

This section describes multiprocessor coordination peripherals provided by Altera® for SOPC Builder systems. These components provide reliable mechanisms for multiple Nios® II processors to communicate with each other, and coordinate operations.

This section includes the following chapters:

- Chapter 23, Scatter-Gather DMA Controller Core
- Chapter 24, DMA Controller Core
- Chapter 25, Video Sync Generator and Pixel Converter Cores
- Chapter 26, Interval Timer Core
- Chapter 27, Mutex Core
- Chapter 28, Mailbox Core
- Chapter 29, Vectored Interrupt Controller Core



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

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

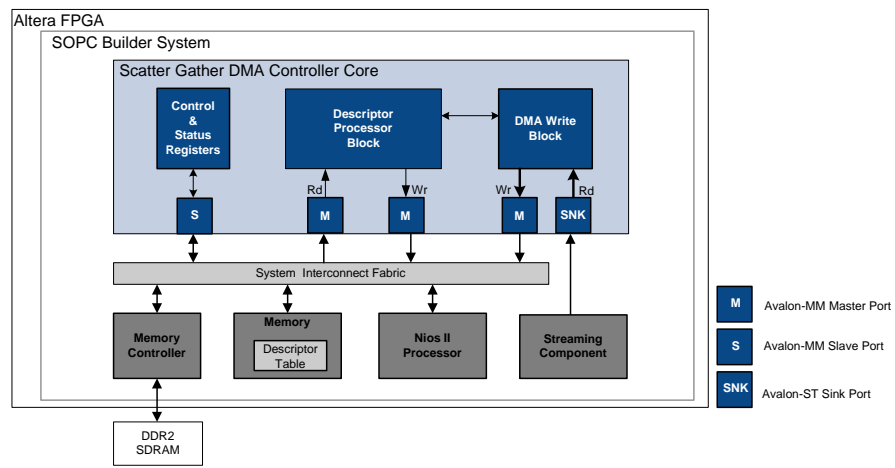
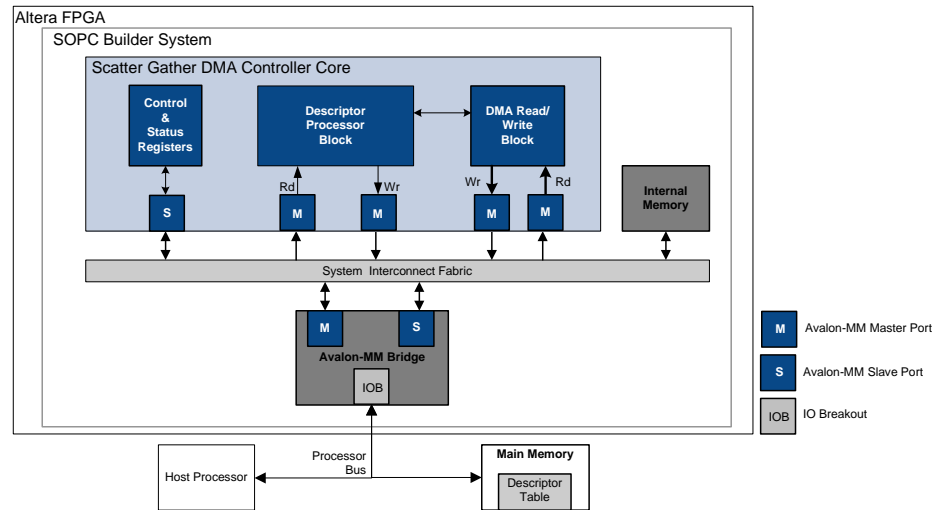


Figure 23-2 shows a different use of the SG-DMA controller core, where the core transfers data between an internal and external memory. The host processor and memory are connected to a system bus, typically either a PCI Express or Serial RapidIO™.

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.

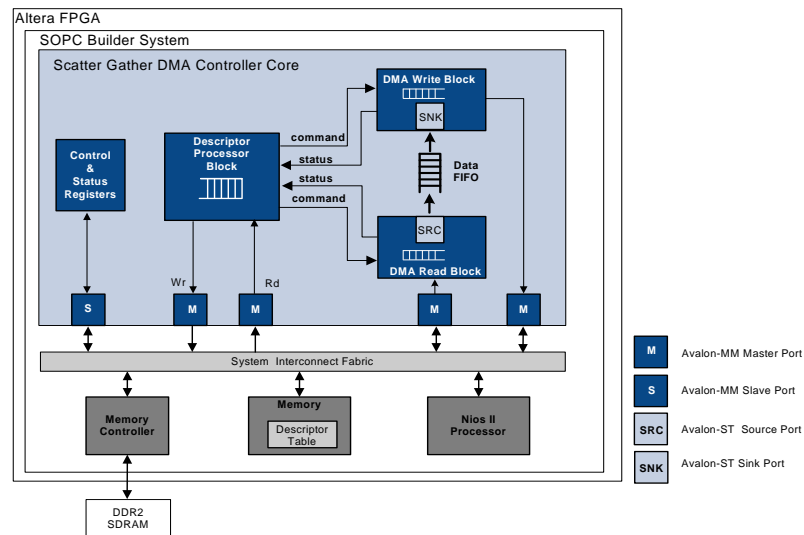
If burst transfer is enabled, an internal write FIFO with a depth of twice the maximum write burst size is instantiated. Each burst write transfers a fixed amount of data equals to the write burst size, except for the last burst. In the last burst, the remaining data is transferred even if the amount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

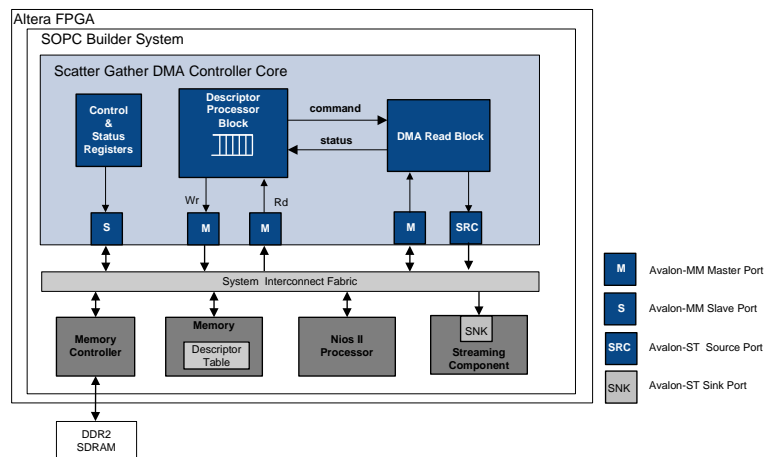


Memory-to-Stream Configuration

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.

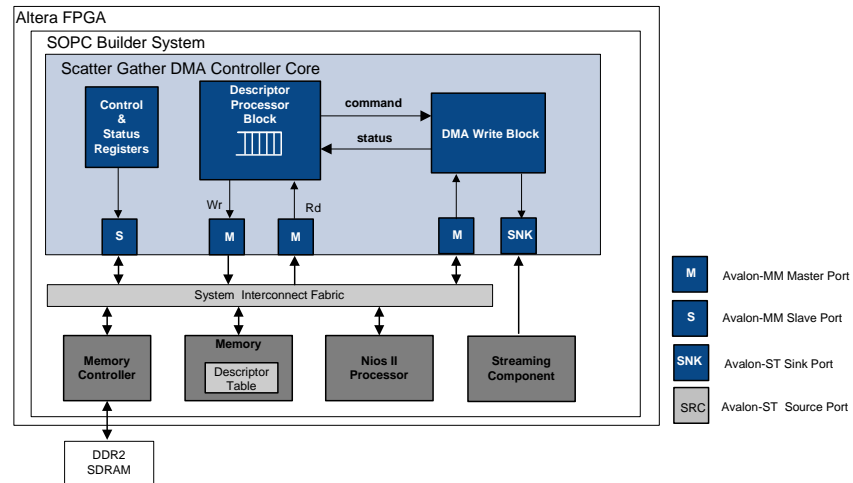
Figure 23-4. Example of Memory-to-Stream Configuration



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 following 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 Memory To Stream 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. 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 <code>burstcount</code> signal. This value determines the maximum burst read size.
Write burstcount signal width	1-16	The width of the write <code>burstcount</code> 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 <code>error</code> signal for the Avalon-ST source port.
Sink error width	0-7	The width of the <code>error</code> 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	<code>status</code>	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 <code>status</code> register map.
base + 4	<code>control</code>	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 <code>control</code> register map.
base + 8	<code>next_descriptor_pointer</code>	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. Altera recommends that user applications clear the <code>RUN</code> bit in the <code>control</code> register and wait until the <code>BUSY</code> bit of the <code>status</code> 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	<code>IE_ERROR</code>	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	<code>IE_EOP_ENCOUNTERED</code>	R/W	When this bit is set to 1, the core generates an interrupt if an EOP is encountered during descriptor processing. (1)
2	<code>IE_DESCRIPTOR_COMPLETED</code>	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 <code>next_descriptor_pointer</code> 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. 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 <code>MAX_DESC_PROCESSED</code> 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. 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	Setting 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: <ul style="list-style-type: none"> ■ 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
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. 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. 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
<code>alt_avalon_sgdma_do_async_transfer()</code>	Starts a non-blocking transfer of a descriptor chain.
<code>alt_avalon_sgdma_do_sync_transfer()</code>	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.
<code>alt_avalon_sgdma_construct_mem_to_mem_desc()</code>	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer.
<code>alt_avalon_sgdma_construct_stream_to_mem_desc()</code>	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.
<code>alt_avalon_sgdma_construct_mem_to_stream_desc()</code>	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer.
<code>alt_avalon_sgdma_check_descriptor_status()</code>	Reads the status of a given descriptor.
<code>alt_avalon_sgdma_register_callback()</code>	Associates a user-specific callback routine with the SG-DMA interrupt handler.
<code>alt_avalon_sgdma_start()</code>	Starts the DMA engine. This is not required when <code>alt_avalon_sgdma_do_async_transfer()</code> and <code>alt_avalon_sgdma_do_sync_transfer()</code> are used.
<code>alt_avalon_sgdma_stop()</code>	Stops the DMA engine. This is not required when <code>alt_avalon_sgdma_do_async_transfer()</code> and <code>alt_avalon_sgdma_do_sync_transfer()</code> are used.
<code>alt_avalon_sgdma_open()</code>	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>, <altera_avalon_sgdma_regs.h>

Parameters: *dev—a pointer to an SG-DMA device structure.
*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 beginning 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>, <altera_avalon_sgdma_regs.h>

Parameters: *dev—a pointer to an SG-DMA device structure.
*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 beginning 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.
- *next—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
- *read_addr—the first read address for the SG-DMA transfer.
- *write_addr—the first write address for the SG-DMA transfer.
- length—the number of bytes for the transfer.
- read_fixed—if non-zero, the SG-DMA reads from a fixed address.
- 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.

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.

You must properly allocate 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_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.
- *next—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
- *write_addr—the first write address for the SG-DMA transfer.
- 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.
- 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.

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.

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.

