BEEcube Platform Studio
Version 4.5

User Manual

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction to BEEcube Platform Studio

BEEcube Platform Studio (BPS) is a system-level, hardware/software co-development environment which runs on top of the MathWorks™ Simulink® framework. BPS provides automatic generation of all platform specific hardware interfaces and corresponding software drivers, as well as a direct algorithm-to-implementation conversion without requiring user knowledge of the low level implementation details.

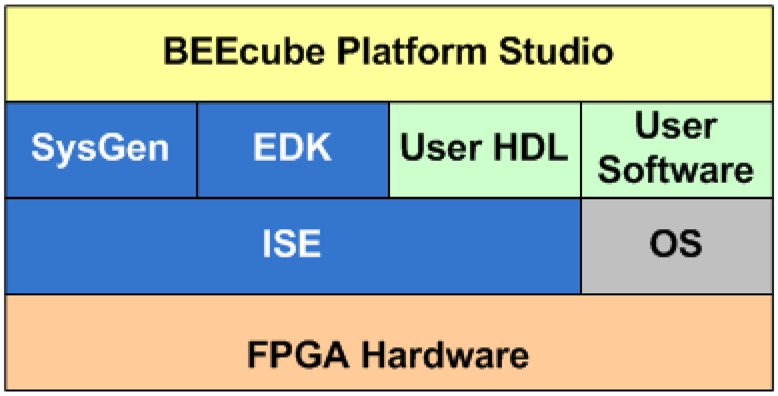


Figure 1: BEEcube Platform Studio

What is a BPS Platform?

Each BPS platform is a collection of hardware devices and associated software tailored for use on a specific physical system. The BPS platform has been designed to abstract hardware-specific details away from the end user and accelerate the design of the application itself.

The smallest unit of the BPS platform is a single-FPGA design.

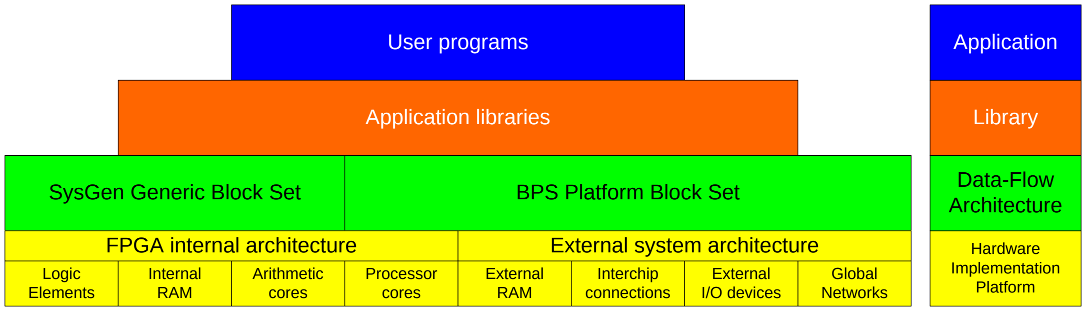


Figure 2: BPS Platform

A typical design in the BPS design environment starts with the core algorithm design in Simulink with Xilinx System Generator for DSP (XSG). From the end-user perspective, Simulink designs only exist in an idealized sandbox with the synchronous data flow execution model; all connections outside the core algorithm are virtually mapped through BPS interface block sets.

The BPS blocksets are created by FPGA experts to replace generic XSG “gateways” as interfaces between the core algorithm design in Simulink and the system-level devices.

A processor core, currently the MicroBlaze™ soft-core processor, is implicitly included in all BPS designs. The processor core can communicate with the user’s XSG design through a variety of shared memory components, such as software registers, Block RAMs, FIFOs, or DRAM. Users can specify the desired data structure by selecting the corresponding BPS blocks in Simulink. All external network, I/O, and memory devices are abstracted as Simulink data sources or sinks, with a simple FIFO communication protocol.

For all supported hardware platforms, the BPS framework provides *base packages* as complete Xilinx Embedded Development Kit (EDK) projects, as well as a corresponding Simulink BPS block with interface options for all external devices. Each base package includes all the essential system device IP cores, an initial hardware system configuration, and an extensible embedded software environment.

The back-end implementation files for the user-selected external devices are dynamically generated by the BPS tools on top of the base package. These are then automatically combined and connected with all necessary hardware and software components.

Base Package

The BPS base package is a preconfigured EDK project created in the Xilinx Platform Studio (XPS) format which contains the minimal infrastructure required to support all available library blocks on the hardware platform. BPS base packages are currently available for the BEE3, BEE4, miniBEE4, and BEE7 hardware platforms, as well as the Xilinx ML505, ML506, ML507, XUPV5-LX110T (ML509), and ML605 Evaluation Platform boards.

For example, the standard BPS base package for BEE3 includes:

- MicroBlaze soft processor core for system integration, management, and debugging
- Simple, extensible software monitoring and debugging shell (NectarOS shell)
- UART for console access to the embedded shell
- IIC controller for reading PROM contents (FPGA ID, MAC address, etc.)
- 100MHz crystal input and clock generator for all base infrastructure components

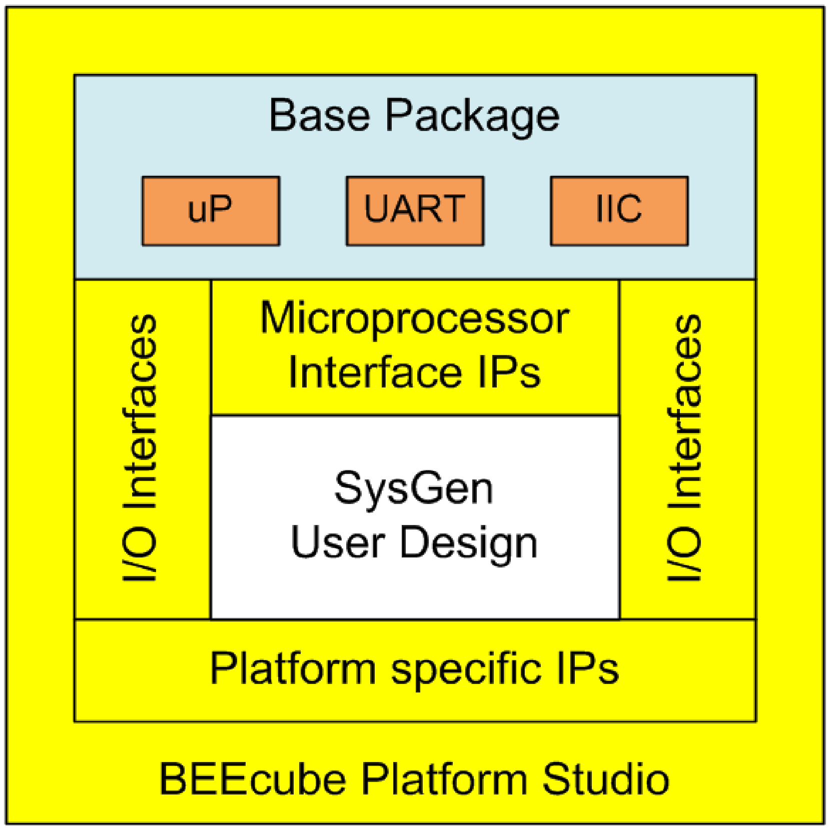


Figure 3: BPS Base Package

Tool Flow

BPS is built on top of existing Xilinx tool flows. Whereas Xilinx System Generator (XSG) provides an excellent blockset for mapping DSP algorithms, Xilinx Embedded Development Kit (EDK) provides reliable microprocessor and system-level integration, and Xilinx ISE and Vivado software provides the back-end for logic synthesis, place and route, and bitfile generation.

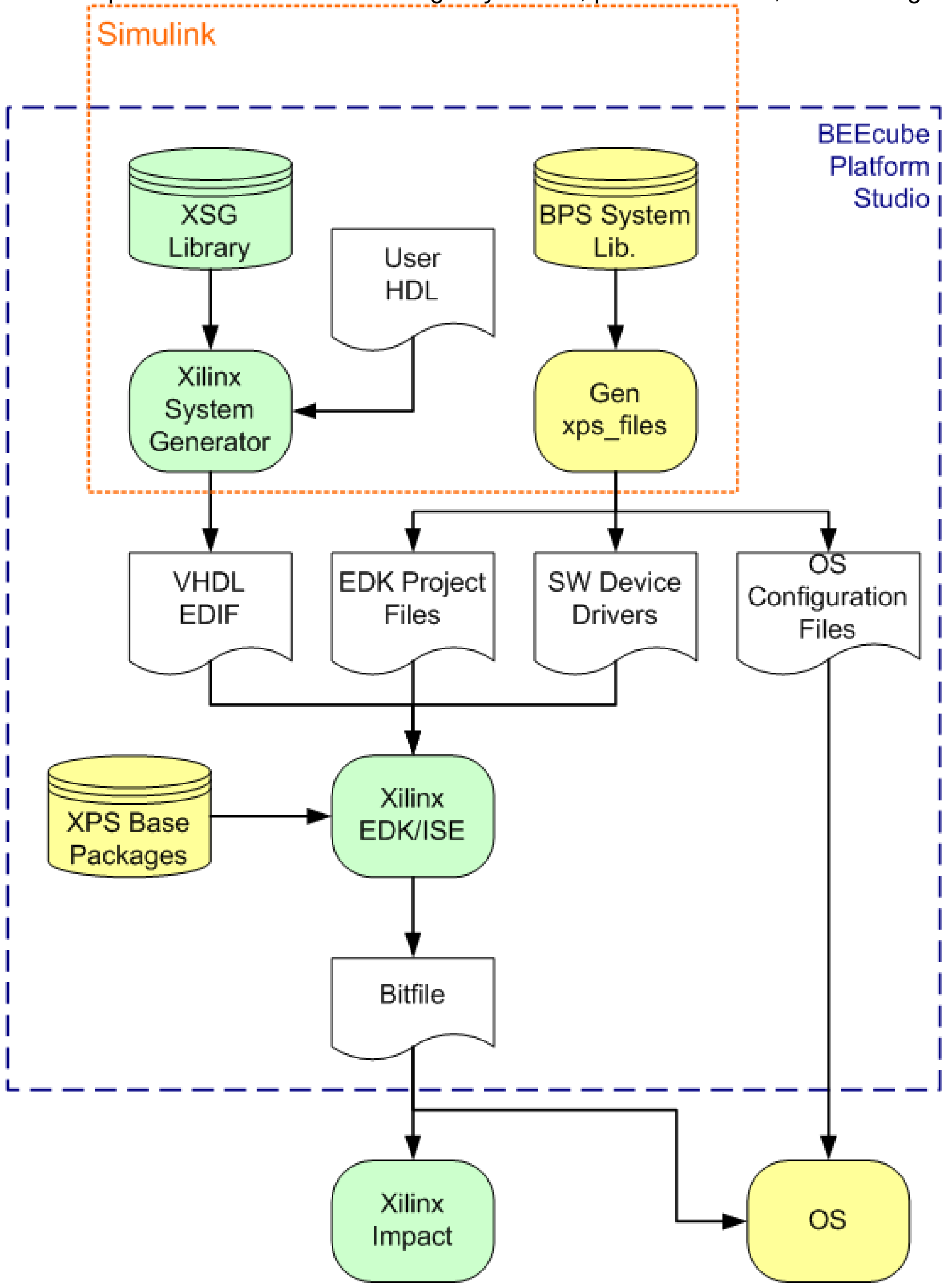


Figure 4: BPS Tool Flow

Overview of BPS Library Blocks

BPS provides a Simulink based blockset for each supported hardware platform. The BPS library blocks currently available for all supported platforms include:

- BPS platform independent blocks
  - Software Register (with and without strobe)
  - Shared FIFO
  - Shared BRAM
  - ChipScope Configuration
  - ChipScope Probe
- BEE3 platform specific blocks
  - BEE3 Platform Configuration
  - GPIO
  - Multi-port Memory Controller
  - MPMC Direct Interface
  - MPMC Video Frame Buffer Interface
  - Ethernet Interface
  - 10 Gb Ethernet Interface
  - PCI Express Endpoint
  - Aurora Streaming Serial Link
  - High-speed Inter-FPGA I/O
  - ADC Expansion Board Interface
  - DAC Expansion Board Interface
- BEE4 platform specific blocks
  - BEE4 Platform Configuration
  - GPIO
  - High-speed Inter-FPGA I/O
  - FMC101 Expansion Board ADC Interface
  - FMC101 Expansion Board DAC Interface
  - FMC104 Expansion Board ADC Interface
  - FMC105 Expansion Board DAC Interface
  - FMC111 Control
  - FMC111 RX Channel
  - FMC111 TX Channel
  - CPRI Interface
  - Ethernet MAC LocalLink
  - 10 Gb Ethernet Interface
  - Aurora Streaming Serial Link
  - XAUI Interface
  - DDR3 MIG
  - DDR3 FIFO
  - FMC DVI Input
  - FMC DVI Output
- miniBEE4 platform specific blocks
  - miniBEE4 Platform Configuration
  - GPIO
  - FMC101 Expansion Board ADC Interface
  - FMC101 Expansion Board DAC Interface
  - FMC104 Expansion Board ADC Interface
  - FMC105 Expansion Board DAC Interface

  - FMC111 Control
  - FMC111 RX Channel
  - FMC111 TX Channel
  - Ethernet MAC LocalLink
  - 10 Gb Ethernet Interface
  - RXAUI Interface
  - Aurora Streaming Serial Link
  - DDR3 MIG
  - DDR3 FIFO
  - FMC DVI Input
  - FMC DVI Output
- BEE7 platform specific blocks
  - BEE7 Platform Configuration
  - GPIO
  - SerDes I/O
  - FMC104 Expansion Board ADC Interface
  - FMC105 Expansion Board DAC Interface
  - FMC108 Expansion Board ADC Interface
  - FMC108 Expansion Board ADC DES Interface
  - FMC108 Expansion Board ADC DES 2-to-1 Interface
  - FMC109 Expansion Board DAC Interface
  - FMC109 Expansion Board DAC 2to1 Interface
  - Aurora 64b66b Streaming Serial Link
  - DDR3 Configuration
  - DDR3 FIFO
- Xilinx ML50x platform specific blocks
  - ML50x Platform Configuration
  - GPIO
  - Multi-port Memory Controller
  - MPMC Direct Interface
  - MPMC Video Frame Buffer Interface
  - SRAM Interface
  - Ethernet Interface
  - PCI Express Endpoint
  - Aurora Streaming Serial Link
  - VGA Analog Video Input
  - DVI Digital Video Output
- Xilinx ML60x platform specific blocks
  - ML60x Platform Configuration
  - GPIO
  - Multi-port Memory Controller
  - MPMC Direct Interface
  - MPMC Video Frame Buffer Interface
  - Ethernet Interface
  - FMC101 Expansion Board ADC Interface
  - FMC101 Expansion Board DAC Interface
  - HTG-FMC-SFP-PLUS XAUI Interface
  - FMC DVI Input
  - FMC DVI Output

A complete reference of the contents, behavior, and available customization for each of these blocks is contained in Chapter 5.

Chapter 2: Installing BPS

The BPS tool flow for hardware/software generation operates entirely within the Matlab environment and does not strictly require that it be installed locally on your workstation. However, an installer is provided for your convenience which will automatically unpack the BPS software distribution and configure your Matlab environment to include the BPS tool set. Alternatively, you may unpack the distribution into a directory of your choice and manually set up your Matlab environment to suit your usage model. The remainder of this chapter describes both methods for installing BPS in your design environment.

System Requirements

This version of BPS has been designed for and tested with the following third-party tools, listed by vendor:

- The Mathworks
  - Matlab/Simulink R2013a
- Xilinx
  - Vivado 2013.4
  - ISE 14.7
  - EDK 14.7
  - System Generator for DSP 14.7
- Mentor Graphics (optional)
  - ModelSim 6.6d

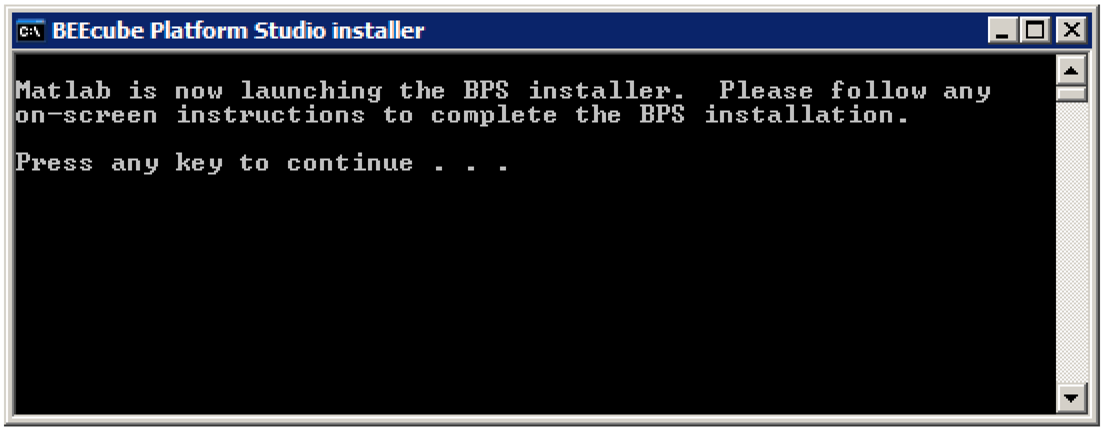
BPS is supported for use on 64-bit Windows and 64-bit Linux operating systems only. Please refer to the operating system requirements imposed by each of the third-party tools above for specifically supported versions.

Using the BPS Installer (Windows)

BPS comes with an installation utility for Windows which will automatically check your environment for all required third-party tools and versions, prompt you for a destination directory, unpack the BPS distribution, and attempt to add the necessary BPS directories to the default Matlab path. Use of the installer is recommended even in shared network environments when the Matlab installation may not be modifiable, as it will still check for a number of system requirements and perform all other installation tasks other than changing the default Matlab path.

Because the BPS installer will verify that supported versions of all required third-party tools are present on the system, please be sure that all required tools in the System Requirements section are installed and properly configured in your current desktop session before continuing. In addition, it will be necessary to have write permission to the desired install destination directory, so please be sure that the currently logged-in account has sufficient permission to complete the installation.

To run the BPS installer, launch the script setup.cmd which came with the full BPS distribution. If the script is able to run and successfully launch Matlab, the following screen will appear.

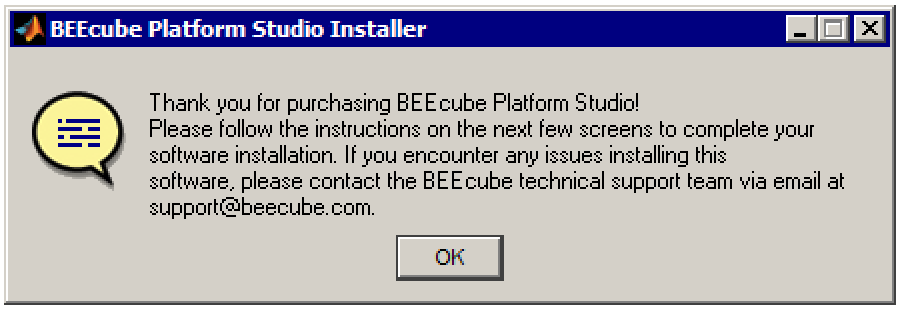


BEEcube Platform Studio installer

```
Matlab is now launching the BPS installer.  Please follow any
on-screen instructions to complete the BPS installation.

Press any key to continue . . .
```

Figure 5: Installer launch window

Depending on the speed of your system and/or network, it may take several seconds for Matlab to launch and start the installation. Once the installer is able verify the required tool versions and locate the runtime software archive, the following welcome dialog will appear.

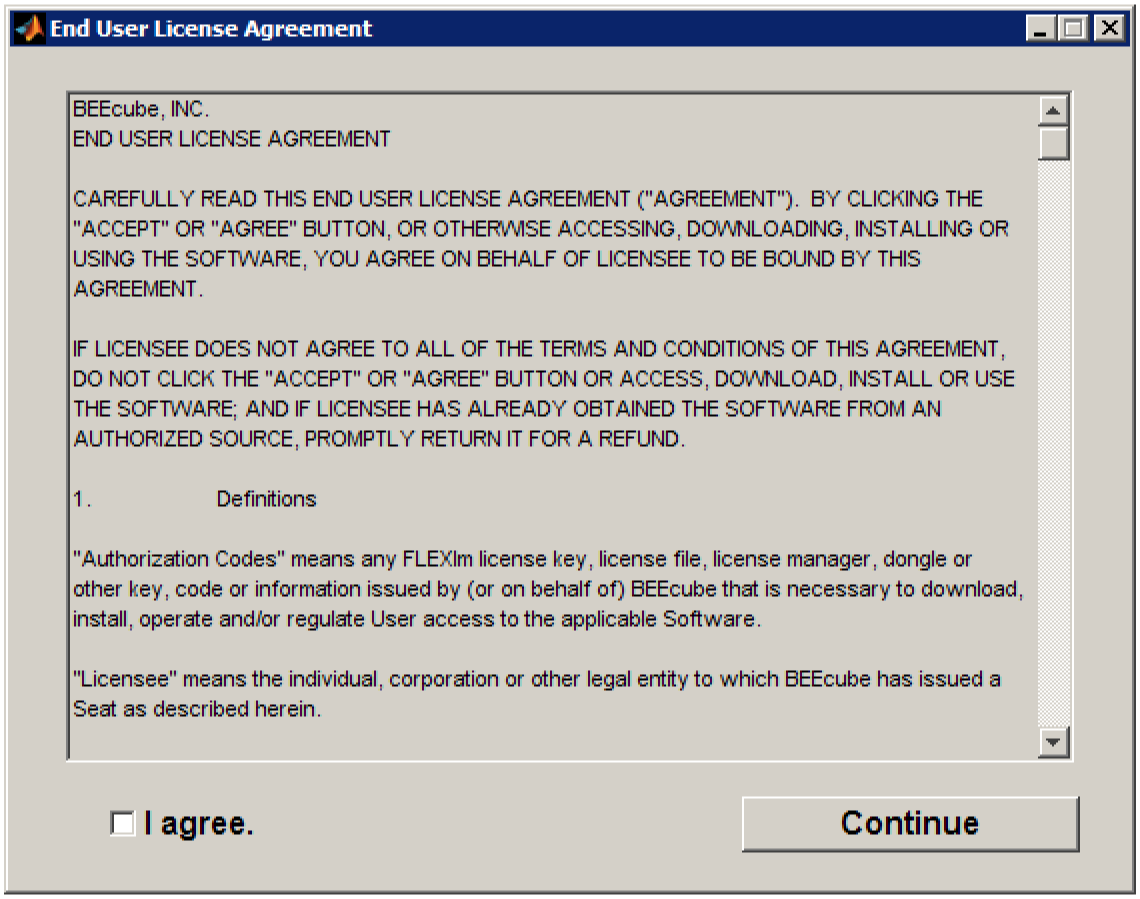


BEEcube Platform Studio Installer

Thank you for purchasing BEEcube Platform Studio!
Please follow the instructions on the next few screens to complete your software installation. If you encounter any issues installing this software, please contact the BEEcube technical support team via email at support@beecube.com.

Figure 6: Installer welcome window

Click “OK” to continue, and you will be asked to accept the End User License Agreement for this software, as shown in the following dialog.



BEEcube, INC.
END USER LICENSE AGREEMENT

CAREFULLY READ THIS END USER LICENSE AGREEMENT ("AGREEMENT"). BY CLICKING THE "ACCEPT" OR "AGREE" BUTTON, OR OTHERWISE ACCESSING, DOWNLOADING, INSTALLING OR USING THE SOFTWARE, YOU AGREE ON BEHALF OF LICENSEE TO BE BOUND BY THIS AGREEMENT.

IF LICENSEE DOES NOT AGREE TO ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT CLICK THE "ACCEPT" OR "AGREE" BUTTON OR ACCESS, DOWNLOAD, INSTALL OR USE THE SOFTWARE; AND IF LICENSEE HAS ALREADY OBTAINED THE SOFTWARE FROM AN AUTHORIZED SOURCE, PROMPTLY RETURN IT FOR A REFUND.

1. Definitions

"Authorization Codes" means any FLEXlm license key, license file, license manager, dongle or other key, code or information issued by (or on behalf of) BEEcube that is necessary to download, install, operate and/or regulate User access to the applicable Software.

"Licensee" means the individual, corporation or other legal entity to which BEEcube has issued a Seat as described herein.

Figure 7: Installer license agreement window

Click “I agree.”, and then “Continue” to accept the agreement and continue with the installation. You will then be prompted to select the destination directory for the installation, as shown below.

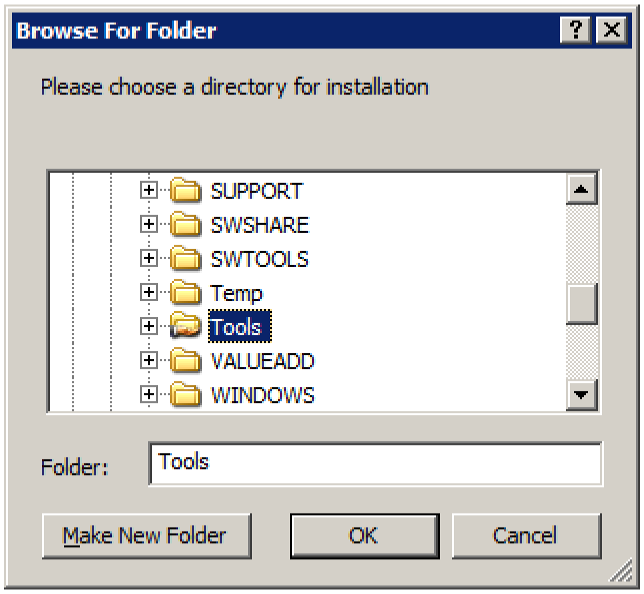
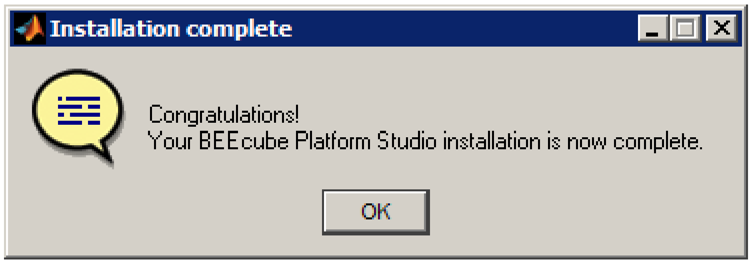


Figure 8: Installer destination directory window

If the selected directory is not already named `BPS`, the installer will create a subdirectory named `BPS` first before extracting the runtime software. Under the resulting directory named `BPS`, the installer will then extract the runtime software into a directory with the full name of the release. For example, if the directory `C:\Tools` was chosen from the "Browse for Folder" dialog, the installer will extract the software into the directory `C:\Tools\BPS\BPSv`*`NN`*`_r`*`DDDDDDDD`*, where *`NN`* is the BPS version number, and *`DDDDDDDD`* is the build date of the release. Once the software is successfully extracted, you will see the following dialog appear.



Congratulations!
Your BEEcube Platform Studio installation is now complete.

Figure 9: Installer complete window

The actual dialog that appears may vary, based on whether any warnings were encountered during the installation. If warnings were reported, please follow any instructions in the dialog to finalize the installation process.

Installing BPS Manually (Windows and Linux)

NOTE: *Manual installation of the BPS tools implies acceptance of the End User License Agreement for this software. Please read and review the terms of this license before proceeding with any of the steps in this section.*

You may unpack the BPS runtime software into any location on any filesystem which is accessible from the workstation(s) which will be running Matlab and the BPS tools. Note that the runtime software is included as its own archive within the complete BPS distribution. The archive filename will be in the format `BPSv`*`NN`*`_r`*`DDDDDDDD`*`.zip`, where *`NN`* is the BPS version number, and *`DDDDDDDD`* is the specific build date of the software.

In order to use the BPS tools, the user only needs to call the startup script included with BPS from within the Matlab environment. This script only needs to be called once per Matlab session, and will be called automatically if Matlab is launched from same directory in which the file `startup.m` exists.

For example, if you unpacked the BPS distribution into a directory named `C:\BPS`, you should either launch Matlab in the `C:\BPS` directory, or launch Matlab normally and call the `C:\BPS\startup.m` script before using any BPS blocks or features:

```
>> cd C:/BPS
>> startup
```

Once the startup script is called and run successfully, the Matlab console will display the following message to notify the user that BPS is ready for use:

```
BEEcube Platform Studio configuration complete
```

The primary purpose of the `startup.m` script is to set up the Matlab path to include all the BPS library functions and Blocksets. Therefore, if you are comfortable with the Matlab environment, you may also manually add the paths required by BPS to your own custom Matlab startup script or to the Matlab configuration for your own user account via the integrated Matlab path tools. This would prevent the need to call `startup.m` at the beginning of each Matlab session.

BPS Configuration Options

BPS references several environment variables which may be defined to customize your own computing environment.

- **BPS_LICENSE_PATH** A colon-separated list of directories to search for valid BPS license files. Each directory will be checked for a valid license file in the order specified.
- **BPS_BASELIB_PATH** A colon-separated list of directories to search for user-created custom base packages. When a design specifies the use of a custom base package, these directories will be searched for a matching base package before checking the default location.
- **BPS_USER_IP_IMPORT_PATH** A directory output initialization for pcores with the use of ip_import. Also the search location for valid imported pcores during Hardware generation.

License Installation

BPS requires a valid license to be detected on the system before performing various protected actions. When searching for a license file, BPS will check any directories listed in the BPS_LICENSE_PATH environment variable (described above) followed by the rest of the Matlab path. This gives each user the flexibility to store all BPS license files in a common location referenced by BPS_LICENSE_PATH, or locally on each machine licensed to run BPS.

Failure to detect a valid license file, or finding a license that has already expired, will cause an error and description of the license state to be displayed in the Matlab console. Please contact your BEEcube sales representative regarding any license-related issues.

Chapter 3: Using BPS

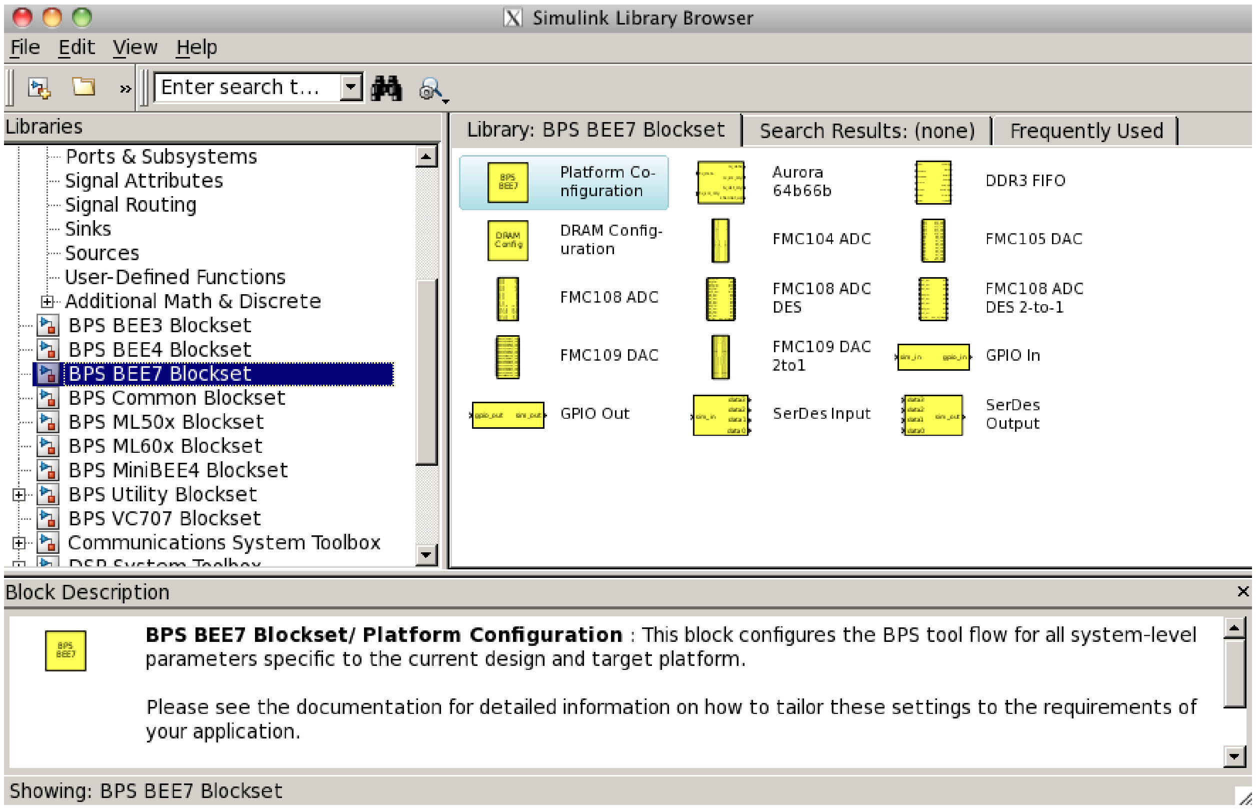
BPS provides the user with three sets of features to streamline the hardware design process. First is a collection of Simulink block libraries which provide simple, easy-to-use design abstractions for all the internal and external data interfaces available on each supported hardware platform. Second is an automated hardware/software generation flow which takes the Simulink-based model and converts the system into an FPGA implementation as a standard Xilinx EDK project. Third is an extensible software environment, the BPS NectarOS embedded shell, which provides runtime access to on-chip data resources for convenient system monitoring and debugging directly on the FPGA. The rest of this chapter describes each of these feature sets and how they work together within BPS.

Designing with Simulink and BPS Blocksets

In the Simulink design environment, each library of blocks is called a *blockset*. BPS provides its own blocksets which abstract away the low-level details of the underlying physical hardware and allow the user to communicate with a large number of data interfaces in a simple manner. This section discusses how to use the BPS blocksets to build hardware implementations with rich user-accessible interfaces.

Most BPS library blocks have attributes that you can specify individually to match your application. BPS associates a variable, or *block parameter*, with each user-configurable attribute of a block. You configure each attribute by setting its associated parameter to the desired value via the *block configuration dialog*, which can be accessed by double-clicking the block once it has been placed into a Simulink model.

In your Simulink Library Browser, you can find BPS blocks by selecting **BPS Common Blockset**, **BPS BEE3 Blockset**, **BPS BEE4 Blockset**, **BPS miniBEE4 Blockset**, **BPS BEE7 Blockset, BPS ML50x Blockset**, **BPS ML60x Blockset**, or **BPS Utility Blockset**.



BPS BEE7 Blockset/ Platform Configuration : This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Figure 10: BPS blocksets in Simulink Library Browser

The following sections present each of the different BPS blocksets and a brief description of the blocks provided by each. For a complete, detailed description of the non-utility blocks and all their customizable parameters, please refer to the BPS design component reference in Chapter 5.

BPS Utility Blockset

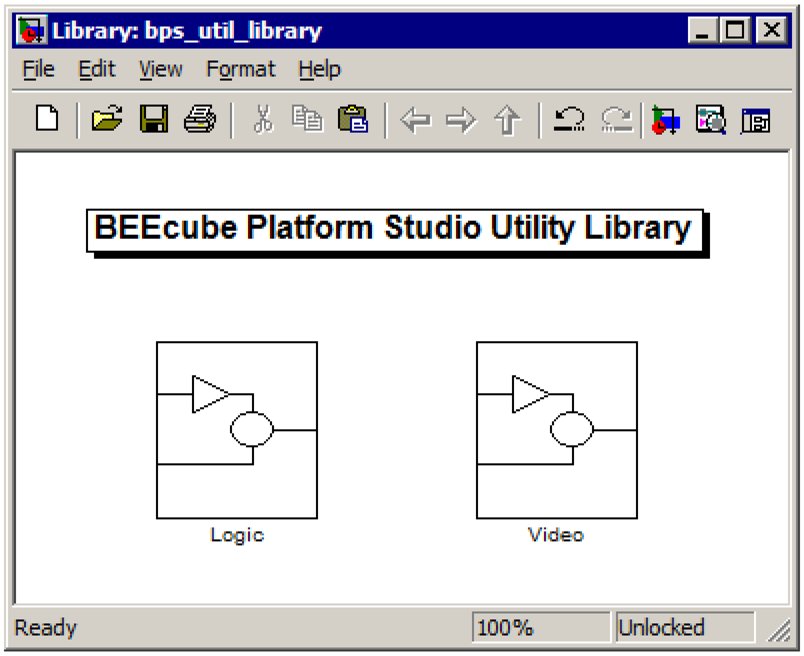

Library: bps_util_library
File Edit View Format Help
BEEcube Platform Studio Utility Library
Logic
Video
Ready
100%
Unlocked

Figure 11: BPS Utility Blockset

The BPS Utility Blockset contains a set of useful blocks which may be used to assist with the Xilinx System Generator portion of the hardware design for commonly used logic functions. Within the BPS Utility Blockset are two subsystems, one containing general logic functions (edge detection, pulse generators, etc.) and another containing video-specific functions (color space conversion, VGA timing generation, etc.). A brief description of each block can be seen by double-clicking the block, or by selecting the block in the Simulink Library Browser. To see the underlying implementation of each block, first add the block to a Simulink model, then right-click the block and select ‘Look Under Mask’.

BPS Common Blockset

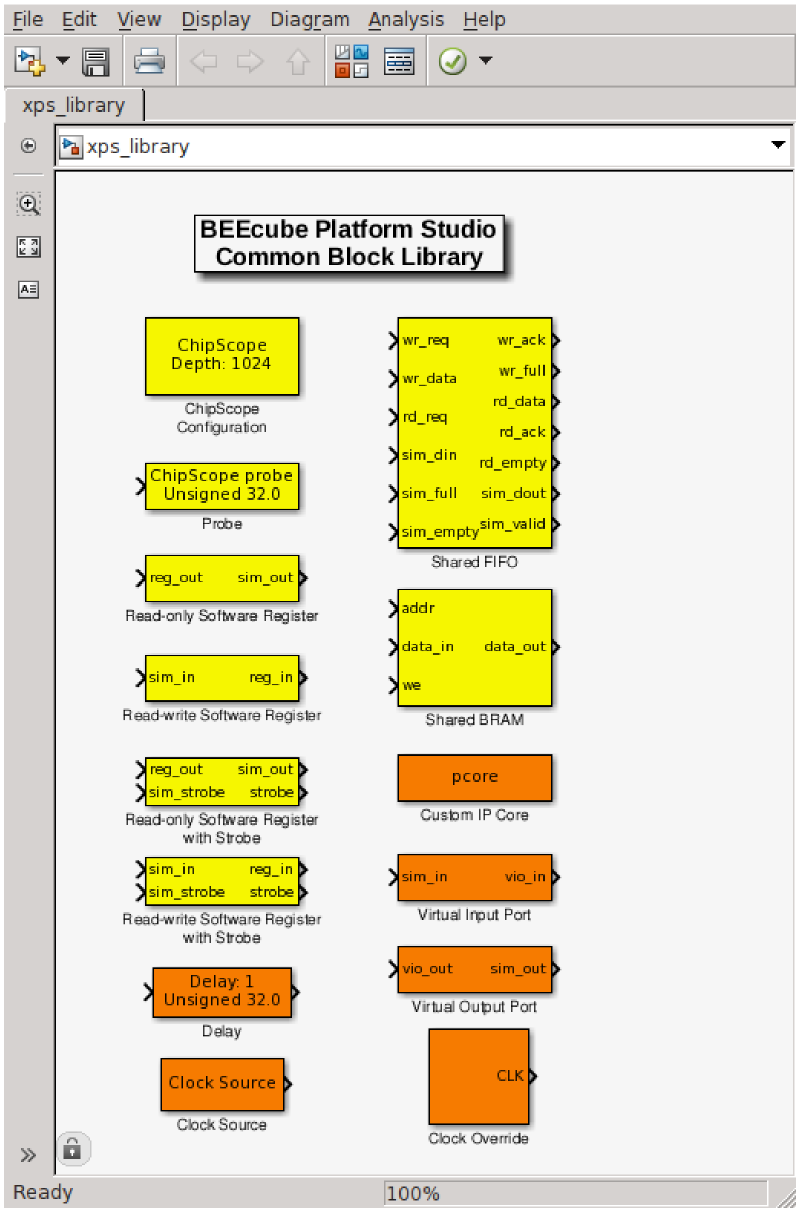


Figure 12: BPS Common Blockset

The BPS Common Blockset contains the general-purpose BPS library blocks that are not specific to the capabilities of the underlying hardware itself. This section describes all the functionality and parameters of each of the general-purpose blocks.

Table 1: Description of common BPS blocks

Block Name	**Description**
ChipScope Configuration	Enables support for ChipScope probes in the design.
Probe	Adds a ChipScope probe to a signal. Requires a ChipScope Configuration block at the top of the system.
Read-only Software Register	Creates a software-accessible 32-bit register which can be read by the embedded processor.
Read-write Software Register	Creates a software-accessible 32-bit register which can be read or written by the embedded processor.
Read-only Software Register with Strobe	Creates a software-accessible 32-bit register which can be read by the embedded processor. A strobe port is included for detecting accesses made by the host processor.
Read-write Software Register with Strobe	Creates a software-accessible 32-bit register which can be read or written by the embedded processor. A strobe port is included for detecting accesses made by the host processor.
Shared FIFO	Creates a software-accessible 32-bit FIFO which can be read and written from the embedded processor.
Shared BRAM	Creates a software-accessible 32-bit dual-port Block RAM which can be read and written from the embedded processor.
Custom IP Core	This block is not intended to be used directly except by advanced users who are familiar with the internals of BPS. Its primary purpose is to serve as a template for custom IP components which are automatically generated by the BPS IP Import Utility.
Virtual Input Port	This block is only intended for use by advanced users who plan on exporting the BPS-generated EDK project into an alternate design environment for further modification. An input port will be added to the EDK implementation which is not associated with any other interface component. This block is not safe for use when the EDK project is implemented by BPS, as an arbitrary I/O pin will be added to the FPGA which could have undesired effects on the hardware.
Virtual Output Port	This block is only intended for use by advanced users who plan on exporting the BPS-generated EDK project into an alternate design environment for further modification. An output port will be added to the EDK implementation which is not associated with any other interface component. This block is not safe for use when the EDK project is implemented by BPS, as an arbitrary I/O pin will be added to the FPGA which could have undesired effects on the hardware.

BPS BEE3 Blockset

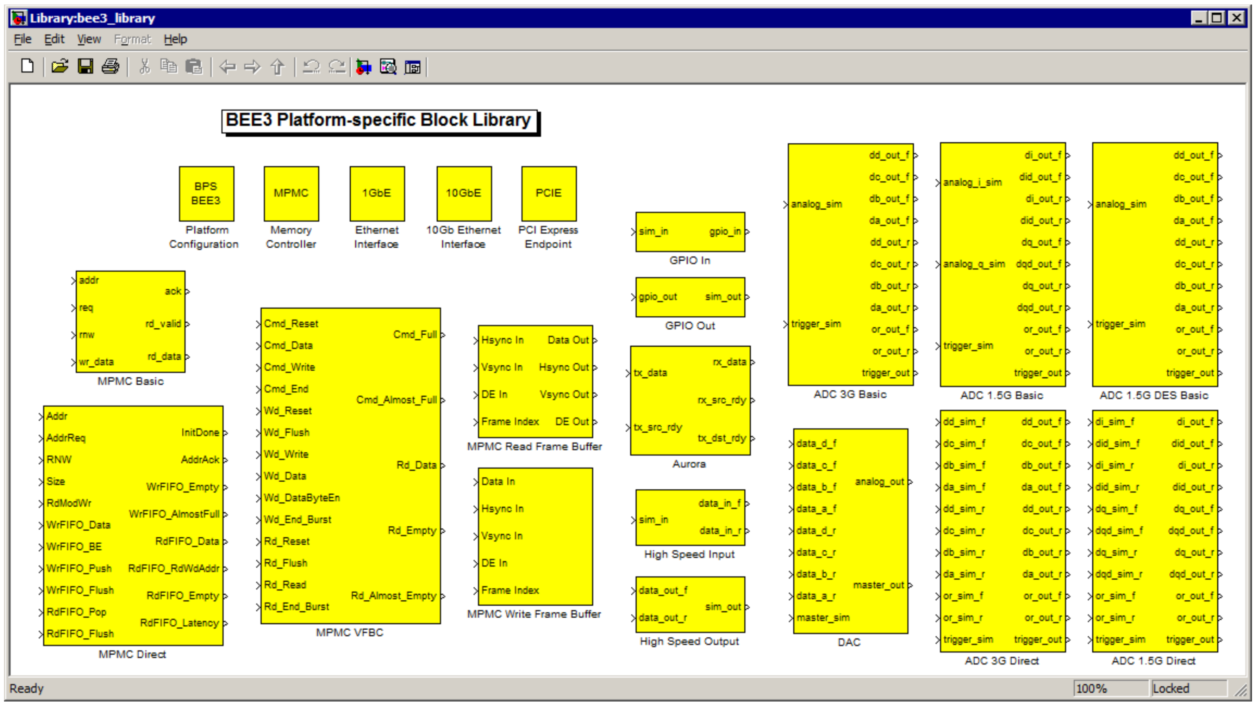


Figure 13: BPS BEE3 Blockset

BPS provides the BEE3-specific library blocks to support components and configurations specific to the BEE3 hardware platform. This section describes all the functionality and parameters of each of the BEE3-specific blocks.

Table 2: Description of BEE3-specific BPS blocks

Block Name	Description
Platform Configuration	Must exist at the top level of each Simulink system which represents the entire design contents of a single FPGA in the BEE3 system. Specifies system-level parameters such as target device, base package selection, and clock generation.
ADC 3G Basic ADC 1.5G Basic ADC 1.5G DES Basic	Adds a simplified interface to a BEE3 ADC expansion board with the corresponding ADC configuration. Includes an ADC Simulink simulation model and scales all outputs to be zero-biased with unit peak-to-peak magnitude.
ADC 3G Direct ADC 1.5G Direct	Adds a direct interface to a BEE3 ADC expansion board with the corresponding ADC configuration. The raw 8-bit ADC outputs are passed to the system without any bias or scaling adjustments.
DAC	Adds a direct interface to a BEE3 DAC expansion board. Raw 9-bit unsigned magnitude inputs are passed directly to the DAC without any bias or scaling adjustments.

Aurora	Inserts an Aurora streaming serial duplex I/O link that uses the FPGA Gigabit Transceivers connected to the front panel CX4 ports.
Ethernet Interface	Adds the hard MAC and FIFOs to support an Ethernet hardware interface for use by the embedded processor.
10Gb Ethernet Interface	Adds a XAUI-based 10Gb Ethernet PHY and custom MAC to the hardware implementation which includes embedded support for automated data transfer to shared memory components.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
High-speed I/O (HSIO)	Creates an automatically-calibrated high-speed data interface over one of the available inter-FPGA buses. Speeds up to DDR800 are supported with a total of up to 90 bits.
Memory Controller	Creates a multi-port memory controller (MPMC) for DDR2 DRAM support. The initial port is connected to main PLB bus for use by the embedded processor. The physical memory channel and DIMM slot are selected as parameters.
MPMC Basic	Adds a simplified memory port connection between the hardware design and MPMC. The data port width and burst length are selected as parameters.
MPMC Direct	Adds a direct Native Port Interface (NPI) connection between the hardware design and MPMC. The data port width is selected as a parameter.
MPMC Read Frame Buffer	Adds a simplified 32-bit read-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC Write Frame Buffer	Adds a simplified 32-bit write-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC VFBC	Adds a direct frame buffer interface between the hardware design and MPMC. The command and data FIFO depths and data port width are set as parameters.
PCI Express Endpoint	Adds the PCIE-PLB bridge and central DMA controller to support a PCI Express Endpoint hardware interface.

BPS ML50x Blockset

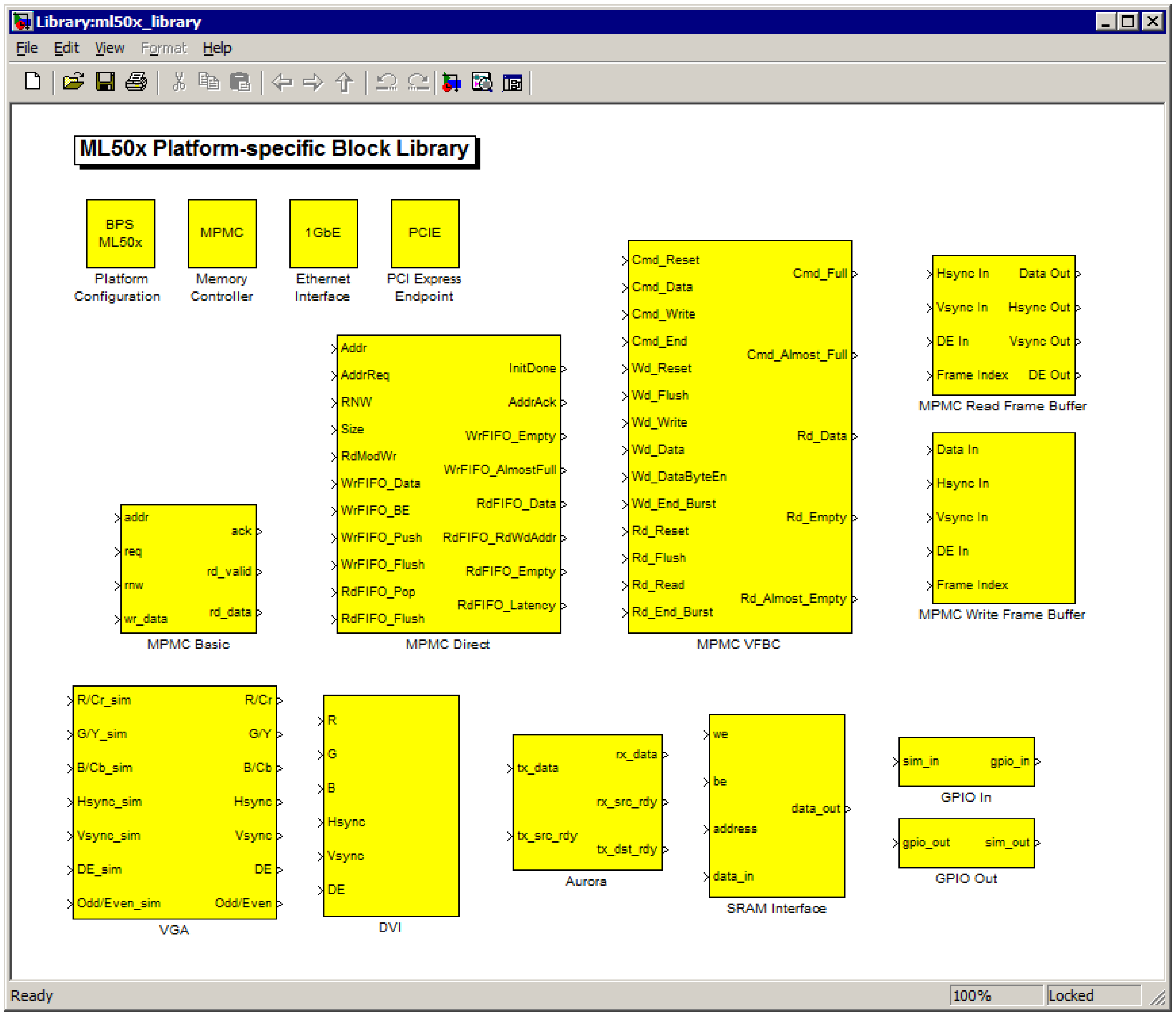

Library:ml50x_library
File Edit View Format Help
ML50x Platform-specific Block Library
BPS ML50x
Platform Configuration
MPMC
Memory Controller
1GbE
Ethernet Interface
PCIE
PCI Express Endpoint
addr req rnw wr_data ack rd_valid rd_data
MPMC Basic
Addr AddrReq RNW Size RdModWr WrFIFO_Data WrFIFO_BE WrFIFO_Push WrFIFO_Flush RdFIFO_Pop RdFIFO_Flush
InitDone AddrAck WrFIFO_Empty WrFIFO_AlmostFull RdFIFO_Data RdFIFO_RdWdAddr RdFIFO_Empty RdFIFO_Latency
MPMC Direct
Cmd_Reset Cmd_Data Cmd_Write Cmd_End Wd_Reset Wd_Flush Wd_Write Wd_Data Wd_DataByteEn Wd_End_Burst Rd_Reset Rd_Flush Rd_Read Rd_End_Burst
Cmd_Full Cmd_Almost_Full Rd_Data Rd_Empty Rd_Almost_Empty
MPMC VFBC
Hsync In Vsync In DE In Frame Index Data Out Hsync Out Vsync Out DE Out
MPMC Read Frame Buffer
Data In Hsync In Vsync In DE In Frame Index
MPMC Write Frame Buffer
R/Cr_sim G/Y_sim B/Cb_sim Hsync_sim Vsync_sim DE_sim Odd/Even_sim
R/Cr G/Y B/Cb Hsync Vsync DE Odd/Even
VGA
R G B Hsync Vsync DE
DVI
tx_data tx_src_rdy rx_data rx_src_rdy tx_dst_rdy
Aurora
we be address data_in data_out
SRAM Interface
sim_in gpio_in
GPIO In
gpio_out sim_out
GPIO Out
Ready
100%
Locked

Figure 14: BPS ML50x Blockset

BPS provides Xilinx ML50x-specific library blocks to support components and configurations specific to the Xilinx ML505, ML506, ML507, and XUPV5-LX110T (ML509) Evaluation Platform hardware boards. This section describes all the functionality and parameters of each of the Xilinx ML50x-specific blocks.

Table 3: Description of ML50x-specific BPS blocks

Block Name	**Description**
Platform Configuration	Must exist at the top level of the Simulink system which represents the entire design contents to be implemented on the ML50x board. Specifies system-level parameters such as target hardware board, base package selection, and clock generation.
Aurora	Inserts an Aurora streaming serial duplex I/O link that uses the FPGA Gigabit Transceivers connected to the SATA ports.

DVI	Implements a Digital Video Interface output via the Chrontel CH7301C device to the DVI port.
Ethernet Interface	Adds the hard MAC and FIFOs to support an Ethernet hardware interface for use by the embedded processor.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
Memory Controller	Creates a multi-port memory controller (MPMC) for DDR2 DRAM support. The initial port is connected to main PLB bus for use by the embedded processor. The physical memory channel and DIMM slot are selected as parameters.
MPMC Basic	Adds a simplified memory port connection between the hardware design and MPMC. The data port width and burst length are selected as parameters.
MPMC Direct	Adds a direct Native Port Interface (NPI) connection between the hardware design and MPMC. The data port width is selected as a parameter.
MPMC Read Frame Buffer	Adds a simplified 32-bit read-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC Write Frame Buffer	Adds a simplified 32-bit write-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC VFBC	Adds a direct frame buffer interface between the hardware design and MPMC. The command and data FIFO depths and data port width are set as parameters.
PCI Express Endpoint	Adds the PCIE-PLB bridge and central DMA controller to support a PCI Express Endpoint hardware interface.
SRAM Interface	Creates a memory port to the 256K x 36 ZBT Synchronous SRAM device.
VGA	Inserts an interface to the VGA analog video input port via the Analog Devices AD9980 device.

BPS BEE4 Blockset

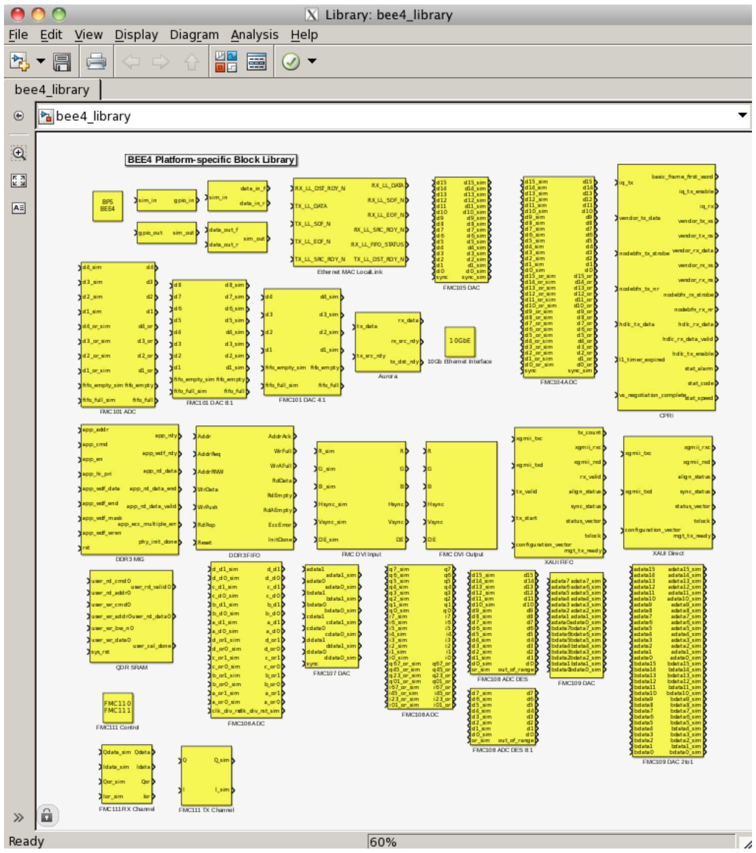


Figure 15: BPS BEE4 Blockset

BPS provides the BEE4-specific library blocks to support components and configurations specific to the BEE4 hardware platform. This section describes all the functionality and parameters of each of the BEE4-specific blocks.

Table 4: Description of BEE4-specific BPS blocks

Block Name	**Description**
Platform Configuration	Must exist at the top level of each Simulink system which represents the entire design contents of a single FPGA in the BEE4 system. Specifies system-level parameters such as target device, base package selection, and clock generation.
FMC101 ADC	Adds a direct interface to the ADC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC101 DAC 4:1 FMC101 DAC 8:1	Adds a direct interface to the DAC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary DAC inputs are passed out of the system without any bias or scaling adjustments.

FMC104 ADC	Adds a direct interface to the ADC on the BEEcube FMC104 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC105 DAC	Adds a direct interface to the DAC device on the BEEcube FMC105 expansion board. The raw 12-bit offset binary DAC outputs are passed out of the system without any bias or scaling adjustments.
FMC 106 ADC	Adds a direct interface to the ADC on the BEEcube FMC106 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC111 Control	Instantiates control busses and selects between the BEEcube FMC110 or FMC111.
FMC111 RX Channel	Adds a direct interface to the BEEcube FMC110 or FMC111 ADC. This block delivers one I and one Q digital value, per FPGA clock cycle. These values are sampled by the ADC at the system clock rate. The ADC device itself samples at its input clock rate.
FMC111 TX Channel	Adds a direct interface to the BEEcube FMC110 or FMC111 DAC. This interface accepts two 16 bit digital words per FPGA clock cycle. The DAC devices themselves each feature a 16 bit data bus, which the FPGA drives with 2:1 DDR signal, delivering one I and one Q sample per FPGA clock.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
High-speed I/O (HSIO)	Creates an automatically-calibrated high-speed data interface over one of the available inter-FPGA buses. Speeds up to DDR400 are currently supported.
Ethernet MAC LocalLink	Adds a hardware interface to the integrated Virtex-6 Ethernet MAC. A Xilinx LocalLink packet FIFO interface is used for asynchronous clocking and to simplify the control protocol
XAUI DIrect	Creates a Xilinx XAUI core and connects the XGMII interface directly to the user logic. The system clock must be locked to the XAUI clock. Made for interfacing over the QSFP ports.
XAUI FIFO	Adds an additional custom FIFO interface between the user logic and the Xilinx XAUI core. Allows the system clock to run independent of the XAUI clock.
10Gb Ethernet Interface	Adds a XAUI-based 10Gb Ethernet PHY and custom MAC to the hardware implementation which includes embedded support for automated data transfer to shared memory components.
Aurora	Inserts an Aurora streaming serial duplex I/O link that uses the FPGA Gigabit Transceivers connected to the QSFP, FMC, or inter-FPGA interfaces.

DDR3 MIG	Creates a Xilinx MIG DDR3 memory controller on the FPGA and connects the control interface directly to the user design. Produces a highly efficient implementation, but the system clock must be locked to the memory clock.
DDR3 FIFO	Adds an additional custom FIFO interface between the user logic and the Xilinx MIG DDR3 memory controller. Provides a simpler data protocol and allows the system clock to run independent of the memory clock.
CPRI	Adds a hardware and software interface to a Xilinx CPRI v3.2 core to the system. Both Master and Slave cores are supported at line rates of 2.4576, 4.9152, and 6.1440 Gbps. The implementation is tailored for use with the BEEcube PCIE101 PCI-Express to SFP+ breakout card.
FMC DVI Input	Adds a direct interface to the TFP403 TMDS de-serializer input device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock is taken directly from the board, and a simple video frame port interface is provided to the user.
FMC DVI Output	Adds a direct interface to the TFP410 TMDS serializer output device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock used for the outgoing stream is taken directly from the system clock source of the FPGA implementation. A simple video frame port interface is provided to the user.

BPS miniBEE4 Blockset

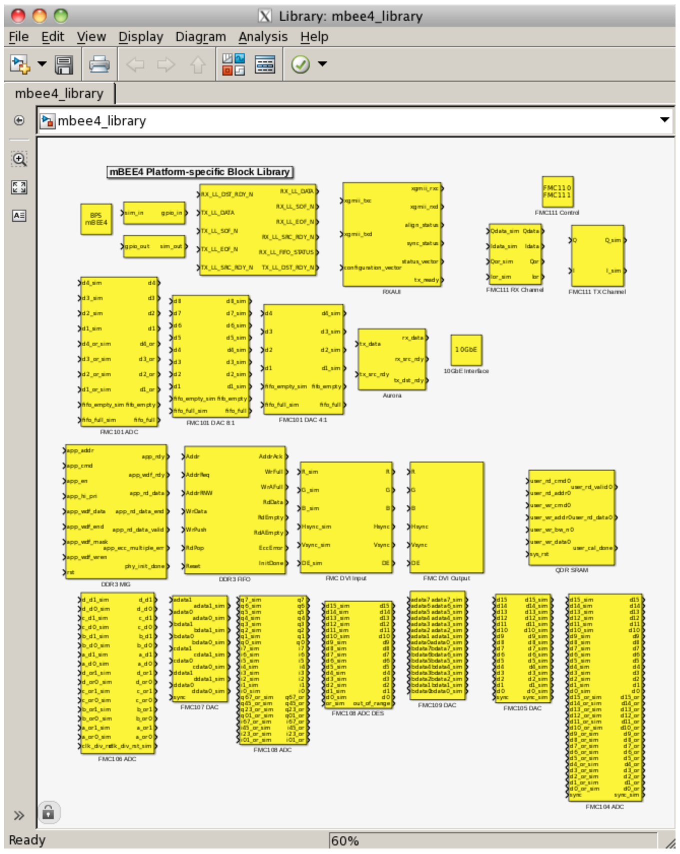


Figure 16: BPS miniBEE4 Blockset

BPS provides the BEE4-specific library blocks to support components and configurations specific to the BEE4 hardware platform. This section describes all the functionality and parameters of each of the BEE4-specific blocks.

Table 5: Description of miniBEE4-specific BPS blocks

Block Name	**Description**
Platform Configuration	Must exist at the top level of each Simulink system which represents the entire design contents of a single FPGA in the miniBEE4 system. Specifies system-level parameters such as target device, base package selection, and clock generation.
FMC101 ADC	Adds a direct interface to the ADC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC101 DAC 4:1 FMC101 DAC 8:1	Adds a direct interface to the DAC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary DAC inputs are passed out of the system without any bias or scaling adjustments.

FMC104 ADC	Adds a direct interface to the ADC on the BEEcube FMC104 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC105 DAC	Adds a direct interface to the DAC device on the BEEcube FMC105 expansion board. The raw 12-bit offset binary DAC outputs are passed out of the system without any bias or scaling adjustments.
FMC 106 ADC	Adds a direct interface to the ADC on the BEEcube FMC106 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC111 Control	Instantiates control busses and selects between the BEEcube FMC110 or FMC111.
FMC111 RX Channel	Adds a direct interface to the BEEcube FMC110 or FMC111 ADC. This block delivers one I and one Q digital value, per FPGA clock cycle. These values are sampled by the ADC at the system clock rate. The ADC device itself samples at its input clock rate.
FMC111 TX Channel	Adds a direct interface to the BEEcube FMC110 or FMC111 DAC. This interface accepts two 16 bit digital words per FPGA clock cycle. The DAC devices themselves each feature a 16 bit data bus, which the FPGA drives with 2:1 DDR signal, delivering one I and one Q sample per FPGA clock.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
Ethernet MAC LocalLink	Adds a hardware interface to the integrated Virtex-6 Ethernet MAC. A Xilinx LocalLink packet FIFO interface is used for asynchronous clocking and to simplify the control protocol
RXAUI	Creates a Xilinx RXAUI core and connects the XGMII interface directly to the user logic. The system clock must be locked to the RXAUI clock. Made for interfacing with the NLP 10 GbE PHY and the SFP+ ports.
10Gb Ethernet Interface	Adds a XAUI-based 10Gb Ethernet PHY and custom MAC to the hardware implementation which includes embedded support for automated data transfer to shared memory components.
Aurora	Inserts an Aurora streaming serial duplex I/O link that uses the FPGA Gigabit Transceivers connected to the QSFP, FMC, or inter-FPGA interfaces.
DDR3 MIG	Creates a Xilinx MIG DDR3 memory controller on the FPGA and connects the control interface directly to the user design. Produces a highly efficient implementation, but the system clock must be locked to the memory clock.

DDR3 FIFO	Adds an additional custom FIFO interface between the user logic and the Xilinx MIG DDR3 memory controller. Provides a simpler data protocol and allows the system clock to run independent of the memory clock.
FMC DVI Input	Adds a direct interface to the TFP403 TMDS de-serializer input device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock is taken directly from the board, and a simple video frame port interface is provided to the user.
FMC DVI Output	Adds a direct interface to the TFP410 TMDS serializer output device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock used for the outgoing stream is taken directly from the system clock source of the FPGA implementation. A simple video frame port interface is provided to the user.

BPS BEE7 Blockset

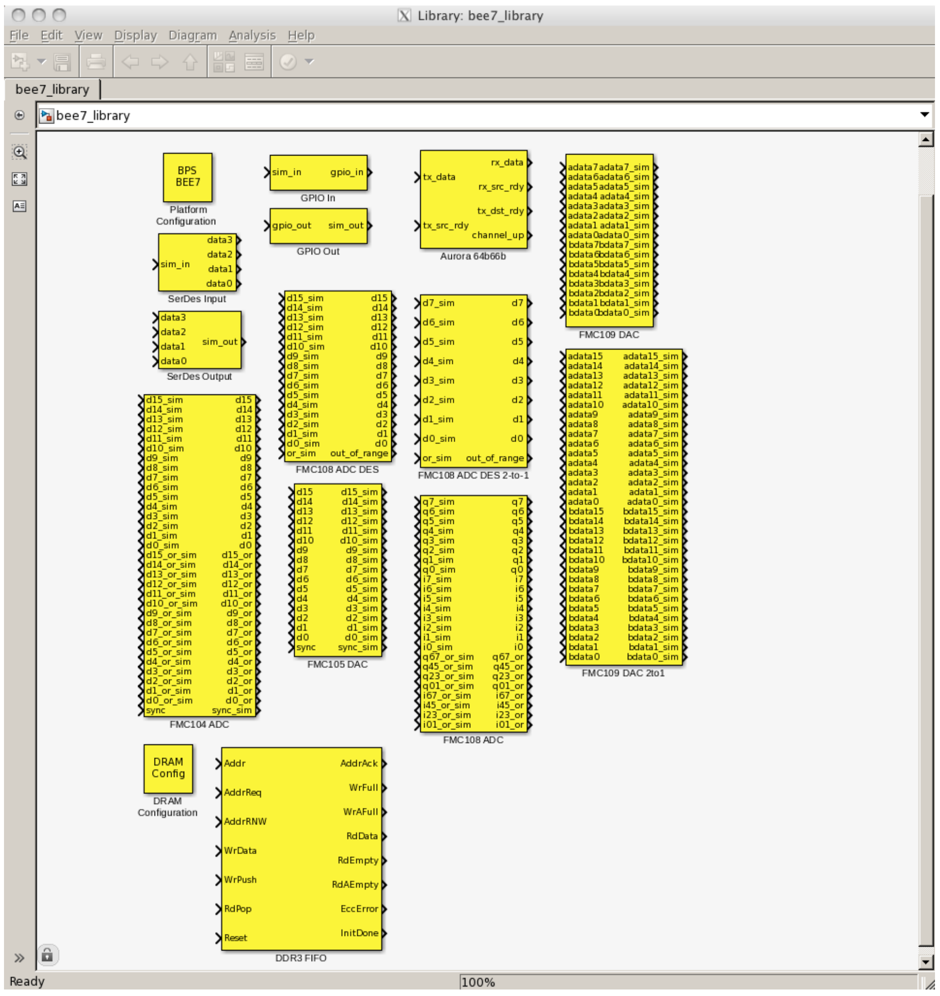


Figure 17: BPS BEE7 Blockset

BPS provides the BEE7-specific library blocks to support components and configurations specific to the BEE7 hardware platform. This section describes all the functionality and parameters of each of the BEE7-specific blocks.

Table 6: Description of BEE7-specific BPS blocks

Block Name	**Description**
Platform Configuration	Must exist at the top level of each Simulink system which represents the entire design contents of a single FPGA in the BEE7 system. Specifies system-level parameters such as target device, base package selection, and clock generation.
FMC104 ADC	Adds a direct interface to the ADC on the BEEcube FMC104 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC105 DAC	Adds a direct interface to the DAC device on the BEEcube FMC105 expansion board. The raw 12-bit offset binary DAC outputs are passed out of the system without any bias or scaling adjustments.
FMC 106 ADC	Adds a direct interface to the ADC on the BEEcube FMC106 expansion board. The raw 10-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC 108 ADC	Adds a direct interface to the ADC device on the BEEcube FMC108 expansion board. The digital outputs are fixed to be raw format delivered by the ADC, a 12-bit offset binary integer. In DESI and DESQ modes, the samples are interleaved, with the q sample first (e.g. q0, i0, q1, i1, etc.).
FMC 108 ADC DES	Adds a direct interface to the ADC device on the BEEcube FMC108 expansion board. The digital outputs are fixed to be raw format delivered by the ADC, a 12-bit offset binary integer.
FMC 108 ADC DES 2-to-1	Adds a direct interface to the ADC device on the BEEcube FMC108 expansion board. The digital outputs are fixed to be raw format delivered by the ADC, a 12-bit offset binary integer.
FMC 109 DAC	Adds a direct interface to the DAC device on the BEEcube FMC109 expansion board. Instantiates a FMC109 DAC interface for up to 2.5 GSps operation.
FMC 109 DAC 2-to-1	Adds a direct interface to the DAC device on the BEEcube FMC109 expansion board. Instantiates a FMC109 DAC interface for up to 2.5 GSps operation.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
SerDes I/O	Creates an automatically-calibrated high-speed data interface over one of the available inter-FPGA buses. Speeds up to DDR1000 are currently supported.

Aurora 64b66b	Inserts an Aurora streaming serial duplex I/O link that uses the FPGA Gigabit Transceivers connected to the IMOT, RTM, FMC, or inter-FPGA interfaces.
DDR3 Configuration	Creates a Xilinx MIG DDR3 memory controller for the selected DIMMs on the FPGA and connects the SMRI interface to DRAM internally.
DDR3 FIFO	Adds an additional custom FIFO interface between the user logic and the Xilinx MIG DDR3 memory controller. Provides a simple data protocol and allows the system clock to run independent of the memory clock.

