

**SoC Blockset™**

Reference



**MATLAB® & SIMULINK®**

R2020b



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*SoC Blockset™ Reference*

© COPYRIGHT 2019–2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

March 2019	Online Only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)
March 2020	Online only	Revised for Version 1.2 (Release 2020a)
September 2020	Online only	Revised for Version 1.3 (Release 2020b)

## Blocks

1

## Configuration Parameters

2

<b>Hardware Implementation Pane</b> .....	<b>2-2</b>
Hardware Implementation Pane Overview .....	2-2
Hardware board settings .....	2-2
Design Mapping .....	2-2
Task profiling in simulation .....	2-3
Task profiling on processor .....	2-3
Operating system/scheduler .....	2-3
Task and memory simulation .....	2-4
Board Parameters .....	2-4
Processor .....	2-4
Board Options .....	2-4
Clocking .....	2-4
External Mode .....	2-5
FPGA design (top-level) .....	2-5
FPGA design (mem controllers) .....	2-5
FPGA design (mem channels) .....	2-6
FPGA design (debug) .....	2-6
<b>Hardware Board Settings</b> .....	<b>2-8</b>
Processing Unit .....	2-8
<b>Design Mapping</b> .....	<b>2-9</b>
View/Edit Task Map .....	2-9
View/Edit Peripheral Map .....	2-9
<b>Task Profiling in Simulation</b> .....	<b>2-10</b>
Show in SDI .....	2-10
Save to file .....	2-10
Overwrite file .....	2-10
<b>Task Profiling on Processor</b> .....	<b>2-11</b>
Show in SDI .....	2-11
Save to file .....	2-11
Overwrite file .....	2-11
Instrumentation .....	2-11
Profiling duration .....	2-11

<b>Kernel latency</b> .....	<b>2-13</b>
Settings .....	<b>2-13</b>
<b>Task and Memory Simulation</b> .....	<b>2-14</b>
Set seed for simulating task duration and memory access .....	<b>2-14</b>
Seed Value .....	<b>2-14</b>
Cache input data at task start .....	<b>2-14</b>
<b>Processor</b> .....	<b>2-15</b>
Number of cores .....	<b>2-15</b>
<b>Clocking</b> .....	<b>2-16</b>
CPU Clock (MHz) .....	<b>2-16</b>
<b>Build Action</b> .....	<b>2-17</b>
Settings .....	<b>2-17</b>
<b>External Mode</b> .....	<b>2-18</b>
Communication Interface .....	<b>2-18</b>
Run external mode in a background thread .....	<b>2-18</b>
Port .....	<b>2-18</b>
Verbose .....	<b>2-19</b>
<b>FPGA design (top-level)</b> .....	<b>2-20</b>
View/Edit Memory Map .....	<b>2-20</b>
Include a JTAG master for host-based interaction .....	<b>2-20</b>
Include processing system .....	<b>2-20</b>
Interrupt latency (s) .....	<b>2-20</b>
Register configuration clock frequency (MHz) .....	<b>2-20</b>
IP core clock frequency (MHz) .....	<b>2-20</b>
<b>FPGA design (mem controllers)</b> .....	<b>2-22</b>
Controller clock frequency (MHz) .....	<b>2-22</b>
Controller data width (bits) .....	<b>2-22</b>
Bandwidth derating (%) .....	<b>2-22</b>
First write transfer latency (clocks) .....	<b>2-22</b>
Last write transfer latency (clocks) .....	<b>2-23</b>
First read transfer latency (clocks) .....	<b>2-23</b>
Last read transfer latency (clocks) .....	<b>2-23</b>
<b>FPGA design (mem channels)</b> .....	<b>2-24</b>
Interconnect clock frequency (MHz) .....	<b>2-24</b>
Interconnect data width (bits) .....	<b>2-24</b>
Interconnect FIFO depth (num bursts) .....	<b>2-24</b>
Interconnect almost-full depth .....	<b>2-24</b>
<b>FPGA design (debug)</b> .....	<b>2-25</b>
Memory channel diagnostic level .....	<b>2-25</b>
Include AXI interconnect monitor .....	<b>2-25</b>
Trace capture depth .....	<b>2-25</b>

**Functions**

**3**

**Objects**

**4**

**Tools**

**5**



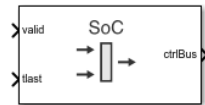
# Blocks

---

# 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 <b>Control protocol</b> to Data stream and <b>Control type</b> to Ready.	<b>ready</b>
	Write data stream	Set <b>Control protocol</b> to Data stream and <b>Control type</b> to Valid.	<b>valid</b> <b>tlast</b>
Pixel stream	Read video stream	Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready.	<b>ready</b>
	Write video stream	Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Valid.	<b>hStart</b> <b>hEnd</b> <b>vStart</b> <b>vEnd</b> <b>valid</b>
	Read video stream with frame sync	Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready frame with sync.	<b>ready</b> <b>fsync</b>
Random access read	Read data	Set <b>Control protocol</b> to Random access read and <b>Control type</b> to Ready.	<b>rd_addr</b> <b>rd_len</b> <b>rd_avalid</b> <b>rd_dready</b>



Protocol Interface Type	Mode of Operation	Parameter Configuration	Enabled Input Ports
Random access write	Write data	Set <b>Control protocol</b> to Random access write and <b>Control type</b> to Valid.	<b>wr_addr</b> <b>wr_len</b> <b>wr_valid</b>

## 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\_valid — 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 <b>Control protocol</b> to Data stream and <b>Control type</b> to Ready.	StreamS2MBusObj
Set <b>Control protocol</b> to Data stream and <b>Control type</b> to Valid.	StreamM2SBusObj
Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready.	StreamVideoS2MBusObj
Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Valid.	pixelcontrol
Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready frame with sync.	StreamvideoFsyncS2MBusObj
Set <b>Control protocol</b> to Random access read and <b>Control type</b> to Ready.	ReadControlM2SBusObj
Set <b>Control protocol</b> to Random access write and <b>Control type</b> 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**

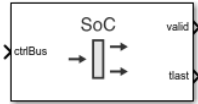
SoC Bus Selector

## **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 <b>Control protocol</b> to Data stream and <b>Control type</b> to Valid.	<b>valid</b> <b>tlast</b>
	Write stream data	Set <b>Control protocol</b> to Data stream and <b>Control type</b> to Ready.	<b>ready</b>
Pixel stream	Read video stream	Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Valid.	<b>hStart</b> <b>hEnd</b> <b>vStart</b> <b>vEnd</b> <b>valid</b>
	Write video stream	Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready.	<b>ready</b>
Random access read	Read data	Set <b>Control protocol</b> to Random access read and <b>Control type</b> to Valid.	<b>rd_aredy</b> <b>rd_dvalid</b>
Random access write	Write data	Set <b>Control protocol</b> to Random access write and <b>Control type</b> to Ready.	<b>wr_ready</b> <b>wr_bvalid</b> <b>wr_complete</b>

## 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 <b>Control protocol</b> to Data stream and <b>Control type</b> to Valid.	StreamM2SBusObj
Set <b>Control protocol</b> to Data stream and <b>Control type</b> to Ready.	StreamS2MBusObj
Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Valid.	pixelcontrol
Set <b>Control protocol</b> to Pixel stream and <b>Control type</b> to Ready.	StreamVideoS2MBusObj
Set <b>Control protocol</b> to Random access read and <b>Control type</b> to Valid.	ReadControls2MBusObj
Set <b>Control protocol</b> to Random access write and <b>Control type</b> 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\_aredy — 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 1-8.

### **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 1-8.

## 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

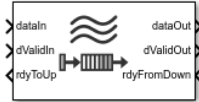
SoC Bus Creator

## 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` | `uint64` | `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` | `uint64` | `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**

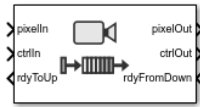
Video Stream FIFO

**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` | `uint64` | `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` | `uint64` | `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**

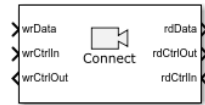
Stream FIFO

**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` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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**

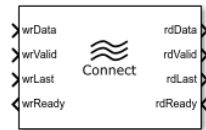
Stream Connector

## **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` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `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

##### **DSInx — 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 **DSInx**, 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

##### **DSx — Output signal**

Boolean scalar

Output signal that returns the state of the switch. Each DIP switch has a port, named **DS1** to **DSx**, 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<sub>n</sub> — Selected DIP switches

Off (default) | On

To enable the *n*th DIP switch port, select `On` for the **DS<sub>n</sub>** 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**

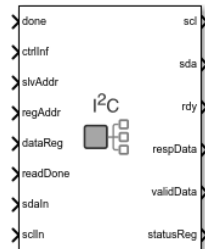
LED | Push Button

**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
- Support 7-bit and 10-bit address I2C slave devices
- Burst mode support with a maximum burst size of 256 bytes
- Support multiple transmission speed modes
- 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 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**, **sclIn**, **scl**, and **sda** ports. To communicate with a 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.

Register Address	Port and Register Name	Register Size in Bits	Operation Mode
0x100	<b>ctrlInf</b> — Control information	32	Write
0x104	<b>slvAddr</b> — Slave address	32	Write
0x108	<b>regAddr</b> — Register address	32	Write
0x10C	<b>dataReg</b> — Data register	32	Write

Register Address	Port and Register Name	Register Size in Bits	Operation Mode
0x110	<b>readDone</b> — Read done register	32	Write
0x114	<b>done</b> — Done register	32	Write
0x118	<b>rdy</b> — Ready register	32	Read
0x11C	<b>respData</b> — First response data register	32	Read
0x120	<b>validData</b> — Response data valid register	32	Read
0x124	<b>statusReg</b> — Status register	32	Read

To perform read and write operations using the I2C Master block, you need to follow a proper sequence. This section provides information about the sequence flow for read and write operations.

### Read Sequence

To read data from an external slave device:

- 1 Send the **ctrlInf** register information.
- 2 Send the **slvAddr** register information.
- 3 Send the **regAddr** register information.
- 4 Set the **done** register to 1 after sending one set of register information to the block and then set it to 0.
- 5 Read the response data from the external slave device. After reading the data from the **respData** register, set the **readDone** register to 1 and then set it to 0 immediately.
- 6 Set the **readDone** register to 1 again, to read more than 4 bytes of data. After the read operation, set it to 0 immediately.

In read sequence, one set of register information is a combination of **ctrlInf**, **slvAddr**, and **regAddr** registers.

### Write Sequence

To write data to an external slave device:

- 1 Send the **ctrlInf** register information.
- 2 Send the **slvAddr** register information.
- 3 Send the **regAddr** register information.
- 4 Send the **dataReg** register that contains the data to write to the slave device register.
- 5 Set the **done** register to 1 after writing one set of register information to the block, and then set it to 0.
- 6 Set the **done** register to 1 again, to write more than 4 bytes of data. After the write operation, set it to 0 immediately.

In write sequence, one set of register information is a combination of **ctrlInf**, **slvAddr**, **regAddr**, and **dataReg** registers.



## Ports

### Input

#### **ctrlInf – Control information**

scalar

Control information register contains configuration information on how the block communicates with the slave device, specified as a scalar. This register is a combination of read or write operation indication bit, number of bytes of slave-device register address, number of bytes of slave-device data register, and slave device address type bit. 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: <ul style="list-style-type: none"> <li>• One byte (8 bits), set this value to 00</li> <li>• Two bytes (16 bits), set this value to 01</li> <li>• Three bytes (24 bits), set this value to 10</li> <li>• Four bytes (32 bits), set this value to 11</li> </ul>

Bit	Purpose	Value Description
[10:3]	Set the data size of the slave-device register.	<p>If the slave-device register supports:</p> <ul style="list-style-type: none"> <li>• One byte of data, set this value to 00000000</li> <li>• Two bytes of data, set this value to 00000001</li> <li>• Three bytes of data, set this value to 00000010</li> <li>• Four bytes of data, set this value to 00000011</li> <li>• Five bytes of data, set this value to 00000100</li> <li>• Six bytes of data, set this value to 00000101</li> <li>• Seven bytes of data, set this value to 00000110</li> <li>• Eight bytes of data, set this value to 00000111</li> <li>• Nine bytes of data, set this value to 00001000</li> <li>• Ten bytes of data, set this value to 00001001</li> <li>• Eleven bytes of data, set this value to 00001010</li> <li>• Twelve bytes of data, set this value to 00001011</li> <li>• Thirteen bytes of data, set this value to 00001100</li> <li>• Fourteen bytes of data, set this value to 00001101</li> <li>• Fifteen bytes of data, set this value to 00001110</li> <li>• Sixteen bytes of data, set this value to 00001111</li> </ul> <p>.....</p> <ul style="list-style-type: none"> <li>• 256 bytes of data, set this value to 11111111</li> </ul>
11	Set the slave device type	To configure 7-bit address slave device, set this value to 0. To configure 10-bit address slave device, set this value to 1.

Data Types: uint16

**sLvAddr — Slave address**

scalar

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

Data Types: uint16

**regAddr — Register address**

scalar

Register address of the slave device, specified as a scalar.

Data Types: uint32

**dataReg — Data register**

scalar

Data register, specified as a scalar. The block uses this port to write data to the slave-device register.

Data Types: uint32

**readDone — Read done signal**

Boolean scalar

Read done signal, specified as a Boolean scalar. When this value is 1 (true), the user is ready to read the response data from the block that is received from the slave device. 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.

Data Types: Boolean

**sdaIn — Input serial data**

Boolean scalar

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

Data Types: Boolean

**sclIn — Input serial clock**

Boolean scalar

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

Data Types: Boolean

**Output****scl — Output serial clock**

Boolean scalar

Output serial clock, specified 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, specified 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 — Response data register**

scalar

Response data register containing the data from the slave-device register, returned as a scalar.

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

**statusReg — I2C bus status indicator**

scalar

Indicates the status of the I2C bus, returned as a scalar.

Bit	Purpose	Value Description
[7:4]	Reserved	Reserved
3	Indicates the status of the I2C bus.	When this value is 1, it indicates that the I2C bus is busy. When this value is 0, it indicates that the I2C bus is idle and ready for configuration.
2	Indicates the acknowledgment status from the slave device to the I2C Master.	When this value is 1, it indicates that the slave device has not acknowledged the I2C Master. When this value is 0, it indicates that the slave device has acknowledged the I2C Master.
[1:0]	Reserved	Reserved

Data Types: uint32

**Parameters**

**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

## Compatibility Considerations

### **I2C Master block has one data register input port and one data register output port**

*Behavior changed in R2020a*

In R2019a, the I2C Master block has input data register ports **dataReg**, **dataReg1**, **dataReg2**, and **dataReg3** and output response data register ports **respData**, **respData1**, **respData2**, and **respData3**. These data register ports support a maximum of 16 bytes per transaction. In R2020a, these register ports are replaced with single data register ports: input port **dataReg** and output port **respData**. Each of these single data register ports support a maximum of 256 bytes per transaction.

In R2020a, Simulink® errors if you open a model that was created in an earlier release and that contains an I2C Master block. In this case, connections to ports **dataReg1**, **dataReg2**, **dataReg3**, **respData1**, **respData2**, and **respData3** are either missing or reconnected to empty ports on the block. Manually check and update the port connections in your model to proceed further.

## 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™.

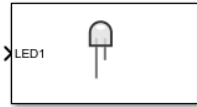
## See Also

**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

#### **PBInx — 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 **PBInx**, 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

#### **PBx — Output signal**

Boolean scalar

Output signal that returns the state of the push button. Each push button has a port, named **PB1** to **PBx**, 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<sub>*n*</sub> — Selected push buttons

Off (default) | On

To enable the *n*th push-button port, select `On` for the **PB<sub>*n*</sub>** 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**

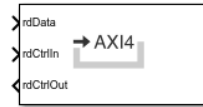
DIP Switch | LED

**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 | uint64 | 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\_aredy — 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` | `int64` | `uint16` | `uint32` | `uint64` | `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”.

### 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**

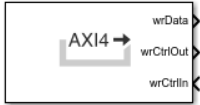
AXI4 Master Source

**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` | `uint64` | `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 | int64 | uint16 | uint32 | uint64 | 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”.

### 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

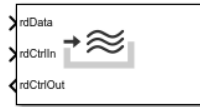
**Introduced in R2019a**



# 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 | uint64 | 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` | `uint64` | `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 | int64 | uint16 | uint32 | uint64 | 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”.

### 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

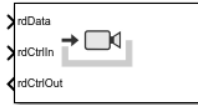
Stream Data Sink

### Introduced in R2019a

# 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**

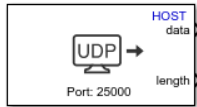
Video Test Sink

**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 is 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

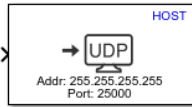
UDP Write (HOST)

**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 is 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**

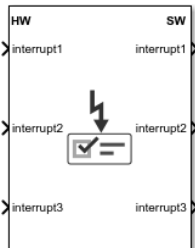
UDP Read (HOST)

**Introduced in R2019a**

## Interrupt Channel

Send interrupt to processor from hardware

**Library:** SoC Blockset / Memory



### Description

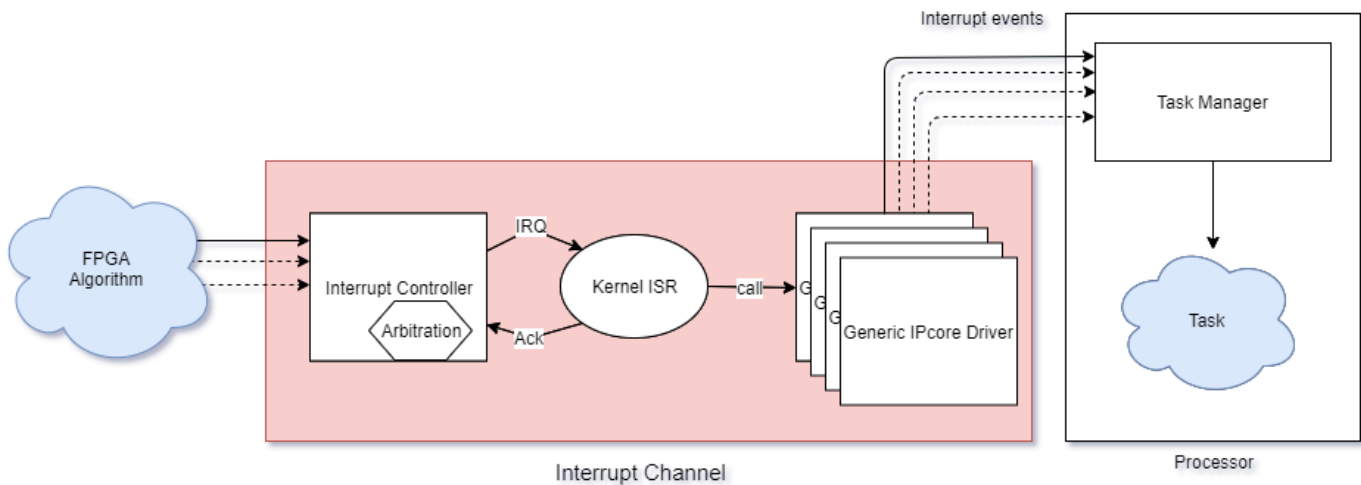
The Interrupt Channel block receives interrupt requests from FPGA logic or the Memory Channel block, arbitrates the requests, and triggers an event-driven software task to the Task Manager block. You can connect up to 16 devices to the interrupt controller, with one interrupt per device. The block consists of these three parts:

- Interrupt Controller - This part accepts interrupt requests (IRQs) and arbitrates them according to a user-specified priority. When concurrent requests to the interrupt controller exist, requests with a higher priority are processed before those with a lower priority.
- Kernel Interrupt Service Routine (ISR) - This part receives an interrupt request from the Interrupt Controller, serves the interrupt, and sends an acknowledge signal back to the Interrupt Controller, so that it can process the next IRQ.
- IPCore Driver (one per interrupt) - This part receives a request from the Kernel ISR and triggers an event-driven task in the processor.

Even though the interrupt channel can have more than one interrupt output toward the processor, it sends no more than one active interrupt event to the processor at any given time.

This image shows a conceptual view of an Interrupt Channel block, that accepts interrupt requests from an FPGA algorithm. After arbitration, the kernel serves the request and triggers an event to a processor algorithm.





## Ports

### Input

#### *interruptN* – Interrupt request from hardware

True | False

Each interrupt is assigned a port pair: one input port and one output port. By default, the *N*th interrupt port is named *interruptN*. You can change interrupt names by clicking **Edit** in the **Interrupts** parameter.

Connect this port to a Boolean signal from the FPGA logic or an event from a Memory Channel or Event Source block.

### Dependencies

The number of input ports depends on the number of interrupts in the interrupt table.

Data Types: `rteEvent` | `Boolean`

### Output

#### *interruptN* – Interrupt request signal to processor

scalar

Each interrupt is assigned a port pair: one input port and one output port. By default, the *N*th interrupt port is named *interruptN*. You can change interrupt names by clicking **Edit** in the **Interrupts** parameter.

Connect this port to a task event input port in the Task Manager block.

### Dependencies

The number of output ports depends on the number of interrupts in the interrupt table.

Data Types: `rteEvent`

## Parameters

### Interrupts – Interrupt name, trigger type, and priority

table

This parameter includes a table, where each of its lines corresponds to an interrupt in the Interrupt Channel block. Edit the table to add or edit an interrupt. The interrupt channel can have up to 16 interrupts.

For each interrupt, you can edit these values.

- **Interrupt Name** - Specify the interrupt name. This value changes the input and output port names for this interrupt.
- **Trigger Type** - Select the trigger type for the interrupt by choosing either of these options.
  - **Rising edge** - When the interrupt originates in FPGA logic
  - **SoC event** - When the interrupt originates in the Memory Channel block or Event Source block
- **Priority** - Set the priority for each interrupt is set in the **Priority** column. This value remains static. The top row represents the highest interrupt. Click **Move Up** to increase the priority of an interrupt. Click **Move Down** to decrease the priority of an interrupt.

### Interrupt processing time – Processing time for interrupt access

100e-6 (default) | positive scalar

This sample time represents the time required for the interrupt channel to arbitrate and execute an interrupt request. It is defined as the time required for the Interrupt Controller arbitration, Kernel ISR execution, and additional delay for the device driver.

Specify the processing time by entering a number, in seconds.

## 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

### Blocks

Event Source | Memory Channel | Task Manager

### Introduced in R2020b

# IP Core Register Read

Model register writes from software to hardware

**Library:** SoC Blockset / Memory



## Description

The IP Core Register Read block models a write operation from a processor to hardware logic. The block receives data sent with a Register Write block from the processor. You can define the register offset in the **Memory Mapper** tool.

## Ports

### Output

#### **data** — Data output

vector

This port outputs the data vector received from the processor, starting at the offset address from the base address of the IP core. Set the offset address in the **Memory Mapper** tool.

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

## Parameters

### **Register name** — Name of register

RegA (default) | character vector

Match this name to the **Register name** parameter specified in the Register Write block.

Example: AddressReg1

### **Output data type** — Data type of output data

uint16 (default) | single | int8 | uint8 | int16 | int32 | uint32 | boolean | fixed point  
data type

Select the data type for the output data. This value must match the value selected for the Register Write block.

### **Output vector size** — Vector size of output data

1 (default) | positive integer

Specify the vector size of the output data as a positive integer. This value must match the value selected for the Register Write block.

### **Sample time** — Simulation interval of sampling

-1 (default) | nonnegative scalar

Specify a discrete time interval, in seconds, at which the block outputs data. If this value is -1 (default), the sample time is inherited from the model.

## **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**

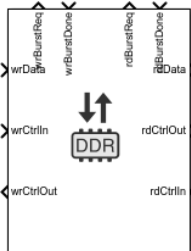
Register Channel | Register Write

## **Introduced in R2020a**

# Memory Channel

Stream data through a memory channel

**Library:** SoC Blockset / Memory



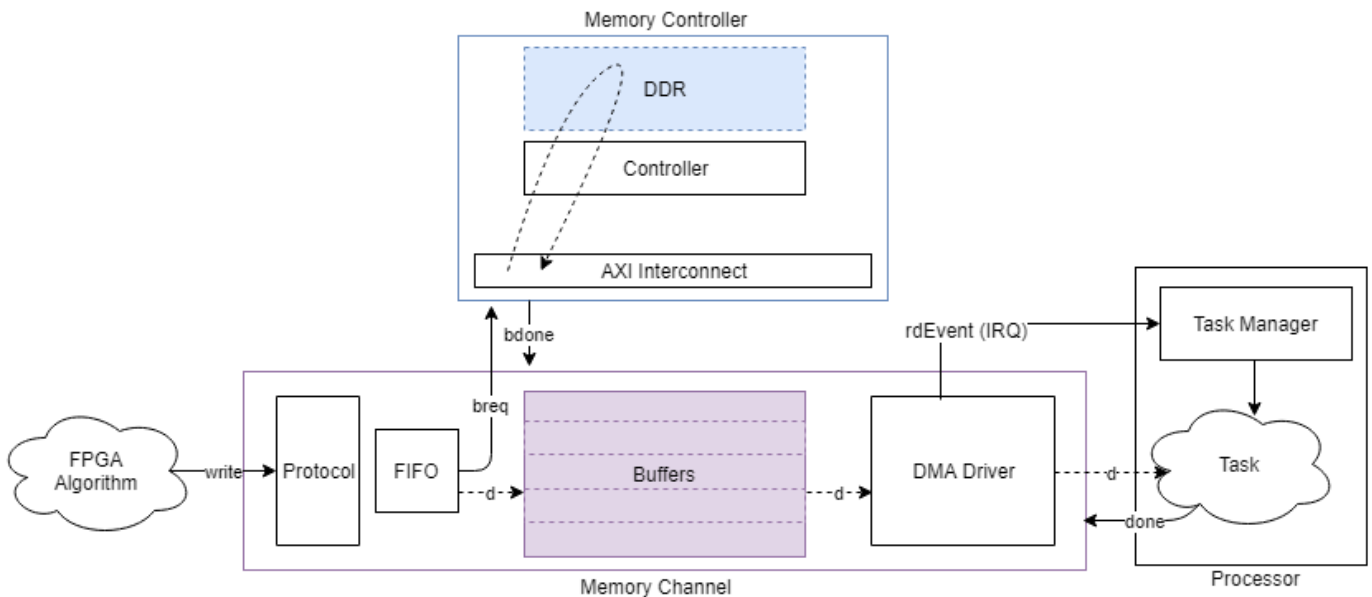
## Description

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:

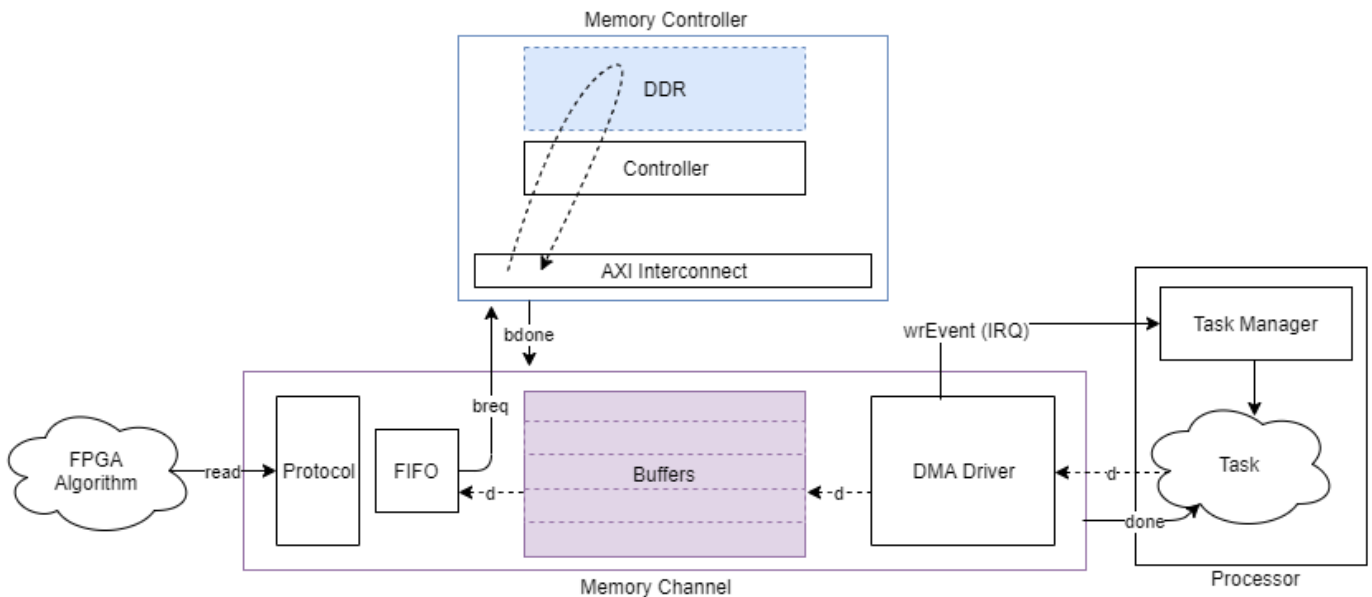
- **AXI4-Stream to Software via DMA** - Model a connection between hardware logic and a software task through external memory. The writer puts data into the channel using a MathWorks® simplified AXI stream protocol and the reader (processor) gets data from a DMA driver interface. The channel models the 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.