You must properly allocate 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_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.
 - *next—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
 - *read_addr—the first read address for the SG-DMA transfer.
 - length—the number of bytes for the transfer.
 - read_fixed—if non-zero, the SG-DMA reads from a fixed address.
 - generate_sop—if non-zero, the SG-DMA generates a SOP on the Avalon-ST interface when commencing the transfer.
 - generate_eop—if non-zero, the SG-DMA generates an EOP on the Avalon-ST interface when completing the transfer.
 - 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.
- 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.
- 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. callback—a pointer to the callback routine to execute at interrupt level. chain_control—the SG-DMA control register contents. *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 <i>Nios II Software Developer's Handbook</i> . 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 v9.1.0	<ul style="list-style-type: none"> ■ Revised descriptions of register fields and bits. ■ Added description to the memory-to-stream configurations. ■ Added descriptions to <code>alt_avalon_sgdma_do_sync_transfer()</code> and <code>alt_avalon_sgdma_do_async_transfer()</code> API. ■ Added a list on error signals implementation. 	—
March 2009 v9.0.0	Added description of Enable bursting on descriptor read master .	—
November 2008 v8.1.0	<ul style="list-style-type: none"> ■ Changed to 8-1/2 x 11 page size. ■ Added section DMA Descriptors in Functional Specifications ■ Revised descriptions of register fields and bits. ■ Reorganized sections Software Programming Model and Programming with SG-DMA Controller Core. 	—
May 2008 v8.0.0	<ul style="list-style-type: none"> ■ 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](#).

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:

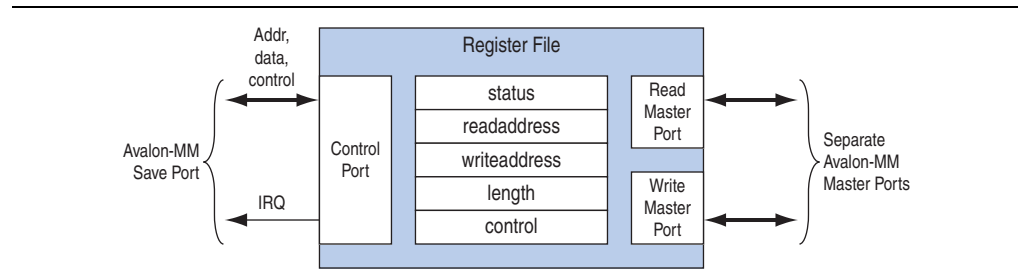
- [“Functional Description”](#)
- [“Instantiating the Core in SOPC Builder” on page 24–4](#)
- [“Device Support” on page 24–5](#)
- [“Software Programming Model” on page 24–5](#)

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](#).

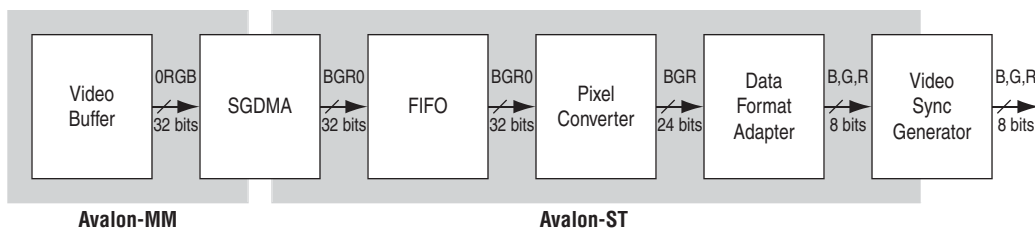
Core Overview

The video sync generator core accepts a continuous stream of pixel data in RGB format, and outputs the data to an off-chip display controller with proper timing. You can configure the video sync generator core to support different display resolutions and synchronization timings.

The pixel converter core transforms the pixel data to the format required by the video sync generator. [Figure 25–1](#) shows a typical placement of the video sync generator and pixel converter cores in a system.

In this example, the video buffer stores the pixel data in 32-bit unpacked format. The extra byte in the pixel data is discarded by the pixel converter core before the data is serialized and sent to the video sync generator core.

Figure 25–1. Typical Placement in a System



The video sync generator and pixel converter cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

These cores are deployed in the Nios II Embedded Software Evaluation Kit (NEEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

This chapter contains the following sections:

- [“Video Sync Generator” on page 25–2](#)
- [“Pixel Converter” on page 25–5](#)
- [“Device Support” on page 25–6](#)
- [“Hardware Simulation Considerations” on page 25–6](#)

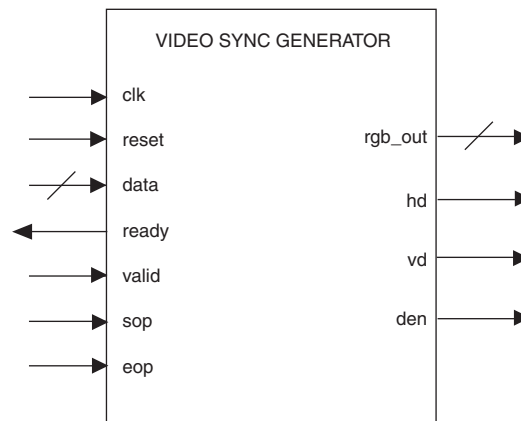
Video Sync Generator

This section describes the hardware structure and functionality of the video sync generator core.

Functional Description

The video sync generator core adds horizontal and vertical synchronization signals to the pixel data that comes through its Avalon® (Avalon-ST) input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data. [Figure 25-2](#) shows a block diagram of the video sync generator.

Figure 25-2. Video Sync Generator Block Diagram



You can configure various aspects of the core and its Avalon-ST interface to suit your requirements. You can specify the data width, number of beats required to transfer each pixel and synchronization signals. See [“Instantiating the Core in SOPC Builder” on page 25-3](#) for more information on the available options.

To ensure incoming pixel data is sent to the display controller with correct timing, the video sync generator core must synchronize itself to the first pixel in a frame. The first active pixel is indicated by an `sop` pulse.

The video sync generator core expects continuous streams of pixel data at its input interface and assumes that each incoming packet contains the correct number of pixels (Number of rows * Number of columns). Data starvation disrupts synchronization and results in unexpected output on the display.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the video sync generator core in SOPC Builder to configure the core. Table 25–1 lists the available parameters in the MegaWizard interface.

Table 25–1. Video Sync Generator Parameters

Parameter Name	Description
Horizontal Sync Pulse Pixels	The width of the h-sync pulse in number of pixels.
Total Vertical Scan Lines	The total number of lines in one video frame. The value is the sum of the following parameters: Number of Rows , Vertical Blank Lines , and Vertical Front Porch Lines .
Number of Rows	The number of active scan lines in each video frame.
Horizontal Sync Pulse Polarity	The polarity of the h-sync pulse; 0 = active low and 1 = active high.
Horizontal Front Porch Pixels	The number of blanking pixels that follow the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Sync Pulse Polarity	The polarity of the v-sync pulse; 0 = active low and 1 = active high.
Vertical Sync Pulse Lines	The width of the v-sync pulse in number of lines.
Vertical Front Porch Lines	The number of blanking lines that follow the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Number of Columns	The number of active pixels in each line.
Horizontal Blank Pixels	The number of blanking pixels that precede the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Total Horizontal Scan Pixels	The total number of pixels in one line. The value is the sum of the following parameters: Number of Columns , Horizontal Blank Pixel , and Horizontal Front Porch Pixels .
Beats Per Pixel	<p>The number of beats required to transfer one pixel. Valid values are 1 and 3. This parameter, when multiplied by Data Stream Bit Width must be equal to the total number of bits in one pixel. This parameter affects the operating clock frequency, as shown in the following equation:</p> $\text{Operating clock frequency} = (\text{Beats per pixel}) * (\text{Pixel_rate}), \text{ where}$ $\text{Pixel_rate (in MHz)} = ((\text{Total Horizontal Scan Pixels}) * (\text{Total Vertical Scan Lines}) * (\text{Display refresh rate in Hz}))/1000000.$
Vertical Blank Lines	The number of blanking lines that proceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Data Stream Bit Width	The width of the inbound and outbound data.

Signals

Table 25-2 lists the input and output signals for the video sync generator core.

Table 25-2. Video Sync Generator Core Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	System clock.
reset	1	Input	System reset.
Avalon-ST Signals			
data	Variable-width	Input	Incoming pixel data. The datawidth is determined by the parameter Data Stream Bit Width .
ready	1	Output	This signal is asserted when the video sync generator is ready to receive the pixel data.
valid	1	Input	This signal is not used by the video sync generator core because the core always expects valid pixel data on the next clock cycle after the <code>ready</code> signal is asserted.
sop	1	Input	Start-of-packet. This signal is asserted when the first pixel is received.
eop	1	Input	End-of-packet. This signal is asserted when the last pixel is received.
LCD Output Signals			
rgb_out	Variable-width	Output	Display data. The datawidth is determined by the parameter Data Stream Bit Width .
hd	1	Output	Horizontal synchronization pulse for display.
vd	1	Output	Vertical synchronization pulse for display.
den	1	Output	This signal is asserted when the video sync generator core outputs valid data for display.

Timing Diagrams

The horizontal and vertical synchronization timings are determined by the parameters setting. Figure 25-3 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 8 and 3, respectively.

Figure 25-3. Horizontal Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel

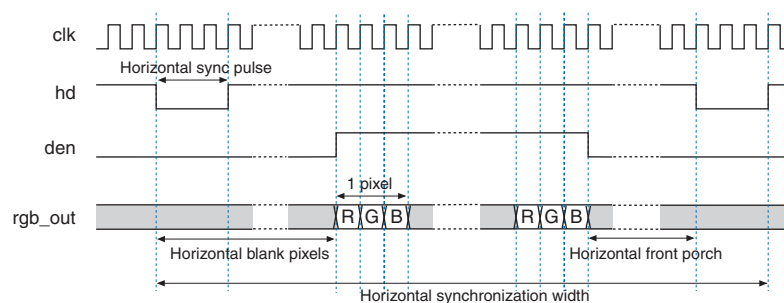


Figure 25-4 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 24 and 1, respectively.

Figure 25-4. Horizontal Synchronization Timing—24 Bits DataWidth and 1 Beat Per Pixel

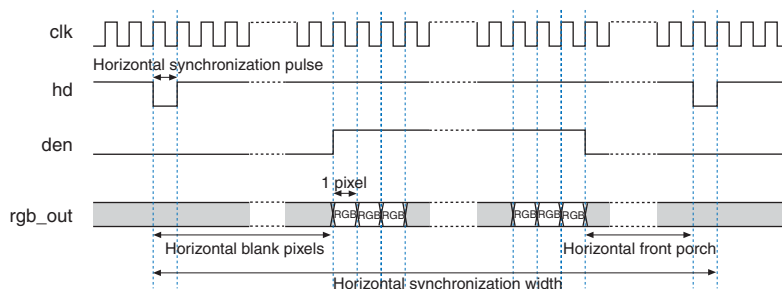
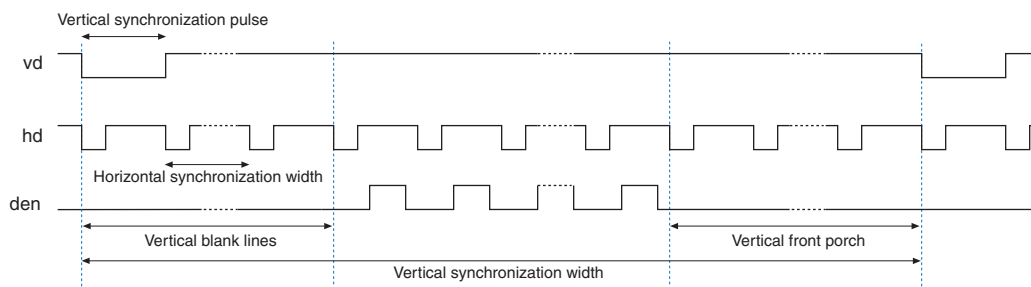


Figure 25-5 shows the vertical synchronization timing.

