32 位微控制器

HC32F4A0 系列的外部存储器控制器

EXMC

本产品支持芯片系列如下

F 系列	**HC32F4A0**

目　录

1 摘要

本篇应用笔记主要介绍 HC32F4A0 系列外部存储器控制器（External Memory Controller，EXMC）模块，并通过 NAND Flash、SDRAM、SRAM 样例，简要说明如何配置和访问外部存储器。

2 EXMC 简介

外部存储器控制器 EXMC 是一个用来访问各种片外存储器、实现数据交换的独立模块。

EXMC 通过配置可以把内部的 AMBA 协议接口转换为各种类型的专用片外存储器通信协议接口。

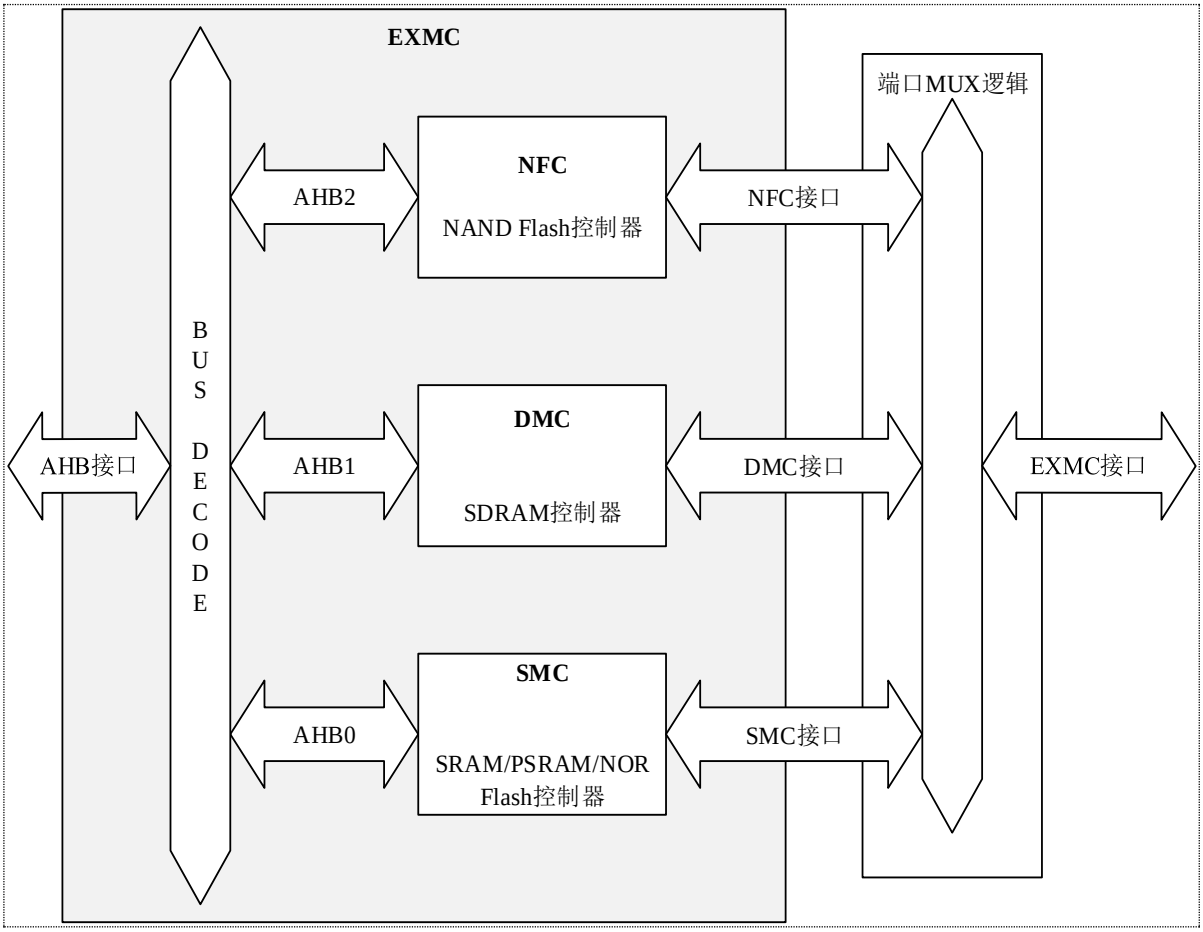
2.1 主要特性

- 支持的外部存储器类型：SRAM，PSRAM，NOR Flash，NAND Flash，SDRAM
- 接口协议：支持 AMBA 协议与各种外部存储器的接口转换协议
- 总线位宽：支持 8 位、16 位、32 位总线宽度
- 总线复用：NOR Flash 和 PSRAM 支持地址线和数据线复用
- 自动分割：AMBA 总线位宽和外部存储器不匹配时，支持自动分割及字节选择控制

2.2 基本框图

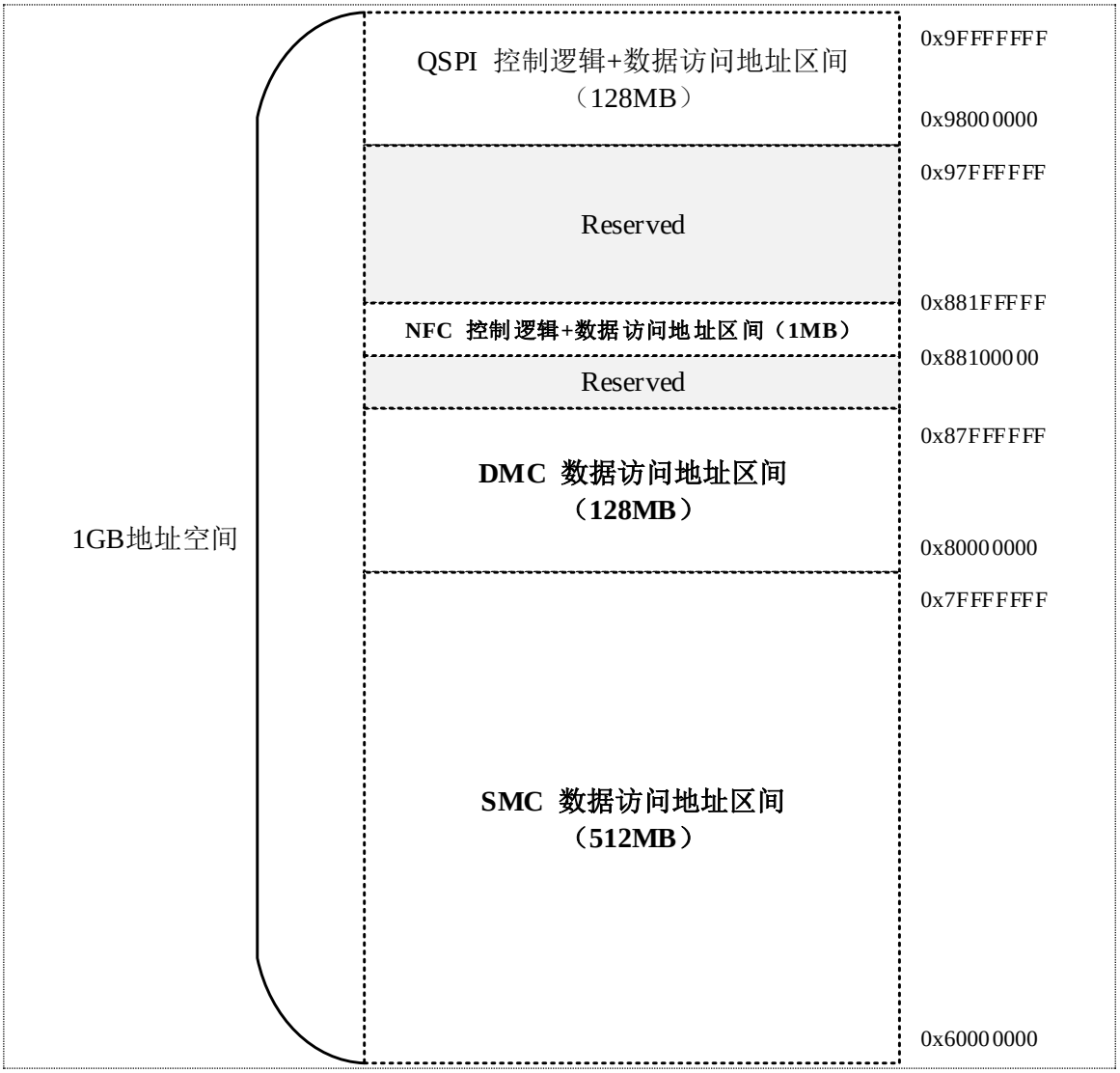
EXMC 的基本架构框图如下图所示，各种类型的外部存储器控制器独立生成对应协议的接口送至端口 MUX 逻辑。端口 MUX 逻辑将各种外部存储器的地址、数据、控制信号等共享在相同的端口上后再从芯片的端口上输出。因此 EXMC 控制器一次只能访问一个外部器件。

注：架构图中将 SRAM/PSRAM/NOR Flash 控制器定义为 SMC（Static Memory Controller）、SDRAM 控制器定义为 DMC（Dynamic Memory Controller）、将 NAND Flash 控制器定义为 NFC（NAND Flash Memory Controller），后面提到 SMC、DMC、NFC 时，与对应的控制器有相同的含义。



2.3 地址映射

HC32F4A0 系列中定义了总大小为 1GB 的外部存储器访问区间，用于不同类型的外部存储器和内部数据交换，包括 EXMC 和 QSPI 等。该 1GB 的空间按下图的方案分配，从 EXMC 的角度看，SRAM/PSRAM/NOR Flash（SMC）的数据访问映射 512MB 的地址空间、SDRAM（DMC）的数据访问映射 128MB 的地址空间、NAND Flash（NFC）的控制和数据访问映射 1MB 的地址空间。



3 EXMC 接口

SMC、DMC、NFC 对应不同类型的存储器，各自有不同类型的协议接口信号。EXMC 通过该接口协议实现和外部存储器的数据交换。以下为 EXMC 管脚和各种类型存储器所需要的协议接口说明。

EXMC 引脚名	方向	有效电平	功能描述		
			SMC	DMC	NFC
EXMC_CLK	O	H/L	SMC_CLK	DMC_CLK	-
EXMC_ADD29~0	O	H/L	SMC_ADD[29:18]	-	-
	O	H/L	SMC_ADD[17:16]	DMC_BA[1:0]	-
	O	H/L	SMC_ADD[15:0]	DMC_ADD[15:0]	-
EXMC_DATA31~0	IO	H/L	SMC_DATA[31:16]	DMC_DATA[31:16]	-
	IO	H/L	SMC_DATA[15:0]	DMC_DATA[15:0]	NFC_DATA[15:0]
EXMC_WE	O	L	SMC_WE	DMC_WE	NFC_WE
EXMC_CE7~0	O	L	SMC_BLS[3:0]	DMC_DQM[3:0]	NFC_CE[7:4]
	O	L	SMC_CS[3:0]	DMC_CS[3:0]	NFC_CE[3:0]
EXMC_OE	O	L	SMC_OE	DMC_RAS	NFC_RE
EXMC_BAA	O	L	SMC_BAA	DMC_CAS	NFC_WP
EXMC_ADV	O	L	SMC_ADV	-	-
EXMC_ALE	O	H	SMC_CRE	DMC_CKE	NFC_ALE
EXMC_CLE	O	H	-	DMC_AP	NFC_CLE
EXMC_RB7~0	I	L	-	-	NFC_RB[7:1]
	I	L	SMC_WAIT	-	NFC_RB[0]

表 3-1 EXMC 管脚

3.1 SMC 协议接口

SMC 功能基本特性如下：

- 支持 16 位、32 位外部存储器数据带宽
- AHB 字、半字、字节访问
- 为每个存储器 Chip 提供独立的片选控制
- 字节选择信号输出
- 地址线、数据线复用

- 可编程的协议时序参数
- 可编程速率的自动刷新动作（PSRAM 时使用）
- 具有 2 个 47 位的命令 FIFO
- 具有 4 个 36 位的写数据 FIFO
- 具有 4 个 32 位的读数据 FIFO
- 低功耗管理

