SSI 使用例程说明

HelloM3 应用笔记

北京锐鑫同创科技有限公司

www.realsense.com.cn

www.hellom3.cn

SPI 操作

本章来讲一讲 LM3S9B96 上 SPI 的操作方法。SPI 总线系统是一种同步串行外设接口，它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。它可直接与各个厂家生产的多种标准外围器件直接接口，该接口一般使用 4 条线：串行时钟线(SCK)、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SS(有的 SPI 接口芯片带有中断信号线 INT、有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI)。

SPI 的通信原理很简单，它以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以(用于单向传输时，也就是半双工方式)。也是所有基于 SPI 的设备共有的，它们是 SDI(数据输入)，SDO(数据输出)，SCK(时钟)，CS(片选)。

Stellaris® LM3S9B96 微控制器内置两个同步串行接口(SSI)模块，每个 SSI 模块都能以主机或从机方式与片外器件进行同步串行通信，支持的同步串行接口格式包括飞思卡尔(Freescale)的 SPI、MICROWIRE 以及德州仪器(TI)同步串行接口(SSI)。

Stellaris® LM3S9B96 的两个同步串行接口模块，它们具有如下特性：

- 对 Freescale SPI，MICROWIRE 或 TI 的同步串行接口通信的可编程操作
- 主机或从机操作
- 可编程的时钟位速率和预分频
- 独立的发送和接受 FIFO，每一个都具有 16 位的宽度和 8 单元的深度
- 内部的回送测试模式，可进行诊断/调试测试
- 标准的基于先进先出(FIFI-BASE)和发送结束(End-of-Transmission)中断
- 通过直接内存访问控制器的高效的传输
  - 独立的发送和接受通道
  - 当 FIFO 中只有一个数据时，接受
  - 当 FIFO 中只有一个数据时，发送

SSI 模块对从外部设备接收到的数据执行从串行到并行的转换。CPU 只需访问数据、控制和状态信息，发送和接受由内部的 FIFO 单元进行缓冲。该 FIFO 可在发送和接受模式下独立存储 8 个 16 位的值。SSI 还支持 uDMA 接口。发送和接受 FIFO 可以被编程控制为 uDMA 的目的地址和源地址。可以通过设置 SSIDMACTL 寄存器相应的位来使能 uDMA 操作。

一般说来，发送 FIFO 和接收 FIFO 是一个 16 位宽度，8 单位深度的先入先出的缓存区。CPU 通过向 SSI 数据寄存器(SSIDR)中读出或写入数据来从 FIFO 中读取数据或向 FIFO 中写如入数据。写入到发送 FIFO 中的数据会一直保存在 FIFO 中，直到被发送逻辑读走。

下面就以一个小例子来说明一下 SPI 作为主机的时候发送数据的配置方法

例 1、SPI 主机发送

在贴出程序之前首先说明一下作为主机时引脚的配置情况。

PA2 ———————— CLK

PA3 ———————— Fss

PA4 ———————— RX

PA5 ———————— TX

下面是源程序

```
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"

#define BUFFER_SIZE       16

unsigned char temp[BUFFER_SIZE];

int main(){
      unsigned char i;
      //
      //    运行在外部 16MHz 模式下
      //
      SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                              SYSCTL_XTAL_16MHZ);
      //
      //    使能 SSI0 模块
      //
      SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

      //
      //    使能 SSI0 的输入输出引脚 PortA[5..2]
      //
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

      //
      //    配置相关引脚
      //
       GPIOPinConfigure(GPIO_PA2_SSI0CLK);
      GPIOPinConfigure(GPIO_PA3_SSI0FSS);
      GPIOPinConfigure(GPIO_PA4_SSI0RX);
      GPIOPinConfigure(GPIO_PA5_SSI0TX);

      //
      //    配置管脚相应的类型
      //
      GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
                              GPIO_PIN_2);

      //
      //    设置 SSI 协议、工作模式、位速率和数据宽度
      //
      //    SSI0_BASE                 -      使用 SSI0
```

```
    //    SysCtlClockGet()      -     提供到 SSI0 的时钟速率
    //    SSI_FRF_MOTO_MODE_0   -     数据传输协议，极性：0，相位：0
    //    SSI_MODE_MASTER       -     模式选择：配置为主机模式
    //    1000000               -     设定 SSI 模块位速率为 1M
    //    8                     -     8 位数据宽度
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
                       SSI_MODE_MASTER, 1000000, 8);

    //
    //    使能 SSI 模块
    //
    SSIEnable(SSI0_BASE);

    //
    //    清除缓冲区的数据，确保读到的数据时正确的。
    //
    //while(SSIDataGetNonBlocking(SSI0_BASE, &ulDataRx[0]));

    //
    //    初始化发送数组
    //
    for(i = 0;i < BUFFER_SIZE; i++){
        temp[i] = i;
    }
    //
    //    进入死循环
    //
    while(1){
        SSIDataPut(SSI0_BASE, temp[(i++) % BUFFER_SIZE]);
        while(!SSIBusy(SSI0_BASE));
    }
}
```