- **Software to AXI4-Stream via DMA** - Model a connection between hardware logic and a software task through external memory. The writer (processor) streams data into the channel via a DMA driver using a MathWorks simplified AXI stream protocol. 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 from the software task. For more information about the MathWorks simplified AXI stream protocol, see "AXI4-Stream Interface".

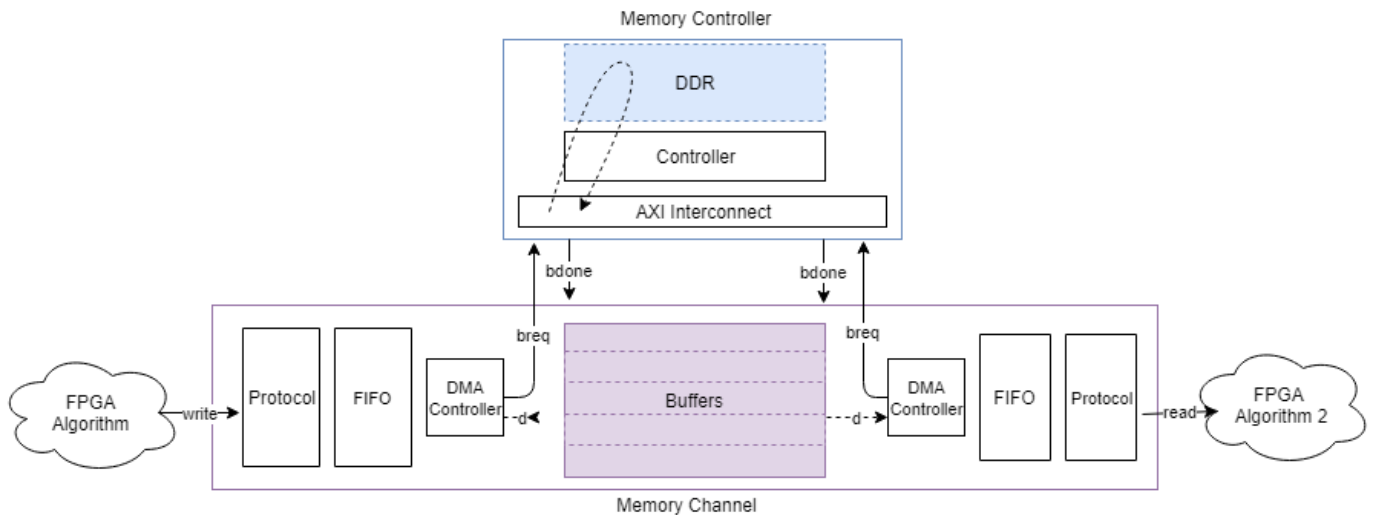
This image is a conceptual view of a Memory Channel block, streaming data from a processor algorithm to an FPGA algorithm.



- **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.

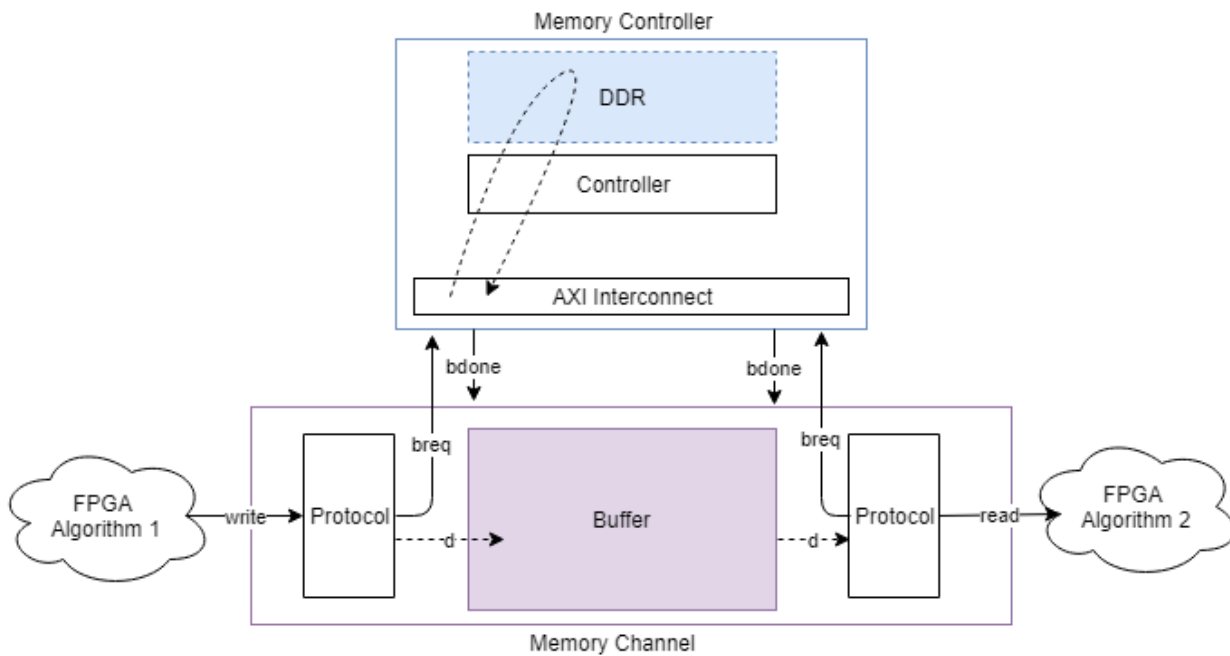
The channel behaves as a first in first out (FIFO) memory. The channel models the datapath of the connection. The Memory Channel block includes an intermediate burst-level FIFO, DMA engine, interconnect, and external memory. The external memory itself is managed as a circular buffer, where a buffer must 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.



- **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 | int64 | uint8 | uint16 | uint32 | uint64 | 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



**Dependencies**

To enable this port, set the **Channel type** parameter to a value other than `Software to AXI4-Stream via DMA`.

Data Types: `StreamM2SBusObj | pixelcontrol | WritecontrolM2SBusObj`

**rdCtrlIn — Reader input control signal**

bus

This port accepts a bus from a data consumer block, signaling that the consumer block is ready to accept read data. For streaming protocols, the **rdCtrlIn** port is a backpressure signal from a data consumer to the Memory Channel block. For the AXI4 Random Access protocol, this input 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
Software to AXI4-Stream via DMA	StreamS2MBusObj
AXI4 Stream FIFO	StreamS2MBusObj
AXI4 Stream Video FIFO	StreamVideoS2MBusObj
AXI4 Stream Video Frame Buffer	StreamVideoFSyncS2MBusObj
AXI4 Random Access	ReadControlM2SBusObj

**Dependencies**

To enable this port, set the **Channel type** parameter to a value other than `AXI4-Stream to Software via DMA`.

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

**rdDone — Notification message of completed read**

scalar

This message port receives a notification from the connected Stream Read block. The notification indicates that a read transaction completed. For more information on messages, see “Messages”.

**Dependencies**

To enable this port, set the **Channel type** parameter to `AXI4-Stream to Software via DMA`.

Data Types: `Boolean`

**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 “Messages”.

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 “Messages”.

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 “Messages”.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `SoCData`

#### **rdEvent — Task read event signal**

scalar

This port sends a task event signal that triggers the Task Manager block to execute the associated event-driven read.

#### **Dependencies**

To enable this port, set the **Channel type** parameter to `AXI4-Stream to Software via DMA`.

Data Types: `rteEvent`

#### **wrEvent — Task write event signal**

scalar

This port sends a task event signal that triggers the Task Manager block to execute the associated event-driven write.

#### **Dependencies**

To enable this port, set the **Channel type** parameter to `Software to AXI4-Stream via DMA`.

Data Types: `rteEvent`

#### **wrDone — Notification of freed buffer in memory**

scalar

This message port sends a notification to the connected Stream Write block. This notification indicates that a read transaction completed, and that a buffer in memory is available for writing.

#### **Dependencies**

To enable this port, set the **Channel type** parameter to `Software to AXI4-Stream via DMA`.

Data Types: `Boolean`

#### **rdCtrlOut — Reader control signal from memory channel to data consumer**

bus

Control signal from channel to data consumer. The contents of this signal 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
Software to AXI4-Stream via DMA	StreamM2SBusObj
AXI4 Stream FIFO	StreamM2SBusObj
AXI4 Stream Video FIFO	pixelcontrol
AXI4 Stream Video Frame Buffer	pixelcontrol
AXI4 Random Access	ReadControlS2MBusObj

#### Dependencies

To enable this port, set the **Channel type** parameter to a value other than AXI4-Stream to Software via DMA.

Data Types: StreamM2SBusObj | ReadControlS2MBusObj | pixelcontrol

#### 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

#### Dependencies

To enable this port, set the **Channel type** parameter to a value other than Software to AXI4-Stream via DMA.

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 “Messages”.

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 “Messages”.

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 | Software to AXI4-Stream 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
- Software to AXI4-Stream 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 128 bits.

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) <i>tdata</i> 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	Good	This results in a 36-bit packed pixel which extends to 64-bit <i>tdata</i> . This data is compliant with the current limit of 128-bit <i>tdata</i> .
fixdt(0,48,0)	[1 3]	1024	8192	Bad	This results in a 144-bit packed pixel violates the current limit of 128-bit <i>tdata</i> .
uint8	[120 160 3]	1024	4096	Bad	The scalar data is 24-bit, padded to a 32-bit <i>tdata</i> . 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 <i>tdata</i> . 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.

### Limitations

When you set the **Channel type** parameter to AXI4-Stream to Software via DMA or Software to AXI4-Stream via DMA, the **Number of buffers** parameter must be an integer from 3 to 64.

**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.

**Dependencies**

This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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 2-24.

**Dependencies**

This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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)**

**Dependencies**

This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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**.
- This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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**.
- This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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**.
- This parameter is not visible when **Channel type** is set to AXI4 Random Access.

**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 | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | 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 | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | 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.

#### Sample time — Time interval of sampling

1 (default) | positive scalar

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

**Dependencies**

To enable this parameter, clear the **Output data signal matches input** check box, and set **Channel type** as AXI4 Random Access.

**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

Interrupt Channel | Memory Controller | Memory Traffic Generator

### Topics

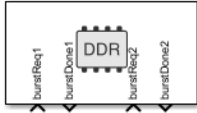
“External Memory Channel Protocols”

### Introduced in R2019a

# Memory Controller

Arbitrate memory transactions for one or more Memory Channel blocks

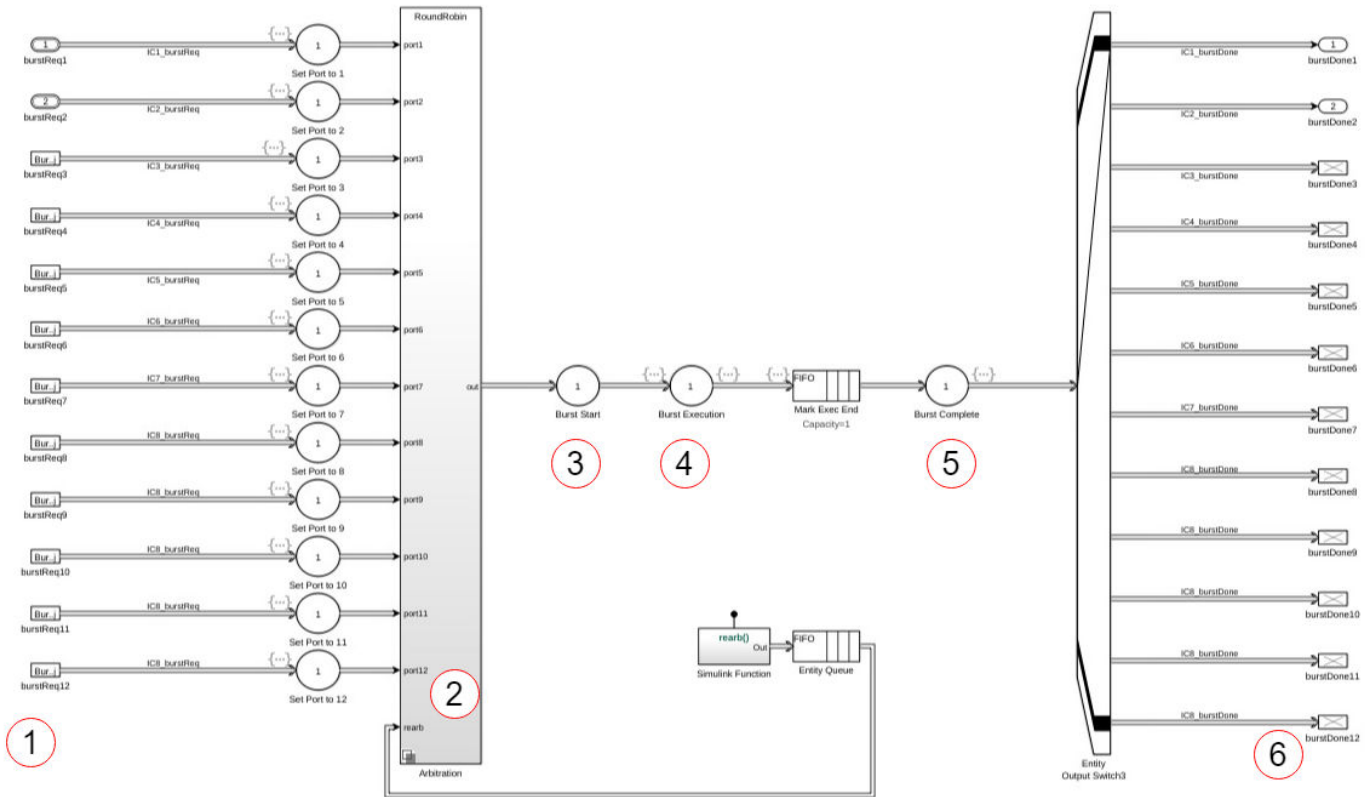
**Library:** SoC Blockset / Memory



## 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 2-22 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”.

## Limitations

- When **Interconnect arbitration** is set to Round Robin, the model does not support simulation stepping. For more information on simulation stepping, see “Simulation Stepper”.

## 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 “Messages”.

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 “Messages”.

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 2-22.

#### 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”

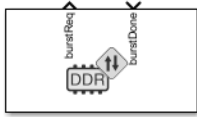
### **Introduced in R2019a**



# Memory Traffic Generator

Generate traffic towards memory controller

**Library:** SoC Blockset / Memory



## 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 Memory Traffic Generator block gains memory access, competes in arbitration, and releases access. The Memory Traffic Generator 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 "Messages".

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 "Messages".

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 the Memory Channel block is configured with **Data width (bits)** set to 64 (8 bytes), then **Burst length** 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

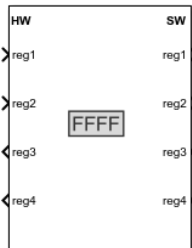
Memory Controller | Memory Channel | Register Channel

### 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 regN. 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 regN. 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 32 registers.

For each register, you can edit these values:

- **Register Name** - 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`
  - `int64`
  - `uint32`
  - `uint64`
  - `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**

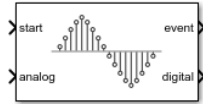
“Memory and Register Data Transfers”

### **Introduced in R2019a**

# ADC Interface

Convert analog signal on ADC input pin to digital signal

**Library:** SoC Blockset / Peripherals



## Description

The ADC Interface block simulates the analog-to-digital conversion (ADC) of a hardware board. The input analog signal gets sampled and converted into a representative digital value. A start event message signals the block to sample the input analog voltage signal. When the conversion completes, the block emits the digital representation of the analog signal and sends an event to a Task Manager block. At this point, a connected task can execute with the new ADC sample.

## Ports

### Input

#### **start** — Start analog to digital conversion

start an analog to digital conversion event

Specify an event signal to start the sampling and measurement of the **analog** input port signal.

Data Types: `rteEvent`

#### **analog** — Analog voltage signal

scalar

Specify an Input analog voltage signal to convert into a digital measurement.

Data Types: `double` | `single`

### Output

#### **digital** — SoC message data

scalar

This port sends the ADC Interface input signal data as a message to the **msg** input port of the ADC Read block.

Data Types: `SoCData`

#### **event** — Task event signal

scalar

This port sends a message at each analog to digital signal conversion event. This output connects to the input of the Task Manager block to execute the associated event-driven task after executing the ADC event.

Data Types: `rteEvent`

## Parameters

### Resolution (bits) – Resolution of digital measurement

12 (default) | 16

An input analog signal can be represented in digital values in the form of 12 or 16 bits. The minimum value of an analog signal that can be represented in 1 bit is called resolution. One bit represents the minimum voltage resolution measurable by the ADC. The minimum voltage resolution can be determined using the following equation:

$$\Delta V_{\min} = \frac{V_{ref}}{2^n}$$

where  $n$  is the **Resolution (bits)** and  $V_{ref}$  is the **Voltage reference (V)** parameter values.

Example: 16

### Voltage reference (V) – Reference voltage in ADC

3 (default) | 3.3

The reference voltage determines the total voltage range that the ADC can convert into a digital value without saturating. Any voltage signal higher than this value produces the maximum possible value that can be represented by the **Resolution (bits)** parameter.

Example: 3.3

### Acquisition time (s) – Time required for ADC to capture input voltage

320e-9 (default) | positive scalar

Specify the time required for the ADC to capture the input voltage during sampling.

Example: 200e-9

### Conversion time (s) – Time to convert physical voltage sample to digital value

240e-9 (default) | positive scalar

Specify the required time to convert the physical voltage sample to the digital representation and output the value.

Example: 20e-9

## See Also

ADC Read | PWM Interface | PWM Write

## Topics

“Get Started with SoC Blocks on MCUs”

“Integrate MCU Scheduling and Peripherals in Motor Control Application”

## External Websites

[https://en.wikipedia.org/wiki/Analog-to-digital\\_converter](https://en.wikipedia.org/wiki/Analog-to-digital_converter)

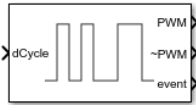
## Introduced in R2020b



# PWM Interface

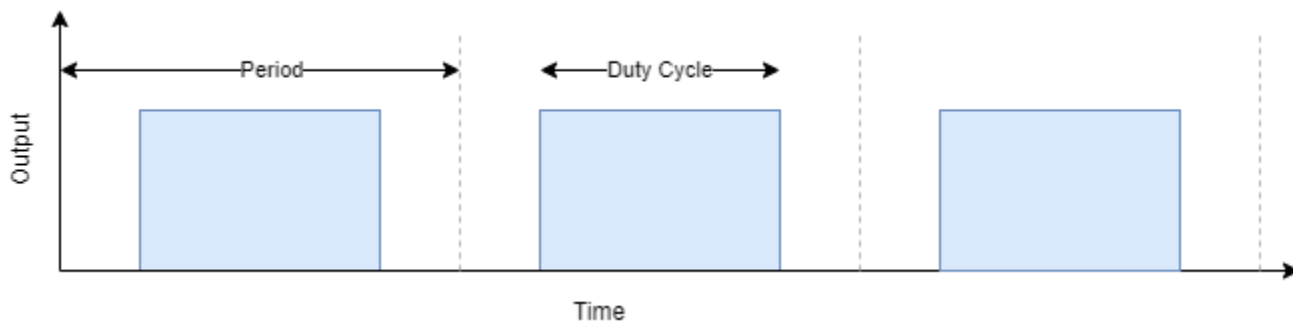
Simulate pulse width modulation (PWM) output from hardware

**Library:** SoC Blockset / Peripherals



## Description

The PWM Interface block simulates the PWM output of a hardware board. This block gets duty cycle data messages from a connected PWM Write block that can either generate a switching pulse-width-modulated waveform or pass the duty cycle value to the output.



## Ports

### Input

#### **dCycle** — Duty cycle

scalar

This port receives data from the output port of the PWM Write block.

Data Types: SoCData

### Output

#### **PWM** — Pulse-width-modulated signal

scalar

This port outputs the pulse-width-modulated rectangular wave defined by the **dCycle** input port.

### Dependencies

To enable this port, set the **Output mode** parameter to Switching.

Data Types: double

**~PWM — Complimentary pulse-width-modulated signal**

scalar

This port outputs the complimentary **PWM** signal.

**Dependencies**

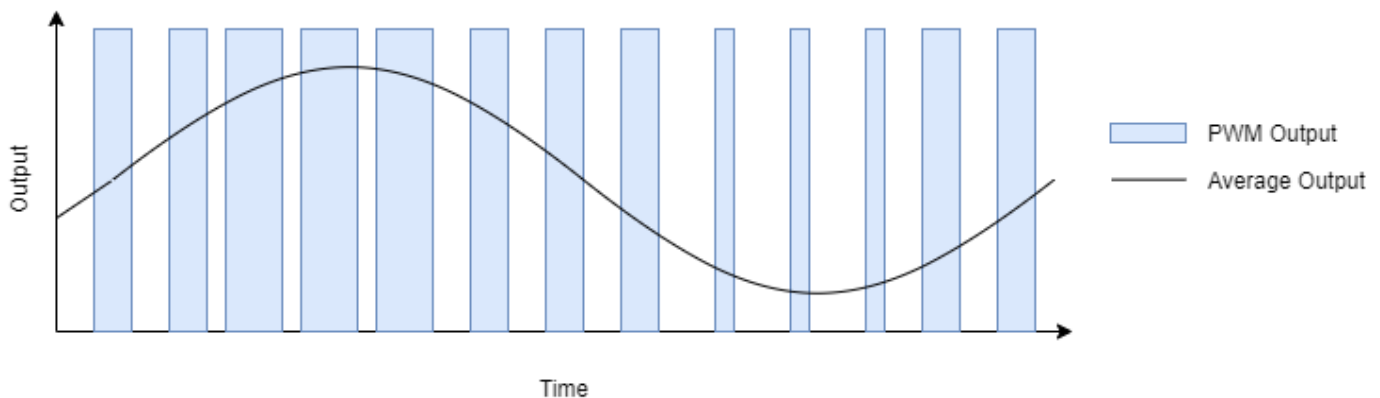
To enable this port, set the **Output mode** parameter to Switching.

Data Types: double

**dCycle — Analog approximation of pulse-width-modulated signal**

scalar

This port emits the averaged value of the PWM waveform, which is a pass-through of the duty cycle input value. This image shows the average output signal equivalent to the PWM output.

**Dependencies**

To enable this port, set the **Output mode** parameter to Average.

Data Types: double

**event — Event emitted on each PWM cycle**

scalar

This port sends a message during each PWM output event that can connect to the **start** port of the ADC Interface block to synchronize ADC and PWM events in closed-loop systems.

Data Types: rteEvent

**Parameters****PWM Period (s) — Period of PWM waveform**

50e-6 (default) | positive scalar

Specify the period of the PWM waveform in seconds.

**Output mode — Output mode**

Switching (default) | Average

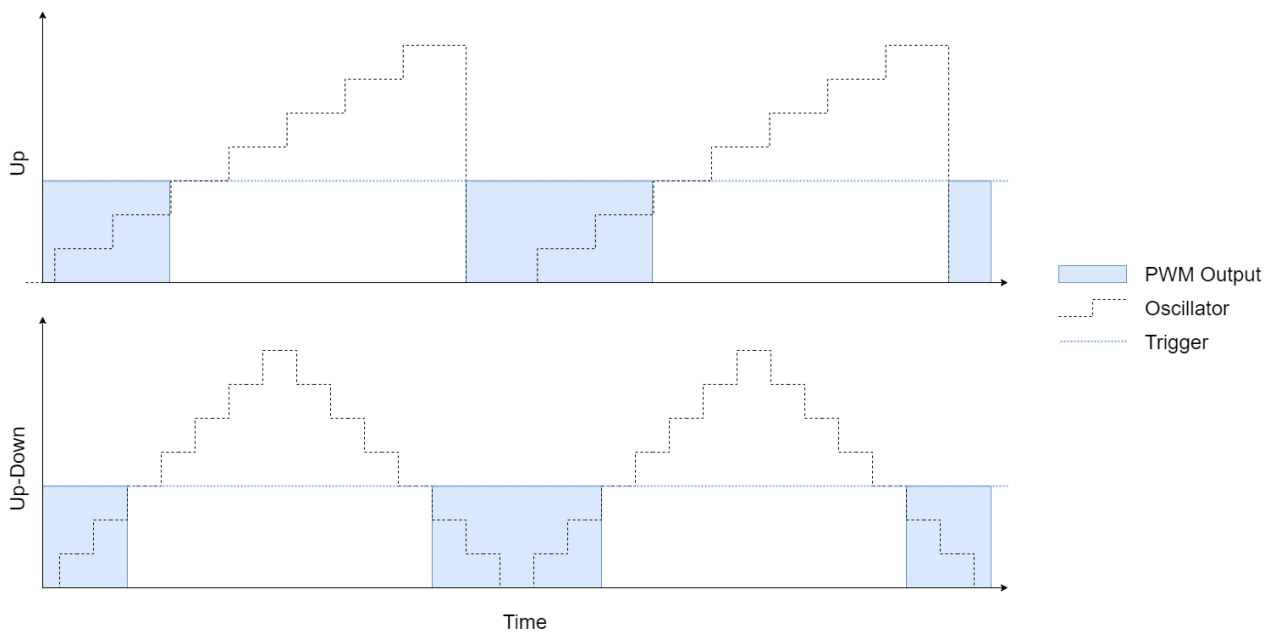
Simulate the output signal as either a true PWM waveform by specifying `Switching` or as the average of the duty cycle by specifying `Average`.

Example: `50e-6`

**Counter mode – Counter waveform**

Up-Down (default) | Up | Down

The counter mode specifies the shape of the underlying sawtooth waveform that drives the PWM output signal inside the PWM module. In `Down` mode, the sawtooth counter counts increments to the maximum and then resets to zero on each period. In `UP` mode, the sawtooth counter decrements to zero then resets to the maximum. In `Up-Down` mode, the sawtooth counter oscillates from zero to the maximum value.



