PrimeCell® µDMA Controller (PL230)

Revision: r0p0

Technical Reference Manual

ARM®

ARM DDI 0417A

PrimeCell µDMA Controller (PL230)
Technical Reference Manual

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
19 March 2007	A	Non-Confidential	First release for r0p0

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
PrimeCell µDMA Controller (PL230) Technical Reference Manual

List of Tables
PrimeCell µDMA Controller (PL230) Technical Reference Manual

List of Figures
PrimeCell µDMA Controller (PL230) Technical Reference Manual

Preface

This preface introduces the *PrimeCell µDMA Controller (PL230) Technical Reference Manual*. It contains the following sections:

About this manual

This is the *Technical Reference Manual* (TRM) for the *PrimeCell µDMA Controller (PL230)*.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This manual is written for system designers, system integrators, and verification engineers who are designing a *System-on-Chip* (SoC) device that includes the *µDMA Controller* (µDMAC). The manual describes the external functionality of the µDMAC.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for a high-level view of the controller and a description of its features.

Chapter 2 *Functional Overview*

Read this chapter for a description of the major components of the controller and how they operate.

Chapter 3 *Programmer's Model*

Read this chapter for a description of the registers.

Chapter 4 *Programmer's Model for Test*

Read this chapter for a description of the test registers.

Appendix A *Signal Descriptions*

Read this appendix for a description of the input and output signals.

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xii
- *Signals* on page xii
- *Numbering* on page xiii.

Typographical

The typographical conventions are:

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace` Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic` Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold` Denotes language keywords when used outside example code.

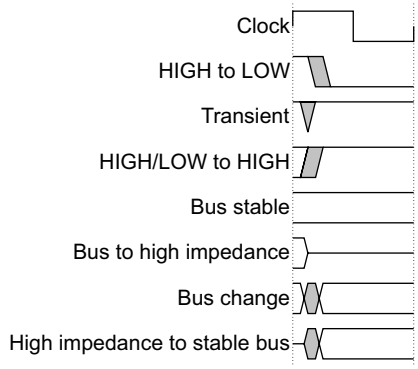
< and > Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example:

- `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
- The Opcode_2 value selects which register is accessed.

Timing diagrams

The figure named *Key to timing diagram conventions* explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.

Lower-case n Denotes an active-LOW signal.

Prefix H Denotes *Advanced High-performance Bus* (AHB) signals.

Prefix P Denotes *Advanced Peripheral Bus* (APB) signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog® method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications by ARM, and by third parties.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the µDMAC. See the following documents for other relevant information:

- *PrimeCell µDMA Controller (PL230) Implementation Guide* (ARM DII 0181)
- *PrimeCell µDMA Controller (PL230) Integration Manual* (ARM DII 0182)
- *PrimeCell µDMA Controller (PL230) Configuration Guide*
- *AMBA® 3 AHB-Lite Protocol v1.0 Specification* (ARM IHI 0033)
- *AMBA Specification (Rev 2.0)* (ARM IHI 0011).

Other publications

This section lists relevant documents published by third parties:

- JEDEC Solid State Technology Association, *JEP106, Standard Manufacturer's Identification Code*, obtainable at `http://www.jedec.org`.

Feedback

ARM welcomes feedback on the µDMAC and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the µDMAC. It contains the following sections:

1.1 About the µDMAC

The µDMAC is an *Advanced Microcontroller Bus Architecture* (AMBA) compliant *System-on-Chip* (SoC) peripheral that is developed, tested, and licensed by ARM.

It is a very low gate count DMA controller that is compatible with the AMBA AHB-Lite protocol.

1.1.1 Features of the controller

The principal features are that:

- it is compatible with AHB-Lite for the DMA transfers
- it is compatible with APB for programming the registers
- it has a single AHB-Lite master for transferring data using a 32-bit address bus and 32-bit data bus
- it has a configurable number of DMA channels
- each DMA channel has dedicated handshake signals
- each DMA channel has a programmable priority level
- each priority level arbitrates using a fixed priority that is determined by the DMA channel number
- it supports multiple transfer types:
  - memory-to-memory
  - memory-to-peripheral
  - peripheral-to-memory
- it supports multiple DMA cycle types
- it supports multiple DMA transfer data widths
- each DMA channel can access a primary, and alternate, channel control data structure
- all the channel control data is stored in system memory in little-endian format
- it performs all DMA transfers using the SINGLE AHB-Lite burst type
- the destination data width is equal to the source data width
- the number of transfers in a single DMA cycle can be programmed from 1 to 1024
- the transfer address increment can be greater than the data width

- it has a single output to indicate when an ERROR condition occurs on the AHB bus.

1.2 Terminology

This manual uses the following terminology:

Alternate The alternate channel control data structure. You must configure the register that *Channel primary-alternate set* on page 3-19 describes, to enable the controller to use this data structure.

C Identifies a specific DMA channel number. For example:

C=1 DMA channel 1.

C=23 DMA channel 23.

Channel You can configure the controller to contain up to 32 channels. Each channel contains independent and dedicated handshakes that can trigger a DMA transfer.

Channel control data

A data structure located in your system memory. You must program this data structure, so that the controller can perform the DMA transfers that you require. The controller must be able to access the location in system memory.

Note

Where this document mentions *data structure* it is referring to a channel control data structure.

DMA cycle All the DMA transfers that the controller must perform, to transfer the N data packets.

DMA transfer

The action of transferring a single byte, halfword, or word.

N The total number of DMA transfers that the controller performs for a channel.

`PL230_DMA_CHNLS`

The number of DMA channels that the controller contains. You must use the tools provided, to configure this configuration option before you integrate the controller in a SoC design.

Ping-pong For a given channel, the controller receives an initial request and then it performs a DMA cycle using the primary, or alternate, data structure. After it completes this DMA cycle, it starts a DMA cycle using the other data structure. The controller signals the completion of each DMA cycle to enable the host processor to reconfigure the inactive data structure. The controller continues to switch from primary to alternate to primary… until it reads a data structure that is invalid, or completes without switching to the opposite data structure.

Primary The primary channel control data structure. The controller uses this data structure when the corresponding bit in the chnl_pri_alt_set Register is 0.

R R is raised to the power of 2 and sets the number of DMA transfers that can occur before the controller rearbitrates. The number of DMA transfers are programmable from 1 to 1024, in binary steps from 2^0 to 2^{10}.

Scatter-gather

For a given channel, the controller receives a request from a peripheral and then performs four DMA transfers using the primary data structure, which configures the alternate data structure. It then immediately starts a DMA cycle using the alternate data structure. After this cycle completes, if the peripheral provides another request, the controller performs another four transfers using the primary data structure, which reprograms the alternate data structure. It then immediately starts a DMA cycle using the alternate data structure.

The controller continues to switch from primary to alternate to primary… until either it reads an invalid data structure, or the host processor configures the alternate data structure for a basic cycle. The controller asserts **dma_done[ ]** when the scatter-gather transaction completes using a basic cycle.

Chapter 2
Functional Overview

This chapter describes the major functional blocks of the µDMAC. It contains the following sections:

2.1 Functional description

Figure 2-1 shows a simplified block diagram of the controller.

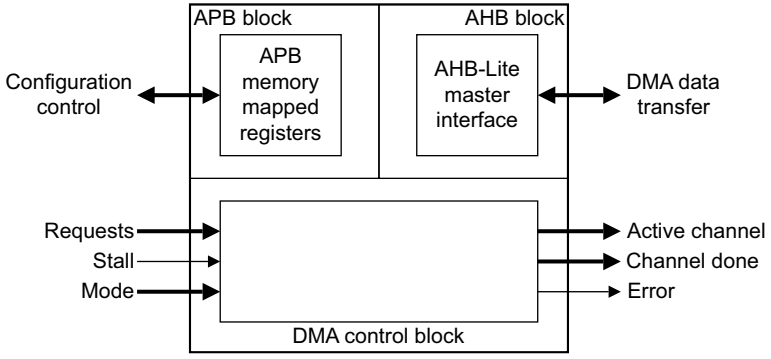


Figure 2-1 Block diagram

The controller contains the following main functional blocks:

- *APB block*
- *AHB block* on page 2-3
- *DMA control block* on page 2-3.

2.1.1 APB block

The APB block contains the registers that enable you to configure the controller by using the APB slave interface. It has 4KB of memory allocated to it and Chapter 3 *Programmer's Model* and Chapter 4 *Programmer's Model for Test* describe the registers.

Figure 2-2 shows the APB external connections.

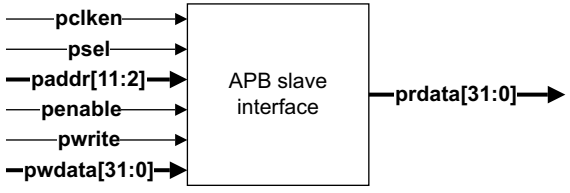


Figure 2-2 APB slave interface

2.1.2 AHB block

The controller contains a single AHB-Lite master that enables it to transfer data from a source AHB slave to a destination AHB slave using a 32-bit data bus.

The controller is compliant to the AMBA 3 AHB-Lite protocol. For detailed information about the AHB-Lite interface, see the *AMBA 3 AHB-Lite Protocol v1.0 Specification*.

Figure 2-3 shows the AHB-Lite master external connections.

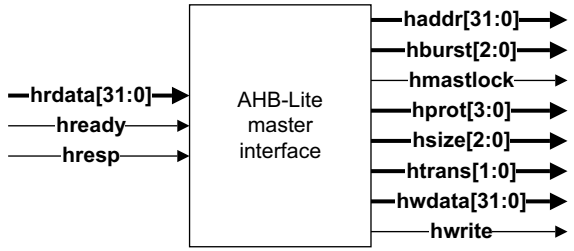


Figure 2-3 AHB-Lite master interface

2.1.3 DMA control block

This block contains the control logic that provides the following features:

- arbitrates the incoming requests
- indicates which channel is active
- indicates when a channel is complete
- indicates when an ERROR has occurred on the AHB-Lite interface
- enables slow peripherals to stall the completion of a DMA cycle
- waits for a request to clear before completing a DMA cycle
- performs multiple or single DMA transfers for each request
- perform the following types of DMA transfers:
  - memory-to-memory
  - memory-to-peripheral
  - peripheral-to-memory.

Note

Peripheral-to-peripheral transactions are not supported because each channel only provides a single DMA request interface.

Figure 2-4 shows the DMA control external connections.

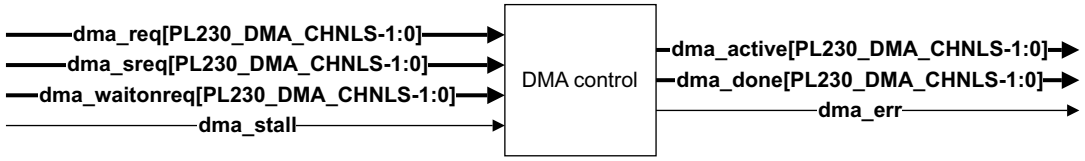


Figure 2-4 DMA control

2.1.4 Example system configuration

Figure 2-5 shows an example system containing the controller.

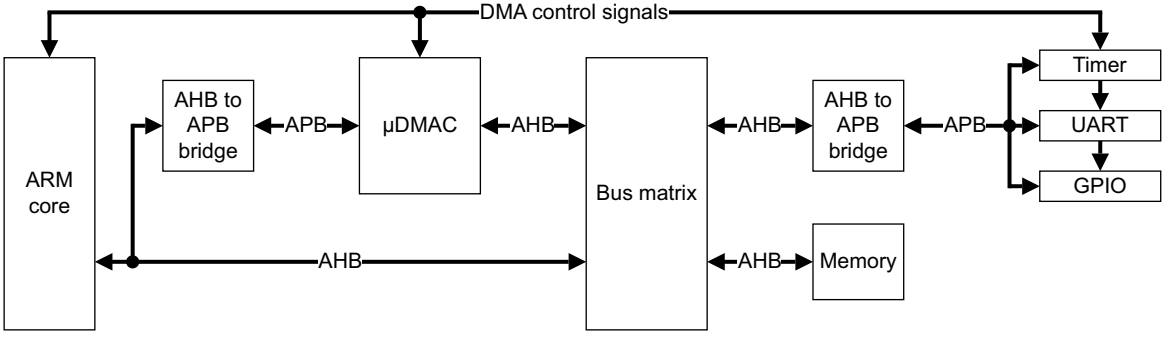


Figure 2-5 Example system

2.2 Functional operation

The following sections describe the operational functionality of the controller:

- *APB slave interface*
- *AHB master interface*
- *DMA control* on page 2-8
- *Channel control data structure* on page 2-35.

2.2.1 APB slave interface

The APB slave interface connects the controller to the APB and provides a host controller with access to the registers.

The APB slave interface supports the following features:

- read and write word accesses
- 32-bit data bus interface.

The controller implements the AMBA 2.0 APB protocol because the APB slave interface does not support wait states or error responses. For detailed information about the APB interface, see the *AMBA Specification (Rev 2.0)*.

2.2.2 AHB master interface

The following sections describe the features of this interface:

- *Transfer types*
- *Transfer data width* on page 2-6
- *Protection control* on page 2-6
- *Address increments* on page 2-7.

Transfer types

The controller does not support burst transfers and therefore it ties the **HBURST** signals LOW. The controller performs SINGLE AHB-Lite bus transfers using the **HTRANS** signaling combinations that Table 2-1 lists.

Table 2-1 HTRANS signaling

HTRANS[1]	HTRANS[0][a]	Description
0	0	IDLE transfer
1	0	NONSEQ transfer

a. This signal is tied LOW.

The absence of burst transfers has minimal effect on system performance because burst transfers are more effective in single-layer AHB systems, where a device has to request the bus, or access off-chip memory. The µDMAC is intended to be used in multi-layer AHB-Lite systems that include on-chip memory.

Transfer data width

The controller supports data transfer sizes of 8, 16, or 32 bits. Table 2-2 lists the supported **HSIZE** signaling combinations.

Table 2-2 HSIZE signaling

HSIZE[2][a]	HSIZE[1]	HSIZE[0]	Data width size (bits)
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	-[b]

a. This signal is tied LOW.
b. The controller does not provide this combination of **HSIZE** signaling.

The controller always uses 32-bit data transfers when it accesses a channel control data structure. You must set the source data transfer size to be identical to the destination data transfer size.

Protection control

The controller enables you to configure the AHB-Lite protection control signals, **HPROT[3:1]**. You can set these signals to indicate the following protection states:

- cacheable
- bufferable
- privileged.

Table 2-3 lists the **HPROT** signal encoding.

Table 2-3 Protection signaling

HPROT[3] Cacheable	HPROT[2] Bufferable	HPROT[1] Privileged	HPROT[0] Data/Opcode	Description
-	-	-	1 [a]	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Non-bufferable
-	1	-	-	Bufferable
0	-	-	-	Non-cacheable
1	-	-	-	Cacheable

a. The controller ties **HPROT[0]** HIGH, to indicate a data access.

For each DMA cycle, you can configure the source transfer and destination transfer to use different protection control settings. See *Control data configuration* on page 2-41 for information about how to configure protection control for these accesses.

You can also configure separate protection control settings when you access the channel control data structure. See *DMA configuration* on page 3-7 for information about how to configure protection control for these accesses.

Address increments

The controller enables you to configure the address increments that it uses when it reads the source data or when it writes the destination data. The increments available depend on the size of data packet being transferred. Table 2-4 lists the possible combinations.

Table 2-4 Address increments

Packet data width (bits)	Size of address increment
8	byte, halfword, or word
16	halfword or word
32	word

The minimum address increment must always be equal in size to the width of the data packet. The maximum address increment that the controller permits is one word.

To change the address increment, see *Control data configuration* on page 2-41. This provides you with information about which bits to configure in the channel control data structure.

Note

If you require the source or destination address to remain constant, for example, when accessing a FIFO then you can configure the controller to use a fixed address. See *Control data configuration* on page 2-41.

2.2.3 DMA control

This section describes:

- *Handshake rules*
- *DMA signaling* on page 2-11
- *DMA arbitration rate* on page 2-18
- *Priority* on page 2-19
- *DMA cycle types* on page 2-21
- *Error signaling* on page 2-35.

Handshake rules

The controller uses the DMA handshake rules that Table 2-5 on page 2-9 lists, when the following conditions apply:

- You enable a channel. That is, the chnl_enable_set [C] and master_enable bits are set to 1.
- You do not mask the **dma_req[C]** and **dma_sreq[C]** request inputs. That is, the chnl_req_mask_set [C] bit is 0.
- The controller is not operating in test mode. That is, the int_test_en bit is 0, see *Integration configuration* on page 4-3.

Table 2-5 Rules when channels and enabled and requests are not masked

Rule	Description
1	When **dma_active[C]** is LOW, then setting **dma_req[C]** or **dma_sreq[C]** HIGH for one or more **hclk** cycles, contiguous or non-contiguous, starts a transfer for channel C.
2	The controller only permits a single **dma_active[]** to be HIGH.
3	The controller sets **dma_active[C]** HIGH, when it starts a transfer for channel C.
4	For DMA cycle types other than peripheral scatter-gather, **dma_active[C]** remains HIGH until the controller completes the lesser number of transfers, that either 2^R or n_minus_1[a] specifies. In peripheral scatter-gather mode, **dma_active[C]** remains HIGH during each primary-alternate pair of DMA transfers. That is, the controller performs 2^R transfers using the primary data structure, and then without arbitrating, it performs the lesser number of transfers, that either 2^R or n_minus_1[a] specifies, using the alternate data structure. After completing the latter DMA transfer, **dma_active[C]** goes LOW.
5	The controller sets **dma_active[C]** LOW, for at least one **hclk** cycle, before it sets **dma_active[C]** or **dma_active[]** HIGH.
6	For channels that are enabled, the controller only permits a single **dma_done[]** to be HIGH.
7	If **dma_req[C]** is HIGH when **dma_active[C]** or **dma_stall** are HIGH then the controller only detects a request if **dma_req[C]** was low for the previous clock cycle.
8	If you set the cycle_ctrl bits for a channel to 3'b100, 3'b101, 3'b110, or 3'b111 then **dma_done[C]** is never set HIGH.
9	When all transfers for a channel are complete, and the cycle_ctrl[a] bits enable the assertion of **dma_done[]**, then at the falling edge of **dma_active[]**: • if **dma_stall** is LOW, the controller sets **dma_done[]** HIGH for a duration of one **hclk** cycle • if **dma_stall** is HIGH, it stalls the controller. After **dma_stall** goes LOW then the controller sets **dma_done[]** HIGH for a duration of one **hclk** cycle.
10	The status of **dma_waitonreq[C]** must only change when channel C is disabled.
11	When **dma_waitonreq[C]** is HIGH then **dma_active[C]** does not go LOW until: • the controller completes 2^R or n_minus_1[a] transfers • **dma_req[C]** is LOW • **dma_sreq[C]** is LOW.
12	If, on the **hclk** cycle immediately prior to it setting **dma_active[C]** LOW, you set **dma_stall** HIGH, then: • the controller sets **dma_active[C]** LOW on the next **hclk** cycle • channel C does not complete until you set **dma_stall** LOW.
13	The controller ignores **dma_sreq[C]** when **dma_waitonreq[C]** is LOW.

Table 2-5 Rules when channels and enabled and requests are not masked (continued)

Rule	Description
14	The controller ignores **dma_sreq[C]** when the chnl_useburst_set [C] bit is a 1[b].
15	For DMA cycle types other than peripheral scatter-gather, on completion of 2^R transfers, the controller sets the read value of the chnl_useburst_set [C] bit to 0, if the number of remaining transfers is less than 2^R. In peripheral scatter-gather mode, the controller only sets the read value of the chnl_useburst_set [C] bit to 0, if the number of transfers remaining in the alternate data structure is less than 2^R.
16	For DMA cycle types other than peripheral scatter-gather, when **dma_sreq[C]** and **dma_waitonreq[C]** are HIGH, and **dma_req[C]** is LOW, on the **hclk** cycle before **dma_active[C]** goes HIGH, then the controller performs a single DMA transfer. In peripheral scatter-gather mode, when **dma_sreq[C]** and **dma_waitonreq[C]** are HIGH, and **dma_req[C]** is LOW, on the **hclk** cycle before **dma_active[C]** goes HIGH, then the controller performs 2^R transfers using the primary data structure, and then without arbitrating, it performs a single DMA transfer using the alternate data structure.
17	For DMA cycle types other than peripheral scatter-gather, when **dma_req[C]** and **dma_sreq[C]** are HIGH, on the **hclk** cycle before **dma_active[C]** goes HIGH, then the **dma_req[C]** request takes precedence and the controller performs 2^R, or n_minus_1[a], transfers. In peripheral scatter-gather mode, when **dma_req[C]** and **dma_sreq[C]** are HIGH, on the **hclk** cycle before **dma_active[C]** goes HIGH, then the **dma_req[C]** request takes precedence and the controller performs 2^R transfers using the primary data structure, and then without arbitrating, it performs the lesser number of transfers, that either 2^R or n_minus_1[a] specifies, using the alternate data structure.
18	When the chnl_req_mask_set [C] bit is a 1, the controller ignores requests on **dma_req[C]** and **dma_sreq[C]**.

a. The channel_cfg memory location contains these bits. See Table 2-16 on page 2-42.

b. You must take care in setting this bit. When n_minus_1 is less than 2^R then the controller does not clear the chnl_useburst_set and therefore requests on **dma_sreq[C]** are masked. If the peripheral does not set **dma_req[C]** HIGH then the controller never performs the outstanding transfers. See *Channel useburst set* on page 3-12.

When you disable a channel, the controller uses the DMA handshake rules that Table 2-6 lists.

Table 2-6 Rules for disabled channels

Rule	Description
19	When **dma_req[C]** is HIGH, the controller sets **dma_done[C]** HIGH. This enables the controller to alert the host processor to a request, even when the channel is disabled.
20	When **dma_sreq[C]** is HIGH, the controller sets **dma_done[C]** HIGH provided that **dma_waitonreq[C]** is HIGH and the chnl_useburst_set [C] bit is a 0. This enables the controller to alert the host controller to a request, when the channel is disabled.
21	**dma_active[C]** is always LOW.

DMA signaling

The following sections provide examples of the functional operation of the controller using the rules that *Handshake rules* on page 2-8 describe:

- *Pulse request*
- *Level request* on page 2-13
- *Done signaling* on page 2-14
- *Wait on request signaling* on page 2-16.

Note

For all DMA signaling shown in Figure 2-6 on page 2-12 to Figure 2-10 on page 2-17 inclusive, the following conditions apply:

- **hready** is HIGH
- the AHB slave always provides an OKAY response.

Pulse request

Figure 2-6 on page 2-12 shows the DMA request timing when a peripheral uses pulse signaling.

