第七章习题解答

1. 中断控制器应该具有哪些功能？
   解答：
   中断控制器应具有以下功能：中断请求信号的保持与清除，中断源的识别，中断允许控制，中断优先级设置。

2. 什么是中断类型码？什么是中断向量？
   解答：中断类型码：中断源的编码；中断向量：中断服务程序的入口地址

3. 简述 intel80x86 微处理器在实模式下进入中断处理的过程。
   解答：中断请求——中断响应（获取中断类型码、入栈保护断点、现场）——根据中断类型码查找中断向量表获得中断服务程序入口地址从而进入中断处理

4. intel80x86 微处理器在实模式下中断类型码为 8 的中断向量存放在中断向量表的哪个地址空间？
   解答：N=8，N*4=0x20, 中断类型码为 8 的中断向量存放在中断向量表的物理地址为 0x20~0x23 的 4 个字节地址空间

5. 简述 intel80x86 微处理器与 Microblaze 微处理器进入中断服务程序过程的异同。
   解答：intel80x86 微处理器进入中断服务程序是由 CPU 根据中断类型码查找中断向量表或中断描述符表获取中断服务程序的入口地址来实现的，该过程由硬件实现
   Microblaze 微处理器进入中断服务程序是总的中断服务程序查找其自身预设的数据结构中保存的中断服务程序句柄，通过调用该句柄来实现，该过程由软件实现

6. 简述 Intel80x86 与 MicroBlaze 识别中断源的异同。
   解答：Intel80x86 通过中断类型码来识别不同种类的中断源
   MicroBlaze 识别中断源需要通过读取中断控制器的中断请求寄存器，逐个查询才能识别。

7. 修改例 7. 3 的程序，利用定时器实现 100ms 的延时，控制 8 位 LED 灯同时闪烁显示。
   解答：定时器实现 100ms 的延时，假定采用减计数，定时器需初始化为：10,000,000.
   8 位 LED 灯同时闪烁显示也就是在时钟的中断服务程序里每次将原来输出的值全部取反之后再输出，而不是移位。因此针对程序例 7.3 仅需要修改两处：
   1）#define RESET_VALUE　　10000000//计数 10M 个时钟脉冲的初始值
   2）

```
voidTimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber)
{
   Xil_Out32(XPAR_LEDS_8BITS_BASEADDR,LedBits);//产生中断时，输出 LED 显示值
   LedBits=~LedBits；//将 LED 各位取反
}
```

8. 已知某共阳极 7 段数码管的连接电路如图 7- 34 所示，试采用双通道 AXI GPIO 并行接口

以及 AXI timer 采用动态显示控制方式，控制 4 个 7 段数码管同时显示数字 0，1，2，3。要求采用硬件定时器实现时延控制。画出基于 MicroBLAZE 微处理器 AXI 总线的电路原理图并基于 standalone 提供的库函数编写控制程序。

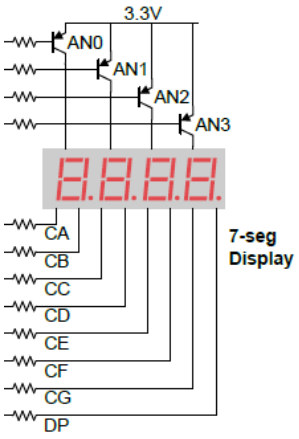
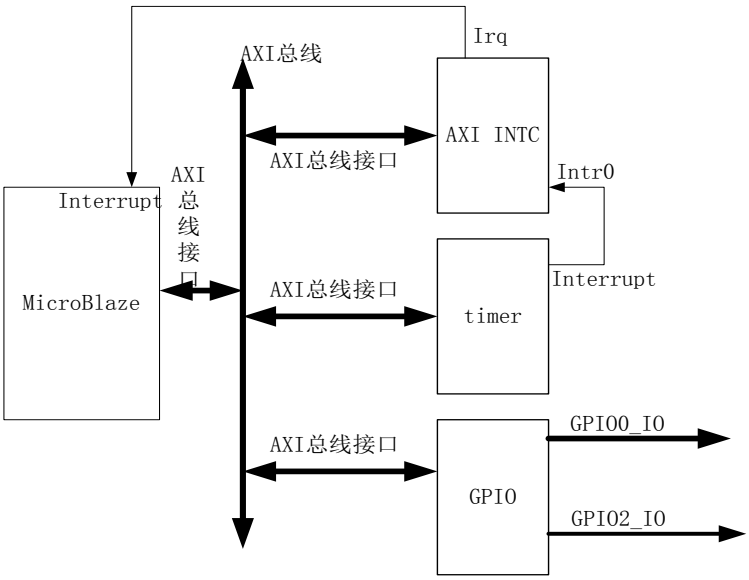


