SMART ARM-based Microcontrollers

AT17417: Usage of XDMAC on SAM S/SAM E/SAM V

APPLICATION NOTE

Introduction

This application note describes the features of XDMAC peripheral, which is present in the Atmel® SAM S, SAM E, and SAM V microcontroller family. This application note also provides information about various memory transfer, memory striding functionalities and linked list descriptor operations with a help of a sample application code implemented on ATSAMV71Q21 device (SAM V71 Xplained Ultra Evaluation Kit).

Table of Contents

Introduction....1

1. Glossary....3

2. XDMAC Basics....4
   - 2.1. Memory to Memory Transfer....4
   - 2.2. Peripheral to Memory Transfer....5
   - 2.3. Memory to Peripheral Transfer....6

3. Single Block Memory Transfer....7

4. Multi Block Memory Transfer....8
   - 4.1. Linked List Formation....8
   - 4.2. Linked List Descriptors....9

5. Memory Striding....14
   - 5.1. Data Striding....14
   - 5.2. Microblock Striding....18
   - 5.3. Block Striding....22

6. Application Code - Getting Started....23

7. Application Code - Demonstration....24
   - 7.1. Task 1....24
   - 7.2. Task 2....25
   - 7.3. Task 3....26
   - 7.4. Task 4....28

8. Cache Coherence Management....29

9. Conclusion....30

10. Frequently Asked Questions (FAQs)....31

11. References....32

12. Revision History....33

1. Glossary

XDMAC	Extensible Direct Memory Access Controller
DMA	Direct Memory Access
UART	Universal Asynchronous Receiver Transmitter
SPI	Serial Peripheral Interface
TWI	Two Wire Interface
AES	Advanced Encryption Standard
HSMCI	High Speed Multimedia Card Interface
FIFO	First In First Out
ASF	Atmel Software Framework
Atmel Studio	Integrated Development Environment (IDE) for Atmel Microcontrollers

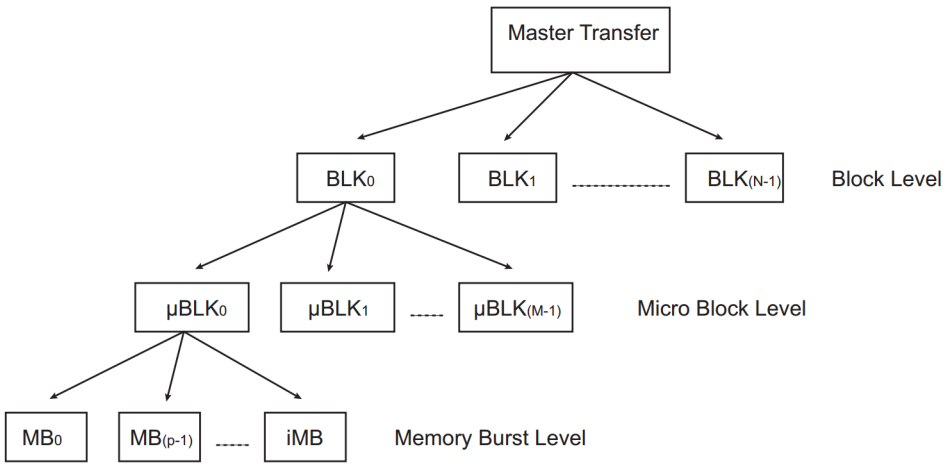
2. XDMAC Basics

XDMAC supports the following types of data transfers.

2.1. Memory to Memory Transfer

XDMAC reads data from source memory location and writes to destination memory location.

Figure 2-1. Memory Transfer Hierarchy



Memory to memory data transfer has totally four levels of data transactions. They are Master, Block, Microblock, and Burst level transactions.

XDMAC Master Transfer: The Master Transfer is a multi-block data transfer, which is performed using a linked list of descriptors (blocks). Each descriptor in the linked list is configured to do a block transfer. The XDMAC channel configuration parameters can be modified at the inter block boundary (between descriptors). In multi-block transfer, interrupts can be generated on a per block basis or when the end of linked list event occurs. Chapter-4. Multi Block Memory Transfer gives more information about multi-block transfer.

Note: Master transfer (Multi-block transfer) is optional. It is not mandatory. A single block transfer can be done.

XDMAC Block: An XDMAC block is composed of programmable number of microblocks. The block length (number of microblocks) is configured in BLEN field of XDMAC Channel Block Control Register (XDMAC_CBCx). The block length (BLEN) indicates the number of microblocks in a block. The XDMAC channel configuration parameters remain unchanged at the inter microblock boundary.

Note: Block transfer is mandatory. At least, one block should be transmitted with one microblock.

XDMAC Microblock: A microblock is composed of programmable number of data. The microblock length is configured in UBLEN field of XDMAC Channel Microblock Control Register (XDMAC_CUBCx). The microblock length (UBLEN) indicates the number of data (bytes or half words or words based on the data width setting) present in a microblock. The XDMAC channel configuration parameters remain unchanged at the data boundary as well.

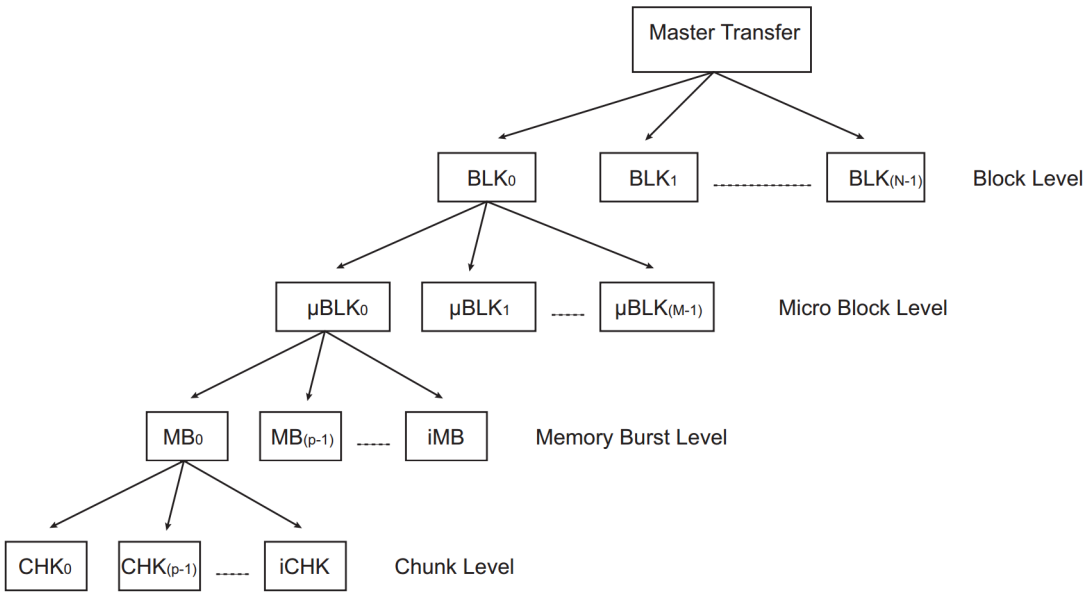
XDMAC Burst and Incomplete Burst: In order to improve the overall performance when accessing dynamic external memory, burst access is mandatory. Each data of the microblock is considered as a part of a memory burst. The programmable burst value indicates the largest memory burst allowed on a per channel basis. The burst size (in WORDS) is configured in MBSIZE field of XDMAC Channel

Configuration Register (XDMAC_CCx). When the microblock length is not an integral multiple of the burst size, an incomplete burst is performed to read or write the last trailing bytes.

2.2. Peripheral to Memory Transfer

XDMAC reads data from the source peripheral and writes to the destination memory location.

Figure 2-2. Peripheral to Memory Transfer Hierarchy



It is a peripheral synchronized transfer, which means the memory transaction is synchronized with the hardware trigger that comes from the corresponding peripheral. It is also possible to use software trigger to initiate data transfer. Peripheral to memory transfer has totally five levels of data transactions. They are Master, Block, Microblock, Burst, and Chunk level transactions. Master, Block, Microblock, and Burst level transactions work exactly the same way as explained earlier in the memory to memory data transfer section. In peripheral to memory data transfer, the burst level transaction is further split into chunk level data transaction to have higher granularity.

