# 实验一 ARM 开发基础

## 实验目的

* 了解“EMSBC2410实验平台”的基本硬件组成；
* 初步学会使用 μVision3 IDE for ARM 开发环境及ARM 软件模拟器；
* 通过实验掌握简单 ARM 汇编指令的使用方法。

## 实验内容

* 熟悉开发环境的使用并使用 ldr/str，mov 等指令访问寄存器或存储单元；
* 学习使用 add/sub/lsl/lsr/and/orr 等指令，完成基本算术/逻辑运算；
* 设计实现表达式y=a\*8+b/2-c，并验证之；
* 完成分支程序设计，要求判断参数，根据不同参数，调用不同的子程序。

## 实验原理

ARM9 处理器内核共有 37 个寄存器：

* 31 个通用寄存器，包括程序计数器(PC)。这些寄存器都是 32 位的；
* 6 个状态寄存器。这些寄存器也是 32 位的，但是只是使用了其中的 12 位。

这里简要介绍通用寄存器，关于状态寄存器的介绍，请参照下一节。

1．ARM 通用寄存器

通用寄存器（R0-R15）可分为三类：

* 不分组寄存器 R0~R7；
* 分组寄存器 R8~R14；
* 程序计数器 PC。

1. 不分组寄存器 R0~R7

不分组寄存器 R0~R7 在所有处理器模式下，它们每一个都访问一样的 32 位寄存器。它们是真正的通用寄存器，没有体系结构所隐含的特殊用途。

1. 分组寄存器 R8~R14

分组寄存器 R8～R14 对应的物理寄存器取决于当前的处理器模式。若要访问特定的物理寄存器而不依赖当前的处理器模式，则要使用规定的名字。

寄存器 R8~R12 各有两组物理寄存器：一组为 FIQ 模式，另一组为除了 FIQ 以外的所有模式。寄存器 R8~R12 没有任何指定的特殊用途，只是在作快速中断处理时使用。寄存器 R13，R14 各对应 6 个分组的物理寄存器，1 个用于用户模式和系统模式，其它 5 个分别用于 5 种异常模式。寄存器 R13 通常用做堆栈指针，称为

SP；寄存器 R14 用作子程序链接寄存器，也称为 LR。



1. 程序计数器 PC

寄存器 R15 用做程序计数器 (PC)。

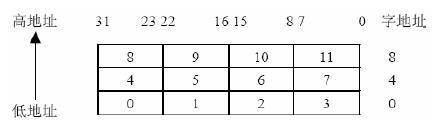
在本实验中，ARM 核工作在用户模式，R0~R15 可用。

2．存储器格式

ARM 体系结构将存储器看作是从零地址开始的字节的线性组合。字节零到字节三放置第一个字（WORD），字节四到字节七存储第二个字，以此类推。ARM 体系结构可以用两种方法存储字数据，分别称为大端格式和小端格式。

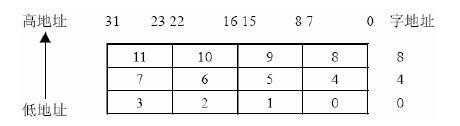
* 大端格式

在这种格式中，字数据的高位字节存储在低地址中，而字数据的低位字节则存放在高地址中，如图 4-1 所示。

 图 4-1 大端格式

* 小端格式

在这种格式中，字数据的高位字节存储在高地址中，而字数据的低位字节则存放在低地址中，如图 4-2 所示。

 图 4-2 小端格式

3．Realview 基础知识

μVision3 IDE 集成了 REALVIEW 汇编器 AARM、编译器 CARM、链接器 LARM，若采用 GNU 编译器则需要下载安装相应的工具包。本书所有例程代码均按照 REALVIEW 的语法和规则来书写。关于 AARM、CARM 和 LARM 的规范和具体使用，可参照 μVision3 IDE 所带的帮助文档，在此不再赘述。在 ARM 汇编中，这里简单介绍几个相关基本知识：

* **ENTRY**

设置程序默认入口点，一个程序可有多个 ENTRY，但一个源文件最多只有一个 ENTRY。

* **EQU**

EQU 伪操用于将数字常量、基于寄存器的值和程序中的标号定义为一个字符名称。语法格式：

symbol EQU expression

其中，expression 可以是一个寄存器的名字，也可是由程序标号、常量或者 32

位的地址常量组成的表达式。symbol 是 EQU 伪操作所定义的字符名称。示例：

COUNT EQU 0X1FFF

* **EXTERN/IMPORT**

IMPORT (EXTERN 功能完全相同)用于声明在其他模块中定义但需要在本文件中使用的符号。EXTRN 声明的变量必须是在其他模块中用 EXPORT 或

GLOBAL 声明过的。语法格式：

IMPORT class (symbol, symbol ...)

其中，class 为变量的类型，可以为 ARM、CODE16、CODE32、DATA、CONST、

THUMB； symbol 为所声明的变量名。

* **EXPORT/GLOBAL**

EXPORT（GLOBAL 功能完全相同）用于声明在本文件中定义但能在其他模块中使用的变量，相当于定义了一个全局变量。语法格式：

EXPORT symbol, symbol... 其中，symbol 为所声明的变量名。

* **AREA**

AREA 用于定义一个代码段或数据段，ARM 汇编程序设计采用分段式设计，一个 ARM 源程序至少有一个代码段，大的程序会有若干个代码段和数据段。语法格式：

AREA segment-name, class-name, attributes ,...

其中，segment-name 为所定义段的名称；class-name 为所定义段的类型名称，

可以为系统类型（CODE, CONST, DATA, ERAM）或用户定义类型；attributes 为段的属性。

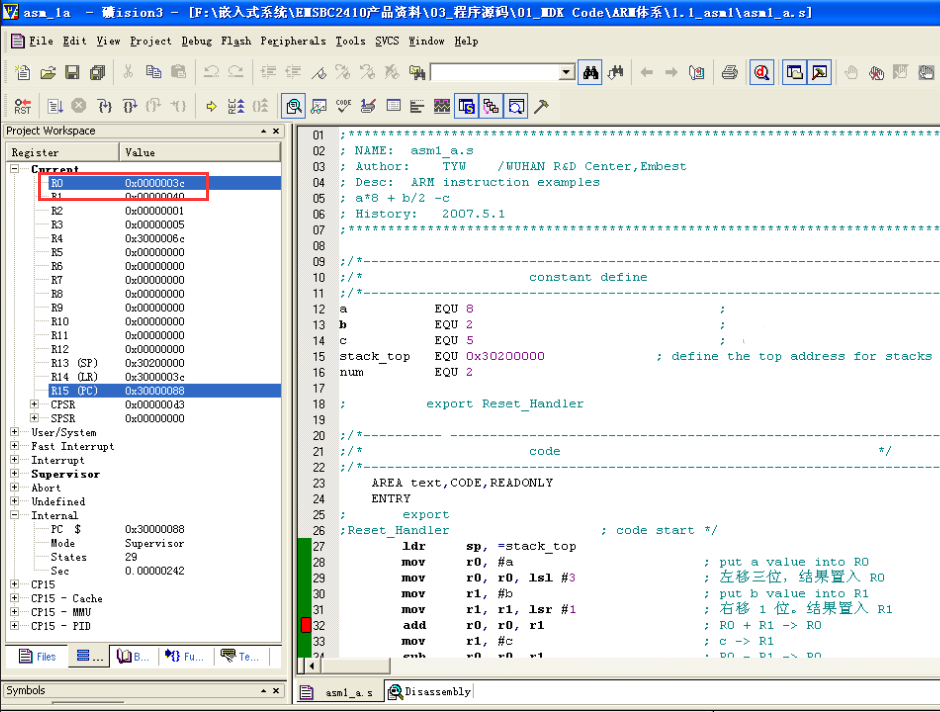
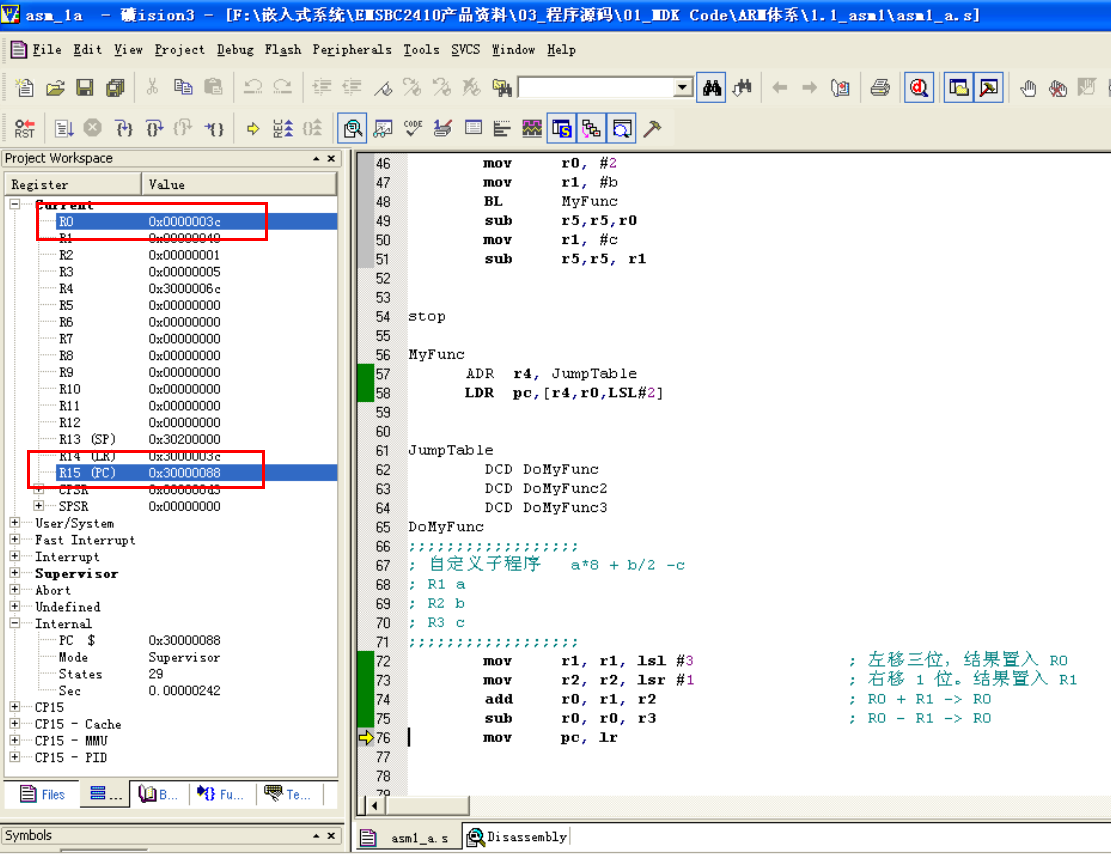
* **END**

END 用于标记汇编文件的结束行，即标号后的代码不作处理。

## 实现步骤

1. 验证示例源码
   * 拷贝整个实验例程源码目录到本地磁盘自己的工作目录下；
   * 使用μVision IDE for ARM 通过ULINK2仿真器连接实验板，打开实验例程目录ARM体系\1.1*asm1子目录下的asm*1a.Uv2 例程，编译链接工程；
   * 该工程配置为在模拟器中运行，可调试程序，观察结果；
   * 验证实验例程目录ARM体系\1.2\_asm2子目录下的asm2.Uv2 例程。
2. 设计实现自己的汇编程序
   * 拷贝示例实验源码工程；
   * 编辑汇编源程序
   * 设计实现表达式y=a\*8+b/2-c，并验证之；
   * 设计利用函数跳转表实现子程序调用，并验证之。

