
Tiva 实验指导书

2014 年 2 月 27 日

目录

第 1 章	GPIO 中断实验	4
1.1	实验介绍	4
1.2	实验目的	4
1.3	实验原理	4
1.4	实验步骤和流程图	5
1.5	实验现象	5
1.6	关键代码分析	5
第 2 章	Timer 中断计数实验	7
2.1	实验介绍	7
2.2	实验目的	7
2.3	实验原理	7
2.4	实验步骤和流程图	7
2.5	实验现象	8
2.6	关键代码分析	9
第 3 章	12864 点阵 LCD 文本显示	11
3.1	实验介绍	11
3.2	实验目的	11
3.3	实验原理	11
3.4	实验步骤和流程图	12
3.5	实验现象	13
3.6	关键代码分析	14
第 4 章	看门狗定时器	18
4.1	实验介绍	18
4.2	实验目的	18
4.3	实验原理	18
4.4	实验步骤和流程图	19
4.5	关键代码分析	19
第 5 章	PWM 实验	21
5.1	实验介绍	21
5.2	实验目的	21
5.3	实验原理	21
5.4	实验步骤和流程图	22
5.5	实验现象	23
5.6	关键代码分析	23
第 6 章	UART 发送与接收实验	26
6.1	实验介绍	26
6.2	实验目的	26
6.3	实验原理	26
6.4	实验步骤和流程图	27
6.5	实验现象	28
6.6	关键代码分析	28
第 7 章	SPI 模式读写 SD 实验	30

7.1	实验介绍.....	30
7.2	实验目的.....	30
7.3	实验原理.....	30
7.4	实验步骤.....	32
7.5	实验代码流程图.....	33
7.6	实验现象.....	33
7.7	思考题.....	34
第 8 章	LM75A 温度采集实验.....	35
8.1	实验介绍.....	35
8.2	实验目的.....	36
8.3	实验原理.....	36
8.4	实验步骤及流程图.....	37
8.5	实验现象.....	38
8.6	关键代码分析.....	38
第 9 章	电位器输入实验.....	40
9.1	实验介绍.....	40
9.2	实验目的.....	40
9.3	实验原理.....	40
9.4	实验步骤及流程图.....	41
9.5	实验现象.....	42
9.6	关键代码分析.....	42
第 10 章	AC 模拟比较器.....	44
10.1	实验介绍.....	44
10.2	实验目的.....	44
10.3	实验原理.....	44
10.4	实验步骤及流程图.....	44
10.5	实验现象.....	45
10.6	关键代码分析.....	45
第 11 章	DAC 实验—扬声器播放.....	47
11.1	实验介绍.....	47
11.2	实验目的.....	47
11.3	实验原理.....	47
11.4	实验步骤.....	48
11.5	实验现象.....	49
11.6	实验流程图.....	50
11.7	关键代码分析.....	50
11.8	思考题.....	51

第 1 章 GPIO 中断实验

1.1 实验介绍

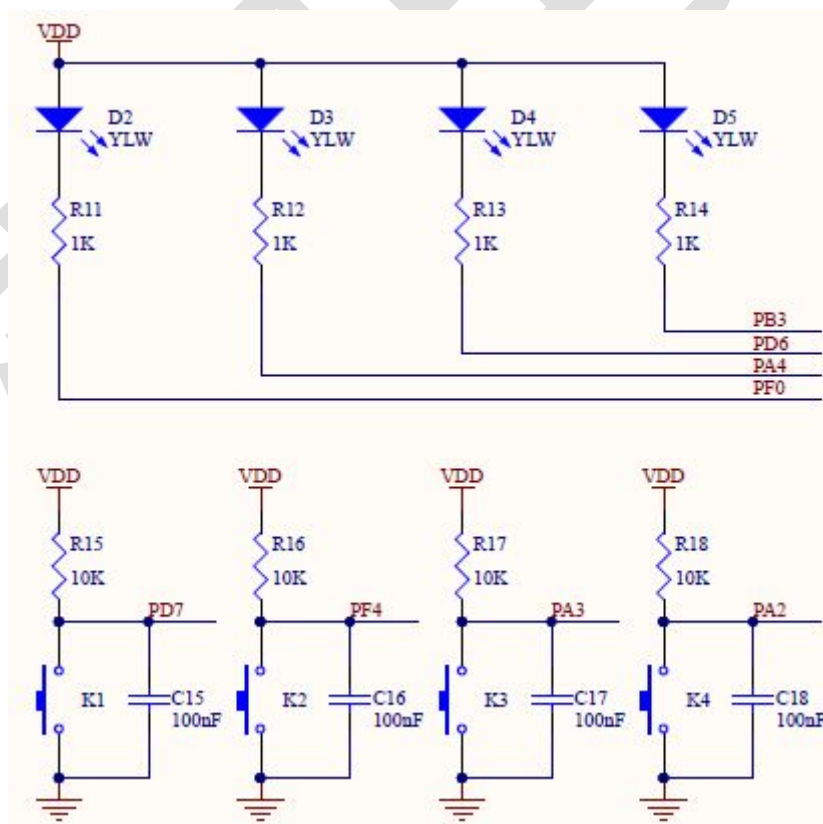
本实验通过中断的方式演示了按键的操作。与查询方式相比，中断按键具有响应速度快，占用 MCU 资源少的优点。按键产生中断，MCU 就会进入程序的中断处理函数进行中断处理，结束之后再返回到中断之前的位置。

1.2 实验目的

- (1). 熟悉 Keil 开发环境的使用；
- (2). 了解 Tiva C 系列 LaunchPad 的中断系统；
- (3). 掌握 LaunchPad 的中断编程方法。

1.3 实验原理



各个 LED 灯和按键对应的 GPIO 如下图所示：



电路原理图

本实验中选择按键 K2 和 D3 作为实验资源。

1.4 实验步骤和流程图

- (1). 将 LaunchPad 通过 USB 线连接 PC
- (2). 打开 Keil 集成开发工具，打开 GPIO 中断的示例 project。
- (3). 选择  对工程进行编译链接，成功后点击  将程序烧写入开发板，按下开发板的复位键，运行程序。

初始化流程：

- (1). 设备时钟设置
- (2). 按键和 LED 灯对应的 GPIO 初始化配置
- (3). 中断函数初始化、使能
- (4). 按键产生中断

实验流程图如下图所示：



实验流程图

1.5 实验现象

按下 K2 后，D3 的状态反转（即暗变亮，亮变暗）。

1.6 关键代码分析

```
//this code is the handler function for key1  
void Key2IntHandler()
```

```
{
GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_4,
              (0x10^GPIOPinRead(GPIO_PORTA_BASE, GPIO_PIN_4)));
GPIOIntClear(GPIO_PORTF_BASE, GPIO_INT_PIN_4);
}

//this code is the main function
int main(void)
{
SysCtlClockSet(SYSCTL_SYSDIV_1|SYSCTL_USE_OSC|SYSCTL_XTAL_16MHZ
|SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_4);    //PA4
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4);    //PF4
GPIOIntRegister(GPIO_PORTF_BASE, Key2IntHandler);
GPIOIntTypeSet(GPIO_PORTF_BASE, GPIO_PIN_4, GPIO_FALLING_EDGE);    //PA2
GPIOIntEnable(GPIO_PORTF_BASE, GPIO_PIN_4);    //Enable Key4 handler
While(1)
{}
}
```

第2章 Timer 中断计数实验

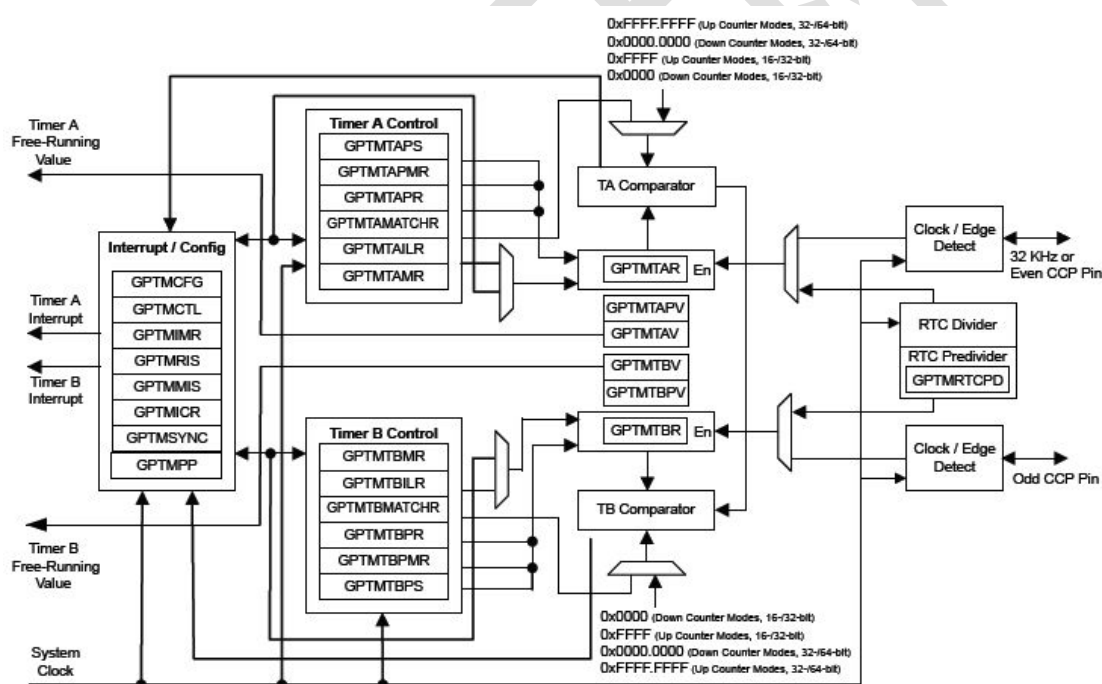
2.1 实验介绍

Tiva C 系列含有丰富的定时器资源，本实验运用 LaunchPad 的 Timer 模块自动产生中断并计数。

2.2 实验目的

- (1). 熟悉 Keil 开发环境的使用；
- (2). 熟悉 LaunchPad 的 Timer 的资源概况；
- (3). 掌握控制 LaunchPad 的 Timer 的步骤和方法。



2.3 实验原理



上图是通用定时器的结构图，实验中我们采用系统时钟作为 Timer 时钟源，利用 TimerB0 进行实验。定时器可以配置成单次/周期定时器模式、RTC 模式、输入边沿计数模式、输入边沿定时模式、PWM 模式。大家实验时要记得配置正确定时器模式。

2.4 实验步骤和流程图

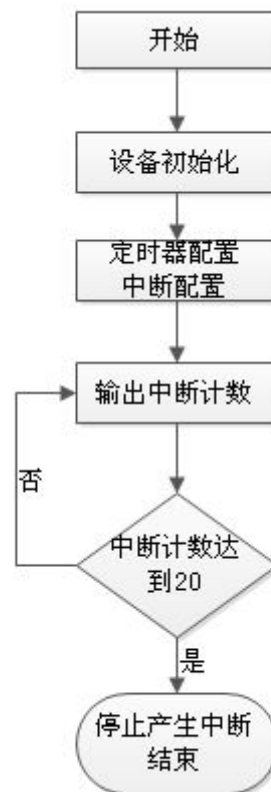
- (1). 将 LaunchPad 通过 USB 线连接 PC
- (2). 打开 Keil 集成开发工具，打开 Timer 的示例 project。