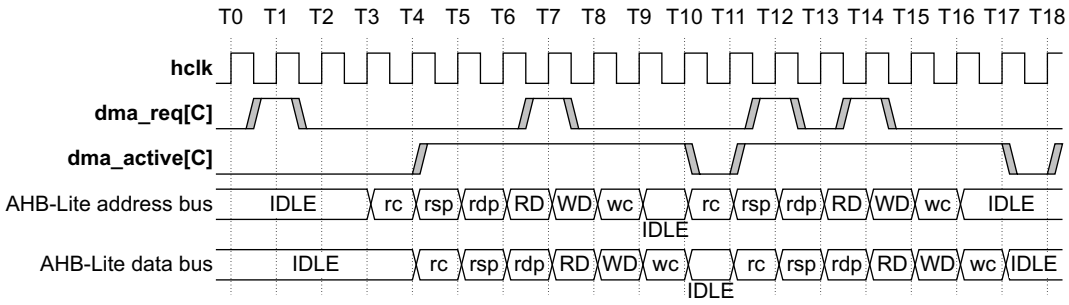


Figure 2-6 DMA signaling when peripherals use pulse requests

In Figure 2-6:

T1 The controller detects a request on channel C (see rule 1) provided that chnl_req_mask_set [C] is 0 (see rule 18).

T4 The controller asserts **dma_active[C]** (see rule 2 and rule 3) and starts the DMA transfer for channel C.

T4-T7 The controller reads the data structure, where:

- **rc** Reads channel configuration, channel_cfg.
- **rsp** Reads source data end pointer, src_data_end_ptr.
- **rdp** Reads destination data end pointer, dst_data_end_ptr.

T7 With **dma_active[C]** HIGH and provided that chnl_req_mask_set [C] is 0 (see rule 18), the controller detects a request on channel C that was not present on the previous clock cycle (see rule 7). The controller includes this request during the next arbitration process.

T7-T9 The controller performs the DMA transfer for channel C, where:

- **RD** Reads data.
- **WD** Writes data.

T9-T10 The controller writes the channel_cfg, where:

- **wc** Writes channel configuration, channel_cfg.

T10 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

T10-T11 The controller holds **dma_active[C]** LOW for at least one **hclk** cycle (see rule 5).

T11 If channel C is the highest priority request then the controller asserts **dma_active[C]** because of the request at T7 (see rule 2 and rule 3).

T12 With **dma_active[C]** HIGH and provided that chnl_req_mask_set [C] is 0 (see rule 18), the controller detects a request on channel C that was not present on the previous clock cycle (see rule 7). The controller includes this request during the next arbitration process.

T14 The controller ignores the request on channel C because of the pending request at T12.

T17 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

T17-T18 The controller holds **dma_active[C]** LOW for at least one **hclk** cycle (see rule 5).

T18 If channel C is the highest priority request then the controller asserts **dma_active[C]** because of the request at T12 (see rule 2 and rule 3).

Level request

Figure 2-7 shows the DMA request timing when a peripheral uses level signaling.

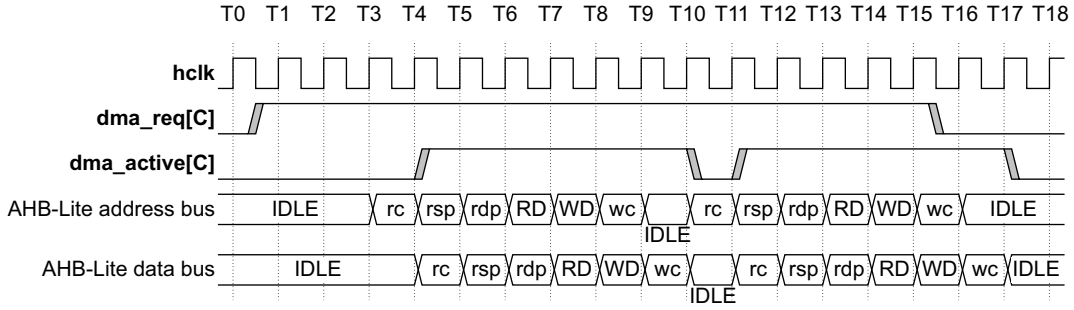


Figure 2-7 DMA signaling when peripherals use level requests

In Figure 2-7:

T1 The controller detects a request on channel C (see rule 1) provided that chnl_req_mask_set [C] is 0 (see rule 18).

T4 The controller asserts **dma_active[C]** (see rule 2 and rule 3) and starts the DMA transfer for channel C.

T4-T7 The controller reads the data structure, where:

- **rc** Reads channel configuration, channel_cfg.
- **rsp** Reads source data end pointer, src_data_end_ptr.
- **rdp** Reads destination data end pointer, dst_data_end_ptr.

T7-T9 The controller performs the DMA transfer for channel C, where:

- **RD** Reads data.
- **WD** Writes data.

T9-T10 The controller writes the channel_cfg, where:

- **wc** Writes channel configuration, channel_cfg.

T10 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

The controller detects a request on channel C (see rule 1) provided that chnl_req_mask_set [C] is 0 (see rule 18).

T10-T11 The controller holds **dma_active[C]** LOW for at least one **hclk** cycle (see rule 5).

T11 If channel C is the highest priority request then the controller asserts **dma_active[C]** and starts the second DMA transfer for channel C.

T11-T14 The controller reads the data structure.

T14-T16 The controller performs the DMA transfer for channel C.

T15-T16 The peripheral acknowledges that the transfer has started and deasserts **dma_req[C]**.

T16-T17 The controller writes the channel_cfg.

T17 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

When using level request signaling, if a peripheral does not require additional DMA transfers but is too slow to remove the request, then it must assert **dma_stall**. Asserting **dma_stall** prevents the controller from completing the current transfer (see rule 7).

Done signaling

Figure 2-8 on page 2-15 shows the **dma_done[ ]** signaling under the following different conditions:

- **dma_stall** and **dma_waitonreq[ ]** are LOW
- **dma_stall** is HIGH
- **dma_waitonreq[ ]** is HIGH.

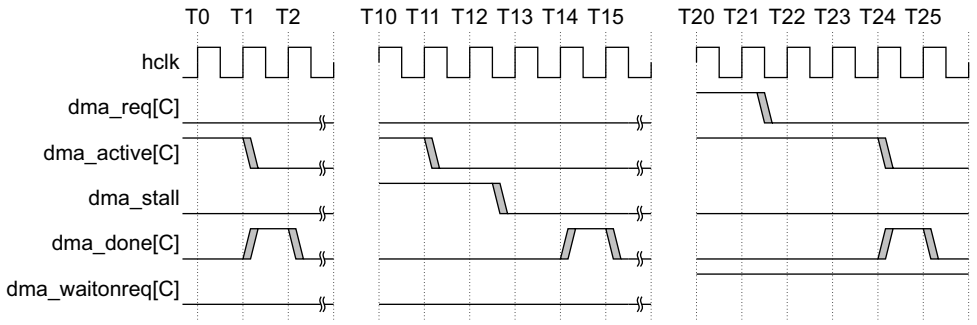


Figure 2-8 dma_done signaling

In Figure 2-8, from T0 to T2:

T1 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

T1-T2 The controller completes the DMA cycle and if the cycle_ctrl [2] bit is 0 then it asserts **dma_done[C]** for one **hclk** cycle (see rule 8 and rule 9). For all other enabled channels, **dma_done[ ]** is LOW (see rule 6).

In Figure 2-8, from T10 to T15:

T11 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

Note

The controller does not assert **dma_done[C]** because **dma_stall** was HIGH on the previous **hclk** cycle (see rule 9 and rule 12).

T12-T13 The peripheral deasserts **dma_stall**.

T14-T15 The controller completes the DMA cycle and if the cycle_ctrl [2] bit is 0 then it asserts **dma_done[C]** for one **hclk** cycle (see rule 8 and rule 9). For all other enabled channels, **dma_done[ ]** is LOW (see rule 6).

In Figure 2-8, from T20 to T25:

T20 The controller has performed the DMA transfers but because **dma_waitonreq[C]** is HIGH, it must wait for **dma_req[C]** to go LOW before it can deassert **dma_active[C]** (see rule 11) and assert **dma_done[C]** (see rule 9).

T21-T22 The peripheral deasserts **dma_req[C]**.

T24 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 4).

T24-T25 The controller completes the DMA cycle and if the cycle_ctrl [2] bit is 0 then it asserts **dma_done[C]** for one **hclk** cycle (see rule 8 and rule 9). For all other enabled channels, **dma_done[ ]** is LOW (see rule 6).

Wait on request signaling

The following figures show examples of using wait on request signaling to perform a 2^R transfer and a single transfer:

- *DMA signaling when peripherals use dma_waitonreq*
- *DMA signaling when peripherals use dma_waitonreq and dma_sreq* on page 2-17.

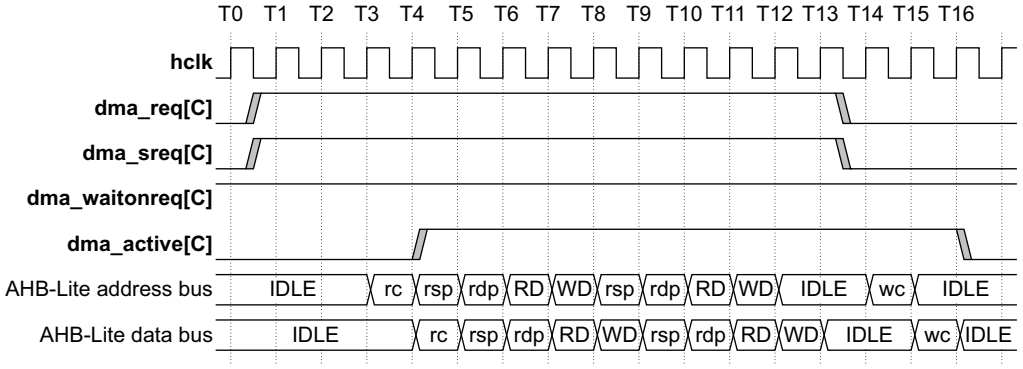


Figure 2-9 DMA signaling when peripherals use dma_waitonreq

In Figure 2-9:

T0-T16 The peripheral must hold the status of **dma_waitonreq[C]** constant (see rule 10).

T0-T1 The controller detects a request on channel C (see rule 1) provided that chnl_req_mask_set [C] is 0 (see rule 18).

T3-T4 The peripheral holds **dma_req[ ]** and **dma_sreq[ ]** HIGH. The controller ignores the **dma_sreq[ ]** request and responds to the **dma_req[ ]** request (see rule 16 and rule 17).

T4 The controller asserts **dma_active[C]** (see rule 2 and rule 3) and starts the DMA transfer for channel C.

T4-T7 The controller reads the data structure, where:

- **rc** Reads channel configuration, channel_cfg.
- **rsp** Reads source data end pointer, src_data_end_ptr.
- **rdp** Reads destination data end pointer, dst_data_end_ptr.

T7-T9 The controller performs a DMA transfer for channel C, where:

- **RD** Reads data.
- **WD** Writes data.

T9-T11 The controller reads the two end pointer addresses, rsp and rdp.

T11-T13 The controller performs a DMA transfer for channel C. In this example, R=1 and hence the controller performs $2^1=2$ DMA transfers.

T13-T14 The peripheral deasserts **dma_req[C]** and **dma_sreq[C]**.

T15-T16 The controller writes the channel_cfg, where:

- **wc** Writes channel configuration, channel_cfg.

T16 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 11).

The controller sets the read value of the chnl_useburst_set [C] bit to 0, if the number of remaining transfers is less than 2^R (see rule 15).

Figure 2-10 shows the DMA signaling when **dma_waitonreq[]** is HIGH and the controller performs a single DMA transfer.

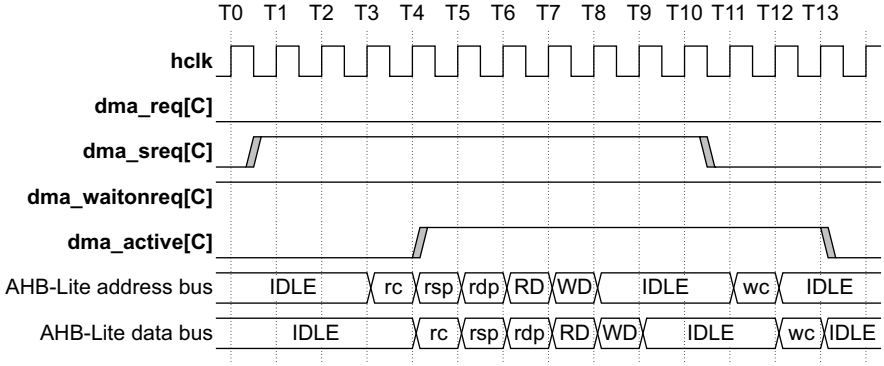


Figure 2-10 DMA signaling when peripherals use dma_waitonreq and dma_sreq

In Figure 2-10 on page 2-17:

T0-T13 The peripheral must hold the status of **dma_waitonreq[C]** constant (see rule 10).

T0-T1 The controller detects a request on channel C (see rule 1) provided that chnl_useburst_set [C] is 0 (see rule 13 and rule 14).

T3-T4 The controller responds to the **dma_sreq[ ]** request (see rule 16).

T4 The controller asserts **dma_active[C]** (see rule 2 and rule 3) and starts the DMA transfer for channel C.

T4-T7 The controller reads the data structure, where:

- **rc** Reads channel configuration, channel_cfg.
- **rsp** Reads source data end pointer, src_data_end_ptr.
- **rdp** Reads destination data end pointer, dst_data_end_ptr.

T7-T9 The controller performs a DMA transfer for channel C, where:

- **RD** Reads data.
- **WD** Writes data.

This is a request in response to **dma_sreq[ ]** so R=0 and hence the controller performs $2^0=1$ DMA transfer.

T10-T11 The peripheral deasserts **dma_sreq[C]**.

T12-T13 The controller writes the channel_cfg, where:

- **wc** Writes channel configuration, channel_cfg.

T13 The controller deasserts **dma_active[C]** to indicate that the DMA transfer has completed (see rule 11).

DMA arbitration rate

You can configure when the controller arbitrates during a DMA transfer. This enables you to reduce the latency to service a higher priority channel.

The controller provides four bits that configure how many AHB bus transfers occur before it rearbitrates. These bits are known as the R_power bits because the value you enter, *R*, is raised to the power of two and this determines the arbitration rate. For example, if R=4 then the arbitration rate is 2^4, that is, the controller arbitrates every 16 DMA transfers.

Table 2-7 lists the arbitration rates.

Table 2-7 AHB bus transfer arbitration interval

R_power	Arbitrate after *x* DMA transfers
b0000	x=1
b0001	x=2
b0010	x=4
b0011	x=8
b0100	x=16
b0101	x=32
b0110	x=64
b0111	x=128
b1000	x=256
b1001	x=512
b1010-b1111	x=1024

Note

You must take care not to assign a low-priority channel with a large R_power because this prevents the controller from servicing high-priority requests, until it rearbitrates.

When $N > 2^R$ and is not an integer multiple of 2^R then the controller always performs sequences of 2^R transfers until $N < 2^R$ remain to be transferred. The controller performs the remaining N transfers at the end of the DMA cycle.

You store the value of the R_power bits in the channel control data structure. See *Control data configuration* on page 2-41 for more information about the location of the R_power bits in the data structure.

Priority

When the controller arbitrates, it determines the next channel to service by using the following information:

- the channel number
- the priority level, default or high, that is assigned to the channel.

You can configure each channel to use either the default priority level or a high priority level by setting the chnl_priority_set Register. See *Channel priority set* on page 3-23.

Channel number zero has the highest priority and as the channel number increases, the priority of a channel decreases. Table 2-8 lists the DMA channel priority levels in descending order of priority.

Table 2-8 DMA channel priority

Channel number	Priority level setting	Descending order of channel priority
0	High	Highest-priority DMA channel
1	High	-
2	High	-
-	High	-
-	High	-
-	High	-
30	High	-
31	High	-
0	Default	-
1	Default	-
2	Default	-
-	Default	-
-	Default	-
-	Default	-
30	Default	-
31	Default	Lowest-priority DMA channel

After a DMA transfer completes, the controller polls all the DMA channels that are available. Figure 2-11 on page 2-21 shows the process it uses to determine which DMA transfer to perform next.

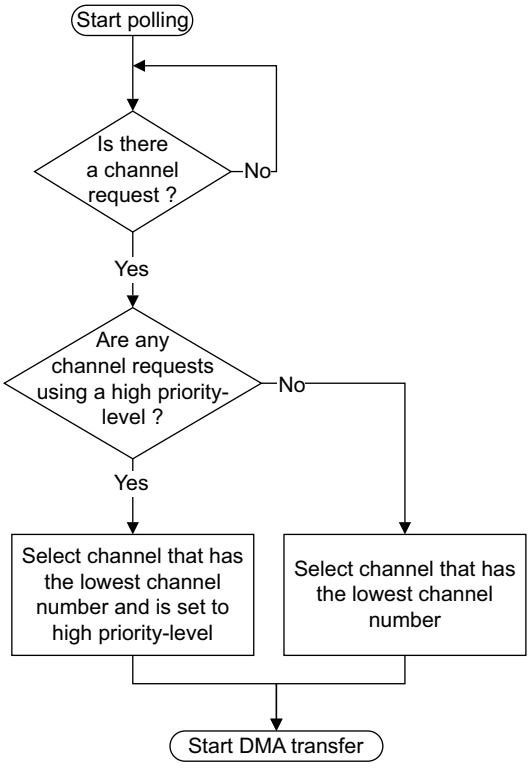


Figure 2-11 Polling flowchart

DMA cycle types

The cycle_ctrl bits control how the controller performs a DMA cycle. You can set the cycle_ctrl bits as Table 2-9 lists.

Table 2-9 DMA cycle types

cycle_ctrl	Description
b000	Channel control data structure is invalid
b001	Basic DMA transfer
b010	Auto-request
b011	Ping-pong
b100	Memory scatter-gather using the primary data structure

Table 2-9 DMA cycle types (continued)

cycle_ctrl	Description
b101	Memory scatter-gather using the alternate data structure
b110	Peripheral scatter-gather using the primary data structure
b111	Peripheral scatter-gather using the alternate data structure

Note

The cycle_ctrl bits are located in the channel_cfg memory location that *Control data configuration* on page 2-41 describes.

For all cycle types, the controller arbitrates after 2^R DMA transfers. If you set a low-priority channel with a large 2^R value then it prevents all other channels from performing a DMA transfer, until the low-priority DMA transfer completes. Therefore, you must take care when setting the R_power, that you do not significantly increase the latency for high-priority channels.

The following sections describe the cycle types:

- *Invalid*
- *Basic*
- *Auto-request* on page 2-23
- *Ping-pong* on page 2-23
- *Memory scatter-gather* on page 2-27
- *Peripheral scatter-gather* on page 2-31.

Invalid

After the controller completes a DMA cycle it sets the cycle type to invalid, to prevent it from repeating the same DMA cycle.

Basic

In this mode, you configure the controller to use either the primary, or alternate, data structure. After you enable the channel, and the controller receives a request then the flow for this DMA cycle is:

1. The controller performs 2^R transfers. If the number of transfers remaining is zero the flow continues at step 3.

2. The controller arbitrates:
   - if a higher-priority channel is requesting service then the controller services that channel
   - if the peripheral or software signals a request to the controller then it continues at step 1.
3. The controller sets **dma_done[C]** HIGH for one **hclk** cycle. This indicates to the host processor that the DMA cycle is complete.

Auto-request

When the controller operates in this mode, it is only necessary for it to receive a single request to enable it to complete the entire DMA cycle. This enables a large data transfer to occur, without significantly increasing the latency for servicing higher priority requests, or requiring multiple requests from the processor or peripheral.

You can configure the controller to use the primary, or alternate, data structure. After you enable the channel, and the controller receives a request for this channel, then the flow for this DMA cycle is:

1. The controller performs 2^R transfers for channel C. If the number of transfers remaining is zero the flow continues at step 3.
2. The controller arbitrates. When channel C has the highest priority then the DMA cycle continues at step 1.
3. The controller sets **dma_done[C]** HIGH for one **hclk** cycle. This indicates to the host processor that the DMA cycle is complete.

Ping-pong

In ping-pong mode, the controller performs a DMA cycle using one of the data structures and it then performs a DMA cycle using the other data structure. The controller continues to switch from primary to alternate to primary… until it reads a data structure that is invalid, or until the host processor disables the channel.

Figure 2-12 on page 2-24 shows an example of a ping-pong DMA transaction.

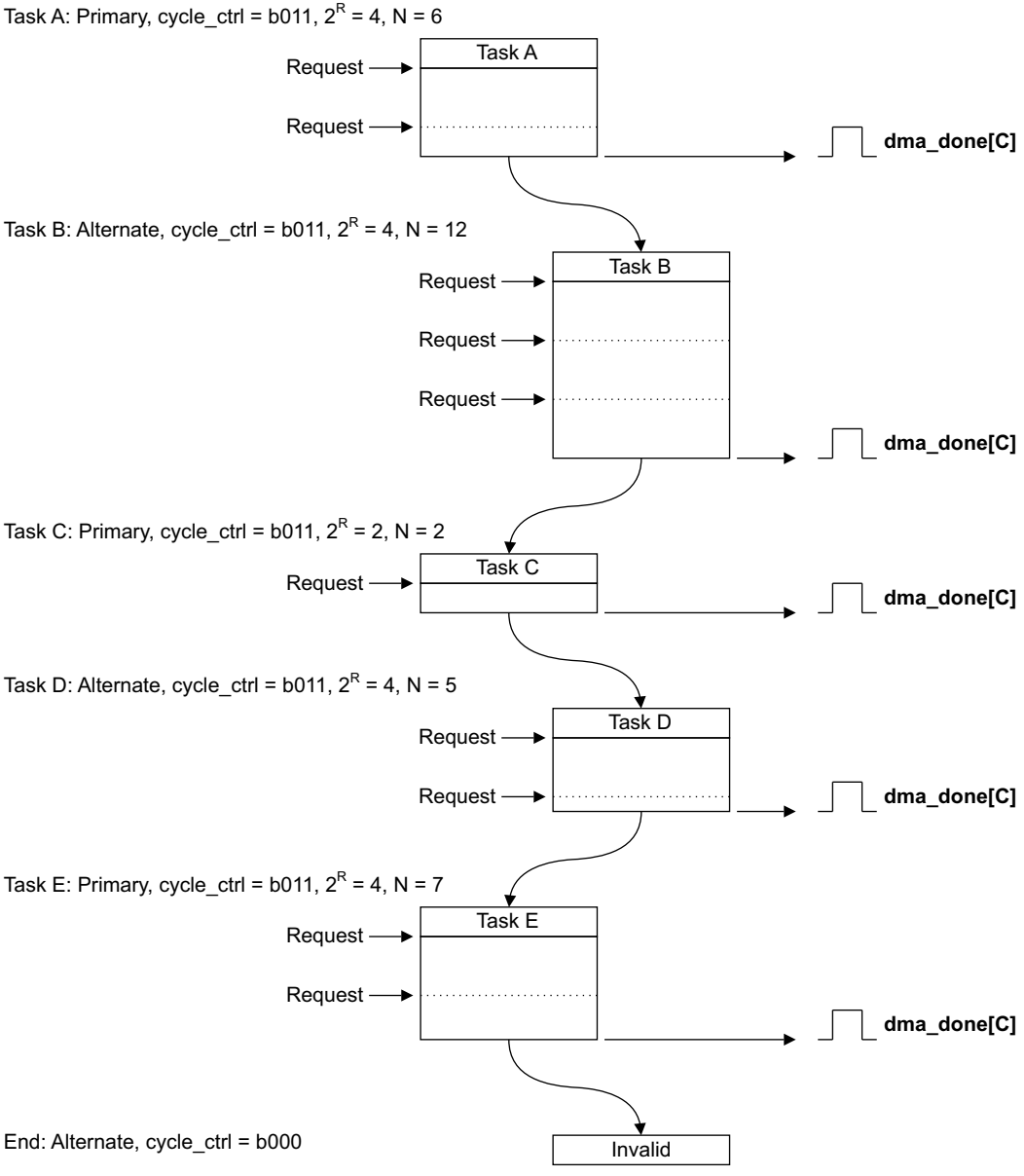


Figure 2-12 Ping-pong example

In Figure 2-12 on page 2-24:

Task A

1. The host processor configures the primary data structure for task A.
2. The host processor configures the alternate data structure for task B. This enables the controller to immediately switch to task B after task A completes, provided that a higher priority channel does not require servicing.
3. The controller receives a request and performs four DMA transfers.
4. The controller arbitrates. After the controller receives a request for this channel, the flow continues if the channel has the highest priority.
5. The controller performs the remaining two DMA transfers.
6. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

After task A completes, the host processor can configure the primary data structure for task C. This enables the controller to immediately switch to task C after task B completes, provided that a higher priority channel does not require servicing.