Figure 25-5. Vertical Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel / 24 Bits DataWidth and 1 Beat Per Pixel



Pixel Converter

This section describes the hardware structure and functionality of the pixel converter core.

Functional Description

The pixel converter core receives pixel data on its Avalon-ST input interface and transforms the pixel data to the format required by the video sync generator. The least significant byte of the 32-bit wide pixel data is removed and the remaining 24 bits are wired directly to the core's Avalon-ST output interface.

Instantiating the Core in SOPC Builder

Use the MegaWizard interface for the pixel converter core in SOPC Builder to add the core to a system. You can configure the following parameter:

Source symbols per beat—The number of symbols per beat on the Avalon-ST source interface.

Signals

Table 25-3 lists the input and output signals for the pixel converter core.

Table 25-3. Pixel Converter Input Interface Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	Not in use.
reset_n	1	Input	
Avalon-ST Signals			
data_in	32	Input	Incoming pixel data. Contains four 8-bit symbols that are transferred in 1 beat.
data_out	24	Output	Output data. Contains three 8-bit symbols that are transferred in 1 beat.
sop_in	1	Input	Wired directly to the corresponding output signals.
eop_in	1	Input	
ready_in	1	Input	
valid_in	1	Input	
empty_in	1	Input	
sop_out	1	Output	Wired directly from the input signals.
eop_out	1	Output	
ready_out	1	Output	
valid_out	1	Output	
empty_out	1	Output	

Device Support

The video sync generator and pixel converter cores support all Altera device families.

Hardware Simulation Considerations

For a typical 60 Hz refresh rate, set the simulation length for the video sync generator core to at least 16.7 μ s to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

Referenced Documents


This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 25-4 shows the revision history for this chapter.

Table 25-4. 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	Added new parameters for both cores.	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](#).

Core Overview

The Interval Timer core with Avalon® interface is an interval timer for Avalon-based processor systems, such as a Nios® II processor system. The core provides the following features:

- 32-bit and 64-bit counters.
- Controls to start, stop, and reset the timer.
- Two count modes: count down once and continuous count-down.
- Count-down period register.
- Option to enable or disable the interrupt request (IRQ) when timer reaches zero.
- Optional watchdog timer feature that resets the system if timer ever reaches zero.
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero.
- Compatible with 32-bit and 16-bit processors.

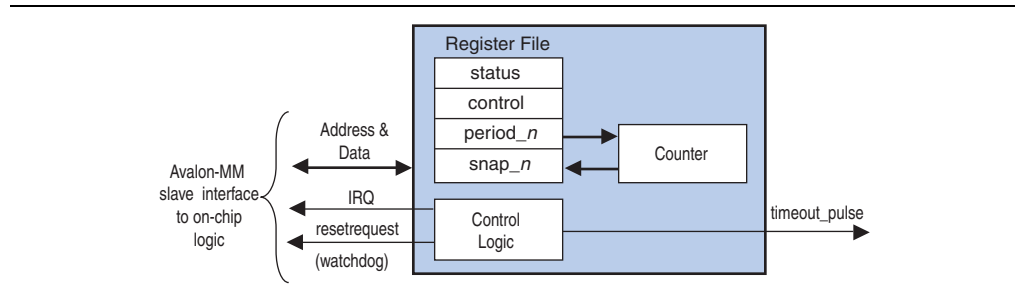
Device drivers are provided in the HAL system library for the Nios II processor. The interval timer core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- [“Functional Description”](#)
- [“Device Support” on page 26–2](#)
- [“Instantiating the Core in SOPC Builder” on page 26–3](#)
- [“Software Programming Model” on page 26–5](#)

Functional Description

Figure 26–1 shows a block diagram of the interval timer core.

Figure 26–1. Interval Timer Core Block Diagram



The interval timer core has two user-visible features:

- The Avalon Memory-Mapped (Avalon-MM) interface that provides access to six 16-bit registers
- An optional pulse output that can be used as a periodic pulse generator

All registers are 16-bits wide, making the core compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the core is configured with a fixed period, the period registers do not exist in hardware.

The following sequence describes the basic behavior of the interval timer core:

- An Avalon-MM master peripheral, such as a Nios II processor, writes the core's `control` register to perform the following tasks:
 - Start and stop the timer
 - Enable/disable the IRQ
 - Specify count-down once or continuous count-down mode
- A processor reads the `status` register for information about current timer activity.
- A processor can specify the timer period by writing a value to the period registers.
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers.
- A processor can read the current counter value by first writing to one of the `snap` registers to request a coherent snapshot of the counter, and then reading the `snap` registers for the full value.
- When the count reaches zero, one or more of the following events are triggered:
 - If IRQs are enabled, an IRQ is generated.
 - The optional pulse-generator output is asserted for one clock period.
 - The optional watchdog output resets the system.

Avalon-MM Slave Interface

The interval timer core implements a simple Avalon-MM slave interface to provide access to the register file. The Avalon-MM slave port uses the `resetrequest` signal to implement watchdog timer behavior. This signal is a non-maskable reset signal, and it drives the reset input of all Avalon-MM peripherals in the SOPC Builder system. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. For more information, refer to [“Configuring the Timer as a Watchdog Timer” on page 26-4](#).

Device Support

The interval timer core supports all Altera® device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the interval timer core in SOPC Builder to specify the hardware features. This section describes the options available in the MegaWizard Interface.

Timeout Period

The **Timeout Period** setting determines the initial value of the period registers. When the **Writeable period** option is on, a processor can change the value of the period by writing to the period registers. When the **Writeable period** option is off, the period is fixed and cannot be updated at runtime. See [“Hardware Options” on page 26-3](#) for information on register options.

The **Timeout Period** is an integer multiple of the **Timer Frequency**. The **Timer Frequency** is fixed at the frequency setting of the system clock associated with the timer. The **Timeout Period** setting can be specified in units of **µs** (microseconds), **ms** (milliseconds), **seconds**, or **clocks** (number of cycles of the system clock associated with the timer). The actual period depends on the frequency of the system clock associated with the timer. If the period is specified in **µs**, **ms**, or **seconds**, the true period will be the smallest number of clock cycles that is greater or equal to the specified **Timeout Period** value. For example, if the associated system clock has a frequency of 30 **ns**, and the specified **Timeout Period** value is 1 **µs**, the true timeout period will be 1.020 microseconds.

Counter Size

The **Counter Size** setting determines the timer's width, which can be set to either 32 or 64 bits. A 32-bit timer has two 16-bit period registers, whereas a 64-bit timer has four 16-bit period registers. This option applies to the snap registers as well.

Hardware Options

The following options affect the hardware structure of the interval timer core. As a convenience, the **Preset Configurations** list offers several pre-defined hardware configurations, such as:

- **Simple periodic interrupt**—This configuration is useful for systems that require only a periodic IRQ generator. The period is fixed and the timer cannot be stopped, but the IRQ can be disabled.
- **Full-featured**—This configuration is useful for embedded processor systems that require a timer with variable period that can be started and stopped under processor control.
- **Watchdog**—This configuration is useful for systems that require watchdog timer to reset the system in the event that the system has stopped responding. Refer to [“Configuring the Timer as a Watchdog Timer” on page 26-4](#).

Register Options

Table 26-1 shows the settings that affect the interval timer core's registers.

Table 26-1. Register Options

Option	Description
Writeable period	When this option is enabled, a master peripheral can change the count-down period by writing to the period registers. When disabled, the count-down period is fixed at the specified Timeout Period , and the period registers do not exist in hardware.
Readable snapshot	When this option is enabled, a master peripheral can read a snapshot of the current count-down. When disabled, the status of the counter is detectable only via other indicators, such as the <code>status</code> register or the IRQ signal. In this case, the snap registers do not exist in hardware, and reading these registers produces an undefined value.
Start/Stop control bits	When this option is enabled, a master peripheral can start and stop the timer by writing the START and STOP bits in the <code>control</code> register. When disabled, the timer runs continuously. When the System reset on timeout (watchdog) option is enabled, the START bit is also present, regardless of the Start/Stop control bits option.

Output Signal Options

Table 26-2 shows the settings that affect the interval timer core's output signals.

Table 26-2. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is on, the core outputs a signal <code>timeout_pulse</code> . This signal pulses high for one clock cycle whenever the timer reaches zero. When this option is off, the <code>timeout_pulse</code> signal does not exist.
System reset on timeout (watchdog)	When this option is on, the core's Avalon-MM slave port includes the <code>resetrequest</code> signal. This signal pulses high for one clock cycle whenever the timer reaches zero resulting in a system-wide reset. The internal timer is stopped at reset. Explicitly writing the START bit of the <code>control</code> register starts the timer. When this option is off, the <code>resetrequest</code> signal does not exist. Refer to "Configuring the Timer as a Watchdog Timer".

Configuring the Timer as a Watchdog Timer

To configure the core for use as a watchdog, in the MegaWizard Interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

A watchdog timer wakes up (comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's `START` bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state.

Software Programming Model

The following sections describe the software programming model for the interval timer core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the interval timer core using the HAL application programming interface (API) functions.

HAL System Library Support

The Altera-provided drivers integrate into the HAL system library for Nios II systems. When possible, HAL users should access the core via the HAL API, rather than accessing the core's registers directly.

Altera provides a driver for both the HAL timer device models: system clock timer, and timestamp timer.

System Clock Driver

When configured as the system clock, the interval timer core runs continuously in periodic mode, using the default period set in SOPC builder. The system clock services are then run as a part of the interrupt service routine for this timer. The driver is interrupt-driven, and therefore must have its interrupt signal connected in the system hardware.

The Nios II integrated development environment (IDE) allows you to specify system library properties that determine which timer device will be used as the system clock timer.


Timestamp Driver

The interval timer core may be used as a timestamp device if it meets the following conditions:

- The timer has a writeable `period` register, as configured in SOPC Builder.
- The timer is not selected as the system clock.

The Nios II IDE allows you to specify system library properties that determine which timer device will be used as the timestamp timer.

If the timer hardware is not configured with writeable `period` registers, calls to the `alt_timestamp_start()` API function will not reset the timestamp counter. All other HAL API calls will perform as expected.

 For more information about using the system clock and timestamp features that use these drivers, refer to the *Nios II Software Developer's Handbook*. The Nios II Embedded Design Suite (EDS) also provides several example designs that use the interval timer core.

Limitations

The HAL driver for the interval timer core does not support the watchdog reset feature of the core.