本程序非常简单，也就 Hello World 级别的。程序一开始配置好时钟后就配置要用到的相应的端口，这个基本上算是惯例了。然后配置 SSI 模块，最后用 SSI 模块不断的向外发送字符。

先来看看效果(如图 1 所示)，后面再讲解一下程序的配置过程。

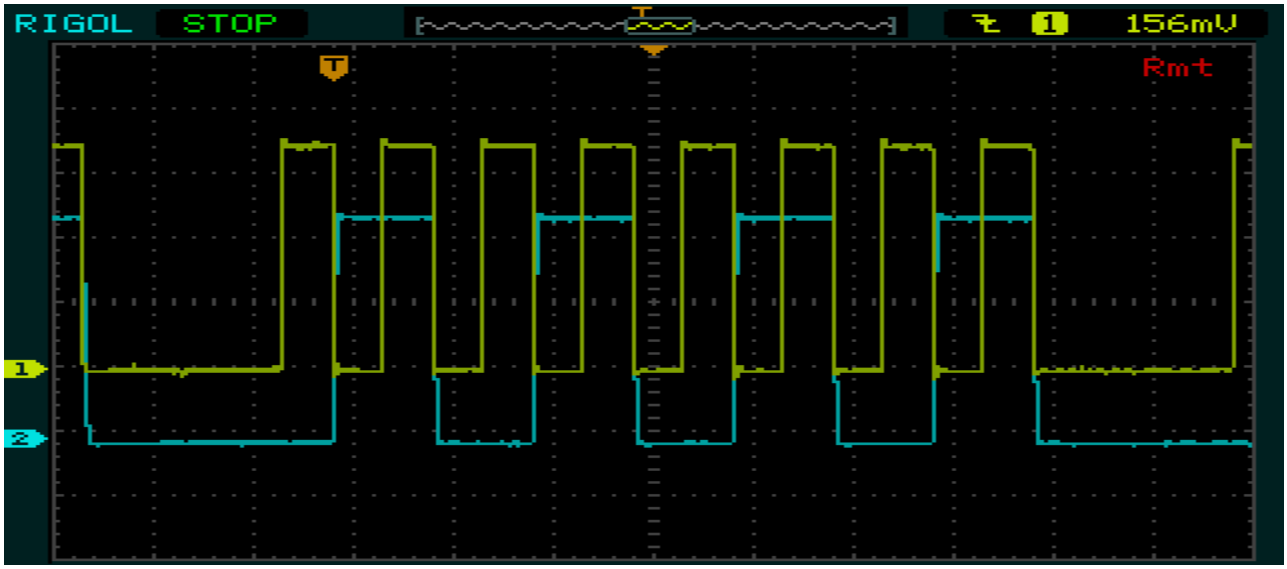

图 1、SPI 主机时的发送

前面的管脚配置照猫画虎就行，就掠过不说了。

SSIConfigSetExpClk 这一句是设置 SSI 协议、工作模式、位速率和数据宽度的，直接调用该函数就可以把 SSI 设置为正确的格式。第一个参数是选择使用哪个 SSI 模块，这里选用了 SSI0 模块。第二个参数是当前的时钟速率，为了修改和移植等的方便起见，调用了 SysCtlClockGet()函数来直接获得。第三个参数是选择传输的协议，其中可选的协议和含义如表 1 所示。第四个参数是选择模式，可选择的模式如表 2 所示，这里选择主机模式。第五个参数是为 SSI 模块设定位速率，此处我们使用的是 1MHz。最后一个是使用的数据的宽度，宽度一般为 8 位。