图 7- 1 7 段数码管连接电路

解答：

电路原理框图如下：



假定 GPIO 与 7 段数码管之间的引脚连接与例 6.6 相同，定时器延时时间为 50ms，以保证 4 个 7 段数码管能够在 0.2S 内全部显示一遍。控制程序如下：

```
#include "xparameters.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "xil_exception.h"

#define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID//来自 xparameters.h 的宏
#define INTC_DEVICE_ID        XPAR_INTC_0_DEVICE_ID//来自 xparameters.h 的宏
#define TMRCTR_INTERRUPT_IDXPAR_INTC_0_TMRCTR_0_VEC_ID//来自 xparameters.h 的宏
#define TIMER_CNTR_0      0//定时器 0
#define RESET_VALUE 5000000//计数 5M 个时钟脉冲的初始值
```

```
intTmrCtrIntrExample(XIntc* IntcInstancePtr,
                XTmrCtr* InstancePtr,
                u16DeviceId,
                u16IntrId,
                u8 TmrCtrNumber);//定时器中断示例函数
staticintTmrCtrSetupIntrSystem(XIntc* IntcInstancePtr,
                        XTmrCtr* InstancePtr,
                        u16DeviceId,
                        u16IntrId,
                        u8 TmrCtrNumber);//定时器中断系统初始化函数
void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber);//定时器中断服务程序
XIntcInterruptController;   /* 实例化中断控制器 */
XTmrCtrTimerCounterInst;    /* 实例化定时器 */
u32 LedBits;//LED 显示器的显示位置标识
charsegcode[4]={0x8c,0x88,0x92,0x92};
char pos=0xf7;     //初始化位码
inti;

int main(void)
{
   int Status;
   Xil_Out8(XPAR_SEG_0_BASEADDR+0x4,0x0);     //使 GPIO_IO 通道输出
   Xil_Out8(XPAR_SEG_0_BASEADDR+0xc,0x0);     //使 GPIO2_IO 通道输出
   Status = TmrCtrIntrExample(&InterruptController,
                        &TimerCounterInst,
                           TMRCTR_DEVICE_ID,
                           TMRCTR_INTERRUPT_ID,
                           TIMER_CNTR_0);
   if (Status != XST_SUCCESS) {
         return XST_FAILURE;
   }
   return XST_SUCCESS;
}

intTmrCtrIntrExample(XIntc* IntcInstancePtr,   XTmrCtr* TmrCtrInstancePtr,
                u16DeviceId,   u16 IntrId,u8 TmrCtrNumber)
{
   int Status;
   Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
//初始化定时器数据结构，并清除定时器中断产生标志以及装载标志
   if (Status != XST_SUCCESS) {
         return XST_FAILURE;
   }
```

```
    Status = TmrCtrSetupIntrSystem(IntcInstancePtr,
                        TmrCtrInstancePtr,
                        DeviceId,
                        IntrId,
                        TmrCtrNumber);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    XTmrCtr_SetHandler(TmrCtrInstancePtr, TimerCounterHandler,
                        TmrCtrInstancePtr);//注册定时器中断服务程序
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
    XTC_INT_MODE_OPTION|XTC_AUTO_RELOAD_OPTION| XTC_DOWN_COUNT_OPTION);
  //设置定时器 0：允许中断、自动装载、减计数
    XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);
//设置定时器初始值到 TLR
    XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);
  //首先设置 LOAD 标志为 1，然后在清除 load 标志的同时，设置定时器使能标志 ENT=1
    while (1) {
    }//死循环等待时钟中断
    return XST_SUCCESS;
  }

  voidTimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber)
  {
    Xil_Out8(XPAR_SEG_0_BASEADDR,pos);     //输出位码
    Xil_Out8(XPAR_SEG_0_BASEADDR+0x8,segcode[i]);//输出段码
    pos=pos>>1;
    i++;
    if(i==4){
        i=0;
        pos=0xf7;
        }
  }

  staticintTmrCtrSetupIntrSystem(XIntc* IntcInstancePtr,
                    XTmrCtr* TmrCtrInstancePtr,
                    u16DeviceId,
                    u16IntrId,
                    u8TmrCtrNumber)
  {
    int Status;
    Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);//初始化中断控制器数据结构
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
```

```
    }
    Status = XIntc_Connect(IntcInstancePtr, IntrId,
                           (XInterruptHandler)XTmrCtr_InterruptHandler,
                           (void *)TmrCtrInstancePtr);
  //将定时器主中断服务程序注册到相应 Intr 引脚对应的中断向量
    if (Status != XST_SUCCESS) {
          return XST_FAILURE;
    }
    Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
  //设置中断控制器接受硬件中断，并发出中断请求
    if (Status != XST_SUCCESS) {
          return XST_FAILURE;
    }
    XIntc_Enable(IntcInstancePtr, IntrId);//使能定时器对应 Intr 的中断请求
    microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler, IntcInstancePtr);
  //将中断控制器的总中断服务程序注册到 0X0000 0010 地址处
    microblaze_enable_interrupts();
  //使能微处理器的中断控制位
    return XST_SUCCESS;
    }
```

