

32 位微控制器

HC32F4A0 系列的串行外设接口 SPI

本产品支持芯片系列如下

F 系列	HC32F4A0
------	----------

目 录

1	摘要	3
2	HC32F4A0 系列的 SPI 简介	3
2.1	系统框图	3
2.2	主要特性	4
3	SPI 模式配置	5
3.1	管脚说明	5
3.2	通信方式	7
3.3	数据格式	8
3.4	数据传送格式	9
4	SPI 应用代码	12
4.1	代码介绍	12
4.2	工作流程	17
4.3	代码运行	18
5	总结	19
6	版本信息 & 联系方式	20

1 摘要

本篇应用笔记主要介绍 HC32F4A0 串行外设接口(Serial Peripheral Interface, SPI)模块的基本功能、各种模式应用配置以及应用样例示范。

2 HC32F4A0 系列的 SPI 简介

本系列的 MCU 搭载通用串行接口 (SPI) 6 个单元, 支持高速全双工串行同步传输, 用户可根据需要进行三线/四线, 主机/从机, 波特率及数据通信格式的配置。

2.1 系统框图

本系列芯片 SPI 外设系统框图如图所示。

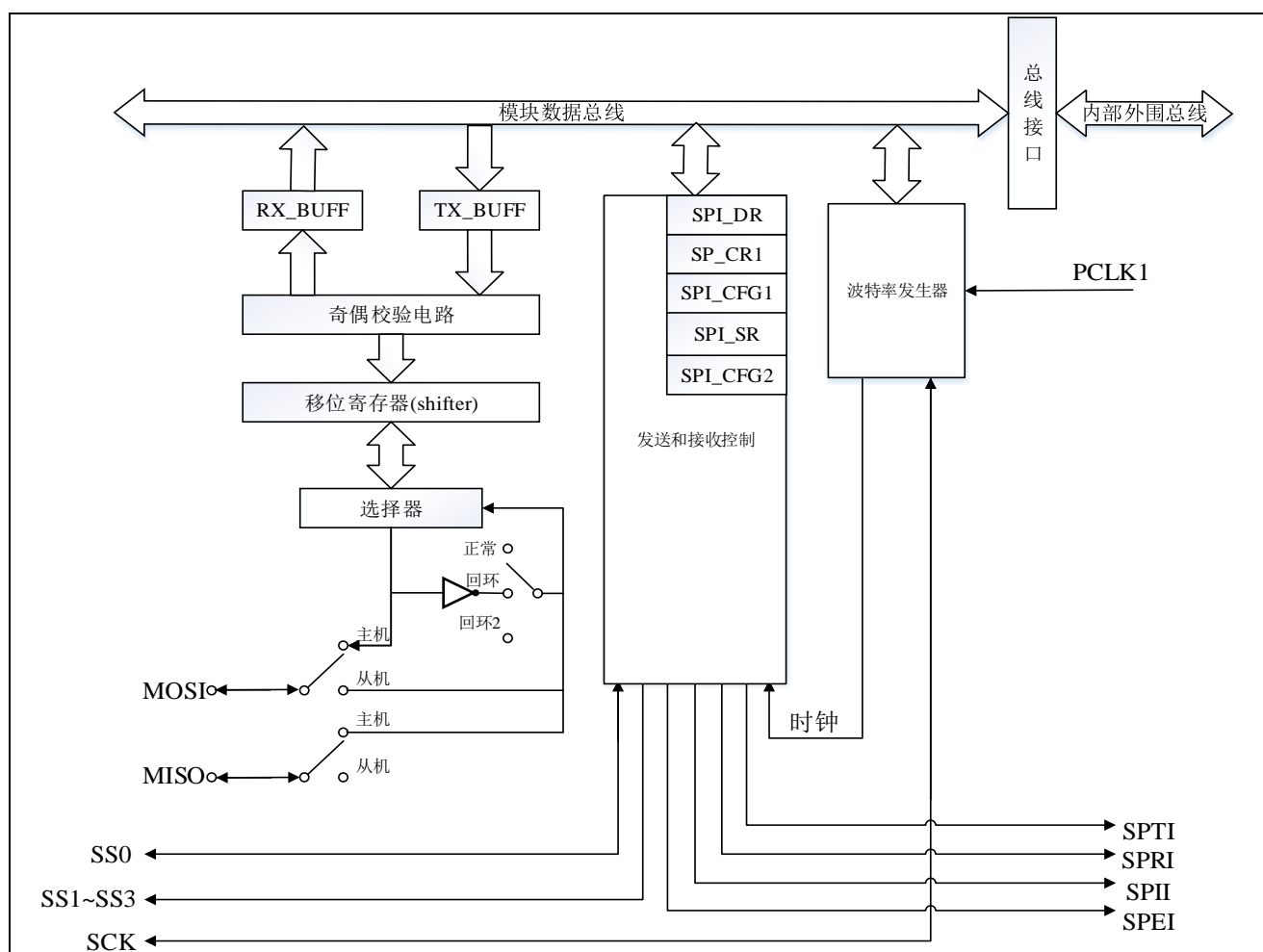


图 1 系统框图

2.2 主要特性

每个单元的 SPI 主要特性如下表所示。

要点	描述
通道数	<ul style="list-style-type: none"> 1通道
串行通信功能	<ul style="list-style-type: none"> 支持4线式SPI模式和3线式时钟同步运行模式 支持全双工和只发送两种通信方式 可调整通信时钟SCK的极性和相位
数据格式	<ul style="list-style-type: none"> 可选择数据移位顺序:MSB开始/LSB开始 可选择数据宽度: 4/5/6/7/8/9/10/11/12/13/14/15/16/20/24/32位 单次最多可传送或接收4帧宽度为32位的数据
波特率	<ul style="list-style-type: none"> 主机模式下可通过内置专用波特率发生器对波特率进行调整, 波特率范围为PCLK1的2分频~256分频 从机模式下允许的最大波特率为PCLK1的6分频
错误监测	<ul style="list-style-type: none"> 模式故障错误监测 数据过载错误监测 数据欠载错误监测 奇偶校验错误监测
片选信号控制	<ul style="list-style-type: none"> 每个通道配置四根片选信号线 可对片选信号和通信时钟的相对时序关系进行调整 可对连续两次通信之间的片选信号无效时间进行调整 极性可调
主机模式下的传输控制	<ul style="list-style-type: none"> 通过将数据写入数据寄存器启动传输 通信自动挂起功能
中断/AOS源	<ul style="list-style-type: none"> 接收数据区域已满 发送数据区域已空 SPI错误 (模式/过载/欠载/奇偶校验) SPI闲置 传输完成
低功耗控制	<ul style="list-style-type: none"> 可设置模块停止
其他功能	<ul style="list-style-type: none"> SPI初始化功能