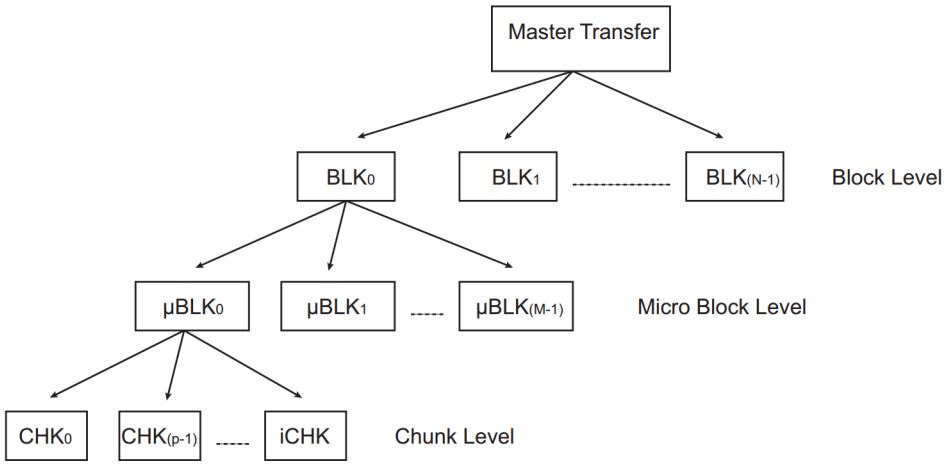
XDMAC Chunk and Incomplete Chunk: When a peripheral to memory transfer is activated, the burst level transaction is further split into a number of data chunks. The chunk size is configured in CSIZE field of XDMAC Channel Configuration Register (XDMAC_CCx). The chunk size denotes the number of ‘data’ to be transferred from the corresponding peripheral receive register to memory. In general, the chunk size is set as ‘1 data’ in most of the peripherals (example: - UART, SPI, TWI, etc.), as the maximum size of their receive register is ‘1 data’. In specific scenarios, the chunk size is chosen more than 1 data. For example, the data receive/input registers of AES and HSMCI modules can hold more than ‘1 data’. So, the chunk size can be chosen as '2/4/8/16 data' accordingly. In this case, the larger the chunk size is, the better the performance is. When the amount of data chunks read becomes equal to the memory burst size, the actual data transaction starts (as a memory burst). During ‘peripheral to memory’ transfer, the data chunks are first read and stored into XDMAC's internal FIFO buffer. If their size becomes equal to the memory burst size, the FIFO buffer gets flushed out automatically, which makes ‘memory burst transfer’. When the microblock size is not a multiple of the chunk size, the last chunk being transferred contains the last trailing data.

Note: In case if the chunk size is chosen as more than '1 data' for peripherals like UART, SPI, TWI, etc., then XDMAC will read the same data register (receive/input register) multiple times. As a result, we will get multiple copies of same data being stored in memory.

2.3. Memory to Peripheral Transfer

XDMAC reads data from source memory location and writes to the destination peripheral.

Figure 2-3. Memory to Peripheral Transfer Hierarchy



Memory to Peripheral transfer is also a peripheral synchronized transfer. It has totally four levels of data transactions. They are Master, Block, Microblock, and Chunk level transactions. Master, Block, and Microblock level transactions work exactly the same way as explained earlier in the memory to memory data transfer section. In memory to peripheral data transfer, the burst level transaction is not present. The microblock is directly split into chunk level data transaction.

XDMAC Chunk and Incomplete Chunk: When a memory to peripheral transfer is activated, the microblock level transaction is directly split into a number of data chunks. The chunk size is configured in CSIZE field of XDMAC Channel Configuration Register (XDMAC_CCx). The chunk size denotes the number of ‘data’ to be transferred from memory to the corresponding peripheral transmit register. In general, the chunk size is set as ‘1 data’ in most of the peripherals (example: - UART, SPI, TWI, etc.), as the maximum size of their transmit register is ‘1 data’. In specific scenarios, the chunk size is chosen more than 1 data. For example, the data transmit/output registers of AES and HSMCI modules can hold more than ‘1 data’. So, the chunk size can be chosen as '2/4/8/16 data' accordingly. In this case, the larger the chunk size is, the better the performance is. During ‘memory to peripheral’ transfer, the data chunks are immediately transferred when there is a hardware/software trigger. Memory burst size doesn't play any role here. When the microblock size is not a multiple of the chunk size, the last chunk being transferred contains the last trailing data.

Note: In case if the chunk size is chosen as more than '1 data' for peripherals like UART, SPI, TWI, etc., then XDMAC will overwrite the same data register (transmit/output register) with multiple data. As a result, only the last data gets transmitted.

3. Single Block Memory Transfer

A basic single block of DMA transfer can be done by just configuring XDMAC channel registers directly. After configuring XDMAC channel registers, the corresponding channel needs to be enabled. This will trigger the data transaction, if it is a memory to memory transfer. The peripheral synchronized transfer will still wait for the hardware or software trigger to occur. Refer to the device datasheet for more information on initialization sequence and software flow.

The channel configuration parameters (example: source/destination addresses, block/microblock length, stride length, etc.) remain same throughout the block transfer. They cannot be modified between microblocks.

4. Multi Block Memory Transfer

Multi block memory transfer is needed when there is a change in XDMAC channel configuration parameters (example: source/destination addresses, block/microblock length, stride length, etc.) between blocks transferred in the same channel. XDMAC supports multi block DMA transfer using Linked list operation. Each descriptor in the linked list contains register settings needed to transfer a 'block' of memory. When linked list execution is started, XDMAC fetches the first descriptor from the linked list, and starts copying the register settings to XDMAC channel register conditionally. Then it performs the first memory block transfer. After finishing the first block transfer, XDMAC fetches the second descriptor from the linked list and copy its register settings to channel registers to perform second block transfer. This process continues until the end of linked list is reached. This is how a multi block memory transfer is performed in XDMAC.

4.1. Linked List Formation

XDMAC has four types of linked list descriptors namely View 0, View 1, View 2, and View 3. A linked list can be formed either with descriptors of same type or with descriptors of different types. Figure 4-1 shows a simple linked list formed with three View 0 descriptors. Figure 4-2 shows a complex linked list having all four types (View 0, View 1, View 2, and View 3) of descriptors.

Figure 4-1. Simple Linked List Example

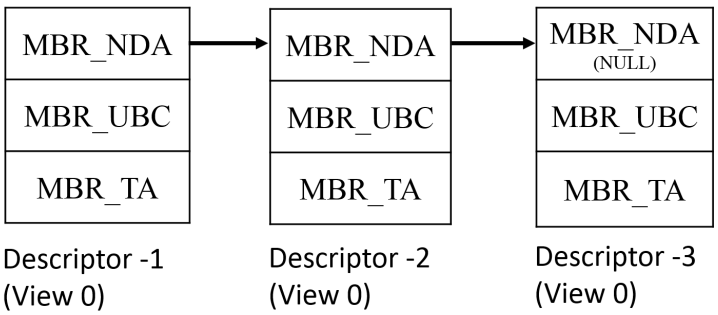
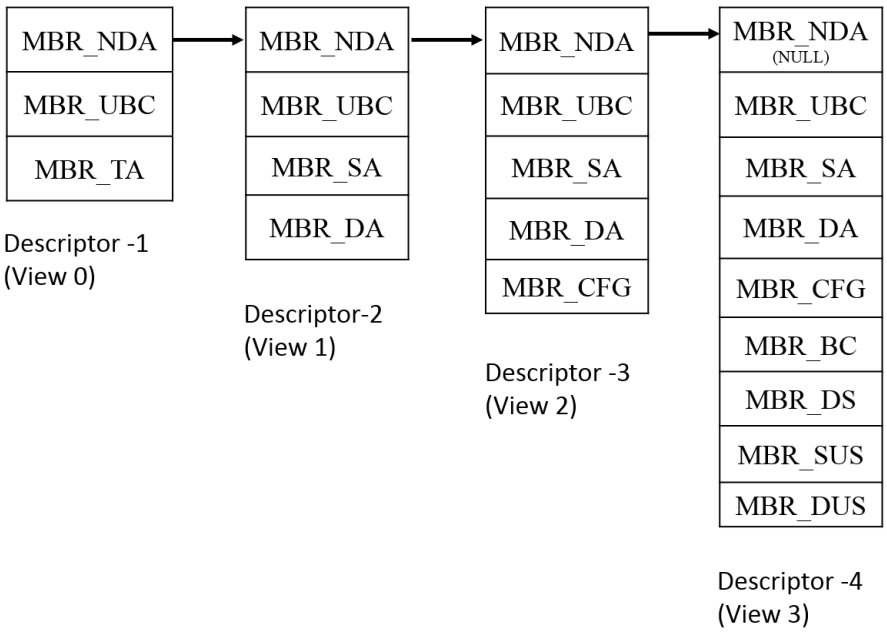


Figure 4-2. Complex Linked List Example



View 0, View 1, and View 2 descriptors are generally used to perform peripheral synchronized transfer along with ring buffers. For example, we can consider the case of UART reception. Let us assume we continuously receive data from UART and we need to process the received data once we receive 50 data in the buffer. At the same we need to continue receive upcoming data without a miss. In this case we can configure a circular linked list having two descriptors of View 0 type. The first descriptor should be configured to receive 50 data (1 block = 1 microblock = 50 data) into the first buffer. The first descriptor should point to the second descriptor. The second descriptor should be configured to receive the next 50 data into the second buffer. The second descriptor should point to the first descriptor again (circular linked list). 'End of block interrupt' can be enabled, so that we can get an interrupt for each block (50 data) being received. When an interrupt occurs we can go and process the data.