SRAM/PSRAM/NOR Flash 控制（SMC）访问所需的协议接口如下表所示。

协议接口名	方向	有效电平	功能描述
SMC_CLK	out	-	SMC 的时钟输出
SMC_ADD[29:0]	out	-	SMC 的地址输出
SMC_DATA[31:0]	in/out	-	SMC 的数据
SMC_WE	out	L	SMC 的写使能
SMC_OE	out	L	SMC 的输出使能
SMC_CS[3:0]	out	L	SMC 的片选信号
SMC_BLS[3:0]	out	L	SMC 的字节选通信号
SMC_ADV	out	L	SMC 的地址锁存信号
SMC_CRE	out	H	SMC 的配置寄存器模式信号
SMC_WAIT	in	L	SMC 的输入等待信号
SMC_BAA	out	L	SMC 的地址提示信号

表 3-2 SMC 管脚

3.2 DMC 协议接口

DMC 功能基本特性如下：

- 支持 16 位、32 位外部存储器数据带宽
- 支持多达 16 位行地址、12 位列地址、2 位内部 Bank 地址
- AHB 字、半字、字节访问
- 为每个存储器 Chip 提供独立的片选控制
- 每个存储器 Chip 大小可独立配置
- 字节选择信号输出

- 自动进行行和 Bank 边界管理
- 可编程的协议时序参数
- 可编程速率的自动刷新动作
- 具有 2 个 41 位的命令 FIFO
- 具有 10 个 36 位的写数据 FIFO
- 具有 10 个 32 位的读数据 FIFO
- 低功耗管理

SDRAM 控制（DMC）访问所需的协议接口如下表所示。

协议接口名	方向	有效电平	功能描述
DMC_CLK	out	-	DMC 的时钟输出
DMC_ADD[15:0]	out	-	DMC 的地址输出
DMC_DATA[31:0]	inout	-	DMC 的数据
DMC_CKE	out	H	DMC 的 CLK 输出使能
DMC_AP	out	H	DMC 的自动充电信号
DMC_WE	out	L	DMC 的写使能
DMC_RAS	out	L	DMC 的行地址选通信号
DMC_CAS	out	L	DMC 的列地址选通信号
DMC_BA[1:0]	out	-	DMC 的 Bank 地址信号
DMC_CS[3:0]	out	L	DMC 的片选信号
DMC_DQM[3:0]	out	L	DMC 的字节选通信号

表 3-3 DMC 管脚

3.3 NFC 协议接口

NFC 功能基本特性如下：

- 支持 ONFI 协议，能够简单的实现对 SLC 和 MLC NAND Flash 设备的访问
- 支持页大小 2KB/4KB/8KB 的设备
- 支持 8bit 和 16bit 位宽的设备
- 支持每 512byte 为单位的 1bit 汉明码 ECC 纠错
- 能够计算每 512byte 为单位的 4bit BCH 码，并在 ECC 错误时给出伴随式

- 访问时序可通过寄存器配置

NAND Flash 控制（NFC）访问所需的协议接口如下表所示。

协议接口名	方向	有效电平	功能描述
NFC_CLE	out	H	NFC 的命令锁存信号
NFC_ALE	out	H	NFC 的地址锁存信号
NFC_DATA[15:0]	inout	-	NFC 的地址和数据
NFC_CE[7:0]	out	L	NFC 的片选信号
NFC_WE	out	L	NFC 的写使能
NFC_RE	out	L	NFC 的读使能
NFC_WP	out	L	NFC 的写保护信号
NFC_RB[7:0]	in	L	NFC 的输入忙信号

表 3-4 NFC 管脚

4 EXMC 应用

针对 HC32F4A0 评估用板（EV_F4A0_LQ176_V10），分别通过 SRAM、SDRAM 和 NAND Flash 存储器说明 EXMC 如何配置。

4.1 SMC 接口非总线复用模式的异步 16 位 SRAM 存储器

4.1.1 硬件连接

本文中使用评估用板（EV_F4A0_LQ176_V10）IS62WV51216BLL 存储器作为 SMC 例子说明。

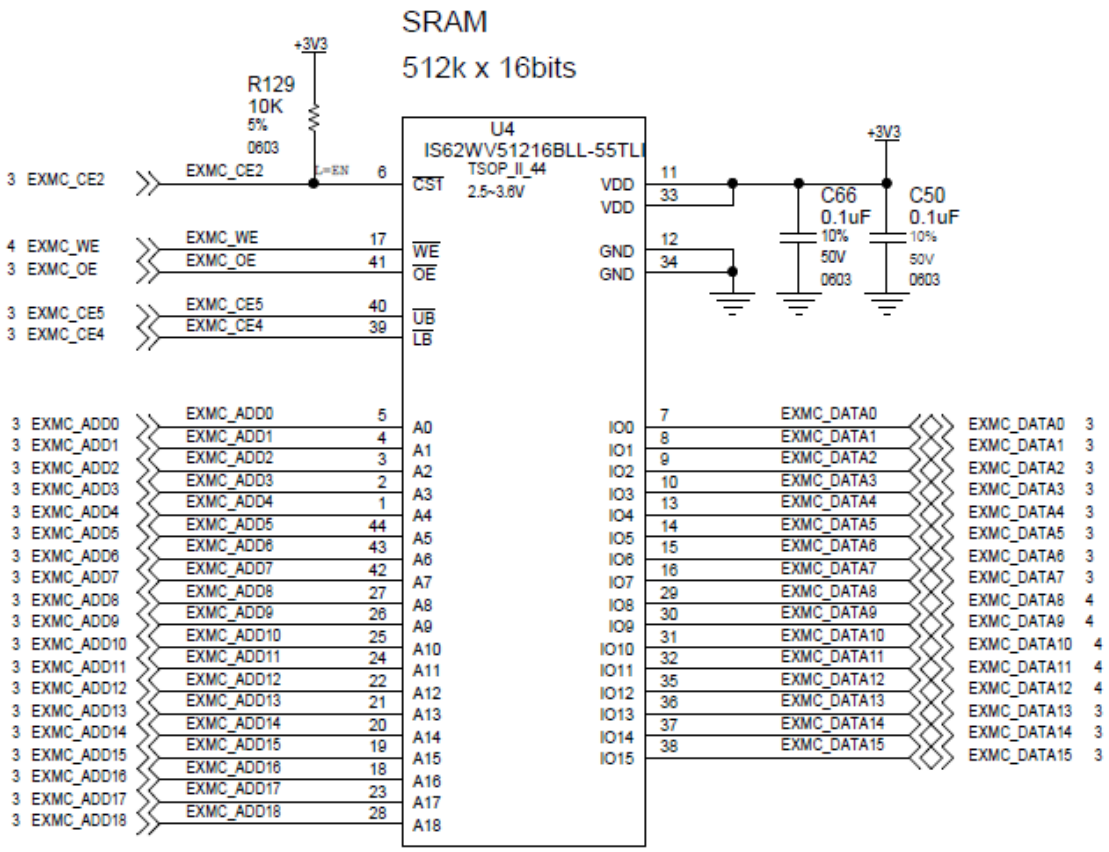


图 4-1 16 位 SRAM：IS62WV51216BLL 连接至 HC32F4A0 的 EXMC 管脚

SRAM 信号与 EXMC 管脚的对应如下表所示。

存储器信号	EXMC 管脚	信号说明
A[18:0]	EXMC_ADD[18:0]	地址线 A0 至 A18
IO[15:0]	EXMC_DATA[15:0]	数据线 D0 至 D15
$\overline{CS1}$	EXMC_CS2	芯片使能
$\overline{WE}$	EXMC_WE	写使能
$\overline{OE}$	EXMC_OE	输出使能
$\overline{LB}$	EXMC_CE4	低字节控制
$\overline{UB}$	EXMC_CE5	高字节控制

表 4-1 IS62WV51216BLL 与 EXMC 管脚的对应

4.1.2 SMC 配置

IS62WV51216BLL 是一个非总线复用、异步的 16 位存储器。

SMC 配置如下：

- 芯片使能：Chip2
- 读写访问方式为异步：SMC_CPCR 寄存器 RSYN/WSYNC 设置为’b0’
- 读突发数据长度为 1 次读传输：SMC_CPCR 寄存器 RBL 设置为’b000’
- 数据总线为 16 位宽：SMC_CPCR 寄存器 MW 设置为’b01’
- 存储器为非总线复用： SMC_BACR 寄存器 MUXMD 设置为’b0’
- 保持其它的所有参数为清除状态

4.1.3 时序配置

4.1.3.1 SMC 异步 16 位 SRAM 典型时序

1） 单次读动作的基本时序图和设定例

基本设定	MW	RSYN	RBL	WSYN	WBL	BAA	ADV	BLS
	<set>	<set>	b000	-	-	-	b0	b0
时序设定	t_rc	t_wc	t_ceoe	t_wp	t_pc	t_tr	- 表示不关注	
	b0011	-	b001	-	-	-	<set> 表示用户设定值	

表 4-2 单次读动作基本设定例

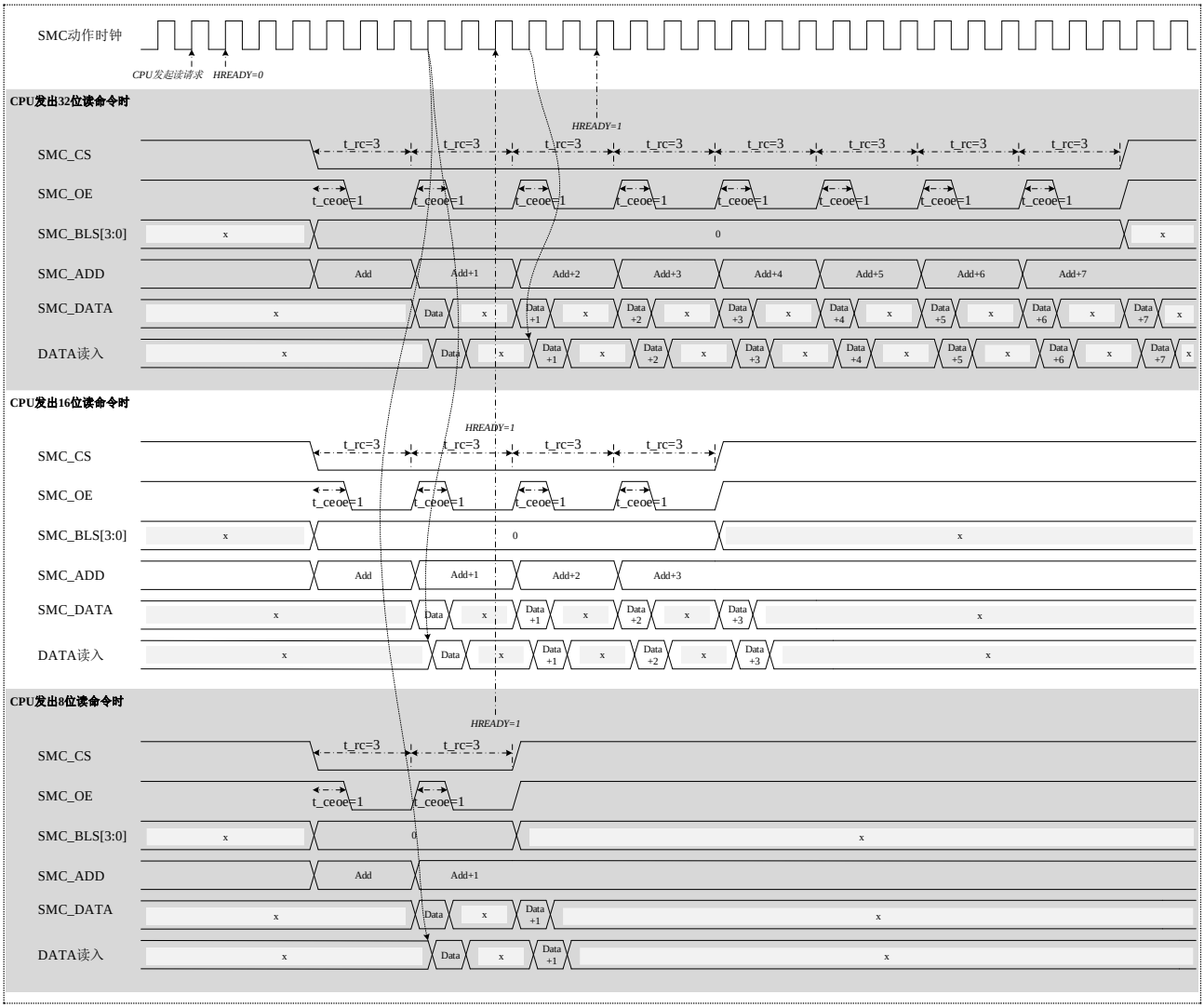


图 4-2 单次读动作基本时序（异步方式（RSYN=0）&16 位位宽（MW=01））

2） 单次写动作的基本时序图和设定例

基本设定	MW	RSYN	RBL	WSYN	WBL	BAA	ADV	BLS
	<set>	-	-	<set>	b000	-	b0	<set>
时序设定	t_rc	t_wc	t_ceoe	t_wp	t_pc	t_tr	- 表示不关注	
	-	b0100	-	b010	-	-	<set> 表示用户设定值	