9. 已知某按键的连接电路如图 7- 35 所示，试采用 AXI GPIO 并行接口中断方式读入按键状态，将其状态保存在字节变量 button 中。画出基于 MicroBLAZE 微处理器 AXI 总线的电路原理图并基于 standalone 提供的库函数编写控制程序。

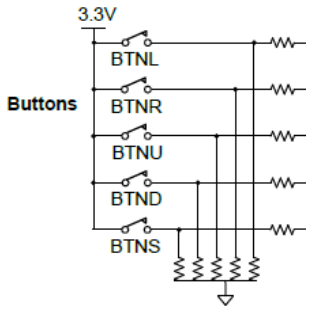
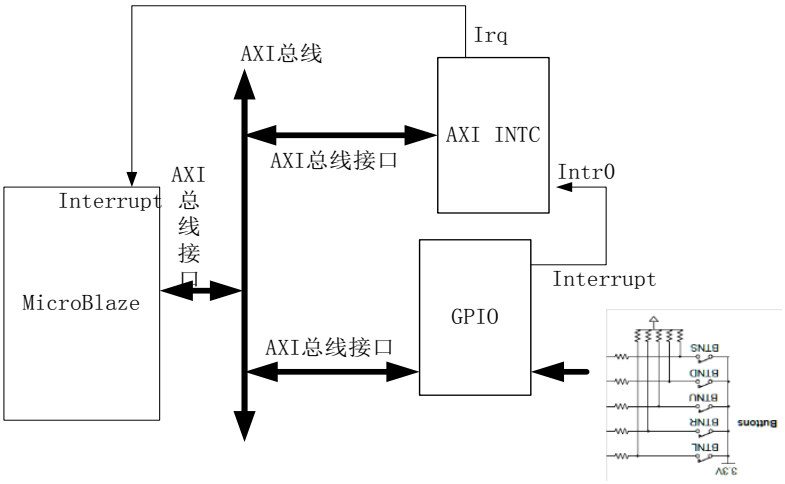