Software Files

The interval timer core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_timer_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_timer.h, altera_avalon_timer_sc.c, altera_avalon_timer_ts.c, altera_avalon_timer_vars.c**—These files implement the timer device drivers for the HAL system library.

Register Map

You do not need to access the interval timer core directly via its registers if using the standard features provided in the HAL system library for the Nios II processor. 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 correctly.

Table 26-3 shows the register map for the 32-bit timer. The interval timer core uses native address alignment. For example, to access the `control` register value, use offset 0x4.

Table 26-3. Register Map—32-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)					RUN	TO
1	control	RW	(1)		STOP	START	CONT	ITO	
2	periodl	RW	Timeout Period – 1 (bits [15:0])						
3	periodh	RW	Timeout Period – 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

Note to Table 26-3:

(1) Reserved. Read values are undefined. Write zero.


 For more information about native address alignment, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces*.

Table 26-4 shows the register map for the 64-bit timer.

Table 26-4. Register Map—64-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)					RUN	TO
1	control	RW	(1)		STOP	START	CONT	ITO	
2	period_0	RW	Timeout Period – 1 (bits [15:0])						
3	period_1	RW	Timeout Period – 1 (bits [31:16])						
4	period_2	RW	Timeout Period – 1 (bits [47:32])						
5	period_3	RW	Timeout Period – 1 (bits [63:48])						
6	snap_0	RW	Counter Snapshot (bits [15:0])						
7	snap_1	RW	Counter Snapshot (bits [31:16])						
8	snap_2	RW	Counter Snapshot (bits [47:32])						
9	snap_3	RW	Counter Snapshot (bits [63:48])						

Note to Table 26-4:

(1) Reserved. Read values are undefined. Write zero.

status Register

The status register has two defined bits, as shown in Table 26-5.

Table 26-5. status Register Bits

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

control Register

The control register has four defined bits, as shown in Table 26-6.

Table 26-6. control Register Bits (Part 1 of 2)

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START (1)	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.

Table 26-6. control Register Bits (Part 2 of 2)

Bit	Name	R/W/C	Description
3	STOP (1)	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with Start/Stop control bits off, writing the STOP bit has no effect.

Note to Table 26-6:

(1) Writing 1 to both START and STOP bits simultaneously produces an undefined result.

period_n Registers

The `period_n` registers together store the timeout period value. The internal counter is loaded with the value stored in these registers whenever one of the following occurs:

- A write operation to one of the `period_n` register
- The internal counter reaches 0

The timer's actual period is one cycle greater than the value stored in the `period_n` registers because the counter assumes the value zero for one clock cycle.

Writing to one of the `period_n` registers stops the internal counter, except when the hardware is configured with **Start/Stop control bits** off. If **Start/Stop control bits** is off, writing either register does not stop the counter. When the hardware is configured with **Writeable period** disabled, writing to one of the `period_n` registers causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snap_n Registers

A master peripheral may request a coherent snapshot of the current internal counter by performing a write operation (write-data ignored) to one of the `snap_n` registers. When a write occurs, the value of the counter is copied to `snap_n` registers. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

Interrupt Behavior

The interval timer core generates an IRQ whenever the internal counter reaches zero and the ITO bit of the `control` register is set to 1. Acknowledge the IRQ in one of two ways:

- Clear the TO bit of the `status` register
- Disable interrupts by clearing the ITO bit of the `control` register

Failure to acknowledge the IRQ produces an undefined result.

Referenced Documents


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

Document Revision History

Table 26-7 shows the revision history for this chapter.

Table 26-7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	Revised descriptions of register fields and bits.	The timer component is using native address alignment.
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Updated the core's name to reflect the name used in SOPC Builder.	—
May 2008 v8.0.0	Added a new parameter and register map for the 64-bit timer.	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](#).

Core Overview

Multiprocessor environments can use the mutex core with Avalon® interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor to enable use of the hardware mutex.

The mutex core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- [“Functional Description”](#)
- [“Device Support” on page 27-2](#)
- [“Instantiating the Core in SOPC Builder” on page 27-2](#)
- [“Software Programming Model” on page 27-2](#)
- [“Mutex API” on page 27-4](#)

Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to two memory-mapped, 32-bit registers. [Table 27-1](#) shows the registers.

Table 27-1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description				
			31	16	15	1	0
0	mutex	RW	OWNER		VALUE		
1	reset	RW	Reserved			RESET	

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the VALUE field is 0x0000, the mutex is unlocked and available. Otherwise, the mutex is locked and unavailable.
- The mutex register is always readable. Avalon-MM master peripherals, such as a processor, can read the mutex register to determine its current state.

- The mutex register is writable only under specific conditions. A write operation changes the mutex register only if one or both of the following conditions are true:
 - The VALUE field of the mutex register is zero.
 - The OWNER field of the mutex register matches the OWNER field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the OWNER field, and writing a non-zero value to the VALUE field. The processor then checks if the acquisition succeeded by verifying the OWNER field.
- After system reset, the RESET bit in the reset register is high. Writing a one to this bit clears it.

Device Support

The mutex core supports all Altera device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the mutex core in SOPC Builder to specify the core's hardware features. The MegaWizard Interface provides the following options:

- **Initial Value**—the initial contents of the VALUE field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the OWNER field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

Software Programming Model

The following sections describe the software programming model for the mutex core. For Nios II processor users, Altera provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its cpuid control register to the OWNER field of the mutex register.

Software Files

Altera provides the following software files accompanying the mutex core:

- **altera_avalon_mutex_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h**—Defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—Contains the implementations of the functions to access the mutex core

Hardware Access Routines

This section describes the low-level software constructs for manipulating the mutex core. The file `altera_avalon_mutex.h` declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares routines for accessing the mutex hardware structure, listed in [Table 27-2](#).

Table 27-2. Hardware Access Routines

Function Name	Description
<code>altera_avalon_mutex_open()</code>	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
<code>altera_avalon_mutex_trylock()</code>	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
<code>altera_avalon_mutex_lock()</code>	Locks the mutex. Will not return until it has successfully claimed the mutex.
<code>altera_avalon_mutex_unlock()</code>	Unlocks the mutex.
<code>altera_avalon_mutex_is_mine()</code>	Determines if this CPU owns the mutex.
<code>altera_avalon_mutex_first_lock()</code>	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section [“Mutex API” on page 27-4](#).

The code shown in [Example 27-1](#) demonstrates opening a mutex device handle and locking a mutex.

Example 27-1. Opening and Locking a mutex

```
#include <altera_avalon_mutex.h>
/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( "/dev/mutex" );
/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );
/*
 * Access a shared resource here.
 */
/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Mutex API

This section describes the application programming interface (API) for the mutex core.

altera_avalon_mutex_is_mine()

Prototype: `int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: `dev`—the mutex device to test.
Returns: Returns non zero if the mutex is owned by this CPU.
Description: `altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

altera_avalon_mutex_first_lock()

Prototype: `int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: `dev`—the mutex device to test.
Returns: Returns 1 if this mutex has not been released since reset, otherwise returns 0.
Description: `altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

altera_avalon_mutex_lock()

Prototype: `void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: `dev`—the mutex device to acquire.
`value`—the new value to write to the mutex.
Returns: —
Description: `altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

altera_avalon_mutex_open()

Prototype: `alt_mutex_dev* alt_hardware_mutex_open(const char* name)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: name—the name of the mutex device to open.
Returns: A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.
Description: `altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

altera_avalon_mutex_trylock()

Prototype: `int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: dev—the mutex device to lock.
value—the new value to write to the mutex.
Returns: 0 = The mutex was successfully locked.
Others = The mutex was not locked.
Description: `altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

altera_avalon_mutex_unlock()

Prototype: `void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mutex.h>`
Parameters: dev—the mutex device to unlock.
Returns: Null.
Description: `altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

Document Revision History

Table 27-3 shows the revision history for this chapter.

Table 27-3. 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	No change from previous release.	—



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

Core Overview

Multiprocessor environments can use the mailbox core with Avalon® interface to send messages between processors.

The mailbox core contains mutexes to ensure that only one processor modifies the mailbox contents at a time. The mailbox core must be used in conjunction with a separate shared memory that is used for storing the actual messages.

The mailbox core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor. The mailbox core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- “Functional Description”
- “Device Support” on page 28–2
- “Instantiating the Core in SOPC Builder” on page 28–2
- “Software Programming Model” on page 28–3
- “Mailbox API” on page 28–5

Functional Description

The mailbox core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to four memory-mapped, 32-bit registers. [Table 28–1](#) shows the registers.

Table 28–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description			
			31	16	15	1
0	mutex0	RW	OWNER	VALUE		
1	reset0	RW	Reserved			RESET
2	mutex1	RW	OWNER	VALUE		
3	reset1	RW	Reserved			RESET

The mailbox component contains two mutexes: One to ensure unique write access to shared memory and one to ensure unique read access from shared memory. The mailbox core is used in conjunction with a separate memory in the system that is shared among multiple processors.

Mailbox functionality using the mutexes and memory is implemented entirely in the software. Refer to [“Software Programming Model” on page 28–3](#) for details about how to use the mailbox core in software.



For a detailed description of the mutex hardware operation, refer to the [Mutex Core](#) chapter in volume 5 of the *Quartus II Handbook*.

Device Support

The mailbox core supports all Altera® device families.

Instantiating the Core in SOPC Builder

You can instantiate and configure the mailbox core in an SOPC Builder system using the following process:

1. Decide which processors share the mailbox.
2. On the SOPC Builder **System Contents** tab, instantiate a memory component to serve as the mailbox buffer. Any RAM can be used as the mailbox buffer. The mailbox buffer can share space in an existing memory, such as program memory; it does not require a dedicated memory.
3. On the SOPC Builder **System Contents** tab, instantiate the mailbox component. The mailbox MegaWizard™ Interface presents the following options:
 - **Memory module**—Specifies which memory to use for the mailbox buffer. If the **Memory module** list does not contain the desired shared memory, the memory is not connected in the system correctly. Refer to Step 4 on page 28-2.
 - **CPUs available with this memory**—Shows all the processors that can share the mailbox. This field is always read-only. Use it to verify that the processor connections are correct. If a processor that needs to share the mailbox is missing from the list, refer to Step 4 on page 28-2.
 - **Shared mailbox memory offset**—Specifies an offset into the memory. The mailbox message buffer starts at this offset.
 - **Mailbox size (bytes)**—Specifies the number of bytes to use for the mailbox message buffer. The Nios II driver software provided by Altera uses eight bytes of overhead to implement the mailbox functionality. For a mailbox capable of passing only one message at a time, **Mailbox size (bytes)** must be at least 12 bytes.
 - **Maximum available bytes**—Specifies the number of bytes in the selected memory available for use as the mailbox message buffer. This field is always read-only.
4. If not already connected, make component connections on the SOPC Builder **System Contents** tab.
 - a. Connect each processor's data bus master port to the mailbox slave port.
 - b. Connect each processor's data bus master port to the shared mailbox memory.

Software Programming Model

The following sections describe the software programming model for the mailbox core. For Nios II processor users, Altera provides routines to access the mailbox core hardware. These functions are specific to the mailbox core and directly manipulate low-level hardware.

The mailbox software programming model has the following characteristics and assumes there are multiple processors accessing a single mailbox core and a shared memory:

- Each mailbox message is one 32-bit word.
- There is a predefined address range in shared memory dedicated to storing messages. The size of this address range imposes a maximum limit on the number of messages pending.
- The mailbox software implements a message FIFO between processors. Only one processor can write to the mailbox at a time, and only one processor can read from the mailbox at a time, ensuring message integrity.
- The software on both the sending and receiving processors must agree on a protocol for interpreting mailbox messages. Typically, processors treat the message as a pointer to a structure in shared memory.
- The sending processor can post messages in succession, up to the limit imposed by the size of the message address range.
- When messages exist in the mailbox, the receiving processor can read messages. The receiving processor can block until a message appears, or it can poll the mailbox for new messages.
- Reading the message removes the message from the mailbox.

Software Files

Altera provides the following software files accompanying the mailbox core hardware:

- **altera_avalon_mailbox_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mailbox.h**—Defines data structures and functions to access the mailbox core hardware.
- **altera_avalon_mailbox.c**—Contains the implementations of the functions to access the mailbox core.

Programming with the Mailbox Core

This section describes the software constructs for manipulating the mailbox core hardware.

The file **altera_avalon_mailbox.h** declares a structure `alt_mailbox_dev` that represents an instance of a mailbox device. It also declares functions for accessing the mailbox hardware structure, listed in [Table 28-2](#). For a complete description of each function, refer to ["Mailbox API" on page 28-5](#).

Table 28-2. Mailbox API Functions

Function Name	Description
<code>altera_avalon_mailbox_close()</code>	Closes the handle to a mailbox.
<code>altera_avalon_mailbox_get()</code>	Returns a message if one is present, but does not block waiting for a message.
<code>altera_avalon_mailbox_open()</code>	Claims a handle to a mailbox, enabling all the other functions to access the mailbox core.
<code>altera_avalon_mailbox_pend()</code>	Blocks waiting for a message to be in the mailbox.
<code>altera_avalon_mailbox_post()</code>	Posts a message to the mailbox.

Example 28-1 demonstrates writing to and reading from a mailbox. For this example, assume that the hardware system has two processors communicating via mailboxes. The system includes two mailbox cores, which provide two-way communication between the processors.

Example 28-1. Writing to and Reading from a Mailbox