BPS ML60x Blockset

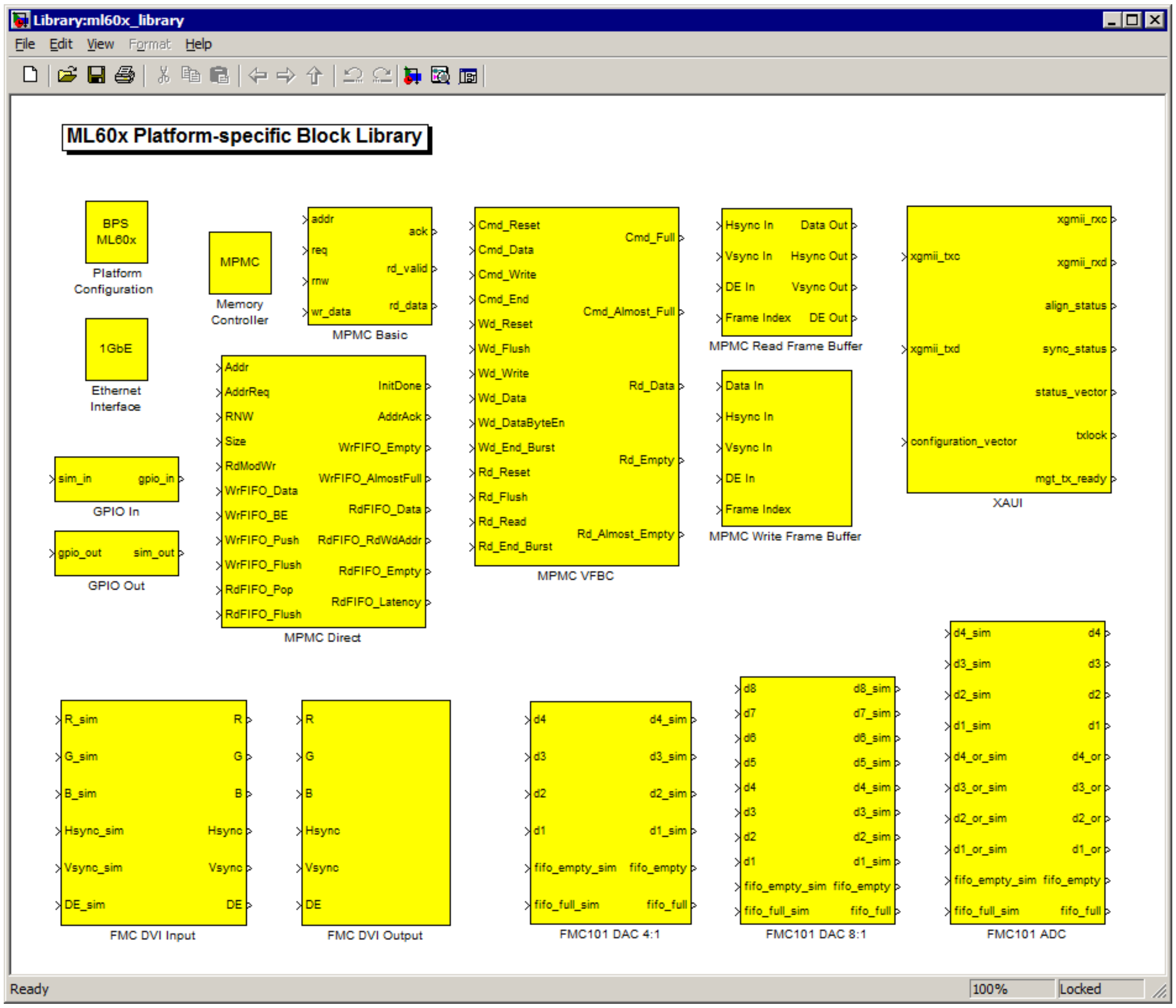


Figure 18: BPS ML60x Blockset

BPS provides Xilinx ML60x-specific library blocks to support components and configurations specific to the Xilinx ML605 Evaluation Platform hardware board. This section describes all the functionality and parameters of each of the Xilinx ML60x-specific blocks.

Table 7: Description of ML60x-specific BPS blocks

Block Name	**Description**
Platform Configuration	Must exist at the top level of the Simulink system which represents the entire design contents to be implemented on the ML50x board. Specifies system-level parameters such as target hardware board, base package selection, and clock generation.
Ethernet Interface	Adds the hard MAC and FIFOs to support an Ethernet hardware interface for use by the embedded processor.

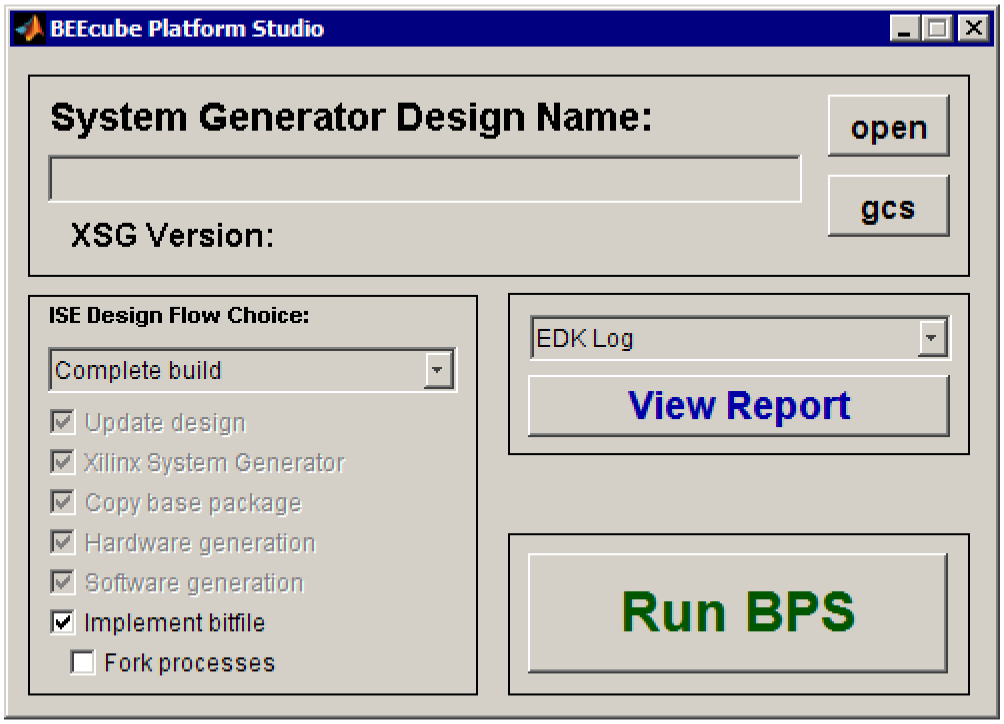
FMC101 ADC	Adds a direct interface to the ADC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary ADC outputs are passed into the system without any bias or scaling adjustments.
FMC101 DAC 4:1 FMC101 DAC 8:1	Adds a direct interface to the DAC device on the BEEcube FMC101 expansion board. The raw 12-bit offset binary DAC inputs are passed out of the system without any bias or scaling adjustments.
General Purpose I/O (GPIO)	Creates a direct connection to FPGA I/O pins. The I/O pin group, offset, direction, sample phase, and DDR signaling are selected as parameters.
Memory Controller	Creates a multi-port memory controller (MPMC) for DDR2 DRAM support. The initial port is connected to main PLB bus for use by the embedded processor. The physical memory channel and DIMM slot are selected as parameters.
MPMC Basic	Adds a simplified memory port connection between the hardware design and MPMC. The data port width and burst length are selected as parameters.
MPMC Direct	Adds a direct Native Port Interface (NPI) connection between the hardware design and MPMC. The data port width is selected as a parameter.
MPMC Read Frame Buffer	Adds a simplified 32-bit read-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC Write Frame Buffer	Adds a simplified 32-bit write-only frame buffer connection between the hardware design and MPMC. The screen geometry and buffer depth are selected as parameters.
MPMC VFBC	Adds a direct frame buffer interface between the hardware design and MPMC. The command and data FIFO depths and data port width are set as parameters.
XAUI	Direct XGMII and control interface to a Xilinx XAUI core, configured specifically for communicating with the NetLogic AEL2005 10-gigabit PHY devices on the HiTech Global HTG-FMC-SFP-PLUS board (in conjunction with a BEEcube FMC102 voltage translator board to provide Virtex-6 compliant access to the device’s MDIO port).
FMC DVI Input	Adds a direct interface to the TFP403 TMDS de-serializer input device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock is taken directly from the board, and a simple video frame port interface is provided to the user.
FMC DVI Output	Adds a direct interface to the TFP410 TMDS serializer output device of a Xilinx/Avnet AES-FMC-DVI-G FMC module. The video clock used for the outgoing stream is taken directly from the system clock source of the FPGA implementation. A simple video frame port interface is provided to the user.

Generating an FPGA Implementation

BPS provides tools at multiple levels to assist with the generation of an FPGA implementation from a Simulink model. The entry point into this implementation infrastructure is the *BPS implementation GUI*, which is a simple MATLAB user interface that provides direct access to each phase of the FPGA implementation flow. The user interface enables fully automatic compilation from the Simulink BPS design to the final hardware bitfile.

In addition to the BPS implementation GUI, a number of scripts are also generated by BPS to allow manual control over the back-end FPGA tools. The FPGA implementation itself is generated by BPS as a standard Xilinx EDK project for Virtex 6 and before or a Xilinx Vivado project for Virtex 7 and later, which can be built into an FPGA bitfile by Xilinx Platform Studio (XPS) on any machine capable of running the EDK/Vivado tools. In this manner, the back-end tool flow can be run on a different machine than BPS, such as a remote compute server or cluster. Finally, several programming scripts are also generated by BPS which can be used to download the final bitfile and/or software image onto the hardware platform, based on the platform configuration options specified for the design. The FPGA implementation output generated by BPS is discussed later in this section.

BPS Implementation GUI



ISE Design Flow Choice: Complete build

- [✓] Update design
- [✓] Xilinx System Generator
- [✓] Copy base package
- [✓] Hardware generation
- [✓] Software generation
- [✓] Implement bitfile
- [ ] Fork processes

Figure 19: BPS Implementation GUI

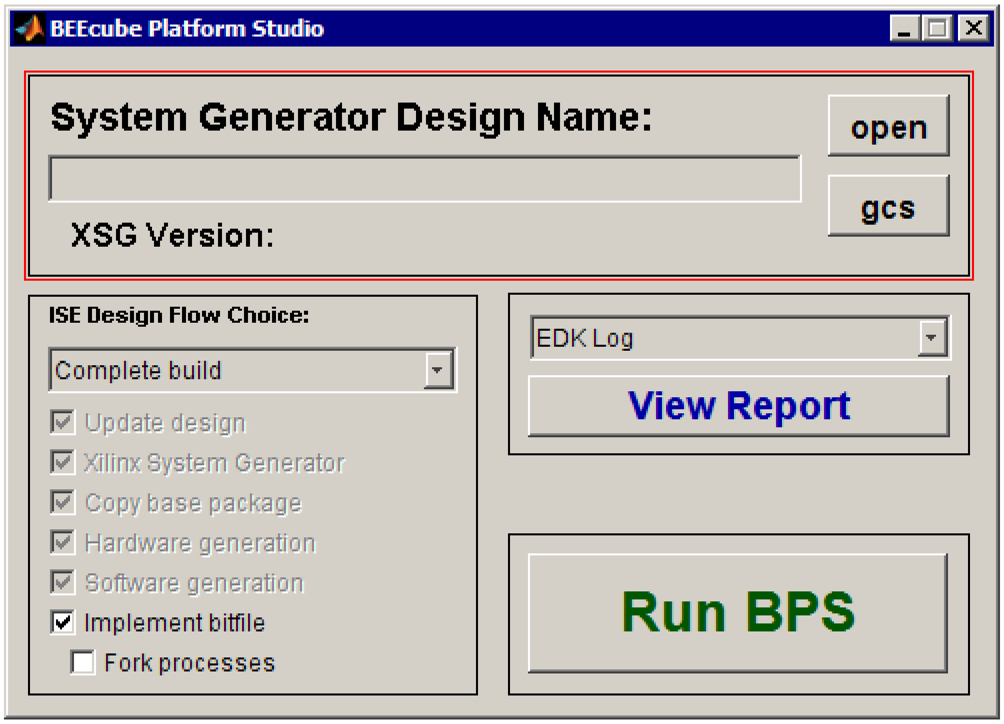
The BPS dialog box is split into the following panels:

- System Generator Design
- Design Flow Options
- Run BPS
- Report Analysis

System Generator Design

The System Generator Design Name panel allows you to select which Simulink design to compile under BPS. You can do one of the following:

- Type the name of a Simulink model in the text box and click **open**.
- Click **gcs** to automatically select the currently active Simulink model.



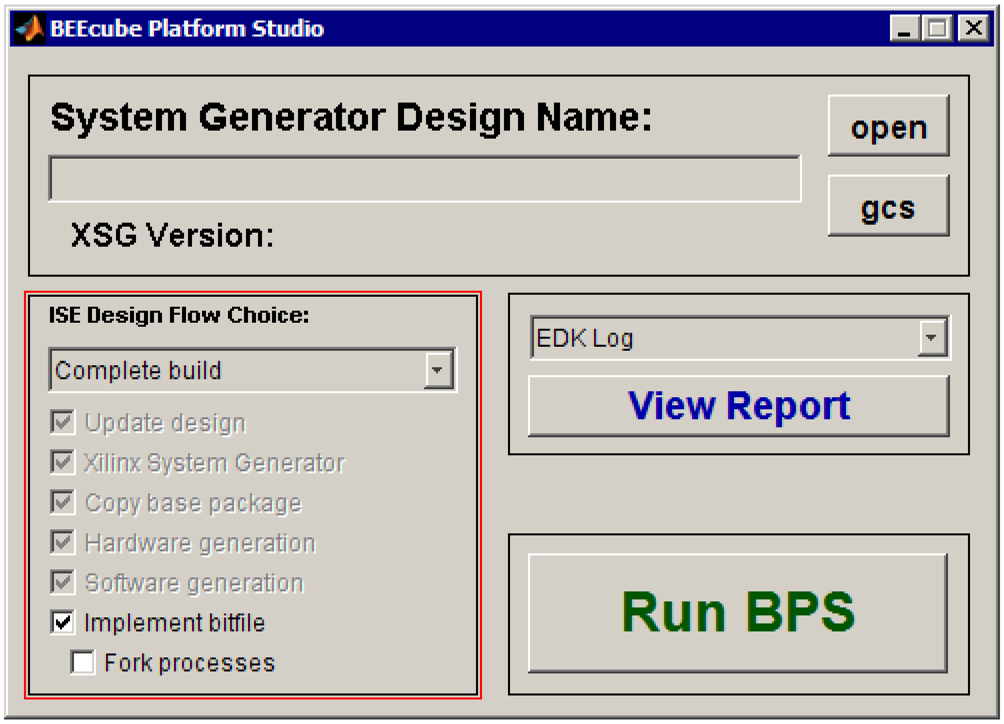
- [✓] Update design
- [✓] Xilinx System Generator
- [✓] Copy base package
- [✓] Hardware generation
- [✓] Software generation
- [✓] Implement bitfile
- [ ] Fork processes

Figure 20: System Generator Design panel

Design Flow Options

From the ISE Design Flow Choice panel, you can choose from several predefined tool flow choices in the drop-down list:

- **Complete build** runs all necessary steps to generate the final FPGA bitfile. This option must be used the first time a design is built to create the proper directory structure. Note that *Copy Base Package* is enabled for this option, so if this option is selected for an existing system, any intermediate output data or custom modifications to the implementation directory will be lost.
- **Rebuild existing design** runs all steps other than *Copy Base Package*, which can result in a shorter back-end execution time since unchanged core instances do not need to be regenerated.
- **Rerun software generation** runs only the BPS software generation step. This option is useful whenever new custom software files are added to the project and need to be stitched in by BPS. This will cause the embedded software to be recompiled and merged into the FPGA bitfile, but should not trigger the hardware implementation flow.
- **Design rule check** updates the Simulink model and runs all BPS DRC checks on the design. This option is useful for checking the validity of a system incrementally during the design process.
- **Custom** will enable all individual design flow phases to be selected manually. This option is reserved for advanced users who understand each phase of the implementation process used by BPS, but can result in errors if not driven properly.



ISE Design Flow Choice: Complete build

- [✓] Update design
- [✓] Xilinx System Generator
- [✓] Copy base package
- [✓] Hardware generation
- [✓] Software generation
- [✓] Implement bitfile
- [ ] Fork processes

Figure 21: Design Flow Options panel

Note that most individual design flow phases cannot be modified for all build options other than *Custom*. This is done to simplify the build process and prevent potential errors due to incorrect flow settings. Each of the individual phases is further described below.

Table 8: Description of BPS build phases

Build Phase	Description
Update design	Perform a diagram update in Simulink, create all BPS block objects, and perform a BPS DRC. This phase will automatically be run before any other standard build phases to guarantee that all blocks are up-to-date and have valid configurations.
Xilinx System Generator	Run Xilinx System Generator (XSG) to create a synthesized netlist for the XSG portion of the design. This phase is typically required whenever any changes have been made to the Simulink model.
Copy base package	Delete the current EDK build directory and extract a fresh copy of the selected base package. Note that this will delete any intermediate implementation files and/or manually edited design files. The build directory should be backed up or renamed before running this phase to preserve any existing data.
Hardware generation	Generate hardware implementations for all BPS library blocks and add them on top of the base package. Copies of all original file versions from the base package are preserved with a `.bac` extension. Also copies over any imported IP pcores needed for the design if the environment variable BPS_USER_IP_IMPORT_PATH is set.
Software generation	Generate all necessary extensions to NectarOS and the EDK/Vivado project to provide software support for all BPS blocks in the design. If custom user source code has been modified or added to the design, this phase must be rerun to incorporate any relevant changes into the software environment. Copies of all original file versions from the base package are preserved with a `.bac` extension.
Implement bitfile	Runs Xilinx Platform Studio (XPS) in the EDK build directory to generate the final FPGA bitfile for Virtex 6 and earlier, or Vivado for Virtex 7 and later. Note that this option is user modifiable for all design flow options other than *Design Rule Check*. This is done so that the user has the option of running the back-end tools on a different machine from where BPS is running. By not checking this box, BPS will not launch the back-end tool flow. The user may then run the back-end tools manually on any other machine.
Fork processes	When this option is checked, the back-end XPS/Vivado process for bitfile generation will be run in a separate command window and BPS will return without waiting to check the results. When this option is not checked, the XPS process will be run directly in the Matlab console and BPS will wait until the build has completed, check the critical report logs, and copy the final result into the `bitfiles` directory. This option has no effect on any other phases of the build process. Note that this option is only available when *Implement bitfile* is selected.

Run BPS

Clicking the **Run BPS** button will launch the BPS tool flow and perform all the necessary steps to complete each build phase enabled in the Design Flow Options panel for each FPGA design in the selected system. A Simulink model can potentially contain multiple parallel systems, each of which contains a Platform Configuration block at the top level. BPS will identify each of these independent systems as a unique FPGA design and execute all selected build phases for each design sequentially. Please refer to the walkthrough in Chapter 4 for a guided demonstration of how to include multiple FPGA designs within a single Simulink model.

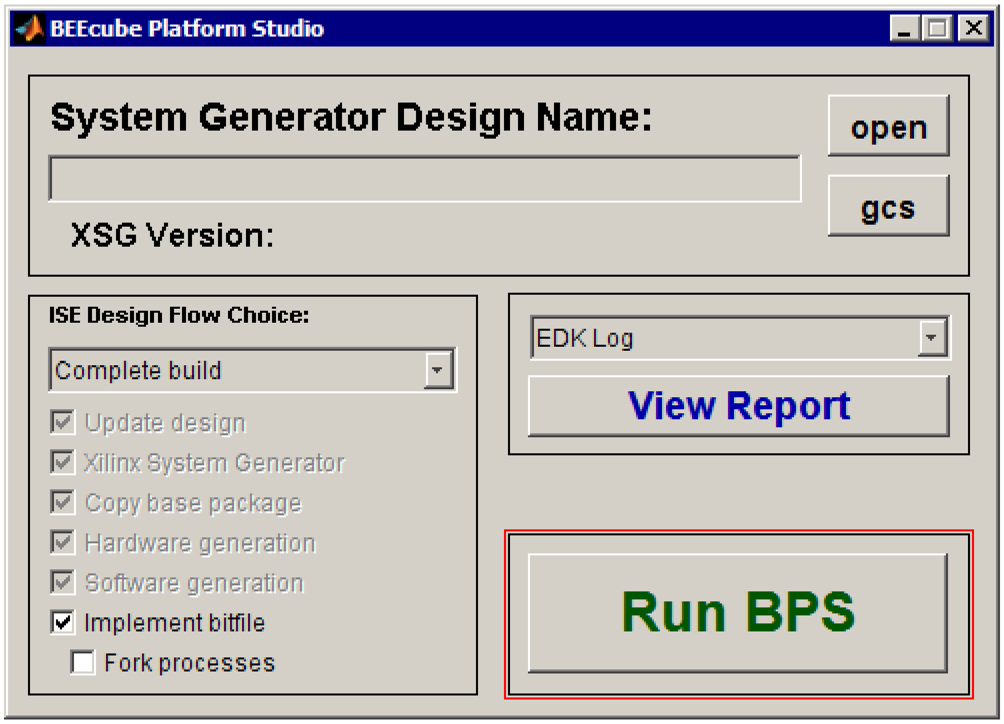
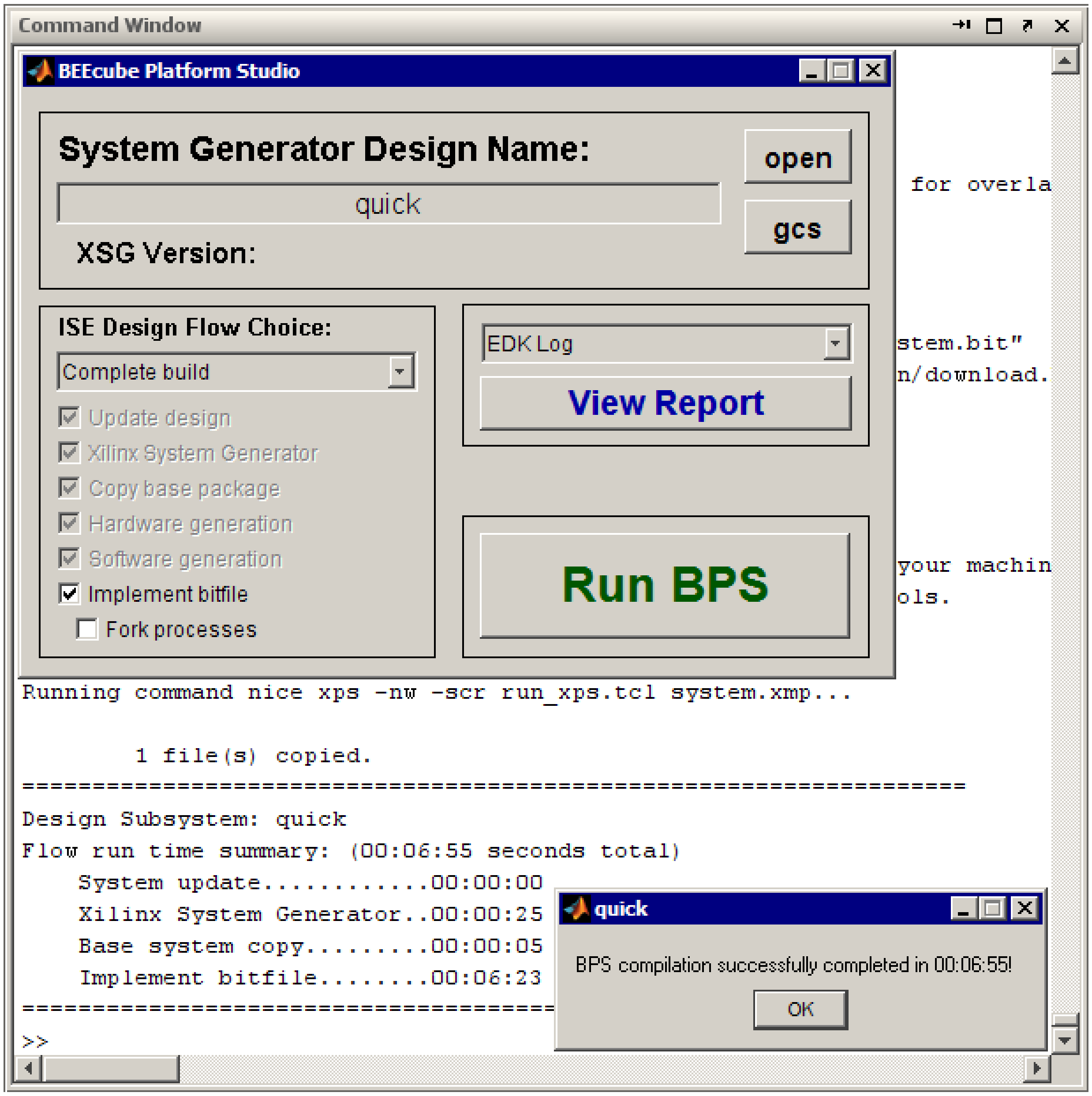


Figure 22: Run BPS panel

After a successful BPS compilation, an informational dialog pops up with the total compilation run time. The detailed step-by-step run time summary appears in the MATLAB command window. The feedback provided by BPS after a successful run is shown in Figure 23.



```
Running command nice xps -nw -scr run_xps.tcl system.xmp...

        1 file(s) copied.
===================================================================
Design Subsystem: quick
Flow run time summary: (00:06:55 seconds total)
    System update...........00:00:00
    Xilinx System Generator..00:00:25
    Base system copy.........00:00:05
    Implement bitfile........00:06:23
=====================================
>>
```

BPS compilation successfully completed in 00:06:55!

Figure 23: BPS feedback after successful run

When the **Implement bitfile** step is selected and **Fork processes** is unselected in the design flow choices list, BPS launches the back-end tool flow as part of the Matlab process. After the final bitfile is generated, it is copied into the `output` directory, which exists in parallel to the EDK build directory. This bitfile is named with the exact time of creation and name of the design to help identify the implementation at a later time. Second, for Virtex 6 systems and earlier, BPS parses the ISE backend tool output logs to check for any timing errors. When the final timing score is non-zero, a warning dialog pops up and automatically opens the timing report for the design.

When the **Implement bitfile** and **Fork processes** steps are both selected in the design flow choices list, BPS will launch the back-end tool flow in a separate console window. In this case, the BPS completion pop-up will appear immediately after the back-end tools have been launched and the Matlab console will return to the command prompt. The flow is the same as when **Fork processes** is not checked, but the process does not block Matlab. This method is convenient for running multiple implementation processes in parallel, or for running a single job through the back-end tools (such as to verify current timing results) while continuing to work in the Matlab environment.

When the **Implement bitfile** step is not selected, BPS will generate all necessary FPGA implementation files as a complete EDK or Vivado project, but will not launch the back-end tools to produce the final bitfile. This method is provided so that the back-end tools can be run on a different machine (such as a remote batch-processing cluster with the Xilinx ISE/EDK/Vivado tools installed, but which may not be able to run BPS).

For Virtex 6 and earlier, BPS automatically generates two scripts which can automatically run the back-end tools: `run_xps.cmd` for Windows hosts, and `run_xps.sh` for Linux hosts. These scripts are located in the EDK project directory, which is located underneath the BPS project directory and will have a name starting with `"XPS_"`. If it is necessary to copy the implementation files to a different directory accessible by the remote machine, be sure to copy the entire BPS project directory (the directory with a name that matches the model and subsystem being built), not only the EDK project directory.

For Virtex 7, BPS automatically generates a Tcl script which can Vivado can use to generate the bitfile: `run_vivado.tcl`. This script is located in the Vivado project directory, which is located underneath the BPS project directory and will be named `"vivado"`. If it is necessary to copy the implementation files to a different directory accessible by the remote machine, be sure to copy the entire BPS project directory (the directory with a name that matches the model and subsystem being built), not only the EDK project directory or the Vivado project directory.

Report Analysis

After BPS compilation, you can view the following reports directly from the BPS GUI by toggling the drop-down list above the **View Report** button:

- EDK log: opens `system.log` file created by XPS flow
- XFLOW log: opens `xflow.log` file created by ISE flow
- MAP report: opens `system_map.mrp` file generated during mapping
- Timing report: opens `system.twr` file generated during post-PAR timing analysis

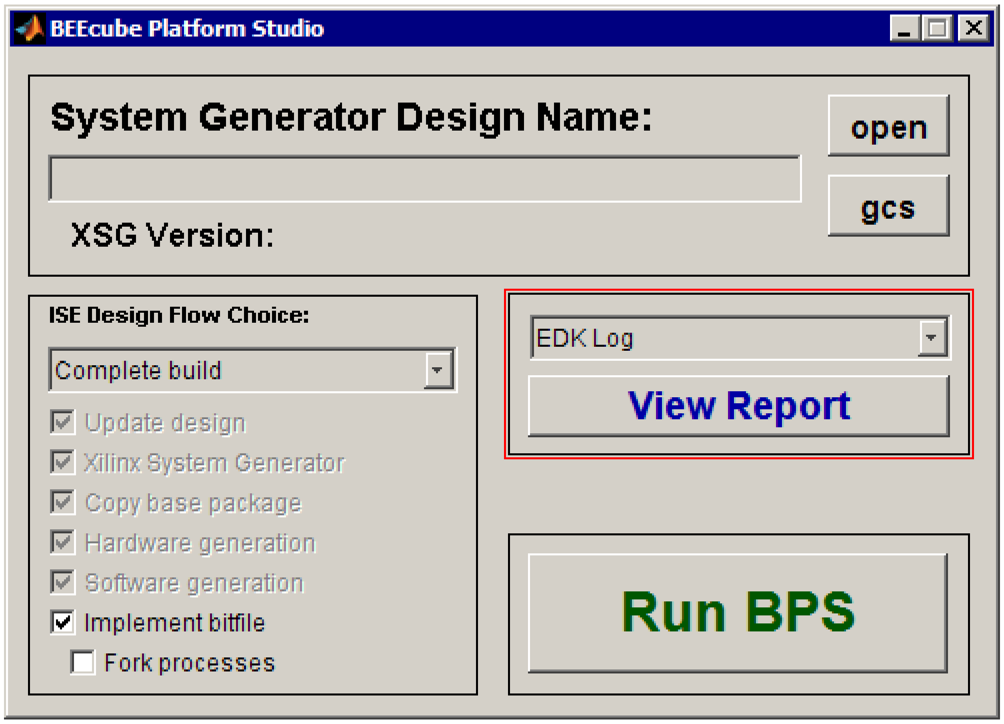


Figure 24: Report Analysis panel

After selecting the appropriate report, click **View Report** to view the file. All reports open in the Matlab built-in text editor. Note that these reports are only available for systems with Virtex 6 and earlier.

FPGA Implementation Output

The FPGA implementation generated by BPS is a standard Xilinx EDK project or Xilinx Vivado project, which can be built into a bitfile by the Xilinx Platform Studio (XPS) or Vivado tool. BPS also generates several scripts which can be used to manually launch XPS to build an FPGA bitfile, as well as scripts which can be used to physically program the FPGA. BPS will also preserve all intermediate output files produced by each intermediate phase of the implementation process, which can be useful for advanced users to inspect the detailed performance characteristics of the implementation.

By default, the **BPS project directory** is created in the same directory as the Simulink model itself, and will inherit its name from the Simulink path of the top-level subsystem being built. For example, if a Simulink model named *my_model* contains a top-level subsystem named *fpga1* which represents an FPGA design, the BPS project directory for the FPGA implementation will be called `my_model_fpga1`. Note that any non-alphanumeric characters in the Simulink model or subsystem path name will be converted into a single underscore character in the directory name.

Within the BPS project directory are several subdirectories used by BPS during the FPGA implementation process. These directories are automatically created the first time BPS is run on a given design, and are described in the following table.

Table 9: BPS project directory contents

Directory name	Description
`output`	Contains time-stamped subdirectories of each FPGA output file generated directly by BPS, including the programming file and ChipScope CDC/LTX files. More information on the BPS implementation GUI design flow options can be found above.
`repository`	Can be used to place custom EDK pcores to be included in the final system. This directory is typically used for adding custom IP components imported using the BPS IP Import Utility, which is further described in the Appendix.
`src`	Contains any custom source code to be stitched into the embedded software application by BPS. Whenever new source files have been added to this directory or new functions have been added to any existing source files, the BPS *Software generation* phase should be rerun.
`sysgen`	Used as the runtime output directory for Xilinx System Generator when launched by BPS. The contents of this directory should not be modified by the user, but can be used as a reference for any advanced output from XSG.
`vivado`	Contains the top level of the Xilinx Vivado project if the system is built for Virtex 7 and later. This is referred to as the **Vivado project directory**.
`XPS_${VER}_${HW}_${BP}`	Contains the top level of the Xilinx EDK project, and is referred to as the **EDK project directory**. The name of this directory will change based on the system-level parameters of the design. `$VER` will be replaced with the Xilinx tool versions being used, `$HW` will be replaced with the targeted hardware platform, and `$BP` will be replaced with the selected base package configuration.

The EDK project directory is generated by first extracting the contents of the appropriate BPS base package, and then extending the system with all the hardware and software components required by the BPS blocks used in the Simulink design. The general format of this directory is consistent with any Xilinx EDK project created for use with Xilinx Platform Studio (XPS). For more information on EDK, XPS, Vivado and all related FPGA tools, please refer to the corresponding product documentation from Xilinx.

In addition to the standard EDK source and configuration files, BPS also creates several scripts for launching back-end tools after the generation process. Each of these scripts is described in the table below.

Table 10: BPS additions to EDK project directory (for Virtex 6 and earlier systems only)

Script name	**Description**
`run_xps.sh`	BASH shell script which will launch XPS and build the final hardware/software FPGA implementation.
`run_xps.cmd`	Windows batch file which will launch XPS and build the final hardware/software FPGA implementation.
`util/prog/build_ace*.sh`	Sample BASH shell script which will generate an ACE programming file containing the FPGA implementation (on supported platforms). The resulting ACE file can be placed on a CompactFlash card to program the hardware platform via SystemACE.
`util/prog/jtag_prog.sh`	Sample BASH shell script which will program the FPGA via JTAG over a Xilinx programming cable.

These scripts are provided for convenience and are not guaranteed to work exactly as written across all possible host configurations. Some modifications may be required to match your host environment and/or target application.

Interacting with the Embedded Shell

The NectarOS embedded shell is included as the default software environment for all FPGA implementations. This section includes a description of all of the NectarOS commands which are created by BPS. In addition to this reference, the Tab key can be used at the prompt to list all available commands, or to list the arguments expected once a known command has been typed.

For all commands which take a numeric address or value as an argument, NectarOS recognizes both decimal and hexadecimal constants. Hexadecimal constants must begin with a lowercase *x*, such as `x400` to represent 1024.

In each of the descriptions below, arguments are enclosed in angle brackets `<>`, optional arguments are enclosed in square brackets `[]`, and argument formats are enclosed in parentheses `()`.

Note that in addition to the intrinsic commands listed below, many BPS blocks also provide their own runtime commands. These commands are only added to the shell if one or more of the corresponding blocks are present in the design. Information on all block-specific NectarOS commands are included as part of the BPS design component reference in Chapter 5.

Intrinsic NectarOS Commands

The commands listed here are always included with NectarOS, regardless of which types of cores are being used in a particular design.

`help`

Usage: `help`

The `help` command will print a list of helpful information (keystrokes, commands, etc.) for using NectarOS.

`read`

Usage: `read <start_address> [<end_address>]`

The `read` command will read data from any address or address range on the main processor PLB bus. If only a start address is given, one 32-bit word will be read at the address specified. If an end address is also given, the entire memory range between the addresses will be read and printed to the console.

`write`

Usage: `write <access_type (b|s|l)> <address> <value>`

The `write` command will write one value to a given memory location. The access type argument determines the size of the data operation, which can be either `b` (byte, or 8-bit), `s` (short, or 16-bit), or `l` (long, or 32-bit).

`readrtc` (BEE3/BEE4 only)

Usage: `readrtc`

The `readrtc` command will read all the registers in the attached Real-Time Clock (RTC) chip and print the results to the console. The output will include the status of all configuration registers, as well as the current date and time in the proper format.

`time` *(BEE3/BEE4 only)*

Usage: `time <time (HH:MM:SS)>`

The `time` command will set the current time of day in the RTC chip. The argument to this command must be in the format `HH:MM:SS`, where `HH` is the current hour in 24-hour format, and `MM` and `SS` are the minutes and seconds, respectively.

`date` *(BEE3/BEE4 only)*

Usage: `date <time (MM-DD-YYYY)>`

The `date` command will set the current date in the RTC chip. The argument to this command must be in the format `MM-DD-YYYY`, where `MM` is the month, `DD` is the day, and `YYYY` is the year.

`day` *(BEE3/BEE4 only)*

Usage: `day < day (Mo|Tu|We|Th|Fr|Sa|Su)>`

The `day` command will set the current day of the week in the RTC chip. The argument to this command must be a two-character string from the list: `Mo`, `Tu`, `We`, `Th`, `Fr`, `Sa`, `Su`.

`readprom (not supported on BEE7)`

Usage: `readprom <prom_address>`

The `readprom` command will read a single byte from the EEPROM chip at the PROM address specified. The byte value will be printed to the console.

`writeprom (not supported on BEE7)`

Usage: `writeprom <prom_address> <value>`

The `writeprom` command will write the given byte value to the user space of the EEPROM chip at the PROM address specified. On BEE3 platforms, the lower 128 bytes of the EEPROM are reserved for system use, and attempts to write to these addresses will be denied. On ML50x platforms, extreme care should be taken not to write into the factory-programmed address locations, as the factory PROM layout is expected to exist in other NectarOS routines.

`bee3_info` *(BEE3 only)*

Usage: `bee3_info`

The `bee3_info` command will print out information about the BEE3 system and FPGA on which the shell is running.

`bee4_info` *(BEE4 only)*

Usage: `bee4_info`

The `bee4_info` command will print out information about the BEE4 system and FPGA on which the shell is running.

`listdev`

Usage: `listdev`

The `listdev` command will print out a list of all the BPS-generated cores in the design and their corresponding base addresses.

jc_spi_write (BEE7 only)

Usage: jc_spi_write <data>

The jc_spi_write command lets you manually edit the registers of the jitter cleaner for BEE7. It is not recommended to use this command unless you are familiar with the registers inside the chip.

jc_macro_write (BEE7 only)

Usage: jc_macro_write <macro_num (1-6)>

The jc_macro_write command has several different macros for configuring the jitter cleaner. The different settings for each macro can be found in the base package under Software/jc_spi.h.

jc_set_input_freq_cmd (BEE7 only)

Usage: jc_set_input_freq_cmd <input freq (khz)> <pdf_val [min|max|freq in hz] (optional)> <cp_current [100|200|400|1600] (optional)>

The jc_set_input_freq_cmd configures the jitter cleaner based on the input frequency given in kHz. For additional configuration of the jitter cleaner you can set the pdf frequency and charge pump current as well.

jc_show_reg_vals_cmd (BEE7 only)

Usage: jc_show_reg_vals_cmd

The jc_show_reg_vals_cmd shows the current register settings of the jitter cleaner.

jc_enable_sma_cmd (BEE7 only)

Usage: jc_enable_sma_cmd <true/false>

The jc_enable_sma_cmd will enable or disable the jitter cleaner output to the SMA. By default it is off.

jc_set_output_freq_cmd (BEE7 only)

Usage: jc_set_output_freq_cmd <gth/sma> <output freq (khz)>

The jc_set_output_freq_cmd sets the jitter cleaner output frequency for the gth reference or for the SMA output.

jc_get_status_cmd (BEE7 only)

Usage: jc_get_status_cmd

The jc_get_status_cmd reports whether the jitter cleaner is locked or not.

Chapter 4: Additional Resources

Guided tutorials are provided with BPS for each supported hardware platform. These tutorials can be accessed in the Matlab Start menu by navigating to *Start > Blocksets > Beecube Platform Studio.*

Below is a list of additional resources that may be useful for using BPS with a particular hardware platform, including a description of the guided tutorials.

- BEE3
    - *BEE3 Hardware Platform: Getting Started* (GS100)
      GS100 provides a brief overview of the hardware platform and a brief guide to loading the provided reference design.

    - *BEE3 Hardware Platform: User Manual* (UM100)
      UM100 provides a detailed description of the hardware platform, including the platform's architecture and pin assignments.

    - *Getting Started with BPS on the BEE3 Hardware Platform* (GS200)
      GS200 provides a guided tutorial to introduce modeling in Simulink, building designs in BPS, interacting with the NectarOS shell, and communicating with the BEE3 from within Matlab.

- BEE4
    - *Getting Started with the BEE4 Hardware Platform* (GS101)
      GS101 provides a brief overview of the hardware platform, a brief guide to loading the reference design, and some basic information on using the BEE4 system (e.g. installation, powering the board, programming the FPGAs, etc.)

    - *BEE4 Hardware Platform User Manual* (UM103)
      UM103 provides a detailed description of the hardware platform, including the platform's architecture, clocking, and pin assignments. It also briefly describes the host side, such as how to use the provided shell commands and how to program the board.

    - *Getting Started with BPS on the BEE4 Hardware Platform* (GS202)
      GS202 provides a guided tutorial to introduce modeling in Simulink, building designs in BPS, interacting with the NectarOS shell, and communicating with the BEE4 from within Matlab.

- MiniBEE4
    - *Getting Started with the miniBEE4 Hardware Platform* (GS102)
      GS102 provides a brief overview of the hardware platform, a brief guide to loading the reference design, and some basic information on using the miniBEE4 system (e.g. installation, powering the board, programming the FPGA, etc.)

    - *miniBEE4 Hardware Platform User Manual* (UM105)
      UM103 provides a detailed description of the hardware platform, including the platform's architecture, clocking, and pin assignments. It also briefly describes the host side, such as how to use the provided shell commands and how to

program the board.

- ML50x/ML60x boards
  - Xilinx's respective user guides
    These provide information on the hardware platform.
    - ML505/ML506/ML507: UG347
    - ML605: UG534
  - *Getting Started with BPS on the Xilinx ML Hardware Platforms* (GS201)
    GS201 provides a guided tutorial to introduce modeling in Simulink, building designs in BPS, interacting with the NectarOS shell, and communicating with the ML platform from within Matlab.

Chapter 5: BPS Design Component Reference

This chapter contains a complete reference of all the design components available in BPS. For each component, a full description of the Simulink block is provided, including definitions of all user-customizable block parameters as well as port information. For blocks which feature platform-specific variants tailored to multiple hardware targets, separate subsections are included for each variation.

In addition to the Simulink block specification, a complete reference of all software features included with each block in the NectarOS embedded shell is also provided. Many blocks have their own NectarOS commands to assist the user with interacting with the underlying hardware resources at runtime. These commands are automatically compiled into the NectarOS application whenever an instance of the given block is present in an FPGA design.

Lastly, some design components also come with their own software driver as part of the EDK design implementation. For each component with an EDK software driver, an additional subsection is included which describes all data structures and functions present in the EDK driver.

10Gb Ethernet

The 10Gb Ethernet support included in BPS provides a custom XAUI-based PHY interface to a standard 10 Gigabit Ethernet network. On BEE3, this is implemented using the CX4 ports on the front panel of the system, without any need for an external PHY or transceiver when connecting to 10GBASE-CX4 networks. On BEE4, this is implemented using the QSFP ports, and on miniBEE4, this is implemented using the 10 GbE SFP+ ports.

A custom MAC is integrated into the hardware implementation which performs all necessary address filtering and frame validation functionality. An IP layer is also included as a combination of hardware and embedded software which supports a custom UDP-based protocol for transferring data at high speeds directly to/from shared memory components in the design, such as Software Registers, Shared BRAM, and DRAM. The IP layer automatically responds to ARP requests, manages an ARP cache, and will respond to ping (ICMP Echo) requests to assist with network integration. Both IP and MAC addresses are defined by the user via integrated NectarOS routines, described below.

For host-side programming of the data transfer protocol, a 64-bit Linux static library is included with the BPS distribution along with a header file for all supported routines. Both this library and several source code examples can be found in the *examples/src/host/linux/tengb_ethernet* directory of the BPS installation. Descriptions of the routines themselves are also included below.

10Gb Ethernet Interface block for BEE3, BEE4, and miniBEE4

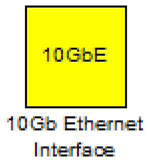
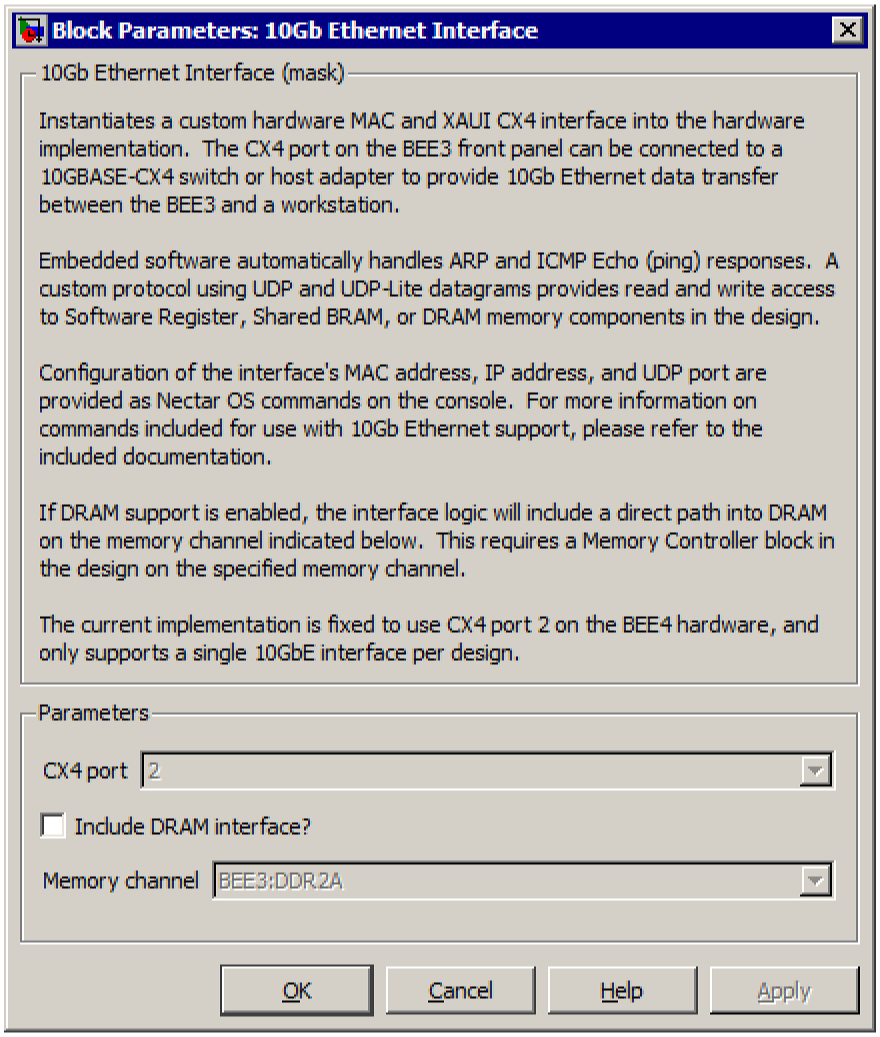


Figure 25: 10Gb Ethernet Interface block

Adding the 10Gb Ethernet Interface block to the top level of an FPGA design will instruct BPS to add full 10Gb Ethernet support to the hardware and software implementations. Because all data transactions are handled internally between the MAC/PHY interface and memory components, there are no ports or signals exposed to in the Simulink domain. Any Software Register, Shared BRAM, or DRAM instantiated in the system is made accessible to the 10Gb Ethernet data transfer protocol. However, on BEE4 and miniBEE4, DRAM support is currently unavailable via BPS.

The only configurable parameter for the 10Gb Ethernet Interface block is whether or not to include DRAM support. DRAM data transactions are handled specially via direct hardware translation on the FPGA, and result in the highest throughput (Software Register and Shared BRAM accesses are managed by the embedded microcontroller and exhibit slightly lower performance). If DRAM access via 10Gb Ethernet is desired in the system, the appropriate box should be checked, and the memory controller which the logic should be connected to must be selected from the drop-down box.



Block Parameters: 10Gb Ethernet Interface

10Gb Ethernet Interface (mask)

Instantiates a custom hardware MAC and XAUI CX4 interface into the hardware implementation. The CX4 port on the BEE3 front panel can be connected to a 10GBASE-CX4 switch or host adapter to provide 10Gb Ethernet data transfer between the BEE3 and a workstation.

Embedded software automatically handles ARP and ICMP Echo (ping) responses. A custom protocol using UDP and UDP-Lite datagrams provides read and write access to Software Register, Shared BRAM, or DRAM memory components in the design.

Configuration of the interface's MAC address, IP address, and UDP port are provided as Nectar OS commands on the console. For more information on commands included for use with 10Gb Ethernet support, please refer to the included documentation.

If DRAM support is enabled, the interface logic will include a direct path into DRAM on the memory channel indicated below. This requires a Memory Controller block in the design on the specified memory channel.

The current implementation is fixed to use CX4 port 2 on the BEE4 hardware, and only supports a single 10GbE interface per design.

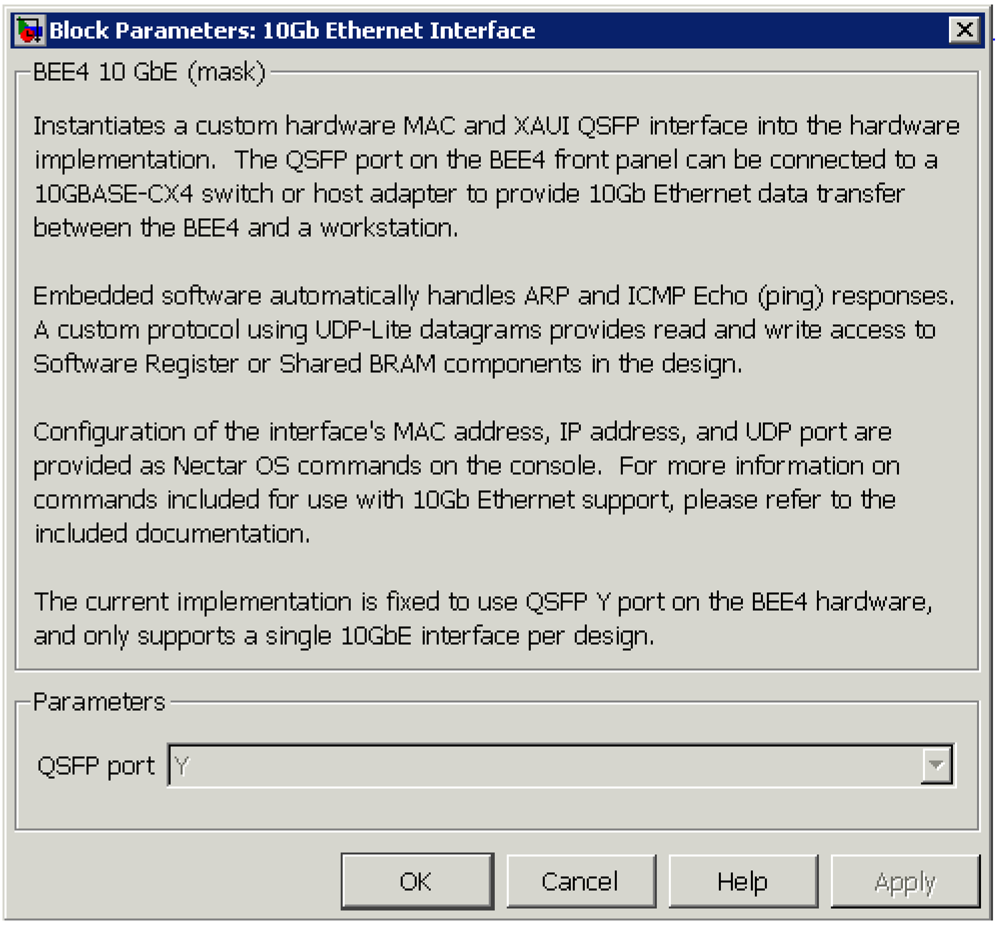
Parameters

CX4 port: 2

☐ Include DRAM interface?

Memory channel: BEE3:DDR2A

Figure 26: BEE3 10Gb Ethernet Interface dialog



Block Parameters: 10Gb Ethernet Interface

BEE4 10 GbE (mask)

Instantiates a custom hardware MAC and XAUI QSFP interface into the hardware implementation. The QSFP port on the BEE4 front panel can be connected to a 10GBASE-CX4 switch or host adapter to provide 10Gb Ethernet data transfer between the BEE4 and a workstation.

Embedded software automatically handles ARP and ICMP Echo (ping) responses. A custom protocol using UDP-Lite datagrams provides read and write access to Software Register or Shared BRAM components in the design.

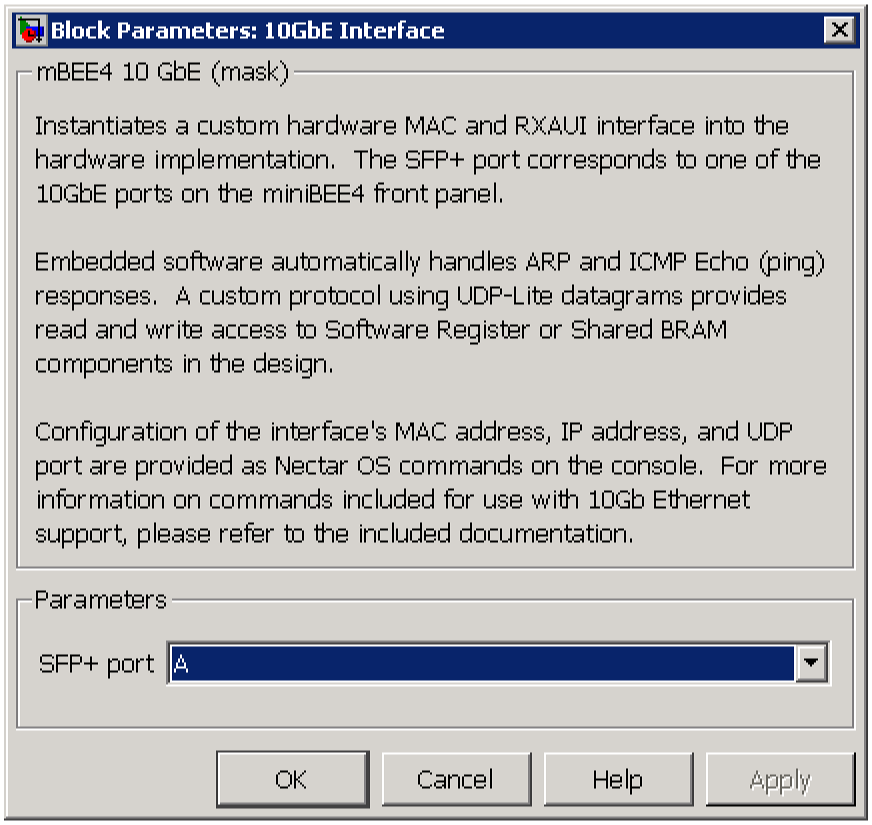
Configuration of the interface's MAC address, IP address, and UDP port are provided as Nectar OS commands on the console. For more information on commands included for use with 10Gb Ethernet support, please refer to the included documentation.

The current implementation is fixed to use QSFP Y port on the BEE4 hardware, and only supports a single 10GbE interface per design.

Parameters

QSFP port: Y

Figure 27: BEE4 10 Gb Ethernet Interface dialog


Block Parameters: 10GbE Interface

mBEE4 10 GbE (mask)

Instantiates a custom hardware MAC and RXAUI interface into the hardware implementation. The SFP+ port corresponds to one of the 10GbE ports on the miniBEE4 front panel.

Embedded software automatically handles ARP and ICMP Echo (ping) responses. A custom protocol using UDP-Lite datagrams provides read and write access to Software Register or Shared BRAM components in the design.

Configuration of the interface's MAC address, IP address, and UDP port are provided as Nectar OS commands on the console. For more information on commands included for use with 10Gb Ethernet support, please refer to the included documentation.

Parameters

SFP+ port: A

Figure 28: miniBEE4 10 Gb Ethernet Interface dialog

10Gb Ethernet NectarOS functions

BPS adds the following functions to NectarOS whenever the 10Gb Ethernet Interface block is used in a design. The source code for these functions are contained in the *drivers/xps_ten_gb_ethernet* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

```
xge_repeat
```

The `xge_repeat` function first checks to see if there are any unfinished software-based read sessions which require data to be sent and initiates transmission. Next, it checks for the presence of any new incoming software-based sessions that must be handled. Currently, three types of sessions must be handled in software: ARP (Request or Response), ICMP (Echo Request), and BPS memory component data requests.

Commands

```
xge_ifconfig <core> <ip (x.x.x.x)> <mac (000102030405)> <port>
```

The `xge_ifconfig` command configures the 10Gb Ethernet interface with the specified parameters. The `core` argument must be the name of the 10Gb Ethernet core in NectarOS (i.e. the name reported by the `listdev` command). The IP address should be specified in standard dot-decimal notation, and the MAC address should be specified as a string of hexadecimal characters without any colons or dashes. The `port` argument can be any user-defined UDP port number to be used as the destination port for all remote BPS data transfer sessions.

```
xge_arping <core> <target IP>
```

The `xge_arping` command works just like the standard Unix `arping` command by sending out an ARP request packet for the IP address given as the second argument. The ARP response data will not be reported on the console automatically. Rather, you may use the `xge_info` command with the `-v` switch to print out the internal ARP cache (see below), or display the ARP cache manually on the target IP machine to check for presence of the FPGA MAC/IP information.

```
xge_info <core> (-v)
```

The `xge_info` command will print out the currently defined IP, MAC, and UDP port information for the 10Gb Ethernet interface specified by the `core` argument. If the optional `-v` switch is added, then the complete ARP table will also be printed to the console. Note that any unknown entries in the ARP table are filled in by default with the FPGAs own MAC address.

10Gb Ethernet remote access protocol

The host-side interface to the 10Gb Ethernet data transfer protocol is provided as a set of library routines, one for each type of supported transaction. A combination of UDP and UDP-Lite datagrams are used for high performance (as compared to TCP) to create a custom session-based protocol that guarantees reliable data transfer. Data integrity is guaranteed by the MAC layer error correction, and flow control is managed by maintaining session state (unique IDs and sequential packet counts) inside the host and FPGA. If any packets had been lost when the transmitter closes a session, the session will be restarted with half the total size until reliable transmission is achieved.

The header (*include/xge.h)*, static library, and some source code examples for these data transactions can be found in the BPS installation under the *examples/src/host/linux/tengb_ethernet* subdirectory. Descriptions of each function and their usage are also included here for reference.

```
xgeStatus XGE_DRAMWrite(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint32_t len,
     int fd);
```

This routine writes `len` number of bytes from the open file referenced by `fd` into FPGA DRAM starting at base address `baseAddr`. Because the DRAM interface only writes in complete bursts for maximum efficiency, both the length and address must be 64-byte aligned. The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

```
xgeStatus XGE_DRAMRead(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint64_t size,
     int fd);
```

This routine reads `size` number of bytes from FPGA DRAM starting at base address `baseAddr` into the open file referenced by `fd`. Because the DRAM interface only writes in complete bursts for maximum efficiency, both the length and address must be 64-byte aligned. The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

```
xgeStatus XGE_BRAMWrite(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint32_t len,
     int fd);
```

This routine writes `len` number of bytes from the open file referenced by `fd` into the BRAM memory location referenced by the base address `baseAddr`. Because all Shared BRAM elements are fixed 32-bit values, both the length and address must be 4-byte aligned. It is the user's responsibility to make sure that the address is a valid byte address which corresponds to a BRAM memory location on the FPGA (this address may include an offset with the BRAM memory space as long as it remains 32-bit aligned). The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

```
xgeStatus XGE_BRAMRead(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint64_t size,
     int fd);
```

This routine reads `size` number of bytes from the BRAM memory location referenced by the base address `baseAddr` into the open file referenced by `fd`. Because all Shared BRAM elements are fixed 32-bit values, both the length and address must be 4-byte aligned. It is the user's responsibility to make sure that the address is a valid byte address which corresponds to a BRAM memory location on the FPGA (this address may include an offset with the BRAM memory space as long as it remains 32-bit aligned). The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

```
xgeStatus XGE_RegRead(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint32_t *regData);
```

This routine reads a 32-bit value from the Software Register referenced by the base address `baseAddr` into the 32-bit variable referenced by the pointer `regData`. The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

```
xgeStatus XGE_RegWrite(
     struct sockaddr_in *src_sockaddr,
     struct sockaddr_in *dst_sockaddr,
     uint64_t baseAddr,
     uint32_t reg_val);
```

This routine writes the 32-bit value `reg_val` into the Software Register referenced by the base address `baseAddr`. The `src_sockaddr` and `dst_sockaddr` pointers must reference standard Internet socket address structures with the family set to `AF_INET` and the source and destination IP addresses filled in. The destination address structure must also have the port property set to the UDP port number defined in the `xge_ifconfig` call on the FPGA. The return value is a status code as defined in the library header file.

Aurora

BPS supports the streaming mode of the Xilinx Aurora protocol for use with high-speed serial data I/O over the Virtex-5 GTP, Virtex-6 GTX, and Virtex-7 GTX or GTH transceivers. Separate blocks are provided in both the BPS BEE3 Blockset, the BPS BEE4 Blockset, the BPS miniBEE4 Blockset, the BPS ML50x Blockset, the BPS VC707 Blockset, and the BPS BEE7 Blockset as the set of configurable parameters for each target platform are slightly different. The BPS Aurora implementation also includes NectarOS functions for initialization and dynamic configuration of the hardware, as well as an EDK software driver for use by custom software applications.

Aurora block for BEE3

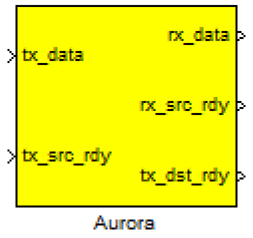
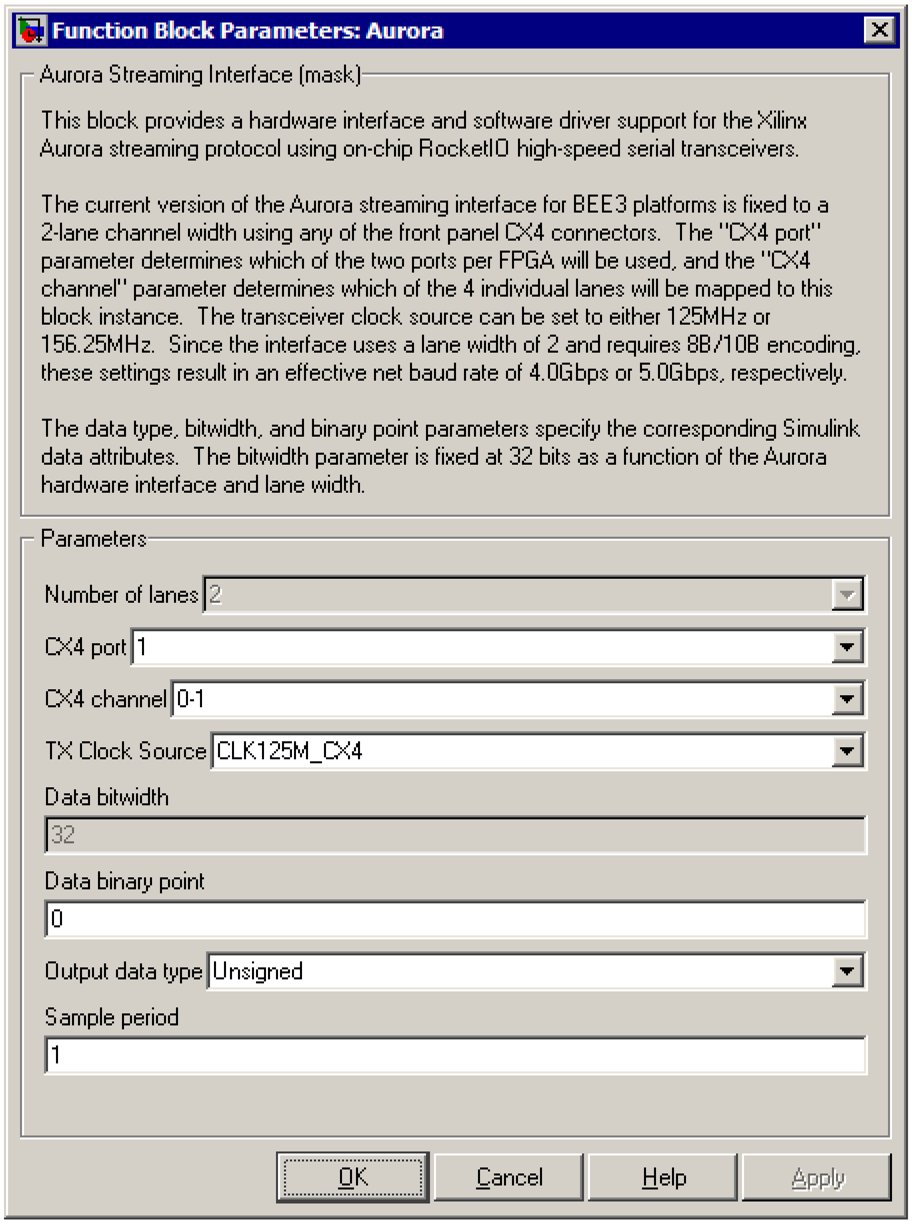


Figure 29: BEE3 Aurora block



Aurora Streaming Interface (mask)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip RocketIO high-speed serial transceivers.

The current version of the Aurora streaming interface for BEE3 platforms is fixed to a 2-lane channel width using any of the front panel CX4 connectors. The "CX4 port" parameter determines which of the two ports per FPGA will be used, and the "CX4 channel" parameter determines which of the 4 individual lanes will be mapped to this block instance. The transceiver clock source can be set to either 125MHz or 156.25MHz. Since the interface uses a lane width of 2 and requires 8B/10B encoding, these settings result in an effective net baud rate of 4.0Gbps or 5.0Gbps, respectively.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is fixed at 32 bits as a function of the Aurora hardware interface and lane width.

Parameters

Number of lanes: 2

CX4 port: 1

CX4 channel: 0-1

TX Clock Source: CLK125M_CX4

Data bitwidth: 32

Data binary point: 0

Output data type: Unsigned

Sample period: 1

Figure 30: BEE3 Aurora dialog

The Aurora block for the BEE3 system uses the system FPGAs' gigabit transceivers to implement a streaming serial duplex I/O link between devices. The Aurora streaming link features clock correction to adjust for minor differences in frequency between the recovered remote clock and the local clock. Transmit and receive side data FIFOs allow the Aurora link and the system core logic to be in different clock domains.

The FPGA Aurora transceivers connect via the CX4 connectors on the front panel of the BEE3 system. Each FPGA in the BEE3 system is connected to two CX4 connectors, named Port 1 and Port 2. Each CX4 connector port can support two Aurora channels. One Aurora channel consists of two lanes, and each Aurora channel (pair of Aurora lanes) may be assigned to a pair of CX4 channels; either 0 – 1, or 2 – 3. Thus a single FPGA can support up to 4 Aurora links.

CX4 cables are used to make the physical connection between the FPGAs on one or more BEE3 systems. A CX4 to SATA breakout cable is used when connecting a BEE3 system to a Xilinx ML50x evaluation platform.

The system side data input (`tx_data`) and output (`rx_data`) are 32-bit wide buses. The single bit wide control input `tx_src_ready` and control outputs `tx_dst_rdy` and `rx_src_rdy` are provided for data handshaking. Data is sent as one continuous frame. There are no means for the system logic on the receive data side to back-pressure the Aurora link.

You may select between a 125 MHz or a 156.25 MHz TX clock for a 4.0 Gbps or a 5.0 Gbps line rate after 8B/10B conversion, respectively. Both ends of the Aurora link must be set to the same nominal TX clock frequency. Note that only the 125 MHz TX clock frequency is common to both the BEE3 system and the ML50x evaluation platforms.

Table 11: Aurora block parameters

Parameter	**Description**
Number of lanes	This parameter defines the number of physical signals (lanes) per Aurora channel. Fixed at 2.
CX4 port	This parameter assigns the Aurora channel to either the top (*1*) or the bottom (*2*) BEE3 front panel CX4 connector wired to the FPGA containing the Aurora link.
CX4 channel	This parameter selects which pair of CX4 channels (*0-1* or *2-3*) is used for the Aurora link on the CX4 port.
TX Clock Source	This parameter chooses either the 125 MHz (*CLK125M_CX4*) or the 156.25 MHz (*CLK156M_CX4*) TX reference clock source.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 32.
Data binary point	This value determines the inferred binary point position for both the data buses.
Data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 12: BEE3 Aurora port definitions

Port	Direction	Bit Width	Description
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.

Aurora block for BEE4 and miniBEE4

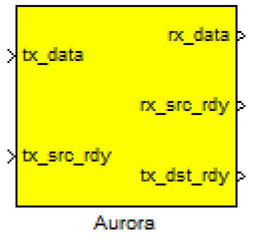


Figure 31: BEE4 Aurora block

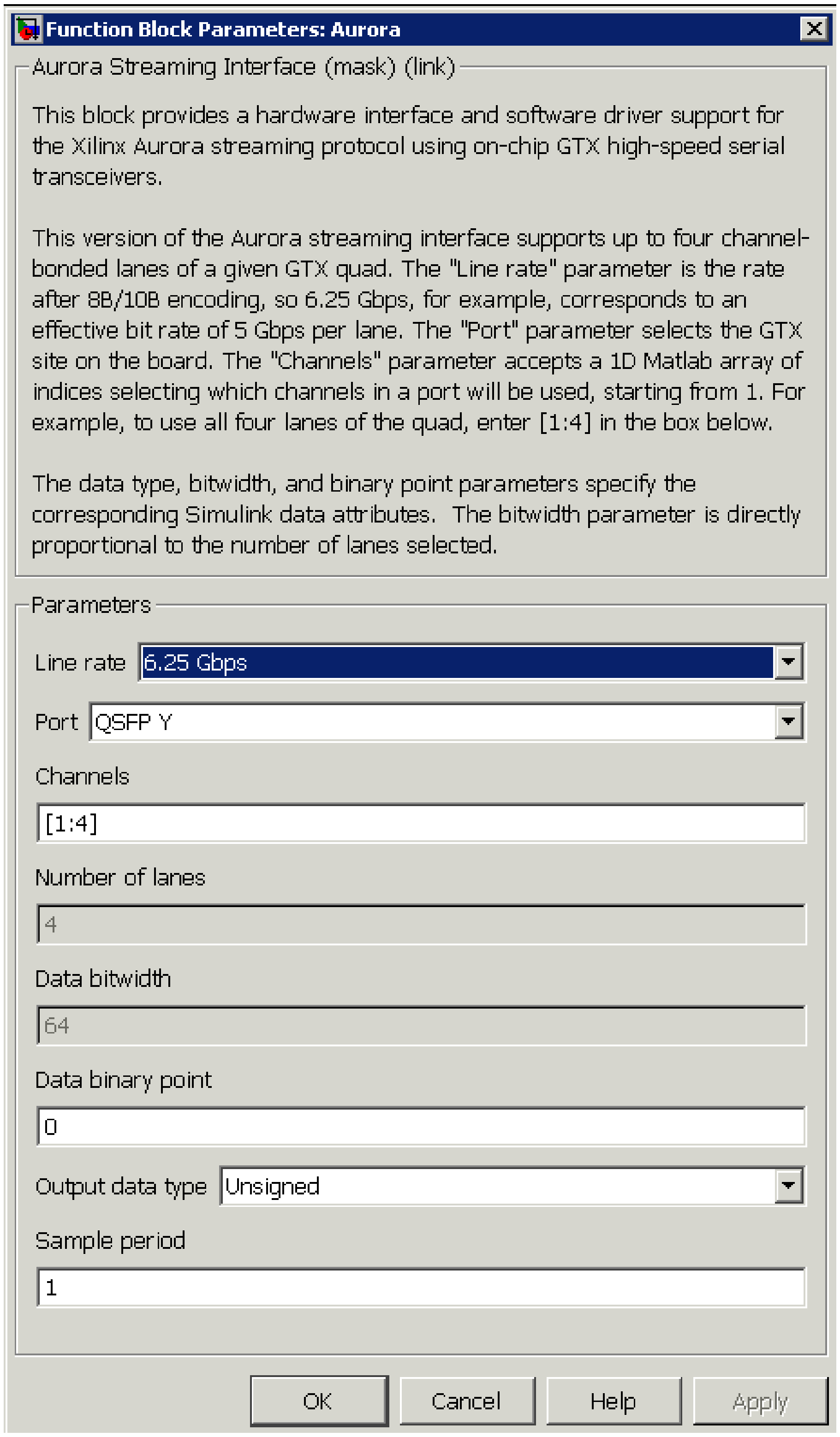
The Aurora block for the BEE4 system uses the system FPGAs' gigabit transceivers to implement a streaming serial duplex I/O link between devices. The Aurora streaming link features clock correction to adjust for minor differences in frequency between the remote clock and the local clock. Transmit and receive side data FIFOs allow the Aurora link and the system core logic to be in different clock domains.

For the BEE4, the FPGA Aurora transceivers can be used on several different interfaces, including the QSFP X and Y ports, FMC SerDes lanes, and inter-FPGA SerDes lanes (if available on the FPGA). On the miniBEE4, Aurora may be used with the SFP+ ports on the front panel. One Aurora channel consists of up to four bonded lanes, and each channel may be assigned to any of the lanes within the selected interface.

The system side data input (`tx_data`) and output (`rx_data`) bus width varies with the number of selected lanes. Each lane is 16 bits wide; thus, the bus width may be up to 64 bits wide. The single bit wide control input `tx_src_ready` and control outputs `tx_dst_rdy` and `rx_src_rdy` are provided for data handshaking. Data is sent as one continuous frame. There is no means for the system logic on the receive data side to apply back-pressure to the Aurora link.

