DisplayPort MegaCore Function

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-01131-2.0

Document last updated for Altera Complete Design Suite version: 13.0
Document publication date: May 2013

Feedback Subscribe

Contents

Chapter 10. DisplayPort Sink Register Map and DCPD Locations

Additional Information

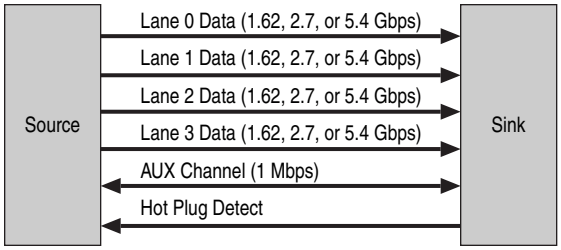
1. About This MegaCore Function

This document describes the Altera® DisplayPort MegaCore® function, which provides support for next-generation video display interface technology. The Video Electronics Standards Association (VESA) defines the DisplayPort standard as an open digital communications interface for use in internal connections such as:

- Interfaces within a PC or monitor
- External display connections, including interfaces between a PC and monitor or projector, between a PC and TV, or between a device such as a DVD player and TV display

The Altera DisplayPort source has a scalable main link with 1, 2, or 4 lanes for a total of 21.6 Gbps bandwidth. A bidirectional AUX channel with 1 Mbps Manchester encoding provides side-band communication. The sink uses a hot plug detect (HPD) signal to announce its presence, and the source uses the same signal to initiate link configuration. Refer to Figure 1–1.

Figure 1–1. DisplayPort Source and Sink Communication



The main link has three selectable data rates: 1.62, 2.7, and 5.4 Gbps.

Features

The DisplayPort MegaCore function has the following features:

- Conforms to the Video Electronics Standards Association (VESA) specification version 1.2a
- Scalable main data link
  - 1, 2, or 4 lane operation
  - 1.62, 2.7, and 5.4 Gbps per lane with an embedded clock
- Color support
  - 16, 18, 20, 24, 30, 32, 36, or 48 bits per pixel (bpp) color depths
  - RGB and YCrCb color modes

- Source
  - Embedded controller AUX channel operation
  - Accepts standard H-sync and V-sync RGB and YCrCb input video formats
  - Supports audio and video streams
- Sink
  - Finite state machine (FSM) and embedded controller AUX channel operation
  - Produces a proprietary video output
- Auxiliary channel for 2-way communication (link and device management)
- Hot plug detect (HPD)
  - Sink announces its presence
  - Sink requests the source's attention
- AC coupling and low EMI
- Avalon® Memory-Mapped (Avalon-MM) interfaces for run-time control input and connections to external memory blocks
- Easy-to-use parameter editor for parameterization and hardware generation
- IEEE encrypted simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore® Plus evaluation
- Qsys ready

Device Family Support

Table 1–1 defines the device support levels for Altera IP cores.

Table 1–1. Altera IP Core Device Support Levels

FPGA Device Families	HardCopy Device Families
Preliminary support—The IP core is verified with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.	**HardCopy Companion**—The IP core is verified with preliminary timing models for the HardCopy companion device. The IP core meets all functional requirements, but might still be undergoing timing analysis for the HardCopy device family. It can be used in production designs with caution.
Final support—The IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.	**HardCopy Compilation**—The IP core is verified with final timing models for the HardCopy device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 1–2 lists the level of support offered by the DisplayPort MegaCore function for each Altera device family.

Table 1–2. Device Family Support

Device Family	Support
Arria® V	Preliminary support
Arria V GZ	Preliminary support
Stratix® V	Preliminary support
Other device families	No support

IP Core Verification

Before releasing a publicly available version of the DisplayPort IP core, Altera runs a comprehensive verification suite in the current version of the Quartus® II software. These tests use standalone methods and the Qsys system integration tool to create the instance files. These files are tested in simulation and hardware to confirm functionality. Altera tests and verifies the DisplayPort MegaCore function in hardware for different platforms and environments.

The DisplayPort core has been tested at VESA Plugtest events and passes the Unigraf DisplayPort Link Layer CTS tests.

Design Examples

The IP core includes a hardware demonstration as well as simulation design examples for various device families. These examples provide a starting point for you to understand the Altera video design methodology quickly, enabling you to build full video processing systems on an FPGA.

For more information about the design examples, refer to:

- Chapter 5, DisplayPort MegaCore Function Hardware Demonstration
- Chapter 6, DisplayPort MegaCore Function Simulation Example
- Chapter 7, DisplayPort MegaCore Function Compilation Example

Performance and Resource Utilization

This section shows typical expected performance for the DisplayPort MegaCore function with the Quartus® II software targeting Arria V, Arria V GZ, and Stratix V devices. Refer to Table 1–3.

Table 1–3. DisplayPort Performance, Duplex Mode, 4 Lanes, 24 BPP Color Depth

Device Family	Combinational ALUTs	Logic Registers	Memory		
			Bits	M10K	M20K
Arria V (1)	6,557	6,493	71 K	12	–
Arria V GZ (2)	6,666	6,700	71 K	–	12
Stratix V (3)	6,666	6,700	71 K	–	12

Notes to Table 1–3:

(1) EP5AGXFB3H4F40C5ES device.
(2) EP5AGZME12H29C3 device.
(3) EP5SGXEA7K2F40C2 device.

Release Information

Table 1–4 provides information about this release of the DisplayPort MegaCore function.

Table 1–4. Display Port Release Information

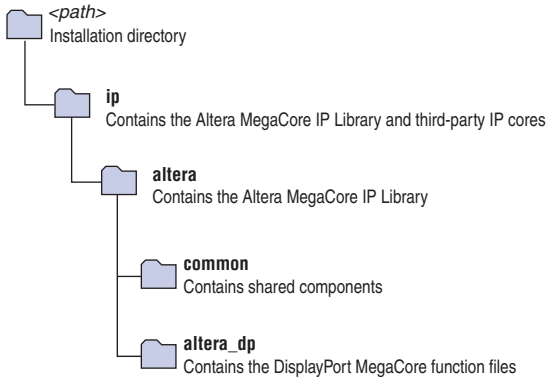
Item	Description
Version	13.0
Release Date	May 2013
Ordering Code	IP-DP-v1.1a
Product ID	0109
Vendor ID(s)	6AF7

Installation and Licensing

The DisplayPort IP core is part of the Altera MegaCore IP Library, which is distributed with the Quartus II software and downloadable from the Altera web site, www.altera.com.

Figure 1–2 shows the directory structure after you install the DisplayPort IP core, where <*path*> is the installation directory. The default installation directory on Windows is **C:\altera\<***version number***>**; on Linux it is **/opt/altera<***version number***>.**

Figure 1–2. IP core Directory Structure



You can use Altera's free OpenCore Plus evaluation feature to evaluate the IP core in simulation and in hardware before you purchase a license. You must purchase a license for the IP core only when you are satisfied with its functionality and performance, and you want to take your design to production.

After you purchase a license for the DisplayPort IP core, you can request a license file from the Altera website at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

OpenCore Plus Evaluation

With the Altera free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera IP core or AMPP^{SM} megafunction) in your system using the Quartus II software and Altera-supported Verilog HDL simulators.
- Verify the functionality of your design and evaluate its size and speed quickly and easily.
- Generate time-limited device programming files for designs that include IP cores.
- Program a device and verify your design in hardware.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation supports the following two operation modes:

- *Untethered*—the design runs for a limited time.
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.

For Altera IP cores, the untethered time-out is 1 hour; the tethered time-out value is indefinite.

After the hardware evaluation time expires, the DisplayPort IP core behaves as if its reset signal were held asserted, and your design stops working.

For Information About	Refer To
Installation and licensing	Altera Software Installation and Licensing
Open Core Plus	*AN 320: OpenCore Plus Evaluation of Megafunctions*

This chapter provides an overview of the DisplayPort design flow. The IP core is installed as part of the Quartus II installation process.

Design Flows

You can customize the DisplayPort IP core to support a wide variety of applications. You can instantiate this IP core in the MegaWizard Plug-In Manager or in the Qsys system integration tool.

The MegaWizard Plug-In Manager flow offers the following advantages:

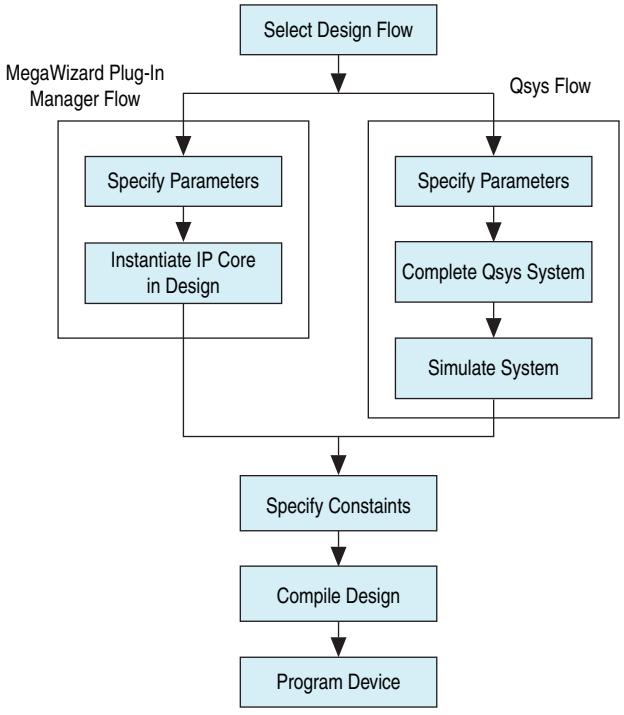
- Allows you to parameterize the IP core to create a variation that you can instantiate manually in your design.

The Qsys flow offers the following advantages:

- Allows you to integrate other Altera-provided custom components with the IP core easily in your design.
- Provides visualization of hierarchical designs.
- Automatically generates interconnect fabric and inserts adapters.

Figure 2–1 shows the stages for creating a system with the DisplayPort IP core and the Quartus II software. Each stage is described in detail in subsequent sections.

Figure 2–1. DisplayPort IP Core Design Flow



MegaWizard Plug-In Manager Design Flow

You use the MegaWizard Plug-In Manager in the Quartus II software to parameterize a custom IP core variation. When you select the DisplayPort IP core in the MegaWizard Plug-In Manager, the DisplayPort parameter editor appears. The DisplayPort parameter editor lets you interactively set parameter values and select optional ports. This flow is best for manual instantiation of an IP core in your design. The following sections describe this design flow.

Specifying Parameters

To specify DisplayPort IP core parameters using the MegaWizard Plug-In Manager, follow these steps:

1. Create a Quartus II project using the **New Project Wizard** available from the File menu.

   Make sure to select a device family that supports the IP core. Refer to Table 1–2 on page 1–3 for device support information.

2. On the Tools menu, click **MegaWizard Plug-In Manager**.
3. Follow the prompts in the MegaWizard Plug-In Manager interface to create a custom megafunction variation.
4. Under **Installed Plug-Ins**, click **Interfaces > DisplayPort > DisplayPort v**<*version*>.
5. Select the output file type and name.
6. click **Next**. The DisplayPort parameter editor appears.
7. Specify the parameters in the DisplayPort parameter editor. For details about these parameters, refer to "Source Parameters" on page 3–5 and "Sink Parameters" on page 4–4.
8. Click **Finish** to generate the IP core and supporting files, including simulation models.

   You may have to wait several minutes for file generation to complete.

9. Click **Exit** when file generation completes.
10. If you generate the DisplayPort IP core instance in a Quartus II project, you are prompted to add the Quartus II IP File (**.qip**) to the current Quartus II project. You can also turn on **Automatically add Quartus II IP Files to all projects**.

    The **.qip** is generated by the parameter editor, and contains information about the generated IP core. In most cases, the **.qip** contains all of the necessary assignments and information required to process the IP core or system in the Quartus II compiler. The MegaWizard Plug-In Manager generates a single **.qip** for each IP core.

11. Click **Exit** to close the MegaWizard Plug-In Manager.

You can now integrate your custom IP core variation in your design, simulate, and compile.

You must add a dynamic reconfiguration block (Transceiver Reconfiguration Controller) to your design and connect it to the DisplayPort IP core PHY IP reconfiguration signals. The design compiles without the Transceiver Reconfiguration Controller, but it cannot function correctly in hardware.

An informational message in the DisplayPort parameter editor tells you the number of reconfiguration interfaces you must configure in your dynamic reconfiguration block.

Refer to Chapter 5, DisplayPort MegaCore Function Hardware Demonstration for an example design implementing a DisplayPort source and sink.

Simulating the Design

You can simulate your DisplayPort IP core variation using the simulation model that the MegaWizard Plug-In Manager generates. The simulation model files are generated in vendor-specific subdirectories of your project directory.

The DisplayPort IP core includes a simulation example. Refer to Chapter 6, DisplayPort MegaCore Function Simulation Example for more information.

The following sections teach you how to simulate your MegaWizard Plug-In Manager flow generated DisplayPort IP core variation with the generated simulation model.

Simulating with the ModelSim Simulator

To simulate using the Mentor Graphics ModelSim simulator, perform the following steps:

1. Start the ModelSim simulator.
2. In ModelSim, change directory to the project simulation directory *<variation>*_**sim/mentor**.
3. Type the following commands to set up the required libraries and compile the generated simulation model:

```
do msim_setup.tcl
ld
run -all
```

Simulating with the VCS Simulator

To simulate using the Synopsys VCS simulator, type the following commands:

```
cd <variation>_sim/synopsys/vcs
sh vcs_setup.sh
./simv
```

For Information About	Refer To
Quartus II software	See the Quartus II Help topics:
MegaWizard Plug-In Manager	"About the Quartus II Software" "About the MegaWizard Plug-In Manager"
Altera simulation models	*Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook*

Qsys Design Flow

The Qsys design flow enables you to integrate a DisplayPort IP core in a Qsys system. The Qsys design flow allows you to connect component interfaces with the system interconnect, eliminating the requirement to design low-level interfaces and significantly reducing design time. When you add a DisplayPort IP core instance to your design, the DisplayPort parameter editor guides you in selecting the properties of the DisplayPort IP core instance.

You can use Qsys to build a system that contains your customized DisplayPort IP core. You can easily add other components and quickly create a Qsys system. Qsys can automatically generate HDL files that include all of the specified components and interconnections. The HDL files are ready to be compiled by the Quartus II software to produce output files for programming an Altera device.

Specifying Parameters

To specify DisplayPort parameters using the Qsys flow, follow these steps:

1. Create a new Quartus II project using the **New Project Wizard** available from the File menu.
2. On the Tools menu, click **Qsys**.
3. On the **System Contents** tab, in the **Component Library** pane, select **DisplayPort** and click **Add**. The DisplayPort parameter editor appears.

   ☞ You can find **DisplayPort** by expanding **Library > Interface Protocols > DisplayPort**.

4. Specify the required parameters on all tabs of the DisplayPort parameter editor. For detailed explanations of these parameters, refer to Chapter 3, Parameter Settings.
5. Click **Finish** to complete the DisplayPort IP core instance and add it to the Qsys system.

Completing the Qsys System

To complete the Qsys system, follow these steps:

1. Add and parameterize any additional components.
2. Connect the components using the Connection panel on the **System Contents** tab.
3. If some signals are not displayed, click the Filter icon to display the **Filters** dialog box. In the **Filter** list, click **All Interfaces**. Alternatively, if you right-click in the System Contents tab, a **Filter** menu option appears.

You must add a dynamic reconfiguration block (Transceiver Reconfiguration Controller) to your design and connect it to the DisplayPort IP core PHY IP reconfiguration signals. The design compiles without the Transceiver Reconfiguration Controller, but it cannot function correctly in hardware.

An informational message in the DisplayPort parameter editor tells you the number of reconfiguration interfaces you must configure in your dynamic reconfiguration block.

4. If you intend to simulate your Qsys system, on the **Generation** tab, turn on **Create simulation model** and select **Verilog HDL** or **VHDL** to generate a functional simulation model.
5. Click **Generate** to generate the system. Qsys generates the system and produces the <*system_name*>**.qip** file that contains the assignments and information required to process the IP core or system in the Quartus II Compiler.
6. In the Quartus II software, on the Project menu, click **Add/Remove Files in Project**.
7. In the **Settings** dialog box, under **Category**, highlight **Files**.
8. Browse to the **.qip** file and add it to your project.

Refer to Chapter 5, DisplayPort MegaCore Function Hardware Demonstration for an example design implementing a DisplayPort source and sink.

Simulating the System

During system generation, Qsys optionally generates a DisplayPort functional simulation model in the HDL you specify.

For information about simulating Qsys systems, refer to the *Creating a System with Qsys* chapter in volume 1 of the *Quartus II Handbook*.

Compiling the Full Design and Programming the FPGA

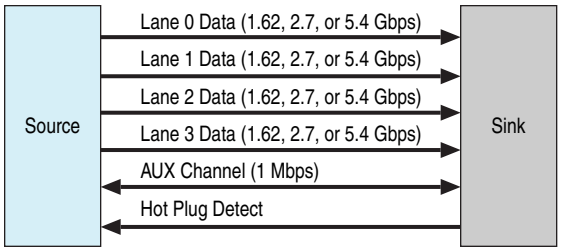
You can use the **Start Compilation** command on the Processing menu in the Quartus II software to compile your design. After successfully compiling your design, program the targeted Altera device with the Programmer and verify the design in hardware.

For Information About	Refer To
Compiling your design	*Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
Programming the device	Device Programming section in volume 3 of the *Quartus II Handbook*

Source Overview

The DisplayPort source has a scalable main link with 1, 2, or 4 lanes for a total of 21.6 Gbps bandwidth. A bidirectional AUX channel with 1 Mbps Manchester encoding provides side-band communication. Refer to Figure 3–1.

Figure 3–1. DisplayPort Source



The main link has three selectable data rates: 1.62, 2.7, and 5.4 Gbps. The source device sets the lane count and link rate combination (referred to as the policy) according to the sink's capabilities and required video bandwidth. The IP core transmits the video and audio streams on the main link with embedded clocking. The DisplayPort protocol embeds the clocks such that the pixel and audio clocks are decoupled from the transmission clock.

The IP core transmits data in a scrambled ANSI 8B/10B format. The data transmission includes redundancy for error detection. The secondary data stream, such as an audio stream, uses a Reed-Solomon encoder for error correction.

The AUX channel is an AC-coupled differential pair for bidirectional communication. The signaling is a self-clocked Manchester encoding at 1 Mbps. As in the 100-T Ethernet protocol, the encoder uses a preceding synchronization pattern in each 16-byte maximum packet. The AUX channel uses a master-slave hierarchy in which the source (master) initiates all communication.

Source Functional Description

The DisplayPort source has a complete set of parameters for optimizing device resources. The DisplayPort source consists of a DisplayPort encoder block, a transceiver management block, and a controller interface block with an Avalon-MM interface for connecting with an embedded controller such as a Nios II processor. Figure 3–2 shows the top-level design. You configure the ports using an RTL wrapper instantiation or by implementing the IP core as a Qsys component

Figure 3–2. DisplayPort Source Top-Level Block Diagram

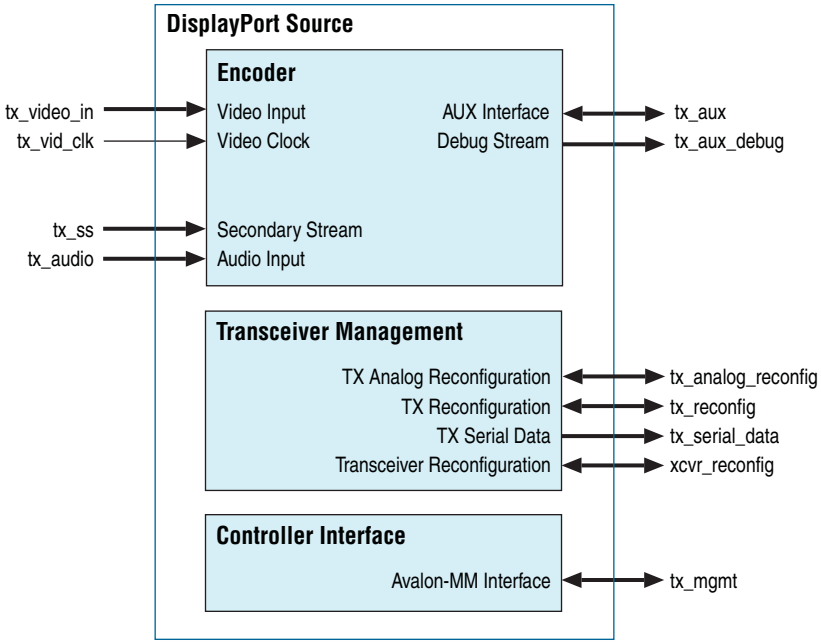
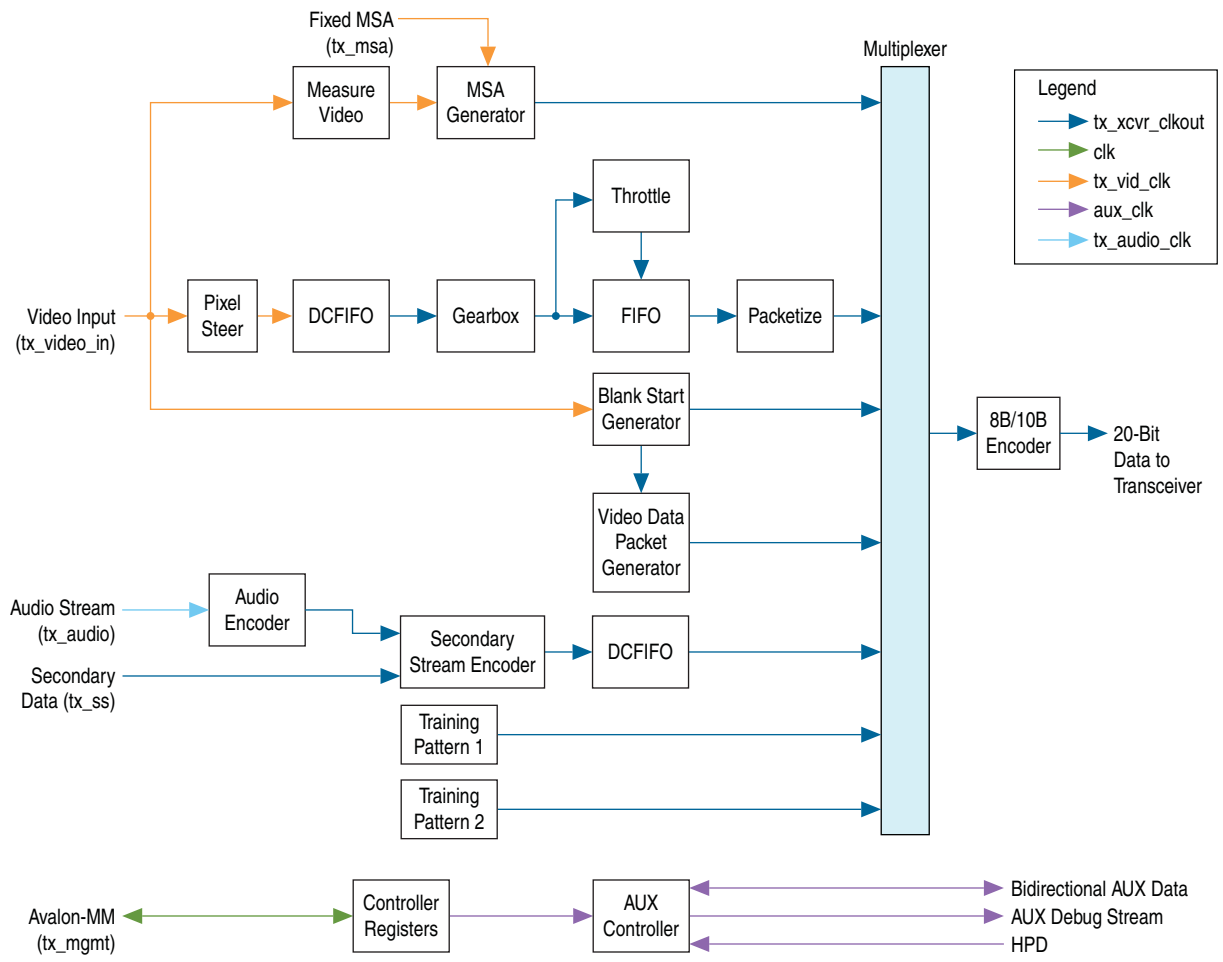


Figure 3–3 shows the DisplayPort source functional block diagram.

Figure 3–3. DisplayPort Source Block Diagram



The source accepts a standard H-sync and V-sync video stream for encoding. The IP core latches and processes the video data before processing it using the `tx_vid_clk` input. The video horizontal synchronization data width supports 6 to 16 bits-per color (bpc), and is user selectable. The IP core forwards video input to three parallel paths.

Main Data Path

The main data path consists of the packetizer, measurement, and blank generator paths. The IP core multiplexes data from these three paths and outputs it through an 8B/10B encoder.

Packetizer Path

The packetizer path provides video data resampling and packetization, and consists of the following steps:

1. The pixel steer block decimates the data to the requested lane count (1, 2, or 4).
2. The DCFIFO crosses the data into the main link clock domain (`tx_xcvr_clkout`, generated by the transceiver), which can be 270, 135, or 81 MHz depending on the actual main link rate requested.
3. The gearbox resamples the video data according to the specified color depth. You can optimize the gearbox by implementing fewer color depths. For example, you can reduce the resources required to implement the system by supporting only the color depths you need instead of the complete set of color depths specified in the DisplayPort specification.

   ☞ A minimal DisplayPort system should support both 6 and 8 bpc. Additionally, current LCD monitors rarely support more than 10 bpc. The VESA DisplayPort specification requires support for a mandatory VGA failsafe mode (640 x 480 at 6 bpc).