(3). 选择  对工程进行编译链接，成功后点击  将程序烧写入开发板，按下开发板的复位键，运行程序。

初始化流程：

- (1). 设备时钟设置
- (2). USART 外设使能
- (3). 定时器配置
- (4). 中断配置，中断处理函数注册
- (5). 使能定时器
- (6). 计数达到装载值时停止产生中断

实验流程如下图所示：



实验流程图

2.5 实验现象

本实验代码中加入了 UART 部分，用来显示程序的运行情况。如果运行正确，串口调试助手将显示中断计数，达到 20 之后停止。

2.6 关键代码分析

```
//配置 Timer0B 为周期性定时器，每隔 2s 触发一次中断。当 20 次中断之后，暂停触发//
中断
#define NUMBER_OF_INTS    20 //最大中断数
//定义变量计数中断的触发次数
static volatile uint32_t g_ui32Counter = 0;
//初始化 UART 用来显示程序运行情况
void InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTStdioConfig(0, 115200, 16000000);
}

//中断处理函数
void Timer0BIntHandler(void)
{
    //清除定时器中断标志
    TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    //更新中断计数值
    g_ui32Counter++;
    //判断是否达到最大计数值 20
    if(g_ui32Counter == NUMBER_OF_INTS)
    {
        IntDisable(INT_TIMER0B);
        //禁用 TIMER0 中断
        TimerIntDisable(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
        //清除挂起的中断
        TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    }
}

// main
int main(void)
{
    uint32_t ui32PrevCount = 0;
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
        SYSCTL_XTAL_16MHZ);
    //使能 TIMER0 外设
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
```

```
InitConsole();
//输出初始化信息
    UARTprintf("16-Bit Timer Interrupt ->");
    UARTprintf("\n    Timer = TimerOB");
    UARTprintf("\n    Mode = Periodic");
    UARTprintf("\n    Number of interrupts = %d", NUMBER_OF_INTS);
    UARTprintf("\n    Rate = 1ms\n\n");
    //配置 TIMEROB 定时器
TimerConfigure(TIMERO_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_B_PERIODIC);
    //配置定时器装载值
TimerLoadSet(TIMERO_BASE, TIMER_B, SysCtlClockGet() / 50);
    //使能中断
IntMasterEnable();
//配置 TIMEROB 中断事件为定时器超时
    TimerIntEnable(TIMERO_BASE, TIMER_TIMB_TIMEOUT);
    //使能 NVIC 中的 TIMEROB 中断
IntEnable(INT_TIMEROB);
    //为定时器中断指定中断处理函数
TimerIntRegister(TIMERO_BASE, TIMER_B, TimerOBIntHandler);
    g_ui32Counter = 0;
    TimerEnable(TIMERO_BASE, TIMER_B);
    while(1)
    {
        //当两值不相等时, 说明有新中断产生, 调用了中断处理函数
        if(ui32PrevCount != g_ui32Counter)
        {
            UARTprintf("Number of interrupts: %d\r", g_ui32Counter);
            ui32PrevCount = g_ui32Counter;
        }
    }
}
```

第 3 章 12864 点阵 LCD 文本显示

3.1 实验介绍

本节演示了 Tiva 实验板的 LCD 显示屏示例,通过 TM4C123G 控制 SSI 总线,驱动 UC1705 控制芯片对液晶屏进行驱动。本实例演示的为 LCD 串口模式,因此只需通过 SSI 总线像液晶屏发送 Command/Data 控制液晶屏的数据输出。

3.2 实验目的

- (1). 掌握 SSI 总线, 以及库函数对 SSI 的操作、使用;
- (2). 熟悉 LaunchPad 与 LCD 模块的交互;
- (3). 掌握控制 LCD 的数据格式与方法, 并学习使用如何通过 `printf` 函数将信息输出到液晶屏。

3.3 实验原理

12864 点阵液晶显示模块(LCM)就是由 128*64 个液晶显示点组成的一个 128 列*64 行的阵列。每个显示点对应一位二进制数, 1 表示亮, 0 表示灭。存储这些点阵信息的 RAM 称为显示数据存储单元。要显示某个图形或汉字就是将相应的点阵信息写入到相应的存储单元中。图形或汉字的点阵信息由自己设计, 问题的关键就是显示点在液晶屏上的位置(行和列)与其在存储器中的地址之间的关系。每屏有一个 512*8 bits 显示数据 RAM。左右半屏驱动电路及存储器分别由片选信号 CS1 和 CS2 选择。显示点在 128*64 液晶屏上的位置由行号(line, 0~63)与列号(column, 0~63)确定。512*8 bits RAM 中某个存储单元的地址由页地址(Xpage, 0~7)和列地址(Yaddress, 0~63)确定。每个存储单元存储 8 个液晶点的显示信息。

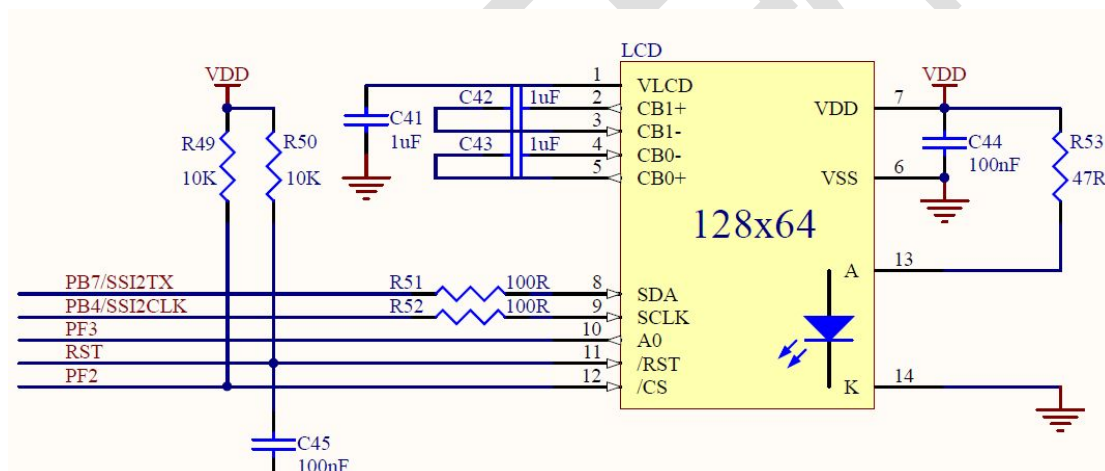
为了使液晶点位置信息与存储地址的对应关系更直观, 将 128*64 液晶屏从上至下 8 等分为 8 个显示块, 每块包括 8 行*64 列个点阵。每列中的 8 行点阵信息构成一个 8bits 二进制数, 存储在一个存储单元中。(注意: 二进制的高低有效位顺序与行号对应关系因不同商家而不同)存放一个显示块的 RAM 区称为存储页。即 128*64 液晶屏的点阵信息存储在 8 个存储页中, 每页 64 个字节, 每个字节存储一行(8 行)点阵信息。因此存储单元地址包括页地址(Xpage, 0~7)和列地址(Yaddress, 0~63)。例如点亮 128*64 的屏中(20, 30)位置上的液晶点, 因列地址 30 小于 64, 该点在左半屏第 29 列, 所以 CS1 有效; 行地址 20 除以 8 取整得 2, 取余得 4, 该点在 RAM 中页地址为 2, 在字节中的序号为 4; 所以将二进制数据 00010000 (也可能是 00001000, 高低顺序取决于制造商)写入 Xpage=2, Yaddress=29 的存储单元中即点亮(20, 30)上的液晶点。

字模虽然也是一组数字, 但它的意义却与数字的意义有了根本的变化, 它是用数字的各位信息来记载英文或汉字的形状, 如英文的'A'在字模的记载方式如图所示:

英文字模	位代码	字模信息
	0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0	0x00
	0 0 0 1 0 0 0	0x10
	0 0 1 1 1 0 0	0x38
	0 1 1 0 1 1 0	0x6c
	1 1 0 0 0 1 1	0xc6
	1 1 0 0 0 1 1	0xc6
	1 1 1 1 1 1 1	0xfe
	1 1 0 0 0 1 1	0xc6
	1 1 0 0 0 1 1	0xc6
	1 1 0 0 0 1 1	0xc6
	1 1 0 0 0 1 1	0xc6
	0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0	0x00

“A”字模图

在数字电路中，所有的数据都是以 0 和 1 保存的，对 LCD 控制器进行不同的数据操作，可以得到不同的结果。对于显示英文操作，由于英文字母种类很少，只需要 8 位（一字节）即可。而对于中文，常用却有 6000 以上，于是我们的 DOS 前辈想了一个办法，就是将 ASCII 表的高 128 个很少用到的数值以两个为一组来表示汉字，即汉字的内码。而剩下的低 128 位则留给英文字符使用，即英文的内码。



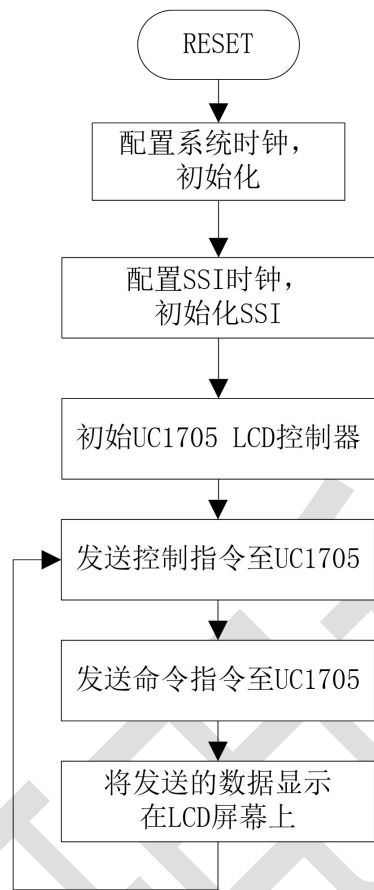
LCD 电路图

上图是 12864LCD 结构图，实验中我们采用 SSI2 作为数据传输总线，利用 PF3 作为 GPIO 输出，用于控制液晶屏传输的是数据还是控制指令。

3.4 实验步骤和流程图

- (1) 将 PC 和板载仿真器通过 USB 线相连；
- (2) 打开 Keil 项目工程文件
- (3) 编译项目工程，并烧写到 Tiva 开发板中
- (4) 观察 LCD 的屏幕显示内容。

