

# HDL Verifier™ Support Package for Intel® FPGA Boards

User's Guide



# MATLAB® & SIMULINK®

R2023a



## 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

*HDL Verifier™ Support Package for Intel® FPGA Boards User's Guide*

© COPYRIGHT 2014–2023 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

October 2014	Online only	Revised for Version 14.2.0 (R2014b)
March 2015	Online only	Revised for Version 15.1.0 (R2015a)
September 2015	Online only	Revised for Version 15.2.0 (R2015b)
March 2016	Online only	Revised for Version 16.1.0 (R2016a)
September 2016	Online only	Revised for Version 16.2.0 (R2016b)
March 2017	Online only	Revised for Version 17.1.0 (R2017a)
September 2017	Online only	Revised for Version 17.2.0 (R2017b)
March 2018	Online only	Revised for Version 18.1.0 (R2018a)
September 2018	Online only	Revised for Version 18.2.0 (R2018b)
March 2019	Online only	Revised for Version 19.1.0 (R2019a)
September 2019	Online only	Revised for Version 19.2.0 (R2019b)
March 2020	Online only	Revised for Version 20.1.0 (R2020a)
September 2020	Online only	Revised for Version 20.2.0 (R2020b)
March 2021	Online only	Revised for Version 21.1.0 (R2021a)
September 2021	Online only	Revised for Version 21.2.0 (R2021b)
March 2022	Online only	Revised for Version 22.1.0 (R2022a)
September 2022	Online only	Revised for Version 22.2.0 (R2022b)
March 2023	Online only	Revised for Version 23.1.0 (R2023a)

<b>1</b>	<b>HDL Verifier Support for Intel FPGA Boards</b>	
	<b>Intel FPGA Board Support from HDL Verifier</b> .....	<b>1-2</b>
	Supported Intel FPGA Boards .....	<b>1-2</b>
	<b>Supported EDA Tools and Hardware</b> .....	<b>1-5</b>
	Software .....	<b>1-5</b>
	Board Connections .....	<b>1-5</b>
	<b>Download HDL Verifier FPGA Board Support Packages</b> .....	<b>1-7</b>
	HDL Verifier Support Package for Intel FPGA Boards .....	<b>1-7</b>
	Install Support Package with Connection to Internet .....	<b>1-7</b>
	Install Support Package Offline .....	<b>1-8</b>
	<b>Intel FPGA Board Customization</b> .....	<b>1-10</b>

	<b>Setup and Configuration</b>	
<b>2</b>	<b>Guided Hardware Setup</b> .....	<b>2-2</b>
	Select Board and Interface .....	<b>2-2</b>
	Setup Checklist .....	<b>2-2</b>
	Setup Steps .....	<b>2-3</b>
	Configure NIC on Host Computer .....	<b>2-5</b>
	Install PCI Express Driver .....	<b>2-6</b>
	Verify Setup .....	<b>2-6</b>
	Open Examples .....	<b>2-6</b>

	<b>AXI Manager</b>	
<b>3</b>	<b>Set Up AXI Manager</b> .....	<b>3-2</b>
	Integrate AXI Manager IP in FPGA Design .....	<b>3-2</b>
	JTAG Considerations .....	<b>3-3</b>
	<b>Use Simulink to Access FPGA Locations</b> .....	<b>3-5</b>
	<b>PCI Express AXI Manager</b> .....	<b>3-7</b>
	PCIe AXI Manager IP .....	<b>3-7</b>
	PCI Express Core .....	<b>3-8</b>

<b>Ethernet AXI Manager</b> .....	<b>3-12</b>
Ethernet MAC Hub IP .....	<b>3-12</b>
UDP AXI Manager IP .....	<b>3-14</b>

## **AXI Manager Reference**

### **4**

## **FPGA Data Capture**

### **5**

<b>Data Capture Workflow</b> .....	<b>5-2</b>
Generate and Integrate Data Capture IP Using HDL Workflow Advisor . . .	<b>5-3</b>
Configure and Generate IP Core for an Existing HDL Design .....	<b>5-3</b>
Integrate IP into FPGA .....	<b>5-4</b>
Capture Data .....	<b>5-4</b>
<b>Triggers</b> .....	<b>5-6</b>
What Is a Trigger Condition? .....	<b>5-6</b>
Sequential Trigger .....	<b>5-6</b>
Configure a Trigger Condition .....	<b>5-7</b>
Trigger Position .....	<b>5-8</b>
<b>Design Considerations for Data Capture</b> .....	<b>5-10</b>
Signals to Capture .....	<b>5-10</b>
Capture Timing .....	<b>5-10</b>
JTAG Considerations .....	<b>5-10</b>
<b>Capture Conditions</b> .....	<b>5-12</b>
What Is Capture Condition? .....	<b>5-12</b>
Configure Capture Condition .....	<b>5-12</b>
Differences Between Triggers and Capture Conditions .....	<b>5-13</b>

## **Data Capture Reference**

### **6**

## **HDL Verifier Support Package for Intel FPGA Boards Examples**

### **7**

<b>Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture</b> .....	<b>7-2</b>
--	------------

<b>Access FPGA External Memory Using AXI Manager .....</b>	<b>7-13</b>
<b>Stream Audio Signal from Intel FPGA Board Using Ready-to-Capture Signal .....</b>	<b>7-17</b>
<b>IP Core Generation Workflow with Ethernet-Based AXI Manager .....</b>	<b>7-23</b>
<b>Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow .....</b>	<b>7-28</b>



# HDL Verifier Support for Intel FPGA Boards

---

- “Intel FPGA Board Support from HDL Verifier” on page 1-2
- “Supported EDA Tools and Hardware” on page 1-5
- “Download HDL Verifier FPGA Board Support Packages” on page 1-7
- “Intel FPGA Board Customization” on page 1-10

## Intel FPGA Board Support from HDL Verifier

HDL Verifier automates the verification of HDL code on FPGA boards by providing connections between your FPGA board and your simulations in Simulink® or MATLAB®.

- FPGA-in-the-loop (FIL) enables you to run a Simulink or MATLAB simulation that is synchronized with an HDL design running on an FPGA board.
- FPGA data capture is a way to observe signals from your design while the design is running on the FPGA. It captures a window of signal data from the FPGA, based on your configuration and trigger settings, and returns the data to MATLAB or Simulink.
- AXI manager provides access to live on-board memory locations from Simulink or MATLAB. You must include the AXI manager IP in your FPGA design.

To use each of these features, you must have a supported FPGA board connected to your MATLAB host computer using a supported connection type, and a supported synthesis tool.

### Supported Intel FPGA Boards

This support package enables FIL simulation for the boards in the table. FPGA data capture and AXI manager are available on those boards that have JTAG USB Blaster I or USB Blaster II connections.

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPGA Data Capture	AXI Manager	FIL	FPGA Data Capture	AXI Manager	FIL <sup>a</sup>	FPGA Data Capture	AXI Manager	
Intel Arria® II	Arria II GX FPGA Development Kit	x		x	x	x	x			x	
Intel Arria V	Arria V SoC Development Kit			x	x	x	x				
	Arria V Starter Kit	x		x	x	x	x			x	
Intel Arria 10	Arria 10 SoC Development Kit	x		x	x	x	x				
	Arria 10 GX	x		x	x	x	x	x		x	Quartus® Prime 18.0 is not recommended for Arria 10 GX over PCI Express.
Intel Cyclone® IV	Cyclone IV GX FPGA Development Kit	x		x	x	x	x			x	



Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPGA Data Capture	AXI Manager	FIL	FPGA Data Capture	AXI Manager	FIL <sup>a</sup>	FPGA Data Capture	AXI Manager	
	DE2-115 Development and Education Board	x		x	x	x	x				The Altera® DE2-115 FPGA development board has two Ethernet ports. FIL uses only Ethernet 0 port. Make sure that you connect your host computer with the Ethernet 0 port on the board via an Ethernet cable.
	BeMicro SDK	x		x	x	x	x				
Intel Cyclone III	Cyclone III FPGA Starter Kit			x	x	x	x				Altera Cyclone III boards are supported with Quartus II 13.1  <b>Note</b> Support for Cyclone III device family will be removed in a future release.
	Cyclone III FPGA Development Kit	x		x	x	x	x				
	Altera Nios II Embedded Evaluation Kit, Cyclone III Edition	x		x	x	x	x				
Intel Cyclone V	Cyclone V GX FPGA Development Kit	x		x	x	x	x			x	
	Cyclone V SoC Development Kit			x	x	x	x				
	Cyclone V GT FPGA Development Kit	x		x	x	x	x	x		x	
	Terasic Atlas-SoC Kit / DE0-Nano SoC Kit			x	x	x	x				

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPGA Data Capture	AXI Manager	FIL	FPGA Data Capture	AXI Manager	FIL <sup>a</sup>	FPGA Data Capture	AXI Manager	
	Arrow <sup>®</sup> SoCKit Development Kit			x	x	x	x				
Intel Cyclone 10 LP	Altera Cyclone 10 LP Evaluation Kit			x	x	x	x				
Intel Cyclone 10 GX	Altera Cyclone 10 GX FPGA Development Kit			x	x	x	x			x	Must be used with Quartus Prime Pro
Intel MAX <sup>®</sup> 10	Arrow MAX 10 DECA	x		x	x	x	x				
Intel Stratix <sup>®</sup> IV	Stratix IV GX FPGA Development Kit	x		x	x	x	x			x	
Intel Stratix V	DSP Development Kit, Stratix V Edition	x		x	x	x	x	x		x	

<sup>a</sup> FIL over PCI Express<sup>®</sup> connection is supported only for 64-bit Windows<sup>®</sup> operating systems.

## See Also

### More About

- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”
- “Data Capture Workflow” on page 5-2
- “Set Up AXI Manager” on page 3-2

# Supported EDA Tools and Hardware

## Software

### Intel Quartus Prime

Use this support package with these recommended versions:

- Intel Quartus Prime Standard 21.1
- Intel Quartus Prime Pro 21.3 (supported for Intel Arria 10 and Cyclone 10 GX only)
- Intel Quartus II 13.1 (supported for Intel Cyclone III boards only)

For tool setup instructions, see “Set Up FPGA Design Software Tools”.

## Board Connections

### JTAG Connection

You can run FPGA-in-the-loop, FPGA data capture, or AXI manager over a JTAG cable to your board. However, each feature requires exclusive use of the JTAG cable, so you cannot run more than one feature at the same time. To allow other tools access to the JTAG cable, such as programming the FPGA, and Quartus SignalTap, you must discontinue the JTAG connection in MATLAB. To release the JTAG cable:

- FPGA-in-the-loop — Close the Simulink model, or call the `release` method of the System object™.
- FPGA data capture — Close the FPGA Data Capture app, release the System object, or close the Simulink model.
- AXI manager — Call the `release` method of the object.

However, the nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. For more information, see the "Simultaneous Use of FPGA Data Capture and AXI Manager" section of “JTAG Considerations” on page 5-10.

For Intel boards, the JTAG clock frequency is 12 or 24 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

Required Hardware	Required Software
<ul style="list-style-type: none"> <li>• USB Blaster I or USB Blaster II download cable</li> </ul>	<ul style="list-style-type: none"> <li>• USB Blaster I or II driver</li> <li>• For Windows operating systems: Quartus Prime executable directory must be on system path.</li> <li>• For Linux® operating systems: <ul style="list-style-type: none"> <li>• Versions below Quartus II 13.1 are not supported.</li> <li>• Quartus II 14.1 is not supported.</li> <li>• Only 64-bit Quartus is supported.</li> <li>• Quartus library directory must be on LD_LIBRARY_PATH before starting MATLAB.</li> <li>• Prepend the Linux distribution library path before the Quartus library on LD_LIBRARY_PATH. For example, /lib/x86_64-linux-gnu:\$QUARTUS_PATH.</li> </ul> </li> </ul>

### Ethernet Connection

You can run FPGA-in-the-loop over an Ethernet connection.

Required Hardware	Supported Interfaces	Required Software
<ul style="list-style-type: none"> <li>• Gigabit Ethernet card</li> <li>• Cross-over Ethernet cable</li> <li>• FPGA board with supported Ethernet connection</li> </ul>	<ul style="list-style-type: none"> <li>• Gigabit Ethernet — GMII</li> <li>• Gigabit Ethernet — RGMII</li> <li>• Gigabit Ethernet — SGMII</li> <li>• Ethernet — MII</li> <li>• Ethernet — RMII</li> </ul>	<p>There are no software requirements for an Ethernet connection, but ensure that the firewall on the host computer does not prevent UDP communication.</p>

### PCI Express

FPGA-in-the-loop over a PCI Express connection is supported only for 64-bit Windows operating systems.

Board	Required Software
<ul style="list-style-type: none"> <li>• Cyclone V GT FPGA Development Kit</li> <li>• DSP Development Kit, Stratix V Edition</li> <li>• Arria 10 GX</li> </ul>	<p>Altera Quartus II 15.0</p>

### See Also

#### More About

- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”
- “Data Capture Workflow” on page 5-2
- “Set Up AXI Manager” on page 3-2

# Download HDL Verifier FPGA Board Support Packages

## HDL Verifier Support Package for Intel FPGA Boards

The support package for Intel FPGA boards contains the board definition files for FPGA-in-the-loop (FIL) simulation, FPGA data capture, or AXI manager access, with HDL Verifier and supported Intel hardware. To use these features with supported Intel FPGA boards, first download the Intel FPGA board support package.

To install support packages:

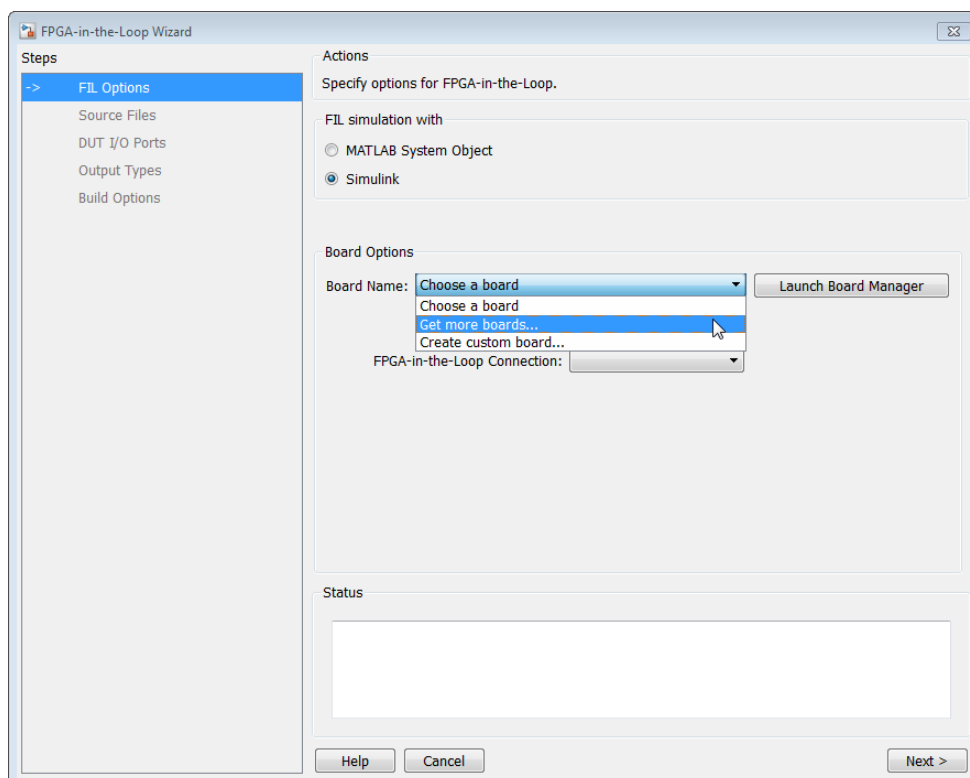
- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

You can also download FPGA board support packages from within the FPGA-in-the-Loop Wizard or the FPGA Board Manager.

## Install Support Package with Connection to Internet

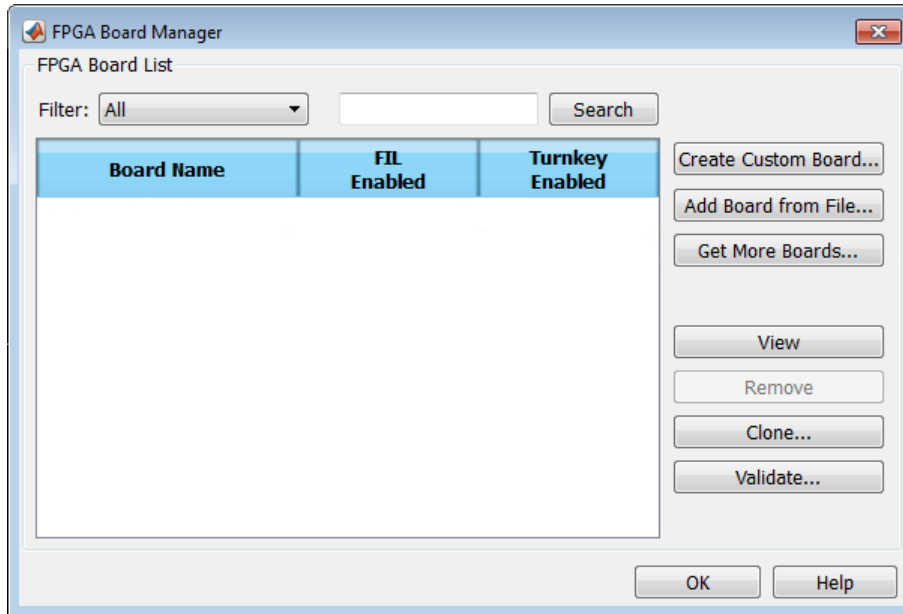
### From the FIL Wizard

- 1 In the MATLAB command window, enter the following command:  
`filWizard`
- 2 In the **FIL Options** pane, at **Board Name**, select **Get more boards...** from the drop-down menu.



### From the FPGA Board Manager

- 1 In the MATLAB command window, enter the following command:  
`fpgaBoardManager`
- 2 In the FPGA Board Manager dialog box, click **Get More Boards**.



### Install Support Package Offline

To install the support packages without an Internet connection, first download the packages on a computer that *does* have an Internet connection.

- 1 On the computer with the Internet connection, start MATLAB.
- 2 On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
- 3 Select your desired support package, and use the **Install** button pull-down menu to select **Download Only...**
- 4 Accept the license and select a folder for the download.
- 5 Copy the entire downloaded folder, for example, the R2016b folder, to a shared network drive or removable media, such as a USB drive.

Then, on the computer where you want to install the support packages:

- 1 Copy the downloaded folder to the host computer.
- 2 To start the installer, run the `install_supportsoftware.exe` executable file.
- 3 Follow the installer prompts to install the support package. If you do actually have an Internet connection, you are prompted to log in to your MathWorks® account.

## **See Also**

### **Related Examples**

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”
- “Intel FPGA Board Customization” on page 1-10

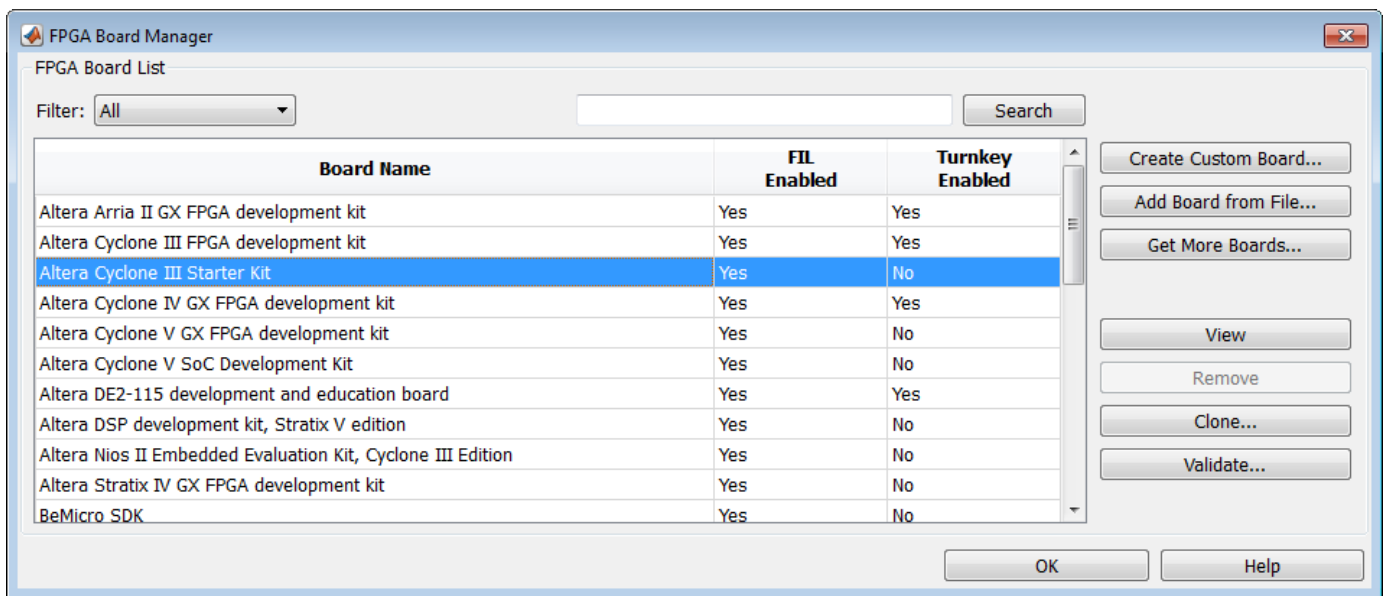
## Intel FPGA Board Customization

You might want to change part of the board definition file for an FPGA board to customize it. For example, you might want to add a new Ethernet interface using the Ethernet daughter card on the Altera Cyclone III Starter Kit. However, you cannot change the board definition file directly. Instead, make a copy of the file and then change the copied file.

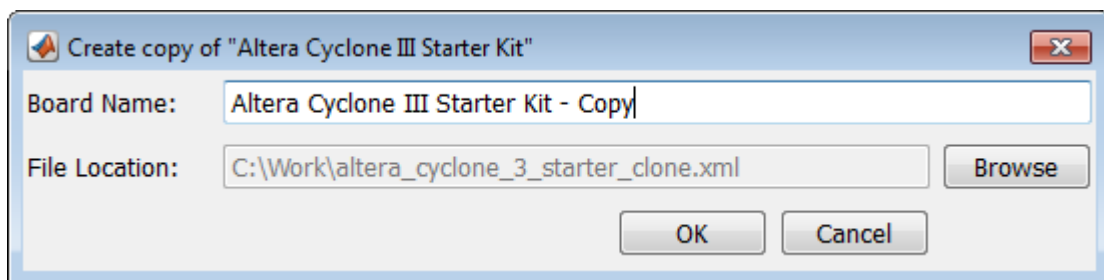
Create a copy of the board definition file using the FPGA Board Manager. Follow these steps:

- 1 Start the FPGA Board Manager by typing the following at the MATLAB command prompt:  

```
fpgaBoardManager
```
- 2 In the FPGA Board List, select the board you want to copy and click **Clone**.



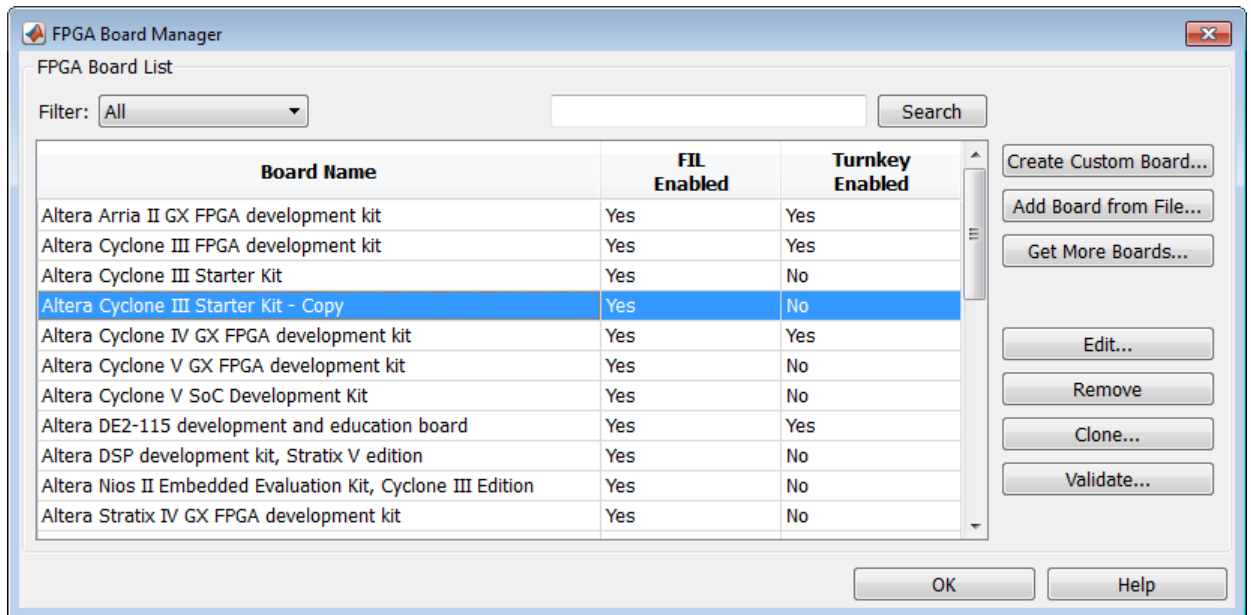
- 3 In the Create copy dialog box, specify the file name and location and click **OK**. Choose a new file name (and, optionally, a location) to preserve the original definition file.



- 4 Specify FPGA board information for the copy to customize it. By default, the copy inherits its information from the original, so you have to change only those fields that differ.
- 5 Click **OK**.

The copy of the original board now appears in the FPGA Board List. It also appears on the Board Name list in the FPGA-in-the-Loop Wizard and you can use the board with FIL simulation.





## See Also

### Related Examples

- “FPGA Board Customization”
- “FPGA Board Manager”



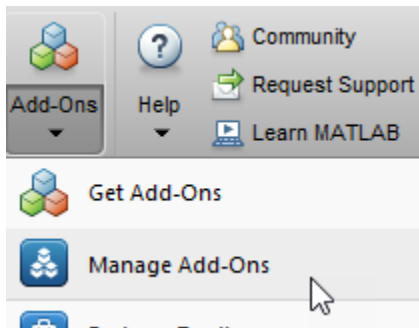
# Setup and Configuration


---

# Guided Hardware Setup

Before you can use the features in the HDL Verifier Support Package for Intel FPGA Boards, you must establish communication between the host computer and the hardware board. After the installer completes the support package installation, it guides you through the process of establishing communication with the hardware board.

If the support package is already installed, you can start the hardware setup by opening the Add-On Manager.



In the Add-On Manager, start the hardware setup process by clicking the **Gear** icon .

The setup process includes these steps:

- Specify a hardware board and interface.
- Configure the network interface card in the host computer (for the Ethernet interface only).
- Install the PCIe driver on the host computer (for the PCI Express interface only).
- Verify the connection between the host computer and the hardware board.

## Select Board and Interface

Choose a hardware board and an interface to use with this board from the list. For the full list of supported boards and interfaces, see “Supported Intel FPGA Boards” on page 1-2. HDL Verifier supports a PCI Express connection for FPGA-in-the-loop (FIL) with Windows operating systems only.

## Setup Checklist

The guided setup wizard displays a checklist of the hardware requirements. Confirm that you have the hardware required to complete the setup process.

---

**Note** Do not connect to the board or turn it on until you are prompted at a later step.

---

### Ethernet Requirements

- FPGA or SoC development board
- USB-JTAG cable (for FPGA boards only)

- Installed Quartus software (for FPGA boards only)
- Dedicated Gigabit network interface card (NIC) or USB 3.0 Gigabit Ethernet adapter dongle
- Ethernet cable
- Power supply adapter (if the board requires one)

### **JTAG Requirements**

- FPGA or SoC development board
- USB-JTAG cable
- Installed Quartus software
- Power supply adapter (if the board requires one)

### **PCI Express Requirements**

- FPGA development board
- USB-JTAG cable
- Installed Quartus software
- PCI Express slot and available space on the motherboard
- Power supply adapter (if the board requires one)

## **Setup Steps**

The guided setup wizard displays the setup steps for the selected interface. Follow these steps to set up your hardware board with the selected interface.

### **Ethernet**

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Connect the AC power cord to the power plug and plug the power supply adapter cable into the hardware board.
- 3** Use the crossover Ethernet cable to connect the Ethernet connector on the hardware board directly to the Ethernet adapter on your host computer.
- 4** Use the USB-JTAG download cable to connect the hardware board to the host computer.
- 5** Make sure that all the jumpers on the hardware board are in the factory default position.
- 6** Turn the power switch of the hardware board on.

### **JTAG**

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Connect the AC power cord to the power plug and plug the power supply adapter cable into the hardware board.
- 3** Use the USB-JTAG download cable to connect the hardware board to the host computer.
- 4** Make sure that all the jumpers on the hardware board are in the factory default position.
- 5** Turn the power switch of the hardware board on.

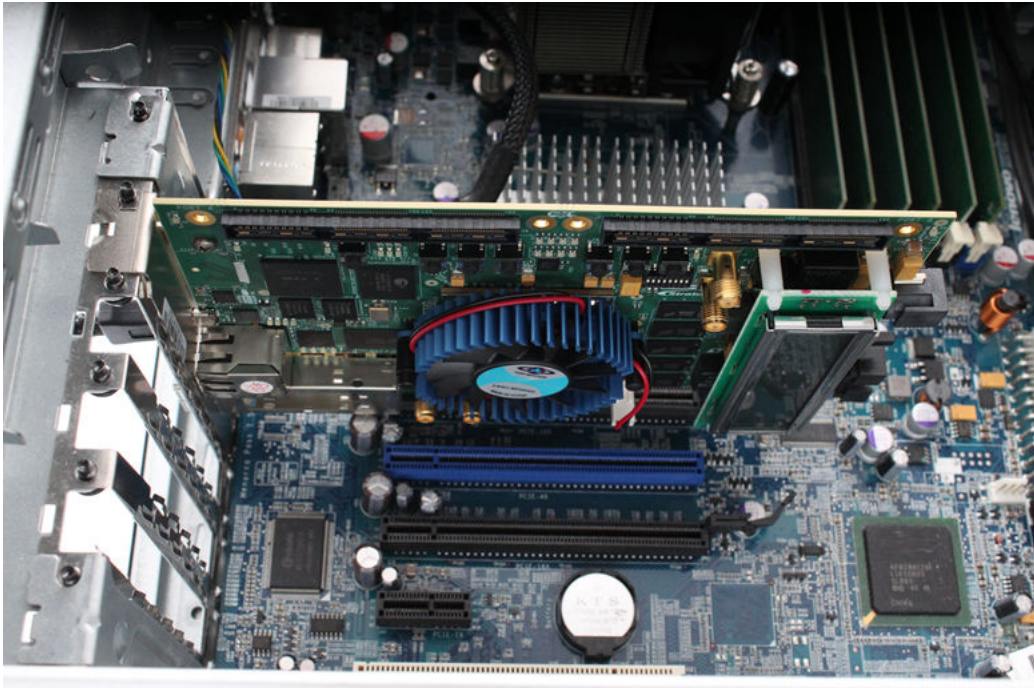
**PCI Express**

- 1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2 Select the maximum number of PCI Express lanes that the board supports. For details, refer to the user manual for the board.

Supported Board	PCI Express Setup	Documentation
DSP Development Kit, Stratix V Edition	Set the three switches (PCIE_PRSNT2nx1, x4, x8) in dip switch SW6 to ON. This setting selects 8-lane PCIe (default board setting).	<a href="https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/stratix/v-gs.html">https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/stratix/v-gs.html</a>
Cyclone V GT FPGA Development Kit	Set the two switches(PCIE_x1, x4) in dip switch SW3 to ON. This setting selects 4-lane PCIe (default board setting).	<a href="https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/cyclone/v-gt.html">https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/cyclone/v-gt.html</a>

- 3 Turn the host computer off.
- 4 Install the hardware board in a PCI Express slot inside the host computer.

This figure shows the Stratix V board installed in a host computer. This installation applies to all supported Intel VC boards.



- 5 Connect the JTAG cable to the hardware board and the host computer. The JTAG cable is required to program the FPGA.



- 6 Turn the power switch of the hardware board on.
- 7 Start up the host computer.

## Configure NIC on Host Computer

This step is required only when you select the Ethernet interface.

In this step, you configure the host computer so that it can communicate with the hardware board. You must have a dedicated Gigabit Ethernet NIC for the hardware board, with an Ethernet cable connecting the card to your hardware board. If you also want simultaneous internet access and you do not have a wireless connection, your host computer requires a second Ethernet NIC.

In the guided setup, select the NIC that you want to connect with the hardware board. If you have already configured the NIC, select **Skip this step if your network card is already configured for communicating with the FPGA or SoC board.**

The list displays the connected NICs detected on your host computer. The menu options show each NIC as (In Use) or (Available). The installer marks an NIC as (In Use) when the NIC is connected to a device and has an assigned IP address.

If you do not see your NIC listed, click **Refresh** to trigger the NIC detection and refresh the list. Refreshing the list is useful when, for example, you plug in a USB Ethernet adapter dongle while viewing this pane.

- If all the NICs listed are in use, free up a NIC for use with the hardware and then click **Refresh**.
- If the NIC list is empty, VMWare software, if present, can interfere with NIC detection. To get an accurate list of NICs on your host computer, remove the VMWare software.
- Check whether the missing NIC is disabled in the control panel. If the NIC is disabled, enable it.

Leave the IP address for the NIC as the default. Alternatively, specify the IP address in dotted quad format, for example, 192.168.0.1.

When you click **Next**, the software configures the NIC.

### Install PCI Express Driver

This step is required only when you select the PCI Express interface.

If you have already installed the PCI Express drivers, skip this step.

Install the PCI Express drivers before you use FIL, FPGA data capture, or AXI manager with a PCI Express connection. This step performs the driver installation for you. The process can take 10 or more minutes to install, and might require system administrator privileges.

You can install the drivers now, or you can choose to perform the setup again later. To run the support package setup, on the MATLAB **Home** tab, in the **Environment** section, select **Help > Check for Updates**.

### Verify Setup

You can verify the hardware setup for Ethernet and JTAG interfaces. This step runs the tests to verify the connection between the host computer and the hardware board based on the selected interface. Before you run the test, make sure that:

- 1 You have installed the appropriate vendor tool and the tool is on the MATLAB path. See “Set Up FPGA Design Software Tools”.
- 2 The board is turned on.

This step runs these tests to verify the connection for the selected interface.

#### Ethernet

- 1 Generate an FPGA programming file for your hardware board.
- 2 Program the FPGA.
- 3 Detect an Ethernet connection.

#### JTAG

- 1 Generate an FPGA programming file for your board.
- 2 Program the FPGA.
- 3 Perform the data transaction between the FPGA and the host computer.

If the connection is not successful, the most common reasons are that the board is not connected properly or it is not turned on. Check the cable connections and power switch and try again.

### Open Examples

When the installer completes your hardware setup, you can exit the installer or open the examples to get started.