3 SPI 模式配置

本系列芯片 SPI 外设的三线/四线模式及时钟极性相位可以配置，支持主机/从机、全双工/半双工，传送数据格式可灵活配置，并且有发送空接收满 SPI 错误等中断事件功能配合应用使用，更多功能详见本系列芯片手册的相关章节。

本章节重点讲解常用的 SPI 外设可配置的基本特性，如下：

串行通信模式	四线（SPI 模式）/三线（时钟同步模式） 时钟极性（Clock Polarity）可配置 时钟相位（Clock Phase）可配置
数据格式	MSB/LSB 数据位宽 4/5/6/7/8/9/10/11/12/13/14/15/16/20/24/32 位 单次传输最多 4 帧宽度为 32 位的数据

3.1 管脚说明

本系列芯片 SPI 通信硬件连接相关 IO 如下：

管脚名	端口方向		功能
	主机	从机	
SCK	输出	输入	通信时钟
MOSI	输出	输入	主机数据输出、从机数据输入
MISO	输入	输出	主机数据输入、从机数据输出
SS0	输出	输入	片选信号
SS1~SS3	输出	无效	片选信号，仅主机有效

主机通信配置为四线 SPI 模式时可以通过寄存器 SPI_CFG2.SSA 位选择 SS0~SS3 其中一个信号作为从机片选，并且可以通过寄存器 SPI_CFG1.SS0PV~SS3PV 分别配置片选信号的有效电平。默认配置为选择信号线 SS0，并且片选信号低电平有效。从机通信配置为四线 SPI 模式时，SS0 信号有效。

SPI 工作在主机模式下时，各个管脚状态如下：

模式		管脚名	管脚状态 (PCRxy.NOD=0)	管脚状态 (PCRxy.NOD=1)
四线 SPI 模式 (SPIMDS=0)	主机模式 (MSTR=1、 MODFE=0)	SCK	CMOS 输出	OD 输出
		SS0~SS3	CMOS 输出	OD 输出
		MOSI	CMOS 输出	OD 输出
		MISO	输入	输入
三线时钟同步模式 (SPIMDS=1)	主机模式 (MSTR=1)	SCK	CMOS 输出	OD 输出
		SS0~SS3 (不使用)	Hi-Z (可作为通用 I/O)	Hi-Z (可作为通用 I/O)
		MOSI	CMOS 输出	OD 输出
		MISO	输入	输入

SPI 工作在从机模式下时，各个管脚状态如下：

模式		管脚名	管脚状态 (PCRxy.NOD=0)	管脚状态 (PCRxy.NOD=1)
四线 SPI 模式 (SPIMDS=0)	从机模式 (MSTR=0、 MODFE=0)	SCK	输入	输入
		SS0	输入	输入
		SS1~SS3 (不使用)	Hi-Z (可作为通用 I/O)	Hi-Z (可作为通用 I/O)
		MOSI	输入	输入
		MISO	CMOS 输出/Hi-Z	OD 输出/Hi-Z
三线时钟同步模式 (SPIMDS=1)	从机模式 (MSTR=0)	SCK	输入	输入
		SS0~SS3 (不使用)	Hi-Z (可作为通用 I/O)	Hi-Z (可作为通用 I/O)
		MOSI	输入	输入
		MISO	CMOS 输出	OD 输出

注意：

- 无论是主机或是从机模式，管脚输入类型请设定为 CMOS 输入，输出设定为高驱动力模式，设定方法请参照本系列芯片手册【通用 IO (GPIO)】章节通用控制寄存器 PCRxy。