实验流程如下图所示：



实验流程图

3.5 实验现象

本实验代码中加入了 UART 部分，用来显示程序的运行情况。如果运行正确，串口调试助手将显示中断计数，达到 20 之后停止。



实验效果

3.6 关键代码分析

UC1701 初始化、显示数据发送代码

```
void UC1701Init(unsigned long ulSpiClock)
{
    //
    // 使能 UC1701 GPIO 输入
    //
    //MAP_SysCtlPeripheralEnable(LCD_PERIPH_BKL);
    MAP_SysCtlPeripheralEnable(LCD_PERIPH_SPI_CS);
    MAP_SysCtlPeripheralEnable(LCD_PERIPH_CD);
    MAP_SysCtlPeripheralEnable(LCD_PERIPH_RESET);
    MAP_GPIOPinTypeGPIOOutput(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS);    // PB2 - CS

    //
    // 使能 CS
    MAP_GPIOPinTypeGPIOOutput(LCD_GPIO_CD, LCD_PIN_CD);    // PD2 - CD

    MAP_GPIOPinTypeGPIOOutput(LCD_GPIO_RESET, LCD_PIN_RESET);    // PE0 - RST

    //
    // 配置 SSI 时钟的模式、带宽
    //
    MAP_SSIConfigSetExpClk(LCD_PIN_SPI_PORT, SysCtlClockGet(),
    SSI_FRF_MOTO_MODE_3,
```

```

        SSI_MODE_MASTER, ulSpiClock, 8);
MAP_SSIEnable(LCD_PIN_SPI_PORT);

//
// 开机时关闭 UC1701
MAP_GPIOPinWrite(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS, LCD_PIN_SPI_CS); // CS

// Power on the bkl
//MAP_GPIOPinWrite(LCD_GPIO_BKL, LCD_PIN_BKL, LCD_PIN_BKL); // BKL

delay_ms(6); // wait > 5ms

// 向液晶屏发送初始化信息
UC1701CmdWrite(0xE2); // system reset
delay_ms(200);

UC1701CmdWrite(0xFA); // set adv program control
UC1701CmdWrite(0x93);

UC1701CmdWrite(0xA1); // set seg direction

UC1701CmdWrite(0xC8); // set com direction

UC1701CmdWrite(0x2F); // set power control ?!

UC1701CmdWrite(0xA2); // set lcd bias ratio

UC1701CmdWrite(0x81); // set electronic volume
UC1701CmdWrite(0x1F);

UC1701CmdWrite(0xAF); // set display enable
}

void UC1701DataWrite(unsigned char ucData)
{
    unsigned long ulTemp;

    //GPIOPinConfigure(SDC_RX_GPIO_CONFIG); // switch rx pin to ssi
mode
    //MAP_GPIOPinTypeGPIOOutput(LCD_GPIO_CD, LCD_PIN_CD); // PD2 - CD

    //
    //Step 1 Select Chip
    //

```

```
MAP_GPIOPinWrite(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS, ~LCD_PIN_SPI_CS); // CS

//
//Step 2 Set Display Enable
//
ulTemp = ucData;
MAP_GPIOPinWrite(LCD_GPIO_CD, LCD_PIN_CD, LCD_PIN_CD); // CD
MAP_SSIDataPut(LCD_PIN_SPI_PORT, ulTemp);

// TODO: use isr handle this
while(MAP_SSIBusy(LCD_PIN_SPI_PORT))
    ;

//
//Step 3 Disable chip select
//
MAP_GPIOPinWrite(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS, LCD_PIN_SPI_CS); // CS
}

// 向 UC1701 发送命令
void UC1701CmdWrite(unsigned char ucCmd)
{
    unsigned int ulTemp;
    //
    //Step 1 Select Chip
    //
    MAP_GPIOPinWrite(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS, ~LCD_PIN_SPI_CS); // CS

    //
    //Step 2 Send a command
    //
    ulTemp = ucCmd;
    MAP_GPIOPinWrite(LCD_GPIO_CD, LCD_PIN_CD, ~LCD_PIN_CD); // CD

    MAP_SSIDataPut(LCD_PIN_SPI_PORT, ulTemp);

    while(SSIBusy(LCD_PIN_SPI_PORT))
        ;

    //
    //Step 3 Disable chip select
    //
    MAP_GPIOPinWrite(LCD_GPIO_SPI_CS, LCD_PIN_SPI_CS, LCD_PIN_SPI_CS); // CS
}
```

第 4 章 看门狗定时器

4.1 实验介绍

本实验演示了看门狗定时器监控处理器的用法。实验设置的定时时间为 350ms，使能复位功能，配置后锁定。实验过程为解锁→喂狗→锁定，同时使 LED 闪亮。程序一开始便点亮 LED，延时，再熄灭，以表示已复位。然后在主循环里每隔 500ms 喂狗一次，由于看门狗具有二次超时特性，因此不会产生复位，除非喂狗间隔超过了 $2 \times 350\text{ms}$ 。

TM4C123GH6PM 微控制器包含了两个看门狗定时器模块，分别由系统时钟（WT0）和 PIOSC 时钟驱动（WT1）。与 WT0 不同的是，WDT1 处在不同的时钟域，因此 WDT1 需要同步。WDT1 的看门狗控制寄存器中有一位定义了对 WDT1 的写入是否已完成，通过访问此位可确保本次对寄存器的写入操作在下次写入前完成。

4.2 实验目的

- (1) 掌握看门狗的实验原理。
- (2) 熟练应用 TM4C123GH6PM 的 WDT 模块。
- (3) 该实验实现看门狗的超时复位功能。

4.3 实验原理

WDT（Watch Dog Timer）俗称看门狗，是一个定时器，只不过在定时到达时，可以复位 Launchpad。这个功能对于实际工程应用中的产品非常有用。在很多应用中，Launchpad 要连续工作，如果期间 Launchpad 由于各种意外死机，则 Launchpad 就会停止工作，有了看门狗，就可以避免这种意外的发生。

看门狗的原理为“定时喂狗，狗饿复位”；其主要原理如下：


1. Launchpad 都是循环工作的，比如完成整个循环所需时间最长不超过 0.5 秒，则可以把看门狗定时器的定时值设为 1 秒，在主循环中加入看门狗定时值清零的代码（俗称喂狗）。
2. 这样一来，假如程序运行正常，则总会在看门狗定时器到点前“喂狗”，从而避免 Launchpad 复位。
3. 如果程序死机，则不会及时“喂狗”，Launchpad 复位。复位后看门狗依然默认开启，继续守护着程序的正常运行。

在实际的 MCU 应用系统中，由于常常会受到来自外界的某些干扰，有可能（对规范的设计概率极小）造成程序的跑飞而陷入死循环，从而导致整个系统的陷入停滞状态并且不会自动恢复到可控的工作状态。


看门狗电路所起的作用是一旦 MCU 运行出现故障，就强制对 MCU 进行硬件复位，使整个系统重新处于可控状态。

4.4 实验步骤和流程图


实验步骤如下：

- (1) 将 PC 和 Tiva 板通过 USB 线相连；
- (2) 打开 MDK 集成开发工具，选择 Project->Rebuild all target files 对工程进行编译或直接点击  编译程序；

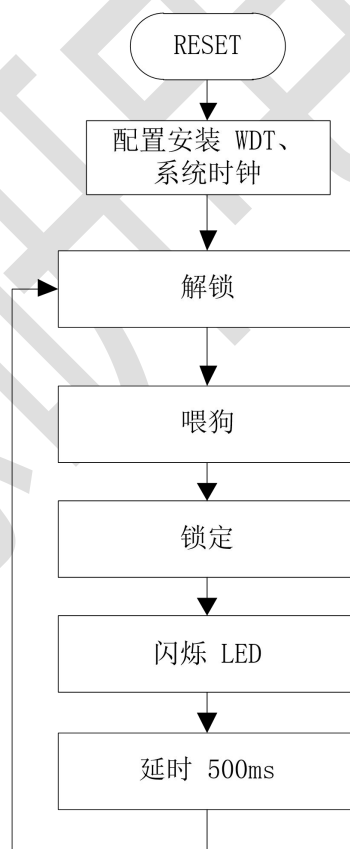
- (3) 编译后，点击 ，将程序下载到实验板中。按下实验板上的复位键即可看到实验现象。

或者，选择 Debug->Start/stop debug session，或直接点击  在线仿真试验程序。点击

后，会变为调试界面。 ，从左到右分别是单步进入函数调试、单步越过

执行下一条语句、单步越过执行下一条语句、执行到光标位置。再次点击  可退出调试，界面恢复正常。

其实验流程图如下：



实验流程图

4.5 关键代码分析

```
// 看门狗初始化
```

```
void wdogInit(void)
{
    uint32_t ulValue = 350 * (TheSysClock / 1000);           // 准备定时
350ms
    SysCtlPeriEnable(SYSCTL_PERIPH_WDOG);                    // 使能看门狗模块
    WatchdogResetEnable(WATCHDOG_BASE);                      // 使能看门狗复位功能
    WatchdogStallEnable(WATCHDOG_BASE);                      // 使能调试器暂停看门
                                                    狗计数
    WatchdogReloadSet(WATCHDOG_BASE, ulValue);              // 设置看门狗装载值
    WatchdogEnable(WATCHDOG_BASE);                          // 使能看门狗
    WatchdogLock(WATCHDOG_BASE);                            // 锁定看门狗
}
// 喂狗操作
void wdogFeed(void)
{
    WatchdogUnlock(WATCHDOG_BASE);                          // 解除锁定
    WatchdogIntClear(WATCHDOG_BASE);                        // 清除中断状态，即喂狗操作
    WatchdogLock(WATCHDOG_BASE);                            // 重新锁定
    GPIOPinWrite(LED_PORT, LED_PIN, 0x00);                 // 点亮 LED
    SysCtlDelay(2 * (TheSysClock / 3000));                  // 短暂延时
    GPIOPinWrite(LED_PORT, LED_PIN, 0xFF);                  // 熄灭 LED
}
```

第 5 章 PWM 实验

5.1 实验介绍

脉冲宽度调制（PWM）是一种功能强大的对模拟信号电平进行数字编码的技术。使用高分辨率计数器产生一个方波，方波的占空比被调制成一个模拟信号的编码。典型应用包括开关电源和电机控制。

TM4C123GH6PM 微控制器包含两个 PWM 模块，每个模块有四个 PWM 发生器模块和一个控制模块组成，一共可以产生 16 个 PWM 输出。控制模块决定了 PWM 信号的极性，以及能够通过管脚的信号。

每个 PWM 发生器模块产生两个 PWM 信号，这两个信号拥有相同的计时器和频率，我们可以编程对它们今次那个单独操作，也可以把它们作为能够插入死区延迟的一对互补信号。PWM 发生器模块产生的输出信号称为 PWMA 和 PWMB，它们在被送到设备管脚之前由输出控制模块管理，之后作为 MnPWM0 和 MnPWM1，或者是 MnPWM2 和 MnPWM3 信号使用。