Example: `Up`

**Sampling mode – Sampling mode**

End of PWM period (default) | Mid of PWM period | Mid or End of PWM period

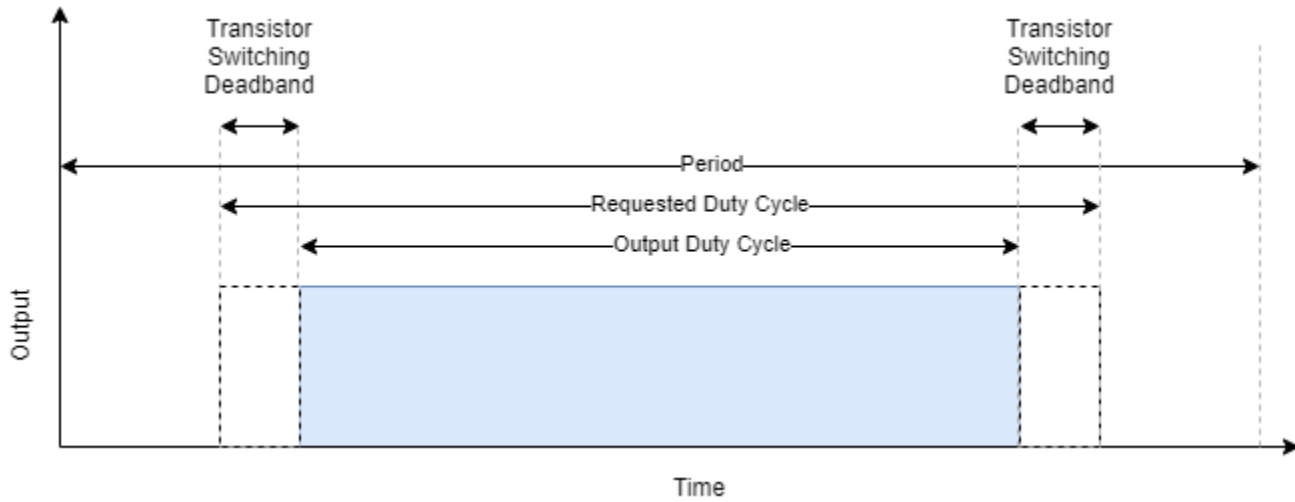
Specify the time at which the input duty cycle is sampled.

Example: `Mid or End of PWM period`

**Dead time (s) – Dead band switching delay**

1e-6 (default) | positive scalar

A time delay is introduced between turning off one of the transistors of a leg of an inverter and turning on the other transistor to ensure that a dead short circuit does not occur. This diagram shows the expected duty cycle and the delay introduced by the transistor switching the dead band.

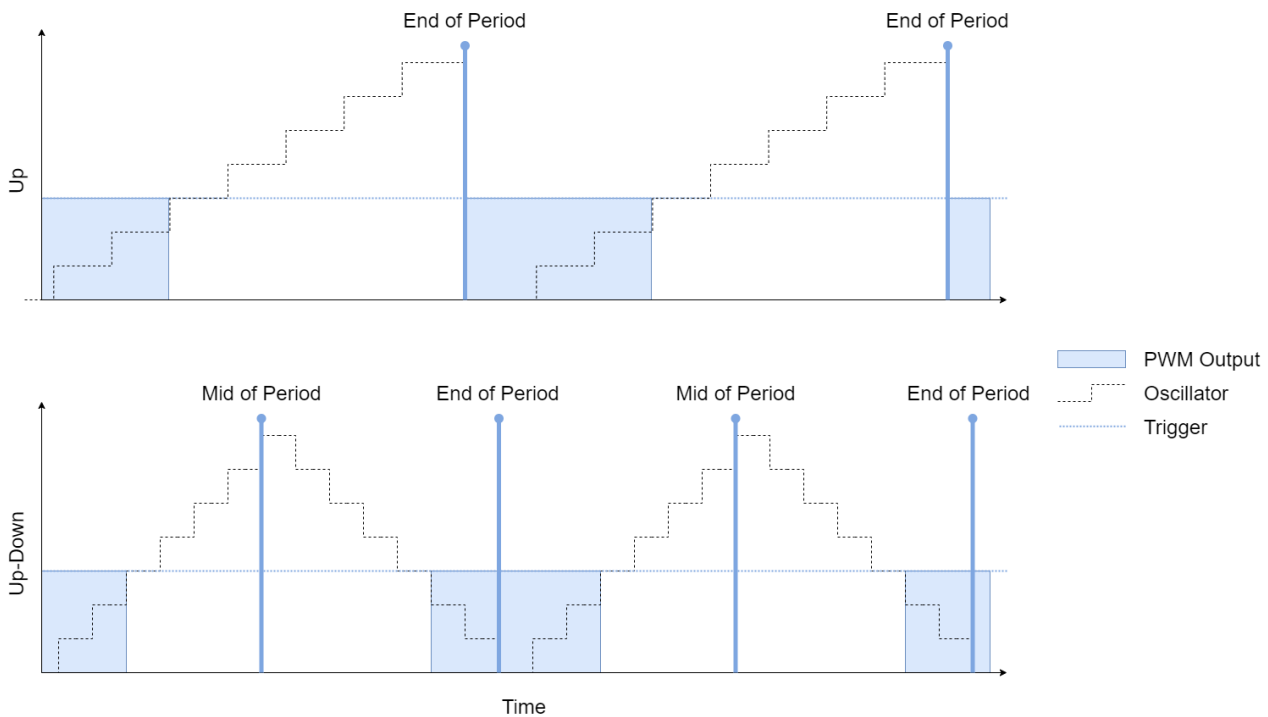


Example: 450e-9

**Event trigger mode – Trigger mode relative to PWM waveform**

End of PWM period (default) | Mid of PWM period | Mid or End of PWM period

Specify when this block triggers an event relative to the PWM waveform.



Example: Mid or End of PWM period

**See Also**

ADC Interface | PWM Write

**Topics**

“Get Started with SoC Blocks on MCUs”

“Integrate MCU Scheduling and Peripherals in Motor Control Application”

**External Websites**

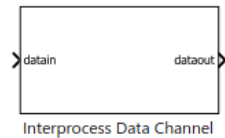
[https://en.wikipedia.org/wiki/Pulse-width\\_modulation](https://en.wikipedia.org/wiki/Pulse-width_modulation)

**Introduced in R2020b**

## Interprocess Data Channel

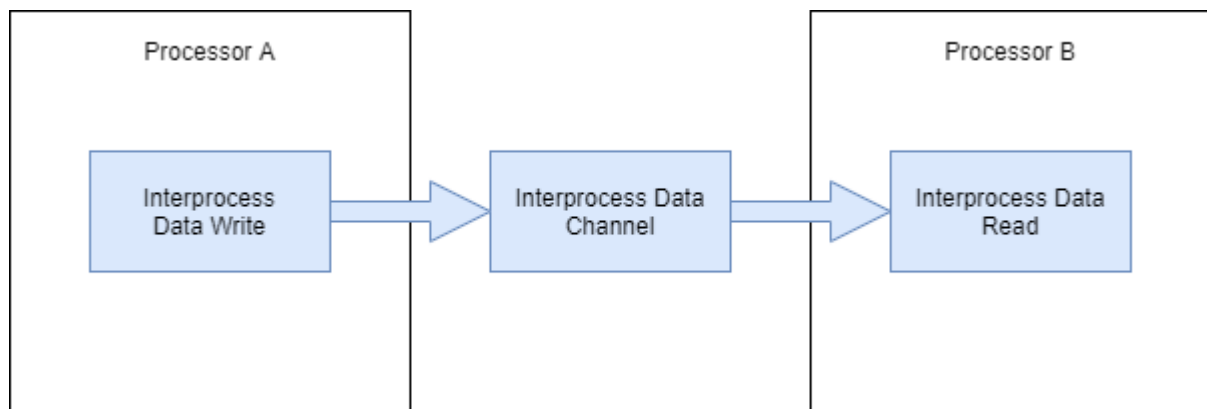
Model interprocessor data channel between two processors

**Library:** SoC Blockset / Processor Interconnect



### Description

The Interprocess Data Channel block simulates the interprocessor data channel available in multiprocessor or OS managed SoC hardware board families. The block provides a channel for asynchronous data transfer between two processors. This diagram shows a generalized view of the interprocessor data connection.



### Ports

#### Input

##### **datain** – Input data message

scalar

This message port receives input data as a message from a connected Interprocess Data Write block. For more information on messages, see “Messages”.

Data Types: SoCData

#### Output

##### **dataout** – Output data message

scalar

This message port sends output data as a message to a connected Interprocess Data Read block. For more information on messages, see “Messages”.

Data Types: SoCData

**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 the **Show event port** parameter.

Data Types: `rteEvent`

**Parameters****Show event port — Option to enable task event ports**`off (default) | on`

Enable an event port that, when connected to the Task Manager block, can execute event-driven tasks.

**See Also**

[Interprocess Data Read](#) | [Interprocess Data Write](#)

**Topics**

[“Multiprocessor Execution”](#)

[“Interprocess Data Communication via Dedicated Hardware Peripheral”](#)

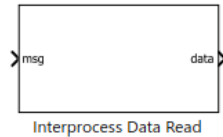
[“Interprocess Data Communication in Operating Systems”](#)

**Introduced in R2020b**

# Interprocess Data Read

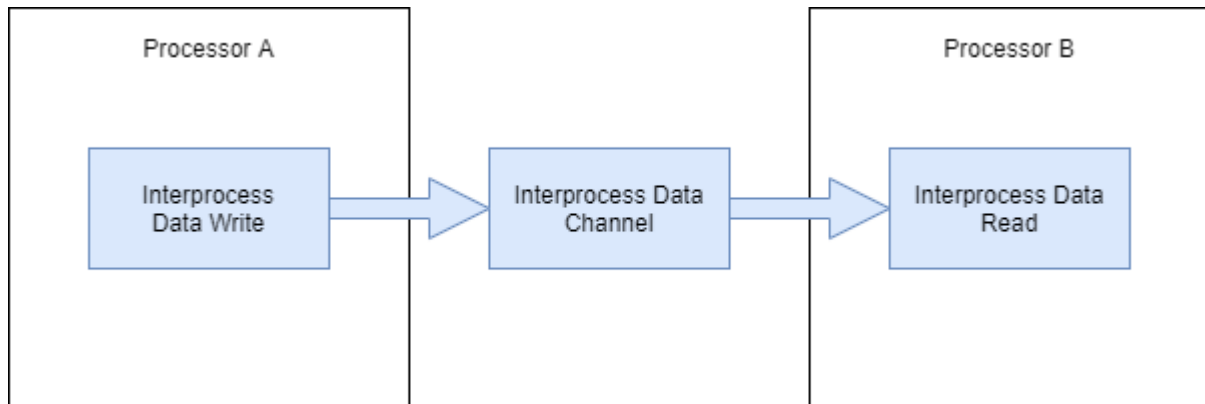
Receive messages from another processor using interprocess communication channel

**Library:** SoC Blockset / Processor Interconnect



## Description

The Interprocess Data Read block asynchronously receives messages from another processor in an SoC using an interprocess data channel. The Interprocess Data Read block connects to an Interprocess Data Channel block that similarly connects to an Interprocess Data Write block contained in a separate processor reference model. In simulation, data from another processor is asynchronously received and processed in the processor containing the Interprocess Data Read block and associated asynchronous task. This diagram shows a generalized view of the interprocessor data channel connection.



## Ports

### Input

**msg** — Data message from interprocess data channel

scalar

This message port receives data messages from the connected Interprocess Data Channel block. The messages process when the Task Manager block triggers the task containing the this block. For more information on messages, see “Messages”.

### Dependencies

Data Types: SoCData



## Output

### **data** — Data frame read from another processor

vector

This port emits a data frame read from another processor connected via the Interprocess Data Channel block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean`

## Parameters

### **Data type** — Data type of interprocess data channel

`double` (default) | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `uint64` | `Boolean`

Enter the data type used by the interprocess data channel.

### **Buffer size** — Size of data vector read from interprocess data channel

1 (default) | positive integer

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

### **Number of buffers** — Number of data buffers in interprocess data channel

4 (default) | positive integer

Enter the number of data buffers in the interprocess data channel.

### **Sample time** — Sample time

-1 (default) | positive scalar

Enter the sample time of the block to apply to the timer-driven task subsystem.

## See Also

IPC Channel | IPC Write

## Topics

“Multiprocessor Execution”

“Interprocess Data Communication via Dedicated Hardware Peripheral”

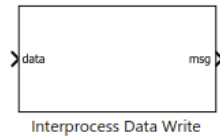
“Interprocess Data Communication in Operating Systems”

## Introduced in R2020b

## Interprocess Data Write

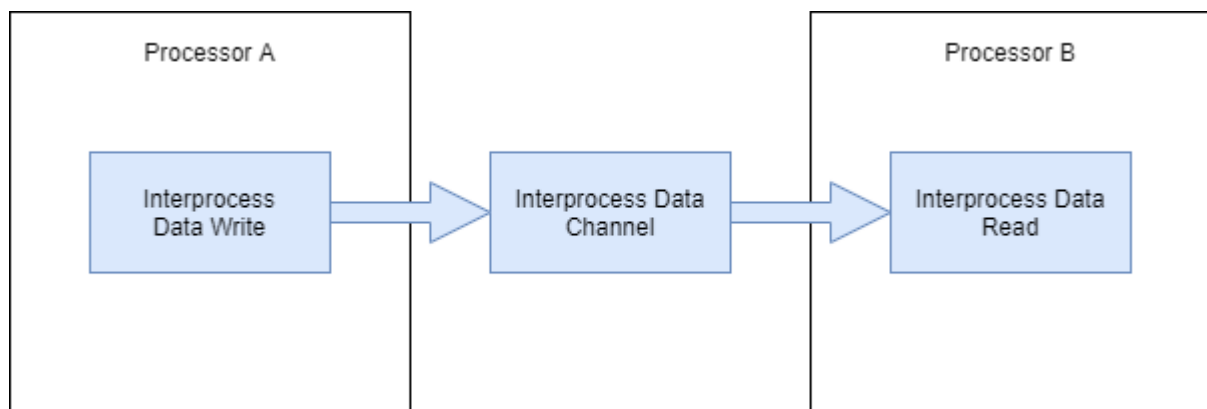
Send messages to another processor using interprocessor data write

**Library:** SoC Blockset / Processor Interconnect



### Description

The Interprocess Data Write block asynchronously sends messages to another processor in an SoC using an interprocess data channel. The Interprocess Data Write block connects to an Interprocess Data Channel block that similarly connects to an Interprocess Data Read block contained in a separate processor reference model. In simulation, data from the current processor is asynchronously sent and processed in the processor containing the Interprocess Data Read block and associated asynchronous task. This diagram shows a generalized view of the interprocess data channel.



### Ports

#### Input

##### data — Data input

vector

This port receives a data vector to send to another processor over the interprocess data channel.

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

#### Output

##### msg — Output data message

scalar

This message port sends the output data as a message to the connected Interprocess Data Channel block. For more information on messages, see “Messages”.

Data Types: SoCData

## **See Also**

Interprocess Data Channel | Interprocess Data Read

## **Topics**

“Multiprocessor Execution”

“Interprocess Data Communication via Dedicated Hardware Peripheral”

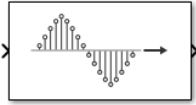
“Interprocess Data Communication in Operating Systems”

**Introduced in R2020b**

## ADC Read

Read ADC data values from ADC Interface block

**Library:** SoC Blockset / Processor I/O



### Description

The ADC Read block converts the message received from the ADC Interface block to a signal that can be used by an algorithm. The data type of the output signal is the same as the data type in the contained data message.

### Ports

#### Input

**msg — Data message from register**

scalar

This message port receives ADC value messages from a connected ADC Interface block. The messages process when the Task Manager block triggers a task containing the ADC Read block. For more information on messages, see “Messages”.

Data Types: SoCData

#### Output

**data — Output signal**

scalar

This port emits a measurement from the ADC Interface block.

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

### Parameters

**Data type — Data type of ADC measurements received**

double (default) | single | uint8 | int8 | int16 | int32 | uint16 | uint32

Select the data type of the input data. Match this data type with data type of the ADC hardware.

**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.

**See Also**

ADC Interface | PWM Interface | PWM Write

**Topics**

“Get Started with SoC Blocks on MCUs”

“Integrate MCU Scheduling and Peripherals in Motor Control Application”

**External Websites**

[https://en.wikipedia.org/wiki/Analog-to-digital\\_converter](https://en.wikipedia.org/wiki/Analog-to-digital_converter)

**Introduced in R2020b**

## PWM Write

Send pulse width modulation (PWM) signal configuration to PWM Interface block

**Library:** SoC Blockset / Processor I/O



### Description

The PWM Write block sets the duty cycle for a PWM peripheral. In simulation, the block passes through the duty cycle input to drive the PWM Interface block that simulates the PWM switching signals produced by the hardware. When deployed to hardware, the PWM Write block writes to the appropriate PWM drivers on the hardware.

### Ports

#### Input

##### In — Duty cycle

scalar

Specify the duty cycle value of the PWM waveform. This input must be a numeric scalar from 0 to 1.

Data Types: `single` | `double`

#### Output

##### msg — PWM duty cycle values

scalar

This message port sends duty cycle values as messages to a connected PWM Interface block. For more information on messages, see “Messages”.

---

**Note** This output is used only during simulation and is ignored in code generation and external mode simulation.

---

Data Types: `SoCData`

### See Also

PWM Interface

### Topics

“Get Started with SoC Blocks on MCUs”

“Integrate MCU Scheduling and Peripherals in Motor Control Application”

### External Websites

[https://en.wikipedia.org/wiki/Pulse-width\\_modulation](https://en.wikipedia.org/wiki/Pulse-width_modulation)

**Introduced in R2020b**

## 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 “Messages”.

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.

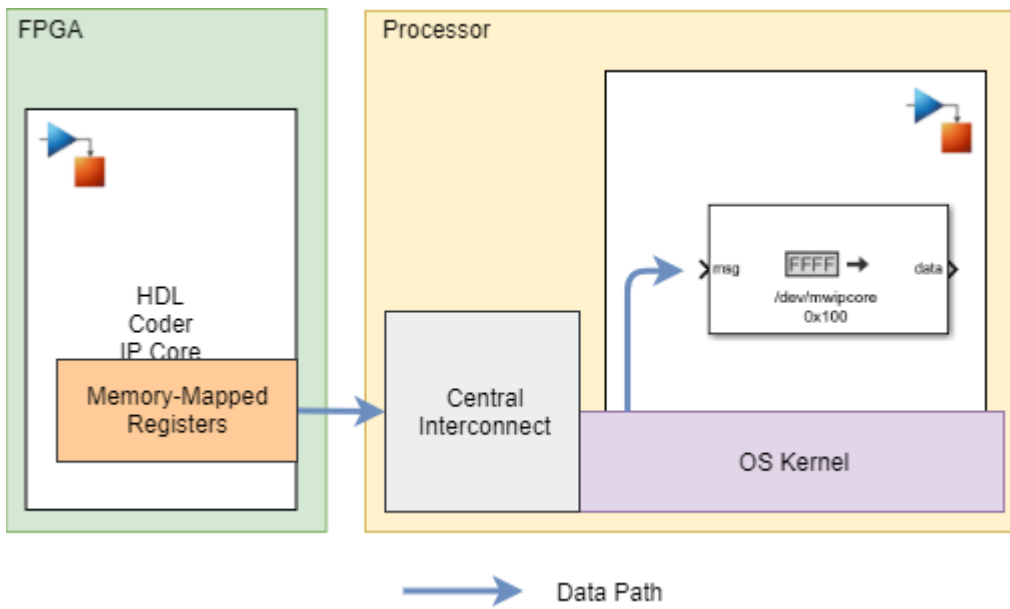
## Extended Capabilities

**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**

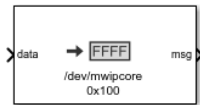
Register Channel

**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 “Messages”.

Data Types: `SoCData`

## Parameters

#### **Output sink** — Direct output to port or base workspace

To output port (default) | Base workspace | IP core register

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. Select `IP core register` to write to an IP Core Register Read block with the same `Register name` parameter.

---

**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

#### **Dependencies**

To enable this parameter, set `Output sink` to Base workspace.

### **Register name — Name of a register in IP Core**

RegA (default) | character vector

Name of register defined in an IP Core Register Read block located in the FPGA reference model.

Example: RegA

#### **Dependencies**

To enable this parameter, set `Output sink` to IP core register.

### **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**

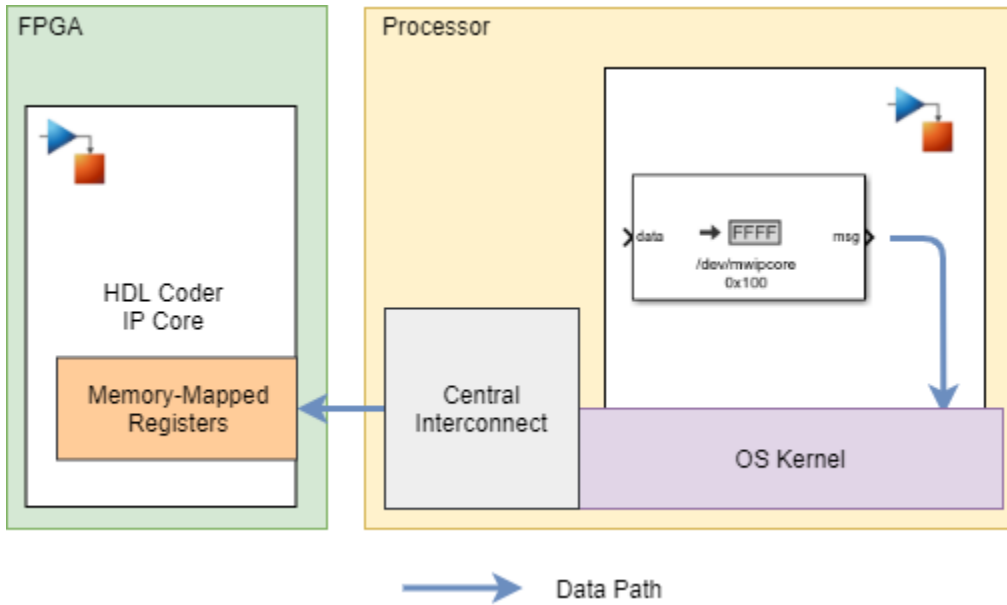
### **C/C++ Code Generation**

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 | uint64

#### **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 Memory Channel or IO Data Source block that the read completed. For more information on messages, see “Messages”.

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 “Messages”.

### Dependencies

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

Data Types: SoCData

## Parameters

### Main

#### Device name — Name of IP core device

ip:s2mm0 (default) | colon-separated list of IP core name and channel

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. This number should match the **Number of buffers** parameter in the Memory Channel block or IO Data Source block.

#### 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 full buffer is available in the 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.

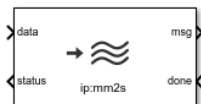




# Stream Write

Stream data from processor algorithms to shared memory

**Library:** SoC Blockset / Processor I/O



## Description

The Stream Write block streams data from your processor algorithm to shared memory in the Memory Channel block. The Stream Write block has an internal counter that keeps track of the number of empty buffers in the shared memory. After a successful read from memory, the memory sends a done signal to the Stream Write block. Then, the block increments the counter, asserting that a buffer is available in the memory. A write transaction is successful if at least one buffer is available for writing. The Stream Write block sends a status of `True` back to the software. You can use this status signal to perform actions such as counting dropped frames or issuing rewrite requests.

In simulation, a timer-driven or event-driven task subsystem contains the Stream Write block. The data signals from the software algorithm connect to the Stream Write block. The write transaction is issued as a message to the Memory channel block. After a read operation from shared memory, the Memory Channel block notifies the Stream Write block of the read event via the done signal.

## Ports

### Input

#### **data** — Data frame from software algorithm

vector

This port receives a data frame from the software algorithm. The block then streams the data as a message to a region of shared memory defined in the Memory Channel block.

Data Types: `uint32` | `uint64`

#### **done** — Notification of available buffer in memory

scalar

This message port receives a notification from the connected Memory Channel or IO Data Sink block. The notification indicates that a read transaction completed and that a buffer in memory is available for writing.

Data Types: `Boolean`

### Output

#### **msg** — Data message to memory

scalar

When buffer space is available in the memory, this message port emits data messages to the connected Memory Channel or IO Data Sink block. For more information on messages, see “Messages”.

Data Types: SoCData

**status — Status of completed write transaction**

scalar

This port sends a true status(1) to the processor after a write transaction to memory occurred. Use this status to count dropped frames.

Data Types: Boolean

## Parameters

**Device name — Name of IP core device**

ip:MM2S (default) | colon-separated list of IP core name and channel

The device name parameter is generated by the **SoC Builder** tool. Enter the name and channel of the IP core on the FPGA as a colon-separated list.

**Number of buffers — Number of data buffers**

3 (default) | positive integer

Enter the number of data frame buffers in the physical memory. This number must match the **Number of buffers** parameter in the Memory Channel or IO Data Sink block.

**Enable event-based execution — Option to enable event-driven task execution**

on (default) | off

- Select this parameter to use this block in event-driven task subsystems. In this case, the block writes to the Memory Channel block each time an empty buffer is available in the shared memory region.
- Clear this parameter to use this block in timer-driven task subsystems. In this case, the block writes the data in the shared memory region at each sample time.

## 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 **SoC Builder** tool implements the Stream Write, 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 enables high-speed streaming of data between the FPGA and the embedded processor through external memory.

## **See Also**

### **Blocks**

Memory Channel | Stream Read | Task Manager

### **Topics**

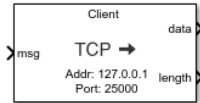
“Event-Driven Tasks”

### **Introduced in R2020b**

# 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 “Messages”.

---

**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.

**Sample time — Sample time**

-1 (default) | nonnegative scalar

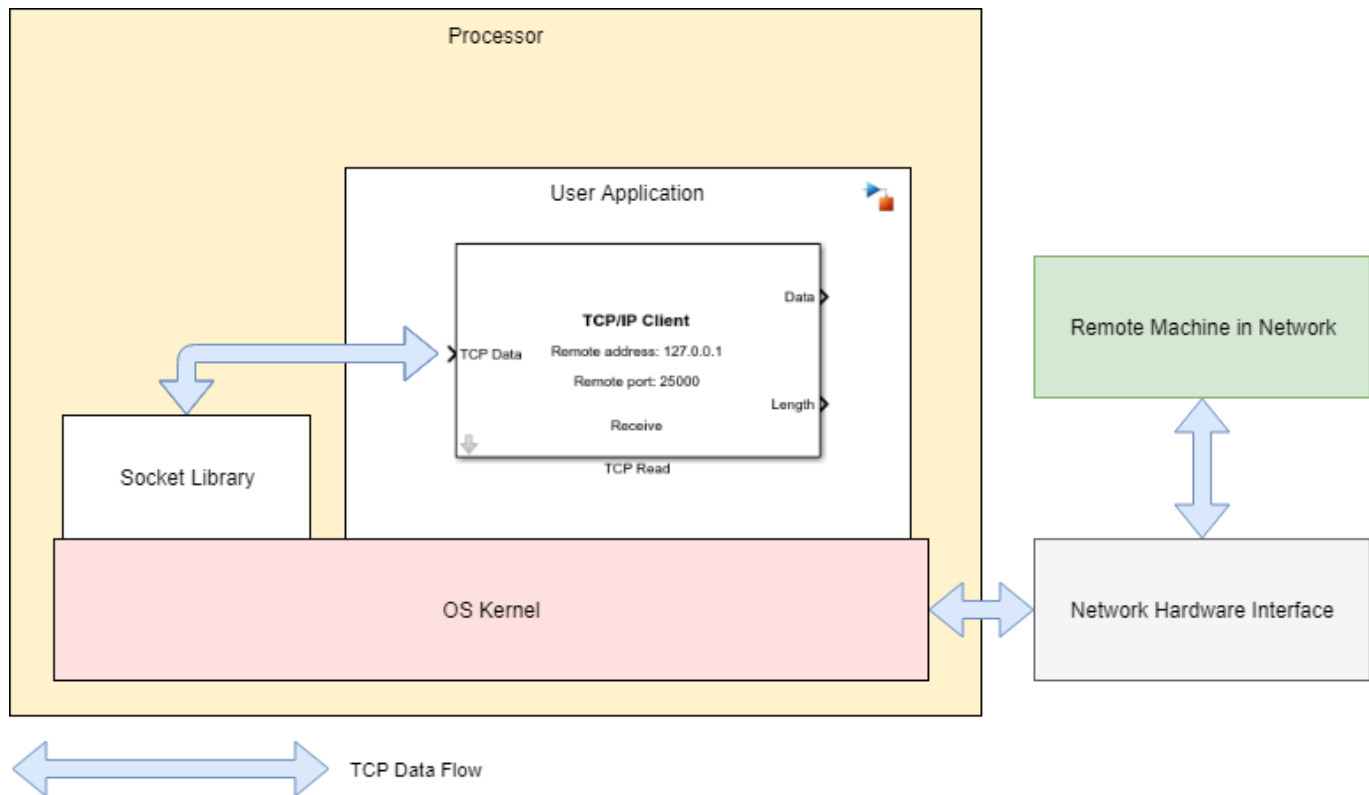
Specify how often the scheduler runs this block. If this value is -1 (default), the scheduler assigns the sample time for the block.

