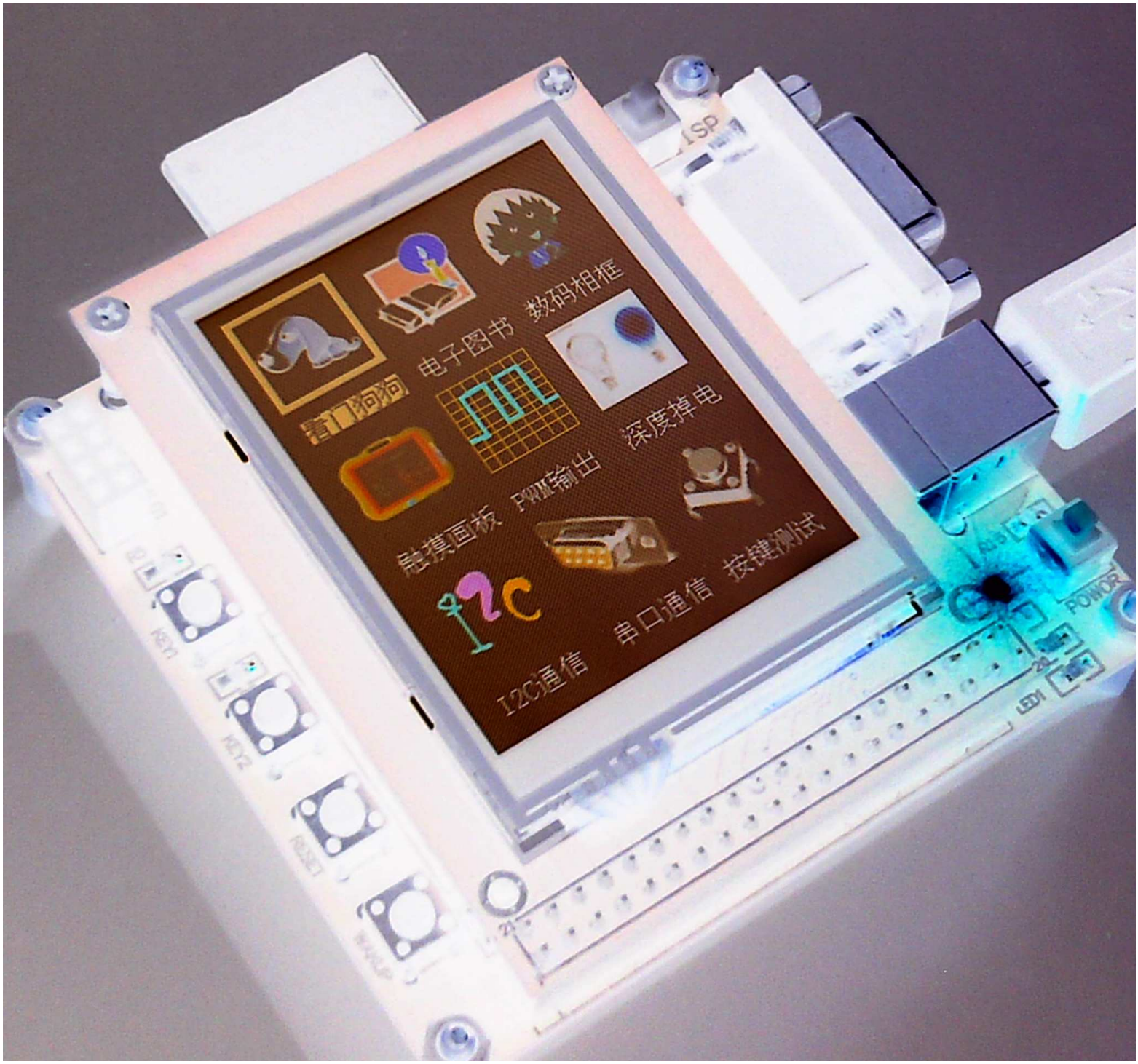
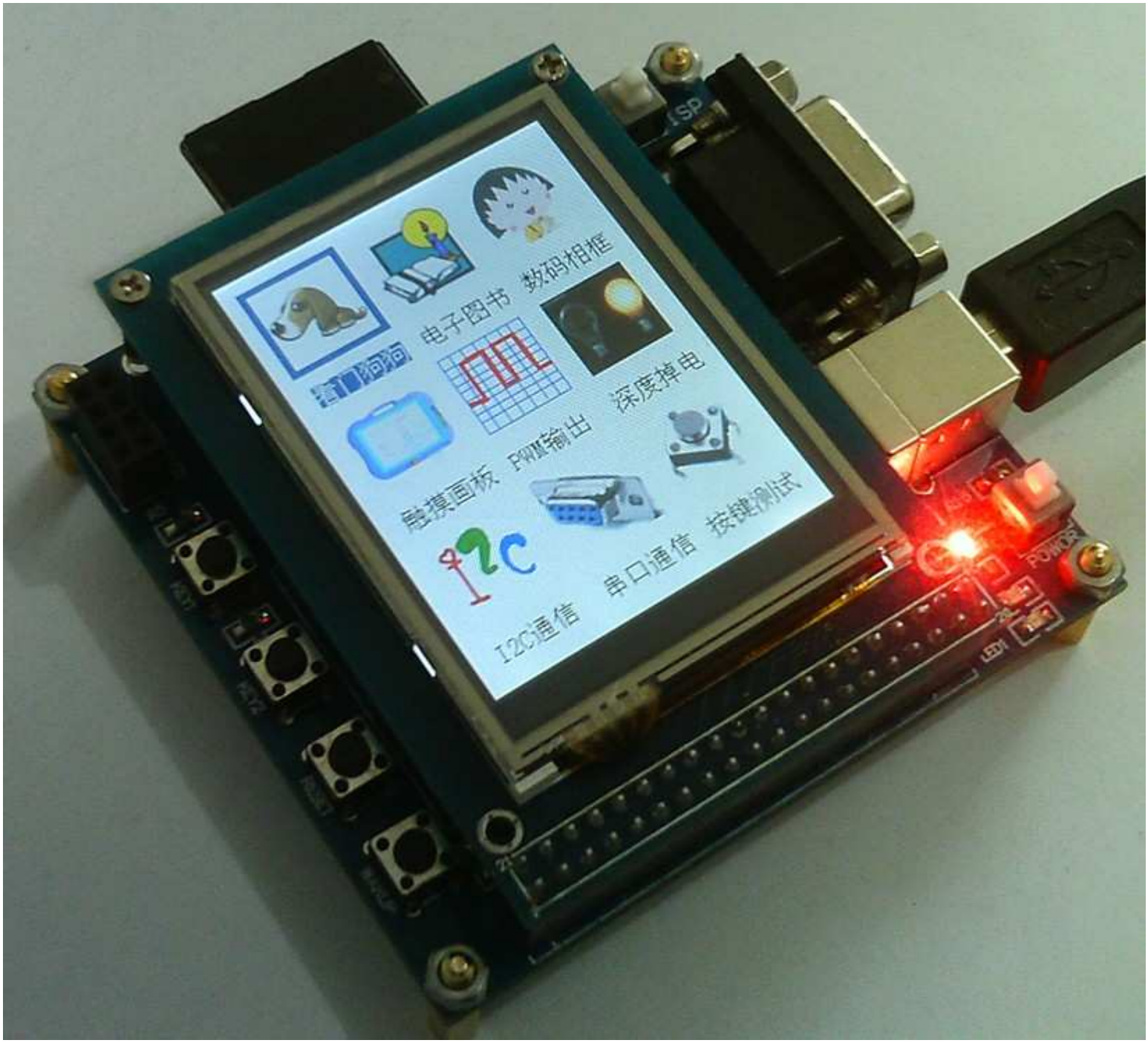

电子图书
数码相框
深度掉电
PWM输出
触摸画板
I2C通信
串口通信
按键测试


电子图书
数码相框
深度掉电
PWM输出
触摸画板
按键测试
串口通信
I2C通信
ISP
POWER

总览 LPC1114（有 51 思维困扰必读）

下面是我们即将要搞定的 32 位单片机 LPC1114 的真面目！

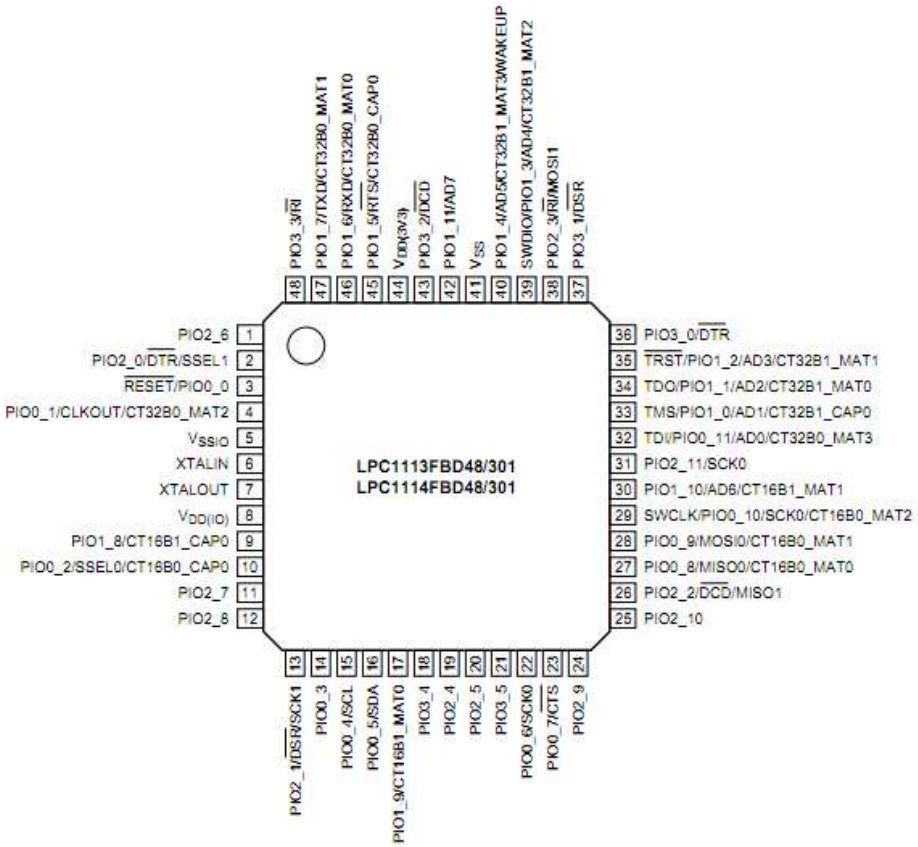
它一共有 48 个脚！

其中 2 个晶振引脚，4 个电源引脚，42 个通用输入输出脚。（2+4+42=48 根脚^_^）

普通的芯片不是只有 VCC 和 GND 两个电源脚吗？它为什么有四个？这是因为在这个芯片的内部集成了 ADC（模拟数字转换器，功能和你们知道的 ADC0804 是一样的，也就是说有了它，连外部的 ADC0804 也省了，直接把要测得电压信号连接到它的测量脚上，它就知道是多少 V 的电压了，到底是哪根脚，后面介绍！），除了正常的 VCC 和 GND 脚，还有两个就是 ADC 的参考电源正脚和电源地！一般情况下，VCC 和参考 VCC 相连，GND 和参考 GND 相连！

42 个通用输出输入脚(GPIO)：P0 口（P0.0~P0.11），P1 口（P1.0~P1.11），P2 口（P2.0~P2.11），P3 口（P3.0~P3.5）。看到了吧！P0，P1，P2 口各有 12 根脚，P3 口有 6 根脚！

下面是 LPC1114 的芯片引脚原理图！



看到上面的图，是不是感觉到很乱呀！它的引脚有时候是要复用功能的！比如第 9 脚！找到第 9 脚，你会看到它的标识为：PIO1_8/CT16B0_CAP0。它的意思就是说这根脚既可以作为 PIO1_8 脚用，也可以作为 CT16B0_CAP0 脚用。（PIO1_8 就是我们常说的 P1.8 脚；CT16B0_CAP0 就是我们常说的 16 位定时器计数输入脚）其实和我们用过的 AT89C2051 单片机上 P3.0 脚 P3.1 脚可以复用为 RXD 脚和 TXD 脚是一样一样的，只不过，LPC1114 的复用引脚要比 AT89C2051 多很多而已！

LPC1114 作为一个 32 位的单片机，它的寄存器也基本上都是 32 位的。先来回顾一下 8 位单片机的寄存器。比如“IE”“SBUF”等，很眼熟吧！IE 作为 51 单片机的中断控制寄存器：它的定义是下面这个样子！

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
EA	保留	ET2	ES	ET1	EX1	ET0	EX0

再来看一下 32 位单片机的寄存器！比如 AHBCLKCTRL：

bit19~31	bit18	bit17	bit16	bit15	bit14	bit13	bit12	bit11	bit10
保留	SSP1	保留	IOCON	WDT	保留	ADC	UART	SSP0	CT32B1
bit9	**bit8**	**bit7**	**bit6**	**bit5**	**bit4**	**bit3**	**bit2**	**bit1**	**bit0**
CT32B0	CT16B1	CT16B0	GPIO	I2C	FLASH2	FLASH1	RAM	ROM	SYS

由上面的表格看到，32 位的寄存器里面能放更多的控制位，但也有很多是保留位。

看看上面的引脚图，你就应该想到，它既然有这么多的复用引脚，要能有效的控制它们的话，就必然会有很多的寄存器！大概有好几百个，我也没数过，诸如控制寄存器，状态寄存器，数据寄存器等等！不过，没有关系！我们没有必要把它们全都记住它们，当我们要用到它们的时候，看数据手册就可以了，我们要做的只是，把它们用熟练了！

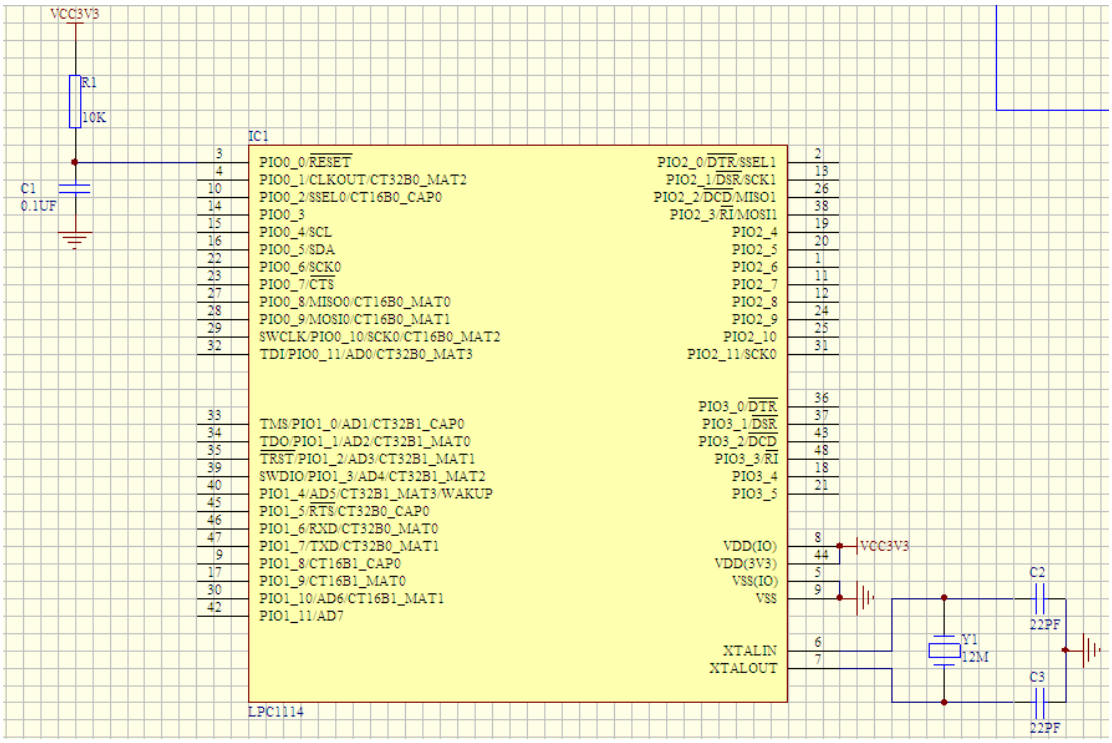
那么，除了上面提到过的 ADC，在 LPC1114 的内部还有些什么东西呢？看下面的表格就知道了：

GPIO	时钟控制单元	PMU（能耗管理）
WTG（看门狗）	CortexM0 内核	UART（串口）
ADC（模数转换器）	ROM(32K)	SSP(SPI)（串行通行）
CT(计数器)	RAM(8K)	I2C（I2C 通信）

其中，Cortex-M0 内核也有自己的外设：

NVIC	也就是我们常说的中断控制器（高效能的）
SysTick	系统定时器，专为跑操作系统设计；如果不上操作系统，可以把它当做一个普通的定时器用。比如作为延时函数用，相当划算！
SCB	系统控制模块（一般不用）
Debug	系统调试接口，配合 JTAG 或 SWD 调试

学 51 都是从构建最小系统开始的！那么我们也先给 LPC1114 构建个最小系统吧！如下图：



由上图可以看到，它和 51 的最小系统也是一样一样的！有一个区别就是，LPC1114 的 RESET 引脚（也就是 51 当中的 RST 引脚）也是复用的，它复用到了 P0.0 脚上，该引脚在默认情况下是 RESET 引脚，所以系统一上电，就可以正常工作，当然，你也可以通过配置 IOCON(引脚配置寄存器)把它当做 P0.0 脚使唤！

（在这里，我觉得有必要说一下单片机上电的过程和复位引脚的一些知识，你若知道，可以跳出这段蓝体字，若想听我唠叨，就接着往下看吧。我这里所提到的单片机，是指广泛意义上的单片机，也就是说绝大部分单片机都是通过下面的方式进行的：我们都知道单片机里面都有“程序存储器”，也就是我们常说的有多大容量的 FLASH，FLASH 通常被做为单片机里面的程序存储器，相当于 PC 机上的 ROM，硬盘之类的东西，我们在 PC 机上的开发环境里面写好了程序以后，会下载到单片机里的 FLASH，当单片机上电的时候，就会执行 FLASH 里面的程序了，同时我们还要求要从 FLASH 的程序开始地址执行，要不然就会乱，所以才有了 RESET（RST）引脚，这个引脚的外面连接有 RC 振荡器，RC 振荡器的工作原理是这样的：电容充电，电阻放电。当系统刚上电的时候，电容两端一下子突然有了电，就在这一瞬间，这个直流电对于电容来说就是交流电，电容通交流阻直流的道理大家应该都懂吧，这时候，电容相当于短路，RESET 引脚接到了地上，是低电平，然后过了一段时间（这段时间对于人类来讲就微不足道了）电容充满了电，(电容就是一个电池了^_^),电容连接到 RESET 引脚的这一端的电位为 3.3V，RESET 引脚又变成高电平了，这时候，单片机就要开始从程序地址 0 执行了。注意：如果刚才我提到的那个电容由低变高的时间不够长的话，单片机就不敢保证从程序地址 0 执行了，所有要选好 RC 的值，保证有总够长的时间！这个时间最小要多长，不同的单片机有不同的要求，总之，都应该是微秒级别的！所以再看 LPC1114，为什么它的第 3 脚默认为是 RESET 引脚，而不是 P0.0 脚，可不可以默认为是 P0.0，话都讲了这么多了，不用我解释了吧！）

其实！这还只是 LPC1114 的最小系统！殊不知！它还有最最小系统！最最小系统就是可以把上图中的晶振电路去掉！因为，LPC1114 的内部已经有一个 12M 的时钟源了，只不过精度没有晶振的精度高而已！如果你不用串口通信或精确定时的话，你的产品就可以完全不用设计晶振上去，对于缩小产品体积是很有帮助

的。聪明的你！现在应该一定能想到一个问题，既然可以去掉了外部晶振正常工作，系统为了保证正常运行，刚上电的时候系统默认为就是选择内部时钟发生器（数据手册上叫做 IRC）进行工作的！而一般情况下，我们利用外部晶振工作，数据手册上说，Cortex-M0 可以工作在 50MHz 下，并不是意味着如果我们希望它工作在 50MHz 下就需要外部接一个 50Mhz 的晶振。因为它里面还有一个倍频器（数据手册上叫做 PLL），看到名字你就知道了，它可以把频率翻倍，如果我们利用它把外部 12M 晶振倍频 4 倍的话，系统就可以工作在 48Mhz 时钟频率下了！（你之前在网络上看的视频就是 LPC1114 工作在 48Mhz 的效果）由此！你可以想到！我们给 LPC1114 编程的时候，首要的系统初始化工作就是把的时钟配置好了！其它的工作都是在此之后的！这一点，也是 ARM 单片机与普通单片机的一个区别！配置好了时钟，就可以开始其它的工作了，比如点亮一个 LED，控制 LCD 显示等等！

第一章（起步）系统时钟配置

一、入门引导
二、时钟配置图(CGU)详解
三、时钟配置程序设计
四、时钟配置程序详解：
五、CLKOUT 引脚输出时钟程序设计
六、CLKOUT 引脚输出时钟程序详解
七、实验程序下载和使用说明

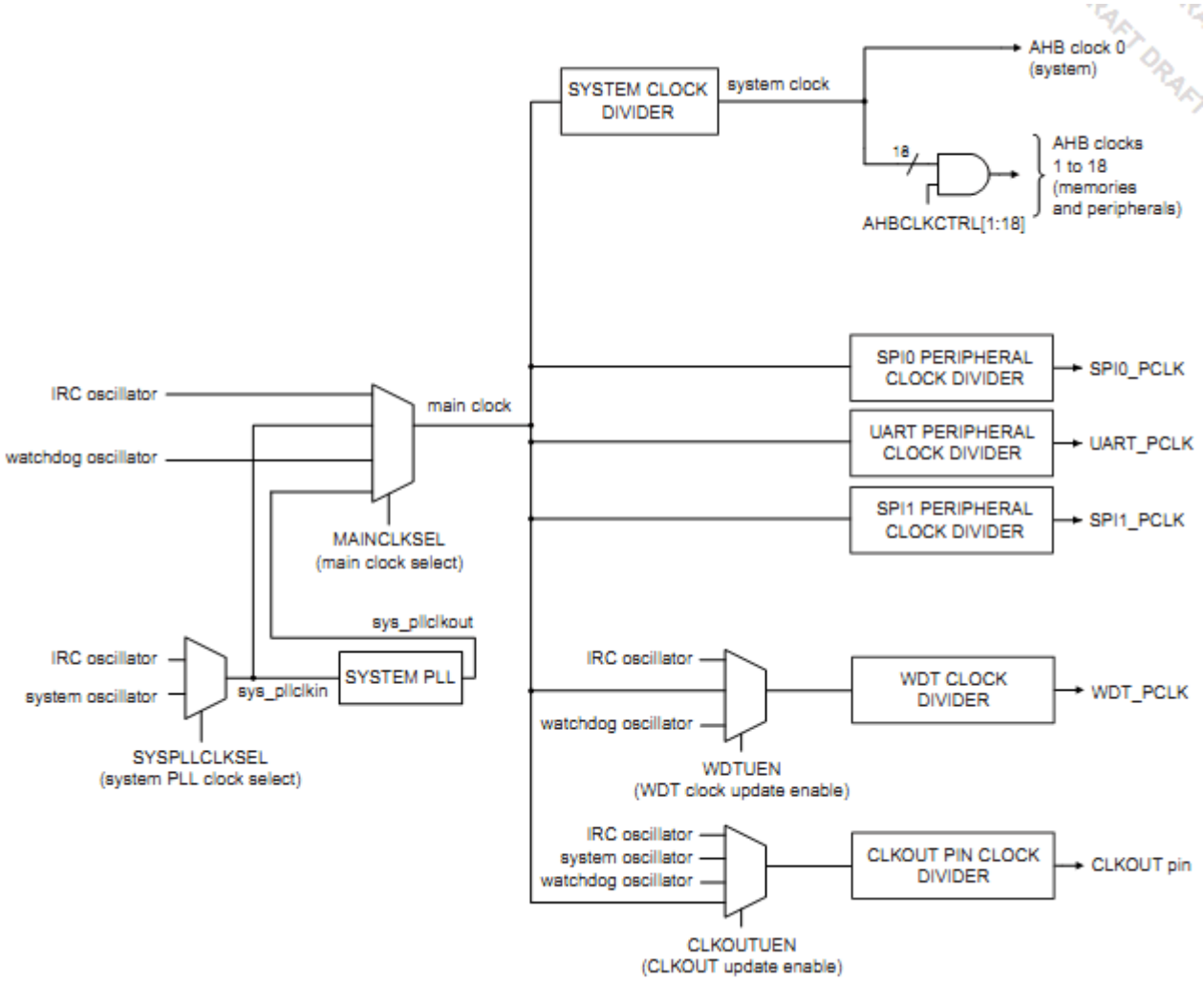
一、入门引导

亲爱的电工（“电子工程师”的简称）朋友，让我们开始吧！

ARM 微控制器有一个显著的特点，就是都可以把时钟频率倍频到很高，具体到多高，每个系列的微控制器都有一个指标，我们现在要学的 Cortex-M0 内核处理器 LPC1114 最高能到 50MHz，当然，其它的 ARM 内核微处理器可以倍频到更高，现在好多手机都采用了 ARM 内核处理器，比如卖的很火的诺基亚 5233 就是采用了 ARM11 处理器，ARM11 的处理器的主频为 433MHz，比 Cortex-M0 的 50MHz 高多了吧！所以 Cortex-M0 处理器被 ARM 称为入门级的内核！

二、时钟配置图(CGU)详解

要实现对系统时钟的配置，下面这个图是必须要看懂的！因为它比文字更具有参考价值，看上这张图配置时钟，绝对不会出现漏洞！（我建议你把这张图打印出来贴到你的墙上，我就是这么做的，你看着办吧！）接下来，我将一步一步引领你彻底看懂这张“时钟配置图”。



注意了，要开始讲图了！（这张图就是数据手册说的时钟产生单元：CGU（Clock generation unit））

LPC1114 内部含有 3 个时钟振荡器：**系统振荡器，IRC 振荡器，看门狗振荡器**。系统振荡器就是需要配合外部晶振工作的振荡器（这是任何一款单片机都有的）；IRC 振荡器就是内部 RC 振荡器，就是我在上面“总览 LPC1114”中提到的那个 LPC1114 一上电就默认选择的 12MHz 时钟振荡器，它的精度没有配合外部晶振的系统振荡器高；看门狗振荡器就是给看门狗提供的时钟振荡器！这么说大家明白了吧，在接下来的叙述里面，一提到系统振荡器就是指利用外部晶振的时钟振荡器，IRC 振荡器就是指 LPC1114 的内部时钟振荡器，可不要搞混了哦！

我们先从图的中心点看起，找到“主时钟”三个字，看“主时钟”的左面，有四条线到了“主时钟”的框上，这四条线就是“主时钟”的来源，它们分别是：IRC 振荡器，看门狗振荡器，倍频之前的时钟（sys_pllclkin）和倍频之后的时钟（sys_pllclkout）。也就是主时钟可以在这四个时钟源当中选择一个做为主时钟！通过操纵（人家专业名词不叫“操纵”，叫“访问”）“主时钟源选择寄存器（MAINCLKSEL）”实现。这个 32 位的主时钟源选择寄存器 MAINCLKSEL 只用到了两位（谁让两位就可以表示四种状态呢！），剩下的全都是保留位，如下：

位（bit）	符号	值	描述	复位值
1:0	SEL	00	选择 IRC 振荡器	00
		01	选择输入到 PLL 之前的时钟	
		10	选择看门狗振荡器	
		11	选择 PLL 之后的时钟	
31:2	-	-	保留	0

看复位值，系统默认情况下就是选择 IRC 振荡器作为系统的主时钟的。我们为了让 LPC1114 发挥出它最大的性能，就喜欢选择 PLL（PLL 就是倍频的意思）后的时钟，在程序中这样写：

```
SYSCON->MAINCLKSEL      = 0x00000003;          //主时钟源选择 PLL 后的时钟
```

接下来看图上，找到“系统 PLL”方框，看它左面倒梯形方框的左面，有两条线，这两条线就是可以做为倍频时钟源的时钟源。这两个时钟源分别是：IRC 振荡器和系统振荡器。该选择谁捏？这就要操纵“系统倍频时钟源选择寄存器（SYSPLLCLKSEL）”了。这个 32 位的寄存器也是只用到了两位：

（两位就可以表示四种状态了，三个状态当然是绰绰有余！）

位（bit）	符号	值	描述	复位值
1:0	SEL	00	选择 IRC 振荡器	00
		01	选择系统振荡器	
		10	保留	
		11	保留	
31:2	-	-	保留	0

看复位值，系统默认情况下就是选择 IRC 振荡器作为 PLL 输入时钟源的。既然我们外部安插了精确的 12M 晶振，就是想把它做为时钟源的，选择上面表

格当中的 01，就是选择了外部 12M 晶振！（我在先前提到过，“系统振荡器”就是代表外部的晶振，为了防止看的不仔细的朋友存在，我还是再说一遍吧！）

程序中这样写：

```
SYSCON->SYSPLLCLKSEL    = 0x00000001;          //PLL 时钟源选择“系统振荡器”
```

当然，操作顺序应该是先选择 PLL 的时钟源，再选择主时钟源！

到现在，“主时钟”左面的部分就看完了，接下来看“主时钟”右面的！

右面部分从上往下看，首先呢，是“系统时钟分频器”方框，方框的右面横线上写着“系统时钟”四个字。怎么样！迷惑了吧！这里方框中所提到的“系统时钟分频器”其实就是“系统 AHB 时钟分频器（SYSAHBCLKDIV）”。这个寄存器的名字会把好多人迷惑的！因为这个分频器可不仅仅给 AHB（LPC1114 的 AHB 只有 GPIO，关于什么是 AHB，什么是 APB，去百度搜一下吧！介绍需要两页纸哦！）提供时钟的，它除了给 AHB 提供时钟，还给内核，存储器以及 APB 提供时钟。一定意义上说，它就是“系统时钟分频器”了，给这个寄存器写 0，LPC1114 就不工作了；给这个寄存器写 1，LPC1114 的系统时钟就是主时钟除以 1；写 2，LPC1114 的系统时钟就是主时钟除以 2，以此类推！假如把外部晶振倍频了 4 倍作为主时钟，主时钟就是 48MHz，对 SYSAHBCLKDIV 写 4，系统时钟就是 12MHz。这时候有人就会有疑问了：“神经病啊！既然都倍频起来了，还要缩小”！其实这是因为有时候我们的电路板上的其它芯片不能够在很快的频率下工作，否则就会出错，比如无线通信芯片 NRF24L01 的速率就不能超过 10MHz，所以某些时候，需要多分频了。规定最多可以分频 255，所以你就可以想到，这个寄存器只用 8 位就可以了：

位（bit）	符号	值	描述	复位值
7:0	DIV	00000000	关闭系统时钟	00000001
		00000001	用 1 除	
		00000010	用 2 除	
		……	……	
		11111111	用 255 除	
31:8	-	-	保留	0

一般情况下，我们写 1，程序如下：（这条语句可以不用写，因为默认值就是 1）

```
SYSCON->SYSAHBCLKDIV    = 0x01;                //AHB 时钟分频值为 1
```

再往下看图，数一下，有 6 个分频器，这 6 个分频器是：SSP0 分频器，SSP1 分频器，UART 分频器，SysTick 分频器，看门狗分频器和 CLKOUT 引脚分频器。这些分频器寄存器和 SYSAHBCLKDIV 是一样的，都是用了 8 位，都是可以最多分频 255，我这里就不把表格画出来了，唯一不同的是，这 6 个分频器寄存器的复位值为 0，而不是 1。也就是说，在默认情况下，这些外设都是不工作的（没有时钟怎么工作！）这完全是为了节能做贡献，不用就不让它浪费电，用的时候再开！

看最后两个分频器！通过上面的介绍，你现在也可以看懂了，图上说：看门

狗的时钟源可以有 3 个来源，不仅仅只有“看门狗振荡器”可以给它提供，还可以用主时钟或是 IRC 振荡器！多么灵活的 LPC1114 呀!

LPC1114 上的第四引脚是：PIO0_1/CLKOUT/CT32B0MAT2。这个脚可以当做 P0.1 脚，CLKOUT 引脚和 32 位定时器的输出脚。CLKOUT 引脚，顾名思义，它是用来输出时钟的，输出时钟有什么用？

用处 1：给别的需要时钟的芯片提供时钟；

用处 2：用示波器观察此引脚上的频率可以判断你写的时钟配置程序是否正确。

这个引脚在默认的情况下是 P0.1 脚，假如你要看看到底有没有把外部的 12MHz 晶振倍频到 48MHz，你可以把这只脚配置为 CLKOUT 引脚，用示波器观察观察！

由图中可知，它可以选择 IRC 振荡器，系统振荡器，看门狗振荡器以及主时钟源作为时钟源，选择谁作为它的时钟源，你就可以看到谁的频率到底是多少了。（在下面会给出实现的程序，不要急哦！）我曾经用这个脚观察了一下 IRC 振荡器的频率，值在 12.01MHz 和 12.00MHz 之间来回跳！后来又看了一下外部晶振的频率，稳稳的显示 12.00MHz。

到现在，这张图就看完了，你也应该看懂了！

三、时钟配置程序设计

除了上面提到的“选择寄存器”，还需要有“使能寄存器”的配合才能使选择的时钟源起作用。下面是一个典型的时钟配置函数：

```
/*************************************************/
/*   函数功能：配置系统时钟                      */
/*   说明：     选择外部 12M 晶振作为系统时钟，并通*/
/*              过倍频器（PLL）把时钟倍频 4 倍     */
/*   注意：     使用其它值晶振时修改倍频值，最后 */
/*              时钟要满足<=50MHz                  */
/*************************************************/
void SysCLK_config(void)
{
    uint8 i;
    /*执行以下代码选择外部 12M 晶振作为时钟源*/
    SYSCON->PDRUNCFG        &= ~(1 << 5);          //系统振荡器上电
    SYSCON->SYSOSCCTRL      = 0x00000000;          //振荡器未被旁路，1~20Mhz 频率输入
    for (i = 0; i < 200; i++) __nop();             //等待振荡器稳定
    SYSCON->SYSPLLCLKSEL    = 0x00000001;          //PLL 时钟源选择“系统振荡器”
    SYSCON->SYSPLLCLKUEN    = 0x01;                //更新 PLL 选择时钟源
    SYSCON->SYSPLLCLKUEN    = 0x00;                //先写 0，再写 1 达到更新时钟源的目的
    SYSCON->SYSPLLCLKUEN    = 0x01;
    while (!(SYSCON->SYSPLLCLKUEN & 0x01));        //确定时钟源更新后向下执行
    /*执行以下代码倍频为 48MHz*/
```

```
    SYSCON->SYSPLLCTRL      = 0x00000023;       //设置 M=4;P=2; FCLKOUT=12*4=48Mhz
    SYSCON->PDRUNCFG        &= ~(1 << 7);         //PLL 上电
    while (!(SYSCON->SYSPLLSTAT & 0x01));       //确定 PLL 锁定以后向下执行
    /*主时钟源选择倍频以后的时钟*/
    SYSCON->MAINCLKSEL      = 0x00000003;           //主时钟源选择 PLL 后的时钟
    SYSCON->MAINCLKUEN      = 0x01;                 //更新主时钟源
    SYSCON->MAINCLKUEN      = 0x00;                 //先写 0，再写 1 达到更新时钟源的目的
    SYSCON->MAINCLKUEN      = 0x01;
    while (!(SYSCON->MAINCLKUEN & 0x01));       //确定主时钟锁定以后向下执行
    SYSCON->SYSAHBCLKDIV    = 0x01;         //AHB 时钟分频值为 1，使 AHB 时钟设置为 48Mhz
    SYSCON->SYSAHBCLKCTRL |= (1<<6);        //使能 GPIO 时钟
}
```

四、时钟配置程序详解

在看程序详解之前，你最好先看一遍程序。

（如果你是一位刚刚从 51 单片机接触 ARM 单片机的朋友，你会发现这个函数里面的语句书写方式完全和以前写 51 程序不一样啊，以前给 51 的寄存器写值，是用这么一种形式：

SBUF = 0X88;

而现在是用这么一种形式：

SYSCON->MAINCLKSEL=0X00000001;

这里为什么不直接写成：

MAINCLKSEL=0X00000001;

这其实是因为在 NXPLPC11XX.H 文件中对系统寄存器的定义采用了结构体（Struct）的形式。现在，你可以在打开 51 单片机寄存器的定义文件 REG51.H 文件看一下，它对寄存器的地址定义是这样的：

sfr SBUF = 0x99;

现在你再打开一下 LPC1114 对寄存器地址定义的 NXPLPC11XX.H 文件！全都是结构体的定义，而且是纯 C 语言写的，再也找不到“sfr”这样的 C51 语言了。关于 NXPLPC11XX.H 文件请看瑞嵌制作的《NXPLPC11XX.H 文件详解》。）

（在以后的程序中，我们会经常看到&=~（1<<3）和|=（1<<3）;这样的句子，这些句子是对位操作用的。因为我们经常要对 32 位寄存器的某一位操作，还同时不影响其它位的值，所以才有了上面这样的形式。比如我们说我们要对某个寄存器的 bit5(注意：可不是第 5 位，位是从 0 开始的)写 0，这样写：

寄存器&=~（1<<5）;

对寄存器的 bit5 写 1，这样写：

寄存器|=（1<<5）;

现在运用你的 C 语言知识分析一下，把十进制的 1 写成二进制 32 位数就是：

00000000000000000000000000000001

（1<<5）就是把 1 右移 5 下，左面补零，执行完这句话以后数就变成：

00000000000000000000000000100000

~（1<<5）就是再把这个数反相：

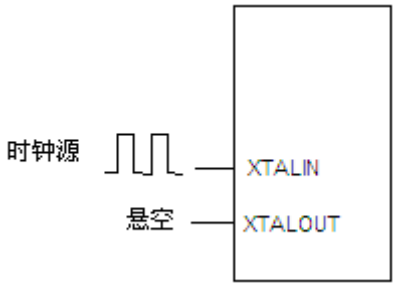
11111111111111111111111111011111

最后呢！再把这个数&给寄存器，&的操作即是遇到 0 与 1 等于 0，1 与 0 或 1 都还是 1，所以执行完以后，

除了 bit5 被改成了 0，其它的位都没有变。按照相同的方法，你可以分析一下对 bit5 写 1 的操作。）

现在，我们首先来看一下函数里的第一个语句是对 PDRUNCFG 寄存器操作，如果你的英语好的话，一眼就看出来这个寄存器是干嘛的了，就是“掉电配置寄存器”，之所以不叫“上电配置寄存器”是因为它是对某位写“1”掉电，写“0”上电。这个寄存器的描述请看官方数据手册第三章。看这个寄存器的 bit5，该位控制着系统振荡器的上电与掉电，默认是 1，就是掉电状态，我们既然已经决定了要用外部晶振作为时钟源，那么现在就该把它上电了，于是就有了这条语句。

接下来这条语句是对 SYSOSCCTRL 寄存器操作，这个寄存器叫做“系统振荡器控制寄存器”。（在后面的学习中，你会经常看到，系统内部的模块需要好几道门槛配置以后才能用，除了上电，还得控制，有的还需要再允许一下。这样做看似麻烦，其实灵活！）系统控制寄存器只用了 2 个 bit，bit0 控制着系统振荡器有没有被旁路，bit1 要根据外部晶振的值是多少来写 1 或 0。先说 bit0，“被旁路”的意思就是“让它不起作用”；“未被旁路”的意思就是“没有让它不起作用”。写 0 表示“未被旁路”，写 1 表示“被旁路”。那么什么时候被旁路呢？答：在有外部的“直接时钟源”的时候。如果你 51 单片机学的很棒的话，你现在就应该明白了，不明白的那就听我给你解释吧。其实 51 单片机也有不利用外部晶振而是利用“直接时钟源”的时候，电路图是这个样子的：



这里我们不需要旁路晶振，所以对该位写 0。bit1 是根据外部晶振的值来定的，对该位写 0 表示外部晶振频率值在 1~20MHz 范围内，写 1 表示外部晶振频率值在 15~50MHz 范围内。在我们的开发板上用的晶振为 12MHz，所以对该位写 0。

再往下是一条短暂延时程序，利用__nop();实现。给它一点时间完成任务。

再接下来的 5 条语句你可以把它看成一个整体，对 PLL 时钟源的更新都是这个样子的。关于 SYSPLLCLKSEL 寄存器，前面已经讲过了。SYSPLLCLKUEN 是 PLL 时钟源更新允许寄存器，根据官方数据手册上的规定，要想实现更新，需要对该寄存器 toggle 一下，也就是对该寄存器先写 0，再写 1。while 语句等待我们刚才写的 1 运输到 SYSPLLCLKUEN 里面。时钟源的更新往往是需要一定时间的。

接下来，就该把选择的时钟源翻倍了。SYSPLLCTRL 是系统倍频控制寄存器，通过它可以确定倍频的倍数。倍频器是一个很有特点的东西。它除了可以用在单片机当中，还可以用在好多需要它的地方，比如射频无线芯片当中可以用它来提高发射功率。倍频器运用了模拟电子技术和数字电子技术。有时集成到芯片当中，有时单独做成一块芯片！关于 LPC1114 的倍频器（PLL）的详细描述，请看官方数据手册第三章第九节。SYSPLLCTRL 的 bit0~bit4 确定 M 值，bit5 和 bit6 确定 P 值，bit7 是 DIRECT 位，bit8 是 BYPASS 位。其它位保留。bit7 和 bit8 我