本节实验利用 PWM 模块产生产生固定占空比的方波。

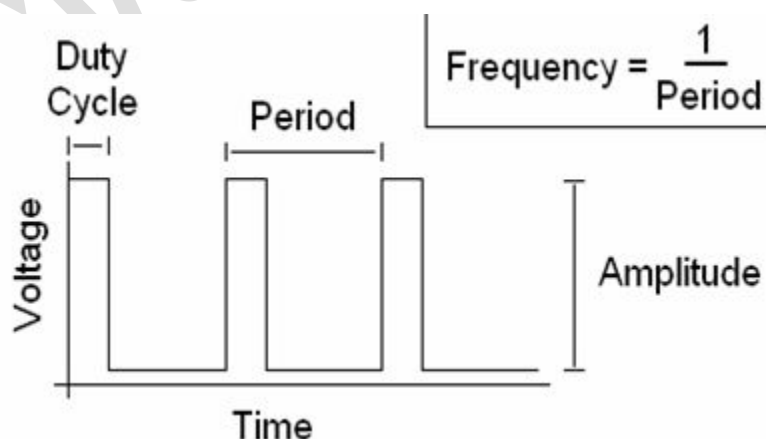
5.2 实验目的

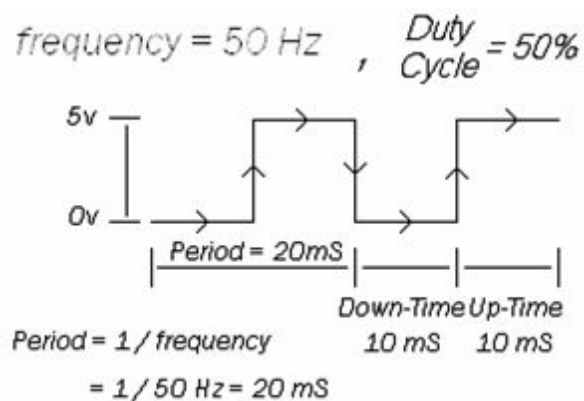
- (1). 了解 PWM 技术；
- (2). 掌握 PWM 控制技术的原理；
- (3). 掌握 LaunchPad 产生 PWM 输出的方法；
- (4). 掌握示波器测量频率的方法。

5.3 实验原理



PWM 技术的三个要素：

- (1) Frequency 时钟频率
- (2) Duty cycle 占空比
- (3) Amplitude 信号幅度





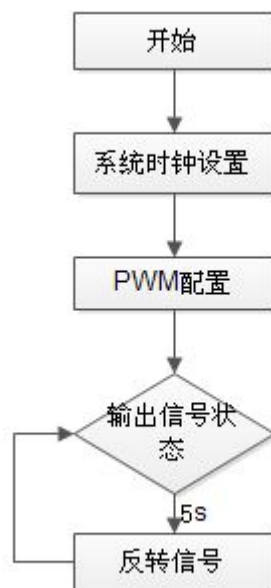
5.4 实验步骤和流程图

- (1). 将 LaunchPad 通过 USB 线连接 PC
- (2). 打开 Keil 集成开发工具，打开 PWM 的示例 project。
- (3). 选择  对工程进行编译链接，成功后点击  将程序烧写入开发板，按下开发板的复位键，运行程序。

初始化流程：

- (1). 系统时钟、PWM 时钟设置
- (2). 配置 UART 用于显示信号状态
- (3). 配置 PWM 参数（周期、占空比等）
- (4). 输出信号状态，延迟 5s
- (5). 反转信号，重复（4）

实验流程图如下图所示：



实验流程图

5.5 实验现象

如果利用示波器测试相应管脚，调整好示波器，可以看到固定占空比的方波并且每 5s 反转一次。

本实验考虑到众多同学无示波器情况，代码中加入了利用了 UART 间接显示实验结果的部分，也可以看到串口输出方波的状态每 5s 变换一次。

5.6 关键代码分析

```
//配置 PWM0 输出频率为 250Hz，占空比为 25%。每隔 5s 信号极性反转一次。
//*****UARTORX - PA0
//*****UARTOTX - PA1
//*****PWM0 - PB6
//InitConsole() 配置 UART 参数用于实验信息输出
void InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_UORX);
    GPIOPinConfigure(GPIO_PA1_UOTX);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTStdioConfig(0, 115200, 16000000);
}
```

```
//main 函数
int main(void)
{    //设置系统时钟
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                    SYSCTL_XTAL_16MHZ);

    //设置 PWM 时钟为系统时钟的 1 分频
    SysCtlPWMClockSet(SYSCTL_PWMDIV_1);
    //配置串口用于显示程序状态信息
    InitConsole();
    //显示相关配置信息
    UARTprintf("PWM ->\n");
    UARTprintf("  Module: PWM0\n");
    UARTprintf("  Pin: PD0\n");
    UARTprintf("  Configured Duty Cycle: 25%\n");
    UARTprintf("  Inverted Duty Cycle: 75%\n");
    UARTprintf("  Features: PWM output inversion every 5 seconds.\n\n");
    UARTprintf("Generating PWM on PWM0 (PD0) -> State = ");
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    GPIOPinConfigure(GPIO_PB6_MOPWM0);
    //配置引脚为 PWM 功能
    GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_6);
    //配置 PWM 发生器
    PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_UP_DOWN |
                    PWM_GEN_MODE_NO_SYNC);

    //配置 PWM 周期
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 64000);
    //配置 PWM 占空比
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0,
                      PWMGenPeriodGet(PWM0_BASE, PWM_OUT_0) / 4);

    //使能 PWM0 输出
    PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true);
    //使能 PWM 发生器模块
    PWMGenEnable(PWM0_BASE, PWM_GEN_0);
    while(1)
    {    //打印 PWM 输出的极性
        UARTprintf("Normal  \b\b\b\b\b\b\b\b");

        //延迟 5s
        SysCtlDelay((SysCtlClockGet() * 5) / 3);
        //信号反转
        PWMOutputInvert(PWM0_BASE, PWM_OUT_0_BIT, true);
        UARTprintf("Inverted\b\b\b\b\b\b\b\b");
        SysCtlDelay((SysCtlClockGet() * 5) / 3);
        //恢复原来状态，信号不反转
```

```
        PWMOutputInvert(PWM0_BASE, PWM_OUT_0_BIT, false);  
    }  
}
```

德研电子

第 6 章 UART 发送与接收实验

6.1 实验介绍

RS232 接口标准是在串口通信中最基础、最简单的通信协议。本实验将利用 LaunchPad 的 UART 模块与 PC 进行 RS232 数据通信，先使用 PC 向 LaunchPad 的 UART 发送数据，而后 LaunchPad 再将数据回传给 PC 端，程序十分简单。通过该实验，希望大家可以掌握串口通信的原理，并熟悉 Tiva 开发板的 UART 初始化和使用。

6.2 实验目的

- (1). 熟悉串口通信接口标准概况；
- (2). 熟悉 LaunchPad 的 UART 模块的使用；
- (3). 实现 MCU 与 PC 之间的串口通信。

6.3 实验原理

RS232 简介

RS232 接口是1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标准。它的全名是“数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准”。

在RS-232 的通讯标准中是以一个25 针的接口来定义的，并在早期的计算机如PC 或 XT 机型上广泛使用，但在AT 机以后的机型上，实际均采用了9 针的简化版本应用，现在所说的232 通讯均默认为9 针的接口。以下是各管脚的说明：

旧制JIS名称	新制JIS名称	全称	说明
FG	SG	Frame Ground	连到机器的接地线
TXD	SD	Transmitted Data	数据输出线
RXD	RD	Received Data	数据输入线
RTS	RS	Request to Send	要求发送数据
CTS	CS	Clear to Send	回应对方发送的RTS 的发送许可，告诉对方可以发送
DSR	DR	Data Set Ready	告知本机在待命状态
DTR	ER	Data Terminal Ready	告知数据终端处于待命状态
CD	CD	Carrier Detect	载波检出，用以确认是否收到Modem的载波
SG	SG	Signal Ground	信号线的接地线（严

			格的说是信号线的零标 准线)
--	--	--	-------------------

实际应用中，电子工程师在设计计算机与外围设备的通信时，通常在9 针的基础再进行简化，只用其中的2、3、5 三个管脚进行通信。这三个管脚分别是接收线、发送线和地线，在一般情况下即可满足通讯的要求。

RS232 区别于一般协议的地方，在于它仅仅规定了通信的硬件规范，而软件上的通信协议完全由使用者确定。LaunchPad自带的UART模块即可非常好的完成这一点，即通用异步接收/发送。



异步通讯特性概述：

1. 7 位或 8 位数据位，支持奇偶校验
2. 独立的发送和接收移位寄存器
3. 独立的发送和接收缓存
4. 可选择先发送（接收）MSB 还是LSB
5. 空闲位多机模式和地址位多机模式
6. 通过有效的起始位检测将LaunchPad从低功耗唤醒
7. 状态标志检测错误或者地址位
8. 独立的接收和发送中断
9. 可编程实现波特率的调整

6.4 实验步骤和流程图

(1). 将 LaunchPad 通过 USB 线连接 PC

(2). 打开 Keil 集成开发工具，打开 UART 的示例 project。

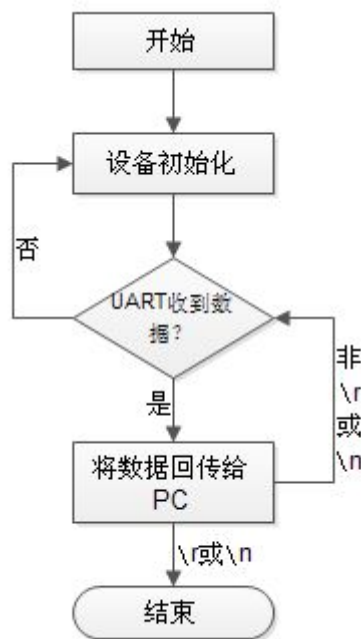
(3). 选择  对工程进行编译链接，成功后点击  将程序烧写入开发板，按下开发板的复位键，运行程序。

注意：实验过程中必须将串口的 RX 和 TX 连接形成回路！！

初始化流程：

- (1). 设备时钟设置
- (2). UART 外设使能，GPIO 外设使能
- (3). GPIO 端口模式设置
- (4). 串口参数初始化
- (5). 接收发送字符

实验流程图如下图所示：



实验流程图

6.5 实验现象

使用串口通信助手发送字符，接收窗口能够得到同样的字符反馈。

6.6 关键代码分析

```
// 配置 UART 和 PC 进行读写通信.
//*****UARTORX  - PA0
//*****UARTOTX  - PA1
//*****
int main(void)
{
    char  cThisChar;
    //配置设备时钟为 16MHz，时钟源为外部晶振
        SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                        SYSCTL_XTAL_16MHZ);
    //外设使能
        SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    //GPIO 引脚配置
        GPIOPinConfigure(GPIO_PA0_UORX);
        GPIOPinConfigure(GPIO_PA1_UOTX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    //配置 UART  参数
        UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
```

```
(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |  
    UART_CONFIG_PAR_NONE));  
//发送 ‘!’ 字符表示设备初始化成功，等待接收字符  
UARTCharPut(UART0_BASE, '!');  
do{  
    cThisChar = UARTCharGet(UART0_BASE);  
    //将接收到的字符发送；判断是否为 ‘\r’ 或者 ‘\n’，是则结束循环，否则进入下一个循  
环  
    UARTCharPut(UART0_BASE, cThisChar);  
}while((cThisChar != '\n') && (cThisChar != '\r'));  
    return(0);  
}
```

