SoC Blockset™

Reference

MATLAB&SIMULINK®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

SoC Blockset™ Reference

Trademarks

Patents

Revision History

Contents

3

Functions

4

Objects

5

Tools — Alphabetical List

Blocks

SoC Bus Creator

Convert control signals to bus
Library:　　　　SoC Blockset / Hardware Logic Connectivity
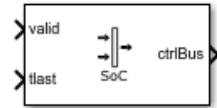
Description

The SoC Bus Creator block combines a set of signals into a bus. The block accepts control signals and outputs a bus.

You can configure this block to support multiple protocol interface types. Parameter and port configurations for this block vary based on your desired protocol interface type and mode of operation, as outlined in this table.

Protocol Interface Type	Mode of Operation	Parameter Configuration	Enabled Input Ports
Data stream	Read data stream	Set **Control protocol** to Data stream and **Control type** to Ready.	**ready**
	Write data stream	Set **Control protocol** to Data stream and **Control type** to Valid.	**valid**
			tlast
Pixel stream	Read video stream	Set **Control protocol** to Pixel stream and **Control type** to Ready.	**ready**
	Write video stream	Set **Control protocol** to Pixel stream and **Control type** to Valid.	**hStart**
			hEnd
			vStart
			vEnd
			valid

Protocol Interface Type	Mode of Operation	Parameter Configuration	Enabled Input Ports
	Read video stream with frame sync	Set **Control protocol** to `Pixel stream` and **Control type** to `Ready frame with sync`.	**ready**
			fsync
Random access read	Read data	Set **Control protocol** to `Random access read` and **Control type** to `Ready`.	**rd_addr**
			rd_len
			rd_avalid
			rd_dready
Random access write	Write data	Set **Control protocol** to `Random access write` and **Control type** to `Valid`.	**wr_addr**
			wr_len
			wr_valid

Ports

Input

valid — Valid control signal
Boolean scalar

Valid control signal, specified as a scalar. You can use this port for data stream and pixel stream protocols only.

Dependencies

To enable this port, set the **Control protocol** parameter to either `Data stream` or `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

tlast — Indication of end of data packet
Boolean scalar

Indication of end of the data packet, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Data stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

ready — Ready control signal
Boolean scalar

Ready control signal, specified as a Boolean scalar. This port is available for `Data stream` and `Pixel stream` control protocols.

Dependencies

To enable this port, set the **Control protocol** parameter to either `Data stream` or `Pixel stream` and the **Control type** parameter to `Ready` or `Ready with frame sync`.

Data Types: `Boolean`

hStart — First pixel in horizontal line of frame
Boolean scalar

First pixel in a horizontal line of a frame, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

hEnd — Last pixel in horizontal line of frame
Boolean scalar

Last pixel in a horizontal line of a frame, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

`vStart` — First pixel in first (top) line of frame
Boolean scalar

First pixel in the first (top) line of a frame, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

`vEnd` — Last pixel in last (bottom) line of frame
Boolean scalar

Last pixel in the last (bottom) line of a frame, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

`fsync` — Frame synchronization
Boolean scalar

Frame synchronization, specified as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Ready with frame sync`.

Data Types: `Boolean`

`rd_addr` — Reader address
scalar

Reader address, specified as a scalar. It is the starting address for the read transaction that is sampled at the first cycle of the transaction.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `uint32`

rd_len — Reader data length
scalar

Reader data length, specified as a scalar. It means the number of data values that you want to read, sampled at the first cycle of the transaction.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `uint32`

rd_avalid — Reader valid status
Boolean scalar

Reader valid status, specified as a Boolean scalar. It indicates whether the read request is valid.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `Boolean`

rd_dready — Reader ready status
Boolean scalar

Reader ready status, specified as a Boolean scalar. It indicates when the hardware logic can start accepting data.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `Boolean`

wr_addr — Writer address
scalar

Specify the starting address to which the hardware writes.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `uint32`

wr_len — Writer data length
scalar

Specify the number of data elements in the write transaction.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `uint32`

wr_valid — Writer valid data
Boolean scalar

Writer valid data, specified as a scalar. It indicates the data signal sampled at the output is valid.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `Boolean`

Output

ctrlBus — Output control bus
bus

Output control bus, returned as a bus.

The data type of the output control bus depends on the values of the **Control protocol** and **Control type** parameters.

Parameter Configuration	Output Data Type
Set **Control protocol** to `Data stream` and **Control type** to `Ready`.	`StreamS2MBusObj`
Set **Control protocol** to `Data stream` and **Control type** to `Valid`.	`StreamM2SBusObj`

Parameter Configuration	Output Data Type
Set **Control protocol** to Pixel stream and **Control type** to Ready.	StreamVideoS2MBusObj
Set **Control protocol** to Pixel stream and **Control type** to Valid.	pixelcontrol
Set **Control protocol** to Pixel stream and **Control type** to Ready frame with sync.	StreamvideoFsyncS2MBusObj
Set **Control protocol** to Random access read and **Control type** to Ready.	ReadControlM2SBusObj
Set **Control protocol** to Random access write and **Control type** to Valid.	WriteControlM2SBusObj

Data Types: StreamS2MBusObj | StreamM2SBusObj | StreamVideoS2MBusObj | pixelcontrol | StreamvideoFsyncS2MBusObj | ReadControlM2SBusObj | WriteControlM2SBusObj

Parameters

Control protocol — Protocol interface selection
Data stream (default) | Pixel stream | Random access read | Random access write

Specify the protocol interface as one of these values:

- Data stream — Use this protocol if you require AXI4 data stream.
- Pixel stream — Use this protocol if you require AXI4 video stream.
- Random access read — Use this protocol if you require AXI4 read.
- Random access write — Use this protocol if you require AXI4 write.

The input ports of the block vary based on the type of **Control protocol** and **Control type** that you select. For more details, see "Description" on page 1-2.

Control type — Control type selection
Valid (default) | Ready | Ready with frame sync

Specify the type of control.

To enable the `Ready with frame sync` option, set the **Control protocol** parameter to `Pixel stream`.

The input ports of the block vary based on the type of **Control protocol** and **Control type** that you select. For more details, see "Description" on page 1-2.

Extended Capabilities

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

SoC Bus Selector

Introduced in R2019a

SoC Bus Selector

Convert bus to control signals
Library: SoC Blockset / Hardware Logic Connectivity
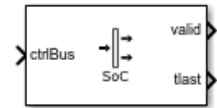
Description

The SoC Bus Selector block converts a set of control signals from a bus. The block accepts a bus and outputs control signals.

You can configure this block to support multiple protocol interface types. Parameter and port configurations for this block vary based on your desired protocol interface type and mode of operation, as outlined in this table.

Protocol Interface Type	Mode of Operation	Parameter Configuration	Enabled Output Ports
Data stream	Read stream data	Set **Control protocol** to `Data stream` and **Control type** to `Valid`.	**valid**
			tlast
	Write stream data	Set **Control protocol** to `Data stream` and **Control type** to `Ready`.	**ready**
Pixel stream	Read video stream	Set **Control protocol** to `Pixel stream` and **Control type** to `Valid`.	**hStart**
			hEnd
			vStart
			vEnd
			valid

Protocol Interface Type	Mode of Operation	Parameter Configuration	Enabled Output Ports
	Write video stream	Set **Control protocol** to `Pixel stream` and **Control type** to `Ready`.	**ready**
Random access read	Read data	Set **Control protocol** to `Random access read` and **Control type** to `Valid`.	**rd_aready**
			rd_dvalid
Random access write	Write data	Set **Control protocol** to `Random access write` and **Control type** to `Ready`.	**wr_ready**
			wr_bvalid
			wr_complete

Ports

Input

ctrlBus — Input control bus
bus

Input control bus, specified as a bus.

The data type of the input control bus depends on the values of the **Control protocol** and **Control type** parameters.

Parameter Configuration	Input Data Type
Set **Control protocol** to `Data stream` and **Control type** to `Valid`.	`StreamM2SBusObj`
Set **Control protocol** to `Data stream` and **Control type** to `Ready`.	`StreamS2MBusObj`
Set **Control protocol** to `Pixel stream` and **Control type** to `Valid`.	`pixelcontrol`

Parameter Configuration	Input Data Type
Set **Control protocol** to Pixel stream and **Control type** to Ready.	StreamVideoS2MBusObj
Set **Control protocol** to Random access read and **Control type** to Valid.	ReadControlS2MBusObj
Set **Control protocol** to Random access write and **Control type** to Ready.	WriteControlS2MBusObj

Data Types: StreamM2SBusObj | StreamS2MBusObj | pixelcontrol | StreamVideoS2MBusObj | ReadControlS2MBusObj | WriteControlS2MBusObj

Output

valid — Valid control signal
Boolean scalar

Valid control signal, returned as a scalar. You can use this port for data stream and pixel stream protocols only.

Dependencies

To enable this port, set the **Control protocol** parameter to either Data stream or Pixel stream and the **Control type** parameter to Valid.

Data Types: Boolean

tlast — Indication of end of data packet
Boolean scalar

Indication of end of the data packet, returned as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to Data stream and the **Control type** parameter to Valid.

Data Types: Boolean

ready — Ready control signal
Boolean scalar

Ready control signal, returned as a Boolean scalar. This port is available for `Data stream` and `Pixel stream` control protocols.

Dependencies

To enable this port, set the **Control protocol** parameter to either `Data stream` or `Pixel stream` and the **Control type** parameter to `Ready`.

Data Types: `Boolean`

hStart — First pixel in horizontal line of frame
Boolean scalar

First pixel in a horizontal line of a frame, returned as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

hEnd — Last pixel in horizontal line of frame
Boolean scalar

Last pixel in a horizontal line of a frame, returned as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

vStart — First pixel in first (top) line of frame
Boolean scalar

First pixel in the first (top) line of a frame, returned as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

vEnd — Last pixel in last (bottom) line of frame
Boolean scalar

Last pixel in the last (bottom) line of a frame, returned as a Boolean scalar.

Dependencies

To enable this port, set the **Control protocol** parameter to `Pixel stream` and the **Control type** parameter to `Valid`.

Data Types: `Boolean`

rd_aready — Accept read requests
Boolean scalar

Accept read requests, returned as a scalar. It indicates when to accept read requests.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `Boolean`

rd_dvalid — Read request valid
Boolean scalar

Read request valid, returned as a Boolean scalar. It is the control signal that indicates the data returned from the read request is valid.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access read`.

Data Types: `Boolean`

wr_ready — Write ready signal
Boolean scalar

Write ready signal, returned as a Boolean scalar. It corresponds to the backpressure from the slave IP core or external memory. When this value is `1` (`high`), it indicates that data can be sent. When this value is `0` (`low`), it indicates that the hardware logic must stop sending data within one clock cycle.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `Boolean`

`wr_bvalid` — Write valid signal
Boolean scalar

Write valid signal, returned as a Boolean scalar. It is the response signal from the slave IP core that you can use for diagnosis purposes. This value becomes `1` (`high`) after the AXI4 interconnect accepts each burst transaction.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `Boolean`

`wr_complete` — Write transaction complete
Boolean scalar

Write transaction complete, specified as a Boolean scalar. It is the control signal that when remains high for one clock cycle indicates that the write transaction has completed. This signal asserts at the last `wr_bvalid` of the burst.

Dependencies

To enable this port, set the **Control protocol** parameter to `Random access write`.

Data Types: `Boolean`

Parameters

`Control protocol` — Protocol interface selection
`Data stream` (default) | `Pixel stream` | `Random access read` | `Random access write`

Specify the protocol interface as one of these values:

- `Data stream` — Use this protocol if you require AXI4 data stream.
- `Pixel stream` — Use this protocol if you require AXI4 video stream.
- `Random access read` — Use this protocol if you require AXI4 read.
- `Random access write` — Use this protocol if you require AXI4 write.

The output ports of the block vary based on the type of **Control protocol** and **Control type** that you select. For more details, see "Description" on page 1-10.

`Control type` — Control type selection
`Valid` (default) | `Ready`

Specify the type of control.

The output ports of the block vary based on the type of **Control protocol** and **Control type** that you select. For more details, see "Description" on page 1-10.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
SoC Bus Creator

Introduced in R2019a

Stream FIFO

Control backpressure between hardware logic and upstream data interface
Library: SoC Blockset / Hardware Logic Connectivity
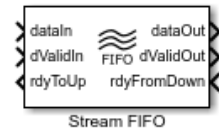
Description

The Stream FIFO block controls the backpressure from the hardware logic to the upstream data interface. It also controls the flow between the upstream and downstream data interfaces of the hardware logic. Integrate this block as a configurable first-in, first-out (FIFO) block for AXI4 data stream applications. The block enables you to configure its depth and set its almost full threshold value.

Ports

Input

dataIn — Input stream data
scalar

Input stream data from the data source. Specify this value as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

dValidIn — Indication of valid input stream data
Boolean scalar

Control signal that indicates if the input stream data from the data source is valid. When this value is 1 (true), the block accepts the values on the **dataIn** port. When this value is 0 (false), the block ignores the values on the **dataIn** port.

Data Types: `Boolean`

rdyFromDown — Ready signal from downstream interface
Boolean scalar

Control signal that indicates if the block can send stream data to the downstream interface. When this value is 1 (true), the downstream interface is ready, and the block can send the stream data. When this value is 0 (false), the downstream interface is not ready, and the block cannot send the stream data.

Data Types: Boolean

Output

dataOut — Output stream data
scalar

Output stream data to the downstream interface. The data type of this output data is the same as the data type of the input data.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | fixed point

dValidOut — Indication of valid output stream data
Boolean scalar

Control signal that indicates if the output stream data is valid. When this value is 1 (true), the output stream data on the **dataOut** port is valid. When this value is 0 (false), the output stream data on the **dataOut** port is not valid.

Data Types: Boolean

rdyToUp — Ready signal to upstream interface
Boolean scalar

Control signal that indicates if the block is ready to receive stream data from the upstream interface. When this value is 1 (true), the block is ready to accept stream data from the upstream interface. When this value is 0 (false), the block is not ready to accept stream data from the upstream interface.

Data Types: Boolean

Parameters

Depth of FIFO — FIFO depth
16 (default) | positive integer

Specify the depth of the FIFO. This value must be a positive integer and is the maximum number of entries that can be buffered before data gets dropped.

Almost full threshold — Almost full threshold value
8 (default) | positive integer

Specify a value that asserts a back-pressure signal from the block to the data source.

To avoid dropping data, set a value allowing the data source enough time to react to backpressure. This value must be a positive integer and smaller than the FIFO depth.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Video Stream FIFO

Introduced in R2019a

Video Stream FIFO

Control backpressure between hardware logic and upstream video interface
Library: SoC Blockset / Hardware Logic Connectivity

Description

The Video Stream FIFO block controls the back-pressure from the hardware logic to the upstream video interface. It also controls the flow between the upstream and downstream pixel data interfaces of hardware logic. Integrate this block as a configurable first-in, first-out (FIFO) block for AXI4 video stream applications. The block enables you to configure its depth and set its almost full threshold value.

Ports

Input

pixelIn — Input pixel data
scalar

Input pixel data from the data source. Specify this value as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

ctrlIn — Control signals accompanying input pixel data
`pixelcontrol` bus

Control signals accompanying the pixel stream, specified as a `pixelcontrol` bus containing five signals. The signals describe the validity of the pixel and its location in the frame.

Data Types: `pixelcontrol`

`rdyFromDown` — Ready signal from downstream interface
Boolean scalar

Control signal that indicates if the block can send pixel data to the downstream interface. When this value is 1 (true), the downstream interface is ready, and the block can send the pixel data. When this value is 0 (false), the downstream interface is not ready, and the block cannot send the pixel data.

Data Types: `Boolean`

Output

`pixelOut` — Output pixel data
scalar

Output pixel data to the downstream interface. The data type of this output data is the same as the data type of the input data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

`ctrlOut` — Control signals accompanying output pixel data
`pixelcontrol` bus

Control signals accompanying output pixel stream, returned as a `pixelcontrol` bus containing five signals. The signals describe the validity of the pixel and its location in the frame.

Data Types: `pixelcontrol`

`rdyToUp` — Ready signal to upstream interface
Boolean scalar

Control signal that indicates if the block is ready to receive pixel data from the upstream interface. When this value is 1 (true), the block is ready to accept pixel data from the upstream interface. When this value is 0 (false), the block is not ready to accept pixel data from the upstream interface.

Data Types: `Boolean`

Parameters

Depth of FIFO — FIFO depth
16 (default) | positive integer

Specify the depth of the FIFO. This value must be a positive scalar integer and is the maximum number of entries that can be buffered before data gets dropped.

Almost full threshold — Almost full threshold value
8 (default) | positive integer

Specify a value that asserts a back-pressure signal from the block to the data source.

To avoid dropping data, set a value allowing the data source enough time to react to backpressure. This value must be a positive integer and smaller than the FIFO depth.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Stream FIFO

Introduced in R2019a

Video Stream Connector

Connect two IPs with video streaming interfaces
Library: SoC Blockset / Hardware Logic Connectivity
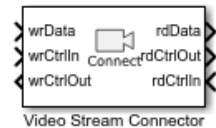
Description

The Stream Connector block connects two IPs with video streaming interfaces. Use this block in the FPGA model of an SoC application to connect two IPs.

Ports

Input

wrData — Input video data
scalar

Input video data from the data source. Specify this value as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

wrCtrlIn — Input control signals accompanying pixel stream
`pixelControl` bus

Control signals accompanying the pixel stream, specified as a `pixelcontrol` bus containing five signals. The signals describe the validity of the pixel and its location in the frame. For additional information about the `pixelcontrol` bus type, see "AXI4-Stream Video Interface".

Data Types: `pixelcontrol`

rdCtrlIn — Ready signal from downstream interface
boolean scalar

Control signal that indicates if the block can send video data to downstream interface. When this value is (true), the downstream block is ready to receive data.

Output

`rdData` — Output video data
scalar

Output video data to the downstream destination IP.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`rdCtrlOut` — Output control signals accompanying output pixel stream
`pixelcontrol` bus

Control signals accompanying the output video data, specified as a `pixelcontrol` bus containing five signals. The signals describe the validity of the pixel and its location in the frame.

Data Types: `pixelcontrol`

`wrCtrlOut` — Ready signal to the upstream interface
boolean scalar

Control signal that indicates that the block can receive stream data from upstream interface.

Data Types: `Boolean`

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Stream Connector

Introduced in R2019a

Stream Connector

Connect two IPs with data streaming interfaces
Library: SoC Blockset / Hardware Logic Connectivity
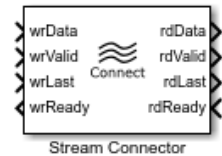
Description

The Stream Connector block connects two IPs with data streaming interfaces. Use this block in the FPGA model of an SoC application to connect two IPs.

Ports

Input

wrData — Input stream data
scalar

Input stream data from the data source. Specify this value as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

wrValid — Indication of valid input stream data
boolean scalar

Control signal that indicates if the input data from the data source is valid. When this value is (true), the block accepts the values on the **wrData** port. When this value is (false), the block ignores the value on the **wrData** port.

Data Types: `Boolean`

wrLast — Indication of last beat in burst
boolean scalar

Control signal that indicates the last beat of data from the upstream IP.

Data Types: `Boolean`

rdReady — Ready signal from downstream interface
boolean scalar

Control signal that indicates if the block can send stream data to the downstream interface. When this value is (true), the downstream block is ready to receive data.

Data Types: `Boolean`

Output

rdData — Output stream data
scalar

Output stream data to the downstream destination IP.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

rdValid — Indication of valid output stream data
boolean scalar

Control signal that indicates if the output stream data is valid.

Data Types: `Boolean`

rdLast — Indicates last beat in burst
boolean scalar

Control signal that indicates that the output stream data now has last beat of burst data.

Data Types: `Boolean`

wrReady — Ready signal to upstream interface
boolean scalar

Control signal that indicates if the block can receive stream data from the upstream interface.

Data Types: `Boolean`

Extended Capabilities

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Video Stream Connector

Introduced in R2019a

DIP Switch

Connect signals attached to DIP switches on hardware board
Library: SoC Blockset / Hardware Logic I/O
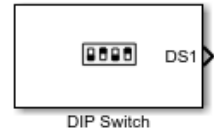
Description

The DIP Switch block controls the hardware logic. The hardware logic signals connected to a DIP Switch block are equivalent to the signals connected to the dual inline package (DIP) switches on the hardware board.

Ports

Input

DSIn*x* — Input signal
Boolean scalar

Input signal to control the hardware logic. Using this port, you can dynamically control the hardware logic during simulation at run time. Each DIP switch has a port, named **DSIn1** to **DSIn*x***, where *x* is **Number of DIP switches**.

Dependencies

To enable this port, set the **Specify DIP switches via** parameter to InputPort.

Data Types: Boolean

Output

DS*x* — Output signal
Boolean scalar

Output signal that returns the state of the switch. Each DIP switch has a port, named **DS1** to **DS***x*, where *x* is **Number of DIP switches**.

Data Types: `Boolean`

Parameters

`Hardware board` — View selected hardware
None (default) | Supported Xilinx® or Intel ® boards | Custom boards

This parameter is read-only. To choose a hardware board and configure board parameters, see "Hardware Implementation Pane" on page 2-2.

`View DIP switches location` — View DIP switches
button

To view a diagram with the location of the DIP switches on the selected hardware board, click the **View DIP switches location** button.

This button is enabled only when you select specific Xilinx or Intel boards. For more information about these boards, refer to "Supported Third-Party Tools and Hardware".

`IO logic` — IO logic indicator
None (default) | Active High | Active Low

This parameter is read-only. Indicates the IO logic level on the selected hardware board.

When the **IO logic** parameter is shown as **Active Low**, the DIP Switch block accepts and outputs active low signals when you set the **Specify DIP switches via** parameter to `InputPort` and outputs active low signals when you set the **Specify DIP switches via** parameter to `Dialog`. The block represents these port names prefixed with letter *n*. For example, **nDS1**.

`Specify DIP switches via` — DIP switch source
`Dialog` (default) | `InputPort`

To control the hardware logic by using the block parameters, select `Dialog`. To control the hardware logic from the input port, select `InputPort`.

`Number of DIP switches` — DIP switch selection
1 (default) | list of integers in the range [1, n]

To specify the required number of DIP switch ports, select a value from the **Number of DIP switches** list. *n* represents the number of available DIP switches on the specified hardware board. For example, if you select 3 from the list, the block shows three DIP switch ports.

To use only the *n*th DIP switch, set the **Number of DIP switches** parameter to *n* and terminate the unused DIP switch ports.

DS*n* — Selected DIP switches
Off (default) | On

To enable the *n*th DIP switch port, select On for the DS*n* parameter. *n* represents the number of available DIP switches on the specified hardware board.

Dependencies

To enable this parameter, set the **Specify push buttons via** parameter to Dialog.

Sample time — System sample time
-1 (default) | positive scalar

Specify the time interval a DIP switch toggles between On and Off.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
LED | Push Button

Introduced in R2019a

I2C Master

Configure and communicate with I2C slave device
Library: SoC Blockset / Hardware Logic I/O
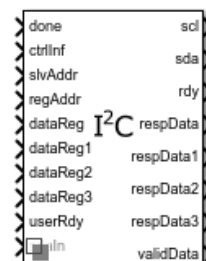
Description

The I2C Master block configures and communicates with an inter-integrated circuit communications (I2C) slave device connected to a field programmable gate array (FPGA). This block contains an I2C master controller with an AXI-Lite interface to perform the configuration.

The I2C Master block supports these features:

- AXI4-Lite interface support for configuration and access
- Single-master and multi-slave support
- 7-bit addressing support
- Multiple transmission speeds support, which includes these modes:

    - Standard mode (up to100 kHz)
    - Fast mode (up to 400 kHz)
    - Fast-plus mode (up to 1 MHz)
- Burst mode support with a maximum burst size of 16 bytes
- An HDL-IP compatible model with code generation capability

The block uses the AXI-Lite interface to configure and create a control path interface to communicate with an I2C slave device. The block enables you to perform simulation and generation processes separately. The hardware generated from the generation process

contains an AXI-Lite register interface and two hardware interfaces, serial clock (SCL) and serial data (SDA). SCL and SDA connect the I2C Master block and the slave device.

Each port represented in the block is an AXI-Lite register, except the **sdaIn**, **scl**, and **sda** ports. To communicate with the slave device, the AXI-Lite register interface configures the register information in the I2C Master block. This table contains the I2C Master AXI-Lite register information.

Address	Port and Register Name	Register Size in Bits	Operation Mode
0x100	**ctrlInf** — Control information	32	Write
0x104	**slvAddr** — Slave address	32	Write
0x108	**regAddr** — Register address	32	Write
0x10C	**dataReg** — First data register	32	Write
0x110	**dataReg1** — Second data register	32	Write
0x114	**dataReg2** — Third data register	32	Write
0x118	**dataReg3** — Fourth data register	32	Write
0x11C	**userRdy** — User-ready register	32	Write
0x120	**done** — Done register	32	Write
0x124	**rdy** — Ready register	32	Read
0x128	**respData** — First response data register	32	Read
0x12C	**respData1** — Second response data register	32	Read
0x130	**respData2** — Third response data register	32	Read

Address	Port and Register Name	Register Size in Bits	Operation Mode
0x134	**respData3** — Fourth response data register	32	Read
0x138	**validData** — Response data valid register	32	Read

Ports

Input

ctrlInf — Control information
scalar

Control information register that contains configuration information on how the block communicates with the slave device. Specify this value as a scalar, as described in this table. You can modify the configuration based on your requirement.

Bit	Purpose	Value Description
0	Set write or read mode.	To write to the slave-device register, set this value to 0. To read from the slave-device register, set this value to 1.
[2:1]	Set the size of the slave-device register address.	If the slave-device register address size is: • One byte (8 bits), set this value to 00 • Two bytes (16 bits), set this value to 01 • Three bytes (24 bits), set this value to 10 • Four bytes (32 bits), set this value to 11

Bit	Purpose	Value Description
[6:3]	Set the data size of the slave-device register.	If the slave-device register supports: • One byte of data, set this value to 0000 • Two bytes of data, set this value to 0001 • Three bytes of data, set this value to 0010 • Four bytes of data, set this value to 0011 • Five bytes of data, set this value to 0100 • Six bytes of data, set this value to 0101 • Seven bytes of data, set this value to 0110 • Eight bytes of data, set this value to 0111 • Nine bytes of data, set this value to 1000 • Ten bytes of data, set this value to 1001 • Eleven bytes of data, set this value to 1010 • Twelve bytes of data, set this value to 1011 • Thirteen bytes of data, set this value to 1100 • Fourteen bytes of data, set this value to 1101 • Fifteen bytes of data, set this value to 1110 • Sixteen bytes of data, set this value to 1111
7	Reserved	Reserved

Data Types: `uint8`

`slvAddr` — Slave address
scalar

Slave-address register that contains the address of the slave device, specified as a scalar.

Data Types: `uint32`

`regAddr` — Register address
scalar

Specify the register address of the slave device as a scalar.

Data Types: `uint32`

`dataReg` — First data register
scalar

First data register that the block uses to write the first 4 bytes of data to the slave-device register, specified as a scalar. This register includes the least significant bit (LSB).

Data Types: `uint32`

`dataReg1` — Second data register
scalar

Second data register that the block uses to write the second set of 4 bytes of data to the slave-device register, specified as a scalar.

Data Types: `uint32`

`dataReg2` — Third data register
scalar

Third data register that the block uses to write the third set of 4 bytes of data to the slave-device register, specified as a scalar.

Data Types: `uint32`

`dataReg3` — Fourth data register
scalar

Fourth data register that the block uses to write the fourth set of 4 bytes of data to the slave-device register, specified as a scalar. This register includes the most significant bit (MSB).

Data Types: `uint32`

userRdy — User-ready signal
`Boolean` scalar

User-ready signal, specified as a Boolean scalar. When this value is `1` (true), the user is ready to read the response data from the block. When this value is `0` (false), the user is not ready to read the response data from the block.

Data Types: `Boolean`

done — Done signal
`Boolean` scalar

Done signal, specified as a Boolean scalar. This value indicates the block when to read the AXI-Lite register information.

Set this value to `1` (true) after sending one set of register information to the block. The block reads the AXI-Lite register information only when you set this value to `1`. Set this value to `0` (false) immediately after this operation.

One set of register information is a combination of **ctrlInf** (Control information), **slvAddr** (Slave address), **regAddr** (Register address), and **dataReg**, **dataReg1**, **dataReg2**, and **dataReg3** (Data registers).

Data Types: `Boolean`

sdaIn — Input serial data
`Boolean` scalar

Input serial data, specified as a Boolean scalar. This port provides a serial data signal to the block from the slave device.

Dependencies

To enable this port, set the **Type of model** parameter to `Simulation`.

Data Types: `Boolean`

Output

scl — Output serial clock
Boolean scalar

Output serial clock, returned as a Boolean scalar. This port provides a serial clock signal from the block to the slave device.

Data Types: Boolean

sda — Output serial data
Boolean scalar

Output serial data, returned as a Boolean scalar. This port provides a serial data signal from the block to the slave device.

Data Types: Boolean

rdy — Ready signal
Boolean scalar

Ready signal, returned as a Boolean scalar. When this value is 1 (true), the block is ready to accept the configuration data. When this value is 0 (false), the block is not ready to accept the configuration data.

Data Types: Boolean

respData — First response data register
scalar

First response data register that contains the first 4 bytes of data from the slave-device register, returned as a scalar. This register includes the least significant bit (LSB).

Data Types: uint32

respData1 — Second response data register
scalar

Second response data register that contains the second set of 4 bytes of data from the slave-device register, returned as a scalar.

Data Types: uint32

respData2 — Third response data register
scalar

Third response data register that contains the third set of 4 bytes of data from the slave-device register, returned as a scalar.

Data Types: `uint32`

respData3 — Fourth response data register
scalar

Fourth response data register that contains the fourth set of 4 bytes of data from the slave-device register, returned as a scalar. This data includes the most significant bit (MSB).

Data Types: `uint32`

validData — Indication of valid response data
`Boolean` scalar

Control signal that indicates if the response data is valid, returned as a Boolean scalar. When this value is `1` (true), the response data from response data registers is valid. When this value is `0` (false), the response data from response data registers is not valid.

Data Types: `Boolean`

Parameters

Type of model — Model-type selection
`Generation` (default) | `Simulation`

Specify the model type.

- To use this block for simulation purposes, set this parameter to `Simulation`.
- To use this block for generation purposes, set this parameter to `Generation`. The generation process creates a hardware IP with AXI-Lite interface, SCL, and SDA interfaces.

Speed — Speed-mode selection
`Standard Mode` (default) | `Fast Mode` | `Fast Plus Mode`

Specify the speed mode as one of these values:

- `Standard Mode` — Supports frequencies up to 100 KHz

- `Fast Mode` — Supports frequencies up to 400 KHz
- `Fast Plus Mode` — Supports frequencies up to 1 MHz

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Introduced in R2019a

LED

Connect signals attached to LEDs on hardware board
Library: SoC Blockset / Hardware Logic I/O
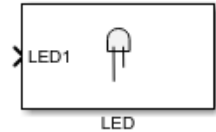
Description

The LED block indicates the status of a signal. The hardware logic signals connected to an LED block are equivalent to the signals connected to the light emitting diodes (LED) on the hardware board.

Ports

Input

LED*x* — Input signal
`Boolean` scalar

Input signal from the hardware logic. Each LED has a port, named **LED1** to **LED*x***, where *x* is **Number of LEDs**.

Data Types: `Boolean`

Parameters

`Hardware board` — View selected hardware
None (default) | Supported Xilinx or Intel boards | Custom boards

This parameter is read-only. To choose a hardware board and configure board parameters, see "Hardware Implementation Pane" on page 2-2.

View LEDs location — View LEDs
button

To view a diagram of the location of the LEDs on the selected hardware board, click the **View LEDs location** button.

This button is enabled only when you select specific Xilinx or Intel boards. For more information about these boards, refer to "Supported Third-Party Tools and Hardware".

IO logic — IO logic indicator
None (default) | Active High | Active Low

This parameter is read-only. Indicates the IO logic level on the selected hardware board.

When the **IO logic** parameter is shown as **Active Low**, the LED block accepts active low signals and represents the port names prefixed with letter *n*. For example, **nLED1**.

Number of LEDs — LED selection
1 (default) | list of integers in the range [1, n]

To specify the required number of LED ports, select a value from the **Number of LEDs** list. *n* represents the number of available LEDs on the specified hardware board. For example, if you select 4 from the list, the block shows four LED ports.

To use only the *n*th LED, set the **Number of LEDs** parameter to *n* and leave the unused LED ports unconnected.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

DIP Switch | Push Button

Introduced in R2019a

Push Button

Connect signals attached to push buttons on hardware board
Library: SoC Blockset / Hardware Logic I/O
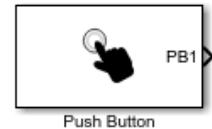
Description

The Push Button block controls the hardware mechanism. The hardware logic signals connected to a Push Button block are equivalent to the signals connected to the push buttons on the hardware board.

Ports

Input

PBIn*x* — Input signal
`Boolean` scalar

Input signal to control the hardware logic. Using these ports, you can dynamically control the hardware logic during simulation at run time. Each push button has a port, named **PBIn1** to **PBIn*x***, where *x* is **Number of push buttons**.

Dependencies

To enable this port, set the **Specify push buttons via** parameter to `InputPort`.

Data Types: `Boolean`

Output

PB*x* — Output signal
`Boolean` scalar

Output signal that returns the state of the push button. Each push button has a port, named **PB1** to **PB***x*, where *x* is **Number of push buttons**.

Data Types: `Boolean`

Parameters

`Hardware board` — View selected hardware
None (default) | Supported Xilinx or Intel boards | Custom boards

This parameter is read-only. To choose a hardware board and configure board parameters, see "Hardware Implementation Pane" on page 2-2.

`View push buttons location` — View push buttons
button

To view a diagram of the location of the push buttons on the selected hardware board, click the **View push buttons location** button.

This button is enabled only when you select specific Xilinx or Intel boards. For more information about these boards, refer to "Supported Third-Party Tools and Hardware".

`IO logic` — IO logic indicator
None (default) | Active High | Active Low

This parameter is read-only. Indicates the IO logic level on the selected hardware board.

When the **IO logic** parameter is shown as **Active Low**, the Push Button block accepts and outputs active low signals when you set the **Specify push buttons via** parameter to `InputPort` and outputs active low signals when you set the **Specify push buttons via** parameter to `Dialog`. The block represents these port names prefixed with letter *n*. For example, **nPB1**.

`Specify push buttons via` — Push-button source
`Dialog` (default) | `InputPort`

To control the hardware logic by using the block parameters, select `Dialog`. To control the hardware logic from the input port, select `InputPort`.

`Number of push buttons` — Push-button selection
1 (default) | list of integers in the range [1, n]

To specify the required number of push-button ports, select a value from the **Number of push buttons** list. *n* represents the number of available push buttons on the specified hardware board. For example, if you select 3 from the list, the block shows three push-button ports.

To use only the *n*th push button, set the **Number of push buttons** parameter to *n* and terminate the unused push button ports.

PB*n* — Selected push buttons
Off (default) | On

To enable the *n*th push-button port, select On for the PB*n* parameter. *n* represents the number of available push buttons on the specified hardware board.