4. The IP core packetizes the re-sampled data. The DisplayPort specification requires data to be sent in a transfer unit (TU), which can be 32 to 64 link symbols long. To reduce complexity, the DisplayPort source uses a fixed 64-symbol TU. The specification also requires that the video data be evenly distributed within the TUs composing a full active video line. A throttle function distributes the data and regulates it such that the TUs leaving the IP core are evenly packed.

   The packetizer punctuates the outgoing 16-bit data stream with the correct packet comma codes. Internally, it uses a symbol and a TU counter to ensure that it respects the TU boundaries.

Measurement Path

The measurement path determines the video geometry required for the DisplayPort main stream attributes (MSA), which are sent once every vertical blanking interval. Optionally, the IP core can “import” a fixed MSA data parameter from an external port, removing the measurement logic. This feature is useful for embedded systems that only use known resolutions and synchronous pixel clocks.

Blank Generator Path

The blank generator path determines when to send the blank start comma codes with their corresponding video data packets.

Multiplexer

The IP core multiplexes the packetized data, MSA data, and blank generator data into a single stream. The combined data goes through 8B/10B encoding and is available as a 20–bit double-rate DisplayPort encoded video port. The 20-bit port interfaces directly to the Arria V or Stratix V high-speed output transceiver.

During training periods, the source can send the DisplayPort clock recovery and symbol lock test patterns (training pattern 1 and training pattern 2, respectively).

The source also implements an AUX channel controller, which you access using an embedded controller. The embedded controller acts as an Avalon-MM master and sends read/write commands to the Avalon-MM slave interface. The IP core clocks the AUX channel using a 16 MHz clock input (`aux_clk`). Refer to "Controller Interface" on page 3–8 for more details.

Source Parameters

You set parameters for the source using the DisplayPort parameter editor. Table 3–1 lists the DisplayPort source parameters.

Table 3–1. Source Parameters

Parameter	Description
Device family	Targeted device family (Arria V, Arria V GZ, or Stratix V); matches the project device family.
Support DisplayPort source	Enable DisplayPort source.
Maximum video input color depth	Video input interface port `bits per color` (bpc). Determines top-level video input port width (e.g, 6 bpc = 18 bits, 16 bpc = 48 bits).
TX maximum link rate	Maximum link rate. 20 = 5.4 Gbps, 10 = 2.7 Gbps, 6 = 1.62 Gbps.
Maximum lane count	Maximum lanes used (1, 2, or 4).
Enable AUX debug stream	Send source AUX traffic output to an Avalon-ST port.
Import fixed MSA	Used fixed MSA.
Interlaced input video	Interlace the input video. Turn on for interlaced, turn off for progressive.
Support secondary data channel	Enables secondary data.
Support audio data channel	Enables audio packet encoding.
Number of audio data channels	Number of audio channels supported.
Support CTS test automation	Support CTS test automation.
6-bpc RGB or YCbCr 4:4:4 (18 bpp)	Support 18 bpp decoding.
8-bpc RGB or YCbCr 4:4:4 (24 bpp)	Support 24 bpp decoding.
10-bpc RGB or YCbCr 4:4:4 (30 bpp)	Support 30 bpp decoding.
12-bpc RGB or YCbCr 4:4:4 (36 bpp)	Support 36 bpp decoding.
16-bpc RGB or YCbCr 4:4:4 (48 bpp)	Support 48 bpp decoding.
8-bpc YCbCr 4:2:2 (16 bpp)	Support 16 bpp decoding.
10-bpc YCbCr 4:2:2 (20 bpp)	Support 20 bpp decoding.
12-bpc YCbCr 4:2:2 (24 bpp)	Support 24 bpp decoding.
16-bpc YCbCr 4:2:2 (32 bpp)	Support 32 bpp decoding.
Invert transceiver polarity	Invert transceiver polarity.
Scrambler seed value	Initial seed for scrambler block. Use 16'hFFFF for normal DP and 16'hFFFE for eDP.
Support analog reconfiguration	Enable the analog reconfiguration interface if you are not using an external re-driver solution.

Source Interfaces

Table 3–2, Table 3–3, Table 3–4, Table 3–5, and Table 3–6 list the source's port interfaces. Your instantiation contains only the interfaces that you have enabled. The following sections provide details on these interfaces.

Table 3–2. Controller Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
clk	Clock	N/A	N/A	Clock for embedded controller.	clk
reset	Reset	clk	N/A	Reset for embedded controller.	reset
tx_mgmt	AV-MM	clk	reset	Avalon-MM interface for embedded controller.	tx_mgmt_address[8:0]
					tx_mgmt_chipselect
					tx_mgmt_read
					tx_mgmt_write
					tx_mgmt_writedata[31:0]
					tx_mgmt_readdata[31:0]
					tx_mgmt_waitrequest
tx_mgmt_irq	IRQ	clk	reset	IRQ for embedded controller.	tx_mgmt_irq

Table 3–3. Transceiver Management Interface (Part 1 of 2)

Interface	Port Type	Clock Domain	Reset	Description	Port (1)
xcvr_mgmt_clk	Clock	N/A	N/A	Transceiver management clock.	xcvr_mgmt_clk
xcvr_refclk	Conduit	N/A	N/A	Transceiver reference clocks.	xcvr_refclk[1:0]
tx_serial_data	Conduit	tx_xcvr_clkout	N/A	Transceiver serial data out.	tx_serial_data[*n*-1:0]
tx_analog_reconfig	Conduit	xcvr_mgmt_clk	reset	Transceiver analog reconfiguration handshaking.	tx_vod[2*n* - 1:0]
					tx_emp[2*n* - 1:0]
					tx_reconfig_req
					tx_reconfig_ack
					tx_reconfig_busy
tx_reconfig	Conduit	xcvr_mgmt_clk	reset	Transceiver link rate reconfiguration handshaking.	tx_link_rate (2)
					tx_reconfig_req
					tx_reconfig_ack
					tx_reconfig_busy

Table 3–3. Transceiver Management Interface (Part 2 of 2)

Interface	Port Type	Clock Domain	Reset	Description	Port (1)
xcvr_reconfig	Conduit	xcvr_mgmt_clk	N/A	Transceiver reconfiguration.	reconfig_to_xcvr[(2*n* + *m*) × 70 - 1:0]
					reconfig_from_xcvr[(2*n* + *m*) × 46 - 1:0]

Note:

(1) *n* is the number of TX lanes and *m* is the number of RX lanes.

(2) The tx_link_rate signal is 2 bits wide for 5.4 Gbps support.

Table 3–4. Video Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
tx_vid_clk	Clock	N/A	N/A	Video clock.	tx_vid_clk
tx_video_in	Conduit	tx_vid_clk	reset	Standard H/V synchronization video port input.	tx_vid_data[3*v* - 1:0] (1)
					tx_vid_v_sync
					tx_vid_h_sync
					tx_vid_f
					tx_vid_de

Note:

(1) *v* is the number of bits per color.

Table 3–5. AUX Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
aux_clk	Clock	N/A	N/A	AUX channel clock.	aux_clk
aux_reset	Reset	aux_clk	N/A	AUX channel reset.	aux_reset
tx_aux	Conduit	aux_clk	aux_reset	AUX channel interface.	tx_aux_in
					tx_aux_out
					tx_aux_oe
					tx_hpd
tx_aux_debug	AV-ST	aux_clk	aux_reset	Avalon-ST stream of AUX data for debugging.	tx_aux_debug_data[31:0]
					tx_aux_debug_valid
					tx_aux_debug_sop
					tx_aux_debug_eop
					tx_aux_debug_err
					tx_aux_debug_cha

Table 3–6. Secondary Interface

Interface	Signal Type	Clock Domain	Reset	Description	Port
tx_xcvr_clkout	Clock	N/A	N/A	Clock.	tx_xcvr_clkout
MSA (tx_msa)	Conduit	tx_xcvr_clkout	N/A	Input port for fixed MSA parameters.	tx_msa[191:0]
Secondary Stream (tx_ss)	Clock	N/A	N/A	Clock for secondary stream.	tx_xcvr_clkout
	AV-ST	tx_xcvr_clkout	N/A	Secondary stream interface.	tx_ss_data[31:0]
					tx_ss_valid
					tx_ss_ready
					tx_ss_sop
					tx_ss_eop
Audio (tx_audio)	Clock	N/A	N/A	Audio clock.	tx_audio_clk
	Conduit	tx_audio_clk	N/A	Audio sample data interface.	tx_audio_lpcm_data[n*31-1:0]
					tx_audio_valid
					tx_audio_mute

Controller Interface

The controller interface allows you to control the source from an external or on-chip controller, such as the Nios II processor. The controller can control the DisplayPort link parameters and the AUX channel controller.

The AUX channel controller interface works with a simple serial-port-type peripheral that operates in a polled mode. Because the DisplayPort AUX protocol is a master-slave interface, the DisplayPort source (the master) starts a transaction by sending a request and then waits for a reply from the attached sink.

The controller interface includes a single interrupt source. The interrupt notifies the controller of an HPD signal state change. Your system can interrogate the DP_TX_STATUS register to determine the cause of the interrupt. Writing to the DP_TX_STATUS register clears the pending interrupt event.

For a detailed description of the source register map, refer to Chapter 9, DisplayPort Source Register Map.

AUX Interface

The IP core has three ports that control the serial data across the AUX channel:

- Data input (tx_aux_in)
- Data output (tx_aux_out)
- Output enable (tx_aux_oe). The output enable port controls the direction of data across the bidirectional link.

These ports are clocked by the source's 16 MHz clock (aux_clk). The AUX channel's physical layer is a bidirectional 2.5 V SSTL Class II interface.

Refer to *AN 522: Implementing Bus LVDS Interface in Supported Altera Device Families* for more information.

The source's AUX controller allows you to capture all bytes sent from and received by the AUX channel, which is useful for debugging. The IP core provides a standard stream interface that you can use to drive an Avalon-ST FIFO component directly. Table 3–7 describes the debugging ports.

Table 3–7. Source AUX Debug Interface Ports

Port	Comments
tx_aux_debug_data[31:0]	The source AUX debug interface inserts a 1 µs timestamp counter in bits [31:8]; bits [7:0] represent the byte received or transmitted.
tx_aux_debug_valid	Qualifies valid stream data.
tx_aux_debug_sop	Indicates the message packet's first byte.
tx_aux_debug_eop	Indicates the message packet's last byte. The last byte should be ignored and is not part of the message.
tx_aux_debug_err	Indicates if the IP core detects an error in the current byte.
tx_aux_debug_cha	Indicates the direction of the current byte. 1 = byte transmitted by the source, 0 = byte received from the sink.

Video Interface

The core inputs video to be encoded via the tx_video_in interface, which provides a standard H-sync and V-sync input with support for interlaced or progressive video. You specify the data input width via a parameter. The same input port transfers RGB and YCrCb data in either 4:4:4 or 4:2:2 color format. Data is most-significant bit aligned and formatted for 4:4:4 as shown in Figure 3–4.

Figure 3–4. Video Input Data Format (18 bpp to 48 bpp Port Width)

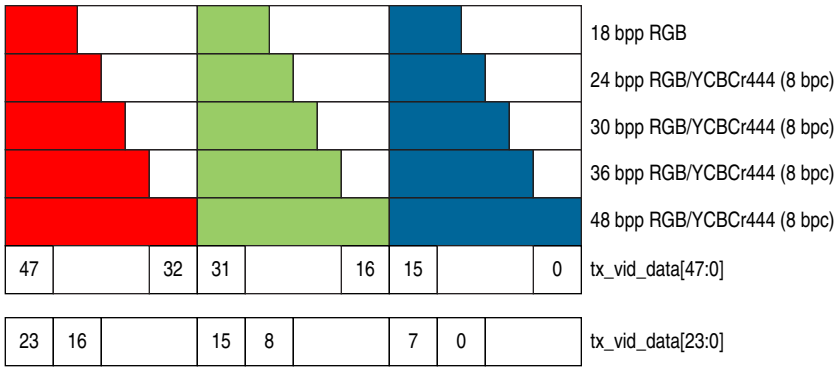
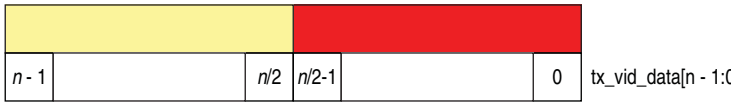


Figure 3–5 shows the sub-sampled 4:2:2 color format for a video port width of *n*. The most-significant half of the video port always transfers the Y component while the least-significant half of the video port transfers the alternate Cr or Cb component. If the Y/Cb/Cr component widths are less than *n*/2, they must be most-significant bit aligned with respect to the *n* and *n*/2-1 boundaries.

Figure 3–5. Sub-Sampled 4:2:2 Color Format Video Port



Transceiver Management Interface

The 20-bit transceiver management interface consists of a native PHY block on the RX and TX with a CMU PLL. The design uses a soft 8B/10B encoder. The transceiver can be reconfigured to use one of two reference clocks:

- 162 MHz clock for reduced bit rate (RBR)
- 270 MHz clock for high bit rate (HBR or HBR2).

You use the Transceiver Reconfiguration Controller to switch between the two reference clocks. To switch them, you reconfigure the logical reference clock source for the TX CMU PLL. The IP core sets the `tx_link_rate` to:

- 00 (RBR)
- 01 (HBR)
- 10 (HBR2)

When the core makes a request, the `tx_reconfig_req` port goes high. The user logic asserts `tx_reconfig_ack` and then reconfigures the transceiver. During reconfiguration, the user logic holds `tx_reconfig_busy` high; the user logic drives it low when reconfiguration completes.

The transceiver requires a reconfiguration controller.

- Refer to the *Altera Transceiver PHY IP Core User Guide* for more information about how to reconfigure the transceiver.
- Refer to *AN 645: Dynamic Reconfiguration of PMA Controls in Stratix V Devices* for more information about using the Transceiver Reconfiguration Controller to reconfigure the Stratix V Physical Media Attachment (PMA) controls dynamically.
- Refer to *AN 645: Using the Transceiver Reconfiguration Controller for Dynamic Reconfiguration in Arria V and Cyclone V Devices* for more information about using the Transceiver Reconfiguration Controller to reconfigure the Arria V Physical Media Attachment (PMA) controls dynamically.
- Refer to *AN 678: High-Speed Link Tuning Using Signal Conditioning Circuitry in Stratix V Transceivers* for information about link tuning.

Transceiver Analog Reconfiguration Interface

The tx_analog_reconfig interface uses the `tx_vod` and `tx_emp` transceiver management control ports. You must map these ports for the device you are using. To change these values, the core drives `tx_analog_reconfig_req` high. Then, the user logic sets `tx_analog_reconfig_ack` high to acknowledge and drives `tx_analog_reconfig_busy` high during reconfiguration. When reconfiguration completes, the user logic drives `tx_analog_reconfig_busy` low.

Secondary Stream Interface

You can transmit the secondary stream data over the DisplayPort main link via the secondary stream (tx_ss) interface. This interface uses handshaking and back pressure to control packet delivery. Internally, the core uses a FIFO to store packets until a slot becomes available on the main link. If the FIFO fills up, the secondary stream interface stops accepting packets and applies back pressure. The packet must be available at the time of sending because the ss_tx port does not support forward pressure.

The tx_ss interface input data format corresponds to four, 15-nibble code words as specified by the DisplayPort version 1.1a specification section 2.2.6.3. These 15-nibble code words are supplied by the upstream Reed-Solomon encoder. The format differs for header and payload as shown in Figure 3–6.

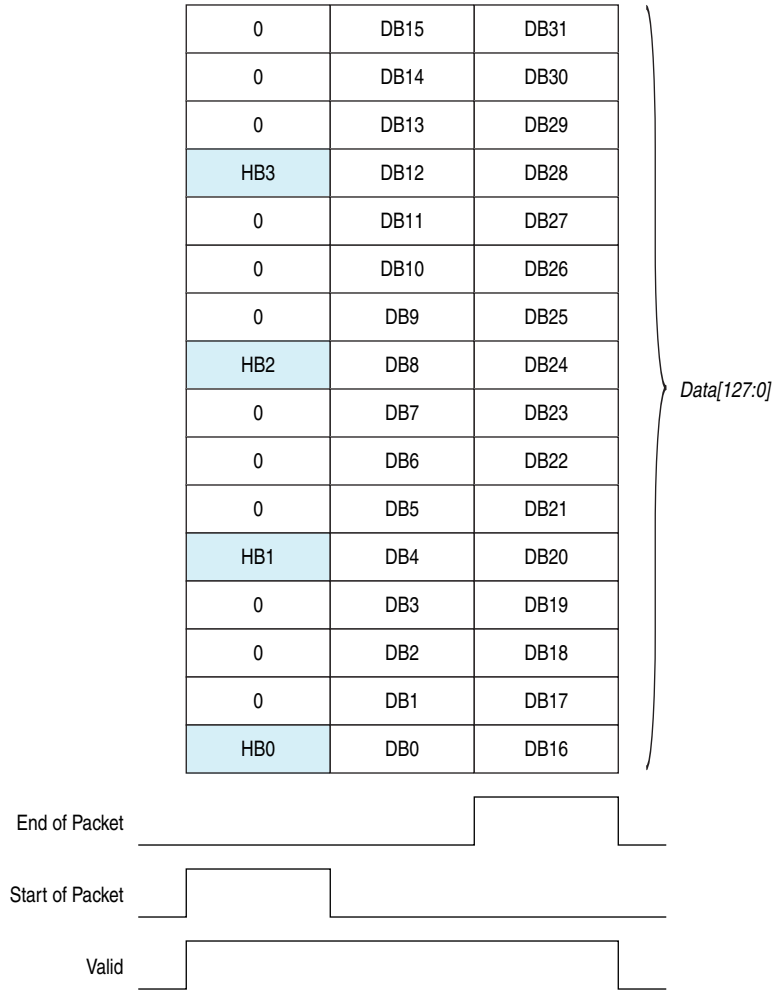
Figure 3–6. Secondary Stream Input Data Format

15-Nibble Code Word for Packet Payload	15-Nibble Code Word for Packet Header
0	0
0	0
0	0
0	0
0	0
nb0	0
nb1	0
nb2	0
nb3	0
nb4	0
nb5	0
nb6	nb0
nb7	nb1
p0	p0
p1	p1

Figure 3–7 shows a typical secondary stream packet with a four byte header (HB0, HB1, HB2 and HB3) and a 32-byte payload (DB0 … DB31). The core calculates the associated parity bytes. The secondary stream interface uses the start-of-packet (SOP) and end-of-packet (EOP) to determine if the current input is a header or payload.

Payloads that only contain the first 16 bytes can assert the EOP on the second cycle to terminate the packet sequence. Data is clocked in to the secondary stream interface via the `tx_xcvr_clk`. This clock is the same phase and frequency as the main-link lane 0 clock.

Figure 3–7. Typical Secondary Stream Packet



Audio Interface

The audio encoder is upstream of the secondary stream encoder. It generates the audio infoframe, timestamp, and audio sample packets from the incoming audio sample data stream. Then, it sends the three packet types to the secondary stream encoder before they are transmitted to the downstream sink device.

The audio port is parameterized for the number of audio channels required in the design. You can use 2 to 8 channels. Each channel's audio data is input on the `lpcm_data` port.

The IP core requires a `valid` signal for designs in which the `tx_audio_clk` signal is higher than the actual sample clock. The `valid` signal qualifies the audio data on the `lpcm_data` input. Table 3–8 describes the audio signals.

Table 3–8. Audio Signals

Signal	Comments
`tx_audio_clk`	Audio interface input clock.
`valid`	Audio input data valid.
`mute`	When asserted, indicates that audio muting is enabled.
`lpcm_data[n*32-1:0]`	*n*-channel, 32-bit audio sample data (refer to Figure 3–8).

Figure 3–8 shows the audio sample data bit fields. The packing format uses an IEC-60958-type encoding.

Figure 3–8. Audio Sample Data Bits

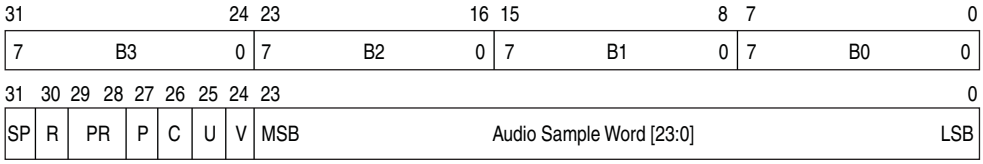


Table 3–9 shows the audio sample field definitions.

Table 3–9. Audio Sample Bit Field Definitions

Bit Name	Bit Position	Description
Audio sample word	Byte 2, bits 7:0 Byte 1, bits 7:0 Byte 0, bits 7:0	Audio data. The data content depends on the audio coding type. For LPCM audio, the audio most significant bit (MSB) is placed in byte 2, bit 7. If the audio data size is less than 24 bits, unused least significant bits (LSB) must be zero padded.
V	Byte 3, bit 0	Validity flag.
U	Byte 3, bit 1	User bit.
C	Byte 3, bit 2	Channel status.
P	Byte 3, bit 3	Parity bit.
PR	Byte 3, bits 4 - 5	Preamble code and its correspondence with IEC-60958 preamble: 00: Subframe 1 and start of the audio block (11101000 preamble) 01: Subframe1 (1110010 preamble) 10: Subframe 2 (1110100 preamble)
R	Byte3, bit 6	Reserved bit; must be 0.
SP	Byte 3, bit 7	Sample present bit: 1: Sample information is present and can be processed. 0: Sample information is not present. All one-sample channels, used or unused, must have the same sample present bit value. This bit is useful for situations in which 2-channel audio is transported over a 4-lane main link. In this operation, main link lanes 2 and 3 may or may not have the audio sample data. This bit indicates whether the audio sample is present or not.

The source automatically generates the audio infoframe and fills it with only information about the number of channels used. Use the audio channel status to provide any information about the audio stream needed by downstream devices.

MSA Interface

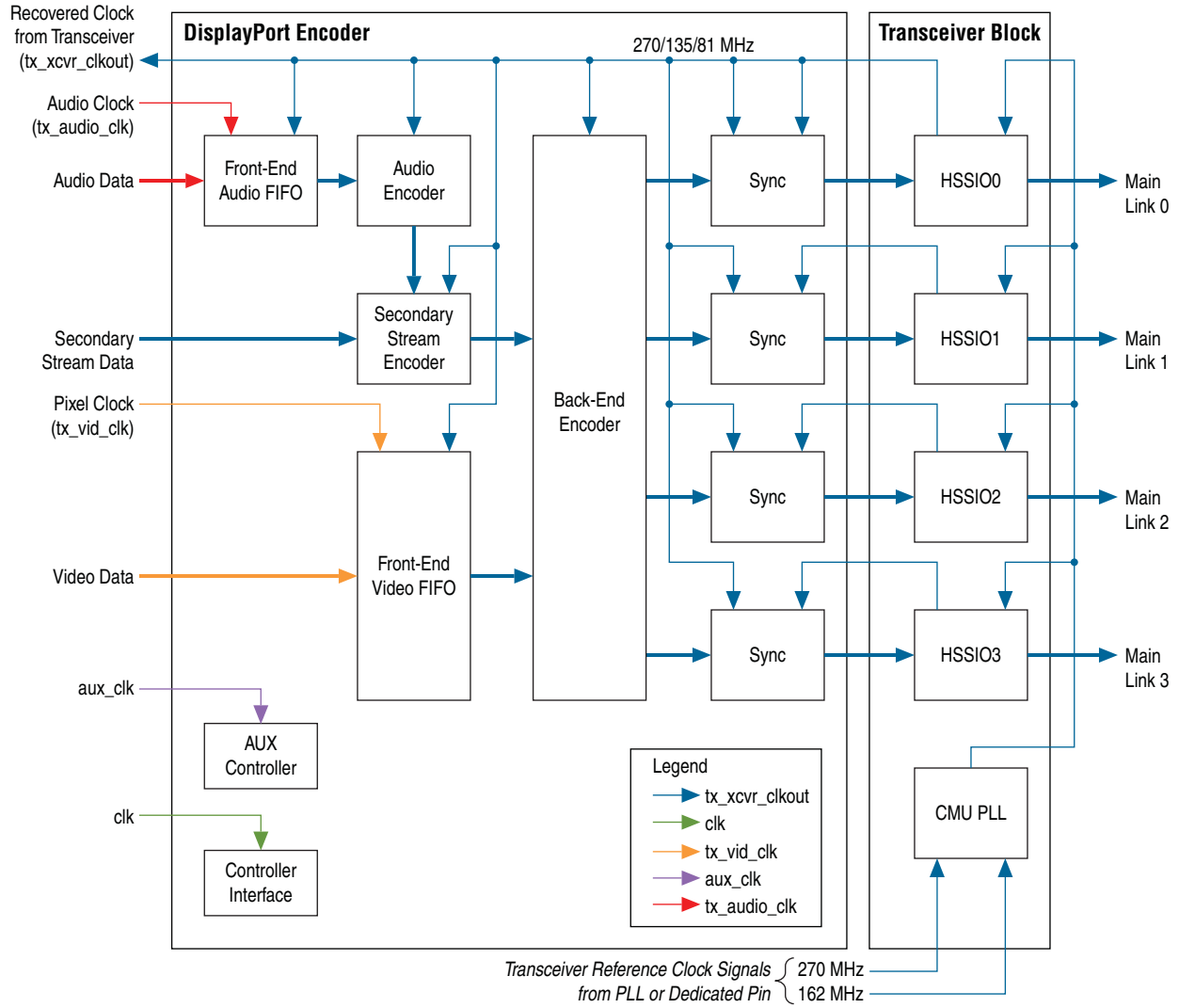
For applications that use a known video source signal, the added resource of video measurement can be removed. In this scenario, the DP Source uses the MSA values presented on the tx_msa_conduit signal bundle. The bundle contents is shown below,

```
wire [191:0] tx_msa_conduit ={Mvid[23:0],Nvid[23:0], Htotal[15:0],
Vtotal[15:0], HSP, HSW[14:0], Hstart[15:0], Vstart[15:0], VSP, VSW[14:0],
Hwidth[15:0], Vheight[15:0], MISC0[7:0], MISC1[7:0]};
```

Source Clock Tree

Figure 3–9 shows the source's clock tree.