View 3 descriptor is especially used when we have multiple microblocks per block. It is also useful when we vary memory stride lengths and MEMSET patterns between different block transfers.

4.2. Linked List Descriptors

Linked list descriptors are usually stored in data memory. They have multiple members associated, which are explained below in following sections. When a descriptor is fetched from the linked list, all of its members are copied to their relevant XDMAC channel register fields for execution. But there are few exceptions. MBR_TA, MBR_SA and MBR_DA are copied based on conditions. It is explained in the following sections.

The following descriptor member fields are meant to control the next descriptor, not the current descriptor itself.

- Next Descriptor Address Member (MBR_NDA)
- Next Descriptor Enable (MBR_UBC.NDE)
- Next Descriptor Source Update (MBR_UBC. NSEN)
- Next Descriptor Destination Update (MBR_UBC. NDEN)
- Next Descriptor View (MBR_UBC.NVIEW)

So, a question arises here! Which fields will control the first descriptor of the linked list? The following channel register fields should be directly initialized for the first descriptor. Therefore, the first descriptor is fetched based on the following channel registers.

- XDMAC Channel x Next Descriptor Address Register (XDMAC_CNDAx)
- XDMAC Channel x Next Descriptor Control Register (XDMAC_CNDCx)
  - Channel x Next Descriptor Enable (XDMAC_CNDCx.NDE)
  - Channel x Next Descriptor Source Update (XDMAC_CNDCx.NDSUP)
  - Channel x Next Descriptor Destination Update (XDMAC_CNDCx.NDDUP)
  - Channel x Next Descriptor View (XDMAC_CNDCx.NDVIEW)

After fetching the first descriptor, the above mentioned channel register fields will be again updated with 'first descriptor member fields' as shown below. This will help to fetch and execute the second (next) descriptor. The same process continues until the end of the linked list.

- MBR_NDA → XDMAC_CNDAx
- MBR_UBC.NDE → XDMAC_CNDCx.NDE
- MBR_UBC. NSEN → XDMAC_CNDCx.NDSUP
- MBR_UBC. NDEN → XDMAC_CNDCx.NDDUP
- MBR_UBC.NVIEW → XDMAC_CNDCx.NDVIEW

4.2.1. View 0 Descriptor

Next Descriptor Address Member (MBR_NDA)
Microblock Control Member (MBR_UBC)
Transfer Address Member (MBR_TA)

View 0 is the simplest descriptor having just three members. They are explained below.

Next Descriptor Address Member (MBR_NDA):

MBR_NDA is similar to XDMAC Channel Next Descriptor register (**XDMAC_CNDAx**). XDMAC_CNDAx register is initialized to the address of the first descriptor of the linked list, whereas Next Descriptor Address Member (MBR_NDA) is initialized to the address of the subsequent descriptor to be fetched from the linked list. If there are no further descriptors present in the linked list, then MBR_NDA should be initialized with 0. When a descriptor is fetched, XDMAC_CNDAx register is updated with MBR_NDA value for the execution of the next descriptor (Block).

Microblock Control Member (MBR_UBC):

Microblock Control Member has the following fields.

UBLEN (Microblock Length):

This field indicates the number of data (bytes or half words or words based on XDMAC_CCx.DWIDTH setting) in the microblock. So each microblock contains UBLEN data. The UBLEN field can be varied for each descriptor. When a descriptor is fetched, XDMAC_CUBCx.UBLEN register field is updated with MBR_UBC.UBLEN value for the execution of the current descriptor (Block).

NDE (Next Descriptor Enable):

0: No further descriptors will be fetched. So NDE should be set as '0' for the last descriptor of the linked list.

1: The next descriptor pointed by MBR_NDA will be fetched next.

NSEN (Next Descriptor Source Update):

0: The Channel Source Address register XDMAC_CSAx remains unchanged during the next descriptor fetch.

1: When the next descriptor is fetched, the channel source address register XDMAC_CSAx is updated with Transfer Address member (MBR_TA) of the same descriptor, if the data transfer is from Memory to Peripheral. XDMAC_CSAx remains unchanged for Memory to Memory and Peripheral to Memory data transfers.

NDEN (Next Descriptor Destination Update):

0: The Channel Destination Address register XDMAC_CDAx remains unchanged during the next descriptor fetch.

1: When the next descriptor is fetched, the channel destination address register XDMAC_CDAx is updated with Transfer Address member (MBR_TA) of the same descriptor, if the data transfer is from Memory to Memory or Peripheral to Memory. XDMAC_CDAx remains unchanged for Memory to Peripheral data transfer.

NVIEW (Next Descriptor View):

0: The next descriptor is of View 0 type

1: The next descriptor is of View 1 type

2: The next descriptor is of View 2 type

3: The next descriptor is of View 3 type

Transfer Address Member (MBR_TA):

Transfer Address Member (MBR_TA) should be written with the destination address during Memory to Memory and Peripheral to Memory transfers. When the descriptor is fetched, the Channel Destination Address register (XDMAC_CDAx) is updated with the valued stored in Transfer Address member (MBR_TA) based on the previous descriptor's MBR_UBC.NDEN value.

Transfer Address Member (MBR_TA) should be written with the source address during memory to peripheral transfer. When the descriptor is fetched, the Channel Source Address register (XDMAC_CSAx) is updated with the valued stored in Transfer Address member (MBR_TA) based on the previous descriptor's MBR_UBC.NSEN value.

Note: For the first descriptor of the linked list, XDMAC_CNDCx.NDSUP and XDMAC_CNDCx.NDDUP values should be directly initialized. These values decide whether to update Channel Source / Destination Address registers with MBR_TA or not.

4.2.2. View 1 Descriptor

Next Descriptor Address Member (MBR_NDA)
Microblock Control Member (MBR_UBC)
Source Address Member (MBR_SA)
Destination Address Member (MBR_DA)

View 1 descriptor has both Source Address Member (MBR_SA) and Destination Address Member (MBR_DA).

Next Descriptor Address Member(MBR_NDA):

Same as explained in View 0 descriptor.

Microblock Control Member (MBR_UBC):

Microblock Control Member has the following fields.

UBLEN (Microblock Length):

Same as explained in View 0 descriptor.

NDE (Next Descriptor Enable):

Same as explained in View 0 descriptor.

NSEN (Next Descriptor Source Update):

0: The Channel Source Address register XDMAC_CSAx remains unchanged during the next descriptor fetch.

1: When the next descriptor is fetched, the channel source address register XDMAC_CSAx is updated with Source Address Member (MBR_SA) of the same descriptor.

NDEN (Next Descriptor Destination Update):

0: The Channel Destination Address register XDMAC_CDAx remains unchanged during the next descriptor fetch.

1: When the next descriptor is fetched, the channel destination address register XDMAC_CDAx is updated with Destination Address Member (MBR_DA) of the same descriptor.

NVIEW (Next Descriptor View):

Same as explained in View 0 descriptor.

Source Address Member (MBR_SA):

It contains the source address value of the corresponding descriptor (BLOCK). The value of MBR_SA is copied to the Channel Source Address register (XDMAC_CSAx) based on the previous MBR_UBC.NSEN settings. Therefore, the source address can be changed between different descriptors (BLOCKS).

Note: For the first descriptor of the linked list, the XDMAC_CNDCx.NDSUP value should be directly initialized. This value decides whether to update Channel Source Address register (XDMAC_CSAx) with MBR_SA or not.

Destination Address Member (MBR_DA):

It contains the destination address value of the corresponding descriptor (BLOCK). The value of MBR_DA is copied to the Channel Destination Address register (XDMAC_CDAx) based on the previous MBR_UBC. NDEN settings. Therefore, the destination address can be changed between different descriptors (BLOCKS).

Note: For the first descriptor of the linked list, the XDMAC_CNDCx.NDDUP value should be directly initialized. This value decides whether to update Channel Destination Address register (XDMAC_CDAx) with MBR_DA or not.

4.2.3. View 2 Descriptor

Next Descriptor Address Member (MBR_NDA)
Microblock Control Member (MBR_UBC)
Source Address Member (MBR_SA)
Destination Address Member (MBR_DA)
Configuration Member (MBR_CFG)

View 2 descriptor has Configuration Member (MBR_CFG) in addition to View 1 descriptor. All other members (MBR_NDA, MBR_UBC, MBR_SA, and MBR_DA) are similar to View 1 descriptor.

Configuration Member (MBR_CFG):

MBR_CFG is similar to XDMAC_CCx register. During the descriptor fetch, the value of MBR_CFG is copied to the XDMAC_CCx register.

4.2.4. View 3 Descriptor

Next Descriptor Address Member (MBR_NDA)
Microblock Control Member (MBR_UBC)
Source Address Member (MBR_SA)
Destination Address Member (MBR_DA)
Configuration Member (MBR_CFG)
Block Control Member (MBR_BC)

Data Stride Member (MBR_DS)
Source Microblock Stride Member (MBR_SUS)
Destination Microblock Stride Member (MBR_DUS)

View 3 descriptor has totally nine members. First five members (MBR_NDA, MBR_UBC, MBR_SA, MBR_DA, MBR_CFG) are very similar to View 2 descriptor members. View 3 descriptor additionally has the following four members.