3.2 通信方式

通过寄存器 SPI_CR1.SPIMDS 位可以配置 SPI 通信方式为 4 线 SPI 模式或 3 线时钟同步模式。



图 2 四线 SPI 模式



图 3 三线时钟同步模式

3.3 数据格式

1) 数据帧格式

通过寄存器 `SPI_CFG2.DSIZE` 配置 SPI 数据帧长度为 4~16、20、24、32 位，寄存器 `SPI_CFG2.LSBF` 配置数据格式为 MSB 或 LSB，寄存器 `SPI_CR1.PAE` 配置数据奇偶校验是否使能。

当数据帧长度配置为 32 位时各种数据格式时序如下：

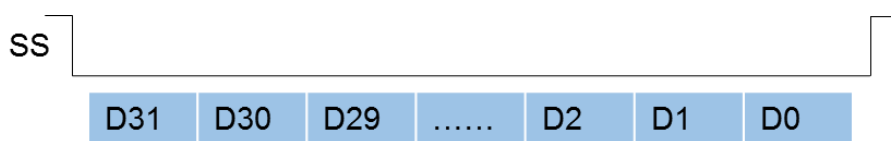


图 4 MSB 奇偶校验不使能



图 5 LSB 奇偶校验不使能

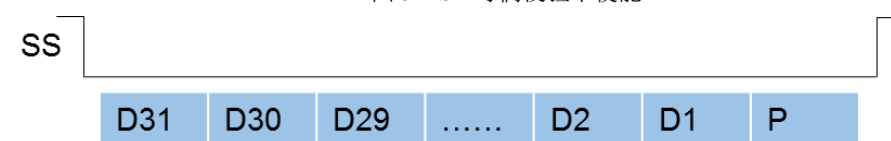


图 6 MSB 奇偶校验使能

注：P 为 D1~D31 计算出的奇偶校验位

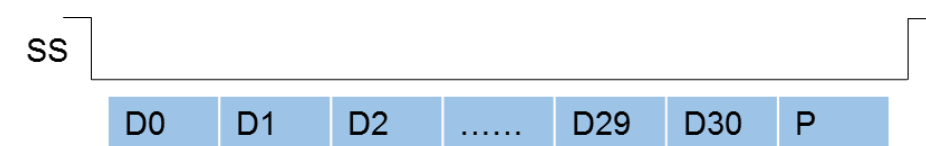


图 7 LSB 奇偶校验使能

注：P 为 D0~D30 计算出的奇偶校验位

2) 数据传送帧数

通过寄存器 `SPI_CFG1.FTHLV` 可配置数据传送帧数分别为 1 帧~4 帧，即启动一次数据传送的数据帧长度。当作为主机发送或接收数据时，向 `DTR` 寄存器写完所配置数据帧数目后才能启动一次数据传送；当作为从机接收数据时，收到了所配置数据帧数目的数据后才能产生 `RDFF`（接收缓冲器满）标志。

3.4 数据传送格式

1) SPI 的模式定义

SPI 外设通过 SPI_CFG2.CPHA 配置时钟相位（Clock Phase）和 SPI_CFG2.CPOL（Clock Polarity）配置时钟极性。因为各个厂家对 SPI 模式的定义没有统一的标准，下表是应用领域对 SPI 模式的一种比较通用的定义，如下：

SPI 模式	时钟相位	时钟极性
SPI 模式 (0, 0) CPHA=0、CPOL=0	奇数边沿数据采样、偶数边沿数据更新	空闲时 SCK 为低电平
SPI 模式 (0, 1) CPHA=0、CPOL=1	奇数边沿数据采样、偶数边沿数据更新	空闲时 SCK 为高电平
SPI 模式 (1, 0) CPHA=1、CPOL=0	奇数边沿数据更新、偶数边沿数据采样	空闲时 SCK 为低电平
SPI 模式 (1, 1) CPHA=1、CPOL=1	奇数边沿数据更新、偶数边沿数据采样	空闲时 SCK 为高电平

2) SPI 模式时序

当 CPHA=0 的时序图如下，当 SS 有效后数据发送方准备数据，奇数个时钟边沿采集数据，偶数个时钟边沿准备下一位数据。按此时序规则在三线模式下作为从机时，在第一个时钟边沿之前没有一个可以触发从机准备数据的时机或动作。因此三线时钟同步模式的从机无法支持 SPI 模式 (0, 0) 和 SPI 模式 (0, 1)。