Figure 3–9. Source Clock Tree



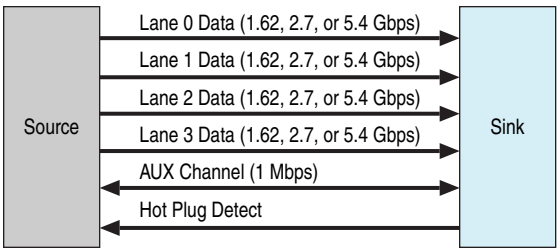
The source uses the following clocks:

- Local pixel clock (`tx_vid_clk`), which clocks video data into the IP core.
- Main link clock (`tx_xcvr_clkout`), which clocks data out of the IP core and into the high-speed serial output (HSSO) components. The IP core generates the main link clock internally by the transceiver's CMU PLL. The CMU PLL must be supplied with either a 270 or 162 MHz clock according to the actual data rate requested on the `tx_link_rate` port. You can use other frequencies by changing the CMU PLL divider ratios and/or performing reconfiguring the transceiver.
- The IP core also requires a 16 MHz clock (`aux_clk`) to encode/decode the AUX channel. The IP core drives the Avalon-MM interface by a separate clock (`clk`).
- The core uses `tx_audio_clk` for the audio interface.

Sink Overview

The DisplayPort sink has a scalable main link with 1, 2, or 4 lanes for a total of 21.6 Gbps bandwidth. A bidirectional AUX channel with 1 Mbps Manchester encoding provides side-band communication. The sink drives a hot plug detect (HPD) signal to notify the source that a sink is present. Additionally, it provides an interrupt mechanism so that the sink can get the source's attention. Refer to Figure 4–1.

Figure 4–1. DisplayPort Sink Block Diagram



The main link has three selectable data rates: 1.62, 2.7, and 5.4 Gbps. The source device sets the lane count and link rate combination (referred to as the policy) according to the sink's capabilities and required video bandwidth as shown in Figure 4–1.

The AUX channel is an AC-coupled differential pair for bidirectional communication. The signaling is a self-clocked Manchester encoding at 1 Mbps. Like 100-T Ethernet, the encoder uses a preceding synchronization pattern in each 16-byte maximum packet. The AUX channel uses a master/slave hierarchy in which the source (master) initiates all communication.

Sink Functional Description

The DisplayPort sink has a complete set of parameters for optimizing device resources. The DisplayPort sink consists of a DisplayPort decoder block, a transceiver management block, and a controller interface block with an Avalon-MM interface for connecting with an embedded controller such as the Nios II processor. Figure 4–2 shows the overall top-level design. You can configure the ports using an RTL wrapper instantiation or implementing the IP core as a Qsys component.

Figure 4–2. DisplayPort Sink Top-Level Block Diagram

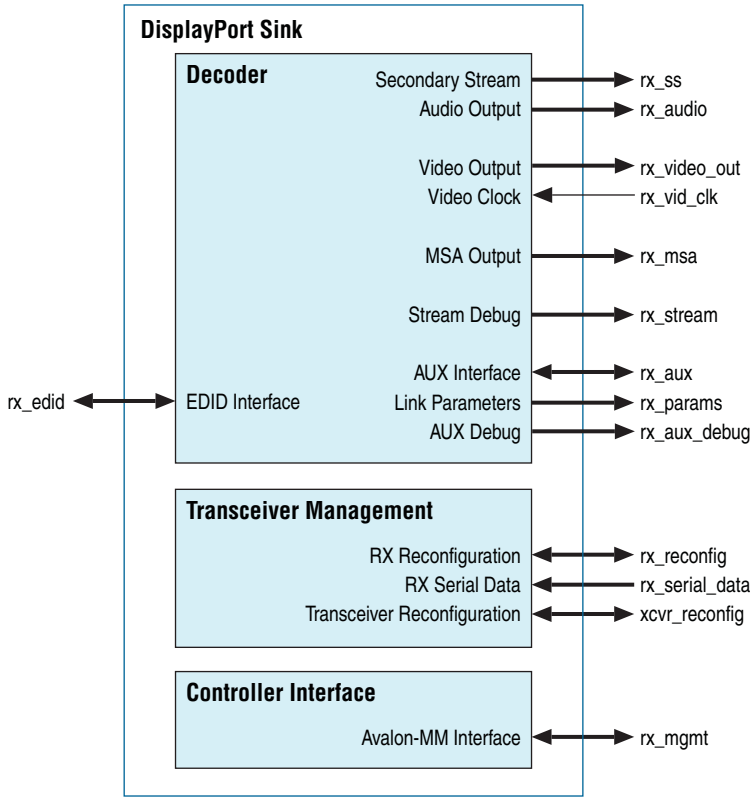
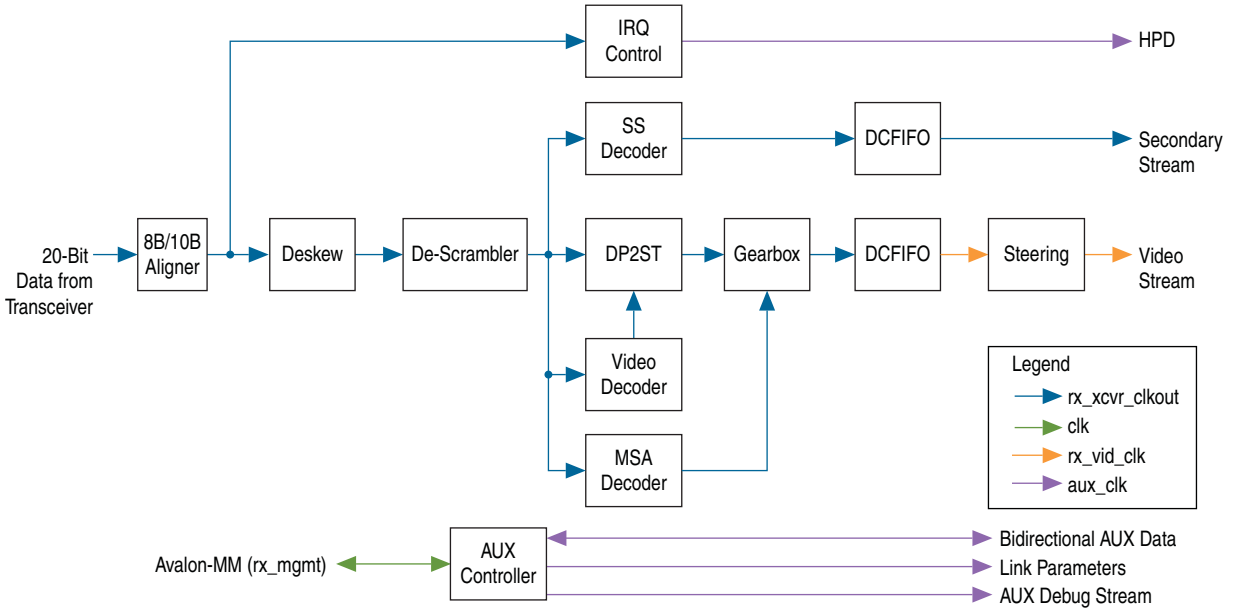


Figure 4–3 shows the DisplayPort sink functional block diagram.

Figure 4–3. DisplayPort Sink Block Diagram



The device transceiver sends 20-bit (double rate) parallel DisplayPort data to the sink. Each data lane is clocked in to the IP core by its own respective clock output from the transceiver. Inside the sink, the four independent clock domains are synchronized to the lane 0 clock. Then, the IP core performs the following actions:

1. The IP core aligns the data stream and performs 8B/10B decoding.
2. The IP core deskews the data and then descrambles it.
3. The IP core splits the unscrambled data stream into parallel paths.
   - The SS decoder block performs secondary stream decoding, which the core transfers into the `rx_xcvr_clkout` domain via a DCFIFO.
   - The main data path extracts all pixel data from the incoming stream. Then, the gearbox block re-samples the pixel data into the current bit-per-pixel data width. Next, the IP core crosses the pixel data into the `rx_vid_clk` domain via a DCFIFO. Finally, the IP core steers the data into a single pixel data stream.
   - MSA decode path.
   - Video decode path.

You configure the sink to output the video data as a proprietary data stream. You specify the output pixel data width at 6, 8, 10, 12, or 16 bpc. This format can interface with downstream Altera Video and Image Processing (VIP) Suite components.

The AUX controller can operate in an autonomous mode in which the sink controls all AUX channel activity without an external embedded controller. The IP core outputs an AUX debugging stream so that you can inspect the activity on the AUX channel in realtime.

Sink Parameters

You set parameters for the sink using the DisplayPort parameter editor. Table 4–1 lists the DisplayPort sink parameters.

Table 4–1. Sink Parameters

Parameter	Description
Device family	Targeted device family (Arria V, Arria V GZ, or Stratix V); matches the project device family.
Support DisplayPort sink	Enable DisplayPort sink.
IEEE OUI	Specify an IEEE organizationally unique identifier (OUI) as part of the DPCD registers.
Maximum video output color depth	Video output interface port bits per color (bpc). Determines top level video input port width (e.g., 6 bpc = 18 bits, 16 bpc = 48 bits).
RX maximum link rate	Maximum link rate. 20 = 5.4 Gbps, 10 = 2.7 Gbps, 6 = 1.62 Gbps
Maximum lane count	Maximum lanes used (1, 2, or 4).
Sink scrambler seed value	Scrambler block initial seed value. Use 16'hFFFF for DP and 16'hFFFFE for eDP.
Enable AUX debug stream	Enable AUX traffic output to an Avalon-ST port.
Enable GPU control	Use an embedded controller to control the sink.
Export MSA	Outputs MSA on top level port interface.
Support secondary data channel	Enable secondary data.
Support audio data channel	Enable audio packet decoding.
Number of audio data channels	Number of audio channels supported.
Support CTS test automation	Support automated test features.
6-bpc RGB or YCbCr 4:4:4 (18 bpp)	Support 18 bpp decoding.
8-bpc RGB or YCbCr 4:4:4 (24 bpp)	Support 24 bpp decoding.
10-bpc RGB or YCbCr 4:4:4 (30 bpp)	Support 30 bpp decoding.
12-bpc RGB or YCbCr 4:4:4 (36 bpp)	Support 36 bpp decoding.
16-bpc RGB or YCbCr 4:4:4 (48 bpp)	Support 48 bpp decoding.
8-bpc YCbCr 4:2:2 (16 bpp)	Support 16 bpp decoding.
10-bpc YCbCr 4:2:2 (20 bpp)	Support 20 bpp decoding.
12-bpc YCbCr 4:2:2 (24 bpp)	Support 24 bpp decoding.
16-bpc YCbCr 4:2:2 (32 bpp)	Support 32 bpp decoding.

Sink Interfaces

Table 4–2, Table 4–3, Table 4–4, Table 4–5, Table 4–6, and Table 4–7 summarize the sink's interfaces. Your instantiation contains only the interfaces that you have enabled. The following sections describe these interfaces.

Table 4–2. Controller Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
clk	Clock	N/A	N/A	Clock for embedded controller.	clk
reset	Reset	clk	N/A	Reset for embedded controller.	reset
rx_mgmt	AV-MM	clk	reset	Avalon-MM interface for embedded controller.	rx_mgmt_address[8:0]
					rx_mgmt_chipselect
					rx_mgmt_read
					rx_mgmt_write
					rx_mgmt_writedata[31:0]
					rx_mgmt_readdata[31:0]
					rx_mgmt_waitrequest
rx_mgmt_irq	IRQ	clk	reset	IRQ for embedded controller.	rx_mgmt_irq

Table 4–3. Transceiver Management Interface

Interface	Port Type	Clock Domain	Reset	Description	Port (1)
xcvr_mgmt_clk	Clock	N/A	N/A	Transceiver management clock.	xcvr_mgmt_clk
xcvr_refclk	Conduit	N/A	N/A	Transceiver reference clocks.	xcvr_refclk[1:0]
rx_serial_data	Conduit	rx_xcvr_clkout	N/A	Transceiver serial data out.	rx_serial_data[*m*-1:0]
rx_reconfig	Conduit	xcvr_mgmt_clk	reset	Transceiver reconfiguration handshaking.	rx_link_rate (2)
					rx_reconfig_req
					rx_reconfig_ack
					rx_reconfig_busy
xcvr_reconfig	Conduit	xcvr_mgmt_clk	N/A	Transceiver reconfiguration.	reconfig_to_xcvr[(2*n* + *m*) × 70 - 1:0]
					reconfig_from_xcvr[(2*n* + *m*) × 46 - 1:0]

Note:

(1) *n* is the number of TX lanes and *m* represents the RX lanes.

(2) The tx_link_rate signal is 2 bits wide for 5.4 Gbps support.

Table 4–4. Video Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
rx_vid_clk	Clock	N/A	N/A	Video clock.	rx_vid_clk
rx_video_out	Conduit	rx*N*_vid_clk	reset	Video output.	rx_vid_valid
					rx_vid_sol
					rx_vid_eol
					rx_vid_sof
					rx_vid_eof
					rx_vid_locked
					rx_vid_data[3*v* - 1:0] (1)

Note:

(1) *v* is the number bits per color.

Table 4–5. AUX Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
aux_clk	Clock	N/A	N/A	AUX channel clock.	aux_clk
aux_reset	Reset	aux_clk	N/A	AUX channel reset.	aux_reset
rx_aux	Conduit	aux_clk	aux_reset	AUX channel interface.	rx_aux_in
					rx_aux_out
					rx_aux_oe
					rx_hpd
rx_aux_debug	AV-ST	aux_clk	aux_reset	Avalon-ST stream of AUX data for debugging.	rx_aux_debug_data[31:0]
					rx_aux_debug_valid
					rx_aux_debug_sop
					rx_aux_debug_eop
					rx_aux_debug_err
					rx_aux_debug_cha
EDID (rx_edid)	AV-MM	aux_clk	aux_reset	Avalon-MM master interface to external on-chip memory for EDID.	rx_edid_address[7:0]
					rx_edid_read
					rx_edid_write
					rx_edid_writedata[7:0]
					rx_edid_readdata[7:0]
					rx_edid_waitrequest

Table 4–6. Debugging Interface

Interface	Port Type	Clock Domain	Reset	Description	Port
Link Parameters (rx_params)	Conduit	rx_vid_clk	reset		rx_lane_count[4:0]
Debugging (rx_stream)	Conduit	rx_xcvr_clkout	reset	Raw symbol output stream.	rx_stream_data[63:0]
					rx_stream_ctrl[7:0]
					rx_stream_valid
					rx_stream_clk

Table 4–7. Secondary Interface

Interface	Signal Type	Clock Donaim	Reset	Description	Port
rx_xcvr_clkout	Clock	N/A	N/A	Clock.	rx_xcvr_clkout
MSA (rx_msa)	Conduit	rx_xcvr_clkout	reset	Output for current MSA parameters received from the source.	rx_msa[215:0]
Secondary Stream (rx_ss)	AV-ST	rx_xcvr_clkout	reset	Secondary stream interface.	rx_ss_data[159:0]
					rx_ss_valid
					rx_ss_sop
					rx_ss_eop
Audio (rx_audio)	Conduit	rx_xcvr_clkout	reset	Decoded audio data and clock interface.	rx_audio_lpcm_data[]
					rx_audio_valid
					rx_audio_mute
					rx_audio_infoframe[39:0]

Controller Interface

The optional controller interface allows you to control the sink from an external or on-chip controller, such as the Nios II processor for debugging. The controller interface is an Avalon-MM slave that also gives access to the sink's internal status registers.

The sink asserts the rx_mgmt_irq port when issuing an interrupt to the controller. Refer to "DPRX_AUX_IRQ_EN" on page 10–20 for more details.

For a detailed description of the sink's register map, refer to Chapter 10, DisplayPort Sink Register Map and DCPD Locations.

AUX Interface

The IP core has three ports to control the serial data across the AUX channel:

- Data input (rx_aux_in)
- Data output (rx_aux_out)
- Output enable (rx_aux_oe). The output enable port controls the direction of data across the bidirectional link.

The AUX channel's physical layer is a bidirectional 2.5 V SSTL Class II interface.

A state machine decodes the incoming AUX channel's Manchester encoded data using the 16 MHz clock. The message parsing drives the state machine input directly. The state machine performs all lane training and EDID link-layer services.

The sink's AUX interface also generates appropriate HPD IRQ ports. These ports occur if the sink's main link decoder detects a signal loss.

AUX Debug Interface

The AUX controller lets you capture all bytes sent from and received by the AUX channel, which is useful for debugging. The IP core supports a standard stream interface that can drive an Avalon-ST FIFO component directly. Table 4–8 describes the stream ports.

Table 4–8. Sink AUX Debug Interface Ports

Port	Comments
rx_aux_debug_data[31:0]	The sink AUX debug interface inserts a 1 µs timestamp counter in bits [31:8]. Bits [7:0] represent the bytes received or transmitted.
rx_aux_debug_valid	Qualifies valid stream data.
rx_aux_debug_sop	Indicates the message packet's first byte.
rx_aux_debug_eop	Indicates the message packet's last byte. The last byte should be ignored and is not part of the message.
rx_aux_debug_err	Indicates if the core detects an error in the current byte.
rx_aux_debug_cha	Indicates the direction of the current byte. 1 = byte transmitted by the source, 0 = byte received from the sink.

EDID Interface

You can use the Avalon-MM EDID interface to access an on-chip memory region containing the sink's EDID data. The AUX sink controller reads and writes to this memory according to traffic on the AUX channel.

The Avalon-MM interface uses an 8-bit address with an 8-bit data bus. The interface assumes a read latency of 1.

The IP core does not instantiate this interface if your design uses a controller to control the sink.

Refer to the *VESA Enhanced Extended Display Identification Data Implementation Guide* for more information.

Debugging Interface

The IP core provides the following debugging interfaces.

Link Parameters Interface

The sink provides link level data for debugging and configuring external components using the lane_count port.

Video Stream Out Interface

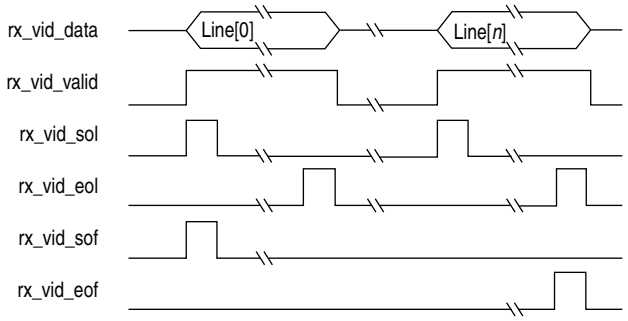
This interface provides access to the post-scrambler DisplayPort data, which is useful for low-level debugging source equipment.

Video Interface

This interface (rx_video_out) allows access to the video data as a non-Avalon-ST stream. You can use this streams to interface with an external pixel clock recovery function. The stream provides synchronization pulses at the start and end of active lines, and at the start and end of active frames as shown in Figure 4–4.

Refer to the *Video and Image Processing Suite User Guide* for more information on interfacing with components in the Altera VIP Suite.

Figure 4–4. Video Out Image Port Timing Diagram



rx_video_out can interface with a clock video input (CVI). CVI accepts the following video signals with a separate synchronization mode: `datavalid`, `de`, `h_sync`, `v_sync`, `f`, `locked`, and `data`. The DisplayPort rx_video_out interface has the following signals: `rx_vid_valid`, `rx_vid_sol`, `rx_vid_eol`, `rx_vid_sof`, `rx_vid_eof`, `rx_vid_locked`, and `rx_vid_data`. Table 4–9 describes how to connect the CVI and DisplayPort sink signals.

Table 4–9. Connecting CVI Signals to DisplayPort Sink Signals

CVI Signal	DisplayPort Sink Signal	Comment
`data`	`rx_vid_data`	–
`datavalid`	–	Drive high because the data is not oversampled.
`f`	–	Drive low because the video is not progressive.
`locked`	`rx_vid_locked`	–
`de`	`rx_vid_valid`	–
`h_sync`	`rx_vid_eol`	The `rx_vid_eol` signal generates the `h_sync` pulse by delaying it (by 1 clock cycle) to appear in the horizontal blanking period after the active video ends (VALID is deasserted).
`v_sync`	`rx_vid_eof`	The `rx_vid_eof` signal generates the `v_sync` pulse by delaying it (by 1 clock cycle) to appear in the vertical blanking period after the active video ends (VALID is deasserted).

Example 4–1 provides a Verilog HDL code example for the CVI interface.

Example 4–1. Verilog HDL CVI Example

```
// CVI V-sync, H-sync, and datavalid are derived from delayed versions
// of the eol and eof signals
always @ (posedge clk_video)
begin
    rx_vid_h_sync <= rx_vid_eol;
    rx_vid_v_sync <= rx_vid_eof;
end

assign data      = rx_vid_data;
assign datavalid = 1'b1;
assign f         = 1'b0;
assign locked    = rx_vid_locked;
assign de        = rx_vid_valid;
assign h_sync    = rx_vid_h_sync;
assign v_sync    = rx_vid_v_sync;
```

Transceiver Management Interface

The 20-bit transceiver management interface consists of a native PHY block on the RX and TX with a CMU PLL. The design uses a soft 8B/10B encoder. The transceiver can be reconfigured to use one of two reference clocks:

- 162 MHz clock for reduced bit rate (RBR)
- 270 MHz clock for high bit rate (HBR or HBR2).

You use the Transceiver Reconfiguration Controller to switch between the two reference clocks. To switch them, you reconfigure the logical reference clock source for the RX CDR PLLs. The IP core sets the `rx_link_rate` to:

- 00 (RBR)
- 01 (HBR)
- 10 (HBR2)

When the core makes a request, the `rx_reconfig_req` port goes high. The user logic asserts `rx_reconfig_ack` and then reconfigures the transceiver. During reconfiguration, the user logic holds `rx_reconfig_busy` high; drive it low when reconfiguration completes.

The transceiver requires a reconfiguration controller.

- Refer to the *Altera Transceiver PHY IP Core User Guide* for more information about how to reconfigure the transceiver.
- Refer to *AN 645: Dynamic Reconfiguration of PMA Controls in Stratix V Devices* for more information about using the Transceiver Reconfiguration Controller to reconfigure the Stratix V Physical Media Attachment (PMA) controls dynamically.

- Refer to *AN 645: Using the Transceiver Reconfiguration Controller for Dynamic Reconfiguration in Arria V and Cyclone V Devices* for more information about using the Transceiver Reconfiguration Controller to reconfigure the Arria V Physical Media Attachment (PMA) controls dynamically.
- Refer to *AN 678: High-Speed Link Tuning Using Signal Conditioning Circuitry in Stratix V Transceivers* for information about link tuning.

Secondary Stream Interface

The secondary streams data can be received via the rx_ss interfaces. The interfaces do not allow for back-pressure and assume the downstream logic can handle complete packets. The rx_ss interface does not distinguish between the types of packets it receives.

The format rx_ss interface output corresponds to four 15-nibble code words as specified by the DisplayPort 1.1a specification section 2.2.6.3. These 15-nibble code words would typically be supplied to the downstream Reed-Solomon decoder. The format differs for both header and payload, as shown in Figure 4–5.

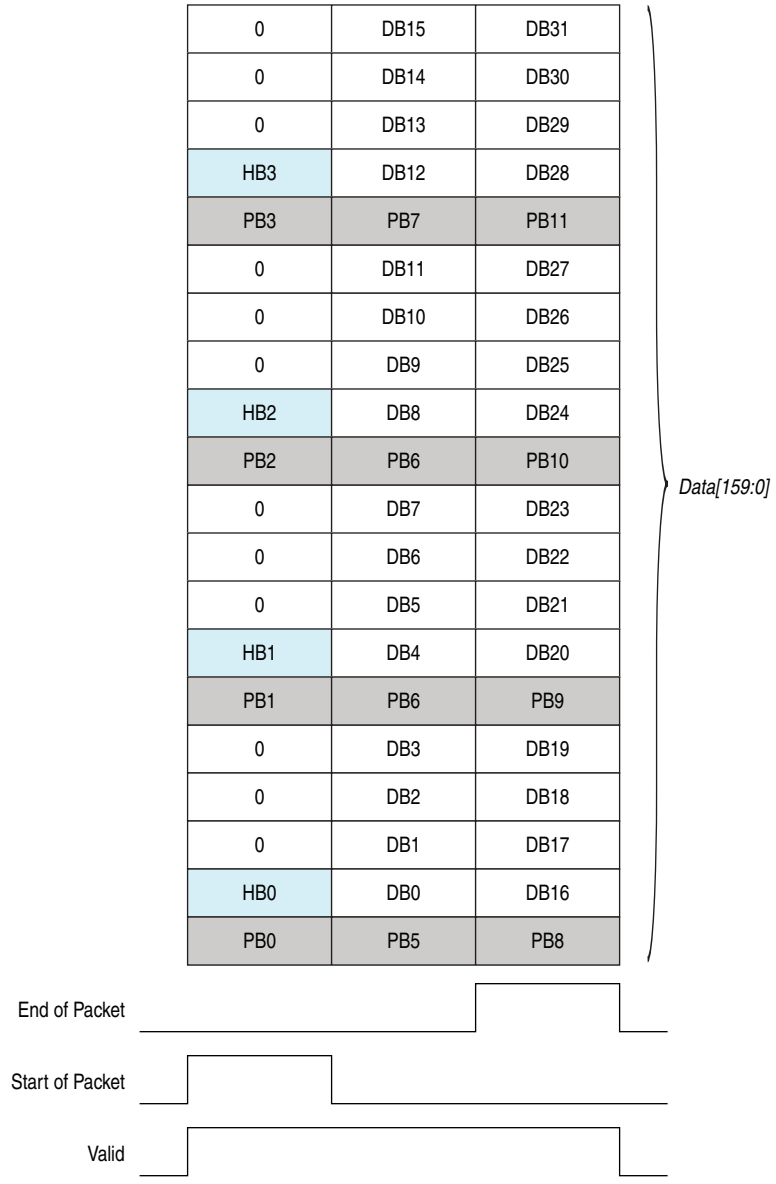
Figure 4–5. rx_ss Input Data Format

15-Nibble Code Word for Packet Payload	15-Nibble Code Word for Packet Header
0	0
0	0
0	0
0	0
0	0
nb0	0
nb1	0
nb2	0
nb3	0
nb4	0
nb5	0
nb6	nb0
nb7	nb1
p0	p0
p1	p1

Figure 4–6 shows a typical secondary stream packet with the four byte header (HB0, HB1, HB2, and HB3) and 32-byte payload (DB0, ..., DB31). Each symbol has an associated parity nibble (PB0, ..., PB11). Downstream logic can use the start-of-packet and end-of-packet to determine if the current input is a header or payload symbol.