Block Control Member (MBR_BC):

MBR_BC is similar to the Channel Block Control Register (XDMAC_CBCx). During the descriptor fetch, the value of MBR_BC is copied to the XDMAC_CBCx register.

Data Stride Member (MBR_DS):

MBR_DS is similar to the Channel Data Stride Memory Set Pattern Register (XDMAC_CDS_MSPx). During the descriptor fetch, the value of MBR_DS is copied to the XDMAC_CDS_MSPx register.

Source Microblock Stride Member (MBR_SUS):

MBR_SUS is similar to the Channel Source Microblock Stride Register (XDMAC_CSUSx). During the descriptor fetch, the value of MBR_SUS is copied to the XDMAC_CSUSx register.

Destination Microblock Stride Member (MBR_DUS):

MBR_DUS is similar to the Channel Destination Microblock Stride Register (XDMAC_CDUSx). During the descriptor fetch, the value of MBR_DUS is copied to the XDMAC_CDUSx register.

5. Memory Striding

Memory Striding is a method of accessing memory in an interleaved (discontinuous) manner. Memory striding can be done in both incremented (forward direction) and decremented (reverse direction) manners. XDMAC supports memory striding in all three types of data transfers (Memory to Memory / Peripheral to Memory / Memory to Peripheral). Memory striding is an useful feature especially in image processing field. For example, an image can be rotated easily with the help of XDMA memory stride feature. Thus we can reduce processor overhead to a large extend. XDMAC supports the following types of memory striding options.

5.1. Data Striding

Data striding is a method of accessing data in an interleaved (discontinuous) manner. 'Data Stride Length' is the memory gap (number of **bytes**) between successive data elements (Byte/Half Word/Word) stored in the memory. The default value of data stride length in XDMAC module is '0'. Therefore, the data elements are read/written in a continuous manner. If the value of data stride length is greater than zero (1, 2, 3, 4,..), then the data elements are accessed (read/written) in incremented interleaved manner. If the value of data stride length is set as -1 then the same data element (fixed address) is accessed again and again. If the value of data stride length is less than one (-2, -3, -4,..), then the data elements are accessed (read/written) in decremented interleaved manner. XDMAC supports data striding on both source and destination sides.

5.1.1. Destination Data Striding

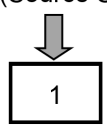
In this method, the data striding operation is performed only on destination side. So the source addressing mode is set as either Fixed Addressing Mode (FIXED_AM) or Incremented Addressing Mode (INCREMENTED_AM) based on the application requirement. The destination addressing mode should be set as **UBS_DS_AM** (the microblock stride is added at the microblock boundary, the data stride is added at the databoundary) in the **XDMAC_CCx.DAM** register field. The 'Data Stride Length' should be set in the **DDS_MSP** field of **XDMAC_CDS_MSPx**register. The following examples show how destination data striding is performed with different **DDS_MSP** values, but with fixed microblock length (ex: UBLEN =5). The following examples also assume that destination buffers are initialized with zero '0'.

Example 1:

Destination Data Stride Length is set as **1** in the XDMAC_CDS_MSPx.DDS_MSP register field.

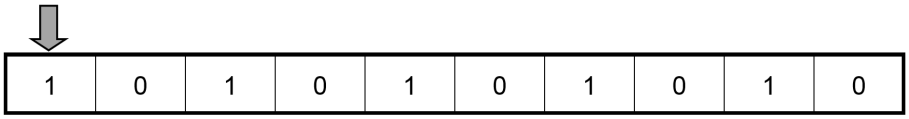
Source Data (Addressing Mode: FIXED_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_DS_AM)

(Destination Starting Address)

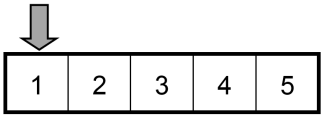


Example 2:

Destination Data Stride Length is set as **2** in the XDMAC_CDS_MSPx.DDS_MSP register field.

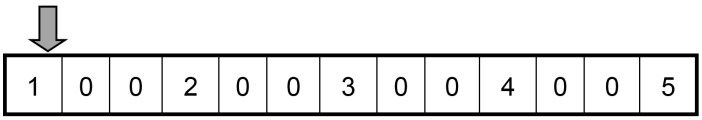
Source Data (Addressing Mode: INCREMENTED_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_DS_AM)

(Destination Starting Address)

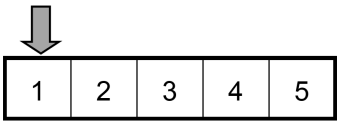


Example 3:

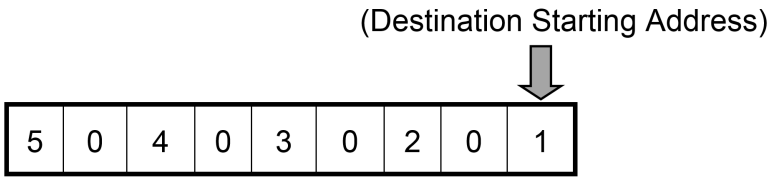
Destination Data Stride Length is set as **-3** in the XDMAC_CDS_MSPx.DDS_MSP register field.

Source Data (Addressing Mode: INCREMENTED_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_DS_AM)



5.1.2. Source Data Striding

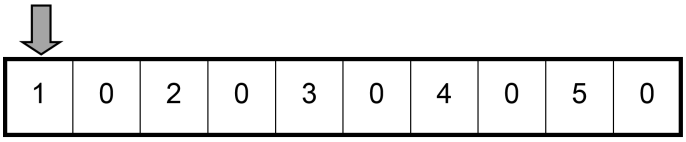
In this method, the data striding operation is performed only on source side. So the destination addressing mode is set as either Fixed Addressing Mode (FIXED_AM) or Incremented Addressing Mode (INCREMENTED_AM) based on the application requirement. The source addressing mode should be set as **UBS_DS_AM** (the microblock stride is added at the microblock boundary, the data stride is added at the data boundary) in the **XDMAC_CCx.SAM** register field. The 'Data Stride Length' should be set in the **SDS_MSP** field of **XDMAC_CDS_MSPx** register. The following examples show how source data striding is performed with different **SDS_MSP** values, but with fixed microblock length (ex: UBLEN =5).

Example 1:

Source Data Stride Length is set as **1** in the XDMAC_CDS_MSPx.SDS_MSP register field.

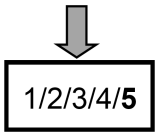
Source Data (Addressing Mode: UBS_DS_AM)

(Source Starting Address)



Destination Data (Addressing Mode: FIXED_AM)

(Destination Starting Address)



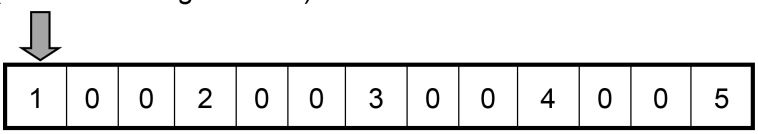
Note: Here the fixed destination address is overwritten with values 1, 2, 3, 4, and 5. So the destination address finally holds the value 5.

Example 2:

Source Data Stride Length is set as **2** in the XDMAC_CDS_MSPx.SDS_MSP register field.

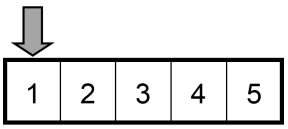
Source Data (Addressing Mode: UBS_DS_AM)

(Source Starting Address)



Destination Data (Addressing Mode: INCREMENTED_AM)

(Destination Starting Address)

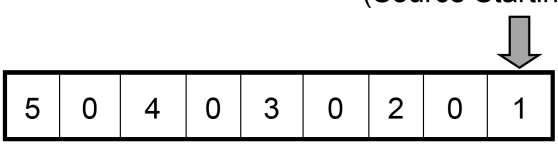


Example 3:

Source Data Stride Length is set as **-3** in the XDMAC_CDS_MSPx.SDS_MSP register field.

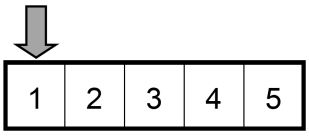
Source Data (Addressing Mode: UBS_DS_AM)

(Source Starting Address)



Destination Data (Addressing Mode: INCREMENTED_AM)

(Destination Starting Address)



5.1.3. Source and Destination Data Striding

In this method, the data striding operation is performed both on source and destination sides. Therefore, both source and destination addressing modes should be set as **UBS_DS_AM** in **XDMAC_CCx.SAM** and **XDMAC_CCx.DAM** register fields. The respective source and destination 'Data Stride Lengths' should be set in **SDS_MSP** and **DDS_MSP** fields of **XDMAC_CDS_MSPx** register. The following examples show how source and destination data striding are performed together with different **SDS_MSP** and **DDS_MSP** values, but with fixed microblock length (e.g.: UBLEN =5). The following examples also assume that destination buffers are initialized with zero '0'.

