NV32F100x SPI 通讯接口编程

第一章　所有库函数简介

库函数列表

void SPI_Init(SPI_Type *pSPI, SPI_ConfigType *pConfig)
SPI 始化函数
void SPI_SetBaudRate(SPI_Type *pSPI,uint32_t u32BusClock,uint32_t u32Bps)
配置 SPI 时钟速率函数
ResultType SPI_TransferWait(SPI_Type *pSPI, SPI_WidthType* pRdBuff, SPI_WidthType *pWrBuff,uint32 uiLength)
SPI 发送和读取数据函数

寄存器操作的内联函数，调用内联函数和直接操作寄存器效率一样高
void SPI_Enable(SPI_Type *pSPI);
打开 SPI 使能
void SPI_Disable(SPI_Type *pSPI);
关闭 SPI 使能
void SPI_SetLSBFirst(SPI_Type *pSPI);
设置数据发送低位优先
void SPI_SetMSBFirst(SPI_Type *pSPI);
设置数据发送高位优先
void SPI_IntEnable(SPI_Type *pSPI);
打开 SPI 中断使能
void SPI_IntDisable(SPI_Type *pSPI);
关闭 SPI 中断使能
void SPI_SetMasterMode(SPI_Type *pSPI);
设置 SPI 为主机模式
void SPI_SetSlaveMode(SPI_Type *pSPI);
设置 SPI 为从机模式
void SPI_TxIntEnable(SPI_Type *pSPI);
打开发送中断使能
void SPI_TxIntDisable(SPI_Type *pSPI);
关闭发送中断使能
void SPI_SSOutputEnable(SPI_Type *pSPI);
打开从机输出使能
void SPI_SSOutputDisable(SPI_Type *pSPI);
关闭从机输出使能
void SPI_MatchIntEnable(SPI_Type *pSPI);
打开 SPI 匹配中断使能
void SPI_MatchIntDisable(SPI_Type *pSPI);

关闭 SPI 匹配中断使能
void SPI_ModfDisable(SPI_Type *pSPI);
打开 SPI 主机故障功能使能
void SPI_ModfEnable(SPI_Type *pSPI);
关闭 SPI 主机故障功能使能
void SPI_BidirOutEnable(SPI_Type *pSPI);
打开双向模式输出使能
void SPI_BidirOutDisable(SPI_Type *pSPI);
关闭双向模式输出使能
void SPI_ClockStopDisable(SPI_Type *pSPI);
配置 SPI 当 MCU 在等待模式下停止工作
void SPI_ClockStopEnable(SPI_Type *pSPI);
配置 SPI 当 MCU 在等待模式下继续工作
void SPI_BidirPinEnable(SPI_Type *pSPI);
配置 SPI 引脚为双向模式
void SPI_BidirPinDisable(SPI_Type *pSPI);
配置 SPI 引脚为单向模式
void SPI_SetClockPol(SPI_Type *pSPI,uint8_t u8PolLow);
设置 SPI 的 CPHA 值
void SPI_SetClockPhase(SPI_Type *pSPI,uint8_t u8Phase);
设置 SPI 的 CPOL 值
uint8_t SPI_IsSPRF(SPI_Type *pSPI);
返回 SPRF 位的值
uint8_t SPI_IsSPMF(SPI_Type *pSPI);
返回 SPMF 位的值
uint8_t SPI_IsSPTEF(SPI_Type *pSPI);
返回 SPTEF 位的值
uint8_t SPI_IsMODF(SPI_Type *pSPI);
返回 MODF 位的值
uint8_t SPI_ReadDataReg(SPI_Type *pSPI);
读取 SPI 接收的数据
void SPI_WriteDataReg(SPI_Type *pSPI, uint8_t u8WrBuff);
发送 SPI 数据
void SPI_WriteMatchValue(SPI_Type *pSPI, uint8_t u8WrBuff);
写入 SPI 数据匹配值
void SPI_SetCallback(SPI_Type *pSPI,SPI_CallbackType pfnCallback);
SPI 回调函数

