0]0 1 2 3 ... 127
//[1]0 1 2 3 ... 127
//[2]0 1 2 3 ... 127
//[3]0 1 2 3 ... 127
//[4]0 1 2 3 ... 127
//[5]0 1 2 3 ... 127
//[6]0 1 2 3 ... 127
//[7]0 1 2 3 ... 127
u8 OLED_GRAM[128][8];
//更新显存到 LCD
void OLED_Refresh_Gram(void)
{
    u8 i，n;
    for(i=0;i<8;i++)
    {
```

```
            OLED_WR_Byte (0xb0+i，OLED_CMD);      //设置页地址（0~7）
            OLED_WR_Byte (0x00，OLED_CMD);        //设置显示位置—列低地址
            OLED_WR_Byte (0x10，OLED_CMD);        //设置显示位置—列高地址
            for(n=0;n<128;n++)OLED_WR_Byte(OLED_GRAM[n][i]，OLED_DATA);
        }
}
#if OLED_MODE==1
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0，表示命令;1，表示数据;
void OLED_WR_Byte(u8 dat，u8 cmd)
{
    DATAOUT(dat);
    OLED_RS=cmd;
    OLED_CS=0;
    OLED_WR=0;
    OLED_WR=1;
    OLED_CS=1;
    OLED_RS=1;
}
#else
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0，表示命令;1，表示数据;
void OLED_WR_Byte(u8 dat，u8 cmd)
{
    u8 i;
    OLED_RS=cmd; //写命令
    OLED_CS=0;
    for(i=0;i<8;i++)
    {
        OLED_SCLK=0;
        if(dat&0x80)OLED_SDIN=1;
        else OLED_SDIN=0;
        OLED_SCLK=1;
        dat<<=1;
    }
    OLED_CS=1;
    OLED_RS=1;
}
#endif

//开启 OLED 显示
```

```
void OLED_Display_On(void)
{
    OLED_WR_Byte(0X8D，OLED_CMD);  //SET DCDC 命令
    OLED_WR_Byte(0X14，OLED_CMD);  //DCDC ON
    OLED_WR_Byte(0XAF，OLED_CMD);  //DISPLAY ON
}
//关闭 OLED 显示
void OLED_Display_Off(void)
{
    OLED_WR_Byte(0X8D，OLED_CMD);  //SET DCDC 命令
    OLED_WR_Byte(0X10，OLED_CMD);  //DCDC OFF
    OLED_WR_Byte(0XAE，OLED_CMD);  //DISPLAY OFF
}
//清屏函数，清完屏，整个屏幕是黑色的!和没点亮一样!!!
void OLED_Clear(void)
{
    u8 i，n;
    for(i=0;i<8;i++)for(n=0;n<128;n++)OLED_GRAM[n][i]=0X00;
    OLED_Refresh_Gram();//更新显示
}
//画点
//x:0~127
//y:0~63
//t:1 填充 0，清空
void OLED_DrawPoint(u8 x，u8 y，u8 t)
{
    u8 pos，bx，temp=0;
    if(x>127||y>63)return;//超出范围了.
    pos=7-y/8;
    bx=y%8;
    temp=1<<(7-bx);
    if(t)OLED_GRAM[x][pos]|=temp;
    else OLED_GRAM[x][pos]&=~temp;
}
//x1，y1，x2，y2 填充区域的对角坐标
//确保 x1<=x2;y1<=y2 0<=x1<=127 0<=y1<=63
//dot:0，清空;1，填充
void OLED_Fill(u8 x1，u8 y1，u8 x2，u8 y2，u8 dot)
{
    u8 x，y;
    for(x=x1;x<=x2;x++)
    {
        for(y=y1;y<=y2;y++)OLED_DrawPoint(x，y，dot);
```

```
        }
        OLED_Refresh_Gram();//更新显示
    }
    //在指定位置显示一个字符，包括部分字符
    //x:0~127
    //y:0~63
    //mode:0，反白显示;1，正常显示
    //size:选择字体 16/12
    void OLED_ShowChar(u8 x，u8 y，u8 chr，u8 size，u8 mode)
    {
        u8 temp，t，t1;
        u8 y0=y;
        chr=chr-' ';//得到偏移后的值
        for(t=0;t<size;t++)
        {
            if(size==12)temp=asc2_1206[chr][t];  //调用 1206 字体
            else temp=asc2_1608[chr][t];         //调用 1608 字体
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)OLED_DrawPoint(x，y，mode);
                else OLED_DrawPoint(x，y，!mode);
                temp<<=1;
                y++;
                if((y-y0)==size)
                {
                    y=y0;
                    x++;
                    break;
                }
            }
        }
    }
    //m^n 函数
    u32 mypow(u8 m，u8 n)
    {
        u32 result=1;
        while(n--)result*=m;
        return result;
    }
    //显示 2 个数字
    //x，y :起点坐标
    //len :数字的位数
    //size:字体大小
```

```
//mode:模式  0，填充模式;1，叠加模式
//num:数值(0~4294967295);
void OLED_ShowNum(u8 x，u8 y，u32 num，u8 len，u8 size)
{
    u8 t，temp;
    u8 enshow=0;
    for(t=0;t<len;t++)
    {
        temp=(num/mypow(10，len-t-1))%10;
        if(enshow==0&&t<(len-1))
        {
            if(temp==0)
            {
                OLED_ShowChar(x+(size/2)*t，y，' '，size，1);
                continue;
            }else enshow=1;

        }
        OLED_ShowChar(x+(size/2)*t，y，temp+'0'，size，1);
    }
}
//显示字符串
//x，y:起点坐标
//*p:字符串起始地址
//用 16 字体
void OLED_ShowString(u8 x，u8 y，const u8 *p)
{
#define MAX_CHAR_POSX 122
#define MAX_CHAR_POSY 58
    while(*p!='\0')
    {
        if(x>MAX_CHAR_POSX){x=0;y+=16;}
        if(y>MAX_CHAR_POSY){y=x=0;OLED_Clear();}
        OLED_ShowChar(x，y，*p，16，1);
        x+=8;
        p++;
    }
}
//初始化 SSD1303
void OLED_Init(void)
{
    RCC->APB2ENR|=1<<3;     //使能 PORTB 时钟
    RCC->APB2ENR|=1<<4;     //使能 PORTC 时钟
```

```
#if OLED_MODE==1
    RCC->APB2ENR|=1<<0;     //开启辅助时钟
    AFIO->MAPR=0X04000000; //关闭 JTAG

    GPIOB->CRL=0X33333333;
    GPIOB->ODR|=0XFFFF;

    GPIOC->CRH&=0XFFFFFF00;
    GPIOC->CRL&=0X00FFFFFF;
    GPIOC->CRH|=0X00000033;
    GPIOC->CRL|=0X33000000;
    GPIOC->ODR|=0X03C0;
#else
    GPIOB->CRL&=0XFFFFFF00;
    GPIOB->CRL|=0XF0000033;
    GPIOB->ODR|=0X03;

    GPIOC->CRH&=0XFFFFFF00;
    GPIOC->CRH|=0X00000033;
    GPIOC->ODR|=3<<8;
#endif
    //OLED_RST=0;
    //delay_ms(100);
    //OLED_RST=1;
    OLED_WR_Byte(0xAE，OLED_CMD); //关闭显示
    OLED_WR_Byte(0xD5，OLED_CMD); //设置时钟分频因子，震荡频率
    OLED_WR_Byte(80，OLED_CMD);   //[3:0]，分频因子;[7:4]，震荡频率
    OLED_WR_Byte(0xA8，OLED_CMD); //设置驱动路数
    OLED_WR_Byte(0X3F，OLED_CMD); //默认 0X3F(1/64)
    OLED_WR_Byte(0xD3，OLED_CMD); //设置显示偏移
    OLED_WR_Byte(0X00，OLED_CMD); //默认为 0

    OLED_WR_Byte(0x40，OLED_CMD); //设置显示开始行 [5:0]，行数.

    OLED_WR_Byte(0x8D，OLED_CMD); //电荷泵设置
    OLED_WR_Byte(0x14，OLED_CMD); //bit2，开启/关闭
    OLED_WR_Byte(0x20，OLED_CMD); //设置内存地址模式
    OLED_WR_Byte(0x02，OLED_CMD); //[1:0]，00，列地址模式;01，行地址模式;10，
页地址模式;默认 10;
    OLED_WR_Byte(0xA1，OLED_CMD); //段重定义设置，bit0:0，0->0;1，0->127;
    OLED_WR_Byte(0xC0，OLED_CMD); //设置 COM 扫描方向;bit3:0，普通模式;1，重
定义模式 COM[N-1]->COM0;N:驱动路数
    OLED_WR_Byte(0xDA，OLED_CMD); //设置 COM 硬件引脚配置
```

```
    OLED_WR_Byte(0x12，OLED_CMD); //[5:4]配置

    OLED_WR_Byte(0x81，OLED_CMD); //对比度设置
    OLED_WR_Byte(0xEF，OLED_CMD); //1~255;默认 0X7F (亮度设置，越大越亮)
    OLED_WR_Byte(0xD9，OLED_CMD); //设置预充电周期
    OLED_WR_Byte(0xf1，OLED_CMD); //[3:0]，PHASE 1;[7:4]，PHASE 2;
    OLED_WR_Byte(0xDB，OLED_CMD); //设置 VCOMH 电压倍率
    OLED_WR_Byte(0x30，OLED_CMD); //[6:4] 000，0.65*vcc;001，0.77*vcc;011，0.83*vcc;

    OLED_WR_Byte(0xA4，OLED_CMD); //全局显示开启;bit0:1，开启;0，关闭;(白屏/黑屏)
    OLED_WR_Byte(0xA6，OLED_CMD); //设置显示方式;bit0:1，反相显示;0，正常显示

    OLED_WR_Byte(0xAF，OLED_CMD); //开启显示
    OLED_Clear();
}
```

这里代码明显比之前的例程多了，函数也比较多，这里我们仅针对几个比较重要的函数进行介绍。

首先要介绍的是我们定义在 STM32 内部的 GRAM，u8 OLED_GRAM[128][8];此部分 GRAM 对应 OLED 模块上的 GRAM。在操作的时候，我们只要修改 STM32 内部的 GRAM 就可以了，然后通过 OLED_Refresh_Gram 函数把 GRAM 一次刷新到 OLED 的 GRAM 上。该函数代码如下：

```
void OLED_Refresh_Gram(void)
{
    u8 i，n;
    for(i=0;i<8;i++)
    {
        OLED_WR_Byte (0xb0+i，OLED_CMD);//设置页地址（0~7）
        OLED_WR_Byte (0x00，OLED_CMD); //设置显示位置—列低地址
        OLED_WR_Byte (0x10，OLED_CMD); //设置显示位置—列高地址
        for(n=0;n<128;n++)OLED_WR_Byte(OLED_GRAM[n][i]，OLED_DATA);
    }
}
```

函数先设置页地址，然后写入列地址（也就是纵坐标），然后从 0 开始写入 128 个字节，写满该页，最后循环把 8 页的内容都写入，就实现了整个从 STM32 显存到 OLED 显存的拷贝。 。

OLED_Refresh_Gram 函数还用到了一个外部函数 OLED_WR_Byte，该函数直接和硬件相关，该函数代码如下：

```
#if OLED_MODE==1
void OLED_WR_Byte(u8 dat，u8 cmd)
{
    DATAOUT(dat);
    OLED_RS=cmd;
    OLED_CS=0;
```

```
        OLED_WR=0;
        OLED_WR=1;
        OLED_CS=1;
        OLED_RS=1;
    }
    #else
    void OLED_WR_Byte(u8 dat，u8 cmd)
    {
        u8 i;
        OLED_RS=cmd; //写命令
        OLED_CS=0;
        for(i=0;i<8;i++)
        {
            OLED_SCLK=0;
            if(dat&0x80)OLED_SDIN=1;
            else OLED_SDIN=0;
            OLED_SCLK=1;
            dat<<=1;
        }
        OLED_CS=1;
        OLED_RS=1;
    }
    #endif
```

这里有 2 个一样的函数，通过宏定义 OLED_MODE 来决定使用哪一个。如果 OLED_MODE=1，就定义为并口模式，选择第一个函数，而如果为 0，则为 4 线串口模式，选择第二个函数。这两个函数输入参数均为 2 个：dat 和 cmd，dat 为要写入的数据，cmd 则表明该数据是命令还是数据。这两个函数的时序操作就是根据上面我们对 8080 接口以及 4 线 SPI 接口的时序来编写的。

OLED_GRAM[128][8]中的 128 代表列数，也就是 x 坐标，而 8 代表的是页，每个代表 8 个列，从高到底对应列数从小到大。比如，我们要在 x=100，y=29 这个点写入 1，则可以用这个句子实现：

```
OLED_GRAM[100][4]|=1<<2；
```

一个通用的在点（x，y）置 1 表达式为：

```
OLED_GRAM[x][y/8]|=1<<(7-y%8)；
```

因此，我们可以得出下一个画点函数，void OLED_DrawPoint(u8 x，u8 y，u8 t)；代码如下：

```
void OLED_DrawPoint(u8 x，u8 y，u8 t)
{
    u8 pos，bx，temp=0;
    if(x>127||y>63)return;//超出范围了.
    pos=7-y/8;
    bx=y%8;
    temp=1<<(7-bx);
    if(t)OLED_GRAM[x][pos]|=temp;
```

```
        else OLED_GRAM[x][pos]&=~temp;
    }
```

该函数有 3 个参数，前两个是坐标，第三个 t 为要写入 1 还是 0。该函数实现了我们在 OLED 模块上任意位置画点的功能。

在介绍完画点函数之后，我们介绍一下显示字符函数，OLED_ShowChar，在介绍之前，我们来介绍一下字符（ASCII 字符集）是怎么显示在 OLED 模块上去的。要显示字符，我们先要有字符的点阵数据，ASCII 常用的字符集总共有 95 个，从空格符开始，分别为：!"#$%&'()*+,-0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~.

我们先要得到这个字符集的点阵数据，这里我们介绍一个款很好的字符提取软件：PCtoLCD2002 完美版。该软件可以提供各种字符，包括汉字（字体和大小都可以自己设置）阵提取，且取模方式可以设置好几种，常用的取模方式，该软件都支持。该软件还支持图形模式，也就是用户可以自己定义图片的大小，然后画图，根据所画的图形再生成点阵数据，这功能在制作图标或图片的时候很有用。

该软件的界面如下：

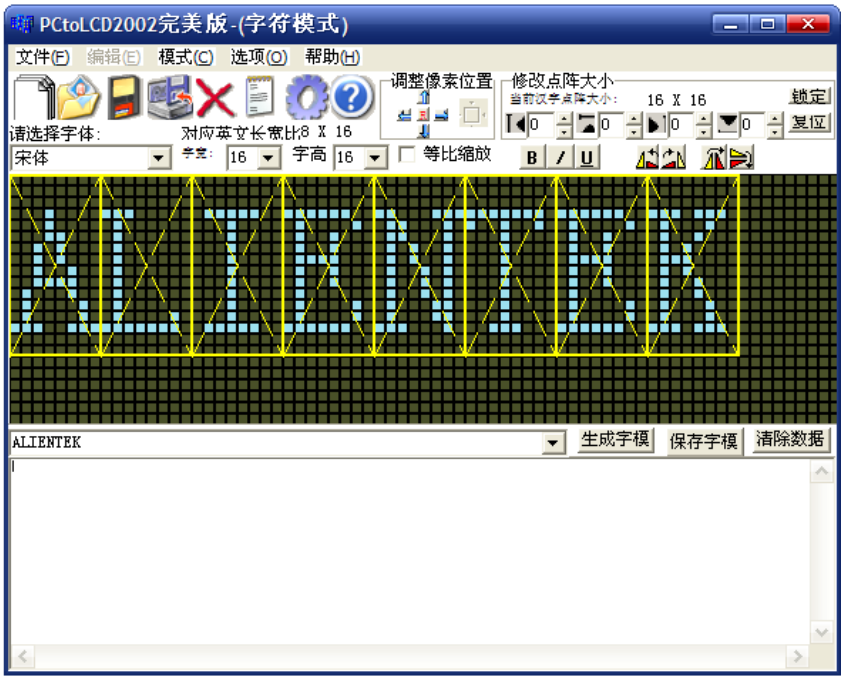


图 3.9.3.1　PCtoLCD2002 软件界面

然后我们选择设置，在设置里面设置取模方式入下图所示：

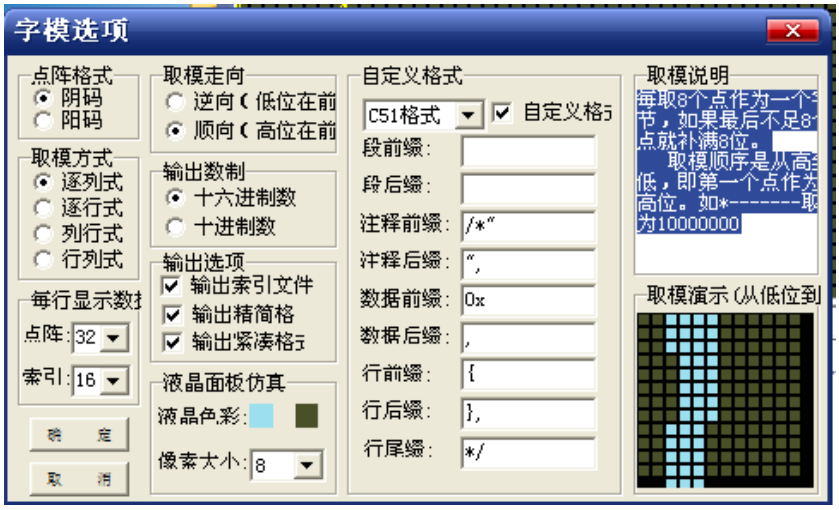


图 3.9.3.2　设置取模方式

上图设置的取模方式，在右上角的取模说明里面有，即：从第一列开始向下每取 8 个点作为一个字节，如果最后不足 8 个点就补满 8 位。取模顺序是从高到低，即第一个点作为最高位。如*-------取为 10000000。其实就是按如下这种方式：

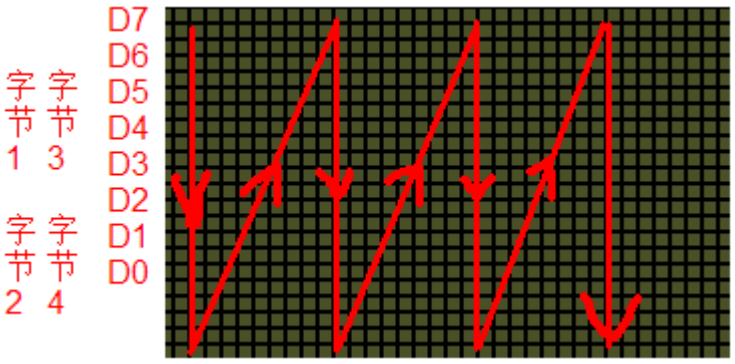


图 3.9.3.3　取模方式图解

从上到下，从左到右，高位在前。我们按这样的取模方式，然后把 ASCII 字符集按 12*6 大小和 16*0 大小取模出来（对应汉字大小为 12*12 和 16*16，字符的只有汉字的一半大！），保存在 font.h 里面，每个 12*6 的字符占用 12 个字节，每个 16*8 的字符占用 16 个字节。具体见 font.h 部分代码（该部分我们不再这里列出来了，请大家参考光盘里面的代码）。

在知道了取模方式之后，我们就可以根据取模的方式来编写显示字符的代码了，这里我们针对以上取模方式的显示字符代码如下：

```
void OLED_ShowChar(u8 x，u8 y，u8 chr，u8 size，u8 mode)
{
    u8 temp，t，t1;
    u8 y0=y;
    chr=chr-' ';//得到偏移后的值
    for(t=0;t<size;t++)
    {
        if(size==12)temp=asc2_1206[chr][t];  //调用 1206 字体
        else temp=asc2_1608[chr][t];         //调用 1608 字体
```

```
            for(t1=0;t1<8;t1++)
            {
                if(temp&0x80)OLED_DrawPoint(x，y，mode);
                else OLED_DrawPoint(x，y，!mode);
                temp<<=1;
                y++;
                if((y-y0)==size)
                {
                    y=y0;
                    x++;
                    break;
                }
            }
        }
    }
```

该函数为字符以及字符串显示的核心部分，函数中 chr=chr-' ';这句是要得到在字符点阵数据里面的实际地址，因为我们的取模是从空格键开始的，例如 asc2_1206[0][12]，代表的是空格键的点阵码。在接下来的代码，我们也是按照从上到小，从左到右的取模方式来编写的，先得到最高位，然后判断是写 1 还是 0，画点；接着读第二位，如此循环，直到一个字符的点阵全部取完为止。这其中涉及到列地址和行地址的自增，根据取模方式来理解，就不难了。

最后，我们介绍一下初始化函数，void OLED_Init(void)，该函数代码比较长，我们就不列出来了，见上面 oled.c 代码部分。 该函数的结构比较简单，开始是对 IO 口的初始化，这里我们也是用了宏定义 OLED_MODE 来决定要设置的 IO 口，其他就是一些初始化序列了，我们按照厂家提供的资料来做就可以。最后要说明一点的是，因为 OLED 是无背光的，在初始化之后，我们把显存都清空了，所以我们在屏幕上是看不到任何内容的，跟完全通电一个样，不要以为这就是初始化失败，要写入数据模块才会显示的。

oled.c 的内容就为大家介绍到这里，将 oled.c 保存，然后加入到 HARDWARE 组下。接下来我们在 oled.h 中输入如下代码：

```
#ifndef __OLED_H
#define __OLED_H
#include "sys.h"
#include "stdlib.h"
//OLED 模式设置
//0:4 线串行模式
//1:并行 8080 模式
#define OLED_MODE 1

//-----------------OLED 端口定义----------------
#define OLED_CS PCout(9)
//#define OLED_RST   PBout(14)//在 MINISTM32 上直接接到了 STM32 的复位脚！
#define OLED_RS PCout(8)
#define OLED_WR PCout(7)
#define OLED_RD PCout(6)
```

```
//PB0~7，作为数据线
#define DATAOUT(x) GPIOB->ODR=(GPIOB->ODR&0xff00)|(x&0x00FF); //输出

//使用 4 线串行接口时使用
#define OLED_SCLK PBout(0)
#define OLED_SDIN PBout(1)

#define OLED_CMD  0     //写命令
#define OLED_DATA 1 //写数据
//OLED 控制用函数
void OLED_WR_Byte(u8 dat，u8 cmd);
void OLED_Display_On(void);
void OLED_Display_Off(void);
void OLED_Refresh_Gram(void);

void OLED_Init(void);
void OLED_Clear(void);
void OLED_DrawPoint(u8 x，u8 y，u8 t);
void OLED_Fill(u8 x1，u8 y1，u8 x2，u8 y2，u8 dot);
void OLED_ShowChar(u8 x，u8 y，u8 chr，u8 size，u8 mode);
void OLED_ShowNum(u8 x，u8 y，u32 num，u8 len，u8 size);
void OLED_ShowString(u8 x，u8 y，const u8 *p);
#endif
```

该部分比较简单，OLED_MODE 的定义也在这个文件里面，我们必须根据自己 OLED 模块 BS0~2 的设置（仅支持 8080 和 4 线 SPI）来确定 OLED_MODE 的值。

保存好 oled.h 之后，我们就可以在主程序里面编写我们的应用层代码了，该部分代码如下：

```
int main(void)
{
    u8 t=0;
    Stm32_Clock_Init(9); //系统时钟设置
    delay_init(72);      //延时初始化
    uart_init(72，9600);  //串口初始化
    LED_Init();                 //初始化与 LED 连接的硬件接口

    OLED_Init();            //初始化液晶
    OLED_ShowString(0，0，  "0.96' OLED TEST");
    OLED_ShowString(0，16，"ATOM@ALIENTEK");
    OLED_ShowString(0，32，"2010/06/3");

    OLED_ShowString(0，48，"ASCII:");
    OLED_ShowString(63，48，"CODE:");
    OLED_Refresh_Gram();
```

```
        t=' ';
        while(1)
        {
            OLED_ShowChar(48，48，t，16，1);//显示 ASCII 字符
            OLED_Refresh_Gram();
            t++;
            if(t>'~')t=' ';
            OLED_ShowNum(103，48，t，3，16);//显示 ASCII 字符的码值
            delay_ms(300);
            LED0=!LED0;
        }
    }
```

该部分代码用于在 OLED 上显示一些字符，然后从空格键开始不停的循环显示 ASCII 字符集，并显示该字符的 ASCII 值。注意在 test.c 文件里面包含 oled.h 头文件，同时把 oled.c 文件加入到 HARDWARE 组下，然后我们编译此工程，直到编译成功为止。

3.9.4 下载与测试

将代码下载到 MiniSTM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 OLED 模块显示如下内容：

图 3.9.4.2　OLED 显示效果

最后一行不停的显示 ASCII 字符以及其码值。通过这一节的学习，我们学会了 ALIENTEK OLED 模块的使用，在调试代码的时候，又多了一种显示信息的途径，在以后的代码中，大家可以好好利用。

3.10 TFTLCD显示实验

上一节我们介绍了 OLED 模块及其显示，但是该模块只能显示单色/双色，不能显示彩色，这一节我们将介绍 ALIENTEK TFT LCD 模块，该模块采用 TFTLCD 面板，可以显示 16 位色的真彩图片。本节将利用 TFTLCD 来显示字符和数字，并显示各种颜色。本节分为如下几个部分：

3.10.1 TFTLCD 简介

3.10.2 硬件设计

3.10.3 软件设计

3.10.4 下载与测试

3.10.1 TFTLCD 简介

TFT-LCD 即薄膜晶体管液晶显示器。其英文全称为：Thin Film Transistor-Liquid Crystal Display。TFT-LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同，它在液晶显示屏的每一个象素上都设置有一个薄膜晶体管（TFT），可有效地克服非选通时的串扰，使显示液晶屏的静态特性与扫描线数无关，因此大大提高了图像质量。TFT-LCD 也被叫做真彩液晶显示器。

上一节介绍了 OLED 模块，这一节，我们给大家介绍 ALIENTEK TFTLCD 模块，该模块有如下特点：

1，2.4’/2.8’两种大小的屏幕可选。

2，320×240 的分辨率。

3，16 位真彩显示。

4，自带触摸屏，可以用来作为控制输入。

5，通用的接口，除了 ALIENTEK MiniSTM32 开发板，该液晶模块还可以使用在优异特、STMSKY、红牛等开发板上。

本节，我们以 2.8 寸的 ALIENTEK TFTLCD 模块为例介绍，该模块采用的是显尚光电的 DST2001PH TFTLCD，DST2001PH 的控制器为 ILI9320，采用 26 万色的 TFTLCD 屏，分辨率为 320×240，采用 16 位的 80 并口。

该模块的外观图如下：

图 3.10.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如下：

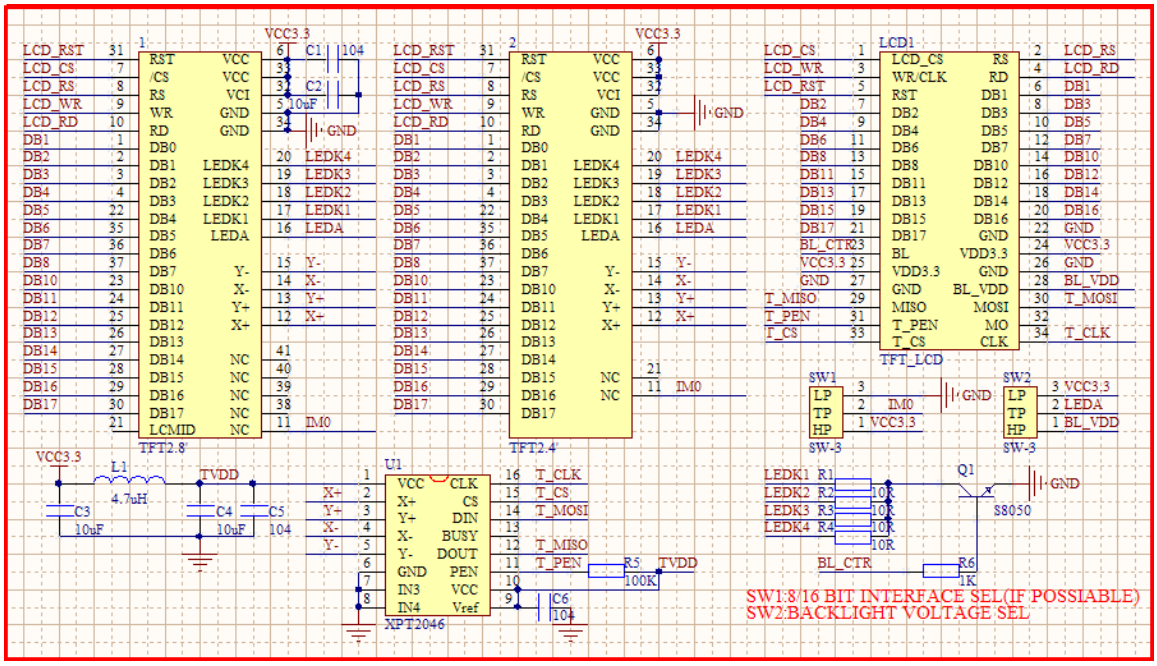


图 3.10.1.2　ALIENTEK 2.8 寸 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，接口图如下：

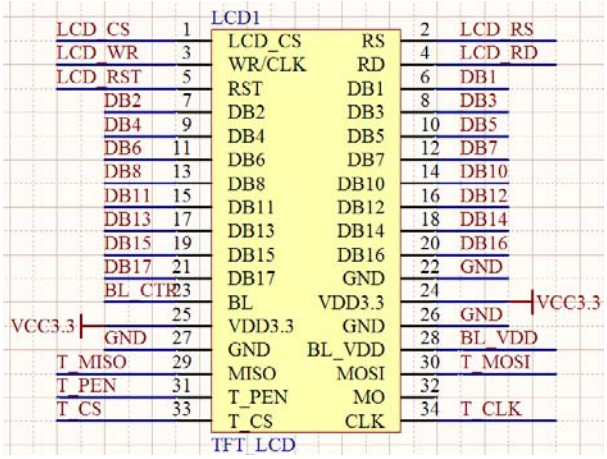


图 3.10.1.3　ALIENTEK 2.8 寸 TFTLCD 模块接口图

该接口同目前主流的几款 STM32 开发板的接口完全兼容，所以模块除了用在 ALIENTEK MiniSTM32 开发板上，也可以用在其他开发板上，当然你也可以使用其他接口一样的 LCD 模块放到我们的 ALIENTEK MiniSTM32 开发板上使用。ALIENTEK TFTLCD 模块采用 80 并口口方与外部链接，采用 16 位数据线（低了速度太慢，用彩色就没什么效果了）。该模块的 80 并口有如下一些信号线：

CS：TFTLCD 片选信号。

WR：向 TFTLCD 写入数据。

RD：从 TFTLCD 读取数据。

D[15:0]：16 位双向数据线。

RST：硬复位 TFTLCD。

RS：命令/数据标志（0，读写命令；1，读写数据）。

80 并口在上一节我们已经有详细的介绍了，这里我们就不在介绍，需要说明的是，TFTLCD 模块的 RST 信号线和 OLED 模块一样，也是直接接到 STM32 的复位脚上，并不由软件控制，

这样可以省下来一个 IO 口。另外我们还需要一个背光控制线来控制 TFTLCD 的背光。所以，我们总共需要的 IO 口数目为 21 个。

模块的控制器为 ILI9320（可能为其他型号，但是他们的设置很相似，除了初始化序列有些区别，其他大都是一摸一样的，这里仅以 9320 为例介绍），该控制器自带显存，其显存总大小为 172820（240*320*18/8），即 18 位模式（26 万色）下的显存量。模块的 16 位数据线与显寸的对应关系为 565 方式，如下图所示：

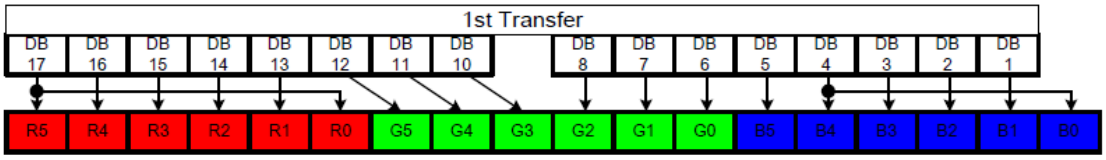


图 3.10.1.4　16 位数据与显存对应关系图

最低 5 位代表蓝色，中间 6 位为绿色，最高 5 位为红色。数值越大，表示该颜色越深。

接下来，我们介绍一下 ILI9320 的几个重要命令，因为 ILI9320 的命令很多，我们这里不可能一一介绍，有兴趣的大家可以找到 ILI9320 的 datasheet 看看。里面对这些命令有详细的介绍。这里我们要介绍的命令列表如下：

编号	指令	各位描述																命令
	HEX	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
R0	0X00	1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	OSC	打开振荡器/读取控制器型号
		1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	
R3	0X03	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0	入口模式
R7	0X07	0	0	PTDE1	PTDE0	0	0	0	BASEE	0	0	GON	DTE	CL	0	D1	D0	显示控制
R32	0X20	0	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	行地址（X）设置
R33	0X21	0	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	列地址（Y）设置
R34	0X22	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	NC	写数据到GRAM
R80	0X50	0	0	0	0	0	0	0	0	HSA7	HSA6	HSA5	HSA4	HSA3	HSA2	HSA1	HSA0	行起始地址（X）设置
R81	0X51	0	0	0	0	0	0	0	0	HEA7	HEA6	HEA5	HEA4	HEA3	HEA2	HEA1	HEA0	行结束地址（X）设置
R82	0X52	0	0	0	0	0	0	0	VSA8	VSA7	VSA6	VSA5	VSA4	VSA3	VSA2	VSA1	VSA0	列起始地址（Y）设置
R83	0X53	0	0	0	0	0	0	0	VEA8	VEA7	VEA6	VEA5	VEA4	VEA3	VEA2	VEA1	VEA0	列结束地址（Y）设置

表 3.10.1.1 ILI9320 常用命令表

R0，这个命令，有两个功能，如果对它写，则最低位为 OSC，用于开启或关闭振荡器。而如果对它读操作，则返回的是控制器的型号。这个命令最大的功能就是通过读它可以得到控制器的型号，而我们代码在知道了控制器的型号之后，可以针对不同型号的控制器，进行不同的初始化。因为 93xx 系列的初始化，其实都比较类似，我们完全可以用一个代码兼容好几个控制器。

R3，入口模式命令。我们重点关注的是 I/D0、I/D1、AM 这 3 个位，因为这 3 个位控制了屏幕的显示方向。

AM：控制 GRAM 更新方向。当 AM=0 的时候，地址以行方向更新。当 AM=1 的时候，地址以列方向更新。

I/D[1:0]：当更新了一个数据之后，根据这两个位的设置来控制地址计数器自动增加/减少 1，其关系如下图：

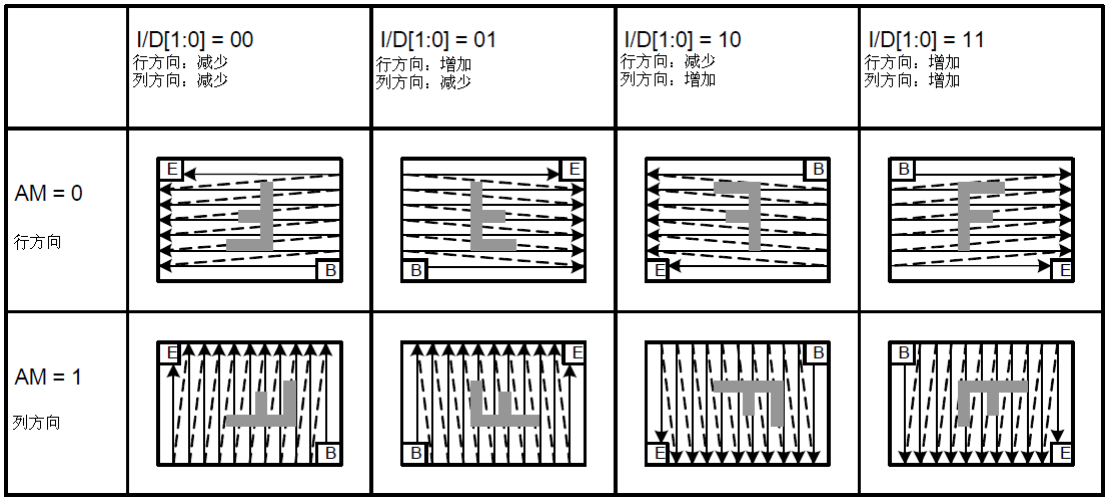


图 3.10.1.5　GRAM 显示方向设置图

通过这几个位的设置，我们就可以控制屏幕的显示方向了。

R7，显示控制命令。该命令 CL 位用来控制是 8 位彩色，还是 26 万色。为 0 时 26 万色，为 1 时八位色。D1、D0、BASEE 这三个位用来控制显示开关与否的。当全部设置为 1 的时候开启显示，全 0 是关闭。我们一般通过该命令的设置来开启或关闭显示器，以降低功耗。

R32，R33，设置 GRAM 的行地址和列地址。R32 用于设置列地址（X 坐标，0~239），R33 用于设置行地址（Y 坐标，0~319）。当我们要在某个指定点写入一个颜色的时候，先通过这两个命令设置到改点，然后写入颜色值就可以了。

R34，写数据到 GRAM 命令，当写入了这个命令之后，地址计数器才会自动的增加和减少。该命令是我们要介绍的这一组命令里面唯一的单个操作的命令，只需要写入该值就可以了，其他的都是要先写入命令编号，然后写入操作数。

R80~R83，行列 GRAM 地址位置设置。这几个命令用于设定你显示区域的大小，我们整个屏的大小为 240*320，但是有时候我们只需要在其中的一部分区域写入数据，如果用先写坐标，后写数据这样的方式来实现，则速度大打折扣。此时我们就可以通过这几个命令，在其中开辟一个区域，然后不停的丢数据，地址计数器就会根据 R3 的设置自动增加/减少，这样就不需要频繁的写地址了，大大提高了刷新的速度。

命令部分，我们就为大家介绍到这里，我们接下来看看要如何才能驱动 ALIENTEK TFTLCD 模块，这里 TFTLCD 模块的初始化和我们前面介绍的 OLED 模块的初始化框图是一样的，只是初始化代码部分不同。接下来我们也是将该模块用来来显示字符和数字。通过以上介绍，我们可以得出 TFTLCD 显示需要的相关设置步骤如下：

1）设置 STM32 与 TFTLCD 模块相连接的 IO。

这一步，先将我们与 TFTLCD 模块相连的 IO 口设置为输出，具体使用哪些 IO 口，这里需要根据连接电路以及 TFTLCD 模块的设置来确定。

2）初始化 TFTLCD 模块。

其实这里就是上和上面 OLED 模块的初始化过程差不多。通过向 TFTLCD 写入一系列的设置，来启动 TFTLCD 的显示。为后续显示字符和数字做准备。

3）通过函数将字符和数字显示到 TFTLCD 模块上。

这里就是通过我们设计的程序，将要显示的字符送到 TFTLCD 模块就可以了，这些函数将在软件设计部分向大家介绍。

通过以上三步，我们就可以使用 ALIENTEK TFTLCD 模块来显示字符和数字了， 并且可以显示各种颜色的背景。

3.10.2 硬件设计

TFTLCD 模块的电路在上一部分已有详细说明了，这里我们介绍 TFTLCD 模块与 ALIETEK MiniSTM32 开发板的连接，MiniSTM32 开发板底板的 LCD 接口和 ALIENTEK TFTLCD 模块直接可以对插，连接如下图：

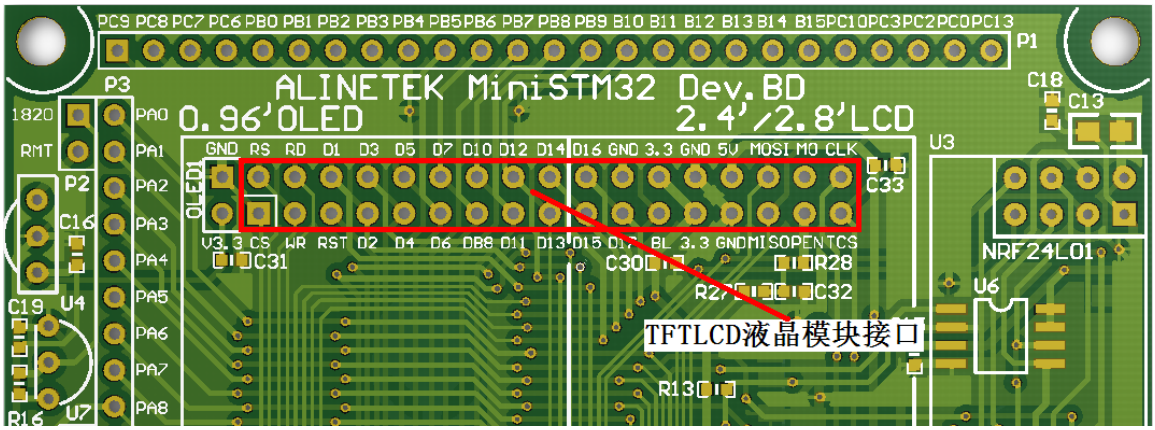


图 3.10.2.1　TFTLCD 与开发板连接示意图

图中红色线圈出来的部分就是连接 TFTLCD 模块的接口，这里在硬件上，TFTLCD 模块与 MiniSTM32 开发板的 IO 口对应关系如下：

LCD_LED 对应 PC10;

LCD_CS 对应 PC9;

LCD _RS 对应 PC8;

LCD _WR 对应 PC7;

LCD _RD 对应 PC6;

LCD _D[17:1]对应 PB[15:0];

这些线的连接，MiniSTM32 的内部已经连接好了，我们只需要将 TFTLCD 模块插上去就好了。实物连接如下图所示：

图 3.10.2.2　TFTLCD 与开发板连接实物图

3.10.3 软件设计

软件设计我们依旧在之前的工程上面增加，首先在 HARDWARE 文件夹下新建一个 LCD 的文件夹。然后打开 USER 文件夹下的工程，新建一个 ILI93xx.c 的文件和 lcd.h 的头文件，保存在 LCD 文件夹下，并将 LCD 文件夹加入头文件包含路径。

在 ILI93xx.c 里面要输入的代码比较多，我们这里就不完全列出来了，只针对几个重要的函数进行讲解。完整版的代码见光盘->ALIENTEK MiniSTM32 开发板例程->实验 10 的 ILI93xx.c 文件。

第一个是 LCD_WR_DATA 函数，该函数在 lcd.h 里面，通过宏定义的方式申明。该函数通过 80 并口向 LCD 模块写入一个 16 位的数据，使用频率是最高的，这里我们采用了宏定义的方式，以提高速度。其代码如下：

```
//写 8 位数据函数
//用宏定义,提高速度.
#ifdef LCD_FAST_IO //快速 IO
#define LCD_WR_DATA(data){\
LCD_RS_SET;\
LCD_CS_CLR;\
DATAOUT(data);\
LCD_WR_CLR;\
LCD_WR_SET;\
LCD_CS_SET;\
}
#else//正常 IO
#define LCD_WR_DATA(data){\
LCD_RS=1;\
LCD_CS=0;\
DATAOUT(data);\
LCD_WR=0;\
LCD_WR=1;\
LCD_CS=1;\
}
#endif
```

上面函数中的‘\’是 C 语言中的一个转义字符，用来连接上下文，因为宏定义只能是一个串，而当你的串过长（超过一行的时候），就需要换行了，此时就必须通过反斜杠来连接上下文。这里的‘\’正是起这个作用。在上面的函数中，我们使用了一个宏定义 LCD_FAST_IO，通过这个宏定义来控制是否使用快速 IO（下同），也就是 BRR 和 BSRR 来控制 IO 口的状态，这样可以快速的控制 IO 状态，从而提高速度，这里我们默认是开启快速 IO 的（可以让刷屏达到 28 帧左右，比正常 IO 快了一倍。

第二个是 LCD_WR_REG 函数，该函数也是通过 80 并口向 LCD 模块写入 8 位的寄存器命

令，因为该函数使用频率不是很高，我们不采用宏定义来做（宏定义占用 FLASH 较多），通过 LCD_RS 来标记是写入命令（LCD_RS=0）还是数据（LCD_RS=1）。该函数代码如下：

```
#ifdef LCD_FAST_IO //快速 IO
void LCD_WR_REG(u8 data) //写寄存器函数
{
    LCD_RS_CLR;//写地址
    LCD_CS_CLR;
    DATAOUT(data);
    LCD_WR_CLR;
    LCD_WR_SET;
    LCD_CS_SET;
}
#else//正常 IO
void LCD_WR_REG(u8 data) //写寄存器函数
{
    LCD_RS=0;//写地址
    LCD_CS=0;
    DATAOUT(data);
    LCD_WR=0;
    LCD_WR=1;
    LCD_CS=1;
}
#endif
```

既然有写命令/数据的函数，那就有读命令/数据的函数。接下来介绍 LCD_ReadReg 函数，该函数用来读取某个寄存器的值。在读某个寄存器的值之前，先要写入该寄存器的编号(命令号)，然后设置 D[15：0]为输入，再读取该改寄存器的值，读完数据之后，我们再设置 IO 口为输出。其代码如下：

```
//读寄存器
u16 LCD_ReadReg(u8 LCD_Reg)
{
    u16 t;
    LCD_WR_REG(LCD_Reg);   //写入要读的寄存器号
    GPIOB->CRL=0X88888888; //PB0-7  上拉输入
    GPIOB->CRH=0X88888888; //PB8-15 上拉输入
    GPIOB->ODR=0XFFFF;     //全部输出高
#ifdef LCD_FAST_IO //快速 IO
    LCD_RS_SET;
    LCD_CS_CLR;
    //读取数据(读寄存器时,并不需要读 2 次)
    LCD_RD_CLR;
    LCD_RD_SET;
    t=DATAIN;
    LCD_CS_SET;
```

```
#else
    LCD_RS=1;
    LCD_CS=0;
    //读取数据(读寄存器时,并不需要读 2 次)
    LCD_RD=0;
    LCD_RD=1;
    t=DATAIN;
    LCD_CS=1;
#endif
    GPIOB->CRL=0X33333333; //PB0-7   上拉输出
    GPIOB->CRH=0X33333333; //PB8-15  上拉输出
    GPIOB->ODR=0XFFFF;      //全部输出高
    return t;
}
```

第四个要介绍的是读取 GRAM 的函数，这里说明一下，为什么 OLED 模块没做读 GRAM 的函数，而这里做了。因为 OLED 模块是单色的，所需要全部 GRAM 也就 1K 个字节，而 TFTLCD 模块为彩色的，点数也比 OLED 模块多很多，以 16 位色计算，一款 320×240 的液晶，需要 320×240×2 个字节来存储颜色值，也就是也需要 150K 字节，这对任何一款 ARM 来说，都不是一个小数目了。而且我们在图形叠加的时候，可以先读回原来的值，然后写入新的值，在完成叠加后，我们又恢复原来的值。这样在做一些简单菜单的时候，是很有用的。这里我们读取 TFTLCD 模块数据的函数为 LCD_ReadRAM，该函数直接返回读到的 GRAM 值。该函数使用之前要先设置读取的 GRAM 地址，通过 LCD_SetCursor 函数来实现。LCD_ReadRAM 的代码如下：

```
//读取个某点的颜色值
//x:0~239
//y:0~319
//返回值:此点的颜色
u16 LCD_ReadPoint(u16 x,u16 y)
{
    u16 t;
    if(x>=LCD_W||y>=LCD_H)return 0;//超过了范围,直接返回
    LCD_SetCursor(x,y);
    LCD_WR_REG(R34);            //选择 GRAM 地址
    GPIOB->CRL=0X88888888; //PB0-7   上拉输入
    GPIOB->CRH=0X88888888; //PB8-15  上拉输入
    GPIOB->ODR=0XFFFF;          //全部输出高
#ifdef LCD_FAST_IO //快速 IO
    LCD_RS_SET;
    LCD_CS_CLR;
    //读取数据(读 GRAM 时,需要读 2 次)
    LCD_RD_CLR;
    LCD_RD_SET;
    //dummy READ
```

```
        LCD_RD_CLR;
        LCD_RD_SET;
        t=DATAIN;
        LCD_CS_SET;
    #else
        LCD_RS=1;
        LCD_CS=0;
        //读取数据(读 GRAM 时,需要读 2 次)
        LCD_RD=0;
        LCD_RD=1;
        //dummy READ
        LCD_RD=0;
        LCD_RD=1;
        t=DATAIN;
        LCD_CS=1;
    #endif
        GPIOB->CRL=0X33333333; //PB0-7   上拉输出
        GPIOB->CRH=0X33333333; //PB8-15  上拉输出
        GPIOB->ODR=0XFFFF;      //全部输出高
        if(DeviceCode==0X4531)return t;//4531 驱动 IC
        else return LCD_BGR2RGB(t);
    }
```

在 LCD_ReadPoint 函数中，我们使用了一个全局变量 DeviceCode，该变量用来记录 LCD 的 ID，因为不是所有的驱动 IC，读出来颜色的数据格式都相同。9320 等型号的驱动 IC，读出来的颜色，和写入的颜色是有区别的，写入的时候为 RGB565，而读出来的时候，变成了 BGR565 了，所以，它们需要做一个颜色转换，这个工作由 LCD_BGR2RGB 函数来完成。但是 4531 这款驱动 IC，写进去的颜色值和读出来的是一致的，所以它就不需要颜色转换了。这里的 DeviceCode 是一个全局变量，用来记录 LCD 的驱动 IC 型号。

第五个要介绍的函数为 LCD_SetCursor 函数，该函数用来设置坐标的，其代码如下：

```
void LCD_SetCursor(u16 Xpos， u16 Ypos)
{
    LCD_WriteReg(R32， Xpos);
    LCD_WriteReg(R33， Ypos);
}
```

（Xpos，Ypos）为要写入或读取的像素点坐标，这里的设置其实就是利用了 R32，R33 这两个命令，在前面寄存器介绍的时候已经介绍过了。

第六个要介绍的是画点函数 LCD_DrawPoint，该函数带 2 个参数（x，y）就代表 TFTLCD 上的坐标，x 的范围为 0~239，代表横坐标。Y 的范围为 0~319，代表纵坐标，写入点的颜色是根据全局变量 POINT_COLOR 来决定的。该函数代码如下：

```
void LCD_DrawPoint(u16 x，u16 y)
{
    LCD_SetCursor(x，y);//设置光标位置
    LCD_WR_REG(R34);//开始写入 GRAM
```

```
    LCD_WR_DATA(POINT_COLOR);
}
```

这里顺带介绍一下 2 个全局变量 POINT_COLOR 和 BACK_COLOR，这两个 16 位的全局变量，第一个 POINT_COLOR 代表要写入的点的颜色，而 BACK_COLOR 则代表背景色。

第七个要介绍的函数是字符显示函数 LCD_ShowChar，该函数同前面 OLED 模块的自负显示函数差不多，但是这里的字符显示函数多了 1 个功能，就是可以以叠加方式显示，或者以非叠加方式显示。叠加方式显示多用于在显示的图片上再显示字符。非叠加方式一般用于普通的显示。

这里要注意的是 TFTLCD 模块的 ASCII 字符集取模方式与 OLED 模块的 ASCII 字符集取模方式不一样，这里的字符写入函数，采取的开辟窗口的形式来做的，并不是采用画点函数来做的。开辟窗口我们用的是前面介绍的 R80~R83 四个命令来做的，具体请参考前面的命令介绍部分。这里，我们的 ASCII 字符集取模还是采用 PCtoLCD2002 完美版来做，只是取模方式设置成如下方式：

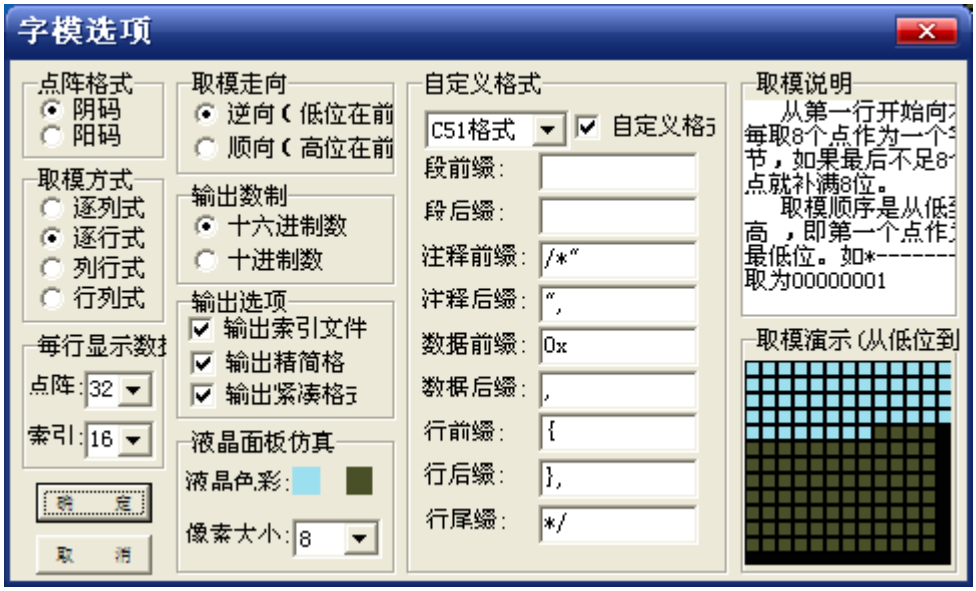


图 3.10.3.1　TFTLCD 显示字符取模方式设置

LCD_ShowChar 的代码如下：

```
void LCD_ShowChar(u8 x，u16 y，u8 num，u8 size，u8 mode)
{
#define MAX_CHAR_POSX 232
#define MAX_CHAR_POSY 304
    u8 temp;
    u8 pos，t;
    if(x>MAX_CHAR_POSX||y>MAX_CHAR_POSY)return;
    //设置窗口
    LCD_WriteReg(R80，x);                  //水平方向 GRAM 起始地址
    LCD_WriteReg(R81，x+(size/2-1));//水平方向 GRAM 结束地址
    LCD_WriteReg(R82，y);                  //垂直方向 GRAM 起始地址
    LCD_WriteReg(R83，y+size-1);       //垂直方向 GRAM 结束地址
    LCD_SetCursor(x，y);                   //设置光标位置
    LCD_WriteRAM_Prepare();            //开始写入 GRAM
```

```
        num=num-' ';//得到偏移后的值
        if(!mode) //非叠加方式
        {
            for(pos=0;pos<size;pos++)
            {
                if(size==12)temp=asc2_1206[num][pos];//调用 1206 字体
                else temp=asc2_1608[num][pos];          //调用 1608 字体
                for(t=0;t<size/2;t++)
                {
                    if(temp&0x01)
                    {
                        LCD_WR_DATA(POINT_COLOR);
                    }else LCD_WR_DATA(BACK_COLOR);
                    temp>>=1;
                }
            }
        }else//叠加方式
        {
            for(pos=0;pos<size;pos++)
            {
                if(size==12)temp=asc2_1206[num][pos];//调用 1206 字体
                else temp=asc2_1608[num][pos];          //调用 1608 字体
                for(t=0;t<size/2;t++)
                {
                    if(temp&0x01)LCD_DrawPoint(x+t，y+pos);//画一个点
                    temp>>=1;
                }
            }
        }
        //恢复窗体大小
        LCD_WriteReg(R80，  0x0000); //水平方向 GRAM 起始地址
        LCD_WriteReg(R81，  0x00EF); //水平方向 GRAM 结束地址
        LCD_WriteReg(R82，  0x0000); //垂直方向 GRAM 起始地址
        LCD_WriteReg(R83，  0x013F); //垂直方向 GRAM 结束地址
    }
```

在 LCD_ShowChar 函数里面，我们采用开辟窗口的方式来做字符显示，可以提高写入速度，因为这样我们可以省掉每次都设置坐标，而是直接写入数据，LCD 自动完成坐标的递增。这样的写入方式，在绘图的时候，尤为见效，可以很大程度上提高图片的刷新速度。

最后，我们再介绍一下 TFTLCD 模块的初始化函数 LCD_Init，该函数先初始化 STM32 与 TFTLCD 连接的 IO 口，然后读取控制芯片的型号，根据控制 IC 的型号执行不同的初始化代码，其简化代码如下：

```
    void LCD_Init(void)
    {
```

```
RCC->APB2ENR|=1<<3;//先使能外设 PORTB 时钟
RCC->APB2ENR|=1<<4;//先使能外设 PORTC 时钟
RCC->APB2ENR|=1<<0;       //开启辅助时钟
JTAG_Set(SWD_ENABLE);    //开启 SWD
//PORTC6~10 复用推挽输出
GPIOC->CRH&=0XFFFFF000;
GPIOC->CRH|=0X00000333;
GPIOC->CRL&=0X00FFFFFF;
GPIOC->CRL|=0X33000000;
GPIOC->ODR|=0X07C0;
//PORTB  推挽输出
GPIOB->CRH=0X33333333;
GPIOB->CRL=0X33333333;
GPIOB->ODR=0XFFFF;
delay_ms(50); // delay 50 ms
LCD_WriteReg(0x0000,0x0001);
delay_ms(50); // delay 50 ms
DeviceCode = LCD_ReadReg(0x0000);
printf(" LCD ID:%x\n",DeviceCode);//打印 LCD ID
if(DeviceCode==0x9325||DeviceCode==0x9328)//ILI9325
{
      ……//9325/9328 初始化代码
}
else if(DeviceCode==0x9320||DeviceCode==0x9300)
{
      ……//9320/9300 初始化代码
} else if(DeviceCode==0x5408)
{
      ……//5408 初始化代码
}
else if(DeviceCode==0x1505)
{
      ……//1505 初始化代码
}
else if(DeviceCode==0x8989)
{
      ……//8989 初始化代码
} else if(DeviceCode==0x4531)
{
      ……//4531 初始化代码
}
LCD_LED=1;//点亮背光
LCD_Clear(WHITE);
```

```
}
```

保存 ILI93xx.c，并将该代码加入到 HARDWARE 组下。在介绍完了 ILI93xx.c 的内容之后，然后我们在 lcd.h 里面输入如下内容：

```
#ifndef __LCD_H
#define __LCD_H
#include "sys.h"
#include "stdlib.h"
//2.4/2.8 寸 TFT 液晶驱动
//支持驱动 IC 型号包括:ILI9325/RM68021/ILI9320/LGDP4531/SPFD5408 等
//ALIENTEK Mini STM32 开发板
//TFTLCD  驱动代码

//TFTLCD 部分外要调用的函数
extern u16   POINT_COLOR;//默认红色
extern u16   BACK_COLOR; //背景颜色.默认为白色
//定义 LCD 的尺寸
#define LCD_W 240
#define LCD_H 320
//-----------------LCD 端口定义----------------
#define   LCD_LED PCout(10) //LCD 背光              PC10
//如果使用快速 IO，则定义下句，如果不使用，则去掉即可!
//使用快速 IO，刷屏速率可以达到 28 帧每秒!
//普通 IO，只能 14 帧每秒!
#define LCD_FAST_IO   //快速 IO
#ifdef LCD_FAST_IO //快速 IO
#define   LCD_CS_SET   GPIOC->BSRR=1<<9     //片选端口          PC9
#define   LCD_RS_SET     GPIOC->BSRR=1<<8     //数据/命令         PC8
#define   LCD_WR_SET     GPIOC->BSRR=1<<7     //写数据            PC7
#define   LCD_RD_SET     GPIOC->BSRR=1<<6     //读数据            PC6

#define   LCD_CS_CLR   GPIOC->BRR=1<<9      //片选端口          PC9
#define   LCD_RS_CLR     GPIOC->BRR=1<<8      //数据/命令         PC8
#define   LCD_WR_CLR     GPIOC->BRR=1<<7      //写数据            PC7
#define   LCD_RD_CLR     GPIOC->BRR=1<<6      //读数据            PC6
#else //慢速 IO
#define   LCD_CS PCout(9)   //片选端口              PC9
#define   LCD_RS PCout(8)   //数据/命令             PC8
#define   LCD_WR      PCout(7)   //写数据           PC7
#define   LCD_RD PCout(6)   //读数据                PC6
#endif
//PB0~15,作为数据线
#define DATAOUT(x) GPIOB->ODR=x; //数据输出
#define DATAIN         GPIOB->IDR;     //数据输入
```

```
//////////////////////////////////////////////////////////////////
//画笔颜色
#define WHITE                0xFFFF
…… //省略部分颜色定义
#define LBBLUE               0X2B12 //浅棕蓝色(选择条目的反色)
extern u16 BACK_COLOR, POINT_COLOR ;
void LCD_Init(void);
void LCD_DisplayOn(void);
void LCD_DisplayOff(void);
void LCD_Clear(u16 Color);
void LCD_SetCursor(u16 Xpos, u16 Ypos);
void LCD_DrawPoint(u16 x,u16 y);//画点
u16    LCD_ReadPoint(u16 x,u16 y); //读点
void Draw_Circle(u16 x0,u16 y0,u8 r);
void LCD_DrawLine(u16 x1, u16 y1, u16 x2, u16 y2);
void LCD_DrawRectangle(u16 x1, u16 y1, u16 x2, u16 y2);
void LCD_Fill(u16 xsta,u16 ysta,u16 xend,u16 yend,u16 color);
void LCD_ShowChar(u16 x,u16 y,u8 num,u8 size,u8 mode);//显示一个字符
void LCD_ShowNum(u16 x,u16 y,u32 num,u8 len,u8 size);    //显示一个数字
void LCD_Show2Num(u16 x,u16 y,u16 num,u8 len,u8 size,u8 mode);//显示 2 个数字
void LCD_ShowString(u16 x,u16 y,const u8 *p);          //显示一个字符串,16 字体

void LCD_WriteReg(u8 LCD_Reg, u16 LCD_RegValue);
u16 LCD_ReadReg(u8 LCD_Reg);
void LCD_WriteRAM_Prepare(void);
void LCD_WriteRAM(u16 RGB_Code);
u16 LCD_ReadRAM(void);

//写 8 位数据函数
//用宏定义,提高速度.
#ifdef LCD_FAST_IO //快速 IO
#define LCD_WR_DATA(data){\
LCD_RS_SET;\
LCD_CS_CLR;\
DATAOUT(data);\
LCD_WR_CLR;\
LCD_WR_SET;\
LCD_CS_SET;\
}
#else//正常 IO
#define LCD_WR_DATA(data){\
LCD_RS=1;\
LCD_CS=0;\
```

```
DATAOUT(data);\
LCD_WR=0;\
LCD_WR=1;\
LCD_CS=1;\
}
#endif
//9320/9325 LCD 寄存器
#define R0                  0x00
…… //省略部分寄存器定义
#define R229                0xE5
#endif
```

这段代码用两种方式来控制 IO 口，当使用快速模式来控制的时候，我们可以有效提升速度，是本手册推荐使用的方式。另外这段代码对颜色和驱动器的寄存器进行了很多宏定义，限于篇幅考虑，我们没有完全贴出来，省略了其中绝大部分。此部分我们就不多说了。接下来，我们在 test.c 里面修改 main 函数如下：

```
int main(void)
{
    u8 x=0;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();
    LCD_Init();
    POINT_COLOR=RED;
    while(1)
    {
        switch(x)
        {
            case 0:LCD_Clear(WHITE);break;
            case 1:LCD_Clear(BLACK);break;
            case 2:LCD_Clear(BLUE);break;
            case 3:LCD_Clear(RED);break;
            case 4:LCD_Clear(MAGENTA);break;
            case 5:LCD_Clear(GREEN);break;
            case 6:LCD_Clear(CYAN);break;

            case 7:LCD_Clear(YELLOW);break;
            case 8:LCD_Clear(BRRED);break;
            case 9:LCD_Clear(GRAY);break;
            case 10:LCD_Clear(LGRAY);break;
            case 11:LCD_Clear(BROWN);break;
        }
        POINT_COLOR=RED;
```

```
            LCD_ShowString(30,50,"Mini STM32 ^_^");
            LCD_ShowString(30,70,"2.4'/2.8' TFTLCD TEST");
            LCD_ShowString(30,90,"ATOM@ALIENTEK");
            LCD_ShowString(30,110,"2010/12/30");
            x++;
            if(x==12)x=0;
            LED0=!LED0;
            delay_ms(1000);
        }
    }
```

该部分代码将显示一些固定的字符，然后不停的切换背景颜色，每 1s 切换一次。而 LED0 也会不停的闪烁，指示程序已经在运行了。这里我们因为是在前面 OLED 模块驱动代码的基础上修改的，而两个模块的代码都有各自的 font.h，这两个 font.h 并不能通用，所以我们要先删除 oled.c。然后把 ILI93xx.c 加入到 HARDWARE 组下，在 test.c 里面加入 lcd.h 头文件。编译此工程，在编译完之后，我们开始下一步工作。

3.10.4 下载与测试

将代码下载到 MiniSTM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块显示如下内容：

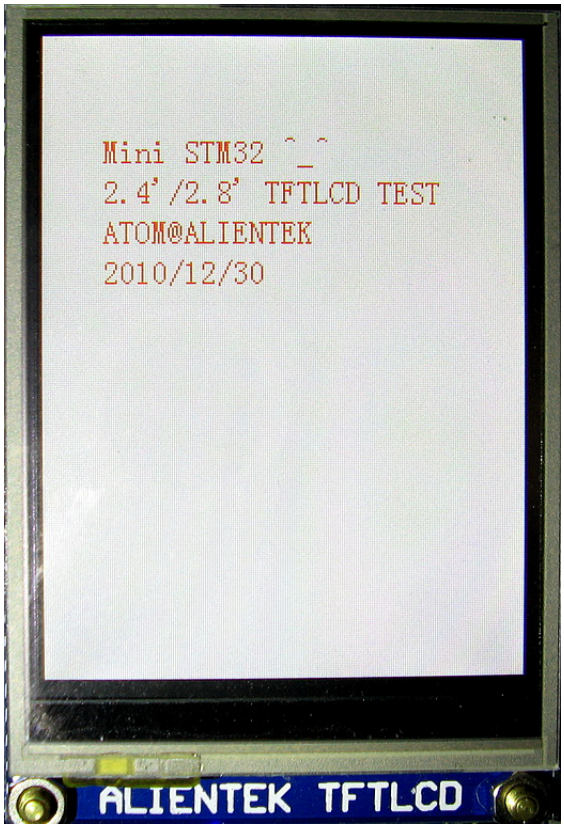


图 3.10.4.1 TFTLCD 显示效果图

我们可以看到屏幕的背景是不停切换的。

3.11 RTC实时时钟实验

前面两节我们介绍了两款液晶模块，这一节我们将介绍 ALIENTEK MiniSTM32 开发板的实时时钟（RTC）。本节将利用 ALIENTEK 2.8’的 TFTLCD 模块来显示日期和时间，本节将顺带向大家介绍 BKP 的使用。本节分为如下几个部分：

3.11.1 STM32 RTC 时钟简介

3.11.2 硬件设计

3.11.3 软件设计

3.11.4 下载与测试

3.11.1 STM32 RTC 时钟简介

实时时钟（RTC）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)是在后备区域，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。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前， 先要取消备份区域（BKP）写保护。

RTC 由两个主要部分组成(见下图)。第一部分(APB1 接口)用来和 APB1 总线相连。此单元还包含一组 16 位寄存器，可通过 APB1 总线对其进行读写操作。APB1 接口由 APB1 总线时钟驱动，用来与 APB1 总线接口。

另一部分(RTC 核心)由一组可编程计数器组成，分成两个主要模块。第一个模块是 RTC 的预分频模块，它可编程产生最长为 1 秒的 RTC 时间基准 TR_CLK。RTC 的预分频模块包含了一个 20 位的可编程分频器(RTC 预分频器)。如果在 RTC_CR 寄存器中设置了相应的允许位，则在每个 TR_CLK 周期中 RTC 产生一个中断(秒中断)。第二个模块是一个 32 位的可编程计数器，可被初始化为当前的系统时间，一个 32 位的时钟计数器，按秒钟计算，可以记录 4294967296 秒，约合 136 年左右，作为一般应用，这已经是足够了的。

RTC 还有一个闹钟寄存器 RTC_ALR，用于产生闹钟。系统时间按 TR_CLK 周期累加并与存储在 RTC_ALR 寄存器中的可编程时间相比较，如果 RTC_CR 控制寄存器中设置了相应允许位，比较匹配时将产生一个闹钟中断。

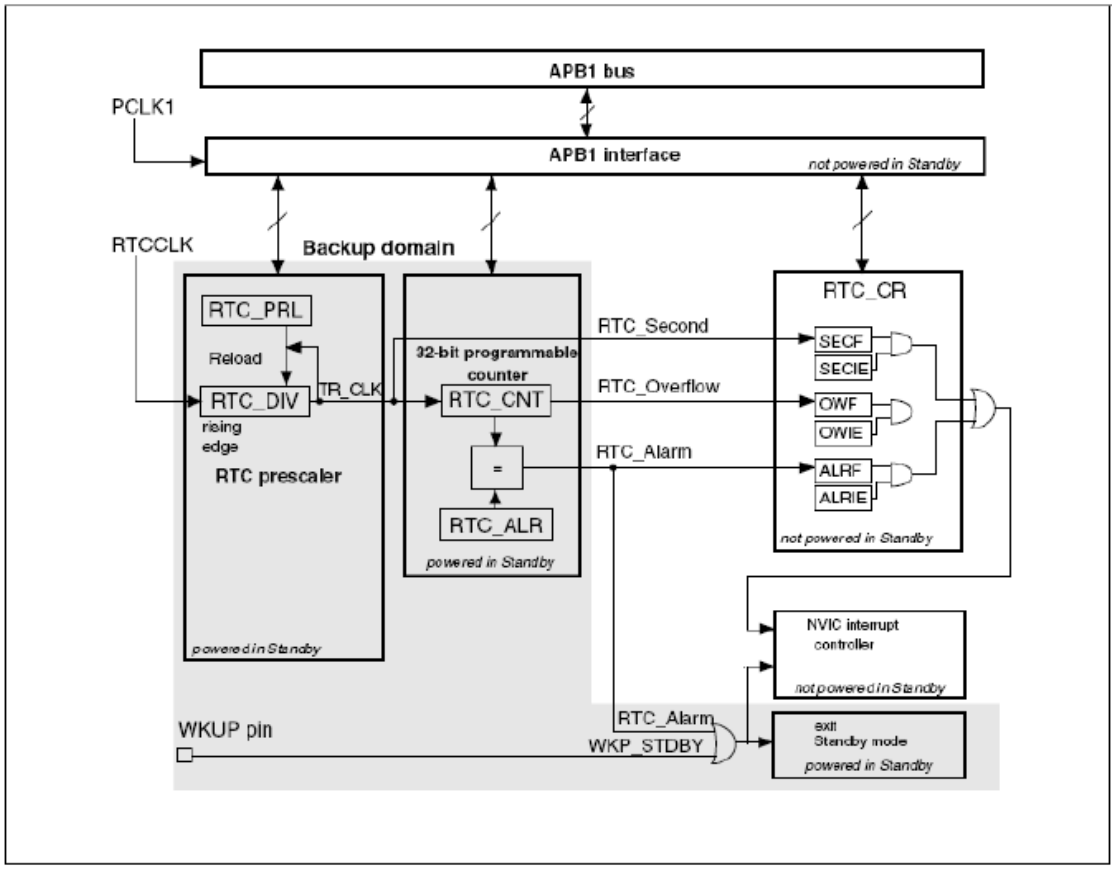


图 3.11.1.1 RTC 简化框图

RTC 内核完全独立于 RTC APB1 接口，而软件是通过 APB1 接口访问 RTC 的预分频值、计数器值和闹钟值的。但是相关可读寄存器只在 RTC APB1 时钟进行重新同步的 RTC 时钟的上升沿被更新，RTC 标志也是如此。这就意味着，如果 APB1 接口刚刚被开启之后，在第一次的内部寄存器更新之前，从 APB1 上都处的 RTC 寄存器值可能被破坏了（通常读到 0）。因此，若在读取 RTC 寄存器曾经被禁止的 RTC APB1 接口，软件首先必须等待 RTC_CRL 寄存器的 RSF 位（寄存器同步标志位，bit3）被硬件置 1。

接下来，我们介绍一下 RTC 相关的几个寄存器。首先要介绍的是 RTC 的控制寄存器，RTC 总共有 2 个控制寄存器 RTC_CRH 和 RTC_CRL，两个都是 16 位的。RTC_CRH 的各位描如下图所示：

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 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 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 0：屏蔽(不允许)秒中断 1：允许秒中断

图 3.11.1.2 寄存器 RTC_CRH 各位描述

该寄存器用来控制中断的，我们这一节将要用到秒钟中断，所以在该寄存器必须设置最低位为 1，以允许秒钟中断。我们再看看 RTC_CRL 寄存器。该寄存器各位描述如下图所示：

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 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 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 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 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 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。

图 3.11.1.3 寄存器 RTC_CRL 各位描述

这一节我们用到的是该寄存器的 0、3~5 这几个位，第 0 位是秒钟标志位，我们在进入闹钟中断的时候，通过判断这位来决定是不是发生了秒钟中断。然后必须通过软件将该位清零（写 0）。第 3 位为寄存器同步标志位，我们在修改控制寄存器 RTC_CRH/CRL 之前，必须先判断该位，是否已经同步了，如果没有则等待同步，在没同步的情况下修改 RTC_CRH/CRL 的值是不行的。第 4 位为配置标位，在软件修改 RTC_CNT/RTC_ALR/RTC_PRL 的值的时候，必须先软件置位该位，以允许进入配置模式。第 5 位为 RTC 操作位，该位由硬件操作，软件只读。通过该位可以判断上次对 RTC 寄存器的操作是否完成，如果没有，我们必须等待上一次操作结束才能开始下一次操作。

第二个要介绍的寄存器是 RTC 预分频装载寄存器，也有 2 个寄存器组成，RTC_PRLH 和 RTC_PRLL。这两个寄存器用来配置 RTC 时钟的分频数的，比如我们使用外部 32.768K 的晶振作为时钟的输入频率，那么我们要设置这两个寄存器的值为 32767，以得到一秒钟的计数频率。RTC_PRLH 的各位描述如下：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留												PRL[19:16]			
												w	w	w	w

位15:6	保留，被硬件强制为0。
位3:0	**PRL[19:16]**：RTC预分频装载值高位 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：不推荐使用0值，否则无法正确的产生RTC中断和标志位。

图 3.11.1.4 寄存器 RTC_PRLH 各位描述

RTC_PRLL 的各位描述如下：

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

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： $f_{TR\_CLK} = f_{RTCCLK}/(PRL[19:0]+1)$

图 3.11.1.5 寄存器 RTC_PRLL 各位描述

在介绍完这两个寄存器之后，我们介绍 RTC 预分频器余数寄存器，该寄存器也有 2 个寄存器组成 RTC_DIVH 和 RTC_DIVL，这两个寄存器的作用就是用来获得比秒钟更为准确的时钟，比如可以得到 0.1 秒，或者 0.01 秒等。该寄存器的值自减的，用于保存还需要多少时钟周期获得一个秒信号。在一次秒钟更新后，由硬件重新装载。这两个寄存器和 RTC 预分频装载寄存器的各位是一样的，这里我们就不列出来了。

接着要介绍的是 RTC 最重要的寄存器，RTC 计数器寄存器。该寄存器由 2 个 16 位的寄存器组成 RTC_CNTH 和 RTC_CNTL，总共 32 位，用来记录秒钟值（一般情况下）。此两个计数器也比较简单，我们也不多说了。注意一点，在修改这个寄存器的时候要先进入配置模式。

最后我们介绍 RTC 部分的最后一个寄存器，RTC 闹钟寄存器，该寄存器也是由 2 个 16 为的寄存器组成 RTC_ALRH 和 RTC_ALRL。总共也是 32 位，用来标记闹钟产生的时间（以秒为单位），如果 RTC_CNT 的值与 RTC_ALR 的值相等，并使能了中断的话，会产生一个闹钟中断。该寄存器的修改也要进入配置模式才能进行。

因为我们使用到备份寄存器来存储 RTC 的相关信息（我们这里主要用来标记时钟是否已经经过了配置），我们这里顺便介绍一下 STM32 的备份寄存器。

备份寄存器是 42 个 16 位的寄存器（大容量产品才有，我们的 MiniSTM32 开发板使用的是 STM32F103RBT6，属于小容量产品，只有 10 个 16 为的寄存器），可用来存储 84 个字节的用户应用程序数据。他们处在备份域里，当 VDD 电源被切断，他们仍然由 VBAT 维持供电。当系统在待机模式下被唤醒，或系统复位或电源复位时，他们也不会被复位。

此外，BKP 控制寄存器用来管理侵入检测和 RTC 校准功能。

复位后，对备份寄存器和 RTC 的访问被禁止，并且备份域被保护以防止可能存在的意外的写操作。执行以下操作可以使能对备份寄存器和 RTC 的访问：

1）通过设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位来打开电源和后备接口的时钟

2）电源控制寄存器(PWR_CR)的 DBP 位来使能对后备寄存器和 RTC 的访问。

我们一般用 BKP 来存储 RTC 的校验值或者记录一些重要的数据，相当于一个 EEPROM，不过这个 EEPROM 并不是真正的 EEPROM，而是需要电池来维持它的数据。关于 BKP 的详细

介绍请看《STM32 参考手册》的第 47 页，5.1 一节。

最后，我们还要介绍一下备份区域控制寄存器 RCC_BDCR。该寄存器的个位描述如下：

位31:17	保留，始终读为0。
位16	**BDRST**：备份域软件复位 由软件置'1'或清'0' 0：复位未激活； 1：复位整个备份域。
位15	**RTCEN**：RTC时钟使能 由软件置'1'或清'0' 0：RTC时钟关闭； 1：RTC时钟开启。
位14:10	保留，始终读为0。
位9:8	**RTCSEL[1:0]**：RTC时钟源选择 由软件设置来选择RTC时钟源。一旦RTC时钟源被选定，直到下次后备域被复位，它不能在被改变。可通过设置BDRST位来清除。 00：无时钟； 01：LSE振荡器作为RTC时钟； 10：LSI振荡器作为RTC时钟； 11：HSE振荡器在128分频后作为RTC时钟。
位7:3	保留，始终读为0。
位2	**LSEBYP**：外部低速时钟振荡器旁路 在调试模式下由软件置'1'或清'0'来旁路LSE。只有在外部32kHz振荡器关闭时，才能写入该位 0：LSE时钟未被旁路； 1：LSE时钟被旁路。
位1	**LSERDY**：外部低速LSE就绪 由硬件置'1'或清'0'来指示是否外部32kHz振荡器就绪。在LSEON被清零后，该位需要6个外部低速振荡器的周期才被清零。 0：外部32kHz振荡器未就绪； 1：外部32kHz振荡器就绪。
位0	**LSEON**：外部低速振荡器使能 由软件置'1'或清'0' 0：外部32kHz振荡器关闭； 1：外部32kHz振荡器开启。

图 3.11.1.6 寄存器 RCC_ BDCR 各位描述

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的，所以我们在 RTC 操作之前先要通过这个寄存器选择 RTC 的时钟源，然后才能开始其他的操作。

寄存器介绍就给大家介绍到这里了，我们下面来看看要经过哪几个步骤的配置才能使 RTC 正常工作。RTC 正常工作的一般配置步骤如下：

1）使能电源时钟和备份区域时钟。

前面已经介绍了，我们要访问 RTC 和备份区域就必须先使能电源时钟和备份区域时钟。这个通过 RCC_APB1ENR 寄存器来设置。

2）取消备份区写保护。

要向备份区域写入数据，就要先取消备份区域写保护（写保护在每次硬复位之后被使能），

否则是无法向备份区域写入数据的。我们需要用到向备份区域写入一个字节，来标记时钟已经配置过了，这样避免每次复位之后重新配置时钟。

3）复位备份区域，开启外部低速振荡器。

在取消备份区域写保护之后，我们可以先对这个区域复位，以清除前面的设置，当然这个操作不要每次都执行，因为备份区域的复位将导致之前存在的数据丢失，所以要不要复位，要看情况而定。然后我们使能外部低速振荡器，注意这里一般要先判断 RCC_BDCR 的 LSERDY 位来确定低速振荡器已经就绪了才开始下面的操作。

4）选择 RTC 时钟，并使能。

这里我们将通过 RCC_BDCR 的 RTCSEL 来选择选择外部 LSI 作为 RTC 的时钟。然后通过 RTCEN 位使能 RTC 时钟。

5）设置 RTC 的分频，以及配置 RTC 时钟。

在开启了 RTC 时钟之后，我们要做的就是设置 RTC 时钟的分频数，通过 RTC_PRLH 和 RTC_PRLL 来设置，然后等待 RTC 寄存器操作完成，并同步之后，设置秒钟中断。然后设置 RTC 的允许配置位(RTC_CRH 的 CNF 位)，设置时间(其实就是设置 RTC_CNTH 和 RTC_CNTL 两个寄存器)。

6）更新配置，设置 RTC 中断。

在设置完时钟之后，我们将配置更新，这里还是通过 RTC_CRH 的 CNF 来实现。在这之后我们在备份区域 BKP_DR1 中写入 0X5050 代表我们已经初始化过时钟了，下次开机（或复位）的时候，先读取 BKP_DR1 的值，然后判断是否是 0X5050 来决定是不是要配置。接着我们配置 RTC 的秒钟中断，并进行分组。

7）编写中断服务函数。

最后，我们要编写中断服务函数，在秒钟中断产生的时候，读取当前的时间值，并显示到 TFTLCD 模块上。

通过以上几个步骤，我们就完成了对 RTC 的配置，并通过秒钟中断来更新时间。接下来我们将进行下一步过程。

3.11.2 硬件设计

这一节，我们使用 TFTLCD 模块作为输出显示，和上一节的电路完全一样，这里我们就不多说了，但是这里我们如果想让时间在断电后还可以继续走，那么必须确保 ALIENTEK MiniSTM32 的电池有电。ALIENTEK MiniSTM32 开发板板都是配电池的，这颗电池理论上可以使用 16 年，足够使用的了。

3.11.3 软件设计

找到上一节的工程，首先在 HARDWARE 文件夹下新建一个 RTC 的文件夹。然后打开 USER 文件夹下的工程，新建一个 rtc.c 的文件和 rtc.h 的头文件，保存在 RTC 文件夹下，并将 RTC 文件夹加入头文件包含路径。

打开 rtc.c，输入如下代码：