The line rate is also configurable. The available options are 2.5 GT/s, 3.125 GT/s, 5.0 GT/s, or 6.25 GT/s. After 8B/10B conversion, the effective bit rates are 2.0 Gbps, 2.5 Gbps, 4.0 Gbps, and 5.0 Gbps, respectively. If an Aurora instance is made of 4 channel-bonded lanes at 6.25 GT/s, the maximum data rate is therefore 20 Gbps.

Note that the QSFP ports may require optical QSFP cables to operate at maximum speed. In addition, the EDK API is not supported on the BEE4.



Function Block Parameters: Aurora

Aurora Streaming Interface (mask) (link)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip GTX high-speed serial transceivers.

This version of the Aurora streaming interface supports up to four channel-bonded lanes of a given GTX quad. The "Line rate" parameter is the rate after 8B/10B encoding, so 6.25 Gbps, for example, corresponds to an effective bit rate of 5 Gbps per lane. The "Port" parameter selects the GTX site on the board. The "Channels" parameter accepts a 1D Matlab array of indices selecting which channels in a port will be used, starting from 1. For example, to use all four lanes of the quad, enter [1:4] in the box below.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is directly proportional to the number of lanes selected.

Parameters

Line rate: 6.25 Gbps

Port: QSFP Y

Channels: [1:4]

Number of lanes: 4

Data bitwidth: 64

Data binary point: 0

Output data type: Unsigned

Sample period: 1

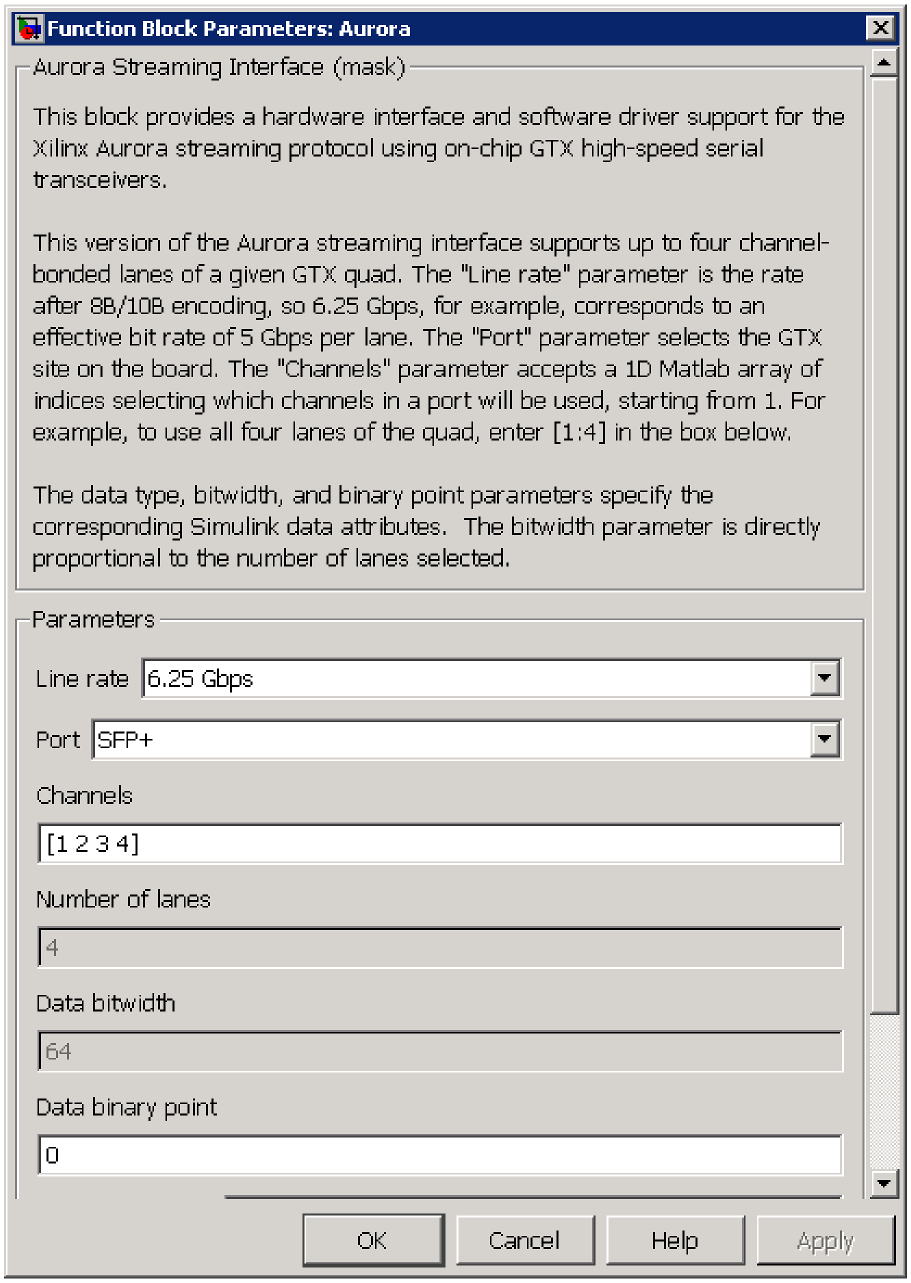
Figure 32: BEE4 Aurora dialog

Parameter	Description
Line rate	This parameter defines the line rate for the Aurora instance for each lane. Note that Aurora is 8b10b encoded.
Port	This parameter assigns the Aurora instance to an interface on the BEE4. This can be one of *QSFP X*, *QSFP Y*, *FMC 0-3*, *FMC 4-7*, *DIAG*, *RING UP*, or *RING DOWN*.
Lanes	This parameter defines the mapping of Aurora lanes to physical pins. Expects a Matlab vector of indices from the set [1:4].
Number of lanes	This parameter reports the number of lanes composing the Aurora channel. Derived from the Lanes parameter.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 16 * Number of lanes.
Data binary point	This value determines the inferred binary point position for both the data buses.
Output data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Figure 33: BEE4 Aurora block parameters

Port	Direction	Bit Width	Description
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.

Figure 34: BEE4 Aurora port definitions


Function Block Parameters: Aurora

Aurora Streaming Interface (mask)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip GTX high-speed serial transceivers.

This version of the Aurora streaming interface supports up to four channel-bonded lanes of a given GTX quad. The "Line rate" parameter is the rate after 8B/10B encoding, so 6.25 Gbps, for example, corresponds to an effective bit rate of 5 Gbps per lane. The "Port" parameter selects the GTX site on the board. The "Channels" parameter accepts a 1D Matlab array of indices selecting which channels in a port will be used, starting from 1. For example, to use all four lanes of the quad, enter [1:4] in the box below.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is directly proportional to the number of lanes selected.

Parameters

Line rate: 6.25 Gbps

Port: SFP+

Channels: [1 2 3 4]

Number of lanes: 4

Data bitwidth: 64

Data binary point: 0

Figure 35: miniBEE4 Aurora dialog

Parameter	**Description**
Line rate	This parameter defines the line rate for the Aurora instance for each lane. Note that Aurora is 8b10b encoded.
Port	This parameter assigns the Aurora instance to an interface on the miniBEE4. Fixed to the SFP+ ports.
Lanes	This parameter defines the mapping of Aurora lanes to physical pins. Expects a Matlab vector of indices from the set

	[1:4].
Number of lanes	This parameter reports the number of lanes composing the Aurora channel. Derived from the Lanes parameter.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 16 * Number of lanes.
Data binary point	This value determines the inferred binary point position for both the data buses.
Output data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Figure 36: miniBEE4 Aurora block parameters

Port	**Direction**	**Bit Width**	**Description**
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.

Figure 37: miniBEE4 Aurora port definitions

Aurora block for ML50x

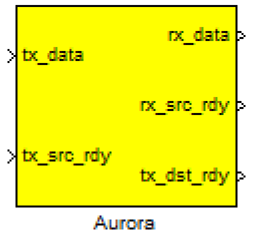
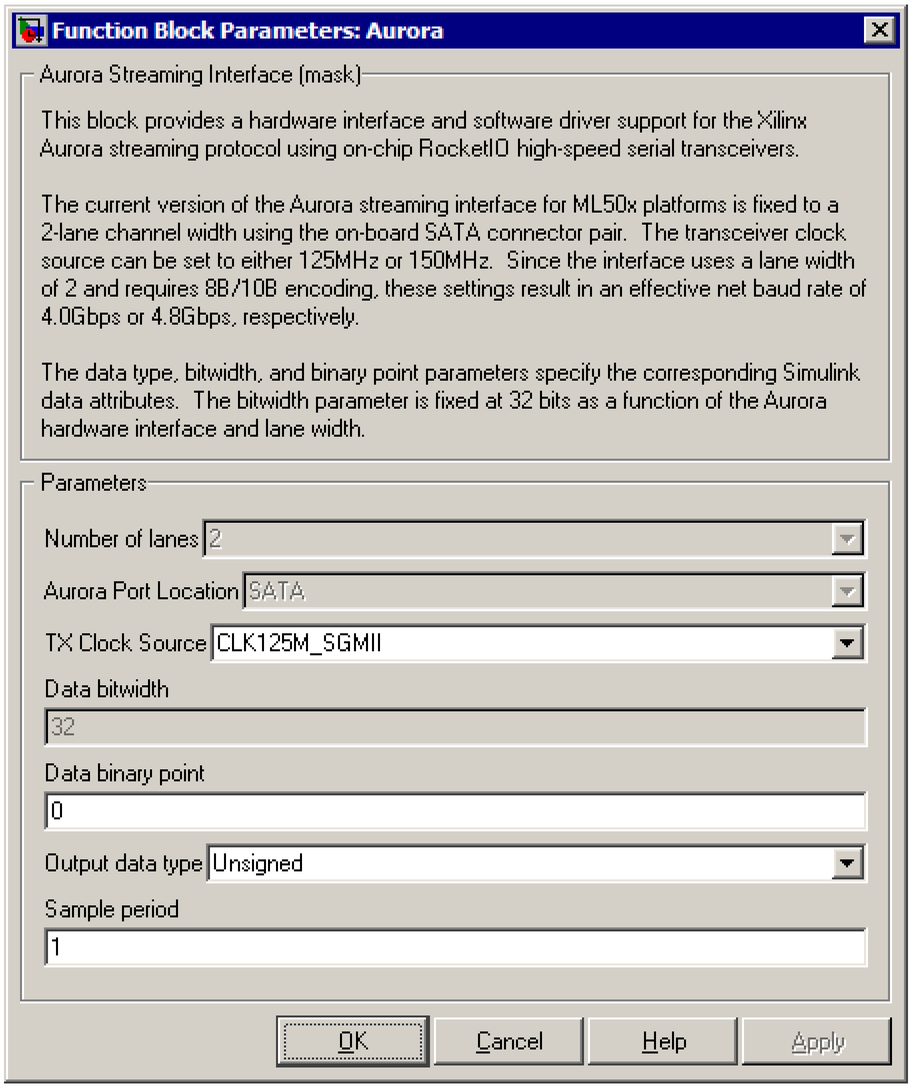


Figure 38: ML50x Aurora block


Function Block Parameters: Aurora

Aurora Streaming Interface (mask)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip RocketIO high-speed serial transceivers.

The current version of the Aurora streaming interface for ML50x platforms is fixed to a 2-lane channel width using the on-board SATA connector pair. The transceiver clock source can be set to either 125MHz or 150MHz. Since the interface uses a lane width of 2 and requires 8B/10B encoding, these settings result in an effective net baud rate of 4.0Gbps or 4.8Gbps, respectively.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is fixed at 32 bits as a function of the Aurora hardware interface and lane width.

Parameters

Number of lanes: 2

Aurora Port Location: SATA

TX Clock Source: CLK125M_SGMII

Data bitwidth: 32

Data binary point: 0

Output data type: Unsigned

Sample period: 1

Figure 39: ML50x Aurora dialog

The Aurora block for the Xilinx ML50x evaluation platform uses the FPGAs' gigabit transceivers to implement a streaming serial duplex I/O link between devices. The Aurora streaming link features clock correction to adjust for minor differences in frequency between the recovered remote clock and the local clock. Transmit and receive side data FIFOs allow the Aurora link and the system core logic to be in different clock domains.

The FPGA Aurora transceivers connect via the two SATA connectors on the board. Both connectors are required for the Aurora link. Crossed pair (TX to RX) SATA cables are used to make the physical connection between Xilinx ML50x evaluation platform boards. A CX4 to SATA breakout cable is used when connecting the Xilinx ML50x evaluation platform to a BEE3 system.

The system side data input (`tx_data`) and output (`rx_data`) are 32-bit wide buses. The single bit wide control input `tx_src_ready` and control outputs `tx_dst_rdy` and `rx_src_rdy` are provided for data handshaking. Data is sent as one continuous frame. There are no means for the system logic on the receive data side to back-pressure the Aurora link.

You may select between a 125 MHz or a 150 MHz TX clock for a 4.0 Gbps or a 4.8 Gbps line rate after 8B/10B conversion, respectively. Both ends of the Aurora link must be set to the same nominal TX clock frequency. Note that only the 125 MHz TX clock frequency is common to both ML50x evaluation platforms and BEE3 systems.

Note that the ML50x Aurora block is currently not supported on the Xilinx ML507 evaluation platform because its FPGA uses different type of Gigabit Transceiver than the kind used by the other ML50x boards and BEE3 system.

Table 13: ML50x Aurora block parameters

Parameter	**Description**
Number of lanes	This parameter defines the number of physical signals (lanes) per Aurora channel. Fixed at 2.
Aurora Port Location	This parameter assigns the Aurora channel to connectors on the ML50x board. The Aurora link is fixed on SATA ports.
TX Clock Source	This parameter chooses either the 125 MHz (*CLK125M_SGMII*) or the 150 MHz (*CLK150M_SATA*) TX reference clock source.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 32.
Data binary point	This value determines the inferred binary point position for both the data buses.
Data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 14: ML50x Aurora port definitions

Port	Direction	Bit Width	Description
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.

Aurora block for BEE7 and VC707

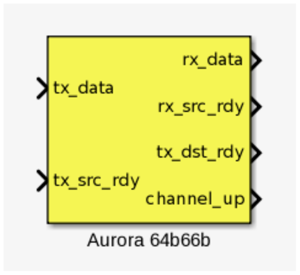


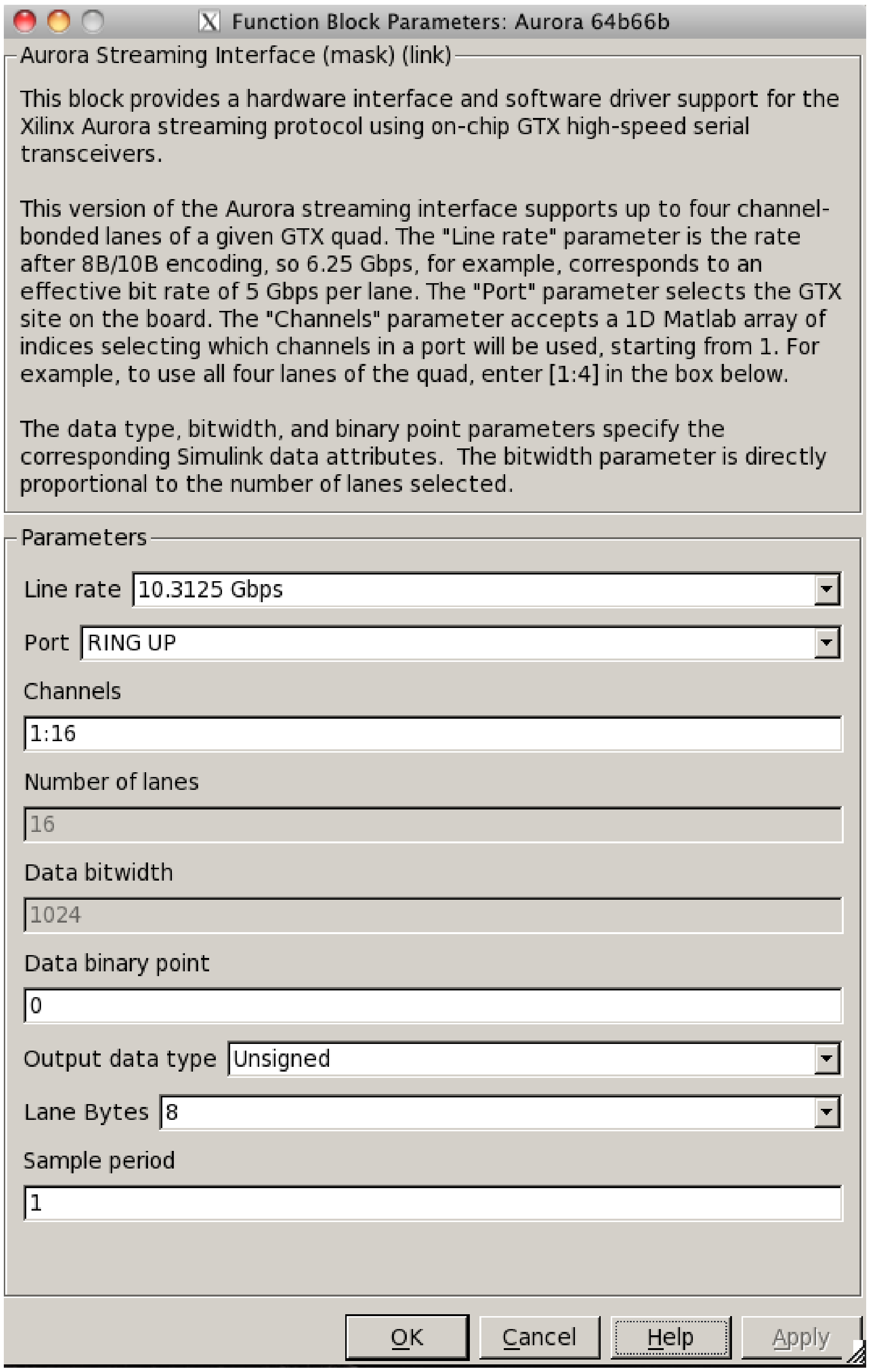
Figure 40: Aurora block

The Aurora block for the BEE7 system uses the system FPGAs' gigabit transceivers to implement a streaming serial duplex I/O link between devices. The Aurora streaming link features clock correction to adjust for minor differences in frequency between the remote clock and the local clock. Transmit and receive side data FIFOs allow the Aurora link and the system core logic to be in different clock domains.

For the BEE7, the FPGA Aurora transceivers can be used on several different interfaces, including the RTM SFP+ or QSFP, FMC SerDes lanes, Front IMOT (Integrated Multi-gigabit Optical Transceiver) and inter-FPGA SerDes lanes. On the VC707, Aurora may be used with the SFP+ port. One Aurora channel consists of up to four bonded lanes, and each channel may be assigned to any of the lanes within the selected interface.

The system side data input (`tx_data`) and output (`rx_data`) bus width varies with the number of selected lanes. Each lane is either 32 bits wide for the 2 to 1 FIFO mode or 64 bits wide. The single bit wide control input `tx_src_ready` and control outputs `tx_dst_rdy` and `rx_src_rdy` are provided for data handshaking. Data is sent as one continuous frame. There is no means for the system logic on the receive data side to apply back-pressure to the Aurora link. `Channel_up is a single bit output denoting the channel status.`

The line rate for the 64B/66B Aurora core is 10.3125Gbps. If an Aurora instance is made of 4 channel-bonded lanes, the maximum data rate is therefore 41.25 Gbps.



Aurora Streaming Interface (mask) (link)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip GTX high-speed serial transceivers.

This version of the Aurora streaming interface supports up to four channel-bonded lanes of a given GTX quad. The "Line rate" parameter is the rate after 8B/10B encoding, so 6.25 Gbps, for example, corresponds to an effective bit rate of 5 Gbps per lane. The "Port" parameter selects the GTX site on the board. The "Channels" parameter accepts a 1D Matlab array of indices selecting which channels in a port will be used, starting from 1. For example, to use all four lanes of the quad, enter [1:4] in the box below.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is directly proportional to the number of lanes selected.

Parameters

Line rate: 10.3125 Gbps

Port: RING UP

Channels: 1:16

Number of lanes: 16

Data bitwidth: 1024

Data binary point: 0

Output data type: Unsigned

Lane Bytes: 8

Sample period: 1

Figure 41: BEE7 Aurora dialog

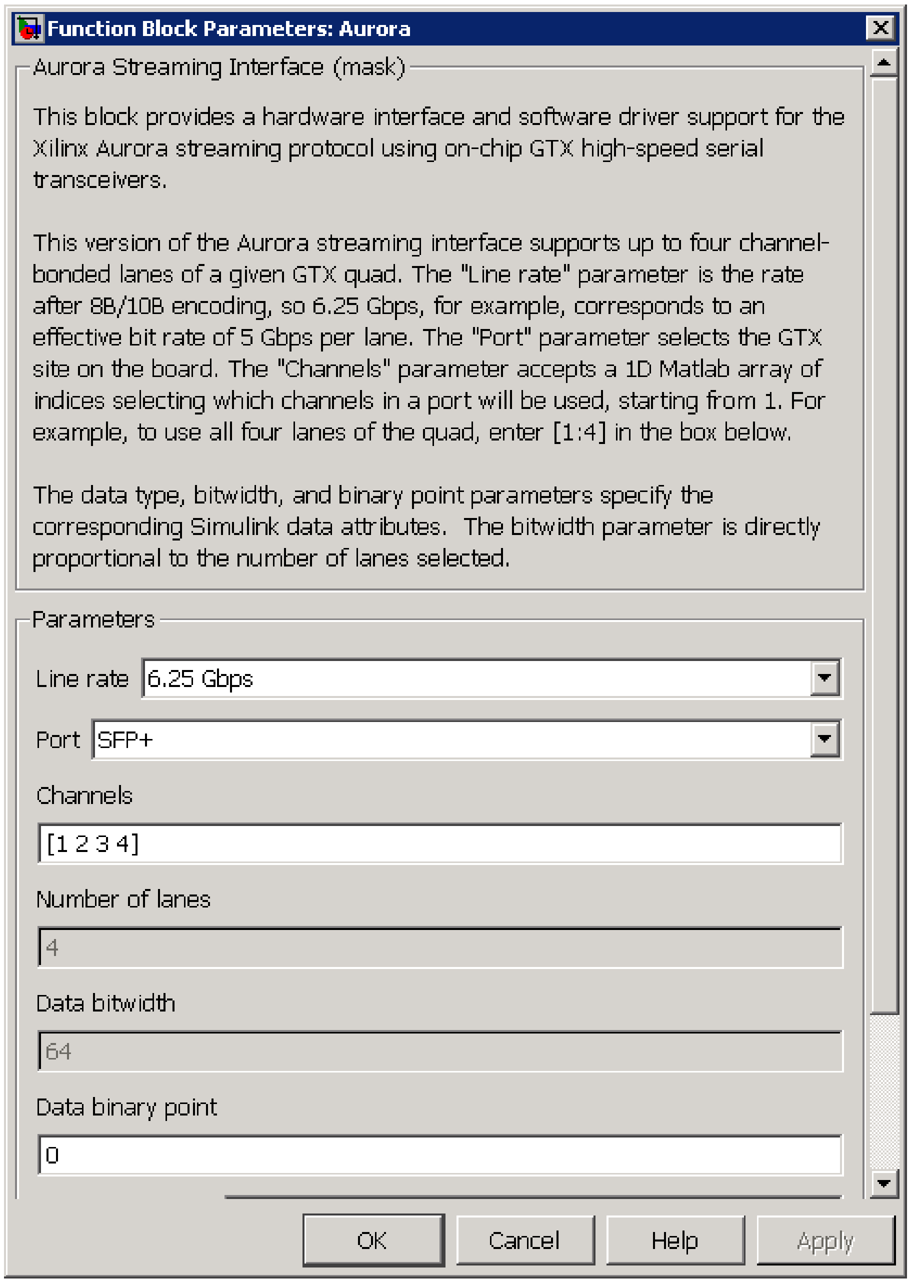
Parameter	**Description**
Line rate	This parameter defines the line rate for the Aurora instance for each lane. Note that this Aurora is 64b66b encoded.

Port	This parameter assigns the Aurora instance to an interface on the BEE7. This can be one of *DIAG, FMC, FRONT, RING UP, RING DOWN,* or *RTM*.
Channels	This parameter defines the mapping of Aurora lanes to physical pins. Expects a Matlab vector of indices from the set [1:16]. Maximum utilization per port is *DIAG* 1:8*, FMC* 1:8*, FRONT* 1:12*, RING UP* 1:16*, RING DOWN* 1:16*,* or *RTM* 1:9 or 1:16 depending on RTM40 or RTM64.
Number of lanes	This parameter reports the number of lanes composing the Aurora channel. Derived from the Channels parameter.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 32* Number of lanes if the 2 to 1 FIFO is utilized, otherwise it is 64 * Number of lanes.
Data binary point	This value determines the inferred binary point position for both the data buses.
Output data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Lane Bytes	This pulldown determines the 2 to 1 FIFO option. 4 bytes per lane instantiates the 2 to 1 FIFO, 8 bytes per lane incorporates a full rate FIFO.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Figure 42: BEE7 Aurora block parameters

Port	**Direction**	**Bit Width**	**Description**
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.
`channel_up`	Out	1	The `channel_up` port goes high when all the Aurora lanes and Aurora channel go up.

Figure 43: BEE7 Aurora port definitions


Function Block Parameters: Aurora

Aurora Streaming Interface (mask)

This block provides a hardware interface and software driver support for the Xilinx Aurora streaming protocol using on-chip GTX high-speed serial transceivers.

This version of the Aurora streaming interface supports up to four channel-bonded lanes of a given GTX quad. The "Line rate" parameter is the rate after 8B/10B encoding, so 6.25 Gbps, for example, corresponds to an effective bit rate of 5 Gbps per lane. The "Port" parameter selects the GTX site on the board. The "Channels" parameter accepts a 1D Matlab array of indices selecting which channels in a port will be used, starting from 1. For example, to use all four lanes of the quad, enter [1:4] in the box below.

The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The bitwidth parameter is directly proportional to the number of lanes selected.

Parameters

Line rate 6.25 Gbps

Port SFP+

Channels

[1 2 3 4]

Number of lanes

4

Data bitwidth

64

Data binary point

0

Figure 44: VC707 Aurora dialog

Parameter	Description
Line rate	This parameter defines the line rate for the Aurora instance for each lane. Note that this Aurora is 64b66b encoded.
Port	This parameter assigns the Aurora instance to an interface on the VC707. This can be *SFP*.
Channels	This parameter defines the mapping of Aurora lanes to physical pins. Expects a Matlab vector of indices from the set [1]. Maximum utilization per port is *SFP* 1.
Number of lanes	This parameter reports the number of lanes composing the Aurora channel. Derived from the Channels parameter.
Data bitwidth	This value defines the total width of the `tx_data` and `rx_data` buses in bits. Fixed at 32* Number of lanes if the 2 to 1 FIFO is utilized, otherwise it is 64 * Number of lanes.
Data binary point	This value determines the inferred binary point position for both the data buses.
Output data type	This parameter determines the inferred arithmetic type for both the data buses. This can be either *Unsigned*, or *Signed (2's comp)*.
Lane Bytes	This pulldown determines the 2 to 1 FIFO option. 4 bytes per lane instantiates the 2 to 1 FIFO, 8 bytes per lane incorporates a full rate FIFO.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Figure 45: VC707 Aurora block parameters

Port	Direction	Bit Width	Description
`tx_data`	in	32	The `tx_data` port is the data to be transmitted by the Aurora link to the remote FPGA.
`rx_data`	out	32	The `rx_data` port is the data that is received by the Aurora link from the remote FPGA.
`tx_src_rdy`	in	1	The `tx_src_rdy` port must be driven high to indicate the data on the `tx_data` port is valid.
`rx_src_rdy`	out	1	The `rx_src_rdy` port goes high when the data on the `rx_data` port is valid.
`tx_dst_rdy`	out	1	The `tx_dst_rdy` port goes high when the Aurora link is initialized and the `tx_data` port is ready to receive data.
`channel_up`	Out	1	The `channel_up` port goes high when all the Aurora lanes and Aurora channel go up.

Figure 46: VC707 Aurora port definitions

Aurora block NectarOS functions (BEE3, ML50x)

The Aurora block includes a set of utility functions that enable the user to control the Aurora link directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever an Aurora block is used in a design. The source code for these functions are contained in the *drivers/xps_aurora_streaming* subdirectory of the BPS-generated EDK project directory. The NectarOS functions and commands are the same for both the BEE3 and ML50x implementations of the Aurora block.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
aurora_reset <aurora instance name>
```

The `aurora_reset` function takes the Aurora block instance name string as an argument. It clears the state of the Aurora link and makes it ready to stream data. Under most circumstances issuing an `aurora_reset` is all that is required to initialize the Aurora block. Other initialization functions tune the Aurora link PHY to remedy signal integrity issues that may be encountered in some special circumstances:

```
aurora_diffctrl <aurora instance name> <value>
```

The `aurora_diffctrl` function takes the Aurora block instance name string and an integer value as arguments. It adjusts the TX differential voltage swing of the Aurora PHY.

Table 15: `aurora_diffctrl` values

Value	Description
0	1100 mV pp swing
1	1050 mV pp swing
2	1000 mV pp swing
3	900 mV pp swing
4	800 mV pp swing
5	600 mV pp swing
6	400 mV pp swing
7	0 mV pp swing

`aurora_preemp <aurora instance name> <value>`

The `aurora_preemp` function takes the Aurora block instance name string and an integer value as arguments. It adjusts the TX pre-emphasis percentage of the Aurora PHY.

Table 16: Valid values for `aurora_preemp`

Value	Description
0	2% TX pre-emphasis
1	3% TX pre-emphasis
2	4% TX pre-emphasis
3	10.5% TX pre-emphasis
4	18.5% TX pre-emphasis
5	28% TX pre-emphasis
6	39% TX pre-emphasis
7	52% TX pre-emphasis

`aurora_eq_mix <aurora instance name> <value>`

The `aurora_eq_mix` function takes the Aurora block instance name string and an integer value as arguments. It adjusts the RX equalization wideband / high pass ratio of the Aurora PHY.

Table 17: Valid values for `aurora_eq_mix`

Value	Description
0	50% wideband, 50% high pass
1	62.5% wideband, 37.5% high pass
2	75% wideband, 25% high pass
3	37.5% wideband, 62.5% high pass

`aurora_modes <aurora instance name>`

The `aurora_modes` command takes the Aurora block instance name string as an argument, and will print out the Aurora block configuration register values.

`aurora_status <aurora instance name>`

The `aurora_status` command takes the Aurora block instance name string as an argument, and will print out the current Aurora block status register values.

`aurora_soft_error <aurora instance name>`

The `aurora_soft_error` command takes the Aurora block instance name string as an argument, and will print out the current Aurora block soft error counter value.

```
aurora_soft_reset <aurora instance name>
```

The `aurora_soft_reset` command takes the Aurora block instance name string as an argument, and will reset the Aurora block soft error counter values, reset both the Aurora RX and TX FIFOs, and clear the latched `rx_fifo_full` value in the Aurora block status register.

Aurora block EDK API functions (BEE3, ML50x)

The EDK API is the same for both the BEE3 and ML50x Aurora block implementations. The API encapsulates common Aurora control register read and write operations as C functions that can be used in custom software applications. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE3 or ML50x base package. Include the header *aurora_streaming.h* for access to the API.

Data Structures

The following are the data structures used by the Aurora block EDK functions:

Table 18: `AuroraConfig` Structure Contents

Variable name	Type	Definition
`tx_diff_ctrl`	Unsigned 8 bits	transmitter differential voltage swing
`tx_preemp`	Unsigned 8 bits	transmitter pre-emphasis
`rx_eq_pole`	Unsigned 8 bits	receiver equalization high pass frequency
`rx_eq_mix`	Unsigned 8 bits	receiver equalization wideband/high pass mix
`rx_eq_en`	Unsigned 8 bits	receiver equalization enable
`loopback_mode`	Unsigned 8 bits	transceiver loopback mode
`power_down`	Unsigned 8 bits	power down transceiver

See *Xilinx Virtex-5 FPGA Aurora 3.0 User Guide* (ug353.pdf) parameter setting details.

Table 19: `AuroraStatus` Structure Contents

Variable name	Type	Definition
`tx_lock`	Unsigned 8 bits	`0x01` when TX refclk is up
`dcm_not_locked`	Unsigned 8 bits	`0x00` when user side DCM locked
`hard_error`	Unsigned 8 bits	`0x00` when no hard errors
`lane_up`	Unsigned 8 bits	`0x03` when both lanes are up
`channel_up`	Unsigned 8 bits	`0x01` when both lanes are bonded
`rx_fifo_empty`	Unsigned 8 bits	`0x01` when receive FIFO is empty
`rx_fifo_full`	Unsigned 8 bits	`0x01` when receive FIFO has been full
`tx_fifo_empty`	Unsigned 8 bits	`0x01` when transmit FIFO is empty

Functions

The following are the Aurora block EDK functions:

```
Xuint32
aurora_streaming_get_config(void * baseaddr_p,
                            AuroraConfig * ConfigSet)
```

Gets the Aurora configuration register and stores it as values in the `AuroraConfig` structure. Returns the raw Aurora configuration register bits as an unsigned 32 bit value.

```
XStatus
aurora_streaming_set_config(void * baseaddr_p,
                            AuroraConfig * ConfigSet)
```

Sets the Aurora configuration register according to the `AuroraConfig` structure. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
aurora_streaming_reset(void * baseaddr_p,
                       int reset_mode)
```

Sets the Aurora block reset mode. Bit 0 (bit mask 0x1) of `reset_mode` resets the Aurora PHY, and bit 1 (bit mask 0x2) resets the Aurora RX FIFO, TX FIFO, and soft error counter. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
Xuint32
aurora_streaming_get_softerrors(void * baseaddr_p)
```

Returns soft error count as an unsigned 32 bit value.

```
Xuint32
aurora_streaming_get_status(void * baseaddr_p,
                            AuroraStatus * Status)
```

Gets the Aurora status register and stores it as values in the `AuroraStatus` structure. Returns the raw Aurora configuration register bits as an unsigned 32 bit value.

Aurora block NectarOS functions (BEE4, miniBEE4)

The Aurora block includes a set of utility functions that enable the user to control the Aurora link directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever an Aurora block is used in a design. The source code for these functions are contained in the *drivers/xps_aurora_stream* subdirectory of the BPS-generated EDK project directory. The NectarOS functions and commands are specific to the BEE4 implementation of the Aurora block.

Commands

```
aurora_reset <aurora instance name>
```

The `aurora_reset` function takes the Aurora block instance name string as an argument. It clears the state of the Aurora link and makes it ready to stream data. Under most circumstances issuing an `aurora_reset` is all that is required to initialize the Aurora block. Other initialization functions tune the Aurora link PHY to remedy signal integrity issues that may be encountered in some special circumstances:

```
aurora_diffctrl <aurora instance name> <value>
```

The `aurora_diffctrl` function takes the Aurora block instance name string and a value as arguments. It adjusts the TX differential voltage swing of the Aurora PHY. This function sets TXDIFFCTRL in the GTX directly. Consult Xilinx's documentation for valid values (UG366).

```
aurora_preemp <aurora instance name> <value>
```

The `aurora_preemp` function takes the Aurora block instance name string and a value as arguments. It adjusts the TX pre-emphasis by directly setting the TXPREEMPHASIS port. Consult Xilinx's documentation for valid values (UG366).

```
aurora_eq_mix <aurora instance name> <value>
```

The `aurora_eq_mix` function takes the Aurora block instance name string and a value as arguments. It adjusts the RX equalization by directly setting the RXEQMIX port. Consult Xilinx's documentation for valid values (UG366).

```
aurora_status <aurora instance name>
```

The `aurora_status` command takes the Aurora block instance name string as an argument, and will print out the current Aurora block status register values.

Aurora block NectarOS functions (BEE7, VC707)

The Aurora block includes a set of utility functions that enable the user to control the Aurora link directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever an Aurora block is used in a design. The source code for these functions are contained in the *drivers/xps_aurora_64b66b* subdirectory of the BPS-generated EDK project directory. The NectarOS functions and commands are specific to the BEE7 implementation of the Aurora block.

Commands

```
aurora_get_status <core>
```

`The aurora_get_status function takes the Aurora core instance string as an argument. It returns the status of the individual lanes and the channel.`

```
aurora_reset <core>
```

The `aurora_reset` function takes the Aurora core instance name string as an argument. It clears the state of the Aurora link and makes it ready to stream data. Under most circumstances issuing an `aurora_reset` is all that is required to initialize the Aurora block. Other initialization functions tune the Aurora link PHY to remedy signal integrity issues that may be encountered in some special circumstances:

```
aurora_diffctrl <core> <lane> <swing (See MGT docs for TXDIFFCTRL>
```

The `aurora_diffctrl` function takes the Aurora core instance name string, the individual lane, and a value as arguments. It adjusts the TX differential voltage swing of the Aurora PHY. This function sets TXDIFFCTRL in the GTX directly. Consult Xilinx's documentation for valid values.

```
aurora_setprecursor <core> <lane> <emphasis (See MGT docs for
TXPRECURSOR; use negative value to invert)>
```

The `aurora_setprecursor` function takes the Aurora core instance name string, the individual lane, and a value as arguments. It adjusts the TX pre-cursor by directly setting the TXPRECURSOR port. Consult Xilinx's documentation for valid values.

`aurora_setpostcursor <core> <lane> <emphasis (See MGT docs for TXPOSTCURSOR; use negative value to invert)>`

The `aurora_setpostcursor` function takes the Aurora core instance name string, the individual lane, and a value as arguments. It adjusts the TX post-cursor by directly setting the TXPOSTCURSOR port. Consult Xilinx's documentation for valid values.

`aurora_setmaincursor <core> <lane> <emphasis (See MGT docs for TXMAINCURSOR; use negative value to invert)>`

The `aurora_setmaincursor` function takes the Aurora core instance name string, the individual lane, and a value as arguments. It adjusts the TX main-cursor by directly setting the TXMAINCURSOR port. Consult Xilinx's documentation for valid values.

`aurora_gt_drp_write <core> <gt_mask> <address> <data>`

The `aurora_gt_drp_write` function takes the Aurora block core instance name string, a gt mask, address, and data as arguments. It directly writes through drp.

`aurora_gt_drp_read <core> <gt_mask> <address>`

The `aurora_gt_drp_read` function takes the Aurora block core instance name string, a gt mask, and address as arguments. It returns the data directly through drp.

BEE3 ADC

BPS has built-in support for the complete family of analog-to-digital converter (ADC) expansion boards available for the BEE3 hardware platform. Through the use of an ADC expansion board, real-world analog signals can be processed and analyzed in real time on one or more FPGAs. BPS provides two types of Simulink blocks to represent the ADC data interface: *Direct* blocks which expose the exact outputs from the ADC device without any scaling or biasing, and *Basic* blocks which are wrappers around the *Direct* interface which de-bias, convert, and scale the ADC outputs to represent their true numeric quantities. The *Basic* blocks also feature a convenient ADC model in Simulink, which allows continuous analog Simulink signals to be used as a stimulus to the system. All ADC implementations feature a hardware logic interface which controls many different aspects of the ADC device operation as well as a software API, including both EDK-level drivers for custom software routines in addition to several NectarOS functions for convenient runtime control of the interface.

Note that because no buffering of data is performed between the hardware design and the ADC device, the hardware design must use the ADC digital output clock as the clock source for the system. This check is enforced by BPS at build time. Please refer to the Platform Configuration component for more information on selecting the system clock source.

The remainder of this section describes the complete functionality of the ADC block in BPS and its underlying hardware/software interface. It does not, however, attempt to describe the functionality or configuration of the ADC device itself. For more information on the ADC expansion board, please refer to the corresponding board and device user manual and datasheets.

ADC 3G Direct block for BEE3

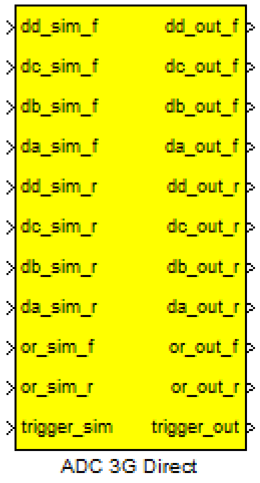
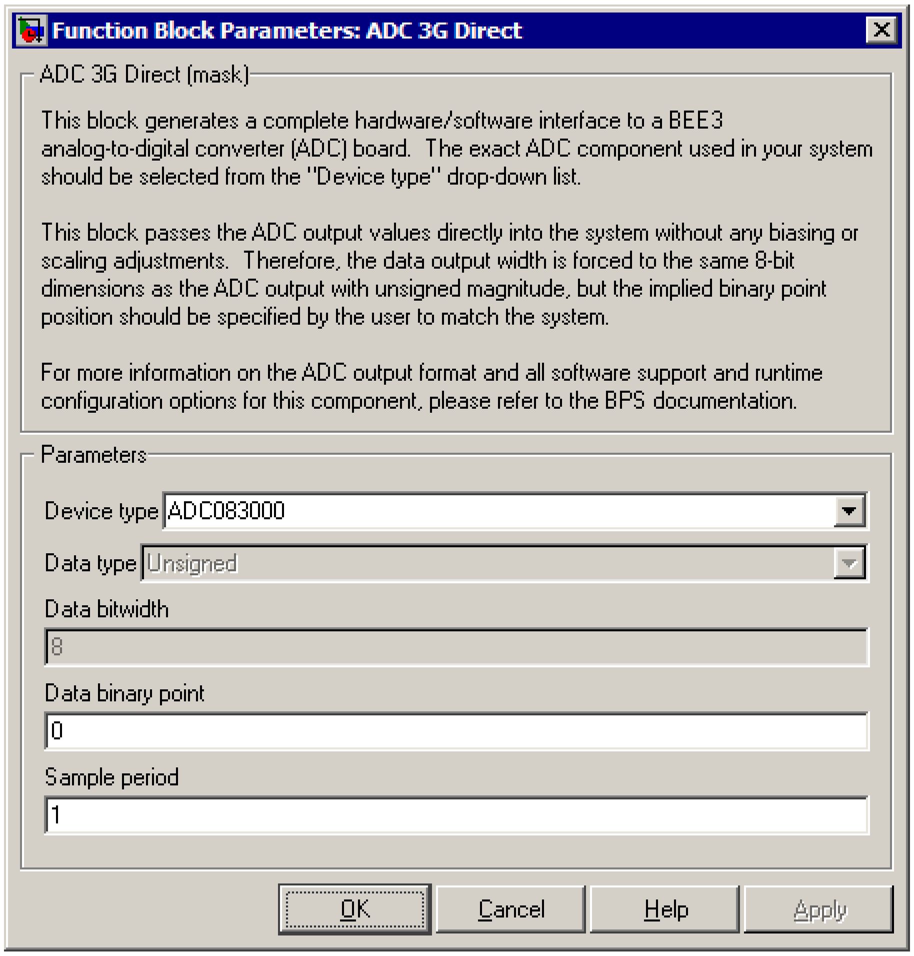


Figure 47: ADC 3G Direct block



Function Block Parameters: ADC 3G Direct

ADC 3G Direct (mask)

This block generates a complete hardware/software interface to a BEE3 analog-to-digital converter (ADC) board. The exact ADC component used in your system should be selected from the "Device type" drop-down list.

This block passes the ADC output values directly into the system without any biasing or scaling adjustments. Therefore, the data output width is forced to the same 8-bit dimensions as the ADC output with unsigned magnitude, but the implied binary point position should be specified by the user to match the system.

For more information on the ADC output format and all software support and runtime configuration options for this component, please refer to the BPS documentation.

Parameters

Device type: ADC083000

Data type: Unsigned

Data bitwidth: 8

Data binary point: 0

Sample period: 1

Figure 48: ADC 3G Direct dialog

The ADC 3G Direct block provides a direct, unmodified interface to the data outputs of the 3GSps (gigasample per second) ADC. Because the ADC to FPGA physical interface uses DDR (double data rate) signaling, two full sets of outputs are provided: one set for ADC data latched on the rising clock edge (the `*_r` signals), and one set for ADC data latched on the falling clock edge (the `*_f` signals). Within each set of outputs, each data port corresponds to one 8-bit sample taken from the ADC in sequence, with `Da` representing the first ("oldest") time-ordered sample, and `Dd` representing the last ("newest") time-ordered sample. Given this ordering, the output ports of the block are ordered in a complete sequence of 8 samples, with the most recent sample on top (`dd_out_f`) and the first sample on bottom (`da_out_r`).

For complete information on ADC sampling, please refer to the datasheet for your ADC device.

In addition to the ADC data outputs, the ADC out of range (OR) output is also provided in DDR format. The trigger signal is taken directly from the external trigger input connector on the ADC expansion board, and must be provided by the user. Finally, an input port is provided for each signal which may be driven by a Simulink data source during simulation.

Table 20: ADC 3G Direct block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 8 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 21: ADC 3G Direct port definitions

Port	Direction	Bit Width	Description
`dd_sim_f` `dc_sim_f` `db_sim_f` `da_sim_f`	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.
`dd_sim_r` `dc_sim_r` `db_sim_r` `da_sim_r`			
`or_sim_f`			
`or_sim_r`			
`trigger_sim`			
`dd_out_f` `dc_out_f` `db_out_f` `da_out_f`	out	8	Direct outputs from the ADC, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`dd_out_r` `dc_out_r` `db_out_r` `da_out_r`			Direct outputs from the ADC, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`or_out_f`	out	1	ADC out-of-range (OR) signal latched on the falling edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`or_out_r`	out	1	ADC out-of-range (OR) signal latched on the rising edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`trigger_out`	out	1	Trigger signal generated externally by the user.

ADC 1.5G Direct block for BEE3

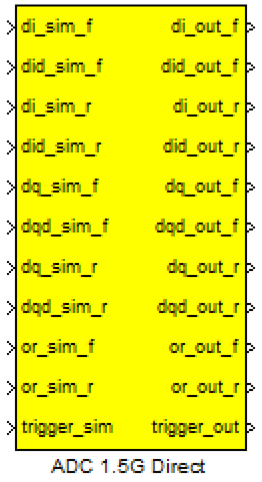
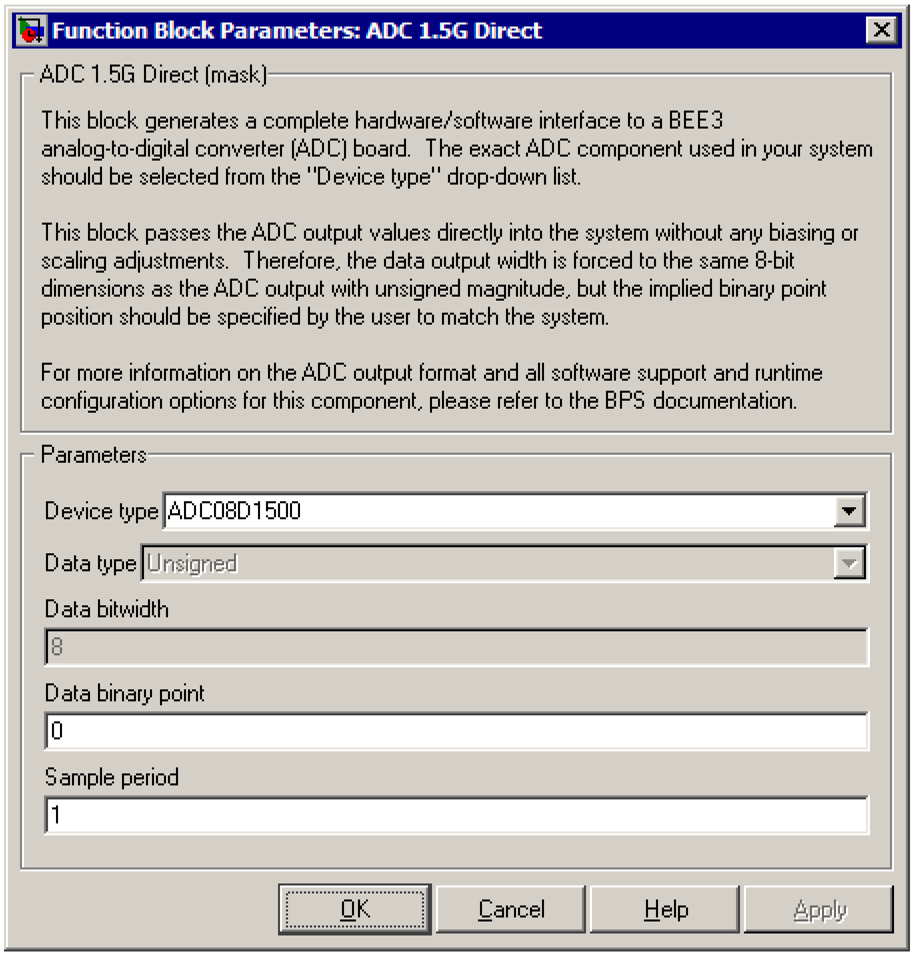


Figure 49: ADC 1.5G Direct block



ADC 1.5G Direct (mask)

This block generates a complete hardware/software interface to a BEE3 analog-to-digital converter (ADC) board. The exact ADC component used in your system should be selected from the "Device type" drop-down list.

This block passes the ADC output values directly into the system without any biasing or scaling adjustments. Therefore, the data output width is forced to the same 8-bit dimensions as the ADC output with unsigned magnitude, but the implied binary point position should be specified by the user to match the system.

For more information on the ADC output format and all software support and runtime configuration options for this component, please refer to the BPS documentation.

Parameters

Device type: ADC08D1500

Data type: Unsigned

Data bitwidth: 8

Data binary point: 0

Sample period: 1

Figure 50: ADC 1.5G Direct dialog

The ADC 1.5G Direct block provides a direct, unmodified interface to the data outputs of the 1.5GSps (gigasample per second) ADC. Because the ADC to FPGA physical interface uses DDR (double data rate) signaling, two full sets of outputs are provided: one set for ADC data latched on the rising clock edge (the `*_r` signals), and one set for ADC data latched on the falling clock edge (the `*_f` signals).

The 1.5GSps device is a dual-channel ADC which can operate in one of two modes: normal (non-DES) mode treats the I and Q channels as purely independent, while DES (dual edge sampling) mode samples the I-channel on both edges of the sampling clock and ignores the Q-channel input. The DES operating mode of the ADC is selected in BPS by using a device type which ends in `DES` (for example, for the ADC08D1500 device, the `ADC08D1500` device type selects normal mode, while the `ADC08D1500DES` device type selects DES mode).

In normal mode, the I and Q channels are fully independent, which results in four 8-bit samples per channel per hardware clock cycle. For example, for the I-channel, the time ordering of samples from the most recent (newest) to earliest (oldest) is `di_out_f`, `did_out_f`, `di_out_r`, `did_out_r`.

In DES mode, the Q channel is replaced with the samples taken on the I-channel on the inverse edge of the sampling clock. This effectively provides eight sequential 8-bit samples per hardware clock cycle. In this case, the time ordering of samples is interleaved between the I and Q outputs in the following order (from newest to oldest): `di_out_f`, `dq_out_f`, `did_out_f`, `dqd_out_f`, `di_out_r`, `dq_out_r`, `did_out_r`, `dqd_out_r`.

Note that for 1.5GSps devices, the I-channel analog input on the ADC expansion board is inverted, and must be corrected digitally in the hardware implementation. This is done automatically in the *Basic* block versions, which are described below.

For complete information on ADC sampling, please refer to the datasheet for your ADC device.

In addition to the ADC data outputs, the ADC out of range (OR) output is also provided in DDR format. OR outputs are common to both the I and Q channels regardless of the DES mode. The trigger signal is taken directly from the external trigger input connector on the ADC expansion board, and must be provided by the user. Finally, an input port is provided for each signal which may be driven by a Simulink data source during simulation.

Table 22: ADC 1.5G Direct block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board as well as normal or DES sampling mode.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 8 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 23: ADC 1.5G Direct port definitions

Port	Direction	Bit Width	Description
di_sim_f did_sim_f	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.
di_sim_r did_sim_r			
dq_sim_f dqd_sim_f			
dq_sim_r dqd_sim_r			
or_sim_f			
or_sim_r			
trigger_sim			
di_out_f did_out_f	out	8	Direct outputs from the ADC I-channel, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
di_out_r did_out_r			Direct outputs from the ADC I-channel, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
dq_out_f dqd_out_f			Direct outputs from the ADC Q-channel, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
dq_out_r dqd_out_r			Direct outputs from the ADC Q-channel, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
or_out_f	out	1	ADC out-of-range (OR) signal latched on the falling edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
or_out_r	out	1	ADC out-of-range (OR) signal latched on the rising edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
trigger_out	out	1	Trigger signal generated externally by the user.

ADC 3G Basic Block for BEE3

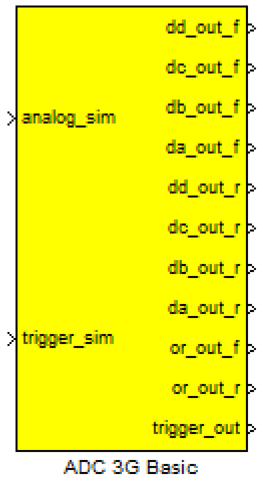
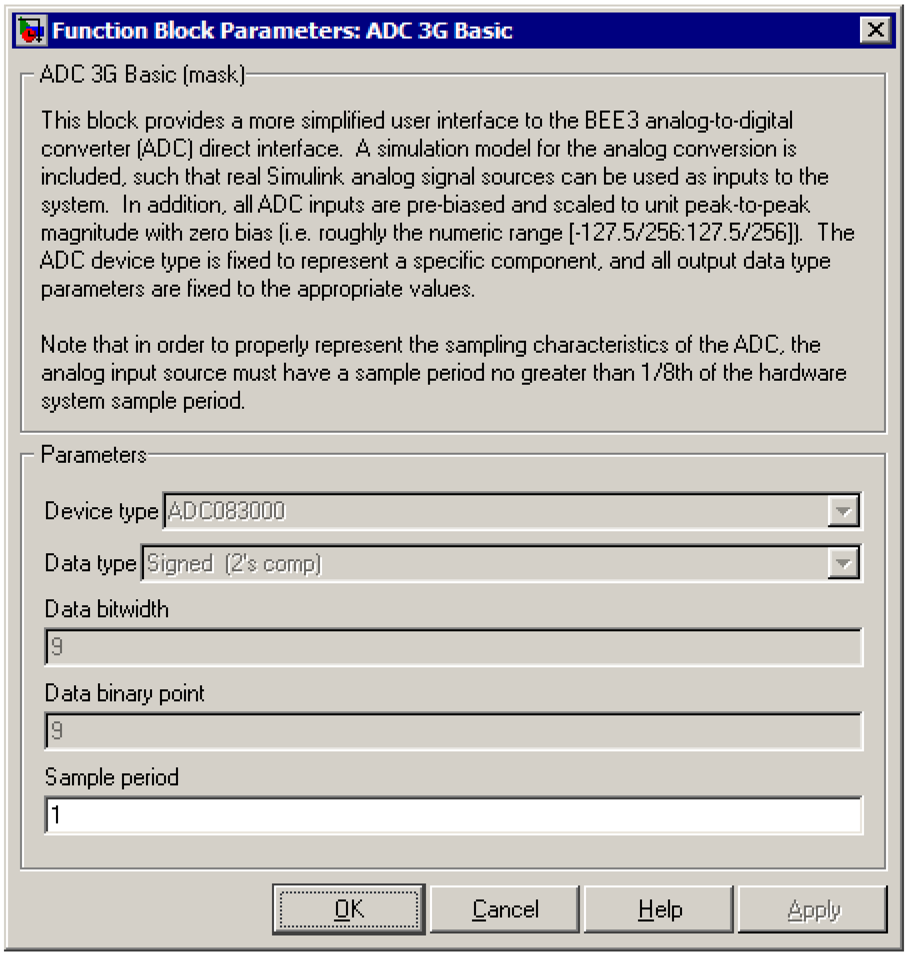


Figure 51: ADC 3G Basic Block



ADC 3G Basic (mask)

This block provides a more simplified user interface to the BEE3 analog-to-digital converter (ADC) direct interface. A simulation model for the analog conversion is included, such that real Simulink analog signal sources can be used as inputs to the system. In addition, all ADC inputs are pre-biased and scaled to unit peak-to-peak magnitude with zero bias (i.e. roughly the numeric range [-127.5/256:127.5/256]). The ADC device type is fixed to represent a specific component, and all output data type parameters are fixed to the appropriate values.

Note that in order to properly represent the sampling characteristics of the ADC, the analog input source must have a sample period no greater than 1/8th of the hardware system sample period.

Parameters

Device type: ADC083000

Data type: Signed (2's comp)

Data bitwidth: 9

Data binary point: 9

Sample period: 1

Figure 52: ADC 3G Basic dialog

The ADC 3G Basic block provides a more simplified interface to the ADC, including a digital sampling simulation model and pre-scaled output values for the hardware implementation.

The 3GSps ADC device generates digital samples as 8-bit integer values in the range [0:255], where 0 represents the most negative input level and 255 represents the most positive input level. The Basic block interface effectively converts these values into zero-biased signed fractions with unit peak-to-peak amplitude (approximately the range [-0.5:0.5]). This requires only a minimal amount of hardware (a single subtraction by a constant), and is pipelined by three cycles to preserve the high clock frequency typically required by ADC applications. Note that because the precise value required to achieve zero bias is -127.5, one additional bit of fractional precision is added to the ADC outputs, resulting in a signed 9-bit value.

The Simulink simulation model built into the block interface automatically downsamples, quantizes, and demultiplexes any discrete or continuous Simulink data source which is connected to the `analog_sim` input port of the block. The dynamic range of the input signal is expected to be in the same range as the hardware outputs. Any input values with a magnitude greater than 127.5/256 will be saturated and result in the out-of-range (OR) output being asserted.

Table 24: ADC 3G Basic block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board. It is fixed to match a single 3GSps ADC device.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Signed (2's comp)* to match the true range of the ADC output samples.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed to 9 to match the true range of the ADC output samples.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed to 9 to match the true range of the ADC output samples.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 25: ADC 3G Basic port definitions

Port	Direction	Bit Width	Description
`analog_sim`	in	N/A	Inputs which can be used to provide simulation data within Simulink. The analog signal input will automatically be sampled, quantized, and demultiplexed across the 8 digital outputs. Note that this requires the input signal to have a sample period which is at least 1/8 the hardware sample period.
`trigger_sim`			
`dd_out_f` `dc_out_f` `db_out_f` `da_out_f`	out	8	Direct outputs from the ADC, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`dd_out_r` `dc_out_r` `db_out_r` `da_out_r`			Direct outputs from the ADC, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`or_out_f`	out	1	ADC out-of-range (OR) signal latched on the falling edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`or_out_r`	out	1	ADC out-of-range (OR) signal latched on the rising edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`trigger_out`	out	1	Trigger signal generated externally by the user.

ADC 1.5G Basic block for BEE3

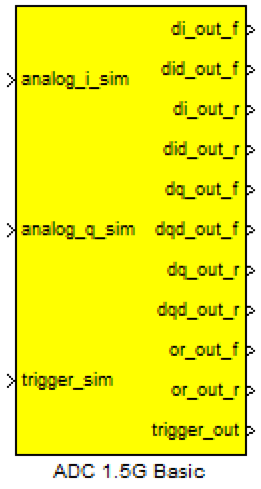
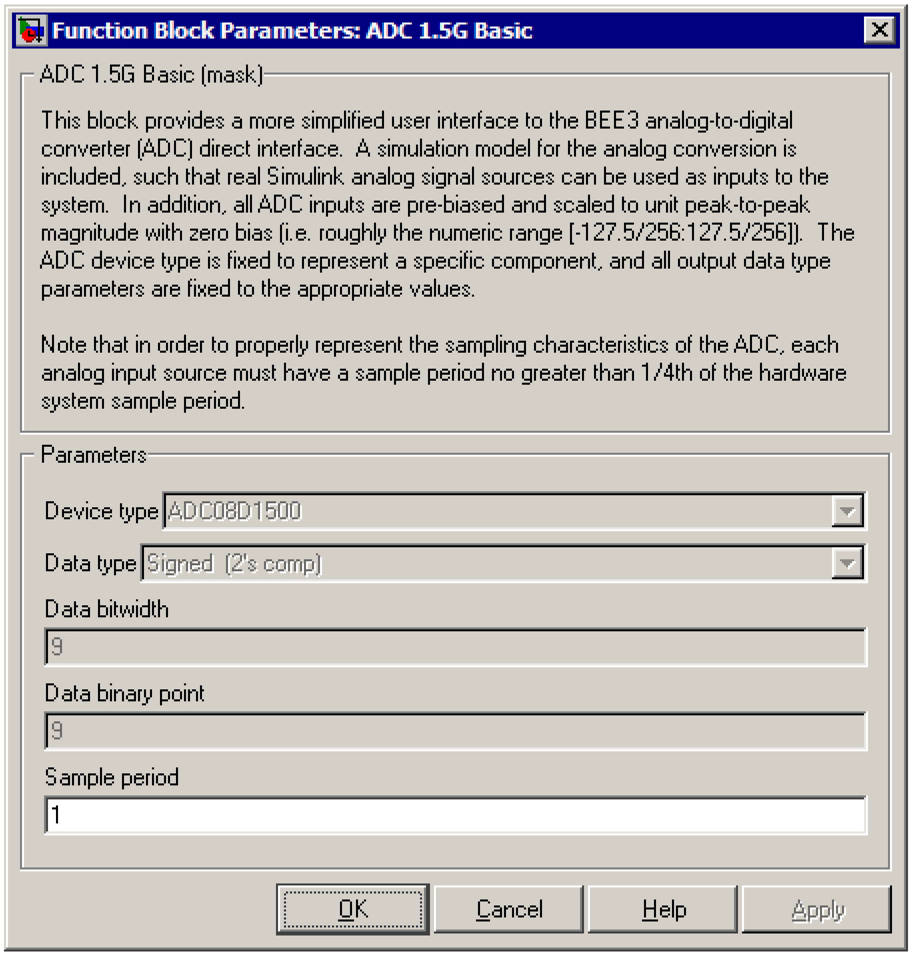


Figure 53: ADC 1.5G Basic block



ADC 1.5G Basic (mask)

This block provides a more simplified user interface to the BEE3 analog-to-digital converter (ADC) direct interface. A simulation model for the analog conversion is included, such that real Simulink analog signal sources can be used as inputs to the system. In addition, all ADC inputs are pre-biased and scaled to unit peak-to-peak magnitude with zero bias (i.e. roughly the numeric range [-127.5/256:127.5/256]). The ADC device type is fixed to represent a specific component, and all output data type parameters are fixed to the appropriate values.

Note that in order to properly represent the sampling characteristics of the ADC, each analog input source must have a sample period no greater than 1/4th of the hardware system sample period.

Parameters

Device type: ADC08D1500

Data type: Signed (2's comp)

Data bitwidth: 9

Data binary point: 9

Sample period: 1

Figure 54: ADC 1.5G Basic dialog

The ADC 1.5G Basic block provides a more simplified interface to the ADC, including a digital sampling simulation model and pre-scaled output values for the hardware implementation.

The 1.5GSps ADC device generates digital samples as 8-bit integer values in the range [0:255], where 0 represents the most negative input level and 255 represents the most positive input level. The Basic block interface effectively converts these values into zero-biased signed fractions with unit peak-to-peak amplitude (approximately the range [-0.5:0.5]). This requires only a minimal amount of hardware (a single subtraction by a constant), and is pipelined by three cycles to preserve the high clock frequency typically required by ADC applications. Note that because the precise value required to achieve zero bias is -127.5, one additional bit of fractional precision is added to the ADC outputs, resulting in a signed 9-bit value.

Since the 1.5GSps ADC expansion boards invert the analog I-channel input, all I-channel outputs from the ADC are automatically inverted digitally by reversing the inputs to each bias subtraction. This effectively makes the numeric inversion of the samples “free” in terms of hardware resources and preserves the properly scaled values of each ADC sample.

The Simulink simulation model built into the block interface automatically downsamples, quantizes, and demultiplexes any discrete or continuous Simulink data source which is connected to the `analog_sim` input port of the block. The dynamic range of the input signal is expected to be in the same range as the hardware outputs. Any input values with a magnitude greater than 127.5/256 will be saturated and result in the out-of-range (OR) output being asserted.

Table 26: ADC 1.5G Basic block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board. It is fixed to match a single 1.5GSps ADC device.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Signed (2’s comp)* to match the true range of the ADC output samples.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed to 9 to match the true range of the ADC output samples.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed to 9 to match the true range of the ADC output samples.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 27: ADC 1.5G Basic port definitions

Port	Direction	Bit Width	Description
`analog_i_sim`	in	N/A	Inputs which can be used to provide simulation data within Simulink. The analog signal inputs will automatically be sampled, quantized, and delayed across the 4 digital outputs of each corresponding channel. Note that this requires the input signal to have a sample period which is at least 1/4 the hardware sample period.
`analog_q_sim`			
`trigger_sim`			
`di_out_f` `did_out_f`	out	8	Direct outputs from the ADC I-channel, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`di_out_r` `did_out_r`			Direct outputs from the ADC I-channel, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`dq_out_f` `dqd_out_f`			Direct outputs from the ADC Q-channel, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`dq_out_r` `dqd_out_r`			Direct outputs from the ADC Q-channel, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`or_out_f`	out	1	ADC out-of-range (OR) signal latched on the falling edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`or_out_r`	out	1	ADC out-of-range (OR) signal latched on the rising edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`trigger_out`	out	1	Trigger signal generated externally by the user.

ADC 1.5G DES Basic block for BEE3

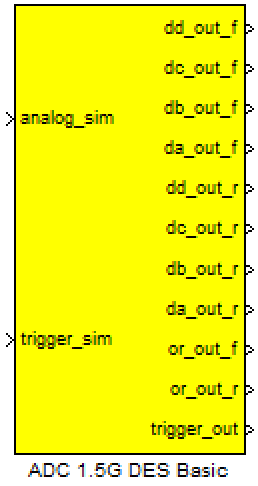
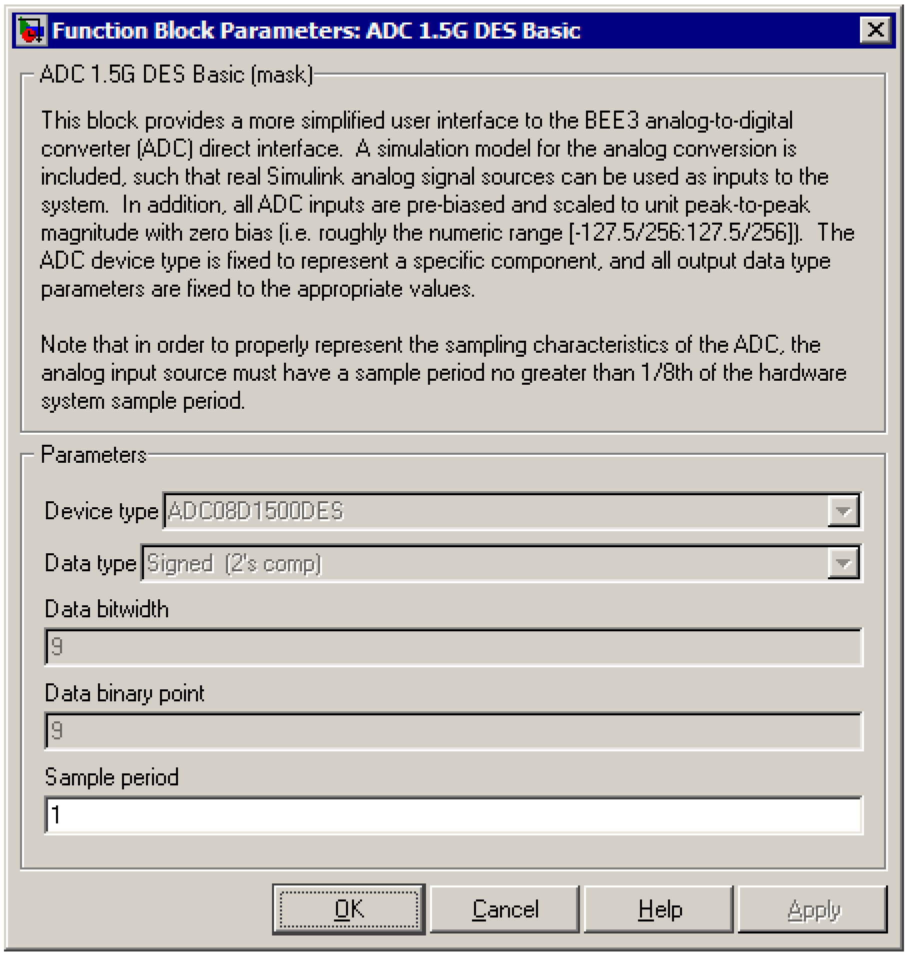


Figure 55: ADC 1.5G DES Basic block


Function Block Parameters: ADC 1.5G DES Basic

ADC 1.5G DES Basic (mask)

This block provides a more simplified user interface to the BEE3 analog-to-digital converter (ADC) direct interface. A simulation model for the analog conversion is included, such that real Simulink analog signal sources can be used as inputs to the system. In addition, all ADC inputs are pre-biased and scaled to unit peak-to-peak magnitude with zero bias (i.e. roughly the numeric range [-127.5/256:127.5/256]). The ADC device type is fixed to represent a specific component, and all output data type parameters are fixed to the appropriate values.

Note that in order to properly represent the sampling characteristics of the ADC, the analog input source must have a sample period no greater than 1/8th of the hardware system sample period.

Parameters

Device type: ADC08D1500DES

Data type: Signed (2's comp)

Data bitwidth: 9

Data binary point: 9

Sample period: 1

Figure 56: ADC 1.5G DES Basic dialog

The ADC 1.5G DES Basic block provides a more simplified interface to the ADC, including a digital sampling simulation model and pre-scaled output values for the hardware implementation.

The 1.5GSps ADC device generates digital samples as 8-bit integer values in the range [0:255], where 0 represents the most negative input level and 255 represents the most positive input level. The Basic block interface effectively converts these values into zero-biased signed fractions with unit peak-to-peak amplitude (approximately the range [-0.5:0.5]). This requires only a minimal amount of hardware (a single subtraction by a constant), and is pipelined by three cycles to preserve the high clock frequency typically required by ADC applications. Note that because the precise value required to achieve zero bias is -127.5, one additional bit of fractional precision is added to the ADC outputs, resulting in a signed 9-bit value.

Since the 1.5GSps ADC expansion boards invert the analog I-channel input, and in DES mode only the I-channel is sampled, all outputs from the ADC are automatically inverted digitally by reversing the inputs to each bias subtraction. This effectively makes the numeric inversion of the samples "free" in terms of hardware resources and preserves the properly scaled values of each ADC sample.

Note that since the 1.5GSps ADC in DES mode effectively behaves like a single ADC with twice the sample rate, the output ports for the ADC 1.5G DES Basic block are renamed to match their behavior, which is similar to the 3GSps ADC component. This prevents the user from having to manually re-order and re-interpret the ADC outputs.

The Simulink simulation model built into the block interface automatically downsamples, quantizes, and demultiplexes any discrete or continuous Simulink data source which is connected to the `analog_sim` input port of the block. The dynamic range of the input signal is expected to be in the same range as the hardware outputs. Any input values with a magnitude greater than 127.5/256 will be saturated and result in the out-of-range (OR) output being asserted.

Table 28: ADC 1.5G DES Basic block parameters

Parameter	Description
Device type	This parameter defines the type of ADC device present on the ADC expansion board. It is fixed to match a single 1.5GSps ADC device.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Signed (2's comp)* to match the true range of the ADC output samples.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed to 9 to match the true range of the ADC output samples.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed to 9 to match the true range of the ADC output samples.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 29: ADC 1.5G DES Basic port definitions

Port	Direction	Bit Width	Description
`analog_sim` `trigger_sim`	in	N/A	Inputs which can be used to provide simulation data within Simulink. The analog signal input will automatically be sampled, quantized, and delayed across the 8 digital outputs. Note that this requires the input signal to have a sample period which is at least 1/8 the hardware sample period.
`dd_out_f` `dc_out_f` `db_out_f` `da_out_f`	out	8	Direct outputs from the ADC, latched on the falling edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`dd_out_r` `dc_out_r` `db_out_r` `da_out_r`	out	8	Direct outputs from the ADC, latched on the rising edge of the system clock. Refer to the ADC device datasheet for complete precision and sampling information.
`or_out_f`	out	1	ADC out-of-range (OR) signal latched on the falling edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`or_out_r`	out	1	ADC out-of-range (OR) signal latched on the rising edge of the system clock. Indicates if an out-of-range input caused saturation in the ADC.
`trigger_out`	out	1	Trigger signal generated externally by the user.

ADC block NectarOS functions

BPS adds the following functions to NectarOS whenever an ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_adc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`adc_init`

The `adc_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will write default values into all the ADC device's control registers, initiate an ADC self-calibration, reset the ADC digital output clock, reset the internal design clock PLL, reset all internal I/O logic, and set the input pad delays for all signals received from the ADC to their default value.

Repeated function calls

(none)

Commands

`adc_get_status`

The `adc_get_status` function will print the current status of the internal ADC interface logic. Note that because all ADC configuration registers are write-only, this function will not return any information about the state of the ADC device itself. For a description of all the interface logic status bits, please see the `XAdc_Status` data structure reference below.

`adc_pll_reset`

The `adc_pll_reset` function will trigger a reset of the internal design clock PLL. Note that this will cause an interruption in the clock driven to the user design logic, but will not interfere with the BPS base infrastructure (such as the embedded processor and NectarOS). It may be necessary to call this function if the ADC clock is reset or disconnected. The current locked state of the design clock PLL can be checked via the `adc_get_status` command.

`adc_dclk_reset`

The `adc_dclk_reset` function will trigger a reset of the ADC digital output clock. Asserting this reset will cause the `DCLK_RST` pin on the ADC to be asserted and then released. Note that if the ADC digital output clock is reset, it will most likely be necessary to reset the internal PLL via the `adc_pll_reset` command before the design clock will become reliable.

`adc_tpsync_reset`

The `adc_tpsync_reset` function will reset the internal test pattern synchronization state machines embedded into the ADC interface logic. While this command will always trigger a reset of the synchronization logic, it only has a useful effect when the ADC device in use supports the test pattern generation feature and the test pattern mode is currently active. When the ADC test pattern mode is active and the synchronization logic has been reset, the synchronization state of all inputs from the ADC can be checked via the `adc_get_status` command.

`adc_reg_write <reg_addr> <value>`

The `adc_reg_write` function will write `value` into the ADC configuration register at address `reg_addr`. For example, to adjust the input voltage bias of the ADC083000 device upwards by three steps, you would use the command `adc_reg_write x2 x037F`, since the offset adjust register is located at address 2, the offset value is located in bits 8-15, the sign bit is located in bit 7, and the remaining bits must be set to 1.

`adc_test_pattern <(start|on|stop|off)>`

The `adc_test_pattern` function will configure the ADC device to either enter or exit its test pattern mode. The test pattern mode can be used to validate the ADC interface logic and/or to retune the input pad delay elements. The arguments `start` or `on` will cause the ADC to enter the test pattern mode, and the arguments `stop` or `off` will cause the ADC to exit the test pattern mode. Note that not all ADC devices support the test pattern generation mode – if a design is built for an ADC device type that does not support test pattern mode, an indicator message will be printed on the NectarOS console if this command is called.

`adc_delay_set <idelay_val>`

The `adc_delay_set` function sets the input pad delay for all ADC signals. The `idelay_val` argument must be an integer between 0 and 63, and corresponds to the delay setting that will be assigned to all inputs coming from the ADC. On the FPGA, this input delay is implemented using the Xilinx Virtex-5 IDELAY primitive. The default value should be correct for all board configurations at the peak ADC sample clock rate, but may need to be adjusted in certain applications. The `adc_delay_sweep` command should be used to determine the ideal delay value, assuming the ADC device being used supports the required test pattern mode.

`adc_delay_sweep <usleep_duration>`

The `adc_delay_sweep` function will sweep across all ADC input pins and possible input delay values and plot a table of whether each combination passed or failed. The `usleep_duration` argument determines approximately how many microseconds the processor will wait before checking the synchronization status bits (10000 is a reasonable value for most cases). Note that before running this command, the ADC must already be in its test pattern mode, which implies that the AD device itself must provide test pattern support. Also, the delay settings for each pin will remain in the maximum delay state at the end of this function, and must be manually set to the desired value via the `adc_delay_set` function before resuming normal operation.

ADC block EDK API functions

The underlying hardware/software interface for the ADC component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the ADC. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE3 base package. Include the header *xadc.h* for access to the API.

Data Structures

There are two data structures used by several of the ADC EDK driver function calls which correspond to all the software-accessible control and status bits contained in the hardware interface logic. Each of these data structures are described in the following tables.

Table 30: `XAdc_Control` Structure Contents

Variable name	**Type**	**Definition**
`adc_tpsync_rst`	Unsigned 8 bits	ADC test pattern synchronization logic reset
`adc_dclk_rst`	Unsigned 8 bits	ADC digital output clock reset
`adc_cal`	Unsigned 8 bits	ADC on-demand calibration pin
`idelayctrl_rst`	Unsigned 8 bits	Input pad IDELAY control logic reset
`adc_pll_rst`	Unsigned 8 bits	Internal PLL reset

Table 31: `XAdc_Status` Structure Contents

Variable name	**Type**	**Definition**
`da_sync`	Unsigned 8 bits	ADC `Da` output test pattern synchronization status (only valid when ADC test pattern mode is active)
`db_sync`	Unsigned 8 bits	ADC `Db` output test pattern synchronization status (only valid when ADC test pattern mode is active)
`dc_sync`	Unsigned 8 bits	ADC `Dc` output test pattern synchronization status (only valid when ADC test pattern mode is active)
`dd_sync`	Unsigned 8 bits	ADC `Dd` output test pattern synchronization status (only valid when ADC test pattern mode is active)
`or_sync`	Unsigned 8 bits	ADC `OR` output test pattern synchronization status (only valid when ADC test pattern mode is active)
`pll_lock`	Unsigned 8 bits	Internal PLL locked status
`cal_run`	Unsigned 8 bits	ADC calibration running status
`idelayctrl_rdy`	Unsigned 8 bits	Input pad IDELAY logic ready status

Functions