## 实验结果

* μVision IDE 可以正常编译、运行例程，模拟器可以进行程序的调试，并可以实时观察寄存器值和内存单元内容，并可进行实时修改。
* 表达式 y=a\*8+b/2-c 计算无误，取 a = 8 ，b = 2，c = 5 进行计算，结果为 0x3c （60），结果存储在 R0 中
* 
* 利用函数跳转表实现子程序调用，编写了子程序计算 a\*8 + b/2 - c ， R0 中得到了正确的计算结果。子程序调用完毕后，R15（PC）寄存器恢复，程序回到主函数中继续运行
* 

## 程序说明

1. 表达式的计算
   * 通过 mov 指令，将立即数移动到寄存器中，使用 lsr lsl ，指令，通过移位的方式计算 a\*8 、b/2
   * mov r0, #a ; put a value into R0   
      mov r0, r0, lsl #3 ; 左移三位，结果置入 R0   
      mov r1, #b ; put b value into R1   
      mov r1, r1, lsr #1 ; 右移 1 位。结果置入 R1
   * 通过 add sub 指令实现表达式的计算
   * add r0, r0, r1 ; R0 + R1 -> R0   
      mov r1, #c ; c -> R1   
      sub r0, r0, r1 ; R0 - R1 -> R0
2. 利用函数跳转表实现子程序调用

* 定义函数跳转表
* JumpTable   
   DCD DoMyFunc   
   DCD DoMyFunc2   
   DCD DoMyFunc3
* 通过 ldr 指令，根据跳转表内容，修改 pc 寄存器的值，实现子程序调用
* MyFunc   
   ADR r4, JumpTable   
   LDR pc,[r4,r0,LSL#2]
* 子程序结束，通过修改 pc 寄存器的值，返回主程序
* mov pc, lr

完整源代码如下：

;/\*-----------------------------------------------------------------------------------------\*/   
;/\* constant define \*/   
;/\*-----------------------------------------------------------------------------------------\*/   
a EQU 8 ; a=8   
b EQU 2 ; b=2   
c EQU 5 ; c=5   
stack\_top EQU 0x30200000 ; define the top address for stacks   
num EQU 2   
   
; export Reset\_Handler   
   
;/\*---------- ------------------------------------------------------------------------------\*/   
;/\* code \*/   
;/\*-----------------------------------------------------------------------------------------\*/   
 AREA text,CODE,READONLY   
 ENTRY   
; export   
;Reset\_Handler ; code start \*/   
 ldr sp, =stack\_top   
 mov r0, #a ; put a value into R0   
 mov r0, r0, lsl #3 ; 左移三位，结果置入 R0   
 mov r1, #b ; put b value into R1   
 mov r1, r1, lsr #1 ; 右移 1 位。结果置入 R1   
 add r0, r0, r1 ; R0 + R1 -> R0   
 mov r1, #c ; c -> R1   
 sub r0, r0, r1 ; R0 - R1 -> R0   
 NOP   
 NOP   
 mov r0, #0   
 mov r1, #a   
 mov r2, #b   
 mov r3, #c   
 BL MyFunc   
 mov r0, #1   
 mov r1, #a   
 BL MyFunc   
 mov r5, r0   
 mov r0, #2   
 mov r1, #b   
 BL MyFunc   
 sub r5,r5,r0   
 mov r1, #c   
 sub r5,r5, r1   
   
   
stop   
   
MyFunc   
 ADR r4, JumpTable   
 LDR pc,[r4,r0,LSL#2]   
   
   
JumpTable   
 DCD DoMyFunc   
 DCD DoMyFunc2   
 DCD DoMyFunc3   
DoMyFunc   
;;;;;;;;;;;;;;;;;;   
; 自定义子程序 a\*8 + b/2 -c   
; R1 a   
; R2 b   
; R3 c   
;;;;;;;;;;;;;;;;;;   
 mov r1, r1, lsl #3 ; 左移三位，结果置入 R0   
 mov r2, r2, lsr #1 ; 右移 1 位。结果置入 R1   
 add r0, r1, r2 ; R0 + R1 -> R0   
 sub r0, r0, r3 ; R0 - R1 -> R0   
 mov pc, lr   
   
   
   
   
DoMyFunc2   
;;;;;;;;;;;;;;;;;;   
; 自定义子程序 a\*8   
; R1 a   
;;;;;;;;;;;;;;;;;;   
 mov r1, r1, lsl #3 ; 左移三位，结果置入 R0   
 mov pc, lr   
DoMyFunc3   
;;;;;;;;;;;;;;;;;;   
; 自定义子程序 a/2   
; R1 a   
;;;;;;;;;;;;;;;;;;   
 mov r1, r1, lsr #1 ; 左移三位，结果置入 R0   
 mov pc, lr   
   
   
 end

## 总结

本次实验是嵌入式系统的第一次实验，我也是第一次编写 ARM 汇编程序，与 X86 汇编不同，通过这次实验，我感受到了 ARM 汇编极大的灵活性，特别是寻址方式和寄存器操作。ARM 提供的大量寄存器也给编写汇编程序提供了很大的便利。

# 实验二 S3C2410 基本接口实验

## 实验目的

* 掌握S3C2410X 芯片的I/O 控制寄存器的配置；
* 通过实验掌握ARM 芯片使用I/O 口控制LED 显示；
* 了解ARM 芯片中复用I/O 口的使用方法；
* 通过实验掌握键盘控制与设计方法。
* 熟练编写 ARM 核处理器S3C2410X 中断处理程序。

## 实验内容

* 编写程序，控制实验平台的发光二极管LED1、LED2、LED3、LED4，使它们有规律的点亮和熄灭；
* 使用实验板上 5x4 用户键盘，编写程序接收键盘中断；
* 使用键盘控制发光二极管，按照不同模式点亮。

## 实验原理

**1.** 常规键盘电路设计原理

用户设计行列键盘接口，一般常采用三种方法读取键值。一种是中断式，另两种是扫描法和反转法。

中断式

在键盘按下时产生一个外部中断通知 CPU，并由中断处理程序通过不同的地址读取数据线上的状态，判断哪个按键被按下。本实验采用中断方式实现用户键盘接口。

扫描法

对键盘上的某一行发送低电平，其他为高电平，然后读取列值，若列值中有一位是低，表明该行与低电平对应列的键被按下。否则扫描下一行。

反转法

先将所有行扫描线输出低电平，读列值，若列值有一位是低，表明有键按下；

接着所有列扫描线输出低电平，再读行值。根据读到的值组合就可以查表得到键码。

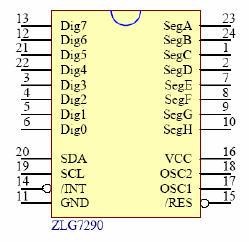
**2.** 使用**ZLG7290** 的键盘电路设计原理

**1) ZLG7290** 的特点

* IIC 串行接口，提供键盘中断信号，方便与处理器接口；
* 可驱动 8 位共阴数码管或 64 只独立 LED 和 64 个按键；
* 可控扫描位数，可控任一数码管闪烁；
* 提供数据译码和循环，移位，段寻址等控制；
* 8 个功能键，可检测任一键的连击次数；
* 无需外接元件即直接驱动 LED，可扩展驱动电流和驱动电压；
* 提供工业级器件，多种封装形式 PDIP24，SO24。

1. **ZLG7290** 的引脚说明

采用 24 引脚封装，引脚图如下所示。其引脚功能分述如下：

图 5-2 ZLG7290 引脚图

1. **ZLG7290** 的寄存器说明

系统寄存器（SystemReg）： 地址 00H，复位值 11110000B。系统寄存器保存

ZLG7290 的

系统状态，并可对系统运行状态进行配置。

KeyAvi（SystemReg.0）：置 1 时表示有效的按键动作（普通键的单击，连击，和功能键状态变化），/INT 引脚信号有效变为低电平； 清 0 表示无按键动作/INT 引脚信号无效变为高阻态有效的按键动作消失后或读 Key 后 KeyAvi 位自动清 0。键值寄存器（Key）： 地址 01H，复位值 00H。Key 表示被压按键的键值。当Key=0 时，表示没有键被压按。

连击次数计数器（RepeatCnt）： 地址 02H，复位值 00H 。RepeatCnt=0 时，表示单击键。RepeatCnt 大于 0 时，表示键的连击次数。用于区别出单击键或连击键，判断连击次数可以检测被按时间。

功能键寄存器（FunctionKey）： 地址 03H，复位值 0FFH。FunctionKey 对应位的值=0 表示对应功能键被压按（FunctionKey.7 ~FunctionKey.0 对应 S64 S57）。

命令缓冲区（CmdBuf0~CmdBuf1）： 地址 07H~08H，复位值 00H~00H。用于传输指令。

闪烁控制寄存器（FlashOnOff）： 地址 0CH，复位值 0111B/0111B。高 4 位表示闪烁时亮的时间，低 4 位表示闪烁时灭的时间，改变其值同时也改变了闪烁频率，也能改变亮和灭的占空比。FlashOnOff 的 1 个单位相当于 150~250ms（亮和灭的时间范围为：1~16，0000B 相当 1 个时间单位），所有象素的闪烁频率和占空比相同。

扫描位数寄存器（ScanNum）： 地址 0DH，复位值 7。用于控制最大的扫描显示位数（有效范围为 0~7，对应的显示位数为：1~8），减少扫描位数可提高每位显示扫描时间的占空比，以提高 LED 亮度。不扫描显示的显示缓存寄存器则保持不变。如 ScanNum=3 时，只显示 DpRam0~DpRam3 的内容。

显示缓存寄存器（DpRam0~DpRam7）： 地址 10H~17H，复位值 00H~00H。缓存中位置 1 表示该像素亮，DpRam7~DpRam0 的显示内容对应 Dig7~Dig0 引脚。

**4) ZLG7290** 的通信接口

ZLG7290 的 IIC 接口传输速率可达 32kbit/s，容易与处理器接口通信，并提供键盘中断信号，提高主处理器时间效率。ZLG7290 的从地址 slave address 为 70H (01110000B)。我们从它的键值寄存器（01H）中读取按键值（ucChar 用于保存读到的键值）：

**iic\_read(0x70, 0x1, &ucChar);**

有效的按键动作都会令系统寄存器（SystemReg）的 KeyAvi 位置 1，/INT 引脚信号有效（变为低电平）。用户的键盘处理程序可由/INT 引脚低电平中断触发，以提高程序效率；也可以不采样/INT 引脚信号节省系统的 I/O 数，而轮询系统寄存器的 KeyAvi 位。要注意读键值寄存器会令 KeyAvi 位清 0，并会令/INT 引脚信号无效。

ZLG7290 内可通过 IIC 总线访问的寄存器地址范围为：00H~17H，任一寄存器都可按字节直接读写，也可以通过命令接口间接读写或按位读写，请参考 ZLG7290 芯片手册。ZLG7290 支持自动增址功能（访问一寄存器后寄存器子地址自动加一）和地址翻转功能（访问最后一寄存器后寄存器子地址翻转为 00H ）。 ZLG7290 的控制和状态查询全部都是通过读/写寄存器实现的，用户只需象读写 24C02 内的单元一样即可实现对 ZLG7290 的控制，关于 IIC 总线访问的细节请参考 IIC 总线规范。