表 1、可选择的协议

协议	说明
SSI_FRF_MOTO_MODE_0	Motorola 格式，极性 0，相位 0
SSI_FRF_MOTO_MODE_1	Motorola 格式，极性 0，相位 1
SSI_FRF_MOTO_MODE_2	Motorola 格式，极性 1，相位 0
SSI_FRF_MOTO_MODE_3	Motorola 格式，极性 1，相位 1
SSI_FRF_TI	TI 的帧格式
SSI_FRF_NMW	National MicroWire 的帧格式

表 2、可选择的模式

模式	说明
SSI_MODE_MASTER	SSI 主机模式
SSI_MODE_SLAVE	SSI 从机模式
SSI_MODE_SLAVE_OD	SSI 从机只接收模式，输出被禁止

到此为止配置就完成了。好像挺简单的。为了让我显得不是说的太潦草，我还是在说一下这个协议的选择的问题吧。Motorola 有好几种格式，这几种格式代表的是什么意思呢，极性相位是什么呢。

Motorola 格式最大的特点就是 SSIClk 信号的不活动状态和相位均可通过 SSISCR0 控制寄存器中的 SPO 和 SPH 位来设置。在没有进行数据传传输的时候，SPO 时钟极性控制位为低时，它在 SSIClk 管脚上产生稳定的低电平值。如果 SPO 位为高，它在 SSIClk 管脚上产生一个稳定的高电平值。SPH 相位控制位用来选择捕获数据的时钟边沿并允许边沿改变状态。SPH 在第一个传输位上的影响最大，因为它可以在第一个数据捕获边沿之前允许或不允许一次时钟转换。当 SPH 相位控制位为低时，在第一个时钟边沿转换时捕获数据。如果 SPH 位为高，则在第二个时钟边沿转换时捕获数据。

下面结合我们图 1 示波器上的内容来分析一下相位和极性的关系。很明显，示波器上第 1 路为 SSIClk 的波形，也就是黄色的线。我们选择的是相位 0，极性 0，根据上面那段话的意思，相位 0 说明 SSIClk 管脚在没有数据传输的时候为低电平，极性 0 就是在第一个时钟边沿转换时捕获数据，因为 SSIClk 在空闲的时候是低电平，它第一个时钟边沿应该是上升沿，即从第一个上升沿开始读取数据。

SSIEnable 这一句就是使能 SSI。前面配置完毕之后使能 SSI 模块就开始准备发送或接收数据了。

剩下的死循环里这两句第一句是从 SSI 模块发送数据的函数，第二个是等待数据发送完毕。至于读取数据的函数呢就是 SSIDataGet 了

看上去 SSI 作为主机时的操作并不难。

下面再以一个例子演示一下 SSI 作为从机时的配置和操作方法

例 2、SSI 作为从机

下面是 SSI 作为从机时的源程序，和作为主机的时候有点相似。

```
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"

#define BUFFER_SIZE   1024

unsigned long i = 0;
unsigned char temp[BUFFER_SIZE];

int main(){
   unsigned long i = 0;
   //
   //    运行在外部 16MHz 模式下
   //
   SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                  SYSCTL_XTAL_16MHZ);
   //
   //    使能 SSI0 模块
   //
   SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

   //
   //    使能 SSI0 的输入输出引脚 PortA[5..2]
   //
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

   //
   //    配置相关引脚
   //
   GPIOPinConfigure(GPIO_PA2_SSI0CLK);
      GPIOPinConfigure(GPIO_PA3_SSI0FSS);
      GPIOPinConfigure(GPIO_PA4_SSI0RX);
      GPIOPinConfigure(GPIO_PA5_SSI0TX);

   //
   //    配置管脚相应的类型
   //
   GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
                  GPIO_PIN_2);

   //
   //    设置 SSI 协议、工作模式、位速率和数据宽度
   //
   //    SSI0_BASE               -    使用 SSI0
   //    SysCtlClockGet()        -    提供到 SSI0 的时钟速率
   //    SSI_FRF_MOTO_MODE_0     -    数据传输协议，极性：0，相位：0
   //    SSI_MODE_SLAVE          -    模式选择：配置为从机模式
   //    1000000                 -    设定 SSI 模块位速率为 1M
   //    8                       -    8 位数据宽度
```

```
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
                                  SSI_MODE_SLAVE, 1000000, 8);

    //
    //    使能 SSI 模块
    //
    SSIEnable(SSI0_BASE);

    //
    //    清除缓冲区的数据，确保读到的数据时正确的。
    //
    while(SSIDataGetNonBlocking(SSI0_BASE, (unsigned long *)temp));

    //
    //     接收数据
    //
    for(i = 0;i < BUFFER_SIZE; i++){
          SSIDataGet(SSI0_BASE, (unsigned long *)(temp + i));
    }

    //
    //    进入死循环
    //
    while(1);
}
```