1.1 SPI 模块初始化

对 SPIx_C1 配置，打开 SPI 使能，中断使能，配置 CPOL 和 CPHA 的值来决定 SPI 的模式

位	描述
7 SPIE	SPI 中断使能：对于 SPRF 和 MODF 对于 SPI 接收数据缓冲区满标志（SPRF）和模式错误标志（MODF）的中断使能。 0 从 SPRF 和 MODF 的中断被禁止——使用轮询 1 当 SPRF 和 MODF 为 1 时硬件中断请求
6 SPE	SPI 系统使能 启用 SPI 系统和将 SPI 端口引脚应用于 SPI 系统功能。如果 SPE 被清除，SPI 被禁止，被迫进入空闲状态，并且在 S 寄存器中的所有状态位都被复位。 0 SPI 系统被闲置 1 SPI 系统被启用
5 SPTIE	SPI 发送中断使能 这是一个发送数据缓存区为空的中断使能位。中断发生时，SPI 发送缓冲区为空（SPTEF 置 1）。 0 从 SPTEF 的中断被禁止（使用轮询） 1 当 SPTEF 为 1，有硬件中断请求
4 MSTR	主/从机选择 选择主机或从机模式 0 SPI 模块配置为从机模式 1 SPI 模块配置为主机模式
3 CPOL	时钟极性 选择一个倒置或不倒置 SPI 时钟。为了 SPI 模块之间的数据能相互传输 SPI 模块必须具有相同的 CPOL 值。 该位在主 SPI 设备或一个从 SPI 设备的串行时钟中有效放置了反相器。 0 高有效 SPI 时钟（空闲低） 1 低有效 SPI 时钟（空闲高）
2 CPHA	时钟相位 选择两种时钟格式给不同类型的同步串行外围设备。 0 SPSCK 第一边沿在数据传送的第一次循环的中间发生。 1 SPSCK 第一边沿在数据传送的第一次循环的起始发生。
1 SSOE	从机选择输出使能 此位与在 C2 寄存器的模式故障使能（MODFEN）字段和主/从（MSTR）控制位结合使用，以决定 SS 引脚的功能。 0 当 C2[MODFEN]为 0：在主机模式中，SS 引脚的功能是通用引脚。在从机模式中，SS 引脚是从机选择输入。当 C2[MODFEN]为 1：在主机模式中，SS 引脚的功能是模式错误的 SS 信号输出。在从机模式中，SS 引脚是从机选择输入。 1 当 C2[MODFEN]为 0：在主机模式中，SS 引脚的功能是通用引脚。在从机模式中，SS 引脚是从机选择输入。当 C2[MODFEN]为 1：在主机模式中，SS 引脚的功能是自动的 SS 信号输出。在从机模式中，SS 引脚是从机选择输入。
0 LSBFE	LSB 优先（移位方向） 该位不影响 MSB 和 LSB 在数据寄存器中的位置。数据寄存器的读取和写入总是有 MSB 的第 7 位。 0 最高有效位的串行数据传输开始 1 最低有效位的串行数据传输开始