```
XStatus
XAdc_CfgRegWrite(void * baseaddr_p,
                 Xuint8 reg_addr,
                 Xuint16 reg_val);
```

Writes the value `reg_val` to the ADC configuration register at address `reg_addr` using the ADC serial control interface. The pointer `baseaddr_p` should be the base address of the ADC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XAdc_CtlRegWrite(void * baseaddr_p,
                 XAdc_Control * ControlSet);
```

Writes the set of values contained in the `XAdc_Control` structure referenced by the pointer `ControlSet` to the internal ADC interface control register. The pointer `baseaddr_p` should be the base address of the ADC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XAdc_SetIdelayPinMask(void * baseaddr_p,
                      Xuint32 data_pin_mask,
                      Xuint32 or_pin_mask);
```

Sets the pin mask for any IDELAY control operations. A value of 1 will enable any IDELAY operations to modify the corresponding pin delay values, while a value of 0 will ignore any IDELAY operations for that pin. The pointer `baseaddr_p` should be the base address of the ADC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XAdc_SetIdelayValue(void * baseaddr_p,
                    Xuint8 delay_val);
```

Sets the IDELAY value for all enabled pins to the value specified in `delay_val`. This requires a clear of all IDELAY settings, followed by an incremental increase up to `delay_val`. The pointer `baseaddr_p` should be the base address of the ADC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XAdc_GetStatus(void * baseaddr_p,
               XAdc_Status * Status);
```

Gets the current values of all internal ADC interface status bits and sets them in the `XAdc_Status` structure referenced by the pointer `Status`. The pointer `baseaddr_p` should be the base address of the ADC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

BEE3 DAC

BPS has built-in support for the complete family of digital-to-analog converter (DAC) expansion boards available for the BEE3 hardware platform. Through the use of a DAC expansion board, one or more FPGAs can be used to generate analog signals for use in a variety of signal processing and communications applications. The DAC implementation provided by BPS features a hardware logic interface which controls many different aspects of the DAC device operation as well as a software API, including both EDK-level drivers for custom software routines in addition to several NectarOS functions for convenient runtime control of the interface.

The DAC expansion board is currently only supported when inserted into the lower pair of QSH slots on BEE3. Each DAC expansion board is manufactured with a default master-slave setting, where the left DAC is configured as a slave and the right DAC is configured as a master. In this configuration, both DAC devices must receive the same high-speed sample clock source, and the low-speed output clock is only driven from the slave DAC to the FPGA. The FPGA connected to the master DAC receives its clock from the slave FPGA, which forwards the output clock using a pin on the on-board ring bus. BPS automatically performs all device and interface configuration by detecting the current FPGA ID at runtime – no steps are required by the user to assign the proper settings.

Note that because no buffering of data is performed between the hardware design and the DAC device, the hardware design must use the DAC low-speed output clock as the clock source for the system. This check is enforced by BPS at build time. Please refer to the Platform Configuration component for more information on selecting the system clock source.

The remainder of this section describes the complete functionality of the DAC block in BPS and its underlying hardware/software interface. It does not, however, attempt to describe the functionality or configuration of the DAC device itself. For more information on the DAC expansion board, please refer to the corresponding board and device user manual and datasheets.

DAC block for BEE3

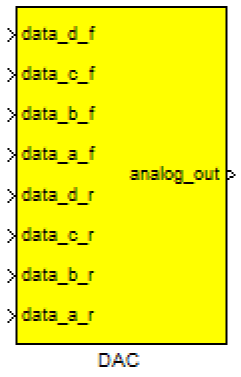
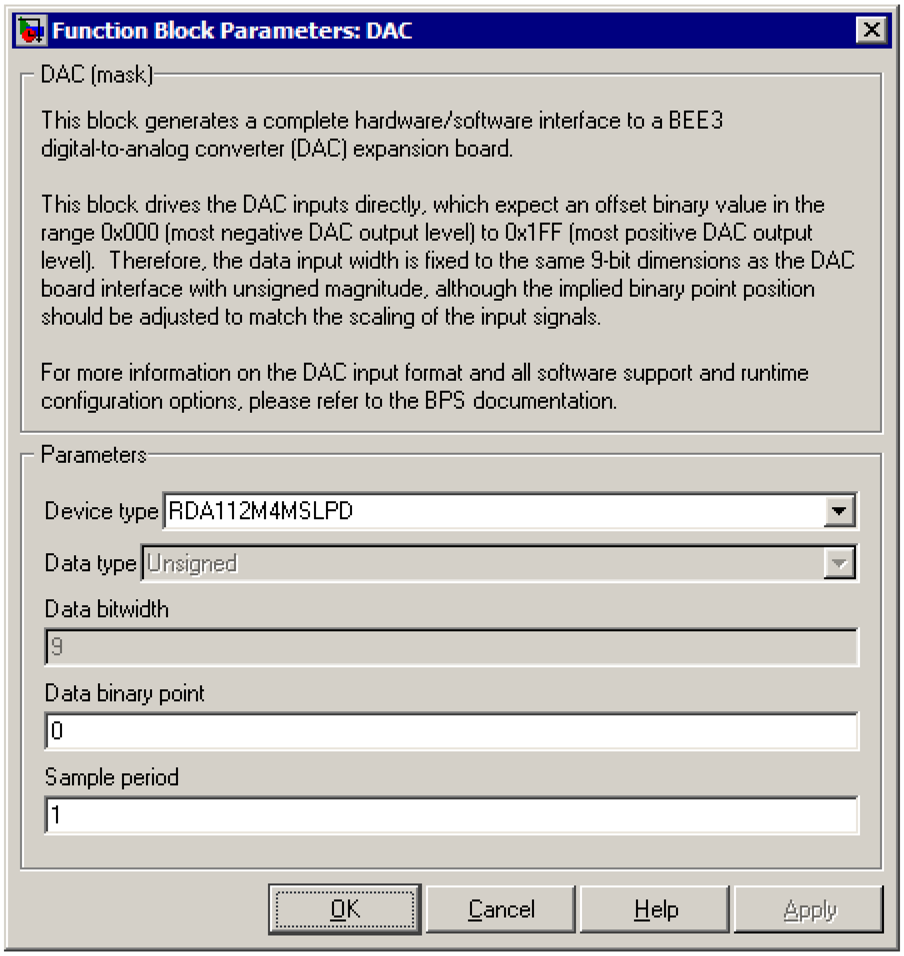


Figure 57: DAC block



DAC (mask)

This block generates a complete hardware/software interface to a BEE3 digital-to-analog converter (DAC) expansion board.

This block drives the DAC inputs directly, which expect an offset binary value in the range 0x000 (most negative DAC output level) to 0x1FF (most positive DAC output level). Therefore, the data input width is fixed to the same 9-bit dimensions as the DAC board interface with unsigned magnitude, although the implied binary point position should be adjusted to match the scaling of the input signals.

For more information on the DAC input format and all software support and runtime configuration options, please refer to the BPS documentation.

Parameters

Device type: RDA112M4MSLPD

Data type: Unsigned

Data bitwidth: 9

Data binary point: 0

Sample period: 1

Figure 58: DAC dialog

The DAC block provides a direct, unmodified interface to the data inputs of the DAC device. Because the FPGA to DAC physical interface uses DDR (double data rate) signaling, two full sets of inputs are provided: one set for data to be driven to the DAC on the rising clock edge (the `*_r` signals), and one set for data to be driven to the DAC on the falling clock edge (the `*_f` signals). Within each set of inputs, each data port corresponds to a value which will be driven by the DAC on its analog output in sequence, with `data_a` representing the first time-ordered sample to be driven, and `data_d` representing the last time-ordered sample to be driven. This effectively means that all 8 inputs to the DAC block represent a sequence of 8 samples to be driven by the DAC in order from bottom (`data_a_r`) to top (`data_d_f`).

For complete information on DAC operation, please refer to the datasheet for your DAC device.

In addition to the DAC inputs, an output port (`analog_out`) is provided for convenience during simulation in Simulink. This output will represent the analog waveform which would be driven by the DAC on the hardware. While this signal is provided only for simulation, it can be very helpful for ensuring that the hardware design is using the correct sample and bit ordering.

Table 32: DAC block parameters

Parameter	**Description**
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Data type	This parameter determines the inferred arithmetic type for the signal. This value is fixed to *Unsigned* to match the input format expected by the DAC.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 9 bits to match the DAC data inputs in 4:1 demux mode.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 33: DAC port definitions

Port	Direction	Bit Width	Description
`data_d_f` `data_c_f` `data_b_f` `data_a_f`	in	9	Direct inputs to the DAC, driven on the falling edge of the system clock. Refer to the DAC device datasheet for complete precision and sampling information.
`data_d_r` `data_c_r` `data_b_r` `data_a_r`			Direct inputs to the DAC, driven on the rising edge of the system clock. Refer to the DAC device datasheet for complete precision and sampling information.
`analog_out`	out	N/A	Output which can be used to validate the DAC input data during simulation in Simulink. The sample rate of the output will be 8 times the fundamental sample period of the hardware design with an additional first-order hold to smooth the waveform.

DAC block NectarOS functions

BPS adds the following functions to NectarOS whenever an ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_dac* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`dac_init`

The `dac_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will automatically detect which FPGA is running the system and configure the DAC interface accordingly. In addition, a reset of the ODELAY elements and a reset of the internal hardware clock PLL will be asserted.

Repeated function calls

(none)

Commands

`dac_get_status`

The `dac_get_status` function will print the current state of the internal DAC interface logic. For a description of all the interface logic status and control bits, please see the `XDac_Status` and `XDac_Control` data structure references below.

`dac_pll_reset`

The `dac_pll_reset` function will trigger a reset of the internal design clock PLL. Note that this will cause an interruption in the clock driven to the user design logic, but will not interfere with the BPS base infrastructure (such as the embedded processor and NectarOS). It may be necessary to call this function if the DAC clock is reset or disconnected. The current locked state of the design clock PLL can be checked via the `dac_get_status` command.

`dac_idelay_reset`

The `dac_idelay_reset` function will trigger a reset of the IDELAYCTRL instance which manages all ODELAY elements used by the FPGA outputs to the DAC. This reset is also performed during initialization. Since all ODELAY elements use fixed delay value, use of this reset should not be necessary, but is provided for debugging purposes.

`dac_force_clock <(master|slave)>`

The `dac_force_clock` function will override the clock settings automatically derived from the current FPGA ID and force the specified mode. In master mode, the internal PLL will receive its input clock from the slave DAC, which is forwarded over the on-board ring bus by the slave FPGA. In slave mode, the internal PLL will receive its input clock directly from the DAC. This option should not be used except for debugging or when working with custom hardware.

`dac_set_phase <(0|1|2|3)>`

The `dac_set_phase` function will assign the given value to the DSEL pins on the DAC device. The DSEL pins determine which phase of the low-speed clock the DAC will use to latch the data values driven by the FPGA. Since BPS configures the DAC to operate in 4:1 demux mode, this value can be any integer between 0 and 3, each of which represents a different phase of the low-speed clock with respect to the high-speed clock input connected to the DAC board. A typical value is automatically set by BPS during the initialization routine based on the maximum DAC clock frequency, but different settings may be required under varying frequencies and temperatures.

`dac_tp_start <a1> <b1> <c1> <d1> <a2> <b2> <c2> <d2>`

The `dac_tp_start` function will switch the DAC interface logic into test pattern mode. In test pattern mode, the FPGA will drive a fixed sequence of 8 values to the DAC device. The order in which the samples will be driven by the DAC is the same as the order of the arguments: `a1` is the first sample in the sequence, and `d2` is the last sample. This mode is provided to help verify proper operation of the DAC board and any external equipment. For example, the command `dac_tp_start 0 511 0 511 0 511 0 511` will cause the DAC generate a maximum-frequency sine wave by configuring the test pattern to use alternating peak-to-peak values.

`dac_tp_stop`

The `dac_tp_stop` function will cause the DAC interface to exit test pattern mode and return to normal operation.

DAC block EDK API functions

The underlying hardware/software interface for the DAC component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the DAC. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE3 base package. Include the header *xdac.h* for access to the API.

Data Structures

There are three data structures used by several of the DAC EDK driver function calls which correspond to all the software-accessible control and status bits contained in the hardware interface logic. Each of these data structures are described in the following tables.

Table 34: `XDac_Control` Structure Contents

Variable name	**Type**	**Definition**
`dac_clksel`	Unsigned 8 bit	Drives DAC CLKSEL pin
`dac_mxsel`	Unsigned 8 bit	Drives DAC MXSEL pin
`dac_mxsel_n`	Unsigned 8 bit	Should be set to inverse of MXSEL to properly configure on-board switches
`dac_dsel`	Unsigned 8 bit	Drives DAC DSEL pins
`test_pat_en`	Unsigned 8 bit	Sets DAC interface to drive 8-sample test pattern to DAC instead of normal inputs
`idelay_rst`	Unsigned 8 bit	Connected to IDELAYCTRL reset input
`clk_src`	Unsigned 8 bit	Determines internal PLL clock source (0 = forwarded clock from adjacent FPGA; 1 = DAC low-speed output clock)
`pll_rst`	Unsigned 8 bit	Connected to internal PLL reset input

Table 35: `XDac_Status` Structure Contents

Variable name	**Type**	**Definition**
`idelay_rdy`	Unsigned 8 bit	Connected to IDELAYCTRL ready output
`pll_lock`	Unsigned 8 bit	Connected to internal PLL locked output

Table 36: `XDac_TestPattern` Structure Contents

Variable name	**Type**	**Definition**
`data_a_r`	Unsigned 16 bit	DAC data input A, driven on rising clock edge
`data_b_r`	Unsigned 16 bit	DAC data input B, driven on rising clock edge
`data_c_r`	Unsigned 16 bit	DAC data input C, driven on rising clock edge
`data_d_r`	Unsigned 16 bit	DAC data input D, driven on rising clock edge
`data_a_f`	Unsigned 16 bit	DAC data input A, driven on falling clock edge
`data_b_f`	Unsigned 16 bit	DAC data input B, driven on falling clock edge
`data_c_f`	Unsigned 16 bit	DAC data input C, driven on falling clock edge
`data_d_f`	Unsigned 16 bit	DAC data input D, driven on falling clock edge

Functions

```
XStatus
XDac_SetControl(void * baseaddr_p,
                XDac_Control * Control);
```

Writes the set of control signals defined in the structure referenced by `Control` into the DAC interface control register. The pointer `baseaddr_p` should be the base address of the DAC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XDac_GetControl(void * baseaddr_p,
                XDac_Control * Control);
```

Fills the structure referenced by `Control` with the current control signal values read from the DAC interface control register. The pointer `baseaddr_p` should be the base address of the DAC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XDac_GetStatus(void * baseaddr_p,
               XDac_Status * Status);
```

Fills the structure referenced by `Status` with the current status signal values read from the DAC interface control register. The pointer `baseaddr_p` should be the base address of the DAC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XDac_StartTestPattern(void * baseaddr_p,
                      XDac_TestPattern * Pattern);
```

Writes the set of data samples defined in the structure referenced by Pattern into the test pattern value registers and then puts the DAC interface into test pattern output mode. Note that although software defines the test pattern values to be unsigned 16-bit halfwords, only the 9 least significant bits are used by hardware. When putting the DAC into test pattern mode, a read-modify-write operation is performed on the control register to preserve the values of all other control bits. The pointer `baseaddr_p` should be the base address of the DAC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XDac_StopTestPattern(void * baseaddr_p);
```

Clears the test pattern enable bit in the DAC interface control register, returning the DAC to normal operation. The pointer `baseaddr_p` should be the base address of the DAC interface hardware instance on the main processor PLB bus. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

ChipScope

BPS supports the use of the Xilinx ChipScope JTAG debugging and verification tool in any FPGA design. In addition to automatically inserting all the necessary logic components for ChipScope control, triggering, and data capture, BPS will also generate a ChipScope Definition and Connection (CDC) file which can be imported by the ChipScope Analyzer for Virtex 6 and earlier, or a (LTX) probes file which can be imported by the Vivado Hardware Manager for Virtex 7 and later to retrieve complete signal names and types from the original Simulink system. All ChipScope components created by BPS are platform-independent, therefore all Simulink blocks are contained in the BPS Common Blockset.

ChipScope Configuration block

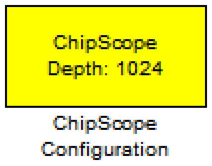
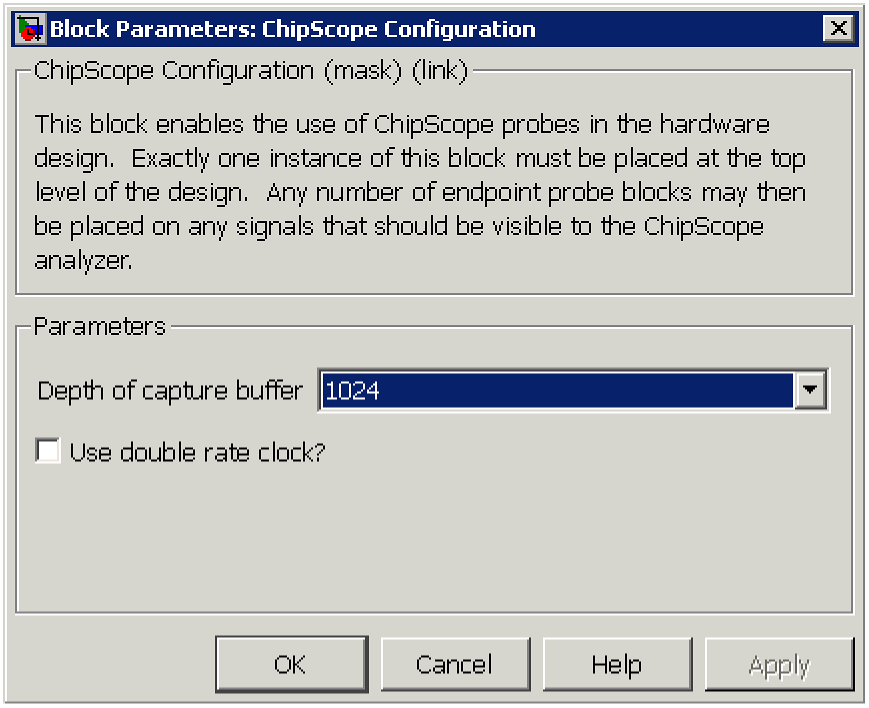


Figure 59: ChipScope Configuration block


Block Parameters: ChipScope Configuration

ChipScope Configuration (mask) (link)

This block enables the use of ChipScope probes in the hardware design. Exactly one instance of this block must be placed at the top level of the design. Any number of endpoint probe blocks may then be placed on any signals that should be visible to the ChipScope analyzer.

Parameters

Depth of capture buffer: 1024

☐ Use double rate clock?

Figure 60: ChipScope Configuration dialog

The ChipScope Configuration block is used by BPS to enable support for ChipScope probes in the design. Exactly one ChipScope Configuration block must exist in the system in order to use ChipScope for runtime signal capture on the FPGA. Placing any Probe blocks in the system without the presence of a ChipScope Configuration block will cause an error.

Table 37: ChipScope Configuration block parameters

Parameter	Description
Capture buffer depth	This value determines the overall depth of the ChipScope capture buffer to be used on the FPGA. Larger values consume more resources in hardware, but allow a deeper sample history to be observed per capture.
Use double rate clock?	Check the box to enable ChipScope probes to sample at twice the fundamental system frequency.

ChipScope Probe block

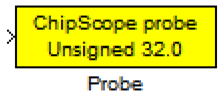
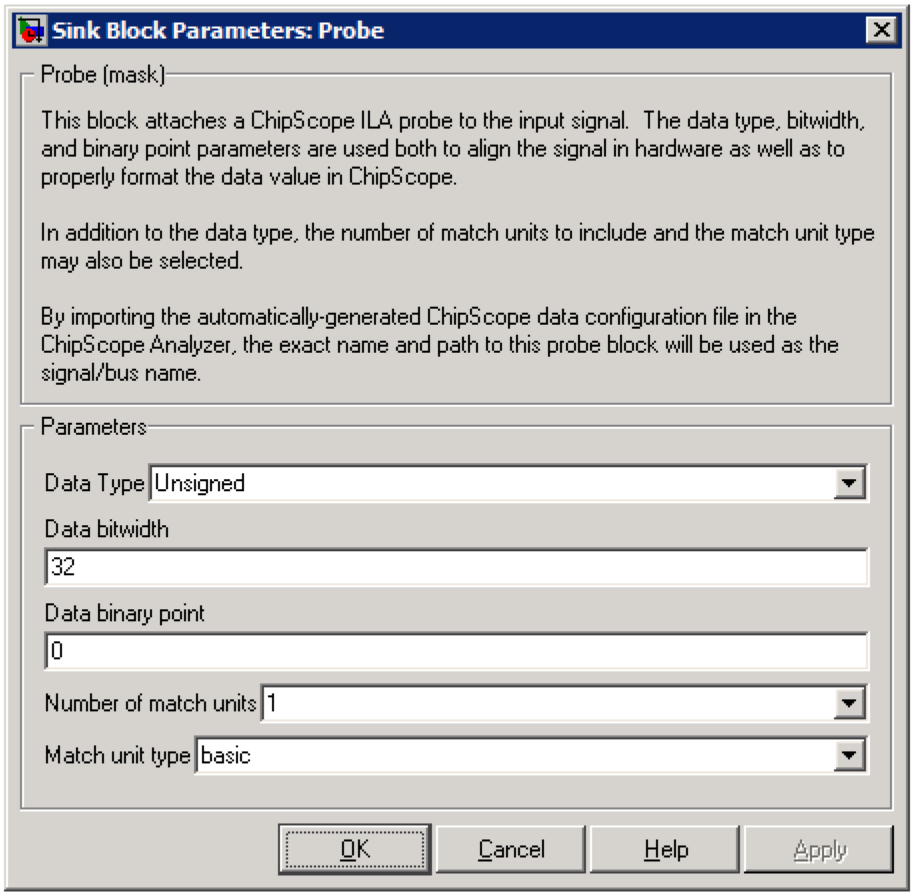


Figure 61: ChipScope Probe block



Probe (mask)

This block attaches a ChipScope ILA probe to the input signal. The data type, bitwidth, and binary point parameters are used both to align the signal in hardware as well as to properly format the data value in ChipScope.

In addition to the data type, the number of match units to include and the match unit type may also be selected.

By importing the automatically-generated ChipScope data configuration file in the ChipScope Analyzer, the exact name and path to this probe block will be used as the signal/bus name.

Parameters

Data Type: Unsigned

Data bitwidth: 32

Data binary point: 0

Number of match units: 1

Match unit type: basic

Figure 62: ChipScope Probe dialog

The ChipScope Probe block will add a signal as an input to the ChipScope Integrated Logic Analyzer (ILA) core. By placing a Probe block on a signal in the System Generator portion of the design, that signal can be used as a trigger and captured within ChipScope for runtime debugging of the hardware.

When Probe blocks are used in a design, BPS will automatically generate either a ChipScope Definition and Connection (CDC) file in the XPS project directory (the directory with the same name as the base package used for the design) for Virtex 6 and earlier, or a (LTX) probes file in the same output directory as the generated bit file for Virtex 7 and later. This file contains all the naming and data type information for each probed signal, allowing ChipScope to display each value in the proper format. Within ChipScope, each signal will have the exact same name as the Probe block itself in the Simulink design.

For more detailed information on the ChipScope ILA and features such as match units, please refer to the documentation included with Xilinx ChipScope or Vivado.

Table 38: ChipScope Probe block parameters

Parameter	**Description**
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Number of match units	This setting defines the number of match units to be included for this signal in the ChipScope ILA.
Match unit type	This setting defines the type of match unit to be used for this signal in the ChipScope ILA.

Clock Override

The Clock Override block overrides the clock connected to all BPS blocks within the same subsystem, including its children. It does *not* override the clock connected to System Generator blocks, and thus, it cannot be used to clock System Generator state elements at an alternate rate.

WARNING: This block is for advanced use only; using multiple clocks in a design can complicate meeting timing. The user is responsible for crossing clock domains safely and ensuring the design will meet timing.

Clock Override block

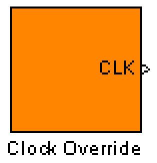
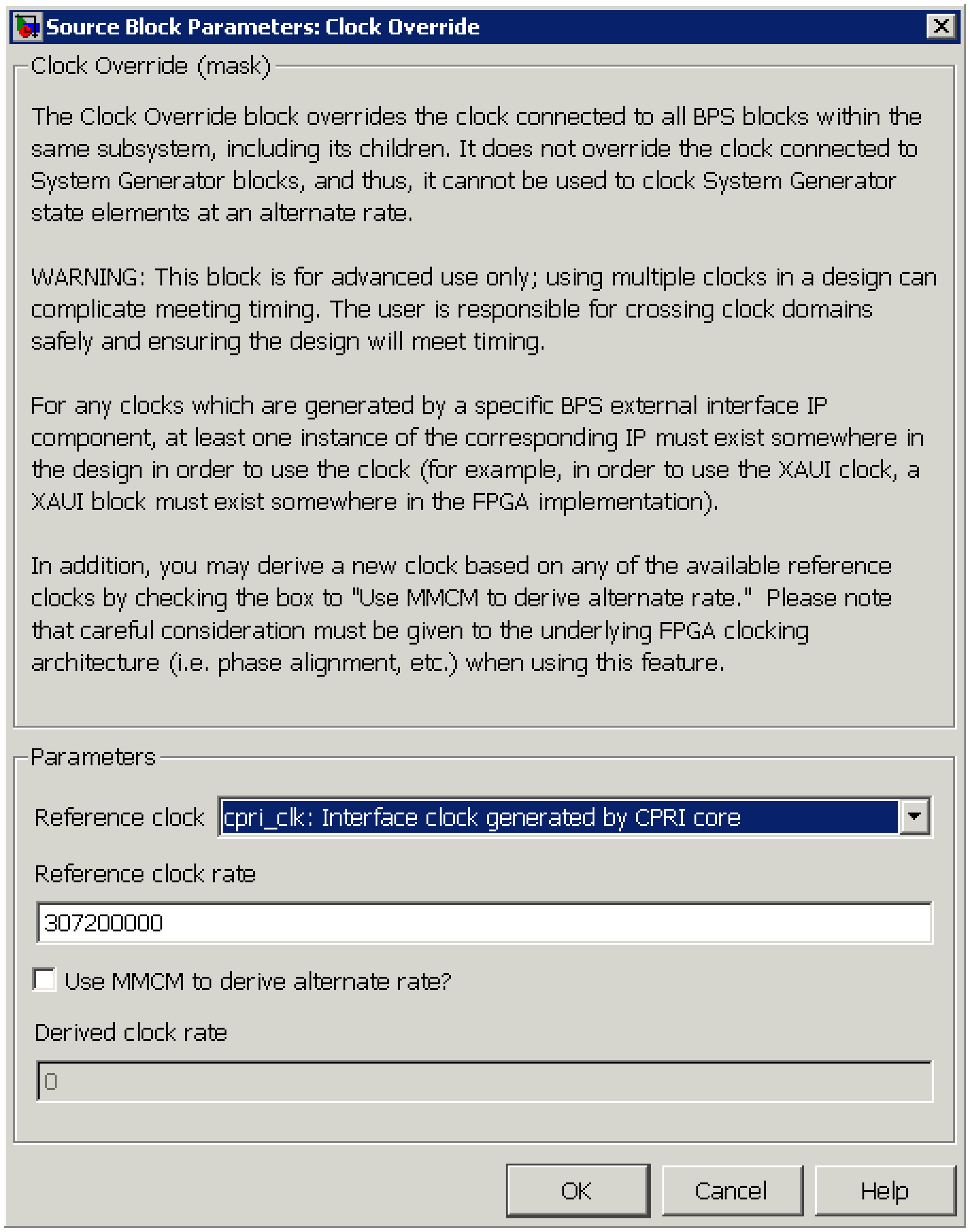


Figure 63: Clock Override block

The Clock Override block has a single output port which represents the clock signal itself which is used by the entire subsystem. Note that this signal is *not* a normal signal and cannot be used as an input to any BPS or Xilinx blocks in the design, other than imported custom IP blocks created by the IP Import Utility. This feature allows for clocks to be driven into imported IP to create the proper connectivity in the FPGA without any manual modification of the design.

For any clocks which are generated by a specific BPS external interface IP component, at least one instance of the corresponding IP must exist somewhere in the design in order to use the clock (for example, in order to use the XAUI clock, a XAUI block must exist somewhere in the FPGA implementation).

In addition, you may derive a new clock based on any of the available reference clocks by checking the box to "Use MMCM to derive alternate rate." Please note that careful consideration must be given to the underlying FPGA clocking architecture (i.e. phase alignment, etc.) when using this feature.



Source Block Parameters: Clock Override

Clock Override (mask)

The Clock Override block overrides the clock connected to all BPS blocks within the same subsystem, including its children. It does not override the clock connected to System Generator blocks, and thus, it cannot be used to clock System Generator state elements at an alternate rate.

WARNING: This block is for advanced use only; using multiple clocks in a design can complicate meeting timing. The user is responsible for crossing clock domains safely and ensuring the design will meet timing.

For any clocks which are generated by a specific BPS external interface IP component, at least one instance of the corresponding IP must exist somewhere in the design in order to use the clock (for example, in order to use the XAUI clock, a XAUI block must exist somewhere in the FPGA implementation).

In addition, you may derive a new clock based on any of the available reference clocks by checking the box to "Use MMCM to derive alternate rate." Please note that careful consideration must be given to the underlying FPGA clocking architecture (i.e. phase alignment, etc.) when using this feature.

Parameters

Reference clock: cpri_clk: Interface clock generated by CPRI core

Reference clock rate: 307200000

☐ Use MMCM to derive alternate rate?

Derived clock rate: 0

Figure 64: Clock Override dialog

Table 39: Platform Configuration block parameters

Parameter	**Description**
Reference clock	The clock used as a reference for any generated clock. If no MMCM is used to generate a new rate, the reference clock is used directly. If the clock is sourced from IP, that IP block must be instantiated somewhere in the FPGA design; it is not required to be in the same subsystem.
Reference clock rate	The frequency of the reference clock. This is usually fixed, but some cores may generate a user-specified rate (e.g. CPRI); where applicable, that frequency must be entered here.
Use MMCM to derive alt rate?	Check this box to use an additional MMCM on the FPGA to generate a new clock frequency based on the reference clock specified above.
Derived clock rate	Enter the desired clock frequency (in Hz) for the clock generated by the MMCM. This is only available if the “Use MMCM to derive alt rate?” box is checked.

CPRI

BPS provides support for the Common Public Radio Interface (CPRI) standard, up to the highest currently available data rate of 6.1440 Gbps. The hardware implementation generated by this version of BPS is based on the Xilinx v3.2 LogiCORE IP CPRI component, and is targeted for use with the BEEcube PCIE101 PCI-Express to SFP+ breakout board.

CPRI block for BEE4

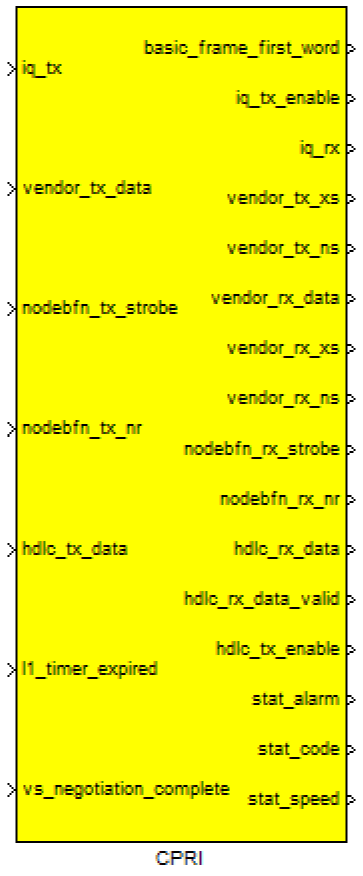
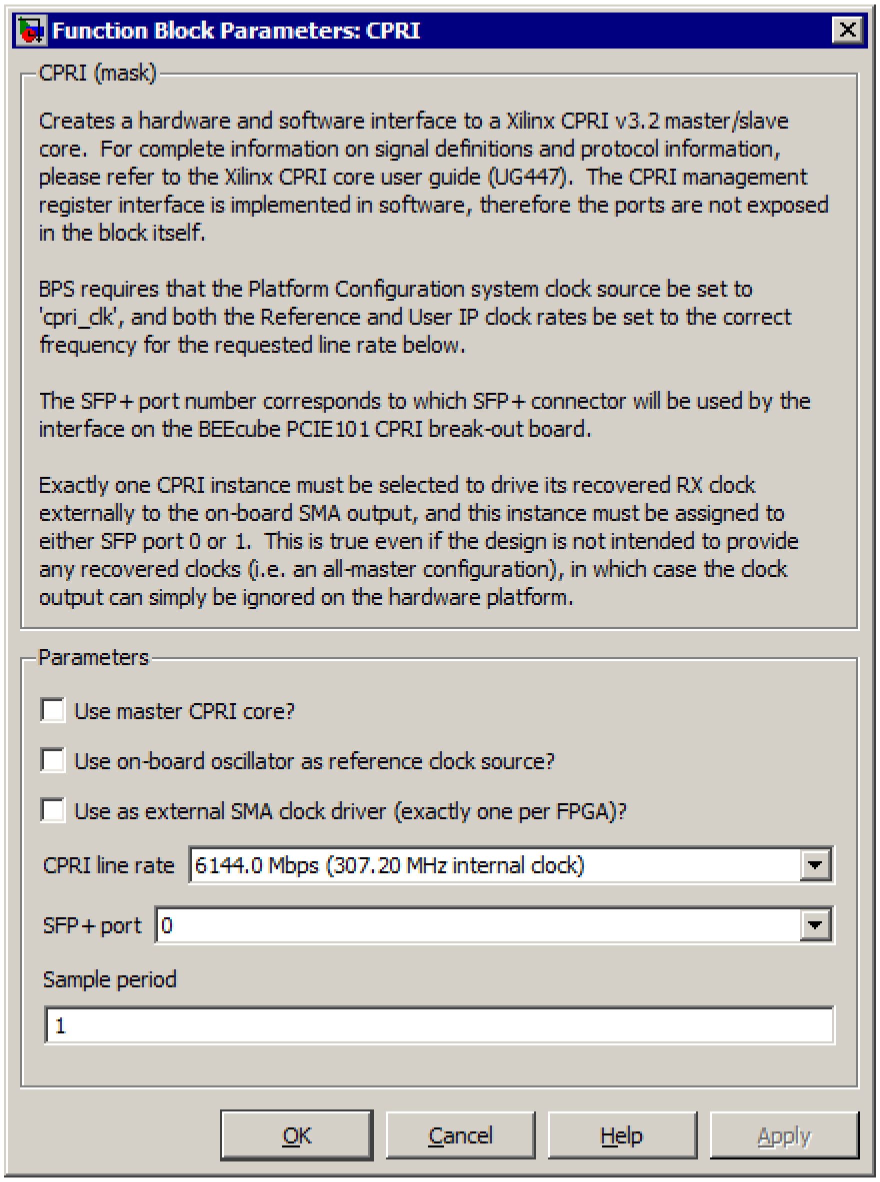


Figure 65: CPRI block

The CPRI block for BEE4 will add a hardware and software interface for a Xilinx v3.2 LogiCORE IP CPRI instance to the hardware implementation. BPS supports up to a total of 6 CPRI instances per FPGA design, one for each SFP+ port available on the PCIE101 PCI-Express to SFP+ breakout board. Each port can be configured to operate as either a CPRI Master or a CPRI Slave, and can also choose as a reference clock either the attached 307.2 MHz oscillator provided on the PCIE101 board or the on-board GTX SMA input clock on the BEE4 main board. The latter reference clock option allows for the recovered clock from a CPRI Slave instance to be used as the reference clock for any other CPRI instance in the same BEE4 module. Note that it is the responsibility of the user to ensure that the clocks are selected and connected correctly on the hardware platform, as this is application-dependent.

A description of each of the parameters available for the CPRI block are described below in the configuration dialog figure and parameter table. The ports of the block themselves correspond exactly to the ports defined by the Xilinx LogiCORE CPRI IP, and are not altered or modified by the BPS hardware instance. For a full description of each of the hardware ports and the required communication protocols, please refer to the *Xilinx LogiCORE IP CPRI v3.2 User Guide* (UG447). The Management Interface for the CPRI core, is not included in the ports of the block, as this interface is handled in software by the BPS CPRI core driver. Information on how to use the management interface function calls is included in the "CPRI block NectarOS functions" subsection following the parameter table.



CPRI (mask)

Creates a hardware and software interface to a Xilinx CPRI v3.2 master/slave core. For complete information on signal definitions and protocol information, please refer to the Xilinx CPRI core user guide (UG447). The CPRI management register interface is implemented in software, therefore the ports are not exposed in the block itself.

BPS requires that the Platform Configuration system clock source be set to 'cpri_clk', and both the Reference and User IP clock rates be set to the correct frequency for the requested line rate below.

The SFP+ port number corresponds to which SFP+ connector will be used by the interface on the BEEcube PCIE101 CPRI break-out board.

Exactly one CPRI instance must be selected to drive its recovered RX clock externally to the on-board SMA output, and this instance must be assigned to either SFP port 0 or 1. This is true even if the design is not intended to provide any recovered clocks (i.e. an all-master configuration), in which case the clock output can simply be ignored on the hardware platform.

Parameters

- [ ] Use master CPRI core?
- [ ] Use on-board oscillator as reference clock source?
- [ ] Use as external SMA clock driver (exactly one per FPGA)?

CPRI line rate: 6144.0 Mbps (307.20 MHz internal clock)

SFP+ port: 0

Sample period: 1

Figure 66: CPRI dialog

Table 40: CPRI block parameters

Parameter	Description
Use master CPRI core?	When checked, BPS will use a CPRI Master core instance in the hardware implementation. When unchecked, BPS will use a CPRI Slave core instance.
Use on-board oscillator as reference clock source?	When checked, BPS will connect the GTX reference clock input of the CPRI core to the 307.2 MHz oscillator present on the PCIE101 board. When unchecked, BPS will connect the GTX reference clock input of the CPRI core to the SMA GTX clock input on the BEE4 main board. Note that this is independent of the master/slave option above, which allows for a "master hop" implementation.
Use as external SMA clock driver (exactly one per FPGA)?	When checked, BPS will use the recovered RX clock from this CPRI instance as the clock to be driven out onto the BEE4 global clock tree. Exactly one CPRI instance per FPGA must have this box checked, even if the clock is not being used on the physical hardware. The SFP+ port for the core on which this option is selected must be either 0 or 1 due to architectural requirements on the Virtex-6 FPGA.
CPRI line rate	The CPRI line rate at which the core will operate is selected from the drop-down list. The CPRI core is forced into a single line mode via Management Register 0x0D (Line Rate Capability). This is necessary since all CPRI cores in the system share the same logic clock and therefore cannot auto-select their rate independently. Note that the system logic clock frequency is derived from the CPRI line rate. The User IP Clock Rate parameter in the Platform Configuration block must be set to match this frequency.
SFP+ port	This parameter defines which SFP+ port on the PCIE101 board will be mapped to this CPRI instance.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

CPRI block NectarOS functions

BPS adds the following functions to NectarOS whenever a CPRI block is used in a design. The source code for these functions is contained in the *drivers/xps_cpri* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`init_pcie101`

The `init_pcie101` function initializes all the GPIO devices on the PCIE101 board to proper pin directions and output levels for normal operation. The PLL is also set into the correct mode for a 307.2 MHz input.

Repeated function calls

(none)

Commands

`pcie101_iic_write <device> <address> <value>`

The `pcie101_iic_write` function will execute a raw IIC write command to the PCIE101 IIC bus. This function is only intended to be used in advanced applications where the default configuration must be modified, or for system debugging.

`pcie101_iic_read <device> <address>`

The `pcie101_iic_read` function will execute a raw IIC read command to the PCIE101 IIC bus. This function is only intended to be used in advanced applications where the default configuration has been modified, or for system debugging.

`pcie101_switch_set <channel_mask>`

The `pcie101_switch_set` function will force the IIC switches on the PCIE101 board into a specific state. The switches are typically controlled automatically by other commands which operate directly on the SFP+ ports or PLL, and therefore this function is only intended to be used in advanced applications or for system debugging.

`pcie101_iic_list`

The `pcie101_iic_list` function will list which devices on the PCIE101 are visible by sending a simple read command to each device. Note that downstream devices behind the IIC switches will only be seen when their corresponding switch channel is activated, and therefore it is normal for some devices to not be detected. This function is only intended be to used in advanced applications for system debugging.

```
sfp_set_mode <sfp_port (0-5)> <tx_disable (0|1) <rs1_rs0_val (0-3)>
```

The `sfp_set_mode` function will set the output levels of the standard SFP+ signals TX_DISABLE and RS0/RS1 for a given SFP+ port on the PCIE101 board. Note that by default, the initialization function automatically sets up all ports to be active and to operate properly at all supported line rates, so no intervention by the user is required for normal operation.

```
sfp_get_status <sfp_port (0-5)>
```

The `sfp_get_status` function will report the current status signal levels for the given SFP+ port on the PCIE101 board. This includes the standard SFP+ signals TX_FAULT, MOD_ABS, and RX_LOS. Note that LEDs are also present on the PCIE101 board for each of these signals which can be observed visually on the hardware.

```
pll_set_mode <frqsel_str> <bwsel_str> <frqtbl_str>
```

The `pll_set_mode` function will set the levels of the relevant control pins of the PLL on the PCIE101 board and perform a reset of the device. The arguments should follow the exact format used in Table 8 of the Silicon Labs Si5317 product datasheet. By default, the initialization function automatically sets the PLL into the proper mode for operation at 307.2 MHz, which is the required reference clock frequency for the CPRI implementation used by BPS. Therefore, no intervention is required by the user for normal operation.

```
pll_get_status
```

The `pll_get_status` function reports the current state of the PLL status signals LOL and LOS. Note that LEDs are also present on the PCIE101 board for each of these signals which can be observed visually on the hardware.

CPRI EDK API functions

The underlying hardware/software interface for the CPRI component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the CPRI core. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE4 base package. Include the header *xcpri.h* for access to the API.

Data Structures

There are three data structures used by several of the CPRI EDK driver function calls which correspond to all the software-accessible control and status bits contained in the hardware interface logic. Each of these data structures are described in the following tables.

Table 41: `XCpri_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base address of hardware instance
`sfp_port`	Unsigned 8 bit	SFP port location on PCIE101 board
`is_clk_drv`	Unsigned 8 bit	Indicates if instance is driving out recovered clock
`line_rate`	Unsigned 8 bit	CPRI core line rate setting (1, 2, 3, 4, 5, or 6)

Table 42: `XCpri_Status` Structure Contents

Variable name	Type	Definition
`reset`	Unsigned 8 bit	State of device reset signal
`ext_clk_sel`	Unsigned 8 bit	External clock driver select value (0: recovered clock, 1: 307.2MHz reference)
`ref_clk_ok`	Unsigned 8 bit	External clock OK signal (must be driven by software to indicate external PLL is locked)
`core_alarm`	Unsigned 8 bit	CPRI core status alarm signal
`core_code`	Unsigned 8 bit	CPRI core status code signal
`core_speed`	Unsigned 8 bit	CPRI core status speed signal

Functions

```
XCpri_Config *
XCpri_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor bus. The return value of this function should be used as the `XCpri_Handle` argument required by other driver routines.

```
XStatus
XCpri_Reset(XCpri_Handle handle);
```

Asserts the reset signal to the CPRI core for a short period of time, and then releases the reset signal to resume normal operation. The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XCpri_GetStatus (XCpri_Handle handle, XCpri_Status *status);
```

Fills the members of CPRI status structure referenced by the pointer `status` with the current status of the CPRI core. The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
u8
XCpri_IsExtClkSrc (XCpri_Handle handle);
```

Returns a boolean value which indicates whether the given CPRI instance contains the necessary output logic and is driving its recovered RX clock out of the FPGA. The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XCpri_SetExtClkSel (XCpri_Handle handle, Xuint8 value);
```

Expects a value of either 0 or 1, and sets the state of the mux for the external clock driver. A value of 0 uses the recovered RX clock from the CPRI core itself, and a value of 1 uses the 307.2 MHz reference clock directly. The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XCpri_SetRefClkOk (XCpri_Handle handle, Xuint8 value);
```

Expects a value of either 0 or 1, and sets the state of the external clock OK signal driven into the CPRI wrapper logic. This can be used by software to ensure that the external PLL is locked before the CPRI core will begin normal operation. The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XCpri_MgmntRegWrite (XCpri_Handle handle, Xuint8 addr, Xuint32 data);
```

Performs a write to the Management Interface on the CPRI core. The argument `addr` should be the address of the management register to write, and the argument `data` should be the value to be written. For information on the management register map of the CPRI core, please refer to the *Xilinx LogiCORE IP CPRI v3.2 User Guide* (UG447). The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XCpri_MgmntRegRead (XCpri_Handle handle, Xuint8 addr, Xuint32 *data);
```

Performs a read from the Management Interface on the CPRI core. The argument `addr` should be the address of the management register to be read, and the current register value will be stored in the location reference by the pointer `data`. For information on the management register map of the CPRI core, please refer to the *Xilinx LogiCORE IP CPRI v3.2 User Guide* (UG447). The argument `handle` should be a configuration structure pointer as returned by the `XCpri_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

DDR3 MIG

BEE4 and miniBEE

BPS supports the use of DDR3 DRAM memory on BEE4 and miniBEE via the Xilinx MIG (Memory Interface Generator) UI (User Interface). The controller is guaranteed to run at memory clock speeds of at least 400MHz (DDR800), with a peak throughput of 6.4GB/s to and from the DIMM. The physical memory controller only supports 2GB and 4GB single rank DIMMs, although dual or quad rank DIMMs may safely be used, as all unused ranks and their control signals are held inactive.

There are two different interface protocols available from user logic. The first provides a direct connection to the MIG UI for maximum performance and minimal logic utilization in the block itself. However, due to the direct interface to the memory controller and the absence of any clock domain crossing logic, the system clock is forced to run synchronously to the memory clock. The second interface protocol features a custom asynchronous FIFO between the user logic and the memory controller which allows the system clock to run at any frequency, independent of the memory clock.

Only one DDR3 MIG block may be used per DIMM, and the user must provide their own arbitration logic in order to share the interface between multiple sources in the hardware design. The DDR3 MIG block does not map any of the DRAM address space to the embedded processor, and therefore is only accessible from within the hardware design.

DDR3 MIG block for BEE4 and miniBEE4

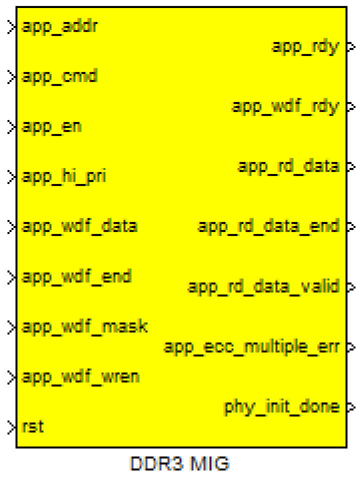


Figure 67: DDR3 MIG block

The DDR3 MIG block provides a direct interface to the UI (User Interface) layer of the Xilinx DDR3 memory controller. The UI is a simplified wrapper around the Native Interface which provides a FIFO-like command port that accepts read and write commands to physical memory addresses with no need for understanding of the low-level details of the memory configuration. For detailed information on the UI protocol, including the complete port specification, please refer to the Xilinx Virtex-6 Memory Interfaces user guide (UG406).

The physical memory controller will run at a speed directly related to the system clock of the hardware design. This is affected by the Memory Clock Ratio parameter in the block configuration dialog, which can be set to a ratio of either 1:1 or 2:1 (see parameter information below). It is the responsibility of the user to make sure that the system clock frequency is compatible with the DIMM components in their system, or else the physical layer will not initialize properly (as indicated by the `phy_init_done` signal provided by the block). For example, a 2:1 ratio with a system clock frequency of 200MHz will yield a 400MHz memory clock (DDR800 bus speed).

An example illustrating the use of the DDR3 MIG interface can be found in the included demonstration models.

DDR3 FIFO block for BEE4 and miniBEE

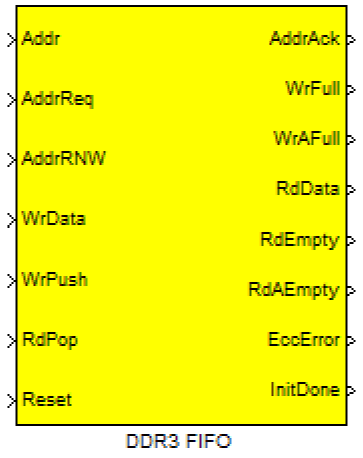


Figure 68: DDR3 FIFO block

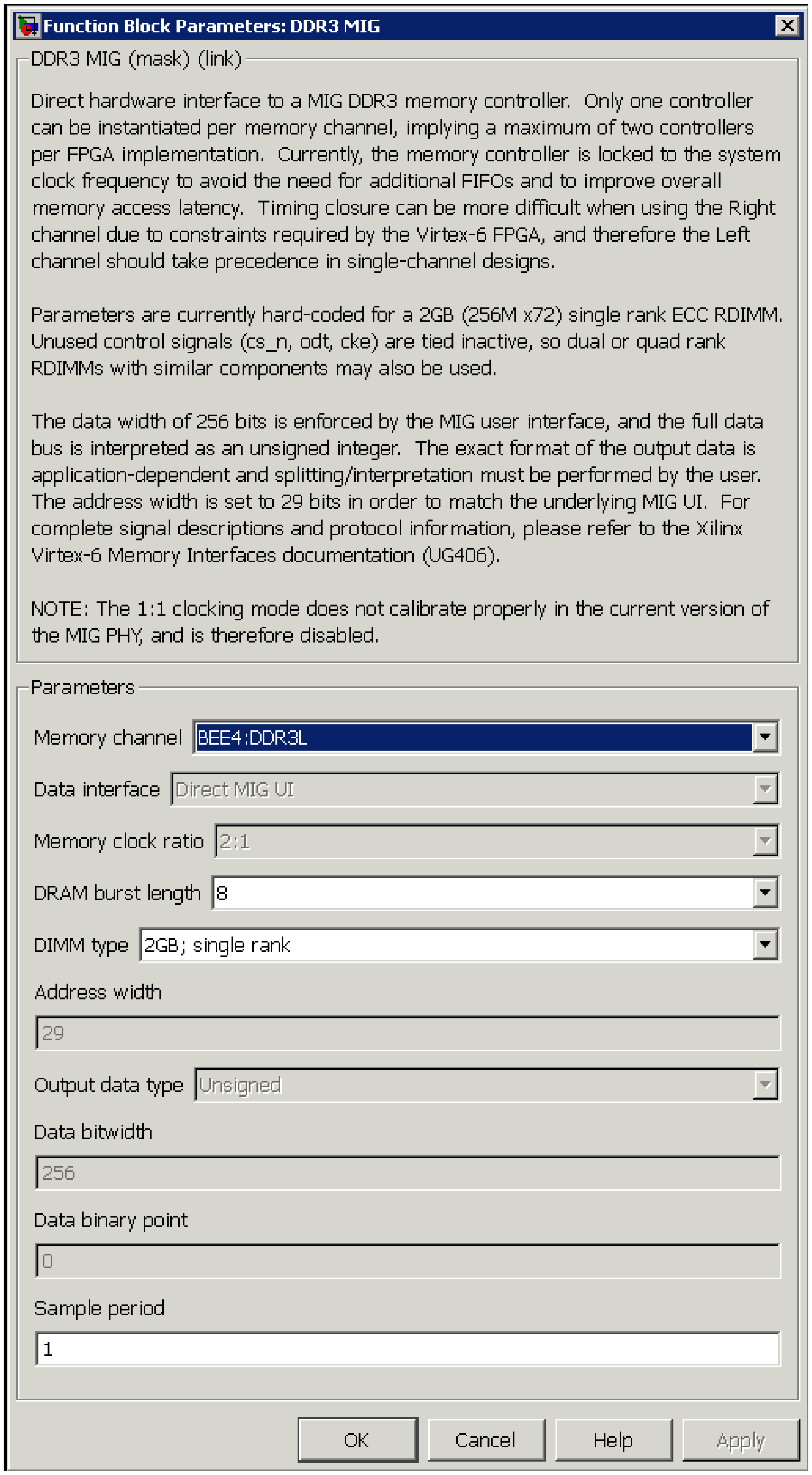
The DDR3 FIFO block adds an asynchronous 1:2 FIFO between the user logic and the MIG memory controller, allowing the system clock to run at any arbitrary frequency independent of the memory clock. Independent FIFOs are used for address requests, write data, and read data, with flow control and DRAM commands handled automatically by the underlying logic.

The width of the data FIFOs is fixed at 128 bits, one-half the width of the native MIG data bus. The user is responsible for making sure that the number of commands issued on the address request FIFO is kept in sync with the amount of data read/written on the data FIFOs, or else a deadlock condition could potentially occur. For DDR3 memory, each transaction corresponds to one complete burst to/from the DIMM. Since the BEE4/miniBEE4 memory configuration uses a 64-bit datapath (72-bit ECC DIMMs), this implies that each burst cycle consumes 64 bits. For example, if the DRAM burst length is set to 8 cycles (the ideal setting) in the DDR3 FIFO block, each transaction consumes 512 bits of data, which equates to 4 cycles worth of FIFO data at 128 bits per cycle. Therefore, with a DRAM burst length of 8, 4 cycles of data must be pushed into the write FIFO (or popped from the read FIFO) for each address request sent.

An example illustrating the use of the DDR3 FIFO interface can be found in the included demonstration models.

Table 43: DDR3 FIFO port definitions

Port	Direction	Bit Width	Description
`Addr`	in	29 or 30	Physical DRAM address to be used for a memory transaction. Since the DIMM has a 64-bit datapath, this implies that each address location refers to one 64-bit word in memory. Note that this physical address must also be burst-aligned.
`AddrRNW`	in	1	Specifies whether a memory transaction should be a read (logic high) or write (logic low).
`AddrReq`	in	1	Issues a memory transaction defined by the values on `Addr` and `AddrRNW` into the command FIFO. Should be held high until `AddrAck` is asserted.
`AddrAck`	out	1	Indicates that the current memory transaction request has been accepted into the command FIFO.
`RdData`	out	128	Data returned by the memory controller as a result of a read transaction. Only valid when `RdEmpty` is not asserted.
`RdPop`	in	1	Pops the currently valid data from the read FIFO. Only valid when `RdEmpty` is low.
`RdEmpty` `RdAEmpty`	out	1	Indicates that the read FIFO is empty or almost empty (only one data word remaining), respectively.
`WrData`	in	128	Data to be sent to the memory controller for a write transaction.
`WrPush`	in	1	Pushes the current data value into the write FIFO. May only be asserted when `WrFull` is not high.
`WrFull` `WrAFull`	out	1	Indicates that the write FIFO is full for almost full (only one slot remaining), respectively.
`EccError`	out	1	Indicates that ECC data integrity errors have been reported by the memory controller.
`InitDone`	out	1	Indicates that the physical memory controller calibration and initialization have completed. No transactions should be issued unless this signal is high.
`Reset`	in	1	Issues a reset to the FIFO interface and memory controller. Because this causes a reset of the physical memory controller, the `InitDone` signal should be monitored following a reset. Note that some memory locations also may be overwritten or become corrupted during a controller reset.



DDR3 MIG (mask) (link)

Direct hardware interface to a MIG DDR3 memory controller. Only one controller can be instantiated per memory channel, implying a maximum of two controllers per FPGA implementation. Currently, the memory controller is locked to the system clock frequency to avoid the need for additional FIFOs and to improve overall memory access latency. Timing closure can be more difficult when using the Right channel due to constraints required by the Virtex-6 FPGA, and therefore the Left channel should take precedence in single-channel designs.

Parameters are currently hard-coded for a 2GB (256M x72) single rank ECC RDIMM. Unused control signals (cs_n, odt, cke) are tied inactive, so dual or quad rank RDIMMs with similar components may also be used.

The data width of 256 bits is enforced by the MIG user interface, and the full data bus is interpreted as an unsigned integer. The exact format of the output data is application-dependent and splitting/interpretation must be performed by the user. The address width is set to 29 bits in order to match the underlying MIG UI. For complete signal descriptions and protocol information, please refer to the Xilinx Virtex-6 Memory Interfaces documentation (UG406).

NOTE: The 1:1 clocking mode does not calibrate properly in the current version of the MIG PHY, and is therefore disabled.

Parameters

Memory channel	BEE4:DDR3L
Data interface	Direct MIG UI
Memory clock ratio	2:1
DRAM burst length	8
DIMM type	2GB; single rank
Address width	29
Output data type	Unsigned
Data bitwidth	256
Data binary point	0
Sample period	1

Figure 69: DDR3 MIG dialog

Table 44: DDR3 MIG block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Data interface	This parameter defines the port interface and protocol for connecting to the memory controller. This parameter is fixed to *Direct MIG UI* for the MIG block and *Async 1:2 FIFO* for the FIFO block.
Memory clock ratio	This parameter defines the ratio between the memory clock and the system clock. The currently released version of the physical memory controller will not calibrate in 1:1 mode, therefore this parameter is temporarily fixed to 2:1.
Burst length	This parameter sets the expected burst length of the interface. Valid values are 4 (one UI clock per DRAM burst) or 8 (two UI clocks per DRAM burst).
Address width	This parameter defines the address width of the UI. This is fixed to 29 or 30, the physical address width of the supported DIMM configuration. Note that for single-rank DIMMs, the MSB will always be zero, resulting in a usable address width of 28 or 29 bits, respectively.
Output data type	This parameter determines the inferred arithmetic type for the signal. This is forced to *Unsigned*, since the raw output of the memory bus contains multiple bytes and must be split and interpreted as desired by the user.
Data bitwidth	This value defines the total width of the signal in bits. This is forced to 256 bits, the width of the memory bus.
Data binary point	This value determines the inferred binary point position for the signal. This is forced to 0, since the raw output of the memory bus contains multiple bytes and must be split and interpreted as desired by the user.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

BEE7

BPS supports the use of DDR3 DRAM memory on BEE7 via the Xilinx MIG (Memory Interface Generator) controller. The controller is guaranteed to run at memory clock speeds of at least 667MHz (DDR1333). The physical memory controller currently only supports 8 GB, dual rank DIMMs.

For BEE7, the DDR3 FIFO block provides the only interface protocol available in hardware, a custom asynchronous FIFO between the user logic and the memory controller. This protocol allows the system clock to run at any frequency, independent of the memory clock.

In addition to the hardware interface, BPS supports a SMRI transport that allows users to transfer data to/from DRAM in Matlab. See the `bee7_dram_read` and `bee7_dram_write` commands for more information.

Only one DDR3 FIFO block may be used per DIMM, and the user must provide their own arbitration logic in order to share the interface between multiple sources in the hardware design. The DDR3 MIG block does not map any of the DRAM address space to the embedded processor, and therefore is only accessible from within the hardware design.

DRAM Configuration block for BEE7

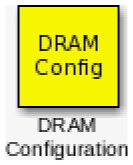
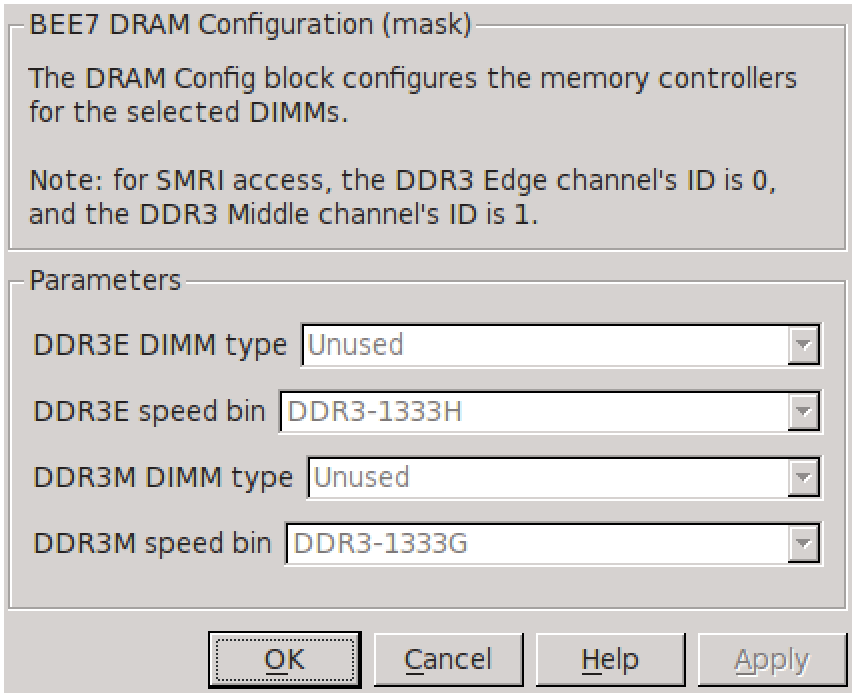


Figure 70: DRAM Configuration block

The DRAM Configuration block controls which channels are used in the design and what DIMMs populate them. This block is *required* for any design that uses DRAM (but should be omitted for designs without DRAM). Currently, only one part (an 8 GB, dual rank DIMM) is supported. Unpopulated channels must be marked *Unused*.


BEE7 DRAM Configuration (mask)

The DRAM Config block configures the memory controllers for the selected DIMMs.

Note: for SMRI access, the DDR3 Edge channel's ID is 0, and the DDR3 Middle channel's ID is 1.

Parameters

DDR3E DIMM type: Unused

DDR3E speed bin: DDR3-1333H

DDR3M DIMM type: Unused

DDR3M speed bin: DDR3-1333G

Figure 71: BEE7 DRAM Configuration dialog

Table 45: BEE7 DRAM Configuration parameters

Parameter	**Description**
DDR3E DIMM type	This parameter indicates which kind of DIMM, if any, is in the DDR3 Edge connector on the board. The only options currently supported are *Unused* and *MT18KDF1G72PDF-1G4*.
DDR3E speed bin	This parameter indicates the speed bin associated with the DIMM in the DDR3 Edge connector.
DDR3M DIMM type	This parameter indicates which kind of DIMM, if any, is in the DDR3 Middle connector on the board. The only options currently supported are *Unused* and *MT18KDF1G72PDF-1G4*.
DDR3M speed bin	This parameter indicates the speed bin associated with the DIMM in the DDR3 Middle connector.

DDR3 FIFO block for BEE7

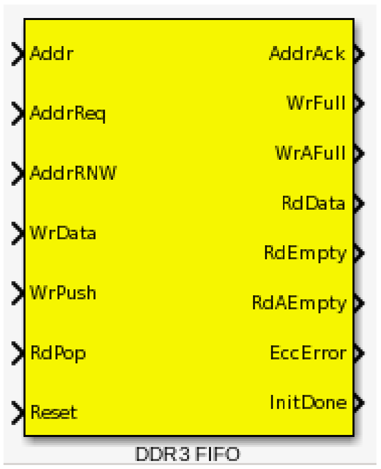


Figure 72: DDR3 FIFO block for BEE7

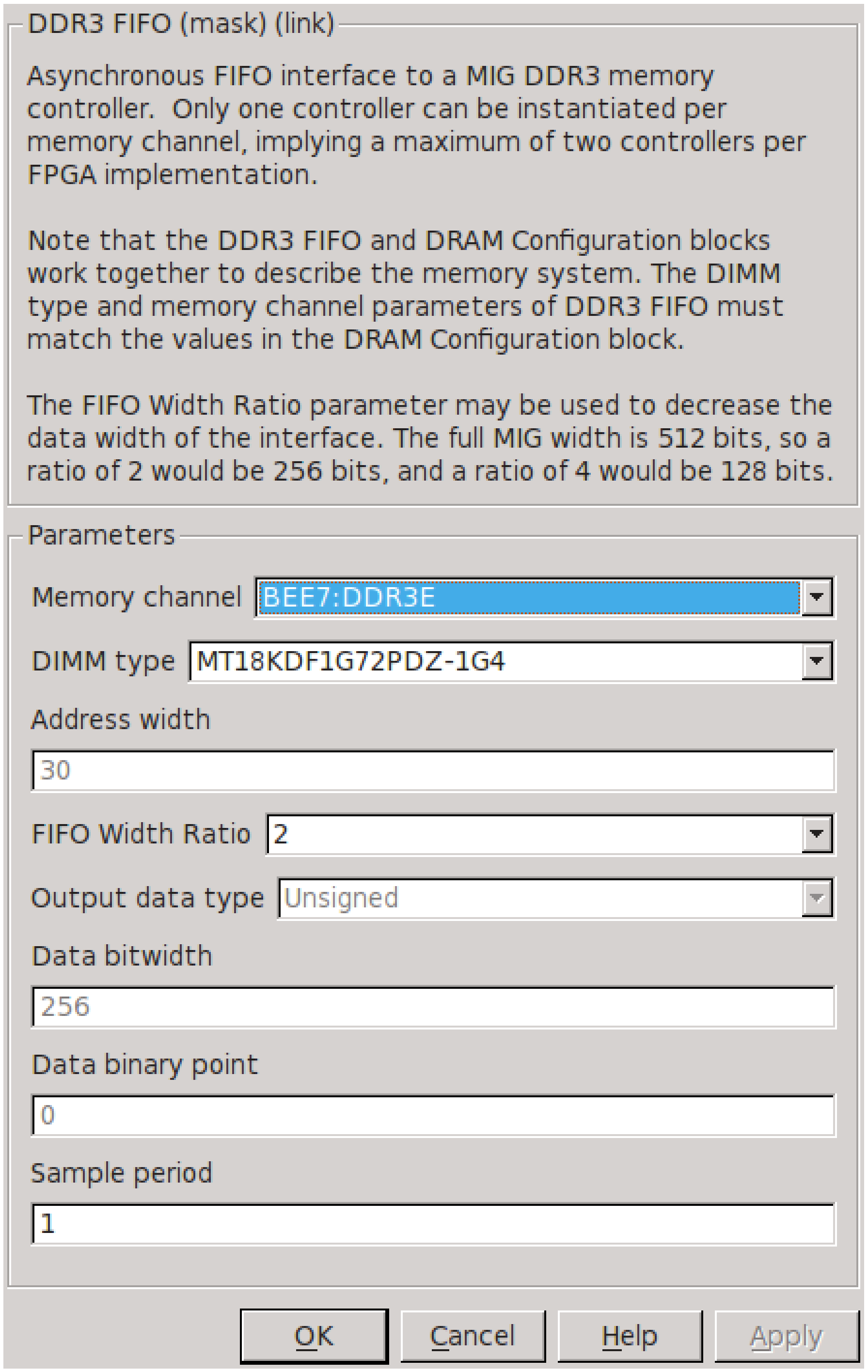
The DDR3 FIFO block adds an asynchronous 1:2 or 1:4 FIFO between the user logic and the MIG memory controller, allowing the system clock to run at any arbitrary frequency independent of the memory clock. Independent FIFOs are used for address requests, write data, and read data, with flow control and DRAM commands handled automatically by the underlying logic.

The width of the data FIFOs is 128 bits or 256 bits, depending on the FIFO width ratio selected. The user is responsible for making sure that the number of commands issued on the address request FIFO is kept in sync with the amount of data read/written on the data FIFOs, or else a deadlock condition could potentially occur. For DDR3 memory, each transaction corresponds to one complete burst to/from the DIMM. Since the BEE7 memory configuration uses a 64-bit datapath (72-bit ECC DIMMs), this implies that each burst cycle consumes 64 bits. For example, since the DRAM burst length is 8 cycles, each transaction consumes 512 bits of data, which equates to 4 cycles worth of FIFO data at 128 bits per cycle (and 2 cycles worth at 256 bits per cycle). Therefore, 4 cycles of data must be pushed into the write FIFO (or popped from the read FIFO) for each address request sent.

An example illustrating the use of the DDR3 FIFO interface can be found in the included demonstration models.