```
#include "sys.h"
```

```
#include "rtc.h"
#include "delay.h"
#include "usart.h"
tm timer;//时钟结构体
//实时时钟配置
//初始化 RTC 时钟,同时检测时钟是否工作正常
//BKP->DR1 用于保存是否第一次配置的设置
//返回 0:正常；其他:错误代码
u8 RTC_Init(void)
{
    //检查是不是第一次配置时钟
    u8 temp=0;
    if(BKP->DR1!=0X5050)//第一次配置
    {
        RCC->APB1ENR|=1<<28;        //使能电源时钟
        RCC->APB1ENR|=1<<27;        //使能备份时钟
        PWR->CR|=1<<8;              //取消备份区写保护
        RCC->BDCR|=1<<16;           //备份区域软复位
        RCC->BDCR&=~(1<<16);        //备份区域软复位结束
        RCC->BDCR|=1<<0;            //开启外部低速振荡器
        while((!(RCC->BDCR&0X02))&&temp<250)//等待外部时钟就绪
        {
            temp++;
            delay_ms(10);
        };
        if(temp>=250)return 1;//初始化时钟失败,晶振有问题

        RCC->BDCR|=1<<8; //LSI 作为 RTC 时钟
        RCC->BDCR|=1<<15;//RTC 时钟使能
        while(!(RTC->CRL&(1<<5)));//等待 RTC 寄存器操作完成
        while(!(RTC->CRL&(1<<3)));//等待 RTC 寄存器同步
        RTC->CRH|=0X01;             //允许秒中断
        while(!(RTC->CRL&(1<<5)));//等待 RTC 寄存器操作完成
        RTC->CRL|=1<<4;             //允许配置
        RTC->PRLH=0X0000;
        RTC->PRLL=32767;    //时钟周期设置(有待观察,看是否跑慢了?)理论值：32767
        Auto_Time_Set();
        //RTC_Set(2009,12,2,10,0,55);   //设置时间
        RTC->CRL&=~(1<<4);          //配置更新
        while(!(RTC->CRL&(1<<5)));    //等待 RTC 寄存器操作完成

        BKP->DR1=0X5050;
        //BKP_Write(1,0X5050);;//在寄存器 1 标记已经开启了
```

```
            //printf("FIRST TIME\n");
        }else//系统继续计时
        {
            while(!(RTC->CRL&(1<<3)));//等待 RTC 寄存器同步
            RTC->CRH|=0X01;                 //允许秒中断
            while(!(RTC->CRL&(1<<5)));//等待 RTC 寄存器操作完成
            //printf("OK\n");
        }
        MY_NVIC_Init(0,0,RTC_IRQChannel,2);//RTC,G2,P2,S2.优先级最低
        RTC_Get();//更新时间
        return 0; //ok
}
//RTC 中断服务函数
//RTC 时钟中断
//每秒触发一次
void RTC_IRQHandler(void)
{
        if(RTC->CRL&0x0001)//秒钟中断
        {
            RTC_Get();//更新时间
            //printf("CRL:%d\n"，RTC->CRL);
        }
        if(RTC->CRL&0x0002)//闹钟中断
        {
            //printf("Alarm!\n");
            RTC->CRL&=~(0x0002);//清闹钟中断
            //闹钟处理
        }
        RTC->CRL&=0X0FFA;               //清除溢出，秒钟中断标志
        while(!(RTC->CRL&(1<<5)));//等待 RTC 寄存器操作完成

}
//判断是否是闰年函数
//月份    1  2  3  4  5  6  7  8  9  10 11 12
//闰年    31 29 31 30 31 30 31 31 30 31 30 31
//非闰年  31 28 31 30 31 30 31 31 30 31 30 31
//输入:年份
//输出:该年份是不是闰年.1，是.0，不是
u8 Is_Leap_Year(u16 year)
{
        if(year%4==0) //必须能被 4 整除
        {
            if(year%100==0)
```

```
            {
                if(year%400==0)return 1;//如果以 00 结尾，还要能被 400 整除
                else return 0;
            }else return 1;
        }else return 0;
}
//设置时钟
//把输入的时钟转换为秒钟
//以 1970 年 1 月 1 日为基准
//1970~2099 年为合法年份
//返回值:0，成功;其他:错误代码.
//月份数据表
u8 const table_week[12]={0，3，3，6，1，4，6，2，5，0，3，5}; //月修正数据表
//平年的月份日期表
const u8 mon_table[12]={31，28，31，30，31，30，31，31，30，31，30，31};
u8 RTC_Set(u16 syear，u8 smon，u8 sday，u8 hour，u8 min，u8 sec)
{
    u16 t;
    u32 seccount=0;
    if(syear<1970||syear>2099)return 1;
    for(t=1970;t<syear;t++) //把所有年份的秒钟相加
    {
        if(Is_Leap_Year(t))seccount+=31622400;//闰年的秒钟数
        else seccount+=31536000;                    //平年的秒钟数
    }
    smon-=1;
    for(t=0;t<smon;t++)          //把前面月份的秒钟数相加
    {
        seccount+=(u32)mon_table[t]*86400;//月份秒钟数相加
        if(Is_Leap_Year(syear)&&t==1)seccount+=86400;//闰年 2 月份增加一天的秒钟数

    }
    seccount+=(u32)(sday-1)*86400;//把前面日期的秒钟数相加
    seccount+=(u32)hour*3600;//小时秒钟数
    seccount+=(u32)min*60;       //分钟秒钟数
    seccount+=sec;//最后的秒钟加上去

    //设置时钟
    RCC->APB1ENR|=1<<28;//使能电源时钟
    RCC->APB1ENR|=1<<27;//使能备份时钟
    PWR->CR|=1<<8;      //取消备份区写保护
    //上面三步是必须的!
    RTC->CRL|=1<<4;     //允许配置
```

```
    RTC->CNTL=seccount&0xffff;
    RTC->CNTH=seccount>>16;
    RTC->CRL&=~(1<<4);//配置更新
    while(!(RTC->CRL&(1<<5)));//等待 RTC 寄存器操作完成
    return 0;
}
//得到当前的时间
//返回值:0，成功;其他:错误代码.
u8 RTC_Get(void)
{
    static u16 daycnt=0;
    u32 timecount=0;
    u32 temp=0;
    u16 temp1=0;

    timecount=RTC->CNTH;//得到计数器中的值(秒钟数)
    timecount<<=16;
    timecount+=RTC->CNTL;

    temp=timecount/86400;    //得到天数(秒钟数对应的)
    if(daycnt!=temp)//超过一天了
    {
        daycnt=temp;
        temp1=1970;  //从 1970 年开始
        while(temp>=365)
        {
            if(Is_Leap_Year(temp1))//是闰年
            {
                if(temp>=366)temp-=366;//闰年的秒钟数
                else break;
            }
            else temp-=365;        //平年
            temp1++;
        }
        timer.w_year=temp1;//得到年份
        temp1=0;
        while(temp>=28)//超过了一个月
        {
            if(Is_Leap_Year(timer.w_year)&&temp1==1)//当年是不是闰年/2 月份
            {
                if(temp>=29)temp-=29;//闰年的秒钟数
                else break;
            }
```

```
                else
                {
                    if(temp>=mon_table[temp1])temp-=mon_table[temp1];//平年
                    else break;
                }
                temp1++;
            }
            timer.w_month=temp1+1;//得到月份
            timer.w_date=temp+1;  //得到日期
        }
        temp=timecount%86400;        //得到秒钟数
        timer.hour=temp/3600;        //小时
        timer.min=(temp%3600)/60; //分钟
        timer.sec=(temp%3600)%60; //秒钟
        timer.week=RTC_Get_Week(timer.w_year，timer.w_month，timer.w_date);//获取星期
        return 0;
}
//获得现在是星期几
//功能描述:输入公历日期得到星期(只允许 1901-2099 年)
//输入参数：公历年月日
//返回值：星期号
u8 RTC_Get_Week(u16 year，u8 month，u8 day)
{
        u16 temp2;
        u8 yearH，yearL;

        yearH=year/100;    yearL=year%100;
        // 如果为 21 世纪，年份数加 100
        if (yearH>19)yearL+=100;
        // 所过闰年数只算 1900 年之后的
        temp2=yearL+yearL/4;
        temp2=temp2%7;
        temp2=temp2+day+table_week[month-1];
        if (yearL%4==0&&month<3)temp2--;
        return(temp2%7);
}
```

该代码里面，我们重点介绍 RTC_Init 这个函数，其他的我们就不介绍了，代码都有详细的注解，是比较好理解的。函数 RTC_Init 的代码见上面，我们这里仅说明一下这个函数的实现，该函数用来初始化 RTC 时钟，但是只在第一次的时候设置时间，以后如果重新上电/复位都不会再进行时间设置了（前提是备份电池有电），在第一次配置的时候，我们是按照上面介绍的 RTC 初始化步骤来做的，这里就不在多说了。而在复位之后，该函数直接就是跳过时间设置，仅仅使能秒钟中断一下，就进行中断分组，然后返回了。这样不会重复设置时间，使得我们设置的时间不会因复位或者断电而丢失。

该函数还有返回值，返回值代表此次操作的成功与否，如果返回 0，则代表初始化 RTC 成功，如果返回值非零则代表错误代码了。

以上代码还用到了一个 tm 的结构体，tm 是我们在 rtc.h 里面将要定义的一个时间结构体，用来存放时钟的年月日时分秒等信息。因为 STM32 的 RTC 只有秒钟计数器，而年月日，时分秒这些需要我们自己软件计算。我们把计算好的值保存在 tm 里面，方便其他程序调用。

保存 rtc.c，然后将 rtc.c 加入 HARDWARE 组下，然后在 rtc.h 里面输入如下代码：

```
#ifndef __RTC_H
#define __RTC_H
//时间结构体
typedef struct
{
    u8 hour;
    u8 min;
    u8 sec;
    //公历日月年周
    u16 w_year;
    u8  w_month;
    u8  w_date;
    u8  week;
}tm;
extern tm timer;
extern u8 const mon_table[12];//月份日期数据表
void Disp_Time(u8 x，u8 y，u8 size);//在制定位置开始显示时间
void Disp_Week(u8 x，u8 y，u8 size，u8 lang);//在指定位置显示星期
u8 RTC_Init(void);          //初始化 RTC，返回 0，失败;1，成功;
u8 Is_Leap_Year(u16 year);//平年，闰年判断
u8 RTC_Get(void);           //更新时间
u8 RTC_Get_Week(u16 year，u8 month，u8 day);
u8 RTC_Set(u16 syear，u8 smon，u8 sday，u8 hour，u8 min，u8 sec);//设置时间
#endif
```

从上面代码可以看到 tm 结构体所包含的东西，是一个完整的公历信息，包括年、月、日、周、时、分、秒等 7 个元素。我们以后要知道当前时间，只需要通过 RTC_Get 函数，执行时钟转换，然后就可以从 tm 里面读出当前的公历时间了。

在 test.c 里面，我们修改 main 函数如下：

```
const 8* Week[7]={"Sunday"，"Monday"，"Tuesday"，"Wednesday"，"Thursday"，"Friday"，
"Saturday"};
int main(void)
{
    u8 t=0;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72，9600); //串口 1 初始化
    LED_Init();
```

```
LCD_Init();
RTC_Init();
//RTC_Set(2004,12,31,23,59,55);  //设置时间
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60，50，"Mini STM32");
LCD_ShowString(60，70，"RTC TEST");
LCD_ShowString(60，90，"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2010/12/31");
//显示时间
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(60，130，"      -  -  ");
LCD_ShowString(60，162，"  :  :  ");
while(1)
{
    if(t!=timer.sec)
    {
        t=timer.sec;
        LCD_ShowNum(60，130，timer.w_year，4，16);

        LCD_ShowNum(100，130，timer.w_month，2，16);

        LCD_ShowNum(124，130，timer.w_date，2，16);
        switch(timer.week)
        {
            case 0:
                LCD_ShowString(60，148，"Sunday    ");
                break;
            case 1:
                LCD_ShowString(60，148，"Monday    ");
                break;
            case 2:
                LCD_ShowString(60，148，"Tuesday   ");
                break;
            case 3:
                LCD_ShowString(60，148，"Wednesday");
                break;
            case 4:
                LCD_ShowString(60，148，"Thursday ");
                break;
            case 5:
                LCD_ShowString(60，148，"Friday    ");
                break;
            case 6:
```

```
                    LCD_ShowString(60，148，"Saturday ");
                    break;
                }
                LCD_ShowNum(60，162，timer.hour，2，16);

                LCD_ShowNum(84，162，timer.min，2，16);

                LCD_ShowNum(108，162，timer.sec，2，16);
                LED0=!LED0;
            }
            delay_ms(10);
        };
    }
```

这部分代码就不再需要详细解释了，在包含了 rtc.h 之后，通过判断 tm.sec 是否改变来决定要不要更新时间显示。同时我们设置 LED0 每 2 秒钟闪烁一次，用来提示程序已经开始跑了。上面代码中，我们屏蔽了如下这句：

```
//RTC_Set(2004,12,31,23,59,55);  //设置时间
```

该句用于设置时间，因为我们在第一次设置之后就不需要再设置了，所以我们把该句屏蔽掉了，如果大家需要设置新的时间，只要修改相应的参数，并使用该函数就好了。然后编译整个工程，开始下载程序观看实际结果。

至此，RTC 实时时钟的软件设计就完成了，接下来开始下载与测试。

3.11.4 下载与测试

将代码下载到 MiniSTM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块开始显示时间，实际显示效果如下图所示：

图 3.11.4.1 RTC 实验效果图

大家可以试试按复位按钮，或者把电断掉几秒钟，再次接上电源，看看时间是否还在继续走。

3.12 待机唤醒实验

这一节我们将向大家介绍 ALIENTEK MiniSTM32 开发板的待机唤醒功能。本节将利用 WK_UP 按键来实现唤醒和进入待机模式功能，然后利用 DS0 指示状态。本节分为如下几个部分：

3.12.1 STM32 待机模式简介
3.12.2 硬件设计
3.12.3 软件设计
3.12.4 下载与测试

3.12.1 STM32 待机模式简介

很多单片机都有低功耗模式，STM32 也不例外。在系统或电源复位以后，微控制器处于运行状态。运行状态下的 HCLK 为 CPU 提供时钟，内核执行程序代码。当 CPU 不需继续运行时，可以利用多个低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗，最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。STM32 的 3 中低功耗模式我们在第二章也粗略介绍了一下，这里我们再回顾一下。

STM32 的低功耗模式有 3 种：

1)睡眠模式（CM3 内核停止，外设仍然运行）

2)停止模式（所有时钟都停止）

3)待机模式（1.8V 内核电源关闭）

在运行模式下，我们也可以通过降低系统时钟关闭 APB 和 AHB 总线上未被使用的外设的时钟来降低功耗。三种低功耗模式一览表：

模式	进入操作	唤醒	对1.8V区域时钟的影响	对VDD区域时钟的影响	电压调节器
睡眠(SLEEP-NOW或SLEEP-ON-EXIT)	WFI	任一中断	CPU时钟关，对其他时钟和ADC时钟无影响	无	开
	WFE	唤醒事件			
停机	PDDS和LPDS位+SLEEPDEEP位+WFI或WFE	任一外部中断(在外部中断寄存器中设置)	所有使用1.8V的区域的时钟都已关闭，HSI和HSE的振荡器关闭		在低功耗模式下可进行开/关设置(依据电源控制寄存器(PWR_CR)的设定)
待机	PDDS位+SLEEPDEEP位+WFI或WFE	WKUP引脚的上升沿、RTC警告事件、NRST引脚上的外部复位、IWDG复位			关

表 3.12.1.1 STM32 低功耗一览表

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 20uA 左右。最后就是睡眠模式了。用户可以根据自己的需求来决定使用哪种低功耗模式。

这一节，我们就针对 STM32 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32 的最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。只有备份的寄存器和待机电路维持供电。

那么我们如何进入待机模式呢？其实很简单，只要按下表的步骤执行就可以了：

待机模式	说明
进入	在以下条件下执行WFI或WFE指令： - 设置Cortex™-M3系统控制寄存器中的SLEEPDEEP位 - 设置电源控制寄存器(PWR_CR)中的PDDS位 - 清除电源控制/状态寄存器(PWR_CSR)中的WUF位被
退出	WKUP引脚的上升沿、RTC闹钟、NRST引脚上外部复位、IWDG复位。
唤醒延时	复位阶段时电压调节器的启动。

图 3.12.1.1 STM32 进入及退出待机模式条件

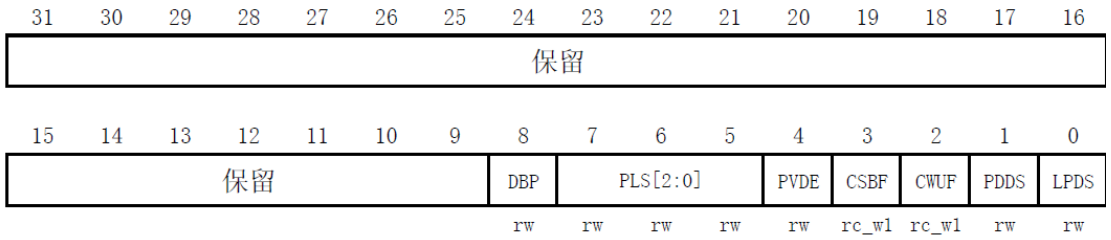
上表还列出了退出待机模式的操作，从上表可知，我们有 4 种方式可以退出待机模式，即当一个外部复位(NRST 引脚)、IWDG 复位、WKUP 引脚上的上升沿或 RTC 闹钟事件发生时，微控制器从待机模式退出。从待机唤醒后，除了电源控制/状态寄存器(PWR_CSR)，所有寄存器被复位。

从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚，读取复位向量等)。电源控制/状态寄存器(PWR_CSR)将会指示内核由待机状态退出。

在进入待机模式后，除了复位引脚以及被设置为防侵入或校准输出时的 TAMPER 引脚和被是能的唤醒引脚（WK_UP 脚），其他的 IO 引脚都将处于高阻态。

上表已经清楚的说明了进入待机模式的通用步骤，其中涉及到 2 个寄存器，也就是电源控制寄存器（PWR_CR）和电源控制/状态寄存器（PWR_CSR）。下面我们介绍一下这两个寄存器：

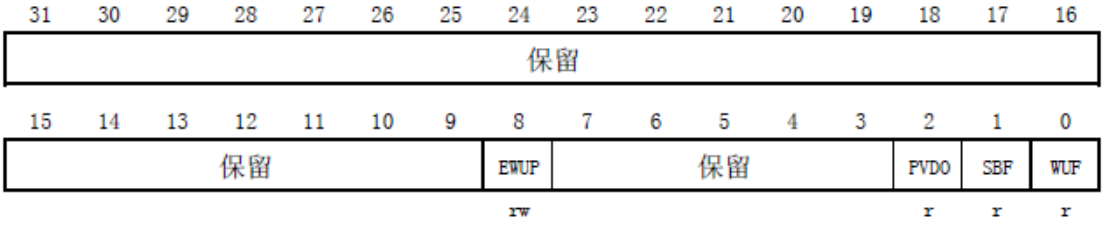
电源控制寄存器（PWR_CR），该寄存器的各位描述如下：



位 31:9	保留。始终读为0。
位 8	**DBP**：取消后备区域的写保护 在复位后，RTC和后备寄存器处于被保护状态以防意外写入。设置这位允许写入这些寄存器。 0：禁止写入RTC和后备寄存器 1：允许写入RTC和后备寄存器
位 7:5	**PLS[2:0]**：PVD电平选择 这些位用于选择电源电压监测器的电压阀值 000：2.2V　100：2.6V 001：2.3V　101：2.7V 010：2.4V　110：2.8V 011：2.5V　111：2.9V 注：详细说明参见数据手册中的电气特性部分。
位 4	**PVDE**：电源电压监测器(PVD)使能 0：禁止PVD 1：开启PVD
位 3	**CSBF**：清除待机位 始终读出为0 0：无功效 1：清除SBF待机位(写)
位 2	**CWUF**：清除唤醒位 始终读出为0 0：无功效 1：2个系统时钟周期后清除WUF唤醒位(写)
位 1	**PDDS**：掉电深睡眠 与LPDS位协同操作 0：当CPU进入深睡眠时进入停机模式，调压器的状态由LPDS位控制。 1：CPU进入深睡眠时进入待机模式。
位 0	**LPDS**：深睡眠下的低功耗 PDDS=0时，与PDDS位协同操作 0：在停机模式下电压调压器开启 1：在停机模式下电压调压器处于低功耗模式

图 3.12.1.2 寄存器 PWR_CR 各位描述

这里我们通过设置 PWR_CR 的 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时我们通过 CWUF 位，清除之前的唤醒位。电源控制/状态寄存器（PWR_CSR）的各位描述如下：



位31:9	保留。始终读为0。
位 8	**EWUP**：使能WKUP管脚 0：WKUP管脚为通用I/O。WKUP管脚上的事件不能将CPU从待机模式唤醒 1：WKUP管脚用于将CPU从待机模式唤醒，WKUP管脚被强置为输入下拉的配置(WKUP管脚上的上升沿将系统从待机模式唤醒) 注：在系统复位时清除这一位。
位 7:3	保留。始终读为0。
位 2	**PVDO**：PVD输出 当PVD被PVDE位使能后该位才有效 0：V_{DD}/V_{DDA}高于由PLS[2:0]选定的PVD阀值 1：V_{DD}/V_{DDA}低于由PLS[2:0]选定的PVD阀值 注：在待机模式下PVD被停止。因此，待机模式后或复位后，直到设置PVDE位之前，该位为0。
位 1	**SBF**：待机标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CSBF位清除。 0：系统不在待机模式 1：系统进入待机模式
位 0	**WUF**：唤醒标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CWUF位清除。 0：没有发生唤醒事件 1：在WKUP管脚上发生唤醒事件或出现RTC闹钟事件。 注：当WKUP管脚已经是高电平时，在(通过设置EWUP位)使能WKUP管脚时，会检测到一个额外的事件。

图 3.12.1.3 寄存器 PWR_ CSR 各位描述

这里，我们通过设置 PWR_CSR 的 EWUP 位，来使能 WKUP 引脚用于待机模式唤醒。我们还可以从 WUF 来检查是否发生了唤醒事件。不过这一节我们没有用到。

通过以上介绍，我们了解了进入待机模式的方法，以及设置 WK_UP 引脚用于把 STM32 从待机模式唤醒的方法。具体步骤如下：

1）设置 SLEEPDEEP 位。

该位在系统控制寄存器（SCB_SCR）的第二位（详见《CM3 权威指南》，第 182 页表 13.1），我们通过设置该位，作为进入待机模式的第一步。

2）使能电源时钟，设置 WK_UP 引脚作为唤醒源。

因为要配置电源控制寄存器，所以必须先使能电源时钟。然后再设置 PWR_CSR 的 EWUP 位，使能 WK_UP 用于将 CPU 从待机模式唤醒。

3）设置 PDDS 位，执行 WFI 指令，进入待机模式。

接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。

4）最后编写 WK_UP 中断函数。

因为我们通过 WK_UP 中断（PA0 中断）来唤醒 CPU，所以我们有必要设置一下该中断函数，同时我们也通过该函数里面进入待机模式。

通过以上几个步骤的设置，我们就可以使用 STM32 的待机模式了，并且可以通过 WK_UP 来唤醒 CPU，我们最终要实现这样一个功能：通过长按（3 秒）WK_UP 按键开机，并且通过 DS0 的闪烁指示程序已经开始运行，再次长按该键，则进入待机模式，DS0 关闭，程序停止运行。类似于手机的开关机。

3.12.2 硬件设计

这一节，我们使用了 WK_UP 按键用于唤醒和进入待机模式。然后通过 DS0 来指示程序是否在运行。因为 DS0 和 WK_UP 在 MiniSTM32 开发板上都是直接连在 STM32 的 IO 口上的，不需要任何修改，这里我们就不在贴图了。大家可以参考 3.1 节相关内容。

3.12.3 软件设计

找到上一节的工程，首先在 HARDWARE 文件夹下新建一个 WKUP 的文件夹。然后打开 USER 文件夹下的工程，新建一个 wkup.c 的文件和 wkup.h 的头文件，保存在 WKUP 文件夹下，并将 WKUP 文件夹加入头文件包含路径。

打开 wkup.c，输入如下代码：

```
#include "wkup.h"
#include "led.h"
#include "delay.h"
//Mini STM32 开发板
//待机唤醒  驱动代码
//正点原子@ALIENTEK
//2010/6/7
//系统进入待机模式
void Sys_Enter_Standby(void)
{
    //关闭所有外设(根据实际情况写)
    RCC->APB2RSTR|=0X01FC;//复位所有 IO 口
    Sys_Standby();//进入待机模式
}
//检测 WKUP 脚的信号
//返回值 1:连续按下 3s 以上
//       0:错误的触发
u8 Check_WKUP(void)
{
    u8 t=0;
    u8 tx=0;//记录松开的次数
    LED0=0; //亮灯 DS0
    while(1)
```

```
        {
            if(WKUP_KD)//已经按下了
            {
                t++;
                tx=0;
            }else
            {
                tx++; //超过 300ms 内没有 WKUP 信号
                if(tx>3)
                {
                    LED0=1;
                    return 0;//错误的按键，按下次数不够
                }
            }
            delay_ms(30);
            if(t>=100)//按下超过 3 秒钟
            {
                LED0=0;    //点亮 DS0
                return 1; //按下 3s 以上了
            }
        }
}
//中断，检测到 PA0 脚的一个上升沿.
//中断线 0 线上的中断检测
void EXTI0_IRQHandler(void)
{
    EXTI->PR=1<<0;   //清除 LINE10 上的中断标志位
    if(Check_WKUP())//关机?
    {
        Sys_Enter_Standby();
    }
}
//PA0 WKUP 唤醒初始化
void WKUP_Init(void)
{
    RCC->APB2ENR|=1<<2;        //先使能外设 IO PORTA 时钟
    RCC->APB2ENR|=1<<0;        //开启辅助时钟

    GPIOA->CRL&=0XFFFFFFF0;//PA0 设置成输入
    GPIOA->CRL|=0X00000008;
    Ex_NVIC_Config(GPIO_A，0，RTIR);//PA0 上升沿触发

    //(检查是否是正常开)机
```

```
    if(Check_WKUP()==0)Sys_Standby();      //不是开机，进入待机模式
    MY_NVIC_Init(2，2，EXTI0_IRQChannel，2);//抢占 2，子优先级 2，组 2
}
```

这里我们要删除 exti.c，因为该函数里面也有 void EXTI0_IRQHandler(void)函数，如果不删除，MDK 就会报错。该部分代码比较简单，我们在这里说明 2 点：1，在 void Sys_Enter_Standby(void)函数里面，我们要在进入待机模式前把所有开启的外设全部关闭，我们这里仅仅复位了所有的 IO 口，使得 IO 口全部为浮空输入。其他外设（比如 ADC 等），大家根据自己所开启的情况进行一一关闭就可，这样才能达到最低功耗！2，在 void WKUP_Init(void)函数里面，我们要先判断 WK_UP 是否按下了 3 秒钟，来决定要不要开机，如果没有按下 3 秒钟，程序直接就进入了待机模式。所以在下载完代码的时候，是看不到任何反应的。我们必须先按 WK_UP 按键 3 秒钟以开机，才能看到 DS0 闪烁。

保存 wkup.c，并加入到 HARDWARE 组下，然后我们在 wkup.h 里面加入如下代码：

```
#ifndef __WKUP_H
#define __WKUP_H
#include "sys.h"
//Mini STM32 开发板
//待机唤醒  驱动代码
//正点原子@ALIENTEK
//2010/6/7
#define WKUP_KD PAin(0)   //PA0  检测是否外部 WK_UP 按键按下

u8 Check_WKUP(void);   //检测 WKUP 脚的信号
void WKUP_Init(void); //PA0 WKUP 唤醒初始化
void Sys_Enter_Standby(void);//系统进入待机模式
#endif
```

该部分代码，也很简单，我们就不多说了。最后我们在 test.c 里面修改 main 函数如下：

```
int main(void)
{
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);      //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();
    WKUP_Init();
    while(1)
    {
        LED0=!LED0;
        delay_ms(250);
    }
}
```

这里我们先初始化 LED 和 WK_UP 按键（通过 WKUP_Init（）函数初始化），在死循环里面等待 WK_UP 中断的到来，在得到中断后，判断 WK_UP 按下的时间长短，来决定是否进入待机模式。在 WKUP_Init 函数里面，我们有检测 WK_UP 是否按下 3 秒来决定是否开机，这点在前面已经介绍了。大家在下载完代码的时候要注意一下。

3.12.4 下载与测试

在代码编译成功之后，我们通过 USB 串口线下载代码到 ALIENTEK MiniSTM32 开发板上，这里不能像前面的代码一样，下载完就可以测试了，这里我们必须先把 B0 通过跳线帽连接到 GND，然后再按复位键，才能开始测试。这是因为：虽然我们在 mcuisp 软件里面设置了编程后执行，但是代码运行的时候，我们并没有长按 WK_UP 按键 3 秒，所以程序就进入了待机模式。然后由于 BOOT0 的设置是串口下载模式，我们不能通过按复位键来重新启动程序。所以必须把 BOOT0 接到 GND，使得复位键可以唤醒 CPU。然后才能开始测试。

将 B0 接到 GND 后，我们按复位键，然后再长按 WK_UP 按键 3 秒钟左右，可以看到 DS0 开始闪烁。然后再长按 WK_UP，DS0 会灭掉，程序再次进入待机模式。

3.13 ADC实验

这一节我们将向大家介绍 STM32 的 ADC。本节将利用 STM32 的 ADC1 通道 0 来采样外部电压值值，并在 TFTLCD 模块上显示出来。本节分为如下几个部分：

3.13.1 STM32 ADC 简介

3.13.2 硬件设计

3.13.3 软件设计

3.13.4 下载与测试

3.13.1 STM32 ADC 简介

STM32 拥有 1~3 个 ADC，这些 ADC 可以独立使用，也可以使用双重模式（提高采样率）。STM32 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它有 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。

STM32F103 系列最少都拥有 2 个 ADC，我们选择的 STM32F103RBT6 也包含有 2 个 ADC。STM32 的 ADC 最大的转换速率为 1Mhz，也就是转换时间为 1us（在 ADCCLK=14M,采样周期为 1.5 个 ADC 时钟下得到），不要让 ADC 的时钟超过 14M，否则将导致结果准确度下降。

STM32 将 ADC 的转换分为 2 个通道组：规则通道组和注入通道组。规则通道相当于你运行的程序，而注入通道呢，就相当于中断。在你程序正常执行的时候，中断是可以打断你的执行的。同这个类似，注入通道的转换可以打断规则通道的转换， 在注入通道被转换完成之后，规则通道才得以继续转换。

通过一个形象的例子可以说明：假如你在家里的院子内放了 5 个温度探头，室内放了 3 个温度探头；你需要时刻监视室外温度即可，但偶尔你想看看室内的温度；因此你可以使用规则通道组循环扫描室外的 5 个探头并显示 AD 转换结果，当你想看室内温度时，通过一个按钮启动注入转换组(3 个室内探头)并暂时显示室内温度，当你放开这个按钮后，系统又会回到规则通道组继续检测室外温度。从系统设计上，测量并显示室内温度的过程中断了测量并显示室外温度的过程，但程序设计上可以在初始化阶段分别设置好不同的转换组，系统运行中不必再变更循环转换的配置，从而达到两个任务互不干扰和快速切换的结果。可以设想一下，如果没有规则组和注入组的划分，当你按下按钮后，需要从新配置 AD 循环扫描的通道，然后在施放按钮后需再此配置 AD 循环扫描的通道。

上面的例子因为速度较慢，不能完全体现这样区分(规则通道组和注入通道组)的好处，但在工业应用领域中有很多检测和监视探头需要较快地处理，这样对 AD 转换的分组将简化事件处理的程序并提高事件处理的速度。

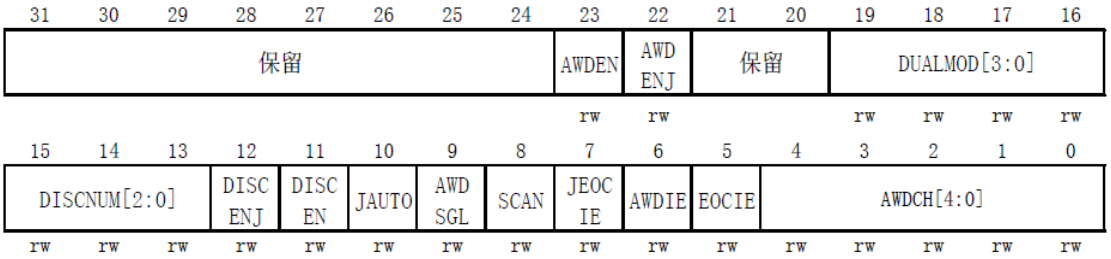
STM32ADC 的规则通道组最多包含 16 个转换，而注入通道组最多包含 4 个通道。关于这两个通道组的详细介绍，请参考《STM32 参考手册的》第 113 页，第 10 章。

STM32 的 ADC 可以进行很多种不同的转换模式，这些模式在《STM32 参考手册》的第 10 章也都有详细介绍，我们这里就不在一一列举了。我们本节仅介绍如何使用规则通道的单次转换模式。

STM32 的 ADC 在单次转换模式下，只执行一次转换，该模式可以通过 ADC_CR2 寄存器的 ADON 位（只适用于规则通道）启动，也可以通过外部触发启动（适用于规则通道和注入通道），这是 CONT 位为 0。

以规则通道为例，一旦所选择的通道转换完成，转换结果将被存在 ADC_DR 寄存器中，EOC（转换结束）标志将被置位，如果设置了 EOCIE，则会产生中断。然后 ADC 将停止，直到下次启动。

接下来，我们介绍一下我们执行规则通道的单次转换，需要用到的 ADC 寄存器。第一个要介绍的是 ADC 控制寄存器（ADC_CR1 和 ADC_CR2）。ADC_CR1 的各位描述如下：



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

图 3.13.1.1 寄存器 ADC_CR1 各位描述

这里我们不再详细介绍每个位，而是抽出几个我们这一节要用到的位进行针对性的介绍，详细的说明及介绍，请参考《STM32 参考手册》第 11 章的相关章节。

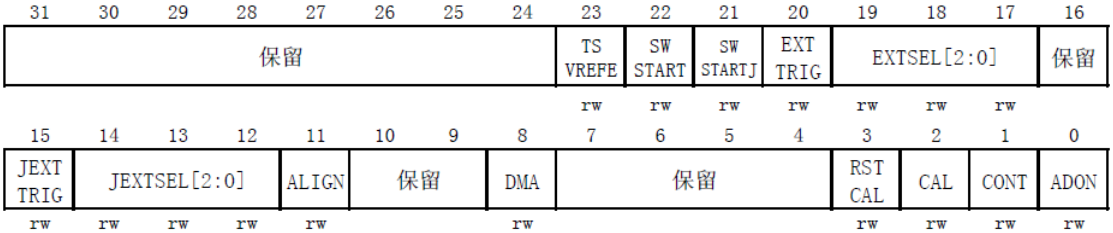
ADC_CR1 的 SCAN 位，该位用于设置扫描模式，由软件设置和清除，如果设置为 1，则使用扫描模式，如果为 0，则关闭扫描模式。在扫描模式下，由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道被转换。如果设置了 EOCIE 或 JEOCIE，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。

ADC_CR1[19:16]用于设置 ADC 的操作模式，详细的对应关系如下：

位19:16	**DUALMOD[3:0]**：双模式选择 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+快速交替模式 0100：混合同步注入+慢速交替模式 0101：注入同步模式 0110：规则同步模式 0111：快速交替模式 1000：慢速交替模式 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。

图 3.13.1.2 ADC 操作模式

本节我们要使用的是独立模式，所以设置这几位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如下：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								TSVREFE	SWSTART	SWSTARTJ	EXTTRIG	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
JEXTTRIG	JEXTSEL[2:0]			ALIGN	保留		DMA	保留				RSTCAL	CAL	CONT	ADON
rw	rw	rw	rw	rw			rw					rw	rw	rw	rw

图 3.13.1.3 寄存器 ADC_CR2 操作模式

该寄存器我们也只针对性的介绍一些位：ADCON 位用于开关 AD 转换器。而 CONT 位用于设置是否进行连续转换，我们使用单次转换，所以 CONT 位必须为 0。CAL 和 RSTCAL 用于 AD 校准。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTSEL[2:0]用于选择启动规则转换组转换的外部事件，详细的设置关系如下：

位19:17	**EXTSEL[2:0]**：选择启动规则通道组转换的外部事件 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 010：定时器1的CC3事件　　110：EXTI线11/ TIM8_TRGO，仅大容量产品具有TIM8_TRGO功能 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

图 3.13.1.4 ADC 全气动规则转换事件设置

我们这里使用的是软件触发（SWSTART），所以设置这 3 个位为 111。ADC_CR2 的 SWSTART 位用于开始规则通道的转换，我们每次转换（单次转换模式下）都需要向该位写 1。AWDEN 为用于使能温度传感器和 Vrefint。STM32 内部的温度传感器我们将在下一节介绍。

第二个要介绍的是 ADC 采样事件寄存器（ADC_SMPR1 和 ADC_SMPR2），这两个寄存器用于设置通道 0~17 的采样时间，每个通道占用 3 个位。ADC_SMPR1 的各位描述如下：

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
SMP 15_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 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14，15，16，17与Vss相连

图 3.13.1.5　寄存器 ADC_SMPR1 各位描述

ADC_SMPR2 的各位描述如下图所示：

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 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

图 3.13.1.6　寄存器 ADC_SMPR2 各位描述

对于每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率。ADC 的转换时间可以由下式计算：

Tcovn=采样时间+12.5 个周期

其中：Tcovn 为总转换时间，采样时间是根据每个通道的 SMP 位的设置来决定的。例如，当 ADCCLK=14Mhz 的时候，并设置 1.5 个周期的采样时间，则得到:Tcovn=1.5+12.5=14 个周期=1us。

第三个要介绍的是 ADC 规则序列寄存器（ADC_SQR1~3）,该寄存器总共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC_SQR1，该寄存器的各位描述如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[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23:20	**L[3:0]**：规则通道序列长度 这些位定义了在规则通道转换序列中转换总数。 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 这些位定义了转换序列中的第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
位9:5	**SQ14[4:0]**：规则序列中的第14个转换
位4:0	**SQ13[4:0]**：规则序列中的第13个转换

图 3.13.1.7　寄存器 ADC_ SQR1 各位描述

L[3：0]用于存储规则序列的长度，我们这里只用了 1 个，所以设置这几个位的值为 0。其他的 SQ13~16 则存储了规则序列中第 13~16 个通道的编号（0~17）。另外两个规则序列寄存器同 ADC_SQR1 大同小异，我们这里就不再介绍了，要说明一点的是：我们选择的是单次转换，

所以只有一个通道在规则序列里面，这个序列就是 SQ0，通过 ADC_SQR3 的最低 5 位设置。

第四个要介绍的是 ADC 规则数据寄存器(ADC_DR)。规则序列中的 AD 转化结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。ADC_DR 的各位描述如下：

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 - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见10.9：双ADC模式 - 在ADC2中：不用这些位。
位15:0	**DATA[15:0]**：规则转换的数据 这些位为只读，包含了规则通道的转换结果。数据是左或右对齐，如图25和图26所示。

图 3.13.1.8　寄存器 ADC_ JDRx 各位描述

这里要提醒一点的是，该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

最后一个要介绍的 ADC 寄存器为 ADC 状态寄存器（ADC_SR），该寄存器保存了 ADC 转换时的各种状态。该寄存器的各位描述如下：

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											STRT	JSTRT	JEOC	EOC	AWD
											rw	rw	rw	rw	rw

位31:15	保留。必须保持为0。
位4	**STRT**：规则通道开始位 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 该位由硬件在注入通道组转换开始时设置，由软件清除。 0：注入通道转换未开始 1：注入通道转换已开始
位2	**JEOC**：注入通道转换结束位 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 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 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

图 3.13.1.9　寄存器 ADC_ SR 各位描述

这里我们要用到的是 EOC 位，我们通过判断该位来决定是否此次规则通道的 AD 转换已经完成，如果完成我们就从 ADC_DR 中读取转换结果，否则等待转换完成。

通过以上介绍，我们了解了 STM32 的单次转换模式下的相关设置，这一节我们使用 ADC1 的通道 0 来进行 AD 转换，其详细设置步骤如下：

1）开启 PA 口时钟，设置 PA0 为模拟输入。

STM32F103RBT6 的 ADC 通道 0 在 PA0 上，所以，我们先要使能 PORTA 的时钟，然后设置 PA0 为模拟输入。

2）使能 ADC1 时钟，并设置分频因子。

要使用 ADC1，第一步就是要使能 ADC1 的时钟，在使能完时钟之后，进行一次 ADC1 的复位。接着我们就可以通过 RCC_CFGR 设置 ADC1 的分频因子。分频因子要确保 ADC1 的时钟（ADCCLK）不要超过 14Mhz。

3）设置 ADC1 的工作模式。

在设置完分频因子之后，我们就可以开始 ADC1 的模式配置了，设置单次转换模式、触发方式选择、数据对齐方式等都在这一步实现。

4）设置 ADC1 规则序列的相关信息。

接下来我们要设置规则序列的相关信息，我们这里只有一个通道，并且是单次转换的，所以设置规则序列中通道数为 1，然后设置通道 0 的采样周期。

5）开启 AD 转换器，并校准。

在设置完了以上信息后，我们就开启 AD 转换器，执行复位校准和 AD 校准，注意这两步是必须的！不校准将导致结果很不准确。

6）读取 ADC 值。

在上面的校准完成之后，ADC 就算准备好了。接下来我们要做的就是设置规则序列 0 里面的通道，然后启动 ADC 转换。在转换结束后，读取 ADC1_DR 里面的值就是了。

通过以上几个步骤的设置，我们就可以正常的使用 STM32 的 ADC1 来执行 AD 转换操作了。

3.13.2 硬件设计

这一节，我们通过 ADC1 的通道 0（PA0）来读取外部电压值，MiniSTM32 开发板没有设计电压源在上面，但是板上有几个可以提供测试的地方：1，3.3V 电源。2，GND。3，后备电池。注意：这里不能接到板上 5V 电源上去测试，这可能会烧坏 ADC!。

因为要连接到其他地方测试电压，所以我们需要 1 跟杜邦线，或者自备的连接线也可以，一头插在 PA0 上，另外一头就接你要测试的电压点（确保该电压不大于 3.3V 即可）。

这里需要用到 TFTLCD 模块，所以，要把 LCD 模块先接到板子上。除此之外，硬件上其他地方没什么要改的。

以测试 3.3V 电源电压为例，实物连接如下图所示：

图 3.13.2.1 ADC 实验实物连接图

3.13.3 软件设计

找到上一节的工程，首先在HARDWARE文件夹下新建一个ADC的文件夹。然后打开USER文件夹下的工程，新建一个 adc.c 的文件和 adc.h 的头文件，保存在 ADC 文件夹下，并将 ADC 文件夹加入头文件包含路径。

打开 adc.c，输入如下代码：

```
#include <stm32f10x_lib.h>
#include "adc.h"
//Mini STM32 开发板
//ADC  驱动代码
//正点原子@ALIENTEK
//2010/6/7
//初始化 ADC
//这里我们仅以规则通道为例
//我们默认将开启通道 0~3
void    Adc_Init(void)
{
    //先初始化 IO 口
    RCC->APB2ENR|=1<<2;       //使能 PORTA 口时钟
    GPIOA->CRL&=0XFFFF0000;//PA0 1 2 3 anolog 输入
```

```
    //通道 10/11 设置
    RCC->APB2ENR|=1<<9;       //ADC1 时钟使能
    RCC->APB2RSTR|=1<<9;      //ADC1 复位
    RCC->APB2RSTR&=~(1<<9);//复位结束
    RCC->CFGR&=~(3<<14);      //分频因子清零
    //SYSCLK/DIV2=12M ADC 时钟设置为 12M,ADC 最大时钟不能超过 14M!
    //否则将导致 ADC 准确度下降!
    RCC->CFGR|=2<<14;

    ADC1->CR1&=0XF0FFFF;      //工作模式清零
    ADC1->CR1|=0<<16;         //独立工作模式
    ADC1->CR1&=~(1<<8);       //非扫描模式
    ADC1->CR2&=~(1<<1);       //单次转换模式
    ADC1->CR2&=~(7<<17);
    ADC1->CR2|=7<<17;         //软件控制转换
    ADC1->CR2|=1<<20;         //使用用外部触发(SWSTART)!!! 必须使用一个事件来触发
    ADC1->CR2&=~(1<<11);      //右对齐

    ADC1->SQR1&=~(0XF<<20);
    ADC1->SQR1&=0<<20;        //1 个转换在规则序列中  也就是只转换规则序列 1

    //设置通道 0~3 的采样时间
    ADC1->SMPR2&=0XFFFFF000;//通道 0,1,2,3 采样时间清空
    ADC1->SMPR2|=7<<9;        //通道 3   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<6;        //通道 2   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<3;        //通道 1   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<0;        //通道 0   239.5 周期,提高采样时间可以提高精确度

    ADC1->CR2|=1<<0;          //开启 AD 转换器
    ADC1->CR2|=1<<3;          //使能复位校准
    while(ADC1->CR2&1<<3);    //等待校准结束
    //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。
    ADC1->CR2|=1<<2;          //开启 AD 校准
    while(ADC1->CR2&1<<2);    //等待校准结束
    //该位由软件设置以开始校准，并在校准结束时由硬件清除
}
//获得 ADC 值
//ch:通道值  0~3
u16 Get_Adc(u8 ch)
{
    //设置转换序列
    ADC1->SQR3&=0XFFFFFFE0;//规则序列 1  通道 ch
    ADC1->SQR3|=ch;
```

```
    ADC1->CR2|=1<<22;           //启动规则转换通道
    while(!(ADC1->SR&1<<1));//等待转换结束
    return ADC1->DR;        //返回 adc 值
}
```

此部分代码就 2 个函数，Adc_Init 函数用于初始化 ADC1。这里基本上是按我们上面的步骤来初始化的，只是这里我们开通了 4 个通道，通道 0~3 都开通了。第二个函数 Get_Adc，用于读取某个通道的 ADC 值，例如我们读取通道 0 上的 ADC 值，就可以通过 Get_Adc（0）得到。

保寸 adc.c 代码，并将该代码加入 HARDWARE 组下。接下来在 adc.h 文件里面输入如下代码：

```
#ifndef __ADC_H
#define __ADC_H
//Mini STM32 开发板
//ADC  驱动代码
//正点原子@ALIENTEK
#define ADC_CH0    0 //通道 0
#define ADC_CH1    1 //通道 1
#define ADC_CH2    2 //通道 2
#define ADC_CH3    3 //通道 3

void Adc_Init(void);
u16    Get_Adc(u8 ch);
#endif
```

该部分代码很简单，这里我们就不多说了，这里定义的 4 个通道的宏定义，我们在 main 函数将会用到 ADC_CH0。

接下来我们在 test.c 里面，修改 main 函数如下：

```
int main(void)
{
    u16 adcx;
    float temp;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();
    LCD_Init();
    Adc_Init();

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"ADC TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2010/12/30");
    //显示时间
```

```
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(60,130,"ADC_CH0_VAL:");
    LCD_ShowString(60,150,"ADC_CH0_VOL:0.000V");
    while(1)
    {
        adcx=Get_Adc(ADC_CH0);
        LCD_ShowNum(156,130,adcx,4,16);//显示 ADC 的值
        temp=(float)adcx*(3.3/4096);
        adcx=temp;
        LCD_ShowNum(156,150,adcx,1,16);//显示电压值
        temp-=adcx;
        temp*=1000;
        LCD_ShowNum(172,150,temp,3,16);
        LED0=!LED0;
        delay_ms(250);
    }
}
```

此部分代码，我们在 TFTLCD 模块上显示一些信息后，将每隔 250ms 读取一次 ADC 通道 0 的值，并显示读到的 ADC 值（数字量），以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。

3.13.3 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

图 3.13.3.1 ADC 实验实际测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。大家可以试试把杜邦线接到其他地方，看看电压值是否准确？但是一定别接到 5V 上面去，否则可能烧坏 ADC！

通过这一节的学习，我们了解了 STM32 ADC 的使用，但这仅仅是 STM32 强大的 ADC 功能的一小点应用。STM32 的 ADC 在很多地方都可以用到，其 ADC 的 DMA 功能是很不错的，建议有兴趣的大家深入研究下 STM32 的 ADC，相信会给你以后的开发带来方便。

3.14 内部温度传感器实验

这一节我们将向大家介绍 STM32 的内部温度传感器。本节将利用 STM32 的内部温度传感器来显示温度值，并在 TFTLCD 模块上显示出来。本节分为如下几个部分：

3.14.1 STM32 内部温度传感器简介

3.14.2 硬件设计

3.14.3 软件设计

3.14.4 下载与测试

3.14.1 STM32 内部温度传感器简介

STM32 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度(TA)。该温度传感器在内部和 ADCx_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值。温度传感器模拟输入推荐采样时间是 17.1μs。 STM32 的内部温度传感器支持的温度范围为：-40~125 度。精度比较差，为±1.5℃左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部通道就差不多了。关于 ADC 的设置，我们在上一节已经进行了详细的介绍，这里就不再多说。接下来我们介绍一下和温度传感器设置相关的 2 个地方。

第一个地方，我们要使用 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CR2 的 AWDEN 位（bit23）设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32 的内部温度传感器固定的连接在 ADC 的通道 16 上，所以，我们在设置好 ADC 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。计算公式如下：

T（℃）={（V25-Vsense）/Avg_Slope}+25

该式中：

V25=Vsense 在 25 度时的数值（典型值为：1.43）。

Avg_Slope=温度与 Vsense 曲线的平均斜率（单位为 mv/℃或 uv/℃）（典型值为 4.3Mv/℃）。

利用以上公式，我们就可以方便的计算出当前温度传感器的温度了。

现在，我们就可以总结一下 STM32 内部温度传感器使用的步骤了，如下：

设置 ADC，并开启 ADC_CR2 的 AWDEN 位。

关于如何设置 ADC，上一节已经介绍了，我们采用与上一节一样的设置，这里我们只要增加使能 AWDEN 位这一句就可以了。

读取通道 16 的 AD 值，计算结果。

在设置完之后，我们就可以读取温度传感器的电压值了，得到该值就可以用上面的公式计算温度值了。

3.14.2 硬件设计

这部分硬件只需要 1 个 TFTLCD 模块，其他的就不需要了。TFTLCD 的连接在前面已经介绍过了，这里就不再介绍了。

3.14.3 软件设计

打开上一节的工程，打开 adc.c，修改 Adc_Init 函数代码如下：

```
void  Adc_Init(void)
{
    //先初始化 IO 口
    RCC->APB2ENR|=1<<2;       //使能 PORTA 口时钟
    GPIOA->CRL&=0XFFFF0000;//PA0 1 2 3 anolog 输入
```

```
    //通道 10/11 设置
    RCC->APB2ENR|=1<<9;       //ADC1 时钟使能
    RCC->APB2RSTR|=1<<9;      //ADC1 复位
    RCC->APB2RSTR&=~(1<<9);//复位结束
    RCC->CFGR&=~(3<<14);      //分频因子清零
    //SYSCLK/DIV2=12M ADC 时钟设置为 12M,ADC 最大时钟不能超过 14M!
    //否则将导致 ADC 准确度下降!
    RCC->CFGR|=2<<14;
    ADC1->CR1&=0XF0FFFF;      //工作模式清零
    ADC1->CR1|=0<<16;         //独立工作模式
    ADC1->CR1&=~(1<<8);       //非扫描模式
    ADC1->CR2&=~(1<<1);       //单次转换模式
    ADC1->CR2&=~(7<<17);
    ADC1->CR2|=7<<17;         //软件控制转换
    ADC1->CR2|=1<<20;         //使用用外部触发(SWSTART)!!! 必须使用一个事件来
触发
    ADC1->CR2&=~(1<<11);      //右对齐
    ADC1->CR2|=1<<23;         //使能温度传感器
    ADC1->SQR1&=~(0XF<<20);
    ADC1->SQR1&=0<<20;        //1 个转换在规则序列中 也就是只转换规则序列 1
    //设置通道 0~3 的采样时间
    ADC1->SMPR2&=0XFFFFF000;//通道 0,1,2,3 采样时间清空
    ADC1->SMPR2|=7<<9;        //通道 3   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<6;        //通道 2   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<3;        //通道 1   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR2|=7<<0;        //通道 0   239.5 周期,提高采样时间可以提高精确度
    ADC1->SMPR1&=~(7<<18);    //清除通道 16 原来的设置
    ADC1->SMPR1|=7<<18;       //通道 16   239.5 周期,提高采样时间可以提高精确度
    ADC1->CR2|=1<<0;          //开启 AD 转换器
    ADC1->CR2|=1<<3;          //使能复位校准
    while(ADC1->CR2&1<<3);    //等待校准结束
    //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。
    ADC1->CR2|=1<<2;          //开启 AD 校准
    while(ADC1->CR2&1<<2);    //等待校准结束
    //该位由软件设置以开始校准，并在校准结束时由硬件清除
}
```

这部分代码与上一节的 Adc_Init 代码几乎一摸一样，我们仅仅在里面增加了 ADC1->CR2|=1<<23; 这一句话，通过该句使能温度传感器，我们就能使用 STM32 的内部温度传感器了。

然后，我们保存一下该文件，接着打开 adc.h，修改文件如下：

```
#ifndef __ADC_H
#define __ADC_H
//Mini STM32 开发板
```

```
//ADC 驱动代码
//正点原子@ALIENTEK
#define ADC_CH0    0   //通道 0
#define ADC_CH1    1   //通道 1
#define ADC_CH2    2   //通道 2
#define ADC_CH3    3   //通道 3
#define TEMP_CH    16 //温度传感器通道
void Adc_Init(void); //ADC 通道初始化
u16    Get_Adc(u8 ch); //获得某个通道值
#endif
```

这里我们也只增加了一句，就是宏定义多增加了一个温度传感器通道 TEMP_CH。接下来我们就可以开始读取温度传感器的电压了。在 test.c 文件里面我们修改 main 函数如下：

```
int main(void)
{
    u16 adcx;
    float temp;
    float temperate;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();
    LCD_Init();
    Adc_Init();
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"TEMPERATE TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2010/12/30");
    //显示时间
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(60,130,"TEMP_VAL:");
    LCD_ShowString(60,150,"TEMP_VOL:0.000V");
    LCD_ShowString(60,170,"TEMPERATE:00.00C");
    while(1)
    {
        adcx=Get_Adc(TEMP_CH);
        LCD_ShowNum(132,130,adcx,4,16);//显示 ADC 的值
        temp=(float)adcx*(3.3/4096);
        temperate=temp;//保存温度传感器的电压值
        adcx=temp;
        LCD_ShowNum(132,150,adcx,1,16);        //显示电压值整数部分
        temp-=(u8)temp;                                  //减掉整数部分
        LCD_ShowNum(148,150,temp*1000,3,16);//显示电压小数部分
```

```
            temperate=(1.43-temperate)/0.0043+25;//计算出当前温度值
            LCD_ShowNum(140,170,(u8)temperate,2,16); //显示温度整数部分
            temperate-=(u8)temperate;
            LCD_ShowNum(164,170,temperate*100,2,16);//显示温度小数部分
            LED0=!LED0;
            delay_ms(250);
        }
    }
```

这里同上一节的主函数也大同小异，上面红色部分代码将温度传感器得到的电压值，换算成温度值。然后，我们在 TFTLCD 模块上显示出来。

代码设计部分就为大家讲解到这里，下面我们开始下载与测试。

3.14.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

图 3.14.4.1 内部温度传感器实验测试图

伴随 DS0 的不停闪烁，提示程序在运行。大家可以看看你的温度值与实际相差是否很大？（因为芯片会发热，一般会比实际温度稍高一些）

3.15 DMA实验

这一节我们将向大家介绍 STM32 的 DMA。本节将利用 STM32 的 DMA 来实现串口数据传送，并在 TFTLCD 模块上显示当前的传送进度。本节分为如下几个部分：

3.15.1 STM32 DMA 简介

3.15.2 硬件设计

3.15.3 软件设计

3.15.4 下载与测试

3.15.1 STM32 DMA 简介

DMA，全称为：Direct Memory Access，即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。

STM32 最多有 2 个 DMA 控制器（DMA2 仅存在大容量产品中），DMA1 有 7 个通道。DMA2 有 5 个通道。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁起来协调各个 DMA 请求的优先权。

STM32 的 DMA 有以下一些特性：

●每个通道都直接连接专用的硬件 DMA 请求，每个通道都同样支持软件触发。这些功能通过软件来配置。

●在七个请求间的优先权可以通过软件编程设置(共有四级：很高、高、中等和低)，假如在相等优先权时由硬件决定(请求 0 优先于请求 1，依此类推) 。

●独立的源和目标数据区的传输宽度(字节、半字、全字)，模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。

●支持循环的缓冲器管理

●每个通道都有 3 个事件标志(DMA 半传输，DMA 传输完成和 DMA 传输出错)，这 3 个事件标志逻辑或成为一个单独的中断请求。

●存储器和存储器间的传输

●外设和存储器，存储器和外设的传输

●闪存、SRAM、外设的 SRAM、APB1 APB2 和 AHB 外设均可作为访问的源和目标。

●可编程的数据传输数目：最大为 65536

STM32F103RBT6 只有 1 个 DMA 控制器，DMA1，下面我们就针对 DMA1 进行介绍。

从外设（TIMx、ADC、SPIx、I2Cx 和 USARTx）产生的 DMA 请求，通过逻辑或输入到 DMA 控制器，这就意味着同时只能有一个请求有效。外设的 DMA 请求，可以通过设置相应的外设寄存器中的控制位，被独立地开启或关闭。

DMA1 各通道一览表：

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC	ADC1						
SPI		SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

表 3.15.1.1 DMA1 个通道一览表

这里解释一下上面说的逻辑或，例如通道 1 的几个 DMA1 请求（ADC1、TIM2_CH3、TIM4_CH1），这几个是通过逻辑或到通道 1 的，这样我们在同一时间，就只能使用其中的一个。其他通道也是类似的。

这里我们要使用的是串口 1 的 DMA 传送，也就是要用到通道 4。接下来，我们介绍一下 DMA 设置相关的几个寄存器。

第一个是 DMA 中断状态寄存器（DMA_ISR）。该寄存器的各位描述如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5
				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
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**TEIFx**：通道x的传输错误标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输错误(TE)； 1：在通道x发生了传输错误(TE)。
位26，22，18，14，10，6，2	**HTIFx**：通道x的半传输标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有半传输事件(HT)； 0：在通道x产生了半传输事件(HT)。
位25，21，17，13，9，5，1	**TCIFx**：通道x的传输完成标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输完成事件(TC)； 0：在通道x产生了传输完成事件(TC)。
位24，20，16，12，8，4，0	**GIFx**：通道x的全局中断标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有TE、HT或TC事件； 0：在通道x产生了TE、HT或TC事件。

图 3.15.1.1 寄存器 DMA_ISR 各位描述

我们如果开启了 DMA_ISR 中这些中断，在达到条件后就会跳到中断服务函数里面去，即使没开启，我们也可以通过查询这些位来获得当前 DMA 传输的状态。这里我们常用的是 TCIFx，即通道 DMA 传输完成与否的标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除。

第二个是 DMA 中断标志清除寄存器（DMA_IFCR）。该寄存器的各位描述如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				CTEIF7	CHTIF7	CTCIF7	CGIF7	CTEIF6	CHTIF6	CTCIF6	CGIF6	CTEIF5	CHTIF5	CTCIF5	CGIF5
				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
CTEIF4	CHTIF4	CTCIF4	CGIF4	CTEIF3	CHTIF3	CTCIF3	CGIF3	CTEIF2	CHTIF2	CTCIF2	CGIF2	CTEIF1	CHTIF1	CTCIF1	CGIF1
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**CTEIFx**：清除通道x的传输错误标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 1：清除DMA_ISR寄存器中的对应TEIF标志。
位26，22，18，14，10，6，2	**CHTIFx**：清除通道x的半传输标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应HTIF标志。
位25，21，17，13，9，5，1	**CTCIFx**：清除通道x的传输完成标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应TCIF标志。
位24，20，16，12，8，4，0	**CGIFx**：清除通道x的全局中断标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应的GIF、TEIF、HTIF和TCIF标志。

图 3.15.1.2 寄存器 DMA_IFCR 各位描述

DMA_IFCR 的各位就是用来清除 DMA_ISR 的对应位的，通过写 0 清除。在 DMA_ISR 被置位后，我们必须通过向该位寄存器对应的位写入 0 来清除。

第三个是 DMA 通道 x 配置寄存器（DMA_CCRx）（x=1~7，下同）。该寄存器的我们在这里就不贴出来了，见《STM32 参考手册》第 108 页 9.4.3 一节。该寄存器控制着 DMA 的很多相关信息，包括数据宽度、外设及存储器的宽度、通道优先级、增量模式、传输方向、中断允许、使能等都是通过该寄存器来设置的。所以 DMA_CCRx 是 DMA 传输的核心控制寄存器。

第四个是 DMA 通道 x 传输数据量寄存器（DMA_CNDTRx）。这个寄存器控制 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值会随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成了。所以可以通过这个寄存器的值来知道当前 DMA 传输的进度。

第五个是 DMA 通道 x 的外设地址寄存器（DMA_CPARx）。该寄存器用来存储 STM32 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40013804（其实就是&USART1_DR）。如果使用其他外设，就修改成相应外设的地址就行了。

最后一个是 DMA 通道 x 的存储器地址寄存器（DMA_CMARx），该寄存器和 DMA_CPARx 差不多，但是是用来放存储器的地址的。比如我们使用 SendBuf[5200]数组来做存储器，那么我们在 DMA_CMARx 中写入&SendBuff 就可以了。

DMA 相关寄存器就为大家介绍到这里，此节我们要用到串口 1 的发送，属于 DMA1 的通道 4，接下来我们就介绍下 DMA1 通道 4 的配置步骤：

1）设置外设地址。

设置外设地址通过 DMA1_CPAR4 来设置，我们只要在这个寄存器里面写入&USART1_DR 的值就可以了。该地址将作为 DMA 传输的目标地址。

2）设置存储器地址。

设置存储器地址，我们通过 DMA1_CMAR4 来设置，假设我们要把数组 SendBuf 作为存储器，那么我们在该寄存器写入&SendBuf 就可以了。该地址将作为 DMA 传输的源地址。

3）设置传输数据量。

通过 DMA1_CNDTR4 来设置 DMA1 通道 4 的数据传输量,这里面写入此次你要传输的数据量就可以了，也就是 SendBuf 的大小。该寄存器的数值将在 DMA 启动后自减，每次新的 DMA 传输，都重新向该寄存器写入要传输的数据量。

4）设置通道 4 的配置信息。

配置信息通过 DMA1_CCR4 来设置。这里我们设置存储器和外设的数据位宽均为 8，且模式是存储器到外设的存储器增量模式。优先级可以随便设置，因为我们只有一个通道被开启了。假设有多个通道开启（最多 7 个），那么就要设置优先级了，DMA 仲裁器将根据这些优先级的设置来决定先执行那个通道的 DMA。优先级越高的，越早执行，当优先级相同的时候，根据硬件上的编号来决定哪个先执行（编号越小越优先）。

5）使能 DMA1 通道 4，启动传输。

在以上配置都完成了之后，我们就使能 DMA1_CCR4 的最低位开启 DMA 传输，这里注意要设置 USART1 的使能 DMA 传输位，通过 USART1->CR3 的第七位设置。

通过以上 5 步设置，我们就可以启动一次 USART1 的 DMA 传输了。

3.15.2 硬件设计

这一节我们将利用外部按键 KEY0 来控制 DMA 的传送，每按一次 KEY0，DMA 就传送一次数据到 USART1，然后在 TFTLCD 模块上显示进度等信息。DS0 还是用来做为程序运行的指示灯。

这里我们使用到的硬件资源如下：

1)按键 KEY0。

2)指示灯 DS0。

3)TFTLCD 液晶显示模块。

这些硬件的电路连接，我们在前面都介绍过了，请大家参阅前面的章节。

3.15.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 DMA 的文件夹。然后新建一个 dma.c 的文件和 dma.h 的头文件，保存在 DMA 文件夹下，并将 DMA 文件夹加入头文件包含路径。

打开 dma.c 文件，输入如下代码：

```
#include "dma.h"
//Mini STM32 开发板
//DMA 驱动代码
//正点原子@ALIENTEK
u16 DMA1_MEM_LEN;//保存 DMA 每次数据传送的长度
//DMA1 的各通道配置
//这里的传输形式是固定的,这点要根据不同的情况来修改
//从存储器->外设模式/8 位数据宽度/存储器增量模式
//DMA_CHx:DMA 通道 CHx
```

```
//cpar:外设地址
//cmar:存储器地址
//cndtr:数据传输量
void MYDMA_Config(DMA_Channel_TypeDef*DMA_CHx,u32 cpar,u32 cmar,u16 cndtr)
{
    u32 DR_Base;  //做缓冲用,不知道为什么.非要不可
    RCC->AHBENR|=1<<0;//开启 DMA1 时钟
    DR_Base=cpar;
    DMA_CHx->CPAR=DR_Base;    //DMA1 外设地址
    DMA_CHx->CMAR=(u32)cmar; //DMA1,存储器地址
    DMA1_MEM_LEN=cndtr;       //保存 DMA 传输数据量
    DMA_CHx->CNDTR=cndtr;     //DMA1,传输数据量
    DMA_CHx->CCR=0X00000000;//复位
    DMA_CHx->CCR|=1<<4;  //从存储器读
    DMA_CHx->CCR|=0<<5;  //普通模式
    DMA_CHx->CCR|=0<<6;  //外设地址非增量模式
    DMA_CHx->CCR|=1<<7;  //存储器增量模式
    DMA_CHx->CCR|=0<<8;  //外设数据宽度为 8 位
    DMA_CHx->CCR|=0<<10; //存储器数据宽度 8 位
    DMA_CHx->CCR|=1<<12; //中等优先级
    DMA_CHx->CCR|=0<<14; //非存储器到存储器模式
}
//开启一次 DMA 传输
void MYDMA_Enable(DMA_Channel_TypeDef*DMA_CHx)
{
    DMA_CHx->CCR&=~(1<<0);        //关闭 DMA 传输
    DMA_CHx->CNDTR=DMA1_MEM_LEN; //DMA1,传输数据量
    DMA_CHx->CCR|=1<<0;           //开启 DMA 传输
}
```

该部分代码仅仅 2 个函数，MYDMA_Config 函数，基本上就是按照我们上面介绍的步骤来初始化 DMA 的，该函数在外部只能修改通道、源地址、目标地址和传输数据量等几个参数，跟多的其他设置只能在该函数内部修改。

MYDMA_Enable 函数用来产生一次 DMA 传输，该函数每执行一次，DMA 就发送一次。

保存 dma.c，并把 dma.c 加入到 HARDWARE 组下，接下来打开 dma.h，输入如下内容：

```
#ifndef __DMA_H
#define __DMA_H
#include "sys.h"
//Mini STM32 开发板
//DMA 驱动代码
//正点原子@ALIENTEK
void MYDMA_Config(DMA_Channel_TypeDef*DMA_CHx,u32 cpar,u32 cmar,u16 cndtr);//
配置 DMA1_CHx
void MYDMA_Enable(DMA_Channel_TypeDef*DMA_CHx);//使能 DMA1_CHx
```

```
#endif
```

保存 dma.h，最后我们在 test.c 里面修改 main 函数如下：

```
u8 SendBuff[5200];
const u8 TEXT_TO_SEND[]={"ALIENTEK MiniSTM32 DMA 串口实验"};
int main(void)
{
    u16 i;
    u8 t=0;
    u8 j,mask=0;
    float pro=0;//进度
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();
    KEY_Init();
    LCD_Init();
    MYDMA_Config(DMA1_Channel4,(u32)&USART1->DR,(u32)SendBuff,5200);//DMA1  通
道 4,外设为串口 1,存储器为 SendBuff,长度 5200.