表 4-3 单次写动作基本设定例

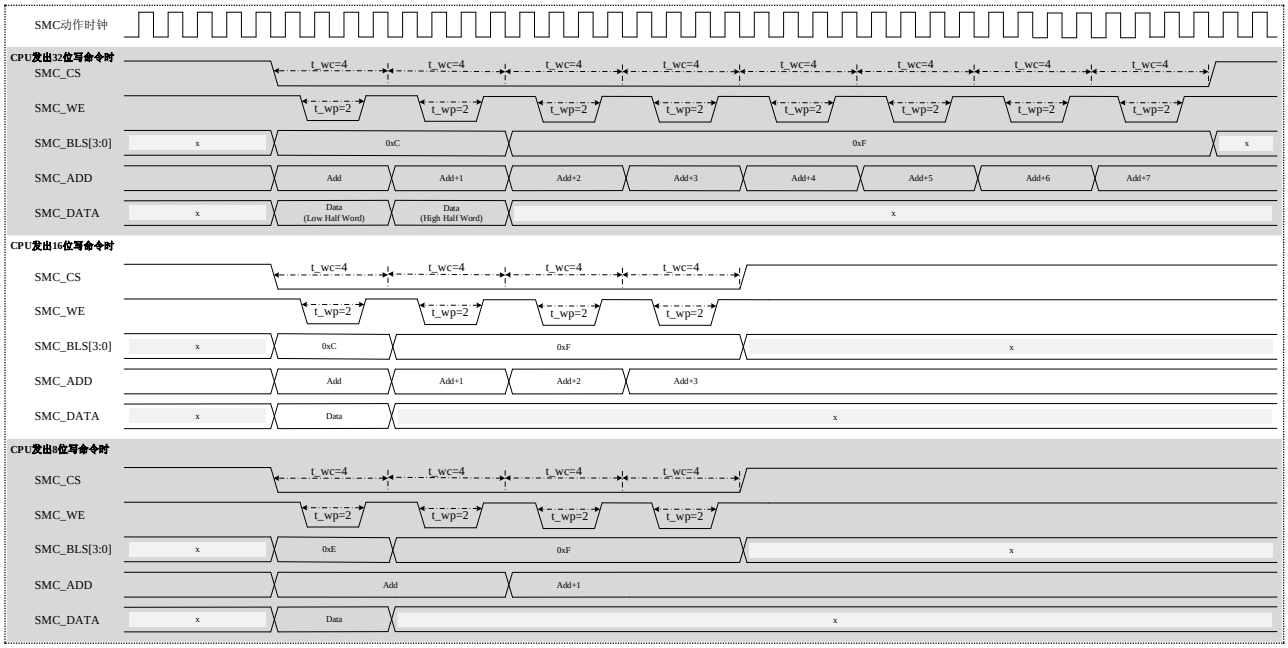


图 4-3 单次写动作基本时序（异步方式（WSYN=0）&16 位位宽（MW=01）&BLS=0）

4.1.3.2 时序参数计算

读写时序中，有几个比较重要的时间参数，必须按照 SRAM 存储器的数据手册给出的时序数据，计算和设置下列参数：

时间参数	时间要求	说明
t_{RC}	不小于 55 ns	读周期时间
t_{WC}	不小于 55 ns	写周期时间
t_{DOE}	最迟不大于 25ns	从接收到读使能信号到给出有效数据的时间
t_{PWE}	不小于 40ns	从接收到写使能信号到数据采样的时间

表 4-4 IS62WV51216BLL-55ns 型号 SRAM 的时间参数

系统时钟为 240MHz，EXMC 总线时钟为 60MHz，使用上述存储器的时序参数和 EXMC 时序配置对应，可以得到：

SMC_TMCR 寄存器 t_rc 位为 4；

SMC_TMCR 寄存器 t_wc 位为 4；

SMC_TMCR 寄存器 t_ceoe 位为 1；

SMC_TMCR 寄存器 t_wp 位为 3；

4.1.4 程序示例

SRAM 操作通过设备驱动库（Device Driver Library, DDL）的样例 exmc_smc_sram_is62wv51216 展示。下面主要介绍比较关键的代码。

1) 根据 IS62WV51216BLL-55ns 型号时序，修改 BSP 初始化配置参数，初始化 SRAM

```
en_result_t BSP_SMC_IS62WV51216_Init(void)
{
    en_result_t enRet;
    stc_exmc_smc_init_t stcSmcInit;

    /* Initialize SMC port. */
    EV_EXMC_SMC_PortInit();

    /* Enable SMC module clk */
    PWC_Fcg3PeriphClockCmd(PWC_FCG3_SMC, Enable);

    /* Enable SMC. */
    EXMC_SMC_Cmd(Enable);

    EXMC_SMC_ExitLowPower();
    while (EXMC_SMC_READY != EXMC_SMC_GetStatus())
    {}

    /* Configure SMC width && CS &chip & timing. */
    stcSmcInit.stcChipCfg.u32ReadMode = EXMC_SMC_MEM_READ_ASYNC;
    stcSmcInit.stcChipCfg.u32ReadBurstLen = EXMC_SMC_MEM_READ_BURST_1;
    stcSmcInit.stcChipCfg.u32WriteMode = EXMC_SMC_MEM_WRITE_ASYNC;
    stcSmcInit.stcChipCfg.u32WriteBurstLen = EXMC_SMC_MEM_WRITE_BURST_1;
    stcSmcInit.stcChipCfg.u32SmcMemWidth = EXMC_SMC_MEMORY_WIDTH_16BIT;
    stcSmcInit.stcChipCfg.u32BAA = EXMC_SMC_BAA_PORT_DISABLE;
    stcSmcInit.stcChipCfg.u32ADV = EXMC_SMC_ADV_PORT_DISABLE;
    stcSmcInit.stcChipCfg.u32BLS = EXMC_SMC_BLS_SYNC_CS;
    stcSmcInit.stcChipCfg.u32AddressMatch = IS62WV51216_SMC_MATCH_ADDR;
    stcSmcInit.stcChipCfg.u32AddressMask = IS62WV51216_SMC_MASK_ADDR;
    stcSmcInit.stcTimingCfg.u32RC = 4UL;
    stcSmcInit.stcTimingCfg.u32WC = 4UL;
    stcSmcInit.stcTimingCfg.u32CEOE = 1UL;
    stcSmcInit.stcTimingCfg.u32WP = 3UL;
    stcSmcInit.stcTimingCfg.u32PC = 0UL;
    stcSmcInit.stcTimingCfg.u32TR = 0UL;
    (void)EXMC_SMC_Init(IS62WV51216_MAP_SMC_CHIP, &stcSmcInit);

    /* Set command: updateregs */
    EXMC_SMC_SetCommand(IS62WV51216_MAP_SMC_CHIP, EXMC_SMC_CMD_UPDATEREGS,
0UL, 0UL);

    /* Check timing status */
    do
    {
```

```
        enRet = EXMC_SMC_CheckTimingStatus(IS62WV51216_MAP_SMC_CHIP,
&stcSmcInit.stcTimingCfg);
    } while (Ok != enRet);

    /* Check chip status */
    do
    {
        enRet = EXMC_SMC_CheckChipStatus(IS62WV51216_MAP_SMC_CHIP,
&stcSmcInit.stcChipCfg);
    } while (Ok != enRet);

    return Ok;
}
```

2) SRAM 样例主程序

```
int32_t main(void)
{
    uint32_t i;
    uint32_t j;
    uint32_t u32MemAddr;

    /* MCU Peripheral registers write unprotected */
    Peripheral_WE();

    /* Initialize system clock: */
    BSP_CLK_Init();

    /* PCLK0, HCLK  Max 240MHz */
    /* PCLK1, PCLK4 Max 120MHz */
    /* PCLK2, PCLK3 Max 60MHz  */
    /* EXCLK 60MHz */
    CLK_ClkDiv(CLK_CATE_ALL, (CLK_PCLK0_DIV1 | CLK_PCLK1_DIV2 | \
                              CLK_PCLK2_DIV4 | CLK_PCLK3_DIV4 | \
                              CLK_PCLK4_DIV2 | CLK_EXCLK_DIV4 | CLK_HCLK_DIV1));

    /* Initialize UART print */
    (void)DDL_PrintfInit();

    /* Initialize LED */
    BSP_IO_Init();
    BSP_LED_Init();

    /* Initialize test data. */
    for (i = 0UL; i < DATA_BUFFER_LEN; i++)
    {
        m_au8ReadData[i] = 0U;
        m_au8WriteData[i] = 0x12U;

        m_au16ReadData[i] = 0U;
        m_au16WriteData[i] = 0x5678U;

        m_au32ReadData[i] = 0UL;
        m_au32WriteData[i] = 0xAABBCCDDUL;
    }
```

```
    /* Configure SRAM. */
    (void)IS62WV51216_Init();
    IS62WV51216_GetMemInfo(&m_u32MemStartAddr, &m_u32MemByteSize);

    /* MCU Peripheral registers write protected */
    Peripheral_WP();

    m_u32MemHalfwordSize = m_u32MemByteSize/2UL;
    m_u32MemWordSize = m_u32MemByteSize/4UL;

    (void)printf("Memory start address: 0x%.8x \r\n", (unsigned int)m_u32MemStartAddr);
    (void)printf("Memory end   address: 0x%.8x \r\n", (unsigned int)(m_u32MemStartAddr +
m_u32MemByteSize - 1UL));
    (void)printf("Memory size  (Bytes): 0x%.8x \r\n\r\n", (unsigned int)m_u32MemByteSize);

    for (;;)
    {
      m_u32TestCnt++;
      (void)printf("******** Write/read test times: %u ********\r\n", (unsigned int)m_u32TestCnt);

      /****************** Byte read/write *****************/
      m_u32ByteTestErrorCnt = 0UL;
      u32MemAddr = m_u32MemStartAddr;
      for (i = 0UL; i < m_u32MemByteSize; i += DATA_BUFFER_LEN)
      {
        (void)IS62WV51216_WriteMem8(u32MemAddr, m_au8WriteData, DATA_BUFFER_LEN);
        (void)IS62WV51216_ReadMem8(u32MemAddr, m_au8ReadData, DATA_BUFFER_LEN);

        /* Verify write/read data. */
        for (j = 0UL; j < DATA_BUFFER_LEN; j++)
        {
          if (m_au8WriteData[j] != m_au8ReadData[j])
          {
            m_u32ByteTestErrorCnt++;
            (void)printf("Byte read/write error: address = 0x%.8x; write data = 0x%x; read data =
0x%x\r\n",
                (unsigned int)(u32MemAddr+j), (unsigned int)m_au8WriteData[j], (unsigned
int)m_au8ReadData[j]);
          }
        }

        u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au8ReadData[0]));
        (void)memset(m_au8ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au8ReadData[0])));
        BSP_LED_Toggle(LED_BLUE);
      }

      (void)printf("    Byte read/write error data count: %u \r\n", (unsigned int)m_u32ByteTestErrorCnt);

      /**************** Halfword read/write ****************/
      BSP_LED_Toggle(LED_BLUE);
      m_u32HalfwordTestErrorCnt = 0UL;
      u32MemAddr = m_u32MemStartAddr;
      for (i = 0UL; i < m_u32MemHalfwordSize; i += DATA_BUFFER_LEN)
      {
        (void)IS62WV51216_WriteMem16(u32MemAddr, m_au16WriteData, DATA_BUFFER_LEN);
        (void)IS62WV51216_ReadMem16(u32MemAddr, m_au16ReadData, DATA_BUFFER_LEN);
```

```
            /* Verify write/read data. */
            for (j = 0UL; j < DATA_BUFFER_LEN; j++)
            {
               if (m_au16WriteData[j] != m_au16ReadData[j])
               {
                  m_u32HalfwordTestErrorCnt++;
                  (void)printf("Halfword read/write error: address = 0x%.8x; write data = 0x%x; read data =
0x%x\r\n",
                        (unsigned int)(u32MemAddr+j), (unsigned int)m_au16WriteData[j], (unsigned
int)m_au16ReadData[j]);
               }
            }

            u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au16ReadData[0]));
            (void)memset(m_au16ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au16ReadData[0])));
            BSP_LED_Toggle(LED_BLUE);
         }

         (void)printf(" Halfword read/write error data count: %u \r\n", (unsigned
int)m_u32HalfwordTestErrorCnt);

         /****************** Word read/write ******************/
         BSP_LED_Toggle(LED_BLUE);
         m_u32WordTestErrorCnt = 0UL;
         u32MemAddr = m_u32MemStartAddr;
         for (i = 0UL; i < m_u32MemWordSize; i += DATA_BUFFER_LEN)
         {
            (void)IS62WV51216_WriteMem32(u32MemAddr, m_au32WriteData, DATA_BUFFER_LEN);
            (void)IS62WV51216_ReadMem32(u32MemAddr, m_au32ReadData, DATA_BUFFER_LEN);

            /* Verify write/read data. */
            for (j = 0UL; j < DATA_BUFFER_LEN; j++)
            {
               if (m_au32WriteData[j] != m_au32ReadData[j])
               {
                  m_u32WordTestErrorCnt++;
                  (void)printf("Word read/write error: address = 0x%.8x; write data = 0x%.8x; read data =
0x%.8x\r\n",
                        (unsigned int)(u32MemAddr+j), (unsigned int)m_au32WriteData[j], (unsigned
int)m_au32ReadData[j]);
               }
            }

            u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au32ReadData[0]));
            (void)memset(m_au32ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au32ReadData[0])));
            BSP_LED_Toggle(LED_BLUE);
         }

         (void)printf("     Word read/write error data count: %u \r\n\r\n", (unsigned
int)m_u32WordTestErrorCnt);

         /****************** Error check ******************/
         if ((m_u32ByteTestErrorCnt > 0UL)     || \
            (m_u32HalfwordTestErrorCnt > 0UL) || \
            (m_u32WordTestErrorCnt > 0UL))
         {
            BSP_LED_On(LED_RED);
```

```
        }
    }
}
```