Table 46: DDR3 FIFO port definitions

Port	Direction	Bit Width	Description
`Addr`	in	29, 30	Physical DRAM address to be used for a memory transaction. Since the DIMM has a 64-bit datapath, this implies that each address location refers to one 64-bit word in memory. Note that this physical address must also be burst-aligned.
`AddrRNW`	in	1	Specifies whether a memory transaction should be a read (logic high) or write (logic low).
`AddrReq`	in	1	Issues a memory transaction defined by the values on `Addr` and `AddrRNW` into the command FIFO. Should be held high until `AddrAck` is asserted.
`AddrAck`	out	1	Indicates that the current memory transaction request has been accepted into the command FIFO.
`RdData`	out	128, 256	Data returned by the memory controller as a result of a read transaction. Only valid when `RdEmpty` is not asserted.
`RdPop`	in	1	Pops the currently valid data from the read FIFO. Only valid when `RdEmpty` is low.
`RdEmpty` `RdAEmpty`	out	1	Indicates that the read FIFO is empty or almost empty (only one data word remaining), respectively.
`WrData`	in	128, 256	Data to be sent to the memory controller for a write transaction.
`WrPush`	in	1	Pushes the current data value into the write FIFO. May only be asserted when `WrFull` is not high.
`WrFull` `WrAFull`	out	1	Indicates that the write FIFO is full for almost full (only one slot remaining), respectively.
`EccError`	out	1	Indicates that ECC data integrity errors have been reported by the memory controller.
`InitDone`	out	1	Indicates that the physical memory controller calibration and initialization have completed. No transactions should be issued unless this signal is high.
`Reset`	in	1	Issues a reset to the FIFO interface. For BEE7, this signal does not reset the memory controller.


DDR3 FIFO (mask) (link)

Asynchronous FIFO interface to a MIG DDR3 memory controller. Only one controller can be instantiated per memory channel, implying a maximum of two controllers per FPGA implementation.

Note that the DDR3 FIFO and DRAM Configuration blocks work together to describe the memory system. The DIMM type and memory channel parameters of DDR3 FIFO must match the values in the DRAM Configuration block.

The FIFO Width Ratio parameter may be used to decrease the data width of the interface. The full MIG width is 512 bits, so a ratio of 2 would be 256 bits, and a ratio of 4 would be 128 bits.

Parameters

Memory channel: BEE7:DDR3E

DIMM type: MT18KDF1G72PDZ-1G4

Address width: 30

FIFO Width Ratio: 2

Output data type: Unsigned

Data bitwidth: 256

Data binary point: 0

Sample period: 1

Figure 73: BEE7 DDR3 FIFO dialog

Table 47: BEE7 DDR3 FIFO block parameters

Parameter	Description
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
DIMM type	This parameter defines the type of DIMM plugged into the slot. This parameter should match the same in the DRAM Configuration block.
FIFO Width Ratio	This parameter sets the ratio of MIG burst length to FIFO data width. For a ratio of 2, the FIFO data width is 256 bits, and 2 cycles of write data per write address request are required. For a ratio of 4, the FIFO data width is 128 bits, and 4 cycles of write data per write address request are required.
Address width	This parameter defines the address width of the UI. This is fixed to 29 or 30, the physical address width of the supported DIMM configuration. This parameter is derived from the DIMM type parameter.
Output data type	This parameter determines the inferred arithmetic type for the signal. This is forced to *Unsigned*, since the raw output of the memory bus contains multiple bytes and must be split and interpreted as desired by the user.
Data bitwidth	This value defines the total width of the signal in bits. This is derived from the FIFO Width Ratio parameter.
Data binary point	This value determines the inferred binary point position for the signal. This is forced to 0, since the raw output of the memory bus contains multiple bytes and must be split and interpreted as desired by the user.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

DVI

BPS provides an output interface to the DVI video transceiver chip on the ML50x hardware platform. In addition to the physical hardware, BPS also generates NectarOS routines for configuring the FPGA interface and the external IC, and also provides an EDK software driver for use in custom software applications.

DVI block for ML50x

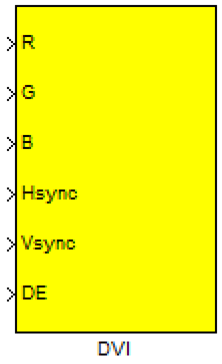
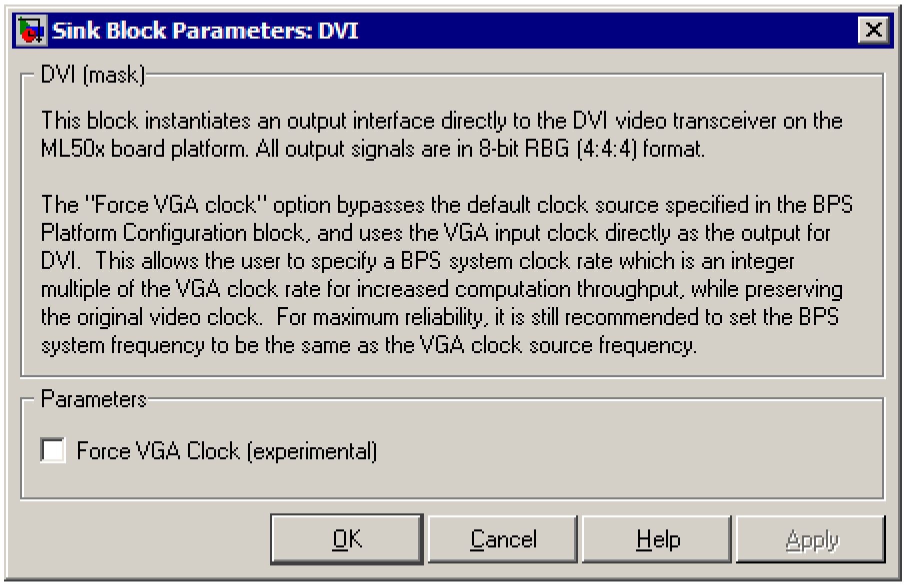


Figure 74: DVI block



This block instantiates an output interface directly to the DVI video transceiver on the ML50x board platform. All output signals are in 8-bit RBG (4:4:4) format.

The "Force VGA clock" option bypasses the default clock source specified in the BPS Platform Configuration block, and uses the VGA input clock directly as the output for DVI. This allows the user to specify a BPS system clock rate which is an integer multiple of the VGA clock rate for increased computation throughput, while preserving the original video clock. For maximum reliability, it is still recommended to set the BPS system frequency to be the same as the VGA clock source frequency.

☐ Force VGA Clock (experimental)

Figure 75: DVI dialog

The DVI block implements a Digital Visual Interface output via the Chrontel CH7301C device and the DVI connector on the ML50x board. The block functions as a data sink in a Simulink model. The video data comes in to the block as three 8-bit buses represented as RGB with a 4:4:4 format. The DVI block uses the BPS system clock by default, but if you are also using a VGA input block in your design you may force the DVI block to use the VGA input clock instead. In this case you must make sure that *video_fx* is an integer multiple of *vga_clk*.

Your system must drive the `DE` signal high when valid video data is available on the `R`, `G`, and `B` buses. Your system must also provide active-low pulses on the `Hsync` and `Vsync` inputs at intervals appropriate for the video format you wish to display.

The DVI hardware is configured through an I^{2}C bus that communicates with the BPS embedded system. BEEcube provides API routines that simplify initializing and configuring the DVI hardware for different video display formats. The DVI API is described later in this section.

Table 48: DVI block parameters

Parameter	**Description**
Force VGA Clock	If a VGA block is used in the design, use the VGA input clock instead of the system clock selected in the BPS platform configuration block for clocking the DVI signals.

Table 49: DVI port definitions

Port	**Direction**	**Bit Width**	**Description**
`R`	in	8	Red video data in 4:4:4 format.
`G`	in	8	Green video data in 4:4:4 format.
`B`	in	8	Blue video data in 4:4:4 format.
`Hsync`	in	1	Horizontal sync pulse, active low.
`Vsync`	in	1	Vertical sync pulse, active low.
`DE`	in	1	Video Data Enable must be driven high when the video data on the `R`, `G` and `B` ports are valid. Active high.

DVI block NectarOS functions

The DVI block includes a set of utility functions that enable the user to control the DVI block directly from a NectarOS shell window. BPS adds the following functions to NectarOS whenever the DVI block is used in a design. The source code for these functions are contained in the *drivers/xps_dvi* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

```
dvi_init
```

The `dvi_init` function takes no arguments. It initializes the DVI output device to operate in RGB 4:4:4 output mode.

Repeated function calls

(none)

Commands

```
writevideoiic <IIC device id> <prom_address> <value>
```

The `writevideoiic` command takes the DVI device I^2C ID, register address, and write data value as unsigned 8 bit arguments.

The DVI device I^2C ID is always 0xEC on the Xilinx ML50x evaluation platform boards.

Refer to Chrontel *CH7301C Data Sheet* (201-0000-056 Rev. 1.32, 5/24/2005), *AN-41* (206-0000-041 Rev. 1.3, 5/15/2003), and *AN-52* (206-0000-052 Rev 1.1, 4/28/2002) for a description of the CH7301C device register addresses and function of the data values.

DVI block EDK API functions

The EDK API used by the DVI block is shared with other I^2C devices. It encapsulates common I^2C device register write operations as C functions that can be used in embedded application programs. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any ML50x base package. Include the header *xps_iic_wo.h* for access to the API.

Functions

The following EDK function is used by the DVI block:

```
XStatus
iic_wo_write(void * baseaddr_p,
             u8 dev_id,
             u8 address,
             u8 val);
```

Sets the I^2C device `dev_id` configuration register `address` to `val`. The pointer `baseaddr_p` should be the base address of the I^2C controller on the PLB bus.

The DVI I^2C `dev_id` is 0xEC for Xilinx ML50x emulation platform boards.

Refer to Chrontel *CH7301C Data Sheet* (201-0000-056 Rev. 1.32, 5/24/2005), *AN-41* (206-0000-041 Rev. 1.3, 5/15/2003), and *AN-52* (206-0000-052 Rev 1.1, 4/28/2002) for a description of the CH7301C device register address `address` and function of the data `val`.

Ethernet (LWIP)

BPS can generate full-featured networking support into an FPGA implementation, which includes the underlying hardware components necessary for Ethernet connectivity as well as the LWIP lightweight TCP/IP stack. MAC addresses for the Ethernet interface are read from the on-board EEPROM for all supported hardware platforms, and IP address information is assigned via DHCP. In addition, the embedded software will also automatically launch a TCP service which can be used to dynamically perform data transfer operations to and from any on-FPGA device at runtime.

Ethernet Interface block

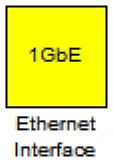
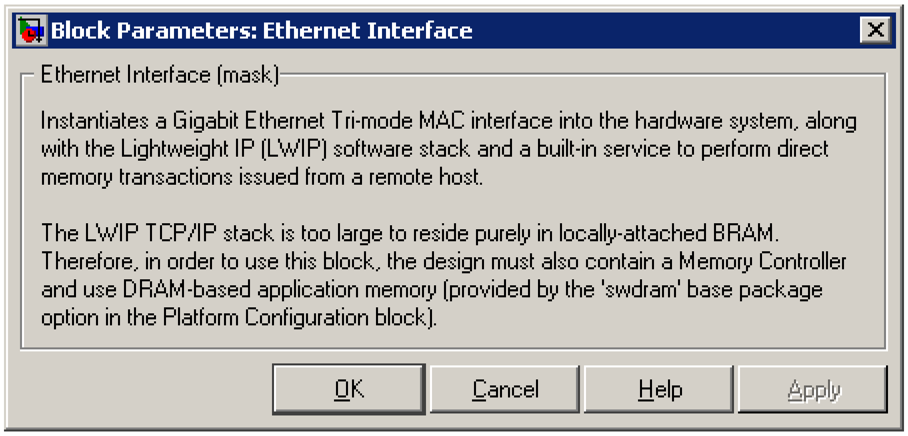


Figure 76: Ethernet Interface block



Block Parameters: Ethernet Interface

Ethernet Interface (mask)

Instantiates a Gigabit Ethernet Tri-mode MAC interface into the hardware system, along with the Lightweight IP (LWIP) software stack and a built-in service to perform direct memory transactions issued from a remote host.

The LWIP TCP/IP stack is too large to reside purely in locally-attached BRAM. Therefore, in order to use this block, the design must also contain a Memory Controller and use DRAM-based application memory (provided by the 'swdram' base package option in the Platform Configuration block).

Figure 77: Ethernet Interface dialog

The Ethernet Interface block instantiates all the hardware components necessary for Ethernet connectivity, which includes the Virtex-5 tri-mode Ethernet hard MAC, an interrupt controller, a timer, and a Soft DMA (SDMA) port directly to the on-chip multi-port memory controller (MPMC). All hardware parameters are automatically determined by BPS, and therefore there are no user-customizable parameters.

As described in the block dialog, the LWIP TCP/IP stack which is automatically configured and linked into the software application is too large to fit into local processor Block RAM. For this reason, a Memory Controller block must be used in any design that requires Ethernet support. In addition, the Platform Configuration block must be set to use the `swdram` base package configuration, which will locate all application memory segments into DRAM rather than BRAM. If either of these two requirements are not met, BPS will report an error during DRC.

Ethernet NectarOS functions

BPS adds the following functions to NectarOS whenever the Ethernet Interface block is used in a design. The source code for these functions are contained in the *drivers/xps_ethernet* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`eth_init`

The `eth_init` function performs all the initialization routines necessary to set up the Ethernet hardware components, initialize the LWIP TCP/IP stack, request an IP address from a DHCP server, and start the TCP data transfer service. In order for this function to complete, the physical media must be connected and a DHCP server must be available and responding to requests. If either of these conditions are not met, the LWIP initialization process will wait indefinitely.

Repeated function calls

`eth_process_input`

The `eth_process_input` function simply calls the `xemacif_input` LWIP function as long as the network interface is up. This call is part of the LWIP RAW-mode API, and must be called continuously for incoming packet data to be processed by the LWIP networking stack.

Commands

`netstat`

The `netstat` command does not take any arguments, and will print out the current status of the network interface, including the IP address information and the state of the TCP data transfer service.

Ethernet remote access protocol

The embedded LWIP TCP/IP stack is used to automatically launch a data transfer service, which can accept read and write commands from a remote host. This protocol is defined in detail in Chapter 6.

Ethernet (MAC)

BPS also provides support for directly interfacing custom logic to the integrated Ethernet MAC on the Virtex-6 FPGA. This direct hardware interface offers the maximum level of performance available from the device in terms of high throughput and low latency. The current FPGA implementation forces the Ethernet hardware to run at 1000Base-T (gigabit) speed only, and it will not link at either of the lower two (10Base-T or 100Base-T) line rates. The MAC itself is instantiated with all the default parameters for the core, which place the MAC into promiscuous mode (address filter disabled) with all other advanced features (VLAN, jumbo frames, in-band FCS, etc.) inactive.

Note that direct Ethernet MAC support for the ML60x hardware platform requires an RGMII interface to the PHY device. This is different than the default mode on the ML60x platform, and therefore requires that jumper J67 be removed and placed on jumper J68 instead. For more information, please refer to UG534 (*ML605 Hardware User Guide*).

Ethernet MAC LocalLink block for BEE4, miniBEE4, and ML60x

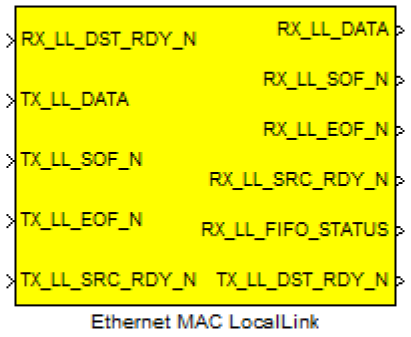


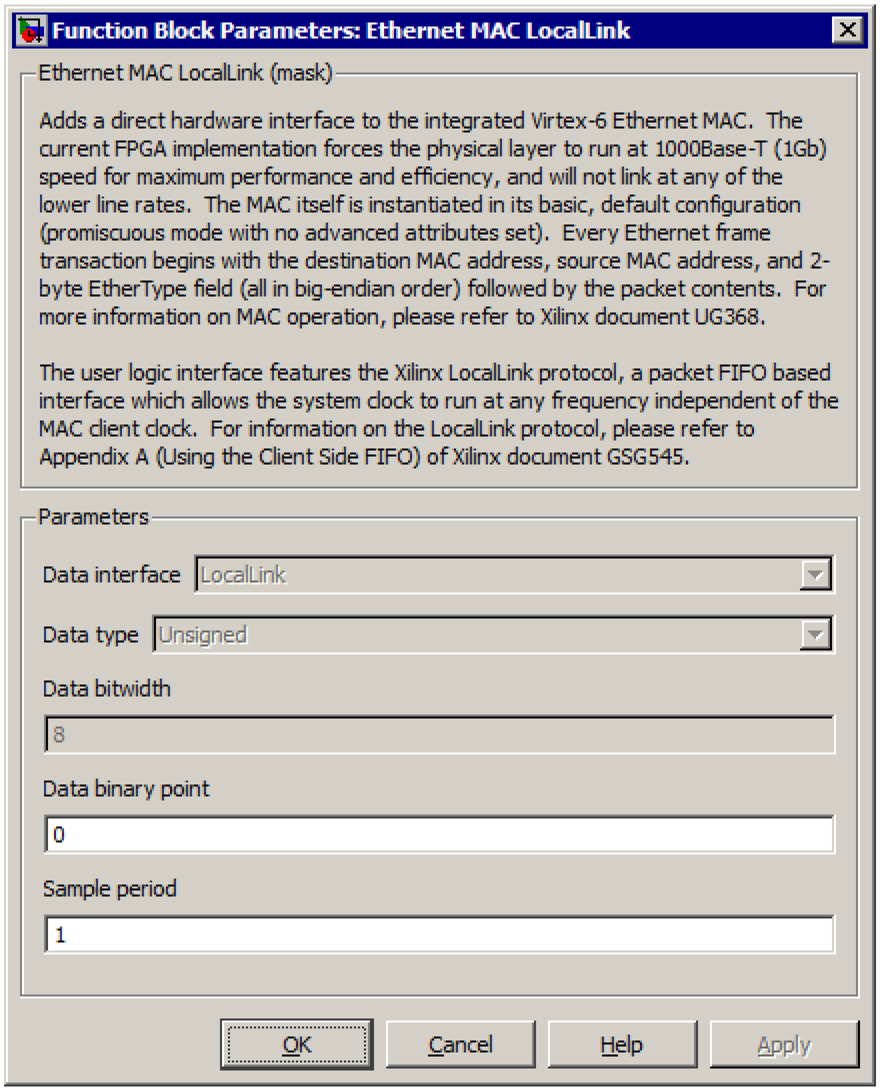
Figure 78: Ethernet MAC LocalLink block

The Ethernet MAC LocalLink block features a Xilinx LocalLink packet-based FIFO interface between the user logic and the MAC itself. This both provides a simplified protocol for communication and allows the system clock to run at any arbitrary frequency independent of the Ethernet MAC client clock, which is fixed at 125MHz in 1000Base-T mode.

There is no filtering of data returned/expected by the MAC over the LocalLink interface. Therefore, the Ethernet frame format used by the Xilinx MAC is preserved entirely at the user interface. This format requires that every frame start with the destination MAC address, source MAC address, and 16-bit EtherType field followed by the frame data contents. All values in this pseudo-header are delivered in big-endian order (most significant byte first). Any other fields in the standard 802.3 Ethernet MAC frame are automatically computed within the MAC. For more information on the Xilinx MAC operating mode and data formats, please refer to the Xilinx document UG368. More information on the LocalLink FIFO protocol can be found in Appendix A of the Xilinx document GSG545, which is included with Core Generator. A brief explanation of the LocalLink signals is included below, and a reference design using the Ethernet MAC LocalLink block is included in the set of BPS demos.

Table 50: Ethernet MAC LocalLink port definitions

Port	Direction	Bit Width	Description
TX_LL_DATA	in	8	Byte data to be transmitted as an Ethernet frame. Only valid when TX_LL_SRC_RDY_N is asserted.
TX_LL_SOF_N	in	1	Indicates the first byte of a frame. Must be asserted along with TX_LL_SRC_RDY_N and the first byte of data on TX_LL_DATA (always the MSB of the destination MAC address). Active low.
TX_LL_EOF_N	in	1	Indicates the last byte of a frame. Must be asserted along with TX_LL_SRC_RDY_N and the last byte of data on TX_LL_DATA. Active low.
TX_LL_SRC_RDY_N	in	1	Indicates that the data on TX_LL_DATA is valid and should be pushed into the transmit FIFO. Active low.
TX_LL_DST_RDY_N	out	1	Indicates that the transmit FIFO is available and can accept new data. Data is only accepted into the transmit FIFO when this signal is asserted. Active low.
RX_LL_DATA	out	8	Byte data received from an Ethernet frame. Only valid when RX_LL_SRC_RDY_N is asserted.
RX_LL_SOF_N	out	1	Indicates the first byte of a frame. Will be asserted along with RX_LL_SRC_RDY_N and the first byte of data on RX_LL_DATA (always the MSB of the destination MAC address). Active low.
RX_LL_EOF_N	out	1	Indicates the last byte of a frame. Will be asserted along with RX_LL_SRC_RDY_N and the last byte of data on RX_LL_DATA. Active low.
RX_LL_SRC_RDY_N	out	1	Indicates that the data on RX_LL_DATA is valid. Data is only removed from the FIFO if RX_LL_DST_RDY_N is also asserted. Active low.
RX_LL_DST_RDY_N	in	1	Indicates that the user logic is ready to consume data from the FIFO. Data is removed from the FIFO if RX_LL_SRC_RDY_N is also asserted. Active low.
RX_LL_FIFO_STATUS	out	4	Reports the occupancy of the receive FIFO as an integer in 1/16th increments of its total capacity.



Ethernet MAC LocalLink (mask)

Adds a direct hardware interface to the integrated Virtex-6 Ethernet MAC. The current FPGA implementation forces the physical layer to run at 1000Base-T (1Gb) speed for maximum performance and efficiency, and will not link at any of the lower line rates. The MAC itself is instantiated in its basic, default configuration (promiscuous mode with no advanced attributes set). Every Ethernet frame transaction begins with the destination MAC address, source MAC address, and 2-byte EtherType field (all in big-endian order) followed by the packet contents. For more information on MAC operation, please refer to Xilinx document UG368.

The user logic interface features the Xilinx LocalLink protocol, a packet FIFO based interface which allows the system clock to run at any frequency independent of the MAC client clock. For information on the LocalLink protocol, please refer to Appendix A (Using the Client Side FIFO) of Xilinx document GSG545.

Parameters

Data interface: LocalLink

Data type: Unsigned

Data bitwidth: 8

Data binary point: 0

Sample period: 1

Figure 79: Ethernet MAC LocalLink dialog

Table 51: Ethernet MAC LocalLink block parameters

Parameter	**Description**
Data interface	This parameter defines the user logic interface protocol to the MAC. For the Ethernet MAC LocalLink block, this is fixed to *LocalLink*.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the raw byte stream to/from the MAC.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 8 bits to match the raw byte stream to/from the MAC.
Data binary point	This value determines the inferred binary point position for all data values on the interface.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

FMC101 ADC

This block provides a direct interface to the ADC device on the BEEcube FMC101 expansion board for the BEE4, miniBEE4, and ML605 hardware platforms.

FMC101 ADC block for BEE4, miniBEE4, and ML60x

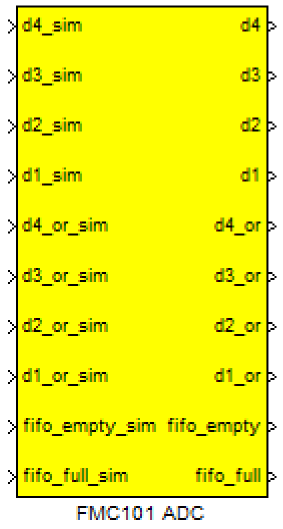
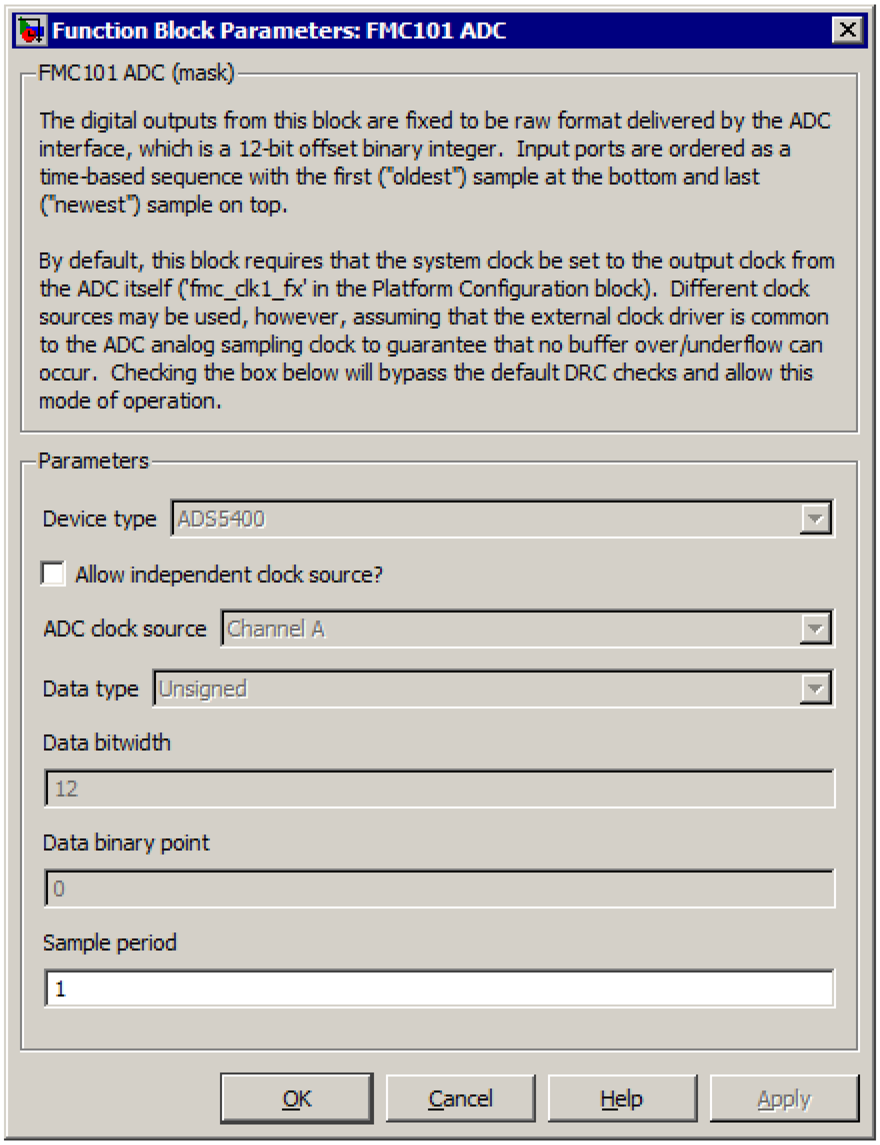


Figure 80: FMC101 ADC block

The FMC101 ADC block delivers 4 digital values per FPGA clock cycle, which were sampled by the ADC at 4x the system clock rate. The ADC device itself features 2 independent data buses which use DDR signaling, resulting in a digital clock frequency of 1/4th the sampling clock rate. Therefore, the system clock rate must be the same as the native ADC digital clock rate. For example, if the ADC sampling clock rate is 1GHz, the digital clock rate used by the 2 data buses is 250MHz. Therefore, the FPGA system clock rate must also be set to 250MHz. Note that the system clock rate must sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This typically means that the ADC digital clock itself will be used as the reference clock source for the system, but this requirement may be overridden in cases where the sampling clock is also derived synchronously by an external source.



Function Block Parameters: FMC101 ADC

FMC101 ADC (mask)

The digital outputs from this block are fixed to be raw format delivered by the ADC interface, which is a 12-bit offset binary integer. Input ports are ordered as a time-based sequence with the first ("oldest") sample at the bottom and last ("newest") sample on top.

By default, this block requires that the system clock be set to the output clock from the ADC itself ('fmc_clk1_fx' in the Platform Configuration block). Different clock sources may be used, however, assuming that the external clock driver is common to the ADC analog sampling clock to guarantee that no buffer over/underflow can occur. Checking the box below will bypass the default DRC checks and allow this mode of operation.

Parameters

Device type: ADS5400

☐ Allow independent clock source?

ADC clock source: Channel A

Data type: Unsigned

Data bitwidth: 12

Data binary point: 0

Sample period: 1

Figure 81: FMC101 ADC dialog

Table 52: FMC101 ADC block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board.
Allow independent clock source?	Overrides the usual DRC checking, which requires that the ADC digital output clock be used as the source for the rest of the system. This box should only be checked when the user has guaranteed that the system clock source is generated by the same reference clock as the ADC analog sampling clock. This mode of operation is safe, since a common reference clock will guarantee that the ADC input FIFOs will not under/overflow during normal operation.
ADC clock source	The ADC digital output clock which should be used as the source for the rest of the system. This value is currently fixed to Channel A.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 12 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 53: FMC101 ADC port definitions

Port	Direction	Bit Width	Description

`d4_sim`
`d3_sim`
`d2_sim`
`d1_sim`
`d4_or_sim`
`d3_or_sim`
`d2_or_sim`
`d1_or_sim`
`fifo_empty_sim`
`fifo_full_sim`
	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.

`d4`
`d3`
`d2`
`d1`
	out	12	Direct outputs from the ADC, representing 4 consecutive samples by the device. The samples are ordered with `d1` being the earliest and `d4` being the latest in temporal order. Data values are raw, 12-bit, offset binary integers without any digital scaling or bias.

`d4_or`
`d3_or`
`d2_or`
`d1_or`
	out	1	ADC out-of-range (OR) signal for each of the 4 corresponding data samples. Indicates if an out-of-range input caused saturation in the ADC.

`fifo_empty` `fifo_full`	out	2	Empty and full status signals for the input FIFO which is part of the underlying interface logic. These signals are provided for debugging purposes only, and are never expected to be asserted during normal operation. *Note: these may be removed from the block in a future release.*

FMC101 ADC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC101 ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc101_adc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`adc_init`

The `adc_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will automatically calibrate the input delays of the ADC interface and reset all internal FIFOs and interface logic.

Repeated function calls

(none)

Commands

`adc_reset`

The `adc_reset` function asserts a reset to the ADC interface logic, including the input FIFOs.

`adc_reg_write <reg_addr> <value>`

The `adc_reg_write` function will write an arbitrary value into any one of the ADC's internal configuration registers. The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and `value` is the 8-bit value which should be written.

`adc_test_pattern <(on|off)>`

The `adc_test_pattern` function will put the ADC into test pattern generation mode, which is currently a basic toggle pattern of alternating 1's and 0's. This command may be used to validate proper connectivity of the card and input clocks, and successful calibration of the input delays.

`adc_delay_set <data_mask> <or_mask> <idelay_val>`

The `adc_delay_set` function will override the previously calculated delay values for each input pin with the value specified in the argument `idelay_val`. The `data_mask` argument is a 32-bit value, with pins in Bus A aligned in big-endian order to the MSB of the upper halfword (0xFFF00000), and pins in Bus B aligned in big-endian order to the MSB of the lower halfword (0x0000FFF0). The `or_mask` argument is also a 32-bit value, with the OR output for Bus A in the MSB (0x80000000) followed by the OR output for Bus B (0x40000000).

`adc_calibrate_delays <usleep_duration>`

The `adc_calibrate_delays` function will re-perform an automatic calibration of the input delays for each of the ADC pins and plot the results in a table. The `usleep_duration` argument specifies the amount of time in microseconds to wait after setting each delay value before sampling the synchronization state of the pins. In order for this command to work, the ADC must have already been put into test pattern mode via the `adc_test_pattern` command.

FMC101 DAC

This block provides a direct interface to the DAC device on the BEEcube FMC101 expansion board for the BEE4, miniBEE4, and ML605 hardware platforms. Two different versions of the DAC interface are provided, one with a 4:1 multiplexing ratio, and another with an 8:1 multiplexing ratio. The two options allow for flexibility between system clock frequency and parallel resource requirements in the FPGA.

FMC101 DAC 4:1 block for BEE4, miniBEE4, and ML60x

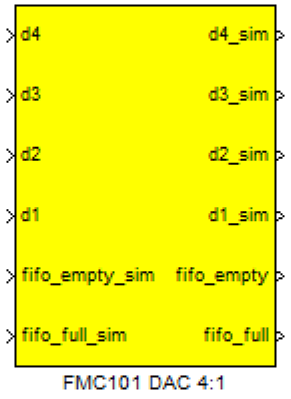
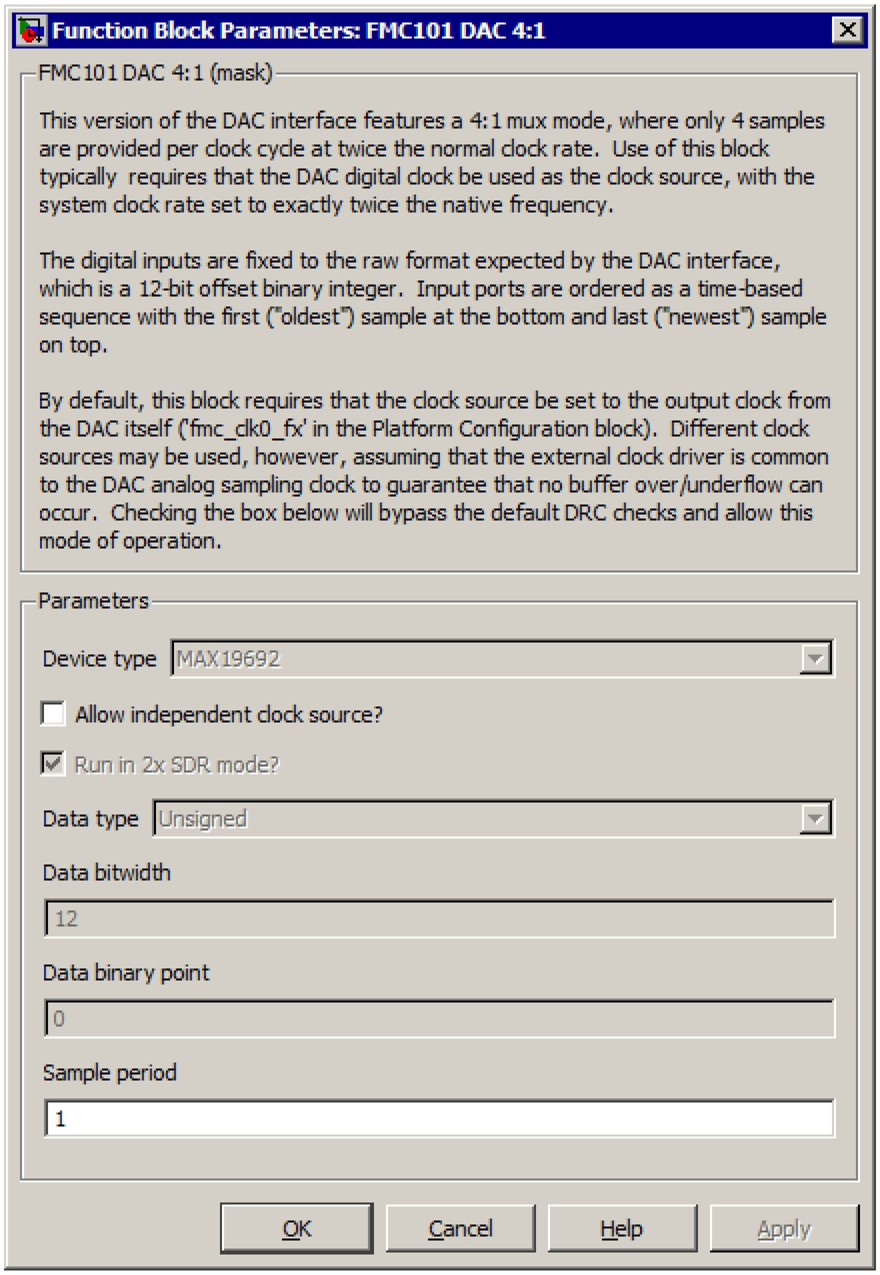


Figure 82: FMC101 DAC 4:1 block

The 4:1 version of the DAC interface accepts 4 digital values per FPGA clock cycle, which are driven by the DAC onto the analog output at 4x the system clock rate. The DAC device itself features 4 independent data buses which use DDR signaling, resulting in a digital clock frequency of 1/8th the sampling clock rate. Therefore, when using the 4:1 interface, the system clock rate must actually be twice the native DAC digital clock rate (which is referred to as *2x SDR mode*). For example, if the DAC sampling clock rate is 1GHz, the digital clock rate used by the 4 data buses is 125MHz. Therefore, the FPGA system clock rate must be set to 250MHz. Note that the system clock rate must sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This typically means that the DAC digital clock itself will be used as the reference clock source for the system, but this requirement may be overridden in cases where the sampling clock is also derived synchronously by an external source (such as the FMC101 ADC clock, which natively runs at 1/4th the sampling clock rate and can be used directly when the ADC and DAC share the same sampling clock).



Function Block Parameters: FMC101 DAC 4:1

FMC101 DAC 4:1 (mask)

This version of the DAC interface features a 4:1 mux mode, where only 4 samples are provided per clock cycle at twice the normal clock rate. Use of this block typically requires that the DAC digital clock be used as the clock source, with the system clock rate set to exactly twice the native frequency.

The digital inputs are fixed to the raw format expected by the DAC interface, which is a 12-bit offset binary integer. Input ports are ordered as a time-based sequence with the first ("oldest") sample at the bottom and last ("newest") sample on top.

By default, this block requires that the clock source be set to the output clock from the DAC itself ('fmc_clk0_fx' in the Platform Configuration block). Different clock sources may be used, however, assuming that the external clock driver is common to the DAC analog sampling clock to guarantee that no buffer over/underflow can occur. Checking the box below will bypass the default DRC checks and allow this mode of operation.

Parameters

Device type: MAX19692

☐ Allow independent clock source?

☑ Run in 2x SDR mode?

Data type: Unsigned

Data bitwidth: 12

Data binary point: 0

Sample period: 1

Figure 83: FMC101 DAC 4:1 dialog

Table 54: FMC101 DAC 4:1 block parameters

Parameter	Description
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Allow independent clock source?	Overrides the usual DRC checking, which requires that the DAC digital output clock be used as the source for the rest of the system. This box should only be checked when the user has guaranteed that the system clock source is generated by the same reference clock as the DAC analog sampling clock. This mode of operation is safe, since a common reference clock will guarantee that the DAC output FIFOs will not under/overflow during normal operation.
Run in 2x SDR mode?	When checked, the interface logic will be generated such that the system clock must run at twice the native DAC digital clock rate. This is required in the 4:1 configuration and is checked by default.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled DAC data inputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 12 bits to match the DAC data inputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled DAC data inputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 55: FMC101 DAC 4:1 port definitions

Port	Direction	Bit Width	Description

`d4_sim`
`d3_sim`
`d2_sim`
`d1_sim`
`fifo_empty_sim`
`fifo_full_sim`
	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the inputs below.

`d4`
`d3`
`d2`
`d1`
	in	12	Direct inputs to the DAC, representing 4 consecutive samples to be driven by the device. The samples are ordered with `d1` being the earliest and `d4` being the latest in temporal order. Data values must be raw, 12-bit, offset binary integers without any digital scaling or bias.

`fifo_empty` `fifo_full`	out	2	Empty and full status signals for the output FIFO which is part of the underlying interface logic. These signals are provided for debugging purposes only, and are never expected to be asserted during normal operation. *Note: these may be removed from the block in a future release.*

FMC101 DAC 8:1 block for BEE4 and ML60x

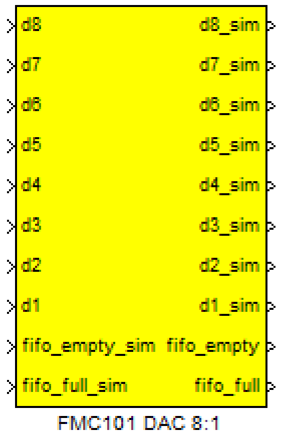
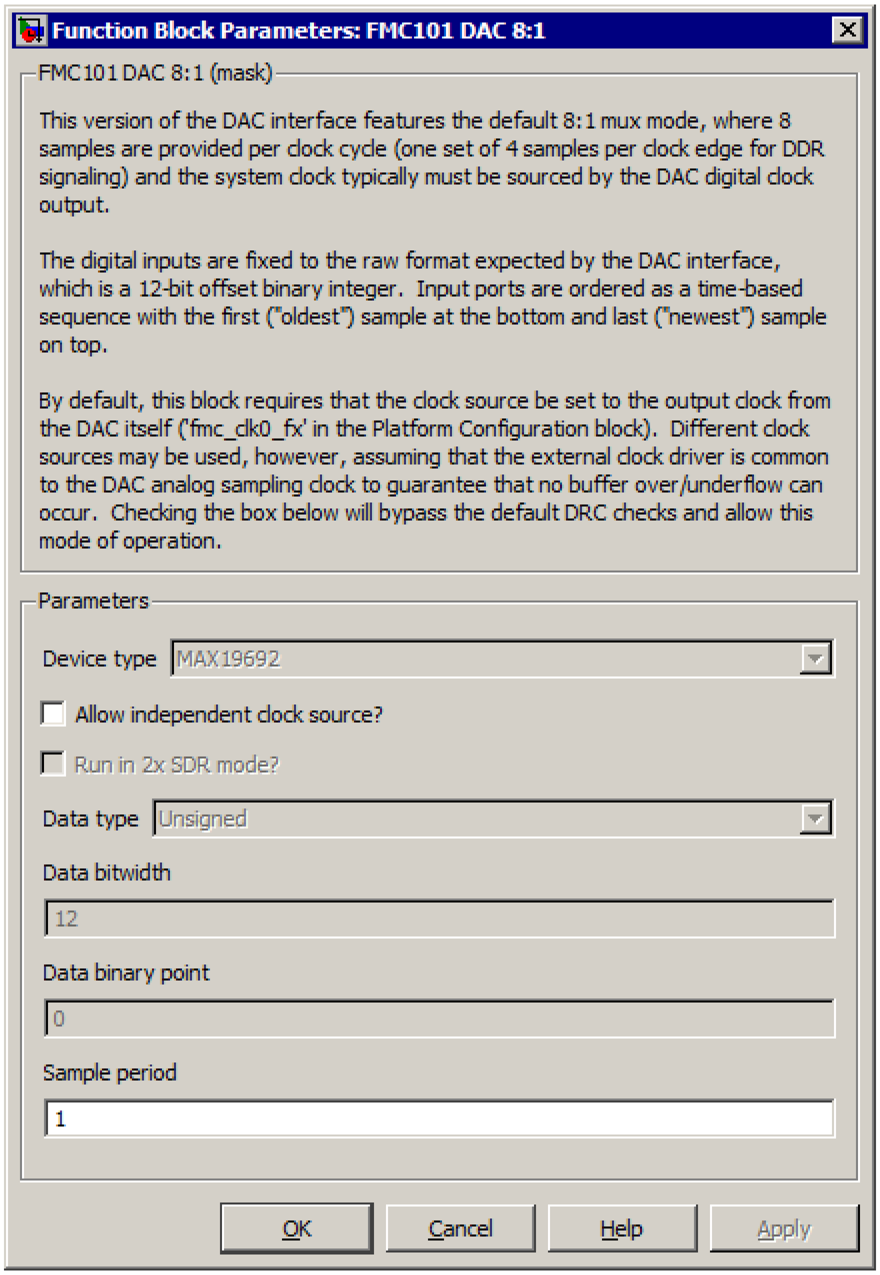


Figure 84: FMC101 DAC 8:1 block

The 8:1 version of the DAC interface accepts 8 digital values per FPGA clock cycle, which are driven by the DAC onto the analog output at 8x the system clock rate. The DAC device itself features 4 independent data buses which use DDR signaling, resulting in a digital clock frequency of 1/8th the sampling clock rate. Therefore, when using the 8:1 interface, the system clock rate must be equal to the native DAC digital clock rate. For example, if the DAC sampling clock rate is 1GHz, the digital clock rate used by the 4 data buses is 125MHz with DDR signaling (two samples per cycle). Since this version of the DAC interface provides all 8 samples per system clock cycle, the FPGA system clock rate must also be set to 125MHz. Note that the system clock rate must sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This typically means that the DAC digital clock itself will be used as the reference clock source for the system, but this requirement may be overridden in cases where the sampling clock is also derived synchronously by an external source.


Function Block Parameters: FMC101 DAC 8:1

FMC101 DAC 8:1 (mask)

This version of the DAC interface features the default 8:1 mux mode, where 8 samples are provided per clock cycle (one set of 4 samples per clock edge for DDR signaling) and the system clock typically must be sourced by the DAC digital clock output.

The digital inputs are fixed to the raw format expected by the DAC interface, which is a 12-bit offset binary integer. Input ports are ordered as a time-based sequence with the first ("oldest") sample at the bottom and last ("newest") sample on top.

By default, this block requires that the clock source be set to the output clock from the DAC itself ('fmc_clk0_fx' in the Platform Configuration block). Different clock sources may be used, however, assuming that the external clock driver is common to the DAC analog sampling clock to guarantee that no buffer over/underflow can occur. Checking the box below will bypass the default DRC checks and allow this mode of operation.

Parameters

Device type: MAX19692

☐ Allow independent clock source?

☐ Run in 2x SDR mode?

Data type: Unsigned

Data bitwidth: 12

Data binary point: 0

Sample period: 1

Figure 85: FMC101 DAC 8:1 dialog

Table 56: FMC101 DAC 8:1 block parameters

Parameter	**Description**
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Allow independent clock source?	Overrides the usual DRC checking, which requires that the DAC digital output clock be used as the source for the rest of the system. This box should only be checked when the user has guaranteed that the system clock source is generated by the same reference clock as the DAC analog sampling clock. This mode of operation is safe, since a common reference clock will guarantee that the DAC output FIFOs will not under/overflow during normal operation.
Run in 2x SDR mode?	When unchecked, the interface logic will be generated such that the system clock must run at the same rate as the native DAC digital clock. Since the DAC uses DDR signaling, this implies that two samples must be provided per bus per cycle (a total of 8). This is required in the 8:1 configuration and is unchecked by default.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled DAC data inputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 12 bits to match the DAC data inputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled DAC data inputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 57: FMC101 DAC 8:1 port definitions

Port	Direction	Bit Width	Description
`d8_sim` `d7_sim` `d6_sim` `d5_sim` `d4_sim` `d3_sim` `d2_sim` `d1_sim` `fifo_empty_sim` `fifo_full_sim`	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the inputs below.
`d8` `d7` `d6` `d5` `d4` `d3` `d2` `d1`	in	12	Direct inputs to the DAC, representing 8 consecutive samples to be driven by the device. The samples are ordered with `d1` being the earliest and `d8` being the latest in temporal order. Data values must be raw, 12-bit, offset binary integers without any digital scaling or bias.
`fifo_empty` `fifo_full`	out	2	Empty and full status signals for the output FIFO which is part of the underlying interface logic. These signals are provided for debugging purposes only, and are never expected to be asserted during normal operation. *Note: these may be removed from the block in a future release.*

FMC101 DAC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC101 DAC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc101_dac* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`fmc101_dac_init`

The `fmc101_dac_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will trigger a calibration of the device and reset all internal FIFOs and logic in the DAC interface.

Repeated function calls

(none)

Commands

`dac_reset`

The `dac_reset` function asserts a reset to the DAC interface logic, including the output FIFOs.

`dac_delay_set <data_ab_mask> <data_cd_mask> <odelay_val>`

The `dac_delay_set` function will override the default delay values for each output pin with the value specified in the argument `odelay_val`. The `data_ab_mask` argument is a 32-bit value, with pins in Bus A aligned in big-endian order to the MSB of the upper halfword (0xFFF00000), and pins in Bus B aligned in big-endian order to the MSB of the lower halfword (0x0000FFF0). The `data_cd_mask` argument is a 32-bit value, with pins in Bus C aligned in big-endian order to the MSB of the upper halfword (0xFFF00000), and pins in Bus D aligned in big-endian order to the MSB of the lower halfword (0x0000FFF0).

`dac_tp_start <a1> <b1> <c1> <d1> <a2> <b2> <c2> <d2>`

The `dac_tp_start` function will put the DAC interface logic into test pattern mode. In this mode, the digital outputs will be held in a fixed 8-sample pattern which is repeated indefinitely. Each argument is one 12-sample value, ordered from earliest to latest. For example, in order to have the DAC generate a maximum amplitude, maximum frequency output, use the command:
`dac_tp_start 0 xFFF 0 xFFF 0 xFFF 0 xFFF`

`dac_tp_stop`

The `dac_tp_stop` function will force the DAC to exit test pattern mode and resume normal operation, with all output samples taken from the input ports to the block.

```
dac_mode_set <(nrz|rz|rf)>
```

The `dac_mode_set` function will set the corresponding control pins on the DAC device to one of its three output operating modes: NRZ, RZ, and RF. After FPGA programming, the DAC device is set into NRZ mode by default. If another operating mode is desired, a call to set the proper mode should be added to a custom initialization routine.

FMC104 ADC

This block provides a direct interface to the ADC device on the BEEcube FMC104 expansion board for the BEE4, BEE7, and miniBEE4 hardware platforms.

FMC104 ADC block for BEE4, BEE7, and miniBEE4

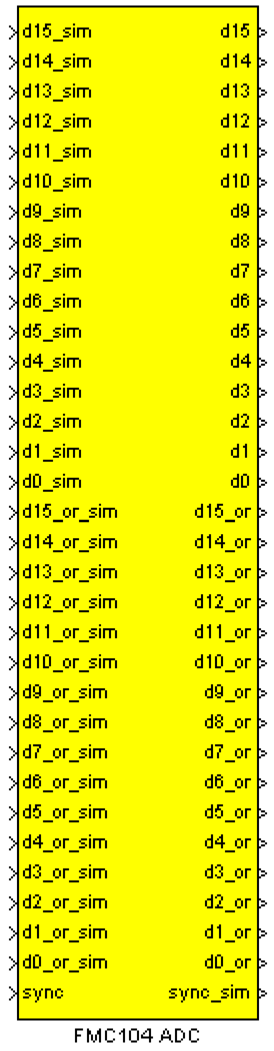
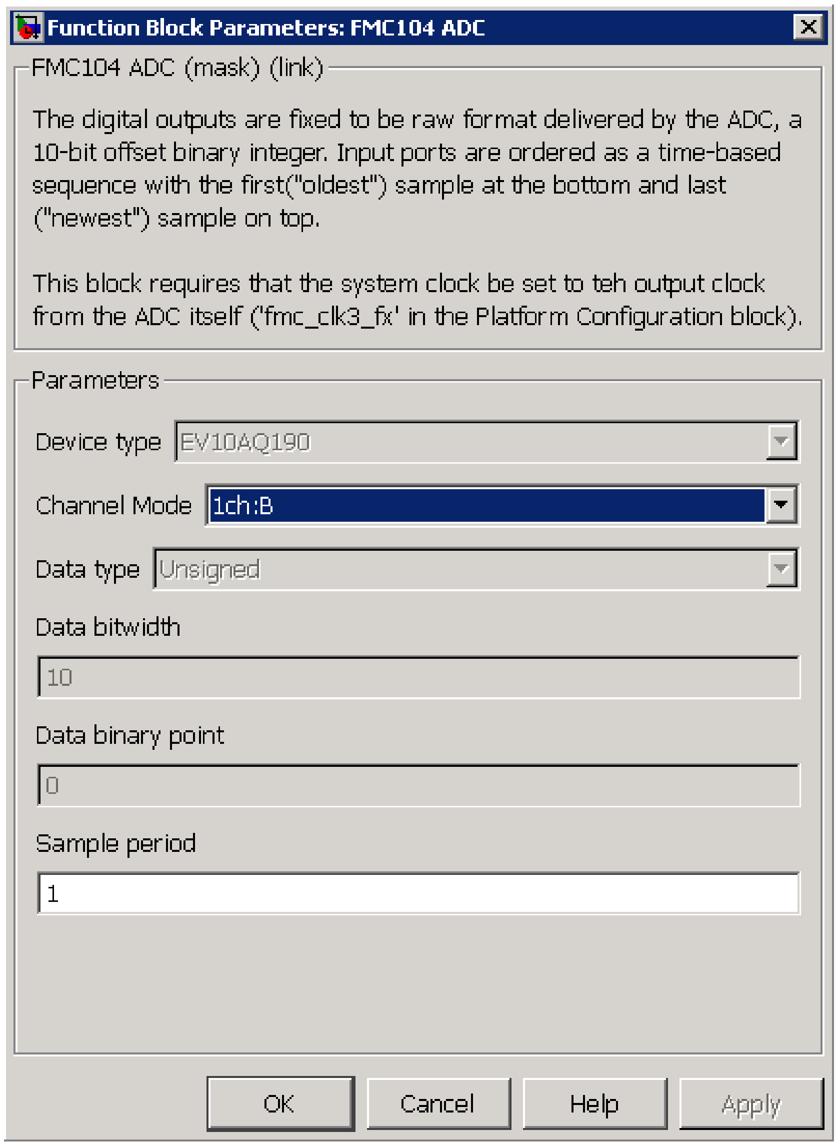


Figure 86: FMC104 ADC block

The FMC104 ADC block delivers 16 digital values per FPGA clock cycle, which were sampled by the ADC at 16x the system clock rate. The ADC device itself samples at twice its native clock rate. For example, when the ADC receives a 2.5GHz clock, its sampling rate is 5GHz.

On the FPGA side, the ADC features 4 independent data buses, which are sampled by 4:1 SerDes devices. This results in a digital clock frequency of 1/16th the sampling rate (or 1/8th the native ADC digital clock rate). Therefore, the system clock rate must be 1/8th the native ADC digital clock rate. For example, if the ADC sampling clock rate is 2.5GHz, the digital clock rate used by the 4 data buses is 312.5MHz. Therefore, the FPGA system clock rate also must be set to 312.5 MHz. Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface.



The digital outputs are fixed to be raw format delivered by the ADC, a 10-bit offset binary integer. Input ports are ordered as a time-based sequence with the first("oldest") sample at the bottom and last ("newest") sample on top.

This block requires that the system clock be set to teh output clock from the ADC itself ('fmc_clk3_fx' in the Platform Configuration block).

Parameters

Device type: EV10AQ190

Channel Mode: 1ch:B

Data type: Unsigned

Data bitwidth: 10

Data binary point: 0

Sample period: 1

Figure 87: FMC104 ADC dialog

Table 58: FMC104 ADC block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board.
Channel mode	This parameter determines which channel sources are used. The outputs of the block are consistent with 1-channel mode on the ADC: samples 0-3 are from ADC A, 4-7 are from ADC C, 8-11 are from ADC B, and 12-15 are from ADC D. In 2-channel mode, samples 0-3 and 8-11 are from the AB pair, and samples 4-7 and 12-15 are from the CD pair.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 10 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 59: FMC104 ADC port definitions

Port	Direction	Bit Width	Description
d15_sim d14_sim d13_sim d12_sim d11_sim d10_sim d9_sim d8_sim d7_sim d6_sim d5_sim d4_sim d3_sim d2_sim d1_sim d0_sim	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.
d15_or_sim d14_or_sim d13_or_sim d12_or_sim d11_or_sim d10_or_sim d9_or_sim d8_or_sim d7_or_sim d6_or_sim d5_or_sim d4_or_sim d3_or_sim d2_or_sim d1_or_sim d0_or_sim			

`d15`
`d14`
`d13`
`d12`
`d11`
`d10`
`d9`
`d8`
`d7`
`d6`
`d5`
`d4`
`d3`
`d2`
`d1`
`d0`
	out	10	Direct outputs from the ADC, representing 16 consecutive samples by the device. The samples are ordered with `d0` being the earliest and `d15` being the latest in temporal order. Data values are raw, 10-bit, offset binary integers without any digital scaling or bias.

`d15_or`
`d14_or`
`d13_or`
`d12_or`
`d11_or`
`d10_or`
`d9_or`
`d8_or`
`d7_or`
`d6_or`
`d5_or`
`d4_or`
`d3_or`
`d2_or`
`d1_or`
`d0_or`
	out	1	ADC out-of-range (OR) signal for each of the 16 corresponding data samples. Indicates if an out-of-range input caused saturation in the ADC.

FMC104 ADC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC101 ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc101_adc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
fmc104_get_status <core>
```

The `fmc104_get_status` function reports whether the FPGA has locked onto the clock sourced from the ADC.

```
fmc104_reset <core>
```

The `fmc104_reset` function asserts a reset to the ADC interface logic (but not the ADC itself).

```
fmc104_reg_write <core> <reg_addr> <value>
```

The `fmc104_reg_write` function will write an arbitrary value into any one of the ADC's internal configuration registers. The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and `value` is the 16-bit value which should be written.

```
fmc104_test_pattern <core> <(on|off)>
```

The `fmc104_test_pattern` function will put the ADC into test pattern generation mode, which is currently a repeating ramp pattern from 0 to 2047. This command may be used to validate proper connectivity of the card and input clocks, and successful calibration of the input delays.

```
fmc104_set_delay <core> <lane_mask> <bit_mask> <idelay_val>
```

The `fmc104_set_delay` function will override the previously calculated delay values for each input pin with the value specified in the argument `idelay_val`. The `lane_mask` argument is a 4-bit mask selecting which of the four lanes (from ADCs A, C, B, and D) to change. The `bit_mask` argument is a 11-bit value, with pins aligned in little-endian order and bit 10 being the OR bit.

`fmc104_calibrate_delays <core>`

The `fmc104_calibrate_delays` function will re-perform an automatic calibration of the input delays for each of the ADC pins and plot the results in a table.

`fmc104_print_calibration_table <core>`

The `fmc104_print_calibration_table` function will print out the results of the calibration along with the selected taps. A ~ represents values for which a particular bit and tap value passed its test, @ represents the selected tap value, and . represents non-passing tap values.

`fmc104_calibrate_adc <core> <channel> <gain> <offset> <phase>`

The `fmc104_calibrate_adc` function is used to set the `gain`, `offset`, and `phase` parameters of an individual ADC on the board. Valid values for the channel are A, B, C, and D. Refer to the EV10AQ190A data sheet for the encoding of `gain`, `offset`, and `phase`.

FMC105 DAC

This block provides a direct interface to the DAC device on the BEEcube FMC105 expansion board for the BEE4, BEE7, and miniBEE4 hardware platforms.

FMC105 DAC block for BEE4, BEE7, and miniBEE4

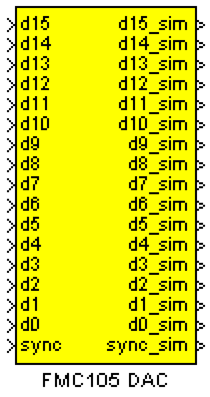
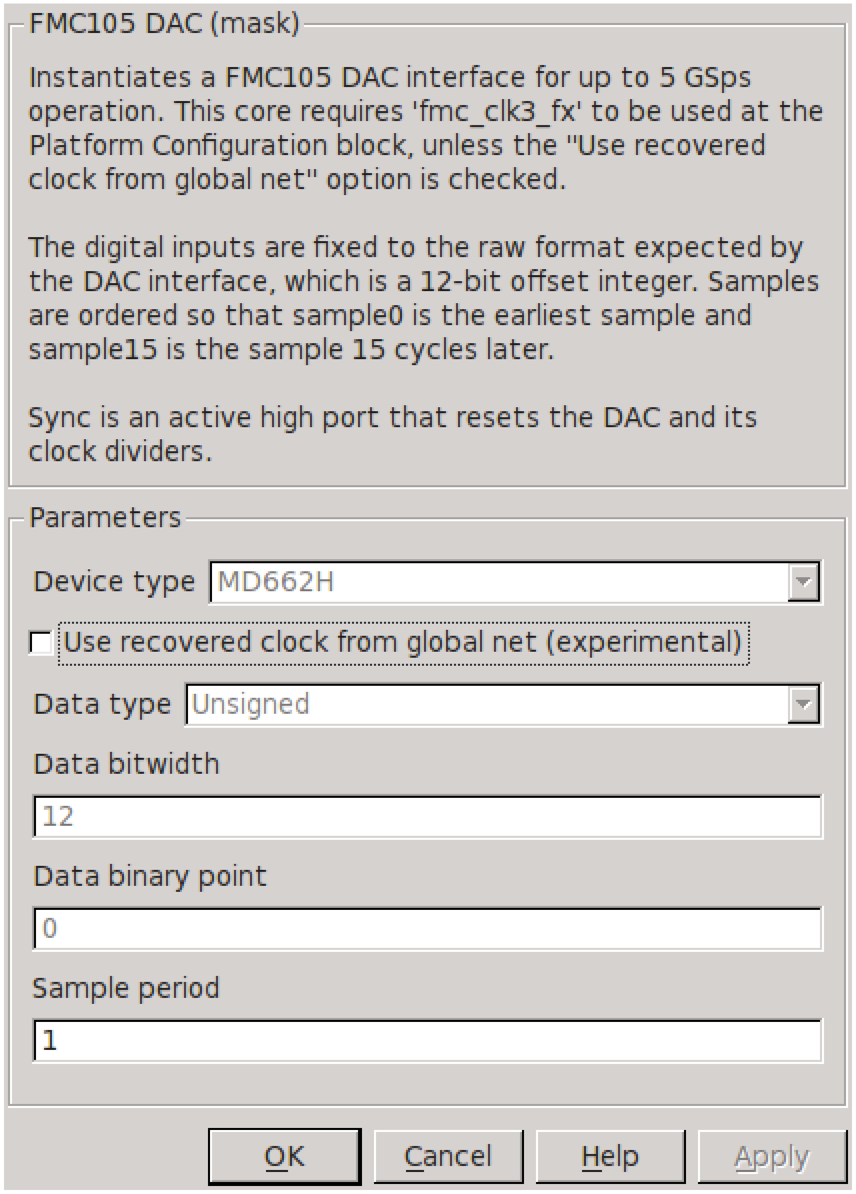


Figure 88: FMC105 DAC block

The DAC interface accepts 16 digital values per FPGA clock cycle, which are driven by the DAC onto the analog output at 16x the system clock rate. The DAC device itself features 4 independent data buses, which the FPGA drives with 4:1 SerDes devices. The resulting digital clock frequency is 1/16th the sampling rate. The DAC itself samples at twice its input clock rate (e.g. a 2.5 GHz clock into the DAC yields a 5 GHz sampling rate). Therefore, the system clock rate must be 1/8th the native DAC digital clock rate. For example, if the DAC's input clock rate is 2.5 GHz, the FPGA system clock rate must be set to 312.5 MHz.

Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This means that the DAC digital clock itself will be used as the reference clock source for the system. In Platform Configuration, this is the fmc_clk3_fx clock for BEE4 and miniBEE, and this is the fmc_clk0_fx clock for BEE7.



FMC105 DAC (mask)

Instantiates a FMC105 DAC interface for up to 5 GSps operation. This core requires 'fmc_clk3_fx' to be used at the Platform Configuration block, unless the "Use recovered clock from global net" option is checked.

The digital inputs are fixed to the raw format expected by the DAC interface, which is a 12-bit offset integer. Samples are ordered so that sample0 is the earliest sample and sample15 is the sample 15 cycles later.

Sync is an active high port that resets the DAC and its clock dividers.

Parameters

Device type: MD662H

☐ Use recovered clock from global net (experimental)

Data type: Unsigned

Data bitwidth: 12

Data binary point: 0

Sample period: 1

Figure 89: FMC105 DAC dialog

Table 60: FMC105 DAC block parameters

Parameter	**Description**
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Use recovered clock from global net (experimental)	This parameter permits the design to source its clock from a global net, instead of the DAC. This may be useful for synchronization across multiple FPGAs, and it causes DRC on the clock net to be bypassed.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled DAC data inputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 12 bits to match the DAC data inputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled DAC data inputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 61: FMC105 DAC port definitions

Port	Direction	Bit Width	Description

`d15_sim`
`d14_sim`
`d13_sim`
`d12_sim`
`d11_sim`
`d10_sim`
`d9_sim`
`d8_sim`
`d7_sim`
`d6_sim`
`d5_sim`
`d4_sim`
`d3_sim`
`d3_sim`
`d2_sim`
`d0_sim`
	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the inputs below.

`d15`
`d14`
`d13`
`d12`
`d11`
`d10`
`d9`
`d8`
`d7`
`d6`
`d5`
`d4`
`d3`
`d2`
`d1`
`d0`
	in	12	Direct inputs to the DAC, representing 16 consecutive samples to be driven by the device. The samples are ordered with `d0` being the earliest and `d15` being the latest in temporal order. Data values must be raw, 12-bit, offset binary integers without any digital scaling or bias.

FMC105 DAC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC101 DAC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc101_dac* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
fmc105_get_status <core>
```

The `fmc105_get_status` returns which of the 4 sampling phases is currently in use on the DAC and whether the FPGA has locked onto the clock sourced from the DAC.

```
fmc105_reset <core>
```

The `fmc105_reset` function asserts a reset to the DAC interface logic.

```
fmc105_set_delay <core> <bit_mask_hi> <bit_mask_lo> <odelay_val>
```

The `fmc105_set_delay` function will override the default delay values for each output pin with the value specified in the argument `odelay_val`. The `bit_mask_hi` argument is a 16-bit value and the `bit_mask_lo` argument is a 32-bit value representing the combined 48 differential pairs used. Pins in lane D are aligned in little-endian order to the MSB of the lower halfword of `bit_mask_hi` (xFFF0), pins in lane C are split between the lowest nibble of `bit_mask_hi` and the upper byte of `bit_mask_lo` (xF and xFF000000, respectively), pins in lane B are in the next 3 nibbles of `bit_mask_lo` (x00FFF000), and pins in lane A are in the lowest 3 nibbles of `bit_mask_lo` (x00000FFF)

FMC106 ADC

This block provides a direct interface to the ADC device on the BEEcube FMC106 expansion board for the BEE7, BEE4, and miniBEE4 hardware platforms.

FMC106 ADC block for BEE7, BEE4, and miniBEE4

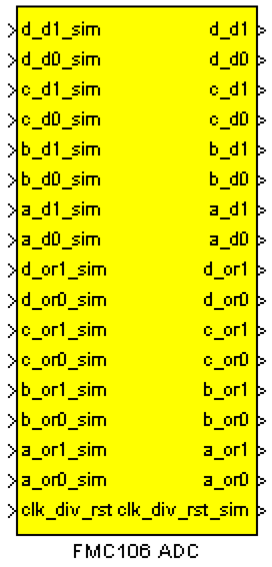
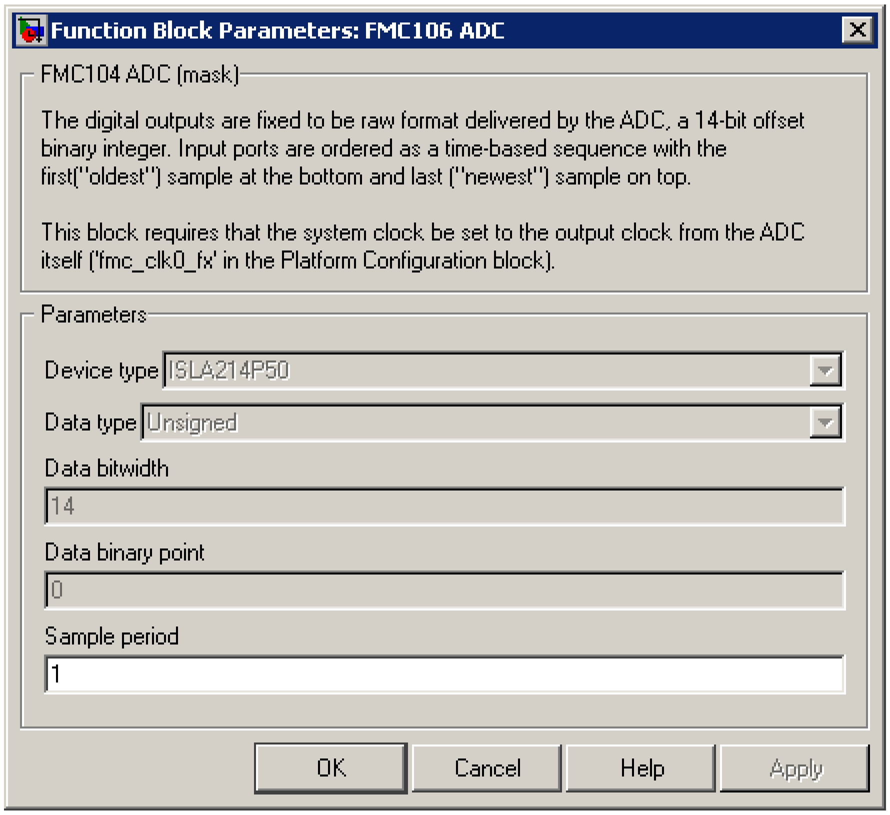


Figure 90: FMC106 ADC block

The FMC106 ADC block delivers 2 digital values per ADC chip/channel, per FPGA clock cycle. These values were sampled by the ADC at 2x the system clock rate. Each ADC chip itself samples at its input clock rate.

On the FPGA side, each channel of the ADC is latched on the FPGA at a DDR rate. This results in a digital clock frequency of ½ the sampling rate. Therefore, the system clock rate must be ½ the ADC sampling clock rate. For example, if the ADC sampling clock rate is 500MHz, the system clock rate used by the 4 data buses is 250MHz. Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface.


Function Block Parameters: FMC106 ADC

FMC104 ADC (mask)

The digital outputs are fixed to be raw format delivered by the ADC, a 14-bit offset binary integer. Input ports are ordered as a time-based sequence with the first("oldest") sample at the bottom and last ("newest") sample on top.

This block requires that the system clock be set to the output clock from the ADC itself ('fmc_clk0_fx' in the Platform Configuration block).

Parameters

Device type: ISLA214P50

Data type: Unsigned

Data bitwidth: 14

Data binary point: 0

Sample period: 1

Figure 91: FMC106 ADC dialog

Table 62: FMC106 ADC block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 14 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 63: FMC106 ADC port definitions

Port	Direction	Bit Width	Description

`d_d1_sim`
`d_d0_sim`
`c_d1_sim`
`c_d0_sim`
`b_d1_sim`
`b_d0_sim`
`a_d1_sim`
`a_d0_sim`
	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.

`d_or1_sim`
`d_or0_sim`
`c_or1_sim`
`c_or0_sim`
`b_or1_sim`
`b_or0_sim`
`a_or1_sim`
`a_or0_sim`
	in	N/A	Inputs which can be used to provide simulation out-of-range data within Simulink. Each input corresponds directly to each of the outputs below.

`d_d1`
`d_d0`
`c_d1`
`c_d0`
`b_d1`
`b_d0`
`a_d1`
`a_d0`
	out	14	Direct outputs from the ADC, representing 2 consecutive samples by each of the ADC channels. The samples are ordered with `a0` being the earliest and `a1` being the latest in temporal order. Data values are raw, 14-bit, offset binary integers without any digital scaling or bias.

`d_or1`
`d_or0`
`c_or1`
`c_or0`
`b_or1`
`b_or0`
`a_or1`
`a_or0`
	out	1	ADC out-of-range (OR) signal for the corresponding data samples. Indicates if an out-of-range input caused saturation in the ADC.

FMC106 ADC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC106 ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc106_adc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
fmc106_get_status <core>
```

The `fmc106_get_status` function reports whether the FPGA has locked onto the clock sourced from the ADC.

```
fmc106_reset <core>
```

The `fmc106_reset` function asserts a reset to the ADC interface logic (but not the ADC itself).

```
fmc106_reg_write <core> <channel mask> <reg_addr> <value>
```

The `fmc106_reg_write` function will write an arbitrary value into any one of the ADC's internal configuration registers. The `channel_mask` argument is a bitmask selecting each of the four ADCs on the board; channel A is mapped to bit 0 and channel D is mapped to bit 3 (e.g. 0xF selects all ADCs, and 0xA selects channels B and D). The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and `value` is the 8-bit value which should be written.

```
fmc106_reg_read <core> <channel 0-3> <reg_addr>
```

The `fmc106_reg_read` function will print values from any one of the ADC's internal configuration registers. The `channel` number selects one of the ADCs to read from; channel A is mapped to 0, B to 1, C to 2, and D to 3. The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and the function will print out the value stored within.

```
fmc106_test_pattern <core> <(on|off)>
```

The `fmc106_test_pattern` function will put the ADCs into test pattern generation mode, which is defined in the device datasheet. This command may be used to validate proper connectivity of the card and input clocks, and successful calibration of the input delays.

```
fmc106_set_delay <core> <channel_mask> <bit_mask> <idelay_val>
```

The `fmc106_set_delay` function will override the previously calculated delay values for each input pin with the value specified in the argument `idelay_val`. The `channel_mask` argument is a bitmask of which channels to change, with bit 3 representing the D channel and bit 0 representing the A channel. The `bit_mask` argument is a 14-bit value, with pins aligned in little-endian order.