**See Also**

**Hardware Setup** | “FPGA-in-the-Loop Simulation” | “Data Capture Workflow” on page 5-2 | “Set Up AXI Manager” on page 3-2

**See Also****Related Examples**

- Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop
- “Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2
- “Access FPGA External Memory Using AXI Manager” on page 7-13
- “IP Core Generation Workflow with Ethernet-Based AXI Manager” on page 7-23

# Hardware Setup

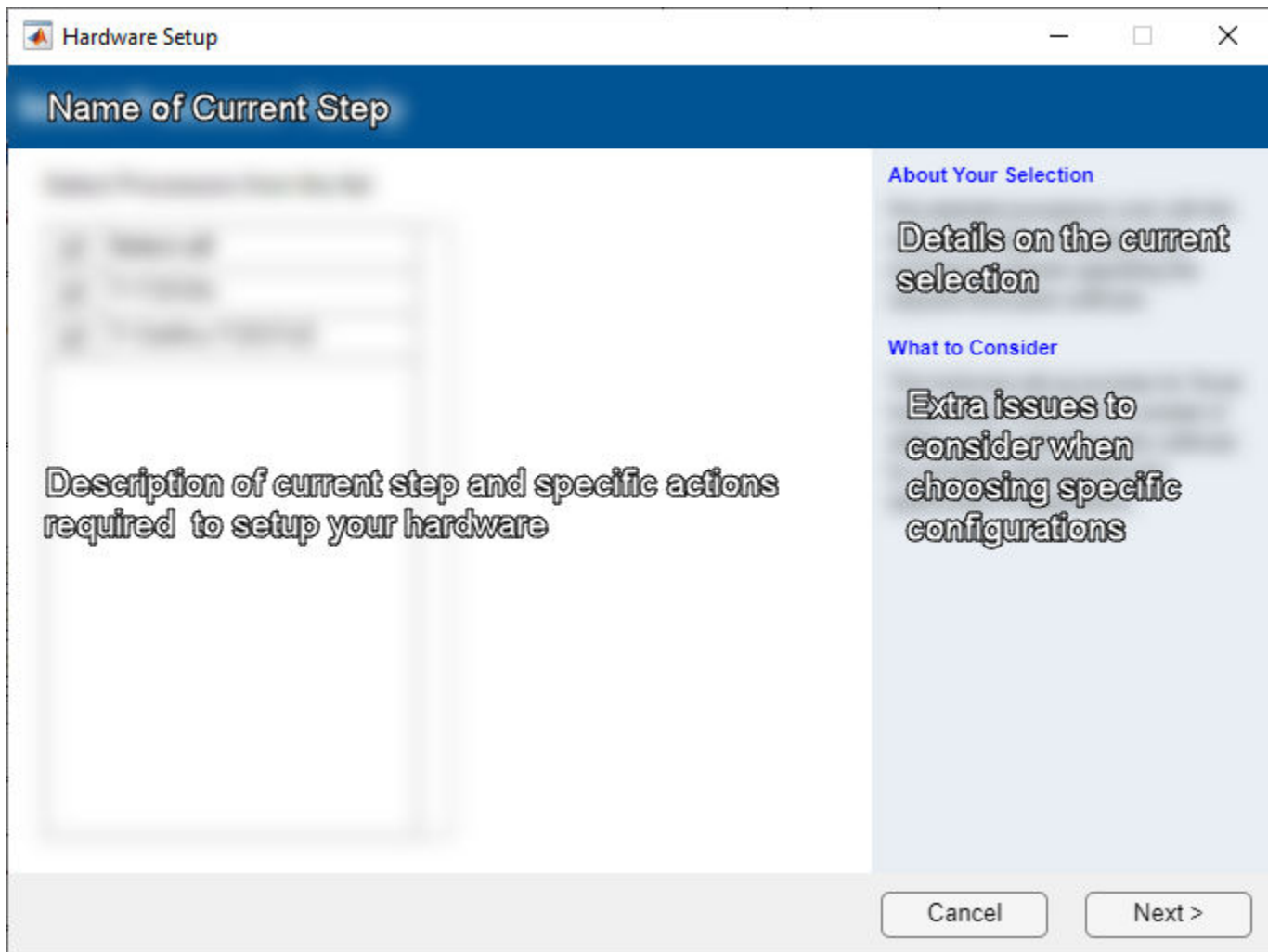
Set up and connect your hardware boards

## Description


Hardware boards supported by MathWorks require additional setup steps to connect to MATLAB and Simulink software. The **Hardware Setup** tool guides you through the hardware setup process. Use this tool to configure a target hardware board for use with FPGA-in-the-loop (FIL), FPGA data capture, and AXI manager over the JTAG, Ethernet, and PCI Express interfaces. For the full list of supported boards and interfaces, see “Supported Intel FPGA Boards” on page 1-2.

The hardware setup process for the HDL Verifier Support Package for Intel FPGA Boards includes these steps:

- Specify a hardware board and interface.
- Configure the network interface card in the host computer (for the Ethernet interface only).
- Install the PCIe driver on the host computer (for the PCI Express interface only).
- Verify the connection between the host computer and the hardware board.



## Open the Hardware Setup

- In the install window, at the end of the installation process, click the **Setup Now** button.
- After installing the HDL Verifier Support Package for Intel FPGA Boards, use **Get and Manage Add-Ons**. When the installation is complete, in the Add-On Manager, click the **Gear** icon .

## Version History

Introduced in R2016a

### See Also

“Guided Hardware Setup” on page 2-2



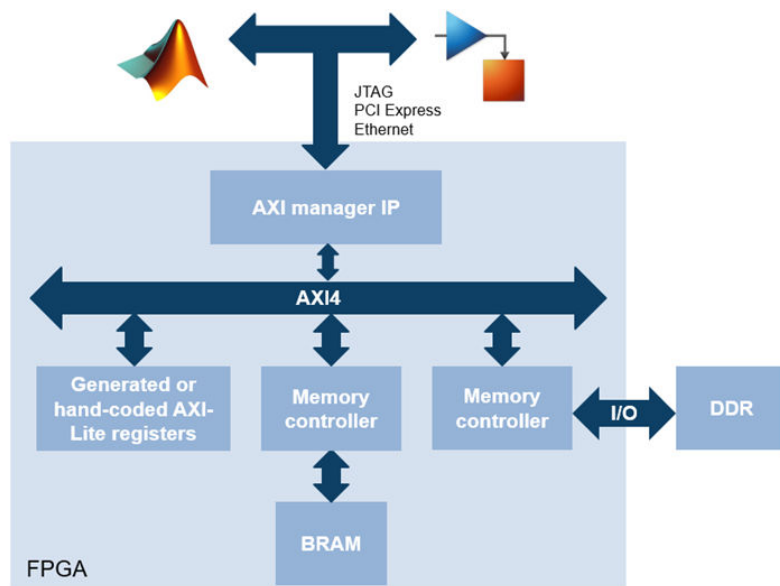
# AXI Manager

---

## Set Up AXI Manager

**Note** MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To access on-board memory locations from MATLAB or Simulink, you must include the AXI manager IP in your FPGA design. This IP connects to subordinate memory locations on the board. The IP also responds to read and write commands from MATLAB or Simulink, over JTAG, PCI Express, or Ethernet cable.



### Integrate AXI Manager IP in FPGA Design

To set up the AXI manager IP for access from MATLAB or Simulink, follow these setup steps:

- 1 Add the path for the AXI manager IP files to your project using the `setupAXIManagerForQuartus` function.
- 2 Open Quartus, and from the IP Catalog select the AXI manager IP in your FPGA design.
  - When using JTAG as a physical connection, select AXI Manager.
  - When using Ethernet as a physical connection, select UDP AXI Manager and Ethernet MAC Hub and add them to your project.
  - When using PCIe as a physical connection, select PCIe AXI Manager and add it to your project.
- 3 In your FPGA project, specify which addresses the AXI manager IP is allowed to access.

**Note** The AXI manager IP supports AXI4 Lite, AXI4, and Altera Avalon slave memory locations. The FPGA interconnect automatically converts AXI4 transactions to the protocol of each address.

- 4 Compile your FPGA project, including the AXI manager IP.
- 5 Connect your FPGA board to your host computer using a physical cable (JTAG, PCI Express, or Ethernet cable).
- 6 Program the FPGA with your compiled design.

---

**Note** Alternatively, you can perform these steps in the HDL Coder™ guided workflow by using a sample reference design, such as the one included in this example: “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” on page 7-28.

---

After loading the design on your FPGA, you can access memory-mapped locations on the board.

To access the board from MATLAB, create an `aximanager` object and use the `readmemory` and `writememory` methods to read and write memory-mapped locations on the board.

To access the board from Simulink, create a Simulink model and include AXI Manager Write and AXI Manager Read in it. Configure the blocks to read and write memory-mapped locations on the board. For more information, see “Use Simulink to Access FPGA Locations” on page 3-5.

## JTAG Considerations

When using JTAG as a physical connection to your board, you might have additional IPs that use the same JTAG connection. Such IPs include Intel SignalTap II or Xilinx® Vivado® Logic Analyzer cores. However, only one of these applications can use the JTAG cable at a time. You must release the `aximanager` object to return the JTAG resource for use by other applications.

However, the nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. In this capture mode, you do not need to close or release the JTAG resource to switch between FPGA data capture and AXI manager. For more information, see “Simultaneous Use of FPGA Data Capture and AXI Manager” on page 5-11.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. You must stop any FPGA data capture or AXI manager JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Intel boards, the JTAG clock frequency is 12 MHz or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 MHz or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

## See Also

`aximanager` | AXI Manager Read | AXI Manager Write

## Related Examples

- “Access FPGA External Memory Using AXI Manager” on page 7-13
- “Ethernet AXI Manager” on page 3-12
- “PCI Express AXI Manager” on page 3-7

- “Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow” on page 7-28



## Use Simulink to Access FPGA Locations

**Note** MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To read and write memory-mapped locations on your FPGA board using Simulink, you must first integrate an AXI manager IP into your FPGA design. For more information, see "Integrate AXI Manager IP in FPGA Design" on page 3-2.

After integrating an AXI manager IP into your FPGA design, load the design on the FPGA. Then, create a Simulink model that includes source, sink, AXI Manager Write, and AXI Manager Read blocks, as in this figure.

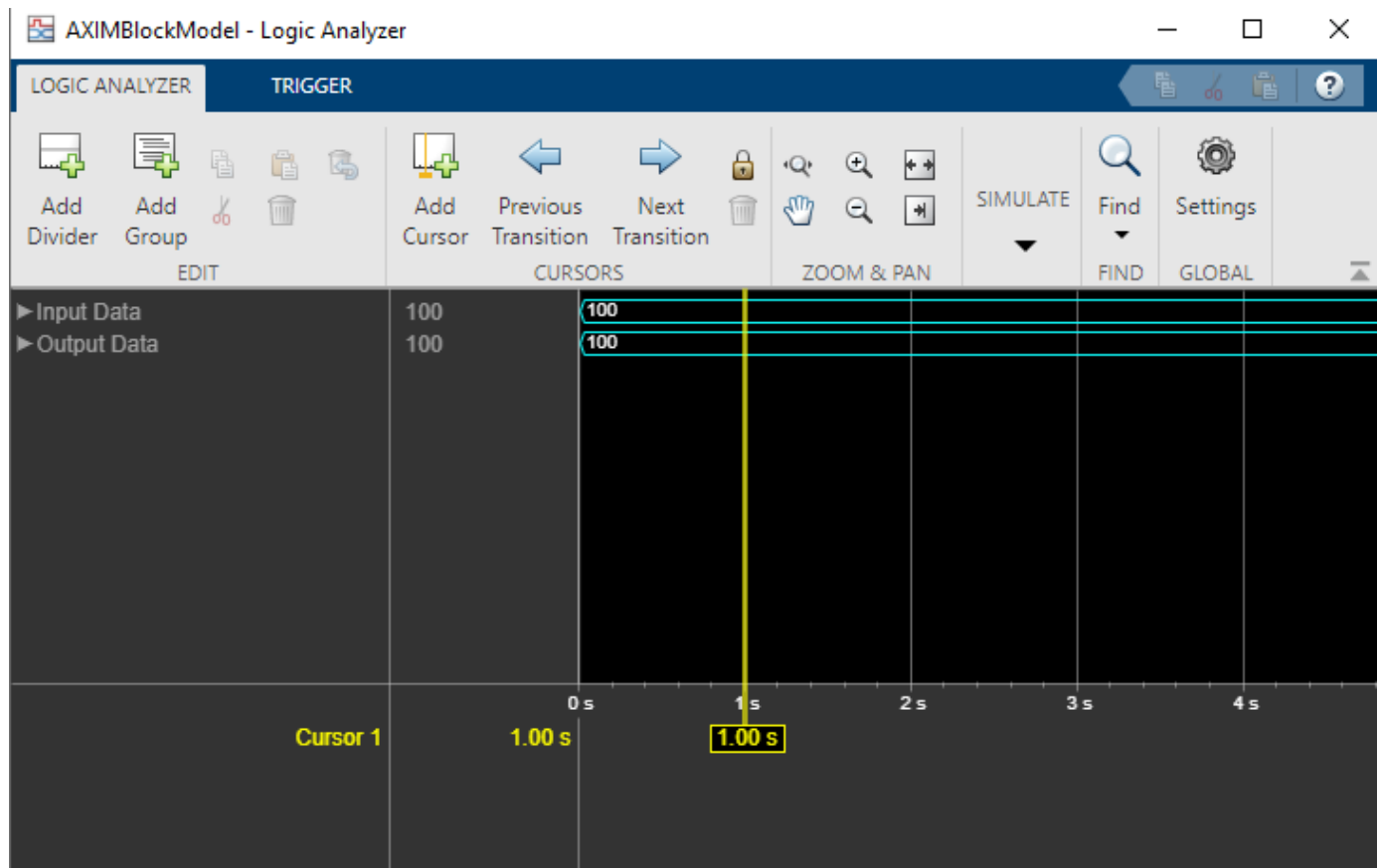


Configure the AXI Manager Write block. Set the write **Address** and **Burst type** parameters. On the **Interface** tab, select the type of interface used for communication with the FPGA board by using the **Type** parameter. Then, click **Configure global parameters** to configure the global interface parameters for that AXI manager interface.

Next, configure the AXI Manager Read block. Set the read **Address**, **Burst type**, **Output data type**, and **Output vector size** parameters. On the **Interface** tab, select the type of interface used for communication with the FPGA board by using the **Type** parameter.

Run the simulation. For each Simulink step, the write block writes to the FPGA, and the read block reads from the FPGA. View the results by using the **Logic Analyzer** app, or directing the data to a file.

This figure shows the input and output data displayed in the **Logic Analyzer** app. In this example, the AXI Manager Write block writes 100 to address 0, and the AXI Manager Read block reads from the same address.



### See Also

AXI Manager Read | AXI Manager Write

### More About

- "Set Up AXI Manager" on page 3-2

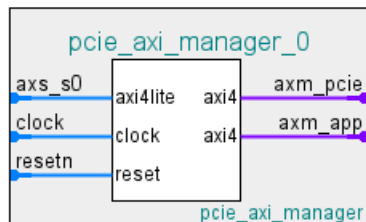
## PCI Express AXI Manager

**Note** PCI Express AXI master has been renamed to PCI Express AXI manager and the PCIe MATLAB as AXI Master IP has been renamed to the PCIe AXI Manager IP. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

When using PCI Express AXI manager, you must first include the following two intellectual property (IP) blocks in your Quartus Qsys project.

### PCIe AXI Manager IP

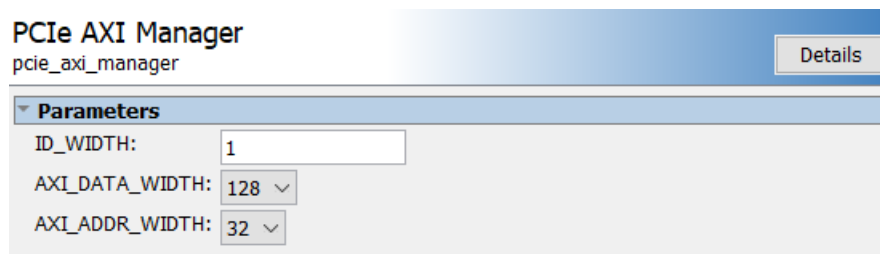
*PCIe AXI Manager* is an HDL IP provided by MathWorks. This IP connects the PCI Express (PCIe) core to your application code. It also has a configuration port for accessing configuration registers. This block diagram shows the interface to the HDL IP.



The interface includes the following parts:

- `clock` and `resetn` are the clock and reset inputs. Connect them to the AXI clock and reset.
- `axs_s0` is a 32-bit subordinate interface and is used for accessing the PCIe configuration registers. Connect this interface to the Avalon Memory Mapped master interface.
- `axm_pcie` is a 128-bit AXI manager interface. Connect this interface to the TX subordinate port on the PCIe core.
- `axm_app` is a 128-bit AXI manager interface. Connect this interface to your application logic.

After instantiating this IP in your design, open the block parameters for configuration.



Configure these parameters:

- **ID\_WIDTH** - This parameter is the ID width in bits. Its value must match the ID width of the AXI subordinate.
- **AXI\_DATA\_WIDTH** - This parameter is the data bus width. The IP supports 128-bit or 256-bit data. This parameter is not identical to the data width of the `axi_manager` object or the AXI

Manager Read or AXI Manager Write blocks. If the data width is set to 32 bits, and the **AXI\_DATA\_WIDTH** of your IP is set to 128 bits, HDL Verifier packs four 32-bit words to transfer on the 128-bit bus.

- **AXI\_ADDR\_WIDTH** - This parameter is the address bus width. The IP supports 32-bit address.

## PCI Express Core

The *Hard IP for PCI Express Core* is a board-specific IP provided by Intel. Use this IP for configuring and integrating the PCI Express port.

After instantiating the PCIe core HDL IP in your Quartus Qsys project, configure the PCIe core using these steps (this example is for an Arria 10 board).

- 1 On the **Physical Function 0 IDs** tab, set the parameters as shown in this figure.

**System:** system\_soc **Path:** pcie\_a10\_hip\_0

### Arria 10 Hard IP for PCI Express

altera\_pcie\_a10\_hip

Details  
Generate Example Design...

IP Settings Example Designs

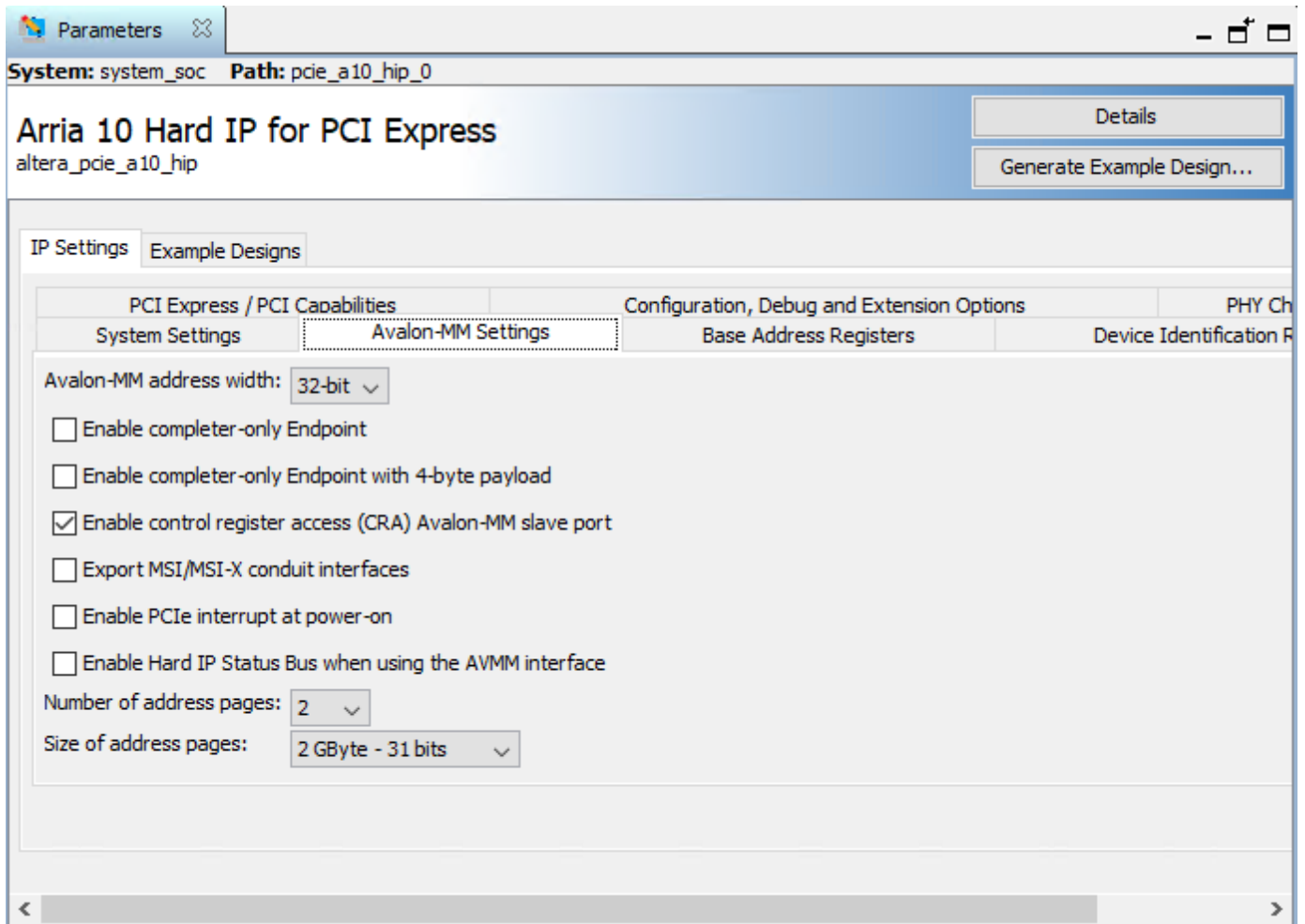
PCI Express / PCI Capabilities Configuration, Debug and Extension Options PHY Ch

System Settings Avalon-MM Settings Base Address Registers Device Identification F

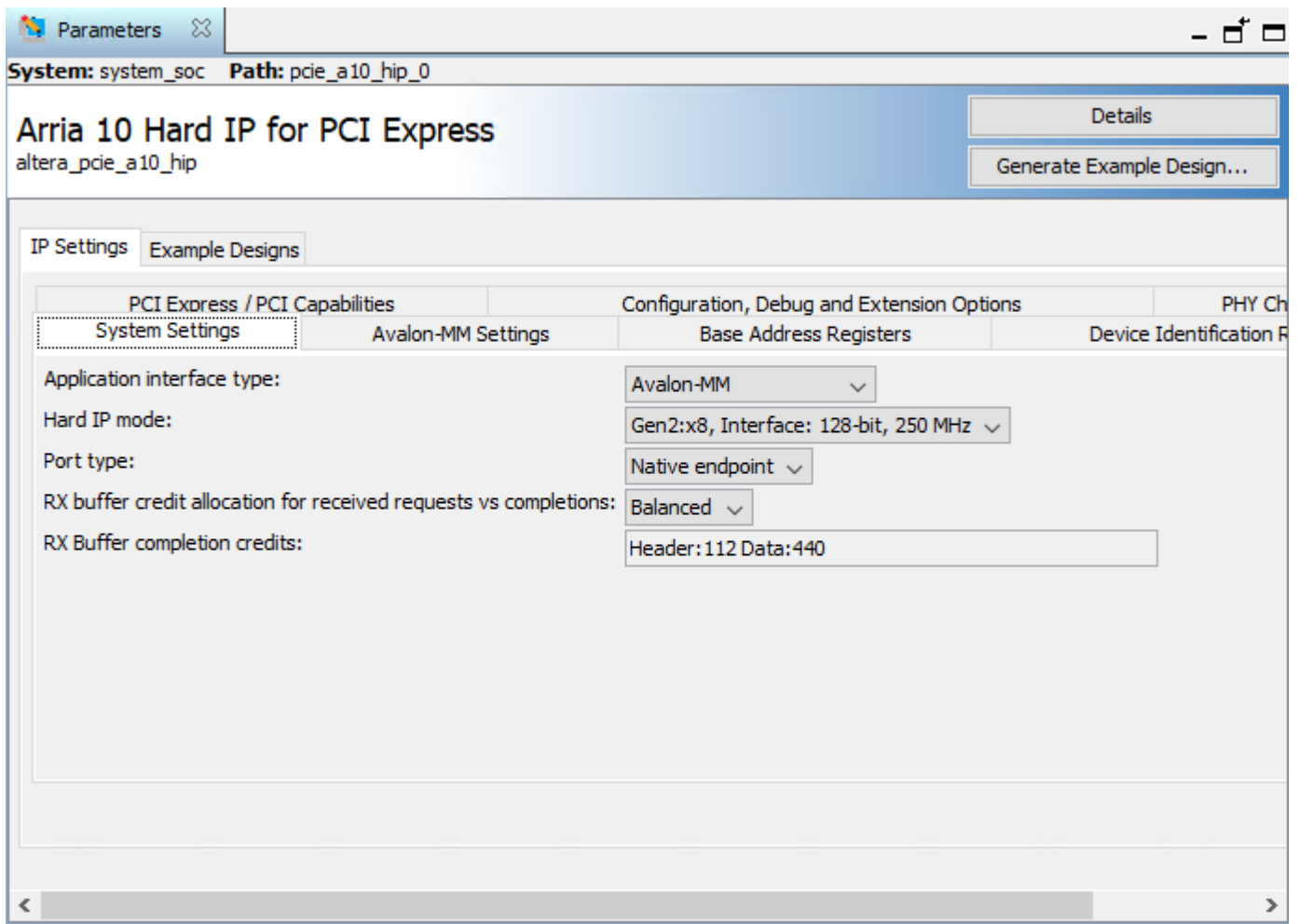
Physical Function 0 IDs

Vendor ID:	0x00001172
Device ID:	0x0000f001
Revision ID:	0x00000001
Class code:	0x00ff0000
Subsystem Vendor ID:	0x0000015b
Subsystem Device ID:	0x00000001

- 2 On the **Avalon-MM Settings** tab, set the parameters as shown in this figure.



- 3 On the **System Settings** tab, select a **Hard IP mode** with a bus width to match the **AXI\_DATA\_WIDTH** parameter previously set in the PCIe AXI Manager IP block parameters. (The IP currently supports 128-bit mode).



- 4 Connect the PCIe AXI Manager IP to the PCIe core (this example shows Arria 10 Hard IP for PCI Express).

Use	Connections	Name	Description	Export	Clock	Base	
<input checked="" type="checkbox"/>		<b>emif_0</b>	Arria 10 External Memory Interfaces				
		emif_usr_reset_reset...	Reset Output	<i>Double-click to export</i>			
		emif_usr_clk_clock_so...	Clock Output	<i>Double-click to export</i>	emif_0_emif...		
		ctrl_ecc_user_interru...	Conduit	<i>Double-click to export</i>			
		ctrl_amm_avalon_slav...	Avalon Memory Mapped Slave	<i>Double-click to export</i>	emif_0_emif...	# 0x0000_0000	
		global_reset_reset_sink	Reset Input	<i>Double-click to export</i>			
		pll_ref_clk_clock_sink	Clock Input		<b>ddr_ref_clk</b>	<b>exported</b>	
		oct_conduit_end	Conduit		<b>ddr_oct</b>		
		mem_conduit_end	Conduit		<b>ddr_mem</b>		
		status_conduit_end	Conduit	<i>Double-click to export</i>			
<input checked="" type="checkbox"/>			<b>pcie_axi_manager_0</b>	PCIe AXI Manager			
		axs_s0	AXI4Lite Slave	<i>Double-click to export</i>	[clock]	# 0x0000_4000	
		clock	Clock Input	<i>Double-click to export</i>	<b>pcie_a10_...</b>		
		reseth	Reset Input	<i>Double-click to export</i>	[clock]		
		axm_pcie	AXI4 Master	<i>Double-click to export</i>	[clock]		
	axm_app	AXI4 Master	<i>Double-click to export</i>	[clock]			
<input checked="" type="checkbox"/>		<b>pcie_a10_hip_0</b>	Intel Arria 10/Cyclone 10 Hard IP for P...				
	coredkout_hip	Clock Output	<i>Double-click to export</i>	pcie_a10_hi...			
	refclk	Clock Input		<b>refclk</b>	<b>exported</b>		
	npwr	Conduit		<b>pcie_reset_in</b>			
	app_nreset_status	Reset Output	<i>Double-click to export</i>	pcie_a10_hi...			
	hip_ctrl	Conduit	<i>Double-click to export</i>				
	hip_pipe	Conduit	<i>Double-click to export</i>				
	hip_serial	Conduit	<i>Double-click to export</i>	<b>hip_serial</b>			
	txs	Avalon Memory Mapped Slave	<i>Double-click to export</i>	pcie_a10_hi...	# 0x0000_0000		
	cra	Avalon Memory Mapped Slave	<i>Double-click to export</i>	pcie_a10_hi...	# 0x0000_0000		
	cra_irq	Interrupt Sender	<i>Double-click to export</i>				
	rxm_bar0	Avalon Memory Mapped Master	<i>Double-click to export</i>	pcie_a10_hi...			
	rxm_irq	Interrupt Receiver	<i>Double-click to export</i>		IRQ 0		
<input checked="" type="checkbox"/>		<b>iopll_0</b>	IOPLL Intel FPGA IP				
	reset	Reset Input	<i>Double-click to export</i>				
	refclk	Clock Input	<i>Double-click to export</i>	<b>pcie_a10_...</b>			
	outclk0	Clock Output	<i>Double-click to export</i>	pcie_a10_...			
<input checked="" type="checkbox"/>		<b>DUT_ip_0</b>	DUT_ip				
	ip_clk	Clock Input	<i>Double-click to export</i>	<b>iopll_0_out...</b>			
	ip_rst	Reset Input	<i>Double-click to export</i>	[ip_clk]			
	axi_clk	Clock Input	<i>Double-click to export</i>	<b>iopll_0_out...</b>			
	axi_reset	Reset Input	<i>Double-click to export</i>	[axi_clk]			
	s_axi	AXI4 Slave	<i>Double-click to export</i>	[axi_clk]	# 0x8000_0000		
	AXI4_Master	AXI4 Master	<i>Double-click to export</i>	[ip_clk]			

- 5 Compile and build your FPGA project.
- 6 Insert the FPGA board into the PCI Express slot on the motherboard of the host computer.
- 7 Program the FPGA with your compiled design.
- 8 Restart the host machine.

Once the program is running on your FPGA board, you can create an AXI manager object. For more information, see `aximanager`. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` functions of this object.

## See Also

`aximanager`

## More About

- “Set Up AXI Manager” on page 3-2

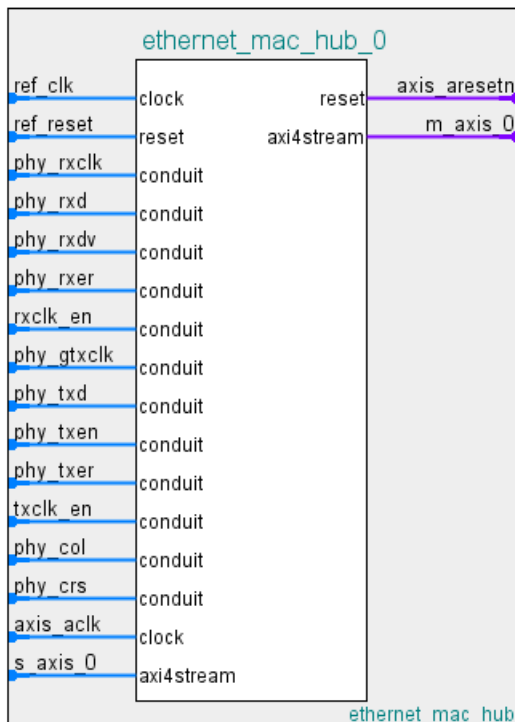
## Ethernet AXI Manager

**Note** Ethernet AXI master has been renamed to Ethernet AXI manager and the UDP MATLAB as AXI Master IP has been renamed to the UDP AXI Manager IP. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

Integrate and configure AXI manager over Ethernet using user datagram protocol (UDP). To use Ethernet AXI manager, you must first include these two intellectual property (IP) blocks in your project: Ethernet media access controller (MAC) Hub and UDP AXI Manager.

### Ethernet MAC Hub IP

The Ethernet MAC Hub HDL IP supports the gigabit media independent interface (GMII). This Ethernet MAC Hub IP connects the Ethernet physical layer (PHY) to the UDP AXI Manager IP.



### Interface of Ethernet MAC Hub IP

The interface of the Ethernet MAC Hub IP includes the ports in this table.

Port	Description
<b>s_axis_0</b>	AXI-stream subordinate interface. Connect this port to the <b>m_axis</b> port on the UDP AXI Manager IP.



Port	Description
<b>m_axis_0</b>	AXI-stream manager interface. Connect this port to the <b>s_axis</b> port on the UDP AXI Manager IP.

### Ethernet MAC Hub IP Ports

Port	Direction	Description
<b>ref_clk</b>	Input	Reference clock signal that drives <b>phy_gtxclk</b> . The frequency of <b>ref_clk</b> must be the same as the <b>phy_rxclk</b> clock frequency.
<b>ref_reset</b>	Input	IP reset signal.
<b>phy_rxclk</b>	Input	Receive clock from PHY.
<b>phy_rxd</b>	Input	Receive data signal from PHY.
<b>phy_rxdv</b>	Input	Receive data valid control signal from PHY.
<b>phy_rxer</b>	Input	Receive error signal from PHY.
<b>rxclk_en</b>	Input	Receiver clock enable.
<b>phy_gtxclk</b>	Output	Clock to PHY.
<b>phy_txd</b>	Output	Transmit data signal to PHY.
<b>phy_txen</b>	Output	Transmit enable control signal to PHY.
<b>phy_txer</b>	Output	Transmit error signal to PHY.
<b>txclk_en</b>	Input	Transmitter clock enable.
<b>phy_col</b>	Input	Collision detect signal from PHY.
<b>phy_crs</b>	Input	Carrier sense detect signal from PHY.
<b>axis_aclk</b>	Input	Clock signal for AXI-stream interface.
<b>axis_aresetn</b>	Output	Active-low reset. Reset signal for AXI-stream interface. You can use this port to reset the downstream AXI peripherals.

After instantiating the Ethernet MAC Hub IP in your design, open the block parameters for configuration.