Data is clocked out of the rx_ss port using the `rx_ss_clk` signal. This signal is the same phase/frequency as the main link lane 0 clock.

Figure 4–6. Typical Secondary Stream Packet



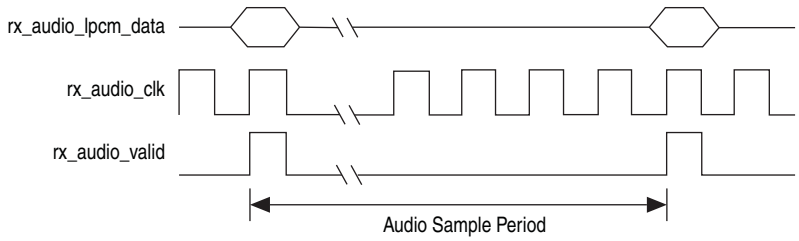
Audio Interface

The audio interfaces are downstream from the secondary stream decoder. They extract and decode the audio infoframe packets, audio timestamp packets, and audio sample data.

The audio timestamp packet payload contains *M* and *N* values, which the sink uses to recover the source's audio sample clock. The rx_audio port uses the values to generate the `valid` signal according to sample audio data. Data is clocked out using the `rx_xcvr_clkout` signal, which is driven at half the current link speed data rate. The sink generates the `rx_audio_valid` signal using the *M* and *N* values, and asserts it at the current audio sample clock rate. Figure 4–7 shows the data clocking.

The `rx_audio_mute` signal indicates if audio data is present on the DisplayPort interface.

Figure 4–7. rx_audio Data Output



The captured audio infoframe is available on the audio port. The 5-byte port corresponds to the 5 bytes used in the audio infoframe (refer to CEA-861-D). The audio infoframe describes the type of audio content.

MSA Interface

The rx_msa_conduit ports allow designs access to the MSA and VB-ID parameters on a top-level port. Table 4–10 show the 216-bit port bundle assignments. The prefixes `msa` and `vbid` denote parameters from the MSA and VB-ID packets, respectively.

The sink asserts bit `msa_valid` when all `msa_` signals are valid and de-asserted during MSA update. The MSA parameters are assigned to zero when the sink is not receiving valid video data.

The sink asserts bit `vbid_strobe` for one clock cycle when the VB-ID is detected and all `vbid_` signals are valid to be read.

Table 4–10. rx_msa_conduit Port Signals (Part 1 of 2)

Bit	Signal	Comments
215	`vbid_strobe`	0 = VB-ID fields invalid, 1 = VB-ID fields valid.
214:209	`vbid_vbid[5:0]`	VB-ID bit field (refer to the DisplayPort 1.2a Specification, Table 2-3).
208:201	`vbid_Mvid[7:0]`	Refer to the DisplayPort 1.2a Specification.
200:193	`vbid_Maud[7:0]`	Refer to the DisplayPort 1.2a Specification.
192	`msa_valid`	0 = MSA fields invalid, 1 = MSA fields valid.
191:168	`msa_Mvid[23:0]`	Refer to the DisplayPort 1.2a Specification.
167:144	`msa_Nvid[23:0]`	Refer to the DisplayPort 1.2a Specification.
143:128	`msa_Htotal[15:0]`	Refer to the DisplayPort 1.2a Specification.
127:112	`msa_Vtotal[15:0]`	Refer to the DisplayPort 1.2a Specification.
111	`msa_HSP`	Refer to the DisplayPort 1.2a Specification.

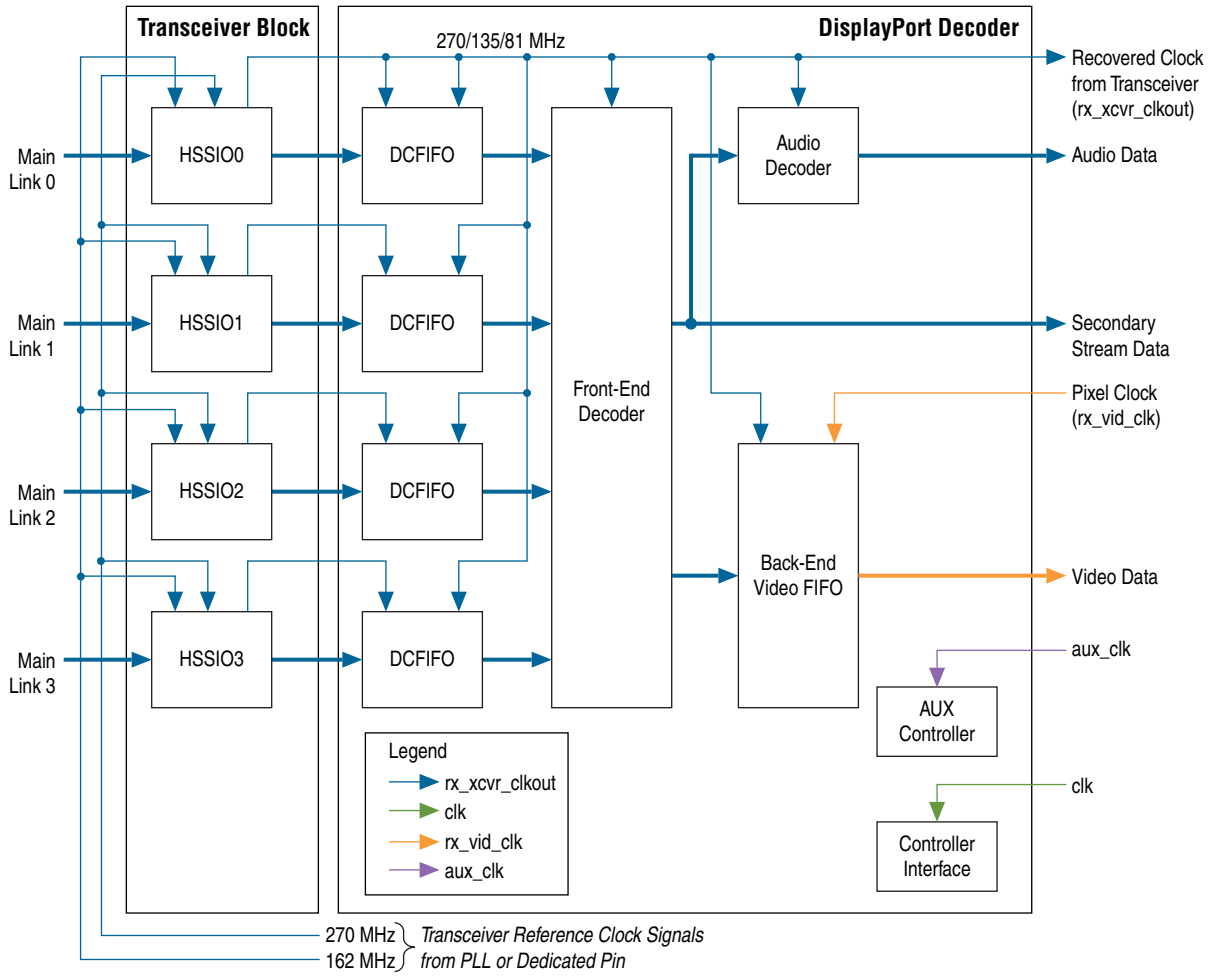
Table 4–10. rx_msa_conduit Port Signals (Part 2 of 2)

110:96	msa_HSW[14:0]	Refer to the DisplayPort 1.2a Specification.
95:80	msa_Hstart[15:0]	Refer to the DisplayPort 1.2a Specification.
79:64	msa_Vstart[15:0]	Refer to the DisplayPort 1.2a Specification.
63	msa_VSP	Refer to the DisplayPort 1.2a Specification.
62:48	msa_VSW[14:0]	Refer to the DisplayPort 1.2a Specification.
47:32	msa_Hwidth[15:0]	Refer to the DisplayPort 1.2a Specification.
31:16	msa_Vheight[15:0]	Refer to the DisplayPort 1.2a Specification.
15:8	msa_MISC0[7:0]	Refer to the DisplayPort 1.2a Specification.
7:0	msa_MISC1[7:0]	Refer to the DisplayPort 1.2a Specification.

Sink Clock Tree

Figure 4–8 shows the DisplayPort sink clock tree.

Figure 4–8. Sink Clock Tree



The IP core receives DisplayPort serial data via the high-speed serial interface (HSSI). The HSSI requires a 162 or 270 MHz clock for correct data locking. You can supply these two frequencies to the HSSI using a reference clock provided by an Altera PLL or pins.

The IP core synchronizes HSSI 20-bit data to a single HSSI[0] clock that clocks the data into the DisplayPort front-end decoder. The IP core uses a double data rate, therefore, this clock is 270, 135, or 81 MHz.

The IP core crosses the reconstructed pixel data into a local pixel clock (`rx_vid_clk`) via an output DCFIFO, which drives the pixel stream output. The `rx_vid_clk` must be greater than or equal to the pixel clock in the up-stream source. If `rx_vid_clk` is lower than the up-stream pixel clock, the DCFIFO overflows. If the `rx_vid_clk` is greater than the up-stream source pixel clock, the output port experiences a de-assertion of the valid port on cycles where pixel data is not available. The optimum frequency is the exact clock rate in the up-stream source. However, determining this clock frequency requires pixel clock recovery techniques that are beyond the scope of this document.

Secondary stream data is clocked by `rx_xcvr_clkout`. The sink IP core also requires a 16 MHz clock (`aux_clk`) to drive the internal AUX controller and an Avalon clock for the Avalon-MM interface (`clk`).

5. DisplayPort MegaCore Function Hardware Demonstration

Introduction

The Altera DisplayPort hardware demonstration allows you to evaluate the functionality of the DisplayPort MegaCore function and provides a starting point for you to create your own design. The example design uses a fully functional OpenCore Plus evaluation version, giving you the freedom to explore the core and understand it's performance on a hardware level.

This hardware demonstration targets the following boards:

- Production Stratix V GX C2 revision C development board or Arria V GX revision C4 ES development board
- Bitec DisplayPort HSMC revision 5 daughter card

Refer to www.bitec-dsp.com/hsmc_dp.html for more information on the Bitec daughter card.

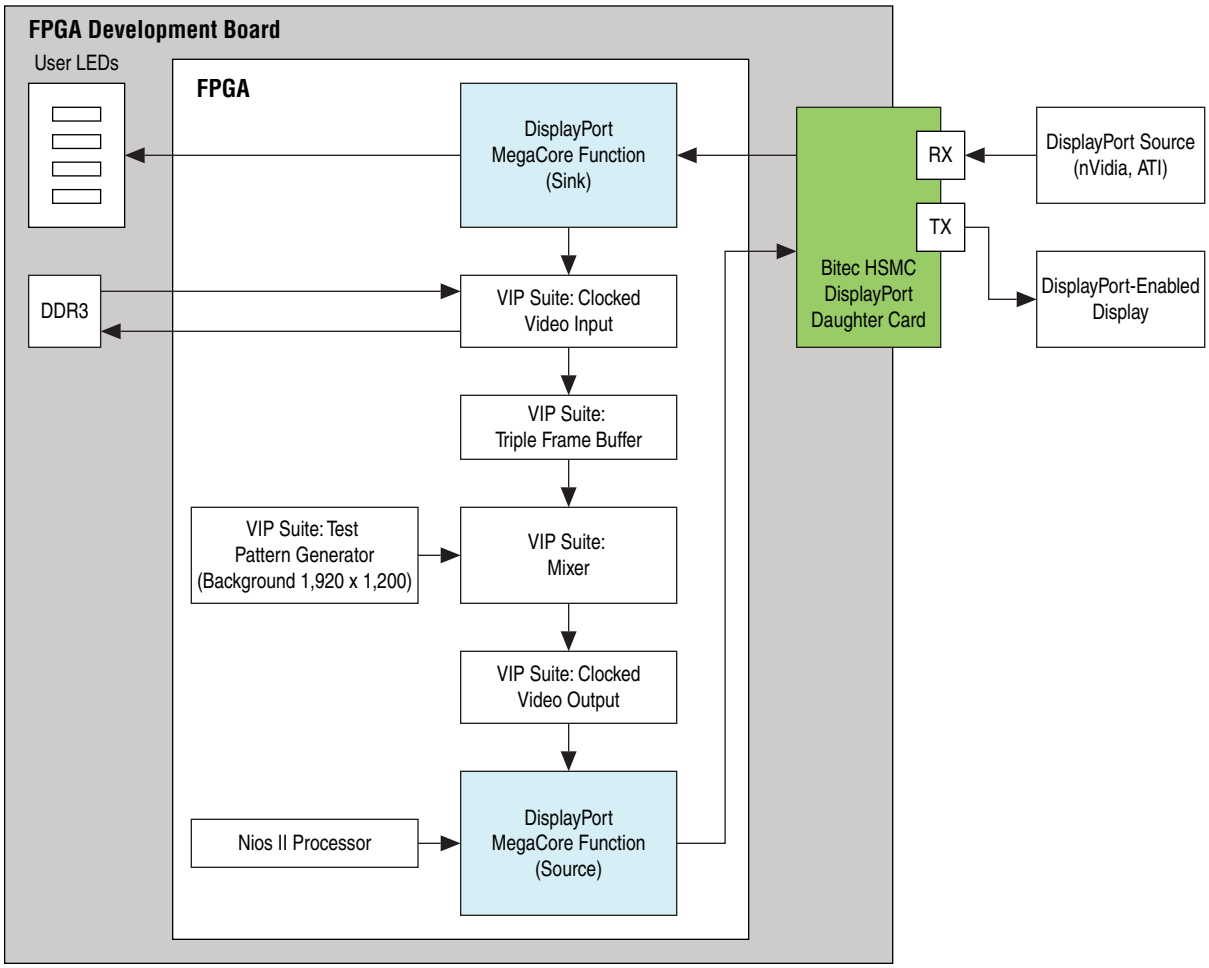
The design performs an SDRAM loop-through for a standard DisplayPort video stream. You connect a DisplayPort-enabled device—such as an nVidia or ATI graphics card transmitter—to the DisplayPort sink input. The DisplayPort sink decodes the port into a standard video stream and triple frame buffers it into external SDRAM. The hardware demonstration mixes a buffered image with a 1,920 x 1,200 color bar image and sends it to the DisplayPort source. The HSMC daughter card's DisplayPort source port transmits the image to a monitor.

If you use another Altera development board, you must change the device assignments and the pin assignments. You make these changes in the **assignments.tcl** file (this file is described in a later section).

If you use another DisplayPort daughter card, you must change the pin assignments, Qsys system, and software.

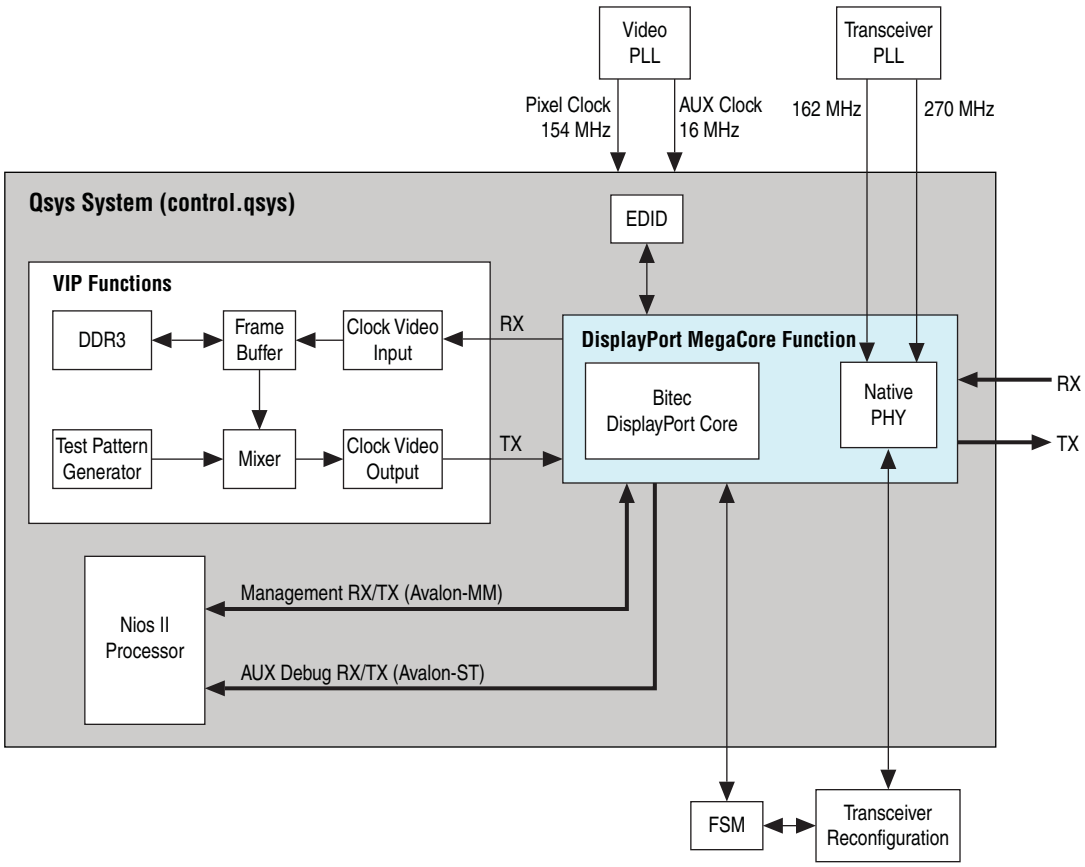
Figure 5–1 shows an overview of the hardware demonstration.

Figure 5–1. Hardware Demonstration Overview



The DisplayPort sink uses its internal state machine to negotiate link training upon power up. A Nios II embedded processor performs the source link management; software performs the link training management. Refer to Figure 5–2.

Figure 5–2. Hardware Demonstration Block Diagram



During operation, you can adjust the DisplayPort source resolution (graphics card) from the PC and observe the effect on the IP core. The Nios II software prints the source and sink AUX channel activity. Pressing a push-button prints the current TX and RX main stream attributes (MSA). The development board user LEDs illuminate to indicate the function shown in Table 5–1.

Table 5–1. LED Function

Stratix V LED	Arria V LED	Function
USER_LED_G0	USER1_LED_G0	This LED indicates that source has successfully lane trained and is sending video (`rx_vid_locked` drives this LED). The LED turns off if the source is not driving good video.
USER_LED_G6	USER1_LED_G6	This LED indicates that PLL is locked and stable; if it flickers, there is an issue with the board itself.
USER_LED_G7	USER1_LED_G7	This LED is for the system reset (tracks the reset that waits for PLL lock before releasing). It goes high while the chip is resetting.
USER_LED_G1	USER1_LED_G1	This LED illuminates for 1-lane designs.
USER_LED_G2	USER1_LED_G2	This LED illuminates for 2-lane designs.
USER_LED_G3	USER1_LED_G3	This LED illuminates for 4-lane designs.

When creating your own design, note the following design tips:

- The Bitec daughter card has inverted transceiver polarity. When creating your own sink (RX) design, use the **Invert transceiver polarity** option to enable or disable inverted polarity.
- The DisplayPort standard reverses the RX and TX transceiver channels to minimize noise for one- or two-lane applications. If you create your own design targeting the Bitec daughter card, ensure that the following signals share the same transceiver channel:
  - `TX0` and `RX3`
  - `TX1` and `RX2`
  - `TX2` and `RX1`
  - `TX3` and `RX0`

  Refer to the **assignments.tcl** file for an example of how the channels are assigned in the hardware demonstration.

Clocking

The device's Gigabit transceivers operate at 5.4, 2.7, and 1.62 Gbps and require reference clocks of 270 and 162 MHz. When the link rate changes, a small state machine reconfigures the transceiver to select the appropriate reference clock. and changes the transceiver PLL settings.

Currently, only Stratix V devices support 5.4 Gbps operation.

Required Hardware

The hardware demonstration requires the following hardware:

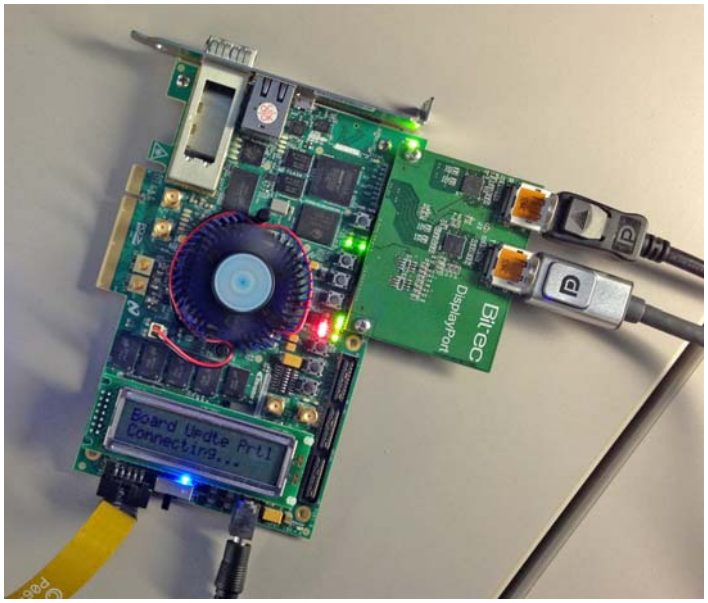
- Altera FPGA kit (includes USB cable to connect the board to your PC)
- Bitec DisplayPort HSMC daughter card
- PC with a DisplayPort output

- Monitor with a DisplayPort input
- Two DisplayPort cables
  - One cable connects from the graphics card to the FPGA development board
  - The other cable connects from the FPGA development board to the monitor

Altera recommends that you first test the PC and monitor by connecting the PC directly to the monitor to ensure that you have all drivers installed correctly.

Figure 5–3 shows an example hardware setup using the FPGA development board, Bitec daughter card, and cables.

Figure 5–3. Example Hardware Setup



Design Walkthrough

Setting up and running the DisplayPort hardware demonstration consists of the following steps:

1. Set up the hardware.
2. Copy the design files to your working directory.
3. Build the FPGA design.
4. Build the software, download it into the FPGA, and run the software.
5. Power-up the DisplayPort monitor and view the results.

A variety of scripts automate these steps. The following sections describe the process in detail.

Set Up the Hardware

Set up the hardware using the following steps:

1. Connect the Bitec daughter card to the FPGA development board.
2. Connect the development board to your PC using a USB cable.

   ☞ The FPGA development board has an On-Board USB-Blaster™ II connection. If your version of the board does not have this connection, you can use an external USB-Blaster cable. Refer to the documentation for your board for more information.

3. Connect a DisplayPort cable from the DisplayPort TX on the Bitec HSMC daughter card to a DisplayPort monitor (do not power up the monitor).
4. Power-up the development board.
5. Connect one end of a DisplayPort cable to your PC (do not connect the other end to anything).

Copy the Design Files to Your Working Directory

In this step, you copy the hardware demonstration files to your working directory. Copy the files using the command:

`cp -r <IP root directory>\altera\altera_dp\hw_demo\<device> <working directory>` ↵

where <*device*> is **sv** for Stratix V devices and **av** for Arria V devices.

Your working directory should contain the files shown in Table 5–2.

Table 5–2. Hardware Demonstration Files (Part 1 of 2)

File Type	File (1)	Description
Verilog HDL design files	<*prefix*>**_dp_demo.v**	Top-level design file.
	dp_mif_mappings.v	Table translating MIF mappings for transceiver reconfiguration.
	dp_analog_mappings.v	Table translating VOD and pre-emphasis settings.
	reconfig_mgmt_hw_ctrl.v	Reconfiguration manager top-level.
	reconfig_mgmt_write.v	Reconfiguration manager FSM for a single write command.
MegaWizard files	<*prefix*>**_video_pll.v** <*prefix*>**_xcvr_pll.v** <*prefix*>**_aux_buffer.v** <*prefix*>**_xcvr_reconfig.v**	MegaWizard variants for the various helper MegaCore functions.
Qsys system	<*prefix*>**_control.qsys**	Qsys system file.
Scripts	**runall.tcl**	Script to set up the project, generate the IP and Qsys system, and compile.
	assignments.tcl	Top-level TCL file to create the project assignments.
Miscellaneous	<*prefix*>**_dp_demo.sdc**	Top-level SDC file.
	edid_memory.hex	Initial content for the EDID ROM.

Table 5–2. Hardware Demonstration Files (Part 2 of 2)

File Type	File (1)	Description
Software files (in the software directory)	**batch_script.sh**	Master script to program the device and build/run the software.
	rerun.sh	Script to rerun the software without rebuilding.
	dp_demo_src	Directory containing the example application source code.
	btc_dprx_syslib	System library for the RX API.
	btc_dptx_syslib	System library for the TX API.