们现在还不深究（要深究的话，还需要好好学习倍频器的结构），只需要知道它俩是来控制 PLL 的工作模式的，我们一般让 PLL 工作在“普通模式”下，保持这俩位的默认值就可以。那么现在只剩 M 和 P 了。在普通模式下，PLL 输出频率的计算公式如下所示：

$$Fclkout = M \times Fclkin = (FCCO)/(2 \times P)$$

看到上式，你可能会产生一个疑问：直接用 M 乘以 PLL 的输入频率 Fclkin 不行吗？答案当然不行！为什么要确定 P 值呢？这个是 PLL 的机构决定的，在普通模式下，输出频率实际上是由 FCCO 产生的，而为了能让 PLL 正常工作，FCCO 需要在 156~320MHz 之间。现在，我们知道 PLL 的输入频率 Fclkin 的值为 12MHz，LPC1114 的允许最大工作频率为 50MHz，现在我们只能把它倍频四倍到 48MHz 了，所以 M 值定位 4。根据数据手册上的规定，P 可以定为四个值，即 1，2，4，8。这里只有当 P=2 的时候，FCCO 的值为 48*2*2=192，在 156~320 之间。所以，我们一般情况下，就选 M=4，P=2 了。SYSPLLSTAT 是倍频状态寄存器，专门用来看 PLL 有没有锁定的，它是一个只读寄存器。

再往下的 5 条语句是更新主时钟用的。和上面提到的更新 PLL 时钟的语句如出一辙，我就不多讲了，相信大家现在已经能看懂了！

该函数的倒数第二条语句，就是给 SYSAHBCLKDIV 写 1，确定分频值为 1。这个寄存器在前面已经很详细的讲过了，这里就不啰嗦了！

最后一条语句涉及到的寄存器是 SYSAHBCLKCTRL。这个寄存器的名字叫做“系统外围时钟配置寄存器”。它管理着 LPC1114 几乎所有的独立模块的时钟开启与关闭。寄存器描述见下表：

bit19~31	bit18	bit17	bit16	bit15	bit14	bit13	bit12	bit11	bit10
保留	SSP1	保留	IOCON	WDT	保留	ADC	UART	SSP0	CT32B1
bit9	**bit8**	**bit7**	**bit6**	**bit5**	**bit4**	**bit3**	**bit2**	**bit1**	**bit0**
CT32B0	CT16B1	CT16B0	GPIO	I2C	FLASH2	FLASH1	RAM	ROM	SYS

由上表可知，bit6 是通用“输入输出口”时钟控制位，对对应的位写 0 关闭该位所控制的时钟，写 1 开启。

到此！这个函数就都讲完了。很好理解吧！这个函数就是每个工程里 main 函数里都会出现的初始化函数了。而且是必须的！为了使用方便，我已经把这个函数放到了 NXPLPC11XX.C 文件里面，在你写的 main 函数里直接调用函数名就可以了。

五、CLKOUT 引脚输出时钟程序设计

```
/*********************************************/
/*   函数功能：使能 CLKOUT 脚输出频率                */
/*   入口参数：CLKOUT_DIV,即 CLKOUT 分频值，1~255 */
/*   说明：       此函数可用来测试时钟真实性          */
/*********************************************/
void CLKOUT_EN(uint8 CLKOUT_DIV)
{
    SYSCON->SYSAHBCLKCTRL |= (1<<16);              // 使能 IOCON 时钟
```

```
    IOCON->PIO0_1=0XD1;                          //把 P0.1 脚设置为 CLKOUT 引脚
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);           // 禁能 IOCON 时钟
    SYSCON->CLKOUTDIV     = CLKOUT_DIV;          //CLKOUT 时钟值为（48/CLKOUT_DIV） MHz
    SYSCON->CLKOUTCLKSEL= 0X00000003;            //CLKOUT 时钟源选择为主时钟
    SYSCON->CLKOUTUEN     =0X01;
    SYSCON->CLKOUTUEN     =0X00;
    SYSCON->CLKOUTUEN     =0X01;
    while (!(SYSCON->CLKOUTUEN & 0x01));         //确定时钟源更新后向下执行
}
```

六、CLKOUT 引脚输出时钟程序详解

第一条语句，是把 P0.1 脚设置为 CLKOUT 引脚。（注意：PIO0_1 寄存器的 bit6 和 bit7 是保留位，这两个保留位默认为 1，而不是 0，我们不要随意修改保留位的值，所以写的值是 0XD1，而不是 0X01。）后面几条语句和 PLL 时钟源更新还有 MAIN 时钟源更新都是一样的形式，这里也就不啰嗦了，只要你看懂了上面的 SysCLK_config()函数的程序详解，这里肯定就没有问题了。还有一个需要注意的地方就是在对引脚的复用功能进行配置的话，也就是要使用 IOCON 寄存器的话，需要先在 SYSAHBCLKCTRL 寄存器里面把 IOCON 的时钟打开，否则配置将不起作用。当配置完以后，为了节能省电，就把 IOCON 的时钟关闭。当然，如果你不在乎功耗的话，不关也可以！

七、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。卸或不卸液晶显示器都可以。

此时在 CLKOUT 引脚上将会有 1MHz 的脉冲频率输出。CLKOUT 引脚即 P0.1 脚。在开发板的背面已经明确表明了每个引脚的名字，你可以很快的找这个标有 P0.1 的引脚。在 SWD 调试接口上，有一个标有 GND 的地方，你可以把示波器的探头地接到这个地方，然后把探头指针接到 P0.1 脚上，此时观察示波器，将会有 1MHz 的脉冲波形出现，由于这个频率的根源是晶振，所以波形会很标准。在我的示波器上，示波器上显示的频率为 1.00MHz，如果你的不是 1.00，而是在 1.00 附近的话，那说明你的示波器需要校准了。晶振的频率是非常标准的，这就是为什么会有 11.0592 这个值。

（注意：在测试频率的时候，要防止开发板线路短路，虽然开发板上的电源和地都接有保险丝）

SysCLK_config();函数一般不用修改，在你每写一个 LPC1114 项目之前，记得在 main 函数的一开始就调用这句话。执行完这句话后，工作主频将会是 48MHz。

CLKOUT_EN(div);函数用来在 CLKOUT 引脚上输出时钟，一般只是测试的时候用，或者是给其他芯片提供时钟的时候用。在试验程序中，取了 div 的值为 48，所以输出的时钟会是 48/48=1MHz。如果你的示波器可以测量 48MHz 的频率，把 div 写为 1 也可以。

第二章 按键输入与驱动 LED

学完这一章，你将会学到最基本的单片机输入与输出功能。学会了这些，就可以驱动液晶显示器了。

一、硬件连接

在 MINI LPC1114 V1.6 开发板上对应的按键连接方式是：

KEY1------P1.0
KEY2------P1.1
LED1------P1.9
LED2------P1.10

二、GPIO 控制寄存器

LPC1114 的引脚和普通的 51 单片机的引脚有一个区别，就是 LPC1114 的引脚到底是作为输出还是输入引脚是需要用寄存器配置的，也就是说，在同一时间，只能作为输入引脚或者是输出引脚。而普通的 51 单片机是不需要设置引脚是作为输入功能还是输出功能。

当 LPC1114 需要观察自己的某一根脚电平时，必须先把对应的脚设置为输入模式，才能读引脚电平。当用某一根引脚驱动外部器件时，必须设置为输出，才可以控制它的高低。这是所有高级单片机几乎都具备的功能。从这一点来看，确实是比普通 51 单片机复杂了，不过，复杂也会带来优点，那就是灵活。

在系统一上电，默认情况下引脚是作为输入引脚的。在控制 51 单片机的引脚高低时，是这样写的：

```
P3.2 = 1;
```

在控制 LPC1114 的引脚高低电平时，可不是给 PIO3_2 写 0 或 1，因为 PIO3_2 是 P3.2 脚的“功能配制寄存器”，LPC1114 有专用来控制引脚电平高低的寄存器是 DATA。当作为输入引脚时，读这个寄存器可以知道引脚上的电平，当作为输出引脚时，写这个寄存器可以控制引脚的电平高低。

例如：当需要 P3.2 脚作为输出引脚时，先写寄存器 PIO3_2，把 43 脚设置为 P3.2 脚：

```
IOCON->PIO3_2    &= ~0X07;
```

再把 P3.2 脚设置为输出引脚：

```
GPIO3->DIR |= 0X01;
```

让 P3.2 脚输出高电平：

```
GPIO3->DATA |=    (1<<2);
```

对比一下上面 LPC1114 让 P3.2 脚输出高电平和 51 单片机输出高电平，51 单片机只用了一条语句，LPC1114 用了 3 条语句。（看似复杂，其实灵活！）

上面出现了额 3 个寄存器：(下面的的 x 代表数字)

```
IOCON->PIOx_x
GPIOx->DIR
GPIOx->DATA
```

IOCON->PIOx_x 是“引脚配置寄存器”。LPC1114 的每根引脚都可以复用为好几个功能。这个寄存器就是用来选择引脚具体复用哪个功能的。默认情况下，引脚都默认为通用输出输入引脚。（例外：PIN3 默认情况下不是 P0.0，而是 RESET）

GPIOx->DIR 是当引脚作为通用输入输出口时，该引脚作为输入脚还是输出脚。（DIR 的英文全称就是“方向”的意思：direction。）对该寄存器写 0 作为输入引脚，写 1 作为输出引脚。默认情况下，该寄存器值为 0，作为输入引脚。

GPIO->DATA 是“数据寄存器”。可以控制引脚的电平高低，前面已经讲过。

三、LED 循环亮灭程序设计

学会了这三个寄存器，就可以控制 LED 的亮灭了。由开发板上的原理图可知，引脚上输出低电平，LED 亮，引脚上输出高电平，LED 灭。具体实现的程序如下：

```
int main(void)
{
    SysCLK_config();
    // 使能 GPIO 时钟
    SYSCON->SYSAHBCLKCTRL |= (1<<6);
    // 把 P1.9 和 P1.10 引脚设置为输出
    GPIO1->DIR |= (1<<9);
    GPIO1->DIR |= (1<<10);

    while(1)
    {
        //关 LED1   LED2
        GPIO1->DATA |= (1<<9);
        GPIO1->DATA |= (1<<10);
        delay_ms(1000);
        //开 LED1   LED2
        GPIO1->DATA &= ~(1<<9);
        GPIO1->DATA &= ~(1<<10);
        delay_ms(1000);
    }
}
```

四、LED 循环亮灭程序详解：

主函数第一条语句是引用了一个函数：时钟配置函数。这个函数在《第一章》已经做了详细的介绍，它的功能就是把外部 12M 晶振倍频 4 倍后作为系统时钟，使得 LPC1114 工作在 48MHz 主频下。

第二条语句涉及的寄存器：SYSAHBCLKCTRL。这个寄存器的名字叫做“系统外围时钟配置寄存器”。在上一章已经讲过了，它管理着 LPC1114 几乎所有的独立模块的时钟开启与关闭。寄存器描述见下表：

bit19~31	bit18	bit17	bit16	bit15	bit14	bit13	bit12	bit11	bit10
保留	SSP1	保留	IOCON	WDT	保留	ADC	UART	SSP0	CT32B1
bit9	**bit8**	**bit7**	**bit6**	**bit5**	**bit4**	**bit3**	**bit2**	**bit1**	**bit0**
CT32B0	CT16B1	CT16B0	GPIO	I2C	FLASH2	FLASH1	RAM	ROM	SYS

由上表可知，bit16 是“引脚配制时钟”控制位，对对应的位写 0 关闭该位所控制的时钟，写 1 开启。当我们需要配置引脚的复用功能时，需要先开启 IOCON 时钟，再复用引脚。例：

```
// 使能 IOCON 时钟(bit16)
SYSCON->SYSAHBCLKCTRL |= (1<<16);
// 把 LPC1114 的 PIN3 复用为 P0.0
IOCON->PIO0_0 &= ~0x07;    // 此语句的作用是先把低 3 位都改成 0
IOCON->PIO0_0 |= 0x02;     // 选择功能 PIO0_0
```

在本程序中，需要将引脚设置成输出，点亮 LED，所以需要先开启 GPIO 时钟，不过，LPC1114 系统默认 GPIO 时钟是开启的，不要这条语句也可以。因为 PIO1_10 和 PIO1_9 两个引脚默认为 GPIO 引脚，所以这里省略了配置 IOCON 的语句。如果这根引脚在之前复用了成了其它功能，就需要把省略的语句加上，如下：

```
// 把 LPC1114 的 PIN17 复用为 P1.9
IOCON->PIO1_9 &= ~0x07;
// 把 LPC1114 的 PIN30 复用为 P1.10
IOCON->PIO1_10 &= ~0x07;
```

第三条语句和第四条语句用到了寄存器 DIR，写 0 设置为输入，写 1 设置为输出。

再接下来就是 while 循环，它的功能就是让 LED1 和 LED2 亮一秒，灭一秒，持续执行。

DATA 寄存器前面也已经讲到了，写 1 拉高，写 0 拉低。

该程序就讲完了！

上面用到的延时函数可以在学完下一章 SysTick 后，用 SysTick 实现。

五、按键控制 LED 亮灭程序设计

在按键没有按下之前，对应引脚被外部上拉电阻拉高，按下按键以后，对应引脚被拉低。接下来，我们写一个程序，当按下 KEY1 键，点亮 LED1，放开 KEY1，LED1 熄灭；当按下 KEY2 键，点亮 LED2，放开 KEY2，LED2 熄灭。

程序如下：

```
int main(void)
{
    SysCLK_config();  // 时钟配置

    SYSCON->SYSAHBCLKCTRL |= (1<<16); //使能 IOCON 时钟(bit16)
    IOCON->PIO1_0 = 0XD1; //把 PIN33 设置为 P1.0 脚
```

```
    IOCON->PIO1_1 = 0XD1; //把 PIN34 设置为 P1.1 脚
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);//禁能 IOCON 时钟(bit16)（引脚配置完成后关闭该时钟节能）

    //把 P1.0 和 P1.1 设置为输入
    GPIO1->DIR &= ~(1<<0);
    GPIO1->DIR &= ~(1<<1);
    //把 P1.9 和 P1.10 引脚设置为输出
    GPIO1->DIR |= (1<<9);
    GPIO1->DIR |= (1<<10);
    //关 LED1   LED2
    GPIO1->DATA |= (1<<9);
    GPIO1->DATA |= (1<<10);



    while(1)
    {
        if((GPIO1->DATA&(1<<0))!=(1<<0))          // 如果是 KEY1 被按下
        {
            GPIO1->DATA &= ~(1<<9);                    // 开 LED1
            while((GPIO1->DATA&(1<<0))!=(1<<0));   // 等待按键释放
            GPIO1->DATA |= (1<<9);                     // 关 LED1
        }
        else if((GPIO1->DATA&(1<<1))!=(1<<1))    // 如果是 KEY2 被按下
        {
            GPIO1->DATA &= ~(1<<10);                   // 开 LED2
            while((GPIO1->DATA&(1<<1))!=(1<<1));   // 等待按键释放
            GPIO1->DATA |= (1<<10);                    // 关 LED2
        }
    }
}
```

六、按键控制 LED 亮灭程序详解：

程序的好多语句前面已经讲过了。看第 3 和第 4 条语句，功能是把 PIN33 和 PIN34 设置为 P1.0 和 P1.1 脚，看数据手册 PIO1_0 寄存器和 PIO1_1 寄存器的复位值可以知道，这两个引脚在默认情况下，并不是 GPIO 引脚，所以需要设置一下。

while 循环是在不断检测按键的状态。

while 循环的第 1 条语句是读取 P1 口的值，读取该寄存器，将读取到 P1 口被设置为输入引脚的引脚上电平值，其它作为输出的值对应的位值无效。P1 口上只有 P1.0 和 P1.1 脚被设置成了输入引脚，所以读到的值只有 bit0 和 bit1 有效，

其它位无效。我们也没必要看其它位的值，只需要看 bit0 和 bit1 的值就可以了。第一个 if 条件当中的语句是在判断 bit0 的值，第 2 个 if 条件当中的语句是在判断 bit1 的值。当 bit0 的值为 1，表示 KEY1 没有被按下，当 bit0 的值为 0，表示 KEY1 被按下。当 bit1 的值为 1，表示 KEY2 没有被按下，当 bit1 的值为 0，表示 KEY2 被按下。写这两条 if 语句的时候，极容易犯的一个错误，就是运算符优先级引起的错误。所以为了犯这种低级错误，尽量把能加括号的地方括起来。

七、用中断方式实现按键控制 LED 亮灭程序设计

```
/*********************************************/
/*  函数名称：GPIO1 中断服务函数                */
/*********************************************/
void PIOINT1_IRQHandler(void)
{
    if((GPIO1->MIS&0x001)==0x001)        // 检测是不是 P1.0 引脚产生的中断
    {
        GPIO1->DATA &= ~(1<<9);                  // 开 LED1
        while((GPIO1->DATA&(1<<0))!=(1<<0));  // 等待按键释放
        GPIO1->DATA |= (1<<9);                   // 关 LED1
    }
    else if((GPIO1->MIS&0x002)==0x002) // 检测是不是 P1.1 引脚产生的中断
    {
        GPIO1->DATA &= ~(1<<10);             // 开 LED2
        while((GPIO1->DATA&(1<<1))!=(1<<1));  // 等待按键释放
        GPIO1->DATA |= (1<<10);                  // 关 LED2
    }
    GPIO1->IC = 0x3FF;    // 清除 GPIO1 上的中断
}

/*  主函数  */
int main(void)
{

    SysCLK_config(); // 时钟配置

    SYSCON->SYSAHBCLKCTRL |= (1<<16); // 使能 IOCON 时钟(bit16)
    IOCON->PIO1_0 = 0XD1; //把 PIN33 设置为 P1.0 脚
    IOCON->PIO1_1 = 0XD1; //把 PIN34 设置为 P1.1 脚
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16); // 禁能 IOCON 时钟(bit16)

    //把 P1.0 和 P1.1 设置为输入
    GPIO1->DIR &= ~(1<<0);
    GPIO1->DIR &= ~(1<<1);
    //把 P1.9 和 P1.10 引脚设置为输出
```

```
    GPIO1->DIR |= (1<<9);
    GPIO1->DIR |= (1<<10);
    //关 LED1    LED2
    GPIO1->DATA |= (1<<9);
    GPIO1->DATA |= (1<<10);

    GPIO1->IS &= ~(1<<0); //选择 P1.0 为边沿触发
    GPIO1->IEV &= ~(1<<0);//选择 P1.0 为下降沿触发
    GPIO1->IE |= (1<<0);    //设置 P1.0 中断不被屏蔽

    GPIO1->IS &= ~(1<<1); //选择 P1.1 为边沿触发
    GPIO1->IEV &= ~(1<<1);//选择 P1.1 为下降沿触发
    GPIO1->IE |= (1<<1);    //设置 P1.1 中断不被屏蔽
    NVIC_EnableIRQ(EINT1_IRQn);//  使能 GPIO1 中断

    while(1)
    {
        ;
    }
}
```

八、用中断方式实现按键控制 LED 亮灭程序详解

上面有两个函数，第一个是 GPIO1 中断服务函数。第二个是主函数。主函数的配置和前面讲到查询方式主函数几乎没有区别。只是多了几条对 GPIO 中断控制寄存器写值的语句，还多了一句开启中断的语句：

```
NVIC_EnableIRQ(EINT1_IRQn);//  使能 GPIO1 中断
```

这条语句是调用了一个函数，这个函数是开启中断函数。它的入口参数决定了开启什么中断。要正确写入对应的入口参数，需要进入 NXPLPC11XX.H 文件找到中断号定义结构体，找到对应的名称。这里，我们需要处理 P1.0 和 P1.1 引脚上的中断。P1.0 和 P1.1 是 LPC1114 的 GPIO1 口，所以需要开启 GPIO1 的中断，查中断号后，得到这个值为 EINT1_IRQn。执行完这个函数以后，所有发生在 GPIO1 上的中断都会被检测到。

在程序中，出现了 3 个新的寄存器。

GPIO1->IE\GPIO1->IEV\GPIO1->IS.

这三个寄存器的名称分别是：

GPIOx->IS　　中断感应寄存器

GPIOx->IE　　中断屏蔽寄存器

GPIOx->IEV　中断事件寄存器

看这名称，云里雾里的，还是讲的简单些吧！

GPIOx->IS　设置边沿触发还是电平触发中断的寄存器
GPIOx->IEV 设置低电平触发还是高电平触发的寄存器或者是设置下降沿触发还是上升沿触发或者是双边沿触发的寄存器
GPIOx->IE　负责开启和屏蔽对应引脚中断的寄存器

具体 bit 位对应的控制参数，可以看数据手册。

再看中断服务函数，又出现了两个寄存器，GPIOx->MIS 和 GPIOx->IC
这两个寄存器的名称分别为：
GPIOx->MIS　　中断状态寄存器（可以用来观察是哪个引脚上引发的中断）
GPIOx->IC　中断清除寄存器（用来在中断发生以后，把对应的位清除）
　　注意：用来清除边沿触发引起的中断

上面讲到的寄存器一般都是前 12 位控制着对应 GPIO 的对应引脚。

九、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。卸或不卸液晶显示器都可以。

“查询方式”和“中断方式”的效果一模一样，都是按下 KEY1 键后，LED1 亮，按下 KEY2 键以后，LED2 亮。按键一直按着，灯一直亮着，放开按键后，灯灭。

在不使用仿真器的情况下，按键和 LED 显得非常有用，它可以被用来调试程序。我后面写的好多程序，都是用 LED 亮灭指示调试成功的，帮了我很大忙。

第三章 系统定时器 SysTick

一、系统定时器 SysTick 简介
二、程序设计
三、程序详解
四、实验程序下载和使用说明

一、系统定时器 SysTick 简介

延时函数在众多函数都应用十分广泛。我们最简单的延时函数就是使用 for 循环或者是 while 循环函数，但是这样的函数有一个缺点就是，在不同单片机和单片机工作在不同频率下，这样的函数很难精确定时。采用定时器定时可以精确，但当它作为延时函数用的时候就很难再做它用了，对于资源有限的单片机来说，我们总是不希望浪费定时器。在 ARM 内核的处理器当中，基本上都配有一个系统定时器（SysTick），这个定时器不同于一般的定时器，这个定时器主要是为操作系统服务的，一般“操作系统”的系统定时器都是 10ms 中断一次。LPC1114 也带有一个 SysTick，该定时器其实是 Cortex-M0 内核自带的，所以，只要是 Cortex-M0 内核的单片机，都有此 Systick。该系统定时器还有一个与其它定时器不同的就是它是 24 位的倒计时定时器。（它是 Cortex-M0 内核的组成部分，并不是 LPC1114 的外围）。如果我们的 MCU 不上操作系统的话，不用它显得有些浪费，把它作为延时函数用的定时器，是个明智的选择！下面介绍两个用 SysTick 做的通用函数：

```
void delay_ms(uint32 ms); //毫秒延时函数
void delay_us(uint32 us); //微秒延时函数
```

二、程序设计

下面这两个函数是用系统定时器 SysTick 实现的，程序如下所示：

```
/*********************************************/
/*   函数功能：SysTick 延时                  */
/*   入口参数：毫秒值或微秒值                */
/*   说明：    利用系统定时器实现            */
/*             48Mhz 时钟工作下              */
/*********************************************/
static volatile uint32 TimeTick = 0;

void SysTick_Handler(void)                          //系统定时器中断服务函数
{
    TimeTick++;
}

void delay_ms(uint32 ms)
{
    SYSTICK->STRELOAD =  (((24000)*ms)-1);    //往重载计数器里写值
    SYSTICK->STCURR    =   0;                       //计数器清零
    SYSTICK->STCTRL   |=   ((1<<1)|(1<<0));      //开启计数器,开启计数器中断
    while(!TimeTick);
    TimeTick = 0;
    SYSTICK->STCTRL =0;                           //关闭定时器
}

void delay_us(uint32 us)
```

```
{
    SYSTICK->STRELOAD = (((24)*us)-1);    //往重载计数器里写值
    SYSTICK->STCURR    =   0;                    //计数器清零
    SYSTICK->STCTRL   |=   ((1<<1)|(1<<0));      //开启计数器,开启计数器中断
    while(!TimeTick);
    TimeTick = 0;
    SYSTICK->STCTRL =0;                          //关闭定时器
}
```

三、程序详解：

程序一开始，先定义了一个静态全局变量 TimeTick。

此变量的作用是：用来观察计时有没有到。0 没有到，1 到。

此变量具体的用法是：先在函数里给变量赋值 0，在中断函数里面把它置为 1。观察此变量的值即可知道定时时间有没有到！

接下来是一个系统定时器中断服务函数。（中断服务函数的函数名是系统固定的，不可以更改，里面的实现语句是需要用户写进去的。想要知道中断服务函数名怎么写或者是想更改成你想要的函数名称，可以打开 startup_LPC11xx.s 文件，在向量表（__Vectors）里面可以看到函数命名。）

看到该函数里面就有一条语句，就是把 TimeTick 置为 1。

再往下的两个函数就是微秒和毫秒延时函数了。

这两个函数只有一个不同的地方：毫秒延时函数里面第一条语句的值 24000 在微秒延时函数里面是 24。其它语句都一样。

在函数当中用到了 3 个寄存器：

SYSTICK->STRELOAD 系统定时器重载值寄存器

SYSTICK->STCURR 系统定时器当前值寄存器

SYSTICK->STCTRL 系统定时器控制和状态寄存器

（其实系统定时器一共才有四个寄存器，除了上面的三个寄存器外，还有一个是 SYSTICK->STCALIB(系统定时器校准值寄存器)，这个寄存器我们不需要更改。）

“重载值寄存器”听起来应该不生疏吧，它和 51 单片机里面的重载值寄存器功能是一样一样的！不过，用户可以给这个寄存器写 24 位的值。

读“当前值寄存器”可以得到定时器在工作时的当前值，写当前值寄存器可以清零，不管写什么值都可以，一般为了程序的可读性，清零该寄存器就写 0。

“控制和状态寄存器”的 bit0 控制系统定时器的开启和关闭（相当于 51 单片机的 TR0 和 TR1）。bit1 控制系统定时器中断的开启和关闭。bit16 是系统定时器的状态位，当 STCURR 为 0 时，该位置 1。

注意：以上两个延时函数中，毫秒延时函数最大值 699，微秒延时函数最大值 699*1000，在时钟频率 48MHz 下工作取值 24，在 24MHz 下工作取值 12，以此类推！

四、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。卸或不卸液晶显示器都可以。

效果是 LED1 和 LED2 交替亮灭。间隔时间为 1 秒钟。由于本质上 SysTick 的时钟根源为晶振，所以定时时间非常精确。

千万注意：

①入口参数的范围，ms 值为 1~699，us 值为 1~699000；用的时候不要超过这个范围。

问：为什么是这么多？

答：因为这个定时器是 24 位的定时器，取值不能超过 24 位值。

问：最大定时 699ms，想定时 1 秒怎么办？

答：用两次，如下所示：

```
delay_ms(500);
delay_ms(500);
```

②不要在中断里面用此函数。

问：为什么

答：因为此函数是用中断实现的。嵌套中断比较麻烦，不进行有效设置，用了也不起作用。

第四章 驱动液晶显示器

一、入门引导
二、硬件连接
三、程序设计
四、程序解释
五、总结
六、实验程序下载和使用说明

一、入门引导

在现在的市场上，小尺寸的液晶显示器应用越来越来广泛了，在很多 MP3，MP4，掌上游戏机，手机领域当中应用十分广泛。在以前，TFT 对于电子工程师来说还是一个很专业的东西，随着单片机技术在近几年来突飞猛进的发展，TFT 对于我们来说已经是一个非常简单的东西了，难易程度只相当于控制 1602 显示器。

在这一章里，我将用最简单的表达方式让你学会简单的 TFT。

开发板上带的 TFT 为 2.4 寸液晶屏，分辨率是 240*320。这种屏是目前市场上用的最多的屏，当然也是最具代表性的屏。它的显示是点阵显示原理，和电脑上的液晶显示器显示原理是一样一样的。分辨率 240*320 的意思就是说屏幕上横着有 240 个点，竖着有 320 个点，只是比你原来 51 单片机开发板上带的 16*16 点阵多了很多点而已。

它一共有 240*320=76800 个点。每个点都是由三原色“红绿蓝 RGB”组成的，每个点都可以配置成不同的颜色（一共有 65536 种颜色，专业术语叫做 65K 色，也就是 16 位的颜色，学过 51 单片机的都知道，16 位的定时器可以从 0 计数到 65535，那 16 位的颜色当然也就可以表示 65536 种颜色了，这么说更明白些吧！）。假如要让屏幕显示白色，就需要把每个点配置成白色，白色的 16 位码是 0XFFFF，我们经常听到的刷屏的速度就是说，显示一个整屏所需要的时间，在这里我们可以说成点亮 76800 个点所需的时间。当然是越快越好。我们要想让 TFT 显示我们所需的内容，其实就是一个点一个点的显示不同颜色形成的效果。你仔细看一下你的电脑屏幕（要液晶的哦！）或者是你的手机屏幕再或者是看一下我们开发板上的 TFT，你会发现它就是一个点一个点的形成图像的。我们可以把 TFT 上面的任意一个点设置成任意一种颜色。TFT 模块里面有一个液晶控制器，名字叫做 ILI9325，专门为没有液晶控制器的单片机设计。我们就是操作 ILI9325 的寄存器，来实现 TFT 的显示的。所以很有必要学一下 ILI9325 了。

ILI9325 只是当今液晶控制器家族里面的一个成员，不过，不管是哪种控制器，原理和引脚几乎都是一样的。学会了一种，其它的也就会了。

ILI9325 有四种总线接口控制 TFT：

①i80 系统 MPU 接口

②VSYNC 接口

③SPI 接口

④RGB 接口

其中“i80 系统 MPU 接口”和“SPI 接口”是单片机操纵 TFT 最常用的两种总线接口方式。“SPI 接口”是串行传输方式，“i80 系统 MPU 接口”是并行方式，并行传输和串行传输各自的优缺点，大家都知道了，我就不用在这里说了吧，所以我们一般情况下选择“i80 系统 MPU 接口”。

“i80 系统 MPU 接口”的数据总线宽度可以是 8 位，9 位，16 位和 18 位。51 单片机的 IO 口都是 8 位的，所以一般用 8 位的接法。STM32 的 IO 口是 16 位的，所以一般用 16 位的接法。LPC1114 的 IO 口是 12 位的，权衡一下，还是用 8 位数据宽度显示速度最快。

二、硬件连接

开发板上的2.4寸液晶显示屏与LPC1114的引脚连接采用8位数据总线方式：

DB0~DB7： 空脚。

DB8~DB16：接与 P2.4~P2.11 连接（8 位数据总线连接方式用高 8 位连）。

LCD_CS： 液晶显示器片选引脚，低电平有效，与 P3.1 引脚相连。

LCD_RS： “写数据”“写命令”选择引脚。高：写数据；低：写命令。与 P3.0 相连。

LCD_WR： 写信号输入引脚，低电平有效。与 P3.2 相连。

LCD_RD： 读信号输入引脚，低电平有效。与 P3.3 相连。

LCD_RST： 液晶显示器复位引脚，与 LPC1114 的复位脚相连，所以 LPC1114 和 TFT 一定是同时复位的。

三、程序设计

```
extern void LCD_Init(void);      // 初始化液晶显示器
extern void LCD_DisplayOn(void); // 开显存
extern void LCD_DisplayOff(void);   // 关显存
extern void LCD_Clear(uint16 color);   // 整屏显示
extern void LCD_DrawPoint(uint16 x,uint16 y); // 在屏上画一个像素的点
extern uint16 LCD_ReadPoint(uint16 x,uint16 y);     // 读屏上一个像素的点的颜色值
extern void LCD_DrawLine(uint16 x1, uint16 y1, uint16 x2, uint16 y2); // 画一条直线
extern void LCD_DrawRectage(uint16 xstart,uint16 ystart,uint16 xend,uint16 yend,uint16 color);  // 画矩形
extern void LCD_Fill(uint16 xstart ,uint16 ystart ,uint16 xend ,uint16 yend ,uint16 color);    // 画带填充的矩形
extern void LCD_DrawCircle(uint8 x0, uint16 y0, uint8 r); // 画一个圆
extern void LCD_ShowNum(uint8 x,uint16 y,uint32 num,uint8 len);  // 显示数字
extern void LCD_ShowChar(uint16 x,uint16 y,uint16 num);     // 显示一个英文字符
extern void LCD_Show_hz(uint16 x,uint16 y,uint8 *hz); // 显示一个 GBK 汉字
extern void LCD_ShowString(uint16 x,uint16 y,uint8 *p);// 显示字符串（英文和中文都可以）
```

以上是在 ILI9325.H 文件中定义的函数。

四、程序解释

由于程序太多，而且为了让朋友很快的学会 LPC1114，下面只对函数做简单的解释。达到会用的程度即可。

开发板上的 2.4 寸液晶显示器内置液晶显示控制芯片：ILI9325。对液晶显示屏的显示控制，实际上就是对 ILI9325 寄存器的操作。

函数①：LCD_Init(void);

在程序中找到液晶显示器的初始化函数 LCD_Init(void);这个函数的功能其实就是让液晶显示器开始工作。

这个函数用到了子函数：

```
void LCD_WR_REG_DATA(uint16 REG, uint16 VALUE);
```

该子函数的功能是往 ILI9325 的寄存器里面写数据，入口参数有两个，第一个是 ILI9325 的寄存器地址，第二个是要给寄存器写的值。

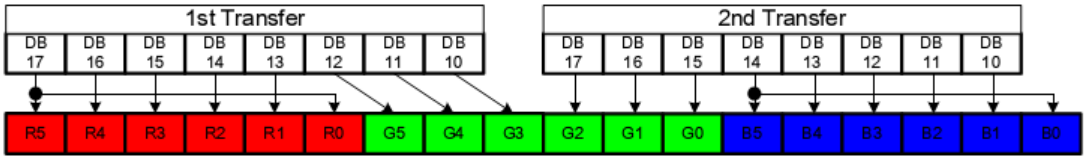
在上面的初始化函数里，一共用了 49 次 LCD_WR_REG_DATA 函数，也就是说，初始化液晶显示器需要往 ILI9325 的 49 个寄存器里面写对应的数据。这 49 个寄存器这里就不做详细介绍了。初始化函数是对液晶显示器的接口选择以及显示方向的控制。执行完这个函数，你就可以在液晶显示器的任意一个点上显示你想显示的颜色了，这些点组合起来，就可以显示线条，字体，图形了。

函数②： void LCD_Clear(uint16 color)

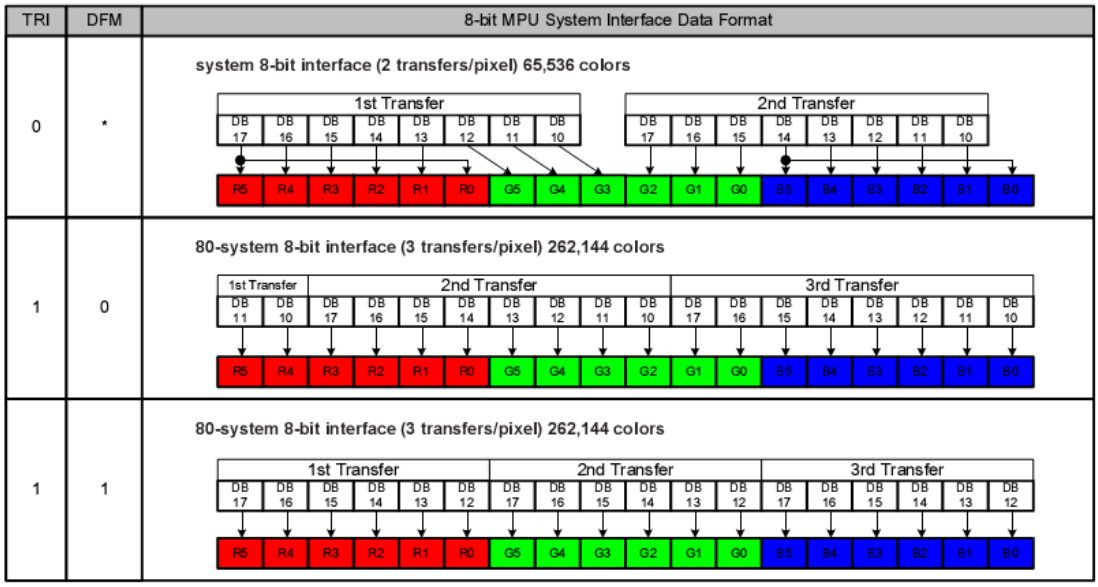
在液晶显示的过程中，我们用到的最多的一个函数就是“清屏函数”了，这个函数可以让一整屏显示一种颜色。程序如下：

```
void LCD_Clear(uint16 color)
{
    uint32 temp;
    LCD_WR_REG_DATA(0x0020,0);//设置 X 坐标位置
    LCD_WR_REG_DATA(0x0021,0);//设置 Y 坐标位置
    LCD_WR_REG(0x0022);          //指向 ILI9325 显存，准备写数据到显存
    for(temp=0;temp<76800;temp++)
    {
        LCD_WR_DATA(color);
    }
}
```

此程序有一个入口参数，就是想要在屏幕上显示的颜色。该颜色值为 16 位值，比如白色值为 0xFFFF，黑色为 0x0000，红色为 0xF800。这个值是怎么得到的？看下图：



上图是这么个意思：红、绿、蓝都由 6 位值组成，红色值（R5~R0）绿色值（G5~G0）蓝色值（B5~B0）。三六十八，要表示一种颜色需要 18 位值，但我们的 8 位总线传输方式是先写高 8 位，再写低 8 位，一共才 16 位，所以 ILI9325 定义了上图这种定义方式。由上图可知，R0 和 R5 值相同，B0 和 B5 值相同，所以就可以用 16 位表示 18 位值了。前面提到了我们开发板上的任意一个点都可以显示 65536（65K）种颜色，其实，ILI9325 最大可以显示 262K 色，也就是 18 位值色，一般情况下，65K 色足可以满足我们的要求，如果想要显示 262K 色，就需要把设置 ILI9325 的寄存器位 TRI 和 DFM，并且每次传输颜色数据就要 3 次。所以提高显示速度，我们还是选择了每次传输 2 次颜色数据的 65K 色方式。完整的颜色值显示方式图如下所示：



（8 位总线接口颜色数据传输图）

图解：

第一行：TRI=0，DFM=任意值。65K 色方式，两次数据传输。先写高 8 位，再写低 8 位。

第二行：TRI=1，DFM=0，262K 色方式，三次数据传输。第一次写的数据取低两位作为 R5 和 R4 值，第二次写进去的数据和第三次写进去的数据分别对应其它值，见图。

第三行：TRI=1，DFM=1，262K 色方式，三次数据传输。第一次写的数据取高 6 位作为红色值；第二次写的数据取高 6 位作为绿色值；第三次写的数据取高六位作为蓝色值。

“TRI 位”和“DFM 位”分别位于 ILI9325“Entry Mode 寄存器”的 bit15 和 bit14，该寄存器的地址为 0x0003，再回去看一下液晶显示器的初始化函数中对该寄存器的操作，写的数据是 0x1030，由这个数据得到，bit15=0，bit14=0，所以我们选择了 65K 色方式。

看此程序中的第二、三条语句，是对寄存器地址 0x0020 和 0x0021 分别写 0，从 ILI9325 的数据寄存器手册上可以查到，0x0020 和 0x0021 分别是 ILI9325 显存指向的 X 和 Y 坐标，写这两个值可以定义屏幕上的任意一个位置。此程序中 X 和 Y 值都写了 0，说明要从坐标（0，0）开始写起，ILI9325 的数据手册上说，显存地址可以自动加 1，所以，当我们给（0，0）这个点写完颜色以后，下一个写的颜色数据就会写到（1，0）这个点，等自动涨到（239，0）这个点后，就会自动到（0，1）这个点，以此类推！下面的 for 循环，最大值为 76800，就是用 240 乘以 320 得出的。

程序中出现的两个新函数是：

```
void LCD_WR_DATA(uint16 val);
void LCD_WR_REG(uint16 reg);
```

先看一下 LCD_WR_REG_DATA 函数的定义：

```
void LCD_WR_REG_DATA(uint16 REG, uint16 VALUE)
```

```
{
    LCD_WR_REG(REG);
    LCD_WR_DATA(VALUE);
}
```

LCD_WR_REG_DATA 函数实际上就是先调用 LCD_WR_REG 函数确定要写的寄存器地址，然后在把调用 LCD_WR_DATA 函数把数据写进去。

在清屏函数当中，这两个函数是分开调用的，先调用写寄存器地址函数 LCD_WR_REG(0x0022)，（0x0022 是 ILI9325 的显存寄存器地址）然后再用一个 for 循环给显存寄存器里面不断地写颜色数据进去。

清屏函数讲完了。

函数③： void LCD_DrawPoint(uint16 x,uint16 y)

接下来，讲画点函数。此函数可以在液晶显示屏上的 240*320 矩阵当中的任意一个点上显示定义的颜色。画点函数如下：

```
void LCD_DrawPoint(uint16 x,uint16 y)
{
    LCD_WR_REG_DATA(0x0020,x);//设置 X 坐标位置
    LCD_WR_REG_DATA(0x0021,y);//设置 Y 坐标位置
    LCD_WR_REG(0x0022);          //开始写入显存 GRAM
    LCD_WR_DATA(POINT_COLOR);
}
```

仔细看一下此函数，实际上就是清屏函数的 76800 分之 1。因为此函数是画一个点，而清屏函数是画 76800 个点！

函数④： uint16 LCD_ReadPoint(uint16 x,uint16 y);//读屏上一个像素的点的颜色值

此函数可以读出屏幕上任意一点的颜色值，此函数主要用在 GUI 方面。

举个例子：

你在屏幕上设计了这么个功能：单击屏幕某图标，该图标的边框变蓝，表示已经选择该边框。此时便可以读某一点的颜色来判断现在选择了哪个图标。

函数⑤： LCD_DrawLine(uint16 x1, uint16 y1, uint16 x2, uint16 y2); // 画一条直线

该函数可以在屏幕上画一条直线，横线，竖线或者是斜线都可以实现，该直线的颜色有 POINT_COLOR 决定。

函数⑥： LCD_DrawRectage(uint16 xstart,uint16 ystart,uint16 xend,uint16 yend,uint16 color);