**3.** 键盘硬件电路设计

1. 键盘连接电路

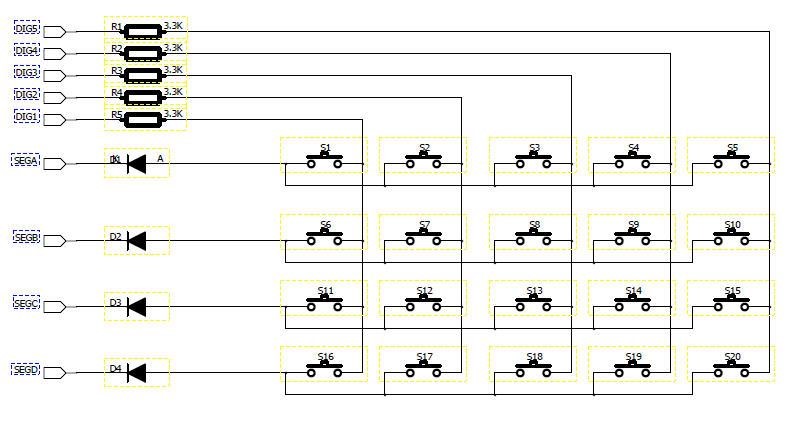


图 5-3 键盘连接电路

1. 键盘控制电路

键盘控制电路使用芯片 ZLG7290 控制，如图 5-4。对应下图中的 14 引脚

KEY\_INT 捕捉由键盘按下产生的中断触发信号。

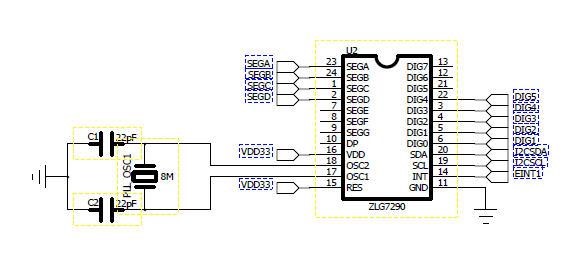


图 5-4 键盘控制电路

1. 工作过程

键盘动作由芯片 ZLG7290 检测，当键盘按下时，芯片检测到后在 INT 引脚产生中断触发电平通知处理器，EINT1 为处理器 S3C2410 的 I/O 引脚（可软件配置成中断功能）；处理器通过 IIC 总线读取芯片 ZLG7290 键值寄存器（01H）中保存的键值（具体的读取方法参看实验原码）。

## 实现步骤

1. 验证示例源码
   * 拷贝整个实验例程源码目录到本地磁盘自己的工作目录下；
   * 使用μVision IDE for ARM 通过ULINK2仿真器连接实验板，打开实验例程目录02*led*test 子目录下的led\_test. Uv2 例程，编译链接工程；
   * 调试程序，观察运行结果；
   * 打开实验例程目录12\_KeyBoardTest子目录下的KeyBoardTest.Uv2 例程，编译链接工程
   * 调试程序，观察运行结果。
2. 设计实现自己的I/O控制程序
   * 拷贝示例实验源码工程；
   * 设计程序，实现使用键盘控制发光二极管按照不同模式点亮。

## 实验结果

1. 通过仿真器连接实验版，并烧写示例程序后，示例程序可以正常运行。连接串口线缆，打开超级终端，按动实验板上按键，终端给出对应输出。示例程序 02\_led\_test 运行后，实验板上发光二极管闪烁。
2. 编写 I/O控制程序 并编译运行后，按动按键，发光二极管按照设定模式点亮：
   * 按 0 键：二极管全部点亮
   * 按 1 键：二极管 1 熄灭，其余全部点亮
   * 按 2 键：二极管 1、2 熄灭，其余全部点亮
   * 按 3 键：二极管 1、2、3 熄灭，4 点亮
   * 按 4 键：二极管全部熄灭
   * 按 5 键：二极管 1、2、3、4 依次全部点亮，全部点亮后，按照 4、3、2、1 的顺序依次熄灭。
   * 按 6 键：全部二极管快速闪烁 3 次

## 程序说明

keyboard\_test() 函数通过查询 key*set 按键表，获取当前按键键值，并在 keyboard*test 函数中判断返回键值，调用不同的二极管控制函数。

void keyboard\_test(void)   
{   
 UINT8T ucChar;   
   
 uart\_printf("\n Keyboard Test Example\n");   
 keyboard\_init();   
 while(1)   
 {   
 while(g\_nKeyPress)   
 {   
 g\_nKeyPress = 0;   
 iic\_read\_keybd(0x70, 0x1, &ucChar); // get data from Key(register of ZLG7290)   
 if(ucChar != 0)   
 {   
 ucChar = key\_set(ucChar); // key map for EduKitII   
 if(ucChar < 10) ucChar += 0x30;   
 else if(ucChar < 16) ucChar += 0x37;   
 if(ucChar < 255)   
 uart\_printf("press key %c\n", ucChar);   
 switch(ucChar){   
 case '1':   
 keyfunc1();   
 break;   
 case '2':   
 keyfunc2();   
 break;   
 case '3':   
 keyfunc3();   
 break;   
 case '4':   
 keyfunc4();   
 break;   
 case '5':   
 keyfunc5();   
 break;   
 case '6':   
 keyfunc6();   
 break;   
 }   
 }   
 }   
 }   
// uart\_printf(" end.\n");   
}

rGPBDAT 是个16位整数, 其意义如下

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ... |
|  |  |  |  |  | led4 | led3 | led2 | led1 | ... |

这些位为0时, 灯亮, 为1时, 灯灭

通过位运算置他们为0, 1即可达到操作led的目的

## 总结

通过本次实验，我掌握了烧写编译好的程序到实验板中的方法，并成功通过程序实现了对硬件 （LED）灯的控制。并通过读取按键中断的返回结果实现了按键交互功能，并掌握了中断处理，按键控制的基本原理和方法。

# 实验三 人机接口实验

## 实验目的

* 掌握液晶屏的使用及其电路设计方法；
* 掌握S3C2410X 处理器的LCD 控制器的使用；
* 通过实验掌握液晶显示文本及图形的方法与程序设计；
* 通过实验掌握触摸屏（TSP）的设计与控制方法。

## 实验内容

* 掌握液晶屏作为人机接口界面的设计方法，并编写程序实现；
* 编程实现触摸屏坐标转换为液晶对应坐标；
* 编程实现由液晶屏和触摸屏构成的可以互动的人机界面，至少实现3屏。

## 实验原理

**1.** 液晶显示屏（**LCD**）

液晶屏（LCD：Liquid Crystal Display）主要用于显示文本及图形信息。液晶显示屏具有轻薄、体积小、低耗电量、无辐射危险、平面直角显示以及影像稳定不闪烁等特点，因此在许多电子应用系统中，常使用液晶屏作为人机界面。

 主要类型及性能参数液晶显示屏按显示原理分为 STN 和 TFT 两种：

**STN**（**Super Twisted Nematic**，超扭曲向列）液晶屏

STN 液晶显示器与液晶材料、光线的干涉现象有关，因此显示的色调以淡绿色与橘色为主。STN 液晶显示器中，使用 X、Y 轴交叉的单纯电极驱动方式，即 X、

Y 轴由垂直与水平方向的驱动电极构成，水平方向驱动电压控制显示部分为亮或暗，垂直方向的电极则负责驱动液晶分子的显示。STN 液晶显示屏加上彩色滤光片，并将单色显示矩阵中的每一像素分成三个子像素，分别通过彩色滤光片显示红、绿、蓝三原色，也可以显示出色彩。单色液晶屏及灰度液晶屏都是 STN 液晶屏。

TFT（Thin Film Transistor，薄膜晶体管）彩色液晶屏

随着液晶显示技术的不断发展和进步， TFT 液晶显示屏被广泛用于制作成电脑中的液晶显示设备。 TFT 液晶显示屏既可在笔记本电脑上应用（现在大多数笔记本电脑都使用 TFT 显示屏），也常用于主流台式显示器。 分 65536 色及 26 万色，1600 万色三种，其显示效果非常出色。TFT 的显示采用“背透式”照射方式—— 假想的光源路径不是像 TN 液晶那样从上至下，而是从下向上。这样的作法是在液晶的背部设置特殊光管，光源照射时通过下偏光板向上透出。由于上下夹层的电极改成 FET 电极和共通电极，在 FET 电极导通时，液晶分子的表现也会发生改变，可以通过遮光和透光来达到显示的目的，响应时间大大提高到 80ms 左右。使用液晶显示屏时，主要考虑的参数有外形尺寸、象素、点距、色彩等。以下是 S3C2410 实验板所选用的液晶屏（LQ080V3DG01 TFT）主要参数：表 6-1-1 LQ080V3DG01 TFT 液晶屏主要技术参数

型号 LQ080V3DG01 外形尺寸 183×141×14 mm 重量 390g 像素 320 × 240 点 距 0.2535 X0.2535 mm 色彩 262144 电压 5V（25℃） 对比度 250 附加带驱动逻辑液晶屏外形如图 6-1 所示：



图 **6-1** LQ080V3DG01 TFT 液晶屏外形

 驱动与显示

液晶屏的显示要求设计专门的驱动与显示控制电路。驱动电路包括提供液晶屏的驱动电源和液晶分子偏置电压，以及液晶显示屏的驱动逻辑；显示控制部分可由专门的硬件电路组成，也可以采用集成电路（IC）模块，比如 EPSON 的视频驱动器等；还可以使用处理器外围 LCD 控制模块。实验板的驱动与显示系统包括

S3C2410X 片内外设 LCD 控制器、液晶显示屏的驱动逻辑以及外围驱动电路。

**2. S3C2410X LCD** 控制器

 **LCD** 控制器特点

S3C2410X 处理器集成了 LCD 控制器，主要功能是 S3C2410X LCD 控制器用于传输显示数据和产生控制信号。它并支持屏幕水平和垂直滚动显示。数据的传送采用 DMA（直接内存访问）方式，以达到最小的延迟。它可以支持多种液晶屏：

**STN LCD**

* 支持 3 种类型的扫描方式：4 位单扫描，4 位双扫描和 8 位单扫描
* 支持单色，4 级灰度和 16 级灰度显示
* 支持 256 色和 4096 色彩色 STN LCD
* 支持多种屏幕大小

典型的实际屏幕大小是：640×480，320×240，160×160 和其它最大虚拟屏幕占内存大小为 4M 字节 256 色模式下最大虚拟屏幕大小：4096×1024, 2048×2048,

1024×4096 和其它

**TFT LCD**

* 支持 1，2，4 或 8bpp 彩色调色显示
* 支持 16bpp 和 24bpp 非调色真彩显示
* 在 24bpp 模式下，最多支持 16M 种颜色
* 支持多种屏幕大小
* 典型的实际屏幕大小是： 640×480, 320×240, 160×160 和其它
* 最大虚拟屏幕占内存大小为 4M 字节
* 64K 色模式下最大虚拟屏幕大小：2048×1024 和其它