    POINT_COLOR=RED;//设置字体为蓝色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"DMA USART TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2010/6/7");
    LCD_ShowString(60,130,"Press KEY0 To Start");
    j=sizeof(TEXT_TO_SEND);
    for(i=0;i<5200;i++)//填充 ASCII 字符集数据
    {
        if(t>=j)//加入换行符
        {
            if(mask)
            {
                SendBuff[i]=0x0a;
                t=0;
            }else
            {
                SendBuff[i]=0x0d;
                mask++;
            }
        }else//复制 TEXT_TO_SEND 语句
        {
            mask=0;
            SendBuff[i]=TEXT_TO_SEND[t];
```

```
            t++;
        }
    }
    POINT_COLOR=BLUE;//设置字体为蓝色
    i=0;
    while(1)
    {
        t=KEY_Scan();
        if(t==1)//KEY0 按下
        {
            LCD_ShowString(60,150,"Start Transimit....");
            LCD_ShowString(60,170,"   %");//显示百分号
            printf("\n\nDMA DATA:\n");
            USART1->CR3=1<<7;           //使能串口 1 的 DMA 发送
            MYDMA_Enable(DMA1_Channel4);//开始一次 DMA 传输!
            //等待 DMA 传输完成，此时我们来做另外一些事，点灯
            //实际应用中，传输数据期间，可以执行另外的任务
            while(1)
            {
                if(DMA1->ISR&(1<<13))//等待通道 4 传输完成
                {
                    DMA1->IFCR|=1<<13;//清除通道 4 传输完成标志
                    break;
                }
                pro=DMA1_Channel4->CNDTR;//得到当前还剩余多少个数据
                pro=1-pro/5200;//得到百分比
                pro*=100;      //扩大 100 倍
                LCD_ShowNum(60,170,pro,3,16);
            }
            LCD_ShowNum(60,170,100,3,16);//显示 100%
            LCD_ShowString(60,150,"Transimit Finished!");//提示传送完成
        }
        i++;
        delay_ms(1);
        if(i==200)
        {
            LED0=!LED0;//提示系统正在运行
            i=0;
        }
    }
}
```

至此，DMA 串口传输的软件设计就完成了。

3.15.4 下载与测试

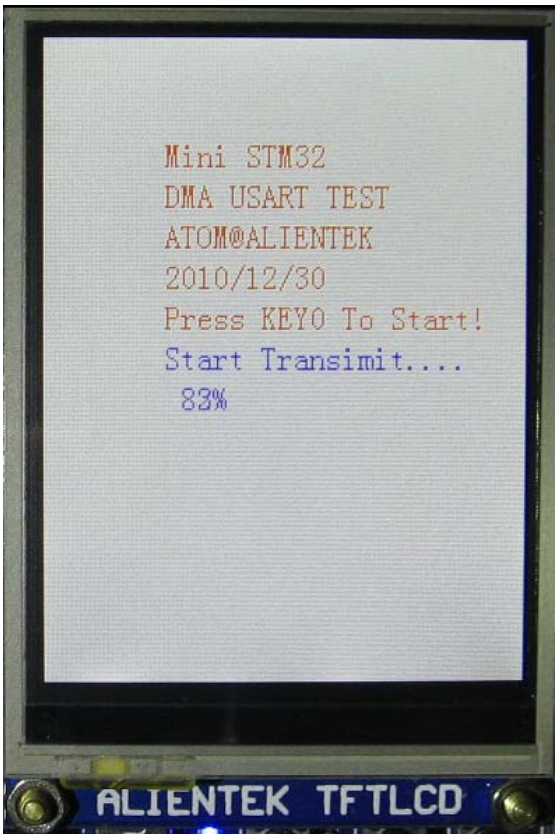
在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

图 13.15.4.1 DMA 串口实验实物测试图

伴随 DS0 的不停闪烁，提示程序在运行。我们打开串口调试助手，然后按 KEY0，可以看到串口显示如下内容：

图 13.15.4.2 串口收到的数据内容

同时可以看到 TFTLCD 上显示了进度等信息，如下图所示：



13.15.4.3 DMA 串口数据传输中

至此，我们整个 DMA 串口实验就结束了，希望大家通过这节的学习，掌握 STM32 的 DMA 使用。DMA 是个非常好的功能，他不但能减轻 CPU 负担，还能提高数据传输速度，合理的应用 DMA，往往能让您的程序设计变得简单。

3.16 IIC实验

这一节我们将向大家介绍 IIC。本节将利用 IIC 来实现 24C02 的读写，并将结果显示在 TFTLCD 模块上。本节分为如下几个部分：

3.16.1 IIC 简介

3.16.2 硬件设计

3.16.3 软件设计

3.16.4 下载与测试

3.16.1 IIC 简介

IIC(Inter－Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，高速 IIC 总线一般可达 400kbps 以上。

I2C 总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

这些信号中，起始信号是必需的，结束信号和应答信号，都可以不要。IIC 总线时序图如下：

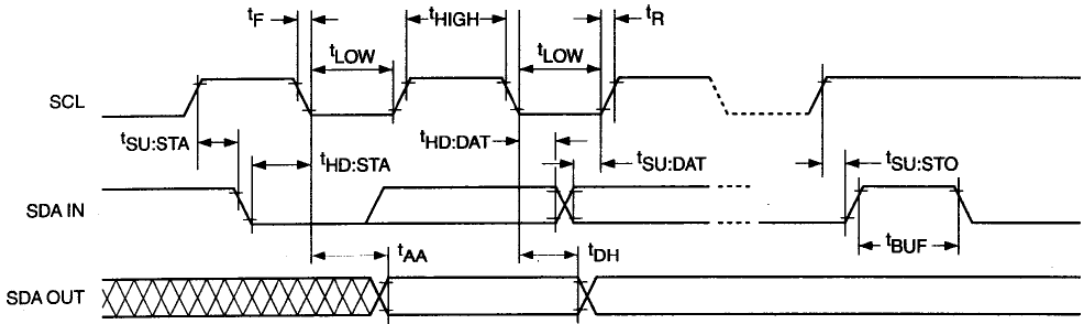


图 3.16.1.1 IIC 总线时序图

ALIENTEK MiniSTM32 开发板板载了 EEPROM 芯片：。该芯片的总容量是 256 个字节，该芯片通过 IIC 总线与外部连接，我们本节就通过 STM32 来实现 24C02 的读写。

目前大部分 MCU 都带有 IIC 总线接口，STM32 也不例外。但是这里我们不使用 STM32 的硬件 IIC 来读写 24C02，而是通过软件模拟。因为 STM32 的 IIC 实在太难用了，一个很简单的东西，ST 的人把它弄得很复杂，不得不说 STM32 的 IIC 很鸡肋。所以我们这里就通过模拟来实现了。有兴趣的大家可以研究一下 STM32 的硬件 IIC。

本节实验功能简介：开机的时候先检测 24C02 是否存在，然后在主循环里面用 1 个按键用来执行写入 24C02 的操作，另外一个按键用来执行读出操作，在 TFTLCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

3.16.2 硬件设计

本节所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）KEY0 和 KEY2。

4）TFTLCD 液晶模块。

5）24C02。

前面 4 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里只介绍 24C02 与 STM32 的连接，板上的 24C02 是直接连在 STM32F103RBT6 上的，连接关系如下图：

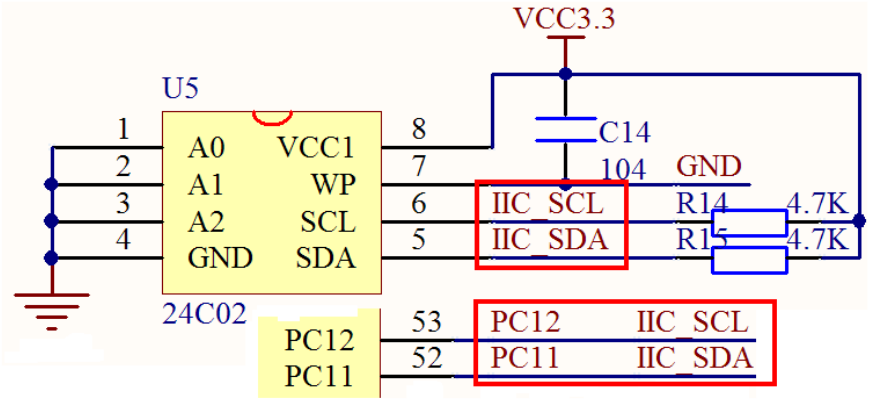


图 3.16.2.1　STM32F103RBT6 与 24C02 连接图

3.16.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 24CXX 的文件夹。然后新建一个 24cxx.c、myiic.c 的文件和 24cxx.h、myiic.h 的头文件，保存在 24CXX 文件夹下，并将 24CXX 文件夹加入头文件包含路径。

打开 myiic.c 文件，输入如下代码：

```
#include "myiic.h"
#include "delay.h"
//STM32 软件模拟 IIC，STM32 的硬件 IIC 太难用了！
//Mini STM32 开发板
//IIC  驱动函数
//正点原子@ALIENTEK
//初始化 IIC
void IIC_Init(void)
{
    RCC->APB2ENR|=1<<4;//先使能外设 IO PORTC 时钟
    GPIOC->CRH&=0XFFF00FFF;//PC11/12  推挽输出
    GPIOC->CRH|=0X00033000;
    GPIOC->ODR|=3<<11;        //PC11,12  输出高
}
//产生 IIC 起始信号
void IIC_Start(void)
{
    SDA_OUT();        //sda 线输出
```

```
    IIC_SDA=1;
    IIC_SCL=1;
    delay_us(4);
    IIC_SDA=0;//START:when CLK is high,DATA change form high to low
    delay_us(4);
    IIC_SCL=0;//钳住 I2C 总线，准备发送或接收数据
}
//产生 IIC 停止信号
void IIC_Stop(void)
{
    SDA_OUT();//sda 线输出
    IIC_SCL=0;
    IIC_SDA=0;//STOP:when CLK is high DATA change form low to high
    delay_us(4);
    IIC_SCL=1;
    IIC_SDA=1;//发送 I2C 总线结束信号
    delay_us(4);
}
//等待应答信号到来
//返回值：1，接收应答失败
//        0，接收应答成功
u8 IIC_Wait_Ack(void)
{
    u8 ucErrTime=0;
    SDA_IN();       //SDA 设置为输入
    IIC_SDA=1;delay_us(1);
    IIC_SCL=1;delay_us(1);
    while(READ_SDA)
    {
        ucErrTime++;
        if(ucErrTime>250)
        {
            IIC_Stop();
            return 1;
        }
    }
    IIC_SCL=0;//时钟输出 0
    return 0;
}
//产生 ACK 应答
void IIC_Ack(void)
{
    IIC_SCL=0;
```

```
    SDA_OUT();
    IIC_SDA=0;
    delay_us(2);
    IIC_SCL=1;
    delay_us(2);
    IIC_SCL=0;
}
//不产生 ACK 应答
void IIC_NAck(void)
{
    IIC_SCL=0;
    SDA_OUT();
    IIC_SDA=1;
    delay_us(2);
    IIC_SCL=1;
    delay_us(2);
    IIC_SCL=0;
}
//IIC 发送一个字节
//返回从机有无应答
//1，有应答
//0，无应答
void IIC_Send_Byte(u8 txd)
{
    u8 t;
    SDA_OUT();
    IIC_SCL=0;//拉低时钟开始数据传输
    for(t=0;t<8;t++)
    {
        IIC_SDA=(txd&0x80)>>7;
        txd<<=1;
        delay_us(2);    //对 TEA5767 这三个延时都是必须的
        IIC_SCL=1;
        delay_us(2);
        IIC_SCL=0;
        delay_us(2);
    }
}
//读 1 个字节，ack=1 时，发送 ACK，ack=0，发送 nACK
u8 IIC_Read_Byte(unsigned char ack)
{
    unsigned char i,receive=0;
    SDA_IN();//SDA 设置为输入
```

```
    for(i=0;i<8;i++ )
    {
        IIC_SCL=0;
        delay_us(2);
        IIC_SCL=1;
        receive<<=1;
        if(READ_SDA)receive++;
        delay_us(1);
    }
    if (!ack)
        IIC_NAck();//发送 nACK
    else
        IIC_Ack(); //发送 ACK
    return receive;
}
```

该部分为 IIC 驱动代码，实现包括 IIC 的初始化（IO 口）、IIC 开始、IIC 结束、ACK、IIC 读写等功能，在其他函数里面，只需要调用相关的 IIC 函数就可以和外部 IIC 器件通信了，这里并不局限于 24C02。

保存该部分代码，把 myiic.c 加入到 HARDWARE 组下面，然后在 myiic.h 里面输入如下代码：

```
#ifndef __MYIIC_H
#define __MYIIC_H
#include "sys.h"
//Mini STM32 开发板
//IIC  驱动函数
//正点原子@ALIENTEK
//IO 方向设置
#define SDA_IN()  {GPIOC->CRH&=0XFFFF0FFF;GPIOC->CRH|=8<<12;}
#define SDA_OUT() {GPIOC->CRH&=0XFFFF0FFF;GPIOC->CRH|=3<<12;}
//IO 操作函数
#define IIC_SCL     PCout(12) //SCL
#define IIC_SDA     PCout(11) //SDA
#define READ_SDA    PCin(11)  //输入 SDA
//IIC 所有操作函数
void IIC_Init(void);                    //初始化 IIC 的 IO 口
void IIC_Start(void);                   //发送 IIC 开始信号
void IIC_Stop(void);                    //发送 IIC 停止信号
void IIC_Send_Byte(u8 txd);             //IIC 发送一个字节
u8 IIC_Read_Byte(unsigned char ack);//IIC 读取一个字节
u8 IIC_Wait_Ack(void);                  //IIC 等待 ACK 信号
void IIC_Ack(void);                     //IIC 发送 ACK 信号
void IIC_NAck(void);                    //IIC 不发送 ACK 信号
```

```
void IIC_Write_One_Byte(u8 daddr,u8 addr,u8 data);
u8 IIC_Read_One_Byte(u8 daddr,u8 addr);
#endif
```

接下来我们在 24cxx.c 文件里面输入如下代码：

```
#include "24cxx.h"
#include "delay.h"
//Mini STM32 开发板
//24CXX 驱动函数(适合~24C16,24C32~256 未经过测试!有待验证!)
//正点原子@ALIENTEK
//初始化 IIC 接口
void AT24CXX_Init(void)
{
    IIC_Init();
}
//在 AT24CXX 指定地址读出一个数据
//ReadAddr:开始读数的地址
//返回值  :读到的数据
u8 AT24CXX_ReadOneByte(u16 ReadAddr)
{
    u8 temp=0;
IIC_Start();
    if(EE_TYPE>AT24C16)
    {
        IIC_Send_Byte(0XA0);      //发送写命令
        IIC_Wait_Ack();
        IIC_Send_Byte(ReadAddr>>8);//发送高地址
        IIC_Wait_Ack();
    }else  IIC_Send_Byte(0XA0+((ReadAddr/256)<<1));      //发送器件地址  0XA0,写数据
IIC_Wait_Ack();
    IIC_Send_Byte(ReadAddr%256);     //发送低地址
    IIC_Wait_Ack();
    IIC_Start();
    IIC_Send_Byte(0XA1);                //进入接收模式
    IIC_Wait_Ack();
    temp=IIC_Read_Byte(0);
    IIC_Stop();//产生一个停止条件
    return temp;
}
//在 AT24CXX 指定地址写入一个数据
//WriteAddr  :写入数据的目的地址
//DataToWrite:要写入的数据
void AT24CXX_WriteOneByte(u16 WriteAddr,u8 DataToWrite)
{
```

```
    IIC_Start();
    if(EE_TYPE>AT24C16)
    {
        IIC_Send_Byte(0XA0);        //发送写命令
        IIC_Wait_Ack();
        IIC_Send_Byte(WriteAddr>>8);//发送高地址
    }else
    {
        IIC_Send_Byte(0XA0+((WriteAddr/256)<<1));    //发送器件地址 0XA0,写数据
    }
    IIC_Wait_Ack();
    IIC_Send_Byte(WriteAddr%256);    //发送低地址
    IIC_Wait_Ack();
    IIC_Send_Byte(DataToWrite);        //发送字节
    IIC_Wait_Ack();
    IIC_Stop();//产生一个停止条件
    delay_ms(10);
}
//在 AT24CXX 里面的指定地址开始写入长度为 Len 的数据
//该函数用于写入 16bit 或者 32bit 的数据.
//WriteAddr  :开始写入的地址
//DataToWrite:数据数组首地址
//Len        :要写入数据的长度 2,4
void AT24CXX_WriteLenByte(u16 WriteAddr,u32 DataToWrite,u8 Len)
{
    u8 t;
    for(t=0;t<Len;t++)
    {
        AT24CXX_WriteOneByte(WriteAddr+t,(DataToWrite>>(8*t))&0xff);
    }
}

//在 AT24CXX 里面的指定地址开始读出长度为 Len 的数据
//该函数用于读出 16bit 或者 32bit 的数据.
//ReadAddr   :开始读出的地址
//返回值     :数据
//Len        :要读出数据的长度 2,4
u32 AT24CXX_ReadLenByte(u16 ReadAddr,u8 Len)
{
    u8 t;
    u32 temp=0;
    for(t=0;t<Len;t++)
```

```
    {
        temp<<=8;
        temp+=AT24CXX_ReadOneByte(ReadAddr+Len-t-1);
    }
    return temp;
}
//检查 AT24CXX 是否正常
//这里用了 24XX 的最后一个地址(255)来存储标志字.
//如果用其他 24C 系列,这个地址要修改
//返回 1:检测失败
//返回 0:检测成功
u8 AT24CXX_Check(void)
{
    u8 temp;
    temp=AT24CXX_ReadOneByte(255);//避免每次开机都写 AT24CXX
    if(temp==0X55)return 0;
    else//排除第一次初始化的情况
    {
        AT24CXX_WriteOneByte(255,0X55);
        temp=AT24CXX_ReadOneByte(255);
        if(temp==0X55)return 0;
    }
    return 1;
}

//在 AT24CXX 里面的指定地址开始读出指定个数的数据
//ReadAddr :开始读出的地址 对 24C02 为 0~255
//pBuffer   :数据数组首地址
//NumToRead:要读出数据的个数
void AT24CXX_Read(u16 ReadAddr,u8 *pBuffer,u16 NumToRead)
{
    while(NumToRead)
    {
        *pBuffer++=AT24CXX_ReadOneByte(ReadAddr++);
        NumToRead--;
    }
}
//在 AT24CXX 里面的指定地址开始写入指定个数的数据
//WriteAddr :开始写入的地址 对 24C02 为 0~255
//pBuffer     :数据数组首地址
//NumToWrite:要写入数据的个数
void AT24CXX_Write(u16 WriteAddr,u8 *pBuffer,u16 NumToWrite)
{
```

```
        while(NumToWrite--)
        {
            AT24CXX_WriteOneByte(WriteAddr,*pBuffer);
            WriteAddr++;
            pBuffer++;
        }
    }
```

这部分代码理论上是可以支持 24Cxx 所有系列的芯片的（地址引脚必须都设置为 0），但是我们测试只测试了 24C02，其他器件有待测试。大家也可以验证一下，24CXX 的型号定义在 24cxx.h 文件里面，通过 EE_TYPE 设置。

保存该部分代码，把 24cxx.c 加入到 HARDWARE 组下面，然后在 24cxx.h 里面输入如下代码：

```
#ifndef __24CXX_H
#define __24CXX_H
#include "myiic.h"
//Mini STM32 开发板
//24CXX 驱动函数(适合~24C16,24C32~256 未经过测试!有待验证!)
//正点原子@ALIENTEK
#define AT              127
#define AT24C02         255
#define AT24C04         511
#define AT24C08         1023
#define AT24C16         2047
#define AT24C32         4095
#define AT24C64         8191
#define AT24C128 16383
#define AT24C256 32767
//Mini STM32 开发板使用的是 24C02，所以定义 EE_TYPE 为 AT24C02
#define EE_TYPE AT24C02
u8 AT24CXX_ReadOneByte(u16 ReadAddr);                           //指定地址读取一个字节
void AT24CXX_WriteOneByte(u16 WriteAddr,u8 DataToWrite);        //指定地址写入一个字节
void AT24CXX_WriteLenByte(u16 WriteAddr,u32 DataToWrite,u8 Len);//指定地址开始写入指定长度的数据
u32 AT24CXX_ReadLenByte(u16 ReadAddr,u8 Len);                   //指定地址开始读取指定长度数据
void AT24CXX_Write(u16 WriteAddr,u8 *pBuffer,u16 NumToWrite);   //从指定地址开始写入指定长度的数据
void AT24CXX_Read(u16 ReadAddr,u8 *pBuffer,u16 NumToRead);      //从指定地址开始读出指定长度的数据

u8 AT24CXX_Check(void);  //检查器件
```

```
void AT24CXX_Init(void); //初始化 IIC
#endif
```

最后，我们在 main 函数里面编写应用代码，在 test.c 里面，修改 main 函数如下：

```
//要写入到 24C02 的字符串数组
const u8 TEXT_Buffer[]={"MiniSTM32 IIC TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];

    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LED_Init();         //LED 初始化
    KEY_Init();         //按键初始化
    LCD_Init();         //TFTLCD 液晶初始化
    AT24CXX_Init();     //IIC 初始化
    POINT_COLOR=RED;//设置字体为蓝色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"IIC TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2011/1/1");
    while(AT24CXX_Check())//检测不到 24C02
    {
        LCD_ShowString(60,130,"24C02 Check Failed!");
        delay_ms(500);
        LCD_ShowString(60,130,"Please Check!      ");
        delay_ms(500);
        LED0=!LED0;//DS0 闪烁
    }
    LCD_ShowString(60,130,"24C02 Ready!");
    //显示提示信息
    LCD_ShowString(60,150,"KEY0:Write KEY2:Read");

    POINT_COLOR=BLUE;//设置字体为蓝色
    while(1)
    {
        key=KEY_Scan();
        if(key==1)//KEY0 按下,写入 24C02
        {
            LCD_Fill(0,170,239,319,WHITE);//清除半屏
```

```
                LCD_ShowString(60,170,"Start Write 24C02....");
                AT24CXX_Write(0,(u8*)TEXT_Buffer,SIZE);
                LCD_ShowString(60,170,"24C02 Write Finished!");//提示传送完成
            }
            if(key==3)//KEY1 按下,读取字符串并显示
            {
                LCD_ShowString(60,170,"Start Read 24C02.... ");
                AT24CXX_Read(0,datatemp,SIZE);
                LCD_ShowString(60,170,"The Data Readed Is:   ");//提示传送完成
                LCD_ShowString(60,190,datatemp);//显示读到的字符串
            }
            i++;
            delay_ms(1);
            if(i==200)
            {
                LED0=!LED0;//提示系统正在运行
                i=0;
            }
        }
    }
```

至此，我们的软件设计部分就结束了。

3.16.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

图 3.16.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。我们先按下 KEY0，可以看到如图 13.16.4.2 所示的内容，证明数据已经被写入到 24C02 了。

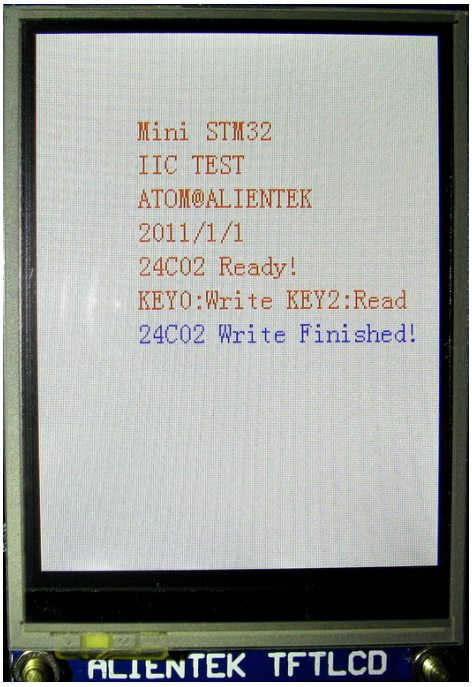


图 3.16.4.2 数据成功写入 24C02

接着我们按 KEY2，可以看我们刚刚写入的数据被显示出来了，如下图所示：

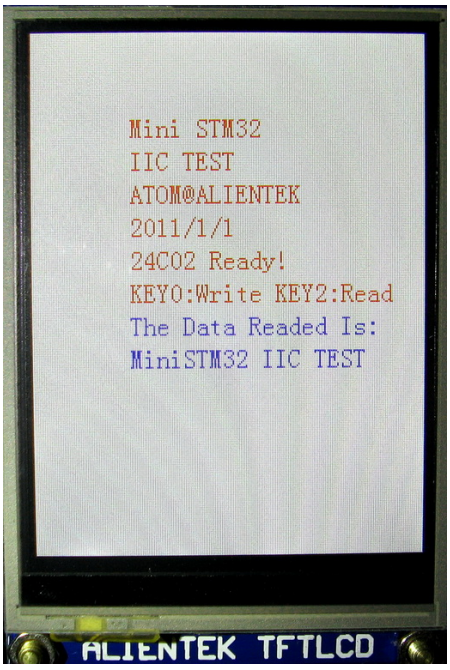


图 3.16.4.3 显示写入的内容

程序在开机的时候会检测 24C02 是否存在，如果不存在则会在 TFTLCD 模块上显示错误信息，同时 DS0 慢闪。大家可以通过跳线帽把 PC11 和 PC12 短接就可以看到报错了。

3.17 SPI 实验

这一节我们将向大家介绍 SPI。本节将利用 SPI 来实现对外部 FLASH（W25X16）的读写，并将结果显示在 TFTLCD 模块上。本节分为如下几个部分：

3.17.1 SPI 简介
3.17.2 硬件设计
3.17.3 软件设计
3.17.4 下载与测试

3.17.1 SPI 简介

SPI 是英语 Serial Peripheral interface 的缩写，顾名思义就是串行外围设备接口。是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要应用在 EEPROM，FLASH，实时时钟，AD 转换器，还有数字信号处理器和数字信号解码器之间。SPI，是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议，STM32 也有 SPI 接口。

SPI 接口一般使用 4 条线：

MISO 主设备数据输入，从设备数据输出。

MOSI 主设备数据输出，从设备数据输入。

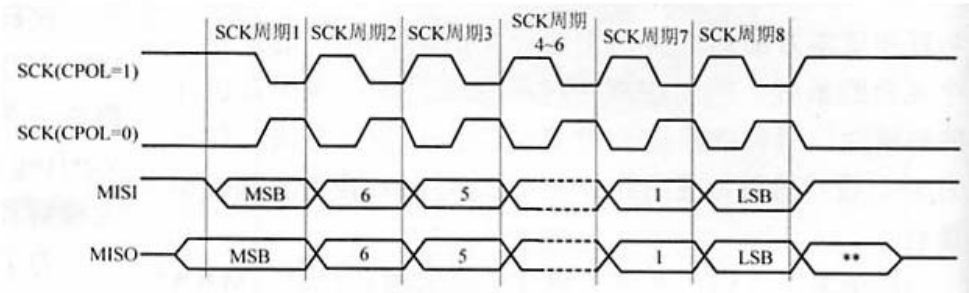
SCLK 时钟信号，由主设备产生。

CS 从设备片选信号，由主设备控制。

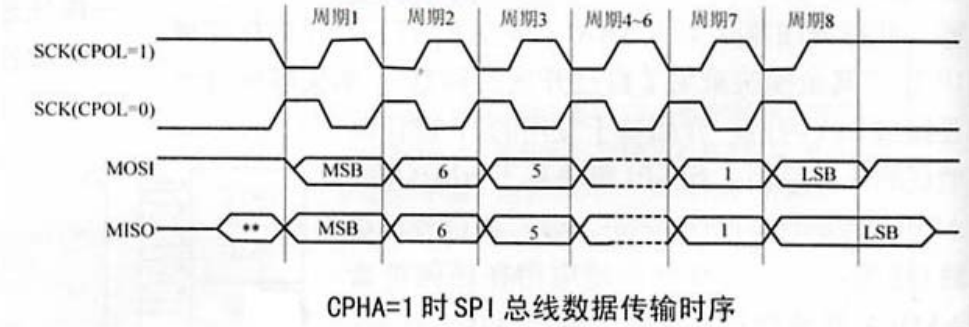
SPI 主要特点有：可以同时发出和接收串行数据；可以当作主机或从机工作；提供频率可编程时钟；发送结束中断标志；写冲突保护；总线竞争保护等。

SPI 总线四种工作方式 SPI 模块为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大的影响。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。时钟相位（CPHA）能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0，在串行同步时钟的第一个跳变沿（上升或下降）数据被采样；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样。SPI 主模块和与之通信的外设备时钟相位和极性应该一致。

不同时钟相位下的总线数据传输时序见下图：



CPHA=0 时 SPI 总线数据传输时序



CPHA=1 时 SPI 总线数据传输时序

图 3.17.1.1 不同时钟相位下的总线传输时序（CPHA=0/1）

STM32 的 SPI 功能很强大，SPI 时钟最多可以到 18Mhz，支持 DMA，可以配置为 SPI 协议或者 I2S 协议。

本节，我们将利用 STM32 的 SPI 来读取外部 SPI FLASH 芯片（W25X16），实现类似上节的功能。这里对 SPI 我们只简单介绍一下 SPI 的使用，STM32 的 SPI 详细介绍请参考《STM32 参考手册》第 457 页，23 节。然后我们再介绍下 SPI FLASH 芯片。

这节，我们使用 STM32 的 SPI1 的主模式，下面就来看看 SPI1 部分的设置步骤吧，STM32 的主模式配置步骤如下：

1）配置相关引脚的复用功能，使能 SPI1 时钟。

我们要用 SPI1，第一步就要是能 SPI1 的时钟，SPI1 的时钟通过 APB2ENR 的第 12 位来设置。其次要设置 SPI1 的相关引脚为复用输出，这样才会连接到 SPI1 上否则这些 IO 口还是默认的状态，也就是标准输入输出口。这里我们使用的是 PA5、6、7 这 3 个（SCK.、MISO、MOSI，CS 使用软件管理方式），所以设置这三个为复用 IO。

2）设置 SPI1 工作模式。

这一步全部是通过 SPI1_CR1 来设置，我们设置 SPI1 为主机模式，设置数据格式为 8 位，然后通过 CPOL 和 CPHA 位来设置 SCK 时钟极性及采样方式。并设置 SPI1 的时钟频率（最大 18Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。

3）使能 SPI1。

这一步通过 SPI1_CR1 的 bit6 来设置，以启动 SPI1，在启动之后，我们就可以开始 SPI 通讯了。

SPI1 的使用就介绍到这里，接下来介绍一下 W25X16。W25X16 是华邦公司推出的继 W25X10/20/40/80（从 1Mb~8Mb）后容量更大的 FLASH 产品，W25X16 的容量为 16Mb，还有容量更大的 W25X32/64，ALIENTEK 所选择的 W25X16 容量为 16Mb，也就是 2M 字节，同 AT45DB161 是一样大小的。

W25X16 将 2M 的容量分为 32 个块（Block），每个块大小为 64K 字节，每个块又分为 16 个扇区（Sector），每个扇区 4K 个字节。W25X16 的最少擦除单位为一个扇区，也就是每次必须擦除 4K 个字节。这样我们需要给 W25X16 开辟一个至少 4K 的缓存区，这样对 SRAM 要求比较高（相对于 AT45DB161 来说），但是它有价格及供货上的优势。

W25X16 的差些周期为 10000 次，具有 20 年的数据保存期限，支持电压为 2.7~3.6V，W25X16 支持标准的 SPI，还支持双输出的 SPI，最大 SPI 时钟可以到 75Mhz（双输出时相当于 150Mhz），更多的 W25X16 的介绍，请参考 W25X16 的 DATASHEET。

3.17.2 硬件设计

本节实验功能简介：开机的时候先检测 W25X16 是否存在，然后在主循环里面用 1 个按键用来执行写入 W25X16 的操作，另外一个按键用来执行读出操作，在 TFTLCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）KEY0 和 KEY2。

4）TFTLCD 液晶模块。

5）W25X16。

前面 4 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里只介绍 W25X16 与 STM32 的连接，板上的 W25X16 是直接连在 STM32F103RBT6 上的，连接关系如下图：

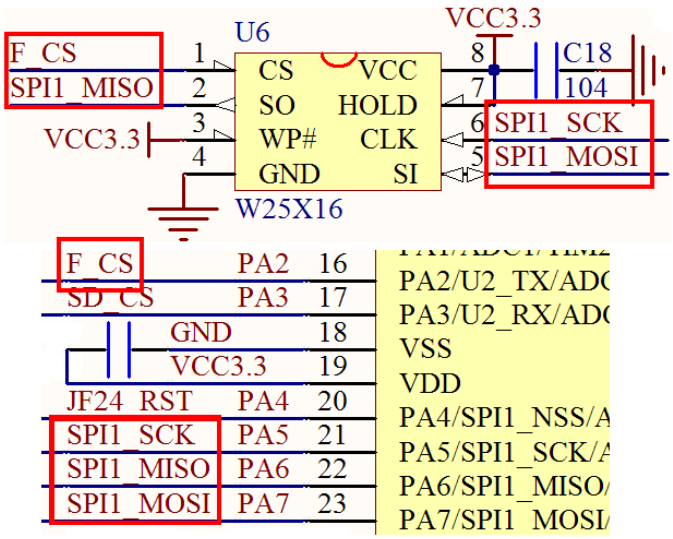


图 3.17.2.1 STM32F103RBT6 与 W25X16 连接电路图

3.17.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 FLASH 的文件夹和 SPI 的文件夹。然后新建一个 flash.c 和 flash.h 的文件保存在 FLASH 文件夹下，新建 spi.c 和 spi.h 的文件，保存在 SPI 文件夹下，并将这两个文件夹加入头文件包含路径。

打开 spi.c 文件，输入如下代码：

```
#include "spi.h"
//Mini STM32 开发板
//SPI  驱动  V1.2
//正点原子@ALIENTEK
//SPI 口初始化
//这里针是对 SPI1 的初始化
void SPIx_Init(void)
{
    RCC->APB2ENR|=1<<2;          //PORTA 时钟使能
    RCC->APB2ENR|=1<<12;         //SPI1 时钟使能
    //这里只针对 SPI 口初始化
    GPIOA->CRL&=0X000FFFFF;
    GPIOA->CRL|=0XBBB00000;//PA5.6.7 复用
    GPIOA->ODR|=0X7<<5;      //PA5.6.7 上拉
    SPI1->CR1|=0<<10;//全双工模式
    SPI1->CR1|=1<<9; //软件 nss 管理
    SPI1->CR1|=1<<8;
```

```
    SPI1->CR1|=1<<2; //SPI 主机
    SPI1->CR1|=0<<11;//8bit 数据格式
    SPI1->CR1|=1<<1; //空闲模式下 SCK 为 1 CPOL=1
    SPI1->CR1|=1<<0; //数据采样从第二个时间边沿开始,CPHA=1
    SPI1->CR1|=7<<3; //Fsck=Fcpu/256
    SPI1->CR1|=0<<7; //MSBfirst
    SPI1->CR1|=1<<6; //SPI 设备使能
    SPIx_ReadWriteByte(0xff);//启动传输
}
//SPI 速度设置函数
//SpeedSet:
//SPI_SPEED_2     2 分频     (SPI 36M@sys 72M)
//SPI_SPEED_8     8 分频     (SPI 9M@sys 72M)
//SPI_SPEED_16    16 分频    (SPI 4.5M@sys 72M)
//SPI_SPEED_256 256 分频  (SPI 281.25K@sys 72M)
void SPIx_SetSpeed(u8 SpeedSet)
{
    SPI1->CR1&=0XFFC7;//Fsck=Fcpu/256
    if(SpeedSet==SPI_SPEED_2)//二分频
    {
        SPI1->CR1|=0<<3;//Fsck=Fpclk/2=36Mhz
    }else if(SpeedSet==SPI_SPEED_8)//八分频
    {
        SPI1->CR1|=2<<3;//Fsck=Fpclk/8=9Mhz
    }else if(SpeedSet==SPI_SPEED_16)//十六分频
    {
        SPI1->CR1|=3<<3;//Fsck=Fpclk/16=4.5Mhz
    }else                           //256 分频
    {
        SPI1->CR1|=7<<3; //Fsck=Fpclk/256=281.25Khz  低速模式
    }
    SPI1->CR1|=1<<6; //SPI 设备使能
}
//SPIx 读写一个字节
//TxData:要写入的字节
//返回值:读取到的字节
u8 SPIx_ReadWriteByte(u8 TxData)
{
    u8 retry=0;
    while((SPI1->SR&1<<1)==0)//等待发送区空
    {
        retry++;
        if(retry>200)return 0;
```

```
    }
    SPI1->DR=TxData;                //发送一个 byte
    retry=0;
    while((SPI1->SR&1<<0)==0) //等待接收完一个 byte
    {
        retry++;
        if(retry>200)return 0;
    }
    return SPI1->DR;                //返回收到的数据
}
```

此部分代码主要初始化 SPI，这里我们选择的是 SPI1，所以在 SPIx_Init 函数里面，其相关的操作都是针对 SPI1 的，其初始化步骤和我们上面介绍的一样。在初始化之后，我们就可以开始使用 SPI1 了，在 SPIx_Init 函数里面，把 SPI1 的波特率设置成了最低（281.25Khz）。在外部函数里面，我们通过 SPIx_SetSpeed 来设置 SPI1 的速度，而我们的数据发送和接收则是通过 SPIx_ReadWriteByte 函数来实现的。、

保存 spi.c，并把该文件加入 RDWARE 组下面，然后我们打开 spi.h 在里面输入如下代码：

```
#ifndef __SPI_H
#define __SPI_H
#include "sys.h"
//Mini STM32 开发板
//SPI  驱动  V1.1
//正点原子@ALIENTEK
// SPI 总线速度设置
#define SPI_SPEED_2     0
#define SPI_SPEED_8     1
#define SPI_SPEED_16    2
#define SPI_SPEED_256 3
void SPIx_Init(void);                   //初始化 SPI 口
void SPIx_SetSpeed(u8 SpeedSet); //设置 SPI 速度
u8 SPIx_ReadWriteByte(u8 TxData);//SPI 总线读写一个字节
#endif
```

此部分代码我们就不多介绍了，保存 spi.h，然后我们打开 flash.c，在里面输入如下代码：

```
#include "flash.h"
#include "spi.h"
#include "delay.h"
//Mini STM32 开发板
//W25X16  驱动函数
//正点原子@ALIENTEK
//V1.0
//4Kbytes 为一个 Sector
//16 个扇区为 1 个 Block
//W25X16
```

```
//容量为 2M 字节,共有 32 个 Block,512 个 Sector
//初始化 SPI FLASH 的 IO 口
void SPI_Flash_Init(void)
{
    RCC->APB2ENR|=1<<2;          //PORTA 时钟使能
    //这里
    GPIOA->CRL&=0XFFF000FF;
    GPIOA->CRL|=0X00033300;//PA2.3.4 推挽
    GPIOA->ODR|=0X7<<2;      //PA2.3.4 上拉
    SPIx_Init();             //初始化 SPI
}

//读取 SPI_FLASH 的状态寄存器
//BIT7  6   5   4   3   2   1   0
//SPR     RV   TB BP2 BP1 BP0 WEL BUSY
//SPR:默认 0,状态寄存器保护位,配合 WP 使用
//TB,BP2,BP1,BP0:FLASH 区域写保护设置
//WEL:写使能锁定
//BUSY:忙标记位(1,忙;0,空闲)
//默认:0x00
u8 SPI_Flash_ReadSR(void)
{
    u8 byte=0;
    SPI_FLASH_CS=0;                          //使能器件
    SPIx_ReadWriteByte(W25X_ReadStatusReg);    //发送读取状态寄存器命令
    byte=SPIx_ReadWriteByte(0Xff);             //读取一个字节
    SPI_FLASH_CS=1;                          //取消片选
    return byte;
}
//写 SPI_FLASH 状态寄存器
//只有 SPR,TB,BP2,BP1,BP0(bit 7,5,4,3,2)可以写!!!
void SPI_FLASH_Write_SR(u8 sr)
{
    SPI_FLASH_CS=0;                          //使能器件
    SPIx_ReadWriteByte(W25X_WriteStatusReg);   //发送写取状态寄存器命令
    SPIx_ReadWriteByte(sr);                  //写入一个字节
    SPI_FLASH_CS=1;                          //取消片选
}
//SPI_FLASH 写使能
//将 WEL 置位
void SPI_FLASH_Write_Enable(void)
{
    SPI_FLASH_CS=0;                          //使能器件
```

```
    SPIx_ReadWriteByte(W25X_WriteEnable);       //发送写使能
    SPI_FLASH_CS=1;                             //取消片选
}
//SPI_FLASH 写禁止
//将 WEL 清零
void SPI_FLASH_Write_Disable(void)
{
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_WriteDisable);      //发送写禁止指令
    SPI_FLASH_CS=1;                             //取消片选
}
//读取芯片 ID W25X16 的 ID:0XEF14
u16 SPI_Flash_ReadID(void)
{
    u16 Temp = 0;
    SPI_FLASH_CS=0;
    SPIx_ReadWriteByte(0x90);//发送读取 ID 命令
    SPIx_ReadWriteByte(0x00);
    SPIx_ReadWriteByte(0x00);
    SPIx_ReadWriteByte(0x00);
    Temp|=SPIx_ReadWriteByte(0xFF)<<8;
    Temp|=SPIx_ReadWriteByte(0xFF);
    SPI_FLASH_CS=1;
    return Temp;
}
//读取 SPI FLASH
//在指定地址开始读取指定长度的数据
//pBuffer:数据存储区
//ReadAddr:开始读取的地址(24bit)
//NumByteToRead:要读取的字节数(最大 65535)
void SPI_Flash_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead)
{
    u16 i;
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_ReadData);          //发送读取命令
    SPIx_ReadWriteByte((u8)((ReadAddr)>>16));   //发送 24bit 地址
    SPIx_ReadWriteByte((u8)((ReadAddr)>>8));
    SPIx_ReadWriteByte((u8)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPIx_ReadWriteByte(0XFF);    //循环读数
    }
    SPI_FLASH_CS=1;                             //取消片选
```

```
}
//SPI 在一页(0~65535)内写入少于 256 个字节的数据
//在指定地址开始写入最大 256 字节的数据
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大 256),该数不应该超过该页的剩余字节数!!!
void SPI_Flash_Write_Page(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u16 i;
    SPI_FLASH_Write_Enable();                  //SET WEL
    SPI_FLASH_CS=0;                            //使能器件
    SPIx_ReadWriteByte(W25X_PageProgram);      //发送写页命令
    SPIx_ReadWriteByte((u8)((WriteAddr)>>16)); //发送 24bit 地址
    SPIx_ReadWriteByte((u8)((WriteAddr)>>8));
    SPIx_ReadWriteByte((u8)WriteAddr);
    for(i=0;i<NumByteToWrite;i++)SPIx_ReadWriteByte(pBuffer[i]);//循环写数
    SPI_FLASH_CS=1;                            //取消片选
    SPI_Flash_Wait_Busy();                     //等待写入结束
}
//无检验写 SPI FLASH
//必须确保所写的地址范围内的数据全部为 0XFF,否则在非 0XFF 处写入的数据将失败!
//具有自动换页功能
//在指定地址开始写入指定长度的数据,但是要确保地址不越界!
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大 65535)
//CHECK OK
void SPI_Flash_Write_NoCheck(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u16 pageremain;
    pageremain=256-WriteAddr%256; //单页剩余的字节数
    if(NumByteToWrite<=pageremain)pageremain=NumByteToWrite;//不大于 256 个字节
    while(1)
    {
        SPI_Flash_Write_Page(pBuffer,WriteAddr,pageremain);
        if(NumByteToWrite==pageremain)break;//写入结束了
        else //NumByteToWrite>pageremain
        {
            pBuffer+=pageremain;
            WriteAddr+=pageremain;

            NumByteToWrite-=pageremain;          //减去已经写入了的字节数
            if(NumByteToWrite>256)pageremain=256; //一次可以写入 256 个字节
```

```
                else pageremain=NumByteToWrite;         //不够 256 个字节了
            }
        };
    }
    //写 SPI FLASH
    //在指定地址开始写入指定长度的数据
    //该函数带擦除操作!
    //pBuffer:数据存储区
    //WriteAddr:开始写入的地址(24bit)
    //NumByteToWrite:要写入的字节数(最大 65535)
    u8 SPI_FLASH_BUF[4096];
    void SPI_Flash_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
    {
        u32 secpos;
        u16 secoff;
        u16 secremain;
        u16 i;
        secpos=WriteAddr/4096;//扇区地址  0~511 for w25x16
        secoff=WriteAddr%4096;//在扇区内的偏移
        secremain=4096-secoff;//扇区剩余空间大小
        if(NumByteToWrite<=secremain)secremain=NumByteToWrite;//不大于 4096 个字节
        while(1)
        {
            SPI_Flash_Read(SPI_FLASH_BUF,secpos*4096,4096);//读出整个扇区的内容
            for(i=0;i<secremain;i++)//校验数据
            {
                if(SPI_FLASH_BUF[secoff+i]!=0XFF)break;//需要擦除
            }
            if(i<secremain)//需要擦除
            {
                SPI_Flash_Erase_Sector(secpos);//擦除这个扇区
                for(i=0;i<secremain;i++)            //复制
                {
                    SPI_FLASH_BUF[i+secoff]=pBuffer[i];
                }
                SPI_Flash_Write_NoCheck(SPI_FLASH_BUF,secpos*4096,4096);//写入整个扇
区

            }else  SPI_Flash_Write_NoCheck(pBuffer,WriteAddr,secremain);//写已经擦除了的,
直接写入扇区剩余区间.
            if(NumByteToWrite==secremain)break;//写入结束了
            else//写入未结束
            {
```

```
            secpos++;//扇区地址增 1
            secoff=0;//偏移位置为 0

            pBuffer+=secremain;  //指针偏移
            WriteAddr+=secremain;//写地址偏移
            NumByteToWrite-=secremain;                  //字节数递减
            if(NumByteToWrite>4096)secremain=4096;//下一个扇区还是写不完
            else secremain=NumByteToWrite;              //下一个扇区可以写完了
        }
    };
}
//擦除整个芯片
//整片擦除时间:
//W25X16:25s
//W25X32:40s
//W25X64:40s
//等待时间超长...
void SPI_Flash_Erase_Chip(void)
{
    SPI_FLASH_Write_Enable();                   //SET WEL
    SPI_Flash_Wait_Busy();
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_ChipErase);         //发送片擦除命令
    SPI_FLASH_CS=1;                             //取消片选
    SPI_Flash_Wait_Busy();                      //等待芯片擦除结束
}
//擦除一个扇区
//Dst_Addr:扇区地址  0~511 for w25x16
//擦除一个山区的最少时间:150ms
void SPI_Flash_Erase_Sector(u32 Dst_Addr)
{
    Dst_Addr*=4096;
    SPI_FLASH_Write_Enable();                   //SET WEL
    SPI_Flash_Wait_Busy();
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_SectorErase);       //发送扇区擦除指令
    SPIx_ReadWriteByte((u8)((Dst_Addr)>>16));  //发送 24bit 地址
    SPIx_ReadWriteByte((u8)((Dst_Addr)>>8));
    SPIx_ReadWriteByte((u8)Dst_Addr);
    SPI_FLASH_CS=1;                             //取消片选
    SPI_Flash_Wait_Busy();                      //等待擦除完成
}
//等待空闲
```

```
void SPI_Flash_Wait_Busy(void)
{
    while ((SPI_Flash_ReadSR()&0x01)==0x01);    // 等待 BUSY 位清空
}
//进入掉电模式
void SPI_Flash_PowerDown(void)
{
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_PowerDown);         //发送掉电命令
    SPI_FLASH_CS=1;                             //取消片选
    delay_us(3);                                //等待 TPD
}
//唤醒
void SPI_Flash_WAKEUP(void)
{
    SPI_FLASH_CS=0;                             //使能器件
    SPIx_ReadWriteByte(W25X_ReleasePowerDown);  // send W25X_PowerDown command 0xAB
    SPI_FLASH_CS=1;                             //取消片选
    delay_us(3);                                //等待 TRES1
}
```

此部分代码里面一个最关键的函数就是 void SPI_Flash_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)，该函数可以在 W25X16 的任意地址开始写入任意长度（必须不超过 W25X16 的容量）的数据。我们这里简单介绍一下思路：先获得首地址（WriteAddr）所在的扇区，并计算在扇区内的偏移，然后判断要写入的数据长度是否超过本扇区所剩下的长度，如果不超过，再先看看是否要删除，如果不要，则直接写入数据即可，如果要则读出整个扇区，在偏移处开始写入指定长度的数据，然后擦除这个扇区，再一次性写入。当所需要写入的数据长度超过一个扇区的长度的时候，我们先按照前面的步骤把扇区剩余部分写完，再在新扇区内执行同样的操作，如此循环，直到写入结束。

其他的代码就比较简单了，我们这里不介绍了。保存 falsh.c，然后加入到 HARDWARE 组下面，再打开 flahs.h，在该文件里面输入如下代码：

```
#ifndef __FLASH_H
#define __FLASH_H
#include "sys.h"
//Mini STM32 开发板
//W25X16  驱动函数
//正点原子@ALIENTEK
//V1.0
#define   SPI_FLASH_CS PAout(2)   //选中 FLASH
////////////////////////////////////////////////////////////////////////////////
//W25X16 读写
#define FLASH_ID 0XEF14
//指令表
```

```
#define W25X_WriteEnable        0x06
#define W25X_WriteDisable       0x04
#define W25X_ReadStatusReg      0x05
#define W25X_WriteStatusReg     0x01
#define W25X_ReadData           0x03
#define W25X_FastReadData       0x0B
#define W25X_FastReadDual       0x3B
#define W25X_PageProgram        0x02
#define W25X_BlockErase         0xD8
#define W25X_SectorErase        0x20
#define W25X_ChipErase          0xC7
#define W25X_PowerDown          0xB9
#define W25X_ReleasePowerDown   0xAB
#define W25X_DeviceID           0xAB
#define W25X_ManufactDeviceID   0x90
#define W25X_JedecDeviceID      0x9F
void SPI_Flash_Init(void);
u16    SPI_Flash_ReadID(void);          //读取 FLASH ID
u8     SPI_Flash_ReadSR(void);            //读取状态寄存器
void SPI_FLASH_Write_SR(u8 sr);     //写状态寄存器
void SPI_FLASH_Write_Enable(void);   //写使能
void SPI_FLASH_Write_Disable(void);      //写保护
void SPI_Flash_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead);     //读取 flash
void SPI_Flash_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite);//写入 flash
void SPI_Flash_Erase_Chip(void);                //整片擦除
void SPI_Flash_Erase_Sector(u32 Dst_Addr);//扇区擦除
void SPI_Flash_Wait_Busy(void);                 //等待空闲
void SPI_Flash_PowerDown(void);                 //进入掉电模式
void SPI_Flash_WAKEUP(void);                    //唤醒
#endif
```

这里面就定义了一些与 W25X16 操作相关的命令，这些命令在 W25X16 的数据手册上都有详细的介绍，感兴趣的大家可以参考该数据手册，其他的就没啥好说的了。保存此部分代码。

最后，我们在 test.c 里面，修改 main 函数如下：

```
//要写入到 W25X16 的字符串数组
const u8 TEXT_Buffer[]={"MiniSTM32 SPI TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);      //延时初始化
```

```
uart_init(72,9600); //串口 1 初始化
LED_Init();                 //LED 初始化
KEY_Init();                 //按键初始化
LCD_Init();                 //TFTLCD 液晶初始化
SPI_Flash_Init();     //SPI FLASH  初始化
POINT_COLOR=RED;//设置字体为蓝色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"SPI TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2011/1/1");
while(SPI_Flash_ReadID()!=FLASH_ID)//检测不到 W25X16
{
    i=SPI_Flash_ReadID();
    printf("ID:%d",i);
    LCD_ShowString(60,130,"W25X16 Check Failed!");
    delay_ms(500);
    LCD_ShowString(60,130,"      Please Check!       ");
    delay_ms(500);
    LED0=!LED0;//DS0 闪烁
}
LCD_ShowString(60,130,"W25X16 Ready!");
//显示提示信息
LCD_ShowString(60,150,"KEY0:Write KEY2:Read");
POINT_COLOR=BLUE;//设置字体为蓝色
while(1)
{
    key=KEY_Scan();
    if(key==1)//KEY0 按下,写入 SPI FLASH
    {
        LCD_Fill(0,170,239,319,WHITE);//清除半屏
        LCD_ShowString(60,170,"Start Write W25X16....");
        SPI_Flash_Write((u8*)TEXT_Buffer,1000,SIZE);//从 1000 字节处开始,写入
SIZE 长度的数据
        LCD_ShowString(60,170,"W25X16 Write Finished!");//提示传送完成
    }
    if(key==3)//KEY1 按下,读取写入的字符传字符串并显示
    {
        LCD_ShowString(60,170,"Start Read W25X16.... ");
        SPI_Flash_Read(datatemp,1000,SIZE);//从 1000 地址处开始,读出 SIZE 个字节
        LCD_ShowString(60,170,"The Data Readed Is:   ");//提示传送完成
        LCD_ShowString(60,190,datatemp);//显示读到的字符串
    }
    i++;
```

```
            delay_ms(1);
            if(i==200)
            {
                LED0=!LED0;//提示系统正在运行
                i=0;
            }
        }
    }
```

这部分代码和 IIC 实验那部分代码大同小异，我们就不多说了，实现的功能就和 IIC 差不多，不过此次写入和读出的是 SPI FLASH，而不是 EEPROM。

3.17.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

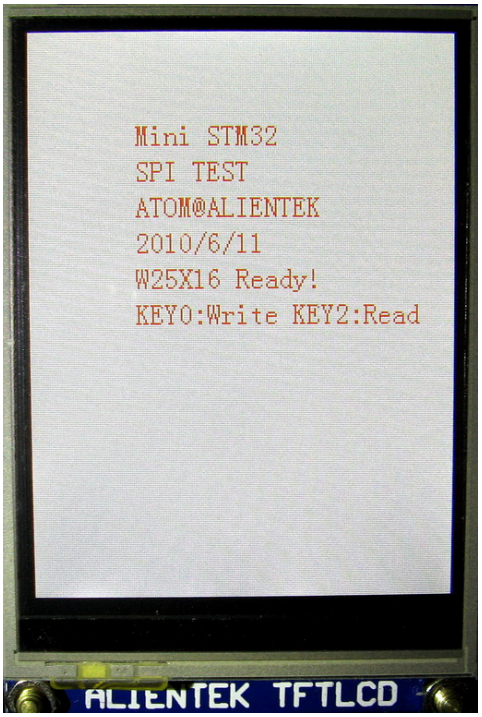


图 3.17.4.1 程序运行时界面

伴随 DS0 的不停闪烁，提示程序在运行。我们先按下 KEY0，可以看到如图 13.17.4.2 所示的内容，证明数据已经被写入到 W25X16 了。

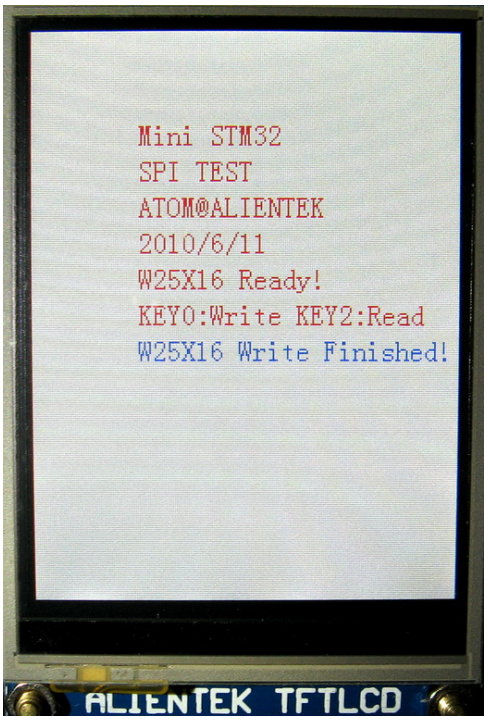


图 3.17.4.2 数据被成功写入到 W25X16

接着我们按 KEY2，可以看我们刚刚写入的数据被显示出来了，如下图所示：

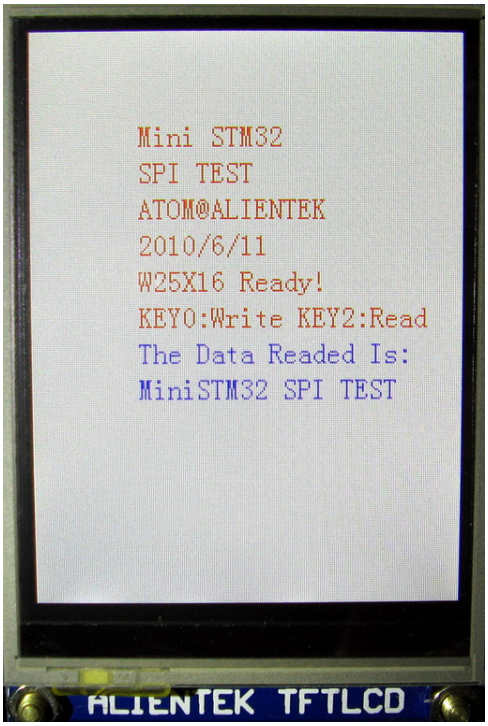


图 3.17.4.3 显示写入到 W25X16 的内容

程序在开机的时候会检测 W25X16 是否存在，如果不存在则会在 TFTLCD 模块上显示错误信息，同时 DS0 慢闪。大家可以通过跳线帽把 PA5 和 PA6 短接就可以看到报错了。

3.18 触摸屏实验

ALIENTKE MiniSTM3 开发板本身并没有触摸屏控制器，但是它支持触摸屏，可以通过外接带触摸屏的 LCD 模块（比如 ALIENTEK TFTLCD 模块），来实现触摸屏控制。这一节我们将向大家介绍 STM32 控制 ALIENTKE TFTLCD 模块。本节将利用软件模拟 SPI 来实现对 TFTLCD 模块的触摸屏控制，最终实现一个手写的功能。本节分为如下几个部分：

3.18.1 触摸屏简介
3.18.2 硬件设计
3.18.3 软件设计
3.18.4 下载与测试

3.18.1 触摸屏简介

我们一般液晶所用的触摸屏，最多的就是电阻式触摸屏了（多点触摸属于电容式触摸屏，比如 M8，IPhone 等支持多点触摸的手机所用的屏就是电容式的触摸屏），ALIENTEK TFTLCD 自带的触摸屏属于电阻式触摸屏，下面简单介绍下电阻式触摸屏的原理。

电阻式触摸屏利用压力感应进行控制。电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，它以一层玻璃或硬塑料平板作为基层，表面涂有一层透明氧化金属（透明的导电电阻）导电层，上面再盖有一层外表面硬化处理、光滑防擦的塑料层、它的内表面也涂有一层涂层、在他们之间有许多细小的（小于 1/1000 英寸）的透明隔离点把两层导电层隔开绝缘。 当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，电阻发生变化，在 X 和 Y 两个方向上产生信号，然后送触摸屏控制器。控制器侦测到这一接触并计算出（X，Y）的位置，再根据获得的位置模拟鼠标的方式运作。这就是电阻技术触摸屏的最基本的原理。

电阻屏的特点有：

1）是一种对外界完全隔离的工作环境，不怕灰尘、水汽和油污。

2）可以用任何物体来触摸，可以用来写字画画，这是它们比较大的优势。

3）电阻触摸屏的精度只取决于 A/D 转换的精度，因此都能轻松达到 4096*4096。

从以上介绍可知，触摸屏都需要一个 AD 转换器， 一般来说是需要一个控制器的。ALIENTEK TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7843、ADS7846、TSC2046、XPT2046 和 AK4182 等。这几款芯片的驱动基本上是一样的，也就是你只要写出了 ADS7843 的驱动，这个驱动对其他几个芯片也是有效的。而且封装也有一样的，完全 PINTOPIN。所以在替换起来，很方便。

ALIENTEK TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置， 除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。 在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16,QFN-16(0.75mm 厚度)和 VFBGA－48。工作温度范围为-40℃～+85℃。

该芯片完全是兼容 ADS7843 和 ADS7846 的，关于这个芯片的详细使用，可以参考这两个芯片的 datasheet。

3.18.2 硬件设计

本节实验功能简介：开机的时候先通过 24C02 的数据判断触摸屏是否已经校准过，如果没有校准，则执行校准程序，校准过后再进入手写程序。如果已经校准了，就直接进入手写程序，此时可以通过按动屏幕来实现手写输入。屏幕上会有一个清空的操作区域（RST），点击这个地方就会将输入全部清除，恢复白板状态。程序会设置一个强制校准，就是通过按 KEY0 来实现，只要按下 KEY0 就会进入强制校准程序，这个强制校准程序是必须的。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）KEY0。

4）TFTLCD 液晶模块。

5）24C01。

所有这些资源与 STM32 的连接图，在前面都已经介绍了，这里我们只针对 TFTLCD 模块与 STM32 的连接端口再说明一下，TFTLCD 模块的触摸屏总共有 5 跟线与 STM32 连接，连接电路图如下：

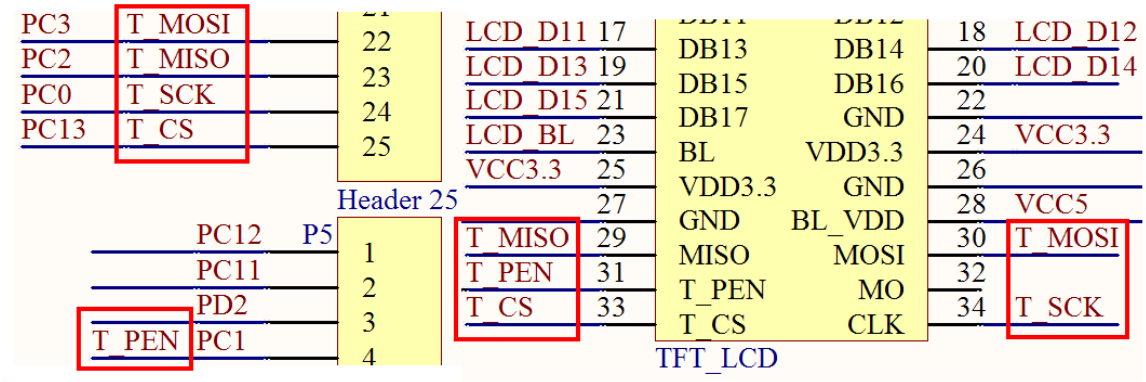


图 3.18.2.1 触摸屏与 STM32F103RBT6 连接图

3.18.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 TOUCH 文件夹。然后新建一个 touch.c 和 touch.h 的文件保存在 TOUCH 文件夹下，并将这个文件夹加入头文件包含路径。

打开 touch.c 文件，输入如下代码：

```
#include "touch.h"
#include "lcd.h"
#include "delay.h"
#include "stdlib.h"
#include "math.h"
#include "24cxx.h"
//Mini STM32 开发板
//ADS7843/7846/UH7843/7846/XPT2046/TSC2046 驱动函数
//正点原子@ALIENTEK
//V1.3
Pen_Holder Pen_Point;//定义笔实体
```

```
//SPI 写数据
//向 7843 写入 1byte 数据
void ADS_Write_Byte(u8 num)
{
    u8 count=0;
    for(count=0;count<8;count++)
    {
        if(num&0x80)TDIN=1;
        else TDIN=0;
        num<<=1;
        TCLK=0;//上升沿有效
        TCLK=1;
    }
}
//SPI 读数据
//从 7846/7843/XPT2046/UH7843/UH7846 读取 adc 值
u16 ADS_Read_AD(u8 CMD)
{
    u8 count=0;
    u16 Num=0;
    TCLK=0;//先拉低时钟
    TCS=0; //选中 ADS7843
    ADS_Write_Byte(CMD);//发送命令字
    delay_us(6);//ADS7846 的转换时间最长为 6us
    TCLK=1;//给 1 个时钟，清除 BUSY
    TCLK=0;
    for(count=0;count<16;count++)
    {
        Num<<=1;
        TCLK=0;//下降沿有效
        TCLK=1;
        if(DOUT)Num++;
    }
    Num>>=4;   //只有高 12 位有效.
    TCS=1;//释放 ADS7843
    return(Num);
}
//读取一个坐标值
//连续读取 READ_TIMES 次数据,对这些数据升序排列,
//然后去掉最低和最高 LOST_VAL 个数,取平均值
#define READ_TIMES 15 //读取次数
#define LOST_VAL 5    //丢弃值
u16 ADS_Read_XY(u8 xy)
```

```
{
    u16 i, j;
    u16 buf[READ_TIMES];
    u16 sum=0;
    u16 temp;
    for(i=0;i<READ_TIMES;i++)
    {
        buf[i]=ADS_Read_AD(xy);
    }
    for(i=0;i<READ_TIMES-1; i++)//排序
    {
        for(j=i+1;j<READ_TIMES;j++)
        {
            if(buf[i]>buf[j])//升序排列
            {
                temp=buf[i];
                buf[i]=buf[j];
                buf[j]=temp;
            }
        }
    }
    sum=0;
    for(i=LOST_VAL;i<READ_TIMES-LOST_VAL;i++)sum+=buf[i];
    temp=sum/(READ_TIMES-2*LOST_VAL);
    return temp;
}
//带滤波的坐标读取
//最小值不能少于100.
u8 Read_ADS(u16 *x,u16 *y)
{
    u16 xtemp,ytemp;
    xtemp=ADS_Read_XY(CMD_RDX);
    ytemp=ADS_Read_XY(CMD_RDY);

    if(xtemp<100||ytemp<100)return 0;//读数失败
    *x=xtemp;
    *y=ytemp;
    return 1;//读数成功
}
//2次读取ADS7846,连续读取2次有效的AD值,且这两次的偏差不能超过
//50,满足条件,则认为读数正确,否则读数错误.
//该函数能大大提高准确度
#define ERR_RANGE 50 //误差范围
```

```
u8 Read_ADS2(u16 *x,u16 *y)
{
    u16 x1,y1;
    u16 x2,y2;
    u8 flag;
    flag=Read_ADS(&x1,&y1);
    if(flag==0)return(0);
    flag=Read_ADS(&x2,&y2);
    if(flag==0)return(0);
    if(((x2<=x1&&x1<x2+ERR_RANGE)||(x1<=x2&&x2<x1+ERR_RANGE))//前后两次采样在
+-50 内
    &&((y2<=y1&&y1<y2+ERR_RANGE)||(y1<=y2&&y2<y1+ERR_RANGE)))
    {
        *x=(x1+x2)/2;
        *y=(y1+y2)/2;
        return 1;
    }else return 0;
}
//读取一次坐标值
//仅仅读取一次,知道 PEN 松开才返回!
u8 Read_TP_Once(void)
{
    u8 t=0;
    Pen_Int_Set(0);//关闭中断
    Pen_Point.Key_Sta=Key_Up;
    Read_ADS2(&Pen_Point.X,&Pen_Point.Y);
    while(PEN==0&&t<=250)
    {
        t++;
        delay_ms(10);
    };
    Pen_Int_Set(1);//开启中断
    if(t>=250)return 0;//按下 2.5s 认为无效
    else return 1;
}

//////////////////////////////////////////////////
//与 LCD 部分有关的函数
//画一个触摸点
//用来校准用的
void Drow_Touch_Point(u8 x,u16 y)
{
    LCD_DrawLine(x-12,y,x+13,y);//横线
```

```
    LCD_DrawLine(x,y-12,x,y+13);//竖线
    LCD_DrawPoint(x+1,y+1);
    LCD_DrawPoint(x-1,y+1);
    LCD_DrawPoint(x+1,y-1);
    LCD_DrawPoint(x-1,y-1);
    Draw_Circle(x,y,6);//画中心圈
}
//画一个大点
//2*2 的点
void Draw_Big_Point(u8 x,u16 y)
{
    LCD_DrawPoint(x,y);//中心点
    LCD_DrawPoint(x+1,y);
    LCD_DrawPoint(x,y+1);
    LCD_DrawPoint(x+1,y+1);
}
//////////////////////////////////////////////////
//转换结果
//根据触摸屏的校准参数来决定转换后的结果,保存在 X0,Y0 中
void Convert_Pos(void)
{
    if(Read_ADS2(&Pen_Point.X,&Pen_Point.Y))
    {
        Pen_Point.X0=Pen_Point.xfac*Pen_Point.X+Pen_Point.xoff;
        Pen_Point.Y0=Pen_Point.yfac*Pen_Point.Y+Pen_Point.yoff;
    }
}
//中断,检测到 PEN 脚的一个下降沿.
//置位 Pen_Point.Key_Sta 为按下状态
//中断线 0 线上的中断检测
void EXTI1_IRQHandler(void)
{
    Pen_Point.Key_Sta=Key_Down;//按键按下
    EXTI->PR=1<<1;  //清除 LINE1 上的中断标志位
}
//PEN 中断设置
void Pen_Int_Set(u8 en)
{
    if(en)EXTI->IMR|=1<<1;   //开启 line1 上的中断
    else EXTI->IMR&=~(1<<1); //关闭 line1 上的中断
}
//////////////////////////////////////////////////////////////////////////////
//此部分涉及到使用外部 EEPROM,如果没有外部 EEPROM,屏蔽此部分即可
```

```
#ifdef ADJ_SAVE_ENABLE
// 保存在 EEPROM 里面的地址区间基址，占用 13 个字节(RANGE:SAVE_ADDR_BASE~SAVE_ADDR_BASE+12)
#define SAVE_ADDR_BASE 40
//保存校准参数
void Save_Adjdata(void)
{
    s32 temp;
    //保存校正结果!
    temp=Pen_Point.xfac*100000000;//保存 x 校正因素
    AT24CXX_WriteLenByte(SAVE_ADDR_BASE,temp,4);
    temp=Pen_Point.yfac*100000000;//保存 y 校正因素
    AT24CXX_WriteLenByte(SAVE_ADDR_BASE+4,temp,4);
    //保存 x 偏移量
    AT24CXX_WriteLenByte(SAVE_ADDR_BASE+8,Pen_Point.xoff,2);
    //保存 y 偏移量
    AT24CXX_WriteLenByte(SAVE_ADDR_BASE+10,Pen_Point.yoff,2);

    temp=AT24CXX_ReadOneByte(SAVE_ADDR_BASE+12);
    temp&=0XF0;
    temp|=0X0A;//标记校准过了
    AT24CXX_WriteOneByte(SAVE_ADDR_BASE+12,temp);
}
//得到保存在 EEPROM 里面的校准值
//返回值：1，成功获取数据
//        0，获取失败，要重新校准
u8 Get_Adjdata(void)
{
    s32 tempfac;
    tempfac=AT24CXX_ReadOneByte(52);//第五十二字节的第四位用来标记是否校准过!

    if((tempfac&0X0F)==0X0A)//触摸屏已经校准过了
    {
        tempfac=AT24CXX_ReadLenByte(40,4);
        Pen_Point.xfac=(float)tempfac/100000000;//得到 x 校准参数
        tempfac=AT24CXX_ReadLenByte(44,4);
        Pen_Point.yfac=(float)tempfac/100000000;//得到 y 校准参数
        //得到 x 偏移量
        tempfac=AT24CXX_ReadLenByte(48,2);
        Pen_Point.xoff=tempfac;
        //得到 y 偏移量
        tempfac=AT24CXX_ReadLenByte(50,2);
        Pen_Point.yoff=tempfac;
```

```
        return 1;
    }
    return 0;
}
#endif
//触摸屏校准代码
//得到四个校准参数
void Touch_Adjust(void)
{
    u16 pos_temp[4][2];//坐标缓存值
    u8  cnt=0;
    u16 d1,d2;
    u32 tem1,tem2;
    float fac;
    cnt=0;
    POINT_COLOR=BLUE;
    BACK_COLOR =WHITE;
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=RED;//红色
    LCD_Clear(WHITE);//清屏
    Drow_Touch_Point(20,20);//画点 1
    Pen_Point.Key_Sta=Key_Up;//消除触发信号
    Pen_Point.xfac=0;//xfac 用来标记是否校准过,所以校准之前必须清掉!以免错误
    while(1)
    {
        if(Pen_Point.Key_Sta==Key_Down)//按键按下了
        {
            if(Read_TP_Once())//得到单次按键值
            {
                pos_temp[cnt][0]=Pen_Point.X;
                pos_temp[cnt][1]=Pen_Point.Y;
                cnt++;
            }
            switch(cnt)
            {
                case 1:
                    LCD_Clear(WHITE);//清屏
                    Drow_Touch_Point(220,20);//画点 2
                    break;
                case 2:
                    LCD_Clear(WHITE);//清屏
                    Drow_Touch_Point(20,300);//画点 3
                    break;
```

```
case 3:
    LCD_Clear(WHITE);//清屏
    Drow_Touch_Point(220,300);//画点 4
    break;
case 4:  //全部四个点已经得到
    //对边相等
    tem1=abs(pos_temp[0][0]-pos_temp[1][0]);//x1-x2
    tem2=abs(pos_temp[0][1]-pos_temp[1][1]);//y1-y2
    tem1*=tem1;
    tem2*=tem2;
    d1=sqrt(tem1+tem2);//得到 1,2 的距离

    tem1=abs(pos_temp[2][0]-pos_temp[3][0]);//x3-x4
    tem2=abs(pos_temp[2][1]-pos_temp[3][1]);//y3-y4
    tem1*=tem1;
    tem2*=tem2;
    d2=sqrt(tem1+tem2);//得到 3,4 的距离
    fac=(float)d1/d2;
    if(fac<0.95||fac>1.05||d1==0||d2==0)//不合格
    {
        cnt=0;
        LCD_Clear(WHITE);//清屏
        Drow_Touch_Point(20,20);
        continue;
    }
    tem1=abs(pos_temp[0][0]-pos_temp[2][0]);//x1-x3
    tem2=abs(pos_temp[0][1]-pos_temp[2][1]);//y1-y3
    tem1*=tem1;
    tem2*=tem2;
    d1=sqrt(tem1+tem2);//得到 1,3 的距离

    tem1=abs(pos_temp[1][0]-pos_temp[3][0]);//x2-x4
    tem2=abs(pos_temp[1][1]-pos_temp[3][1]);//y2-y4
    tem1*=tem1;
    tem2*=tem2;
    d2=sqrt(tem1+tem2);//得到 2,4 的距离
    fac=(float)d1/d2;
    if(fac<0.95||fac>1.05)//不合格
    {
        cnt=0;
        LCD_Clear(WHITE);//清屏
        Drow_Touch_Point(20,20);
        continue;
```

```
                    }//正确了

                    //对角线相等
                    tem1=abs(pos_temp[1][0]-pos_temp[2][0]);//x1-x3
                    tem2=abs(pos_temp[1][1]-pos_temp[2][1]);//y1-y3
                    tem1*=tem1;
                    tem2*=tem2;
                    d1=sqrt(tem1+tem2);//得到 1,4 的距离

                    tem1=abs(pos_temp[0][0]-pos_temp[3][0]);//x2-x4
                    tem2=abs(pos_temp[0][1]-pos_temp[3][1]);//y2-y4
                    tem1*=tem1;
                    tem2*=tem2;
                    d2=sqrt(tem1+tem2);//得到 2,3 的距离
                    fac=(float)d1/d2;
                    if(fac<0.95||fac>1.05)//不合格
                    {
                        cnt=0;
                        LCD_Clear(WHITE);//清屏
                        Drow_Touch_Point(20,20);
                        continue;
                    }//正确了
                    //计算结果

    Pen_Point.xfac=(float)200/(pos_temp[1][0]-pos_temp[0][0]);//得到 xfac

    Pen_Point.xoff=(240-Pen_Point.xfac*(pos_temp[1][0]+pos_temp[0][0]))/2;// 得 到
xoff



    Pen_Point.yfac=(float)280/(pos_temp[2][1]-pos_temp[0][1]);//得到 yfac

    Pen_Point.yoff=(320-Pen_Point.yfac*(pos_temp[2][1]+pos_temp[0][1]))/2;// 得 到
yoff
                    POINT_COLOR=BLUE;
                    LCD_Clear(WHITE);//清屏
                    LCD_ShowString(35,110,"Touch Screen Adjust OK!");//校正完成
                    delay_ms(1000);
                    LCD_Clear(WHITE);//清屏
                    return;//校正完成
                }
            }
        }
```

```
}
//外部中断初始化函数
void Touch_Init(void)
{
    //注意,时钟使能之后,对 GPIO 的操作才有效
    //所以上拉之前,必须使能时钟.才能实现真正的上拉输出
    RCC->APB2ENR|=1<<4;     //PC 时钟使能
    RCC->APB2ENR|=1<<0;     //开启辅助时钟
    GPIOC->CRL&=0XFFFF0000;//PC0~3
    GPIOC->CRL|=0X00003883;
    GPIOC->CRH&=0XFF0FFFFF;//PC13
    GPIOC->CRH|=0X00300000;//PC13 推挽输出
    GPIOC->ODR|=0X200f;     //PC0~3 13 全部上拉
    Read_ADS(&Pen_Point.X,&Pen_Point.Y);//第一次读取初始化
    MY_NVIC_Init(2,0,EXTI1_IRQChannel,2);
    RCC->APB2ENR|=0x01;     //使能 io 复用时钟
    AFIO->EXTICR[0]|=0X0020; //EXTI13 映射到 PC1
    EXTI->IMR|=1<<1;         //开启 line1 上的中断
    EXTI->EMR|=1<<1;         //不屏蔽 line1 上的事件
    EXTI->FTSR|=1<<1;        //line1 上事件下降沿触发
#ifdef ADJ_SAVE_ENABLE
    AT24CXX_Init();//初始化 24CXX
    if(Get_Adjdata())return;//已经校准
    else                //未校准?
    {
        LCD_Clear(WHITE);//清屏
        Touch_Adjust();  //屏幕校准
        Save_Adjdata();
    }
    Get_Adjdata();
#else
    LCD_Clear(WHITE);//清屏
    Touch_Adjust();  //屏幕校准,带自动保存
#endif
//  printf("Pen_Point.xfac:%f\n",Pen_Point.xfac);
//  printf("Pen_Point.yfac:%f\n",Pen_Point.yfac);
//  printf("Pen_Point.xoff:%d\n",Pen_Point.xoff);
//  printf("Pen_Point.yoff:%d\n",Pen_Point.yoff);
}
```

此部分代码，最核心的应该属触摸屏校准代码了。触摸屏的校准通过 void Touch_Adjust(void)函数实现。在这里，给大家介绍一下我们这里所使用的触摸屏校正原理：我们传统的鼠标是一种相对定位系统，只和前一次鼠标的位置坐标有关。而触摸屏则是一种绝对坐标系统，要选哪就直接点哪，与相对定位系统有着本质的区别。绝对坐标系统的特点是每

一次定位坐标与上一次定位坐标没有关系，每次触摸的数据通过校准转为屏幕上的坐标，不管在什么情况下，触摸屏这套坐标在同一点的输出数据是稳定的。不过由于技术原理的原因，并不能保证同一点触摸每一次采样数据相同，不能保证绝对坐标定位，点不准，这就是触摸屏最怕出现的问题：漂移。对于性能质量好的触摸屏来说，漂移的情况出现并不是很严重。所以很多应用触摸屏的系统启动后，进入应用程序前，先要执行校准程序。 通常应用程序中使用的 LCD 坐标是以像素为单位的。比如说：左上角的坐标是一组非 0 的数值，比如（20，20），而右下角的坐标为（220，300）。这些点的坐标都是以像素为单位的，而从触摸屏中读出的是点的物理坐标，其坐标轴的方向、XY 值的比例因子、偏移量都与 LCD 坐标不同，所以，可以在程序中使用一个函数（我们采用 Convert_Pos 函数）中把物理坐标首先转换为像素坐标，然后再赋给 POS 结构，达到坐标转换的目的。

校正思路：在了解了校正原理之后，我们可以得出下面的一个从屋里坐标到像素坐标的转换关系式：

LCDx=xfac*Px+xoff；

LCDy=yfac*Py+yoff；

其中(LCDx,LCDy)是在 LCD 上的像素坐标，（Px,Py）是从触摸屏读到的物理坐标。xfac，yfac 分别是 X 轴方向和 Y 轴方向的比例因子，而 xoff 和 yoff 则是这两个方向的偏移量。

这样我们只要事先在屏幕上面显示 4 个点（这四个点的坐标是已知的），分别按这四个点就可以从触摸屏读到 4 个物理坐标，这样就可以通过待定系数法求出 xfac、yfac、xoff、yoff 这四个参数。我们保存好这四个参数，在以后的使用中，我们把所有得到的物理坐标都按照这个关系式来计算，得到的就是准确的屏幕坐标。达到了触摸屏校准的目的。

Touch_Adjust 就是根据上面的原理设计的校准函数。其他的函数我们这里就不多介绍了，保存 touch.c 文件，并把该文件加入到 HARDWARE 组下。接下来打开 touch.h 文件，在该文件里面输入如下代码：

```
#ifndef __TOUCH_H__
#define __TOUCH_H__
#include "sys.h"
//Mini STM32 开发板
//ADS7843/7846/UH7843/7846/XPT2046/TSC2046 驱动函数
//正点原子@ALIENTEK
//V1.3
//按键状态
#define Key_Down 0x01
#define Key_Up   0x00
//笔杆结构体
typedef struct
{
    u16 X0;//LCD 坐标
    u16 Y0;
    u16 X; //物理坐标/暂存坐标
    u16 Y;
    u8  Key_Sta;//笔的状态
//触摸屏校准参数
    float xfac;
```

```
    float yfac;
    short xoff;
    short yoff;
}Pen_Holder;
extern Pen_Holder Pen_Point;
//与触摸屏芯片连接引脚
#define PEN    PCin(1)     //PC1    INT
#define DOUT PCin(2)     //PC2    MISO
#define TDIN PCout(3)    //PC3    MOSI
#define TCLK PCout(0)    //PC0    SCLK
#define TCS    PCout(13) //PC13 CS
//ADS7843/7846/UH7843/7846/XPT2046/TSC2046 指令集
#define CMD_RDY 0X90    //0B10010000 即用差分方式读 X 坐标
#define CMD_RDX       0XD0    //0B11010000 即用差分方式读 Y 坐标
#define TEMP_RD 0XF0    //0B11110000 即用差分方式读 Y 坐标
//使用保存
#define ADJ_SAVE_ENABLE
void Touch_Init(void);            //初始化
u8 Read_ADS(u16 *x,u16 *y);       //带舍弃的双方向读取
u8 Read_ADS2(u16 *x,u16 *y); //带加强滤波的双方向坐标读取
u16 ADS_Read_XY(u8 xy);           //带滤波的坐标读取(单方向)
u16 ADS_Read_AD(u8 CMD);      //读取 AD 转换值
void ADS_Write_Byte(u8 num); //向控制芯片写入一个数据
void Drow_Touch_Point(u8 x,u16 y);//画一个坐标叫准点
void Draw_Big_Point(u8 x,u16 y);    //画一个大点
void Touch_Adjust(void);                //触摸屏校准
void Save_Adjdata(void);                //保存校准参数
u8 Get_Adjdata(void);                   //读取校准参数
void Pen_Int_Set(u8 en);                //PEN 中断使能/关闭
void Convert_Pos(void);                 //结果转换函数
#endif
```

该部分代码我们不做多的介绍，最后我们打开 test.c，修改代码如下：

```
//清屏，重新装载对话界面
void Load_Drow_Dialog(void)
{
    LCD_Clear(WHITE);//清屏
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(216,0,"RST");//显示清屏区域
    POINT_COLOR=RED;//设置画笔蓝色
}
int main(void)
{
    u8 key;
```

```
u8 i=0;
Stm32_Clock_Init(9);//系统时钟设置
delay_init(72);       //延时初始化
uart_init(72,9600); //串口 1 初始化
LCD_Init();                  //初始化液晶
KEY_Init();                  //按键初始化
LED_Init();                 //LED 初始化
POINT_COLOR=RED;//设置字体为蓝色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"TOUCH TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2011/1/1");
LCD_ShowString(60,130,"Press KEY0 to Adjust");
Touch_Init();
delay_ms(1500);
Load_Drow_Dialog();
while(1)
{
    key=KEY_Scan();
    if(Pen_Point.Key_Sta==Key_Down)//触摸屏被按下
    {
        Pen_Int_Set(0);//关闭中断
        do
        {
            Pen_Point.Key_Sta=Key_Up;
            Convert_Pos();
            if(Pen_Point.X0>216&&Pen_Point.Y0<16)Load_Drow_Dialog();//清除
            else Draw_Big_Point(Pen_Point.X0,Pen_Point.Y0);//画点
            delay_us(50);
        }while(PEN==0);//如果 PEN 一直有效,则一直执行
        Pen_Int_Set(1);//开启中断
    }else delay_ms(1);
    if(key==1)//KEY0 按下,则执行校准程序
    {
        LCD_Clear(WHITE);//清屏
        Touch_Adjust();  //屏幕校准
        Save_Adjdata();
        Load_Drow_Dialog();
    }
    i++;
    if(i==200)
    {
        i=0;
```