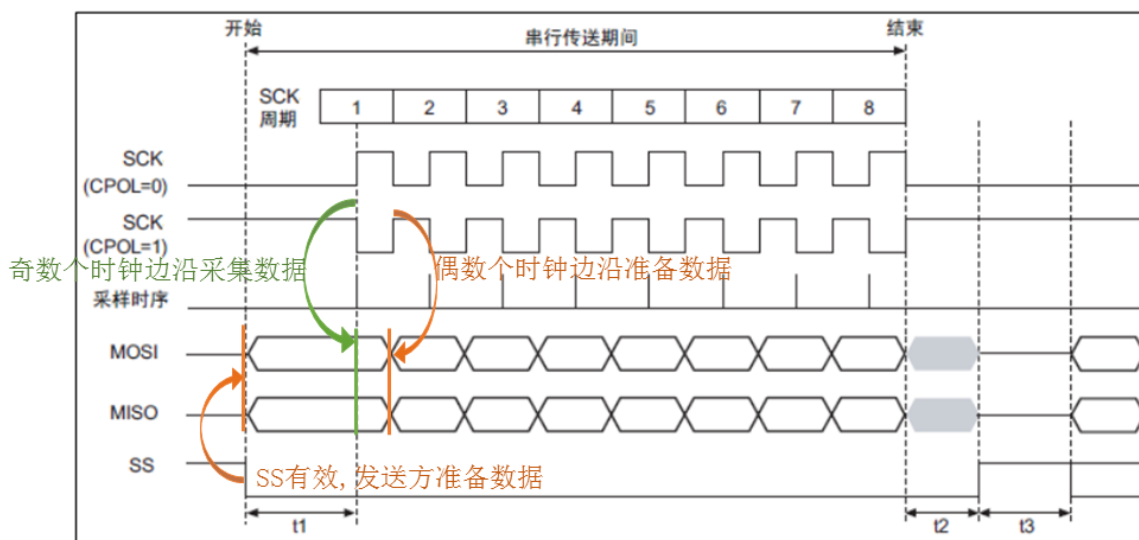


图 8 CPHA=0 SPI 数据传送时序

当 CPHA=1 的时序图如下，奇数个时钟边沿准备数据，偶数个时钟边沿采集数据。

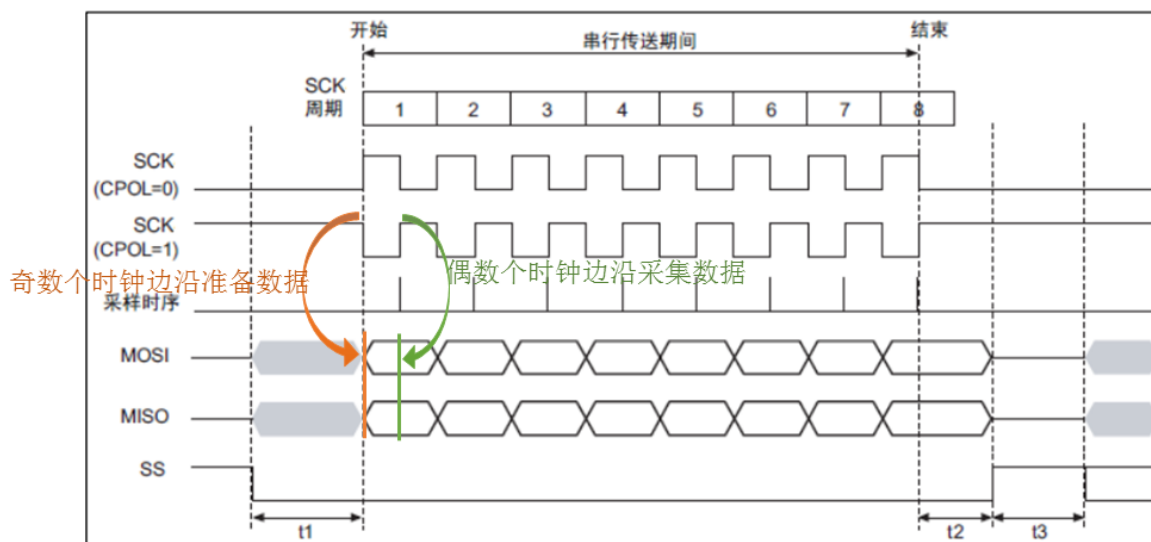


图 9 CPHA=1 SPI 数据传送时序

由图 8 和图 9 的时序图看出，CPOL 对 SCK 极性的配置可以根据应用需求任意选择。

3) SPI 时序时间配置

图 8 和图 9 时序图中时间 t1、t2、t3 由主机寄存器配置决定，定义如下：

标号	定义	描述	范围
t1	SCK 延迟时间	SS 信号有效到 SCK 振荡的间隔时间，由寄存器 MSSIE 设定，寄存器 MSSIE 使能。	1 个 SCK ~ 8 个 SCK
t2	SS 无效延迟时间	SCK 振荡停止到 SS 信号变成无效的间隔时间，由寄存器 MSSDL 设定，寄存器 MSSDLE 使能。	1 个 SCK ~ 8 个 SCK
t3	下次存取延迟时间	串行传送结束后到下一次传送开始的最小等待时间，由寄存器 MIDI 设定，寄存器 MIDIE 使能。	1 个 SCK+2 个 PCLK1 ~ 8 个 SCK+2 个 PCLK1

以 SPI 模式 (0, 0) 为例，本系列芯片 SPI 连续发送或接收数据的时序如下图所示，连续数据传送时，每帧数据之间的最小间隔为 $t1+t2+t3$ 等于 3 个 SCK+2 个 PCLK1，并且在三线时钟同步模式的数据间隔也遵循此公式。

