APPLICATION NOTE

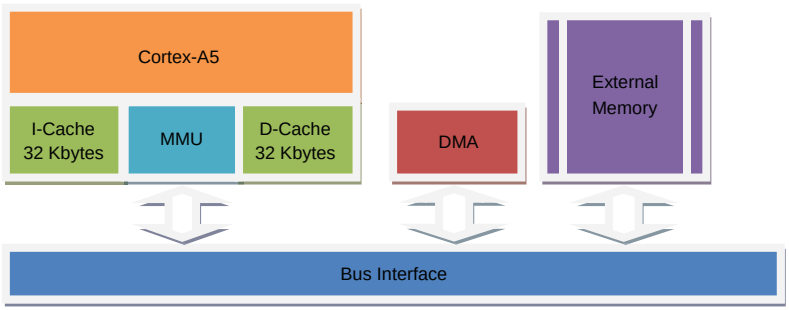
MMU Cache Coherence on SAMA5D3 Products in a Non-Linux System

Atmel | SMART SAMA5D3 Series

Introduction

The Atmel® | SMART SAMA5D3 device is equipped with a 32-Kbyte data cache and a 32-Kbyte instruction cache. It also includes interfaces to enable connection to a wide range of external memories or to parallel peripherals. External memories can be accessed directly by the device using DMA to increase system performance. However, in such case, care must be taken to maintain coherence between the data cache and the buffer filled via DMA transfer.

Figure 1. MMU and Cache



This Application Note describes how to address the issue of cache coherence with software-based solutions. The examples provided are taken mainly from the SAMA5D3 software package.

Reference Documents

- ARM® Architecture Reference Manual
- IAR C/C++ Development Guide Compiling and Linking for Advanced RISC Machines Ltd's ARM Cores
- The GNU linker ld (Sourcery G++ Lite 2011.03-42) Version 2.20.51

Table of Contents

1. Memory Management Unit (MMU)

1.1 About the MMU

The Cortex-A5 MMU is an ARM-Architecture v7 MMU. The Cortex-A5 processor operates using virtual addresses (VAs). The Memory Management Unit (MMU) translates these VAs into physical addresses (PAs) which are used to access the memory system.

For more information, see the *ARM Architecture Reference Manual*.

1.2 Cortex-A5 MMU Feature

The MMU generates physical address locations from the virtual addresses that the processor generates, and it is able to control program access to memory, and to configure the memory attributes for the translation lookaside table.

The maintenance and configuration operations of the SAMA5D3 TLB (Translation Lookaside Buffer) are controlled through a dedicated coprocessor, CP15, integrated to the core. This coprocessor provides a standard mechanism for configuring the L1 cache memory system.

1.3 Translation Lookaside Table

All relevant CP15 registers must be programmed before the MMU is enabled. This includes setting up suitable translation lookaside tables in the memory.

For more information on how to configure a translation lookaside table, please see Section “Translation Lookaside Table” in *ARM Architecture Reference Manual*.

1.3.1 TLB Configuration Example in Software Package

To simplify the usage of the MMU, the goal of the SAMA5D3 software package is to create a simple identity mapping across the entire address space between the virtual and physical memory addresses, which means that the address space from the CPU’s point of view remains the same after the MMU has been enabled.

The ARM MMU supports entries in the translation lookaside tables, which can represent 1 Mbyte (section), 64 Kbytes (large page), 4 Kbytes (small page) or 1 Kbyte (tiny page) of virtual memory.

When the MMU is enabled, it is able to automatically convert virtual addresses into physical addresses. However, a set of translation lookaside tables must be stored in the physical memory.

1.3.1.1 Example Code for TLB Configuration

To create an identity mapping, the example code configures each entry in the table. Therefore, 4096 entries which point to a corresponding range of physical memory are the same virtual addresses.

```
/* Initializes MMU with the start address of the translation lookaside table */
MMU_Initialize((uint32_t *)0x30C000);
```

In order to create an identity mapping, the values used for the section base address should start with 0x000 for the first entry and then be incremented by 1 MByte as follows:

pTB [000] (virtual address) = 0x000xxxxx (due to entry 0 for ROM boot memory)

pTB [001] (virtual address) = 0x001xxxxx (due to entry 1 for ROM space)

pTB [002] (virtual address) = 0x002xxxxx (due to entry2 for NFC SRAM)

pTB [003] (virtual address) = 0x003xxxxx (due to entry3 for SRAM)

…

pTB [F00] (virtual address) = 0xF00xxxxx (due to entryn for internal peripherals)

The next step is to populate the tables with associated attributes.