Note:

(1) Files are named with <*prefix*>_<*name*>.<*extension*> where <*prefix*> represents the device (**sv** for Stratix V devices and **av** for Arria V devices).

Build the FPGA Design

In this step you use a script to build and compile the FPGA design. Type the command:

```
quartus_sh -t runall.tcl ↵
```

This script executes the following commands where <*prefix*> is **sv** for Stratix V devices and **av** for Arria V devices:

- Load required packages:

```
load_package flow
load_package misc
```

- Regenerate the IP:

```
qexec "qmegawiz -silent <prefix>_video_pll.v"
qexec "qmegawiz -silent <prefix>_xcvr_pll.v"
qexec "qmegawiz -silent <prefix>_aux_buffer.v"
qexec "qmegawiz -silent <prefix>_xcvr_reconfig.v"
```

- Regenerate the Qsys system:

```
qexec "ip-generate --project-directory=./  \
--output-directory=./<prefix>_control/synthesis/ \
--file-set=QUARTUS_SYNTH \
--report-file=sopcinfo:./<prefix>_control.sopcinfo \
--report-file=html:./<prefix>_control.html \
--report-file=qip:./<prefix>_control/synthesis/<prefix>_control.qip \
--component-file=./<prefix>_control.qsys"
```

- Create the project, overwriting any previous settings files:

```
project_new <prefix>_dp_demo -overwrite
```

- Add the assignments to the project:

```
source assignments.tcl
```

- Run quartus_map to generate a netlist for the DDR pin assignments script:

```
execute_module -tool map
```

- Close the project before running the pin assignment script:

```
project_close
```

- Run the DDR pin assignments script generated by Qsys:

```
qexec "quartus_sta -t ./<prefix>_control/synthesis/submodules/
  <prefix>_control_ddr_p0_pin_assignments.tcl <prefix>_dp_demo"
```

- Re-open the project and do a full compile:

```
project_open <prefix>_dp_demo
```

- Compile the project:

```
execute_flow -compile
```

- Clean up by closing the project:

```
project_close
```

Build, Load, and Run the Software

In this step you build the software, load it into the device, and run the software.

1. In a Windows Command Prompt, navigate to the hardware demonstration **software** directory.
2. Launch a Nios II command shell. You can launch it using several methods, for example, from the Windows task bar or within the Qsys system. To run this command from the Windows Command Prompt, use the command:

```
start "" %SOPC_KIT_NIOS2%\"Nios II Command Shell.bat" ↵
```

3. From within the Nios II command shell execute the following command to build the software, program the device, download the Nios II program, and launch a debug terminal:

```
./batch_script.sh <USB cable number> ↵
```

To find <*USB cable number*>, use the `jtagconfig` command.

The script also creates the **dp_demo**, and **dp_demo_bsp** subdirectories inside the software directory.

If you have already built the software, use the **rerun.sh** script to program the device, download the Nios II program, and launch the terminal:

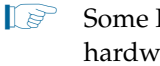
```
./rerun.sh ↵
```

Refer to *Chapter 15: Nios II Software Build Tools Reference* in the Nios II Software Developer's Handbook for a description of the commands used in these scripts.

View the Results

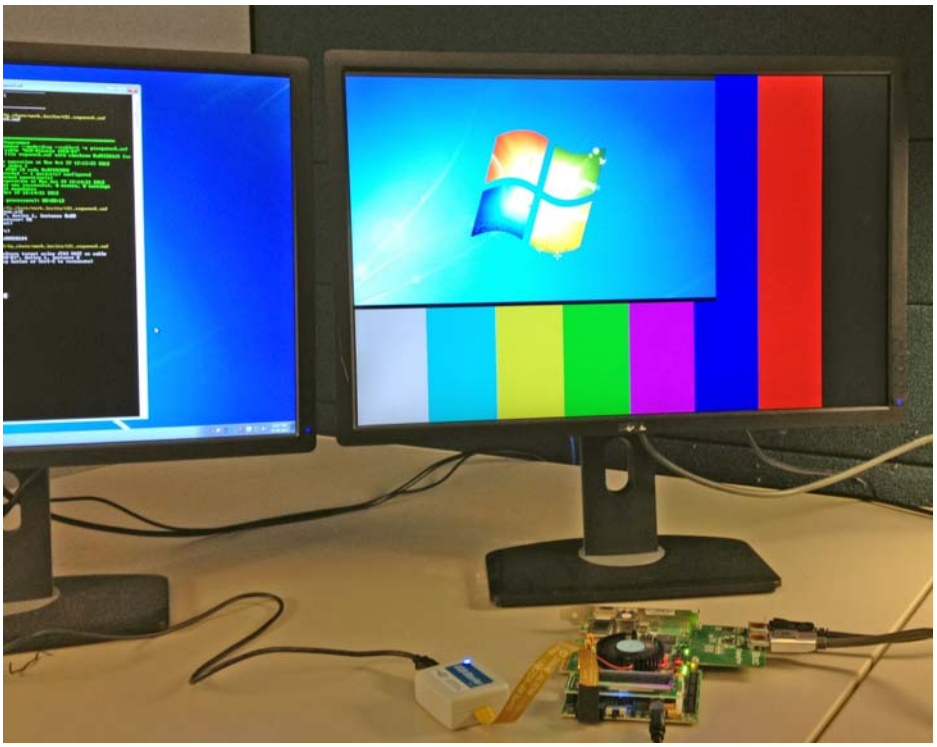
In this step you view the results of the hardware demonstration in the Nios II command shell and on the DisplayPort monitor.

1. Power-up the connected DisplayPort monitor. The hardware demonstration displays a VIP test pattern (color bars).
2. Connect the free end of the Display Port cable that you connected to your PC to the DisplayPort RX on the Bitec HSMC daughter card. The PC now has the DisplayPort monitor available as a second monitor. Depending on your setup, the hardware demonstration displays a VIP test pattern (color bars) mixed with your graphic card output (refer to Figure 5–4).



Some PC drivers and graphic card adapters do not enable the DisplayPort hardware automatically upon hot plug detection. You may need to start the adapter's control utility (e.g., Catalist Control Center, nVidia Control Panel, etc.) and manually enable the DisplayPort display.

Figure 5–4. Color Bars Mixed with Graphic Card Output

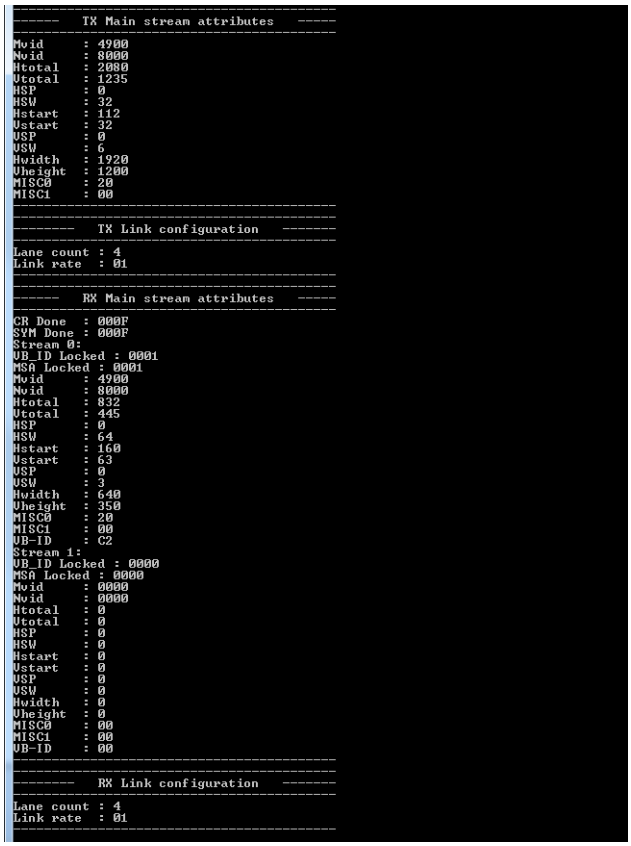


3. Open your graphic card adapter's control utility; it shows a monitor named BITECDP01. Using the control panel, you can adjust the resolution of the BITECDP01 monitor, which typically results in link training, related AUX channel traffic, and a corresponding new image size on the monitor.

If you do not see visible output on the monitor, press pushbutton 2 (USER_PB2 for Stratix V or USER1_PB2 for Arria V) to generate a reset, causing the DisplayPort TX core to re-train the link.

Pressing pushbutton 0 (USER_PB0 for Stratix V or USER1_PB0 for Arria V) retrieves MSA statistics from the source and sink connections. The Nios II Command Shell displays the AUX channel traffic during link training with the monitor. Figure 5–5 shows typical output.

Figure 5–5. MSA Output

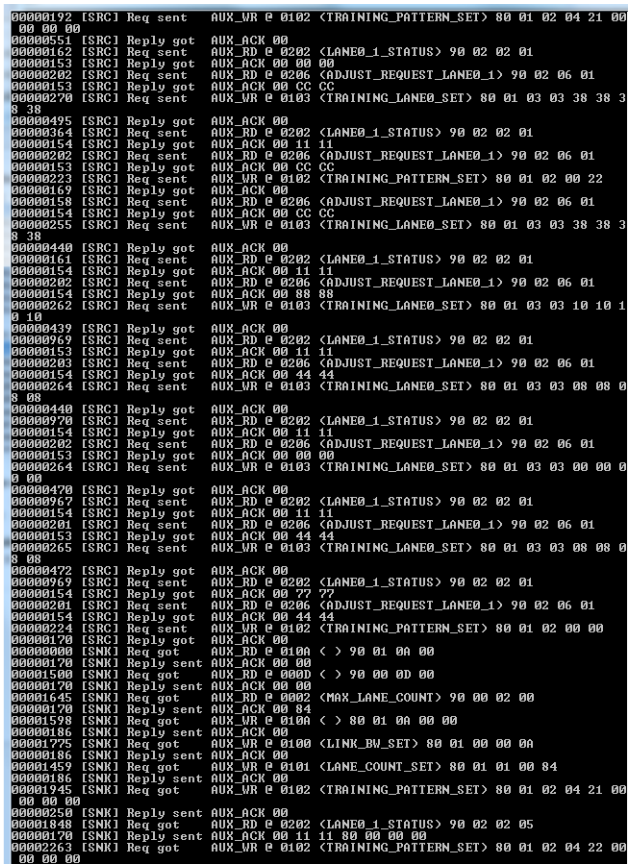


The Nios II AUX printout shows each message packet on a separate line.

- The first field is the incremental timestamp in microseconds.
- The second field indicates whether the message packet is from or to the DisplayPort sink IP core (SNK) or DisplayPort source IP core (SRC).
- The following two fields show the request and response headers and payloads. The DPCD address field on request messages are decoded into their respective DPCD location names.

When connected and enabled, USER_LED_G0 (Stratix V) or USER1_LED_G0 (Arria V) on the development board illuminates to indicate that the DisplayPort receiver has locked correctly. The Nios II terminal also displays the AUX channel traffic related to link training between the graphics adapter and the DisplayPort sink (refer to Figure 5–6).

Figure 5–6. Typical Sink Link Training Output



6. DisplayPort MegaCore Function Simulation Example

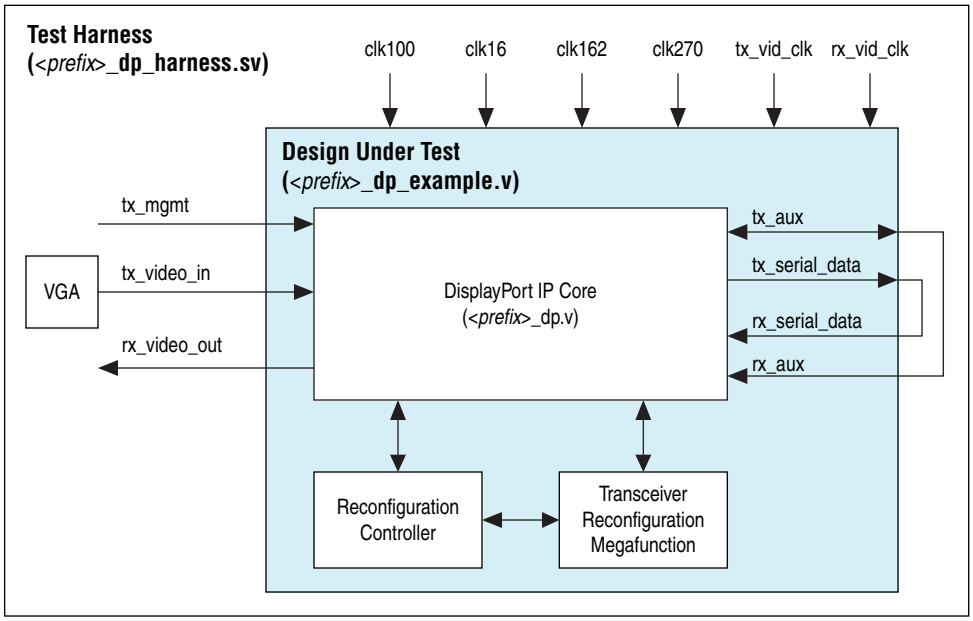
Introduction

The Altera DisplayPort simulation example allows you to evaluate the functionality of the DisplayPort MegaCore function and provides a starting point for you to create your own simulation. This example targets the ModelSim SE simulator.

The simulation example instantiates the DisplayPort IP core with default settings, TX and RX enabled, and 8 bpc. The core has the **Support CTS test automation** option turned on, which is required for the simulation to pass.

The test harness instantiates the design under test (DUT) and a VGA driver. It also generates the clocks and top-level stimulus. The design manipulates the tx_mgmt interface in the main loop to establish a link and send several frames of video data. The test harness checks that the sent data's CRC matches the received data's CRC for three frames. Refer to Figure 6–1.

Figure 6–1. Design Simulation Example Block Diagram *(1)*



Note:

(1) Files are named with <*prefix*>_<*name*>.<*extension*> where <*prefix*> represents the device (**sv** for Stratix V devices and **av** for Arria V devices).

Design Walkthrough

Setting up and running the DisplayPort simulation example consists of the following steps:

1. Copy the simulation files to your target directory.
2. Generate the IP simulation files and scripts, and compile and simulate.
3. View the results.

You use a script to automate these steps as described in the following sections.

Copy the Simulation Files to Your Working Directory

Copy the simulation example files to your working directory using the command:

`cp -r` *<IP root directory>*`\altera\altera_dp\sim_example\`*<device>* *<working directory>* ↵

where *<device>* is **sv** for Stratix V devices and **av** for Arria V devices.

Your working directory should contain the files shown in Table 6–1.

Table 6–1. Simulation Example Files (Part 1 of 2) *(1)*

File Type	File	Description
System Verilog HDL design files	*<prefix>*_**dp_harness.sv**	Top-level test harness.
Verilog HDL design files	*<prefix>*_**dp_example.v**	Design under test (DUT).
	dp_mif_mappings.v	Table translating MIF mappings for transceiver reconfiguration.
	dp_analog_mappings.v	Table translating VOD and pre-emphasis settings.
	reconfig_mgmt_hw_ctrl.v	Reconfiguration manager top-level.
	reconfig_mgmt_write.v	Reconfiguration manager FSM for a single write command.
	clk_gen.v	Clock generation file.
	vga_driver.v	VGA driver.
MegaWizard files	*<prefix>*_**dp.v**	MegaWizard variant for the DisplayPort MegaCore function.
	*<prefix>*_**xcvr_reconfig.v**	MegaWizard variant for the transceiver reconfiguration core.
Scripts	**runall.sh**	This script generates the IP simulation files and scripts, and compiles and simulates them.
	msim_dp.tcl	Compiles and simulates the design in the ModelSim software.
Waveform **.do** files	**all.do**	Waveform that shows a combination of all waveforms.
	reconfig.do	Waveform that shows the signals involved in reconfiguring the transceiver.
	rx_video_out.do	Waveform that shows the rx_video_out signals from the DisplayPort core mapped to CVI input.
	tx_video_in.do	Waveform that shows the `vsync`, `hsync`, `de`, and `vid_data[23:0]` signals at 256 pixels per line, 8 bpp, i. Range of `vid_data_r`, `vid_data_g`, and `vid_data_b` is 00 to ff.

Table 6–1. Simulation Example Files (Part 2 of 2) (1)

File Type	File	Description
Miscellaneous files	**readme.txt**	Documentation for the simulation example.
	edid_memory.hex	Initial content for the EDID ROM.

Note:

(1) Files are named with <*prefix*>_<*name*>.<*extension*> where <*prefix*> represents the device (**sv** for Stratix V devices and **av** for Arria V devices).

Generate the IP Simulation Files and Scripts, and Compile and Simulate

In this step you use a script to generate the IP simulation files and scripts, and compile and simulate them. Type the command:

```
sh runall.sh ↵
```

This script executes the following commands (where <*prefix*> is **sv** for Stratix V devices and **av** for Arria V devices):

- Generate the simulation files for the DisplayPort and transceiver reconfiguration IP cores:
  ```
  qmegawiz -silent <prefix>_xcvr_reconfig.v
  qmegawiz -silent <prefix>_dp.v
  ```
- Merge the two resulting **msim_setup.tcl** scripts to create a single **mentor/msim_setup.tcl**:
  ```
  ip-make-simscript --spd=./<prefix>_xcvr_reconfig.spd
       --spd=./<prefix>_dp.spd
  ```
- Compile and simulate the design in the ModelSim software:
  ```
  vsim -c -do msim_dp.tcl
  ```

The simulation sends several frames of video after reconfiguring the DisplayPort source (TX) and sink (RX) to use the HBR (2.7 G) rate. A successful result is seen by the CTS test automation logic's CRC checks. These checks compare the CRC of the transmitted image with the result measured at the sink. The result is successful if the sink detects three matching frames.

An example successful result is as follows:

```
# SINK CRC_R = 9b40, CRC_G = 9b40, CRC_B = 9b40,
# SOURCE CRC_R = 9b40, CRC_G = 9b40, CRC_B = 9b40,
# Pass: Test Completed
```

View the Results

You can view the results in the ModelSim GUI by loading various **.do** files in the Wave viewer.

1. Launch the ModelSim GUI with the **vsim** command.
2. In the ModelSim Tcl window, execute the **dataset open** command:

   ```
   dataset open vsim.wlf ↵
   ```
3. Choose **View > Open Wave files**.

4. Load the **.do** files to view the waveforms (refer back to Table 6–1 for a listing of the files).

Figure 6–2 shows an example RX reconfiguration waveform. In this example, `rx_link_rate` is set to 1 (HBR). When the core makes a request, the `rx_reconfig_req` port goes high. The user logic asserts `rx_reconfig_ack` and then reconfigures the transceiver. During reconfiguration, the user logic holds `rx_reconfig_busy` high; the user logic drives it low when reconfiguration completes.

Figure 6–2. RX Reconfiguration Waveform

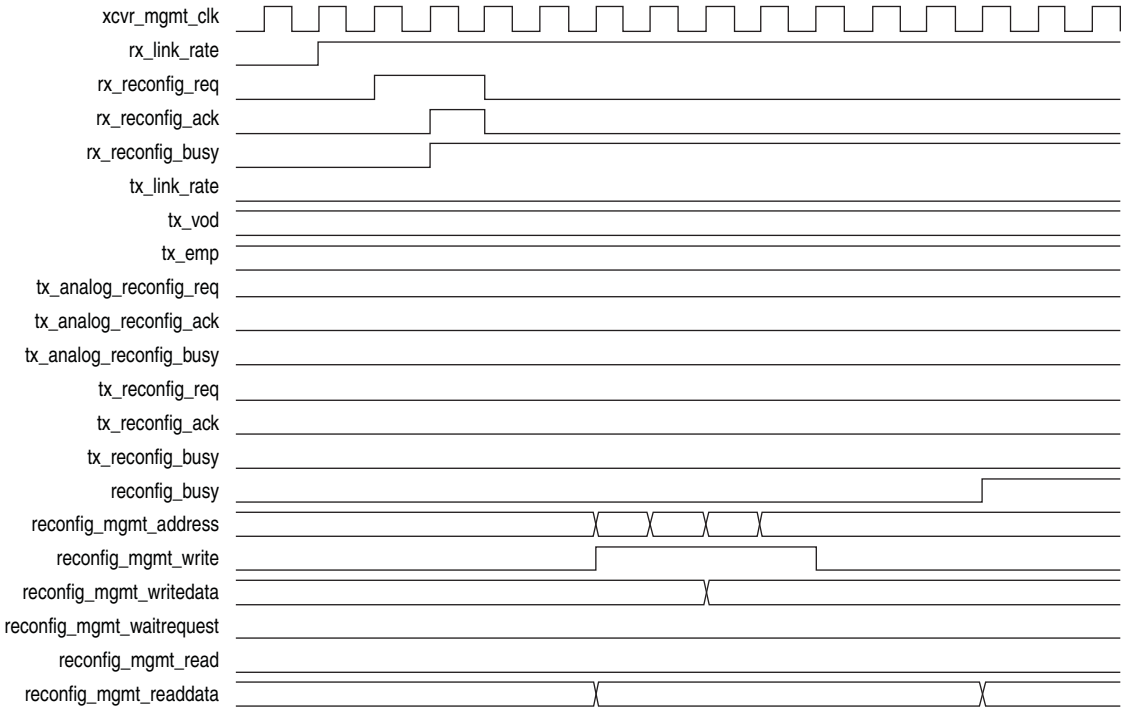


Figure 6–3 shows an example TX reconfiguration waveform. In this example, `tx_link_rate` is set to 1 (HBR). When the core makes a request, the `tx_reconfig_req` port goes high. The user logic asserts `tx_reconfig_ack` and then reconfigures the transceiver. During reconfiguration, the user logic holds `tx_reconfig_busy` high; the user logic drives it low when reconfiguration completes.

Figure 6–3. TX Reconfiguration Waveform

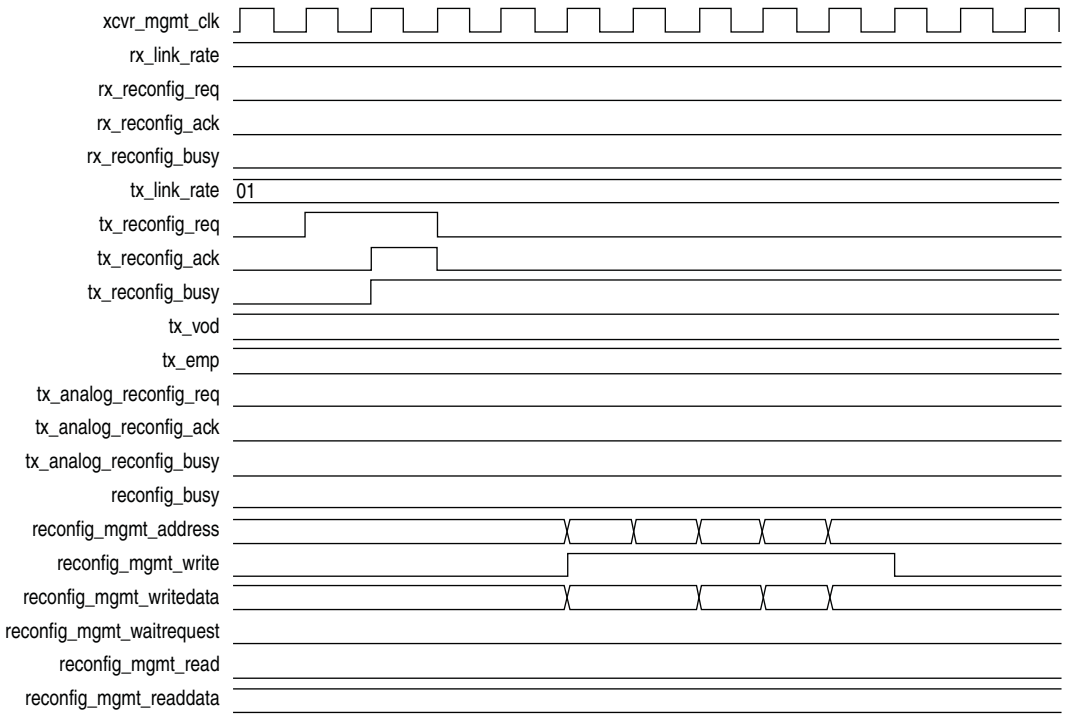


Figure 6–4 shows an example TX analog reconfiguration waveform. In this example, `tx_vod` and `tx_emp` are both set to 00. When the core makes a request, the `tx_analog_reconfig_req` port goes high. The user logic asserts `tx_analog_reconfig_ack` and then reconfigures the transceiver. During reconfiguration, the user logic holds `tx_analog_reconfig_busy` high; the user logic drives it low when reconfiguration completes.