After the controller receives a new request for the channel and it has the highest priority then task B commences:

Task B

7. The controller performs four DMA transfers.
8. The controller arbitrates. After the controller receives a request for this channel, the flow continues if the channel has the highest priority.
9. The controller performs four DMA transfers.
10. The controller arbitrates. After the controller receives a request for this channel, the flow continues if the channel has the highest priority.
11. The controller performs the remaining four DMA transfers.
12. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

After task B completes, the host processor can configure the alternate data structure for task D.

After the controller receives a new request for the channel and it has the highest priority then task C commences:

Task C

13. The controller performs two DMA transfers.
14. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

After task C completes, the host processor can configure the primary data structure for task E.

After the controller receives a new request for the channel and it has the highest priority then task D commences:

Task D

15. The controller performs four DMA transfers.
16. The controller arbitrates. After the controller receives a request for this channel, the flow continues if the channel has the highest priority.
17. The controller performs the remaining DMA transfer.
18. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

After the controller receives a new request for the channel and it has the highest priority then task E commences:

Task E

19. The controller performs four DMA transfers.
20. The controller arbitrates. After the controller receives a request for this channel, the flow continues if the channel has the highest priority.
21. The controller performs the remaining three DMA transfers.
22. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

If the controller receives a new request for the channel and it has the highest priority then it attempts to start the next task. However, because the host processor has not configured the alternate data structure, and on completion of task D the controller set the cycle_ctrl bits to b000, then the ping-pong DMA transaction completes.

Note

You can also terminate the ping-pong DMA cycle in Figure 2-12 on page 2-24, if you configure task E to be a basic DMA cycle by setting the cycle_ctrl field to 3’b001.

Memory scatter-gather

In memory scatter-gather mode the controller receives an initial request and then performs four DMA transfers using the primary data structure. After this transfer completes, it starts a DMA cycle using the alternate data structure. After this cycle completes, the controller performs another four DMA transfers using the primary data structure. The controller continues to switch from primary to alternate to primary… until either:

- the host processor configures the alternate data structure for a basic cycle
- it reads an invalid data structure.

Note

After the controller completes the N primary transfers it invalidates the primary data structure by setting the cycle_ctrl field to b000.

The controller only asserts **dma_done[C]** when the scatter-gather transaction completes using a basic cycle.

In scatter-gather mode, the controller uses the primary data structure to program the alternate data structure. Table 2-10 lists the fields of the channel_cfg memory location for the primary data structure, that you must program with constant values and those that can be user defined.

Table 2-10 channel_cfg for a primary data structure, in memory scatter-gather mode

Bit	Field	Value	Description
Constant-value fields:			
[31:30}	dst_inc	b10	Configures the controller to use word increments for the address
[29:28]	dst_size	b10	Configures the controller to use word transfers
[27:26]	src_inc	b10	Configures the controller to use word increments for the address
[25:24]	src_size	b10	Configures the controller to use word transfers
[17:14]	R_power	b0010	Configures the controller to perform four DMA transfers
[3]	next_useburst	0	For a memory scatter-gather DMA cycle, this bit must be set to zero
[2:0]	cycle_ctrl	b100	Configures the controller to perform a memory scatter-gather DMA cycle
User defined values:			

Table 2-10 channel_cfg for a primary data structure, in memory scatter-gather mode (continued)

Bit	Field	Value	Description
[23:21]	dst_prot_ctrl	-	Configures the state of **HPROT** when the controller writes the destination data
[20:18]	src_prot_ctrl	-	Configures the state of **HPROT** when the controller reads the source data
[13:4]	n_minus_1	N[a]	Configures the controller to perform *N* DMA transfers, where *N* is a multiple of four

a. Because the R_power field is set to four, you must set *N* to be a multiple of four. The value given by *N*/4 is the number of times that you must configure the alternate data structure.

See *Control data configuration* on page 2-41 for more information.

Figure 2-13 on page 2-29 shows a memory scatter-gather example.

Initialization: 1. Configure primary to enable the copy A, B, C, and D operations: cycle_ctrl = b100, 2^R = 4, N = 16.
2. Write the primary source data to memory, using the structure shown in the following table.

	src_data_end_ptr	dst_data_end_ptr	channel_cfg	Unused
Data for Task A	0x0A000000	0x0AE00000	cycle_ctrl = b101, 2^R = 4, N = 3	0xXXXXXXXX
Data for Task B	0x0B000000	0x0BE00000	cycle_ctrl = b101, 2^R = 2, N = 8	0xXXXXXXXX
Data for Task C	0x0C000000	0x0CE00000	cycle_ctrl = b101, 2^R = 8, N = 5	0xXXXXXXXX
Data for Task D	0x0D000000	0x0DE00000	cycle_ctrl = b001, 2^R = 4, N = 4	0xXXXXXXXX

Memory scatter-gather transaction:

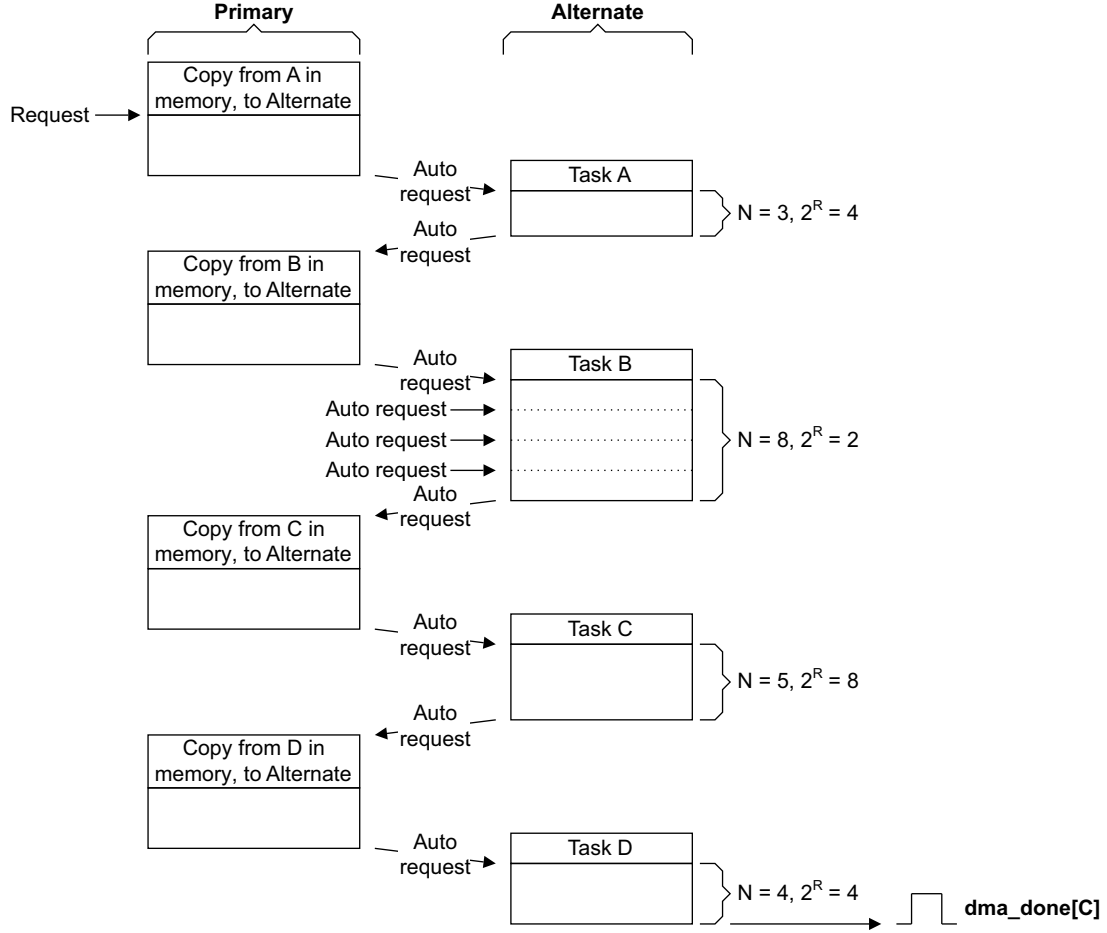


Figure 2-13 Memory scatter-gather example

In Figure 2-13 on page 2-29:

Initialization

1. The host processor configures the primary data structure to operate in memory scatter-gather mode by setting cycle_ctrl to b100. Because a data structure for a single channel consists of four words then you must set 2^R to 4. In this example, there are four tasks and therefore N is set to 16.
2. The host processor writes the data structure for tasks A, B, C, and D to the memory locations that the primary src_data_end_ptr specifies.
3. The host processor enables the channel.

The memory scatter-gather transaction commences when the controller receives a request on **dma_req[]** or a manual request from the host processor. The transaction continues as follows:

Primary, copy A

1. After receiving a request, the controller performs four DMA transfers. These transfers write the alternate data structure for task A.
2. The controller generates an auto-request for the channel and then arbitrates.

Task A

3. The controller performs task A. After it completes the task, it generates an auto-request for the channel and then arbitrates.

Primary, copy B

4. The controller performs four DMA transfers. These transfers write the alternate data structure for task B.
5. The controller generates an auto-request for the channel and then arbitrates.

Task B

6. The controller performs task B. After it completes the task, it generates an auto-request for the channel and then arbitrates.

Primary, copy C

7. The controller performs four DMA transfers. These transfers write the alternate data structure for task C.
8. The controller generates an auto-request for the channel and then arbitrates.

Task C

9. The controller performs task C. After it completes the task, it generates an auto-request for the channel and then arbitrates.

Primary, copy D

10. The controller performs four DMA transfers. These transfers write the alternate data structure for task D.
11. The controller sets the cycle_ctrl bits of the primary data structure to b000, to indicate that this data structure is now invalid.
12. The controller generates an auto-request for the channel and then arbitrates.

Task D

13. The controller performs task D using a basic cycle.
14. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

Peripheral scatter-gather

In peripheral scatter-gather mode the controller receives an initial request from a peripheral and then it performs four DMA transfers using the primary data structure. It then immediately starts a DMA cycle using the alternate data structure, without rearbitrating or **dma_active[C]** going LOW.

Note

These are the only circumstances, where the controller does not enter the arbitration process after completing a transfer using the primary data structure.

After this cycle completes, the controller rearbitrates and if the controller receives a request from the peripheral that has the highest priority then it performs another four DMA transfers using the primary data structure. It then immediately starts a DMA cycle using the alternate data structure, without re-arbitrating or **dma_active[C]** going LOW. The controller continues to switch from primary to alternate to primary… until either:

- the host processor configures the alternate data structure for a basic cycle
- it reads an invalid data structure.

Note

After the controller completes the N primary transfers it invalidates the primary data structure by setting the cycle_ctrl field to b000.

The controller asserts **dma_done[C]** when the scatter-gather transaction completes using a basic cycle.

In scatter-gather mode, the controller uses the primary data structure to program the alternate data structure. Table 2-11 lists the fields of the channel_cfg memory location for the primary data structure, that you must program with constant values and those that can be user defined.

Table 2-11 channel_cfg for a primary data structure, in peripheral scatter-gather mode

Bit	Field	Value	Description
Constant-value fields:			
[31:30}	dst_inc	b10	Configures the controller to use word increments for the address
[29:28]	dst_size	b10	Configures the controller to use word transfers
[27:26]	src_inc	b10	Configures the controller to use word increments for the address
[25:24]	src_size	b10	Configures the controller to use word transfers
[17:14]	R_power	b0010	Configures the controller to perform four DMA transfers
[2:0]	cycle_ctrl	b110	Configures the controller to perform a peripheral scatter-gather DMA cycle
User defined values:			
[23:21]	dst_prot_ctrl	-	Configures the state of **HPROT** when the controller writes the destination data
[20:18]	src_prot_ctrl	-	Configures the state of **HPROT** when the controller reads the source data
[13:4]	n_minus_1	N[a]	Configures the controller to perform *N* DMA transfers, where *N* is a multiple of four
[3]	next_useburst	-	When set to 1, the controller sets the chnl_useburst_set [C] bit to 1 after the alternate transfer completes

a. Because the R_power field is set to four, you must set *N* to be a multiple of four. The value given by *N*/4 is the number of times that you must configure the alternate data structure.

See *Control data configuration* on page 2-41 for more information.

Figure 2-14 on page 2-33 shows a peripheral scatter-gather example.

Initialization: 1. Configure primary to enable the copy A, B, C, and D operations: cycle_ctrl = b110, 2^R = 4, N = 16.
2. Write the primary source data in memory, using the structure shown in the following table.

	src_data_end_ptr	dst_data_end_ptr	channel_cfg	Unused
Data for Task A	0x0A000000	0x0AE00000	cycle_ctrl = b111, 2^R = 4, N = 3	0xXXXXXXXX
Data for Task B	0x0B000000	0x0BE00000	cycle_ctrl = b111, 2^R = 2, N = 8	0xXXXXXXXX
Data for Task C	0x0C000000	0x0CE00000	cycle_ctrl = b111, 2^R = 8, N = 5	0xXXXXXXXX
Data for Task D	0x0D000000	0x0DE00000	cycle_ctrl = b001, 2^R = 4, N = 4	0xXXXXXXXX

Peripheral scatter-gather transaction:

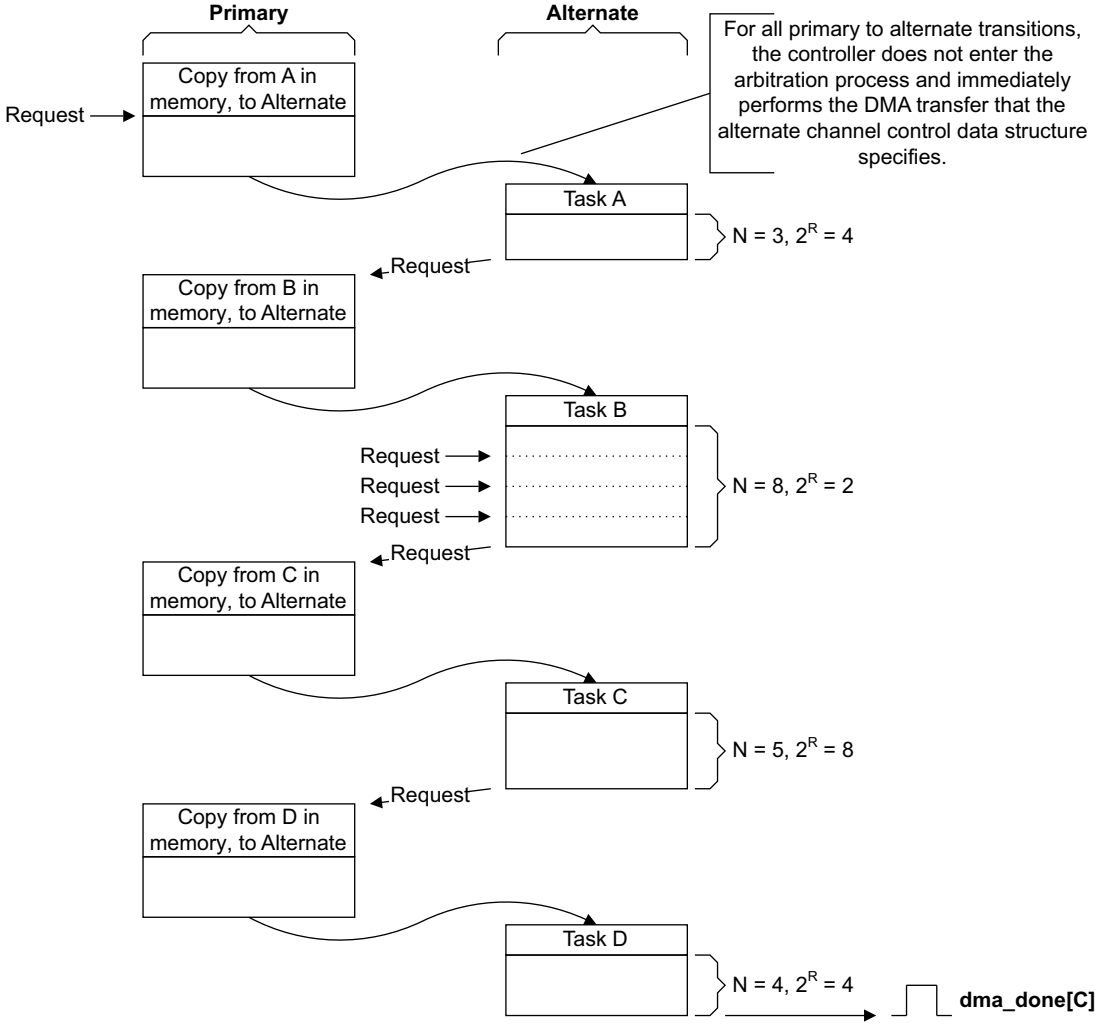


Figure 2-14 Peripheral scatter-gather example

In Figure 2-14 on page 2-33:

Initialization

1. The host processor configures the primary data structure to operate in peripheral scatter-gather mode by setting cycle_ctrl to b110. Because a data structure for a single channel consists of four words then you must set 2^R to 4. In this example, there are four tasks and therefore N is set to 16.
2. The host processor writes the data structure for tasks A, B, C, and D to the memory locations that the primary src_data_end_ptr specifies.
3. The host processor enables the channel.

The peripheral scatter-gather transaction commences when the controller receives a request on **dma_req[]**. The transaction continues as follows:

Primary, copy A

1. After receiving a request, the controller performs four DMA transfers. These transfers write the alternate data structure for task A.

Task A

2. The controller performs task A.
3. After the controller completes the task it enters the arbitration process.

After the peripheral issues a new request and it has the highest priority then the process continues with:

Primary, copy B

4. The controller performs four DMA transfers. These transfers write the alternate data structure for task B.

Task B

5. The controller performs task B. To enable the controller to complete the task, the peripheral must issue a further three requests.
6. After the controller completes the task it enters the arbitration process.

After the peripheral issues a new request and it has the highest priority then the process continues with:

Primary, copy C

7. The controller performs four DMA transfers. These transfers write the alternate data structure for task C.

Task C

8. The controller performs task C.
9. After the controller completes the task it enters the arbitration process.

After the peripheral issues a new request and it has the highest priority then the process continues with:

Primary, copy D

10. The controller performs four DMA transfers. These transfers write the alternate data structure for task D.
11. The controller sets the cycle_ctrl bits of the primary data structure to b000, to indicate that this data structure is now invalid.

Task D

12. The controller performs task D using a basic cycle.
13. The controller sets **dma_done[C]** HIGH for one **hclk** cycle and enters the arbitration process.

Error signaling

If the controller detects an ERROR response on the AHB-Lite master interface, it:

- disables the channel that corresponds to the ERROR
- sets **dma_err** HIGH.

After the host processor detects that **dma_err** is HIGH, it must check which channel was active when the ERROR occurred. It can do this by:

1. Reading the chnl_enable_set Register to create a list of disabled channels.

When a channel asserts **dma_done[ ]** then the controller disables the channel. The program running on the host processor must always keep a record of which channels have recently asserted their **dma_done[ ]** outputs.

2. It must compare the disabled channels list from step 1, with the record of the channels that have recently set their **dma_done[ ]** outputs. The channel with no record of **dma_done[C]** being set is the channel that the ERROR occurred on.

2.2.4 Channel control data structure

You must provide an area of system memory to contain the channel control data structure. This system memory must:

- provide a contiguous area of system memory that the controller and host processor can access

- have a base address that is an integer multiple of the total size of the channel control data structure.

Figure 2-15 shows the memory that the controller requires for the channel control data structure, when it uses all 32 channels and the optional alternate data structure.