**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

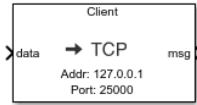
IO Data Source | TCP Write | Task Manager

**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 “Messages”.

---

**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 listening 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.

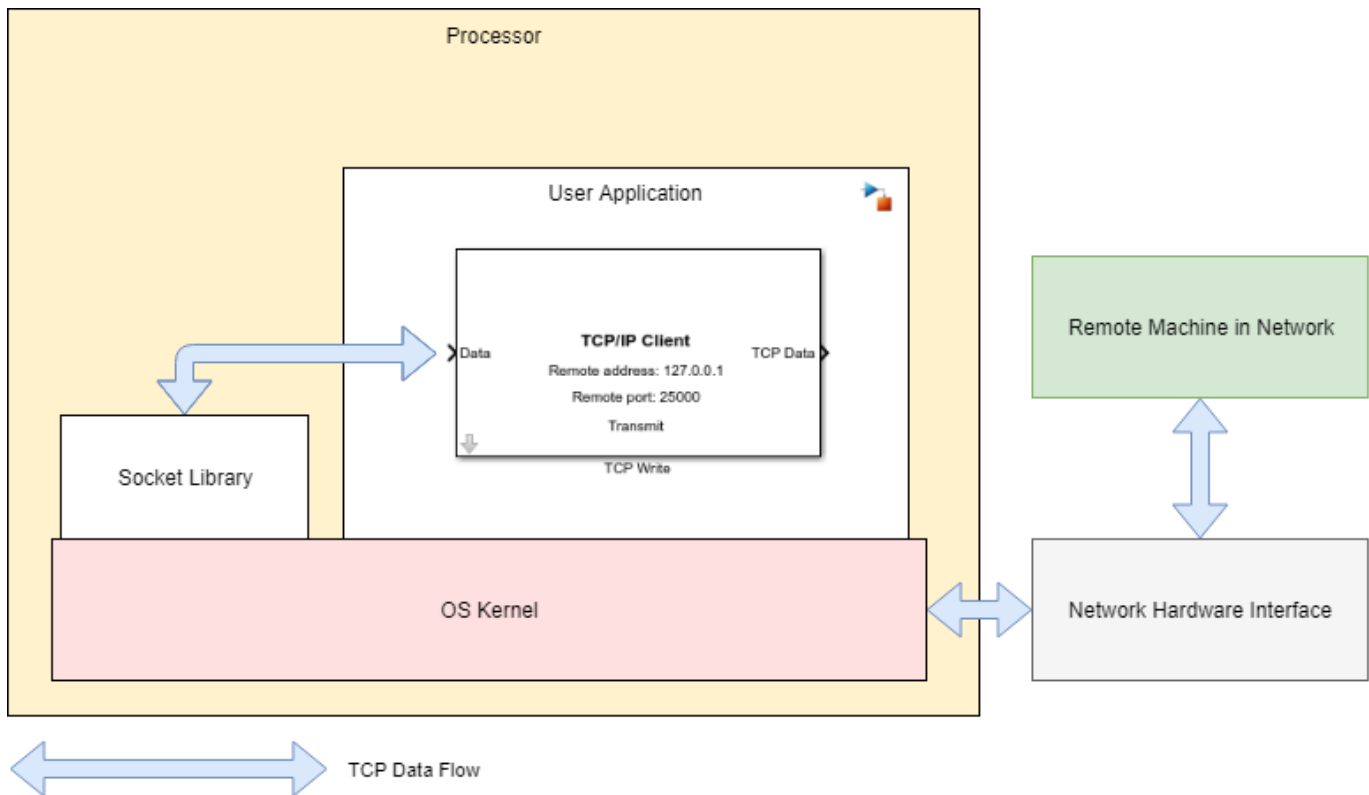
## Extended Capabilities

### 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

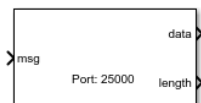
[IO Data Sink](#) | [TCP Read](#) | [Task Manager](#)

**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 “Messages”.

---

**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.

### **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.

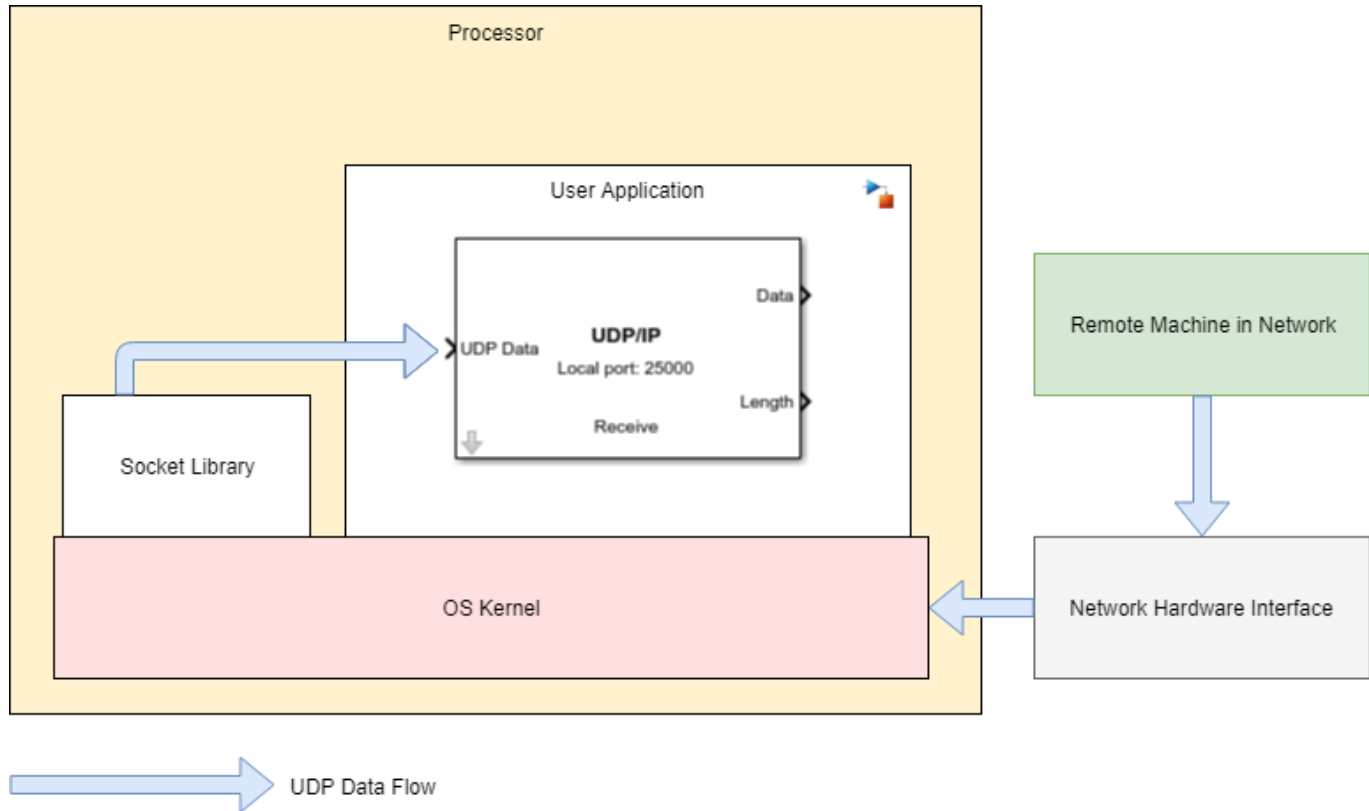
## Extended Capabilities

### **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.



**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

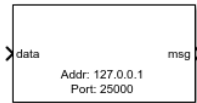
[IO Data Source](#) | [Task Manager](#) | [UDP Write](#)

**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 “Messages”.

---

**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.

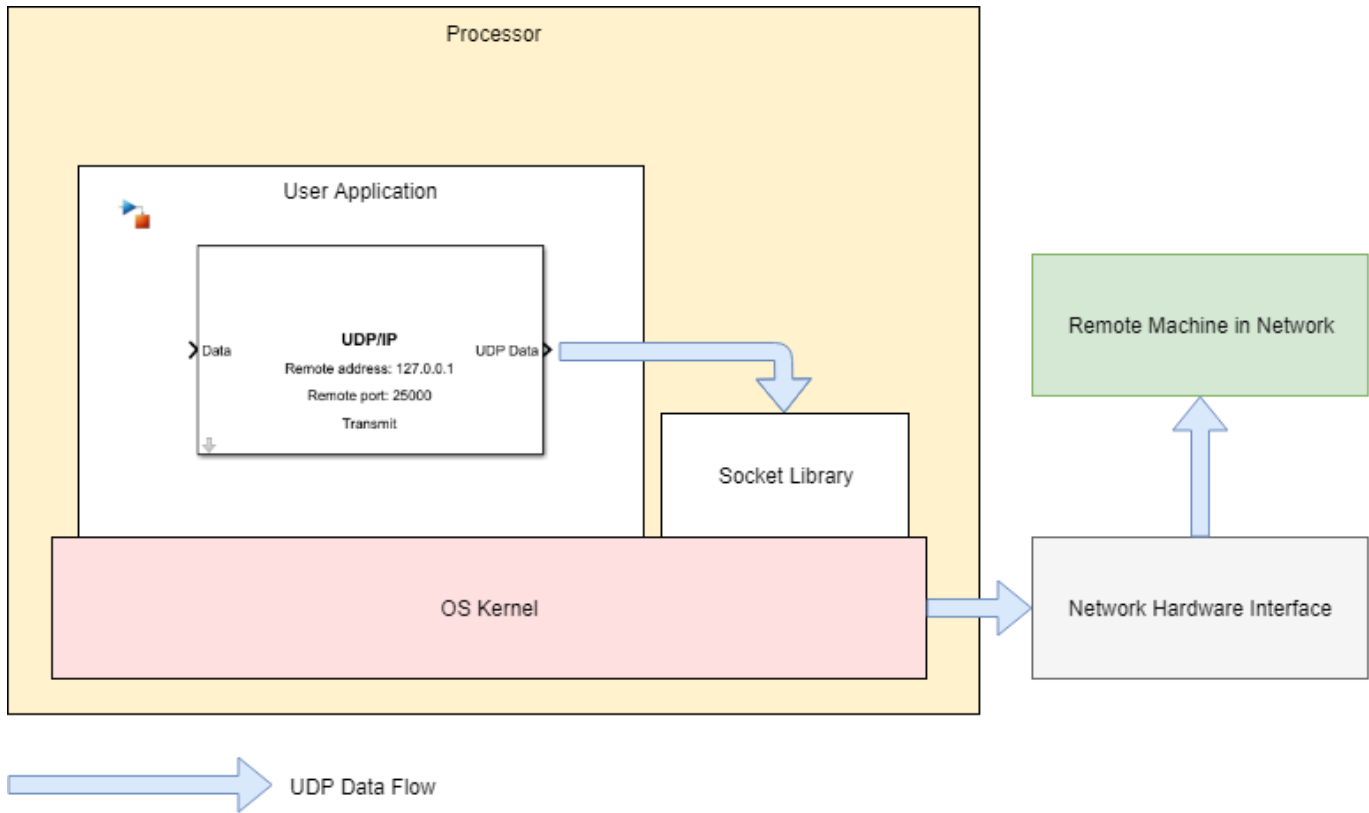
## Extended Capabilities

**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.

## Limitations

- A model containing a Task Manager blocks does not support simulation stepping. For more information on simulation stepping, see “Simulation Stepper”.

## 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

#### **Task1Event — 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 “Messages”.

### Dependencies

To show a *Task1Event* port, then *Task1* must have **Type** set to Event-driven.

Data Types: `rteEvent`

#### **Task1Dur — 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 scheduler's 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”.

---

**Note** No more than 2 instances of a task can overrun execution when Drop tasks that overrun is set to off. Any additional task instances that overrun drop automatically.

---

## 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**

“Get Started with SoC Blocks on MCUs”

“What is Task Execution?”

“Task Duration”

**Introduced in R2019a**

# Proxy Task

A placeholder for a task in your application

**Library:** SoC Blockset / Processor Task Execution



## Description

The Proxy Task block simulates the effect of a timer-driven task in your SoC application without an explicit implementation. This block can be used as a placeholder for timer-driven tasks to be developed in the future, or implemented simultaneously by another developer.

When added to your processor reference Model block, it causes the processor model, when set to schedule rates by ports, to display a periodic event port with a sample time equal to **Sample Time** parameter. Connect the periodic event port to a timer-driven task output port on the Task Manager block.

## Ports

### Input

#### Port\_1 — Function-call from Task Manager block

scalar

A function-call signal that triggers the Proxy Task block when operating as an event-driven task.

### Dependencies

To enable this port, set the Type property to Event-driven.

Data Types: `function-call`

## Parameters

### Type — Type of task

Timer-driven (default) | Event-driven

Select the type of task as either Timer-driven or Event-driven.

### 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.
- 

### **Dependencies**

To enable this parameter, set the Type property to Timer-driven.

### **See Also**

Task Manager | Testbench Task

### **Topics**

“Timer-Driven Task”

### **Introduced in R2019b**



# Event Source

Simulate and playback recorded task events

**Library:** SoC Blockset / Processor Testbench



## Description

The Event Source block, connected to the Task Manager block, enables you to simulate tasks events in your Simulink model. Task timing data can be provided from the block mask, an external file, or from an input port driven by other model signals.

## Ports

### Input

#### **data — Input data**

numeric vector

Input data, specified as a numeric vector. The block converts this data into an event that corresponds to the rate of the input data. You can use this event to drive an event-driven task in the Task Manager block.

### Dependencies

To enable this port, set the **Input** parameter to `From input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

### 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.

Data Types: `rteEvent`

## Parameters

### **Input — Source of input data**

`From dialog (default)` | `From input port` | `From timeseries object`

Set the input data source for the block by selecting one of these values.

- `From dialog` — Input a one-dimensional array of data by using a function. Specify this function for the **Value** parameter.

- From `input port` — Input data and signals using input ports on the block.
- From `timeseries object` — Input data and time values using a `timeseries` object that you created in MATLAB. For more information see “Time Series Objects and Collections”.

**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** parameter to `From dialog`.

**Object name — Name of timeseries object**

[] (default) | `timeseries` object

Specify a `timeseries` object. This `timeseries` object provides the input data for the block. For more information about time series objects, see “Time Series Objects and Collections”.

**Dependencies**

To enable this parameter, set the **Input** parameter to `From timeseries object`.

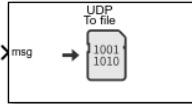
**See Also**

**Introduced in R2020b**

# 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 Stream WriteTCP Write, UDP Write, or Register Write 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.

### Dependencies

To enable this port, set the **Output** parameter to To output port.

Data Types: uint32 | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | Boolean | fixedpoint

#### **length** — Length of output data

nonnegative scalar

Length of output data, returned as a nonnegative scalar.

### Dependencies

To enable this port, set the **Output** parameter to To output port.

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.

**Dependencies**

To enable this port, set the **Output** parameter to `To output port`.

Data Types: Boolean

**done — Completion of data streaming**

Boolean scalar

When **done** is 1, the block has no more stream output data to return in the **data** port. When **done** is 0, the block has more stream data to return in the **data** port.

**Dependencies**

To enable this port, set the **Device type** parameter to `Stream`.

Data Types: Boolean

**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, set the **Show port** parameter to `Data and Event`.

Data Types: `rteEvent`**Parameters****Output — Sink of output data from 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` — Enable the **msg** input port to receive UDP data as message from a **msg** port of UDP Write block.
- `TCP` — Enable the **msg** input port to receive TCP data as message from a **msg** port of TCP Write block.

- **Register** — Enable the **msg** input port to receive Register data as message from a **msg** port of Register Write block.
- **Stream** — Enable the **msg** input port to receive Stream data as message from a **msg** port of Stream Write block.

#### Show port — Enable output ports

Data (default) | Data and Event

Select one of these values to enable the corresponding output ports towards the writing source.

- **Data** — Enable only the **msg** input port.
- **Data and event** — Enable the **msg** input and **event** output ports.

#### Number of buffers — Number of data buffers

8 (default) | nonnegative scalar

Specify the number of data elements to store in the data queue. This parameter must match the **Number of buffers** parameter specified in the Memory Channel block.

#### Dependencies

To enable this parameter, set the **Device type** parameter to **Stream**.

#### 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** 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** parameter to **To file**.

#### Data type — Data type of output data

uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | boolean | fixedpoint

Select the data type of the output data. This value must match the data type of the input data.

#### Dependencies

To enable this parameter, set the **Output** parameter to **To file** or **To output 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 Write, UDP Write, or Register Write block.

### **See Also**

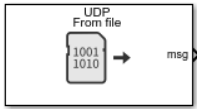
[IO Data Source](#) | [Register Write](#) | [Stream Write](#) | [TCP Write](#) | [Task Manager](#) | [UDP Write](#)

**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** 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** parameter to `From input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

**Length — Length of input data**

nonnegative scalar

Length of input data, specified as a nonnegative 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 — Notification of freed buffer in memory**

False (default) | True

This message port receives a notification from the connected Memory Channel or IO Data Sink block that a read transaction completed, and that a buffer in memory is available for writing.

**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, set the **Show port** parameter to Event or Data and event.

Data Types: rteEvent

**msg — SoC message data**

numeric vector

This port sends the data vector as a message to the **msg** input port of processor I/O blocks, which includes Register Read, Stream Read, UDP Read, and TCP Read blocks.

**Dependencies**

To enable this port, set the **Show port** parameter to Data or Data and Event.

Data Types: SoCData



## Parameters

### Input — Source of input data

From file (default) | From dialog | From input port | From timeseries object

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.
- From dialog — Input a one-dimensional array of data by using a function. Specify this function for the **Value** parameter.
- From input port — Input data and signals using input ports on the block.
- From timeseries object — Input data and time values using a timeseries object that you created in MATLAB. For more information see “Time Series Objects and Collections”.

### 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** parameter to From dialog.

### Data type — Data type of input data

uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | 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** 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** 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** 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** parameter to From file.

**Dataset name — Name of recorded file**

no default | file path

Specify the full path to a recorded data file on the host PC or browse and select a 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** 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** parameter to From file.

**Number of buffers — Number of data buffers**

1024 (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.

**Show port — Enable output ports**

Data (default) | Event | Data and event

Select one of these values to enable the corresponding output ports.

- Data — Enable only the **msg** output port.
- Event — Enable only the **event** output port.
- Data and event — Enable the **msg** and **event** output ports.

**Object name — Name of timeseries object**

[] (default) | timeseries object

Specify a `timeseries` object. This `timeseries` object provides the input data for the block. For more information about time series objects, see “Time Series Objects and Collections”.

### **Dependencies**

To enable this parameter, set the **Input** parameter to `From timeseries object`.

## **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

An external timer-driven task load on your 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

This port accepts a 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**

# Configuration Parameters

---

# Hardware Implementation Pane

## Hardware Implementation Pane Overview

Hardware board: Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

Code Generation system target file: [ert.tlc](#)

Device vendor: ARM Compatible Device type: ARM 64-bit (LP64)

▶ Device details

Hardware board settings

- ▼ Task profiling in simulation
  - Show in SDI
  - Save to file  Overwrite file
- ▼ Task profiling on processor
  - Show in SDI
- ▼ Operating system/scheduler
  - Kernel latency:
- ▼ Task and memory simulation
  - Set seed for simulating task duration and memory access
  - Cache input data at task start
- ▼ Target hardware resources
  - Groups
    - Board Parameters
    - Build options
    - Clocking
    - External mode
    - FPGA design (top-level)
      - Include 'MATLAB as AXI Master' IP for host-based interaction
      - Include processing system
    - FPGA design (mem controllers)
      - Register configuration clock frequency (MHz):
    - FPGA design (mem channels)
      - IP core clock frequency (MHz):
    - FPGA design (debug)

### Hardware board settings

Parameter	Description	Default Value
"Processing Unit" on page 2-8	Processor for model reference block in the SoC model.	None

### Design Mapping

Parameter	Description	Default Value
"View/Edit Task Map" on page 2-9	Open the task mapping tool.	not applicable

Parameter	Description	Default Value
“View/Edit Peripheral Map” on page 2-9	Open the task mapping tool.	not applicable

### Task profiling in simulation

Parameter	Description	Default Value
“Show in SDI” on page 2-10	Show the task execution data collected in simulation in the <b>Simulation Data Inspector</b> application.	on
“Save to file” on page 2-10	Save the task execution data to a file.	on
“Overwrite file” on page 2-10	Overwrite the last task execution data file.	off

### Task profiling on processor

Parameter	Description	Default Value
“Show in SDI” on page 2-11	Show the task execution data collected on hardware in the <b>Simulation Data Inspector</b> application.	off
“Save to file” on page 2-11	Save the task execution data to a file.	off
“Overwrite file” on page 2-11	Overwrite the last task execution data file.	off
“Instrumentation” on page 2-11	Choose to perform code instrumentation or Kernel instrumentation.	Code
“Profiling duration” on page 2-11	Choose whether to perform Kernel profiling for an unlimited or limited time duration.	Unlimited

### Operating system/scheduler

Parameter	Description	Default Value
“Kernel latency” on page 2-13	Specify the Kernel latency of the OS in simulation of a task.	0

## Task and memory simulation

Parameter	Description	Default Value
“Set seed for simulating task duration and memory access” on page 2-14	Set the random number generator seed.	off
“Seed Value” on page 2-14	Specify the seed value for the simulation of task duration deviation.	default
“Cache input data at task start” on page 2-14	Cache the input data at the start of a task.	off

## Board Parameters

Parameter	Description	Default Value
Device Address	Network address of hardware board or device.	192.168.1.10
Username	Login username on hardware board or device.	root
Password	Login password on hardware board or device.	root

## Processor

Parameter	Description	Default Value
“Number of cores” on page 2-15	Set the number of CPU cores in the processor.	1

## Board Options

Parameter	Description	Default Value
“Build Action” on page 2-17	Defines how <b>SoC Builder</b> tool responds when you build your model.	Build, load, and run

## Clocking

Parameter	Description	Default Value
CPU Clock (MHz) on page 2-16	The CPU clock frequency in MHz.	1000



## External Mode

Parameter	Description	Default Value
“Communication Interface” on page 2-18	Transport layer used to exchange data between the development computer and hardware.	TCP/IP
“Run external mode in a background thread” on page 2-18	Execute the external mode engine in the generated code in a background task.	disabled
“Port” on page 2-18	IP address port on hardware board.	17725
“Verbose” on page 2-19	Enable view of the external mode execution progress and updates in the Diagnostic Viewer.	disabled

## FPGA design (top-level)

Parameter	Description	Default Value
“View/Edit Memory Map” on page 2-20	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 2-20	Use host-based scripts with an integrated JTAG master on the target platform.	on
“Include processing system” on page 2-20	For processor-based platforms, include the processing system.	on
“Interrupt latency (s)” on page 2-20	The latency from hardware asserting an interrupt to the start of the interrupt service routine.	0.00001
“Register configuration clock frequency (MHz)” on page 2-20	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 2-20	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 2-22	Frequency of datapath between memory interconnect and memory controller.	200

Parameter	Description	Default Value
“Controller data width (bits)” on page 2-22	Bit width of datapath between memory interconnect and memory controller.	64
“Bandwidth derating (%)” on page 2-22	For every 100 clocks, will hold off all transaction execution for this number of clocks.	2.3
“First write transfer latency (clocks)” on page 2-22	Number of clock cycles between write request and start of transfer.	4
“Last write transfer latency (clocks)” on page 2-23	Number of clock cycles between the end of write transfer and completion of transaction.	4
“First read transfer latency (clocks)” on page 2-23	Number of clock cycles between read request and start of transfer.	5
“Last read transfer latency (clocks)” on page 2-23	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 2-24	Frequency of the master datapath to the interconnect controller in MHz.	200
“Interconnect data width (bits)” on page 2-24	Data width of master datapath to interconnect controller in bits.	64
“Interconnect FIFO depth (num bursts)” on page 2-24	Maximum number of bursts that can be buffered before data is dropped.	12
“Interconnect almost-full depth” on page 2-24	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 2-25	The internal operation of the memory channel can be instrumented for debug or diagnostic analysis.	Basic diagnostic signals

<b>Parameter</b>	<b>Description</b>	<b>Default Value</b>
"Include AXI interconnect monitor" on page 2-25	Gather performance metrics of the memory interconnect such as data throughput, latency, and number of bursts executed.	off
"Trace capture depth" on page 2-25	Maximum number of Trace entries to be logged in trace mode	1024

## Hardware Board Settings

### Processing Unit

Choose the processor in the MCU onto which to deploy the model reference block in the SoC model. The top level SoC model is set to None.

### Settings

**Default:** None

### See Also

“Multiprocessor Execution” | “Run Multiprocessor Models in External Mode”

## Design Mapping

### **View/Edit Task Map**

Open the Task Mapping tool to map tasks in the model to available hardware interrupt sources for the selected hardware board.

### **View/Edit Peripheral Map**

Open the Peripheral Configuration tool to map tasks in the model to available hardware interrupt sources for the selected hardware board.

### **See Also**

## **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 Processor

### Show in SDI

Show the task execution data collected on a processor in the Simulation Data Inspector (SDI) application. For more information on visualizing tasks in the SDI application, see “Task Visualization in Simulation Data Inspector”.

**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”.

**Default:** off

### Overwrite file

Overwrite the last task execution data file. For more information on recording task execution data, see “Recording Tasks for Use in Simulation”.

**Default:** off

### 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”.

**Default:** Code

### Profiling duration

Choose the profiling duration for Kernel profiling.

If you select Kernel to specify Kernel profiling, set **Profiling duration** to **Unlimited** or **Limited**.