```
                LED0=!LED0;
            }
        };
    }
```

此函数就实现了我们上面介绍的本节所实现的功能。当然这里还用到我们之前写的 24CXX 的代码，用来保存和调用触摸屏的校准信息（在触摸屏校准函数和初始化函数里面）。

3.18.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

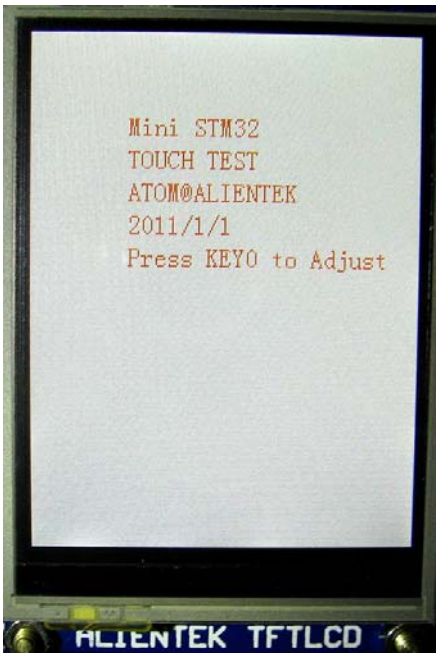


图 3.18.4.1 程序运行效果

如果已经校准过了，则在等待 1.5s 之后进入手写界面，同时 DS0 开始闪烁，界面如下图所示：

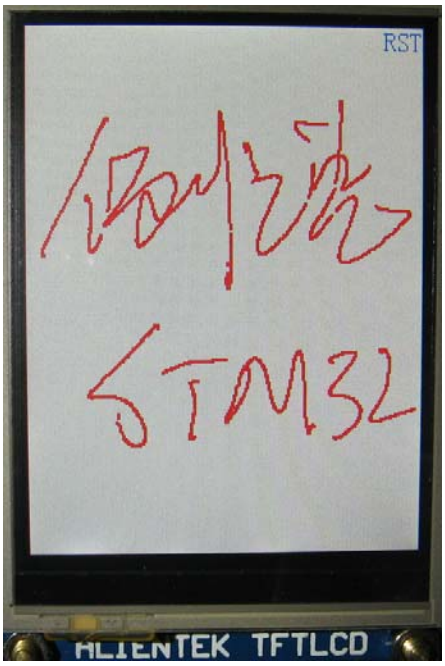


图 3.18.4.2 手写界面

此时，我们就可以在该界面下用笔或者手指输入信息了。如果没有校准过，则会自动进入校准程序（当你发现精度不行的时候，也可以通过按 KEY0 进入校准程序），如下图所示，在校准完成之后自动进入手写界面。

图 3.18.4.3 校准界面

3.19 无线通信实验

ALIENTKE MiniSTM3 开发板带有 2 种 2.4G 无线模块的接口：一个是 NRF24L01 模块；一个是安阳新世纪的 JF24C 模块。前者采用 8 脚插针方式与开发板连接，而后者采用贴板 10PIN 的方式与开发板连接，前者安装拆卸比较方便，后者价格更低。我们将以 NRF24L01 模块为例向大家介绍如何在 ALIENTEK MiniSTM32 开发板上实现无线通信。本节分为如下几个部分：

3.19.1 NRF24L01/JF24C 无线模块简介

3.19.2 硬件设计

3.19.3 软件设计

3.19.4 下载与测试

3.19.1 NRF24L01 无线模块简介

NRF24L01 无线模块，采用的芯片是 NRF24L01，该芯片的主要特点如下：

1）2.4G 全球开放的 ISM 频段，免许可证使用。

2）最高工作速率 2Mbps，高校的 GFSK 调制，抗干扰能力强。

3）125 个可选的频道，满足多点通信和调频通信的需要。

4）内置 CRC 检错和点对多点的通信地址控制。

5）低工作电压（1.9~3.6V）。

6）可设置自动应答，确保数据可靠传输。

该芯片通过 SPI 与外部 MCU 通信，最大的 SPI 速度可以达到 10Mhz。关于该芯片的详细介绍，请参考 24L01 的技术手册。

JF24C 与 NRF24L01 类似，同属 2.4G 频段的无线通信模块，与 NRF24L01 相比，JF24C 的主要优势在成本上面，JF24C 模块成本大概是 NRF24L01 模块的 65%左右。其次是 JF24C 的 RF 功率比 NRF24L01 要大，JF24C 的 RF 输出功率最大可设置到 10dbm，而 NRF24L01 最大也就 0dbm。不过 JF24C 的最大传输速率只有 1Mbps，且没有 NRF24L01 的自动应答功能，如果要实现可靠传输，就必须用户程序来设计，事实上这种方式能更好的实现可靠传输，有更大的实用性，缺点就是编程比较麻烦。JF24C 模块的资料，请大家参考其技术手册。

这两个无线模块各有优势，具体应用的时候，大家可以根据自己的实际情况来选择使用哪个模块。

3.19.2 硬件设计

本节实验功能简介：开机的时候先检测 24L01 是否存在，在检测到 24L01 之后，根据 KEY0 和 KEY1 的设置来决定 24L01 的工作模式，在设定好工作模式之后，就会不停的发送/接收数据，同样用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）KEY0，KEY1。

4）TFTLCD 液晶模块。

5）NRF24L01 模块。

前面 4 个我们在之前的例程里面已经有介绍了，这里我们只介绍最后一个：NRF24L01 模块。我们实验选择的是云佳科技的模块，该模块的引脚图如下：

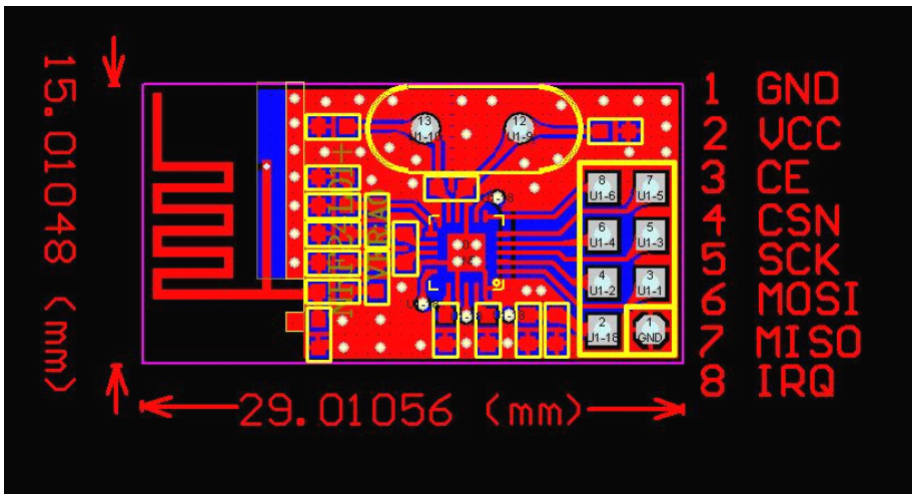


图 3.19.2.1 NRF24L01 模块引脚图

开发板的 2 个无线模块与 CPU 接口原理图如下：

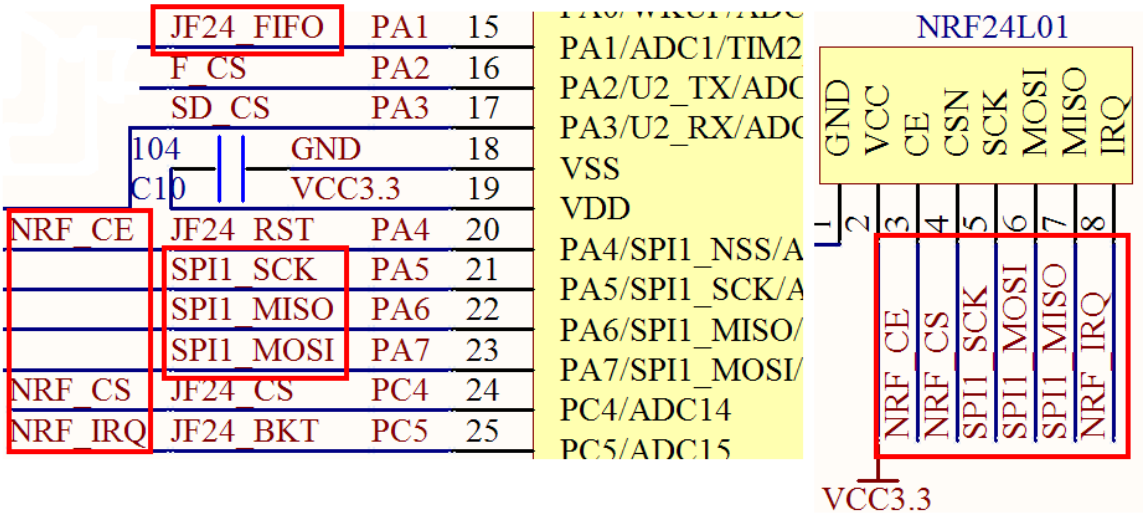


图 3.19.2.2 NRF24L01 模块、JF24C 模块连接原理图

由于无线通信实验是双向的，所以至少要有两个模块同时能工作，这里我们使用2套ALIENTEK MiniSTM32开发板来向大家演示。

3.19.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 NRF24L01 的文件夹。然后新建一个 24l01.c 和 24l01.h 的文件保存在 NRF24L01 文件夹下，并将这个文件夹加入头文件包含路径。

打开 24l01.c 文件，输入如下代码：

```
#include "24l01.h"
#include "lcd.h"
#include "delay.h"
#include "spi.h"
```

```
//Mini STM32 开发板
//NRF24L01 驱动函数
//正点原子@ALIENTEK
const u8 TX_ADDRESS[TX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
const u8 RX_ADDRESS[RX_ADR_WIDTH]={0x34,0x43,0x10,0x10,0x01}; //发送地址
//初始化 24L01 的 IO 口
void NRF24L01_Init(void)
{
    RCC->APB2ENR|=1<<2;    //使能 PORTA 口时钟
    RCC->APB2ENR|=1<<4;    //使能 PORTC 口时钟
    GPIOA->CRL&=0XFFF000FF;//PA4 输出
    GPIOA->CRL|=0X00033300;
    GPIOA->ODR|=7<<2;      //PA2.3.4 输出 1
    GPIOC->CRL&=0XFF00FFFF;//PC4 输出 PC5 输出
    GPIOC->CRL|=0X00830000;
    GPIOC->ODR|=3<<4;      //上拉
    SPIx_Init();    //初始化 SPI
    NRF24L01_CE=0;  //使能 24L01
    NRF24L01_CSN=1; //SPI 片选取消
}
//检测 24L01 是否存在
//返回值:0，成功;1，失败
u8 NRF24L01_Check(void)
{
    u8 buf[5]={0XA5,0XA5,0XA5,0XA5,0XA5};
    u8 i;
    SPIx_SetSpeed(SPI_SPEED_8); //spi 速度为 9Mhz（24L01 的最大 SPI 时钟为 10Mhz）

    NRF24L01_Write_Buf(WRITE_REG+TX_ADDR,buf,5);//写入 5 个字节的地址.
    NRF24L01_Read_Buf(TX_ADDR,buf,5); //读出写入的地址
    for(i=0;i<5;i++)if(buf[i]!=0XA5)break;
    if(i!=5)return 1;//检测 24L01 错误
    return 0;        //检测到 24L01
}
//SPI 写寄存器
//reg:指定寄存器地址
//value:写入的值
u8 NRF24L01_Write_Reg(u8 reg,u8 value)
{
    u8 status;
    NRF24L01_CSN=0;                 //使能 SPI 传输
    status =SPIx_ReadWriteByte(reg);//发送寄存器号
    SPIx_ReadWriteByte(value);      //写入寄存器的值
```

```
    NRF24L01_CSN=1;                 //禁止SPI传输
    return(status);                 //返回状态值
}
//读取SPI寄存器值
//reg:要读的寄存器
u8 NRF24L01_Read_Reg(u8 reg)
{
    u8 reg_val;
    NRF24L01_CSN = 0;           //使能SPI传输
    SPIx_ReadWriteByte(reg);    //发送寄存器号
    reg_val=SPIx_ReadWriteByte(0XFF);//读取寄存器内容
    NRF24L01_CSN = 1;           //禁止SPI传输
    return(reg_val);            //返回状态值
}
//在指定位置读出指定长度的数据
//reg:寄存器(位置)
//*pBuf:数据指针
//len:数据长度
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Read_Buf(u8 reg,u8 *pBuf,u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN = 0;           //使能SPI传输
    status=SPIx_ReadWriteByte(reg);//发送寄存器值(位置),并读取状态值
    for(u8_ctr=0;u8_ctr<len;u8_ctr++)pBuf[u8_ctr]=SPIx_ReadWriteByte(0XFF);//
读出数据
    NRF24L01_CSN=1;         //关闭SPI传输
    return status;          //返回读到的状态值
}
//在指定位置写指定长度的数据
//reg:寄存器(位置)
//*pBuf:数据指针
//len:数据长度
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Write_Buf(u8 reg, u8 *pBuf, u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN = 0;           //使能SPI传输
    status = SPIx_ReadWriteByte(reg);//发送寄存器值(位置),并读取状态值
    for(u8_ctr=0; u8_ctr<len; u8_ctr++)SPIx_ReadWriteByte(*pBuf++); //写入数据

    NRF24L01_CSN = 1;       //关闭SPI传输
    return status;          //返回读到的状态值
```

```
}
//启动 NRF24L01 发送一次数据
//txbuf:待发送数据首地址
//返回值:发送完成状况
u8 NRF24L01_TxPacket(u8 *txbuf)
{
    u8 sta;
    SPIx_SetSpeed(SPI_SPEED_8);//spi 速度为 9Mhz（24L01 的最大 SPI 时钟为 10Mhz）
    NRF24L01_CE=0;
    NRF24L01_Write_Buf(WR_TX_PLOAD,txbuf,TX_PLOAD_WIDTH);//写数据到 TX BUF  32
个字节
    NRF24L01_CE=1;//启动发送
    while(NRF24L01_IRQ!=0);//等待发送完成
    sta=NRF24L01_Read_Reg(STATUS);  //读取状态寄存器的值
    NRF24L01_Write_Reg(WRITE_REG+STATUS,sta); //清除 TX_DS 或 MAX_RT 中断标志
    if(sta&MAX_TX)//达到最大重发次数
    {
        NRF24L01_Write_Reg(FLUSH_TX,0xff);//清除 TX FIFO 寄存器
        return MAX_TX;
    }
    if(sta&TX_OK)//发送完成
    {
        return TX_OK;
    }
    return 0xff;//其他原因发送失败
}
//启动 NRF24L01 发送一次数据
//txbuf:待发送数据首地址
//返回值:0，接收完成；其他，错误代码
u8 NRF24L01_RxPacket(u8 *rxbuf)
{
    u8 sta;
    SPIx_SetSpeed(SPI_SPEED_8); //spi 速度为 9Mhz（24L01 的最大 SPI 时钟为 10Mhz）
    sta=NRF24L01_Read_Reg(STATUS);  //读取状态寄存器的值
    NRF24L01_Write_Reg(WRITE_REG+STATUS,sta); //清除 TX_DS 或 MAX_RT 中断标志
    if(sta&RX_OK)//接收到数据
    {
        NRF24L01_Read_Buf(RD_RX_PLOAD,rxbuf,RX_PLOAD_WIDTH);//读取数据
        NRF24L01_Write_Reg(FLUSH_RX,0xff);//清除 RX FIFO 寄存器
        return 0;
    }
    return 1;//没收到任何数据
}
```

```
//该函数初始化 NRF24L01 到 RX 模式
//设置 RX 地址,写 RX 数据宽度,选择 RF 频道,波特率和 LNA HCURR
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
void RX_Mode(void)
{
    NRF24L01_CE=0;
    NRF24L01_Write_Buf(WRITE_REG+RX_ADDR_P0,(u8*)RX_ADDRESS,RX_ADR_WIDTH);//写
RX 节点地址

    NRF24L01_Write_Reg(WRITE_REG+EN_AA,0x01);     //使能通道 0 的自动应答
    NRF24L01_Write_Reg(WRITE_REG+EN_RXADDR,0x01);//使能通道 0 的接收地址
    NRF24L01_Write_Reg(WRITE_REG+RF_CH,40);       //设置 RF 通信频率
    NRF24L01_Write_Reg(WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH);//选择通道 0 的有效数
据宽度
    NRF24L01_Write_Reg(WRITE_REG+RF_SETUP,0x0f);// 设 置 TX 发 射 参 数,0db 增
益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG+CONFIG,  0x0f);// 配 置 基 本 工 作 模 式 的 参
数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
    NRF24L01_CE = 1; //CE 为高,进入接收模式
}
//该函数初始化 NRF24L01 到 TX 模式
//设置 TX 地址,写 TX 数据宽度,设置 RX 自动应答的地址,填充 TX 发送数据,选择 RF 频道,
波特率和 LNA HCURR
//PWR_UP,CRC 使能
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
//CE 为高大于 10us,则启动发送.
void TX_Mode(void)
{
    NRF24L01_CE=0;
    NRF24L01_Write_Buf(WRITE_REG+TX_ADDR,(u8*)TX_ADDRESS,TX_ADR_WIDTH);//写 TX
节点地址
    NRF24L01_Write_Buf(WRITE_REG+RX_ADDR_P0,(u8*)RX_ADDRESS,RX_ADR_WIDTH);  //
设置 TX 节点地址,主要为了使能 ACK

    NRF24L01_Write_Reg(WRITE_REG+EN_AA,0x01);     //使能通道 0 的自动应答
    NRF24L01_Write_Reg(WRITE_REG+EN_RXADDR,0x01); //使能通道 0 的接收地址
    NRF24L01_Write_Reg(WRITE_REG+SETUP_RETR,0x1a);//设置自动重发间隔时间:500us
+ 86us;最大自动重发次数:10 次
    NRF24L01_Write_Reg(WRITE_REG+RF_CH,40);       //设置 RF 通道为 40
    NRF24L01_Write_Reg(WRITE_REG+RF_SETUP,0x0f);  //设置 TX 发射参数,0db 增
益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG+CONFIG,0x0e);    //配置基本工作模式的参
数;PWR_UP,EN_CRC,16BIT_CRC,接收模式,开启所有中断
```

```
    NRF24L01_CE=1;//CE 为高,10us 后启动发送
}
```

此部分代码我们不多介绍，在这里强调一个要注意的地方，在 NRF24L01_Init 函数里面，我们调用了 SPIx_Init()函数，但是这里痛实例 17 的初始化有点点区别，在实例 17 里面，SPI 空闲的时候 SCK 是高电平的，而 NRF24L01 的 SPI 通信时序如下：

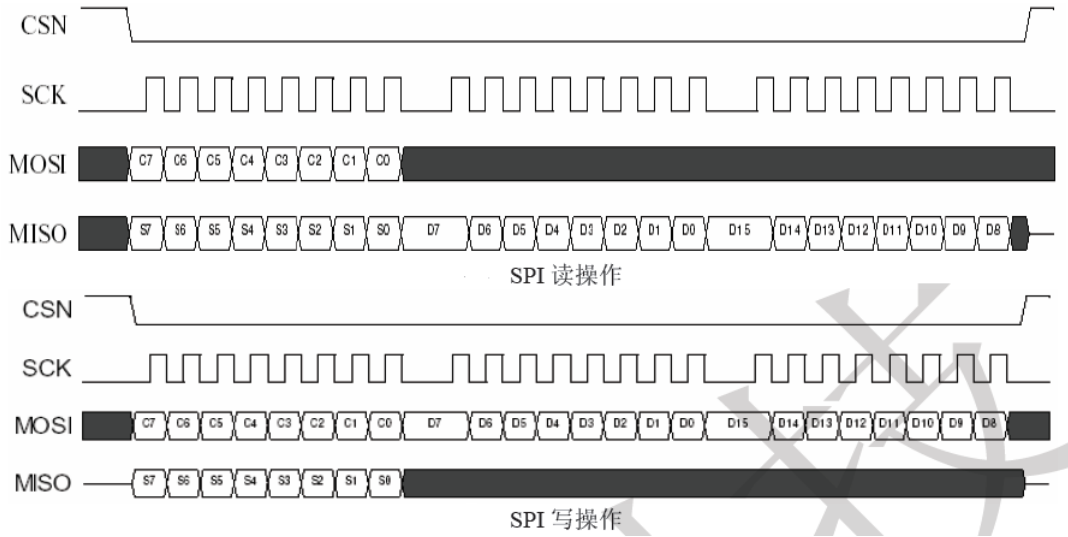


图 3.19.3.1 NRF24L01 读写操作时序

上图中 Cn 代表指令位，Sn 代表状态寄存器位，Dn 代表数据位。从图中可以看出，SCK 在平时是低电平的，而数据在 SCK 的上升沿被读写。所以，我们需要设置 SPI 的 CPOL 和 CPHA 均为 0，来满足 NRF24L01 对 SPI 操作的要求。

保存 24l01.c 文件，加入到 HARDWARE 组下。接下来打开 24l01.h，输入如下代码：

```
#ifndef __24L01_H
#define __24L01_H
#include "sys.h"
//Mini STM32 开发板
//NRF24L01 驱动函数
//正点原子@ALIENTEK
//////////////////////////////////////////////////////////////////////////////////
//NRF24L01 寄存器操作命令
#define READ_REG        0x00  //读配置寄存器,低 5 位为寄存器地址
#define WRITE_REG       0x20  //写配置寄存器,低 5 位为寄存器地址
#define RD_RX_PLOAD     0x61  //读 RX 有效数据,1~32 字节
#define WR_TX_PLOAD     0xA0  //写 TX 有效数据,1~32 字节
#define FLUSH_TX        0xE1  //清除 TX FIFO 寄存器.发射模式下用
#define FLUSH_RX        0xE2  //清除 RX FIFO 寄存器.接收模式下用
#define REUSE_TX_PL     0xE3  //重新使用上一包数据,CE 为高,数据包被不断发送.
#define NOP             0xFF  //空操作,可以用来读状态寄存器
//SPI(NRF24L01)寄存器地址
#define CONFIG          0x00  //配置寄存器地址;bit0:1 接收模式,0 发射模式;bit1:
电选择;bit2:CRC 模式;bit3:CRC 使能;
                              //bit4:中断 MAX_RT(达到最大重发次数中断)使
```

```
能;bit5:中断 TX_DS 使能;bit6:中断 RX_DR 使能
    #define EN_AA           0x01  //使能自动应答功能  bit0~5,对应通道 0~5
    #define EN_RXADDR       0x02  //接收地址允许,bit0~5,对应通道 0~5
    #define SETUP_AW          0x03  //设置地址宽度(所有数据通道):bit1,0:00,3 字
节;01,4 字节;02,5 字节;
    #define SETUP_RETR      0x04  //建立自动重发;bit3:0,自动重发计数器;bit7:4,自动
重发延时 250*x+86us
    #define RF_CH           0x05  //RF 通道,bit6:0,工作通道频率;
    #define RF_SETUP        0x06  //RF 寄存器;bit3:传输速率(0:1Mbps,1:2Mbps);bit2:1,
发射功率;bit0:低噪声放大器增益
    #define STATUS          0x07  //状态寄存器;bit0:TX FIFO 满标志;bit3:1,接收数据
通道号(最大:6);bit4,达到最多次重发
                                  //bit5:数据发送完成中断;bit6:接收数据中断;
    #define MAX_TX      0x10  //达到最大发送次数中断
    #define TX_OK       0x20  //TX 发送完成中断
    #define RX_OK       0x40  //接收到数据中断

    #define OBSERVE_TX      0x08  //发送检测寄存器,bit7:4,数据包丢失计数器;bit3:0,
重发计数器
    #define CD              0x09  //载波检测寄存器,bit0,载波检测;
    #define RX_ADDR_P0      0x0A  //数据通道 0 接收地址,最大长度 5 个字节,低字节在前
    #define RX_ADDR_P1      0x0B  //数据通道 1 接收地址,最大长度 5 个字节,低字节在前
    #define RX_ADDR_P2      0x0C  //数据通道 2 接收地址,最低字节可设置,高字节,必须同
RX_ADDR_P1[39:8]相等;
    #define RX_ADDR_P3      0x0D  //数据通道 3 接收地址,最低字节可设置,高字节,必须同
RX_ADDR_P1[39:8]相等;
    #define RX_ADDR_P4      0x0E  //数据通道 4 接收地址,最低字节可设置,高字节,必须同
RX_ADDR_P1[39:8]相等;
    #define RX_ADDR_P5      0x0F  //数据通道 5 接收地址,最低字节可设置,高字节,必须同
RX_ADDR_P1[39:8]相等;
    #define TX_ADDR           0x10  //发送地址(低字节在前),ShockBurstTM 模式
下,RX_ADDR_P0 与此地址相等
    #define RX_PW_P0        0x11  //接收数据通道 0 有效数据宽度(1~32 字节),设置为 0
则非法
    #define RX_PW_P1        0x12  //接收数据通道 1 有效数据宽度(1~32 字节),设置为 0
则非法
    #define RX_PW_P2        0x13  //接收数据通道 2 有效数据宽度(1~32 字节),设置为 0
则非法
    #define RX_PW_P3        0x14  //接收数据通道 3 有效数据宽度(1~32 字节),设置为 0
则非法
    #define RX_PW_P4        0x15  //接收数据通道 4 有效数据宽度(1~32 字节),设置为 0
则非法
    #define RX_PW_P5        0x16  //接收数据通道 5 有效数据宽度(1~32 字节),设置为 0
```

则非法

```
#define FIFO_STATUS        0x17   //FIFO 状态寄存器;bit0,RX FIFO 寄存器空标志;bit1,RX FIFO 满标志;bit2,3,保留
                                  //bit4,TX FIFO 空标志;bit5,TX FIFO 满标志;bit6,1,循环发送上一数据包.0,不循环;
//////////////////////////////////////////////////////////////////////////////////
//24L01 操作线
#define NRF24L01_CE    PAout(4) //24L01 片选信号
#define NRF24L01_CSN   PCout(4) //SPI 片选信号
#define NRF24L01_IRQ   PCin(5)  //IRQ 主机数据输入
//24L01 发送接收数据宽度定义
#define TX_ADR_WIDTH    5   //5 字节的地址宽度
#define RX_ADR_WIDTH    5   //5 字节的地址宽度
#define TX_PLOAD_WIDTH  32  //20 字节的用户数据宽度
#define RX_PLOAD_WIDTH  32  //20 字节的用户数据宽度


void NRF24L01_Init(void);//初始化
void RX_Mode(void);//配置为接收模式
void TX_Mode(void);//配置为发送模式
u8 NRF24L01_Write_Buf(u8 reg, u8 *pBuf, u8 u8s);//写数据区
u8 NRF24L01_Read_Buf(u8 reg, u8 *pBuf, u8 u8s);//读数据区
u8 NRF24L01_Read_Reg(u8 reg);           //读寄存器
u8 NRF24L01_Write_Reg(u8 reg, u8 value);//写寄存器
u8 NRF24L01_Check(void);//检查 24L01 是否存在
u8 NRF24L01_TxPacket(u8 *txbuf);//发送一个包的数据
u8 NRF24L01_RxPacket(u8 *rxbuf);//接收一个包的数据
#endif
```

该部分代码，主要定义了一些 24L01 的命令字，以及函数声明，这里还通过 TX_PLOAD_WIDTH 和 RX_PLOAD_WIDTH 决定了发射和接收的数据宽度，也就是我们每次发射和接受的有效字节数。NRF24L01 每次最多传输 32 个字节，再多的字节传输则需要多次传送。

保存 24l01.h 文件，接下来我们在主函数里面写入我们的实现代码，来达到我们所要求的功能。打开 teset.c 文件在该文件内修改 main 函数如下：

```
//收发都做在一个函数里面,通过按键来确定进入发送模式,还是接收模式
int main(void)
{
    u8 key,mode;
    u16 t=0;
    u8 tmp_buf[33];

    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72,9600); //串口 1 初始化
```

```
LCD_Init();          //初始化液晶
KEY_Init();     //按键初始化
LED_Init();          //LED 初始化
NRF24L01_Init();    //初始化 NRF24L01
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"NRF24L01 TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2011/1/1");
while(NRF24L01_Check())//检测不到 24L01
{
    LCD_ShowString(60,130,"24L01 Check Failed!");
    delay_ms(500);
    LCD_ShowString(60,130,"Please Check!      ");
    delay_ms(500);
    LED0=!LED0;//DS0 闪烁
}
LCD_ShowString(60,130,"24L01 Ready!");
LCD_ShowString(10,150,"KEY0:RX_Mode  KEY1:TX_Mode");
while(1)//在该部分确定进入哪个模式!
{
    key=KEY_Scan();
    if(key==1)
    {
        mode=0;
        break;
    }else if(key==2)
    {
        mode=1;
        break;
    }
    t++;
    if(t==100) //闪烁显示提示信息
    {
        LCD_ShowString(10,150,"                              ");//清空显示
    }
    if(t==200)
    {
        t=0;
        LCD_ShowString(10,150,"KEY0:RX_Mode  KEY1:TX_Mode");
    }
    delay_ms(5);
```

```
}
LCD_Fill(10,150,240,166,WHITE);//清空上面的显示
POINT_COLOR=BLUE;//设置字体为蓝色
if(mode==0)//RX 模式
{
    LCD_ShowString(60,150,"NRF24L01 RX_Mode");
    LCD_ShowString(60,170,"Received DATA:");
    RX_Mode();
    while(1)
    {
        if(NRF24L01_RxPacket(tmp_buf)==0)//一旦接收到信息,则显示出来.
        {
            tmp_buf[32]=0;//加入字符串结束符
            LCD_ShowString(0,190,tmp_buf);
        }else delay_us(100);
        t++;
        if(t==10000)//大约 1s 钟改变一次状态
        {
            t=0;
            LED0=!LED0;
        }
    };
}else//TX 模式
{
    LCD_ShowString(60,150,"NRF24L01 TX_Mode");
    TX_Mode();
    mode=' ';//从空格键开始
    while(1)
    {
        if(NRF24L01_TxPacket(tmp_buf)==TX_OK)
        {
            LCD_ShowString(60,170,"Sended DATA:");
            LCD_ShowString(0,190,tmp_buf);
            key=mode;
            for(t=0;t<32;t++)
            {
                key++;
                if(key>('~'))key=' ';
                tmp_buf[t]=key;
            }
            mode++;
            if(mode>'~')mode=' ';
            tmp_buf[32]=0;//加入结束符
```

```
            }else
            {
                LCD_ShowString(60,170,"Send Failed ");
                LCD_Fill(0,188,240,218,WHITE);//清空上面的显示
            };
            LED0=!LED0;
            delay_ms(1500);
        };
    }
}
```

至此，我们整个实验的软件设计就完成了。

3.19.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容（默认 NRF24L01 已经接上了）：

图 3.19.4.1 选择工作模式界面

通过 KEY0 和 KEY1 来选择 NRF24L01 模块所要进入的工作模式，我们两个开发板一个选择发送，一个选择接收就可以了。

设置好后通信界面如下：

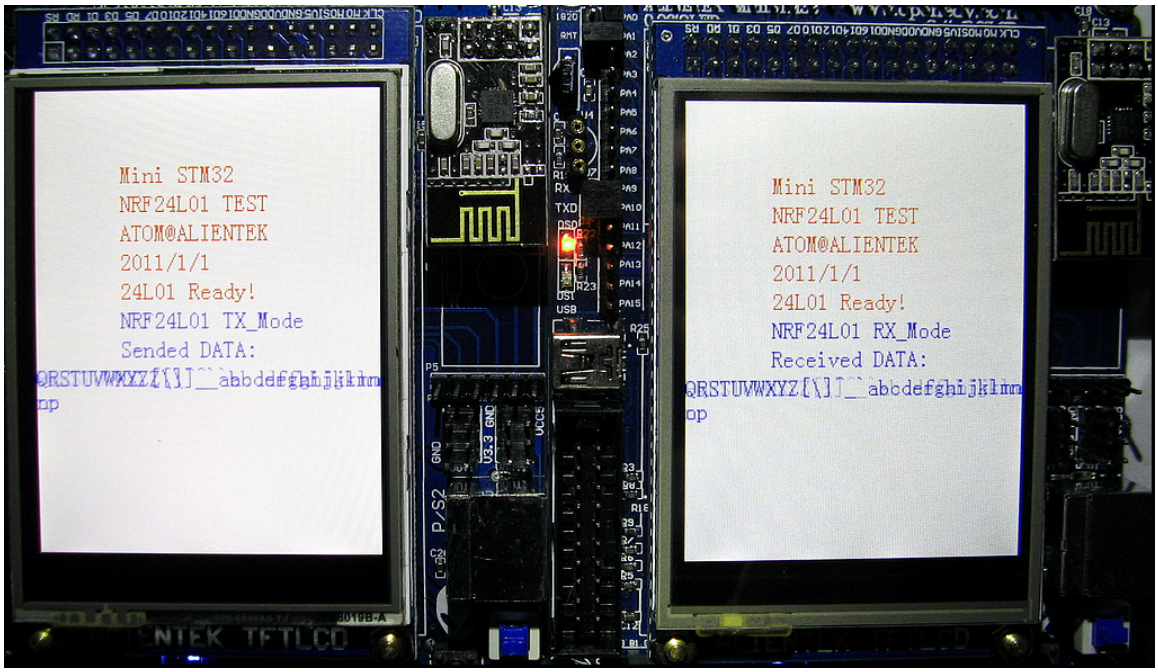

Mini STM32
NRF24L01 TEST
ATOM@ALIENTEK
2011/1/1
24L01 Ready!
NRF24L01 TX_Mode
Sended DATA:
Mini STM32
NRF24L01 TEST
ATOM@ALIENTEK
2011/1/1
24L01 Ready!
NRF24L01 RX_Mode
Received DATA:
QRSTUVWXYZ[\]^_`abcdefghijklmn
op
RX
TXD
P/S2
NTEK TFTLCD

图 3.19.4.2 通信界面

3.20 SD卡实验

很多单片机系统都需要大容量存储设备，以存储数据。目前常用的有 U 盘，FLASH 芯片，SD 卡等。他们各有优点，综合比较，最适合单片机系统的莫过于 SD 卡了，它不仅容量可以做到很大（32Gb 以上），而且支持 SPI 接口，方便移动，有几种体积的尺寸可供选择（标准的 SD 卡尺寸，以及 TF 卡尺寸），能满足不同应用的要求。只需要 4 个 IO 口，就可以外扩一个最大达 32GB 以上的外部存储器，容量选择尺度很大，更换也很方便，而且方便移动，编程也比较简单，是单片机大容量外部存储器的首选。

ALIENTKE MiniSTM3 开发板就带有 SD 卡接口，利用 STM32 自带的 SPI 接口，最大通信速度可达 18Mbps，每秒可传输数据 2M 字节以上，对于一般应用足够了。本节将向大家介绍，如何在 ALIENTEK MiniSTM32 开发板上读取 SD 卡。本节分为如下几个部分：

3.20.1 SD 卡简介
3.20.2 硬件设计
3.20.3 软件设计
3.20.4 下载与测试

3.20.1 SD 卡简介

SD 卡（Secure Digital Memory Card）中文翻译为安全数码卡，是一种基于半导体快闪记忆器的新一代记忆设备，它被广泛地于便携式装置上使用，例如数码相机、个人数码助理(PDA)和多媒体播放器等。SD 卡由日本松下、东芝及美国 SanDisk 公司于 1999 年 8 月共同开发研制。大小犹如一张邮票的 SD 记忆卡，重量只有 2 克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。

SD 卡一般支持 2 种操作模式：

1，SD 卡模式；

2，SPI 模式；

主机可以选择以上任意一种模式同 SD 卡通信，SD 卡模式允许 4 线的高速数据传输。SPI 模式允许简单的通过 SPI 接口来和 SD 卡通信，这种模式同 SD 卡模式相比就是丧失了速度。

SD 卡的引脚排序如下图所示：

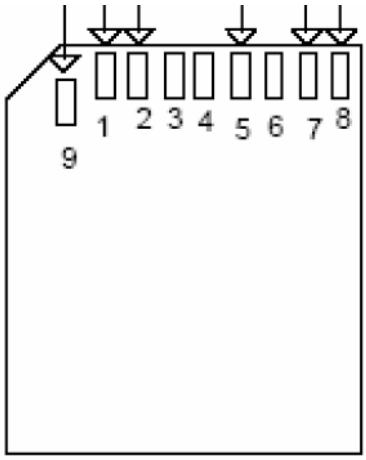


图 3.20.1.1 SD 卡引脚排序图

SD 卡引脚功能描述如下表所示：

针脚	1	2	3	4	5	6	7	8	9
SD卡模式	CD/DAT3	CMD	VSS	VCC	CLK	VSS	DAT0	DAT1	DAT2
SPI模式	CS	MOSI	VSS	VCC	CLK	VSS	MISO	NC	NC

表 3.20.1.1 SD 卡引脚功能表

SD 卡只能使用 3.3V 的 IO 电平，所以，MCU 一定要能够支持 3.3V 的 IO 端口输出。注意：在 SPI 模式下，CS/MOSI/MISO/CLK 都需要加 10~100K 左右的上拉电阻。

SD 卡要进入 SPI 模式很简单，就是在 SD 卡收到复位命令（CMD0）时，CS 为有效电平（低电平）则 SPI 模式被启用。不过在发送 CMD0 之前，要发送>74 个时钟，这是因为 SD 卡内部有个供电电压上升时间，大概为 64 个 CLK，剩下的 10 个 CLK 用于 SD 卡同步，之后才能开始 CMD0 的操作，在卡初始化的时候，CLK 时钟最大不能超过 400Khz！。

ALENTEK MiniSTM32 开发板使用的是 SPI 模式来读写 SD 卡，下面我们就重点介绍一下 SD 卡在 SPI 模式下的相关操作。

首先介绍 SPI 模式下几个重要的操作命令，如下表所示：

命令	参数	回应	描述
CMD0(0X00)	NONE	R1	复位 SD 卡
CMD9(0X09)	NONE	R1	读取卡特定数据寄存器
CMD10(0X0A)	NONE	R1	读取卡标志数据寄存器
CMD16(0X10)	块大小	R1	设置块大小（字节数）
CMD17(0X11)	地址	R1	读取一个块的数据
CMD24(0X18)	地址	R1	写入一个块的数据
CMD41(0X29)	NONE	R1	引用命令的前命令
CMD55(0X37)	NONE	R1	开始卡的初始化
CMD59(0X3B)	仅最后一位有效	R1	设置 CRC 开启(1)或关闭(0)

表 3.20.1.2 SPI 模式下 SD 卡部分操作指令

其中 R1 的回应格式如下表所示：

SD卡 R1回应格式							
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0	参数错误	地址错误	连续擦除错误	命令CRC错误	非法命令	擦除复位	IDLE状态

表 3.20.1.3 SD 卡 R1 回应格式

接着我们看看 SD 卡的初始化，SD 卡的典型初始化过程如下：

1、初始化与 SD 卡连接的硬件条件（MCU 的 SPI 配置，IO 口配置）；

2、上电延时（>74 个 CLK）；

3、复位卡（CMD0）；

4、激活卡，内部初始化并获取卡类型（CMD1（用于 MMC 卡）、CMD55、CMD41）；

5.、查询 OCR，获取供电状况（CMD58）；

6、是否使用 CRC（CMD59）；

7、设置读写块数据长度（CMD16）；

8、读取 CSD，获取存储卡的其他信息（CMD9）；

9、发送 8CLK 后，禁止片选；

这样我们就完成了对 SD 卡的初始化，这里面我们一般设置读写块数据长度为 512 个字节，并禁止使用 CRC。在完成了初始化之后，就可以开始读写数据了。

SD 卡读取数据，这里通过 CMD17 来实现，具体过程如下：

1、发送 CMD17；

2、接收卡响应 R1；

3、接收数据起始令牌 0XFE；

4、接收数据；

5、接收 2 个字节的 CRC，如果没有开启 CRC，这两个字节在读取后可以丢掉。

6、8CLK 之后禁止片选；

以上就是一个典型的读取 SD 卡数据过程，SD 卡的写于读数据差不多，写数据通过 CMD24 来实现，具体过程如下：

1、发送 CMD24;

2、接收卡响应 R1；

3、发送写数据起始令牌 0XFE；

4、发送数据；

5、发送 2 字节的伪 CRC；

6、8CLK 之后禁止片选；

以上就是一个典型的写 SD 卡过程。关于 SD 卡的介绍，我们就介绍到这里，更详细的介绍请参考 SD 卡的参考资料。

3.20.2 硬件设计

本节实验功能简介：开机的时候先初始化 SD 卡，如果 SD 卡初始化完成，则读取扇区 0 的数据，然后通过串口打印到电脑上。如果没初始化通过，则在 LCD 上提示初始化失败。同样用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）串口 1。

4）TFTLCD 液晶模块。

5）SD 卡。

前面四部分，在之前的实例已经介绍过了，这里我们介绍一下SD卡在开发板上 的连接方式，SD卡与MCU的连接原理图如下：

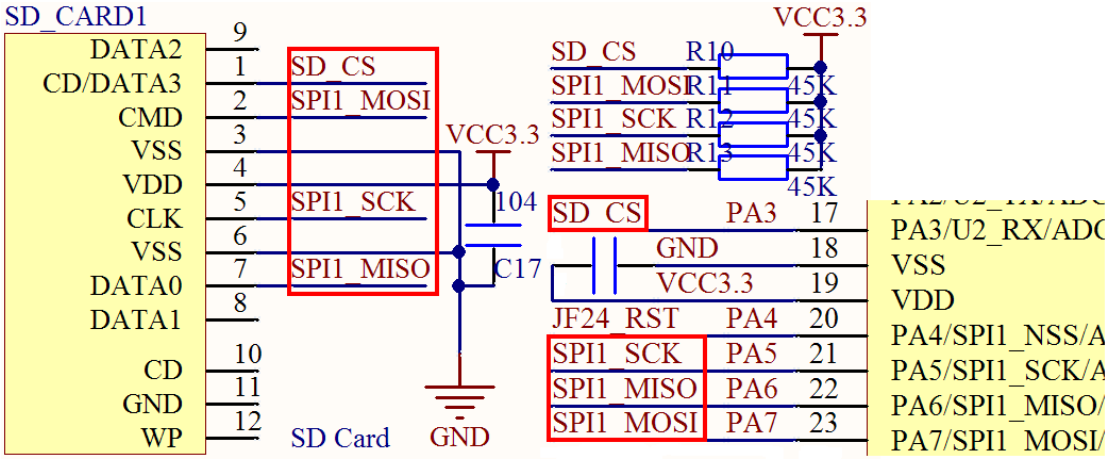


图3.20.2.1 SD卡与STM32连接电路图

3.20.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 SD 的文件夹。然后新建一个 MMC_SD.C 和 MMC_SD.H 的文件保存在 SD 文件夹下，并将这个文件夹加入头文件包含路径。

打开 MMC_SD.C 文件，输入如下代码：

```
#include "sys.h"
#include "mmc_sd.h"
#include "spi.h"
#include "usart.h"
#include "delay.h"
u8  SD_Type=0;//SD 卡的类型
```

```
//Mini STM32 开发板
//SD 卡 驱动
//正点原子@ALIENTEK
//增加了一些延时,实测可以支持 TF 卡(1G/2G),金士顿 2G,4G 16G SD 卡
//2010/6/24
//加入了 u8 SD_GetResponse(u8 Response)函数
//修改了 u8 SD_WaitDataReady(void)函数
//增加了 USB 读卡器支持的 u8 MSD_ReadBuffer(u8* pBuffer, u32 ReadAddr, u32 NumByteToRead);
//和 u8 MSD_WriteBuffer(u8* pBuffer, u32 WriteAddr, u32 NumByteToWrite);两个函数
//等待 SD 卡回应
//Response:要得到的回应值
//返回值:0,成功得到了该回应值
//    其他,得到回应值失败
u8 SD_GetResponse(u8 Response)
{
    u16 Count=0xFFF;//等待次数
    while ((SPIx_ReadWriteByte(0XFF)!=Response)&&Count)Count--;//等待得到准确的回应
    if (Count==0)return MSD_RESPONSE_FAILURE;//得到回应失败
    else return MSD_RESPONSE_NO_ERROR;//正确回应
}
//等待 SD 卡写入完成
//返回值:0,成功;
//    其他,错误代码;
u8 SD_WaitDataReady(void)
{
    u8 r1=MSD_DATA_OTHER_ERROR;
    u32 retry;
    retry=0;
    do
    {
        r1=SPIx_ReadWriteByte(0xFF)&0X1F;//读到回应
        if(retry==0xfffe)return 1;
        retry++;
        switch (r1)
        {
            case MSD_DATA_OK://数据接收正确了
                r1=MSD_DATA_OK;
                break;
            case MSD_DATA_CRC_ERROR:  //CRC 校验错误
                return MSD_DATA_CRC_ERROR;
```

```
            case MSD_DATA_WRITE_ERROR://数据写入错误
                return MSD_DATA_WRITE_ERROR;
            default://未知错误
                r1=MSD_DATA_OTHER_ERROR;
                break;
        }
    }while(r1==MSD_DATA_OTHER_ERROR); //数据错误时一直等待
    retry=0;
    while(SPIx_ReadWriteByte(0XFF)==0)//读到数据为 0,则数据还未写完成
    {
        retry++;
        //delay_us(10);//SD 卡写等待需要较长的时间
        if(retry>=0XFFFFFFFE)return 0XFF;//等待失败了
    };
    return 0;//成功了
}
//向 SD 卡发送一个命令
//输入: u8 cmd   命令
//      u32 arg  命令参数
//      u8 crc   crc 校验值
//返回值:SD 卡返回的响应
u8 SD_SendCommand(u8 cmd, u32 arg, u8 crc)
{
    u8 r1;
    u8 Retry=0;
    SD_CS=1;
    SPIx_ReadWriteByte(0xff);//高速写命令延时
    SPIx_ReadWriteByte(0xff);
    SPIx_ReadWriteByte(0xff);
    //片选端置低，选中 SD 卡
    SD_CS=0;
    //发送
    SPIx_ReadWriteByte(cmd | 0x40);//分别写入命令
    SPIx_ReadWriteByte(arg >> 24);
    SPIx_ReadWriteByte(arg >> 16);
    SPIx_ReadWriteByte(arg >> 8);
    SPIx_ReadWriteByte(arg);
    SPIx_ReadWriteByte(crc);
    //等待响应，或超时退出
    while((r1=SPIx_ReadWriteByte(0xFF))==0xFF)
    {
        Retry++;
        if(Retry>200)break;
```

```
    }
    //关闭片选
    SD_CS=1;
    //在总线上额外增加 8 个时钟，让 SD 卡完成剩下的工作
    SPIx_ReadWriteByte(0xFF);
    //返回状态值
    return r1;
}

//向 SD 卡发送一个命令(结束是不失能片选，还有后续数据传来)
//输入:u8 cmd   命令
//     u32 arg  命令参数
//     u8 crc   crc 校验值
//返回值:SD 卡返回的响应

u8 SD_SendCommand_NoDeassert(u8 cmd, u32 arg, u8 crc)
{
    u8 Retry=0;
    u8 r1;
    SPIx_ReadWriteByte(0xff);//高速写命令延时
    SPIx_ReadWriteByte(0xff);
    SD_CS=0;//片选端置低，选中 SD 卡
    //发送
    SPIx_ReadWriteByte(cmd | 0x40); //分别写入命令
    SPIx_ReadWriteByte(arg >> 24);
    SPIx_ReadWriteByte(arg >> 16);
    SPIx_ReadWriteByte(arg >> 8);
    SPIx_ReadWriteByte(arg);
    SPIx_ReadWriteByte(crc);
    //等待响应，或超时退出
    while((r1=SPIx_ReadWriteByte(0xFF))==0xFF)
    {
        Retry++;
        if(Retry>200)break;
    }
    //返回响应值
    return r1;
}
//把 SD 卡设置到挂起模式
//返回值:0,成功设置
//       1,设置失败
u8 SD_Idle_Sta(void)
{
```

```
    u16 i;
    u8 retry;
    for(i=0;i<0xf00;i++);//纯延时，等待 SD 卡上电完成
    //先产生>74 个脉冲，让 SD 卡自己初始化完成
    for(i=0;i<10;i++)SPIx_ReadWriteByte(0xFF);
    //----------------SD 卡复位到 idle 开始-----------------
    //循环连续发送 CMD0，直到 SD 卡返回 0x01,进入 IDLE 状态
    //超时则直接退出
    retry = 0;
    do
    {
        //发送 CMD0，让 SD 卡进入 IDLE 状态
        i = SD_SendCommand(CMD0, 0, 0x95);
        retry++;
    }while((i!=0x01)&&(retry<200));
    //跳出循环后，检查原因：初始化成功？or 重试超时？
    if(retry==200)return 1; //失败
    return 0;//成功
}
//初始化 SD 卡
//如果成功返回,则会自动设置 SPI 速度为 18Mhz
//返回值:0：NO_ERR
//       1：TIME_OUT
//      99：NO_CARD
u8 SD_Init(void)
{
    u8 r1;      // 存放 SD 卡的返回值
    u16 retry;  // 用来进行超时计数
    u8 buff[6];
    //设置硬件上与 SD 卡相关联的控制引脚输出
    //避免 NRF24L01/W25X16 等的影响
    RCC->APB2ENR|=1<<2;        //PORTA 时钟使能
    GPIOA->CRL&=0XFFF000FF;
    GPIOA->CRL|=0X00033300;//PA2.3.4 推挽
    GPIOA->ODR|=0X7<<2;     //PA2.3.4 上拉
    SPIx_Init();
    SPIx_SetSpeed(SPI_SPEED_256);//设置到低速模式
    SD_CS=1;
    if(SD_Idle_Sta()) return 1;//超时返回 1 设置到 idle 模式失败
    //----------------SD 卡复位到 idle 结束-----------------
    //获取卡片的 SD 版本信息
    SD_CS=0;
    r1 = SD_SendCommand_NoDeassert(8, 0x1aa,0x87);
```

```
//如果卡片版本信息是 v1.0 版本的，即 r1=0x05，则进行以下初始化
if(r1 == 0x05)
{
    //设置卡类型为 SDV1.0，如果后面检测到为 MMC 卡，再修改为 MMC
    SD_Type = SD_TYPE_V1;
    //如果是 V1.0 卡，CMD8 指令后没有后续数据
    //片选置高，结束本次命令
    SD_CS=1;
    //多发 8 个 CLK，让 SD 结束后续操作
    SPIx_ReadWriteByte(0xFF);
    //-----------------SD 卡、MMC 卡初始化开始-----------------
    //发卡初始化指令 CMD55+ACMD41
    // 如果有应答，说明是 SD 卡，且初始化完成
    // 没有回应，说明是 MMC 卡，额外进行相应初始化
    retry = 0;
    do
    {
        //先发 CMD55，应返回 0x01；否则出错
        r1 = SD_SendCommand(CMD55, 0, 0);
        if(r1 == 0XFF)return r1;//只要不是 0xff,就接着发送
        //得到正确响应后，发 ACMD41，应得到返回值 0x00，否则重试 200 次
        r1 = SD_SendCommand(ACMD41, 0, 0);
        retry++;
    }while((r1!=0x00) && (retry<400));
    // 判断是超时还是得到正确回应
    // 若有回应：是 SD 卡；没有回应：是 MMC 卡
    //----------MMC 卡额外初始化操作开始------------
    if(retry==400)
    {
        retry = 0;
        //发送 MMC 卡初始化命令（没有测试）
        do
        {
            r1 = SD_SendCommand(1,0,0);
            retry++;
        }while((r1!=0x00)&& (retry<400));
        if(retry==400)return 1;   //MMC 卡初始化超时
        //写入卡类型
        SD_Type = SD_TYPE_MMC;
    }
    //----------MMC 卡额外初始化操作结束------------
    //设置 SPI 为高速模式
    SPIx_SetSpeed(SPI_SPEED_4);
```

```
        SPIx_ReadWriteByte(0xFF);
        //禁止 CRC 校验
        r1 = SD_SendCommand(CMD59, 0, 0x95);
        if(r1 != 0x00)return r1;  //命令错误，返回 r1
        //设置 Sector Size
        r1 = SD_SendCommand(CMD16, 512, 0x95);
        if(r1 != 0x00)return r1;//命令错误，返回 r1
        //----------------SD 卡、MMC 卡初始化结束----------------

    }//SD 卡为 V1.0 版本的初始化结束
    //下面是 V2.0 卡的初始化
    //其中需要读取 OCR 数据，判断是 SD2.0 还是 SD2.0HC 卡
    else if(r1 == 0x01)
    {
        //V2.0 的卡，CMD8 命令后会传回 4 字节的数据，要跳过再结束本命令
        buff[0] = SPIx_ReadWriteByte(0xFF);  //should be 0x00
        buff[1] = SPIx_ReadWriteByte(0xFF);  //should be 0x00
        buff[2] = SPIx_ReadWriteByte(0xFF);  //should be 0x01
        buff[3] = SPIx_ReadWriteByte(0xFF);  //should be 0xAA
        SD_CS=1;
        SPIx_ReadWriteByte(0xFF);//the next 8 clocks
        //判断该卡是否支持 2.7V-3.6V 的电压范围
        //if(buff[2]==0x01 && buff[3]==0xAA) //不判断，让其支持的卡更多
        {
            retry = 0;
            //发卡初始化指令 CMD55+ACMD41
            do
            {
                r1 = SD_SendCommand(CMD55, 0, 0);
                if(r1!=0x01)return r1;
                r1 = SD_SendCommand(ACMD41, 0x40000000, 0);
                if(retry>200)return r1;  //超时则返回 r1 状态
            }while(r1!=0);
            //初始化指令发送完成，接下来获取 OCR 信息
            //----------鉴别 SD2.0 卡版本开始----------
            r1 = SD_SendCommand_NoDeassert(CMD58, 0, 0);
            if(r1!=0x00)
            {
                SD_CS=1;//释放 SD 片选信号
                return r1;  //如果命令没有返回正确应答，直接退出，返回应答

            }//读 OCR 指令发出后，紧接着是 4 字节的 OCR 信息
            buff[0] = SPIx_ReadWriteByte(0xFF);
```

```
            buff[1] = SPIx_ReadWriteByte(0xFF);
            buff[2] = SPIx_ReadWriteByte(0xFF);
            buff[3] = SPIx_ReadWriteByte(0xFF);
            //OCR 接收完成，片选置高
            SD_CS=1;
            SPIx_ReadWriteByte(0xFF);
            //检查接收到的 OCR 中的 bit30 位（CCS），确定其为 SD2.0 还是 SDHC
            //如果 CCS=1：SDHC    CCS=0：SD2.0
            if(buff[0]&0x40)SD_Type = SD_TYPE_V2HC;    //检查 CCS
            else SD_Type = SD_TYPE_V2;
            //-----------鉴别 SD2.0 卡版本结束-----------
            //设置 SPI 为高速模式
            SPIx_SetSpeed(SPI_SPEED_4);
        }
    }
    return r1;
}

//从 SD 卡中读回指定长度的数据，放置在给定位置
//输入: u8 *data(存放读回数据的内存>len)
//      u16 len(数据长度）
//      u8 release(传输完成后是否释放总线 CS 置高 0：不释放 1：释放）
//返回值:0：NO_ERR
//       other：错误信息

u8 SD_ReceiveData(u8 *data, u16 len, u8 release)
{
    // 启动一次传输
    SD_CS=0;
    if(SD_GetResponse(0xFE))//等待 SD 卡发回数据起始令牌 0xFE
    {
        SD_CS=1;
        return 1;
    }
    while(len--)//开始接收数据
    {
        *data=SPIx_ReadWriteByte(0xFF);
        data++;
    }
    //下面是 2 个伪 CRC（dummy CRC）
    SPIx_ReadWriteByte(0xFF);
    SPIx_ReadWriteByte(0xFF);
    if(release==RELEASE)//按需释放总线，将 CS 置高
```

```
    {
        SD_CS=1;//传输结束
        SPIx_ReadWriteByte(0xFF);
    }
    return 0;
}

//获取 SD 卡的 CID 信息，包括制造商信息
//输入: u8 *cid_data(存放 CID 的内存，至少 16Byte）
//返回值:0：NO_ERR
//       1：TIME_OUT
//       other：错误信息

u8 SD_GetCID(u8 *cid_data)
{
    u8 r1;
    //发 CMD10 命令，读 CID
    r1 = SD_SendCommand(CMD10,0,0xFF);
    if(r1 != 0x00)return r1;  //没返回正确应答，则退出，报错
    SD_ReceiveData(cid_data,16,RELEASE);//接收 16 个字节的数据
    return 0;
}

//获取 SD 卡的 CSD 信息，包括容量和速度信息
//输入:u8 *cid_data(存放 CID 的内存，至少 16Byte）
//返回值:0：NO_ERR
//       1：TIME_OUT
//       other：错误信息

u8 SD_GetCSD(u8 *csd_data)
{
    u8 r1;
    r1=SD_SendCommand(CMD9,0,0xFF);//发 CMD9 命令，读 CSD
    if(r1)return r1;  //没返回正确应答，则退出，报错
    SD_ReceiveData(csd_data, 16, RELEASE);//接收 16 个字节的数据
    return 0;
}
//获取 SD 卡的容量（字节）
//返回值:0：  取容量出错
//       其他:SD 卡的容量(字节)
u32 SD_GetCapacity(void)
{
    u8 csd[16];
```

```
u32 Capacity;
u8 r1;
u16 i;
u16 temp;
//取 CSD 信息，如果期间出错，返回 0
if(SD_GetCSD(csd)!=0) return 0;
//如果为 SDHC 卡，按照下面方式计算
if((csd[0]&0xC0)==0x40)
{
    Capacity=((u32)csd[8])<<8;
    Capacity+=(u32)csd[9]+1;
    Capacity = (Capacity)*1024;//得到扇区数
    Capacity*=512;//得到字节数
}
else
{
    i = csd[6]&0x03;
    i<<=8;
    i += csd[7];
    i<<=2;
    i += ((csd[8]&0xc0)>>6);
    //C_SIZE_MULT
    r1 = csd[9]&0x03;
    r1<<=1;
    r1 += ((csd[10]&0x80)>>7);
    r1+=2;//BLOCKNR
    temp = 1;
    while(r1)
    {
        temp*=2;
        r1--;
    }
    Capacity = ((u32)(i+1))*((u32)temp);
    // READ_BL_LEN
    i = csd[5]&0x0f;
    //BLOCK_LEN
    temp = 1;
    while(i)
    {
        temp*=2;
        i--;
    }
    //The final result
```

```
        Capacity *= (u32)temp;//字节为单位
    }
    return (u32)Capacity;
}

//读 SD 卡的一个 block
//输入:u32 sector 取地址（sector 值，非物理地址）
//     u8 *buffer 数据存储地址（大小至少 512byte）
//返回值:0： 成功
//       other：失败

u8 SD_ReadSingleBlock(u32 sector, u8 *buffer)
{
    u8 r1;
    //设置为高速模式
    SPIx_SetSpeed(SPI_SPEED_4);
    //如果不是 SDHC，给定的是 sector 地址，将其转换成 byte 地址
    if(SD_Type!=SD_TYPE_V2HC)
    {
        sector = sector<<9;
    }
    r1 = SD_SendCommand(CMD17, sector, 0);//读命令
    if(r1 != 0x00)return r1;
    r1 = SD_ReceiveData(buffer, 512, RELEASE);
    if(r1 != 0)return r1;   //读数据出错!
    else return 0;
}
/////////////////下面 2 个函数为 USB 读写所需要的//////////////////////////
//定义 SD 卡的块大小
#define BLOCK_SIZE 512
//写入 MSD/SD 数据
//pBuffer:数据存放区
//ReadAddr:写入的首地址
//NumByteToRead:要写入的字节数
//返回值:0,写入完成
//    其他,写入失败
u8 MSD_WriteBuffer(u8* pBuffer, u32 WriteAddr, u32 NumByteToWrite)
{
    u32 i,NbrOfBlock = 0, Offset = 0;
    u32 sector;
    u8 r1;
    NbrOfBlock = NumByteToWrite / BLOCK_SIZE;//得到要写入的块的数目
    SD_CS=0;
```

```
    while (NbrOfBlock--)//写入一个扇区
    {
        sector=WriteAddr+Offset;
        if(SD_Type==SD_TYPE_V2HC)sector>>=9;//执行与普通操作相反的操作

        r1=SD_SendCommand_NoDeassert(CMD24,sector,0xff);//写命令
        if(r1)
        {
            SD_CS=1;
            return 1;//应答不正确，直接返回
        }
        SPIx_ReadWriteByte(0xFE);//放起始令牌 0xFE
        //放一个 sector 的数据
        for(i=0;i<512;i++)SPIx_ReadWriteByte(*pBuffer++);
        //发 2 个 Byte 的 dummy CRC
        SPIx_ReadWriteByte(0xff);
        SPIx_ReadWriteByte(0xff);
        if(SD_WaitDataReady())//等待 SD 卡数据写入完成
        {
            SD_CS=1;
            return 2;
        }
        Offset += 512;
    }
    //写入完成，片选置 1
    SD_CS=1;
    SPIx_ReadWriteByte(0xff);
    return 0;
}
//读取 MSD/SD 数据
//pBuffer:数据存放区
//ReadAddr:读取的首地址
//NumByteToRead:要读出的字节数
//返回值:0,读出完成
//    其他,读出失败
u8 MSD_ReadBuffer(u8* pBuffer, u32 ReadAddr, u32 NumByteToRead)
{
    u32 NbrOfBlock=0,Offset=0;
    u32 sector=0;
    u8 r1=0;
    NbrOfBlock=NumByteToRead/BLOCK_SIZE;
    SD_CS=0;
    while (NbrOfBlock --)
```

```
        {
            sector=ReadAddr+Offset;
            if(SD_Type==SD_TYPE_V2HC)sector>>=9;//执行与普通操作相反的操作

            r1=SD_SendCommand_NoDeassert(CMD17,sector,0xff);//读命令

            if(r1)//命令发送错误
            {
                SD_CS=1;
                return r1;
            }
            r1=SD_ReceiveData(pBuffer,512,RELEASE);
            if(r1)//读数错误
            {
                SD_CS=1;
                return r1;
            }
            pBuffer+=512;
            Offset+=512;
        }
        SD_CS=1;
        SPIx_ReadWriteByte(0xff);
        return 0;
}
//////////////////////////////////////////////////////////////////////////////
//写入SD卡的一个block(未实际测试过)

//输入:u32 sector 扇区地址（sector值，非物理地址）
//     u8 *buffer 数据存储地址（大小至少512byte）
//返回值:0： 成功
//       other：失败

u8 SD_WriteSingleBlock(u32 sector, const u8 *data)
{
    u8 r1;
    u16 i;
    u16 retry;

    //设置为高速模式
    //SPIx_SetSpeed(SPI_SPEED_HIGH);
    //如果不是SDHC，给定的是sector地址，将其转换成byte地址
    if(SD_Type!=SD_TYPE_V2HC)
    {
```

```
    sector = sector<<9;
}
r1 = SD_SendCommand(CMD24, sector, 0x00);
if(r1 != 0x00)
{
    return r1;  //应答不正确，直接返回
}

//开始准备数据传输
SD_CS=0;
//先放 3 个空数据，等待 SD 卡准备好
SPIx_ReadWriteByte(0xff);
SPIx_ReadWriteByte(0xff);
SPIx_ReadWriteByte(0xff);
//放起始令牌 0xFE
SPIx_ReadWriteByte(0xFE);

//放一个 sector 的数据
for(i=0;i<512;i++)
{
    SPIx_ReadWriteByte(*data++);
}
//发 2 个 Byte 的 dummy CRC
SPIx_ReadWriteByte(0xff);
SPIx_ReadWriteByte(0xff);

//等待 SD 卡应答
r1 = SPIx_ReadWriteByte(0xff);
if((r1&0x1F)!=0x05)
{
    SD_CS=1;
    return r1;
}

//等待操作完成
retry = 0;
while(!SPIx_ReadWriteByte(0xff))
{
    retry++;
    if(retry>0xfffe)        //如果长时间写入没有完成，报错退出
    {
        SD_CS=1;
        return 1;           //写入超时返回 1
```

```
        }
    }
    //写入完成，片选置 1
    SD_CS=1;
    SPIx_ReadWriteByte(0xff);

    return 0;
}
//读 SD 卡的多个 block(实际测试过)
//输入:u32 sector 扇区地址（sector 值，非物理地址）
//     u8 *buffer 数据存储地址（大小至少 512byte）
//     u8 count 连续读 count 个 block
//返回值:0： 成功
//       other：失败

u8 SD_ReadMultiBlock(u32 sector, u8 *buffer, u8 count)
{
    u8 r1;
    //SPIx_SetSpeed(SPI_SPEED_HIGH);//设置为高速模式
    //如果不是 SDHC，将 sector 地址转成 byte 地址
    if(SD_Type!=SD_TYPE_V2HC)sector = sector<<9;
    //SD_WaitDataReady();
    //发读多块命令
    r1 = SD_SendCommand(CMD18, sector, 0);//读命令
    if(r1 != 0x00)return r1;
    do//开始接收数据
    {
        if(SD_ReceiveData(buffer, 512, NO_RELEASE) != 0x00)break;
        buffer += 512;
    } while(--count);
    //全部传输完毕，发送停止命令
    SD_SendCommand(CMD12, 0, 0);
    //释放总线
    SD_CS=1;
    SPIx_ReadWriteByte(0xFF);
    if(count != 0)return count;   //如果没有传完，返回剩余个数
    else return 0;
}
//写入 SD 卡的 N 个 block(未实际测试过)
//输入:u32 sector 扇区地址（sector 值，非物理地址）
//     u8 *buffer 数据存储地址（大小至少 512byte）
//     u8 count 写入的 block 数目
//返回值:0： 成功
```

```
//       other：失败

u8 SD_WriteMultiBlock(u32 sector, const u8 *data, u8 count)
{
    u8 r1;
    u16 i;
    //SPIx_SetSpeed(SPI_SPEED_HIGH);//设置为高速模式
    if(SD_Type != SD_TYPE_V2HC)sector = sector<<9;//如果不是 SDHC，给定的是sector 地址，将其转换成 byte 地址
    if(SD_Type != SD_TYPE_MMC) r1 = SD_SendCommand(ACMD23, count, 0x00);//如果目标卡不是 MMC 卡，启用 ACMD23 指令使能预擦除
    r1 = SD_SendCommand(CMD25, sector, 0x00);//发多块写入指令
    if(r1 != 0x00)return r1;  //应答不正确，直接返回
    SD_CS=0;//开始准备数据传输
    SPIx_ReadWriteByte(0xff);//先放 3 个空数据，等待 SD 卡准备好
    SPIx_ReadWriteByte(0xff);
    //--------下面是 N 个 sector 写入的循环部分
    do
    {
        //放起始令牌 0xFC 表明是多块写入
        SPIx_ReadWriteByte(0xFC);
        //放一个 sector 的数据
        for(i=0;i<512;i++)
        {
            SPIx_ReadWriteByte(*data++);
        }
        //发 2 个 Byte 的 dummy CRC
        SPIx_ReadWriteByte(0xff);
        SPIx_ReadWriteByte(0xff);

        //等待 SD 卡应答
        r1 = SPIx_ReadWriteByte(0xff);
        if((r1&0x1F)!=0x05)
        {
            SD_CS=1;    //如果应答为报错，则带错误代码直接退出
            return r1;
        }
        //等待 SD 卡写入完成
        if(SD_WaitDataReady()==1)
        {
            SD_CS=1;    //等待 SD 卡写入完成超时，直接退出报错
            return 1;
        }
```

```
        }while(--count);//本 sector 数据传输完成
        //发结束传输令牌 0xFD
        r1 = SPIx_ReadWriteByte(0xFD);
        if(r1==0x00)
        {
            count =  0xfe;
        }
        if(SD_WaitDataReady()) //等待准备好
        {
            SD_CS=1;
            return 1;
        }
        //写入完成，片选置 1
        SD_CS=1;
        SPIx_ReadWriteByte(0xff);
        return count;   //返回 count 值，如果写完则 count=0，否则 count=1
    }
    //在指定扇区,从 offset 开始读出 bytes 个字节
    //输入:u32 sector 扇区地址（sector 值，非物理地址）
    //     u8 *buf     数据存储地址（大小<=512byte）
    //     u16 offset  在扇区里面的偏移量
    //     u16 bytes   要读出的字节数
    //返回值:0： 成功
    //       other：失败
    u8 SD_Read_Bytes(unsigned long address,unsigned char *buf,unsigned int
offset,unsigned int bytes)
    {
        u8 r1;u16 i=0;
        r1=SD_SendCommand(CMD17,address<<9,0);//发送读扇区命令
        if(r1)return r1;  //应答不正确，直接返回
        SD_CS=0;//选中 SD 卡
        if(SD_GetResponse(0xFE))//等待 SD 卡发回数据起始令牌 0xFE
        {
            SD_CS=1; //关闭 SD 卡
            return 1;//读取失败
        }
        for(i=0;i<offset;i++)SPIx_ReadWriteByte(0xff);//跳过 offset 位
        for(;i<offset+bytes;i++)*buf++=SPIx_ReadWriteByte(0xff);//读取有用数据

        for(;i<512;i++) SPIx_ReadWriteByte(0xff);     //读出剩余字节
        SPIx_ReadWriteByte(0xff);//发送伪 CRC 码
        SPIx_ReadWriteByte(0xff);
        SD_CS=1;//关闭 SD 卡
```

```
    return 0;
}
```

此部分代码我们在本节主要用到的就是 SD 卡初始化函数 SD_Init 和 SD 卡读取函数 SD_ReadSingleBlock，通过前面的介绍，相信大家不难理解，这里我们就不多说了，接下来保存 MMC_SD.C 文件，并加入到 HARDWARE 组下，然后打开 MMC_SD.H，在该文件里面输入如下代码：

```
#ifndef _MMC_SD_H_
#define _MMC_SD_H_
#include <stm32f10x_lib.h>
//Mini STM32开发板
//SD卡 驱动
//正点原子@ALIENTEK
//SD传输数据结束后是否释放总线宏定义
#define NO_RELEASE          0
#define RELEASE             1
// SD卡类型定义
#define SD_TYPE_MMC         0
#define SD_TYPE_V1          1
#define SD_TYPE_V2          2
#define SD_TYPE_V2HC        4
// SD卡指令表
#define CMD0       0        //卡复位
#define CMD1       1
#define CMD9       9        //命令9 ，读CSD数据
#define CMD10      10       //命令10，读CID数据
#define CMD12      12       //命令12，停止数据传输
#define CMD16      16       //命令16，设置SectorSize 应返回0x00
#define CMD17      17       //命令17，读sector
#define CMD18      18       //命令18，读Multi sector
#define ACMD23     23       //命令23，设置多sector写入前预先擦除N个block
#define CMD24      24       //命令24，写sector
#define CMD25      25       //命令25，写Multi sector
#define ACMD41     41       //命令41，应返回0x00
#define CMD55      55       //命令55，应返回0x01
#define CMD58      58       //命令58，读OCR信息
#define CMD59      59       //命令59，使能/禁止CRC，应返回0x00
//数据写入回应字意义
#define MSD_DATA_OK                0x05
#define MSD_DATA_CRC_ERROR         0x0B
#define MSD_DATA_WRITE_ERROR       0x0D
#define MSD_DATA_OTHER_ERROR       0xFF
//SD卡回应标记字
#define MSD_RESPONSE_NO_ERROR      0x00
```

```
#define MSD_IN_IDLE_STATE          0x01
#define MSD_ERASE_RESET            0x02
#define MSD_ILLEGAL_COMMAND        0x04
#define MSD_COM_CRC_ERROR          0x08
#define MSD_ERASE_SEQUENCE_ERROR   0x10
#define MSD_ADDRESS_ERROR          0x20
#define MSD_PARAMETER_ERROR        0x40
#define MSD_RESPONSE_FAILURE       0xFF

//这部分应根据具体的连线来修改!
//Mini STM32使用的是PA3作为SD卡的CS脚.
#define  SD_CS   PAout(3) //SD卡片选引脚
extern u8   SD_Type;//SD卡的类型
//函数申明区
u8 SD_WaitReady(void);                                  //等待SD卡就绪
u8 SD_SendCommand(u8 cmd, u32 arg, u8 crc);       //SD卡发送一个命令
u8 SD_SendCommand_NoDeassert(u8 cmd, u32 arg, u8 crc);
u8 SD_Init(void);                                    //SD卡初始化
u8 SD_Idle_Sta(void);                                //设置SD卡到挂起模式
u8 SD_ReceiveData(u8 *data, u16 len, u8 release);//SD卡读数据
u8 SD_GetCID(u8 *cid_data);                          //读SD卡CID
u8 SD_GetCSD(u8 *csd_data);                          //读SD卡CSD
u32 SD_GetCapacity(void);                            //取SD卡容量
//USB 读卡器 SD卡操作函数
u8 MSD_WriteBuffer(u8* pBuffer, u32 WriteAddr, u32 NumByteToWrite);
u8 MSD_ReadBuffer(u8* pBuffer, u32 ReadAddr, u32 NumByteToRead);

u8 SD_ReadSingleBlock(u32 sector, u8 *buffer);   //读一个sector
u8 SD_WriteSingleBlock(u32 sector, const u8 *buffer);          //写一个sector
u8 SD_ReadMultiBlock(u32 sector, u8 *buffer, u8 count);  //读多个sector
u8 SD_WriteMultiBlock(u32 sector, const u8 *data, u8 count);//写多个sector
u8 SD_Read_Bytes(unsigned long address,unsigned char *buf,unsigned int
offset,unsigned int bytes);//读取一byte
#endif
```

该部分代码主要是一些命令的宏定义以及函数声明，在这里我们设定了 SD 卡的 CS 管脚为 PA3。保存 MMC_SD.H，就可以在主函数里面编写我们的应用代码了，打开 test.c 文件，在该文件中修改 main 函数如下：

```
u8 buf[512];//SD 卡数据缓存区
int main(void)
{
    u32 sd_size;
    u8 t=0;
    Stm32_Clock_Init(9);//系统时钟设置
```

```
delay_init(72);      //延时初始化
uart_init(72,9600); //串口 1 初始化
LCD_Init();          //初始化液晶
LED_Init();          //LED 初始化
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"SD Card TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2011/1/1");
while(SD_Init()!=0)//检测不到 SD 卡
{
    LCD_ShowString(60,130,"SD Card Failed!");
    delay_ms(500);
    LCD_ShowString(60,130,"Please Check!      ");
    delay_ms(500);
    LED0=!LED0;//DS0 闪烁
}
//检测 SD 卡成功
LCD_ShowString(60,130,"SD Card Checked OK ");
LCD_ShowString(60,150,"SD Card Size:    Mb");
sd_size=SD_GetCapacity();
LCD_ShowNum(164,150,sd_size>>20,4,16);//显示 SD 卡容量
while(1)
{
    if(t==30)//每 6s 钟执行一次
    {
        if(SD_ReadSingleBlock(0,buf)==0)//读取 MBR 扇区
        {
            LCD_ShowString(60,170,"USART1 Sending Data...");
            printf("SECTOR 0 DATA:\n");
            for(sd_size=0;sd_size<512;sd_size++)//打印 MBR 扇区数据
            {
                printf("%x ",buf[sd_size]);
            }
            printf("\nDATA ENDED\n");
            LCD_ShowString(60,170,"USART1 Send Data Over!");
        }
        t=0;
    }
    t++;
    delay_ms(200);
    LED0=!LED0;
}
```

```
}
```

这里我们通过 SD_GetCapacity 函数来得到 SD 卡的容量，然后在液晶上显示出来，接着我们读取 SD 卡的扇区 0，然后把这部分数据通过串口打印出来。每 6s 钟执行一次。至此，我们的软件设计就结束了。

3.20.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容（默认 SD 卡已经接上了）：

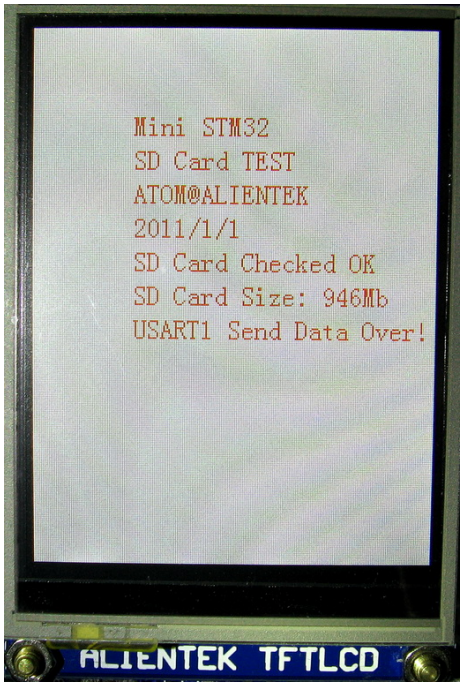


图 3.19.4.1 SD 卡实验 LCD 显示内容

此时我们打开串口调试助手，就可以看到从开发板发回来的数据了，如下图示：

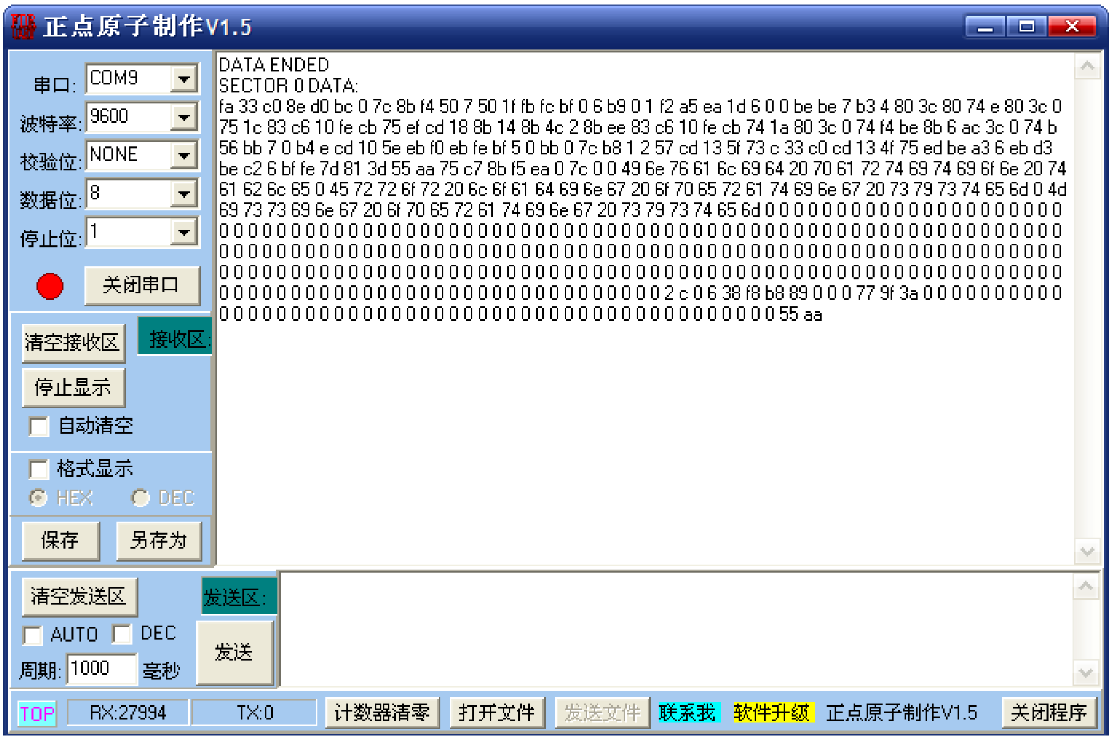


图 3.19.4.2 串口收到的 SD 卡扇区 0 内容

3.21 红外遥控实验

ALIENTKE MiniSTM3 开发板给使用者配备了标准的红外接收头和一个很小巧的红外遥控器。本节将向大家介绍，如何在 ALIENTEK MiniSTM32 开发板上实现红外遥控器的控制。本节分为如下几个部分：

3.21.1 红外遥控简介

3.21.2 硬件设计

3.21.3 软件设计

3.21.4 下载与测试

3.21.1 红外遥控简介

红外遥控是一种无线、非接触控制技术，具有抗干扰能力强，信息传输可靠，功耗低，成本低，易实现等显著优点，被诸多电子设备特别是家用电器广泛采用，并越来越多的应用到计算机系统中。

由于红外线遥控不具有像无线电遥控那样穿过障碍物去控制被控对象的能力，所以，在设计红外线遥控器时，不必要像无线电遥控器那样，每套(发射器和接收器)要有不同的遥控频率或编码(否则，就会隔墙控制或干扰邻居的家用电器)，所以同类产品的红外线遥控器，可以有相同的遥控频率或编码，而不会出现遥控信号“串门”的情况。这对于大批量生产以及在家用电器上普及红外线遥控提供了极大的方面。由于红外线为不可见光，因此对环境影响很小，再由红外光波动波长远小于无线电波的波长，所以红外线遥控不会影响其他家用电器，也不会影响临近的无线电设备。

红外遥控的编码目前广泛使用的是：NEC Protocol 的 PWM(脉冲宽度调制)和 Philips RC-5 Protocol 的 PPM(脉冲位置调制)。我们配套的遥控器使用的是 NEC 协议，其特征如下：

1、8 位地址和 8 位指令长度；

2、地址和命令 2 次传输（确保可靠性）

3、PWM 脉冲位置调制，以发射红外载波的占空比代表“0”和“1”；

4、载波频率为 38Khz；

5、位时间为 1.125ms 或 2.25ms；

NEC 码的位定义：一个脉冲对应 560us 的连续载波，一个逻辑 1 传输需要 2.25ms（560us 脉冲+1680us 低电平），一个逻辑 0 的传输需要 1.125ms（560us 脉冲+560us 低电平）。而遥控接收头在收到脉冲的时候为低电平，在没有脉冲的时候为高电平，这样，我们在接收头端收到的信号为：逻辑 1 应该是 560us 低+1680us 高，逻辑 0 应该是 560us 低+560us 高。

NEC 遥控指令的数据格式为：同步码头、地址码、地址反码、控制码、控制反码。同步码由一个 9ms 的低电平和一个 4.5ms 的高电平组成，地址码、地址反码、控制码、控制反码均是 8 位数据格式。按照低位在前，高位在后的顺序发送。采用反码是为了增加传输的可靠性（可用于校验）。

我们遥控器的按键 2 按下时，从红外接收头端收到的波形如下：

图 3.21.1.1 按键 2 所对应的红外波形

从图中可以看到，其地址码为 0，控制码为 168。可以看到在 100ms 之后，我们还受到了几个脉冲，这是 NEC 码规定的连发码(由 9ms 低电平+2.5m 高电平+0.56ms 低电平+97.94ms 高电平组成)，如果在一帧数据发送完毕之后，按键仍然没有放开，则发射重复码，即连发码，可以通过统计连发码的次数来标记按键按下的长短/次数。

3.21.2 硬件设计

本实验采用中断解码(也可以采用输入捕获解码)，本节实验功能简介：开机在 LCD 上显示一些信息之后，即进入等待红外触发，如过接收到正确的红外信号，则解码，并在 LCD 上显示键值和所代表的意义，以及按键次数等信息。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）TFTLCD 液晶模块。

4）红外接收头。

5）红外遥控器。

前面三部分，在之前的实例已经介绍过了，遥控器属于外部器件，遥控接收头在板子上，与MCU的连接原理图如下：

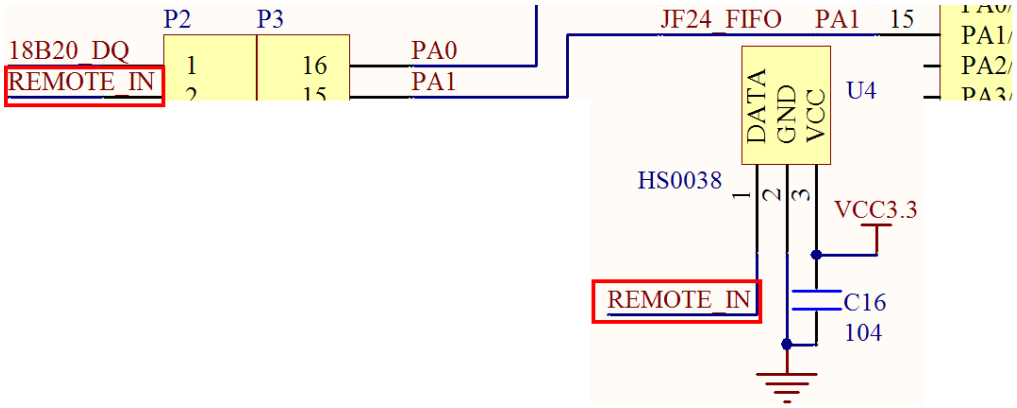


图 3.21.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头与 STM32 的 PA1 通过跳线帽连接，这个在硬件上要连接上，否则 PA1 将得不到信号，我们采用中断解码，PA1 对应的中断为中断 1，所以在程序设计的时候，我们只要开启中断 1，然后在中断里执行解码就可以了。 开发板配套的红外遥控器外观如下：

图 3.21.2.2 红外遥控器

3.21.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 REMOTE 的文件夹。然后新建一个 remote.c 和 remote.h 的文件保存在 REMOTE 文件夹下，并将这个文件夹加入头文件包含路径。

打开 remote.c 文件，输入如下代码：