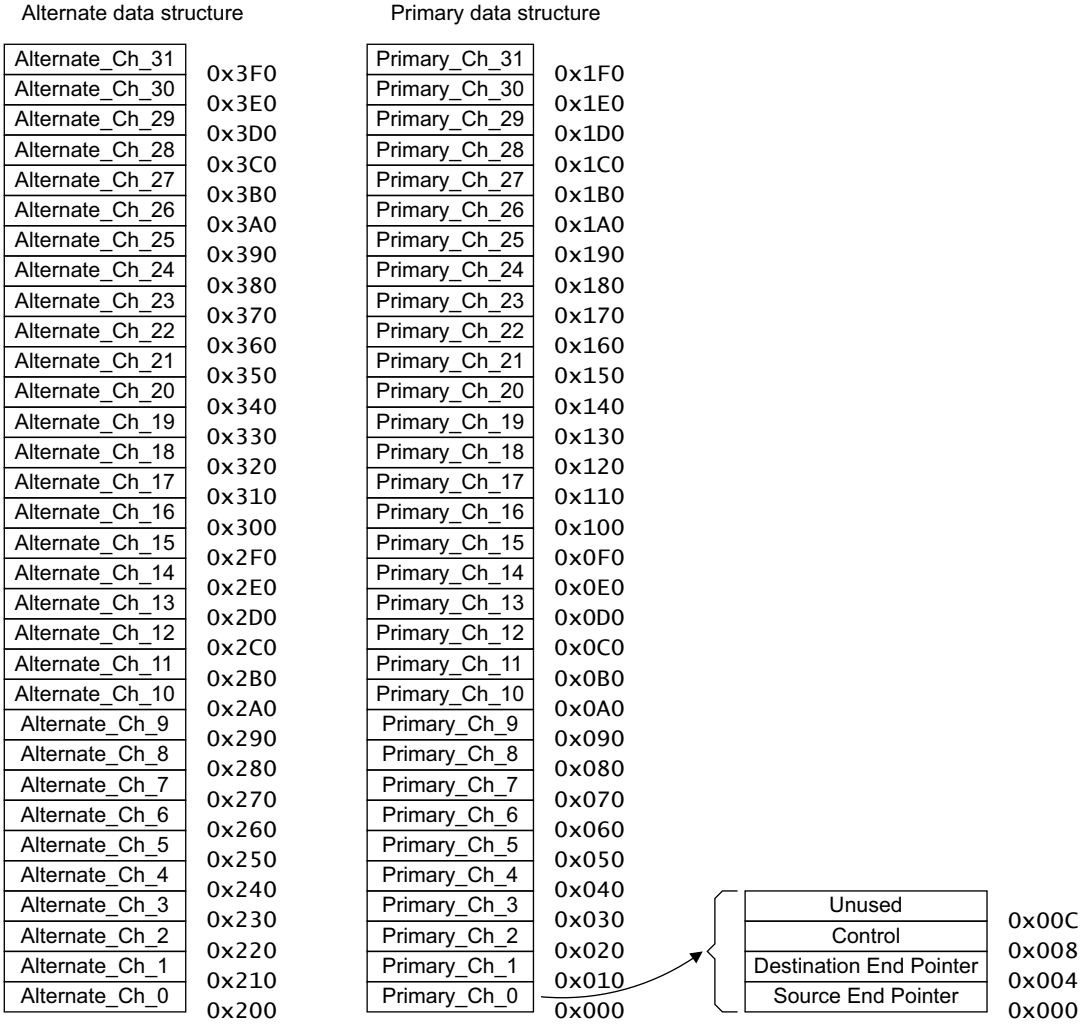


Figure 2-15 Memory map for 32 channels, including the alternate data structure

The example structure in Figure 2-15 on page 2-36 uses 1KB of system memory. In this example, the controller uses the lower 10 address bits to enable it to access all of the elements in the structure and therefore the base address must be at `0xXXXXX000`, `0xXXXXX400`, `0xXXXXX800`, or `0xXXXXXC00`.

You can configure the base address for the primary data structure by writing the appropriate value in the ctrl_base_ptr Register. See *Channel control data base pointer* on page 3-8.

The amount of system memory you require depends on:

- the number of DMA channels you configure the controller to use
- if you configure a DMA channel to use the alternate data structure. See *Channel primary-alternate set* on page 3-19.

Table 2-12 lists the address bits that the controller uses when it accesses the elements of the channel control data structure, depending on the number of channels that the controller contains.

Table 2-12 Address bit settings for the channel control data structure

	Address bits						
Number of DMA channels implemented	**[9]**	**[8]**	**[7]**	**[6]**	**[5]**	**[4]**	**[3:0]**
1						A	`0x0`, `0x4`, or `0x8`
2					A	C[0]	
3-4				A	C[1]	C[0]	
5-8			A	C[2]	C[1]	C[0]	
9-16		A	C[3]	C[2]	C[1]	C[0]	
17-32	A	C[4]	C[3]	C[2]	C[1]	C[0]	

Where:

A Selects one of the channel control data structures:

- **A = 0** Selects the primary data structure.
- **A = 1** Selects the alternate data structure.

C[*x*:0] Selects the DMA channel.

Address[3:0] Selects one of the control elements:

- `0x0` Selects the source data end pointer.
- `0x4` Selects the destination data end pointer.
- `0x8` Selects the control data configuration.
- `0xC` The controller does not access this address location. If required, you can enable the host processor to use this memory location as system memory.

Note

It is not necessary for you to calculate the base address of the alternate data structure because the alt_ctrl_base_ptr Register provides this information. See *Channel alternate control data base pointer* on page 3-9.

Figure 2-16 on page 2-39 shows an example implementation where the controller uses three DMA channels and the alternate data structure.

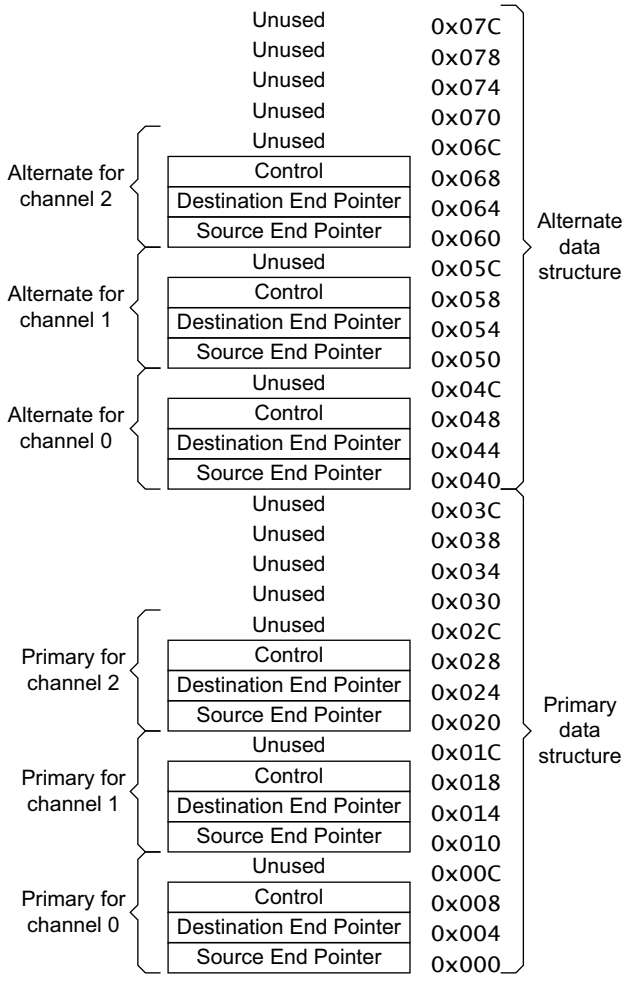


Figure 2-16 Memory map for three DMA channels, including the alternate data structure

The example structure in Figure 2-16 uses 128 bytes of system memory. In this example, the controller uses the lower six address bits to enable it to access all of the elements in the structure and therefore the base address must be at `0xXXXXXX00` or `0xXXXXXX80`.

Table 2-13 on page 2-41 lists the permitted base address values that you can assign for the primary data structure, depending on the number of channels that the controller contains.

Table 2-13 Permitted base addresses

Number of DMA channels	Permitted base addresses[a] for the primary data structure
1	0xXXXXXX00, 0xXXXXXX20, 0xXXXXXX40, 0xXXXXXX60, 0xXXXXXX80, 0xXXXXXXA0, 0xXXXXXXC0, 0xXXXXXXE0
2	0xXXXXXX00, 0xXXXXXX40, 0xXXXXXX80, 0xXXXXXXC0
3-4	0xXXXXXX00, 0xXXXXXX80
5-8	0xXXXXX000, 0xXXXXX100, 0xXXXXX200, 0xXXXXX300, 0xXXXXX400, 0xXXXXX500, 0xXXXXX600, 0xXXXXX700, 0xXXXXX800, 0xXXXXX900, 0xXXXXXA00, 0xXXXXXB00, 0xXXXXXC00, 0xXXXXXD00, 0xXXXXXE00, 0xXXXXXF00
9-16	0xXXXXX000, 0xXXXXX200, 0xXXXXX400, 0xXXXXX600, 0xXXXXX800, 0xXXXXXA00, 0xXXXXXC00, 0xXXXXXE00
17-32	0xXXXXX000, 0xXXXXX400, 0xXXXXX800, 0xXXXXXC00

a. Where X is a hexadecimal.

The controller uses the system memory to enable it to access two pointers and the control information that it requires for each channel. The following subsections describe these 32-bit memory locations and how the controller calculates the DMA transfer address:

- *Source data end pointer*
- *Destination data end pointer* on page 2-41
- *Control data configuration* on page 2-41
- *Address calculation* on page 2-47.

Source data end pointer

The src_data_end_ptr memory location contains a pointer to the end address of the source data. Table 2-14 lists the bit assignments for this memory location.

Table 2-14 src_data_end_ptr bit assignments

Bit	Name	Description
[31:0]	src_data_end_ptr	Pointer to the end address of the source data

Before the controller can perform a DMA transfer, you must program this memory location with the end address of the source data. The controller reads this memory location when it starts a 2^R DMA transfer.

Note

The controller does not write to this memory location.

Destination data end pointer

The dst_data_end_ptr memory location contains a pointer to the end address of the destination data. Table 2-15 lists the bit assignments for this memory location.

Table 2-15 dst_data_end_ptr bit assignments

Bit	Name	Description
[31:0]	dst_data_end_ptr	Pointer to the end address of the destination data

Before the controller can perform a DMA transfer, you must program this memory location with the end address of the destination data. The controller reads this memory location when it starts a 2^R DMA transfer.

Note

The controller does not write to this memory location.

Control data configuration

For each DMA transfer, the channel_cfg memory location provides the control information for the controller. Figure 2-17 shows the bit assignments for this memory location.

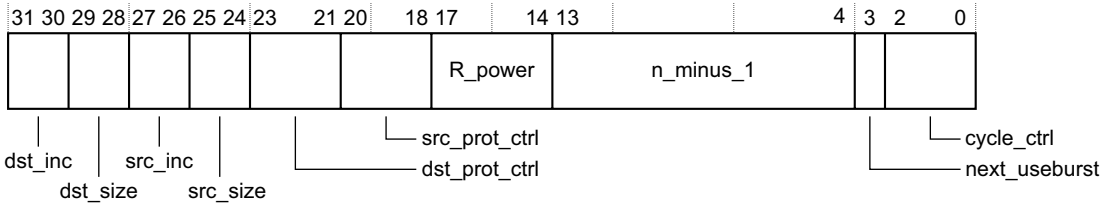


Figure 2-17 channel_cfg bit assignments

Table 2-16 lists the bit assignments for this memory location.

Table 2-16 channel_cfg bit assignments

Bit	Name	Description
[31:30]	dst_inc	Destination address increment. The address increment depends on the source data width as follows: **Source data width = byte** b00 = byte. b01 = halfword. b10 = word. b11 = no increment. Address remains set to the value that the dst_data_end_ptr memory location contains. **Source data width = halfword** b00 = reserved. b01 = halfword. b10 = word. b11 = no increment. Address remains set to the value that the dst_data_end_ptr memory location contains. **Source data width = word** b00 = reserved. b01 = reserved. b10 = word. b11 = no increment. Address remains set to the value that the dst_data_end_ptr memory location contains.
[29:28]	dst_size	Destination data size. **Note** You must set dst_size to contain the same value that src_size contains.

Table 2-16 channel_cfg bit assignments (continued)

Bit	Name	Description
[27:26]	src_inc	Set the bits to control the source address increment. The address increment depends on the source data width as follows: **Source data width = byte** b00 = byte. b01 = halfword. b10 = word. b11 = no increment. Address remains set to the value that the src_data_end_ptr memory location contains. **Source data width = halfword** b00 = reserved. b01 = halfword. b10 = word. b11 = no increment. Address remains set to the value that the src_data_end_ptr memory location contains. **Source data width = word** b00 = reserved. b01 = reserved. b10 = word. b11 = no increment. Address remains set to the value that the src_data_end_ptr memory location contains.
[25:24]	src_size	Set the bits to match the size of the source data: b00 = byte b01 = halfword b10 = word b11 = reserved.
[23:21]	dst_prot_ctrl	Set the bits to control the state of **HPROT[3:1]** when the controller writes the destination data. **Bit [23]** Controls the state of **HPROT[3]** as follows: 0 = **HPROT[3]** is LOW and the access is non-cacheable. 1 = **HPROT[3]** is HIGH and the access is cacheable. **Bit [22]** Controls the state of **HPROT[2]** as follows: 0 = **HPROT[2]** is LOW and the access is non-bufferable. 1 = **HPROT[2]** is HIGH and the access is bufferable. **Bit [21]** Controls the state of **HPROT[1]** as follows: 0 = **HPROT[1]** is LOW and the access is non-privileged. 1 = **HPROT[1]** is HIGH and the access is privileged.

Table 2-16 channel_cfg bit assignments (continued)

Bit	Name	Description
[20:18]	src_prot_ctrl	Set the bits to control the state of **HPROT[3:1]** when the controller reads the source data. **Bit [20]** Controls the state of **HPROT[3]** as follows: 0 = **HPROT[3]** is LOW and the access is non-cacheable. 1 = **HPROT[3]** is HIGH and the access is cacheable. **Bit [19]** Controls the state of **HPROT[2]** as follows: 0 = **HPROT[2]** is LOW and the access is non-bufferable. 1 = **HPROT[2]** is HIGH and the access is bufferable. **Bit [18]** Controls the state of **HPROT[1]** as follows: 0 = **HPROT[1]** is LOW and the access is non-privileged. 1 = **HPROT[1]** is HIGH and the access is privileged.
[17:14]	R_power	Set these bits to control how many DMA transfers can occur before the controller rearbitrates. The possible arbitration rate settings are: **b0000** Arbitrates after each DMA transfer. **b0001** Arbitrates after 2 DMA transfers. **b0010** Arbitrates after 4 DMA transfers. **b0011** Arbitrates after 8 DMA transfers. **b0100** Arbitrates after 16 DMA transfers. **b0101** Arbitrates after 32 DMA transfers. **b0110** Arbitrates after 64 DMA transfers. **b0111** Arbitrates after 128 DMA transfers. **b1000** Arbitrates after 256 DMA transfers. **b1001** Arbitrates after 512 DMA transfers. **b1010-b1111** Arbitrates after 1024 DMA transfers. This means that no arbitration occurs during the DMA transfer because the maximum transfer size is 1024.

Table 2-16 channel_cfg bit assignments (continued)

Bit	Name	Description
[13:4]	n_minus_1	Prior to the DMA cycle commencing, these bits represent the total number of DMA transfers that the DMA cycle contains. You must set these bits according to the size of DMA cycle that you require. The 10-bit value indicates the number of DMA transfers, minus one. The possible values are: b000000000 = 1 DMA transfer b000000001 = 2 DMA transfers b000000010 = 3 DMA transfers b000000011 = 4 DMA transfers b000000100 = 5 DMA transfers . . . b111111111 = 1024 DMA transfers. The controller updates this field immediately prior to it entering the arbitration process. This enables the controller to store the number of outstanding DMA transfers that are necessary to complete the DMA cycle.
[3]	next_useburst	Controls if the chnl_useburst_set [C] bit is set to a 1, when the controller is performing a peripheral scatter-gather and is completing a DMA cycle that uses the alternate data structure. **Note** Immediately prior to completion of the DMA cycle that the alternate data structure specifies, the controller sets the chnl_useburst_set [C] bit to 0 if the number of remaining transfers is less than 2^R. The setting of the next_useburst bit controls if the controller performs an additional modification of the chnl_useburst_set [C] bit. In peripheral scatter-gather DMA cycle then after the DMA cycle that uses the alternate data structure completes, either: 0 = the controller does not change the value of the chnl_useburst_set [C] bit. If the chnl_useburst_set [C] bit is 0 then for all the remaining DMA cycles in the peripheral scatter-gather transaction, the controller responds to requests on **dma_req[ ]** and **dma_sreq[ ]**, when it performs a DMA cycle that uses an alternate data structure. 1 = the controller sets the chnl_useburst_set [C] bit to a 1. Therefore, for the remaining DMA cycles in the peripheral scatter-gather transaction, the controller only responds to requests on **dma_req[ ]**, when it performs a DMA cycle that uses an alternate data structure.

Table 2-16 channel_cfg bit assignments (continued)

Bit	Name	Description
[2:0]	cycle_ctrl	The operating mode of the DMA cycle. The modes are: **b000** Stop. Indicates that the data structure is invalid. **b001** Basic. The controller must receive a new request, prior to it entering the arbitration process, to enable the DMA cycle to complete. **b010** Auto-request. The controller automatically inserts a request for the appropriate channel during the arbitration process. This means that the initial request is sufficient to enable the DMA cycle to complete. **b011** Ping-pong. The controller performs a DMA cycle using one of the data structures. After the DMA cycle completes, it performs a DMA cycle using the other data structure. After the DMA cycle completes and provided that the host processor has updated the original data structure, it performs a DMA cycle using the original data structure. The controller continues to perform DMA cycles until it either reads an invalid data structure or the host processor changes the cycle_ctrl bits to b001 or b010. See *Ping-pong* on page 2-23. **b100** Memory scatter/gather. See *Memory scatter-gather* on page 2-27. When the controller operates in memory scatter-gather mode, you must only use this value in the primary data structure. **b101** Memory scatter/gather. See *Memory scatter-gather* on page 2-27. When the controller operates in memory scatter-gather mode, you must only use this value in the alternate data structure. **b110** Peripheral scatter/gather. See *Peripheral scatter-gather* on page 2-31. When the controller operates in peripheral scatter-gather mode, you must only use this value in the primary data structure. **b111** Peripheral scatter/gather. See *Peripheral scatter-gather* on page 2-31. When the controller operates in peripheral scatter-gather mode, you must only use this value in the alternate data structure.

At the start of a DMA cycle, or 2^R DMA transfer, the controller fetches the channel_cfg from system memory. After it performs 2^R, or N, transfers it stores the updated channel_cfg in system memory.

The controller does not support a dst_size value that is different to the src_size value. If it detects a mismatch in these values, it uses the src_size value for source and destination and when it next updates the n_minus_1 field, it also sets the dst_size field to the same as the src_size field.

After the controller completes the N transfers it sets the cycle_ctrl field to b000, to indicate that the channel_cfg data is invalid. This prevents it from repeating the same DMA transfer.

Address calculation

To calculate the source address of a DMA transfer, the controller performs a left shift operation on the n_minus_1 value by a shift amount that src_inc specifies, and then subtracts the resulting value from the source data end pointer. Similarly, to calculate the destination address of a DMA transfer, it performs a left shift operation on the n_minus_1 value by a shift amount that dst_inc specifies, and then subtracts the resulting value from the destination end pointer.

Depending on the value of src_inc and dst_inc, the source address and destination address can be calculated using the equations:

src_inc=b00 and dst_inc=b00

- source address = src_data_end_ptr - n_minus_1
- destination address = dst_data_end_ptr - n_minus_1.

src_inc=b01 and dst_inc=b01

- source address = src_data_end_ptr - (n_minus_1 << 1)
- destination address = dst_data_end_ptr - (n_minus_1 << 1).

src_inc=b10 and dst_inc=b10

- source address = src_data_end_ptr - (n_minus_1 << 2)
- destination address = dst_data_end_ptr - (n_minus_1 << 2).

src_inc=b11 and dst_inc=b11

- source address = src_data_end_ptr
- destination address = dst_data_end_ptr.

Table 2-17 lists the destination addresses for a DMA cycle of six words.

Table 2-17 DMA cycle of six words using a word increment

Initial values of channel_cfg, prior to the DMA cycle				
src_size=b10, dst_inc=b10, n_minus_1=b101, cycle_ctrl=1				
	End Pointer	**Count**	**Difference**[a]	**Address**
DMA transfers	0x2AC	5	0x14	0x298
	0x2AC	4	0x10	0x29C
	0x2AC	3	0xC	0x2A0
	0x2AC	2	0x8	0x2A4
	0x2AC	1	0x4	0x2A8
	0x2AC	0	0x0	0x2AC
Final values of channel_cfg, after the DMA cycle				
src_size=b10, dst_inc=b10, n_minus_1=0, cycle_ctrl=0				

a. This value is the result of count being shifted left by the value of dst_inc.

Table 2-18 lists the destination addresses for a DMA transfer of 12 bytes using a halfword increment.

Table 2-18 DMA cycle of 12 bytes using a halfword increment

Initial values of channel_cfg, prior to the DMA cycle				
src_size=b00, dst_inc=b01, n_minus_1=b1011, cycle_ctrl=1, R_power=b11				
	End Pointer	**Count**	**Difference**[a]	**Address**
DMA transfers	0x5E7	11	0x16	0x5D1
	0x5E7	10	0x14	0x5D3
	0x5E7	9	0x12	0x5D5
	0x5E7	8	0x10	0x5D7
	0x5E7	7	0xE	0x5D9
	0x5E7	6	0xC	0x5DB
	0x5E7	5	0xA	0x5DD
	0x5E7	4	0x8	0x5DF
Values of channel_cfg after 2^R DMA transfers				
src_size=b00, dst_inc=b01, n_minus_1=b011, cycle_ctrl=1, R_power=b11				
	End Pointer	**Count**	**Difference**	**Address**
DMA transfers	0x5E7	3	0x6	0x5E1
	0x5E7	2	0x4	0x5E3
	0x5E7	1	0x2	0x5E5
	0x5E7	0	0x0	0x5E7
Final values of channel_cfg, after the DMA cycle				
src_size=b00, dst_inc=b01, n_minus_1=0, cycle_ctrl=0[b], R_power=b11				

a. This value is the result of count being shifted left by the value of dst_inc.
b. After the controller completes the DMA cycle it invalidates the channel_cfg memory location by clearing the cycle_ctrl field.

Chapter 3
Programmer's Model

This chapter describes the µDMAC registers and provides information for programming the controller. It contains the following sections:

- *About the programmer's model* on page 3-2
- *Register descriptions* on page 3-3.

3.1 About the programmer's model

The following applies to the registers that the controller provides:

- The base address of the controller is not fixed and can be different for any particular system implementation. However, the offset of any particular register from the base address is fixed.
- You must not access reserved or unused address locations because this can result in unpredictable behavior of the controller.
- You must write reserved or unused bits of registers as zero, and ignore them on read unless otherwise stated in the relevant text.
- A system or power-on reset resets all register bits to a logic 0 unless otherwise stated in the relevant text.
- All registers support read/write accesses unless otherwise stated in the relevant text. A write updates the contents of a register and a read returns the contents of the register.

3.2 Register descriptions

This section describes the registers with the exception of the test registers that Chapter 4 *Programmer's Model for Test* describes. Table 3-1 lists the registers in base offset order.