该函数可以在屏幕上画一个矩形。

函数⑦： LCD_Fill(uint16 xstart ,uint16 ystart ,uint16 xend ,uint16 yend ,uint16 color);

该函数的功能是画一个填充矩形

函数⑧： LCD_DrawCircle(uint8 x0, uint16 y0, uint8 r); // 画一个圆

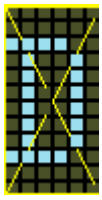
该函数的功能时画一圆，可以定义原点和半径。

函数⑨： LCD_ShowNum(uint8 x,uint16 y,uint32 num,uint8 len); // 显示数字

该函数的功能是显示数字，入口参数包括横纵坐标，要显示的数字，和数字的长度。比如要显示 12345 在屏幕上，len 可以最小值取 5，因为一共有 5 个数字，如果改值取了 6 或者是 7，也只会显示 5 个数字，前面的 0 会在程序中自动去掉。

函数十： LCD_ShowChar(uint16 x,uint16 y,uint16 num);

写字符函数。假如我们要在一个 6*12 点阵里面显示一个字符“D”，就必须要知道它的字模，字模如下：



上图的小方格就是液晶显示器点阵放大的效果，显示一个字符的原理：依次写 6*12 点阵数据，在“D”的显示线上显示与背景不同的颜色。

该函数可以显示英文大小写字母和各类常用标点符号。

函数十一： LCD_Show_hz(uint16 x,uint16 y,uint8 *hz);

该函数可以显示一个汉字。需要配合 W25X16 芯片使用。关于 W25X16 后面的章节会讲到。

函数十二： LCD_ShowString(uint16 x,uint16 y,uint8 *p);

该函数用来显示大于一个的英文或中文，自动换行，在函数内部，可以设置行间距和边间距，也可设置字间距。

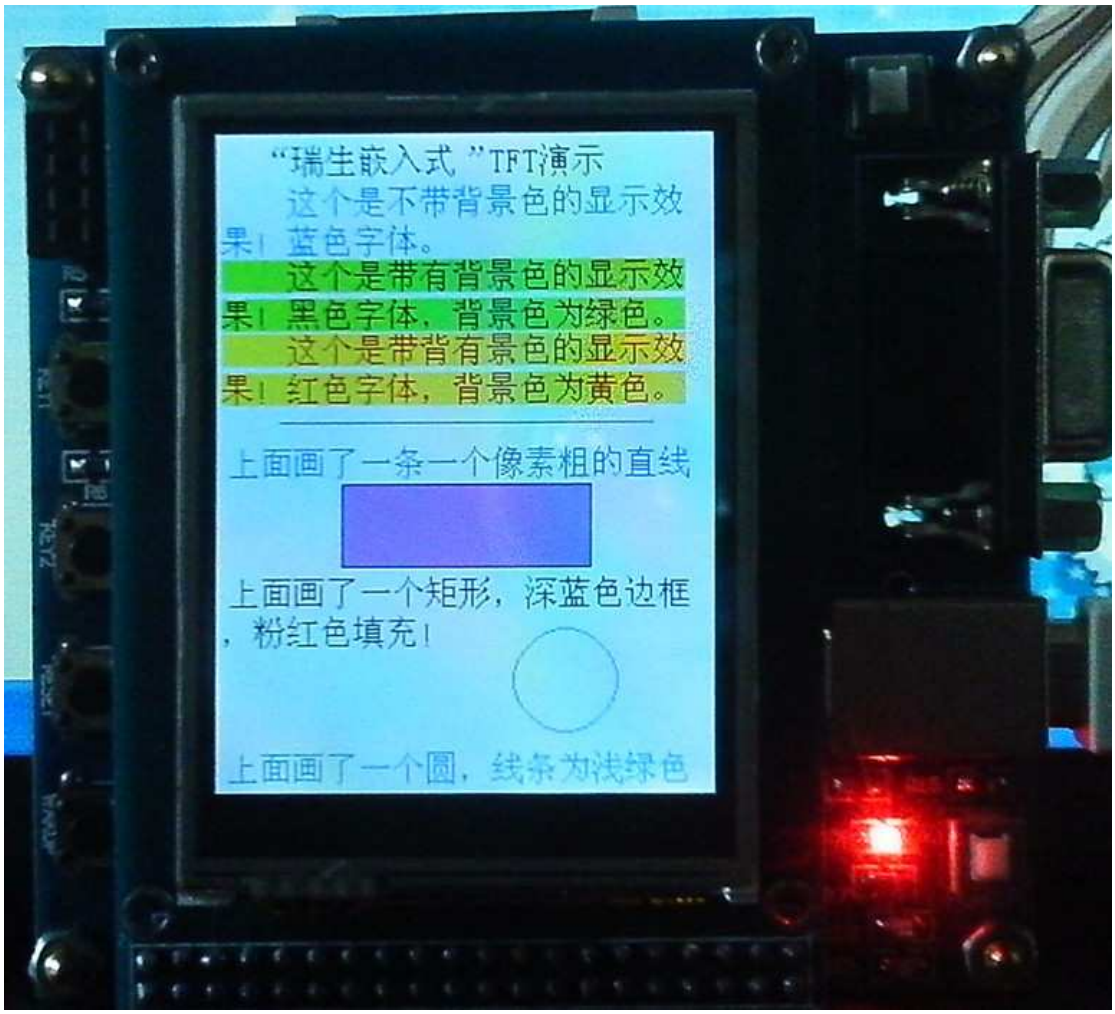
五、总结

有了上面这些函数，基本上控制液晶显示器的各类显示效果已经搓搓有余了。在后面的章节当中，还会学到利用这些函数，制作 WINDOWS 桌面效果的 GUI 图形。

六、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

效果如下图所示：



第五章 串口 UART

欢迎来到“串口”学习章节！

一、入门引导：

二、UART 寄存器及操作详解

㈠由浅入深之最简部分

程序详解

㈡由浅入深之提高

程序详解

三、实验程序下载和使用说明

一、入门引导：

UART 的英文全称为：Universal Asynchronous Receiver Transmitter，即通用串行异步收发器（我们平常所说的 SPI 是通用串行同步收发器）。几乎任何一款单片机上都带有串口，从 51 单片机走过来的我们，一说到串口，就会想到两根引脚：TXD 与 RXD。没有学过 51 单片机的也不用怕（好像没有没学过 51 的吧^_^），串口其实很简单！我们单片机上的串口用的用途一般有两个，一个是和电脑通讯，另外一个是控制一些串转并（例如：4094）的芯片！而现在大多数芯片，都已经改成了 SPI 控制或 I2C 控制。所以，我们只需要掌握了单片机上的串口怎样和电脑（专业术语称为“PC”或是“上位机”）通讯可以了。

LPC11XX 上只有一个串口，但不仅仅只有 RXD 和 TXD，它拥有 9 针串口的所有引脚，所以用它来开发调制解调器（Modem）控制器是很方便的。不过，我们一般的和电脑通信的应用，只需要用 RXD 和 TXD 两只脚就完全可以了。

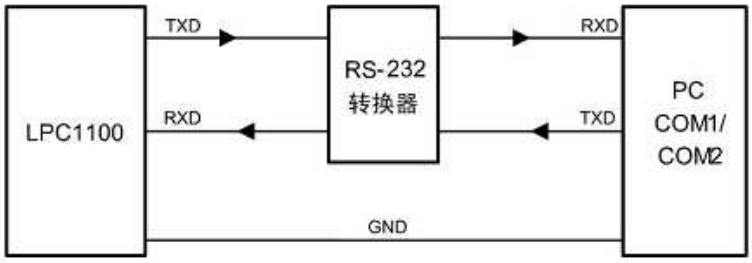


图 1：LPC11XX 与电脑通信

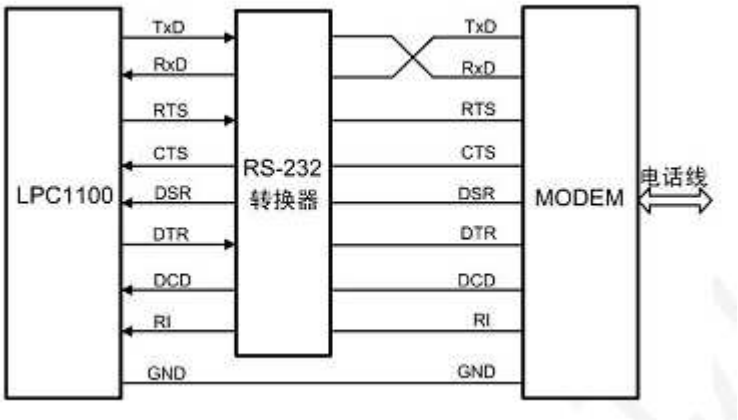


图 2：LPC11XX 与 Modem 通信

二、UART 寄存器及操作详解

关于 UART 的寄存器一共有 19 个。

下面是 NXPLPC11XX.H 文件中对 UART 模块所有的寄存器的存储器地址映射

```
/*--------------------- 通用异步收发器模块(UART) 基址：0x40008000 -------------*/
typedef struct
{
```

```
    union
    {
        R_en   RBR;              /*接收缓冲寄存器，      地址偏移：0x000 (R)   */
        W_en   THR;              /*发送保持寄存器，      地址偏移：0x000 (W)   */
        RW_en  DLL;              /*除数锁存低位寄存器，地址偏移：0x000 (R/W) */
    };
    union
    {
        RW_en  DLM;              /*除数锁存高位寄存器，地址偏移：0x004 (R/W) */
        RW_en  IER;              /*中断允许寄存器，      地址偏移：0x004 (R/W) */
    };
    union
    {
        R_en   IIR;              /*中断状态寄存器，        地址偏移：0x008 (R) */
        W_en   FCR;              /*FIFO 控制寄存器，      地址偏移：0x008 (W) */
    };
    RW_en  LCR;                  /*数据格式控制寄存器，              地址偏移：0x00C (R/W) */
    RW_en  MCR;                  /*Medom 控制寄存器，                地址偏移：0x010 (R/W) */
    R_en   LSR;                  /*发送接收状态寄存器，              地址偏移：0x014 (R)   */
    R_en   MSR;                  /*Modem 状态寄存器，                地址偏移：0x018 (R)   */
    RW_en  SCR;                  /*暂存寄存器，                      地址偏移：0x01C (R/W) */
    RW_en  ACR;                  /*自动波特率控制寄存器，            地址偏移：0x020 (R/W) */
   uint32 RESERVED0;             /*                保留区，禁止在此操作                    */
    RW_en  FDR;                  /*分数分频寄存器，                  地址偏移：0x028 (R/W) */
   uint32 RESERVED1;             /*                保留区，禁止在此操作                    */
    RW_en  TER;                  /*发送允许寄存器，                  地址偏移：0x030 (R/W) */
   uint32 RESERVED2[6];          /*                保留区，禁止在此操作                    */
    RW_en  RS485CTRL;            /*RS-485/EIA-485 控制寄存器，       地址偏移：0x04C (R/W) */
    RW_en  ADRMATCH;             /*RS-485/EIA-485 地址匹配寄存器，   地址偏移：0x050 (R/W) */
    RW_en  RS485DLY;             /*RS-485/EIA-485 方向控制延迟寄存器，地址偏移：0x054 (R/W) */
    R_en   FIFOLVL;              /*FIFO 状态寄存器，                 地址偏移：0x058 (R)   */
}UART_TypeDef;
/*------------------------------------------------------------------------------*/
```

我们要与电脑通信需要用到的寄存器一共有 14 个。她们分别是：

（寄存器以如下顺序描述，相信比看 LPC11XX 的数据手册要明了多了）

LCR	数据传输格式控制寄存器
DLM	除数锁存高位寄存器
DLL	除数锁存低位寄存器
FDR	分数分频寄存器

RBR	接收缓存寄存器
THR	发送保持寄存器
LSR	发送接收状态寄存器
FCR	FIFO 控制寄存器
FIFOLVL	FIFO 状态寄存器
IER	中断允许寄存器
IIR	中断状态寄存器
TER	发送允许寄存器
SCR	暂存寄存器
ACR	自动波特率控制寄存器

怎么这么多！不必急！实现最简单的和电脑通讯，只需要用到 7 个寄存器，她们分别是：

LCR	数据传输格式控制寄存器
DLM	除数锁存高位寄存器
DLL	除数锁存低位寄存器
FCR	FIFO 控制寄存器
RBR	接收缓存寄存器
THR	发送保持寄存器
LSR	发送接收状态寄存器

㈠由浅入深之最简部分

先来个示例程序，下面程序包括 3 个函数：1.初始化 UART 口；2.发送；3.接收。

1.初始化 UART 口

```
void UART_init(uint32 baudrate)
{
    uint32 DL_value,Clear=Clear;
    //把 P1.6 脚设置为 RXD
    IOCON->PIO1_6 &= ~0x07;
    IOCON->PIO1_6 |= 0x01;
    //把 P1.7 脚设置为 TXD
    IOCON->PIO1_7 &= ~0x07;
    IOCON->PIO1_7 |= 0x01;
    //允许 UART 时钟
```

```
    SYSCON->SYSAHBCLKCTRL |= (1<<12);
    //时钟分频值设置为 1
    SYSCON->UARTCLKDIV = 0x1;
    //8 位传输，1 个停止位，无几偶校验，允许访问除数锁存器
    UART->LCR = 0x83;
    //计算该波特率要求的除数锁存寄存器值
    DL_value = 48000000/16/baudrate ;
    //写除数锁存器高位值
    UART->DLM = DL_value / 256;
    //写除数锁存器低位值
    UART->DLL = DL_value % 256;
    //DLAB 置 0 ，禁止对除数锁存器的访问
    UART->LCR = 0x03;
    //允许 FIFO，清空 RxFIFO 和 TxFIFO
    UART->FCR = 0x07;
    //读 UART 状态寄存器将清空残留状态
    Clear = UART->LSR;

 }
```

2.发送数据到 PC

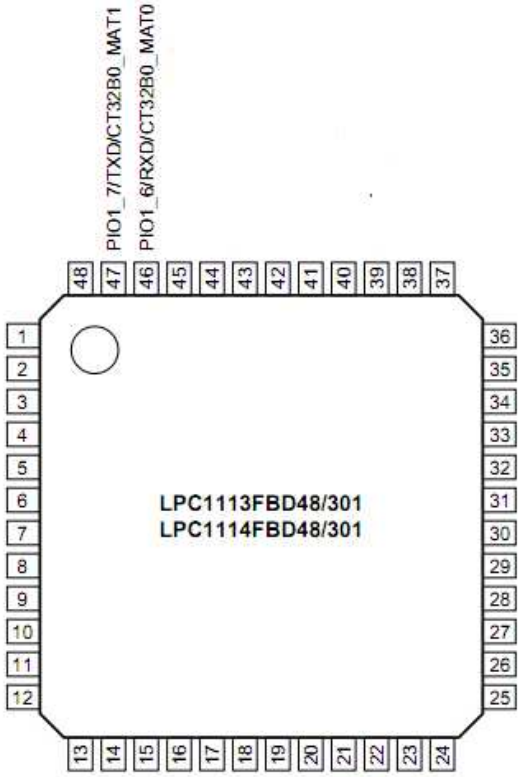
```
void UART_send(uint8 *Buffer, uint32 Length)
{
    while(Length != 0)//观察数据有没有发送完
    {
        while ( !(UART->LSR & (1<<5)) );//等待发送完成
        UART->THR = *Buffer;//把数据写到发送寄存器
        BufferPtr++;//取下一个数据
        Length--;//取下一个数据
    }
}
```

3.从 PC 接收数据

```
uint8 UART_recive(void)
{
    while(!(UART->LSR & (1<<0)));//等待接收到数据
    return(UART->RBR);           //读出数据
}
```

程序详解：

程序 1：初始化串口（UART）



我们可以看到：

RXD 与 P1.6 和 CT32B0_MAT0 复用

TXD 与 P1.7 和 CT32B0_MAT1 复用

这两根引脚默认的情况下是 P1.6 和 P1.7,并且是输入状态！

我们首先需要把这两个引脚重新设置为 RXD 和 TXD 引脚：

```
//把 P1.6 脚设置为 RXD
IOCON->PIO1_6 &= ~0x07;
IOCON->PIO1_6 |= 0x01;
//把 P1.7 脚设置为 TXD
IOCON->PIO1_7 &= ~0x07;
IOCON->PIO1_7 |= 0x01;
```

PIO1_6 寄存器的 bit0,bit1,bit2 这 3 个位共同决定了这个引脚是什么功能。

（引脚配置在官方数据手册的第 7 章）

PIO1_6

Bit2	Bit1	Bit0	选择功能
0	0	0	PIO1_6(默认值)
0	0	1	RXD
0	1	0	CT32B0_MAT0

PIO1_7

Bit2	Bit1	Bit0	选择功能
0	0	0	PIO1_7（默认值）
0	0	1	TXD
0	1	0	CT32B0_MAT1

配置用了两句话。第一句话是用来先清空原来的状态，第二句话就是把她重

新设置为 RXD 或 TXD。(之所以要先清空原来的状态是因为，假如在设置为 RXD 或 TXD 之前她被用来做 CT32B0_MAT 脚的话，写 IOCON->PIO1_7 |= 0x01;之后，这 3 位的值其实是 011，看功能表，什么都不代表，而且写这个数也是数据手册不允许的)

配置完引脚后，就该使能 UART 的时钟了：

```
//允许 UART 时钟
SYSCON->SYSAHBCLKCTRL |= (1<<12);
```

写 SYSAHBCLKCTRL(系统外围时钟控制器)的 bit12 为 0 或 1，可以控制 UART 时钟的有或无。

设置 UART 分频值：

```
//时钟分频值设置为 1
SYSCON->UARTCLKDIV = 0x1;
```

UART 时钟值=系统时钟/分频值，所以当分频值为 1 的时候，UART 时钟最大，传输数据也就最快！

初始化为外围以后，就该操纵 UART 的寄存器了，首先，我们设置串口通信传输格式，最常用的一种传输格式就是：8 位数据，1 个停止位，无几偶校验。格式设置在 LCR 这个寄存器里面设置。

位	符号	值	描述	复位值
1:0	Word Length Select	00 01 10 11	5 位字符长度 6 位字符长度 7 位字符长度 8 位字符长度	0
2	Stop Bit Select	0 1	1 个停止位 2 个停止位（若 U0LCR[1:0]=00 时为 1.5 个停止位）	0
3	Parity Enable	0 1	禁止校验的产生和检测 使能校验的产生和检测	0
5:4	Parity Select	00 01 10 11	奇校验。1s 内的发送字符数和附加校验位为奇数 偶校验。1s 内的发送字符数和附加校验位为偶数 强制“1”奇偶校验（stick parity） 强制“0”奇偶校验（stick parity）	0
6	Break Control	0 1	禁止间隔传输 使能间隔传输。当 U0LCR[6]是高电平有效时，强制使输出管脚 UART TXD 为逻辑 0	0
7	Divisor Latch Access Bit (DLAB)	0 1	禁止对除数锁存器的访问 使能对除数锁存器的访问	0
31:8	-	-	保留	-

除了格式设置外，LCR 寄存器的 bit7 还控制着能否对除数锁存器进行访问(除数锁存器是设置波特率要用到的除数，不用急，再往下看就讲到了)

根据我们要求的条件：

```
//8 位传输，1 个停止位，无几偶校验，允许访问除数锁存器
```

```
UART->LCR = 0x83;
```

设置完了格式，再设置波特率。波特率有一个计算公式，如下：

$$UART波特率 = \frac{Fpclk}{16\times(U0DLM:U0DLL)}\times\frac{MULVAL}{MULVAL+DIVADDVAL}$$

由计算公式可知：波特率是由 Fpclk\U0DLM\U0DLL\MULVAL\DIVADDVAL 决定的！
Fpclk:是 UART 分频以后的时钟；
U0DLM:是除数锁存器高八位值；
U0DLL:是除数锁存器低八位值；
MULVAL:小数分频器的乘数值；
DIVADDVAL:小数分频器的除数值；

小数分频器寄存器（FDR）：

位	功能	描述	复位值
3:0	DIVADDVAL	产生波特率的预分频除数值。如果该字段为 0，小数波特率发生器将不会影响 UART 的波特率	0
7:4	MULVAL	波特率预分频乘数值。不管是否使用小数波特率发生器，为了让 UART 正常运作，该字段必须大于或等于 1	1
31:8	-	保留。用户软件不应对其写入 1。从保留位读出的值未定义	0

由上图寄存器值可知，在默认情况下的波特率为：

$$UART波特率 = \frac{Fpclk}{16\times(U0DLM:U0DLL)}\times 1$$

在一般情况下，我们不必要用小数分频器，所以，由上式可知，我们只需要确定了除数锁存器的值，就可以设置波特率了。在应用的时候，我们要求在一定的波特率下通信，所以实际上是知道波特率，求除数锁存器的值。

DL_value = Fpclk/16/UART 波特率

```
//计算该波特率要求的除数锁存寄存器值
DL_value = 48000000/16/baudrate ;
//写除数锁存器高位值
UART->DLM = DL_value / 256;
//写除数锁存器低位值
UART->DLL = DL_value % 256;
```

设置好了波特率，为了稳定，我们就不希望程序去意外的修改这个值，所以现在，我们该把除数锁存器访问允许位关掉了。

```
//DLAB 置 0 ，禁止对除数锁存器的访问
UART->LCR = 0x03;
```

LPC11XX 与 PC 通信，数据是在 RxFIFO 和 TxFIFO 中进进出出的，而默认情况下，FIFO 还处于关闭的状态。我们现在该把她打开了：

```
//允许 FIFO，清空 RxFIFO 和 TxFIFO
UART->FCR = 0x07;
```

FIFO 控制寄存器（FCR）

位	符号	描述	复位值
0	FIFO 使能	0：UART FIFO 被禁止。禁止在应用中使用 1：高电平有效，使能对 UART Rx FIFO 和 Tx FIFO 以及 U0FCR[7:1]的访问。该位必须置位以实现正确的 UART 操作。该位的任何变化都将使 UART FIFO 清空	0
1	Rx FIFO 复位	0：对两个 UART FIFO 均无影响 1：写 1 到 U0FCR[1]将会清零 UART Rx FIFO 中的所有字节，并复位指针逻辑，该位可以自动清零	0
2	Tx FIFO 复位	0：对两个 UART FIFO 均无影响 1：写 1 到 U0FCR[2]将会清零 UART Tx FIFO 中的所有字节，并复位指针逻辑。该位会自动清零	0
3	-	保留	0
5:4	-	保留。用户软件不应对其写入 1。从保留位读出的值未定义	NA
7:6	Rx 触发选择	这两个位决定了接收 UART FIFO 在激活中断前必须写入的字符数量 00：触发点 0（默认 1 字节或 0x01） 01：触发点 1（默认 4 字节或 0x04） 10：触发点 2（默认 8 字节或 0x08） 11：触发点 3（默认 14 字节或 0x0E）	0
31:8	-	保留	-

LPC11XX 怎么知道有没有收到数据，或是有没有发送出去数据，是通过读“发送接收状态寄存器（LSR）”来判断的。（关于 LSR 的详细描述见官方数据手册）
读 LSR 的 bit0 可以知道有没有数据接收到：0 为空，1 为接收到数据；
读 LSR 的 bit5 可以知道有没有发送完数据：0 为有数据，1 为空。
除了 bit0 和 bit5，其它的 bit 有些需要读一下，才能清掉以前的状态。为了每次都能保证我们要发送和接收的字节都是我们所要求的。在这里，我们先读一下这个寄存器，清掉原来的状态。

```
//读 UART 状态寄存器将清空残留状态
Clear = UART->LSR;
```

到这里，我们的初始化函数就完成了。可以开始接收和发送数据了。
（解释一下，初始化函数里定义变量用

```
uint32 Clear=Clear;
```

其实和

```
uint32 Clear;
```

是一样一样的，这样定义是为了解决编译以后产生的一个 Warining，

```
Warining : variable ”Clear” was set but never used
```

```
)
```

程序 2：发送数据到 PC

这个程序看起来比较简单吧

```
void UART_send(uint8 *Buffer, uint32 Length)
{
  while(Length != 0)//观察数据有没有发送完
  {
      while ( !(UART->LSR & (1<<5)) );//等待发送完成
      UART->THR = *Buffer;//把数据写到发送寄存器
      BufferPtr++;//取下一个数据
      Length--;//取下一个数据
  }
}
```

此程序有两个入口参数：

Buffer 是要发送的数据数组；

Length 是要发送数据的字节数；

看了初始化函数的详细解释，看这个函数就很轻松了！

程序 3：从 PC 接收数据

```
uint8 UART_recive(void)
{
    while(!(UART->LSR & (1<<0)));//等待接收到数据
    return(UART->RBR);           //读出数据
}
```

这个函数就更简单了，执行一次这个函数，就可以从 PC 接收一个字节，之所以没有写接收一个数组的函数，是本人觉得接收一个数组的函数没有意义！为什么呢？原因很简单，第一、我们在接收数据之前不知道要接收到多少字节，但发送的时候当然是已知的；第二、我们利用这个接收一个字节的函数，就可以接收一个任意长度的数据数组了。下面主函数程序的功能就是实现接收到一组任意长度的数组数据后，在给电脑发回相同的数组数据，并在我们开发板的 LCD 上显示接收到的数据。

```
int main(void)
{
    uint8 byte,xposition=50;//定义接收字节存放和 LCD 显示横坐标位置
    uint8 buf[]={0x00};//定义发送数据数组

    SysCLK_config();//系统时钟设置
    GPIO_init();//GPIO 初始化
    UART_init(115200);//初始化串口
    LCD_Init();//初始化 LCD
    LCD_Clear(WHITE);//LCD 底色选为白色

    while(1)
    {
        byte=UART_recive();//等待接收 PC 传过来的数据
        buf[0]=byte;//把接收到的字节给了发送寄存器
```

```
            LCD_ShowString(xposition, 200, buf);//在 LCD 上显示接收到数据
            xposition = xposition + 6;
            UART_send(buf,1);    //把接收到数据传回电脑
        }
    }
```

以上函数的执行效果为，假如你在电脑的“串口调试助手”上写入“hello”并发送，你会看到在开发板的 LCD 上显示出了“hello”，并且在“串口调试助手”的“接收区域”也显示“hello”。

㈡由浅入深之提高

细心的你还有专业的你一定会发现，上面是采用查询的方式实现的串口通信。“查询方法”在单片机应用中是一种最没有效率的方法了。在实际应用中，我们想让单片机做更多的事情，而不是一直在等待 PC 发送数据过来。这就要用到串口的中断了。

我们先来看一下关于串口中断的两个寄存器：

(中断允许寄存器)IER

(中断状态寄存器)IIR

以 ATMEL AT89 系列典型 51 单片机为例，启动串口中断方式为：先开总中断 EA，再开串口中断 ES

```
EA = 1;
ES = 1;
```

开了这个 ES 以后，就可以产生串口中断了，当 SBUF 有数据发送和接收到数据以后，就会产生中断。它的缺点是显而易见的，即不能够分别控制启动关闭接收中断和发送中断。只要 ES=1，开了串口中断，不管发送还是接收，都会产生中断。

在 Cortex-M0 内核的 LPC1114 上的串口中断，可以实现接收中断和发送中断的分别控制开启和关闭。不仅如此，她还可以产生其它的比如串口数据错误中断等等。

首先，开启 NVIC 中断：NVIC_EnableIRQ(UART_IRQn);执行完这句话以后，串口还不能够产生中断，需要进一步对串口模块的中断允许寄存器 IER 的某些位进行设置才可以。在这个 32 位的寄存器 IER 中，我们可以控制它的 bit0、bit1、bit2、bit8、bit9。你可以看到，只用到了 5 个 bit。

bit0:接收中断允许和接收超时中断允许；

bit1:发送中断允许；

bit2:RX 线中断允许（即决定产生诸如几偶校验错误、停止位错误后是否产生中断）

bit8:自动波特率结束中断；

bit9:自动波特率超时中断；

在这里，我们讲一下用的最多的“开启接收中断”方法接收数据。先看示例程序：

```
uint16 xpos=50,ypos=50; // 定义液晶显示器初始化 XY 坐标

void UART_IRQHandler(void)
{
    uint32 IRQ_ID;         // 定义读取中断 ID 号变量
    uint8 buf[1]={0x00};  // 定义接收数据变量数组

    IRQ_ID = UART->IIR;    // 读中断 ID 号
    IRQ_ID =((IRQ_ID>>1)&0x7);// 检测 bit4:bit1
    if(IRQ_ID == 0x02 )        // 检测是不是接收数据引起的中断
    {
        buf[0] = UART->RBR;    // 从 RXFIFO 中读取接收到的数据
        LCD_ShowString(xpos,ypos,buf); // 把数据显示在液晶显示器上
        xpos+=6;               // 显示下一个数据做准备
        if(xpos>200)
        {
            xpos=50;
            ypos+=18;
        }
    }
    return;
}

int main(void)
{
    //配置时钟
    SysCLK_config();
    //使能 GPIO 时钟(bit6)，使能 IOCON 时钟(bit16)
    SYSCON->SYSAHBCLKCTRL |= (1<<6)|(1<<16);

    UART_init(115200); // 初始化串口，波特率 115200
    UART->IER = 0x01;   //只允许接收中断，关闭其他中断
    NVIC_EnableIRQ(UART_IRQn); //开启串口中断

    LCD_Init();         // 初始化液晶显示器
    LCD_Clear(GREEN);  // 整屏显示绿色
    POINT_COLOR=BLACK; // 定义笔的颜色为黑色
    BACK_COLOR=GREEN;  // 定义笔的背景色为绿色

    while(1);
    {
        ;
    }
```

```
}
```

程序详解：

上面程序的执行结果是：在串口调试软件输入框里面输入数据，点击发送，将会在液晶显示器上面显示出来。比如输入 hello world，点击发送，将会在液晶显示器上显示 hello world。

上面的程序包含一个主函数，一个串口中断处理函数。

主函数里面包括四个部分，一是时钟配置，二是串口配置，三是液晶显示器配置，四是个 while 循环，这里面可以写入用户需要执行的其他程序。

时钟配置部分和液晶显示配置部分不是这里的重点，这里讲一下串口的配置。首先执行串口初始化函数，使能串口，并把波特率设置为 115200，接下来的这条语句 UART->IER = 0x01;即允许串口接收中断。（注意：允许串口接收中断的时候，同时会允许串口接收超时中断）允许了串口接收中断以后，接下来，就需要正式开启串口中断了 NVIC_EnableIRQ(UART_IRQn);之后，如果串口接收到了数据，才会进入“串口中断处理函数”

串口中断处理函数：在这个函数当中，由于我们只允许了接收中断，所以只是检测了一下 IIR 寄存器内容，看看是不是接收有效数据引起的中断。如果是，就执行 if 条件里面的语句。这些语句即读取 RXFIFO 中的数据后，显示在液晶屏幕上。

看到这里，你会发现，在主函数的 while 里面，我们可以让 LPC1114 去执行其它的事情，只有在有数据从串口上过来，才会进行串口中断处理函数。这样的话，就比之前的那种查询方式效率高多了吧！

关于“发送中断”和其它“线中断”，大家可以自己深入研究一下。本手册只带领大家入门，其它的就靠自己了。

三、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

打开“串口调试助手”

双击此图标。打开后，如下图所示：

设置说明也在图片上表明了。其实设置的地方就两个，一个是你现在所用的串口，另外一个是波特率要改为 115200，即最大值。

“查询方式”的程序支持发送中文。

“中断方式”的程序只做了发送英文的程序。

注意：在使用串口的时候关掉 ISP 开关


串口调试助手 SComAssistant V2.2 For WIN9X/NT/2000
串口
COM1
波特率
11520
校验位
NONE
数据位
8
停止位
1
关闭串口
清空接收区
接收区
停止显示
自动清空
十六进制显示
保存显示数据
更改
C:\COMDATA
选择你的串口
在这里把波特率设置为115200，即最大值
最后点击这里
清空重填
发送的字符/数据
hello ration
这里输入要发送的字库
十六进制发送
手动发送
自动发送（周期改变后重选）
自动发送周期：
1000
毫秒
选择发送文件
还没有选择文件
发送文件
MAIL
WEB
关闭程序
STATUS：COM1 OPENED，
RX:0
TX:0
计数清零
帮助

第六章 I2C 总线接口

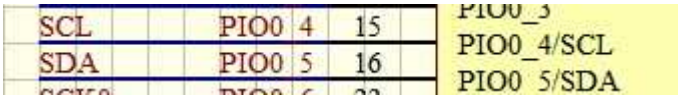
一、入门导读

I2C 总线是由飞利浦公司开发的两线制串行总线接口，现在被广泛的应用在很多的单片机或者是其它的芯片上，形成了一定的标准。比如我们很熟悉的 EEPROM 存储芯片 AT24CXX 系列，还有 LM75 集成温度传感器，MMA7455 加速度传感器等都可以用 I2C 总线对其进行控制。

LPC1114 上集成有一个 I2C 总线接口。

下面，我们采用两种方式来学习。
第一种方式：用 LPC1114 的 GPIO 口模拟 I2C 总线读写 AT24C16；
第二种方式：用 LPC1114 内部集成的 I2C 模块读写 AT24C16。

二、硬件连接



LPC1114 的 PIN15 和 PIN16 有两个功能，一个是 GPIO 引脚 P0.4 和 P0.5，另外一个是 I2C 通信用的 SCL 和 SDA 引脚。我们要选择这个引脚作为什么功能，就需要用 IOCON 寄存器，这两个脚默认是 GPIO 引脚 P0.4 和 P0.5。

下面介绍的“模拟 I2C”和“硬件 I2C”都是用这两个引脚，只不过，在用“模拟 I2C”通信的时候，把这个脚配置为 GPIO 引脚。在用“硬件 I2C 通信”的是后，把这两个脚配置为 SCL 和 SDA 引脚。

三、模拟 I2C 通信时序读写 AT24C16

3.1 软件设计

```
/*****************************************/
/* 函数功能：初始化 I2C 引脚（模拟 I2C）    */
/*****************************************/
void I2C_Init(void)
{
    GPIO0->DIR |= (1<<4)|(1<<5);   //     设置 P0.4,P0.5 脚为输出
    GPIO0->DATA |= (1<<4)|(1<<5); // 输出高电平  SCL=1 SDA=1
}

/*****************************************/
/* 函数功能：发 I2C 起始信号（模拟 I2C）    */
/*****************************************/
void I2C_Start(void)
{
    GPIO0->DIR |= (1<<5);        // 设置 SDA 引脚为输出
    GPIO0->DATA   |= (1<<5);     // SDA=1;
    delay_us(5);
    GPIO0->DATA   |= (1<<4);     // SCL=1;
```

```
    delay_us(5);
    GPIO0->DATA    &= ~(1<<5);    // SDA=0;
    delay_us(5);
}

/*****************************************/
/* 函数功能：发 I2C 结束信号（模拟 I2C）      */
/*****************************************/
void I2C_Stop(void)
{
    GPIO0->DIR |= (1<<5);         // SDA 输出
    GPIO0->DATA    &= ~(1<<5);    // SDA=0;
    delay_us(5);
    GPIO0->DATA    |= (1<<4);     // SCL=1;
    delay_us(5);
    GPIO0->DATA    |= (1<<5);     // SDA=1;
    delay_us(5);
}

/*****************************************/
/* 函数功能：等待应答信号 （模拟 I2C）      */
/* 出口参数：1：接收应答失败                */
/*           0：接收应答成功                */
/*****************************************/
uint8 I2C_Wait_Ack(void)
{
    uint8 ack_sign;

    GPIO0->DIR|=(1<<5);         // 设置 SDA 引脚为输出
    GPIO0->DATA|=(1<<5);        // SDA=1;
    GPIO0->DATA|=(1<<4);        //SCL=1;
    GPIO0->DIR    &= ~(1<<5);   //SDA 设置为输入
    delay_us(5);
    if((GPIO0->DATA&(1<<5))==(1<<5))ack_sign=1;
    else ack_sign=0;
    GPIO0->DATA&=~(1<<4);       //SCL=0;

    return ack_sign;
}

/*****************************************/
/* 函数功能：发送一个字节数据（模拟 I2C）*/
/* 入口参数：wbyte:要发送的字节           */
```

```
/*****************************************/
void I2C_Send_Byte(uint8 wbyte)
{
    uint8 i,temp,temp1;
    GPIO0->DIR |= (1<<5);       // 设置 SDA 引脚为输出
    temp1=wbyte;
    for(i=0;i<8;i++)
    {
        GPIO0->DATA&=~(1<<4); // SCL=0;
        delay_us(5);
        temp=temp1;
        temp=temp&0x80;
        if(temp==0x80)
        GPIO0->DATA |= (1<<5); // SDA=1;
        else
        GPIO0->DATA &= ~(1<<5);// SDA=0;
        delay_us(5);
        GPIO0->DATA   |= (1<<4); // SCL=1;
        delay_us(5);
        GPIO0->DATA   &= ~(1<<4);// SCL=0;
        delay_us(5);
        temp1<<=1;
    }
}

/*****************************************/
/* 函数功能：读一个字节数据（模拟 I2C）    */
/* 出口参数：rebyte:读出的字节              */
/*****************************************/
uint8 I2C_Read_Byte(void)
{
    uint8 i,rebyte=0;
    GPIO0->DIR &= ~(1<<5);   //SDA 设置为输入
    for(i=0;i<8;i++ )
    {
        rebyte<<=1;
        delay_us(5);
        GPIO0->DATA   &= ~(1<<4);   // SCL=0;
        delay_us(5);
        GPIO0->DATA   |= (1<<4);     // SCL=1;
        delay_us(5);
        if((GPIO0->DATA&(1<<5))==(1<<5))//if(SDA==1)
        rebyte|=0x01;
```

```
    }
    return rebyte;
}
```

3.2 程序详解

有上面的程序可以看出，我们并没有开启 LPC1114 的内部 I2C 模块，而是直接控制 GPIO 口的高低电平在实现 I2C 时序的，其实，这个程序并不陌生，如果你曾今写过 51 单片机读写 24C16 的程序，就会发现了。上面这些程序即是我把前的 1 单片机读写 24C16 程序稍微修改了以后得到的。需要注意的地方就是，要合理的转换 GPIO 口的输入和输出状态，当我们给 AT24C16 写数据或命令的时候，SDA 引脚是作为输出引脚的，当读 AT24C16 中的数据的时候，SDA 引脚是作为输入引脚的，如果此时你不把 SDA 引脚设置为输入，读出的值将会永远是 0xFF。

以上的程序是在 i2c.c 文件中定义的。要实现读写 AT24C16，需要根据 AT24C16 的时序写好读写函数。AT24C16 的时序问题，这里就不涉及了，你可以在该章对应的程序中的 at24c16.c 文件当中找到相关函数。

四、硬件 I2C 模块读写 AT24C16

4.1 软件设计

```
/*****************************************/
/* 函数功能：初始化 LPC1114 I2C 模块          */
/* 入口参数：Mode :0，慢速模式               */
/*                 1，快速模式               */
/*****************************************/
void I2C_Init(uint8 Mode)
{
    SYSCON->PRESETCTRL |= (1<<1); // De-asserted I2C 模块（在启动 I2C 模块之前，必须向该位写 1）
    SYSCON->SYSAHBCLKCTRL |= (1<<5); // 使能 I2C 时钟
    SYSCON->SYSAHBCLKCTRL |= (1<<16); // 使能 IOCON 时钟
    IOCON->PIO0_4 &= ~0x3F;
    IOCON->PIO0_4 |= 0x01;          // 把 P0.4 脚配置为 I2C SCL
    IOCON->PIO0_5 &= ~0x3F;
    IOCON->PIO0_5 |= 0x01;          // 把 P0.5 脚配置为 I2C SDA
    SYSCON->SYSAHBCLKCTRL |= (1<<16); // 禁能 IOCON 时钟
    if(Mode == 1) // 快速 I2C 通信 (大约 400KHz 传输速率)(AT24C16 支持 400K 快速模式)
    {
        I2C->SCLH = 47;
        I2C->SCLL = 93;
    }
    else       // 低速 I2C 通信    (大约 100KHz 传输速率)
    {
        I2C->SCLH = 47*4;
        I2C->SCLL = 93*4;
```

```
    }
    I2C->CONCLR = 0xFF;   // 清所有标志
    I2C->CONSET |= I2CONSET_I2EN;   // 使能 I2C 接口
}

/*****************************************************/
/* 函数功能：发送停止信号                            */
/*****************************************************/
void I2C_Stop(void)
{
    I2C->CONCLR = I2CONCLR_SIC;      // 清 SI 标志位
    I2C->CONSET |= I2CONSET_STO;     // 发送停止信号
}

/*****************************************************/
/* 函数功能：I2C 发送命令数据                        */
/* 入口参数：Ctrl：命令+地址字节                     */
/* 出口参数：0：成功                                 */
/*           1：失败                                 */
/*****************************************************/
uint8 I2C_Send_Ctrl(uint8 CtrlAndAddr)
{
    uint8 res;

    if(CtrlAndAddr & 1)      // 如果是读命令
        res = 0x40; // 40H 代表开始信号和读命令已经传输完毕，并且已经接收到 ACK
    else                     // 如果是写命令
        res = 0x18; // 18H 代表开始信号和写命令已经传输完毕，并且已经接收到 ACK
    // 发送开始信号
    I2C->CONCLR = 0xFF; // 清所有标志位
    I2C->CONSET |= I2CONSET_I2EN | I2CONSET_STA;   // 使能发送开始信号
    while(!(I2C->CONSET & I2CONSET_SI));    // 等待开始信号发送完成
    // 发送命令+地址字节
    I2C->DAT = CtrlAndAddr;    // 把要发送的字节给了 DAT 寄存器
    I2C->CONCLR = I2CONCLR_STAC | I2CONCLR_SIC; // 清除开始 START 位和 SI 位
    while(!(I2C->CONSET & I2CONSET_SI));   // 等待数据发送完成
    if(I2C->STAT != res)       // 观察 STAT 寄存器响应的状态，判断是否正确执行读或写命
令
    {
        I2C_Stop();            // 没有完成任务，发送停止信号，结束 I2C 通信
        return 1;              // 返回 1，表明失败！
    }
    return 0;              // 如果正确执行返回 0
```

```
}
/*****************************************************/
/* 函数功能：I2C 发送一字节数据                      */
/* 入口参数：sebyte：要发送的字节                    */
/*****************************************************/
void I2C_Send_Byte(uint8 sebyte)
{
    I2C->DAT = sebyte;    // 把字节写入 DAT 寄存器
    I2C->CONCLR = I2CONSET_SI;    // 清除 SI 标志
    while(!(I2C->CONSET & I2CONSET_SI));   // 等待数据发送完成
}

/*****************************************************/
/* 函数功能：I2C 接收一字节数据                      */
/* 入口参数：rebyte：要接收的字节                    */
/*****************************************************/
uint8 I2C_Recieve_Byte(void)
{
    uint8 rebyte;

    I2C->CONCLR = I2CONCLR_AAC | I2CONCLR_SIC;   // 清 AA 和 SI 标志
    while(!(I2C->CONSET & I2CONSET_SI));   // 等待接收数据完成
    rebyte = (uint8)I2C->DAT;   // 把接收到的数据给了 rebyte

    return rebyte;
}
```