```
#include <stdio.h>
#include "altera_avalon_mailbox.h"

int main()
{
    alt_u32 message = 0;
    alt_mailbox_dev* send_dev, rcv_dev;
    /* Open the two mailboxes between this processor and another */
    send_dev = altera_avalon_mailbox_open("/dev/mailbox_0");
    rcv_dev = altera_avalon_mailbox_open("/dev/mailbox_1");

    while(1)
    {
        /* Send a message to the other processor */
        altera_avalon_mailbox_post(send_dev, message);

        /* Wait for the other processor to send a message back */
        message = altera_avalon_mailbox_pend(rcv_dev);
    }
    return 0;
}
```

Mailbox API

This section describes the application programming interface (API) for the mailbox core.

altera_avalon_mailbox_close()

Prototype: `void altera_avalon_mailbox_close (alt_mailbox_dev* dev);`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mailbox.h>`
Parameters: `dev`—the mailbox to close.
Returns: Null.
Description: `altera_avalon_mailbox_close()` closes the mailbox.

altera_avalon_mailbox_get()

Prototype: `alt_u32 altera_avalon_mailbox_get (alt_mailbox_dev* dev, int* err);`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mailbox.h>`
Parameters: `dev`—the mailbox handle.
`err`—pointer to an error code that is returned.
Returns: Returns a message if one is available in the mailbox, otherwise returns 0. The value pointed to by `err` is 0 if the message was read correctly, or `EWOULDBLOCK` if there is no message to read.
Description: `altera_avalon_mailbox_get()` returns a message if one is present, but does not block waiting for a message.

altera_avalon_mailbox_open()

Prototype: `alt_mailbox_dev* altera_avalon_mailbox_open (const char* name);`
Thread-safe: Yes.
Available from ISR: No.
Include: `<altera_avalon_mailbox.h>`
Parameters: `name`—the name of the mailbox device to open.
Returns: Returns a handle to the mailbox, or NULL if this mailbox does not exist.
Description: `altera_avalon_mailbox_open()` opens a mailbox.

altera_avalon_mailbox_pend()

Prototype: `alt_u32 altera_avalon_mailbox_pend (alt_mailbox_dev* dev);`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mailbox.h>`

Parameters: `dev`—the mailbox device to read a message from.

Returns: Returns the message.

Description: `altera_avalon_mailbox_pend()` is a blocking routine that waits for a message to appear in the mailbox and then reads it.

altera_avalon_mailbox_post()

Prototype: `int altera_avalon_mailbox_post (alt_mailbox_dev* dev, alt_u32 msg);`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mailbox.h>`

Parameters: `dev`—the mailbox device to post a message to.
`msg`—the value to post.

Returns: Returns 0 on success, or `EWOULDBLOCK` if the mailbox is full.

Description: `altera_avalon_mailbox_post()` posts a message to the mailbox.

Document Revision History

Table 28-3 shows the revision history for this chapter.

Table 28-3. 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	No change from previous release.	—



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

Core Overview

The vectored interrupt controller (VIC) core serves the following main purposes:

- Provides an interface to the interrupts in your system
- Reduces interrupt overhead
- Manages large numbers of interrupts

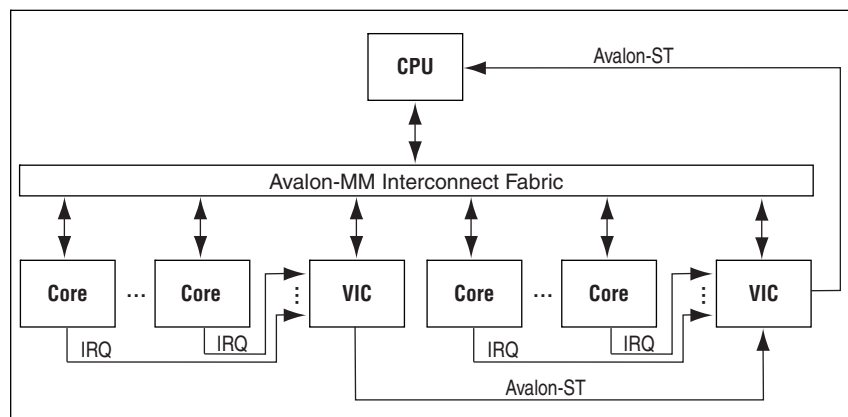
The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt. When external interrupts occur in a system containing a VIC, the VIC determines the highest priority interrupt, determines the source that is requesting service, computes the requested handler address (RHA), and provides information, including the RHA, to the processor.

The VIC core contains the following interfaces:

- Up to 32 interrupt input ports per VIC core
- One Avalon[®] Memory-Mapped (Avalon-MM) slave interface to access the internal control status registers (CSR)
- One Avalon Streaming (Avalon-ST) interface output interface to pass information about the selected interrupt
- One optional Avalon-ST interface input interface to receive the Avalon-ST output in systems with daisy-chained VICs

Figure 29–1 outlines the basic layout of a system containing two VIC components.

Figure 29–1. Sample System Layout



To use the VIC, the processor in your system needs to have a matching Avalon-ST interface to accept the interrupt information, such as the Nios[®] II processor's external interrupt controller interface.

The characteristics of each interrupt port are configured via the Avalon-MM slave interface. When you need more than 32 interrupt ports, you can daisy chain multiple VICs together.

The VIC core provides the following features:

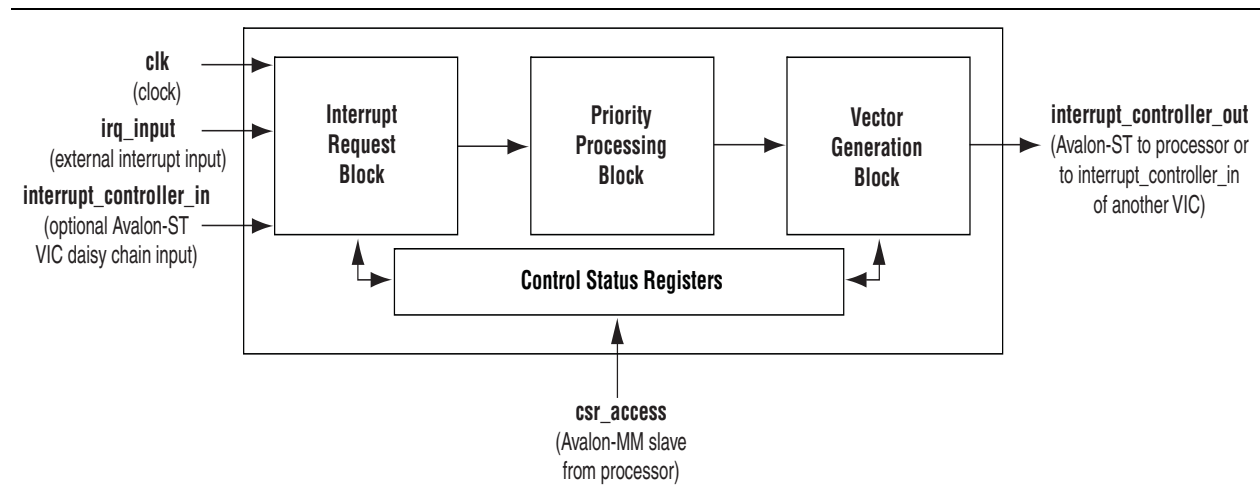
- Separate programmable requested interrupt level (RIL) for each interrupt
- Separate programmable requested register set (RRS) for each interrupt, to tell the interrupt handler which processor register set to use
- Separate programmable requested non-maskable interrupt (RNMI) flag for each interrupt, to control whether each interrupt is maskable or non-maskable
- Software-controlled priority arbitration scheme

The VIC core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios II processor, Altera provides Hardware Abstraction Layer (HAL) driver routines for the VIC core. Refer to “[Altera HAL Software Programming Model](#)” on page 29-10 for HAL support details.

Functional Description

Figure 29-2 shows a high-level block diagram of the VIC core.

Figure 29-2. VIC Block Diagram



External Interfaces

The following sections describe the external interfaces for the VIC core.

clk

clk is a system clock interface. This interface connects to your system’s main clock source. The interface’s signals are clk and reset_n.

irq_input

irq_input comprises up to 32 single-bit, level-sensitive Avalon interrupt receiver interfaces. These interfaces connect to interrupt sources. There is one irq signal for each interface.

interrupt_controller_out

interrupt_controller_out is an Avalon-ST output interface, as defined in Table 29-2, configured with a ready latency of 0 cycles. This interface connects to your processor or to the interrupt_controller_in interface of another VIC. The interface's signals are valid and data. Table 29-1 shows the interface's parameters and the corresponding parameter values.

Table 29-1. interrupt_controller_out and interrupt_controller_in Parameters

Parameter	Value
Symbol width	45 bits
Ready latency	0 cycles

interrupt_controller_in

interrupt_controller_in is an optional Avalon-ST input interface, as defined in Table 29-2, configured with a ready latency of 0 cycles. Include this interface in the second, third, etc, VIC components of a daisy-chained multiple VIC system. This interface connects to the interrupt_controller_out interface of the immediately-preceding VIC in the chain. The interface's signals are valid and data. Table 29-1 shows the interface's parameters and the corresponding parameter values.

The interrupt_controller_out and interrupt_controller_in interfaces have identical Avalon-ST formats so you can daisy chain VICs together in SOPC Builder when you need more than 32 interrupts. interrupt_controller_out always provides valid data and cannot be back-pressured. Table 29-2 shows the fields of the VIC's 45-bit Avalon-ST interface.

Table 29-2. VIC Avalon-ST Interface Fields

44	43	42	41	40	39	38	38	37	...	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RHA (1)											RRS (2)				RNMI (2)	RIL (2)														

Notes to Table 29-2:


- (1) RHA contains the 32-bit address of the interrupt handling routine.
- (2) Refer to Table 29-6 for a description of this field.

csr_access

csr_access is a VIC CSR interface consisting of an Avalon-MM slave interface. This interface connects to the data master of your processor. The interface's signals are read, write, address, readdata, and writedata. Table 29-3 shows the interface's parameters and the corresponding parameter values.

Table 29-3. csr_access Parameters

Parameter	Value
Read wait	1 cycle
Write wait	0 cycles
Ready latency	4 cycles

 For information about the Avalon-MM slave and Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Functional Blocks

The following main design blocks comprise the VIC core:

- Interrupt request block
- Priority processing block
- Vector generation block

The following sections describe each functional block.

Interrupt Request Block

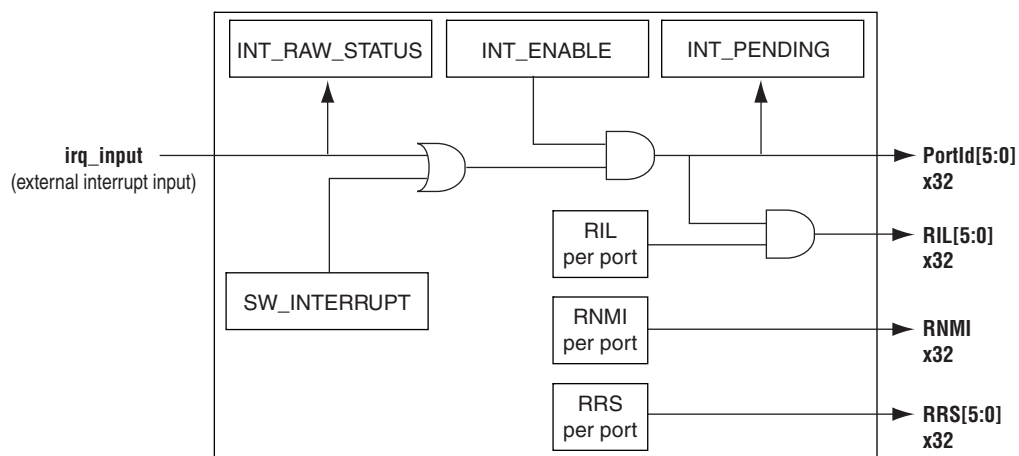
The interrupt request block controls the input interrupts, providing functionality such as setting interrupt levels, setting the per-interrupt programmable registers, masking interrupts, and managing software-controlled interrupts. You configure the number of interrupt input ports when you create the component. Refer to “*Instantiating the Core in SOPC Builder*” on page 29-9 for configuration options.

This block contains the majority of the VIC CSRs. The CSRs are accessed via the Avalon-MM slave interface.

Optional output from another VIC core can also come into the interrupt request block. Refer to “*Daisy Chaining VIC Cores*” on page 29-5 for more information.

Figure 29-3 shows the details of the interrupt request block. Each interrupt can be driven either by its associated `irq_input` signal (connected to a component with an interrupt source) or by a software trigger controlled by a CSR (even when there is no interrupt source connected to the `irq_input` signal).

Figure 29-3. Interrupt Request Block



Priority Processing Block

The priority processing block chooses the interrupt with the highest priority. The block receives information for each interrupt from the interrupt request block and passes information for the highest priority interrupt to the vector generation block.

The interrupt request with the numerically-largest RIL has priority. If multiple interrupts are pending with the same numerically-largest RIL, the numerically-lowest IRQ index of those interrupts has priority.

The RIL is a programmable interrupt level per port. An RIL value of zero disables the interrupt. You configure the bit width of the RIL when you create the component. Refer to “[Instantiating the Core in SOPC Builder](#)” on page 29-9 for configuration options.

Vector Generation Block

The vector generation block receives information for the highest priority interrupt from the priority processing block. The vector generation block uses the port identifier passed from the priority processing block along with the vector base address and bytes per vector programmed in the CSRs during software initialization to compute the RHA. [Equation 29-1](#) shows the RHA formula.


Equation 29-1. RHA Calculation

$$RHA = (\text{port identifier} \times \text{bytes per vector}) + \text{vector base address}$$

The information then passes out of the vector generation block and the VIC using the Avalon-ST interface. Refer to [Table 29-2 on page 29-3](#) for details about the outgoing information. The output from the VIC typically connects to a processor or another VIC, depending on the design.


Daisy Chaining VIC Cores

You can create a system with more than 32 interrupts by daisy chaining multiple VIC cores together. This is done by connecting the `interrupt_controller_out` interface of one VIC to the optional `interrupt_controller_in` interface of another VIC. For information about enabling the optional input interface, refer to “[Instantiating the Core in SOPC Builder](#)” on page 29-9.

 For performance reasons, always directly connect VIC components. Do not include other components between VICs.

When daisy chain input comes into the VIC, the priority processing block considers the daisy chain input along with the hardware and software interrupt inputs from the interrupt request block to determine the highest priority interrupt. If the daisy chain input has the highest RIL value, then the vector generation block passes the daisy chain port values unchanged directly out of the VIC.

You can daisy chain VICs with fewer than 32 interrupt ports. The number of daisy chain connections is only limited to the hardware and software resources. Refer to “[Latency Information](#)” for details about the impact of multiple VICs.