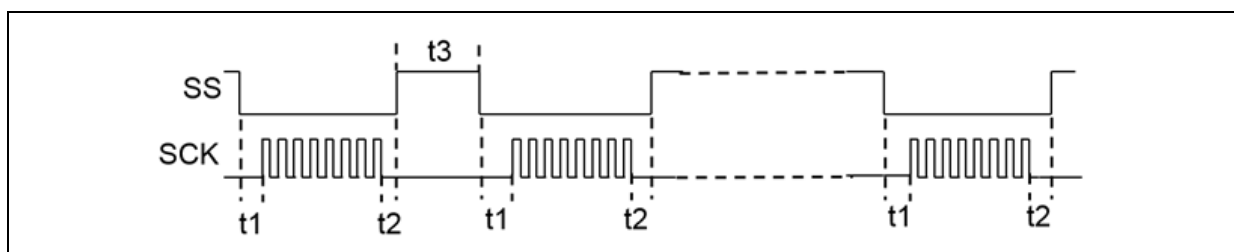


图 10 SPI 连续传送数据时序

如图 10 所示本系列芯片配置为 4 线 SPI 模式时的 SS 在每一帧的数据传送时都会有一次拉低拉高（当 SS 低电平有效时）的有效过程，所以在读写某些 SPI 接口的设备时，不能配置为 4 线 SPI 模式，可以通过配置为 SPI 三线时钟同步模式并且采用 GPIO 来模拟 SS 片选功能实现。

4 SPI 应用代码

用户可以通过华大半导体的网站下载到设备驱动库（Device Driver Library, DDL）的样例代码并使用其中的 SPI 的样例进行 SPI 外设功能学习及验证。

4.1 代码介绍

以下部分简要介绍样例 SPI flash 读写（spi_write_read_flash）样例代码所涉及的配置和流程。SPI 外设作为主机对 SPI flash 读写操作时，只能配置为 3 线时钟同步模式，并且采用 GPIO 模拟 SS 片选信号进行通信。

1) 系统时钟初始化

```
BSP_CLK_Init();
```

2) SPI 通信 IO 初始化

```
/* Port configurate */
(void)GPIO_StructInit(&stcGpioCfg);

/* High driving capacity for output pin. */
stcGpioCfg.u16PinDir = PIN_DIR_OUT;
stcGpioCfg.u16PinDrv = PIN_DRV_HIGH;
stcGpioCfg.u16PinState = PIN_STATE_SET;
(void)GPIO_Init(SPI_NSS_PORT, SPI_NSS_PIN, &stcGpioCfg);

(void)GPIO_StructInit(&stcGpioCfg);
stcGpioCfg.u16PinDrv = PIN_DRV_HIGH;
(void)GPIO_Init(SPI_SCK_PORT, SPI_SCK_PIN, &stcGpioCfg);
(void)GPIO_Init(SPI_MOSI_PORT, SPI_MOSI_PIN, &stcGpioCfg);

/* CMOS input for input pin */
stcGpioCfg.u16PinDrv = PIN_DRV_LOW;
stcGpioCfg.u16PinIType = PIN_ITYPE_CMOS;
(void)GPIO_Init(SPI_MISO_PORT, SPI_MISO_PIN, &stcGpioCfg);

/* Configure SPI Port function for master */
GPIO_SetFunc(SPI_SCK_PORT, SPI_SCK_PIN, SPI_SCK_FUNC, PIN_SUBFUNC_DISABLE);
GPIO_SetFunc(SPI_MOSI_PORT, SPI_MOSI_PIN, SPI_MOSI_FUNC, PIN_SUBFUNC_DISABLE);
GPIO_SetFunc(SPI_MISO_PORT, SPI_MISO_PIN, SPI_MISO_FUNC, PIN_SUBFUNC_DISABLE);
```

3) SPI 初始化函数

```
/**
 * @brief Configure SPI peripheral function
 *
 * @param [in] None
 */
```

```

* @retval None
*/
static void Spi_Config(void)
{
    stc_spi_init_t stcSpiInit;
    stc_spi_delay_t stcSpiDelayCfg;

    /* Clear initialize structure */
    (void)SPI_StructInit(&stcSpiInit);
    (void)SPI_DelayStructInit(&stcSpiDelayCfg);

    /* Configure peripheral clock */
    PWC_Fcg1PeriphClockCmd(SPI_UNIT_CLOCK, Enable);

    /* SPI De-initialize */
    SPI_DeInit(SPI_UNIT);

    /* Configuration SPI structure */
    stcSpiInit.u32WireMode      = SPI_WIRE_3;
    stcSpiInit.u32TransMode    = SPI_FULL_DUPLEX;
    stcSpiInit.u32MasterSlave   = SPI_MASTER;
    stcSpiInit.u32SuspMode     = SPI_COM_SUSP_FUNC_OFF;
    stcSpiInit.u32Modfe        = SPI_MODFE_DISABLE;
    stcSpiInit.u32Parity       = SPI_PARITY_INVALID;
    stcSpiInit.u32SpiMode      = SPI_MODE_0;
    stcSpiInit.u32BaudRatePrescaler = SPI_BR_PCLK1_DIV256;
    stcSpiInit.u32DataBits     = SPI_DATA_SIZE_8BIT;
    stcSpiInit.u32FirstBit     = SPI_FIRST_MSB;
    (void)SPI_Init(SPI_UNIT, &stcSpiInit);

    stcSpiDelayCfg.u32IntervalDelay = SPI_INTERVAL_TIME_8SCK_2PCLK1;
    stcSpiDelayCfg.u32ReleaseDelay = SPI_RELEASE_TIME_8SCK;
    stcSpiDelayCfg.u32SetupDelay = SPI_SETUP_TIME_1SCK;
    (void)SPI_DelayTimeCfg(SPI_UNIT, &stcSpiDelayCfg);

    SPI_FunctionCmd(SPI_UNIT, Enable);
}