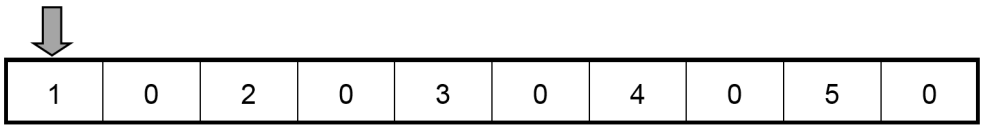
Example 1:

Source Data Stride Length is set as **1** in the XDMAC_CDS_MSPx.SDS_MSP register field.

Destination Data Stride Length is set as **-2** in the XDMAC_CDS_MSPx.DDS_MSP register field.

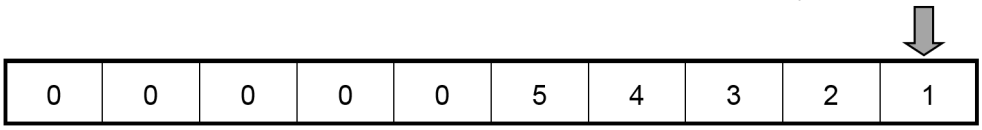
Source Data (Addressing Mode: UBS_DS_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_DS_AM)

(Destination Starting Address)



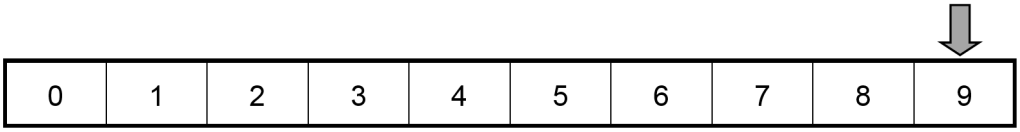
Example 2:

Source Data Stride Length is set as **-3** in the XDMAC_CDS_MSPx.SDS_MSP register field.

Destination Data Stride Length is set as **1** in the XDMAC_CDS_MSPx.DDS_MSP register field.

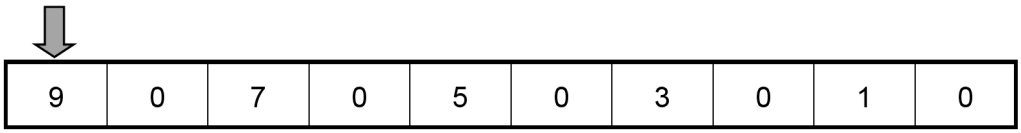
Source Data (Addressing Mode: UBS_DS_AM)

(Source Starting Address)



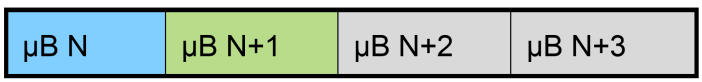
Destination Data (Addressing Mode: UBS_DS_AM)

(Destination Starting Address)

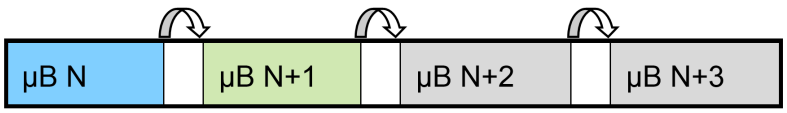


5.2. Microblock Striding

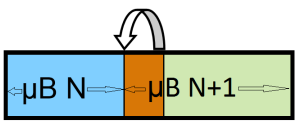
Microblock striding is a method of accessing microblocks in an interleaved (discontinuous) manner. 'Microblock Stride Length' is the memory gap (number of **bytes**) between successive microblocks stored in the memory. The default value of microblock stride length in XDMAC module is '0'. So there are no memory gaps (in **bytes**) between two successive microblocks as shown in the following picture.



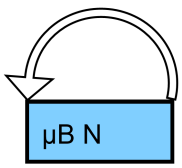
If the value of microblock stride length is greater than zero **(1, 2, 3,..)**, then there will be memory gaps (1 byte, 2 bytes, 3 bytes,..) between two successive microblocks as shown in the following picture.



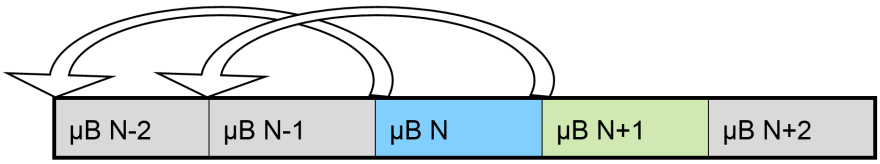
If the value of microblock stride length is less than zero **(-1, -2, -3,..)**, then there will be memory overlaps (1 byte, 2 bytes, 3 bytes,..) between two successive microblocks as shown in the following picture.



If the value of microblock stride length is set as **(–UBLEN)**, then the same microblock is accessed again and again, as shown in the following picture.



If the value of microblock stride length is set as **–(2*UBLEN)**, then we can access previous microblocks in decremented fashion as shown in the following picture.



Therefore, it becomes possible to access microblocks in different ways using microblock striding option. XDMAC supports microblock striding on both source and destination sides.

5.2.1. Destination Microblock Striding

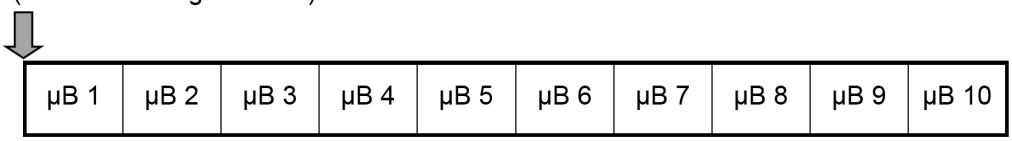
In this method, the microblock striding operation is performed only on destination side. So the source addressing mode is set as either Fixed Addressing Mode (FIXED_AM) or Incremented Addressing Mode (INCREMENTED_AM) based on the application requirement. The destination addressing mode should be set as either **UBS_AM** (The microblock stride is added at the microblock boundary) or **UBS_DS_AM** (the microblock stride is added at the microblock boundary, the data stride is added at the data boundary) in the **XDMAC_CCx.DAM** register field. The 'Microblock Stride Length' should be set in the **XDMAC_CDUSx** register. The following examples show how destination microblock striding is performed with different **XDMAC_CDUSx** register values, but with fixed block length (BLEN =4) and microblock length (UBLEN). The following examples also assume that destination buffers are initialized with zero '0'.

Example 1:

Destination Microblock Stride Length **XDMAC_CDUSx** is set equal to **UBLEN**.

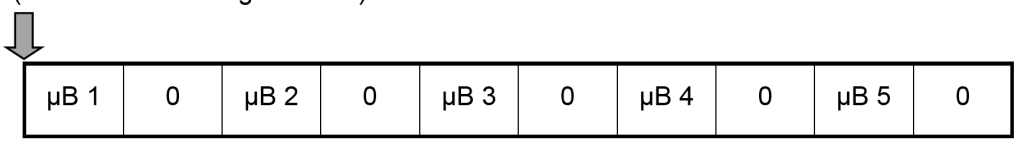
Source Data (Addressing Mode: INCREMENTED_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_AM or UBS_DS_AM)

(Destination Starting Address)

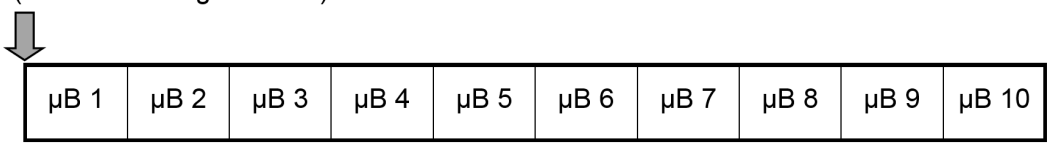


Example 2:

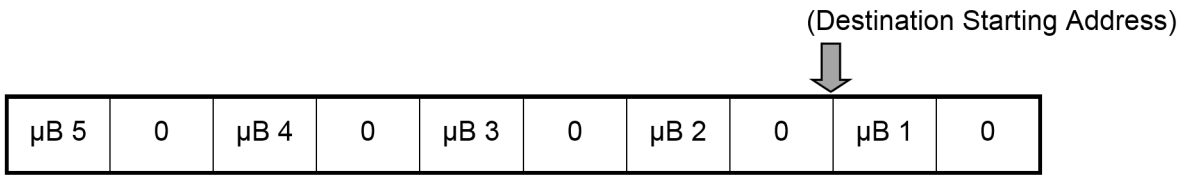
Destination Microblock Stride Length **XDMAC_CDUSx** is set equal to **–(3*UBLEN)**.