3) SRAM 8Bit 读写操作

```
en_result_t IS62WV51216_WriteMem8(uint32_t u32Address,
                                  const uint8_t au8SrcBuffer[],
                                  uint32_t u32BufferSize)
{
    uint32_t i;
    en_result_t enRet = ErrorInvalidParameter;

    if (0UL != u32BufferSize)
    {
        DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
        DDL_ASSERT((u32Address + u32BufferSize) <= IS62WV51216_END_ADDRESS);

        for (i = 0UL; i < u32BufferSize; i++)
        {
            *(uint8_t *)(u32Address + i) = au8SrcBuffer[i];
        }
        enRet = Ok;
    }

    return enRet;
}

en_result_t IS62WV51216_ReadMem8(uint32_t u32Address,
                                 uint8_t au8DstBuffer[],
                                 uint32_t u32BufferSize)
{
    uint32_t i;
    en_result_t enRet = ErrorInvalidParameter;

    if ((NULL != au8DstBuffer) && (0UL != u32BufferSize))
    {
        DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
        DDL_ASSERT((u32Address + u32BufferSize) <= IS62WV51216_END_ADDRESS);

        for (i = 0UL; i < u32BufferSize; i++)
        {
            au8DstBuffer[i] = *(uint8_t *)(u32Address + i);
        }
        enRet = Ok;
    }

    return enRet;
}
```

4) SRAM 16Bit 读写操作

```
en_result_t IS62WV51216_WriteMem16(uint32_t u32Address,
                                   const uint16_t au16SrcBuffer[],
                                   uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (0UL != u32BufferSize)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_HALFWORD(u32Address));
      DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 1UL)) <= IS62WV51216_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         *((uint16_t *)u32Address) = au16SrcBuffer[i];
         u32Address += 2UL;
      }
      enRet = Ok;
   }

   return enRet;
}

en_result_t IS62WV51216_ReadMem16(uint32_t u32Address,
                                  uint16_t au16DstBuffer[],
                                  uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if ((NULL != au16DstBuffer) && (0UL != u32BufferSize))
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_HALFWORD(u32Address));
      DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 1UL)) <= IS62WV51216_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         au16DstBuffer[i] = *((uint16_t *)u32Address);
         u32Address += 2UL;
      }
      enRet = Ok;
   }

   return enRet;
}
```

5) SRAM 32Bit 读写操作

```
en_result_t IS62WV51216_WriteMem32(uint32_t u32Address,
                                   const uint32_t au32SrcBuffer[],
                                   uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (0UL != u32BufferSize)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_WORD(u32Address));
      DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 2UL)) <= IS62WV51216_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         *((uint32_t *)u32Address) = au32SrcBuffer[i];
         u32Address += 4UL;
      }
      enRet = Ok;
   }

   return enRet;
}

en_result_t IS62WV51216_ReadMem32(uint32_t u32Address,
                                  uint32_t au32DstBufferBuf[],
                                  uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (NULL != au32DstBufferBuf)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_WORD(u32Address));
      DDL_ASSERT(u32Address >= IS62WV51216_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 2UL)) <= IS62WV51216_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         au32DstBufferBuf[i] = *((uint32_t *)u32Address);
         u32Address += 4UL;
      }
      enRet = Ok;
   }

   return enRet;
}
```

4.2 DMC 接口 16 位 SDRAM 存储器

4.2.1 硬件连接

本文中使用评估用板（EV_F4A0_LQ176_V10）IS42S16400J-7TL 存储器作为 DMC 例子说明。

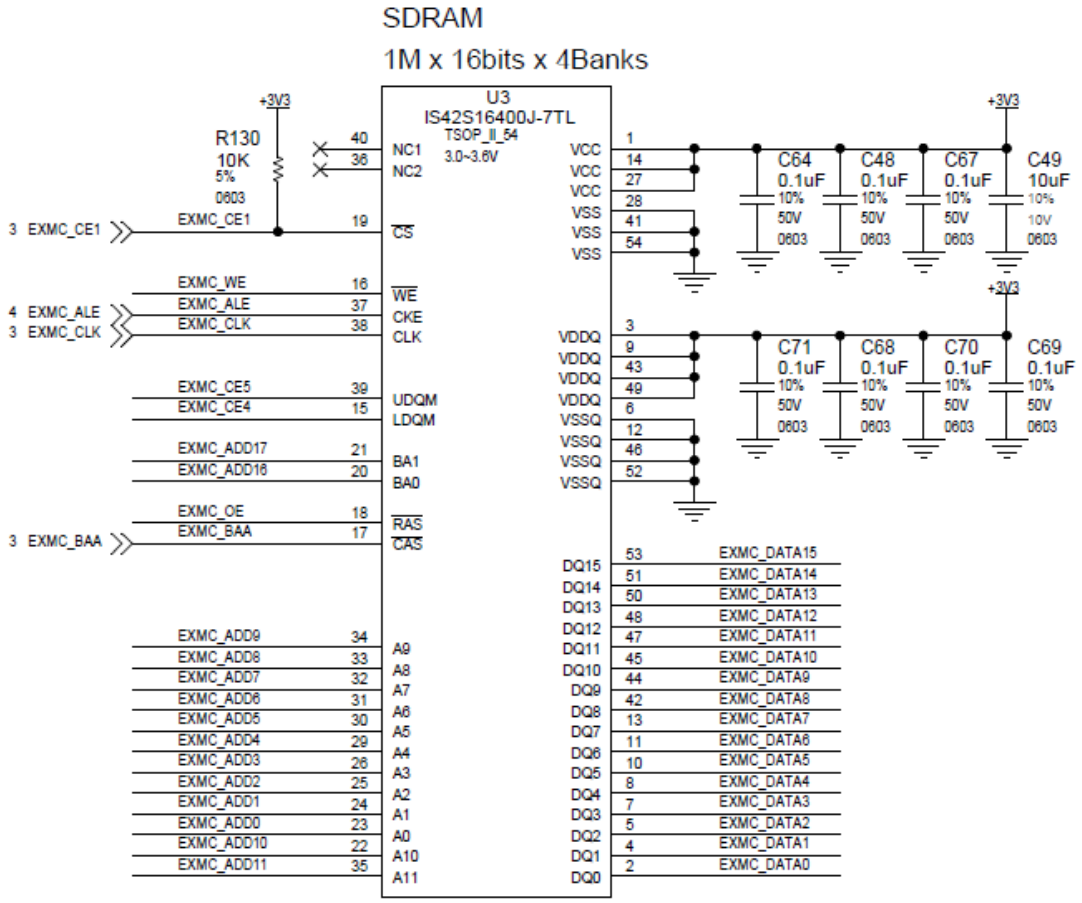


图 4-4 16 位 SDRAM：IS42S16400J-7TL 连接至 HC32F4A0 的 EXMC 管脚

SDRAM 信号与 EXMC 管脚的对应如下表所示。

存储器信号	EXMC 管脚	信号说明
A[11:0]	EXMC_ADD[11:0]	地址线 A0 至 A11
DQ[15:0]	EXMC_DATA[15:0]	数据线 D0 至 D15
$\overline{CS}$	EXMC_CS1	芯片使能
$\overline{WE}$	EXMC_WE	写使能
CKE	EXMC_ALE	时钟使能

CLK	EXMC_CLK	时钟
LDQM	EXMC_CE4	输入/输出屏蔽
UDQM	EXMC_CE5	输入/输出屏蔽
BA0	EXMC_ADD16	Bank 地址
BA1	EXMC_ADD17	Bank 地址
$\overline{RAS}$	EXMC_OE	列地址选通脉冲
$\overline{CAS}$	EXMC_BAA	行地址选通脉冲

表 4-5 IS42S16400J-7TL 与 EXMC 管脚的对应

4.2.2 DMC 配置

IS42S16400J-7TL 是 16 位存储器。参考该芯片数据手册，可得到参数如下：行地址为 A0~A11，列地址为 A0~7，自动预充电引脚为 A10，位宽为 16 位，刷新频率位 64ms/4096 行。

要求最慢在 64ms 内对 4096 行（对应 12-bit 行地址空间）完成刷新，所以每刷一次的时间间隔是 64ms÷4096=15.625us。换算成 EXMC 的 CLK（60MHz）：15.625us÷(1/60MHz)=938。为了保证刷新周期在 SDRAM 的任何工作状态下都能小于 64ms，要再减去 38 个 CLK，即 938-38=900。