Figure 6–4. TX Analog Reconfiguration Waveform

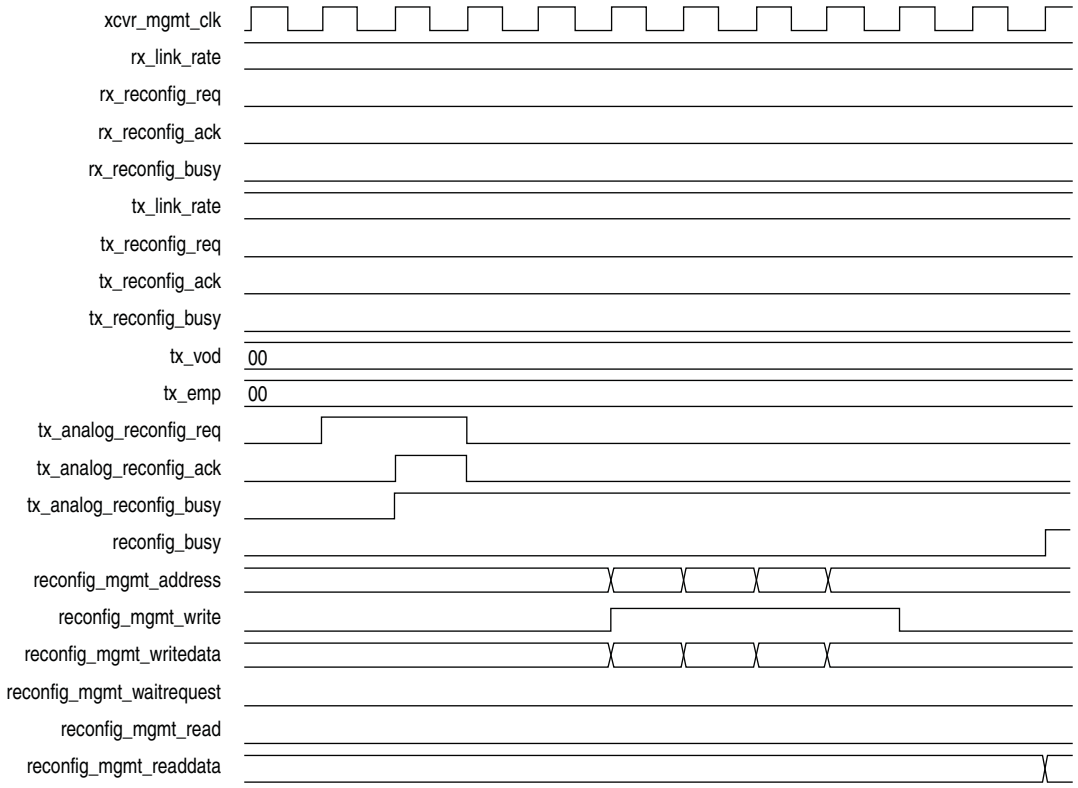
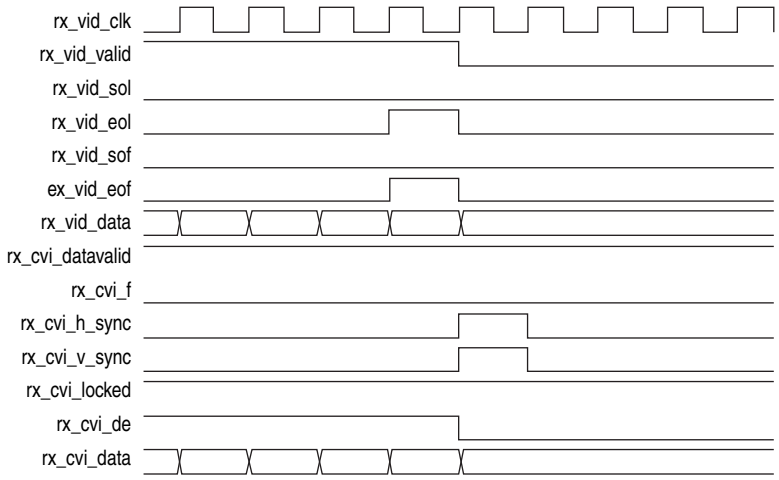


Figure 6–5 shows an example RX video waveform when interfacing to CVI. The `rx_valid_eol` signal generates the `h_sync` pulse by delaying it (by 1 clock cycle) to appear in the horizontal blanking period after the active video ends (VALID is deasserted). The `rx_valid_eof` signal generates the `v_sync` pulse by delaying it (by 1 clock cycle) to appear in the vertical blanking period after the active video ends (VALID is deasserted).

Figure 6–5. RX Video Waveform



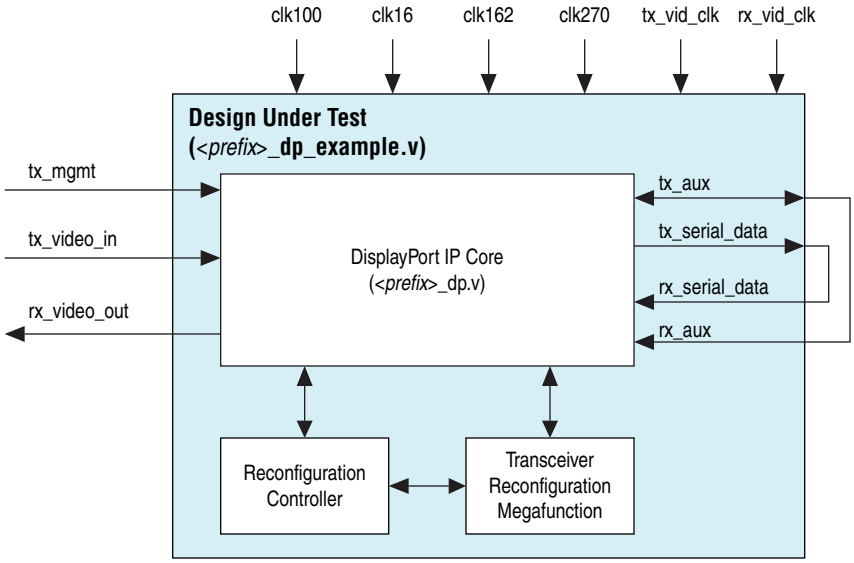
7. DisplayPort MegaCore Function Compilation Example

Introduction

The Altera DisplayPort compilation example allows you to evaluate the timing and resource requirements of the DisplayPort MegaCore function. The compilation example instantiates the DisplayPort IP core with default settings, TX and RX enabled, and 8 bpc. The core has the **Support CTS test automation** option turned on.

Figure 7–1 shows a block diagram of the example.

Figure 7–1. Design Compilation Example Block Diagram



Design Walkthrough

Setting up and running the DisplayPort compilation example consists of the following steps:

1. Copy the compilation files to your target directory.
2. Generate the IP compilation files, compile, and view the results.

You use a script to automate these steps as described in the following sections.

Copy the Compilation Files to Your Working Directory

Copy the compilation example files to your working directory using the command:

```
cp -r <IP root directory>\altera\altera_dp\example\<device> <working directory> ↵
```

where <*device*> is **sv** for Stratix V devices and **av** for Arria V devices. Your working directory should contain the files shown in Table 7–1.

Table 7–1. Compilation Example Files (1)

File Type	File	Description
Verilog HDL design files	<*prefix*>**_dp_example.v**	Design under test (DUT).
	dp_mif_mappings.v	Table translating MIF mappings for transceiver reconfiguration.
	dp_analog_mappings.v	Table translating VOD and pre-emphasis settings.
	reconfig_mgmt_hw_ctrl.v	Reconfiguration manager top-level.
	reconfig_mgmt_write.v	Reconfiguration manager FSM for a single write command.
MegaWizard files	<*prefix*>**_dp.v**	MegaWizard variant for the DisplayPort MegaCore function.
	<*prefix*>**_xcvr_reconfig.v**	MegaWizard variant for the transceiver reconfiguration core.
Scripts	**runall.tcl**	This script generates the IP compilation files and scripts, and compiles and simulates them.
	assignments.tcl	This script defines the project settings.
Miscellaneous files	**readme.txt**	Documentation for the compilation example.

Note:

(1) Files are named with <*prefix*>_<*name*>.<*extension*> where <*prefix*> represents the device (**sv** for Stratix V devices and **av** for Arria V devices).

Generate the IP Compilation Files, Compile, and View thhhhhhe Results

Use the following command to generate the IP compilation files and compile them:

```
quartus_sh -t runall.tcl ↵
```

This script executes the following commands:

- Load required packages:

```
load_package flow
load_package misc
```

- Regenerate the IP:

```
qexec "qmegawiz -silent <prefix>_xcvr_reconfig.v"
qexec "qmegawiz -silent <prefix>_dp.v"
```

- Create the project overwriting any previous settings files:

```
project_new <prefix>_dp_example -overwrite
```

- Aadd the assignments to the project:

```
source assignments.tcl
```

- Compile the project:

```
execute_flow -compile
```

- Clean up by closing the project:

```
project_close
```

View the compilation results in the Quartus II GUI.

You can use the DisplayPort MegaCore function to instantiate sources and sinks. Source instantiations require an embedded controller (Nios II processor or another controller) to act as the policy maker. Sink instantiations greatly benefit from and may optionally use a controller.

Altera provides software for source and sink instantiations as two C system libraries (`btc_dptx_syslib` and `btc_dprx_syslib`, respectively). The IP core includes an example main program (`dp_demo`), which demonstrates basic system library use.

Using the Library

Figure 8–1 describes a typical user application flow. The user application must initialize the library as its first operation. Next, the application should initialize the instantiated devices (sink and/or source), partly in the `btc_dptx_syslib` and `btc_dprx_syslib` data structures and partly in the user application. You must also implement ISRs to handle interrupts generated by the DisplayPort core.

When initialization completes, the user application should periodically invoke the library monitoring function.

Figure 8–1. Typical User Application Flow

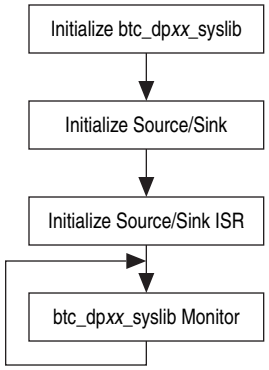
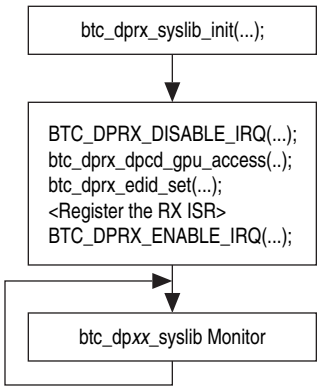


Figure 8–2 shows a sink application. The user application must initialize the DPCD, the EDID content, and register the RX interrupt ISR. For a source application, it only has to register the TX interrupt ISR.

Figure 8–2. Typical Sink User Application Library Calls



Sink instantiations issue an interrupt to the controller when it receives an AUX channel request from the connected source. Example 8–1 shows a typical ISR implementation.

Example 8–1. Typical Sink ISR Implementation

```
BTC_DPRX_DISABLE_IRQ(...);
btc_dprx_aux_get_request(0,&cmd,&address,&length,data);
btc_dprx_aux_handler(0,cmd,address,length,data);
BTC_DPRX_ENABLE_IRQ(…);
```

Source instantiations issue an interrupt to the controller when the source detects a logic state change on the connected DisplayPort sink's HPD signal. Example 8–2 shows a typical ISR implementation.

Example 8–2. Typical Source ISR Implementation

```
if (HPD asserted)
{
    <read Sink EDID>
    <set video output resolution>
    btc_dptx_link_training(...);
}
else if (HPD deasserted)
    btc_dptx_video_enable(..., 0);
else if (IRQ_HPD)
{
    <check link status>
    if (Test Automation request)
    btc_dptx_test_autom(…);
}
```

btc_dprx_syslib API Reference

This section provides information on the DisplayPort sink system library functions (`btc_dprx_syslib`), including:

- C prototype
- Function description
- Whether the function is thread-safe when running in a multi- threaded environment
- Whether the function can be invoked from an ISR
- Example

btc_dprx_aux_get_request

Prototype:
```
int btc_dprx_aux_get_request(
     BYTE           *cmd,
     unsigned int   *address,
     BYTE           *length,
     BYTE           *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`cmd`	Pointer to command
`address`	Pointer to address
`length`	Pointer to length (0 - 16)
`data`	Pointer to data received

Description: This function retrieves an AUX channel request issued by the connected DisplayPort source. The command and address are the command byte and the address in the original request received, respectively (refer to the DisplayPort specification for more details). When the request is a write, `*data` fills with the data bytes sent by the source. To support address-only requests, `length` is the original `len` byte sent by the source incremented by one.

Example:
```
btc_dprx_aux_get_request(pcmd, padd, plen, pwrdata);
```

See also: btc_dprx_aux_handler

btc_dprx_aux_handler

Prototype:
```
int btc_dprx_aux_handler(
     BYTE          cmd,
     unsigned int  address,
     BYTE          length,
     BYTE          *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`cmd`	Command
`address`	Address
`length`	Length (0 - 16)
`data`	Pointer to data being written

Description: This function processes an AUX channel request issued by the connected DisplayPort source. The command and address are the command byte and the address in the original request received, respectively (refer to the DisplayPort specification for more details). When the request is a write, `data` must point to the data bytes sent by the source. To support address-only requests, `length` is the original `len` byte sent by the source incremented by one. When the request is a read, `data` is not used and can be NULL.

This function provides all the functionality of the DPCD registers implemented inside the system library, including:

- DPCD locations read/write support
- EDID read support
- Link training execution
- Forwarding of AUX channel replies back to the source

Example:
```
btc_dprx_aux_handler(pcmd, padd, plen, pwrdata);
```

See also: btc_dprx_aux_get_request

btc_dprx_aux_post_reply

Prototype:
```
int btc_dprx_aux_post_reply(
     BYTE           cmd,
     BYTE           size,
     BYTE           *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`cmd`	Command
`size`	Number of data bytes transmitted (0 - 16)
`data`	Pointer to data transmitted

Description: This function transmits an AUX channel reply to the connected DisplayPort source. `command` is the reply command byte (refer to the DisplayPort specification for more details). When the reply includes read data, `*data` fills with the data bytes sent to the source. To support replies with no data returned, `size` is the actual `len` byte sent to the source incremented by one.

Example:
```
btc_dprx_aux_post_reply (0x10, 0, NULL); //Reply AUX_NACK
```

See also: btc_dprx_aux_get_request

btc_dprx_baseaddr

Prototype:	`unsigned int btc_dprx_baseaddr(void)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	`<btc_dprx_syslib.h>`
Return:	0 = success, 1 = fail
Description:	This function returns the RX instance's base address connected to the given port number.
Example:	`addr = btc_dprx_baseaddr();`

btc_dprx_dpcd_gpu_access

Prototype:
```
int btc_dprx_dpcd_gpu_access(
     BYTE          wrcmd,
     unsigned int  address,
     BYTE          length,
     BYTE          *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`wrcmd`	0 = read 1 = write
`address`	Address
`length`	Length (1 - 255)
`data`	Pointer to data

Description: This function allows the controller to access the sink's DPCD locations (implemented in the system library) for reading/writing data. `data` must point to a location containing `length` bytes (writes) or be able to accommodate `length` bytes (reads).

Example:
```
btc_dprx_dpcd_gpu_access(1, 0x00000, 1, pwrdata);
```

btc_dprx_edid_set

Prototype:
```
int btc_dprx_edid_set(
     BYTE          port,
     BYTE          *edid_data,
     BYTE          num_blocks)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`port`	RX port (stream) number (0 – 3)
`edid_data`	Pointer to data to initialize the EDID
`num_blocks`	EDID size in blocks

Description: This function allows the controller to set the content of the sink's EDID (implemented in the system library). The IP core copies data pointed to by `edid_data` into the library EDID. One block is 128-bytes long. The system library accepts a maximum of 4 blocks (512-byte long EDIDs). Each streaming sink port has its own EDID.

Example:
```
btc_dprx_edid_set(0, pmy_edid, 2);
```

btc_dprx_hpd_get

Prototype:	`int btc_dprx_hpd_get(void)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dprx_syslib.h>`
Return:	0 = success, 1 = fail
Parameters:	`level` 0 or 1
Description:	Returns the current logic level of the RX HPD.
Example:	`btc_dprx_hpd_get();`
See also:	btc_dprx_hpd_set btc_dprx_hpd_pulse

btc_dprx_hpd_pulse

Prototype:	`void btc_dprx_hpd_pulse(void)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dprx_syslib.h>`
Return:	–
Description:	This function deasserts (i.e., sets to 0) the RX HPD for 750 μs. You can use this function to send an `IRQ_HPD` pulse to the connected DisplayPort source. Before invoking this function, you must have invoked `btc_dprx_hpd_set` with level = 1 (i.e., HPD must be set to 1).
Example:	`btc_dprx_pulse(0);`
See also:	btc_dprx_hpd_set btc_dprx_hpd_get

btc_dprx_hpd_set

Prototype:	`void btc_dprx_hpd_set(` `  int  level)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dprx_syslib.h>`
Return:	–
Parameters:	`level` 0 or 1
Description:	This function allows the controller to set the logic level of its HPD.
Example:	`btc_dprx_hpd_set(1);`
See also:	btc_dprx_hpd_get btc_dprx_hpd_pulse

btc_dprx_syslib_init

Prototype:
```
int btc_dprx_syslib_init(
     unsigned int  rx_base_addr,
     unsigned int  rx_irq_id,
     unsigned int  rx_irq_num,
     unsigned int  rx_num_of_sinks)
```

Thread-safe: No

Available from ISR: No

Include: `<btc_dprx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`rx_base_addr`	RX base address
`rx_irq_id`	RX IRQ ID
`rx_irq_num`	RX IRQ number
`rx_num_of_sinks`	Number of streaming sinks used (1 - 4)

Description: This function initializes the system library. It should be invoked as the first function in the library by `main()`.

Example:
```
btc_dprx_syslib_init
     (BITEC_DP_0_AV_RX_CONTROL_BASE,
     BITEC_DP_0_AV_RX_CONTROL_IRQ_INTERRUPT_CONTROLLER_ID,
     BITEC_DP_0_AV_RX_CONTROL_IRQ,
     2);
```

btc_dprx_syslib_monitor

Prototype:	`int btc_dprx_syslib_monitor(void)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dprx_syslib.h>`
Return:	0 = success, 1 = fail
Description:	This function calls the system library sink housekeeping monitor, which is responsible for: ■ Handling RX-side received sideband message requests. ■ Forwarding RX-side sideband message replies. The software should invoke this function periodically or at least every 50 ms.
Example:	`btc_dprx_syslib_monitor();`

btc_dptx_syslib API Reference

This section provides information on the DisplayPort source system library functions (`btc_dptx_syslib`), including:

- C prototype
- Function description
- Whether the function is thread-safe when running in a multi- threaded environment
- Whether the function can be invoked from an ISR
- Example

btc_dptx_aux_i2c_read

Prototype:
```
int btc_dptx_aux_i2c_read(
     BYTE          address,
     BYTE          size,
     BYTE          *data,
     BYTE          mot)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`address`	I^2C address
`size`	Number of bytes (1 - 16)
`data`	Pointer to data to be read
`mot`	Middle of transaction (0 or 1)

Description: This function reads 1 to 16 data bytes from the connected DisplayPort sink's I^2C interface mapped over the AUX channel.

Example:
```
btc_dptx_aux_i2c_read(0x50, 16, data, 1);
```

See also: btc_dptx_aux_i2c_write

btc_dptx_aux_i2c_write

Prototype:
```
int btc_dptx_aux_i2c_write(
     BYTE          address,
     BYTE          size,
     BYTE          *data,
     BYTE          mot)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`address`	I^2C address
`size`	Number of bytes (1 - 16)
`data`	Pointer to data to be written
`mot`	Middle of transaction (0 or 1)

Description: This function writes 1 to 16 data bytes from the connected DisplayPort sink's I^2C interface mapped over the AUX channel.

Example:
```
btc_dptx_aux_i2c_write(0x50, 1, data, 1);
```

See also: btc_dptx_aux_i2c_read

btc_dptx_aux_read

Prototype:
```
int btc_dptx_aux_read(
     unsigned int  address,
     BYTE          size,
     BYTE          *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return:
- 0 = AUX_ACK replied
- 1 = Source internal error
- 2 = Reply timeout
- 3 = AUX_NACK replied
- 4 = AUX_DEFER replied
- 5 = Invalid reply

Parameters

`address`	DPCD start address
`size`	Number of bytes (1 - 16)
`data`	Pointer for data to be read

Description: This function reads 1 to 16 data bytes from the connected DisplayPort sink's DPCD.

Example:
```
btc_dptx_aux_read(0x202, 2, &status);
```

See also: btc_dptx_aux_write

btc_dptx_aux_write

Prototype:
```
int btc_dptx_aux_write(
     unsigned int  address,
     BYTE          size,
     BYTE          *data)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return:
- 0 = AUX_ACK replied
- 1 = Source internal error
- 2 = Reply timeout
- 3 = AUX_NACK replied
- 4 = AUX_DEFER replied
- 5 = Invalid reply

Parameters

`address`	DPCD start address
`size`	Number of bytes (1 - 16)
`data`	Pointer to data to be written

Description: This function writes 1 to 16 data bytes to the connected DisplayPort sink's DPCD.

Example:
```
btc_dptx_aux_write(0x600, 1, data_ptr);
```

See also: btc_dptx_aux_read

btc_dptx_baseaddr

Prototype:	`unsigned int btc_dptx_baseaddr(void)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	`<btc_dptx_syslib.h>`
Return:	0 = success, 1 = fail
Description:	This function returns the base address of the TX instance connected to the given port number.
Example:	`addr = btc_dptx_baseaddr();`

btc_dptx_edid_block_read

Prototype:	`int btc_dptx_edid_block_read(`
	`  BYTE  block,`
	`  BYTE  *data)`

Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dptx_syslib.h>`
Return:	0 = success, 1 = fail
Parameters:	`block` Block number (0 - 3)
	`data` Pointer for data to be read
Description:	Reads one block (128 bytes) from the EDID of the connected DisplayPort Sink.
Example:	`btc_dptx_edid_block_read(2, pdata);`
See also:	btc_dptx_edid_read

btc_dptx_edid_read

Prototype:	`int btc_dptx_edid_read(BYTE *data)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dptx_syslib.h>`
Return:	0 = success, 1 = fail
Parameters:	`data` Pointer for data to be read
Description:	This function reads the complete EDID of the connected DisplayPort sink. `data` must be able to contain the whole EDID (allow for 512 bytes).
Example:	`btc_dptx_edid_read(pdata);`
See also:	btc_dptx_edid_block_read

btc_dptx_fast_link_training

Prototype:
```
int btc_dptx_fast_link_training(
     unsigned int  link_rate,
     unsigned int  lane_count,
     unsigned int  volt_swing,
     unsigned int  pre_emph,
     unsigned int  new_cfg)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`link_rate`	Link rate (Gbps): 0 = 1.62 1 = 2.70 2 = 5.40
`lane_count`	1, 2, or 4
`volt_swing`	0, 1, 2, or 3
`pre_emph`	0, 1, 2, or 3
`new_cfg`	0 = ignore the other parameters 1 = use provided parameters

Description: This function performs fast link training with the connected DisplayPort sink. When performing fast link training, the IP core outputs training pattern 1 for 1 ms followed by training pattern 2 for 1 ms.

The function returns a 1 if link training fails or if the DPCD flag `NO_AUX_HANDSHAKE_LINK_TRAINING` = 0 (at location 00103h).

- If `new_cfg` = 1, the IP core updates the sink's DPCD with the provided `link_rate` and `lane_count`, sets its own transceiver with the provided `volt_swing` and `pre_emph`, and then performs fast link training.
- If `new_cfg` = 0, the IP core uses the current transceiver setting, link rate, and lane count, and performs fast link training.

Example:
```
btc_dptx_fast_link_training(1, 4, 1, 0, 1);
```

See also: btc_dptx_link_training

btc_dptx_link_training

Prototype:	`int btc_dptx_link_training(`
	`  unsigned int  link_rate,`
	`  unsigned int  lane_count)`	

Thread-safe:	No	
Available from ISR:	Yes	
Include:	`<btc_dptx_syslib.h>`	
Return:	0 = success, 1 = fail	
Parameters:	`link_rate`	Link rate (Gbps): 0 = 1.62 1 = 2.70 2 = 5.40
	`lane_count`	1, 2, or 4
Description:	This function performs link training with the connected DisplayPort sink.	
Example:	`btc_dptx_link_training(1, 4);`	

btc_dptx_set_color_space

Prototype:
```
int btc_dptx_set_color_space(
     BYTE          format,
     BYTE          bpc,
     BYTE          range,
     BYTE          colorimetry)
```

Thread-safe: No

Available from ISR: Yes

Include: `<btc_dptx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`format`	0 = RGB
	1 = YCbCr 4:2:2
	2 = YCbCr 4:4:4
`bpc`	Color depth (bpc):
	0 = 6
	1 = 8
	2 = 10
	3 = 12
	4 = 16
`range`	0 = VESA
	1 = CEA
`colorimetry`	0 = BT601-5
	1 = BT709-5

Description: This function sets the color space for TX transmitted video.

Example:
```
btc_dptx_set_color_space(0, 1, 0, 0);
```

btc_dptx_syslib_init

Prototype:
```
int btc_dptx_syslib_init(
     unsigned int  tx_base_addr,
     unsigned int  tx_irq_id,
     unsigned int  tx_irq_num)
```

Thread-safe: No

Available from ISR: No

Include: `<btc_dptx_syslib.h>`

Return: 0 = success, 1 = fail

Parameters:

`tx_base_addr`	TX base address
`tx_irq_id`	TX IRQ ID
`tx_irq_num`	TX IRQ number

Description: Initializes the system library. Should be invoked as the first function in the library by `main()`. Set the base address of TX or RX to `BTC_NOT_PRESENT` if TX or RX not instantiated.

