

## General Description

The Nios<sup>®</sup> DMA module is an Altera<sup>®</sup> SOPC Builder library component included in the Nios development kit. The DMA module allows for efficient bulk data transfer between peripherals and memory by removing the CPU from the data path. The SOPC Builder DMA library component has available system choices to define device logic and interface signals on the Nios development board. The DMA's Verilog HDL or VHDL source code is available for development and includes the necessary software subroutines for easy system integration.

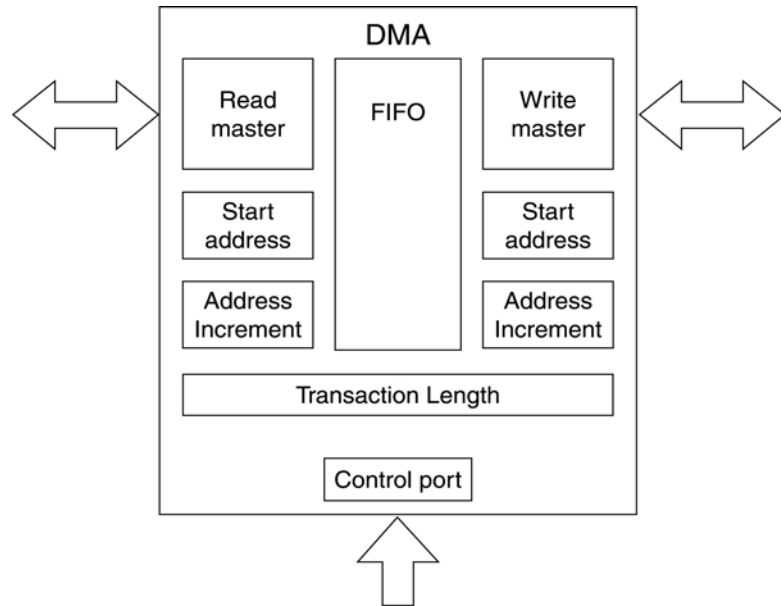
## Functional Description

The Nios direct memory access (DMA) peripheral is used to perform DMA data transfers between two memories, between a memory and a peripheral, or between two peripherals. The DMA peripheral is used in conjunction with streaming-capable peripherals, and allows data transfers of fixed or variable length without intervention from the CPU.

The DMA peripheral has two Avalon master ports—a master read port and a master write port—and one Avalon slave port for controlling the DMA as shown in [Figure 1 on page 2](#). The DMA determines its configuration by inspecting the slaves to which it is connected. The user can override the default values.



For a details on streaming Avalon data transfers and streaming Avalon peripherals, see the [Avalon Bus Specification Reference Manual](#).

**Figure 1. DMA Peripheral with Master & Slave Ports**

A typical DMA transfer proceeds as follows:

1. Software configures the DMA to transfer data by writing to the control port.
2. Software enables the DMA peripheral. The DMA peripheral then begins transferring data without additional intervention from the CPU.
3. The DMA's master read port reads data from the read address, which may be a memory or a peripheral, while the master write port writes the data to the destination address (a memory or peripheral). A shallow first-in first-out (FIFO) may buffer data between the read and write ports.
4. The DMA transfer ends when a specified number of bytes are transferred, or an "end of packet" (EOP) symbol is transferred. The DMA peripheral may issue an interrupt request at the end of the transfer.
5. During or after the transfer, software may determine if a transfer is in progress, or if the transfer ended (and how) by examining the DMA's status register.

Table 1 lists and describes the DMA register map

## DMA Registers

<b>Table 1. DMA Register Map</b>														
A2..A0	Register Name	R/W	Description/Register Bits											
			31	...	9	8	7	6	5	4	3	2	1	0
0	status <sup>(1)</sup>	RW								len	weop	reop	busy	done
1	readaddress	RW	Read master start address											
2	writeaddress	RW	Write master start address											
3	length	RW	Length in bytes											
4	reserved1	–	Reserved											
5	reserved2	–	Reserved											
6	control	RW			wcon	rcon	leen	ween	reen	i_en	go	word	hw	byte
7	reserved3	–	Reserved											

### Note

(1) A write operation to the status register clears the len, weop, reop, and done bits.

