

Core Overview

The direct memory access (DMA) controller core with Avalon® interface performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon Memor-Mapped (Avalon-MM) master peripheral, such as a CPU, can offload memory transfer tasks to the DMA controller. While the DMA controller performs memory transfers, the master is free to perform other tasks in parallel.

The DMA controller transfers data as efficiently as possible, reading and writing data at the maximum pace allowed by the source or destination. The DMA controller is capable of performing Avalon transfers with flow control, enabling it to automatically transfer data to or from a slow peripheral with flow control (for example, UART), at the maximum pace allowed by the peripheral.

The DMA controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library. See [“Software Programming Model” on page 24–5](#) for details of HAL support.

This chapter contains the following sections:

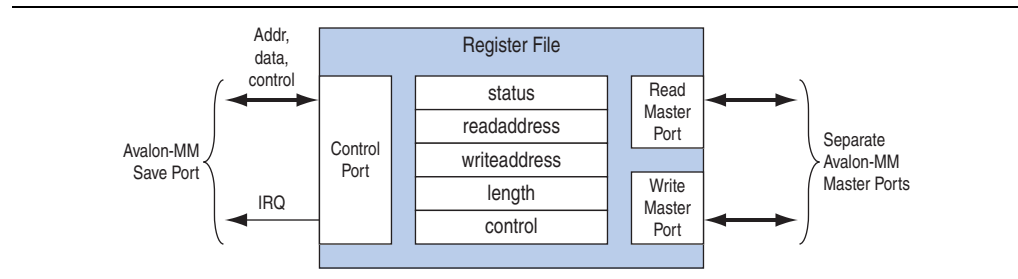
- [“Functional Description”](#)
- [“Instantiating the Core in SOPC Builder” on page 24–4](#)
- [“Device Support” on page 24–5](#)
- [“Software Programming Model” on page 24–5](#)

Functional Description

You can use the DMA controller to perform data transfers from a source address-space to a destination address-space. The controller has no concept of endianness and does not interpret the payload data. The concept of endianness only applies to a master that interprets payload data.

The source and destination may be either an Avalon-MM slave peripheral (for example, a constant address) or an address range in memory. The DMA controller can be used in conjunction with peripherals with flow control, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. A transaction is a sequence of one or more Avalon transfers initiated by the DMA controller core.

The DMA controller has two Avalon-MM master ports—a master read port and a master write port—and one Avalon-MM slave port for controlling the DMA as shown in [Figure 24–1](#).

Figure 24-1. DMA Controller Block Diagram

A typical DMA transaction proceeds as follows:

1. A CPU prepares the DMA controller for a transaction by writing to the control port.
2. The CPU enables the DMA controller. The DMA controller then begins transferring data without additional intervention from the CPU. The DMA's master read port reads data from the read address, which may be a memory or a peripheral. The master write port writes the data to the destination address, which can also be a memory or peripheral. A shallow FIFO buffers data between the read and write ports.
3. The DMA transaction ends when a specified number of bytes are transferred (a fixed-length transaction) or an end-of-packet signal is asserted by either the sender or receiver (a variable-length transaction). At the end of the transaction, the DMA controller generates an interrupt request (IRQ) if it was configured by the CPU to do so.
4. During or after the transaction, the CPU can determine if a transaction is in progress, or if the transaction ended (and how) by examining the DMA controller's status register.

Setting Up DMA Transactions

An Avalon-MM master peripheral sets up and initiates DMA transactions by writing to registers via the control port. The Avalon-MM master programs the DMA engine using byte addresses which are byte aligned. The master peripheral configures the following options:

- Read (source) address location
- Write (destination) address location
- Size of the individual transfers: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit) or quadword (128-bit)
- Enable interrupt upon end of transaction
- Enable source or destination to end the DMA transaction with end-of-packet signal
- Specify whether source and destination are memory or peripheral

The master peripheral then sets a bit in the control register to initiate the DMA transaction.