Example:
```
btc_dptx_syslib_init(
     BITEC_DP_0_AV_TX_CONTROL_BASE,
     BITEC_DP_0_AV_TX_CONTROL_IRQ_INTERRUPT_CONTROLLER_ID,
     BITEC_DP_0_AV_TX_CONTROL_IRQ);
```

btc_dptx_syslib_monitor

Prototype:	`int btc_dptx_syslib_monitor(void)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dptx_syslib.h>`
Return:	0 = success, 1 = fail
Description:	This function calls the system library source housekeeping monitor. The software should invoke this function periodically or at least every 50 ms.
Example:	`btc_dptx_syslib_monitor();`

btc_dptx_test_autom

Prototype:	`int btc_dptx_test_autom(void)`
Thread-safe:	No
Available from ISR:	Yes
Include:	`<btc_dptx_syslib.h>`
Return:	0 = success, 1 = fail
Description:	This function handles test automation requests from the connected DisplayPort sink. You should invoke this function after the IP core senses an HPD IRQ and identifies it as a test automation request. The function implements `TEST_LINK_TRAINING` and `TEST_EDID_READ`.
Example:	`btc_dptx_test_autom();`

btc_dptx_video_enable

Prototype:	`int btc_dptx_video_enable(BYTE enabled)`	
Thread-safe:	No	
Available from ISR:	Yes	
Include:	`<btc_dptx_syslib.h>`	
Return:	0 = success, 1 = fail	
Parameters:	`enabled`	0 = output idle pattern 1 = output active video
Description:	This function enables the TX to output either active video or an idle pattern. After successful link training, the TX outputs active video by default.	
Example:	`btc_dptx_video_enable(1);`	

btc_dpxx_syslib Additional Types

In addition to the standard ANSI C defined types, `btc_dpxx_syslib` uses the following types:

- `#define BYTE unsigned char`

btc_dprx_syslib Supported DPCD Locations

Refer to "Sink-Supported DPCD Locations" on page 10–21 for a list of DPCD locations currently supported in `btc_dprx_syslib` sink instantiations. Read accesses to unsupported locations are replied with `NATIVE_ACK` and data content is set to zero. Write accesses to unsupported locations are replied with `NATIVE_NACK`.

9. DisplayPort Source Register Map

DisplayPort source instantiations require an embedded controller (Nios II processor or another controller) to act as the policy maker. This section describes the register map. Table 9–1 describes the notation used in this section.

Table 9–1. Notation

Shorthand	Definition
RW	Read/write
RO	Read only
WO	Write only
CRO	Clear on read or write, read only
CWO	Clear on read or write, write only

General Registers

This section describes the general registers.

DPTX_TX_CONTROL

Table 9–2 shows the register for the TX control bits. Setting LANE_COUNT to 00000 causes the transmitter GXB to always send a logical zero (i.e., a constant voltage level). You can use this function to cause a "power down" for link layer compliance testing.

Address: 0x0000

Direction: RW

Reset: 0x00000004

Table 9–2. DPTX_TX_CONTROL Bits (Part 1 of 2)

Bit	Bit Name	Function
31	IRQ_EN	Enables an IRQ issued to the Nios II processor on an HPD event: 0 = disable 1 = enable
30:21		Unused
20	RESERVED	Reserved
19	ENHANCED_FRAME	0 = Standard framing 1 = Enhanced framing
18:17	TX_LINK_RATE	Main link rate: 0 = 1.62 Gbps 1 = 2.7 Gbps 2 = 5.4 Gbps
16:15		Unused

Table 9–2. DPTX_TX_CONTROL Bits (Part 2 of 2)

Bit	Bit Name	Function
14	ASYNC_CLOCK	0 = Synchronous 1 = Asynchronous
13	COLORIMETRY	0 = ITU-R BT601-5 1 = ITU-R BT709-5
12	DYNAMIC_RANGE	0 = VESA (from 0 to maximum) 1 = CEA range
11:10	COMPONENT_FORMAT	00 = RGB 01 = YCbCr 4:2:2 10 = YCbCr 4:4:4 11 = Reserved
9:5	LANE_COUNT	Lane count: 00000 = Reserved 00001 = 1 00010 = 2 00100 = 4
4:2	BPP	Bits-per-pixel format: 000 = 6 bpc 001 = 8 bpc 010 = 10 bpc 011 = 12 bpc 100 = 16 bpc
1:0	TP	Current training pattern: 00 = Normal video 01 = Training pattern 1 10 = Training pattern 2 11 = Video idle pattern

DPTX_TX_STATUS

Table 9–3 shows the register for the TX status bits. The IP core issues an IRQ to the Nios II processor if the DPTX_TX_CONTROL registers IRQ_EN is 1 and the IP core detects a new HPD event. HPD_EVENT provides information about the event that caused the interrupt. The interrupt and HPD_EVENT bit fields are both cleared by reading the DPTX_TX_STATUS register.

- 01 = HPD plug event (long HPD)
- 10 = HPD IRQ (short HPD)
- 11 = Reserved

Address: 0x0001

Direction: CRO

Reset: 0x00000000

Table 9–3. DPTX_TX_STATUS Bits

Bit	Bit Name	Function
31:4		Unused
3	RESERVED	Reserved
2	HPD_LEVEL	Current HPD logic level
1:0	HPD_EVENT	HPD event causing IRQ (read to clear): 00 = No event

MSA Registers

This section describes the MSA registers.

DPTX_MSA_MVID

Table 9–4 shows the register for the DPTX_MSA_MVID bits.

Address: 0x0002

Direction: RO

Reset: 0x00000000

Table 9–4. DPTX_MSA_MVID Bits

Bit	Bit Name	Function
31:24		
23:0	MVID	Main stream attribute MVID

DPTX_MSA_NVID

Table 9–5 shows the register for the DPTX_MSA_NVID bits.

Address: 0x0003

Direction: RO

Reset: 0x00000000

Table 9–5. DPTX_MSA_NVID Bits

Bit	Bit Name	Function
31:24		
23:0	NVID	Main stream attribute NVID

DPTX_MSA_HTOTAL

Table 9–6 shows the register for the DPTX_MSA_HTOTAL bits.

Address: 0x0004

Direction: RO

Reset: 0x00000000

Table 9–6. DPTX_MSA_HTOTAL Bits

Bit	Bit Name	Function
31:16		
15:0	HTOTAL	Main stream attribute HTOTAL

DPTX_MSA_VTOTAL

Table 9–7 shows the register for the DPTX_MSA_VTOTAL bits.

Address: 0x0005

Direction: RO

Reset: 0x00000000

Table 9–7. DPTX_MSA_VTOTAL Bits

Bit	Bit Name	Function
31:16		
15:0	MVID	Main stream attribute VTOTAL

DPTX_MSA_HSP

Table 9–8 shows the register for the DPTX_MSA_HSP bits.

Address: 0x0006

Direction: RO

Reset: 0x00000000

Table 9–8. DPTX_MSA_HSP Bits

Bit	Bit Name	Function
31:1		
0	HSP	Main stream attribute horizontal sync polarity: 0 = Positive 1 = Negative

DPTX_MSA_HSW

Table 9–9 shows the register for the DPTX_MSA_HSW bits.

Address: 0x0007

Direction: RO

Reset: 0x00000000

Table 9–9. DPTX_MSA_HSW Bits

Bit	Bit Name	Function
31:15		
14:0	HSW	Main stream attribute horizontal sync width

DPTX_MSA_HSTART

Table 9–10 shows the register for the DPTX_MSA_HSTART bits.

Address: 0x0008

Direction: RO

Reset: 0x00000000

Table 9–10. DPTX_MSA_HSTART Bits

Bit	Bit Name	Function
31:16		
15:0	HSTART	Main stream attribute HSTART

DPTX_MSA_VSTART

Table 9–11 shows the register for the DPTX_MSA_VSTART bits.

Address: 0x0009

Direction: RO

Reset: 0x00000000

Table 9–11. DPTX_MSA_VSTART Bits

Bit	Bit Name	Function
31:16		
15:0	VSTART	Main stream attribute VSTART

DPTX_MSA_VSP

Table 9–12 shows the register for the DPTX_MSA_VSP bits.

Address: 0x000a

Direction: RO

Reset: 0x00000000

Table 9–12. DPTX_MSA_VSP Bits

Bit	Bit Name	Function
31:1		
0	VSP	Main stream attribute vertical sync polarity 0 = Positive 1 = Negative

DPTX_MSA_VSW

Table 9–13 shows the register for the DPTX_MSA_VSW bits.

Address: 0x000b

Direction: RO

Reset: 0x00000000

Table 9–13. DPTX_MSA_VSW Bits

Bit	Bit Name	Function
31:15		
14:0	VSW	Main stream attribute vertical sync width

DPTX_MSA_HWIDTH

Table 9–14 shows the register for the DPTX_MSA_HWIDTH bits.

Address: 0x000c

Direction: RO

Reset: 0x00000000

Table 9–14. DPTX_MSA_HWIDTH Bits

Bit	Bit Name	Function
31:16		
15:0	HWIDTH	Main stream attribute HWIDTH

DPTX_MSA_VHEIGHT

Table 9–15 shows the register for the DPTX_MSA_VHEIGHT bits.

Address: 0x000d

Direction: RO

Reset: 0x00000000

Table 9–15. DPTX_MSA_VHEIGHT Bits

Bit	Bit Name	Function
31:16		
15:0	VHEIGHT	Main stream attribute VHEIGHT

DPTX_MSA_MISC0

Table 9–16 shows the register for the DPTX_MSA_MISC0 bits.

Address: 0x000e

Direction: RO

Reset: 0x00000000

Table 9–16. DPTX_MSA_MISC0 Bits

Bit	Bit Name	Function
31:8		
7:0	MISC0	Main stream attribute MISC0

DPTX_MSA_MISC1

Table 9–17 shows the register for the DPTX_MSA_MISC1 bits.

Address: 0x000f

Direction: RO

Reset: 0x00000000

Table 9–17. DPTX_MSA_MISC1 Bits

Bit	Bit Name	Function
31:8		
7:0	MISC1	Main stream attribute MISC1

Link Voltage and Pre-Emphasis Controls

This section describes the registers for the link voltage and pre-emphasis controls.

DPTX_PRE_VOLT0

These ports drive the respective gxb_reconfig_conduit ports. Table 9–18 shows the register for the DPTX_PRE_VOLT0 bits.

Address: 0x0010

Direction: RW

Reset: 0x00000000

Table 9–18. DPTX_PRE_VOLT0 Bits

Bit	Bit Name	Function
31:4		Unused
3:2	PRE0	Pre-emphasis output on lane 0
1:0	VOLT0	Voltage swing output on lane 0

DPTX_PRE_VOLT1

These ports drive the respective gxb_reconfig_conduit ports. Table 9–19 shows the register for the DPTX_PRE_VOLT1 bits.

Address: 0x0011

Direction: RW

Reset: 0x00000000

Table 9–19. DPTX_PRE_VOLT1 Bits

Bit	Bit Name	Function
31:4		Unused
3:2	PRE1	Pre-emphasis output on lane 1
1:0	VOLT1	Voltage swing output on lane 1

DPTX_PRE_VOLT2

These ports drive the respective gxb_reconfig_conduit ports. Table 9–20 shows the register for the DPTX_PRE_VOLT2 bits.

Address: 0x0012

Direction: RW

Reset: 0x00000000

Table 9–20. DPTX_PRE_VOLT2 Bits

Bit	Bit Name	Function
31:4		Unused
3:2	PRE2	Pre-emphasis output on lane 2
1:0	VOLT2	Voltage swing output on lane 2

DPTX_PRE_VOLT3

These ports drive the respective gxb_reconfig_conduit ports. Table 9–21 shows the register for the DPTX_PRE_VOLT3 bits.

Address: 0x0013

Direction: RW

Reset: 0x00000000

Table 9–21. DPTX_PRE_VOLT3 Bits

Bit	Bit Name	Function
31:4		Unused
3:2	PRE3	Pre-emphasis output on lane 3
1:0	VOLT3	Voltage swing output on lane 3

DPTX_DO_CONFIG

CONFIG drives the gxb_reconfig_conduit[0] port. Table 9–22 shows the register for the DPTX_DO_CONFIG bits.

Address: 0x0014

Direction: RW

Reset: 0x00000000

Table 9–22. DPTX_DO_CONFIG Bits

Bit	Bit Name	Function
31:1		Unused
0	CONFIG	0 = Keep current GXB configuration 1 = Reconfigure GXB with values in DPTX_PRE_VOLT0-3

Link Quality Pattern Generation Register

Table 9–23 shows the register for the DPTX_LINK_QUALITY_PATTERN bits. If you use this register, you can force the source to output a standard PRBS7 test pattern on each of the selected lanes, which can be useful for performing BER measurements.

Address: 0x0015

Direction: RW

Reset: 0x00000000

Table 9–23. DPTX_LINK_QUALITY_PATTERN Bits

Bit	Bit Name	Function
31:12		Unused
11:9	LQ_PATT_LANE3	Pattern selection for lane 3: 000 = No training pattern (normal mode) 011 = PRBS7 transmitted
8:6	LQ_PATT_LANE2	Pattern selection for lane 2: 000 = No training pattern (normal mode) 011 = PRBS7 transmitted
5:3	LQ_PATT_LANE1	Pattern selection for lane 1: 000 = No training pattern (normal mode) 011 = PRBS7 transmitted
2:0	LQ_PATT_LANE0	Pattern selection for lane 0: 000 = No training pattern (normal mode) 011 = PRBS7 transmitted

AUX Controller Interface

This section describes the registers that connect with the AUX controller interface.

DPTX_AUX_CONTROL

For transaction requests:

1. Wait for READY to be 1.
2. Write registers DPTX_AUX_COMMAND to DPTX_AUX_BYTE18 with the transaction command, address, length (0 – 15) fields, and data payload.
3. Write LENGTH with the transaction's total message length (3 for header + 1 for length byte + 0 to 16 for data bytes). The request transmission begins.

For transaction replies:

1. Issue a transaction request.
2. Wait for READY to be 1. Implement a timeout.
3. Read the transaction reply command from the DPTX_AUX_COMMAND register.
4. Read the transaction reply's total length from LENGTH.

5. Read the transaction reply data payload from registers DPTX_AUX_BYTE0 to DPTX_AUX_BYTE15 (read LENGTH - 1 bytes).

Table 9–24 shows the register for the DPTX_AUX_CONTROL bits.

Address: 0x0100

Direction: RW

Reset: 0x00000000

Table 9–24. DPTX_AUX_CONTROL Bits

Bit	Bit Name	Function
31	READY	AUX Controller status (RO): 0 = Busy sending a request or waiting for a reply 1 = Ready to send a request or a reply has been completely received.
30:5		Unused
4:0	LENGTH	For the next transaction request, total length of message to be transmitted (3 – 20), for the last received transaction reply, total length of message received (1 – 17).

DPTX_AUX_CMD

Table 9–25 shows the register for the DPTX_AUX_COMMAND bits.

Address: 0x0101

Direction: RW

Reset: 0x00000000

Table 9–25. DPTX_AUX_CMD Bits

Bit	Bit Name	Function
31:8		Unused
7:0	COMMAND	AUX transaction command for the next request or received in the last reply (refer to the DisplayPort specification for details)

DPTX_AUX_BYTE0

Table 9–26 shows the register for the DPTX_AUX_BYTE0 bits.

Address: 0x0102

Direction: RW

Reset: 0x00000000

Table 9–26. DPTX_AUX_BYTE0 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction address [15:8] for the next request, or data(0) received in the last reply

DPTX_AUX_BYTE1

Table 9–27 shows the register for the DPTX_AUX_BYTE1 bits.

Address: 0x0103

Direction: RW

Reset: 0x00000000

Table 9–27. DPTX_AUX_BYTE1 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction address [7:1] for the next request, or data(1) received in the last reply

DPTX_AUX_BYTE2

Table 9–28 shows the register for the DPTX_AUX_BYTE2 bits.

Address: 0x0104

Direction: RW

Reset: 0x00000000

Table 9–28. DPTX_AUX_BYTE2 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction length[3:0] for the next request, or data(2) received in the last reply (refer to the DisplayPort specification for details)

DPTX_AUX_BYTE3

Table 9–29 shows the register for the DPTX_AUX_BYTE3 bits.

Address: 0x0105

Direction: RW

Reset: 0x00000000

Table 9–29. DPTX_AUX_BYTE3 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(0) for the next request, or data(3) received in the last reply

DPTX_AUX_BYTE4

Table 9–30 shows the register for the DPTX_AUX_BYTE4 bits.

Address: 0x0106

Direction: RW

Reset: 0x00000000

Table 9–30. DPTX_AUX_BYTE4 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(1) for the next request, or data(4) received in the last reply

DPTX_AUX_BYTE5

Table 9–31 shows the register for the DPTX_AUX_BYTE5 bits.

Address: 0x0107

Direction: RW

Reset: 0x00000000

Table 9–31. DPTX_AUX_BYTE5 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(2) for the next request, or data(5) received in the last reply

DPTX_AUX_BYTE6

Table 9–32 shows the register for the DPTX_AUX_BYTE6 bits.

Address: 0x0108

Direction: RW

Reset: 0x00000000

Table 9–32. DPTX_AUX_BYTE6 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(3) for the next request, or data(6) received in the last reply

DPTX_AUX_BYTE7

Table 9–33 shows the register for the DPTX_AUX_BYTE7 bits.

Address: 0x0109

Direction: RW

Reset: 0x00000000

Table 9–33. DPTX_AUX_BYTE7 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(4) for the next request, or data(7) received in the last reply

DPTX_AUX_BYTE8

Table 9–34 shows the register for the DPTX_AUX_BYTE8 bits.

Address: 0x010a

Direction: RW

Reset: 0x00000000

Table 9–34. DPTX_AUX_BYTE8 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(5) for the next request, or data(8) received in the last reply

DPTX_AUX_BYTE9

Table 9–35 shows the register for the DPTX_AUX_BYTE9 bits.

Address: 0x010b

Direction: RW

Reset: 0x00000000

Table 9–35. DPTX_AUX_BYTE9 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(6) for the next request, or data(9) received in the last reply

DPTX_AUX_BYTE10

Table 9–36 shows the register for the DPTX_AUX_BYTE10 bits.

Address: 0x010c

Direction: RW

Reset: 0x00000000

Table 9–36. DPTX_AUX_BYTE10 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(7) for the next request, or data(10) received in the last reply

DPTX_AUX_BYTE11

Table 9–37 shows the register for the DPTX_AUX_BYTE11 bits.

Address: 0x010d

Direction: RW

Reset: 0x00000000

Table 9–37. DPTX_AUX_BYTE11 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(8) for the next request, or data(11) received in the last reply

DPTX_AUX_BYTE12

Table 9–38 shows the register for the DPTX_AUX_BYTE12 bits.

Address: 0x010e

Direction: RW

Reset: 0x00000000

Table 9–38. DPTX_AUX_BYTE12 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(9) for the next request, or data(12) received in the last reply

DPTX_AUX_BYTE13

Table 9–39 shows the register for the DPTX_AUX_BYTE13 bits.

Address: 0x010f

Direction: RW

Reset: 0x00000000

Table 9–39. DPTX_AUX_BYTE13 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(10) for the next request, or data(13) received in the last reply

DPTX_AUX_BYTE14

Table 9–40 shows the register for the DPTX_AUX_BYTE14 bits.

Address: 0x0110

Direction: RW

Reset: 0x00000000

Table 9–40. DPTX_AUX_BYTE14 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(11) for the next request, or data(14) received in the last reply

DPTX_AUX_BYTE15

Table 9–41 shows the register for the DPTX_AUX_BYTE15 bits.

Address: 0x0111

Direction: RW

Reset: 0x00000000

Table 9–41. DPTX_AUX_BYTE15 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(12) for the next request, or data(15) received in the last reply

DPTX_AUX_BYTE16

Table 9–42 shows the register for the DPTX_AUX_BYTE16 bits.

Address: 0x0112

Direction: RW

Reset: 0x00000000

Table 9–42. DPTX_AUX_BYTE16 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(13) for the next request

DPTX_AUX_BYTE17

Table 9–43 shows the register for the DPTX_AUX_BYTE17 bits.

Address: 0x0113

Direction: RW

Reset: 0x00000000

Table 9–43. DPTX_AUX_BYTE17 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(14) for the next request

DPTX_AUX_BYTE18

Table 9–44 shows the register for the DPTX_AUX_BYTE18 bits.

Address: 0x0114

Direction: RW

Reset: 0x00000000

Table 9–44. DPTX_AUX_BYTE18 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(15) for the next request

DPTX_AUX_RESET

Table 9–45 shows the register for the DPTX_AUX_RESET bits.

Address: 0x0117

Direction: WO

Reset: 0x00000000

Table 9–45. DPTX_AUX_RESET Bits

Bit	Bit Name	Function
31:1		Unused
0	CLEAR	Asserting CLEAR resets the AUX Controller state machine: 0 = No action 1 = AUX Controller reset

10. DisplayPort Sink Register Map and DCPD Locations

DisplayPort sink instantiations greatly benefit from and may optionally use an embedded controller (Nios II processor or another controller). This section describes the register map. Table 10–1 describes the notation used in this section.

Table 10–1. Notation

Shorthand	Definition
RW	Read/write
RO	Read only
WO	Write only
CRO	Clear on read or write, read only
CWO	Clear on read or write, write only

General Registers

The following section describes the general registers.

DPRX_RX_CONTROL

Table 10–2 describes the RX control register, `DPRX_RX_CONTROL`.

Address: 0x0000

Direction: RW

Reset: 0x00000000

Table 10–2. DPRX_RX_CONTROL Bits (Part 1 of 2)

Bit	Bit Name	Function
31:13		Unused
12	HPD_IRQ	When written at 1, generates a 0.75 ms HPD IRQ (low pulse). This bit is WO. For HPD_IRQ to function, HPD_EN must be equal to 1
11	HPD_EN	HPD logic level: 0 = De-asserted (low) 1 = Asserted (high)
10	GXB_RESET	0 = Sink GXB enabled 1 = Sink GXB reset
9:8	TP	Current training pattern: 00 = Normal video 01 = Training pattern 1 10 = Training pattern 2

Table 10–2. DPRX_RX_CONTROL Bits (Part 2 of 2)

Bit	Bit Name	Function
7	SCRAMBLER_DISABLE	0 = Scrambler enabled 1 = Scrambler disabled
6:5	TX_LINK_RATE (1)	Main link rate: 0 = 1.62 Gbps 1 = 2.7 Gbps 2 = 5.4 Gbps
4:0	LANE_COUNT (1)	Lane count: 00001 = 1 00010 = 2 00100 = 4

Note:

(1) The TX_LINKL_RATE and LANE_COUNT bits are also available in read only (RO) mode when not using a controller.

DPRX_RX_STATUS

Table 10–3 describes the RX status register, DPRX_RX_STATUS.

Address: 0x0001

Direction: CRO

Reset: 0x00000000

Table 10–3. DPRX_RX_STATUS Bits (Part 1 of 2)

Bit	Bit Name	Function
31:17		
16	SYNC_LOSS	This flag can be reset by writing it to 1: 0 = Symbol lock on all lanes in use 1 = Symbol lock lost on one or more of the used lanes
15:8		Unused
7	SYM_LOCK3	0 = Symbol unlocked (lane 3) 1 = Symbol locked (lane 3)
6	SYM_LOCK2	0 = Symbol unlocked (lane 2) 1 = Symbol locked (lane 2)
5	SYM_LOCK1	0 = Symbol unlocked (lane 1) 1 = Symbol locked (lane 1)
4	SYM_LOCK0	0 = Symbol unlocked (lane 0) 1 = Symbol locked (lane 0)
3	CR_LOCK3	0 = Clock unlocked (lane 3) 1 = Clock locked (lane 3)
2	CR_LOCK2	0 = Clock unlocked (lane 2) 1 = Clock locked (lane 2)

Table 10–3. DPRX_RX_STATUS Bits (Part 2 of 2)

Bit	Bit Name	Function
1	CR_LOCK1	0 = Clock unlocked (lane 1) 1 = Clock locked (lane 1)
0	CR_LOCK0	0 = Clock unlocked (lane 0) 1 = Clock locked (lane 0) This register is also available in read only (RO) mode when not using a controller.

DPRX_BER_CONTROL

Table 10–4 describes the bit error count control register, DPRX_BER_CONTROL.

Address: 0x0010

Direction: CRW

Reset: 0x00000000

Table 10–4. DPRX_BER_CONTROL Bits (Part 1 of 2)

Bit	Bit Name	Function
31:20		Unused
19	RST3	Writing this bit at 1 resets lane 3 bit-error counter in register DPRX_BER_CNT1. Always reads as 0.
18	RST2	Writing this bit at 1 resets the lane 2 bit-error counter in register DPRX_BER_CNT1. Always reads as 0.
17	RST1	Writing this bit at 1 resets lane 1 bit-error counter in register DPRX_BER_CNT0. Always reads as 0.
16	RST0	Writing this bit at 1 resets lane 0 bit-error counter in register DPRX_BER_CNT0. Always reads as 0.
15:14		Unused
13:11	PATT3	Pattern selection for lane 3: 000 = No training pattern (normal mode) 011 = PRBS7
10:8	PATT2	Pattern selection for lane 2: 000 = No training pattern (normal mode) 011 = PRBS7
7:5	PATT1	Pattern selection for lane 1: 000 = No training pattern (normal mode) 011 = PRBS7

Table 10–4. DPRX_BER_CONTROL Bits (Part 2 of 2)

Bit	Bit Name	Function
4:2	PATT0	Pattern selection for lane 0: 000 = No training pattern (normal mode) 011 = PRBS7
1:0	CNTSEL	Count selection: 00 = Disparity and illegal comma codes 01 = Disparity 10 = Illegal comma codes 11 = Reserved

DPRX_BER_CNT0

Table 10–5 describes the bit-error counters for lane 0 and lane 1, DPRX_BER_CNT0.

Address: 0x0011

Direction: RO

Reset: 0x00000000

Table 10–5. DPRX_BER_CNT0 Bits

Bit	Bit Name	Function
31		Unused
30:16	CNT1	Symbol error counter for lane 1
15		Unused
14:0	CNT0	Symbol error counter for lane 0

DPRX_BER_CNT1

Table 10–6 describes the bit-error counter register for lane 2 and lane 3, DPRX_BER_CNT1.

Address: 0x0012

Direction: RO

Reset: 0x00000000

Table 10–6. DPRX_BER_CNT1 Bits

Bit	Bit Name	Function
31		Unused
30:16	CNT3	Symbol error counter for lane 3
15		Unused
14:0	CNT2	Symbol error counter for lane 2

MSA Registers

These registers are allocated at addresses 0x0020 through 0x002f.

DPRX0_MSA_MVID

Table 10–7 describes the MSA MVID register, DPRX0_MSA_MVID.