Dependencies

To enable this parameter, set the **Specify push buttons via** parameter to Dialog.

Sample time — Sampling interval
-1 (default) | positive scalar

Specify the time interval a push button toggles between On and Off.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
DIP Switch | LED

Introduced in R2019a

AXI4 Master Sink

Receive random access memory data
Library: SoC Blockset / Hardware Logic Testbench
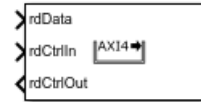
Description

The AXI4 Master Sink block receives random access memory data from advanced extensible interface AXI4-based data interface blocks. You can use this block as a test sink block for simulating AXI4-based data applications.

The block accepts data along with a control bus and outputs a control bus.

Ports

Input

`rdData` — Input data
scalar | vector

Input data from the data source. This value must be a scalar or vector.

Before reading the data, set the required data type. To set the data type, see the **Data type** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

`rdCtrlIn` — Input control bus
bus

Input control bus from the data producer, specified as a bus. This control bus comprises these control signals:

- rd_aready — Indicates the data source accepted the read request
- rd_dvalid — Indicates the data returned for the read request is valid

Data Types: `ReadControlS2MBusObj`

Output

`rdCtrlOut` — Output control bus
bus

Output control bus to the data source indicating the block is ready to accept data, returned as a scalar. This control bus comprises these control signals:

- rd_addr — Starting address for the read transaction that is sampled at the first cycle of the transaction
- rd_len — Number of data values you want to read, sampled at the first cycle of the transaction
- rd_avalid — Control signal that specifies whether the read request is valid
- rd_dready — Control signal that indicates when the block can read data

Data Types: `ReadControlM2SBusObj`

Parameters

`Data type` — Input data type
`uint8` (default) | `double` | `single` | `int8` | `int16` | `int32` | `uint16` | `uint32` | `fixdt(1,16,0)`

Select the data type format for the input AXI data.

Click the >> button to display the **Data Type Assistant**, which helps you to set the data type for the **rdData** input port. For details, see "Specify Data Types Using Data Type Assistant" (Simulink).

`Dimensions` — Input data dimensions
`10` (default) | positive integer | array

Specify the dimensions of the input data as a positive scalar or an array. This value defines the length of the transaction.

Example: 1 specifies a scalar sample.

Example: [10 1] specifies a vector of ten scalars.

Initial address — Start address
0 (default) | nonnegative scalar integer

Specify the address from which the block reads the data. This value must be a nonnegative integer.

Initial delay — Initial delay
0 (default) | nonnegative scalar

Specify the initial time after which the read operation starts.

Sample time — Time interval of sampling
1 (default) | scalar

Specify a discrete time at which the block accepts data. This value must be a scalar.

Save data in workspace — Save to workspace
off (default) | on

Select this parameter to save the input data to the MATLAB® workspace.

Variable name — Workspace variable name
simOut (default) | any MATLAB-supported variable name

Specify the workspace variable to which input data is saved. This parameter can be any MATLAB-supported variable name.

Dependencies

To enable this parameter, select the **Save data in workspace** parameter.

See Also
AXI4 Master Source

Introduced in R2019a

AXI4 Master Source

Generate random access memory data
Library: SoC Blockset / Hardware Logic Testbench
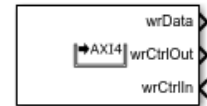
Description

The AXI4 Master Source block generates random access memory data to advanced extensible interface AXI4-based data interface blocks. You can use this block as a test source block for simulating AXI4-based data applications.

The block accepts a control bus and outputs data along with a control bus.

Ports

Input

`wrCtrlIn` — Input control bus
bus

Control bus from the data consumer signaling that data consumer is ready to accept data, specified as a scalar. This control bus comprises these control signals:

- wr_ready — Indicates the block can send data to the data consumer
- wr_complete — Indicates the write transaction has completed at the data consumer
- wr_bvalid — Indicates the data consumer has accepted the transaction

Data Types: `WriteControlS2MBusObj`

Output

wrData — Output AXI data
scalar | vector

Output AXI data to the data consumer. This value is returned as a scalar or vector.

You can change the data type of the output data. For more information, see the **Data type** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | fixed point

wrCtrlOut — Output control bus
bus

Control bus to the data consumer, returned as a bus. This control bus comprises these control signals:

- wr_addr — Specifies the starting address that the block writes
- wr_len — Specifies the number of data elements in the write transaction
- wr_valid — Indicates the data sampled at the **wrData** output port is valid

Data Types: WriteControlM2SBusObj

Parameters

Data type — Output data type
uint8 (default) | double | single | int8 | int16 | int32 | uint16 | uint32 | fixdt(1,16,0)

Select the data type format for the output AXI data.

Click the >> button to display the **Data Type Assistant**, which helps you to set the data type for the **wrData** output port. For details, see "Specify Data Types Using Data Type Assistant" (Simulink).

Dimensions — Output data dimensions
10 | positive scalar | array

Specify the dimensions of the output data as a positive scalar or an array. This value defines the length of the transaction.

Example: 1 specifies a scalar sample.

Example: [10 1] specifies a vector of ten scalars.

Initial address — Start address
0 (default) | nonnegative integer

Specify the address to which the block writes the data. This value must be a nonnegative integer.

Initial delay — Initial delay
0 (default) | nonnegative scalar

Specify the initial time after which the write operation starts. This value must be a nonnegative scalar.

Data generation — Output generation type
counter (default) | random | ones | workspace

Specify the generation type for the output as one of these values:

- counter — Generate data from a counter, based on the selected data type.
- random — Generate random data.
- ones — Generate data with all the bits as ones, based on the selected data type.
- workspace — Generate data from the MATLAB workspace.

Counter init value — Initial counter value
0 (default) | scalar

Specify the value from which the counter starts. The valid range of counter values depends on the selected value for the **Data type** parameter. If this value is out of the valid range, it is rounded off to the nearest valid value.

For example, if **Data type** is uint8 and this value is 6.787, this value is rounded to 7.

Dependencies

To enable this parameter, set the **Data generation** parameter to counter.

`Variable name` — Workspace variable name
`simOut` (default) | any MATLAB-supported variable name

Specify the workspace variable from which output data is generated. This parameter can be any MATLAB-supported variable name.

Note The workspace variable must be a numerical array.

Dependencies

To enable this parameter, set the **Data generation** parameter to `workspace`.

`Sample time` — Time interval of sampling
1 (default) | scalar

Specify the discrete time at which the block outputs data. This value must be a scalar.

See Also
AXI4 Master Sink

Introduced in R2019a

Stream Data Sink

Receive continuous stream data
Library: SoC Blockset / Hardware Logic Testbench

Description

The Stream Data Sink block receives continuous stream data from advanced extensible interface AXI4-based stream data interface blocks. You can use this block as a test sink block for simulating AXI4-based stream data applications.

The block accepts stream data along with a control bus and outputs a control bus.

Ports

Input

`rdData` — Input stream data
scalar | vector

Input stream data from the data source. This value must be a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

`rdCtrlIn` — Input control bus
bus

Input control bus from the data source. This control bus comprises the following control signals:

* valid — Indicates the input stream data on the **rdData** input port is valid
* tlast — Indicates the end of the data transaction

Data Types: `StreamM2SBusObj`

Output

`rdCtrlOut` — Output control bus
bus

Output control bus to the data source, indicating that the block is ready to accept stream data. This control bus comprises a ready signal.

Data Types: `StreamS2MBusObj`

Parameters

`Save data in workspace` — Save data in workspace
`off` (default) | `on`

Select this parameter to save the input stream data to the MATLAB workspace.

`Variable name` — Workspace variable name
`simOut` (default) | any MATLAB-supported variable name

Specify the workspace variable to which input stream data is saved. This parameter can be any MATLAB-supported variable name.

Dependencies

To enable this parameter, select the **Save data in workspace** parameter.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

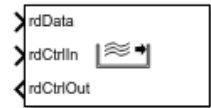Stream Data Source

Introduced in R2019a

Stream Data Source

Generate continuous stream data
Library: SoC Blockset / Hardware Logic Testbench
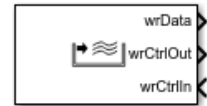
Description

The Stream Data Source block generates stream data to advanced extensible interface AXI4-based stream data interface blocks. You can use this block as a test source block for simulating AXI4-based stream data applications.

The block accepts a control bus and outputs stream data along with a control bus.

Ports

Input

wrCtrlIn — Input control bus
bus

Control bus from the data consumer signaling that data consumer is ready to accept stream data. This control bus comprises a ready signal.

Data Types: `StreamS2MBusObj`

Output

wrData — Output stream data
scalar | vector

Output stream data to the data consumer. This value is returned as a scalar or vector.

You can change the data type of the output stream data. For more information, see the **Data type** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

wrCtrlOut — Output control bus
bus

Control bus to the data consumer, returned as a bus. This control bus comprises these control signals:

- valid — Indicates the output data on the **wrData** output port is valid
- tlast — Indicates the end of the data transaction

Data Types: `StreamM2SBusObj`

Parameters

Data type — Output data type
`uint8` (default) | `double` | `single` | `int8` | `int16` | `int32` | `uint16` | `uint32` | `fixdt(1,16,0)`

Select the data type format for the output stream data.

Click the ▶ `>>` button to display the **Data Type Assistant**, which helps you to set the data type for the **wrData** output port. For details, see "Specify Data Types Using Data Type Assistant" (Simulink).

Dimensions — Output data dimensions
`10` (default) | positive integer | array

Specify the dimensions of the output stream data as a positive scalar or an array.

Example: `1` specifies a scalar sample.

Example: `[10 1]` specifies a vector of ten scalars.

Burst length — Length of single burst
`20` (default) | positive integer

Length of the single burst, specified as a positive integer.

Total bursts — Total number of bursts
4 (default) | positive integer

Total number of bursts generated from the block, specified as a positive integer.

Data generation — Output generation type
counter (default) | random | ones | workspace

Specify the generation type for the output as one of these values:

- counter — Generate data from a counter, based on the selected data type.
- random — Generate a random data.
- ones — Generate data with all the bits as ones, based on the selected data type.
- workspace — Generate data from the MATLAB workspace.

Counter init value — Initial counter value
0 (default) | scalar

Specify the value from which the counter starts. The valid range of counter values depends on the selected value for the **Data type** parameter. If this value is out of the valid range, it is rounded off to the nearest valid value.

For example, if **Data type** is uint8 and this value is 6.787, this value is rounded to 7.

Dependencies

To enable this parameter, set the **Data generation** parameter to counter.

Variable name — Workspace variable name
simOut (default) | any MATLAB supported variable name

Specify the variable name from which output stream data is generated. This parameter can be any MATLAB-supported variable name.

Note The workspace variable must be a numerical array.

Dependencies

To enable this parameter, set the **Data generation** parameter to workspace.

Sample time — Time interval for sampling
1 (default) | scalar

Specify the discrete time at which the block outputs data. This value must be a scalar.

Transfer delay (in samples) — Delay between bursts
0 (default) | nonnegative integer

Time after which the next burst occurs. This value must be a nonnegative integer.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Stream Data Sink

Introduced in R2019a

1-63

Video Test Sink

Receive continuous video stream data
Library: SoC Blockset / Hardware Logic Testbench
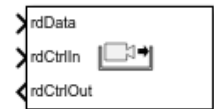
Description

The Video Test Sink block receives continuous video stream data from advanced extensible interface AXI4-based video stream data interface blocks. You can use this block as a test sink block for simulating AXI4-based video stream data applications.

The block accepts video stream data along with a control bus and outputs a control bus.

Ports

Input

rdData — Input video stream data
vector

Input video stream data from the data source. This value must be a vector.

Data Types: uint8

rdCtrlIn — Input control bus
bus

Input control bus from the data source, specified as a bus. This control bus comprises these signals:

- hStart — First pixel in a horizontal line of a frame
- hEnd — Last pixel in a horizontal line of a frame

- vStart — First pixel in the first (top) line of a frame
- vEnd — Last pixel in the last (bottom) line of a frame
- valid — Indicates the input pixel data on **rdData** input port is valid

Data Types: `pixelcontrol`

Output

`rdCtrlOut` — Output control bus
bus

Output control bus to the data source signaling that the block is ready to accept video stream data. This control bus comprises a ready signal.

Data Types: `StreamVideoS2MBusObj`

Parameters

`Frame size` — Frame dimensions
160x120p (default) | ...

Select the frame dimensions as one of these values:

- `576p SDTV (720x576p)`
- `720p HDTV (1280x720p)`
- `1080p HDTV (1920x1080p)`
- `160x120p`
- `320x240p`
- `640x480p`
- `800x600p`
- `1024x768p`
- `1280x768p`
- `1280x1024p`
- `1360x768p`
- `1366x768p`

- 1400x1050p
- 1600x1200p
- 1680x1050p
- 1920x1200p

The **Frame size** value must be same as that of the data source.

Color space — Color space
YCbCr422 (default) | RGB | YOnly

Select the type of color space as YCbCr422, RGB, or YOnly. The **Color space** value must be same as that of the data source.

Reorder input frame — Input frame reorder
off (default) | on

Select this option to reorder input pixels. Streaming pixel format order is from left to right across each line, then down to the next line, or *row-major*. However, some matrix operations use *column-major* order, that is, from top to bottom and then right to the next column. Depending on how your design has manipulated the pixels, you may need to reorder them for correct display.

Save data in workspace — Save data in workspace
off (default) | on

Select this parameter to save the input video stream data to the MATLAB workspace.

Variable name — Workspace variable name
simOut (default) | any MATLAB-supported variable name

Specify the workspace to which input video stream data is saved. This parameter can be any MATLAB-supported variable name.

Dependencies

To enable this parameter, select the **Save data in workspace** parameter.

View input — Display input in MATLAB
off (default) | on

Select this parameter to view the input video stream data in the MATLAB viewer.

Extended Capabilities

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Video Test Source

Introduced in R2019a

Video Test Source

Generate continuous video stream data
Library: SoC Blockset / Hardware Logic Testbench

Description

The Video Test Source block generates continuous video stream data to advanced extensible interface AXI4-based video stream data interface blocks. You can use this block as a test source block for simulating AXI4-based video stream data applications.

The block accepts a control bus and outputs video stream data along with a control bus.

Ports

Input

wrCtrlIn — Input control bus
bus

Control bus from the data consumer signaling that data consumer is ready to accept video stream data. This control bus comprises a ready signal.

Data Types: StreamVideoS2MBusObj

Output

wrData — Output video stream data
vector

Output video stream data to the data consumer. This value is returned as a vector.

Data Types: uint8

wrCtrlOut — Output control bus
bus

Control bus to the data consumer, returned as a bus. This control bus comprises these control signals:

- hStart — First pixel in a horizontal line of a frame
- hEnd — Last pixel in a horizontal line of a frame
- vStart — First pixel in the first (top) line of a frame
- vEnd — Last pixel in the last (bottom) line of a frame
- valid — Indicates the output pixel data on the **wrData** output port is valid

Data Types: `pixelcontrol`

Parameters

Frame size — Frame dimensions
160x120p (default) | ...

Select the frame dimensions as one of these values:

- `480p SDTV (720x480p)`
- `576p SDTV (720x576p)`
- `720p HDTV (1280x720p)`
- `1080p HDTV (1920x1080p)`
- `160x120p`
- `320x240p`
- `640x480p`
- `800x600p`
- `1024x768p`
- `1280x768p`
- `1280x1024p`
- `1360x768p`
- `1366x768p`

- 1400x1050p
- 1600x1200p
- 1680x1050p
- 1920x1200p

Video source — Select video source type
Video file (default) | Color bar | Ramp

Select the type of video source as Video file, Color bar, or Ramp.

Input file name — Select input video file
handshake_left.avi (default) | any supported video file format

Select the input video file by clicking the **Browse** button and navigating to the video file location.

Dependencies

To enable this parameter, set the **Video source** parameter to Video file.

Color space — Color space
YCbCr422 (default) | RGB | YOnly

Select the type of color space as YCbCr422, RGB, or YOnly.

Reorder output frame — Output frame reordering
off (default) | on

Select this option to reorder output pixels to *column-major* order. Streaming pixel format order is from left to right across each line, then down to the next line, or *row-major*. However, some matrix operations use *column-major* order, that is, from top to bottom and then right to the next column.

Frame sample time — Frame sample time
1/60 (default) | positive scalar

Specify the frame sample time as a positive scalar. The denominator in default value 1/60 denotes the number of frames per second.

Extended Capabilities

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Video Test Sink

Introduced in R2019a

UDP Read (HOST)

Receive UDP packets on local host computer from remote host
Library: SoC Blockset / Host I/O

Description

The UDP Read (HOST) block receives UDP (User Datagram Protocol) packets from remote host on the local host. The local host in the host computer on which you want to receive UDP packets. The remote host is the host computer or hardware from which you want to receive UDP packets.

Ports

Output

data — UDP packet received from remote host
numeric vector

UDP packet received on local host computer, returned as a numeric vector. The **Data type for Message** and **Length** parameters set this output data type and packet length, respectively.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

length — Length of UDP packet
nonnegative scalar

Length of UDP packet returned on the **data** port.

This port is unnamed until you clear the **Output variable-size signal** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Parameters

`Local IP port` — **IP port number of local host**
25000 (default) | integer from 1 to 65,535

Specify the IP port number of local host.

Note On Linux®, to set the local IP port number to a value less than 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

`sudo matlab`

`Remote IP address ('0.0.0.0' to accept all)` — **IP address of remote host**
`'0.0.0.0'` (default) | dotted-quad expression

Specify the IP address of the remote host. Set this value to a specific IP address, to block UDP packets from all other IP addresses. To accept UDP packets from all IP addresses, use the default value `'0.0.0.0'`.

`Receive buffer size (bytes)` — **Maximum number of data bytes in received data**
8192 (default) | positive integer

Specify the maximum number of data bytes of UDP packets you want to store in the local buffer. Set this value large enough to avoid data loss caused by buffer overflows.

`Maximum length for Message` — **Maximum length of data**
255 (default) | positive integer scalar

Specify the maximum length of the output UDP packet. Set this parameter to a value equal to or greater than the data size of the UDP packet. The block truncates any data that exceeds this length.

The maximum payload size of a UDP packet is 65,507 bytes. The **Maximum length for Message** is equal to the maximum payload size of a UDP packet in bytes divided by the

data type size of the UDP packet. For example, if the output data type is double, then set **Maximum length for Message** value to 65507/8 = 8118.

Data type for Message — Data type of output UDP packet
uint8 (default) | single | double | int8 | int16 | uint16 | int32 | uint32 | boolean

Select the data type for the vector elements of output UDP packet. Match this data type with the data type of the UDP packets sent by the remote host.

Blocking time (seconds) — Time to wait for UDP packet
0 (default) | nonnegative scalar

Specify the duration of time to wait for a UDP packet before returning control to the scheduler for each sample.

Sample time (seconds) — Sample time
0.01 (default) | nonnegative scalar

Specify how often the scheduler runs this block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
UDP Write (HOST)

Introduced in R2019a

UDP Write (HOST)

Send UDP packets from host computer to remote host
Library: SoC Blockset / Host I/O
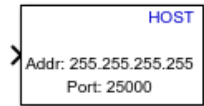
Description

The UDP Write (HOST) block sends UDP (User Datagram Protocol) packets from a local host to a remote host. The local host in the host computer from which you want to send UDP packets. The remote host is the host computer or hardware to which you want to send UDP packets. The remote host is identified by the remote IP address and remote IP port parameters from host computer.

Ports

Input

Port_1 — Input signal
numeric vector

Input signal, specified as a numeric vector. The block sends this data as a UDP packets to the specified remote IP address and port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Parameters

Remote IP address ('255.255.255.255' for broadcast) — IP address of remote host
`'255.255.255.255'` (default) | dotted-quad expression

Specify the IP address of the remote host. To broadcast UDP packets, use the default value, `'255.255.255.255'`.

`Remote IP port` — IP port number of remote host
`25000` (default) | integer from 1 to 65,535

Specify the IP port number of the remote host.

`Local IP port source` — Source of local IP port source
`Automatically determine` (default) | `Specify via dialog`

Set the source of Local IP port for the block by selecting one of these values:

- `Automatically determine` — Assigns an available local IP port number randomly from which UDP packets are sent.
- `Specify via dialog` — Allows you to specify the local IP port number using the **Local IP port** parameter.

`Local IP port` — IP port number of local host
`-1` (default) | integer from 1 to 65,535

Specify the port number of the local host. If this value is set to `-1` (default), the block sets the local IP port number to a random available port number and uses that port to send the UDP packets. If the remote host accepts UDP packets from a particular IP port number, specify that IP port number for this value.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
`UDP Read (HOST)`

Introduced in R2019a

Memory Channel

Stream data through a memory channel
Library: SoC Blockset / Memory
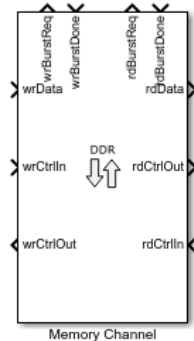
Description

The Memory Channel block streams data through external memory. Conceptually, it models data transfer between one algorithm and another, through shared memory. The algorithm can be hardware logic (HW), a processor model, or I/O devices. The writer algorithm requests access to memory from the Memory Controller block. After access is granted the writer algorithm writes to a memory buffer. In the model, the data storage is modeled as buffers in the channel. When deploying on hardware, the data is routed to an external shared memory.

This block can be configured to support any of these protocols:

- **AXI4-Stream to Software via DMA** – Model a connection between hardware logic and a software task through external memory. The writer puts data into the channel using a MathWorks® simplified AXI stream protocol and the reader gets data from a DMA driver interface. The channel models the datapath and software stack of that connection including a FIFO, DMA engine, interconnect and external memory, interrupts, kernel buffer management of the DMA driver, and data transfers to the software task. For more information about MathWorks simplified AXI stream protocol, see "AXI4-Stream Interface".

This image is a conceptual view of a Memory Channel block, streaming data from an FPGA algorithm to a processor algorithm.
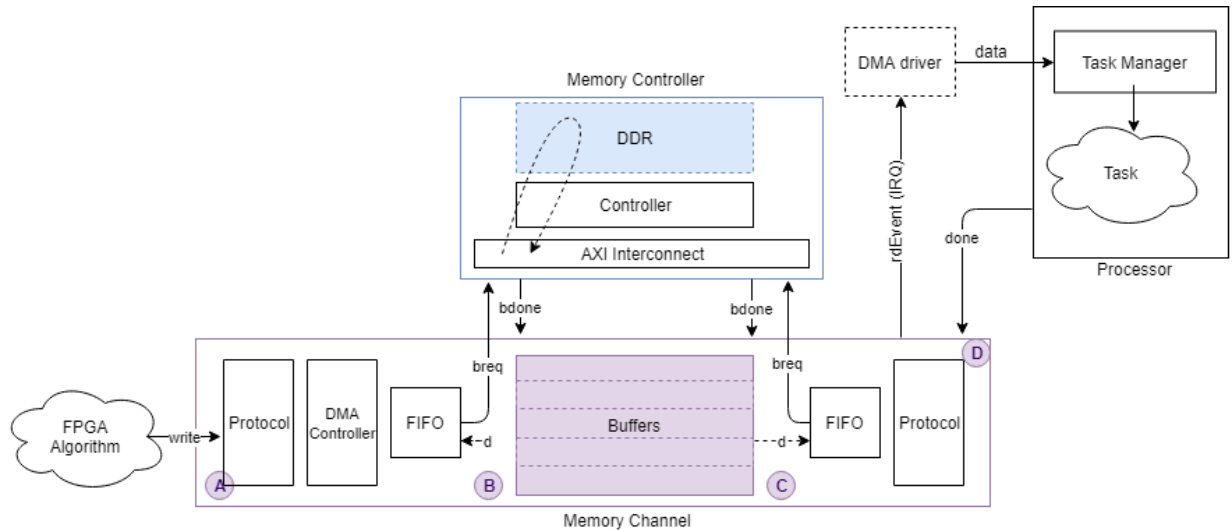
- **AXI4-Stream FIFO** – Model a connection between two FPGA algorithms through external memory. The writer puts data into the channel as a master using the MathWorks simplified AXI stream protocol and the reader receives data from the channel as a slave using the same protocol. The channel behaves as a first in first out (FIFO) memory. The channel models the datapath of the connection. The Memory Channel block includes an intermediate burst-level FIFO, DMA engine, interconnect, and external memory. The external memory itself is managed as a circular buffer, where a buffer must be written before it can be read. For more information about the MathWorks simplified AXI stream protocol, see "AXI4-Stream Interface".

This image is a conceptual view of a Memory Channel block, streaming data from one FPGA algorithm to another FPGA algorithm.
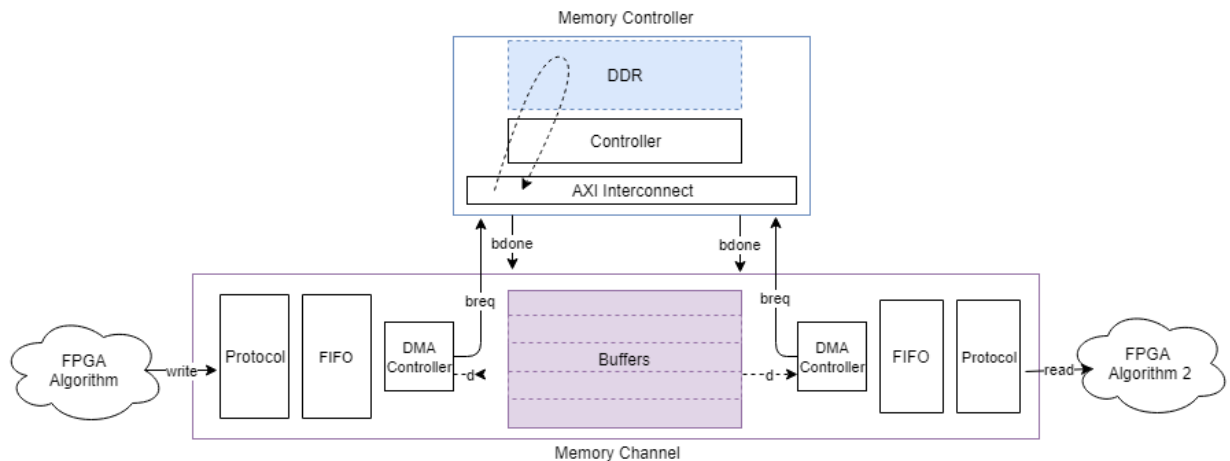
Memory Channel

- **AXI4-Stream Video FIFO** – Model a connection between two hardware algorithms through external memory. This channel structure is similar to the **AXI4 Stream FIFO** configuration, but the writer and reader are using the MathWorks streaming pixel protocol, along with a back-pressure signal. For more information, see "AXI4-Stream Video Interface".

- **AXI4-Stream Video Frame Buffer** – Model a connection between two hardware algorithms through external memory, using full video frame buffers. The protocol is the MathWorks streaming pixel protocol with back pressure. Also, the reader can ensure that the frame buffer is synchronized with downstream video timings by asserting an FSYNC protocol signal. The datapath includes a Video-DMA (VDMA) engine and the external memory buffers are managed as a circular buffer of full video frames. The channel structure is identical to the structure of **AXI4 Stream FIFO** channel type.

- **AXI4-Random Access** – Model a connection between two hardware algorithms through external memory, using the MathWorks simplified AXI4-Master protocol. Both the writer and the reader are masters, the channel is a slave in both cases. The external memory is unmanaged (there are no logical buffers, and no circular buffer). It is up to the reader and writer to coordinate timing on accesses to ensure the integrity of the data. For more information, see "Simplified AXI4 Master Interface"

  This image is a conceptual view of a Memory Channel block, with random-access to the memory for writing, and random-access to the memory for reading.
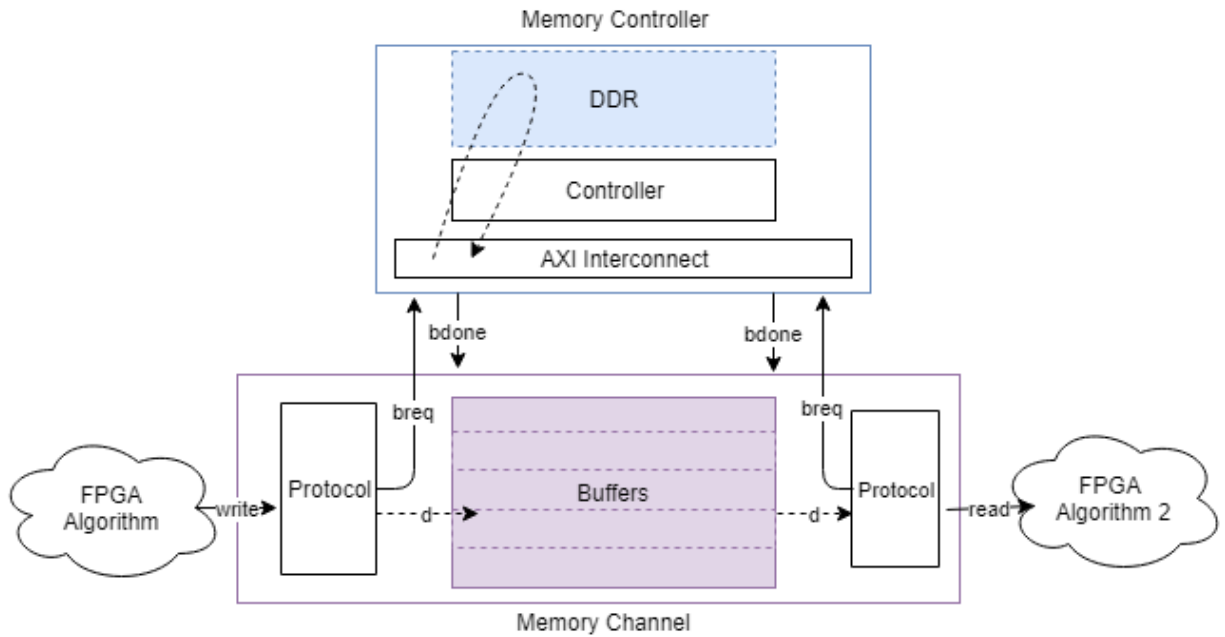
For more information on the available protocols, see "External Memory Channel Protocols".

Ports

Input

wrData — Writer data bus signal
scalar | vector | matrix

Drive data from a data producer to a memory subsystem.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

wrCtrlIn — Writer input control signal
bus

This port represents the protocol from the data producer to the memory channel. The Memory Channel block checks this signal when using **wrData**. The signals on the bus depend on the `Channel type` parameter. Use the SoC Bus Creator block to create this control bus. For more information about bus types, see "External Memory Channel Protocols".

Channel Type Configuration	Bus Type
AXI4-Stream to Software via DMA	`StreamM2SBusObj`
AXI4 Stream FIFO	`StreamM2SBusObj`
AXI4 Stream Video FIFO	`pixelcontrol`
AXI4 Stream Video Frame Buffer	`pixelcontrol`
AXI4 Random Access	`WriteControlM2SBusObj`

Data Types: `StreamM2SBusObj` | `pixelcontrol` | `WritecontrolM2SBusObj`

`rdCtrlIn` — Reader input control signal
bus | scalar

A bus from a data consumer block, signaling that it is ready to accept read data. For the streaming protocols, the **rdCtrlIn** is a backpressure signal from a data consumer to the memory channel block. For the **AXI4 Random Access** protocol, this is a read-request from the reader. The signals on the bus depend on the `Channel type` parameter. Use the SoC Bus Creator block to create this control bus.

Channel Type Configuration	Bus Type
AXI4-Stream to Software via DMA	Message carrying a `Boolean`
AXI4 Stream FIFO	`StreamS2MBusObj`
AXI4 Stream Video FIFO	`StreamVideoS2MBusObj`
AXI4 Stream Video Frame Buffer	`StreamVideoFSyncS2MBusObj`
AXI4 Random Access	`ReadControlM2SBusObj`

Note If the data consumer is a processor, this bus is carried on an message from the processor and the port name is **rdDone**. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `Boolean` | `StreamS2MBusObj` | `StreamVideoS2MBusObj` | `StreamVideoFSyncS2MBusObj` | `ReadControlM2SBusObj`

`wrBurstDone` — Writer control input from memory controller
scalar

This message port receives control messages from a connected Memory Controller block that the requested burst transaction completed. Connect the **burstDone** output signal from the Memory Controller block to this port. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `BurstRequest2BusObj`

`rdBurstDone` — Reader control input from memory controller
scalar

This message port receives control messages from a connected Memory Controller block that the requested burst transaction completed. Connect the **burstDone** output signal from the Memory Controller block to this port. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `BurstRequest2BusObj`

Output

`rdData` — Output data bus to data consumer
scalar

This bus contains the data read from the memory channel.

Note When connected to a processor subsystem, this port sends the output data, as a message, to the connected Stream Read block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `SoCData`

`rdCtrlOut` — Reader control signal from memory channel to data consumer
bus | sclara

Control signal from channel to data consumer. Its contents depend on the `Channel type` parameter. Connect this signal to the data consumer. Use the SoC Bus Selector block to separate the signal from the bus.

Channel Type Configuration	Bus Type
AXI4-Stream to Software via DMA	`rteEvent`
AXI4 Stream FIFO	`StreamM2SBusObj`
AXI4 Stream Video FIFO	`pixelcontrol`
AXI4 Stream Video Frame Buffer	`pixelcontrol`
AXI4 Random Access	`ReadControlS2MBusObj`

Note When connected to a processor subsystem, this port sends the output data, as a message, to the connected Task Manager block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `StreamM2SBusObj` | `ReadControlS2MBusObj` | `pixelcontrol` | `rteEvent`

`wrCtrlOut` — Writer control signal from memory channel to data producer
bus

This bus represents the protocol bus from the memory channel to the data producer. The signals on the bus depend on the `Channel type` parameter. Use the SoC Bus Selector block to separate the signal from the bus.

Channel Type Configuration	Bus Type
AXI4-Stream to Software via DMA	`StreamS2MBusObj`
AXI4 Stream FIFO	`StreamS2MBusObj`
AXI4 Stream Video FIFO	`StreamVideoS2MBusObj`
AXI4 Stream Video Frame Buffer	`StreamVideoS2MBusObj`
AXI4 Random Access	`WriteControlS2MBusObj`

Data Types: `StreamS2MBusObj` | `WriteControlS2MBusObj` | `StreamVideoS2MBusObj`

`wrBurstReq` — Write burst request
scalar

This message port sends control signal requesting burst access from the memory controller. Connect it to the **burstReq** input of the Memory Controller block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `BurstRequestBusObj`

`rdBurstReq` — Read burst request
scalar

This message port sends control signal requesting burst access from the memory controller. Connect it to the **burstReq** input of the Memory Controller block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `BurstRequestBusObj`

Parameters

`Hardware board` — View or modify current hardware settings
name of selected hardware board

This property is read-only.

This parameter shows a link to the currently selected hardware board. Click the link to open the configuration parameters, and adjust the settings, or choose a different board.

To learn more about configuration parameters, see "FPGA design (mem channels)" on page 2-6.

`Show implementation info` — View channel information
text window

This property is read-only.

This parameter shows a link to the implementation information specific to the model. Click the link to view the information (opens in new window).