图 7- 2 按键电路连接

解答：

电路原理框图如下：



控制程序如下：

```
#include"xparameters.h"//The hardware configuration describing
constants
#include"xgpio.h" //GPIO API functions
#include"xintc.h"//Interrupt Controller API functions
#include"stdio.h"
voidInitialize();//初始化函数（包含中断初始化）
voidDelay_50ms();//延时函数
voidPushBtnHandler(void *CallBackRef);//按键的中断函数
XGpioBtns;//定义GPIO外设变量
XIntcintCtrl;//定义一个XINTC外设变量
intpshBtn;//作为中断标志
charbuffer;//存储Btns的状态值
intmain()
{
    Initialize();
    xil_printf("\r\nRunningGpioInputInterrupt!\r\n");
    while(1)
    {
        if(pshBtn)//若按下按键，则打印相关信息
        {
            xil_printf("Button Interrupt Triggered!!!the state is
0x%X\n\r",state1);
            pshBtn=0;
        }
    }
    return 0;
}
voidInitialize()
{
        //初始化Btns实例，并设定其为输入方式
```

```
    XGpio_Initialize(&Btns,XPAR_BUTTON_DEVICE_ID);
    XGpio_SetDataDirection (&Btns, 1, 0xff);
    //初始化intCtrl实例
    XIntc_Initialize(&intCtrl,XPAR_AXI_INTC_0_DEVICE_ID );
    //GPIO中断使能
    XGpio_InterruptEnable(&Btns, 1);
    XGpio_InterruptGlobalEnable(&Btns);
    //对中断控制器进行中断源使能
    XIntc_Enable(&intCtrl,XPAR_AXI_INTC_0_BUTTON_IP2INTC_IRPT_INTR);
    //注册中断服务函数
    XIntc_Connect(&intCtrl,XPAR_AXI_INTC_0_BUTTON_IP2INTC_IRPT_INTR,
           (XInterruptHandler)PushBtnHandler,(void *)0);
    microblaze_enable_interrupts();//允许处理器处理中断
    //注册中断控制器处理函数
microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler,
(void *)&intCtrl);
    XIntc_Start(&intCtrl, XIN_REAL_MODE);//启动中断控制器
}
voidDelay_50ms()
{
    inti;
    for(i=0;i<5000000;i++);
}
voidPushBtnHandler(void *CallBackRef)
{
    state1=XGpio_DiscreteRead(&Btns,1);//读取按键的状态值
    pshBtn=1;
    XGpio_InterruptDisable(&Btns, 1);//暂时禁止button中断
    Delay_50ms();//延时50ms，忽略按键弹起再次触发的中断
    XGpio_InterruptClear(&Btns,1);//清除中断标志位
    XGpio_InterruptEnable(&Btns, 1);//再次开放按键中断
}
```

10. 修改例 7.4 程序，编写控制程序控制 DAC 分别输出满量程三角波、方波，半量程三角波、方波。

解答：

三角波的数据首先连续增大然后再连续减少，方波固定为两个值，满量程则意味着最大值为 2^{12}-1，半量程则意味着最大值为 2^{11}.输出不同波形仅需要对例 7.4 程序的 while(1)循环体进行修改。

三角波需要在 main 函数里设定一个增减标志，假定为

char inc=1；1 表示上升边沿，0 表示下降边沿

满量程三角波的循环体如下：

```
while(1){
    WriteBuffer[0] = (u8)(Count);//SPI输出数据的低8位
```

```
    WriteBuffer[1] = (u8)(Count>>8)&0x0f;
// SPI输出数据的高8位，其中最高4位清0，使得Vout正常输出电压
    if（inc==1）{
        Count++;
        if (Count==4096){
            inc=0;
            Count--;//确保最大值2^12-1仅输出1次
            Count--;
            }
        }
        else {
            Count--;
            if (Count==0)
            inc=1;
        }
    TransferInProgress = TRUE;
    //  设置传输状态标志为1
    XSpi_Transfer(SpiInstancePtr, WriteBuffer, (void*)0, 2);
    //一次传输2个字节
    while (TransferInProgress);
    //等待传输结束
    }
```

半量程的三角波循环体如下：

```
while(1){
    WriteBuffer[0] = (u8)(Count);//SPI输出数据的低8位
    WriteBuffer[1] = (u8)(Count>>8)&0x0f;
// SPI输出数据的高8位，其中最高4位清0，使得Vout正常输出电压
    if（inc==1）{
        Count++;
        if (Count==2048){
            inc=0;确保最大值2^11输出1次
            }
        }
        else {
            Count--;
            if (Count==0)
            inc=1;
        }
    TransferInProgress = TRUE;
    //  设置传输状态标志为1
    XSpi_Transfer(SpiInstancePtr, WriteBuffer, (void*)0, 2);
    //一次传输2个字节
    while (TransferInProgress);
    //等待传输结束
```

```
    }
```