Ethernet MAC Hub	
ethernet_mac_hub	
<a href="#">Details</a>	
Parameters	
Number of AXI Stream Channels:	<input type="text" value="1"/>
MAC Address:	<input type="text" value="0xa3502218a"/>
IP Address Byte1:	<input type="text" value="192"/>
IP Address Byte2:	<input type="text" value="168"/>
IP Address Byte3:	<input type="text" value="1"/>
IP Address Byte4:	<input type="text" value="2"/>
UDP Port For Channel 1:	<input type="text" value="50101"/>
UDP Port For Channel 2:	<input type="text" value="50102"/>
UDP Port For Channel 3:	<input type="text" value="50103"/>
UDP Port For Channel 4:	<input type="text" value="50104"/>
UDP Port For Channel 5:	<input type="text" value="50105"/>
UDP Port For Channel 6:	<input type="text" value="50106"/>
UDP Port For Channel 7:	<input type="text" value="50107"/>
UDP Port For Channel 8:	<input type="text" value="50108"/>

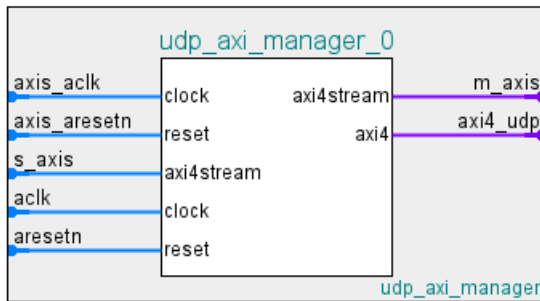
### Ethernet MAC Hub IP Parameters

Configure these parameters:

- **Number of AXI Stream Channels** — This parameter decides the number of AXI-stream channels in the Ethernet MAC Hub IP. Select this value as an integer from 1 to 8. The default value is 1.
- **IP Address Byte1, IP Address Byte2, IP Address Byte3, IP Address Byte4** — These parameters set the four bytes in the range from 0 to 255 composing the UDP internet protocol (IP) address of the device. This address must match the DeviceAddress property value of the aximanager object.
- **UDP Port For Channel 1, UDP Port For Channel 2, UDP Port For Channel 3, UDP Port For Channel 4, UDP Port For Channel 5, UDP Port For Channel 6, UDP Port For Channel 7, UDP Port For Channel 8** — These parameters set the UDP port numbers. Specify each parameter value as an integer from 255 to 65,535. These port numbers must match the Port property value of the aximanager object.

### UDP AXI Manager IP

The UDP AXI Manager HDL IP connects the Ethernet MAC Hub IP to your application IP. The UDP AXI Manager IP acts as a bridge that translates data between an AXI peripheral and MATLAB.



### Interface of UDP AXI Manager IP

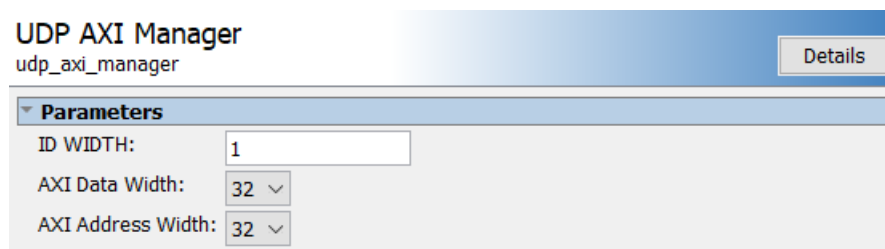
The interface of the UDP AXI Manager IP includes the ports in this table.

Port	Description
<b>s_axis</b>	AXI-stream subordinate interface.
<b>m_axis</b>	AXI-stream manager interface.
<b>axi4_udp</b>	AXI4-full manager interface.

### UDP AXI Manager IP Ports

Port	Direction	Description
<b>axis_aclk</b>	Input	Clock signal for AXI-stream interface.
<b>axis_aresetn</b>	Input	Active-low reset signal for AXI-stream interface.
<b>aclk</b>	Input	Clock signal for AXI4-full interface.
<b>aresetn</b>	Input	Active-low reset. Reset signal for AXI4-full interface.

After instantiating the UDP AXI Manager IP in your design, open the block parameters for configuration.



### UDP AXI Manager IP Parameters

Configure these parameters:

- **ID Width** — This parameter is the ID width in bits. Its value must match the ID width of the AXI4 subordinate.
- **AXI Data Width** — This parameter is the data bus width in bits. The IP supports 32 bits or 64 bits.
- **AXI Address Width** — This parameter is the address bus width in bits. The IP supports 32 bits or 64 bits.

When the program is running on your FPGA board, you can create an AXI manager object using the `aximanager` object. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` object functions.

### **See Also**

`aximanager`

### **Related Examples**

- “Access FPGA External Memory Using AXI Manager” on page 7-13

### **More About**

- “Set Up AXI Manager” on page 3-2

# AXI Manager Reference

---

## aximanager

Read and write memory locations on FPGA board from MATLAB

### Description

The `aximanager` object communicates with the AXI manager IP when it is running on an FPGA board. The object forwards read and write commands to the IP to access subordinate memory locations on the FPGA board. Before using this object, follow the steps in “Set Up AXI Manager” on page 3-2.

---

**Note** The `aximaster` object has been renamed to the `aximanager` object. For more information, see “Compatibility Considerations” on page 4-5.

---

### Creation

#### Syntax

```
mem = aximanager(vendor)
mem = aximanager(vendor,Name,Value)
```

#### Description

`mem = aximanager(vendor)` returns an object, that controls an AXI4 manager IP for the FPGA that is running on your board. `vendor` specifies the FPGA brand name. This connection enables you to access memory locations in an SoC design from MATLAB.

`mem = aximanager(vendor,Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name and value in quotes. For example, `'DeviceAddress','192.168.0.10'` specifies the internet protocol (IP) address of the FPGA board as 192.168.0.10.

#### Input Arguments

##### **vendor** — FPGA brand name

`'Intel' | 'Xilinx'`

FPGA brand name, specified as `'Intel'` or `'Xilinx'`. This value specifies the manufacturer of the FPGA board. The AXI manager IP varies depending on the type of FPGA that you specify.

### Properties

#### **Interface** — Type of interface used for communication with FPGA board

`'JTAG' (default) | 'PCIe' | 'UDP'`

Type of interface used for communication with the FPGA board, specified as `'JTAG'` (default), `'PCIe'`, or `'UDP'`. This value specifies the interface type for communicating between the host and the FPGA.

**JTAGCableName — Name of JTAG cable used for communication with FPGA board**

'auto' (default) | character vector | string scalar

Name of the JTAG cable used for communication with the FPGA board, specified as a character vector or string scalar representing a JTAG cable name. 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 more details, see “Select from Multiple JTAG Cables” on page 4-5.

Data Types: char | string

**DeviceAddress — IP address of Ethernet port on FPGA board**

'192.168.0.2' (default) | character vector | string scalar

Internet protocol (IP) address of the Ethernet port on the FPGA board, specified as a character vector or string scalar representing an IP address.

Example: '192.168.0.10'

**Dependencies**

To enable this property, set the Interface to 'UDP'.

Data Types: char | string

**DeviceType — Type of target device**

'FPGA' (default) | 'SoC'

Type of target device, specified as 'FPGA' (default) or 'SoC'. When you are using a Xilinx Zynq or an Intel SoC as a target device, specify this property as 'SoC'.

Example: 'SoC'

**Port — UDP port number of target FPGA board**

'50101' (default) | integer

User datagram protocol (UDP) port number of the target FPGA board, specified as an integer.

Example: '12345'

**Dependencies**

To enable this property, set the Interface property to 'UDP' and the DeviceType property to 'FPGA'.

Data Types: uint16

**Object Functions**

readmemory	Read data out of AXI4 memory-mapped subordinates
release	Release JTAG or Ethernet cable resource
writememory	Write data to AXI4 memory-mapped subordinates

**Examples**

## Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on an Intel® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA External Memory Using AXI Manager” on page 7-13 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Intel')
mem =
    aximanager with properties:
        Vendor: 'Intel'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
rd_d =
    uint32
    10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
rd_d =
    1x10 uint32 row vector
    10  11  12  13  14  15  16  17  18  19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
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 data 10 times to the same address. In this case, 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
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
rd_d = readmemory(mem,0x1c,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 no longer need to access the board, release the JTAG connection.

```
release(mem);
```

## Select from Multiple JTAG Cables

This example shows how to select the required JTAG cable from the multiple JTAG cables that are connected to your host computer.

If two cables of the same type are connected to your host computer, specify the “JTAGCableName” on page 4-0 property identifier for the board where the AXI manager IP is running. To see the JTAG cable identifiers, attempt to create an aximanager object. The object returns a list of the current JTAG cable names.

```
h = aximanager('Intel')

Error using fpgadebug_mex
Found more than one JTAG cable:
0 (Max10): #tpt_0001#ptc_0002#210203991642
1 (Arria): #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 AXI Manager
to learn how to set the cable name.
```

To communicate with the Arria board, specify the matching JTAG cable name.

```
h = aximanager('Intel','JTAGCableName','#tpt_0001#ptc_0002#210319789795');
```

## Version History

Introduced in R2017a

**R2022a: aximaster renamed to aximanager**

*Warns starting in R2022a*

The `aximaster` object has been renamed to the `aximanager` object. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To create an AXI manager object, use the `aximanager` object. Using the `aximaster` object is not recommended and will be removed in a future release. If you use the `aximaster` object, the object now gives this warning message.

```
aximaster has been renamed to aximanager in R2022a. aximaster will be removed
in a future release. Use aximanager instead.
```

**See Also****Topics**

"Access FPGA External Memory Using AXI Manager" on page 7-13

"Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow" (HDL Coder)

"Set Up AXI Manager" on page 3-2

# readmemory

**Package:** hdlverifier

Read data out of AXI4 memory-mapped subordinates

## 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 then incrementing the address for each word. The function typecasts the data to the `uint32` or `uint64` data type (depending on the data size of the AXI manager IP). The address, `addr`, must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. The `aximanager` object, `mem`, manages the connection between MATLAB and the AXI manager IP.

`data = readmemory(mem,addr,size,Name,Value)` specifies options using one or more name-value arguments.

## Examples

### Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on an Intel® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA External Memory Using AXI Manager” on page 7-13 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Intel')
mem =
    aximanager with properties:
        Vendor: 'Intel'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
```

```
rd_d =
    uint32
    10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
    1x10 uint32 row vector
    10  11  12  13  14  15  16  17  18  19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
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 data 10 times to the same address. In this case, 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
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
```

```
rd_d = readmemory(mem,0x1c,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 no longer need to access the board, release the JTAG connection.

```
release(mem);
```

## Input Arguments

### **mem** — Connection to AXI manager IP on FPGA board

`aximanager` object

Connection to the AXI manager IP on your FPGA board, specified as an `aximanager` object.

### **addr** — Starting address for read operation

nonnegative integer multiple of 4 | nonnegative hexadecimal value multiple of 4

Starting address for the read operation, specified as a nonnegative integer multiple of 4 or hexadecimal value multiple of 4. The function supports the address width of 32, 40, and 64 bits. The function casts the address to the `uint32` or `uint64` data type, depending on the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

### Memory-Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, 0x1 is an illegal address and emits an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, 0x1 and 0x4 are illegal and emit errors.
- If the AXI manager IP data width is 32 bits and you set the 'BurstType' argument to 'Increment', the address has 4-byte increments.
- If the AXI manager IP data width is 64 bits and you set the 'BurstType' argument to 'Increment', the address has 8-byte increments.
- If the AXI manager IP data width is 32 bits and you set the `OutputDataType` argument to 'half', the function reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits and you set the `OutputDataType` argument to 'half', the function reads the lower 2 bytes and ignores the higher 6 bytes.
- Do not use a 64-bit AXI manager for accessing 32-bit registers.

Example: 0xa4, specifies a starting address of 0xa4.

Data Types: `uint32` | `uint64`

### **size** — Number of memory locations to read

nonnegative integer

Number of memory locations to read, specified as a nonnegative integer. By default, the function reads data from a contiguous address block, incrementing the address for each operation. To disable address incrementation and read repeatedly from the same location, set the 'BurstType' argument to 'Fixed'.

When you specify a large operation size, such as reading a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size of 256 words.

Example: 5 specifies five contiguous memory locations.

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'BurstType', 'Fixed'` directs the AXI manager to read all data from the same address.

### **BurstType — AXI4 burst type**

`'Increment'` (default) | `'Fixed'`

AXI4 burst type, specified as one of these options:

- `'Increment'` — The AXI manager reads a vector of data from contiguous memory locations, starting with the specified address.
- `'Fixed'` — The AXI manager reads all data from the same address.

---

**Note** The `'Fixed'` burst type is not supported for the PCI Express interface. Use the `'Increment'` burst type instead.

---

### **OutputDataType — Data type assigned to read data**

`'uint32'` (default) | `'uint8'` | `'int8'` | `'uint16'` | `'int16'` | `'half'` | `'int32'` | `'single'` | `'uint64'` | `'int64'` | `'double'` | numeric type object

Data type assigned to the read data, specified as one of these options:

- `'int8'`
- `'uint8'`
- `'uint16'`
- `'int16'`
- `'half'`
- `'uint32'`
- `'int32'`
- `'single'`
- `'uint64'`
- `'int64'`
- `'double'`
- numeric type object

The function typecasts the data read out of the FPGA to the specified data type. `double` is supported for 64-bit UDP connections only.

### **Output Arguments**

#### **data — Read data**

scalar | vector

Read data, returned as a scalar or vector depending on the value you specify for the `size`. The function typecasts the data to the data type specified by the `'OutputDataType'` input.

## Version History

Introduced in R2017a

### R2023a: Support for half data type

The function reads `half` data from the memory locations on the FPGA board. The address for the read operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the function reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the function reads the lower 2 bytes and ignores the higher 6 bytes.

### See Also

`writememory` | `aximanager`

## release

**Package:** hdlverifier

Release JTAG or Ethernet cable resource

### Syntax

```
release(mem)
```

### Description

`release(mem)` releases the JTAG cable or Ethernet cable resource, depending on the interface that you use.

- When you use a JTAG interface, the function releases the JTAG cable resource, freeing the cable for use to reprogram the FPGA. After initialization, the AXI manager object, `mem`, holds the JTAG cable resource, and other programs cannot access the JTAG cable. While you have an active AXI manager object, FPGA programming over JTAG fails. Call the `release` function before reprogramming the FPGA.
- When you use an Ethernet interface, the function closes the Ethernet communications channel and clears the associated resources. During the creation of AXI manager object `mem`, the object initializes a communication stream to enable the exchange of data between the host computer and the target processor. Call the `release` function when you no longer need to access the board.

### Examples

#### Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on an Intel® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA External Memory Using AXI Manager” on page 7-13 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Intel')  
  
mem =  
    aximanager with properties:  
        Vendor: 'Intel'  
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.



```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
```

```
rd_d =
    uint32
     10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
rd_d =
```

```
1x10 uint32 row vector
    10    11    12    13    14    15    16    17    18    19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
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 data 10 times to the same address. In this case, 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
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
rd_d = readmemory(mem,0x1c,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 no longer need to access the board, release the JTAG connection.

```
release(mem);
```

## Input Arguments

**mem** — **Connection to JTAG-based AXI manager IP or to Ethernet-based AXI manager IP**  
aximanager object

Connection to a JTAG-based AXI manager IP or to an Ethernet-based AXI manager IP, specified as an aximanager object.

## Version History

**Introduced in R2017a**

## See Also

readmemory | writememory

# writememory

**Package:** hdlverifier

Write data to AXI4 memory-mapped subordinates

## 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. The address, `addr`, must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. The AXI manager object, `mem`, manages the connection between MATLAB and the AXI manager IP.

`writememory(mem, addr, data, Name, Value)` specifies options using one or more name-value arguments.

## Examples

### Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on an Intel® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA External Memory Using AXI Manager” on page 7-13 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Intel')
mem =
    aximanager with properties:
        Vendor: 'Intel'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem, 140, [10:19]);
rd_d = readmemory(mem, 140, 1)
rd_d =
```

```
uint32
```

```
10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
10 11 12 13 14 15 16 17 18 19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
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 data 10 times to the same address. In this case, 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
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
```

```
rd_d = readmemory(mem,0x1c,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 no longer need to access the board, release the JTAG connection.

```
release(mem);
```

## Input Arguments

### **mem** — Connection to AXI manager IP on FPGA board

`aximanager` object

Connection to the AXI manager IP on your FPGA board, specified as an `aximanager` object.

### **addr** — Starting address for write operation

nonnegative integer multiple of 4 | nonnegative hexadecimal value multiple of 4

Starting address for the write operation, specified as a nonnegative integer multiple of 4 or hexadecimal value multiple of 4. The function supports the address width of 32, 40, and 64 bits. The function casts the address to the `uint32` or `uint64` data type, according to the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

### Memory-Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, 0x1 is an illegal address and emits an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, 0x1 and 0x4 are illegal and emit errors.
- If the AXI manager IP data width is 32 bits and you set the 'BurstType' argument to 'Increment', the address has 4-byte increments.
- If the AXI manager IP data width is 64 bits and you set the 'BurstType' argument to 'Increment', the address has 8-byte increments.
- If the AXI manager IP data width is 32 bits and the input data is `half`, the function writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits and the input data is `half`, the function writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.
- Do not use a 64-bit AXI manager for accessing 32-bit registers.

Example: 64, specifies a starting address of 64.

Data Types: `uint32` | `uint64`

### **data** — Data words to write

scalar | vector

Data words to write, specified as a scalar or vector. By default, the function writes the data to a contiguous address block, incrementing the address for each operation. To disable address incrementation and write each data value to the same location, set the 'BurstType' argument to 'Fixed'.

Before sending the write request to the FPGA, the function typecasts the input data to the `uint32`, `int32`, `uint64`, or `int64` data type. The type conversion follows these rules:

- If the input data is `double`, then the data is typecast to `int32` or `int64`, depending on the AXI manager IP data width.

- If the input data is `single`, then the data is typecast to `uint32` or `uint64`, depending on the AXI manager IP data width.
- If the input data is `half`, then the data is typecast to `uint16` and packed to `uint32` or `uint64`, depending on the AXI manager IP data width.
- If the bit width of the input data type is less than the AXI manager IP data width, then the data is sign-extended to the width of the AXI manager IP data width.
- If the bit width of the input data type is greater than the AXI manager IP data width, then the data is typecast to `int32`, `uint32`, `int64`, `uint64`. The data is typecast to match the AXI manager IP data width and 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 of 256 words.

Example: `[1:100]` specifies 100 contiguous memory locations.

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

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'BurstType', 'Fixed'` directs the AXI manager to write all data to the same address.

### **BurstType — AXI4 burst type**

`'Increment'` (default) | `'Fixed'`

AXI4 burst type, specified as one of these options:

- `'Increment'` — The AXI manager writes a vector of data to contiguous memory spaces, starting with the specified address.
- `'Fixed'` — The AXI manager writes all data to the same address.

---

**Note** The `'Fixed'` burst type is not supported for the PCI Express interface. Use the `'Increment'` burst type instead.

---

## **Version History**

**Introduced in R2017a**

### **R2023a: Support for half data type**

The function writes `half` data to the memory locations on the FPGA board. Before sending the write request to the FPGA, the function typecasts the `half` input data to the `uint16` and then packs the data to `uint32` or `uint64`, depending on the AXI manager IP data width.

The address for the write operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the function writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the function writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.

**See Also**

readmemory | aximanager

## setupAXIManagerForQuartus

Add AXI manager IP path to Quartus project

### Syntax

```
setupAXIManagerForQuartus(projectName)  
setupAXIManagerForQuartus()
```

### Description

setupAXIManagerForQuartus(projectName) adds the AXI manager IP folder to the path of the Quartus project, projectName.

---

**Note** The setupAXIMasterForQuartus function has been renamed to the setupAXIManagerForQuartus function. For more information, see “Compatibility Considerations” on page 4-21.

---

setupAXIManagerForQuartus() displays the location of the AXI manager IP that is included with this support package.

### Examples

#### Add AXI Manager IP to FPGA Project

Call the setup function with a project file name. If the project is not in the current working folder, include the path.

```
setupAXIManagerForQuartus('aximaster_deca.qpf')
```

The location of the AXI manager IP is added to your project IP search path.

#### Find Location of AXI Manager IP

To find the folder in your MATLAB® installation that contains the AXI manager IP, call the setup function without a project file name.

```
setupAXIManagerForQuartus
```

```
The Quartus project was not specified. You can manually add ...  
C:\Program Files\MATLAB\R2022a\toolbox\hdlverifier\supportpackages ...  
\fpgadebug_intel\hdlverifier\+fpga\+quartus ...  
to the IP search path setting of your Qsys file.
```



To use the IP, add this path to your FPGA project.

## Input Arguments

### **projectName** — File name of Quartus project

character vector

File name of an existing Quartus project. This function modifies the project to add the location of the AXI manager IP to the IP search path. If you do not specify this argument, the function displays the path to the AXI manager IP.

## Version History

**Introduced in R2017a**

### **R2022a: setupAXIMasterForQuartus renamed to setupAXIManagerForQuartus**

*Warns starting in R2022a*

The `setupAXIMasterForQuartus` function has been renamed to the `setupAXIManagerForQuartus` function. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To display the path to the AXI manager IP, use the `setupAXIManagerForQuartus` function. Using the `setupAXIMasterForQuartus` function is not recommended and will be removed in a future release. If you use the `setupAXIMasterForQuartus` function, the function now gives this warning message.

`setupAXIMasterForQuartus` has been renamed to `setupAXIManagerForQuartus` in R2022a. `setupAXIMasterForQuartus` will be removed in a future release. Use `setupAXIManagerForQuartus` instead.

## See Also

### **Classes**

`aximanager`

### **Topics**

"Access FPGA External Memory Using AXI Manager" on page 7-13

"Set Up AXI Manager" on page 3-2



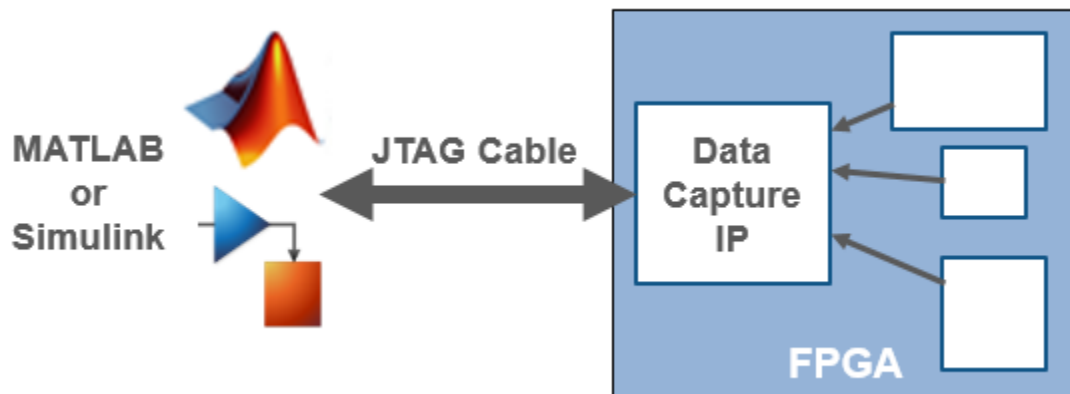
# FPGA Data Capture

---

- “Data Capture Workflow” on page 5-2
- “Triggers” on page 5-6
- “Design Considerations for Data Capture” on page 5-10
- “Capture Conditions” on page 5-12

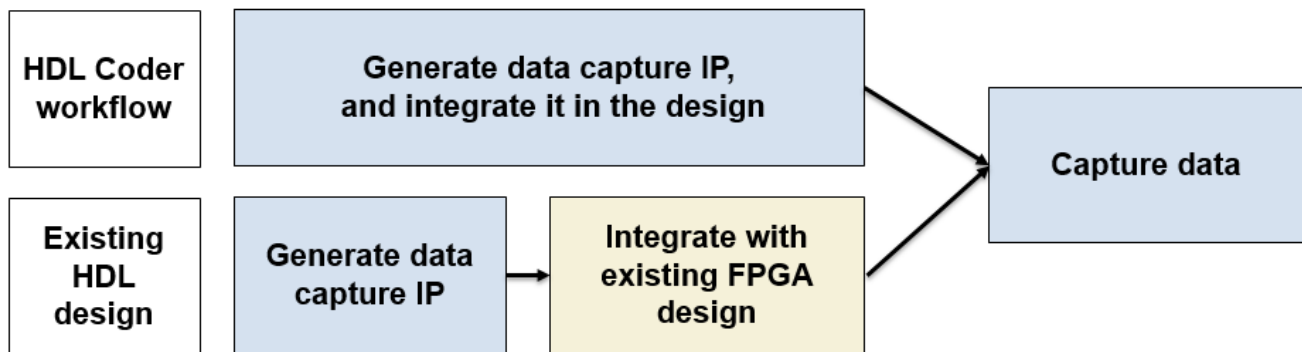
## Data Capture Workflow

Use FPGA data capture to observe signals from your design while the design is running on the FPGA. This feature captures a window of signal data from the FPGA and returns the data to MATLAB or Simulink.



There are two workflows to capture data from your FPGA board into MATLAB or Simulink:

- First workflow — When you generate the HDL IP with HDL Coder, use the **HDL Workflow Advisor** tool to generate the data capture IP and integrate it in the design.
- Second workflow — If you have an existing HDL design, HDL Verifier provides tools to generate the data capture IP. Then, manually integrate the generated IP into your FPGA design.



To capture signals from your design, HDL Verifier generates an IP core that communicates with MATLAB. Use the HDL Coder workflow for automatic integration of the data capture IP core in your design. Otherwise, manually integrate this IP core into your HDL project and deploy it to the FPGA along with the rest of your design. Then, use one of the following methods to capture data.

- For capturing data to MATLAB - HDL Verifier generates a customized tool that returns the captured signal data. Alternatively, you can use the generated System object to capture data programmatically.

- For capturing data to Simulink - HDL Verifier generates a block that has output ports corresponding to the signals you captured.

In both cases, you can specify data types for the captured data, number of windows to capture, trigger condition that controls when to capture the data, and capture condition that controls which data to capture.

When the design is running on the FPGA, first the generated IP core waits for the trigger condition that you specify. Define a trigger condition by specific values matched on one or more signals. When the trigger is detected, the logic captures the designated signals to a buffer and returns the data over the JTAG interface to the host machine. You can then analyze and display these signals in your MATLAB workspace or Simulink model.

To make the best use of the buffer size and capture only the valid data, you can also define a capture condition. Define a capture condition in the same way as you define the trigger condition. When both the trigger is detected and the capture condition is satisfied, the logic captures only the valid values of the designated signals.

## Generate and Integrate Data Capture IP Using HDL Workflow Advisor

When you use the **HDL Workflow Advisor** tool to generate your HDL design, first mark interesting signals as test points in Simulink. Configure your design using the **HDL Workflow Advisor** tool to:

- Select the type of connection channel by setting the **FPGA Data Capture (HDL Verifier required)** parameter in the **Set Target Reference Design** task. For more information, see “Set Target Reference Design” (HDL Coder).
- Enable test point generation by selecting the **Enable HDL DUT port generation for test points** parameter in the **Set Target Interface** task. For more information, see “Set Target Interface” (HDL Coder).
- Connect test point signals to the FPGA Data Capture interface in the **Set Target Interface** task.
- Set up buffer size and maximum sequence depth for data collection in the **Generate RTL Code and IP Core** task. To include capture condition logic in the IP core, select **Include capture condition logic in FPGA Data Capture**. For more information, see “Generate RTL Code and IP Core” (HDL Coder).

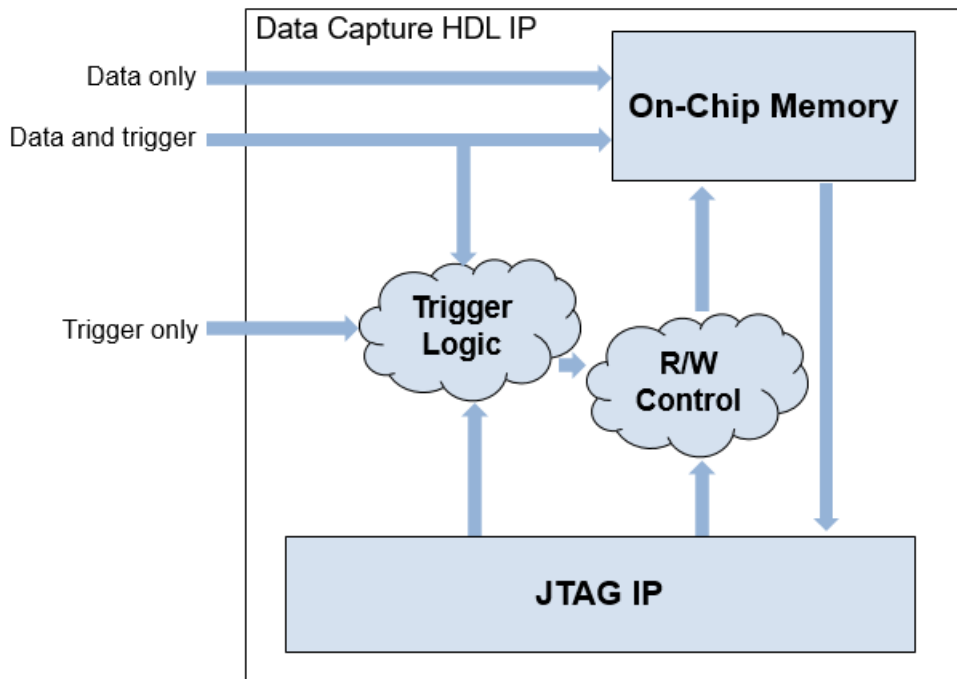
Then, run through the remaining steps to generate HDL for your design and program the FPGA. The data capture IP core is integrated in the generated design. You are now ready to “Capture Data” on page 5-4.

## Configure and Generate IP Core for an Existing HDL Design

Before you can capture FPGA data, first specify which signals to capture, and how many samples to return. Use the **FPGA Data Capture Component Generator** to configure these and other settings, and to generate the HDL IP core. The IP core contains:

- A port for each signal you want to capture or use as part of a trigger condition
- Memory to capture the number of samples you requested for each signal
- JTAG interface logic to communicate with MATLAB
- Trigger and capture condition logic that can be configured at run time

- A ready-to-capture signal to control data flow from the FPGA



The tool also generates a customized **FPGA Data Capture** tool, System object, and model that communicate with the FPGA.

## Integrate IP into FPGA

For MATLAB to communicate with the FPGA, you must integrate the generated HDL IP core into your FPGA design. If you used the **HDL Workflow Advisor** tool to generate your data capture IP, this step is automated. In this case, data capture IP operates on a single-clock rate, which is the primary clock of your device under test (DUT). If you did not use the **HDL Workflow Advisor** tool, follow the instructions in the generation report. Add the generated HDL files in the `hdlsrc` folder into your FPGA project. Then, instantiate the HDL IP core, `datacapture`, in your HDL code and connect it to the signals you requested for capture and triggers. Compile the project and program the FPGA with the new image.

## Capture Data

The FPGA data capture IP core communicates over the JTAG cable between your FPGA board and the host computer. Make sure that the JTAG cable is connected. Before capturing data, you can set data types for the captured data, set trigger condition that specifies when to capture the data, and set a capture condition that specifies the data to be captured. To configure these options and capture data, you can:

- Open the **FPGA Data Capture** tool. Set the trigger, capture condition, and data type parameters, and then capture data into the MATLAB workspace.

- Use the generated System object derived from `hdlverifier.FPGADataReader`. Set the data types, trigger condition, and capture condition using the methods and properties of the System object, and then call the object to capture data.
- In Simulink, open the generated model and configure the parameters of the FPGA Data Reader block. Then run the model to capture data.

After you capture the data and import it into the MATLAB workspace or Simulink model, you can analyze, verify, and display the data.

## See Also

**FPGA Data Capture Component Generator | FPGA Data Capture |**  
`hdlverifier.FPGADataReader` | FPGA Data Reader

## Related Examples

- “Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

## Triggers

### What Is a Trigger Condition?

A trigger condition is a logical statement that defines when to capture data from the FPGA. Use a trigger condition to capture data around an event of interest on the FPGA. Capture multiple occurrences of an event by setting Number of capture windows to the desired value. A trigger condition is composed of value comparison tests on one or more FPGA signals. For example:

```
counter == 100
```

All trigger comparisons are synchronous. When you specify an edge condition for a Boolean signal, the IP core compares the current sampled value with the sampled value from the previous clock cycle.

```
fifo_full == 'Rising edge'
```

The trigger condition is met when all terms of the condition are true on the same clock cycle. You can use only a single value comparison per signal.

```
receiver_state == 3 OR message_detected == 'High'
```

```
fifo_cnt == 0 AND fifo_pop == 'High'
```

You can use only a single type of logical operator in the trigger condition. You cannot mix AND and OR conditions.

```
fifo_empty == 'Rising edge' OR fifo_full == 'Rising edge' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 5
```

You can use multiple comparison operators in the trigger condition.

```
fifo_empty == 'Rising edge' OR fifo_full != 'LOW' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr > 148 AND pkt_type >= 5
```

You can use X or x (don't-care value) in the trigger condition. While comparing, the trigger condition ignores the place values with X. When the trigger condition is 0b1X1, the possible trigger condition values are 0b101 or 0b111.

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 0b1X1
```

### Sequential Trigger

A sequential trigger enables you to give a set of trigger conditions in multiple stages to capture specified data from an FPGA. With a sequential trigger, you can read data to MATLAB or Simulink only after all of the specified trigger conditions happen in sequence. For multiple trigger stages, set the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool to a value greater than 1. **Max trigger stages** sets the maximum number of trigger stages for providing trigger conditions. For example, if **Max trigger stages** is 3, the **Trigger** tab in the **FPGA Data Capture** tool or in the FPGA Data Reader block can have maximum of 3 trigger stages.



Trigger   Capture Condition   Data Types

Sample depth: 128  
 Number of capture windows: 1   Number of trigger stages: 3  
 Trigger position: 0

Trigger Stage 1

Signal	Operator	Value	Change operator
signal1	+		AND

Trigger Stage 2

Signal	Operator	Value	Change operator
signal1	+	<input type="checkbox"/> Trigger time out 1	AND

Trigger Stage 3

Signal	Operator	Value	Change operator
signal1	+	<input type="checkbox"/> Trigger time out 1	AND

Status: Not started   On trigger   Capture Data

Define a trigger condition by specific values matched on one or more signals in each stage. For example, if the number of trigger stages is 3 and 10 signals exist, you can set these trigger conditions.

