# 23. Scatter-Gather DMA Controller Core

QII55003-9.1.0

## Core Overview

The Scatter-Gather Direct Memory Access (SG-DMA) controller core implements high-speed data transfer between two components. You can use the SG-DMA controller core to transfer data from:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SG-DMA controller core transfers and merges non-contiguous memory to a continuous address space, and vice versa. The core reads a series of descriptors that specify the data to be transferred.

For applications requiring more than one DMA channel, multiple instantiations of the core can provide the required throughput. Each SG-DMA controller has its own series of descriptors specifying the data transfers. A single software module controls all of the DMA channels.

The SG-DMA controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the Hardware Abstraction Layer (HAL) system library, allowing software to access the core using the provided driver.

### Example Systems

Figure 23–1 shows a SG-DMA controller core in a block diagram for the DMA subsystem of a printed circuit board. The SG-DMA core in the FPGA reads streaming data from an internal streaming component and writes data to an external memory. A Nios II processor provides overall system control.

**Figure 23–1.** SG-DMA Controller Core with Streaming Peripheral and External Memory

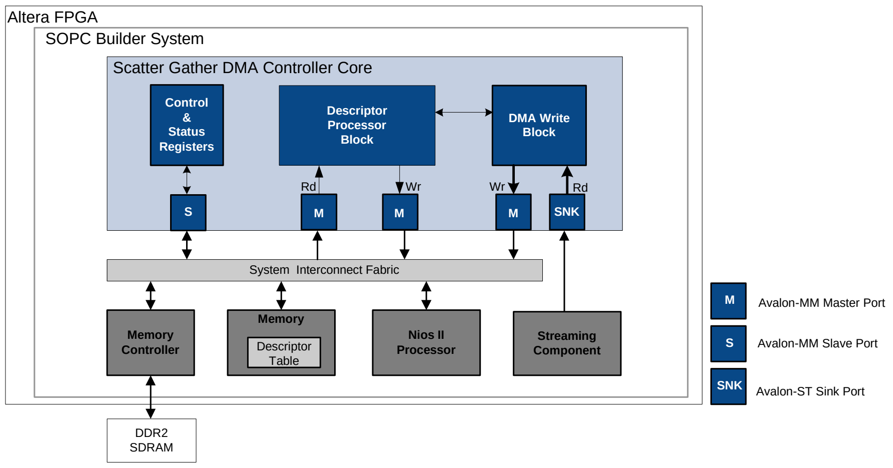
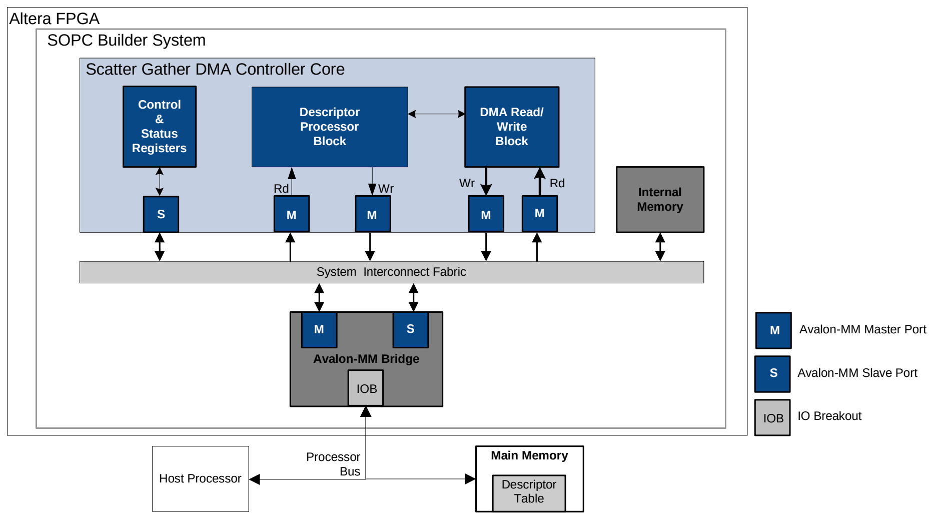


Figure 23–2 shows a different use of the SG-DMA controller core, where the core transfers data between an internal and external memory. The host processor and memory are connected to a system bus, typically either a PCI Express or Serial RapidIO™.

**Figure 23–2.** SG-DMA Controller Core with Internal and External Memory



## Comparison of SG-DMA Controller Core and DMA Controller Core

The SG-DMA controller core provides a significant performance enhancement over the previously available DMA controller core, which could only queue one transfer at a time. Using the DMA Controller core, a CPU had to wait for the transfer to complete before writing a new descriptor to the DMA slave port. Transfers to non-contiguous memory could not be linked; consequently, the CPU overhead was substantial for small transfers, degrading overall system performance. In contrast, the SG-DMA controller core reads a series of descriptors from memory that describe the required transactions and performs all of the transfers without additional intervention from the CPU.

## In This Chapter

This chapter contains the following sections:

- "Functional Description" on page 23–3
- "Device Support" on page 23–9
- "Instantiating the Core in SOPC Builder" on page 23–9
- "Simulation Considerations" on page 23–10
- "Software Programming Model" on page 23–10
- "Programming with SG-DMA Controller" on page 23–15

## Resource Usage and Performance

Resource utilization for the core is 600–1400 logic elements, depending upon the width of the datapath, the parameterization of the core, the device family, and the type of data transfer. Table 23–1 provides the estimated resource usage for a SG-DMA controller core used for memory to memory transfer. The core is configurable and the resource utilization varies with the configuration specified.