DMC 配置如下：

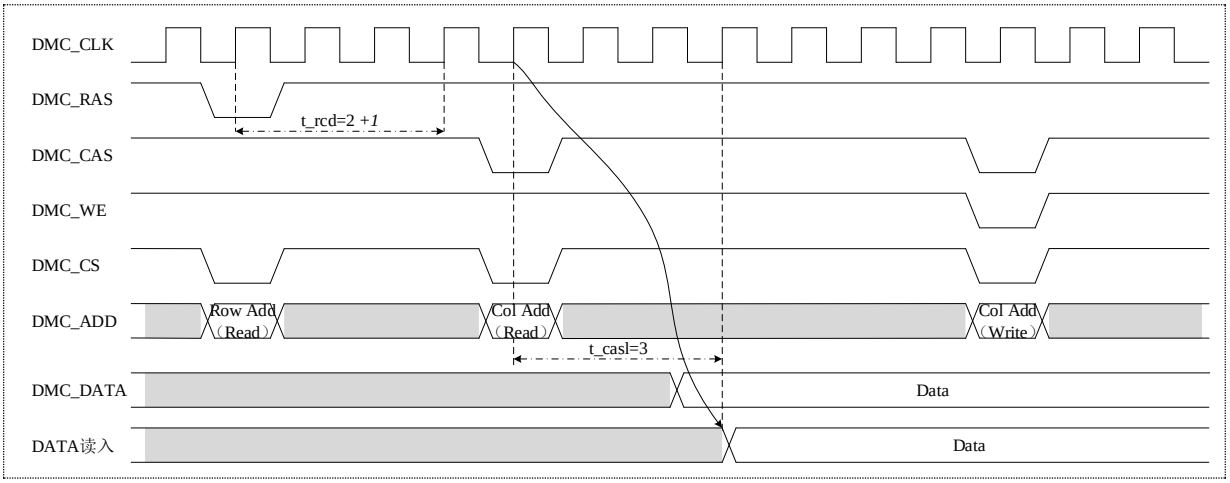
- 芯片使能：Chip1
- DMC_RFTR 寄存器设置为 900
- 数据总线为 16 位宽：DMC_BACR 寄存器 DMCMW 设置为’b00’
- 列地址为 A0~A11：DMC_CPCR 寄存器 COLBS 设置为’b000’
- 行地址为 A0~A7：DMC_CPCR 寄存器 ROWBS 设置为’b001’
- 自动预充电引脚为 A10：DMC_CPCR 寄存器 APBS 设置为’b0’
- 自动刷新 Chip 数目：DMC_CPCR 寄存器 ACTCP 设置为’b01’
- 保持其它的所有参数为清除状态

4.2.3 时序配置

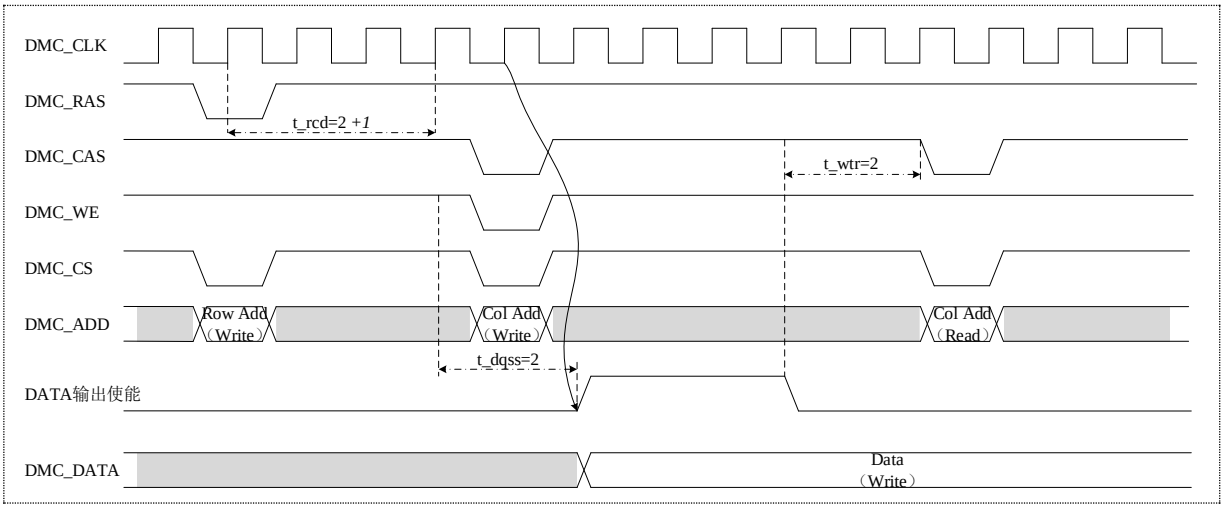
4.2.3.1 DMC 典型时序

DMC 可以把 AHB 的单次或突发读写操作转换成存储器的读写操作。与外部 SDRAM 存储器进行数据读写的方式有下面几种：

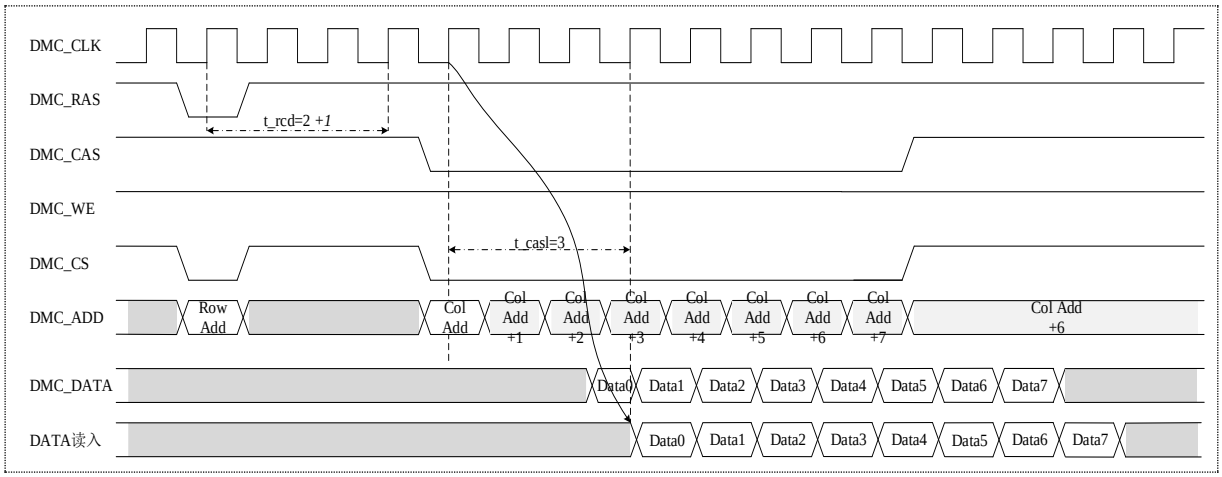
1） 单次读动作后对同一行其它列执行写动作的基本时序图



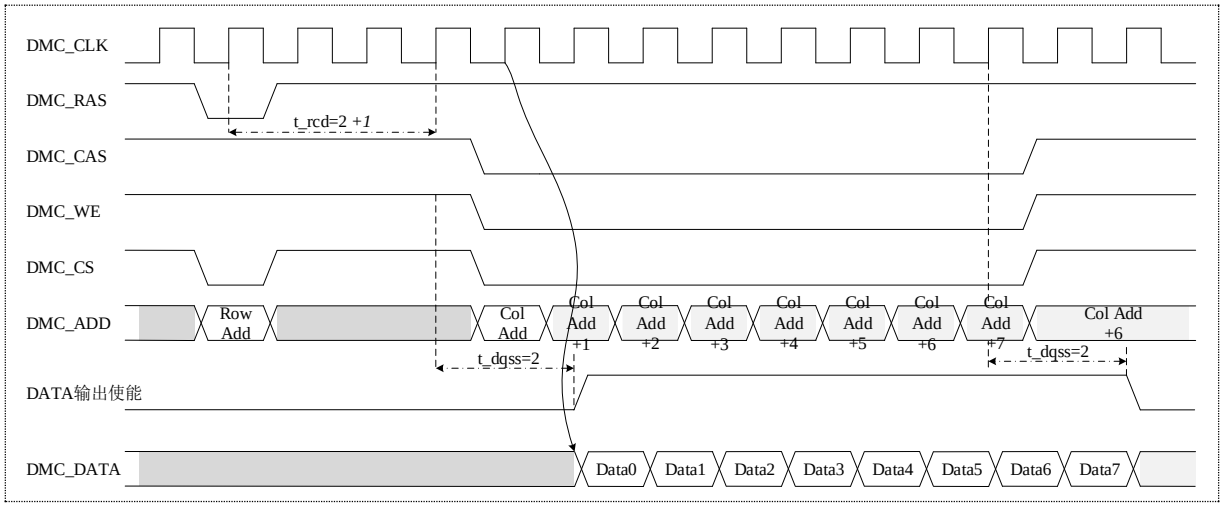
2） 单次写动作后对同一行的其它列执行读动作的基本时序图



3） 突发读动作的基本时序图



4） 突发写动作的基本时序图



4.2.3.2 时序参数计算

IS42S16400J-7TL 时序参数参数如下：

时间参数	时间要求	说明
CAS Latency	等于 2	列地址选通延迟
tMRD	2 CLK	加载模式寄存器激活延迟
tRC	不小于63ns	两个行有效命令之间的延迟，以及两个相邻刷新命令之间的延迟
tRAS	不小于42ns	前一个"行有效"与"预充电"命令之间的时间
tRP	不小于15ns	预充电命令与其它命令之间的延迟
tRCD	不小于15ns	行有效命令到列读写命令之间的延迟
tWR	2 CLK	写命令与预充电之间的延迟
tPED	1 CLK	退出低功耗延迟
tXSR	不小于70ns	退出自刷新命令后的延迟

表 4-6 IS42S16400J-7TL 的时间参数

系统工作电压为 3.3V，时钟为 240MHz，EXMC 总线时钟为 60MHz，使用上述存储器的时序参数，可以设置 EXMC DMC 时序参数如下：

DMC_TMCR_t_casl 寄存器 t_casl 位为 2;

DMC_TMCR_t_dqss 寄存器 t_dqss 位为 0;

DMC_TMCR_t_mrd 寄存器 t_mrd 位为 2;

DMC_TMCR_t_ras 寄存器 t_ras 位为 3;

DMC_TMCR_t_rc 寄存器 t_rc 位为 4;

DMC_TMCR_t_rcd 寄存器 t_rcd 位为 1;

DMC_TMCR_t_rfc 寄存器 t_rfc 位为 4;

DMC_TMCR_t_rp 寄存器 t_rp 位为 1;

DMC_TMCR_t_rrd 寄存器 t_rrd 位为 1;

DMC_TMCR_t_wr 寄存器 t_wr 位为 2;

DMC_TMCR_t_wtr 寄存器 t_wtr 位为 1;

DMC_TMCR_t_xp 寄存器 t_xp 位为 1;

DMC_TMCR_t_xsr 寄存器 t_xsr 位为 5;

DMC_TMCR_t_esr 寄存器 t_esr 位为 5;

4.2.4 程序示例

SDRAM 操作通过设备驱动库（Device Driver Library, DDL）的样例 exmc_dmc_sdram_is42s16400j7tli 展示。下面主要介绍比较关键的代码。