```
#include "remote.h"
#include "delay.h"
//Mini STM32 开发板
//红外遥控接收 驱动函数
//正点原子@ALIENTEK
u32 Remote_Odr=0;    //命令暂存处
u8  Remote_Cnt=0;    //按键次数,此次按下键的次数
u8  Remote_Rdy=0;    //红外接收到数据
//初始化红外接收引脚的设置
//开启中断,并映射
void Remote_Init(void)
{
    RCC->APB2ENR|=1<<2;        //PA 时钟使能
    GPIOA->CRL&=0XFFFFFF0F;
    GPIOA->CRL|=0X00000080; //PA1 输入
    GPIOA->ODR|=1<<1;        //PA.1 上拉
    Ex_NVIC_Config(GPIO_A,1,FTIR);//将 line1 映射到 PA.1，下降沿触发.
    MY_NVIC_Init(2,1,EXTI1_IRQChannel,2);
}
//检测脉冲宽度
//最长脉宽为 5ms
//返回值:x,代表脉宽为 x*20us(x=1~250);
u8 Pulse_Width_Check(void)
{
    u8 t=0;
    while(RDATA)
    {
        t++;delay_us(20);
        if(t==250)return t; //超时溢出
    }
    return t;
}
//处理红外接收
/*-------------------------协议--------------------------
开始拉低 9ms,接着是一个 4.5ms 的高脉冲,通知器件开始传送数据了
```

接着是发送 4 个 8 位二进制码,第一二个是遥控识别码(REMOTE_ID),第一个为
正码(0),第二个为反码(255),接着两个数据是键值,第一个为正码
第二个为反码.发送完后 40ms,遥控再发送一个 9ms 低,2ms 高的脉冲,
表示按键的次数,出现一次则证明只按下了一次,如果出现多次,则可
以认为是持续按下该键.
----------*/

```
//外部中断服务程序
void EXTI1_IRQHandler(void)
{
    u8 res=0;
    u8 OK=0;
    u8 RODATA=0;
    while(1)
    {
        if(RDATA)//有高脉冲出现
        {
            res=Pulse_Width_Check();//获得此次高脉冲宽度
            if(res==250)break;//非有用信号
            if(res>=200&&res<250)OK=1; //获得前导位(4.5ms)
            else if(res>=85&&res<200)  //按键次数加一(2ms)
            {
                Remote_Rdy=1;//接受到数据
                Remote_Cnt++;//按键次数增加
                break;
            }
            else if(res>=50&&res<85)RODATA=1;//1.5ms
            else if(res>=10&&res<50)RODATA=0;//500us
            if(OK)
            {
                Remote_Odr<<=1;
                Remote_Odr+=RODATA;
                Remote_Cnt=0; //按键次数清零
            }
        }
    }
    EXTI->PR=1<<1;      //清除中断标志位
}
//处理红外键盘
//返回相应的键值
u8 Remote_Process(void)
{
    u8 t1,t2;
    t1=Remote_Odr>>24; //红外解码
```

```
        t2=(Remote_Odr>>16)&0xff;
        Remote_Rdy=0;//清除标记
        if(t1==(u8)~t2&&t1==REMOTE_ID)//检验遥控识别码(ID)
        {
            t1=Remote_Odr>>8;
            t2=Remote_Odr;
            if(t1==(u8)~t2)return t1; //处理键值
        }
        return 0;
    }
```

该部分代码比较简单，主要是通过中断解码，解码程序是按照前面介绍的 NEC 码来解的，在这里我们就不再多说了。保存 remote.c，然后把该文件加入 HARDWARE 组下。接下来打开 remote.h 在该文件里面加入如下代码：

```
    #ifndef __RED_H
    #define __RED_H
    #include "sys.h"
    //Mini STM32 开发板
    //红外遥控接收 驱动函数
    //正点原子@ALIENTEK
    #define RDATA PAin(1)     //红外数据输入脚
    //红外遥控识别码(ID),每款遥控器的该值基本都不一样,但也有一样的.
    //我们选用的遥控器识别码为 0
    #define REMOTE_ID 0

    extern u8 Remote_Cnt;     //按键次数,此次按下键的次数
    extern u8 Remote_Rdy;     //红外接收到数据
    extern u32 Remote_Odr;    //命令暂存处
    void Remote_Init(void);     //红外传感器接收头引脚初始化
    u8 Remote_Process(void);    //红外接收到数据处理
    u8 Pulse_Width_Check(void);//检查脉宽
    #endif
```

这里的 REMOTE_ID 就是我们开发板配套的遥控器的识别码，对于其他遥控器可能不一样，只要修改这个为你所使用的遥控器的一致就可以了。其他是一些函数的声明，我们保存此部分代码，然后在 test.c 里面修改主函数如下：

```
    int main(void)
    {
        u8 key;
        u8 t;
        Stm32_Clock_Init(9);//系统时钟设置
        delay_init(72);      //延时初始化
        uart_init(72,9600); //串口 1 初始化
        LCD_Init();          //初始化液晶
        LED_Init();          //LED 初始化
```

```
Remote_Init();      //初始化红外接收

POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"REMOTE TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2010/6/17");

LCD_ShowString(30,130, "KEYVAL:");
LCD_ShowString(130,130,"KEYCNT:");
LCD_ShowString(30,150, "SYMBOL:");
while(1)
{
    if(Remote_Rdy)
    {
        key=Remote_Process();
        LCD_ShowNum(86,130,key,3,16);//显示键值
        LCD_ShowNum(186,130,Remote_Cnt,3,16);//显示按键次数
        switch(key)
        {
            case 0:
                LCD_ShowString(86,150, "ERROR");
                break;
            case 162:
                LCD_ShowString(86,150, "POWER ");
                break;
            case 98:
                LCD_ShowString(86,150, "VOL+  ");
                break;
            case 2:
                LCD_ShowString(86,150, "VOL-  ");
                break;
            case 226:
                LCD_ShowString(86,150, "CH+   ");
                break;
            case 194:
                LCD_ShowString(86,150, "CH-   ");
                break;
            case 34:
                LCD_ShowString(86,150, "RECALL");
                break;
            case 56:
                LCD_ShowString(86,150, "0     ");
```

```
                break;
            case 224:
                LCD_ShowString(86,150, "1     ");
                break;
            case 168:
                LCD_ShowString(86,150, "2     ");
                break;
            case 144:
                LCD_ShowString(86,150, "3     ");
                break;
            case 104:
                LCD_ShowString(86,150, "4     ");
                break;
            case 152:
                LCD_ShowString(86,150, "5     ");
                break;
            case 176:
                LCD_ShowString(86,150, "6     ");
                break;
            case 48:
                LCD_ShowString(86,150, "7     ");
                break;
            case 24:
                LCD_ShowString(86,150, "8     ");
                break;
            case 122:
                LCD_ShowString(86,150, "9     ");
                break;
            case 16:
                LCD_ShowString(86,150, "ZOOM  ");
                break;
            case 90:
                LCD_ShowString(86,150, "STOP  ");
                break;
            case 66:
                LCD_ShowString(86,150, "REC   ");
                break;
            case 82:
                LCD_ShowString(86,150, "TSHIFT");
                break;
        }
    }else delay_ms(2);
    t++;
```

```
            if(t==200)
            {
                t=0;
                LED0=!LED0;
            }
        }
    }
```

至此，我们的软件设计部分就结束了。

3.21.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容：

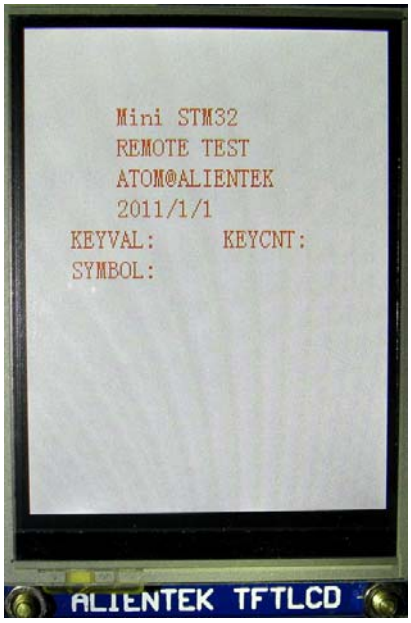


图 3.21.4.1 开机后 LCD 显示内容

此时我们通过遥控器按下不同的按键，则可以看到 LCD 上显示了不同按键的键值以及按键次数和对应的遥控器上的符号。如下图所示：

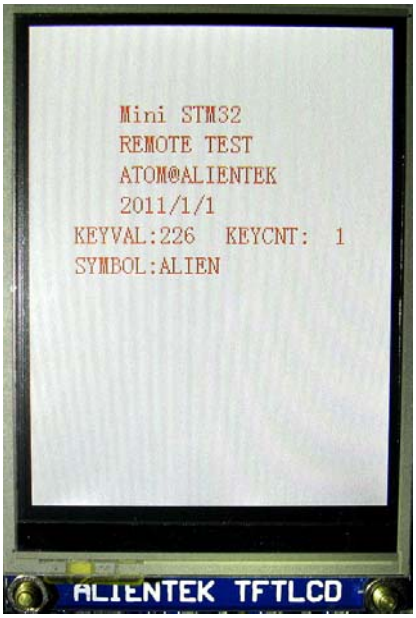

Mini STM32
REMOTE TEST
ATOM@ALIENTEK
2011/1/1
KEYVAL:226 KEYCNT: 1
SYMBOL:ALIEN
ALIENTEK TFTLCD

图 3.21.4.2 遥控接收

3.22 DS18B20 实验

STM32 虽然内部自带了温度传感器，但是因为芯片温升较大等问题，与实际温度差别较大，所以 ALIENTEK MiniSTM32 开发板还预留了目前最常用的数字温度传感器：DS18B20 的接口，用户只需要自己找一个 DS18B20 焊接上去，就可以使用了。本节将向大家介绍，如何在 ALIENTEK MiniSTM32 开发板上，通过 DS18B20 来读取环境温度值。本节分为如下几个部分：

3.22.1 DS18B20 简介

3.22.2 硬件设计

3.22.3 软件设计

3.22.4 下载与测试

3.22.1 DS18B20 简介

DS18B20 是由 DALLAS 半导体公司推出的一种的“一线总线”接口的温度传感器。与传统的热敏电阻等测温元件相比，它是一种新型的体积小、适用电压宽、与微处理器接口简单的数字化温度传感器。一线总线结构具有简洁且经济的特点，可使用户轻松地组建传感器网络，从而为测量系统的构建引入全新概念，测量温度范围为-55~+125℃ ，精度为±0．5℃。现场温度直接以“一线总线”的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~l2 位的数字值读数方式。它工作在 3—5．5 V 的电压范围，采用多种封装形式，从而使系统设计灵活、方便，设定分辨率及用户设定的报警温度存储在 EEPROM 中，掉电后依然保存。其内部结构见下图：

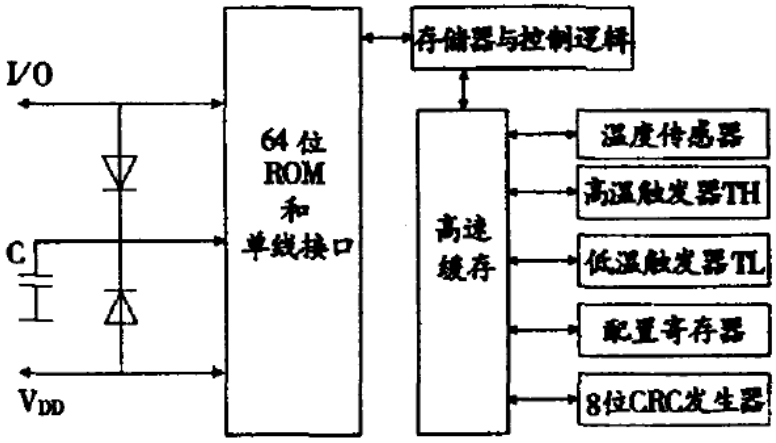


图 3.22.1.1 DS18B20 内部结构图

ROM 中的 64 位序列号是出厂前被光记好的，它可以看作是该 DS18B20 的地址序列码，每 DS18B20 的 64 位序列号均不相同。64 位 ROM 的排列是：前 8 位是产品家族码，接着 48 位是 DS18B20 的序列号，最后 8 位是前面 56 位的循环冗余校验码(CRC=X8+X5 +X4 +1)。ROM 作用是使每一个 DS18B20 都各不相同，这样就可实现一根总线上挂接多个。

所有的单总线器件要求采用严格的信号时序，以保证数据的完整性。DS18B20 共有 6 种信号类型：复位脉冲、应答脉冲、写 0、写 1、读 0 和读 1。所有这些信号，除了应答脉冲以外，都由主机发出同步信号。并且发送所有的命令和数据都是字节的低位在前。这里我们简单介绍这几个信号的时序：

1）复位脉冲和应答脉冲

单总线上的所有通信都是以初始化序列开始。主机输出低电平，保持低电平时间至少 480 us,，以产生复位脉冲。接着主机释放总线，4.7K 的上拉电阻将单总线拉高，延时 15～60 us，并进入接收模式(Rx)。接着 DS18B20 拉低总线 60~240 us，以产生低电平应答脉冲，

若为低电平，再延时 480 us。

2）写时序

写时序包括写 0 时序和写 1 时序。所有写时序至少需要 60us，且在 2 次独立的写时序之间至少需要 1us 的恢复时间，两种写时序均起始于主机拉低总线。写 1 时序：主机输出低电平，延时 2us，然后释放总线，延时 60us。写 0 时序：主机输出低电平，延时 60us，然后释放总线，延时 2us。

3）读时序

单总线器件仅在主机发出读时序时，才向主机传输数据，所以，在主机发出读数据命令后，必须马上产生读时序，以便从机能够传输数据。所有读时序至少需要 60us，且在 2 次独立的读时序之间至少需要 1us 的恢复时间。每个读时序都由主机发起，至少拉低总线 1us。主机在读时序期间必须释放总线，并且在时序起始后的 15us 之内采样总线状态。典型的读时序过程为：主机输出低电平延时 2us，然后主机转入输入模式延时 12us，然后读取单总线当前的电平，然后延时 50us。

在了解了单总线时序之后，我们来看看 DS18B20 的典型温度读取过程，DS18B20 的典型温度读取过程为：复位→发 SKIP ROM 命令（0XCC）→发开始转换命令（0X44）→延时→复位→发送 SKIP ROM 命令（0XCC）→发读存储器命令（0XBE）→连续读出两个字节数据(即温度)->结束。

DS18B20 的介绍就到这里，更详细的介绍，请大家参考 DS18B20 的技术手册。

3.22.2 硬件设计

由于开发板上标准配置是没有 DS18B20 这个传感器的，只有接口，所以大家需要自己焊接一个 DS18B20 上去。

本节实验功能简介：开机的时候先检测是否有 DS18B20 存在，如果没有，则提示错误。只有在检测到 DS18B20 之后才开始读取温度并显示在 LCD 上，如果发现了 DS18B20，则程序每隔 200ms 左右读取一次数据，并把温度显示在 LCD 上。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）TFTLCD 液晶模块。

4）DS18B20 温度传感器。

前三部分，在之前的实例已经介绍过了，DS18B20与STM32的连接电路则很简单，入下图所示：

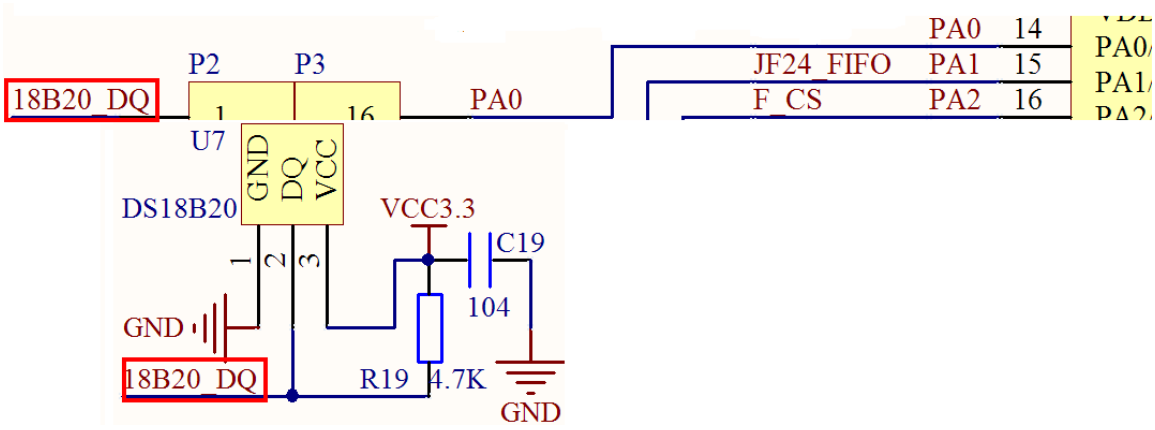


图 3.22.2.1 DS18B20 与 STM32 的连接电路图

DS18B20 与 STM32 的 PA0 通过跳线帽连接，这个在硬件上要连接上，否则 PA0 将得不到信号。

3.22.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 DS18B20 的文件夹。然后新建一个 ds18b20.c 和 ds18b20.h 的文件保存在 DS18B20 文件夹下，并将这个文件夹加入头文件包含路径。

打开 ds18b20.c 在该文件下输入如下代码：

```
#include "ds18b20.h"
#include "delay.h"
//Mini STM32 开发板
//DS18B20  驱动函数
//正点原子@ALIENTEK
//复位 DS18B20
void DS18B20_Rst(void)
{
    DS18B20_IO_OUT(); //SET PA0 OUTPUT
    DS18B20_DQ_OUT=0; //拉低 DQ
    delay_us(750);    //拉低 750us
    DS18B20_DQ_OUT=1; //DQ=1
    delay_us(15);     //15US
}
//等待 DS18B20 的回应
//返回 1:未检测到 DS18B20 的存在
//返回 0:存在
u8 DS18B20_Check(void)
{
    u8 retry=0;
    DS18B20_IO_IN();//SET PA0 INPUT
    while (DS18B20_DQ_IN&&retry<200)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=200)return 1;
    else retry=0;
    while (!DS18B20_DQ_IN&&retry<240)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=240)return 1;
    return 0;
}
//从 DS18B20 读取一个位
```

```
//返回值：1/0
u8 DS18B20_Read_Bit(void)                    // read one bit
{
    u8 data;
    DS18B20_IO_OUT();//SET PA0 OUTPUT
    DS18B20_DQ_OUT=0;
    delay_us(2);
    DS18B20_DQ_OUT=1;
    DS18B20_IO_IN();//SET PA0 INPUT
    delay_us(12);
    if(DS18B20_DQ_IN)data=1;
    else data=0;
    delay_us(50);
    return data;
}
//从 DS18B20 读取一个字节
//返回值：读到的数据
u8 DS18B20_Read_Byte(void)    // read one byte
{
    u8 i,j,dat;
    dat=0;
    for (i=1;i<=8;i++)
    {
        j=DS18B20_Read_Bit();
        dat=(j<<7)|(dat>>1);
    }
    return dat;
}
//写一个字节到 DS18B20
//dat：要写入的字节
void DS18B20_Write_Byte(u8 dat)
 {
    u8 j;
    u8 testb;
    DS18B20_IO_OUT();//SET PA0 OUTPUT;
    for (j=1;j<=8;j++)
    {
        testb=dat&0x01;
        dat=dat>>1;
        if (testb)
        {
            DS18B20_DQ_OUT=0;// Write 1
            delay_us(2);
```

```
            DS18B20_DQ_OUT=1;
            delay_us(60);
        }
        else
        {
            DS18B20_DQ_OUT=0;// Write 0
            delay_us(60);
            DS18B20_DQ_OUT=1;
            delay_us(2);
        }
    }
}
//开始温度转换
void DS18B20_Start(void)// ds1820 start convert
{
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc);// skip rom
    DS18B20_Write_Byte(0x44);// convert
}
//初始化 DS18B20 的 IO 口 DQ 同时检测 DS 的存在
//返回 1:不存在
//返回 0:存在
u8 DS18B20_Init(void)
{
    RCC->APB2ENR|=1<<2;     //使能 PORTA 口时钟
    GPIOA->CRL&=0XFFFFFFF0;//PORTA.0 推挽输出
    GPIOA->CRL|=0X00000003;
    GPIOA->ODR|=1<<0;       //输出 1
    DS18B20_Rst();
    return DS18B20_Check();
}
//从 ds18b20 得到温度值
//精度：0.1C
//返回值：温度值 （-550~1250）
short DS18B20_Get_Temp(void)
{
    u8 temp;
    u8 TL,TH;
    short tem;
    DS18B20_Start ();                       // ds1820 start convert
    DS18B20_Rst();
    DS18B20_Check();
```

```
    DS18B20_Write_Byte(0xcc);// skip rom
    DS18B20_Write_Byte(0xbe);// convert
    TL=DS18B20_Read_Byte(); // LSB
    TH=DS18B20_Read_Byte(); // MSB

    if(TH>7)
    {
        TH=~TH;
        TL=~TL;
        temp=0;//温度为负
    }else temp=1;//温度为正
    tem=TH; //获得高八位
    tem<<=8;
    tem+=TL;//获得底八位
    tem=(float)tem*0.625;//转换
    if(temp)return tem; //返回温度值
    else return -tem;
}
```

该部分代码就是根据我们前面介绍的单总线操作时序来读取 DS18B20 的温度值的，DS18B20 的温度通过 DS18B20_Get_Temp 函数读取，该函数的返回值为带符号的短整形数据，返回值的范围为-550~1250，其实就是温度值扩大了 10 倍。保存 ds18b20.c，并把该文件加入到 HARDWARE 组下，然后我们打开 ds18b20.h，在该文件下输入如下内容：

```
#ifndef __DS18B20_H
#define __DS18B20_H
#include "sys.h"
//Mini STM32 开发板
//DS18B20 驱动函数
//正点原子@ALIENTEK
//IO 方向设置
#define DS18B20_IO_IN()  {GPIOA->CRL&=0XFFFFFFF0;GPIOA->CRL|=8<<0;}
#define DS18B20_IO_OUT() {GPIOA->CRL&=0XFFFFFFF0;GPIOA->CRL|=3<<0;}
////IO 操作函数
#define DS18B20_DQ_OUT PAout(0) //数据端口  PA0
#define DS18B20_DQ_IN  PAin(0)  //数据端口  PA0

u8 DS18B20_Init(void);//初始化 DS18B20
short DS18B20_Get_Temp(void);//获取温度
void DS18B20_Start(void);//开始温度转换
void DS18B20_Write_Byte(u8 dat);//写入一个字节
u8 DS18B20_Read_Byte(void);//读出一个字节
u8 DS18B20_Read_Bit(void);//读出一个位
u8 DS18B20_Check(void);//检测是否存在 DS18B20
void DS18B20_Rst(void);//复位 DS18B20
```

```
#endif
```

关于这段代码，我们就不做多的解释了，接下来我们先保存这段代码，然后打开 test.c，在该文件下修改 main 函数如下：

```
int main(void)
{
    short temp;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);     //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LCD_Init();         //初始化液晶
    LED_Init();         //LED 初始化
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"DS18B20 TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2011/1/1");
    while(DS18B20_Init())//初始化 DS18B20,兼检测 18B20
    {
        LCD_ShowString(60,130,"DS18B20 Check Failed!");
        delay_ms(500);
        LCD_ShowString(60,130,"Please Check!        ");
        delay_ms(500);
        LED0=!LED0;//DS0 闪烁
    }
    LCD_ShowString(60,130,"DS18B20 Ready!     ");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(60,150, "Temperate:   . C");
    while(1)
    {
        temp=DS18B20_Get_Temp();
        if(temp<0)
        {
            temp=-temp;
            LCD_ShowChar(140,150,'-',16,0);//显示负号
        }
        LCD_ShowNum(148,150,temp/10,2,16);//显示温度值
        LCD_ShowNum(172,150,temp%10,1,16);//显示温度值
        //printf("t1:%d\n",temp);
        delay_ms(200);
        LED0=!LED0;
    }
}
```

至此，我们本节的软件设计就结束了。

3.22.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示开始显示当前的温度值（假定 DS18B20 已经焊接上去了），如下图所示：

图 3.22.4.1 DS18B20 读取到的温度值

该程序还可以读取并显示负温度值的，只是由于本人在广州，是没办法看到了（除非放到冰箱），具备条件的读者可以测试一下。

3.23 PS2 鼠标实验

PS/2 作为电脑的标准输入接口，用于鼠标键盘等设备。PS/2 只需要一个简单的接口（2 个 IO 口），就可以外扩鼠标、键盘等，是单片机理想的输入外扩方式。

MiniSTM32 也自带了 PS/2 接口，可以用来驱动标准的鼠标、键盘等外设，也可以用来驱动一些 PS/2 接口的小键盘。本节将向大家介绍，如何在 ALIENTEK MiniSTM32 开发板上，通过 PS/2 接口来驱动电脑鼠标。本节分为如下几个部分：

3.23.1 PS/2 简介

3.23.2 硬件设计

3.23.3 软件设计

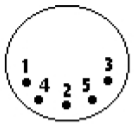
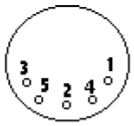
3.23.4 下载与测试

3.23.1 PS/2 简介

PS/2 是电脑上常见的接口之一，用于鼠标、键盘等设备。一般情况下，PS/2 接口的鼠标为绿色，键盘为紫色。

PS/2 接口是输入装置接口，而不是传输接口。所以 PS2 口根本没有传输速率的概念，只有扫描速率。在 Windows 环境下，ps/2 鼠标的采样率默认为 60 次/秒，USB 鼠标的采样率为 120 次/秒。较高的采样率理论上可以提高鼠标的移动精度。

物理上的 PS/2 端口可有 2 种，一种是 5 脚的，一种是六脚的。下面给出这两种 PS/2 接口的引脚定义图：

Male 公的	Female 母的	5-pin DIN (AT/XT):	5 脚 DIN(AT/XT)
		1 - Clock	1－时钟
		2 - Data	2－数据
		3 - Not Implemented	3－未实现，保留
		4 - Ground	4－电源地
(Plug) 插头	(Socket) 插座	5 - +5v	5－电源+5V

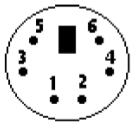
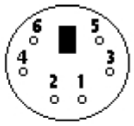
Male 公的	Female 母的	6-pin Mini-DIN (PS/2):	6 脚 Mini-DIN(PS/2)
		1 - Data	1－数据
		2 - Not Implemented	2－未实现，保留
		3 - Ground	3－电源地
		4 - +5v	4－电源+5V
(Plug) 插头	(Socket) 插座	5 - Clock	5－时钟
		6 - Not Implemented	6－未实现，保留

图 3.23.1.1 PS/2 引脚定义图

从图中可以看出，不管是 5 脚还是 6 脚的 PS/2 接头，都是有 4 跟有用的线连接：时钟线、数据线、电源线、地线。PS/2 设备的电源是 5V 的，而数据线和时钟线均是集电极开路的，这两根信号线都需要接一个上拉电阻（开发板上使用的是 10K）。

PS/2 鼠标和键盘遵循一种双向同步串行协议，换句话说每次数据线上发送一位数据并且每在时钟线上发一个脉冲就被读入。键盘/鼠标可以发送数据到主机，而主机也可以发送数据到设备，但主机总是在总线上有优先权，它可以在任何时候抑制来自于键盘/鼠标的通讯，只要把时钟拉低即可。

从设备到主机的数据在时钟信号的下降沿被主机读取，而从主机到设备的数据在时钟信号的上升沿被设备读取。不论通信方向如何，时钟总是由设备产生的，最大的时钟频率为 33Khz，大多数设备工作在 10~20Khz。

鼠标键盘，采用的是一种每帧包含 11/12 位的串行协议，这些位的含义如下：

bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
应答位	停止位	校验位	8位数据,低位在前								起始位
仅在主机到设备通信中存在	总为1	奇校验	MSB							LSB	总为0

图 3.23.1.2 鼠标/键盘帧数据格式

上图中校验位的含义是：如果数据位中包含偶数个 1，则校验位为 1；如果数据位中包含奇数个 1，则校验位为 0。数据位中的 1 的个数加上校验位总为奇数（奇校验），用于数据侦错。

当主机发送数据给键盘/鼠标的时候，设备会发送一个握手信号来应答数据已经被收到了，该位不会出现在设备到主机的通信中。

设备到主机的通信过程：

正常情况下数据线和时钟线都是高电平，当键盘/鼠标有数据要发送时，它先检测时钟线，确认时钟线是高电平。如果不是，则是主机抑制了通信，设备必须缓冲任何要发送的数据，直到重新获得总线的控制权（键盘有 16 字节的缓冲区而鼠标的缓冲区仅存储最后一个要发送的数据包）。如果时钟线是高电平，设备就可以开始传送数据了。

设备到主机的数据在时钟线的下降沿被主机读入，如下图所示：

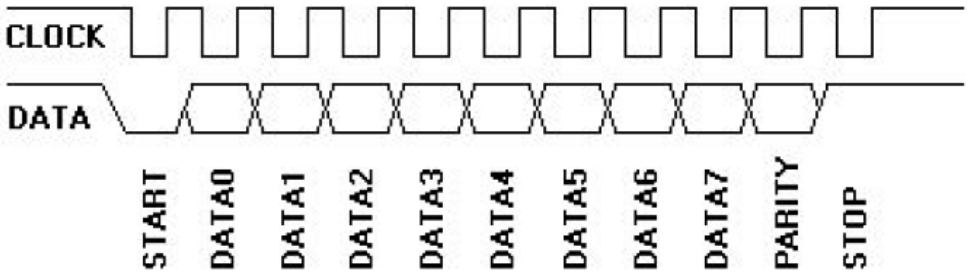


图 3.23.1.3 设备到主机通信时序图

主机可以在设备发送数据的时候拉低时钟线来来放弃当前数据的传送。

主机到设备的通信过程：

主机到设备的通信与设备到主机的通信有点不同，因为 PS/2 的时钟总是由设备产生的，如果主机要发送数据，则它必须首先把时钟线和数据线设置为请求发送状态。请求发送状态通过如下过程实现：

1. 拉低时钟线至少 100us 以抑制通信。

2. 拉低数据线，以应用“请求发送”，然后释放时钟线。

设备在不超过 10ms 的时间内就会检测这个状态，当设备检测到这个状态后，它将开始产生时钟信号，并且在设备提供的时钟脉冲驱动下输入八个数据位和一个停止位。主机仅当时钟线为低的时候改变数据线，而数据在时钟脉冲的上升沿被锁存，这与发生在设备到主机通讯的过程中正好相反。

主机到设备的通信时序图如下图所示：

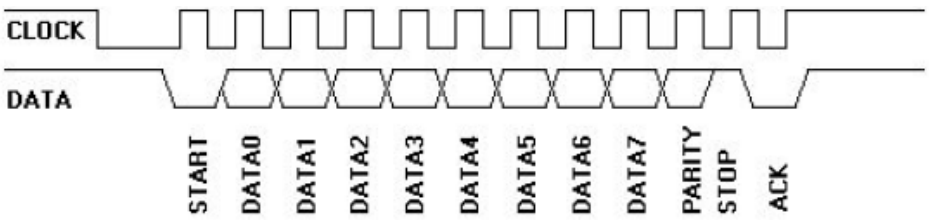


图 3.23.1.4 主机到设备通信时序图

以上简单介绍了 PS/2 协议的通信过程，更多的介绍请参考《PS/2 技术参考》一文。本节我们要驱动一个 PS/2 鼠标，所以接下来简单介绍一下 PS/2 鼠标的相关信息。

标准的 PS/2 鼠标支持下面的输入：X（左右）位移、Y（上下）位移、左键、中键和右键。但是我们目前用到鼠标大都还有滚轮，有的还有更多的按键，这就是所谓的 Intellimouse。它支持 5 个鼠标按键和三个位移轴（左右、上下和滚轮）。

标准的鼠标有两个计数器保持位移的跟踪：X 位移计数器和 Y 位移计数器。可存放 9 位的 2 进制补码，并且每个计数器都有相关的溢出标志。它们的内容连同三个鼠标按钮的状态一起以三字节移动数据包的形式发送给主机，位移计数器表示从最后一次位移数据包被送往主机

后所发生的位移量。

标准 PS/2 鼠标发送唯一和按键信息以 3 字节的数据包格式发给主机，三个数据包的意义如下图：

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X Movement							
Byte 3	Y Movement							

图 3.23.1.5 标准鼠标位移数据包格式

位移计数器是一个 9 位 2 的补码整数，其最高位作为符号位出现在位移数据包的第一个字节里。这些计数器在鼠标读取输入发现有位移时被更新。这些值是自从最后一次发送位移数据包给主机后位移的累计量（即最后一次包发给主机后位移计数器被复位位移计数器可表示的值的范围是-255 到+255）。如果超过了范围，相应的溢出位就会被置位，并在复位之前，计数器不会再增减。

而所谓的 Intellimouse，因为多了 2 个按键和一个滚轮，所以 Intellimouse 的一个位移数据包由 4 个字节组成，如下图所示：

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X Movement							
Byte 3	Y Movement							
Byte 4	Always 0	Always 0	5th Btn	4th Btn	Z3	Z2	Z1	Z0

图 3.23.1.6 Intellimouse 鼠标位移数据包格式

Z0-Z3 是 2 的补码，用于表示从上次数据报告以来滚轮的位移量。有效范围从-8 到+7，第四键如果按下，则 4th Btn 位被置位，如果没有按下，则 4th Btn 位为 0。第五键也与此类似。

鼠标的介绍我们就简单的介绍到这里，详细的说明请参考《PS/2 技术参考》第三章 PS/2 鼠标接口（第 36 页）。

3.23.2 硬件设计

本节实验功能简介：开机的时候先检测是否有鼠标接入，如果没有/检测错误，则提示错误代码。只有在检测到 PS/2 鼠标之后才开始后续操作，当检测到鼠标之后，就在 LCD 上显示鼠标位移数据包的内容，并转换为坐标值，在 LCD 上显示，如果有按键按下，则会提示你按下的是哪个按键。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）TFTLCD 液晶模块。

4）PS2 鼠标。

前面三部分，在之前的实例已经介绍过了，下面简单看一下PS/2接口与STM32的连接电路图，如下图所示：

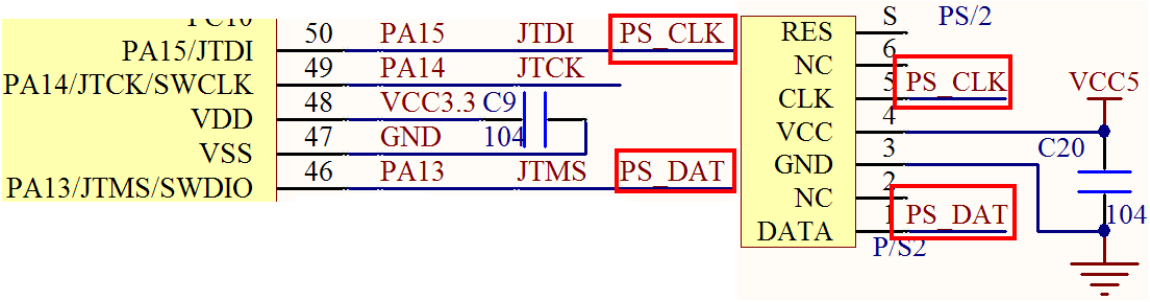


图 3.23.2.1 PS/2 与 STM32 的连接电路图

可以看到，PS/2 接口与 STM32 的连接仅仅 2 个 IO 口，但是是与 JTAG/SMD 复用的，所以在开启了 PS/2 以后，JTAG 与 SMD 都不能使用了。

3.23.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹下新建一个 PS2 和 MOUSE 的文件夹。在 PS2 文件夹里面新建 ps2.c 和 ps2.h 两个文件。然后在 MOUSE 文件夹下新建 mouse.c 和 mouse.h 两个文件。并将这个两个文件夹加入头文件包含路径。

打开 ps2.c，输入如下代码：

```
#include "ps2.h"
#include "usart.h"
//Mini STM32 开发板
//PS2  驱动代码
//正点原子@ALIENTEK
//PS2 产生的时钟频率在 10~20Khz(最大 33K)
//高/低电平的持续时间为 25~50us 之间.

//PS2_Status 当前状态标志
//[5:4]:当前工作的模式;[7]:接收到一次数据
//[6]:校验错误;[3:0]:收到的数据长度;
u8 PS2_Status=CMDMODE; //默认为命令模式
u8 PS2_DATA_BUF[16];     //ps2 数据缓存区

//位计数器
u8 BIT_Count=0;

//中断 15~10 处理函数
//每 11 个 bit,为接收 1 个字节
//每接收完一个包(11 位)后,设备至少会等待 50ms 再发送下一个包
//只做了鼠标部分,键盘部分暂时未加入
//CHECK OK 2010/5/2
void EXTI15_10_IRQHandler(void)
{
    static u8 tempdata=0;
```

```
static u8 parity=0;
if(EXTI->PR&(1<<15))//中断 15 产生了相应的中断
{
    EXTI->PR=1<<15;   //清除 LINE15 上的中断标志位
    if(BIT_Count==0)
    {
        parity=0;
        tempdata=0;
    }
    BIT_Count++;
    if(BIT_Count>1&&BIT_Count<10)//这里获得数据
    {
        tempdata>>=1;
        if(PS2_SDA)
        {
            tempdata|=0x80;
            parity++;//记录 1 的个数
        }
    }else if(BIT_Count==10)//得到校验位
    {
        if(PS2_SDA)parity|=0x80;//校验位为 1
    }
    if(BIT_Count==11)//接收到 1 个字节的数据了
    {
        BIT_Count=parity&0x7f;//取得 1 的个数

if(((BIT_Count%2==0)&&(parity&0x80))||((BIT_Count%2==1)&&(parity&0x80)==0))// 奇 偶校验 OK
        {
            //PS2_Status|=1<<7;//标记得到数据
            BIT_Count=PS2_Status&0x0f;
            PS2_DATA_BUF[BIT_Count]=tempdata;//保存数据
            if(BIT_Count<15)PS2_Status++;       //数据长度加 1
            BIT_Count=PS2_Status&0x30;            //得到模式
            switch(BIT_Count)
            {
                case CMDMODE://命令模式下,每收到一个字节都会产生接收完成
                    PS2_Dis_Data_Report();//禁止数据传输
                    PS2_Status|=1<<7; //标记得到数据
                    break;
                case KEYBOARD:
                    break;
                case MOUSE:
```

```
                        if(MOUSE_ID==0)//标准鼠标,3 个字节
                        {
                            if((PS2_Status&0x0f)==3)
                            {
                                PS2_Status|=1<<7;//标记得到数据
                                PS2_Dis_Data_Report();//禁止数据传输
                            }
                        }else if(MOUSE_ID==3)//扩展鼠标,4 个字节
                        {
                            if((PS2_Status&0x0f)==4)
                            {
                                PS2_Status|=1<<7;//标记得到数据
                                PS2_Dis_Data_Report();//禁止数据传输
                            }
                        }
                        break;
                    }
                }else
                {
                    PS2_Status|=1<<6;//标记校验错误
                    PS2_Status&=0xf0;//清除接收数据计数器
                }
                BIT_Count=0;
            }
        }
}
//禁止数据传输
//把时钟线拉低,禁止数据传输
//CHECK OK 2010/5/2
void PS2_Dis_Data_Report(void)
{
    PS2_Set_Int(0);     //关闭中断
    PS2_SET_SCL_OUT();//设置 SCL 为输出
    PS2_SCL_OUT=0;      //抑制传输
}
//使能数据传输
//释放时钟线
//CHECK OK 2010/5/2
void PS2_En_Data_Report(void)
{
    PS2_SET_SCL_IN(); //设置 SCL 为输入
    PS2_SET_SDA_IN(); //SDA IN
    PS2_SCL_OUT=1;      //上拉
```

```
    PS2_SDA_OUT=1;
    PS2_Set_Int(1);     //开启中断
}

//PS2 中断屏蔽设置
//en:1，开启;0，关闭;
//CHECK OK 2010/5/2
void PS2_Set_Int(u8 en)
{
    EXTI->PR=1<<15;   //清除 LINE15 上的中断标志位
    if(en)EXTI->IMR|=1<<15;//不屏蔽 line15 上的中断
    else EXTI->IMR&=~(1<<15);//屏蔽 line15 上的中断
}
//等待 PS2 时钟线 sta 状态改变
//sta:1，等待变为 1;0，等待变为 0;
//返回值:0，时钟线变成了 sta;1，超时溢出;
//CHECK OK 2010/5/2
u8 Wait_PS2_Scl(u8 sta)
{
    u16 t=0;
    sta=!sta;
    while(PS2_SCL==sta)
    {
        delay_us(1);
        t++;
        if(t>16000)return 1;//时间溢出  (设备会在 10ms 内检测这个状态)
    }
    return 0;//被拉低了
}
//在发送命令/数据之后,等待设备应带,该函数用来获取应答
//返回得到的值
//返回 0，且 PS2_Status.6=1，则产生了错误
//CHECK OK 2010/5/2
u8 PS2_Get_Byte(void)
{
    u16 t=0;
    u8 temp=0;
    while(1)//最大等待 55ms
    {
        t++;
        delay_us(10);
        if(PS2_Status&0x80)//得到了一次数据
        {
```

```
                temp=PS2_DATA_BUF[PS2_Status&0x0f-1];
                PS2_Status&=0x70;//清除计数器，接收到数据标记
                break;
            }else if(t>5500||PS2_Status&0x40)break;//超时溢出/接收错误
        }
        PS2_En_Data_Report();//使能数据传输
        return temp;
}
//发送一个命令到 PS2.
//返回值:0，无错误,其他,错误代码
u8 PS2_Send_Cmd(u8 cmd)
{
        u8 i;
        u8 high=0;//记录 1 的个数
        PS2_Set_Int(0);     //屏蔽中断
        PS2_SET_SCL_OUT();//设置 SCL 为输出
        PS2_SET_SDA_OUT();//SDA OUT
        PS2_SCL_OUT=0;//拉低时钟线
        delay_us(120);//保持至少 100us
        PS2_SDA_OUT=0;//拉低数据线
        delay_us(10);
        PS2_SET_SCL_IN();//释放时钟线,这里 PS2 设备得到第一个位,开始位
        PS2_SCL_OUT=1;
        if(Wait_PS2_Scl(0)==0)//等待时钟拉低
        {
            for(i=0;i<8;i++)
            {
                if(cmd&0x01)
                {
                    PS2_SDA_OUT=1;
                    high++;
                }else PS2_SDA_OUT=0;
                cmd>>=1;
                //这些地方没有检测错误,因为这些地方不会产生死循环
                Wait_PS2_Scl(1);//等待时钟拉高  发送 8 个位
                Wait_PS2_Scl(0);//等待时钟拉低
            }
            if((high%2)==0)PS2_SDA_OUT=1;//发送校验位  10
            else PS2_SDA_OUT=0;
            Wait_PS2_Scl(1); //等待时钟拉高  10 位
            Wait_PS2_Scl(0); //等待时钟拉低
            PS2_SDA_OUT=1;     //发送停止位    11
            Wait_PS2_Scl(1);//等待时钟拉高  11 位
```

```
            PS2_SET_SDA_IN();//SDA in
            Wait_PS2_Scl(0);//等待时钟拉低
            if(PS2_SDA==0)Wait_PS2_Scl(1);//等待时钟拉高 12 位
            else
            {
                PS2_En_Data_Report();
                return 1;//发送失败
            }
        }else
        {
            PS2_En_Data_Report();
            return 2;//发送失败
        }
        PS2_En_Data_Report();
        return 0;       //发送成功
    }
    //PS2 初始化
    void PS2_Init(void)
    {
        RCC->APB2ENR|=1<<2;       //使能 PORTA 时钟
        JTAG_Set(JTAG_SWD_DISABLE);
        GPIOA->CRH&=0X0F0FFFFF;//PA13,15 设置成输入
        GPIOA->CRH|=0X80800000;//PA13,15 设置成输出
        GPIOA->ODR|=5<<13;
        AFIO->EXTICR[3]&=0XFFFF0FFF;//EXTI15 映射到 PA15
        AFIO->EXTICR[3]|=0<<12;         //EXTI15 映射到 PA15
        EXTI->IMR|=1<<15;               //开启 line15 上的中断
        EXTI->EMR|=1<<15;               //不屏蔽 line15 上的事件
        EXTI->FTSR|=1<<15;              //line15 上事件下降沿触发
        MY_NVIC_Init(1,2,EXTI15_10_IRQChannel,2);//分配到第二组,抢占 2,响应 3
    }
```

该部分为底层的 PS/2 协议驱动程序，采用中断接收 PS/2 设备产生的时钟信号，然后解析。保存 ps2.c 文件，并加入到 HARDWARE 组下，然后打开 ps2.h，在该文件里面输入如下代码：

```
    #ifndef __PS2_H
    #define __PS2_H
    #include "delay.h"
    #include "sys.h"
    //Mini STM32 开发板
    //PS2 驱动代码
    //正点原子@ALIENTEK
    //物理接口定义
    //PS2 输入
    #define PS2_SCL PAin(15)   //PA15
```

```
#define PS2_SDA PAin(13)   //PA13
//PS2 输出
#define PS2_SCL_OUT PAout(15)      //PA15
#define PS2_SDA_OUT PAout(13)      //PA13

//设置 PS2_SCL 输入输出状态.
#define PS2_SET_SCL_IN() {GPIOA->CRH&=0X0FFFFFFF;GPIOA->CRH|=0X80000000;}
#define PS2_SET_SCL_OUT() {GPIOA->CRH&=0X0FFFFFFF;GPIOA->CRH|=0X30000000;}

//设置 PS2_SDA 输入输出状态.
#define PS2_SET_SDA_IN() {GPIOA->CRH&=0XFF0FFFFF;GPIOA->CRH|=0X00800000;}
#define PS2_SET_SDA_OUT() {GPIOA->CRH&=0XFF0FFFFF;GPIOA->CRH|=0X00300000;}

#define MOUSE       0X20 //鼠标模式
#define KEYBOARD 0X10 //键盘模式
#define CMDMODE    0X00 //发送命令
//PS2_Status 当前状态标志
//[5:4]:当前工作的模式;[7]:接收到一次数据
//[6]:校验错误;[3:0]:收到的数据长度;
extern u8 PS2_Status;           //定义为命令模式
extern u8 PS2_DATA_BUF[16]; //ps2 数据缓存区
extern u8 MOUSE_ID;

void PS2_Init(void);
u8 PS2_Send_Cmd(u8 cmd);
void PS2_Set_Int(u8 en);
u8 PS2_Get_Byte(void);
void PS2_En_Data_Report(void);
void PS2_Dis_Data_Report(void);
#endif
```

此部分代码我们不多说，保存。然后打开 mouse.c，输入如下代码：

```
#include "mouse.h"
#include "usart.h"
#include "lcd.h"
//Mini STM32 开发板
//鼠标 驱动代码
//正点原子@ALIENTEK
u8 MOUSE_ID;//用来标记鼠标 ID
PS2_Mouse MouseX;
//处理 MOUSE 的数据
```

```
//CHECK OK 2010/5/2
void Mouse_Data_Pro(void)
{
    MouseX.x_pos+=(signed char)PS2_DATA_BUF[1];
    MouseX.y_pos+=(signed char)PS2_DATA_BUF[2];
    MouseX.z_pos+=(signed char)PS2_DATA_BUF[3];
    MouseX.bt_mask=PS2_DATA_BUF[0]&0X07;//取出掩码
}
//初始化鼠标
//返回:0,初始化成功
//其他:错误代码
//CHECK OK 2010/5/2
u8 Init_Mouse(void)
{
    u8 t;
    PS2_Init();
    PS2_Status=CMDMODE; //进入命令模式
    delay_ms(500);          //等待上电复位完成
    t=PS2_Send_Cmd(PS_RESET);   //复位鼠标
    if(t!=0)return 1;
    t=PS2_Get_Byte();
    if(t!=0XFA)return 2;
    t=0;
    while((PS2_Status&0x80)==0)//等待复位完毕
    {
        t++;
        delay_ms(10);
        if(t>50)return 3;
    }
    PS2_Get_Byte();//得到 0XAA
    PS2_Get_Byte();//得到 ID 0X00

    //进入滚轮模式的特殊初始化序列
    PS2_Send_Cmd(SET_SAMPLE_RATE);//进入设置采样率
    if(PS2_Get_Byte()!=0XFA)return 4;//传输失败
    PS2_Send_Cmd(0XC8);//采样率 200
    if(PS2_Get_Byte()!=0XFA)return 5;//传输失败
    PS2_Send_Cmd(SET_SAMPLE_RATE);//进入设置采样率
    if(PS2_Get_Byte()!=0XFA)return 6;//传输失败
    PS2_Send_Cmd(0X64);//采样率 100
    if(PS2_Get_Byte()!=0XFA)return 7;//传输失败
    PS2_Send_Cmd(SET_SAMPLE_RATE);//进入设置采样率
    if(PS2_Get_Byte()!=0XFA)return 8;//传输失败
```

```
    PS2_Send_Cmd(0X50);//采样率 80
    if(PS2_Get_Byte()!=0XFA)return 9;//传输失败
    //序列完成
    PS2_Send_Cmd(GET_DEVICE_ID); //读取 ID
    if(PS2_Get_Byte()!=0XFA)return 10;//传输失败
    MOUSE_ID=PS2_Get_Byte();//得到 MOUSE ID

    PS2_Send_Cmd(SET_SAMPLE_RATE);//再次进入设置采样率
    if(PS2_Get_Byte()!=0XFA)return 11;//传输失败
    PS2_Send_Cmd(0X0A);//采样率 10
    if(PS2_Get_Byte()!=0XFA)return 12;//传输失败
    PS2_Send_Cmd(GET_DEVICE_ID); //读取 ID
    if(PS2_Get_Byte()!=0XFA)return 13;//传输失败
    MOUSE_ID=PS2_Get_Byte();//得到 MOUSE ID

    PS2_Send_Cmd(SET_RESOLUTION);   //设置分辨率
    if(PS2_Get_Byte()!=0XFA)return 14;//传输失败
    PS2_Send_Cmd(0X03);//8 点/mm
    if(PS2_Get_Byte()!=0XFA)return 15;//传输失败
    PS2_Send_Cmd(SET_SCALING11);    //设置缩放比率为 1:1
    if(PS2_Get_Byte()!=0XFA)return 16;//传输失败

    PS2_Send_Cmd(SET_SAMPLE_RATE); //设置采样率
    if(PS2_Get_Byte()!=0XFA)return 17;//传输失败
    PS2_Send_Cmd(0X28);//40
    if(PS2_Get_Byte()!=0XFA)return 18;//传输失败

    PS2_Send_Cmd(EN_DATA_REPORT);    //使能数据报告
    if(PS2_Get_Byte()!=0XFA)return 19;//传输失败

    PS2_Status=MOUSE;//进入鼠标模式
    return 0;//无错误,初始化成功
}
```

该部分仅 2 个函数，Init_Mouse 用于初始化鼠标，让鼠标进入 Intellimouse 模式，里面的初始化序列完全按照《PS/2 技术参考》里面介绍的来设计。另外一个函数就是将收到的数据简单处理一下。保存 mouse.c，然后打开 mouse.h，输入如下内容：

```
#ifndef __MOUSE_H
#define __MOUSE_H
#include "ps2.h"
//Mini STM32 开发板
//鼠标 驱动代码
//正点原子@ALIENTEK
//2010/6/17
```

```
//HOST->DEVICE 的命令集
#define PS_RESET            0XFF //复位命令  回应 0XFA
#define RESEND                  0XFE //再次发送
#define SET_DEFAULT             0XF6 //使用默认设置  回应 0XFA
#define DIS_DATA_REPORT     0XF5 //禁用数据报告  回应 0XFA
#define EN_DATA_REPORT          0XF4 //使能数据报告  回应 0XFA
#define SET_SAMPLE_RATE         0XF3 //设置采样速率  回应 0XFA
#define GET_DEVICE_ID       0XF2 //得到设备 ID     回应 0XFA+ID
#define SET_REMOTE_MODE     0XF0 //设置到 REMOTE 模式  回应 OXFA
#define SET_WRAP_MODE       0XEE //设置到 WRAP 模式  回应 0XFA
#define RST_WRAP_MODE       0XEC //回到 WRAP 之前的模式  回应 0XFA
#define READ_DATA           0XEB //读取数据  回应 0XFA+位移数据包
#define SET_STREAM_MODE     0XEA //设置到 STREAM 模式  回应 0XFA
#define STATUS_REQUEST      0XE9 //请求得到状态  回应 0XFA+3 个字节
#define  SET_RESOLUTION     0XE8  //设置分辨率  回应  OXFA+读取  1  个字节+应带
0XFA
#define SET_SCALING21       0XE7 //设置缩放比率为 2:1  回应 0XFA
#define SET_SCALING11       0XE6 //设置缩放比率为 1:1  回应 0XFA
//DEVICE->HOST 的指令
#define ERROR                   0XFC //错误
//#define RESEND                0XFE //再次发送

//鼠标结构体
typedef struct
{
    short x_pos;//横坐标
    short y_pos;//纵坐标
    short z_pos;//滚轮坐标
    u8    bt_mask;//按键标识,bit2 中间键;bit1,右键;bit0,左键
} PS2_Mouse;
extern PS2_Mouse MouseX;
extern u8 MOUSE_ID;//鼠标 ID,0X00,表示标准鼠标(3 字节);0X03 表示扩展鼠标(4 字节)

u8 Init_Mouse(void);
void Mouse_Data_Pro(void);
#endif
```

该部分代码定义了一个鼠标结构体，用于存放鼠标相关的数据，并对鼠标的相关命令进行了宏定义。保存此部分代码，我们就剩下最后一步了，打开 test.c 文件，在里面修改 main 函数如下：

```
int main(void)
{
    u8 t;
```

```
    u8 errcnt=0;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LCD_Init();                 //初始化液晶
    //KEY_Init();               //按键初始化
    LED_Init();               //LED 初始化
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"PS2 Mouse TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2011/1/1");
RST:t=Init_Mouse();
    if(t==0)LCD_ShowString(60,130,"Mouse Init OK!");
    else
    {
        printf("ERROR_CODE:%d\n",t);
        LCD_ShowString(60,130,"Error CODE:");
        LCD_ShowNum(148,130,t,3,16);//填充模式
        LED0=!LED0;
        //delay_ms(500);
        goto RST;
    }
    POINT_COLOR=BLUE;
    LCD_ShowString(60,150,"Mouse ID:");
    LCD_ShowNum(132,150,MOUSE_ID,3,16);//填充模式
    LCD_ShowString(30,170,"BUF[0]:");
    LCD_ShowString(30,186,"BUF[1]:");
    LCD_ShowString(30,202,"BUF[2]:");
    if(MOUSE_ID==3)LCD_ShowString(30,218,"BUF[3]:");
    LCD_ShowString(90+30,170,"X    POS:");
    LCD_ShowString(90+30,186,"Y    POS:");
    LCD_ShowString(90+30,202,"Z    POS:");
    if(MOUSE_ID==3)LCD_ShowString(90+30,218,"BUTTON:");
    t=0;
    while(1)
    {
        if(PS2_Status&0x80)//得到了一次数据
        {
            LCD_ShowNum(56+30,170,PS2_DATA_BUF[0],3,16);//填充模式
            LCD_ShowNum(56+30,186,PS2_DATA_BUF[1],3,16);//填充模式
            LCD_ShowNum(56+30,202,PS2_DATA_BUF[2],3,16);//填充模式
            if(MOUSE_ID==3)LCD_ShowNum(56+30,218,PS2_DATA_BUF[3],3,16);//  填
```

充模式

```
                Mouse_Data_Pro();//处理数据
                LCD_ShowNum(146+30,170,MouseX.x_pos,5,16);//填充模式
                LCD_ShowNum(146+30,186,MouseX.y_pos,5,16);//填充模式
                if(MOUSE_ID==3)LCD_ShowNum(146+30,202,MouseX.z_pos,5,16);// 填 充 模
式
                if(MouseX.bt_mask&0x01)LCD_ShowString(146+30,218,"LEFT");
                else LCD_ShowString(146+30,218,"      ");
                if(MouseX.bt_mask&0x02)LCD_ShowString(146+30,234,"RIGHT");
                else LCD_ShowString(146+30,234,"      ");
                if(MouseX.bt_mask&0x04)LCD_ShowString(146+30,250,"MIDDLE");
                else LCD_ShowString(146+30,250,"      ");
                PS2_Status=MOUSE;
                PS2_En_Data_Report();//使能数据报告
            }else if(PS2_Status&0x40)
            {
                errcnt++;
                PS2_Status=MOUSE;
                LCD_ShowNum(86+30,234,errcnt,3,16);//填充模式
            }
            t++;
            delay_ms(1);
            if(t==200)
            {
                t=0;
                LED0=!LED0;
            }
        }
    }
```

至此，PS/2 鼠标实验的软件设计部分就结束了。

3.23.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 显示如下内容（假定 PS/2 鼠标已经接上，并且初始化成功）：

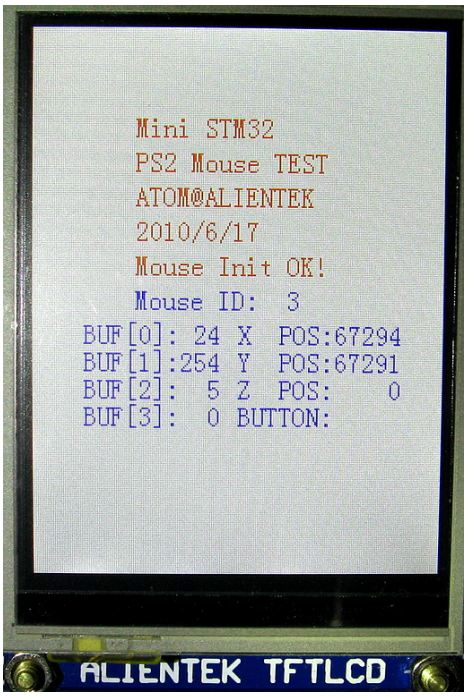


图 3.23.4.1 PS/2 鼠标实验显示结果

移动鼠标，或者按动按键，就可以看到上面的数据不断变化，证明我们的鼠标已经成功被驱动了，接下来我们就可以使用鼠标来控制 STM32 了。

3.24 汉字显示实验

汉字显示在很多单片机系统都需要用到，少则几个字，多则整个汉字库的支持，更有甚者还要支持多国字库，那就更麻烦了。在我们的 ALIENTEK MiniSTM32 上，完全可以显示整个汉字库（GBK 汉字库）的，本节将向大家介绍如何在 ALIENTEK MiniSTM32 开发板上显示汉字。本节分为如下几个部分：

3.24.1 汉字显示原理简介
3.24.2 硬件设计
3.24.3 软件设计
3.24.4 下载与测试

3.24.1 汉字显示原理简介

常用的汉字内码系统有 GB2312，GB13000，GBK，BIG5（繁体）等几种，其中 GB2312 支持的汉字仅有几千个，很多时候不够用，而 GBK 内码不仅完全兼容 GB2312，还支持了繁体字，总汉字数有 2 万多个，完全能满足我们一般应用的要求。

本实例我们将制作一个 GBK 字库，制作好的字库放在 SD 卡里面，然后通过 SD 卡，将字库文件复制到 W25X16 里，这样，W25X16 就相当于一款汉字字库芯片了。

汉字在液晶上的显示原理与前面显示字符的是一样的。汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画。所以要显示汉字，我们首先要知道汉字的点阵数据，这些数据可以由专门的软件来生成。只要知道了一个汉字点阵的生成方法，那么我们在程序里面就可以把这个点阵数据解析成一个汉字。

知道显示了一个汉字，就可以推及整个汉字库了。汉字在电脑里面存储不是以点阵数据的形式存储的（否则那占用的空间就太大了），而是以内码的形式存储的，就是 GB2312/GBK/BIG5 等这几种的一种（最常用的是哪个我也不清楚，但是可以肯定是我们装的简体 XP，汉字一般都能用 GBK 码或 GB2312 码解析），每个汉字对应着一个内码，在知道了内码之后再去字库里面查找这个汉字的点阵数据，然后在液晶上显示出来。这个过程我们是看不到，但是计算机是要去执行的。

单片机要显示汉字也与此类似：汉字内码（GBK/GB2312）->查找点阵库->解析->显示。

所以只要我们有了整个汉字库的点阵，就可以把电脑上的文本信息在单片机上显示出来了。这里我们要解决的最大问题就是制作一个与汉字内码对的上号的汉字点阵库。而且要方便单片机的查找。每个 GBK 码由 2 个字节组成，第一个字节为 0X81~0XFE，第二个字节分为两部分，一是 0X40~0X7E，二是 0X80~0XFE。其中与 GB2312 相同的区域，字完全相同。

我们把第一个字节代表的意义称为区，那么 GBK 里面总共有 126 个区（0XFE-0X81+1），每个区内有 190 个汉字（0XFE-0X80+0X7E-0X40+2），总共就有 126*190=23940 个汉字。我们的点阵库只要按照这个编码规则从 0X8140 开始，逐一建立，每个区的点阵大小为每个汉字所用的字节数*190。这样，我们就可以得到在这个字库里面定位汉字的方法：

当 GBKL<0X7F 时：Hp=((GBKH-0x81)*190+GBKL-0X40)*(size*2);

当 GBKL>0X80 时：Hp=((GBKH-0x81)*190+GBKL-0X41)*(size*2);

其中 GBKH、GBKL 分别代表 GBK 的第一个字节和第二个字节(也就是高位和低位)，size 代表汉字字体的大小（比如 16 字体，12 字体等），Hp 则为对应汉字点阵数据在字库里面的起始地址。

这样我们只要得到了汉字的 GBK 码，就可以显示这个汉字了。从而实现汉字在液晶上的显示。而 XP 在存储文件名的时候，如果是长文件名，则是按照 UNICODE 码存放的，而 UNICODE 码与 GBK 码并不一致，所以如果要支持 UNICODE 内码的汉字显示则需要一个 UNICODE 到 GBK 码的转换码表，通过先将 UNICODE 码转换为 GBK 码，再从 GBK 码字库里面查找点阵数据，从而显示 UNICODE 码的汉字。

UNICODE 码表的制作方法是将 UNICODE 码从低到高顺序排列，然后在对应的位置存放 GBK 码的码值就可以了，这样我们就可以通过 UNICODE 码快速的找到 GBK 码。UNICODE 码中用于存放汉字的字段为 0X4E00~0X9FA5，总共 20902 个汉字。我们将这些对应的汉字的 GBK 码顺序存入相应的位置，就得到了 UNICODE 到 GBK 的转换码表。

关于 UNICODE 转 GBK 就介绍到这里，我们这里提供的 UNICODE 转 GBK 码码表是在网

友波仔制作的 UNICODE 转 GBK 码表基础上修改而来的，波仔制作的并不能显示大写的标点符号，我们把这部分加入了进去，把汉字标点符号对应的 GBK 码按 UNICODE 的编码先后顺序写入到波仔制作的转换表的后面，这样我们在程序里做一点点小改动，就可以实现对 UNICODE 码的标点符号的支持了。加入的标点符号总共 97 个，对应 UNICODE 码的 0XFF01~0XFF61。

每个 GBK 码占用了 2 个字节，所以整个 UNICODE 转 GBK 码码表文件的大小为：2*（20902+97）=41998 个字节（42K）。16*16 大小的汉字每个汉字点阵需要 32 个字节，可以得到整个 GBK 码 16 字库的大小为：32*（23940）=766080 字节（749K）。

字库的生成，我们要用到一款软件，由易木雨软件工作室设计的点阵字库生成器 V3.8。该软件可以在 WINDOWS 系统下生成任意点阵大小的 ASCII，GB2312(简体中文)、GBK(简体中文)、BIG5(繁体中文)、HANGUL(韩文)、SJIS(日文)、Unicode 以及泰文，越南文、俄文、乌克兰文，拉丁文，8859 系列等共二十几种编码的字库，不但支持生成二进制文件格式的文件，也可以生成 BDF 文件，还支持生成图片功能，并支持横向，纵向等多种扫描方式，且扫描方式可以根据用户的需求进行增加。该软件的界面如下：

图 3.24.1.1 点阵字库生成器默认界面

比如我们要生成 16*16 的 GBK 字库，则选择 GBK，字宽和高均选择 16，自他大小选择 12（比较适合），然后模式选择纵向取模方式二（字节高位在前，低位在后），最后点击创建，就可以开始生成我们需要的字库了。具体设置如下图所示：

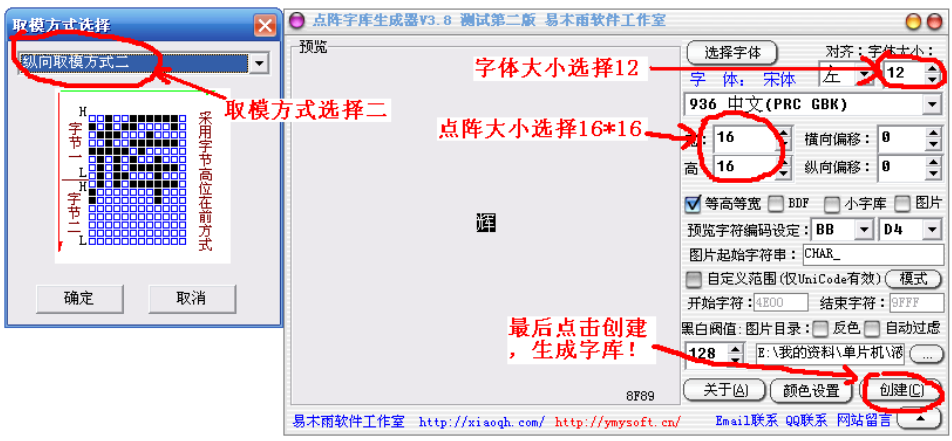


图 3.24.1.2 生成 GBK16*16 字库的设置方法

这里注意，软件里面的字体大小并不是我们生成点阵的大小，12 字体是 XP 的叫法，我们字体的大小以宽和高的大小来决定！可以简单的这么认为：XP 的 12 字体，基本上就等于 16*16 大小。该软件还可以生成其他很多字库，字体也可选，详细的介绍请看软件自带的《点阵字库生成器说明书》。

本节还需要用到 SD 卡和 FLASH 部分，在前面的例子都已经介绍过了。但还有最重要的 FAT 文件系统没有介绍，本节将要用到 FAT(FAT16/32)文件系统来读取 SD 卡上的字库和 UNICODE 到 GBK 的转换码表。

FAT 文件系统本身比较复杂，如果要深入估计介绍都可以写本书了。所以这里我们只简单介绍一下，更多的介绍，请参考我们提供的光盘内的 FAT 资料文件夹，里面有 FAT 文件系统的详细介绍资料。

常用的文件系统有 FAT12/16/32 等，FAT12 是最古老的文件系统，只能管理 8M 左右的空间，现在基本淘汰了。FAT16 则可以管理 2G 的空间(通过特殊处理也能管理 2G 以上的空间)，而 FAT32 则能管理到 2TB（2048GB）的空间。FAT32 较 FAT16 的优势还在于 FAT32 采用了更小的簇，可以更有效的保存信息，而不会造成多的浪费。

XP 在 SD 卡里面建立的文件系统最常用的也就是 FAT16 和 FAT32。这是由 XP 在格式化 SD 卡的时候建立的，通常 SD 卡上的数据信息由 MBR、DBR、FAT、FDT 和数据区 5 个部分组成（有的也没有 MBR）。我们以 FAT32 为例做介绍。

MBR 称为主引导记录区，该区存储了分区表等信息，位于 SD 卡的扇区 0（物理扇区），在其分区信息里面记录了 DBR 所在的位置，SD 卡一般只会有一个分区，所以也就只要找到分区 1 的 DBR 所在位置就可以了。

DBR 称为操作系统引导记录区，如果没有 MBR，那么 DBR 就位于 0 扇区，如果有则必须通过 MBR 区得到 DBR 所在的地址，然后读出 DBR 信息。在 DBR 区，我们可以知道每个扇区所占用的字节数、每个簇的扇区数、FAT 表的份数、每个 FAT 表的扇区数、跟目录簇号、FAT 表 1 所在的扇区等一系列非常重要的信息。

FAT 称为文件分配表（FAT 表），一般一个卡上会存在 2 个 FAT 表，一个用作备份，一个用作使用。FAT 表一般紧随 DBR，另一个 FAT 表则紧随第一个 FAT 表，这样只要知道了第一个 FAT 表的位置及大小，那么第二个 FAT 表的位置也就确定了。FAT 表记录了每个文件的位置和区域，是一种链式结构，FAT 以“F8 FF FF 0F FF FF FF FF”这样的 8 个字节为表头，用以表示 FAT 表的开始，后面的数据每四个字节为一个簇项（从第 2 簇开始）,用来标记下一个簇所在的位置，这样每个位置都存储了下一个簇，只要按着这个表走，就可以找到文件的所有内容。如果找到下一个簇位置，里面记录的是“FF FF FF 0F”，代表这个文件到此就结束了，没有后续簇了，这样一个文件的读取就结束了。

FDT 称为文件根目录表，这个区域固定为 32 个扇区，假设每个扇区为 512 个字节，那么更目录下最多存放 512 个文件（假设都用短文件名存储，每个短文件名占 32 个字节）。

文件目录表是另一个重要的部分，FAT 文件系统中（仅以短文件名介绍），文件目录项在目录表下以 32 个字节的方式记录，个字段定义如下：

FAT32 文件目录项 32 个字节的定义			
字节偏移量	字数量	定义	
0~7	8	文件名	
8~10	3	扩展名	
11	1	属性字节	0x00 （读写）
			0x01 （只读）
			0x02 （隐藏）
			0x04 （系统）
			0x08 （卷标）
			0x10 （子目录）
			0x20 （归档）
12	1	系统保留	
13	1	创建时间的 10 毫秒位	
14~15	2	文件创建时间	
16~17	2	文件创建日期	
18~19	2	文件最后访问日期	
20~21	2	文件起始簇号的高 16 位	
22~23	2	文件的最近修改时间	
24~25	2	文件的最近修改日期	
26~27	2	文件起始簇号的低 16 位	
28~31	4	表示文件的长度	

图 3.24.1.3 文件目录项各字节定义

从上图可只，我们在文件的目录项就可以找到该文件的其实簇，然后在 FAT 表里面找到该簇开始的下一个簇，依次读取这些簇就可以把整个文件读出来了。FAT 文件系统就给大家介绍到这里。

3.24.2 硬件设计

本节实验功能简介：开机的时候先检测 W25X16 中是否已经存在字库，如果存在，则按次序显示汉字。如果没有，则检测 SD 卡和文件系统，并查找 SYSTEM 文件夹下的 FONT 文件夹，在该文件夹内查找 UNI2GBK.SYS 和 GBK16.FON（这两个文件是由我们自己生成的 UNICODE 转 GBK 码表文件和 GBK 字库（16*16）文件）。在检测到这些文件之后，就开始更新字库，更新完毕才开始显示汉字。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）TFTLCD 液晶模块。

4）按键 0

5）W25X16。

6) SD 卡

这几部分分，在之前的实例中都介绍过了，我们在此就不介绍了。

3.24.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹所在文件夹下新建一个 FAT 和 TEXT 的文件夹。在 FAT 文件夹里面新建 fat.c 和 fat.h 两个文件。然后在 TEXT 文件夹下新建 fontupd.c、fontupd.h、text.c、text.h、untogb.c、untogb.h 这六个文件。并将这个两个文件夹加入头文件包含路径。

打开 fat.c，输入如下代码：

```
#include"FAT.h"
#include "usart.h"
//Mini STM32 开发板
//FAT 驱动 V1.0
//正点原子@ALIENTEK

//全局变量区域
DWORD FirstDirClust;      //根目录簇号
DWORD FirstDataSector;    //数据区的第一个扇区
WORD BytesPerSector; //每扇区字节数
DWORD FATsectors;         //FAT 表所占扇区数
WORD SectorsPerClust;//每簇扇区数
DWORD FirstFATSector;     //第一个 FAT 表(FAT1)所在扇区
DWORD FirstDirSector;     //第一个目录扇区 fat32
DWORD RootDirSectors;     //根目录所占用的扇区数
DWORD RootDirCount;           //根目录下目录项数
BYTE FAT32_Enable;        //FAT32 文件系统标志
```

```
DWORD Cur_Dir_Cluster;//当前目录簇号
//DWORD Fat_Dir_Cluster;//父目录簇号  在 FAT 文件夹里面 CopyDirentruyItem 函数中修改!