第 7 章 SPI 模式读写 SD 实验

7.1 实验介绍

本实验演示了如何调用文件系统提供的接口函数来读写 SD 卡。实验是基于 FAT 文件系统，文件系统提供了操作 SD 卡的接口函数。

7.2 实验目的

- (1) 了解如何用 SPI 模式读取 SD 卡；
- (2) 了解读写 SD 卡的过程；
- (3) 学会使用文件系统的基本函数。

7.3 实验原理

SD 卡实质上是一片 FLASH 存储区，如何有效读写，需要首先在 SD 卡上加载一个文件系统，目前用得较多的文件系统有 FAT、NTFS 等，本实验采用较为简单的 FAT 文件系统。这个文件系统是开源的，方便读者更好的理解文件系统机制。对 SD 卡的读写，实质上是调用文件系统的各种函数。本实验演示了从创建文件、向文件写数据到读出文件数据，关闭文件等若干步骤，较为全面地向读者介绍了读写 SD 卡的方式。

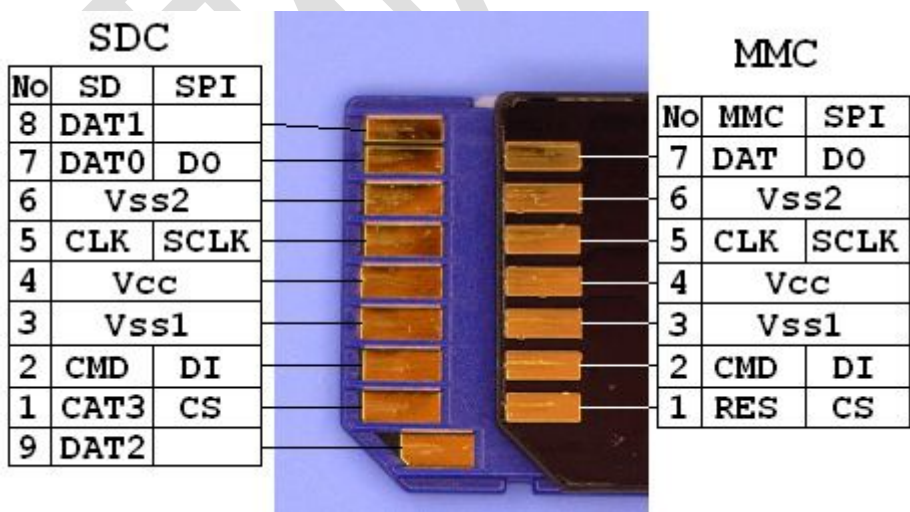


图 SD 卡引脚概览

SD 卡的 SPI 通信接口使其可以通过 SPI 通道进行数据读写。从应用的角度来看，采用 SPI 接口的好处在于，很多单片机内部自带 SPI 控制器，不光给开发上带来方便，同时也见降低了开发成本。然而，它也有不好的地方，如失去了 SD 卡的性能优势，要解决这

一问题，就要用 SD 方式，因为它提供更大的总线数据带宽。SPI 接口的选用是在上电初始时向其写入第一个命令时进行的。以下介绍 SD 卡的驱动方法，只实现简单的扇区读写。

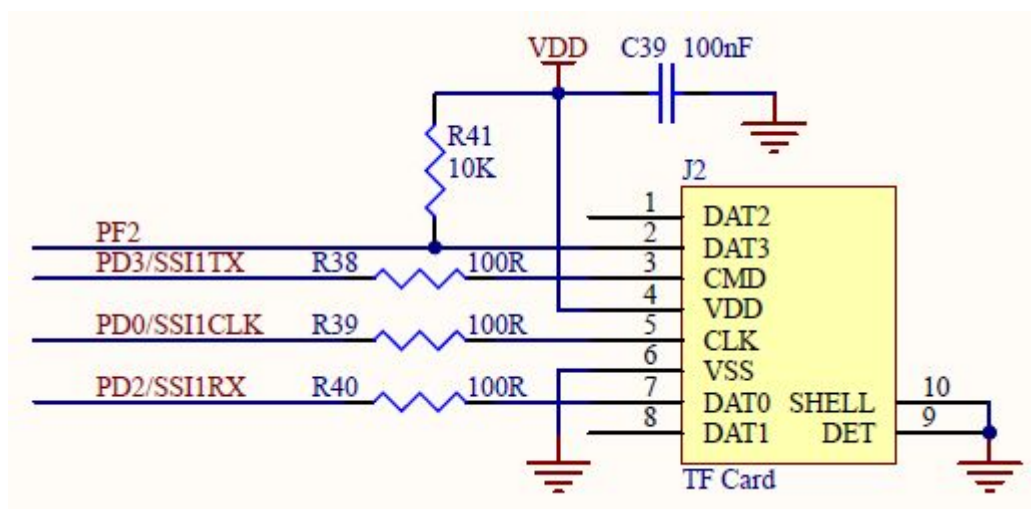


图 Tiva 开发板 SD 卡引脚连接示意图

如图为实验板的 SD 卡模块电路图。TM4C123G 通过 SSI 接口控制 SD 卡的读写。SPI 模式下，读取速度比 SDIO 慢，但是仍能达到数 MB 的速度。

实验用的 SD 卡使用了 FAT32 的文件系统。FAT（FILE ALLOCATION TABLE）是“文件分配表”的意思，是微软 早支持的分区格式，依据 FAT 表中每个簇链的所占位数（有关概念，后面会讲到）分为 fat12、fat16、fat32 三种格式，但其基本存储方式相似，现仍广泛使用。FAT 是用来记录文件所在位置的表格，它对于硬盘的使用是非常重要的，假如丢失文件分配表，那么硬盘上的数据就会因无法定位而不能使用了。SD 卡是现代手持设备主流外部存储设备，是一种基于半导体快闪记忆器的新一代记忆设备。文件是一个二进制、ASCII 或者任何其他格式的流，能够永久存储在 FLASH 设备上并有请求时随时可被恢复。文件可以被存在大容量的 FLASH 设备或者一个 SD 存储卡上。

本实验的 FATFS 库提供了标准高层文件接口函数，它是一个强大的文件系统。提供打开、创建、读写文件等，与 PC 编程对文件的各种操作类似，读者可以参照对比。你可以从标准的 SD 卡和 MMC 卡中读写文件（用 SPI 模式）。本实验采用 SPI 接口连接 SD 卡。

使用 SPI 操作的速度不能太快，在初始化时时钟设为 400k 以下为宜。

下面简要介绍一下本实验中用到的几个主要 API 函数：

- 文件打开函数：
`u8 open_file(FIL *fp, const TCHAR *path, BYTE mode)`

该函数用于在文件系统中的打开一个文件，文件名通过参数 `FIL *fp` 来指定，如果打开文件成功，返回值为 `FRESULT` 枚举类型，如果是 `FR_OK = 0`，则文件打开成功，否则文件打开错误，在错误打开的情况下，建议先使用 `close_file` 关闭一个打开的文件后再打开文件。

- 文件关闭函数：

`FRESULT f_close (FIL *fp)`

该函数调用关闭一个已打开的文件，并释放当前文件占用的文件资源。

- 读文件函数：

`FRESULT f_read (FIL *fp, void *buff, UINT btr, UINT *br)`

该函数完成已打开文件的当前位置读取指定的长度数据，读取的数据存放在缓冲区 `buff` 中，而读取数据的长度是由参数 `UINT btr` 来指定，缓冲区的首地址由参数 `void *buff` 来指定。

- 文件读写位置设定函数：

`FRESULT f_lseek (FIL *fp, DWORD ofs)`

该函数用于文件读写当前位置的设定，`ofs` 是相对文件起始位置偏移量。

- 写文件函数：

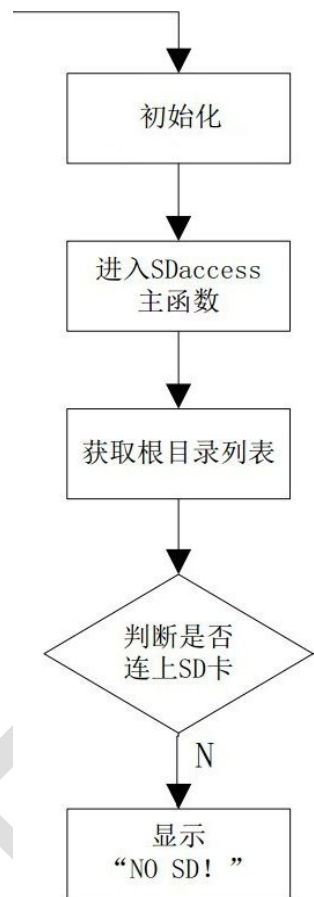
`FRESULT f_write (FIL *fp, const void *buff, UINT btw, UINT *bw)`

该函数实现将缓冲区 `buff` 中指定长度的数据写入一个已打开文件的当前位置，写数据长度由参数 `UINT btr` 指定，缓冲区首地址由参数 `const void *buff` 指定。

7.4 实验步骤

- (1) 将 PC 和板载仿真器通过 USB 线相连；
- (2) 打开 Keil 项目工程文件
- (3) 编译项目工程，并烧写到 Tiva 开发板中
- (4) 打开串口工具，配置波特率为 115200，观察输出结果

7.5 实验代码流程图



实验流程图

7.6 实验现象

本实验板提供直接访问 SD 卡的功能，并输出至 UART 口，以文本的形式显示出来

```
int main(void)
{
    //初始化 SysTick
    RCC_Config();
    //初始化串口
    USART1_Config();
    //初始化 SPI1
    SPI1_Config();
    printf("start to read file\n");
    /* Register volume work area (never fails) */
    f_mount(0, &fatfs);
```



```

    printf("\nOpen a test file (test.txt).\n");
    rc = f_open(&fil, "test.txt", FA_READ);
    if (rc) die(rc);

    printf("\nType the file content.\n");
    for (;;) {
        rc = f_read(&fil, buff, sizeof(buff), &br);    /* 读取文件 */
        if (rc || !br) break;                          /* Error
or end of file */
        for (i = 0; i < br; i++)
            putchar(buff[i]); //逐个输出
    }
    if (rc) die(rc);

    printf("\nClose the file.\n");
    rc = f_close(&fil);
    if (rc) die(rc);

    printf("\nCreate a new file (hello.txt).\n");
    rc = f_open(&fil, "HELLO.TXT", FA_WRITE | FA_CREATE_ALWAYS);
    if (rc) die(rc);

    printf("\nWrite a text data. (Hello world!)\n");
    rc = f_write(&fil, "Hello world!\r\n", 14, &bw);
    if (rc) die(rc);
    printf("%u bytes written.\n", bw);

    printf("\nClose the file.\n");
    rc = f_close(&fil);
    if (rc) die(rc);

    while (1);
}

```

7.7 思考题

- (1) 文件系统的作用是什么？使用文件系统有什么好处？
- (2) 将 PC 和板载仿真器通过 USB 线相连 SPI 接口的 SD 卡，读写速度可以达到多少？做系统开发时，这一点是否要作为考虑因素？
- (3) 使用文件系统的好处是什么？

第 8 章 LM75A 温度采集实验

8.1 实验介绍

利用 TM4C123G，使用 I2C 总线对 LM75A 温度寄存器进行读取操作，通过 LCD 显示当前环境温度数据。

LM75A 是一个高速 I2C 接口的温度传感器，可以在 $-55^{\circ}\text{C}\sim+125^{\circ}\text{C}$ 的温度范围内将温度直接转换为数字信号，并可实现 0.125°C 的精度。MCU 可以通过 I2C 总线直接读取其内部寄存器中的数据，并可通过 I2C 对 4 个数据寄存器进行操作，以设置成不同的工作模式。LM75A 有 3 个可选的逻辑地址管脚，使得同一总线上可同时连接 8 个器件而不发生地址冲突。