1.3.1.2 TLB Example for DDRAM with Cacheable Attributes

```
/* Section DDRAM with cacheable attributes */
/* Memory address 0x2000_0000 */
pTB[0x200] = (0x200<< 20)| // Physical Address equals virtual address
             ( 3 << 10)|   // Access in supervisor mode (AP)
             ( 0xF <<  5)| // Domain 0xF
             ( 1 <<  4)|   // (XN)
             ( 1 <<  3)|   // C bit : cacheable => YES
             ( 1 <<  2)|   // B bit : write-back => YES
             ( 2 <<  0);   // Set as 1 Mbyte section
```

This table sets the 1-Mbyte memory (from 0x2000000) with cacheable attributes.

1.3.1.3 TLB Example for DDRAM with Non-Cacheable Attributes

```
/* Section DDRAM with non-cacheable attributes */
/* Memory address 0x2300_0000 */
pTB[0x230] =(0x230<< 20)| // Physical Address equals virtual address
      ( 3 << 10)|   // Access in supervisor mode (AP)
      ( 0xF <<  5)| // Domain 0xF
      ( 1 <<  4)|   // (XN)
      ( 0 <<  3)|   // C bit : cacheable => No
      ( 0 <<  2)|   // B bit : write-back => No
      ( 2 <<  0);   // Set as 1 Mbyte section
```

This table sets the 1-Mbyte memory (from 0x2300000) with non-cacheable attributes.

One more concept, Domains, should be understood. Each domain has an attribute that allows users to control the access to its associated pages. In this example, we assigned the Page Table entry to domain 0xF, and set it to “Manager”, which means that access permissions are not checked.

1.4 MMU Enabling and Disabling

The MMU can be enabled or disabled by setting/clearing the M-bit in the CP15 c1 control register.

For more information, see Section “Memory System Control Register” in the *ARM Architecture Reference Manual.*

- Enable MMU

```
/*  Enable MMU */
CP15_EnableMMU();
```

- Disable MMU

```
/* Disable MMU */
CP15_DisableMMU();
```

2. Cache

The Cortex-A5 processor uses separate caches for instructions and data. In brief, the cache memory is an on-chip memory that can be accessed very quickly by the CPU. Access to the cache memory is usually faster than access to the off-chip memory. This fast memory acts as a temporary storage for code or data variables that are accessed on a repeated basis, and thereby helps improving system performance.

2.1 Instruction Cache (I-Cache)

The instruction cache holds the instructions of the running programs that the processor is required to execute. The SAMA5D3 device includes a 32-Kbyte I-Cache. The I-Cache caches fetched instructions to be executed by the processor.

2.1.1 I-Cache Enabling and Disabling

The I-Cache can be enabled or disabled by setting/clearing the I-bit in the CP15 c1 control register.

For more information, see Section “Memory System Control Register” in the *ARM Architecture Reference Manual*.

- Enable I-Cache

```
/*  Enable I-Cache */
CP15_EnableIcache();
```

- Disable I-Cache

```
/*  Disable I-Cache */
CP15_DisableIcache();
```

2.2 Data Cache (D-Cache)

The SAMA5D3 device includes a 32-Kbyte D-Cache.

The cache memory is used by the processor to speed up the average memory access time. The data cache holds the data being used by current instructions. The data cache provides improvement to system performance by hiding the memory access for data that has already been accessed.

The system coprocessor (CP15) controller performs cache maintenance operations directly on the data cache.

Note: The Cortex-A5 data cache only supports a write-back policy.

2.2.1 D-Cache Enabling and Disabling

The D-Cache can be enabled or disabled by setting/clearing the C-bit in the CP15 c1 control register.

For more information, see Section “Memory System Control Register” in the *ARM Architecture Reference Manual.*

- Enable D-Cache

```
/*  Enable D-Cache */
CP15_EnableDcache();
```

- Disable D-Cache

```
/*  Disable D-Cache */
CP15_DisableDcache();
```

2.3 Memory Cache Mode

The cache is a type of fast memory located directly on the core's bus. The processor saves time when accessing instructions or data from the cache rather than from the main memory. An access to information already in a cache is known as a “cache hit”, and other accesses are called “cache misses”. Whenever the processor needs to access a cacheable location, the cache is checked. If the access is a “cache hit”, the access occurs in the cache; otherwise, a location is allocated and the cache line is loaded from the memory.