1) 根据 IS42S16400J-7TL 时序，修改 BSP 初始化配置参数，初始化 SDRAM：

```
en_result_t BSP_DMC_IS42S16400J7TLI_Init(void)
{
  uint32_t u32MdRegVal;
  stc_exmc_dmc_init_t stcDmcInit;
  stc_exmc_dmc_cs_cfg_t stcCsCfg;

  /* Initialization DMC port.*/
  EV_EXMC_DMC_PortInit();

  /* Enable DMC module clk */
  PWC_Fcg3PeriphClockCmd(PWC_FCG3_DMC, Enable);

  /* Enable DMC. */
  EXMC_DMC_Cmd(Enable);

  /* Configure DMC width && refresh period & chip & timing. */
  (void)EXMC_DMC_StructInit(&stcDmcInit);
  stcDmcInit.u32RefreshPeriod = 900UL;
  stcDmcInit.stcChipCfg.u32ColumnBitsNumber = EXMC_DMC_COLUMN_BITS_NUM_8;
  stcDmcInit.stcChipCfg.u32RowBitsNumber = EXMC_DMC_ROW_BITS_NUM_12;
  stcDmcInit.stcChipCfg.u32MemBurst = EXMC_DMC_MEM_BURST_1;
  stcDmcInit.stcChipCfg.u32AutoRefreshChips = EXMC_DMC_AUTO_REFRESH_CHIPS_01;

  stcDmcInit.stcTimingCfg.u32CASL = 2UL;
  stcDmcInit.stcTimingCfg.u32DQSS = 0UL;
  stcDmcInit.stcTimingCfg.u32MRD = 2UL;
  stcDmcInit.stcTimingCfg.u32RAS = 3UL;
  stcDmcInit.stcTimingCfg.u32RC = 4UL;
  stcDmcInit.stcTimingCfg.u32RCD = 1UL;
  stcDmcInit.stcTimingCfg.u32RFC = 4UL;
  stcDmcInit.stcTimingCfg.u32RP = 1UL;
  stcDmcInit.stcTimingCfg.u32RRD = 1UL;
  stcDmcInit.stcTimingCfg.u32WR = 2UL;
  stcDmcInit.stcTimingCfg.u32WTR = 1UL;
  stcDmcInit.stcTimingCfg.u32XP = 1UL;
  stcDmcInit.stcTimingCfg.u32XSR = 5UL;
  stcDmcInit.stcTimingCfg.u32ESR = 5UL;
```

```
    (void)EXMC_DMC_Init(&stcDmcInit);

    /* Configure DMC address space. */
    stcCsCfg.u32AddrMask = IS42S16400J7TLI_MAP_DMC_ADDR_MASK;
    stcCsCfg.u32AddrMatch = IS42S16400J7TLI_MAP_DMC_ADDR_MATCH;
    stcCsCfg.u32AddrDecodeMode = EXMC_DMC_CS_DECODE_ROWBANKCOL;
    (void)EXMC_DMC_CsConfig(IS42S16400J7TLI_MAP_DMC_CHIP, &stcCsCfg);

    /* SDRAM initialization sequence. */
    u32MdRegVal = IS42S16400J7TLI_MR_VALUE;

    if (2UL == stcDmcInit.stcTimingCfg.u32CASL)
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_CAS_LATENCY_2;
    }
    else
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_CAS_LATENCY_3;
    }

    if (EXMC_DMC_MEM_BURST_1 == stcDmcInit.stcChipCfg.u32MemBurst)
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_BURST_LEN_1;
    }
    else if (EXMC_DMC_MEM_BURST_2 == stcDmcInit.stcChipCfg.u32MemBurst)
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_BURST_LEN_2;
    }
    else if (EXMC_DMC_MEM_BURST_4 == stcDmcInit.stcChipCfg.u32MemBurst)
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_BURST_LEN_4;
    }
    else
    {
      u32MdRegVal |= IS42S16400J7TLI_MR_BURST_LEN_8;
    }

    EV_EXMC_DMC_InitSequence(IS42S16400J7TLI_MAP_DMC_CHIP, EXMC_DMC_BANK_0,
u32MdRegVal);

    /* Switch state from configure to ready */
    EXMC_DMC_SetState(EXMC_DMC_CTL_STATE_GO);
    EXMC_DMC_SetState(EXMC_DMC_CTL_STATE_WAKEUP);
    EXMC_DMC_SetState(EXMC_DMC_CTL_STATE_GO);

    /* Check status */
    while (EXMC_DMC_CURR_STATUS_READY != EXMC_DMC_GetStatus())
    {}

    return Ok;
  }
```

2) SDRAM 样例主程序：

```
int32_t main(void)
{
    uint32_t i;
    uint32_t j;
    uint32_t u32MemAddr;
    uint32_t u32ToggleCnt = 0UL;

    /* MCU Peripheral registers write unprotected */
    Peripheral_WE();

    /* Initialize system clock: */
    BSP_CLK_Init();

    /* PCLK0, HCLK  Max 240MHz */
    /* PCLK1, PCLK4 Max 120MHz */
    /* PCLK2, PCLK3 Max 60MHz  */
    /* EXCLK 60MHz */
    CLK_ClkDiv(CLK_CATE_ALL, (CLK_PCLK0_DIV1 | CLK_PCLK1_DIV2 | \
                        CLK_PCLK2_DIV4 | CLK_PCLK3_DIV4 | \
                        CLK_PCLK4_DIV2 | CLK_EXCLK_DIV4 | CLK_HCLK_DIV1));

    /* Initialize UART print */
    (void)DDL_PrintfInit();

    /* Initialize LED */
    BSP_IO_Init();
    BSP_LED_Init();

    /* Initialize test data. */
    for (i = 0UL; i < DATA_BUFFER_LEN; i++)
    {
        m_au8ReadData[i] = 0U;
        m_au8WriteData[i] = 0x12U;

        m_au16ReadData[i] = 0U;
        m_au16WriteData[i] = 0x5678U;

        m_au32ReadData[i] = 0UL;
        m_au32WriteData[i] = 0xAABBCCDDUL;
    }

    /* Configure SRAM. */
    (void)IS42S16400J7TLI_Init();
    (void)IS42S16400J7TLI_GetMemInfo(&m_u32MemStartAddr, &m_u32MemByteSize);

    /* MCU Peripheral registers write protected */
    Peripheral_WP();

    m_u32MemHalfwordSize = m_u32MemByteSize/2UL;
    m_u32MemWordSize = m_u32MemByteSize/4UL;

    (void)printf("Memory start address: 0x%.8x \r\n", (unsigned int)m_u32MemStartAddr);
    (void)printf("Memory end   address: 0x%.8x \r\n", (unsigned int)(m_u32MemStartAddr +
m_u32MemByteSize - 1UL));
    (void)printf("Memory size  (Bytes): 0x%.8x \r\n\r\n", (unsigned int)m_u32MemByteSize);
```

```
    for (;;)
    {
        m_u32TestCnt++;
        (void)printf("******** Write/read test times: %u ********\r\n", (unsigned int)m_u32TestCnt);

        /****************** Byte read/write *****************/
        m_u32ByteTestErrorCnt = 0UL;
        u32MemAddr = m_u32MemStartAddr;
        for (i = 0UL; i < m_u32MemByteSize; i += DATA_BUFFER_LEN)
        {
            (void)IS42S16400J7TLI_WriteMem8(u32MemAddr, m_au8WriteData, DATA_BUFFER_LEN);
            (void)IS42S16400J7TLI_ReadMem8(u32MemAddr, m_au8ReadData, DATA_BUFFER_LEN);

            /* Verify write/read data. */
            for (j = 0UL; j < DATA_BUFFER_LEN; j++)
            {
                if (m_au8WriteData[j] != m_au8ReadData[j])
                {
                    m_u32ByteTestErrorCnt++;
                    (void)printf("Byte read/write error: address = 0x%.8x; write data = 0x%x; read data =
0x%x\r\n",
                        (unsigned int)(u32MemAddr+j), (unsigned int)m_au8WriteData[j], (unsigned
int)m_au8ReadData[j]);
                }
            }

            u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au8ReadData[0]));
            (void)memset(m_au8ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au8ReadData[0])));
            LED_TOGGLE((u32ToggleCnt++ % 5UL) == 0UL);
        }

        (void)printf("     Byte read/write error data count: %u \r\n", (unsigned int)m_u32ByteTestErrorCnt);

        /***************** Halfword read/write *****************/
        m_u32HalfwordTestErrorCnt = 0UL;
        u32MemAddr = m_u32MemStartAddr;
        for (i = 0UL; i < m_u32MemHalfwordSize; i += DATA_BUFFER_LEN)
        {
            (void)IS42S16400J7TLI_WriteMem16(u32MemAddr, m_au16WriteData,
DATA_BUFFER_LEN);
            (void)IS42S16400J7TLI_ReadMem16(u32MemAddr, m_au16ReadData, DATA_BUFFER_LEN);

            /* Verify write/read data. */
            for (j = 0UL; j < DATA_BUFFER_LEN; j++)
            {
                if (m_au16WriteData[j] != m_au16ReadData[j])
                {
                    m_u32HalfwordTestErrorCnt++;
                    (void)printf("Halfword read/write error: address = 0x%.8x; write data = 0x%x; read data =
0x%x\r\n",
                        (unsigned int)(u32MemAddr+j), (unsigned int)m_au16WriteData[j], (unsigned
int)m_au16ReadData[j]);
                }
            }

            u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au16ReadData[0]));
```

```
            (void)memset(m_au16ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au16ReadData[0])));
            LED_TOGGLE((u32ToggleCnt++ % 5UL) == 0UL);
        }

        (void)printf(" Halfword read/write error data count: %u \r\n", (unsigned
int)m_u32HalfwordTestErrorCnt);

        /****************** Word read/write ******************/
        m_u32WordTestErrorCnt = 0UL;
        u32MemAddr = m_u32MemStartAddr;
        for (i = 0UL; i < m_u32MemWordSize; i += DATA_BUFFER_LEN)
        {
            (void)IS42S16400J7TLI_WriteMem32(u32MemAddr, m_au32WriteData,
DATA_BUFFER_LEN);
            (void)IS42S16400J7TLI_ReadMem32(u32MemAddr, m_au32ReadData, DATA_BUFFER_LEN);

            /* Verify write/read data. */
            for (j = 0UL; j < DATA_BUFFER_LEN; j++)
            {
                if (m_au32WriteData[j] != m_au32ReadData[j])
                {
                    m_u32WordTestErrorCnt++;
                    (void)printf("Word read/write error: address = 0x%.8x; write data = 0x%.8x; read data =
0x%.8x\r\n",
                            (unsigned int)(u32MemAddr+j), (unsigned int)m_au32WriteData[j], (unsigned
int)m_au32ReadData[j]);
                }
            }

            u32MemAddr += (DATA_BUFFER_LEN * sizeof(m_au32ReadData[0]));
            (void)memset(m_au32ReadData, 0, (DATA_BUFFER_LEN * sizeof(m_au32ReadData[0])));
            LED_TOGGLE((u32ToggleCnt++ % 5UL) == 0UL);
        }

        (void)printf("     Word read/write error data count: %u \r\n\r\n", (unsigned
int)m_u32WordTestErrorCnt);

        /****************** Error check ******************/
        if ((m_u32ByteTestErrorCnt > 0UL)     || \
            (m_u32HalfwordTestErrorCnt > 0UL) || \
            (m_u32WordTestErrorCnt > 0UL))
        {
            BSP_LED_On(LED_RED);
        }
    }
  }
```

3) SDRAM 8Bit 读写操作：

```
en_result_t IS42S16400J7TLI_WriteMem8(uint32_t u32Address,
                        const uint8_t au8SrcBuffer[],
                        uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (0UL != u32BufferSize)
   {
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + u32BufferSize) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         *(uint8_t *)(u32Address + i) = au8SrcBuffer[i];
      }
      enRet = Ok;
   }

   return enRet;
}

en_result_t IS42S16400J7TLI_ReadMem8(uint32_t u32Address,
                        uint8_t au8DstBuffer[],
                        uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if ((NULL != au8DstBuffer) && (0UL != u32BufferSize))
   {
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + u32BufferSize) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         au8DstBuffer[i] = *(uint8_t *)(u32Address + i);
      }
      enRet = Ok;
   }

   return enRet;
}
```

4) SDRAM 16Bit 读写操作：

```
en_result_t IS42S16400J7TLI_WriteMem16(uint32_t u32Address,
                                       const uint16_t au16SrcBuffer[],
                                       uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (0UL != u32BufferSize)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_HALFWORD(u32Address));
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 1UL)) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         *((uint16_t *)u32Address) = au16SrcBuffer[i];
         u32Address += 2UL;
      }
      enRet = Ok;
   }

   return enRet;
}

en_result_t IS42S16400J7TLI_ReadMem16(uint32_t u32Address,
                                      uint16_t au16DstBuffer[],
                                      uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if ((NULL != au16DstBuffer) && (0UL != u32BufferSize))
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_HALFWORD(u32Address));
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 1UL)) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         au16DstBuffer[i] = *((uint16_t *)u32Address);
         u32Address += 2UL;
      }
      enRet = Ok;
   }

   return enRet;
}
```