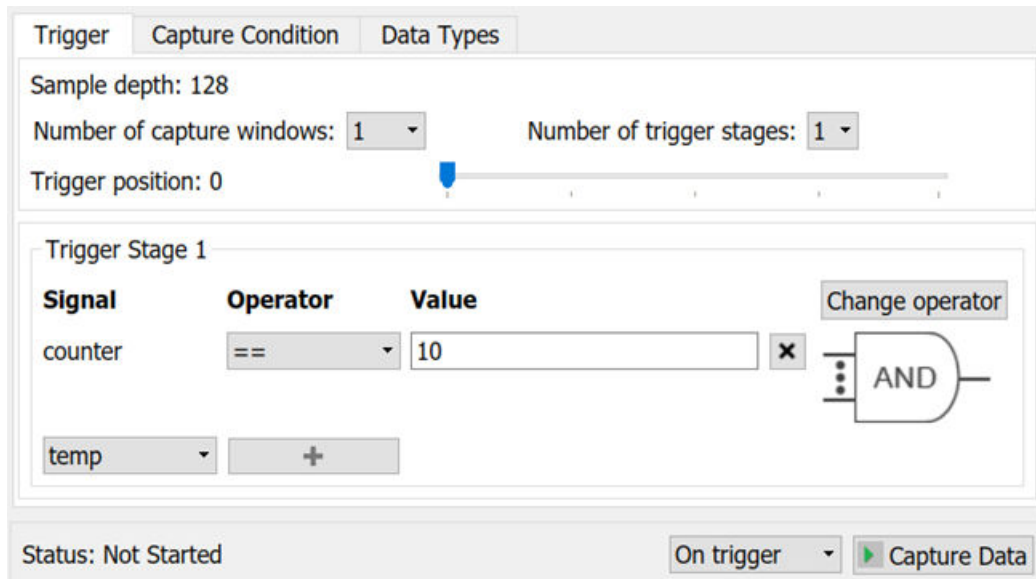
- Trigger condition for stage 1:  
`((signal1 > 10) and (signal3 == true) and (signal7 < 5));`
- Trigger condition for stage 2:  
`((signal1 == 0b0110) or (signal4 == 0XXX) or (signal8 < 5));`
- Trigger condition for stage 3:  
`((signal2 != 5) and (signal6 == true) and (signal8 == 8));`

## Configure a Trigger Condition

At generation time, specify which signals you want to be available for use in trigger conditions. A signal can be a trigger without capturing data, or it can be both a trigger and a captured signal. You can modify the trigger condition at capture time, using any signals you specified as triggers. The data capture IP core on the FPGA receives the trigger definition from MATLAB and configures on-chip muxes to detect the event.

When you use the **FPGA Data Capture** tool, or the FPGA Data Reader block, set the trigger condition on the **Trigger** tab. Each line in the table is the value comparison for one signal. To

combine the signal values, use the trigger combination operator. To show a signal on this tab, you must specify the signal as a trigger at generation time.



For an `hdlverifier.FPGADataReader` System object, configure the trigger condition using the `setTriggerCondition`, `setTriggerComparisonOperator`, and `setTriggerCombinationOperator` object functions. To check your configuration, call the `displayTriggerCondition` object function.

If you do not enable a trigger condition on any signal, the data capture IP core captures data immediately.

## Trigger Position

You can change the relative position of the trigger detection cycle within the capture buffer. Use this feature to capture the relevant data, whether it is before or after the trigger event.

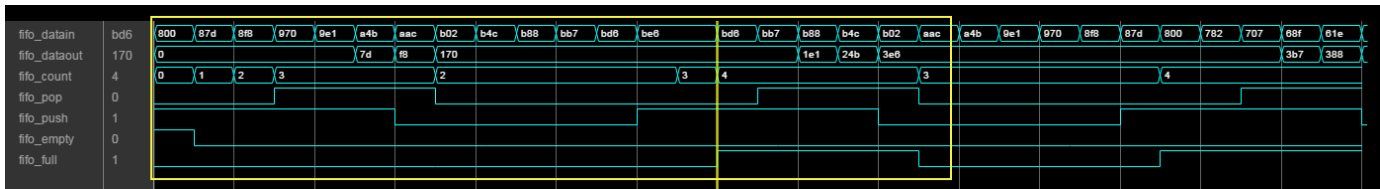
Suppose you want to debug the rates of pushes and pops to a FIFO design. You can set a trigger on a High value of the signals `fifo_empty` or `fifo_full`.



By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. The IP core captures a buffer starting from the cycle when `fifo_full` changes to high.



To debug the `fifo_full` condition, observe the signals before the trigger condition occurs. In the capture settings, change the **Trigger position** to 3/4 of the window depth using the tic mark on the slider. For example, if **Sample depth** is 128, **Number of capture windows** is 1, and **Number of trigger stages** is 1, then *window depth* is 128. The trigger event is at sample 96 of that window. The IP core captures a buffer that contains 96 samples before the trigger event, and 36 samples after the trigger event. This setting captures data that shows the lead-up to the trigger event, and the aftermath. The location of the trigger event is shown with the vertical cursor at `fifo_full`.



You can set the **Trigger position** to a number of samples between 0 and the *window depth*-1, inclusive. When you set the trigger position equal to *window depth*-1, the last sample corresponds to the cycle when the trigger occurs.

To observe more than one occurrence of the trigger event, change the **Number of capture windows** to the desired number.

In this example, **Number of capture windows** is 4, **Number of trigger stages** is 1, **Sample depth** is 128, and **Trigger position** is 0. HDL IP captures four windows, where each *window depth* is 32 samples, starting when `fifo_full` changes to high.



## See Also

### Tools

FPGA Data Capture Component Generator | FPGA Data Capture

### Objects

hdlverifier.FPGADataReader

### Blocks

FPGA Data Reader

## Related Examples

- “Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

## More About

- “Data Capture Workflow” on page 5-2

## Design Considerations for Data Capture

### Signals to Capture

To get started with FPGA data capture, you must specify port names and sizes for the generated IP. You then connect these ports to the signals in your design that you want to capture. You can specify bit widths between 1 and 128 bits. The default data type of the captured data depends on this bit width.

The FPGA data capture tools do not limit the total number of signals or bits you can capture. You are limited only by the hardware resource usage on your FPGA. When you select signals and the depth of the capture buffer, consider the memory and signal routing resources required on the FPGA.

In the **FPGA Data Capture Component Generator**, you can specify a signal for use as data or trigger. When you specify a signal as data, the signal is captured to the sample buffer and returned to MATLAB, but it cannot contribute to a trigger condition and capture condition. A data signal uses memory resources on the FPGA. When you specify a signal as a trigger, it is available for defining a trigger condition and capture condition at capture time, but is not captured and returned to MATLAB. A trigger signal uses logic resources on the FPGA. You can also specify that the signal is used as both trigger and data.

At capture time, you can configure the data type of the variable returned to MATLAB or Simulink. You can select built-in types, or, with Fixed-Point Designer™, you can specify fixed-point data types. If you do not have Fixed-Point Designer, data capture can only return built-in data types, such as `uint8`. In this case, you must specify ports for the generated IP that match the sizes of the built-in data types, that is, 1, 8, 16, 32, or 64 bits.

### Capture Timing

The data capture feature captures a fixed-size buffer of data each time you request a capture. The feature does not stream continuous data from your FPGA into MATLAB or Simulink. You can capture a buffer immediately, or you can configure a logical trigger condition to control when the buffer is captured. You can configure the timing of the capture relative to the cycle the trigger is detected in, and configure the capturing of multiple windows of trigger events. You can also configure a logical capture condition to filter the data to be captured. While the data capture IP waits for a trigger, captures data, and returns the captured data to MATLAB, you cannot initiate a new capture request. Therefore, you cannot capture back-to-back buffers from the FPGA.

Use this feature to investigate design behavior around a specific event or to sample data occasionally, rather than for continuous observation. For more information about how to use trigger condition and capture condition, see “Triggers” on page 5-6 and “Capture Conditions” on page 5-12, respectively.

### JTAG Considerations

The generated data capture IP can coexist in your design with other IPs that use the JTAG connection, such as Altera SignalTap II or Xilinx Vivado Logic Analyzer cores. However, only one of these applications can use the JTAG cable at a time. You must close the **FPGA Data Capture** tool or model, or release the object, to return the JTAG resource for use by other applications.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. You must stop any FPGA data capture or AXI manager JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Intel boards, the JTAG clock frequency is 12 or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

### **Simultaneous Use of FPGA Data Capture and AXI Manager**

The nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. You do not need to close or release the JTAG resource to switch between FPGA data capture and AXI manager.

FPGA data capture supports these two capture modes.

- Blocking mode — FPGA data capture blocks MATLAB while retrieving captured data. In this capture mode, the JTAG resource is allocated to either FPGA data capture or AXI manager at a time.
- Nonblocking mode — FPGA data capture does not block MATLAB while retrieving captured data. In this capture mode, you can use FPGA data capture and AXI manager simultaneously.

By default, FPGA data capture is configured in blocking mode. Change the capture mode to nonblocking mode by using the `CaptureMode` property for an `hdlverifier.FPGADataReader` System object. After changing the capture mode to nonblocking, you can use the command line interface or graphical user interface for performing the remaining steps in FPGA data capture and AXI manager. For an example, see “Debug IP Core Using FPGA Data Capture” (HDL Coder).

### **See Also**

#### **More About**

- “Intel FPGA Board Support from HDL Verifier” on page 1-2
- “Supported EDA Tools and Hardware” on page 1-5
- “Data Capture Workflow” on page 5-2

## Capture Conditions

### What Is Capture Condition?

A capture condition is a logical statement that controls which data to capture from the FPGA. Use a capture condition when you want to:

- Capture only the valid data to debug custom designs with FPGA data capture.
- Filter the data to capture based on trigger conditions.
- Optimize the use of an FPGA data capture buffer.
- Efficiently analyze the captured data when you have only a few captured samples of interest.

A capture condition is composed of value comparison tests on one or more FPGA signals. For example:

```
counter == 100
```

All capture comparisons are synchronous. When you specify an edge condition for a Boolean signal, the IP core compares the current sampled value with the sampled value from the previous clock cycle.

```
fifo_full == 'Rising edge'
```

The capture condition is met when all terms of the condition are true on the same clock cycle. You can use only a single value comparison per signal.

```
receiver_state == 3 OR message_detected == 'High'
```

```
fifo_cnt == 0 AND fifo_pop == 'High'
```

You can use only a single type of logical operator in the capture condition. You cannot mix AND and OR conditions.

```
fifo_empty == 'Rising edge' OR fifo_full == 'Rising edge' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 5
```

You can use multiple comparison operators in the capture condition.

```
fifo_empty == 'Rising edge' OR fifo_full != 'LOW' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr > 148 AND pkt_type >= 5
```

You can use X or x (don't-care value) in the capture condition. While comparing, the capture condition ignores the place values with X. When the capture condition is 0b1X1, the possible trigger condition values are 0b101 or 0b111.

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 0b1X1
```

### Configure Capture Condition

At generation time, specify which signals you want to be available for use in the capture condition. You can use a signal containing only a trigger or both a trigger and captured data. You can modify the capture condition at capture time using any signals you specify as triggers. Also, at generation time, you can include capture condition logic to use the capture condition. The data capture IP core on the

FPGA receives the capture definition from MATLAB and configures on-chip muxes to detect the capture event.

When you use the **FPGA Data Capture** tool or the FPGA Data Reader block, on the **Capture Condition** tab, select the **Enable capture condition logic** parameter and then set the capture condition. Each line in the table is the value comparison for one signal. To combine the signal values, use the capture condition combination operator. To show a signal on this tab, you must specify the signal as a trigger at generation time.

Signal	Operator	Value
valid	==	High
ready	+	

Status: Not started      On trigger      Capture Data

For an `hdlverifier.FPGADataReader` System object, enable capture condition logic using the `EnableCaptureCtrl` property. Then configure the capture condition using the `setCaptureCondition`, `setCaptureConditionComparisonOperator`, and `setCaptureConditionCombinationOperator` object functions. To check your configuration, call the `displayCaptureCondition` object function.

## Differences Between Triggers and Capture Conditions

The trigger condition controls when to capture data from the FPGA. Once the trigger condition is satisfied, the data capture IP core captures data from that trigger event. The capture condition controls which data to capture. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition.

FPGA data capture operates in two modes: immediate mode and trigger mode. The data capture IP core captures data from the FPGA without checking for the trigger condition in immediate mode and based on a trigger condition in trigger mode. You can provide a capture condition in both modes.

Filter the data to capture using a capture condition. In immediate mode, use a capture condition to capture data only when a certain condition is met. In trigger mode, use a capture condition to capture data only when a certain condition is met after satisfying the trigger condition.

Data Capture Mode	Trigger Condition Only	Capture Condition Only	Both Trigger and Capture Conditions
Immediate	<p>Ignores all trigger conditions</p> <p>Captures data immediately at each clock cycle</p>	Captures data only when the capture condition is true	<p>Ignores all trigger conditions</p> <p>Captures data only when the capture condition is true</p>
Trigger	Waits until the trigger condition is true, then captures data	Not supported, use immediate mode instead	Waits until the trigger condition is true, then captures data only when the capture condition is true

## See Also

### Tools

**FPGA Data Capture Component Generator | FPGA Data Capture**

### Objects

`hdlverifier.FPGADataReader`

### Blocks

FPGA Data Reader

## More About

- “Data Capture Workflow” on page 5-2



# Data Capture Reference

---

# FPGA Data Capture Component Generator

Configure and generate FPGA data capture components

## Description

The **FPGA Data Capture Component Generator** tool configures and generates components for capturing data from a design running on an FPGA. The generated components capture a window of signal data from the FPGA and return the data to MATLAB or Simulink.

**Description**  
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

```

graph LR
    A[Generate data capture IP] --> B[Integrate with existing FPGA design]
    B --> C[Capture data]
  
```

[Read more about the data capture workflow](#)

**Ports**

Port Name	Bit Width	Use As	
	1	Both trigger and data	<input type="button" value="Add"/> <input type="button" value="Remove"/>

**Target**

Generated IP name:

FPGA vendor:

Generated IP language:

Connection type:

Destination folder:

**Capture**

Sample depth:

Max trigger stages:

Include capture condition logic

To use this tool, you must have an existing HDL design and FPGA project. To capture the signals, HDL Verifier generates an IP core that you must integrate into your HDL project, and deploy to the FPGA along with the rest of your design.

The **Generate** button in this tool generates these components:

- HDL IP core, for integration into your FPGA design. Connect the signals you want to capture and use as triggers, and connect a clock and clock enable.
- Generation report, with list of generated files and instructions for next steps.
- Tool to set capture parameters and capture data to the MATLAB workspace. See **FPGA Data Capture**.
- Customized version of the `hdlverifier.FPGADataReader` System object that provides an alternative, programmatic, way to configure and capture data.
- Simulink model that contains a customized FPGA Data Reader block. If you have a DSP System Toolbox™ license, this model streams the captured signals into the **Logic Analyzer** waveform viewer. Otherwise, the Scope block displays the signals.
- MAT file in the `datacapture_gensettings.mat` format, where `datacapture` is the name of the generated HDL IP core. This MAT file holds the data capture build parameters. To reload the same design in your next iteration, provide this MAT file as an input argument to the `generateFPGADataCaptureIP` function.

For a workflow overview, see “Data Capture Workflow” on page 5-2.

## Open the FPGA Data Capture Component Generator

At the MATLAB command prompt, enter this command.

```
generateFPGADataCaptureIP
```

To reload the parameters of the most recent design, use the `restore` argument.

```
generateFPGADataCaptureIP('restore',true);
```

To reload the parameters of a design you already generated and saved in a MAT file, use the `matFile` argument.

```
generateFPGADataCaptureIP('datacapture_gensettings.mat');
```

Where `datacapture` is the name of the generated HDL IP core that you specify in the **Generated IP name** parameter.

## Examples

- “Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

## Parameters

### Ports

**Port Name** — Name of input port on generated IP  
character vector | string scalar

The name does not have to match the signal name in your HDL files. This name is used for:

- Input port on the generated HDL IP core. Internal to the IP, this signal is routed to the capture buffer, or to use as part of trigger condition and capture condition, depending on your selection for **Use As**.
- Structure field in the captured data returned to the MATLAB workspace
- Port on the generated Simulink block
- Table of signals in the trigger, capture condition, and data types parameters editor at capture time

Data Types: `char` | `string`

**Bit Width** — Number of bits in signal  
positive integer

This number is used to generate the HDL IP port definition, and contributes to the total width of the capture buffer. You can specify the data type for the captured data at capture time.

---

**Note** If you do not have Fixed-Point Designer, data capture can only return built-in data types, such as `uint8`. You must specify ports for the generated IP that match the sizes of the built-in data types, that is 1, 8, 16, 32, or 64 bits. We recommend Fixed-Point Designer to enable fixed-point data types and captured signals of any size.

---

**Use As** — How signal is routed inside IP logic  
`Both trigger and data` (default) | `Data` | `Trigger`

When you specify a signal as `Data`, the signal is captured to the sample buffer and returned to MATLAB, but it cannot contribute to a trigger condition and capture condition. When you specify a signal as `Trigger`, it is available for defining a trigger condition and capture condition at capture time, but is not captured and returned to MATLAB. You can also specify that the signal is used as `Both trigger and data`.

### Target

**Generated IP name** — Name of generated components  
`datacapture` (default) | character vector

This name is used for the generated HDL IP core, the System object, and the Simulink model.

**FPGA vendor** — FPGA and software vendor  
`Altera` (default) | `Xilinx`

The available vendors depend on which HDL Verifier support package you have installed. There are separate support packages for Intel (Altera) and Xilinx boards.

**Generated IP language** — Language used for generated HDL IP core  
`VHDL` (default) | `Verilog`

Select the language used for the generated HDL IP core as `Verilog` or `VHDL`.

**Connection type** — Type of connection channel  
`JTAG` (default) | `Ethernet`

Select the type of connection channel as `JTAG` or `Ethernet`.

---

**Note** Ethernet connection is available for Xilinx FPGA boards only.

---

**Destination folder** — Location to save generated files

hdlsrc (default) | character vector | string scalar

Location to save the generated files, specified as the name of a folder on the host computer.

Data Types: char | string

### Capture

**Sample depth** — Number of samples captured for each signal

128 (default) | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 131072 | 262144 | 524288 | 1048576

Use this parameter to specify the size of the memory in the generated HDL IP core. The width of the memory is the total bit width of the data signals.

When you specify the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the sample depth divided by the number of capture windows. Specify the number of capture windows by using the **Number of capture windows** parameter in the **FPGA Data Capture** tool or by using the property for an `hdlverifier.FPGADataReader` System object.

For example, if the sample depth is 4096 and the number of capture windows is 4, then each capture window has a window depth of 1024.

**Max trigger stages** — Maximum number of trigger stages for providing trigger conditions

1 (default) | integer from 1 to 10

Use this parameter to enable a sequential trigger. To capture specified data from an FPGA, give a set of trigger conditions in multiple stages. For more information on sequential trigger, see “Sequential Trigger” on page 5-6.

When you specify the **Max trigger stages**, consider the maximum number of trigger stages in which you plan to configure the trigger conditions. Specify the number of trigger stages by using the **Number of trigger stages** parameter in the **FPGA Data Capture** tool or by using the `NumTriggerStages` property for an `hdlverifier.FPGADataReader` System object.

For example, if the maximum number of trigger stages is 4, then the number of trigger stages can be 1, 2, 3, or 4.

**Include capture condition logic** — Option to include capture condition logic in HDL IP core

off (default) | on

Select this parameter to include capture condition logic in the HDL IP core. Include capture condition logic to use a capture condition to control which data to capture from the FPGA. The HDL IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 5-12.

Set up a capture condition in the **FPGA Data Capture** tool or the `hdlverifier.FPGADataReader` System object.

## **Version History**

**Introduced in R2017a**

### **See Also**

**FPGA Data Capture** | generateFPGADataCaptureIP

### **Topics**

“Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

“Data Capture Workflow” on page 5-2

# generateFPGADataCaptureIP

Open FPGA Data Capture Component Generator

## Syntax

```
generateFPGADataCaptureIP  
generateFPGADataCaptureIP('restore',true)  
generateFPGADataCaptureIP(matFile)
```

## Description

generateFPGADataCaptureIP opens the **FPGA Data Capture Component Generator**.

generateFPGADataCaptureIP('restore',true) opens the **FPGA Data Capture Component Generator** and reloads the parameters of the most recent design.

generateFPGADataCaptureIP(matFile) opens the **FPGA Data Capture Component Generator** and reloads the parameters of a design you already generated and saved in matFile.

## Examples

### Launch FPGA Data Capture Component Generator

This example shows how to launch the **FPGA Data Capture Component Generator** tool.

### Open Tool With Default Settings

Run the following command to launch the **FPGA Data Capture Component Generator** tool:

```
generateFPGADataCaptureIP
```

FPGA Data Capture Component Generation
✕

---

**Description**

Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP

→

Integrate with existing FPGA design

→

Capture data

[Read more about the data capture workflow](#)

---

**Ports**

Port Name	Bit Width	Use As	
sig1	1	Both trigger and data	<div style="border: 1px solid gray; padding: 2px; width: 100px; margin: 0 auto;">Add</div> <div style="border: 1px solid gray; padding: 2px; width: 100px; margin: 0 auto;">Remove</div>

---

**Target**

Generated IP name:

FPGA vendor:

Generated IP language:

Connection type:

Destination folder:

**Capture**

Sample depth:

Max trigger stages:

Include capture condition logic

## Input Arguments

### **restore** – Option to reload most recent design parameters

false (default) | true

Option to reload the most recent design parameters, specified as `true` or `false`. When you specify this input as `true`, the **FPGA Data Capture Component Generator** opens and reloads the parameters of the most recent design. Otherwise, the tool opens with default settings.

Example: `'restore', true`

Data Types: `logical`

### **matFile** – MAT file containing design parameters

character vector | string scalar



MAT file containing design parameters, specified as a character vector or string scalar. Specify this input in the form `datacapture_gensettings.mat`, replacing `datacapture` with the generated HDL IP core name.

This MAT file is a generated file in the same folder as your other generated data capture components. When you specify this input, the **FPGA Data Capture Component Generator** opens and reloads the parameters of the design specified by this value.

You must provide only the MAT file name if the file is in the current working folder.

Example: `'datacapture_gensettings.mat'`

You must provide the full path to the saved MAT file if the file is not in the current working folder.

Example: `'C:/home/user/datacapture_gensettings.mat'`

Data Types: `char` | `string`

## Version History

Introduced in R2017a

### R2023a: Reload previous design parameters

The function reloads the parameters of a design you already generated and saved in a MAT file.

To reload the same design in your next iteration, provide the MAT file containing design parameters as an input argument to the function.

```
generateFPGADataCaptureIP('datacapture_gensettings.mat');
```

Where `datacapture` is the **Generated IP name** that you specify in the **FPGA Data Capture Component Generator** tool.

## See Also

**FPGA Data Capture Component Generator**

### Topics

“Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

# FPGA Data Capture

Capture data from live FPGA into MATLAB workspace interactively

## Description

The **FPGA Data Capture** tool captures data from a design running on an FPGA and returns it to the MATLAB workspace. You can configure the data types of the returned values, specify the number of capture windows and number of trigger stages, set up a trigger condition to control when the data is captured, and set up a capture condition to control which data to capture.

The screenshot shows the 'FPGA Data Capture' tool window. It includes a description, a workflow diagram, output settings, and trigger configuration options.

**Description:** Capture data from a design running on your FPGA board. Specify data types for the returned data structure, and specify a logical trigger condition that defines when the data is captured.

**Workflow:** Generate data capture IP → Integrate with existing FPGA design → Capture data

**Output:** Output variable name: dataCaptureOut.  Display data with Logic Analyzer

**Trigger Configuration:**

- Sample depth: 4096
- Number of capture windows: 1
- Number of trigger stages: 2
- Trigger position: 0

**Trigger Stage 1:**

Signal	Operator	Value
in1	+	

Change operator: AND

**Trigger Stage 2:**

Signal	Operator	Value
in1	+	

Change operator: AND

Trigger time out: 1

**Status:** Not started. **Immediately**

Before using this tool, you must have generated the customized data capture components using the **FPGA Data Capture Component Generator** tool. You must also integrate the generated IP code

into your project and deploy it to the FPGA. The tool communicates with the FPGA over a JTAG cable. Make sure that the JTAG cable is connected between the board and the host computer.

The tool is a wrapper over your generated `hdlverifier.FPGADataReader` System object. The **FPGA Data Capture** tool defines the variable, `fpgadc_obj` in the workspace. If this variable already exists, the tool opens using the existing object, and saves modifications to that object.

For a workflow overview, see “Data Capture Workflow” on page 5-2.

## Open the FPGA Data Capture

- MATLAB command prompt: Enter `launchDataCaptureApp`. This function is a generated script in the same folder as your other generated data capture components.

## Examples

- “Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

## Parameters

**Capture Data** — When to capture data

`Immediately` (default) | `On trigger`

The default setting, capture `Immediately`, ignores any trigger condition and captures the buffer of data when you click **Capture Data**. To capture data that includes a particular event in the FPGA logic, configure a trigger condition and select `On trigger`. In this case, the data capture logic waits until the trigger condition is true, then captures the buffer of data.

When you click **Capture Data**, a window with a **Stop** button opens. If you want to cancel the capture attempt (for example, if the trigger condition does not occur), click **Stop** to return control to the tool. When you abort a capture attempt, no data is returned to MATLAB.

### Output

**Output variable name** — Name of structure in which to return captured data

`dataCaptureOut` (default) | character vector

The captured data is returned to a structure variable in the base MATLAB workspace. The data returned from each signal is a vector of `Sample depth` values. Each signal becomes a field in the structure. The field name in the structure is the same as the **Signal Name**.

**Display data with Logic Analyzer** — Automatically display data in Logic Analyzer

`on` (default) | `off`

This option appears if you have a DSP System Toolbox license. When you select this option, after data capture is complete, the tool opens the Logic Analyzer window to display the captured data. The time axes is measured in samples. The cursor location indicates the time the trigger was detected.

### Trigger

**Sample depth** — Number of samples captured for each signal

integer power of two

This parameter is read-only. It reflects the value you specified at generation time.

**Number of capture windows** — Number of data capture recurrences

1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, and cannot be greater than **Sample depth**. When specifying the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the **Sample depth** divided by the **Number of capture windows**. Specify **Sample depth** in the **FPGA Data Capture Component Generator** tool.

For example: If **Sample depth** is 4096 and **Number of capture windows** is 4, then each capture window has a window depth of 1024.

**Number of trigger stages** — Number of trigger stages for providing trigger conditions

$M$  (default) | integer from 1 to  $M$

Specify the number of trigger stages. This value must be an integer from 1 to  $M$ , where  $M$  is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

For example, if **Max trigger stages** is 4, then **Number of trigger stages** can be 1, 2, 3, or 4.

**Trigger position** — Position of the trigger detection cycle within the capture buffer

0 (default) | integer up to  $window\ depth - 1$

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to any number from 0 to  $window\ depth - 1$ , inclusive. When the trigger position is equal to the  $window\ depth - 1$ , the last sample corresponds to the cycle when the trigger occurs. If **Number of capture windows** is greater than one, the same trigger position applies to all windows. For more information, see “Triggers” on page 5-6.

**Signal** — Trigger component signal name

character vector

This parameter is read-only. The signal names you specified at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the trigger condition.

**Operator** — Operator to compare signals within trigger condition

== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

**Value** — Value to compare signal to as part of overall trigger condition

decimal | binary | hexadecimal | Low | High | Falling edge | Rising edge | Both edges

The trigger condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values

with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the trigger condition discards place values with X or x and provides the output.

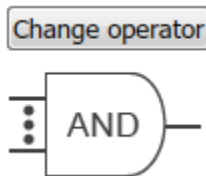
To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX` and a 32-bit hexadecimal value as `0xAB_CDEFX`.

For **boolean** signals, select a level or edge condition. For more information, see “Triggers” on page 5-6.

**Trigger combination operator** — Logical operator for creating trigger condition

AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The trigger condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. Suppose three signals, A, B, and C, make up the trigger condition. The options are:

`A == 10 AND B == 'Falling edge' AND C == 0`

or

`A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For more information, see “Triggers” on page 5-6.

**Trigger time out** — Maximum number of data capture IP core clock cycles within which trigger condition must occur in a trigger stage

1 (default) | integer from 1 to 65,536

Within this many data capture IP core clock cycles, the trigger condition must occur in a trigger stage in which you are enabling this parameter. You can specify any integer value from 1 to 65,536 according to your requirements. Select this parameter to enable trigger time out in a trigger stage. A trigger time out is not allowed in **Trigger Stage 1**.

**Capture Condition**

**Enable capture condition logic** — Option to enable capture condition logic

off (default) | on

Select this parameter to enable capture condition logic in the data capture IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 5-12.

**Dependencies**

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

**Signal** — Capture component signal name  
character vector

This parameter is read-only. The signal names you specified as triggers at generation time are listed in the drop-down menu at the bottom. Click the **+** button to add a signal to the capture condition.

**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Operator** — Operator to compare signals within capture condition  
== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Value** — Value to compare signal to as part of overall capture condition  
decimal | binary | hexadecimal | Low | High | Falling edge | Rising edge | Both edges

The capture condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the capture condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX` and a 32-bit hexadecimal value as `0xAB_CDEFX`.

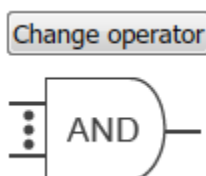
For `boolean` signals, select a level or edge condition. For more information, see “Capture Conditions” on page 5-12.

**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Capture condition combination operator** — Logical operator for creating capture condition  
AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The capture condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. You cannot mix and match the combination operators. For more information, see “Capture Conditions” on page 5-12.

### Dependencies

To enable this parameter, select **Enable capture condition logic**.

### Data Types

**Signal Name** — Captured signal name  
character vector

This parameter is read-only. It reflects the value you specified at generation time. This name is the name of the field in the structure variable.

**Bit Width** — Number of bits in the signal  
positive integer

This parameter is read-only. It reflects the value you specified at generation time.

**Data Type** — Data type for captured data  
built-in type | numeric type

The **Data Type** menu provides data type suggestions that match the bit width of the captured signal. This size is the width you specified for the port on the generated IP. You can type in this field to specify a custom data type. If the signal is 8, 16, or 32 bits, the default is `uint`. If the signal has one bit, the default is `boolean`. If the signal is a different width, the default is `numeric type(0, bitWidth, 0)`.

The tool supports these data types, depending on the signal bit width: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numeric type`.

## Version History

Introduced in R2017a

### See Also

#### Tools

FPGA Data Capture Component Generator

#### Objects

`hdlverifier.FPGADataReader`

#### Blocks

FPGA Data Reader

#### Topics

“Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

“Data Capture Workflow” on page 5-2

“Triggers” on page 5-6

“Capture Conditions” on page 5-12

## hdlverifier.FPGADataReader

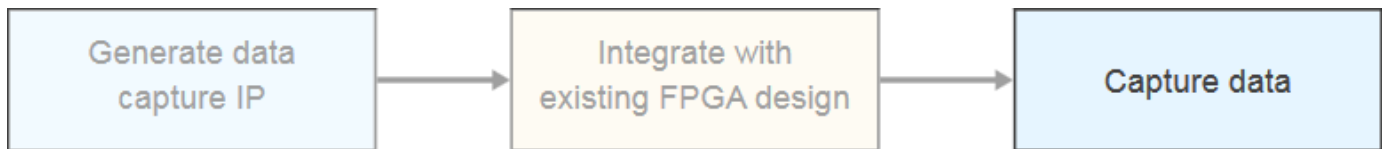
**Package:** hdlverifier

Capture data from live FPGA into MATLAB workspace

### Description

The `hdlverifier.FPGADataReader` System object communicates with a generated HDL IP core running on an FPGA board to capture signals from the FPGA into MATLAB.

The `hdlverifier.FPGADataReader` System object cannot be created directly. To use it, run **FPGA Data Capture Component Generator** and generate your own customized `FPGADataReader` System object. You can use the generated object directly or use the wrapper tool, **FPGA Data Capture**, to set trigger condition, capture condition, and data types, and capture data.



Before you create the System object, you must have previously generated the customized data capture components. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The object communicates with the FPGA over a JTAG cable. Make sure that the JTAG cable is connected between the board and the host computer.

For a workflow overview, see “Data Capture Workflow” on page 5-2.

---

**Note** Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

### Creation

`DC = mydc` creates a customized object, `DC`, that captures data from a design running on an FPGA. `mydc` is the component name you specified in the **FPGA Data Capture Component Generator** tool.

### Properties

#### **Timeout** — Number of seconds until aborting data capture

10 (default) | positive integer

If a trigger condition is enabled, but the HDL IP core does not detect the condition, the data capture request times out after the specified number of seconds. If the data capture is aborted, no data is returned to MATLAB.

When you use the tool for data capture, this property is ignored. Use the **Stop** button on the pop-up window to abort a capture using the tool.



**NumCaptureWindows — Number of data capture recurrences**

1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, and cannot be greater than *Sample depth*. When specifying the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the *Sample depth* divided by the *Number of capture windows*. Specify *Sample depth* in the **FPGA Data Capture Component Generator** tool.

For example: If *Sample depth* is 4096 and *Number of capture windows* is 4, then each capture window has a window depth of 1024.

**NumTriggerStages — Number of trigger stages for providing trigger conditions***M* (default) | integer from 1 to *M*

Specify the number of trigger stages. This value must be an integer from 1 to *M*, where *M* is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

For example, if **Max trigger stages** is 4, then NumTriggerStages can be 1, 2, 3, or 4.

**TriggerPosition — Position of the trigger detection cycle within the capture buffer**0 (default) | integer up to *window depth-1*

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to an integer from 0 to *window depth-1*, inclusive. When the trigger position is equal to *window depth-1*, the last sample corresponds to the cycle when the trigger occurs. For more information, see “Triggers” on page 5-6.

**EnableCaptureCtrl — Argument to enable capture condition logic**

false (default) | true

Set this property to true to enable capture condition logic in the HDL IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The HDL IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 5-12.

**Dependencies**

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

**CaptureMode — Capture mode**

'blocking' (default) | 'nonblocking'

Specify the capture mode as one of these options:

- 'blocking' — The data capture object blocks MATLAB while retrieving captured data. In this capture mode, the JTAG resource is allocated to either FPGA data capture or AXI manager at a time.

- 'nonblocking' — The data capture object does not block MATLAB while retrieving captured data. In this capture mode, you can use FPGA data capture and AXI manager simultaneously.

If your development board has multiple FPGAs or multiple JTAG connections, the data capture software cannot detect the location of your FPGA in the JTAG chain. Specify these advanced parameters to locate the FPGA that contains the data capture IP core.

### Advanced Board Setup

#### JTAGCableName — Name of JTAG cable used for data capture

'auto' (default) | character vector

Name of the JTAG cable used for data capture, specified as a character vector. Use this argument when the board is connected to two JTAG cables of the same type

### Object Functions

checkStatus	Check current status of FPGA data capture in nonblocking mode
clone	Create hdlverifier.FPGADataReader System object with same property values
collectData	Collect captured data from FPGA to host in nonblocking mode
displayCaptureCondition	Display overall capture condition
displayDataTypes	Display data types for all captured signals
displayTriggerCondition	Display overall trigger condition
isLocked	Locked status
launchApp	Open FPGA Data Capture app
release	Release control of JTAG interface
setCaptureCondition	Configure comparison for each signal value
setCaptureConditionCombinationOperator	Configure operator that combines individual signal value comparisons into overall capture condition
setCaptureConditionComparisonOperator	Configure operator that compares individual signal values within capture condition
setDataType	Configure data type for the data captured from a signal
setNumberOfTriggerStages	Configure number of trigger stages for capturing data
setRunImmediateFlag	Configure data capture to run immediately without any trigger condition
setTriggerCombinationOperator	Configure operator that combines individual signal value comparisons into overall trigger condition
setTriggerComparisonOperator	Configure operator that compares individual signal values within trigger condition
setTriggerCondition	Configure each signal value comparison
setTriggerTimeOut	Configure maximum number of FDC IP core clock cycles within which trigger condition must occur in a trigger stage
step	Capture one buffer of data from HDL IP core running on FPGA
stop	Stop FPGA data capture execution based on current status in nonblocking mode

### Examples

## Capture Data from FPGA over JTAG Connection

This example shows how to use the `hdlverifier.FPGADatReader` System object™ to capture data from a design running on an FPGA over a JTAG connection. The `hdlverifier.FPGADatReader` System object provides a programmatic way to configure and capture data. Generate an FPGA data reader System object by using the **FPGA Data Capture Component Generator** tool. Then use the generated System object directly to set data types and trigger conditions and capture data.

### Generate `hdlverifier.FPGADatReader` System Object

To generate a customized `hdlverifier.FPGADatReader` System object, open the **FPGA Data Capture Component Generator** tool by entering following command at the MATLAB® command prompt. To use this tool, you must have an existing HDL design and FPGA project.