FAT_TABLE FAT_TAB;//TINY FAT 表

//文件信息缓存区

FileInfoStruct F_Info[3];
//外部全局变量

//FAT  数据缓存区
u8 fat_buffer[512];//用于存储 FAT 数据
u8 LongNameBuffer[MAX_LONG_NAME_SIZE];//长文件名的缓存区
BOOL LongNameFlag = 0;//是否存在长文件名的标志


//文件类型
//3gp,3g2,m4a,mp4 也是支持的.
//返回值:对应的类型
//0,mp1;1,mp2;2,mp3;3,mp4;4,m4a;5,3gp;6,3g2;7,ogg;8,acc;9,wma;10,wav;11,midi;12,flac;
//13,lrc;14,txt;15,c;16,h;17,file;18,FON;19,SYS;20,bmp;21,jpg;22,jpeg;
const unsigned char *filetype[23]=
{
"MP1","MP2","MP3","MP4","M4A","3GP","3G2","OGG","ACC","WMA","WAV","MID","FL
A",
"LRC","TXT","C    ","H    ","      ","FON","SYS","BMP","JPG","JPE"
};
//返回扩展名类型
//输入:exName  文件扩展名
//返回值:文件的类型,目前支持 20 种文件类型,最大支持 32 种文件类型
u32 FileType_Tell(u8 * exName)
{
    u8 i;
    u8 t;
    for(i=0;i<20;i++)
    {
        for(t=0;t<3;t++)if(exName[t]!=filetype[i][t])break;
        if(t==3)break;
    }
    return 1<<i;//返回文件类型
}
//FAT 初始化，不含 SD 的初始化，用之前应先调用 sd 的初始化
//返回值:0,初始化成功
```

```
//      其他,初始化失败
unsigned char FAT_Init(void)//Initialize of FAT    need initialize SD first
{
    bootsector710 *bs    = 0;
    bpb710              *bpb = 0;
    partrecord       *pr    = 0;

    DWORD hidsec=0;
    u32 Capacity;
    Capacity = SD_GetCapacity();
    if(Capacity<0xff)return 1;
    if(SD_ReadSingleBlock(0,fat_buffer))return 2;
    bs = (bootsector710 *)fat_buffer;
    pr = (partrecord *)((partsector *)fat_buffer)->psPart;//first partition
    hidsec = pr->prStartLBA;//the hidden sectors
    if(hidsec >= Capacity/512)hidsec = 0;
    else
    {
        if(SD_ReadSingleBlock(pr->prStartLBA,fat_buffer))return 3;//read the bpb sector
        bs = (bootsector710 *)fat_buffer;
        if(bs->bsJump[0]!=0xE9 && bs->bsJump[0]!=0xEB)
        {
            hidsec = 0;
            if(SD_ReadSingleBlock(0,fat_buffer))return 4;//read the bpb sector
            bs = (bootsector710 *)fat_buffer;
        }
    }
    if(bs->bsJump[0]!=0xE9 && bs->bsJump[0]!=0xEB)return 5;//对付没有 bootsect 的 sd 卡
//dead with the card which has no bootsect
    bpb = (bpb710 *)bs->bsBPB;
    if(bpb->bpbFATsecs)//detemine thd FAT type    //do not support FAT12
    {
        FAT32_Enable=0;  //FAT16
        FATsectors         = bpb->bpbFATsecs;//FAT 表占用的扇区数
        FirstDirClust = 2;
    }
    else
    {
        FAT32_Enable=1;  //FAT32
        FATsectors         = bpb->bpbBigFATsecs;//FAT 占用的扇区数
        FirstDirClust = bpb->bpbRootClust;
    }
```

```
    BytesPerSector      = bpb->bpbBytesPerSec;          //每扇区字节数
    SectorsPerClust     = (BYTE)bpb->bpbSecPerClust;//每簇扇区数
    FirstFATSector      = bpb->bpbResSectors+hidsec;//第一个 FAT 表扇区
    RootDirCount = bpb->bpbRootDirEnts;             //根目录项数
    RootDirSectors      = (RootDirCount*32)>>9;         //根目录占用的扇区数
    FirstDirSector = FirstFATSector+bpb->bpbFATs*FATsectors;//第一个目录扇区
    FirstDataSector     = FirstDirSector+RootDirSectors;//第一个数据扇区
    return 0;
}
//将 FAT 表,从头到尾 COPY 过来,如果没 COPY 完,则 Fat_Over=0,否则为 1
//cluster:文件的首簇
void Copy_Fat_Table(unsigned long cluster)
{
    u32 bcluster;
    u8 fat_base;//0~Fat_Table_Size
    FAT_TAB.Fat_Over=0;
    for(fat_base=0;fat_base<Fat_Table_Size;fat_base++)
    {
        FAT_TAB.Fat_Base_Tab[fat_base]=0;
        FAT_TAB.Fat_Base_Len[fat_base]=0;//全部清空
    }
    fat_base=0;
    bcluster=cluster;//起始簇,不能丢失的.
    FAT_TAB.Fat_Base_Tab[fat_base]=bcluster;
    FAT_TAB.Fat_Base_Len[fat_base]=1;//有 1 个数据了,就是最新的 bcluster
    while(1)
    {
        bcluster=FAT_NextCluster(bcluster);

    if((FAT32_Enable==0&&bcluster==0xffff)||bcluster==0x0ffffff8||bcluster==0x0fffffff)// 文 件 结
束
        {
            FAT_TAB.Fat_Over=1;//文件 COPY 结束
            break;
        }

    if(bcluster-FAT_TAB.Fat_Base_Tab[fat_base]-FAT_TAB.Fat_Base_Len[fat_base]!=0)// 是 否 满
足偏移条件
        {
            fat_base++;//启用下一个 BASE
            if(fat_base>=Fat_Table_Size)//超出了缓冲区范围,文件太大了/磁盘太零散
了!!!
            {
```

```
                    FAT_TAB.Fat_Over=0;//文件 COPY 结束
                    break;
                }
                FAT_TAB.Fat_Base_Tab[fat_base]=bcluster;
                FAT_TAB.Fat_Base_Len[fat_base]=1;//有 1 个数据了,就是最新的 bcluster

            }else FAT_TAB.Fat_Base_Len[fat_base]++;//基址偏移量增加
        }
        //监控用
        //printf("Fat_Over:%d\n",FAT_TAB.Fat_Over);
        //printf("Fat_Head_Pos:%d\n",FAT_TAB.Fat_Head_Pos);
        //printf("Fat_Base_Tab[0]:%d\n",FAT_TAB.Fat_Base_Tab[0]);
        //printf("Fat_Base_Tab[1]:%d\n",FAT_TAB.Fat_Base_Tab[1]);
        //printf("Fat_Base_Tab[2]:%d\n",FAT_TAB.Fat_Base_Tab[2]);
    }
    //在 TINY FAT 表里面查找 cluster 的上一个簇号
    //cluster:当前簇号
    //返回值:cluster,表示不能再向上了
    //              其他值,cluster 的上一个簇
    u32 FatTab_Prev_Cluster(unsigned long cluster)
    {
        u8 t;
        u32 tempclust;
    RSTP:
        for(t=0;t<Fat_Table_Size;t++)
        {

    if(cluster<=(FAT_TAB.Fat_Base_Tab[t]+FAT_TAB.Fat_Base_Len[t]-1)&&cluster>=FAT_TAB.
Fat_Base_Tab[t])break;//在这个 BASE 内
        }
        if(cluster==FAT_TAB.Fat_Base_Tab[t])//是在 BASE,但是是第一个
        {
            if(t==0)//这份 FAT 表  全部找完还没找到
            {
                if(FAT_TAB.Fat_Head_Pos>0)//不超过范围
                {
                    FAT_TAB.Fat_Head_Pos--;
                    tempclust=FAT_TAB.Fat_Base_Head[FAT_TAB.Fat_Head_Pos];//拷贝上一
个 tinyFAT 表的表头
                }else return cluster;     //无法继续向上
                Copy_Fat_Table(tempclust);//COPY  上一个 tinyFAT 表
                goto RSTP;
                //return
```

```
FAT_TAB.Fat_Base_Tab[Fat_Table_Size-1]+FAT_TAB.Fat_Base_Len[Fat_Table_Size-1]-1;//返回上一个簇号
                }
                return FAT_TAB.Fat_Base_Tab[t-1]+FAT_TAB.Fat_Base_Len[t-1]-1;//上一个簇号
        }else return --cluster;//返回上一个簇
    }
    //在 TINY FAT 表里面查找 cluster 的下一个簇号
    //cluster:当前簇号
    //返回值:0x0ffffff8,表示没有后续簇了
    //                  其他值,对应簇号
    u32 FatTab_Next_Cluster(unsigned long cluster)
    {
        u8 t;
    RESN:
        for(t=0;t<Fat_Table_Size;t++)
        {

    if(cluster<=(FAT_TAB.Fat_Base_Tab[t]+FAT_TAB.Fat_Base_Len[t]-1)&&cluster>=FAT_TAB.Fat_Base_Tab[t])break;//在这个 BASE 内
        }
        if(cluster==FAT_TAB.Fat_Base_Tab[t]+FAT_TAB.Fat_Base_Len[t]-1)//是在 BASE,但是是最后一个了
        {
            if((t+1)==Fat_Table_Size)//全部找完还没找到
            {
                if(FAT_TAB.Fat_Over)return 0x0ffffff8;//文件结束了
                if(FAT_TAB.Fat_Head_Pos<Fat_Head_Size-1)//不超过范围
                {

    FAT_TAB.Fat_Base_Head[FAT_TAB.Fat_Head_Pos]=FAT_TAB.Fat_Base_Tab[0];//拷贝当前 tinyFAT 表的第一个簇
                    FAT_TAB.Fat_Head_Pos++;
                }
                Copy_Fat_Table(cluster);//COPY 余下的 FAT 表
                goto RESN;
            }
            if(FAT_TAB.Fat_Base_Len[t+1]==0)return 0x0ffffff8;//没有后续簇了
            return FAT_TAB.Fat_Base_Tab[t+1];//下一个簇号
        }else return ++cluster;//返回下一个簇
    }
    //在 SD 卡上的 FAT 表中查找下一簇号
    //cluster:当前簇号
    //返回值:0x0ffffff8,表示没有后续簇了
```

```
//                  其他值,下一簇号
unsigned long FAT_NextCluster(unsigned long cluster)
{
    DWORD sector;
    DWORD offset;

    if(FAT32_Enable)offset = cluster/128;//FAT32 的 FAT 表中,用四个字节表示一个簇地址.512/4=128
    else offset = cluster/256;              //FAT16 的 FAT 表中,用两个字节表示一个簇地址.512/2=256
    if(cluster<2)return 0x0ffffff8;         //簇 0,1 不能用于存放
    sector=FirstFATSector+offset;//计算该簇实际所在扇区

    if(SD_ReadSingleBlock(sector,fat_buffer))return 0x0ffffff8;//读取 FAT 表,发生错误是返回 0x0ffffff8
    if(FAT32_Enable)
    {
        offset=cluster%128;//计算在扇区内的偏移
        sector=((unsigned long *)fat_buffer)[offset];//u32
    }
    else
    {
        offset=cluster%256;//计算在扇区内的偏移
        sector=((unsigned short *)fat_buffer)[offset];//u16
    }
    return (unsigned long)sector;//return the cluste number
}
//将簇号转变为扇区号
//cluster:要变为扇区的簇号
//返回值:cluster 对应的扇区号
u32 fatClustToSect(u32 cluster)
{
    return FirstDataSector+(DWORD)(cluster-2)*(DWORD)SectorsPerClust;
}
//复制记录项信息
//将 Source 的相关内容复制到 Desti 里面
void CopyDirentruyItem(FileInfoStruct *Desti,direntry *Source)
{
    u8 i;
    u8 t;
    for(i=0;i<8;i++)Desti->F_ShortName[i]=Source->deName[i];//复制短文件名
    Desti->F_Type               = FileType_Tell(Source->deExtension);
    Desti->F_StartCluster       = Source->deStartCluster + (((unsigned
```

```
long)Source->deHighClust)<<16);//不用管
        Desti->F_Size             = Source->deFileSize;
        Desti->F_Attr             = Source->deAttributes;
        Desti->F_CurClust         = 0;//扇区...
        Desti->F_Offset           = 0;//偏移 0

        //FAT 的簇号不能是 0(更目录簇号)
        if(FAT32_Enable&&Desti->F_StartCluster==0)
        {
            Desti->F_StartCluster=FirstDirClust;//改变这个簇号.使其等于根目录所在簇号!!
        }
        if(LongNameFlag)//存在长文件名
        {
            LongNameBuffer[MAX_LONG_NAME_SIZE-1] = 0;
            LongNameBuffer[MAX_LONG_NAME_SIZE-2] = 0;
            //UniToGB(LongNameBuffer);   //把 Unicode 代码转换为 ASICII 码
            for(i=0;i<80;i++)Desti->F_Name[i] = LongNameBuffer[i];//复制长文件名
        }else //短文件名
        {   //2E:当前目录所在簇.2E 2E:父目录所在簇.
            if(Source->deName[0]==0x2e)//得到一个父目录(修改为":\")
            {
                //保存父目录簇号
                //Fat_Dir_Cluster=Desti->F_StartCluster;
                Desti->F_Name[0]=':';
                Desti->F_Name[1]=0x5c;//'\'
                Desti->F_Name[2]='\0';//加入结束符
            }else                               //普通文件
            {
                t=7;//从最后一个短文件名开始,找空格,并丢掉
                while(t>0)
                {
                    if(Source->deName[t]!=' ')break;
                    t--;
                }
                for(i=0;i<t+1;i++)Desti->F_Name[i] = Source->deName[i];//复制短文件名
                if(Desti->F_Attr==0X20&&(Source->deExtension[0]!=0x20))//归档文件且文件
后缀不为空
                {
                    Desti->F_Name[i++]='.';//加入"."
                    for(t=0;t<3;t++)Desti->F_Name[i+t]  =  Source->deExtension[t];//复制后缀

                    Desti->F_Name[i+t]='\0';//加入结束符
                }else Desti->F_Name[i]='\0';//加入结束符
```

```
            }
        }
        return ;
}
//浏览目标文件夹下面的一个文件类
//dir_clust:当前目录所在簇号
//FileInfo :目标文件的实体对象(FileInfoStruct 体)
//type:要查找的文件类型:1<<0,mp1;1<<1,mp2;1<<2,mp3;1<<3,mp4;1<<4,m4a;1<<5,3gp;
// 1<<6,3g2;1<<7,ogg;1<<8,acc;1<<9,wma;1<<10,wav;1<<11,mid;
// 1<<12,flac;1<<13,lrc;1<<14,txt;1<<15,c;1<<16,h;1<<17,file;
//1<<18,fon;1<<19,sys;1<<20,bmp;1<<21,jpg;1<<22,jpeg;
//count :0,返回当前目录下,该类型文件的个数;不为零时,返回第 count 个文件的详细信息
//返回值     :1,操作成功.0,操作失败
u8 Get_File_Info(u32 dir_clust,FileInfoStruct *FileInfo,u32 type,u16 *count)
{
        DWORD sector;
        DWORD cluster=dir_clust;
        DWORD tempclust;
        unsigned char cnt;
        unsigned int offset;
        unsigned short cont=0;//文件索引标志  <65536
        unsigned char j; //long name fat_buffer offset;
        unsigned char *p;//long name fat_buffer pointer
        direntry *item = 0;
        winentry *we =0;
        cont=0;
        LongNameFlag = 0;//清空长文件名标志

        //SD_Init();//初始化 SD 卡，在意外拔出之后可以正常使用
        //goto SD;
        if(cluster==0 && FAT32_Enable==0)//FAT16 根目录读取
        {
                for(cnt=0;cnt<RootDirSectors;cnt++)
                {
                        if(SD_ReadSingleBlock(FirstDirSector+cnt,fat_buffer))return  0;// 读 数 错 误

                        for(offset=0;offset<512;offset+=32)
                        {
                                item=(direntry *)(&fat_buffer[offset]);//指针转换
                                //找到一个可用的文件

if((item->deName[0]!=0x2E)&&(item->deName[0]!=0x00)&&(item->deName[0]!=0xe5)
                                ||((item->deName[0]==0x2E)&&(item->deName[1]==0x2E)))//找到一个合
```

```
法文件.忽略".",使用".."
                    {
                        if(item->deAttributes == 0x0f)//找到一个长文件名
                        {
                            we = (winentry *)(&fat_buffer[offset]);
                            j = 26 *( (we->weCnt-1) & WIN_CNT);//长文件名的长度
                            if(j<MAX_LONG_NAME_SIZE-25)
                            {
                                p = &LongNameBuffer[j];//偏移到目标地址
                                for (j=0;j<10;j++)   *p++ = we->wePart1[j];
                                for (j=0;j<12;j++)   *p++ = we->wePart2[j];
                                for (j=0;j<4;j++)    *p++ = we->wePart3[j];
                                if (we->weCnt & 0x40) (*(unsigned int *)p) = 0;

                                if ((we->weCnt & WIN_CNT) == 1) LongNameFlag = 1;//最
后一个长文件项找到了
                            }
                        }else
                        {

                            if(type&FileType_Tell(item->deExtension))//找到一个目标文件
                            {
                                cont++;//文件索引增加
                             }
                             //查找该目录下,type 类型的文件个数
                            if(*count&&cont==*count)
                            {
                                ////printf("\ncount:%d",*count);
                                CopyDirentruyItem(FileInfo,item);//复制目录项,提取详细信
息

                                return 1;//找到目标文件成功
                            }
                            LongNameFlag=0;//清空长文件名
                        }
                    }
                }
            }
        }
        else//其他文件夹/FAT32 系统
        {
            tempclust=cluster;
            while(1)
            {
```

```
                sector=fatClustToSect(tempclust);
                for(cnt=0;cnt<SectorsPerClust;cnt++)
                {
                    if(SD_ReadSingleBlock(sector+cnt,fat_buffer))return 0;
                    for(offset=0;offset<512;offset+=32)
                    {
                        item=(direntry *)(&fat_buffer[offset]);

    if((item->deName[0]!=0x2E)&&(item->deName[0]!=0x00)&&(item->deName[0]!=0xe5)
                            ||((item->deName[0]==0x2E)&&(item->deName[1]==0x2E)))//找到一
个合法文件.忽略".",使用".."
                            {
                                if(item->deAttributes == 0x0f) //得到一个长文件名
                                {
                                    we = (winentry *)(&fat_buffer[offset]);
                                    j = 26 *( (we->weCnt-1) & WIN_CNT);
                                    if(j<MAX_LONG_NAME_SIZE-25)
                                    {
                                        p = &LongNameBuffer[j];//p 指向长文件名的存放地址
                                        for (j=0;j<10;j++)   *p++ = we->wePart1[j];
                                        for (j=0;j<12;j++)   *p++ = we->wePart2[j];
                                        for (j=0;j<4;j++)    *p++ = we->wePart3[j];
                                        if (we->weCnt & 0x40) (*(unsigned int *)p) = 0;

                                       if ((we->weCnt & WIN_CNT) == 1) LongNameFlag = 1;

                                    }

                                }
                                else
                                {
                                    if(type&FileType_Tell(item->deExtension))//找到一个目标
文件
                                    {
                                        cont++;//文件索引增加
                                     }
                                     //查找该目录下,type 类型的文件个数
                                    if(*count&&cont==*count)
                                    {
                                        CopyDirentruyItem(FileInfo,item);//复制目录项,提取详
细信息
                                        return 1;//找到目标文件成功
                                    }
```

```
                                    LongNameFlag=0;//清空长文件名
                                }
                            }
                        }
                    }
                    tempclust=FAT_NextCluster(tempclust);//查找下一个簇号
                    if(tempclust==0x0fffffff||tempclust==0x0ffffff8
||(FAT32_Enable==0&&tempclust==0xffff))break;
                }
            }
            if(*count==0)
            {
                *count=cont;//得到总共文件数目
                return 1;     //操作成功,找到 cont 个符合条件的文件了
            }else return 0; //操作失败,没找到文件,或者出错
        }
        //打开文件
        //FileInfo:文件信息
        void F_Open(FileInfoStruct *FileInfo)
        {
            FileInfo->F_CurClust=FileInfo->F_StartCluster;//当前簇为首簇
            FileInfo->F_Offset=0;//偏移扇区为 0
        }
        //读取 512 个字节
        //FileInfo:要读取的文件
        //buf        :数据缓存区
        //返回值    :0,操作失败,1,操作成功
        unsigned char F_Read(FileInfoStruct *FileInfo,u8 *buf)
        {
            DWORD sector;
            sector=fatClustToSect(FileInfo->F_CurClust);//得到当前簇号对应的扇区号

            if(SD_ReadSingleBlock(sector+FileInfo->F_Offset,buf))return 0;//读数错误
            FileInfo->F_Offset++;
            if(FileInfo->F_Offset==SectorsPerClust)     //簇的尽头,换簇
            {
                FileInfo->F_Offset=0;
                FileInfo->F_CurClust=FAT_NextCluster(FileInfo->F_CurClust);//读取下一个簇号
                if((FAT32_Enable==0&&FileInfo->F_CurClust==0xffff) \
                ||FileInfo->F_CurClust==0x0ffffff8||FileInfo->F_CurClust       ==       0x0fffffff)return
0;//error
            }
            return 1;//读取成功
```

```
}
//比较两个字符串相等不
//相等,返回 1,不相等,返回 0;
u8 mystrcmp(u8*s1,u8*s2)
{
	u8 len1,len2;
	len1=len2=0;
	while(*s1!='\0')
	{
		len1++;s1++;
	}
	while(*s2!='\0')
	{
		len2++;s2++;
	}
	if(len1!=len2)return 0;//不相等
	s1-=len1;s2-=len1;
	while(*s1!='\0')
	{
		if(*s1!=*s2)return 0;//不相等
		s1++;s2++;
	}
	return 1;
}
//查找系统文件
//在指定目录下,找寻一个指定类型的指定名字的文件
//cluster:文件夹的簇号!!!
//Name	:文件的名字
//type	:文件类型
//返回值 :该文件的详细信息/如果 FileInfo.F_StartCluster=0 则说明此次寻找失败
FileInfoStruct F_Search(u32 cluster,unsigned char *Name,u32 type)
{
	DWORD sector;
	DWORD tempclust;
	unsigned char cnt;
	unsigned int offset;
	direntry *item = 0;
	FileInfoStruct FileInfo;
	if(cluster==0 && FAT32_Enable==0)//FAT16 根目录读取
	{
		for(cnt=0;cnt<RootDirSectors;cnt++)
		{
			if(SD_ReadSingleBlock(FirstDirSector+cnt,fat_buffer))
```

```
                {
                    FileInfo.F_StartCluster=0;//读数错误
                    return FileInfo;
                }
                for(offset=0;offset<512;offset+=32)
                {
                    item=(direntry *)(&fat_buffer[offset]);//指针转换
                    //找到一个可用的文件
                    if((item->deName[0] != 0x00) && (item->deName[0] != 0xe5))//找到一个
合法文件
                    {
                        if(item->deAttributes != AM_LFN)//忽略长文件名
                        {
                            CopyDirentruyItem(&FileInfo,item);//复制目录项,提取详细信息

                            if(FileInfo.F_Type&type)//找到一个合适的类型了
                            {
                               // //printf("File Name:%s\n",FileInfo.F_Name);

                                //找到了文件,返回这个文件的首簇
                                if(mystrcmp(Name,FileInfo.F_Name))
                                {
                                    return FileInfo;
                                }
                            }
                            LongNameFlag=0;//清空长文件名
                        }
                    }
                }
            }
        }else//其他文件夹/FAT32 系统
        {
            tempclust=cluster;
            while(1)
            {
                sector=fatClustToSect(tempclust);
                for(cnt=0;cnt<SectorsPerClust;cnt++)
                {
                    if(SD_ReadSingleBlock(sector+cnt,fat_buffer))
                    {
                        FileInfo.F_StartCluster=0;//读数错误
                        return FileInfo;
                    }
```

```
                    for(offset=0;offset<512;offset+=32)
                    {
                        item=(direntry *)(&fat_buffer[offset]);
                        if((item->deName[0] != 0x00) && (item->deName[0] != 0xe5))
                        {
                            if(item->deAttributes != AM_LFN) //忽略长文件名
                            {
                                CopyDirentruyItem(&FileInfo,item);//复制目录项,提取详细
信息

                                if(FileInfo.F_Type&type)//找到一个合适的类型了
                                {   /*
                                    //printf("F_Info->F_Name:%s\n",FileInfo.F_Name);
                                    //printf("F_Info->F_Type:%d\n",FileInfo.F_Type);
                                    //printf("F_Info->F_Size:%d\n",FileInfo.F_Size);

    //printf("F_Info->F_StartClusterH:%x\n",FileInfo.F_StartCluster>>8);

    //printf("F_Info->F_StartClusterL:%x\n\n",FileInfo.F_StartCluster&0xff); */
                                    //找到了文件,返回这个文件的首簇
                                    if(mystrcmp(Name,FileInfo.F_Name))
                                    {
                                        return FileInfo;
                                    }
                                }
                                LongNameFlag=0;//清空长文件名
                            }
                        }
                    }
                }
                tempclust=FAT_NextCluster(tempclust);//查找下一个簇号
                if(tempclust==0x0fffffff||tempclust==0x0ffffff8
||(FAT32_Enable==0&&tempclust==0xffff))break;
            }
        }
        FileInfo.F_StartCluster=0;//读数错误
        return FileInfo;
    }
```

此部分就是 FAT 管理部分源码，里面有提到 TINY FAT 表，其实该表的作用就是将某个文件的簇号全部 COPY 出来，放在 SRAM 里面（其核心思想是利用了 FAT 表簇的连续性），从而实现快速的在文件里面前后查找。详见此贴：

http://www.openedv.com/posts/list/0/228.htm?fromAll=0。

保存这部分代码，并在工程里面新增一个组：FAT。然后把 fat.c 加入到这个组下接下来在

fat.h 里面输入如下代码：

```
#ifndef __FAT_H__
#define __FAT_H__
#include <stm32f10x_lib.h>
#include "mmc_sd.h"
//#include "untogb.h"
#include <string.h>
//Mini STM32 开发板
//FAT 驱动 V1.0
//正点原子@ALIENTEK
//重定义区
typedef char    CHAR;
typedef short SHORT;
typedef long    LONG;
typedef unsigned long      DWORD;
typedef int                BOOL;
typedef u8                 BYTE;
typedef unsigned short    WORD;
#define FALSE 0
#define TRUE    1
#define MSDOSFSROOT         0                       // cluster 0 means the root dir
#define CLUST_FREE          0                       // cluster 0 also means a free cluster
#define MSDOSFSFREE         CLUST_FREE
#define CLUST_FIRST         2                       // first legal cluster number
#define CLUST_RSRVD         0xfff6                  // reserved cluster range
#define CLUST_BAD           0xfff7                  // a cluster with a defect
#define CLUST_EOFS          0xfff8                  // start of eof cluster range
#define CLUST_EOFE          0xffff          // end of eof cluster range

//dir_clust:当前目录簇号
//FileInfo: 文件指针
//type:         要浏览的文件类型
//返回值:     当前文件索引
//文件夹中 ".":代表当前目录,"..":代表上一级目录
//文件目录项的文件属性位
#define    AM_RDO      0x01      //只读文件
#define    AM_HID0x02      //隐藏文件
#define    AM_SYS0x04      //系统文件
#define    AM_VOL      0x08      //卷标
#define AM_LFN   0x0F      //长文件名字段
#define AM_DIR    0x10      //文件夹
#define AM_ARC   0x20      //归档文件
```

```
//文件信息结构体
#define T_MP1   1<<0
#define T_MP2   1<<1
#define T_MP3   1<<2
#define T_MP4   1<<3
#define T_M4A   1<<4
#define T_3GP   1<<5
#define T_3G2   1<<6
#define T_OGG   1<<7
#define T_ACC   1<<8
#define T_WMA   1<<9
#define T_WAV   1<<10
#define T_MID   1<<11
#define T_FLAC 1<<12

#define T_LRC   1<<13
#define T_TXT   1<<14
#define T_C     1<<15
#define T_H     1<<16

#define T_FILE 1<<17
#define T_FON   1<<18
#define T_SYS   1<<19

#define T_BMP   1<<20
#define T_JPG   1<<21
#define T_JPEG 1<<22

#define MAX_LONG_NAME_SIZE 80     /* 26*n+2     n=3 */
//文件信息结构体
//包括文件名,文件类型,文件属性,起始簇和偏移量
typedef struct
{
    BYTE F_ShortName[8];//保存八个字节的短文件名
    BYTE F_Name[MAX_LONG_NAME_SIZE];
    unsigned long F_StartCluster;       //< file starting cluster for last file accessed
    unsigned long F_Size;               //< file size for last file accessed
    u8   F_Attr;                        //< file attr for last file accessed
    u32  F_Type;
    //unsigned short CreateTime;        //< file creation time for last file accessed
    //unsigned short CreateDate;        //< file creation date for last file accessed
    unsigned long F_CurClust;           //当前簇号
    unsigned int   F_Offset;            //在簇号里面的扇区偏移
```

```
}FileInfoStruct;

//使用 packed 以字节分配.避免数据对齐的问题
//硬盘分区表,每个分区占用 16 个字节,64 个字节支持最大 4 个分区
//在 SD 卡上一般使用的是一个,也就是第一个分区.
typedef __packed struct    // length 16 bytes
{
    BYTE    prIsActive;         //0x80  表明该分区是否是活动分区
    BYTE    prStartHead;   //开始磁头
    WORD    prStartCylSect;     //开始扇区(低 6 位)和开始柱面(高 10 位)
    BYTE    prPartType;         //系统 ID
    BYTE    prEndHead;          //结束磁头
    WORD    prEndCylSect;//结束扇区和结束柱面
    DWORD prStartLBA;           //相对扇区数,也就是从这里到逻辑扇区 0 的偏移(以扇区为
单位)
    DWORD prSize;               //总扇区数
}partrecord;
//磁盘引导区  MBR 扇区
typedef __packed struct
{
    BYTE    psPartCode[512-64-2];       //MBR 引导程序代码存放区
    BYTE    psPart[64];                 //硬盘分区表  64 个字节
    BYTE    psBootSectSig0;             //2 个字节的有效结束标志
    BYTE    psBootSectSig1;
#define BOOTSIG0            0x55
#define BOOTSIG1            0xaa
}partsector;
//扩展分区
typedef __packed struct
{
    BYTE    exDriveNumber;      // drive number (0x80)//0x00 for floopy disk 0x80 for hard
disk
    BYTE    exReserved1;       // reserved should always set 0
    BYTE    exBootSignature;     // ext. boot signature (0x29)
#define EXBOOTSIG           0x29
    BYTE    exVolumeID[4];          // volume ID number
    BYTE    exVolumeLabel[11]; // volume label "NO NAME"
    BYTE    exFileSysType[8];    // fs type (FAT12 or FAT)
}extboot;
//FAT16 DBR 区,操作系统引导记录区
typedef __packed struct
{
    BYTE    bsJump[3];                  // jump inst E9xxxx or EBxx90
```

```
    BYTE    bsOemName[8];           // OEM name and version
    BYTE    bsBPB[25];              // BIOS parameter block
    BYTE    bsExt[26];              // Bootsector Extension
    BYTE    bsBootCode[448];        // pad so structure is 512b
    BYTE    bsBootSectSig0;         // boot sector signature byte 0x55
    BYTE    bsBootSectSig1;         // boot sector signature byte 0xAA
#define BOOTSIG0        0x55
#define BOOTSIG1        0xaa
}bootsector50;

//FAT 16 BPB
typedef __packed struct
{
        WORD    bpbBytesPerSec; // bytes per sector          //512 1024 2048 or
4096
        BYTE    bpbSecPerClust; // sectors per cluster       // power of 2
        WORD    bpbResSectors;  // number of reserved sectors  //1 is recommend
        BYTE    bpbFATs;        // number of FATs            // 2 is recommend
        WORD    bpbRootDirEnts; // number of root directory entries
        WORD    bpbSectors;     // total number of sectors
        BYTE    bpbMedia;       // media descriptor          //0xf8 match the
fat[0]
        WORD    bpbFATsecs;     // number of sectors per FAT
        WORD    bpbSecPerTrack; // sectors per track
        WORD    bpbHeads;       // number of heads
        DWORD bpbHiddenSecs;    // # of hidden sectors
        DWORD bpbHugeSectors; // # of sectors if bpbSectors == 0
}bpb50;

//FAT32 DBR 区,操作系统引导记录区
typedef __packed struct
{
    BYTE    bsJump[3];          // jump inst E9xxxx or EBxx90
    BYTE    bsOemName[8];       // OEM name and version
    BYTE    bsBPB[53];          // BIOS parameter block
    BYTE    bsExt[26];          // Bootsector Extension
    BYTE    bsBootCode[418];    // pad so structure is 512b
    BYTE    bsBootSectSig2;     // boot sector signature byte 0x00
    BYTE    bsBootSectSig3;     // boot sector signature byte 0x00
    BYTE    bsBootSectSig0;     // boot sector signature byte 0x55
    BYTE    bsBootSectSig1;     // boot sector signature byte 0xAA
#define BOOTSIG0        0x55
#define BOOTSIG1        0xaa
```

```
#define BOOTSIG2            0x00
#define BOOTSIG3            0x00
}bootsector710;

//FAT 32 BPB
typedef __packed struct
{
        WORD    bpbBytesPerSec;     // bytes per sector
        BYTE    bpbSecPerClust;     // sectors per cluster
        WORD    bpbResSectors;      // number of reserved sectors
        BYTE    bpbFATs;        // number of FATs
        WORD    bpbRootDirEnts;     // number of root directory entries
        WORD    bpbSectors;         // total number of sectors
        BYTE    bpbMedia;           // media descriptor
        WORD    bpbFATsecs;         // number of sectors per FAT
        WORD    bpbSecPerTrack;     // sectors per track
        WORD    bpbHeads;           // number of heads
        DWORD bpbHiddenSecs;    // # of hidden sectors
// 3.3 compat ends here
        DWORD bpbHugeSectors;   // # of sectors if bpbSectors == 0
// 5.0 compat ends here
        DWORD       bpbBigFATsecs;// like bpbFATsecs for FAT32
        WORD        bpbExtFlags;  // extended flags:
#define FATNUM      0xf             // mask for numbering active FAT
#define FATMIRROR 0x80              // FAT is mirrored (like it always was)
        WORD        bpbFSVers;    // filesystem version
#define FSVERS      0               // currently only 0 is understood
        DWORD       bpbRootClust;     // start cluster for root directory
        WORD        bpbFSInfo;    // filesystem info structure sector
        WORD        bpbBackup;    // backup boot sector
        // There is a 12 byte filler here, but we ignore it
}bpb710;

// 文件结构体
typedef __packed struct
{
        BYTE        deName[8];          // filename, blank filled
#define SLOT_EMPTY          0x00                // slot has never been used
#define SLOT_E5             0x05                // the real value is 0xE5
#define SLOT_DELETED        0xE5                // file in this slot deleted
#define SLOT_DIR        0x2E                // a directorymmm
        BYTE        deExtension[3];     // extension, blank filled
        BYTE        deAttributes;       // file attributes
```

```
#define ATTR_NORMAL         0x00                // 普通文件
#define ATTR_READONLY       0x01                // 只读文件
#define ATTR_HIDDEN         0x02                // 隐藏文件
#define ATTR_SYSTEM         0x04                // 系统文件
#define ATTR_VOLUME         0x08                // 卷标
#define ATTR_LONG_FILENAME      0x0F            // 长文件名标志
#define ATTR_DIRECTORY      0x10                // 文件夹文件
#define ATTR_ARCHIVE        0x20                // 新的或者归档文件
        BYTE        deLowerCase;        // NT VFAT lower case flags    (set to zero)
#define LCASE_BASE          0x08                // filename base in lower case
#define LCASE_EXT           0x10                // filename extension in lower case
        BYTE        deCHundredth;       // hundredth of seconds in CTime
        BYTE        deCTime[2];         // create time
        BYTE        deCDate[2];         // create date
        BYTE        deADate[2];         // access date
        WORD        deHighClust;        // high bytes of cluster number
        BYTE        deMTime[2];         // last update time
        BYTE        deMDate[2];         // last update date
        WORD        deStartCluster;     // starting cluster of file
        DWORD       deFileSize;         // size of file in bytes
}direntry;

// number of directory entries in one sector
#define DIRENTRIES_PER_SECTOR 0x10      //when the bpbBytesPerSec=512

// Structure of a Win95 long name directory entry
typedef __packed struct
{
        BYTE        weCnt;              //
#define WIN_LAST            0x40
#define WIN_CNT             0x3f
        BYTE        wePart1[10];
        BYTE        weAttributes;
#define ATTR_WIN95          0x0f
        BYTE        weReserved1;
        BYTE        weChksum;
        BYTE        wePart2[12];
        WORD        weReserved2;
        BYTE        wePart3[4];
}winentry;

//tinyFAT 表结构
//利用 FAT 表的一致性,一般后一个簇比前一个簇只是大一.
```

```
//将 FAT 表压缩成小的 tinyFAT 表
typedef __packed struct
{
    #define Fat_Table_Size 10          //tinyFAT 表大小
    #define Fat_Head_Size   10         //tinyFAT 表头大小
    u8   Fat_Head_Pos;                 //tinyFAT 表的上一个表头位置
    u32 Fat_Base_Head[Fat_Head_Size];//tinyFAT 表的  表头数组
    u32 Fat_Base_Tab[Fat_Table_Size];//文件的 tinyFAT 基址表
    u16 Fat_Base_Len[Fat_Table_Size];//基址偏移量
    u8   Fat_Over;                     //文件的 tinyFAT 表是否全部读出标记位
}FAT_TABLE;
#define WIN_ENTRY_CHARS    13          // Number of chars per winentry

// Maximum filename length in Win95
// Note: Must be < sizeof(dirent.d_name)
#define WIN_MAXLEN          255

// This is the format of the contents of the deTime field in the direntry
// structure.
// We don't use bitfields because we don't know how compilers for
// arbitrary machines will lay them out.
#define DT_2SECONDS_MASK            0x1F       // seconds divided by 2
#define DT_2SECONDS_SHIFT           0
#define DT_MINUTES_MASK             0x7E0      // minutes
#define DT_MINUTES_SHIFT            5
#define DT_HOURS_MASK               0xF800     // hours
#define DT_HOURS_SHIFT              11

// This is the format of the contents of the deDate field in the direntry
// structure.
#define DD_DAY_MASK                 0x1F       // day of month
#define DD_DAY_SHIFT                0
#define DD_MONTH_MASK               0x1E0      // month
#define DD_MONTH_SHIFT              5
#define DD_YEAR_MASK                0xFE00   // year - 1980
#define DD_YEAR_SHIFT               9
//外部函数可能用到的数据
extern BYTE FAT32_Enable;  //FAT32 文件系统标志
extern DWORD FirstDirClust;    //根目录簇号
extern DWORD Cur_Dir_Cluster;//当前目录簇号
extern DWORD Fat_Dir_Cluster;//父目录簇号  在 FAT 文件夹里面 CopyDirentruyItem 函数中修改!
extern WORD SectorsPerClust;
```

```
extern WORD BytesPerSector;
extern FileInfoStruct F_Info[3];
extern u8 fat_buffer[512];//FAT 文件系统操作缓冲区
//FAT 原有的函数
unsigned char FAT_Init(void);//初始化
unsigned long FAT_NextCluster(unsigned long cluster);//查找下一簇号
u32 fatClustToSect(u32 cluster);//将簇号转换为扇区号
u8 Get_File_Info(u32 dir_clust,FileInfoStruct *FileInfo,u32 type,u16 *count);//查找文件
void F_Open(FileInfoStruct *FileInfo);//打开文件
unsigned char F_Read(FileInfoStruct *FileInfo,u8 *buf);//读文件，size=0 代表整个文件
unsigned long FAT_OpenDir(BYTE * dir);//打开目录
FileInfoStruct F_Search(u32 cluster,unsigned char *Name,u32 type);//查找指定名字的文件

u32 FatTab_Next_Cluster(unsigned long cluster);
u32 FatTab_Prev_Cluster(unsigned long cluster);
void Copy_Fat_Table(unsigned long cluster);
#endif
```

此部分代码我们不多介绍了，保存该文件。解下来我们打开 fontupd.c，在该文件内输入如下代码：

```
#include "fontupd.h"
#include "sys.h"
#include "fat.h"
#include "flash.h"
#include "usart.h"
#include "delay.h"
#include "lcd.h"
//Mini STM32 开发板
//中文汉字支持程序 V1.1
//包括字体更新,以及字库首地址获取 2 个函数.
//正点原子@ALIENTEK

//以下下字段一定不要乱改!!!
//字节 0~3,  记录 UNI2GBKADDR;字节 4~7  ,记录 UNI2GBKADDR 的大小
//字节 8~11, 记录 FONT16ADDR ;字节 12~15,记录 FONT16ADDR 的大小
//字节 16~19,记录 FONT12ADDR ;字节 20~23,记录 FONT12ADDR 的大小
//字节 24,用来存放字库是否存在的标志位,0XAA,表示存在字库;其他值,表示字库不存在!

//系统文件夹
const unsigned char *folder[2]=
{
"SYSTEM",
"FONT",
};
```

```
//系统文件定义
const unsigned char *sysfile[3]=
{
"GBK16.FON",
"GBK12.FON",
"UNI2GBK.SYS",
};
//字节 0~3,  记录 UNI2GBKADDR;字节 4~7  ,记录 UNI2GBKADDR 的大小
//字节 8~11,  记录 FONT16ADDR ;字节 12~15,记录 FONT16ADDR 的大小
//字节 16~19,记录 FONT12ADDR ;字节 20~23,记录 FONT12ADDR 的大小
//字体存放,从 100K 处开始
//100K,存放 UNICODE2GBK 的转换码

u32 FONT16ADDR ;//16 字体存放的地址
u32 FONT12ADDR ;//12 字体存放的地址
u32 UNI2GBKADDR;//UNICODE TO GBK  表存放地址

//更新字体文件
//返回值:0,更新成功;
//          其他,错误代码.
//正点原子@ALIENTEK
//V1.1
#ifdef EN_UPDATE_FONT
u8 temp[512];   //零时空间
u8 Update_Font(void)
{
    u32 fcluster=0;
    u32 i;
    //u8 temp[512];   //零时空间  在这里定义,会内存溢出
    u32 tempsys[2];   //临时记录文件起始位置和文件大小
    float prog;
    u8 t;
    FileInfoStruct FileTemp;//零时文件夹
    //得到根目录的簇号
    if(FAT32_Enable)fcluster=FirstDirClust;
    else fcluster=0;
    FileTemp=F_Search(fcluster,(unsigned char *)folder[0],T_FILE);//查找 system 文件夹
    if(FileTemp.F_StartCluster==0)return 1;                    //系统文件夹丢失
    {
        //先查找字体
        FileTemp=F_Search(FileTemp.F_StartCluster,(unsigned char *)folder[1],T_FILE);//在
system 文件夹下查找 FONT 文件夹
        if(FileTemp.F_StartCluster==0)return 2;//字体文件夹丢失
```

```
        fcluster=FileTemp.F_StartCluster;           //字体文件夹簇号
        FileTemp=F_Search(fcluster,(unsigned char *)sysfile[2],T_SYS);//在 system 文件夹下
查找 SYS 文件
        if(FileTemp.F_StartCluster==0)return 3;//FONT12 字体文件丢失
        LCD_ShowString(20,90,"Write UNI2GBK to FLASH...");
        LCD_ShowString(108,110,"%");
        F_Open(&FileTemp);//打开该文件
        i=0;
        while(F_Read(&FileTemp,temp))//成功读出 512 个字节
        {
            if(i<FileTemp.F_Size)//不超过文件大小
            {
                SPI_Flash_Write(temp,i+100000,512);//从 100K 字节处开始写入 512 个数
据
                i+=512;//增加 512 个字节
            }
            prog=(float)i/FileTemp.F_Size;
            prog*=100;
            if(t!=prog)
            {
                t=prog;
                if(t>100)t=100;
                LCD_ShowNum(84,110,t,3,16);//显示数值
            }
        }
        UNI2GBKADDR=100000;//UNI2GBKADDR 从 100K 处开始写入.
        tempsys[0]=UNI2GBKADDR;
        tempsys[1]=FileTemp.F_Size;                 //UNI2GBKADDR 大小
        SPI_Flash_Write((u8*)tempsys,0,8);//记录在地址 0~7 处

        delay_ms(1000);
        //printf("UNI2GBK 写入 FLASH 完毕!\n");
        //printf("写入数据长度:%d\n",FileTemp.F_Size);
        //printf("UNI2GBKSADDR:%d\n\n",UNI2GBKADDR);

        FONT16ADDR=FileTemp.F_Size+UNI2GBKADDR;//F16 的首地址
        FileTemp=F_Search(fcluster,(unsigned char *)sysfile[0],T_FON);//在 system 文件夹
下查找 FONT16 字体文件
        if(FileTemp.F_StartCluster==0)return 4;//FONT16 字体文件丢失

        LCD_ShowString(20,90,"Write FONT16 to FLASH... ");
        //printf("开始 FONT16 写入 FLASH...\n");
```

```
            F_Open(&FileTemp);//打开该文件
            i=0;
            while(F_Read(&FileTemp,temp))//成功读出 512 个字节
            {
                if(i<FileTemp.F_Size)//不超过文件大小
                {
                    SPI_Flash_Write(temp,i+FONT16ADDR,512);//从 0 开始写入 512 个数据
                    i+=512;//增加 512 个字节
                }
                prog=(float)i/FileTemp.F_Size;
                prog*=100;
                if(t!=prog)
                {
                    t=prog;
                    if(t>100)t=100;
                    LCD_ShowNum(84,110,t,3,16);//显示数值
                }
            }
            tempsys[0]=FONT16ADDR;
            tempsys[1]=FileTemp.F_Size;                 //FONT16ADDR 大小
            SPI_Flash_Write((u8*)tempsys,8,8);//记录在地址 8~15 处
            delay_ms(1000);
            //printf("FONT16 写入 FLASH 完毕!\n");
            //printf("写入数据长度:%d\n",FileTemp.F_Size);
            FONT12ADDR=FileTemp.F_Size+FONT16ADDR;//F16 的首地址
        }
        t=0XAA;
        SPI_Flash_Write(&t,24,1);//写入字库存在标志 0XAA
        LCD_ShowString(20,90,"    Font Update Successed    ");
        delay_ms(1000);
        delay_ms(1000);
        return 0;//成功
}
#endif

//用这个函数得到字体地址
//在系统使用汉字支持之前,必须调用该函数
//包括 FONT16ADDR,FONT12ADDR,UNI2GBKADDR;
u8 Font_Init(void)
{
        u32 tempsys[2];//临时记录文件起始位置和文件大小
        u8 fontok=0;
        SPI_Flash_Read(&fontok,24,1);//读出字库标志位,看是否存在字库
```

```
    if(fontok!=0XAA)return 1;
    SPI_Flash_Read((u8*)tempsys,0,8);//读出 8 个字节
    UNI2GBKADDR=tempsys[0];
    //printf("tempsysgbk[0]:%d\n",tempsys[0]);
    //printf("tempsysgbk[1]:%d\n",tempsys[1]);

    SPI_Flash_Read((u8*)tempsys,8,8);//读出 8 个字节
    //printf("tempsysf16[0]:%d\n",tempsys[0]);
    //printf("tempsysf16[1]:%d\n",tempsys[1]);
    FONT16ADDR=tempsys[0];

    SPI_Flash_Read((u8*)tempsys,16,8);//读出 8 个字节
    //printf("tempsysf12[0]:%d\n",tempsys[0]);
    //printf("tempsysf12[1]:%d\n",tempsys[1]);
    FONT12ADDR=tempsys[0];
    return 0;
}
```

此部分代码主要用于字库的更新操作，即将 SD 卡上的字库文件 COPY 到 W25X16，这里面用到了 W25X16 部分的函数，我们在 W25X16 里面划分了前 100K 字节为常用数据区，不用来写字库，而从 100K 字节处开始写入字库。在常用数据区的地址 0~23 用来存储字字库/转换码表的首地址，而第 24 个字节用来存放是否存在字库的标记。

保存该部分代码，并在工程里面新建一个 TEXT 的组，把 fontupd.c 加入到这个组里面，然后打开 fontupd.h 在该文件里面输入如下代码：

```
#ifndef __FONTUPD_H__
#define __FONTUPD_H__
#include <stm32f10x_lib.h>
//Mini STM32 开发板
//中文汉字支持程序 V1.1
//正点原子@ALIENTEK
//2010/5/23
#define EN_UPDATE_FONT //使能字体更新,通过关闭这里实现禁止字库更新
extern u32 FONT16ADDR ;
extern u32 FONT12ADDR ;
extern u32 UNI2GBKADDR;
u8 Update_Font(void);//更新字库
u8 Font_Init(void);//初始化字库
#endif
```

保存此部分代码，然后打开 text.c 文件，在该文件里面输入如下代码：

```
#include "sys.h"
#include "fontupd.h"
#include "flash.h"
#include "lcd.h"
#include "text.h"
```

```
//Mini STM32 开发板
//文本显示程序 V1.1
//正点原子@ALIENTEK
//////////////////////////////////////////////常用函数/////////////////////////////////////
//code 字符指针开始
//从字库中查找出字模
//code 字符串的开始地址,ascii 码
//mat 数据存放地址 size*2 bytes 大小
//正点原子@HYW
//CHECK:09/10/30
void Get_HzMat(unsigned char *code,unsigned char *mat,u8 size)
{
	unsigned char qh,ql;
	unsigned char i;
	unsigned long foffset;
	qh=*code;
	ql=*(++code);
	if(qh<0x81||ql<0x40||ql==0xff||qh==0xff)//非 常用汉字
	{
		for(i=0;i<(size*2);i++)*mat++=0x00;//填充满格
		return; //结束访问
	}
	if(ql<0x7f)ql-=0x40;//注意!
	else ql-=0x41;
	qh-=0x81;
	foffset=((unsigned long)190*qh+ql)*(size*2);//得到字库中的字节偏移量

	if(size==16)SPI_Flash_Read(mat,foffset+FONT16ADDR,32);
	else SPI_Flash_Read(mat,foffset+FONT12ADDR,24);
}
//显示一个指定大小的汉字
//x,y :汉字的坐标
//font:汉字 GBK 码
//size:字体大小
//mode:0,正常显示,1,叠加显示
//正点原子@HYW
//CHECK:09/10/30
void Show_Font(u8 x,u8 y,u8 *font,u8 size,u8 mode)
{
	u8 temp,t,t1;
	u8 y0=y;
	u8 dzk[32];
	u16 tempcolor;
```

```
Get_HzMat(font,dzk,size);//得到相应大小的点阵数据
if(mode==0)//正常显示
{
    for(t=0;t<size*2;t++)
    {
        temp=dzk[t];//得到 12 数据
        for(t1=0;t1<8;t1++)
        {
            if(temp&0x80)LCD_DrawPoint(x,y);
            else
            {
                tempcolor=POINT_COLOR;
                POINT_COLOR=BACK_COLOR;
                LCD_DrawPoint(x,y);
                POINT_COLOR=tempcolor;//还原
            }
            temp<<=1;
            y++;
            if((y-y0)==size)
            {
                y=y0;
                x++;
                break;
            }
        }
    }
}else//叠加显示
{
    for(t=0;t<size*2;t++)
    {
        temp=dzk[t];//得到 12 数据
        for(t1=0;t1<8;t1++)
        {
            if(temp&0x80)LCD_DrawPoint(x,y);
            temp<<=1;
            y++;
            if((y-y0)==size)
            {
                y=y0;
                x++;
                break;
            }
        }
```

```
            }
        }
    }
    //在指定位置开始显示一个字符串
    //支持自动换行
    //(x,y):起始坐标
    //str    :字符串
    //size :字体大小
    //mode:0,非叠加方式;1,叠加方式
    //正点原子@HYW
    //CHECK:09/10/30
    void Show_Str(u8 x,u8 y,u8*str,u8 size,u8 mode)
    {
        u8 bHz=0;        //字符或者中文

        while(*str!=0)//数据未结束
        {
            if(!bHz)
            {
                if(*str>0x80)bHz=1;//中文
                else                        //字符
                {
                    if(x>(LCD_W-size/2))//换行
                    {
                        y+=size;
                        x=0;
                    }
                    if(y>(LCD_H-size))break;//越界返回
                    if(*str==13)//换行符号
                    {
                        y+=size;
                        x=0;
                        str++;
                    }
                    else LCD_ShowChar(x,y,*str,size,mode);//有效部分写入
                    str++;
                    x+=size/2; //字符,为全字的一半
                }
            }else//中文
            {
                bHz=0;//有汉字库
                if(x>(LCD_W-size))//换行
                {
```

```
                y+=size;
                x=0;
            }
            if(y>(LCD_H-size))break;//越界返回
            Show_Font(x,y,str,size,mode); //显示这个汉字,空心显示
            str+=2;
            x+=size;//下一个汉字偏移
        }
    }
}
//得到字符串的长度
//正点原子@HYW
//CHECK OK 091118
u16 my_strlen(u8*str)
{
    u16 len=0;
    while(*str!='\0')
    {
        str++;
        len++;
    }
    return len;
}
//在 str1 后面加入 str2
//正点原子@HYW
//CHECK OK 091118
void my_stradd(u8*str1,u8*str2)
{
    while(*str1!='\0')str1++;
    while(*str2!='\0')
    {
        *str1=*str2;
        str2++;
        str1++;
    }
    *str1='\0';//加入结束符
}
//在指定宽度的中间显示字符串
//如果字符长度超过了 len,则用 Show_Str 显示
//len:指定要显示的宽度
void Show_Str_Mid(u8 x,u16 y,u8*str,u8 size,u8 len)
{
    u16 strlenth=0;
```

```
        strlenth=my_strlen(str);
        strlenth*=size/2;
        if(strlenth>len)Show_Str(x,y,str,size,1);
        else
        {
            strlenth=(len-strlenth)/2;
            Show_Str(strlenth+x,y,str,size,1);
        }
    }
```

此部分代码用于显示汉字，其实就是根据我们之前建立字库的规则，把汉字从字库里面还原出来，前面已经详细介绍了其原理，这里我们也不介绍了，保存此部分代码，并把 text.c 文件加入 TEXT 组下。打开 text.h，在该文件下输入如下代码：

```
#ifndef __TEXT_H__
#define __TEXT_H__
#include <stm32f10x_lib.h>
//Mini STM32 开发板
//文本显示程序 V1.1
//正点原子@ALIENTEK
void Get_HzMat(unsigned char *code,unsigned char *mat,u8 size);//得到汉字的点阵码
void Show_Font(u8 x,u8 y,u8 *font,u8 size,u8 mode);//在指定位置显示一个汉字
void Show_Str(u8 x,u8 y,u8*str,u8 size,u8 mode);//在指定位置显示一个字符串
void Show_Str_Mid(u8 x,u16 y,u8*str,u8 size,u8 len);
void Show_PTStr(u8 x,u8 y,u8*str,u8 size,u8 mode,u8 len,u16 offx);//显示部分字符
void Str_Scroll(u8 x,u8 y,u8*str,u8 size,u8 mode,u8 len,u8 start);//滚动字符串
void my_stradd(u8*str1,u8*str2);//将 str2 与 str1 相加,结果保存在 str1
#endif
```

这些主要是函数申明，我们不多说了，保存此部分代码。然后打开 untogb.h，在该文件下输入如下内容：

```
#include "untogb.h"
#include "flash.h"
#include "usart.h"
#include "fontupd.h"
//Mini STM32 开发板
//UNICODE TO GBK 内码转换程序 V1.1
//正点原子@ALIENTEK
//将 UNICODE 码转换为 GBK 码
//unicode:UNICODE 码
//返回值:GBK 码
u16 UnicodeToGBK(u16 unicode)//用二分查找算法
{
    u32 offset;
    u8 temp[2];
    u16 res;
```

```
    if(unicode<=0X9FA5)offset=unicode-0X4E00;
    else if(unicode>0X9FA5)//是标点符号
    {
        if(unicode<0XFF01||unicode>0XFF61)return 0;//没有对应编码
        offset=unicode-0XFF01+0X9FA6-0X4E00;
    }
    SPI_Flash_Read(temp,offset*2+UNI2GBKADDR,2);//得到 GBK 码
    res=temp[0];
    res<<=8;
    res+=temp[1];
    return res ; //返回找到的编码
}
//将 pbuf 内的 unicode 码转为 gbk 码.
//pbuf:unicode 码存储区,同时也是 gbk 码的输出区.必须小于 80 个字节.
//代码转换 unit code-> GBK
//正点原子@HYW
//CHECK:09/10/30
void UniToGB(u8 *pbuf)
{
    unsigned int    code;
    unsigned char i,m=0;
    for(i=0;i<80;i++)//最长 80 个字符
    {
        code= pbuf[i*2+1]*256+pbuf[i*2];
        if((code==0)||(code==0xffff))break;
        if((code&0xff00)==0)//字母
        {
            if((code>=0x20)&&(code<=0x7e))
            {
                pbuf[m++]=(unsigned char)code;
            }else pbuf[m++]='?';//无法识别的用？代替
            continue;
        }
        if(code>=0X4E00)//是汉字
        {
            code=UnicodeToGBK(code);//把 unicode 转换为 gb2312
            pbuf[m++]=code>>8;
            pbuf[m++]=(u8)code;
        }else pbuf[m++]='?';//无法识别的用？代替
    }
    pbuf[m]='\0';//添加结束符号
}
```

此部分代码用于将 UNICODE 码的内码转换为 GBK 码的内码，也是按照前面介绍的方法

来做的，保存此部分代码，并加入到 TEXT 组下。打开 untogb.h，在该文件下输入如下内容：

```
#ifndef __UNTOGB_H__
#define __UNTOGB_H__
#include <stm32f10x_lib.h>
//Mini STM32 开发板
//UNICODE TO GBK  内码转换程序  V1.1
//正点原子@ALIENTEK
//2010/5/23
void  UniToGB(u8  *pbuf);//将 PBUF 内的 UNICODE 码转为 GBK 码.pbuf 必须小于 80 个字节
#endif
```

此部分就不介绍了。接下来就是最后一步了，我们在 test.c 里面修改 main 函数如下：

```
int main(void)
{
    u32 fontcnt;
    u8 i,j;
    u8 fontx[2];//gbk 码
    u8 key,t;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LCD_Init();                 //初始化液晶
    KEY_Init();                 //按键初始化
    LED_Init();               //LED 初始化
    SPI_Flash_Init();    //SPI FLASH 初始化
    if(Font_Init())//字库不存在,则更新字库
    {
UPD:
        POINT_COLOR=RED;
        LCD_Clear(WHITE);
        LCD_ShowString(60,50,"Mini STM32");
        LCD_ShowString(60,70,"Font Updating...");
        //字体更新
        SD_Init();              //初始化 SD 卡
        while(FAT_Init())//FAT  错误
        {
            LCD_ShowString(60,90,"FAT SYS ERROR");
            i= SD_Init();
            if(i)//SD 卡初始化
            {
                LCD_ShowString(60,110,"SD_CARD ERROR");
            }
            delay_ms(500);
```

```
            LCD_Fill(60,90,240,126,WHITE);//清除显示
            delay_ms(500);
            LED0=!LED0;
        }
        LCD_Fill(60,90,240,126,WHITE);//清除显示
        key=Update_Font();
        while(key!=0)//字体更新出错
        {
            key=Update_Font();
            printf("key:%d\n",key);
            LCD_ShowString(60,90,"SYSTEM FILE LOST");
            delay_ms(500);
            LCD_ShowString(60,90,"Please Check....");
            delay_ms(500);
            LED0=!LED0;
        };
        LCD_Clear(WHITE);
    }
    POINT_COLOR=RED;
    Show_Str(60,50,"Mini STM32 开发板",16,0);
    Show_Str(60,70,"GBK 字库测试程序",16,0);
    Show_Str(60,90,"正点原子@ALIENTEK",16,0);
    Show_Str(60,110,"2011 年 1 月 1 日",16,0);
    Show_Str(60,130,"按 KEY0,更新字库",16,0);
    POINT_COLOR=BLUE;
    Show_Str(60,150,"内码高字节:",16,0);
    Show_Str(60,170,"内码低字节:",16,0);
    Show_Str(60,190,"对应汉字为:",16,0);
    Show_Str(60,210,"汉字计数器:",16,0);
    while(1) //遍历 GBK 字库
    {
        fontcnt=0;
        for(i=0x81;i<0xff;i++)
        {
            fontx[0]=i;
            LCD_ShowNum(148,150,i,3,16);//显示内码高字节
            for(j=0x40;j<0xfe;j++)
            {
                if(j==0x7f)continue;
                fontcnt++;
                LCD_ShowNum(148,170,j,3,16);//显示内码低字节
                LCD_ShowNum(148,210,fontcnt,5,16);//显示内码低字节
                fontx[1]=j;
```

```
                    Show_Font(148,190,fontx,16,0);
                    t=200;
                    while(t--)//延时,同时扫描按键
                    {
                        delay_ms(1);
                        key=KEY_Scan();
                        if(key==1)goto UPD;
                    }
                    LED0=!LED0;
                }
            }
        }
    }
```

此部分代码就实现了我们在硬件描述部分所描述的功能，至此整个软件设计就完成了。这节有太多的代码，而且工程也增加了不少，我们来看看工程的截图吧：

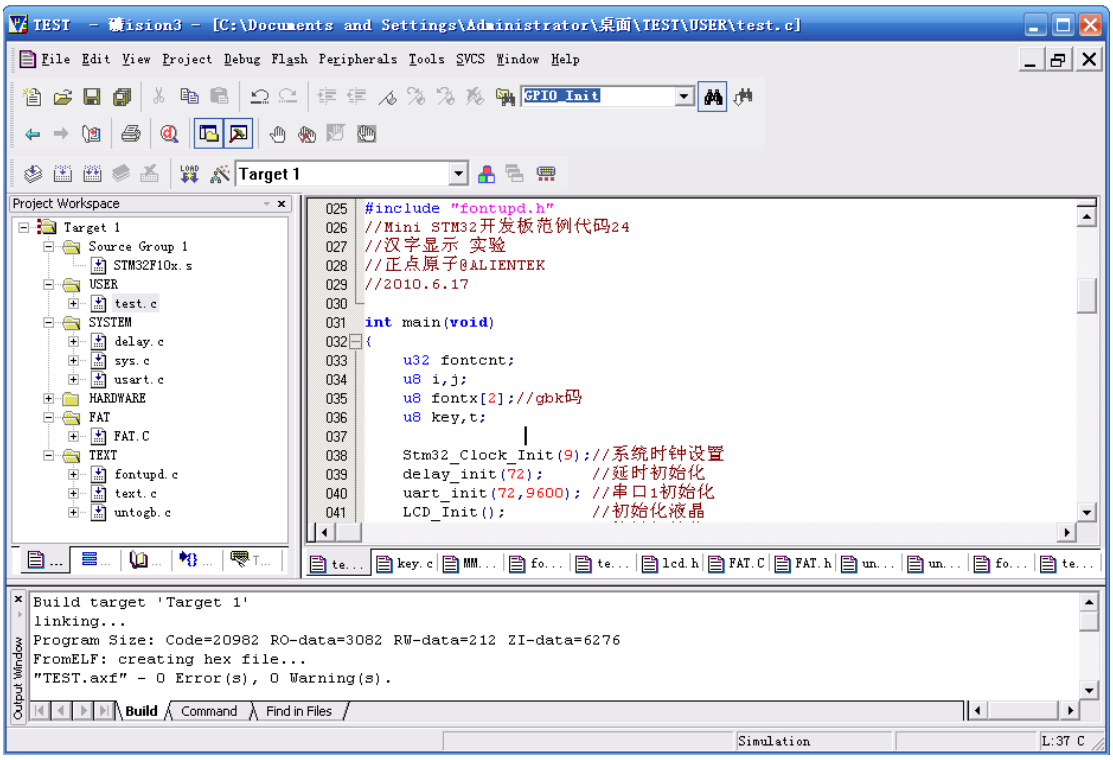


图 3.24.3.1 工程建成截图

3.24.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 开始显示汉字及汉字内码，如下图所示：

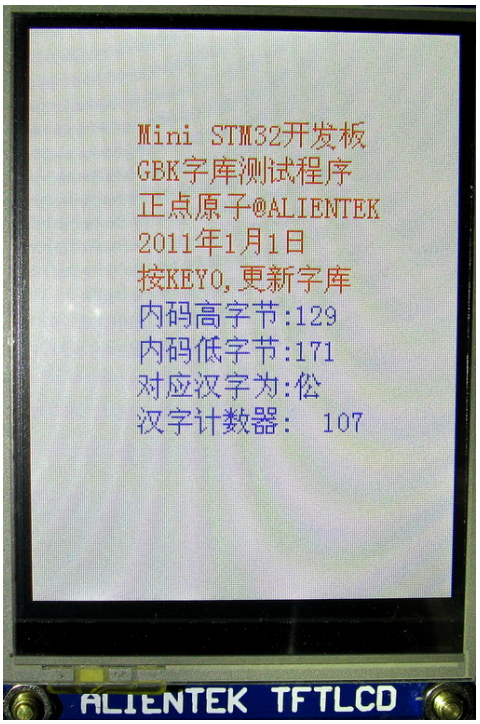


图 3.24.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为 ALIENTEK MiniSTM32 开发板在出厂的时候都是测试过的，里面刷了综合测试程序，已经把字库写入到了 W25X16 里面，所以并不会提示更新字库。如果你想要更新字库，那么则必须先找一张 SD 卡，把我们提供的 SYSTEM 文件夹 COPY 到 SD 卡根目录下，重启程序。然后，在显示汉字的时候，按下 KEY0，就可以开始更新字库了。

字库更新界面如下图所示：

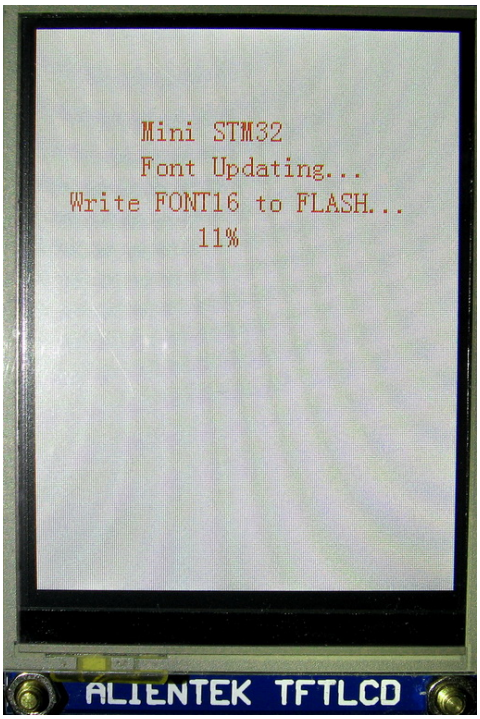


图 3.24.4.2 汉字字库更新界面

3.25 图片显示实验

数码相框日渐流行，数码相框显示的图片一般为 BMP/JPG/JPEG 等格式，其实用我们的 MiniSTM32 也可以显示这些图片，本节，我们将向大家介绍如何在 ALIENTEK MiniSTM32 开发板上显示 BMP/JPG/JPEG 等格式的图片。本节分为如下几个部分：

3.25.1 图片显示原理简介

3.25.2 硬件设计

3.25.3 软件设计

3.25.4 下载与测试

3.25.1 图片显示原理简介

BMP 是一种与硬件设备无关的图像文件格式，使用非常广。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。BMP 文件的图像深度可选 lbit、4bit、8bit、16bit、24bit 及 32bit。BMP 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

典型的 BMP 图像文件由三部分组成：位图文件头数据结构，它包含 BMP 图像文件的类型、显示内容等信息；位图信息数据结构，它包含有 BMP 图像的宽、高、压缩方法，以及定义颜色等信息。

JPEG 是 Joint Photographic Experts Group(联合图像专家组)的缩写，文件后辍名为"．jpg"或"．jpeg"，是最常用的图像文件格式，由一个软件开发联合会组织制定，是一种有损压缩格式，能够将图像压缩在很小的储存空间，图像中重复或不重要的资料会被丢失，因此容易造成图像数据的损伤。尤其是使用过高的压缩比例，将使最终解压缩后恢复的图像质量明显降低，如果追求高品质图像，不宜采用过高压缩比例。但是 JPEG 压缩技术十分先进，它用有损压缩方式去除冗余的图像数据，在获得极高的压缩率的同时能展现十分丰富生动的图像，换句话说，就是可以用最少的磁盘空间得到较好的图像品质。

而且 JPEG 是一种很灵活的格式，具有调节图像质量的功能，允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在 10：1 到 40：1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好。比如可以把 1.37Mb 的 BMP 位图文件压缩至 20.3KB。当然也可以在图像质量和文件尺寸之间找到平衡点。JPEG 格式压缩的主要是高频信息，对色彩的信息保留较好，适合应用于互联网，可减少图像的传输时间，可以支持 24bit 真彩色，也普遍应用于需要连续色调的图像。

JPEG/JPG 的解码过程可以简单的概述为如下几个部分：

1、从文件头读出文件的相关信息。

JPEG 文件数据分为文件头和图像数据两大部分，其中文件头记录了图像的版本、长宽、采样因子、量化表、哈夫曼表等重要信息。所以解码前必须将文件头信息读出，以备图像数据解码过程之用。

2、从图像数据流读取一个最小编码单元(MCU) ，并提取出里边的各个颜色分量单元。

3、将颜色分量单元从数据流恢复成矩阵数据。

利用文件头给出的哈夫曼表，对分割出来的颜色分量单元进行解码，把其恢复成 8×8 的数据矩阵。

4、8×8 的数据矩阵进一步解码。

此部分解码工作以 8×8 的数据矩阵为单位， 其中包括相邻矩阵的直流系数差分解码、利用文件头给出的量化表反量化数据、反 Zig- zag 编码、隔行正负纠正、反向离散余弦变换等 5 个步骤， 最终输出仍然是一个 8×8 的数据矩阵。

5、颜色系统 YCrCb 向 RGB 转换。

将一个 MCU 的各个颜色分量单元解码结果整合起来，将图像颜色系统从 YCrCb 向 RGB 转换。

6、排列整合各个 MCU 的解码数据。

不断读取数据流中的 MCU 并对其解码，直至读完所有 MCU 为止，将各 MCU 解码后的数据正确排列成完整的图像。

JPEG 的解码本身是比较复杂的，在这里一时半会也说不清，更详细的介绍，请大家参考

光盘图片解码的相关资料。

3.25.2 硬件设计

本节实验功能简介：开机的时候先检测 SD 卡是否存在，然后初始化 FAT 文件系统，在这之后开始查找根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片，循环显示，通过按 KEY0 和 KEY1 可以快速浏览下一张和上一张。如果未找到图片文件夹/图片，则提示错误。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0（外部 LED0）。

3）TFTLCD 液晶模块。

4）KEY0，KEY1。

5）SD 卡。

这几部分，在之前的实例中都介绍过了，我们在此就不介绍了。需要注意的是，我们在 SD 卡根目录下要建一个 PICTURE 的文件夹，用来存放 JPEG、JPG、BMP 等图片（不是所有的 JPEG、JPG 图片都能打开，如果不能打开，则用 XP 自带的画图工具保存一下，再放到 PICTURE 文件夹下就可以打开了）。

3.25.3 软件设计

打开上一节的工程，首先在 HARDWARE 文件夹所在文件夹下新建一个 SYSFILE 和 JPEG 的文件夹。在 JPEG 文件夹里面新建 jpegdecode.c 和 jpegbmp.h 两个文件。然后 SYSFILE 文件夹下新建 sysfile.c 和 sysfile.h 两个文件。并将这个两个文件夹加入头文件包含路径。

打开 jpegdecode.c 文件，在里面输入如下代码：

```
#include "jpegbmp.h"
#include "lcd.h"
//Mini STM32 开发板
//JPG/JPEG/BMP 图片显示  代码
//正点原子@ALIENTEK
////////////////////////////////////////////////
//全局变量声明,BMP 和 JPEG 共用
FileInfoStruct *CurFile;//当前解码/操作的文件

//图像信息
typedef struct
{
    u32 ImgWidth; //图像的实际宽度和高度
    u32 ImgHeight;

    u32 Div_Fac;   //缩放系数  (扩大了 10000 倍的)

    u32 S_Height; //设定的高度和宽度
    u32 S_Width;
```

```
    u32  S_XOFF;    //x 轴和 y 轴的偏移量
    u32 S_YOFF;

    u32 staticx;   //当前显示到的 x  y 坐标
    u32 staticy;
}PIC_POS;

PIC_POS PICINFO;//图像位置信息

////////////////////////////////////////////////////
void AI_Drow_Init(void);            //智能画图,初始化.得到比例因子 PICINFO.Div_Fac
////////////////////////////////////////////////////
//在 JPEG 函数里面用到的变量
short                SampRate_Y_H,SampRate_Y_V;
short                SampRate_U_H,SampRate_U_V;
short                SampRate_V_H,SampRate_V_V;
short                H_YtoU,V_YtoU,H_YtoV,V_YtoV;
short                Y_in_MCU,U_in_MCU,V_in_MCU;
unsigned char     *lp;//取代 lpJpegBuf
short                qt_table[3][64];
short                comp_num;
u8              comp_index[3];
u8              YDcIndex,YAcIndex,UVDcIndex,UVAcIndex;
u8              HufTabIndex;
short                *YQtTable,*UQtTable,*VQtTable;
short                code_pos_table[4][16],code_len_table[4][16];
unsigned short code_value_table[4][256];
unsigned short huf_max_value[4][16],huf_min_value[4][16];
short                BitPos,CurByte;//byte 的第几位,当前 byte
short                rrun,vvalue;
short                MCUBuffer[10*64];
short                QtZzMCUBuffer[10*64];
short                BlockBuffer[64];
short                ycoef,ucoef,vcoef;
BOOL                 IntervalFlag;
short                interval=0;
short                Y[4*64],U[4*64],V[4*64];//
DWORD                sizei,sizej;
short                restart;
long iclip[1024];//4k BYTES
long *iclp;
//反 Z 字形编码表
```

```
const int Zig_Zag[8][8]={{0,1,5,6,14,15,27,28},
                         {2,4,7,13,16,26,29,42},
                         {3,8,12,17,25,30,41,43},
                         {9,11,18,24,31,40,44,53},
                         {10,19,23,32,39,45,52,54},
                         {20,22,33,38,46,51,55,60},
                         {21,34,37,47,50,56,59,61},
                         {35,36,48,49,57,58,62,63}
                        };
const BYTE And[9]={0,1,3,7,0xf,0x1f,0x3f,0x7f,0xff};
//数据缓冲区
unsigned char jpg_buffer[1024];//数据缓存区
//////////////////////////////////////////////////
//初始化智能画点
void AI_Drow_Init(void)
{
    float temp,temp1;
    temp=(float)PICINFO.S_Width/PICINFO.ImgWidth;
    temp1=(float)PICINFO.S_Height/PICINFO.ImgHeight;
    if(temp<temp1)temp1=temp;//取较小的那个
    if(temp1>1)temp1=1;
    //使图片处于所给区域的中间
    PICINFO.S_XOFF+=(PICINFO.S_Width-temp1*PICINFO.ImgWidth)/2;
    PICINFO.S_YOFF+=(PICINFO.S_Height-temp1*PICINFO.ImgHeight)/2;
    temp1*=10000;//扩大 10000 倍
    PICINFO.Div_Fac=temp1;
    PICINFO.staticx=500;
    PICINFO.staticy=500;//放到一个不可能的值上面
}
//判断这个像素是否可以显示
//(x,y) :像素原始坐标
//chg    :功能变量.
//返回值:0,不需要显示.1,需要显示
__inline u8 IsElementOk(u16 x,u16 y,u8 chg)
{
    if(x!=PICINFO.staticx||y!=PICINFO.staticy)
    {
        if(chg==1)
        {
            PICINFO.staticx=x;
            PICINFO.staticy=y;
        }
        return 1;
```

```
    }
    else return 0;
}
//智能画图
//FileName:要显示的图片文件  BMP/JPG/JPEG
//(sx,sy) :开始显示的坐标点
//(ex,ey) :结束显示的坐标点
//图片在开始和结束的坐标点范围内显示
BOOL AI_LoadPicFile(FileInfoStruct *FileName,u16 sx,u16 sy,u16 ex,u16 ey)
{
    int  funcret;//返回值
    //得到显示方框大小
    if(ey>sy)PICINFO.S_Height=ey-sy;
    else PICINFO.S_Height=sy-ey;
    if(ex>sx)PICINFO.S_Width=ex-sx;
    else PICINFO.S_Width=sx-ex;
    //显示区域无效
    if(PICINFO.S_Height==0||PICINFO.S_Width==0)
    {
        PICINFO.S_Height=LCD_H;
        PICINFO.S_Width=LCD_W;
        return FALSE;
    }
    //影响速度
    //SD_Init();//初始化 SD 卡，在意外拔出之后可以正常使用
    //显示的开始坐标点
    PICINFO.S_YOFF=sy;
    PICINFO.S_XOFF=sx;
    //文件名传递
    CurFile=FileName;
    if(CurFile->F_Type==T_BMP)//得到一个 BMP 图像
    {
        funcret=BmpDecode(CurFile); //得到一个 BMP 图像
        return funcret;
    }
    else if(CurFile->F_Type==T_JPG||CurFile->F_Type==T_JPEG)//得到 JPG/JPEG 图片
    {
        //得到 JPEG/JPG 图片的开始信息
        F_Open(CurFile);
        //开始时读入 1024 个字节到缓存里面.方便后面提取 JPEG 解码的信息
        F_Read(CurFile,jpg_buffer);      //读第一次
        F_Read(CurFile,jpg_buffer+512);//读第二次
```

```
            InitTable(); //初始化各个数据表
            if((funcret=InitTag())!=FUNC_OK)return FALSE;//初始化表头不成功
            if((SampRate_Y_H==0)||(SampRate_Y_V==0))return FALSE ;//采样率错误
            AI_Drow_Init();   //初始化 PICINFO.Div_Fac,启动智能画图
            funcret=Decode();//解码 JPEG 开始
        }else return FALSE;   //非图片格式!!!
        if(funcret==FUNC_OK)return TRUE;//解码成功
        else return FALSE;     //解码失败
}
//解码这个 BMP 文件
BOOL BmpDecode(FileInfoStruct *BmpFileName)
{
        u16 count;
        u8   rgb ,color_byte;
        u16 x ,y,color,tmp_color ;
        u16 uiTemp;        //x 轴方向像素计数器
        u16 countpix=0;//记录像素
        //x,y 的实际坐标
        u8   realx=0;
        u16 realy=0;
        u8   yok=1;
        BITMAPINFO *pbmp;//临时指针
        CurFile=BmpFileName;
        F_Open(CurFile);//打开文件
        F_Read(CurFile,jpg_buffer);//读出 512 个字节
        pbmp=(BITMAPINFO*)jpg_buffer;//得到 BMP 的头部信息
        count=pbmp->bmfHeader.bfOffBits;              //数据偏移,得到数据段的开始地址
        color_byte=pbmp->bmiHeader.biBitCount/8;//彩色位 16/24/32
        PICINFO.ImgHeight=pbmp->bmiHeader.biHeight;//得到图片高度
        PICINFO.ImgWidth=pbmp->bmiHeader.biWidth;   //得到图片宽度
        //水平像素必须是 4 的倍数!!
        if((PICINFO.ImgWidth*color_byte)%4)
            uiTemp=((PICINFO.ImgWidth*color_byte)/4+1)*4;
        else
            uiTemp=PICINFO.ImgWidth*color_byte;
        AI_Drow_Init();//初始化智能画图
        //开始解码 BMP
        x =0 ;
        y=PICINFO.ImgHeight;
        rgb=0;
        realy=y*PICINFO.Div_Fac/10000;
        while(1)
        {
```

```
while(count<512)   //读取一簇 512 扇区  (SectorsPerClust  每簇扇区数)
{
    if(color_byte==3)     //24 位颜色图
    {
        switch (rgb)
        {
            case 0:
                tmp_color = jpg_buffer[count]>>3 ;
                color |= tmp_color;
                break ;
            case 1:
                tmp_color = jpg_buffer[count]>>2 ;
                tmp_color <<= 5 ;
                color |= tmp_color ;
                break;
            case 2 :
                tmp_color = jpg_buffer[count]>>3 ;
                tmp_color <<= 11 ;
                color |= tmp_color ;
                break ;
        }
    }
    else
    {
        if(color_byte==2)   //16 位颜色图
        {
            switch(rgb)
            {
                case 0 :
                    tmp_color = jpg_buffer[count] ;
                    break ;
                case 1 :
                    color = jpg_buffer[count] ;
                    color<<=8 ;
                    color |= tmp_color ;
                    break ;
            }
        }
        else
        {
            if(color_byte==4)//32 位颜色图
            {
                switch (rgb)
```

```
                    {
                        case 0 :
                            tmp_color=jpg_buffer[count];
                            color|=tmp_color>>3;
                            break ;
                        case 1 :
                            tmp_color=jpg_buffer[count];
                            tmp_color>>=2;
                            color|=tmp_color<<5;
                            break ;
                        case 2 :
                            tmp_color=jpg_buffer[count];
                            tmp_color>>=3;
                            color|=tmp_color<<11;
                            break ;
                        case 3 :break ;
                    }
                }
            }
        }//位图颜色得到
        rgb++;
        count++ ;
        if(rgb==color_byte) //水平方向读取到 1 像素数数据后显示
        {
            if(x<PICINFO.ImgWidth)
            {
                realx=x*PICINFO.Div_Fac/10000;//x 轴实际值
                if(IsElementOk(realx,realy,1)&&yok)//符合条件
                {
                    POINT_COLOR=color;

LCD_DrawPoint(realx+PICINFO.S_XOFF,realy+PICINFO.S_YOFF-1);
                }
            }
            x++;//x 轴增加一个像素
            color=0x00;
            rgb=0;
        }
        countpix++;//像素累加
        if(countpix>=uiTemp)//水平方向像素值到了.换行
        {
            y--;
            if(y<=0)return TRUE;
```