Table 3-1 Register summary

Name	Base offset	Type	Reset value	Description
dma_status	0x000	RO	0x-0nn0000[a]	*DMA status* on page 3-5
dma_cfg	0x004	WO	-	*DMA configuration* on page 3-7
ctrl_base_ptr	0x008	R/W	0x00000000	*Channel control data base pointer* on page 3-8
alt_ctrl_base_ptr	0x00C	RO	0x000000nn[b]	*Channel alternate control data base pointer* on page 3-9
dma_waitonreq_status	0x010	RO	0x00000000	*Channel wait on request status* on page 3-10
chnl_sw_request	0x014	WO	-	*Channel software request* on page 3-11
chnl_useburst_set	0x018	R/W	0x00000000	*Channel useburst set* on page 3-12
chnl_useburst_clr	0x01C	WO	-	*Channel useburst clear* on page 3-14
chnl_req_mask_set	0x020	R/W	0x00000000	*Channel request mask set* on page 3-15
chnl_req_mask_clr	0x024	WO	-	*Channel request mask clear* on page 3-16
chnl_enable_set	0x028	R/W	0x00000000	*Channel enable set* on page 3-17
chnl_enable_clr	0x02C	WO	-	*Channel enable clear* on page 3-18
chnl_pri_alt_set	0x030	R/W	0x00000000	*Channel primary-alternate set* on page 3-19
chnl_pri_alt_clr	0x034	WO	-	*Channel primary-alternate clear* on page 3-21
chnl_priority_set	0x038	R/W	0x00000000	*Channel priority set* on page 3-23
chnl_priority_clr	0x03C	WO	-	*Channel priority clear* on page 3-24
-	0x040-0x48	-	-	Reserved
err_clr	0x04C	R/W	0x00000000	*Bus error clear* on page 3-25
-	0x050-0xDFC	-	-	Reserved
Test registers				
-	0xE00-0xE48	-	-	See Chapter 4 *Programmer's Model for Test*

Table 3-1 Register summary (continued)

Name	Base offset	Type	Reset value	Description
-	0xE4C-0xFCC	-	-	Reserved
Identification registers				
periph_id_4	0xFD0	RO	0x04	*Peripheral identification 4* on page 3-30
-	0xFD4-0xFDC	-	-	Reserved
periph_id_0	0xFE0	RO	0x30	*Peripheral identification 0* on page 3-27
periph_id_1	0xFE4	RO	0xB2	*Peripheral identification 1* on page 3-27
periph_id_2	0xFE8	RO	0x-B[c]	*Peripheral identification 2* on page 3-28
periph_id_3	0xFEC	RO	0x00	*Peripheral identification 3* on page 3-29
pcell_id_0	0xFF0	RO	0x0D	*PrimeCell identification 0* on page 3-31
pcell_id_1	0xFF4	RO	0xF0	*PrimeCell identification 1* on page 3-32
pcell_id_2	0xFF8	RO	0x05	*PrimeCell identification 2* on page 3-32
pcell_id_3	0xFFC	RO	0xB1	*PrimeCell identification 3* on page 3-33

a. The reset value depends on the number of DMA channels that you configure the controller to use and if it includes the integration test logic.
b. The reset value depends on the number of DMA channels that you configure the controller to use.
c. This value depends on the revision status of the controller.

3.2.1 DMA status

The read-only dma_status Register returns the status of the controller. You cannot read this register when the controller is in the reset state. Figure 3-1 shows the bit assignments for this register.

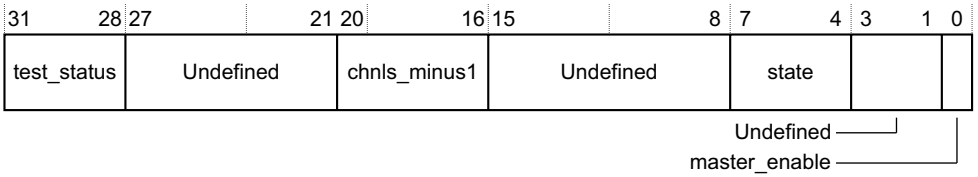


Figure 3-1 dma_status Register bit assignments

Table 3-2 lists the bit assignments for this register.

Table 3-2 dma_status Register bit assignments

Bit	Name	Description
[31:28]	test_status	To reduce the gate count you can configure the controller, to exclude the integration test logic. Read as: `0x0` = controller does not include the integration test logic `0x1` = controller includes the integration test logic `0x2-0xF` = undefined.
[27:21]	-	Undefined.
[20:16]	chnls_minus1	Number of available DMA channels minus one. For example: b00000 = controller configured to use 1 DMA channel b00001 = controller configured to use 2 DMA channels b00010 = controller configured to use 3 DMA channels . . . b11111 = controller configured to use 32 DMA channels.
[15:8]	-	Undefined.

Table 3-2 dma_status Register bit assignments (continued)

Bit	Name	Description
[7:4]	state	Current state of the control state machine. State can be one of the following: b0000 = idle b0001 = reading channel controller data b0010 = reading source data end pointer b0011 = reading destination data end pointer b0100 = reading source data b0101 = writing destination data b0110 = waiting for DMA request to clear b0111 = writing channel controller data b1000 = stalled b1001 = done b1010 = peripheral scatter-gather transition b1011-b1111 = undefined.
[3:1]	-	Undefined.
[0]	master_enable	Enable status of the controller: 0 = controller is disabled 1 = controller is enabled.

3.2.2 DMA configuration

The write-only dma_cfg Register controls the configuration of the controller. Figure 3-2 shows the bit assignments for this register.

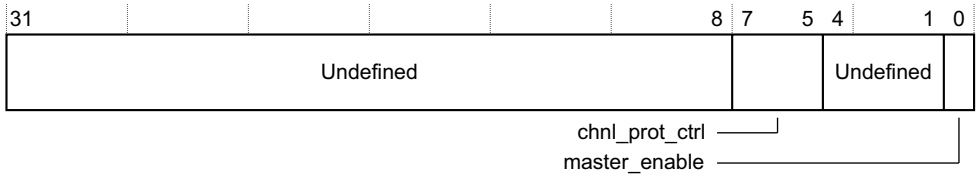


Figure 3-2 dma_cfg Register bit assignments

Table 3-3 lists the bit assignments for this register.

Table 3-3 dma_cfg Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined. Write as zero.
[7:5]	chnl_prot_ctrl	Sets the AHB-Lite protection by controlling the **HPROT[3:1]** signal levels as follows: **Bit [7]** Controls **HPROT[3]** to indicate if a cacheable access is occurring. **Bit [6]** Controls **HPROT[2]** to indicate if a bufferable access is occurring. **Bit [5]** Controls **HPROT[1]** to indicate if a privileged access is occurring. **Note** When bit [*n*] = 1 then the corresponding **HPROT** is HIGH. When bit [*n*] = 0 then the corresponding **HPROT** is LOW.
[4:1]	-	Undefined. Write as zero.
[0]	master_enable	Enable for the controller: 0 = disables the controller 1 = enables the controller.

3.2.3 Channel control data base pointer

The ctrl_base_ptr Register is a read/write register. You must configure this register so that the base pointer points to a location in your system memory.

Note

The controller provides no internal memory for storing the channel control data structure.

The amount of system memory that you must assign to the controller depends on the number of DMA channels and whether you configure it to use the alternate data structure. Therefore, the base pointer address requires a variable number of bits that depend on the system implementation.

You cannot read this register when the controller is in the reset state. Figure 3-3 shows the possible bit assignments for this register, depending on the number of DMA channels that you configure the controller to contain.

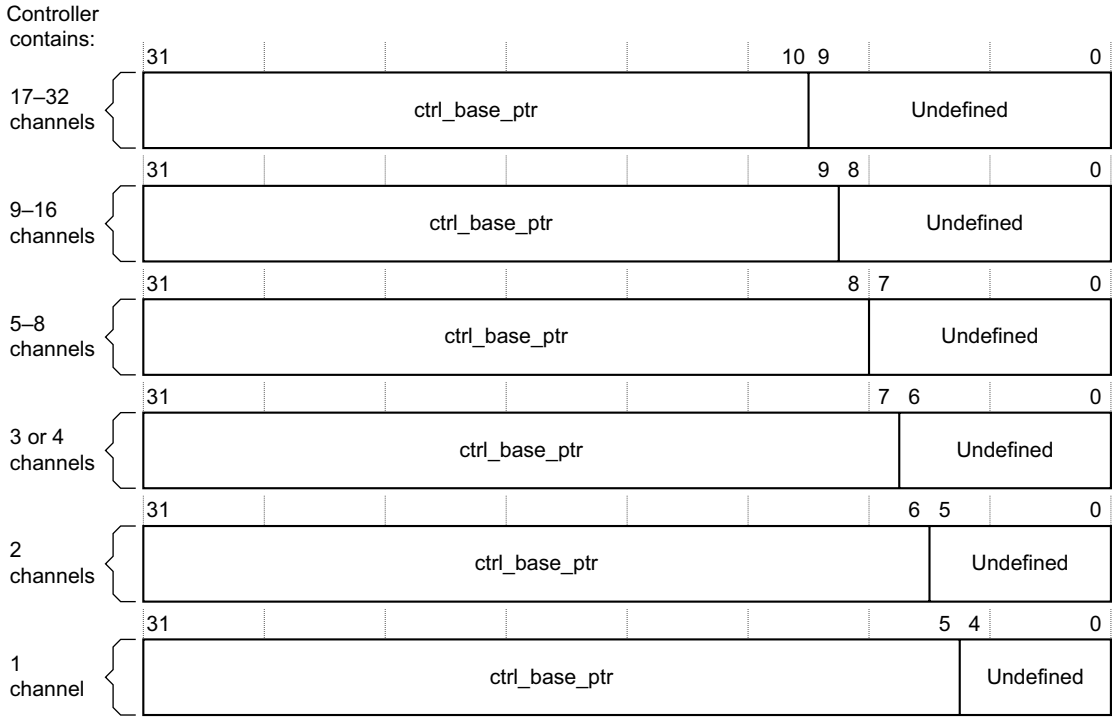


Figure 3-3 ctrl_base_ptr Register bit assignments

Table 3-4 lists the bit assignments for this register.

Table 3-4 ctrl_base_ptr Register bit assignments

Bit	Name	Description
[31:PL230_DMA_CHNL_BITS +5]	ctrl_base_ptr	Pointer to the base address of the primary data structure. See *Channel control data structure* on page 2-35 for information about the data structure.
[PL230_DMA_CHNL_BITS+4:0]	-	Undefined. Write as zero.

Where PL230_DMA_CHNL_BITS is defined as the minimum number of bits required to represent the number of DMA channels, minus one. The values that PL230_DMA_CHNL_BITS can be assigned are:

0 When the controller contains 1 DMA channel.

1 When the controller contains 2 DMA channels.

2 When the controller contains 3 or 4 DMA channels.

3 When the controller contains 5 to 8 DMA channels.

4 When the controller contains 9 to 16 DMA channels.

5 When the controller contains 17 to 32 DMA channels.

3.2.4 Channel alternate control data base pointer

The read-only alt_ctrl_base_ptr Register returns the base address of the alternate data structure. You cannot read this register when the controller is in the reset state. Figure 3-4 shows the bit assignments for this register.

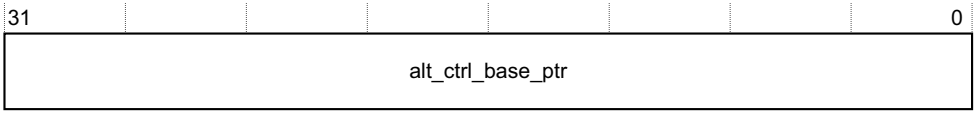


Figure 3-4 alt_ctrl_base_ptr Register bit assignments

This register removes the necessity for application software to calculate the base address of the alternate data structure. Table 3-5 lists the bit assignments for this register.

Table 3-5 alt_ctrl_base_ptr Register bit assignments

Bit	Name	Description
[31:0]	alt_ctrl_base_ptr	Base address of the alternate data structure

3.2.5 Channel wait on request status

The read-only dma_waitonreq_status Register returns the status of **dma_waitonreq[ ]**. You cannot read this register when the controller is in the reset state. Figure 3-5 shows the bit assignments for this register.

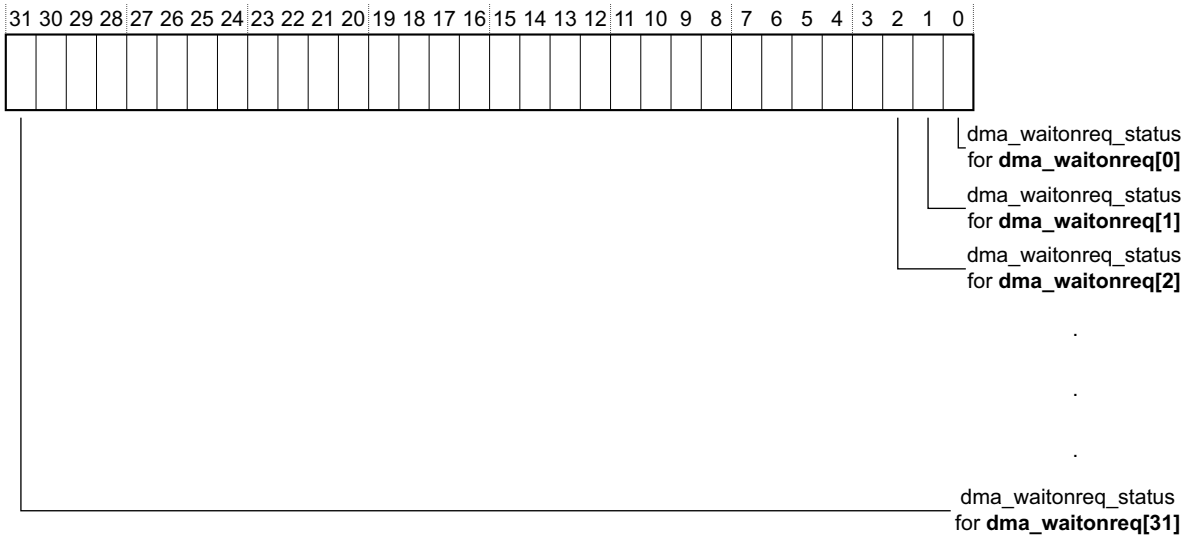


Figure 3-5 dma_waitonreq_status Register bit assignments

Table 3-6 lists the bit assignments for this register.

Table 3-6 dma_waitonreq_status Register bit assignments

Bit	Name	Description
[31:0]	dma_waitonreq_status	Channel wait on request status. Read as: **Bit [*C*] = 0** **dma_waitonreq[**C**]** is LOW. **Bit [*C*] = 1** **dma_waitonreq[**C**]** is HIGH.

3.2.6 Channel software request

The write-only chnl_sw_request Register enables you to generate a software DMA request. Figure 3-6 shows the bit assignments for this register.

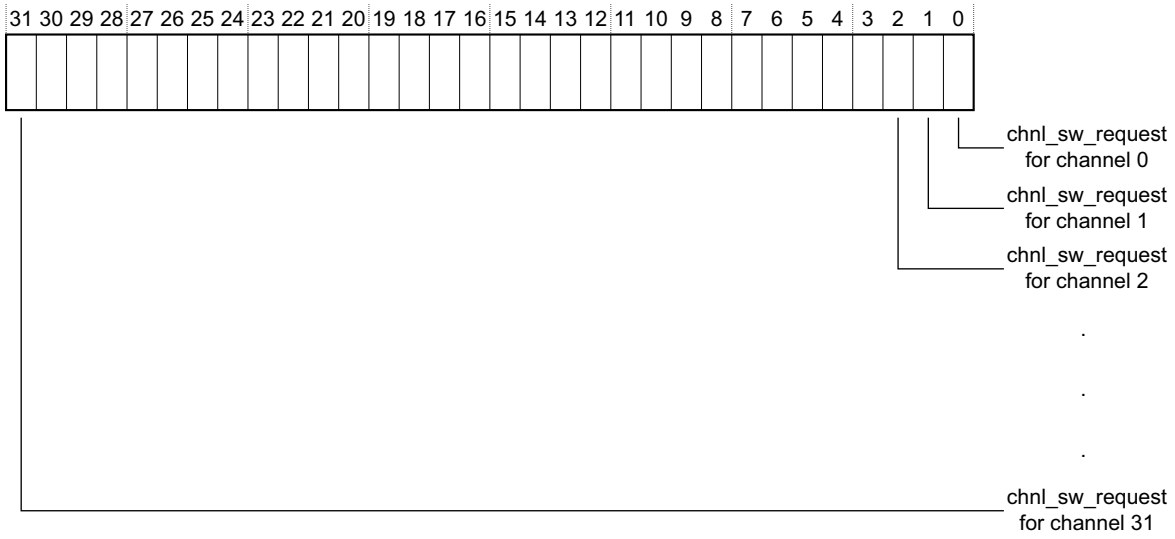


Figure 3-6 chnl_sw_request Register bit assignments

Table 3-7 lists the bit assignments for this register.

Table 3-7 chnl_sw_request Register bit assignments

Bit	Name	Description
[31:0]	chnl_sw_request	Set the appropriate bit to generate a software DMA request on the corresponding DMA channel. Write as: **Bit [*C*] = 0** Does not create a DMA request for channel C. **Bit [*C*] = 1** Creates a DMA request for channel C. Writing to a bit where a DMA channel is not implemented does not create a DMA request for that channel.

3.2.7 Channel useburst set

The read/write chnl_useburst_set Register disables the single request **dma_sreq[ ]** input from generating requests, and therefore only the request, **dma_req[ ]**, generates requests. Reading the register returns the useburst status. Figure 3-7 shows the bit assignments for this register.

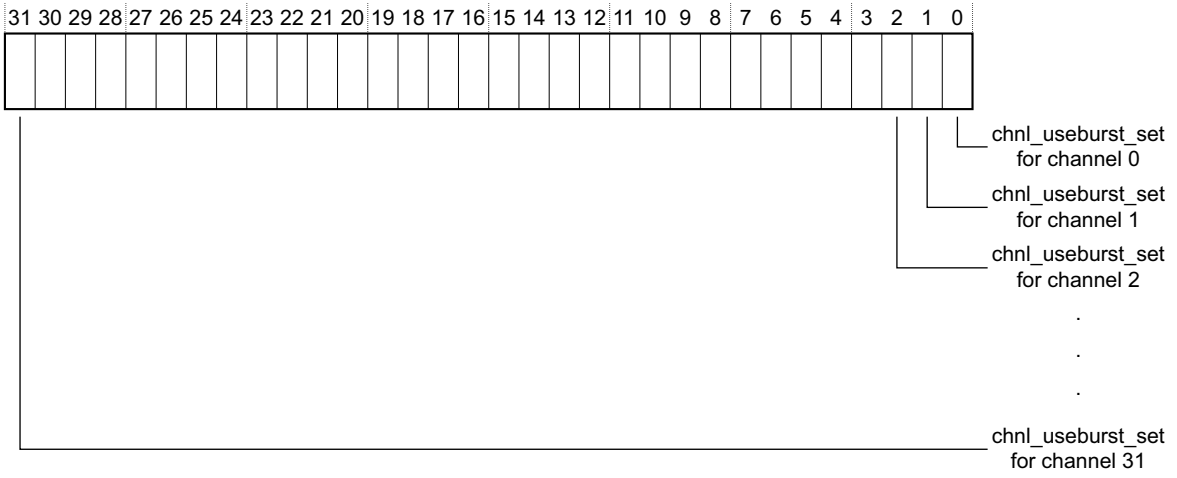


Figure 3-7 chnl_useburst_set Register bit assignments

Table 3-8 lists the bit assignments for this register.

Table 3-8 chnl_useburst_set Register bit assignments

Bit	Name	Description
[31:0]	chnl_useburst_set	Returns the useburst status, or disables **dma_sreq[**C**]** from generating DMA requests. Read as: **Bit [*C*] = 0** DMA channel C responds to requests that it receives on **dma_req[C]** or **dma_sreq[C]**. The controller performs 2^R, or single, bus transfers. **Bit [*C*] = 1** DMA channel C does not respond to requests that it receives on **dma_sreq[C]**. The controller only responds to **dma_req[C]** requests and performs 2^R transfers. Write as: **Bit [*C*] = 0** No effect. Use the chnl_useburst_clr Register to set bit [C] to 0. **Bit [*C*] = 1** Disables **dma_sreq[C]** from generating DMA requests. The controller performs 2^R transfers. Writing to a bit where a DMA channel is not implemented has no effect.

After the penultimate 2^R transfer completes, if the number of remaining transfers, N, is less than 2^R then the controller resets the chnl_useburst_set bit to 0. This enables you to complete the remaining transfers using **dma_req[ ]** or **dma_sreq[ ]**.

Note

If you program channel_cfg with a value of N less than 2^R then you must not set the corresponding chnl_useburst_set bit, if the peripheral does not assert **dma_req[ ]**.

In peripheral scatter-gather mode, if the next_useburst bit is set in channel_cfg then the controller sets the chnl_useburst_set [C] bit to a 1, when it completes the DMA cycle that uses the alternate data structure.

3.2.8 Channel useburst clear

The write-only chnl_useburst_clr Register enables **dma_sreq[ ]** to generate requests. Figure 3-8 shows the bit assignments for this register.

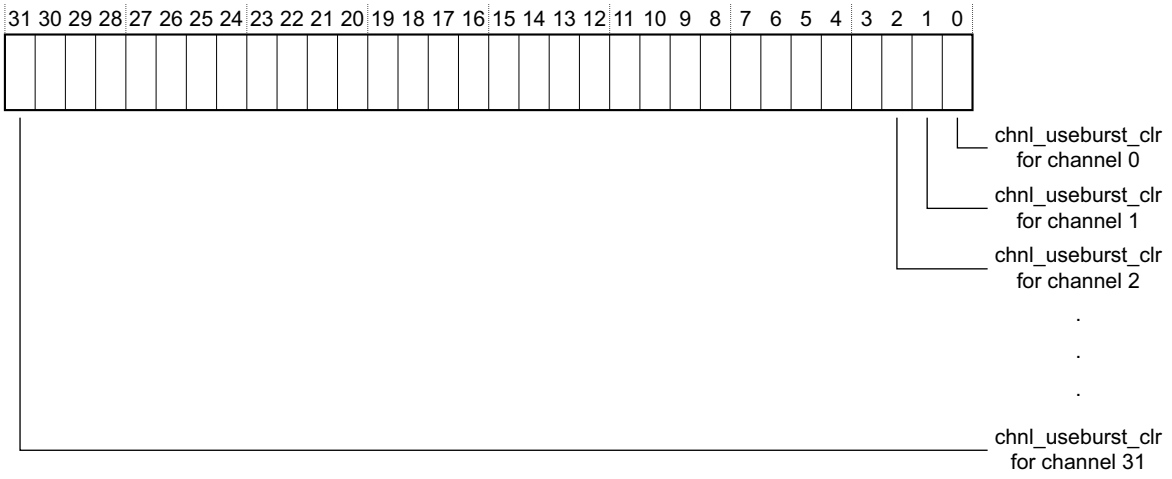


Figure 3-8 chnl_useburst_clr Register bit assignments

Table 3-9 lists the bit assignments for this register.