 **LCD** 控制器内部结构

LCD 控制器主要提供液晶屏显示数据的传送、时钟和各种信号的产生与控制功能。S3C2410X 处理器的 LCD 控制器主要部分框图如图 6-2 所示：

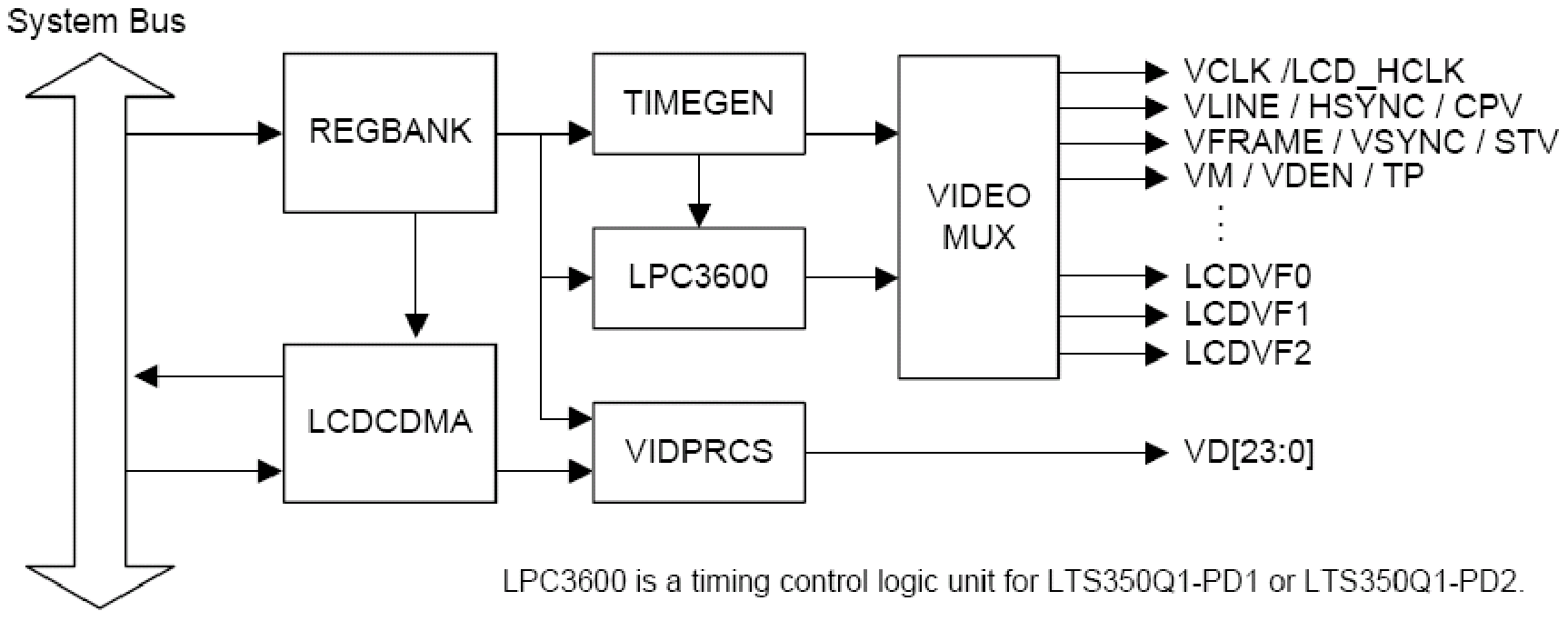


图 **6-2** LCD 控制器框图

S3C2410X LCD 控制器用于传输显示数据和产生控制信号，例如 VFRAME， VLINE，VCLK，VM 等等。除了控制信号之外，S3C2410X 还提供数据端口供显示数据传输，也就是 VD[23:0]。如上图所示。LCD 控制器包含 REGBANK， LCDCDMA，VIDPRCS，TIMEGEN 和 LPC3600 等控制模块。REGBANK 中有 17 个可编程的寄存器组和 256X16 调色板内存用于配置 LCD 控制器。LCDCDMA 是一个专用的 DMA，它负责自动的将帧缓冲区中的显示数据发往 LCD 驱动器。通过特定的 DMA，显示数据可以不需要 CPU 的干涉，自动的发送到屏幕上。VIDPRCS 将 LCDCDMA 发送过来的数据变换为合适的格式（例如 4/8 位单扫描或 4 位双扫描显示模式）之后通过 VD[23:0]发送到 LCD 驱动器。TMIEGEN 包含可编程逻辑用于支持不同 LCD 驱动器对时序以及速率的需求。VFRAME，VLINE，VCLK， VM 等控制信号由 TIMEGEN 产生。在 LCD 控制起的 33 个输出接口中有 24 个用户数据输出，9 个用于控制。

地址从 0x14A0002C 到 0x14A00048 禁止使用，因为这个区域用作测试用保留地址。S3C2410X 能够支持 STN LCD 和 TFT LCD，这两种 LCD 屏在显示的时候有很大的差别，而且所涉及到的寄存器也会不同。S3C2410 实验平台采用的是

TFT LCD，下面对 TFT LCD 的显示过程进行详细的介绍。

**TFT LCD** 显示

**1) LCD** 控制器时间相关参数设定

TIMEGEN 产生 LCD 驱动器所需要的控制信号，例如 VSYNC，HSYNC， VCLK ， VDEN 和 LEND 。这些控制信号又和 REGBANK 中的寄存器 LCDCON1/2/3/4/5 的设置密切相关。可以对 REGBANK 中的这些寄存器进行设置以产生适合于不同种类 LCD 驱动器的控制信号。VSYNC 是帧同步信号，VSYNC 每发出 1 个脉冲，都意味着新的 1 屏视频资料开始发送，而 HSYNC 为行同步信号，每个 HSYNC 脉冲都表明新的 1 行视频资料开始发送。而 VSYNC 和 HSYNC 脉冲的产生则依赖于 LCDCON2／3 寄存器的 HOZVAL 域和 LINEVAL 域的配置。

HOZVAL 和 LNEVAL 的值由 LCD 屏的尺寸决定：

HOZVAL=水平显示尺寸-1

LINEVAL=垂直显示尺寸-1

VCLK 信号的频率取决于 LCDCON1 寄存器中的 CLKVAL 域。VCLK 和

CLKVAL 的关系如下（其中 CLKVAL 的最小值是 0）：VCLK(Hz)=HCLK／

[(CLKVAL+1)x2]一般情况下，帧频率就是 VSYNC 信号的频率，它与 LCDCON1 和

LCDCON2／3／4 寄存器的 VSYNC、VB2PD、VFPD、LINEVAL、HSYNC、HBPD、 HFPD、HOZVAL 和 CLKVAL 都有关系。大多数 LCD 驱动器都需要与显示器相匹配的帧频率，帧频率计算公式如下：

Frame Rate = 1/ [ { (VSPW+1) + (VBPD+1) + (LIINEVAL + 1) + (VFPD+1) }×

{(HSPW+1) +(HBPD +1)+ (HFPD+1) + (HOZVAL + 1) } × { 2×( CLKVAL+1 ) /

( HCLK ) } ]

针对 16 位 TFT 屏 BSWP，HWSWP 这两位用来控制字节交换和半字交换，主要用来大小头的问题，如果输出到屏上的汉字左右互换了，或者输出到屏上的图花屏了，可以更改这个选项。图 6-3 是 LCD 屏幕上点的象素在内存中表示的示意图。