```

4) SPI flash 读 ID 函数

```

/**
 * @brief SPI flash read ID for test
 *
 * @param [in] None
 *
 * @retval uint8_t          Flash ID
 */
static uint8_t SpiFlash_ReadID(void)
{
    uint8_t u8IdRead;
    SPI_NSS_LOW();
    (void)SpiFlash_WriteReadByte(FLASH_READ_MANUFACTURER_ID);
    (void)SpiFlash_WriteReadByte((uint8_t)0x00U);
    (void)SpiFlash_WriteReadByte((uint8_t)0x00U);
    (void)SpiFlash_WriteReadByte((uint8_t)0x00U);
    u8IdRead = SpiFlash_WriteReadByte(FLASH_DUMMY_BYTE_VALUE);
}

```

```
SPI_NSS_HIGH();
return u8IdRead;
}
```

5) SPI flash 擦 4K 扇区函数

```
/**
 * @brief SPI flash erase 4Kb sector function
 *
 * @param [in] u32Addr          Valid flash address
 *
 * @retval Error                Sector erase failed
 * @retval Ok                   Sector erase success
 */
static en_result_t SpiFlash_Erase4KbSector(uint32_t u32Addr)
{
    en_result_t enRet;

    if (u32Addr >= FLASH_MAX_ADDR)
    {
        enRet = Error;
    }
    else
    {
        SpiFlash_WriteEnable();
        /* Send instruction to flash */
        SPI_NSS_LOW();
        (void)SpiFlash_WriteReadByte(FLASH_INSTR_ERASE_4KB_SECTOR);
        (void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF0000UL) >> 16U));
        (void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF00U) >> 8U));
        (void)SpiFlash_WriteReadByte((uint8_t)(u32Addr & 0xFFU));
        //SPI_GetStatus(const M4_SPI_TypeDef *SPIdx, uint32_t u32Flag) //todo
        SPI_NSS_HIGH();
        /* Wait for flash idle */
        enRet = SpiFlash_WaitForWriteEnd();
    }

    return enRet;
}
```

6) SPI flash 写 (Page write) 函数

```
/**
 * @brief SPI flash page write program function
 *
 * @param [in] u32Addr          Valid flash address
 * @param [in] pData            Pointer to send data buffer
 * @param [in] len              Send data length
 *
 * @retval Error                Page write program failed
 * @retval Ok                   Page write program success
 */
static en_result_t SpiFlash_WritePage(uint32_t u32Addr, const uint8_t pData[], uint16_t len)
{

```

```

en_result_t enRet;
uint16_t u16Index = 0U;

if ((u32Addr > FLASH_MAX_ADDR) || (NULL == pData) || (len > FLASH_PAGE_SIZE))
{
    enRet = Error;
}
else
{
    SpiFlash_WriteEnable();
    /* Send data to flash */
    SPI_NSS_LOW();
    (void)SpiFlash_WriteReadByte(FLASH_INSTR_PAGE_PROGRAM);
    (void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF0000UL) >> 16U));
    (void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF00U) >> 8U));
    (void)SpiFlash_WriteReadByte((uint8_t)(u32Addr & 0xFFU));
    while (0U != (len--))
    {
        (void)SpiFlash_WriteReadByte(pData[u16Index]);
        u16Index++;
    }
    SPI_NSS_HIGH();
    /* Wait for flash idle */
    enRet = SpiFlash_WaitForWriteEnd();
}

return enRet;
}

```

7) SPI flash 读函数

```

/**
 * @brief SPI flash read data function
 *
 * @param [in] u32Addr      Valid flash address
 * @param [out] pData       Pointer to receive data buffer
 *
 * @param [in] len          Read data length
 *
 * @retval Error            Read data program failed
 * @retval Ok               Read data program success
 */
static en_result_t SpiFlash_ReadData(uint32_t u32Addr, uint8_t pData[], uint16_t len)
{
    en_result_t enRet = Ok;
    uint16_t u16Index = 0U;

    if ((u32Addr > FLASH_MAX_ADDR) || (NULL == pData))
    {
        enRet = Error;
    }
    else
    {
        SpiFlash_WriteEnable();
        /* Send data to flash */
        SPI_NSS_LOW();

```

```

(void)SpiFlash_WriteReadByte(FLASH_INSTR_STANDARD_READ);
(void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF0000UL) >> 16U));
(void)SpiFlash_WriteReadByte((uint8_t)((u32Addr & 0xFF00U) >> 8U));
(void)SpiFlash_WriteReadByte((uint8_t)(u32Addr & 0xFFU));
while (0U != (len--))
{
    pData[u16Index] = SpiFlash_WriteReadByte(FLASH_DUMMY_BYTE_VALUE);
    u16Index++;
}
SPI_NSS_HIGH();
}

return enRet;
}