### status Register

The status register consists of individual bits that indicate particular conditions inside the DMA. The status register can be read at any time by software. Reading the status register does not change its value.

The status register bits are shown in Table 2:

<b>Table 2. status Register Bits</b>		
Bit Number	Bit Name	Description
0	done	A DMA transfer is completed.
1	busy	A DMA transfer is in progress.
2	reop	Read end of packet occurred.
3	weop	Write end of packet occurred.
4	len	A DMA transfer is completed and the requested number of bytes are transferred.

### *done Bit*

The `done` bit is set to 1 when an end of packet condition is detected or the specified length is completed. A write operation to the `status` register clears the `done` bit, which in turn clears the `IRQ`.

### *busy Bit*

The `busy` bit is 1 when a DMA transfer is in progress.

### *reop Bit*

The `reop` bit is 1 when a transfer is completed due to an end of packet event on the read side.

### *weop Bit*

The `weop` bit is 1 when a transfer is completed due to an end of packet event on the write side.

### *len Bit*

The `len` bit is 1 when a transfer completes with length bytes being transferred.

## **readaddress Register**

The `readaddress` register specifies the first location to be read in a DMA transfer. The `readaddress` register width is determined at system generation time. It is wide enough to address the full range of all slaves mastered by the read port.

## **writeaddress Register**

The `writeaddress` register specifies the first location to be written in a DMA transfer. The `writeaddress` register width is determined at system generation time. It is wide enough to address the full range of all slaves mastered by the write port.

## **length Register**

When written, the `length` register specifies the number of bytes to be transferred from the read port to the write port.

The `length` register width is determined at system generation time. It is wide enough to span any of the slave devices mastered by the read or write ports. The `length` register width can be overridden by supplying a minimum width value in the DMA wizard or in the `lengthwidth` PTF assignment. Overriding the `length` register may be necessary if the DMA master (read or write) only masters a data peripheral, such as a UART. In that case, the slave's address span is small, but it is likely that a large number of data items will need to be read or written.

The `length` register is decremented as each data value is written by the DMA write master. When `length` reaches 0, the `len` bit is set if enabled by the control register's `leen` bit. The `length` register does not decrement below 0.



The `length` register is specified in bytes. For word transfers, the value must be a multiple of 4. For half-word transfers, the value must be a multiple of 2.

## control Register

The control register is composed of individual bits that control the DMA's internal operation. The control register's value can be read at any time by software. The control register bits allow software to determine which, if any, conditions of the DMA transfer result in the end of a transfer and an interrupt request to software, if enabled.

The control register bits are shown in [Table 3](#):

<b>Table 3. control Register Bits</b>		
Bit Number	Bit Name	Description
0	byte	Byte (8-bit) transfer.
1	hw	Half-word (16-bit) transfer.
2	word	Word (32-bit) transfer.
3	go	Enable DMA.
4	i_en	Enable interrupt.
5	reen	Enable read end of packet.
6	ween	Enable write end of packet.
7	leen	End DMA transfer when <code>length</code> register reaches 0.
8	rcon	Read from a fixed address.
9	wcon	Write to a fixed address.

*byte, hw and word Bits*

The data width of a transfer is specified by the `byte`, `hw`, and `word` bits. Set only one of these bits to 1 for any given transfer. The width of the transfer is determined by the narrowest of the two slaves read and written in the transfer. For example, a DMA transfer that reads 16-bit flash and writes 32-bit on-chip memory is a half-word transfer. The setting is: `byte` = 0, `hw` = 1, and `word` = 0.

*go Bit*

When the `go` bit is set to 1, the DMA is enabled to execute its transfers, if any. For example, if the `length` register is non-zero, transfers begin when the `go` bit changes from 0 to 1.

*i\_en Bit*

When the `i_en` bit is 1, the DMA generates an interrupt request (sets its IRQ output to 1) when the status register's `done` bit is set to 1. When the `i_en` bit is 0, the IRQ output is always 0. A write operation to the status register clears the `done` bit, which in turn clears the IRQ.

*reen Bit*

When the `reen` bit is set to 1, a slave on the read side may end a transfer by asserting `endofpacket`.

*ween Bit*

When the `ween` bit is set to 1, a slave on the write side may end a transfer by asserting `endofpacket`.