Main

`Channel type` — Choose channel protocol
AXI4-Stream FIFO (default) | AXI4-Stream to Software via DMA | AXI4-Stream Video FIFO | AXI4-Stream Video Frame Buffer | AXI4 Random Access

Specify the protocol for the channel. Choose one of the following values:

- AXI4-Stream to Software via DMA
- AXI4 Stream FIFO
- AXI4 Stream Video FIFO
- AXI4 Stream Video Frame Buffer
- AXI4 Random Access

For additional information about memory channel protocols, see "External Memory Channel Protocols".

Region size (bytes) — Size of memory allocated for region, in bytes
calculated

This property is read-only.

The size in bytes of the region. This value is calculated as the number of buffers multiplied by buffer size.

Example: If Buffer size is 1024, and the number of buffers is set to 8, then Region size is 8192.

Buffer size (bytes) — Size of buffer, in bytes
1024 (default) | scalar

Specify the size in bytes of each buffer in the region.

The following rules apply when setting burst and buffer sizes.

1. The Burst Length of a given channel interface, calculated in bytes, must be less than 4096 bytes. To calculate the burst size in bytes, the channel interface scalar datatype is converted to bytes and then multiplied by the Burst Length.
2. The Burst Length can be set above 256, but will warn if generating to an AXI-based target platform. AXI-based memory systems have a maximum burst length of 256.
3. The Channel Length must be an integer multiple of burst length or the burst length must be an integer multiple of channel length. That is, it must be possible to either chunk the incoming channel data to a whole number of bursts or to gather a whole number of incoming channel data to a single burst.
4. The Buffer Size must be a whole number of bursts. This must be true for both the writer's burst size (after conversion of its Burst Length to bytes) and the reader's burst size (after conversion of its Burst Length to bytes).

5 The calculated number of bursts in a buffer must not exceed 5000. This is a temporary restriction based on the event processing internal to the memory model. This can happen with shared memory regions that have large buffer sizes (such as for 1080p video frames) and channel interfaces that specify smaller burst sizes. Generally, with larger frames, bursts sizes near the 4096 byte limit must be used.

6 The scalar datatype of the channel interface as converted to a flattened channel data width (i.e. *tdata* in the implementation) cannot exceed 32 bits. This is a temporary restriction based on current limitations of the byte conversions in the memory channel model.

The following table provides examples of good and bad parameter sets.

Burst and Buffer parameter examples

Channel Datatype	Channel Dimensions	Burst Length	Burst Size	Good / Bad	Why?
uint8	[1 1]	1024	2048	Good	This is a simple 8-bit data transaction.
uint8	[1 3]	1024	4096	Good	This might represent an RGB pixel from a Vision HDL Toolbox block. It is converted to 24-bit packed data and padded with 8 bits to become a 32-bit (4-Byte) *tdata* bus to the memory. The Burst size is 1024*4B = 4096B.
fixdt(0,10,0)	[1 3]	1024	4096	Good	This is converted to a 30-bit packed pixel with 2 bits of padding.
fixdt(0,12,0)	[1 3]	1024	8192	Bad	This results in a 36-bit packed pixel which extends to 64-bit *tdata*. This data width violates the current limit of 32-bit *tdata*.
uint8	[120 160 3]	1024	4096	Bad	The scalar data is 24-bit, padded to a 32-bit *tdata*. The Channel Length is 120*160=19200. The burst length of 1024 does not evenly divide 19200.
uint8	[120 160 3]	120	480	Good	The scalar data is 24-bit, padded to a 32-bit *tdata*. The Channel Length is 120*160, and since the burst length is 120, Channel length is 160 bursts in size. The buffer size is exactly 1 frame (120*160*4) as calculated in bytes.

Number of buffers — Number of buffers in region
8 (default) | integer

Divide the region into buffers. A disparate rate between a reader and a writer slows down the faster device. For example, a slow reader causes the writer to run out of buffers and block the writer, effectively slowing down to the reader rate. Likewise, a slow writer causes the reader to run out of buffers and block the reader, effectively slowing it down to the writer rate.

- Specifying 1 – With a single buffer, access is controlled to ensure that a buffer is written, then it is read, then the next buffer is written, and so on.
- Specifying 2: With two buffers, memory access switches in a back-and-forth pattern. The writer writes the first buffer, then, while the reader is reading it, the writer can write the second buffer.
- Specifying *N* – With *N* buffers, the memory access has a ring-buffer pattern. The writer can continually write as long as buffers are available. When a buffer is completed, it becomes available for the reader. The writer and reader traverse the N buffers in a circular pattern. As long as the writer and reader maintain similar rates, the buffering prevents blockage.

Advanced

`Burst length` — Burst length for memory transactions
1 (default) | scalar

The length of bursts for this connection on the memory bus in units of scalar data. The scalar unit is the packed data type. Specify the burst size for both **Writer** and **Reader** access to the channel.

The channel data is always transferred to the memory model using burst transactions, regardless of the channel-type. For the AXI4 configuration, the algorithm-logic is responsible for defining the burst through the protocol signals.

For the streaming data configurations, the **Burst Length** parameter determines the burst size to the memory, and the channel **data** signal defines the size of each transfer on the interface.

When setting burst length, you must consider the "Buffer size (bytes)" on page 1-0 parameter.

`Use hardware board settings` — Use the Hardware Implementation settings from the configuration parameters
on (default) | off

To use the same model-wide setting as in configuration parameters, select this box. Clear the box to customize the setting for this channel. When using channel-specific settings, values are still checked against hardware-specific constraints. For setting these values in the configuration parameters, see "FPGA design (mem channels)" on page 2-18.

`Reader/Writer use same values` — Reader and writer use the same values
on (default) | off

Select this box to use the same interconnect setting for the reader and the writer of this channel. Clear the box to customize different settings for the reader and the writer. Clearing the **Reader/Writer use same values** allows you to enter a value for the writer side and a value for the reader side, for the following parameters:

- **FIFO depth (number of bursts)**
- **Almost-full depth**
- **Clock Frequency (MHz)**
- **Data width (bits)**

FIFO depth (number of bursts) — Depth of FIFO for data

12 (default) | scalar

Specify depth of data FIFO, in units of bursts. When the writer has no buffers to write to, the FIFO can absorb data until a buffer becomes available. This value is the maximum number of bursts that can be buffered before data gets dropped.

Dependencies

- To enable this parameter, clear the **Use hardware board settings** check box.
- When **Reader/Writer use same values** is cleared, there are two text boxes: one for **Writer** and one for **Reader**.

Almost full depth — Depth of FIFO when backpressure is asserted

8 (default) | scalar

Specify a number that asserts a backpressure signal from the channel to the data source. To avoid dropping data, set a high watermark, allowing the data producer enough time to react to backpressure. This number must be smaller than the FIFO depth.

Dependencies

- To enable this parameter, clear the **Use hardware board settings** check box.
- When **Reader/Writer use same values** is cleared, there are two text boxes: one for **Writer** and one for **Reader**.

Clock frequency (MHz) — Interconnect frequency of master datapath

100 (default)

Frequency of the master datapath to the interconnect controller in MHz.

Dependencies

- To enable this parameter, clear the **Use hardware board settings** check box.
- When **Reader/Writer use same values** is cleared, there are two text boxes: one for **Writer** and one for **Reader**.

`Data width (bits)` — Data width of master datapath
64 (default) | scalar

Data width of master datapath to interconnect controller in bits.

Dependencies

- To enable this parameter, clear the **Use hardware board settings** check box.
- When **Reader/Writer use same values** is cleared, there are two text boxes: one for **Writer** and one for **Reader**.

Signal Attributes

Write data signal

`Dimensions` — Dimensions of input data signal
scalar | array

wrData can be a multidimensional array. Specify the dimension for the array as a whole number.

Example: 1 – a scalar sample.

Example: [10 1] – a vector of ten scalars.

Example: [1080 1920 3] – a 1080p frame. The frame includes 1080 lines of 1920 pixels per line, and each pixel is represented by three values (for red, green and blue).

`Data type` — Data type of writer data
double | single | uint32 | int8 | int16 | int32 | uint8 | uint16 | uint32 | boolean | fixed point

Specify the data type of the **wrData** port. For help, click the **...** button. This expands the menu and shows a **Data Type Assistant**.

`Sample time` — Time interval of sampling
1 (default) | positive scalar

Specify a discrete time at which the block accepts input data, in seconds.

Read data signal

`Output data signal matches input` **— Reader and writer use the same values**
on (default) | off

Select this box to use the same dimensions and data type for the reader and the writer of this channel. Clear the box to customize different settings for the reader and the writer. Clear the box to customize different dimensions and data type for the reader and writer interfaces.

`Dimensions` **— Dimensions of output data signal**
scalar | array

rdData can be a multidimensional array. Specify the dimension for the array as a whole number.

Example: 1 – a scalar sample.

Example: [10 1] – a vector of ten scalars.

Example: [1080 1920 3] – a 1080p frame. The frame includes 1080 lines of 1920 pixels per line, and each pixel is represented by three values (for red, green and blue).

Dependencies

To enable this parameter, clear the **Output data signal matches input** check box.

`Data type` **— Data type of reader data**
double | single | uint32 | int8 | int16 | int32 | uint8 | uint16 | uint32 | boolean | fixed point

Specify the data type of the **rdData** port. For help, click the **...** button. This expands the menu and shows a **Data Type Assistant**.

Dependencies

To enable this parameter, clear the **Output data signal matches input** check box.

`Use pixel clock sample times` **— Use the pixel clock sample time**
on (default) | off

Select this box to use the pixel clock sample time. To use the pixel clock sample time, you must use scalar pixel dimensions. It is only relevant when streaming pixels. If both the reader and the writer are streaming frames, you get an error when checking this box.

Note If both reader and writer are using framed signals, the signal dimensions are not scalar and pixel timing cannot be inferred. Selecting **Use pixel clock sample times** in this case creates an error.

Dependencies

To enable this parameter, set **Channel type** to AXI4-Stream Video FIFO or AXI4-Stream Video Frame Buffer.

Frame size — Frame dimensions
480p SDTV (720x480p) (default) | ...

For video-streaming applications, **Frame size** can often be inferred, and this parameter shows as a read-only value. When it cannot be inferred, select the **Frame size** from a drop-down menu.

- When the reader or the writer are using framed signals of a frame with known porch and blanking timings, the **Frame size** is inferred from those timings. When the reader or the writer is a scalar and the other is a non-standard frame size, the **Frame size** cannot be inferred and you get an error.
- When **Channel type** is set to AXI4-Stream Video Frame Buffer and both reader and writer are using scalar dimensions for pixel streams, **Frame size** is inferred from **BufferSize** and TDATA and it is then a read-only value.
- When **Channel type** is set to AXI4-Stream Video FIFO and both reader and writer are using scalar dimensions for pixel streams, select the **Frame size** as one of these values:

  - 160x120p
  - 480p SDTV (720x480p)
  - 576p SDTV (720x576p)
  - 720p HDTV (1280x720p)
  - 1080p HDTV (1920x1080p)
  - 320x240p
  - 640x480p
  - 800x600p
  - 1024x768p

- `1280x768p`
- `1280x1024p`
- `1360x768p`
- `1400x1050p`
- `1600x1200p`
- `1680x1050p`
- `1920x1200p`
- `16x12p (test mode)`

Dependencies

To enable this parameter, set **Channel type** to `AXI4-Stream Video FIFO` or `AXI4-Stream Video Frame Buffer`, and select **Use pixel clock sample times**.

Performance

`Launch performance plots` — Display performance metrics
button

Clicking the button opens Performance plots for the memory channel in a new window. For more information about performance graphs, see "Simulation Diagnostics".

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Memory Controller | Memory Traffic Generator

Topics

"External Memory Channel Protocols"

Introduced in R2019a

Memory Controller

Arbitrate memory transactions for one or more Memory Channel blocks
Library: SoC Blockset / Memory
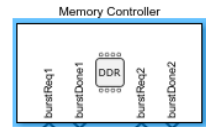
Description

The Memory Controller block arbitrates between masters and grants them unique access to shared memory. Configure this block to support multiple channels with various arbitration protocols. The Memory Controller block is also instrumented to log and display performance data, enabling you to debug and understand the performance of your system at simulation time.

The following image shows the implementation of the Memory Controller block.
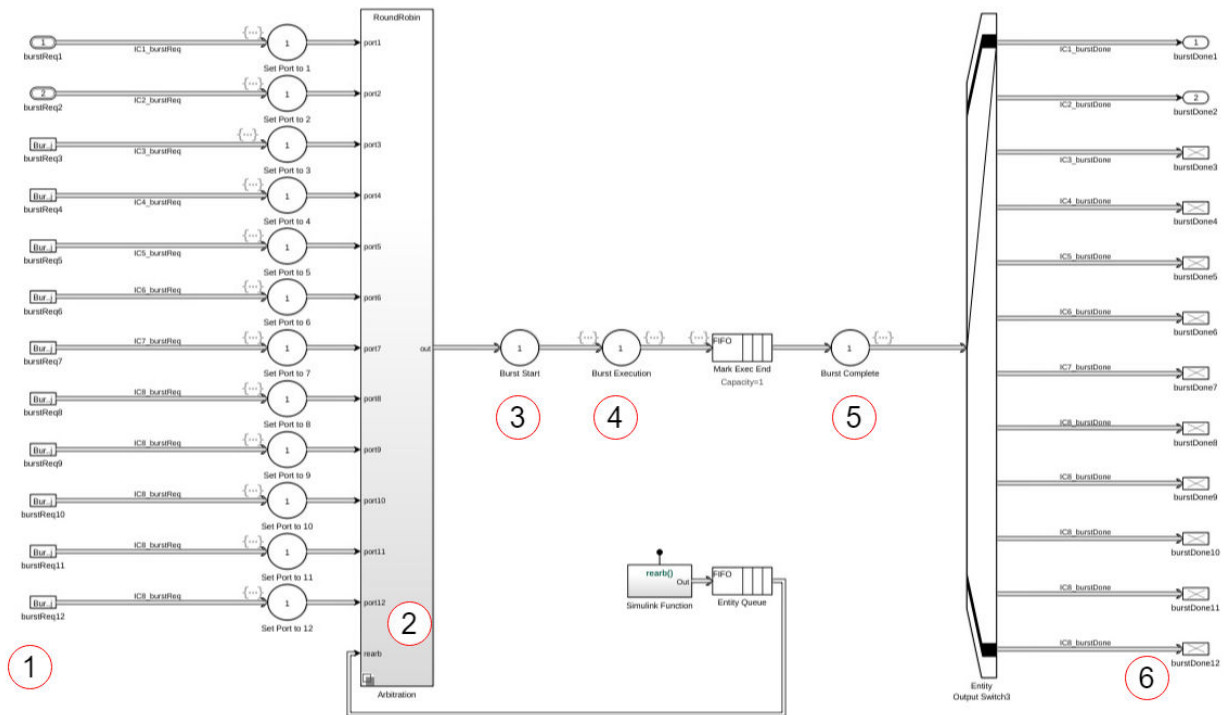
The numbers in the image represent different latency stages of the block.

1 A burst-request enters the block.

2 The request may be delayed by arbitration until it is granted access to the bus. Set the arbitration policy in "Interconnect arbitration" on page 1-0 .

3 If your model requires an additional delay before the first transfer starts, set that value in "Request to first transfer (in clocks)" on page 1-0 .

4 The burst execution latency is calculated by the burst size, the data-width, the clock frequency, and the "Bandwidth derating (%)" on page 2-15 value.

5 If your model requires a delay from burst completion until a burst response is issued to the channel, set that value in "Last transfer to transaction complete (in clocks)" on page 1-0 .

The memory controller has an internal state, which is visible when using a **Logic Analyzer** to view simulation and execution metrics. The state values are:

- `BurstRequest`: A burst request enters the block.
- `BurstExecuting`: A burst is executing.
- `BurstDone`: A burst is done executing.
- `BurstComplete`: The burst is complete and the **burstDone** signal is sent to the master.

For information about visualizing memory controller latencies, see "Memory Controller Latency Plots".

Ports

Input

burstReq*N* — Request for memory access
scalar

This port receives requests for memory access as messages. Connect this input port to one of the burst request message ports (`wrBurstReq` or `rdBurstReq`) from a Memory Channel or Memory Traffic Generator block. For more information on messages, see "Message-Based Communication" (Simulink).

The number of **burstReq*N*** input ports is defined by the **Number of masters** parameter. **burstReq*N*** represents the *N*th input port.

Data Types: `BurstRequest2BusObj`

Output

burstDone*N* — Signal toward master
scalar

After a master is granted access to the memory and the burst transaction has completed, this port sends a message that the transaction completed. Memory access is then given to the next master according to the arbitration scheme. For more information on messages, see "Message-Based Communication" (Simulink).

The number of **burstDone*N*** output ports is defined by the **Number of masters** parameter. **burstDone*N*** represents the *N*th input port

Data Types: `BurstRequest2BusObj`

Parameters

`Hardware board` — View or modify current hardware settings
name of current hardware board

This property is read-only.

This parameter shows a link to the selected hardware board. Click the link to open the configuration parameters, and adjust the settings or choose a different board.

To learn more about configuration parameters for the memory controller, see "FPGA design (mem controllers)" on page 2-5.

Main

`Number of masters` — Number of masters connected to this controller
2 (default) | positive integer

Set this parameter to generate the interface accordingly, and specify how many masters connect to the memory.

Advanced

`Interconnect arbitration` — Arbitration policy
`Round robin` (default) | `Fixed port priority`

Set the arbitration policy for the memory-interconnect block. When multiple masters request for memory access, the policy is determined by the value of this parameter.

- `Round robin` sets a fair arbitration based on last service time.
- `Fixed port priority` sets a fixed priority of **burstReq1**, **burstReq2**, **burstReq3**, and so on, where **burstReq1** gets the highest priority.

`Use hardware board settings` — Use hardware implementation settings from the configuration parameters
`off` (default) | `on`

Select this parameter to use the same model-wide settings as set in the configuration parameters. Clear this parameter to customize the settings for this memory controller.

When using customized settings, values are still checked against hardware-specific constraints. For more information, see "FPGA design (mem controllers)" on page 2-15.

Bandwidth — Bandwidth for transactions towards external memory
scalar

This property is read-only.

This value shows the calculated bandwidth between the memory controller and the external memory. It is calculated as **Frequency (MHz)** multiplied by **Data width (bits)**.

Frequency (MHz) — Controller clock frequency, in MHz
200 (default) | scalar

The clock rate of the bus used to drive interactions with the external memory. The controller frequency determines the overall system bandwidth for external memory that must be shared among all the masters in the model.

Dependencies

To enable this parameter, clear the **Use hardware board settings** parameter.

Data width (bits) — Bit width of datapath
64 (default) | positive integer

Set the width, in bits, of the datapath between the memory controller and the memory interconnect.

Dependencies

To enable this parameter, clear the **Use hardware board settings** parameter.

Bandwidth derating (%) — Memory transaction inefficiencies
0-100

Model memory transaction inefficiencies specified by a derating percentage value. For every 100 clocks, memory transaction execution is paused for the number of clocks equal to **Bandwidth derating**. To set this parameter, measure the maximum bandwidth on your board and reflect the bandwidth derating from your board in this parameter. See an example in "Analyze Memory Bandwidth Using Traffic Generators".

Dependencies

To enable this parameter, clear the **Use hardware board settings** parameter.

`Request to first transfer (in clocks)` — Number of clock cycles between request and start of transfer
nonnegative integer

Specify the delay, in clock cycles, between a read or write request and the start of a transfer. Specify nonnegative integer values in both **Write** and **Read** boxes.

This delay is the number of clock cycles between making a request to the memory controller and until it returns a response. It is reflected in the **Logic Analyzer** waveforms as the time that the memory controller state remains as `BurstAccepted`. For more information about viewing waveforms in simulation, see "Buffer and Burst Waveforms".

To set this value, measure the clock cycles between the burst-request and start of transfer on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Dependencies

To enable this parameter, clear the **Use hardware board settings** parameter.

`Last transfer to transaction complete (in clocks)` — Number of clock cycles between the end of transfer and completion of transaction
nonnegative integer

Specify the delay in clock cycles between the end of a memory transfer and the end of a transaction. Specify nonnegative integer values in both **Write** and **Read** boxes.

To set this value, measure the clock cycles between the end of the burst and the completion of the transaction on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Dependencies

To enable this parameter, clear the **Use hardware board settings** parameter.

Performance

Click **Launch performance app** to open the Performance Metrics window. For additional information, see "Simulation Performance Plots".

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Memory Channel | Memory Traffic Generator | Register Channel

Topics
"External Memory Channel Protocols"

Introduced in R2019a

Memory Traffic Generator

Generate traffic towards memory controller
Library: SoC Blockset / Memory

Memory Traffic Generator
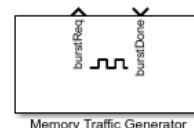
Description

When connected to a memory controller, the Memory Traffic Generator block generates read or write requests to the memory, acting as a master. Use this block to model the impact that a master's memory accesses has on your algorithm without explicitly simulating the behavior of that master. You can also use the Memory Traffic Generator block to characterize performance of your memory subsystem under varying levels of memory access contention.

Note To model memory contention, the block gains memory access, competes in arbitration, and releases access. The block does not actively read or write from memory.

Ports

Input

burstDone — End of burst and access to memory
scalar

This message port receives control messages from a connected Memory Controller block that the requested burst transaction completed. Connect the **burstDone** output signal from the Memory Controller block to this port. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `BurstRequest2BusObj`

Output

burstReq — Request memory access from memory controller
scalar

This message port sends a message requesting burst access from the memory controller. Connect this port to the **burstReq** input port of the Memory Controller block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: BurstRequest2BusObj

Parameters

Request type — Choose between write or read request
Writer (default) | Reader

Choose between a write or read request type for the block to generate. Specify Writer or Reader, respectively.

Total burst requests — Number of burst requests to generate
100 (default) | integer greater than 1

Generate recurring traffic patterns by setting this value to an integer greater than one.

Burst size (bytes) — Size of generated burst transactions
256 (default) | scalar

Specify the size of each burst transaction in bytes. This parameter, along with the width of the datapath (as configured in the configuration parameters), controls the burst length.

For example, if burst size is 256 bytes, and **Datapath width (bits)** is 64 (8 bytes), then **Burst length (beats)** is calculated as 256/8 = 32.

Time between bursts (s) — Simulation time between burst requests
1e-6 (default) | time, in seconds

Specify simulation time between burst requests, in seconds.

Dependencies

To enable this parameter, clear the **Allow simulation only parameters** parameter.

> **Tip** If you cleared **Allow simulation only parameters** and this parameter is not visible – click **Apply** at the bottom of the Block Parameters dialog box.

Allow simulation only parameters — Configure additional parameters for simulation only
on (default) | off

Select this parameter to enable configuration of simulation-only parameters.

First burst time — Simulation time for initial burst request
10e-6 (default) | time, in seconds

Specify simulation time, in seconds, for sending the initial burst request. This value must be a positive real scalar.

Dependencies

To enable this parameter, select **Allow simulation only parameters** parameter.

Random time between bursts (s) — Range of simulation time for recurring requests
[1e-6 1e-6] (default) | vector of the form [*min max*]

Specify the range of simulation time between burst requests with a vector of the form [*min max*].

- *min* is the minimum time, in seconds, between recurring requests.
- *max* is the maximum time, in seconds, between recurring requests.

min and *max* must be nonnegative, and *max* must be greater than *min*.

To specify a deterministic rate, set the minimum and maximum time between requests to the same value. If you want reproducible randomization, specify a seed in the configuration parameters, on the **Hardware Implementation** pane. For more information on setting the seed value, see "Task and memory simulation" on page 2-4.

Dependencies

To enable this parameter, select the **Allow simulation only parameters** parameter.

Wait for burst done — Wait for burst-done signal before generating next request
off (default) | on

Select this parameter to wait for a burst-done signal from the previous burst before generating the next burst request. Clear this parameter to generate burst requests regardless of other master traffic. To get a known data rate, clear this parameter.

Enable assertion — Enable verbose information
off (default) | on

Select this parameter to view diagnostic messages when the Traffic Generator block drops a packet. Clearing this parameter enhances simulation performance.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Memory Controller | Memory Channel | Register Channel

Introduced in R2019a

Register Channel

Timing model for transfer of register values
Library: SoC Blockset / Memory

Description

The Register Channel block provides a timing model for the transfer of register values between a processor and hardware logic. The register channel represents the datapath between a processor and a hardware IP via a common configuration bus. Configure the block to include one or more registers, and configure the direction for each register as write if the processor writes to it, or read if the processor reads from it.

Ports

Input

regN — Register input
scalar

Each register is assigned a port pair: an input and an output. You can configure the processor to be a writer or a reader. If the register is a read register, then the input comes from the hardware (HW) side. If the register is a write register, the input comes from the software (SW) side. By default, the Nth register port is named reg*N*. You can change a register name by clicking **Edit** in the **Registers** parameter dialog box.

Dependencies

The number of input ports depends on the number of registers in the register table.

Output

regN — **Register output**
scalar

Each register is assigned a port pair: an input and an output. You can configure the processor to be a writer or a reader. If the register is configured as a read register, then the output goes to the software (SW) side. If the register is a write register, the output goes to the hardware (HW) side. By default, the Nth register port is named reg*N*. You can change a register name by clicking **Edit** in the **Registers** parameter dialog box.

Dependencies

The number of output ports depends on the number of registers in the register table.

Parameters

`Registers` — **Edit register name, direction, data type, and dimension**
table

This parameter includes a table, where each of its lines corresponds to a register in your IP. Edit the table to add or edit a register configuration, up to ten registers.

For each register, you can edit these values:

- **Register** – Specify the register name. This changes the input and output ports for this register.
- **Direction** – Choose `write` if the processor writes the register. Choose `read` if the processor reads the register.
- **Data Type** – Select the data type for the register. Supported data types are

  - `single`
  - `int8`
  - `uint8`
  - `int16`
  - `uint16`
  - `int32`

1-107

- uint32
- boolean
- fixdt(1,16,0)
- fixdt(1,16,2^0,0)
- fixed point

- **Dimension** – Select the vector size of the register. The default value is 1.

Register write sample time — Sample time for register access
-1 (default) | two element vector

This sample time represents the clock period on the hardware side. Specify an offset time by entering a two-element vector for discrete blocks or configurable subsystems. The first element is the sample time, and the second element is the offset time. For example, an entry of [1.0 0.1] specifies a 1.0-second sample time with a 0.1-second offset. If no offset is specified, the default offset is zero.

When the value is -1, the block inherits its sample time value from the model.

Note When the **Direction** of a register is set to Write, it implies that software is the writer and hardware is the reader, but **Register write sample time** determines the sample time of the signal on the hardware side.

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

To automatically generate HDL code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design".

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Memory Controller | Memory Channel | Memory Traffic Generator

Topics

"Memory and Register Data Transfers"

Introduced in R2019a

Register Read

Read data from a register region on the specified IP core
Library: SoC Blockset / Processor I/O

Description

The Register Read block reads data from a register region on the specified IP core. In simulation, a timer-driven or event-driven task subsystem contains the Register Read block. The data signals from the Register Read block connect to the Register Channel block managing those registers and their transactions.

When developing or analyzing the software side of an SoC application, the Register Read block can be connected to an IO Data Source block. In this configuration, the IO Data Source block provides either previously recorded or artificial data, enabling a more directed simulation of the software and processor side of the application, without need to explicitly model the hardware and memory interactions.

Ports

Input

`msg` — Data message from register
scalar

This message port receives data messages from a connected Register Channel or IO Data Source block. The messages process when the Task Manager block triggers task containing the Register Read block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `SoCData`

Output

`data` — **Output signal**
vector

This port emits the data vector read from the specified registers in the Register Channel starting at **Offset address** from the base address of the IP core.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Parameters

`Device name` — **Path and file name of IP core device**
`/dev/mwipcore` (default) | character array

Enter the path and file name of the IP core device.

`Offset address` — **Offset from the base address of the IP core to the register**
`hex2dec('0100')` (default) | positive integer

Enter the offset from the base address of the IP core to the register. The block reads data from this register. Use the `hex2dec` function when you specify the offset address using a hexadecimal number expressed as a character vector. The offset address can be selected using the Memory Mapper tool.

`Output data type` — **Data type used by IP core**
uint32 (default) | single | int8 | uint8 | int16 | int32 | uint32 | boolean | fixed-point

Enter the data type used by the IP core.

`Output vector size` — **Size of data vector from IP core**
1 (default) | positive integer

Enter the size of the data vector read from the IP core device.

`Sample time` — **Sample time**
`0.1` (default) | positive number

Enter the sample time in seconds. Either the connected Register Channel or IO Data Source blocks get polled at this rate when this block is used in a timer-driven task.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder® license to generate and execute C code for your SoC device.

SoC Builder implements the Register Read block with FPGA and processor IPs that use the AXI4 interface protocol. The AXI4 interface protocol allows the processor algorithm to read vector data from a contiguous group of registers on the FPGA. Use this block for simple, low-throughput memory-mapped communication, such as reading from control and status registers. This diagram shows a generalized representation of the generated code implementation.
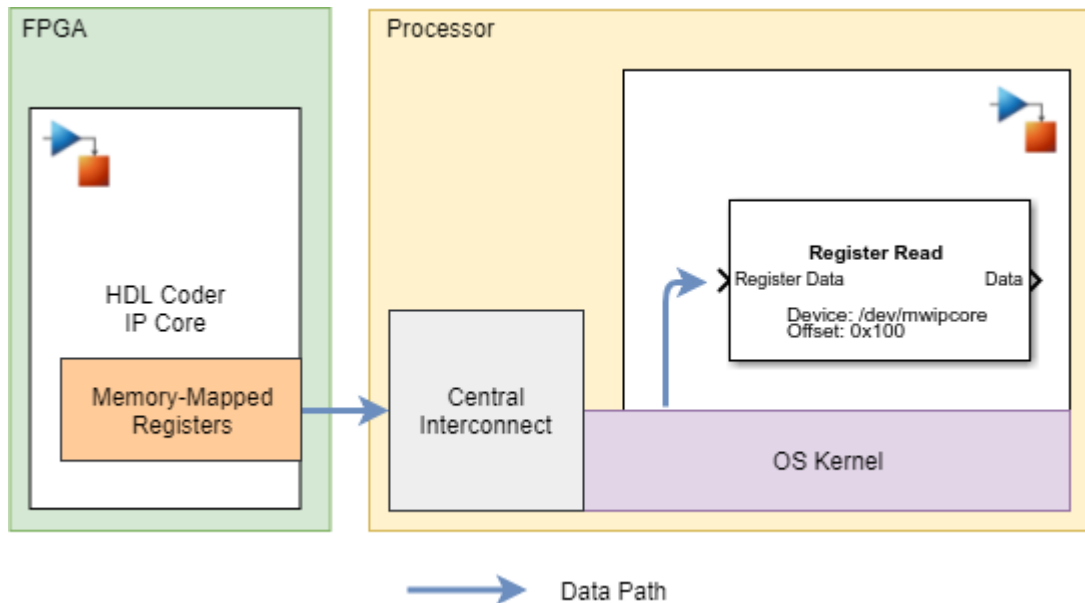
See Also

Register Channel

Introduced in R2019a

Register Write

Write data to a register region on the specified IP core
Library:　　　SoC Blockset / Processor I/O

Description

The Register Write block writes data from your processor algorithm to a register region on the specified IP core. In simulation, a timer-driven or event-driven task subsystem contains the Register Write block. The data signals from the Register Write block connect to the Register Channel block managing those registers and their transactions.

When developing or analyzing the software side of an SoC application, the Register Write block can be connected to an IO Data Sink block. In this configuration, the IO Data Sink block provides either previously recorded or artificial data, enabling a more directed simulation of the software and processor side of the application, without need to explicitly model the hardware and memory interactions.

Ports

Input

data — Data input
vector

This port receives the data vector to write to the registers on the IP core starting at **Offset address** from the base address of the IP core.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

`msg` — Output register data message
scalar

This message port sends the output register data, as a message, to the connected Register Channel or IO Data Sink block. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `SoCData`

Parameters

`Output sink` — Direct output to port or base workspace
`To output port` (default) | `Base workspace`

Select `To output port` to write data to the output port, `msg`. Select `Base workspace` to write data to a variable in the base workspace. When writing to the base workspace, the block updates the value of a `Simulink.Parameter` object with name set by `Simulink.Parameter object name` parameter in the base workspace.

Note Placing the Register Write block inside a Initialize Function block subsystem, writes to a `Simulink.Parameter` object at the start of simulation. A register, represented as a Constant block, in an FPGA reference model can be initialized at the start of simulation with the value of the `Simulink.Parameter` object. This method of writing to FPGA registers requires a constant value throughout the simulation but can reduce the simulation time required by your SoC model.

`Simulink.Parameter object name` — Name of Simulink.Parameter object
A (default) | character vector

Name of `Simulink.Parameter` object to be created in the `Base workspace`.

Example: A

`Device name` — Path and file name of IP core device
`/dev/mwipcore` (default) | character array

Enter the path and file name of the IP core device.

Offset address — Offset from the base address of the IP core to the register
hex2dec('0100') (default) | positive integer

Enter the offset from the base address of the IP core to the register. The block writes data to this register. Use the hex2dec function when you specify the offset address using a hexadecimal number expressed as a character vector. The offset address can be selected using the Memory Mapper tool.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

SoC Builder implements the Register Write block with FPGA and processor IPs that use the AXI4 interface protocol. The AXI4 interface protocol allows the processor to write vector data from the processor to a contiguous group of registers on the FPGA. Use this block for simple, low-throughput memory-mapped communication, such as writing to control and status registers. This diagram shows a generalized representation of the generated code implementation.
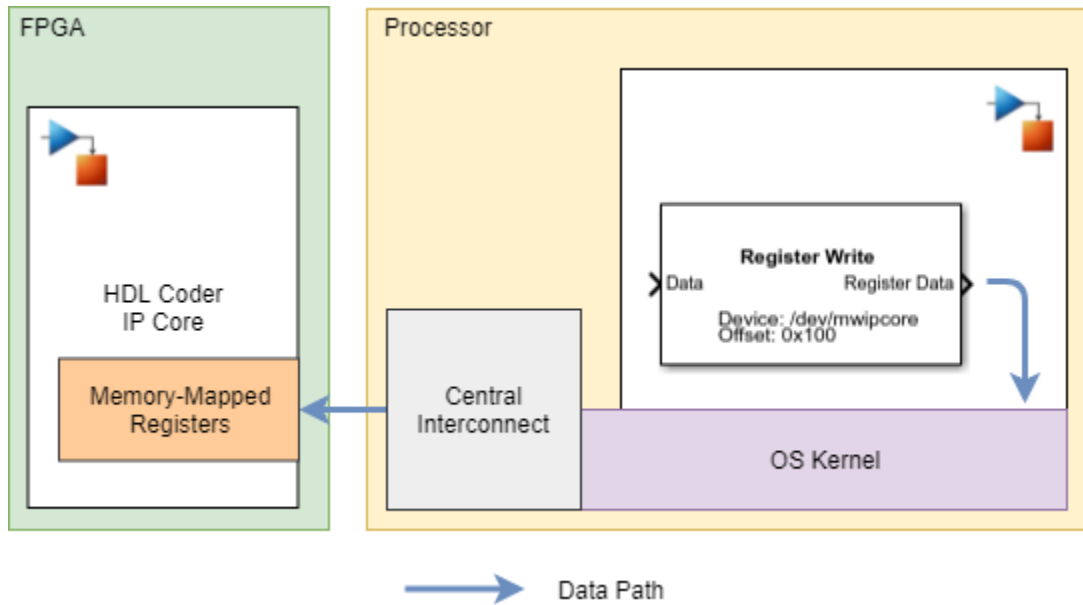
See Also

Register Channel | Register Read

Introduced in R2019a

Stream Read

Stream data to processor algorithms
Library:　　　SoC Blockset / Processor I/O

Description

The Stream Read block streams data from shared memory in the memory channel to your processor algorithm. In simulation, a timer-driven or event-driven task subsystem contains the Stream Read block. The data signals from the Memory Channel block connect to the Stream Read block. Following a write to the shared memory, the Memory Channel notifies the Task Manager block of the write event. The Task Manager block then triggers the event-driven subsystem containing the Stream Read block and associated algorithm.