LM75A 可配置成不同的工作模式。它可设置成在正常工作模式下周期性地对环境温度进行监控，或进入关断模式来将器件功耗降至最低。OS 输出有 2 种可选的工作模式：OS 比较器模式和 OS 中断模式，OS 输出可选择高电平或低电平有效。

正常工作模式下，当器件上电时，OS 工作在比较器模式，温度阈值为 80°C ，滞后阈值为 75°C 。

LM75A 管脚描述

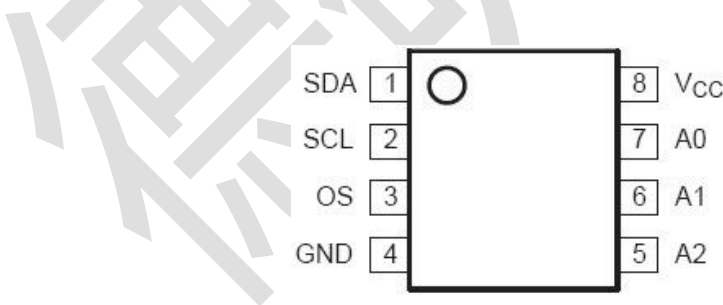


图 LM75A 管脚描述

- SDA: I²C 串行双向数据线，开漏口。
- SCL: I²C 串行时钟输入，开漏口。
- OS: 过热关断输出。开漏输出。
- GND: 地，连接到系统地。
- A2: 用户定义的地址位2。

- A1: 用户定义的地址位1。
- A0: 用户定义的地址位0。
- V_{CC}: 电源。

在主控器的控制下，LM75A 可以通过 SCL 和 SDA 作为从器件连接到 I2C 总线上。主控器 必须提供 SCL 时钟信号，可以通过 SDA 读出器件数据或将数据写入到器件中。注意：必须在 SCL 和 SDA 端分别连接一个外部上拉电阻，阻值大约为 10k Ω 。

LM75A 从地址（7 位地址）的低 3 位可由地址引脚 A2、A1 和 A0 的逻辑电平来决定。地址的高 4 位预先设置为‘1001’。由于输入管脚 SCL、SDA、A2-A0 内部无偏置，因此在任何应用中它们都不能悬空。

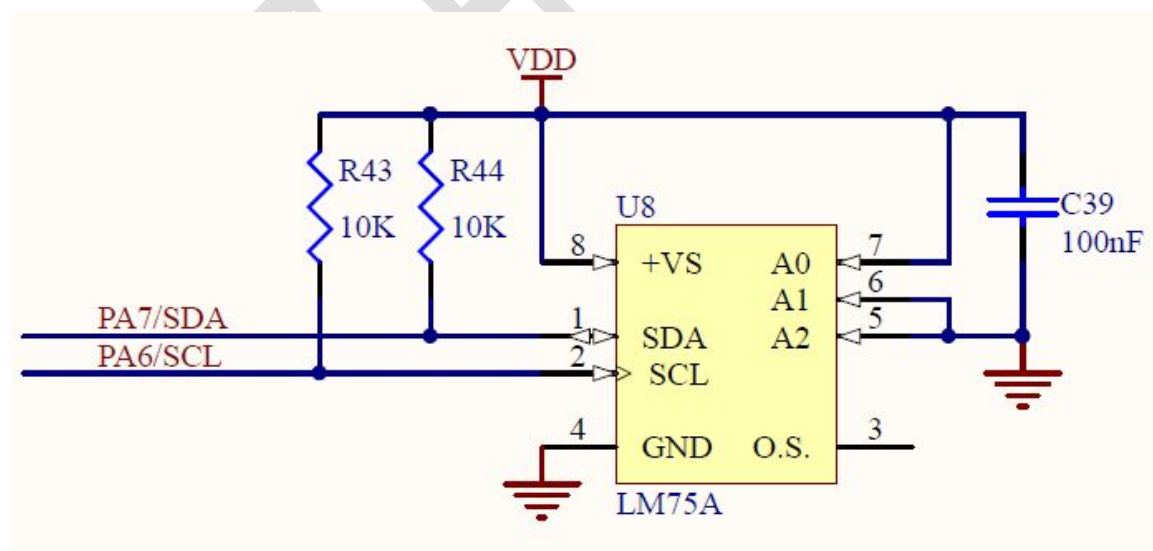
8.2 实验目的

- (1). 熟悉 I2C 外设的使用，以及相关库函数的使用；
- (2). 了解 LM75A 温度传感器的特性，以及使用方法；

8.3 实验原理

实验硬件介绍

LM75A 工作电压为 3.0V~5.5V，采用 5.0V 直接供电。








温度测量电路

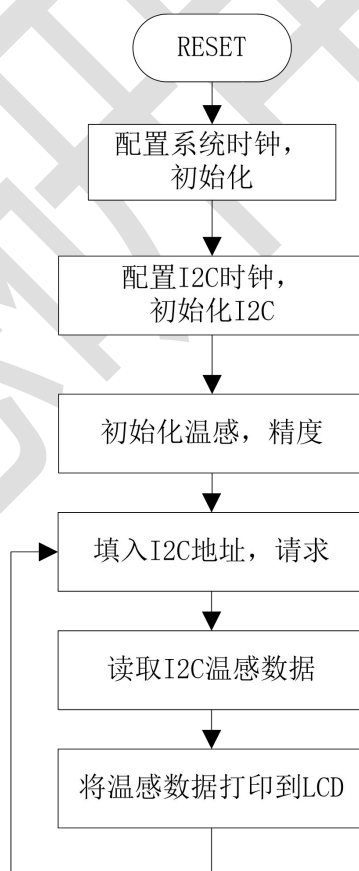
设计中只采用了一个 LM75A， 所以 I2C 的地址线 A1-A2 接地即可。

8.4 实验步骤及流程图

实验步骤如下：

- (1) 将 PC 和 Tiva 板通过 USB 线相连；
- (2) 打开 MDK 集成开发工具，选择 Project->Rebuild all target files 对工程进行编译或直接点击  编译程序；
- (3) 编译后，点击 ，将程序下载到实验板中。按下实验板上的复位键即可看到实验现象。

或者，选择 Debug->Start/stop debug session，或直接点击  在线仿真试验程序。点击后，会变为调试界面。 ，从左到右分别是单步进入函数调试、单步越过执行下一条语句、单步越过执行下一条语句、执行到光标位置。再次点击  可退出调试，界面恢复正常。



实验流程图

8.5 实验现象

可通过 LCD 屏幕实时输出当前摄氏温度

8.6 关键代码分析

本程序主要实现的功能：使用 I2C 总线方式读温度寄存器的值。

```
void TMP100Init(void)
{
#ifdef DY_I2C_SPEED_FIX
    _I2CMasterInitExpClk(TMP_PIN_I2C_PORT, SysCtlClockGet(), false);
#else
    I2CMasterInitExpClk(TMP_PIN_I2C_PORT, SysCtlClockGet(), false);
#endif
}

uint16_t TMP100DataRead(void)
{
    uint8_t i2cWriteBuffer;
    uint32_t i2cReadBuffer[2];

    uint16_t temp_value;

    // send to tmp100:
    i2cWriteBuffer = TMP_TEMP_REG;
    i2cWriteBuffer &= 0x03;          // 2 bit valid

    // frame 1:
    I2CMasterSlaveAddrSet(TMP_PIN_I2C_PORT, TMP_I2C_ADDR, false);

    // frame 2:
    I2CMasterDataPut(TMP_PIN_I2C_PORT, i2cWriteBuffer);
    I2CMasterControl(TMP_PIN_I2C_PORT, I2C_MASTER_CMD_SINGLE_SEND);
    // wait finish
    while(I2CMasterBusBusy(TMP_PIN_I2C_PORT))
        ;

    // read from tmp100:
    // frame 3:
    I2CMasterSlaveAddrSet(TMP_PIN_I2C_PORT, TMP_I2C_ADDR, true);
```

```
//I2CMasterControl(TMP_PIN_I2C_PORT, I2C_MASTER_CMD_SINGLE_RECEIVE);

I2CMasterControl(TMP_PIN_I2C_PORT, I2C_MASTER_CMD_BURST_RECEIVE_START);

// get temperature int value
// frame 4:
i2cReadBuffer[0] = I2CMasterDataGet(TMP_PIN_I2C_PORT);

int a = 500;
while (a--)
    ;

I2CMasterControl(TMP_PIN_I2C_PORT, I2C_MASTER_CMD_BURST_RECEIVE_CONT);

// get the decimal value
// frame 5:
i2cReadBuffer[1] = I2CMasterDataGet(TMP_PIN_I2C_PORT);
// while(I2CMasterBusBusy(TMP_PIN_I2C_PORT))
//     ;

I2CMasterControl(TMP_PIN_I2C_PORT, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);

temp_value = i2cReadBuffer[0] | (i2cReadBuffer[1] << 8);

return temp_value;
}
```

第 9 章 电位器输入实验

9.1 实验介绍

在 Tiva C 系列芯片的实时控制和智能仪表等应用系统中，经常控制或测量对象的有关变量，而这些变量往往是一些连续变化的模拟量，如温度、压力、流量、速度等物理量。利用传感器把各种物理信号测量出来，转换为电信号，经过模数转换（ADC）变成数字量，这样才能被 Tiva C 系列芯片处理和控制。本实验通过 ADC 将电位器的变化实时显示在 LCD 上。

9.2 实验目的

- (1) 掌握 ADC 转换的实验原理。
- (2) 熟练应用 TM4C123GH6PM 的 ADC0、ADC1 模块。

9.3 实验原理

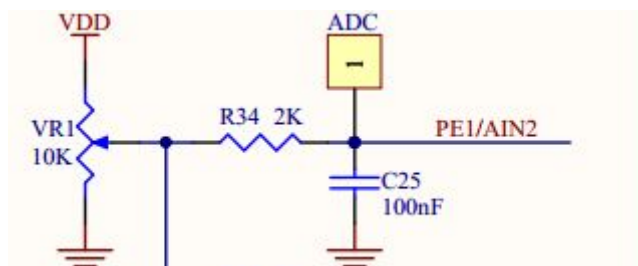
模数转换器（ADC）是用于将连续的模拟电压转换为离散的数字电压的外设。包括两个相同的转换器模块，共享 12 个输入通道。

TM4C123GH6PM ADC 模块具有 12 位转换精度并支持 12 个输入通道，还有一个内部温度传感器。每个 ADC 模块包含四个可编程序列发生器，无需使用控制器情况下，允许多个模拟输入源的采样。每个采样序列发生器提供灵活的编程，有完全可配置的输入源，触发事件，中断的产生，序列发生器的优先级。

此外，转换后的值可以任意地转移到一个数字比较器模块。每个 ADC 模块提供 8 个数字比较器。每个数字的比较评估 ADC 转换的值对两个用户定义的值来确定信号的工作范围。ADC0 和 ADC1 的触发源可以是独立的，或者两个 ADC 模块可工作在相同的触发源和操作相同或不同的输入。移相器可以通过指定的相位角延迟启动采样。当同时使用两个 ADC 模块时，将会配置转换器同时开启转换或附带彼此相对应的相位。