该例程和例 1 基本一样，还得我都没得说了，前面的一段看例 1 去吧。从使能 SSI0 模块开始，和以前有所不同了，因为是从机，所以只能被动挨打。当模块使能后，不管缓冲区里面有没有数据，首先要清空缓冲区，确保读到的数据是正确的。接受数据的时候程序会一直停在 SSIDataGet，直到读到够了数据，最后进入死循环里。

写了两个例程了，但是好像展示不了 LM3S9B96 的 SSI 的精髓，下面再写一个例程，希望可以如偿所愿。

例 3、带 uDMA 的 SSI 操作

下面直接把源程序给出来，结合程序来说明一下

```
#include        ……


#define SYSTICKS_PER_SECOND          10
#define MEM_BUFFER_SIZE              1024
#define TESTSECONDS                  10


unsigned char oldSeconds = 0;
unsigned char newSeconds = 0;
static unsigned long g_ulCPUUsage;
static unsigned long g_uluDMAErrCount = 0;
```

```
static unsigned long g_ulBadISR = 0;
static unsigned char g_ulSrcBuf[MEM_BUFFER_SIZE];
unsigned char ucControlTable[1024] __attribute__ ((aligned(1024)));
unsigned long tick = 0;
void SSI0IntHandler(void){
    unsigned long ulStatus;
    unsigned long ulMode;
    //
    //  读取 SSI0 的中断状态
    //
    ulStatus = SSIIntStatus(SSI0_BASE, 1);
    //
    //  清除 SSI0 的中断
    //
    SSIIntClear(SSI0_BASE, ulStatus);
    //
    //  获取 uDMA 通道的模式
    //
    ulMode = uDMAChannelModeGet(UDMA_CHANNEL_SSI0TX | UDMA_PRI_SELECT);
    if(ulMode == UDMA_MODE_STOP)
    {
        //
        //  设置 uDMA 通道的属性
        //
        uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |          //  通道为 SSI0TX
                               UDMA_PRI_SELECT ,              //  数据结构
                               UDMA_MODE_BASIC ,              //  基本 DMA 模式
                               (void*)(g_ulSrcBuf),           //  源地址
                               (void*)0x40008008 ,            //  目的地址为 SSI0 的数据寄存器
                               256);                          //  一次传输为 8 个数据
    }
    if(!uDMAChannelIsEnabled(UDMA_CHANNEL_SSI0TX))
    {
        //
        //  重新设置通道属性
```

```
            //
            uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |          //  通道为 SSI0TX
                                   UDMA_PRI_SELECT ,              //  数据结构
                                   UDMA_MODE_BASIC ,              //  基本 DMA 模式
                                   (void*)(g_ulSrcBuf),           //  源地址
                                   (void*)0x40008008 ,            //  目的地址为 SSI0 的数据寄存器
                                   256);                          //  一次传输为 8 个数据
            //
            //  重新使能通道
            //
            uDMAChannelEnable(UDMA_CHANNEL_SSI0TX);
        }
    }
    //
    //  该中断来自于 uDMA 内存通道，该中断会增加进入中断的次数，并且重启另一个内存发送
    //
    void uDMAIntHandler(void){
        unsigned long ulMode;
        //
        //  获取 uDMA 通道模式
        //
        ulMode = uDMAChannelModeGet(UDMA_CHANNEL_SSI0TX);
        if(ulMode == UDMA_MODE_STOP)
        {
            //
            //  配置通道属性
            //
            uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |      //  通道为 SSI0TX
                                   UDMA_PRI_SELECT ,          //  数据结构
                                   UDMA_MODE_BASIC ,          //  基本 DMA 模式
                                   (void*)(g_ulSrcBuf),       //  源地址
                                   (void*)0x40008008 ,        //  目的地址为 SSI0 的数据寄存器
                                   256);                      //  一次传输为 8 个数据

            //
```

```
            //    初始化通道
            //
            uDMAChannelEnable(UDMA_CHANNEL_SSI0TX);
            uDMAChannelRequest(UDMA_CHANNEL_SSI0TX);
        }
        //
        //   如果通道没有停止，增加记录出错的个数
        //
        else
        {
            g_ulBadISR++;
        }
}
//
//   uDMA 出错中断处理函数，该中断在试图执行发送的时候总线产生错误的时候发生。
//   该中断中只是简单的将出错次数增加。
//
void uDMAErrorHandler(void){
        unsigned long ulStatus;
        //
        //   检查 uDMA 出错位
        //
        ulStatus = uDMAErrorStatusGet();

        //
        // If there is a uDMA error, then clear the error and increment
        // the error counter.
        //
        if(ulStatus)
        {
            uDMAErrorStatusClear();
            g_uluDMAErrCount++;
        }
}
```

```
//
//  滴答中断。
//
void SystickHandler(void){
    static unsigned char addSeconds = 0;
//  unsigned long ulMode;

    addSeconds++;

    if(!(addSeconds % SYSTICKS_PER_SECOND)){
        newSeconds++;
    }
    //