When developing or analyzing the software side of an SoC application, the Stream Read block can be connected to an IO Data Source block. In this configuration, the IO Data Source block provides either previously recorded or artificial data, enabling a more directed simulation of the software and processor side of the application, without need to explicitly model the hardware and memory interactions.

Ports

Output

data — Data frame from shared memory
vector

This port emits a data frame read from shared a region of shared memory defined in the Memory Channel block.

Data Types: `uint32`

valid — Valid frame data
scalar

A flag indicating a valid data frame read from the memory channel.

Data Types: `Boolean`

done — Notification message of completed read
scalar

This message port sends notification, as a message, to the connected Register Channel or IO Data Sink block that the read completed. For more information on messages, see "Message-Based Communication" (Simulink).

Data Types: `Boolean`

Input

msg — Data message from memory
scalar

This message port receives data messages from the connected Memory Channel or IO Data Source block. The messages process when the Task Manager block triggers the task containing the Stream Read block. For more information on messages, see "Message-Based Communication" (Simulink).

Dependencies

This port appears when `Simulation output` is set to `From input port`.

Data Types: `SoCData`

Parameters

Main

Device name — File name of IP core device
`ip:s2mm0` (default) | character array

Enter the name and channel of the IP core on the FPGA as a colon separated list.

`Output data type` — Data type of IP core
`uint32` (default)

Enter the data type used by the memory channel.

`Samples per frame` — Size of data vector read from IP core
`1024` (default) | positive scalar integer

Enter the size of the data vector read from the memory channel.

`Number of buffers` — Number of data buffers
`16` (default) | positive integer

Enter the number of data frame buffers in physical memory.

`Enable event-based execution` — Enable event-driven task execution
`on` (default) | `off`

To use this block in event-driven task subsystems, select this parameter. To use this block in timer-driven task subsystems, clear this parameter.

When **Enable event-based execution** is selected, this block reads from the Memory Channel each time a write occurs to that shared memory region. When **Enable event-based execution** is cleared, the block reads the data in the shared memory region at each sample time.

`Sample time` — Sample time in seconds
`-1` (default) | positive scalar

Enter the sample time used by the timer-driven task subsystem when the **Enable event-based execution** is cleared.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

SoC Builder implements the Stream Read, Memory Channel, and Task Manager blocks with FPGA and processor IPs that use the AXI4-stream communication protocol. The AXI4-stream protocol uses a direct memory access (DMA) to read a data vector to a shared region on the external memory. This protocol allows for high-speed streaming of data between the FPGA and the embedded processor through external memory. This diagram shows a generalized representation of the generated code implementation.
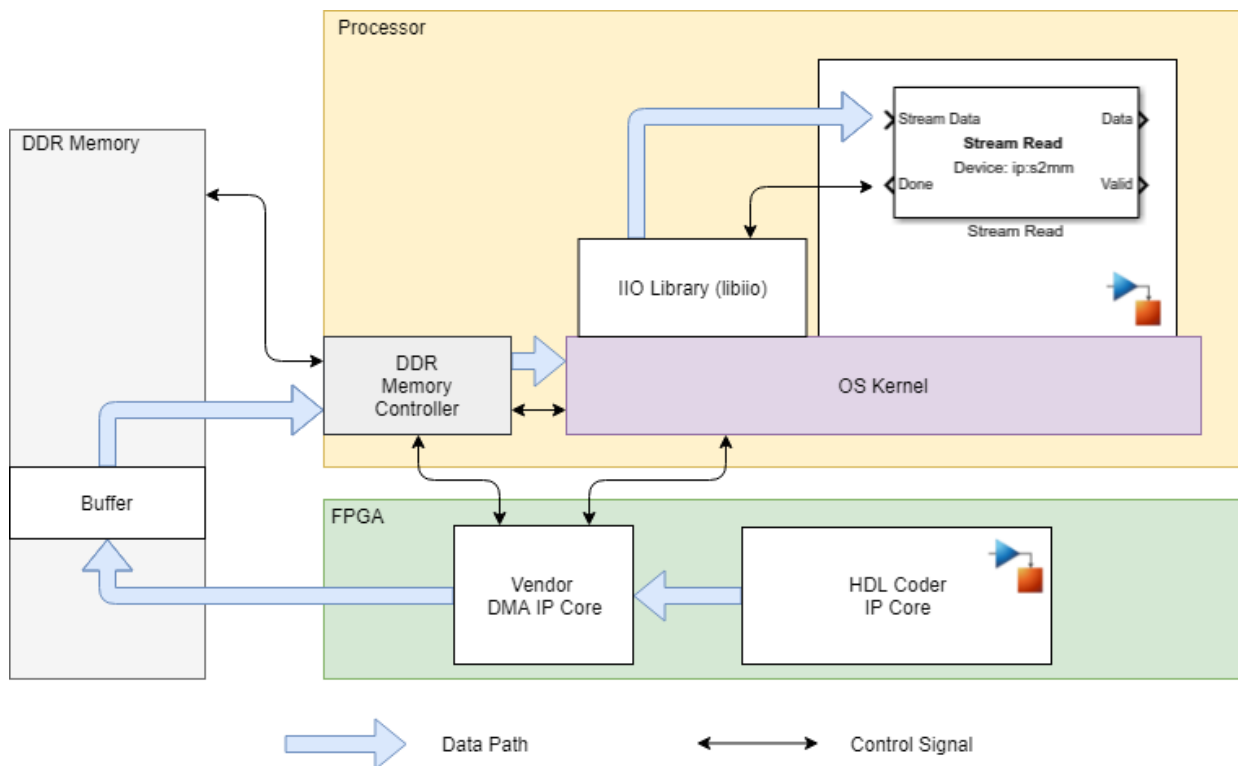
See Also
Memory Channel | Task Manager

Topics
"Packet-Based ADS-B Transceiver"
"Event-Driven Tasks"

Introduced in R2019a

TCP Read

Receive TCP/IP packets from remote host over TCP/IP network
Library: SoC Blockset / Processor I/O
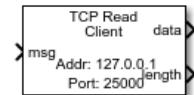
Description

The TCP Read block receives a stream of TCP/IP packets from a remote host over a
TCP/IP (Transmission Control Protocol/Internet Protocol) network.

Ports

Input

msg — Stream of TCP/IP packets received from the remote host
scalar

This message port receives TCP/IP packets, as messages, from a connected IO Data
Source block. The messages process when the Task Manager block triggers task
containing the TCP Read block. For more information on messages, see "Message-Based
Communication" (Simulink).

Note This input is used only during simulation. and does nothing in code generation and
external mode simulation.

Data Types: `SoCData`

Output

data — TCP/IP packet received from remote host
numeric vector

Output TCP/IP packets received from remote host, returned as a numeric vector. The size and data type of this output is same as the size and data type of the input message.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`length` — Length of output TCP/IP packet
nonnegative scalar

Length of output TCP/IP packets returned on the output **data** port.

Data Types: `uint32`

Parameters

`Network role` — Set block as client or server
`Client` (default) | `Server`

To configure this block as a TCP/IP client or server, set this parameter to `Client` or `Server`, respectively.

When you set this parameter to `Client`, you must provide the remote IP address and remote IP port number of the TCP/IP server from which you want to receive TCP/IP packets. Specify this information by using the **Remote address** and **Remote port** parameters.

When you set this parameter to `Server`, you must provide the local IP port number, which acts as the listening port of the TCP/IP server running in the hardware. Specify this information using the **Local port** parameter. When you set this parameter to `Server`, you can only connect to one client at a time.

`Remote address` — IP address of remote server from which TCP/IP packets are received
`127.0.0.1` (default) | dotted-quad expression

Specify the IP address of remote server from which you want to receive TCP/IP packets.

Dependencies

To enable this parameter, set the **Network role** parameter to `Client`.

`Remote port` — IP port on remote server from which TCP/IP packets are received
`25000` (default) | integer from 1 to 65535

Specify the port number of the remote server from which you want to receive TCP/IP packets.

Dependencies

To enable this parameter, set the **Network role** parameter to `Client`.

`Local port — IP port of host on which data is received`
`-1` (default) | integer from 1 to 65,535

Specify the port number of the application on which you want to receive the TCP/IP packets when the **Network role** is set to `Client`. The default value `-1` assigns any random available port as local port when you set the **Network role** parameter to `Client`.

This local port acts as the listening port on the TCP/IP server when the **Network role** is set to `Server`. Specify a value from 1 to 65535 when you set **Network role** parameter to `Server`. Specify this local port number as the remote port number in the sending host from which you want to receive TCP/IP packets.

`Data type — Data type of TCP/IP packets received`
`uint8` (default) | `single` | `double` | `int8` | `int16` | `int32` | `uint16` | `uint32`

Select the data type of the input data. Match this data type with data type of TCP/IP packets sent from the remote host.

`Maximum data length (elements) — Maximum length of output TCP/IP packet`
`1` (default) | positive scalar

Specify the maximum number of data elements that the output **data** port can produce at every time step.

`Enable event-based execution — Enable event-based task execution`
off (default) | on

To generate event-driven code, select this parameter. To generate timer-driven code, clear this parameter.

When **Enable event-based execution** is selected, the block reads TCP/IP packets from the socket buffer whenever any TCP/IP packet is received in the socket buffer irrespective of the sample time. When **Enable event-based execution** is cleared, the block reads available TCP/IP packets from the socket buffer at each sample time. To set the size of the TCP/IP packet that the block can read from the socket buffer, specify the size in the **Receive buffer size** parameter.

1-125

Sample time — Sample time
-1 (default) | nonnegative scalar

Specify how often the scheduler runs this block. If this value is -1 (default), the scheduler assigns the sample time for the block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder generates event-driven or timer-driven code for this block based on the **Enable event-based execution** parameter selection. This diagram shows a generalized representation of the generated code implementation.

Note Timing measurements from generated code might vary within the execution of a task instance compared to the timing of tasks in simulation. You can configure your model to use data caching in task signals to reach improved agreement between the simulation and generated code. For more information, see Value and Caching of Task Subsystem Signals.

See Also
IO Data Source | TCP Write | Task Manager

Introduced in R2019a

TCP Write

Send TCP/IP packets to remote host over TCP/IP network
Library: SoC Blockset / Processor I/O
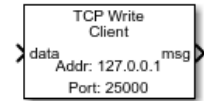
Description

The TCP Write block sends TCP/IP packets to a remote host over a TCP/IP (Transmission Control Protocol/Internet Protocol) network.

Ports

Input

data — Input data
numeric vector

Input data, specified as a numeric vector. The block sends this data over a TCP/IP network to the remote host.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

msg — Stream of TCP/IP packets sent to the remote host
scalar

This message port sends TCP/IP packets, as messages, to a connected IO Data Sink block. For more information on messages, see "Message-Based Communication" (Simulink).

Note This output is used only during simulation. and does nothing in code generation and external mode simulation.

Data Types: `SoCData`

Parameters

`Network role` — Set the block as server or client
`Client` (default) | `Server`

To configure this block as a TCP/IP client or server, set this parameter to `Client` or `Server`, respectively.

When you set this parameter to `Client`, you must provide the remote IP address and remote IP port number of the TCP/IP server to which you want to send TCP/IP packets. Specify this information by using the **Remote address** and **Remote port** parameters.

When you set this parameter to `Server`, you must provide the local IP port number, which acts as the listening port of the TCP/IP server running in the hardware. Specify this information using the **Local port** parameter.

`Remote address` — IP address of remote server to which TCP/IP packets are sent
`127.0.0.1` (default) | dotted-quad expression

Specify the IP address of the remote server to which you want to send TCP/IP packets.

Dependencies

To enable this parameter, set the **Network role** parameter to `Client`.

`Remote port` — IP port of remote server to which TCP/IP packets are sent
`25000` (default) | integer from 1 to 65,535

Specify the port number of the remote server to which you want to send TCP/IP packets.

Dependencies

To enable this parameter, set the **Network role** parameter to `Client`.

`Local port` — IP port on sending host from which TCP/IP packets are sent
`-1` (default) | integer from 1 to 65535

When the **Network role** parameter is set to `Client`, specify the IP port number of the application from which you want to send TCP/IP packets. The default value `-1`, sets this IP port number to a random available port number and uses that port to send the packets.

When the **Network role** parameter is set to `Server`, this local port acts as the listing port of the TCP/IP server running in the hardware. In this case, specify a value from 1 to 65,535 for this parameter.

Byte order — Byte order
LittleEndian (default) | BigEndian

Byte order of the TCP/IP packets, specified as one of these values:

- `LittleEndian` — Sets the byte order of TCP/IP packets to little endian.
- `BigEndian` — Sets the byte order of TCP/IP packets to big endian.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder generates event driven code for this block. This diagram shows a generalized representation of the generated code implementation.

Note Timing measurements from generated code might vary within the execution of a task instance compared to the timing of tasks in simulation. You can configure your model to use data caching in task signals to reach improved agreement between the simulation and generated code. For more information, see Value and Caching of Task Subsystem Signals.

See Also
IO Data Sink | TCP Read | Task Manager

Introduced in R2019a

UDP Read

Receive UDP packets from remote host
Library: SoC Blockset / Processor I/O
                 SoC Blockset / Host I/O
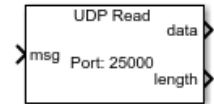
Description

The UDP Read block receives UDP (User Datagram Protocol) packets from a remote host on the application on target. The remote host is the computer or hardware from which you want to receive UDP packets. The block reads UDP packets from UDP socket buffer and returns the UDP packets as a one-dimensional array.

Ports

Input

msg — UDP packet
numeric vector

This message port receives UDP packets, as messages, from a connected IO Data Source block. The messages process when the Task Manager block triggers task containing the UDP Read block. For more information on messages, see "Message-Based Communication" (Simulink).

Note This input is used only during simulation. and does nothing in code generation and external mode simulation.

Data Types: `SoCData`

Output

`data` — Output UDP packet
numeric vector

Output UDP packet, received from a remote host, returned as a numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`length` — Length of received UDP packet
nonnegative scalar

Length of output UDP packet returned on the output **data** port.

Data Types: `uint32`

Parameters

`Local port` — IP port number of local host
25000 (default) | integer from 1 to 65,535

Specify the port number of the application on target in which you want to receive data. Match the local IP port number with the remote IP port number of the remote host.

`Data type` — Data type of received data
uint8 (default) | single | double | int8 | int16 | int32 | uint16 | uint32

Select the type of data the block receives from the sending host. Match the data type with data type of input data.

`Maximum data length (elements)` — Maximum length of output UDP packet
1 (default) | positive integer

Specify the maximum number of data elements that the output **data** port can produce at every time step.

`Receive buffer size (bytes)` — Number of data bytes in received data
65535 (default) | integer from 1 to 65,535

Specify the maximum number of data bytes that the block can receive at each time step.

1-133

Enable event-based execution — Enable or disable event-based task execution
off (default) | on

To generate event-driven code, select this parameter. To generate timer-driven code, clear this parameter.

When **Enable event-based execution** is selected, the block reads data from the socket buffer whenever any UDP data is received in the socket buffer irrespective of the sample time. When **Enable event-based execution** is cleared, the block reads available UDP data from the socket buffer at each sample time. To set the size of the data that the block can read from the socket buffer, specify the size in the **Receive buffer size** parameter.

Sample time — Sample time
-1 (default) | nonnegative scalar

Specify how often the scheduler runs this block. If this value is -1 (default), the scheduler assigns the sample time for the block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder generates event-driven or timer-driven code for this block, based on the **Enable event-based execution** parameter selection. This diagram shows a generalized representation of the generated code implementation.

UDP Data Flow

Note Timing measurements from generated code might vary within the execution of a task instance compared to the timing of tasks in simulation. You can configure your model to use data caching in task signals to reach improved agreement between the simulation and generated code. For more information, see Value and Caching of Task Subsystem Signals.

See Also

IO Data Source | Task Manager | UDP Write

Introduced in R2019a

UDP Write

Send UDP packets to remote host
Library: SoC Blockset / Processor I/O
        SoC Blockset / Host I/O
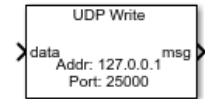
Description

The UDP Write block sends UDP (User Datagram Protocol) packets from the application on target to a remote host. The remote host is the computer or hardware to which you want to send UDP packets.

Ports

Input

data — Input signal
numeric vector

Input data, specified as a numeric vector. The block sends this data as UDP packet to the remote host. To set the byte order in which you want to send this UDP data, set the **Byte order** parameter. The block converts this input data to the specified byte order type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

msg — UDP packet sent to remote host
numeric vector

This message port sends UDP packets, as messages, to a connected IO Data Sink block. For more information on messages, see "Message-Based Communication" (Simulink).

> **Note** This output is used only during simulation. and does nothing in code generation and external mode simulation.

Data Types: `SoCData`

Parameters

Remote IP address (255.255.255.255 for broadcast) — IP address of remote host to which data is sent
`127.0.0.1` (default) | dotted-quad expression

Specify the remote IP address of the host to which you want to send UDP packets.

Remote port — IP port of remote host to which data is sent
`25000` (default) | integer from 1 to 65,535

Specify the port number of the host to which you want to send UDP packets.

Local port — IP port number of application on target from which data is sent
`-1` (default) | integer from 1 to 65,535

Specify the port number of the application on the target from which you want to send the UDP packets. The default value `-1`, sets the local port number to a random available port number and uses that port to send the UDP packets.

Byte order — Byte order
`LittleEndian` (default) | `BigEndian`

Byte order of the UDP packets, specified as one of these values:

- `LittleEndian` — Sets the byte order of UDP packets to little endian.
- `BigEndian` — Sets the byte order of UDP packets to big endian.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder generates event-driven code for this block. This diagram shows a generalized representation of the generated code implementation.

Note Timing measurements from generated code might vary within the execution of a task instance compared to the timing of tasks in simulation. You can configure your model

to use data caching in task signals to reach improved agreement between the simulation and generated code. For more information, see Value and Caching of Task Subsystem Signals.

See Also

IO Data Sink | Task Manager | UDP Read

Introduced in R2019a

Task Manager

Create and manage task executions in Simulink model
Library: SoC Blockset / Processor Task Execution

Description

The Task Manager block simulates the execution of software tasks as they would be expected to behave on an SoC processor. With the Task Manager, you can add and remove tasks from your model that can either be timer-driven or event-driven. Tasks can be represented in a model as rates, for timer-driven tasks, or function-call subsystems, for event-driven tasks, contained inside a single Model block. The Task Manager executes individual tasks based on their parameters, such as period, duration, trigger, priority, or processor core, and the combination of that task with the state of other tasks and their priorities in the running model.

Note The Task Manager block cannot be used in a referenced model. For more information on referenced models, see Model block.

The Task Manager block provides three methods to specify the duration of a task in simulation:

- A probability model of task duration defined in the block mask.
- From a data file recording of either a previous task simulation or from a task on an SoC device.
- Input ports on the block, which you can connect to more dynamic models of task duration.

Ports

Output

Task1 — **Function-call from Task1**
scalar

A function-call signal that can trigger timer-driven and event-driven tasks, represented as rate or function-call subsystems in the processor Model block, respectively.

For a rate port from a timer-driven subsystem, to show on the Model block, set the **Block Parameters** > **Main** > **Schedule rates** and select ports. For a function-call port from an event-driven subsystem contained in a Function-Call Subsystem block to show on the Model block, include an Inport in the processor Model block connected to the function-call trigger port of the subsystem. In the Inport, check **Block Parameters** > **Signal Attributes** > **Output function call**.

Note The `Task1` port must be connected to either a function-call port or scheduled rate signal port on a Model block.

Dependencies

To create or remove a control signal port for a task, add or remove the task from the Task Manager block by clicking the **Add** or **Delete** buttons in the block dialog mask.

Input

*Task1*Event — **Message event notification**
scalar

A message port that triggers the associated event-driven task. The `Task1Event` port receives the message from either a Memory Channel block or IO Data Source block. For more information on messages, see "Message-Based Communication" (Simulink).

Dependencies

To show a *Task1*Event port, then *Task1* must have **Type** set to `Event-driven`.

Data Types: `rteEvent`

Task1`Dur` — **Task duration**
positive scalar

A positive value signal that specifies the execution duration of a task at the present time. For more information on specifying task duration, see "Task Duration".

Dependencies

To enable this port, set the **Specify task duration via** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

`Enable task simulation` — **Enable simulation of task duration**
`on` (default) | `off`

Enable or disable the simulation of task duration. If you clear this parameter, tasks simulate using a function-call generator inheriting their period from the fundamental sample time of the model for event-driven tasks or from the dialog for timer-driven tasks.

`List of tasks` — **List of tasks**
`Task1` (default)

List of the tasks generated by the Task Manager block. Each task has a set of parameters listed in the **Main** and **Simulation** tabs of the block dialog mask.

`Add` — **Add task**
button

Add a task to the Task Manager block. During deployment, each task is encapsulated as an execution thread in the generated code. The properties of the thread are taken from the **Main** parameters for that task. During simulation, the task uses a combination of the **Main** and **Simulation** parameters for that task.

`Delete` — **Delete existing task**
button

Remove a task from the Task Manager.

Dependencies

To enable this parameter, specify at least two tasks.

Main

Name — Name of task
Task1 (default) | character vector

Unique name of the task. The task name must only contain alphanumeric characters and underscores.

Type — Trigger type of task
Timer-driven (default) | Event-driven

Specify the task as timer-driven or event-driven. For more information on timer- and event-driven tasks, see "Timer-Driven Task" and "Event-Driven Tasks", respectively.

Dependencies

To enable this parameter, set Type to Timer-driven.

Period — Timer period
0.1 (default) | positive scalar

Specify the trigger time period for timer-driven tasks.

Core — Processor core to execute task
0 (default) | non-negative integer

Specify the number of the processor core where a task executes. For more information on selecting cores and core execution visualizations, see "Multicore Execution and Core Visualization".

Priority — Priority of task in scheduler
10 (default) | positive integer

Specify the schedulers priority for the event-driven task between 1 and 99. Higher priority tasks can preempt lower priority tasks, and vice versa. The task priority range is limited by the hardware attributes. For more information on task priority, see "Task Priority and Preemption".

Dependencies

To enable this parameter, set Type to Event-driven.

Drop tasks that overrun — Drop tasks that overrun
off (default) | on

Select this parameter to force tasks to drop, rather than catch up, following an overrun instance. For more information on task overruns, see "Task Overruns and Countermeasures".

Simulation

`Play recorded task execution sequence` — Enable playback from file
off (default) | on

Select this parameter for the Task Manager block to play back the recorded execution data provided from the specified **File name** parameter. For more information on replaying task execution, see "Task Execution Playback using Recorded Data".

`Specify task duration via` — Source of task execution time
`Dialog` (default) | `Input port` | `Record task execution statistics`

Specify the source of the timing information for the task execution.

* `Dialog` - Use a normally distributed probabilistic model with **Mean**, **Deviation**, **Min**, and **Max** defined in the block dialog mask.
* `Input port` – When set from *Input port*, the block input port dynamically defines the execution duration.
* `Record task execution statistics` – Use a normally distributed probabilistic model with mean and deviation provided in file specified by **File name**.

For more information on configuring task duration, see "Task Duration".

Task duration settings

`Add` — Adds distribution
button

Adds a distribution to the set of normal distributions that generates an execution duration. For more information on configuring task duration, see "Task Duration".

Note Only a maximum five distributions can be assigned to a single task.

`Delete` — Remove distribution
button

Remove a distribution from the set of normal distributions.

`Percent` — Likelihood of distribution
100 (default) | positive scalar

Specify the likelihood of each normal distribution. The **Percent** weighted sum of normal distributions determines the task duration likelihood. For more information on configuring task duration, see "Task Duration".

Note The sum of **Percent** for all the distributions in a single task must equal 100.

Mean — Mean task duration in simulation
1e-06 (default) | positive scalar

Specify the mean duration of the task during simulation of the task. The simulated task duration uses a normal distribution with a specified **Mean** and **SD** parameter values as a first-order approximation of the task behavior. For more information on configuring task duration, see "Task Duration".

SD — Standard deviation of task duration in simulation
0 (default) | positive scalar

Specify the standard deviation duration of the task during simulation of the task. The simulated task duration uses a normal distribution with a specified **Mean** and **SD** as a first-order approximation of the task behavior. For more information on configuring task duration, see "Task Duration".

`Min` — Lower limit of task duration
1e-06 (default) | positive scalar

Lower limit of a task duration distribution. For more information on configuring task duration, see "Task Duration".

`Max` — Upper limit of task duration
1e-06 (default) | positive scalar

Upper limit of a task duration distribution. For more information on configuring task duration, see "Task Duration".

`File name` — File containing diagnostic scheduling data
filepath

The data in this file specifies the **Mean** and **SD** parameter values. When the **Play recorded task execution sequence** parameter is selected, the specified CSV file provides the explicit task execution timing. The CSV file contains the diagnostic data of the task scheduler previously recorded from the hardware board. For more information on configuring task duration, see "Task Duration".

Dependencies

To enable this parameter, set the **Specify task duration via** parameter to `Recorded task execution statistics`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

The tasks in the Task Manager block execute as threads in the generated code. The task parameters in the Task Manager block specify the priority and execution core of the thread.

See Also
I/O Data Source | Memory Channel

Topics
"What is Task Execution?"
"Task Duration"

Introduced in R2019a

Proxy Task

Placeholder for timer-driven task
Library: SoC Blockset / Processor Task Execution

Proxy Task

Description

The Proxy Task block simulates a timer-driven task in your SoC application without and allows you to model a task without explicitly implementing one. This block can be used as a placeholder for timer-driven tasks that will be developed in the future, or by another developer.

When added to your processor reference Model block, it causes the processor model, when the model is set to schedule rates by ports, to display a periodic event port with a sample time equal to **Sample Time** parameter.

Parameters

`Sample Time` — Sample time
1 (default) | positive scalar

Sample time of the block.

Note

- The `Sample Time` of the Proxy Task block must match the sample time of the corresponding timer-driven task from the Task Manager block.

- The `Sample Time` of the Proxy Task block should be unique within the model. When other blocks in the model use the same sample time, the duration of the task defined by Proxy Task block cannot be guaranteed in code generation.

See Also

Task Manager | Testbench Task

Topics

"Timer-Driven Task"

Introduced in R2019b

IO Data Sink

Record, output, or terminate input message
Library: SoC Blockset / I/O Data Source and Sink
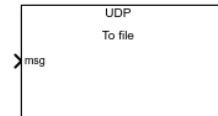
Description

The IO Data Sink block records, outputs, or terminates the received input message signal. The input of this block connects to the output of the TCP Write, UDP Write, or Register Write block. This block enables you to save the received input data to a file that you can play back using the IO Data Source block in the model. You can also terminate the signal or output the signals through an output port which can be fed as an input to IO Data Source block.

Ports

Input

msg — SoC Message Data
numeric vector

This port receives the data vector from the **msg** port of the processor io blocks, which includes Register Read, Stream Read, UDP Read and TCP Read blocks.

Data Types: `SoCData`

Output

data — Output data
numeric vector

Output data, returned as a numeric vector. The block converts the received input message into a data signal.

Data Types: `uint32` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `int64` | `uint64` | `Boolean` | `fixedpoint`

`length` — **Length of output data**
nonnegative numeric scalar

Output data length, returned as a nonnegative numeric scalar.

Data Types: `double`

`valid` — **Indication of valid data**
Boolean scalar

Control signal that indicates whether the output data is valid. When this value is 1 (true), the value on the output **data** port is valid .

Data Types: `Boolean`

`done` — **Data streaming in progress**
Boolean scalar

The block sets **done** to 1 when there is no more stream output **data** to return. The block sets **done** to 0 when the block has more stream data to return.

Dependencies

To enable this port, set the **Device type** parameter to `Stream`.

Data Types: `Boolean`

Parameters

`Output sink` — **Sink of output data from the block**
`To file` (default) | `To output port` | `To terminator`

Set the sink of output data from the block by selecting one of these values:

- `To file` — Save output data to a file.
- `To output port` — Output data and signals by using output ports on the block.
- `To terminator` — Terminate the received input signal.

`Device type` — Device type selection
UDP (default) | TCP | Register | Stream

Select a device type to enable the corresponding input data port.

- UDP — Enables the **msg** input port to receive UDP data as message from **msg** port of UDP Read block.
- TCP — Enables the **msg** input port to receive TCP data as message from **msg** port of TCP Read block.
- Register — Enables the **msg** input port to receive Register data as message from **msg** port of Register Read block.
- Stream — Enables the **msg** input port to receive Stream data as message from **msg** port of Stream Read block.

`Sample time` — Time interval of sampling
-1 (default) | nonnegative numeric scalar

Specify a discrete time interval, in seconds, at which the block outputs data. The default value -1 inherits the sample time from the solver used for simulating the model.

`Dataset name` — Name of data file
no default | file path

Specify the full path to where you want to save the file on the host PC. This block saves the output data as a TGZ file. You can import this file into the model by using the IO Data Source block.

Dependencies

To enable this parameter, set the **Output sink** parameter to To file.

`Source name` — Name of dataset
no default

Specify a name for the output data source in which to save the data in the dataset file.

Dependencies

To enable this parameter, set the **Output sink** parameter to To file.

`Data type` — Data type of output data
uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | boolean | fixedpoint

1-151

Select the data type of the output data. Match this value with the data type of input data.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder does not generate code for this block. In the generated code, the device I/O connects directly to the TCP Write, UDP Write, or Register Write block.
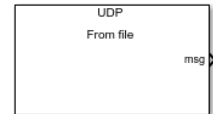
See Also
IO Data Source | Register Write | TCP Write | Task Manager | UDP Write

Introduced in R2019a

IO Data Source

Play back recorded data
Library:　　　SoC Blockset / I/O Data Source and Sink

Description

The IO Data Source block enables you to import recorded hardware IO data and play it back in your Simulink® model. The block converts the input data into a message signal that you can give as input to the TCP Read, UDP Read, Stream Read, or Register Read blocks, depending on the device type you choose. The playback of hardware IO data in your Simulink model helps you develop models with better accuracy than models developed by using randomly generated data during simulation.

When you develop models that use real hardware IO data during deployment, you can choose to use randomly generated synthetic data as hardware IO data in simulation. As physical hardware data accounts for various effects like data loss, time delay, and so on. If you use synthetic data as hardware IO data in simulation for such models, it leads to unexpected results when you deploy it in the hardware board. To evaluate and verify such models, using real hardware IO data during simulation is recommended. For more information on how to record hardware IO data and save it to your host computer, see the `soc.recorder` object.

Note If you have a IO Data Source block with **Input Source** set to `From file`, associated with a `Timer-driven` Task Manager block in your model and you plan to use a fixed-step solver, then enter a step size value lesser than the value set for the **Period** parameter in the Task Manager block. For example, suppose the value of **Period** specified in the Task Manager block is `0.1`, then choose a fixed-step size less than `0.1`.

Ports

Input

data — Input data
numeric vector

Input data, specified as a numeric vector. The block converts this data into a bus signal of the specific device type specified by the **Device type** parameter. Match the data type of this input data with the data type you select in the **Data type** parameter. The output bus signal consists of data values, length of data, and valid status of data.

Dependencies

To enable this port, set the **Input source** parameter to `From input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

length — Length of input data
nonnegative scalar

Data length, specified as a scalar.

Dependencies

To enable this port, set the **Input source** parameter to `From input port`.

Data Types: `uint32`

valid — Valid data signal
`boolean` scalar

When **valid** is 1, the block captures the input data from the **data** and **length** ports. When **valid** is 0, the block considers the input data as invalid and does not capture it.

Dependencies

To enable this port, set the **Input source** parameter to `From input port`.

Data Types: `Boolean`

done — Done
`boolean` scalar

When **done** is `true`, the block stops capturing the input data from the **data** and **length** ports. When **done** is `false`, the block captures the input data from the **data** and **length** ports.

Dependencies

To enable this port, set the **Device type** parameter to `Stream`.

Data Types: `Boolean`

Output

event — Task event signal
scalar

This port sends a task event signal that triggers the Task Manager block to execute the associated event-driven task.

Dependencies

To enable this port, select **Show event port** parameter.

Data Types: `rteEvent`

msg — SoC message data
numeric vector

This port sends the data vector as a message to the **msg** port of Processor io blocks, which includes Register Read, Stream Read, UDP Read and TCP Read blocks.

Data Types: `SoCData`

Parameters

Input Source — Source of input data
From file (default) | From dialog | From input port

Set the input data source for the block by selecting one of these values:

- `From file` — Read data from a recorded data file at the same time interval at which it was recorded on the hardware board.

1-155

- `From dialog` — Input a one-dimensional array of data by using a function. Specify this function in the **Value** parameter box.
- `From input port` — Input data and signals using input ports on the block.

Value — Value of source data

`uint32(1:1024)` (default) | function to generate input data

Specify a MATLAB function that creates a row vector of numeric data. This row vector is captured as the input data for the block.

Dependencies

To enable this parameter, set the **Input source** parameter to `From dialog`.

Data type — Data type of input data

`uint32` (default) | `double` | `single` | `int8` | `uint16` | `int32` | `boolean` | `fixed point`

Select the data type of the input data to be received by the **data** port.

Dependencies

To enable this parameter, set the **Input source** parameter to `From file`.

Device type — Device type of input data source

`UDP` (default) | `TCP` | `Register` | `Stream`

Select a device type to enable the corresponding output data port.

- `UDP` — Enables the **msg** output port to output UDP data as a message.
- `TCP` — Enables the **msg** output port to output TCP data as a message.
- `Register` — Enables the **msg** output port to output Register data as a message.
- `Stream` — Enables the **msg** output port to output Stream data as a message.

Dependencies

To enable this parameter, set the **Input source** parameter to `From input port` or `From dialog`.

Sample time — Time interval of sampling

`-1` (default) | nonnegative scalar

Specify a discrete time interval in seconds at which the block outputs data.

Dependencies

To enable this parameter, set the **Input source** parameter to `From dialog`.

Dimensions — Samples per frame
1024 (default) | nonnegative scalar

Specify the size of the input data. The block reads this number of samples per frame during reading and playback in simulation.

Dependencies

To enable this parameter, set the **Input source** parameter to `From file`.

Dataset name — Name of recorded file
no default | file path

Specify the full path to the recorded data file on the host PC or browse and select the file on the host PC. This block supports only TGZ files created by using the SoC Blockset™ data recording API.

