Application Note 239

Example programs for the CoreLink™ DMA Controller DMA-330

Document number: ARM DAI 0239A
Issued: 23rd September 2010

ARM®

Application Note 239
Example programs for the CoreLink™ DMA Controller DMA-330

Release information

The following table lists the changes made to this application note.

Change history

Date	Issue	Change
September 2010	A	First release

Proprietary notice

Confidentiality status

This document is Non-Confidential. This document has no restriction on distribution.

Feedback on this application note

If you have any comments on content then send an e-mail to `errata@arm.com`. Give:

- the document title
- the document number
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM web address

`http://www.arm.com`

Table of Contents

1 Introduction

This application note provides examples of how to program the CoreLink DMA Controller DMA-330.

1.1 References

A description of the *DMA Controller* (DMAC) including the programmers model and instruction set can be found in the *DMA-330 Technical Reference Manual*, (ARM DDI 0424) available from `http://infocenter.arm.com`.

1.2 Notation

The following conventions are used for the example programs:

- DMAC instructions are written in the following typeface : `DMACODE`
- program comments are designated by two semicolons
- instructions within loops are indented and nested loops are further indented.

1.2.1 Resource requirements

The example programs include comments to indicate how many lines of the DMA Controller's internal MFIFO data buffer are required by the program. *SR* indicates the static requirement and *DR* the dynamic requirement, for example:

```
;; MFIFO data buffer resource requirement: SR 0 DR 16
```

See the *MFIFO Usage Overview* appendix in the *DMA-330 Technical Reference Manual* for more information about the MFIFO data buffer, which is dynamically shared between channels.

2 Basic DMA Programs and Addressing

2.1 Simple copying from memory to memory

2.1.1 Scenario

Copy 32Kbytes from memory to memory.

AXI interface width is 64 bits.

2.1.2 Description

In this program, the bursts are programmed to the maximum AXI burst length of 16 beats so that each loop iteration (one `DMALD` and one `DMAST` instruction) transfers a total of 128 bytes. The loop count is 256, so the program transfers a total of 32Kbytes, using 256 bursts.

2.1.3 Program

```
;; simple block copy
;; MFIFO data buffer resource requirement: SR 0 DR 16
DMAMOV SAR 0xF0008000
DMAMOV DAR 0x10000000
DMAMOV CCR SB16 SS64 DB16 DS64
DMALP lc0 256
   DMALD
   DMAST
DMALPEND lc0
DMAEND
```

Note The lc0 in the `DMALP` and `DMALPEND` instructions specifies that the DMAC uses loop counter 0 to count the iterations. Specifying this is optional, and the DMA-330 assembler selects a loop counter if one is not specified in the source code.

2.1.4 Description

In this variation of the program, the individual AXI bursts are programmed to a length of 4 beats, which might be the 'natural' burst size used by an SDRAM controller, so that each loop iteration now contains 4 `DMALD` and 4 `DMAST` instructions to transfer the same 128 bytes. Using shorter bursts might result in more system-friendly use of the interconnect because it provides more opportunities for inter-burst arbitration. The loop count is 256, so this program also transfers a total of 32Kbytes but using 1024 bursts.

2.1.5 Program

```
;; simple block copy, smaller burst size
;; MFIFO data buffer resource requirement: SR 0 DR 4
DMAMOV SAR 0xF0008000
DMAMOV DAR 0x10000000
DMAMOV CCR SB4 SS64 DB4 DS64
DMALP lc0 256
   DMALD
   DMAST
   DMALD
   DMAST
   DMALD
   DMAST
   DMALD
   DMAST
DMALPEND lc0
DMAEND
```

Note Although the program interleaves the `DMALD` and `DMAST` instructions, the queuing resources in the DMA-330 mean that the AXI master interface might issue four, or more, AXI read transactions before it issues one of the AXI write transactions.

2.2 Template for copying arbitrary byte counts

2.2.1 Scenario

Copy from memory to memory.

The byte count is *not* a multiple of burst size.

AXI interface width is 64 bits.

2.2.2 Description

This program copies 699 bytes from memory to memory. It does this as follows:

1. Five bursts of 16×8 bytes.
2. One burst of 7×8 bytes.
3. One burst of 3 bytes.

This type of program might be used as a template for a software driver that needs to copy an arbitrary numbers of bytes. The constants in the template that control loop counts and burst sizes could be modified dynamically to suit the total number of bytes to transfer.