Table 3-9 chnl_useburst_clr Register bit assignments

Bit	Name	Description
[31:0]	chnl_useburst_clr	Set the appropriate bit to enable **dma_sreq[ ]** to generate requests. Write as: **Bit [*C*] = 0** No effect. Use the chnl_useburst_set Register to disable **dma_sreq[ ]** from generating requests. **Bit [*C*] = 1** Enables **dma_sreq[C]** to generate DMA requests. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.9 Channel request mask set

The read/write chnl_req_mask_set Register disables a HIGH on **dma_req[]**, or **dma_sreq[]**, from generating a request. Reading the register returns the request mask status for **dma_req[]** and **dma_sreq[]**. Figure 3-9 shows the bit assignments for this register.

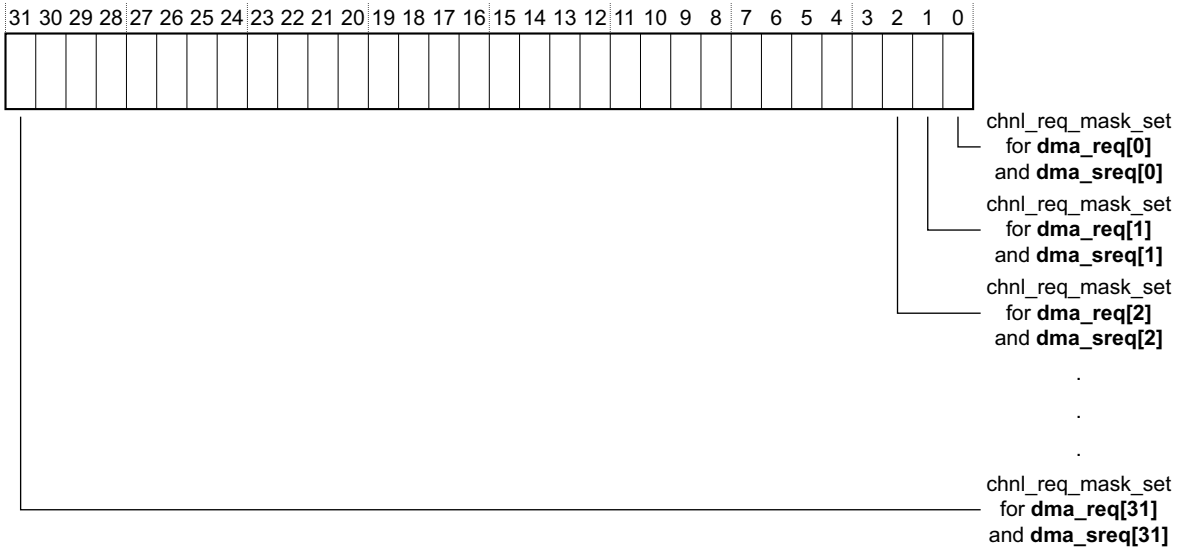


Figure 3-9 chnl_req_mask_set Register bit assignments

Table 3-10 lists the bit assignments for this register.

Table 3-10 chnl_req_mask_set Register bit assignments

Bit	Name	Description
[31:0]	chnl_req_mask_set	Returns the request mask status of **dma_req[]** and **dma_sreq[]**, or disables the corresponding channel from generating DMA requests. Read as: **Bit [*C*] = 0** External requests are enabled for channel C. **Bit [*C*] = 1** External requests are disabled for channel C. Write as: **Bit [*C*] = 0** No effect. Use the chnl_req_mask_clr Register to enable DMA requests. **Bit [*C*] = 1** Disables **dma_req[C]** and **dma_sreq[C]** from generating DMA requests. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.10 Channel request mask clear

The write-only chnl_req_mask_clr Register enables a HIGH on **dma_req[ ]**, or **dma_sreq[ ]**, to generate a request. Figure 3-10 shows the bit assignments for this register.

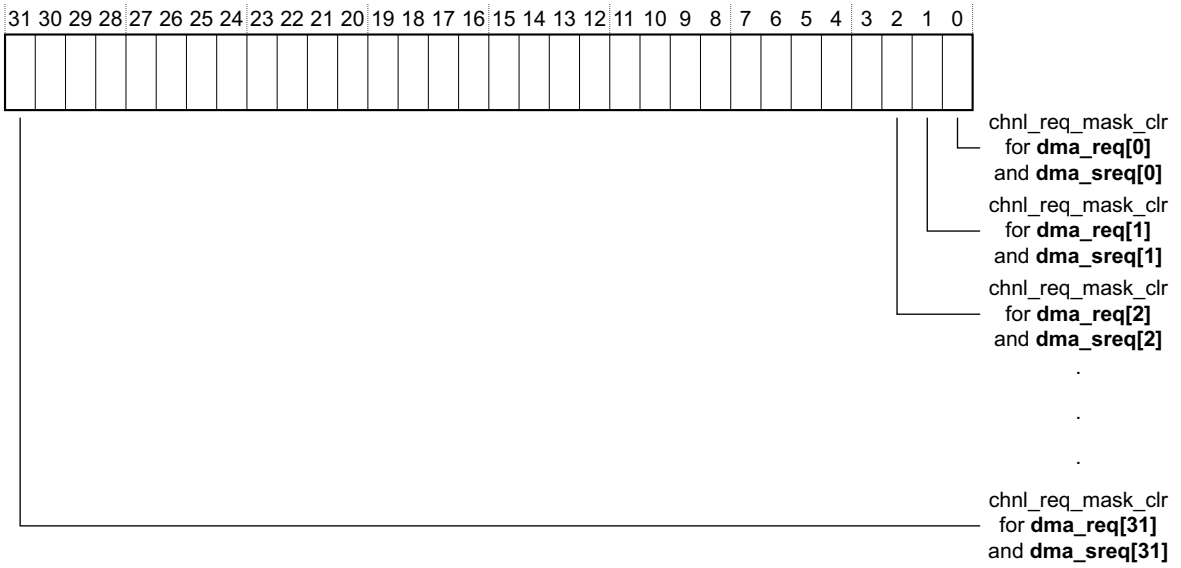


Figure 3-10 chnl_req_mask_clr Register bit assignments

Table 3-11 lists the bit assignments for this register.

Table 3-11 chnl_req_mask_clr Register bit assignments

Bit	Name	Description
[31:0]	chnl_req_mask_clr	Set the appropriate bit to enable DMA requests for the channel corresponding to **dma_req[ ]** and **dma_sreq[ ]**. Write as: **Bit [*C*] = 0** No effect. Use the chnl_req_mask_set Register to disable **dma_req[ ]** and **dma_sreq[ ]** from generating requests. **Bit [*C*] = 1** Enables **dma_req[C]** or **dma_sreq[C]** to generate DMA requests. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.11 Channel enable set

The read/write chnl_enable_set Register enables you to enable a DMA channel. Reading the register returns the enable status of the channels. Figure 3-11 shows the bit assignments for this register.

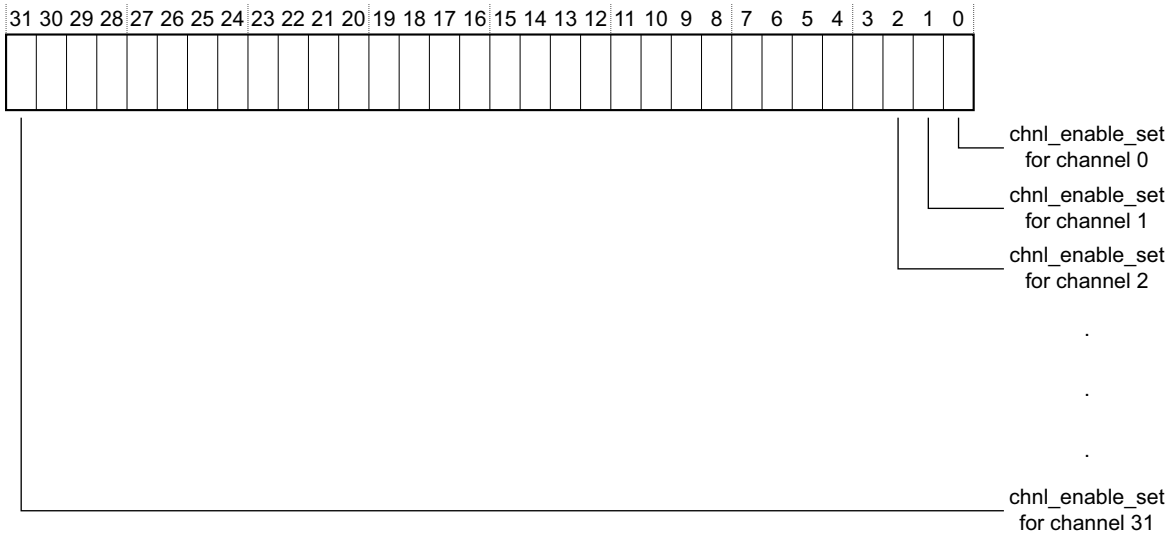


Figure 3-11 chnl_enable_set Register bit assignments

Table 3-12 lists the bit assignments for this register.

Table 3-12 chnl_enable_set Register bit assignments

Bit	Name	Description
[31:0]	chnl_enable_set	Returns the enable status of the channels, or enables the corresponding channels. Read as: **Bit [*C*] = 0** Channel C is disabled. **Bit [*C*] = 1** Channel C is enabled. Write as: **Bit [*C*] = 0** No effect. Use the chnl_enable_clr Register to disable a channel. **Bit [*C*] = 1** Enables channel C. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.12 Channel enable clear

The write-only chnl_enable_clr Register enables you to disable a DMA channel. Figure 3-12 shows the bit assignments for this register.

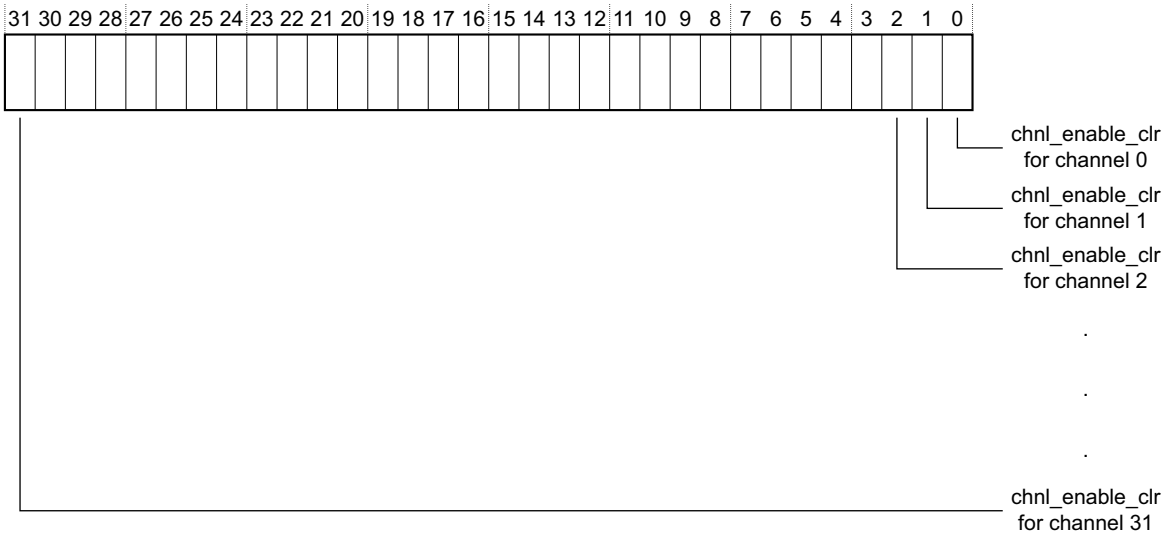


Figure 3-12 chnl_enable_clr Register bit assignments

Table 3-13 lists the bit assignments for this register.

Table 3-13 chnl_enable_clr Register bit assignments

Bit	Name	Description
[31:0]	chnl_enable_clr	Set the appropriate bit to disable the corresponding DMA channel. Write as: **Bit [*C*] = 0** No effect. Use the chnl_enable_set Register to enable DMA channels. **Bit [*C*] = 1** Disables channel C. Writing to a bit where a DMA channel is not implemented has no effect. **Note** The controller disables a channel, by setting the appropriate bit, when either: • it completes the DMA cycle • it reads a channel_cfg memory location which has cycle_ctrl = b000 • an ERROR occurs on the AHB-Lite bus.

3.2.13 Channel primary-alternate set

The read/write chnl_pri_alt_set Register enables you to configure a DMA channel to use the alternate data structure. Reading the register returns the status of which data structure is in use for the corresponding DMA channel. Figure 3-13 shows the bit assignments for this register.

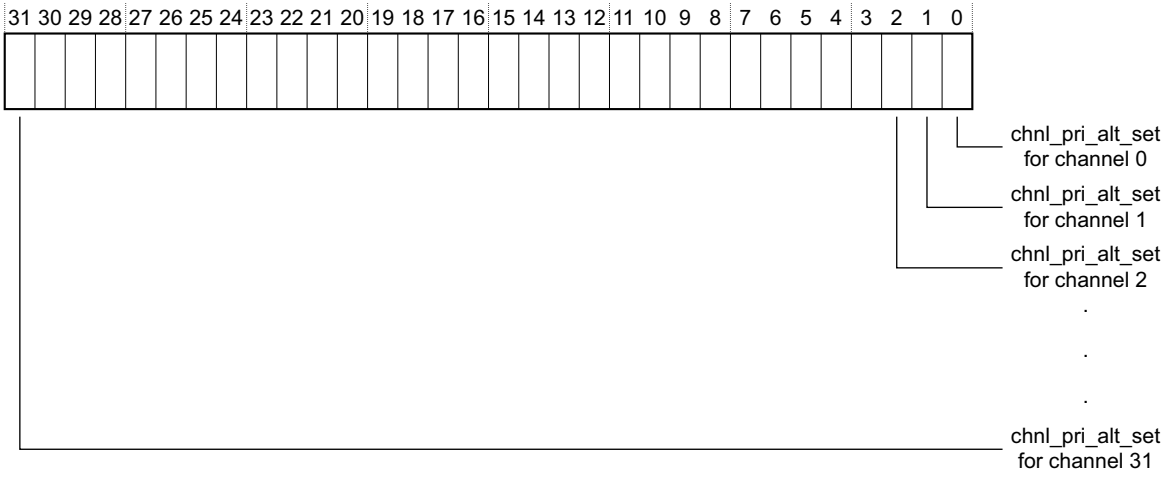


Figure 3-13 chnl_pri_alt_set Register bit assignments

Table 3-14 on page 3-20 lists the bit assignments for this register.

Table 3-14 chnl_pri_alt_set Register bit assignments

Bit	Name	Description
[31:0]	chnl_pri_alt_set	Returns the channel control data structure status, or selects the alternate data structure for the corresponding DMA channel. Read as: **Bit [*C*] = 0** DMA channel C is using the primary data structure. **Bit [*C*] = 1** DMA channel C is using the alternate data structure. Write as: **Bit [*C*] = 0** No effect. Use the chnl_pri_alt_clr Register to set bit [C] to 0. **Bit [*C*] = 1** Selects the alternate data structure for channel C. Writing to a bit where a DMA channel is not implemented has no effect. **Note** The controller toggles the value of the chnl_pri_alt_set [C] bit after it completes: • the four transfers that the primary data structure specifies for a memory scatter-gather, or peripheral scatter-gather, DMA cycle • all the transfers that the primary data structure specifies for a ping-pong DMA cycle • all the transfers that the alternate data structure specifies for the following DMA cycle types: — ping-pong — memory scatter-gather — peripheral scatter-gather.

3.2.14 Channel primary-alternate clear

The write-only chnl_pri_alt_clr Register enables you to configure a DMA channel to use the primary data structure. Figure 3-14 shows the bit assignments for this register.

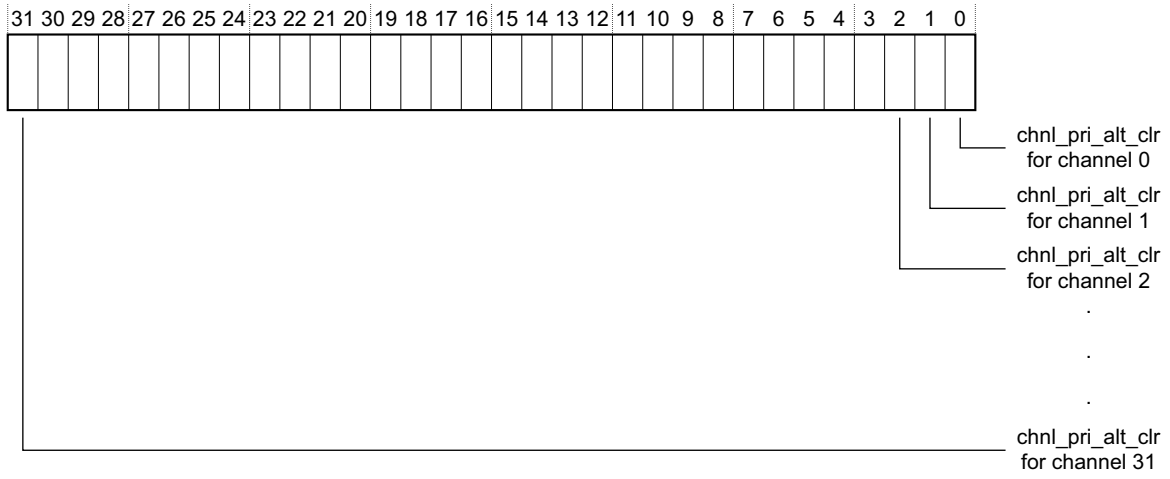


Figure 3-14 chnl_pri_alt_clr Register bit assignments

Table 3-15 on page 3-22 lists the bit assignments for this register.

Table 3-15 chnl_pri_alt_clr Register bit assignments

Bit	Name	Description
[31:0]	chnl_pri_alt_clr	Set the appropriate bit to select the primary data structure for the corresponding DMA channel. Write as: **Bit [*C*] = 0** No effect. Use the chnl_pri_alt_set Register to select the alternate data structure. **Bit [*C*] = 1** Selects the primary data structure for channel C. Writing to a bit where a DMA channel is not implemented has no effect. **Note** The controller toggles the value of the chnl_pri_alt_clr [C] bit after it completes: • the four transfers that the primary data structure specifies for a memory scatter-gather, or peripheral scatter-gather, DMA cycle • all the transfers that the primary data structure specifies for a ping-pong DMA cycle • all the transfers that the alternate data structure specifies for the following DMA cycle types: — ping-pong — memory scatter-gather — peripheral scatter-gather.

3.2.15 Channel priority set

The read/write chnl_priority_set Register enables you to configure a DMA channel to use the high priority level. Reading the register returns the status of the channel priority mask. Figure 3-15 shows the bit assignments for this register.

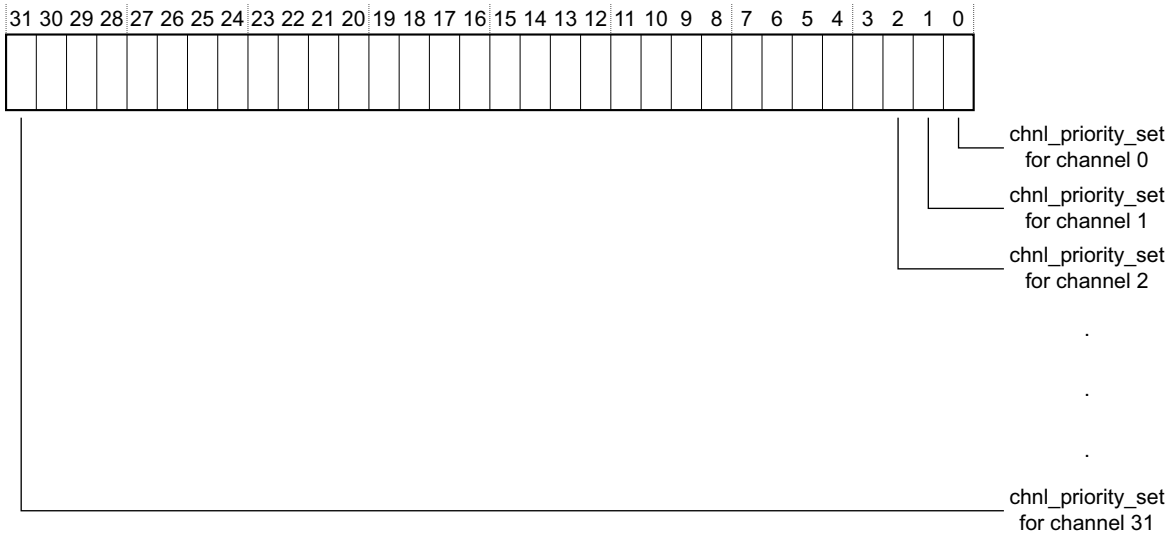


Figure 3-15 chnl_priority_set Register bit assignments

Table 3-16 lists the bit assignments for this register.

Table 3-16 chnl_priority_set Register bit assignments

Bit	Name	Description
[31:0]	chnl_priority_set	Returns the channel priority mask status, or sets the channel priority to high. Read as: **Bit [*C*] = 0** DMA channel C is using the default priority level. **Bit [*C*] = 1** DMA channel C is using a high priority level. Write as: **Bit [*C*] = 0** No effect. Use the chnl_priority_clr Register to set channel C to the default priority level. **Bit [*C*] = 1** Channel C uses the high priority level. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.16 Channel priority clear

The write-only chnl_priority_clr Register enables you to configure a DMA channel to use the default priority level. Figure 3-16 shows the bit assignments for this register.

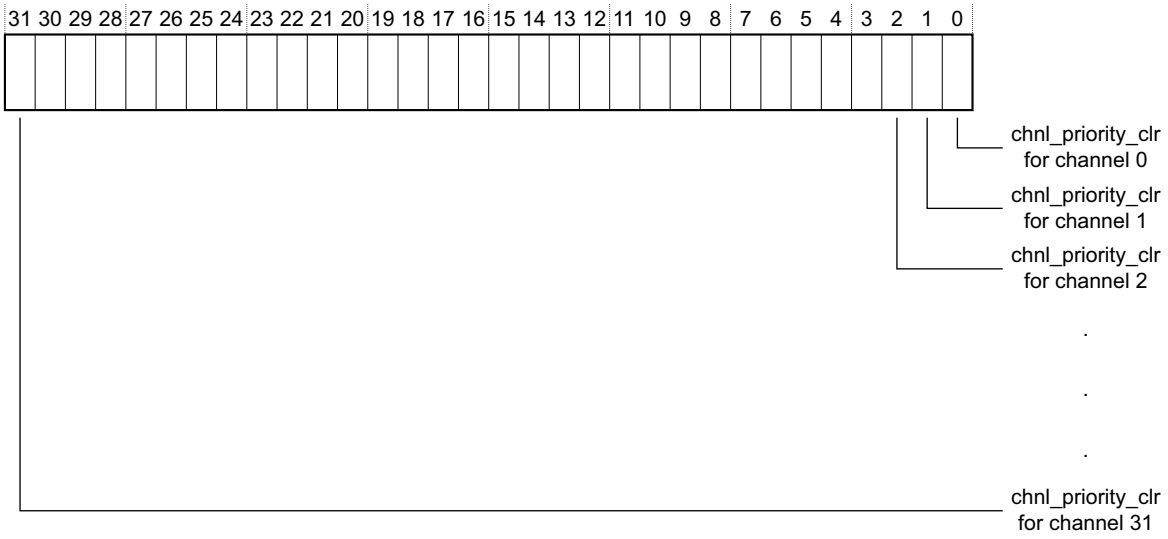


Figure 3-16 chnl_priority_clr Register bit assignments

Table 3-17 lists the bit assignments for this register.