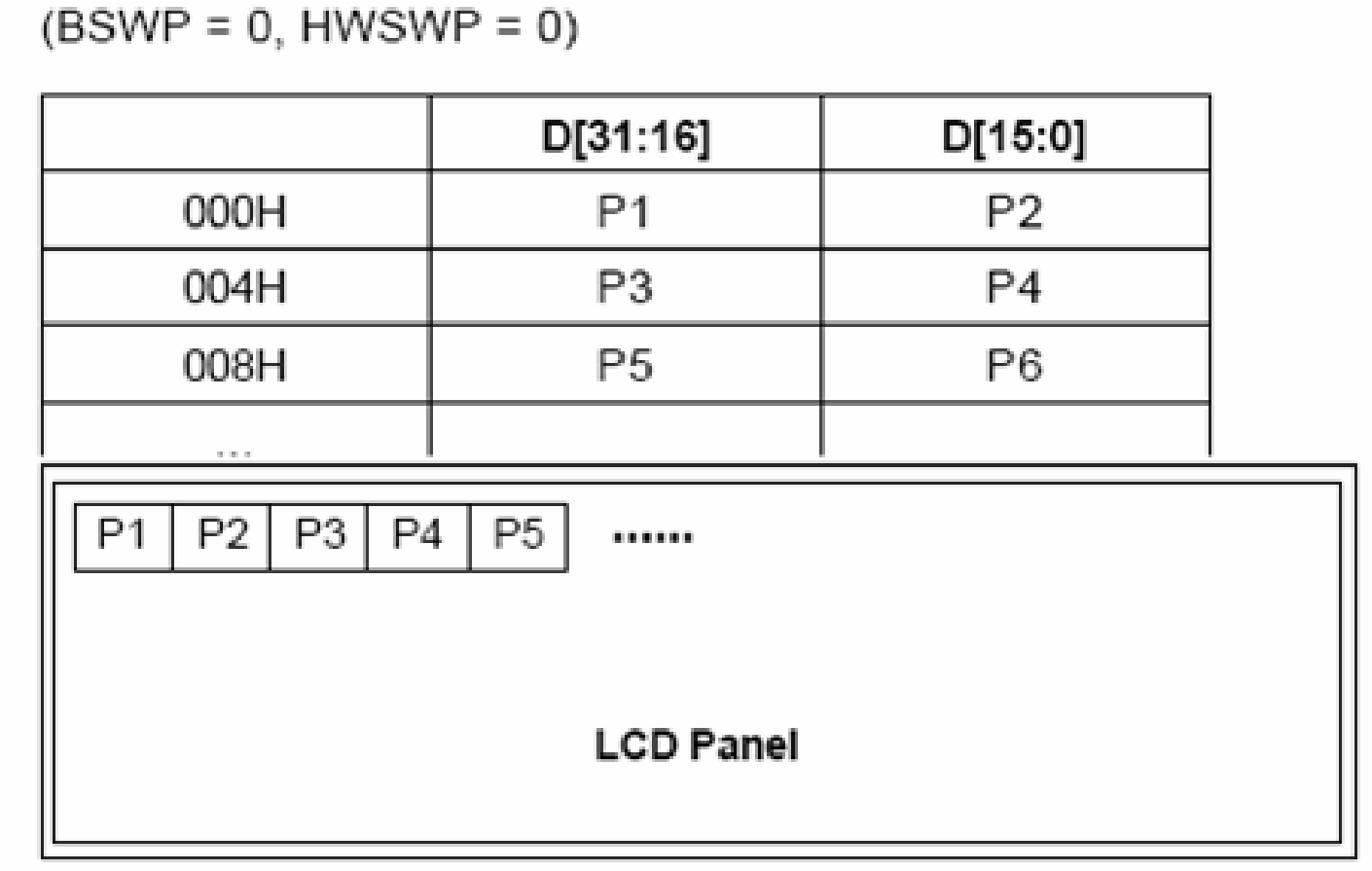


图 6-3 象素在内存中表示的示意图

图 6-4 是说明了 16 位 TFT 如何表示 RGB。

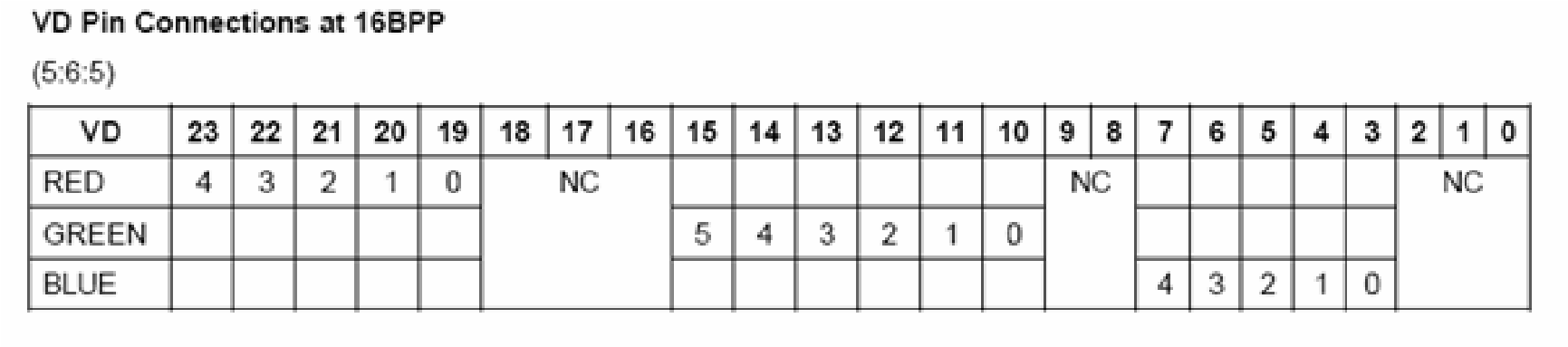


图 **6-4** 16 位 TFT 表示 RGB 示意图

写一个 16 位数据的颜色数据（为了分析的方便，把它写成二进制）

RGB = 10101101 10111001

根据上面的结构可以得出分析一下 RGB 各是多少

1. blue: {offset: 0, length: 5} 偏移量为 0，长度为 5，从 RGB 中提取出来便是

“11001”；

1. green:{offset: 5, length: 6} 偏移量为 5，长度为 6，从 RGB 中提取出来便是

“101101”；

1. red: {offset: 11, length: 5 } 偏移量为 11，长度为 5，从 RGB 中提取出来便

是“10101”。

图 6-5 是表示了对应 16 位 TFT，一个象素点的 RGB 示意图。屏幕上 1 个象素用 16 位表示。

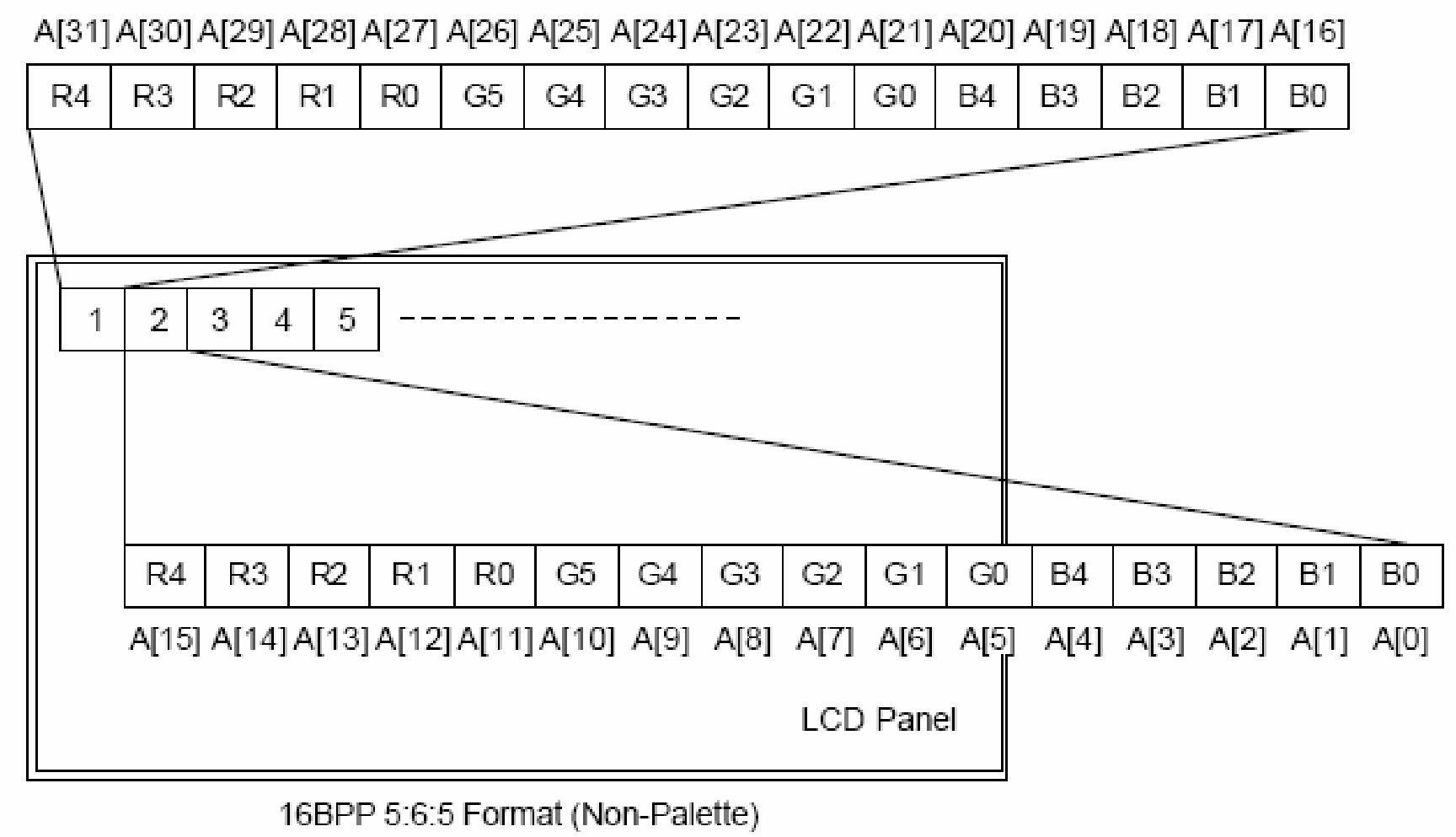


图 **6-5** 象素点的 RGB 示意图

**2) TFT LCD** 控制器信号时序

TFT 屏的典型时序。其中 VSYNC 是帧同步信号，VSYNC 每发出 1 个脉冲，都意味着新的 1 屏视频资料开始发送。而 HSYNC 为行同步信号，每个 HSYNC 脉冲都表明新的 1 行视频资料开始发送。而 VDEN 则用来标明视频资料的有效， VCLK 是用来锁存视频资料的像数时钟。并且在帧同步以及行同步的头尾都必须留有回扫时间，例如对于 VSYNC 来说前回扫时间就是（VSPW+1）＋（VBPD+1），后回扫时间就是（VFPD+1）；HSYNC 亦类同。这样的时序要求是当初 CRT 显示器由于电子枪偏转需要时间，但后来成了实际上的工业标准，乃至于后来出现的

TFT 屏为了在时序上于 CRT 兼容，也采用了这样的控制时序。



图 **6-6** LCD 时序图

## 实现步骤

1. 验证示例源码
   * 拷贝整个实验例程源码目录到本地磁盘自己的工作目录下；
   * 使用μVision IDE for ARM 通过ULINK2仿真器连接实验板，打开实验例程目录11*LCD*Test 子目录下的LCD\_Test.Uv2例程，编译链接工程；
   * 调试程序，观察运行结果；
   * 打开实验例程目录07*TSP*Test子目录下的TSP\_Test.Uv2例程，编译链接工程；
   * 调试程序，观察运行结果。
2. 设计实现自己的人机互动界面程序
   * 拷贝示例实验源码工程；
   * 设计程序，实现由液晶屏和触摸屏构成的可以互动的人机界面，至少实现3屏。

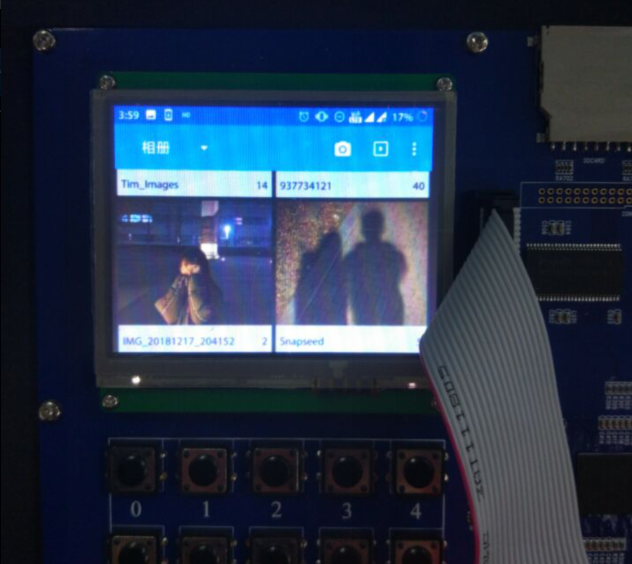
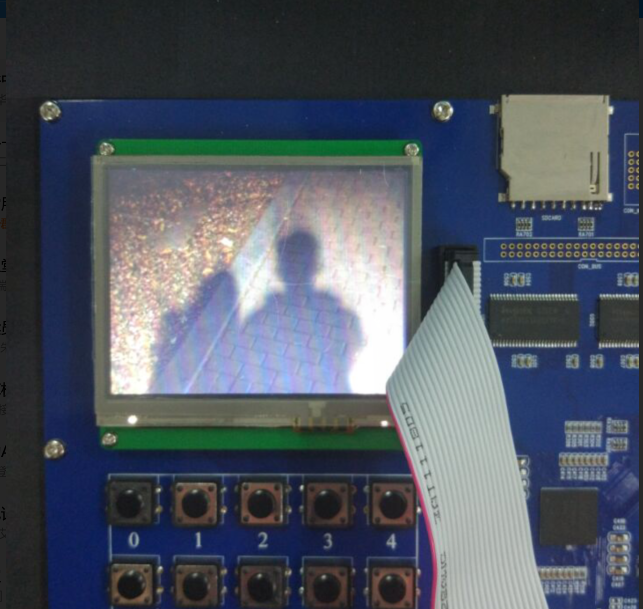
## 实验结果

实验实现了模拟图库浏览照片的交互操作。

在实验板上点击对应位置可以切换到对应界面，点击屏幕右下角实现返回到上一界面。

* 实验板初始化界面：



* 触摸后进入功能选择界面
* 
* 触摸选择 图库 功能，进入相册预览界面
* 
* 点击图片，放大显示对应照片
* 

## 程序说明

main 函数中调用 tsp\_test()；进入交互测试程序。

在 tsp\_test.c 中定义全局变量 int is\_touched，在触摸中断返回前，将 is\_touched 变量值置为1，tsp\_test() 的 while 循环中，对 is\_touched 变量进行检测，检测到其为 1 时，触发界面交互逻辑。

while(1) // only for board test to exit   
 {   
 // or press any key to exit   
 is\_back = 0;   
 if(is\_touched){   
 if (g\_nPosX > 750 && g\_nPosY > 850){   
 status -= 1;   
 is\_back = 1;   
 }   
 if (!is\_back){   
 switch(status){   
 case 0:   
 status = 1;   
 break;   
 case 1:   
 if (g\_nPosX<900 && g\_nPosX > 660 && g\_nPosY < 300 && g\_nPosY > 100){   
 status = 2;   
 }   
 break;   
 case 2:   
 if (g\_nPosY > 600){   
 status = 3;   
 }   
 break;   
   