```
generateFPGADatCaptureIP;
```

This example uses a generated object, `mydc`, that defines two signals for data capture. Signal A is 1 bit and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample depth is 4096 samples. To configure the `hdlverifier.FPGADatReader` System object to operate on these two signals, follow these steps.

- 1 Add one row to the **Ports** table by clicking the **Add** button once.
- 2 Name the first signal A and the second signal B.
- 3 Set **Bit Width** of the two signals to 1 and 8, respectively.
- 4 Specify **Generated IP name** as `mydc`.
- 5 Set **FPGA vendor** to Altera.
- 6 Set **Sample depth** to 4096.
- 7 Set **Max trigger stages** to 2.

This figure shows these tool settings.

**FPGA Data Capture Component Generation**

**Description**

Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

**Ports**

Port Name	Bit Width	Use As
A	1	Both triqquer and data
B	8	Both triqquer and data

**Target**

Generated IP name: mydc

FPGA vendor: Altera

Generated IP language: VHDL

Connection type: JTAG

Destination folder: hdlsrc

**Capture**

Sample depth: 4096

Max trigger stages: 2

Include capture condition logic

Generate Cancel Help

To generate the `hdlverifier.FPGADatReader` System object, click **Generate**. A report shows the results of the generation. Integrate the generated IP code into your existing FPGA project and deploy it to the FPGA. The System object communicates with the FPGA over a JTAG cable. Make sure that the JTAG cable connects the board and the host computer.

Go to the directory where the `hdlverifier.FPGADatReader` System object is generated.

```
cd hdlsrc;
```

Create a data capture object using your generated System object.

```
captureData = mydc
```

```
captureData =
```

```
    mydc with properties:
```

```
        Connection: 'JTAG'
        IsConditionalCapture: 0
```

```

    TriggerPosition: 0
    NumCaptureWindows: 1
    NumTriggerStages: 2
        Timeout: 10
    EnableCaptureCtrl: 0
        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
    MaxNumTriggerStages: 2

```

### Capture Data Immediately

Create a data capture object. The default trigger condition is to trigger immediately. The default configuration of the generated object does not enable any signals as part of the overall trigger condition.

```
captureData = mydc;
```

Display the data types of the captured signals. The default data type for an 8-bit signal is `uint8`.

```
displayDataTypes(captureData);
```

```

Signal Name : Data Type
Capture_Window : uint32
Trigger_Position : boolean
A : boolean
B : uint8

```

Call the object. The data is captured immediately from the FPGA.

```
[Capture_Window,Trigger_Position,dataOut] = captureData();
```

The captured data is returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal captured by the data capture object. The `dataOut` structure contains field `A`, which is a vector of 4096 logical values, and field `B`, which is a vector of 4096 `uint8` values.

### Capture Data on Trigger Event

To debug signal values near a specific event, set up a trigger condition. The trigger condition can be composed of value comparisons of one or more signals. You can combine these value comparisons with only one type of logical operator, either an `AND` or `OR` operator.

Define a trigger condition to capture data when the FPGA detects a high value on `A` at the same time as signal `B` is greater than 7.

```

captureData = mydc;
setTriggerCondition(captureData,'A',true,'High');
setTriggerCondition(captureData,'B',true,7);
setTriggerComparisonOperator(captureData,'B','>');

```

Display the overall trigger condition.

```
displayTriggerCondition(captureData);
```

```

The trigger condition is:
A==High and B>7

```

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

Define a trigger condition to capture data when the FPGA detects a high value on A at the same time as the value of signal B is 0xAX. In signal B, the trigger condition checks the leftmost 4 bits provided as A and ignores the rightmost 4 bits provided as X (X indicates bits for the function to ignore).

```
captureData = mydc;  
setTriggerCondition(captureData, 'A', true, 'High');  
setTriggerCondition(captureData, 'B', true, '0xAX');
```

Display the overall trigger condition.

```
displayTriggerCondition(captureData);
```

```
The trigger condition is:  
A==High and B==0xAX
```

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

`dataOut` is returned after the HDL IP core detects the trigger condition from the signals on the FPGA. `dataOut` contains samples starting from the cycle when the trigger condition is detected.

### Capture Data on Multiple Trigger Events

Define trigger conditions to capture data when the FPGA detects two trigger conditions in sequence.

- Trigger condition 1 - High value on A at the same time as signal B is equal to 7
- Trigger condition 2 - High value on A at the same time as signal B is greater than 15

```
captureData = mydc;  
setNumberOfTriggerStages(captureData,2);  
setTriggerCondition(captureData, 'A', true, 'High');  
setTriggerCondition(captureData, 'B', true,7);  
setTriggerCondition(captureData, 'A', true, 'High',2);  
setTriggerCondition(captureData, 'B', true,15,2);  
setTriggerComparisonOperator(captureData, 'B', '>',2);
```

Display the trigger condition. By default, the function displays the trigger condition in trigger stage 1.

```
displayTriggerCondition(captureData);
```

```
The trigger condition is:  
A==High and B==7
```

Display the trigger condition in trigger stage 2.

```
displayTriggerCondition(captureData,2);
```

```
The trigger condition is:  
A==High and B>15
```

Call the object to capture data on the specified trigger events.

```
[~,~,dataOut] = captureData();
```

`dataOut` is returned when the HDL IP core detects the trigger condition set in trigger stage 2 after detecting the trigger condition set in trigger stage 1, satisfying the set sequence.

## Capture Fixed-Point Data

The default data type for an 8-bit signal is `uint8`, but in your HDL design, you can represent the signal using a fixed-point number. Set the data type of the captured data to cast it to the fixed-point representation.

```
captureData = mydc;
setDataTypes(captureData, 'B', numerictype(1,8,6));
```

Display the data types of the captured signals.

```
displayDataTypes(captureData);
```

```
Signal Name : Data Type
Capture_Window : uint32
Trigger_Position : boolean
A : boolean
B : numerictype(1,8,6)
```

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

In the `dataOut` structure, field `A` is a vector of 4096 logical values and field `B` is a vector of 4096 signed 8-bit fixed-point values, with 6 fractional bits.

## Version History

Introduced in R2017a

## See Also

### Tools

FPGA Data Capture Component Generator | FPGA Data Capture

### Blocks

FPGA Data Reader

### Topics

“Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture” on page 7-2

“Debug IP Core Using FPGA Data Capture” (HDL Coder)

“Data Capture Workflow” on page 5-2

“Triggers” on page 5-6

“Capture Conditions” on page 5-12

## clone

Create `hdlverifier.FPGADataReader` System object with same property values

### Syntax

```
DC2 = clone(DC)
```

### Description

`DC2 = clone(DC)` creates a copy of the specified `hdlverifier.FPGADataReader` System object, with the same property values. If a System object is locked, then `clone` creates a copy that is also locked and has states initialized to the same values as the original. If a System object is not locked, then `clone` creates a new unlocked System object with uninitialized states.

### Input Arguments

#### **DC — Data capture System object**

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### Output Arguments

#### **DC2 — Data capture object**

`hdlverifier.FPGADataReader` System object

Data capture object, with the same property values as input DC, returned as an `hdlverifier.FPGADataReader` System object.

## Version History

Introduced in R2017a

### See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**



# displayDataTypes

Display data types for all captured signals

## Syntax

```
displayDataTypes(DC)
```

## Description

`displayDataTypes(DC)` displays the data type configured for each data capture signal. The default data type depends on the bit width of the captured signal in the specified data capture System object. This size is the width you specified for the port on the generated IP. If the signal is 8, 16, or 32 bits, the default data type is `uint`. If the signal has one bit, the default data type is `boolean`. If the signal is a different width, the default data type is `numerictype(0,bitWidth,0)`.

To modify the data type of a signal, use the `setDataType` object function. The function supports these data types, depending on the bit width of the captured signal: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numerictype`.

## Input Arguments

### DC — Data capture System object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

## Version History

Introduced in R2017a

## See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

# displayTriggerCondition

Display overall trigger condition

## Syntax

```
displayTriggerCondition(DC)  
displayTriggerCondition(DC,N)
```

## Description

`displayTriggerCondition(DC)` displays the signal value comparisons and logical operator that define the overall trigger condition in trigger stage 1. DC is a customized data capture object.

`displayTriggerCondition(DC,N)` displays the signal value comparisons and logical operator that define the overall trigger condition in a trigger stage specified by N. DC is a customized data capture object.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### N — Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. Use N to display the trigger condition in Nth trigger stage. If you do not specify N, by default, the function displays the trigger condition in trigger stage 1.

## Version History

Introduced in R2017a

## See Also

### Objects

`hdlverifier.FPGADataReader`

### Tools

**FPGA Data Capture Component Generator** | **FPGA Data Capture**

# isLocked

Locked status

## Syntax

```
tf = isLocked(DC)
```

## Description

`tf = isLocked(DC)` returns the locked status, of the specified `hdlverifier.FPGADataReader System` object.

`isLocked` returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you call the object. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, `isLocked` returns a `true` value.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

## Output Arguments

### tf — True or false result

1 | 0

True or false result indicating the locked status of the input System object DC, returned as a 1 or 0 of data type `logical`. A 1 indicates the System object is locked.

## Version History

Introduced in R2017a

## See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

# launchApp

Open FPGA Data Capture app

## Syntax

```
launchApp(DC)
```

## Description

launchApp(DC) opens the **FPGA Data Capture** app, which captures data from a design running on an FPGA and returns the captured data to the MATLAB workspace. The app is a wrapper on the specified `hdlverifier.FPGADataReader` System object. Changes that you make in the app are saved in the properties of the System object.

You can configure the data types of the returned values and set up a trigger condition to control when the data is captured. You must have previously generated the customized data capture components by using **FPGA Data Capture Component Generator**. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The tool communicates with the FPGA over a JTAG cable. You must connect the JTAG cable between the board and the host computer.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

## Version History

Introduced in R2017a

## See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

# release

Release control of JTAG interface

## Syntax

```
release(DC)
```

## Description

release(DC) releases system resources, including control of the JTAG interface, of the specified `hdlverifier.FPGADataReader System` object. Releasing the System object enables you to change its properties and input characteristics. While the System object exists and is locked, no other processes can use the JTAG cable.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

## Version History

Introduced in R2017a

## See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

## setDataType

Configure data type for the data captured from a signal

### Syntax

```
setDataType(DC, name, type)
```

### Description

`setDataType(DC, name, type)` specifies the data type, `type`, for the data captured from a signal, `name`.

### Input Arguments

#### **DC — Customized data capture object**

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

#### **name — Trigger component signal**

character vector

Specify a signal name matching one that you configured when you generated the object. The signal must be configured as a data signal.

#### **type — Data type for the captured data**

built-in data type | `numerictype`

The bit width of the data type must match the bit width of the captured signal. This size is the width you specified for the port on the generated IP.

The function supports these data types, depending on the bit width of the captured signal: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numerictype`.

## Version History

**Introduced in R2017a**

### See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

# setTriggerComparisonOperator

Configure operator that compares individual signal values within trigger condition

## Syntax

```
setTriggerComparisonOperator(DC,name,operator)
setTriggerComparisonOperator(DC,name,operator,N)
```

## Description

`setTriggerComparisonOperator(DC,name,operator)` configures a comparison operator that compares individual signal values within the trigger condition in trigger stage 1. `DC` is a customized data capture object, `name` is the name of a trigger component signal.

`setTriggerComparisonOperator(DC,name,operator,N)` configures a comparison operator that compares individual signal values within the trigger condition in a trigger stage specified by `N`. `DC` is a customized data capture object, `name` is the name of a trigger component signal.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### name — Name of trigger component signal

character vector

Name of a trigger component signal, specified as a character vector. This name must match one of the signal names configured on creation of the input System object `DC`. The signal must be configured as a possible trigger signal.

### operator — Operator to compare signals within trigger condition

`==` (default) | `!=` | `<` | `>` | `<=` | `>=`

Operator to compare signals within the trigger condition, specified as one of these operators: `==` (default), `!=`, `<`, `>`, `<=`, or `>=`.

The trigger condition comprises value comparisons of one or more signals. For a multibit signal, specify one of these operators: `==` (default), `!=`, `<`, `>`, `<=`, or `>=`. For a trigger condition containing `X` or `x` (don't-care value), specify either `==` or `!=` operators. For a logical signal, specify one of these operators: `==` or `!=`. For details on trigger conditions, see "Triggers" on page 5-6.

### N — Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 6-0 `parameter` of the **FPGA Data Capture Component Generator** tool. Use *N* to set the trigger comparison operator in *N*th trigger stage. If you do not specify *N*, by default, the function sets the trigger comparison operator in trigger stage 1.

## **Version History**

**Introduced in R2019b**

### **See Also**

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**



# setTriggerCombinationOperator

Configure operator that combines individual signal value comparisons into overall trigger condition

## Syntax

```
setTriggerCombinationOperator(DC,operator)
setTriggerCombinationOperator(DC,operator,N)
```

## Description

`setTriggerCombinationOperator(DC,operator)` configures the logical operator that combines comparisons of individual signals into an overall trigger condition in trigger stage 1. DC is a customized data capture object.

`setTriggerCombinationOperator(DC,operator,N)` configures the logical operator that combines comparisons of individual signals into an overall trigger condition in a trigger stage specified by N. DC is a customized data capture object.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### operator — Logical operator to combine comparisons of individual signals into trigger condition

AND | OR

Logical operator to combine comparisons of individual signals into a trigger condition, specified as AND or OR. The trigger condition comprises value comparisons of one or more signals. To combine value comparisons, you can use only one type of logical operator. For example, suppose three signals, A, B, and C, make up the trigger condition. The options are:

- `A == 10 AND B == 'Falling edge' AND C == 0`
- `A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For details on trigger conditions, see “Triggers” on page 5-6.

### N — Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 6-0 parameter of the **FPGA Data Capture Component Generator** tool. Use N to set the combination operator in Nth trigger stage. If you do not specify N, by default, the function sets the combination operator in trigger stage 1.

## **Version History**

**Introduced in R2017a**

### **See Also**

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

# setTriggerCondition

Configure each signal value comparison

## Syntax

```
setTriggerCondition(DC,name,enable,value)
setTriggerCondition(DC,name,enable,value,N)
```

## Description

`setTriggerCondition(DC,name,enable,value)` configures a trigger value comparison for signal name in trigger stage 1. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall trigger condition.

`setTriggerCondition(DC,name,enable,value,N)` configures a trigger value comparison for signal name in a trigger stage specified by N. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall trigger condition.

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

### name — Name of trigger component signal

character vector

Name of trigger component signal, specified as a character vector.

This name must match one of the signal names configured on creation of the input `System` object DC. The signal must be configured as a possible trigger signal.

Data Types: `char`

### enable — Indication that signal is part of trigger condition

`true` or `1` | `false` or `0`

Indication that the signal is part of the trigger condition, specified as a numeric or logical `1` (`true`) or `0` (`false`). To use this signal in the overall trigger condition, set this value to `1` (`true`). When you set this value to `0` (`false`), the signal is not used for the overall trigger condition.

### value — Value to compare this signal to as part of the trigger condition

decimal | binary | hexadecimal | 'Low' | 'High' | 'Rising edge' | 'Falling edge' | 'Both edges'

The trigger condition comprises value comparisons of one or more signals. This input specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values

with a combination of X or x (don't care value) to enable bit masking. That means, while comparing the values, the trigger condition ignores the place values with X or x and provides the output.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `'0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX'` and a 32-bit hexadecimal value as `'0xAB_CDEFX'`.

For logical signals, specify a string that indicates the level or edge to match. For more information, see “Triggers” on page 5-6.

**N – Trigger stage**

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 6-0 `parameter` of the **FPGA Data Capture Component Generator** tool. Use *N* to set the trigger condition in *N*th trigger stage. If you do not specify *N*, by default, the function sets the trigger condition in trigger stage 1.

## Version History

Introduced in R2017a

**See Also**

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

## step

Capture one buffer of data from HDL IP core running on FPGA

### Syntax

```
dataOut = step(DC)
```

### Description

---

**Note** Alternatively, instead of using the `step` object function to perform the operation defined by the System object, you can call the System object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`dataOut = step(DC)` captures live signal data from a design running on an FPGA. The FPGA must contain an HDL IP core generated from the **FPGA Data Capture Component Generator** tool. `dataOut` is a structure that contains a field for each signal captured. Call the `setDataTypes` object function to specify the data type of each captured signal.

If at least one signal is enabled as part of the trigger condition, the HDL IP core waits for a match of the trigger condition and captures the data. If no signals are enabled as part of the trigger condition, the HDL IP core captures and returns the buffered data immediately. When you create the object, no trigger condition is set by default. Call the `setTriggerCondition` and `setTriggerCombinationOperator` object functions to configure a trigger condition.

### Input Arguments

#### DC — Customized data capture object

FPGADataReader System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### Output Arguments

#### dataOut — Captured data

structure

Captured data, returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal captured by FPGA data capture. The captured signal field is a vector of *Sample depth* values for each signal requested for data capture at generation time. The fields of the structure have these signal names.

- `Capture_Window` — This signal indicates the current capture window.
- `Trigger_Position` — This signal indicates the position of the trigger detection clock cycle within a capture buffer.
- All remaining fields — The signal names you specified at generation time.

## **Version History**

**Introduced in R2017a**

### **See Also**

hdlverifier.FPGADatReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

### **Topics**

"Triggers" on page 5-6

# setNumberOfTriggerStages

Configure number of trigger stages for capturing data

## Syntax

```
setNumberOfTriggerStages(DC,N)
```

## Description

setNumberOfTriggerStages(DC,N) specifies an integer value configures the number of trigger stages, N, for capturing data. DC is customized data capture object.

## Input Arguments

### DC — Customized data capture object

hdlverifier.FPGADataReader System object

Customized data capture object, specified as an hdlverifier.FPGADataReader System object.

### N — Total number of trigger stages

integer from 1 to  $M$

Total number of trigger stages in which you want to capture the data, specified as an integer from 1 to  $M$ , where  $M$  is set by the Max trigger stages on page 6-0 parameter of the **FPGA Data Capture Component Generator** tool.

## Version History

Introduced in R2020b

## See Also

hdlverifier.FPGADataReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

## setTriggerTimeOut

Configure maximum number of FDC IP core clock cycles within which trigger condition must occur in a trigger stage

### Syntax

```
setTriggerTimeOut(DC,enable,value,N)
```

### Description

`setTriggerTimeOut(DC,enable,value,N)` configures the maximum number of FPGA Data Capture (FDC) IP core clock cycles, within which the trigger condition must occur in a trigger stage specified by `N`. `DC` is a customized data capture object. Use `enable` argument to enable the trigger time out in trigger stage `N`, specify the number of FDC IP core clock cycles using `value` argument.

### Input Arguments

#### **DC — Customized data capture object**

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

#### **enable — Indication that trigger time out is part of specified trigger stage**

`true` or `1` | `false` or `0`

Indication that the trigger time out is part of the trigger stage, specified as a numeric or logical `1` (`true`) or `0` (`false`). To use the trigger time out in a particular trigger stage, set this value to `1` (`true`). When you set this value to `0` (`false`), the trigger time out is not used for the specified trigger stage.

#### **value — Number of FDC IP core clock cycles**

integer from 1 to 65,536

Specify an integer from 1 to 65,536. Within this many FDC IP core clock cycles, the trigger condition must occur in a trigger stage specified by `N`.

#### **N — Trigger stage**

integer from 2 to  $M$

Trigger stage, specified as an integer from 2 to  $M$ , where  $M$  is set by the Max trigger stages on page 6-0 parameter of the **FPGA Data Capture Component Generator** tool. Use `N` to set the trigger time out in  $N$ th trigger stage. Trigger time out is not allowed for trigger stage 1.

## Version History

Introduced in R2020b



## **See Also**

hdlverifier.FPGADataReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

## setRunImmediateFlag

Configure data capture to run immediately without any trigger condition

### Syntax

```
setRunImmediateFlag(DC,value)
```

### Description

`setRunImmediateFlag(DC,value)` specifies whether the data capture object `DC` runs in immediate mode. If `value` is `true`, the data capture object runs in immediate mode and captures data immediately without checking for the trigger condition. In this mode, the data capture object captures data either at each clock cycle or based on a capture condition if capture condition logic is enabled. If `value` is `false`, the data capture object does not run in immediate mode.

### Input Arguments

#### **DC — Customized data capture object**

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

#### **value — Flag to capture data immediately**

`true` or `1` | `false` or `0`

Flag to capture data immediately, specified as a numeric or logical `1` (`true`) or `0` (`false`). To ignore the trigger condition and capture data immediately, set this value to `1` (`true`). In this case, the generated object does not enable any signals as part of the overall trigger condition. To capture data that includes a particular event in the FPGA logic, configure a trigger condition and set this value to `0` (`false`). In this case, the data capture object waits until the trigger condition is true, then captures the data.

## Version History

Introduced in R2022a

### See Also

#### **Objects**

`hdlverifier.FPGADataReader`

#### **Tools**

**FPGA Data Capture Component Generator | FPGA Data Capture**

# displayCaptureCondition

Display overall capture condition

## Syntax

```
displayCaptureCondition(DC)
```

## Description

`displayCaptureCondition(DC)` displays the signal value comparisons and logical operator that define the overall capture condition. DC is a customized data capture object.

## Examples

### Display Capture Condition

This example uses a customized data capture object, DC, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is greater than 7.

```
setCaptureCondition(DC, 'A', true, 'High');
setCaptureCondition(DC, 'B', true, 7);
setCaptureConditionComparisonOperator(DC, 'B', '>');
```

Combine comparisons of signals A and B into an overall capture condition using an AND operator.

```
setCaptureConditionCombinationOperator(DC, 'AND');
```

Display the overall capture condition.

```
displayCaptureCondition(DC);
```

```
The capture condition is:
A==High and B>7
```

## Input Arguments

### DC — Customized data capture object

hdlverifier.FPGADataReader System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

## **Version History**

**Introduced in R2022a**

### **See Also**

#### **Objects**

`hdlverifier.FPGADataReader`

#### **Tools**

**FPGA Data Capture Component Generator | FPGA Data Capture**

# setCaptureCondition

Configure comparison for each signal value

## Syntax

```
setCaptureCondition(DC, name, enable, value)
```

## Description

`setCaptureCondition(DC, name, enable, value)` configures a capture value comparison for the signal name. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall capture condition.

## Examples

### Set Up Capture Condition

This example uses a customized data capture object, DC, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A and a value 17 on signal B.

```
setCaptureCondition(DC, 'A', true, 'High');
setCaptureCondition(DC, 'B', true, uint8(17));
```

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

### name — Name of capture component signal

character vector

Name of the capture component signal, specified as a character vector.

This name must match one of the signal names configured on creation of the input System object DC. The signal must be configured as a possible trigger signal.

Data Types: `char`

**enable — Indication that signal is part of capture condition**`true | false`

Indication that the signal is part of the capture condition, specified as `true` or `false`. To use this signal in the overall capture condition, set this value to `true`. When you set this value to `false`, the signal is not used for the overall capture condition.

**value — Value to compare signal to as part of capture condition**`decimal | binary | hexadecimal | 'Low' | 'High' | 'Rising edge' | 'Falling edge' | 'Both edges'`

Value to compare the signal to as part of the capture condition, specified as one of the following.

- Decimal, binary, or hexadecimal value — For a multibit signal, specify a value within the range of the data type associated with the signal. If you specify a binary or hexadecimal value, you can use an `X` or `x` to indicate signals for the function to ignore during the value comparison.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `'0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX'` and a 32-bit hexadecimal value as `'0xAB_CDEFX'`.

- `'Low'`, `'High'`, `'Rising edge'`, `'Falling edge'`, or `'Both edges'` — For a logical signal, specify a string that indicates the level or edge to match. For more information, see “Capture Conditions” on page 5-12.

## Version History

Introduced in R2022a

### See Also

**Objects**`hdlverifier.FPGADataReader`**Tools****FPGA Data Capture Component Generator | FPGA Data Capture**

# setCaptureConditionCombinationOperator

Configure operator that combines individual signal value comparisons into overall capture condition

## Syntax

```
setCaptureConditionCombinationOperator(DC, operator)
```

## Description

`setCaptureConditionCombinationOperator(DC, operator)` configures the logical operator `operator` that combines comparisons of individual signals into an overall capture condition. `DC` is a customized data capture object.

## Examples

### Combine Comparisons of Individual Signals into Overall Capture Condition

This example uses a customized data capture object, `DC`, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is equal to 17.

```
setCaptureCondition(DC, 'A', true, 'High');
setCaptureCondition(DC, 'B', true, uint8(17));
```

Combine comparisons of signals A and B into an overall capture condition using an AND operator.

```
setCaptureConditionCombinationOperator(DC, 'AND');
```

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### operator — Logical operator to combine comparisons of individual signals into capture condition

'AND' (default) | 'OR'

Logical operator to combine comparisons of individual signals into a capture condition, specified as 'AND' or 'OR'. The capture condition comprises value comparisons of one or more signals. To combine value comparisons, you can use only one type of logical operator. For example, suppose three signals, A, B, and C, make up the capture condition. The options are:

- `A == 10 AND B == 'Falling edge' AND C == 0`
- `A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For details on capture conditions, see “Capture Conditions” on page 5-12.

## **Version History**

**Introduced in R2022a**

### **See Also**

#### **Objects**

`hdlverifier.FPGADatReader`

#### **Tools**

**FPGA Data Capture Component Generator | FPGA Data Capture**



# setCaptureConditionComparisonOperator

Configure operator that compares individual signal values within capture condition

## Syntax

```
setCaptureConditionComparisonOperator(DC,name,operator)
```

## Description

`setCaptureConditionComparisonOperator(DC,name,operator)` configures a comparison operator `operator` that compares individual signal values within the capture condition. `DC` is a customized data capture object. `name` is the name of a capture component signal.

## Examples

### Set Up Capture Condition Using Comparison Operator

This example uses a customized data capture object, `DC`, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is greater than 7.

```
setCaptureCondition(DC,'A',true,'High');
setCaptureCondition(DC,'B',true,7);
setCaptureConditionComparisonOperator(DC,'B','>');
```

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

### name — Name of capture component signal

character vector

Name of a capture component signal, specified as a character vector. This name must match one of the signal names configured on creation of the input System object `DC`. The signal must be configured as a possible trigger signal.

**operator — Operator to compare signals within capture condition**`== (default) | != | < | > | <= | >=`

Operator to compare signals within the capture condition, specified as one of these operators: ==, !=, <, >, <=, or >=.

The capture condition comprises value comparisons of one or more signals. For a multibit signal, specify one of these operators: == (default), !=, <, >, <=, or >=. For a capture condition containing X or x (which indicate bits for the function to ignore), specify either the == or != operators. For a logical signal, specify either the == or != operators. For details on capture conditions, see “Capture Conditions” on page 5-12.

## Version History

Introduced in R2022a

### See Also

**Objects**

hdlverifier.FPGADatReader

**Tools**

FPGA Data Capture Component Generator | FPGA Data Capture

# checkStatus

Check current status of FPGA data capture in nonblocking mode

## Syntax

```
status = checkStatus(DC)
```

## Description

`status = checkStatus(DC)` returns the current status of the data capture object DC in nonblocking capture mode.

---

**Note** The `checkStatus` function is not supported in blocking capture mode.

---

## Examples

### Check Current Status of Data Capture Object

Before you use this example, you must have previously generated the customized data capture object using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

Create a data capture object, DC, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
datacapture with properties:
```

```

    Connection: 'JTAG'
IsConditionalCapture: 0
    TriggerPosition: 0
    NumCaptureWindows: 1
    NumTriggerStages: 1
        Timeout: 10
    EnableCaptureCtrl: 0
        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
    MaxNumTriggerStages: 1
```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)
status =
  struct with fields:
    CapturedWindows: 0
    RunStatus: 'Not started'
    TriggerStage: 0
```

Use the `step` function to capture data. The data is captured immediately from the FPGA.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)
status =
  struct with fields:
    CapturedWindows: 1
    RunStatus: 'Successfully captured data from FPGA'
    TriggerStage: 0
```

## Input Arguments

### **DC** — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

## Output Arguments

### **status** — Current status of data capture object

structure

Current status of the data capture object, returned as a structure containing fields for these signals.

- `CapturedWindows` — This signal indicates the total number of windows captured so far.
- `RunStatus` — This signal indicates the current running status of the data capture object using one of these options.
  - `'Not started'` — Data capture not started.
  - `'Waiting for trigger'` — Data capture object is waiting for a trigger event to start data capture.
  - `'Evaluating capture condition'` — Data capture object is evaluating the capture condition.
  - `'Successfully captured data from FPGA'` — Data captured successfully from the FPGA.
  - `'Stopped'` — Data capture has stopped.
- `TriggerStage` — This signal indicates the trigger stage under evaluation for the run status `'Waiting for trigger'`. For the run statuses `'Evaluating capture condition'`,

'Successfully captured data from FPGA', and 'Stopped', this signal indicates the updated value of the trigger stage.

## **Version History**

**Introduced in R2022a**

## **See Also**

### **Objects**

hdlverifier.FPGADatReader

### **Tools**

**FPGA Data Capture Component Generator | FPGA Data Capture**

## collectData

Collect captured data from FPGA to host in nonblocking mode

### Syntax

```
capturedData = collectData(DC)
```

### Description

`capturedData = collectData(DC)` returns the captured data from an FPGA to the host machine in nonblocking capture mode. DC is a customized data capture object.

---

**Note** The `collectData` function is not supported in blocking capture mode.

---

### Examples

#### Collect Captured Data from FPGA

Before you use this example, you must have previously generated the customized data capture object, using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

This example uses a generated object, `datacapture`, that defines two signals for data capture. Signal A is 16 bits and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample depth is 1024 samples.

Create a data capture object, DC, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
    datacapture with properties:
```

```
        Connection: 'JTAG'
    IsConditionalCapture: 0
        TriggerPosition: 0
        NumCaptureWindows: 1
        NumTriggerStages: 1
            Timeout: 10
        EnableCaptureCtrl: 0
            CaptureMode: 'blocking'
            JTAGCableName: 'auto'
    MaxNumTriggerStages: 1
```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:
        CapturedWindows: 0
        RunStatus: 'Not started'
        TriggerStage: 0
```

Define a trigger condition to capture data when the signal B is equal to 10.

```
setTriggerCondition(DC, 'B', true, 10);
```

Use the `step` function to capture data on the specified trigger event.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:
        CapturedWindows: 1
        RunStatus: 'Successfully captured data from FPGA'
        TriggerStage: 0
```

Collect captured data.

```
capturedData = collectData(DC)
```

Captured 1 windows of data from FPGA.

```
capturedData =

    struct with fields:
        Capture_Window: [1024x1 uint32]
        Trigger_Position: [1024x1 logical]
        A: [1024x1 uint16]
        B: [1024x1 uint8]
```

## Input Arguments

### DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

## Output Arguments

### **capturedData — Captured data**

structure

Captured data, returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal obtained by FPGA data capture. The captured signal field is a vector of *Sample depth* values for each signal requested for data capture at generation time. The fields of the structure have these signal names.

- `Capture_Window` — This signal indicates the current capture window.
- `Trigger_Position` — This signal indicates the position of the trigger detection clock cycle within a capture buffer.
- All remaining fields — The signal names you specified at generation time.

## Version History

Introduced in R2022a

### See Also

#### Objects

`hdlverifier.FPGADataReader`

#### Tools

**FPGA Data Capture Component Generator | FPGA Data Capture**



## stop

Stop FPGA data capture execution based on current status in nonblocking mode

### Syntax

```
stop(DC)
```

### Description

`stop(DC)` stops the execution of the data capture object `DC` based on the current status in nonblocking capture mode. Stop execution of the specified data capture object when the current status of the object is 'Waiting for trigger'. If you want to cancel the capture attempt (for example, if the trigger condition does not occur), use this function to return control to the object. When you cancel a capture attempt, no data is returned to MATLAB.

---

**Note** The `stop` function is not supported in blocking capture mode.

---

### Examples

#### Stop FPGA Data Capture Execution

Before you use this example, you must have previously generated the customized data capture object using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

This example uses a generated object, `datacapture`, that defines two signals for data capture. Signal A is 16 bits and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample depth is 1024 samples.

Create a data capture object, `DC`, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
datacapture with properties:
```

```

        Connection: 'JTAG'
    IsConditionalCapture: 0
        TriggerPosition: 0
    NumCaptureWindows: 1
    NumTriggerStages: 1
            Timeout: 10
    EnableCaptureCtrl: 0
```

```
        CaptureMode: 'blocking'  
        JTAGCableName: 'auto'  
        MaxNumTriggerStages: 1
```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)
```

```
status =
```

```
    struct with fields:
```

```
        CapturedWindows: 0  
        RunStatus: 'Not started'  
        TriggerStage: 0
```

Define a trigger condition to capture data when the signal B is equal to 255.

```
setTriggerCondition(DC, 'B', true, 255);
```

Use the `step` function to capture data on the specified trigger event.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)
```

```
status =
```

```
    struct with fields:
```

```
        CapturedWindows: 0  
        RunStatus: 'Waiting for trigger'  
        TriggerStage: 1
```

Stop data capture.

```
stop(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)
```

```
status =
```

```
    struct with fields:
```

```
        CapturedWindows: 0
```

```
RunStatus: 'Stopped'  
TriggerStage: 1
```

## Input Arguments

### DC — Customized data capture object

hdlverifier.FPGADataReader System object

Customized data capture object, specified as an hdlverifier.FPGADataReader System object.