- **Unlimited** — This option performs Kernel profiling on the hardware and streams it to the host PC for an unlimited time duration. Kernel profiling for an unlimited time duration on hardware with low free disk storage or a model with high task rates can result in packet loss of profiling data streamed from the hardware. This packet loss depends on the free memory available on the host PC on which you run MATLAB.
- **Limited** — This option performs Kernel profiling on the hardware and streams it to host PC for a limited time duration. Kernel profiling for a limited time duration on hardware does not result in packet loss of profiling data streamed from hardware. The maximum time duration for limited Kernel profiling depends on the RAM and free disk storage available on the hardware board on which Kernel profiling is performed.

**Default:** Unlimited



## 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

## **Clocking**

### **CPU Clock (MHz)**

The frequency of the CPU core clock in MHz.

#### **Settings**

**Default:** 1000

## Build Action

Defines how **SoC Builder** tool builds your model.

### Settings

**Default:** Build, load, and run

Build, load, and run

With this option, launching **SoC Builder**:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.
- 3 Loads the executable and libraries into the hardware board.
- 4 Runs the executable in the hardware board.

Build

With this option, launching **SoC Builder**:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable on the hardware board.

**See Also**  
**SoC Builder**

## External Mode

### Communication Interface

Select the transport layer that external mode uses to exchange data between the host computer and the target hardware.

#### Settings

**Default:**TCP/IP, XCP on TCP/IP

### Run external mode in a background thread

Force the external mode task in the generated code to execute in a background thread.

When external mode runs in the model thread, external mode executes after each execution step of the model and collects data at the base rate of the model. When model code consumes most of the thread execution time in each time step, external mode execution overruns into the next time step. This overrun delays the start of the next model execution step and degrades the real-time behavior of the deployed model.

You can configure external mode to run in a background thread. When external mode runs in a background thread, it executes in the time between the end of model code of one time step and the start of the next time step. By not blocking the model step, external mode can be used in systems that require real-time execution. This configuration enables direct observation of the deployed model on the hardware board as it would behave in standalone operation.

When model code consumes most of the execution time for each time step, external mode in the background thread starves for execution time. Without sufficient time to collect and transmit data from the hardware board to the host computer, data packets drop. This case results in gaps in the data logging.

To help avoid dropped data packets in deployed models where real-time execution takes priority over data logging, configure external mode to operate as a background task.

---

**Note** Enabling the `Run external mode in a background thread` parameter is not recommended for models that use a very small time step or that might encounter task overruns. These situations can cause Simulink to become unresponsive.

---

#### Settings

**Default:**disabled

### Port

Enter the port for the IP address of the hardware board.

#### Settings

**Default:** 17725

**Verbose**

To view the external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window, select this check box.

**Settings**

**Default:** disabled

## 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','username','r
```

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 =
```

```
1x1 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','username','r
```

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 =
```

```
1x0 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 =
```

```
1x1 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 =
```

```
1x0 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', 'username', 'r
```

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 =
```

```
1x1 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','username','r
```

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 =
```

```
1x1 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 =
```

```
1x0 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','username','r
```

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 =
```

```
1x1 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.

## Output Arguments

**recordingStatus** — Status of data recording session

false (0) | true (1)

Status of data recording session, returned as logical value 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','username','r
```

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 =
```

```
1x1 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.

### **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

## socTaskTimes

Plot histogram of the task durations from a recorded Simulation Data Inspector 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 Simulation Data Inspector 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.

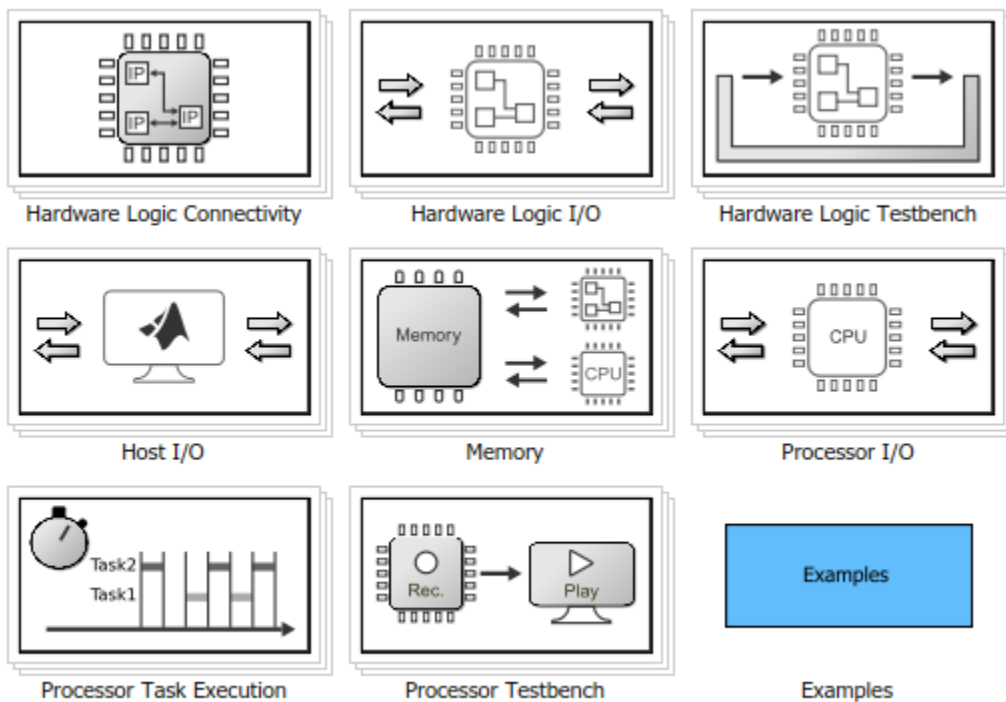
### Examples

#### 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
```



## **See Also**

“Get Started with SoC Blockset”

**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 =
```

```

1x10 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 =
```

```

1x10 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 =
```

```

1x10 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',numeric(0,6,4))
```

```
rd_d =
```

```

Columns 1 through 10
      0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.0000    2.2500
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'` |  
numeric type object

Data type assigned to the read data, specified as `'uint32'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'single'`, or a numeric type 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',numeric(0,6,4))
```

```
rd_d =
```

```
Columns 1 through 10
```

```
0 0.2500 0.5000 0.7500 1.0000 1.2500 1.5000 1.7500 2.0000 2.2500
```

```
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 writes 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 =
```

```
1x10 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 =
```

```
1x10 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',numeric(0,6,4))
```

```
rd_d =
```

```
Columns 1 through 10
```

```
0 0.2500 0.5000 0.7500 1.0000 1.2500 1.5000 1.7500 2.0000 2.2500
```

```
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**

## socFunctionAnalyzer

Estimate number of operations in MATLAB function

### Syntax

```
socFunctionAnalyzer(functionName)
socFunctionAnalyzer(functionName,Name,Value)
report = socFunctionAnalyzer(___)
[report,y1,...,yn] = socFunctionAnalyzer(___)
```

### Description

`socFunctionAnalyzer(functionName)` generates a report with the estimated number of operations in the MATLAB function specified by `functionName`.

The function generates the report as a Microsoft® Excel® spreadsheet and a MAT-file. The function also provides a link to view the report in a separate dialog box.

The report includes information for each mathematical or logical operator in the function, with individual lines for each operator and data type. For example, multiplication with data type `double` and multiplication with data type `uint32` are listed separately. The report lists each instance of the operator as a separate line. The report includes these fields.

- `Path` - The path to the operator within the structural hierarchy of the top function
- `Count` - The number of times the operator is executed in the design
- `Operator` - The operator used
- `DataType` - The data type used for the output of the operator
- `Link` - A link to the location of the operator in the function

For more information, see “Using the Algorithm Analyzer Report”.

`socFunctionAnalyzer(functionName,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'IncludeOperator','+'` specifies that the generated report only includes `'+'` operator counts.

`report = socFunctionAnalyzer(___)` returns a structure of tables that contain report information. Specify any of the input argument combinations from previous syntaxes.

`[report,y1,...,yn] = socFunctionAnalyzer(___)` returns the outputs `y1,...,yn` of the specified function. Specify any of the input argument combinations from previous syntaxes.

### Examples

#### Analyze Resources in Function

This example calculates the number of operators in the function `soc_test_func.m`.

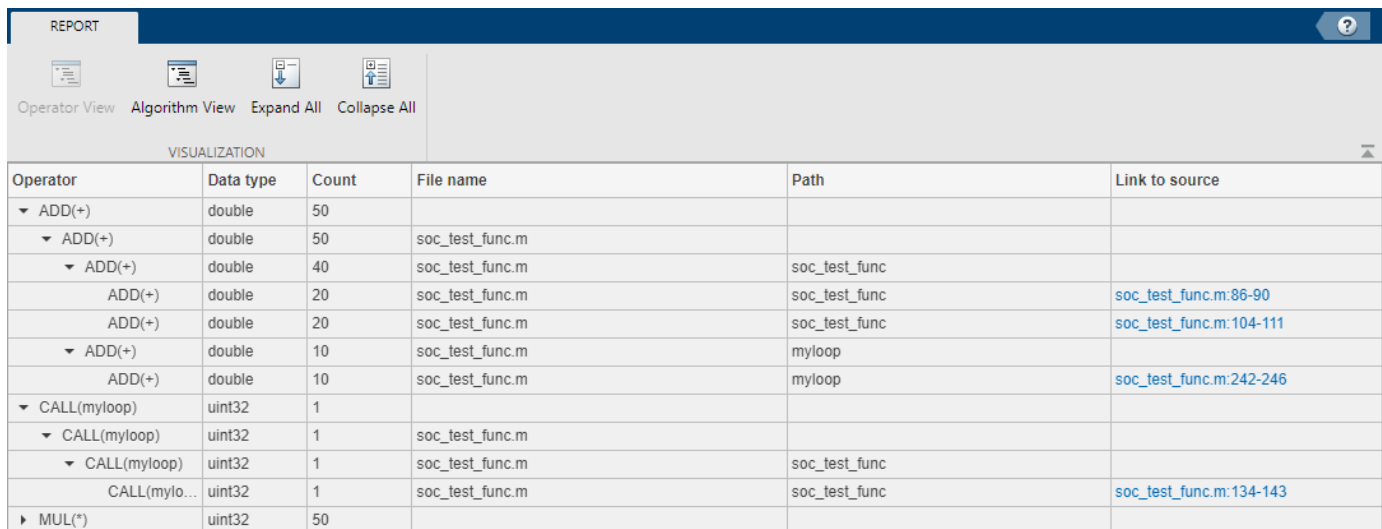
## Analyze Function

`soc_test_func` takes two input arguments of type `uint32`. Use the `FunctionInputs` argument to create a report with 10 and 20 as inputs to the `soc_test_func` function. The report is generated in a folder named `report`.

```
socFunctionAnalyzer('soc_test_func.m', 'FunctionInputs', {10,20}, "Folder", "report");
```

## View Generated Report

After execution, the `socFunctionAnalyzer` function provides a link to the generated report. Click the link titled *Open report viewer*. The report opens in a separate window:



The screenshot shows a window titled "REPORT" with a toolbar containing icons for Operator View, Algorithm View, Expand All, and Collapse All. Below the toolbar is a "VISUALIZATION" section containing a table with the following data:

Operator	Data type	Count	File name	Path	Link to source
▼ ADD(+)	double	50			
▼ ADD(+)	double	50	soc_test_func.m		
▼ ADD(+)	double	40	soc_test_func.m	soc_test_func	
ADD(+)	double	20	soc_test_func.m	soc_test_func	<a href="#">soc_test_func.m:86-90</a>
ADD(+)	double	20	soc_test_func.m	soc_test_func	<a href="#">soc_test_func.m:104-111</a>
▼ ADD(+)	double	10	soc_test_func.m	myloop	
ADD(+)	double	10	soc_test_func.m	myloop	<a href="#">soc_test_func.m:242-246</a>
▼ CALL(myloop)	uint32	1			
▼ CALL(myloop)	uint32	1	soc_test_func.m		
▼ CALL(myloop)	uint32	1	soc_test_func.m	soc_test_func	
CALL(mylo...	uint32	1	soc_test_func.m	soc_test_func	<a href="#">soc_test_func.m:134-143</a>
► MUL(*)	uint32	50			

The result shows that the `ADD` operator is used 50 times with data type `double`. The call to `myloop` executes once with data type `uint32`, and the `MUL` operator is used 50 times with data type `uint32`.

## Input Arguments

### functionName — MATLAB function to analyze

character vector | string scalar

MATLAB function to analyze, specified as a character vector or string scalar that indicates the function name or file name.

Example: `'soc_analyze_FFT_tb.m'`

Data Types: `char` | `string`

### 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: `socFunctionAnalyzer('mySocFunction.m', 'Folder', 'report_sym')`

### FunctionInputs — Inputs for function to be analyzed

cell array

Inputs for the function to be analyzed, specified as the comma-separated pair consisting of 'FunctionInputs' and a cell array. The `socFunctionAnalyzer` function evaluates the function to be analyzed, `functionName`, with these values as inputs. If you do not specify this name-value pair argument, then no arguments are passed to `functionName`.

If `functionName` expects input arguments, then you must specify this name-value pair argument. Otherwise, the `socFunctionAnalyzer` function errors.

Example: 'FunctionInputs',{10,fi(20)}

### **Folder — Folder location of generated report**

current folder (default) | character vector | string scalar

Folder location of generated report, specified as the comma-separated pair consisting of 'Folder' and a character vector or string scalar indicating the folder path. Specify the path to the location for the generated output reports as a full path or relative path.

Example: 'Folder', 'C:/Work/mydir'

### **IncludeOperator — Operators to include in generated report**

all available operators (default) | character vector | string scalar | cell array of strings | cell array of character vectors

Operators to include in the generated report, specified as the comma-separated pair consisting of 'IncludeOperator' and a character vector or string scalar to specify one operator. Use cell array of character vectors or string scalars to specify multiple operators. When you do not specify this name-value pair argument, the `socFunctionAnalyzer` function includes all operators, except for the operators specified by the `ExcludeOperator` name-value pair argument.

Example: 'IncludeOperator', '+'

Example: 'IncludeOperator', {'+', 'IF', 'MUL'}

### **ExcludeOperator — Operators to exclude from generated report**

character vector | string scalar | cell array of strings | cell array of character vectors

Operators to exclude from the generated report, specified as the comma-separated pair consisting of 'ExcludeOperator' and a character vector or string scalar to specify one operator. Use cell array of character vectors or string scalars to specify multiple operators. When you do not specify this name-value pair argument, the `socFunctionAnalyzer` function includes all operators in the report.

Example: 'ExcludeOperator', '-'

Example: 'ExcludeOperator', {'-', 'CALL'}

### **IncludeFunction — Functions to include in generated report**

all functions in function hierarchy (default) | character vector | string scalar | cell array of strings | cell array of character vectors

Functions to include in generated report, specified as the comma-separated pair consisting of 'IncludeFunction' and a character vector or string scalar to specify one function or file name. Use cell array of character vectors or string scalars to specify multiple functions or file names. If you do not specify this name-value pair argument, the `socFunctionAnalyzer` function includes all functions in the report, except for the functions specified by the 'ExcludeFunction' name-value pair argument. Use the 'IncludeFunction' name-value pair when you have a test bench function, and you only want to analyze one of the functions it calls.



Example: 'IncludeFunction','myFunc.m'

Example: 'IncludeFunction',{'myFunc.m','func2'}

### **ExcludeFunction – Functions to exclude from generated report**

character vector | string scalar | cell array of strings | cell array of character vectors

Functions to include in generated report, specified as the comma-separated pair consisting of 'ExcludeFunction' and a character vector or string scalar to specify one function or file name. Use cell array of character vectors or string scalars to specify multiple functions or file names. If you do not specify this name-value pair argument, the socFunctionAnalyzer function includes all functions in the report.

Example: 'ExcludeFunction','myFunc.m'

Example: 'ExcludeFunction',{'myFunc.m','func2'}

### **Verbose – Display verbose messages**

false or 0 (default) | true or 1

Display verbose messages, specified as the comma-separated pair consisting of 'Verbose' and 0 (false) or 1 (true). When this value is 1 (true), the function displays detailed information during the different stages of execution.

Example: 'Verbose',true

## **Output Arguments**

### **report – Operator count raw data**

structure

Function operator count, returned as a structure of five tables:

- OperatorDetailedReport - A fully detailed report per operator
- OperatorAggregatedReport - An aggregated operator view, with one line for each type of operator
- OperatorHierarchicalReport - A hierarchical operator view
- PathAggregatedReport - An aggregated model view
- PathHierarchicalReport - A Hierarchical model view

Each table contains raw data from which the function generates an HTML view, and a link to view the data in a report window. The generated Excel file has five sheets, containing the information from the five tables. For more information about the generated report, see “Using the Algorithm Analyzer Report”.

### **y1, . . . , yn – Analyzed function output (as separate arguments)**

calculated by functionName

Analyzed function output, returned as an output the functionName input function.

## **See Also**

socAlgorithmAnalyzerReport | socModelAnalyzer

**Topics**

“Using the Algorithm Analyzer Report”

**Introduced in R2020a**

# socModelAnalyzer

Estimate number of operations in Simulink model

## Syntax

```
socModelAnalyzer(modelName)
socModelAnalyzer(modelName,Name,Value)
report = socModelAnalyzer(____)
```

## Description

`socModelAnalyzer(modelName)` generates a report with the estimated number of operations in a Simulink model specified by `modelName`.

The function generates the report as a Microsoft Excel spreadsheet and a MAT-file. The function also provides a link to view the report in a separate dialog box.

The report includes information for each mathematical or logical operator in the function, with individual lines for each operator and data type. For example, multiplication with data type `double` and multiplication with data type `uint32` are listed separately. The report lists each instance of the operator as a separate line. The report includes these fields.

- **Path** - The path to the operator within the structural hierarchy of the top function
- **Count** - The number of times the operator is executed in the design
- **Operator** - The operator used
- **DataType** - The data type used for the output of the operator
- **Link** - A link to the location of the operator in the function

For more information, see “Using the Algorithm Analyzer Report”.

`socModelAnalyzer(modelName,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'IncludeOperator','+'` specifies that the generated report only includes `'+'` operator counts.

`report = socModelAnalyzer(____)` returns a structure of tables that contain report information. Specify any of the input argument combinations from previous syntaxes.

## Examples

### Analyze Resources in a Model

Calculate the number of operators in the model `testmdl.slx`.

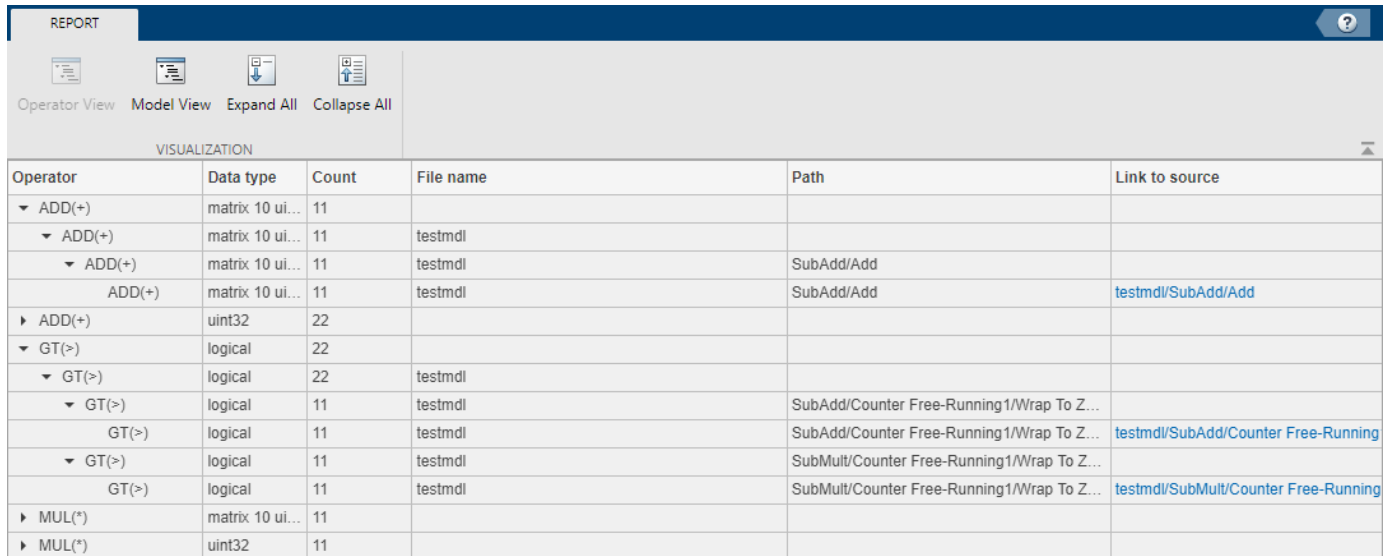
### Analyze the Model

Count operators in `testmdl`, and generate a report in a folder named `report`.

```
socModelAnalyzer('testmdl.slx','Folder','report');
```

## View the Generated Report

After execution, the `socModelAnalyzer` function provides a link to the generated report. Click the link titled *Open report viewer*. The report opens in a separate window:



The screenshot shows a web-based report viewer with a dark blue header labeled 'REPORT' and a help icon. Below the header is a toolbar with icons for 'Operator View', 'Model View', 'Expand All', and 'Collapse All'. The main content area is titled 'VISUALIZATION' and contains a table with the following data:

Operator	Data type	Count	File name	Path	Link to source
▼ ADD(+)	matrix 10 ui...	11			
▼ ADD(+)	matrix 10 ui...	11	testmdl		
▼ ADD(+)	matrix 10 ui...	11	testmdl	SubAdd/Add	
ADD(+)	matrix 10 ui...	11	testmdl	SubAdd/Add	<a href="#">testmdl/SubAdd/Add</a>
▶ ADD(+)	uint32	22			
▼ GT(>)	logical	22			
▼ GT(>)	logical	22	testmdl		
▼ GT(>)	logical	11	testmdl	SubAdd/Counter Free-Running1/Wrap To Z...	
GT(>)	logical	11	testmdl	SubAdd/Counter Free-Running1/Wrap To Z...	<a href="#">testmdl/SubAdd/Counter Free-Running</a>
▼ GT(>)	logical	11	testmdl	SubMult/Counter Free-Running1/Wrap To Z...	
GT(>)	logical	11	testmdl	SubMult/Counter Free-Running1/Wrap To Z...	<a href="#">testmdl/SubMult/Counter Free-Running</a>
▶ MUL(*)	matrix 10 ui...	11			
▶ MUL(*)	uint32	11			

The result shows that the ADD operator is used 11 times with data type `matrix 10 uint32`, and 22 times with data type `uint32`. The GT (greater than) operator was used 22 times total with data type `logical`: 11 times from **SubAdd** model, and 11 times from **SubMult** model. The MUL operator is used 11 times with data type `uint32`, and 11 times with `matrix 10 uint32`.

## Input Arguments

### `modelName` — Simulink model to analyze

character vector | string

Simulink model to analyze, specified as a character vector or string scalar.

Example: `'soc_analyze_FFT_top.slx'`

Data Types: `char` | `string`

### 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: `socModelAnalyzer('mySocModel.slx', 'Folder', 'report_sym')`

### `Folder` — Folder location of generated report

current folder (default) | character vector | string scalar

Folder location of generated report, specified as the comma-separated pair consisting of `'Folder'` and a character vector or string scalar indicating the folder path. Specify the path to the location for the generated output reports as a full path or relative path.

Example: `'Folder', 'C:/Work/mydir'`

**IncludeOperator — Operators to include in generated report**

all available operators (default) | character vector | string scalar | cell array of strings | cell array of character vectors

Operators to include in the generated report, specified as the comma-separated pair consisting of 'IncludeOperator' and a character vector or string scalar to specify one operator. Use cell array of character vectors or string scalars to specify multiple operators. When you do not specify this name-value pair argument, the socModelAnalyzer function includes all operators, except for the operators specified by the ExcludeOperator name-value pair argument.

Example: 'IncludeOperator','+'

Example: 'IncludeOperator',{'+', 'IF', 'MUL'}

**ExcludeOperator — Operators to exclude from generated report**

character vector | string scalar | cell array of strings | cell array of character vectors

Operators to exclude from the generated report, specified as the comma-separated pair consisting of 'ExcludeOperator' and a character vector or string scalar to specify one operator. Use cell array of character vectors or string scalars to specify multiple operators. When you do not specify this name-value pair argument, the socModelAnalyzer function includes all operators in the report.

Example: 'ExcludeOperator','-'

Example: 'ExcludeOperator',{'-', 'CALL'}

**IncludeBlockPath — Models to include in generated report**

all models or blocks in top model hierarchy (default) | character vector | string | cell array of strings | cell array of character vectors

Models or blocks to include in generated report, specified as the comma-separated pair consisting of 'IncludeBlockPath' and a character vector or string scalar to specify one block or model. Use cell array of character vectors or string scalars to specify multiple blocks or models. If you do not specify this name-value pair argument, the socModelAnalyzer function includes all models and blocks in the report, except for the blocks specified by the 'ExcludeBlockPath' name-value pair argument. Use the 'IncludeBlockPath' name-value pair when you have a test bench model, and you only want to analyze one of the models it includes.

Example: 'IncludeBlockPath','myModel.slx'

Example: 'IncludeBlockPath',{'myModel.slx', 'myIfft'}

**ExcludeBlockPath — Models to exclude from generated report**

character vector | string | cell array of strings | cell array of character vectors

Models or blocks to include in generated report, specified as the comma-separated pair consisting of 'ExcludeBlockPath' and a character vector or string scalar to specify one block or model. Use cell array of character vectors or string scalars to specify multiple blocks or models. If you do not specify this name-value pair argument, the socModelAnalyzer function includes all models and blocks in the report.

Example: 'ExcludeBlockPath','myOtherModel.slx'

Example: 'ExcludeBlockPath',{'myOtherModel.slx', 'myIfft'}

**Verbose — Display verbose messages**

false or 0 (default) | true or 1

Display verbose messages, specified as the comma-separated pair consisting of 'Verbose' and 0 (false) or 1 (true). When this value is 1 (true), the function displays detailed information during the different stages of execution.

Example: 'Verbose', true

## Output Arguments

### **report** — Operator count raw data

structure

Model operator count, returned as a structure of five tables:

- `OperatorDetailedReport` - A fully detailed report per operator
- `OperatorAggregatedReport` - An aggregated operator view, with one line for each type of operator
- `OperatorHierarchicalReport` - A hierarchical operator view
- `PathAggregatedReport` - An aggregated model view
- `PathHierarchicalReport` - A Hierarchical model view

Each table contains raw data from which the function generates an HTML view, and a link to view the data in a report window. The generated Excel file has five sheets, containing the information from the five tables. For more information about the generated report, see “Using the Algorithm Analyzer Report”.

## Limitations

- This function does not support AUTOSAR Blockset blocks or models.
- This function does not support Simulink send and receive messages.

## See Also

`socAlgorithmAnalyzerReport` | `socFunctionAnalyzer`

### Topics

“Using the Algorithm Analyzer Report”

**Introduced in R2020a**

# socExportReferenceDesign

Export custom reference design for HDL Workflow Advisor

## Syntax

```
socExportReferenceDesign(topmodelName)
socExportReferenceDesign(topmodelName,Name,Value)
```

## Description

`socExportReferenceDesign(topmodelName)` exports a custom reference design from an SoC Blockset model with name `topmodelName`. To create an SoC Blockset model, you must perform one of these actions.

- Create a model using an SoC Blockset template. For more information, see “Use Template to Create SoC Model”.
- Open Simulink. On the **Apps** tab click **System on Chip (SoC)**.
- In an existing Simulink model, click **Model Settings** in the **Modeling** tab. In the left pane, select **Hardware Implementation**. Then, set **Hardware board** to a supported SoC board. For a list of supported SoC boards, see “Supported Third-Party Tools and Hardware”.

Use this exported design with **HDL Workflow Advisor** (requires HDL Coder™ license). Use this function to eliminate the manual steps for creating a custom reference design, as described in “Custom Reference Design” (HDL Coder). Use the exported reference design in the IP core generation workflow with the HDL Workflow advisor. For more information, see “Hardware Software Co-Design Basics” (HDL Coder).

For more information about the **HDL Workflow Advisor** app, see “Getting Started with the HDL Workflow Advisor” (HDL Coder).

To use this function, you must first install Xilinx Vivado® or Intel Quartus®.

`socExportReferenceDesign(topmodelName,Name,Value)` specifies options using one or more name-value pair arguments.

## Examples

### Export Custom Reference Design from SoC Model

Export a custom reference design from the `soc_image_rotaion.slx` model.