Dependencies

To enable this parameter, set the **Input source** parameter to `From file`.

Source name — Name of dataset
no default

Specify the dataset source name you want to use as the input source available within the recorded data specified in the **Dataset name** parameter. You can either type the name in the **Source name** box or click **Select** to select the name from the list of sources available in the recorded data file.

Dependencies

To enable this parameter, set the **Input source** parameter to `From file`.

Receive queue length — Queue length of input data
1 (default) | nonnegative scalar

Specify the number of data elements to store in the input data queue.

Dependencies

To enable this parameter, set the **Device type** parameter to `Stream`.

1-157

Show event port — Enable Event port
off (default) | on

Select this parameter to enable the **Event** port. Clear this parameter, to disable the **Event** port.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the **SoC Builder** tool. See "Generate SoC Design". You must have an Embedded Coder license to generate and execute C code for your SoC device.

Embedded Coder does not generate code for this block. In the generated code, the device I/O connects directly to the TCP Read, UDP Read, Stream Read, or Register Read block.

See Also
IO Data Sink | Register Read | Stream Read | TCP Read | Task Manager | UDP Read | soc.recorder

Introduced in R2019a

Testbench Task

External timer-driven task load for SoC processor application
Library:　　　SoC Blockset / Processor Testbench
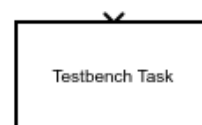
Description

The Testbench Task block simulates a timer-driven task load external to your software application. Using Testbench Task blocks, you can simulate the impact of your software application in the presence of other processes that compete for execution time of the processor.

Note As part of a processor test bench, the Testbench Task block must be placed in the top level of your SoC model.

Ports

Input

Timer Task Function Call — Function-call input port from a timer-driven task
scalar

Function-call event signal from a timer-driven task event port of the Task Manager block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Embedded Coder does not generate code for this block.

See Also

Proxy Task | Task Manager

Topics

"Timer-Driven Task"

Introduced in R2019b

2

Configuration Parameters

Hardware Implementation Pane

Hardware Implementation Pane Overview

Task Profiling in Simulation

Parameter	Description	Default Value
"Show in SDI" on page 2-8	Show task execution data collected in simulation in Simulation Data Inspector.	`on`
"Save to file" on page 2-8	Save task execution data to a file.	`on`
"Overwrite file" on page 2-8	Overwrite last task execution data file.	`off`

Task Profiling on Processor

Parameter	Description	Default Value
"Show in SDI" on page 2-9	Show task execution data collected on hardware in Simulation Data Inspector.	`off`
"Save to file" on page 2-9	Save task execution data to a file.	`off`
"Overwrite file" on page 2-9	Overwrite last task execution data file.	`off`
"Instrumentation" on page 2-9	Choose to perform code instrumentation or kernel Instrumentation.	`Code`

Operating System/Scheduler Settings

Parameter	Description	Default Value
"Kernel Latency" on page 2-10	Simulated kernel latency delay.	`0`

Task and memory simulation

Parameter	Description	Default Value
"Set seed for simulating task duration and memory access" on page 2-11	Set random number generator seed.	`off`
"Seed Value" on page 2-11	Seed for the simulation of task duration deviation.	`default`
"Cache input data at task start" on page 2-11	Cache input data at task start.	`off`

Processor

Parameter	Description	Default Value
"Number of cores" on page 2-12	Set the number of CPU cores in the processor.	`1`

FPGA design (top-level)

Parameter	Description	Default Value
"View/Edit Memory Map" on page 2-13	Choose whether to perform global synthesis or per IP core synthesis.	`Out of Context per IP`
"Include a JTAG master for host-based interaction" on page 2-13	Use host-based scripts with an integrated JTAG master on the target platform.	`on`
"Include processing system" on page 2-13	For processor-based platforms, include the processing system.	`on`
"Interrupt latency (s)" on page 2-13	The latency from hardware asserting an interrupt to the start of the interrupt service routine.	`0.00001`

Parameter	Description	Default Value
"Register configuration clock frequency (MHz)" on page 2-14	The system configuration clock drives the configuration register interfaces for the vendor IP cores in the system.	50
"IP core clock frequency (MHz)" on page 2-14	The clock for all Simulink based generated HDL IP cores.	100

FPGA design (mem controllers)

Parameter	Description	Default Value
"Controller clock frequency (MHz)" on page 2-15	Frequency of datapath between memory interconnect and memory controller.	200
"Controller data width (bits)" on page 2-15	Bit width of datapath between memory interconnect and memory controller.	64
"Bandwidth derating (%)" on page 2-15	For every 100 clocks, will hold off all transaction execution for this number of clocks.	2.3
"First write transfer latency (clocks)" on page 2-15	Number of clock cycles between write request and start of transfer.	4
"Last write transfer latency (clocks)" on page 2-16	Number of clock cycles between the end of write transfer and completion of transaction.	4
"First read transfer latency (clocks)" on page 2-16	Number of clock cycles between read request and start of transfer.	5

Parameter	Description	Default Value
"Last read transfer latency (clocks)" on page 2-17	Number of clock cycles between the end of read transfer and completion of transaction.	1

FPGA design (mem channels)

Parameter	Description	Default Value
"Interconnect clock frequency (MHz)" on page 2-18	Frequency of the master datapath to the interconnect controller in MHz.	200
"Interconnect data width (bits)" on page 2-18	Data width of master datapath to interconnect controller in bits.	64
"Interconnect FIFO depth (num bursts)" on page 2-18	Maximum number of bursts that can be buffered before data is dropped.	12
"Interconnect almost-full depth" on page 2-18	When the almost full depth is reached, the attached channel protocol interface block asserts back pressure to the data source.	8

FPGA design (debug)

Parameter	Description	Default Value
"Memory channel diagnostic level" on page 2-19	The internal operation of the memory channel can be instrumented for debug or diagnostic analysis.	`Basic diagnostic signals`

Parameter	Description	Default Value
"Include AXI interconnect monitor" on page 2-19	Gather performance metrics of the memory interconnect such as data throughput, latency, and number of bursts executed.	off
"Trace capture depth" on page 2-19	Maximum number of Trace entries to be logged in trace mode	1024

Task Profiling in Simulation

Show in SDI

Show task execution data collected in simulation in the Simulation Data Inspector (SDI). For more information on visualizing tasks in SDI, see "Task Visualization in Simulation Data Inspector".

Settings

Default: off

Save to file

Save task execution data to a file. For more information on recording task execution data, see "Recording Tasks for Use in Simulation".

Settings

Default: off

Overwrite file

Overwrite last task execution data file. For more information on recording task execution data, see "Recording Tasks for Use in Simulation".

Settings

Default: off

See Also

Task Profiling on Hardware

Show in SDI

Show task execution data collected on a processor in the Simulation Data Inspector (SDI). For more information on visualizing tasks in SDI, see "Task Visualization in Simulation Data Inspector".

Settings

Default: `off`, `on`

Save to file

Save task execution data to a file. For more information on recording task execution data, see "Recording Tasks for Use in Simulation".

Settings

Default: `off`, `on`

Overwrite file

Overwrite last task execution data file. For more information on recording task execution data, see "Recording Tasks for Use in Simulation".

Settings

Default: `off`, `on`

Instrumentation

Choose the instrumentation method using which task execution data has to be collected from processor to display on SDI. For more information see "Kernel Instrumentation Profiler" and "Code Instrumentation Profiler".

Settings

Default: `Code`, `Kernel`

Kernel Latency

Sets the simulated delay in the start of a task expected by the kernel latency of the operating system (OS). For more information on kernel latency, see "Effect Kernel Latency on Task Execution".

Settings

Default: `default`

See Also

Task and Memory Simulation

Set seed for simulating task duration and memory access

Enable explicit specification of random number seed for task duration simulation.

Settings

Default: `off`

Seed Value

Random number generator seed for the simulation of task duration deviation of the Task Manager block.

Settings

Default: `default`

Cache input data at task start

Cache the data from signals at the start of task execution. Otherwise, evaluate with the signal data at the end of the task execution.

See Also
Task Manager

Processor

Number of cores

Set the number of CPU cores in the processor.

Settings

Default: 1, positive scalar

FPGA design (top-level)

View/Edit Memory Map

Click to view and edit the FPGA memory map.

Include a JTAG master for host-based interaction

Use host-based scripts with an integrated JTAG master on the target platform to initialize configuration registers and memory regions in the generated design. You can also use it to interact with the design while running in order to read back diagnostic information. The JTAG master can be used instead of or in addition to an embedded processor on the target platform.

Settings

Default: on, off

Include processing system

For processor-based platforms, include the processing system. The processing system must be included when using Embedded Coder to generate embedded software.

Settings

Default: off, on

Interrupt latency (s)

The latency from hardware asserting an interrupt to the start of the interrupt service routine.

Settings

Default: 0.00001

Register configuration clock frequency (MHz)

The system configuration clock drives the configuration register interfaces for the vendor IP cores in the system. User-authored Simulink IP cores will utilize the parameter below for its configuration register bus.

Settings

Default: 50

IP core clock frequency (MHz)

The clock for all Simulink-based generated HDL IP cores. A single clock drives all IP and is used for both datapath and configuration register logic.

Settings

Default: 100

FPGA design (mem controllers)

Memory controller pa

Controller clock frequency (MHz)

Frequency of datapath between memory interconnect and memory controller.

The clock rate used to drive transactions to the external memory. The controller clock frequency determines the overall system bandwidth for external memory that must be shared among all the masters in the model.

Settings

Default: 200

Controller data width (bits)

Set the width, in bits, of the datapath between the memory controller and the memory interconnect.

Settings

Default: 64

Bandwidth derating (%)

Model memory transaction inefficiencies specified by a derating percentage value. For every 100 clocks, memory transaction execution is paused for the number of clocks equal to **Bandwidth derating**. To set this parameter, measure the maximum bandwidth on your board and reflect the bandwidth derating from your board in this parameter. See an example in "Analyze Memory Bandwidth Using Traffic Generators".

Settings

Default: 2.3

First write transfer latency (clocks)

Specify the delay, in clock cycles, between a write request and the start of a transfer.

This delay is the number of clock cycles between making a request to the memory controller and until it returns a response. It is reflected in the **Logic Analyzer** waveforms as the time that the memory controller state remains as `BurstAccepted`. For more information about viewing waveforms in simulation, see "Buffer and Burst Waveforms".

To set this value, measure the clock cycles between the burst-request and start of transfer on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Settings

Default: 4

Last write transfer latency (clocks)

Specify the delay in clock cycles between the end of a memory transfer and the end of a write transaction.

To set this value, measure the clock cycles between the end of the burst and the completion of the transaction on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Settings

Default: 4

First read transfer latency (clocks)

Specify the delay, in clock cycles, between a read request and the start of a transfer.

This delay is the number of clock cycles between making a request to the memory controller and until it returns a response. It is reflected in the **Logic Analyzer** waveforms as the time that the memory controller state remains as `BurstAccepted`. For more information about viewing waveforms in simulation, see "Buffer and Burst Waveforms".

To set this value, measure the clock cycles between the burst-request and start of transfer on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Settings

Default: 5

Last read transfer latency (clocks)

Specify the delay in clock cycles between the end of a memory transfer and the end of a read transaction.

To set this value, measure the clock cycles between the end of the burst and the completion of the transaction on your board. For instructions for extracting this information from a hardware execution, see "Configuring and Querying the AXI Interconnect Monitor".

Settings

Default: 1

FPGA design (mem channels)

Interconnect clock frequency (MHz)

Frequency of the master datapath to the interconnect controller in MHz.

Settings

Default: 200

Interconnect data width (bits)

Data width of master datapath to interconnect controller in bits.

Settings

Default: 64

Interconnect FIFO depth (num bursts)

Specify depth of data FIFO, in units of bursts. When the writer has no buffers to write to, the FIFO can absorb data until a buffer becomes available. This value is the maximum number of bursts that can be buffered before data gets dropped.

Settings

Default: 12

Interconnect almost-full depth

Specify a number that asserts a backpressure signal from the channel to the data source. To avoid dropping data, set a high watermark, allowing the data producer enough time to react to backpressure. This number must be smaller than the FIFO depth.

Settings

Default: 8

FPGA design (debug)

Memory channel diagnostic level

The internal operation of the memory channel can be instrumented for debug or diagnostic analysis. When enabled a `diag` output port will be added to the block.

Settings

Default: `Basic diagnostic signals`, `No debug`

Include AXI interconnect monitor

Gather performance diagnostics of the AXI memory interconnect such as data throughput, latency, and number of bursts executed. You can use the AXI master or a processing system on the target to gather the information. When using an AXI master, a host-based script can plot the data using MATLAB. These figures can then be compared against the simulation results.

Settings

Default: `off`

Trace capture depth

Maximum number of Trace entries to be logged in trace mode, choose the depth in powers of 2.

Settings

Default: 1024

Functions

getData

Get data from file reader

Syntax

```
rd = getData(fr,sourceName)
```

Description

`rd = getData(fr,sourceName)` returns the data recorded from the specified source in the file reader. The `fr` input is an `socFileReader` object. The `sourceName` is the source name specified when saving the file by using the `save` object function.

Examples

Read Data from File Reader

Create a file reader to read data from the specified TGZ-compressed file.

```
fr = socFileReader('UDPDataReceived.tgz');
```

Get the data of a specified source from the file using the `getData` function.

```
rd = getData(fr,'UDPDataReceived-Port25000');
```

Input Arguments

fr — File reader
socFileReader object

File reader, returned as an `socFileReader` object.

sourceName — Name of recorded data source
character vector

Name of a recorded data source in fr, specified as a character vector. The function returns the recorded data of this specified source.

Output Arguments

rd — Data from recorded source
timeseries object

Data from recorded source, returned as a timeseries object.

Data Types: timeseries

See Also
record | save | soc.recorder

Introduced in R2019a

setup

Set up hardware for data recording

Syntax

```
setup(dr)
```

Description

setup(dr) sets up any input sources on the SoC hardware board represented by dr to record data. dr is a data recording session on SoC hardware board created using soc.recorder. You must have added at least one source to dr, using the addSource function.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the hw object. The resulting soc.recorder object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
```

```
                      Sources: {}
                    Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
           BlockingTime: 0
    OutputVarSizeSignal: false
             SampleTime: 0.1000
          HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =
```

3-5

```
1×1 cell array

  {'UDPDataOnPort25000'}
```

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

dr — Data recording session specified for SoC hardware board
`soc.recorder` object

Data recording session for specified SoC hardware board, specified as a `soc.recorder` object.

See Also

addSource | record | removeSource | soc.recorder

Introduced in R2019a

addSource

Add a input source to a data recording session

Syntax

```
addSource(dr,src,sourceName)
```

Description

addSource(dr,src,sourceName) adds the specified hardware input source, src to data recording session, dr. dr is a data recording session on SoC hardware board created using soc.recorder.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the hw object. The resulting soc.recorder object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
           BlockingTime: 0
    OutputVarSizeSignal: false
             SampleTime: 0.1000
          HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

3-9

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

    1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

    0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

dr — Data recording session specified for SoC hardware board
soc.recorder object

Data recording session for specified SoC hardware board, specified as a soc.recorder object.

src — Source object for specified input source
soc.iosource object

Source object for specified input source, specified as an soc.iosource object.

sourceName — Name of input source in the data recording session
character vector

Name of input source in the data recording session, specified as a character vector. The function uses this name as the source name when the specified input source is recorded and saved on a dataset file.

Note Setting sourceName to 'all' errors as the sourceName 'all' is used to remove all input sources added to a data recording session using the removeSource function.

See Also
removeSource | soc.iosource | soc.recorder

Introduced in R2019a

removeSource

Remove input source from data recording session

Syntax

```
removeSource(dr,sourceName)
```

Description

`removeSource(dr,sourceName)` removes an already added input source from a data recording session, `dr`.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the `hw` object. The resulting `soc.recorder` object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
           BlockingTime: 0
    OutputVarSizeSignal: false
             SampleTime: 0.1000
         HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)
```

```
recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)
```

```
recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

dr — Data recording session specified for SoC hardware board
soc.recorder object

Data recording session for specified SoC hardware board, specified as a soc.recorder object.

sourceName — Name of specified input source in data recording session
character vector

Name of specified input source in the data recording session, specified as a character vector.

Note You can specify sourceName as 'all' to remove all input sources added to a data recording session.

See Also
soc.iosource | soc.recorder

Introduced in R2019a

record

Record data from hardware using data recorder object

Syntax

```
record(dr,duration)
```

Description

`record(dr,duration)` records hardware input data on the SoC hardware board represented by `dr`, for the specified duration. `dr` is a data recording session on SoC hardware board created using `soc.recorder`.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the `hw` object. The resulting `soc.recorder` object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
           BlockingTime: 0
    OutputVarSizeSignal: false
             SampleTime: 0.1000
          HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

3-17

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

dr — Data recording session specified for SoC hardware board
soc.recorder object

Data recording session for specified SoC hardware board, specified as a soc.recorder object.

duration — Duration of recording session
positive scalar

Duration of recording session, specified as a positive scalar in seconds. Data is recorded on the hardware board for the specified duration of time. You can check the status of the data recording session by calling the isRecording object function.

Data Types: double

See Also

isRecording | setup | soc.recorder

Introduced in R2019a

isRecording

Get data recording status

Syntax

```
recordingStatus = isRecording(dr)
```

Description

`recordingStatus = isRecording(dr)` returns the status of the data recording process on the SoC hardware board represented by `dr`. `dr` is a data recording session on SoC hardware board created using `soc.recorder`.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the `hw` object. The resulting `soc.recorder` object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
           BlockingTime: 0
     OutputVarSizeSignal: false
             SampleTime: 0.1000
          HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

3-21

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

dr — Data recording session specified for SoC hardware board
`soc.recorder` object

Data recording session for specified SoC hardware board, specified as a `soc.recorder` object.

Output Arguments

recordingStatus — Status of data recording session
`false (0)` | `true (1)`

Status of data recording session, returned as logical value of of `false` (`0`) or `true` (`1`). This value is `1` when data recording is in progress and `0` when data recording is complete.

See Also
`record` | `soc.recorder`

Introduced in R2019a

save

Save recorded data from SoC hardware board to file on host PC

Syntax

```
save(dr,filename,description,tags)
```

Description

save(dr,filename,description,tags) saves the recorded data from the SoC
hardware board associated with soc.recorder object dr to a TGZ-compressed file,
filename, on the host PC.

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP
address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the hw object. The
resulting soc.recorder object represents the data recording session on the SoC
hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
               LocalPort: 25000
              DataLength: 1
                DataType: 'uint8'
       ReceiveBufferSize: -1
            BlockingTime: 0
     OutputVarSizeSignal: false
              SampleTime: 0.1000
           HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

3-25

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

Input Arguments

`dr` — Data recording session specified for SoC hardware board
`soc.recorder` object

Data recording session for specified SoC hardware board, specified as a `soc.recorder` object.

`filename` — File name
character vector

File name by which you want to save the recorded data from the SoC hardware board on your host PC, specified as a character vector.

`description` — Description added to file
character vector

Description added to file, specified as character vector. Add a description to the file helps you identify the data file when you read it using an `socFileReader` object. This input is optional.

Data Types: `char`

`tags` — Tags for data set
cell array

Tags for the data set, specified as cell array of character vectors. Adding tags to the file helps you identify the different input sources when you read the file using an `socFileReader` object. This input is optional.

See Also

`soc.recorder` | `socFileReader`

Introduced in R2019a

getServerError

Get latest error log messages from I/O Server running on specified SoC hardware board

Syntax

```
errMsg = getServerError(hw)
```

Description

`errMsg = getServerError(hw)` returns the latest error messages from the log of the I/O server running on the specified SoC hardware board, `hw`.

Examples

Get Latest Error Messages from I/O Server Log

Get latest error messages from log of IO Server running on specified SoC hardware board

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
dr = soc.recorder(hw);
src = soc.source(hw,'AXI Stream Read');
src.devName = 'mwramp_ge23nerator_ip0:s2mm0';
src.dataTypeStr = 'int32';
addSource(dr,src,'AXI4 stream interface');
setup(dr)

Error using matlab.io.linux.BaseHardware/recvResponse (line 374)
Recording Initialization failed for device AXI Stream Read.

Error in matlab.io.linux.BaseHardware/setupRecording (line 303)
            recvResponse(obj);

Error in matlab.io.linux.DataRecorder/setup (line 25)
            setupRecording(obj.HwObj);
```

Several errors occurred. Check the error log of the I/O server to get more information regarding the error.

```
errMsg = getServerError(hwObj)

errMsg =

    'Jan  1 00:48:39 buildroot-zynq user.err sociod: iioRecordSetup(): failed to find
     Jan  1 00:48:39 buildroot-zynq user.err sociod: recorderFunc(): Cannot initialize
```

Input Arguments

hw — Connection to specific SoC hardware board
zynq object | intelsoc object | mpsoc object

Connection to specific SoC hardware board, specified as an `socHardwareBoard` object.

Output Arguments

errMsg — Latest error messages from I/O server log
character array

Latest error messages from log of I/O server running on specified SoC hardware board, returned as a character array.

See Also
socHardwareBoard

Introduced in R2019a

getServerLog

Get log of I/O server running on specified SoC hardware board

Syntax

```
serverLog = getServerLog(hw)
```

Description

`serverLog = getServerLog(hw)` returns log messages from the I/O server running on specified SoC hardware board `hw`.

Examples

Get I/O Server Log

Get log messages from I/O Server running on the specified SoC hardware board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit' ,'hostname','192.168.1.18','us
serverLog = getServerLog(hwObj)

serverLog =

    'Jan  1 00:04:08 buildroot-zynq user.info sociod: ipDiscoveryThread(): IP discovery
     Jan  1 00:04:08 buildroot-zynq user.info sociod: ipDiscoveryThread(): UDP socket b
     Jan  1 00:04:09 buildroot-zynq user.debug sociod: acceptConnection(): Client conne
     Jan  1 00:04:09 buildroot-zynq user.info sociod: receiveRequest(): REQ = [2, 1, 39
     Jan  1 00:04:09 buildroot-zynq user.info sociod: receiveRequest(): REQ = [1, 2, 0]
     Jan  1 00:04:09 buildroot-zynq user.info sociod: eventHandlerThread(): RESP = [0,
     Jan  1 00:43:06 buildroot-zynq user.info sociod: eventHandlerThread(): Client disc
     Jan  1 00:43:16 buildroot-zynq user.debug sociod: acceptConnection(): Client conne
     Jan  1 00:43:16 buildroot-zynq user.info sociod: receiveRequest(): REQ = [2, 1, 39
     Jan  1 00:43:16 buildroot-zynq user.info sociod: receiveRequest(): REQ = [1, 2, 0]
```

```
Jan  1 00:43:16 buildroot-zynq user.info sociod: eventHandlerThread(): RESP = [0,
'
```

Input Arguments

hw — Connection to specific SoC hardware board
`zynq` object | `intelsoc` object | `mpsoc` object

Connection to specific SoC hardware board, specified as an `socHardwareBoard` object.

Output Arguments

serverLog — Server log
character array

Server log messages of the I/O server running on specified SoC hardware board, returned as a character array.

See Also
`socHardwareBoard`

Introduced in R2019a

socTaskTimes

Plot histogram of the task durations from a recorded SDI run

Syntax

```
taskData = socTaskTimes(modelName,runName)
taskData = socTaskTimes(___,suppressPlot)
```

Description

`taskData = socTaskTimes(modelName,runName)` creates an array of structures, one element for each task. Each structure contains the task name, task start times, task durations, and mean and standard deviations of the task durations. The function also plots the histogram of task durations for each task.

`taskData = socTaskTimes(___,suppressPlot)` to suppress the plot generated.

Input Arguments

modelName — Name of the Simulink model
string (default) | character array

Name of the Simulink model associated with run containing tasks.

Data Types: `char` | `string`

runName — Name of the SDI run
string (default) | character array

Name of Simulation Data Inspector run containing a task.

Data Types: `char` | `string`

suppressPlot — Name of the Simulink model
`"SuppressPlot"` (default)

Suppress the automatic generation of task duration plots.

Data Types: char | string

Output Arguments

taskData — Task timing data and statistics
structure

Task timing and duration statistics, returned as a structure with the fields:

See Also

"Task Visualization in Simulation Data Inspector" | "Recording Tasks for Use in Simulation"

Introduced in R2019a

soclib

Open the SoC Blockset block library

Syntax

```
soclib
```

Description

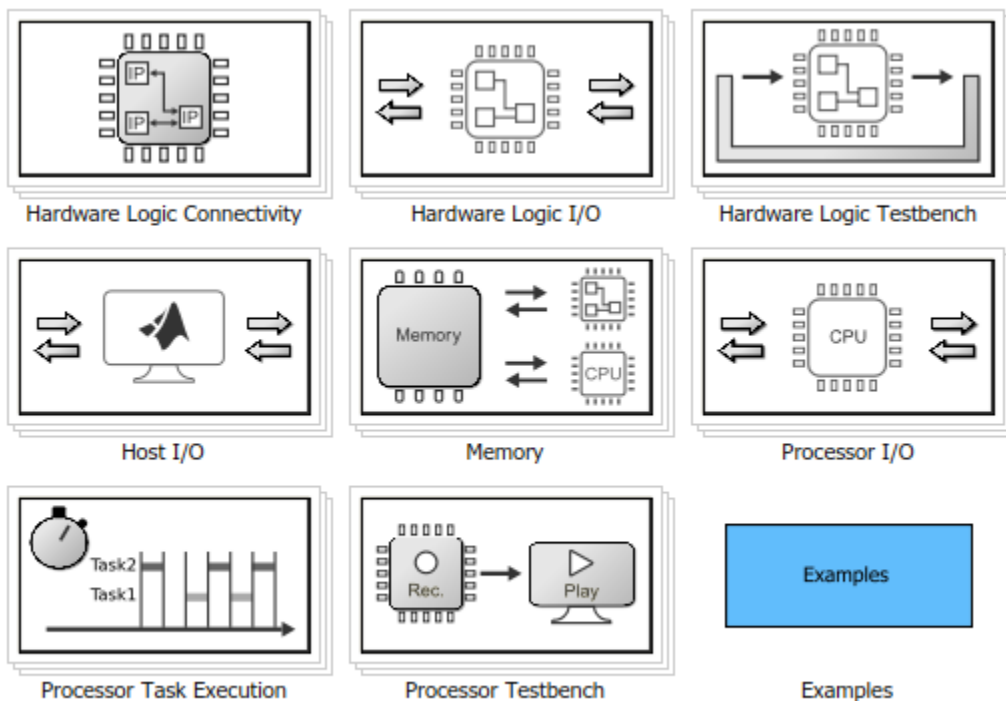`soclib` opens the SoC Blockset block library.

Examples

View the SoC Blockset Library

This example shows how to open and view the SoC Blockset library.

Run the following command to open the SoC Blockset library in Simulink:

```
soclib
```

Hardware Logic Connectivity

Hardware Logic I/O

Hardware Logic Testbench

Host I/O

Memory

Processor I/O

Processor Task Execution

Processor Testbench

Examples

See Also

"Getting Started with SoC Blockset" | "Library Browser" (Simulink)

Introduced in R2019a

collectMemoryStatistics

Retrieve performance data from AXI interconnect monitor

Syntax

```
collectMemoryStatistics(profiler)
```

Description

`collectMemoryStatistics(profiler)` retrieves performance data from the AXI interconnect monitor IP running on your hardware board. The `profiler` object represents a connection to that IP. When the AXI interconnect monitor is configured in `'Profile'` mode, call this function in a loop to retrieve average transaction latency and counts of bursts and bytes while transactions are occurring. In `'Trace'` mode, call this function once after memory transactions are complete to retrieve detailed memory transaction event data.

Examples

Configure and Query AXI Interconnect Monitor

The AXI interconnect monitor (AIM) is an IP core that collects performance metrics for an AXI-based FPGA design. Create an `socIPCore` object to setup and configure the AIM IP, and use the `socMemoryProfiler` object to retrieve and display the data.

For an example of how to configure and query the AIM IP in your design using MATLAB as AXI Master, see "Analyze Memory Bandwidth Using Traffic Generators". Specifically, review the `soc_memory_traffic_generator_axi_master.m` script that configures and monitors the design on the device.

The performance monitor can collect two types of data. Choose *Profile* mode to collect average transaction latency and counts of bytes and bursts. In this mode, you can launch a performance plot tool, and then configure the tool to plot bandwidth, burst count, and

transaction latency. Choose *Trace* mode to collect detailed memory transaction event data and view the data as waveforms.

```
Mode = 'Profile'; % or 'Trace'
```

To obtain diagnostic performance metrics from your generated FPGA design, you must set up a JTAG connection to the device from MATLAB. Load a `.mat` file that contains structures derived from the board configuration parameters. This file was generated by the **SoC Builder** tool. These structures describe the memory interconnect and masters configuration such as buffer sizes and addresses. Use the `socHardwareBoard` object to set up the JTAG connection.

```
load('soc_memory_traffic_generator_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Configure the AIM. The `socIPCore` object provides a function that performs this initialization. Then, create an `socMemoryProfiler` object to gather the metrics.

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',Mode);
initialize(apmCoreObj);
profilerObj = socMemoryProfiler(hwObj,apmCoreObj);
```

Retrieve performance metrics or signal data from a design running on the FPGA by using the `socMemoryProfiler` object functions.

For `'Profile'` mode, call the `collectMemoryStatistics` function in a loop.

```
NumRuns = 100;
for n = 1:NumRuns
    collectMemoryStatistics(profilerObj);
end
```

JTAG design setup time is long relative to FPGA transaction times, and if you have a small number of transactions in your design, they might have already completed by the time you query the monitor. In this case, the bandwidth plot shows only one sample, and the throughput calculation is not accurate. If this situation occurs, increase the total number of transactions the design executes.

For `'Trace'` mode, call the `collectMemoryStatistics` function once. This function stops the IP from writing transactions into the FIFO in the AXI interconnect monitor IP, although the transactions continue on the interconnect. Set the size of the transaction FIFO, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation > Target hardware resources > FPGA design (debug)**.

```
collectMemoryStatistics(profilerObj);
```

Visualize the performance data by using the `plotMemoryStatistics` function. In `'Profile'` mode, this function launches a performance plot tool, and you can configure the tool to plot bandwidth, burst count, and average transaction latency. In `'Trace'` mode, this function opens the **Logic Analyzer** tool to view burst transaction event data.

```
plotMemoryStatistics(profilerObj);
```

Input Arguments

`profiler` — **Memory profiler object**
`socMemoryProfiler` object

Memory profiler object, specified as an `socMemoryProfiler` object that provides access to the AXI memory interconnect IP running on the hardware board.

See Also
"Memory Performance Information from FPGA Execution"

Topics
"Analyze Memory Bandwidth Using Traffic Generators"

Introduced in R2019a

plotMemoryStatistics

Plot performance data obtained from AXI interconnect monitor

Syntax

```
plotMemoryStatistics(profiler)
```

Description

plotMemoryStatistics(profiler) generates visualizations of the performance data from the AXI interconnect monitor IP running on your hardware board. The profiler object represents a connection to that IP. When the AXI interconnect monitor is configured in 'Profile' mode, this function launches a performance plot tool. You can configure the tool to plot bandwidth, burst count, and average transaction latency. In 'Trace' mode, this function opens the **Logic Analyzer** to view detailed memory transaction event data.

Examples

Configure and Query AXI Interconnect Monitor

The AXI interconnect monitor (AIM) is an IP core that collects performance metrics for an AXI-based FPGA design. Create an socIPCore object to setup and configure the AIM IP, and use the socMemoryProfiler object to retrieve and display the data.

For an example of how to configure and query the AIM IP in your design using MATLAB as AXI Master, see "Analyze Memory Bandwidth Using Traffic Generators". Specifically, review the soc_memory_traffic_generator_axi_master.m script that configures and monitors the design on the device.

The performance monitor can collect two types of data. Choose *Profile* mode to collect average transaction latency and counts of bytes and bursts. In this mode, you can launch a performance plot tool, and then configure the tool to plot bandwidth, burst count, and

transaction latency. Choose *Trace* mode to collect detailed memory transaction event data and view the data as waveforms.

```
Mode = 'Profile'; % or 'Trace'
```

To obtain diagnostic performance metrics from your generated FPGA design, you must set up a JTAG connection to the device from MATLAB. Load a .mat file that contains structures derived from the board configuration parameters. This file was generated by the **SoC Builder** tool. These structures describe the memory interconnect and masters configuration such as buffer sizes and addresses. Use the socHardwareBoard object to set up the JTAG connection.

```
load('soc_memory_traffic_generator_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Configure the AIM. The socIPCore object provides a function that performs this initialization. Then, create an socMemoryProfiler object to gather the metrics.

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',Mode);
initialize(apmCoreObj);
profilerObj = socMemoryProfiler(hwObj,apmCoreObj);
```

Retrieve performance metrics or signal data from a design running on the FPGA by using the socMemoryProfiler object functions.

For 'Profile' mode, call the collectMemoryStatistics function in a loop.

```
NumRuns = 100;
for n = 1:NumRuns
    collectMemoryStatistics(profilerObj);
end
```

JTAG design setup time is long relative to FPGA transaction times, and if you have a small number of transactions in your design, they might have already completed by the time you query the monitor. In this case, the bandwidth plot shows only one sample, and the throughput calculation is not accurate. If this situation occurs, increase the total number of transactions the design executes.

For 'Trace' mode, call the collectMemoryStatistics function once. This function stops the IP from writing transactions into the FIFO in the AXI interconnect monitor IP, although the transactions continue on the interconnect. Set the size of the transaction FIFO, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation > Target hardware resources > FPGA design (debug)**.

```
collectMemoryStatistics(profilerObj);
```

Visualize the performance data by using the `plotMemoryStatistics` function. In `'Profile'` mode, this function launches a performance plot tool, and you can configure the tool to plot bandwidth, burst count, and average transaction latency. In `'Trace'` mode, this function opens the **Logic Analyzer** tool to view burst transaction event data.

```
plotMemoryStatistics(profilerObj);
```

Input Arguments

profiler — Memory profiler object
socMemoryProfiler object

Memory profiler object, specified as an `socMemoryProfiler` object that provides access to the AXI memory interconnect IP running on the hardware board.

See Also

"Memory Performance Information from FPGA Execution"

Topics

"Analyze Memory Bandwidth Using Traffic Generators"

Introduced in R2019a

initialize

Initialize IP core corresponding to `socIPCore` object

Syntax

```
initialize(socIP)
```

Description

`initialize(socIP)` initializes the IP core corresponding to `socIP`, an `socIPCore` object.

Examples

Initialize Traffic Generator IP

Create an `socIPCore` object representing a traffic generator IP on an FPGA board. Then initialize it using the `initialize` function.

```
% Create IPCore object for traffic generator IP
trafficGeneratorObj = socIPCore(AXIMasterObj, atg,'TrafficGenerator');
% Initialize traffic generator IP
initialize(trafficGeneratorObj);
```

Input Arguments

socIP — Connection to IP core running on FPGA board
`socIPCore`

Connection to IP core running on FPGA board, specified as an `socIPCore` object.

See Also

socIPCore

Introduced in R2019a

start

Start IP core execution on hardware board

Syntax

```
start(socIP)
```

Description

`start(socIP)` starts the execution of the IP core represented by the `socIP` object.

This function is only applicable when `socIPCore` is an object representing `TrafficGenerator` or `VDMATrigger`.

Examples

Initialize and Start a Traffic Generator IP

Create an `socIPCore` object representing a traffic generator IP on an FPGA board. Then initialize the traffic generator using the `initialize` function.