Source Data (Addressing Mode: INCREMENTED_AM)

(Source Starting Address)



Destination Data (Addressing Mode: UBS_AM or UBS_DS_AM)



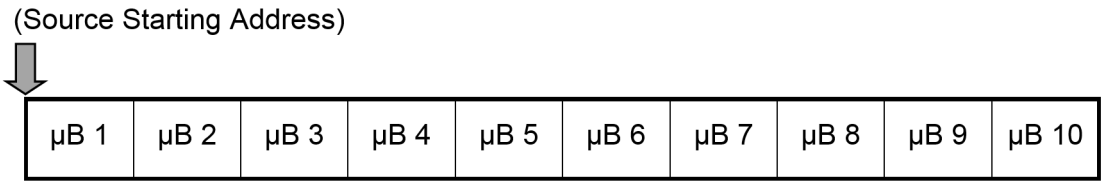
5.2.2. Source Microblock Striding

In this method, the microblock striding operation is performed only on source side. So the destination addressing mode is set as either Fixed Addressing Mode (FIXED_AM) or Incremented Addressing Mode (INCREMENTED_AM) based on the application requirement. The source addressing mode should be set as either **UBS_AM** or **UBS_DS_AM** in the **XDMAC_CCx.SAM** register field. The 'Microblock Stride Length' should be set in the **XDMAC_CSUSx** register. The following examples show how source microblock striding is performed with different **XDMAC_CSUSx** register values, but with fixed block length (e.g.: BLEN =4) and microblock length (UBLEN). The following examples also assume that destination buffers are initialized with zero '0'.

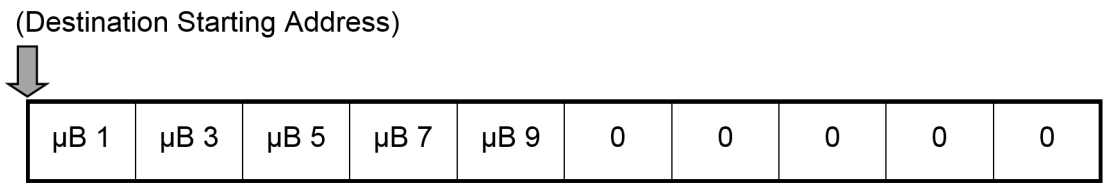
Example 1:

Source Microblock Stride Length **XDMAC_CSUSx** is set equal to **UBLEN**.

Source Data (Addressing Mode: UBS_AM or UBS_DS_AM)



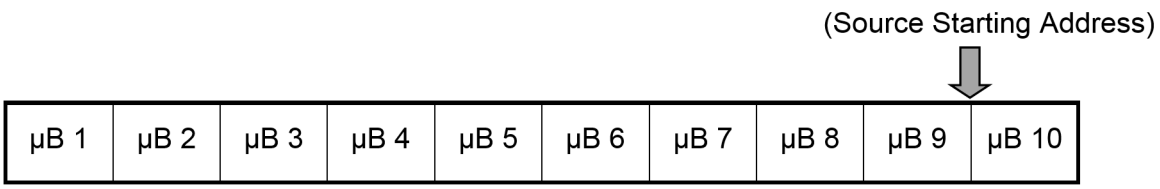
Destination Data (Addressing Mode: INCREMENTED_AM)



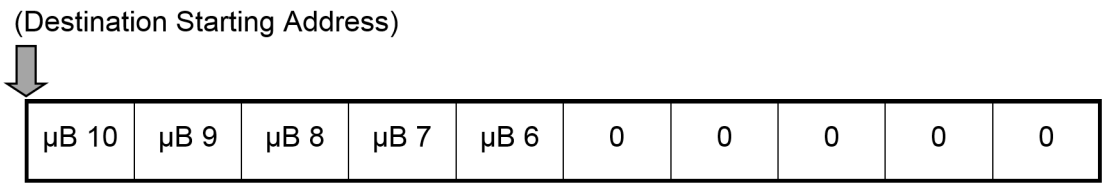
Example 2:

Source Microblock Stride Length **XDMAC_CSUSx** is set equal to **–(2*UBLEN)**.

Source Data (Addressing Mode: UBS_AM or UBS_DS_AM)



Destination Data (Addressing Mode: INCREMENTED_AM)



5.2.3. Source and Destination Microblock Striding

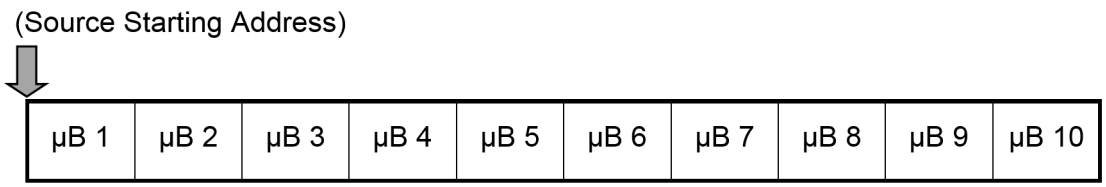
In this method, the microblock striding operation is performed on both source and destination sides. So both source and destination addressing modes should be set as either **UBS_AM** or **UBS_DS_AM** in **XDMAC_CCx.SAM** and **XDMAC_CCx.DAM** register fields. The respective source and destination 'Microblock Stride Lengths' should be set in **XDMAC_CSUSx** and **XDMAC_CDUSx** registers. The following examples show how source and destination microblock striding are performed together with different microblock stide length values, but with fixed block length (e.g.: BLEN=4) and microblock length (UBLEN). The following examples also assume that destination buffers are initialized with zero '0'.

Example 1:

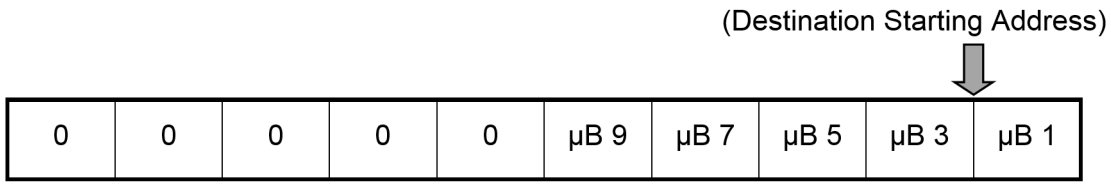
Source Microblock Stride Length **XDMAC_CSUSx** is set equal to **UBLEN**.

Destination Microblock Stride Length **XDMAC_CDUSx** is set equal to **–(2*UBLEN)**.

Source Data (Addressing Mode: UBS_AM or UBS_DS_AM)



Destination Data (Addressing Mode: UBS_AM or UBS_DS_AM)



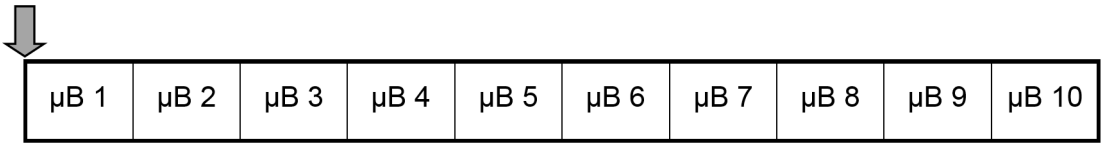
Example 2:

Source Microblock Stride Length **XDMAC_CSUSx** is set equal to **-UBLEN**.

Destination Microblock Stride Length **XDMAC_CDUSx** is set equal to **UBLEN**.

Source Data (Addressing Mode: UBS_AM or UBS_DS_AM)

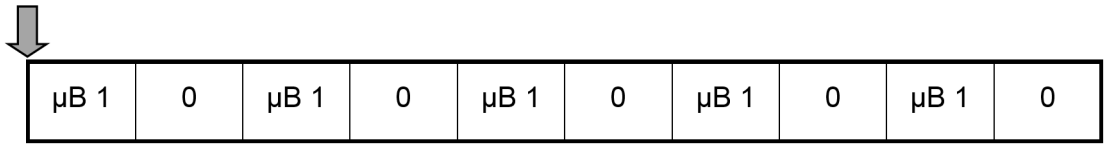
(Source Starting Address)



µB 1	µB 2	µB 3	µB 4	µB 5	µB 6	µB 7	µB 8	µB 9	µB 10

Destination Data (Addressing Mode: UBS_AM or UBS_DS_AM)

(Destination Starting Address)



µB 1	0	µB 1	0	µB 1	0	µB 1	0	µB 1	0

5.3. Block Striding

In XDMAC, block level striding can be performed using a linked list. As each descriptor represents a particular memory block, the source and destination addresses can be directly varied in each descriptor according to the required memory stride length.

6. Application Code - Getting Started

This application note also has a sample application code implemented on ATSAMV71Q21 device (SAM V71 Xplained Ultra Evaluation Kit). This application code is an Atmel Studio 7 project using ASF (Atmel Software Framework) driver functions. In order to program the application and view the output, the following steps are necessary.