```
socExportReferenceDesign('soc_image_rotation')
```

### Export Reference Design Using Specific Arguments

Export a custom reference design from the `soc_hws_stream_top` model.

- Exclude the DUT named "FPGA Algorithm Wrapper" from the reference design.
- Place the generated output in folder C:/Work.
- Generate a board definition file with board name "My ZC706 Board". This name appears in the **Target platform** menu in the **HDL Workflow Advisor** app.
- Generate reference design definition file with the design name My ZC706 Design.

```
socExportReferenceDesign('soc_hws_stream_top',...
    'DUTName','FPGA Algorithm Wrapper',...
    'Folder','C:/Work',...
    'TargetPlatform','My ZC706 Board',...
    'ReferenceDesign','My ZC706 Design')
```

## Input Arguments

### topmodelName — Name of top Simulink model

character vector | string scalar

Name of the top Simulink model, specified as a character vector or string scalar. The reference design is exported from the `topmodelName` model. This model must be an SoC Blockset model.

Example: 'soc\_hw\_sw\_stream\_top' specifies the model with name 'soc\_hw\_sw\_stream\_top'.

Data Types: char | string

### 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:

```
socExportReferenceDesign('soc_image_rotation','Folder','refDesignFolder')
exports a reference design from the model soc_image_rotation, and places the generated files in
a folder named refDesignFolder.
```

### DUTName — Name of DUT subsystem to exclude from reference design

inferred (default) | character vector | string scalar

Name of DUT subsystem to exclude from reference design, specified as a character vector or string scalar. When there is one DUT in the model, the function infers the `DUTName` and sets it as the name of the DUT in the model. You must specify this name-value pair argument when the FPGA model has more than one DUT.

Example: 'soc\_image\_rotation\_fpga/ImageRotation'

Data Types: char | string

### Folder — Folder location for exported reference design files

`topmodelName_refdesign` (default) | character vector | string scalar

Folder location for the exported reference design files, specified as a character vector or string scalar. When not specified, the files are placed in a folder named `topmodelName_refdesign`, where `topmodelName` is the name of the model.

Example: 'C:/Work/refDesign'

Data Types: char | string



**TargetPlatform — Name of target platform**

same as SoC model (default) | character vector | string scalar

Name of the target platform, specified as the comma-separated pair consisting of 'TargetPlatform' and a character vector or string scalar. When you do not specify this value, the name of the target platform matches the **Hardware Board** parameter value in the SoC model configuration parameters. In the **HDL Workflow Advisor** tool, this target platform name appears as *TargetPlatform* (generated by SoC Blockset), where *TargetPlatform* is the value for this name-value pair argument.

Example: 'Xilinx Zynq ZC706 evaluation kit'

Data Types: char | string

**ReferenceDesign — Name of generated reference design**

*topModelName* model (default) | character vector | string scalar

Name of the generated reference design, specified as the comma-separated pair consisting of 'ReferenceDesign' and a character vector or string scalar. When you do not specify this value, the name of the generated reference design is *topModelName* model, where *topModelName* is specified by the input *topModelName*.

Example: 'My ZC706 Design'

Data Types: char | string

**See Also**

**SoC Builder** | hdladvisor

**Topics**

“Custom Reference Design” (HDL Coder)

“Generate SoC Design”

**Introduced in R2020a**

# socAlgorithmAnalyzerReport

Open algorithm analysis report

## Syntax

```
socAlgorithmAnalyzerReport(reportfile)
```

## Description

`socAlgorithmAnalyzerReport(reportfile)` opens the specified report generated by the `socFunctionAnalyzer` or `socModelAnalyzer` function. The report opens in a separate window titled Algorithm Analyzer Report.

## Examples

### Open Algorithm Analysis Report

Use the `socFunctionAnalyzer` function to generate a report. Then, open the generated report:

```
socFunctionAnalyzer('soc_test_func.m', 'FunctionInputs', {10,20}, 'Folder', 'report');  
socAlgorithmAnalyzerReport('report/soc_test_func.mat');
```

## Input Arguments

### reportfile — Path to report file

character vector | string scalar

Path to report file, specified as a character vector or string scalar that indicates the name of a MAT-file, generated by the `socFunctionAnalyzer` or `socModelAnalyzer` function.

Example: 'report/soc\_test\_func.mat'

Data Types: char | string

## See Also

`socFunctionAnalyzer` | `socModelAnalyzer`

## Topics

“Using the Algorithm Analyzer Report”

**Introduced in R2020a**

# Objects

---

## 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

### **Data Type — 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 a 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','username','r
```

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 =
    1x0 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', 'username', 'r
```

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 =
    1x1 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

<code>addSource</code>	Add an input source to a data recording session
<code>removeSource</code>	Remove input source from data recording session
<code>setup</code>	Set up hardware for data recording
<code>record</code>	Record data from hardware using data recorder object
<code>isRecording</code>	Get data recording status
<code>save</code>	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','username','r
```

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 =
```

```
1x0 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 =
```

```

    1x1 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\_aximaster.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 4-27.

### **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 4-27.

### **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    2.0000    2.2500
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 `JTAGCableType` 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

---

## Logic Analyzer

Visualize, measure, and analyze transitions and states over time

### Description

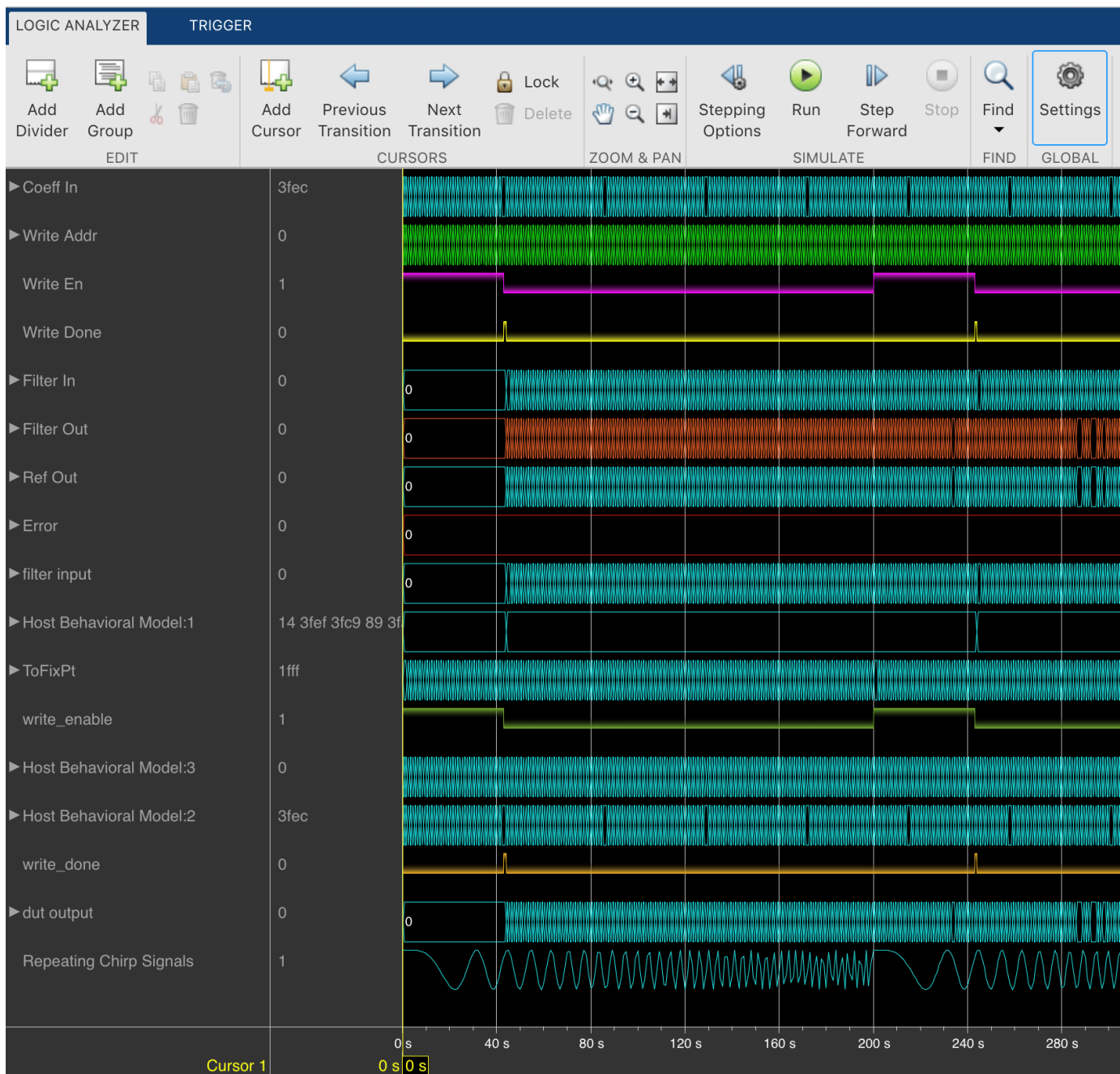
The **Logic Analyzer** is a tool for visualizing and inspecting signals and states in your Simulink model. Using the **Logic Analyzer**, you can:

- 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
<b>Ctrl+X</b>	Cut	Wave is selected
<b>Ctrl+C</b>	Copy	Wave is selected
<b>Ctrl+V</b>	Paste	Wave is selected
<b>Delete</b>	Delete	Wave is selected
<b>Ctrl+-</b>	Zoom out	Always
<b>Shift+Ctrl+-</b>	Zoom out around active cursor	Always
<b>Ctrl++</b>	Zoom in	Always
<b>Shift+Ctrl++</b>	Zoom in around active cursor	Always
<b>Shift+Ctrl+C</b>	Move display to active cursor	When cursor is not in the display range
<b>Space</b>	Zoom out full	Always
<b>Tab, Right Arrow</b>	Next transition	Digital format wave is selected
<b>Shift+Tab, Left Arrow</b>	Previous transition	Digital format wave is selected
<b>Ctrl+A</b>	Select all waves	Always
<b>Up Arrow</b>	Select wave above selected	Wave is selected
<b>Down Arrow</b>	Select wave below selection	Wave is selected
<b>Ctrl+Up Arrow</b>	Move selected waves up	Wave is selected
<b>Ctrl+Down Arrow</b>	Move selected waves down	Wave is selected
<b>Escape</b>	Unselect all signals	Wave is selected
<b>Page Up</b>	Scroll up	Always
<b>Page Down</b>	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.

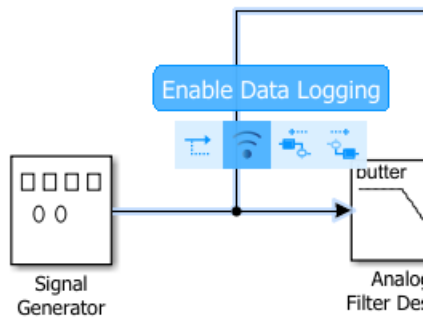
To visualize referenced models, you must open the Logic Analyzer from the referenced model. You should see the name of the referenced model in the Logic Analyzer toolbar.

## Examples

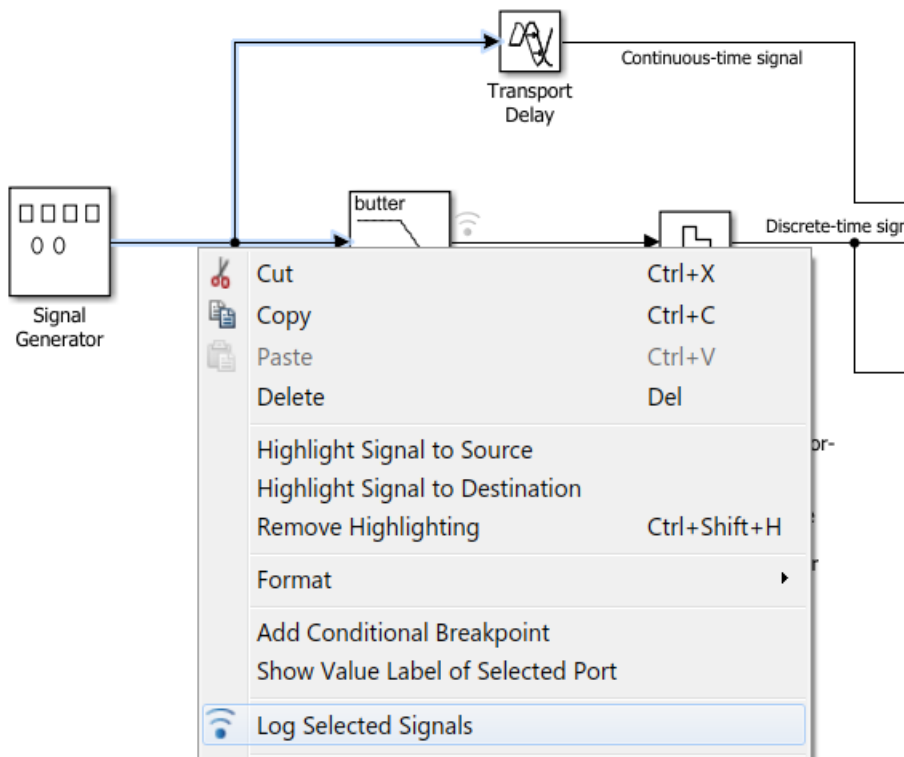
### Select Signals to Analyze

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, on the **Signal** tab, select the **Log Signals** button.

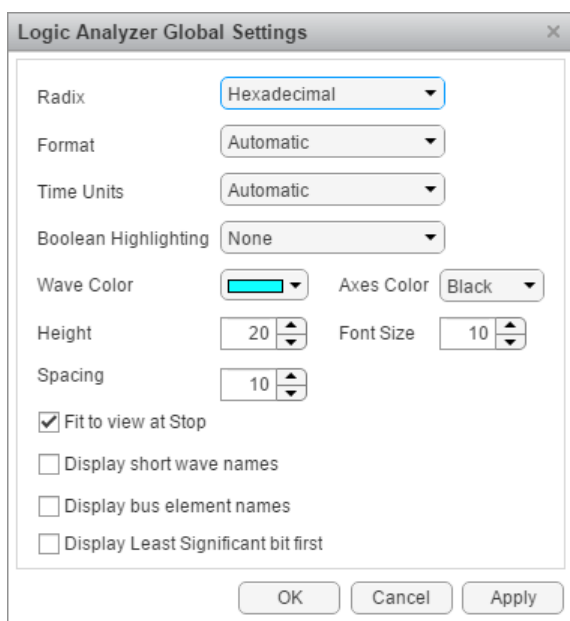


To visualize data in the Logic Analyzer, you must enable signal logging for the model. (Logging is on by default.) To enable signal logging, open **Model Settings** from the toolstrip, navigate to the **Data Import/Export** pane, and select **Signal logging**.

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.

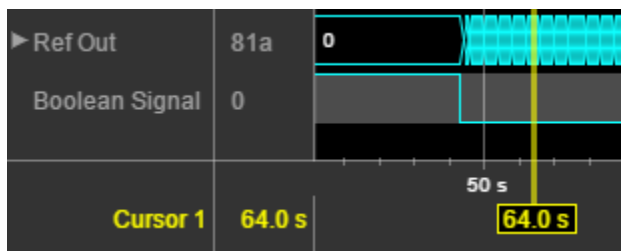
- **Analog** — Displays values as an analog plot
- **Digital** — Displays values as digital transitions

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**

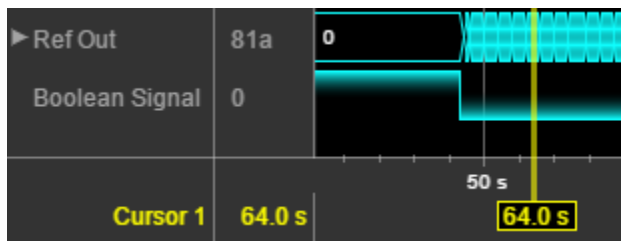
Set the **Boolean Highlighting** to one of the following:

- **None**
- **Rows** — Adds a highlighted background for the entire Boolean signal row.



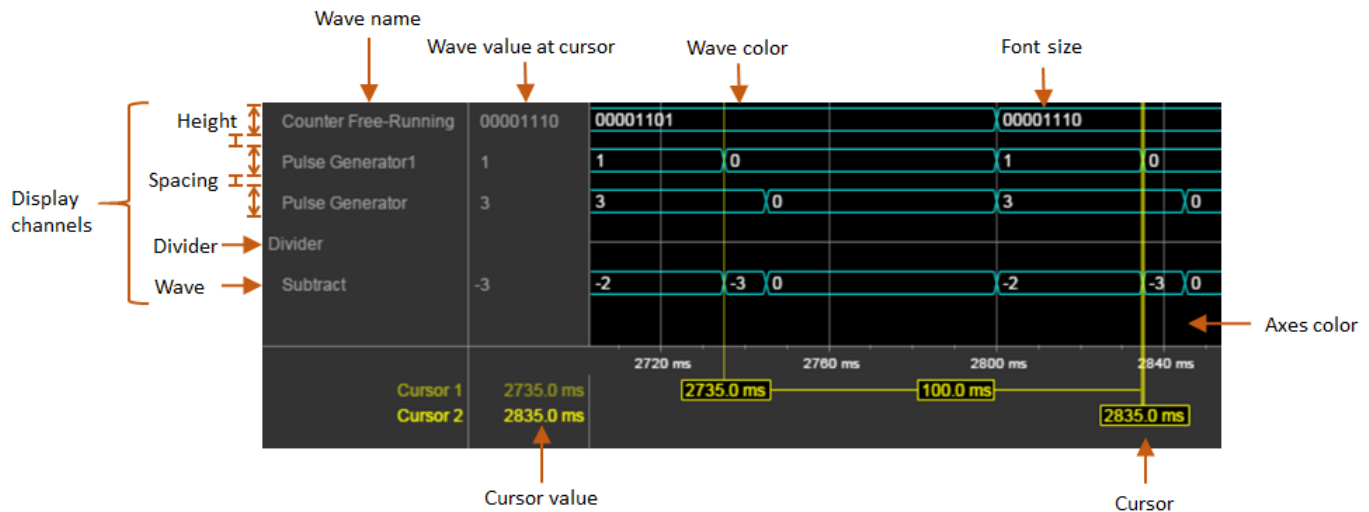
Select **Highlight boolean values** to add highlighting to Boolean signals.

- **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.

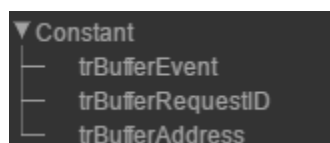
To display the short names of waves without path information, select **Display short wave names**.

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:

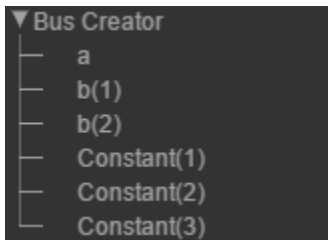
- 1 Add the bus signal for logging.
- 2 In the Logic Analyzer settings, select the **Display bus element names** check box.
- 3 Run the simulation.

When you expand the bus signals, you will see the bus signal names.

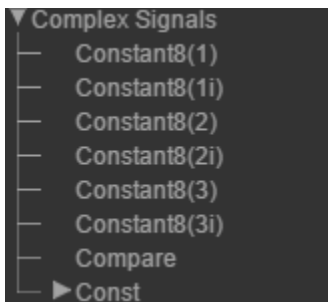


Some special situations:

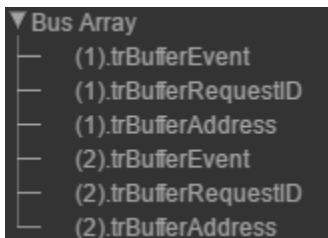
- If the signal has no name, the Logic Analyzer shows the block name instead.
- If the bus is a bus object, the Logic Analyzer shows the bus element names specified in the Bus Object Editor.
- If one of the bus elements contains an array, each element of the array is appended with the element index.



- If a bus element contains an array with complex elements, the real and complex values (i) are split.

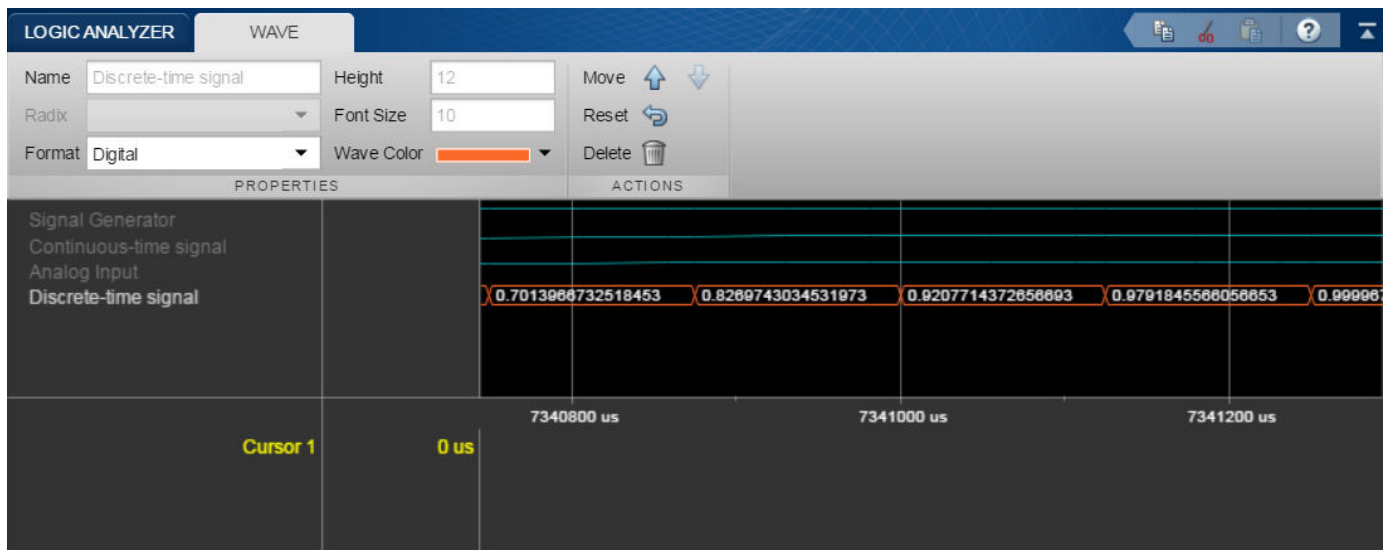


- Bus signals passed through a Gain block are labeled Gain(1), Gain(2),...Gain(n).
- If the bus contains an array of buses, the Logic Analyzer prepends the element name with the bus array index.



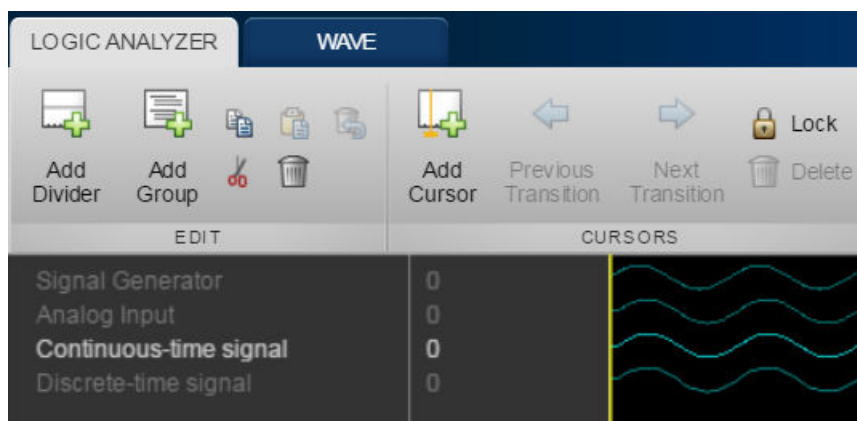
### Modify Individual Wave Settings

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**.



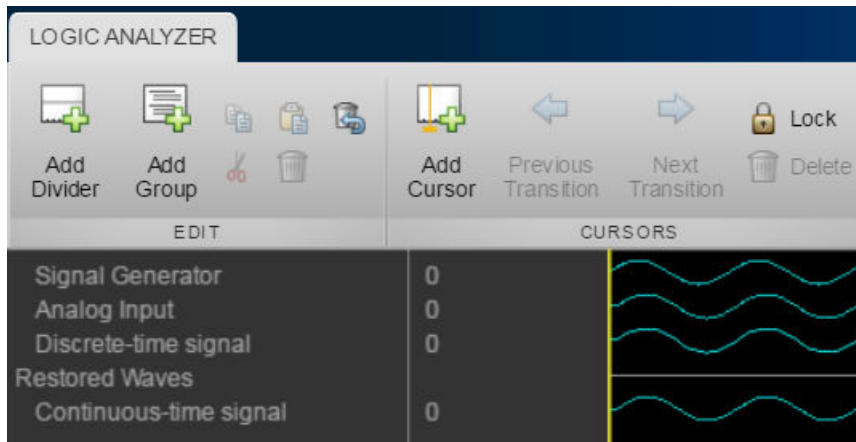
### Delete and Restore Waves

- 1 Open the **Logic Analyzer** and select a wave by clicking the wave name.



- 2 From the **Logic Analyzer** toolbar, click . The wave is removed from the **Logic Analyzer**.
- 3 To restore the wave, from the **Logic Analyzer** toolbar, click .

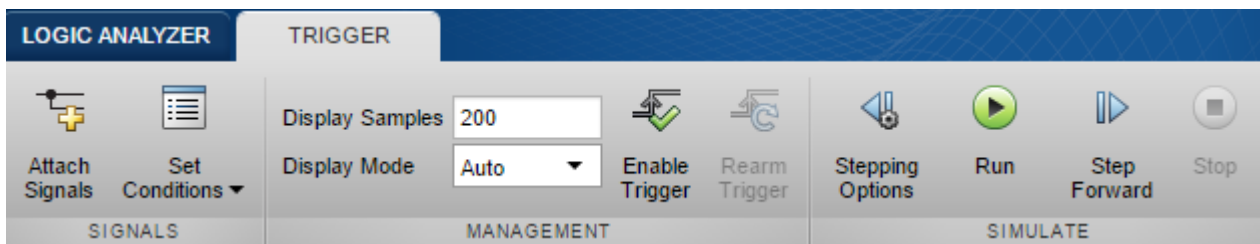
A divider named **Restored Waves** is added to the bottom of your channels, with all deleted waves placed below it.