The Master Read and Write Ports

The DMA controller reads data from the source address through the master read port, and then writes to the destination address through the master write port. You program the DMA controller using byte addresses. Read and write start addresses should be aligned to the transfer size. For example, to transfer data words, if the start address is 0, the address will increment to 4, 8, and 12. For heterogeneous systems where a number of different slave devices are of different widths, the data width for read and write masters matches the width of the widest data-width slave addressed by either the read or the write master. For bursting transfers, the burst length is set to the DMA transaction length with the appropriate unit conversion. For example, if a 32-bit data width DMA is programmed for a word transfer of 64 bytes, the length registered is programmed with 64 and the burst count port will be 16. If a 64-bit data width DMA is programmed for a doubleword transfer of 8 bytes, the length register is programmed with 8 and the burst count port will be 1.

There is a shallow FIFO buffer between the master read and write ports. The default depth is 2, which makes the write action depend on the data-available status of the FIFO, rather than on the status of the master read port.

Both the read and write master ports can perform Avalon transfers with flow control, which allows the slave peripheral to control the flow of data and terminate the DMA transaction.



For details about flow control in Avalon-MM data transfers and Avalon-MM peripherals, refer to *Avalon Interface Specifications*.

Addressing and Address Incrementing


When accessing memory, the read (or write) address increments by 1, 2, 4, 8, or 16 after each access, depending on the width of the data. On the other hand, a typical peripheral device (such as UART) has fixed register locations. In this case, the read/write address is held constant throughout the DMA transaction.

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, as shown in [Table 24-1](#).

Table 24-1. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16

 In systems with heterogeneous data widths, care must be taken to present the correct address or offset when configuring the DMA to access native-aligned slaves. For example, in a system using a 32-bit Nios II processor and a 16-bit DMA, the base address for the UART `txdata` register must be divided by the `dma_data_width/cpu_data_width—2` in this example.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the DMA controller in SOPC Builder to specify the core's configuration. Instantiating the DMA controller in SOPC Builder creates one slave port and two master ports. You must specify which slave peripherals can be accessed by the read and write master ports. Likewise, you must specify which other master peripheral(s) can access the DMA control port and initiate DMA transactions. The DMA controller does not export any signals to the top level of the system module.

DMA Parameters (Basic)

This section describes the parameters you can configure on the **DMA Parameters** page.

Transfer Size

The parameter **Width of the DMA Length Register** specifies the minimum width of the DMA's transaction length register, which can be between 1 and 32. The `length` register determines the maximum number of transfers possible in a single DMA transaction.

By default, the length register is wide enough to span any of the slave peripherals mastered by the read or write ports. Overriding the length register may be necessary if the DMA master port (read or write) masters only data peripherals, such as a UART. In this case, the address span of each slave is small, but a larger number of transfers may be desired per DMA transaction.

Burst Transactions

When **Enable Burst Transfers** is turned on, the DMA controller performs burst transactions on its master read and write ports. The parameter **Maximum Burst Size** determines the maximum burst size allowed in a transaction.

In burst mode, the length of a transaction must not be longer than the configured maximum burst size. Otherwise, the transaction must be performed as multiple transactions.

FIFO Implementation

This option determines the implementation of the FIFO buffer between the master read and write ports. Select **Construct FIFO from Registers** to implement the FIFO using one register per storage bit. This option has a strong impact on logic utilization when the DMA controller's data width is large. See [“Advanced Options” on page 24-5](#).

To implement the FIFO using embedded memory blocks available in the FPGA, select **Construct FIFO from Memory Blocks**.

Advanced Options

This section describes the parameters you can configure on the **Advanced Options** page.

Allowed Transactions

You can choose the transfer datawidth(s) supported by the DMA controller hardware. The following datawidth options can be enabled or disabled:

- Byte
- Halfword (two bytes)
- Word (four bytes)
- Doubleword (eight bytes)
- Quadword (sixteen bytes)

Disabling unnecessary transfer widths reduces the number of on-chip logic resources consumed by the DMA controller core. For example, if a system has both 16-bit and 32-bit memories, but the DMA controller transfers data to the 16-bit memory, 32-bit transfers could be disabled to conserve logic resources.