函数名	SPI_Init
函数原形	SPI_Init(SPI_Type *pSPI, SPI_ConfigType *pConfig)
功能描述	以配置结构体 pConfig 来初始化 SPI
输入参数	配置结构体 SPI_ConfigType，模块结构体 SPI_Type
输出参数	无
返回值	无
先决条件	无
函数使用实例	先设置配置结构体，SPI_Init(SPI0, &SPI_Config);

```
/*****************************************************************************//*!
*
* @简介 打开 spi 使能，中断使能，配置 CPOL 和 CPHA 的值，决定高位先出还是低位先出
*
* @无返回
*****************************************************************************/

void SPI_Init(SPI_Type *pSPI, SPI_ConfigType *pConfig)
{
#if defined(CPU_NV32)
    ASSERT((pSPI == SPI0));
    SIM->SCGC |= SIM_SCGC_SPI0_MASK;
#else
    ASSERT((pSPI == SPI0) ||   (pSPI == SPI1));

    /* 打开 SPI 系统时钟 */
    if( pSPI == SPI0)
    {
        SIM->SCGC |= SIM_SCGC_SPI0_MASK;
    }
    else
    {
        SIM->SCGC |= SIM_SCGC_SPI1_MASK;
    }
#endif
    /* 配置其他控制位 */
    if( pConfig->sSettings.bIntEn)
    {
        SPI_IntEnable(pSPI);
#if defined(CPU_NV32)
        NVIC_EnableIRQ(SPI0_IRQn); //使能 SPI 的 IRQ 中断
#else
        if( pSPI == SPI0 )
        {
            NVIC_EnableIRQ(SPI0_IRQn);
        }
```

```
            else
            {
                NVIC_EnableIRQ(SPI1_IRQn);
            }
#endif
        }

        if( pConfig->sSettings.bTxIntEn)
        {
            SPI_TxIntEnable(pSPI);
#if defined(CPU_NV32M3)
            NVIC_EnableIRQ(SPI0_IRQn);
#else
            if( pSPI == SPI0 )
            {
                NVIC_EnableIRQ(SPI0_IRQn);
            }
            else
            {
                NVIC_EnableIRQ(SPI1_IRQn);
            }
#endif
        }
        if( pConfig->sSettings.bMasterMode)
        {
            SPI_SetMasterMode(pSPI);//主从机模式选择，1 为主机模式，0 为从机模式
        }
        else
        {
            SPI_SetSlaveMode(pSPI);
        }
        if( pConfig->sSettings.bClkPolarityLow)
        {
            SPI_SetClockPol(pSPI,1);//配置低有效 SPI 时钟（空闲高）
        }
        if( pConfig->sSettings.bClkPhase1)
        {
            SPI_SetClockPhase(pSPI,1);//SPSCK 在数据传送的第一次循环的起始发生
        }
        if( pConfig->sSettings.bShiftLSBFirst)
        {
            SPI_SetLSBFirst(pSPI);//配置高位先出还是低位先出
        }
        if( pConfig->sSettings.bMatchIntEn)
```

```
    {
        SPI_MatchIntEnable(pSPI);//使能 SPI 中断使能
    }
    if( pConfig->sSettings.bModeFaultEn)
    {
        SPI_ModfEnable(pSPI);//开启主机模式检测错误功能，对于从机模式无效
    }
    if( pConfig->sSettings.bMasterAutoDriveSS)
    {
        /* 当从机 SS 脚使能后将 SSOE 和 MODFEN 位置 1 */
        SPI_SSOutputEnable(pSPI);
        SPI_ModfEnable(pSPI);
    }
    if( pConfig->sSettings.bPinAsOuput)
    {
        SPI_BidirPinEnable(pSPI);//配置 SPI 为单线双向模式
    }
    if( pConfig->sSettings.bBidirectionModeEn)
    {
        SPI_BidirOutEnable(pSPI);//配置 SPI 的 I/O 引脚使能为输出
    }
    if( pConfig->sSettings.bStopInWaitMode)
    {
        SPI_ClockStopEnable(pSPI);//SPI 时钟在等待模式下关闭
    }
    if(pConfig->sSettings.bMasterMode)
    {
        SPI_SetBaudRate(pSPI,pConfig->u32BusClkHz,pConfig->u32BitRate);//SPI 的波特率设置
    }
    /* 开启 SPI 模式 */
    if( pConfig->sSettings.bModuleEn)
    {
        SPI_Enable(pSPI);
}
```

1.2 设置 SPI 时钟速率

使用该寄存器设置 SPI 主机的分频器和波特率因子。该寄存器可以在任何时间读出或写入。

波特率除数方程如下

BaudRateDivisor=（SPPR+ 1）×2^（SPR+ 1）

波特率可以用下面的公式计算：

波特率=总线时钟/ BaudRateDivisor

位	描述

7 保留	该位保留 该只读位保留并有且仅有值 0	
6-4 SPPR[2:0]	SPI 波特率分频因子 这 3 位字段选择八个因子为 SPI 波特率分频数之一。该分频器的输入是总线时钟速率（BUSCLK）。该分频器的输出驱动 SPI 波特率分频器的输入。	
	000	波特率分频因子为 1
	001	波特率分频因子为 2
	010	波特率分频因子为 3
	011	波特率分频因子为 4
	100	波特率分频因子为 5
	101	波特率分频因子为 6
	110	波特率分频因子为 7
	111	波特率分频因子为 8
3-0 SPR[3:0]	SPI 波特率因子 这 4 位字段选择九个除数之一为 SPI 波特率因子。	
	0000	波特率因子为 2
	0001	波特率因子为 4
	0010	波特率因子为 8
	0011	波特率因子为 16
	0100	波特率因子为 32
	0101	波特率因子为 64
	0110	波特率因子为 128
	0111	波特率因子为 256
	1000	波特率因子为 512
	其他	保留

函数名	SPI_SetBaudRate
函数原形	SPI_SetBaudRate(SPI_Type*pSPI,uint32_t u32BusClock,uint32_t u32Bps)
功能描述	根据公式计算对应寄存器需要传递的值并赋值
输入参数	SPI 速率，SPI 模块时钟，模块结构体 SPI_Type
输出参数	无
返回值	无
先决条件	无
函数使用实例	根据参数的值使用 SPI_SetBaudRate(SPI0,BusClock,BaudRate);

```
/*****************************************************************************//*!
*
* @简介  配置 SPI 时钟速率
*
* @无返回
```