For simpler cases, where the byte count is a suitable multiple that does not require the extra bursts for the few odd bytes at the end, the software driver can choose a simpler template, or can replace the unnecessary instructions with `DMANOP` instructions.

Note See the *MFIFO Usage Overview* appendix in the *DMA-330 Technical Reference Manual* for examples that illustrate performance optimizations when either the source or destination address is not aligned to the burst boundary.

2.2.3 Program

```
;; example template for block copy, not a multiple of burst size
;; MFIFO data buffer resource requirement: SR 0 DR 16
DMAMOV SAR 0x10000000
DMAMOV DAR 0x20000000

;; start by copying 5 bursts of 16 x 8 bytes, total of 640 bytes
DMAMOV CCR SB16 SS64 DB16 DS64
DMALP lc0 5
   DMALD
   DMAST
DMALPEND lc0

;; now copy 1 burst of 7 x 8 bytes, 56+640 = total of 696 bytes
DMAMOV CCR SB7 SS64 DB7 DS64
DMALD
DMAST

;; now copy 1 burst of 3 x 1 byte, 3+696 = total of 699 bytes
DMAMOV CCR SB3 SS8 DB3 DS8
DMALD
DMAST

DMAEND
```

3 Advanced DMA Features

3.1 Scatter/gather

3.1.1 Scenario

Copy the first byte from each of the last 8 words at the end of each 4K block and gather them into a single compact structure.

AXI interface width is 32 bits.

3.1.2 Description

This program walks through 1Mbyte of address space, copying 8 bytes from the end of each 4Kbyte block address and gathering them to a single compact area of memory. The 8 bytes are spaced at addresses with a stride of 4 between them, as might be the case if these were peripheral ID registers on an AMBA APB bus. It uses the `DMAADDH` instruction to stride from one byte to the next, and again to stride from one block to the next.

You can use this program to scan through a peripheral area of address space and create a copy of all of the peripheral ID register values.

3.1.3 Program

```
;; gather operation - 8 bytes from the end of each 4K block
;; MFIFO data buffer resource requirement: SR 0 DR 2
DMAMOV SAR 0xF0000000
DMAMOV DAR 0x10000000
DMAMOV CCR SB1 SS8 DB2 DS32
DMALP lc0 256
   DMAADDH SAR, 4064    ;; advance to 8 words before end of 4K block
   DMALP lc1 8
      DMALD             ;; read one byte
      DMAADDH SAR, 3    ;; advance to start of next word
   DMALPEND lc1
   DMAST                ;; write 8 bytes (2 x 32-bit words)
DMALPEND lc0
DMAEND
```

3.2 Endianness swapping

3.2.1 Scenario

Copy a block of memory and swap the byte order within each 32-bit word.

AXI interface width is 128 bits.

3.2.2 Description

This program copies 4Kbytes from memory to memory and swaps the endianness within each 32-bit word.

This might be used where one processor is interpreting the content of memory as an array of little-endian words, and another is interpreting it as an array of big-endian words. Using this feature of the DMAC could reduce the load on a processor that would otherwise have to perform this reversal in software.

3.2.3 Program

```
;; block copy with endianness reversal equal to data beat size
;; MFIFO data buffer resource requirement: SR 0 DR 4
DMAMOV SAR 0xF0008000
```

```
DMAMOV DAR 0x10000000
DMAMOV CCR SB16 SS32 DB16 DS32 ES32
DMALP lc0 64
   DMALD
   DMAST
DMALPEND lc0
DMAEND
```

3.2.4 Description

This variant of the previous program produces the same end result, but transfers 128 bits of data in each beat to make efficient use of the AXI infrastructure. This illustrates that the DMAC can endian-swap multiple 32-bit words in a single cycle.

3.2.5 Program

```
;; block copy with multiple endianness reversals within each data beat
;; MFIFO data buffer resource requirement: SR 0 DR 16
DMAMOV SAR 0xF0008000
DMAMOV DAR 0x10000000
DMAMOV CCR SB16 SS128 DB16 DS128 ES32
DMALP lc0 16
   DMALD
   DMAST
DMALPEND lc0
DMAEND
```

3.3 Byte reversing a large block of memory

3.3.1 Scenario

Copy a block of memory and reverse the order of all of the bytes.