方波需要在循环体外首先初始化 Count 的值，假定为 Count=0；

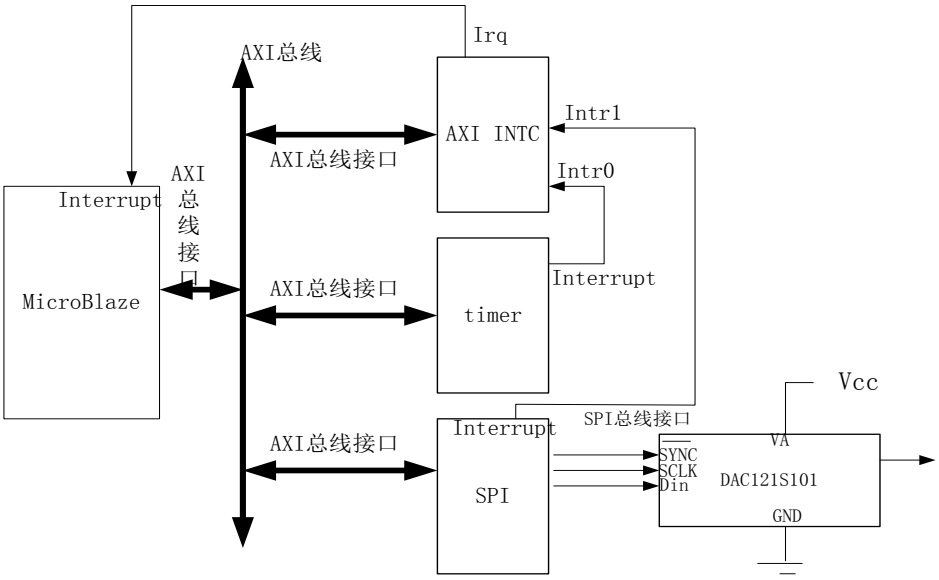
满量程方波的循环体：

```
while(1){
    WriteBuffer[0] = (u8)(Count);//SPI输出数据的低8位
    WriteBuffer[1] = (u8)(Count>>8)&0x0f;
// SPI输出数据的高8位，其中最高4位清0，使得Vout正常输出电压
    Count=Count^0x0fff;//半量程此处修改为Count=Count^0x0800;
    TransferInProgress = TRUE;
    //  设置传输状态标志为1
    XSpi_Transfer(SpiInstancePtr, WriteBuffer, (void*)0, 2);
    //一次传输2个字节
    while (TransferInProgress);
    //等待传输结束
    }
```

11. 利用定时器控制 DAC 输出周期约为 20ms 的锯齿波，画出基于 MicroBLAZE 微处理器 AXI 总线的电路原理图并基于 standalone 提供的库函数编写控制程序。

解答：锯齿波共 2^{12} 个不同的数据，那么每输出一个锯齿波的数据之间的时间间隔为 20ms/4K≈5us，假定定时器时钟频率为 100Mhz，采用减计数，则每计数 500 个时钟脉冲就需产生一次中断，因此定时器的初始值为 500.