 Altera recommends setting the RIL width to the same value in all daisy-chained VIC components. If your RIL widths are different, wider RILs from upstream VICs are truncated.

Latency Information

The latency of an interrupt request traveling through the VIC is the sum of the delay through each of the blocks. Clock delays in the interrupt request block and the vector generation block are constants. The clock delay in the priority processing block varies depending on the total number of interrupt ports. Table 29-4 shows the latency information.

Table 29-4. Clock Delay Latencies

Number of Interrupt Ports	Interrupt Request Block Delay	Priority Processing Block Delay	Vector Generation Block Delay	Total Interrupt Latency
2 – 4	2 cycles	1 cycle	1 cycle	4 cycles
5 – 16	2 cycles	2 cycles	1 cycle	5 cycles
17 – 32	2 cycles	3 cycles	1 cycle	6 cycles

When daisy-chaining multiple VICs, interrupt latency increases as you move through the daisy chain away from the processor. For best performance, assign interrupts with the lowest latency requirements to the VIC connected directly to the processor.

Register Maps

The VIC core CSRs are accessible through the Avalon-MM interface. Software can configure the core and determine current status by accessing the registers.



Each register has a 32-bit interface that is not byte-enabled. You must access these registers with a master that is at least 32 bits wide.

Table 29-5 lists and describes the registers.

Table 29-5. Control Status Registers (Part 1 of 3)

Offset	Register Name	Access	Reset Value	Description
0 – 31	INT_CONFIG<n>	R/W	0	There are 32 interrupt configuration registers (INT_CONFIG0 – INT_CONFIG31). Each register contains fields to configure the behavior of its corresponding interrupt. If an interrupt input does not exist, reading the corresponding register always returns zero, and writing is ignored. Refer to Table 29-6 on page 29-8 for the INT_CONFIG register map.
32	INT_ENABLE	R/W	0	The interrupt enable register. INT_ENABLE holds the enabled status of each interrupt input. The 32 bits of the register map to the 32 interrupts available in the VIC core. For example, bit 5 corresponds to IRQ5. (1) Interrupt that are not enabled are never considered by the priority processing block, even when the interrupt input is asserted.
33	INT_ENABLE_SET	W	0	The interrupt enable set register. Writing a 1 to a bit in INT_ENABLE_SET sets the corresponding bit in INT_ENABLE. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (1)

Table 29-5. Control Status Registers (Part 2 of 3)

Offset	Register Name	Access	Reset Value	Description
34	INT_ENABLE_CLR	W	0	The interrupt enable clear register. Writing a 1 to a bit in INT_ENABLE_CLR clears corresponding bit in INT_ENABLE. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (1)
35	INT_PENDING	R	0	The interrupt pending register. INT_PENDING shows the pending interrupts. Each bit corresponds to one interrupt input. If an interrupt does not exist, reading its corresponding INT_PENDING bit always returns 0, and writing is ignored. Bits in INT_PENDING are set in the following ways: <ul style="list-style-type: none"> ■ An external interrupt is asserted at the VIC interface and the corresponding INT_ENABLE bit is set. ■ An SW_INTERRUPT bit is set and the corresponding INT_ENABLE bit is set. INT_PENDING bits remain set as long as either condition applies. Refer to Figure 29-3 on page 29-4 for details. (1)
36	INT_RAW_STATUS	R	0	The interrupt raw status register. INT_RAW_STATUS shows the unmasked state of the interrupt inputs. If an interrupt does not exist, reading the corresponding INT_RAW_STATUS bit always returns 0, and writing is ignored. A set bit indicates an interrupt is asserted at the interface of the VIC. The interrupt is asserted to the processor only when the corresponding bit in the interrupt enable register is set. (1)
37	SW_INTERRUPT	R/W	0	The software interrupt register. SW_INTERRUPT drives the software interrupts. Each interrupt is ORed with its external hardware interrupt and then enabled with INT_ENABLE. Refer to Figure 29-3 on page 29-4 for details. (1)
38	SW_INTERRUPT_SET	W	0	The software interrupt set register. Writing a 1 to a bit in SW_INTERRUPT_SET sets the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (1)
39	SW_INTERRUPT_CLR	W	0	The software interrupt clear register. Writing a 1 to a bit in SW_INTERRUPT_CLR clears the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (1)
40	VIC_CONFIG	R/W	0	The VIC configuration register. VIC_CONFIG allows software to configure settings that apply to the entire VIC. Refer to Table 29-7 on page 29-9 for the VIC_CONFIG register map.
41	VIC_STATUS	R	0	The VIC status register. VIC_STATUS shows the current status of the VIC. Refer to Table 29-8 on page 29-9 for the VIC_STATUS register map.
42	VEC_TBL_BASE	R/W	0	The vector table base register. VEC_TBL_BASE holds the base address of the vector table in the processor's memory space. Because the table must be aligned on a 4-byte boundary, bits 1:0 must always be 0.

Table 29-5. Control Status Registers (Part 3 of 3)

Offset	Register Name	Access	Reset Value	Description
43	VEC_TBL_ADDR	R	0	The vector table address register. VEC_TBL_ADDR provides the RHA for the IRQ value with the highest priority pending interrupt. If no interrupt is active, the value in this register is 0. If daisy chain input is enabled and is the highest priority interrupt, the vector table address register contains the RHA value from the daisy chain input interface.

Note to Table 29-5:

- (1) This register contains a 1-bit field for each of the 32 interrupt inputs. When the VIC is configured for less than 32 interrupts, the corresponding 1-bit field for each unused interrupt is tied to zero. Reading these locations always returns 0, and writing is ignored. To determine which interrupts are present, write the value 0xffff to the register and then read the register contents. Any bits that return zero do not have an interrupt present.

Table 29-6 provides a bit map for the 32 INT_CONFIG registers.

Table 29-6. The INT_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	RIL	R/W	0	The requested interrupt level field. RIL contains the interrupt level of the interrupt requesting service. The processor can use the value in this field to determine if the interrupt is of higher priority than what the processor is currently doing.
6	RNMI	R/W	0	The requested non-maskable interrupt field. RNMI contains the non-maskable interrupt mode of the interrupt requesting service. When 0, the interrupt is maskable. When 1, the interrupt is non-maskable.
7:12	RRS	R/W	0	The requested register set field. RRS contains the number of the processor register set that the processor should use for processing the interrupt. Software must ensure that only register values supported by the processor are used.
13:31	Reserved			


 For expanded definitions of the terms in Table 29-6, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Table 29-7 provides a bit map for the VIC_CONFIG register.

Table 29-7. The VIC_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:2	VEC_SIZE	R/W	0	The vector size field. VEC_SIZE specifies the number of bytes in each vector table entry. VEC_SIZE is encoded as $\log_2(\text{number of words}) - 2$. Namely: <ul style="list-style-type: none"> ■ 0—4 bytes per vector table entry ■ 1—8 bytes per vector table entry ■ 2—16 bytes per vector table entry ■ 3—32 bytes per vector table entry ■ 4—64 bytes per vector table entry ■ 5—128 bytes per vector table entry ■ 6—256 bytes per vector table entry ■ 7—512 bytes per vector table entry
3	DC	R/W	0	The daisy chain field. DC serves the following purposes: <ul style="list-style-type: none"> ■ Enables and disables the daisy chain input interface, if present. Write a 1 to enable the daisy chain interface; write a 0 to disable it. ■ Detects the presence of the daisy chain input interface. To detect, write a 1 to DC and then read DC. A return value of 1 means the daisy chain interface is present; 0 means the daisy chain interface is not present.
4:31	Reserved			

Table 29-8 provides a bit map for the VIC_STATUS register.

Table 29-8. The VIC_STATUS Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	HI_PRI_IRQ	R	0	The highest priority interrupt field. HI_PRI_IRQ contains the IRQ number of the active interrupt with the highest RIL. When there is no active interrupt (IP is 0), reading from this field returns 0. When the daisy chain input is enabled and it is the highest priority interrupt, then the value read from this field is 32. Bit 5 always reads back 0 when the daisy chain input is not present.
6:30	Reserved			
31	IP	R	0	The interrupt pending field. IP indicates when there is an interrupt ready to be serviced. A 1 indicates an interrupt is pending; a 0 indicates no interrupt is pending.

Device Support

The VIC core supports all Altera device families currently supported by the Quartus® II software.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the VIC core in SOPC Builder to add the core to a system.

Generation-time parameters control the features present in the hardware. Table 29-9 lists and describes the parameters you can configure.

Table 29-9. Parameters for VIC Core

Parameter	Legal Values	Description
Number of interrupts	1 – 32	Specifies the number of <code>irq_input</code> interrupt interfaces.
RIL width	1 – 6	Specifies the bit width of the requested interrupt level.
Daisy chain enable	True / False	Specifies whether or not to include an input interface for daisy chaining VICs together.

Because multiple VICs can exist in a single system, SOPC Builder assigns a unique interrupt controller identification number to each VIC generated.

Keep the following considerations in mind when connecting the core in your SOPC Builder system:

- The CSR access interface (`csr_access`) connects to a data master port on your processor.
- The daisy chain input interface (`interrupt_controller_in`) is only visible when the daisy chain enable option is on.
- The interrupt controller output interface (`interrupt_controller_out`) connects either to the EIC port of your processor, or to another VIC's daisy chain input interface (`interrupt_controller_in`).
- For SOPC Builder interoperability, the VIC core includes an Avalon-MM master port. This master interface is not used to access memory or peripherals. Its purpose is to allow peripheral interrupts to connect to the VIC in SOPC Builder. The port must be connected to an Avalon-MM slave to create a valid SOPC Builder system. Then at system generation time, the unused master port is removed during optimization. The most simple solution is to connect the master port directly into the CSR access interface (`csr_access`).
- SOPC Builder automatically connects interrupt sources when instantiating components. When using the provided HAL device driver for the VIC, daisy chaining multiple VICs in a system requires that each interrupt source is connected to exactly one VIC. You need to manually remove any extra connections.

Altera HAL Software Programming Model

The Altera-provided driver implements a HAL device driver that integrates with a HAL board support package (BSP) for Nios II systems. HAL users should access the VIC core via the familiar HAL API.

Software Files

The VIC driver includes the following software files. These files provide low-level access to the hardware and drivers that integrate with the Nios II HAL BSP. Application developers should not modify these files.

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

- **altera_vic_funnel.h, altera_vic_irq.h, altera_vic_irq.h, altera_vic_irq_init.h**—Define the prototypes and macros necessary for the VIC driver.
- **altera_vic.c, altera_vic_irq_init.c, altera_vic_isr_register.c, altera_vic_sw_intr.c, altera_vic_set_level.c, altera_vic_funnel_non_preemptive_nmi.S, altera_vic_funnel_non_preemptive.S, and altera_vic_funnel_preemptive.S**—Provide the code that implements the VIC driver.
- **altera_<name>_vector_tbl.S**—Provides a vector table file for each VIC in the system. The BSP generator creates these files.

Macros

Macros to access all of the registers are defined in **altera_vic_regs.h**. For example, this file includes macros to access the INT_CONFIG register, including the following macros:

```
#define IOADDR_ALTERA_VIC_INT_CONFIG(base, irq)
    __IO_CALC_ADDRESS_NATIVE(base, irq)
#define IORD_ALTERA_VIC_INT_CONFIG(base, irq)      IORD(base, irq)
#define IOWR_ALTERA_VIC_INT_CONFIG(base, irq, data) IOWR(base, irq,
data)
#define ALTERA_VIC_INT_CONFIG_RIL_MSK (0x3f)
#define ALTERA_VIC_INT_CONFIG_RIL_OFST (0)
#define ALTERA_VIC_INT_CONFIG_RNMI_MSK (0x40)
#define ALTERA_VIC_INT_CONFIG_RNMI_OFST (6)
#define ALTERA_VIC_INT_CONFIG_RRS_MSK (0x1f80)
#define ALTERA_VIC_INT_CONFIG_RRS_OFST (7)
```

For a complete list of predefined macros and utilities to access the VIC hardware, refer to the following files:

- `<install_dir>\ip\altera\altera_vectored_interrupt_controller\top\inc\altera_vic_regs.h`
- `<install_dir>\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_funnel.h`
- `<install_dir>\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_irq.h`

Data Structure

Figure 29-4 shows the data structure for the device.

Figure 29-4. Device Data Structure

```

#define ALT_VIC_MAX_INTR_PORTS          (32)