3.3.2 Description

This simple program reads 256 bytes from addresses in descending order and stores them at addresses in ascending order. It is effectively endian-swapping at a size of 256 bytes. It does not make efficient use of the AXI infrastructure because data is transferred one byte at a time.

3.3.3 Program

```
;; reverse the order of 256 bytes
;; illustrates address arithmetic with subtraction
;; MFIFO data buffer resource requirement: SR 0 DR 1
DMAMOV SAR 0x10000000
DMAMOV DAR 0x20000000
DMAMOV CCR SB1 SS8 DB1 DS8

DMAADDH SAR, 255       ;; adjust source address to point at last byte

DMALP lc0 256
   DMALD               ;; read 1 byte
   DMAADNH SAR, 0xFFFE ;; subtract 2 to skip back behind that byte
   DMAST               ;; write 1 byte
DMALPEND lc0

DMAEND
```

3.3.4 Description

This variant of the previous program uses the endianness-swapping feature of the DMAC to perform the task more efficiently. It reads 64 words from addresses in descending order and writes them to addresses in ascending order. The *ES32* in the `DMAMOV CCR` instruction directs the DMAC to reverse the order of the four bytes in each 32-bit access.

3.3.5 Program

```
;; reverse the order of 256 bytes
;; illustrates address arithmetic with subtraction & endianness-swap
;; MFIFO data buffer resource requirement: SR 0 DR 1
DMAMOV SAR 0x10000000
DMAMOV DAR 0x20000000
DMAMOV CCR SB1 SS32 DB1 DS32 ES32

DMAADDH SAR, 252       ;; adjust source address to point at last word

DMALP lc0 64
   DMALD               ;; read 4 bytes
   DMAADNH SAR, 0xFFF8 ;; subtract 8 to skip back behind that word
   DMAST               ;; write 4 bytes in endian-swapped order
DMALPEND lc0

DMAEND
```

4 Interactions with Software Drivers

4.1 Issuing DMA instructions from a software driver

A software driver running on an ARM processor can interrogate the status and control the operation of the DMAC by accessing the APB slave interfaces. This process is described in more detail in *Using the APB slave interfaces* in the *Functional Overview* chapter of the *DMA-330 Technical Reference Manual*.

A software driver instructs the DMAC to start execution of a DMA channel program by using one of the APB interfaces to inject a `DMAGO` instruction. The driver must poll the DMAC to ensure that a channel is idle before it attempts to inject a `DMAGO` for that channel.

A software driver sends events to a DMA channel program by using one of the APB interfaces to inject a `DMASEV` instruction. The DMA channel program includes a corresponding `DMAWFE` instruction to react to this event. See *Software driver using events to control the progress of* a memory copy on page 11.

A software driver instructs the DMAC to terminate execution of a DMA channel program by using one of the APB interfaces to inject a `DMAKILL` instruction. This might be used in an error case, for example where a peripheral is not able to produce or accept the expected data for a DMA channel program that is in progress. This might also be used to terminate DMA channel programs that use the `DMALPFE` instruction to create an infinite loop, such as the program shown in *Complex interaction with software driver - using WFE invalid* on page 12.

4.2 Signaling to a software driver using interrupts

4.2.1 Scenario

Copy 64Kbytes from memory to memory and send an interrupt to software when complete.

AXI interface width is 32 bits.

4.2.2 Description

In this program, the DMAC sets an event to generate an interrupt to the software driver running on the ARM processor. The `DMAWMB` instruction ensures that all of the queued write operations are complete before the DMAC sends the interrupt. This avoids a race condition between the DMAC and the driver software.

4.2.3 Program

```
;; nested loop block copy with interrupt at end of task
;; MFIFO data buffer resource requirement: SR 0 DR 8
DMAMOV SAR 0x10000000
DMAMOV DAR 0x20000000
DMAMOV CCR SB4 SS32 DB4 DS32    ;; 4 x 4 = 16 bytes per transaction

DMALP lc0 16      ;; 16 loops x 4KBytes
   DMALP lc1 128  ;; 128 loops x 32 bytes
      DMALD
      DMALD
      DMAST
      DMAST
   DMALPEND lc1
DMALPEND lc0

DMAWMB            ;; wait for queued stores to complete
```

```
DMASEV e3         ;; raise interrupt to indicate task finished
DMAEND
```

4.3 Software driver using events to control the progress of a memory copy

4.3.1 Scenario

Copy 64Kbytes from memory to memory, with external software indicating when each block can start.