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

void SPI_SetBaudRate(SPI_Type *pSPI,uint32_t u32BusClock,uint32_t u32Bps)
{
    uint32_t u32BitRateDivisor;
    uint8_t u8Sppr;
    uint8_t u8Spr;
    uint8_t u8ReadFlag;
    u32BitRateDivisor = u32BusClock/u32Bps; /* 计算速率分频 */

    u8ReadFlag = 0;
    /* 根据 spi 速率计算公式算出分频数 */
    for (u8Spr = 0; u8Spr <= 8; u8Spr++)
    {
        for(u8Sppr = 0; u8Sppr <= 7; u8Sppr++)
        {
            if((u32BitRateDivisor>>(u8Spr+1))<=(u8Sppr+1))
            {
                u8ReadFlag = 1;
                break;
            }
        }
        if(u8ReadFlag)
        {
            break;
        }
    }
    if(u8Sppr >=8)
    {
        u8Sppr = 7;
    }
    if(u8Spr >8)
    {
        u8Spr = 8;
    }
    /* 将计算好的分频系数传递给对应的寄存器 */
    pSPI->BR = SPI_BR_SPPR(u8Sppr) | SPI_BR_SPR(u8Spr);
}
```

1.3 发送和读取 SPI 数据

函数名	SPI_TransferWait
函数原形	SPI_TransferWait(SPI_Type *pSPI, SPI_WidthType* pRdBuff,

	SPI_WidthType *pWrBuff,uint32 uiLength)
功能描述	发送和读取 SPI 数据
输入参数	发送变量，接收变量，发送长度，模块结构体 SPI_Type
输出参数	读取的 SPI 数据
返回值	返回输入长度参数错误
先决条件	无
函数使用实例	输入组数据发送，放入空数组接收 SPI_TransferWait(SPI0, RdBuff[0], WrBuff[0],32)

```
/*****************************************************************************//*!
*
* @简介 发送 spi 数据.
*
* @参数 1   发送数据指针*pWrBuff
* @参数 2   接收数据指针*pRdBuff
*
* @返回  如果长度小于等于 0 返回 error
*****************************************************************************/

ResultType SPI_TransferWait(SPI_Type *pSPI, SPI_WidthType* pRdBuff, SPI_WidthType *pWrBuff,uint32 uiLength)
{
    ResultType err = SPI_ERR_SUCCESS;
    uint32_t   i;
    if(!uiLength)
    {
        return (err);
    }
    for(i = 0; i < uiLength; i++)
    {
        while(!SPI_IsSPTEF(pSPI));          //判断 SPTEF 位为 1 才可以发送数据
        SPI_WriteDataReg(pSPI,pWrBuff[i]);
        while(!SPI_IsSPRF(pSPI));           //判断接收数据区是否满
        pRdBuff[i] = SPI_ReadDataReg(pSPI);
    }
    return (err);
}
```

1.4 将 SPI 恢复到复位状态

函数名	SPI_DeInit
函数原形	SPI_DeInit(SPI_Type *pSPI)
功能描述	将 SPI 恢复到复位状态
输入参数	模块结构体 SPI_Type
输出参数	无

返回值	无
先决条件	无
函数使用实例	只需要在参数里定义是哪个 SPI 模块 ,SPI_DeInit(SPI_Type *pSPI);

```
/*****************************************************************************//*!
*
* @简介  初始化一个状态错误的 SPI
*
* @无返回
*****************************************************************************/

void SPI_DeInit(SPI_Type *pSPI)
{
    int16 i;
    pSPI->C1 = SPI_C1_DEFAULT;
    pSPI->C2 = SPI_C2_DEFAULT;
    pSPI->BR = SPI_BR_DEFAULT;
    pSPI->M  = SPI_M_DEFAULT;
    for(i = 0; i<100; i++);                    /* 等待循环至中断发生 */
}
```

第二章　样例程序

2.1 SPI 中断通信-主机

```
/*****************************************************************************
*
* @简介  本例程提供了一个 SPI 主机模式下中断通信的的方法，为用户提供了一个基本的 SPI 主机通信框
*        架。
*
*****************************************************************************/