Each region of the normal memory can be marked as being one of the cache modes below:

- Write-back cacheable mode
  - When a “cache hit” occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than that in the main memory. Any such data is written back to the main memory when the cache line is cleaned or re-allocated.
- Non-cacheable mode

When the CPU writes data to the main memory, the cache controller checks the address to determine if it resides in a data cache-enabled memory region (section). If it is cacheable, the data cache is updated (or added) with new data. If the cache is configured as write-back, the data may not be written to the main memory.

A similar case occurs when the CPU reads data from the main memory. If the address to be read resides in a cache, the data can be loaded directly from the data cache; otherwise, it is loaded from the main memory to the data cache.

2.4 About Cache Coherence

During normal operations, the caches are transparent to the application programmer. However, they can become visible when there is a break in the coherence of the caches. Coherence needs to be addressed if either of the following is true:

- CPU, peripherals, DMA controllers or other external memory share a region of memory for the purpose of data exchange, and this memory region is cacheable by at least one device.
- A memory location in this region has been cached, and this memory location is modified (by any device).

Generally, if multiple devices, such as the CPU or peripherals, share the same cacheable memory region, cache and memory can become incoherent. For example, in a system with a DMA controller that reads memory locations held in the data cache of a processor, a coherence break occurs when the processor has written new data into the data cache, but the DMA controller reads the old data held in memory.

To address this problem, users can ensure the data coherence of caches in several ways. The solutions are discussed in the next section.

3. Direct Memory Access (DMA)

3.1 Overview

DMA enables a peripheral to read/write data directly from/to the main memory. DMA provides special channels to enable the CPU and I/O devices to exchange I/O data, and the memory is used for buffering the I/O data. The CPU can setup the DMA channels and enable DMA to start a data transfer. The CPU can perform other tasks until the DMA transfer is done, and it can access the memory directly to process the data.

3.2 DMA Controller (DMAC)

DMA operates independently of the core processor (CPU). The DMA controller is an AHB-central DMA controller core that transfers data from a source peripheral to a destination peripheral over one or more AMBA buses. DMAC registers are configured to transfer data to or from memory.

The data transfer completion is indicated via a DMA interrupt. For example, a channel DMA can be configured to transmit (memory to peripheral) or receive (peripheral to memory), with generation of a Tx or Rx complete interrupt upon completion. In the case of a transmission, the peripheral DMAC can place data from the memory into the peripheral's data register; in the case of a reception, data from the peripheral's data register is read and placed in the memory.

3.3 SAMA5D3 DMA

- Two 64-bit, 8-channel DMA controllers (AHB-central DMA controller core) that transfer data from a source peripheral to a destination peripheral over one or more AMBA buses
  - HSMCI, SPI, UART, USART, TWI, SSC, DBGU and ADC
  - The DMAC is programmed via the APB interface.
  - The DMAC embeds 8 channels.
  - The DMA Module Supports the Following Transfer Schemes: Peripheral-to-Memory, Memory-to-Peripheral, Peripheral-to-Peripheral and Memory-to-Memory.
- GMAC DMA Master interface
- EMAC DMA Master interface
- UHP EHCI DMA Master interface
- UHP OHCI DMA Master interface
- ISI DMA Master interface
- LCDC DMA Master interface

3.4 Why Use DMA?

The use of a DMA mechanism can greatly increase the throughput to and from a device, because a great deal of computational overhead is eliminated.

- The DMA runs independently of the core.
  - The core sets up the DMA and respond to interrupts.
  - The CPU is typically fully occupied for the entire duration of the read/write operation on I/O and is thus unavailable to perform other work without DMA.
- The core processor cycles are available for processing data.
  - The CPU initiates the DMA transfer, performs other operations while the transfer is in progress, and receives an interrupt from the DMA controller when the operation is done.
- DMA can also be used for "memory to memory" copying, or to move data within the memory.

3.5 Cache Coherence Problems due to DMA

The SAMA5D3 device is equipped with a 32-Kbyte D-Cache, and also includes interfaces to enable connections to a wide range of external memories or to parallel peripherals. External memories (e.g., DRAM) can be accessed directly by the device using DMA. A cache can become incoherent if DMA transfers to/from memory some data that have been cached.

When a DMA transfer changes the contents of the main memory cached by the processor, the data stored in the cache contain the previous values.