**Table 23–1.** SG-DMA Estimated Resource Usage

| Datapath | Cyclone® II | Stratix® (LEs) | Stratix II (ALUTs) |
|---|---|---|---|
| 8-bit datapath | 850 | 650 | 600 |
| 32-bit datapath | 1100 | 850 | 700 |
| 64-bit datapath | 1250 | 1250 | 800 |

The core operating frequency varies with the device and the size of the datapath. Table 23–2 provides an example of expected performance for SG-DMA cores instantiated in several different device families.

**Table 23–2.** SG-DMA Peak Performance

| Device | Datapath | $f_{MAX}$ | Throughput |
|---|---|---|---|
| Cyclone II | 64 bits | 150 MHz | 9.6 Gbps |
| Cyclone III | 64 bits | 160 MHz | 10.2 Gbps |
| Stratix II/Stratix II GX | 64 bits | 250 MHz | 16.0 Gbps |
| Stratix III | 64 bits | 300 MHz | 19.2 Gbps |

## Functional Description

The SG-DMA controller core comprises three major blocks: descriptor processor, DMA read, and DMA write. These blocks are combined to create three different configurations:

- Memory to memory
- Memory to stream
- Stream to memory

The type of devices you are transferring data to and from determines the configuration to implement. Examples of memory-mapped devices are PCI, PCIe and most memory devices. The Triple Speed Ethernet MAC, DSP MegaCore functions and many video IPs are examples of streaming devices. A recompilation is necessary each time you change the configuration of the SG-DMA controller core.

## Functional Blocks and Configurations

The following sections describe each functional block and configuration.

### Descriptor Processor

The descriptor processor reads descriptors from the descriptor list via its Avalon® Memory-Mapped (MM) read master port and pushes commands into the command FIFOs of the DMA read and write blocks. Each command includes the following fields to specify a transfer:

- Source address
- Destination address
- Number of bytes to transfer
- Increment read address after each transfer
- Increment write address after each transfer
- Generate start of packet (SOP) and end of packet (EOP)

After each command is processed by the DMA read or write block, a *status token* containing information about the transfer such as the number of bytes actually written is returned to the descriptor processor, where it is written to the respective fields in the descriptor.

### DMA Read Block

The DMA read block is used in memory-to-memory and memory-to-stream configurations. The block performs the following operations:

- Reads commands from the input command FIFO.
- Reads a block of memory via the Avalon-MM read master port for each command.
- Pushes data into the data FIFO.

If burst transfer is enabled, an internal read FIFO with a depth of twice the maximum read burst size is instantiated. The DMA read block initiates burst reads only when the read FIFO has sufficient space to buffer the complete burst.

### DMA Write Block

The DMA write block is used in memory-to-memory and stream-to-memory configurations. The block reads commands from its input command FIFO. For each command, the DMA write block reads data from its Avalon-ST sink port and writes it to the Avalon-MM master port.

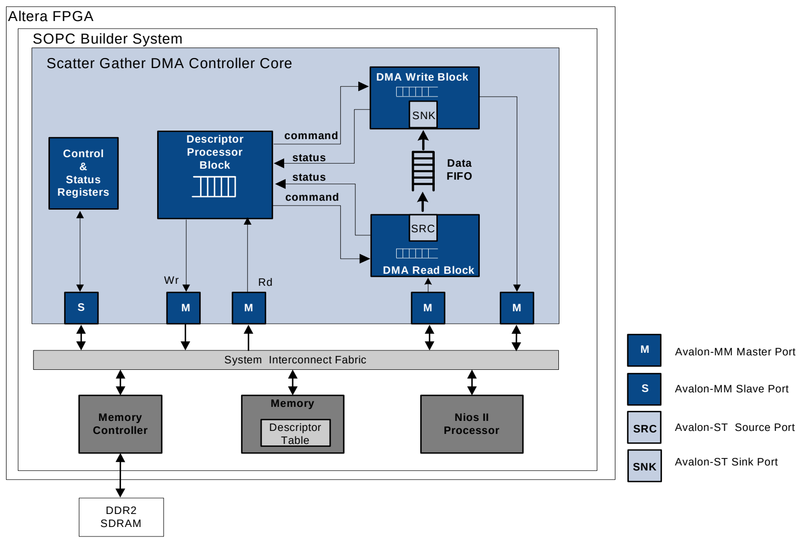
If burst transfer is enabled, an internal write FIFO with a depth of twice the maximum write burst size is instantiated. Each burst write transfers a fixed amount of data equals to the write burst size, except for the last burst. In the last burst, the remaining data is transferred even if the amount of data is less than the write burst size.

## Memory-to-Memory Configuration

Memory-to-memory configurations include all three blocks: descriptor processor, DMA read, and DMA write. An internal FIFO is also included to provide buffering and flow control for data transferred between the DMA read and write blocks.

Figure 23–3 illustrates one possible memory-to-memory configuration with an internal Nios II processor and descriptor list.

**Figure 23–3.** Example of Memory-to-Memory Configuration

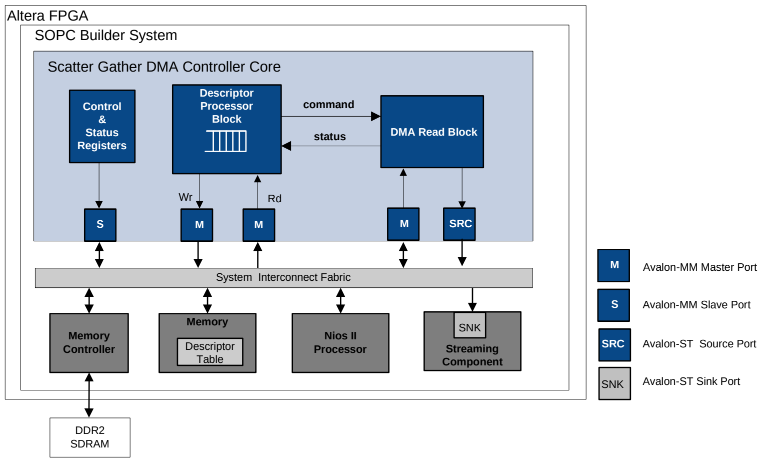


## Memory-to-Stream Configuration

Memory-to-stream configurations include the descriptor processor and DMA read blocks. Figure 23–4 illustrates a memory-to-stream configuration.