4.2 程序详解

看初始化函数，这里通过 IOCON 寄存器把 P0.4 和 P0.5 引脚设置成了 SCL 和 SDA 引脚。而且开启了 I2C 通信时钟。这里需要讲一下先关的寄存器，首先看 PRESETCTRL，这个 32 位寄存器有 4 个我们可以控制的位，其它位保留。这四个位分别对应 SSP0 I2C SSP1 CAN。你可以看到，这个寄存器管理着通信相关的模块。其中第四位 bit3 是 CAN 控制位，只有在 LPC11CXX 控制器上才起作用。这个寄存器的名字叫做“外设复位控制寄存器”，看名字真是搞不懂怎么用这个寄存器。看看说明便明了，数据手册上说，在每次开启 SSP0 I2C SSP1 CAN 模块的时候，都需要先向这个寄存器的对应位写 1，目的是为了 de-assert 对应的模块。在这里，我真是看不懂为什么，所以就不误导你们了，按照数据手册上说的照做就可以了。

接下来，就是开启了 SYSAHBCLKCTRL 的 bit5，该位对应 I2C 时钟。在配置完 SDA 和 SCL 功能之后，就可以对 I2C->SCLH 和 I2C->SCLL 写值，从这个英文字面看，就可以知道，这两个寄存器分别控制着 SCL 的高电平和低电平。

接下来的两个寄存器，CONCLR 和 CONSET。是下面的函数用的最多的两个寄存器。CONCLR 是“清除标志位寄存器”，CONSET 是“设置标志位”寄存

器。简单一点说，CONSET 寄存器控制着 I2C 通信的命令、时序和开启关闭，而且还可以观察状态。CONCLR 寄存器的位对应着 CONSET 的位，对 CONSET 的对应位清除。

```
while(!(I2C->CONSET & I2CONSET_SI));  // 等待接收数据完成
```

看上面这条语句，在上面的函数中出现的次数很多。这就是 CONSET 寄存器用作观察状态的用法，它的 bit3 叫做 SI，这个位几乎在 I2C 通信当中发生任何事情都会被自动置位，这些事情可以是发开开始信号，收到数据等等。

对 CONCLR 寄存器对应的位写 1 清除对应的位标志。

I2C->DAT 寄存器和 GPIO->DATA 寄存器的用法一样。

以上的程序是在 i2c.c 文件中定义的。要实现读写 AT24C16，需要根据 AT24C16 的时序写好读写函数。AT24C16 的时序问题，这里就不涉及了，你可以在该章对应的程序中的 at24c16.c 文件当中找到相关函数。

五、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

主函数的实现原理：

首先，向 24C16 当中写入一个字节数据，然后，读出来检测，通过这种方法来检测 AT24C16 是否存在。

然后分别使用了读写多字节函数。

两种实现方式的主函数一模一样，效果也是一模一样！

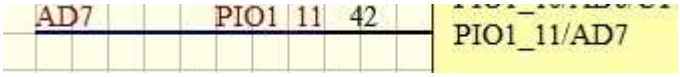
第七章 ADC 测温

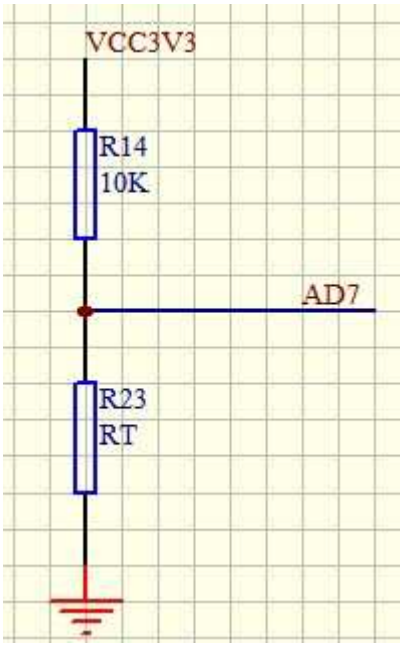
一、入门导读
二、硬件连接
三、软件设计
四、程序详解
五、实验程序下载和使用说明

一、入门导读

LPC1114 内部集成了一个 ADC 模块。可以复用到 8 个引脚上来使用，而且可以 8 个引脚同时使用。分辨率可调，最大分辨率为 10 位，即最小分度值：3.3/1024V = 3.22mV。这个 ADC 模块可以采用软件控制和硬件扫描两种方式测量引脚上的电压。在这里，我将介绍用硬件扫描方式测量电压，对于软件控制方式，周立功的手册上写的很详细，大家可以去看一下。

二、硬件连接





在 MINI LPC1114 上，用了 AD7 这个引脚。图中 RT 即为高精度热敏电阻 MF58，经过实测，这个电阻精度好，稳定性堪称优秀！做完这个例程，你就会感觉到了。

三、软件设计

由上图可知，热敏电阻和 10K 的普通电阻对 3.3V 分压。根据 MF58 的数据手册，可以查到对应温度的电阻值。所以，我们就可以在用 LPC1114 的 ADC 测出电压后，再利用分压公式，计算出 RT 的电阻值，然后根据电阻值，反查温度，即可完成。下面是每一步的程序。为了让东北的朋友和海南的朋友都能用到，我做的测量温度范围为：零下 40 度到零上 50 度。

```
/*****************************************/
/* 函数名称：初始化 ADC 口（AD7）                */
/*****************************************/
void ADC_Init(void)
{
    SYSCON->PDRUNCFG &= ~(0x1<<4);            // ADC 模块上电
    SYSCON->SYSAHBCLKCTRL |= (1<<13);         // 使能 ADC 时钟

    SYSCON->SYSAHBCLKCTRL |= (1<<16);         // 使能 IOCON 时钟
    IOCON->PIO1_11 &= ~0x9F;                  // 把 P1.11 引脚选择模拟输入方式
    IOCON->PIO1_11 |= 0x01;                   // 把 P1.11 引脚设置为 AD7 功能
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);        // 关闭 IOCON 时钟
    ADC->CR = (1<<7)|          /* bit7:bit0     选择通道 7 作为 ADC 输入，即 P1.11 引脚 */
              (23<<8 )|        /* bit15:bit8    把采样时钟频率设置为 2MHz 48/(23+1)*/
              (1<<16 )|        /* bit16         硬件扫描模式                          */
              (0<<17 )|        /* bit19:bit17 10 位模式                               */
              (0<<24 );        /* bit26:bit24  硬件扫描模式下这些位置 0               */
}

/*********************************************/
/* 函数功能：读取电压值（AD7）                    */
/* 出口参数：adc_value，读到的电压值              */
/*********************************************/
uint32 ADC_Read(void)
{
    uint32 adc_value;
    uint8 i;

    adc_value = ADC->DR[7];       // 读取第一个值
    adc_value = ADC->DR[7];       // 读取第二个值

    adc_value=0;                  // 把 adc_value 清零
    for(i=0;i<10;i++)             // 再连续读取 10 个电压值
    {
        adc_value += ((ADC->DR[7]>>6)&0x3FF);
        delay_us(1);
    }
    adc_value = adc_value/10;    // 把读到的 10 个电压值取平均值
    adc_value = (adc_value*Vref)/1024; // 转换为真正的电压值

    return adc_value;       // 返回结果
}
```

```
/*************************************************************/
/* 函数功能：把从 AD7 口读到的电压值转变为热敏电阻的电阻值     */
/* 入口参数：adc_value，电阻值                                 */
/* 出口参数：res_value，电阻值                                 */
/*************************************************************/
uint32 ADC_To_Res(uint32 adc_value)
{
    uint32 res_value;

    res_value = (adc_value*10000)/(Vref-adc_value);

    return res_value;
}

/*************************************************************/
/* 函数功能：把热敏电阻的电阻值转变为温度值                    */
/* 入口参数：res_value，电阻值                                 */
/* 出口参数：temp，温度值                                      */
/* 说    明：本函数根据 MF58 的数据手册编排，结果非常准确！   */
/*************************************************************/
uint8 Res_To_Temp(uint32 res_value)
{
    uint32 k;    // 斜率
    int32 temp;  //    温度值

    if( (res_value<29371)&&(res_value>18680) ) // 0~10 摄氏度
    {
        k = 1069;
        temp = ((29370-res_value)/k)+0;
    }
    else if( (res_value<18681)&&(res_value>12240) )    // 10~20 摄氏度
    {
        k = 644;
        temp = ((18680-res_value)/k)+10;
    }
    else if( (res_value<12241)&&(res_value>8221) ) // 20~30 摄氏度
    {
        k = 402;
        temp = ((12240-res_value)/k)+20;
    }
    else if( (res_value<8222)&&(res_value>5648) )  // 30~40 摄氏度
    {
        k = 257;
```

```
        temp = ((8221-res_value)/k)+30;
    }
    else if( (res_value<5649)&&(res_value>3958) )  // 40~50 摄氏度
    {
        k = 169;
        temp = ((5648-res_value)/k)+40;
    }
    else if( (res_value<47731)&&(res_value>29370) )     // -10~0 摄氏度
    {
        k = 1836;
        temp = ((47730-res_value)/k)-10;
    }
    else if( (res_value<80361)&&(res_value>47730) )     // -20~-10 摄氏度
    {
        k = 3263;
        temp = ((80360-res_value)/k)-20;
    }
    else if( (res_value<140001)&&(res_value>80360) )    // -30~-20 摄氏度
    {
        k = 5964;
        temp = ((140000-res_value)/k)-30;
    }
    else if( (res_value<249600)&&(res_value>140000) )  // -40~-30 摄氏度
    {
        k = 10960;
        temp = ((249600-res_value)/k)-40;
    }

    return((uint8)temp);    // 返回温度值
}

/*****************************************/
/* 函数功能：检测温度是在零上还是零下     */
/* 入口参数：热敏电阻的阻值               */
/* 出口参数：sign_sign, 1：零上          */
/*                      0：零下          */
/*****************************************/
uint8 Fetch_sign(uint32 res_value)
{
    uint8 sign_sign;

    if(res_value<29371)sign_sign = 1;
    else sign_sign = 0;
```

```
    return sign_sign;
}
```

四、程序详解

第一个函数，使能 AD7，并设置为硬件扫描方式。这个函数比较简单，讲了这么多章了，寄存器的用法我就不再手把手教了，不然显得有些啰嗦。现在为止，自己看上数据手册对应程序应该不成问题了。

第二个函数，是用来读取 AD7 引脚上的电压。从 ADC->DR[7]里面读数据即可得到。在这个函数当中，其实我们是读取了 12 个值，去掉了前两个值。去掉前两个值的目的，是为了让电压值变得准确。如果不去掉前两个值，读出的值和实际的值会相差很大的，不知道的人还以为是 LPC1114 的 ADC 模块坏了！

第三个函数，根据分压公式计算出此时热敏电阻的电阻值。

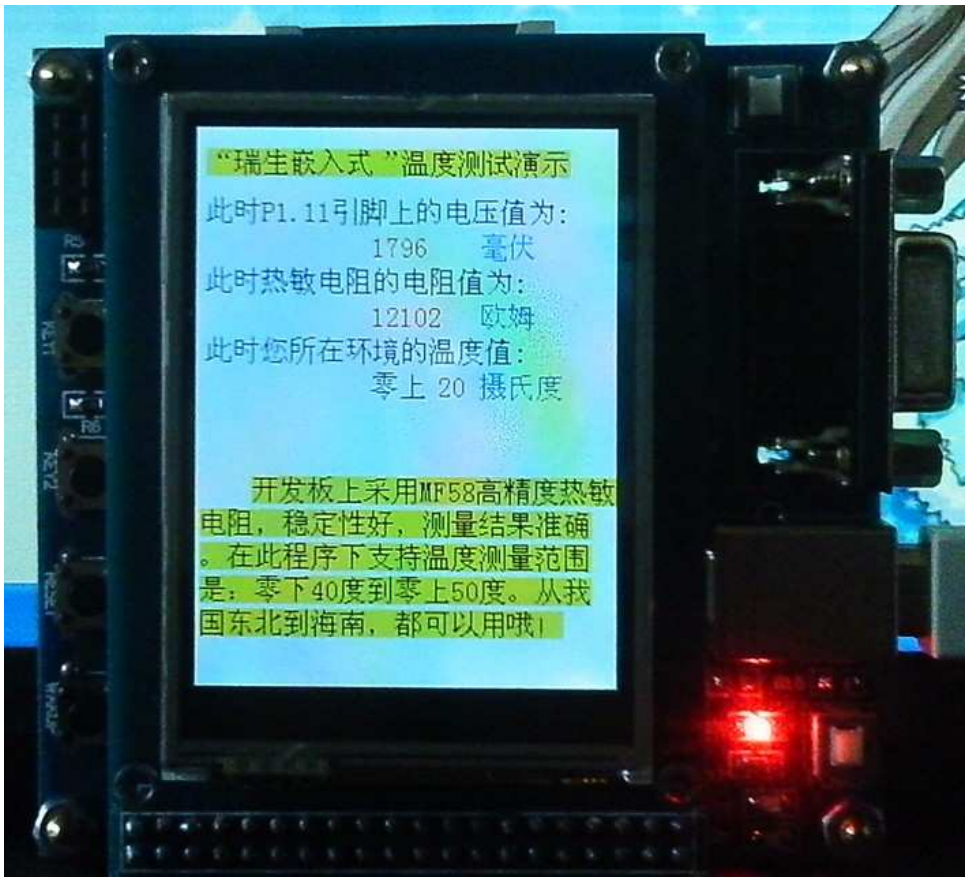
第四个函数，是根据热敏电阻的电阻值，计算出此时的温度值。这个函数是我根据 MF58 的数据手册编写的。感兴趣的朋友可以打开 MF58 的数据手册看看我这个函数的原理，其实很简单，就是假设在小范围内温度和电阻的值是线性变化的。

第五个函数，得到此时的温度是零上还是零下，也是根据电阻值确定的。

五、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

效果如下图所示：



以上的三个值一秒钟变化一次。你可以观察一下这三个值，相当的稳定！

第八章 看门狗的使用

一、入门引导
二、看门狗寄存器介绍
三、程序设计
四、程序详解
五、实验程序下载和使用说明

一、入门引导

看门狗的基本作用是在单片机由于某些不明原因产生错误时程序跑飞，不能及时喂狗而使单片机自动复位。在 51 单片机 AT89S52 里面，也有看门狗，也许有的朋友用过。在某些产品应用当中，看门狗是必须的，在这种情况下，如果单片机没有集成看门狗模块，就需要另外配置硬件看门狗芯片了，这样就增加了采购成本，设计成本，软件成本，还要占用电路板上一块空间。所以，到现在为止，看门狗已经成为单片机的一个不可或缺的模块组成了。

LPC1114 中也集成了看门狗模块，而且可以利用独立的看门狗时钟工作。

看门狗的使用十分简单。LPC1114 的看门狗寄存器一共有四个：

WDT->MOD：看门狗模式选择寄存器

WDT->TC：看门狗定时器值

WDT->FEED：看门狗喂狗寄存器

WDT->TV：看门狗当前值寄存器（这个寄存器一般不用）

二、看门狗寄存器介绍

上面的寄存器我们只用前三个，就可以操作看门狗了。

1. 模式寄存器（MOD）

看门狗有两种工作模式：

模式一：在定时时间内没有喂狗，复位 LPC1114；

模式二：在定时时间内没有喂狗，进入看门狗中断服务函数。

模式一设置：WDEN(bit0)=1　　WDRESET(bit1)=1

例：WDT->MOD |= 0x03; //设置为复位模式

模式二设置：WDEN(bit0)=1　　WDRESET(bit1)=0

例：WDT->MOD |= 0x01; //设置为中断模式

```
NVIC_EnableIRQ(WDT_IRQn); // 开启看门狗中断
```

要设置为模式二时，必须开启看门狗中断，并且要写好看门狗中断服务函数。

2. 定时器值寄存器（TC）

该寄存器是一个完整的 32 位寄存器（寄存器里面没有保留位），该值可以是 0~0XFFFFFFFF。不过，在对该寄存器写入的值小于 255 时，LPC1114 会自动把该值写了 255（0XFF）。

这是一个倒计时定时器，倒计时到 0 时，将置位（置位的意思就是把 0 变为 1）模式寄存器 MOD 当中的 bit2 和 bit3。模式寄存器中的 bit2 是看门狗超时标志 WDTOF，bit3 是看门狗中断标志位 WDINT。其中，WDTOF 可以软件清零，WDINT 不可以软件清零。

3. 喂狗寄存器（FEED）

喂狗时序为：先写 0XAA，再写 0X55。

例：

```
WDT->FEED = 0xAA;       // 喂狗
WDT->FEED = 0x55;
```

三、程序设计

```
/*******************************************/
/* 函数功能：启动看门狗时钟                  */
/* 说明：可以作为看门狗的时钟源有 3 个        */
/*          clksrc=0 选择 IRC 振荡器         */
/*          clksrc=1 选择主时钟              */
/*          clksrc=2 选择看门狗时钟          */
/*******************************************/
void WDT_CLK_Setup(void)
{
    SYSCON->PDRUNCFG &= ~(0x1<<6);     // 看门狗振荡器时钟上电（bit6）
    SYSCON->WDTOSCCTRL = (0x1<<5);     // DIVSEL=0 FREQSEL=1 WDT_OSC_CLK=250KHz
    SYSCON->WDTCLKSEL = 0x2;           // 选择看门狗时钟源
    SYSCON->WDTCLKUEN = 0x01;          // 更新时钟源
    SYSCON->WDTCLKUEN = 0x00;          // 先写 0，再写 1 达到更新目的
    SYSCON->WDTCLKUEN = 0x01;
    while ( !(SYSCON->WDTCLKUEN & 0x01) );  // 等待更新成功
    SYSCON->WDTCLKDIV = 1;             // 设置看门狗分频值为 1
    return;
}

/*******************************************/
/* 函数功能：使能看门狗                      */
/*******************************************/
void WDT_Enable(void)
{

    WDT_CLK_Setup();
    SYSCON->SYSAHBCLKCTRL |= (1<<15);// 允许 WDT 时钟，这个时钟是配置寄存器用的
    WDT->TC = 80000;          // 给看门狗定时器赋值，定时时间大约 1 秒(这是在 wdt_clk=250KHz 时)
    WDT->MOD |= 0x03;         // 写值 0x03：不喂狗产生复位  写值 0x01：不喂狗发生中断
    WDT->FEED = 0xAA;         // 喂看门狗，开启
    WDT->FEED = 0x55;
    return;
}

/*******************************************/
/* 函数功能：看门狗中断服务函数              */
```

```
/* 说明：当 MOD 值设置为 0x01 时，如果没有及时*/
/*          喂狗，将会进入这个中断函数。          */
/*****************************************/
void WDT_IRQHandler(void)
{
    WDT->MOD &= ~(0x1<<2);      // 清看门狗超时标志位 WDTOF
    // 在下面可以写入当看门狗中断发生时你想要做的事情
}
/*****************************************/
/* 函数功能：喂狗                          */
/*****************************************/
void WDTFeed(void)
{
    WDT->FEED = 0xAA;       // 喂狗
    WDT->FEED = 0x55;
    return;
}
```

四、程序详解

上面的程序用到的寄存器用法前面已经讲过很多次了，这里就不啰嗦了。现在需要注意的地方是，看门狗可以引起两种结果，一种是复位 LPC1114，另外一种是进入看门狗中断服务函数。复位 LPC1114 的效果和按了复位键是一个效果，内存中的数据都会丢失，一切从新开始。如果设置为进入中断，则可以在意外不喂狗的情况下，进入中断处理。其实，看门狗的本质用途就是当程序跑飞的情况下产生复位。进入中断，只有在不是由于程序跑飞而不喂狗的情况下，才可以发挥作用。

五、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

程序控制一共喂狗 10 次后停止喂狗，由于设置了“不喂狗产生复位”，所以停止喂狗以后，将会产生复位，复位以后，又开始倒计时喂狗。

第九章 PWM 输出

一、入门导读
二、软件设计
三、程序详解
四、实验程序下载和使用说明

一、入门导读

在 LPC1114 上，有四个通用定时器，其中，两个是 16 位定时/计数器，两个是 32 位定时/计数器。可以实现定时、计数、捕获和 PWM 输出等功能。

定时功能和计数功能是大家很熟悉的。

定时的原理是对内部时钟计数；

计数的原理是对外部的时钟计数。

所以，要实现定时的话，需要根据内部时钟大小来设定。计数的话，不关心内部时钟也可以。

捕获功能的主要用途是可以在捕获引脚上发生电平变化时，记录下当时的定时/计数器瞬间值。

PWM 输出可以用在驱动步进电机，或者是可以当做占空比和频率可调的脉冲发生器。两个用途都非常的有用。

每个定时/计数器都有一个对应的捕获输入引脚，所以在 LPC1114 上一共有四个捕获引脚。

一共有 11 个 PWM 输出引脚，其中，CT32B0，CT32B1，CT16B0 各自有 3 个，CT16B1 有 2 个。

下面的提供的程序例程拿 CT32B0 为例，来实现定时，捕获和 PWM 输出功能。

二、程序设计

```
/*****************************/
/* 函数功能：初始化 TIM32B0   */
/*****************************/
void TIM32B0_init(void)
{
    SYSCON->SYSAHBCLKCTRL |= (1<<9); //使能 TIM32B0 时钟
}

/*********************************************************/
/* 函数功能：初始化 TIM32B0                               */
/* 入口参数：ms ：定时毫秒值                              */
/* 说      明：当定时时间到了以后，会进入 TIM32B0 中断函数  */
/*********************************************************/
void TIM32B0_INT_init(uint32 ms)
{
    SYSCON->SYSAHBCLKCTRL |= (1<<9); //使能 TIM32B0 时钟
    TMR32B0->TCR = 0x02;          //复位定时器（bit1：写 1 复位）
    TMR32B0->PR  = 0x00;          //把预分频寄存器置 0，使 PC+1，TC+1
    TMR32B0->MR0 = ms * 48000;  //在 48Mhz 下工作的值，其它请修改
```

```
    TMR32B0->IR    = 0x01;          //MR0 中断复位,即清中断（bit0:MR0, bit1:MR1, bit2:MR2, bit3:MR3, bit4:CP0）
    TMR32B0->MCR = 0x05;          //MR0 中断产生时停止 TC 和 PC，并使 TCR[0]=0，停止定时器工作，并产生中断
    TMR32B0->TCR = 0x01;          //启动定时器：TCR[0]=1;
    NVIC_EnableIRQ(TIMER_32_0_IRQn);     // 使能 TIM32B0 中断
}

/*********************************/
/* 函数功能：TIM32B0 毫秒延时     */
/*********************************/
void TIM32B0_delay_ms(uint32 ms)
{
    TMR32B0->TCR = 0x02;          //复位定时器（bit1：写 1 复位）
    TMR32B0->PR    = 0x00;          //把预分频寄存器置 0，使 PC+1，TC+1
    TMR32B0->MR0 = ms * 48000;  //在 48Mhz 下工作的值，其它请修改
    TMR32B0->IR    = 0x01;          //MR0 中断复位,即清中断（bit0:MR0, bit1:MR1, bit2:MR2, bit3:MR3, bit4:CP0）
    TMR32B0->MCR = 0x04;          //MR0 中断产生时停止 TC 和 PC，并使 TCR[0]=0，停止定时器工作
    TMR32B0->TCR = 0x01;          //启动定时器：TCR[0]=1;

    while (TMR32B0->TCR & 0x01);//等待定时器计时时间到
}

/*********************************/
/* 函数功能：TIM32B0 微秒延时     */
/*********************************/
void TIM32B0_delay_us(uint32 us)
{
    TMR32B0->TCR = 0x02;          //复位定时器（bit1：写 1 复位）
    TMR32B0->PR    = 0x00;          //把预分频寄存器置 0，使 PC+1，TC+1
    TMR32B0->MR0 = us * 48;       //在 48Mhz 下工作的值，其它请修改
    TMR32B0->IR    = 0x01;          //MR0 中断复位（bit0:MR0, bit1:MR1, bit2:MR2, bit3:MR3, bit4:CP0）
    TMR32B0->MCR = 0x04;          //MR0 中断产生时停止 TC 和 PC，并使 TCR[0]=0，停止定时器工作
    TMR32B0->TCR = 0x01;          //启动定时器：TCR[0]=1;

    while (TMR32B0->TCR & 0x01);//等待定时器计时时间到
}
```

```
/*************************************/
/* 函数功能：MAT0 上输出方波信号          */
/* 注      意：这里不是用 PWM 控制寄存器   */
/*************************************/
void TIM32B0_Square(uint32 cycle_us)
{
    SYSCON->SYSAHBCLKCTRL |= (1<<16);  // 使能 IOCON 时钟
    IOCON->PIO1_6 &= ~0x07;
    IOCON->PIO1_6 |= 0x02;    /* Timer0_32 MAT0 */
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);  // 禁能 IOCON 时钟

    TMR32B0->TCR = 0x02;            //复位定时器（bit1：写 1 复位）
    TMR32B0->PR  = 0x00;            //把预分频寄存器置 0，使 PC+1，TC+1
    TMR32B0->MR0 = (cycle_us/2) * 48;//在 48Mhz 下工作的值，其它请修改
    TMR32B0->IR  = 0x01;            //MR0 中断复位（bit0:MR0，bit1:MR1，bit2:MR2,
bit3:MR3, bit4:CP0）
    TMR32B0->MCR = 0x02;            //MR0 中断产生时复位 TC
    TMR32B0->EMR = 0x31;            //MR0 与 PC 相等时，MAT0 引脚翻转电平
    TMR32B0->TCR = 0x01;            //启动定时器：TCR[0]=1;
}

/************************************************/
/* 函数功能：MAT0 上输出占空比可调脉冲信号          */
/* 注      意：这里用 PWM 控制寄存器                */
/************************************************/
void TIM32B0_PWM(uint32 cycle_us, uint8 duty)
{
    if((duty>=100)&&(duty<=0))return;

    SYSCON->SYSAHBCLKCTRL |= (1<<16);  // 使能 IOCON 时钟
    IOCON->PIO1_6 &= ~0x07;
    IOCON->PIO1_6 |= 0x02;            //把 P1.6 脚设置为 MAT0
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);  // 禁能 IOCON 时钟

    TMR32B0->TCR = 0x02;            //复位定时器（bit1：写 1 复位）
    TMR32B0->PR  = 0x00;            //把预分频寄存器置 0，使 PC+1，TC+1
    TMR32B0->PWMC= 0x01;            //设置 MAT0 为 PWM 输出引脚
    TMR32B0->MCR = 0x02<<9;          //设置 MR3 匹配时复位 TC,也就是把 MR3 当做周
期寄存器
    TMR32B0->MR3 = 48*cycle_us;          //设置周期
    TMR32B0->MR0 = 48*cycle_us*(100-duty)/100;//设置占空比
    TMR32B0->TCR = 0x01;                 //启动定时器
}
```

```
/*************************************/
/* 函数功能：利用 CAP0 进行计数                */
/*************************************/
void TIM32B0_CAP0(void)
{
    SYSCON->SYSAHBCLKCTRL |= (1<<16);   // 使能 IOCON 时钟
    IOCON->PIO1_5 &= ~0x07;
    IOCON->PIO1_5 |= 0x02;              //把 P1.5 脚设置为 CAP0
    SYSCON->SYSAHBCLKCTRL &= ~(1<<16);  // 禁能 IOCON 时钟

    TMR32B0->CTCR  = 0x01;          //选择外来信号的上升沿作为 TC 递增，CAP0 捕获
    TMR32B0->TC    = 0x00;            //TC 清零
    TMR32B0->TCR   = 0x01;            //启动定时器
}

/*************************************/
/* 函数功能：TIM32B0z 中断服务函数          */
/*************************************/
void TIMER32_0_IRQHandler(void)
{
    if((TMR32B0->IR & 0x1)==1) // 检测是不是 MR0 引起的中断
    {
        // 在这里写入你想执行的代码
    }
    TMR32B0->IR = 0x1F; // 清所有中断标志
}
```

三、程序详解

第一个函数：里面只有一条语句，用来开启 CT32B0 的时钟，让 CT32B0 可以工作。

第二个函数：这个函数也是开启 CT32B0，让 CT32B0 可以正常工作，不过，这个带有延时的值，二且延时到了以后，会进入 CT32B0 的中断服务函数，即最后一个函数。

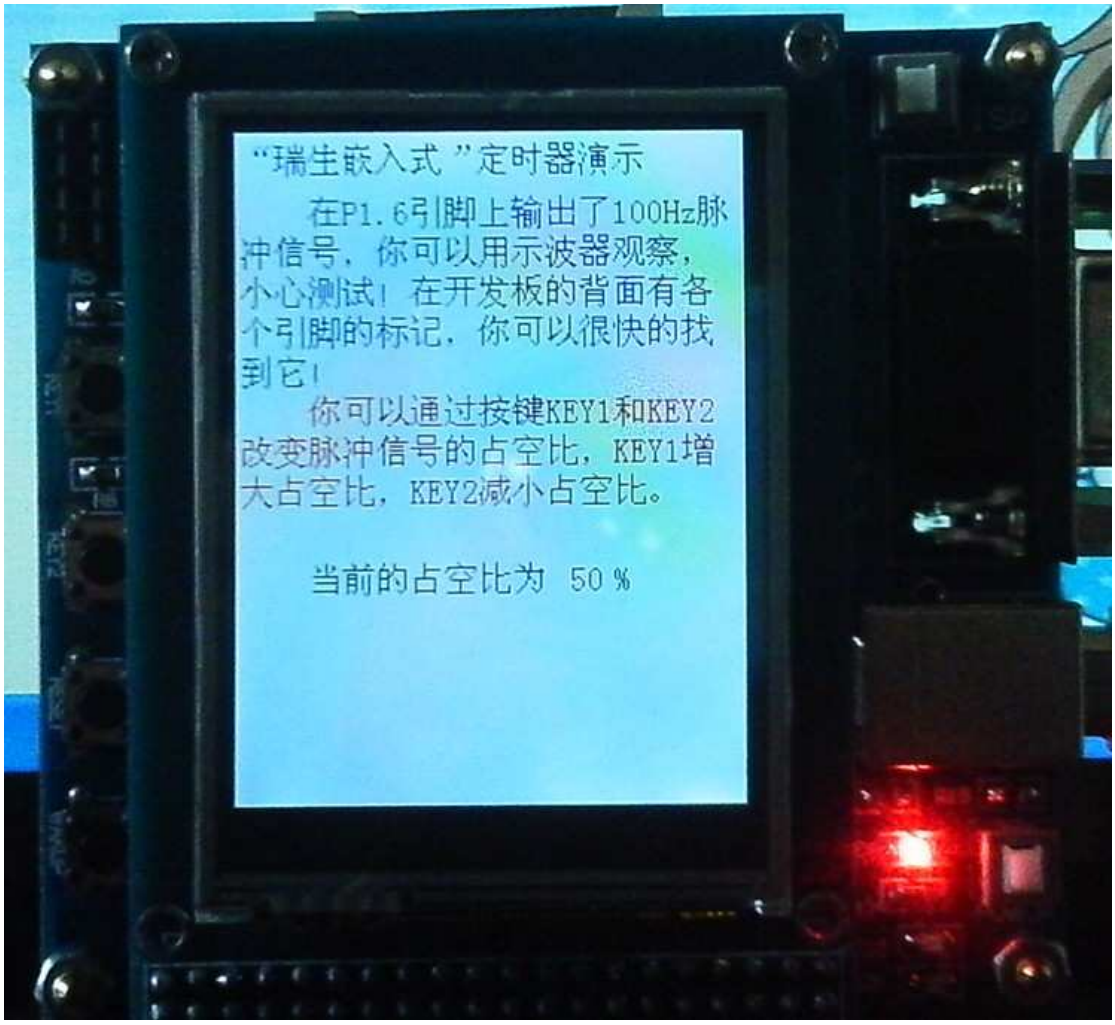
第三个和第四个函数用 CT32B0 实现微秒和毫秒延时，在使用之前，先要执行第一个函数。

第五个和第六个函数，都是实现了在 CT32B0 的 MAT0 引脚上输出脉冲信号，区别是：第五个函数是用定时实现的，输出占空比为 50%的方波信号，频率由入口参数决定。第六个函数是用 PWM 模块实现的，占空比和频率都有入口参数决定。

第七个函数是捕获计数功能。在 CAP0 引脚，即 P1.5 脚上，每发生一个上升沿，TC 值就会加 1。

四、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。画面效果如下图所示：



实验程序使用的是 PWM 输出功能。在 P1.6 引脚上输出了 100Hz 脉冲信号，你可以用示波器观察，小心测试！在开发板的背面有各个引脚的标记，你可以很快的找到它！

你可以通过按键 KEY1 和 KEY2 改变脉冲信号的占空比，KEY1 增大占空比，KEY2 减小占空比。

第十章 WAKUP 与深度掉电模式

一、入门引导
二、程序设计
三、程序详解
四、实验程序下载和使用说明

一、入门引导

LPC1114 有三种低功耗模式，睡眠模式，深度睡眠模式，和深度掉电模式。在睡眠模式下，可以通过复位引脚以及被设置位唤醒引脚的引脚唤醒。在深度掉电模式下，只能通过 WAKUP 引脚（即 P1.4 引脚）唤醒。

二、程序设计

```
void PMU_PowerDown(void)
{
    SCB->SCR |= 0x4;          // 选择“深度掉电”低功耗模式（注意：系统默认“睡眠”低功耗模式）
    PMU->PCON = (0x1<<11);   // 清除深度掉电模式
    PMU->PCON = 0x2;          // DPDEN=1;
    __wfi();                  // 写 wfi 指令进入深度掉电模式

    return;
}
```

三、程序详解

上面这个函数看起来很爽！因为只有四条语句!

第一条语句对 SCB->SCR 寄存器设置。这个寄存器是 Cortex-M0 内核寄存器，是系统控制模块当中的系统控制寄存器。在这里实现的功能是，把单片机的低功耗模式选择为“深度掉电模式”，系统默认的低功耗模式是“睡眠模式”，所以，如果要想进入深度掉电模式，必须设置该位。

第二条语句是对 LPC1114 的功耗管理单元当中的电源控制寄存器设置。清除标志后，开启。

第三条语句是一个 Thumb 指令。执行完这句话后，才可以进入低功耗模式。

去掉上面函数的第一条语句，执行完这个函数，将进入睡眠模式，而不是深度掉电模式，

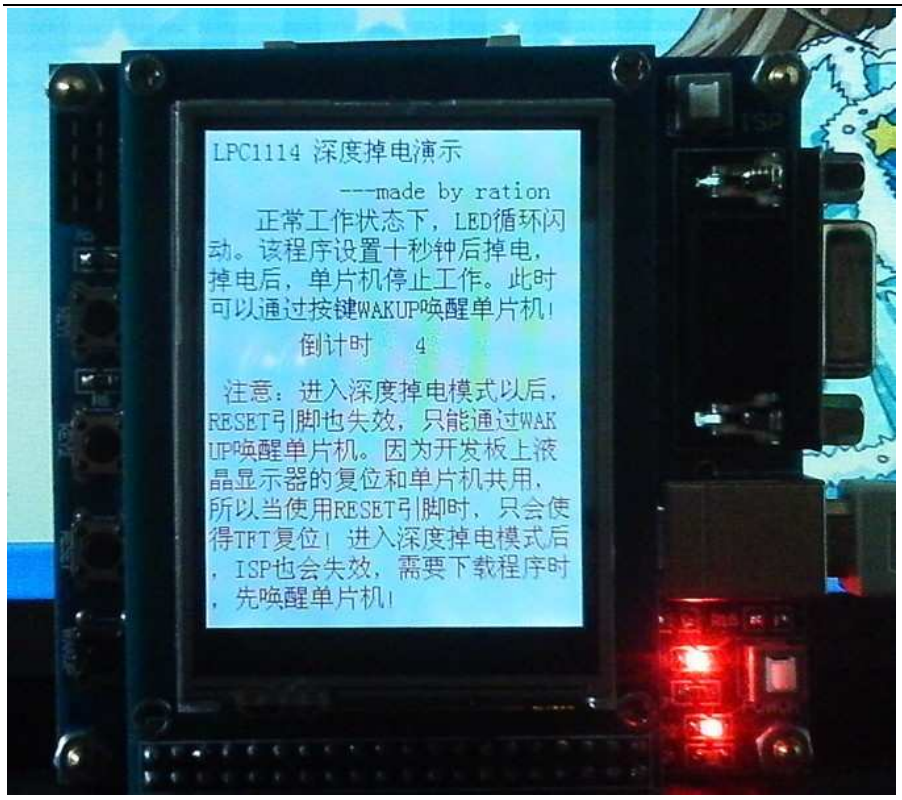
在进入睡眠模式的情况下，可以通过 RESET 引脚复位唤醒，WAKUP 引脚不起作用；

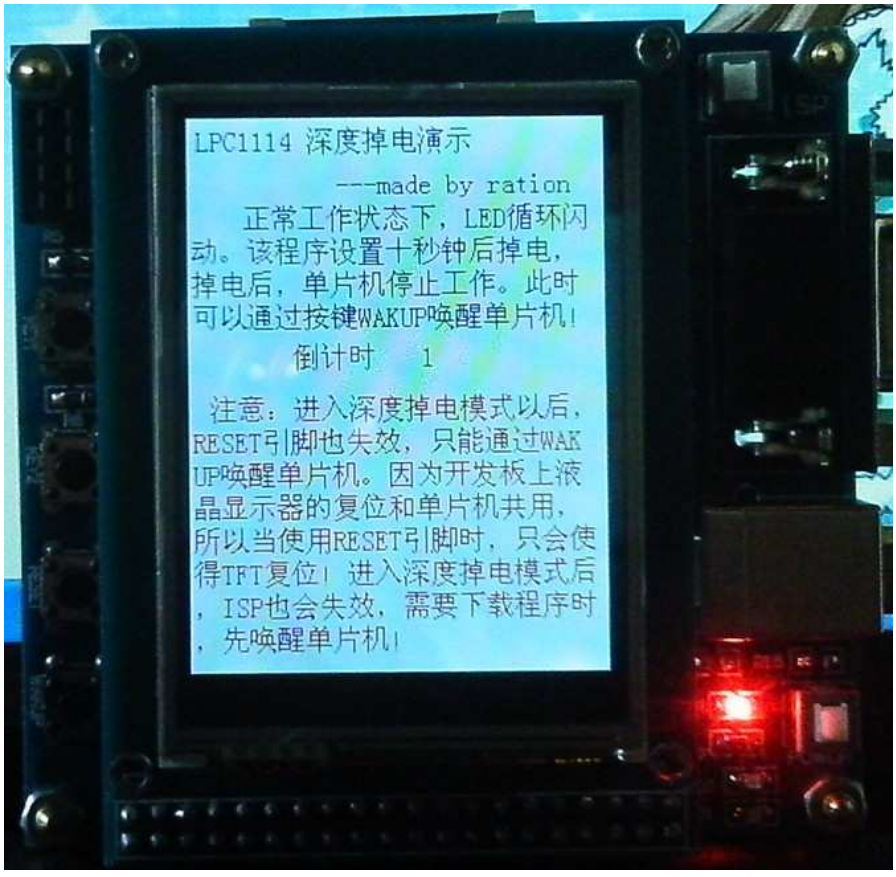
在进入深度掉电模式时，可以通过 WAKUP 引脚唤醒，RESET 引脚不起作用；

注 意：在使用这个函数之前，最好先干点别的事情，当单片机一上电就进入低功耗模式的话，下载程序用的模块也会失效，所以你会发现再也下载不进去程序了，这时，不用着急！先关掉电源，在把 P0.1 脚与 GND 连接好之后，上电，这时，便可以重新下载程序了，下载完以后，关掉电源，把 P0.1 引脚和 GND 断开，再上电，单片机即会开始正常工作。