```

8) 样例主循环

```

for(;;)
{
    while(Pin_Set == GPIO_ReadInputPins(GPIO_PORT_A, GPIO_PIN_00))
    {
        /* Wait */
    }
    DDL_DelayMS(10UL);
    while(Pin_Reset == GPIO_ReadInputPins(GPIO_PORT_A, GPIO_PIN_00))
    {
        /* Wait */
    }

    BSP_LED_Off(LED_RED);
    BSP_LED_Off(LED_BLUE);
    (void)memset(rxBuffer, 0L, sizeof(rxBuffer));
    /* Erase sector */
    (void)SpiFlash_Erase4KbSector(flashAddr);
    /* Write data to flash */
    (void)SpiFlash_WritePage(flashAddr, (uint8_t*)&txBuffer[0], bufferLen);
    /* Read data from flash */
    (void)SpiFlash_ReadData(flashAddr, (uint8_t*)&rxBuffer[0], bufferLen);

    /* Compare txBuffer and rxBuffer */
    if (memcmp(txBuffer, rxBuffer, (uint32_t)bufferLen) != 0)
    {
        BSP_LED_On(LED_RED);
    }
    else
    {
        BSP_LED_On(LED_BLUE);
    }

    /* Flash address offset */
    flashAddr += FLASH_SECTOR_SIZE;
    if (flashAddr >= FLASH_MAX_ADDR)
    {
        flashAddr = 0U;
    }
}