In this example, the Nios II processor and descriptor table are in the FPGA. Data from an external DDR2 SDRAM is read by the SG-DMA controller and written to an on-chip streaming peripheral.

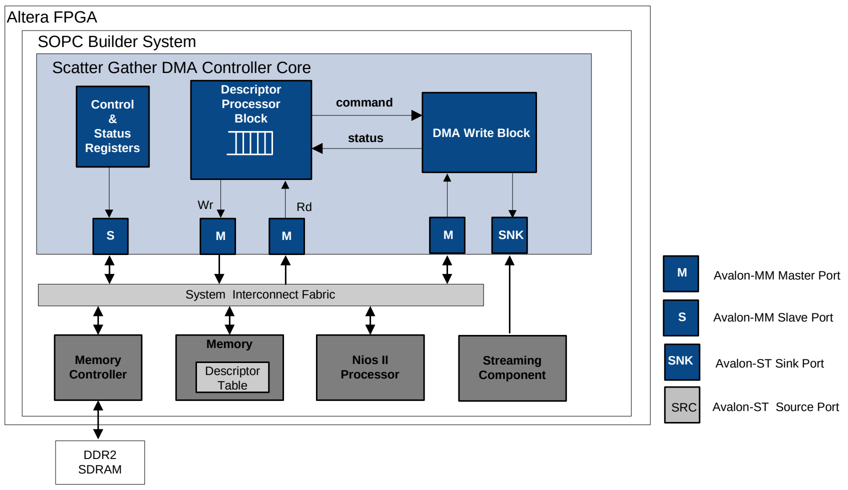
**Figure 23–4.** Example of Memory-to-Stream Configuration



### Stream-to-Memory Configuration

Stream-to-memory configurations include the descriptor processor and DMA write blocks. This configuration is similar to the memory-to-stream configuration as Figure 23–5 illustrates.

**Figure 23–5.** Example of Memory-to-Stream Configuration



## DMA Descriptors

DMA descriptors specify data transfers to be performed. The SG-DMA core uses a dedicated interface to read and write the descriptors. These descriptors, which are stored as a linked list, can be stored on an on-chip or off-chip memory and can be arbitrarily long.

Storing the descriptor list in an external memory frees up resources in the FPGA; however, an external descriptor list increases the overhead involved when the descriptor processor reads and updates the list. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows the core to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor access and processing overhead.

The descriptors must be initialized and aligned on a 32-bit boundary. The last descriptor in the list must have its `OWNED_BY_HW` bit set to 0 because the core relies on a cleared `OWNED_BY_HW` bit to stop processing.

See "DMA Descriptors" on page 23–13 for the structure of the DMA descriptor.

## Descriptor Processing

The following steps describe how the DMA descriptors are processed:

1. Software builds the descriptor linked list. See "Building and Updating Descriptor List" on page 23–8 for more information on how to build and update the descriptor linked list.
2. Software writes the address of the first descriptor to the `next_descriptor_pointer` register and initiates the transfer by setting the `RUN` bit in the `control` register to 1. See "Software Programming Model" on page 23–10 for more information on the registers.

   On the next clock cycle following the assertion of the `RUN` bit, the core sets the `BUSY` bit in the `status` register to 1 to indicate that descriptor processing is executing.
3. The descriptor processor block reads the address of the first descriptor from the `next_descriptor_pointer` register and pushes the retrieved descriptor into the command FIFO, which feeds commands to both the DMA read and write blocks. As soon as the first descriptor is read, the block reads the next descriptor and pushes it into the command FIFO. One descriptor is always read in advance thus maximizing throughput.
4. The core performs the data transfer.
   - In memory-to-memory configurations, the DMA read block receives the source address from its command FIFO and starts reading data to fill the FIFO on its stream port until the specified number of bytes are transferred. The DMA read block pauses when the FIFO is full until the FIFO has enough space to accept more data.

     The DMA write block gets the destination address from its command FIFO and starts writing until the specified number of bytes are transferred. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.
   - In memory-to-stream configurations, the DMA read block reads from the source address and transfers the data to the core's streaming port until the specified number of bytes are transferred or the end of packet is reached. The block uses the end-of-packet indicator for transfers with an unknown transfer size. For data transfers without using the end-of-packet indicator, the transfer size must be a multiple of the data width. Otherwise, the block requires extra logic and may impact the system performance.
   - In stream-to-memory configurations, the DMA write block reads from the core's streaming port and writes to the destination address. The block continues reading until the specified number of bytes are transferred.
5. The descriptor processor block receives a status from the DMA read or write block and updates the `DESC_CONTROL`, `DESC_STATUS`, and `ACTUAL_BYTES_TRANSFERRED` fields in the descriptor. The `OWNED_BY_HW` bit in the `DESC_CONTROL` field is cleared unless the `PARK` bit is set to 1.

Once the core starts processing the descriptors, software must not update descriptors with `OWNED_BY_HW` bit set to 1. It is only safe for software to update a descriptor when its `OWNED_BY_HW` bit is cleared.