- Connect ‘SAM V71 Xplained Ultra Evaluation Kit’ to the PC through ‘DEBUG USB’ port
- Open any of the serial terminal software (e.g.: TeraTerm) and open the “EDBG Virtual COM Port (COMx)” and make following settings:
  - Baud rate: 115200
  - Data: 8 bit
  - Parity: None
  - Stop: 1 bit
  - Flow control: None
- Build the application project (Atmel Studio 7 project) and program it to the kit (SAM V71 Xplained Ultra)
- The XDMAC Application Menu will be displayed in the serial terminal window. It will list down four demo tasks (1, 2, 3, and 4).
- You can choose the demo task by entering the task number (1/2/3/4) through key board

This application example has totally four demo tasks, which are explained in detailed in the following section.

7. Application Code - Demonstration

This application code demonstrates how to use both microblock and data striding together to rotate an array of data in different ways. This technique will be very useful in image processing field. This application code also demonstrates how to perform such different tasks in a continuous fashion (one by one) using a linked list. In this application note, only the 'memory to memory' (SRAM to SRAM) data transfer is demonstrated with various striding techniques. But it is also possible to perform memory striding with 'peripheral to memory' or 'memory to peripheral' data transfers.

All of the demonstration tasks (1, 2, 3, and 4) use the following settings/resources in common.

Data width (DWIDTH):

Data width (DWIDTH) is set as 'BYTE' in all demo tasks.

Microblock length(UBLEN):

The microblock length UBLEN is set as 10 in all demo tasks. Therefore, each microblock contains 10 bytes of data.

Block length(BLEN):

The block length BLEN is set as 9 in all demo tasks. Therefore, each block contains 10 microblocks (BLEN+1).

Source memory array (10x10 bytes):

The source memory array is a two dimensional array of size 10x10. It has pre-stored data values varying from 0 to 9. The first row elements are all filled with '0'. The next row elements are filled with '1' and so on. So the last row elements are filled with '9'. This will be helpful for us to visualize the resultant destination array (rotated) in a better manner. This same source memory array is being used in all demo tasks described below.

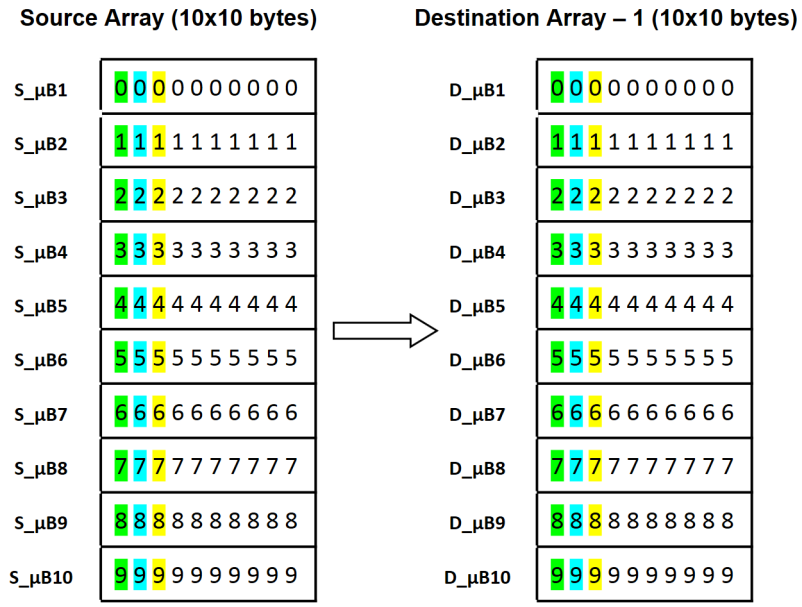
Destination memory arrays 1, 2, 3 (10x10 bytes):

There are totally three destination memory arrays each of same size (10x10 bytes). They are denoted as Destination Array-1, Destination Array-2, and Destination Array-3. All of them are two dimensional arrays, with array elements initialized to '0'.

7.1. Task 1

Task 1 demonstrates a simple memory to memory transfer without performing any memory striding operations. The source array is completely copied into the Destination Array-1 without any modifications. Destination Array-2 and Destination Array-3 are not used in Task 1. They are just initialized to 0. Linked list is also not used in Task 1.

Figure 7-1. Illustration of Task 1



The first row (10 elements) of the source array is considered as Source Microblock-1 (S_µB1). The second row of the source array is considered as Source Microblock-2 (S_µB2) and so on. So the last row elements of the source array are part of Source Microblock-10 (S_µB10). The similar microblock structure is followed in Destination Array-1 as well.

Source Microblock-1 (S_µB1) is completely copied to Destination Microblock-1 (D_µB1) without any modifications. In the same way, the Source Microblock-2 (S_µB2) is completely copied to Destination Microblock-2 (D_µB2) and so on.

Note: For demonstration purpose, the source array (block) is split into 10 microblocks, having 10 elements each. So, here BLEN is set as 9 and UBLEN is set as 10. Note that it is also possible to transfer the entire source array (block) in a single microblock, containing 100 elements. In this case, BLEN and UBLEN fields should be set as 0 and 100 respectively.

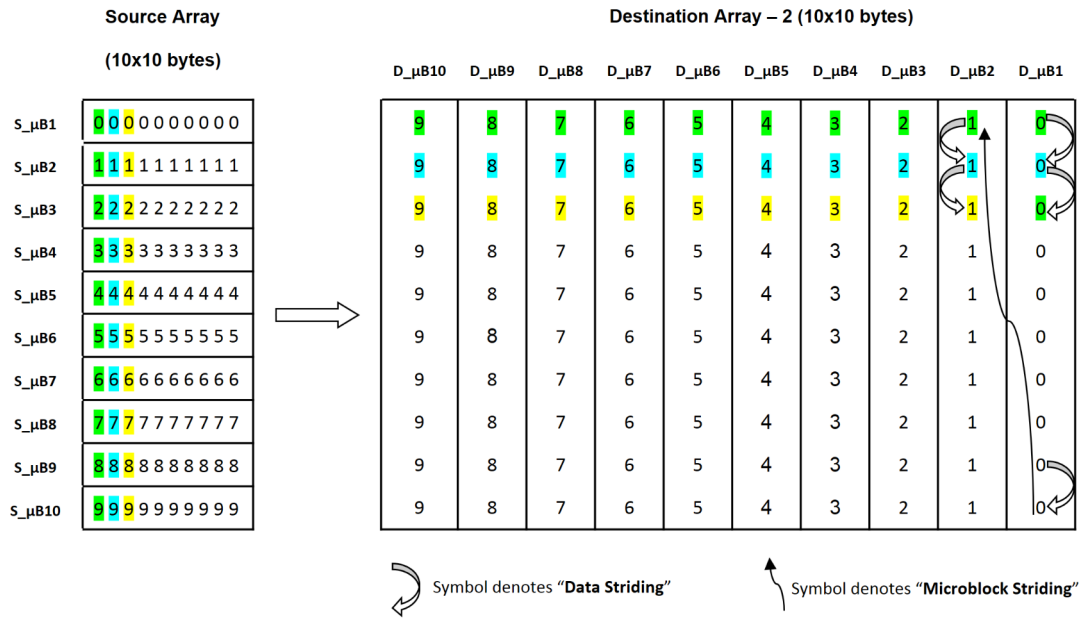
7.2. Task 2

Task 2 demonstrates how to perform ‘rotate right operation’ on a two dimensional array by using both ‘Destination Data Striding’ and ‘Destination Microblock striding’ functionalities together. Here the source array is copied and rotated right and stored into Destination Array-2. Destination Array-1 and Destination Array-3 are not used in Task-2. They are just initialized to 0. Linked list is also not used in Task2.

The first row (10 elements) of the source array is considered as Source Microblock-1 (S_µB1). The second row of the source array is considered as Source Microblock-2 (S_µB2) and so on. So the last row elements of source array are part of Source Microblock-10 (S_µB10).

The microblock structure of Destination Array-2 is different. It is based on the destination microblock striding pattern, which is required for rotating the array right (90° clock wise). The last column (10 elements) of the Destination Array-2 is considered as Destination Microblock-1 (D_µB1). The ninth column of the Destination Array-2 is considered as Destination Microblock-2 (D_µB2) and so on. So the first column elements of the Destination Array-2 are part of Destination Microblock-10 (D_µB10).

Figure 7-2. Illustration of Task 2



In order to rotate the array content right, the source microblocks have to be copied to their respective destination microblocks as shown in the above figure. The first source microblock (S_µB1 / first row) has to be copied to the first destination microblock (D_µB1 / last column). Similarly, the second source microblock (S_µB2 / second row) has to be copied to the second destination microblock (D_µB2 / 9th column) and so on.

To copy the first source microblock (S_µB1 / first row) into the first destination microblock (D_µB1 / last column) the following settings are necessary.

- The destination address has to be initialized with the starting address of the first destination microblock (D_µB1 / last column), which is 'dst_buf_2[0][9]'
- The destination data stride length has to be set as '9' in order to fill all of the remaining elements of the first destination microblock (D_ µB 1)

To copy the second source microblock (S_µB2 / second row) into the second destination microblock (D_µB2 / 9th column) the following settings are necessary.