 }   
 }   
 switch(status){   
 case 1:   
 BitmapView\_320240(gImage\_2);   
 break;   
 case 2:   
 BitmapView\_320240(gImage\_3);   
 break;   
 case 3:   
 BitmapView\_320240(gImage\_4);   
 break;   
   
 }   
 is\_touched = 0;   
   
 }   
 }

交互函数中，使用 状态机 的思想，定义并维护变量 status , 对当前交互界面的状态以及将要切换的状态进行判断，状态切换后，渲染不同的界面，实现交互的功能。

界面图像使用 lmage2Lcd 软件进行制作，选择输出 16 位真彩色，并不包含图像头数据



在 tsp\_test.c 中，对界面图像进行 extern 声明，便可在交互函数中进行使用

完整实验源码如下：

#include "2410lib.h"   
#include "tsp\_test.h"   
//#include "1.c"   
//#include "2.c"   
extern const UINT8T gImage\_1[][76800];   
extern const UINT8T gImage\_2[][76800];   
extern const UINT8T gImage\_3[][76800];   
extern const UINT8T gImage\_4[][76800];   
   
/\*------------------------------------------------------------------------------------------\*/   
/\* constants define \*/   
/\*------------------------------------------------------------------------------------------\*/   
#define ADCPRS 39   
int nSampleNo=5;   
UINT32T g\_nPosX,g\_nPosY;   
int g\_nKeyPress;   
int is\_touched = 0;   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* name: tsp\_int   
\* func: Touch screen interrupt handler (ADC)   
\* para: none   
\* ret: none   
\* modify:   
\* comment:   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void \_\_irq tsp\_int(void)   
{   
 int i;   
 UINT32T szPos[40];   
 rINTSUBMSK |= (BIT\_SUB\_ADC | BIT\_SUB\_TC);// Mask sub interrupt (ADC and TC)   
   
 // TC(Touch screen Control) Interrupt   
 if( rADCTSC & 0x100)   
 {   
 rADCTSC&=0xff; // Set stylus down interrupt   
 }   
 else   
 {   
 uart\_printf(" Stylus Down!!\n");   
 szPos[30] = g\_nPosX;   
 szPos[34] = g\_nPosY;   
   
 // <X-Position Read>   
 //Hi-Z,AIN5,GND,Ext vlt,Pullup Dis,Normal,X-position   
 rADCTSC = (0<<8)|(0<<7)|(1<<6)|(1<<5)|(0<<4)|(0<<3)|(0<<2)|(1);   
 //adc input ain5   
 rADCCON = (1<<14)|(39<<6)|(5<<3)|(0<<2)|(1<<1)|(0);   
 rADCDAT0;   
 delay(10);   
 for(i = 0,g\_nPosX=0; i<nSampleNo; i++)   
 {   
 while(!(0x8000 & rADCCON)); // Check ECFLG   
 szPos[i] = (0x3ff & rADCDAT0);   
 g\_nPosX += szPos[i];   
 }   
 g\_nPosX = g\_nPosX/nSampleNo;   
// g\_nPosX = (g\_nPosX-45)\*640/685;   
 uart\_printf(" X-Posion[AIN5] is %04d\n", g\_nPosX);   
   
 // <Y-Position Read>   
 //GND,Ext vlt,Hi-Z,AIN7,Pullup Dis,Normal,Y-position   
 rADCTSC = (0<<8)|(1<<7)|(0<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(2);   
 //adc input ain7   
 rADCCON = (1<<14)|(39<<6)|(7<<3)|(0<<2)|(1<<1)|(0);   
 rADCDAT1;   
 delay(10);   
 for(i = 0,g\_nPosY=0; i<nSampleNo; i++)   
 {   
 while(!(0x8000 & rADCCON)); // Check ECFLG   
 szPos[i] = (0x3ff & rADCDAT1);   
 g\_nPosY += szPos[i];   
   
 }   
 g\_nPosY = g\_nPosY/nSampleNo;   
 // g\_nPosY = (g\_nPosY-70)\*480/340;   
 uart\_printf(" Y-Posion[AIN7] is %04d\n", g\_nPosY);   
   
 //GND,AIN,Hi-z,AIN,Pullup En,Normal,Waiting mode   
 rADCTSC=(1<<8)|(1<<7)|(1<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(3);   
 }   
#ifdef BOARDTEST   
 sprintf(&szPos, "(X1,Y1):(%d,%d)",szPos[30],szPos[34]);   
 print\_lcd(195,170,0x20,&szPos);   
 sprintf(&szPos, "(X2,Y2):(%d,%d)",g\_nPosX,g\_nPosY);   
 print\_lcd(195,178,0x1c,&szPos);   
#endif   
 rSUBSRCPND |= BIT\_SUB\_TC;   
 rINTSUBMSK = ~(BIT\_SUB\_TC); // Unmask sub interrupt (TC)   
 ClearPending(BIT\_ADC);   
   
 is\_touched = 1;   
   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* name: tsp\_test   
\* func:   
\* para: none   
\* ret: none   
\* modify:   
\* comment:   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void tsp\_test(void)   
{   
 int status = 0;   
 int is\_back = 0;   
 uart\_printf("\n LCD display Test Example (please look at LCD screen)\n");   
   
 Lcd\_port\_init();   
 Lcd\_Init\_16Bit\_320240();   
 Lcd\_Clear(0x0a);   
 Lcd\_PowerEnable(0, 1);   
 Lcd\_EnvidOnOff(1);   
 uart\_printf(" [TFT 64K COLOR(16bit/1pixel) LCD TEST]\n");   
   
 uart\_printf(" Touch Screen Test Example.\n");   
   
   
   
   
 rADCDLY = 50000; // ADC Start or Interval Delay   
 rGPGCON |= 0xFF000000;   
 rGPGUP = 0xFFFF;   
   
 rADCTSC = (0<<8) | (1<<7) | (1<<6) | (0<<5) | (1<<4) | (0<<3) | (1<<2) | (0);   
 //auto sequential x/y position conversion,no operation, XP pull-up   
   
 rADCCON = (1<<14) | (ADCPRS<<6) | (5<<3) | (0<<2) | (0<<1) | (0);   
 // Enable Prescaler,Prescaler,AIN7/5 fix,Normal,Disable read start,No operation   
   
 rADCTSC = (0<<8) | (1<<7) | (1<<6) | (0<<5) | (1<<4) | (0<<3) | (0<<2) | (3);   
 //YM:GND,YP:AIN5,XM:Hi-z,XP:external voltage,XP pullup En,AUTO sequential,Waiting for interrupt mode   
 delay(100);   
 pISR\_ADC = (UINT32T)tsp\_int;   
 rINTMSK &= ~(BIT\_ADC);   
 rINTSUBMSK = ~(BIT\_SUB\_TC);   
 g\_nKeyPress = 1;   
   
 BitmapView\_320240(gImage\_1);   
   
 while(1) // only for board test to exit   
 {   
 // or press any key to exit   
 is\_back = 0;   
 if(is\_touched){   
 if (g\_nPosX > 750 && g\_nPosY > 850){   
 status -= 1;   
 is\_back = 1;   
 }   
 if (!is\_back){   
 switch(status){   
 case 0:   
 status = 1;   
 break;   
 case 1:   
 if (g\_nPosX<900 && g\_nPosX > 660 && g\_nPosY < 300 && g\_nPosY > 100){   
 status = 2;   
 }   
 break;   
 case 2:   
 if (g\_nPosY > 600){   
 status = 3;   
 }   
 break;   
   
 }   
 }   
 switch(status){   
 case 1:   
 BitmapView\_320240(gImage\_2);   
 break;   
 case 2:   
 BitmapView\_320240(gImage\_3);   
 break;   
 case 3:   
 BitmapView\_320240(gImage\_4);   
 break;   
   
 }   
 is\_touched = 0;   
   
 }   
 }   
 rINTSUBMSK |= BIT\_SUB\_TC;   
 rINTMSK |= BIT\_ADC;   
 uart\_printf(" end.\n");   
}

## 总结

本次实验，通过对触摸屏的控制和判断，设计了一个可交互的界面。通过这次实验，我掌握了触摸屏（电阻屏）的基本工作原理，并修改中断处理函数，在主函数中检测是否触发中断，并编写界面切换业务逻辑，实现了多界面的交互功能。看到自己编写的界面运行在实验板上，有着极大的成就感，这跟激发了我对嵌入式学习的兴趣。

# 实验四 μCOS-II系统原理实验

## 实验目的

* 理解任务管理的基本原理，了解任务的各个基本状态及其变迁过程；
* 掌握µCOS-II中任务管理的基本方法；
* 创建、启动、挂起、解挂任务
* 掌握µCOS-II中任务使用信号量的一般原理。

## 实验内容

* 设计多个应用任务，验证任务管理函数；
* 通过实现“哲学家就餐”问题，验证信号量对任务间互斥访问的支持；
* 应用信号量实现任务间的同步。
* 设计7个任务，并顺序执行

## 实验原理

1. **OSTaskCreate**（）

建立一个新任务。任务的建立可以在多任务环境启动之前，也可以在正在运行的任务中建立。中断处理程序中不能建立任务。一个任务可以为无限循环的结构。函数原型：**INT8U OSTaskCreate(void (\*task)(void \*pd), void \*pdata, OS\_STK \*ptos,**

**INT8U prio);**

参数说明：task 是指向任务代码首地址的指针。

**Pdata**：指向一个数据结构，该结构用来在建立任务时向任务传递参数。

返回值：

**OSTaskCreate**（）的返回值为下述之一：

* + OS\_NO\_ERR：函数调用成功。
  + OS\_PRIO\_EXIST：具有该优先级的任务已经存在。
  + OS\_PRIO\_INVALID：参数指定的优先级大于 OS\_LOWEST\_PRIO。
  + OS\_NO\_MORE\_TCB：系统中没有 OS\_TCB 可以分配给任务了。

1. **OSTaskSuspend**（）

无条件挂起一个任务。调用此函数的任务也可以传递参数 OS\_PRIO\_SELF，挂起调用任务本身。当前任务挂起后，只有其他任务才能唤醒被挂起的任务。任务挂起后，系统会重新进行任务调度，运行下一个优先级最高的就绪任务。唤醒挂起任务需要调用函数 OSTaskResume（）。

任务的挂起是可以叠加到其他操作上的。例如，任务被挂起时正在进行延时操作，那么任务的唤醒就需要两个条件：延时的结束以及其他任务的唤醒操作。又如，任务被挂起时正在等待信号量，当任务从信号量的等待对列中清除后也不能立即运行，而必须等到被唤醒后。