The SG-DMA core continues processing the descriptors until an error condition occurs and the `STOP_DMA_ER` bit is set to 1, or a descriptor with a cleared `OWNED_BY_HW` bit is encountered.

### Building and Updating Descriptor List

Altera recommends the following method of building and updating the descriptor list:

1. Build the descriptor list and terminate the list with a non-hardware owned descriptor (`OWNED_BY_HW` = 0). The list can be arbitrarily long.
2. Set the interrupt `IE_CHAIN_COMPLETED`.
3. Write the address of the first descriptor in the first list to the `next_descriptor_pointer` register and set the `RUN` bit to 1 to initiate transfers.
4. While the core is processing the first list, build a second list of descriptors.
5. When the SD-DMA controller core finishes processing the first list, an interrupt is generated. Update the `next_descriptor_pointer` register with the address of the first descriptor in the second list. Clear the `RUN` bit and the `status` register. Set the `RUN` bit back to 1 to resume transfers.
6. If there are new descriptors to add, always add them to the list which the core is not processing. For example, if the core is processing the first list, add new descriptors to the second list and so forth.

This method ensures that the descriptors are not updated when the core is processing them. Because the method requires a response to the interrupt, a high-latency interrupt may cause a problem in systems where stalling data movement is not possible.

## Error Conditions

The SG-DMA core has a configurable error width. Error signals are connected directly to the Avalon-ST source or sink to which the SG-DMA core is connected.

The list below describes how the error signals in the SG-DMA core are implemented in the folowing configurations:

- Memory-to-memory configuration

  No error signals are generated. The error field in the register and descriptor is hardcoded to 0.

- Memory-to-stream configuration

  If you specified the usage of error bits in the core, the error bits are generated in the Avalon-ST source interface. These error bits are hardcoded to 0 and generated in compliance with the Avalon-ST slave interfaces.

- Stream-to-memory configuration

  If you specified the usage of error bits in the core, error bits are generated in the Avalon-ST sink interface. These error bits are passed from the Avalon-ST sink interface and stored in the registers and descriptor.

Table 23–3 lists the error signals when the core is operating in the memory-to-stream configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore® function.

**Table 23–3.** Avalon-ST Transmit Error Types

| Signal Type | Description |
|---|---|
| TSE_transmit_error[0] | Transmit Frame Error. Asserted to indicate that the transmitted frame should be viewed as invalid by the Ethernet MAC. The frame is then transferred onto the GMII interface with an error code during the frame transfer. |

Table 23–4 lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Triple-Speed Ethernet MegaCore function.

**Table 23–4.** Avalon-ST Receive Error Types

| Signal Type | Description |
|---|---|
| TSE_receive_error[0] | Receive Frame Error. This signal indicates that an error has occurred. It is the logical OR of receive errors 1 through 5. |
| TSE_receive_error[1] | Invalid Length Error. Asserted when the received frame has an invalid length as defined by the IEEE 802.3 standard. |
| TSE_receive_error[2] | CRC Error. Asserted when the frame has been received with a CRC-32 error. |
| TSE_receive_error[3] | Receive Frame Truncated. Asserted when the received frame has been truncated due to receive FIFO overflow. |
| TSE_receive_error[4] | Received Frame corrupted due to PHY error. (The PHY has asserted an error on the receive GMII interface.) |
| TSE_receive_error[5] | Collision Error. Asserted when the frame was received with a collision. |

Each streaming core has a different set of error codes. Refer to the respective user guides for the codes.

## Device Support

The SG-DMA Controller core supports all Altera device families.

## Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the SG-DMA Controller core in SOPC Builder to add the core to a system.

☞ The SG-DMA controller core should be given a higher priority (lower IRQ value) than most of the components in a system to ensure high throughput.

Table 23–5 lists and describes the parameters you can configure.

**Table 23–5.** Configurable Parameters

| Parameter | Legal Values | Description |
|---|---|---|
| **Transfer mode** | Memory To Memory<br>Memory To Stream<br>Stream To Memory | Configuration to use. For more information about these configurations, see "Memory-to-Memory Configuration" on page 23–5 |
| **Enable bursting on descriptor read master** | On/Off | If this option is on, the descriptor processor block uses Avalon-MM bursting when fetching descriptors and writing them back in memory. With 32-bit read and write ports, the descriptor processor block can fetch the 256-bit descriptor by performing 8-word burst as opposed to eight individual single-word transactions. |
| **Allow unaligned transfers** | On/Off | If this option is on, the core allows accesses to non-word-aligned addresses. This option doesn't apply for burst transfers.<br>Unaligned transfers require extra logic that may negatively impact system performance. |
| **Enable burst transfers** | On/Off | Turning on this option enables burst reads and writes. |
| **Read burstcount signal width** | 1–16 | The width of the read `burstcount` signal. This value determines the maximum burst read size. |
| **Write burstcount signal width** | 1–16 | The width of the write `burstcount` signal. This value determines the maximum burst write size. |
| **Data width** | 8, 16, 32, 64 | The data width in bits for the Avalon-MM read and write ports. |
| **Source error width** | 0–7 | The width of the `error` signal for the Avalon-ST source port. |
| **Sink error width** | 0 – 7 | The width of the `error` signal for the Avalon-ST sink port. |
| **Data transfer FIFO depth** | 2, 4, 8, 16, 32, 64 | The depth of the internal data FIFO in memory-to-memory configurations with burst transfers disabled. |

## Simulation Considerations

Signals for hardware simulation are automatically generated as part of the Nios II simulation process available in the Nios II IDE.

## Software Programming Model

The following sections describe the software programming model for the SG-DMA controller core.

### HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the SG-DMA controller core via the familiar HAL API and the ANSI C standard library.

### Software Files

The SG-DMA controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_sgdma_regs.h**—defines the core's register map, providing symbolic constants to access the low-level hardware

- **altera_avalon_sgdma.h**—provides definitions for the Altera Avalon SG-DMA buffer control and status flags.
- **altera_avalon_sgdma.c**—provides function definitions for the code that implements the SG-DMA controller core.
- **altera_avalon_sgdma_descriptor.h**—defines the core's descriptor, providing symbolic constants to access the low-level hardware.

## Register Maps

The SG-DMA controller core has three registers accessible from its Avalon-MM interface; `status`, `control` and `next_descriptor_pointer`. Software can configure the core and determines its current status by accessing the registers.

The `control/status` register has a 32-bit interface without byte-enable logic, and therefore cannot be properly accessed by a master with narrower data width than itself. To ensure correct operation of the core, always access the register with a master that is at least 32 bits wide.

Table 23–6 lists and describes the registers.

**Table 23–6.** Register Map

| 32-bit Word Offset | Register Name | Reset Value | Description |
|---|---|---|---|
| base + 0 | `status` | 0 | This register indicates the core’s current status such as what caused the last interrupt and if the core is still processing descriptors. See Table 23–4 on page 23–9 for the `status` register map. |
| base + 4 | `control` | 0 | This register specifies the core’s behavior such as what triggers an interrupt and when the core is started and stopped. The host processor can configure the core by setting the register bits accordingly. See Table 23–4 on page 23–9 for the `control` register map. |
| base + 8 | `next_descriptor_pointer` | 0 | This register contains the address of the next descriptor to process. Set this register to the address of the first descriptor as part of the system initialization sequence.<br>Altera recommends that user applications clear the `RUN` bit in the `control` register and wait until the `BUSY` bit of the `status` register is set to 0 before reading this register. |

Table 23–7 provides a bit map for the `control` register.

**Table 23–7.** Control Register Bit Map (Part 1 of 2)

| Bit | Bit Name | Access | Description |
|---|---|---|---|
| 0 | `IE_ERROR` | R/W | When this bit is set to 1, the core generates an interrupt if an Avalon-ST error occurs during descriptor processing. *(1)* |
| 1 | `IE_EOP_ENCOUNTERED` | R/W | When this bit is set to 1, the core generates an interrupt if an EOP is encountered during descriptor processing. *(1)* |
| 2 | `IE_DESCRIPTOR_COMPLETED` | R/W | When this bit is set to 1, the core generates an interrupt after each descriptor is processed. *(1)* |

**Table 23–7.** Control Register Bit Map (Part 2 of 2)

| Bit | Bit Name | Access | Description |
|---|---|---|---|
| 3 | IE_CHAIN_COMPLETED | R/W | When this bit is set to 1, the core generates an interrupt after the last descriptor in the list is processed, that is when the core encounters a descriptor with a cleared OWNED_BY_HW bit. *(1)* |
| 4 | IE_GLOBAL | R/W | Global signal to enable all interrupts. |
| 5 | RUN | R/W | Set this bit to 1 to start the descriptor processor block which subsequently initiates DMA transactions. Prior to setting this bit to 1, ensure that the register next_descriptor_pointer is updated with the address of the first descriptor to process. The core continues to process descriptors in its queue as long as this bit is 1.<br>Clear this bit to stop the core from processing the next descriptor in its queue. If this bit is cleared in the middle of processing a descriptor, the core completes the processing before stopping. The host processor can then modify the remaining descriptors and restart the core. |
| 6 | STOP_DMA_ER | R/W | Set this bit to 1 to stop the core when an Avalon-ST error is encountered during a DMA transaction. This bit applies only to stream-to-memory configurations. |
| 7 | IE_MAX_DESC_PROCESSED | R/W | Set this bit to 1 to generate an interrupt after the number of descriptors specified by MAX_DESC_PROCESSED are processed. |
| 8 .. 15 | MAX_DESC_PROCESSED | R/W | Specifies the number of descriptors to process before the core generates an interrupt. |
| 16 | SW_RESET | R/W | Software can reset the core by writing to this bit twice. Upon the second write, the core is reset. The logic which sequences the software reset process then resets itself automatically.<br>Executing a software reset when a DMA transfer is active may result in permanent bus lockup until the next system reset. Hence, Altera recommends that you use the software reset as your last resort. |
| 17 | PARK | R/W | Seting this bit to 0 causes the SG-DMA controller core to clear the OWNED_BY_HW bit in the descriptor after each descriptor is processed. If the PARK bit is set to 1, the core does not clear the OWNED_BY_HW bit, thus allowing the same descriptor to be processed repeatedly without software intervention. You also need to set the last descriptor in the list to point to the first one. |
| 18..30 | Reserved | | |
| 31 | CLEAR_INTERRUPT | R/W | Set this bit to 1 to clear pending interrupts. |

**Note to Table 23–11:**

(1) All interrupts are generated only after the descriptor is updated.

Table 23–8 provides a bit map for the status register. Altera recommends that you read the status register only after the RUN bit in the control register is cleared.