```
% Create IPCore object for traffic generator IP
trafficGeneratorObj = socIPCore(AXIMasterObj, atg,'TrafficGenerator');
% Initialize traffic generator IP
initialize(trafficGeneratorObj);
```

Start the traffic generator IP execution on your FPGA board.

```
start(trafficGeneratorObj);
```

Input Arguments

socIP — Connection to IP core running on FPGA board
socIPCore

Connection to IP core running on FPGA board, specified as an socIPCore object.

See Also
socIPCore

Introduced in R2019a

readmemory

Read data from AXI4 memory-mapped slaves

Syntax

```
data = readmemory(mem,addr,size)
data = readmemory(mem,addr,size,Name,Value)
```

Description

`data = readmemory(mem,addr,size)` reads `size` locations of data, starting from the address specified in `addr`, and incrementing the address for each word. By default, the output data type is `uint32`. `addr`, must refer to an AXI slave memory location controlled by the AXI master IP on your hardware board. The `socAXIMaster` object, `mem`, manages the connection between MATLAB and the AXI master IP.

`data = readmemory(mem,addr,size,Name,Value)` reads `size` locations of data, starting from the address specified in `addr`, with additional options specified by one or more `Name,Value` pair arguments.

Examples

Access Memory on SoC Hardware Board from MATLAB

For this example, you must have a design running on a hardware board connected to the MATLAB host machine.

Create a MATLAB AXI master object. The object connects with the hardware board and confirms that the IP is present. You can create the object with a vendor name or an `socHardwareBoard` object.

```
mem = socAXIMaster('Xilinx');
```

Write and read one or more addresses with one command. By default, the functions auto-increment the address for each word of data. For instance, write ten addresses, then read the data back from a single location.

```
writememory(mem,140,[10:19])
rd_d = readmemory(mem,140,1)

rd_d =

  uint32

    10
```

Now, read the written data from ten locations.

```
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

   10   11   12   13   14   15   16   17   18   19
```

Set the BurstType property to 'Fixed' to turn off the auto-increment and access the same address multiple times. For instance, read the written data ten times from the same address.

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')

rd_d =

  1×10 uint32 row vector

   10   10   10   10   10   10   10   10   10   10
```

Write incrementing data ten times to the same address. The final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed')
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

   29   11   12   13   14   15   16   17   18   19
```

Alternatively, specify the address as a hexadecimal string. To cast the read data to a data type other than `uint32`, use the `OutputDataType` property.

```
writememory(mem,'1c',[0:4:64])
rd_d = readmemory(mem,'1c',16,'OutputDataType',numerictype(0,6,4))

rd_d =

  Columns 1 through 10
        0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.0
  Columns 11 through 16
   2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Unsigned
           WordLength: 6
        FractionLength: 4
```

When you are done accessing the board, release the JTAG connection.

```
release(mem)
```

Input Arguments

`mem` — JTAG connection to AXI master IP running on hardware board
`socAXIMaster` object

JTAG connection to AXI master IP running on your hardware board, specified as an `socAXIMaster` object.

`addr` — Starting address for read operation
integer | hexadecimal character vector

Starting address for read operation, specified as an integer or a hexadecimal character vector. The function casts the address to `uint32` data type. The address must refer to an AXI slave memory location controlled by the AXI master IP on your hardware board.

Example: `'a4'`

`size` — Number of locations to read
integer

Number of memory locations to read, specified as an integer. By default, the function reads from a contiguous address block, incrementing the address for each operation. To

turn off the address increment and read repeatedly from the same location, set the `BurstType` property to `'Fixed'`.

When you specify a large operation size, such as reading a block of DDR memory, the object automatically breaks the operation into multiple bursts, using the maximum supported burst size. The maximum supported burst size is 256 words.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `readmemory(mem,140,10,'BurstType','Fixed')`

BurstType — AXI4 burst type
`'Increment'` (default) | `'Fixed'`

AXI4 burst type, specified as the comma-separated pair consisting of `'BurstType'` and either `'Increment'` or `'Fixed'`. If this value is `'Increment'`, the AXI master reads a vector of data from contiguous memory locations, starting with the specified address. If this value is `'Fixed'`, the AXI master reads all data from the same address.

OutputDataType — Data type assigned to read data
`'uint32'` (default) | `'int8'` | `'int16'` | `'int32'` | `'uint8'` | `'uint16'` | `'single'` | `numerictype` object

Data type assigned to the read data, specified as `'uint32'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'single'`, or a `numerictype` object.

Output Arguments

data — Read data
scalar | vector

Read data, returned as scalar or vector depending on the value you specified for `size`. The function casts the data to the data type specified by the `OutputDataType` property.

See Also

writememory

Introduced in R2019a

writememory

Write data to AXI4 memory-mapped slaves

Syntax

```
writememory(mem,addr,data)
writememory(mem,addr,data,Name,Value)
```

Description

writememory(mem,addr,data) writes all words specified in data, starting from the address specified in addr, and then incrementing the address for each word. addr, must refer to an AXI slave memory location controlled by the AXI master IP on your hardware board. The socAXIMaster object, mem, manages the connection between MATLAB and the AXI master IP.

writememory(mem,addr,data,Name,Value) writes all words specified in data, starting from the address specified in addr, with additional options specified by one or more Name,Value pair arguments.

Examples

Access Memory on SoC Hardware Board from MATLAB

For this example, you must have a design running on a hardware board connected to the MATLAB host machine.

Create a MATLAB AXI master object. The object connects with the hardware board and confirms that the IP is present. You can create the object with a vendor name or an socHardwareBoard object.

```
mem = socAXIMaster('Xilinx');
```

Write and read one or more addresses with one command. By default, the functions auto-increment the address for each word of data. For instance, write ten addresses, then read the data back from a single location.

```
writememory(mem,140,[10:19])
rd_d = readmemory(mem,140,1)

rd_d =

  uint32

   10
```

Now, read the written data from ten locations.

```
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

   10   11   12   13   14   15   16   17   18   19
```

Set the BurstType property to 'Fixed' to turn off the auto-increment and access the same address multiple times. For instance, read the written data ten times from the same address.

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')

rd_d =

  1×10 uint32 row vector

   10   10   10   10   10   10   10   10   10   10
```

Write incrementing data ten times to the same address. The final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed')
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

   29   11   12   13   14   15   16   17   18   19
```

Alternatively, specify the address as a hexadecimal string. To cast the read data to a data type other than `uint32`, use the `OutputDataType` property.

```
writememory(mem,'1c',[0:4:64])
rd_d = readmemory(mem,'1c',16,'OutputDataType',numerictype(0,6,4))

rd_d =

  Columns 1 through 10
        0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.(
  Columns 11 through 16
   2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 6
         FractionLength: 4
```

When you are done accessing the board, release the JTAG connection.

```
release(mem)
```

Input Arguments

`mem` — JTAG connection to AXI master IP running on hardware board
`socAXIMaster` object

JTAG connection to AXI master IP running on your hardware board, specified as an `socAXIMaster` object.

`addr` — Starting address for write operation
integer | hexadecimal character vector

Starting address for read operation, specified as an integer or a hexadecimal character vector. The function casts the address to `uint32` data type. The address must refer to an AXI slave memory location controlled by the AXI master IP on your hardware board.

Example: `'a4'`

`data` — Data words to write
scalar | vector

Data words to write, specified as a scalar or a vector. By default, the function writes the data to a contiguous address block, incrementing the address for each operation. To turn

off the address increment and write each data value to the same location, set the BurstType property to 'Fixed'.

Before sending the write request to the board, the function casts the input data to uint32 or int32 data type. The data type conversion follows these rules:

- If the input data type is double, then the data is cast to int32 data type.
- If the input data type is single, then the data is cast to uint32 data type.
- If the bit width of the input data type is less than 32 bits, then the data is sign-extended to 32 bits.
- If the bit width of the input data type is longer than 32 bits, then the data is cast to int32 or uint32 data type, matching the signedness of the original data type.
- If the input data is a fixed-point data type, then the function writes the stored integer value of the data.

When you specify a large operation size, such as writing a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size. The maximum supported burst size is 256 words.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: writememory(mem,140,[20:29],'BurstType','Fixed')

BurstType — AXI4 burst type
'Increment' (default) | 'Fixed'

AXI4 burst type, specified as the comma-separated pair consisting of 'BurstType' and either 'Increment' or 'Fixed'. If this value is 'Increment', the AXI master writes a vector of data from contiguous memory locations, starting with the specified address. If this value is 'Fixed', the AXI master writers all data from the same address.

See Also

readmemory

Introduced in R2019a

release

Release JTAG cable resource

Syntax

```
release(mem)
```

Description

`release(mem)` releases the JTAG cable resource, freeing the cable for use to reprogram the FPGA. After initialization, the AXI master object, `mem`, holds the JTAG cable resource, and other programs cannot access that JTAG cable. When you have an active AXI master object, FPGA programming over JTAG fails. Call the `release` object function before reprogramming the FPGA.

Examples

Access Memory on SoC Hardware Board from MATLAB

For this example, you must have a design running on a hardware board connected to the MATLAB host machine.

Create a MATLAB AXI master object. The object connects with the hardware board and confirms that the IP is present. You can create the object with a vendor name or an `socHardwareBoard` object.

```
mem = socAXIMaster('Xilinx');
```

Write and read one or more addresses with one command. By default, the functions auto-increment the address for each word of data. For instance, write ten addresses, then read the data back from a single location.

```
writememory(mem,140,[10:19])
rd_d = readmemory(mem,140,1)
```

```
rd_d =

  uint32

    10
```

Now, read the written data from ten locations.

```
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

    10    11    12    13    14    15    16    17    18    19
```

Set the `BurstType` property to `'Fixed'` to turn off the auto-increment and access the same address multiple times. For instance, read the written data ten times from the same address.

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')

rd_d =

  1×10 uint32 row vector

    10    10    10    10    10    10    10    10    10    10
```

Write incrementing data ten times to the same address. The final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed')
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

    29    11    12    13    14    15    16    17    18    19
```

Alternatively, specify the address as a hexadecimal string. To cast the read data to a data type other than `uint32`, use the `OutputDataType` property.

```
writememory(mem,'1c',[0:4:64])
rd_d = readmemory(mem,'1c',16,'OutputDataType',numerictype(0,6,4))
```

3-57

```
rd_d =

  Columns 1 through 10
        0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.0
  Columns 11 through 16
    2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

         DataTypeMode: Fixed-point: binary point scaling
          Signedness: Unsigned
          WordLength: 6
       FractionLength: 4
```

When you are done accessing the board, release the JTAG connection.

```
release(mem)
```

Input Arguments

mem — JTAG connection to AXI master IP running on hardware board
socAXIMaster object

JTAG connection to AXI master IP running on your hardware board, specified as an
socAXIMaster object.

See Also
readmemory | writememory

Introduced in R2019a

Objects

CodeInstrumentationTaskProfiler

Manage connection to code instrumentation task profilers

Description

The `CodeInstrumentationTaskProfiler` object manages the connection between the code instrumentation profiler on a deployed Simulink model executing on a hardware processor core and the development computer.

The code instrumentation profiler injects a time measurement function at the start and end the generated task code. These start and end times get recorded and sent to the development computer. Additional processing in MATLAB interprets the execution times and task priorities to estimate the execution behavior of the threads on the processor core.

Creation

Create a code instrumentation task profiler object using `streamingTaskProfiler`.

Properties

`Started` — Status of streaming
0 (default) | 1

A flag indicating if the `CodeInstrumentationTaskProfiler` is running and receiving data from the hardware board.

Data Types: `logical`

Object Functions

See Also

Introduced in R2018a

KernelTaskProfiler

Manage connection to kernel task profilers

Description

The `KernelTaskProfiler` object manages the connection between the kernel profiler on a deployed Simulink model executing on a hardware processor core and the development computer.

The kernel streaming task profiler object directly measures and records thread execution and processor core activity on a deployed Simulink model. The kernel profiler uses the open-source profiler, LTTng, operating in the background to make the measurements of thread states, interrupts, and OS kernel operations. The data recorded by the kernel profiler gets streamed to the development computer over an XCP network connection.

Creation

Create a kernel task profiler object using `streamingTaskProfiler` and deployed model.

Properties

KernelProfilerObj — Kernel profiler object
soc.profiler.KernelTaskProfiler (default)

The internal kernel task profiler object containing the information and connection properties of the kernel profiler and connection.

Object Functions

See Also

Introduced in R2018a

soc.iosource

Input source on SoC hardware board

Description

Create an `soc.iosource` object to connect to an input source on an SoC hardware board. Pass the `soc.iosource` object as an argument to the `addSource` function of the `soc.recorder` object.

The sources available on the design running on the SoC hardware board correspond to the blocks you included in your Simulink model. When you run **SoC Builder**, it connects your FPGA logic with the matching interface on the board.

Source	Block	Action
`'TCP Receive'`	TCP Read	Read UDP (User Datagram Protocol) data from the Linux socket buffer.
`'UDP Receive'`	UDP Read	Read TCP/IP data from Linux socket buffer.
`'AXI Register Read'`	Register Read	Read registers from an IP core using the AXI interface.
`'AXI Stream Read'`	Stream Read	Read AXI-4 Stream data using IIO.

Creation

Syntax

```
availableSources = soc.iosource(hw)
src = soc.iosource(hw,inputSourceName)
```

Description

`availableSources = soc.iosource(hw)` returns a list of input sources available for data logging on the SoC hardware board connected through `hw`. `hw` is an `socHardwareBoard` object.

`src = soc.iosource(hw,inputSourceName)` creates a source object corresponding to `inputSourceName` on the SoC hardware board connected through `hw`.

Input Arguments

hw — Hardware object
`socHardwareBoard` object

Hardware object, specified as a `socHardwareBoard` object that represents the connection to the SoC hardware board.

`inputSourceName` — Name of available input source on SoC hardware board
character vector

Name of an available input source on the SoC hardware board, specified as a character vector. To get the list of input sources available for data logging on the specified SoC hardware board, call the `soc.iosource` function without arguments.

Example: 'UDP Receive'

Data Types: `char`

Output Arguments

availableSources — List of input data sources available for data logging
cell array

List of input data sources available for data logging on the specified SoC hardware board, returned as a cell array. Each cell contains a character vector with the name of an available input data source for data logging on the specified SoC hardware board. Use one of these names as the `inputSourceName` argument when you create a source object.

src — Source object for specified input source
`soc.iosource` object

Source object for specified input source, returned as an `soc.iosource`.

Properties

`DeviceName` — Name of IP core device
character vector

Name of IP core device, specified as a character vector.

Example: `'mwipcore0:s2mm0'`

Dependencies

To enable this property, create a AXI register or AXI stream source object.

Data Types: `char`

`RegisterOffset` — Offset from base address of IP core to register
positive scalar

Offset from the base address of the IP core to the register, specified as a positive scalar.

Dependencies

To enable this property, create a AXI register source object.

Data Types: `uint32`

`LocalPort` — IP port on hardware board where UDP or TCP data is received
25000 (UDP) (default) | `-1` (TCP) | integer from 1 to 65,535

IP port on hardware board where UDP or TCP data is received specified as a scalar from 1 to 65,535. The object reads UDP or TCP data received on this port of the specified SoC hardware board.

For a TCP object with the `NetworkRole` property to `'Client'`, set `LocalPort` to `-1` to assign any random available port on the hardware board as the local port.

Dependencies

To enable this property, create a TCP or UDP source object.

Data Types: `uint16`

`NetworkRole` — Network role
`'Client'` (default) | character vector

Network role, specified as a character vector.

Example: 'Client'

Dependencies

To enable this property, create a TCP source object.

Data Types: `enumerated string`

`RemoteAddress` — IP address of remote server from which data is received
`'127.0.0.1'` (default) | dotted-quad expression

IP address of the remote server from which data is received, specified as a dotted-quad expression.

Dependencies

To enable this property, create a TCP source object.

Data Types: `char`

`RemotePort` — IP port number of remote server from which data is received
25000 (default) | integer from 1 to 65,535

IP port number of the remote server from which data is received, specified as an integer from 1 to 65,535.

Dependencies

To enable this property, create a TCP source object.

Data Types: `double`

`DataLength` — Length of data packet or register data vector
1 (default) | positive scalar

Maximum length of UDP or TCP data packet, or word length of AXI register data vector, specified as a positive scalar.

Data Types: `double`

`SamplesPerFrame` — Size of data vector read from IP core
nonnegative scalar

Size of the data vector read from the IP core, specified as a nonnegative scalar.

Dependencies

To enable this property, create a AXI stream source object.

Data Types: `double`

DataType — Data type of received data
`'uint8'` (default) | `'uint16'` | `'uint32'` | `'int8'` | `'int16'` | `'int32'` | `'double'` | `'single'`

Data type of received data, specified as `'uint8'`, `'uint16'`, `'uint32'`, `'int8'`, `'int16'`, `'int32'`, `'double'` or `'single'`.

Data Types: `char`

ReceiveBufferSize — Internal buffer size of object
65535 (default) | array

Internal buffer size of object, specified as an array.

Dependencies

To enable this property, create a TCP or UDP source object.

Data Types: `double`

Sample Time — Sample time
1 (default) | nonnegative scalar

Sample time, in seconds, at which you want to receive data, specified as an nonnegative scalar.

Data Types: `double`

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the hw object. The
resulting soc.recorder object represents the data recording session on the SoC
hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

         HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
              Sources: {}
            Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, soc.recorder objects have no added input sources. To add an input source
to the data recording session, first create an input source object by using the
soc.iosource function. For this example, create an User Datagram Protocol (UDP)
source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
              LocalPort: 25000
             DataLength: 1
               DataType: 'uint8'
      ReceiveBufferSize: -1
            BlockingTime: 0
     OutputVarSizeSignal: false
              SampleTime: 0.1000
          HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

See Also
soc.recorder | socHardwareBoard

Introduced in R2019a

socHardwareBoard

Connection to SoC hardware board

Description

The `socHardwareBoard` object represents a connection to the specified SoC hardware board from MATLAB. Use this object to create `soc.recorder` and `socAXIMaster` objects that record input data and access memory on the specified SoC hardware board.

Creation

Syntax

```
hwList = socHardwareBoard()
hw = socHardwareBoard(BoardName)
hw = socHardwareBoard(boardName,Name,Value)
```

Description

`hwList = socHardwareBoard()` returns a list of supported SoC hardware boards.

`hw = socHardwareBoard(BoardName)` creates a connection to the specified SoC hardware board. This connection reuses the IP address, username, and password from the most recent connection to that specified SoC hardware board. When you connect MATLAB to an SoC hardware board for the first time, enter the board name, IP address, username, and password of the SoC hardware board as name-value pair arguments.

To see the complete list of supported SoC hardware boards, call the `socHardwareBoard` function without any arguments.

`hw = socHardwareBoard(boardName,Name,Value)` creates a connection to the specified SoC hardware by using the IP address, user name, and password that you specify.

Input Arguments

boardName — Name of supported SoC hardware board
character vector | string scalar

Name of supported SoC hardware board, specified as a character vector or string scalar. Specify the name of hardware board to which you want to establish a connection from MATLAB. To get the list of supported hardware boards, call `socHardwareBoard` function without any arguments.

Example: `'Xilinx Zynq ZC706 evaluation kit'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'username','root'`

hostname — IP address of SoC hardware board
character vector | string scalar

IP address of the SoC hardware board connected to the network, specified as the comma-separated pair consisting of `'hostname'` and a character vector or string scalar.

Example: '192.168.1.18'

Data Types: `char` | `string`

username — Root username used to log into SoC hardware board
character vector | string scalar

Root username used to log in into SoC hardware board connected to the network, specified as the comma-separated pair consisting of `'username'` and a character vector or string scalar.

Example: 'root'

Data Types: `char` | `string`

password — Root password used to log into SoC hardware board
character vector | string scalar

Root password used to log in into SoC hardware board connected to the network, specified as the comma-separated pair consisting of `'password'` and a character vector or string scalar.

Example: 'password'

Data Types: `char` | `string`

Output Arguments

`hwList` — List of supported SoC hardware boards
string array

List of SoC hardware boards that are supported for data logging returned as a string array.

`hw` — Connection to specific SoC hardware board
`socHardwareBoard` object

Connection to specific SoC hardware board, returned as a `socHardwareBoard` object. You can use this connection for data logging of input sources with the `soc.recorder` object, or you can access memory on the board using an `socAXIMaster` object.

Properties

`BoardName` — Name of supported SoC hardware board
character array | string scalar

This property cannot be changed after you create the `socHardwareBoard` object.

Name of supported SoC hardware board, specified as a character array or string scalar.

Example: `'Xilinx Zynq ZC706 evaluation kit'`

Data Types: `char` | `string`

`DeviceAddress` — IP address of SoC hardware board
character array | string scalar

This property cannot be changed after you create the `socHardwareBoard` object.

IP address of SoC hardware board, specified as a character array or string scalar.

Example: '192.168.1.11'

Data Types: `char` | `string`

Port — IP port number of SoC hardware board
integer from 1 to 65,535

This property cannot be changed.

IP port number of SoC hardware board.

Example: `18735`

Data Types: `double`

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the `hw` object. The resulting `soc.recorder` object represents the data recording session on the SoC hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

        HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
             Sources: {}
           Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)
```

```
ans =

  1×0 empty cell array
```

By default, `soc.recorder` objects have no added input sources. To add an input source to the data recording session, first create an input source object by using the `soc.iosource` function. For this example, create an User Datagram Protocol (UDP) source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
               LocalPort: 25000
              DataLength: 1
                DataType: 'uint8'
       ReceiveBufferSize: -1
            BlockingTime: 0
     OutputVarSizeSignal: false
              SampleTime: 0.1000
           HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the isRecording object function. The recording status when data recording is in progress is 1.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

    1
```

The recording status when data recording is complete is 0.

```
isRecording(dr)

recordingStatus =

  logical

    0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file UDPDataReceived.tgz in your working folder of the host PC. You can read this file by using an socFileReader object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the removeSource object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the Sources property of the soc.recorder object.

```
ans =

  1×0 empty cell array
```

Initialize Memory on SoC Hardware Board from MATLAB

For an example of how to configure and use the AXI master IP in your design, see "Random Access of External Memory". Specifically, review the `soc_image_rotation_axi_master.m` script that initializes the memory on the device, starts the FPGA logic, and reads back the modified data. This example shows only the memory initialization step.

Load a `.mat` file that contains structures derived from the board configuration parameters. This file was generated by **SoC Builder**. These structures also describe the IP cores and memory configuration of the design on the board. Set up a JTAG AXI master connection by creating a `socHardwareBoard` and passing it to the `socAXIMaster` object. The `socAXIMaster` object connects with the hardware board and confirms that the IP is present.

```
load('soc_image_rotation_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Initialize the memory contents on the device by loading the figure data and writing it to `Region1`. The FPGA logic is designed to read this data, rotate it, and write it into `Region2`. Clear the contents of `Region2`.

```
load('soc_image_rotation_inputdata.mat');
inputFigure = smallImage;
[x, y] = size(inputFigure);
inputImage = uint32(reshape(inputFigure',1,x*y));
writememory(AXIMasterObj,memRegions.AXI4MasterMemRegion1,inputImage);
writememory(AXIMasterObj,memRegions.AXI4MasterMemRegion2,uint32(zeros(1,x*y)));
```

See Also
`soc.iosource` | `soc.recorder` | `socAXIMaster`

Introduced in R2019a

soc.recorder

Data recording session for specified SoC hardware board

Description

An `soc.recorder` object can configure and log data from input sources on an SoC hardware board connected to MATLAB. You can save the recorded data to a file for future use to playback in MATLAB and Simulink models.

Creation

Syntax

```
dr = soc.recorder(hw)
```

Description

`dr = soc.recorder(hw)` creates a data recording session, `dr`, on the SoC hardware board connection represented by `hw`. `hw` is an `socHardwareBoard` object.

Input Arguments

hw — Hardware object
`socHardwareBoard` object

Hardware object, specified as a `socHardwareBoard` object that represents the connection to the SoC hardware board.

Properties

HardwareName — Name of supported SoC hardware board
character vector

Name of supported SoC hardware board, specified as a character vector.

Data Types: `char`

Sources — List of hardware-peripheral input sources
cell array

List of hardware-peripheral input sources added to data recording session, specified a character vector. To add input sources to a `soc.recorder` object, call the `addSource` object function.

Data Types: `cell`

Recording — Status of data recording session
`false` (0) | `true` (1)

This property is read-only.

Status of data recording session, specified as a logic value of `false` (0) or `true` (1). To get the status of the data recording session, call the `isRecording` object function.

Data Types: `logical`

Object Functions

addSource	Add a input source to a data recording session
removeSource	Remove input source from data recording session
setup	Set up hardware for data recording
record	Record data from hardware using data recorder object
isRecording	Get data recording status
save	Save recorded data from SoC hardware board to file on host PC

Examples

Record Data From SoC Hardware Board

Create a connection from MATLAB to the specified SoC hardware board using the IP address, username, and password of the board.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','192.168.1.18','use
```

Create a data recording session on the SoC hardware board by using the hw object. The
resulting soc.recorder object represents the data recording session on the SoC
hardware board.

```
dr = soc.recorder(hw)

dr =

  DataRecorder with properties:

          HardwareName: 'Xilinx Zynq ZC706 evaluation kit'
               Sources: {}
             Recording: false
```

List the input sources added to the data recording session.

```
dr.Sources(hw)

ans =

  1×0 empty cell array
```

By default, soc.recorder objects have no added input sources. To add an input source
to the data recording session, first create an input source object by using the
soc.iosource function. For this example, create an User Datagram Protocol (UDP)
source object.

```
udpSrc = soc.iosource(hw,'UDP Receive')

udpSrc =

  soc.iosource.UDPRead with properties:

   Main
               LocalPort: 25000
              DataLength: 1
                DataType: 'uint8'
       ReceiveBufferSize: -1
             BlockingTime: 0
      OutputVarSizeSignal: false
              SampleTime: 0.1000
           HideEventLines: true

  Show all properties
```

Add this UDP source object to the data recording session by using the `addSource` object function.

```
addSource(dr,udpSrc,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
dr.Sources

ans =

  1×1 cell array

    {'UDPDataOnPort25000'}
```

Call the setup function to initialize all hardware peripheral input sources added to the data recording session, and start the data recording process.

```
setup(dr)
```

Record data for 60 seconds on the SoC hardware board.

```
record(dr, 60);
```

Check the status of the data recording session by using the `isRecording` object function. The recording status when data recording is in progress is `1`.

```
recordingStatus = isRecording(dr)

recordingStatus =

  logical

   1
```

The recording status when data recording is complete is `0`.

```
isRecording(dr)

recordingStatus =

  logical

   0
```

Save recorded data to a TGZ-compressed file.

```
save(dr,'UDPDataReceived','UDP Data Testing',{'Recorded On Zynq Board'})
```

This function saves the recorded data as the file `UDPDataReceived.tgz` in your working folder of the host PC. You can read this file by using an `socFileReader` object in MATLAB or an IO Data Source block in your Simulink model.

Remove the added source from the data recording session by using the `removeSource` object function.

```
removeSource(dr,'UDPDataReceived-Port25000')
```

Verify the result by inspecting the `Sources` property of the `soc.recorder` object.

```
ans =

  1×0 empty cell array
```

See Also
soc.iosource | socFileReader | socHardwareBoard

Introduced in R2019a

socFileReader

File reader

Description

The `socFileReader` object is a file reader that reads data from a specified TGZ-compressed file and stores the data sets in the object. The data set contains information about the source objects that represent recorded data sources from the specified TGZ-compressed file. The TGZ file format is created by a previous recording session on an SoC hardware board.

Creation

Syntax

```
fr = socFileReader(filename)
```

Description

`fr = socFileReader(filename)` creates an object, `fr`, from the specified file. The object is a file reader that reads data from a specified TGZ-compressed file and stores the data sets in the object. The `filename` must be a file saved using the `save` function of an `soc.recorder` object.

Input Arguments

`filename` — File from previous data recording session
character vector

File from a previous data recording session on SoC hardware board, specified as a character vector with a `tgz` extension.

Example: 'UDPDataReceived.tgz'

Properties

`Description` — User metadata describing data set
character vector

User meta data describing the data set, specified as a character vector. This value is added to the file when you call the `save` object function.

Data Types: `char`

`HardwareBoard` — Name of SoC hardware board
character vector

Name of the SoC hardware board used for data collection in the `soc.recorder` object, specified as a character vector.

Data Types: `char`

`Tags` — User tags
cell array

User tags, specified as a cell array. This value is added to the file when you call the `save` object function.

Data Types: `cell`

`Filename` — Name of recorded data file
character vector

Name of recorded data file, specified as a character vector. This value represents the file name of a file saved using the `save` object function.

Data Types: `char`

`Sources` — List of sources in data set
cell array

List of sources in data set file, returned as a cell array.

Data Types: `cell`

`Date` — Date of data set creation
character vector

Date of data set creation, returned as a character vector.

Data Types: `char` | `string`

Object Functions

getData   Get data from file reader

Examples

Create File Reader Object

Create a file reader to read data from the specified TGZ-compressed file.

```
fr = socFileReader('UDPDataReceived.tgz')

fr =

  socFileReader with properties:

      Description: ''
    HardwareBoard: 'Xilinx Zynq ZC706 evaluation kit'
             Tags: {}
         Filename: 'H:\UDPDataReceived.tgz'
          Sources: {'UDPDataOnPort25000'}
             Date: 28-Dec-2018 15:17:08
```

Get the data of a specified source from the file using the `getData` function.

```
rd = getData(fr,'UDPDataReceived-Port25000');
```

See Also

`save` | `soc.recorder`

Introduced in R2019a

socIPCore

Create object to represent IP core running on FPGA board

Description

The `socIPCore` object represents an active IP core on an FPGA board and provides read and write access to the IP.

Creation

Syntax

```
myCoreObj = socIPCore(axiMaster,IPCoreInfo,IPCoreName)
myCoreObj = socIPCore(axiMaster,IPCoreInfo,IPCoreName,Name,Value)
```

Description

`myCoreObj = socIPCore(axiMaster,IPCoreInfo,IPCoreName)` creates an `socIPCore` object that connects to an IP core running on an FPGA board. The object uses an `socAXIMaster` object to access memory locations in the IP core. `IPCoreInfo` is a structure generated when you run the **SoC Builder** tool and includes the board and IP core configuration parameters from your model.

You can create `socIPCore` objects representing any of these IPs:

- Traffic generator
- Performance monitor
- Direct memory access (DMA)
- Video DMA (VDMA)
- Video timing controller (VTC)
- VDMA trigger

- Frame buffer
- High definition multimedia interface (HDMI)

`myCoreObj = socIPCore(axiMaster,IPCoreInfo,IPCoreName,Name,Value)` sets properties using one or more name-value pairs. For example,

`myIPobj=socIPCore(axiMaster, perf_mon,'PerformanceMonitor','Mode','Profile');`

creates an `socIPCore` object that connects to an IP core on the specified board and sets the performance monitor mode to profile mode.

Input Arguments

`axiMaster` — Name of socAXIMaster object used for memory-mapped access
`socAXIMaster` object

Name of `socAXIMaster` object used for memory-mapped access, specified as an `socAXIMaster` object.

Create an `socAXIMaster` object using the `socAXIMaster` function, and use the created object as an input to `socIPCore`.

Example: `mySocAXIObj = socAXIMaster('Xilinx'); myIPObj = socIPCore(mySocAXIObj,IPCoreInfo,'DMA')`

`IPCoreInfo` — IP core information
structure

IP core information, specified as a structure generated by the **SoC Builder** tool. To access the structure, load the `.mat` file which is generated by **SoC Builder** tool. The file is named *model_name_boardID_*axmaster.mat. Loading the file will load the structures generated by the **SoC Builder** tool to your workspace.

The structures contain information for vendor IP and for user-specified IP which are specific to your model and board. The structures are named as follows:

- `vdma_frame_buffer` – A `struct` representing a frame buffer.
- `perf_mon` – A `struct` representing a performance monitor.
- `vtc` – A `struct` representing a video timing controller.
- `vdma_hdmi_out` – A `struct` representing a VDMA-based HDMI IP.

- `atg` – A `struct` representing an AXI traffic generator.
- *DUT*_ip – A `struct` representing a user IP named "DUT".

Note The `mat` file loads additional `structs` for IPs, for internal access.

IPCoreName — IP core object type
`'TrafficGenerator'` | `'PerformanceMonitor'` | `'VDMA'` | `'DMA'` | `'VDMATrigger'`
| `'VTC'` | `'FrameBuffer'` | `'HDMI'`

IP core object type, specified as one of the values in this table:

Value	Description
`'TrafficGenerator'`	SoC Blockset memory traffic generator
`'PerformanceMonitor'`	SoC Blockset performance monitor
`'VDMA'`	Xilinx VDMA IP
`'DMA'`	Analog Devices® DMA controller IP
`'VTC'`	Video timing controller
`'VDMATrigger'`	An IP used to trigger reading frames from the source (mm2s) VDMA
`'FrameBuffer'`	VDMA-based frame buffer IP
`'HDMI'`	VDMA-based HDMI IP

Data Types: `string` | `character vector`

Properties

PerfMonMode — Type of performance data to collect
`'Profile'` (default) | `'Trace'`

Type of performance data to collect, specified as `'Profile'` or `'Trace'`. Specify
`'Profile'` mode to collect byte and burst counts for bandwidth and latency plots.
`'Trace'` mode to collect burst transaction event data for display as waveforms.

Object Functions

initialize	Initialize IP core corresponding to socIPCore object
start	Start IP core execution on hardware board

See Also

socAXIMaster

Topics

"Analyze Memory Bandwidth Using Traffic Generators"

Introduced in R2019a

socAXIMaster

Read and write memory locations on hardware board from MATLAB

Description

The `socAXIMaster` object communicates with the MATLAB AXI master IP running on a hardware board. The object uses a JTAG connection to forward read and write commands to the IP and access slave memory locations on the hardware board. Pass an `socAXIMaster` object as an argument when you create an `socIPCore` object, so that the object can access memory locations within the IP core on the board.

Creation

Description

`axiMasterObj = socAXIMaster(vendor)` creates an object that connects to an AXI master IP for the specified `vendor`. This connection enables you to access memory locations in an SoC design from MATLAB.

`axiMasterObj = socAXIMaster(hw)` creates an object that connects to an AXI master IP on the specified hardware board.

`axiMasterObj = socAXIMaster( ___ ,Name,Value)` creates an object with additional properties specified by one or more `Name,Value` pair arguments. Enclose each property name in quotes. Specify properties in addition to the input arguments in previous syntaxes.

Input Arguments

vendor — FPGA brand name
`'Intel' | 'Xilinx'`

FPGA brand name, specified as `'Intel'` or `'Xilinx'`. The AXI master IP varies depending on the type of FPGA you have.

hw — Hardware object
socHardwareBoard object

Hardware object, specified as a socHardwareBoard object that represents the connection to the SoC hardware board.