     // Call the CPU usage tick function.    This function will compute the amount
     // of cycles used by the CPU since the last call and return the result in
     // percent in fixed point 16.16 format.
     //
     g_ulCPUUsage = CPUUsageTick();

    GPIOPinWrite(GPIO_PORTF_BASE ,GPIO_PIN_3,
                 GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_3) ^ GPIO_PIN_3);
}
//
//  初始化 LED
//
void ledConfigure(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0);
}
void SSIInit(void){
    unsigned long   ulBitRate   =   SysCtlClockGet() / 20;

     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);          //  使能 SSI0 模块所在的 GPIO 端口
     SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);           //  使能 SSI0 模块
```

```
    //
    //    配置相关引脚
    //
    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
     GPIOPinConfigure(GPIO_PA3_SSI0FSS);
     GPIOPinConfigure(GPIO_PA4_SSI0RX);
     GPIOPinConfigure(GPIO_PA5_SSI0TX);
    //
    //    配置管脚相应的类型
    //
    GPIOPinTypeSSI(     GPIO_PORTA_BASE,
                        GPIO_PIN_5 |
                        GPIO_PIN_4 |
                        GPIO_PIN_3 |
                        GPIO_PIN_2);
    //
    //    设置 SSI 协议、工作模式、位速率和数据宽度
    //
    //    SSI0_BASE                -     使用 SSI0
    //    SysCtlClockGet()     -     提供到 SSI0 的时钟速率
    //    SSI_FRF_MOTO_MODE_0-     数据传输协议，极性：0，相位：0
    //    SSI_MODE_MASTER      -     模式选择：配置为主机模式
    //    1000000                  -     设定 SSI 模块位速率为 1M
    //    8                        -     8 位数据宽度
    //
    SSIConfigSetExpClk( SSI0_BASE,                          //    配置 SSI0 模块
                        SysCtlClockGet(),                   //    获取当前时钟
                        SSI_FRF_MOTO_MODE_0,                //    Motorola 格式，模式 0
                        SSI_MODE_MASTER,                    //    设置为主机
                        1000000,                            //    为速率为 1MHz
                        8);                                 //    数据宽度为 8 位
     SSIDMAEnable(SSI0_BASE , SSI_DMA_TX);
     SSIEnable(SSI0_BASE);
    IntEnable(INT_SSI0);                                    //    使能 SSI 中断
}
```

```
void uDMAInit(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);                    //  使能 DMA 时钟
    uDMAEnable();                                                  //  使能 DMA 模块
    uDMAControlBaseSet(ucControlTable);                            //  设置 DMA 的控制表

    uDMAChannelAttributeDisable(UDMA_CHANNEL_SSI0TX ,              //  清除通道属性
                                UDMA_ATTR_USEBURST |
                                UDMA_ATTR_ALTSELECT |
                                UDMA_ATTR_REQMASK);

    uDMAChannelAttributeEnable(UDMA_CHANNEL_SSI0TX ,
                               UDMA_ATTR_HIGH_PRIORITY);       //  通道为高优先级

    uDMAChannelControlSet(UDMA_CHANNEL_SSI0TX |                //  通道为 SSI0TX
                          UDMA_PRI_SELECT,                     //  数据结构
                          UDMA_SIZE_8 |                        //  数据长度 8 位
                          UDMA_SRC_INC_8 |                     //  源地址累加
                          UDMA_DST_INC_NONE |                  //  目的地址不累加
                          UDMA_ARB_8);                         //  仲裁大小为 8

    uDMAChannelTransferSet(UDMA_CHANNEL_SSI0TX |               //  通道为 SSI0TX
                           UDMA_PRI_SELECT ,                   //  数据结构
                           UDMA_MODE_BASIC ,                   //  基本 DMA 模式
                           (void*)(g_ulSrcBuf),                //  源地址
                           (void*)0x40008008 ,                 //  目的地址为 SSI0 的数据寄存器
                           256);                               //  一次传输为 8 个数据

    uDMAChannelEnable(UDMA_CHANNEL_SSI0TX);                    //  使能 UDMA_CHANNEL_SSI0TX 通道
    while(!uDMAChannelIsEnabled(UDMA_CHANNEL_SSI0TX));         //   等待通道使能
    IntEnable(INT_UDMAERR);                                    //  使能 uDMA 软件中断
}

int main(void){
     static char cStrBuf[40];
    unsigned long ulIndex = 0;
```

```
tContext sContext;
tRectangle sRect;

//
//  设置使用外部 16MHz 晶振
//
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_XTAL_16MHZ |