#include "common.h"
#include "ics.h"
#include "rtc.h"
#include "uart.h"
#include "spi.h"
#include "spi_app.h"
#include "sysinit.h"
#define SPI0_TX_DATA_SIZE          128          //发送接收数据字节数
#define SPI_BIT_RATE               1000000      /* ~1Mbps */
static SPI_WidthType gu8SPI0_RxBuff[SPI0_TX_DATA_SIZE];
```

```
static SPI_WidthType gu8SPI0_TxBuff[SPI0_TX_DATA_SIZE];
static    uint32_t    gu32ErrorCount       = 0;
static    uint8_t     gu8Pattern           = 0;
static uint32_t      gu32Loop             = 0;
int main (void);
void RTC_Task(void);

int main (void)
{
    static uint32_t    i;
    SPI_ConfigType sSPIConfig = {0};
    sysinit();//系统初始化
    printf("\n Running the SPI_MasterInt_demo project.\n");
    LED0_Init();
    LED2_Init();
    UART_WaitTxComplete(TERM_PORT); //等待 UART1 口完成发送
    SPI_InitGlobalVariable();
    //使能 SPI0 系统时钟
    SIM->PINSEL |= SIM_PINSEL_SPI0PS_MASK;
    /*初始化 SPI0*/
    sSPIConfig.u32BitRate = SPI_BIT_RATE;        //配置时钟传输速率
    sSPIConfig.u32BusClkHz = BUS_CLK_HZ;    //配置 SPI 时钟为总线时钟
    sSPIConfig.sSettings.bModuleEn                = 1;//开启 spi 模块使能
    sSPIConfig.sSettings.bMasterMode              = 1;//设置为主机
    sSPIConfig.sSettings.bClkPhase1               = 0;//时钟相位为 0
    sSPIConfig.sSettings.bMasterAutoDriveSS       = 1;//配置片选信号 SS 脚是否为输入输出
    SPI_Init(SPI0, &sSPIConfig);//根据参数配置初始化 SPI0
    gu8Pattern = 0x55;
    /*  初始化数据  */
    for(i = 0; i < SPI0_TX_DATA_SIZE; i++)
    {
        gu8SPI0_TxBuff[i] = i+ gu8Pattern;
    }
    NVIC_EnableIRQ(SPI0_IRQn);//使能 SPI0 中断

    while(1)
    {
        /*  开始传送并接收数据  */
        SPI_Transfer(SPI0,gu8SPI0_RxBuff, gu8SPI0_TxBuff, SPI0_TX_DATA_SIZE);    /*  主机传递数据  */
        /*  一直等待直到传输完成  */
        while(!(SPI_GetTransferStatus(SPI0) & SPI_STATUS_RX_OVER) );
        SPI_ResetTransferStatus(SPI0);//复位 SPI0 的传输状态
        /*  检查收到的数据  */
        for(i = 0; i < SPI0_TX_DATA_SIZE; i++)
```

```
        {
            if(gu8SPI0_RxBuff[i] != gu8SPI0_TxBuff[i])
            {
                gu32ErrorCount++;
                RED_Init();
                break;
            }
        }
        printf("Error counter is %d\n",gu32ErrorCount);
        gu32Loop ++;
        printf("\n SPI communication counter %d\n",gu32Loop);
        for(i=0;i<0xffff;i++);
    }
}

void RTC_Task(void)
{
    /*反转 LED1 */
    LED0_Toggle();
}
```

2.2 SPI 通信-从机

```
/*****************************************************************************
*
* @简介 双机 SPI 通信连线：主从的 CLK 时钟与 SS 端对应连接，主机的 MOSI、MISO 端分别对应连接从
*       机的 MISO、MOSI 端
*
*****************************************************************************/
#include "common.h"
#include "ics.h"
#include "rtc.h"
#include "uart.h"
#include "spi.h"
#include "spi_app.h"
#include "sysinit.h"
#define SPI0_TX_DATA_SIZE          128          //字节数
#define SPI_BIT_RATE               1000000      /* ~1Mbps */

static SPI_WidthType gu8SPI0_RxBuff[SPI0_TX_DATA_SIZE];
static SPI_WidthType gu8SPI0_TxBuff[SPI0_TX_DATA_SIZE];
static   uint32_t   gu32ErrorCount     = 0;
static   uint8_t    gu8Pattern         = 0;
```