**Table 23–8.** Status Register Bit Map

| Bit | Bit Name | Access | Description |
|---|---|---|---|
| 0 | ERROR | R/C *(1) (2)* | A value of 1 indicates that an Avalon-ST error was encountered during a transfer. |
| 1 | EOP_ENCOUNTERED | R/C | A value of 1 indicates that the transfer was terminated by an end-of-packet (EOP) signal generated on the Avalon-ST source interface. This condition is only possible in stream-to-memory configurations. |
| 2 | DESCRIPTOR_COMPLETED | R/C *(1) (2)* | A value of 1 indicates that a descriptor was processed to completion. |
| 3 | CHAIN_COMPLETED | R/C *(1) (2)* | A value of 1 indicates that the core has completed processing the descriptor chain. |
| 4 | BUSY | R *(1) (3)* | A value of 1 indicates that descriptors are being processed. This bit is set to 1 on the next clock cycle after the RUN bit is asserted and does not get cleared until one of the following event occurs: ■ Descriptor processing completes and the RUN bit is cleared. ■ An error condition occurs, the STOP_DMA_ER bit is set to 1 and the processing of the current descriptor completes. |
| 5 .. 31 | Reserved | | |

**Notes to Table 23–8:**

(1) This bit must be cleared after a read is performed. Write one to clear this bit.

(2) This bit is updated by hardware after each DMA transfer completes. It remains set until software writes one to clear.

(3) This bit is continuously updated by the hardware.

## DMA Descriptors

Table 23–9 shows the structure a DMA descriptor entry. See "Data Structure" on page 23–15 for the structure definition.

**Table 23–9.** DMA Descriptor Structure

| Byte Offset | Field Names | | | |
|---|---|---|---|---|
| | 31 24 | 23 16 | 15 8 | 7 0 |
| base | source | | | |
| base + 4 | Reserved | | | |
| base + 8 | destination | | | |
| base + 12 | Reserved | | | |
| base + 16 | next_desc_ptr | | | |
| base + 20 | Reserved | | | |
| base + 24 | Reserved | | bytes_to_transfer | |
| base + 28 | desc_control | desc_status | actual_bytes_transferred | |

Table 23–10 describes the each field in a descriptor entry.

**Table 23–10.** DMA Descriptor Field Description

| Field Name | Access | Description |
|---|---|---|
| `source` | R/W | Specifies the address of data to be read. This address is set to 0 if the input interface is an Avalon-ST interface. |
| `destination` | R/W | Specifies the address to which data should be written. This address is set to 0 if the output interface is an Avalon-ST interface. |
| `next_desc_ptr` | R/W | Specifies the address of the next descriptor in the linked list. |
| `bytes_to_transfer` | R/W | Specifies the number of bytes to transfer. If this field is 0, the SG-DMA controller core continues transferring data until it encounters an EOP. |
| `actual_bytes_transferred` | R | Specifies the number of bytes that are successfully transferred by the core. This field is updated after the core processes a descriptor. |
| `desc_status` | R/W | This field is updated after the core processes a descriptor. See Table 23–12 on page 23–15 for the bit map of this field. |
| `desc_control` | R/W | Specifies the behavior of the core. This field is updated after the core processes a descriptor. See Table 23–11 on page 23–14 for descriptions of each bit. |

Table 23–1 provides a bit map for the `desc_control` field.

**Table 23–11.** DESC_CONTROL Bit Map

| Bit (s) | Field Name | Access | Description |
|---|---|---|---|
| 0 | `GENERATE_EOP` | W | When this bit is set to 1,the DMA read block asserts the EOP signal on the final word. |
| 1 | `READ_FIXED_ADDRESS` | R/W | This bit applies only to Avalon-MM read master ports. When this bit is set to 1, the DMA read block does not increment the memory address. When this bit is set to 0, the read address increments after each read. |
| 2 | `WRITE_FIXED_ADDRESS` | R/W | This bit applies only to Avalon-MM write master ports. When this bit is set to 1, the DMA write block does not increment the memory address. When this bit is set to 0, the write address increments after each write.<br>In memory-to-stream configurations, the DMA read block generates a start-of-packet (SOP) on the first word when this bit is set to 1. |
| [6:3] | Reserved | — | — |
| 7 | `OWNED_BY_HW` | R/W | This bit determines whether hardware or software has write access to the current register.<br>When this bit is set to 1, the core can update the descriptor and software should not access the descriptor due to the possibility of race conditions. Otherwise, it is safe for software to update the descriptor. |

After completing a DMA transaction, the descriptor processor block updates the `desc_status` field to indicate how the transaction proceeded. Table 23–1 provides the bit map of this field.

**Table 23–12.** DESC_STATUS Bit Map

| Bit | Bit Name | Access | Description |
|---|---|---|---|
| [7:0] | `ERROR_0` .. `ERROR_7` | R | Each bit represents an error that occurred on the Avalon-ST interface. The context of each error is defined by the component connected to the Avalon-ST interface. |

## Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

## Programming with SG-DMA Controller

This section describes the device and descriptor data structures, and the application programming interface (API) for the SG-DMA controller core.

### Data Structure

Figure 23–6 shows the data structure for the device.

**Figure 23–6.** Device Data Structure

```
typedef struct alt_sgdma_dev
{
  alt_llist                 llist;               // Device linked-list entry
  const char                *name;               // Name of SGDMA in SOPC System
  void                      *base;               // Base address of SGDMA
  alt_u32                   *descriptor_base;    // reserved
  alt_u32                   next_index;          // reserved
  alt_u32                   num_descriptors;     // reserved
  alt_sgdma_descriptor      *current_descriptor; // reserved
  alt_sgdma_descriptor      *next_descriptor;    // reserved
  alt_avalon_sgdma_callback callback;            // Callback routine pointer
  void                      *callback_context;   // Callback context pointer
  alt_u32                   chain_control;       // Value OR'd into control reg
} alt_sgdma_dev;
```

Figure 23–7 shows the data structure for the descriptors.