四、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。画面效果如下图所示：





在正常工作状态下，LED 灯会循环闪亮，当单片机进入深度掉电模式后，将停止工作，程序设置在 10 秒钟之后进入低功耗模式。

第十一章 读写 SD 卡

一、入门引导
二、硬件连接
三、程序设计
四、程序详解
五、实验程序下载和使用说明

一、入门引导

鉴于 SD 应用非常广泛。而且容易受单片机的控制，所以被用在工控领域或者消费电子领域采集和收集数据，是必然的！

SD 卡有两种通信接口，分别是 SD 模式和 SPI 模式。

SD 模式的读写速度要比 SPI 模式快很多。

现在新出的很多单片机都集成了 SD 接口，可惜 LPC1114 只有 SPI 口，所以，我们只能选择用 SPI 通信模式读写 SD 卡了。

SD 卡其实并不神秘，你可以把它当做一个普通的芯片。它和其它的芯片一样，有自己的命令，自己的寄存器。

下面，列出控制 SD 卡的函数！想深入了解 SD 卡的朋友，可以去购买相关书籍。它有太多的内容。这里就不讲了。

声明：以下程序是参照正点原子的程序修改后再 LPC1114 上实现的。在这里表示感谢！

二、硬件连接

SD_CS----------P1.5
其它 SPI 控制接口接 LPC1114 SSP0 接口

三、程序设计

```
//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
uint8 SD_GetResponse(uint8 Response)
{
    uint16 Count=0xFFF;//等待次数
    while  ((SPI0_communication(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
}

//向 SD 卡发送一个命令
//输入: uint8 cmd      命令
//          uint32 arg    命令参数
//          uint8 crc     crc 校验值
//返回值:SD 卡返回的响应

uint8 SD_SendCommand(uint8 cmd, uint32 arg, uint8 crc)
{
     uint8 r1;
```

```
    uint8 repeat=0;
    SD_CS_High;
     SPI0_communication(0xff);//高速写命令延时
    SPI0_communication(0xff);
    SPI0_communication(0xff);
    //片选端置低，选中 SD 卡
     SD_CS_Low;
    //发送
     SPI0_communication(cmd | 0x40);//分别写入命令
     SPI0_communication(arg >> 24);
     SPI0_communication(arg >> 16);
     SPI0_communication(arg >> 8);
     SPI0_communication(arg);
     SPI0_communication(crc);
    //等待响应，或超时退出
     while((r1=SPI0_communication(0xFF))==0xFF)
    {
        repeat++;
        if(repeat>200)break;
    }
    //关闭片选
    SD_CS_High;
    //在总线上额外增加 8 个时钟，让 SD 卡完成剩下的工作
    SPI0_communication(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:uint8 cmd      命令
//        uint32 arg    命令参数
//        uint8 crc      crc 校验值
//返回值:SD 卡返回的响应

uint8 SD_SendCommand_NoDeassert(uint8 cmd, uint32 arg, uint8 crc)
{
    uint8 repeat=0;
    uint8 r1;
     SPI0_communication(0xff);//高速写命令延时
    SPI0_communication(0xff);
     SD_CS_Low;//片选端置低，选中 SD 卡
    //发送
     SPI0_communication(cmd | 0x40); //分别写入命令
```

```
    SPI0_communication(arg >> 24);
    SPI0_communication(arg >> 16);
    SPI0_communication(arg >> 8);
    SPI0_communication(arg);
    SPI0_communication(crc);
    //等待响应，或超时退出
    while((r1=SPI0_communication(0xFF))==0xFF)
    {
        repeat++;
        if(repeat>200)break;
    }
    //返回响应值
    return r1;
}
/*********************************************/

/* 功      能：初始化 SD 卡                    */
/* 返回结果:  0：NO_ERR                       */
/*            1：TIME_OUT                     */
/*            99：NO_CARD                      */
/*********************************************/

uint8 SD_Init(void)
{
    uint8 r1,i;         // 存放 SD 卡的返回值
    uint16 repeat=0;    // 用来进行超时计数
    uint8 buff[6];

    /*-------  配置控制 SD 卡的引脚 ----------*/
    GPIO1->DIR |= (1<<5);   // P1.5 设置为输出，用作 SD_CS
    GPIO1->DATA |= (1<<5); // SD_CS = 1;
#ifndef SSP0INIT            // 如果没有执行过初始化 SSP0，初始化 SSP0
    SPI0_Init();            // 初始化 SPI0
#define SSP0INIT            // 标记 SSP0 执行过初始化（这个条件编译是为了多个外围芯片共用 SPI 口）
#endif
    SSP0->CPSR = 0xFE;          // 把 SPI0 时钟设置为最低速（初始化 SD 卡的时钟频率不要超过 500K）
    SD_CS_High;

    /*-------  初始化 SD 卡到 SPI 模式  -------*/
    //先产生>74 个脉冲，让 SD 卡自己初始化完成
    for(i=0;i<12;i++)SPI0_communication(0xFF);
```

```
do
 {
      i = SD_SendCommand(CMD0, 0, 0x95);
      repeat++;
 }while((i!=0x01)&&(repeat<200)); // 最大发 200 次 CMD0 命令
 if(repeat==200)return 1; // 失败
repeat = 0;                    // 恢复 repeat 值

/*--------  初始化 SD 卡  --------*/
 //获取卡片的 SD 版本信息
SD_CS_Low;
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_High;
      //多发 8 个 CLK，让 SD 结束后续操作
      SPI0_communication(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
      repeat = 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);
           repeat++;
      }while((r1!=0x00) && (repeat<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(repeat==400)
      {
           repeat = 0;
```

```
            //发送 MMC 卡初始化命令（没有测试）
            do
            {
                r1 = SD_SendCommand(1,0,0);
                repeat++;
            }while((r1!=0x00)&& (repeat<400));
            if(repeat==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
        SSP0->CPSR = 0x04;   //设置 SPI0 为高速模式
      SPI0_communication(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] = SPI0_communication(0xFF);    //should be 0x00
        buff[1] = SPI0_communication(0xFF);    //should be 0x00
        buff[2] = SPI0_communication(0xFF);    //should be 0x01
        buff[3] = SPI0_communication(0xFF);    //should be 0xAA
        SD_CS_High;
        SPI0_communication(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
        {
            repeat = 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(repeat>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_High;//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] = SPI0_communication(0xFF);
                buff[1] = SPI0_communication(0xFF);
                buff[2] = SPI0_communication(0xFF);
                buff[3] = SPI0_communication(0xFF);
                //OCR 接收完成，片选置高
                SD_CS_High;
                SPI0_communication(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-----------

                SSP0->CPSR = 0x04;   //设置 SPI0 为高速模式
            }
        }
        return r1;
}

//从 SD 卡中读回指定长度的数据，放置在给定位置
//输入: uint8 *data(存放读回数据的内存>len)
//          uint16 len(数据长度）
//          uint8  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
uint8 SD_ReceiveData(uint8 *data, uint16 len, uint8 release)
{
      // 启动一次传输
      SD_CS_Low;
```

```
    if(SD_GetResponse(0xFE))//等待 SD 卡发回数据起始令牌 0xFE
    {
        SD_CS_High;
        return 1;
    }
    while(len--)//开始接收数据
    {
        *data=SPI0_communication(0xFF);
        data++;
    }
    //下面是 2 个伪 CRC（dummy CRC）
    SPI0_communication(0xFF);
    SPI0_communication(0xFF);
    if(release==RELEASE)//按需释放总线，将 CS 置高
    {
        SD_CS_High;//传输结束
        SPI0_communication(0xFF);
    }
    return 0;
}

//获取 SD 卡的 CID 信息，包括制造商信息
//输入: uint8 *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

uint8 SD_GetCID(uint8 *cid_data)
{
    uint8 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:uint8 *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
```

```
uint8 SD_GetCSD(uint8 *csd_data)
{
    uint8 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)

uint32 SD_GetCapacity(void)
{
    uint8 csd[16];
    uint32 Capacity;
    uint16 n;
   uint16 csize;
   //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 = ((uint32)csd[8])<<8;
        Capacity += (uint32)csd[9]+1;
         Capacity = (Capacity)*1024;//得到扇区数
        Capacity *= 512;//得到字节数
    }
    else
    {
        n = (csd[5] & 0x0F) + ((csd[10] & 0x80) >> 7) + ((csd[9] & 0x03) << 1) + 2;
        csize = (csd[8] >> 6) + ((uint16)csd[7] << 2) + ((uint16)(csd[6] & 0x03) <<
10) + 1;
        Capacity = (uint32)csize << (n - 9);
        Capacity *= 512;
    }
    return (uint32)Capacity;
}

//读 SD 卡的一个 block
//输入:uint32 sector  取地址（sector 值，非物理地址）
//      uint8 *buffer  数据存储地址（大小至少 512byte）
//返回值:0：  成功
```

```
//          other：失败

uint8 SD_ReadSingleBlock(uint32 sector, uint8 *buffer)
{
    uint8 r1;

    SSP0->CPSR = 0x04;   //设置 SPI 为高速模式 24MHz
    //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;
}

//////////////////////////////////////////////////////////////////////////
//写入 SD 卡的一个 block(未实际测试过)
//输入:uint32 sector  扇区地址（sector 值，非物理地址）
//      uint8 *buffer  数据存储地址（大小至少 512byte）
//返回值:0：  成功
//          other：失败

uint8 SD_WriteSingleBlock(uint32 sector, const uint8 *data)
{
    uint8 r1;
    uint16 i;
    uint16 repeat;

    //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_Low;
//先放 3 个空数据，等待 SD 卡准备好
SPI0_communication(0xff);
SPI0_communication(0xff);
SPI0_communication(0xff);
//放起始令牌 0xFE
SPI0_communication(0xFE);

//放一个 sector 的数据
for(i=0;i<512;i++)
{
    SPI0_communication(*data++);
}
//发 2 个 Byte 的 dummy CRC
SPI0_communication(0xff);
SPI0_communication(0xff);

//等待 SD 卡应答
r1 = SPI0_communication(0xff);
if((r1&0x1F)!=0x05)
{
    SD_CS_High;
    return r1;
}

//等待操作完成
repeat = 0;
while(!SPI0_communication(0xff))
{
    repeat++;
    if(repeat>0xfffe)          //如果长时间写入没有完成，报错退出
    {
        SD_CS_High;
        return 1;              //写入超时返回 1
    }
}
//写入完成，片选置 1
SD_CS_High;
SPI0_communication(0xff);
```

```
    return 0;
}



//读 SD 卡的多个 block(实际测试过)
//输入:uint32 sector  扇区地址（sector 值，非物理地址）
//      uint8 *buffer  数据存储地址（大小至少 512byte）
//      uint8 count  连续读 count 个 block
//返回值:0：  成功
//          other：失败

uint8 SD_ReadMultiBlock(uint32 sector, uint8 *buffer, uint8 count)
{
    uint8 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, 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_High;
    SPI0_communication(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:uint32 sector  扇区地址（sector 值，非物理地址）
//      uint8 *buffer  数据存储地址（大小至少 512byte）
//      uint8 count  写入的 block 数目
//返回值:0：  成功
//          other：失败
```

```
uint8 SD_WriteMultiBlock(uint32 sector, const uint8 *data, uint8 count)
{
    uint8 r1;
    uint16 i;
    uint16 repeat;

    //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(CMD25, 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_Low;
    //先放 3 个空数据，等待 SD 卡准备好
    SPI0_communication(0xff);
    SPI0_communication(0xff);
//      SPI0_communication(0xff);

    do{
        //放起始令牌 0xFE
        SPI0_communication(0xFC);

        //放一个 sector 的数据
        for(i=0;i<512;i++)
        {
            SPI0_communication(*data++);
        }
        //发 2 个 Byte 的 dummy CRC
        SPI0_communication(0xff);
        SPI0_communication(0xff);

        //等待 SD 卡应答
        r1 = SPI0_communication(0xff);

        if((r1&0x1F)!=0x05)
```

```
        {
            SD_CS_High;
            return r1;
        }

        //等待操作完成
        repeat = 0;
        while(!SPI0_communication(0xff))
        {
            repeat++;
            if(repeat>0xfffe)            //如果长时间写入没有完成，报错退出
            {
                SD_CS_High;
                return 1;                //写入超时返回 1
            }
        }
    }while(--count);

    //写入完成，片选置 1
    r1 = SPI0_communication(0xFD);
    SD_CS_High;
    SPI0_communication(0xff);
    delay_ms(6);   // 写完多个块以后，延时稳定！否则不能从 SD 卡读数据！
    return 0;
}
```

四、程序详解

把以上程序的声明写到这里：

```
uint8 SD_SendCommand(uint8 cmd, uint32 arg, uint8 crc);    //SD 卡发送一个命令
uint8 SD_SendCommand_NoDeassert(uint8 cmd, uint32 arg, uint8 crc);   //SD 卡发送一个命令(不拉高片选引脚)
uint8 SD_Init(void);                                                  //SD 卡初始化
uint8 SD_ReceiveData(uint8 *data, uint16 len, uint8 release);         //SD 卡读数据
uint8 SD_GetCID(uint8 *cid_data);                                     //读 SD 卡 CID
uint8 SD_GetCSD(uint8 *csd_data);                                     //读 SD 卡 CSD
uint32 SD_GetCapacity(void);                                          //取 SD 卡容量
uint8 SD_ReadSingleBlock(uint32 sector, uint8 *buffer);               //读一个 BLOCK
uint8 SD_WriteSingleBlock(uint32 sector, const uint8 *buffer);   //写一个 BLOCK
uint8 SD_ReadMultiBlock(uint32 sector, uint8 *buffer, uint8 count);// 读多个 BLOCK
uint8 SD_WriteMultiBlock(uint32 sector, const uint8 *data, uint8 count);//写多个 BLOCK
```

上面的第一个第二个函数是给 SD 卡写命令的。区别是第一个函数执行完以后，把 CS 片选引脚拉高;第二个函数不老高 CS 片选引脚。

第三个函数是初始化 SD 卡的函数。执行完这个函数，SD 卡就可以读写了。返回 0，说明初始化成功。返回 1，初始化失败。

第四个函数，用于读 SD 卡中的数据。

第五个和第六个函数，用于读取 SD 卡的 CID 和 CSD 寄存器值。其中，CSD 寄存器包含很多关于 SD 卡的信息。包括存储容量、SD 卡版本信息等。

第七个函数，用来获取 SD 卡的容量。返回值的单位是字节，我们平常习惯用 M 字节来表示，所以读出以后需要把字节换成 M 字节。注意，这里读出的数是用户可以使用的容量，而不是整体容量。比如 2G 的 SD 卡，读出的数据一般为 1886Mb。

剩下的四个函数，即是读写 SD 卡的单扇区函数和多扇区函数。当直接使用此函数的时候，SD 卡相当于一个存储芯片。这时候，写进入的值不可以被电脑读出。只能通过读取函数读出。因为没有建立文件系统。在后面学习文件系统的时候，可以学到。在建立文件系统以后，文件系统也是调用这四个函数来读取 SD 卡的。

五、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。

注意：在没有建立文件系统的情况系读写 SD 卡的时候，最好先把你的 SD 卡中的东西拷贝出来，因为你写进去的数据有可能会破坏上面的数据。

第十二章 NRF24L01 无线通信模块

一、入门引导

NRF24L01 是一款很流行的无线通信收发芯片。有很多无线鼠标、无线键盘就是用这个芯片做无线收发的。它工作在 2.4GHz~2.5GHz 世界通用 ISM 工作频段。

用了这么多的各种类型芯片，你会发现操作这些芯片都有一个共同的特点，就是对它的寄存器操作，达到控制的作用。要对它的寄存器操作，就需要知道 MCU 与芯片连接需要采用的接口方式。

我们想运用 NRF24L01，就需要先学习它的寄存器和接口方式了。不过，要给 NRF24L01 的寄存器写值或读值，还需要使用特殊的命令。

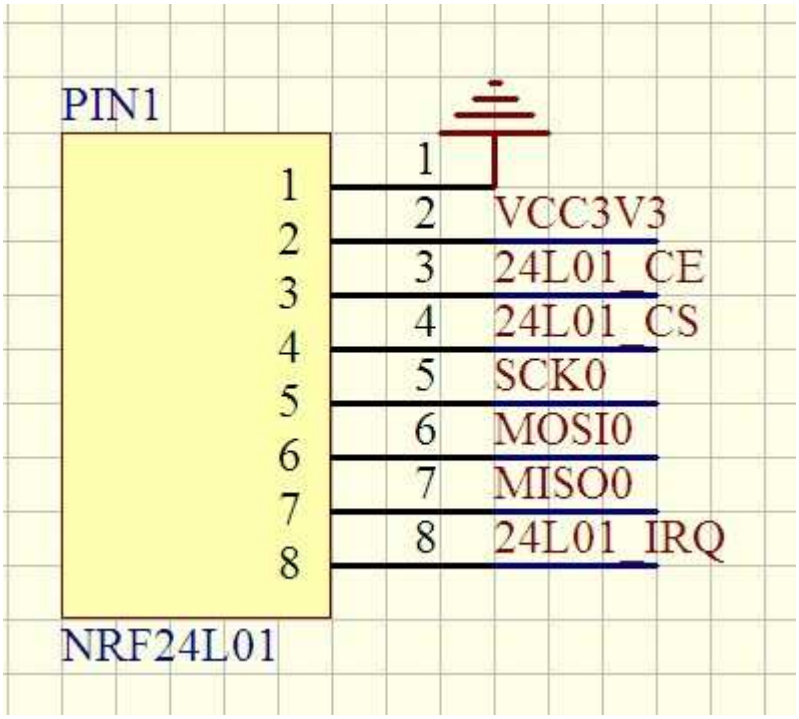
一、接口方式：SPI。

二、命令：8 个。

三、寄存器：一共 23 个寄存器。

二、硬件连接

在我们的开发板上，用 LPC1114 的 SSP0 与 NRF24L01 连接。



24L01_CE----------P3.5
24L01_CS----------P0.3
24L01_IRQ----------P3.4

三、NRF24L01 初始化程序设计

```
/*******************************************/
/* 函数功能：NRF24L01 初始化      */
```

```
/*******************************************/
void NRF24L01_Init()
{
    SPI0_Init();
    GPIO0->DIR |= (1<<3);    //P0.3 脚为输出，用做 CSN
    GPIO0->DATA |= (1<<3);  //CSN=1;
    GPIO3->DIR &= ~(1<<4);   //NRF24L01_IRQ 连接 P3.4 脚，设置 P3.4 脚为输入引脚
    GPIO3->DIR |= (1<<5);      //NRF24L01_CE 连接 P3.5 脚，设置 P3.5 脚为输出引脚
    GPIO3->DATA &= ~(1<<5); //CE 置低，使能 24L01
}
```

四、NRF24L01 初始化程序详解：

上面这个函数即是 LPC1114 与 NRF24L01 硬件接口初始化函数。

首先，执行 SPI0_Init();把 LPC1114 的 SSP0 设置为 SPI0，只是设置了 MOSI0、MISO0、SCK0 引脚，没有设置 SSEL0 引脚。

函数中下面的几条语句是：

P0.3 脚设置为 NRF24L01_CSN 引脚；SPI 片选使能引脚。

P3.4 脚设置为 NRF24L01_IRQ 引脚；中断引脚。

P3.5 脚设置为 NRF24L01_CE 引脚。芯片使能引脚。

五、NRF24L01 命令

```
/**********  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
```

六、NRF24L01 寄存器

```
/**********  NRF24L01 寄存器地址  **************/
#define CONFIG          0x00  //配置寄存器地址
#define EN_AA           0x01  //使能自动应答功能
#define EN_RXADDR       0x02  //接收地址允许
#define SETUP_AW        0x03  //设置地址宽度(所有数据通道)
#define SETUP_RETR      0x04  //建立自动重发
#define RF_CH           0x05  //RF 通道
#define RF_SETUP        0x06  //RF 寄存器
#define STATUS          0x07  //状态寄存器
#define OBSERVE_TX      0x08  // 发送检测寄存器
#define CD              0x09  // 载波检测寄存器
```

```
#define RX_ADDR_P0      0x0A   // 数据通道 0 接收地址
#define RX_ADDR_P1      0x0B   // 数据通道 1 接收地址
#define RX_ADDR_P2      0x0C   // 数据通道 2 接收地址
#define RX_ADDR_P3      0x0D   // 数据通道 3 接收地址
#define RX_ADDR_P4      0x0E   // 数据通道 4 接收地址
#define RX_ADDR_P5      0x0F   // 数据通道 5 接收地址
#define TX_ADDR         0x10   // 发送地址寄存器
#define RX_PW_P0        0x11   // 接收数据通道 0 有效数据宽度(1~32 字节)
#define RX_PW_P1        0x12   // 接收数据通道 1 有效数据宽度(1~32 字节)
#define RX_PW_P2        0x13   // 接收数据通道 2 有效数据宽度(1~32 字节)
#define RX_PW_P3        0x14   // 接收数据通道 3 有效数据宽度(1~32 字节)
#define RX_PW_P4        0x15   // 接收数据通道 4 有效数据宽度(1~32 字节)
#define RX_PW_P5        0x16   // 接收数据通道 5 有效数据宽度(1~32 字节)
#define FIFO_STATUS     0x17   // FIFO 状态寄存器
/*————————————————————————————————————————*/
```

七、NRF24L01 配置程序设计及程序详解

上面已经讲过了 NRF24L01 的初始化函数，执行完初始化函数，就可以开始配置 NRF24L01 了。要配置 NRF24L01，就需要给 NRF24L01 的寄存器写值或从 NRF24L01 的寄存器里面写值。所以，首先，让我们学习一下 NRF24L01 的寄存器读写值函数。在我们的例程当中，一共有 4 个 NRF24L01 读写值函数，它们分别是：

```
uint8 NRF24L01_Write_Reg(uint8 reg,uint8 value);
uint8 NRF24L01_Read_Reg(uint8 reg);
uint8 NRF24L01_Write_Buf(uint8 reg, uint8 *pBuf, uint8 len);
uint8 NRF24L01_Read_Buf(uint8 reg,uint8 *pBuf,uint8 len);
```

注意：NRF24L01 的 23 个寄存器当中，有 3 个寄存器是 40 位的，其余是 8 位的。这 3 个 40 位的寄存器是：

TX_ADDR	发送地址寄存器
RX_ADDR_P0	接收地址通道 0 寄存器
RX_ADDR_P1	接收地址通道 1 寄存器

因为发送地址和接收地址可以是 5 个字节。

上面列出的函数当中，前两个是读写 8 位寄存器的，后两个是专门用来读写 40 位寄存器的。

有了上面的函数，就可以配置 NRF24L01 了，首先为了正确配置 24L01，就需要先判断一下 24L01 是否存在。执行下面的函数：

```
/*********************************************/
/* 函数功能：检测 24L01 是否存在                */
/* 返回值；   0   存在                         */
```

```
/*              1  不存在                       */
/*********************************************/
uint8 NRF24L01_Check(void)
{
    uint8 check_in_buf[5]={0x11,0x22,0x33,0x44,0x55};
    uint8 check_out_buf[5]={0x00};

    NRF24L01_Write_Buf(WRITE_REG+TX_ADDR, check_in_buf, 5);
    NRF24L01_Read_Buf(READ_REG+TX_ADDR, check_out_buf, 5);
    if((check_out_buf[0] == 0x11)&&\
       (check_out_buf[1] == 0x22)&&\
       (check_out_buf[2] == 0x33)&&\
       (check_out_buf[3] == 0x44)&&\
       (check_out_buf[4] == 0x55))return 0;
    else return 1;
}
```

这个函数的检测原理是给 NRF24L01 的寄存器 TX_ADDR 写 5 个字节，然后再读出来一一对比，看前后的值是否一致。

如果检测到了 NRF24L01，就可以配置它了。这时候，就可以把它配置为发送模式或是接收模式了。

```
/*********************************************/
/* 函数功能：设置 24L01 为接收模式  */
/*********************************************/
void NRF24L01_RX_Mode(void)
{
    //CE 拉低，使能 24L01 配置
    GPIO3->DATA &= ~(1<<5);
    //写 RX 接收地址
    NRF24L01_Write_Buf(WRITE_REG+RX_ADDR_P0,(uint8*)RX_ADDRESS,RX_ADR_WIDTH);
    NRF24L01_Write_Reg(WRITE_REG+EN_AA,0x01);     //开启通道 0 自动应答
    NRF24L01_Write_Reg(WRITE_REG+EN_RXADDR,0x01);//通道 0 接收允许
    NRF24L01_Write_Reg(WRITE_REG+RF_CH,40);          //设置 RF 工作通道频率
    //选择通道 0 的有效数据宽度
    NRF24L01_Write_Reg(WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH);
    //设置 TX 发射参数,0db 增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG+RF_SETUP,0x0f);
    //配置基本工作模式的参数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
    NRF24L01_Write_Reg(WRITE_REG+CONFIG,0x0f);
    NRF24L01_Write_Reg(FLUSH_RX,0xff);//清除 RX FIFO 寄存器
    GPIO3->DATA |=  (1<<5);     //CE 置高，使能接收
}
```

```
/**********************************************/
/* 函数功能：设置 24L01 为发送模式  */
/**********************************************/
void NRF24L01_TX_Mode(void)
{
    GPIO3->DATA &= ~(1<<5);    //CE 拉低，使能 24L01 配置
    //写 TX 节点地址
    NRF24L01_Write_Buf(WRITE_REG+TX_ADDR,(uint8*)TX_ADDRESS,TX_ADR_WIDTH);
    //设置 TX 节点地址,主要为了使能 ACK
    NRF24L01_Write_Buf(WRITE_REG+RX_ADDR_P0,(uint8*)RX_ADDRESS,RX_ADR_WIDTH);
    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
    //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+SETUP_RETR,0x1a);
    NRF24L01_Write_Reg(WRITE_REG+RF_CH,40);         //设置 RF 通道为 40
    //设置 TX 发射参数,0db 增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(WRITE_REG+RF_SETUP,0x0f);
    //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_Write_Reg(WRITE_REG+CONFIG,0x0e);
    GPIO3->DATA |=  (1<<5);    //CE 置高，使能发送
}
```

看到上面的函数，你可以发现，要给某个寄存器写值，需要用到写命令 WRITE_REG+寄存器地址。同样，如果需要读寄存器的值时，需要在寄存器地址前面加上读命令 READ_REG，不过，由于 READ_REG=0x00，所以，在读寄存器时，可以不必加 READ_REG。

模式配置完成，就可以用下面的函数发送或接收一包数据了：

```
/**********************************************/
/* 函数功能：24L01 接收数据          */
/* 入口参数：rxbuf 接收数据数组      */
/* 返回值：  0   成功收到数据        */
/*           1   没有收到数据        */
/**********************************************/
uint8 NRF24L01_RxPacket(uint8 *rxbuf)
{
    uint8 state;

    state=NRF24L01_Read_Reg(STATUS);  //读取状态寄存器的值
    //清除 TX_DS 或 MAX_RT 中断标志
    NRF24L01_Write_Reg(WRITE_REG+STATUS,state);if(state&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
}
/************************************************************/
/* 函数功能：设置 24L01 为发送模式                          */
/* 入口参数：txbuf   发送数据数组                           */
/* 返回值；  0x10     达到最大重发次数，发送失败            */
/*           0x20     成功发送完成                          */
/*           0xff     发送失败                              */
/************************************************************/
uint8 NRF24L01_TxPacket(uint8 *txbuf)
{
        uint8 state;

        GPIO3->DATA &= ~(1<<5);     //CE 拉低，使能 24L01 配置
        //写数据到 TX BUF   32 个字节
        NRF24L01_Write_Buf(WR_TX_PLOAD,txbuf,TX_PLOAD_WIDTH);
        GPIO3->DATA |=   (1<<5);//CE 置高，使能发送
        while((GPIO3->DATA&(1<<4))==(1<<4));//等待发送完成
        state=NRF24L01_Read_Reg(STATUS);   //读取状态寄存器的值
        //清除 TX_DS 或 MAX_RT 中断标志
        NRF24L01_Write_Reg(WRITE_REG+STATUS,state);if(state&MAX_TX)
        {
                NRF24L01_Write_Reg(FLUSH_TX,0xff);//清除 TX FIFO 寄存器
                return MAX_TX;
        }
        if(state&TX_OK)//发送完成
        {
                return TX_OK;
        }
        return 0xff;//发送失败
}
```

NRF24L01 的基础函数就都有了，现在就可以利用上面的函数实现通信了。

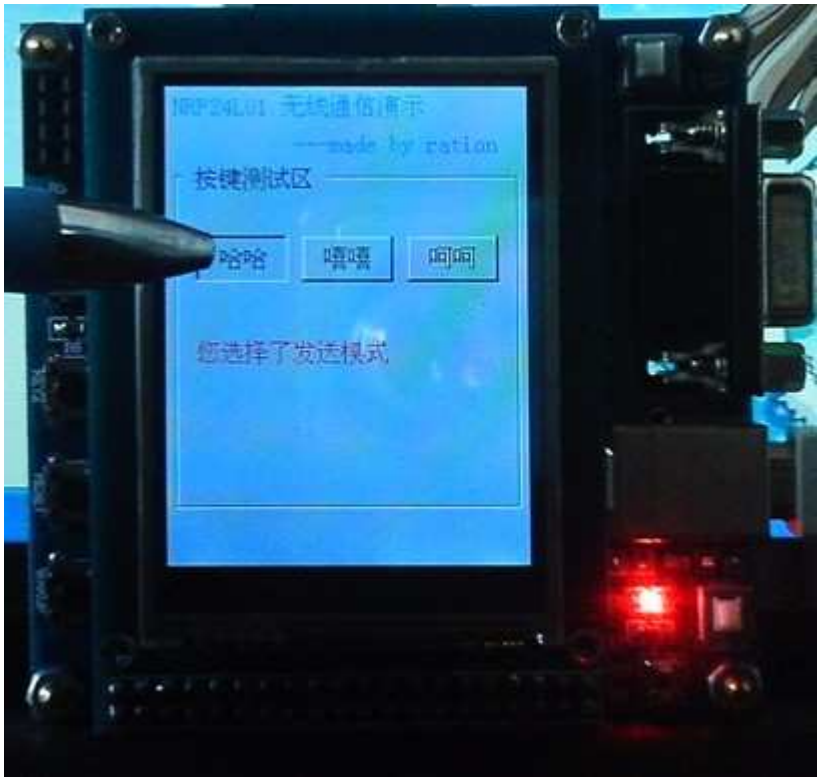
八、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。画面效果如下图所示：

按照画面上的提示，先通过按键 KEY1 和 KEY2 键选择发送或者接收模式。此功能需要两个开发板配合。一个设置为接收模式，一个设置为发送模式。

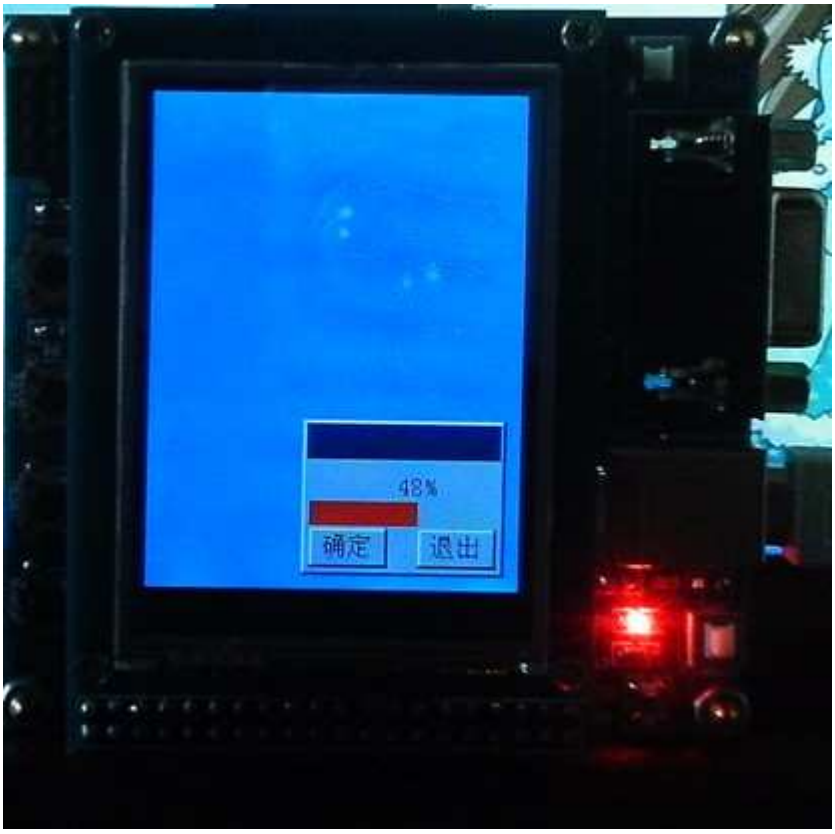
设置好以后，主机（发送模式的）将会控制从机（接收模式的）的按钮。

当按下主机的任意一个按钮后，从机的对应按钮也会被按下。如下图所示。


---made by ration
按键测试区
哈哈
嘻嘻
呵呵
您选择了发送模式

第十三章 显示器 GUI 的实现

一、入门引导
二、程序设计
三、程序详解
四、实验程序下载和使用说明

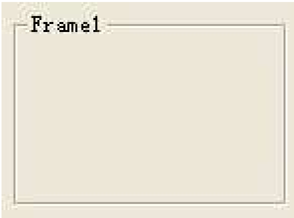


一、入门引导

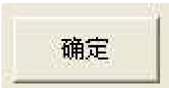
GUI，即 Graphical User Interface，图形用户接口。良好的图形界面可以增加用户的可操作性，而且给人的印象比较好，看看 DOS 和 WINDOWS 就清楚了。

一般的 GUI 包括一下几项内容，这些都是最基本的。

①Frame

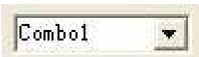


②Button



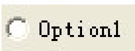
③TextBox

④Combol



⑤Check

⑥Option



等等。。。

二、程序设计

这些 GUI 的原理非常简单，无非就是图形的显示而已。我在 gui.c 中做了一些 GUI 图形，包括如下：

```
/***********************************/
/*  函数功能：显示圆点 5*5              */
/***********************************/
void LCD_Draw5Point(uint16 x, uint16 y, uint16 color)
{
    POINT_COLOR=color;
    LCD_DrawPoint(x-1,y-2);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
```

```
    LCD_DrawPoint(x-2,y-1);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_DrawPoint(x-2,y);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_DrawPoint(x-2,y+1);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_DrawPoint(x-1,y+2);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);

}


/*********************************/
/* 函数功能：显示圆点 9*9          */
/*********************************/
void LCD_Draw9Point(uint16 x, uint16 y, uint16 color)
{
    POINT_COLOR=color;
    LCD_DrawPoint(x-1,y-4);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_DrawPoint(x-4,y-1);
    LCD_DrawPoint(x-4,y);
    LCD_DrawPoint(x-4,y+1);
    LCD_DrawPoint(x+4,y-1);
    LCD_DrawPoint(x+4,y);
    LCD_DrawPoint(x+4,y+1);
    LCD_DrawPoint(x-1,y+4);
    LCD_WR_DATA(POINT_COLOR);
    LCD_WR_DATA(POINT_COLOR);
    LCD_Fill(x-3,y-3,x+3,y+3, color);
}
```

```
/*********************************/
/* 函数功能：显示标准按钮        */
/*********************************/
void Draw_Button(uint16 xstart,uint16 ystart,uint16 xend,uint16 yend)
{
    EscButton(xstart, ystart, xend, yend);
    LCD_Fill(xstart+2, ystart+2, xend-2, yend-2,LGRAY);//填充中间颜色
}

/*************************************/
/* 函数功能：显示按钮选中状态        */
/*************************************/
void SetButton(uint8 xstart,uint16 ystart,uint8 xend,uint16 yend)
{
    POINT_COLOR=BLACK;
    LCD_DrawLine(xstart, ystart, xend, ystart);//画顶部横线 1
    LCD_DrawLine(xstart, ystart, xstart, yend);//画左边竖线 1
    POINT_COLOR=DARKGRAY;
    LCD_DrawLine(xstart+1, ystart+1, xend-1, ystart+1);//画顶部横线 2
    LCD_DrawLine(xstart+1, ystart+1, xstart+1, yend-1);//画左边竖线 2
    POINT_COLOR=LGRAY;
    LCD_DrawLine(xstart+1, yend-1, xend-1, yend-1);//画底部横线 1
    LCD_DrawLine(xend-1, ystart+1, xend-1, yend-1);//画右边竖线 1
    POINT_COLOR=WHITE;
    LCD_DrawLine(xstart, yend, xend, yend);//画底部横线 2
    LCD_DrawLine(xend, ystart, xend, yend);//画右边竖线 2
}

/*************************************/
/* 函数功能：显示按钮取消状态        */
/*************************************/
void EscButton(uint16 xstart,uint16 ystart,uint8 xend,uint16 yend)
{
    POINT_COLOR=LGRAY;
    LCD_DrawLine(xstart, ystart, xend, ystart);//画顶部横线 1
    LCD_DrawLine(xstart, ystart, xstart, yend);//画左边竖线 1
    POINT_COLOR=WHITE;
    LCD_DrawLine(xstart+1, ystart+1, xend-1, ystart+1);//画顶部横线 2
    LCD_DrawLine(xstart+1, ystart+1, xstart+1, yend-1);//画左边竖线 2
    POINT_COLOR=BLACK;
    LCD_DrawLine(xstart, yend, xend, yend);//画底部横线 1
    LCD_DrawLine(xend, ystart, xend, yend);//画右边竖线 1
    POINT_COLOR=DARKGRAY;
```

```
    LCD_DrawLine(xstart+1, yend-1, xend-1, yend-1);//画底部横线 2
    LCD_DrawLine(xend-1, ystart+1, xend-1, yend-1);//画右边竖线 2
}


/************************************/
/* 函数功能：显示一个文字输入框      */
/************************************/
void Draw_TextBox(uint16 xstart, uint16 ystart, uint16 xend, uint16 yend)
{
    POINT_COLOR=DARKGRAY;
    LCD_DrawLine(xstart, ystart, xend, ystart);//画顶部横线 1
    LCD_DrawLine(xstart, ystart+1, xstart, yend);//画左面竖线 1
    POINT_COLOR=BLACK;
    LCD_DrawLine(xstart+1, ystart+1, xend-1, ystart+1);//画顶部横线 2
    LCD_DrawLine(xstart+1, ystart+2, xstart+1, yend-1);//画左面竖线 2
    POINT_COLOR=WHITE;
    LCD_DrawLine(xstart, yend, xend, yend);//画底部横线 1
    LCD_DrawLine(xend, ystart, xend, yend);//画右面竖线 1
    POINT_COLOR=LGRAY;
    LCD_DrawLine(xstart+1, yend-1, xend-1, yend-1);//画底部横线 2
    LCD_DrawLine(xend-1, ystart+1, xend-1, yend-1);//画右面竖线 2
    LCD_Fill(xstart+2, ystart+2, xend-2, yend-2,WHITE);
}
/************************************/
/* 函数功能：显示一个窗口，          */
/* 说    明：caption:标题名称       */
/************************************/
void Draw_Window(uint16 xstart,uint16 ystart,uint8 xend,uint16 yend,uint8*
caption)
{
    Draw_Button(xstart, ystart, xend, yend);    // 显示主体窗口
    LCD_Fill(xstart+3, ystart+3, xend-3, ystart+25, DARKBLUE);    // 显示标题栏
    Draw_TextBox(xstart+3, ystart+29, xend-3, yend-3); // 显示文本输入区
    POINT_COLOR = WHITE;
    BACK_COLOR = DARKBLUE;
    LCD_ShowString(xstart+5, ystart+6, caption);
}
```

```
/*********************************************/
/* 函数功能：显示一个 Frame                    */
/* 说    明：在调用此函数时，需要先把需要    */
/*           设置的范围内背景色设置为 LGRAY */
/*********************************************/
void Draw_Frame(uint16 xstart,uint16 ystart,uint8 xend,uint16 yend,uint8 *FrameName)
{
    POINT_COLOR = DARKGRAY;
    LCD_DrawLine(xstart, ystart, xend, ystart);       // 上边框
    LCD_DrawLine(xstart, yend, xend, yend); // 下边框
    LCD_DrawLine(xstart, ystart, xstart, yend);     // 左边框
    LCD_DrawLine(xend, ystart, xend, yend);      // 右边框

    POINT_COLOR = WHITE;
    LCD_DrawLine(xstart+1, ystart+1, xend-1, ystart+1); // 上边框灯光
    LCD_DrawLine(xstart, yend+1, xend+1, yend+1);   // 下边框灯光
    LCD_DrawLine(xstart+1, ystart+1, xstart+1, yend-1);      // 左边框灯光
    LCD_DrawLine(xend+1, ystart, xend+1, yend+1);         // 右边框灯光

    POINT_COLOR = BLACK;
    BACK_COLOR = LGRAY;
    LCD_ShowString(xstart+5, ystart-6, FrameName);   // 显示 Frame 名称
}


/*********************************/
/* 函数功能：LOADING.....          */
/*********************************/
void POINT_Demo(void)
{
    uint8 x=120,y=160,r=20,t,m,i;

    t = 7*r/10;
    m =50;  //显示速度毫秒值

    LCD_Clear(BLACK);
    POINT_COLOR = WHITE;
    BACK_COLOR = BLACK;
    LCD_ShowString(90, 190, "Loading.....");
    for(i=0;i<2;i++)
    {
        LCD_Draw9Point(x, y-r, WHITE); //画第一个点
```

```
            delay_ms(m);
            LCD_Draw9Point(x+t, y-t, WHITE); //画第二个点
            delay_ms(m);
            LCD_Draw9Point(x, y-r, BLACK); //删除第一个点
            delay_ms(m);
            LCD_Draw9Point(x+r, y, WHITE); //画第三个点
            delay_ms(m);
            LCD_Draw9Point(x+t, y-t, BLACK);    //删除第二个点
            delay_ms(m);
            LCD_Draw9Point(x+t, y+t, WHITE);    //画第四个点
            delay_ms(m);
            LCD_Draw9Point(x+r, y, BLACK); //删除第三个点
            delay_ms(m);
            LCD_Draw9Point(x, y+r, WHITE);    //画第五个点
            delay_ms(m);
            LCD_Draw9Point(x+t, y+t, BLACK);    //删除第四个点
            delay_ms(m);
            LCD_Draw9Point(x-t, y+t, WHITE);    //画第六个点
            delay_ms(m);
            LCD_Draw9Point(x, y+r, BLACK);    //删除第五个点
            delay_ms(m);
            LCD_Draw9Point(x-r, y, WHITE);    //画第七个点
            delay_ms(m);
            LCD_Draw9Point(x-t, y+t, BLACK);    //删除第六个点
            delay_ms(m);
            LCD_Draw9Point(x-t, y-t, WHITE);    //画第八个点
            delay_ms(m);
            LCD_Draw9Point(x-r, y, BLACK);    //删除第七个点
            delay_ms(m);
            LCD_Draw9Point(x-t, y-t, BLACK);    //删除第八个点
            delay_ms(m);
        }
}


/*******************************************/
/* 函数功能：颜色显示，观看刷屏速度          */
/*******************************************/
void RGB_Demo(void)
{
    uint8 i;

    LCD_Clear(BLUE);
```

```
    for(i=0;i<3;i++)
    {
    LCD_Fill(0, 0,100,100,YELLOW);
    LCD_Fill(0, 0,120,120,RED);
    LCD_Fill(0, 0,140,140,GREEN);
    LCD_Fill(0, 0,160,160,PINK);
    LCD_Fill(0, 0,180,180,GRAY);
    LCD_Fill(0, 0,200,200,ORANGE);
    LCD_Fill(0, 0,200,200,PORPO);
    LCD_Fill(0, 0,200,200,LGRAYBLUE);
    LCD_Fill(0, 0,200,200,BLUE);
    }
    LCD_Clear(BLUE);
}

/*******************************************/
/* 函数功能：动感条形报表                  */
/*******************************************/
void BarReport_Demo(void)
{
    uint16 i;

    LCD_Clear(BLACK);
    POINT_COLOR = WHITE;
    //画纵坐标
    LCD_DrawLine(20, 140, 20, 300);
    LCD_DrawLine(10, 150, 20, 140);
    LCD_DrawLine(30, 150, 20, 140);
    //画横坐标
    LCD_DrawLine(20, 300, 220, 300);
    LCD_DrawLine(210, 290, 220, 300);
    LCD_DrawLine(210, 310, 220, 300);
    //画条形
    LCD_Fill(35, 170, 55, 299,RED);
    LCD_Fill(75, 220, 95, 299,YELLOW);
    LCD_Fill(115, 150, 135, 299,BLUE);
    LCD_Fill(155, 180, 175, 299,GREEN);
    //条形渐变
    delay_ms(50);
    for(i=171;i<299;i++) //红色条降低
    {
        LCD_Fill(35, 170, 55, i,BLACK);
        delay_ms(10);
```

```
    }
    for(i=298;i>190;i--) //红色条升高
    {
        LCD_Fill(35, i, 55, 299,RED);
        delay_ms(10);
    }
    for(i=219;i>170;i--) //黄色条升高
    {
        LCD_Fill(75, i, 95, 220,YELLOW);
        delay_ms(10);
    }
    for(i=115;i<250;i++) //蓝色条降低
    {
        LCD_Fill(115, 114, 135, i,BLACK);
        delay_ms(10);
    }
}


/*******************************************/
/* 函数功能：进度条演示                    */
/*******************************************/
void ProgresBar_Demo(void)
{
    uint8 i,num=1;

    LCD_Clear(BLUE);      // 整屏显示红色
    Draw_Button(100, 210, 230, 310);    // 显示主体窗口
    LCD_Fill(103, 213, 227, 235, DARKBLUE);    // 显示标题栏
    Draw_Button(105, 280, 155, 305);   // 显示第一个按钮
    Draw_Button(175, 280, 225, 305);   // 显示第二个按钮
    POINT_COLOR=BLACK;
    BACK_COLOR=LGRAY;
    LCD_ShowString( 114, 284, "确定"); // 按钮上写字
    LCD_ShowString( 184, 284, "退出");
    LCD_ShowString(180, 245, "%");
    for(i=126;i<225;i++)
    {
        LCD_Fill(105, 263, i, 278, RED);
        delay_ms(40);
        LCD_ShowNum(162, 245, num,2);
        num++;
    }
```

```
}


/*****************************************/
/* 函数功能：画黑箭头方向图标(向下)          */
/*****************************************/
void Draw_DirectButton(uint16 xstart, uint16 ystart)
{
    POINT_COLOR=BLACK;
    LCD_DrawLine(xstart+6, ystart+8, xstart+14, ystart+8);
    LCD_DrawLine(xstart+7, ystart+9, xstart+13, ystart+9);
    LCD_DrawLine(xstart+8, ystart+10, xstart+12, ystart+10);
    LCD_DrawPoint(xstart+9,ystart+11);LCD_DrawPoint(xstart+10,ystart+11);LCD
_DrawPoint(xstart+11,ystart+11);
    LCD_DrawPoint(xstart+10,ystart+12);
}

/*****************************************/
/* 函数功能：combo 效果演示                */
/*****************************************/
void ComboDemo(void)
{
    LCD_Clear(GRAY);
    //画一个条形输入框
    Draw_TextBox(50, 50, 200, 73);
    //画下拉列表按钮(19*19)像素
    Draw_Button(179, 52, 198, 71);
    Draw_DirectButton(179,52);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    SetButton(179, 52, 198, 71);
    LCD_Fill(183, 56, 194, 67, LGRAY);
    Draw_DirectButton(180,53);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    EscButton(179, 52, 198, 71);
    LCD_Fill(183, 56, 194, 67, LGRAY);
    Draw_DirectButton(179,52);
    //拉出下拉列表
    LCD_DrawRectage(50, 74, 200, 143, BLACK);
    LCD_Fill(51, 75, 199, 142, WHITE);
```

```
    //写列表中的内容
    LCD_Fill(51, 75, 199, 97, DARKBLUE); //第一个默认为选中状态
    POINT_COLOR=WHITE;
    BACK_COLOR=DARKBLUE;
    LCD_ShowString(53, 79, "NXP ICP Bridge");

    POINT_COLOR=BLACK;
    BACK_COLOR=WHITE;
    LCD_ShowString(53, 101, "NXP PP Bridge");
    LCD_ShowString(53, 124, "None ISP");
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    //选中第二个
    LCD_Fill(51, 75, 199, 97, WHITE);          //先取消第一个
    POINT_COLOR=BLACK;
    BACK_COLOR=WHITE;
    LCD_ShowString(53, 79, "NXP ICP Bridge");

    LCD_Fill(51, 98, 199, 120, DARKBLUE);    //选中第二个
    POINT_COLOR=WHITE;
    BACK_COLOR=DARKBLUE;
    LCD_ShowString(53, 101, "NXP PP Bridge");
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    //选中第三个
    LCD_Fill(51, 98, 199, 120, WHITE);//先取消选中的第二个
    POINT_COLOR=BLACK;
    BACK_COLOR=WHITE;
    LCD_ShowString(53, 101, "NXP PP Bridge");

    LCD_Fill(51, 121, 199, 142, DARKBLUE);//选中第三个
    POINT_COLOR=WHITE;
    BACK_COLOR=DARKBLUE;
    LCD_ShowString(53, 124, "None ISP");
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    //清除
    LCD_Fill(50, 74, 200, 143, LGRAY);
}
```

```
/*******************************************/
/* 函数功能：Window 效果演示                */
/*******************************************/
void Window_Demo(void)
{
    uint16 xstart=0,ystart=0,xend=239,yend=319;
    uint8 i=5;

    do
    {
        Draw_Window(xstart,ystart,xend,yend,"标题栏");
        delay_ms(500);
        delay_ms(500);
        xstart+=15;ystart+=15;xend-=30;yend-=30;
    }while(--i);
}

/*******************************************/
/* 函数功能：Button 效果演示                */
/*******************************************/
void Button_Demo(void)
{
    LCD_Clear(WHITE);
    Draw_TextBox(30, 60, 170, 90);    // 显示一个文字输入框
    Draw_Button(180, 60, 230, 90);          // 显示 1 个按钮
    POINT_COLOR = BLACK;
    BACK_COLOR = LGRAY;
    LCD_ShowString(187,67,"搜索");      // 按钮上写字

    POINT_COLOR = BLUE;
    BACK_COLOR = WHITE;
    LCD_ShowString(30,38,"新闻");
    LCD_DrawLine(30,55,62,55);
    LCD_ShowString(110,38,"图片");
    LCD_DrawLine(110,55,142,55);
    POINT_COLOR = BLACK;
    LCD_ShowString(70,38,"标签");
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);

    LCD_ShowChar(35, 67, 'C');
```

```
    delay_ms(500);
    LCD_ShowChar(43, 67, 'o');
    delay_ms(500);
    LCD_ShowChar(51, 67, 'r');
    delay_ms(500);
    LCD_ShowChar(58, 67, 't');
    delay_ms(500);
    LCD_ShowChar(66, 67, 'e');
    delay_ms(500);
    LCD_ShowChar(74, 67, 'x');
    delay_ms(500);
    LCD_ShowChar(82, 67, '-');
    delay_ms(500);
    LCD_ShowChar(88, 67, 'M');
    delay_ms(500);
    LCD_ShowChar(96, 67, '0');
    delay_ms(500);

    SetButton(180, 60, 230, 90);            // 按下“搜索”按钮
    POINT_COLOR = BLACK;
    BACK_COLOR = LGRAY;
    LCD_ShowString(188,68,"搜索");
    delay_ms(500);

    EscButton(180, 60, 230, 90);            // 放开“搜索”按钮
    POINT_COLOR = BLACK;
    BACK_COLOR = LGRAY;
    LCD_ShowString(187,67,"搜索");
    POINT_COLOR = BLACK;
    BACK_COLOR = WHITE;
    LCD_ShowString(12,100,"ARM Cortex-M0 处理器是现有的最小、能耗最低和能效最高的 ARM 处理器。该处理器硅面积极小、能耗极低并且所需的代码量极少，这使得开发人员能够以 8 位的设备实现 32 位设备的性能，从而省略 16 位设备的研发步骤。Cortex-M0  处理器超低的门数也使得它可以部署在模拟和混合信号设备中。");
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
```

```
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
    delay_ms(500);
}
```