*leen Bit*

When the `leen` bit is set to 1, the DMA transfer ends when the `length` register reaches 0. When this bit is set to 0, `length` reaching 0 does not cause a transfer to end.

*rcon Bit*

The `rcon` bit controls an aspect of read address incrementing. For details, see [“Address Incrementing” on page 7](#).

*wcon Bit*

The `wcon` bit controls an aspect of write address incrementing. For details, see [“Address Incrementing” on page 7](#).

## Address Incrementing

A DMA transfer is a sequence of read (or write) accesses to memory or peripheral devices. For a typical memory access, the read (or write) address increments by 1, 2 or 4 after each access, depending on the transfer's data size. A typical peripheral device (such as UART) has fixed read (or write) register locations—in this case, the read (or write) is held constant throughout the transfer. The rules for address incrementing are, in order of priority:

- If the control register's `rcon` (or `wcon`) bit is set, the read (or write) increment value is 0.
- Otherwise, the read and write increment values are set according to the transfer size specified in the control register:

Bit Name	Transfer Width	Increment
byte	byte	1
hw	half-word	2
word	word	4

## Interrupts

The DMA peripheral has a single IRQ output that equals 1 when the status register's `done` bit equals 1 and the control register's `i_en` bit equals 1.

A typical DMA interrupt handler reads the status register to check the transfer's conditions `len`, `reop`, and `weop` bits. Then it writes the status register to reset the conditions. If appropriate, another DMA transfer can then be started.

## Software Data Structure

```
typedef volatile struct
{
    int np_dmastatus;           // status register
    int np_dmareadaddress;      // read address
    int np_dmawriteaddress;     // write address
    int np_dmalength;           // length in bytes
    int np_dmareserved1;        // reserved
    int np_dmareserved2;        // reserved
    int np_dmacontrol;          // control register
    int np_dmareserved3;        // reserved
} np_dma;
```

## Software Subroutines

**Table 4** lists the DMA subroutines available in the Nios library (**lib** folder in the custom SDK) when one or more DMA peripherals are present in the Nios system. These functions are declared in the include file **nios.h**.

<b>Table 4. DMA Software Subroutines</b>	
<b>Subroutine</b>	<b>Description</b>
<code>nr_dma_copy_1_to_1</code>	Transfers <code>transfer_count</code> units of data between the unchanging source address and destination address.
<code>nr_dma_copy_1_to_range</code>	Transfers <code>transfer_count</code> units of data between the source address and destination address. The same source address is used repeatedly, while the destination address increments by <code>bytes_per_transfer</code> each transfer.
<code>nr_dma_copy_range_to_range</code>	Transfers <code>transfer_count</code> units of data between the source address and destination address. Both the source address and the destination address increment by <code>bytes_per_transfer</code> each transfer.
<code>nr_dma_copy_range_to_1</code>	Transfers <code>transfer_count</code> units of data between the source address and destination address. The source address increments by <code>bytes_per_transfer</code> each transfer, while the same destination address is used repeatedly.

### `nr_dma_copy_1_to_1`

This subroutine transfers `transfer_count` units of data between the unchanging source address and destination address.

#### *Syntax*

```
nr_dma_copy_1_to_1
(
    np_dma *dma,
    int bytes_per_transfer,
    void *source_address,
    void *destination_address,
    int transfer_count
);
```



*Parameters*

Parameter	Description
<code>dma</code>	Indicates which DMA peripheral to use.
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size.
<code>source_address</code>	Address to transfer data from.
<code>destination_address</code>	Address to transfer data to.
<code>transfer_count</code>	Number of individual data transfers to perform.

**`nr_dma_copy_1_to_range`**

This subroutine transfers `transfer_count` units of data between the source address and destination address. The same source address is used repeatedly, while the destination address increments by `bytes_per_transfer` each transfer.

*Syntax*

```
nr_dma_copy_1_to_range
(
    np_dma *dma,
    int bytes_per_transfer,
    void *source_address,
    void *first_destination_address,
    int transfer_count
);
```

*Parameters*

Parameter	Description
<code>dma</code>	Indicates which DMA peripheral to use.
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size.
<code>source_address</code>	Address to transfer data from.
<code>first_destination_address</code>	Address to transfer data to.
<code>transfer_count</code>	Number of individual data transfers to perform.