                        SYSCTL_OSC_MAIN);

//
//  初始化要发送的数据
//
for(ulIndex = 0; ulIndex < MEM_BUFFER_SIZE; ulIndex++){
      g_ulSrcBuf[ulIndex] = (unsigned char)ulIndex;
}

//
//  设置引脚为输出
//
PinoutSet();

//
//  初始化 LCD
//
Lcd240x320x16_8bitInit();

//
//  初始化屏幕区域
//
GrContextInit(&sContext, &g_sLcd240x320x16_8bit);

//
//  画标题栏
//
sRect.sXMin = 0;
```

```
    sRect.sYMin = 0;
    sRect.sXMax = GrContextDpyWidthGet(&sContext) - 1;
    sRect.sYMax = 23;
    GrContextForegroundSet(&sContext, ClrDarkBlue);
    GrRectFill(&sContext, &sRect);

    //
    //  画标题栏边框
    //
    GrContextForegroundSet(&sContext, ClrWhite);
    GrRectDraw(&sContext, &sRect);

    //
    //  写标题
    //
    GrContextFontSet(&sContext, &g_sFontCm20);
    GrStringDrawCentered(&sContext, "Systick Demo", -1,
                         GrContextDpyWidthGet(&sContext) / 2, 10, 0);

    //
    // Say hello using the Computer Modern 40 point font.
    //
    //GrContextFontSet(&sContext, &g_sFontCm40);
   GrContextFontSet(&sContext, &g_sFontCmss18b);
   usnprintf(cStrBuf, sizeof(cStrBuf), "Stellaris @ %u MHz",
             SysCtlClockGet() / 1000000);
    GrStringDrawCentered(&sContext, cStrBuf, -1,
                         GrContextDpyWidthGet(&sContext) / 2,
                         40,//((GrContextDpyHeightGet(&sContext) - 24) / 2) + 24,
                         0);

    //
    // Show static text and field labels on the display.
    //
    GrStringDrawCentered(&sContext, "uDMA Mem Transfers", -1,
```

```
                                GrContextDpyWidthGet(&sContext) / 2, 62, 0);
    GrStringDrawCentered(&sContext, "uDMA SSI0 Transfers", -1,
                                GrContextDpyWidthGet(&sContext) / 2, 84, 0);

    //
    //    初始化 uDMA
    //
    uDMAInit();

    //
    //    初始化 SSI
    //
    SSIInit();

    //
     //   CPU 利用率初始化
     //
     CPUUsageInit(SysCtlClockGet(), SYSTICKS_PER_SECOND, 2);

    ledConfigure();

    //
    //    配置 Systick 每秒的滴答数为 SYSTICKS_PER_SECOND，用来作为时钟参考。
    //    允许 Systick 产生中断。
     //
     SysTickPeriodSet(SysCtlClockGet() / SYSTICKS_PER_SECOND);
     SysTickIntEnable();
     SysTickEnable();

     //
    //    刷新缓冲区的绘图操作。
     //
     GrFlush(&sContext);

     //
```

```
    //  显示更新屏幕
    //
    while(1)
    {
        if(oldSeconds != newSeconds){
            //
            //    在屏幕上显示一条显示 CPU 占用率的信息。该百分比的小数部分被忽略。
            //
            usnprintf(cStrBuf, sizeof(cStrBuf), "CPU utilization %2u%%",
                      g_ulCPUUsage >> 16);
            GrStringDrawCentered(&sContext, cStrBuf, -1,
                                 GrContextDpyWidthGet(&sContext) / 2, 160, 1);

            //
            //    显示还有多久结束操作
            //
            usnprintf(cStrBuf, sizeof(cStrBuf), " Test ends in %d seconds ",
                      TESTSECONDS - oldSeconds);
            GrStringDrawCentered(&sContext, cStrBuf, -1,
                                 GrContextDpyWidthGet(&sContext) / 2, 120, 1);

            //
            //    更新时间
            //
            oldSeconds = newSeconds;
        }

        //
        //    如果 CPU 没有事情做则进入睡眠模式，允许检测 CPU 利用率如果处理器睡眠过度则
        //    调试器连接目标会有难度。
        //
        SysCtlSleep();

        //
        //    检查是否运行到了设定的时间。
```

```
            //
             if(oldSeconds >= TESTSECONDS)
             {
                    break;
             }
         }
        //
        //    在显示屏上显示例程停止。
        //
         GrContextForegroundSet(&sContext, ClrRed);
         GrStringDrawCentered(&sContext, "                    Stopped                    ", -1,
                                    GrContextDpyWidthGet(&sContext) / 2, 120, 1);
        //
        //    停止 SYSTICK 中断
        //
        //SysTickDisable();
        while(1){
             //
             //    显示当 CPU 不休眠的时候 CPU 的占用率
             //
             GrContextForegroundSet(&sContext, ClrWhite);
             usnprintf(cStrBuf, sizeof(cStrBuf), "CPU not sleep and utilization is %2u%%",
                            g_ulCPUUsage >> 16);
            GrStringDrawCentered(&sContext, cStrBuf, -1,
                                          GrContextDpyWidthGet(&sContext) / 2, 160, 1);
        }
    }
```