- The 'destination micoblock stride length' has to be set as '-92' in order to change the destination address from the last element of D_ µB 1 (dst_buf_2[9][9]) to the first element of D_ µB 2 (dst_buf_2[0][8])
- The 'destination data stride length' remains same as '9' in order to fill all of the remaining elements of the second destination microblock (D_µB2 / 9th column)

The above mentioned settings will remain same for copying other microblocks as well.

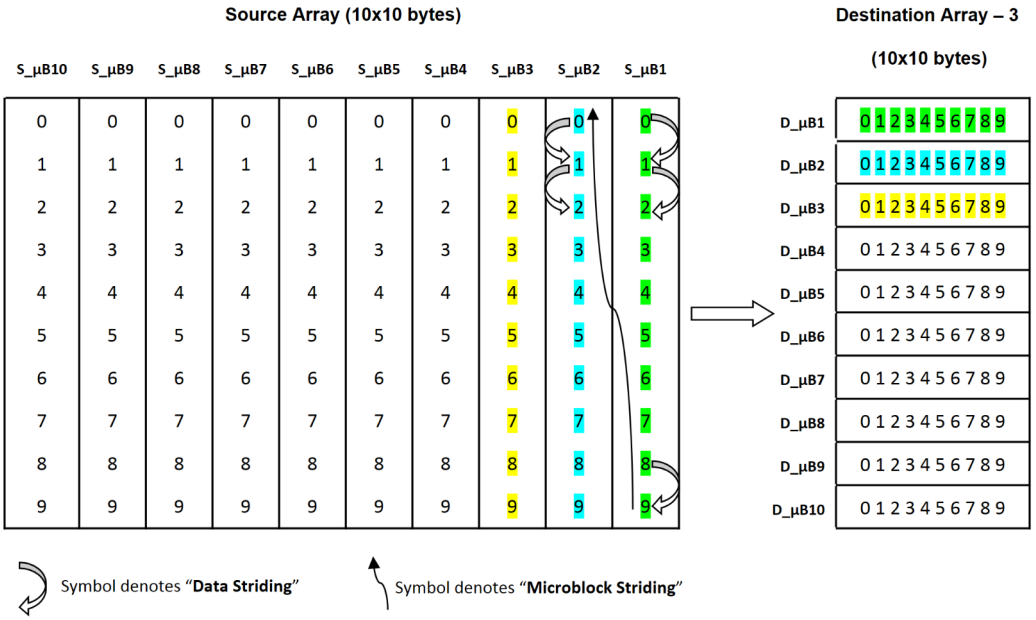
7.3. Task 3

Task 3 demonstrates how to perform 'rotate left operation' on a two dimensional array by using both 'Source Data Striding' and 'Source Microblock striding' functionalities together. Here the source array is copied and rotated left and stored into Destination Array-3. Destination Array-1 and Destination Array-2 are not used in Task-3. They are just initialized to 0. Linked list is also not used in Task3.

The first row (10 elements) of Destination Array-3 is considered as Destination Microblock-1 (D_µB1). The second row of Destination Array-3 is considered as Destination Microblock-2 (D_µB2) and so on. So the last row elements of Destination Array-3 are part of Destination Microblock-10 (D_µB10).

The microblock structure of Source Array is different here. It is based on the source microblock striding pattern, which is required for rotating the array left (90° counter clock wise). The last column (10 elements) of the Source Array is considered as Source Microblock-1 (S_µB1). The ninth column of the Source Array is considered as Source Microblock-2 (S_µB2) and so on. So the first column elements of Source Array are part of Source Microblock-10 (S_µB10).

Figure 7-3. Illustration of Task 3



In order to rotate the array content left, the source microblocks have to be copied to their respective destination microblocks as shown in the above figure. The first source microblock (S_µB1 / last column) has to be copied and stored in the first destination microblock (D_µB1 / first row). Similarly, the second source microblock (S_µB2 / 9th column) has to be copied and stored in the second destination microblock (D_µB2 / 2nd row) and so on.

To copy the first source microblock (S_µB1 / last column) and store it into the first destination microblock (D_µB1 / first row), the following settings are necessary.

- The source address has to be initialized with the starting address of the first source microblock (S_µB1 / last column), which is 'src_buf[0][9]'
- The source data stride length has to be set as '9' in order to fill all of the remaining elements of S_µB1

To copy the second source microblock (S_µB2 / 9th column) into the second destination microblock (D_µB2 / second row), the following settings are necessary.

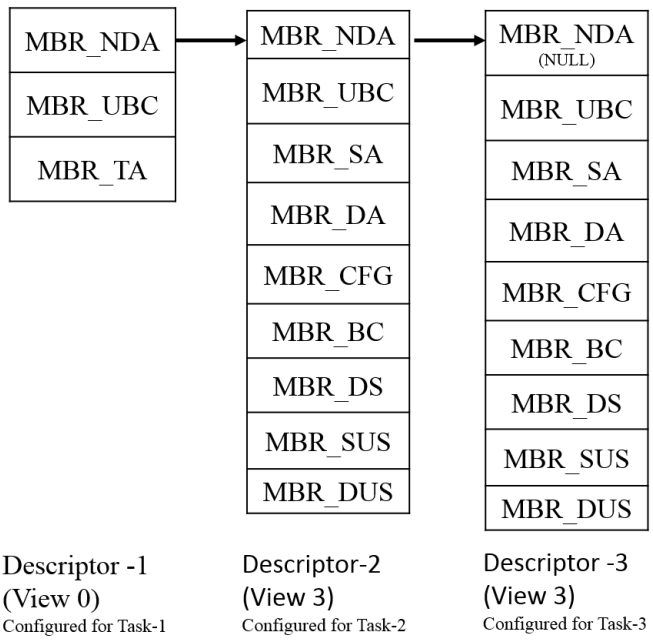
- The source micoblock stride length has to be set as '-92' in order to change the source address from the last element of S_µB1 (src_buf[9][9]) to the first element of S_µB2 (src_buf[0][8])
- The source data stride length remains same as '9' in order to fill all of the remaining elements of S_µB2

The above mentioned settings will remain same for copying other microblocks as well.

7.4. Task 4

Task 4 demonstrates how to perform Tasks 1, 2, and 3 in a sequential fashion using a linked list. Here a linked list is formed using three descriptors of type View 0, View 3, and View 3, which are configured for tasks 1, 2, and 3 respectively. As there are no memory striding operations involved in Task 1, View 0 descriptor is sufficient for Task 1. But Tasks 2 and 3 require View 3 descriptors, as they are mainly based on memory striding operations. Therefore, a linked list is formed as shown in Figure 7-4. and executed in Task 4.

Figure 7-4. Illustration of Task 4



8. Cache Coherence Management

If 'Data Cache' is enabled in the MCU, then ‘cache coherency’ should be maintained between data cache and main memory. Cache coherency management is required when several masters try to access the same memory location. For example, when CPU and XDMAC both try to access the same memory location, the cache coherency problem occurs. To avoid this problem, the following operations are necessary.

- ‘Clean D-Cache Operation’ should be performed before the XDMAC transfer
- ‘Invalidate D-Cache Operation’ should be performed after the XDMAC transfer

The knowledgebase article on Cache Coherence Management gives more information on this topic.

9. Conclusion

Therefore, it is possible to effectively use linked list and various memory striding functionalities of XDMAC peripheral to offload CPU processing time in various applications.

10. Frequently Asked Questions (FAQs)

In a single block transfer, how to get the number of 'Microblocks' transferred so far at a given point of time?

BLEN field of Channel Block Control Register (XDMAC_CBCx) gets decremented by one for each microblock being transferred. So you can calculate the number of microblocks transmitted, from the current BLEN value.

In a single block transfer, how to get the amount of 'Data' transferred so far in a microblock at a given point of time?

UBLEN field of Channel Microblock Control Register (XDMAC_CUBCx) gets decremented by MBSIZE (memory burst size) or CSIZE (chunk size) for each memory burst or chunk transfer. So you can calculate the amount of data transmitted, from the current UBLEN value.

I want to transfer only one block of memory. Do I have to use linked list?

Not necessarily. You can just configure XDMAC global and channel registers and start the data transfer. Linked list is not needed here. Linked list is used in multi block memory transfer.

11. References

SAM V71 Device Datasheet:

http://www.atmel.com/Images/Atmel-44003-32-bit-Cortex®-M7-Microcontroller-SAM-V71Q-SAM-V71N-SAM-V71J_Datasheet.pdf

Knowledgebase Article on Cache Coherence Management:

http://atmel.force.com/support/articles/en_US/Technical_Presentation/How-to-manage-Cortex-M7-Cache-Coherence-with-the-Atmel-SAM-S70-E70-DMAs

Wikipedia Web Page:

https://en.wikipedia.org/wiki/Stride_of_an_array

12. Revision History

Doc Rev.	Date	Comments
42761A	08/2016	Initial document release.

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