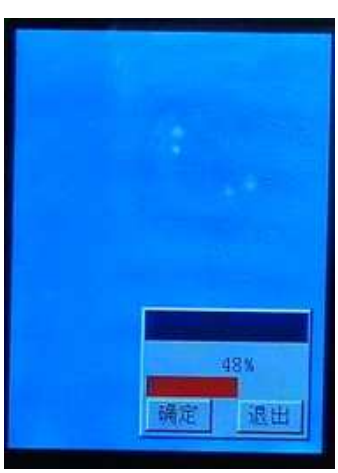
三、程序详解

上面的函数当中。前面几个实现了按钮、窗口、Frame 还有大圆点的函数。后面几个函数是对前面几个函数的应用。在源程序的主函数当中，我都调用了一遍，你把源程序下载到单片机上即可看到。你会发现，和电脑上的 GUI 没有任何区别，因为我就是按照电脑上的显示原理做的。

这些函数都是些画点，画线，画圆，画矩形的函数组成的，没有什么好讲的了。

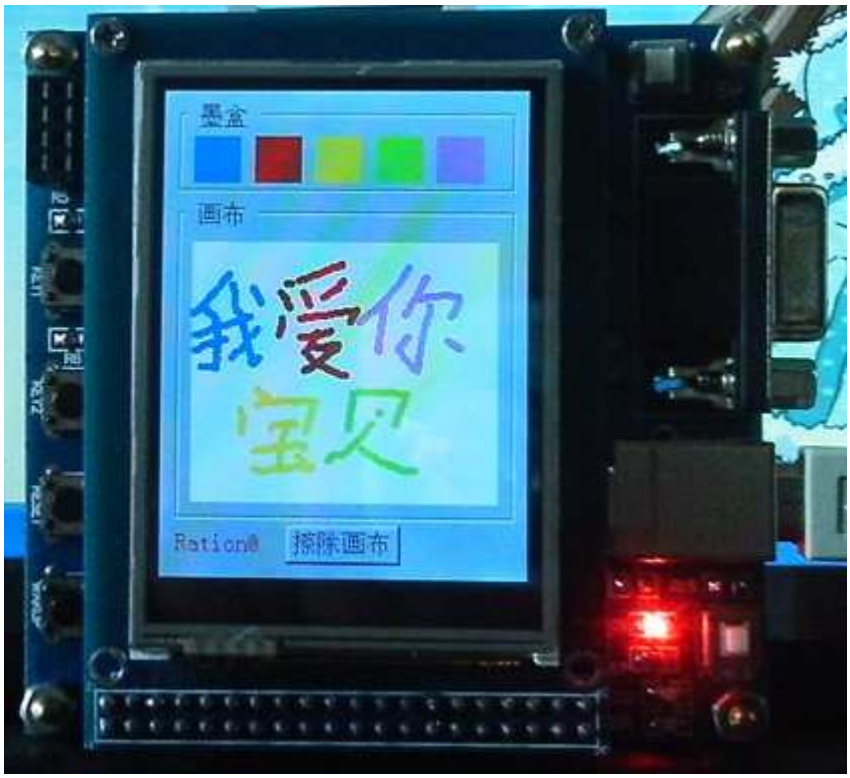
四、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。即可看到效果，整体是一个动画，这里截几张图看看。



第十四章 触摸屏

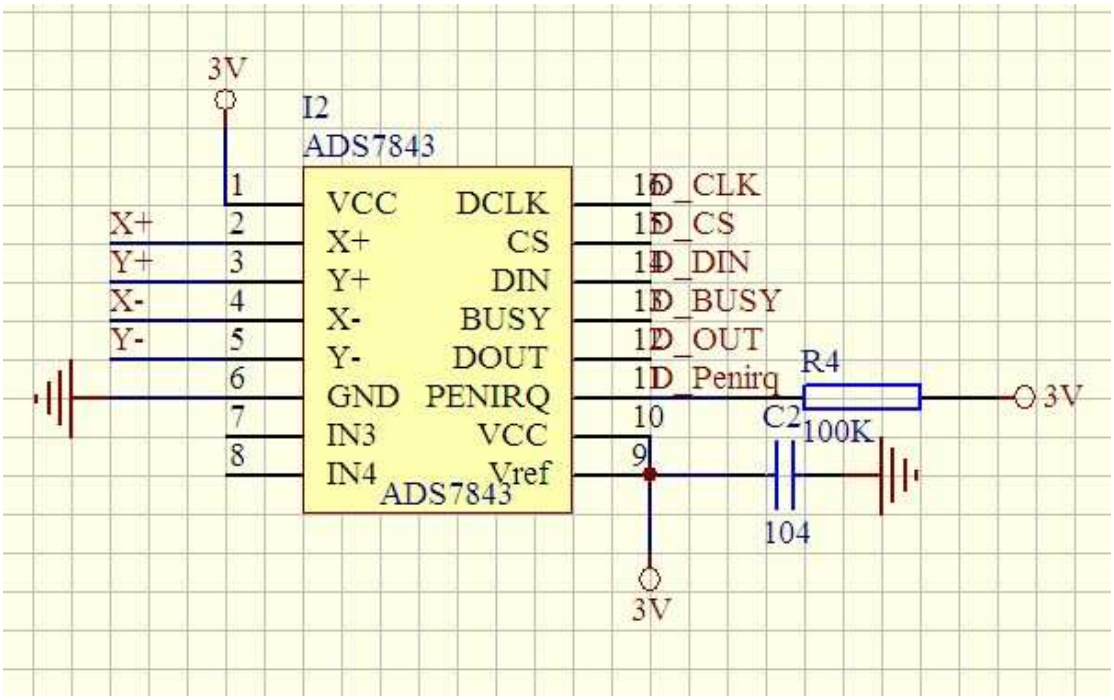
一、入门引导
二、GPIO 口模拟 SPI 时序与 XPT2046 通信实现
三、LPC1114 硬件 SPI 模块与 XPT2046 通信实现
四、实验程序下载和使用说明



一、入门引导

触摸屏的优势：在小型手持设备中，有时候需要执行很多的操作，如果操作比较多的话，只按键就需要好几十个。如果带上触摸屏的话，相当于有了无数的按键，想怎么设置就怎么设置。所以触摸屏可不仅仅只有“人性化”这么简单。

在本开发板上，所用的触摸屏芯片是 XPT2046，这个芯片的本质实际上是一个 12 位的 ADC，电阻式触摸感应原理，它还可以被用做电池电量监控等等。电路连接图如下：



其中，左边的 X+ X- Y+ Y-接到了 TFT 上，右边的以 D 开头的是控制线，连到了单片机上。芯片相当于一座架起单片机与触摸屏的桥梁。当然，你也可以直接用 LPC1114 的 4 个 ADC 引脚连接到 TFT 上，单片机的主要工作可就要分心了。所以我们一般还是选择带上一个触摸屏控制芯片。

与 XPT2046 采用 SPI 通信方式。可以用 LPC1114 的硬件 SPI 口与它通信，也可以用 GPIO 口模拟与它通信。下面将列出两种方式的通信程序。效果一样都很好。D_BUSY 引脚不好用，没有连接这个脚。D_Penirq 是中断引脚。连接到了单片机的 P2.0 脚。采用中断方式感应。

二、GPIO 口模拟 SPI 时序与 XPT2046 通信实现

```
void PIOINT2_IRQHandler(void)
{
    Pen_Point.Pen_Sign=Pen_Down; // 按键按下
    GPIO2->IC |= (1<<0);          // 清除 P2.0 上的中断
}

void Touch_Init(void)
{
    // Penirq 中断引脚设置
```

```
    GPIO2->DIR &= ~(1<<0); //把 P2.0 设置为输入
    GPIO2->IS &= ~(1<<0); //选择 P2.0 为边沿触发
    GPIO2->IEV &= ~(1<<0);//选择 P2.0 为下降沿沿触发
    GPIO2->IE |= (1<<0);    //设置 P2.0 中断不被屏蔽
    NVIC_EnableIRQ(EINT2_IRQn); //  开 GPIO2 中断
    // T_CS 片选引脚设置
    GPIO0->DIR |= (1<<2);//把 P0.2 设置为输出
    GPIO0->DATA |= (1<<2);//P0.2 引脚置 1
    // SPI 通信引脚设置（模拟，非硬件 SPI）
    GPIO2->DIR |= (1<<1);//把 P2.1 设置为输出  用作 SCK
    GPIO2->DIR &= ~(1<<2);//把 P2.2 设置为输入  用作 MISO
    GPIO2->DIR |= (1<<3);//把 P2.3 设置为输出  用作 MOSI
    GPIO2->DATA &= ~(1<<1); // SCK = 0;
    GPIO2->DATA |= (1<<3);     // MOSI = 1;
    Pen_Point.Pen_Sign=Pen_Up;
    Read_ADS(&Pen_Point.X_ADC,&Pen_Point.Y_ADC);
}
//SPI 写数据
void Touch_Write(uint8 wbyte)
{
    uint8 i;

    GPIO2->DATA &= ~(1<<1); // SCK = 0;
    delay_us(1);
    for(i=0;i<8;i++)
    {
        if(wbyte&0x80)
        GPIO2->DATA |= (1<<3);     // MOSI = 1;
        else GPIO2->DATA &= ~(1<<3);  // MOSI = 0;
        wbyte<<=1;
        GPIO2->DATA &= ~(1<<1); // SCK = 0;上升沿读入数据
        __nop();__nop();__nop();
        GPIO2->DATA |= (1<<1);     // SCK = 1;
        __nop();__nop();__nop();
    }
}
```

```
/*************************************************/
/* 功      能：读取 X 轴或 Y 轴的 ADC 值                    */
/* 入口参数：CMD:命令                                  */
/*************************************************/
uint16 ADS_Read_AD(uint8 CMD)
{
    uint8 count=0;
    uint16 Num=0;
    GPIO2->DATA &= ~(1<<1); // SCK = 0;
    GPIO0->DATA &= ~(1<<2); // CS=0  开始 SPI 通信
    Touch_Write(CMD);           //  发送命令字
    delay_us(6);                //  延时等待转换完成
    GPIO2->DATA |= (1<<1);     // SCK = 1;
    __nop();__nop();__nop();
    GPIO2->DATA &= ~(1<<1); // SCK = 0;
    __nop();__nop();__nop();
    for(count=0;count<16;count++)
    {
        Num<<=1;
        GPIO2->DATA &= ~(1<<1); // SCK = 0;
        __nop();__nop();__nop();
        GPIO2->DATA |= (1<<1);     // SCK = 1;
        __nop();__nop();__nop();
        if(GPIO2->DATA&(1<<2))Num++;
    }
    Num>>=4;                    //  只有高 12 位有效.
    GPIO0->DATA |= (1<<2);   // CS=1  结束 SPI 通信
    return(Num);
}
#define READ_TIMES 10 //读取次数
#define LOST_VAL 4      //丢弃值
/*************************************************/
/* 功      能：读取 X 轴或 Y 轴的 ADC 值                    */
/* 入口参数：CMD_RDX:读取 X 的 ADC 值                    */
/*           CMD_RDY:读取 Y 的 ADC 值                    */
/* 说      明：与上一个函数相比，这个带有滤波          */
/*************************************************/
uint16 ADS_Read_XY(uint8 xy)
{
    uint16 i, j;
    uint16 buf[READ_TIMES];
    uint16 sum=0;
    uint16 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;
}
/**************************************************/
/* 功      能：读取 X 轴和轴的 ADC 值                    */
/* 入口参数：&Pen_Point.X_ADC,&Pen_Point.Y_ADC     */
/* 出口参数：0：成功（返回的 X,Y_ADC 值有效）          */
/*           1: 失败（返回的 X,Y_ADC 值无效）          */
/**************************************************/
uint8 Read_ADS(uint16 *x,uint16 *y)
{
    uint16 xtemp,ytemp;
    xtemp=ADS_Read_XY(CMD_RDX);
    ytemp=ADS_Read_XY(CMD_RDY);

    if(xtemp<100||ytemp<100)return 1;//读数失败
    *x=xtemp;
    *y=ytemp;
    return 0;//读数成功
}
```

```
/*************************************************/
/* 功能：连续两次读取 ADC 值                     */
/* 原理：把两次读取的值作比较，在误差范围内可取  */
/* 入口参数：&Pen_Point.X_ADC,&Pen_Point.Y_ADC   */
/* 出口参数：0：成功（返回的 X,Y_ADC 值有效）    */
/*           1: 失败（返回的 X,Y_ADC 值无效）    */
/*************************************************/
#define ERR_RANGE 50 //误差范围
uint8 Read_ADS2(uint16 *x,uint16 *y)
{
    uint16 x1,y1;
    uint16 x2,y2;
    uint8 res;

    res=Read_ADS(&x1,&y1);   // 第一次读取 ADC 值
    if(res==1)return(1);// 如果读数失败，返回 1
    res=Read_ADS(&x2,&y2);   // 第二次读取 ADC 值
    if(res==1)return(1);      // 如果读数失败，返回 1
    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 0;   // 正确读取，返回 0
    }else return 1;   // 前后不在+-50 内，读数错误
}

/*************************************************/
/* 功能：把读出的 ADC 值转换成坐标值             */
/*************************************************/
void Change_XY(void)
{
    Pen_Point.X_Coord=(240-(Pen_Point.X_ADC-100)/7.500); // 把读到的 X_ADC 值转换成 TFT X 坐标值
    Pen_Point.Y_Coord=(320-(Pen_Point.Y_ADC-135)/5.705); // 把读到的 Y_ADC 值转换成 TFT Y 坐标值
}
```

```
/*************************************************/
/* 功能：读取一次 XY 坐标值                      */
/*************************************************/
uint8 Read_Once(void)
{
    Pen_Point.Pen_Sign=Pen_Up;
    if(Read_ADS2(&Pen_Point.X_ADC,&Pen_Point.Y_ADC)==0)// 如果读取数据成功
    {
        while((GPIO2->DATA&0X1)==0);    // 检测笔是不是还在屏上
        Change_XY();    // 把读到的 ADC 值转变成 TFT 坐标值
        return 0; // 返回 0，表示成功
    }
    else return 1;  // 如果读取数据失败，返回 1 表示失败
}
/*************************************************/
/* 功能：持续读取 XY 坐标值                      */
/*************************************************/
uint8 Read_Continue(void)
{
    Pen_Point.Pen_Sign=Pen_Up;
    if(Read_ADS2(&Pen_Point.X_ADC,&Pen_Point.Y_ADC)==0)  // 如果读取数据成功
    {
        Change_XY();    // 把读到的 ADC 值转变成 TFT 坐标值
        return 0;       // 返回 0，表示成功
    }
    else return 1;      // 如果读取数据失败，返回 1 表示失败
}
```

从初始化函数，你可以看出来，这是通过 GPIO 口模拟 SPI 时序与 XPT2046 通信的。读出的触摸屏值为电压值，从上往下，值由 1900 变到 100，从左往右，值由 1900 变到 100。从这个值看出，触摸屏的原点是在左下角。而我们的 TFT 原点是在右上角，正好相反。所以用上面的 void Change_XY(void)函数实现了原点的转换。

三、LPC1114 硬件 SPI 模块与 XPT2046 通信实现

```
void PIOINT2_IRQHandler(void)
{
    Pen_Point.Pen_Sign=Pen_Down;// 按键按下
    GPIO2->IC |= 0x3FF;          // 清除 P2 口上的中断
}

void Touch_Init(void)
{
    // T_CS 片选引脚设置
    GPIO0->DIR |= (1<<2);//把 P0.2 设置为输出
    GPIO0->DATA |= (1<<2);//P0.2 引脚置 1
    // SPI 通信引脚设置
    SPI1_Init();
    // Penirq 中断引脚设置
    GPIO2->DIR &= ~(1<<0); //把 P2.0 设置为输入
    GPIO2->IS &= ~(1<<0); //选择 P2.0 为边沿触发
    GPIO2->IEV &= ~(1<<0);//选择 P2.0 为下降沿沿触发
    GPIO2->IE |= (1<<0);   //设置 P2.0 中断不被屏蔽
    NVIC_EnableIRQ(EINT2_IRQn);// 开 GPIO2 中断
    Pen_Point.Pen_Sign=Pen_Up;
    ADS_Read_AD(CMD_RDX);
    ADS_Read_AD(CMD_RDY);
}

/*************************************************/
/* 功      能：读取 X 轴或 Y 轴的 ADC 值                    */
/* 入口参数：CMD:命令                                  */
/*************************************************/
uint16 ADS_Read_AD(uint8 CMD)
{
    uint16 NUMH,NUML;
    uint16 Num;

    GPIO0->DATA &= ~(1<<2); // CS=0 开始 SPI 通信
    delay_us(1);
    SPI1_communication(CMD);
    delay_us(6);                // 延时等待转换完成
    NUMH=SPI1_communication(0x00);
    NUML=SPI1_communication(0x00);
    Num=((NUMH)<<8)+NUML;
    Num>>=4;                    // 只有高 12 位有效.
    GPIO0->DATA |= (1<<2);   // CS=1 结束 SPI 通信
```

```
    return(Num);
}
#define READ_TIMES 10 //读取次数
#define LOST_VAL 4      //丢弃值
/*****************************************************/
/* 功      能：读取 X 轴或 Y 轴的 ADC 值                    */
/* 入口参数：CMD_RDX:读取 X 的 ADC 值                      */
/*           CMD_RDY:读取 Y 的 ADC 值                      */
/* 说      明：与上一个函数相比，这个带有滤波        */
/*****************************************************/
uint16 ADS_Read_XY(uint8 xy)
{
    uint16 i, j;
    uint16 buf[READ_TIMES];
    uint16 sum=0;
    uint16 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;
}
```

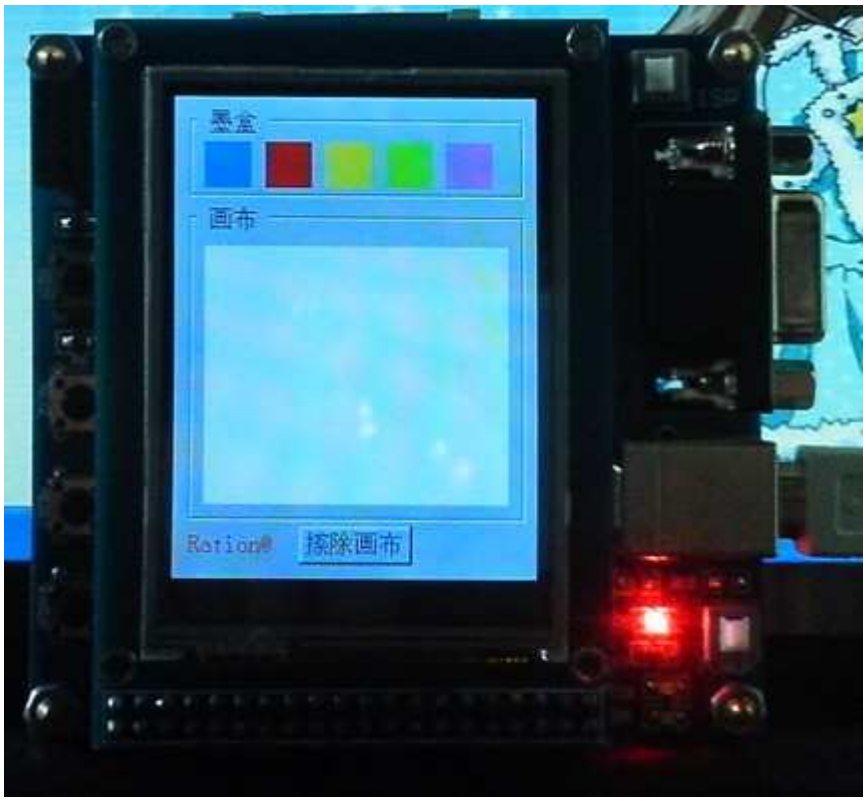
```
/*************************************************/
/* 功      能：读取 X 轴和轴的 ADC 值                    */
/* 入口参数：&Pen_Point.X_ADC,&Pen_Point.Y_ADC    */
/* 出口参数：0：成功（返回的 X,Y_ADC 值有效）          */
/*           1：失败（返回的 X,Y_ADC 值无效）          */
/*************************************************/
uint8 Read_ADS(uint16 *x,uint16 *y)
{
    uint16 xtemp,ytemp;
    xtemp=ADS_Read_XY(CMD_RDX);
    ytemp=ADS_Read_XY(CMD_RDY);

    if(xtemp<100||ytemp<100)return 1;//读数失败
    *x=xtemp;
    *y=ytemp;
    return 0;//读数成功
}
/*************************************************/
/* 功能：连续两次读取 ADC 值                       */
/* 原理：把两次读取的值作比较，在误差范围内可取    */
/* 入口参数：&Pen_Point.X_ADC,&Pen_Point.Y_ADC    */
/* 出口参数：0：成功（返回的 X,Y_ADC 值有效）          */
/*           1：失败（返回的 X,Y_ADC 值无效）          */
/*************************************************/
#define ERR_RANGE 50 //误差范围
uint8 Read_ADS2(uint16 *x,uint16 *y)
{
    uint16 x1,y1;
    uint16 x2,y2;
    uint8 res;

    res=Read_ADS(&x1,&y1);   // 第一次读取 ADC 值
    if(res==1)return(1);// 如果读数失败，返回 1
    res=Read_ADS(&x2,&y2);   // 第二次读取 ADC 值
    if(res==1)return(1);      // 如果读数失败，返回 1
    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 0;   // 正确读取，返回 0
    }else return 1;   // 前后不在+-50 内，读数错误
```

```
}

/*************************************************/
/* 功能：把读出的 ADC 值转换成坐标值                */
/*************************************************/
void Change_XY(void)
{
    Pen_Point.X_Coord=(240-(Pen_Point.X_ADC-100)/7.500); // 把读到的 X_ADC 值转换成 TFT X 坐标值
    Pen_Point.Y_Coord=(320-(Pen_Point.Y_ADC-135)/5.705); // 把读到的 Y_ADC 值转换成 TFT Y 坐标值
}

/*************************************************/
/* 功能：读取一次 XY 坐标值                         */
/*************************************************/
uint8 Read_Once(void)
{
    Pen_Point.Pen_Sign=Pen_Up;
    if(Read_ADS2(&Pen_Point.X_ADC,&Pen_Point.Y_ADC)==0)// 如果读取数据成功
    {
        while((GPIO2->DATA&0X1)==0);    // 检测笔是不是还在屏上
        Change_XY();    // 把读到的 ADC 值转变成 TFT 坐标值
        return 0; // 返回 0，表示成功
    }
    else return 1;  // 如果读取数据失败，返回 1 表示失败
}
/*************************************************/
/* 功能：持续读取 XY 坐标值                         */
/*************************************************/
uint8 Read_Continue(void)
{
    Pen_Point.Pen_Sign=Pen_Up;
    if(Read_ADS2(&Pen_Point.X_ADC,&Pen_Point.Y_ADC)==0)  // 如果读取数据成功
    {
        Change_XY();    // 把读到的 ADC 值转变成 TFT 坐标值
        return 0;       // 返回 0，表示成功
    }
    else return 1;      // 如果读取数据失败，返回 1 表示失败
}
```

在 LPC1114 上面，有两个 SSP 口，这个用到了 SSP1 口，其它的三个器件：NRF24L01\W25X16\SD 卡用的是 SSP0 口，这三个器件的 SPI 时序一模一样，只是用了不同的片选引脚，所以可以同时使用。而 XPT2046 的 SPI 时序和这三个器件有一定的不同。所以用了 SSP1 口，在配置通信时序上，有些差别。这样的话。四个 SPI 器件就可以同时使用了。

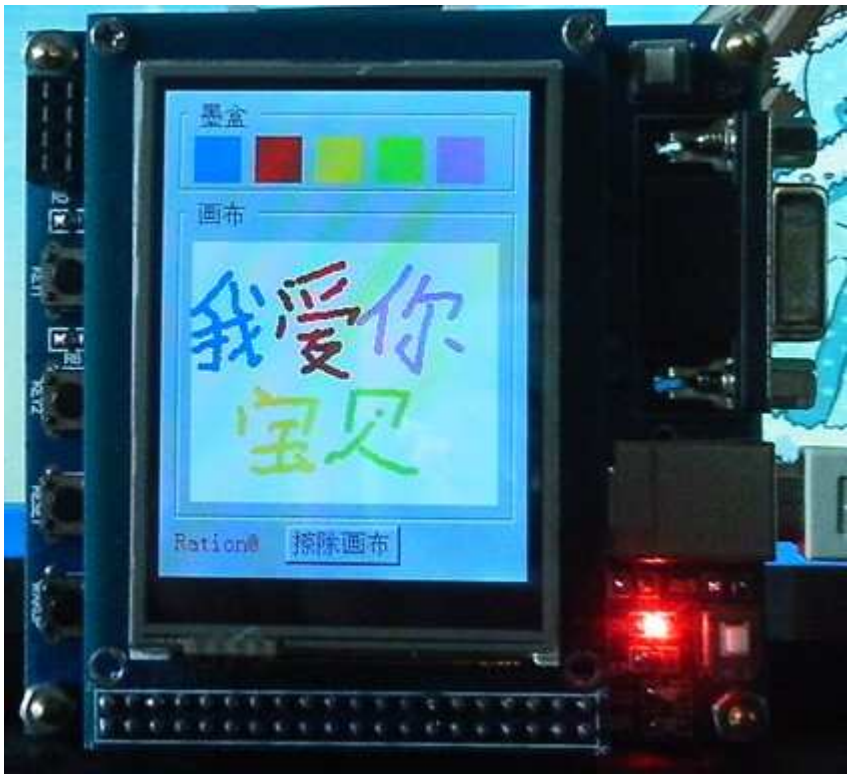
四、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。即可看到效果，如下所示：



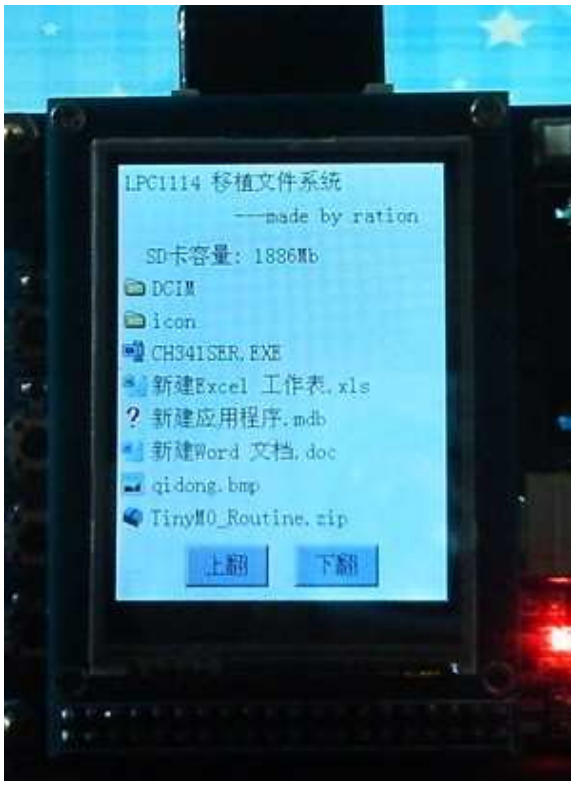
整体分为两个 Frame，第一个是“墨盒”，第二个是“画布”。你可以用笔触摸“墨盒”中的颜色来选择画笔的颜色。然后再画布上画画或写字。默认的颜色为蓝色。

效果如下图：


黑盒
画布
我爱你
宝贝
Ration
擦除画布

第十五章 移植 FatFs 文件系统

一、入门引导
二、移植
三、实验程序下载和使用说明



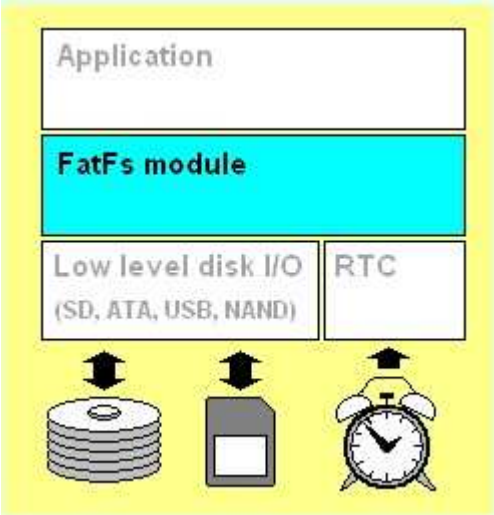
一、入门引导

FatFs 是一个通用的文件系统模块，用于在小型嵌入式系统中实现 FAT 文件系统。 FatFs 的编写遵循 ANSI C，因此不依赖于硬件平台。它可以嵌入到便宜的微控制器中，如 8051，PIC，AVR，SH，Z80，H8，ARM 等等，不需要做任何修改。

这是一真正开源免费的文件系统。为什么是真正呢？因为你不仅可以免费下载到源代码，免费学习研究，最重要的是用在商业用途也不用向作者缴费。最后一点可是很多打着“开源”旗号的东西比不了的。

它的文件结构图如下所示。源代码网址：

http://elm-chan.org/fsw/ff/00index_e.html



支持 FAT，FAT16，FAT32；
支持多扇区读写，使得效率更高；
支持长文件名读写；
支持中文；
移植步骤超级简单。

二、移植

在实验程序里的 FatFs 是移植的 R0.08 版本。是我现在写这个教程时候的最新版。

下载下来以后，里面有这么几个文件：

在 KEIL 下编译，只需修改上面的 diskio.c 文件和 ffconf.h 文件就可以了。其中，ffconf.h 文件是配置文件，可以配置文件系统支持的范围。diskio.c 文件是和硬件连接的几口，把里面的几个函数实现了即可。修改以后的 diskio.c 文件如下所示：

```
/*-----------------------------------------------------------------------*/
/* Inidialize a Drive                                                    */
/*-----------------------------------------------------------------------*/

DSTATUS disk_initialize (
	BYTE drv				/* Physical drive nmuber (0..) */
)
{
	DSTATUS stat;

	stat = 0;

	return stat;
}

/*-----------------------------------------------------------------------*/
/* Return Disk Status                                                    */
/*-----------------------------------------------------------------------*/

DSTATUS disk_status (
	BYTE drv		/* Physical drive nmuber (0..) */
)
{
	DSTATUS stat;

	stat = 0;

	return stat;
}

/*-----------------------------------------------------------------------*/
/* Read Sector(s)                                                        */
/*-----------------------------------------------------------------------*/

DRESULT disk_read (
	BYTE drv,		/* Physical drive nmuber (0..) */
	BYTE *buff,		/* Data buffer to store read data */
	DWORD sector,	/* Sector address (LBA) */
	BYTE count		/* Number of sectors to read (1..255) */
)
{
	if (count < 1)
	{
```

```
        return RES_PARERR;
    }

    if (count == 1)    /* Single block read      */
    {
         if (SD_ReadSingleBlock(sector, buff) != 0)
         {
              return RES_ERROR;
         }
    }
    else                    /* Multiple block read */
    {
        if(SD_ReadMultiBlock(sector, buff, count) != 0)
        {
              return RES_ERROR;
        }
    }

    return RES_OK;
}

/*-----------------------------------------------------------------------*/
/* Write Sector(s)                                                       */
/*-----------------------------------------------------------------------*/
/* The FatFs module will issue multiple sector transfer request
/   (count > 1) to the disk I/O layer. The disk function should process
/   the multiple sector transfer properly Do. not translate it into
/   multiple single sector transfers to the media, or the data read/write
/   performance may be drasticaly decreased. */

#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,               /* Physical drive nmuber (0..) */
    const BYTE *buff,  /* Data to be written */
    DWORD sector,           /* Sector address (LBA) */
    BYTE count              /* Number of sectors to write (1..255) */
)
{
    if (count < 1)
    {
        return RES_PARERR;
    }
```

```
    if (count == 1)    /* Single block write      */
    {
        if (SD_WriteSingleBlock(sector, buff) != 0)
        {
            return RES_ERROR;
        }
    }
    else                    /* Multiple block write */
    {
        if(SD_WriteMultiBlock(sector, buff, count) != 0)
        {
            return RES_ERROR;
        }
    }

    return RES_OK;
}
#endif /* _READONLY */



/*-----------------------------------------------------------------------*/
/* Miscellaneous Functions                                               */
/*-----------------------------------------------------------------------*/

DRESULT disk_ioctl (
    BYTE drv,           /* Physical drive nmuber (0..) */
    BYTE ctrl,          /* Control code */
    void *buff          /* Buffer to send/receive control data */
)
{
    DRESULT res;
    BYTE n, csd[16];
    DWORD csize;

    res = RES_ERROR;

    switch (ctrl)
    {
        case CTRL_SYNC          : res = RES_OK; break;
        case GET_SECTOR_COUNT: /* Get number of sectors on the disk (WORD) */
                                if(SD_GetCSD(csd) == 0)
                                {
```

```
                                        if((csd[0] >> 6) == 1) /* SDC ver 2.00 */
                                        {
                                            csize = ((DWORD)csd[9]) + ((DWORD)csd[8]
<< 8) + 1;
                                            *(DWORD*)buff = (DWORD)csize << 10;
                                        }
                                        else /* MMC or SDC ver 1.XX */
                                        {
                                            n = (csd[5] & 0x0F) + ((csd[10] & 0x80) >> 7) +
((csd[9] & 0x03) << 1) + 2;
                                            csize = (csd[8] >> 6) + ((WORD)csd[7] << 2) +
((WORD)(csd[6] & 0x03) << 10) + 1;
                                            *(DWORD*)buff = (DWORD)csize << (n - 9);
                                        }
                                        res = RES_OK;
                                    }
                                    break;

        case GET_SECTOR_SIZE : /* Get sectors on the disk (WORD) */
                                    *(DWORD*)buff = 512;
                                    res = RES_OK;
                                    break;

        case GET_BLOCK_SIZE    : if (SD_GetCSD(csd) == 0) /* Read CSD */
                                    {
                                        *(DWORD*)buff = (((csd[10] & 0x3F) << 1) +
((WORD)(csd[11] & 0x80) >> 7) + 1) << ((csd[13] >> 6) - 1);
                                        res = RES_OK;
                                    }
                                    break;

        default                : res = RES_OK; break;
    }

    return res;
}

/*-----------------------------------------------------------------------*/
*                                End Of File
/*-----------------------------------------------------------------------*/
```