```


4.2 工作流程

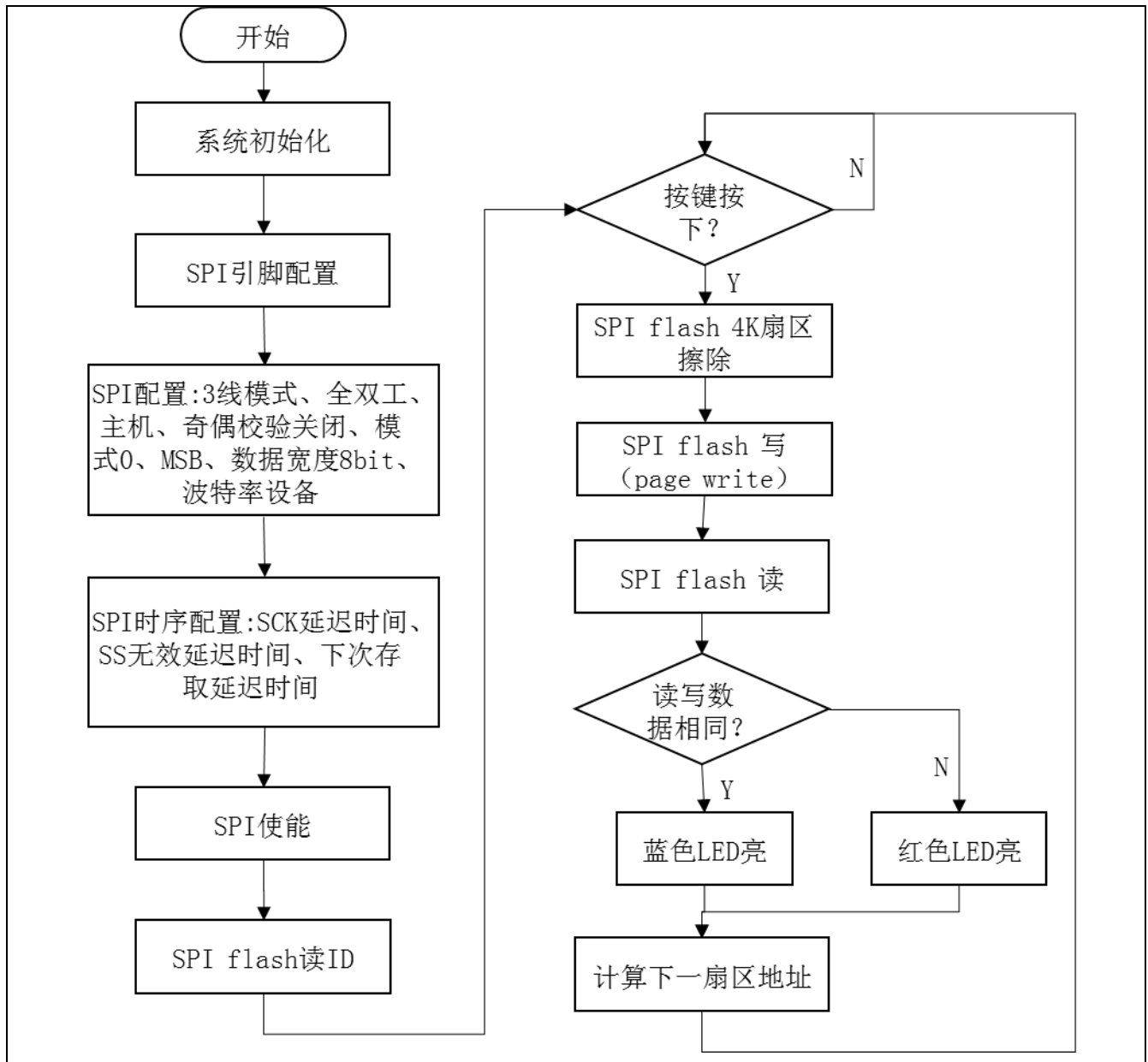


图 11 样例工作流程

4.3 代码运行

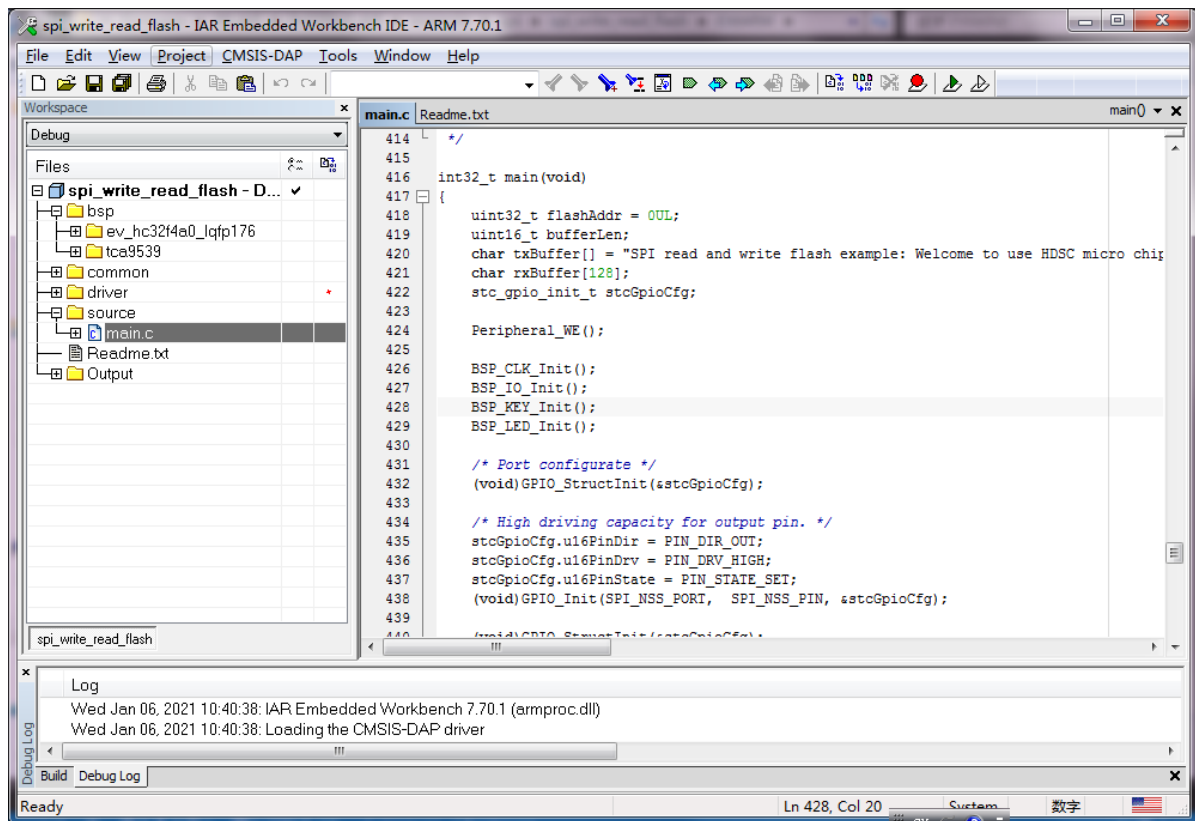
用户可以采用华大半导体网站下载的本系列芯片的 DDL 的样例代码（spi_write_read_flash），并配合评估用板（EV_F4A0_LQ176_V10）来学习使用 SPI 外设。



以下部分主要介绍如何在评估板上运行 spi_write_read_flash 样例代码并观察结果：

确认安装正确的 IAR EWARM v7.7 工具（请从 IAR 官方网站下载相应的安装包，并参考用户手册进行安装）。

下载并运行 spi\spi_write_read_flash\中的工程文件：

- 1) USB Micro 线连接 DAP 功能口 J25。
- 2) 打开 spi_write_read_flash 工程，并打开 ‘main.c’ 如下视图：



- 3) 点击  重新编译整个项目。
- 4) 点击  将代码下载到评估板上，全速运行。
- 5) 短按按键 SW10 触发 flash 擦除、写入、读出后比较功能。
- 6) 比较结果相同则蓝色 LED 亮，不同则红色 LED 亮。
- 7) 重复短按按键 SW10 触发下个扇区的 flash 擦除、写入、读出后比较功能（步骤 5）。

5 总结

以上章节简要介绍了 HC32F4A0 系列芯片的 SPI 外设，详细介绍了 SPI 各种模式的配置，示范了 SPI flash 的读写样例的演示过程，用户可以根据需求参照 DDL 库中的 SPI 相关样例进行学习或开发。

6 版本信息 & 联系方式

日期	版本	修改记录
2021/7/27	Rev1.0	初版发布



如果您在购买与使用过程中有任何意见或建议，请随时与我们联系。

Email: mcu@hdsc.com.cn

网址: www.hdsc.com.cn

通信地址: 上海市浦东新区中科路 1867 号 A 座 10 层

邮编: 201203