5) SDRAM 32Bit 读写操作：

```
en_result_t IS42S16400J7TLI_WriteMem32(uint32_t u32Address,
                        const uint32_t au32SrcBuffer[],
                        uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (0UL != u32BufferSize)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_WORD(u32Address));
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 2UL)) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         *((uint32_t *)u32Address) = au32SrcBuffer[i];
         u32Address += 4UL;
      }
      enRet = Ok;
   }

   return enRet;
}

en_result_t IS42S16400J7TLI_ReadMem32(uint32_t u32Address,
                        uint32_t au32DstBufferBuf[],
                        uint32_t u32BufferSize)
{
   uint32_t i;
   en_result_t enRet = ErrorInvalidParameter;

   if (NULL != au32DstBufferBuf)
   {
      DDL_ASSERT(IS_ADDRESS_ALIGN_WORD(u32Address));
      DDL_ASSERT(u32Address >= IS42S16400J7TLI_START_ADDRESS);
      DDL_ASSERT((u32Address + (u32BufferSize << 2UL)) <= IS42S16400J7TLI_END_ADDRESS);

      for (i = 0UL; i < u32BufferSize; i++)
      {
         au32DstBufferBuf[i] = *((uint32_t *)u32Address);
         u32Address += 4UL;
      }
      enRet = Ok;
   }

   return enRet;
}
```

4.3 NFC 接口 8 位 NAND Flash 存储器

4.3.1 硬件连接

本文中使用评估用板（EV_F4A0_LQ176_V10）MT29F2G08AB 存储器作为 NFC 例子说明。

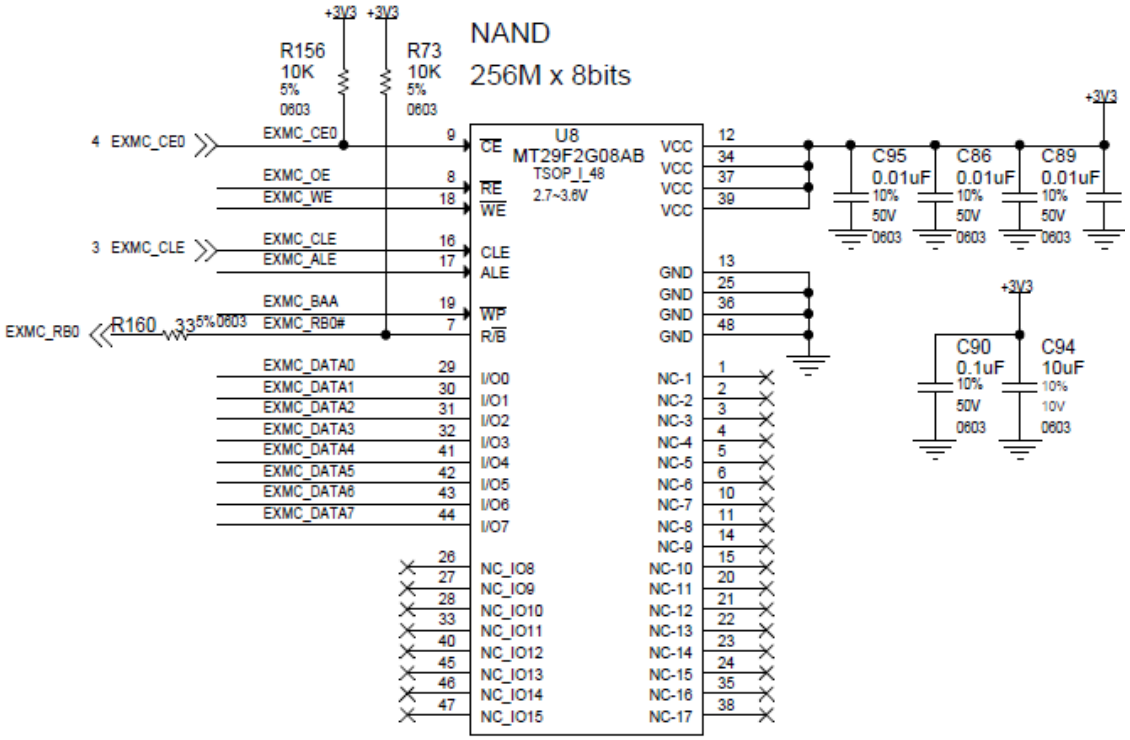


图 4-5 8 位 NAND Flash：MT29F2G08AB 连接至 HC32F4A0 的 EXMC 管脚

NAND Flash 信号与 EXMC 管脚的对应如下表所示。

存储器信号	EXMC 管脚	信号说明
I/O[7:0]	EXMC_DATA[7:0]	数据线 D0 至 D7
$\overline{CE}$	EXMC_CS0	芯片使能
$\overline{WE}$	EXMC_WE	写使能
$\overline{RE}$	EXMC_OE	读使能
CLE	EXMC_CLE	命令锁存使能
ALE	EXMC_ALE	地址锁存使能
$\overline{WP}$	EXMC_BAA	写保护
R/$\overline{B}$	EXMC_R$\overline{B}$0	就绪繁忙信号

表 4-7 MT29F2G08AB 与 EXMC 管脚的对应

4.3.2 NFC 配置

MT29F2G08AB 容量为 2G bits、数据总线位宽为 8 位的存储器。

NFC 配置如下：

- 芯片使能：Chip0
- 数据总线为 8 位宽：NFC_BACR 寄存器 16BIT 设置为’0’
- Bank 数为 1：NFC_BACR 寄存器 BANK 设置为’0’
- Page 为 248 字节：NFC_BACR 寄存器 PAGE 设置为’01’
- 行地址周期为 3： NFC_BACR 寄存器 RAC 设置为’1’
- 保持其它的所有参数为清除状态

4.3.3 时序配置

MT29F2G08AB 时序参数参数如下：

时间参数	时间要求	说明
tALH/tCH/ tCLH	不小于 5ns	CLE/ALE/CE在WE/RE变化后的保持时间
tALS/ tCLS/	不小于 10ns	CLE/ALE在WE/RE变化前的建立时间
tCS	不小于 15ns	CE在WE/RE变化前的建立时间
tWP	不小于 10ns	WE的有效(低电平)脉冲宽度
tRP	不小于10ns	RE的有效(低电平)脉冲宽度
tWH	不小于7ns	WE的无效(高电平)脉冲宽度
tREH	不小于7ns	RE的无效(高电平)脉冲宽度
tRR	不小于20ns	RB上升沿到RE下降沿的时间
tWB	不超过100ns	WE上升沿到RB下降沿的时间
tADL	不小于70ns	ALE到读写数据的时间
tRHW	不小于100ns	RE上升沿到WE下降沿的时间
tWHR	不小于80ns	WE上升沿到RE下降沿的时间

表 4-8 MT29F2G08AB（3.3V）的时间参数

系统工作电压为 3.3V，时钟为 240MHz，EXMC 总线时钟为 60MHz，使用上述存储器的时序参数，可以设置 EXMC NFC 时序参数如下：

NFC_TMCR0 寄存器 TS 位为 4；

NFC_TMCR0 寄存器 TWP 位为 3；

NFC_TMCR0 寄存器 TRP 位为 6；

NFC_TMCR0 寄存器 TH 位为 2UL；

NFC_TMCR1 寄存器 TWH 位为 2；

NFC_TMCR1 寄存器 TRH 位为 3；

NFC_TMCR1 寄存器 TRR 位为 8；

NFC_TMCR1 寄存器 TWB 位为 8；

NFC_TMCR2 寄存器 TCCS 位为 10；

NFC_TMCR2 寄存器 TWTR 位为 13；

NFC_TMCR2 寄存器 TRTW 位为 8；

NFC_TMCR2 寄存器 TADL 位为 10；

4.3.4 程序示例

NAND Flash 操作通过设备驱动库（Device Driver Library, DDL）的样例 exmc_nfc_nandflash_mt29f2g08ab 展示。下面主要介绍比较关键的代码。

1) 根据 MT29F2G08AB 时序，修改 BSP 初始化配置参数，初始化 NAND Flash。

```
en_result_t BSP_NFC_MT29F2G08AB_Init(void)
{
    en_result_t enRet = Error;
    stc_exmc_nfc_init_t stcInit;

    /* Initialize NFC port.*/
    EV_EXMC_NFC_PortInit();

    /* Enable NFC module clk */
    PWC_Fcg3PeriphClockCmd(PWC_FCG3_NFC, Enable);

    /* Enable NFC. */
    EXMC_NFC_Cmd(Enable);

    /* Configure NFC width && refresh period & chip & timing. */
    stcInit.u32OpenPage = EXMC_NFC_OPEN_PAGE_DISABLE;
    stcInit.stcBaseCfg.u32CapacitySize = EXMC_NFC_BANK_CAPACITY_2GBIT;
    stcInit.stcBaseCfg.u32MemWidth = EXMC_NFC_MEMORY_WIDTH_8BIT;
    stcInit.stcBaseCfg.u32BankNum = EXMC_NFC_1_BANK;
    stcInit.stcBaseCfg.u32PageSize = EXMC_NFC_PAGE_SIZE_2KBYTES;
    stcInit.stcBaseCfg.u32WrProtect = EXMC_NFC_WR_PROTECT_DISABLE;
    stcInit.stcBaseCfg.u32EccMode = EXMC_NFC_ECC_1BIT;
    stcInit.stcBaseCfg.u32RowAddrCycle = EXMC_NFC_3_ROW_ADDRESS_CYCLES;
    stcInit.stcBaseCfg.u8SpareSizeForUserData = 0U;

    stcInit.stcTimingReg0.u32TS = 4UL;        /* CLOCK frequency @60MHz: 3.3V */
    stcInit.stcTimingReg0.u32TWP = 3UL;
    stcInit.stcTimingReg0.u32TRP = 6UL;
    stcInit.stcTimingReg0.u32TH = 2UL;

    stcInit.stcTimingReg1.u32TWH = 2UL;
    stcInit.stcTimingReg1.u32TRH = 3UL;
    stcInit.stcTimingReg1.u32TRR = 8UL;
    stcInit.stcTimingReg1.u32TWB = 8UL;

    stcInit.stcTimingReg2.u32TCCS = 10UL;
    stcInit.stcTimingReg2.u32TWTR = 0x0DUL;
```

```
    stcInit.stcTimingReg2.u32TRTW = 8UL;
    stcInit.stcTimingReg2.u32TADL = 10UL;
    if (Ok == EXMC_NFC_Init(&stcInit))
    {
        /* Reset NFC device. */
        if(Ok == EXMC_NFC_Reset(BSP_EV_HC32F4A0_MT29F2G08AB_BANK,
BSP_NFC_RESET_TIMEOUT))
        {
            enRet = Ok;
        }
    }

    return enRet;
}
```

2) NAND Flash 样例主程序。

```
int32_t main(void)
{
    uint8_t au8DevId[4];
    uint8_t u8TestErrCnt = 0U;

    /* MCU Peripheral registers write unprotected */
    Peripheral_WE();

    /* Initialize system clock: */
    BSP_CLK_Init();

    /* PCLK0, HCLK  Max 240MHz */
    /* PCLK1, PCLK4 Max 120MHz */
    /* PCLK2, PCLK3 Max 60MHz  */
    /* EXCLK 60MHz */
    CLK_ClkDiv(CLK_CATE_ALL, (CLK_PCLK0_DIV1 | CLK_PCLK1_DIV2 | \
                              CLK_PCLK2_DIV4 | CLK_PCLK3_DIV4 | \
                              CLK_PCLK4_DIV2 | CLK_EXCLK_DIV4 | CLK_HCLK_DIV1));

    /* Initialize LED */
    BSP_IO_Init();
    BSP_LED_Init();

    /* Configure nandflash */
    (void)MT29F2G08AB_Init();

    /* MCU Peripheral registers write protected */
    Peripheral_WP();

    /* Read ID */
    (void)MT29F2G08AB_ReadId(0UL, au8DevId, sizeof(au8DevId));
    if ((au8DevId[0] == MT29F2G08ABAEA_MANUFACTURER_ID) || \
        (au8DevId[1] == MT29F2G08ABAEA_DEVICE_ID1) || \
        (au8DevId[2] == MT29F2G08ABAEA_DEVICE_ID2) || \
        (au8DevId[3] == MT29F2G08ABAEA_DEVICE_ID3))
    {
        /* Erase nandflash. */
        if (Ok == MT29F2G08AB_EraseBlock(0UL))
        {
```

```
            if (Ok != MT29F2G08AB_MetaWithoutSpareTest(0UL))
            {
               u8TestErrCnt++;
            }

            if (Ok != MT29F2G08AB_MetaWithSpareTest(1UL))
            {
               u8TestErrCnt++;
            }

            if (Ok != MT29F2G08AB_HwEcc1BitTest(2UL))
            {
               u8TestErrCnt++;
            }

            if (Ok != MT29F2G08AB_HwEcc4BitsTest(3UL))
            {
               u8TestErrCnt++;
            }
         }
         else
         {
            u8TestErrCnt++;
         }
      }
      else
      {
         u8TestErrCnt++;
      }

      if (u8TestErrCnt > 0U)
      {
         BSP_LED_On(LED_RED);
      }
      else
      {
         BSP_LED_On(LED_BLUE);
      }

      for (;;)
      {
      }
   }
```