上面一共有五个函数：

```
DSTATUS disk_initialize (BYTE);
DSTATUS disk_status (BYTE);
DRESULT disk_read (BYTE, BYTE*, DWORD, BYTE);
#if  _READONLY == 0
DRESULT disk_write (BYTE, const BYTE*, DWORD, BYTE);
#endif
DRESULT disk_ioctl (BYTE, BYTE, void*);
```

第一个函数用来初始化 SD 卡；
第二个函数返回 0 就可以；
第三个和第四个函数是 SD 卡的写函数，把读写单扇区和多扇区函数放到里面就可以。
第五个函数用来获取 SD 卡的扇区大小等信息。把 GET_SECTOR_COUNT、GET_SECTOR_SIZE、GET_BLOCK_SIZE 这三个命令实现了即可。

然后，再加一个

```
DWORD get_fattime (void)
{
    return 0;
}
```

函数，这个函数放到 ffconf.h 或 ff.c 文件里都可以。这个函数是返回时钟的，由于 LPC1114 没有实时时钟功能，这里直接返回 0。

到这里。文件系统移植就成功了。不过，现在的文件系统还不支持长文件名。

计算机系的学生对短文件名和长文件名应该比较熟悉。

短文件名是指 8.3 格式的文件名。8 是指在点之前最多有 8 个字节，3 是指在点之后最多有 3 个字节。比如 ration.txt 就符合这个要求，richration.txt 就超标了。鉴于短文件名的缺点，后来才有了长文件名。关于长短文件名的历史和故事比较渊源！

为了让文件系统支持长文件名。需要修改 ffconf.h 中的参数。在 ffconf.h 文件中，找到这个：

```
#define  _USE_LFN  0        /* 0 to 3 */
```

把上面的_USE_LFN 改成 1，即可。

这时候，如果想让文件系统支持中文，还需要把 ffconf.h 文件中的_CODE_PAGE 参数修改成 936，如下所示：

```
#define _CODE_PAGE   936
```

然后在工程中添加 cc936.c 文件。这个文件在 option 文件夹中。

这时候，中文的长文件名也可以支持了，你写一个“中华人民共和国万岁.txt”文件都可以了。不过，这时，你编译后，会发现无法通过。原因是 LPC1114 的 RAM,ROM 实在是太小了。原来在 cc936.c 文件当中，存放着 GBK 和 Unicode

的相互转换表。这两张表可谓巨大呀！所以这个文件也要改动一下了。改动方法如下：

一．把两张装换表全部删除。（因为这两张表已经在 W25X16 中了）

二．修改一个函数。（转码函数）

三．完成。

简单吧！

在 cc936.c 文件当中，一共有两张表和两个函数，我们需要修改的函数如下：

```
WCHAR ff_convert (  /* Converted code, 0 means conversion error */
    WCHAR    src,   /* Character code to be converted */
    UINT     dir        /* 0: Unicode to OEMCP, 1: OEMCP to Unicode */
)
{
    WCHAR c;
    uint32 offset;          // W25X16 地址偏移
    uint8 GBKH,GBKL;     // GBK 码高位与低位
    uint8 unigbk[2];     // 暂存 GBK 高位与低位字节
    uint8 gbkuni[2];     // 暂存 UNICODE 高位与低位字节


    if (src < 0x80) {  /* ASCII */
        c = src;
    }
    else
    {
        if(dir == 0)    /* Unicode to OEMCP */
        {
            switch(src)
            {
                case 0x3001: c = 0xA1A2;break;      // 支持符号：  、  中文顿号
                case 0x300A: c = 0xA1B6;break;      // 支持符号：《
                case 0x300B: c = 0xA1B7;break;      //  支持符号:》
                case 0x201C: c = 0xA1B0;break;      // 支持符号：  “ 中文左双引号
                case 0x201D: c = 0xA1B1;break;      // 支持符号:”     中文右双引号
                case 0x2606: c = 0xA1EE;break;      //  支持符号：☆
                case 0x2605: c = 0xA1EF;break;      // 支持符号：  ★
                case 0x2018: c = 0xA1AE;break;      // 支持符号：‘ 中文左单引号
                case 0x2019: c = 0xA1AF;break;      //  支持符号:’中文右单引号
                case 0x3010: c = 0xA1BE;break;      // 支持符号：  【
                case 0x3011: c = 0xA1BF;break;      // 支持符号：  】
                case 0x3016: c = 0xA1BC;break;      //  支持符号:〖
                case 0x3017: c = 0xA1BD;break;      // 支持符号：  〗
                case 0x2299: c = 0xA1D1;break;      // 支持符号：⊙
                case 0x2116: c = 0xA1ED;break;      //  支持符号：№
```

```
                case 0x2236: c = 0xA1C3;break;      // 支持符号：  ：
                case 0x203B: c = 0xA1F9;break;      // 支持符号：※
                case 0x221E: c = 0xA1DE;break;      //  支持符号：∞
            default:
                if( (src > 0x4DFF) && (src < 0x9FA6) )// 汉字区
                {
                    offset = ((((uint32)src - 0x4E00) * 2) + 0x0C0000); /* 得到 W25X16 的 UTG 地址
*/
                    W25X16_Read(unigbk,offset,2); /* 获取 GBK 码 */
                    c = (((uint16)unigbk[0])<<8)+(uint16)unigbk[1];     /* 把 GBK 码给了 c */
                }
                else c = 0xA1A1;  // 如果是其它符号，都用符号：NULL 代替
                break;
            }
        }
        else if(dir == 1)   /* OEMCP to Unicode */
        {
            GBKH=(uint8)(src>>8);   // 获取 GBK 高位字节
            GBKL=(uint8)(src);      // 获取 GBK 低位字节
            GBKH-=0x81;
            GBKL-=0x40;
            offset=((uint32)192*GBKH+GBKL)*2;/* 得到 W25X16 的 GTU 地址 */
            W25X16_Read(gbkuni,offset+0x0D0000,2);    /* 获取 UNICODE 码 */
            c = (((uint16)gbkuni[1])<<8)+(uint16)gbkuni[0];     /* 把 UNICODE 码给了 c */
        }
    }
    return c;
}
```

这个函数的功能是，把 GBK 码换成 Unicode 码或把 Unicode 码换成 GBK 码，关于这两个表的用途，和这两个表在 W25X16 中的位置，你可以在第十八章《中文字库制作》里面找到。这里就不讲了。

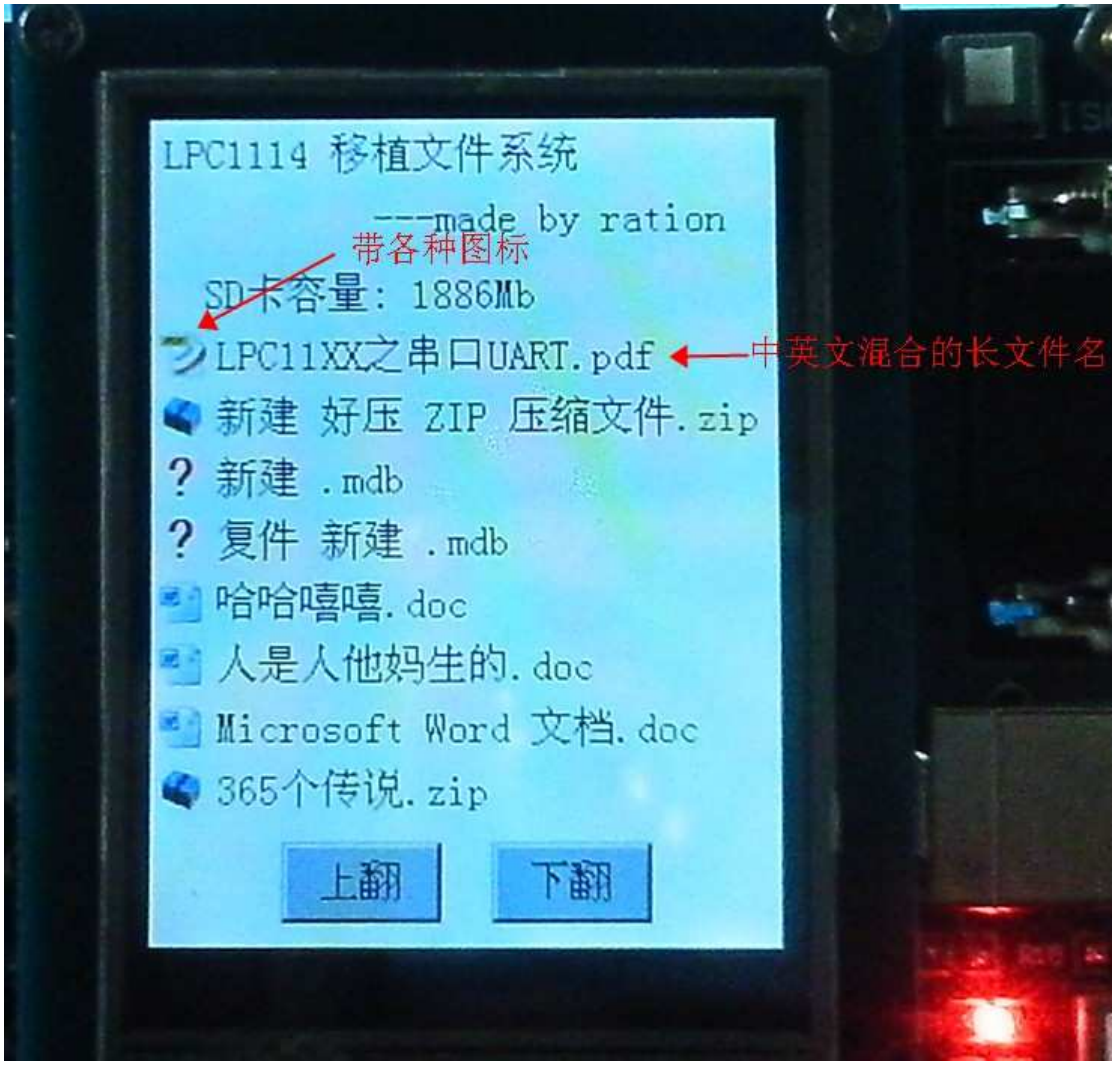
在上面的 GBK 转 Unicode 比较简单，直接从 W25X16 取值即可。

Unicode 转 GBK 码比较复杂一些，因为在我做的转换表当中，只有汉字的码，没有任何符号。所以为了在长文件名中支持一些常用符号，才有了上面的 swtich 结构。在这里，如果你有时间，可以把所有的中日韩符号加上去。不过我觉得还是加上一些常用的就可以了。如果是短文件名的话，支持所有的中日韩符号。因为上面这个函数是在文件系统读写长文件名的时候才会用到的。

三、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。在你的 SD 卡里面把 icon 文件夹拷进去，插好 SD 卡上电，即可以浏览你的 SD 卡上的文件。如果你的 SD 卡中文件比较多，可以通过“下翻”按钮查看。

注意：这里只做了“文件浏览”功能，如需进一步，你可以自己学会了以后，加入你想要的操作。



第十六章 基于 LPC1114 和 FatFs 的电子书

一、入门引导
二、程序设计
三、实验程序下载和使用说明

一、入门引导

电子书程序在小型手持设备上面的应用非常广泛。下面将基于 FatFs 文件系统做一个简易的电子书阅读程序。目前仅支持 TXT 文本的文档，不过，现在的 TXT 小说还挺多的。比如 www.txtbook.com.cn 上面就有好多的电子图书。话不多说了，说多了都是眼泪！看程序！

二、程序设计

程序主要是基于下面的函数：

```
/*******************************************/
/* 函数功能：txt 文本浏览                    */
/* 入口参数：fileName:文件名称，可带路径    */
/*******************************************/
FRESULT TXTViewer(const TCHAR *fileName)
{
    FATFS fs;                   // 建立一个文件系统
    FIL file;                   // 暂存文件
    UINT    br;                 // 字节计数器
    FRESULT res;                // 存储函数执行结果
    uint16 x=6,y=33;                // TFT 横纵坐标
    uint16  i=0;                // Buffer 计数器，在 512 范围内，表示已经显示了多少个字节
    uint8 zbuf[2];              // 双字节缓存
    uint8 tbuf[2];              // 中文半字节处理暂存

    f_mount(0,&fs);             // 加载文件系统
    res = f_open(&file, fileName, FA_OPEN_EXISTING|FA_READ); // 打开文件
    if(res != FR_OK) return res; // 如果没有正确打开文件,返回错误状态
    while(1)
    {
        res = f_read(&file, Buffer, 512, &br);// 读取文件内容，每次 512 个字节
        if(res||br==0)break; // 如果打开文件错误或者是已经读完了数据，跳出 while 循环

next:down_sign = 0;     // 清除文本下翻命令
        while(i<br)
        {
            while((Buffer[i]==13)&&(Buffer[i+1]==10))       //判断回车符和换行符
            {
                y=y+20;  //纵坐标换行，加行间距 4
                x=6;//横坐标
                i=i+2;    //跳过回车符和换行符（在文本文件中，回车符和换行符是同时出现的）
            }
```

```
while(y>265)  //纵坐标超出范围，换页
{
    if(down_sign==1)  //判断"下"按键
    {
        y=33;
        x=6;
        LCD_Fill(5,31,234,278,WHITE); // 清除原来的文本显示区
        goto next;          //继续显示
    }
    if(back_sign==1)goto re;     //判断"返回"键
}
zbuf[0]=Buffer[i];       //每两字节缓存
zbuf[1]=Buffer[i+1];
if(Buffer[i]>0x80)     // 如果是中文
{
    if(i==511)     // 最后一个字节处理
    {
        tbuf[0]=Buffer[i]; //扇区末尾半字节存储
        break;     // 跳出 while(i<br)循环
    }
    if(!tbuf[0])   // 如果没有进行过半字节处理
    {
        LCD_Show_hz(x, y, zbuf); //正常显示
        i=i+2;
        x+=16;
                            //横坐标加 16
    }
    else       // 如果进行过半字节处理
    {
        tbuf[1]=Buffer[i];  //另外半字节
        zbuf[0]=tbuf[0];
        zbuf[1]=tbuf[1];
        LCD_Show_hz(x, y, zbuf);
        i++;
        x+=16;
        tbuf[0]=0;     // 半字节处理清零
    }
    if(x>220)//横坐标超出范围
    {
        x=6;
        y+=20;
    }
}
```

```
                else //英文字符显示
                {
                    LCD_ShowChar(x, y, *zbuf);
                    i=i+1;
                    x+=8;
                    if(x>227)//横坐标超出范围，换行
                    {
                        x=6;
                        y+=20;
                    }
                }
            }
            i=0; //512 个数据显示完，i 清零
        }
        while(!back_sign); // 这条语句的作用是当读完了文本文件最后一页后，等待退出命令，否则你将看不到最后一页的内容了。
re: back_sign = 0;       // 清除退出标志
        f_close(&file);         // 关闭文件，必须和 f_open 函数成对出现
        f_mount(0,0);         // 卸载文件系统

        return FR_OK;
}
```

这是一个 txt 文本浏览函数，比如你想要阅读 SD 卡中的“ebook”文件夹中的“《坏蛋是怎样炼成的》.txt”文件，那么使用方法如下：

TXTViewer(“ebook/《坏蛋是怎样炼成的》.txt”);

执行完这个函数，即可打开第一页，打开第一页之后，将等待一个“进入下一页”的命令和“返回目录”的命令。这个命令可以设计到触摸屏上，也可以设计到电路板的按钮上，在我做的程序中，由于 TFT 上已经有了翻目录的两个按钮，再加上屏幕也不大，为了不再占用空间，让每一页显示更多的内容，我把“向下翻页”命令和“返回目录”命令设计到了电路板的按键 KEY1 和 KEY2 上。

函数中，还有一个需要注意的地方是，由于汉字是由两个字节组成的，所以当读 SD 卡 txt 文件的时候，就有可能在读完一页的时候，刚好读到汉字的半个字节，所以需要处理一下，否则会出现乱码和丢字的情况。在程序中已经做了处理，你可以研究研究。

再有一个注意的地方是，执行“返回目录”命令后，一定要关闭文件，并卸载了文件系统。否则容易出现“死机”现象。上面的“re”写到 f_colse()和 f_mount()函数之前，就是这个原因，千万不可以用 return 返回。

三、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。在你的 SD 卡里面把我已经做好的 ebook 文件夹放进去。然后插好 SD 卡，开机，即可看到 ebook 里面的 txt 文件名称。因为我们移植的文件系统已经支持长文件名了，所以，就不必担心你的文件名很长了。而且还支持书名号。

点击 TFT 上的“目录上翻”和“目录下翻”按钮，可以浏览所有的在 ebook 文件夹下的所有 txt 文件。

直接点击一下对应的文本文件，就会进入阅读模式。

按键 KEY1 的功能是下翻页。

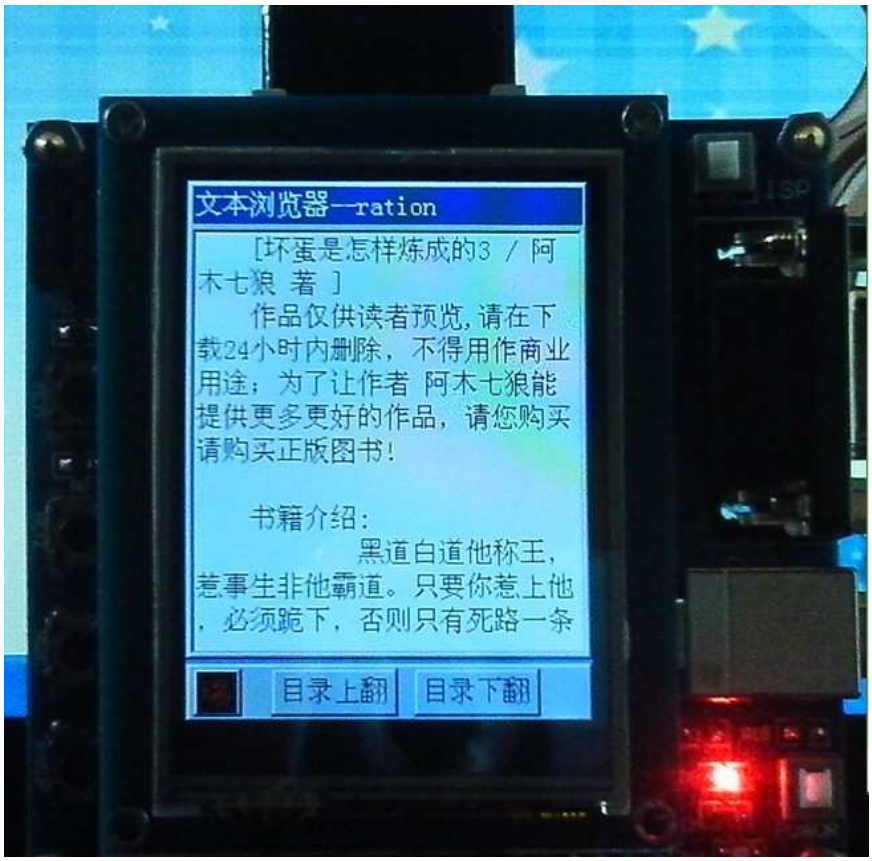
按键 KEY2 的功能是返回目录。

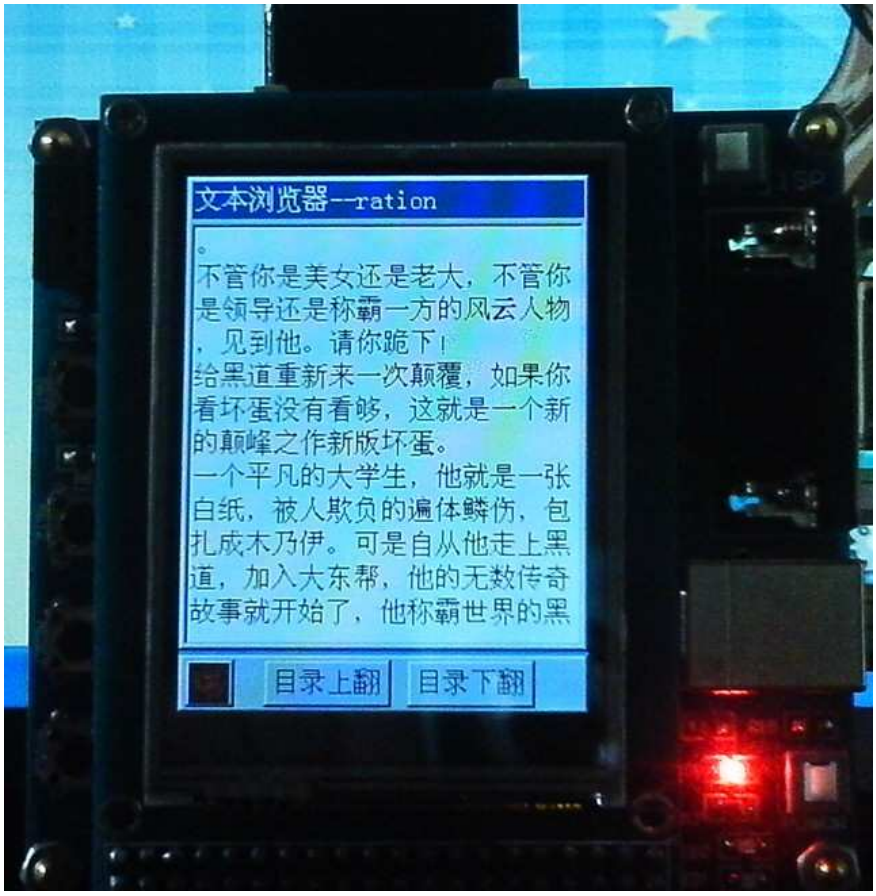
在浏览目录的时候，按键 KEY2 和 KEY1 不起作用。

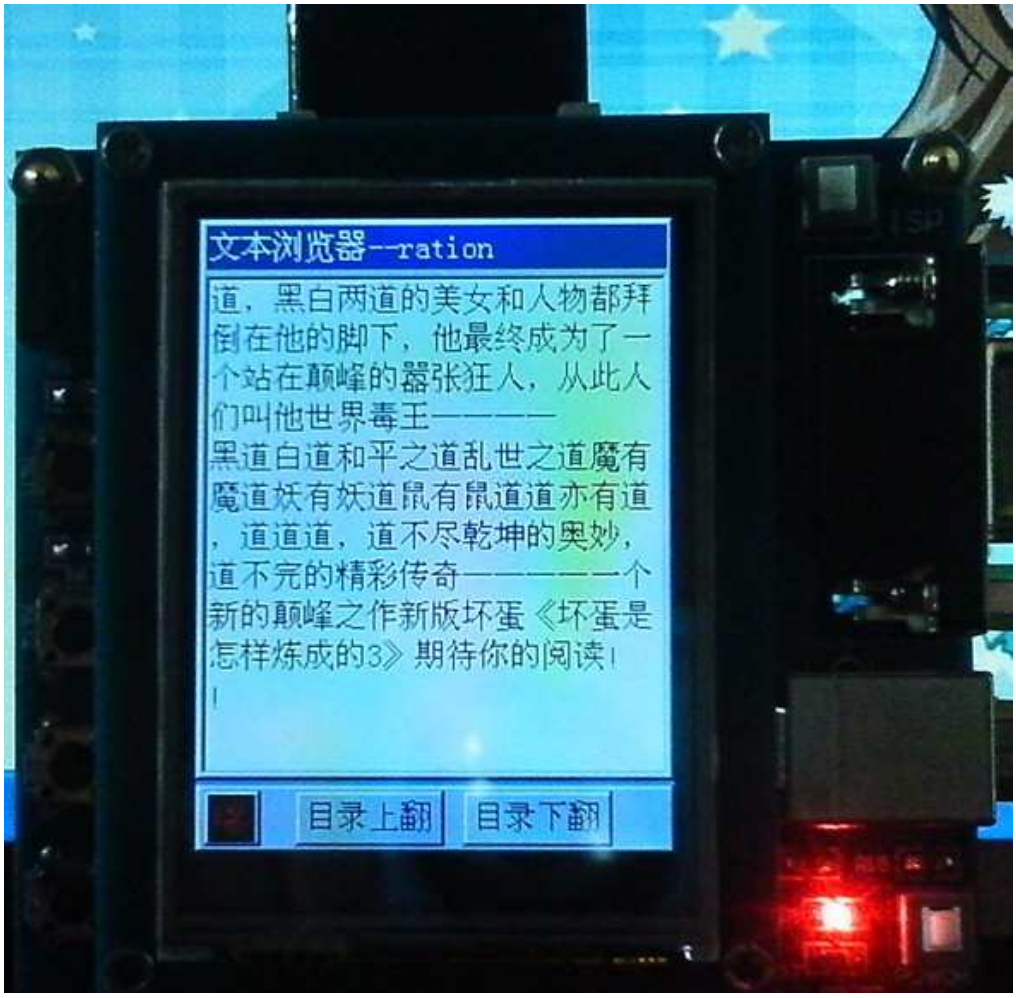
在文本阅读的时候，TFT 上的“目录上翻”和“目录下翻”不起作用。

在返回目录的时候，显示的目录是第一页的目录。

上面实现的功能并不多，所以叫做简易文本浏览器。


文本浏览器--ration
[坏蛋是怎样炼成的3 / 阿木七狼 著]
作品仅供读者预览，请在下载24小时内删除，不得用作商业用途；为了让作者 阿木七狼能提供更多更好的作品，请您购买请购买正版图书！
书籍介绍：
黑道白道他称王，惹事生非他霸道。只要你惹上他，必须跪下，否则只有死路一条
目录上翻
目录下翻


文本浏览器--ration
。
不管你是美女还是老大，不管你是领导还是称霸一方的风云人物，见到他。请你跪下！
给黑道重新来一次颠覆，如果你看坏蛋没有看够，这就是一个新的颠峰之作新版坏蛋。
一个平凡的大学生，他就是一张白纸，被人欺负的遍体鳞伤，包扎成木乃伊。可是自从他走上黑道，加入大东帮，他的无数传奇故事就开始了，他称霸世界的黑
目录上翻
目录下翻


文本浏览器--ration
道，黑白两道的美女和人物都拜倒在他的脚下，他最终成为了一个站在颠峰的嚣张狂人，从此人们叫他世界毒王————
黑道白道和平之道乱世之道魔有魔道妖有妖道鼠有鼠道道亦有道，道道道，道不尽乾坤的奥妙，道不完的精彩传奇————一个新的颠峰之作新版坏蛋《坏蛋是怎样炼成的3》期待你的阅读！！
目录上翻
目录下翻

第十七章 基于 LPC1114 和 FatFs 的数码相框

一、入门引导
二、程序设计
三、实验程序下载和使用说明

一、入门引导

数码相框已经悄悄的走进人们的生活。如果自己能够做个数码相框，然后把想要显示的照片放上去。看着它们在上面来回循环显示，这种感觉很好！市场上卖的数码相框大都是 8 寸的液晶屏。我们这里将实现在 2.4TFT 上显示一幅一幅的图片。效果绝对的好。画面相当的细腻！

我们在这里做一个显示 BMP 图片的数码相框程序。经过实测，显示一幅全屏的图片，需要大约半秒钟左右。如果来回循环播放图片的话，这个时间就变成一种“效果”了。

关于 BMP 图片的格式存储，你可以去百度搜一下。我这里就不做介绍了。

二、程序设计

本质上是对 BMP 图片文件的解码。

```
/*************************************************/
/* 函数功能：获取 BMP 图片的头文件信息                    */
/* 入口参数：buf：暂存缓存                              */
/*************************************************/
BMP_HEADER TFTBmpGetHeadInfo(uint8 *buf)
{
    BMP_HEADER bmpHead;

    bmpHead.bfType = (buf[0] << 8) + buf[1];            // BM
    bmpHead.bfSize    = (buf[5]<<24) + (buf[4]<<16) + (buf[3]<<8) + buf[2];     // 文件大小
    bmpHead.biWidth = (buf[21]<<24) + (buf[20]<<16) + (buf[19]<<8) + buf[18]; // 图像宽度
    bmpHead.biHeight = (buf[25]<<24) + (buf[24]<<16) + (buf[23]<<8) + buf[22];// 图像高度
    bmpHead.biBitCount = (buf[29] << 8) + buf[28];     // 每个像素的位数，单色位图为 1，256 色为 8，16bit 为 16， 24bit 为 24

    return bmpHead;
}
/*************************************************/
/* 函数功能：显示 BMP 图片                             */
/* 入口参数：x,y: 坐标                                */
/*           *bmpName:bmp 图片的名称，可以带路径     */
/*************************************************/
uint8 TFTBmpDisplay(uint8 *bmpName,uint16 x,uint16 y)
{
    FATFS fs;            // 建立文件系统
    FIL file;            // 建立文件
    UINT    br;          // 字节计数器
    FRESULT res;         // 返回值信息
    BMP_HEADER bmpHead;      // 头信息
    uint16 i;
```

```
    f_mount(0, &fs);   // 挂载文件系统
    res = f_open(&file, (const TCHAR *)bmpName, FA_OPEN_EXISTING|FA_READ); // 打开 BMP 文件并读取到 file 中
    if(res != FR_OK)
    {
        return res;
    }
    else
    {
        res = f_read(&file, Buffer, 54, &br);       // 读取头文件信息
        if(res != FR_OK)
        {
            return res;           // 返回错误表示
        }
        else
        {
            bmpHead = TFTBmpGetHeadInfo(Buffer); // 获取头信息

            if (bmpHead.bfType == 0x424D) // 判断是否为 BMP 图像
            {
                LCD_WR_REG_DATA(0x0003, 0x1010); // 由下而上显示
                LCD_XYRAM(x, y, x+bmpHead.biWidth-1, y+bmpHead.biHeight-1);
                LCD_WR_REG_DATA(0x0020,x);//设置 X 坐标位置
                LCD_WR_REG_DATA(0x0021,y+bmpHead.biHeight-1);//设置 Y 坐标位置(注意：在由下而上显示的时候，这里 y 坐标应该是最下边的值)
                LCD_WR_REG(0x0022);     //指向 RAM 寄存器，准备写数据到 RAM
                while(1)
                {
                    res = f_read(&file, Buffer, 240, &br);    //读取 240 个数据
                    if(res||br==0)
                                            //错误跳出
                    break;
                    for(i=0;i<80;i++)
                    {
                        // 在 TFT 上显示一个像素点的颜色
                        LCD_WR_DATA(((Buffer[i*3+2]/8)<<11   |   (Buffer[i*3+1]/4)<<5|(Buffer[i*3]/8)));
                    }
                }
                LCD_WR_REG_DATA(0x0003, 0x1030); // 恢复正常显示
                LCD_XYRAM(0, 0, 239, 319);           // 恢复 GRAM
            }
        }
```

```
    }

    f_close(&file);          // 关闭文件，必须和 f_open 函数成对出现
    f_mount(0, 0);          // 卸载文件系统

    return FR_OK;            // 返回成功标志
}
```

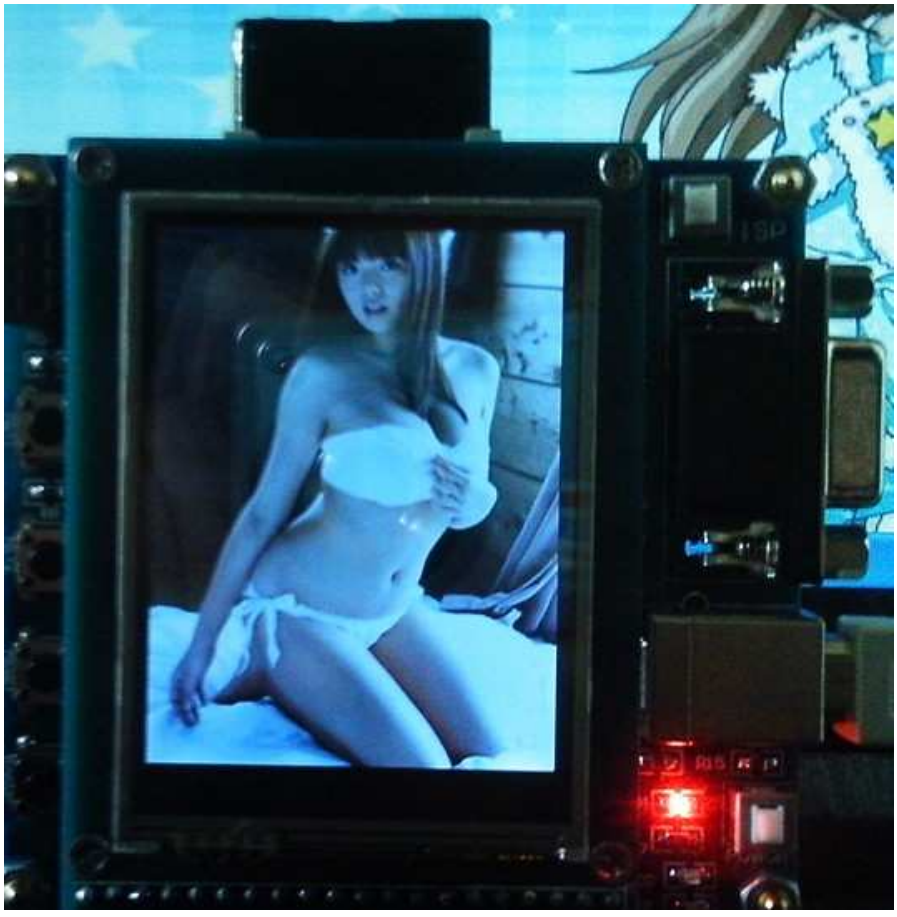
上面的第一个函数获取 BMP 的信息，完整的信息包括 54 个字节。这里只取了有用的。

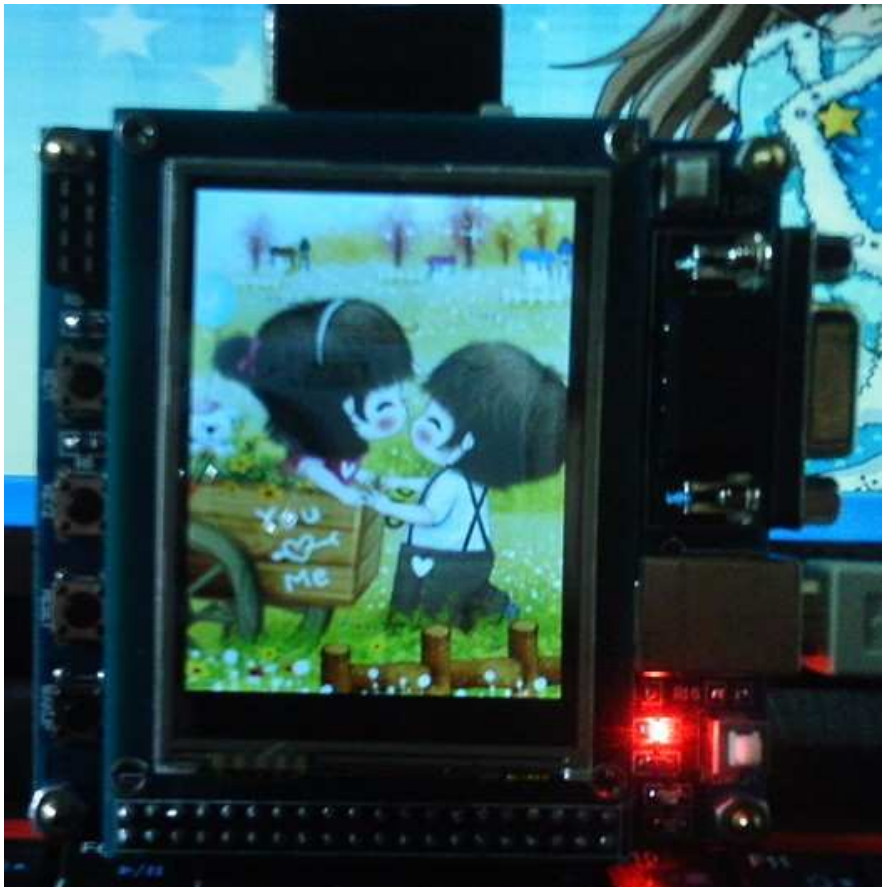
第二个函数即是对 BMP 图片的解码。这里只做了对 24 位图片的解码。

三、实验程序下载和使用说明

按照《MINI LPC1114 程序下载说明》的方法，把此章对应的程序 HEX 文件下载到学习板上。在你的 SD 卡里面把我已经做好的 picture 文件夹放进去。这里面是我做好的示例图片，都是 240*320 大小的 24 位 BMP 图片。

要想显示你自己的图片，要记住把图片搞成 240*320 范围之内，只能小，不能大。然后把格式转换成 24 位的 BMP 图片。用 PS 和画图工具两个软件就可以了。要确保转换完成的图片是真正的 BMP 24 位图片。BMP 图片的特点是每个点都有颜色值，所以一定像素大小对应的占用空间也是一定的。比如一张 240*320 大小的 24 位 BMP 图片大小即为 225KB。




YOU
ME

第十八章 中文字库制作

一、入门导读

在显示器上显示汉字和显示英文字母的原理都是需要对应的点阵。难易程度相差很大。

英文的单词都是由 26 个字母构成了，加上大小写的区别和一些字符，也不过才 95 个。假如要显示 8*16 像素大小的字符，每一个字符需要 16 个字节的字库空间，95 个字符即是 95*16=570 个字节。即占用 570 个字节的 RAM。对于 LPC1114 的 8K 字节 RAM 来说，搓搓有余了。

显示中文的话，必须需要每一个字的字库，16*16 像素大小的中文，每一个中文都要 32 个字节。GBK 收录了中文两万多个，如果要都能正常显示，需要 700 多 K 字节的空间。

所以，我们选择了把这些字库放在外部存储器当中，在开发板上，我们选择了 2M 的 FLASH 存储芯片 W25X16 做为存储媒介。放个 700 多 K 的字库足够了，如果你愿意，同时放两种字体的字库都没问题。

GBK 字库是当今使用最广泛的中文字库，里面包含所有的简体中文，繁体中文，中日韩标点符号。显示任何字都没有问题。

假如我们想在液晶显示器上显示一个 16*16 像素大小的汉字“瑞”，怎么办呢？先看一下“瑞”字在液晶显示器上被放大的效果：

上面的这个瑞即是 16*16 像素大小的字体，你可以数一下上面的像素点。你可以看一下，在上面“亮”的地方用“写点函数”写成和其它点不同的颜色，不就可以了吗！按照这个思想，我们就需要采用一种有效的办法了。从上面图像的右上角开始扫描，每行有 16 个点，即可以用两个字节来表示，“暗”的地方用 0 表示，亮的地方用 1 表示，那么第一行就可以写成十六进制的 00 20，以此类推，可以得出第二行，第三行的字节码，如此一来，一个汉字就需要 32 个字节来表示。搞好了汉字的这些码，在程序中给 TFT 开个 16*16 大小的区域，然后开始扫描这些字节，遇到 0，写背景色，遇到 1，写定义的颜色。汉字显示就是如此了。

那么，我们不可能自己一个一个的把汉字编码。这就需要一个软件。可以把所有的 GBK 字十几秒就变成了字库码。

这个软件可以把整个 GBK 字库在十几秒钟的时间取模完毕。取模完了以后，可以通过 ziku.exe 软件变成二进制码，这时候，就可以下载到 W25X16 了。后面有详细的下载方法。

W25X16 是一个 2M 字节的 FLASH 存储芯片，地址范围：0x000000~0x1FFFFF
一共 32 个 BLOCK，每个 BLOCK 大小 64K
一共 512 个 SECTOR，每个 SECTOR 大小 4K
关于 W25X16 的详细资料，请看 ration 翻译的“W25X16 中文手册”。

二、字库排布

在 MINI LPC1114 V1.6 开发板上的 W25X16 存放着一个 16*16 像素的宋体 GBK 字库，还有两张 Unicode 码和 GBK 码的互相转换的表。

对应的 BLOCK	占用范围	对应内容
BLOCK0~BLOCK11	0x000100—0x0BD100	GBK 字库
BLOCK12	0x0C0000—0x0CA34C	uni2gbk 转换表
BLOCK13	0x0D0000—0x0DBD00	gbk2uni 转换表
BLOCK14~BLOCK31	没有占用这里哦	空（即都为 0xFF）

三、字库用途

GBK 字库：用来显示汉字，可以说是现在最全的汉字库，包括繁体字，各种中日韩符号。
Uni2GBK：这张表是在上了文件系统后，读取 SD 卡中的文件名用的。不上文件系统用不着。
GBK2Uni：这张表是在上了文件系统后，在 SD 卡上写中文文件名或者是在读取以中文命名的文件内容时用的。不上文件系统用不着。