函数原型：**INT8U** OSTaskSuspend **( INT8U prio);**

参数说明：**prio** 为指定要获取挂起的任务优先级，也可以指定参数 OS\_PRIO\_SELF，挂起任务本身。此时，下一个优先级最高的就绪任务将运行。

返回值：

OSTaskSuspend（）的返回值为下述之一：

* OS\_NO\_ERR：函数调用成功。
* OS\_TASK\_ SUSPEND\_IDLE：试图挂起 μC/OS-II 中的空闲任务（Idle task）。此为非法操作。
* OS\_PRIO\_INVALID：参数指定的优先级大于 OS\_LOWEST\_PRIO 或没有设定

OS\_PRIO\_SELF 的值。

* OS\_TASK\_ SUSPEND \_PRIO：要挂起的任务不存在。

**3. OSTaskResume**（）

唤醒一个用 OSTaskSuspend（）函数挂起的任务。OSTaskResume（）也是唯一能“解挂” 挂起任务的函数。

函数原型：NT8U **OSTaskResume ( INT8U prio);**

参数说明：prio 指定要唤醒任务的优先级。返回值：

OSTaskResume （）的返回值为下述之一：

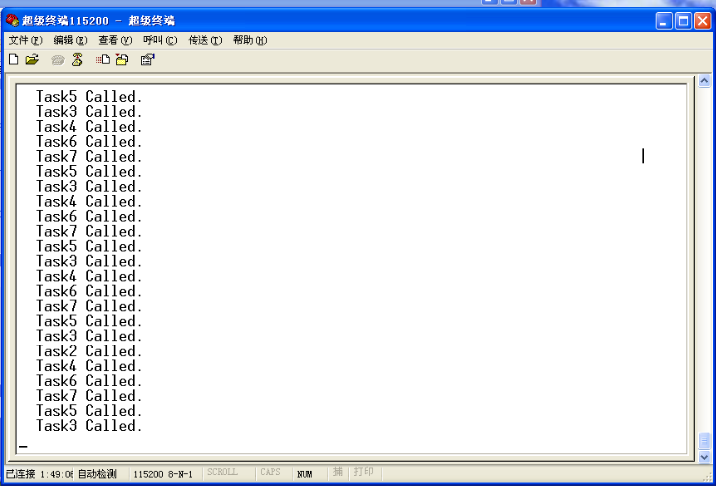
* OS\_NO\_ERR：函数调用成功。
* OS\_TASK\_RESUME\_PRIO：要唤醒的任务不存在。
* OS\_TASK\_NOT\_SUSPENDED：要唤醒的任务不在挂起状态。
* OS\_PRIO\_INVALID：参数指定的优先级大于或等于 OS\_LOWEST\_PRIO。

## 实现步骤

1. 验证示例源码
   * 拷贝整个实验例程源码目录到本地磁盘自己的工作目录下；
   * 使用μVision IDE for ARM 通过ULINK2仿真器连接实验板，打开实验例程目录04-uCOS\2.1*Task*test 子目录下的ucos2.Uv2例程，编译链接工程；
   * 将程序下载到实验平台的 NorFlash 中，观察串口输出；
   * 打开实验例程目录\04-uCOS\2.3*Semaphore*test子目录下的ucos2.Uv2例程，编译链接工程；
   * 下载调试，观察结果
2. 设计实现一个多任务应用程序
   * 拷贝示例实验源码工程
   * 设计7个任务，并用信号量实现7个任务顺序执行，将执行结果在串口上打印出来。

## 实验结果

程序编写完，进行编译链接，写入实验板中，实验板连接串口线，并将实验板复位，打开超级终端，可以观察到 7 个 task 任务输出。



## 程序说明

需要为每一个 task 定义其堆栈：

OS\_STK Stack1[STACKSIZE];   
OS\_STK Stack2[STACKSIZE];   
OS\_STK Stack3[STACKSIZE];   
OS\_STK Stack4[STACKSIZE];   
OS\_STK Stack5[STACKSIZE];   
OS\_STK Stack6[STACKSIZE];   
OS\_STK Stack7[STACKSIZE];

向每一个 task 传入参数，参数值为 task 的任务编号，task 会根据该编号，向终端输出对应的Task Called 语句。

const char Id1 = '1';   
const char Id2 = '2';   
const char Id3 = '3';   
const char Id4 = '4';   
const char Id5 = '5';   
const char Id6 = '6';   
const char Id7 = '7';

Task 中对 信号量 UART\_sem 进行请求，获得信号量后，向终端输出执行语句。

void Task1(void \*Id)   
{   
 while(1)   
 {   
 /\* print task's id \*/   
 OSSemPend(UART\_sem, 0, &err);   
 //uart\_sendstring(" Task1 Called.\r\n");   
 OSSemPost(UART\_sem);   
 }   
}

在 Main 函数中创建 TaskStart 任务， TaskStart 依次创建完 7 个任务后销毁。

void TaskStart(void \*Id)   
{   
 Init\_Timer4();   
   
 /\*create the first Semaphore in the pipeline with 1 to get the task started. \*/   
 UART\_sem = OSSemCreate(1);   
   
 /\*create the tasks in uC/OS and assign decreasing priority to them \*/   
 OSTaskCreate(Task1, (void \*)&Id1, &Stack1[STACKSIZE - 1], 2);   
 OSTaskCreate(Task2, (void \*)&Id2, &Stack2[STACKSIZE - 1], 3);   
 OSTaskCreate(Task3, (void \*)&Id3, &Stack3[STACKSIZE - 1], 4);   
 OSTaskCreate(Task4, (void \*)&Id4, &GetNumStack[STACKSIZE - 1], 5);   
 OSTaskCreate(Task5, (void \*)&Id5, &CaculStack[STACKSIZE - 1], 6);   
 OSTaskCreate(Task6, (void \*)&Id6, &ShowStack[STACKSIZE - 1], 7);   
 OSTaskCreate(Task7, (void \*)&Id7, &ShowStack[STACKSIZE - 1], 8);   
   
 OSTaskDel(OS\_PRIO\_SELF); // Delete current task   
}

## 总结

本次实验是我第一次在真实设备上对嵌入式操作系统进行学习了解，我在真实的设备上，实现了课程中所学的嵌入式操作系统多任务机制，在完成实验之余，惊叹于 μCOS 设计实现的精巧，也更加深刻的体会到嵌入式编程和 PC 机编程的不同之处。

# 实验五 μCOS-II硬件接口实验

## 实验目的

* 理解任务管理的基本原理，掌握µCOS-II中任务管理的基本方法；
* 掌握µCOS-II中任务间通信的一般原理和方法；
* 掌握嵌入式系统中LCD与键盘控制的一般方法。

## 实验内容

* 设计多个应用任务，用其中一个任务控制LED灯的状态；
* 设计多个应用任务，用其中一个任务读取键盘键值，键盘的响应用中断实现；
* 实现一个简易的计算器。
  + Task1：键盘
  + Task2：流水灯
  + Task3：LCD显示

## 实验原理

## 实现步骤

1. 验证示例源码
   * 拷贝整个实验例程源码目录到本地磁盘自己的工作目录下；
   * 使用μVision IDE for ARM 通过ULINK2仿真器连接实验板，打开实验例程目录04-uCOS\3.1*LED*test子目录下的ucos2.Uv2例程，编译链接工程；
   * 将程序下载到实验平台的 NorFlash 中，观察实验结果；
   * 打开实验例程目录\04-uCOS\3.3*keyboard*test子目录下的ucos2.Uv2例程，编译链接工程；
   * 下载调试，观察结果。
2. 设计实现一个简易的计算器
   * 拷贝示例实验源码工程3.3*keyboard*test；
   * 添加LCD驱动程序（参考11*LCD*Test）；
   * 设计多任务。至少LCD显示用一个任务，键盘解析用一个。

## 实验结果



## 程序说明

实验采用多任务的方式实现计算器功能。

定义相关任务如下：

* GetNumTask：从键盘读入数字和运算符
* CaculTask：完成计算
* ShowTask ：显示读入的数字和运算结果

**GetNumTask**：这个函数使用 三个信号量 kbd\_sem, symbol\_sem，num\_sem，

修改按键中断，在按键中断返回前，对 kbd\_sem 执行 p 操作。p 操作执行后会唤醒 GetNumTask 对键盘读入的字符进行处理。

{   
 uart\_sendstring("press key ");   
 uart\_sendstring(&ucChar);   
 uart\_sendstring("\r\n");   
 PRESS\_KEY = ucChar;   
 OSSemPost(kbd\_sem);   
 }

若按键输入为数字，则更新当前输入的数字，若按键输入为运算符，则表示当前输入数字结束，GetNumTask 对num*sem信号量和 symbol*sem 信号量执行 v 操作，唤醒 CaculTask 执行相关的计算逻辑

if (PRESS\_KEY!='+' && PRESS\_KEY!='-' && PRESS\_KEY!='\*' && PRESS\_KEY!='/' && PRESS\_KEY!='E'){   
 num = num \* 10 + PRESS\_KEY - '0';   
 }   
 else{   
 OSSemPost(symbol\_sem);   
 NUM = num;   
 OSSemPost(num\_sem);   
 num = 0;   
 }

**CaculTask** 对 GetNumTask 提供的数字和运算符进行运算，对 num*sem ，symbol*sem 进行 p 操作，获取到信号量之后，访问公共全局变量，获取当前的输入数值，判断输入的运算符，完成对应计算。计算完成后，对 show*ready*sem 执行 p 操作，查询 LCD 显示程序是否就绪，若就绪，改写 SHOW\_CHAR 全局变量为当前待显示的字符，释放 show\_sem，唤醒 ShowTask 完成字符显示。

void CaculTask(void \*Id)   
{   
 int num1 = 0;   
 int num2 = 0;   
 int rs = 0;   
 char c[20];   
 UINT8T symbol;   
 int i;   
 while(1){   
 ///////////////////////////   
 OSSemPend(num\_sem, 0, &err);   
 num1 = NUM;   
 uart\_sendstring(" GET num 1 = ");   
 sprintf(c,"%d",num1);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 /////////////////////////   
 OSSemPend(symbol\_sem , 0, &err);   
 symbol = PRESS\_KEY;   
 ////////////////////////////////   
 OSSemPend(num\_sem, 0, &err);   
 num2 = NUM;   
 uart\_sendstring(" GET num 2 = ");   
 sprintf(c,"%d",num2);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 ////////////////////////   
 OSSemPend(symbol\_sem, 0, &err);   
 switch(symbol){   
 case '+':   
 rs = num1 + num2;   
 break;   
 case '-':   
 rs = num1 - num2;   
 break;   
 case '\*':   
 rs = num1 \* num2;   
 break;   
 case '/':   
 rs = num1 / num2;   
 break;   
   
 }   
 uart\_sendstring(" The result = ");   
 sprintf(c,"%d",rs);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 for(i = 0; i < strlen((const char\*)c); i++){   
 OSSemPend(show\_ready\_sem, 0, &err);   
 SHOW\_CHAR = c[i];   
 OSSemPost(show\_sem);   
 }   
 OSSemPend(show\_ready\_sem, 0, &err);   
 SHOW\_CHAR = '\n';   
 OSSemPost(show\_sem);   
 }   
}

定义初始化任务，在初始化任务中初始化全部信号量，并创建对应的任务函数，创建完成后删除初始化任务自身，新创建的任务保持运行状态，检测用户输入，计算运算结果，完成 LCD 显示

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 文件: Main.C   
\* 描述:application route   
\* 编写:   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
   
#include "Includes.h"   
#include "2410lib.h"   
   