写程序的时候，加入注释和空行使程序具有更好的可读性。但对于像我这样的来说未必是件好事了。我几乎把整个程序都放到这里来了，由于空行和注释，使这边文档的内容白白增长了不少，没有耐心的人估计看两页就看不下去了。但是我觉得我已经很精简了。鉴于本文中增加了许多和 SPI 关系不是很大的内容。所以我按内容的轻重来讲解。

主函数开始，前面配置屏幕的内容我就掠过不讲了，下面从 uDMAInit 这个函数开始讲起吧。该函数是对 uDMA 的初始化，其中的每一句都添加了注释，设置和使用 uDMA 的时候函数调用的顺序如下：

- 调用 uDMAEnable()一次来使能控制器；
- 调用 uDMAControlBaseSet()一次来设置通道控制表；
- 调用 uDMAChannelAttributeEnable()一次或很少调用它来配置通道的操作；

- uDMAChannelControlSet()一般用来设置数据传输的特性。如果数据传输的特性并不发生改变，那么只须调用此函数一次；
- uDMAChannelTransferSet()一般用来设置一次传输的缓冲区指针和尺寸。在开始一次新传输之前调用此函数；
- uDMAChannelEnable()使能一个通道以便执行数据传输；
- uDMAChannelRequest()一般用来开始一个基于软件的传输。这个函数通常不用于 基于外设的传输。