## Version History

Introduced in R2022a

## See Also

### Objects

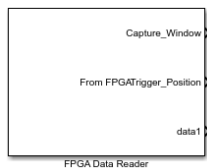
hdlverifier.FPGADataReader

### Tools

FPGA Data Capture Component Generator | FPGA Data Capture

## FPGA Data Reader

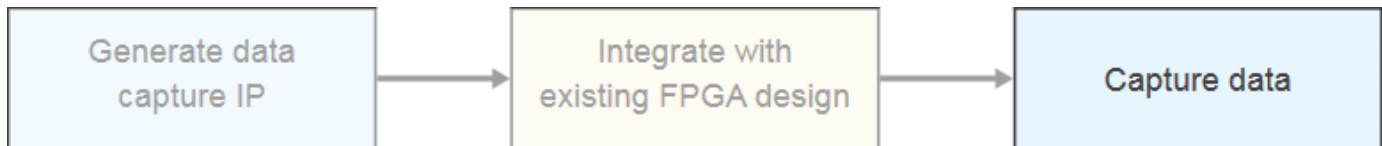
Capture data from live FPGA into Simulink model



**Libraries:**  
Generated

### Description

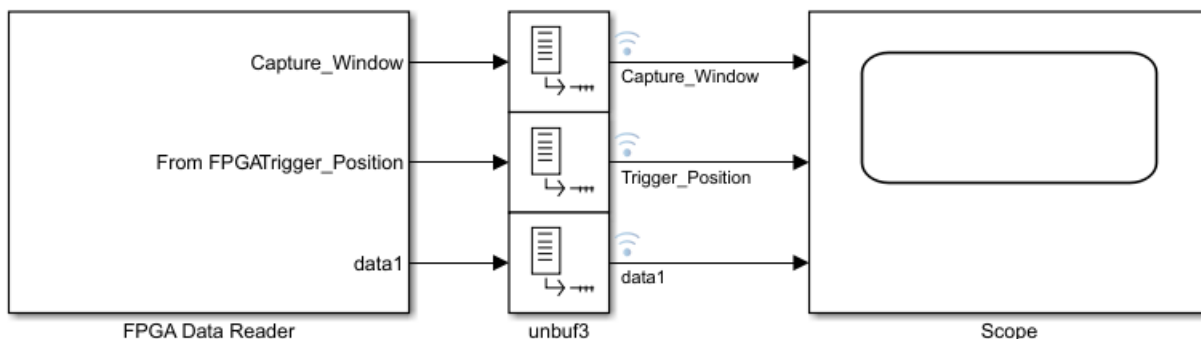
The FPGA Data Reader block communicates with a generated IP core on an FPGA to return captured data into Simulink.



Before you run this block, you must generate the customized data capture components. Integrate the generated HDL IP core into your project and deploy it to the FPGA. The block communicates with the FPGA over a JTAG cable. Make sure that the JTAG cable is connected between the board and the host computer.

For a workflow overview, see “Data Capture Workflow” on page 5-2.

By default, the **FPGA Data Capture Component Generator** tool generates a data capture model that contains this block and a scope. If you have a DSP System Toolbox license, the captured data is streamed to the **Logic Analyzer** tool. Otherwise, the Scope block shows the captured data. You can add other blocks to the model for analysis, verification, and display.



## Ports

The output ports of the FPGA Data Reader block correspond to the signals you requested to capture in **FPGA Data Capture Component Generator**. Set the data types for these ports in the **Signal and Trigger Editor**, opened from the block parameters.

### Output

**Capture\_Window** — Current capture window  
scalar

This output port indicates the current capture window. The value of this output port is an integer from 1 to the value of the **Sample depth** parameter.

**Trigger\_Position** — Position of trigger detection clock cycle within capture buffer  
Boolean scalar

This output port indicates the position of the trigger detection clock cycle within a capture buffer.

## Parameters

**Sample time** — Rate of output signals  
double

The block returns one frame of data per time step, where the frame is the entire capture buffer for each signal. Each frame contains **Sample depth** values, as specified at generation time. The default sample time provides for unbuffering each frame into single samples, which results in a sample time of 1.

### Trigger

**Sample depth** — Number of samples captured for each signal  
integer

This parameter is read-only. It reflects the value you specified at generation time.

**Number of capture windows** — Number of data capture recurrences  
1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, up to **Sample depth**. A *window depth* is defined as **Sample depth / Number of capture windows**. Consider the **Number of capture windows** when setting the **Sample depth**, to allow for sufficient buffering.

**Number of trigger stages** — Number of trigger stages for providing trigger conditions  
 $M$  (default) | integer from 1 to  $M$

Specify the number of trigger stages. This value must be an integer from 1 to  $M$ , where  $M$  is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

**Trigger position** — Position of trigger detection cycle within capture buffer  
0 (default) | integer up to *window depth*-1

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to any number between 0 and *window depth*-1, inclusive. When the trigger position is equal to the *window depth*-1, the last sample corresponds to the cycle when the trigger occurs. If **Number of capture windows** is greater than one, the same trigger position applies to all windows. See “Triggers” on page 5-6.

**Signal** — Trigger component signal name  
character vector

This parameter is read-only. The signal names you specified at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the trigger condition.

**Operator** — Operator to compare signals within trigger condition  
== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

**Value** — Value to compare signal to as part of overall trigger condition  
decimal | binary | hexadecimal | Low | High | Rising edge | Falling edge | Both edges

The trigger condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

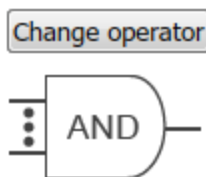
For a multi-bit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the trigger condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use \_ between bits. For example, you can represent a 32-bit binary value as 0b1010\_XXXX\_1011\_XXXX\_1110\_XXXX\_1111XXXX and a 32-bit hexadecimal value as 0xAB\_CDEFX.

For boolean signals, select a level or edge condition. See “Triggers” on page 5-6.

**Trigger combination operator** — Logical operator to combine comparisons of individual signals into overall trigger condition  
AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The trigger condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. Suppose three signals, A, B, and C, make up the trigger condition. The options are:

```
A == 10 AND B == 'Falling edge' AND C == 0
```

or

```
A == 10 OR B == 'Falling edge' OR C == 0
```

You cannot mix and match the combination operators. See “Triggers” on page 5-6.

**Trigger time out** — Maximum number of data capture IP core clock cycles within which trigger condition must occur in a trigger stage

1 (default) | integer from 1 to 65,536

Within this many data capture IP core clock cycles, the trigger condition must occur in a trigger stage in which you are enabling this parameter. You can specify any integer value from 1 to 65,536 according to your requirements. Select this parameter to enable trigger time out in a trigger stage. A trigger time out is not allowed in **Trigger Stage 1**.

**Time out** — Number of seconds to wait before aborting data capture, if the trigger condition is not met

10 (default) | positive integer

If a trigger condition is enabled, but the HDL IP core does not detect the condition, the data capture request times out after this many seconds. No data is returned to Simulink.

### Capture Condition

**Enable capture condition logic** — Option to enable capture condition logic

off (default) | on

Select this parameter to enable capture condition logic in the data capture IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 5-12.

### Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

**Signal** — Capture component signal name

character vector

This parameter is read-only. The signal names you specified as triggers at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the capture condition.

### Dependencies

To enable this parameter, select **Enable capture condition logic**.

**Operator** — Operator to compare signals within capture condition

== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Value** — Value to compare signal to as part of overall capture condition

decimal | binary | hexadecimal | Low | High | Rising edge | Falling edge | Both edges

The capture condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multi-bit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the capture condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX` and a 32-bit hexadecimal value as `0xAB_CDEFX`.

For boolean signals, select a level or edge condition. See “Capture Conditions” on page 5-12.

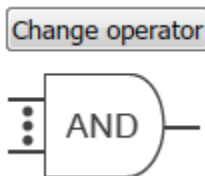
**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Capture condition combination operator** — Logical operator to combine comparisons of individual signals into overall capture condition

AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The capture condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. You cannot mix and match the combination operators. See “Capture Conditions” on page 5-12.

**Dependencies**

To enable this parameter, select **Enable capture condition logic**.

**Data Types**

**Signal Name** — Name of output port

character vector

This parameter is read-only. It reflects the name of the **Capture\_Window** output port, the name of the **Trigger\_Position** output port, and the signal names you specify at generation time.



**Bit Width** — Number of bits in signal  
positive integer

This parameter is read-only. It reflects the value you specified at generation time.

**Data Type** — Data type for captured data  
built-in type | numerictype

The **Data Type** menu provides data type suggestions that match the bit width of the captured signal. This size is the width you specified for the port on the generated IP. You can type in this field to specify a custom data type. If the signal is 8, 16, or 32 bits, the default is `uint`. If the signal has one bit, the default is `boolean`. If the signal is a different width, the default is `numerictype(0,bitWidth,0)`.

The block supports these data types, depending on the signal bit width: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numerictype`.

If your development board has multiple FPGAs or multiple JTAG connections, the data capture software cannot detect the location of your FPGA in the JTAG chain. Specify these advanced parameters to locate the FPGA that contains the data capture IP core.

#### Advanced Board Setup

**JTAG cable name** — Name of JTAG cable used for data capture  
auto (default) | character vector

Name of the JTAG cable used for data capture, specified as a character vector. Use this parameter when the board is connected to two JTAG cables of the same type.

## Version History

Introduced in R2017a

### See Also

**Apps**  
FPGA Data Capture

**Objects**  
`hdlverifier.FPGADatReader`

**Topics**  
"Data Capture Workflow" on page 5-2  
"Triggers" on page 5-6  
"Capture Conditions" on page 5-12

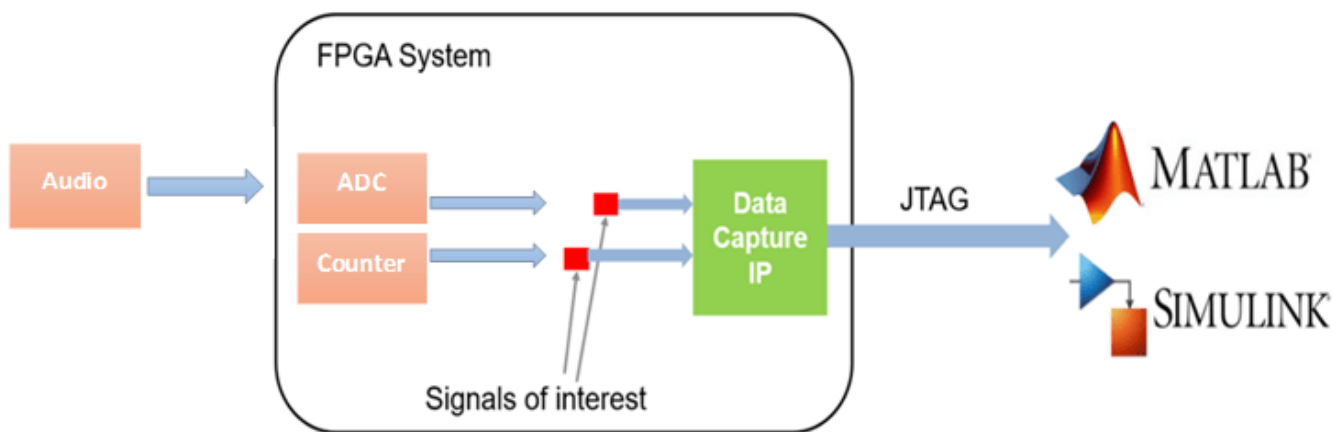


# HDL Verifier Support Package for Intel FPGA Boards Examples

---

## Capture Audio Signal from Intel FPGA Board Using FPGA Data Capture

This example shows how to use FPGA data capture with existing HDL code to read FPGA internal signals. Start with an existing FPGA design that implements an on-chip analog-to-digital converter (ADC) to sample an audio signal. The ADC IP exposes an Avalon® memory-mapped (MM) slave interface for control and an Avalon streaming interface for data output. This FPGA design already contains an Avalon MM master to start the ADC. Use the FPGA data capture feature to collect the ADC output data from the Avalon streaming interface into MATLAB® workspace.



### Requirements and Prerequisites

- MATLAB
- HDL Verifier™
- HDL Verifier Support Package for Intel® FPGA Boards
- Fixed-Point Designer™
- Intel Quartus® Prime Software, with a supported version listed in “Supported EDA Tools and Hardware” on page 1-5
- Arrow® DECA MAX® 10 FPGA development board

### Set Up FPGA Development Board

1. Confirm that the power switch is off.
2. Connect the JTAG download cable between the FPGA development board and the host computer.
3. (Optional) Connect the line-in port of the FPGA board with an audio source, such as your cellphone, via a 3.5 mm audio cable. If you skip this step, the captured data is random noises.

### Prepare Example Resources

Set up the Intel Quartus. This example assumes that the Intel Quartus executable is located in the file `C:\altera\18.0\quartus\bin\quartus.exe`. If the location of your executable is different, use your path instead.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...  
                'ToolPath','C:\altera\18.0\quartus\bin\quartus.exe');
```

### Generate FPGA Data Capture Components

Launch the FPGA Data Capture Component Generator tool by executing this command in MATLAB.

```
generateFPGADataCaptureIP
```

This example monitors two signals from the existing HDL design for the audio system: 12 bit `adc_out` and 8 bit counter. The `adc_out` signal is the digital samples of the audio line-in signal. The next signal `counter` is an 8 bit free-running counter. To configure the data capture components to operate on these two signals, follow these steps.

1. Add one row to the **Ports** table by clicking the **Add** button once.
2. Name the first signal to `adc_out` and the second signal to `counter`.
3. Change the bit widths of the two signals to 12 and 8, respectively.
4. Select **FPGA vendor** as Altera.
5. Select **Generated IP language** as Verilog.
6. Select **Sample depth** as 1024. This value is the number of samples of each signal that the data capture tool returns to MATLAB each time a trigger is detected.
7. Select **Max trigger stages** as 2. This value is the maximum number of trigger stages that you can add during data capture to provide multiple trigger conditions.

This figure shows these tool settings.

**FPGA Data Capture Component Generation**

**Description**  
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

**Ports**

Port Name	Bit Width	Use As
adc_out	12	Both trigger and data
counter	8	Both trigger and data

**Target**

Generated IP name: datacapture  
 FPGA vendor: Altera  
 Generated IP language: Verilog  
 Connection type: JTAG  
 Destination folder: hdlsrc

**Capture**

Sample depth: 1024  
 Max trigger stages: 2  
 Include capture condition logic

Generate Cancel Help

To generate the FPGA data capture component, click **Generate**. A report shows the results of the generation.

### Integrate FPGA Data Capture HDL IP

You must include the generated HDL IP core into the example FPGA design. You can copy the module instance code from the generated report. In this example, connect the generated HDL IP with the ADC output and the 8 bit free-running counter.

Open the `adc_top.v` file provided with this example. Uncomment this code.

```
datacapture u0 (
    .clk(adc_clk),
    .clk_enable(adc_valid),
    .ready_to_capture(),
    .adc_out(adc_out),
    .counter(counter[7:0]));
```

Save `adc_top.v`, compile the modified FPGA design, and create an FPGA programming file by using the following Tcl script.

```
system('quartus_sh -t adc_deca_max10.tcl &')
```

The Tcl scripts that are included in this example perform these steps.

1. Create a new Quartus project.
2. Add example HDL files and the generated FPGA data capture HDL files to the project.
3. Compile the design.
4. Program the FPGA.

Wait until the Quartus process successfully finishes before going to the next step. This process takes approximately 5 to 10 minutes.

### **Capture Data**

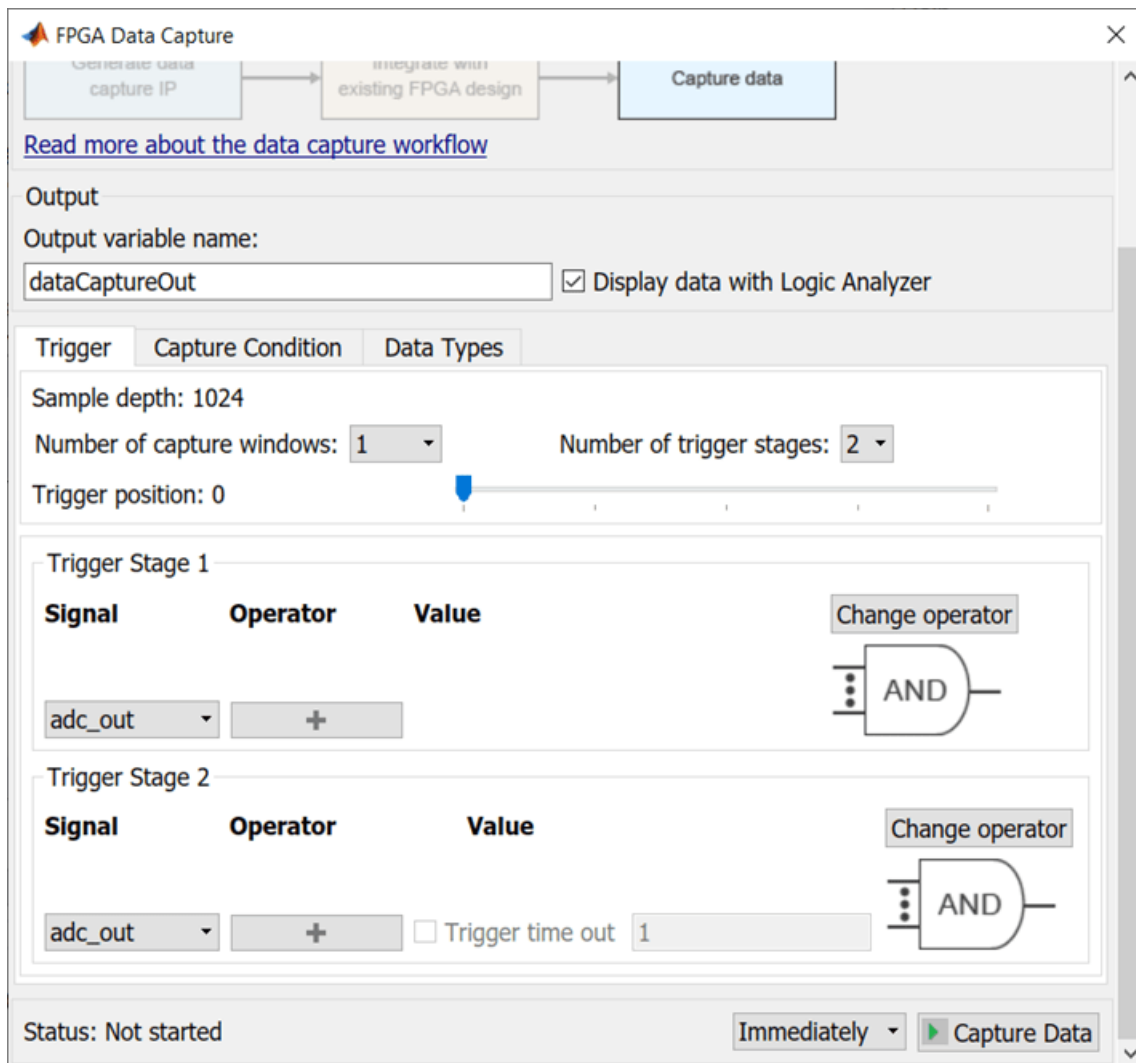
Navigate to the directory where the FPGA data capture component is generated.

```
cd hdlsrc
```

Launch the FPGA Data Capture tool. This tool is customized for the data capture signals.

```
launchDataCaptureApp
```

To start data capture, click **Capture Data**. Data capture object requests one buffer of captured data from the FPGA. The default setting is to capture immediately, without waiting for a trigger condition.



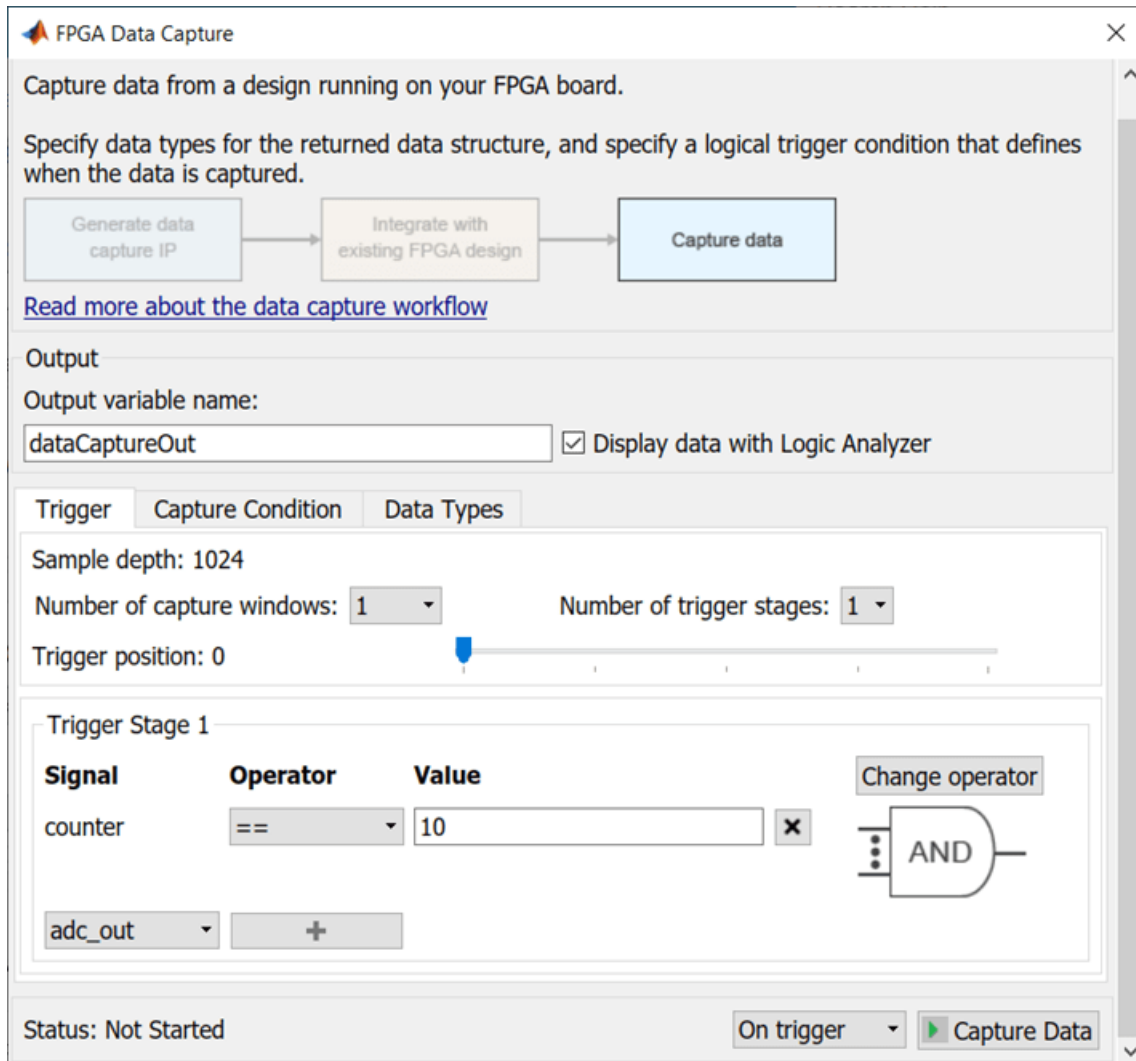
The captured data is saved into a structure, `dataCaptureOut`, in the MATLAB workspace. If you have the DSP System Toolbox™ software, the captured data is also displayed as signal waveforms in the **Logic Analyzer** tool.

### Narrow Scope of Data Capture Using Triggers

To capture data from the FPGA around a particular event, you can configure trigger conditions in the **FPGA Data Capture** tool. For example, to capture the audio data only after a counter reaches a certain value.

Select **Number of trigger stages** as 1. In the **Trigger Stage 1** section, select **Signal** as counter. Enable this trigger signal by clicking the **+** button. Select the corresponding trigger condition value (**Value**) as 10. The trigger mode automatically changes to **On trigger**. This change tells the FPGA to wait for the trigger condition before capturing and returning data. This figure shows these tool settings.

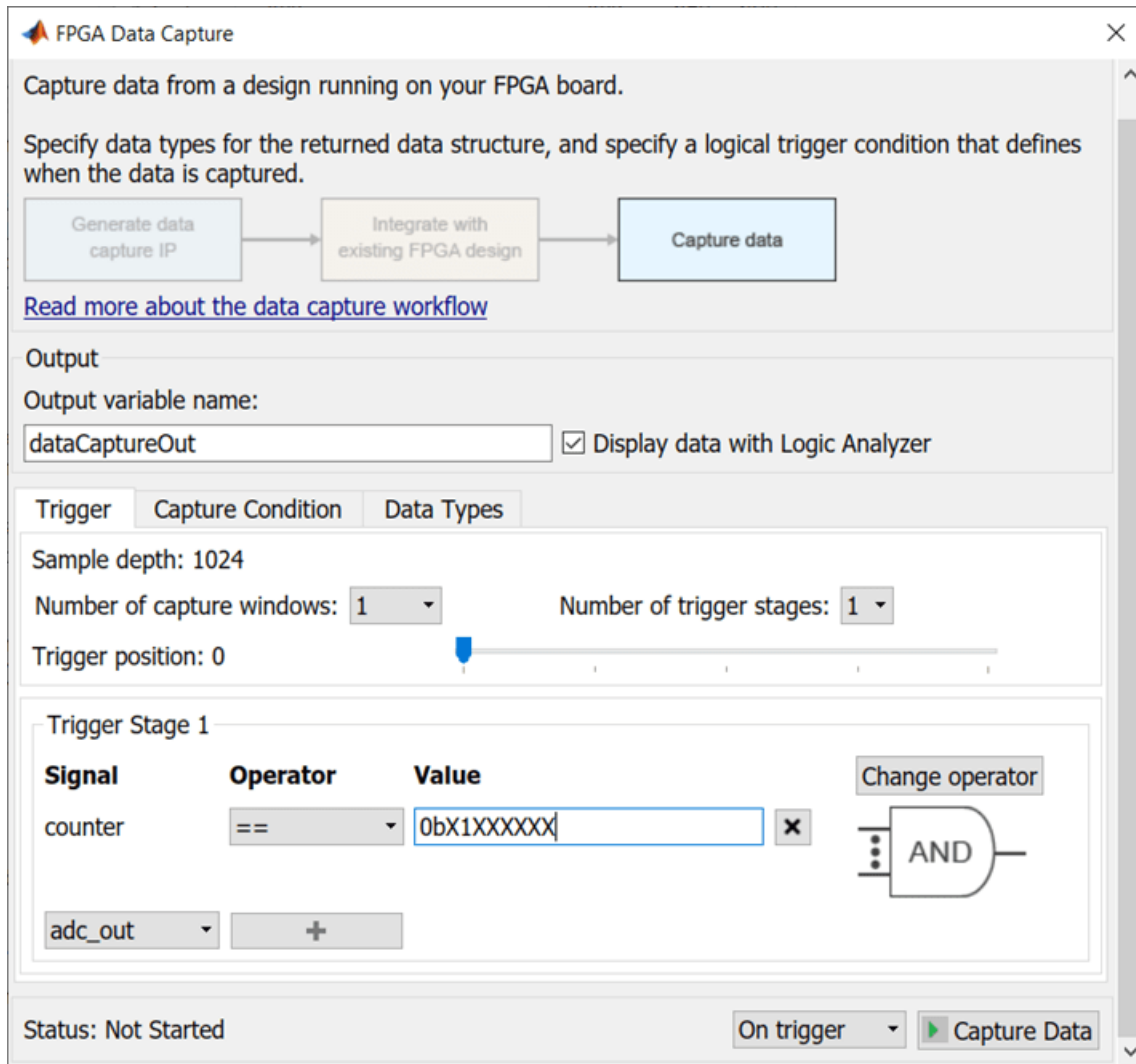




Click **Capture Data** again. This time, the data capture IP returns 1024 samples, captured when it detects that the counter equals 10.

To capture data from the FPGA for specific bits in the trigger value, irrespective of other bits, you can configure the trigger condition with a bit-masked value.

For example, to capture an audio data only when the seventh bit of the counter is 1, set the trigger condition value (**Value**) to `0bX1XXXXXX` as this figure shows.



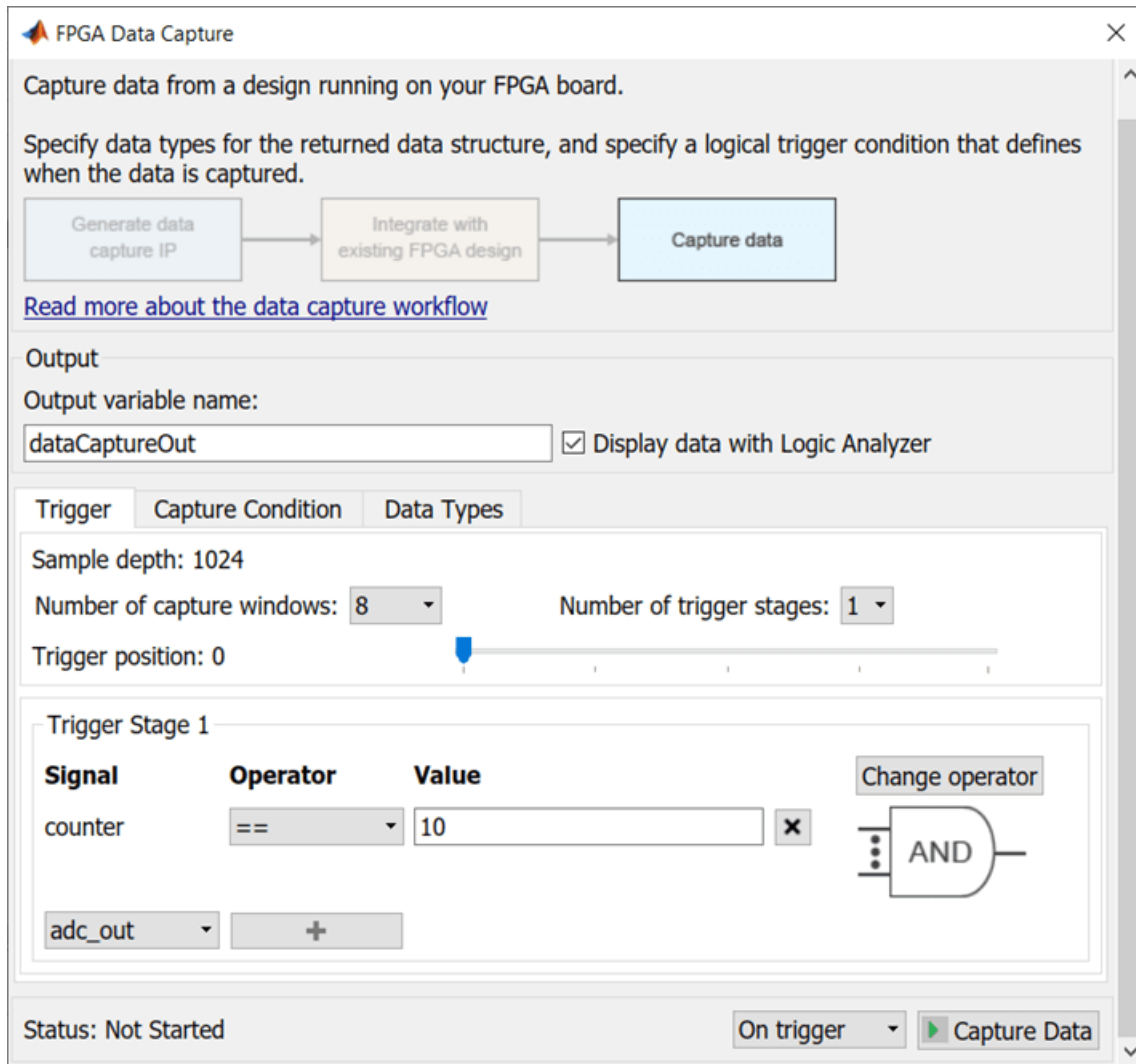
Click **Capture Data** again.

The data capture IP triggers to capture the audio data for counter values in the range [64, 127] and [192, 255].

### Capture Multiple Occurrences of Event

To capture a recurring event from the FPGA, configure **Number of capture windows** in the **FPGA Data Capture** tool.

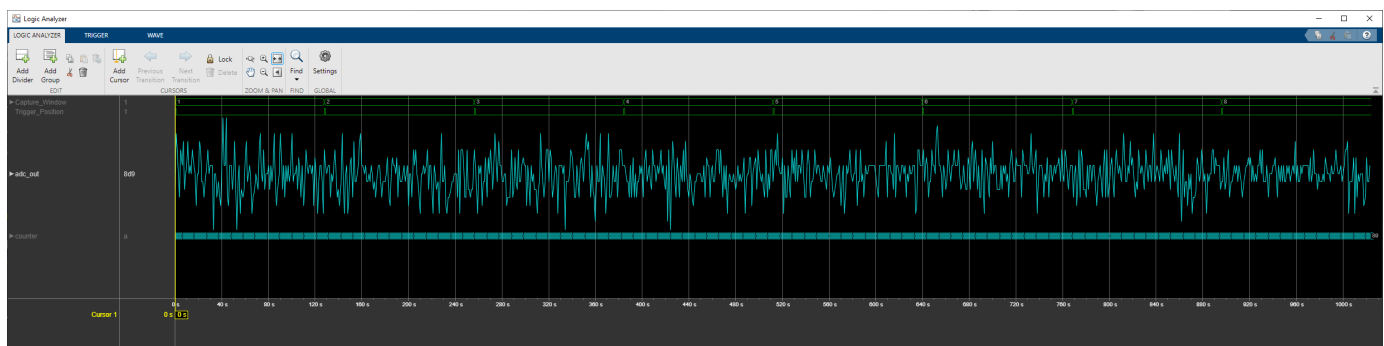
For example, to capture the audio data at eight different time slots, select **Number of capture windows** as 8. This figure shows the updated tool settings.



Click **Capture Data**. The data capture IP returns eight windows of 128 samples each, which amounts to a total sample depth of 1024.

$$\text{Window depth} = \text{Sample depth} / \text{Number of capture windows};$$

The **Logic Analyzer** tool shows this result as eight occurrences of the trigger, with the audio data logged for 128 samples each.



The signals **Capture Window** and **Trigger Position** indicate the corresponding window number and trigger position, respectively.

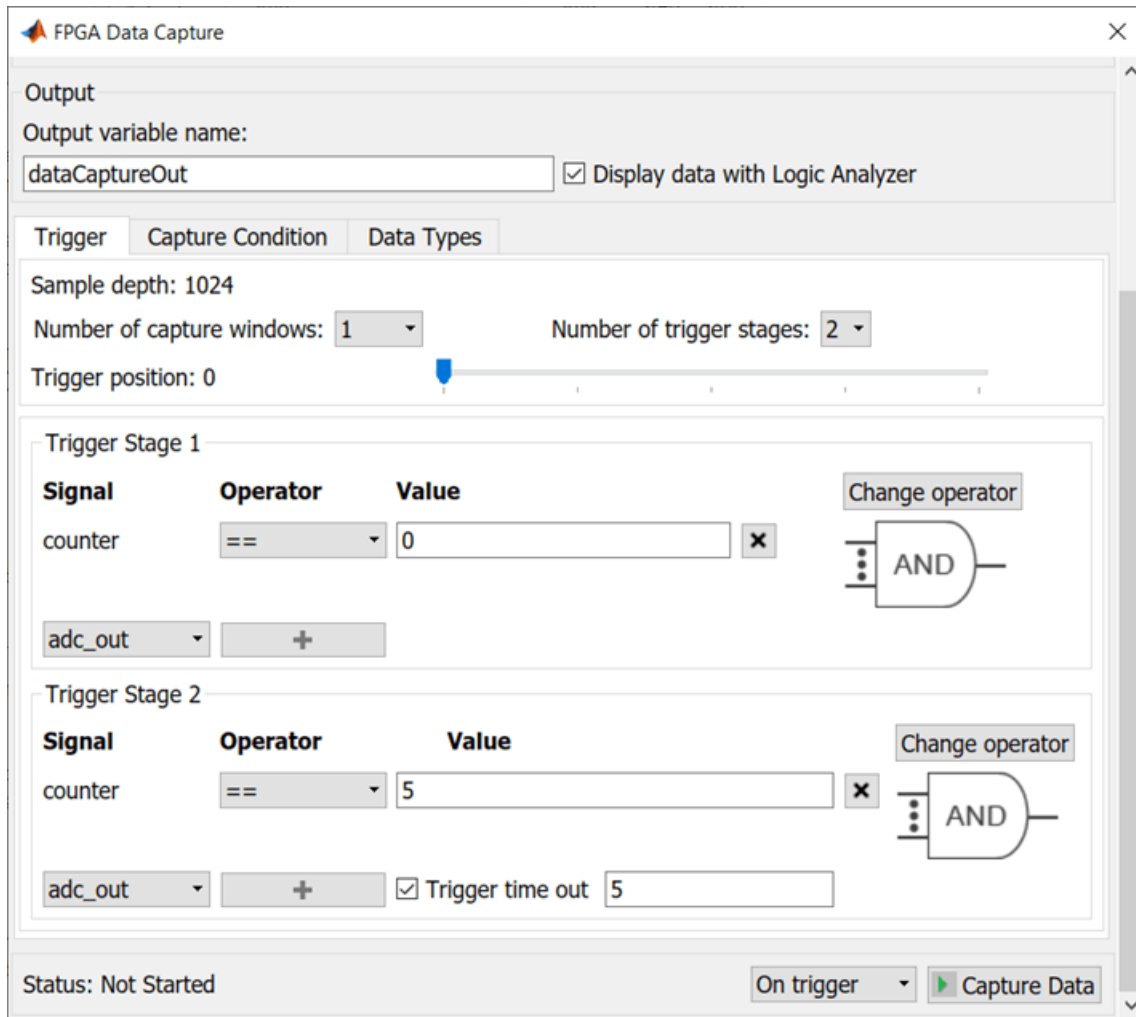
### **Capture Data in Multiple Trigger Stages**

This scenario explains how to capture data by providing a sequence of trigger conditions in multiple trigger stages. For capturing data in multiple trigger conditions, you must select **Number of trigger stages** as a value greater than 1 in the **FPGA Data Capture** tool.

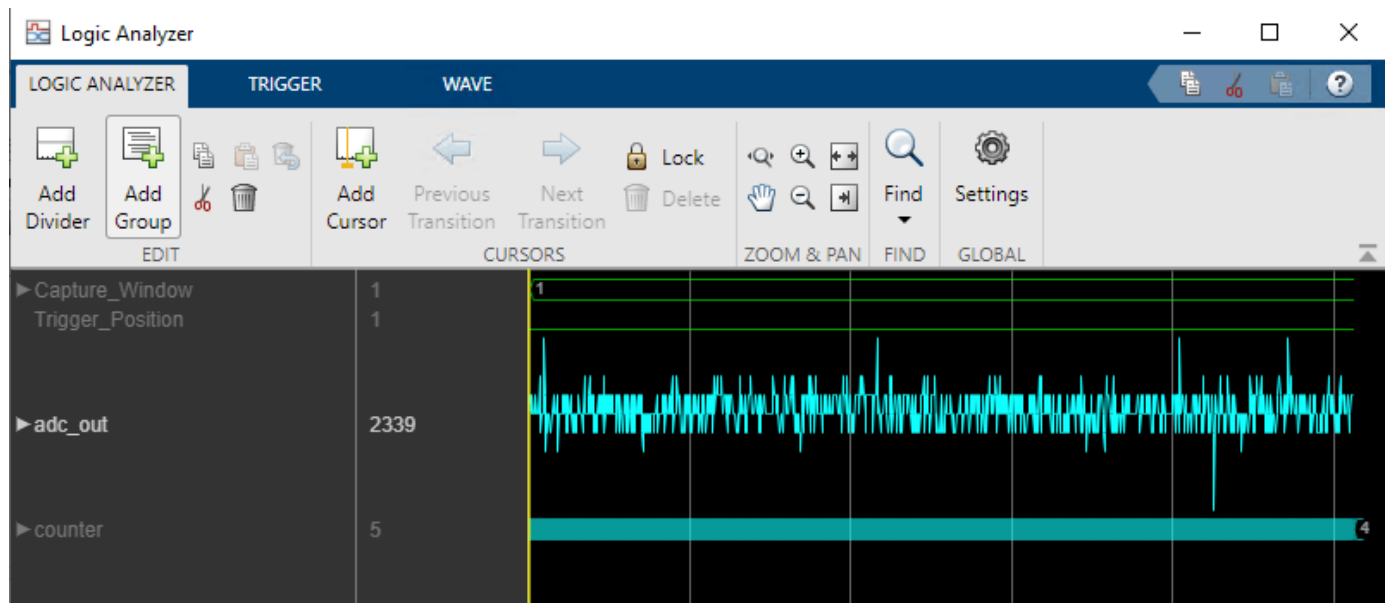
For example, to capture an audio data when the counter value reaches from 0 to 5 in 5 clock cycles, follow these steps.

1. Select **Number of trigger stages** as 2.
2. In the **Trigger Stage 1** section, select **Signal** as counter. Enable this trigger signal by clicking the **+** button . Select the corresponding trigger condition value (**Value**) as 0.
3. In **Trigger Stage 2** section, select Signal as counter. Enable this trigger signal by clicking the **+** button. Select the corresponding trigger condition value (**Value**) as 5. Select **Trigger time out** and set it to 5.

The figure shows the updated tool settings.



Click **Capture Data**. The data capture IP captures 1024 samples when it detects the trigger condition in trigger stage 2 within 5 clock cycles, preceded by the trigger condition detected in trigger stage 1.



## See Also

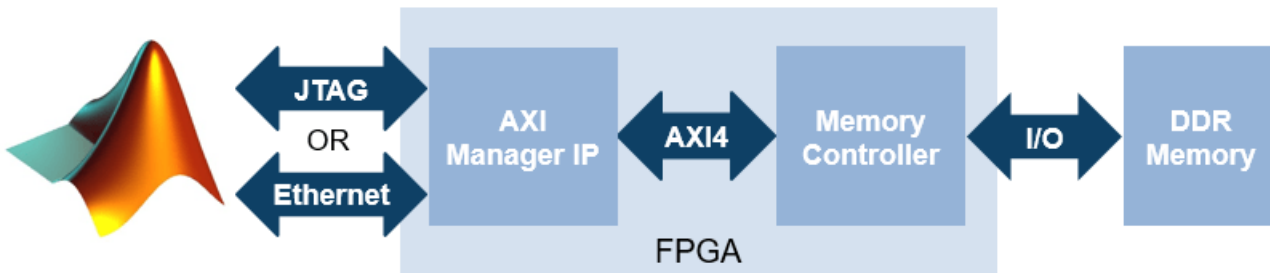
[FPGA Data Capture Component Generator](#) | [FPGA Data Capture](#)

## More About

- “Data Capture Workflow” on page 5-2
- “Triggers” on page 5-6
- “Troubleshooting”

## Access FPGA External Memory Using AXI Manager

This example shows how to use AXI manager to access external DDR memories connected to an FPGA. The FPGA design instantiates an Intel® DDR memory controller for accessing the DDR memories. This memory controller provides a memory-mapped subordinate interface for read and write operations from the FPGA. The AXI manager feature provides an AXI manager IP that allows MATLAB® to access any memory-mapped subordinate IPs in the FPGA. In this example, we demonstrate how to integrate the JTAG AXI manager IP or Ethernet AXI manager IP into a Qsys design, and then read and write the DDR memory from MATLAB.



### Requirements

- Intel Quartus® Prime software with a supported version listed in “Supported EDA Tools and Hardware” on page 1-5
- Intel Arrow® DECA MAX® 10 FPGA development kit
- HDL Verifier™ Support Package for Intel FPGA Boards
- USB-Blaster II™ download cable
- Ethernet cable

### Set Up for Using JTAG AXI Manager

Step 1: Set up FPGA board. Make sure that the DECA board is connected to the host computer via the USB JTAG cable.

Step 2: Prepare example in MATLAB

Set up the Intel Quartus Prime tool path. Use your own Intel Quartus Prime installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...
    'ToolPath','C:\intelFPGA\18.0\quartus\bin64\quartus.exe');
```