```
                realy=y*PICINFO.Div_Fac/10000;//实际 y 值改变
                if(IsElementOk(realx,realy,0))yok=1;//此处不改变 PICINFO.staticx,y 的值

                else yok=0;
                x=0;
                countpix=0;
                color=0x00;
                rgb=0;
            }
        }
        if(!F_Read(CurFile,jpg_buffer))break;//读出 512 个字节,读数失败时自动退出
        count=0 ;
    }
    return TRUE;//BMP 显示结束.
}
//对指针地址进行改变!
//pc      :当前指针
//返回值:当前指针的减少量.在 d_buffer 里面自动进行了偏移
unsigned int P_Cal(unsigned char*pc)
{
    unsigned short cont=0;//计数器
    unsigned long buffer_val=0;   //寄存区首地址
    unsigned long point_val=0;    //指针所指的当前地址

    unsigned char secoff;
    unsigned short t;
    unsigned char *p;
    p=jpg_buffer+512;//偏移到中间

    point_val=(unsigned long)pc;//得到当前指针所指地址
    buffer_val=(unsigned long)&jpg_buffer;//得到缓存区首地址
    cont=point_val-buffer_val;//得到两者之差
    if(cont>=512)//数据超过了中间
    {
        secoff=cont/512;//超出了多少 secoff 个 512 字节
        while(secoff)    //读取 secoff 次 512 个字节
        {
            for(t=0;t<512;t++)jpg_buffer[t]=p[t];//复制后 512 个字节 给前 512 个字节

            if(!F_Read(CurFile,p))//读取 512 个字节到 d_buffer 的后半部分
            {//读取结束了
                //printf("read Fail!\n");
                break;//读数失败!break;
```

```
                }
                secoff--;
            }
        }
        return cont-cont%512;//指针地址缩减
    }
    //初始化 d_buffer 的数据
    int InitTag(void)
    {
        BOOL finish=FALSE;
        u8 id;
        short   llength;
        short   i,j,k;
        short   huftab1,huftab2;
        short   huftabindex;
        u8 hf_table_index;
        u8 qt_table_index;
        u8 comnum;//最长为 256 个字节

        unsigned char   *lptemp;
        short   colorount;

        lp=jpg_buffer+2;//跳过两个字节 SOI(0xFF，0xD8 Start of Image)
        lp-=P_Cal(lp);
        while (!finish)
        {
            id=*(lp+1);//取出低位字节(高位在前，低位在后)
            lp+=2;        //跳过取出的字节
            lp-=P_Cal(lp);
            switch (id)
            {
                case M_APP0: //JFIF APP0 segment marker (0xE0)
                                //标志应用数据段的开始
                    llength=MAKEWORD(*(lp+1),*lp);//得到应用数据段长度
                    lp+=llength;
                    lp-=P_Cal(lp);
                    break;
                case M_DQT: //定义量化表标记(0xFF,0xDB)
                    llength=MAKEWORD(*(lp+1),*lp);//(量化表长度)两个字节
                    qt_table_index=(*(lp+2))&0x0f;//量化表信息 bit 0..3: QT 号(0..3, 否则错
误)
                                                    //bit 4..7: QT 精度, 0 = 8 bit, 否则 16
bit
```

```
                lptemp=lp+3;                        //n 字节的 QT, n = 64*(精度+1)
                //d_buffer 里面至少有有 512 个字节的余度,这里最大用到 128 个字节
                if(llength<80)                      //精度为 8 bit
                {
                    for(i=0;i<64;i++)qt_table[qt_table_index][i]=(short)*(lptemp++);
                }
                else                                //精度为 16 bit
                {
                    for(i=0;i<64;i++)qt_table[qt_table_index][i]=(short)*(lptemp++);
                    qt_table_index=(*(lptemp++))&0x0f;
                    for(i=0;i<64;i++)qt_table[qt_table_index][i]=(short)*(lptemp++);
                }
                lp+=llength;  //跳过量化表
                lp-=P_Cal(lp);
                break;
            case M_SOF0:                            // 帧 开 始 (baseline JPEG
0xFF,0xC0)
                llength=MAKEWORD(*(lp+1),*lp);      //长度 (高字节, 低字节),
8+components*3
                PICINFO.ImgHeight=MAKEWORD(*(lp+4),*(lp+3));//图片高度 (高字节,
低字节), 如果不支持 DNL 就必须 >0
                PICINFO.ImgWidth=MAKEWORD(*(lp+6),*(lp+5));//图片宽度 (高字节,
低字节), 如果不支持 DNL 就必须 >0
                comp_num=*(lp+7);//components 数量(1 u8), 灰度图是 1, YCbCr/YIQ
彩色图是 3, CMYK 彩色图是 4

                if((comp_num!=1)&&(comp_num!=3))return FUNC_FORMAT_ERROR;//
格式错误
                if(comp_num==3)                     //YCbCr/YIQ 彩色图
                {
                    comp_index[0]=*(lp+8);      //component id (1 = Y, 2 = Cb, 3 = Cr, 4 =
I, 5 = Q)
                    SampRate_Y_H=(*(lp+9))>>4;  //水平采样系数
                    SampRate_Y_V=(*(lp+9))&0x0f;//垂直采样系数
                    YQtTable=(short *)qt_table[*(lp+10)];//通过量化表号取得量化表地
址

                    comp_index[1]=*(lp+11);             //component id
                    SampRate_U_H=(*(lp+12))>>4;         //水平采样系数
                    SampRate_U_V=(*(lp+12))&0x0f;       //垂直采样系数
                    UQtTable=(short *)qt_table[*(lp+13)];//通过量化表号取得量化表地
址
```

```
                comp_index[2]=*(lp+14);                    //component id
                SampRate_V_H=(*(lp+15))>>4;                //水平采样系数
                SampRate_V_V=(*(lp+15))&0x0f;              //垂直采样系数
                VQtTable=(short  *)qt_table[*(lp+16)];//通过量化表号取得量化表地
址
            }
            else                                           //component id
            {
                comp_index[0]=*(lp+8);
                SampRate_Y_H=(*(lp+9))>>4;
                SampRate_Y_V=(*(lp+9))&0x0f;
                YQtTable=(short *)qt_table[*(lp+10)];//灰度图的量化表都一样

                comp_index[1]=*(lp+8);
                SampRate_U_H=1;
                SampRate_U_V=1;
                UQtTable=(short *)qt_table[*(lp+10)];

                comp_index[2]=*(lp+8);
                SampRate_V_H=1;
                SampRate_V_V=1;
                VQtTable=(short *)qt_table[*(lp+10)];
            }
            lp+=llength;
            lp-=P_Cal(lp);
            break;
        case M_DHT: //定义哈夫曼表(0xFF,0xC4)
            llength=MAKEWORD(*(lp+1),*lp);//长度  (高字节,  低字节)
            if (llength<0xd0)                  // Huffman Table 信息  (1 u8)
            {
                huftab1=(short)(*(lp+2))>>4;        //huftab1=0,1(HT  类型,0 = DC 1 =
AC)
                huftab2=(short)(*(lp+2))&0x0f;      //huftab2=0,1(HT  号    ,0 = Y    1 =
UV)
                huftabindex=huftab1*2+huftab2;     //0 = YDC 1 = UVDC 2 = YAC 3 =
UVAC
                lptemp=lp+3;//!!!
                //在这里可能出现余度不够,多于 512 字节,则会导致出错!!!!
                for (i=0; i<16; i++)                    //16 bytes:  长度是  1..16  代码的符
号数
                    code_len_table[huftabindex][i]=(short)(*(lptemp++));//码长为 i 的
码字个数
                j=0;
```

```
                    for (i=0; i<16; i++)                //得出 HT 的所有码字的对应值
                    {
                        if(code_len_table[huftabindex][i]!=0)
                        {
                            k=0;
                            while(k<code_len_table[huftabindex][i])
                            {

code_value_table[huftabindex][k+j]=(short)(*(lptemp++));//最可能的出错地方
                                k++;
                            }
                            j+=k;
                        }
                    }
                    i=0;
                    while (code_len_table[huftabindex][i]==0)i++;
                    for (j=0;j<i;j++)
                    {
                        huf_min_value[huftabindex][j]=0;
                        huf_max_value[huftabindex][j]=0;
                    }
                    huf_min_value[huftabindex][i]=0;
                    huf_max_value[huftabindex][i]=code_len_table[huftabindex][i]-1;
                    for (j=i+1;j<16;j++)
                    {

huf_min_value[huftabindex][j]=(huf_max_value[huftabindex][j-1]+1)<<1;

huf_max_value[huftabindex][j]=huf_min_value[huftabindex][j]+code_len_table[huftabindex][j]-
1;
                    }
                    code_pos_table[huftabindex][0]=0;
                    for (j=1;j<16;j++)

code_pos_table[huftabindex][j]=code_len_table[huftabindex][j-1]+code_pos_table[huftabindex][
j-1];
                        lp+=llength;
                        lp-=P_Cal(lp);
                }//if
                else
                {
                    hf_table_index=*(lp+2);
                    lp+=2;
```

```
lp-=P_Cal(lp);
while (hf_table_index!=0xff)
{
    huftab1=(short)hf_table_index>>4;       //huftab1=0,1
    huftab2=(short)hf_table_index&0x0f;     //huftab2=0,1
    huftabindex=huftab1*2+huftab2;
    lptemp=lp+1;
    colorount=0;
    for (i=0; i<16; i++)
    {
        code_len_table[huftabindex][i]=(short)(*(lptemp++));
        colorount+=code_len_table[huftabindex][i];
    }
    colorount+=17;
    j=0;
    for (i=0; i<16; i++)
    {
        if(code_len_table[huftabindex][i]!=0)
        {
            k=0;
            while(k<code_len_table[huftabindex][i])
            {

code_value_table[huftabindex][k+j]=(short)(*(lptemp++));//最可能出错的地方,余度不够
                k++;
            }
            j+=k;
        }
    }
    i=0;
    while (code_len_table[huftabindex][i]==0)i++;
    for (j=0;j<i;j++)
    {
        huf_min_value[huftabindex][j]=0;
        huf_max_value[huftabindex][j]=0;
    }
    huf_min_value[huftabindex][i]=0;
    huf_max_value[huftabindex][i]=code_len_table[huftabindex][i]-1;
    for (j=i+1;j<16;j++)
    {

huf_min_value[huftabindex][j]=(huf_max_value[huftabindex][j-1]+1)<<1;
```

```
huf_max_value[huftabindex][j]=huf_min_value[huftabindex][j]+code_len_table[huftabindex][j]-
1;
                                    }
                                    code_pos_table[huftabindex][0]=0;
                                    for (j=1;j<16;j++)

code_pos_table[huftabindex][j]=code_len_table[huftabindex][j-1]+code_pos_table[huftabindex][
j-1];
                                    lp+=colorount;
                                    lp-=P_Cal(lp);
                                    hf_table_index=*lp;
                                }   //while
                            }   //else
                            break;
                        case M_DRI://定义差分编码累计复位的间隔
                            llength=MAKEWORD(*(lp+1),*lp);
                            restart=MAKEWORD(*(lp+3),*(lp+2));
                            lp+=llength;
                            lp-=P_Cal(lp);
                            break;
                        case M_SOS:   //扫描开始 12 字节
                            llength=MAKEWORD(*(lp+1),*lp);
                            comnum=*(lp+2);
                            if(comnum!=comp_num)return FUNC_FORMAT_ERROR; //格式错误
                            lptemp=lp+3;//这里也可能出现错误
                            //这里也可能出错,但是几率比较小了
                            for (i=0;i<comp_num;i++)//每组件的信息
                            {
                                if(*lptemp==comp_index[0])
                                {
                                    YDcIndex=(*(lptemp+1))>>4;     //Y 使用的 Huffman 表
                                    YAcIndex=((*(lptemp+1))&0x0f)+2;
                                }
                                else
                                {
                                    UVDcIndex=(*(lptemp+1))>>4;     //U,V
                                    UVAcIndex=((*(lptemp+1))&0x0f)+2;
                                }
                                lptemp+=2;//comp_num<256,但是 2*comp_num+3 可能>=512
                            }
                            lp+=llength;
                            lp-=P_Cal(lp);
                            finish=TRUE;
```

```
                    break;
                case M_EOI:return FUNC_FORMAT_ERROR;//图片结束 标记
                default:
                    if ((id&0xf0)!=0xd0)
                    {
                        llength=MAKEWORD(*(lp+1),*lp);
                        lp+=llength;
                        lp-=P_Cal(lp);
                    }
                    else lp+=2;
                    break;
            }   //switch
        } //while
        return FUNC_OK;
}
//初始化量化表，全部清零
void InitTable(void)
{
        short i,j;
        sizei=sizej=0;
        PICINFO.ImgWidth=PICINFO.ImgHeight=0;
        rrun=vvalue=0;
        BitPos=0;
        CurByte=0;
        IntervalFlag=FALSE;
        restart=0;
        for(i=0;i<3;i++) //量化表
            for(j=0;j<64;j++)
                qt_table[i][j]=0;
        comp_num=0;
        HufTabIndex=0;
        for(i=0;i<3;i++)
            comp_index[i]=0;
        for(i=0;i<4;i++)
            for(j=0;j<16;j++){
                code_len_table[i][j]=0;
                code_pos_table[i][j]=0;
                huf_max_value[i][j]=0;
                huf_min_value[i][j]=0;
            }
        for(i=0;i<4;i++)
            for(j=0;j<256;j++)
                code_value_table[i][j]=0;
```

```
        for(i=0;i<10*64;i++){
            MCUBuffer[i]=0;
            QtZzMCUBuffer[i]=0;
        }
        for(i=0;i<64;i++){
            Y[i]=0;
            U[i]=0;
            V[i]=0;
            BlockBuffer[i]=0;
        }
        ycoef=ucoef=vcoef=0;
    }
    //调用顺序: Initialize_Fast_IDCT() :初始化
    //              DecodeMCUBlock()          Huffman Decode
    //              IQtIZzMCUComponent()    反量化、反 DCT
    //              GetYUV()                  Get Y U V
    //              StoreBuffer()             YUV to RGB
    int Decode(void)
    {
        int funcret;

        Y_in_MCU=SampRate_Y_H*SampRate_Y_V;//YDU YDU YDU YDU
        U_in_MCU=SampRate_U_H*SampRate_U_V;//cRDU
        V_in_MCU=SampRate_V_H*SampRate_V_V;//cBDU
        H_YtoU=SampRate_Y_H/SampRate_U_H;
        V_YtoU=SampRate_Y_V/SampRate_U_V;
        H_YtoV=SampRate_Y_H/SampRate_V_H;
        V_YtoV=SampRate_Y_V/SampRate_V_V;
        Initialize_Fast_IDCT();
        while((funcret=DecodeMCUBlock())==FUNC_OK) //After Call DecodeMCUBUBlock()
        {
            interval++;                                    //The Digital has been Huffman
Decoded and
            if((restart)&&(interval % restart==0))//be stored in
MCUBuffer(YDU,YDU,YDU,YDU
                IntervalFlag=TRUE;                    // UDU,VDU) Every DU := 8*8
            else
                IntervalFlag=FALSE;
            IQtIZzMCUComponent(0); //反量化  and IDCT The Data in QtZzMCUBuffer
            IQtIZzMCUComponent(1);
            IQtIZzMCUComponent(2);
            GetYUV(0);                  //得到 Y cR cB
```

```
            GetYUV(1);
            GetYUV(2);
            StoreBuffer();          //To RGB
            sizej+=SampRate_Y_H*8;
            if(sizej>=PICINFO.ImgWidth)
            {
                sizej=0;
                sizei+=SampRate_Y_V*8;
            }
            if ((sizej==0)&&(sizei>=PICINFO.ImgHeight))break;
        }
        return funcret;
}
// 入口 QtZzMCUBuffer 出口 Y[] U[] V[]
//得到 YUV 色彩空间
void   GetYUV(short flag)
{
        short       H,VV;
        short       i,j,k,h;
        short       *buf;
        short       *pQtZzMCU;
        switch(flag)
        {
            case 0://亮度分量
                H=SampRate_Y_H;
                VV=SampRate_Y_V;
                buf=Y;
                pQtZzMCU=QtZzMCUBuffer;
                break;
            case 1://红色分量
                H=SampRate_U_H;
                VV=SampRate_U_V;
                buf=U;
                pQtZzMCU=QtZzMCUBuffer+Y_in_MCU*64;
                break;
            case 2://蓝色分量
                H=SampRate_V_H;
                VV=SampRate_V_V;
                buf=V;
                pQtZzMCU=QtZzMCUBuffer+(Y_in_MCU+U_in_MCU)*64;
                break;
        }
        for (i=0;i<VV;i++)
```

```
            for(j=0;j<H;j++)
                for(k=0;k<8;k++)
                    for(h=0;h<8;h++)
                        buf[(i*8+k)*SampRate_Y_H*8+j*8+h]=*pQtZzMCU++;
    }
    //将解出的字按 RGB 形式存储  lpbmp (BGR),(BGR) .....入口 Y[] U[] V[]  出口 lpPtr
    void StoreBuffer(void)
    {
        short i=0,j=0;
        unsigned char R,G,B;
        int y,u,v,rr,gg,bb;
        u16 color;
        //x,y 的实际坐标
        u16 realx=sizej;
        u16 realy=0;

        for(i=0;i<SampRate_Y_V*8;i++)
        {
            if((sizei+i)<PICINFO.ImgHeight)// sizei 表示行  sizej  表示列
            {
                realy=PICINFO.Div_Fac*(sizei+i)/10000;//实际 Y 坐标

                //在这里不改变 PICINFO.staticx 和 PICINFO.staticy 的值 ,如果在这里改变,则会造成每块的第一个点不显示!!!
                if(!IsElementOk(realx,realy,0))continue;//列值是否满足条件? 寻找满足条件的列


                for(j=0;j<SampRate_Y_H*8;j++)
                {
                    if((sizej+j)<PICINFO.ImgWidth)
                    {
                        realx=PICINFO.Div_Fac*(sizej+j)/10000;//实际 X 坐标
                        //在这里改变 PICINFO.staticx 和 PICINFO.staticy 的值
                        if(!IsElementOk(realx,realy,1))continue;//列值是否满足条件? 寻找满足条件的行

                        y=Y[i*8*SampRate_Y_H+j];
                        u=U[(i/V_YtoU)*8*SampRate_Y_H+j/H_YtoU];
                        v=V[(i/V_YtoV)*8*SampRate_Y_H+j/H_YtoV];
                        rr=((y<<8)+18*u+367*v)>>8;
                        gg=((y<<8)-159*u-220*v)>>8;
                        bb=((y<<8)+411*u-29*v)>>8;
                        R=(unsigned char)rr;
```

```
                    G=(unsigned char)gg;
                    B=(unsigned char)bb;
                    if (rr&0xffffff00) if (rr>255) R=255; else if (rr<0) R=0;
                    if (gg&0xffffff00) if (gg>255) G=255; else if (gg<0) G=0;
                    if (bb&0xffffff00) if (bb>255) B=255; else if (bb<0) B=0;
                    color=R>>3;
                    color=color<<6;
                    color |=(G>>2);
                    color=color<<5;
                    color |=(B>>3);
                    //在这里送给 LCD 显示
                    POINT_COLOR=color;

LCD_DrawPoint(realx+PICINFO.S_XOFF,realy+PICINFO.S_YOFF);//显示图片
                }
                else break;
            }
        }
        else break;
    }
}
//Huffman Decode    MCU 出口 MCUBuffer   入口 Blockbuffer[   ]
int DecodeMCUBlock(void)
{
    short *lpMCUBuffer;
    short i,j;
    int funcret;
    if (IntervalFlag)//差值复位
    {
        lp+=2;
        lp-=P_Cal(lp);
        ycoef=ucoef=vcoef=0;
        BitPos=0;
        CurByte=0;
    }
    switch(comp_num)
    {
        case 3:     //comp_num 指图的类型（彩色图、灰度图）
            lpMCUBuffer=MCUBuffer;
            for (i=0;i<SampRate_Y_H*SampRate_Y_V;i++)   //Y
            {
                funcret=HufBlock(YDcIndex,YAcIndex);//解码 4 * (8*8)
                if (funcret!=FUNC_OK)
```

```
                return funcret;
            BlockBuffer[0]=BlockBuffer[0]+ycoef;//直流分量是差值，所以要累加。
            ycoef=BlockBuffer[0];
            for (j=0;j<64;j++)
                *lpMCUBuffer++=BlockBuffer[j];
        }
        for (i=0;i<SampRate_U_H*SampRate_U_V;i++)    //U
        {
            funcret=HufBlock(UVDcIndex,UVAcIndex);
            if (funcret!=FUNC_OK)
                return funcret;
            BlockBuffer[0]=BlockBuffer[0]+ucoef;
            ucoef=BlockBuffer[0];
            for (j=0;j<64;j++)
                *lpMCUBuffer++=BlockBuffer[j];
        }
        for (i=0;i<SampRate_V_H*SampRate_V_V;i++)    //V
        {
            funcret=HufBlock(UVDcIndex,UVAcIndex);
            if (funcret!=FUNC_OK)
                return funcret;
            BlockBuffer[0]=BlockBuffer[0]+vcoef;
            vcoef=BlockBuffer[0];
            for (j=0;j<64;j++)
                *lpMCUBuffer++=BlockBuffer[j];
        }
        break;
    case 1:    //Gray Picture
        lpMCUBuffer=MCUBuffer;
        funcret=HufBlock(YDcIndex,YAcIndex);
        if (funcret!=FUNC_OK)
            return funcret;
        BlockBuffer[0]=BlockBuffer[0]+ycoef;
        ycoef=BlockBuffer[0];
        for (j=0;j<64;j++)
            *lpMCUBuffer++=BlockBuffer[j];
        for (i=0;i<128;i++)
            *lpMCUBuffer++=0;
        break;
    default:
        return FUNC_FORMAT_ERROR;
}
return FUNC_OK;
```

```
}
//Huffman Decode （8*8） DU 出口 Blockbuffer[ ] 入口 vvalue
int HufBlock(u8 dchufindex,u8 achufindex)
{
    short count=0;
    short i;
    int funcret;
    //dc
    HufTabIndex=dchufindex;
    funcret=DecodeElement();
    if(funcret!=FUNC_OK)return funcret;
    BlockBuffer[count++]=vvalue;//解出的直流系数
    //ac
    HufTabIndex=achufindex;
    while (count<64)
    {
        funcret=DecodeElement();
        if(funcret!=FUNC_OK)
            return funcret;
        if ((rrun==0)&&(vvalue==0))
        {
            for (i=count;i<64;i++)BlockBuffer[i]=0;
            count=64;
        }
        else
        {
            for (i=0;i<rrun;i++)BlockBuffer[count++]=0;//前面的零
            BlockBuffer[count++]=vvalue;//解出的值
        }
    }
    return FUNC_OK;
}
//Huffman 解码 每个元素 出口 vvalue 入口 读文件 ReadByte
int DecodeElement()
{
    int thiscode,tempcode;
    unsigned short temp,valueex;
    short codelen;
    u8 hufexbyte,runsize,tempsize,sign;
    u8 newbyte,lastbyte;

    if(BitPos>=1) //BitPos 指示当前比特位置
    {
```

```
            BitPos--;
            thiscode=(u8)CurByte>>BitPos;//取一个比特
            CurByte=CurByte&And[BitPos];    //清除取走的比特位
        }
        else                                //取出的一个字节已用完
        {                                       //新取
            lastbyte=ReadByte();            //读出一个字节
            BitPos--;                       //and[]:=0x0,0x1,0x3,0x7,0xf,0x1f,0x2f,0x3f,0x4f
            newbyte=CurByte&And[BitPos];
            thiscode=lastbyte>>7;
            CurByte=newbyte;
        }
        codelen=1;
        //与 Huffman 表中的码字匹配，直自找到为止
        while ((thiscode<huf_min_value[HufTabIndex][codelen-1])||
               (code_len_table[HufTabIndex][codelen-1]==0)||
               (thiscode>huf_max_value[HufTabIndex][codelen-1]))
        {
            if(BitPos>=1)//取出的一个字节还有
            {
                BitPos--;
                tempcode=(u8)CurByte>>BitPos;
                CurByte=CurByte&And[BitPos];
            }
            else
            {
                lastbyte=ReadByte();
                BitPos--;
                newbyte=CurByte&And[BitPos];
                tempcode=(u8)lastbyte>>7;
                CurByte=newbyte;
            }
            thiscode=(thiscode<<1)+tempcode;
            codelen++;
            if(codelen>16)return FUNC_FORMAT_ERROR;
        }  //while
        temp=thiscode-huf_min_value[HufTabIndex][codelen-1]+code_pos_table[HufTabIndex][co
delen-1];
        hufexbyte=(u8)code_value_table[HufTabIndex][temp];
        rrun=(short)(hufexbyte>>4);   //一个字节中，高四位是其前面的零的个数。
        runsize=hufexbyte&0x0f;             //后四位为后面字的尺寸
        if(runsize==0)
        {
```

```
            vvalue=0;
            return FUNC_OK;
        }
        tempsize=runsize;
        if(BitPos>=runsize)
        {
            BitPos-=runsize;
            valueex=(u8)CurByte>>BitPos;
            CurByte=CurByte&And[BitPos];
        }
        else
        {
            valueex=CurByte;
            tempsize-=BitPos;
            while(tempsize>8)
            {
                lastbyte=ReadByte();
                valueex=(valueex<<8)+(u8)lastbyte;
                tempsize-=8;
            }   //while
            lastbyte=ReadByte();
            BitPos-=tempsize;
            valueex=(valueex<<tempsize)+(lastbyte>>BitPos);
            CurByte=lastbyte&And[BitPos];
        }   //else
        sign=valueex>>(runsize-1);
        if(sign)vvalue=valueex;//解出的码值
        else
        {
            valueex=valueex^0xffff;
            temp=0xffff<<runsize;
            vvalue=-(short)(valueex^temp);
        }
        return FUNC_OK;
}
//反量化 MCU 中的每个组件     入口  MCUBuffer  出口  QtZzMCUBuffer
void IQtIZzMCUComponent(short flag)
{
        short H,VV;
        short i,j;
        short *pQtZzMCUBuffer;
        short   *pMCUBuffer;
```

```
    switch(flag){
    case 0:
        H=SampRate_Y_H;
        VV=SampRate_Y_V;
        pMCUBuffer=MCUBuffer;
        pQtZzMCUBuffer=QtZzMCUBuffer;
        break;
    case 1:
        H=SampRate_U_H;
        VV=SampRate_U_V;
        pMCUBuffer=MCUBuffer+Y_in_MCU*64;
        pQtZzMCUBuffer=QtZzMCUBuffer+Y_in_MCU*64;
        break;
    case 2:
        H=SampRate_V_H;
        VV=SampRate_V_V;
        pMCUBuffer=MCUBuffer+(Y_in_MCU+U_in_MCU)*64;
        pQtZzMCUBuffer=QtZzMCUBuffer+(Y_in_MCU+U_in_MCU)*64;
        break;
    }
    for(i=0;i<VV;i++)
        for (j=0;j<H;j++)
            IQtIZzBlock(pMCUBuffer+(i*H+j)*64,pQtZzMCUBuffer+(i*H+j)*64,flag);
}
//要量化的字
//反量化  8*8 DU
void IQtIZzBlock(short    *s ,short * d,short flag)
{
    short i,j;
    short tag;
    short *pQt;
    int buffer2[8][8];
    int *buffer1;
    short offset;

    switch(flag)
    {
        case 0:        //亮度
            pQt=YQtTable;
            offset=128;
            break;
        case 1:        //红
            pQt=UQtTable;
```

```
                offset=0;
                break;
        case 2:         //蓝
                pQt=VQtTable;
                offset=0;
                break;
    }

    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
        {
            tag=Zig_Zag[i][j];
            buffer2[i][j]=(int)s[tag]*(int)pQt[tag];
        }
    buffer1=(int *)buffer2;
    Fast_IDCT(buffer1);//反 DCT
    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
            d[i*8+j]=buffer2[i][j]+offset;
}
//快速反 DCT
void Fast_IDCT(int * block)
{
    short i;
    for (i=0; i<8; i++)idctrow(block+8*i);
    for (i=0; i<8; i++)idctcol(block+i);
}
//从源文件读取一个字节
u8 ReadByte(void)
{
    u8 i;
    i=*lp++;
    lp-=P_Cal(lp);//经过 P_Cal 的处理,把指针移动
    if(i==0xff)lp++;
    BitPos=8;
    CurByte=i;
    return i;
}
//初始化快速反 DCT
void Initialize_Fast_IDCT(void)
{
    short i;
```

```
    iclp = iclip+512;
    for (i= -512; i<512; i++)
        iclp[i] = (i<-256) ? -256 : ((i>255) ? 255 : i);
}
//////////////////////////////////////////////////////////////////////////
void idctrow(int * blk)
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    //intcut
    if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x3 = blk[2]) |
        (x4 = blk[1]) | (x5 = blk[7]) | (x6 = blk[5]) | (x7 = blk[3])))
    {
        blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=blk[6]=blk[7]=blk[0]<<3;
        return;
    }
    x0 = (blk[0]<<11) + 128; // for proper rounding in the fourth stage
    //first stage
    x8 = W7*(x4+x5);
    x4 = x8 + (W1-W7)*x4;
    x5 = x8 - (W1+W7)*x5;
    x8 = W3*(x6+x7);
    x6 = x8 - (W3-W5)*x6;
    x7 = x8 - (W3+W5)*x7;
    //second stage
    x8 = x0 + x1;
    x0 -= x1;
    x1 = W6*(x3+x2);
    x2 = x1 - (W2+W6)*x2;
    x3 = x1 + (W2-W6)*x3;
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;
    //third stage
    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181*(x4+x5)+128)>>8;
    x4 = (181*(x4-x5)+128)>>8;
    //fourth stage
    blk[0] = (x7+x1)>>8;
    blk[1] = (x3+x2)>>8;
```

```
        blk[2] = (x0+x4)>>8;
        blk[3] = (x8+x6)>>8;
        blk[4] = (x8-x6)>>8;
        blk[5] = (x0-x4)>>8;
        blk[6] = (x3-x2)>>8;
        blk[7] = (x7-x1)>>8;
}
//////////////////////////////////////////////////////////////////////////
void idctcol(int * blk)
{
        int x0, x1, x2, x3, x4, x5, x6, x7, x8;
        //intcut
        if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) | (x3 = blk[8*2]) |
                (x4 = blk[8*1]) | (x5 = blk[8*7]) | (x6 = blk[8*5]) | (x7 = blk[8*3])))
        {
                blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=blk[8*4]=blk[8*5]
                        =blk[8*6]=blk[8*7]=iclp[(blk[8*0]+32)>>6];
                return;
        }
        x0 = (blk[8*0]<<8) + 8192;
        //first stage
        x8 = W7*(x4+x5) + 4;
        x4 = (x8+(W1-W7)*x4)>>3;
        x5 = (x8-(W1+W7)*x5)>>3;
        x8 = W3*(x6+x7) + 4;
        x6 = (x8-(W3-W5)*x6)>>3;
        x7 = (x8-(W3+W5)*x7)>>3;
        //second stage
        x8 = x0 + x1;
        x0 -= x1;
        x1 = W6*(x3+x2) + 4;
        x2 = (x1-(W2+W6)*x2)>>3;
        x3 = (x1+(W2-W6)*x3)>>3;
        x1 = x4 + x6;
        x4 -= x6;
        x6 = x5 + x7;
        x5 -= x7;
        //third stage
        x7 = x8 + x3;
        x8 -= x3;
        x3 = x0 + x2;
        x0 -= x2;
        x2 = (181*(x4+x5)+128)>>8;
```

```
    x4 = (181*(x4-x5)+128)>>8;
    //fourth stage
    blk[8*0] = iclp[(x7+x1)>>14];
    blk[8*1] = iclp[(x3+x2)>>14];
    blk[8*2] = iclp[(x0+x4)>>14];
    blk[8*3] = iclp[(x8+x6)>>14];
    blk[8*4] = iclp[(x8-x6)>>14];
    blk[8*5] = iclp[(x0-x4)>>14];
    blk[8*6] = iclp[(x3-x2)>>14];
    blk[8*7] = iclp[(x7-x1)>>14];
}
```

此部分代码包含了 JPEG/JPG 以及 BMP 的解码代码，它们的解码时通过 AI_LoadPicFile 函数来实现的，在该函数里面，会先判断文件的类型，来调用不同的解码函数，解码 JPEG 由 Decode 实现，而解码 BMP 则由 Bmp Decode 函数实现。AI_LoadPicFile 函数会将图片以合适的大小显示在液晶上（总是不会超过你给定的区域），对比输入尺寸大的图片，会自动压缩。解码图片完成后返回解码是否成功的信息。

保存 jpegdecode.c，并在工程中新建一个 JPEG 的组，把 jpegdecode.c 加入该组下。然后打开 jpegbmp.h，输入如下代码：

```
#ifndef __JPEGBMP_H__
#define __JPEGBMP_H__
#include "sys.h"
#include "fat.h"
//BMP 信息头
typedef __packed struct
{
    DWORD biSize ;              //说明 BITMAPINFOHEADER 结构所需要的字数。
    LONG    biWidth ;           //说明图象的宽度，以象素为单位
    LONG    biHeight ;      //说明图象的高度，以象素为单位
    WORD    biPlanes ;      //为目标设备说明位面数，其值将总是被设为 1
    WORD    biBitCount ;        //说明比特数/象素，其值为 1、4、8、16、24、或 32
    DWORD biCompression ;   //说明图象数据压缩的类型。其值可以是下述值之一：
    //BI_RGB：没有压缩；
    //BI_RLE8：每个象素 8 比特的 RLE 压缩编码，压缩格式由 2 字节组成(重复象素计数和颜色索引)；
    //BI_RLE4：每个象素 4 比特的 RLE 压缩编码，压缩格式由 2 字节组成
    //BI_BITFIELDS：每个象素的比特由指定的掩码决定。
    DWORD biSizeImage ;//说明图象的大小，以字节为单位。当用 BI_RGB 格式时，可设置为 0
    LONG    biXPelsPerMeter ;//说明水平分辨率，用象素/米表示
    LONG    biYPelsPerMeter ;//说明垂直分辨率，用象素/米表示
    DWORD biClrUsed ;           //说明位图实际使用的彩色表中的颜色索引数
    DWORD biClrImportant ; //说明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。
```

```
}BITMAPINFOHEADER ;
//BMP 头文件
typedef __packed struct
{
    WORD    bfType ;        //文件标志.只对'BM',用来识别 BMP 位图类型
    DWORD bfSize ;        //文件大小,占四个字节
    WORD    bfReserved1 ;//保留
    WORD    bfReserved2 ;//保留
    DWORD bfOffBits ;   //从文件开始到位图数据(bitmap data)开始之间的的偏移量
}BITMAPFILEHEADER ;
//彩色表
typedef __packed struct
{
    BYTE rgbBlue ;      //指定蓝色强度
    BYTE rgbGreen ;     //指定绿色强度
    BYTE rgbRed ;       //指定红色强度
    BYTE rgbReserved ;//保留，设置为 0
}RGBQUAD ;
//位图信息头
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    //RGBQUAD bmiColors[256];
}BITMAPINFO;

typedef RGBQUAD * LPRGBQUAD;//彩色表
//图象数据压缩的类型
#define BI_RGB            0L
#define BI_RLE8           1L
#define BI_RLE4           2L
#define BI_BITFIELDS 3L
#define M_SOF0    0xc0
#define M_DHT     0xc4
#define M_EOI     0xd9
#define M_SOS     0xda
#define M_DQT     0xdb
#define M_DRI     0xdd
#define M_APP0    0xe0
#define W1 2841 /* 2048*sqrt(2)*cos(1*pi/16) */
#define W2 2676 /* 2048*sqrt(2)*cos(2*pi/16) */
#define W3 2408 /* 2048*sqrt(2)*cos(3*pi/16) */
#define W5 1609 /* 2048*sqrt(2)*cos(5*pi/16) */
```

```
#define W6 1108 /* 2048*sqrt(2)*cos(6*pi/16) */
#define W7 565   /* 2048*sqrt(2)*cos(7*pi/16) */


#define MAKEWORD(a, b)          ((WORD)(((BYTE)(a)) | ((WORD)((BYTE)(b))) << 8))
#define MAKELONG(a, b)          ((LONG)(((WORD)(a)) | ((DWORD)((WORD)(b))) << 16))
#define LOWORD(l)               ((WORD)(l))
#define HIWORD(l)               ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))
#define LOBYTE(w)               ((BYTE)(w))
#define HIBYTE(w)               ((BYTE)(((WORD)(w) >> 8) & 0xFF))
//宏定义
#define WIDTHBYTES(i)       ((i+31)/32*4)//??????????
#define PI 3.1415926535
//函数返回值定义
#define FUNC_OK 0
#define FUNC_MEMORY_ERROR 1
#define FUNC_FILE_ERROR 2
#define FUNC_FORMAT_ERROR 3
////////////////////////////////////////////////
//BMP 解码函数
BOOL BmpDecode(FileInfoStruct *BmpFileName);
////////////////////////////////////////////////
//JPEG  解码函数
int   InitTag(void);
void InitTable(void);                                   //初始化数据表
int   Decode(void);                                     //解码
int   DecodeMCUBlock(void);
int   HufBlock(BYTE dchufindex,BYTE achufindex);//哈夫曼解码
int   DecodeElement(void);                              //解码一个像素
void IQtIZzMCUComponent(short flag);                    //反量化
void IQtIZzBlock(short   *s ,short * d,short flag);
void GetYUV(short flag);            //色彩转换的实现,得到色彩空间数据
void StoreBuffer(void);
BYTE ReadByte(void);                    //从文件里面读取一个字节出来
void Initialize_Fast_IDCT(void); //初始化反离散傅立叶变换
void Fast_IDCT(int * block);  //快速反离散傅立叶变换
void idctrow(int * blk);
void idctcol(int * blk);
////对缓冲区数据进行移动处理,使操作 SD 卡就像操作 sram 一样
unsigned int P_Cal(unsigned char*pc);
BOOL AI_LoadPicFile(FileInfoStruct *FileName,u16 sx,u16 sy,u16 ex,u16 ey);//智能显示图片
#endif
```

保存此部分代码，然后打开 sysfile.c，输入如下代码：

```
#include "sysfile.h"
#include "fat.h"
//Mini STM32 开发板
//系统文件查找代码
//正点原子@ALIENTEK
u32 PICCLUSTER=0;//图片文件夹地址
u32 sys_ico[9];   //系统图标缓存区!不能篡改!
u32 file_ico[4]; //文件图标缓存区  folder;mus;pic;book;
//系统文件夹
const unsigned char *folder[]=
{
"SYSTEM",
"FONT",
"SYSICO",
"PICTURE",
"GAME",
"LEVEL1",
"LEVEL2",
"LEVEL3",
};
//系统文件名定义
const unsigned char *sysfile[]=
{
//系统字体图标  0 开始
"GBK16.FON",
"GBK12.FON",
"UNI2GBK.SYS",
//系统文件图标     3 开始
"FOLDER.BMP",
"MUS.BMP",
"PIC.BMP",
"BOOK.BMP",
//系统主界面图标  7 开始
"MUSIC.BMP",
"PICTURE.BMP",
"GAME.BMP",
"ALARM.BMP",
"TIME.BMP",
"SETTING.BMP",
"TXT.BMP",
"RADIO.BMP",
"LIGHT.BMP",
};
```

```
//获取系统文件的存储地址
//次步出错,则无法启动!!!
//返回 0，成功。返回其他，错误代码
//sel:0  系统文件
//sel:1  图片文件夹
u8 SysInfoGet(u8 sel)
{
    u32 cluster=0;
    u32 syscluster=0;
    u8 t=0;
    FileInfoStruct t_file;
    //得到根目录的簇号
    if(FAT32_Enable)cluster=FirstDirClust;
    else cluster=0;

    if(sel==1)//查找图片文件夹
    {
        t_file=F_Search(cluster,(unsigned char *)folder[3],T_FILE);//查找 PICTURE 文件夹
        if(t_file.F_StartCluster==0)return 1;//图片文件夹丢失
        PICCLUSTER=t_file.F_StartCluster;//图片文件夹所在簇号
    }else//查找系统文件
    {
        t_file=F_Search(cluster,(unsigned char *)folder[0],T_FILE);//查找 system 文件夹
        if(t_file.F_StartCluster==0)return 2;//系统文件夹丢失
        syscluster=t_file.F_StartCluster;//保存系统文件夹所在簇号
        t_file=F_Search(syscluster,(unsigned  char  *)folder[2],T_FILE);//在 system 文件夹下查找 SYSICO 文件夹
        if(t_file.F_StartCluster==0)return 3;
        cluster=t_file.F_StartCluster;//保存 SYSICO 文件夹簇号
        for(t=0;t<9;t++)//查找系统图标,九个
        {
            t_file=F_Search(cluster,(unsigned char *)sysfile[t+7],T_BMP);//在 SYSICO 文件夹下查找系统图标
            sys_ico[t]=t_file.F_StartCluster;
            if(t_file.F_StartCluster==0)return 4;//失败
        }
        for(t=3;t<7;t++)//查找文件图标,4 个
        {
            t_file=F_Search(cluster,(unsigned char *)sysfile[t],T_BMP);//在 SYSICO 文件夹下查找文件图标
            file_ico[t-3]=t_file.F_StartCluster;
            if(file_ico[t-3]==0)return 5;//失败
```

```
            }
        }
        return 0;//成功
    }
```

此部分由一个函数 SysInfoGet 构成，该函数用于查找各种系统文件/文件夹以及自定义的文件/文件夹等，具体实现请参考代码。在工程里面新建 SYSFILE 的组，然后把 sysfile.c 文件加入改组下，保存。打开 sysfile.h，输入如下代码：

```
#ifndef _SYSFILE_H_
#define _SYSFILE_H_
#include "sys.h"
//Mini STM32 开发板
//系统文件查找代码
//正点原子@ALIENTEK
extern u32 PICCLUSTER;//图片文件夹首地址
u8 SysInfoGet(u8 sel);//获取系统文件信息
#endif
```

保存此部分代码。最后我们在 test.c 文件里面修改 main 函数如下：

```
int main(void)
{
    u8 i;
    u8 key;
    FileInfoStruct *FileInfo;
    u16 pic_cnt=0;//当前目录下图片文件的个数
    u16 index=0;   //当前选择的文件编号
    u16 time=0;
    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LCD_Init();                 //初始化液晶
    KEY_Init();                 //按键初始化
    LED_Init();               //LED 初始化
    SPI_Flash_Init();     //SPI FLASH 使能
    if(Font_Init())//字库不存在,则更新字库
    {
        POINT_COLOR=RED;
        LCD_ShowString(60,50,"Mini STM32");
        LCD_ShowString(60,70,"Font ERROR");
        while(1);
    }
    POINT_COLOR=RED;
    Show_Str(60,50,"Mini STM32 开发板",16,0);
    Show_Str(60,70,"图片显示  程序",16,0);
    Show_Str(60,90,"正点原子@ALIENTEK",16,0);
```

```
        Show_Str(60,110,"2011 年 1 月 2 日",16,0);
        SD_Init();
        while(FAT_Init())//FAT 错误
        {
            Show_Str(60,130,"文件系统错误!",16,0);
            i= SD_Init();
            if(i)Show_Str(60,150,"SD 卡错误!",16,0);//SD 卡初始化失败
            delay_ms(500);
            LCD_Fill(60,130,240,170,WHITE);//清除显示
            delay_ms(500);
            LED0=!LED0;
        }
        while(SysInfoGet(1))//得到图片文件夹
        {
            Show_Str(60,130,"图片文件夹未找到!",16,0);
            delay_ms(500);
            FAT_Init();
            SD_Init();
            LED0=!LED0;
            LCD_Fill(60,130,240,170,WHITE);//清除显示
            delay_ms(500);
        }
        Show_Str(60,130,"开始显示...",16,0);
        delay_ms(1000);
        Cur_Dir_Cluster=PICCLUSTER;
        while(1)
        {
            pic_cnt=0;
            Get_File_Info(Cur_Dir_Cluster,FileInfo,T_JPEG|T_JPG|T_BMP,&pic_cnt);// 获 取 当
前文件夹下面的目标文件个数
            if(pic_cnt==0)//没有图片文件
            {
                LCD_Clear(WHITE);//清屏
                while(1)
                {
                    if(time%2==0)Show_Str(32,150,"没有图片,请先 COPY 图片到 SD 卡的
PICTURE 文件夹,然后后重启!",16,0);
                    else LCD_Clear(WHITE);
                    time++;
                    delay_ms(300);
                }
            }
            FileInfo=&F_Info[0];//开辟暂存空间.
```

```
            index=1;
            while(1)
            {
                Get_File_Info(Cur_Dir_Cluster,FileInfo,T_JPEG|T_JPG|T_BMP,&index);// 得 到
这张图片的信息
                LCD_Clear(WHITE);//清屏,加载下一幅图片的时候,一定清屏
                AI_LoadPicFile(FileInfo,0,0,240,320);//显示图片
                POINT_COLOR=RED;
                Show_Str(0,0,FileInfo->F_Name,16,1);//显示图片名字
                while(1)//延时 3s
                {
                    key=KEY_Scan();
                    if(key==1)break;//下一张
                    else if(key==2)//上一张
                    {
                        if(index>1)index-=2;
                        else index=pic_cnt-1;
                        break;
                    }
                    delay_ms(1);
                    time++;
                    if(time%100==0)LED0=!LED0;
                    if(time>3000)
                    {
                        time=0;
                        break;
                    }
                }
                index++;
                if(index>pic_cnt)index=1;//显示第一副,循环
            }
        }
    }
```

至此，整个图片显示实验的软件设计部分就结束了。该程序将实现浏览 PICTURE 文件夹下的所有图片及其名字，每隔 3s 左右切换一幅图片。

3.25.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了），如下图所示：

图 3.25.4.1 图片显示实验显示效果

按 KEY0 和 KEY1 可以快速切换到下一张或上一张。

3.26 触控USB鼠标实验

STM32F103 系列芯片都自带了 USB，不过 STM32F103 的 USB 都只能用来做设备，而不能用作主机。既便如此，对于一般应用来说已经足够了。本节，我们将向大家介绍如何在 ALIENTEK MiniSTM32 开发板上虚拟一个 USB 鼠标。本节分为如下几个部分：

3.26.1 USB 简介

3.26.2 硬件设计

3.26.3 软件设计

3.26.4 下载与测试

3.26.1 USB 简介

USB ,是英文 Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通串线，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。USB 接口支持设备的即插即用和热插拔功能。USB 是在 1994 年底由英特尔、康柏、IBM、Microsoft 等多家公司联合提出的。

USB 发展到现在已经有 USB1.0/1.1/2.0/3.0 等多个版本。目前用的最多的就是 USB1.1 和 USB2.0，USB3.0 已经开发出来了相信不久就可以在我们的电脑上见到。STM32F103 自带的 USB 符合 USB2.0 规范。

标准 USB 共四根线组成,除 VCC/GND 外,另外为 D+,D-; 这两根数据线采用的是差分电压的方式进行数据传输的。在 USB 主机上，D-和 D+都是接了 15K 的电阻到低的，所以在没有设备接入的时候，D+、D-均是低电平。而在 USB 设备中，如果是高速设备，则会在 D+上接一个 1.5K 的电阻到 VCC，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 VCC。这样当设备接入主机的时候，主机就可以判断是否有设备接入，并能判断设备是高速设备还是低速设备。接下来，我们简单介绍一下 STM32 的 USB 控制器。

STM32F103 的 MCU 自带 USB 从控制器，符合 USB 规范的通信连接；PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区（专用的 512 字节，和 CAN 共用），最多可用于 16 个单向或 8 个双向端点。USB 模块同 PC 主机通信，根据 USB 规范实现令牌分组的检测，数据发送/接收的处理，和握手分组的处理。整个传输的格式由硬件完成，其中包括 CRC 的生成和校验。

每个端点都有一个缓冲区描述块，描述该端点使用的缓冲区地址、大小和需要传输的字节数。当 USB 模块识别出一个有效的功能/端点的令牌分组时，(如果需要传输数据并且端点已配置)随之发生相关的数据传输。USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后，如果需要，则根据传输的方向，发送或接收适当的握手分组。在数据传输结束时，USB 模块将触发与端点相关的中断，通过读状态寄存器和/或者利用不同的中断来处理。

USB 的中断映射单元：将可能产生中断的 USB 事件映射到三个不同的 NVIC 请求线上：

1、USB 低优先级中断(通道 20)：可由所有 USB 事件触发(正确传输，USB 复位等)。固件在处理中断前应当首先确定中断源。

2、USB 高优先级中断(通道 19)：仅能由同步和双缓冲批量传输的正确传输事件触发，目的是保证最大的传输速率。

3、USB 唤醒中断(通道 42)：由 USB 挂起模式的唤醒事件触发。

USB 设备框图如图 32.1 所示：

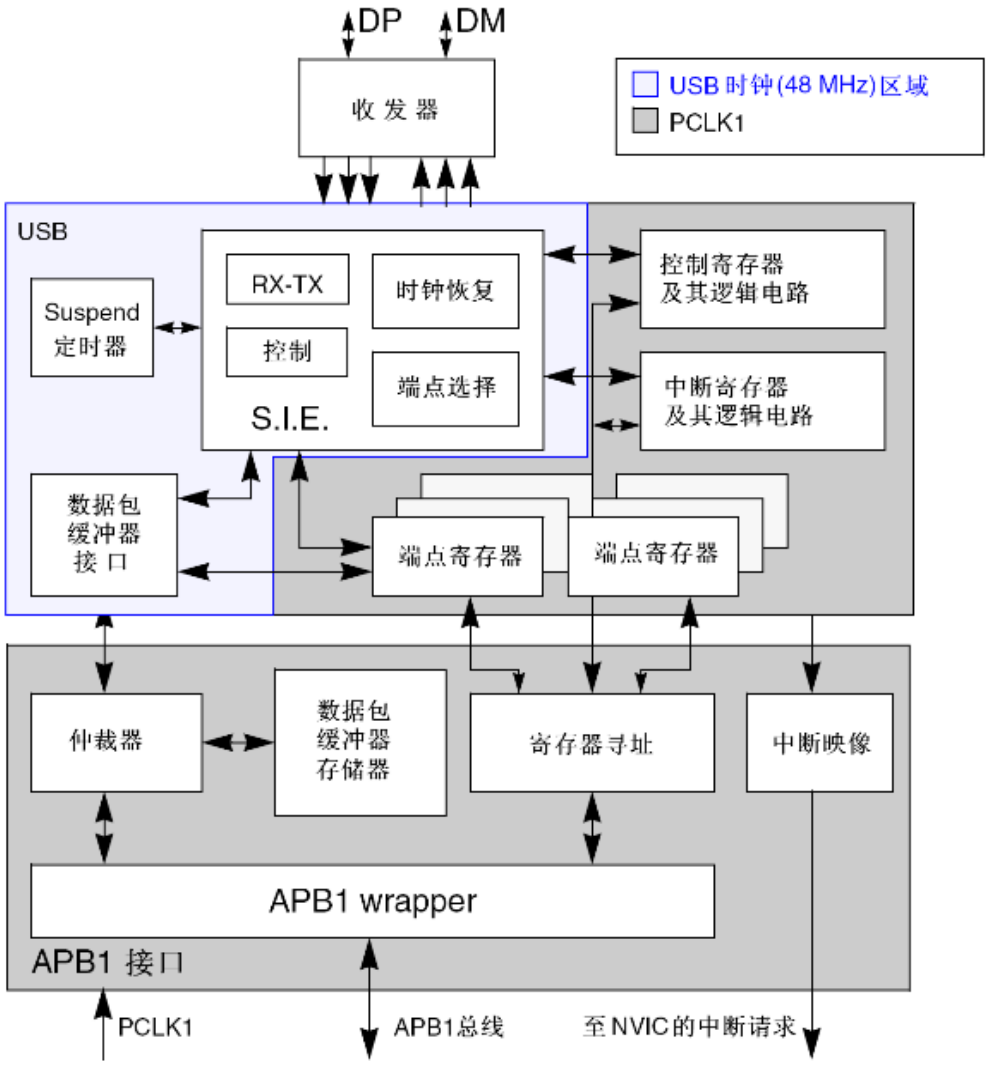


图 3.26.1.1 USB 设备框图

整个 USB 通信的详细过程是很发杂的，这里我们就不再详细介绍其各个环节，感兴趣的读者，可以去看看电脑圈圈的《圈圈教你玩 USB》这本书，该书对 USB 的讲解是很详细的。USB 部分，ST 提供了几个例程，这些例程对于我们了解 STM32F103 的 USB 会有不少帮助，尤其在不懂的时候，看看 ST 的例程，会有意想不到的收获。本实验的 USB 部分就是移植 ST 的 JoyStickMouse 例程相关部分而来，再加上我们的触摸屏，做成一个触控鼠标。ST 提供的 USB 例程在 X:\Keil3.80\ARM\Examples\ST\STM32F10xUSBLib\Demos 文件夹下（X 是您的安装盘）。

3.26.2 硬件设计

本节实验功能简介：开机的时候先检测触摸屏是否校准过，如果没有，则校准。如果校准过了，则开始触摸屏画图，然后将我们的坐标数据上传到电脑（假定 USB 已经配置成功了，DS1 亮），这样就可以用触摸屏来控制电脑的鼠标了。在控制鼠标的同时，如果按键 0 被按下，则强制进入校准程序。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0、DS1（外部 LED0/LED1）。

3）TFTLCD 液晶模块。

4）KEY0。

5）USB 接口。

前面 4 部分，在之前的实例中都介绍过了，我们在此就不介绍了。接下来看看我们电脑 USB 与 STM32 的 USB 连接口。ALIENTEK MiniSTM32 采用的是 5PIN 的 miniUSB 接头，用来和 STM32 的 USB 相连接，连接电路如下图所示：

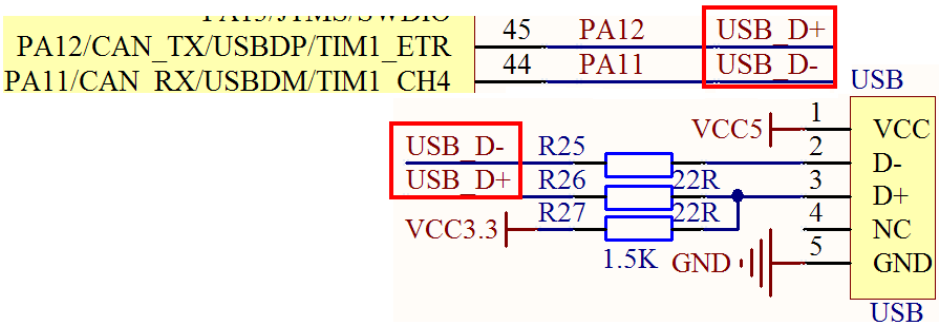


图 3.26.2.1 MiniUSB 与 STM32 的连接电路图

3.26.3 软件设计

这一节，我们在实验 18 的基础上修改，所以先打开实验 18 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，然后在 USB 文件夹下面新建 LIB 和 CONFIG 文件夹，分别用来存放与 USB 核相关的代码以及配置部分代码。这两部分代码我们就不细说了，给大家看看在这两个文件夹内的代码都有哪些，如下图所示：

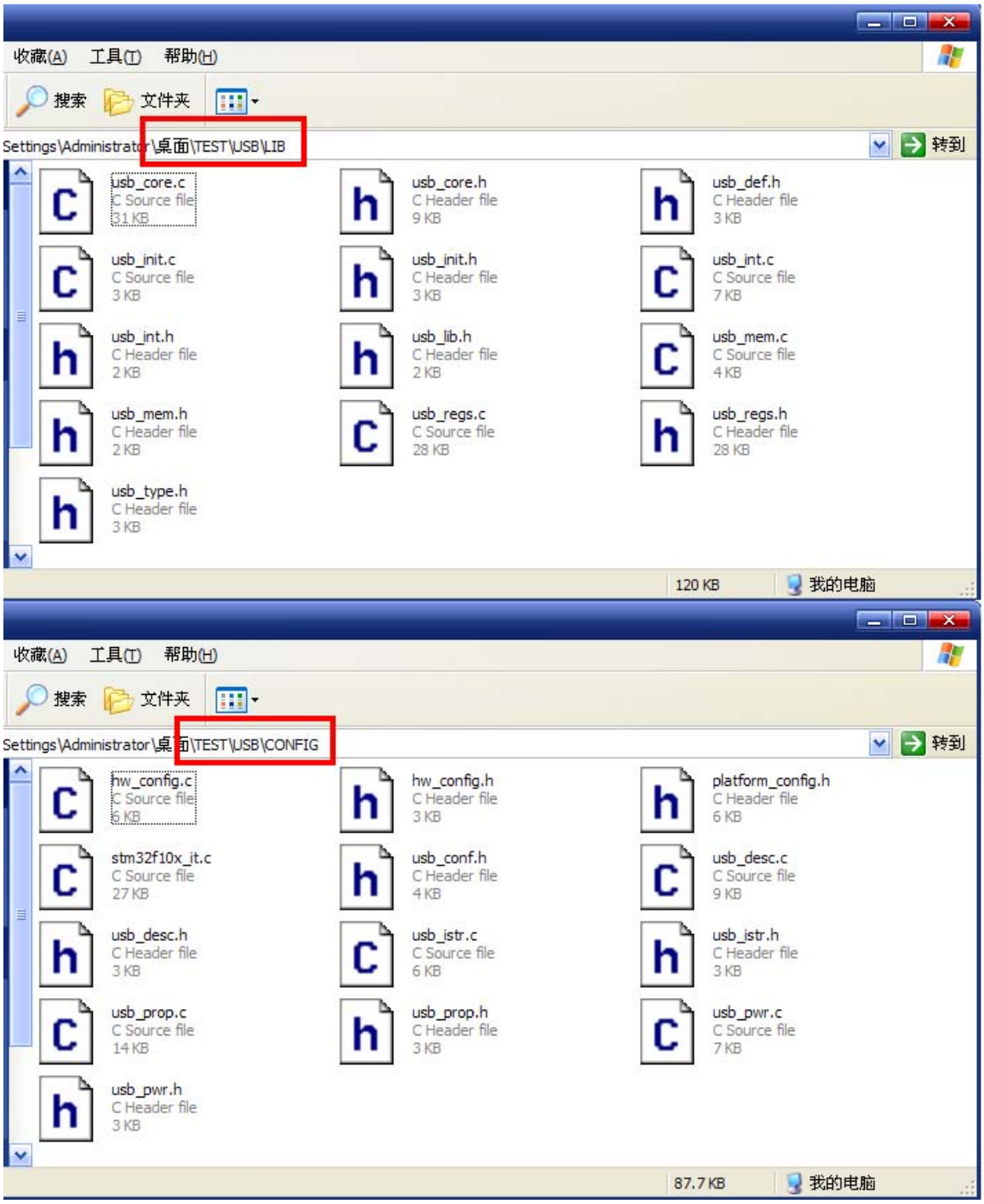


图 3.26.3.1 USB 相关部分代码

以上代码，都是从 ST 提供的例程里面移植过来的，我们只需要针对 ST 提供的例程里面的很少一部分代码进行修改，就可以构建你自己的 USB 通信了。我们在工程文件里面新建 USB 和 USBCFG 组，分别加入 USB\LIB 下面的代码和 USB\CONFIG 下面的代码。然后把 LIB 和 CONFIG 文件夹加入头文件包含路径。

在 test.c 里面，我们修改 main 函数如下：

```
int main(void)
{
    u8 key;
```

```
u8 i=0;
u8 tpx=0;
s8 x0;              //发送到电脑端的坐标值
s8 y0;
short xlast; //最后一次按下的坐标值
short ylast;
Stm32_Clock_Init(9);//系统时钟设置
delay_init(72);       //延时初始化
uart_init(72,9600); //串口 1 初始化
LCD_Init();                 //初始化液晶
KEY_Init();                 //按键初始化
LED_Init();               //LED 初始化
//USB 配置
USB_Interrupts_Config();
Set_USBClock();
USB_Init();
POINT_COLOR=RED;//设置字体为蓝色
LCD_ShowString(60,50,"Mini STM32");
LCD_ShowString(60,70,"USB TEST");
LCD_ShowString(60,90,"ATOM@ALIENTEK");
LCD_ShowString(60,110,"2011/1/2");
LCD_ShowString(60,130,"Press KEY0 to Adjust");
Touch_Init();
delay_ms(1500);
Load_Drow_Dialog();
while(1)
{
    key=KEY_Scan();
    tpx=AI_Read_TP();//得到触点的状态
    if(tpx)//触摸屏被按下
    {
        xlast=Pen_Point.X0;
        ylast=Pen_Point.Y0;
        while(1)
        {
            tpx=AI_Read_TP();
            if(tpx==0)break;//触点松开了
            if(Pen_Point.X0>216&&Pen_Point.Y0<16)Load_Drow_Dialog();//清除
            else Draw_Big_Point(Pen_Point.X0,Pen_Point.Y0);//画图

if((Pen_Point.X0!=xlast||Pen_Point.Y0!=ylast)&&(bDeviceState==CONFIGURED)&&tpx==1)
            {
                x0=(xlast-Pen_Point.X0)*3;//上次坐标值与得到的坐标值之差,扩大 3
```

```
倍
                        y0=(ylast-Pen_Point.Y0)*3;
                        xlast=Pen_Point.X0;              //记录本次的坐标值
                        ylast=Pen_Point.Y0;
                        Joystick_Send(0,-x0,-y0,0); //发送数据到电脑
                        delay_ms(10);
                    }else if((bDeviceState==CONFIGURED)&&tpx==2)//单击
                    {
                        tpx=0X01;
                        Joystick_Send(tpx,0,0,0);//模拟左键按下
                    }
                    delay_us(50);
                }
                Joystick_Send(0,0,0,0);//发送左键松开
            }else delay_ms(1);
            if(bDeviceState==CONFIGURED)LED1=0;//当 USB 配置成功了，LED1 亮，否则，
灭
            else LED1=1;
            if(key==1)//KEY0 按下,则执行校准程序
            {
                LCD_Clear(WHITE);//清屏
                Touch_Adjust();  //屏幕校准
                Save_Adjdata();
                Load_Drow_Dialog();
            }
            i++;
            if(i==200)
            {
                i=0;
                LED0=!LED0;
            }
        };
    }
```

此部分代码用于实现我们在硬件设计部分提到的功能，USB 的配置通过三个函数完成：USB_Interrupts_Config()、Set_USBClock()和 USB_Init()，第一个函数用于开启 USB 唤醒中断和 USB 低优先级数据处理中断，Set_USBClock 函数用于 配置 USB 时钟，也就是从 72M 的主频得到 48M 的 USB 时钟（1.5 分频）。最后 USB_Init()函数用于初始化 USB，最主要的就是调用了 Joystick_init 函数，开启了 USB 部分的电源等。这里需要特别说明的是，USB 配置并没有对 PA11 和 PA12 这两个 IO 口进行设置，是因为，一旦开启了 USB 电源（USB_CNTR 的 PDWN 位清零）PA11 和 PA12 将不再作为其他功能使用，仅供 USB 使用，所以在开启了 USB 电源之后不论你怎么配置这两个 IO 口，都是无效的。要在此获取这两个 IO 口的配置权，则需要关闭 USB 电源，也就是置位 USB_CNTR 的 PDWN 位。

此部分代码对于接收触摸屏数据我们也使用了新的函数 AI_Read_TP()，不同于之前的靠判

断 PEN 引脚信号来决定是否有数据输入。我们通过该函数可以方便的实现滑动检测及定点检测等。具体怎么实现的请参考源码。

USB 数据发送，我们采用 Joystick_Send 来实现，我们将得到的鼠标数据，在 Joystick_Send 函数里面打包，并通过 USB 端点 1 发送到电脑。

软件设计部分，就给大家介绍到这里。

3.26.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，在 USB 没有配置成功的时候，其界面同实验 18 是一模一样的，如下图所示：

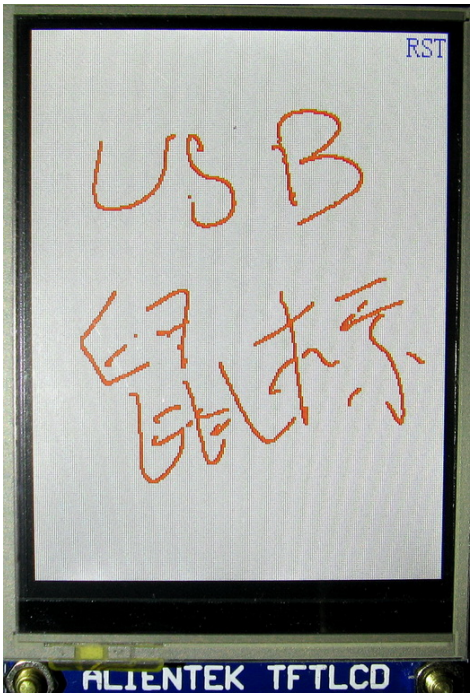


图 3.26.4.1 无 USB 连接时的界面

此时 DS1 不亮，DS0 闪烁，其实就是一个触摸屏画图的功能，而一旦我们将 USB 连接上（将 USB 线接到 USB 接头上，而不是 USB_232 接头上，如果你有两根 USB 线，则可以两个同时都接上，他们不会相互影响），则可以看到 DS1 亮了，而且在电脑上会提示找到新硬件如下图所示：

图 3.26.4.2 电脑提示找到新硬件

在硬件安装完成之后，我们在设备管理器里面可以发现多出了一个人体学输入设备，如下图所示：

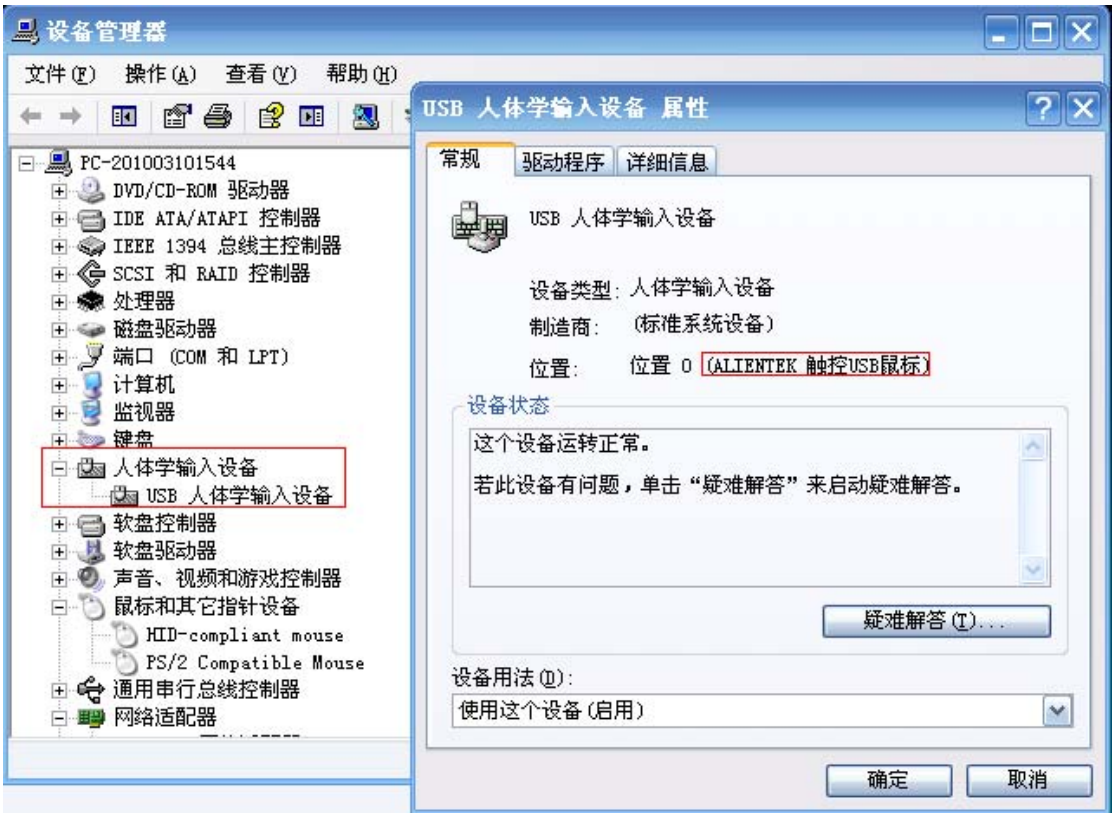


图 3.26.4.3 USB 人体学输入设备

此时我们按动触摸屏，就可以发现电脑屏幕上的光标随着你在触摸屏上的移动而移动了，如果想要打开某个文件，则在触摸屏上双击，要选中则单击。

3.27 USB读卡器实验

上节我们向大家介绍了如何利用 STM32 的 USB 来做一个触控 USB 鼠标，本节我们将利用 STM32 的 USB 来做一个 USB 读卡器。本节分为如下几个部分：

3.27.1 USB 读卡器简介

3.27.2 硬件设计

3.27.3 软件设计

3.27.4 下载与测试

3.27.1 USB 读卡器简介

ALIENTEK MiniSTM32 开发板板载了 SD 卡读卡器，而 STM32F103 又有 USB，且在板上带有 USB 连接头，这样我们便可以通过 STM32 的 USB 来读写 SD 卡，从而实现一个 USB 读卡器。

USB 读卡器的实现最重要的有两个部分：USB 部分和 SD 卡部分。USB 部分同上一节的差不多，只是这一节我们的 STM32F103 被识别成一个大容量存储设备，而不是人体学输入设备。SD 卡部分，最重要的就是 2 个函数，一个 MSD_WriteBuffer 函数，用于向 SD 卡写入数据，当你要 COPY 文件到 SD 卡的时候，就是由这个函数完成的。另外一个是 MSD_ReadBuffer 函数，该函数用于读取 SD 卡上面的数据。

这里的数据并不需要经过文件系统处理，而是完全电脑控制，我们要做的就是读写 SD 卡就够了。本实验我们也是参考 Mass_Storage 例程而来的，不过在 ST 提供的例程 Mass_Storage 里面，使用的是 SDIO 方式来读写 SD 卡的，而我们这里采用的是 SPI 方式读写，所以速度会比较慢一点。

3.27.2 硬件设计

本节实验功能简介：开机的时候先检测 SD 卡是否存在，如果不存在则等待 SD 卡插入，同时在 LCD 上显示提示信息。在 SD 卡插入之后，就开始 USB 的配置，在配置成功之后既可以在电脑上发现可移动磁盘了。我们用 DS1 来指示 USB 正在读写 SD 卡，并在液晶上显示出来，同样我们还是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1）STM32F103RBT6。

2）DS0、DS1（外部 LED0/LED1）。

3）TFTLCD 液晶模块。

4）SD 卡。

5）USB 接口。

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。

3.27.3 软件设计

这一节，我们在实验 20 的基础上修改，先打开实验 18 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，然后在 USB 文件夹下面新建 LIB 和 CONFIG 文件夹，分别用来存放与 USB 核相关的代码以及配置部分代码。这两部分代码我们也不细说（因为 USB 部分我也没告通，仅是移植能用而已），这些代码都是从 ST 提供的例程 Mass_Storage 里面移植过来的。

然后，我们在工程文件里面新建 USB 和 USBCFG 组，分别加入 USB\LIB 下面的代码和 USB\CONFIG 下面的代码。然后把 LIB 和 CONFIG 文件夹加入头文件包含路径。

在 test.c 里面，我们修改 main 函数如下：

```
//bit0:表示电脑正在向 SD 卡写入数据
//bit1:表示电脑正从 SD 卡读出数据
//bit2:SD 卡写数据错误标志位
//bit3:SD 卡读数据错误标志位
```

```
//bit4:1,表示电脑有轮询操作(表明连接还保持着)
u8 Usb_Status_Reg=0;
int main(void)
{
    u8 offline_cnt=0;
    u8 tct=0;
    u8 USB_STA;
    u8 Divece_STA;

    Stm32_Clock_Init(9);//系统时钟设置
    delay_init(72);       //延时初始化
    uart_init(72,9600); //串口 1 初始化
    LCD_Init();                 //初始化液晶
    //KEY_Init();               //按键初始化
    LED_Init();               //LED 初始化
    POINT_COLOR=RED;//设置字体为蓝色
    LCD_ShowString(60,50,"Mini STM32");
    LCD_ShowString(60,70,"USB TEST");
    LCD_ShowString(60,90,"ATOM@ALIENTEK");
    LCD_ShowString(60,110,"2010/6/19");

    while(SD_Init())
    {
        LCD_ShowString(60,130,"SD Init ERR!");
        delay_ms(500);
        LCD_ShowString(60,130,"Please Check");
        delay_ms(500);
    }
    LCD_ShowString(60,130,"SD Card Ready");//提示 SD 卡已经准备了
    Mass_Memory_Size[0]=SD_GetCapacity();//得到 SD 卡容量
    Mass_Block_Size[0] =512;//因为我们在 Init 里面设置了 SD 卡的操作字节为 512 个,所以这里一定是 512 个字节.
    Mass_Block_Count[0]=Mass_Memory_Size[0]/Mass_Block_Size[0];
    LCD_ShowString(60,150,"USB Connecting...");//提示 SD 卡已经准备了
    //USB 配置
    USB_Interrupts_Config();
    Set_USBClock();
    USB_Init();
    while(1)
    {
        delay_ms(1);
        if(USB_STA!=Usb_Status_Reg)//状态改变了
        {
```

```
                LCD_ShowString(60,170,"                    ");//清除
                if(Usb_Status_Reg&0x01)//正在写
                {
                    LCD_ShowString(60,170,"USB Writing...");//提示 USB 正在写入数据
                }
                if(Usb_Status_Reg&0x02)//正在读
                {
                    LCD_ShowString(60,170,"USB Reading...");//提示 USB 正在读出数据