Address: 0x0020

Direction: RO

Reset: 0x00000000

Table 10–7. DPRX0_MSA_MVID Bits

Bit	Bit Name	Function
31:24		Unused
23:0	MVID	Main stream attribute MVID

DPRX0_MSA_NVID

Table 10–8 describes the MSA NVID register, DPRX0_MSA_NVID.

Address: 0x0021

Direction: RO

Reset: 0x00000000

Table 10–8. DPRX0_MSA_NVID Bits

Bit	Bit Name	Function
31:24		Unused
23:0	NVID	Main stream attribute NVID

DPRX0_MSA_HTOTAL

Table 10–9 describes the MSA HTOTAL register, DPRX0_MSA_HTOTAL.

Address: 0x0022

Direction: RO

Reset: 0x00000000

Table 10–9. DPRX0_MSA_HTOTAL Bits

Bit	Bit Name	Function
31:16		Unused
15:0	HTOTAL	Main stream attribute HTOTAL

DPRX0_MSA_VTOTAL

Table 10–10 describes the MSA VTOTAL register, DPRX0_MSA_VTOTAL.

Address: 0x0023

Direction: RO

Reset: 0x00000000

Table 10–10. DPRX0_MSA_VTOTAL Bits

Bit	Bit Name	Function
31:16		Unused
15:0	MVID	Main stream attribute VTOTAL

DPRX0_MSA_HSP

Table 10–11 describes the MSA horizontal synchronization polarity register, DPRX0_MSA_HSP.

Address: 0x0024

Direction: RO

Reset: 0x00000000

Table 10–11. DPRX0_MSA_HSP Bits

Bit	Bit Name	Function
31:1		Unused
0	HSP	Main stream attribute horizontal synchronization polarity 0 = Positive 1 = Negative

DPRX0_MSA_HSW

Table 10–12 describes the MSA horizontal synchronization width register, DPRX0_MSA_HSW.

Address: 0x0025

Direction: RO

Reset: 0x00000000

Table 10–12. DPRX0_MSA_HSW Bits

Bit	Bit Name	Function
31:15		Unused
14:0	HSW	Main stream attribute horizontal synchronization width

DPRX0_MSA_HSTART

Table 10–13 describes the MSA HSTART register, DPRX0_MSA_HSTART.

Address: 0x0026

Direction: RO

Reset: 0x00000000

Table 10–13. DPRX0_MSA_HSTART Bits

Bit	Bit Name	Function
31:16		Unused
15:0	HSTART	Main stream attribute HSTART

DPRX0_MSA_VSTART

Table 10–14 describes the MSA VSTART register, DPRX0_MSA_VSTART.

Address: 0x0027

Direction: RO

Reset: 0x00000000

Table 10–14. DPRX0_MSA_VSTART Bits

Bit	Bit Name	Function
31:16		Unused
15:0	VSTART	Main stream attribute VSTART

DPRX0_MSA_VSP

Table 10–15 describes the MSA vertical synchronization polarity register, DPRX0_MSA_VSP.

Address: 0x0028

Direction: RO

Reset: 0x00000000

Table 10–15. DPRX0_MSA_VSP Bits

Bit	Bit Name	Function
31:1		Unused
0	VSP	Main stream attribute vertical synchronization polarity 0 = Positive 1 = Negative

DPRX0_MSA_VSW

Table 10–16 describes the MSA vertical synchronization width register, DPRX0_MSA_VSW.

Address: 0x0029

Direction: RO

Reset: 0x00000000

Table 10–16. DPRX0_MSA_VSW Bits

Bit	Bit Name	Function
31:15		Unused
14:0	VSW	Main stream attribute vertical synchronization width

DPRX0_MSA_HWIDTH

Table 10–17 describes the TX control register, DPRX0_MSA_HWIDTH.

Address: 0x002a

Direction: RO

Reset: 0x00000000

Table 10–17. DPRX0_MSA_HWIDTH Bits

Bit	Bit Name	Function
31:16		Unused
15:0	HWIDTH	Main stream attribute HWIDTH

DPRX0_MSA_VHEIGHT

Table 10–18 describes the MSA VHEIGHT register, DPRX0_MSA_VHEIGHT.

Address: 0x002b

Direction: RO

Reset: 0x00000000

Table 10–18. DPRX0_MSA_WHEIGHT Bits

Bit	Bit Name	Function
31:16		Unused
15:0	VHEIGHT	Main stream attribute VHEIGHT

DPRX0_MSA_MISC0

Table 10–19 describes the MSA MISC0 register, DPRX0_MSA_MISC0.

Address: 0x002c

Direction: RO

Reset: 0x00000000

Table 10–19. DPRX0_MSA_MISC0 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	MISC0	Main stream attribute MISC0

DPRX0_MSA_MISC1

Table 10–20 describes the MSA MISC1 register, DPRX0_MSA_MISC1.

Address: 0x002d

Direction: RO

Reset: 0x00000000

Table 10–20. DPRX0_MSA_MISC1 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	MISC1	Main stream attribute MISC1

DPRX0_VBID

Table 10–21 describes the VB-ID register, DPRX0_VBID.

Address: 0x002e

Direction: RO

Reset: 0x00000000

Table 10–21. DPRX0_VBID Bits

Bit	Bit Name	Function
31:8		Unused
7	MSA_LOCK	0 = MSA unlocked 1 = MSA locked (on all lanes)
6	VBID_LOCK	0 = VB-ID unlocked 1 = VB-ID locked (on all lanes)
5:0	VBID	VB-ID flags (refer to the DisplayPort specification)

Audio Registers

The following sections describe the audio registers.

DPRX0_AUD_MAUD

Table 10–22 describes the received audio Maud register, DPRX0_AUD_MAUD.

Address: 0x0030

Direction: RO

Reset: 0x00000000

Table 10–22. DPRX0_AUD_MAUD Bits

Bit	Bit Name	Function
31:24		
23:0	MAUD	Received audio Maud

DPRX0_AUD_NAUD

Table 10–23 describes the received audio Naud register, DPRX0_AUD_NAUD.

Address: 0x0031

Direction: RO

Reset: 0x00000000

Table 10–23. DPRX0_AUD_NAUD Bits

Bit	Bit Name	Function
31:24		
23:0	NAUD	Received audio Naud

DPRX0_AUD_AIF0

Table 10–24 describes the received audio InfoFrame register, DPRX0_AUD_AIF0.

Address: 0x0032

Direction: RO

Reset: 0x00000000

Table 10–24. DPRX0_AUD_AIF0 Bits

Bit	Bit Name	Function
31:8		
7:0	AIF	Received audio InfoFrame byte 0 (refer to CEA-861-E specification)

DPRX0_AUD_AIF1

Table 10–25 describes the received audio InfoFrame register, DPRX0_AUD_AIF1.

Address: 0x0033

Direction: RO

Reset: 0x00000000

Table 10–25. DPRX0_AUD_AIF1 Bits

Bit	Bit Name	Function
31:8		
7:0	AIF	Received audio InfoFrame byte 1 (refer to CEA-861-E specification)

DPRX0_AUD_AIF2

Table 10–26 describes the received audio InfoFrame register, DPRX0_AUD_AIF2.

Address: 0x0034

Direction: RO

Reset: 0x00000000

Table 10–26. DPRX0_AUD_AIF2 Bits

Bit	Bit Name	Function
31:8		
7:0	AIF	Received audio InfoFrame byte 2 (refer to CEA-861-E specification)

DPRX0_AUD_AIF3

Table 10–27 describes the received audio InfoFrame register, DPRX0_AUD_AIF3.

Address: 0x0035

Direction: RO

Reset: 0x00000000

Table 10–27. DPRX0_AUD_AIF3 Bits

Bit	Bit Name	Function
31:8		
7:0	AIF	Received audio InfoFrame byte 3 (refer to CEA-861-E specification)

DPRX0_AUD_AIF4

Table 10–28 describes the received audio InfoFrame register, DPRX0_AUD_AIF4.

Address: 0x0036

Direction: R0

Reset: 0x00000000

Table 10–28. DPRX0_AUD_AIF4 Bits

Bit	Bit Name	Function
31:8		
7:0	AIF	Received audio InfoFrame byte 4 (refer to CEA-861-E specification)

AUX Controller Interface

The following sections describe the registers for the AUX Controller interface.

DPRX_AUX_CONTROL

For transaction requests:

1. Wait for READY to be 1, or enable the interrupt in register DPRX_AUX_IRQ_EN and wait for the interrupt request.
2. Read the transaction request command from DPRX_AUX_COMMAND.
3. Read the transaction request total length from LENGTH.
4. Read the transaction request data payload from registers DPRX_AUX_BYTE0 to DPRX_AUX_BYTE15 (read LENGTH - 1 bytes).

For transaction replies:

1. Wait for READY to be 1. Implement a timeout.
2. Write registers DPRX_AUX_COMMAND to DPRX_AUX_BYTE18 with transaction command and data payload.
3. Write LENGTH with the transaction total message length (1 to 17, 1 for the command plus 1 to 16 for the data payload). This sequence starts the reply transmission.

Table 10–29 describes the AUX Controller control register, DPRX_AUX_CONTROL.

Address: 0x0100

Direction: RW

Reset: 0x00000000

Table 10–29. DPRX_AUX_CONTROL Bits

Bit	Bit Name	Function
31	READY	AUX Controller status (RO): 0 = Busy sending a reply or waiting for a request 1 = Ready to send a reply or a request has been completely received
30:5		Unused
4:0	LENGTH	For the next transaction reply, total length of message to be transmitted (1 – 17), for the last received transaction request, total length of message received (1 – 17).

DPRX_AUX_CMD

Table 10–30 describes the AUX transaction command register, DPRX_AUX_COMMAND.

Address: 0x0101

Direction: RW

Reset: 0x00000000

Table 10–30. DPRX_AUX_COMMAND Bits

Bit	Bit Name	Function
31:8		Unused
7:0	COMMAND	AUX transaction command for the next reply or received in the last request (refer to the DisplayPort specification)

DPRX_AUX_BYTE0

Table 10–31 describes the AUX transaction byte 0 register, DPRX_AUX_BYTE0.

Address: 0x0102

Direction: RW

Reset: 0x00000000

Table 10–31. DPRX_AUX_BYTE0 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction address[15:8] received in the last request, or data(0) for the next reply

DPRX_AUX_BYTE1

Table 10–32 describes the AUX transaction byte 1 register, DPRX_AUX_BYTE1.

Address: 0x0103

Direction: RW

Reset: 0x00000000

Table 10–32. DPRX_AUX_BYTE1 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction address[7:1] received in the last request, or data(1) for the next reply

DPRX_AUX_BYTE2

Table 10–33 describes the AUX transaction byte 2 register, DPRX_AUX_BYTE2.

Address: 0x0104

Direction: RW

Reset: 0x00000000

Table 10–33. DPRX_AUX_BYTE2 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction length[3:0] received in the last request, or data(2) for the next reply (refer to DisplayPort specification)

DPRX_AUX_BYTE3

Table 10–34 describes the AUX transaction byte 3 register, DPRX_AUX_BYTE3.

Address: 0x0105

Direction: RW

Reset: 0x00000000

Table 10–34. DPRX_AUX_BYTE3 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(0) received in the last request, or data(3) for the next reply

DPRX_AUX_BYTE4

Table 10–35 describes the AUX transaction byte 4 register, DPRX_AUX_BYTE4.

Address: 0x0106

Direction: RW

Reset: 0x00000000

Table 10–35. DPRX_AUX_BYTE4 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(1) received in the last request, or data(4) for the next reply

DPRX_AUX_BYTE5

Table 10–36 describes the AUX transaction byte 5 register, DPRX_AUX_BYTE5.

Address: 0x0107

Direction: RW

Reset: 0x00000000

Table 10–36. DPRX_AUX_BYTE5 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(2) received in the last request, or data(5) for the next reply

DPRX_AUX_BYTE6

Table 10–37 describes the AUX transaction byte 6 register, DPRX_AUX_BYTE6.

Address: 0x0108

Direction: RW

Reset: 0x00000000

Table 10–37. DPRX_AUX_BYTE6 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(3) received in the last request, or data(6) for the next reply

DPRX_AUX_BYTE7

Table 10–38 describes the AUX transaction byte 7 register, DPRX_AUX_BYTE7.

Address: 0x0109

Direction: RW

Reset: 0x00000000

Table 10–38. DPRX_AUX_BYTE7 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(4) received in the last request, or data(7) for the next reply

DPRX_AUX_BYTE8

Table 10–39 describes the AUX transaction byte 8 register, DPRX_AUX_BYTE8.

Address: 0x010a

Direction: RW

Reset: 0x00000000

Table 10–39. DPRX_AUX_BYTE8 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(5) received in the last request, or data(8) for the next reply

DPRX_AUX_BYTE9

Table 10–40 describes the AUX transaction byte 9 register, DPRX_AUX_BYTE9.

Address: 0x010b

Direction: RW

Reset: 0x00000000

Table 10–40. DPRX_AUX_BYTE9 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(6) received in the last request, or data(9) for the next reply

DPRX_AUX_BYTE10

Table 10–41 describes the AUX transaction byte 10 register, DPRX_AUX_BYTE10.

Address: 0x010c

Direction: RW

Reset: 0x00000000

Table 10–41. DPRX_AUX_BYTE10 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(7) received in the last request, or data(10) for the next reply

DPRX_AUX_BYTE11

Table 10–42 describes the AUX transaction byte 11 register, DPRX_AUX_BYTE11.

Address: 0x010d

Direction: RW

Reset: 0x00000000

Table 10–42. DPRX_AUX_BYTE11Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(8) received in the last request, or data(11) for the next reply

DPRX_AUX_BYTE12

Table 10–43 describes the AUX transaction byte 12 register, DPRX_AUX_BYTE12.

Address: 0x010e

Direction: RW

Reset: 0x00000000

Table 10–43. DPRX_AUX_BYTE12 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(9) received in the last request, or data(12) for the next reply

DPRX_AUX_BYTE13

Table 10–44 describes the AUX transaction byte 13 register, DPRX_AUX_BYTE13.

Address: 0x010f

Direction: RW

Reset: 0x00000000

Table 10–44. DPRX_AUX_BYTE13 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(10) received in the last request, or data(13) for the next reply

DPRX_AUX_BYTE14

Table 10–45 describes the AUX transaction byte 14 register, DPRX_AUX_BYTE14.

Address: 0x0110

Direction: RW

Reset: 0x00000000

Table 10–45. DPRX_AUX_BYTE14 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(11) received in the last request, or data(14) for the next reply

DPRX_AUX_BYTE15

Table 10–46 describes the AUX transaction byte 15 register, DPRX_AUX_BYTE15.

Address: 0x0111

Direction: RW

Reset: 0x00000000

Table 10–46. DPRX_AUX_BYTE15 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(12) received in the last request, or data(15) for the next reply

DPRX_AUX_BYTE16

Table 10–47 describes the AUX transaction byte 16 register, DPRX_AUX_BYTE16.

Address: 0x0112

Direction: RW

Reset: 0x00000000

Table 10–47. DPRX_AUX_BYTE16 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(13) received in the last request

DPRX_AUX_BYTE17

Table 10–48 describes the AUX transaction byte 17 register, DPRX_AUX_BYTE17.

Address: 0x0113

Direction: RW

Reset: 0x00000000

Table 10–48. DPRX_AUX_BYTE17 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(14) received in the last request

DPRX_AUX_BYTE18

Table 10–49 describes the AUX transaction byte 18 register, DPRX_AUX_BYTE18.

Address: 0x0114

Direction: RW

Reset: 0x00000000

Table 10–49. DPRX_AUX_BYTE18 Bits

Bit	Bit Name	Function
31:8		Unused
7:0	BYTE	Transaction data(15) received in the last request

DPRX_AUX_IRQ_EN

The IRQ is asserted when ENABLE = 1 and the flag READY = 1 in the DPRX_AUX_CONTROL register. IRQ is de-asserted by setting ENABLE to 0 or by posting a transaction reply by writing to DPRX_AUX_CONTROL.

Table 10–50 describes the RX register, DPRX_AUX_IRQ_EN.

Address: 0x0116

Direction: RW

Reset: 0x00000000

Table 10–50. DPRX_AUX_IRQ_EN Bits

Bit	Bit Name	Function
31:1		Unused
0	ENABLE	Enables an IRQ issued to the Nios II processor when a transaction request is received from the source: 0 = disable 1 = enable

DPRX_AUX_RESET

Table 10–51 describes the AUX reset register, DPRX_AUX_RESET.

Address: 0x0117

Direction: WO

Reset: 0x00000000

Table 10–51. DPRX_AUX_RESET Bits

Bit	Bit Name	Function
31:1		Unused
0	CLEAR	Asserting CLEAR resets the AUX controller state machine: 0 = No action 1 = AUX Controller reset

Sink-Supported DPCD Locations

Table 10–52 describes the DPCD locations (or location groups) that are supported in DisplayPort sink instantiations.

Table 10–52. DPCD Locations (Part 1 of 4)

Location Name	Address	Without Controller	With Controller
DPCD_REV	0x0000	Yes	Yes
MAX_LINK_RATE	0x0001	Yes	Yes
MAX_LANE_COUNT	0x0002	Yes	Yes
MAX_DOWNSPREAD	0x0003	Yes	Yes
NORP	0x0004	Yes	Yes
DOWNSTREAMPORT_PRESENT	0x0005	Yes	Yes
MAIN_LINK_CHANNEL_CODING	0x0006	Yes	Yes
DOWN_STREAM_PORT_COUNT	0x0007	Yes	Yes
RECEIVE_PORT0_CAP_0	0x0008	Yes	Yes
RECEIVE_PORT0_CAP_1	0x0009	Yes	Yes
RECEIVE_PORT1_CAP_0	0x000A	Yes	Yes
RECEIVE_PORT1_CAP_1	0x000B	Yes	Yes
I2C_SPEED_CONTROL	0x000C		Yes
EDP_CONFIGURATION_CAP	0x000D		Yes
TRAINING_AUX_RD_INTERVAL	0x000E		Yes
ADAPTER_CAP	0x000F		Yes
FAUX_CAP	0x0020		Yes
MST_CAP	0x0021		Yes
NUMBER_OF_AUDIO_ENDPOINTS	0x0022		Yes
GUID	0x0030		Yes
DWN_STRM_PORTX_CAP	0x0080	Yes	Yes
LINK_BW_SET	0x0100	Yes	Yes
LANE_COUNT_SET	0x0101	Yes	Yes
TRAINING_PATTERN_SET	0x0102	Yes	Yes
TRAINING_LANE0_SET	0x0103	Yes	Yes
TRAINING_LANE1_SET	0x0104	Yes	Yes
TRAINING_LANE2_SET	0x0105	Yes	Yes
TRAINING_LANE3_SET	0x0106	Yes	Yes
DOWNSPREAD_CTRL	0x0107	Yes	Yes
MAIN_LINK_CHANNEL_CODING_SET	0x0108	Yes	Yes
I2C_SPEED_CONTROL	0x0109		Yes
EDP_CONFIGURATION_SET	0x010A		Yes
LINK_QUAL_LANE0_SET	0x010B		Yes
LINK_QUAL_LANE1_SET	0x010C		Yes
LINK_QUAL_LANE2_SET	0x010D		Yes

Table 10–52. DPCD Locations (Part 2 of 4)

Location Name	Address	Without Controller	With Controller
LINK_QUAL_LANE3_SET	0x010E		Yes
TRAINING_LANE0_1_SET2	0x010F		Yes
TRAINING_LANE2_3_SET2	0x0110		Yes
MSTM_CTRL	0x0111		Yes
AUDIO_DELAY[7:0]	0x0112		Yes
AUDIO_DELAY[15:8]	0x0113		Yes
AUDIO_DELAY[23:6]	0x0114		Yes
ADAPTER_CTRL	0x01A0		Yes
BRANCH_DEVICE_CTRL	0x01A1		Yes
PAYLOAD_ALLOCATE_SET	0x01C0		Yes
PAYLOAD_ALLOCATE_START_TIME_SLOT	0x01C1		Yes
PAYLOAD_ALLOCATE_TIME_SLOT_COUNT	0x01C2		Yes
SINK_COUNT	0x0200	Yes	Yes
DEVICE_SERVICE_IRQ_VECTOR	0x0201	Yes	Yes
LANE0_1_STATUS	0x0202	Yes	Yes
LANE2_3_STATUS	0x0203	Yes	Yes
LANE_ALIGN_STATUS_UPDATED	0x0204	Yes	Yes
SINK_STATUS	0x0205	Yes	Yes
ADJUST_REQUEST_LANE0_1	0x0206	Yes	Yes
ADJUST_REQUEST_LANE2_3	0x0207	Yes	Yes
SYMBOL_ERROR_COUNT_LANE0	0x0210	Yes	Yes
SYMBOL_ERROR_COUNT_LANE1	0x0212	Yes	Yes
SYMBOL_ERROR_COUNT_LANE2	0x0214	Yes	Yes
SYMBOL_ERROR_COUNT_LANE3	0x0216	Yes	Yes
TEST_LANE_COUNT	0x0220	Yes	
TEST_PATTERN	0x0221	Yes	
TEST_H_TOTAL_LSB	0x0222	Yes	
TEST_H_TOTAL_MSB	0x0223	Yes	
TEST_V_TOTAL_LSB	0x0224	Yes	
TEST_V_TOTAL_MSB	0x0225	Yes	
TEST_H_START_LSB	0x0226	Yes	
TEST_H_START_MSB	0x0227	Yes	
TEST_V_START_LSB	0x0228	Yes	
TEST_V_START_MSB	0x0229	Yes	
TEST_HSYNC_LSB	0x022A	Yes	
TEST_HSYNC_MSB	0x022B	Yes	
TEST_VSYNC_LSB	0x022C	Yes	
TEST_VSYNC_MSB	0x022D	Yes	
TEST_H_WIDTH_LSB	0x022E	Yes	

Table 10–52. DPCD Locations (Part 3 of 4)

Location Name	Address	Without Controller	With Controller
TEST_H_WIDTH_MSB	0x022F	Yes	
TEST_V_HEIGHT_LSB	0x0230	Yes	
TEST_V_HEIGHT_MSB	0x0231	Yes	
TEST_MISC_LSB	0x0232	Yes	
TEST_MISC_MSB	0x0233	Yes	
TEST_REFRESH_RATE_NUMERATOR	0x0234	Yes	
TEST_CRC_R_Cr	0x0240	Yes	
TEST_CRC_G_Y	0x0242	Yes	
TEST_CRC_B_Cb	0x0244	Yes	
TEST_SINK_MISC	0x0246	Yes	
TEST_RESPONSE	0x0260	Yes	
TEST_EDID_CHECKSUM	0x0261	Yes	
TEST_SINK	0x0270	Yes	
PAYLOAD_TABLE_UPDATE_STATUS	0x02C0		Yes
VC_PAYLOAD_ID_SLOT_1 to _63	0x02C1		Yes
IEEE_OUI	0x0300		Yes
IEEE_OUI	0x0301		Yes
IEEE_OUI	0x0302		Yes
DEVICE_IDENTIFICATION_STRING	0x0303		Yes
HARDWARE_REVISION	0x0309		Yes
FWSW_MAJOR	0x030A		Yes
FWSW_MINOR	0x030B		Yes
RESERVED	0x030C		Yes
RESERVED	0x030D		Yes
RESERVED	0x030E		Yes
RESERVED	0x030F		Yes
IEEE_OUI	0x0400		Yes
IEEE_OUI	0x0401		Yes
IEEE_OUI	0x0402		Yes
DEVICE_IDENTIFICATION_STRING	0x0403		Yes
HARDWARE_REVISION	0x0409		Yes
FWSW_MAJOR	0x040A		Yes
FWSW_MINOR	0x040B		Yes
RESERVED (0x040C to 0x04FF)	0x040C		Yes
IEEE_OUI	0x0500	Yes	Yes
IEEE_OUI	0x0501	Yes	Yes
IEEE_OUI	0x0502	Yes	Yes
DEVICE_IDENTIFICATION_STRING	0x0503		Yes
HARDWARE_REVISION	0x0509		Yes

Table 10–52. DPCD Locations (Part 4 of 4)

Location Name	Address	Without Controller	With Controller
FWSW_MAJOR	0x050A		Yes
FWSW_MINOR	0x050B		Yes
RESERVED (0x050C to 0x05FF)	0x050C		Yes
SET_POWER_STATE	0x0600	Yes	Yes
DOWN_REQ (0x1000 to 0x102F)	0x1000		Yes
DOWN_REP (0x1400 to 0x142F)	0x1400		Yes
SINK_COUNT_ESI	0x2002		Yes
DEVICE_SERVICE_IRQ_VECTOR_ESI0	0x2003		Yes
DEVICE_SERVICE_IRQ_VECTOR_ESI1	0x2004		Yes
LINK_SERVICE_IRQ_VECTOR_ESI0	0x2005		Yes
LANE0_1_STATUS	0x200C		Yes
LANE2_3_STATUS_ESI	0x200D		Yes
LANE_ALIGN___STATUS_UPDATED_ESI	0x200E		Yes
SINK_STATUS_ESI	0x200F		Yes

This chapter provides additional information about the document and Altera.

Document Revision History

The following table lists the revision history for this document.

Date	Version	Changes
May 2013	13.0	■ Added information on audio support. ■ Added HBR2 support for Stratix V devices. ■ Added information on secondary data support.
February 2013	12.1 SP1 (Beta)	Second beta release: ■ Updated the filenames for the hardware demonstration and simulation example. ■ Added chapter describing the IP core's compilation example. ■ Miscellaneous updates.
December 2012	12.1 (Beta)	Initial beta release.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table lists the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	The question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	The multimedia icon directs you to a related multimedia presentation.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.
	The feedback icon allows you to submit feedback to Altera about the document. Methods for collecting feedback vary as appropriate for each document.