Table 3-17 chnl_priority_clr Register bit assignments

Bit	Name	Description
[31:0]	chnl_priority_clr	Set the appropriate bit to select the default priority level for the specified DMA channel. Write as: **Bit [*C*] = 0** No effect. Use the chnl_priority_set Register to set channel C to the high priority level. **Bit [*C*] = 1** Channel C uses the default priority level. Writing to a bit where a DMA channel is not implemented has no effect.

3.2.17 Bus error clear

The read/write err_clr Register returns the status of **dma_err**, and enables you to set **dma_err** LOW. Figure 3-17 shows the bit assignments for this register.

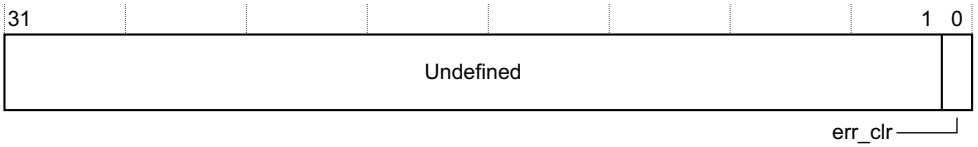


Figure 3-17 err_clr Register bit assignments

Table 3-18 lists the bit assignments for this register.

Table 3-18 err_clr Register bit assignments

Bit	Name	Description
[31:1]	-	Undefined. Write as zero.
[0]	err_clr	Returns the status of **dma_err**, or sets the signal LOW. Read as: 0 = **dma_err** is LOW 1 = **dma_err** is HIGH. Write as: 0 = No effect, status of **dma_err** is unchanged. 1 = Sets **dma_err** LOW. For test purposes, use the err_set register to set **dma_err** HIGH. See *Bus error set* on page 4-11. **Note** If you deassert **dma_err** at the same time as an ERROR occurs on the AHB-Lite bus, then the ERROR condition takes precedence and **dma_err** remains asserted.

3.2.18 Peripheral Identification Registers

The peripheral identification registers are located at the following addresses:

`0xFE0` periph_id_0 Register.

`0xFE4` periph_id_1 Register.

`0xFE8` periph_id_2 Register.

`0xFEC` periph_id_3 Register.

`0xFD0` periph_id_4 Register.

Each register is read-only and provides eight bits of data. You can consider the periph_id_[3:0] Registers conceptually as a single 32-bit register. Figure 3-18 shows the bit assignments for these registers.

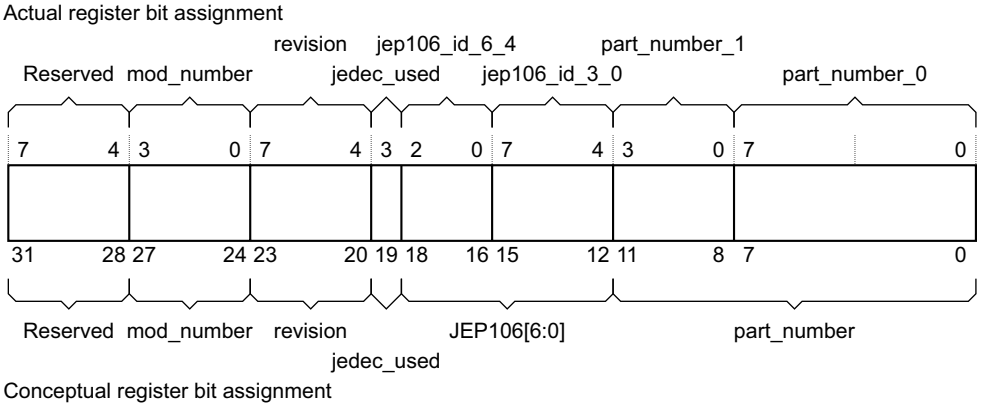


Figure 3-18 periph_id_[3:0] Register bit assignments

Table 3-19 lists the register bit assignments that the conceptual register provides.

Table 3-19 periph_id_[3:0] Register bit assignments

Bits	Name	Description
[31:28]	Reserved	Reserved for future use. Reads are undefined.
[27:24]	mod_number	Identifies data that is relevant to the ARM partner.
[23:20]	revision	Identifies the revision number of the peripheral. The revision number starts from 0 and is revision-dependent.

Table 3-19 periph_id_[3:0] Register bit assignments (continued)

Bits	Name	Description
[19]	jedec_used	Identifies if the controller uses the JEP106 manufacturer's identity code.
[18:12]	JEP106[6:0]	Identifies the designer. This is set to `0x41`, to indicate that ARM designed the peripheral.
[11:0]	part_number	Identifies the peripheral.

The following sections describe the Peripheral Identification Registers:

- *Peripheral identification 0*
- *Peripheral identification 1*
- *Peripheral identification 2* on page 3-28
- *Peripheral identification 3* on page 3-29
- *Peripheral identification 4* on page 3-30.

Peripheral identification 0

The read-only periph_id_0 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-19 shows the bit assignments for this register.

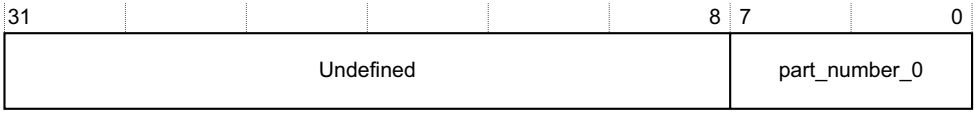


Figure 3-19 periph_id_0 Register bit assignments

Table 3-20 lists the bit assignments for this register.

Table 3-20 periph_id_0 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined
[7:0]	part_number_0	These bits read back as `0x30`

Peripheral identification 1

The read-only periph_id_1 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-20 on page 3-28 shows the bit assignments for this register.

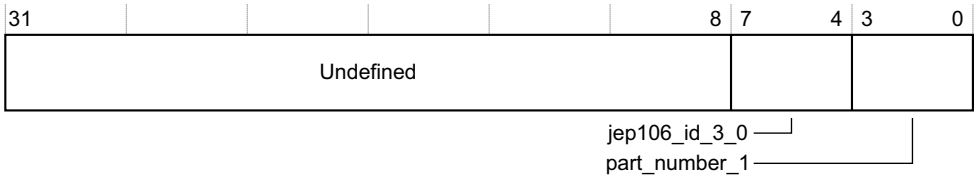


Figure 3-20 periph_id_1 Register bit assignments

Table 3-21 lists the bit assignments for this register.

Table 3-21 periph_id_1 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined.
[7:4]	jep106_id_3_0	JEP106 identity code [3:0]. See the *JEP106, Standard Manufacturer's Identification Code*. These bits read back as `0xB` because ARM is the designer of the peripheral.
[3:0]	part_number_1	These bits read back as `0x2`.

Peripheral identification 2

The read-only periph_id_2 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-21 shows the bit assignments for this register.

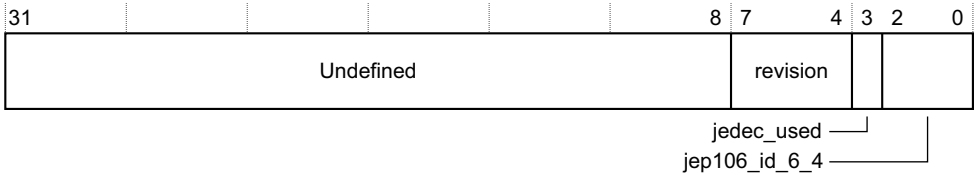


Figure 3-21 periph_id_2 Register bit assignments

Table 3-22 lists the bit assignments for this register.

Table 3-22 periph_id_2 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined.
[7:4]	revision	The revision status of the controller. For revision r0p0, these bits read back as `0x0`.
[3]	jedec_used	This indicates that the controller uses a manufacturer's identity code that was allocated by JEDEC according to JEP106. These bits always read back as `0x1`.
[2:0]	jep106_id_6_4	JEP106 identity code [6:4]. See the *JEP106, Standard Manufacturer's Identification Code*. These bits read back as `0x3` because ARM is the designer of this peripheral.

Peripheral identification 3

The read-only periph_id_3 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-22 shows the bit assignments for this register.

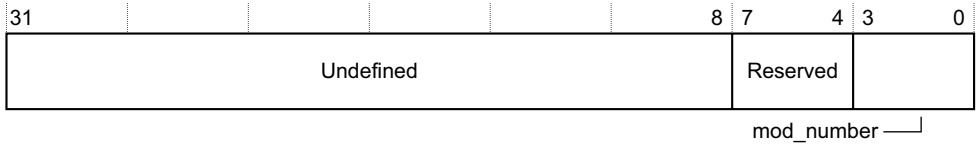


Figure 3-22 periph_id_3 Register bit assignments

Table 3-23 lists the bit assignments for this register.

Table 3-23 periph_id_3 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined.
[7:4]	Reserved	Reserved for future use. Reads are undefined.
[3:0]	mod_number	The customer must update this field if they modify the RTL of the controller. ARM set this to `0x0`.

Peripheral identification 4

The read-only periph_id_4 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-23 shows the bit assignments for this register.

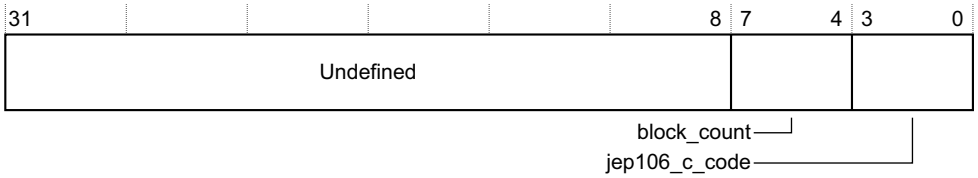


Figure 3-23 periph_id_4 Register bit assignments

Table 3-24 lists the bit assignments for this register.

Table 3-24 periph_id_4 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined.
[7:4]	block_count	The number of 4KB address blocks you require, to access the registers, expressed in powers of 2. These bits read back as `0x0`.
[3:0]	jep106_c_code	The JEP106 continuation code value represents how many `0x7F` continuation characters occur in the manufacturer's identity code. See *JEP106, Standard Manufacturer's Identification Code*. These bits read back as `0x4`.

3.2.19 PrimeCell identification Registers

The PrimeCell ID value is a 32-bit value and to ensure that it is accessible in all systems, the 32 bits are implemented as four 8-bit registers. In each register, only the least significant eight bits contain the data. The PrimeCell identification registers are located at the following addresses:

`0xFF0` pcell_id_0 Register.

`0xFF4` pcell_id_1 Register.

`0xFF8` pcell_id_2 Register.

`0xFFC` pcell_id_3 Register.

You can consider the registers conceptually as a single 32-bit register that contains a 32-bit PrimeCell ID value. You can use the register for automatic BIOS configuration. The pcell_id_[3:0] Register is set to `0xB105F00D`. Figure 3-24 on page 3-31 shows the bit assignments for these registers.

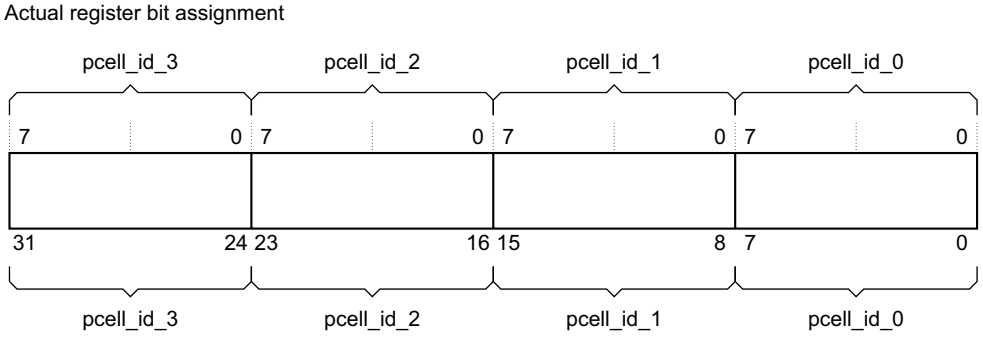


Figure 3-24 pcell_id_[3:0] Register bit assignments

Table 3-25 lists the register bit assignments that the conceptual register provides.

Table 3-25 pcell_id_[3:0] Register bit assignments

Bits	Name	Description
[31:24]	pcell_id_3	These bits read back as `0xB1`
[23:16]	pcell_id_2	These bits read back as `0x05`
[15:8]	pcell_id_1	These bits read back as `0xF0`
[7:0]	pcell_id_0	These bits read back as `0x0D`

The following subsections describe the PrimeCell Identification Registers:

- *PrimeCell identification 0*
- *PrimeCell identification 1* on page 3-32
- *PrimeCell identification 2* on page 3-32
- *PrimeCell identification 3* on page 3-33.

PrimeCell identification 0

The read-only pcell_id_0 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-25 on page 3-32 shows the bit assignments for this register.

Figure 3-25 pcell_id_0 Register bit assignments

Table 3-26 lists the bit assignments for this register.

Table 3-26 pcell_id_0 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined
[7:0]	pcell_id_0	These bits read back as `0x0D`

PrimeCell identification 1

The read-only pcell_id_1 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-26 shows the bit assignments for this register.

Figure 3-26 pcell_id_1 Register bit assignments

Table 3-27 lists the bit assignments for this register.

Table 3-27 pcell_id_1 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined
[7:0]	pcell_id_1	These bits read back as `0xF0`

PrimeCell identification 2

The read-only pcell_id_2 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-27 on page 3-33 shows the bit assignments for this register.

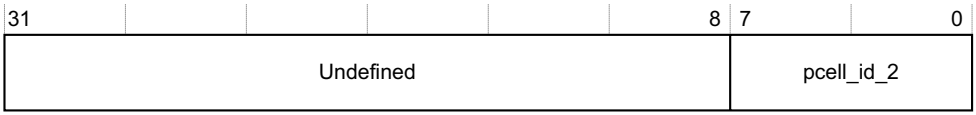


Figure 3-27 pcell_id_2 Register bit assignments

Table 3-28 lists the bit assignments for this register.

Table 3-28 pcell_id_2 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined
[7:0]	pcell_id_2	These bits read back as `0x05`

PrimeCell identification 3

The read-only pcell_id_3 Register is hard-coded, and therefore the fields in the register control the reset value. Figure 3-28 shows the bit assignments for this register.

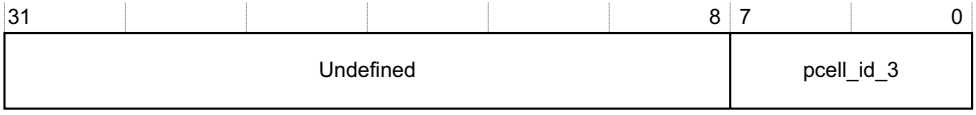


Figure 3-28 pcell_id_3 Register bit assignments

Table 3-29 lists the bit assignments for this register.

Table 3-29 pcell_id_3 Register bit assignments

Bit	Name	Description
[31:8]	-	Undefined
[7:0]	pcell_id_3	These bits read back as `0xB1`

Chapter 4
Programmer's Model for Test

This chapter describes the additional logic for functional verification and production testing. It contains the following section:

- *Register descriptions* on page 4-2.

Note

To enable you to access the test registers then you must configure the controller to include the integration test logic. See the *PrimeCell µDMA Controller (PL230) Configuration Guide* for information about how to include the integration test logic.

4.1 Register descriptions

This section describes the test registers. All register addresses in the controller are fixed relative to its base address. Table 4-1 lists the test registers in base offset order.

Table 4-1 Test register summary

Name	Base offset	Type	Reset value	Description
integration_cfg	0xE00	RW	0x0	*Integration configuration* on page 4-3
-	0xE04	-	-	Reserved
stall_status	0xE08	RO	0x0	*DMA stall status* on page 4-4
-	0xE0C	-	-	Reserved
dma_req_status	0xE10	RO	0x00000000	*DMA request status* on page 4-5
-	0xE14	-	-	Reserved
dma_sreq_status	0xE18	RO	0x00000000	*DMA single request status* on page 4-6
-	0xE1C	-	-	Reserved
dma_done_set	0xE20	RW	0x00000000	*DMA done set* on page 4-7
dma_done_clr	0xE24	WO	-	*DMA done clear* on page 4-8
dma_active_set	0xE28	RW	0x00000000	*DMA active set* on page 4-9
dma_active_clr	0xE2C	WO	-	*DMA active clear* on page 4-10
-	0xE30-0xE44	-	-	Reserved
err_set	0xE48	WO	-	*Bus error set* on page 4-11

4.1.1 Integration configuration

The read/write integration_cfg Register selects which logic controls the **dma_active[]**, **dma_done[]**, and **dma_err** outputs. You cannot read this register when the controller is in the reset state. Figure 4-1 shows the bit assignments for this register.

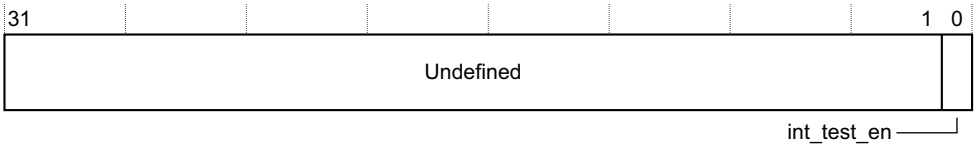


Figure 4-1 integration_cfg Register bit assignments

Table 4-2 lists the bit assignments for this register.

Table 4-2 integration_cfg Register bit assignments

Bit	Name	Description
[31:1]	-	Undefined. Write as zero.
[0]	int_test_en	Enables the integration test logic: 0 = disables the integration test logic. 1 = integration test logic controls the status of: • **dma_active[]** • **dma_done[]** • **dma_err**.

4.1.2 DMA stall status

The read-only dma_stall_status Register returns the status of **dma_stall**, irrespective of the status of the int_test_en bit. You cannot read this register when the controller is in the reset state. Figure 4-2 shows the bit assignments for this register.

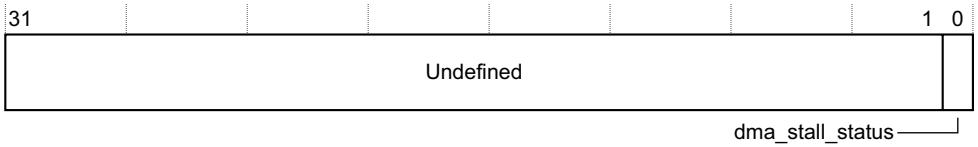


Figure 4-2 dma_stall_status Register bit assignments

Table 4-3 lists the bit assignments for this register.

Table 4-3 dma_stall_status Register bit assignments

Bit	Name	Description
[31:1]	-	Undefined.
[0]	dma_stall_status	Returns the status of **dma_stall**. Read as: 0 = **dma_stall** is LOW 1 = **dma_stall** is HIGH.

4.1.3 DMA request status

The read-only dma_req_status Register returns the status of **dma_req[ ]**, irrespective of the status of the int_test_en bit. You cannot read this register when the controller is in the reset state. Figure 4-3 shows the bit assignments for this register.

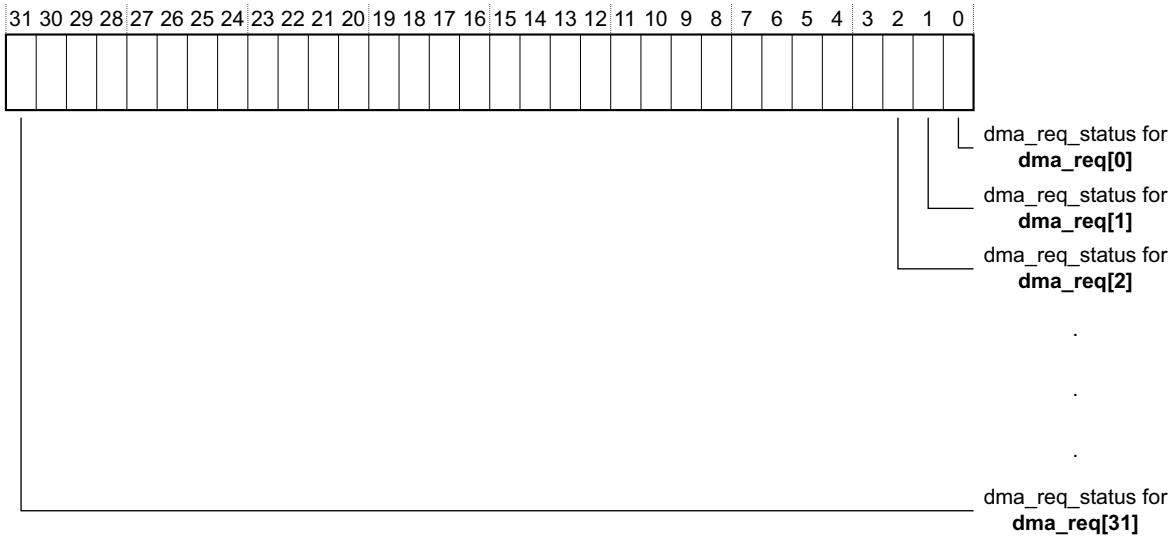


Figure 4-3 dma_req_status Register bit assignments

Table 4-4 lists the bit assignments for this register.

Table 4-4 dma_req_status Register bit assignments

Bit	Name	Description
[31:0]	dma_req_status	Returns the status of the DMA request signals, **dma_req[]**. Read as: **Bit [*C*] = 0** **dma_req[C]** is LOW. **Bit [*C*] = 1** **dma_req[C]** is HIGH.

4.1.4 DMA single request status

The read-only dma_sreq_status Register returns the status of **dma_sreq[ ]**, irrespective of the status of the int_test_en bit. You cannot read this register when the controller is in the reset state. Figure 4-4 shows the bit assignments for this register.