typedef struct alt_vic_dev
{
    void          *base;                /* Base address of VIC */
    alt_u32       intr_controller_id;   /* Interrupt controller ID */
    alt_u32       num_of_intr_ports;    /* Number of interrupt ports */
    alt_u32       ril_width;           /* RIL width */
    alt_u32       daisy_chain_present; /* Daisy-chain input present */
    alt_u32       vec_size;            /* Vector size */
    void          *vec_addr;           /* Vector table base address */
    alt_u32       int_config[ALT_VIC_MAX_INTR_PORTS]; /* INT_CONFIG settings
                                                    for each interrupt */
} alt_vic_dev;

```

VIC API

The VIC device driver provides all the routines required of an Altera HAL external interrupt controller (EIC) device driver. The following functions are required by the Altera Nios II enhanced HAL interrupt API:

- `alt_ic_isr_register ()`
- `alt_ic_irq_enable()`
- `alt_ic_irq_disable()`
- `alt_ic_irq_enabled()`

These functions write to the register map to change the setting or read from the register map to check the status of the VIC component thru a memory-mapped address.



For detailed descriptions of these functions, refer to the [HAL API Reference](#) chapter of the *Nios II Software Developer's Handbook*.

[Table 29-10](#) lists the API functions specific to the VIC core and briefly describes each. Details of each function follow the table.

Table 29-10. Function List

Name	Description
<code>alt_vic_sw_interrupt_set()</code>	Sets the corresponding bit in the <code>SW_INTERRUPT</code> register to enable a given interrupt via software.
<code>alt_vic_sw_interrupt_clear()</code>	Clears the corresponding bit in the <code>SW_INTERRUPT</code> register to disable a given interrupt via software.
<code>alt_vic_sw_interrupt_status()</code>	Reads the status of the <code>SW_INTERRUPT</code> register for a given interrupt.
<code>alt_vic_irq_set_level()</code>	Sets the interrupt level for a given interrupt.

`alt_vic_sw_interrupt_set()`

Prototype: `int alt_vic_sw_interrupt_set(alt_u32 ic_id, alt_u32 irq)`

Thread-safe: No

Available from ISR: No

Include: **altera_vic_irq.h, altera_vic_regs.h**

Parameters: *ic_id*—the interrupt controller identification number as defined in **system.h**
irq—the interrupt value as defined in **system.h**

Returns: Returns zero if successful; otherwise non-zero for one or more of the following reasons:

- The value in *ic_id* is invalid
- The value in *irq* is invalid

Description: Triggers a single software interrupt

alt_vic_sw_interrupt_clear()

Prototype: int alt_vic_sw_interrupt_clear(alt_u32 ic_id, alt_u32 irq)

Thread-safe: No

Available from ISR: Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.

Include: **altera_vic_irq.h, altera_vic_regs.h**

Parameters: *ic_id*—the interrupt controller identification number as defined in **system.h**
irq—the interrupt value as defined in **system.h**

Returns: Returns zero if successful; otherwise non-zero for one or more of the following reasons:

- The value in *ic_id* is invalid
- The value in *irq* is invalid

Description: Clears a single software interrupt

alt_vic_sw_interrupt_status()

Prototype: alt_u32 alt_vic_sw_interrupt_status(alt_u32 ic_id, alt_u32 irq)

Thread-safe: No

Available from ISR: Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.

Include: **altera_vic_irq.h, altera_vic_regs.h**

Parameters: *ic_id*—the interrupt controller identification number as defined in **system.h**
irq—the interrupt value as defined in **system.h**

Returns: Returns non-zero if the corresponding software trigger interrupt is active; otherwise zero for one or more of the following reasons:

- The corresponding software trigger interrupt is disabled
- The value in *ic_id* is invalid
- The value in *irq* is invalid

Description: Checks the software interrupt status for a single interrupt

alt_vic_irq_set_level()

Prototype:	int alt_vic_irq_set_level(alt_u32 ic_id, alt_u32 irq, alt_u32 level)
Thread-safe:	No
Available from ISR:	No
Include:	altera_vic_irq.h, altera_vic_regs.h
Parameters:	<p>ic_id—the interrupt controller identification number as defined in system.h</p> <p>irq—the interrupt value as defined in system.h</p> <p>level—the interrupt level to set</p>
Returns:	<p>Returns zero if successful; otherwise non-zero for one or more of the following reasons:</p> <ul style="list-style-type: none"> ■ The value in ic_id is invalid ■ The value in irq is invalid ■ The value in level is invalid
Description:	<p>Sets the interrupt level for a single interrupt.</p> <p>Altera recommends setting the interrupt level only to zero to disable the interrupt or to the original value specified in your BSP. Writing any other value could violate the overlapping register set, priority level, and other design rules. Refer to “VIC BSP Design Rules for Altera Hal Implementation” on page 29-18 for more information.</p>

Run-time Initialization

During system initialization, software configures the each VIC instance's control registers using settings specified in the BSP. The RIL, RRS, and RNMI fields are written into the interrupt configuration register of each interrupt port in each VIC. All interrupts are disabled until other software registers a handler using the `alt_ic_isr_register()` API.

Board Support Package

The BSP you generate for your Nios II system provides access to the hardware in your system, including the VIC. The VIC driver includes scripts that the BSP generator calls to get default interrupt settings and to validate settings during BSP generation. The Nios II BSP Editor provides a mechanism to edit these settings and generate a BSP for your SOPC Builder design.

The generator produces a vector table file for each VIC in the system, named **altera_<name>_vector.tbl.S**. The vector table's source path is added to the BSP Makefile for compilation along with other VIC driver source code. Its contents are based on the BSP settings for each VIC's interrupt ports.

VIC BSP Settings

The VIC driver scripts provide settings to the BSP. The number and naming of these settings depends on your hardware system's configuration, specifically, the number of optional shadow register sets in the Nios II processor, the number of VIC controllers in the system, and the number of interrupt ports each VIC has.

Certain settings apply to all VIC instances in the system, while others apply to a specific VIC instance. Settings that apply to each interrupt port apply only to the specified interrupt port number on that VIC instance.

The remainder of this section lists details and descriptions of each VIC BSP setting.

altera_vic_driver.enable_preemption

Identifier: ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED
Type: BooleanDefineOnly
Default value: 1 when all components connected to the VICs support preemption. 0 when any of the connected components don't support preemption.
Destination file: **system.h**
Description: Enables global interrupt preemption (nesting). When enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED` is defined in **system.h**.
Two types of ISR preemption are available. This setting must be enabled along with other settings to enable specific types of preemption.
All preemption settings are dependant on whether the device drivers in your BSP support interrupt preemption. For more information about preemption, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.
Occurs: Once per VIC

altera_vic_driver.enable_preemption_into_new_register_set

Identifier: ALTERA_VIC_DRIVER_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED
Type: BooleanDefineOnly
Default value: 0
Destination file: **system.h**
Description: Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, and that higher priority interrupt uses a different register set than the interrupt currently being serviced.
When this setting is enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED` is defined in **system.h** and the Nios II `config.ANI` (automatic nested interrupts) bit is asserted during system software initialization.
Use this setting to limit interrupt preemption to higher priority (RIL) interrupts that use a different register set than a lower priority interrupt that might be executing. This setting allows you to support some preemption while maintaining the lowest possible interrupt response time. However, this setting does not allow an interrupt at a higher priority (RIL) to preempt a lower priority interrupt if the higher priority interrupt is assigned to the same register set as the lower priority interrupt.
Occurs: Once per VIC

altera_vic_driver.enable_preemption_rs_<n>

Identifier: ALTERA_VIC_DRIVER_ENABLE_PREEMPTION_RS_<n>
Type: Boolean
Default value: 0

Destination file: `system.h`

Description: Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, for all interrupts that target the specified register set number.

When this setting is enabled (set to 1), the vector table for each VIC utilizes a special interrupt funnel that manages preemption. All interrupts on all VIC instances assigned to that register set then use this funnel.

When a higher priority interrupt preempts a lower priority interrupt running in the same register set, the interrupt funnel detects this condition and saves the processor registers to the stack before calling the higher priority ISR. The funnel code restores registers and allows the lower priority ISR to continue running once the higher priority ISR completes.

Because this funnel contains additional overhead, enabling this setting increases interrupt response time substantially for all interrupts that target a register set where this type of preemption is enabled.

Use this setting if you must guarantee that a higher priority interrupt preempts a lower priority interrupt, and you assigned multiple interrupts at different priorities to the same Nios II shadow register set.

Occurs: Per register set; `<n>` refers to the register set number.

`altera_vic_driver.linker_section`

Identifier: `ALTERA_VIC_DRIVER_LINKER_SECTION`

Type: `UnquotedString`

Default value: `.text`

Destination file: `system.h`

Description: Specifies the linker section that each VIC's generated vector table and each interrupt funnel link to. The memory device that the specified linker section is mapped to must be connected to both the Nios II instruction and data masters in your SOPC Builder system.

Use this setting to link performance-critical code into faster memory. For example, if your system's code is in DRAM and you have an on-chip or tightly-coupled memory interface for interrupt handling code, assigning the VIC driver linker section to a section in that memory improves interrupt response time.

For more information about linker sections and the Nios II BSP Editor, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Occurs: Once per VIC

`altera_vic_driver.<name>.vec_size`

Identifier: `<name>_VEC_SIZE`

Type: `DecimalNumber`

Default value: `16`

Destination file: `system.h`

- Description:** Specifies the number of bytes in each vector table entry. Legal values are 16, 32, 64, 128, 256, and 512.
- The generated VIC vector tables in the BSP require a minimum of 16 bytes per entry.
- If you intend to write your own vector table or locate your ISR at the vector address, you can use a larger size.
- The vector table's total size is equal to the number of interrupt ports on the VIC instance multiplied by the vector table entry size specified in this setting.
- Occurs:** Per instance; *<name>* refers to the component name you assign in SOPC Builder.

`altera_vic_driver.<name>.irq<n>_rrs`

- Identifier:** ALTERA_VIC_DRIVER_<name>_IRQ<n>_RRS
- Type:** DecimalNumber
- Default value:** Refer to “Default Settings for RRS and RIL”.
- Destination file:** `system.h`
- Description:** Specifies the RRS for the interrupt connected to the corresponding port. Legal values are 1 to the number of shadow register sets defined for the processor.
- Occurs:** Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

`altera_vic_driver.<name>.irq<n>_ril`

- Identifier:** ALTERA_VIC_DRIVER_<name>_IRQ<n>_RIL
- Type:** DecimalNumber
- Default value:** Refer to “Default Settings for RRS and RIL”.
- Destination file:** `system.h`
- Description:** Specifies the RIL for the interrupt connected to the corresponding port. Legal values are 0 to $2^{\text{RIL width}} - 1$.
- Occurs:** Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

`altera_vic_driver.<name>.irq<n>_rnmI`

- Identifier:** ALTERA_VIC_DRIVER_<name>_IRQ<n>_RNMI
- Type:** Boolean
- Default value:** 0
- Destination file:** `system.h`
- Description:** Specifies whether the interrupt port is a maskable or non-maskable interrupt (NMI). Legal values are 0 and 1. When set to 0, the port is maskable. NMIs cannot be disabled in hardware and there are several restrictions imposed for the RIL and RRS settings associated with any interrupt with NNI enabled.
- Occurs:** Per IRQ per instance; *<name>* refers to the VIC's name and *<n>* refers to the IRQ number that you assign in SOPC Builder. Refer to SOPC Builder to determine which IRQ numbers correspond to which components in your design.

Default Settings for RRS and RIL

The default assignment of RRS and RIL values for each interrupt assumes interrupt port 0 on the VIC instance attached to your processor is the highest priority interrupt, with successively lower priorities as the interrupt port number increases. Interrupt ports on other VIC instances connected through the first VIC's daisy chain interface are assigned successively lower priorities.

To make effective use of the VIC interrupt setting defaults, assign your highest priority interrupts to low interrupt port numbers on the VIC closest to the processor. Assign lower priority interrupts and interrupts that do not need exclusive access to a shadow register set, to higher interrupt port numbers, or to another daisy-chained VIC.

The following steps describe the algorithm for default RIL assignment:

1. The formula $2^{\text{RIL width}} - 1$ is used to calculate the maximum RIL value.
2. interrupt port 0 on the VIC connected to the processor is assigned the highest possible RIL.
3. The RIL value is decremented and assigned to each subsequent interrupt port in succession until the RIL value is 1.
4. The RILs for all remaining interrupt ports on all remaining VICs in the chain are assigned 1.

The following steps describe the algorithm for default RRS assignment:

1. The highest register set number is assigned to the interrupt with the highest priority.
2. Each subsequent interrupt is assigned using the same method as the default RIL assignment.

For example, consider a system with two VICs, VIC0 and VIC1. Each VIC has an RIL width of 3, and each has 4 interrupt ports. VIC0 is connected to the processor and VIC1 to the daisy chain interface on VIC0. The processor has 3 shadow register sets. [Table 29-11](#) shows the default RRS and RIL assignments for this example.

Table 29-11. Default RRS and RIL Assignment Example

VIC	IRQ	RRS	RIL
0	0	3	7
0	1	2	6
0	2	1	5
0	3	1	4
1	0	1	3
1	1	1	2
1	2	1	1
1	3	1	1

VIC BSP Design Rules for Altera Hal Implementation

The VIC BSP settings allow for a large number of combinations. This list describes some basic design rules to follow to ensure a functional BSP:

- Each component's interrupt interface in your system should only be connected to one VIC instance per processor.
- The number of shadow register sets for the processor must be greater than zero.
- RRS values must always be greater than zero and less than or equal to the number of shadow register sets.
- RIL values must always be greater than zero and less than or equal to the maximum RIL.
- All RILs assigned to a register set must be sequential to avoid a higher priority interrupt overwriting contents of a register set being used by a lower priority interrupt.



The Nios II BSP Editor uses the term "overlap condition" to refer to nonsequential RIL assignments.

- NMI's cannot share register sets with maskable interrupts.
- NMI's must have RILs set to a number equal to or greater than the highest RIL of any maskable interrupt. When equal, the NMI's must have a lower logical interrupt port number than any maskable interrupt.
- The vector table and funnel code section's memory device must connect to a data master and an instruction master.
- NMI's must use funnels with preemption disabled.
- When global preemption is disabled, enabling preemption into a new register set or per-register-set preemption might produce unpredictable results. Be sure that all interrupt service routines (ISR) used by the register set support preemption.
- Enabling register set preemption for register sets with peripherals that don't support preemption might result in unpredictable behavior.

RTOS Considerations

BSPs configured to use a real time operating system (RTOS) might have additional software linked into the HAL interrupt funnel code using the `ALT_OS_INT_ENTER` and `ALT_OS_INT_EXIT` macros. The exact nature and overhead of this code is RTOS-specific. Additional code adds to interrupt response and recovery time. Refer to your RTOS documentation to determine if such code is necessary.

Referenced Documents

This chapter references the following documents:


- *Avalon Interface Specifications*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 29-12 shows the revision history for this chapter.

Table 29-12. Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	Initial release.	—

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