Create a Quartus project for this example. This tcl script creates the Quartus project, and adds the design files to the project.

```
system('quartus_sh -t create_project_deca.tcl')
```

This command takes a few seconds to finish. When it is done, a Quartus project named "aximaster\_deca.qpf" is created in your current directory.

Step 3: Configure Quartus Prime project to use AXI manager

Copy the IP to the project directory using the following command.

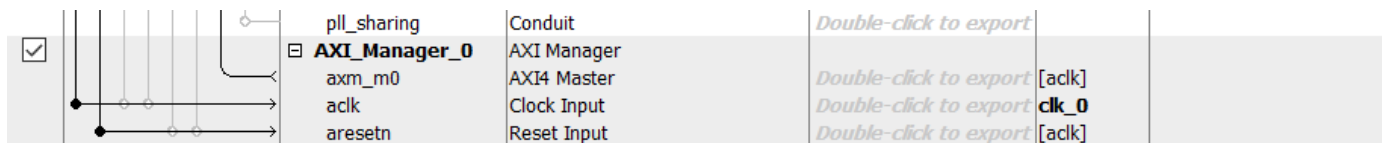
```
setupAXIManagerForQuartus('aximaster_deca.qpf')
```

Open the generated Quartus project in GUI mode. You can double-click the project in a file browser, or execute this command in MATLAB.

```
system('quartus aximaster_deca.qpf &')
```

Step 4: Inspect JTAG AXI manager IP in Qsys design (optional)

In the Quartus GUI, open the Qsys design file, system.qsys, and inspect how the AXI Manager IP is connected to the DDR controller.



Step 5: Generate FPGA programming file and program FPGA

To generate the FPGA programming file, click the "Start Compilation" button in Quartus Prime.

After generating the programming file, program the FPGA in MATLAB using the following command.

```
filProgramFPGA('Altera','output_files/aximaster_deca.sof',1)
```

### Set Up for Using Ethernet AXI Manager

Step 1: Set up FPGA board. Make sure that the DECA board is connected to the host computer via both the USB Blaster II download cable and the Ethernet cable.

Step 2: Prepare example in MATLAB

Set up the Intel Quartus Prime tool path. Use your own Intel Quartus Prime installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...
    'ToolPath','C:\intelFPGA\18.0\quartus\bin64\quartus.exe');
```

Create a Quartus project for this example. This tcl script would create the Quartus project, and add the design files we created to the project.

```
system('quartus_sh -t create_project_eth_deca.tcl')
```

When it is done, a Quartus project named "eth\_aximaster\_deca.qpf" is created in your current directory.

Step 3: Configure Quartus Prime project to use the Ethernet AXI manager

Copy the IP to the project directory using the following command.

```
setupAXIManagerForQuartus('eth_aximaster_deca.qpf')
```

Open the generated Quartus project in GUI mode. Double-click the project in a file browser, or execute this command in MATLAB.



```
system('quartus eth_aximaster_deca.qpf &')
```

Step 4: Inspect Ethernet AXI manager IP in Qsys design (optional)

In the Quartus GUI, open the Qsys design file, aximaster.qsys, and inspect how the UDP AXI Manager IP is connected to the DDR controller.

The Ethernet-based AXI manager IP has been assigned a target IP address of 192.168.1.2 and UDP port value of 50101. These values can be changed by double clicking on the ethernet\_mac\_hub\_IP in Qsys.

Step 5: Generate FPGA programming file and program FPGA

To generate the FPGA programming file, click the "Start Compilation" button in Quartus Prime.

After generating the programming file, program the FPGA in MATLAB using the following command.

```
filProgramFPGA('Altera','output_files/eth_aximaster_deca.sof',1)
```

### Read and Write Operations to the FPGA

After programming the FPGA, you can read and write into the AXI subordinate connected to the AXI manager IP. In this example, the data will be written to the DDR memory connected to the FPGA, and retrieved back into MATLAB.

Create the AXI manager object in MATLAB.

If using JTAG AXI manager:

```
h = aximanager('Intel')
```

If using Ethernet AXI manager:

```
h = aximanager('Intel','interface','UDP','deviceAddress','192.168.1.2','port','50101');
```

Run these two commands to write a single word of value 100 into DDR memory at address 0 and read it back by using the AXI manager object.

```
writememory(h,0,100)
readmemory(h,0,1)
```

You can also read and write large vectors of data into DDR memory using a single read/write command in MATLAB. These commands automatically break down the large amount of data into smaller bursts so that they can be transferred via AXI4 protocol. The function uses the largest possible burst size for each burst to maximize the throughput performance. The following commands write 100000 words into DDR memory and read them back. It also checks if the read back data are correct and reports the execution time.

```
address = 0;
data = 1:100000;
writememory(h,address,data);
r = readmemory(h,address,100000);
assert(all(r==data));
```

### See Also

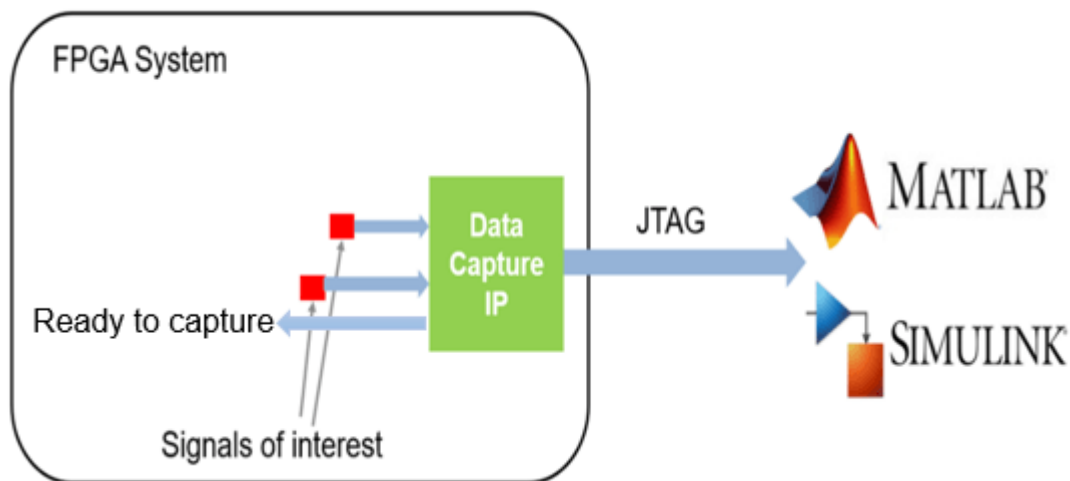
aximanager | writememory | readmemory

## **More About**

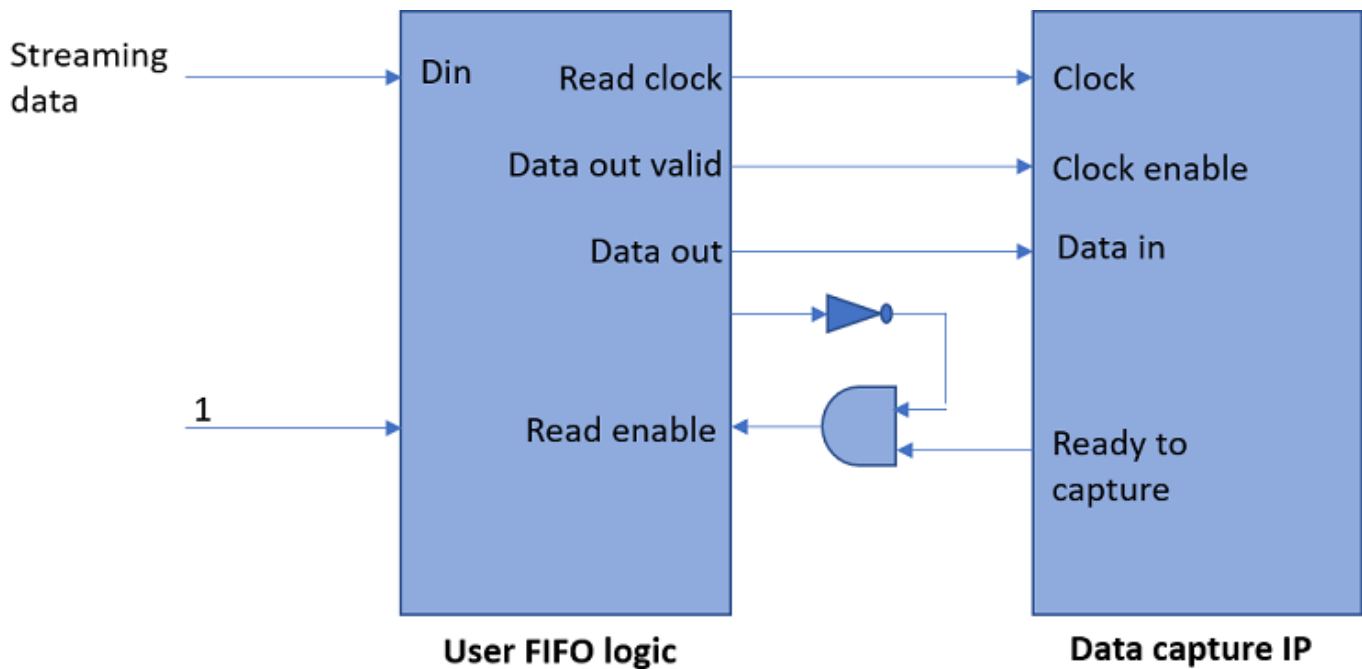
- “Set Up AXI Manager” on page 3-2
- “Ethernet AXI Manager” on page 3-12
- “Troubleshooting”

## Stream Audio Signal from Intel FPGA Board Using Ready-to-Capture Signal

This example shows how to use a ready-to-capture signal in an FPGA data capture with existing HDL code to read FPGA streaming signals. This example starts with an existing FPGA design that implements an on-chip analog-to-digital converter (ADC) to sample an audio signal. The ADC intellectual property (IP) exposes an Avalon® memory-mapped (MM) slave interface for control and an Avalon streaming interface for data output. The existing example contains a simple Avalon MM master to start the ADC. Use FPGA data capture to collect the ADC output data from the Avalon streaming interface and stream the data to the MATLAB® workspace.



The `ready_to_capture` signal is output from the FPGA data capture HDL IP. This output captures continuous data provided that you place a first-in first-out (FIFO) in front of the FPGA data capture HDL IP and check the `ready_to_capture` signal before streaming data to the data capture IP. You might also need to check that the FIFO is not empty. This figure illustrates the continuous data capture workflow.



The audio data from the Avalon streaming interface is written into a FIFO, and the FIFO is read when the `ready_to_capture` signal is asserted. To avoid data overflow, the FIFO must be large enough to capture data when `ready_to_capture` is deasserted. In this example, the FIFO depth is configured to 16 k and the audio sampling frequency is 50 kHz.

### Requirements and Prerequisites

- MATLAB
- HDL Verifier™
- HDL Verifier Support Package for Intel® FPGA Boards
- Fixed-Point Designer™
- Intel Quartus® Prime software with a supported version listed in “Supported EDA Tools and Hardware” on page 1-5
- Arrow® DECA MAX® 10 FPGA Evaluation Kit
- DSP System Toolbox™

### Set Up FPGA Development Board

1. Confirm that the power switch is off.
2. Connect the JTAG download cable between the FPGA development board and the host computer.
3. (Optional) Connect the line-in port of the FPGA board with an audio source, such as your cellphone, via 3.5 mm audio cable. If you skip this step, the captured data will be random noises.

### Prepare Example Resources

Set up the Intel Quartus. This example assumes that the Intel Quartus executable is located in the file `C:\altera\18.0\quartus\bin\quartus.exe`. If the location of your executable is different, use your path instead.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...  
                'ToolPath','C:\altera\18.0\quartus\bin\quartus.exe');
```

### Generate FPGA Data Capture Components

Launch the **FPGA Data Capture Component Generator** tool by executing this command in MATLAB.

```
generateFPGADataCaptureIP
```

This example monitors one signal from the existing HDL code for the audio system. The signal is a 12 bit `adc_out`. The `adc_out` signal is the digital samples of the audio line-in signal. To configure the data capture components to operate on this signal, follow these steps.

1. Name the signal to `adc_out` in the ports table.
2. Change the bit width of the signal to 12.
3. Select **FPGA vendor** as Altera.
4. Select **Generated IP language** as Verilog.
5. Select **Sample depth** as 8192. This value is the number of samples of each signal that the data capture tool returns to MATLAB each time a trigger is detected.

This figure shows these tool settings.

**FPGA Data Capture Component Generation**

**Description**  
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

**Ports**

Port Name	Bit Width	Use As
adc_out	12	Both trigger and data

**Target**

Generated IP name: datacapture

FPGA vendor: Altera

Generated IP language: Verilog

Connection type: JTAG

Destination folder: hdlsrc

**Capture**

Sample depth: 8192

Max trigger stages: 1

Include capture condition logic

Generate Cancel Help

To generate the FPGA data capture component, click **Generate**. A report shows the results of the generation.

### Integrate FPGA Data Capture HDL IP

You must include the generated HDL IP core into the example FPGA design. You can copy the module instance code from the generated report. In this example, connect the generated HDL IP with the ADC output via a FIFO.

Open the `readyToCapture_top.v` file provided with this example. Uncomment this code.

```
datacapture u0(
  .clk(clock),
  .clk_enable(adc_valid),
  .ready_to_capture(ready_to_capture),
  .adc_out(adc_out)
);
```

Save `readyToCapture_top.v`, compile the modified FPGA design, and create an FPGA programming file by using the following Tcl script.

```
system('quartus_sh -t readyToCapture_deca_max10.tcl &')
```

The Tcl scripts that are included in this example perform these steps.

1. Create a new Quartus project.
2. Add example HDL files and the generated FPGA data capture HDL files to the project.
3. Compile the design.
4. Program the FPGA.

Wait until the Quartus process successfully finishes before going to the next step. This process takes approximately 5 to 10 minutes.

### Capture Data

Navigate to the directory where the FPGA data capture component is generated.

```
cd hdlsrc
```

You must set the capture mode to **Immediate** to use a ready-to-capture signal in an FPGA data capture. To capture data in the immediate mode, follow these steps.

1. Create an FPGA data capture System object™ and configure these properties.

```
DataCaptureObj = datacapture;
DataCaptureObj.TriggerPosition = 0;
DataCaptureObj.NumCaptureWindows = 1;
setRunImmediateFlag(DataCaptureObj,1);
```

2. Capture FPGA data continuously. In this example, capture `NumberOfSampledepth` snapshots of data. You can modify the `NumberOfSampledepth` value as needed.

```
NumberOfSampledepth = 10;
Sample_depth = 8192;
adc_out = int16(zeros(NumberOfSampledepth*Sample_depth, 1));
for i = 1:NumberOfSampledepth
    [~,~,adc_out(i*Sample_depth-(Sample_depth-1):i*Sample_depth)] = step(DataCaptureObj);
end
```

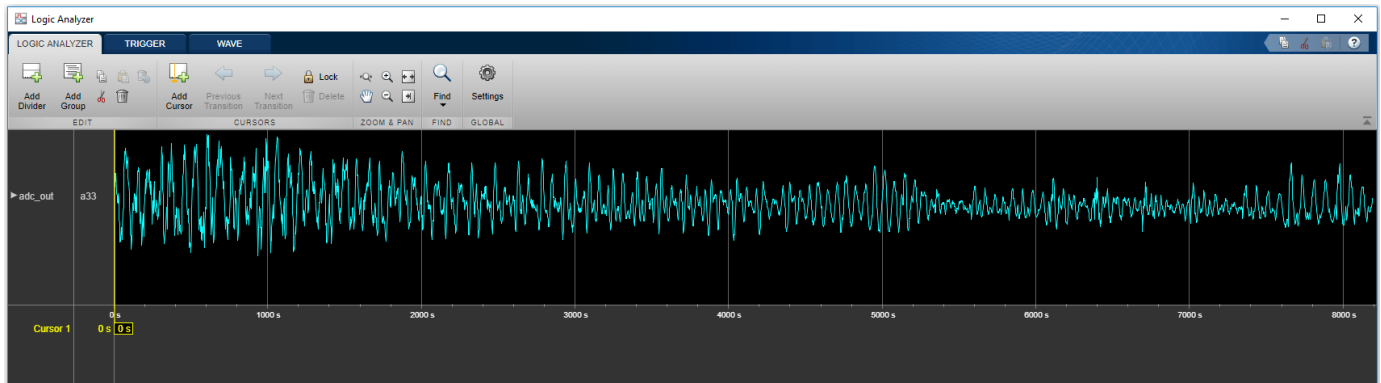
3. Save the captured audio data by writing the captured data to `.wav` format. Process or replay the captured data at a later time.

```
audiowrite('captured_audio_file.wav',adc_out,50000)
```

4. View the captured data in the **Logic Analyzer** tool. The **Logic Analyzer** tool can take a few seconds to load the captured snapshots of data.

```
scope = dsp.LogicAnalyzer('NumInputPorts',1, ...
    'DisplayChannelFormat','Analog', ...
    'DisplayChannelHeight',100);
tags = getDisplayChannelTags(scope);
modifyDisplayChannel(scope,tags{1},'Name','adc_out');
scope(adc_out);
```

This figure shows 8 k samples of audio data in the **Logic Analyzer** tool.



## See Also

### Tools

**FPGA Data Capture Component Generator | FPGA Data Capture**

### Functions

`setRunImmediateFlag`

## More About

- “Data Capture Workflow” on page 5-2
- “Triggers” on page 5-6
- “Troubleshooting”



## IP Core Generation Workflow with Ethernet-Based AXI Manager

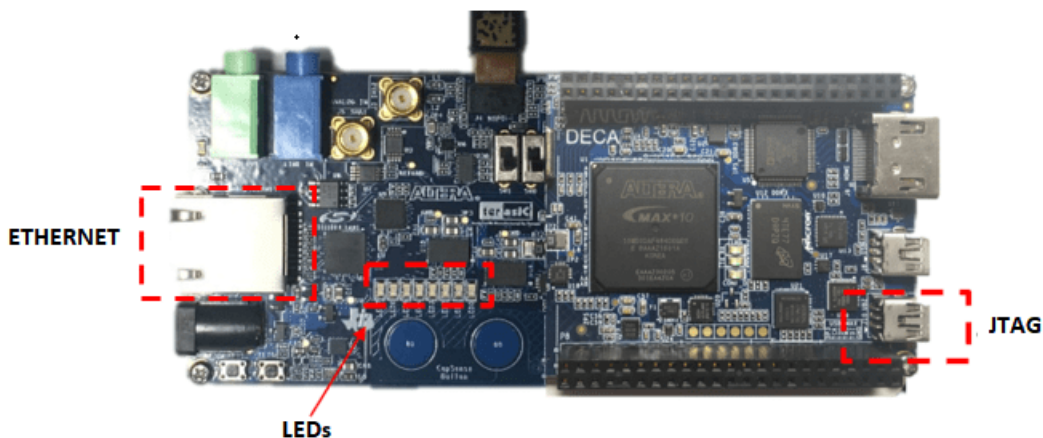
This example uses the HDL Verifier™ Ethernet AXI manager IP to access the HDL Coder™ product generated registers. AXI manager provides access to FPGA registers from MATLAB® directly.

### Requirements

- Intel® Quartus® Prime, with a supported version listed in “Supported EDA Tools and Hardware” on page 1-5
- Arrow® DECA MAX® 10 FPGA evaluation kit
- HDL Coder Support Package for Intel FPGA Boards
- HDL Verifier Support Package for Intel FPGA Boards
- USB-JTAG cable
- Cross-over Ethernet cable

### Arrow DECA MAX 10 FPGA Evaluation Kit

This figure shows the Arrow DECA MAX 10 FPGA evaluation kit.



### Example Reference Design

The reference design, AXI Manager - Ethernet, uses MathWorks® IP and a MATLAB command line interface for issuing read and write operations. To use this design, you must have the HDL Verifier product. The `plugin_rd.m` for this reference design is shown below.

```
function hRD = plugin_rd()
% Reference design definition

% Copyright 2016-2020 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Altera QUARTUS II');

hRD.ReferenceDesignName = 'AXI Manager - Ethernet (requires HDL Verifier)';
hRD.BoardName = 'Arrow DECA MAX 10 FPGA evaluation kit';
```

```

% Tool information
hRD.SupportedToolVersion = {'17.1', '18.1', '19.1', '20.1'};

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign('CustomQsysPrjFile', 'system_soc.qsys');
hRD.CustomConstraints = {'system_soc.sdc', 'create_project_eth_deca.tcl'};
hRD.CustomFiles = {'phy_to_gmii'}; % Adding other files for custom IP core

% Add ip from support package
msg = message('hdlcommon:plugin:IPRepositoryHDLVerifierAlteraNotFound',...
  '<a href="matlab:matlab.addons.supportpackage.internal.explorer.showSupportPackages({'HDLVALTE
hRD.addIPRepository('IPListFunction', 'hdlverifier.fpga.quartus.iplist', 'NotExistMessage', msg);

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',    'altpll_0.c0', ...
    'ResetConnection',    'clk_0.clk_reset', ...
    'DefaultFrequencyMHz', 50);

% add AXI4 slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'udp_axi_manager_0.axi4_udp', ...
    'BaseAddress',          '0x0000_0000', ...
    'InterfaceType',        'AXI4');    % [ 'AXI4-Lite' | 'AXI4' ]

% Specify Embedded Coder Support Package to use for Software Interface
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.None;
    % [ None | Zynq | AlteraSoC ]

```

The corresponding `plugin_rd.m` file for the AXI manager reference design is located at `plugin_rd.m`.

### Execute IP Core Workflow

The reference design generates an HDL IP core that blinks LEDs on the DECA board. Locate the files that are used in the following demonstration at this path.

```
matlab/toolbox/hdlcoder/supportpackages/altera/+hdlipcore/+board/ArrowDECA
```

Follow these steps to execute the IP core workflow for AXI Manager - Ethernet reference design, which uses Ethernet AXI manager IP. Using this reference design, you can generate an HDL IP core that blinks LEDs on the DECA board. To generate an HDL IP core, follow these steps.

1. Set up the Intel Quartus tool path. Use your own Intel Quartus installation path when executing the command.

```
hdlsetuptoolpath('ToolName', 'Altera QUARTUS II', 'ToolPath', ...
  'C:\intelFPGA\20.1\quartus\bin64\quartus.exe');
```

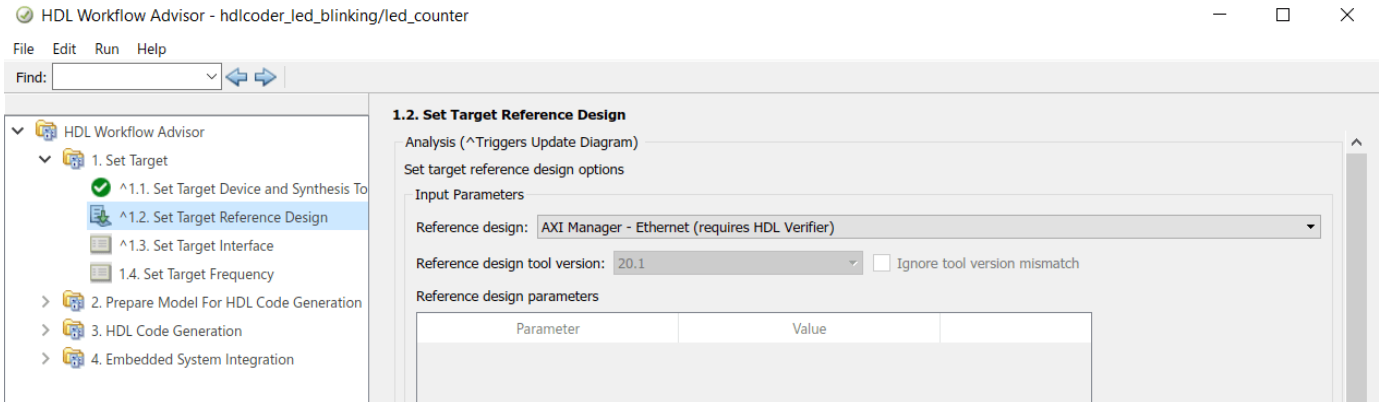
2. Open the Simulink model that implements LED blinking by executing this command in MATLAB.

```
open_system('hdlcoder_led_blinking')
```

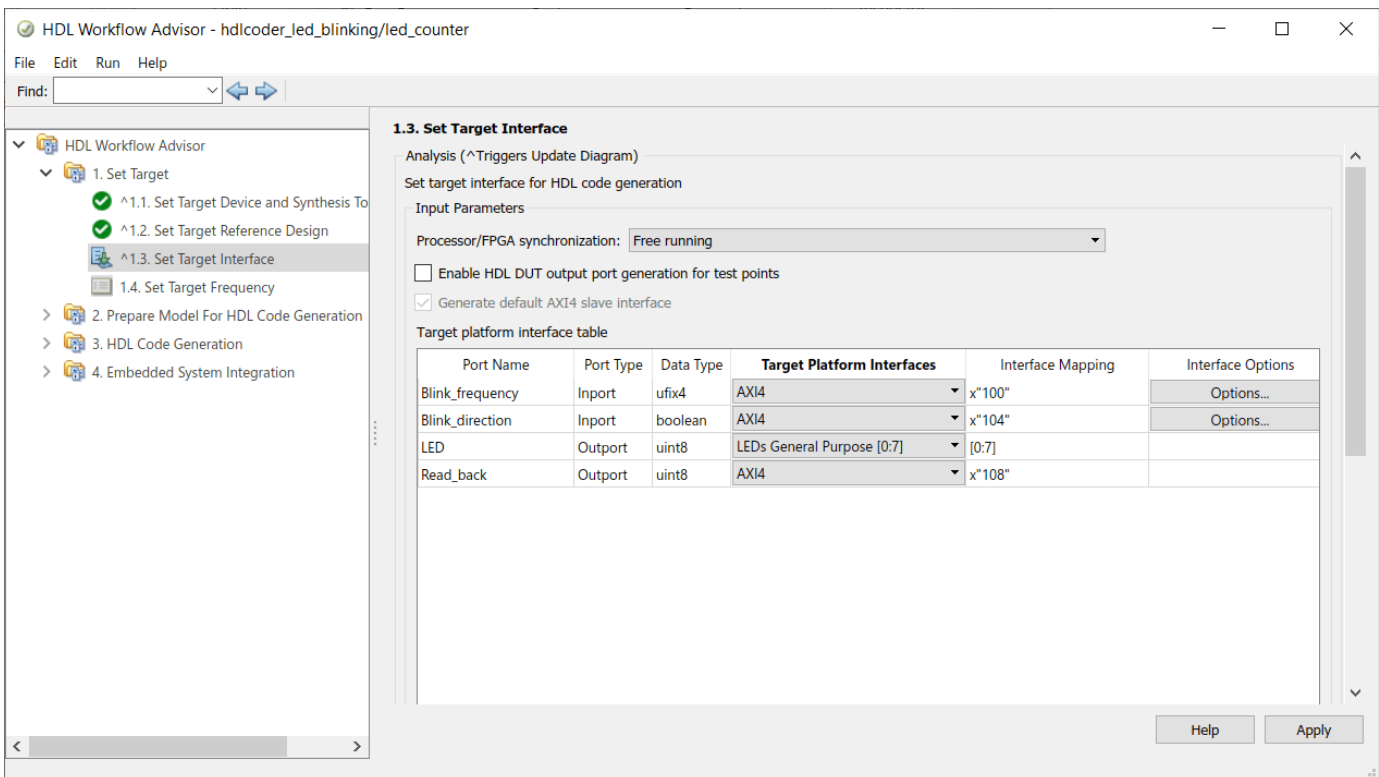
3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and selecting **HDL Code** followed by **HDL Workflow Advisor**.