 #define M\_MDIV 0xa1   
 #define M\_PDIV 0x3   
 #define M\_SDIV 0x1   
#define UINT32T unsigned int   
#define UINT8T unsigned char   
   
OS\_EVENT \*UART\_sem;   
OS\_EVENT \*kbd\_sem;   
OS\_EVENT \*symbol\_sem;   
OS\_EVENT \*num\_sem;   
OS\_EVENT \*InterruptSem;   
OS\_EVENT \*show\_ready\_sem;   
OS\_EVENT \*show\_sem;   
   
UINT8T PRESS\_KEY;   
UINT8T SHOW\_CHAR;   
int NUM;   
   
unsigned char err;   
   
/\* allocate memory for tasks' stacks \*/   
#define STACKSIZE 256   
   
/\* Global Variable \*/   
OS\_STK Stack1[STACKSIZE];   
OS\_STK Stack2[STACKSIZE];   
OS\_STK Stack3[STACKSIZE];   
OS\_STK ShowStack[STACKSIZE];   
OS\_STK CaculStack[STACKSIZE];   
OS\_STK GetNumStack[STACKSIZE];   
OS\_STK StackMain[STACKSIZE];   
   
const char Id1 = '1';   
const char Id2 = '2';   
const char Id3 = '3';   
const char Id4 = '4';   
const char Id5 = '5';   
const char Id6 = '6';   
const char Id7 = '7';   
   
   
unsigned char err;   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
   
void led\_init()   
{   
 rGPBCON |= ((0x1<<14)| (0x1<<16)|(0x1<<18)|(0x1<<20)) ;   
 rGPBCON &= ~((0x1<<15)| (0x1<<17)|(0x1<<19)|(0x1<<21)) ;   
}   
   
   
 void led\_off()   
 {   
 rGPBDAT |= ((0x1<<7)| (0x1<<8)|(0x1<<9)|(0x1<<10)) ;   
   
}   
   
   
void led\_on()   
{   
 rGPBDAT &= ~((0x1<<7)| (0x1<<8)|(0x1<<9)|(0x1<<10)) ;   
   
}   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task1(void \*Id).   
\* 描述: 任务1函数.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void Task1(void \*Id)   
{   
 /\* print task's id \*/   
 OSSemPend(UART\_sem, 0, &err);   
 //uart\_sendstring(" Task1 Called.\r\n");   
 OSSemPost(UART\_sem);   
   
 while(1)   
 {   
 keyboard\_test();   
 OSTimeDly(1);   
   
 }   
   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task2(void \*Id).   
\* 描述: 任务2函数.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void Task2(void \*Id)   
{   
 /\* print task's id \*/   
 OSSemPend(UART\_sem, 0, &err);   
 //uart\_sendstring(" Task2 Called.\r\n");   
 OSSemPost(UART\_sem);   
   
 while(1)   
 {   
 uart\_sendstring(" Task2 Called.\r\n");   
 led\_on();   
 OSTimeDly(50);   
 led\_off();   
 OSTimeDly(50);   
 }   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task3(void \*Id).   
\* 描述: 任务3函数.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void Task3(void \*Id)   
{   
 /\* print task's id \*/   
 OSSemPend(UART\_sem, 0, &err);   
 uart\_sendstring(" Task3 Called.\r\n");   
 OSSemPost(UART\_sem);   
   
 while(1)   
 {   
 //color\_lcd\_test();   
 uart\_sendstring(" Task3 Called.\r\n");   
 OSTimeDly(30);   
 }   
}   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task3(void \*Id).   
\* 描述: show char.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void ShowTask(void \*Id)   
{   
 char show[20];   
 int posx = 0;   
 int posy = 0;   
 while(1)   
 {   
 //OSSemPend(kbd\_sem, 0, &err);   
 //uart\_sendstring(&PRESS\_KEY);   
 OSSemPost(show\_ready\_sem);   
 OSSemPend(show\_sem, 0, &err);   
 posx+=10;   
 if (SHOW\_CHAR == 'E'){   
 posy += 30;   
 if (posy >= 200){   
 posy = 25;   
 Lcd\_Clear(0);   
 }   
 posx = 0;   
 Lcd\_DspAscII8X16(posx, posy, 0xf8002, "=");   
 continue;   
 }   
 if (SHOW\_CHAR == '\n'){   
 posy += 30;   
 if (posy >= 200){   
 posy = 25;   
 Lcd\_Clear(0);   
 }   
 posx = 0;   
 continue;   
 }   
 sprintf(show,"%c",SHOW\_CHAR);   
 Lcd\_DspAscII8X16(posx, posy, 0x1f, show);   
 Lcd\_DspHz24(100, 200, 0x1f, "历史记录计算器");   
 }   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task3(void \*Id).   
\* 描述: show char.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void GetNumTask(void \*Id)   
{ int num = 0;   
 while(1)   
 {   
 OSSemPend(kbd\_sem, 0, &err);   
 uart\_sendstring(&PRESS\_KEY);   
 if (PRESS\_KEY =='F'){   
 PRESS\_KEY = '/';   
 }   
 if (PRESS\_KEY!='+' && PRESS\_KEY!='-' && PRESS\_KEY!='\*' && PRESS\_KEY!='/' && PRESS\_KEY!='E'){   
 num = num \* 10 + PRESS\_KEY - '0';   
 }   
 else{   
 OSSemPost(symbol\_sem);   
 NUM = num;   
 OSSemPost(num\_sem);   
 num = 0;   
 }   
 OSSemPend(show\_ready\_sem, 0, &err);   
 SHOW\_CHAR = PRESS\_KEY;   
 OSSemPost(show\_sem);   
 }   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void Task3(void \*Id).   
\* 描述: show char.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void CaculTask(void \*Id)   
{   
 int num1 = 0;   
 int num2 = 0;   
 int rs = 0;   
 char c[20];   
 UINT8T symbol;   
 int i;   
 while(1){   
 ///////////////////////////   
 OSSemPend(num\_sem, 0, &err);   
 num1 = NUM;   
 uart\_sendstring(" GET num 1 = ");   
 sprintf(c,"%d",num1);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 /////////////////////////   
 OSSemPend(symbol\_sem , 0, &err);   
 symbol = PRESS\_KEY;   
 ////////////////////////////////   
 OSSemPend(num\_sem, 0, &err);   
 num2 = NUM;   
 uart\_sendstring(" GET num 2 = ");   
 sprintf(c,"%d",num2);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 ////////////////////////   
 OSSemPend(symbol\_sem, 0, &err);   
 switch(symbol){   
 case '+':   
 rs = num1 + num2;   
 break;   
 case '-':   
 rs = num1 - num2;   
 break;   
 case '\*':   
 rs = num1 \* num2;   
 break;   
 case '/':   
 rs = num1 / num2;   
 break;   
   
 }   
 uart\_sendstring(" The result = ");   
 sprintf(c,"%d",rs);   
 uart\_sendstring(c);   
 uart\_sendstring("\r\n");   
 for(i = 0; i < strlen((const char\*)c); i++){   
 OSSemPend(show\_ready\_sem, 0, &err);   
 SHOW\_CHAR = c[i];   
 OSSemPost(show\_sem);   
 }   
 OSSemPend(show\_ready\_sem, 0, &err);   
 SHOW\_CHAR = '\n';   
 OSSemPost(show\_sem);   
 }   
}   
   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: void TaskStart(void \*Id).   
\* 描述: 任务起始函数.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void TaskStart(void \*Id)   
{   
 Init\_Timer4();   
   
 /\*create the first Semaphore in the pipeline with 1 to get the task started. \*/   
 UART\_sem = OSSemCreate(1);   
 kbd\_sem = OSSemCreate(0);   
 num\_sem = OSSemCreate(0);   
 symbol\_sem = OSSemCreate(0);   
 show\_sem = OSSemCreate(0);   
 show\_ready\_sem = OSSemCreate(0);   
   
 /\*create the tasks in uC/OS and assign decreasing priority to them \*/   
 OSTaskCreate(Task1, (void \*)&Id1, &Stack1[STACKSIZE - 1], 2);   
 OSTaskCreate(Task2, (void \*)&Id2, &Stack2[STACKSIZE - 1], 3);   
 OSTaskCreate(Task3, (void \*)&Id3, &Stack3[STACKSIZE - 1], 4);   
 OSTaskCreate(GetNumTask, (void \*)&Id4, &GetNumStack[STACKSIZE - 1], 5);   
 OSTaskCreate(CaculTask, (void \*)&Id5, &CaculStack[STACKSIZE - 1], 6);   
 OSTaskCreate(ShowTask, (void \*)&Id6, &ShowStack[STACKSIZE - 1], 7);   
   
 OSTaskDel(OS\_PRIO\_SELF); // Delete current task   
}   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* 函数: int main (void).   
\* 描述: 主程序函数.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
void Main (void)   
{   
 change\_clock\_divider(1,1); // 1:2:4   
 change\_value\_MPLL(M\_MDIV, M\_PDIV, M\_SDIV); // Fin=12MHz FCLK=202.8MHz   
 delay(0); // adjust the delay count   
   
   
 port\_init();   
 led\_init();   
 uart\_init(PCLK, 115200, UART0);   
 uart\_select(UART0);   
 uart\_sendstring("\r\nEmboard Emsbc2410 boot success!\r\n");   
 uart\_sendstring("uCOS-II Running...\r\n");   
 keyboard\_init();   
 color\_lcd\_init();   
   
 OSInit();   
 OSTimeSet(0);   
   
 /\* create the start task \*/   
 OSTaskCreate(TaskStart,(void \*)0, &StackMain[STACKSIZE - 1], 0);   
   
 /\* start the operating system \*/   
 OSStart();   
}   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\* End.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## 总结

本次实验是这五次嵌入式实验中最复杂的一个实验，这次实验里，我更进一步的了解了 μCOS 的多任务机制，以及多任务之间的通信机制。本次实验将一个计算器程序，拆分成多个任务模块，在实验板上同时运行。这种实现方式和我之前掌握的编程思路不同。通过认真思索，按照老师的引导下，对任务功能进行了合理的拆分，并设计多个信号量与全局变量，实现了多个不同任务之间的同步互斥机制和数据通信。并结合之前实验中的中断等控制硬件进行交互的方法，实现了简易计算器的设计。由于时间的关系，程序还是很简陋，只能实现两个多位数的四则运算，但是从其中我掌握了多任务实现功能目标的方法，以及信号量等通信机制，对课程中所学的内容有了深刻的印象和具体的实现，我想这是我在本次实验中最大的收获。

至此，嵌入式系统实现已经全部完成，在这五次实验里面，我对嵌入式系统有了更新以及更深刻的认识，虽然有了巨大的收获，但是由于实验时间的限制，我知道我所学的东西仍有很大不足，但这毕竟为我以后的学习打下了基础。

通过这五次实验以及嵌入式课程所学的知识，改变了我对嵌入式这一领域的看法，嵌入式不仅仅是和硬件打交道的一门学科，由于硬件和使用场景的限制，嵌入式系统能使用的资源极其有限，在这有限的资源中，对任务进行实时、稳定的实现，从硬件设计到操作系统，到软件编写，无不透漏出设计的精妙。学习这门课程，不但使我对嵌入式领域有了更深的认识，嵌入式设计中的思想，无疑会对我以后的学习，研究和工作大有裨益。