TM4C123GH6PM 微控制器包含两个相同的模数转换器模块。这两个模块，ADC0 和 ADC1，共享 12 个相同的模拟输入通道。每个 ADC 模块独立运作，因此，可以执行不同的样本序列，在任何时间的任何模拟输入通道采样，并产生不同的中断和触发器。






ADC 结构图如下图所示：



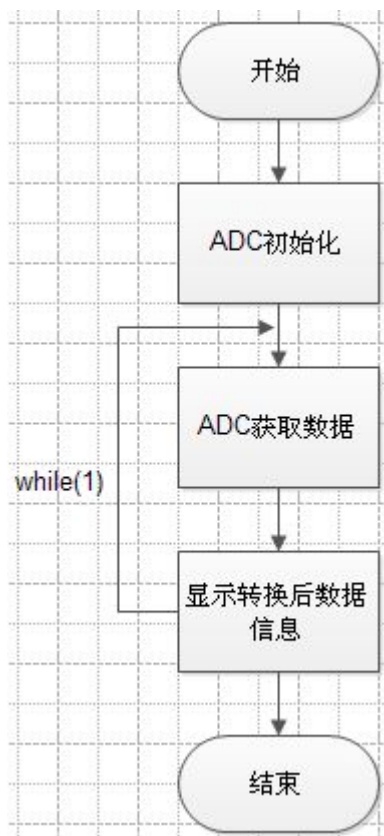
ADC 原理图

9.4 实验步骤及流程图

实验步骤如下：

- (1) 将 PC 和 Tiva 板通过 USB 线相连；
- (2) 打开 MDK 集成开发工具，选择 Project->Rebuild all target files 对工程进行编译或直接点击  编译程序；
- (3) 编译后，点击 ，将程序下载到实验板中。按下实验板上的复位键即可看到实验现象。
或者，选择 Debug->Start/stop debug session，或直接点击  在线仿真试验程序。点击后，会变为调试界面。 ，从左到右分别是单步进入函数调试、单步越过执行下一条语句、单步越过执行下一条语句、执行到光标位置。再次点击  可退出调试，界面恢复正常。
- (4) 本实验选用电位器电压输入到单片机引脚 PE1；故，当旋转扩展板上的 RP1 电位器便可看到 ADC 采集并转换后的信号变化。

实验流程图如下图所示：



实验流程图

9.5 实验现象

调节电位器，观察实验板上的 LCD 显示，随着电位器的转动，LCD 显示输入模拟电压的转换结果 NADC，其范围是 0 至 4095。输入模拟电压的转换结果满足公式： $NADC=4095 \times D/VDD$ 。本实验推荐使用万用表，观察电位计中间引脚对地电压。

9.6 关键代码分析

```
//ADC 初始化程序
void ADC_Init()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_1);
    GPIOPinTypeGPIOInput(GPIO_PORTE_BASE, GPIO_PIN_1);
    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_ALWAYS,
ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH2 | ADC_CTL_IE |
                        ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 3);
}
//读取 ADC 转换后的值
```

```
uint32_t ADC_read()
{
    while(!ADCIntStatus(ADCO_BASE, 3, false))
    {
    }
    ADCIntClear(ADCO_BASE, 3);
    ADCSequenceDataGet(ADCO_BASE, 3, pui32ADC0Value);
    SysCtlDelay(SysCtlClockGet() / 12);
    return pui32ADC0Value[0];
}

//ADC 测试实验主程序
void ADC_TEST()
{
    ADC_Init();
    while(1)
    {
        num0 = ADC_read();
        My_printf(2, 0, "ADC0=%d    ", num0);
        if(a != 3)
            break;
    }
    LCD_Clear();
}
```

第 10 章 AC 模拟比较器

10.1 实验介绍

本实验演示了应用 Tiva C 系列芯片 AC 模块进行电压采集，并与已设定的比较电压进行比较。

10.2 实验目的

- (1) 掌握 AC 转换的实验原理。
- (2) 熟练应用 TM4C123GH6PM 的 AC 模块，

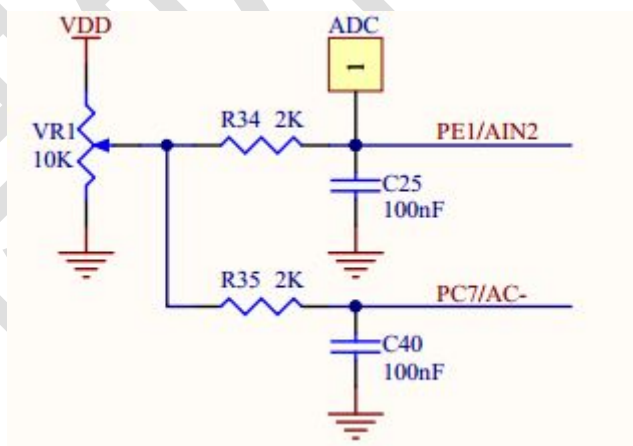
10.3 实验原理

模拟比较器是一个外设，它能比较两个模拟电压，并提供一个逻辑输出来表示比较结果。比较器为它的输出提供一个设备引脚，作为开发板的模拟比较器的替代品。此外，比较器可通过中断或触发 ADC 中的样本序列的起始而表示一个应用程序。中断的产生和 ADC 触发逻辑是分开并独立的。这种灵活性意味着，例如，可以在上升沿产生中断和在下降沿 ADC 触发。

TM4C123GH6PM 微控制器提供两个独立的集成模拟比较器，具有以下功能：

- 将外部引脚输入或内部可编程电压参数与外部引脚输入相比较。
- 将测试电压与以下任一电压作比较：
 - 一个独立的外部电压参数
 - 一个共享的外部电压参数
 - 一个共享内部电压参数

AC 结构图如下图所示：




AC 原理图


10.4 实验步骤及流程图

实验步骤如下：


- (1) 将 PC 和 Tiva 板通过 USB 线相连；
- (2) 打开 MDK 集成开发工具，选择 Project->Rebuild all target files 对工程进行编译或直接点

击  编译程序；

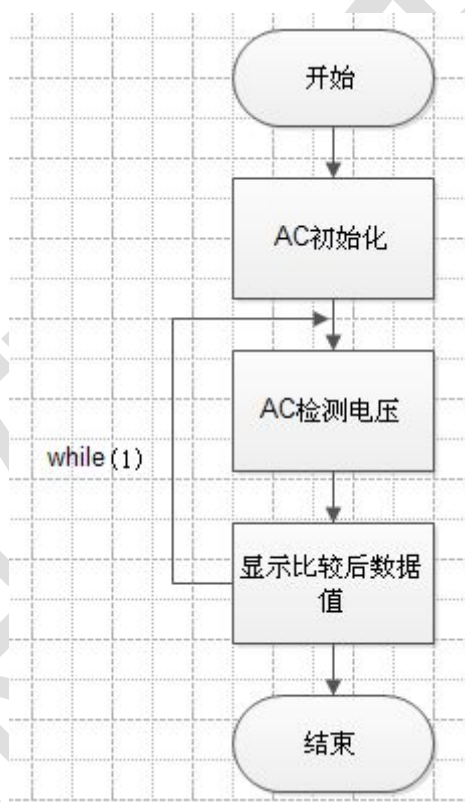
(3) 编译后，点击 ，将程序下载到实验板中。按下实验板上的复位键即可看到实验现象。

或者，选择 Debug->Start/stop debug session，或直接点击  在线仿真试验程序。点击

后，会变为调试界面。 ，从左到右分别是单步进入函数调试、单步越过

执行下一条语句、单步越过执行下一条语句、执行到光标位置。再次点击  可退出调试，界面恢复正常。

(4) 本实验选用电位器电压输入到单片机引脚 PC7；故，当旋转扩展板上的 RP1 电位器，当该电位器达到某一值后，便可看到 AC 值的变化。



实验流程图

10.5 实验现象

调节电位器，观察实验板上的 LCD 显示，随着电位器的转动，当电位器的电压上升超过或下降低于某一值时，AC 比较的值将由 0 变为 1 或相反。

10.6 关键代码分析

```
//AC 初始化程序  
void AC_Init()
```

```
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_COMP0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    GPIOPinTypeComparator(GPIO_PORTC_BASE, GPIO_PIN_7);
    GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_7);
    ComparatorRefSet(COMP_BASE, COMP_REF_1_65V);
    ComparatorConfigure(COMP_BASE, 0, (COMP_TRIG_NONE | COMP_INT_BOTH |
COMP_ASRCP_REF | COMP_OUTPUT_NORMAL));
    delay(200000);
}

//读取 AC 转换后的值
void AC_detect()
{
    bool temp;
    temp = ComparatorValueGet(COMP_BASE, 0);
    if(temp == false)
        num1 = 0;
    else
        num1 = 1;
}

//AC 测试实验主程序
void AC_TEST()
{
    while(1)
    {
        AC_detect();
        My_printf(4, 0, "AC=%d", num1);
    }

    LCD_Clear();
}
```

第 11 章 DAC 实验—扬声器播放

11.1 实验介绍

本节演示了 Tiva 实验板的 DAC 数模转换示例,通过 TM4C123G 控制 DAC 模块参数,使其产生输出一定规则的电压波形(如三角波,正弦波等),可以使用示波器验证波形。在掌握 DAC 工作原理和硬件发声原理的基础上,采用定时器调制正弦波,在程序中规定好调制频率,让 DAC 模块产生的电压驱动扬声器发出不同频率的声音。