一旦 uDMA 控制器使能，你必须告诉它到哪里去查找系统存储器中的通道控制结构。这一步通过使用函数 uDMAControlBaseSet()和把一个指针传给通道控制结构的底部来完成。控制结构必须由应用程序分配。分配的方法是声明一个数据数组类型为 char 或 unsigned char。为了支持全部通道和传输模式，控制表数组应为 1024 个字节，但根据所用的传输模式和实际使用的通道数，它的值可少于 1024 个字节。注意：控制表必须对齐一个 1024 字节边界。

uDMA 控制器支持若干个通道。每个通道都具有一组属性标志来控制某些 uDMA 特性和通道操作 。属性标志由函数 uDMAChannelAttributeEnable() 设 置 和 函 数 uDMAChannelAttributeDisable()清除。通过使用函数 uDMAChannelAttributeGet()就可询问通道属性标志的设置。

下一步，必须设置 DMA 传输的控制参数。这些参数控制着要被传输的数据项目的大小和地址增量。函数 uDMAChannelControlSet()一般用来设置这些控制参数。

全部所提及的函数到目前为止都只是被使用一次或很少被使用来对 uDMA 通道和传输进行设置。为了设置传输地址、传输大小和传输模式，可以使用函数 uDMAChannelTransferSet()。在开始每一个新的传输时，必须要调用这个函数。一旦所有事情都已设置好，那么就可以调用 uDMAChannelEnable()来使能通道，而这一步必须在开始一个新传输之前处理。在完成传输时，uDMA 控制器将会自动禁止通道。调用 uDMAChannelDisable()，就可以手动禁止通道。

用户也可以使用其它的函数来查询通道的状态，无论是通过中断还是查询。uDMAChannelSizeGet()一般用来查找通道中的剩余要传输的数据量。当传输完成时，它的值将会为 0。uDMAChannelModeGet()函数一般用来查找一个 uDMA 通道的传输模式。它一般用来查看模式指示已停止是否意味着之前正在运行的通道的传输已完成。函数 uDMAChannelIsEnabled()一般用来确定一个特殊的通道是否使能。

如果应用程序正在使用运行时（run-time）中断注册（请参考 IntRegister()）,那么可以使用函数 uDMAIntRegister()来安装 uDMA 控制器的一个中断处理程序。这个函数也将会使能系统中断控制器的中断。如果使用编译时（compile-time）中断注册，则可调用函数 IntEnable()来使能 uDMA 中断。当一个中断处理程序已被 uDMAIntRegister()安装完毕后，则可以调用 uDMAIntUnregister()来将其卸载。

这个中断处理程序只供软件开始传输或错误使用。外设的 uDMA 中断发生在外设的专 用中断通道中，这些中断应该由外设中断处理程序进行处理。因此无需应答或清除 uDMA 中断源。这些中断源在进行中断服务时会被自动清除。

uDMA 中断处理程序使用函数 uDMAErrorStatusGet()来测试一个 uDMA 错误是否发生。 如果发生，则调用 uDMAErrorStatusClear()来清除中断。

关于 uDMA 的我觉得已经说的不少了，下面是 SSIInit 函数。该韩式是对 SSI 的初始化，仔细看看，和前面的例程是一样的，应该都明白了吧。

再往后就是 Systick 的一些设置了，Systick 每秒中断的此时在程序前面已经定义。Systick 的中断用来计时和反转 LED 的状态。

uDMA 错误中断其实什么也没干，就是简单的记录了一下出错的次数，然后把中断清除掉了。uDMA 中断来自于内存，进入中断后首先要获取中断的模式，然后判断是否为 UDMA_MODE_STOP，如果是则重新设置通道，并重新初始化该通道。否则记录为出错的此时。

SSI0 中断里，首先获取中断状态，清除中断，然后获得 uDMA 的通道的模式，根据通道的模式设置后面的内容。

该程序执行的时候开始从给定的时间倒计时，此时会通过 uDMA 向 SSI0 发送数据，在此过程中，CPU 会进入睡眠模式，统计 CPU 的利用率大概为 0%。当设定的时间过后会停止发送数据，并且 CPU 不再进入睡眠模式，此时统计的 CPU 利用率，利用率大概为 16%。

好了，对 SPI 的讲解就到处为止了。

附录 A

北京锐鑫同创是 TI 第三方合作伙伴，专注于 TI Stellaris M3 产品的方案设计、市场推广和技术服务，公司以“把握市场脉搏，专注技术创新，提供诚信服务，实现共赢发展！”为核心价值理念，为客户提供实时、高效的技术和服务。

电话：010-82418301
传真：010-82418302
Email：support@realsense.com.cn
网站：www.realsense.com.cn
技术论坛: www.hellom3.cn