**Figure 23–7.** Descriptor Data Structure

```
typedef struct {
    alt_u32   *read_addr;
    alt_u32   read_addr_pad;

    alt_u32   *write_addr;
    alt_u32   write_addr_pad;

    alt_u32   *next;
    alt_u32   next_pad;

    alt_u16   bytes_to_transfer;
    alt_u8    read_burst; /* Reserved field. Set to 0. */
    alt_u8    write_burst;/* Reserved field. Set to 0. */

    alt_u16   actual_bytes_transferred;
    alt_u8    status;
    alt_u8    control;

} alt_avalon_sgdma_packed alt_sgdma_descriptor;
```

## SG-DMA API

Table 23–13 lists all functions provided and briefly describes each.

**Table 23–13.** Function List

| Name | Description |
|---|---|
| `alt_avalon_sgdma_do_async_transfer()` | Starts a non-blocking transfer of a descriptor chain. |
| `alt_avalon_sgdma_do_sync_transfer()` | Starts a blocking transfer of a descriptor chain. This function blocks both before transfer if the controller is busy and until the requested transfer has completed. |
| `alt_avalon_sgdma_construct_mem_to_mem_desc()` | Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer. |
| `alt_avalon_sgdma_construct_stream_to_mem_desc()` | Constructs a single SG-DMA descriptor in the specified memory for an Avalon-ST to Avalon-MM transfer. The function automatically terminates the descriptor chain with a NULL descriptor. |
| `alt_avalon_sgdma_construct_mem_to_stream_desc()` | Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer. |
| `alt_avalon_sgdma_check_descriptor_status()` | Reads the status of a given descriptor. |
| `alt_avalon_sgdma_register_callback()` | Associates a user-specific callback routine with the SG-DMA interrupt handler. |
| `alt_avalon_sgdma_start()` | Starts the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()` and `alt_avalon_sgdma_do_sync_transfer()` are used . |
| `alt_avalon_sgdma_stop()` | Stops the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()` and `alt_avalon_sgdma_do_sync_transfer()` are used . |
| `alt_avalon_sgdma_open()` | Returns a pointer to the SG-DMA controller with the given name. |

## alt_avalon_sgdma_do_async_transfer()

| | |
|---|---|
| **Prototype:** | int alt_avalon_do_async_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc) |
| **Thread-safe:** | No. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>,<br><**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*dev`—a pointer to an SG-DMA device structure.<br>`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its “next” descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain. |
| **Returns:** | Returns 0 success. Other return codes are defined in **errno.h**. |
| **Description:** | Set up and begin a non-blocking transfer of one or more descriptors or a descriptor chain. If the SG-DMA controller is busy at the time of this call, the routine immediately returns EBUSY; the application can then decide how to proceed without being blocked. If a callback routine has been previously registered with this particular SG-DMA controller, the transfer is set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and the application developer must check for and handle errors and completion. The run bit is cleared before the begining of the transfer and is set to 1 to restart a new descriptor chain. |

## alt_avalon_sgdma_do_sync_transfer()

| | |
|---|---|
| **Prototype:** | alt_u8 alt_avalon_sgdma_do_sync_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc) |
| **Thread-safe:** | No. |
| **Available from ISR:** | Not recommended. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>,<br><**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*dev`—a pointer to an SG-DMA device structure.<br>`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its “next” descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain. |
| **Returns:** | Returns the contents of the `status` register. |
| **Description:** | Sends a fully formed descriptor or list of descriptors to the SG-DMA controller for transfer. This function blocks both before transfer, if the SG-DMA controller is busy, and until the requested transfer has completed. If an error is detected during the transfer, it is abandoned and the controller’s `status` register contents are returned to the caller. Additional error information is available in the status bits of each descriptor that the SG-DMA processed. The user application searches through the descriptor or list of descriptors to gather specific error information. The run bit is cleared before the begining of the transfer and is set to 1 to restart a new descriptor chain. |

## alt_avalon_sgdma_construct_mem_to_mem_desc()

| | |
|---|---|
| Prototype: | void alt_avalon_sgdma_construct_mem_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u32 *write_addr, alt_u16 length, int read_fixed, int write_fixed) |
| Thread-safe: | Yes. |
| Available from ISR: | Yes. |
| Include: | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>**,** <**altera_avalon_sgdma_regs.h**> |
| Parameters: | `*desc`—a pointer to the descriptor being constructed.<br>`*next`—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.<br>`*read_addr`—the first read address for the SG-DMA transfer.<br>`*write_addr`—the first write address for the SG-DMA transfer.<br>`length`—the number of bytes for the transfer.<br>`read_fixed`—if non-zero, the SG-DMA reads from a fixed address.<br>`write_fixed`—if non-zero, the SG-DMA writes to a fixed address. |
| Returns: | void |
| Description: | This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-MM to Avalon-MM transfer. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.<br>The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.<br>You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain.<br>Descriptors must be in a memory device mastered by the SG-DMA controller’s chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller. |

## alt_avalon_sgdma_construct_stream_to_mem_desc()

| | |
|---|---|
| **Prototype:** | void alt_avalon_sgdma_construct_stream_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *write_addr, alt_u16 length_or_eop, int write_fixed) |
| **Thread-safe:** | Yes. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*desc`—a pointer to the descriptor being constructed.<br>`*next`—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.<br>`*write_addr`—the first write address for the SG-DMA transfer.<br>`length_or_eop`—the number of bytes for the transfer. If set to zero (0x0), the transfer continues until an EOP signal is received from the Avalon-ST interface.<br>`write_fixed`—if non-zero, the SG-DMA will write to a fixed address. |
| **Returns:** | void |
| **Description:** | This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-ST to Avalon-MM transfer. The source (read) data for the transfer comes from the Avalon-ST interface connected to the SG-DMA controller's streaming read port.<br>The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.<br>The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.<br>You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain.<br>Descriptors must be in a memory device mastered by the SG-DMA controller’s chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller. |