```
fmc106_calibrate_delays <core>
```

The `fmc106_calibrate_delays` function will re-perform an automatic calibration of the input delays for each of the ADC pins and plot the results in a table.

```
fmc106_print_calibration_table <core>
```

The `fmc106_print_calibration_table` function will print out the results of the calibration along with the selected taps. A ~ represents values for which a particular bit and tap value passed its test, @ represents the selected tap value, and . represents non-passing tap values.

FMC107 DAC

This block provides a direct interface to the quad-channel DAC device on the BEEcube FMC107 expansion board for the BEE4 and miniBEE4 hardware platforms.

FMC107 DAC block for BEE4 and miniBEE4

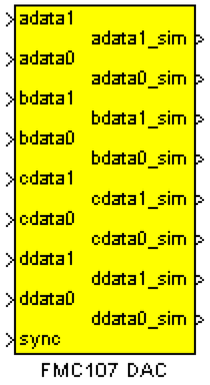
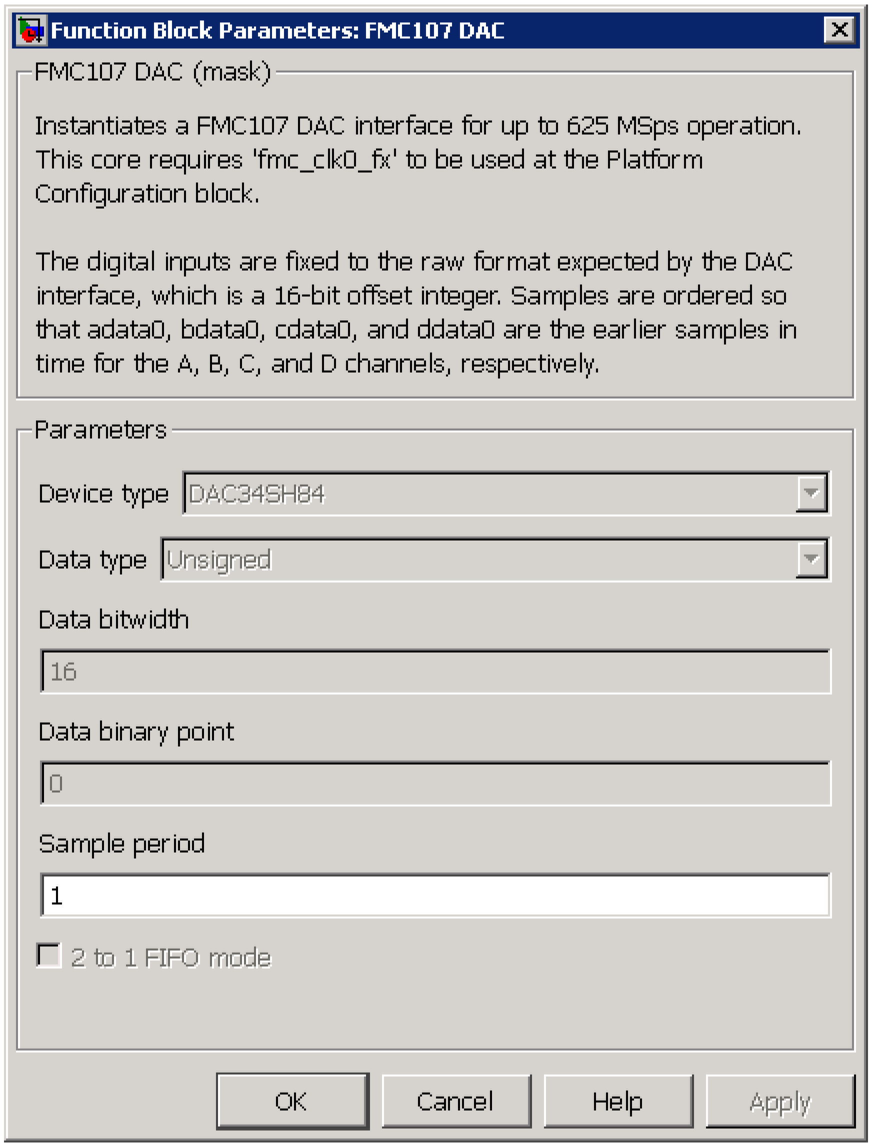


Figure 92: FMC107 DAC block

The DAC interface accepts 8 digital values per FPGA clock cycle (2 per channel), which are driven by the DAC onto the analog output at 4x the system clock rate. With the default interpolation setting of 2x, the digital clock frequency is 2x the sampling rate. For example, if the DAC's input clock rate is 1.25 GHz, the FPGA system clock rate must be set to 312.5 MHz. The resulting update rate will be 625 MHz.

Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This means that the DAC digital clock itself will be used as the reference clock source for the system. In Platform Configuration, this is the fmc_clk0_fx clock.


Function Block Parameters: FMC107 DAC

FMC107 DAC (mask)

Instantiates a FMC107 DAC interface for up to 625 MSps operation. This core requires 'fmc_clk0_fx' to be used at the Platform Configuration block.

The digital inputs are fixed to the raw format expected by the DAC interface, which is a 16-bit offset integer. Samples are ordered so that adata0, bdata0, cdata0, and ddata0 are the earlier samples in time for the A, B, C, and D channels, respectively.

Parameters

Device type: DAC34SH84

Data type: Unsigned

Data bitwidth: 16

Data binary point: 0

Sample period: 1

☐ 2 to 1 FIFO mode

Figure 93: FMC107 DAC dialog

Table 64: FMC107 DAC block parameters

Parameter	**Description**
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled DAC data inputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 16 bits to match the DAC data inputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled DAC data inputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 65: FMC109 DAC port definitions

Port	Direction	Bit Width	Description
`adata1_sim` `adata0_sim` `bdata1_sim` `bdata0_sim` `cdata1_sim` `cdata0_sim` `ddata1_sim` `ddata0_sim`	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the inputs below.
`adata1` `adata0`	in	16	Direct inputs to DAC channel “A”, representing 2 consecutive samples to be driven by the device. The samples are ordered with `adata0` being the earliest and `adata1` being the latest in temporal order. Data values must be raw, 16-bit, offset binary integers without any digital scaling or bias.
`bdata1` `bdata0`	in	16	Direct inputs to DAC channel “B”, representing 2 consecutive samples to be driven by the device. The samples are ordered with `bdata0` being the earliest and `bdata1` being the latest in temporal order. Data values must be raw, 16-bit, offset binary integers without any digital scaling or bias.
`cdata1` `cdata0`	in	16	Direct inputs to DAC channel “C”, representing 2 consecutive samples to be driven by the device. The samples are ordered with `cdata0` being the earliest and `cdata1` being the latest in temporal order. Data values must be raw, 16-bit, offset binary integers without any digital scaling or bias.
`ddata1` `ddata0`	in	16	Direct inputs to DAC channel “D”, representing 2 consecutive samples to be driven by the device. The samples are ordered with `ddata0` being the earliest and `ddata1` being the latest in temporal order. Data values must be raw, 16-bit, offset binary integers without any digital scaling or bias.

FMC107 DAC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC107 DAC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc107_dac* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
fmc107_get_status <core>
```

The `fmc107_get_status` returns whether the FPGA has locked onto the clock sourced from the DAC.

```
fmc107_reset <core>
```

The `fmc107_reset` function asserts a reset to the DAC interface logic.

```
fmc107_set_delay <core> <AB bit_mask> <CD bit_mask> <odelay_val>
```

The `fmc107_set_delay` function will override the default delay values for each output pin with the value specified in the argument `odelay_val`. The `AB bit_mask and CD bit_mask` arguments are 32-bit values representing the 17 differential pairs each used for the A&B channels and C&D channels, respectively. The bitmask is the concatenation of the parity signal with the data port, in little-endian order.

Please note that two DAC channels share one LVDS bus, so an incorrect setting can swap the two channels.

```
fmc107_reg_write <core> <channel_mask> <reg_addr> <value>
```

The `fmc107_reg_write` function will write an arbitrary value into any one of the DAC's internal configuration registers. The address `reg_addr` is the address of the DAC configuration register as defined in the device datasheet, and `value` is the 16-bit value which should be written.

```
fmc107_reg_read <core> <reg_addr>
```

The `fmc107_reg_read` function will print values from any one of the DACs' internal configuration registers. The `reg_addr` argument is the address of the DAC configuration register as defined in the device datasheet, and the function will print out the value stored within.

FMC108 ADC

This block provides a direct interface to the ADC device on the BEEcube FMC108 expansion board for the BEE4, BEE7, and miniBEE4 hardware platforms.

FMC108 ADC block for BEE4, BEE7, and miniBEE4

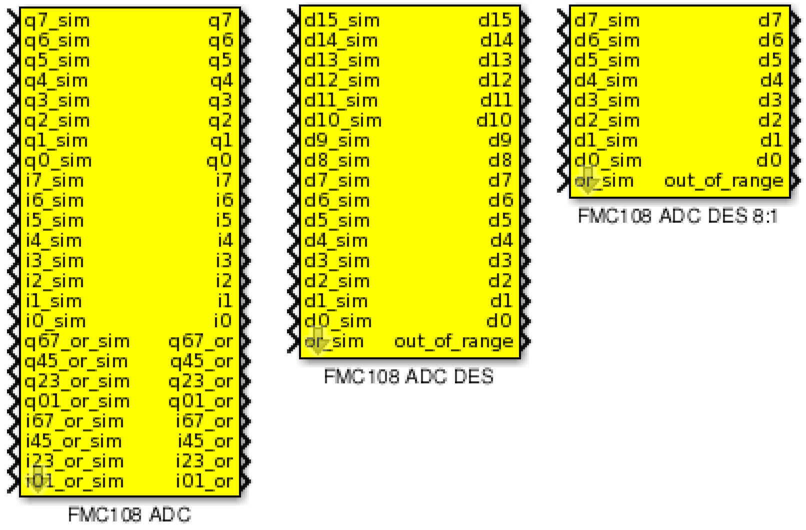
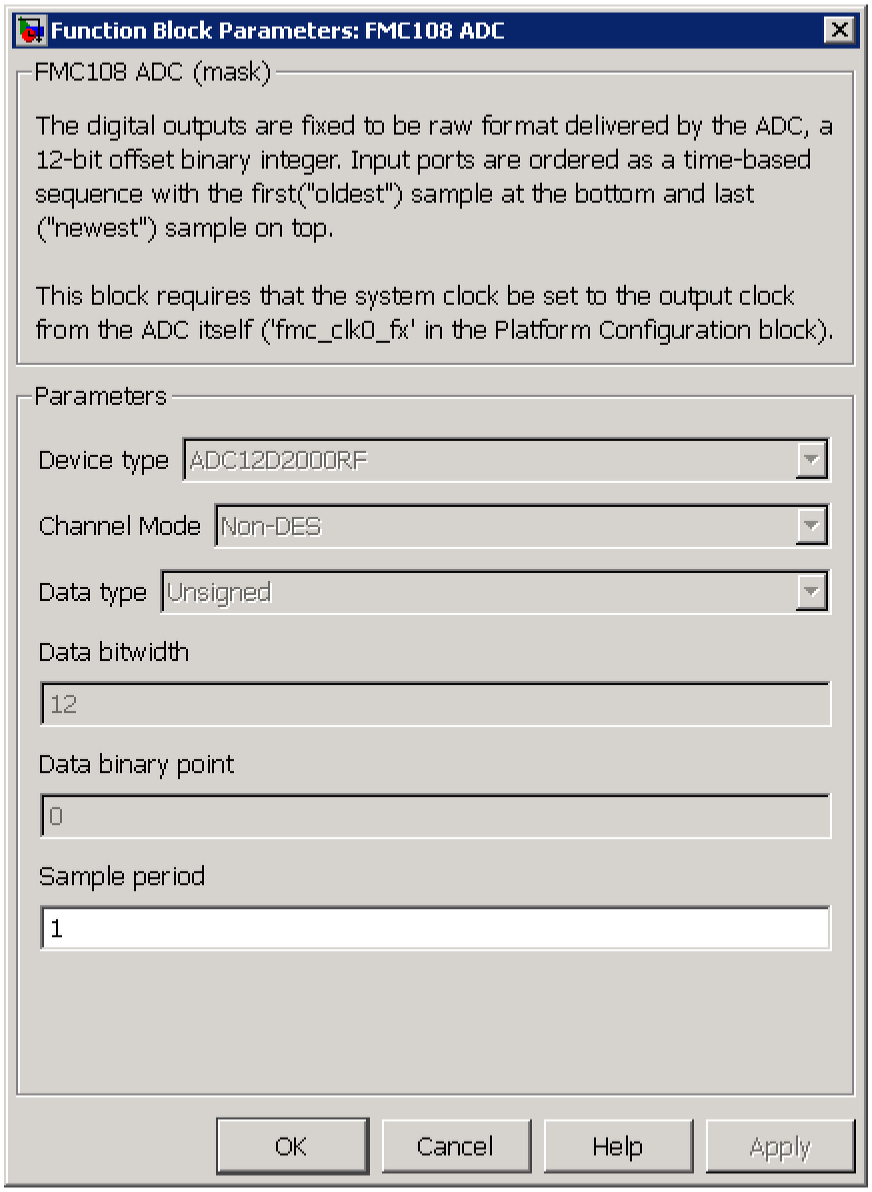

q7_sim q6_sim q5_sim q4_sim q3_sim q2_sim q1_sim q0_sim
i7_sim i6_sim i5_sim i4_sim i3_sim i2_sim i1_sim i0_sim
q67_or_sim q45_or_sim q23_or_sim q01_or_sim i67_or_sim i45_or_sim i23_or_sim i01_or_sim
q7 q6 q5 q4 q3 q2 q1 q0 i7 i6 i5 i4 i3 i2 i1 i0
q67_or q45_or q23_or q01_or i67_or i45_or i23_or i01_or
FMC108 ADC
d15_sim d14_sim d13_sim d12_sim d11_sim d10_sim d9_sim d8_sim d7_sim d6_sim d5_sim d4_sim d3_sim d2_sim d1_sim d0_sim
or_sim
d15 d14 d13 d12 d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
out_of_range
FMC108 ADC DES
d7_sim d6_sim d5_sim d4_sim d3_sim d2_sim d1_sim d0_sim
or_sim
d7 d6 d5 d4 d3 d2 d1 d0
out_of_range
FMC108 ADC DES 8:1

Figure 94: FMC108 blocks

The FMC108 ADC block delivers 8 digital values per channel, per FPGA clock cycle (or 4 for DES 8:1). These values were sampled by the ADC at 8x /4x the system clock rate. The ADC device itself samples at its input clock rate.

On the FPGA side, each channel of the ADC features 2 independent data buses, which are sampled by 4:1 SerDes devices. This results in a digital clock frequency of 1/8th the sampling rate. Therefore, the system clock rate must be 1/8th the ADC sampling clock rate. For example, if the ADC sampling clock rate is 2.5GHz, the digital clock rate used by the 4 data buses is 312.5MHz. Therefore, the FPGA system clock rate also must be set to 312.5 MHz. Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface.

For the FMC108 DES 8:1 block, the digital clock frequency is 1/4th the sampling rate.


Function Block Parameters: FMC108 ADC

FMC108 ADC (mask)

The digital outputs are fixed to be raw format delivered by the ADC, a 12-bit offset binary integer. Input ports are ordered as a time-based sequence with the first("oldest") sample at the bottom and last ("newest") sample on top.

This block requires that the system clock be set to the output clock from the ADC itself ('fmc_clk0_fx' in the Platform Configuration block).

Parameters

Device type: ADC12D2000RF

Channel Mode: Non-DES

Data type: Unsigned

Data bitwidth: 12

Data binary point: 0

Sample period: 1

Figure 95: FMC108 ADC dialog

Table 66: FMC108 ADC block parameters

Parameter	**Description**
Device type	This parameter defines the type of ADC device present on the ADC expansion board.
Channel mode	This parameter determines which mode the ADC is set in. BPS supports the *Non-DES* mode, where both channels run at up to 2 GSps, and the *DESI, DESQ, DESIQ, and DESCLKIQ* modes, where the two channels interleave for an effective sampling rate of up to 4 GSps.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 12 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 67: FMC108 ADC port definitions

Port	Direction	Bit Width	Description

`i7_sim`
`i6_sim`
`i5_sim`
`i4_sim`
`i3_sim`
`i2_sim`
`i1_sim`
`i0_sim`
`q7_sim`
`q6_sim`
`q5_sim`
`q4_sim`
`q3_sim`
`q2_sim`
`q1_sim`
`q0_sim`
`i67_or_sim`
`i45_or_sim`
`i23_or_sim`
`i01_or_sim`
`q67_or_sim`
`q45_or_sim`
`q23_or_sim`
`q01_or_sim`
	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.

`i7`
`i6`
`i5`
`i4`
`i3`
`i2`
`i1`
`i0`
	out	12	Direct outputs from the ADC, representing 8 consecutive samples by the device's I channel. The samples are ordered with `i0` being the earliest and `i7` being the latest in temporal order. Data values are raw, 12-bit, offset binary integers without any digital scaling or bias.

`q7`
`q6`
`q5`
`q4`
`q3`
`q2`
`q1`
`q0`
	out	12	Direct outputs from the ADC, representing 8 consecutive samples by the device's Q channel. The samples are ordered with `q0` being the earliest and `q7` being the latest in temporal order. Data values are raw, 12-bit, offset binary integers without any digital scaling or bias.

`i67_or`
`i45_or`
`i23_or`
`i01_or`
	out	1	ADC out-of-range (OR) signal for pairs of the corresponding data samples. For example, `i01_or` is the out-of-range indicator for samples `i0` and `i1`. Indicates if an out-of-range input caused saturation in the ADC.

`q67_or`
`q45_or`
`q23_or`
`q01_or`
	out	1	ADC out-of-range (OR) signal for pairs of the corresponding data samples. For example, `q01_or` is the out-of-range indicator for samples `q0` and `q1`. Indicates if an out-of-range input caused saturation in the ADC.

FMC108 ADC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC108 ADC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc108_adc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`fmc108_get_status <core>`

The `fmc108_get_status` function reports whether the FPGA has locked onto the clock sourced from the ADC.

`fmc108_reset <core>`

The `fmc108_reset` function asserts a reset to the ADC interface logic (but not the ADC itself).

`fmc108_reg_write <core> <reg_addr> <value>`

The `fmc108_reg_write` function will write an arbitrary value into any one of the ADC's internal configuration registers. The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and `value` is the 16-bit value which should be written.

`fmc108_reg_read <core> <reg_addr>`

The `fmc108_reg_read` function will print values from any one of the ADC's internal configuration registers. The address `reg_addr` is the address of the ADC configuration register as defined in the device datasheet, and the function will print out the value stored within.

`fmc108_test_pattern <core> <(on|off)>`

The `fmc108_test_pattern` function will put the ADC into test pattern generation mode, which is defined in the device datasheet. This command may be used to validate proper connectivity of the card and input clocks, and successful calibration of the input delays.

`fmc108_set_delay <core> <channel_mask Q,I> <bit_mask> <idelay_val>`

The `fmc108_set_delay` function will override the previously calculated delay values for each input pin with the value specified in the argument `idelay_val`. The `channel_mask` argument is a bitmask of which channels to change, with bit 1 representing the Q channel and bit 0 representing the I channel. The `bit_mask` argument is a 25-bit value, with pins aligned in little-endian order and bit 24 being the OR bit. The delayed samples are on the less significant bits; thus, the bits are ordered like so: {I_OR, I, Id} and {Q_OR, Q, Qd}.

`fmc108_calibrate_delays <core>`

The `fmc108_calibrate_delays` function will re-perform an automatic calibration of the input delays for each of the ADC pins and plot the results in a table.

`fmc108_print_calibration_table <core>`

The `fmc108_print_calibration_table` function will print out the results of the calibration along with the selected taps. A ~ represents values for which a particular bit and tap value passed its test, @ represents the selected tap value, and . represents non-passing tap values.

FMC109 DAC

This block provides a direct interface to the 2 DAC devices on the BEEcube FMC109 expansion board for the BEE4, BEE7, and miniBEE4 hardware platforms.

FMC109 DAC block for BEE4, BEE7, and miniBEE4

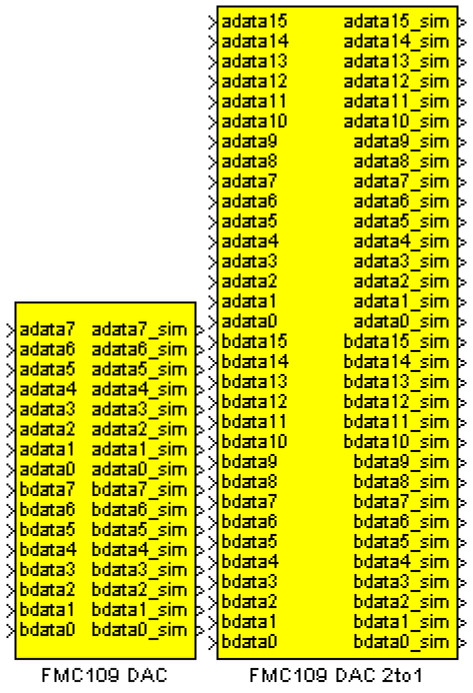
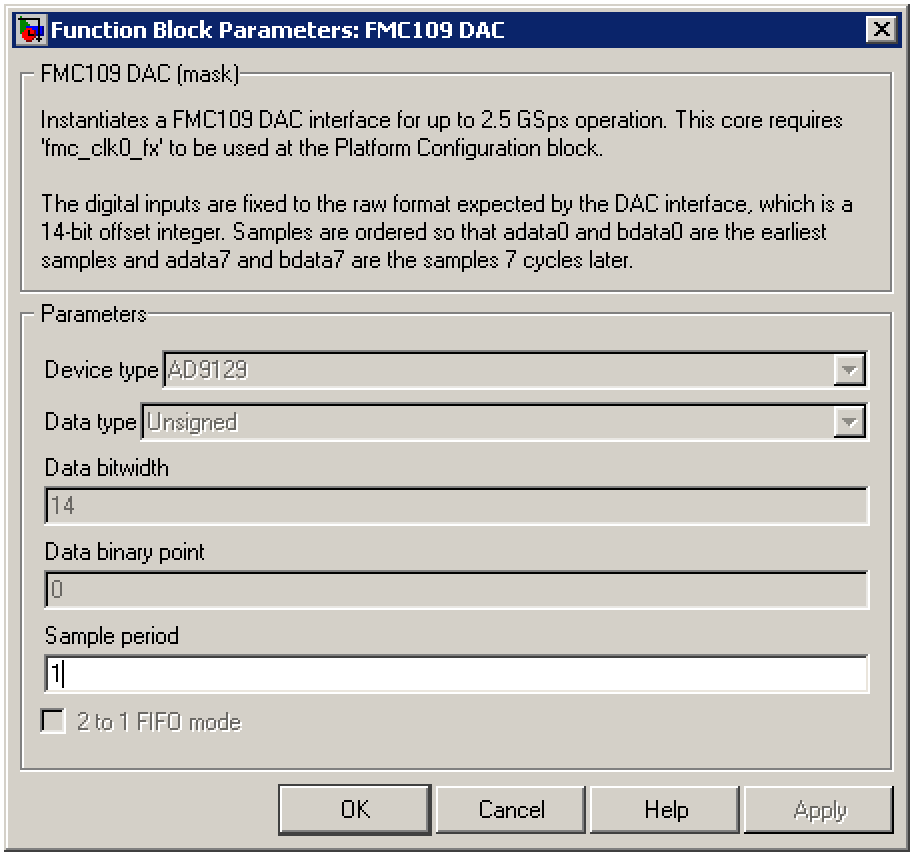

adata7 … adata0, bdata7 … bdata0
adata7_sim … adata0_sim, bdata7_sim … bdata0_sim
FMC109 DAC
adata15 … adata0, bdata15 … bdata0
adata15_sim … adata0_sim, bdata15_sim … bdata0_sim
FMC109 DAC 2to1

Figure 96: FMC109 DAC block

The DAC interface accepts 16 digital values per FPGA clock cycle (8 per device), which are driven by the DAC onto the analog output at 8x the system clock rate. The DAC devices each feature 2 independent data buses, which the FPGA drives with 4:1 SerDes devices. The resulting digital clock frequency is 1/8th the sampling rate. The DAC samples at its input clock rate. Therefore, the system clock rate must be 1/8th the native DAC digital clock rate. For example, if the DAC's input clock rate is 2.5 GHz, the FPGA system clock rate must be set to 312.5 MHz.

A 2-to-1 FIFO version of the DAC interface is also available. In that case, the interface accepts 16 samples per channel and the clock rate is 1/16th the native DAC digital clock rate. For example, if the DAC's input clock rate is 2.5 GHz, the FPGA system clock rate must be set to 156.25 MHz. Note, however, that the reference clock rate is still 312.5 MHz.

Note that the system clock rate must be sourced by the same reference clock as the sampling clock in order to prevent buffer under/overflow in the interface. This means that the DAC digital clock itself will be used as the reference clock source for the system. In Platform Configuration, this is the fmc_clk0_fx clock.



Function Block Parameters: FMC109 DAC

FMC109 DAC (mask)

Instantiates a FMC109 DAC interface for up to 2.5 GSps operation. This core requires 'fmc_clk0_fx' to be used at the Platform Configuration block.

The digital inputs are fixed to the raw format expected by the DAC interface, which is a 14-bit offset integer. Samples are ordered so that adata0 and bdata0 are the earliest samples and adata7 and bdata7 are the samples 7 cycles later.

Parameters

Device type: AD9129

Data type: Unsigned

Data bitwidth: 14

Data binary point: 0

Sample period: 1

☐ 2 to 1 FIFO mode

Figure 97: FMC109 DAC dialog

Table 68: FMC109 DAC block parameters

Parameter	**Description**
Device type	This parameter defines the type of DAC device present on the DAC expansion board.
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Unsigned* to match the unscaled DAC data inputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 14 bits to match the DAC data inputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the unscaled DAC data inputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.
2 to 1 FIFO mode	This check box indicates that this block is used in 2-to-1 FIFO mode, causing the DAC interface to run at half the speed but accept twice as many samples per clock. This parameter is not configurable directly; instead, drop the correct version of the block to achieve the desired level of parallelism.

Table 69: FMC109 DAC port definitions

Port	Direction	Bit Width	Description
(adata15_sim adata14_sim adata13_sim adata12_sim adata11_sim adata10_sim adata9_sim adata8_sim) adata7_sim adata6_sim adata5_sim adata4_sim adata3_sim adata2_sim adata1_sim adata0_sim	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the inputs below.
(bdata15_sim bdata14_sim bdata13_sim bdata12_sim bdata11_sim bdata10_sim bdata9_sim bdata8_sim) bdata7_sim bdata6_sim bdata5_sim bdata4_sim bdata3_sim bdata2_sim bdata1_sim bdata0_sim			

(`adata15` `adata14` `adata13` `adata12` `adata11` `adata10` `adata9` `adata8`) `adata7` `adata6` `adata5` `adata4` `adata3` `adata2` `adata1` `adata0`	in	14	Direct inputs to DAC “A”, representing 8 consecutive samples to be driven by the device. The samples are ordered with `adata0` being the earliest and `adata7` (or `adata15` for 2-to-1 FIFO mode) being the latest in temporal order. Data values must be raw, 14-bit, offset binary integers without any digital scaling or bias.
(`bdata15` `bdata14` `bdata13` `bdata12` `bdata11` `bdata10` `bdata9` `bdata8`) `bdata7` `bdata6` `bdata5` `bdata4` `bdata3` `bdata2` `bdata1` `bdata0`	in	14	Direct inputs to DAC “B”, representing 8 consecutive samples to be driven by the device. The samples are ordered with `bdata0` being the earliest and `bdata7` (or `bdata15` for 2-to-1 FIFO mode) being the latest in temporal order. Data values must be raw, 14-bit, offset binary integers without any digital scaling or bias.

FMC109 DAC block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC109 DAC block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc109_dac* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

```
fmc109_get_status <core>
```

The `fmc109_get_status` returns whether the FPGA has locked onto the clock sourced from the DAC.

```
fmc109_reset <core>
```

The `fmc109_reset` function asserts a reset to the DAC interface logic.

```
fmc109_set_delay <core> <A bit_mask> <B bit_mask> <odelay_val>
```

The `fmc109_set_delay` function will override the default delay values for each output pin with the value specified in the argument `odelay_val`. The `A bit_mask and B bit_mask` arguments are 29-bit values representing the 29 differential pairs each used for DAC "A" and DAC "B," respectively. The bitmask is the concatenation of the parity signal, data port 1, and data port 0 in little-endian order.

```
fmc109_reg_write <core> <channel <A | B | AB>> <reg_addr> <value>
```

The `fmc109_reg_write` function will write an arbitrary value into any one of the DAC's internal configuration registers. The address `reg_addr` is the address of the DAC configuration register as defined in the device datasheet, and `value` is the 8-bit value which should be written. The `channel` argument may be used to select the A channel, the B channel, or both.

```
fmc109_reg_read <core> <channel <A | B>> <reg_addr>
```

The `fmc109_reg_read` function will print values from any one of the DACs' internal configuration registers. The `channel` argument specifies which DAC to address (A or B). The `reg_addr` argument is the address of the DAC configuration register as defined in the device datasheet, and the function will print out the value stored within.

FMC110/111 Software Defined Radio

The FMC110/111 blocks provide a direct interface to the SDR device on the BEEcube FMC110/111 expansion board for the BEE4 and miniBEE4 hardware platforms.

FMC111 Control block

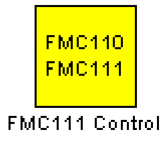
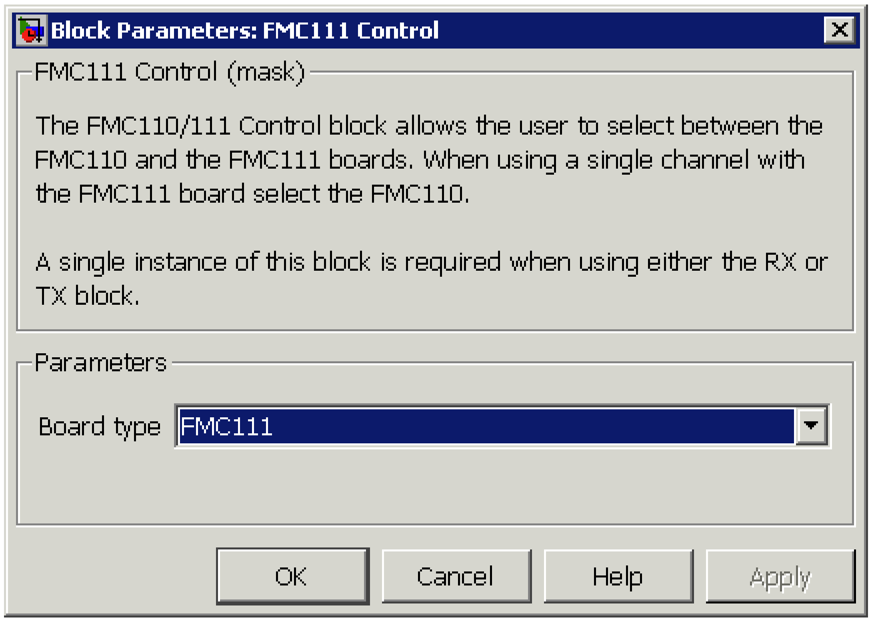


Figure 98: FMC111 Control block

The FMC110/111 Control block allows the user to select between the FMC110 and the FMC111 boards. If using a single channel with the FMC111 board, select the FMC110. A single instance of this block is required when using either the RX or TX block.


Block Parameters: FMC111 Control

FMC111 Control (mask)

The FMC110/111 Control block allows the user to select between the FMC110 and the FMC111 boards. When using a single channel with the FMC111 board select the FMC110.

A single instance of this block is required when using either the RX or TX block.

Parameters

Board type: FMC111

Figure 99: FMC111 Control dialog

Parameter	Description
Board type	Set to FMC111 if using two channels on an FMC111 board. If using a single channel with the FMC111 board, select the FMC110. Select FMC110 when using the FMC110 board.

FMC111 RX block

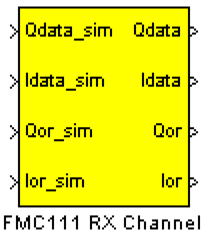
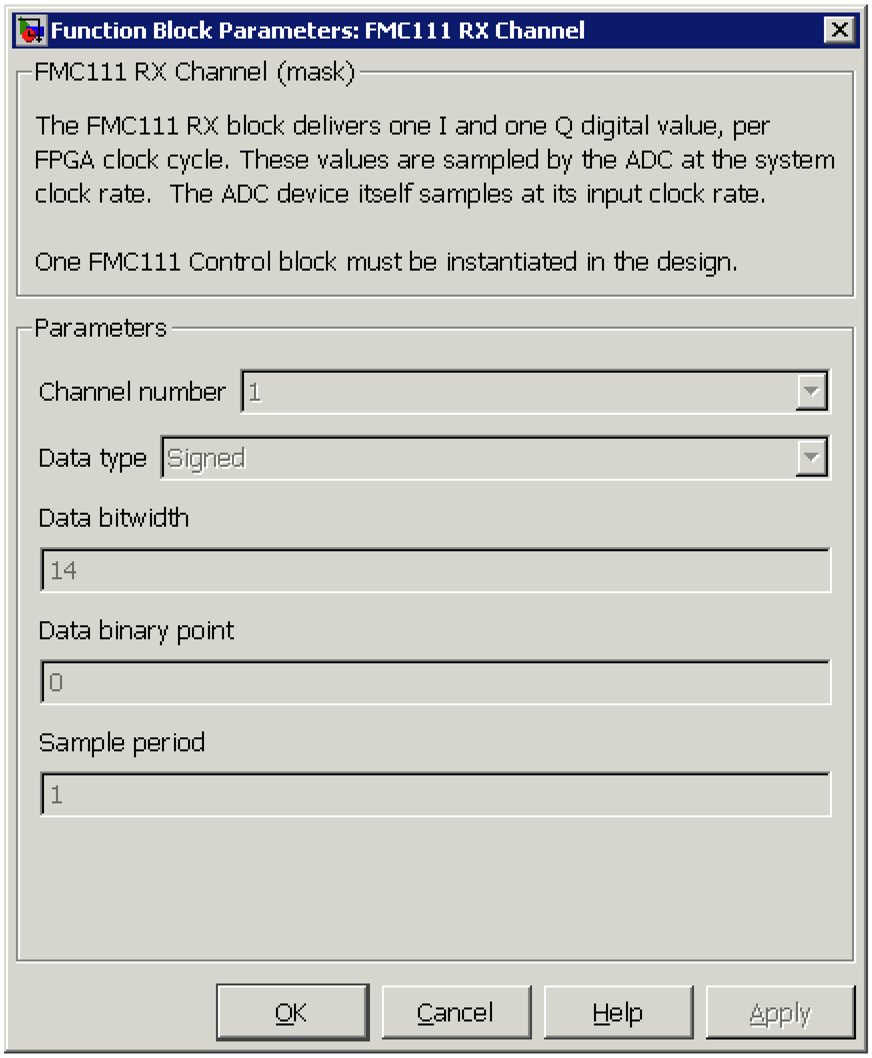


Figure 100: FMC111 RX block

The FMC111 RX block delivers one I and one Q digital value, per FPGA clock cycle. These values are sampled by the ADC at the system clock rate. The ADC device itself samples at its input clock rate. One FMC111 Control block must be instantiated in the design.



Function Block Parameters: FMC111 RX Channel

FMC111 RX Channel (mask)

The FMC111 RX block delivers one I and one Q digital value, per FPGA clock cycle. These values are sampled by the ADC at the system clock rate. The ADC device itself samples at its input clock rate.

One FMC111 Control block must be instantiated in the design.

Parameters

Channel number: 1

Data type: Signed

Data bitwidth: 14

Data binary point: 0

Sample period: 1

Figure 101: FMC111 RX dialog

Table 70: FMC111 RX block parameters

Parameter	Description
Channel number	Selects the channel of the block
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Signed* to match the ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 14 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 71: FMC111 RX port definitions

Port	Direction	Bit Width	Description
`Qdata_sim` `Idata_sim`	in	N/A	Inputs which can be used to provide simulation data within Simulink. Each input corresponds directly to each of the outputs below.
`Qor_sim` `Ior_sim`			
`Qdata` `Idata`	out	14	Direct outputs from the ADC, representing I and Q samples by the device. Data values are raw, 14-bit, signed binary integers without any digital scaling or bias.
`Qor` `Ir`	out	1	ADC out-of-range (OR) signal for the I and Q data samples. Indicates if an out-of-range input caused saturation in the ADC.

FMC111 TX block

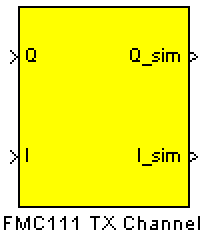
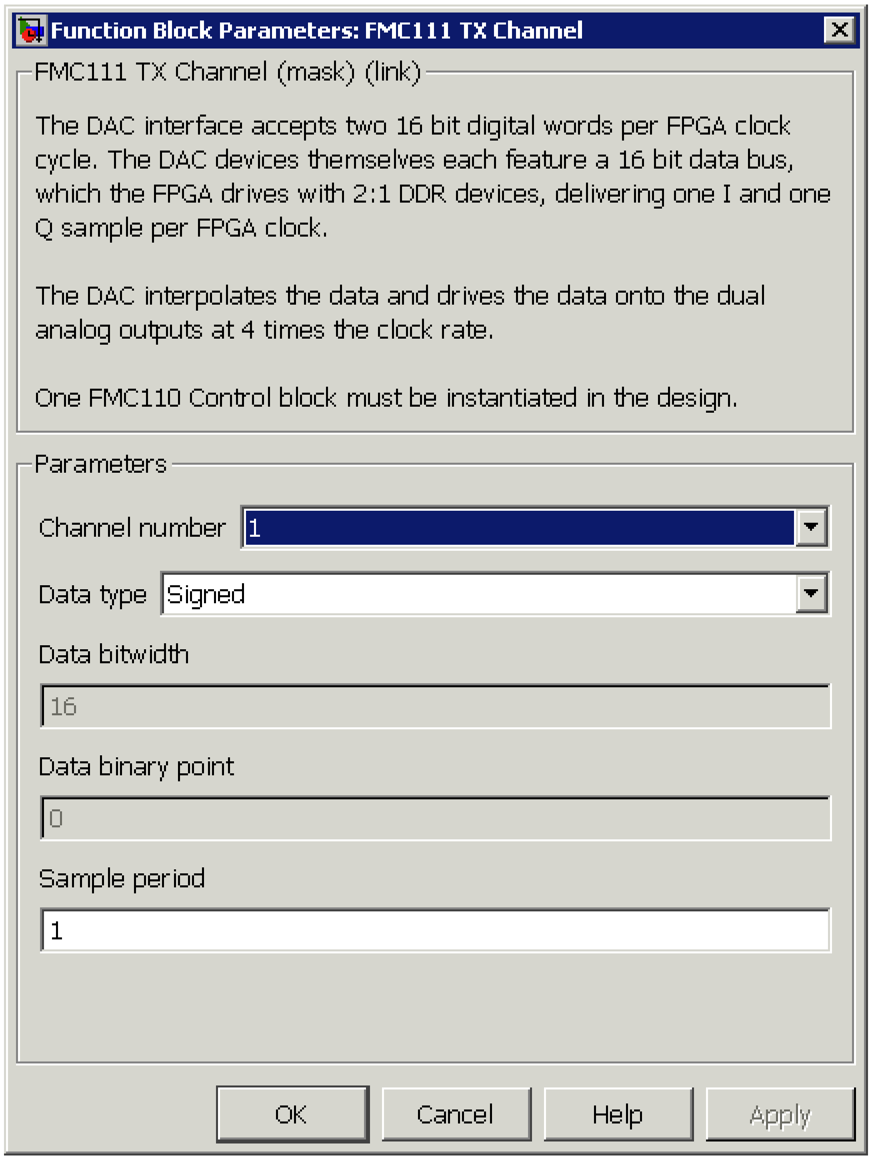


Figure 102: FMC111 TX block

The FMC111 TX interface accepts two 16 bit digital words per FPGA clock cycle. The DAC devices themselves each feature a 16 bit data bus, which the FPGA drives with a 2:1 DDR signal, delivering one I and one Q sample per FPGA clock. By default, the DAC interpolates the data by 2x and drives the data onto the dual analog outputs at 4 times the FPGA clock rate. One FMC111 Control block must be instantiated in the design.


Function Block Parameters: FMC111 TX Channel

FMC111 TX Channel (mask) (link)

The DAC interface accepts two 16 bit digital words per FPGA clock cycle. The DAC devices themselves each feature a 16 bit data bus, which the FPGA drives with 2:1 DDR devices, delivering one I and one Q sample per FPGA clock.

The DAC interpolates the data and drives the data onto the dual analog outputs at 4 times the clock rate.

One FMC110 Control block must be instantiated in the design.

Parameters

Channel number: 1

Data type: Signed

Data bitwidth: 16

Data binary point: 0

Sample period: 1

Figure 103: FMC111 TX dialog

Table 72: FMC111 TX block parameters

Parameter	**Description**
Channel number	Selects the channel of the block
Data type	This parameter determines the inferred arithmetic type for the signal. It is fixed to *Signed* to match the ADC data outputs.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 16 bits to match the ADC data outputs.
Data binary point	This value determines the inferred binary point position for the signal. It is fixed at zero to match the ADC data outputs.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 73: FMC111 TX port definitions

Port	**Direction**	**Bit Width**	**Description**
`Qdata_sim` `Idata_sim`	out	N/A	Outputs which can be used to collect simulation data within Simulink. Each output corresponds directly to each of the input below.
`Qdata` `Idata`	in	16	Direct inputs to the DAC, representing I and Q samples to be driven by the device. Data values are raw, 16-bit, signed binary integers without any digital scaling or bias.

FMC111 Control block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC111 Control block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc111_ctrl* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`fmc111_init`

The `fmc111_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. The init sets up the default states of all of the chips on the FMC110/111.

Repeated function calls

(none)

Commands

`fmc111_cntrl_led_toggle <on|off>`

The `fmc111_cntrl_led_toggle` will toggle the led on the FMC111 board.

`fmc111_cntrl_get_status`

The `fmc111_cntrl_get_status` reports if the user clock generated by the clock generator, RX local oscillator and TX local oscillators are locked.

`fmc111_set_ms_filt <TX|RX> <channel A|B|AB> <filter value>`

The `fmc111_set_ms_filt` sets the market specific filter. The filter value is defined in the table below.

`fmc111_cal_vosc <voltage>`

The `fmc111_cal_vosc` tunes the Vosc using a millivolt value from 0 to 3000. When using single PLL mode for the LMK this is needed to fine tune the VCXO if the reference frequency is off.

`fmc111_lo_set_freq <TX|RX> <channel A|B|AB> <frequency (kHz)>`

The `fmc111_lo_set_freq` sets the local oscillator's output frequency to the frequency defined in kilohertz.

```
fmc111_lo_sweep_freq <TX|RX> <channel A|B|AB> <start freq kHz> <stop
freq kHz> <step freq (kHz)> <wait time (us)>
```

The `fmc111_lo_sweep_freq` sweeps the local oscillator's output frequency. Wait time is how long in micro seconds to wait in between frequency steps.

```
fmc111_set_global_ref <none,sma,fpga> <input freq (khz) (optional)>
```

The `fmc111_set_global_ref` sets the reference frequency used to generate the reference clocks used throughout the fmc111. Options are either from the SMA clk input, the FPGA clk input, or none, which will use single PLL mode. In single PLL mode the user can fine-tune the output frequencies using `fmc111_cal_vosc` function.

```
fmc111_set_local_ref <dac/adc/lo/sma/fpga> <input freq (khz)>
```

The `fmc111_set_local_ref` sets the frequency that the specified device will use as its reference. For DAC/ADC it will set the sampling frequency, for the LO it will set the input reference frequency that is multiplied up (max input for Frac mode is ~60MHz). For the SMA, it refers to the SMA clk output and FPGA refers to the input clk to the FPGA.

```
fmc111_vga_set_gain <channel A|B|AB> <value>
```

The fmc111_vga_set_gain sets the variable gain amplifier in the RX chain from 4.5dB to 20.25dB in 0.25dB steps.

```
fmc111_set_attenuation <TX|RX> <channel A|B|AB> <value>
```

The `fmc111_set_attenuation` sets the attenuators from 2-31 in 0.5dB steps.

```
fmc111_chip_reset
<LMK|DAC1_LO|DAC2_LO|ADC_LO|ADC2_LO|ALL_LO|ALL_CHIPS>
```

The `fmc111_chip_reset` rewrites the cached values to the clock generator (LMK) and or the local oscillator registers.

```
fmc111_dac_reset
```

The `fmc111_dac_reset` triggers a reset of the DACs to their default configurations.

Market Specific Filters

Table 74: FMC111 TX and RX Market Specific Filters

Filter Select Value	Filter Bandwidth	Part Number
0	1.71 – 1.79 GHz	FAR-F6KA-1G7475-D4CY-Z
1	1.80 – 1.88 GHz.	FAR-F6KB-1G8425-B4GA-Z
2	1.85 – 1.91 GHz	FAR-F6KA-1G8800-L4AF-Z

3	1.92 – 1.98 GHz	FAR-F6KA-1G9500-D4DG-Z
4	1.93 – 1.99 GHz	FAR-F6KB-1G9600-B4GB-Z
5	2.11 – 2.17 GHz	FAR-F6KA-2G1400-D4CG-Z
6	2.30 – 2.385 GHz	SAFEA2G34FB0F00R14
7	2.40 – 2.50 GHz	B39252B9430M410
8	2.50 – 2.70 GHz	FI168B259762-T
9	0.40 – 0.80 GHz	FI168L062005
10	0.80 – 3.00 GHz	HFCN-740
11	3.10 – 4.90 GHz	4000BP15U1800E
12 - 15	4.90 – 5.90 GHz	5515BP15C1020E

FMC DVI

The FMC DVI blocks provide interfaces to the DVI input/output devices on the Xilinx/Avnet AES-FMC-DVI-G FMC module. Configuration of the FMC module is performed automatically by BPS upon programming or a soft reset of the FPGA. The hardware interface is presented to the user as a direct video stream using a standard frame/timing format and RGB color values.

FMC DVI Input block for BEE4, miniBEE4, and ML60x

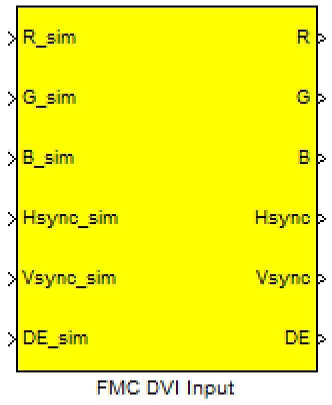
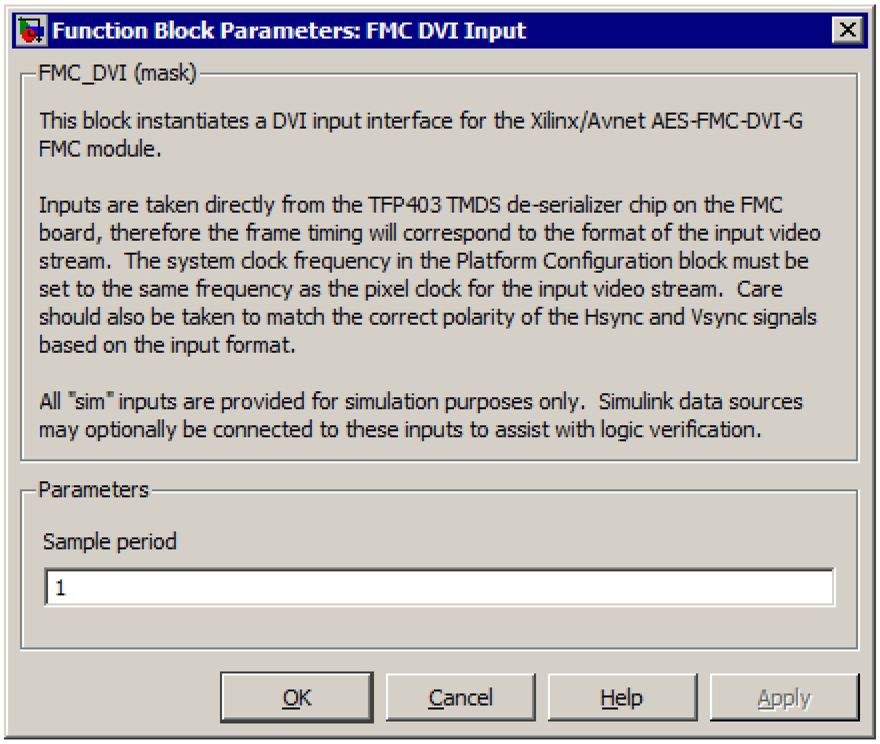


Figure 104: FMC DVI Input block



FMC_DVI (mask)

This block instantiates a DVI input interface for the Xilinx/Avnet AES-FMC-DVI-G FMC module.

Inputs are taken directly from the TFP403 TMDS de-serializer chip on the FMC board, therefore the frame timing will correspond to the format of the input video stream. The system clock frequency in the Platform Configuration block must be set to the same frequency as the pixel clock for the input video stream. Care should also be taken to match the correct polarity of the Hsync and Vsync signals based on the input format.

All "sim" inputs are provided for simulation purposes only. Simulink data sources may optionally be connected to these inputs to assist with logic verification.

Parameters

Sample period

1

Figure 105: FMC DVI Input dialog

The FMC DVI Input block adds a direct interface to the TFP403 TMDS de-serializer device on the FMC module. When this block is used in a system, the User IP Clock source must be set to `fmc_dvi_clk_fx` in the Platform Configuration block, as the inputs are sampled directly from the FMC module and are not buffered. The system clock rate must also be set to the same rate as the pixel clock which is expected from the external video source.

Table 75: FMC DVI Input block parameters

Parameter	**Description**
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 76: FMC DVI Input port definitions

Port	**Direction**	**Bit Width**	**Description**
`R` `R_sim`	out in	8 8	RGB Red video data. RGB Red video data simulation input for software emulation of external source.
`G` `G_sim`	out in	8 8	RGB Green video data. RGB Green video data simulation input.
`B` `B_sim`	out in	8 8	RGB Blue video data. RGB Blue video data simulation input.
`Hsync` `Hsync_sim`	out in	1 1	Horizontal sync, active high. Horizontal sync simulation input.
`Vsync` `Vsync_sim`	out in	1 1	Vertical sync, active high. Vertical sync simulation input.
`DE` `DE_sim`	out in	1 1	Data Enable, active high. Data Enable simulation input.

FMC DVI Output block for BEE4, miniBEE4, and ML60x

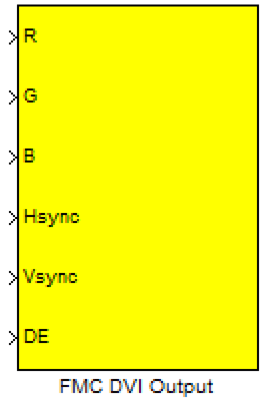
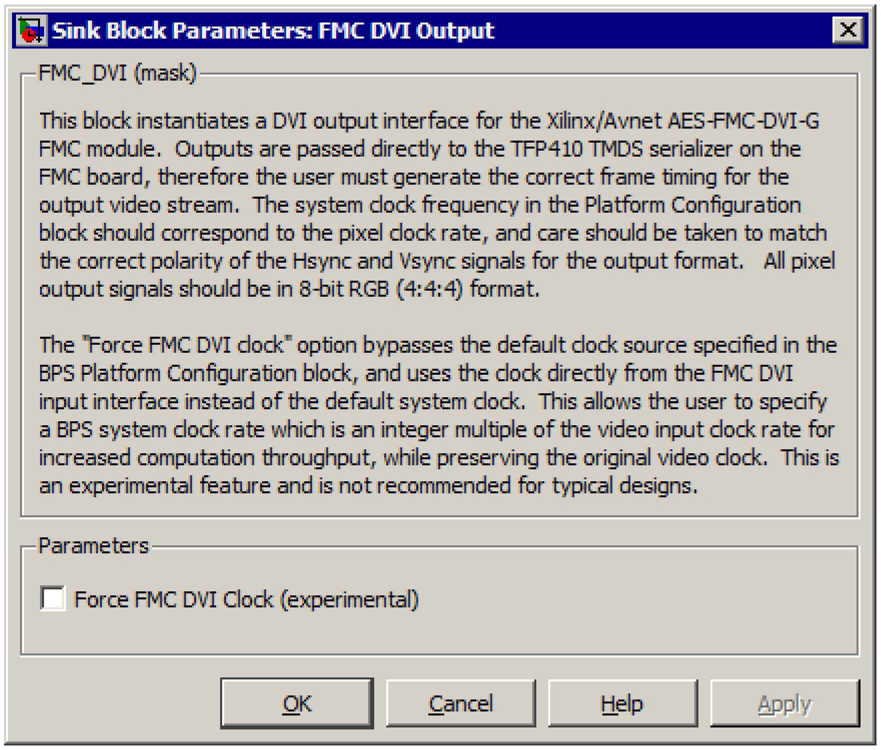


Figure 106: FMC DVI Output block


Sink Block Parameters: FMC DVI Output

FMC_DVI (mask)

This block instantiates a DVI output interface for the Xilinx/Avnet AES-FMC-DVI-G FMC module. Outputs are passed directly to the TFP410 TMDS serializer on the FMC board, therefore the user must generate the correct frame timing for the output video stream. The system clock frequency in the Platform Configuration block should correspond to the pixel clock rate, and care should be taken to match the correct polarity of the Hsync and Vsync signals for the output format. All pixel output signals should be in 8-bit RGB (4:4:4) format.

The "Force FMC DVI clock" option bypasses the default clock source specified in the BPS Platform Configuration block, and uses the clock directly from the FMC DVI input interface instead of the default system clock. This allows the user to specify a BPS system clock rate which is an integer multiple of the video input clock rate for increased computation throughput, while preserving the original video clock. This is an experimental feature and is not recommended for typical designs.

Parameters

☐ Force FMC DVI Clock (experimental)

Figure 107: FMC DVI Output dialog

The FMC DVI Output block adds a direct interface to the TFP410 TMDS serializer device on the FMC module. The system clock may either be taken directly from an FMC DVI Input component by setting the source to `fmc_dvi_clk_fx` in the Platform Configuration block (such as for streaming video processing applications), or if an FMC DVI Input block is not used in the system, the clock source may be set to any other value that corresponds to a valid pixel clock frequency for the video display device being used (such as for video generation applications).

Table 77: FMC DVI Output block parameters

Parameter	Description
Force FMC DVI Clock	If an FMC DVI Input block is used in the design, use its input clock instead of the system clock selected in the BPS platform configuration block for clocking the DVI signals. This allows the system logic to be “overclocked” with respect to input pixel clock. This feature is experimental and may not be guaranteed to work in all applications.

Table 78: FMC DVI Output port definitions

Port	Direction	Bit Width	Description
`R`	in	8	RGB Red video data.
`G`	in	8	RGB Green video data.
`B`	in	8	RGB Blue video data.
`Hsync`	in	1	Horizontal sync, active high.
`Vsync`	in	1	Vertical sync, active high.
`DE`	in	1	Data Enable, active high.

FMC DVI block NectarOS functions

BPS adds the following functions to NectarOS whenever an FMC DVI block is used in a design. The source code for these functions is contained in the *drivers/xps_fmc_dvi_in* and *drivers/xps_fmc_dvi_out* subdirectories of the BPS-generated EDK project directory.

Initialization function calls

`fmc_dvi_in_init`

The `fmc_dvi_in_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will configure an FMC DVI input interface for normal operation.

`fmc_dvi_out_init`

The `fmc_dvi_out_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will configure an FMC DVI output interface for normal operation.

Repeated function calls

(none)

Commands

(none)

General-purpose I/O

For all basic I/O pins which don’t require any higher-level protocols or device configuration, BPS provides a generic General Purpose I/O block. While all the user-configurable parameters for the GPIO block are the same for all hardware platforms, the list of available pin groups are unique. Therefore, separate versions of the GPIO block are included in the hardware-specific block libraries. The GPIO block does not include any software features.

GPIO block

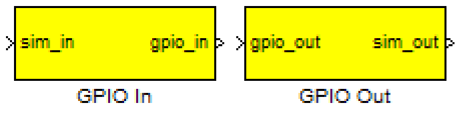
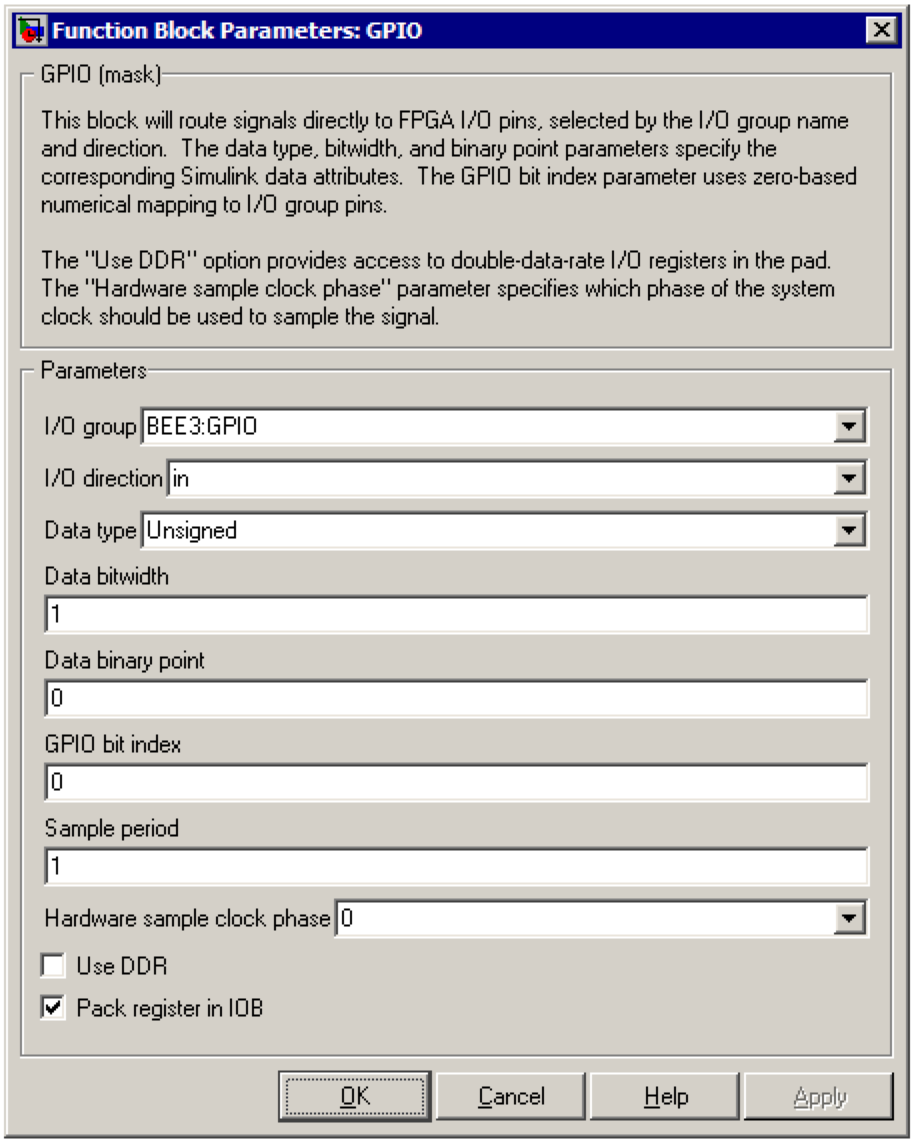


Figure 108: GPIO blocks



Function Block Parameters: GPIO

GPIO (mask)

This block will route signals directly to FPGA I/O pins, selected by the I/O group name and direction. The data type, bitwidth, and binary point parameters specify the corresponding Simulink data attributes. The GPIO bit index parameter uses zero-based numerical mapping to I/O group pins.

The "Use DDR" option provides access to double-data-rate I/O registers in the pad. The "Hardware sample clock phase" parameter specifies which phase of the system clock should be used to sample the signal.

Parameters

I/O group: BEE3:GPIO

I/O direction: in

Data type: Unsigned

Data bitwidth: 1

Data binary point: 0

GPIO bit index: 0

Sample period: 1

Hardware sample clock phase: 0

☐ Use DDR

☑ Pack register in IOB

Figure 109: GPIO dialog

The General Purpose I/O (GPIO) block connects signals directly to or from pins on the FPGA. It can be used either to communicate between FPGAs via on-board interconnect, or to access physical resources such as jumper pins or LEDs.

Bits in the input or output signal are mapped directly to pins in the specified I/O group according to the GPIO bit index parameter. For example, if the GPIO bit index parameter evaluates to the array [10, 11, 12, 13], a 4-bit signal will have its least-significant bit assigned to pin 10 in the I/O group and its most-significant bit assigned to pin 13. The length of the GPIO bit index parameter must be equal to the width of the input or output signal.

In addition to the data type and bit index parameters, each GPIO block can be set to use a given sampling clock phase and DDR signaling. For input GPIO blocks only, the hardware sample clock phase determines which phase of the system clock will be used to sample the pin input. For output GPIO blocks, the signal will always be driven on the rising edge of the system clock (a phase of zero).

When DDR signaling is enabled, the external data value will be driven or sampled at double the system clock rate (on both the rising and falling edges). The internal input or output signal will still operate at the system clock speed and be sampled on the rising edge of the system clock, and therefore must have twice the number of bits as the number of physical pins to maintain throughput. For example, an 8-pin DDR GPIO block must be connected to a 16-bit signal.

Table 79: GPIO block parameters

Parameter	**Description**
I/O group	This parameter defines which group of FPGA pins will be used to assign the GPIO signal. This field will list all the pin groups available on the hardware platform for direct pin assignment.
I/O direction	This parameter sets the direction of data flow for the signal. A direction of *in* represents an FPGA pin input, and a direction of *out* represents an output. Bidirectional (tri-state) pins are not supported. The I/O direction is pre-selected based on whether the block is a GPIO In or GPIO Out.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
GPIO bit index	This value represents the mapping between bits in the internal signal and pins in the FPGA I/O group. Each bit of the signal, starting from the LSB, is assigned to the corresponding pin of the I/O group given in this value. Each pin index is zero-based, and the length of this array must be equal to the width of the signal (or for DDR signaling, the length must be half the width of the signal).
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Sample clock phase	For input signals, this parameter determines which phase of the system clock (in degrees) will be used to sample the pin input. For output signals, this parameter is fixed at zero-phase, as outputs are always driven on the rising edge of the system clock.
Use DDR	When this option is enabled, DDR signaling will be used for the external signal (i.e. data will be driven or sampled on both the rising and falling edges of the sample clock). When DDR is enabled, the internal signal width must be twice the number of pins to maintain throughput.
Pack register in IOB	When this option is enabled, a register may be packed into the FPGA I/O buffer itself.

Table 80: GPIO port definitions

Port	**Description**
`sim_in` `gpio_in`	When direction is *in*: The `sim_in` port is intended for a standard Simulink data source to provide simulation data to emulate external stimuli. The `gpio_in` port drives a System Generator signal with the data type specified by the block.
`gpio_out` `sim_out`	When direction is *out*: The `gpio_out` port takes a System Generator signal as an input and converts it to a standard Simulink signal. The `sim_out` port can be used to capture simulation data with a Scope or any other Simulink data sink.

High-speed I/O

BPS provides support for high-speed data transfer between FPGAs on a single BEE3 or BEE4 board through the High-speed Input and Output blocks. Reliable operation is achieved via a physical hardware interface which is automatically calibrated during system initialization. On-demand re-calibrations can also be requested at runtime to account for changes in clock frequency (such as when external clock sources are used) or large variations in temperature.

Note that the High-speed Input and Output blocks are only intended be used in pairs: a High-speed Input block on a given I/O bus with a certain data bitwidth must be connected to a High-speed Output block on the same I/O bus with the same data bitwidth. Care should be taken when designing each FPGA implementation to be sure that the High-speed I/O interfaces on each side of a connection use the same parameters.

The remainder of this section describes the functionality of the High-speed Input and Output blocks in Simulink, as well as the complete software driver and runtime interfaces which are available for use in custom software designs.

High-speed Input block for BEE3 and BEE4

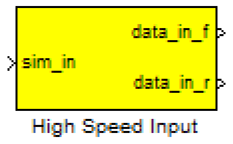
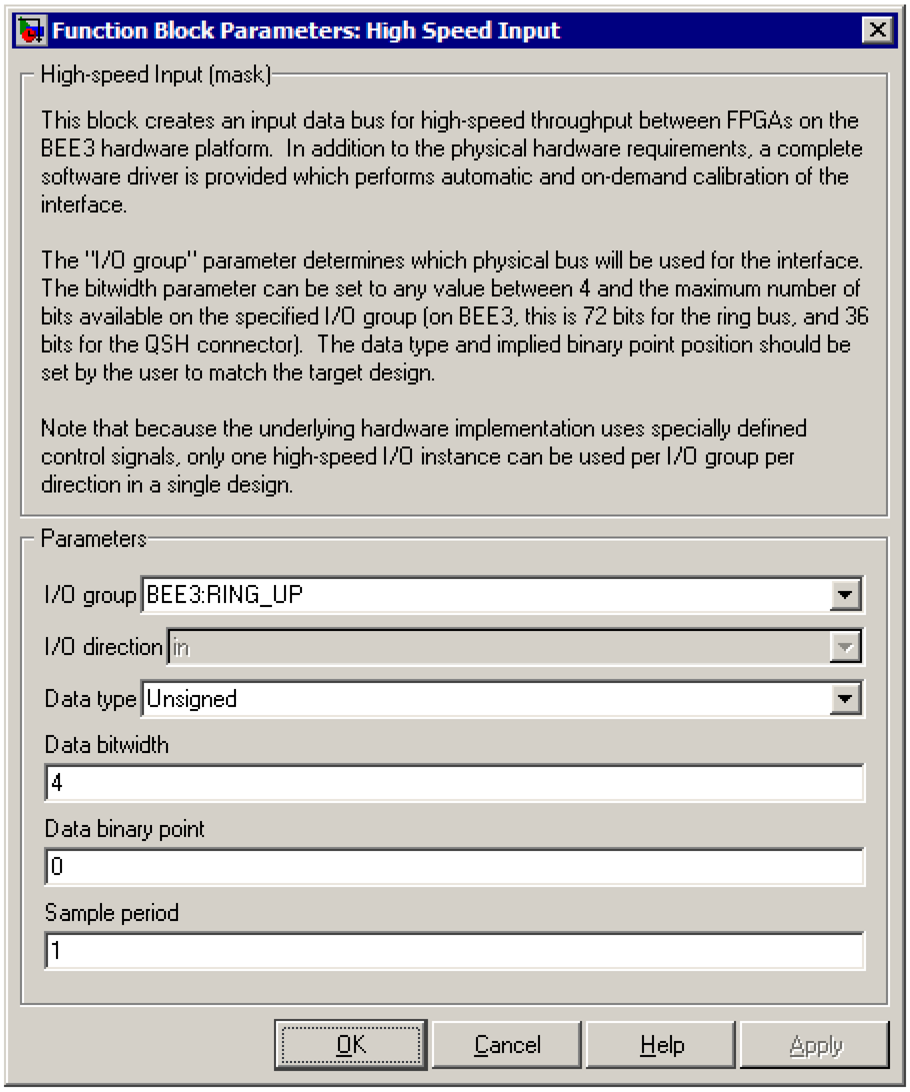


Figure 110: High-speed Input block


Function Block Parameters: High Speed Input

High-speed Input (mask)

This block creates an input data bus for high-speed throughput between FPGAs on the BEE3 hardware platform. In addition to the physical hardware requirements, a complete software driver is provided which performs automatic and on-demand calibration of the interface.

The "I/O group" parameter determines which physical bus will be used for the interface. The bitwidth parameter can be set to any value between 4 and the maximum number of bits available on the specified I/O group (on BEE3, this is 72 bits for the ring bus, and 36 bits for the QSH connector). The data type and implied binary point position should be set by the user to match the target design.

Note that because the underlying hardware implementation uses specially defined control signals, only one high-speed I/O instance can be used per I/O group per direction in a single design.

Parameters

I/O group: BEE3:RING_UP

I/O direction: in

Data type: Unsigned

Data bitwidth: 4

Data binary point: 0

Sample period: 1

Figure 111: High-speed Input dialog for BEE3 (BEE4's is similar)

The High-speed Input block provides an interface for receiving data between FPGAs. Because the underlying hardware implementation uses specific pins on the FPGA, only one instance per I/O group can be present in a single design. Multiple logical signals can always be packed into a single High-speed Input block and separated from each other using Slice blocks from the Xilinx System Generator Blockset. Similarly, the High-speed Input block cannot be combined with standard GPIO blocks assigned to the same I/O group.

Table 81: High-speed Input block parameters

Parameter	Description
I/O group	This parameter defines which group of FPGA pins will be used to assign the signal. This field will list all the pin groups available on the hardware platform for direct pin assignment.
I/O direction	Since the direction of signal flow is pre-determined for each end of the High-speed I/O interface to support automated calibration, this value is fixed to *in* for the input block.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. This can be any integer between 4 (the minimum width imposed by the underlying hardware implementation) and the maximum number of bits available for the given I/O group.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 82: High-speed Input port definitions

Port	Direction	Bit Width	Description
`data_in_f` `data_in_r`	out	*N*	Output ports which drive a hardware signal into the design with the System Generator signal parameters defined in the block configuration dialog (shown above). Note that because the High-speed Input block always uses DDR (double data rate) signaling, two data values are driven per system clock cycle. The `*_r` output represents the data latched on the rising clock edge, and the `*_f` output represents the data latched on the falling clock edge.
`sim_in`	in	N/A	Input port which can be used to provide simulation data within Simulink. Note that this Simulink input signal is valid only during simulation and has no effect on the hardware. The sample time of this input should be half that of the hardware system (i.e. twice the sample rate).

High-speed Output block for BEE3 and BEE4

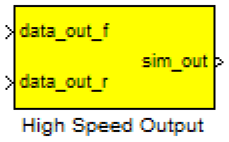
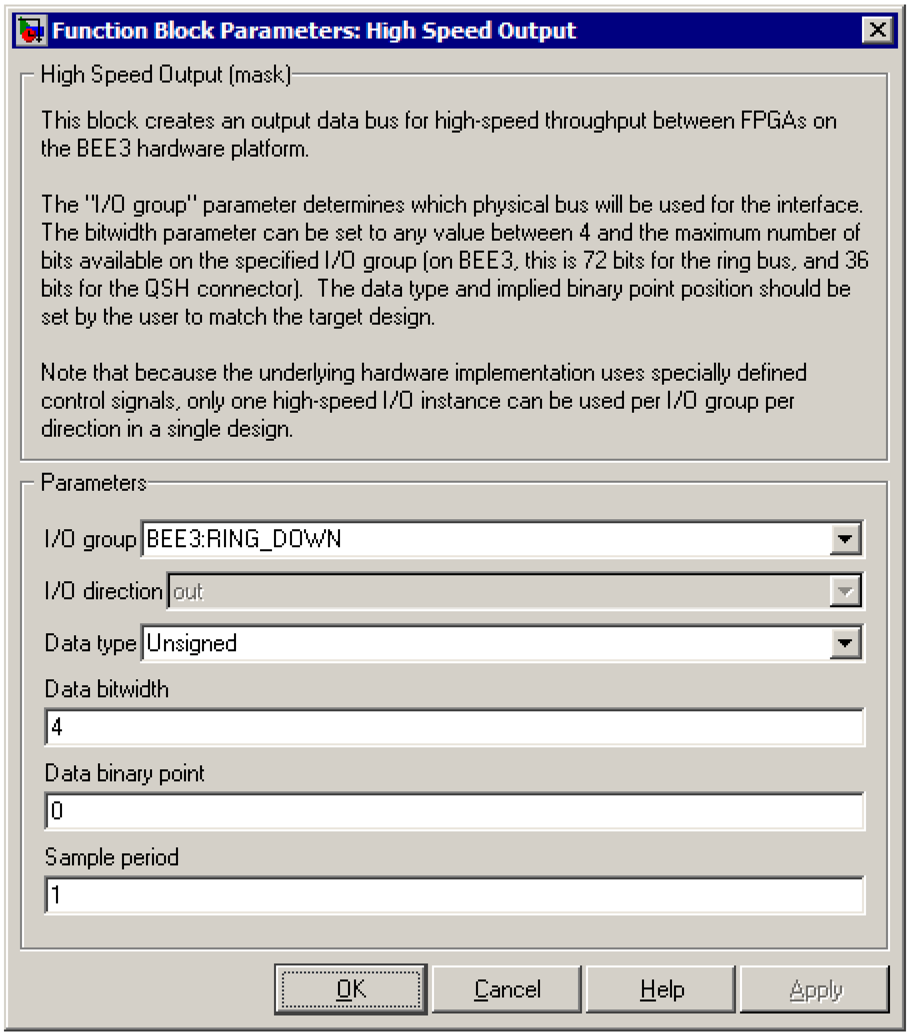


Figure 112: High-speed Output block



Function Block Parameters: High Speed Output

High Speed Output (mask)

This block creates an output data bus for high-speed throughput between FPGAs on the BEE3 hardware platform.