However, when the cache is flushed, the stale data are written back to the main memory, overwriting the new data stored by the DMA.

The data in the main memory is then incorrect.

3.5.1 DMA Writing

To simplify this issue, consider the system shown in Figure 3-1. Suppose that the CPU accesses a memory location that gets subsequently allocated in data cache.

Later, in Figure 3-2, a peripheral writes data to this same location that is meant to be read and processed by the CPU, and the DMA engine receives the data from the peripheral and writes the data to the DMA buffer in the main memory continually, until all I/O data are stored in the buffer of the main memory.

Finally, the DMA engine sends an interrupt to the processor to indicate the completion of the receiving operation.

However, since this memory location is kept in cache, when the processor accesses the memory location to process the data, the memory access hits in cache and the CPU gets the old data (0x11223344…) instead of the new data (0x778899AA). If the cached copy of the address is not invalidated, the CPU operates on a stale value.

Figure 3-1. Data Cached

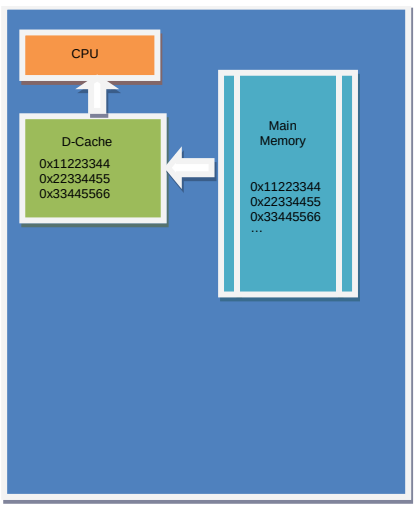
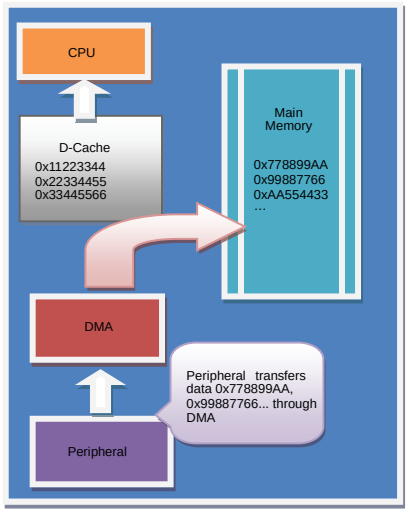


Figure 3-2. DMA Write



3.5.2 DMA Reading

A similar problem occurs if the CPU writes to a DMA-accessible memory location that is cached with a write-back. A write-back cache often contains more recent data than the system memory.

For example, a frame of data from a video/audio stream is frequently updated, and the data is to be read by a peripheral through DMA. The data only gets updated in the cache but not in the main memory, from where the peripheral reads the data. If the cached copy of the address is not flushed, the device receives a stale value.

Figure 3-3. Data Cached

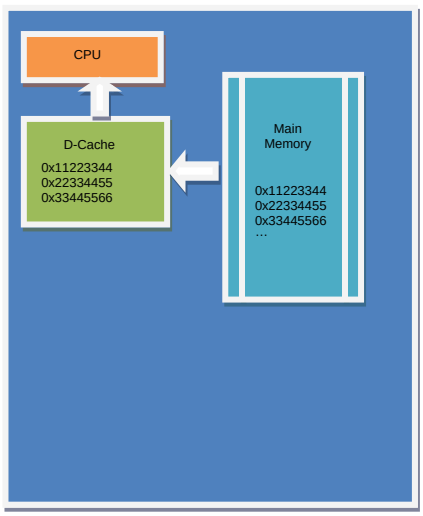
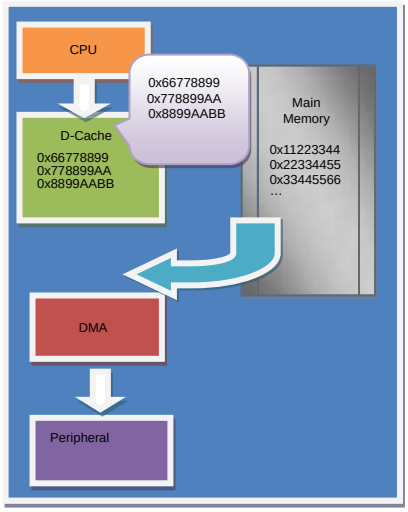


Figure 3-4. DMA Read



3.6 Solutions for Cache Coherence Issue

Different systems have different mechanisms to ensure cache coherence. If this issue is not handled correctly, the driver may corrupt the memory.