3) NAND Flash 数据区读写操作。

```
   static en_result_t MT29F2G08AB_MetaWithoutSpareTest(uint32_t u32Page)
   {
      en_result_t enRet = Error;

      __ALIGN_BEGIN static uint8_t
m_au8ReadDataMeta[MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE];
      __ALIGN_BEGIN static uint8_t
m_au8WriteDataMeta[MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE];

      /* Initialize data. */
```

```
    for (uint32_t i = 0U; i < MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE; i++)
    {
        m_au8WriteDataMeta[i] = (uint8_t)i;
    }

    /* Write page: 2048Bytes */
    if (Ok == MT29F2G08AB_WritePageMeta(u32Page, m_au8WriteDataMeta,
MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE))
    {
        /* Read page: 2048Bytes */
        if (Ok == MT29F2G08AB_ReadPageMeta(u32Page, m_au8ReadDataMeta,
MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE))
        {
            /* Verify write/read data. */
            if (0 == memcmp (m_au8WriteDataMeta, m_au8ReadDataMeta,
MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE))
            {
                enRet = Ok;
            }
        }
    }

    return enRet;
}
```

4) NAND Flash 数据区和冗余区读写操作。

```
static en_result_t MT29F2G08AB_MetaWithSpareTest(uint32_t u32Page)
{
    en_result_t enRet = Error;

    __ALIGN_BEGIN static uint8_t
m_au8ReadDataMetaWithSpare[MT29F2G08AB_PAGE_SIZE_WITH_SPARE];
    __ALIGN_BEGIN static uint8_t
m_au8WriteDataMetaWithSpare[MT29F2G08AB_PAGE_SIZE_WITH_SPARE];

    /* Initialize data. */
    for (uint32_t i = 0U; i < MT29F2G08AB_PAGE_SIZE_WITH_SPARE; i++)
    {
        m_au8WriteDataMetaWithSpare[i] = (uint8_t)i;
    }

    /* Write page: 2048 + 64Bytes */
    if (Ok == MT29F2G08AB_WritePageMeta(u32Page, m_au8WriteDataMetaWithSpare,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE))
    {
        /* Read page: 2048 + 64Bytes */
        if (Ok == MT29F2G08AB_ReadPageMeta(u32Page, m_au8ReadDataMetaWithSpare,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE))
        {
            /* Verify write/read data. */
            if (0 == memcmp (m_au8WriteDataMetaWithSpare, m_au8ReadDataMetaWithSpare,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE))
            {
                enRet = Ok;
            }
```

```
        }
    }

    return enRet;
}
```

5) NADN Flash 读写操作（1Bit ECC 校验）。

```
static en_result_t MT29F2G08AB_HwEcc1BitTest(uint32_t u32Page)
{
    en_result_t enRet = Error;
    uint32_t u32EccTestResult;
    uint32_t u32EccExpectedResult;

    __ALIGN_BEGIN static uint8_t
m_au8ReadDataHwEcc[MT29F2G08AB_PAGE_SIZE_WITH_SPARE];
    __ALIGN_BEGIN static uint8_t
m_au8WriteDataHwEcc[MT29F2G08AB_PAGE_SIZE_WITH_SPARE];
    __ALIGN_BEGIN static uint8_t
m_au8SwEcc1Bit[MT29F2G08AB_PAGE_1BIT_ECC_VALUE_SIZE];

    /* Initialize data. */
    for (uint32_t i = 0UL; i < MT29F2G08AB_PAGE_SIZE_WITH_SPARE; i++)
    {
        m_au8WriteDataHwEcc[i] = (uint8_t)(i);
    }

    /* Enable ECC 1bit: write 2048Bytes */
    if (Ok == MT29F2G08AB_WritePageHwEcc1Bit(u32Page, \
                                m_au8WriteDataHwEcc, \
                                MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE))
    {
        /* Disable ECC 1bit: read 2048 + 64Bytes */
        (void)MT29F2G08AB_ReadPageMeta(u32Page, m_au8ReadDataHwEcc,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE);

        /* Software calculate ECC 1bit */
        (void)NFC_SwCalculateEcc1Bit(&m_au8WriteDataHwEcc[0], &m_au8SwEcc1Bit[0]);
        (void)NFC_SwCalculateEcc1Bit(&m_au8WriteDataHwEcc[512], &m_au8SwEcc1Bit[3]);
        (void)NFC_SwCalculateEcc1Bit(&m_au8WriteDataHwEcc[1024], &m_au8SwEcc1Bit[6]);
        (void)NFC_SwCalculateEcc1Bit(&m_au8WriteDataHwEcc[1536], &m_au8SwEcc1Bit[9]);

        /* Compare software & hardware calculating ECC result */
        if (0 == memcmp (m_au8SwEcc1Bit, \
                &m_au8ReadDataHwEcc[MT29F2G08AB_PAGE_SIZE_WITH_SPARE -
MT29F2G08AB_PAGE_1BIT_ECC_VALUE_SIZE], \
                MT29F2G08AB_PAGE_1BIT_ECC_VALUE_SIZE))
        {
            /* Modify the 2nd byte value from 0x01 to 0x00 */
            m_au8ReadDataHwEcc[1] = 0x00U;

            /* Single bit error: the 2nd byte - bit 0 */
            u32EccExpectedResult = ((0UL << EXMC_NFC_1BIT_ECC_ERR_BIT_POS) | \
                            (1UL << EXMC_NFC_1BIT_ECC_ERR_BYTE_POS) | \
                            EXMC_NFC_1BIT_ECC_SINGLE_BIT_ERR);
```

```
            /* Disable ECC 1bit: write 2048 + 64Bytes */
            (void)MT29F2G08AB_WritePageMeta(u32Page, m_au8ReadDataHwEcc,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE);

            /* Enable ECC 1bit: read 2048Bytes */
            (void)MT29F2G08AB_ReadPageHwEcc1Bit(u32Page, \
                                m_au8ReadDataHwEcc, \
                                MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE);

            /* Get ECC result */
            u32EccTestResult = EXMC_NFC_GetEcc1BitResult(EXMC_NFC_ECC_SECTION0);
            if (u32EccExpectedResult == u32EccTestResult)
            {
               enRet = Ok; /* The result meets the expected */
            }
         }
      }

      return enRet;
   }
```

6) NADN Flash 读写操作（4Bit ECC 校验）。

```
   static en_result_t MT29F2G08AB_HwEcc4BitsTest(uint32_t u32Page)
   {
      en_result_t enRet = Error;
      uint16_t au16SynVal[8];
      int16_t ai16EccTestErrByteNumber[4];
      int16_t ai16EccTestErrByteBit[4];
      int16_t ai16EccExpectedErrByteNumber[4];
      int16_t ai16EccExpectedErrByteBit[4];

      __ALIGN_BEGIN static uint8_t
m_au8ReadDataHwEcc[MT29F2G08AB_PAGE_SIZE_WITH_SPARE];
      __ALIGN_BEGIN static uint8_t
m_au8WriteDataHwEcc[MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE];

      /* Initialize data. */
      for (uint32_t i = 0UL; i < MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE; i++)
      {
         m_au8WriteDataHwEcc[i] = (uint8_t)(i);
      }

      EXMC_NFC_SetEccMode(EXMC_NFC_ECC_4BITS);

      /* Enable ECC 4bit: write 2048Bytes */
      if (Ok == MT29F2G08AB_WritePageHwEcc4Bits(u32Page, \
                              m_au8WriteDataHwEcc, \
                              MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE))
      {
         /* Enable ECC 4bit: read 2048Bytes */
         (void)MT29F2G08AB_ReadPageHwEcc4Bits(u32Page, \
                              m_au8ReadDataHwEcc, \
                              MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE);

         /* Check whether ECC errors occur */
```

```
        if (EXMC_NFC_GetStatus(EXMC_NFC_FLAG_ECC_ERROR) == Reset)
        {
          /* Disable ECC 4bit: read 2048 + 64Bytes */
          (void)MT29F2G08AB_ReadPageMeta(u32Page, m_au8ReadDataHwEcc,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE);

          /* Modify data to create error */
          m_au8ReadDataHwEcc[254] = 0xAEU;  /* Modify the 254th byte value from 0xFE to 0xAE */
          m_au8ReadDataHwEcc[255] = 0xF5U;  /* Modify the 255th byte value from 0xFF to 0xF5 */

          /* Error bit: the 254th byte - 2 bit */
          ai16EccExpectedErrByteNumber[0] = 254;
          ai16EccExpectedErrByteBit[0] = 4;

          /* Error bit: the 254th byte - 3 bit */
          ai16EccExpectedErrByteNumber[1] = 254;
          ai16EccExpectedErrByteBit[1] = 6;

          /* Error bit: the 255th byte - 1 bit */
          ai16EccExpectedErrByteNumber[2] = 255;
          ai16EccExpectedErrByteBit[2] = 1;

          /* Error bit: the 255th byte - 3 bit */
          ai16EccExpectedErrByteNumber[3] = 255;
          ai16EccExpectedErrByteBit[3] = 3;

          /* Disable ECC 4bit: write 2048 + 64Bytes */
          (void)MT29F2G08AB_WritePageMeta(u32Page, m_au8ReadDataHwEcc,
MT29F2G08AB_PAGE_SIZE_WITH_SPARE);

          /* Enable ECC 4bit: read 2048Bytes */
          (void)MT29F2G08AB_ReadPageHwEcc4Bits(u32Page, \
                                m_au8ReadDataHwEcc, \
                                MT29F2G08AB_PAGE_SIZE_WITHOUT_SPARE);

          /* Get ECC syndrome */
          (void)EXMC_NFC_GetSyndrome(EXMC_NFC_ECC_SECTION0, au16SynVal, 8U);

          /* Decode ECC errors location */
          if (4 == NFC_SwDecodeEcc4BitsErrLocation((const int16_t *)((uint32_t)(&au16SynVal)), \
                    ai16EccTestErrByteNumber, ai16EccTestErrByteBit, 4))
          {
            if ((ai16EccExpectedErrByteBit[0] == ai16EccTestErrByteBit[0]) && \
              (ai16EccExpectedErrByteBit[1] == ai16EccTestErrByteBit[1]) && \
              (ai16EccExpectedErrByteBit[2] == ai16EccTestErrByteBit[2]) && \
              (ai16EccExpectedErrByteBit[3] == ai16EccTestErrByteBit[3]) && \
              (ai16EccExpectedErrByteNumber[0] == ai16EccTestErrByteNumber[0]) && \
              (ai16EccExpectedErrByteNumber[1] == ai16EccTestErrByteNumber[1]) && \
              (ai16EccExpectedErrByteNumber[2] == ai16EccTestErrByteNumber[2]) && \
              (ai16EccExpectedErrByteNumber[3] == ai16EccTestErrByteNumber[3]))
            {
              enRet = Ok; /* The result meets the expected */
            }
          }
        }
    }
```

```
    return enRet;
}
```

5 总结

以上章节简要介绍 HC32F4A0 系列的 EXMC SMC、DMC 和 NFC 的使用方法。样例代码演示了如何根据实际硬件及存储器时序参数配置 EXMC。在开发中用户可以根据使用的存储芯片参数，配置 EXMC 模块。

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：http://www.hdsc.com.cn/mcu.htm

通信地址：上海市浦东新区中科路 1867 号 A 座 10 层

邮编：201203