The "I/O group" parameter determines which physical bus will be used for the interface. The bitwidth parameter can be set to any value between 4 and the maximum number of bits available on the specified I/O group (on BEE3, this is 72 bits for the ring bus, and 36 bits for the QSH connector). The data type and implied binary point position should be set by the user to match the target design.

Note that because the underlying hardware implementation uses specially defined control signals, only one high-speed I/O instance can be used per I/O group per direction in a single design.

Parameters

I/O group: BEE3:RING_DOWN

I/O direction: out

Data type: Unsigned

Data bitwidth: 4

Data binary point: 0

Sample period: 1

Figure 113: High-speed Output dialog

The High-speed Output block provides an interface for sending data between FPGAs. Because the underlying hardware implementation uses specific pins on the FPGA, only one instance per I/O group can be present in a single design. Multiple logical signals can always be packed into a single High-speed Output block and using Concat blocks from the Xilinx System Generator Blockset. Similarly, the High-speed Output block cannot be combined with standard GPIO blocks assigned to the same I/O group.

Table 83: High-speed Output block parameters

Parameter	**Description**
I/O group	This parameter defines which group of FPGA pins will be used to assign the signal. This field will list all the pin groups available on the hardware platform for direct pin assignment.
I/O direction	Since the direction of signal flow is pre-determined for each end of the High-speed I/O interface to support automated calibration, this value is fixed to *out* for the output block.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. This can be any integer between 4 (the minimum width imposed by the underlying hardware implementation) and the maximum number of bits available for the given I/O group.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 84: High-speed Output port definitions

Port	**Direction**	**Bit Width**	**Description**
`data_out_f` `data_out_r`	in	*N*	Input ports which accept a hardware signal from the design with the System Generator signal parameters defined in the block configuration dialog (shown above). Note that because the High-speed Output block always uses DDR (double data rate) signaling, two data values are required per system clock cycle. The `*_r` output represents the data to be sent on the rising clock edge, and the `*_f` output represents the data to be sent on the falling clock edge.
`sim_out`	out	N/A	Output port which can be used to provide simulation data within Simulink. Note that this Simulink output signal is valid only during simulation and has no effect on the hardware. The sample time of this output will be half that of the hardware system (i.e. twice the sample rate).

High-speed I/O NectarOS functions

BPS adds the following functions to NectarOS whenever a High-speed Input block is used in a design (the High-speed Output block is managed purely in hardware and does not require a software interface). The source code for these functions is contained in the *drivers/xps_hsio* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`hsio_rx_init`

The `hsio_rx_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will automatically calibrate the hardware interface for guaranteed reliability at the current system frequency and temperature. If the clock frequency is changed (such as if an external clock source is being used) or if an extreme temperature change is suspected, the interface can be manually re-calibrated via the `hsio_calibrate_delays` command, which is described below. Note that this function requires that the transmit side of the high-speed interface is active before calibration will succeed. The function will continuously retry the calibration for approximately one minute before giving up (which can be necessary when the FPGAs in a system are programmed sequentially), after which the user will need to manually force a re-calibration.

Repeated function calls

(none)

Commands

`hsio_print_delays <core_name>`

The `hsio_print_delays` function will print a table of the previous calibration results. The `core_name` argument should be the name of the High-speed Input block in BPS (a list of known cores can be viewed via the `listdev` command).

`hsio_calibrate_delays <core_name> <usec_window>`

The `hsio_calibrate_delays` function will force a re-calibration of the hardware interface. The `core_name` argument should be the name of the High-speed Input block in BPS (a list of known cores can be viewed via the `listdev` command). The `usec_window` argument should be an integer value indicating the number of microseconds to wait before checking the synchronization status of each pin. Larger values can result in greater accuracy, but with a longer run time per calibration. The default value is 1000 microseconds.

```
hsio_set_mask <core_name>
              <data_0_31_mask> <data_32_63_mask> <data_64_71_mask>
              <strobe_mask>
```

The `hsio_set_mask` function will set the pin select mask bits which are used to control all manual input delay operations. It is intended to be used only in conjunction with the `hsio_set_delays` command to manually set the input pin delay for debugging or validation purposes. Each of the mask arguments corresponds to a specific range of bits on the physical bus. Each mask is mapped to the bus in big-endian bit order (i.e. x80000000 corresponds to the first bit of the mask range). The `core_name` argument should be the name of the High-speed Input block in BPS (a list of known cores can be viewed via the `listdev` command).

```
hsio_set_delays <core_name> <idelay_val>
```

The `hsio_set_delays` function will set all pins with their mask bits enabled to the fixed input delay value `idelay_val`. It is intended to be used only in conjunction with the `hsio_set_mask` command for debugging or validation purposes. The `core_name` argument should be the name of the High-speed Input block in BPS (a list of known cores can be viewed via the `listdev` command).

High-speed I/O EDK API functions

The underlying hardware/software interface for the High-speed I/O component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the High-speed I/O interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE3 base package. Include the header *xhsio.h* for access to the API.

Data Structures

There are three data structures used by several of the High-speed I/O EDK driver function calls which correspond to all the software-accessible control and status bits contained in the hardware interface logic. Each of these data structures are described in the following tables.

Table 85: `XHsio_Config` Structure Contents

Variable name	**Type**	**Definition**
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base address of hardware instance
`data_width`	Unsigned 8 bit	Data bus width of hardware instance

Table 86: `XHsio_Control` Structure Contents

Variable name	**Type**	**Definition**
`cal_enable`	Unsigned 8 bit	Enable interface calibration mode
`idelayctrl_rst`	Unsigned 8 bit	IDELAYCTRL reset
`tpsync_rst`	Unsigned 8 bit	Test pattern synchronization state reset
`swap_output`	Unsigned 8 bit	DDR data output order swap

Table 87: `XHsio_Status` Structure Contents

Variable name	**Type**	**Definition**
`data_0_31_sync`	Unsigned 32 bit	Data bits 0 to 31 synchronization state
`data_32_63_sync`	Unsigned 32 bit	Data bits 32 to 63 synchronization state
`data_64_71_sync`	Unsigned 8 bit	Data bits 64 to 71 synchronization state
`strobe_sync`	Unsigned 8 bit	Strobe signal synchronization state

Functions

```
XHsio_Config *
XHsio_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor PLB bus. The return value of this function should be used as the `XHsio_Handle` argument required by other driver routines.

```
XStatus
XHsio_SetControl(XHsio_Handle handle ,
                 XHsio_Control * Control);
```

Writes the set of control signal values contained in the structure referenced by pointer `Control` into the device control register. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_GetControl(XHsio_Handle handle,
                 XHsio_Control * Control);
```

Reads the device control register and fills the corresponding values in the structure referenced by the pointer `Control`. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_GetStatus(XHsio_Handle handle,
                XHsio_Status * Status);
```

Reads the device status register and fills the corresponding values in the structure referenced by the pointer `Status`. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_ResetIdelays();
```

Triggers a reset of all IDELAYCTRL elements contained in any High-speed Input hardware instances. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_ResetIdelay(XHsio_Handle handle);
```

Triggers the reset of the IDELAYCTRL elements contained in a single High-speed Input hardware instance. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_SetIdelayPinMask(XHsio_Handle handle,
                       Xuint32 data_0_31_pin_mask,
                       Xuint32 data_32_63_pin_mask,
                       Xuint8 data_64_71_pin_mask,
                       Xuint8 strobe_pin_mask);
```

Sets the input pin select masks used for all future calls to `XHsio_SetIdelayValue`. Each of the mask arguments correspond to a different set of pins in the hardware interface, with each mask bit mapped to a pin in big-endian order (i.e. 0x40000000 maps to bus pin 33 for the `data_32_63_pin_mask` argument). The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_SetIdelayValue(XHsio_Handle handle,
                     Xuint8 delay_val);
```

Sets the new input delay value for all pins with their select map bits enabled to the value `delay_val`. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_CalibrateAllIdelays(XHsio_Handle handle,
                          Xuint16 usec_window);
```

Automatically calibrates the delay of all input pins via the embedded test pattern synchronization mode. The argument `usec_window` determines how many microseconds the processor will wait before reading the synchronization status values on each iteration. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XHsio_GetCalibrationState(XHsio_Handle handle,
                          XHsio_Status *sync_status_p,
                          Xuint8 *idelay_values);
```

Copies the results of the previous calibration and all current delay values into the data structure arrays referenced by each argument pointer. The driver header file defines the constants `XPAR_XHSIO_IDELAY_DEPTH` and `XPAR_XHSIO_MAX_DATA_WIDTH`, which can be used as the size of the arrays `sync_status_p` and `idelay_values`, respectively. The argument `handle` should be a configuration structure pointer as returned by the `XHsio_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

Multi-Port Memory Controller

BPS supports the use of DDR2 DRAM running at DDR400 speed, which can be shared by both hardware and software via a multi-port memory controller (MPMC). The memory controller architecture for all hardware platforms is based on the Xilinx MPMC, which is included as part of EDK. For precise details on the behavior of the memory controller, please refer to the MPMC documentation provided with EDK. The physical MPMC implementations are highly hardware dependent, therefore separate MPMC blocks are provided in the hardware-specific block libraries. The individual hardware ports provided by BPS, however, function identically regardless of the underlying hardware platform.

A large variety of different memory configurations are supported by BPS, based on the top-level MPMC parameters selected by the designer and the number of type of hardware ports used in the design. Each of the different blocks which are available for accessing DRAM with BPS are described in detail in this section.

Memory Controller block for BEE3

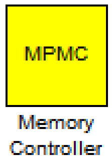
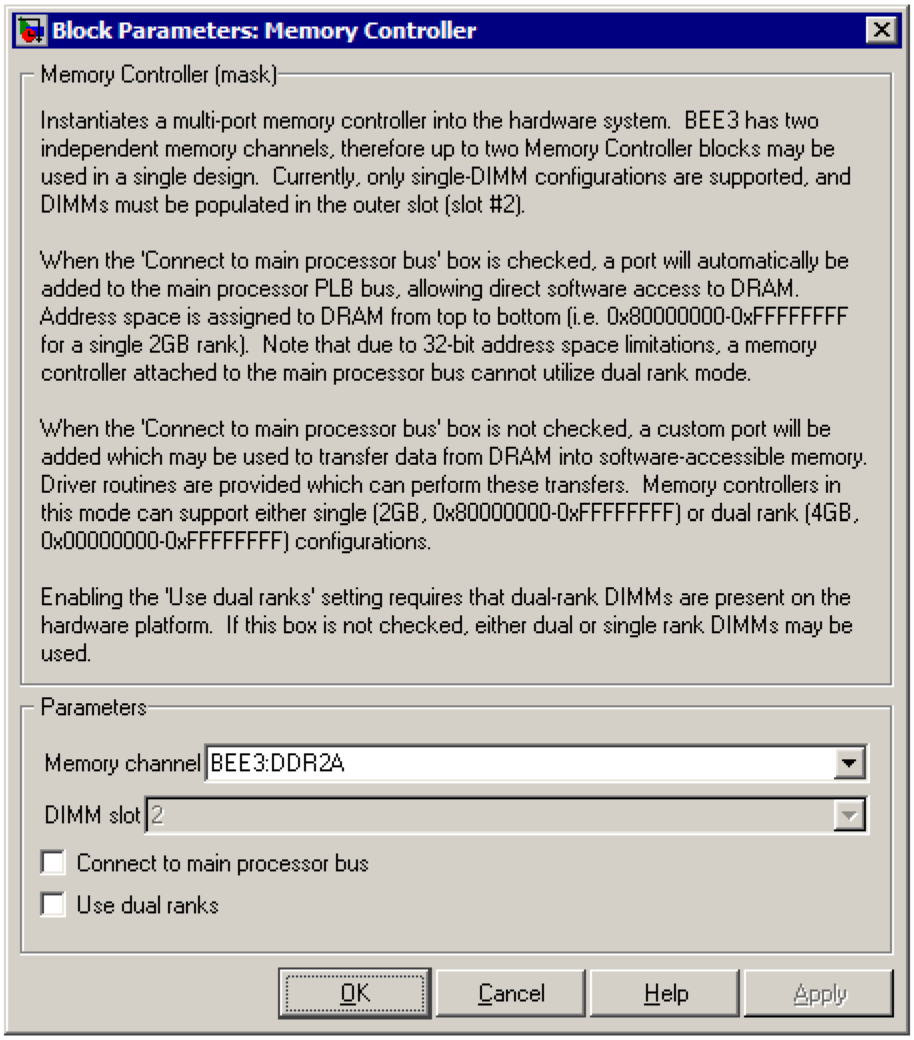


Figure 114: Memory Controller block for BEE3



Memory Controller (mask)

Instantiates a multi-port memory controller into the hardware system. BEE3 has two independent memory channels, therefore up to two Memory Controller blocks may be used in a single design. Currently, only single-DIMM configurations are supported, and DIMMs must be populated in the outer slot (slot #2).

When the 'Connect to main processor bus' box is checked, a port will automatically be added to the main processor PLB bus, allowing direct software access to DRAM. Address space is assigned to DRAM from top to bottom (i.e. 0x80000000-0xFFFFFFFF for a single 2GB rank). Note that due to 32-bit address space limitations, a memory controller attached to the main processor bus cannot utilize dual rank mode.

When the 'Connect to main processor bus' box is not checked, a custom port will be added which may be used to transfer data from DRAM into software-accessible memory. Driver routines are provided which can perform these transfers. Memory controllers in this mode can support either single (2GB, 0x80000000-0xFFFFFFFF) or dual rank (4GB, 0x00000000-0xFFFFFFFF) configurations.

Enabling the 'Use dual ranks' setting requires that dual-rank DIMMs are present on the hardware platform. If this box is not checked, either dual or single rank DIMMs may be used.

Parameters

Memory channel: BEE3:DDR2A

DIMM slot: 2

☐ Connect to main processor bus

☐ Use dual ranks

Figure 115: Memory Controller dialog for BEE3

The BEE3 Memory Controller block will instantiate a DDR2 DRAM MPMC in the design tailored to the ECC RDIMM configuration used on BEE3 with either single or dual rank DIMM modules. Since the BEE3 platform features two DDR2 memory channels, up to two Memory Controller blocks may be used in a single design, each controlling a different memory channel. The MPMC can also optionally be connected to the main processor bus for direct software access. If not connected to the main processor bus, the total addressable DRAM size increases from 2GB to 4GB, however software access to DRAM is limited to 32-word bursts provided by a custom interface created by BPS (information on how to program with this interface is included in the EDK driver section below).

Table 88: Memory Controller block parameters for BEE3

Parameter	Description
Memory channel	This parameter defines which memory channel the MPMC will be attached to. On BEE3, the channel can be either *BEE3:DDR2A* for DRAM channel A, or *BEE3:DDR2B* for DRAM channel B.
DIMM slot	This parameter determines which DIMM slot the MPMC will control. Currently, only DIMM slot 2 is supported by BPS.
Connect to main processor bus	When checked, the memory controller is limited to a single rank configuration, with 2GB mapped directly into the processor's address space. Software can access DRAM using any address in the range 0x80000000-0xFFFFFFFF. When not checked, software cannot access DRAM directly in the memory address space. However, BPS automatically adds a custom port which can read/write 32-word bursts into and out of DRAM. A driver is provided for this custom interface, and is described later in this section.
Use dual ranks	When checked, the memory controller will be configured to use the entire 4GB range available on dual-rank DIMM modules. Due to the address space limitations mentioned above, this option is only available if the memory controller is not connected to the main processor bus. Note that dual-rank DIMM modules will still function properly if this box is not checked. However, this box should never be checked if only single-rank DIMM modules are present on the hardware, or else the memory channel will not be functional.

Memory Controller block for ML50x

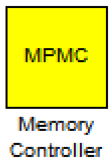
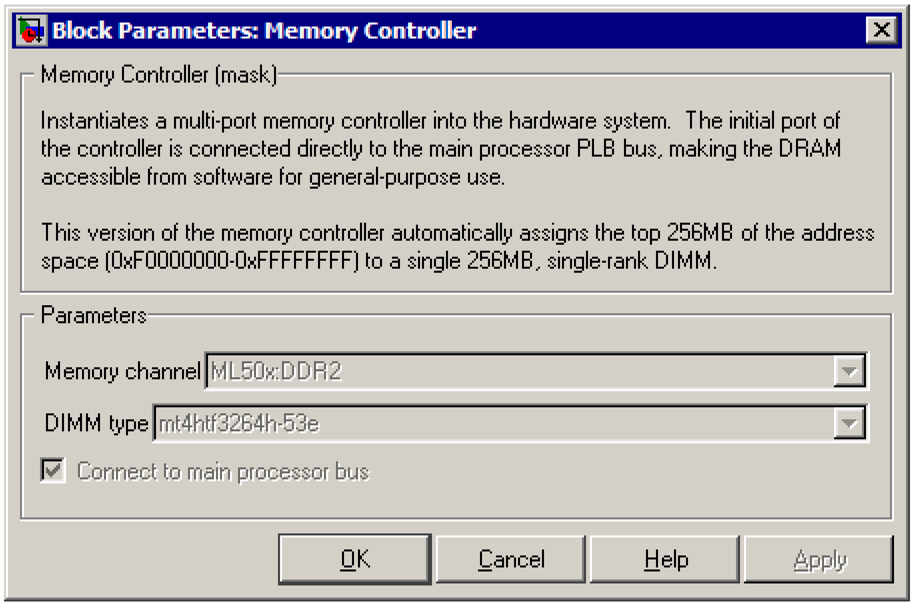


Figure 116: Memory Controller block for ML50x



Memory Controller (mask)

Instantiates a multi-port memory controller into the hardware system. The initial port of the controller is connected directly to the main processor PLB bus, making the DRAM accessible from software for general-purpose use.

This version of the memory controller automatically assigns the top 256MB of the address space (0xF0000000-0xFFFFFFFF) to a single 256MB, single-rank DIMM.

Parameters

Memory channel: ML50x:DDR2

DIMM type: mt4htf3264h-53e

☑ Connect to main processor bus

Figure 117: Memory Controller dialog for ML50x

The ML50x Memory Controller block will instantiate a DDR2 DRAM multi-port memory controller (MPMC) in the design tailored to the standard DIMM configuration used by default on the ML50x platform. The MPMC includes a physical interface to DRAM as well as a bridge to the main processor PLB bus, which is assigned to the top 256MB of the processor address space. Only one Memory Controller block is supported per FPGA design.

Table 89: Memory Controller block parameters for ML50x

Parameter	**Description**
Memory channel	Since the ML50x platform only has a single memory channel, this option is fixed to *ML50x:DDR2*.
DIMM type	This parameter is currently fixed to the standard DIMM type provided with ML50x boards and cannot be changed.
Connect to main processor bus	Since the 256MB DIMM modules provided with the ML50x board does not have any address space consequences, this option is automatically enabled.

MPMC Direct block

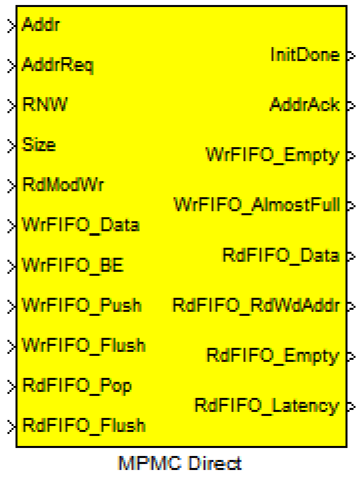
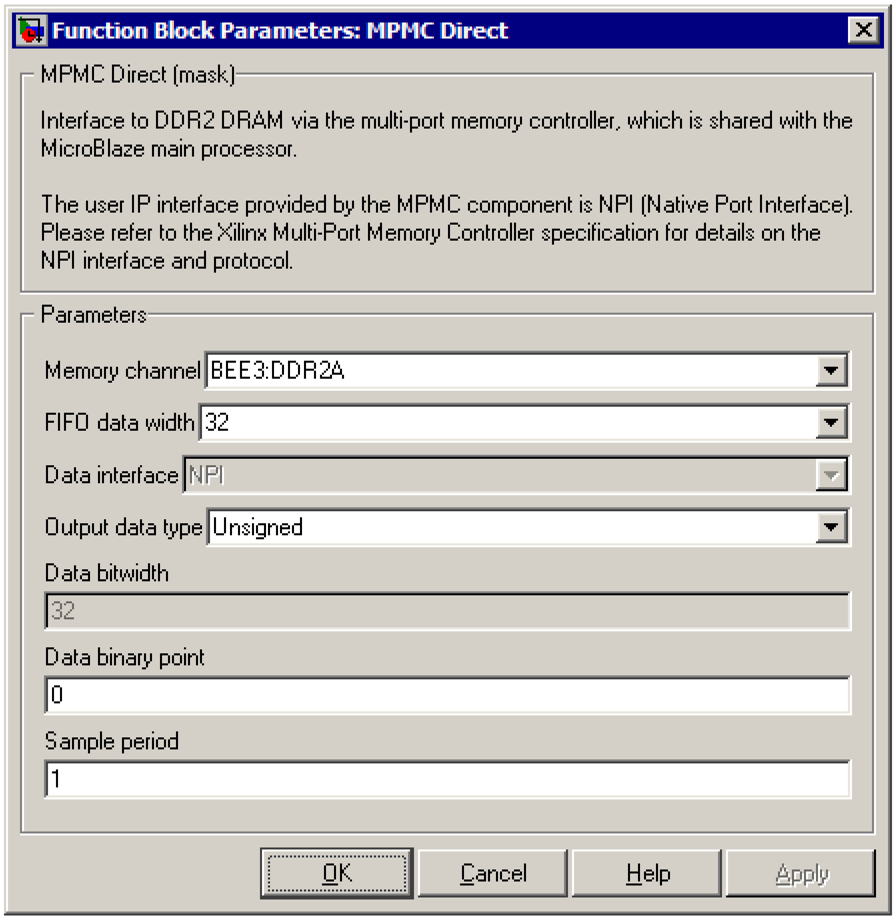


Figure 118: MPMC Direct block



MPMC Direct (mask)

Interface to DDR2 DRAM via the multi-port memory controller, which is shared with the MicroBlaze main processor.

The user IP interface provided by the MPMC component is NPI (Native Port Interface). Please refer to the Xilinx Multi-Port Memory Controller specification for details on the NPI interface and protocol.

Parameters

Memory channel: BEE3:DDR2A

FIFO data width: 32

Data interface: NPI

Output data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Sample period: 1

Figure 119: MPMC Direct dialog

The MPMC Direct block will add a hardware-accessible port to an existing MPMC in a design. This implies that a Memory Controller block must exist in the system for an MPMC Direct block to be valid. The MPMC architecture allows up to a total of 8 ports (this includes any ports reserved by the base components, which is dependent on the platform configuration settings and other peripherals). Note that due to restrictions enforced by the MPMC, any design which uses an MPMC Direct block must run at 200MHz, which is the native frequency of the MPMC itself. If an asynchronous interface is needed so that the system clock can run at a frequency other than 200MHz, a video frame buffer port should be used instead, which is described below.

The MPMC Direct block directly exposes the Native Port Interface (NPI) ports used by the MPMC. For detailed information on the NPI protocol, including the complete port specification, please refer to the MPMC documentation provided with Xilinx EDK.

Table 90: MPMC Direct block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
FIFO data width	This parameter can be set to either 32 or 64, and determines the width of the data ports into and out of the MPMC.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Data interface	This parameter defines the port interface and protocol for connecting to the MPMC. This parameter is restricted to the NPI protocol.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

MPMC Basic block

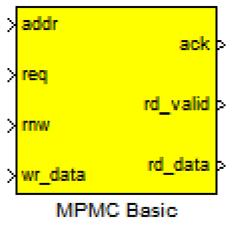
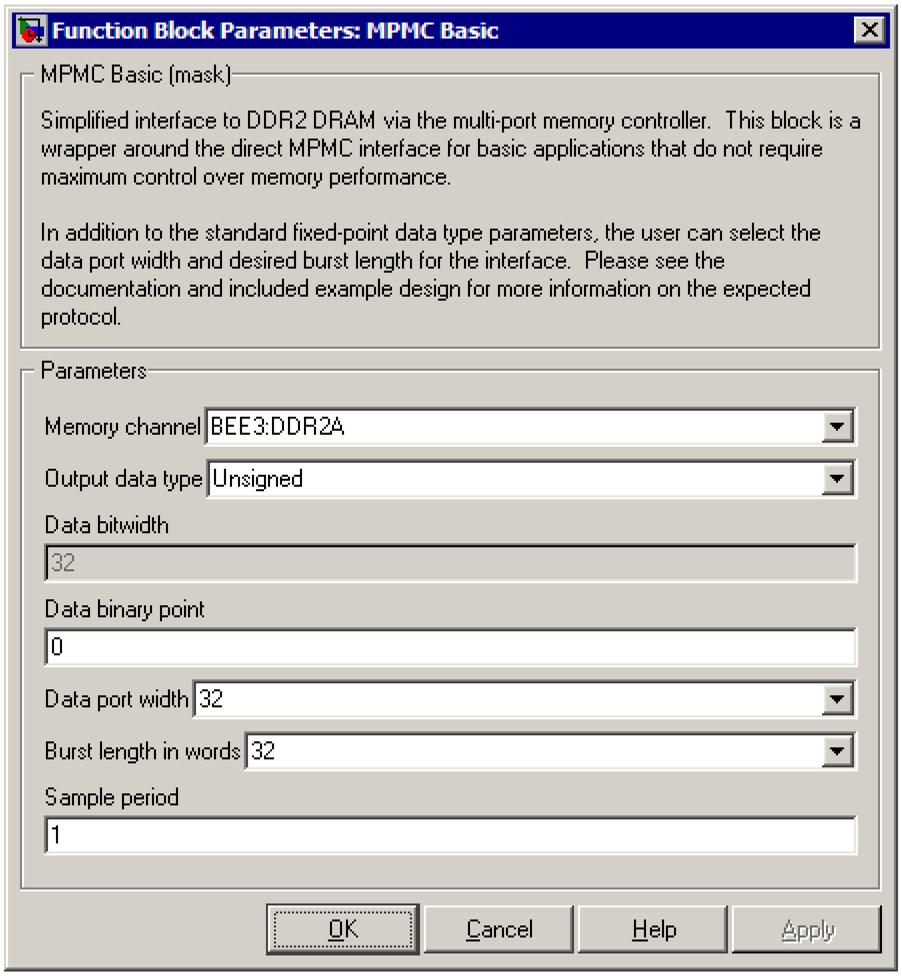


Figure 120: MPMC Basic block



MPMC Basic (mask)

Simplified interface to DDR2 DRAM via the multi-port memory controller. This block is a wrapper around the direct MPMC interface for basic applications that do not require maximum control over memory performance.

In addition to the standard fixed-point data type parameters, the user can select the data port width and desired burst length for the interface. Please see the documentation and included example design for more information on the expected protocol.

Parameters

Memory channel: BEE3:DDR2A

Output data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Data port width: 32

Burst length in words: 32

Sample period: 1

Figure 121: MPMC Basic dialog

The MPMC Basic block is a wrapper around the MPMC Direct interface which provides a simplified method for accessing DRAM. The MPMC Basic interface supports a single, fixed burst length and drives all other MPMC Direct signals automatically. Note that because the underlying hardware interface is based on the MPMC Direct, it shares the same limitation that the system clock must run at 200MHz due to restrictions enforced by the MPMC.

All MPMC Basic transactions are initiated by the `req` signal. For writes, the `req` signal must be driven high for exactly one cycle on the same cycle as the first valid data value. The next *N*-1 data values will also be written to the next sequential addresses in DRAM, where *N* is the burst length. For reads, the `req` signal must be driven high for a single cycle and returned low. Once the transaction is completed from DRAM, data will be returned sequentially, and `rd_valid` will be driven high on every cycle that valid data is present on the read port. For all transactions, the `ack` signal will be driven high once a valid transaction has been received by the MPMC.

For a detailed example of how to use the MPMC Basic block, please see the DRAM demonstrations provided with BPS.

Table 91: MPMC Basic block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Data port width	This parameter can be set to either 32 or 64, and determines the width of the data ports into and out of the MPMC.
Burst length	This parameter sets the burst length which will be used for all MPMC transactions.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 92: MPMC Basic port definitions

Port	Direction	Bit Width	Description
`addr`	in	32	The starting address for all burst transactions. The value on this port is only valid on the cycle that `req` is high.
`req`	in	1	Initiates a burst transaction. For writes, `req` must be driven high for one cycle along with the starting address and the first data value. For reads, `req` must be driven high for one cycle along with the starting address.
`rnw`	in	1	Indicates a read (driven high) or write (driven low) transaction.
`wr_data`	in	Variable	The data to be written on write transactions. The word width can be either 32 or 64 bits, based on the *Data bitwidth* block parameter setting.
`ack`	out	1	Indicates that a transaction has been successfully received by the MPMC.
`rd_valid`	out	1	Indicates that the data on `rd_valid` is valid. This signal will remain high for as many cycles as the burst length.
`rd_data`	out	Variable	The data that has been read from the MPMC. The value on this port is only valid when `rd_valid` is high. The word width can be either 32 or 64 bits, based on the *Data bitwidth* block parameter setting.

MPMC VFBC block

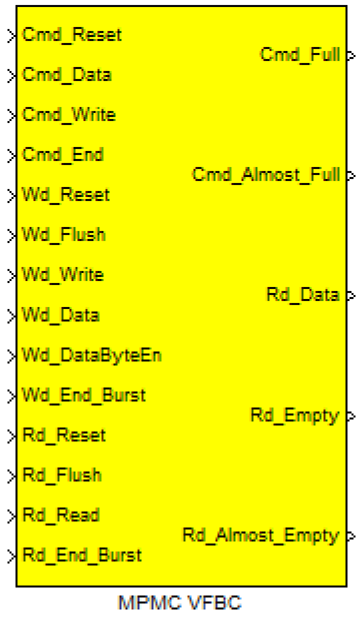
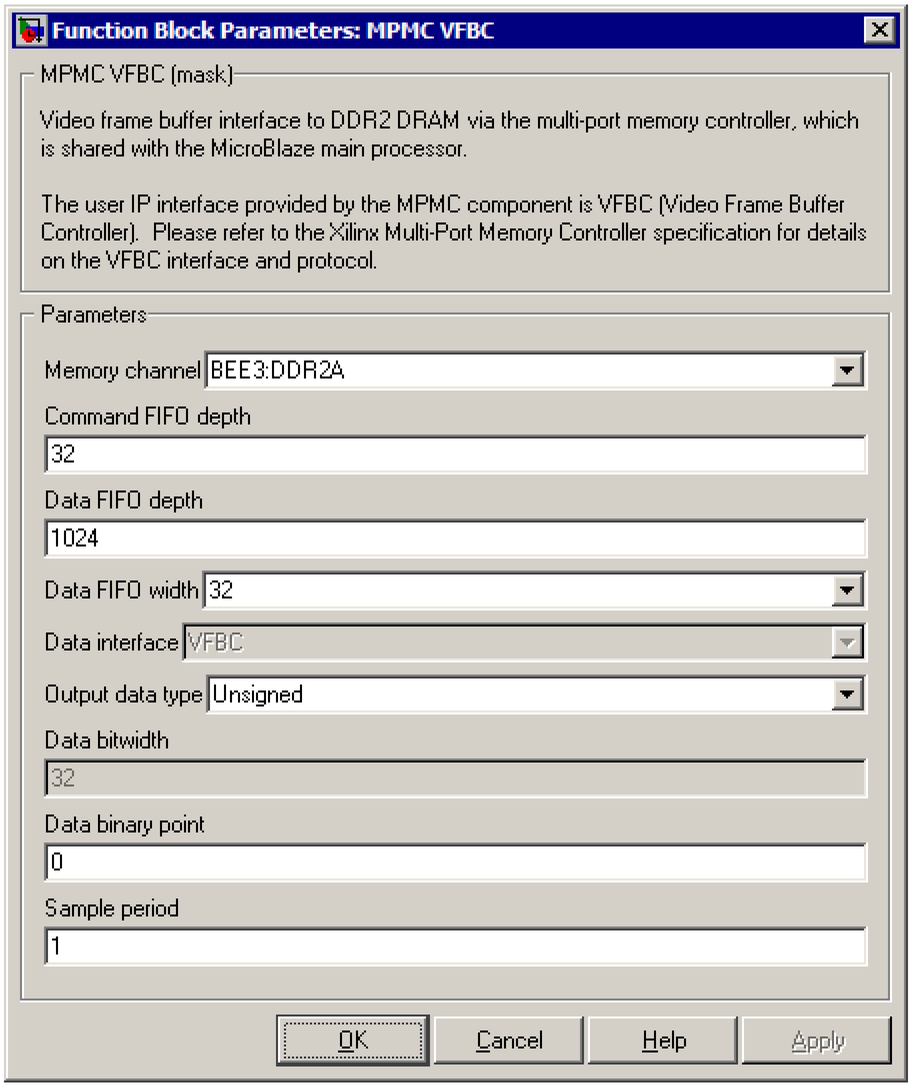


Figure 122: MPMC VFBC block

The MPMC Video Frame Buffer Controller (VFBC) block adds a hardware-accessible port to an existing MPMC in the design. This implies that a Memory Controller block must exist in the system for an MPMC VFBC block to be valid. The MPMC architecture allows up to a total of 8 ports (this includes any ports reserved by the base components, which is dependent on the platform configuration settings and other peripherals). The VFBC port runs asynchronously to the rest of the MPMC logic, and therefore the system clock rate is not limited to any one frequency.

The MPMC Direct block directly exposes the VFBC ports used by the MPMC. For detailed information on the VFBC protocol, including the complete port specification, please refer to the MPMC documentation provided with Xilinx EDK.



MPMC VFBC (mask)

Video frame buffer interface to DDR2 DRAM via the multi-port memory controller, which is shared with the MicroBlaze main processor.

The user IP interface provided by the MPMC component is VFBC (Video Frame Buffer Controller). Please refer to the Xilinx Multi-Port Memory Controller specification for details on the VFBC interface and protocol.

Parameters

Memory channel: BEE3:DDR2A

Command FIFO depth: 32

Data FIFO depth: 1024

Data FIFO width: 32

Data interface: VFBC

Output data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Sample period: 1

Figure 123: MPMC VFBC dialog

Table 93: MPMC VFBC block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Command FIFO depth	This value determines the depth of the command FIFO to be used on the port interface.
Data FIFO depth	This value determines the depth of the data FIFOs to be used on the port interface.
Data interface	This parameter specifies the MPMC port type used for this interface, which is fixed to VFBC.
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Data port width	This parameter can be set to either 32 or 64, and determines the width of the data ports into and out of the MPMC.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

MPMC Read Frame Buffer block

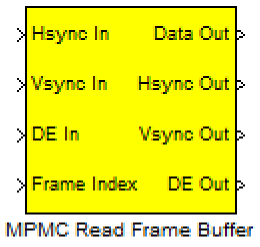
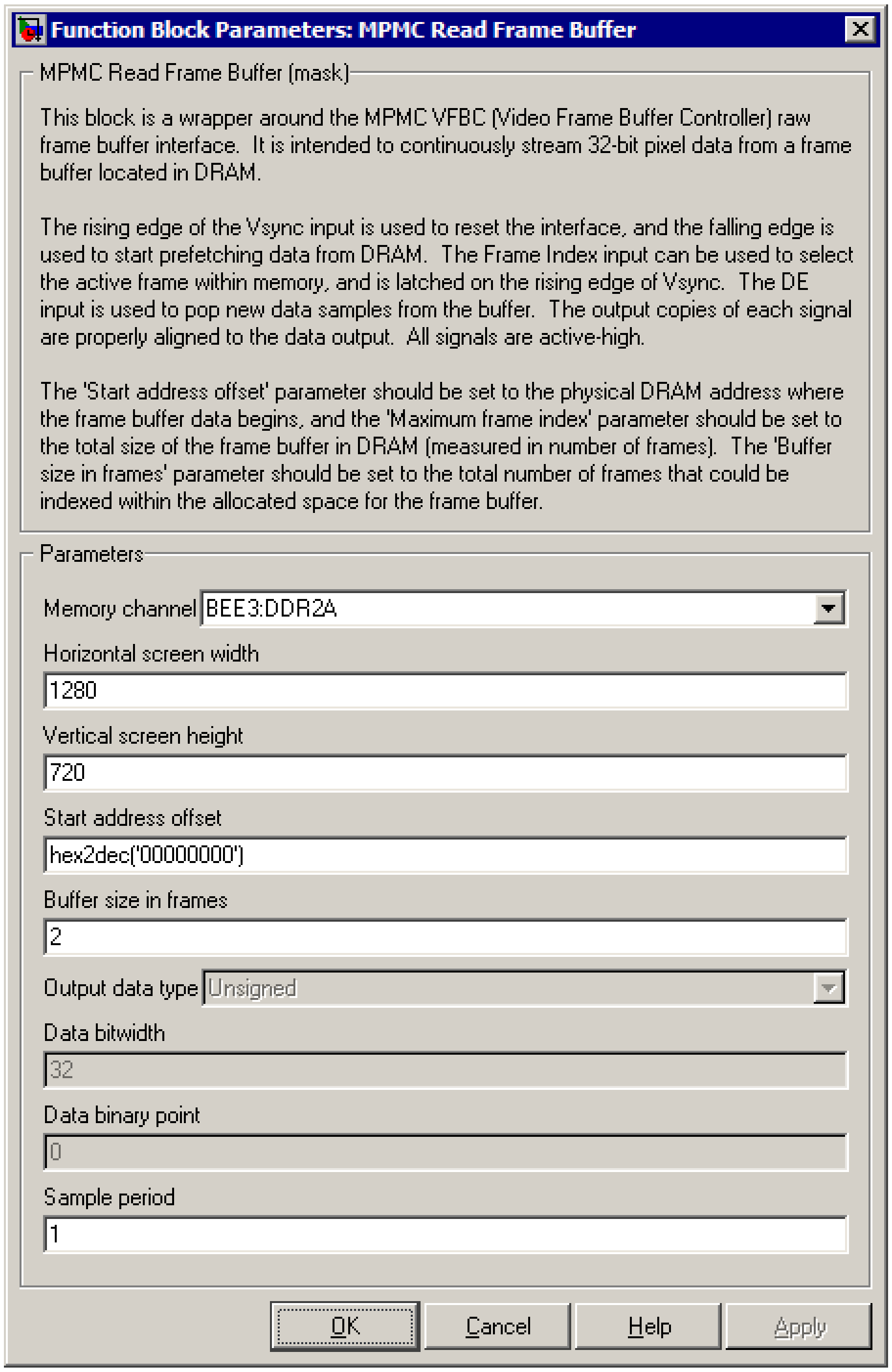


Figure 124: MPMC Read Frame Buffer block

The MPMC Read Frame Buffer block is a wrapper around the MPMC VFBC block which provides a simplified interface for video applications. Standard video timing signals (horizontal and vertical sync) are used as inputs to control the VFBC port as a read-only 2-D frame buffer. The exact behavior of the frame buffer in response to each of these control inputs is described along with each port below. In addition to providing the output pixel data from the frame buffer, the block also properly delays the control inputs by one cycle so that they are aligned with the data coming out of memory. A port is also provided to switch between multiple frames in memory at runtime, allowing an arbitrary amount of video data to be stored in memory and fetched dynamically by the hardware design.



Function Block Parameters: MPMC Read Frame Buffer

MPMC Read Frame Buffer (mask)

This block is a wrapper around the MPMC VFBC (Video Frame Buffer Controller) raw frame buffer interface. It is intended to continuously stream 32-bit pixel data from a frame buffer located in DRAM.

The rising edge of the Vsync input is used to reset the interface, and the falling edge is used to start prefetching data from DRAM. The Frame Index input can be used to select the active frame within memory, and is latched on the rising edge of Vsync. The DE input is used to pop new data samples from the buffer. The output copies of each signal are properly aligned to the data output. All signals are active-high.

The 'Start address offset' parameter should be set to the physical DRAM address where the frame buffer data begins, and the 'Maximum frame index' parameter should be set to the total size of the frame buffer in DRAM (measured in number of frames). The 'Buffer size in frames' parameter should be set to the total number of frames that could be indexed within the allocated space for the frame buffer.

Parameters

Memory channel: BEE3:DDR2A

Horizontal screen width: 1280

Vertical screen height: 720

Start address offset: hex2dec('00000000')

Buffer size in frames: 2

Output data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Sample period: 1

Figure 125: MPMC Read Frame Buffer dialog

Table 94: MPMC Read Frame Buffer block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Horizontal screen width	This parameter defines the line width to be used when accessing the frame buffer. This value must be a multiple of 32 to comply with the VFBC burst size (to use other values, the VFBC must be used directly).
Vertical screen height	This parameters defines the number of lines in each screen when accessing the frame buffer.
Start address offset	This parameter defines the starting address in DRAM to be used as the base address of the frame buffer. The value must be a multiple of 128 bytes to align with the VFBC burst size.
Buffer size in frames	This parameter defines the total number of frames allocated for the frame buffer. The value is used to determine the maximum value that will be accepted on the frame index port.
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Data port width	This parameter can be set to either 32 or 64, and determines the width of the data ports into and out of the MPMC.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 95: MPMC Read Frame Buffer port definitions

Port	**Direction**	**Bit Width**	**Description**
`Hsync In`	in	1	Horizontal sync input, active high. This signal is not used by the VFBC control logic.
`Vsync In`	in	1	Vertical sync input, active high. The rising edge of this signal is used to latch the current value on the `Frame Index` port, and also triggers a reset of the VFBC interface. The falling edge of this signal is used to start prefetching pixel data from memory.
`DE In`	in	1	Data enable input, active high. This signal used to pop pixel data from the frame buffer FIFO, which will be valid on the `Data Out` port after one cycle.
`Frame Index`	in	Variable	Frame index input. This signal is used to determine which frame is being read from in memory. The value is latched on the rising edge of `Vsync In`. It is saturated in hardware to never be greater than or equal to the total buffer size (specified in the block parameters).
`Data Out`	out	32	Pixel data output. This signal contains the 32-bit pixel data loaded from the frame buffer. Data is popped from the read FIFO when `DE In` is high, and is valid one cycle later (which will be when `DE Out` is high).
`Hsync Out`	out	1	Horizontal sync output, active high. This signal is equivalent to `Hsync In`, delayed by one cycle to line up with the pixel data output.
`Vsync Out`	out	1	Vertical sync output, active high. This signal is equivalent to `Vsync In`, delayed by one cycle to line up with the pixel data output.
`DE Out`	out	1	Data enable output, active high. This signal is equivalent to `DE In`, delayed by one cycle to line up with the pixel data output.

MPMC Write Frame Buffer block

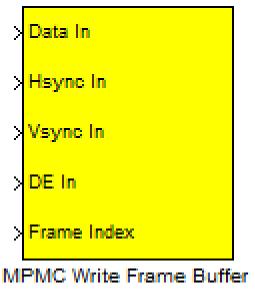
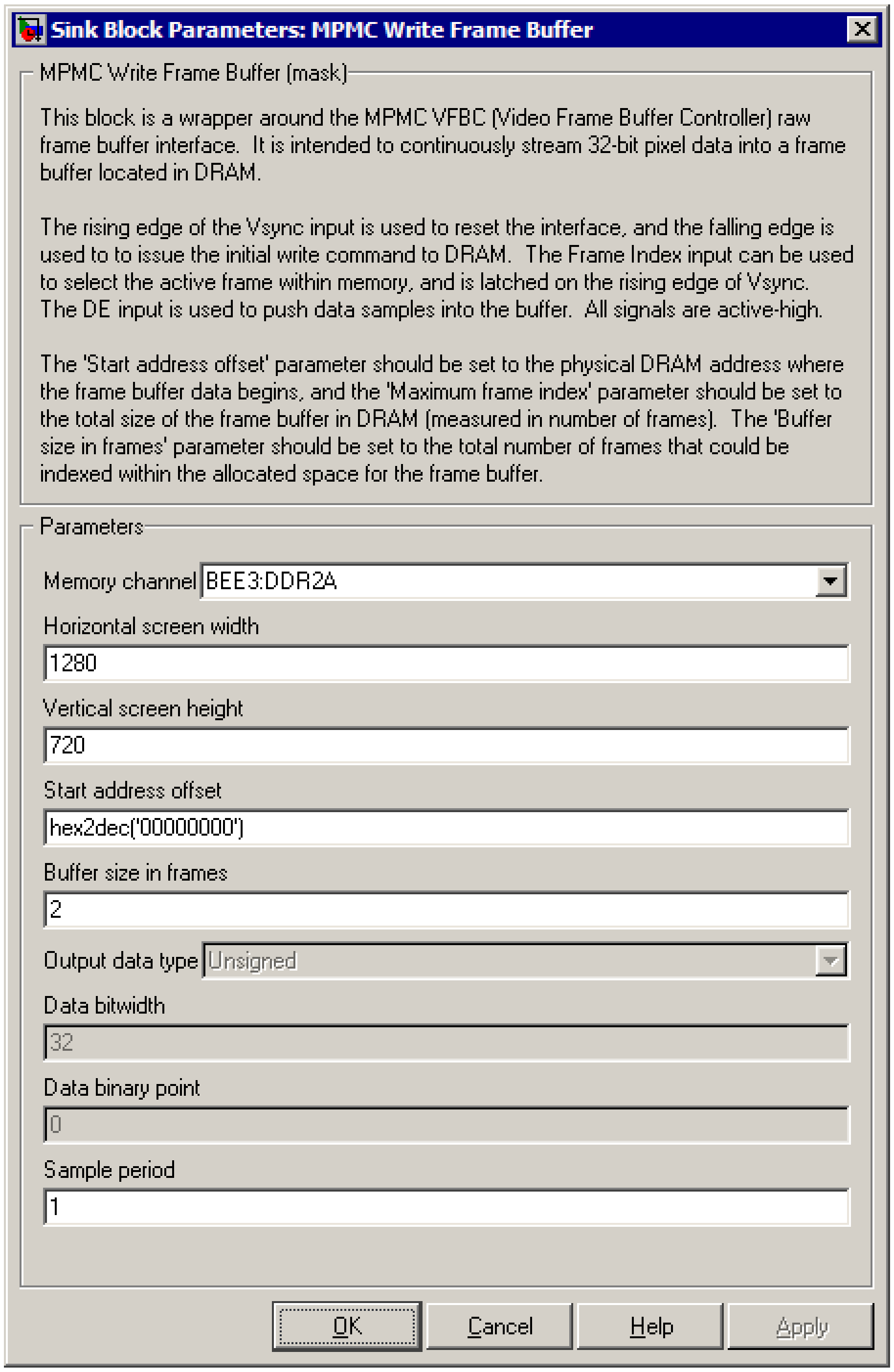


Figure 126: MPMC Write Frame Buffer block

The MPMC Write Frame Buffer block is a wrapper around the MPMC VFBC block which provides a simplified interface for video applications. Standard video timing signals (horizontal and vertical sync) are used as inputs to control the VFBC port as a write-only 2-D frame buffer. The exact behavior of the frame buffer in response to each of these control inputs is described along with each port below. A port is also provided to switch between multiple frames in memory at runtime, allowing an arbitrary amount of video data to be stored in memory and fetched dynamically by the hardware design.



Sink Block Parameters: MPMC Write Frame Buffer

MPMC Write Frame Buffer (mask)

This block is a wrapper around the MPMC VFBC (Video Frame Buffer Controller) raw frame buffer interface. It is intended to continuously stream 32-bit pixel data into a frame buffer located in DRAM.

The rising edge of the Vsync input is used to reset the interface, and the falling edge is used to to issue the initial write command to DRAM. The Frame Index input can be used to select the active frame within memory, and is latched on the rising edge of Vsync. The DE input is used to push data samples into the buffer. All signals are active-high.

The 'Start address offset' parameter should be set to the physical DRAM address where the frame buffer data begins, and the 'Maximum frame index' parameter should be set to the total size of the frame buffer in DRAM (measured in number of frames). The 'Buffer size in frames' parameter should be set to the total number of frames that could be indexed within the allocated space for the frame buffer.

Parameters

Memory channel BEE3:DDR2A

Horizontal screen width
1280

Vertical screen height
720

Start address offset
hex2dec('00000000')

Buffer size in frames
2

Output data type Unsigned

Data bitwidth
32

Data binary point
0

Sample period
1

Figure 127: MPMC Write Frame Buffer dialog

Table 96: MPMC Write Frame Buffer block parameters

Parameter	**Description**
Memory channel	This parameter should be set to the memory channel which this port instance should be attached to. The options for this parameter will vary based on the hardware platform.
Horizontal screen width	This parameter defines the line width to be used when accessing the frame buffer. This value must be a multiple of 32 to comply with the VFBC burst size (to use other values, the VFBC must be used directly).
Vertical screen height	This parameters defines the number of lines in each screen when accessing the frame buffer.
Start address offset	This parameter defines the starting address in DRAM to be used as the base address of the frame buffer. The value must be a multiple of 128 bytes to align with the VFBC burst size.
Buffer size in frames	This parameter defines the total number of frames allocated for the frame buffer. The value is used to determine the maximum value that will be accepted on the frame index port.
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Boolean*, *Unsigned*, or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits.
Data binary point	This value determines the inferred binary point position for the signal.
Data port width	This parameter can be set to either 32 or 64, and determines the width of the data ports into and out of the MPMC.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 97: MPMC Write Frame Buffer port definitions

Port	Direction	Bit Width	Description
`Hsync In`	in	1	Horizontal sync input, active high. This signal is not used by the VFBC control logic, and is only provided for port compatibility with other blocks.
`VSync In`	in	1	Vertical sync input, active high. The rising edge of this signal is used to latch the current value on the `Frame Index` port, and also triggers a reset of the VFBC interface. The falling edge of this signal is used to pre-issue the write command to memory.
`DE In`	in	1	Data enable input, active high. This signal used to push pixel data into the frame buffer FIFO.
`Frame Index`	in	Variable	Frame index input. This signal is used to determine which frame is being written to in memory. The value is latched on the rising edge of `Vsync In`. It is saturated in hardware to never be greater than or equal to the total buffer size (specified in the block parameters) to help prevent data corruption in memory.

Memory Controller NectarOS functions

BPS adds the following functions to NectarOS whenever a Memory Controller block is used in a design. The source code for these functions is contained in the *drivers/xps_bee3_mpmc* or *drivers/xps_mpmc* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`dramtest`

Usage: `dramtest <mode (short|long)>`

The `dramtest` command will run a basic memory test in the DRAM address space. When the `mode` argument is `short`, 4096 bytes of data will be written and verified with 8-bit, 16-bit, and 32-bit operations. When the mode argument is `long`, the same test will be run over the entire DRAM range. Note that this test will destructively write to DRAM. Therefore, if your application has any valid data already stored in DRAM, it may need to be reloaded after running this command to avoid unexpected behavior.

Memory Controller EDK API functions

On the BEE3 hardware platform, any Memory Controller blocks which are not mapped directly into the embedded processor's address space are accessed in software by a custom memory port generated by BPS called an *FSL PIM*. This port uses specially formatted commands passed over a MicroBlaze FSL port to read or write bursts of data from DRAM. The underlying FSL PIM component is implemented as an EDK pcore/driver combination. The EDK driver contains data structures and functions which can be used by custom software applications to gain low-level access to the FSL PIM interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any BEE3 base package. Include the header *xmpmc.h* for access to the API.

Data Structures

There is one data structure used by several of the FSL PIM EDK driver function calls which defines the specific configuration details of the hardware device.

Table 98: `XFslPim_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`fsl_slot`	Unsigned 8 bit	MicroBlaze FSL slot used by PIM
`channel_id`	Unsigned 8 bit	Numeric ID for BEE3 memory channel being used
`burst_length`	Unsigned 8 bit	Built-in burst length used for all data operations

Functions

```
XFslPim_Config *
XFslPim_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor PLB bus. The return value of this function should be used as the `XFslPim_Handle` argument required by other driver routines.

```
XStatus
XFslPim_GetStatusReg(XFslPim_Handle handle,
                     Xuint32 * value_p);
```

Reads the raw contents of the FSL PIM status register and writes the result into the location referenced by `value_p`. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful. This is a low-level debugging routine – the format of the status register is derived by the FSL PIM HDL implementation.

```
XStatus
XFslPim_GetNpiSignals(XFslPim_Handle handle,
                      Xuint32 * value_p);
```

Reads the current values of all relevant MPMC NPI port control signals and writes the results as a 32-bit word into the location referenced by `value_p`. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful. This is a low-level debugging routine – the format of the output word is derived by the FSL PIM HDL implementation.

```
XStatus
XFslPim_SetRnw(XFslPim_Handle handle,
               Xuint32 value);
```

Sets the current read/write state of the PIM. A ‘0’ in the LSB of `value` defines a write transaction, while a ‘1’ defines a read transaction. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_GetAddress(XFslPim_Handle handle,
                   Xuint32 * address_p);
```

Gets the current base physical address being used for all memory transactions. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_SetAddress(XFslPim_Handle handle,
                   Xuint32 address);
```

Sets the base physical address to be used for all memory transactions. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_IssueRequest(XFslPim_Handle handle);
```

Issues a transaction on the NPI port interface. This function is typically called last, after the read/write state and base address have been set. For write requests, the data should already have been pushed into the PIM FIFO (such as by `XFslPim_PushWord`). For read requests, data will be fetched by the memory controller and can be popped from the PIM FIFO once the FIFO status is non-empty. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_Reset(XFslPim_Handle handle);
```

Reset the FSL PIM and NPI port interface. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_PushWord(XFslPim_Handle handle,
                 Xuint32 value);
```

Pushes the word `value` into the MPMC write FIFO. The data will not be written until a burst write transaction is issued. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_PopWord(XFslPim_Handle handle,
                Xuint32 * value_p);
```

Pops a word from the MPMC read FIFO and writes it into the location referenced by `value_p`. The data is only valid if the read FIFO is not empty, which will be true after a burst read transaction. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_BurstWrite(XFslPim_Handle handle,
                   Xuint32 address,
                   Xuint32 * data_p);
```

Performs one full write burst, copying the data starting at the location referenced by `data_p` into DRAM at the physical address `address`. The burst length is a fixed parameter of the FSL PIM, which can be found in the device configuration structure. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XFslPim_BurstRead(XFslPim_Handle handle,
                  Xuint32 address,
                  Xuint32 * data_p);
```

Performs one full read burst, copying the data from DRAM physical address `address` into local memory at the location referenced by `data_p`. The burst length is a fixed parameter of the FSL PIM, which can be found in the device configuration structure. The argument `handle` should be a configuration structure pointer as returned by the `XFslPim_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

PCI Express

BPS supports the use of PCI Express as a high-speed data interface between the FPGA hardware platform and a host PC or workstation. Because the underlying physical implementation for PCI Express is highly platform dependent, separate PCI Express blocks are contained in the hardware-specific block libraries. Use of the PCI Express block will also cause BPS to automatically add the necessary initialization routines into NectarOS to enable the hardware interface.

PCI Express Endpoint block for BEE3

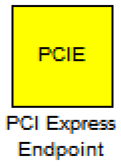
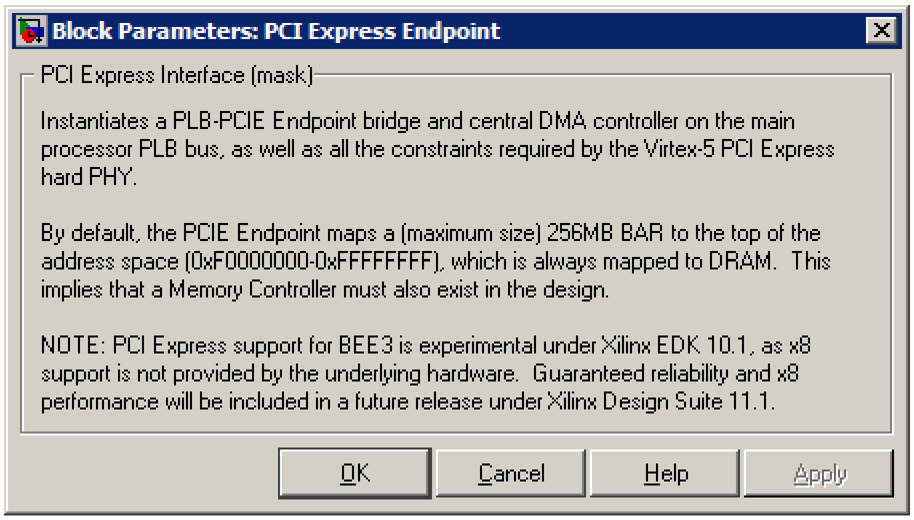


Figure 128: PCI Express Endpoint block for BEE3


Block Parameters: PCI Express Endpoint
PCI Express Interface (mask)
OK
Cancel
Help
Apply

Instantiates a PLB-PCIE Endpoint bridge and central DMA controller on the main processor PLB bus, as well as all the constraints required by the Virtex-5 PCI Express hard PHY.

By default, the PCIE Endpoint maps a (maximum size) 256MB BAR to the top of the address space (0xF0000000-0xFFFFFFFF), which is always mapped to DRAM. This implies that a Memory Controller must also exist in the design.

NOTE: PCI Express support for BEE3 is experimental under Xilinx EDK 10.1, as x8 support is not provided by the underlying hardware. Guaranteed reliability and x8 performance will be included in a future release under Xilinx Design Suite 11.1.

Figure 129: PCI Express Endpoint dialog for BEE3

The PCI Express Endpoint block instantiates a PCI Express Endpoint to PLB bus bridge and a central DMA controller on the main processor PLB bus. This configuration allows any read or write transaction initiated from a host PC to be handled in hardware. The host-side driver and communication protocol used for transferring data between the FPGA and a host PC are provided separately.

Note that on BEE3, PCI Express support under Xilinx ISE/EDK 10.1 is experimental. The Xilinx PCI Express Endpoint to PLB bridge does not support the x8 lane width, which is used by BEE3 and the optional PCI Express expansion kit. The x1 lane width used under Xilinx EDK 10.1 can cause instability in the PCI Express link, the severity of which can vary from board to board. Support for x8 lane width under Xilinx Design Suite 11 has already been validated and will be added in an upcoming release of BPS.

PCI Express Endpoint block for ML50x

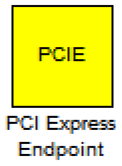
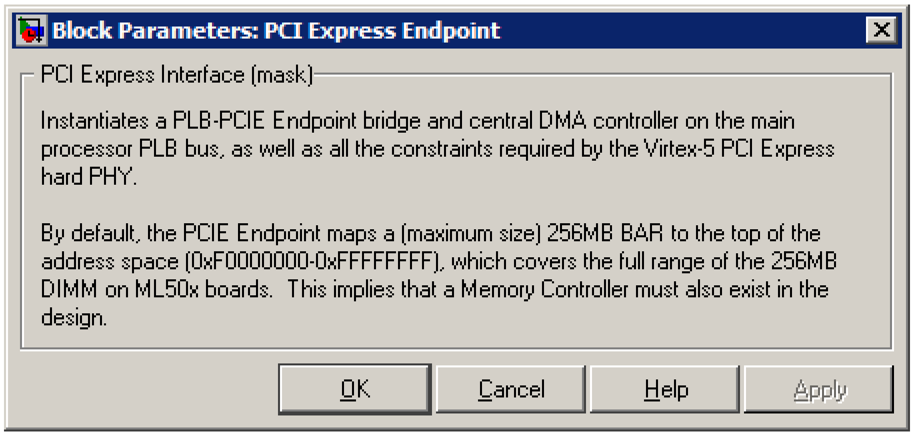


Figure 130: PCI Express Endpoint block for ML50x



PCI Express Interface (mask)

Instantiates a PLB-PCIE Endpoint bridge and central DMA controller on the main processor PLB bus, as well as all the constraints required by the Virtex-5 PCI Express hard PHY.

By default, the PCIE Endpoint maps a (maximum size) 256MB BAR to the top of the address space (0xF0000000-0xFFFFFFFF), which covers the full range of the 256MB DIMM on ML50x boards. This implies that a Memory Controller must also exist in the design.

Figure 131: PCI Express Endpoint dialog for ML50x

The PCI Express Endpoint block instantiates a PCI Express Endpoint to PLB bus bridge and a central DMA controller on the main processor PLB bus. This configuration allows any read or write transaction initiated from a host PC to be handled in hardware. The host-side driver and communication protocol used for transferring data between the FPGA and a host PC are provided separately.

Platform Configuration

The Platform Configuration block is used by BPS to determine all the system-level parameters for a given design, and must be included at the top level of each FPGA implementation. Separate Platform Configuration blocks are provided in each hardware-specific block library to reflect all the unique configuration settings supported by each platform.

Platform Configuration block

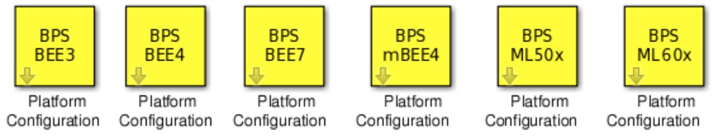
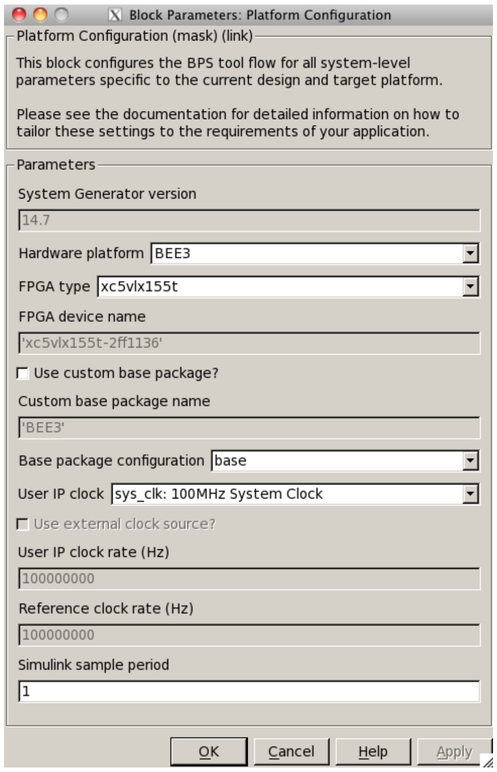


Figure 132: Platform Configuration blocks

A Platform Configuration block must exist at the top level of each system which represents an FPGA implementation which should be generated by BPS. While multiple Platform Configuration blocks may be used in independent parallel systems in the Simulink hierarchy to model a multiple-FPGA application, is it not possible to have more than one Platform Configuration block within the same hierarchy. Doing so will cause BPS to report an error.

In addition to defining the top level of an FPGA implementation, the Platform Configuration block allows you to define a variety of system-level parameters which apply to the whole FPGA design, such as the target hardware platform, base package selection, and clocking configuration.


Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version: 14.7

Hardware platform: BEE3

FPGA type: xc5vlx155t

FPGA device name: 'xc5vlx155t-2ff1136'

☐ Use custom base package?

Custom base package name: 'BEE3'

Base package configuration: base

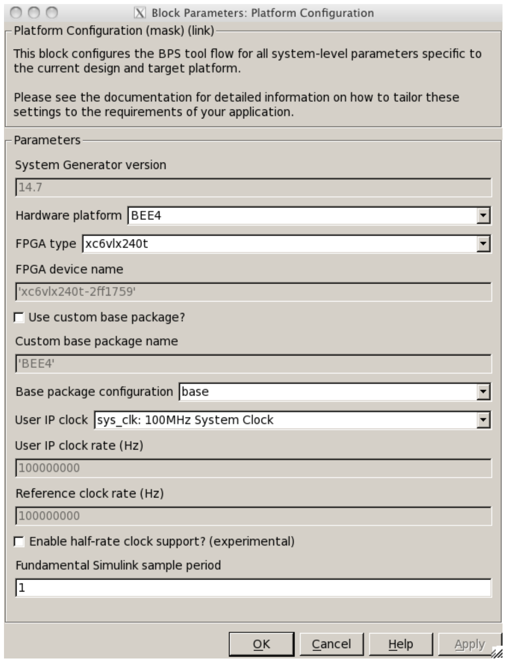
User IP clock: sys_clk: 100MHz System Clock

☐ Use external clock source?

User IP clock rate (Hz): 100000000

Reference clock rate (Hz): 100000000

Simulink sample period: 1


Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version: 14.7

Hardware platform: BEE4

FPGA type: xc6vlx240t

FPGA device name: 'xc6vlx240t-2ff1759'

☐ Use custom base package?

Custom base package name: 'BEE4'

Base package configuration: base

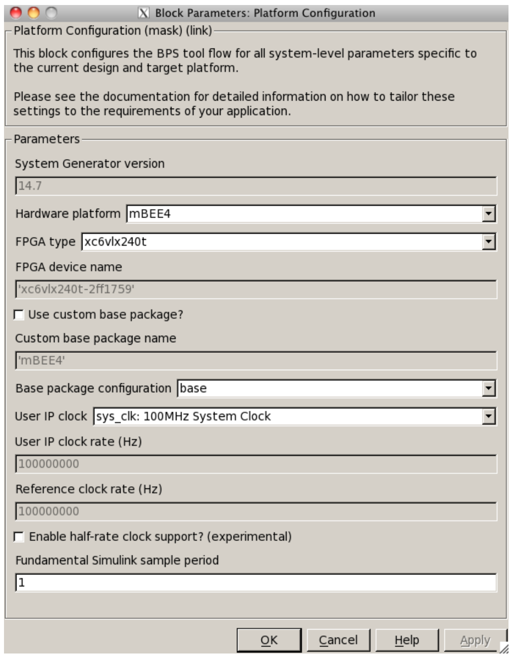
User IP clock: sys_clk: 100MHz System Clock

User IP clock rate (Hz): 100000000

Reference clock rate (Hz): 100000000

☐ Enable half-rate clock support? (experimental)

Fundamental Simulink sample period: 1


Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version: 14.7

Hardware platform: mBEE4

FPGA type: xc6vlx240t

FPGA device name: 'xc6vlx240t-2ff1759'

☐ Use custom base package?

Custom base package name: 'mBEE4'

Base package configuration: base

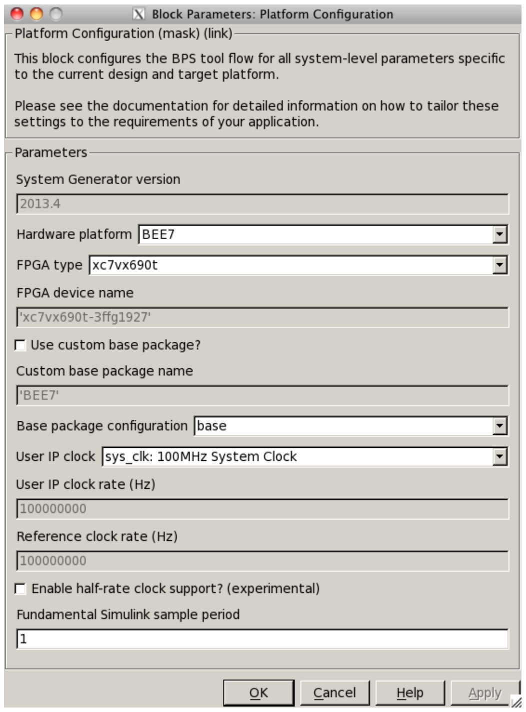
User IP clock: sys_clk: 100MHz System Clock

User IP clock rate (Hz): 100000000

Reference clock rate (Hz): 100000000

☐ Enable half-rate clock support? (experimental)

Fundamental Simulink sample period: 1


Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version: 2013.4

Hardware platform: BEE7

FPGA type: xc7vx690t

FPGA device name: 'xc7vx690t-3ffg1927'

☐ Use custom base package?

Custom base package name: 'BEE7'

Base package configuration: base

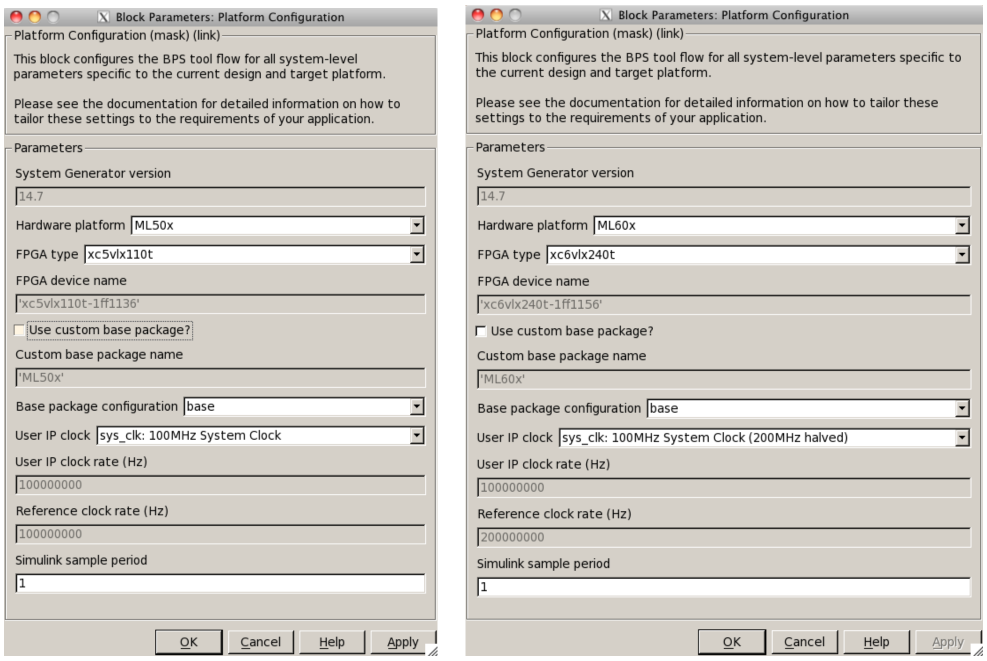
User IP clock: sys_clk: 100MHz System Clock

User IP clock rate (Hz): 100000000

Reference clock rate (Hz): 100000000

☐ Enable half-rate clock support? (experimental)

Fundamental Simulink sample period: 1



Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version	14.7
Hardware platform	ML50x
FPGA type	xc5vlx110t
FPGA device name	'xc5vlx110t-1ff1136'
Use custom base package?	☐
Custom base package name	'ML50x'
Base package configuration	base
User IP clock	sys_clk: 100MHz System Clock
User IP clock rate (Hz)	100000000
Reference clock rate (Hz)	100000000
Simulink sample period	1

Block Parameters: Platform Configuration

Platform Configuration (mask) (link)

This block configures the BPS tool flow for all system-level parameters specific to the current design and target platform.

Please see the documentation for detailed information on how to tailor these settings to the requirements of your application.

Parameters

System Generator version	14.7
Hardware platform	ML60x
FPGA type	xc6vlx240t
FPGA device name	'xc6vlx240t-1ff1156'
Use custom base package?	☐
Custom base package name	'ML60x'
Base package configuration	base
User IP clock	sys_clk: 100MHz System Clock (200MHz halved)
User IP clock rate (Hz)	100000000
Reference clock rate (Hz)	200000000
Simulink sample period	1

Figure 133: Platform Configuration dialogs

Table 99: Platform Configuration block parameters

Parameter	**Description**
System Generator version	This is the version of System Generator (and consequently, the version of the back-end implementation tools) for which BPS will generate the FPGA implementation. This parameter is automatically derived by BPS based on the tool versions detected in the current environment.
Hardware platform	This is the hardware platform which will be used to determine the physical constraints for all external interfaces in the system. This option should be set to `BEE3` for the standard BEE3 platform configuration, or to `Custom` for advanced users who wish to provide their own customized infrastructure packages.
FPGA type	This is the FPGA model that will be targeted in the hardware implementation. This value should be set to the same FPGA part that was built into your system. By setting this value to Custom, you will have to manually enter the FPGA device string below.
FPGA device name	This is the exact type of FPGA device that will be targeted in the hardware implementation. This value will automatically be set by BPS based on your hardware platform and FPGA type settings.
Use custom base package	By checking this box, the *Custom base package name* field will become active, and BPS will search for the specified base package name, rather than the default value.
Custom base package name	This field must evaluate to a string value which represents the name of the custom base package to be used. BPS will search for the base package file `XPS_%VER%_%NAME%_base.zip`, where `%VER%` is the System Generator version string and `%NAME%` is the string defined in this field. If the environment variable `BPS_BASELIB_PATH` is set, BPS will search those directories first, followed by the Matlab path.
Base package configuration	This is the type of base package configuration that will be used for the FPGA implementation. The following options are available: • `base` – the basic base package configuration with standard BPS system components • `swdram` – software application memory mapped to DRAM rather than locally attached BRAM (allows larger memory footprints and requires an MPMC in the design, not yet supported on BEE4 or miniBEE4)

User IP clock	This parameter defines the source of the System Generator clock domain. The System Generator design itself, as well as the user-logic side of all BPS blocks, will be driven by this clock. The set of available clock sources are hardware platform and design independent.
Use external clock source?	*(BEE3 only)* When `usr_clk_fx` is selected as the design clock source, this checkbox is used to determine whether the reference clock should be taken from the on-board 133MHz crystal (unchecked), or if the user will provide an independent clock source via the external clock input (checked). Note that this also requires a corresponding switch setting on the BEE3 main board.
User IP clock rate (Hz)	When any custom-generated clock frequency (clock sources ending in `_fx`) is selected as the design clock source, this parameter determines the exact frequency that will be generated by the on-chip clock generator. Note that BPS must be able to find a valid ratio between the value entered here and the *Reference clock rate* frequency below, or else an error will be reported by BPS during implementation.
Reference clock rate (Hz)	For all clocks using the on-board crystals as the reference source, this parameter will be automatically set by BPS based on the value chosen for *User IP clock* above. For clocks using a reference source provided by the user, this field must be set to the appropriate frequency manually.
Enable half-rate clock support?	*(BEE4 only)* Selecting this option will provide support for using a half-rate clock (twice the fundamental sample period) in the System Generator portion of the design. This can be useful for applications where real-time BPS blocks for external interfaces require a high-speed clock, but complex portions of the algorithm logic might be unable to run at such a high speed. In this configuration, internal System Generator logic blocks may be set to use a sample period either equal to or exactly twice the fundamental period, but all BPS blocks must still be set to the fundamental period only. This option is currently only available when using a custom-generated frequency (clock sources ending in `_fx`).
Simulink sample period	This value defines the fundamental sample period of the System Generator portion of the Simulink design. The value entered here will automatically be passed to the System Generator block itself. All inputs to BPS blocks must use the same sample period as the fundamental value defined here, regardless of any other settings.