Properties

JTAGCableType — Type of JTAG cable used for communication with FPGA board (Xilinx boards only)
'auto' (default) | 'FTDI'

Type of JTAG cable used for communication with the FPGA board (Xilinx boards only), specified as 'auto' or 'FTDI'. This property is most useful when more than one cable is connected to the host computer.

When this property is set to 'auto' (default), the object autodetects the JTAG cable type. The object prioritizes searching for Digilent® cables and uses this process to autodetect the cable type.

1 The socAXIMaster object searches for a Digilent cable. If the object finds:

- Exactly one Digilent cable –– The object uses that cable for communication with the FPGA board.

- More than one Digilent cable –– The object returns an error. To resolve this error, specify the desired cable using the JTAGCableName property.

- No Digilent cables –– The object searches for an FTDI cable (see step 2).

2 If no Digilent cable is found, the socAXIMaster object searches for an FTDI cable. If the object finds:

- Exactly one FTDI cable –– The object uses that cable for communication with the FPGA board.

- More than one FTDI cable –– The object returns an error. To resolve this error, specify the desired cable using the JTAGCableName property.

- No FTDI cables –– The object returns an error. To resolve this error, connect a Digilent or FTDI cable.

The cable search in `'auto'` mode prioritizes connection using a Digilent cable. If one Digilent and one FTDI cable are connected to the host computer and this property is set to `'auto'`, the object selects the Digilent cable for communication with the FPGA board.

When this property is set to `'FTDI'`, the object searches for FTDI cables. If the object finds:

- Exactly one FTDI cable –– The object uses that cable for communication with the FPGA board.
- More than one FTDI cable –– The object returns an error. To resolve this error, specify the desired cable using the `JTAGCableName` property.
- No FTDI cables –– The object returns an error. To resolve this error, connect a Digilent or FTDI cable.

For an example, see "Select from Multiple JTAG Cables" on page 4-39.

JTAGCableName — Name of JTAG cable used for communication with FPGA board
`'auto'` (default) | character vector

Name of JTAG cable user for communication with FPGA board, specified as `'auto'` or a character vector. Specify this property if more than one JTAG cable of the same type are connected to the host computer. If the host computer has more than one JTAG cable and you do not specify this property, the object returns an error. The error message contains the names of the available JTAG cables. For an example, see "Select from Multiple JTAG Cables" on page 4-39.

TckFrequency — JTAG clock frequency
15 (default) | positive integer

JTAG clock frequency, in MHz, specified as a positive integer. For Intel FPGAs the JTAG clock frequency must be 12 MHz or 24 MHz. For Xilinx FPGAs, the JTAG clock frequency must be 33 MHz or 66 MHz. The JTAG clock frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

JTAGChainPosition — Position of FPGA in JTAG chain (Xilinx boards only)
1 (default) | positive integer

Position of FPGA in JTAG chain (Xilinx boards only), specified as a positive integer. Specify this property value if more than one FPGA or Zynq® device is on the JTAG chain.

IRLengthBefore — Sum of instruction register length for all devices before target FPGA (Xilinx boards only)
0 (default) | nonnegative integer

Sum of instruction register length for all devices before target FPGA (Xilinx boards only), specified as a nonnegative integer. Specify this property value if more than one FPGA or Zynq device is on the JTAG chain.

IRLengthAfter — Sum of instruction register length for all devices after target FPGA (Xilinx boards only)
0 (default) | nonnegative integer

Sum of instruction register length for all devices after target FPGA (Xilinx boards only), specified as a nonnegative integer. Specify this property value if more than one FPGA or Zynq device is on the JTAG chain.

Object Functions

readmemory   Read data from AXI4 memory-mapped slaves
release   Release JTAG cable resource
writememory   Write data to AXI4 memory-mapped slaves

Examples

Initialize Memory on SoC Hardware Board from MATLAB

For an example of how to configure and use the AXI master IP in your design, see "Random Access of External Memory". Specifically, review the soc_image_rotation_axi_master.m script that initializes the memory on the device, starts the FPGA logic, and reads back the modified data. This example shows only the memory initialization step.

Load a .mat file that contains structures derived from the board configuration parameters. This file was generated by **SoC Builder**. These structures also describe the IP cores and memory configuration of the design on the board. Set up a JTAG AXI master connection by creating a socHardwareBoard and passing it to the socAXIMaster object. The socAXIMaster object connects with the hardware board and confirms that the IP is present.

```
load('soc_image_rotation_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Initialize the memory contents on the device by loading the figure data and writing it to
Region1. The FPGA logic is designed to read this data, rotate it, and write it into
Region2. Clear the contents of Region2.

```
load('soc_image_rotation_inputdata.mat');
inputFigure = smallImage;
[x, y] = size(inputFigure);
inputImage = uint32(reshape(inputFigure',1,x*y));
writememory(AXIMasterObj,memRegions.AXI4MasterMemRegion1,inputImage);
writememory(AXIMasterObj,memRegions.AXI4MasterMemRegion2,uint32(zeros(1,x*y)));
```

Access Memory on SoC Hardware Board from MATLAB

For this example, you must have a design running on a hardware board connected to the
MATLAB host machine.

Create a MATLAB AXI master object. The object connects with the hardware board and
confirms that the IP is present. You can create the object with a vendor name or an
socHardwareBoard object.

```
mem = socAXIMaster('Xilinx');
```

Write and read one or more addresses with one command. By default, the functions auto-
increment the address for each word of data. For instance, write ten addresses, then read
the data back from a single location.

```
writememory(mem,140,[10:19])
rd_d = readmemory(mem,140,1)

rd_d =

  uint32

    10
```

Now, read the written data from ten locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =

  1×10 uint32 row vector

   10    11    12    13    14    15    16    17    18    19
```

Set the `BurstType` property to `'Fixed'` to turn off the auto-increment and access the same address multiple times. For instance, read the written data ten times from the same address.

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')

rd_d =

  1×10 uint32 row vector

   10    10    10    10    10    10    10    10    10    10
```

Write incrementing data ten times to the same address. The final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed')
rd_d = readmemory(mem,140,10)

rd_d =

  1×10 uint32 row vector

   29    11    12    13    14    15    16    17    18    19
```

Alternatively, specify the address as a hexadecimal string. To cast the read data to a data type other than `uint32`, use the `OutputDataType` property.

```
writememory(mem,'1c',[0:4:64])
rd_d = readmemory(mem,'1c',16,'OutputDataType',numerictype(0,6,4))

rd_d =

  Columns 1 through 10
        0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.0
  Columns 11 through 16
   2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
```

```
      WordLength: 6
   FractionLength: 4
```

When you are done accessing the board, release the JTAG connection.

```
release(mem)
```

Select from Multiple JTAG Cables

When multiple JTAG cables are connected to your host computer, the object prioritizes Digilent cables over FTDI cables. To use an FTDI cable, specify the JTAG cable type property.

```
h = socAXIMaster('Xilinx', 'JTAGCableType', 'FTDI')
```

If two cables of the same type are connected to your host computer, specify the `JTAGCableName` property for the board where the JTAG master IP is running. To see the JTAG cable identifiers, attempt to create an `socAXIMaster` object, which, in this case, errors and returns a list of the current JTAG cable names.

```
h = socAXIMaster('Xilinx')
```

```
Error using fpgadebug_mex
Found more than one JTAG cable:
0 (JtagSmt1): #tpt_0001#ptc_0002#210203991642
1 (Arty): #tpt_0001#ptc_0002#210319789795
Please disconnect the extra cable, or specify the cable name as an input argument.
See documentation of FPGA Data Capture or MATLAB as AXI master to learn how to set
the cable name.
```

To communicate with this Arty board, specify the matching JTAG cable name.

```
h = socAXIMaster('Xilinx','JTAGCableName','#tpt_0001#ptc_0002#210319789795')
```

See Also
socIPCore

Topics
"Random Access of External Memory"

Introduced in R2019a

socMemoryProfiler

Retrieve and display memory performance data

Description

This object collects and displays two types of memory performance data from an AXI memory interconnect IP running on your SoC hardware board. You can collect average transaction latency and counts of bytes and bursts and then plot bandwidth, burst counts, and transaction latency, or collect detailed memory transaction event data and view the data as waveforms.

Creation

Syntax

```
profiler = socMemoryProfiler(hw,performanceMonitor)
```

Description

`profiler = socMemoryProfiler(hw,performanceMonitor)` creates an object that accesses the AXI interconnect monitor IP on the board specified by the `socHardwareBoard` object, `hardware`, and uses the IP configuration from the IP core object, `performanceMonitor`.

Input Arguments

hw — Hardware object
`socHardwareBoard` object

Hardware object, specified as a `socHardwareBoard` object that represents the connection to the SoC hardware board.

performanceMonitor — AXI interconnect monitor IP core object
socIPCore object

AXI interconnect monitor IP core object, specified as an socIPCore object that was
created with the IPCoreName argument set to 'PerformanceMonitor', and then
initialized. For example,

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',perfMonMode);
initialize(apmCoreObj);
```

- AXIMasterObj is an socAXIMaster object.
- perf_mon is a structure generated by the **SoC Builder** tool.
- perfMonMode is a string equal to either 'Profile' or 'Trace'. 'Profile' mode
  collects byte and burst counts for bandwidth and latency plots. 'Trace' mode collects
  burst transaction event data for display as waveforms.

Object Functions

collectMemoryStatistics   Retrieve performance data from AXI interconnect monitor
plotMemoryStatistics   Plot performance data obtained from AXI interconnect monitor

Examples

Configure and Query AXI Interconnect Monitor

The AXI interconnect monitor (AIM) is an IP core that collects performance metrics for an
AXI-based FPGA design. Create an socIPCore object to setup and configure the AIM IP,
and use the socMemoryProfiler object to retrieve and display the data.

For an example of how to configure and query the AIM IP in your design using MATLAB
as AXI Master, see "Analyze Memory Bandwidth Using Traffic Generators". Specifically,
review the soc_memory_traffic_generator_axi_master.m script that configures
and monitors the design on the device.

The performance monitor can collect two types of data. Choose *Profile* mode to collect
average transaction latency and counts of bytes and bursts. In this mode, you can launch
a performance plot tool, and then configure the tool to plot bandwidth, burst count, and
transaction latency. Choose *Trace* mode to collect detailed memory transaction event data
and view the data as waveforms.

```
Mode = 'Profile'; % or 'Trace'
```

To obtain diagnostic performance metrics from your generated FPGA design, you must set up a JTAG connection to the device from MATLAB. Load a `.mat` file that contains structures derived from the board configuration parameters. This file was generated by the **SoC Builder** tool. These structures describe the memory interconnect and masters configuration such as buffer sizes and addresses. Use the `socHardwareBoard` object to set up the JTAG connection.

```
load('soc_memory_traffic_generator_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Configure the AIM. The `socIPCore` object provides a function that performs this initialization. Then, create an `socMemoryProfiler` object to gather the metrics.

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',Mode);
initialize(apmCoreObj);
profilerObj = socMemoryProfiler(hwObj,apmCoreObj);
```

Retrieve performance metrics or signal data from a design running on the FPGA by using the `socMemoryProfiler` object functions.

For `'Profile'` mode, call the `collectMemoryStatistics` function in a loop.

```
NumRuns = 100;
for n = 1:NumRuns
    collectMemoryStatistics(profilerObj);
end
```

JTAG design setup time is long relative to FPGA transaction times, and if you have a small number of transactions in your design, they might have already completed by the time you query the monitor. In this case, the bandwidth plot shows only one sample, and the throughput calculation is not accurate. If this situation occurs, increase the total number of transactions the design executes.

For `'Trace'` mode, call the `collectMemoryStatistics` function once. This function stops the IP from writing transactions into the FIFO in the AXI interconnect monitor IP, although the transactions continue on the interconnect. Set the size of the transaction FIFO, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.

```
collectMemoryStatistics(profilerObj);
```

Visualize the performance data by using the `plotMemoryStatistics` function. In `'Profile'` mode, this function launches a performance plot tool, and you can configure the tool to plot bandwidth, burst count, and average transaction latency. In `'Trace'` mode, this function opens the **Logic Analyzer** tool to view burst transaction event data.

```
plotMemoryStatistics(profilerObj);
```

See Also
"Memory Performance Information from FPGA Execution"

Topics
"Analyze Memory Bandwidth Using Traffic Generators"

Introduced in R2019a

Tools — Alphabetical List

Logic Analyzer

Visualize, measure, and analyze transitions and states over time

Description

The **Logic Analyzer** is a tool for visualizing and inspecting signals in your Simulink model. Using the **Logic Analyzer**, you can:

- Debug and analyze models
- Trace and correlate many signals simultaneously
- Detect and analyze timing violations
- Trace system execution
- Detect signal changes using triggers

For keyboard shortcuts, click More.

Keyboard Shortcuts

Actions	Description	Applicable When
Ctrl+X	Cut	Wave is selected
Ctrl+C	Copy	Wave is selected
Ctrl+V	Paste	Wave is selected
Delete	Delete	Wave is selected
Ctrl+-	Zoom out	Always
Shift+Ctrl+-	Zoom out around active cursor	Always
Ctrl++	Zoom in	Always
Shift+Ctrl++	Zoom out around active cursor	Always
Shift+Ctrl+C	Move display to active cursor	When cursor is not in the display range

Actions	Description	Applicable When
Space	Zoom out full	Always
Tab, **Right Arrow**	Next transition	Digital format wave is selected
Shift+Tab, **Left Arrow**	Previous transition	Digital format wave is selected
Ctrl+A	Select all waves	Always
Up Arrow	Select wave above selected	Wave is selected
Down Arrow	Select wave below selection	Wave is selected
Ctrl+Up Arrow	Move selected waves up	Wave is selected
Ctrl+Down Arrow	Move selected waves down	Wave is selected
Escape	Unselect all signals	Wave is selected
Page Up	Scroll up	Always
Page Down	Scroll down	Always

Open the Logic Analyzer App

On the Simulink toolstrip Simulation tab, click the **Logic Analyzer** app button. If the button is not displayed, expand the review results app gallery. Your most recent choice for data visualization is saved across Simulink sessions.

To visualize referenced models, you must open the Logic Analyzer from the referenced model. You should see the name of the referenced model in the Logic Analyzer toolbar.

Examples

Select Signals to Analyze

The **Logic Analyzer** supports several methods for selecting data to visualize.

- Select a signal in your model. When you select a signal, an ellipsis appears above the signal line. Hover over the ellipsis to view options and then select the **Enable Data Logging** option.
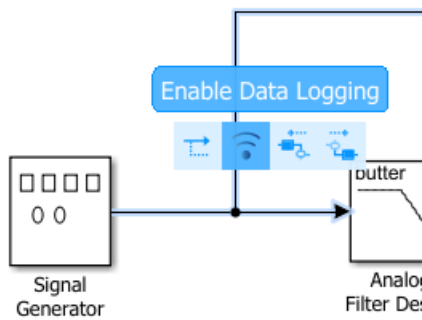
- Right-click a signal in your model to open an options dialog box. Select the **Log Selected Signals** option.
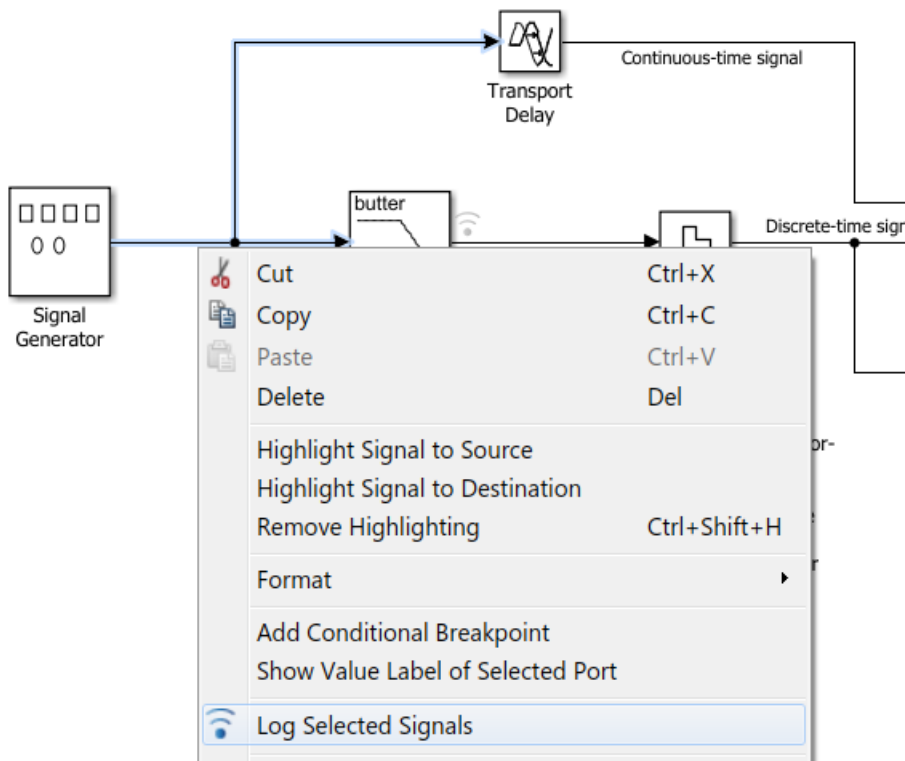
- Use any method to select multiple signal lines in your model. For example, use **Shift** +click to select multiple lines individually or **CTRL+A** to select all lines at once. Then click the **Logic Analyzer** button arrow and select **Log Selected Signals**.
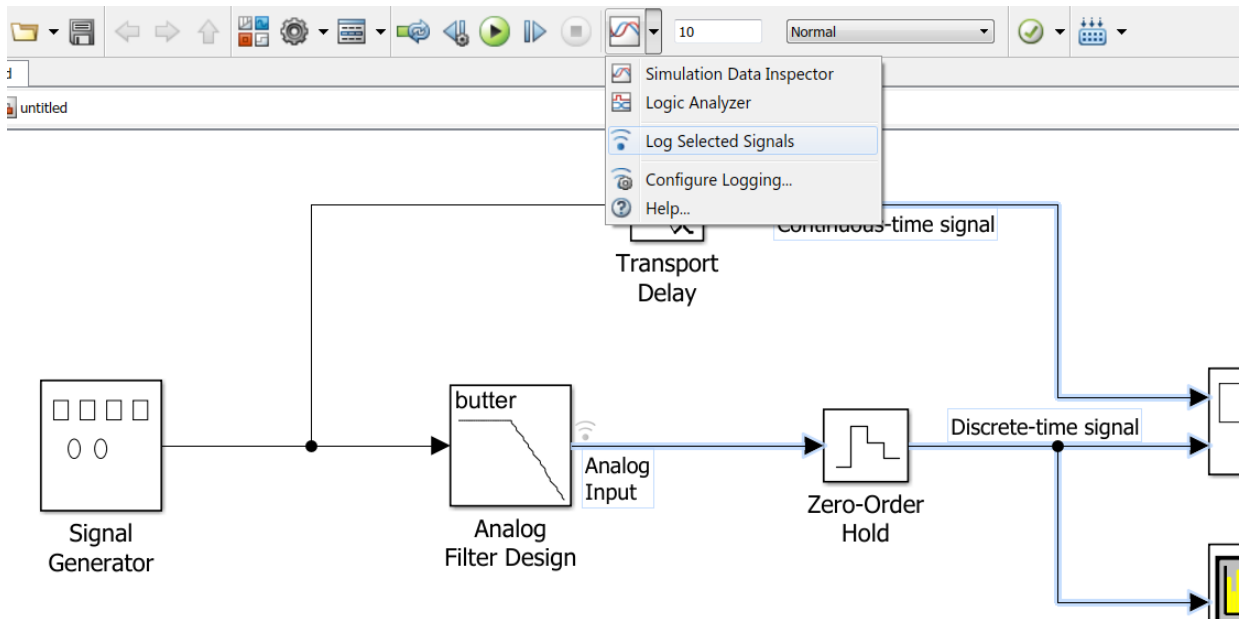
When you open the **Logic Analyzer**, all signals marked for logging are listed. You can add and delete waves from your **Logic Analyzer** while it is open. Adding and deleting signals does not disable logging, only removes the signal from the Logic Analyzer.

Modify Global Settings

Open the **Logic Analyzer** and select **Settings** from the toolstrip. A global settings dialog box opens. Any setting you change for an individual signal supersedes the global setting. The Logic Analyzer saves any setting changes with the model (Simulink) or System object™ (MATLAB).
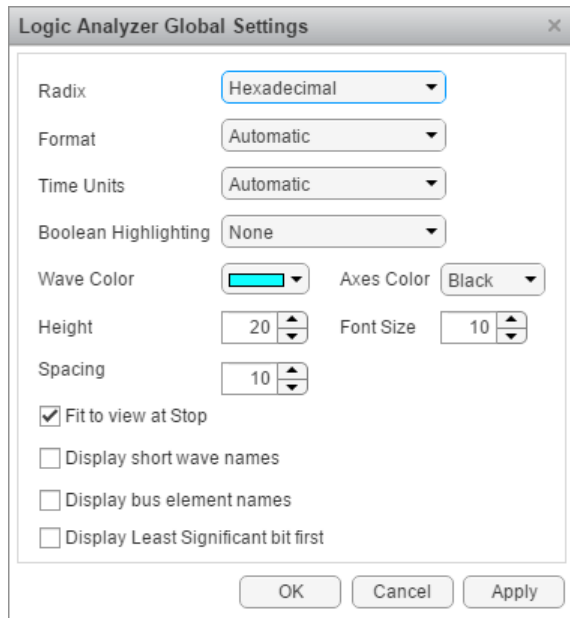
Set the display **Radix** of your signals as one of the following:

- `Hexadecimal` — Displays values as symbols from zero to nine and A to F
- `Octal` — Displays values as numbers from zero to seven
- `Binary` — Displays values as zeros and ones
- `Signed decimal` — Displays the signed, stored integer value
- `Unsigned decimal` — Displays the stored integer value

Set the display **Format** as one of the following:

- `Automatic` — Displays floating point signals in `Analog` format and integer and fixed-point signals in `Digital` format. Boolean signals are displayed as zero or one.
- `Analog` — Displays values as an analog plot
- `Digital` — Displays values as digital transitions

Set the display **Time Units** to one of the following:

- `Automatic` — Uses a time scale appropriate to the time range shown in the current plot

- seconds
- milliseconds
- microseconds
- nanoseconds
- picoseconds
- femtoseconds

Set the **Boolean Highlighting** to one of the following:

- None
- Rows — Adds a highlighted background for the entire Boolean signal row.
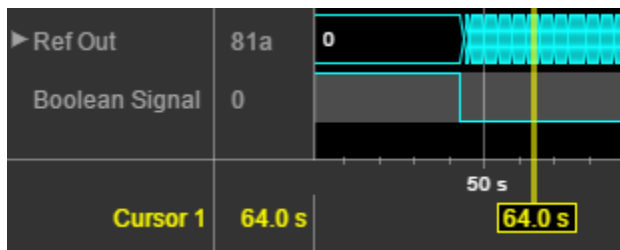
Select **Highlight boolean values** to add highlighting to Boolean signals.

- Gradient— Adds color highlighting to Boolean signals based on value. If the signal value is true, the highlight fades out below. If the signal value is false, the signal fades out above. With this option, you can visually deduce the value of the signal.
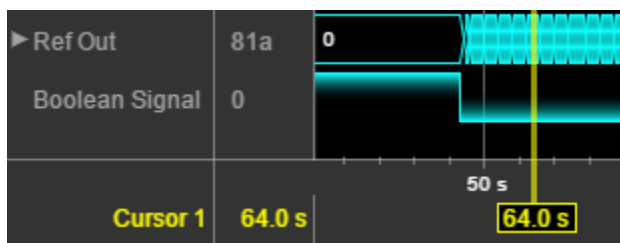
Inspect the graphic for an explanation of the global settings: Wave Color, Axes Color, Height, Font Size, and Spacing. Font Size applies only to the text within the axes.
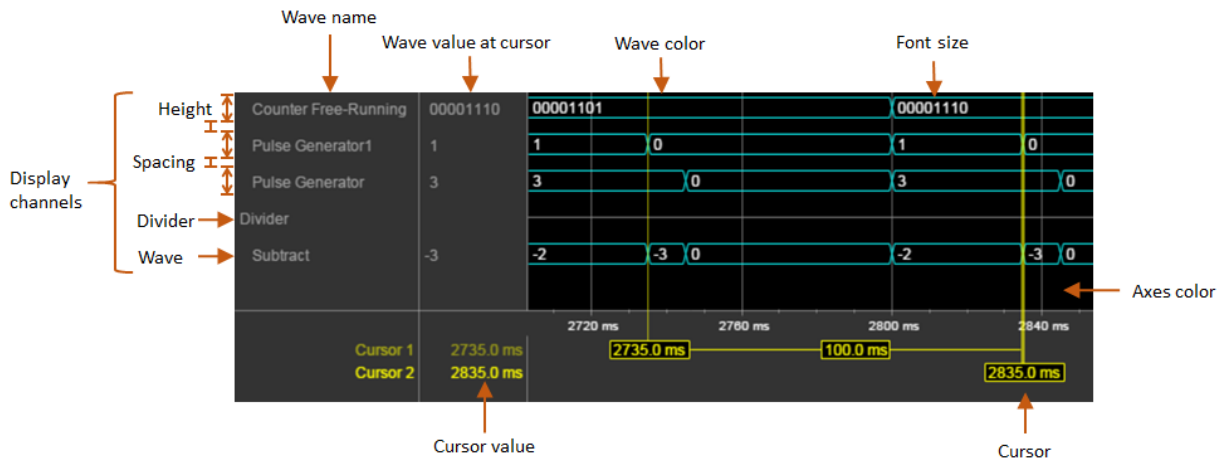
By default, when your simulation stops, the Logic Analyzer shows all the data for the simulation time on one screen. If you do not want this behavior, clear **Fit to view at Stop**. This option is disabled for long simulation times.

To display the short names of waves without path information, select **Display short wave names**.

You can expand fixed-point and integer signals and view individual bits. The **Display Least Significant bit first** option enables you to reverse the order of the displayed bits.

If you stream logged bus signals to the Logic Analyzer, you can display the names of the signals inside the bus using the **Display bus element names** option. To show bus element names:

1 Add the bus signal for logging.

2 In the Logic Analyzer settings, select the **Display bus element names** check box.

3 Run the simulation.

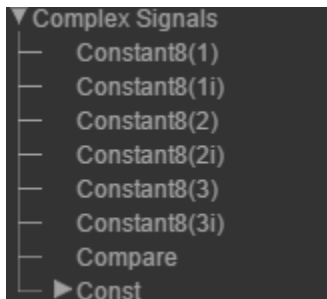When you expand the bus signals, you will see the bus signal names.
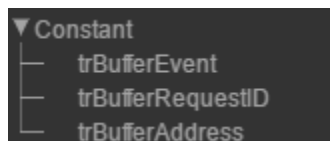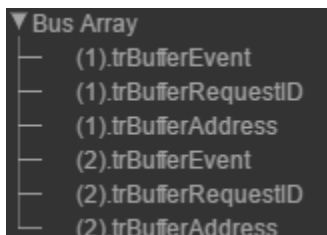
Some special situations:

- If the signal has no name, the Logic Analyzer shows the block name instead.
- If the bus is a bus object, the Logic Analyzer shows the bus element names specified in the Bus Object Editor.
- If one of the bus elements contains an array, each element of the array is appended with the element index.

```
▼ Bus Creator
├─   a
├─   b(1)
├─   b(2)
├─   Constant(1)
├─   Constant(2)
└─   Constant(3)
```
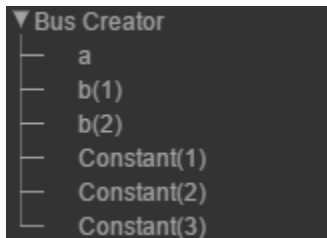
- If a bus element contains an array with complex elements, the real and complex values (`i`) are split.

```
▼ Complex Signals
├─   Constant8(1)
├─   Constant8(1i)
├─   Constant8(2)
├─   Constant8(2i)
├─   Constant8(3)
├─   Constant8(3i)
├─   Compare
└─ ▶ Const
```

- Bus signals passed through a Gain block are labeled `Gain(1)`, `Gain(2)`,…`Gain(n)`.
- If the bus contains an array of buses, the Logic Analyzer prepends the element name with the bus array index.

```
▼ Bus Array
├─   (1).trBufferEvent
├─   (1).trBufferRequestID
├─   (1).trBufferAddress
├─   (2).trBufferEvent
├─   (2).trBufferRequestID
├─   (2).trBufferAddress
```

Modify Individual Wave Settings

Open the **Logic Analyzer** and select a wave by double-clicking the wave name. Then from the **Wave** tab, set parameters specific to the individual wave you selected. Any setting made on individual signals supersedes the global setting. To return individual wave parameters to the global settings, click **Reset**.
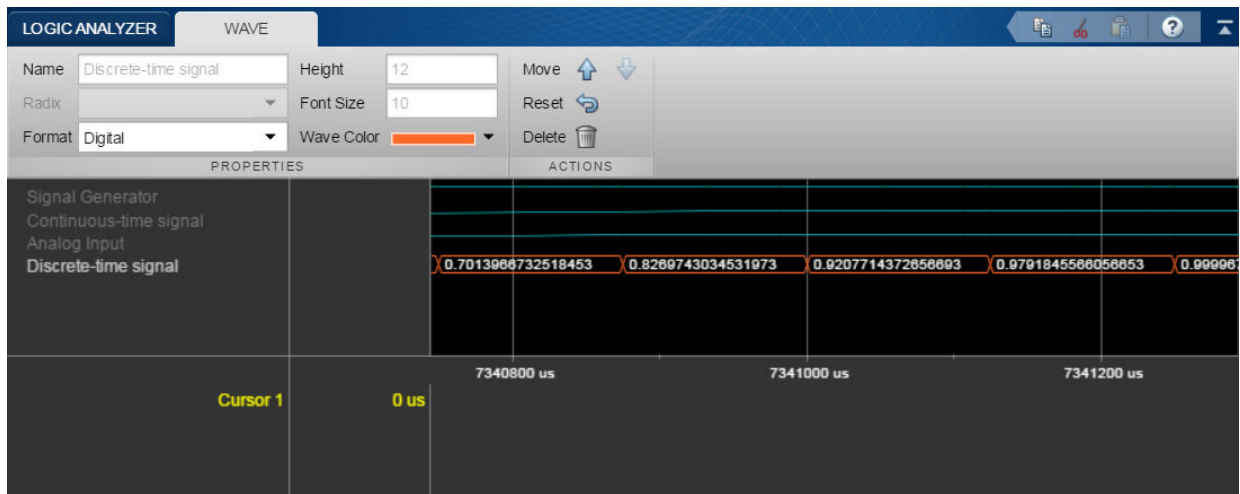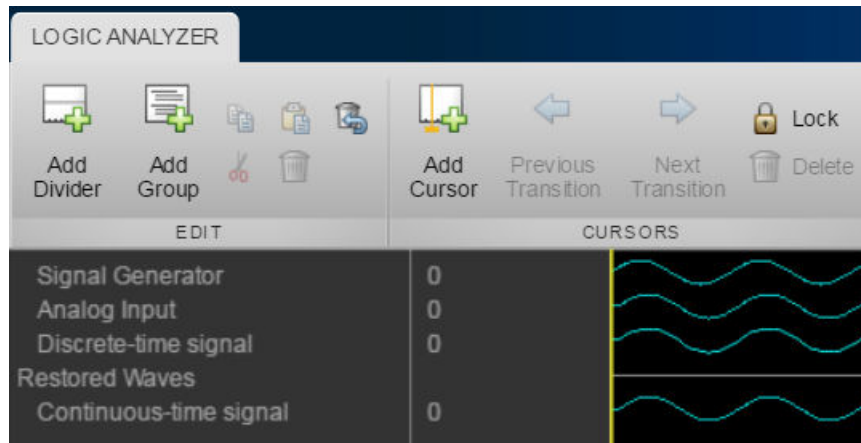

Delete and Restore Waves

1 Open the **Logic Analyzer** and select a wave by clicking the wave name.
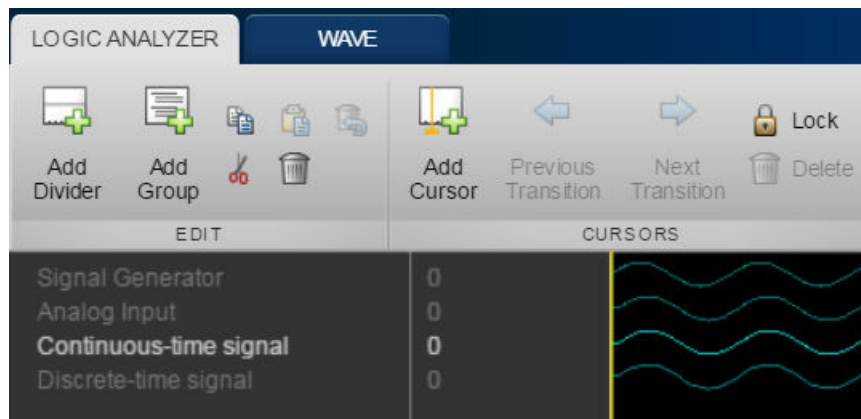
2 From the **Logic Analyzer** toolstrip, click 🗑. The wave is removed from the **Logic Analyzer**.

**3** To restore the wave, from the **Logic Analyzer** toolstrip, click 🔄.

A divider named **Restored Waves** is added to the bottom of your channels, with all deleted waves placed below it.

Add Trigger

The **Logic Analyzer** trigger allows you to find data points based on certain conditions. This feature is useful for debugging or testing when you need to find a specific signal change.

1. Open the **Logic Analyzer** and select the **Trigger** tab.
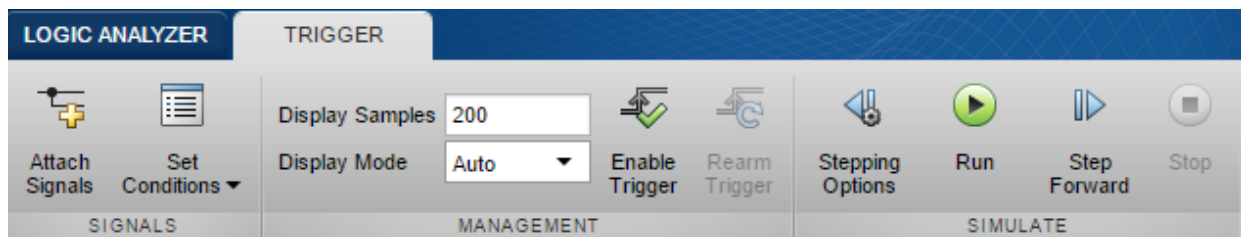
2. To attach a signal to the trigger, select **Attach Signals**, then select the signal you want to trigger on. You can attach up to 20 signals to the trigger. Each signal can have only one triggering condition.

3. By default, the trigger looks for rising edges in the attached signals. You can set the trigger to look for rising or falling edges, bit sequences, or a comparison value. To change the triggering conditions, select **Set Conditions**.

    If you add multiple signals to the trigger, control the trigger logic using the **Operator** option:

- `AND` - match all conditions.
- `OR` - match any condition.

4 To control how many samples you see before triggering, set the **Display Samples** option. For example, if you set this option to `500`, the **Logic Analyzer** tries to give you 500 samples before the trigger. Depending on the simulation, the **Logic Analyzer** may show more or fewer than 500 samples before the trigger. However, if the trigger is found before the 500th sample, the Logic Analyzer still shows the trigger.

5 Control the trigger mode using **Display Mode**.

- `Once` - The **Logic Analyzer** marks only the first location matching the trigger conditions and stops showing updates to the Logic Analyzer. If you want to reset the trigger, select **Rearm Trigger**. Relative to the current simulation time, the **Logic Analyzer** shows the next matching trigger event.
- `Auto` - The **Logic Analyzer** marks every location matching the trigger conditions.

6 Before running the simulation, select **Enable Trigger**. A blue cursor appears as time 0. Then, run the simulation. When a trigger is found, the **Logic Analyzer** marks the location with a locked blue cursor.

Choose Visible Instance of Multi-Reference Model Block

The **Logic Analyzer** can stream only a single instance of a multi-instance Model block. If the same model is opened across different windows, those models will share the same Logic Analyzer. This example shows how to select an instance of a multi-instance Model block for logging on the **Logic Analyzer**.
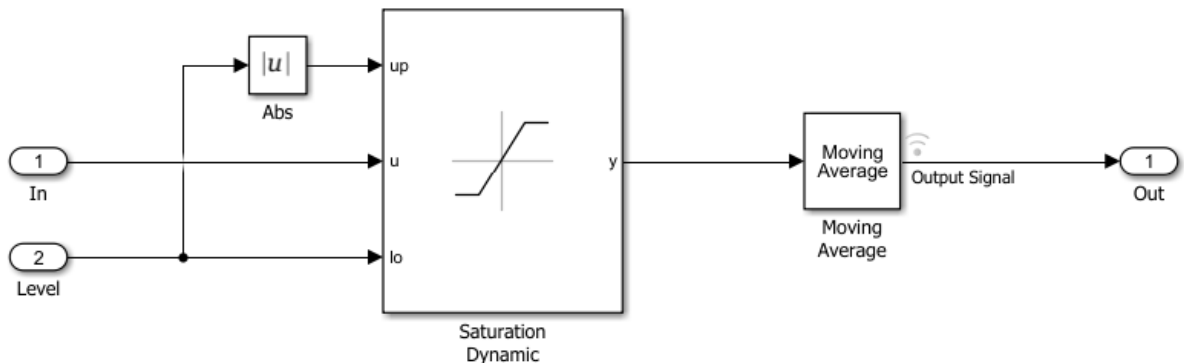
1 Open the multipleModelInstances model:

```
open_system(fullfile(matlabroot,'examples','dsp','multipleModelInstances.slx'))
```
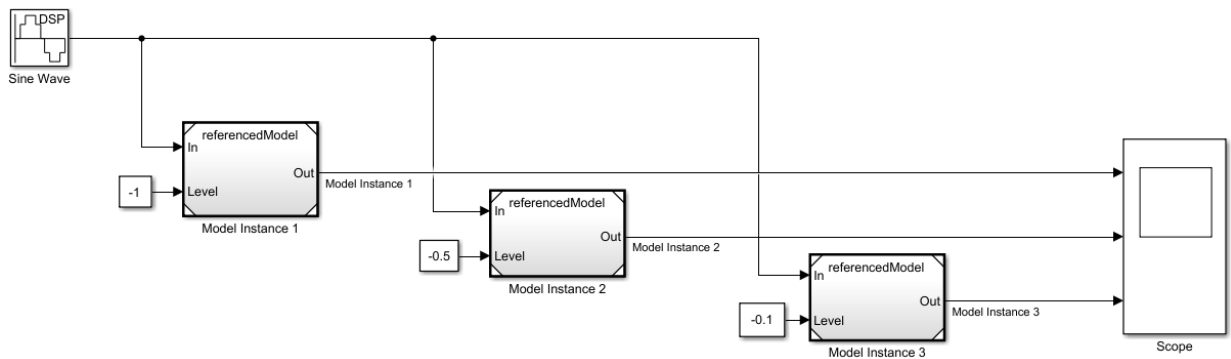
The model contains three instances of the `referencedModel` model.

2   Double-click any of the Model blocks to open the model referenced by all three Model blocks.

3   Open the **Logic Analyzer** in the referenced model by double-clicking the logging symbol next to the Moving Average block.

You should see `referencedModel - [multipleModelInstances]` in the toolbar of the Logic Analyzer.

4   From the Logic Analyzer window, run the model. By running the simulation from a referenced version of `referencedModel`, Simulink runs the top model (`multipleModelInstances`) and referenced models (`referencedModel`). The **Logic Analyzer** displays a single instance of a multi-instant Model block.
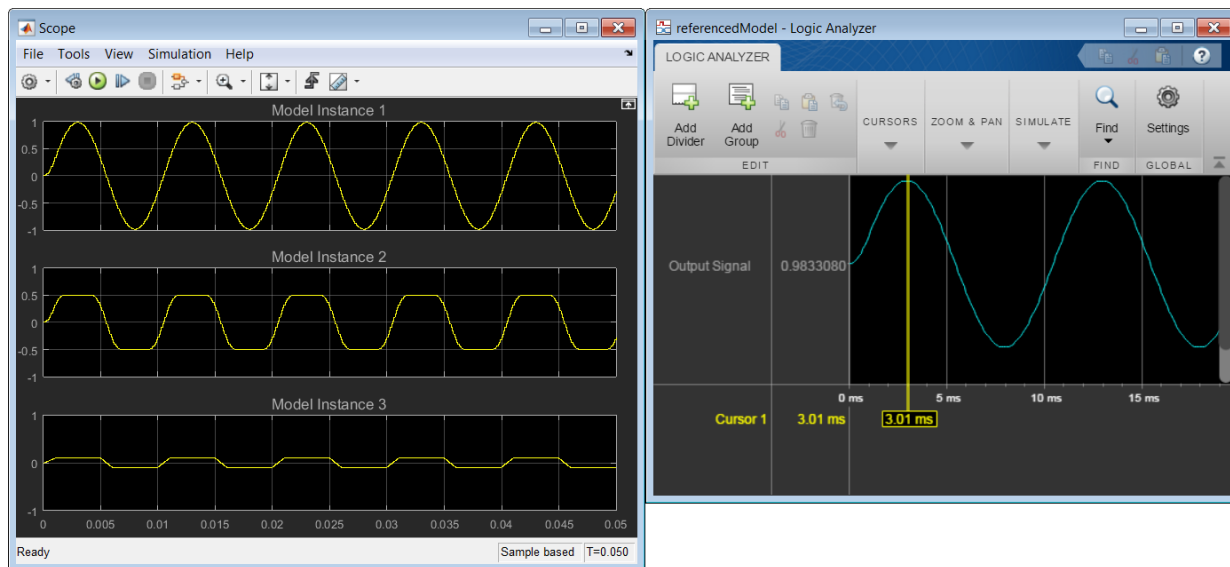
Note When you run a simulation, the logic analyzer runs the model listed in the Logic Analyzer toolbar. If this model is a referenced model, the toolbar also lists the top model and you will see results from running the top model.

To view results from the referenced model in isolation, you must open the referenced model as a top model.

5   To switch between instances, from the Simulink Editor menu, on the Simulation tab open the Prepare gallery and select **Normal Mode Diagram** > **Subsystem & Model Reference** > **Model Block Normal Mode Visibility**. Select **Model Instance 3** and then click **OK**.
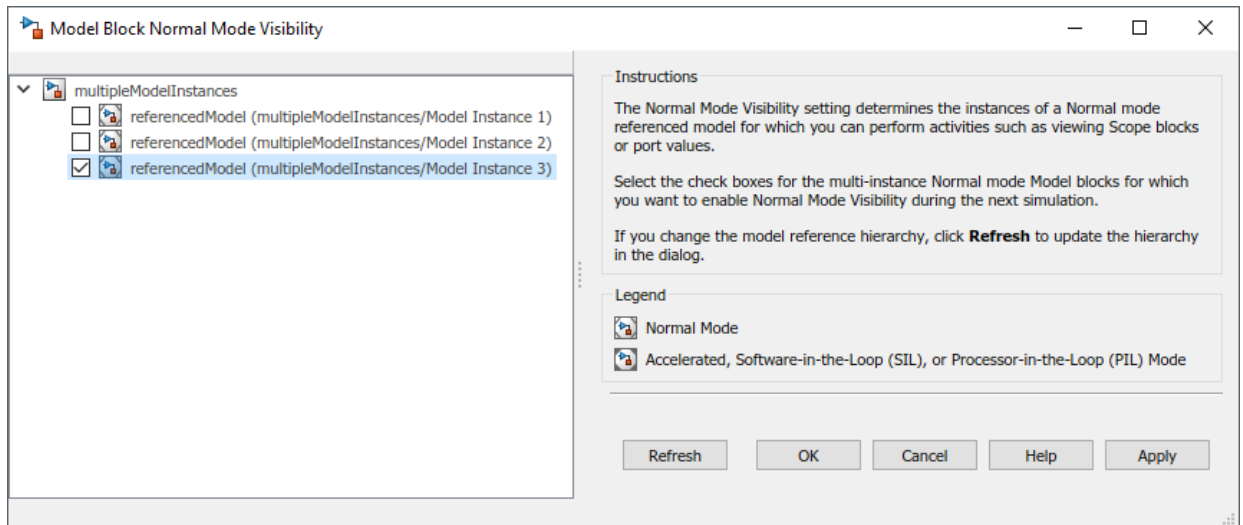
6 Run the `multipleModelInstances` model again. The **Logic Analyzer** displays `Model Instance 3` data.
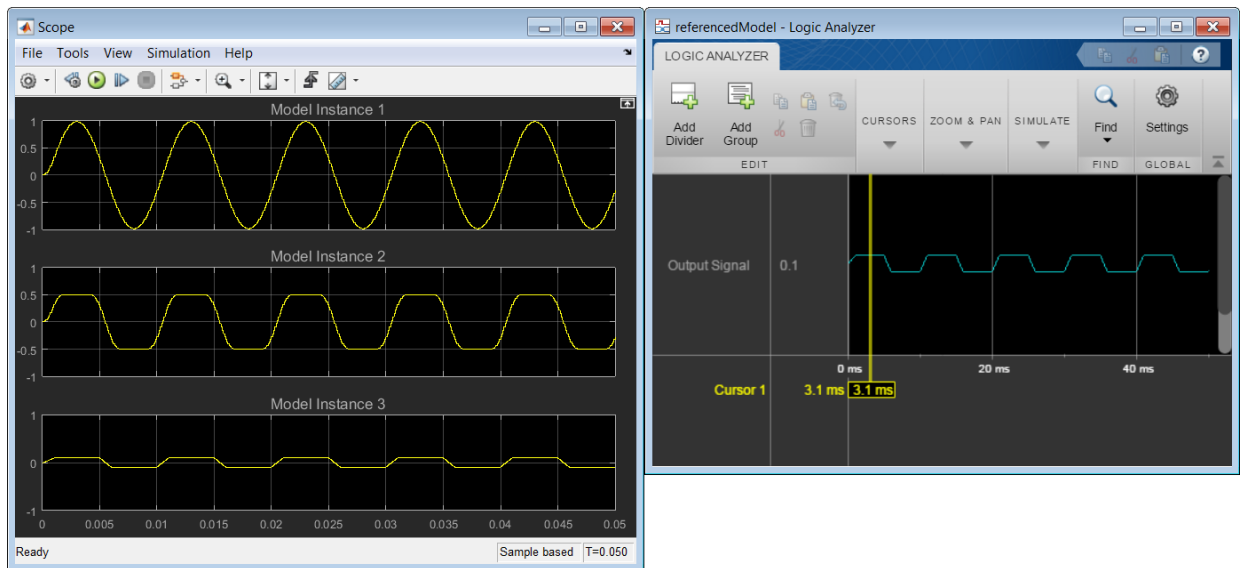
View Bit-Expanded Wave and Reverse Display Order of Bits

The **Logic Analyzer** enables you to bit-expand fixed-point and integer waves.
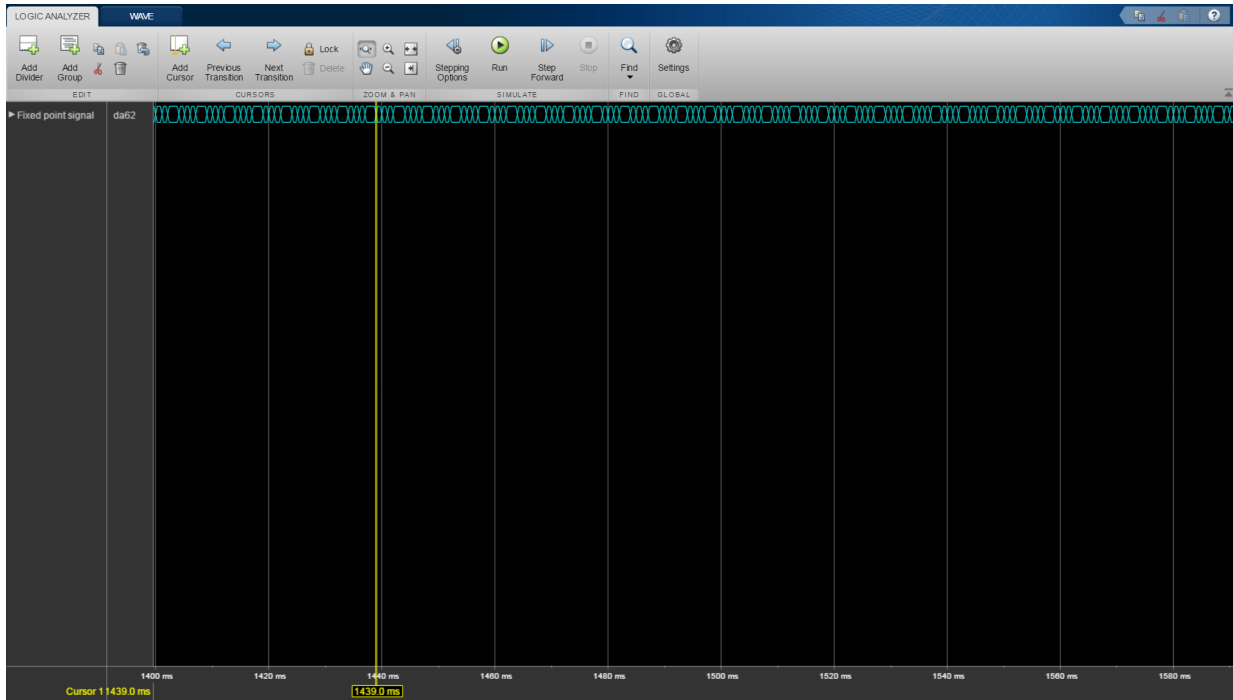
1. In the **Logic Analyzer**, click the arrow next to a fixed-point or integer wave to view the bits.

   The least significant bit and the most significant bit are marked with **lsb** and **msb** next to the wave names.

2   Click Settings, and then select **Display Least Significant bit first** to reverse the order of the displayed bits.
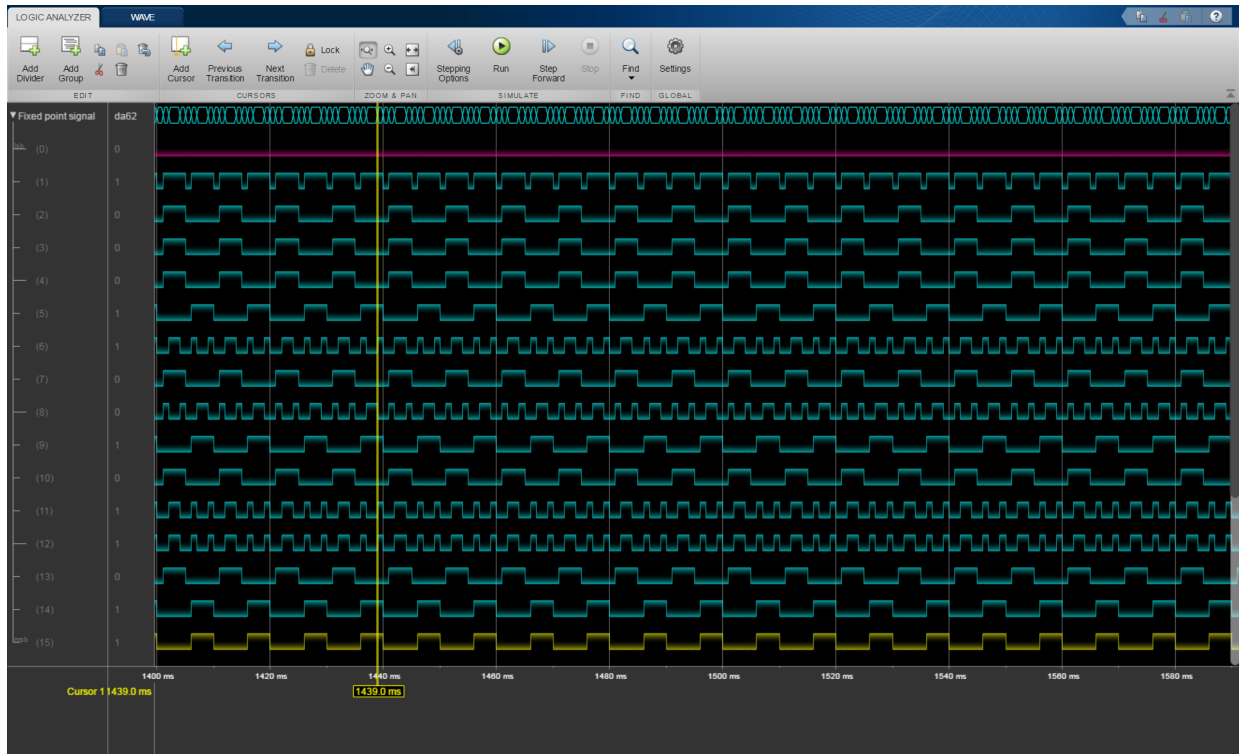
Limitations

- The **Logic Analyzer** does not support frame-based processing.

- If you enable the configuration parameter **Log Dataset data to file**, you cannot stream logged data to the **Logic Analyzer**.

- While the simulation is running, you cannot zoom, pan, or modify the trigger.

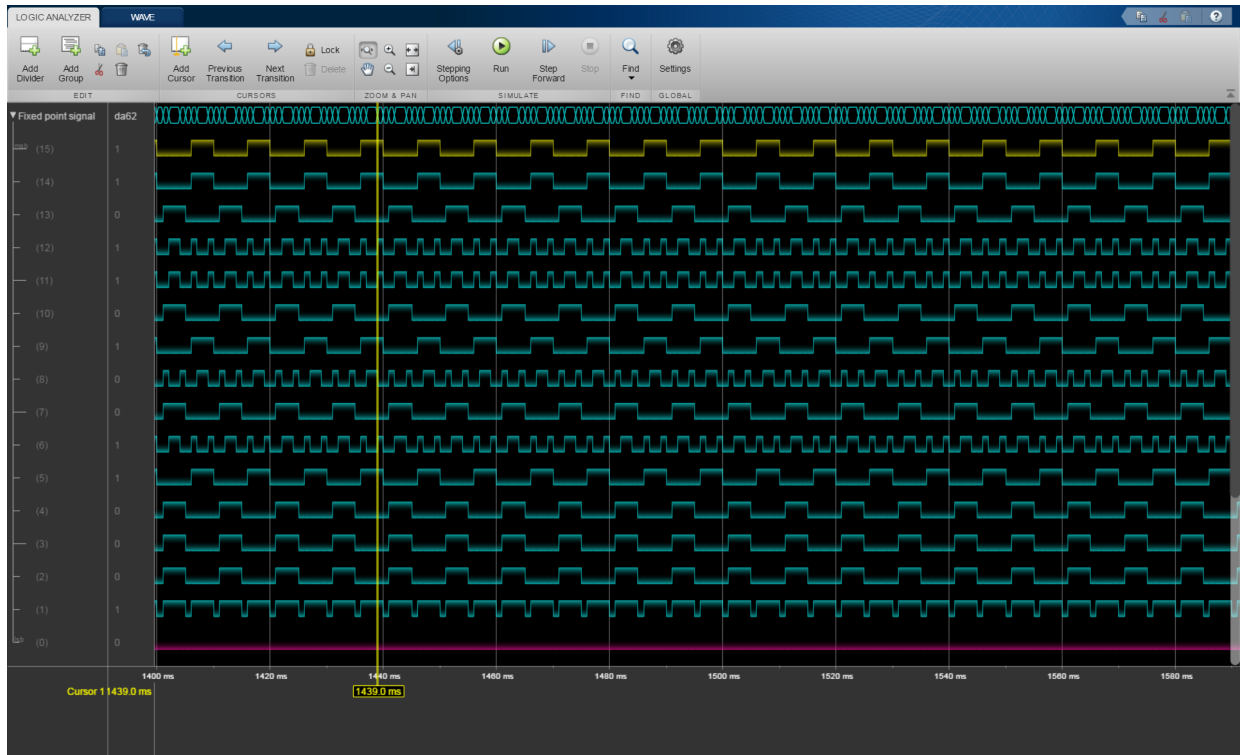- Signals marked for logging for the **Logic Analyzer** must have fewer than 8000 samples per simulation step.

- To visualize constant signals, in the settings, you must set the **Format** to `Digital`. Constants marked for logging are visualized as a continuous transition.

- Signals marked for logging using `Simulink.sdi.markSignalForStreaming` or visualized with a Dashboard Scope do not appear on the **Logic Analyzer**.

- Integers larger than 64 bits and fixed-point signals larger than 128 bits are not supported.
- You cannot visualize Data Store Memory block signals in the **Logic Analyzer** if you set the `Log data store data` parameter to on.
- To visualize signals in referenced models, open the **Logic Analyzer** from the referenced model.
- You may see performance degradation in the **Logic Analyzer** for large matrices (greater than 500 elements) and buses with more than 1000 signals.
- The **Logic Analyzer** does not support Stateflow® state or chart signals.
- The following table shows which simulation modes are supported by the **Logic Analyzer**:

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	You cannot use the **Logic Analyzer** to visualize signals in Model blocks with **Simulation mode** set to `Accelerator`.
Rapid Accelerator	Yes	Data is not available in the **Logic Analyzer** during simulation. If you simulate a model with the simulation mode set to rapid accelerator, after simulation the following signals cannot be visualized in the **Logic Analyzer**: • Multi-instance model reference signals • Nonvirtual bus signals
Processor-in-the-loop (PIL)	No	
Software-in-the-loop (SIL)	No	
External	No	

For more information about these modes, see "How Acceleration Modes Work" (Simulink).

See Also

Objects

Topics
"Generate HDL Code for Programmable FIR Filter" (HDL Coder)
"Packet-Based ADS-B Transceiver"

Introduced in R2016b

Memory Mapper

Configure memory map for SoC application

Description

View and edit memory regions of an SoC application. Edit device base addresses and offsets for memory-mapped devices.

Using the **Memory Mapper** tool, you can:

- View and edit base addresses, offsets, and memory locations of various channels and memory-mapped components in your design.
- Check the memory map of your model for any conflicts between different memory channel configurations.
- Reset the memory map to its default settings.
- Reconcile an edited map to match model settings.

Open the Memory Mapper

- In the Configuration Parameters dialog box, select **Hardware Implementation** from the left pane. Under **Target hardware resources**, select **FPGA design (top-level)** and click **View/Edit Memory Map**.
- In the SoC Builder tool, in the **Review Memory Map** section, click **View/Edit**.

Examples

Reconcile Model with Memory Map

Consider a model with three masters (represented by Memory Traffic Generator blocks), connected to a Memory Controller block.
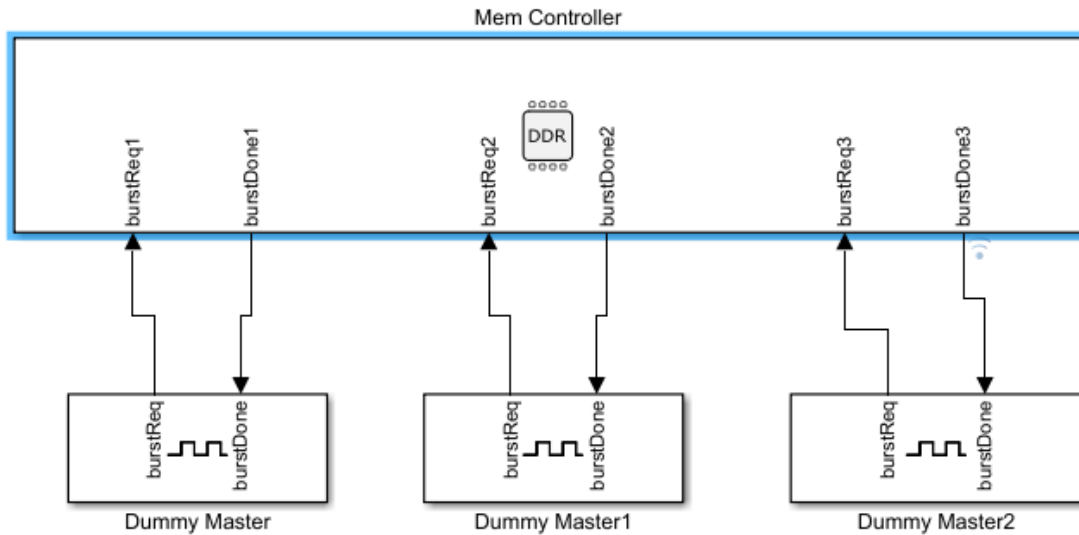
To open the **Memory Mapper** tool, first open the Configuration Parameters dialog box, and then select **Hardware Implementation** from the left pane. Under **Target hardware resources**, select **FPGA design (top-level)** and click **View/Edit Memory Map**.
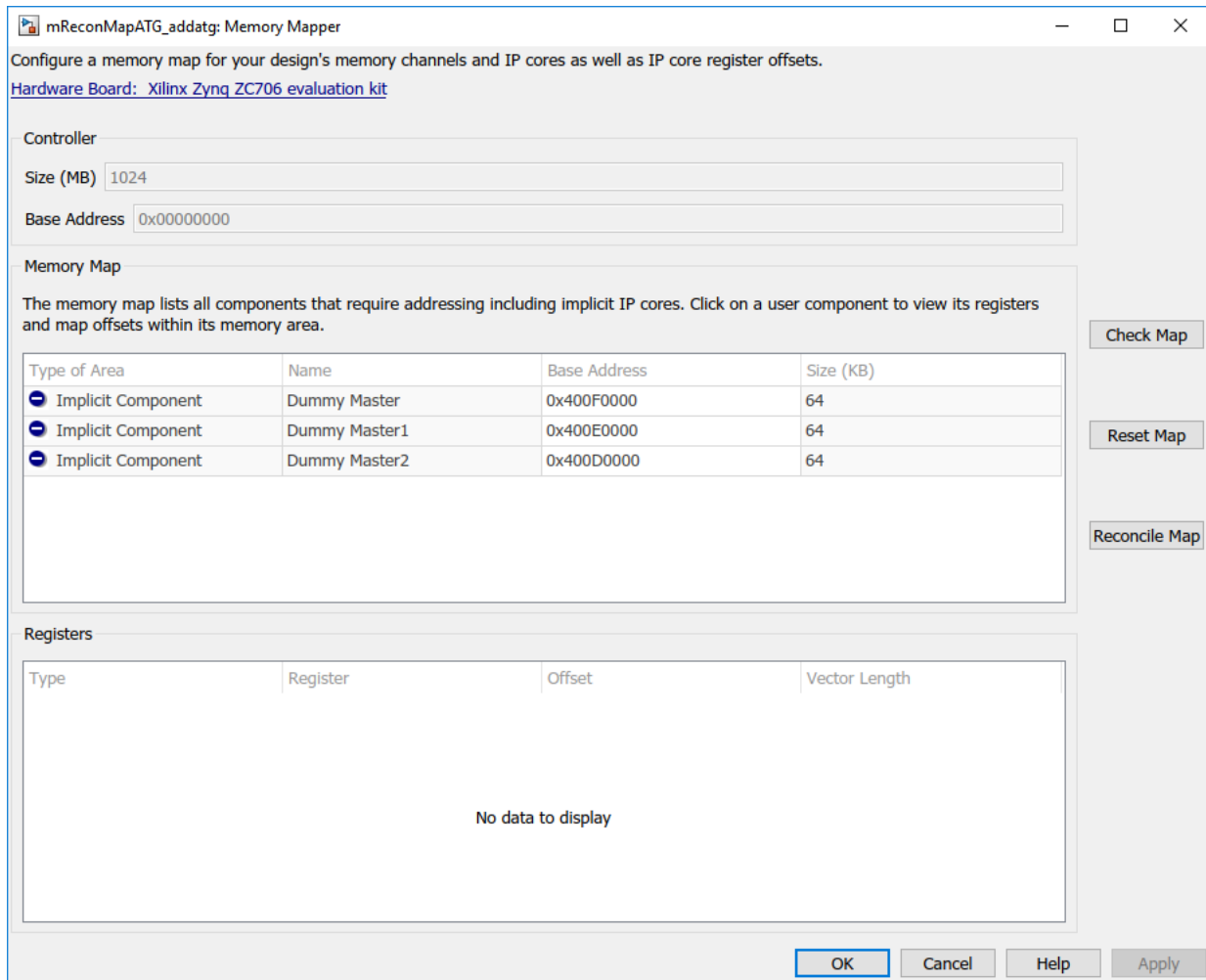
The **Memory Mapper** lists the three masters in the design. Edit their base addresses as per your requirements. Add another channel to your model.
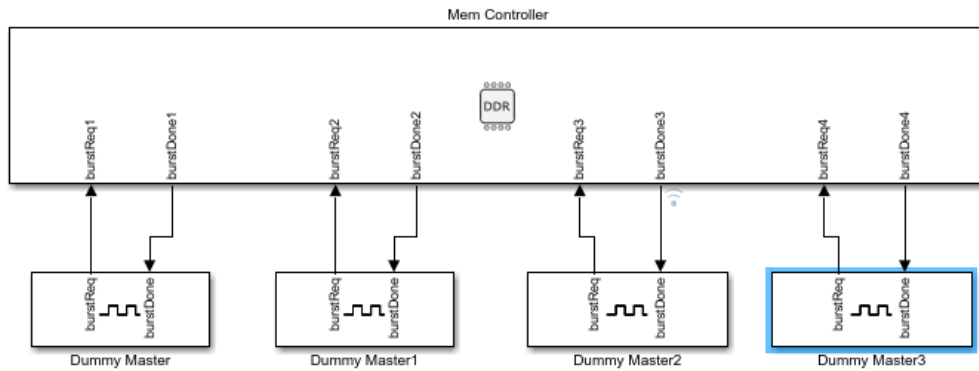
The model consists of four memory channels, while the **Memory Map** section shows only three. To resolve this conflict, click **Reconcile Map**. This adds another line, which represents the added channel, to the memory map table.

mReconMapATG_addatg: Memory Mapper

Configure a memory map for your design's memory channels and IP cores as well as IP core register offsets.

Hardware Board: Xilinx Zynq ZC706 evaluation kit

Controller

Size (MB)	1024

Base Address	0x00000000

Memory Map

The memory map lists all components that require addressing including implicit IP cores. Click on a user component to view its registers and map offsets within its memory area.

Type of Area	Name	Base Address	Size (KB)
⊖ Implicit Component	Dummy Master	0x400F0000	64
⊖ Implicit Component	Dummy Master1	0x400E0000	64
⊖ Implicit Component	Dummy Master2	0x400D0000	64
⊖ Implicit Component	Dummy Master3	0x40100000	64

Check Map

Reset Map

Reconcile Map

Registers

Type	Register	Offset	Vector Length

No data to display

OK   Cancel   Help   Apply

Reset Map

Click **Reset Map** to create a new, autogenerated map. The base addresses of the channels are reset to a default value.

| mReconMapATG_addatg: Memory Mapper | — □ × |

Configure a memory map for your design's memory channels and IP cores as well as IP core register offsets.

Hardware Board: Xilinx Zynq ZC706 evaluation kit

Controller

| Size (MB) | 1024 |
| Base Address | 0x00000000 |

Memory Map

The memory map lists all components that require addressing including implicit IP cores. Click on a user component to view its registers and map offsets within its memory area.

[Check Map]

Type of Area	Name	Base Address	Size (KB)
⊖ Implicit Component	Dummy Master	0x40010000	64
⊖ Implicit Component	Dummy Master1	0x40020000	64
⊖ Implicit Component	Dummy Master2	0x40030000	64
⊖ Implicit Component	Dummy Master3	0x40040000	64

[Reset Map]

[Reconcile Map]

Registers

Type	Register	Offset	Vector Length

No data to display

[OK] [Cancel] [Help] [Apply]

Parameters

Hardware Board — View selected hardware board
selected hardware board

This property is read-only.

This Parameter shows the targeted hardware board. Click the link to open the configuration parameters on the **Hardware Implementation** pane, and change any of the hardware configurations. To learn more about board configuration parameters, see **Hardware Implementation Pane Overview**.

Controller

Size (MB) — External memory size in megabytes
positive integer

This property is read-only.

This parameter shows the size of the external memory available for the selected hardware board in megabytes. This value is derived from the hardware board selected in the configuration parameters.

Base Address — Base address of memory
0x00000000 (default) | 32-bit hexadecimal address

This property is read-only.

This parameter shows the base address of the external memory. This value is a 32-bit hexadecimal value.

Memory Map

Check Map — Check memory map
button

Check that the memory map has no overlapping regions or registers, and that memory addresses are properly aligned.

Reset Map — Reset memory map
button

Reset the memory map to its initial values.

Reconcile Map — Reconcile memory map with existing model
button

Reconcile the memory map with the existing model. After adding or deleting a channel or a memory-mapped register to your model, click this button to synchronize between the

model and the memory map. To verify that the reconciled memory map is valid, click **Check Map** after reconciling.

Note Clicking **Reconcile Map** matches the memory map to the model but does not reset the base address values of the memory areas.

See Also

Memory Channel | Memory Controller | **SoC Builder**

Topics

"Random Access of External Memory"

Introduced in R2019a

SoC Builder

Build, load, and execute SoC model on SoC-FPGA board

Description

The **SoC Builder** tool steps through the various stages for building and executing an SoC model on an FPGA board.

Using this tool, you can:

- Review the model information provided to the tool.
- Review the memory map and edit it if needed.
- Set up a folder to store all generated files.
- Choose between different build actions.
- Validate that the model has all required components for generating a programming file.
- Build the model using Xilinx Vivado® or Intel Quartus® tool families.
- Configure the Ethernet connectivity.
- Load the programming file to your FPGA board.
- Run the application.

Open the SoC Builder

- Simulink Toolstrip: On the **System on Chip** tab, click **Configure, Build & Deploy**.
- Simulink Toolstrip: click the **System on Chip** tab, and then press **Ctrl+B**.
- MATLAB command prompt: Enter `socBuilder('modelname')`.

Note If the **System on Chip** tab is not visible, on the **Apps** tab, under **Setup to Run on Hardware** click the **System on Chip (SoC)** app icon.

Examples

- "Generate SoC Design"

Programmatic Use

`socBuilder('modelname')` opens SoC Builder and loads the specified model into the tool.

See Also

Memory Mapper

Topics

"Generate SoC Design"

Introduced in R2019a