Users can ensure the data coherence of caches in the following ways:

- Using the caches in situations where coherence issues cannot arise, which can be achieved by:
  - Using non-cacheable or, in some cases, write-through cacheable memory for the caches
  - Not enabling caches in the system
- Using cache maintenance operations to manage coherence issues in software

4. Managing Data Cache Coherence by Using DMA

4.1 Using Non-Cacheable Memory Regions

Although the I-cache/D-cache can be disabled entirely, this is not recommended, as access to a cache is much faster than access to an external memory.

Another simple way is to define some specific memory regions as cache-disabled, and to place the DMA descriptor buffer and data buffers in non-cacheable memory. It is then unnecessary to maintain coherence between the CPU and the DMA.

4.1.1 Defining Non-Cacheable Memory Regions in TLB

Non-cacheable memory regions do not need any cache-coherence strategies, and they typically represent I/O device spaces. This results in a significant reduction in the complexity of the hardware and software aspects of a coherence issue. For more information, see 1.3.1.3 “TLB Example for DDRAM with Non-Cacheable Attributes” .

4.1.2 Section Handling for Non-Cacheable Region in Linker Script

The “section directive” in an EWARM IAR or GCC system makes it possible to create empty sections with specific sizes for some special attributes. The examples below show how to define a section/region in an EWARM IAR/GCC linker script. This special region is already pre-defined as “non-cacheable” in the MMU.

The section named “region_dma_nocache” resides in a non-cacheable region of the memory, so it is not required to maintain coherence between the CPU and the DMA.

4.1.2.1 Defining a Non-Cacheable Region in an EWARM IAR Linker Script

```
define symbol __ICFEDIT_region_DDRAM_BUF_start__ = 0x23000000;
define symbol __ICFEDIT_region_DDRAM_BUF_end__   = 0x23FFFFFF;
…
define region DMA_BUF_region = mem:[from __ICFEDIT_region_DDRAM_BUF_start__ to
__ICFEDIT_region_DDRAM_BUF_end__];
…
place in DMA_BUF_region {section region_dma_nocache };
place in DDRAM_region { block IRQ_STACK, block SYS_STACK, block CSTACK, block
HEAP };
```

For more information, see Section “The linker configuration file” in *IAR C/C++ Development Guide*.

4.1.2.2 Defining a Non-Cacheable Region in a GNU Linker Script

```
MEMORY
{
   …
dma_ddr (RWX) : ORIGIN = 0x23000000, LENGTH = 1M
}
SECTIONS
{
…
.region_dma_nocache (NOLOAD):
    {
        . = ALIGN(4);
        *(.region_dma_nocache)
    } >dma_ddr
}
```

For more information, see Section “Linker Scripts” in *The GNU linker ld (Sourcery G++ Lite 2011.03-42).*

4.1.3 DMA Non-Cacheable Memory Example

In applications, users only need to specify the previously defined section (region_dma_nocache) for DMA buffers. Taking GMAC as an example: the GMAC includes an AHB DMA interface. When the GMAC is configured to use DMA, the code can place the TX/RX buffer and the TX/RX descriptor in section “region_dma_nocache” by using “#pragma location = xxx”. Coherence is maintained because the area that is accessible by multiple masters (core and DMA engines) is not cached and the memory area never resides in a cache.

4.1.3.1 Example Code to Place Buffers in a Non-Cacheable Region in EWARM IAR

```
#if defined ( __ICCARM__ ) /* IAR Ewarm */
#pragma data_alignment=8
#pragma location = "region_dma_nocache"
#endif
static sGmacTxDescriptor gTxDs[TX_BUFFERS];

#if defined ( __ICCARM__ ) /* IAR Ewarm */
#pragma data_alignment=8
#pragma location = "region_dma_nocache"
#endif
static uint8_t pTxBuffer[TX_BUFFERS * GMAC_TX_UNITSIZE];
```

4.1.3.2 Example Code to Place Buffers in a Non-Cacheable Region in GNU

```
#if defined ( __GNUC__ ) /* GCC CS3 */
__attribute__((aligned(8), __section__(".region_dma_nocache")))
#endif
static sGmacTxDescriptor gTxDs[TX_BUFFERS];

#if defined ( __GNUC__ ) /* GCC CS3 */
__attribute__((aligned(8), __section__(".region_dma_nocache")))
#endif
static uint8_t pTxBuffer[TX_BUFFERS * GMAC_TX_UNITSIZE];
```