### 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**.

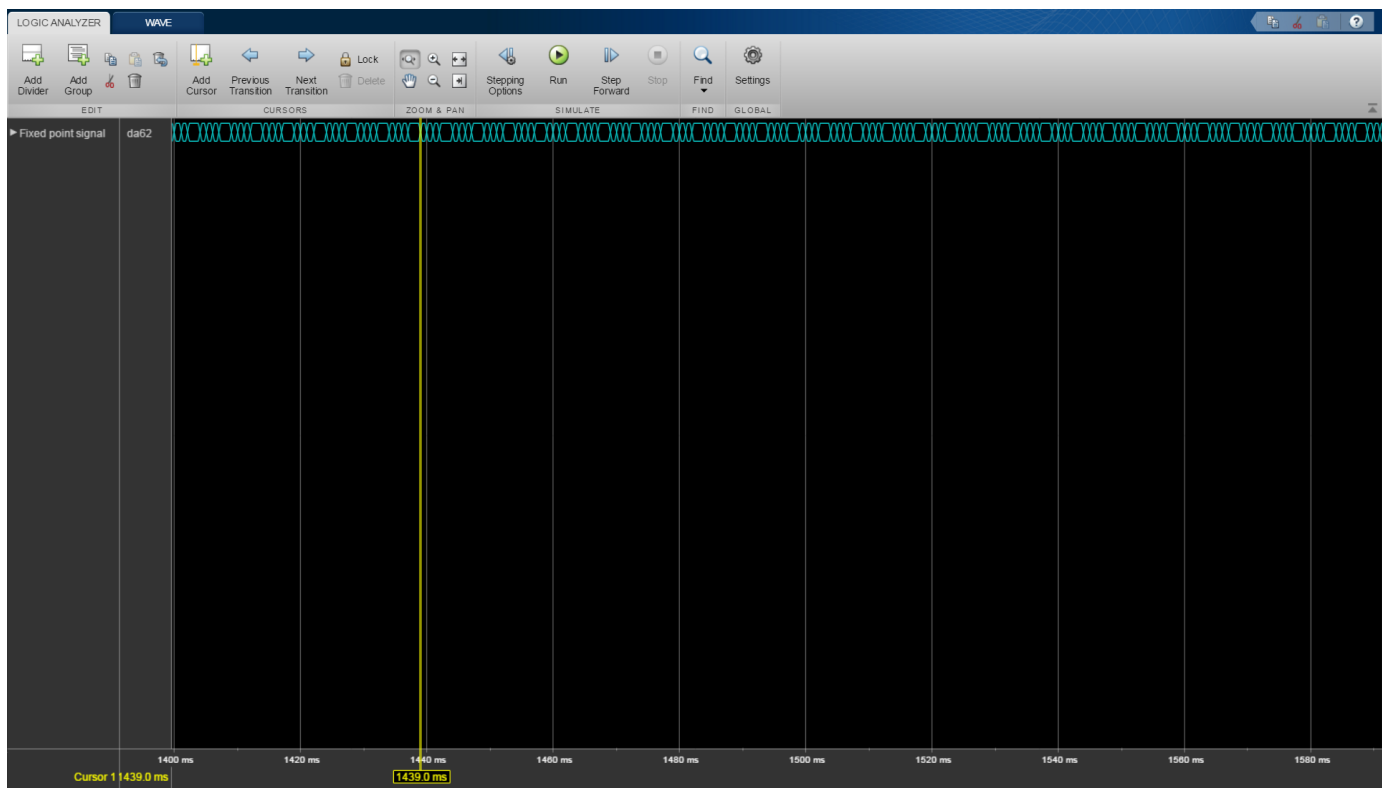
If you add multiple signals to the trigger, control the trigger logic using the **Operator** option:

- 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.

- 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.

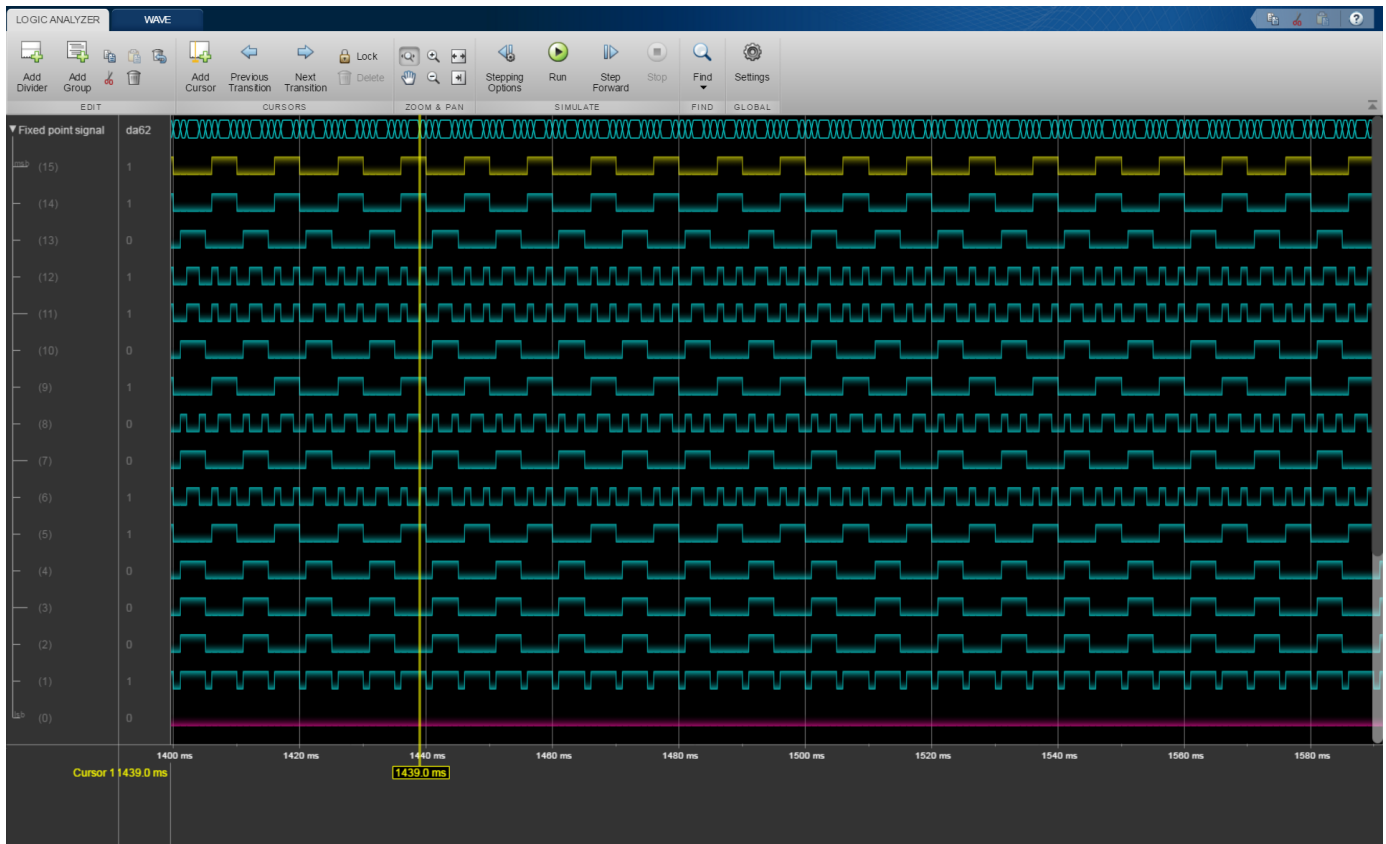
### View Bit-Expanded Wave and Reverse Display Order of Bits

The **Logic Analyzer** enables you to bit-expand fixed-point and integer waves.

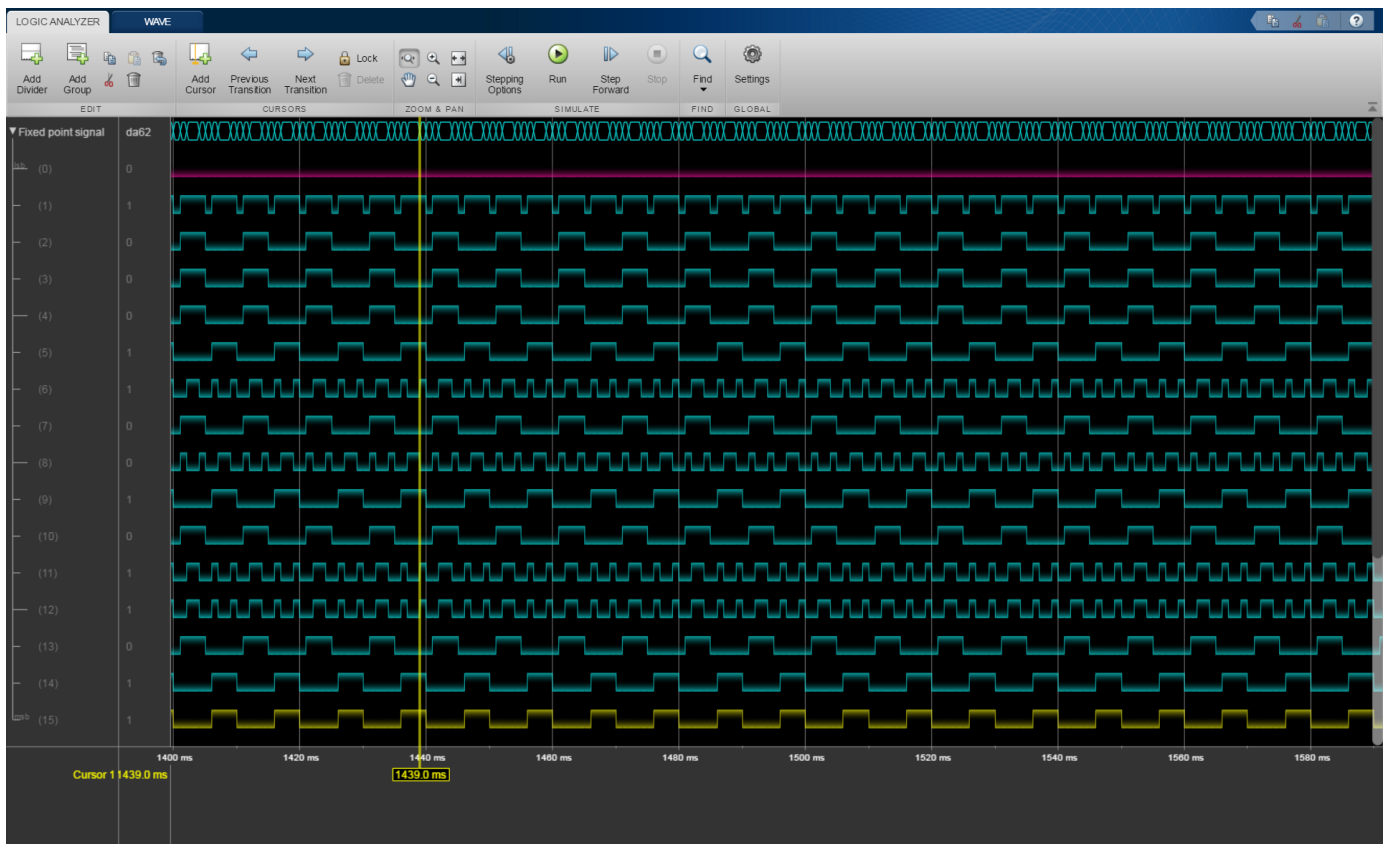


- In the **Logic Analyzer**, click the arrow next to a fixed-point or integer wave to view the bits.

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

### Logging Settings

- If you enable the configuration parameter **Log Dataset data to file**, you cannot stream logged data to the **Logic Analyzer**.
- Signals marked for logging using `Simulink.sdi.markSignalForStreaming` or visualized with a Dashboard Scope do not appear on the **Logic Analyzer**.
- You cannot visualize Data Store Memory block signals in the **Logic Analyzer** if you set the **Log data store data** parameter to on.

### Input Signal Limitations

- Signals marked for logging for the **Logic Analyzer** must have fewer than 8000 samples per simulation step.
- The **Logic Analyzer** does not support frame-based processing.
- Integers larger than 64 bits and fixed-point signals larger than 128 bits are not supported.
- 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 data output.

## Graphical Settings

- While the simulation is running, you cannot zoom, pan, or modify the trigger.
- To visualize constant signals, in the settings, you must set the **Format** to **Digital**. Constants marked for logging are visualized as a continuous transition.

## Supported Simulation Modes

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	You cannot use the <b>Logic Analyzer</b> to visualize signals in Model blocks with <b>Simulation mode</b> set to <b>Accelerator</b> .
Rapid Accelerator	Yes	Data is not available in the <b>Logic Analyzer</b> 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 <b>Logic Analyzer</b> : <ul style="list-style-type: none"> <li>• Multi-instance model reference signals</li> <li>• Nonvirtual bus signals</li> </ul>
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”.

## See Also

### Objects

### Topics

“Programmable FIR Filter for FPGA” (HDL Coder)

“Packet-Based ADS-B Transceiver”

“Log Simulation Output for States and Data” (Stateflow)

“View Stateflow States in the Logic Analyzer” (Stateflow)

### Introduced in R2016b



# Memory Mapper

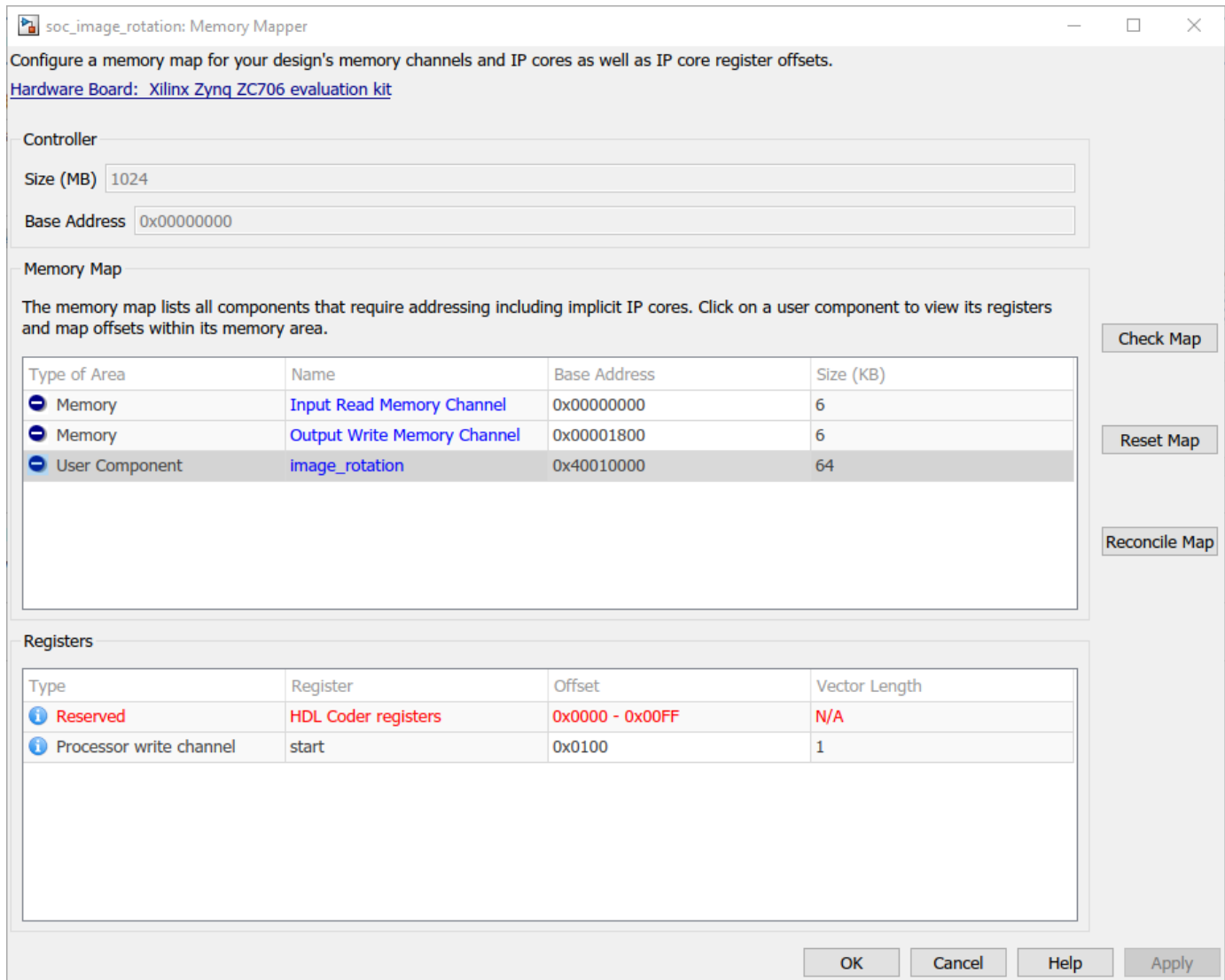
Configure memory map for SoC application

## 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.



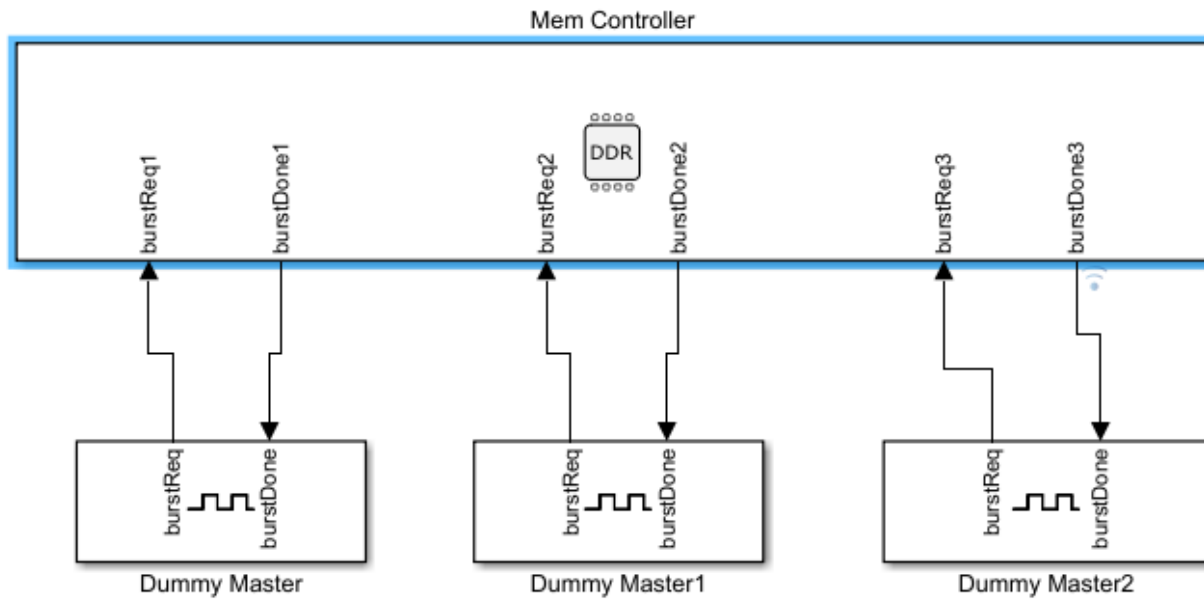
## Open the Memory Mapper

- 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**.

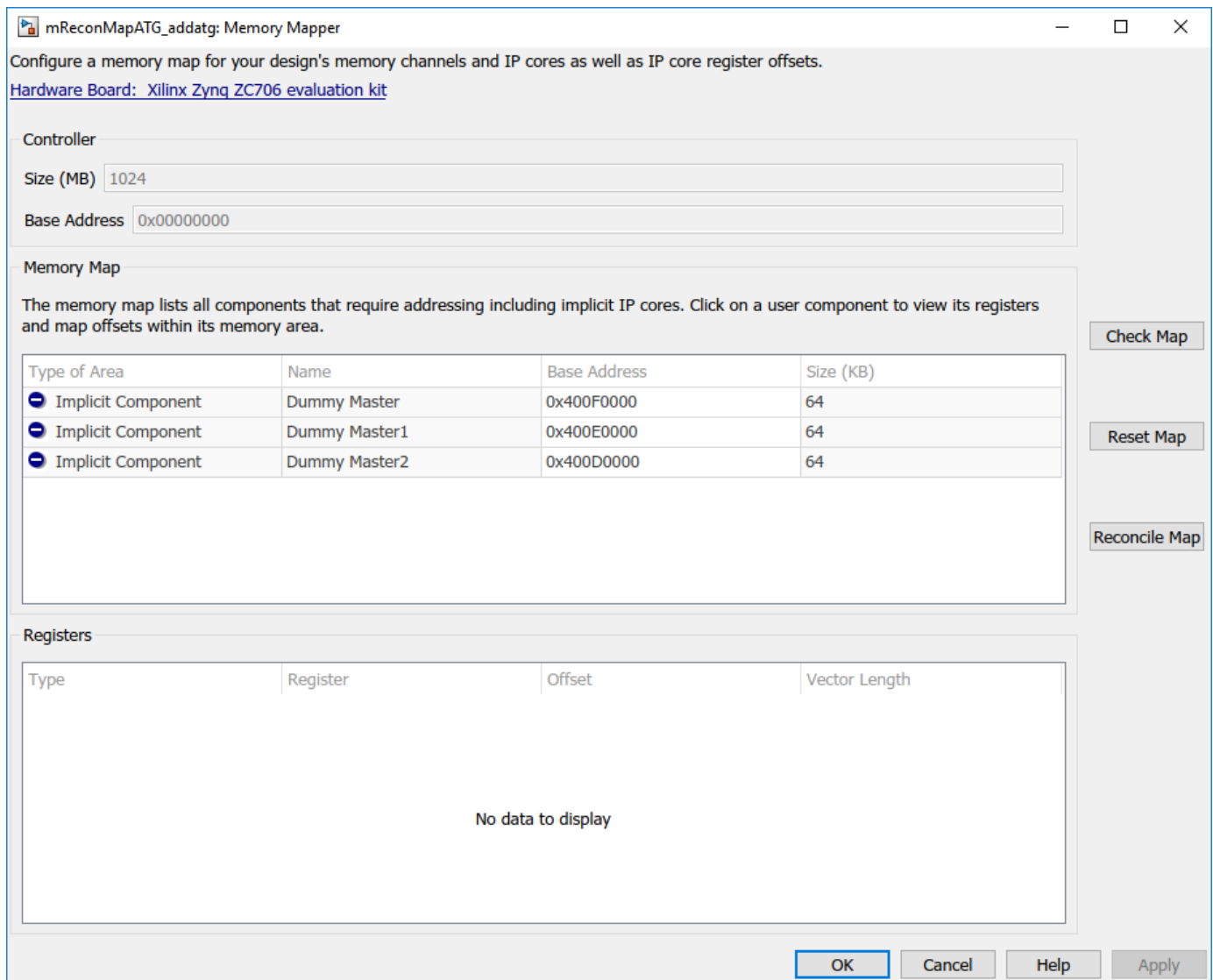
## Examples

### Reconcile Model with Memory Map

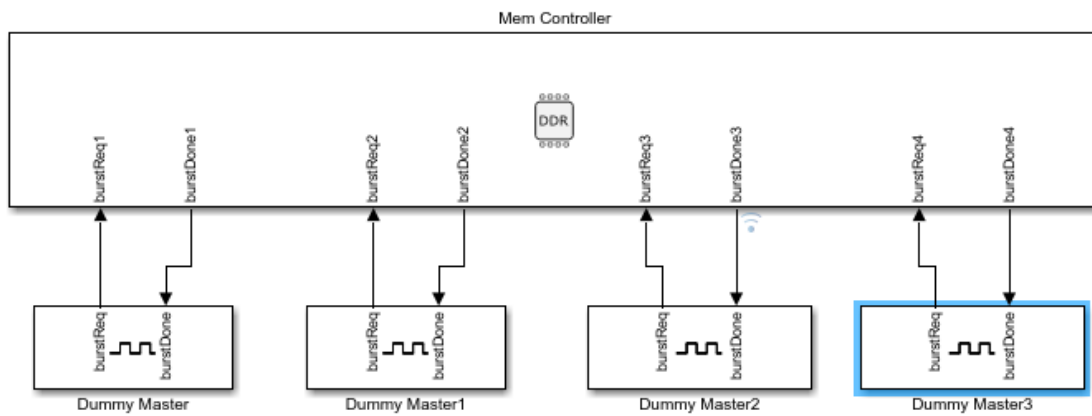
Consider a model with three masters (represented by Memory Traffic Generator blocks), connected to a Memory Controller block.



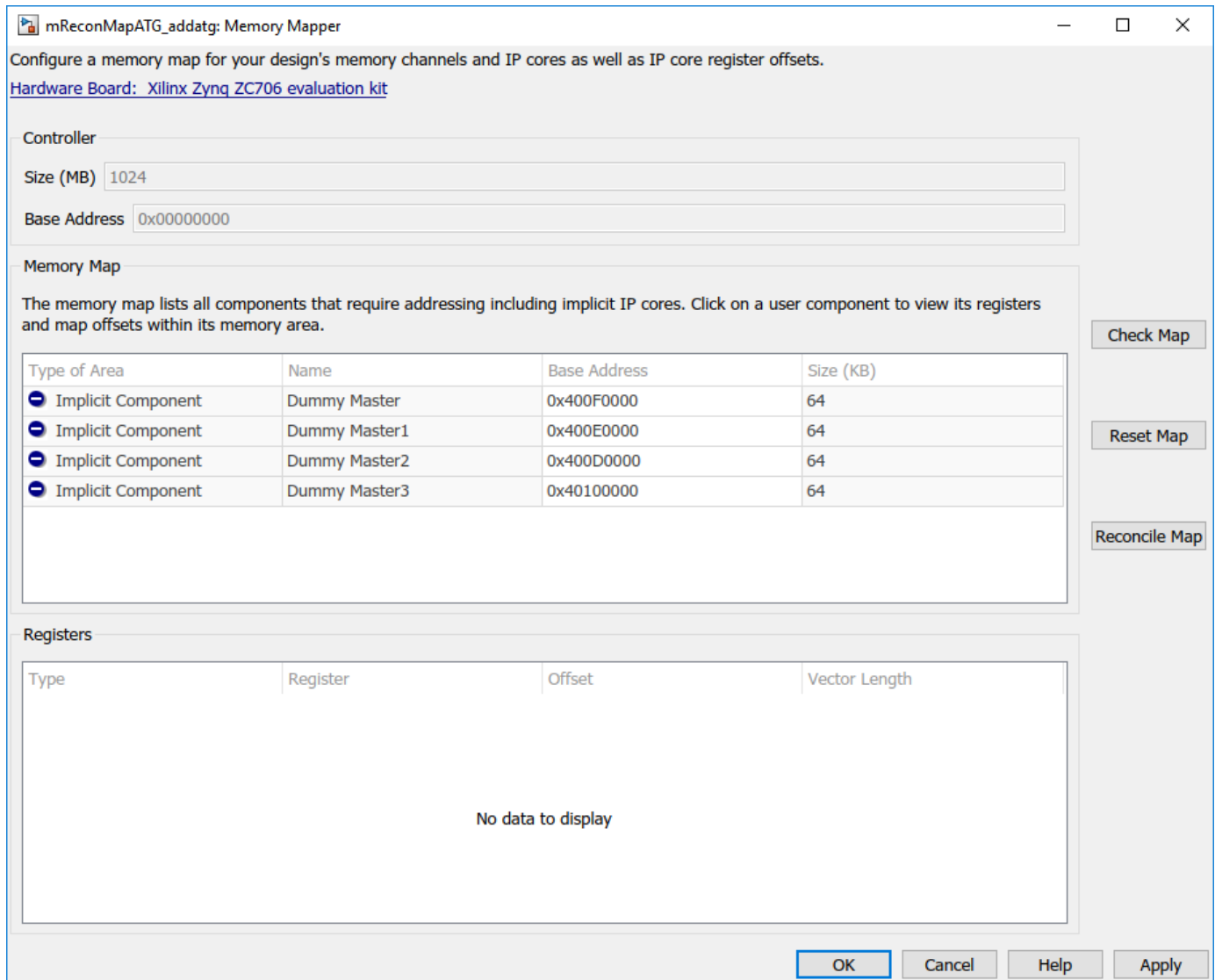
To open the **Memory Mapper** tool, first open the Configuration Parameters dialog box, and then select **Hardware Implementation** from the left pane. Under **Target hardware resources**, select **FPGA design (top-level)** and click **View/Edit Memory Map**.