```
static uint32_t     gu32Loop = 0;
int main (void);

int main (void)
{
    uint32_t i;
    SPI_ConfigType sSPIConfig = {0};
    sysinit();
    printf("\nRunning the SPI_Slave_demo project.\n");
    LED0_Init();
    LED2_Init();

    UART_WaitTxComplete(TERM_PORT);

    SPI_InitGlobalVariable();
    SIM->PINSEL |= SIM_PINSEL_SPI0PS_MASK;

    /* 配置 SPI0 为从机模式      */
    sSPIConfig.u32BitRate = SPI_BIT_RATE;          //配置时钟传输速率
    sSPIConfig.u32BusClkHz = BUS_CLK_HZ;       //配置 SPI 时钟为总线时钟
    sSPIConfig.sSettings.bModuleEn                  = 1;//开启 spi 模块使能
    sSPIConfig.sSettings.bMasterMode                = 0;//关闭 SPI 主机模式
    sSPIConfig.sSettings.bClkPhase1                 = 0;//配置时钟相位
    sSPIConfig.sSettings.bMasterAutoDriveSS       = 1;//配置片选信号 SS 脚是否为输入输出
    SPI_Init(SPI0, &sSPIConfig);                //根据参数配置初始化 SPI0

    gu8Pattern = 0x55;
    /* 初始化数据 */
    for(i = 0; i < SPI0_TX_DATA_SIZE; i++)
    {
        gu8SPI0_TxBuff[i] = i+ gu8Pattern;
    }

    NVIC_EnableIRQ(SPI0_IRQn);        //打开 SPI0 中断向量
    while(1)
    {

        /* 开始发送并接收数据 */
        SPI_Transfer(SPI0,gu8SPI0_RxBuff, gu8SPI0_TxBuff, SPI0_TX_DATA_SIZE);   /* 主机发送 */

        /* 等待发送完成 */
        while(!(SPI_GetTransferStatus(SPI0) & SPI_STATUS_TX_OVER) );

        SPI_ResetTransferStatus(SPI0);
```

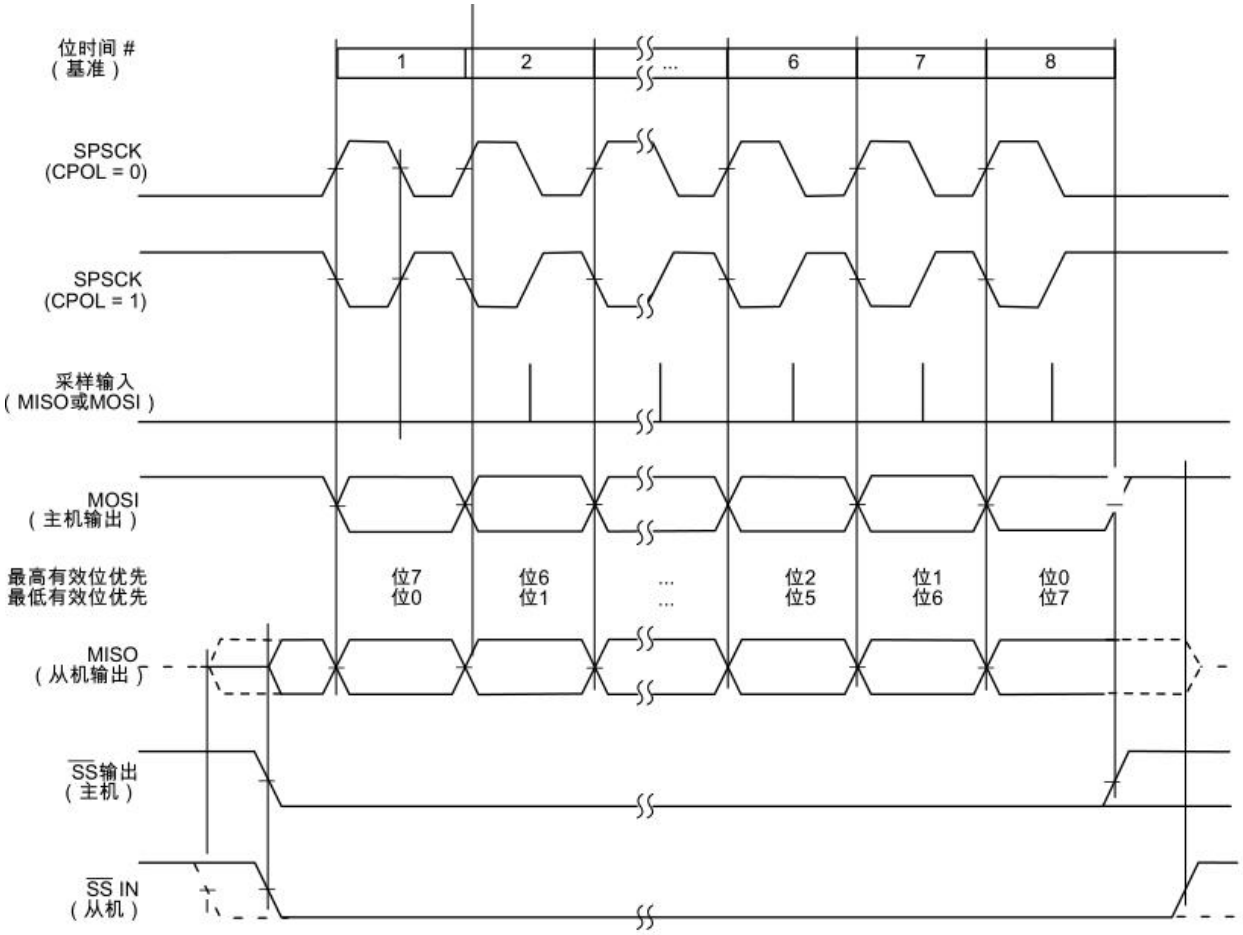
```
        /* 检查接收的数据 */
        for(i = 0; i < SPI0_TX_DATA_SIZE; i++)
        {
            if(gu8SPI0_RxBuff[i] != gu8SPI0_TxBuff[i])
            {
                gu32ErrorCount++;
                RED_Init();                         /* 初始化 LED */
                break;
            }
        }