                }
                if(Usb_Status_Reg&0x04)LCD_ShowString(60,190,"USB  Write  Err  ");//提示写
入错误
                else LCD_ShowString(60,190,"                    ");//清除错误
                if(Usb_Status_Reg&0x08)LCD_ShowString(60,210,"USB Read    Err ");//提示读
出错误
                else LCD_ShowString(60,210,"                    ");//清除错误
                USB_STA=Usb_Status_Reg;//记录最后的状态
            }
            if(Divece_STA!=bDeviceState)
            {
                if(bDeviceState==CONFIGURED)LCD_ShowString(60,150,"USB      Connected
");//提示 USB 连接已经建立
                else LCD_ShowString(60,150,"USB DisConnected ");//提示 USB 被拔出了
                Divece_STA=bDeviceState;
            }
            tct++;
            if(tct==200)
            {
                tct=0;
                LED0=!LED0;//提示系统在运行
                if(Usb_Status_Reg&0x10)
                {
                    offline_cnt=0;//USB 连接了,则清除 offline 计数器
                    bDeviceState=CONFIGURED;
                }else//没有得到轮询
                {
                    offline_cnt++;
                    if(offline_cnt>10)bDeviceState=UNCONNECTED;//2s 内没收到在线标记,
代表 USB 被拔出了
                }
                Usb_Status_Reg=0;
            }
        };
```

}

此部分代码就实现了我们之前在硬件设计部分描述的功能，这里我们用到了一个全局变量 Usb_Status_Reg，用来标记 USB 的相关状态，这样我们就可以在液晶上显示当前 USB 的状态了。

软件设计部分就为大家介绍到这里。

3.27.4 下载与测试

在代码编译成功之后，我们通过下载代码到 ALIENTEK MiniSTM32 开发板上，在 USB 配置成功并有 SD 卡接入的时候，LCD 显示效果如下图所示：

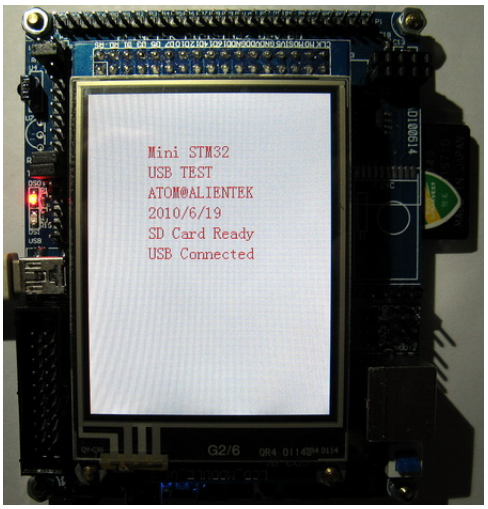


图 3.27.4.1 USB 连接成功

此时 DS1 不亮，DS0 闪烁，并且在电脑上可以看到我们的磁盘，如下图所示：

图 3.27.4.2 电脑找到的 USB 读卡器盘符

我们打开设备管理器，在通用串行总线控制器里面可以发现多出了一个 USB Mass Storage Device，如下图所示：

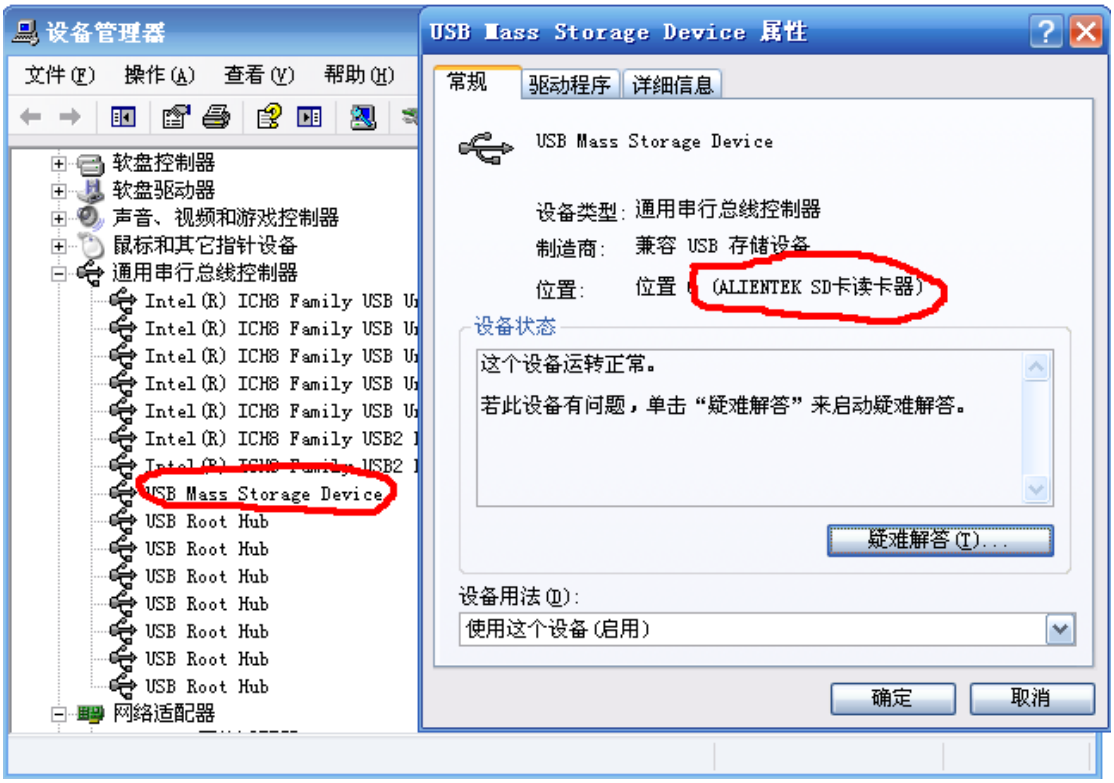


图 3.27.4.3 电脑发现了大容量存储设备

我们试着在移动磁盘里面拷贝数据出来或者写入数据进去，就可以看到 DS1 亮，并且会在液晶上显示当前的读写状态。

3.28 综合测试实验

前面已经给大家讲了 27 个实例了，本节将设计一个综合实例，用于测试 ALIENTEK MiniSTM32 开发板上的所有硬件资源，本实例也是本手册的最后一个实例。本节将不会对代码进行贴出，而只讲述功能。本节将分为如下几个部分：

3.28.1 系统启动
3.28.2 电子图书
3.28.3 数码相框
3.28.4 拼图游戏
3.28.5 触控画板
3.28.6 系统时间
3.28.7 鼠标画板
3.28.8 USB 连接
3.28.9 红外遥控
3.28.10 无线传输

本实验综合了很多前面讲到的实验，把他们集成在一个程序里面，以实现各种功能。该综合实验总共包含了 9 大功能，我们下面将一一向大家介绍。

3.28.1 系统启动

首先，在程序下载完之后或复位之后，系统就开始启动了，在启动的时候显示启动信息如下图所示：

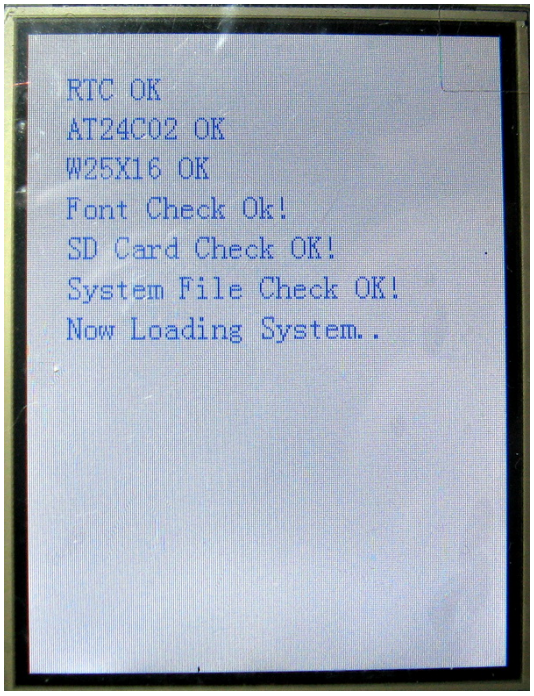


图 3.28.1.1 启动界面

上图显示了启动过程的一些信息，用于我们识别是否有硬件存在问题，相当于一个硬件自检过程。上图我们看到第一个显示的是 RTC OK，其实在这之前，我们还对系统的时钟、串口等进行了初始化，还有对触摸屏校准的检测以及按键检测等，这里我们并没有显示出来。

如果在启动的时候（按下复位键的同时，按键检测发生在 RTC 检测之前！），按下了 KEY0，则系统会强制设置当前时间为编译时间，如果按下了 KEY1，则会强制进行屏幕校准，如果按下 WK_UP 则会强制进行字库更新。

详细的启动过程为：首先配置时钟、串口、延时等，在这些配置好之后，就初始化 SPI FLASH，接着初始化触摸屏，在初始化触摸屏的时候，会检测是否被校准过，如果没有校准，则校准，并把校准数据放入 24C02。如果已经校准了，则直接读取 24C02 里面的校准数据，完成初始化。

在完成触摸屏初始化之后，开始检测按键，如果有按键按下，则执行对应的操作（上面红色字体部分有详细介绍）。

之后，我们开始 RTC 初始化，也就是屏幕上看到的第一项。接着检测 24C02，检测完 24C02 之后才开始检查 W25X16 中的字库是否存在，如果不存在则执行字库更新，在更新完后执行下一步。如果存在，则直接开始下一步。

检测完字库之后执行的操作就是检测 SD 卡，并初始化文件系统，在这两步完成之后，就会进入到下一步，开始查找系统文件，这里需要大家拷贝：光盘->SD 卡根目录文件->SYSTEM，把 SYSTEM 文件夹拷贝到 SD 卡的根目录。在找到系统文件之后，则开始启动主界面了，启动完成后，如下图所示：

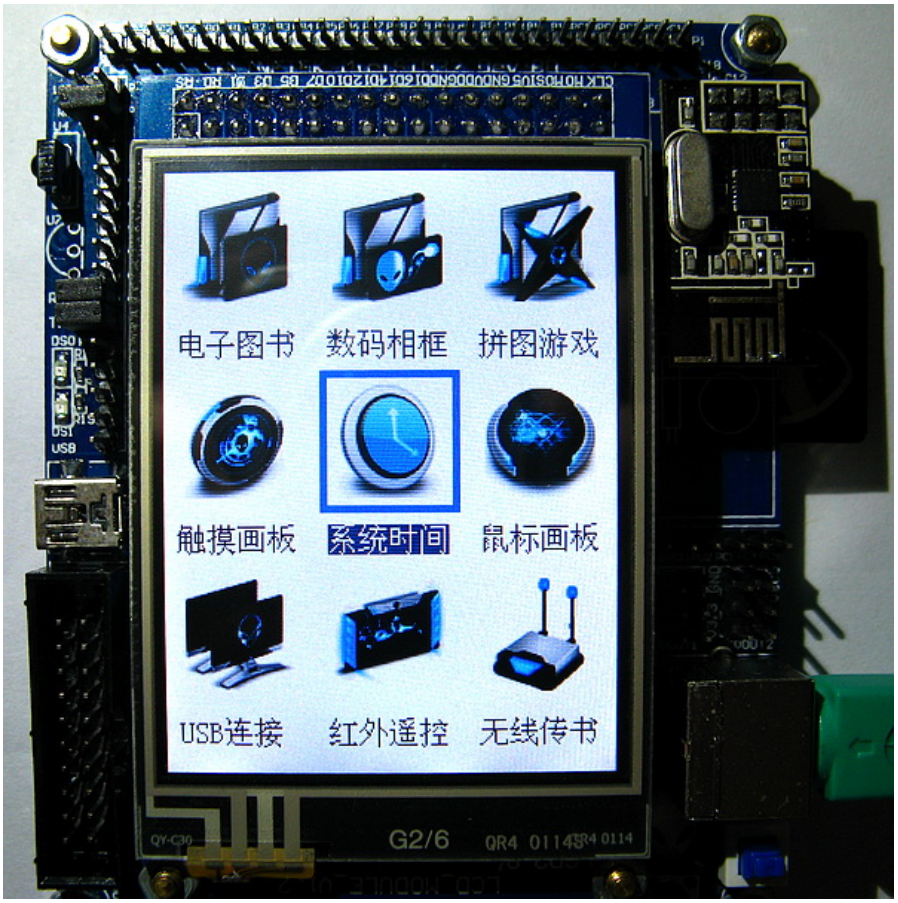


图 3.28.1.2 主界面

此时，系统就完成了启动，DS0 会每隔 2s 左右闪一次，提示程序正在运行（在其他各个子功能里面，都会有 DS0 闪烁，以提示程序在运行中），接下来我们就可以通过触摸屏点击屏幕上的各个图标，进入相应的功能了，下面我们一一介绍这些功能。

3.28.2 电子图书

在介绍完启动之后，我们首先给大家介绍系统的电子图书功能。双击电子图书图标，进入如下界面：

图 3.28.2.1 电子图书文件浏览界面

在此界面下我们可以浏览整个 SD 卡上面的内容，在电子图书功能下面，我们只查找能打开的文件，支持的文件有：LRC、TXT、.C 文件、.H 文件。文件的选择，和触摸手机是一样的，只是翻页有些不一样，向上滑动/向左滑动可以实现向上翻页，而向下翻页，则是通过向下滑动/向左滑动。找到.txt 文件如下图所示：

图 3.28.2.2 找到 txt 文件

双击该 TXT 文件，打开，如下图所示：

图 3.28.2.3 打开 TXT 文件

打开后，我们就可以开始浏览这个文本文件了，通过点击屏幕的下 1/5 部分向下翻页，该程序不支持想上翻页！如果想退出来，则直接点中屏幕的中央 3s 中左右（不要移动笔头哦），就会弹出返回按钮出来，如下图所示：

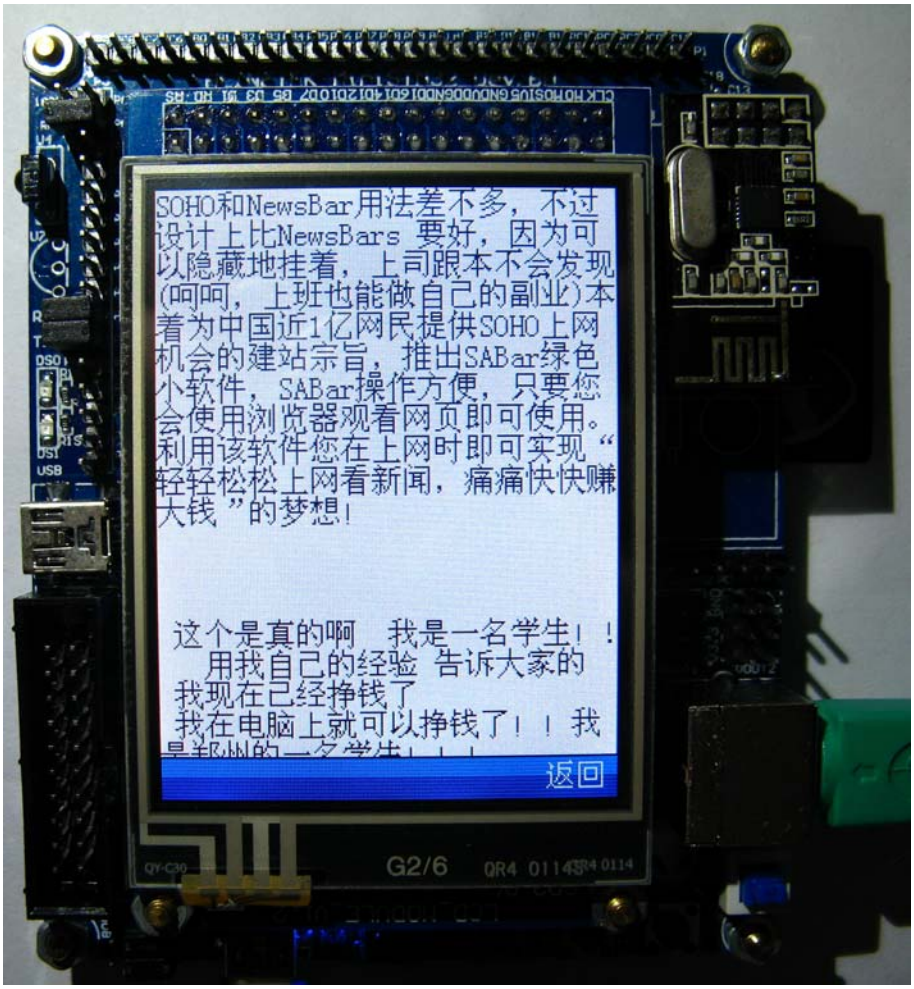


图 3.28.2.4 返回按钮弹出

在文件结束的时候，也会自动弹出返回按钮的，当文件已经没有后续的内容了，再点击下一页的时候，就会弹出返回按钮，界面和上图是一样的。

3.28.3 数码相框

上面给大家介绍了电子图书的功能，接下来介绍第二个功能：数码相框。这个功能可以用来浏览 SD 卡上的图片，包括 BMP、JPEG、JPG 等格式，大小没有限制，但是 BMP 只限 16、24、32 位，并不支持单色和 256 色的 BMP 图片。对 JPG 和 JPEG，其格式也要求是 JFIF 格式的，对 EXIF 格式不支持，但是你可以通过 XP 的画图工具打开 EXIF 的 JPG，再保存，就会转为 JFIF 格式。如果图片太大的话，系统会自动压缩图片大小以使其适合我们观看，当然最好不要放太大的图片，否则要解码很久才能看到。

双击数码相框的图标，进入如下界面：

图 3.28.3.1 数码相框文件浏览界面

这里和电子图书的浏览是一样的，只是这里浏览的是 JPG/BMP/JPEG 格式的图片，而不是文本文件罢了。找到你需要打开的图片，双击打开，开始浏览图片，如下图所示：

图 3.28.3.2 图片显示效果 1

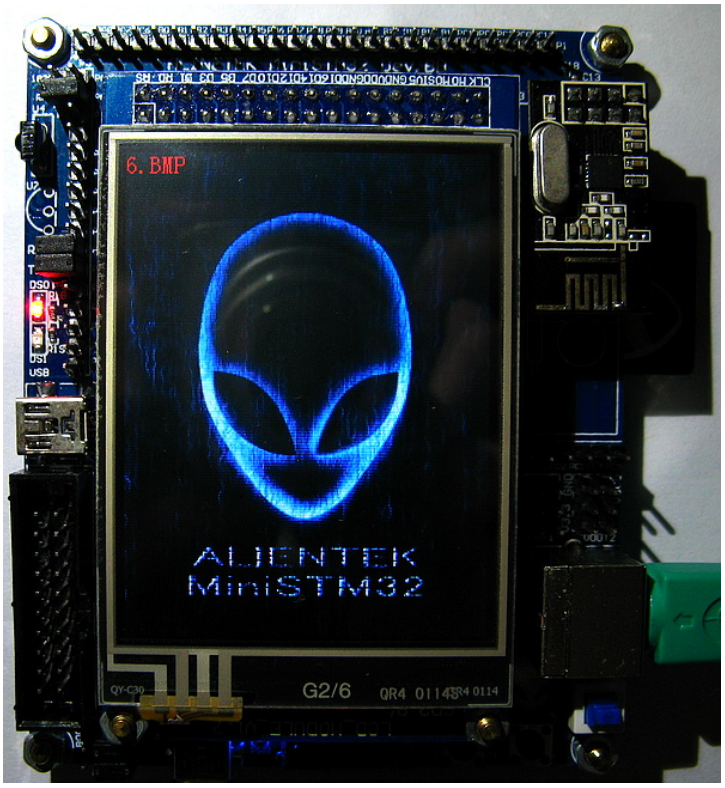


图 3.28.3.3 图片显示效果 2

系统会循环播放当前目录下的所有图片，通过单次点击屏幕中央可以暂停（此时 DS0 不再闪烁，如果不是暂停状态，DS0 会闪烁的），通过按屏幕的下 1/5 处切换为下一幅图片，通过点击屏幕的上 1/5 切换为上一幅图片。和电子图书一样，是通过长按中间区域弹出返回按钮，如

下图所示：

图 3.28.3.4 返回按钮弹出

通过点击返回，退出数码相框功能。

3.28.4 拼图游戏

这是本系统的第三个功能，在之前的实验中并没有提到。其实就是一个简单的拼图游戏，双击拼图游戏图标，进入游戏选择界面，如下图所示：

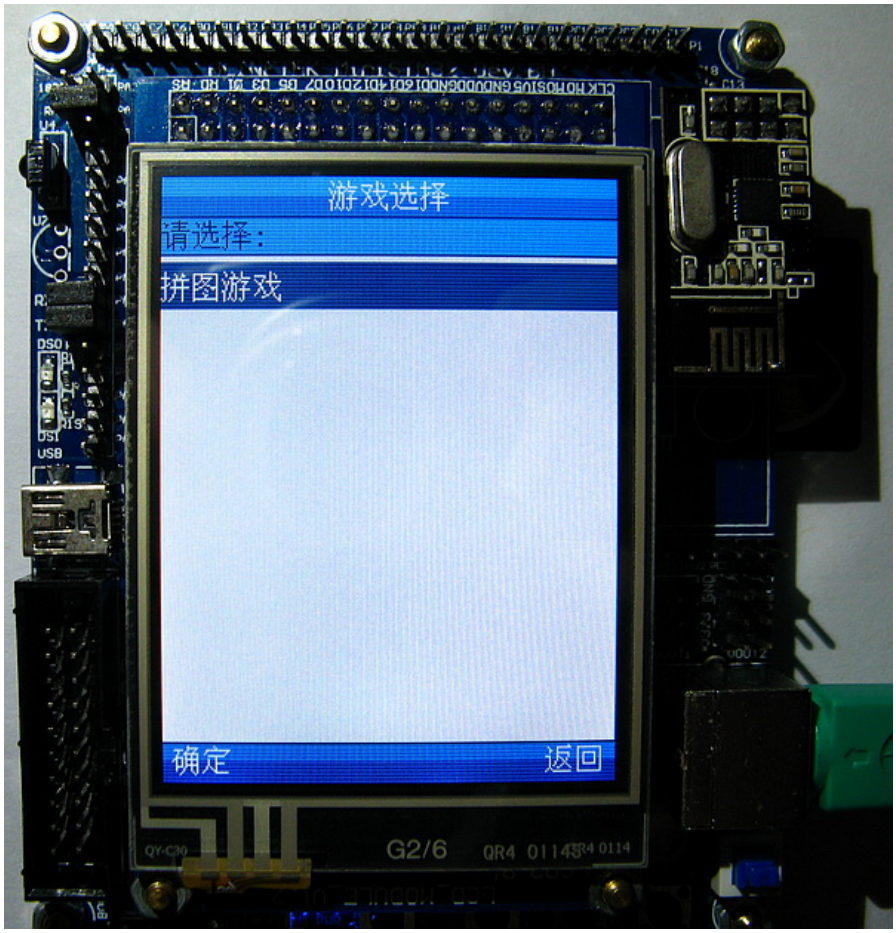


图 3.28.4.1 游戏选择

因为里面只有一个游戏，所以在进入该界面的时候只有一个选项，也就是拼图游戏。大家可以通过往程序里面添加其他游戏程序，嵌入到这个界面下，就可以选择多个游戏了。

这里我们只能点击拼图游戏，进入如下界面：

图 3.28.4.2 拼图游戏选择

在这个界面下，有 3 或 4 个选项，当你之前玩过，而没玩完的时候，就有四个选项(多了一个继续选择项)，当你之前玩完了，或者选择了新的难度等级，则只有三个选项（没有继续这个选择项）。

选择继续，则会继续之前的游戏，而选择新游戏，则重新开始一次新的游戏。选择最佳排行，则可以查看历史最佳数据，通过游戏设置可以设置游戏的难度。

我们点击新游戏，进入如下界面：

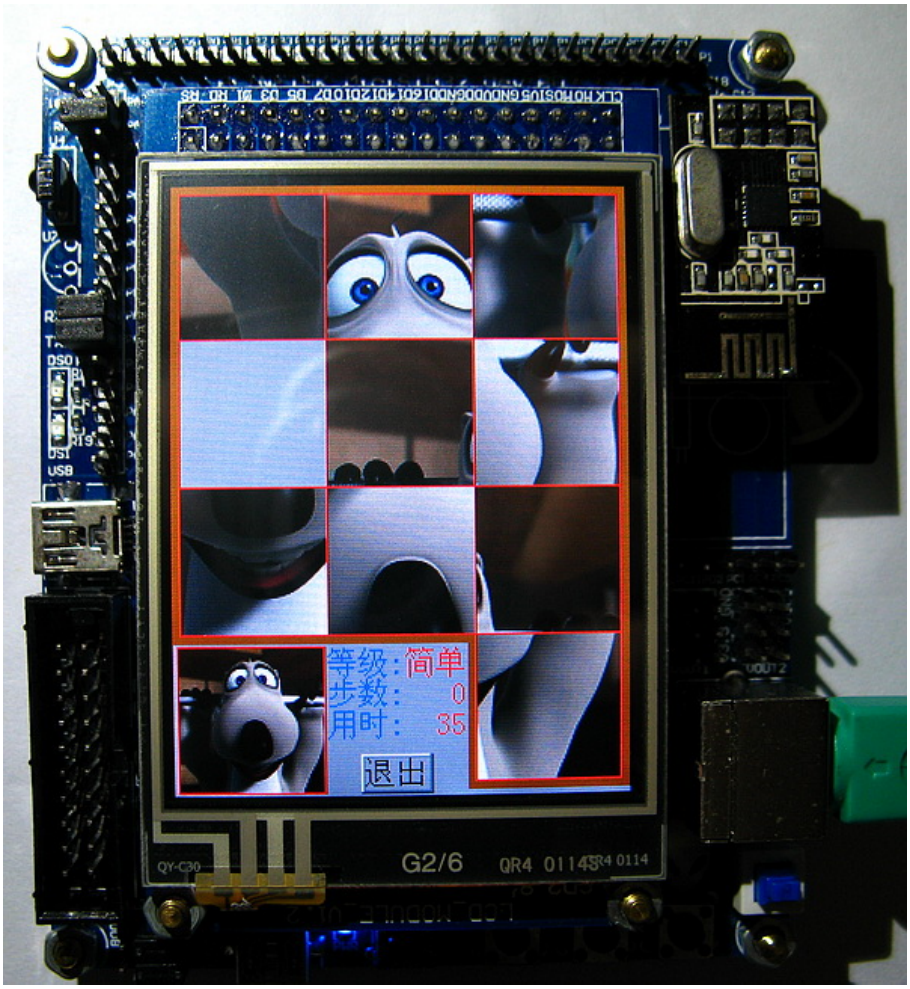


图 3.28.4.3 游戏进行中

图中显示我们的游戏等级为简单，也就是最简单的一级，只有 3*3 大小，很容易就完成了，在游戏完成的时候，程序会判断你的成绩是否创纪录了，如果创纪录，则会显示如下信息：

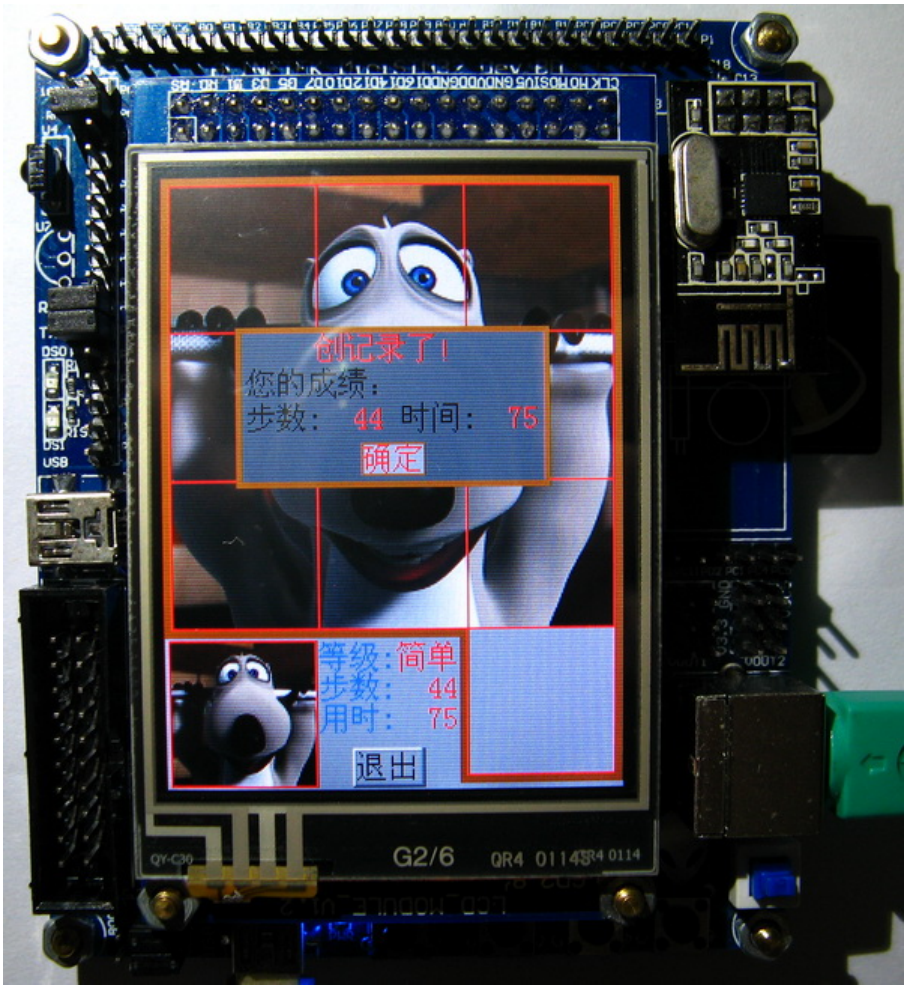


图 3.28.4.4 游戏完成

上图中，因为我们第一次玩，所以不论你玩的怎么样，只要完成了就是创纪录了。而当你的成绩没有之前最佳成绩那么好的时候，就会提示恭喜过关，而不是创纪录了。

我们可以通过最佳排行查看游戏的最佳结果，如下图所示：

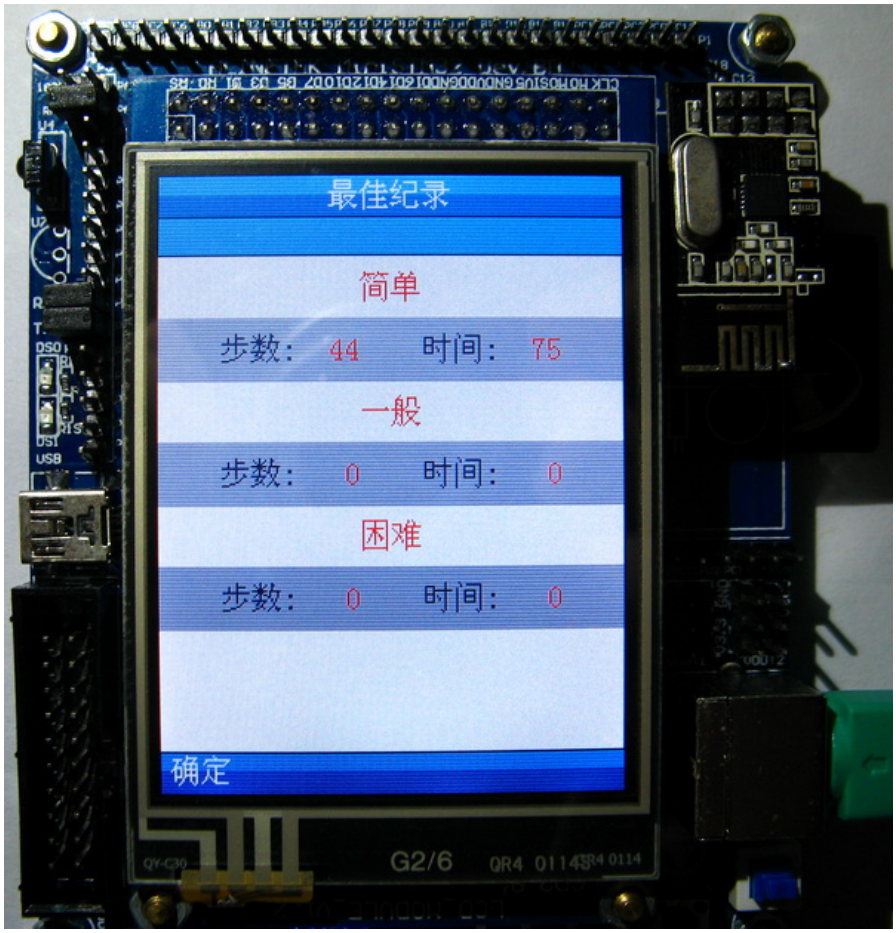


图 3.28.4.5 最佳排行

上图中显示了三个难度的最佳排行，因为一般和困难这两个等级我们都没有玩过，所以都显示为 0，而我们刚刚创造的简单级别的记录被保存了下来。

接下来我们看看难度选择，其界面如下图所示：

图 3.28.4.6 难度选择

这里有 3 个难度级别供我们选择：简单、一般、困难。简单属于 3*3 的拼图，一般则为 4*4 的拼图，而困难则为 5*5 的拼图。3*3 的拼图我们在之前已经看到了，下面我们看看 4*4 和 5*5 的拼图界面，如下面两幅图所示：

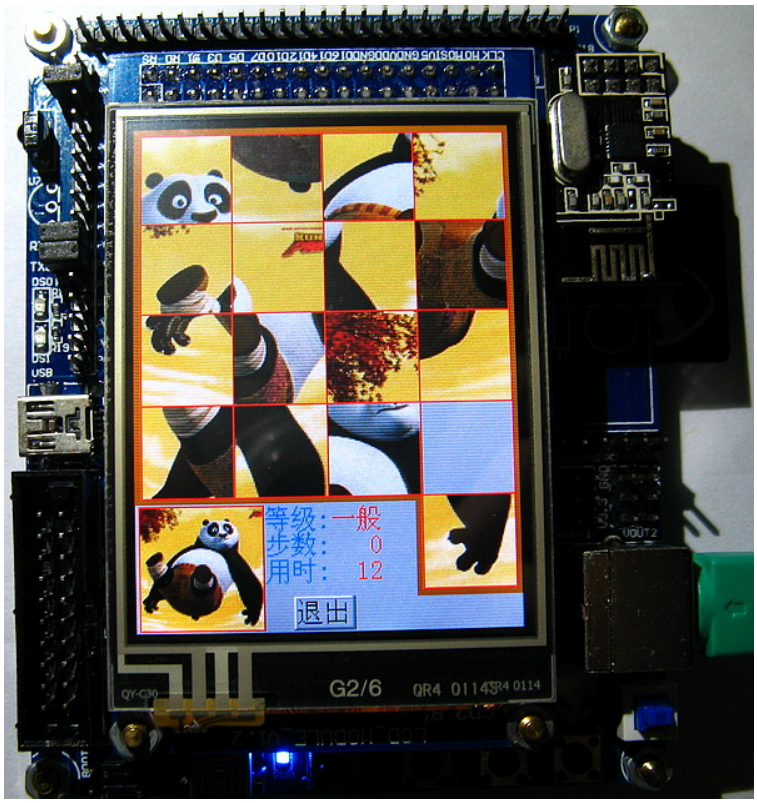


图 3.28.4.7　4*4 拼图界面

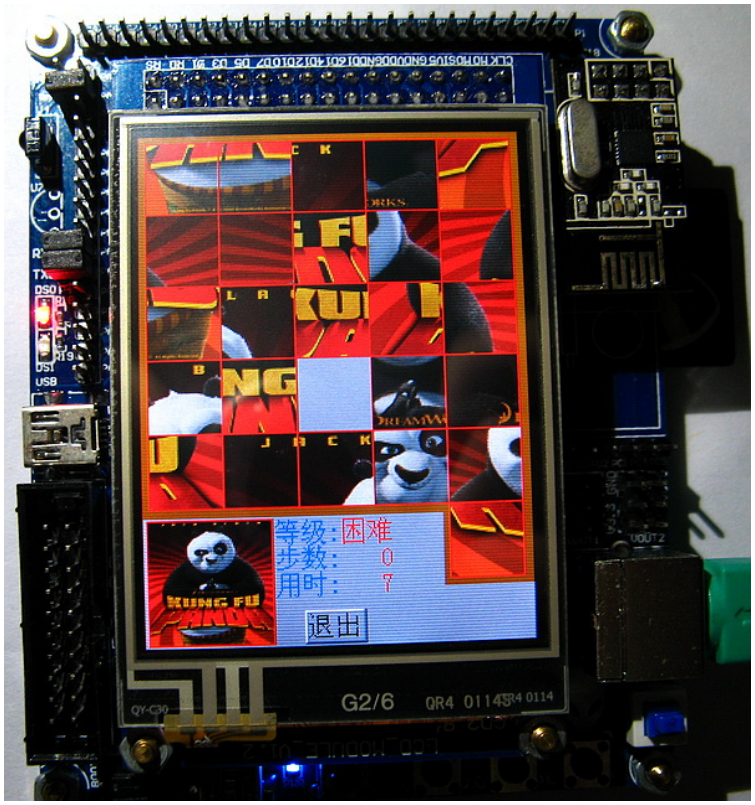


图 3.28.4.8　5*5 拼图界面

至此，我们的拼图游戏部分就介绍完了。

3.28.5 触摸画板

这是本系统的第四个功能，其实就是将我们之前的画图实验进行了一点扩充，在本功能下面，我们可以画一些简单的图，并可以切换画笔的颜色。

双击触摸画板图标，进入如下界面：

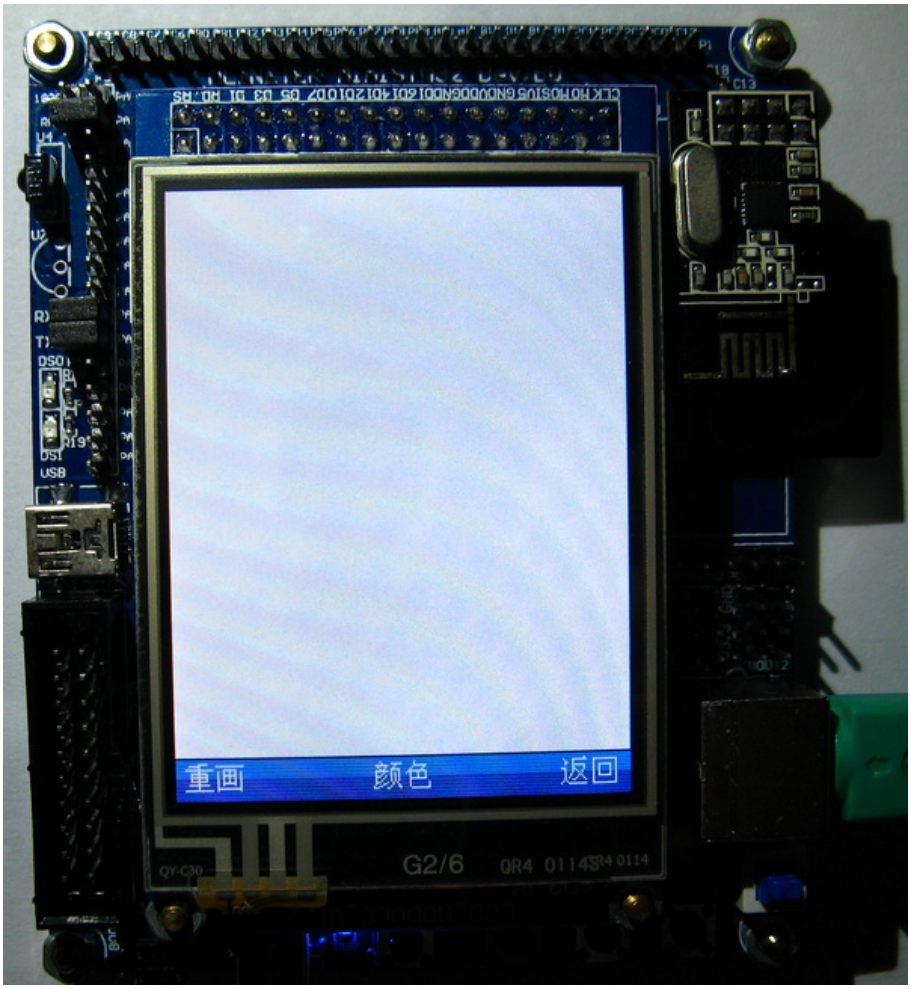


图 3.28.5.1　画图功能主界面

在该界面下，我们可以自己绘制喜欢的图片，默认的画笔颜色为红色，可以通过点中间的颜色区域来切换颜色。并且可以点重画按钮重新开始画图，实际画图效果如下面 2 副图片所示：

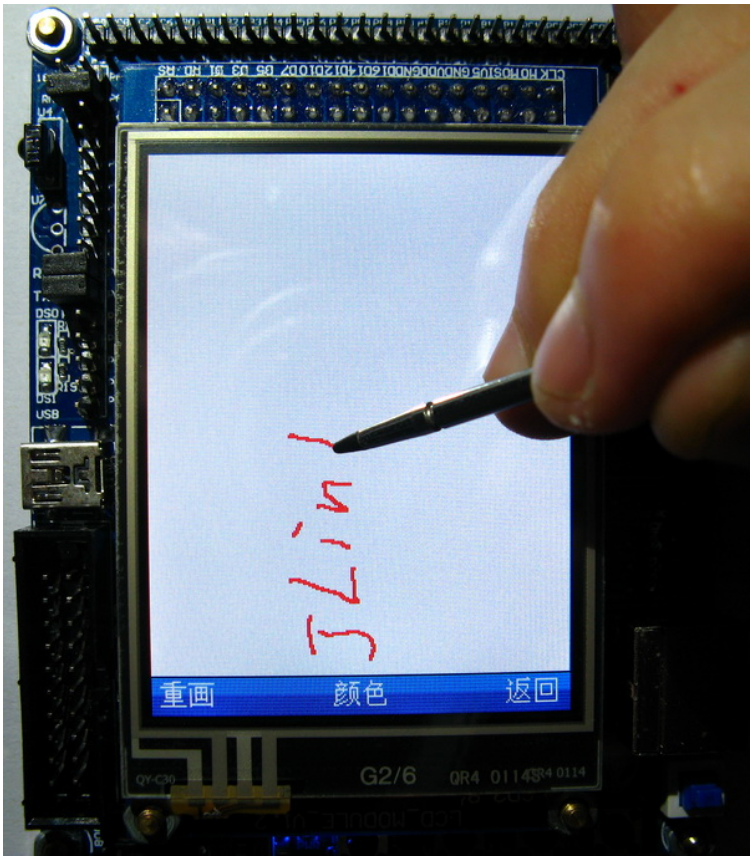


图 3.28.5.2　画图效果 1

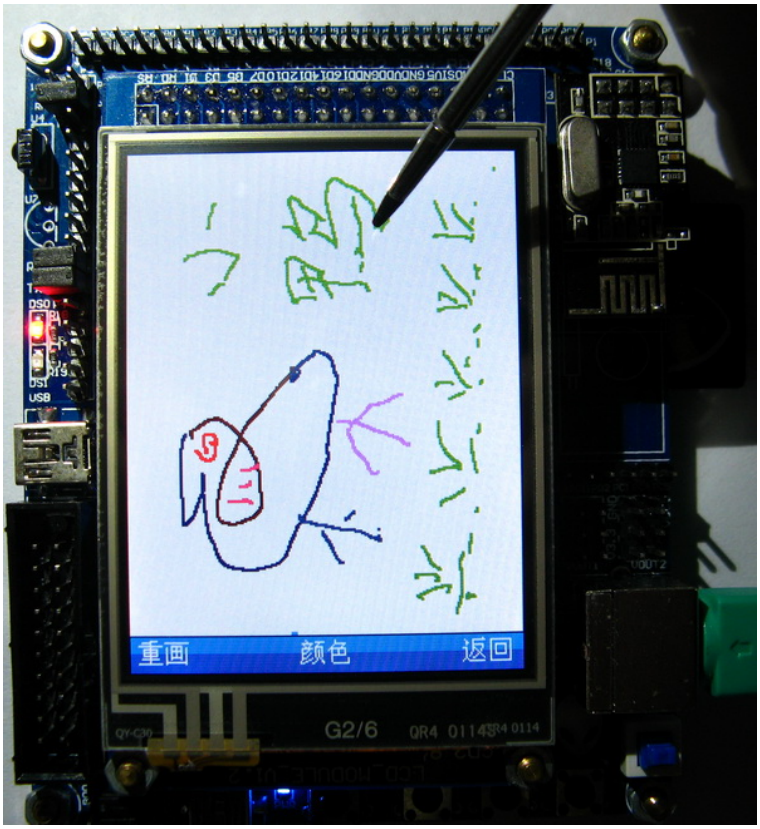


图 3.28.5.3　画图效果 2

我们通过点击颜色按钮，弹出颜色对话框，在这个框里面，可以选择你喜欢的颜色来画图，只需要在对应的小方格里面单击即可（注意不要点到颜色框的边框，如果点到边框，将会是无效的操作），如下图所示：

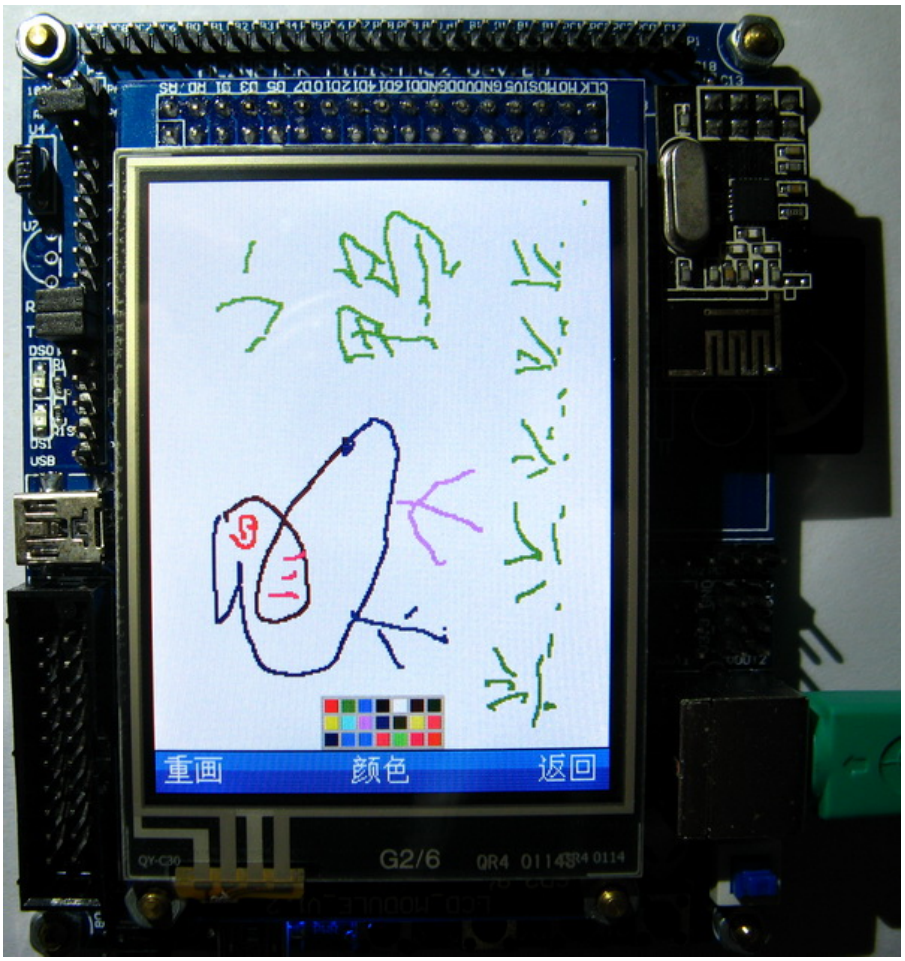


图 3.28.5.4　颜色选择

至此触摸画板就介绍到这里，通过点击返回回到主界面。

3.28.6 系统时间

这是本系统第 5 个功能，系统时间功能，这里仅用来演示时间功能（显示的时间是编译器编译时间+RTC 配置后运行的时间），同步显示温度功能。

双击系统时间，程序开始检测 DS18B20，如果不存在 DS18B20 则启用内部温度传感器，并提示相关信息，如下图所示：

图 3.28.6.1 系统时间启动界面

在程序启动完成之后，就进入了时钟界面了，如下图所示：

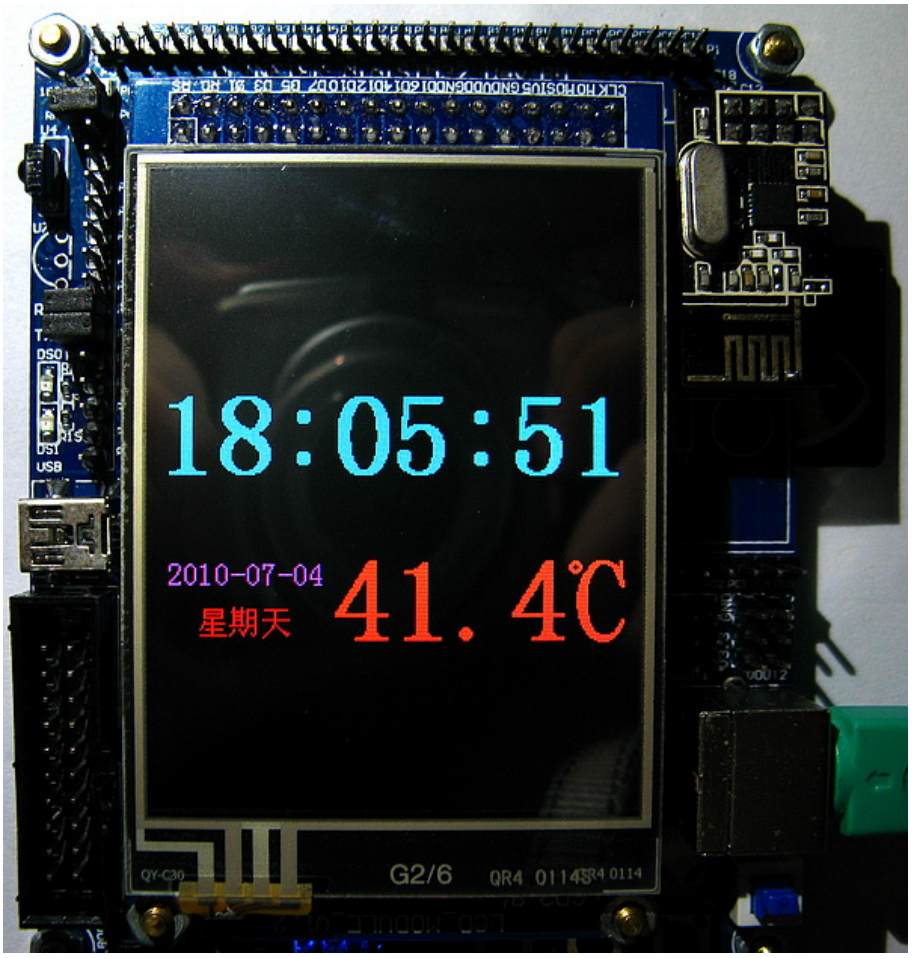


图 3.28.6.2 时钟显示

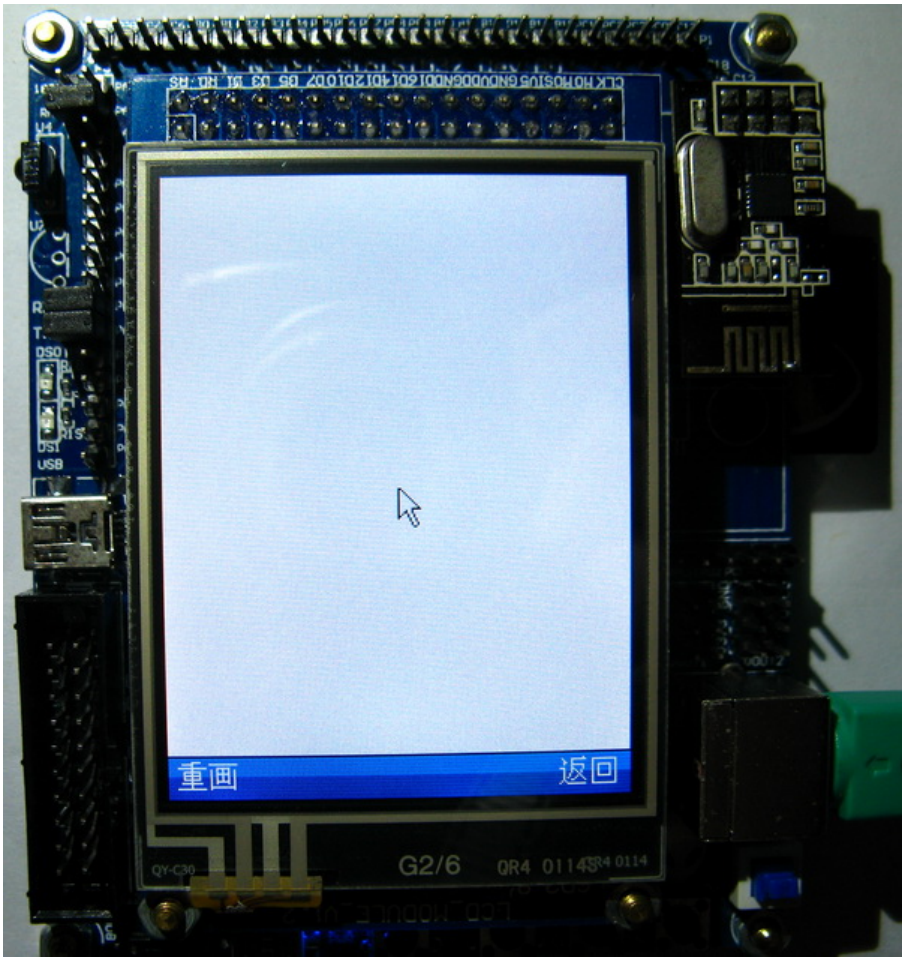
可以看到图中显示了时间，日期，温度等几个信息。温度比较高，是因为没有使用外部温度传感器，而用的内部温度传感器，而 STM32 的内部温度传感器所显示的温度是芯片内部温度，并不是我们实际的环境温度，而这个温度一般会比环境温度高不少。此处仅作参考而已。

系统时钟就介绍到这里，我们通过向左滑动退出该界面，回到主界面。

3.28.7 鼠标画板

这是本系统的第六个功能，该功能通过 PS/2 鼠标控制，在 LCD 上实现画图。

双击鼠标画板图标，进入如下界面（这里假定鼠标已经接上，并且被成功初始化，否则会提示错误，并自动返回）：

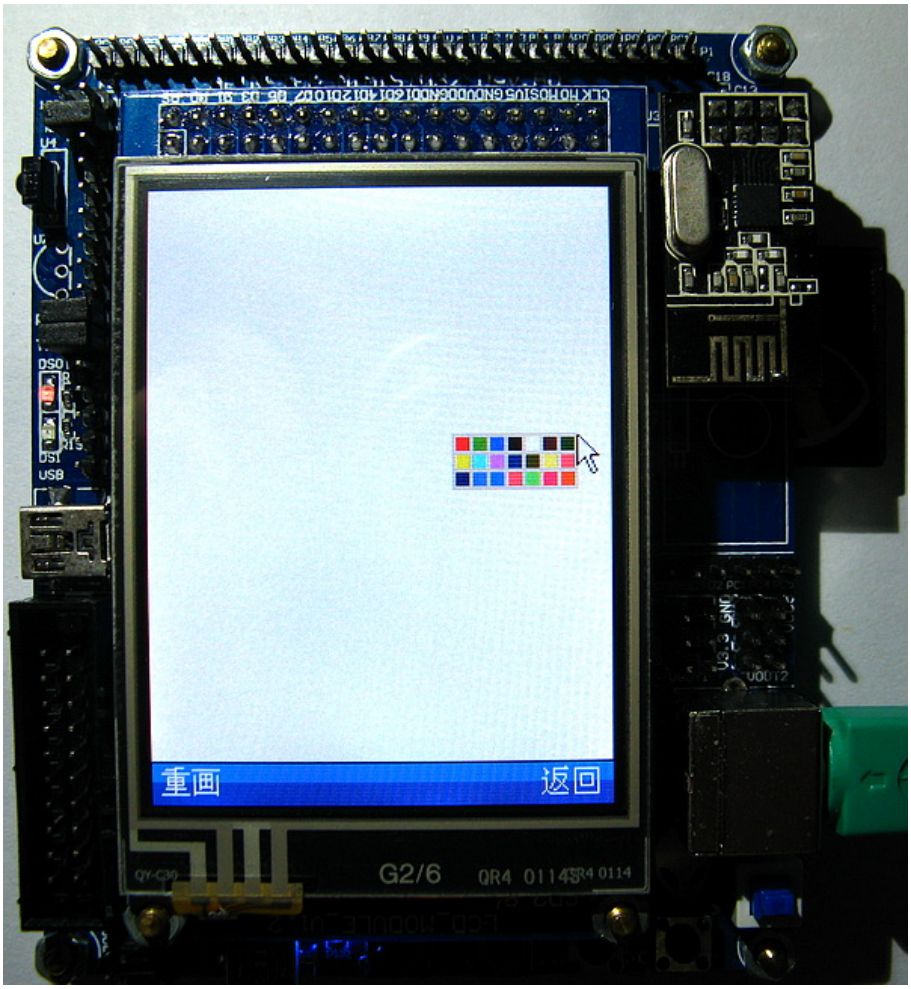


3.28.7.1 鼠标画板

可以看到屏幕中央显示除了光标，我们移动鼠标就可以看到光标跟着你的移动而移动了，就像在电脑上使用鼠标一样。

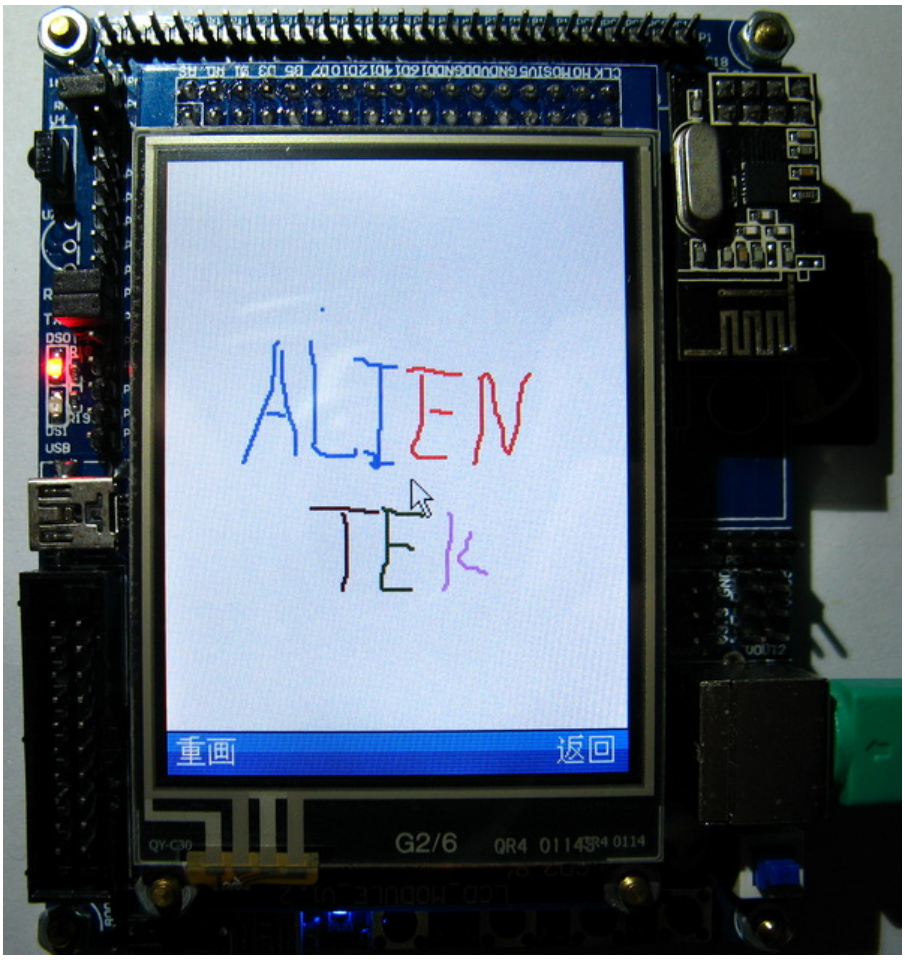
通过按住鼠标左键并移动来画图，这和 XP 上的画图是一样的，不一样的地方是我们的光标会变成当前画笔的颜色，而在你松开的时候，又恢复标准的光标。我们还可以通过按下鼠标滚轮（中间键）来擦除内容，此时光标会变为黑色。

可以通过鼠标右键弹出颜色框，来选择你所需要的颜色，如下图所示：



3.28.7.2 右键选择画笔颜色

图中，我们只需要把光标移到所需要设置的颜色上面，再单击鼠标左键，就可以实现画笔颜色设置了。实际画图效果如下图所示：



3.28.7.3 鼠标画图效果

同样，我们可以点击重画来清除之前所画的内容。通过点击返回按钮回到主界面。鼠标画板就给大家介绍到这里。

3.28.8 USB连接

这是本系统 的第七个功能，这个功能完全就是实验 27 的内嵌，也就是 USB 读卡器的功能。通过此功能，我们就可以随时往开发板的 SD 卡发送文件或者读取文件了，而不需要拔卡，然后搞个 SD 卡读卡器，再重启。

双击 USB 连接，进入如下界面（假定 USB 已经连接到了电脑）：

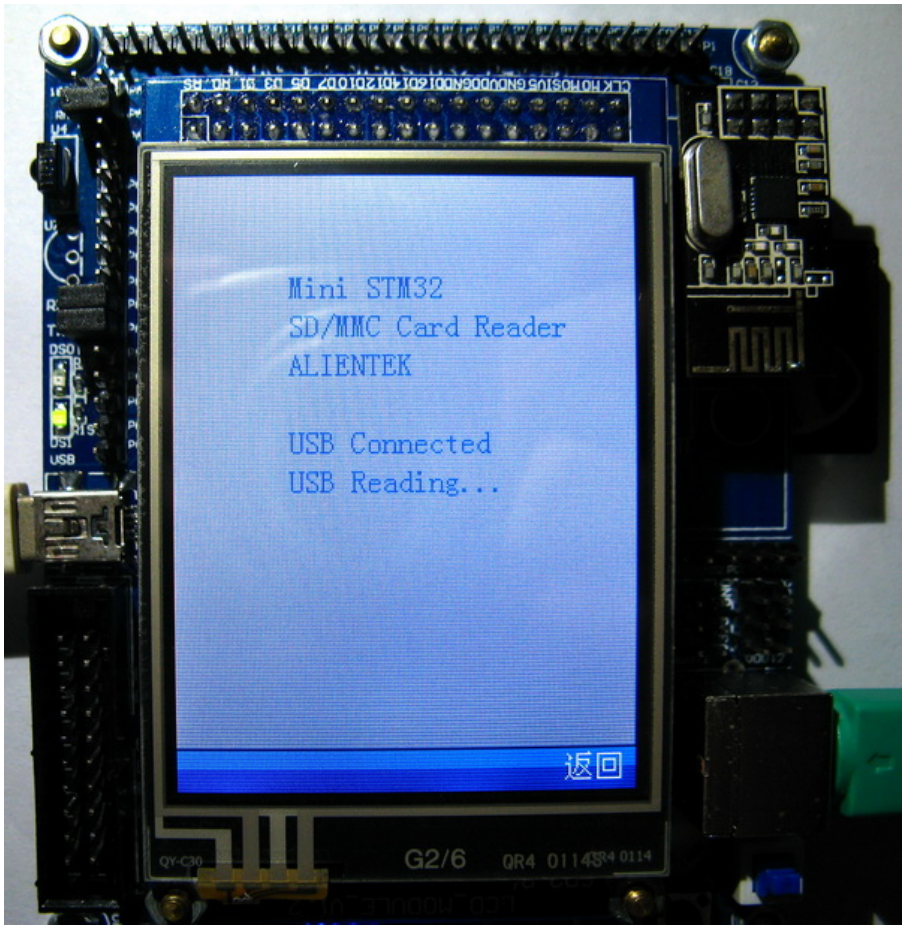


图 3.28.8.1　USB 连接中

此时 DS1 开始闪烁，该灯用于只是 USB 与 SD 卡之间有数据通信。在 USB 配置成功之后，我们可以看到电脑上显示出了 SD 卡的盘符，如下图所示：

图 3.28.8.2　电脑找到 SD 卡盘符

上图中，CANON_DC 就是我们开发板上的 SD 卡盘符，证明我们电脑已经识别出了 SD 卡，此时我们就可以往 SD 卡里面写入数据，或者读取数据了。

在开发板的液晶上面，显示如下内容：

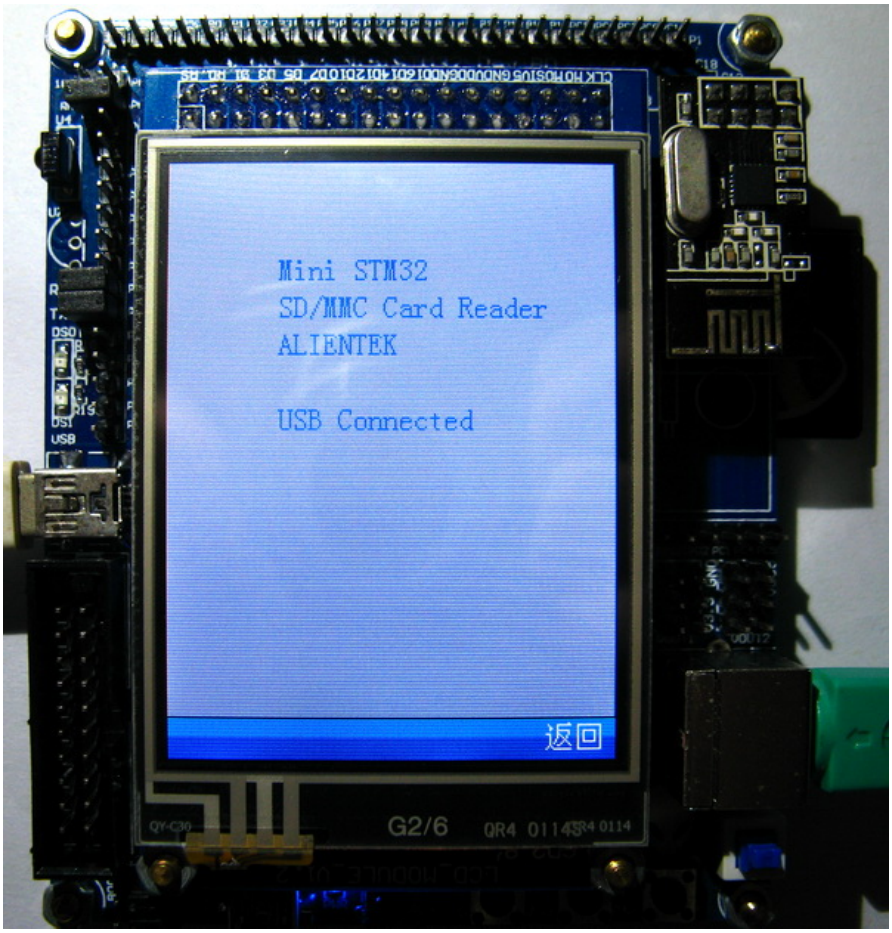


图 3.28.8.3　USB 已经连接

此时程序正在等带 USB 的操作，并且会监控 USB 的状态，如果被删除了，则会显示 USB DisConnected，提示 USB 连接已经断开。

当我们向 SD 卡写入数据的时候，可以看到 LCD 上显示 USB Writing…提示正在进行 USB 写操作，如下图所示。在 USB 读写期间，我们是不能通过返回按钮返回的。必须在 USB 空闲的时候才能返回！

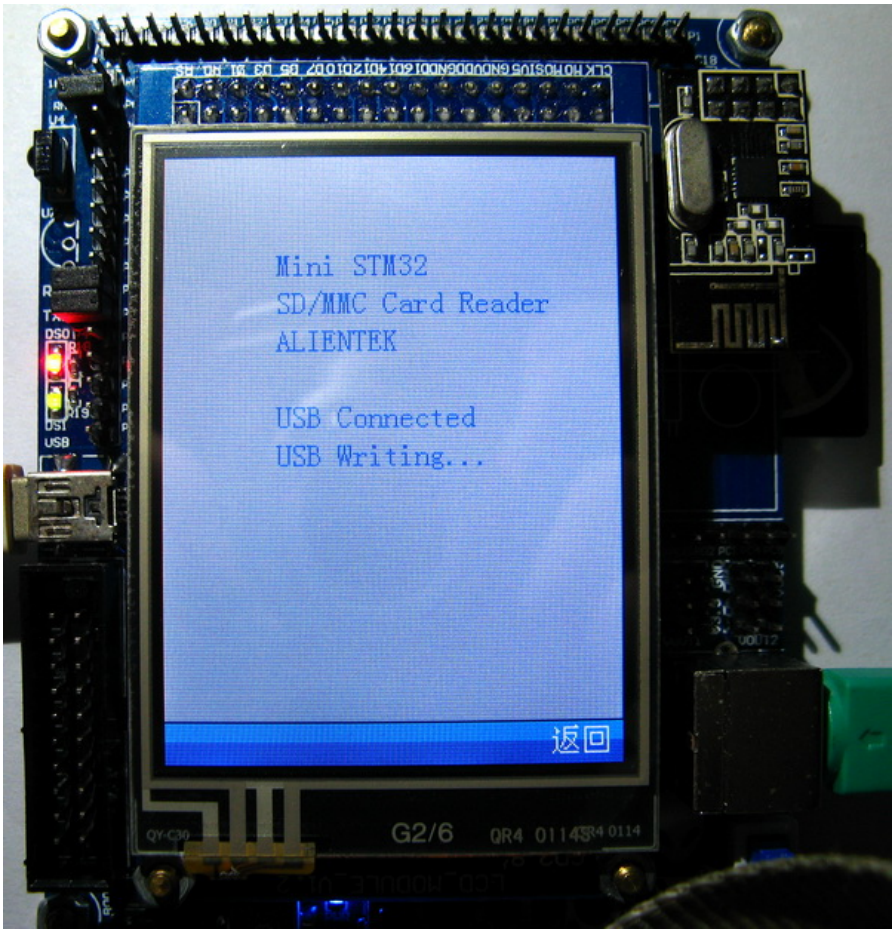


图 3.28.8.4　USB 写数据

USB 连接部分，我们就为大家介绍到这里。

3.28.9 红外遥控

这是本系统的第八个功能，实现红外遥控的接收及解码，这里也仅仅是实验 21 的内嵌而已。双击红外遥控图标，进入如下界面：

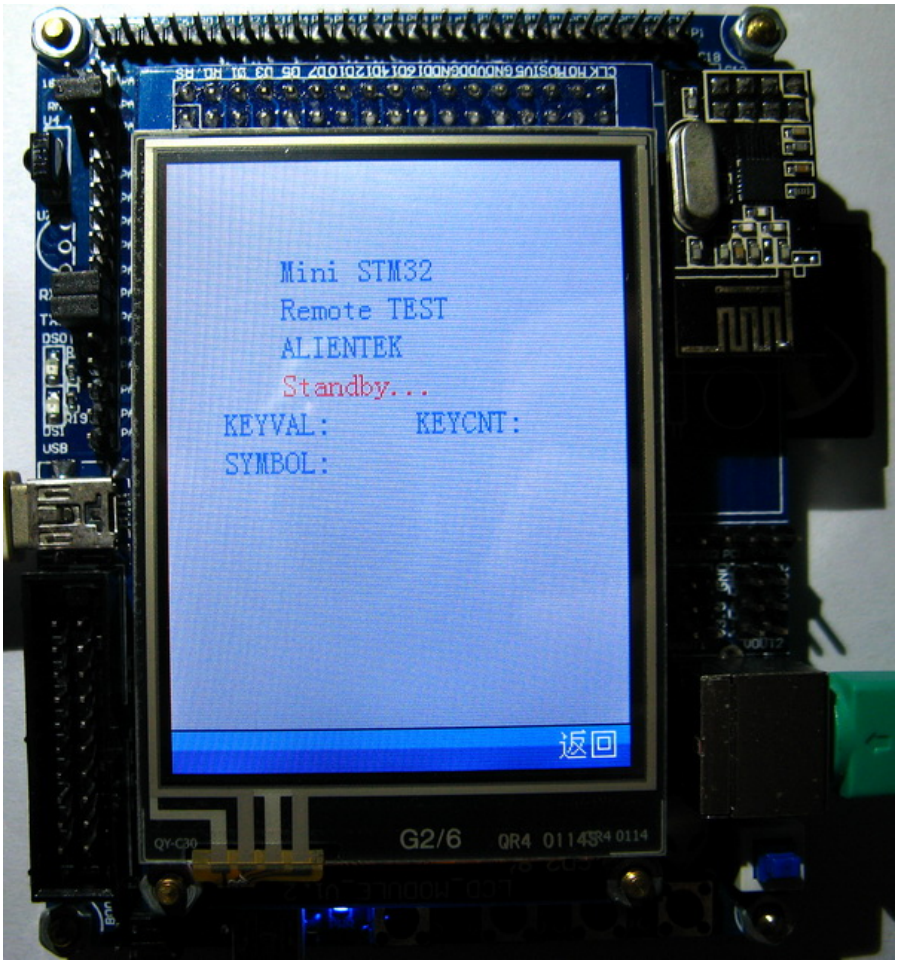


图 3.28.9.1 红外遥控接收界面

此时，可以看到 Standby…字样不停闪烁，提示红外遥控接收已经准备好了，此时只要你按下遥控器的任何一个按钮，就可以看到按钮的键值、图标以及按键次数了。如下图所示：

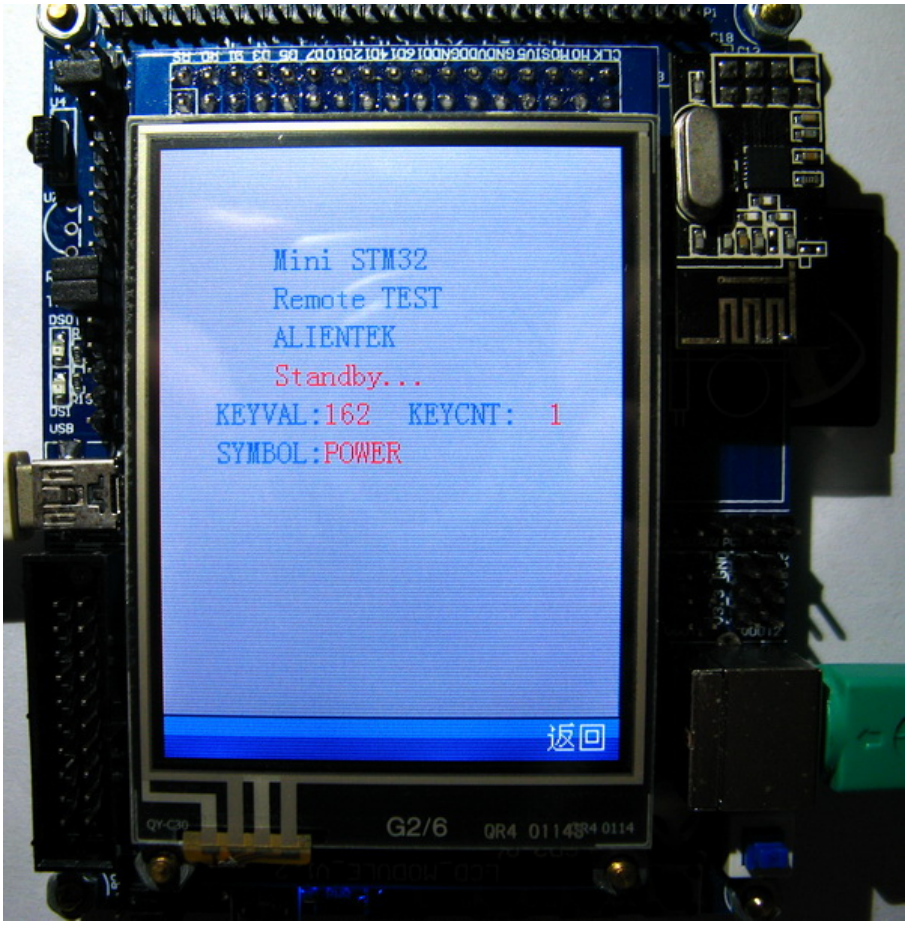


图 3.28.9.2 红外遥控解码

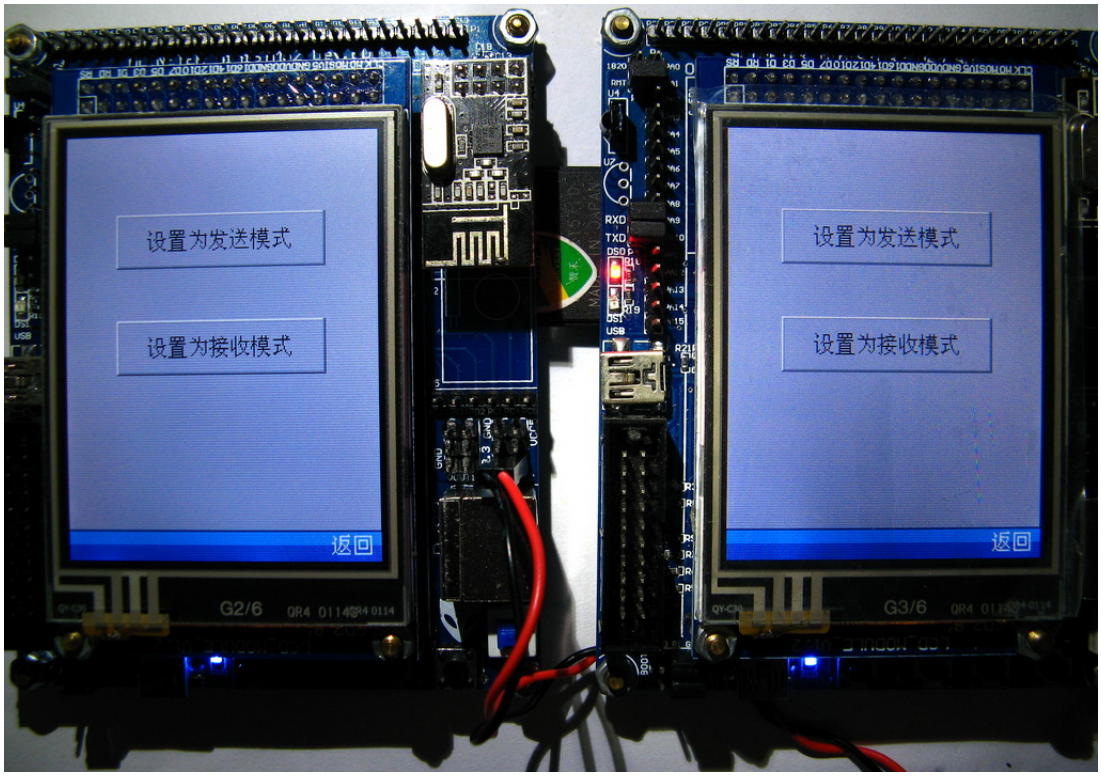
在红外遥控这个功能下面，我们的触摸屏被禁用了（因为红外解码和触摸中断共用了一个中断），必须通过按钮才能返回，我们对应的返回按钮是 KEY1，通过按 KEY1 可以返回到主界面。回到主界面，触摸屏即恢复正常使用。

红外遥控部分就介绍到这里。

3.28.10 无线传书

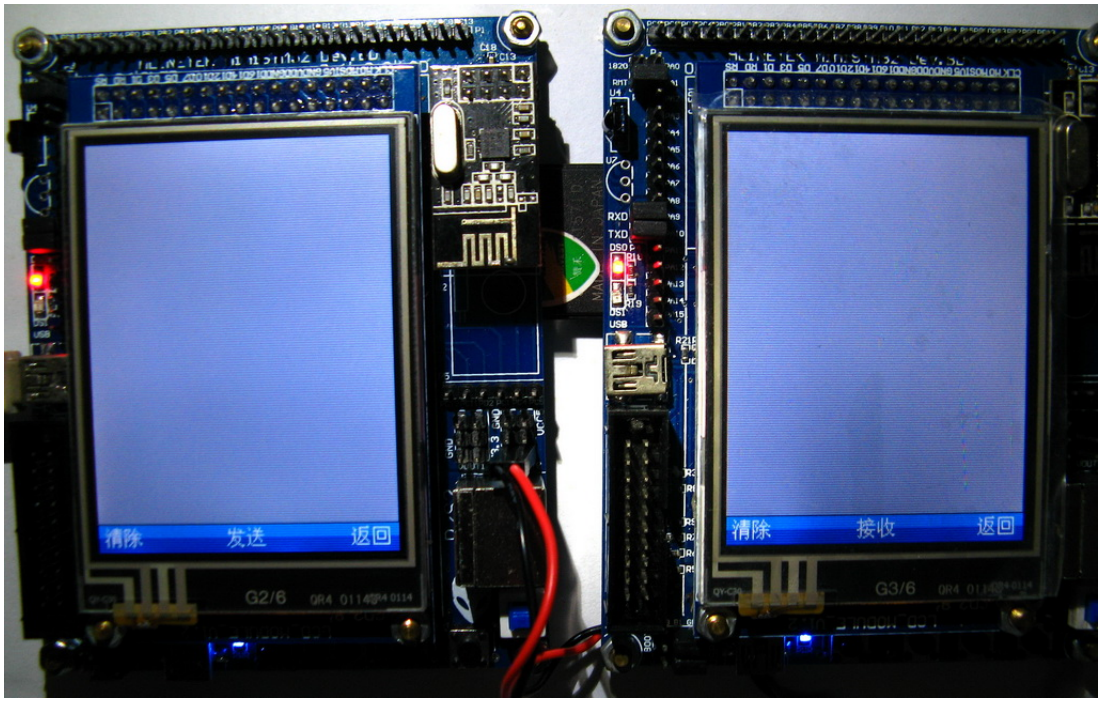
这是本系统的最后一个功能，用于两个及以上的开发板之间的无线数据传输，此部分需要用到 24L01 模块，至少 2 套开发板。

我们设置一个开发板为发送模式，另外一个为接收模式（如果有 2 个以上，除了一个做发送，其他的都设置为接收）。双击无线传书的图标，系统会先检测是否有无线模块存在，如果没有则提示错误，并返回。如过有，则进入如下界面：



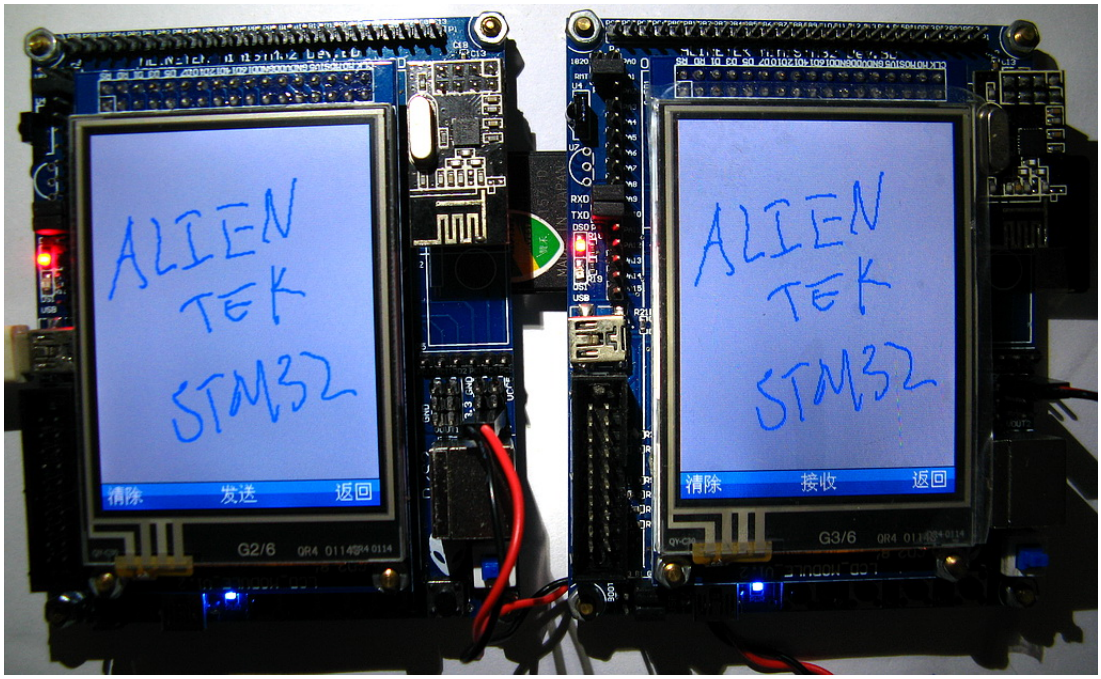
3.28.10.1 无线传书设置

我们将两个开发板一个设置为接收模式，一个设置为发送模式，在设置完成之后如下图所示：



3.28.10.2 设置完成

此时，我们在设置为发送的开发板上用手写字，则可以在另外一台开发板上看到一模一样的内容，如下图所示：



3.28.10.3 传书成功

这样我们就实现了无线传书的功能，我们也可以通过按清除按钮，把所写内容清除掉，在主机按清除按钮可以将从机的也顺带清除，而从机的清除，则只是清除自己的。在主机按返回，从机也会跟着返回，而在从机按返回，主机则不会返回。

无线传书就介绍到这里。

至此，整个实验的功能，给大家介绍了一遍。很多功能做得不完善，希望大家可以在此基础上增加自己的代码，从而丰富整个功能。