4. In step 1.1, select **Target workflow** as IP Core Generation and **Target platform** as Arrow DECA MAX 10 FPGA evaluation kit. Click **Run This Task**.

5. In step 1.2, select **Reference design** as AXI Manager - Ethernet (requires HDL Verifier).

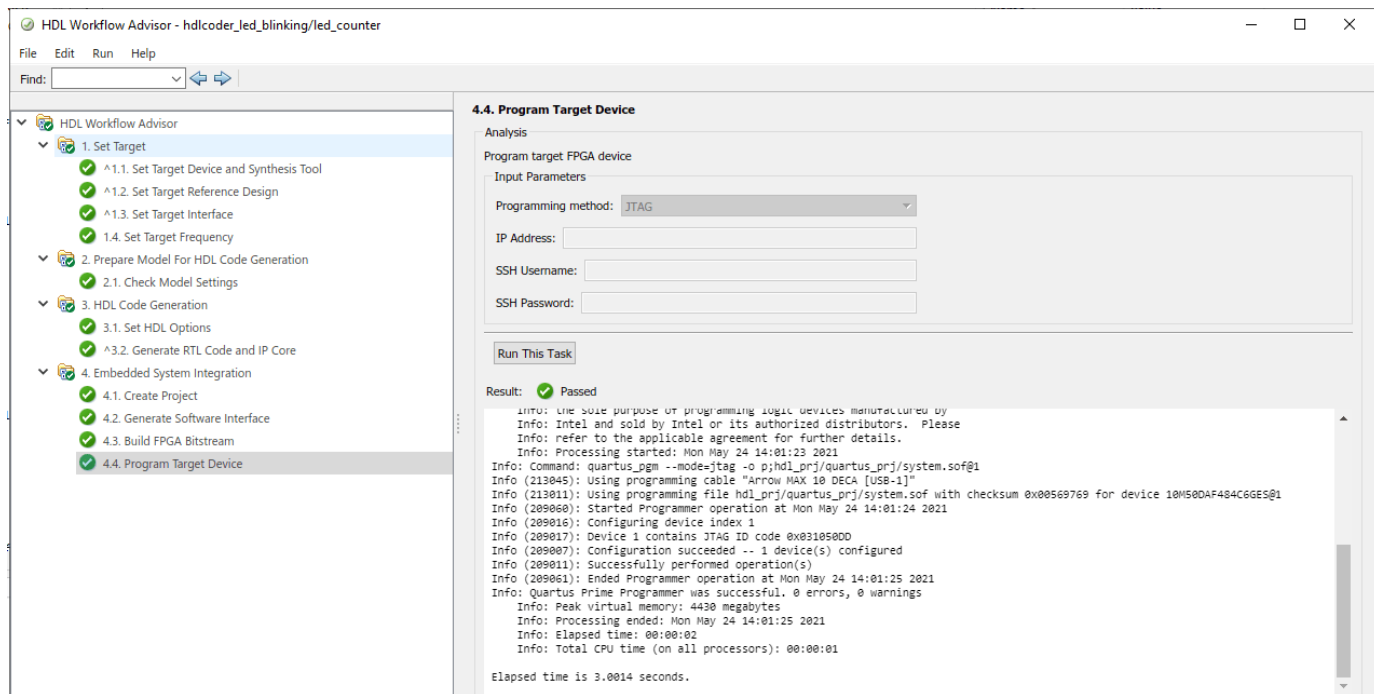


6. In step 1.3, assign `Blink_frequency`, `Blink_direction`, and `Read_back` ports to the AXI4 interface. Assign the LED port to LEDs General Purpose [0:7].



7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

The Ethernet AXI manager IP in this design has a default target IP address of 192.168.1.2 and default UDP port value of 50101. Configure the network interface card (NIC) of the host machine accordingly.



### Determine Addresses from IP Core Report

The base address for an HDL Coder IP core is defined in the reference design `plugin_rd.m`. For this design, the base address is `0x0000_0000`. The IP core report register address mapping table shows the offsets.

### Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 1901161646
Blink_frequency_Data	0x100	data register for Inport Blink_frequency
Blink_direction_Data	0x104	data register for Inport Blink_direction
Read_back_Data	0x108	data register for Output Read_back

### HDL Verifier AXI Manager - Ethernet Command Line Interface

If you have the HDL Verifier support package for Intel FPGA boards and select the AXI Manager - Ethernet reference design, then you can use the MATLAB command line interface to access the IP core that is generated by the HDL Coder product.

To write and read from the DDR memory, follow these steps.

1. Create an AXI manager object.

```
h = aximanager('Intel', 'interface', 'UDP', 'DeviceAddress', '192.168.1.2')
```

2. Issue a write command. For example, disable the DUT.

```
h.writememory('4',0)
```

3. Re-enable the DUT.

```
h.writememory('4',1)
```

4. Read the current counter value.

```
h.readmemory('108',1)
```

## See Also

[aximanager](#) | [writememory](#) | [readmemory](#)

## More About

- “Set Up AXI Manager” on page 3-2
- “Ethernet AXI Manager” on page 3-12
- “Getting Started with the HDL Workflow Advisor” (HDL Coder)
- “Troubleshooting”

## Access DUT Registers on Intel Pure FPGA Board Using IP Core Generation Workflow

This example shows how to use the HDL Coder™ IP core generation workflow to develop reference designs for Intel® parts that do not use an embedded ARM® processor present but that still utilize the HDL Coder generated AXI interface to control the design under test (DUT). This example uses the HDL Verifier™ AXI Manager IP to access the HDL Coder generated DUT registers from MATLAB®. Alternatively, you can use the Intel Qsys JTAG to Avalon® Master Bridge IP to access the FPGA registers using Tcl commands in the Qsys system console. For the Intel JTAG AXI Master, you must create a custom reference design. The FPGA design is implemented on the Arrow® DECA MAX® 10 FPGA evaluation kit.

### Requirements

- Intel Quartus® Prime Standard, with a supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder)
- Arrow DECA MAX 10 FPGA evaluation kit
- HDL Coder Support Package for Intel FPGA Boards
- HDL Verifier Support Package for Intel FPGA Boards
- HDL Coder Support Package for Intel SoC Devices (required only when you want to integrate the IP core into your own custom reference design)

### Arrow DECA MAX 10 FPGA Evaluation Kit

This figure shows the Arrow DECA MAX 10 FPGA evaluation kit.



### Example Reference Designs

Designs that can benefit from using the HDL Coder IP core generation workflow without using either an embedded ARM processor or an Embedded Coder™ support package but still leverage the HDL Coder generated AXI4 registers can include one of these IP sets.

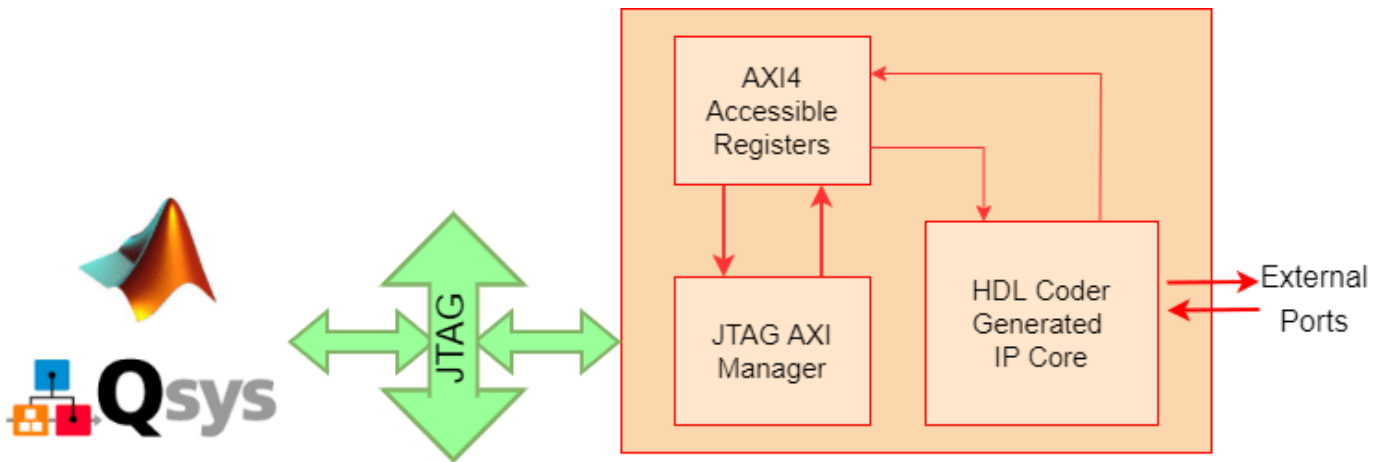
- HDL Verifier AXI Manager + HDL Coder IP Core
- JTAG Master + HDL Coder IP Core
- Nios® II + HDL Coder IP Core
- PCIe® Endpoint + HDL Coder IP Core

This example includes two reference designs.

- The Default System reference design uses MathWorks® IP and a MATLAB command line interface for issuing read and write commands. To use this design, you must have the HDL Verifier product.
- The Intel JTAG to AXI Master reference design uses Quartus IP for the JTAG to AXI Master and requires using the Quartus Tcl console to issue read and write commands.

Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		clk_0	Clock Source				
		clk_in	Clock Input	clk	exported		
		clk_in_reset	Reset Input	reset	[clk_in]		
		clk	Clock Output	Double-click to export	clk_0		
		clk_reset	Reset Output	Double-click to export	clk_0		
<input checked="" type="checkbox"/>		altpll_0	Avalon ALTPLL				
		indk_interface	Clock Input	Double-click to export	clk_0		
		indk_interface_reset	Reset Input	Double-click to export	[indk_interf...		
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	[indk_interf...		
		c0	Clock Output	Double-click to export	altpll_0_c0		
		areset_conduit	Conduit	Double-click to export			
		locked_conduit	Conduit	Double-click to export			
		phasedone_conduit	Conduit	Double-click to export			
<input checked="" type="checkbox"/>		master_0	JTAG to Avalon Master Bridge				
		clk	Clock Input	Double-click to export	altpll_0_c0		
		clk_reset	Reset Input	Double-click to export	[clk]		
		master	Avalon Memory Mapped Master	Double-click to export			
		master_reset	Reset Output	Double-click to export			
<input checked="" type="checkbox"/>		led_count_ip_0	led_count_ip				
		ip_clk	Clock Input	Double-click to export	altpll_0_c0		
		ip_rst	Reset Input	Double-click to export	[ip_clk]		
		axi_clk	Clock Input	Double-click to export	altpll_0_c0		
		axi_reset	Reset Input	Double-click to export	[axi_clk]		
		s_axi	AXI4 Slave	Double-click to export	[axi_clk]	0x0000_0000	0x0000_ffff
		led_count_ip_0_GPLED	Conduit	led_count_ip_0_GPLED	[ip_clk]		

The two reference designs differ by only the JTAG manager IP that they use.



### HDL Verifier AXI Manager Reference Design

In the IP core generation workflow of the HDL Workflow Advisor, in the **Set Target Reference Design** step, enable the **Insert AXI Manager (HDL Verifier required)** parameter. This option adds AXI Manager IP automatically into the reference design and connects the added IP to the DUT IP using the AXI4-slave interface. The next section details the steps to auto-insert the JTAG AXI Manager IP in the reference design.

### Execute IP Core Workflow

Follow these steps to execute the IP core workflow for the Default System reference design, which uses JTAG AXI Manager IP. Using this reference design, you can generate an HDL IP core that blinks LEDs on the DECA board. To generate the HDL IP core, follow these steps.

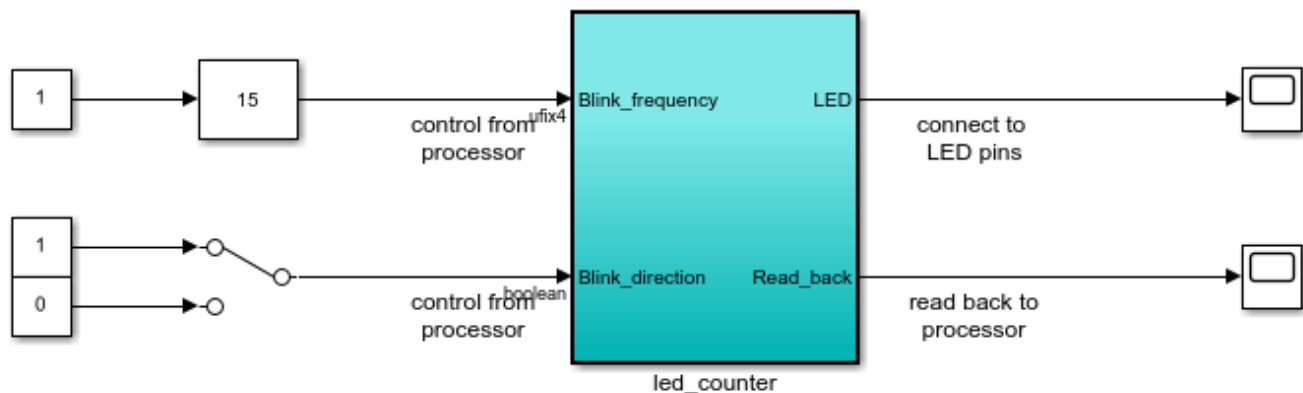
1. Set up the Intel Quartus® tool path. Use your own Intel Quartus installation path by executing this command in MATLAB.

```
hdlsetuptoolpath('ToolName','Altera QUARTUS II', ...
    'ToolPath','C:\intelFPGA\20.1\quartus\bin64\quartus.exe');
```

2. Open the Simulink model that implements LED blinking by executing this command in MATLAB.

```
open_system('hdlcoder_led_blinking')
```

### Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:  
`hdladvisor('hdlcoder_led_blinking/led_counter')`

**Launch HDL Workflow Advisor**

**Run Demo**

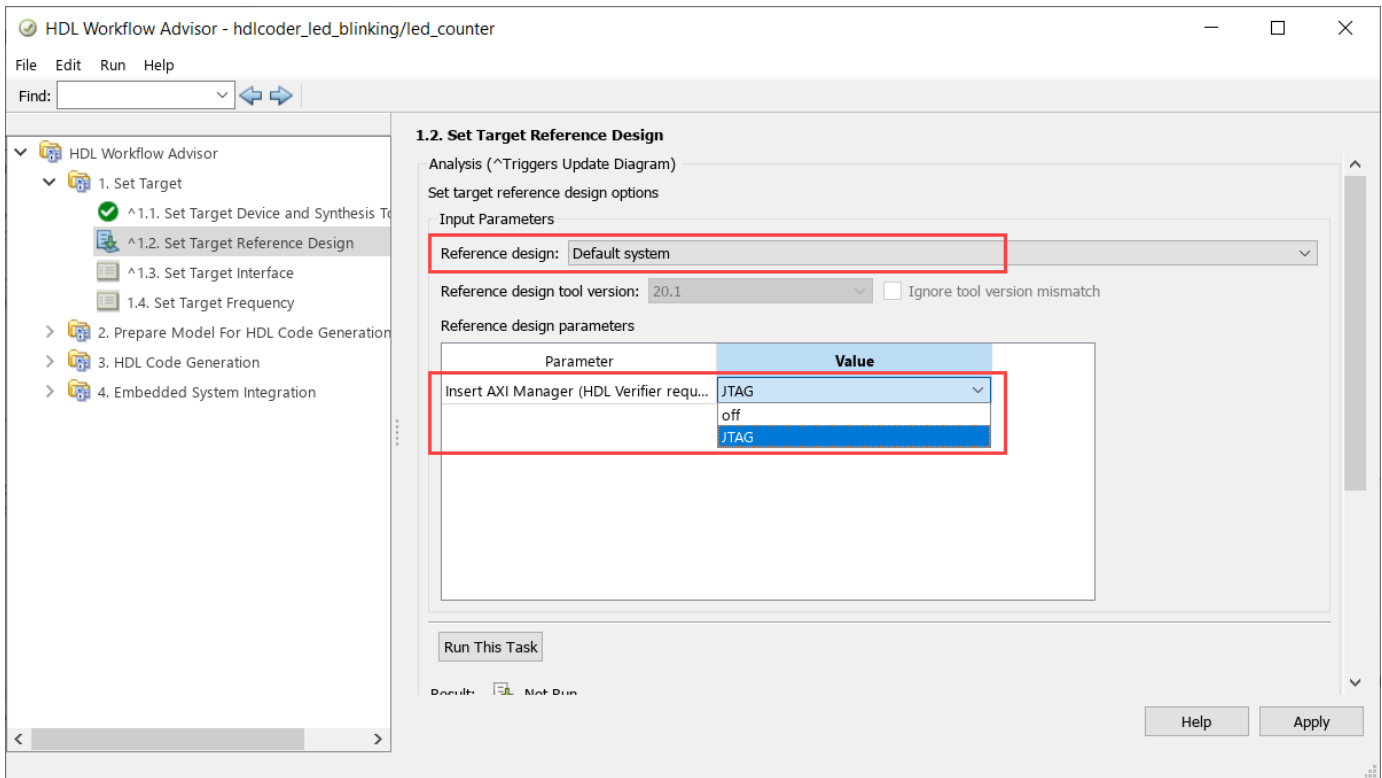
Copyright 2012 The MathWorks, Inc.

3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and selecting **HDL Code** followed by **HDL Workflow Advisor**.

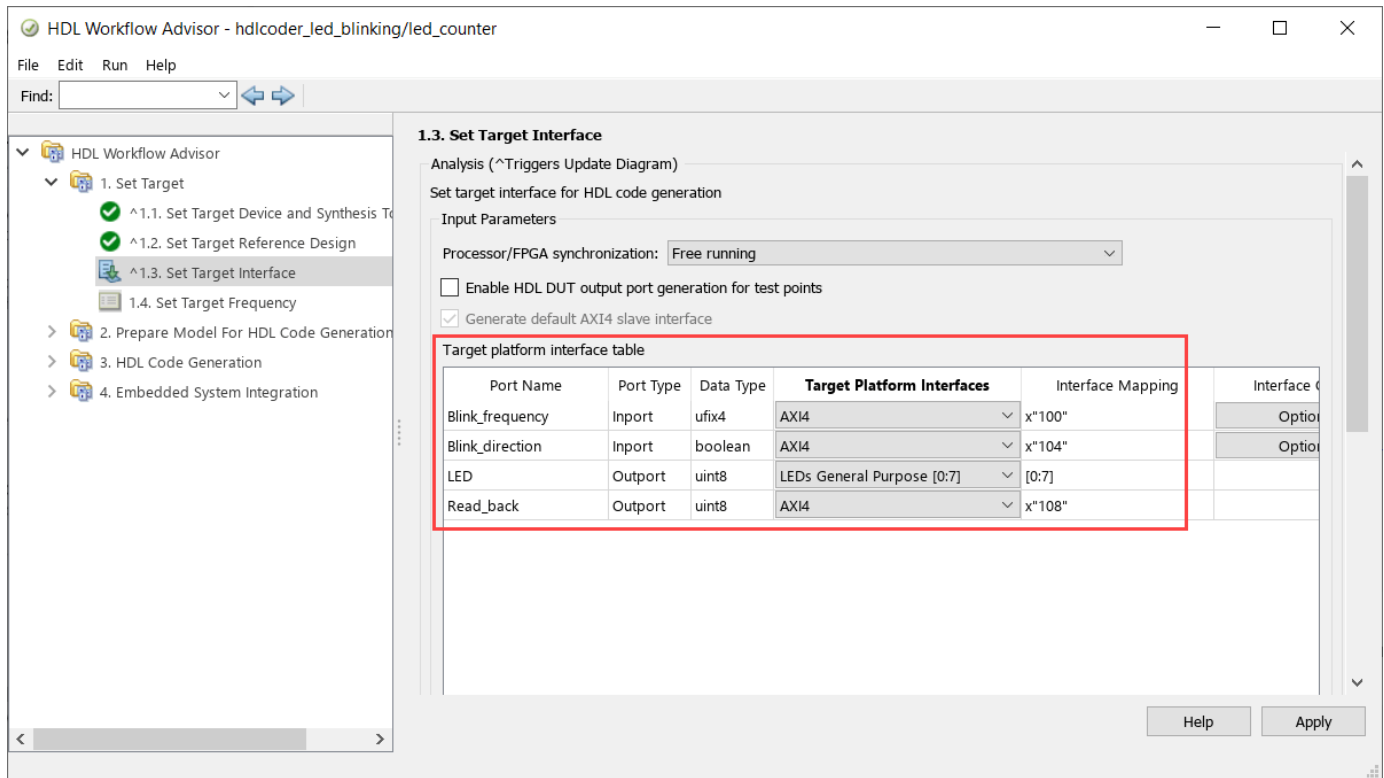
4. In step 1.1, set **Target workflow** to IP Core Generation and **Target platform** to Arrow DECA MAX 10 FPGA evaluation kit. Click **Run This Task**.

5. In step 1.2, set **Reference design** to Default system. Under **Reference design parameters**, set **Insert AXI Manager (HDL Verifier required)** to JTAG.

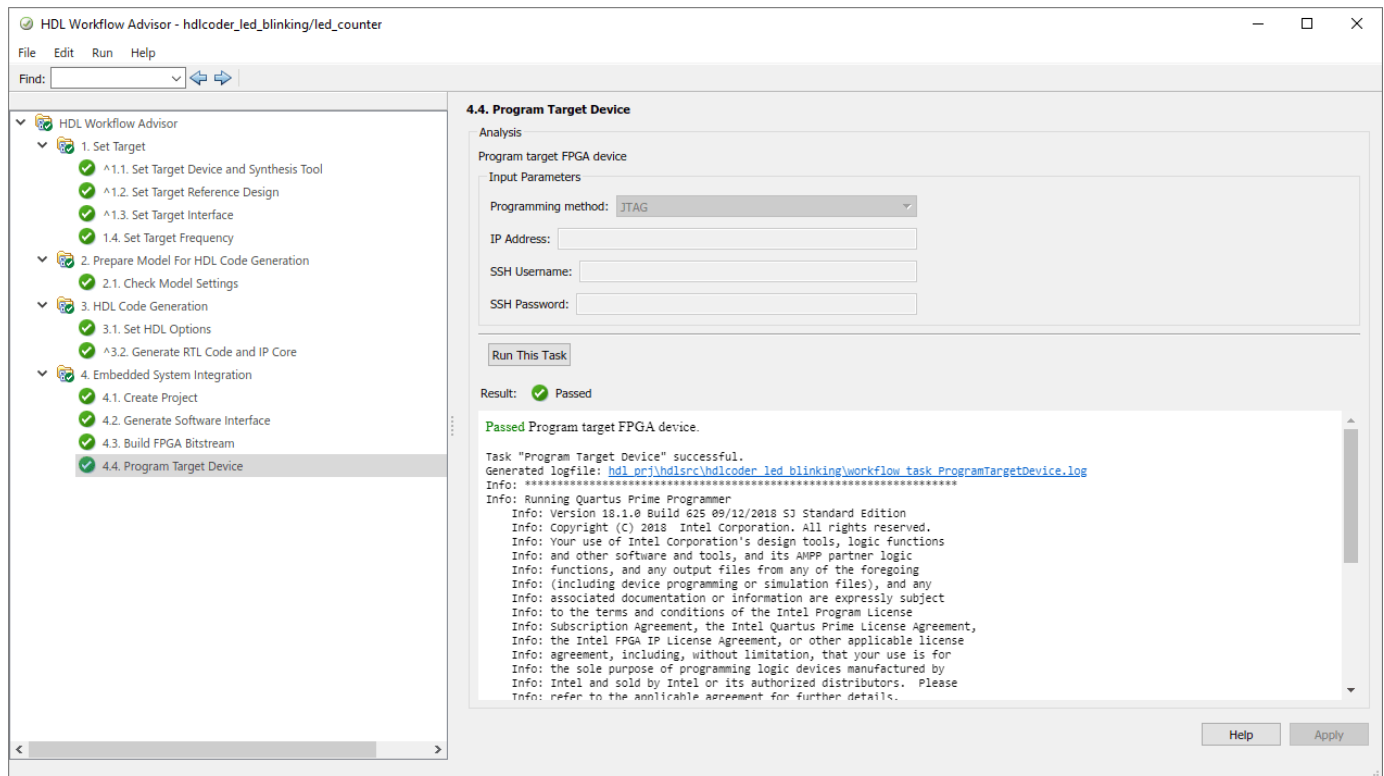




6. In step 1.3, set the interface of the **Blink\_frequency**, **Blink\_direction**, and **Read\_back** ports to AXI4. Set the interface of the **LED** port to LEDs General Purpose [0:7].



7. Run the remaining steps in the workflow to generate a bitstream and program the target device.



## Determine Addresses from IP Core Report

The base address for an HDL Coder IP core is defined as  $0x00000000$  for the Default System reference design, which uses the AXI Manager IP. You can see address setting in the generated IP core report as shown this figure.

**Use JTAG AXI Master to control the IP core from MATLAB**

In 1.2 Step "Set Target Reference design", "Insert JTAG AXI Manager" is turned "on". This adds Matlab as an "AXI Manager" to control the DUT IP core using AXI4 interface as shown.

**JTAG Interface**      **AXI4 Interface**

MATLAB      Reference Design (MATLAB JTAG AXI Master IP)      DUT IP Core

LE

Requires a HDL Verifier license to use this feature. After that use MATLAB® Command line interface to access the DUT IP core registers. **The Base Address of AXI4 Slave is  $0x0000\ 0000$  .**

An example JTAG AXI Master commands to access the DUT IP register is :

1. Create the AXI master object  
`h = aximanager('ToolName')` Here ToolName = xilinx/altera
2. Command to write into IP Core registers:  
`h.writememory('BaseAddress+AddressOffset', WriteValue)`

The IP core report register address mapping table shows the offsets.

The screenshot shows a window titled "Code Generation Report" with a search bar and "Match Case" option. The left sidebar contains a "Contents" list with "IP Core Generation Report" highlighted. The main content area is titled "Register Address Mapping" and contains the following text:

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address	Offset	Description
IPCore_Reset	0x0		write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4		enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8		contains unique IP timestamp (yymmddHHMM): 2201202309
Blink_frequency_Data	0x100		data register for Inport Blink_frequency
Blink_direction_Data	0x104		data register for Inport Blink_direction
Read_back_Data	0x108		data register for Output Read_back

Following are the AXI4 slave Base address and Master address space specified in the reference design:

**Default system.**  
 AXI4 Slave Base Address: **0x0000 0000**  
 AXI4 Slave Master connection: **AXI\_Manager\_0.axm\_m0**  
 Use the AXI4 Slave Base Address plus Address offset to access the IP Core registers shown in Register Address Mapping table

The AXI4 slave write register readback is OFF for the IP core.  
 The register address mapping is also in the following C header file for you to use when programming the processor:  
[include/led\\_count\\_ip\\_addr.h](#)  
 The IP core name is appended to the register names to avoid name conflicts.

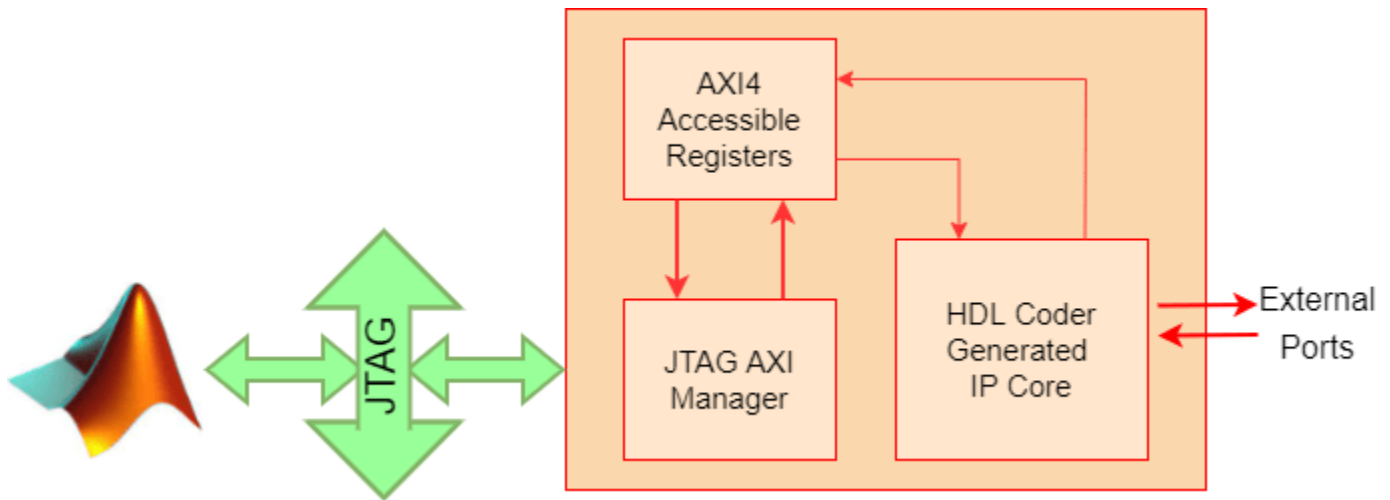
**IP Core User Guide**

**Theory of Operation**

At the bottom of the window are "OK" and "Help" buttons.

## HDL Verifier Command Line Interface

If you have the HDL Verifier support package for Intel FPGA boards and select the AXI Manager reference design, then you can use the MATLAB command line interface to access the IP core that is generated by the HDL Coder product.



To write and read from the DDR memory, follow these steps.

1. Create an AXI manager object.

```
h = aximanager('Altera')
```

2. Issue a write commands. For example, disable the DUT.

```
h.writememory('4',0)
```

3. Re-enable the DUT.

```
h.writememory('4',1)
```

4. Read the current counter value.

```
h.readmemory('108',1)
```

5. Delete the object to free up the JTAG resource. If you do not delete the object, other JTAG operations, such as programming the FPGA, fail.

```
delete(h)
```

### Intel JTAG to AXI Master Reference Design

Create a custom reference design to use the Intel JTAG to AXI Master IP in the reference design, and then add reference design files to the MATLAB path using the `addpath` command.

Access the HDL Coder IP core registers using the Intel JTAG to AXI Master IP by using the base address that is defined in reference design plugin file.

### Qsys System Console Tcl Commands for AXI Read and Write

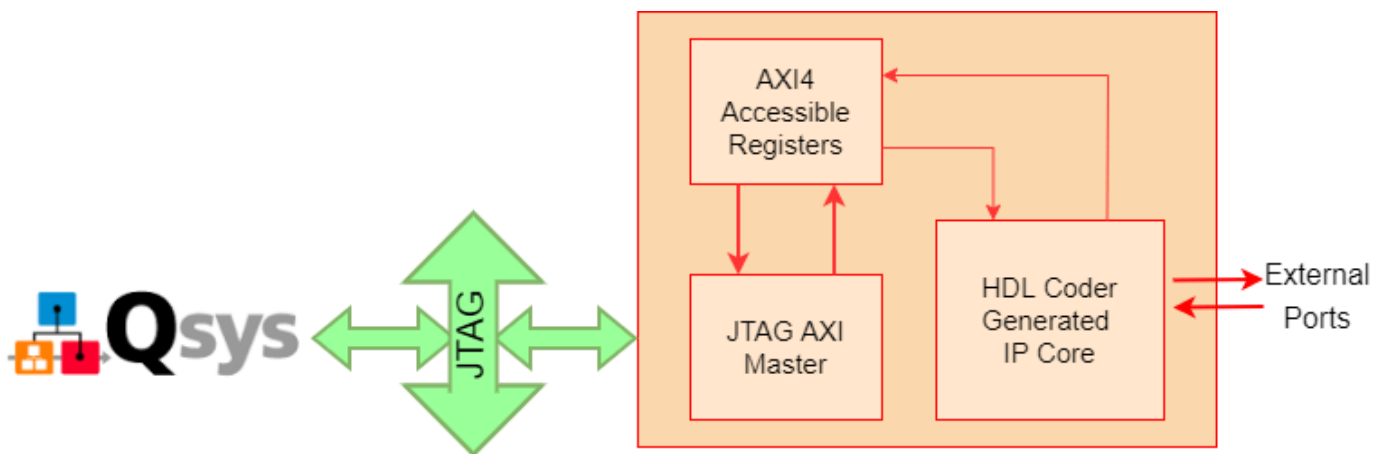
Before opening a system console, choose the appropriate Qsys read and write commands. For this example, because all of the HDL Coder generated IP core registers are currently 32 bits, use these read and write commands.

```
% master_write_32 <service-path> <start-address> <list-of-32-bit-values>
% master_read_32 <service-path> <start-address> <size-in-multiples-of-32-bits>
```

For example, write the 32 bit hex value 0x12345678 to the IP core register defined by offset 0x100 using a previously defined service path stored in the variable \$jtag.

```
% master_write_32 $jtag 0x100 0x12345678
```

Before you can generate reads and writes, you must first launch a system console and open a connection to the JTAG Master that issues the register reads and writes.



To open a connection to the JTAG Master, first set a variable that stores the service path (in this case, only one master exists).

```
% set jtag [lindex [get_service_paths master] 0]
```

Use the variable to open the JTAG Master in master mode.

```
% open_service master $jtag
```

Launch the Altera® system console and enter the commands to open the JTAG Master.

```
system('C:\intel\FPGA\17.1\quartus\sopc_builder\bin\system-console&')
```

```
Tcl Console
%
% set jtag [lindex [get_service_paths master] 0]
/devices/10M50DA(.|ES)|10M50DC@1#USB-1#Arrow MAX 10 DECA/(link)/JTAG/(110:132 v1 #0)/phy_0/master
% open_service master $jtag
% master_write_32 $jtag 0x04 0x00
% master_write_32 $jtag 0x04 0x01
% master_read_32 $jtag 0x108 1
0x000000f0
% close_service master $jtag
%
```

When you are done using the JTAG Master, close the connection by using this Tcl command.

```
close_service master $jtag
```

### **Summary**

You can use the JTAG AXI Manager IP to interface with HDL Coder IP core registers in systems that do not have an embedded ARM processor, such as the MAX 10. You can use this IP as a first step to debug standalone HDL Coder IP cores, prior to hand coding software for soft processors, (such as Nios II), or as a way to tune parameters on a running system.

### **See Also**

[axi\\_manager](#) | [writememory](#) | [readmemory](#)

### **More About**

- “Set Up AXI Manager” on page 3-2
- “Getting Started with the HDL Workflow Advisor” (HDL Coder)
- “Custom IP Core Generation” (HDL Coder)
- “Troubleshooting”