Device Support

The DMA Controller Core with Avalon Interface supports all Altera device families.

Software Programming Model

This section describes the programming model for the DMA controller, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the DMA controller core using the HAL API for DMA devices.

HAL System Library Support

The Altera-provided driver implements a HAL DMA device driver that integrates into the HAL system library for Nios II systems. HAL users should access the DMA controller via the familiar HAL API, rather than accessing the registers directly.



If your program uses the HAL device driver to access the DMA controller, accessing the device registers directly interferes with the correct behavior of the driver.

The HAL DMA driver provides both ends of the DMA process; the driver registers itself as both a receive channel (`alt_dma_rxchan`) and a transmit channel (`alt_dma_txchan`). The *Nios II Software Developer's Handbook* provides complete details of the HAL system library and the usage of DMA devices.

ioctl() Operations

`ioctl()` operation requests are defined for both the receive and transmit channels, which allows you to control the hardware-dependent aspects of the DMA controller. Two `ioctl()` functions are defined for the receiver driver and the transmitter driver: `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`. Table 24-2 lists the available operations. These are valid for both the transmit and receive channels.

Table 24-2. Operations for `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`

Request	Meaning
<code>ALT_DMA_SET_MODE_8</code>	Transfers data in units of 8 bits. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_SET_MODE_16</code>	Transfers data in units of 16 bits. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_SET_MODE_32</code>	Transfers data in units of 32 bits. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_SET_MODE_64</code>	Transfers data in units of 64 bits. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_SET_MODE_128</code>	Transfers data in units of 128 bits. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_RX_ONLY_ON (1)</code>	Sets a DMA receiver into streaming mode. In this case, data is read continuously from a single location. The parameter <code>arg</code> specifies the address to read from.
<code>ALT_DMA_RX_ONLY_OFF (1)</code>	Turns off streaming mode for a receive channel. The parameter <code>arg</code> is ignored.
<code>ALT_DMA_TX_ONLY_ON (1)</code>	Sets a DMA transmitter into streaming mode. In this case, data is written continuously to a single location. The parameter <code>arg</code> specifies the address to write to.
<code>ALT_DMA_TX_ONLY_OFF (1)</code>	Turns off streaming mode for a transmit channel. The parameter <code>arg</code> is ignored.

Note to Table 24-2:

- (1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

Limitations

Currently the Altera-provided drivers do not support 64-bit and 128-bit DMA transactions.

This function is not thread safe. If you want to access the DMA controller from more than one thread then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Software Files

The DMA controller is accompanied by the following software files. These files define the low-level interface to the hardware. Application developers should not modify these files.

- **`altera_avalon_dma_regs.h`**—This file defines the core’s register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **`altera_avalon_dma.h`, `altera_avalon_dma.c`**—These files implement the DMA controller’s device driver for the HAL system library.

Register Map

Programmers using the HAL API never access the DMA controller hardware directly via its registers. In general, the register map is only useful to programmers writing a device driver.



The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 24-3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 24-3. DMA Controller Register Map

Offset	Register Name	Read/Write	31	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	status (1)	RW	(2)										LEN	WEOP	REOP	BUSY	DONE
1	readaddress	RW	Read master start address														
2	writeaddress	RW	Write master start address														
3	length	RW	DMA transaction length (in bytes)														
4	—	—	Reserved (3)														
5	—	—	Reserved (3)														
6	control	RW	(2)	SOFTWARERESET	QUADWORD	DOUBLEWORD	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE	
7	—	—	Reserved (3)														

Notes to Table 24-3:

- (1) Writing zero to the `status` register clears the `LEN`, `WEOP`, `REOP`, and `DONE` bits.
- (2) These bits are reserved. Read values are undefined. Write zero.
- (3) This register is reserved. Read values are undefined. The result of a write is undefined.

status Register

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

The `status` register bits are shown in Table 24-4.