## alt_avalon_sgdma_construct_mem_to_stream_desc()

| | |
|---|---|
| **Prototype:** | void alt_avalon_sgdma_construct_mem_to_stream_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u16 length, int read_fixed, int generate_sop, int generate_eop, alt_u8 atlantic_channel) |
| **Thread-safe:** | Yes. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*desc`—a pointer to the descriptor being constructed.<br>`*next`—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.<br>`*read_addr`—the first read address for the SG-DMA transfer.<br>`length`—the number of bytes for the transfer.<br>`read_fixed`—if non-zero, the SG-DMA reads from a fixed address.<br>`generate_sop`—if non-zero, the SG-DMA generates a SOP on the Avalon-ST interface when commencing the transfer.<br>`generate_eop`—if non-zero, the SG-DMA generates an EOP on the Avalon-ST interface when completing the transfer.<br>`atlantic_channel`—an 8-bit Avalon-ST channel number. Channels are currently not supported. Set this parameter to 0. |
| **Returns:** | void |
| **Description:** | This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma-descriptor *desc` for an Avalon-MM to Avalon-ST transfer. The destination (write) data for the transfer goes to the Avalon-ST interface connected to the SG-DMA controller's streaming write port. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.<br>The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.<br>You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller’s chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller. |

## alt_avalon_sgdma_check_descriptor_status()

| | |
|---|---|
| **Prototype:** | int alt_avalon_sgdma_check_descriptor_status(alt_sgdma_descriptor *desc) |
| **Thread-safe:** | Yes. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*desc`—a pointer to the constructed descriptor to examine. |
| **Returns:** | Returns 0 if the descriptor is error-free, not owned by hardware, or a previously requested transfer completed normally. Other return codes are defined in **errno.h**. |
| **Description:** | Checks a descriptor previously owned by hardware for any errors reported in a previous transfer. The routine reports: errors reported by the SG-DMA controller, the buffer in use. |

## alt_avalon_sgdma_register_callback()

| | |
|---|---|
| **Prototype:** | void alt_avalon_sgdma_register_callback(alt_sgdma_dev *dev, alt_avalon_sgdma_callback callback, alt_u16 chain_control, void *context) |
| **Thread-safe:** | Yes. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*dev`—a pointer to the SG-DMA device structure.<br>`callback`—a pointer to the callback routine to execute at interrupt level.<br>`chain_control`—the SG-DMA control register contents.<br>`*context`—a pointer used to pass context-specific information to the ISR. `context` can point to any ISR-specific information. |
| **Returns:** | void |
| **Description:** | Associates a user-specific routine with the SG-DMA interrupt handler. If a callback is registered, all non-blocking transfers enables interrupts that causes the callback to be executed. The callback runs as part of the interrupt service routine, and care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the *Nios II Software Developer's Handbook*.<br>To disable callbacks after registering one, call this routine with 0x0 as the callback argument. |

## alt_avalon_sgdma_start()

| | |
|---|---|
| **Prototype:** | void alt_avalon_sgdma_start(alt_sgdma_dev *dev) |
| **Thread-safe:** | No. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*dev`—a pointer to the SG-DMA device structure. |
| **Returns:** | void |
| **Description:** | Starts the DMA engine and processes the descriptor pointed to in the controller's next descriptor pointer and all subsequent descriptors in the chain. It is not necessary to call this function when `do_sync` or `do_async` is used. |

## alt_avalon_sgdma_stop()

| | |
|---|---|
| **Prototype:** | void alt_avalon_sgdma_stop(alt_sgdma_dev *dev) |
| **Thread-safe:** | No. |
| **Available from ISR:** | Yes. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `*dev`—a pointer to the SG-DMA device structure. |
| **Returns:** | void |
| **Description:** | Stops the DMA engine following completion of the current buffer descriptor. It is not necessary to call this function when `do_sync` or `do_async` is used. |

## alt_avalon_sgdma_open()

| | |
|---|---|
| **Prototype:** | alt_sgdma_dev* alt_avalon_sgdma_open(const char* name) |
| **Thread-safe:** | Yes. |
| **Available from ISR:** | No. |
| **Include:** | <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**> |
| **Parameters:** | `name`—the name of the SG-DMA device to open. |
| **Returns:** | A pointer to the SG-DMA device structure associated with the supplied name, or NULL if no corresponding SG-DMA device structure was found. |
| **Description:** | Retrieves a pointer to a hardware SG-DMA device structure. |

## Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

## Document Revision History

Table 23–14 shows the revision history for this chapter.

**Table 23–14.** Revision History

| Date and Document Version | Changes Made | Summary of Changes |
|---|---|---|
| November 2009<br>v9.1.0 | ■ Revised descriptions of register fields and bits.<br>■ Added description to the memory-to-stream configurations.<br>■ Added descriptions to alt_avalon_sgdma_do_sync_transfer() and alt_avalon_sgdma_do_async_transfer() API.<br>■ Added a list on error signals implementation. | — |
| March 2009<br>v9.0.0 | Added description of **Enable bursting on descriptor read master**. | — |
| November 2008<br>v8.1.0 | ■ Changed to 8-1/2 x 11 page size.<br>■ Added section DMA Descriptors in Functional Specifications<br>■ Revised descriptions of register fields and bits.<br>■ Reorganized sections Software Programming Model and Programming with SG-DMA Controller Core. | — |
| May 2008<br>v8.0.0 | ■ Added sections on burst transfers. | 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.