4.2 Managing DMA Coherence Issues in Software

4.2.1 Software Cache Coherence Solution Strategies

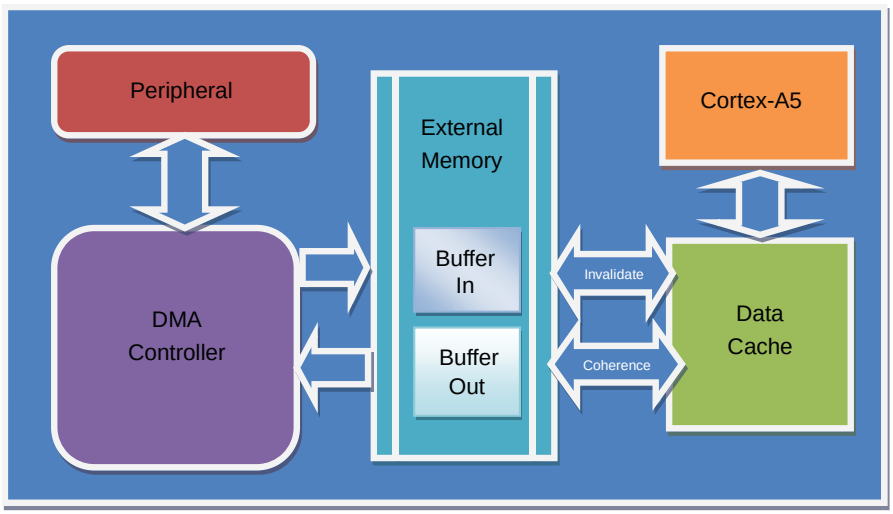
Dynamic software cache coherence strategies use the information of program sharing behavior to manage caches at run-time defined by the application.

DMA operates without using the CPU, so an additional API must be present to bring the CPU caches into sync with the memory changed by DMA. That means that DMA must be limited to non-cacheable buffers, or that care must be taken to invalidate or clean caches before the start of DMA transfers to cached buffers.

The coherence between a data cache and a buffer filled via DMA transfer must be maintained in the following cases:

- The cache must be “invalidated” to ensure that the “old” data in cache is up to date when the CPU processes the most recent buffers after a DMA read.
  - DMA transfers data from the peripheral into the external memory via an input buffer.
  - The input buffer must be invalidated before CPU processing.
- The cache must be “coherent” before the data transfer through DMA.
  - The output buffer to be transferred via DMA should be taken into account.
  - The DMA descriptor should be taken into account.
- Other potential coherence problems can occur when some data structures share the same cache line.
  - Avoid using only one buffer for both transmit and receive.
  - Make sure that the general-usage memory and the peripheral-usage memory do not share the same buffer.

Figure 4-1. Software Cache Coherence Solution Strategies



4.2.2 Functions of Cache Coherence Operations

The chip library in the software package provides a set of routines that allows the required cache coherence operations to be initiated.

- CP15_coherent_dcache_for_dma (uint32_t bufStartAddress, uint32_t bufSize).

This function ensures that the I-Cache and D-Cache are coherent within the specified address. The start address of the buffer in the external memory and the number of bytes need to be specified. This function is typically used when some code has been written to a cacheable memory region and is to be executed by DMA.

- CP15_invalidate_dcache_for_dma (uint32_t bufStartAddress, uint32_t bufSize).

This function invalidates the D-Cache within the specified region. The start address of the buffer in external memory and the number of bytes need to be specified. This function is typically used when old data must be purged from the cache after performing a DMA operation in this region.

4.2.3 Receiving Data from a Peripheral Using DMA

DMA provides special channels to enable the CPU and peripherals to exchange I/O data, and the memory is used for buffering the I/O data. A programmable DMA channel may be assigned by the CPU firmware to serve each peripheral for fast and direct transfers between the peripheral and the external memory without the involvement of the CPU. DMA transfer is supported for transmit and receive.

4.2.3.1 Typical DMA Operations to Receive Data from a Peripheral

- The device driver creates a descriptor for a DMA buffer.
- The driver allocates a DMA buffer in the memory and initializes the descriptor with the start address, size and transfer characteristics of the DMA buffer. The driver provides the descriptor's source address, which comes from the peripheral, configures the destination start address at the external memory, and then enables the DMA channel.
- Once the data have been transferred from the peripheral, the DMA controller reprograms the channel registers before each buffer transfer by fetching the buffer descriptor for that buffer from the system memory.
- The DMA controller transfers continuously until the end of a buffer transfer is matched.