The **Memory Mapper** lists the three masters in the design. Edit their base addresses as per your requirements. Add another channel to your model.

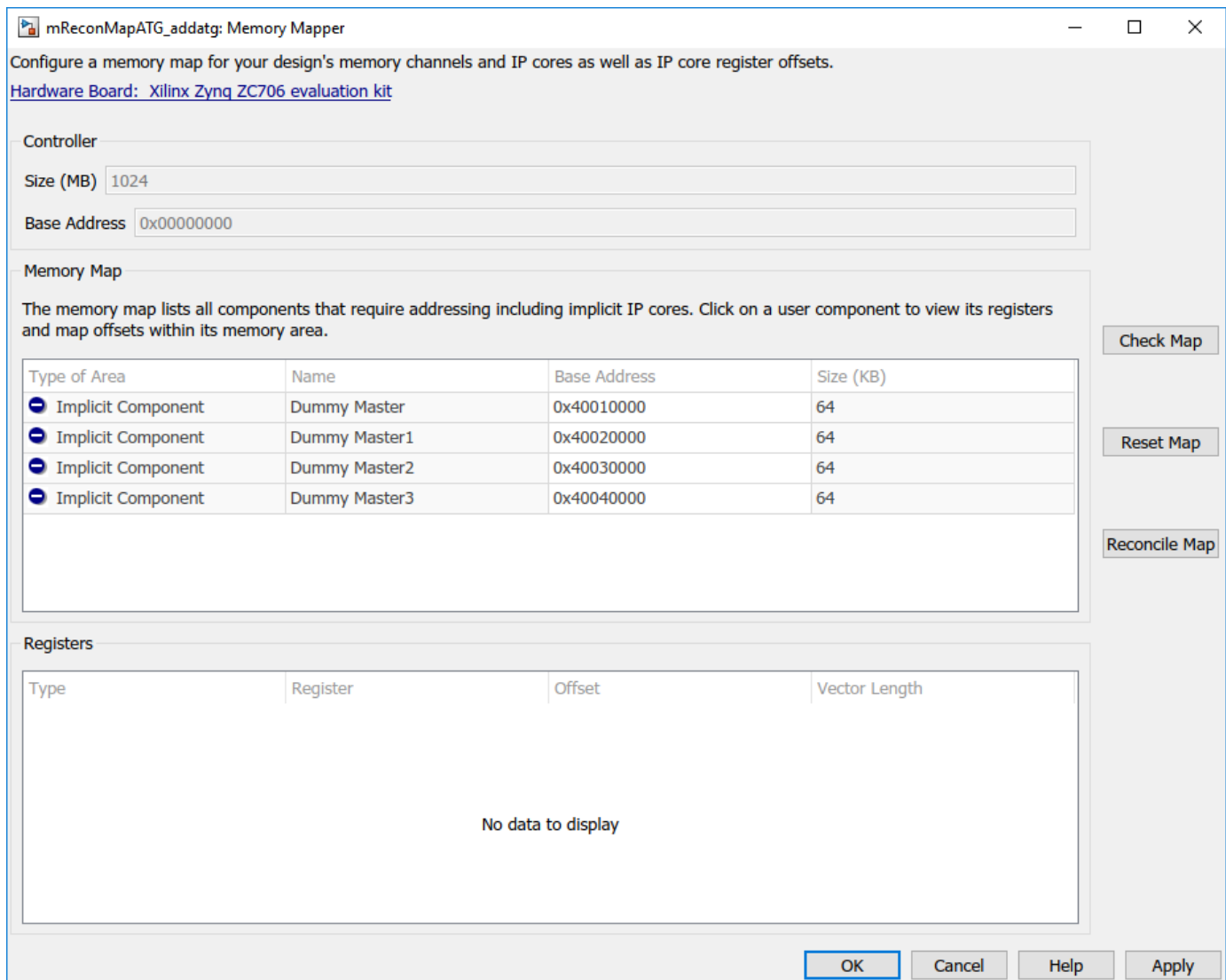


The model consists of four memory channels, while the **Memory Map** section shows only three. To resolve this conflict, click **Reconcile Map**. This adds another line, which represents the added channel, to the memory map table.



### Reset Map

Click **Reset Map** to create a new, autogenerated map. The base addresses of the channels are reset to a default value.



## 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**



# Task Mapping

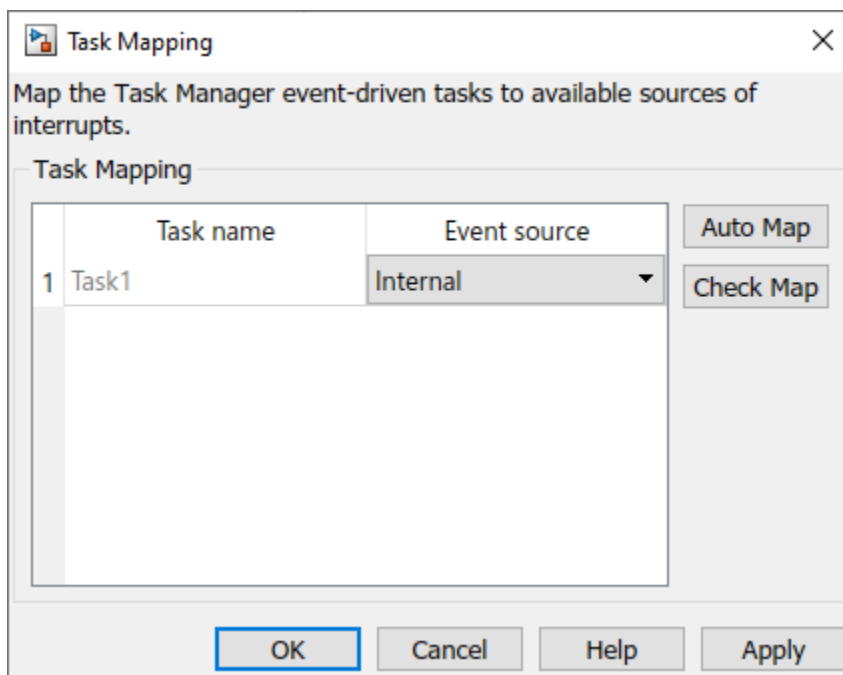
Map tasks in the SoC to interrupt service routines on the hardware board

## Description

View and edit the map of tasks in the SoC to interrupt service routines (ISRs) on the hardware board.

Using the **Task Mapper** tool, you can:

- View and edit the assignment of tasks to MCU interrupts.
- Check the task to interrupt map of your model for any conflicts between tasks.
- Automatically map tasks to ISRs.



## Open the Task Mapping

- In the Configuration Parameters dialog box, select **Hardware Implementation** from the left pane. Under **Hardware board settings > Design mapping**, click **View/Edit Task Map**.
- In the SoC Builder tool, in the **Review Memory and Interrupt Map** section, click **View/Edit Task Map**.

## Parameters

### Task Mapping

#### **Task Name — Name of the tasks from Task Managers**

task name

This property is read-only.

This parameter is the name of the tasks assigned in Task Manager for the current processor.

#### **Event Source — Base address of memory**

Internal (default) | Undefined | Interrupt name

This property is read-only.

This parameter shows the tag name of the interrupt service routine (ISR). This value is a drop down list for that given hardware.

### Task Mapping

#### **Auto Map — Automatically map tasks to ISRs**

button

Automatically map tasks in the current model to the most appropriate interrupt service routines (ISRs) available for the selected hardware board.

#### **Check Map — Check task to ISR map**

button

Check that each task maps to a unique interrupt service routine (ISR) source. Two tasks cannot be mapped to the same ISR source.

### See Also

Task Manager

**Introduced in R2020b**

# Peripheral Configuration

Map peripherals in the SoC model to peripheral registers in the MCU

## Description

View and edit the map of peripherals in the SoC model to the hardware peripherals.

Using the **Peripheral Configuration** tool, you can:

- View and edit the assignment of peripherals to MCU peripheral registers.
- Check the peripheral to register map of your model for any conflicts between peripherals.

**Peripheral Configuration for TI Delfino F28379D LaunchPad**

ADC	Simulink block:	RefModel/PWM Write	View block	<p><b>Additional Information</b> For more information on Peripheral Configuration tool, see <a href="#">Peripheral Configuration</a>.</p> <p><b>Description</b> Configure the <a href="#">PWM Write</a> blocks in the model to map to the PWM peripherals on your hardware board.</p>
PWM	<b>Parameters:</b>			
	PWM Module:	ePWM1		
	High speed clock divider:	1		
	Timer base clock divider:	1		
	Period (clock cycles):	64000		
	<input type="checkbox"/> Enable phase offset			
	Count mode:	Up-Down		
	Action on counter=zero:	Do nothing		
	Action on counter=period:	Do nothing		
	Action on counter=CMPA on up count:	Clear		
	Action on counter=CMPA on down count:	Set		
	<input checked="" type="checkbox"/> Enable shadow mode			
	Reload CMPA register:	Counter equals to zero (CTR=Zero)		
	ADC Start of conversion for ePWMxA module:	Disable		
	Dead band (cycles):	0		

OK Cancel Apply

## Open the Peripheral Configuration

- In the Configuration Parameters dialog box, select **Hardware Implementation** from the left pane. Under **Hardware board settings > Design mapping**, click **View/Edit Peripheral Map**.
- In the SoC Builder tool, in the **Review Memory and Interrupt Map** section, click **View/Edit Peripheral Map**.

## Parameters

### ADC

#### Simulink block — Select ADC Read block in model

reference-model-name / block-name

Select an ADC Read block from the model to apply the code generation parameter configurations.

Example: RefModel/ADC Read

#### View block — View the ADC Read block in model

button

Open the ADC Read block selected in the **Simulink block** parameter in the model.

#### Module — Hardware ADC Module

A (default) | B | C | D

Select the ADC module A through D on the hardware board.

#### Start of conversion — Start of conversion trigger

S0C0 (default) | S0C0 | ... | S0C15

Identify the start-of-conversion trigger by number.

#### Conversion channel — Input channel to apply ADC

Internal (default) | Undefined | Interrupt name

Select the input channel to which this ADC conversion applies.

#### S0Cx Acquisition window (cycles) — Length of ADC acquisition period

positive scalar integer

Define the length of the acquisition period in ADC clock cycles. The value of this parameter depends on the SYSCLK and the minimum ADC sample time.

#### S0Cx Trigger source — SoC trigger source

name of SoC event

Select the event source that triggers the start of the conversion.

#### ADCINT will trigger S0Cx — Use ADCINT interrupt to trigger start of conversion

No ADCINT (default) | ADCINT1 | ADCINT2

At the end of conversion, use the ADCINT1 or ADCINT2 interrupt to trigger a start of conversion. This loop creates a continuous sequence of conversions. The default selection, No ADCINT disables this

parameter. To set the interrupt, select the `Post interrupt` at EOC trigger option, and choose the appropriate interrupt.

**Enable interrupt at EOC – Enable post interrupts when the ADC triggers end of conversion pulses**

false (default) | true

Enable post interrupts when the ADC triggers EOC pulses. When you select this option, the dialog box displays the **Interrupt selection** and **Interrupt continuous mode** options.

**Interrupt selection – ADC interrupt selection**

ADCINT1 (default) | ADCINT2 | ADCINT3 | ADCINT4

Select which ADCINT# interrupt the ADC posts to after triggering an EOC pulse.

**Interrupt continuous mode – Generate new EOC signal overriding previous interrupt flag status**

false (default) | true

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt, whether the previous interrupt flag has been acknowledged or not.

**PWM**

**Simulink block – Select PWM Write block in model**

reference-model-name / block-name

Select an PWM Write block from the model to apply the code generation parameter configurations.

Example: RefModel/PWM Write

**View block – View the PWM Read block in model**

button

Open the PWM Write block selected in the **Simulink block** parameter in the model.

**PWM Module – Indicates which ePWM module to use**

ePWM1 (default) | ePWM2 | ... | ePWMx

Select the appropriate ePWM module, ePWMx, where x is a positive integer.

**High speed clock divider – High speed time base clock prescaler divider HSPCLKDIV**

1 (default) | 2 | 4 | 6 | 8 | 10 | 12 | 14

Set the high speed time base clock prescaler divider, HSPCLKDIV.

**Timerbase clock divider – Time base clock TBCLK prescaler divider corresponding to CLKDIV**

1 (default) | 2 | 4 | 8 | 16 | 32 | 64 | 128

Use the Time base clock, TBCLK, prescaler divider, CLKDIV, and the high speed time base clock, HSPCLKDIV, prescaler divider, HSPCLKDIV, to configure the Time-base clock speed, TBCLK, for the ePWM module. Calculate TBCLK using this equation:  $TBCLK = PWM\ clock / (HSPCLKDIV * CLKDIV)$ .

For example, the default values of both CLKDIV and HSPCLKDIV are 1, and the default frequency of PWM clock is 200 MHz, so:  $TBCLK$  in Hz =  $200 \text{ MHz}/(1 * 1) = 200 \text{ MHz}$   $TBCLK$  in seconds =  $1/TBCLK$  in Hz =  $1/200 \text{ MHz} = 0.005 \mu\text{s}$ .

### Period (clock cycles) – Period of ePWM counter

1 (default) | 2 | 4 | 8 | 16 | 32 | 64 | 128

Set the period of the ePWM counter waveform.

The timer period is in clock cycles:

Count Mode	Calculation	Example
Up or down	The value entered in clock cycles is used to calculate time-base period, $TBPRD$ , for the ePWM timer register. The period of the ePWM timer is $TCTR = (TBPRD + 1) * TBCLK$ , where $TCTR$ is the timer period in seconds, and $TBCLK$ is the time-base clock.	For ePWM clock, EPWMCLK, frequency = 200 MHz, and $TBCLK = 5 \text{ ns}$ . EPWMCLK will be equal to $SYSCLKOUT$ or $SYSCLKOUT/2$ depending on the ePWM clock divider, EPWMCLKDIV, parameter setting. When the timer period is entered in clock cycles $TBPRD = 9999$ , and the ePWM timer period is calculated as $TCTR = 50 \mu\text{s}$ . For the default action settings on the ePWMx tab, the ePWM period = $50 \mu\text{s}$ .
Up-down	The value entered in clock cycles is used to calculate the time-base period, $TBPRD$ , for the ePWM timer register. The period of the ePWM timer is $TCTR = 2 * TBPRD * TBCLK$ , where $TCTR$ is the timer period in seconds and $TBCLK$ is the time-base clock.	For EPWMCLK frequency = 200 MHz and $TBCLK = 5 \text{ ns}$ . When the timer period is entered in clock cycles, $TBPRD = 10000$ , and the ePWM timer period is calculated as $TCTR = 100 \mu\text{s}$ . For the default action settings on the ePWMx tab, the ePWM period = $100 \mu\text{s}$ .

The initial duty cycle of the waveform from the time the PWM peripheral starts operation until the ePWM input port receives a new value for the duty cycle is  $\text{Timer period} / 2$ .

### Enable phase offset – Enables the phase offset

false (default) | true

Enables to provide a timer phase offset value.

### Timer phase offset – Enables the phase offset

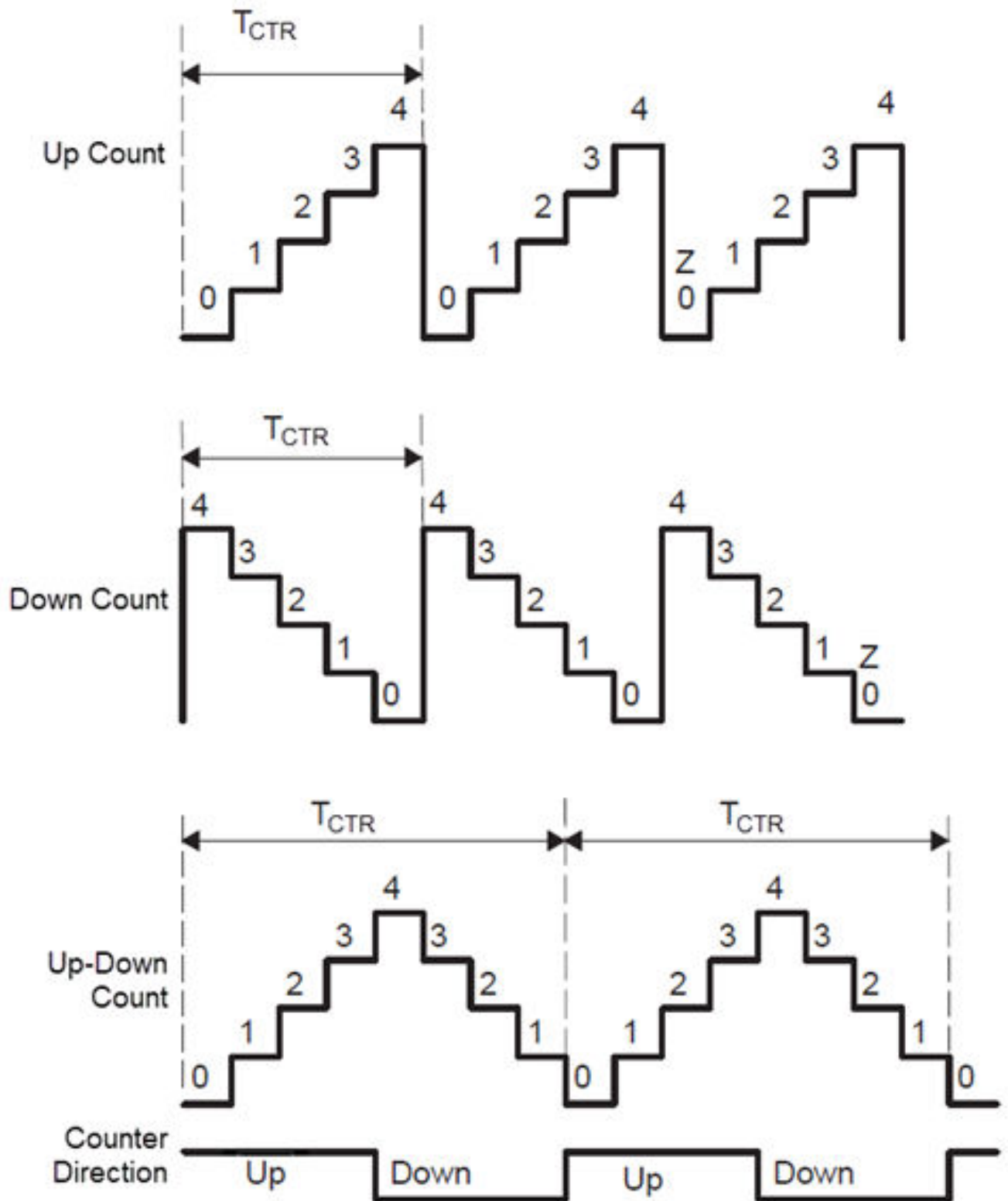
0 (default) | integer between 0 and 65535

The specified offset value is loaded in the time base counter on a synchronization event. Enter the phase offset value,  $TBPHS$ , in  $TBCLK$  cycles from 0 to 65535.

### Count mode – Indicates counting mode of ePWM counter

Up-Down (default) | Down | Up

Specify the counting mode of the PWM internal counter. This figure shows three counting waveforms.



**Action on counter=zero — Behavior of action qualifier (AQ) submodule at zero count**

Do nothing (default) | Clear | Set

This group determines the behavior of the action qualifier (AQ) submodule. The AQ module determines which events are converted into one of the various action types, producing the required switched waveforms of the ePWMA circuit. The ePWMB always generates a complement signal of ePWMA.

**Action on counter=period — Behavior of action qualifier (AQ) submodule at period count**

Do nothing (default) | Clear | Set

This group determines the behavior of the Action Qualifier (AQ) submodule. The AQ module determines which events are converted into one of the various action types, producing the required switched waveforms of the ePWMA circuit. The ePWMB always generates a complement signal of ePWMA.

**Action on counter=CMPA on up count — Behavior of Action Qualifier (AQ) submodule at CMPA on up count**

Clear (default) | Do nothing | Set

This group determines the behavior of the action qualifier (AQ) submodule. The AQ module determines which events are converted into one of the various action types, producing the required switched waveforms of the ePWMA circuit. The ePWMB always generates a complement signal of ePWMA.

**Action on counter=CMPA on down count — Behavior of action qualifier (AQ) submodule at CMPA on down count**

Set (default) | Clear | Do nothing

This group determines the behavior of the action qualifier (AQ) submodule. The AQ module determines which events are converted into one of the various action types, producing the required switched waveforms of the ePWMA circuit. The ePWMB always generates a complement signal of ePWMA.

**Enable shadow mode — Enable the shadow mode**

Disable (default) | Enable

When shadow mode is not enabled, the CMPA register refreshes immediately. Provide different reload mode for CMPA register.

**Reload CMPA register — Time at which the counter period is reset**

Counter equals to zero (CTR=Zero) (default) | Counter equals to period (CTR=PRD) | Counter equals to Zero or period (CTR=Zero or CTR=PRD) | Freeze

The time when the counter period resets based on the following condition:

- Counter equals to zero (CTR=Zero) - Refreshes the counter period when the value of the counter is 0.
- Counter equals to period (CTR=PRD) - Refreshes the counter period when the value of the counter is period.
- Counter equals to Zero or period (CTR=Zero or CTR=PRD) - Refreshes the counter period when the value of the counter is 0 or period.
- Freeze - Refreshes the counter period when the value of the counter is freeze.



**ADC Start of conversion for ePWM module – Trigger condition for an ADC start of the conversion event**

Counter equals to zero (CTR=Zero) (default) | Counter equals to period (CTR=PRD) | Counter equals to Zero or period (CTR=Zero or CTR=PRD) | Disable

This parameter specifies the counter match condition that triggers an ADC start of the conversion event. The choices are:

- Counter equals to zero (CTR=Zero) - Triggers an ADC start of the conversion event when the ePWM counter reaches 0.
- Counter equals to period (CTR=PRD) - Triggers an ADC start of the conversion event when the ePWM counter reaches the period value.
- Counter equals to Zero or period (CTR=Zero or CTR=PRD) - Triggers an ADC start of the conversion event when the time base counter, TBCTR, reaches zero or when the time base counter reaches the period, TBCTR = TBPRD.
- Disable - Disable ADC start of conversion event.

**Dead band (cycles) – Enables the phase offset**

0 (default) | integer between 0 and 65535

This parameter specifies the deadband delay for rising edge and falling edge in time-base clock cycles.

**See Also**

PWM Write | ADC Read

**Introduced in R2020b**

## SoC Builder

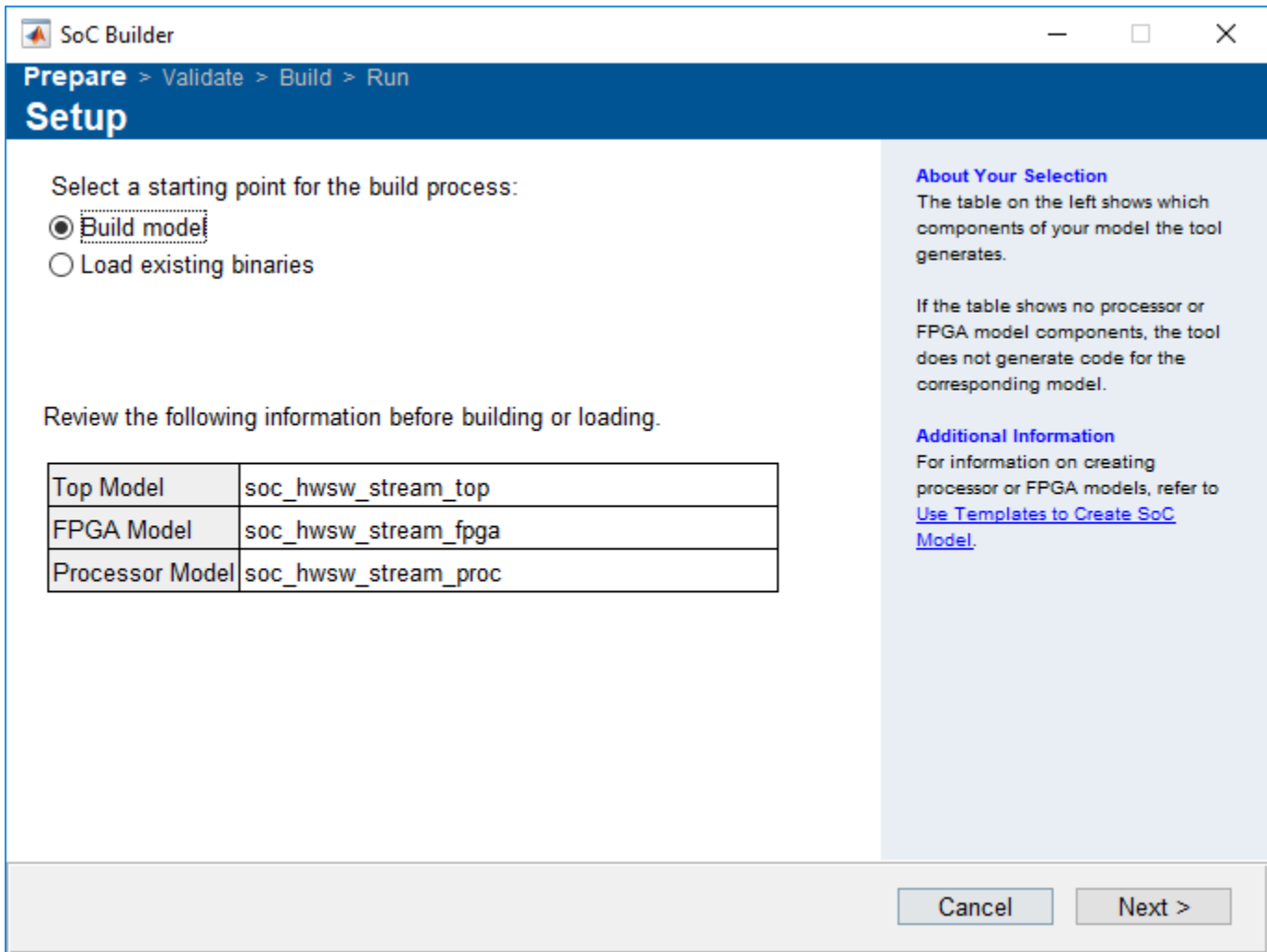
Build, load, and execute SoC model on SoC, FPGA, and MCU boards

### Description

The **SoC Builder** tool steps through the various stages for building and executing an SoC model on an SOC, FPGA, or MCU board.

Using this tool, you can:

- Review the model information provided to the tool.
- Review the memory map and edit it if needed.
- Configure the peripheral register settings.
- Map model tasks to interrupt service routines.
- 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, Intel Quartus, Texas Instruments™ Code Composer Studio™ 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 | Peripheral Map | Task Mapping**

## **Topics**

“Generate SoC Design”

**Introduced in R2019a**