电路原理图如下：



控制程序如下：

```
#include"xparameters.h"
#include"xtmrctr.h"
#include"xintc.h"
#include"xil_exception.h"
#include"xspi.h"
#define TMRCTR_DEVICE_ID XPAR_TMRCTR_0_DEVICE_ID//来自xparameters.h的宏
```

```
#define INTC_DEVICE_ID      XPAR_INTC_0_DEVICE_ID//来自xparameters.h的
宏
#define TMRCTR_INTERRUPT_ID XPAR_INTC_0_TMRCTR_0_VEC_ID//来自
xparameters.h的宏
#define TIMER_CNTR_0  0//定时器0
#define RESET_VALUE   500//计数500个时钟脉冲的初始值
#define SPI_DEVICE_ID       XPAR_SPI_0_DEVICE_ID
#define SPI_IRPT_INTR       XPAR_INTC_0_SPI_0_VEC_ID
#define BUFFER_SIZE          2
voidSpiIntrHandler(void *CallBackRef, u32StatusEvent, u32ByteCount);
//用户定义的SPI中断服务程序
voidTimerCounterHandler(void *CallBackRef, u8TmrCtrNumber);//定时器中断
服务程序
XIntcInterruptController;  /* 实例化中断控制器 */
XTmrCtrTimerCounterInst;   /* 实例化定时器 */
short Count;
staticXSpiSpiInstance;
volatileintTransferInProgress;
volatileintTimeOut;
//中断结束状态标志
int Error;
u8ReadBuffer[BUFFER_SIZE];
u8WriteBuffer[BUFFER_SIZE];

intmain(void)
{
    int Status;
Status = XTmrCtr_Initialize(&TimerCounterInst, TMRCTR_DEVICE_ID);
//初始化定时器数据结构，并清除定时器中断产生标志以及装载标志
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
XSpi_Config *ConfigPtr;
//  查找SPI接口配置项
    ConfigPtr = XSpi_LookupConfig(SPI_DEVICE_ID);
    if (ConfigPtr == NULL) {
        return XST_DEVICE_NOT_FOUND;
    }
//根据配置项参数，初始化SPI接口参数
    Status = XSpi_CfgInitialize(&SpiInstance, ConfigPtr,
              ConfigPtr->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
```

```
Status = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);//初始化中断控制器数据结构
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Status = XIntc_Connect(&InterruptController, TMRCTR_INTERRUPT_ID,
               (XInterruptHandler) XTmrCtr_InterruptHandler,
               (void *) &TimerCounterInst);
//将定时器主中断服务程序注册到相应Intr引脚对应的中断向量
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
Status = XIntc_Connect(&InterruptController, SPI_IRPT_INTR,
               (XInterruptHandler) XSpi_InterruptHandler,
               (void *) &SpiInstance);
//配置SPI中断输入引脚的中断处理函数
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
//设置中断控制器接受硬件中断，并发出中断请求
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
XIntc_Enable(&InterruptController, TMRCTR_INTERRUPT_ID);//使能定时器对应Intr的中断请求
XIntc_Enable(&InterruptController, SPI_IRPT_INTR);//是能SPI对应的中断请求
microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler,
&InterruptController);
//将中断控制器的总中断服务程序注册到0X0000 0010地址处
microblaze_enable_interrupts();
//使能微处理器的中断控制位

    XTmrCtr_SetHandler(&TimerCounterInst, TimerCounterHandler,
                  (void *) &TimerCounterInst);//注册定时器中断服务程序
    XTmrCtr_SetOptions(&TimerCounterInst, TIMER_CNTR_0,
    XTC_INT_MODE_OPTION|XTC_AUTO_RELOAD_OPTION|
XTC_DOWN_COUNT_OPTION);
//设置定时器0：允许中断、自动装载、减计数
    XTmrCtr_SetResetValue(&TimerCounterInst, TIMER_CNTR_0,
RESET_VALUE);
//设置定时器初始值到TLR
    XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0);
```

```
//首先设置LOAD标志为1，然后在清除load标志的同时，设置定时器使能标志ENT=1

//设置SPI接口用户中断服务函数
    XSpi_SetStatusHandler(&SpiInstance, &SpiInstance,
                (XSpi_StatusHandler)SpiIntrHandler);
//配置SPI接口工作模式
    Status = XSpi_SetOptions(&SpiInstance, XSP_MASTER_OPTION
|XSP_CLK_PHASE_1_OPTION);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
//设置从设备选择信号
    Status = XSpi_SetSlaveSelect(&SpiInstance, 1);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
//使能SPI接口
    XSpi_Start(&SpiInstance);
//循环输出数据到SPI接口控制DAC输出锯齿波
    while(1){
    WriteBuffer[0] = (u8)(Count);//SPI输出数据的低8位
    WriteBuffer[1] = (u8)(Count>>8)&0x0f;
// SPI输出数据的高8位，其中最高4位清0，使得Vout正常输出电压
    Count++;
    if (Count==4096)
        Count=0;
//12位DAC转换数据到达最大值时，恢复到0
    TransferInProgress = TRUE;
    TimeOut=0;
    //  设置传输状态标志为1
    XSpi_Transfer(&SpiInstance, WriteBuffer, (void*)0, 2);
//一次传输2个字节
    while (TransferInProgress||(!TimeOut));
//等待传输结束
    }
    return XST_SUCCESS;
}
voidTimerCounterHandler(void *CallBackRef, u8TmrCtrNumber)
{
    TimeOut=1;
}
voidSpiIntrHandler(void *CallBackRef, u32StatusEvent, u32ByteCount)
{
    TransferInProgress = FALSE;
```

```
//进入中断表示传输结束，修改传输状态标志为0
    if (StatusEvent != XST_SPI_TRANSFER_DONE) {
        Error++;
    }
  }
```