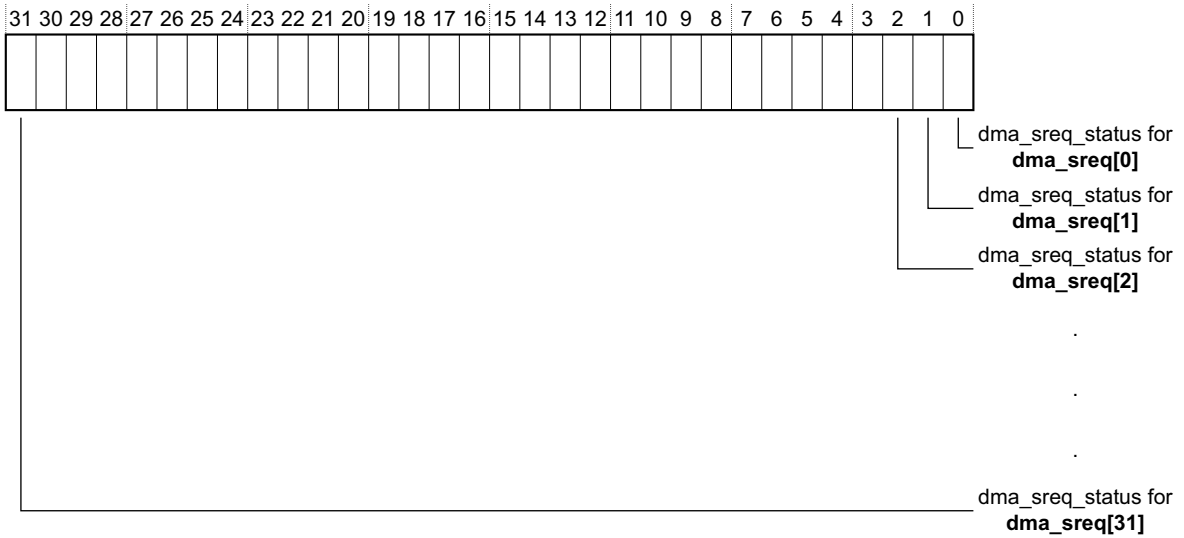


Figure 4-4 dma_sreq_status Register bit assignments

Table 4-5 lists the bit assignments for this register.

Table 4-5 dma_sreq_status Register bit assignments

Bit	Name	Description
[31:0]	dma_sreq_status	Returns the status of the DMA single request signals, **dma_sreq[]**. Read as: **Bit [*C*] = 0** **dma_sreq[C]** is LOW. **Bit [*C*] = 1** **dma_sreq[C]** is HIGH.

4.1.5 DMA done set

The read/write dma_done_set Register enables you to assert the **dma_done[ ]** signals. Reading the register returns the status of **dma_done[ ]**. Figure 4-5 shows the bit assignments for this register.

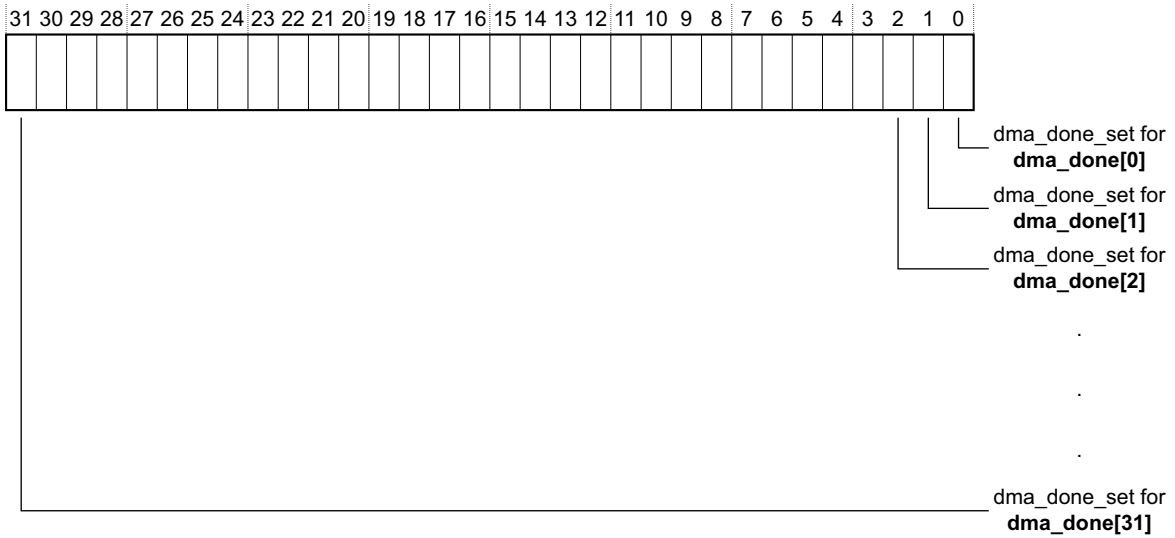


Figure 4-5 dma_done_set Register bit assignments

Table 4-6 lists the bit assignments for this register.

Table 4-6 dma_done_set Register bit assignments

Bit	Name	Description
[31:0]	dma_done_set	Returns the status of **dma_done[]**, or sets the signal HIGH. Reads, when int_test_en = 1[a]: **Bit [*C*] = 0** **dma_done[C]** is LOW. **Bit [*C*] = 1** **dma_done[C]** is HIGH. Write as: **Bit [*C*] = 0** No effect. Use the dma_done_clr Register to set **dma_done[C]** LOW. **Bit [*C*] = 1** Sets **dma_done[C]** HIGH, if int_test_en = 1[b]. Writing to a bit where a DMA channel is not implemented has no effect.

a. When int_test_en = 0, reads might not return the correct status of **dma_done[C]**.
b. When int_test_en = 0, writes have no effect on the status of **dma_done[C]**.

4.1.6 DMA done clear

The write-only dma_done_clr Register enables you to deassert the **dma_done[]** signals. Figure 4-6 shows the bit assignments for this register.

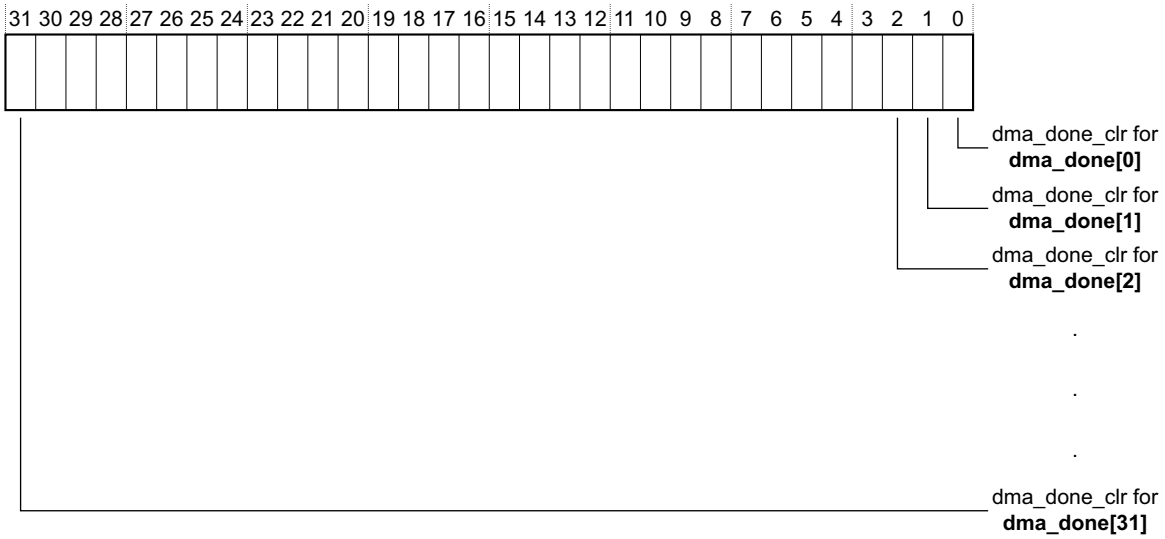


Figure 4-6 dma_done_clr Register bit assignments

Table 4-7 lists the bit assignments for this register.

Table 4-7 dma_done_clr Register bit assignments

Bit	Name	Description
[31:0]	dma_done_clr	Enables you to set the **dma_done[]** signals LOW. Write as: **Bit [*C*] = 0** No effect. Use the dma_done_set Register to set **dma_done[C]** HIGH. **Bit [*C*] = 1** Sets **dma_done[C]** LOW, if int_test_en = 1[a]. Writing to a bit where a DMA channel is not implemented has no effect.

a. When int_test_en = 0, writes have no effect on the status of **dma_done[C]**.

4.1.7 DMA active set

The read/write dma_active_set Register enables you to assert the **dma_active[ ]** signals. Reading the register returns the status of **dma_active[ ]**. Figure 4-7 shows the bit assignments for this register.

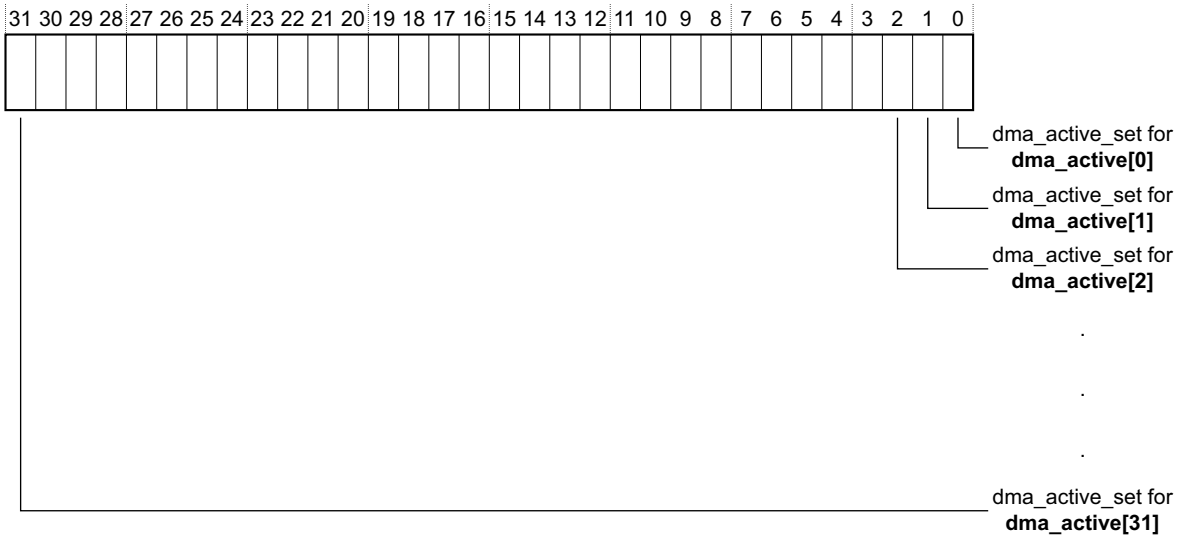


Figure 4-7 dma_active_set Register bit assignments

Table 4-8 lists the bit assignments for this register.

Table 4-8 dma_active_set Register bit assignments

Bit	Name	Description
[31:0]	dma_active_set	Returns the status of **dma_active[]**, or sets the signal HIGH. Reads, when int_test_en = 1[a]: **Bit [*C*] = 0** **dma_active[C]** is LOW. **Bit [*C*] = 1** **dma_active[C]** is HIGH. Write as: **Bit [*C*] = 0** No effect. Use the dma_active_clr Register to set **dma_active[C]** LOW. **Bit [*C*] = 1** Sets **dma_active[C]** HIGH, if int_test_en = 1[b]. Writing to a bit where a DMA channel is not implemented has no effect.

a. When int_test_en = 0, reads might not return the correct status of **dma_active[C]**.
b. When int_test_en = 0, writes have no effect on the status of **dma_active[C]**.

4.1.8 DMA active clear

The write-only dma_active_clr Register enables you to deassert the **dma_active[ ]** signals. Figure 4-8 shows the bit assignments for this register.

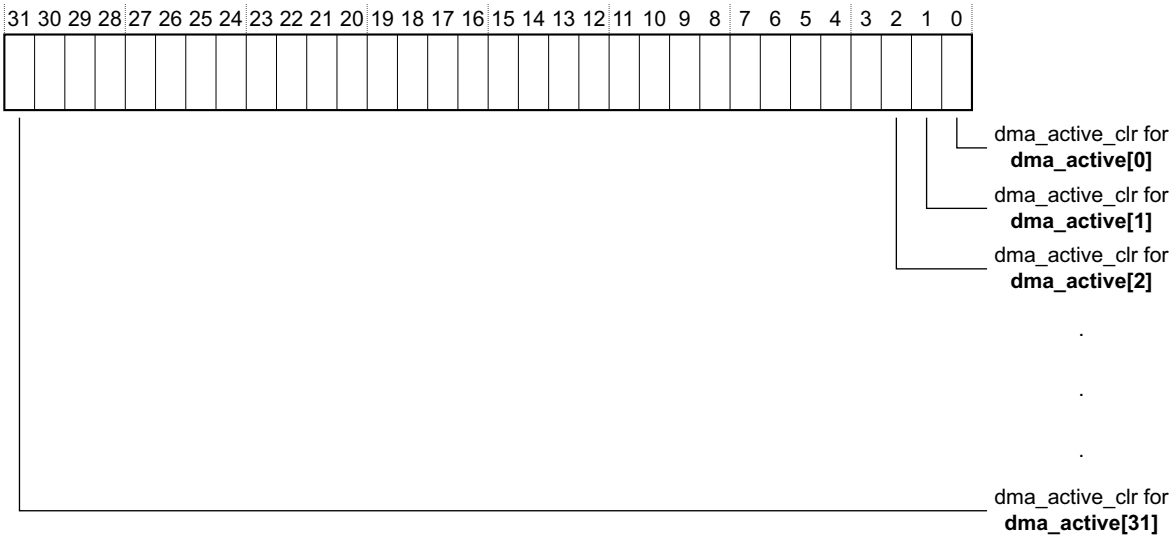


Figure 4-8 dma_active_clr Register bit assignments

Table 4-9 lists the bit assignments for this register.

Table 4-9 dma_active_clr Register bit assignments

Bit	Name	Description
[31:0]	dma_active_clr	Enables you to set the **dma_active[]** signals LOW. Write as: **Bit [*C*] = 0** No effect. Use the dma_active_set Register to set **dma_active[C]** HIGH. **Bit [*C*] = 1** Sets **dma_active[C]** LOW, if int_test_en = 1[a]. Writing to a bit where a DMA channel is not implemented has no effect.

a. When int_test_en = 0, writes have no effect on the status of **dma_active[C]**.

4.1.9 Bus error set

The write-only err_set Register enables you to assert **dma_err**. Figure 4-9 shows the bit assignments for this register.

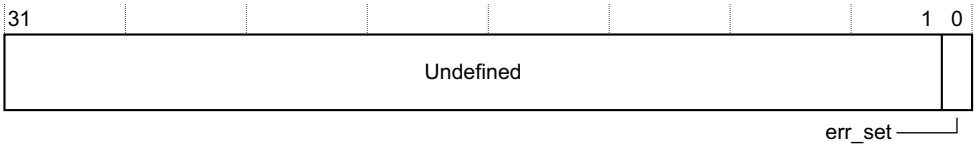


Figure 4-9 err_set Register bit assignments

Table 4-10 lists the bit assignments for this register.

Table 4-10 err_set Register bit assignments

Bit	Name	Description
[31:1]	-	Undefined. Write as zero.
[0]	err_set	Sets **dma_err** HIGH. Write as: 0 = no effect. Use the err_clr Register to set **dma_err** LOW. See *Bus error clear* on page 3-25. 1 = sets **dma_err** HIGH, if int_test_en = 1[a].

a. When int_test_en = 0, writes have no effect on the status of **dma_err**.

Appendix A
Signal Descriptions

This appendix describes the signals that the µDMAC uses. It contains the following sections:

A.1 Clock and reset signals

Table A-1 lists the clock and reset signals.

Table A-1 Clock and reset signals

Signal	Type	Source	Description
hclk	Input	Clock source	Clock for the AHB domain.
hresetn	Input	Reset source	Reset for the AHB domain. This signal is active LOW.

A.2 AHB-Lite master interface signals

The controller has the following AHB-Lite master signals:

- **haddr[31:0]**
- **hburst[2:0]**
- **hmastlock**
- **hprot[3:0]**
- **hrdata[31:0]**
- **hready**
- **hresp**
- **hsize[2:0]**
- **htrans[1:0]**
- **hwdata[31:0]**
- **hwrite**.

See the *AMBA 3 AHB-Lite Protocol v1.0 Specification* for a description of these signals.

Note

The controller ties some of these signals to a LOW, as *Signals tied to a steady state* describes.

A.2.1 Signals tied to a steady state

The controller does not implement all of the functionality that the *AMBA 3 AHB-Lite Protocol v1.0 Specification* describes and therefore some signals are tied to a steady state. Table A-2 lists these signals and the functionality that they provide.

Table A-2 Steady state signals

Signal	Type	Destination	Description
hburst[2:0]	Output	AHB bus	The controller does not support burst transfers, therefore these signals are tied LOW, to indicate a SINGLE transfer.
hmastlock	Output	AHB bus	The controller does not support locked transfers and therefore this signal is tied LOW.

Table A-2 Steady state signals (continued)

Signal	Type	Destination	Description
hprot[0]	Output	AHB bus	The controller always signals a data access using this protection signal and therefore this signal is tied HIGH.
hsize[2]	Output	AHB bus	The controller does not support data bus widths greater than 32 bits and therefore this signal is tied LOW.
htrans[1]	Output	AHB bus	The controller does not support BUSY or SEQ transfers and therefore this signal is tied LOW.

A.3 APB interface signals

The controller uses the following APB signals:

- **paddr[11:0]**
- **penable**
- **prdata[31:0]**
- **psel**
- **pwdata[31:0]**
- **pwrite**.

See the *AMBA Specification (Rev 2.0)* for a description of these signals.

paddr[11:0] deviates from the functionality described in the *AMBA Specification (Rev 2.0).* Table A-3 lists the functionality.

Table A-3 paddr[] bus

Signal	Type	Source	Description
paddr[11:0]	Input	APB master	The controller only supports single-word 32-bit accesses. The controller does not use the **paddr[1:0]** signals and it therefore interprets any byte or half-word access, as a word access.

The controller also provides an additional signal, **pclken**. This signal is not included in the *AMBA Specification (Rev 2.0)*. Table A-4 lists this signal and the functionality that it provides.

Table A-4 pclken signal

Signal	Type	Source	Description
pclken	Input	Clock generator	Clock enable signal that enables the APB interface to operate at either: • the **hclk** frequency • a divided **hclk** frequency that is an integer multiple of **hclk**. **Note** If **pclken** is not used then it must be tied HIGH. This results in the APB interface being clocked directly by **hclk**.

A.4 DMA control signals

Table A-5 lists the DMA control signals.

Table A-5 DMA control signals

Signal	Type	Source/ destination	Description
dma_active[n:0]	Output	Peripheral	When HIGH, it indicates that the controller is servicing the corresponding DMA channel. Only one **dma_active[ ]** signal can be active at any one time.
dma_done[n:0]	Output	Interrupt controller	Pulses HIGH, for a single AHB clock period, on completion of the corresponding DMA cycle. For enabled channels, only one **dma_done[ ]** signal can be active at any one time. When you disable a channel then: **dma_done[C]** = **dma_req[C]** OR (**dma_sreq[C]**[a] AND **dma_waitonreq[C]**)
dma_req[n:0]	Input	Peripheral	Request. When HIGH, it indicates that the peripheral is requesting the controller to service the corresponding DMA channel. The controller services the request by performing the DMA cycle using 2^R DMA transfers.
dma_sreq[n:0]	Input	Peripheral	Single request. When HIGH, it indicates that the peripheral is requesting the controller to service the corresponding DMA channel. The controller services the request by performing the DMA cycle using single DMA transfers.
dma_stall	Input	Peripheral	When HIGH, it indicates that a peripheral is requesting the controller to stall the current DMA transfer.
dma_waitonreq[n:0]	Input	Peripheral	When HIGH, it prevents **dma_active[ ]** from deasserting until **dma_req[ ]** and **dma_sreq[ ]** are LOW.

a. The corresponding chnl_useburst_set [C] bit must be 0. See Table 2-6 on page 2-11.

Where n = PL230_DMA_CHNLS-1.

Note

PL230_DMA_CHNLS is a configuration option. See the *PrimeCell µDMA Controller (PL230) Configuration Guide* for information about how to configure this option.

A.5 Interrupt signal

Table A-6 lists the interrupt signal.

Table A-6 Interrupt signal

Signal	Type	Destination	Description
dma_err	Output	Interrupt controller	When HIGH, it indicates that an ERROR has occurred on the AHB bus. When an ERROR occurs, the controller disables the active DMA channel by writing to the appropriate bit in the chnl_enable_set Register. To clear the interrupt, you must use the *Bus error clear* on page 3-25 to set **dma_err** LOW.

Glossary

This glossary describes some of the terms used in technical documents from ARM.

Advanced High-performance Bus (AHB)
A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave IP developments and ARM recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

Advanced Peripheral Bus (APB)
: A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

AHB
: *See* Advanced High-performance Bus.

AHB-Lite
: A subset of the full AMBA AHB protocol specification. It provides all of the basic functions required by the majority of AMBA AHB slave and master designs, particularly when used with a multi-layer AMBA interconnect. In most cases, the extra facilities provided by a full AMBA AHB interface are implemented more efficiently by using an AMBA AXI protocol interface.

Aligned
: A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

AMBA
: *See* Advanced Microcontroller Bus Architecture.

APB
: *See* Advanced Peripheral Bus.

Big-endian
: Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

  See also Little-endian and Endianness.

Byte
: An 8-bit data item.

Direct Memory Access (DMA)
: An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

DMA
: *See* Direct Memory Access.

Endianness
: Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

  See also Little-endian and Big-endian.

Halfword
: A 16-bit data item.

Little-endian
: Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

  See also Big-endian and Endianness.

Memory bank One of two or more parallel divisions of interleaved memory, usually one word wide, that enable reads and writes of multiple words at a time, rather than single words. All memory banks are addressed simultaneously and a bank enable or chip select signal determines which of the banks is accessed for each transfer. Accesses to sequential word addresses cause accesses to sequential banks. This enables the delays associated with accessing a bank to occur during the access to its adjacent bank, speeding up memory transfers.

Microprocessor *See* Processor.

Multi-layer An interconnect scheme similar to a cross-bar switch. Each master on the interconnect has a direct link to each slave, The link is not shared with other masters. This enables each master to process transfers in parallel with other masters. Contention only occurs in a multi-layer interconnect at a payload destination, typically the slave.

Multi-master AHB Typically a shared, not multi-layer, AHB interconnect scheme. More than one master connects to a single AMBA AHB link. In this case, the bus is implemented with a set of full AMBA AHB master interfaces. Masters that use the AMBA AHB-Lite protocol must connect through a wrapper to supply full AMBA AHB master signals to support multi-master operation.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Unaligned A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Unpredictable For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

Word A 32-bit data item.