Table 24-4. status Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
0	DONE	R/C	A DMA transaction is complete. The <code>DONE</code> bit is set to 1 when an end of packet condition is detected or the specified transaction length is completed. Write zero to the <code>status</code> register to clear the <code>DONE</code> bit.
1	BUSY	R	The <code>BUSY</code> bit is 1 when a DMA transaction is in progress.

Table 24-4. status Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
2	REOP	R	The REOP bit is 1 when a transaction is completed due to an end-of-packet event on the read side.
3	WEOP	R	The WEOP bit is 1 when a transaction is completed due to an end of packet event on the write side.
4	LEN	R	The LEN bit is set to 1 when the length register decrements to zero.

readaddress Register

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

writeaddress Register

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

length Register

The length register specifies the number of bytes to be transferred from the read port to the write port. The length register is specified in bytes. For example, the value must be a multiple of 4 for word transfers, and a multiple of 2 for halfword transfers.

The length register is decremented as each data value is written by the write master port. When length reaches 0 the LEN bit is set. The length register does not decrement below 0.

The length register width is determined at system generation time. It is at least wide enough to span any of the slave ports mastered by the read or write master ports, and it can be made wider if necessary.

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. The control register bits determine which, if any, conditions of the DMA transaction result in the end of a transaction and an interrupt request.

The control register bits are shown in [Table 24-5](#).

Table 24-5. Control Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.

Table 24-5. Control Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0, the DMA is prevented from executing transfers. When the GO bit is set to 1 and the length register is non-zero, transfers occur.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a slave port with flow control on the read side may end the DMA transaction by asserting its end-of-packet signal.
6	WEEN	RW	Ends transaction on write-side end-of-packet. When the WEEN bit is set to 1, a slave port with flow control on the write side may end the DMA transaction by asserting its end-of-packet signal.
7	LEEN	RW	Ends transaction when the length register reaches zero. When the LEEN bit is 1, the DMA transaction ends when the length register reaches 0. When this bit is 0, length reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write master port.
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see “Addressing and Address Incrementing” on page 24-3 .
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see “Addressing and Address Incrementing” on page 24-3 .
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.
12	SOFTWARERESET	RW	Software can reset the DMA engine by writing this bit to 1 twice. Upon the second write of 1 to the SOFTWARERESET bit, the DMA control is reset identically to a system reset. The logic which sequences the software reset process then resets itself automatically.

The data width of DMA transactions is specified by the BYTE, HW, WORD, DOUBLEWORD, and QUADWORD bits. Only one of these bits can be set at a time. If more than one of the bits is set, the DMA controller behavior is undefined. The width of the transfer is determined by the narrower of the two slaves read and written. For example, a DMA transaction that reads from a 16-bit flash memory and writes to a 32-bit on-chip memory requires a halfword transfer. In this case, HW must be set to 1, and BYTE, WORD, DOUBLEWORD, and QUADWORD must be set to 0.

To successfully perform transactions of a specific width, that width must be enabled in hardware using the **Allowed Transaction** hardware option. For example, the DMA controller behavior is undefined if quadword transfers are disabled in hardware, but the QUADWORD bit is set during a DMA transaction.



Executing a DMA software reset when a DMA transfer is active may result in permanent bus lockup (until the next system reset). The `SOFTWARERESET` bit should therefore not be written except as a last resort.

Interrupt Behavior

The DMA controller has a single IRQ output that is asserted when the `status` register's `DONE` bit equals 1 and the control register's `I_EN` bit equals 1.

Writing the `status` register clears the `DONE` bit and acknowledges the IRQ. A master peripheral can read the `status` register and determine how the DMA transaction finished by checking the `LEN`, `REOP`, and `WEOP` bits.

Referenced Documents

This chapter references [Avalon Interface Specifications](#).

Document Revision History

[Table 24-6](#) shows the revision history for this chapter.

Table 24-6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	No change from previous release.	—
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the Functional Description of the core.	Updates made to comply with the Quartus II software version 8.0 release.



For previous versions of the *Quartus II Handbook*, refer to the [Quartus II Handbook Archive](#).