Once the DMA transfers are completed, the data located in the external memory (following DMA transfer from the peripheral) can be read and processed by the CPU. Because the data is transferred to the main memory directly by DMA, the caches between that memory and the CPU are probably not coherent.

The CPU may get data from the cache, which may be different from the data in memory that has been updated by DMA. So, the cache and the memory can be incoherent.

The "cache invalidate" procedure must be called after the device has transferred the data, but before the CPU attempts to read it.

4.2.3.2 Example Code for **Cache Invalidate**

```
/**
*   DMA interrupt handler.
*  This function handles DMA interrupts.
*/
static void DmaHandler(void)
{
     /* Invalidate the data cache within the specified region to purge the old
data in the cache. */
     CP15_invalidate_dcache_for_dma ((uint32_t)pDataBuf, (uint32_t)dataSize);
   …
   /* Data process */
…
}
```

4.2.4 Transmitting Data to a Peripheral Using DMA

4.2.4.1 Typical DMA Operations to Transmit Data to a Peripheral

- The device driver creates a descriptor for a DMA buffer.
- The driver allocates a DMA buffer in the memory and initializes the descriptor with the start address, size and transfer characteristics of that buffer. The driver provides the descriptor's source address, which comes from the external memory, configures the destination start address of the peripheral I/O, and then enables the DMA channel.

- Once the data have been transferred to the peripheral, the DMA controller reprograms the channel registers before each buffer transfer by fetching the buffer descriptor for that buffer from the system memory.
- The DMA controller transfers continuously until the end of a buffer transfer is matched.

First, when a buffer is mapped for DMA, the driver must ensure that all the data in that buffer have actually been written to the external memory. It is likely that some data are in the processor's cache when DAMC_Enable() is issued. Such data must be explicitly flushed.

Data written to the buffer by the processor after the flush may not be visible to the device. The CPU executes the cache flush instructions immediately before the output operations.

These instructions are broadcast to the processor(s) and require it (them) to flush their caches by writing the specified dirty cache lines back to the main memory.

After the DMA buffer has been flushed, the DMA operation can proceed. It is then guaranteed to transfer the most up-to-date data. Before the DMA operation is completed, avoid updating the buffer used for DMA after the flush, otherwise the buffer could go into some undefined status.

4.2.4.2 Example Code for Cache Flush before DMA Transfer

```
/**
 *  \brief Start DMA transmit.
 */
static void _DmaTx( void )
{
sDmaTransferDescriptor td;
/* Cache coherent function to flush the buffer in the main memory */
    CP15_coherent_dcache_for_dma ((uint32_t)&pBuffer, ((uint32_t)(&pBuffer) +
BUFFER_SIZE));
    td.dwSrcAddr = (uint32_t) pBuffer;
    td.dwDstAddr = (uint32_t)&BASE_USART->US_THR;
    td.dwCtrlA   = BUFFER_SIZE | DMAC_CTRLA_SRC_WIDTH_BYTE |
DMAC_CTRLA_DST_WIDTH_BYTE;
    td.dwCtrlB   = DMAC_CTRLB_SRC_DSCR | DMAC_CTRLB_DST_DSCR
                 | DMAC_CTRLB_SIF_AHB_IF0 | DMAC_CTRLB_DIF_AHB_IF2
                 | DMAC_CTRLB_FC_MEM2PER_DMA_FC
                 | DMAC_CTRLB_SRC_INCR_INCREMENTING
                 | DMAC_CTRLB_DST_INCR_FIXED;
    td.dwDscAddr = 0;
    DMAD_PrepareSingleTransfer(&dmad, usartDmaTxChannel, &td);
    DMAD_StartTransfer(&dmad, usartDmaTxChannel);
}
```

5. Performance Test Results in SMC_NAND Example

To collect the performance test results of NAND Flash, the benchmark was run on SAMA5D3 with the following configurations:

- 2 x 512-Mbyte DDRAMs
- MMU with 32-Kbyte D-Cache, 32-Kbyte I-Cache
- SLC NAND Flash (page size: 2048 bytes)
- MCK runs at 133 MHz

5.1 NAND Flash Test Results

To compare the performance under different configurations (with or without MMU, DMA, etc.), NAND Flash read/write measurements were taken during several runs under the configurations described below. The example for this measurement is mainly taken from the smc_nandflash example in the SAMA5D3 software package.