## **nr\_dma\_copy\_range\_to\_range**

This subroutine transfers `transfer_count` units of data between the source address and destination address. Both the source address and the destination address increment by `bytes_per_transfer` each transfer.

### *Syntax*

```
nr_dma_copy_range_to_range
(
    np_dma *dma,
    int bytes_per_transfer,
    void *first_source_address,
    void *first_destination_address,
    int transfer_count
);
```

### *Parameters*

Parameter	Description
<code>dma</code>	Indicates which DMA peripheral to use.
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size.
<code>first_source_address</code>	Address to transfer data from.
<code>first_destination_address</code>	Address to transfer data to.
<code>transfer_count</code>	Number of individual data transfers to perform.

## nr\_dma\_copy\_range\_to\_1

This subroutine transfers `transfer_count` units of data between the source address and destination address. The source address increments by `bytes_per_transfer` each transfer, while the same destination address is used repeatedly.

### Syntax

```
nr_dma_copy_range_to_1
(
    np_dma *dma,
    int bytes_per_transfer,
    void *first_source_address,
    void *destination_address,
    int transfer_count
);
```

### Parameters

Parameter	Description
<code>dma</code>	Indicates which DMA peripheral to use.
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size.
<code>first_source_address</code>	Address to transfer data from.
<code>destination_address</code>	Address to transfer data to.
<code>transfer_count</code>	Number of individual data transfers to perform.

## PTF Assignments

Table 5 lists the DMA's PTF file assignments. All of these assignments are located in the system PTF's `MODULE/WIZARD_SCRIPT_ARGUMENTS` section.

<i>Table 5. DMA PTF Assignments</i>			
Parameter	Type	Allowed Values	Default
<code>lengthwidth</code>	Decimal	1 .. 32	8
<code>fifo_depth</code>	Decimal	$\geq 2$	2
<code>fifo_in_logic_elements</code>	Boolean	1, 0	1

### **lengthwidth**

The `lengthwidth` assignment specifies the minimum value for the DMA length register's width. The length register width will not fall below the number specified in this assignment.

### **fifo\_depth**

The `fifo_depth` assignment specifies the minimum value for the number of locations in the internal memory.

### **fifo\_in\_logic\_elements**

When `fifo_in_logic_elements` is set to 1, the FIFO memory is built out of logic elements (LEs). When `fifo_in_logic_elements` is set to 0, the FIFO memory is built out of embedded system blocks (ESBs), embedded array blocks (EABs), and so on.

## Advanced PTF Assignments

Table 6 lists advanced PTF parameters for the DMA. Each parameter is a representation of a register or register bit and specifies its reset value. control register reset values are specified on a per-bit basis.

<b>Table 6. Advanced DMA PTF Parameters</b>			
<b>Parameter</b>	<b>Type</b>	<b>Allowed Values</b>	<b>Default Value</b>
allow_byte_transactions	Boolean	1, 0	1
allow_doubleword_transactions	Boolean	1, 0	1
allow_hw_transactions	Boolean	1, 0	1
allow_quadword_transactions	Boolean	1, 0	1
allow_word_transactions	Boolean	1, 0	1
control_byte_reset_value	Boolean	1,0	0
control_go_reset_value	Boolean	1,0	0
control_hw_reset_value	Boolean	1,0	0
control_i_en_reset_value	Boolean	1,0	0
control_leen_reset_value	Boolean	1,0	0
control_rcon_reset_value	Boolean	1,0	0
control_reen_reset_value	Boolean	1,0	0
control_wcon_reset_value	Boolean	1,0	0
control_ween_reset_value	Boolean	1,0	0
control_word_reset_value	Boolean	1,0	1
length_reset_value	Hexadecimal	The useful range of these values depends upon the corresponding register width.	0x0
readaddress_reset_value	Hexadecimal		0x0
writeaddress_reset_value	Hexadecimal		0x0



101 Innovation Drive  
San Jose, CA 95134  
(408) 544-7000  
<http://www.altera.com>  
[Applications Hotline:](#)  
(800) 800-EPLD  
[Literature Services:](#)  
[lit\\_req@altera.com](mailto:lit_req@altera.com)

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