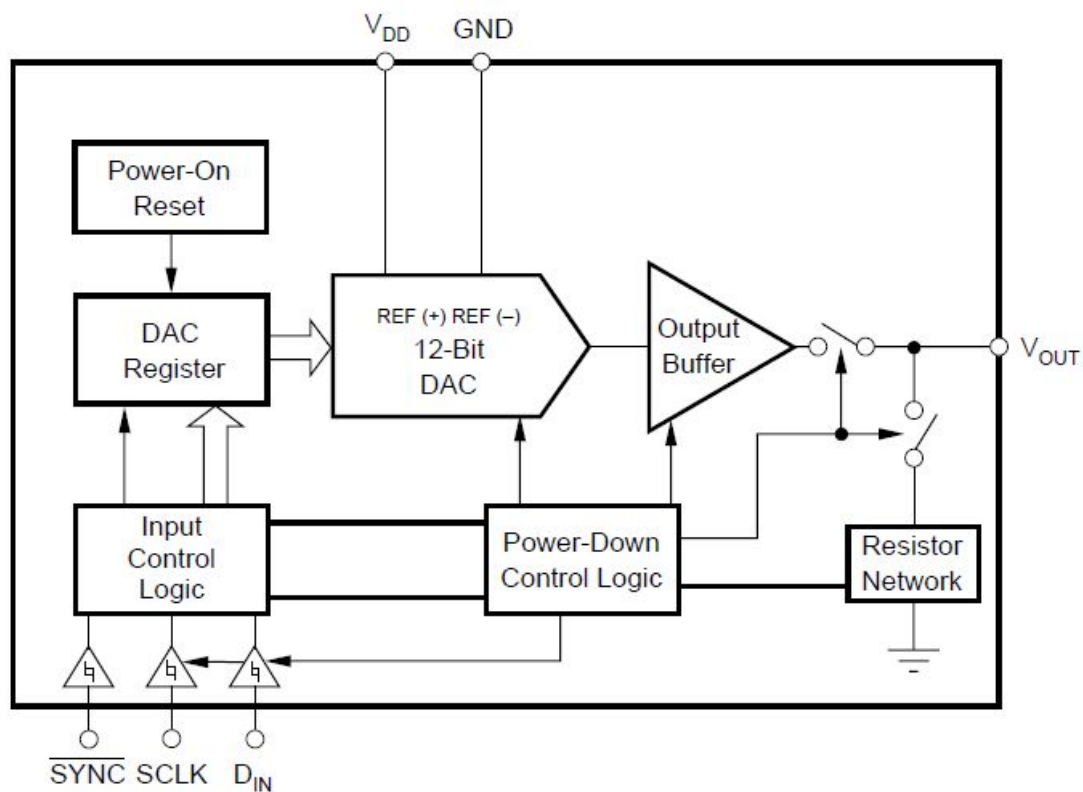
11.2 实验目的

- (1) 了解 DAC 数模转换原理;
- (2) 学习配置 TM4C123G 控制 DAC 参数,能够用 DAC 产生输出电压,并能输出一定规则的波形(如三角波,正弦波等);
- (3) 了解硬件发声原理,编程实现让 DAC 模块驱动扬声器唱歌。

11.3 实验原理

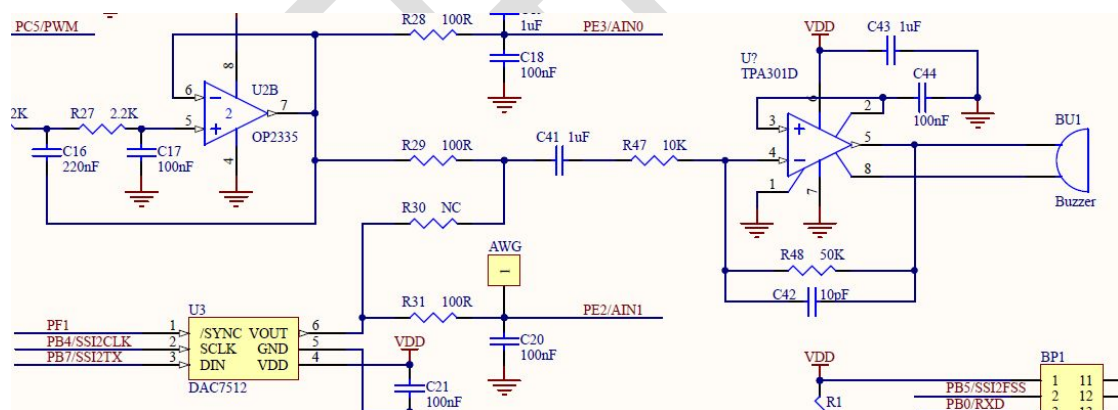
DAC(Digital to Analog Converter)数模转换器可以把处理器运算处理后输出的数字信号转换成模拟信号,完成对数字信号的复原工作。DA 转换,即把数字信号转换为模拟信号输出。简单的说,就是把数字信号按照一定的参考电压转换成电压值输出。例如,12 位分辨率时,数据 0XFFF 值对应满程参考电压,那么 0x7FF 就会输出半程参考电压。TM4C123G 的 DAC7512 模块的控制位较多,可以满足多种输出需求。

Tiva 实验板上的 DAC7512 支持二进制数或 2 的补码的数据格式。在 12 位分辨率时,采用二进制数时,输出数据范围是 0000h--0FFFh(8 位分辨率的是 000h--0FFh),满程电压输出为 0FFFh。采用 2 的补码时,输出数据范围是 0800h--07FFh(8 位分辨率的是 080h--7FFh)。为了输入输出的准确和稳定,



DAC_A 模块原理图

如图为实验板的扬声器模块电路图。主控芯片 DAC7512 模块控制输出到喇叭的电压。



扬声器电路图

11.4 实验步骤

- (1) 将 PC 和板载仿真器通过 USB 线相连;
- (2) 打开 Keil 项目工程文件
- (3) 编译项目工程, 并烧写到 Tiva 开发板中
- (4) 示波器连接 DAC 输出引脚, 观察输出波形。

11.5 实验现象

实验 1 实现输出斜波（在示波器上显示波形）。实际上这是一个阶梯波。配合延迟程序或者定时程序，每隔一段固定的时间，输出一个幅度递增（或递减）的恒定值，这样可以形成一个阶梯状的波形。如果间隔时间足够小，看上去是上升的直线。我们可以通过控制每个时间间隔的输出电压幅度值来产生我们需要的波形。

实验 2 采用同样的原理，输出数值改为正弦波即可。可以用示波器看到 P7.6 引脚输出正弦波形。

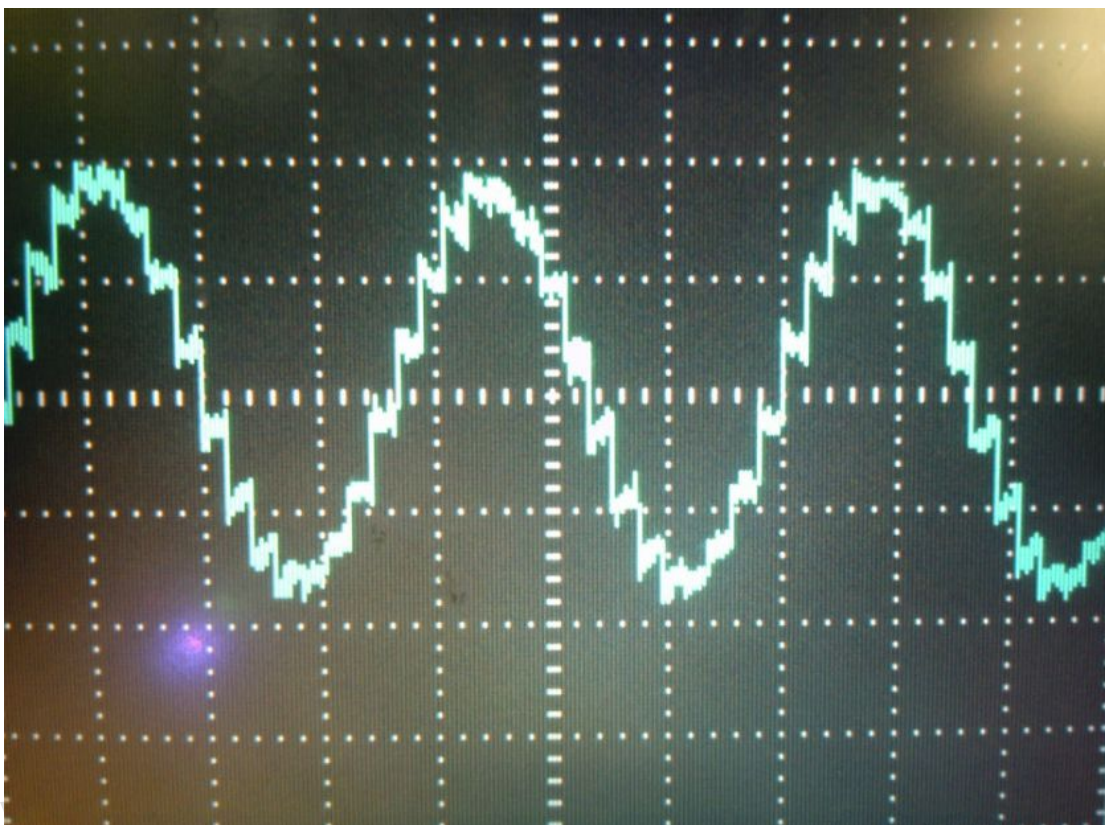
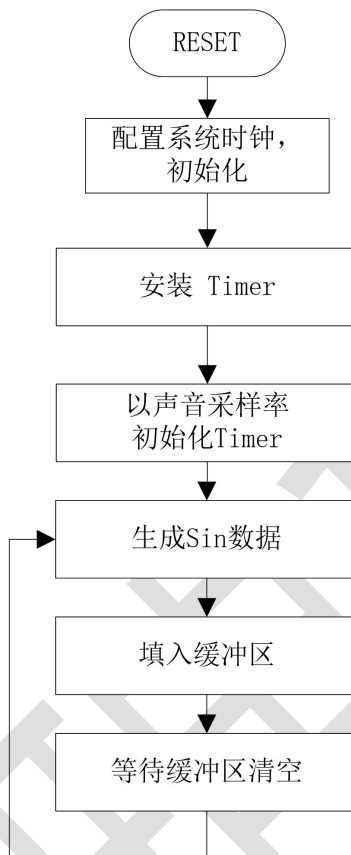


图 DAC 输出 Sin 波形

11.6 实验流程图



实验流程图

11.7 关键代码分析

定时器中断回调:

```
void timer0a_int_isr(void)
{
    TimerIntClear(TIMER_PIN_PORT, TIMER_TIMA_TIMEOUT);
    //清除中断
#ifdef DEBUG
    if (dac_buffer.count > 0)
    {
#endif // DEBUG
    if 0
        MAP_GPIOPinWrite(DAC_GPIO_SPI_CS, DAC_PIN_SPI_CS, ~DAC_PIN_SPI_CS);
    // ^SYNC 拉低

    MAP_SSIDataPut(DAC_PIN_SPI_PORT, dac_buffer.elems[dac_buffer.start]);

    // TODO: use isr handle this
```

```

        while (MAP_SSIBusy(DAC_PIN_SPI_PORT))
            ;

        MAP_GPIOPinWrite(DAC_GPIO_SPI_CS,    DAC_PIN_SPI_CS,    DAC_PIN_SPI_CS);
// SYNC 拉高
        #else
            DAC7512Write(dac_buffer.elems[dac_buffer.start]);
        #endif

        dac_buffer.start ++;
        dac_buffer.start %= (DAC_BUFFER_LEN/2);
        dac_buffer.count --;
#ifdef DEBUG
    }
#endif // DEBUG
}

```

Sin 波生成:

```

void dac_sinwave(uint16_t freq)
{
    float x, sample_count, theta;
    uint16_t delay;

    uint16_t data;
    uint32_t sys_ctl_clock, sample_delay;

    sys_ctl_clock = SysCtlClockGet(); // 获取系统时钟
    sample_delay = (sys_ctl_clock/6)/SAMPLERATE; //(sys_ctl_clock/(1000*6))*ms;
    sample_count = SAMPLERATE/freq; // 频率;

    for (x = 0; x < sample_count; x ++)
    {
        theta = x/sample_count;
        data = (AMPITUDE/2 - 1)*sin(2*PI*theta) + AMPITUDE/2;
        delay = sample_delay; // 时延
        DAC7512Write(data);
    }
}

```

11.8 思考题

- (1) 利用 DAC 模块，编制一个输出 2V 模拟电压的程序。
- (2) 利用 DAC7512 输出频率变化的正弦波。