四、各类文件说明

- stziku16.bin 和 htziku16.bin　这个是已经做好的宋体 16 像素大小 GBK 字库和黑体 16 像素大小 GBK 字库，是用此文件夹里的 ziku.exe 和 gbk_ziku.txt 文件和 Mold.exe 制作的。这两个 exe 程序都是 ourdev 的阿莫制作的。

- gbk2uni.sys　这个是已经做好的 GBK 转 Unicode 的表，是用 gbk_ziku.txt 文件和 WINHEX 软件制作的。原理：用 WINHEX 软件把对应的汉字转换成 Unicode 码保存。

- uni2gbk.sys　这个是已经做好的 unicode 转 GBK 的表，实际上它就是 unicode_ziku.txt 文件，你可以把这个文件的后缀 sys 改为 txt，然后打开看看。这张表的原理是把 GBK 汉字按照 unicode 码的顺序排放。这张表的缺点是只有汉字，没有任何符号。所以在文件系统下显示 SD 卡中的带“非英文符号”中文文件名时，就需要做相应的处理。我已经在程序里面加入了 switch 结构，你可以任意添加想要支持显示的各类符号，你只需增加 case 即可，有必要的话，即使把所有的中日韩符号添加进来也可以。如下图所示，我已经把常用的符号添加进去了。

```
case 0x3001: c = 0xA1A2;break;     // 支持符号：、  中文顿号
case 0x300A: c = 0xA1B6;break;     // 支持符号：《
case 0x300B: c = 0xA1B7;break;     // 支持符号：》
case 0x201C: c = 0xA1B0;break;     // 支持符号：“  中文左双引号
case 0x201D: c = 0xA1B1;break;     // 支持符号：”  中文右双引号
case 0x2606: c = 0xA1EE;break;     // 支持符号：☆
case 0x2605: c = 0xA1EF;break;     // 支持符号：★
case 0x2018: c = 0xA1AE;break;     // 支持符号：‘中文左单引号
case 0x2019: c = 0xA1AF;break;     // 支持符号：’中文右单引号
case 0x3010: c = 0xA1BE;break;     // 支持符号：【
case 0x3011: c = 0xA1BF;break;     // 支持符号：】
case 0x3016: c = 0xA1BC;break;     // 支持符号：〖
case 0x3017: c = 0xA1BD;break;     // 支持符号：〗
case 0x2299: c = 0xA1D1;break;     // 支持符号：⊙
case 0x2116: c = 0xA1ED;break;     // 支持符号：№
case 0x2236: c = 0xA1C3;break;     // 支持符号：∶
case 0x203B: c = 0xA1F9;break;     // 支持符号：※
case 0x221E: c = 0xA1DE;break;     // 支持符号：∞
```

- GBK_Proj.hex\U2G.hex\G2U.hex 文件是 LPC1114 上下载的程序，用来把字库和转换码表下载到 W25X16 里面或者是单独擦除对应的区域。这三个文件提供源程序。具体使用方法可以用 KEIL 打开源程序文件后，在 main.c 文件的最上方看到或者是看 MINI LPC1114 用户手册。

五、显示汉字的程序编写

```
uint8 buf[32]; //用于存放 16*16 点阵汉字数据，一个汉字 32 个字节

/*********************************************/
/* 函数功能：从 W25X16 中提取点阵码                */
/* 入口参数：code:GBK 码第一个字节                 */
/*           dz_data:存放点阵码的数组              */
/*********************************************/
void Get_GBK_DZK(uint8 *code, uint8 *dz_data)
{
    uint8 GBKH,GBKL;    // GBK 码高位与低位
    uint32 offset;      // 点阵偏移量

    GBKH=*code;
    GBKL=*(code+1);     // GBKL=*(code+1);
    if(GBKH>0XFE||GBKH<0X81)return;
    GBKH-=0x81;
    GBKL-=0x40;
    offset=((uint32)192*GBKH+GBKL)*32;//得到字库中的字节偏移量
    W25X16_Read(dz_data,offset+0x100,32);
    return;
}
/*********************************************/
/* 函数功能：显示 16*16 点阵中文                   */
/* 入口参数：x,y :起点坐标                         */
/*           *hz：汉字                             */
/*********************************************/
void LCD_Show_hz(uint16 x,uint16 y,uint8 *hz)
{
    uint8 i,j,temp;
    uint8 dz_data[32];

    Get_GBK_DZK(hz, dz_data);

    LCD_WR_REG_DATA(0x0020,x); //设置 X 坐标位置
    LCD_WR_REG_DATA(0x0021,y); //设置 Y 坐标位置
    /*开辟显存区域*/
    LCD_XYRAM(x,y,x+15,y+15);  // 设置 GRAM 坐标
    LCD_WR_REG(0x0022);        //指向 RAM 寄存器，准备写数据到 RAM

    for(i=0;i<32;i++)
    {
```

```
            temp=dz_data[i];
            for(j=0;j<8;j++)
            {
                if(temp&0x80)LCD_WR_DATA(POINT_COLOR);
                else LCD_WR_DATA(BACK_COLOR);
                temp<<=1;
            }
        }

        /* 恢复显存显示区域 240*320 */
        LCD_XYRAM(0x0000 ,0x0000 ,0x00EF ,0X013F); // 恢复 GRAM 整屏显示

        return;
}
```

每一个汉字都对应着一个唯一的 GBK 码，对应两个字节。比如汉字“瑞”的 GBK 码是十六进制的 C8 F0，你可以通过 ASCII.exe 软件查到，如下所示：

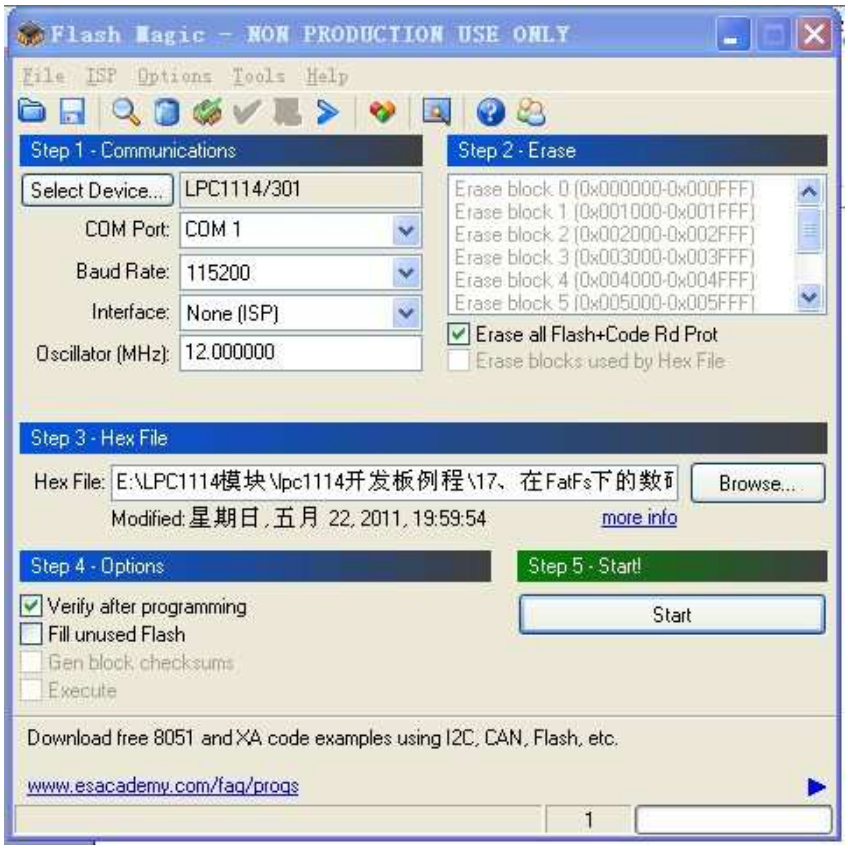
上面的函数 void Get_GBK_DZK(uint8 *code, uint8 *dz_data)用来从 W25X16 中取出一个 32 字节的数据

再用 void LCD_Show_hz(uint16 x,uint16 y,uint8 *hz)这个函数，把读出来的 32 个字节数据显示到 TFT 上，即为一个汉字。

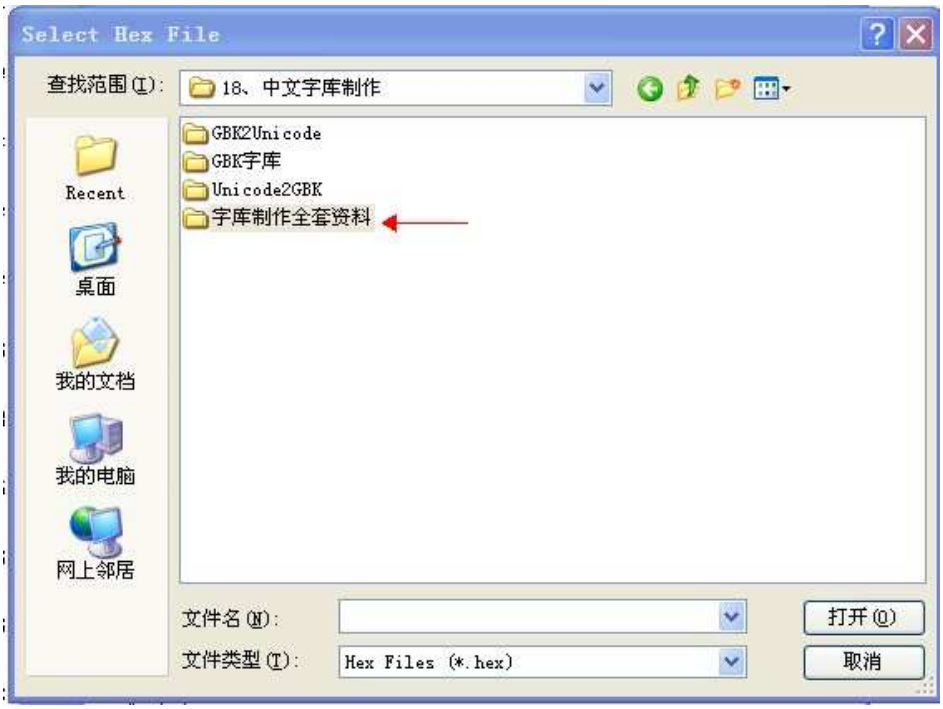
六、图解中文字库下载流程

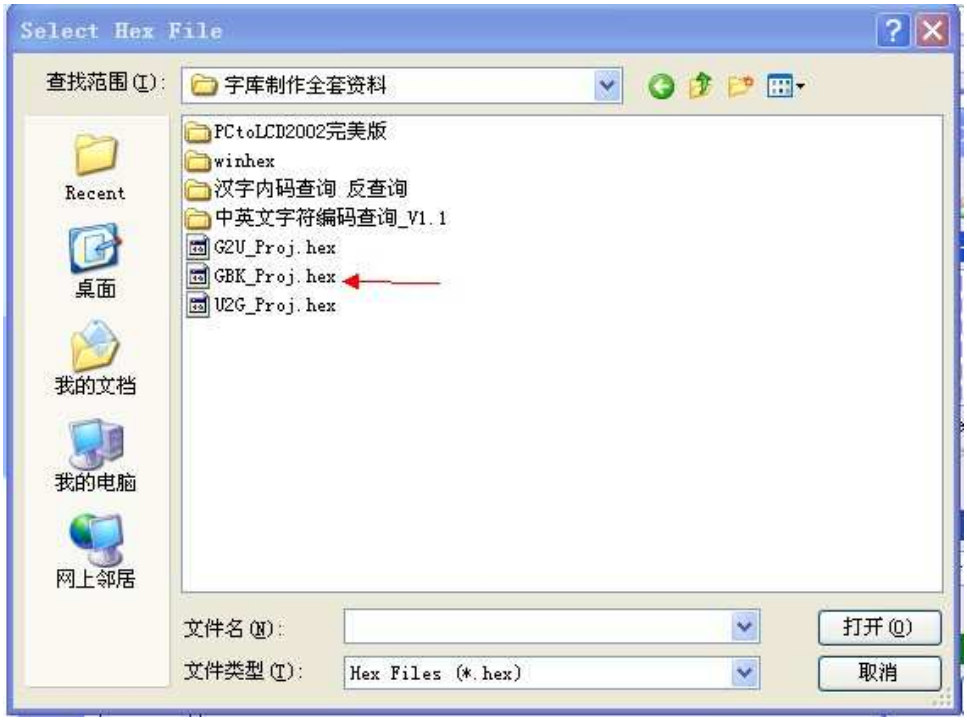
①下载 GBK 字库

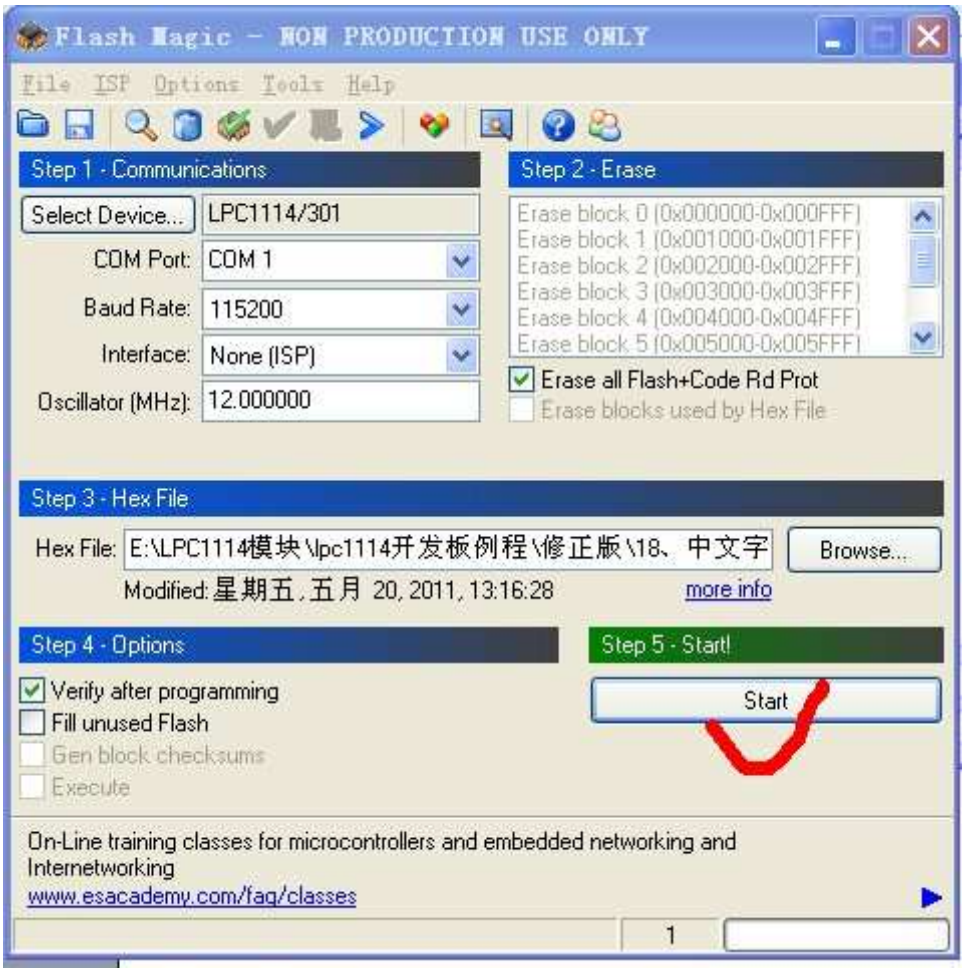
第一步，打开 FLASH Magic



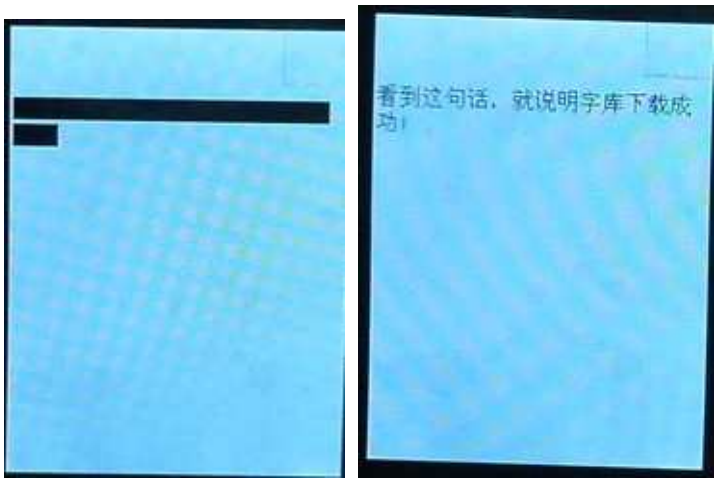
第二步：点击 Browse 按钮，选择要下载的文件




Select Hex File
查找范围(I):
字库制作全套资料
PCtoLCD2002完美版
winhex
汉字内码查询 反查询
中英文字符编码查询_V1.1
G2U_Proj.hex
GBK_Proj.hex
U2G_Proj.hex
Recent
桌面
我的文档
我的电脑
网上邻居
文件名(N):
文件类型(T):
Hex Files (*.hex)
打开(O)
取消


Flash Magic - NON PRODUCTION USE ONLY
File ISP Options Tools Help
Step 1 - Communications
Select Device...
LPC1114/301
COM Port:
COM 1
Baud Rate:
115200
Interface:
None (ISP)
Oscillator (MHz):
12.000000
Step 2 - Erase
Erase block 0 (0x000000-0x000FFF)
Erase block 1 (0x001000-0x001FFF)
Erase block 2 (0x002000-0x002FFF)
Erase block 3 (0x003000-0x003FFF)
Erase block 4 (0x004000-0x004FFF)
Erase block 5 (0x005000-0x005FFF)
Erase all Flash+Code Rd Prot
Erase blocks used by Hex File
Step 3 - Hex File
Hex File:
E:\LPC1114模块\lpc1114开发板例程\修正版\18、中文字
Browse...
Modified: 星期五, 五月 20, 2011, 13:16:28
more info
Step 4 - Options
Verify after programming
Fill unused Flash
Gen block checksums
Execute
Step 5 - Start!
Start
On-Line training classes for microcontrollers and embedded networking and Internetworking
www.esacademy.com/faq/classes
1

第三步：按键 KEY2。



如果出现上面左面的图示，代表 W25X16 中还没有字库，如果出现右边这个，即代表 W25X16 中已经有字库了。此时，按键 KEY1 可以擦除字库，擦除时间需要几秒钟，按键 KEY1，后开始擦除，同时 LED1 亮起，擦除完毕之后，LED1 将灭。擦除了以后，再按键 KEY2 即会出现上面左图的效果。

第四步：打开串口调试助手，设置串口和波特率，把波特率改成 115200

第五步：点击“选择发送文件”

把上图打对勾的地方变成“所有文件”

根据个人喜好，选择宋体 GBK 字库或者黑体 GBK 字库，单击“打开”。如上图。

然后点击“发送文件”，等待 1 分零 5 秒钟。等待期间，不要操作开发板上的任何按钮。如下图所示，正在发送。

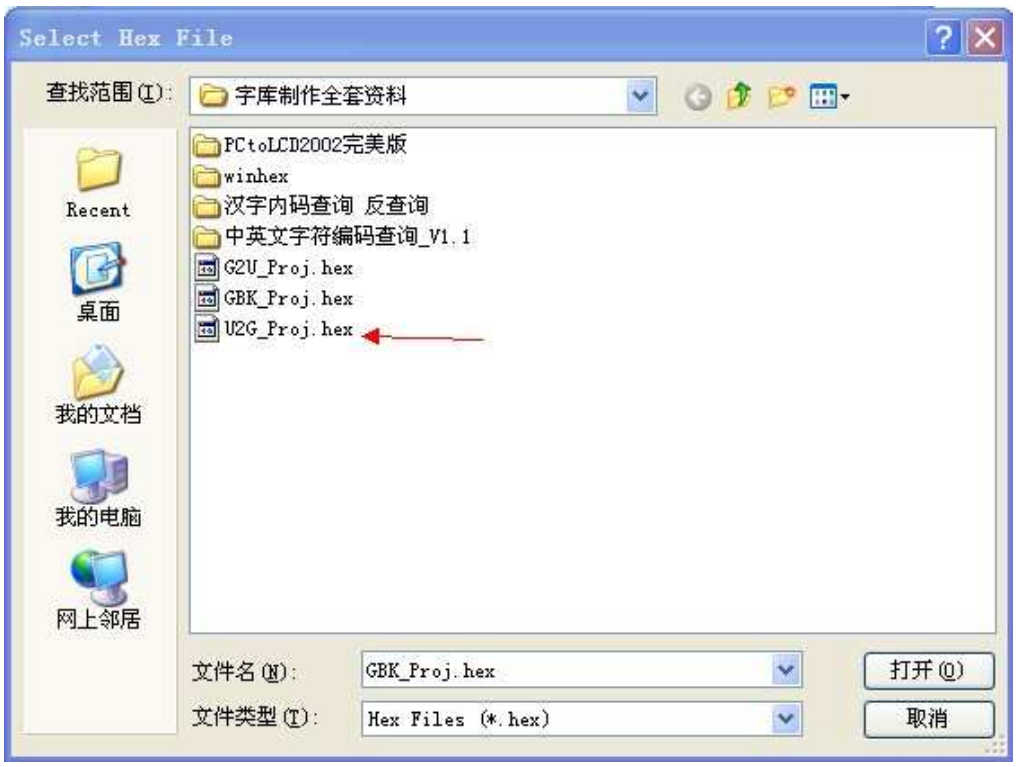
发送完以后，会有提示：

这时候，再按键 KEY2（千万不要按成 KEY1 哦），即会出现一行中文提示。代表字库下载成功。

②下载 Unicode2gbk 字库

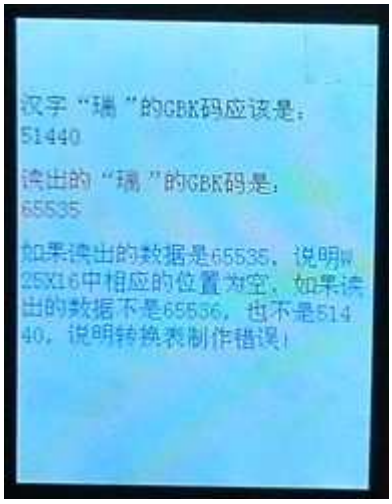
执行过程大体相同。

第一步：选择 HEX 文件到 LPC1114。



下载时，注意先关闭串口调试助手，

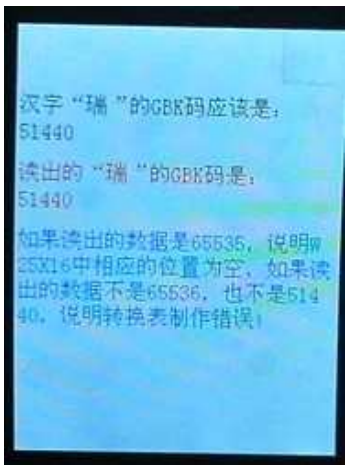
第二步：按键 KEY2



如上所示，如果读出的 GBK 码不对，代表没有转换表在里面，如果正确，则已经下载过转换表了。这时，按键 KEY1 会擦除这个转换表对应的存储区域。不会擦除 GBK 字库对应的区域，请放心！

第三步：选择要下载的文件

按照上面下载 GBK 的步骤，点击“发送文件”，经过 2 秒钟左右即可下载成功。下载完以后，再按键 KEY2，即可得到正确的 GBK 码，如下所示：



③下载 GBK2Unicode

执行过程如上，这里就不讲了。

第十九章 W25X16 中存放图片及其显示

一、W25X16 中存放图片的步骤

二、显示 W25X16 中存放的图片

一、W25X16 中存放图片的步骤

原理：先用解码软件生成二进制 BIN 文件，再利用串口下载到 W25X16 中。

步骤一：利用软件生成图片二进制 BIN 文件

1. 双击图标，打开软件。

2. 点击“打开”选择图片

3. 确定配置，依据下图配置


Image2LCD
打开
保存
设置
重新载入
上一幅
下一幅
帮助
关于
输出数据类型：
二进制(*.bin)
扫描模式：
水平扫描
输出灰度：
16位真彩色
最大宽度
最大高度
240
320
包含图像头数据
字节内象素数据反序
自右至左扫描
自底至顶扫描
高位在前(MSB First)
恢复缺省值
颜色反转
正常显示
亮度：
对比度：
输出图像调整
256色
4096色
16位彩色
24位彩色
32位彩色
输入图像：mm14.bmp (240,320)
输出图像：(240,320)

4. 点击“保存”

5. 保存到你所指定的文件夹里面。

到了这步，图片二进制数据就制作好了。

二、显示 W25X16 中存放的图片

原理：依据图片的长和宽，开 TFT 显示内存。然后循环从 W25X16 中读出数据塞入显存。

注意：W25X16 中的字库占了前半部分。所以图片数据最好从 0x0F0000 开始存放。关于 W25X16 中字库存放结构，可以看光盘中“字库制作全套资料”文件夹里面的“开发板 W25X16 字库存放结构.pdf”文件。

给 W25X16 中写数据要确保对应区域为空，如果之前存放过数据，一定要记得擦除以后，才能存放新的数据。

程序请看光盘文件中的“例程源代码”第十九。

第二十章 W25X16 中存放 ASCII 字库及其显示

这里所说的 ASCII 字库是指 ASCII 码位于 0x20~0x7E 的 95 个英文大小写字母和符号。

一般情况下，我们只是把这些 ASCII 码生成 16 进制的字库文件后，放在一个数组中显示，如果要显示 8*16 大小的字母或符号的话，需要 16*95=1520 个字节。即需要占用大约 1.5K 的内存空间。如果要显示更大的字母和符号，占用的内存空间就更大了。

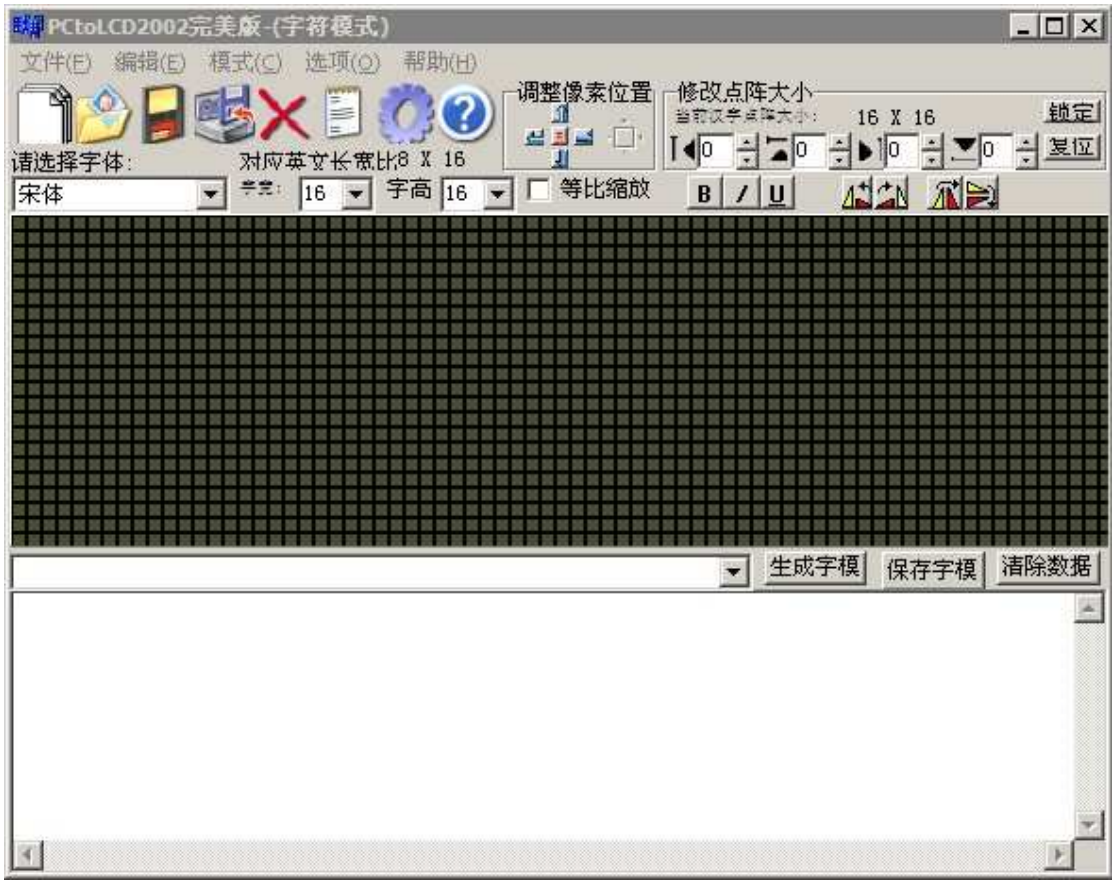
有些时候，我们需要节省内存空间做别的事情的话，就可以把这个点阵字库文件放到外部存储器中。在我们的开发板上，就可以放到 W25X16 中，显示的速度和放在内存中相比，凭人的眼睛是根本区别不出来的。

要把这些点阵字库放入 W25X16，就需要生成二进制的字库文件。所以我们需要用到下面这个软件。

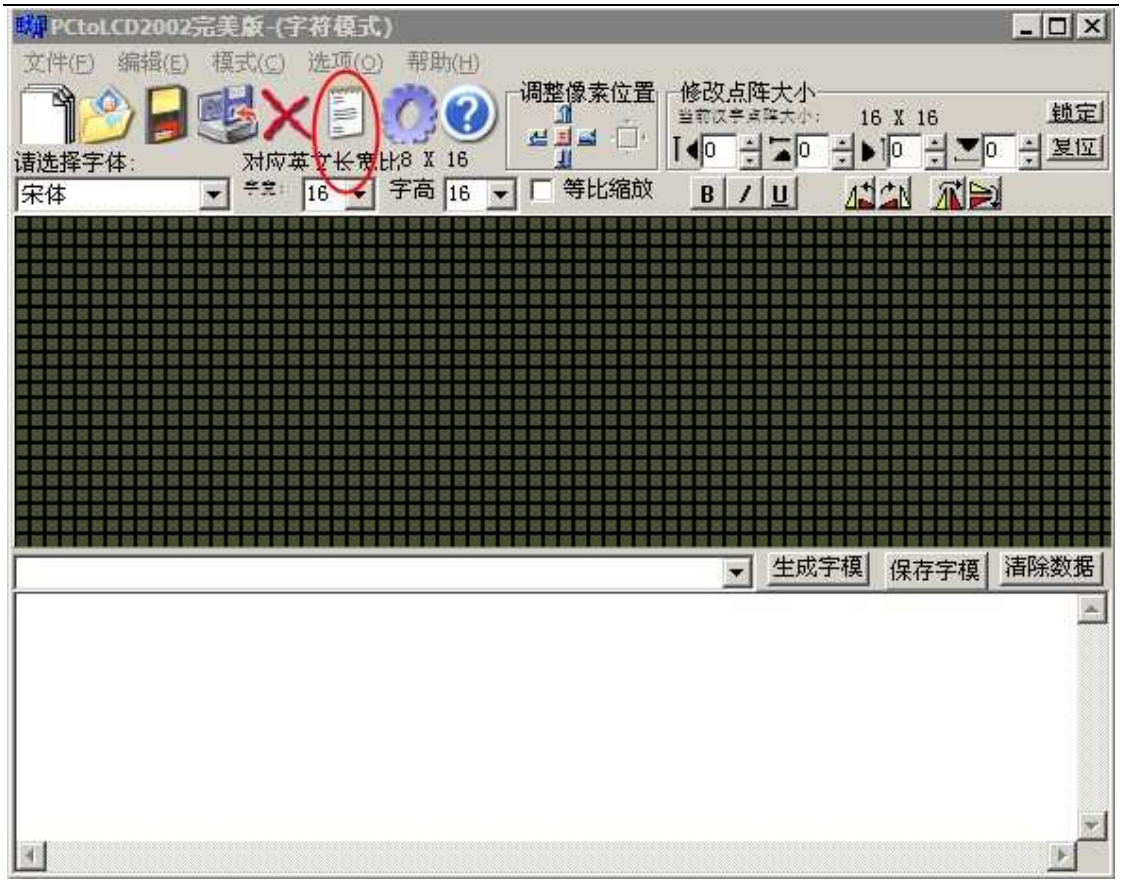
和下面这个字库文件

步骤如下：

一、打开软件



二、点击下图中被椭圆圈起来的图标



三、弹出窗口后，按照如下配置

四、点击下图中被椭圆圈起来的图标

五、选择 ASCII 码文件后打开，如下图所示

六、点击下图中被椭圆圈起来的图标

七、选择你要保存的地点后保存，到这里就生成了 ASCII 字库的二进制点阵文件。

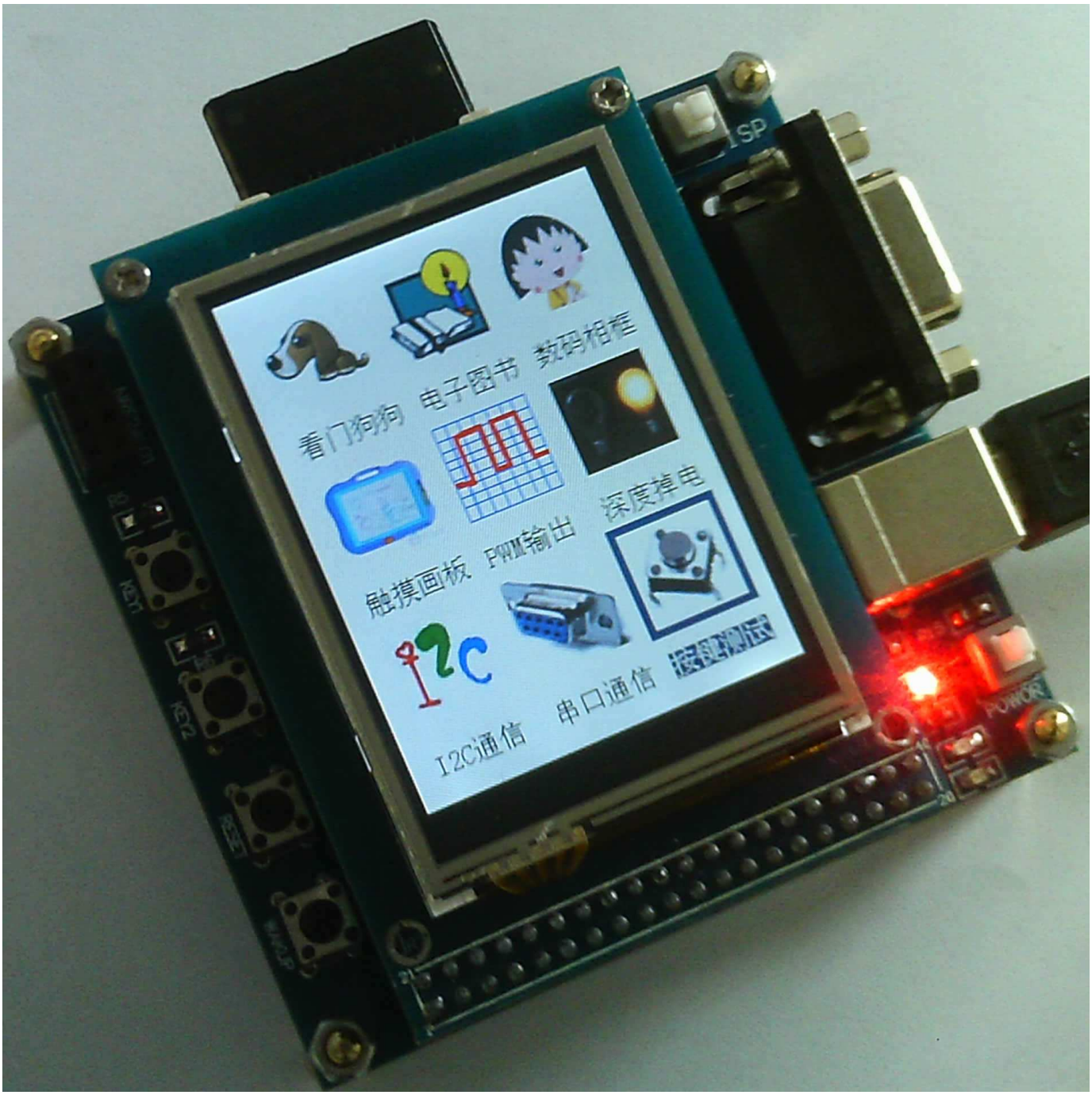
注意：

点击下图中被椭圆圈起来的的图标可以选择你所想要的取模方式

下载这个二进制文件的方法和下载其它字库的方法一样，请参照相关说明。

显示程序的编写原理和显示 GBK 汉字的显示原理一样。详见源程序！

第二十一章 综合实验---用手持方式实现



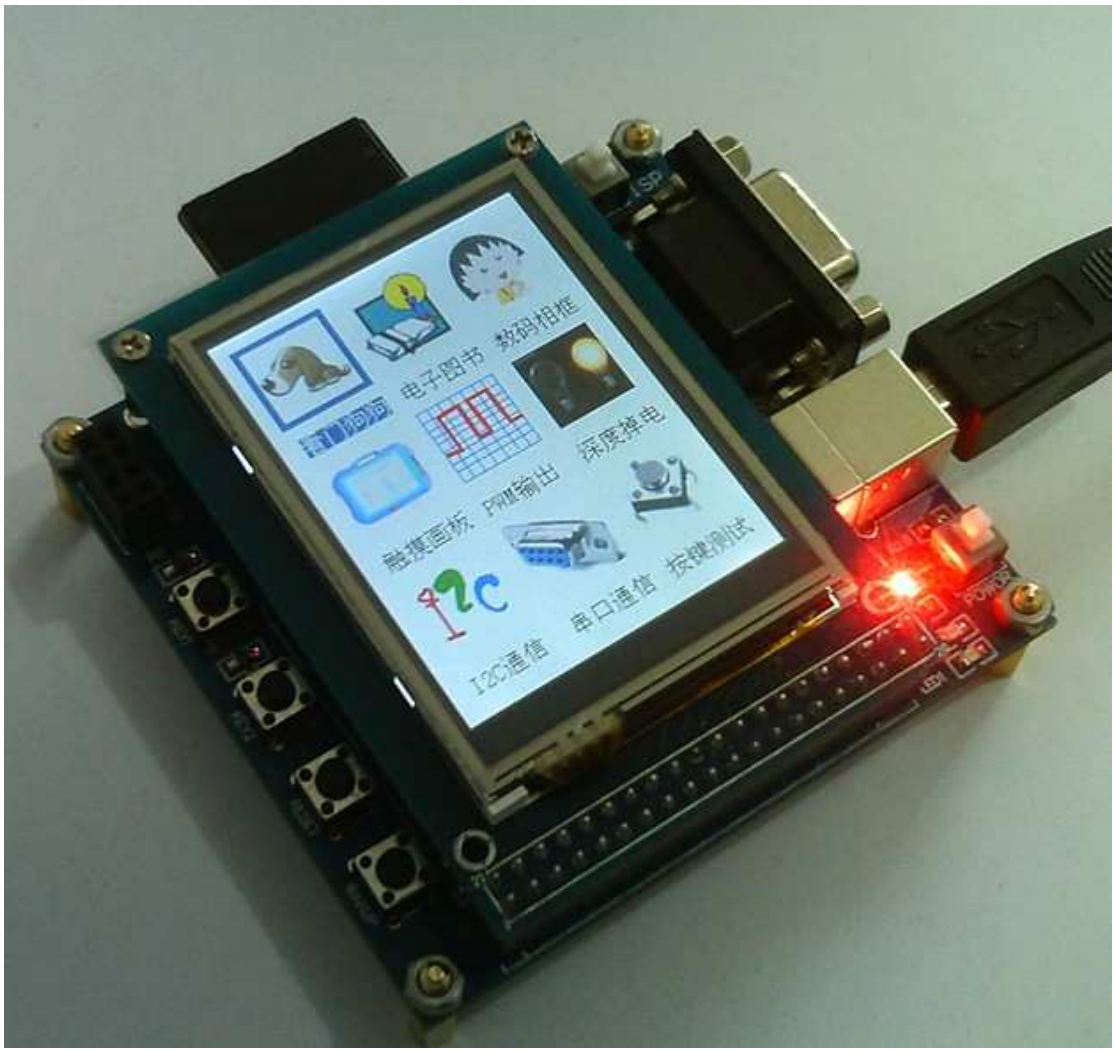
使用说明：

在这个例程中，由于内容比较多，所以把英文字母和符号一共 95 个放到了 W25X16 中，这样可以节省 1.5K 的内存空间。（方法参看第二十章）

本例程需要插入 SD 卡，在 SD 卡上放置如下几个文件夹，在光盘中的**“文件系统实验 SD 卡文件”**文件夹里面可以找到。

把以上四个文件放入 SD 卡中，插入开发板的 SD 卡插槽，下载例程 21 的 HEX 文件到 LPC1114，即可。

单击一下屏幕上的图标，会显示选中，再单击一下进入。如下图是单击“看门狗狗”图标以后的效果。



再次单击一下上一次单击的图标，即可进入对应功能。

单击与上次不同的图标，会选中这次单击的图标，同时取消上次的选中图标。

除了“电子图书”和“数码相框”没有明显的“退出”按钮，其它的都有。

关于“电子图书”功能的退出：显示的画面和效果与第十六章的一样，使用方法见第十六章。只是多了一个退出功能。这个退出功能做到了“瑞”图标上。在目录浏览模式下，点击“瑞”图标会退出“电子图书”，回到主界面。在内容浏览模式下，点击“瑞”图标无用，所以要想退出“电子图书”，需要回到文件名目录浏览模式才可以。

关于“数码相框”功能的退出：这里的数码相框和第十七章的不一样。在第十七章，系统自动循环浏览显示 SD 卡中“picture”文件夹下的图片。在这里，是手动切换图片，可以实现“上一张”“下一张”的人为操作。进入“数码相框”功能后，首先出现一个提示，背景为黑色，意在说明，点击屏幕左边，是“上一张”，点击屏幕右边，是“下一张”，点击屏幕下部，是“退出”。

最后，感谢您对 ARM CORTEX-M0 RATION LPC1114 开发板的支持。