Table 5-1. NAND Flash Read/Write Performance Test Results

Test configurations						Test results	
MMU	**DMA**	**I-Cache**	**D-Cache**	**Using non-cacheable region**	**Software-enforced coherence**	**Write (Kbytes/s)**	**Read (Kbytes/s)**
OFF	OFF	OFF	OFF	-	-	1659	1680
ON	OFF	OFF	OFF	-	-	1771	2788
ON	OFF	ON	OFF	-	-	2788	3360
ON	OFF	ON	ON	-	-	4360	4681
OFF	ON	OFF	OFF	-	-	3120	3855
ON	ON	OFF	OFF	-	-	3449	4681
ON	ON	ON	OFF	-	-	5041	8192
ON	ON	ON	ON	ON	-	3640	5041
ON	ON	ON	ON	-	ON	6241	11915

5.2 NAND Flash Example Code for DMA Software-Enforced Coherence

5.2.1 Configuring the DMA Channels for Tx

```
/**
 * \brief Configure the DMA Channels for Tx.
 * \param ramAddr Source address to be transferred.
 * \param destAddr Destination address to be transferred.
 * \param size Transfer size in byte.
 * \returns 0 if the DMA channel configures and transfers successfully,
otherwise returns NandCommon_ERROR_XXX.
 */
uint8_t NandFlashDmaTransferToNand(
uint32_t srcAddr,
uint32_t destAddr,
uint32_t size )
{
    if (!pCurrentDma) return NandCommon_ERROR_DMA;
    CP15_coherent_dcache_for_dma(srcAddr, srcAddr + size );
    dmaNandTxDesc.dwSrcAddr = srcAddr;
    dmaNandTxDesc.dwDstAddr = destAddr;
    dmaNandTxDesc.dwCtrlA   = ((size + 3) >> 2) | DMAC_CTRLA_SRC_WIDTH_WORD |
DMAC_CTRLA_DST_WIDTH_WORD;
    if (DMAD_PrepareSingleTransfer( pCurrentDma, nandDmaTxChannel,
&dmaNandTxDesc )) return NandCommon_ERROR_DMA;
    if (DMAD_StartTransfer( pCurrentDma, nandDmaTxChannel ))
        return NandCommon_ERROR_DMA;
    while (DMAD_IsTransferDone( pCurrentDma, nandDmaTxChannel ));
    return 0;
}
```

5.2.2 Configuring the DMA Channels for Rx

```
/**
 * \brief Configure the DMA Channels for Rx.
 * \param scrAddr Source address to be transferred.
 * \param destAddr Destination address to be transferred.
 * \param size Transfer size in byte.
 * \returns 0 if the dma channel configures and transfers successfully;
otherwise returns NandCommon_ERROR_XXX.
 */
uint8_t NandFlashDmaTransferFromNand(
uint32_t srcAddr,
uint32_t destAddr,
uint32_t size )
{
    if (!pCurrentDma) return NandCommon_ERROR_DMA;
    CP15_coherent_dcache_for_dma(destAddr, destAddr + size );
    dmaNandRxDesc.dwSrcAddr = srcAddr;
    dmaNandRxDesc.dwDstAddr = destAddr;
    dmaNandRxDesc.dwCtrlA   = ((size + 3) >> 2) | DMAC_CTRLA_SRC_WIDTH_WORD |
DMAC_CTRLA_DST_WIDTH_WORD;
    if (DMAD_PrepareSingleTransfer( pCurrentDma, nandDmaRxChannel,
&dmaNandRxDesc )) return NandCommon_ERROR_DMA;
    if (DMAD_StartTransfer( pCurrentDma, nandDmaRxChannel ))
        return NandCommon_ERROR_DMA;
    while (DMAD_IsTransferDone( pCurrentDma, nandDmaRxChannel ));
    CP15_flush_dcache_for_dma(destAddr, destAddr + size );
    return 0;
}
```

6. Conclusion

This Application Note provides two ways to manage cache coherence:

- using non-cacheable regions
- applying cache-clean-invalidate and cache-flush procedures.

The choice of the method is application-dependent, and should be based on the amount of data to be processed.

7. Revision History

In the table that follows, the most recent version of the document appears first.

Table 7-1. SAMA5D3 MMU Cache Coherence 11239A Revision History

Doc. Rev.	Comments
A	First issue

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**