        printf("Error counter is %d\n",gu32ErrorCount);

        gu32Loop ++;

        printf("SPI communication counter %d\n",gu32Loop);

        for(i=0;i<0xfff;i++);

    }

}

/*****************************************************************************//*!
*****************************************************************************/
void RTC_Task(void)
{
    /* toggle LED1 */
    LED0_Toggle();
}
/*****************************************************************************
*****************************************************************************/
```

第三章 SPI 时序分析

为了支持不同制造商的各种同步串行外设，SPI 系统的控制寄存器有一个时钟极性(CPOL)位和一个时钟相位(CPHA)控制位，用于选择四种时钟格式中的一种进行数据传输。C1[CPOL]可视情况选择插入一个与时钟串联的反相器。C1[CPHA]选择时钟与数据之间的两种不同时钟相位关系中的一种。

下图所示为 CPHA = 1 时的时钟格式。图顶部显示的是供参考的 8 位时间，其中，位 1 开始于第一个 SPSCK 边沿，位 8 结束于第八个 SPSCK 边沿后的半个 SPSCK 周期。最高有效位优先和最低有效位优先行显示 SPI 数据位的顺序，它取决于 LSBFE 的设置。图中显示了 SPSCK 极性的两种形态，但对于特定传输，只有一种

波形适用，具体取决于 C1[CPOL]中的值。采样输入波形适用于从机的 MOSI 输入或主机的 MISO 输入。MOSI 波形适用于主机的 MOSI 输出引脚， MISO 波形适用于从机的 MISO 输出。SS OUT 波形适用于主机的从机选择输出（前提是 C2[MODFEN]和 C1[SSOE] = 1） 。主机 SS 输出在传输开始前的半个 SPSCK 周期变为低电平有效，在传输的第八位时间结束时变回高电平。SS IN 波形适用于从机的从机选择输入。



当 CPHA=1

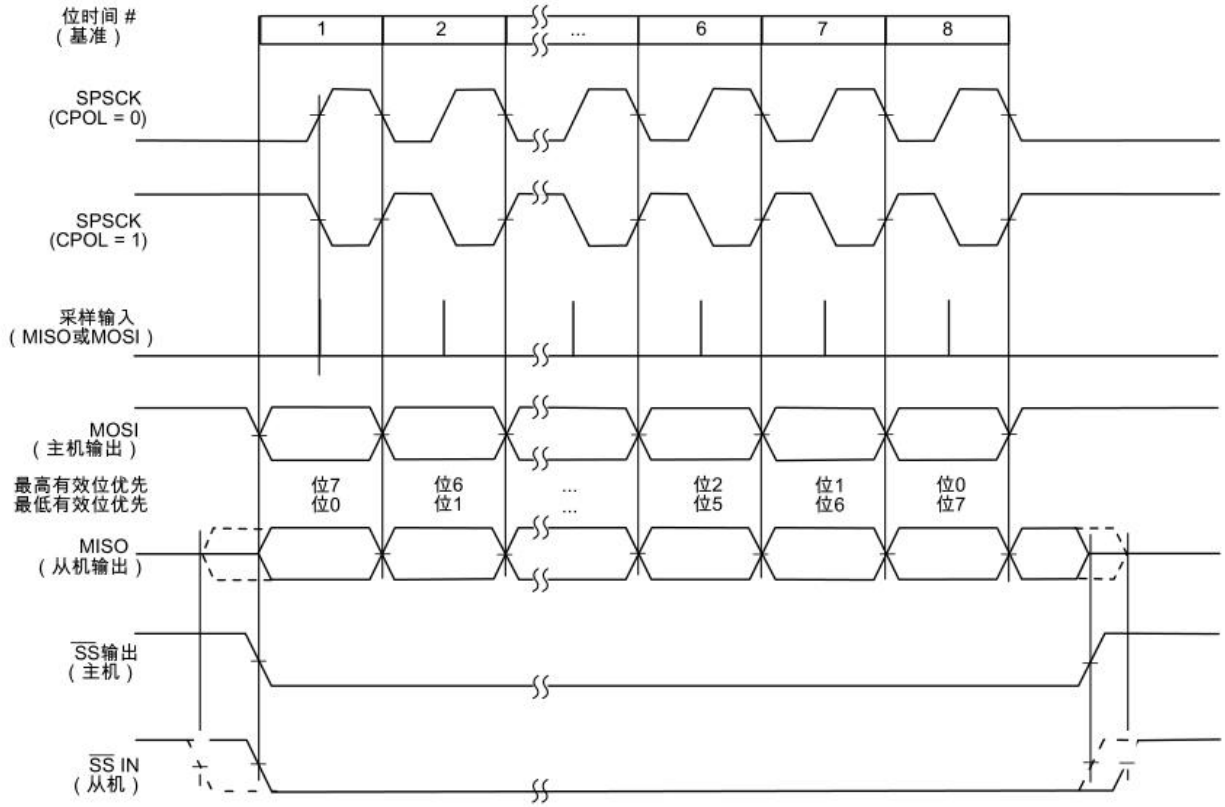
当 C1[CPHA] = 1 时，从机在 SS 变为低电平有效时开始驱动其 MISO 输出，但数据要等到第一个 SPSCK 边沿后才定义。第一个 SPSCK 边沿将数据的第一位从移位器移出到主机的 MOSI 输出和从机的 MISO 输出上。在下一个 SPSCK 边沿，主机和从机分别对 MISO 和 MOSI 输入上的数据位值进行采样。在第三个 SPSCK 边沿，SPI 移位器移动一个位位置，以便移入刚刚采样的位值，并将第二数据位值从移位器的另一端移出到主机的 MOSI 输出和从机的 MISO 输出。当 C1[CPHA] = 1 时，从机的 SS 输入在两次传输之间无需变为高电平无效状态。采用该时钟格式时，可能会发生背靠背传输，如下所述：

1. 一个传输正在进行。
2. 一个新数据字节在正在进行的传输完成之前被写入发送缓冲区。
3. 正在进行的传输完成后，新的已就绪数据会被立即发送。

在这两次连续传输之间，无需插入暂停；SS 引脚保持低电平。

下图所示为 C1[CPHA] = 0 时的时钟格式。图顶部显示的是供参考的 8 位时间，其中，位 1 在从机被选中时（SS 输入变为低电平）开始，位 8 结束于最后一个 SPSCK 边沿。最高有效位优先和最低有效位优先行显示 SPI 数据位的顺序，它取决于 LSBFE 的设置。图中显示了 SPSCK 极性的两种形态，但对于特定传

输，只有一种波形适用，具体取决于 CPOL 中的值。采样输入波形适用于从机的 MOSI 输入或主机的 MISO 输入。MOSI 波形适用于主机的 MOSI 输出引脚，MISO 波形适用于从机的 MISO 输出。SS OUT 波形适用于主机的从机选择输出（前提是 C2[MODFEN]和 C1[SSOE] = 1） 。主机 SS 输出在传输的第一位时间开始时变为低电平有效， 在传输的第八位时间结束后的半个 SPSCK 周期变回高电平。SS IN 波形适用于从机的从机选择输入。



当 CPHA=0