AXI interface width is 32 bits.

4.3.2 Description

In this program, the DMAC pauses before each 4Kbyte block until the software driver on the ARM processor signals that it can continue. For example, this might be used if software is gradually producing the data to be moved, or to throttle the load that the DMAC places on a memory controller that is shared with other bus masters.

When the DMAC reaches the `DMAWFE` instruction it pauses until the software driver has written to the event register to set the event (e1). Then the DMAC clears the event and continues execution – performing one complete inner loop of 128×2 read bursts and 128×2 write bursts to transfer 4Kbytes, and then sending an interrupt (e2) to indicate that it has finished that block of data.

Note The ordering between the DMAC executing the first `DMAWFE e1` instruction and the software driver writing to the event register is unimportant. If the DMAC reaches the `DMAWFE` instruction *before* the software driver has set the event (e1) then the DMAC channel thread pauses until that event is set. If the DMAC reaches the `DMAWFE` instruction *after* the software driver has set the event then the DMAC pauses for just one cycle to clear the event, and then immediately continues execution.

4.3.3 Program

```
;; block copy, throttled using events
;; MFIFO data buffer resource requirement: SR 0 DR 8
DMAMOV SAR 0x10000000
DMAMOV DAR 0x20000000
DMAMOV CCR SB4 SS32 DB4 DS32

DMALP lc0 16
   DMAWFE e1     ;; wait for CPU driver software to signal to DMAC
   DMALP lc1 128 ;; transfer 4Kbytes in inner loop
      DMALD
      DMALD
      DMAST
      DMAST
   DMALPEND lc1
   DMASEV e2     ;; raise interrupt to indicate that 4K was processed
DMALPEND lc0

DMAWMB           ;; wait for queued stores to complete
DMASEV e3        ;; raise interrupt to indicate whole task finished
DMAEND
```

4.4 Complex interaction with software driver - using WFE invalid

4.4.1 Scenario

Copy 4Kbyte blocks from memory to memory, with external software updating the source and destination addresses before each block is copied.

AXI interface width is 32 bits.

4.4.2 Description

In this program, the DMAC pauses before each 4Kbyte block until the software driver on the ARM processor signals that it can continue. The DMAC then executes the `DMAMOV` instructions that set the source and destination address for that block.

This program uses the `DMAWFE e1, invalid` instruction to invalidate (flush) the DMAC instruction cache, to ensure that the DMAC uses the address values contained in the *updated* `DMAMOV` opcodes.

Note A `DMASEV e4` instruction to signal from the DMAC to the ARM processor follows immediately after the `DMAMOV` instructions. Therefore, after the DMAC loads its address registers with the *current* block addresses, the processor can begin updating the opcodes in the DMA channel program memory with the values for the *next* block to be copied. When the DMAC completes the 4Kbyte block copy and returns to the `DMAWFE e1` instruction, the processor might have already signaled event e1 so that the DMAC can proceed without stalling.

For convenience, the software driver that inserts the 32-bit address values into the opcodes, might store these values at word-aligned addresses. The two `DMANOP` instructions, prior to the `DMAMOV DAR` instruction, adjust the alignment of the opcode bytes to ensure this.

To terminate the infinite loop in this program, the software driver can use an APB interface to inject a `DMAKILL` instruction.

4.4.3 Program

```
;; block copy, addresses updated by software
;; MFIFO data buffer resource requirement: SR 0 DR 8
DMAMOV CCR SB4 SS32 DB4 DS32
DMALPFE        ;; loop for ever

   DMAWFE e1, invalid      ;; wait for CPU to signal to DMAC
   DMAMOV SAR 0x00000000   ;; operand value updated by CPU
   DMANOP                  ;; adjust alignment of operand in opcode
   DMANOP
   DMAMOV DAR 0x00000000   ;; operand value updated by CPU
   DMASEV e4   ;; raise interrupt to indicate addresses have been read

   DMALP lc1 128           ;; transfer 4Kbytes in inner loop
      DMALD
      DMALD
      DMAST
      DMAST
   DMALPEND lc1

   DMAWMB      ;; wait for queued stores to complete
   DMASEV e3   ;; raise interrupt to indicate that 4K was processed

DMALPEND       ;; loop for ever

DMAEND         ;; never executed because of infinite loop
```