QDR SRAM

BPS provides a MIG interface to the SRAM chips on the DIMM101 SRAM DIMMs. There are no software components required by the SRAM block. For more information about the user interface signals, see Xilinx's *Virtex-6 FPGA Memory Interface Solutions User Guide* (UG406).

QDR SRAM block for BEE4 and miniBEE4

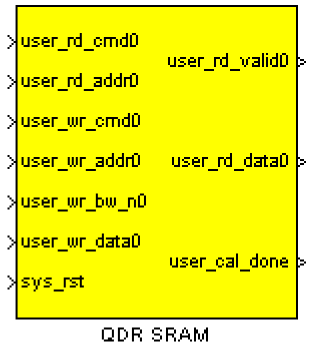
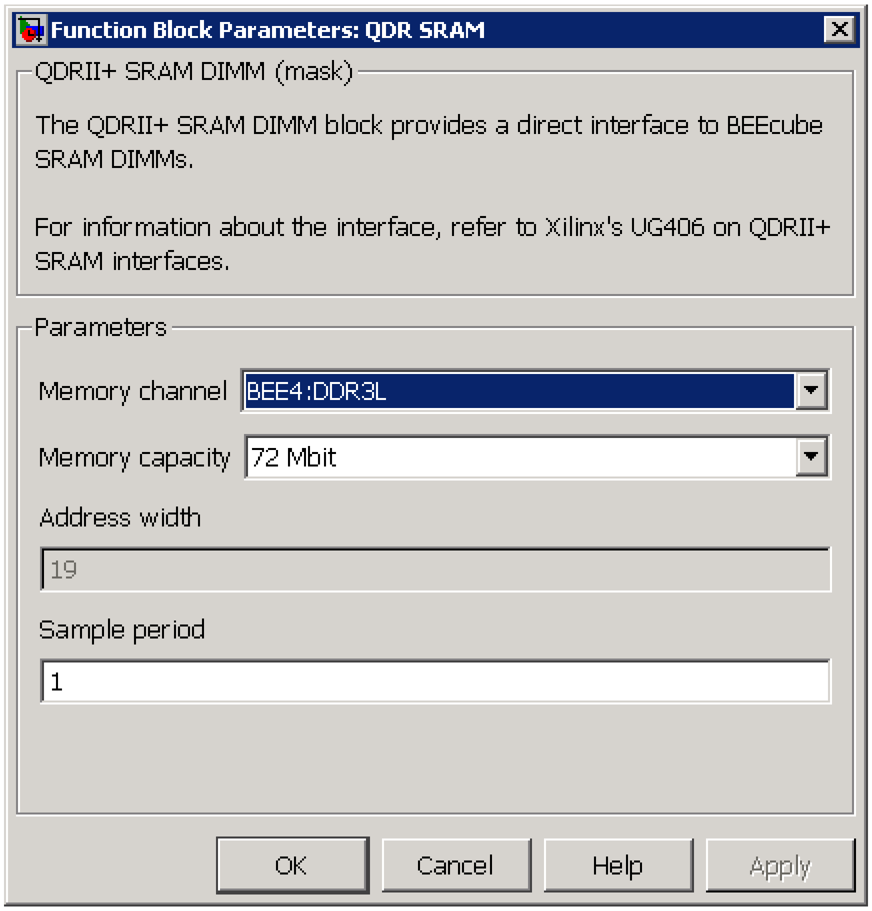


Figure 134: QDR SRAM block



QDRII+ SRAM DIMM (mask)

The QDRII+ SRAM DIMM block provides a direct interface to BEEcube SRAM DIMMs.

For information about the interface, refer to Xilinx's UG406 on QDRII+ SRAM interfaces.

Parameters

Memory channel BEE4:DDR3L

Memory capacity 72 Mbit

Address width

19

Sample period

1

Figure 135: QDR SRAM dialog

Table 100: QDR SRAM block parameters

Parameter	**Description**
Memory channel	This parameter sets the memory channel (and the corresponding DIMM slot) used on the board.
Memory capacity	This parameter defines the total capacity of the SRAM DIMM. As there are two chips per DIMM, this number is twice the size of the individual chip.
Address width	This value defines the total width of the address bus in bits. It is set by the memory capacity parameter and is provided for informational purposes.
Sample period	The Simulink sample period to be declared for this block.

RXAUI

BPS provides support for accessing the native RXAUI interface of the NetLogic PHY devices on the miniBEE4 hardware platform.

RXAUI block

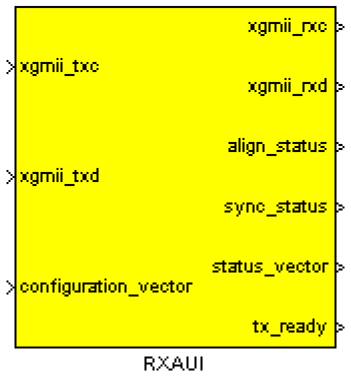
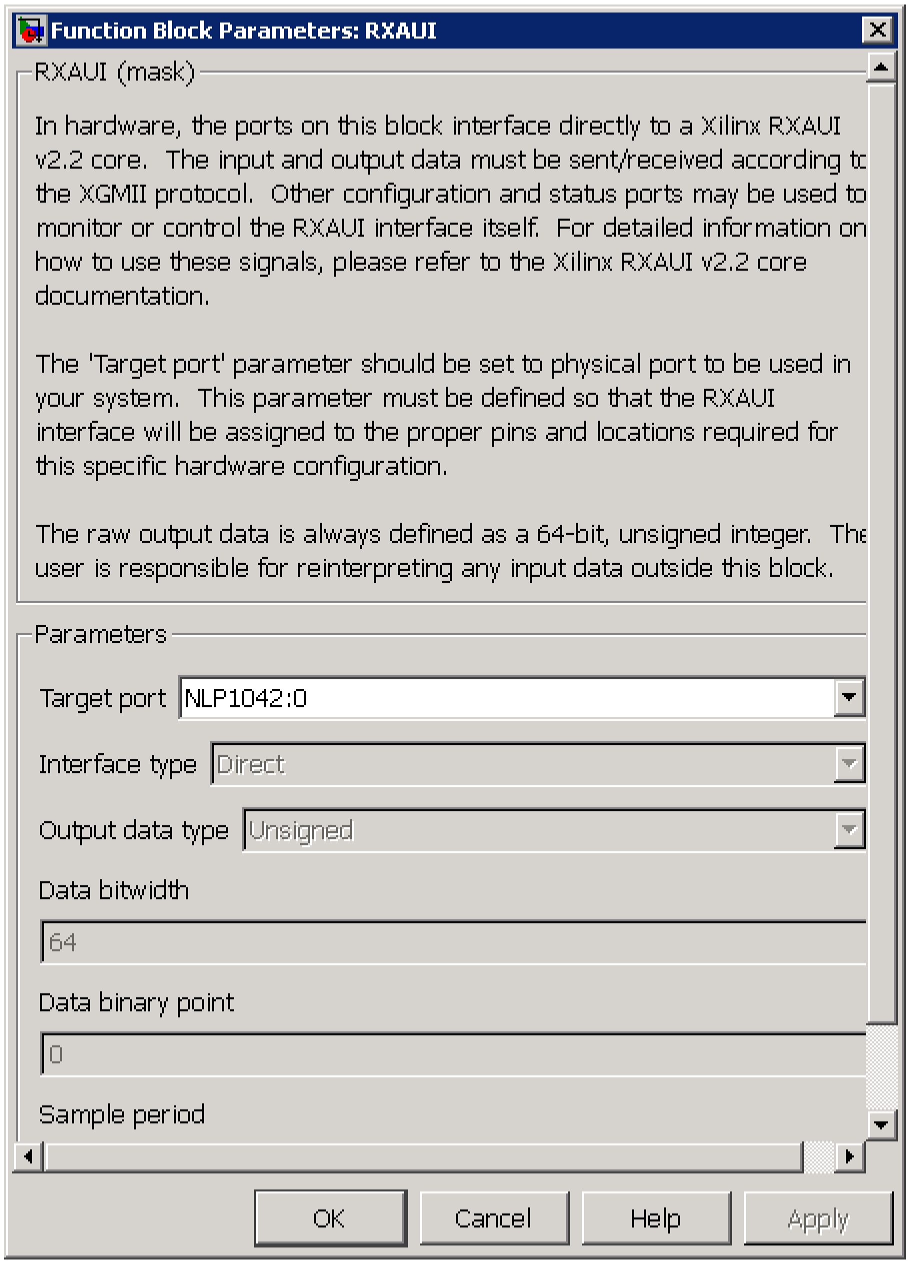


Figure 136: RXAUI block

The RXAUI block provides an interface which is identical to that of the Xilinx RXAUI v2.3 component in Core Generator. The data inputs and outputs adhere to the standard XGMII protocol. For more information on the usage of the auxiliary control and status ports, please refer to the XAUI core documentation in Core Generator.

Use of this block requires that the system clock be sourced by the 156.25MHz `rxaui_clk`, as the XGMII interface must run synchronously to the RXAUI core.



RXAUI (mask)

In hardware, the ports on this block interface directly to a Xilinx RXAUI v2.2 core. The input and output data must be sent/received according tc the XGMII protocol. Other configuration and status ports may be used to monitor or control the RXAUI interface itself. For detailed information on how to use these signals, please refer to the Xilinx RXAUI v2.2 core documentation.

The 'Target port' parameter should be set to physical port to be used in your system. This parameter must be defined so that the RXAUI interface will be assigned to the proper pins and locations required for this specific hardware configuration.

The raw output data is always defined as a 64-bit, unsigned integer. The user is responsible for reinterpreting any input data outside this block.

Parameters

Target port: NLP1042:0

Interface type: Direct

Output data type: Unsigned

Data bitwidth: 64

Data binary point: 0

Sample period

Figure 137: RXAUI dialog

Table 101: RXAUI block parameters

Parameter	**Description**
Data type	This parameter determines the inferred arithmetic type for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to *Unsigned*.
Data bitwidth	This value defines the total width of the signal in bits. The standard data rate (SDR) version of XGMII requires that this value be fixed to 64 bits.
Data binary point	This value determines the inferred binary point position for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to zero.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 102: RXAUI port definitions

Port	**Direction**	**Bit Width**	**Description**
`xgmii_txc` `xgmii_txd`	in	8 64	XGMII transmit control and data ports. Please refer to the Xilinx RXAUI v2.3 and XGMII standard documentation for protocol information.
`configuration_vector`	in	7	Vector of configuration bits for the XAUI core. Please refer to the Xilinx RXAUI v2.3 documentation for signal information.
`xgmii_rxc` `xgmii_rxd`	out	8 64	XGMII receive control and data ports. Please refer to the Xilinx RXAUI v2.3 and XGMII standard documentation for protocol information.
`align_status` `sync_status`	out	1 4	Vectors indicating high-speed serial lane alignment and synchronization status. Alignment status is a single bit indicating that all 2 RXAUI lanes are aligned, and synchronization status is a 2-bit value with the individual link status of each lane.
`status_vector`	out	8	Vector of status bits for the RXAUI core. Please refer to the Xilinx RXAUI v2.3 documentation for signal information.

`txlock`	out	1	Status bit indicating that the GTX transmitter clock is locked. *This signal is intended for debug use only and may be removed in a future release.*
`mgt_tx_ready`	out	1	Status bit indicating that the GTX transmitter interface is ready to accept outgoing traffic. *This signal is intended for debug use only and may be removed in a future release.*

RXAUI block NectarOS functions

The RXAUI block includes a set of utility functions that enable the user to control the RXAUI block directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever the RXAUI block is used in a design. The source code for these functions are contained in the *drivers/xps_rxaui* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`mdio_init`

The `mdio_init` function performs all initialization steps required to set up the physical MDIO interface to the PHY devices, and to configure the PHY devices for 10-gigabit optical Ethernet communication.

Repeated function calls

(none)

Commands

`mdio_phy_init`

The `mdio_phy_init` function will re-perform a complete initialization of both PHY devices.

`mdio_phy_reset`

The `mdio_phy_reset` function will assert a reset of the MDIO logic and the physical reset pin of the PHY devices.

`mdio_phy_read <reg_addr>`

The `mdio_phy_read` function will read the contents of the PHY register `reg_addr` on all PHY devices and print the result on the console.

`mdio_phy_write <reg_addr> <value>`

The `mdio_phy_write` function will write `value` to the PHY register `reg_addr` on all PHY devices

SerDes I/O

BPS provides support for high-speed data transfer between FPGAs on a single BEE7 board through the SerDes Input and Output blocks. Reliable operation is achieved via a physical hardware interface which is automatically calibrated during system initialization. On-demand re-calibrations can also be requested at runtime to account for changes in clock frequency (such as when external clock sources are used) or large variations in temperature.

Note that the SerDes Input and Output blocks are only intended be used in pairs: a SerDes Input block on a given I/O bus with a certain data bitwidth must be connected to a SerDes Output block on the same I/O bus with the same data bitwidth. Care should be taken when designing each FPGA implementation to be sure that the SerDes I/O interfaces on each side of a connection use the same parameters.

The remainder of this section describes the functionality of the SerDes Input and Output blocks in Simulink, as well as the complete software driver and runtime interfaces which are available for use in custom software designs.

SerDes Input block for BEE7

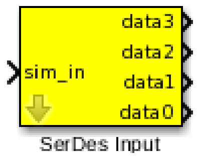
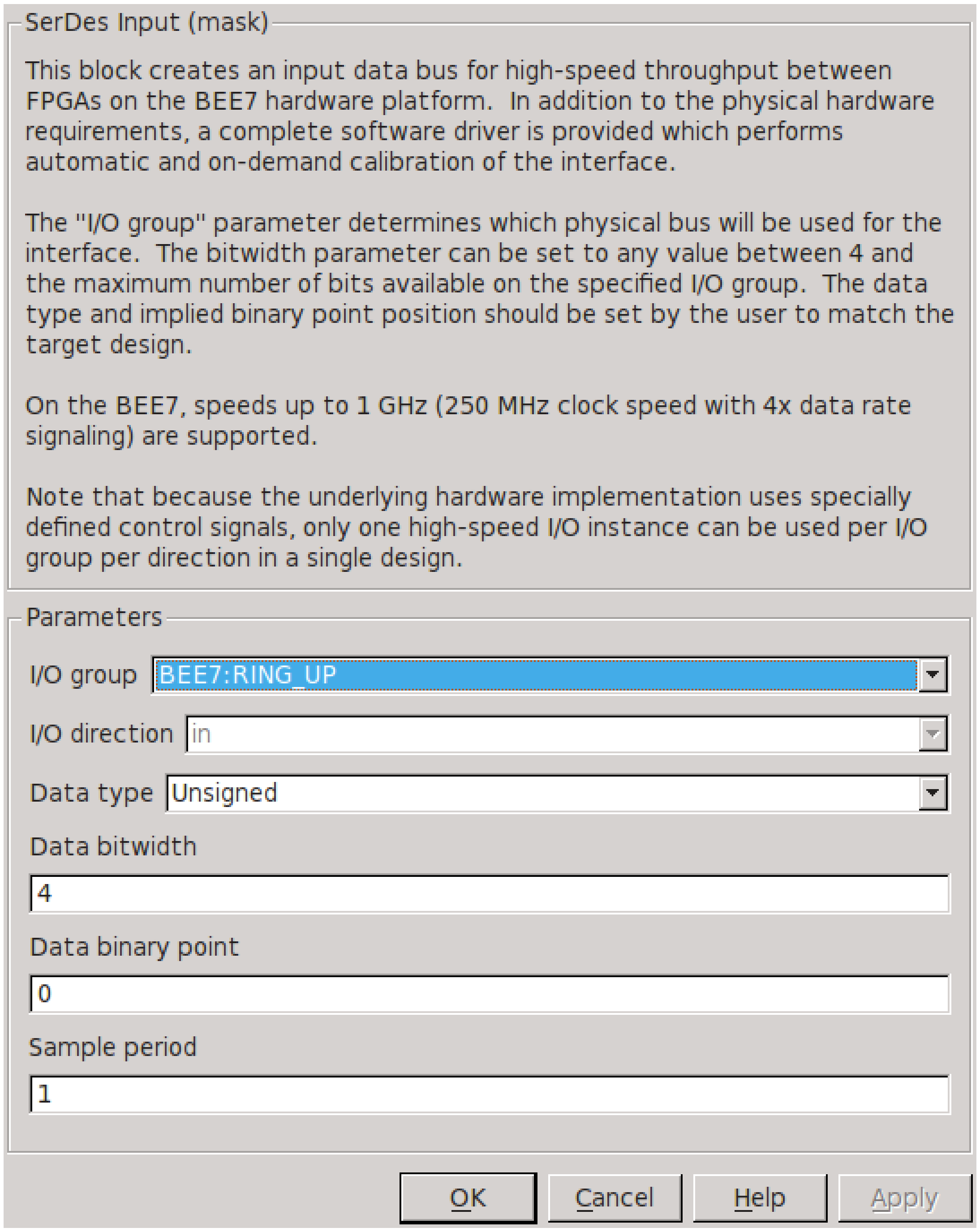


Figure 138: SerDes Input block



SerDes Input (mask)

This block creates an input data bus for high-speed throughput between FPGAs on the BEE7 hardware platform. In addition to the physical hardware requirements, a complete software driver is provided which performs automatic and on-demand calibration of the interface.

The "I/O group" parameter determines which physical bus will be used for the interface. The bitwidth parameter can be set to any value between 4 and the maximum number of bits available on the specified I/O group. The data type and implied binary point position should be set by the user to match the target design.

On the BEE7, speeds up to 1 GHz (250 MHz clock speed with 4x data rate signaling) are supported.

Note that because the underlying hardware implementation uses specially defined control signals, only one high-speed I/O instance can be used per I/O group per direction in a single design.

Parameters

I/O group: BEE7:RING_UP

I/O direction: in

Data type: Unsigned

Data bitwidth: 4

Data binary point: 0

Sample period: 1

Figure 139: SerDes Input dialog

The SerDes Input block provides an interface for receiving data between FPGAs. Because the underlying hardware implementation uses specific pins on the FPGA, only one instance per I/O group can be present in a single design. Multiple logical signals can always be packed into a single SerDes Input block and separated from each other using Slice blocks from the Xilinx System Generator Blockset. Similarly, the SerDes Input block cannot be combined with standard GPIO blocks assigned to the same I/O group.

Table 103: SerDes Input block parameters

Parameter	**Description**
I/O group	This parameter defines which group of FPGA pins will be used to assign the signal. This field will list all the pin groups available on the hardware platform for direct pin assignment.
I/O direction	Since the direction of signal flow is pre-determined for each end of the SerDes I/O interface to support automated calibration, this value is fixed to *in* for the input block.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. This can be any integer between 4 (the minimum width imposed by the underlying hardware implementation) and the maximum number of bits available for the given I/O group.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 104: SerDes Input port definitions

Port	**Direction**	**Bit Width**	**Description**

`data_in3`
`data_in2`
`data_in1`
`data_in0`
	out	*N*	Output ports which drive a hardware signal into the design with the System Generator signal parameters defined in the block configuration dialog (shown above). Note that because the SerDes Input block always uses signaling at 4x the clock rate, four data values are driven per system clock cycle. The `data0` output represents the data latched first, and the `data3` output represents the data latched last.

`sim_in`	in	N/A	Input port which can be used to provide simulation data within Simulink. Note that this Simulink input signal is valid only during simulation and has no effect on the hardware. The sample time of this input should be 1/4 that of the hardware system (i.e. 4x the sample rate).

SerDes Output block for BEE7

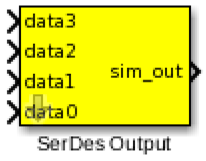
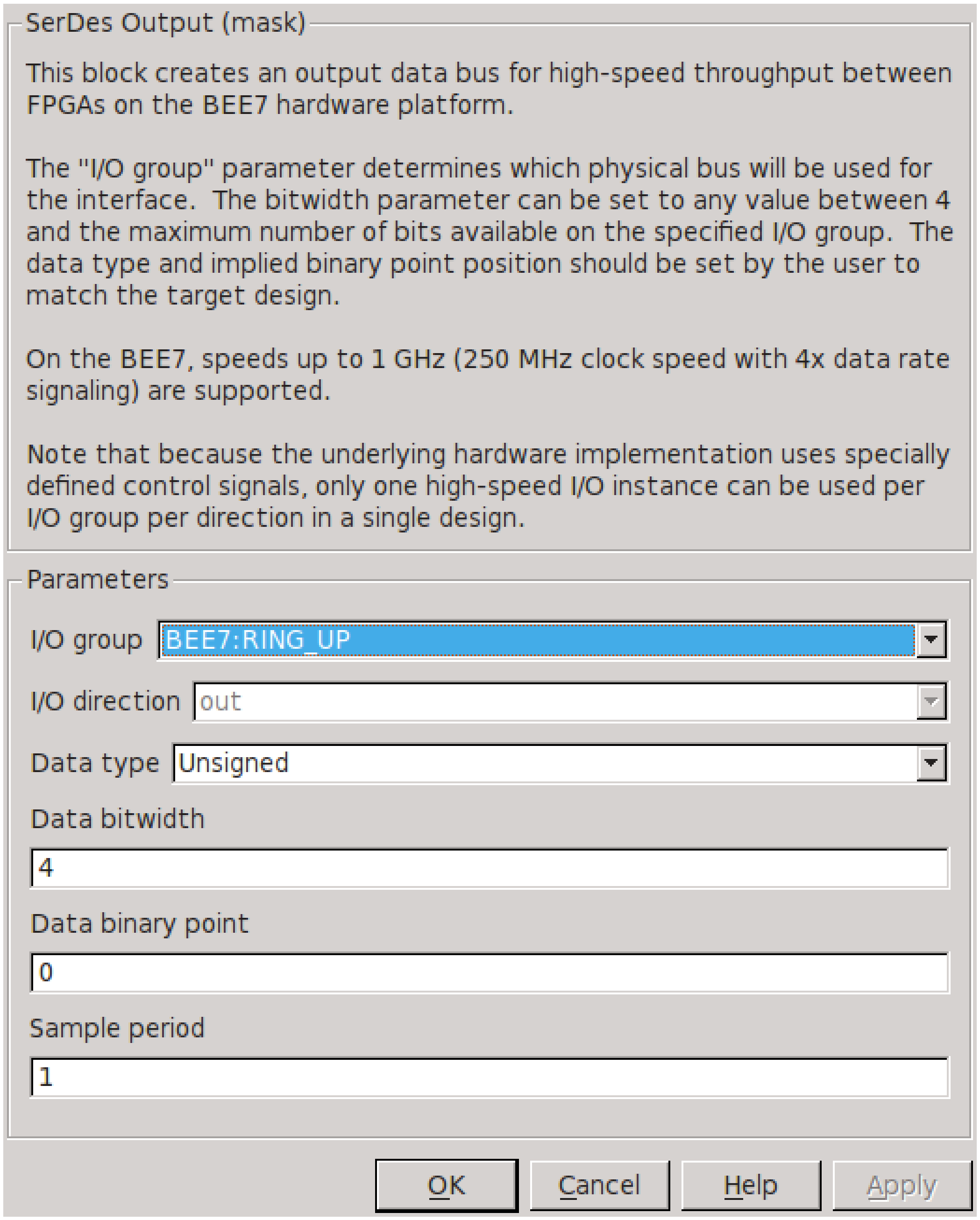


Figure 140: SerDes Output block


SerDes Output (mask)

This block creates an output data bus for high-speed throughput between FPGAs on the BEE7 hardware platform.

The "I/O group" parameter determines which physical bus will be used for the interface. The bitwidth parameter can be set to any value between 4 and the maximum number of bits available on the specified I/O group. The data type and implied binary point position should be set by the user to match the target design.

On the BEE7, speeds up to 1 GHz (250 MHz clock speed with 4x data rate signaling) are supported.

Note that because the underlying hardware implementation uses specially defined control signals, only one high-speed I/O instance can be used per I/O group per direction in a single design.

Parameters

I/O group: BEE7:RING_UP

I/O direction: out

Data type: Unsigned

Data bitwidth: 4

Data binary point: 0

Sample period: 1

Figure 141: SerDes Output dialog

The SerDes Output block provides an interface for sending data between FPGAs. Because the underlying hardware implementation uses specific pins on the FPGA, only one instance per I/O group can be present in a single design. Multiple logical signals can always be packed into a single SerDes Output block and using Concat blocks from the Xilinx System Generator Blockset. Similarly, the SerDes Output block cannot be combined with standard GPIO blocks assigned to the same I/O group.

Table 105: SerDes Output block parameters

Parameter	**Description**
I/O group	This parameter defines which group of FPGA pins will be used to assign the signal. This field will list all the pin groups available on the hardware platform for direct pin assignment.
I/O direction	Since the direction of signal flow is pre-determined for each end of the SerDes I/O interface to support automated calibration, this value is fixed to *out* for the output block.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. This can be any integer between 4 (the minimum width imposed by the underlying hardware implementation) and the maximum number of bits available for the given I/O group.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 106: SerDes Output port definitions

Port	**Direction**	**Bit Width**	**Description**

`data3`
`data2`
`data1`
`data0`
	in	*N*	Input ports which accept a hardware signal from the design with the System Generator signal parameters defined in the block configuration dialog (shown above). Note that because the SerDes Output block always uses signaling at 4x the clock rate, four data values are required per system clock cycle. The `data0` output represents the data to be sent first, and the `data3` output represents the data to be sent last.

`sim_out`	out	N/A	Output port which can be used to provide simulation data within Simulink. Note that this Simulink output signal is valid only during simulation and has no effect on the hardware. The sample time of this output will be 1/4 that of the hardware system (i.e. 4x the sample rate).

SerDes I/O NectarOS functions

BPS adds the following functions to NectarOS whenever a SerDes Input block is used in a design (the SerDes Output block is managed purely in hardware and does not require a software interface). The source code for these functions is contained in the *drivers/xps_hsio* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`serdes_io_init`

The `serdes_io_init` function is automatically called each time NectarOS is initialized, which occurs either after the FPGA is programmed or after a system reset. This function will automatically calibrate the hardware interface for guaranteed reliability at the current system frequency and temperature. If the clock frequency is changed (such as if an external clock source is being used) or if an extreme temperature change is suspected, the interface can be manually re-calibrated via the `serdes_io_calibrate_delays` command, which is described below. Note that this function requires that the transmit side of the SerDes interface is active before calibration will succeed. The function will continuously retry the calibration for approximately one minute before giving up (which can be necessary when the FPGAs in a system are programmed sequentially), after which the user will need to manually force a re-calibration.

Repeated function calls

(none)

Commands

`serdes_io_get_status <core>`

The `serdes_io_get_status` function will print a status for the specific core. The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

`serdes_io_reset <core>`

The `serdes_io_reset` function will trigger a SerDes digital interface reset for a specific core. The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

`serdes_io_test_pattern <core> <on|off>`

The `serdes_io_test_pattern` function will trigger a start or stop of the test pattern sequence for a specific core. The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

```
serdes_io_print_calibration_table <core>
```

The `serdes_io_print_calibration_table` function will print a table of the previous calibration results. The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

```
serdes_io_calibrate_delays <core>
```

The `serdes_io_calibrate_delays` function will force a re-calibration of the hardware interface. The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

```
serdes_io_set_delay <core> <bit_mask> <idelay_val>
```

The `serdes_io_set_delay` function will set all pins with their mask bits enabled to the fixed input delay value `idelay_val`. Each of the `bit_mask` arguments corresponds to a specific range of bits on the physical bus. Each mask is mapped to the bus in big-endian bit order (i.e. x80000000 corresponds to the first bit of the mask range). The `core` argument should be the name of the SerDes Input block in BPS (a list of known cores can be viewed via the `listdev` command).

SerDes I/O EDK API functions

The underlying hardware/software interface for the SerDes I/O component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the SerDes I/O interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any base package. Include the header *xserdes_out.h* or *xserdes_in.h* for access to the API.

Data Structures

There are three data structures used by several of the SerDes I/O EDK driver function calls which correspond to all the software-accessible control and status bits contained in the hardware interface logic. Each of these data structures are described in the following tables.

Table 107: `XSerDes_In_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base address of hardware instance
`io_up`	Unsigned 32 bit	I/O up bank
`io_center`	Unsigned 32 bit	I/O center bank
`io_down`	Unsigned 32 bit	I/O down bank

Table 108: `XSerDes_Out_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Enable interface calibration mode
`base_address`	Unsigned 32 bit	Base address of hardware instance

Table 109: `XSerDes_In_Status` Structure Contents

Variable name	Type	Definition
`sync`	Unsigned 32 bit	32 pins data bits synchronization state
`taps`	Unsigned 8 bit	32 pins tap value
`slips`	Unsigned 8 bit	Total number of slips experienced

Functions

```
XSerDes_In_Config *
XSerDes_In_GetConfig(void * baseaddr);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr` on the main processor PLB bus.

```
XSerDes_Out_Config *
XSerDes_Out_GetConfig(void * baseaddr);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr` on the main processor PLB bus.

```
U32
XSerDes_In_Init(XSerDes_In_Config * dev);
```

Initializes the SerDes device. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function. Returns an unsigned 32 bit value.

```
U32
XSerDes_Out_Init(XSerDes_Out_Config * dev);
```

Initializes the SerDes device. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_Out_GetConfig` function. Returns an unsigned 32 bit value.

```
void
XSerDes_In_SoftReset(XSerDes_In_Config * dev);
```

Performs a digital interface reset on the SerDes device. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
void
XSerDes_Out_SoftReset(XSerDes_Out_Config * dev);
```

Performs a digital interface reset on the SerDes device. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_Out_GetConfig` function.

```
void
XSerDes_In_CalibrateAllDelays(XSerDes_In_Config * dev);
```

Calibrates all data input delays. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
XSerDes_In_Status *
XSerDes_In_GetCalibrationStatus(XSerDes_In_Config * dev);
```

Reports the calibration status. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function. Returns a pointer to the status structure for the hardware instance described by the structure argument.

```
void
XSerDes_In_SetDataDelay(XSerDes_In_Config * dev
                        u32 bit_mask,
                        u8 tap);
```

Takes in arguments for an unsigned 32 bit mask and unsigned 8 bit tap value. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
void
XSerDes_In_PrintTapTable(XSerDes_In_Config * dev);
```

Displays a graphical representation of the input data tap values. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
void
XSerDes_In_EnableTestMode(XSerDes_In_Config * dev);
```

Starts the test pattern sequence. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
void
XSerDes_Out_EnableTestMode(XSerDes_Out_Config * dev);
```

Starts the test pattern sequence. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_Out_GetConfig` function.

```
void
XSerDes_In_EnableTestMode(XSerDes_In_Config * dev);
```

Stops the test pattern sequence. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_In_GetConfig` function.

```
void
XSerDes_Out_EnableTestMode(XSerDes_Out_Config * dev);
```

Stops the test pattern sequence. The argument `dev` should be a configuration structure pointer as returned by the `XSerDes_Out_GetConfig` function.

Shared BRAM

As one of several shared memory components in BPS, shared Block RAM can be used to access medium-sized blocks of data in both hardware and software. In addition to the underlying physical hardware, BPS also creates a name for each Shared BRAM block in the design and several utility commands within NectarOS for interacting with the component at runtime. An EDK driver is also included for use with custom software applications. Shared BRAM is platform independent and is contained in the BPS Common Blockset.

Shared BRAM block

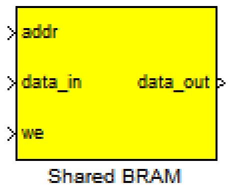
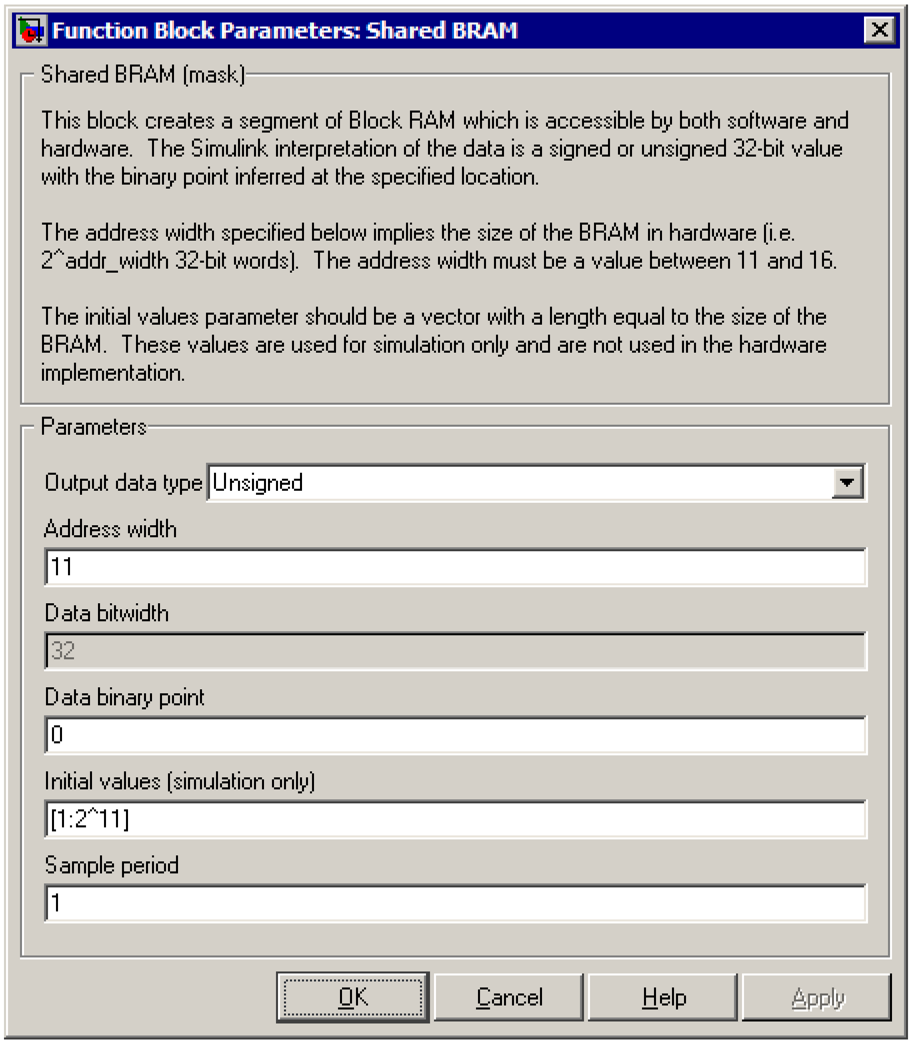


Figure 142: Shared BRAM block



Shared BRAM (mask)

This block creates a segment of Block RAM which is accessible by both software and hardware. The Simulink interpretation of the data is a signed or unsigned 32-bit value with the binary point inferred at the specified location.

The address width specified below implies the size of the BRAM in hardware (i.e. 2^addr_width 32-bit words). The address width must be a value between 11 and 16.

The initial values parameter should be a vector with a length equal to the size of the BRAM. These values are used for simulation only and are not used in the hardware implementation.

Parameters

Output data type: Unsigned

Address width: 11

Data bitwidth: 32

Data binary point: 0

Initial values (simulation only): [1:2^11]

Sample period: 1

Figure 143: Shared BRAM dialog

The Shared BRAM block creates a dual-port Block RAM on the FPGA which is shared between software and hardware. The hardware interface is connected to port A of the Block RAM, and the processor is connected to port B of the Block RAM via a PLB/AXI slave core.

Table 110: Shared BRAM block parameters

Parameter	**Description**
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Address width	This value determines the width of the address bus in bits. Consequently, the total size of the Block RAM will be 2^N words, where *N* is the address width. This value must be between 11 and 16 due to inherent limits of the Block RAM interface.
Data bitwidth	This value defines the total width of the signal in bits. It is currently fixed at 32 bits to match the embedded processor word size.
Data binary point	This value determines the inferred binary point position for the signal.
Initial values	The value entered in this field must evaluate to either a scalar or an array of real numbers with the same length as the Block RAM size. The initial contents of the Block RAM will be taken from this array and mapped to each address sequentially. If the value is a scalar, the entire Block RAM is initialized to a constant value. *This parameter is only used during simulation and does not initialize the physical Block RAM in the FPGA configuration bitstream.*
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 111: Shared BRAM port definitions

Port	Direction	Bit Width	Description
`addr`	in	Variable	The address to be loaded from the Block RAM on the next cycle.
`data_in`	in	32	Data to be written into Block RAM.
`we`	in	1	When asserted high, writes the current value of `data_in` to the address present on `addr`.
`data_out`	out	32	The data contents in Block RAM loaded from the previous value of `addr`.

Shared BRAM NectarOS functions

BPS directly maps each Block RAM into a segment of the address space and adds it to the known devices list within NectarOS. The name of the Block RAM is again inherited from the Simulink block itself. BPS adds the following functions to NectarOS whenever one or more Shared BRAM blocks are used in a design. The source code for these functions are contained in the *drivers/xps_bram* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`bramread`

Usage: `bramread <bram_name> <offset>`

The `bramread` command will read one 32-bit word from a Shared BRAM core at the given offset within BRAM. The resulting value will be printed to the console.

`bramwrite`

Usage: `bramwrite <bram_name> <offset> <value>`

The `bramwrite` command will write the given value as a 32-bit word to a Shared BRAM core at the given offset within BRAM.

`bramdump`

Usage: `bramdump <bram_name> <size>`

The `bramdump` command will read the specified number of words (starting at offset zero) from a Shared BRAM block and print the results to the console.

Shared BRAM EDK API functions

The underlying hardware/software interface for the Shared BRAM component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the Shared BRAM interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any base package. Include the header *xshbram.h* for access to the API.

Data Structures

There is one data structure used by most Shared BRAM EDK driver function calls which defines the specific configuration details of each hardware instance.

Table 112: `XShBram_Config` Structure Contents

Variable name	**Type**	**Definition**
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base byte address of hardware instance
`high_address`	Unsigned 32 bit	Highest byte address of hardware instance

Functions

```
XShBram_Config *
XShBram_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor bus. The return value of this function should be used as the `XShBram_Handle` argument required by other driver routines.

```
XStatus
XShBram_ReadWord(XShBram_Handle handle,
                 Xuint32 offset,
                 Xuint32 * buf);
```

Reads a single word from BRAM at offset `offset` and writes the result into the location referenced by `buf`. The argument `handle` should be a configuration structure pointer as returned by the `XShBram_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShBram_ReadBlock(XShBram_Handle handle,
                  Xuint32 offset,
                  Xuint32 * buf,
                  Xuint32 count);
```

Reads a block of `count` words from BRAM starting at offset `offset` and writes the results starting at the location referenced by `buf`. The argument `handle` should be a configuration structure pointer as returned by the `XShBram_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShBram_WriteWord(XShBram_Handle handle,
                 Xuint32 offset,
                 Xuint32 value);
```

Writes the word `value` into BRAM at offset `offset`. The argument `handle` should be a configuration structure pointer as returned by the `XShBram_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShBram_WriteBlock(XShBram_Handle handle,
                  Xuint32 offset,
                  Xuint32 * buf,
                  Xuint32 count);
```

Writes a block of `count` words starting from the location referenced by `buf` into BRAM starting at offset `offset`. The argument `handle` should be a configuration structure pointer as returned by the `XShBram_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful

Shared FIFO

The shared FIFO is another memory component which can be used to access data in both hardware and software. In addition to the underlying physical hardware, BPS also creates a name for each Shared FIFO block in the design and several utility commands within NectarOS for interacting with the component at runtime. An EDK driver is also included for use with custom software applications. Shared FIFOs are platform independent and their library block is contained in the BPS Common Blockset.

Shared FIFO block

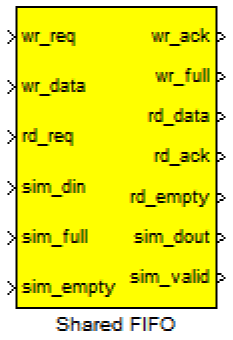
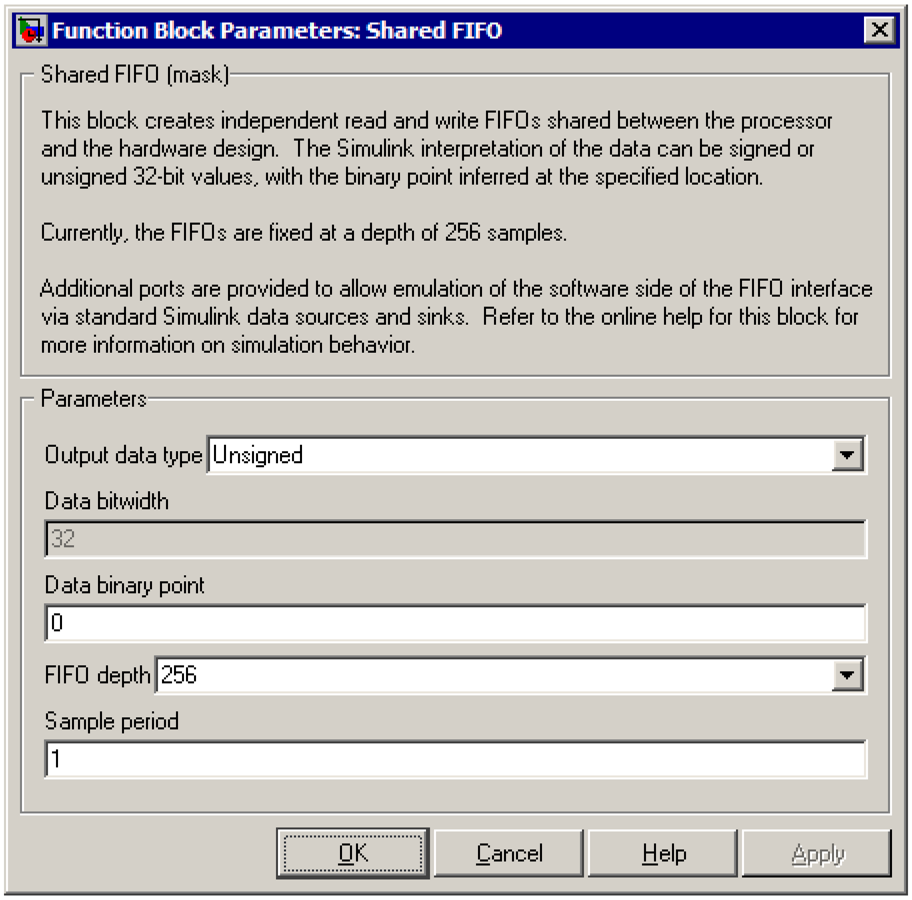


Figure 144: Shared FIFO block



Shared FIFO (mask)

This block creates independent read and write FIFOs shared between the processor and the hardware design. The Simulink interpretation of the data can be signed or unsigned 32-bit values, with the binary point inferred at the specified location.

Currently, the FIFOs are fixed at a depth of 256 samples.

Additional ports are provided to allow emulation of the software side of the FIFO interface via standard Simulink data sources and sinks. Refer to the online help for this block for more information on simulation behavior.

Parameters

Output data type: Unsigned

Data bitwidth: 32

Data binary point: 0

FIFO depth: 256

Sample period: 1

Figure 145: Shared FIFO dialog

The Shared FIFO block creates a bidirectional FIFO (both read and write ports) accessible by both software and hardware. Data values pushed into the FIFO by the embedded processor can be read by the hardware, and values written into the FIFO by the hardware can be read by the embedded processor. Full and empty signals are provided on both ends to assist with flow control.

BPS automatically allocates space in the address map for the FIFO control registers and connects the embedded processor to the FIFO via a PLB/AXI slave core.

Table 113: Shared FIFO block parameters

Port	**Description**
Output data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. It is currently fixed at 32 bits to match the embedded processor word size.
Data binary point	This value determines the inferred binary point position for the signal.
FIFO depth	This value determines the depth of the FIFO that will be created. This can range from 4 to 16,384.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 114: Shared FIFO port definitions

Parameter	**Direction**	**Bit Width**	**Description**
`wr_req`	in	1	Notifies the write FIFO that the current wr_data is valid and should be written.
`wr_data`	in	32	Input data for write FIFO.
`rd_req`	in	1	Requests a data value from read FIFO.
`wr_ack`	out	1	Indicates that the previous cycle's write request was valid and data was accepted.
`wr_full`	out	1	Indicates that the write FIFO is full and cannot accept new data.
`rd_data`	out	32	Output data for read FIFO.
`rd_ack`	out	1	Indicates that the current rd_data is valid and was removed from the read FIFO.
`rd_empty`	out	1	Indicates that the read FIFO is empty and cannot provide new data.
`sim_din` `sim_full` `sim_empty` `sim_dout` `sim_valid`	-	-	Simulink source and sink blocks can be used in conjunction with these ports to provide a model for processor data operations during simulation. A very simple pass-through emulation mechanism is used, which is shown in Figure 146 below.

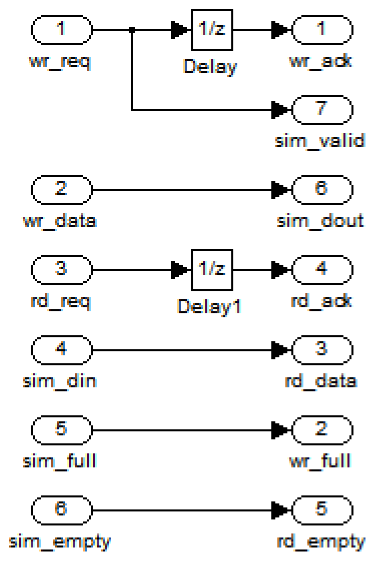


Figure 146: FIFO behavior during simulation

Shared FIFO NectarOS functions

Shared FIFO blocks will be added to the known devices list within NectarOS, and each will inherit its name from the Simulink block itself. BPS also adds the following functions to NectarOS whenever one or more Shared BRAM blocks are used in a design. The source code for these functions are contained in the *drivers/xps_fifo* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`fifo_pop`

Usage: `fifo_pop <fifo_name> <blocking (0|1)>`

The `fifo_pop` command will pop one 32-bit word from the specified Shared FIFO. The blocking argument can be either 0 or 1. If 1, this command will poll the read FIFO until it is not empty, effectively blocking the shell until data is present. If 0, the command will simply read the current (potentially invalid) FIFO output without regard to the empty bit.

`fifo_push`

Usage: `fifo_push <fifo_name> <blocking (0|1)> <value>`

The `fifo_push` command will push the given value as a 32-bit word into the specified Shared FIFO. The blocking argument can be either 0 or 1. If 1, this command will poll the write FIFO until it is not full, effectively blocking the shell until space is available. If 0, the command will simply write the value into the FIFO without regard to the full bit.

`fifo_reset`

Usage: `fifo_reset <fifo_name>`

The `fifo_reset` command will reset both the read and write FIFOs, purging any data currently stored in the FIFO and restoring all signals to their initial values.

Shared FIFO EDK API functions

The underlying hardware/software interface for the Shared FIFO component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the Shared FIFO interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any base package. Include the header *xshfifo.h* for access to the API.

Data Structures

There is one data structure used by most Shared FIFO EDK driver function calls which defines the specific configuration details of each hardware instance. In addition, a device status structure is defined which contains all relevant status bits exported by the hardware component.

Table 115: `XShFifo_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base address of hardware instance
`fifo_depth`	Unsigned 32 bit	Depth (in words) of the FIFO component

Table 116: `XShFifo_Status` Structure Contents

Variable name	Type	Definition
`rdfifo_empty`	Unsigned 8 bit	Read FIFO empty condition
`rdfifo_almost_empty`	Unsigned 8 bit	Read FIFO almost empty condition
`rdfifo_deadlock`	Unsigned 8 bit	Read FIFO deadlock condition
`rdfifo_occ_scaling`	Unsigned 8 bit	Read FIFO occupancy scaling enabled
`rdfifo_data_width`	Unsigned 8 bit	Read FIFO encoded data port width
`rdfifo_occupancy`	Unsigned 16 bit	Read FIFO occupancy
`wrfifo_full`	Unsigned 8 bit	Write FIFO full condition
`wrfifo_almost_full`	Unsigned 8 bit	Write FIFO almost full condition
`wrfifo_deadlock`	Unsigned 8 bit	Write FIFO deadlock condition
`wrfifo_vac_scaling`	Unsigned 8 bit	Write FIFO vacancy scaling enabled
`wrfifo_data_width`	Unsigned 8 bit	Write FIFO encoded data port width
`wrfifo_dre_present`	Unsigned 8 bit	Write FIFO DRE present
`wrfifo_vacancy`	Unsigned 16 bit	Write FIFO vacancy

Functions

```
XShFifo_Config *
XShFifo_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor bus. The return value of this function should be used as the `XShFifo_Handle` argument required by other driver routines.

```
XStatus
XShFifo_Reset(XShFifo_Handle handle);
```

Resets both the read and write FIFOs. The argument `handle` should be a configuration structure pointer as returned by the `XShFifo_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShFifo_GetStatus(XShFifo_Handle handle,
                  XShFifo_Status * status);
```

Reads both the read and write FIFO status registers and fills all corresponding values in the Shared FIFO status structure referenced by `status`. The argument `handle` should be a configuration structure pointer as returned by the `XShFifo_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShFifo_Push(XShFifo_Handle handle,
             Xuint32 value,
             Xuint8 blocking);
```

Pushes the 32-bit word `value` into the write FIFO. If `blocking` evaluates as true, the write FIFO full bit will be polled until it is low before pushing data, otherwise the data is pushed without regard to the full bit. The argument `handle` should be a configuration structure pointer as returned by the `XShFifo_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XShFifo_Pop(XShFifo_Handle handle,
             Xuint32 * value_p,
             Xuint8 blocking);
```

Pops a 32-bit word from the read FIFO and writes it into the location referenced by `value_p`. If `blocking` evaluates as true, the read FIFO empty bit will be polled until it is low before popping data, otherwise the data is popped without regard to the empty bit. The argument `handle` should be a configuration structure pointer as returned by the `XShFifo_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

Software Register

As one of several shared memory components in BPS, software registers can be used to access single 32-bit data values in both hardware and software. In addition to the underlying physical hardware, BPS also creates a name for each Software Register block in the design and several utility commands within NectarOS for interacting with the component at runtime. An EDK driver is also included for use with custom software applications. Software Registers are platform independent and their library block is contained in the BPS Common Blockset.

Software Register blocks

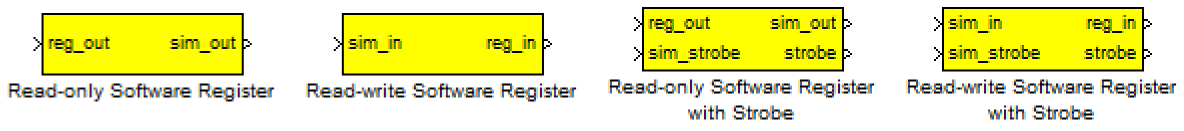
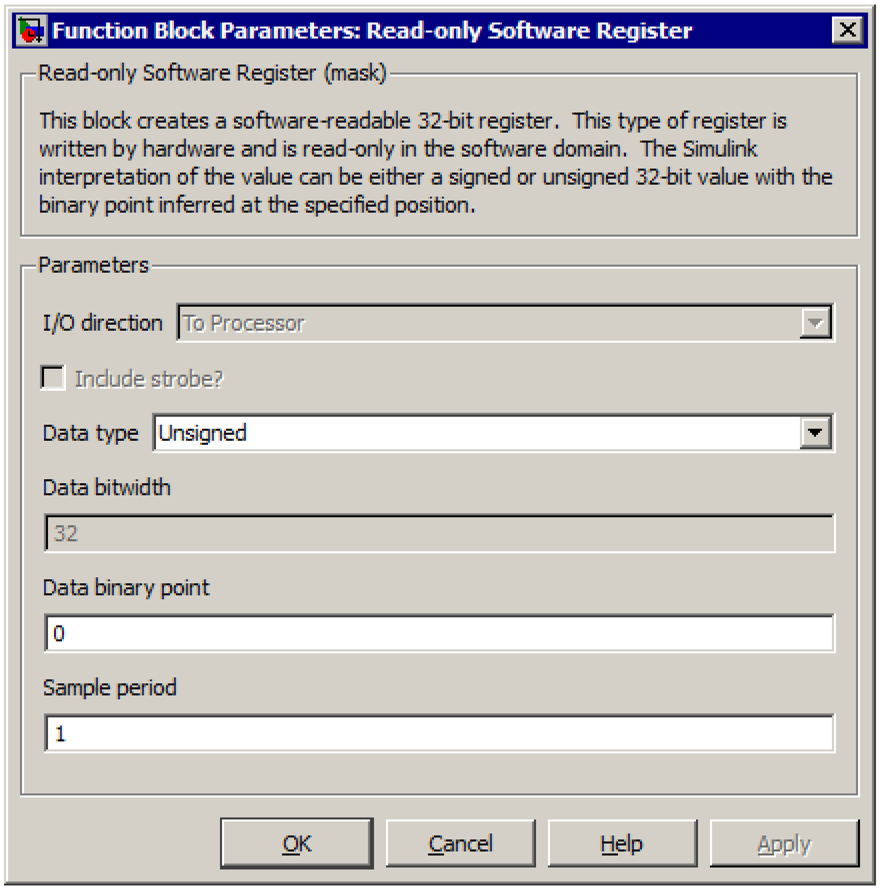

reg_out
sim_out
Read-only Software Register
sim_in
reg_in
Read-write Software Register
reg_out
sim_out
sim_strobe
strobe
Read-only Software Register with Strobe
sim_in
reg_in
sim_strobe
strobe
Read-write Software Register with Strobe

Figure 147: Software Register blocks


Function Block Parameters: Read-only Software Register

Read-only Software Register (mask)

This block creates a software-readable 32-bit register. This type of register is written by hardware and is read-only in the software domain. The Simulink interpretation of the value can be either a signed or unsigned 32-bit value with the binary point inferred at the specified position.

Parameters

I/O direction: To Processor

Include strobe? [ ]

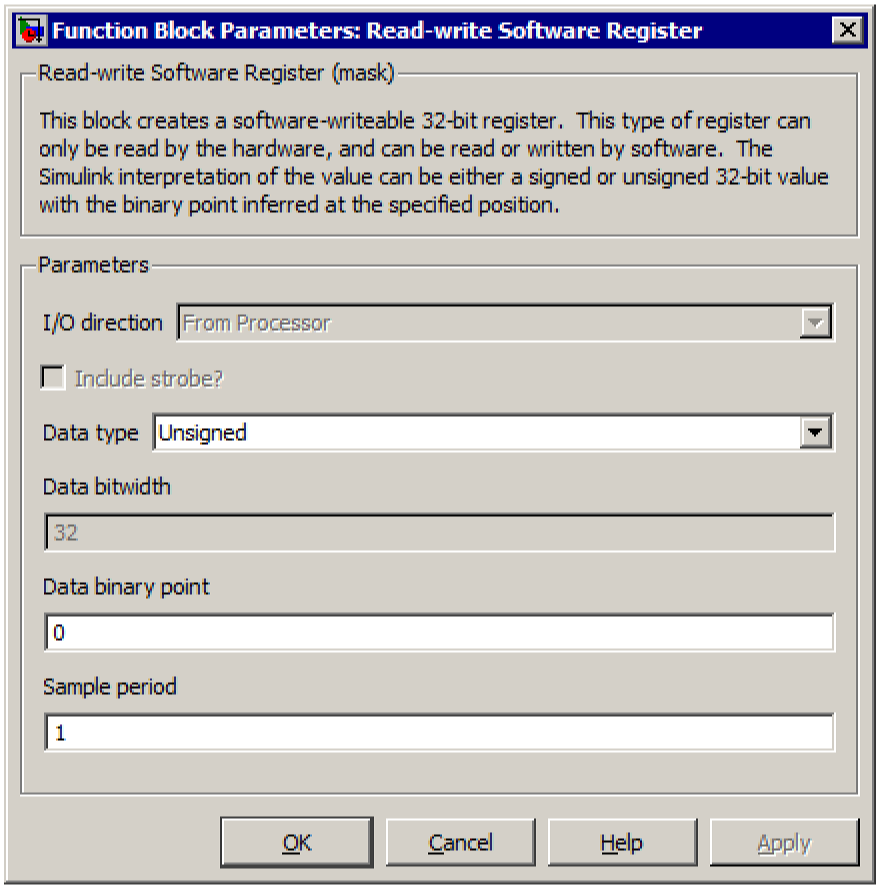
Data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Sample period: 1

Figure 148: Read-only Software Register dialog


Function Block Parameters: Read-write Software Register

Read-write Software Register (mask)

This block creates a software-writeable 32-bit register. This type of register can only be read by the hardware, and can be read or written by software. The Simulink interpretation of the value can be either a signed or unsigned 32-bit value with the binary point inferred at the specified position.

Parameters

I/O direction: From Processor

☐ Include strobe?

Data type: Unsigned

Data bitwidth: 32

Data binary point: 0

Sample period: 1

Figure 149: Read-write Software Register dialog

The Software Register block defines a named 32-bit register which is accessible by both the embedded processor and the hardware design. Each software register can either be written to by the processor and act as an input to the hardware design, or be read by the processor and act as an output from the hardware design. An optional strobe output is also available for applications where the hardware should be notified when the processor has accessed the register. This strobe will be pulsed for exactly one cycle when the processor reads the register value (for read-only registers) or writes a new register value (for read-write registers). Note that a strobe will not be generated when a read-write register is read, as this has no impact on the hardware.

Each software register is automatically mapped the address space and bus architecture, and acts as a slave on the PLB/AXI bus.

Table 117: Software Register block parameters

Parameter	Description
I/O direction	This value determines the direction of data flow for the register. A *From Processor* register can be written to by the processor and acts as a data source to the hardware. A *To Processor* register acts as a data sink in hardware which can be ready by the processor. A *From Processor* register can also be read (for example, to double-check the current value), but a *To Processor* register can never be written in software. The I/O direction is pre-selected based on whether the block is Read-only or Read-write.
Include strobe?	This parameter determines whether the underlying hardware will generate a one-cycle strobe whenever the register is accessed by software.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. It is currently fixed at 32 bits to match the embedded processor word size.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 118: Software Register port definitions

Port	Description
`sim_in` `reg_in`	*From Processor:* The `sim_in` input can be driven by standard Simulink blocks to provide data values during simulation. The `reg_in` output provides a System Generator value with the specified data type and precision.
`reg_out` `sim_out`	*To Processor:* The `reg_out` input receives the System Generator data which will be written into the register. The `sim_out` output can be used to drive a Scope or other Simulink block for data capture during simulation.

Software Register NectarOS functions

BPS automatically manages the address map and processor bus structure in hardware and adds the software register to the list of known devices in NectarOS. The human-readable name of the register within NectarOS is taken directly from the name of the Software Register block itself in Simulink.

BPS adds the following functions to NectarOS whenever one or more Software Register blocks are used in a design. The source code for these functions are contained in the *drivers/xps_sw_reg* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`regread`

Usage: `regread <reg_name>`

The `regread` command will read the contents of a 32-bit Software Register and print the resulting value to the console.

`regreadall`

Usage: `regreadall`

The `regreadall` command will read the contents of all Software Registers in the design and print each register name and its value to the console.

`regwrite`

Usage: `regwrite <reg_name> <value>`

The `regwrite` command will write the given value to a Software Register as a 32-bit word.

Software Register EDK API functions

The underlying hardware/software interface for the Software Register component is implemented as a single EDK pcore/driver combination. The EDK driver contains several data structures and functions which can be used by custom software applications to gain low-level access to the software register interface. Partial source code for the EDK pcore (hardware) and driver (software) can be found in the *pcores* and *drivers* subdirectories of any base package. Include the header *xswreg.h* for access to the API.

Data Structures

There is one data structure used by most Software Register EDK driver function calls which defines the specific configuration details of each hardware instance.

Table 119: `XSwReg_Config` Structure Contents

Variable name	Type	Definition
`device_id`	Unsigned 8 bit	Unique ID for hardware instance
`base_address`	Unsigned 32 bit	Base address of hardware instance
`is_readable`	Unsigned 8 bit	Indicates if register is readable from software
`is_writeable`	Unsigned 8 bit	Indicates if register is writeable from software

Functions

```
XSwReg_Config *
XSwReg_GetConfig(void * baseaddr_p);
```

Returns a pointer to the configuration structure for the hardware instance located at address `baseaddr_p` on the main processor bus. The return value of this function should be used as the `XSwReg_Handle` argument required by other driver routines.

```
XStatus
XSwReg_RegWrite(XSwReg_Handle handle,
                Xuint32 data);
```

Performs necessary low-level handshake to write the 32-bit value `data` into the hardware register. The argument `handle` should be a configuration structure pointer as returned by the `XSwReg_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

```
XStatus
XSwReg_RegRead(XSwReg_Handle handle,
               Xuint32 * data_p);
```

Performs the necessary low-level handshake to read the 32-bit value from the hardware register, and writes the result into the location referenced by `data_p`. The argument `handle` should be a configuration structure pointer as returned by the `XSwReg_GetConfig` function. Returns a Xilinx driver status code, which is `XST_SUCCESS` when successful.

SRAM

BPS provides a direct interface to the on-board SRAM chip on the ML50x hardware platform, which is 256K words deep with a word size of 36 bits (four bytes, each with parity bits). The hardware interface generated by BPS manages all the low-level physical signals of the SRAM chip interface (clock/chip enables, tri-state data bus, etc.) and exposes only the functional data ports to the Simulink design domain. There are no software components required by the SRAM block.

SRAM block for ML50x

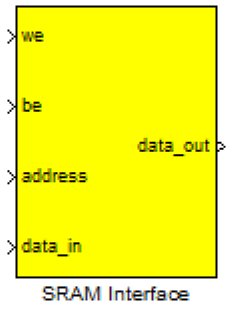
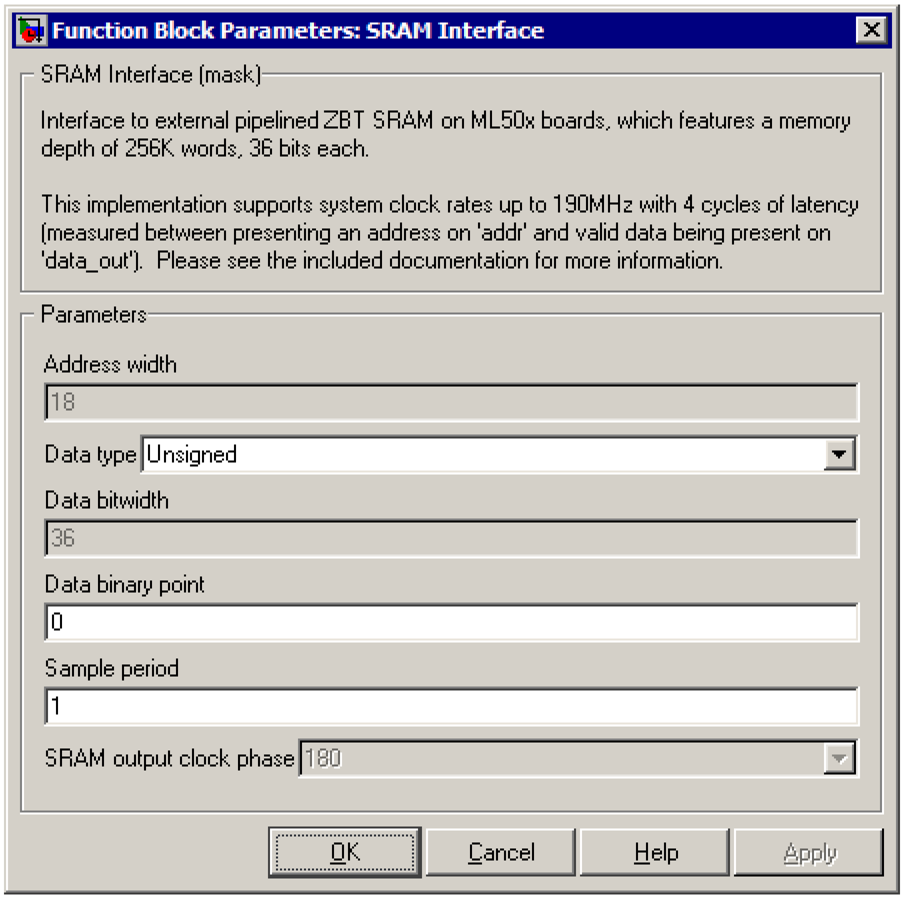


Figure 150: SRAM block


Function Block Parameters: SRAM Interface

SRAM Interface (mask)

Interface to external pipelined ZBT SRAM on ML50x boards, which features a memory depth of 256K words, 36 bits each.

This implementation supports system clock rates up to 190MHz with 4 cycles of latency (measured between presenting an address on 'addr' and valid data being present on 'data_out'). Please see the included documentation for more information.

Parameters

Address width: 18

Data type: Unsigned

Data bitwidth: 36

Data binary point: 0

Sample period: 1

SRAM output clock phase: 180

Figure 151: SRAM dialog

Table 120: SRAM block parameters

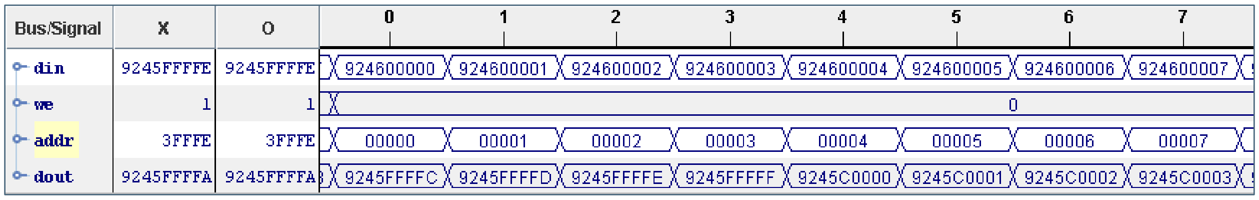
Parameter	**Description**
Address width	This parameter sets the width of the SRAM address port. It is fixed at 18 bits to match the SRAM chip configuration.
Data type	This parameter determines the inferred arithmetic type for the signal. This can be either *Unsigned* or *Signed (2's comp)*.
Data bitwidth	This value defines the total width of the signal in bits. It is fixed at 36 bits to match the SRAM chip configuration.
Data binary point	This value determines the inferred binary point position for the signal.
Sample period	The Simulink sample period to be declared for this block.

SRAM output clock phase	This parameter determines the phase of the system clock to be used as the output clock driven to the SRAM chip. It is currently fixed at 180 degrees, which achieves the highest peak frequency in most cases.

Table 121: SRAM port definitions

Port	**Direction**	**Bit Width**	**Description**
`we`	in	1	Indicates a write operation, active high. The value on `data_in` will be written to SRAM at the address `addr` when this signal is asserted.
`be`	in	4	Byte-enable for write operations, active high. Each bit controls whether a corresponding 9-bit “byte” from the data bus will be written into SRAM (`be[0]` controls `data_in[0:8]`, `be[1]` controls `data_in[9:17]`, etc.).
`addr`	in	18	The SRAM address for all operations.
`data_in`	in	36	The data which will be written into SRAM when we is asserted.
`data_out`	out	36	The data stored in SRAM at the address `addr` when `we` is not asserted. Note that there are 4 cycles of latency between the desired address being driven on `addr` and the corresponding valid data appearing at `data_out`.

The SRAM chip itself is pipelined, which causes two cycles of latency within the chip. The physical hardware interface generated by BPS requires one cycle of latency at the I/O pads of the FPGA, which results in a net latency of 4 cycles between asserting an address and receiving valid data on the data read port. The timing of the data bus is shown in the figure below, which was captured at runtime by ChipScope.



Bus/Signal	X	O	0	1	2	3	4	5	6	7
din	9245FFFFE	9245FFFFE	924600000	924600001	924600002	924600003	924600004	924600005	924600006	924600007
we	1	1						0		
addr	3FFFE	3FFFE	00000	00001	00002	00003	00004	00005	00006	00007
dout	9245FFFFA	9245FFFFA	9245FFFFC	9245FFFFD	9245FFFFE	9245FFFFF	9245C0000	9245C0001	9245C0002	9245C0003

Figure 152: SRAM block hardware timing model

VGA

BPS provides an input interface to the VGA video transceiver chip on the ML50x hardware platform. In addition to the physical hardware interface, several utility functions are added to NectarOS for configuring the interface logic and the IC itself, and an EDK software driver is included as well for use in custom software applications.

VGA block for ML50x

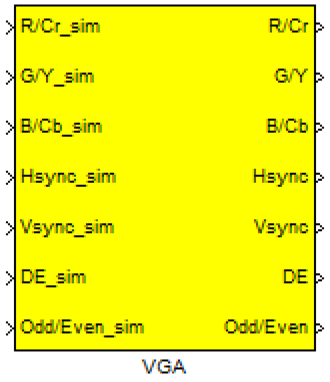
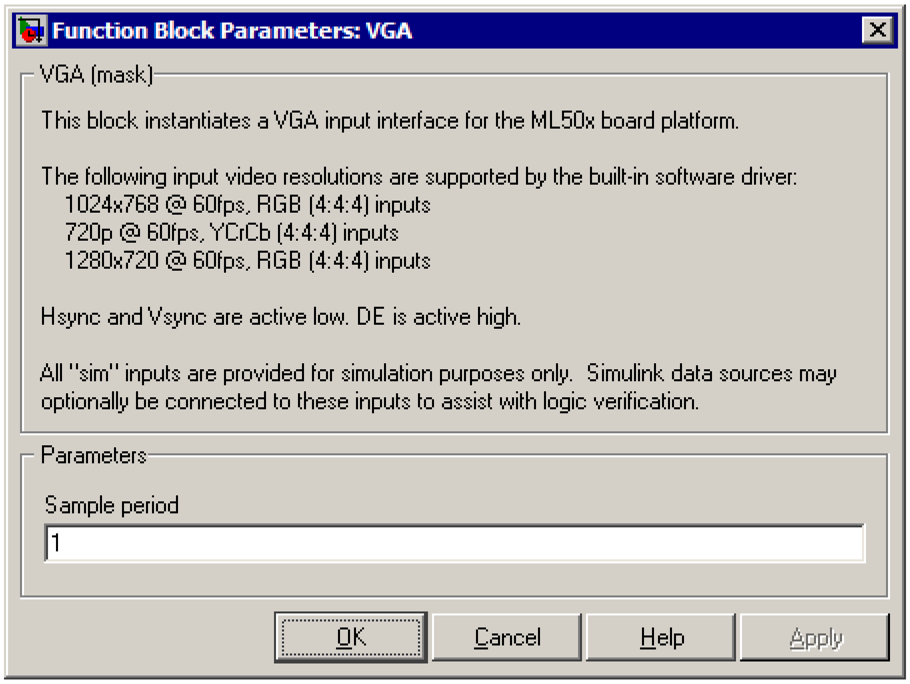


Figure 153: VGA block for ML50x



This block instantiates a VGA input interface for the ML50x board platform.

The following input video resolutions are supported by the built-in software driver:
1024x768 @ 60fps, RGB (4:4:4) inputs
720p @ 60fps, YCrCb (4:4:4) inputs
1280x720 @ 60fps, RGB (4:4:4) inputs

Hsync and Vsync are active low. DE is active high.

All "sim" inputs are provided for simulation purposes only. Simulink data sources may optionally be connected to these inputs to assist with logic verification.

Sample period: 1

Figure 154: VGA dialog for ML50x

The VGA block implements a Video Graphics Array input via the Analog Devices AD9980 device and the VGA connector on the ML50x board. The block functions as a data source in a Simulink model. The video data comes out of the block as three 8-bit buses represented as RGB or YCrCB with a 4:4:4 format.

The VGA block also outputs horizontal sync (`Hsync`) and vertical sync (`Vsync`) pulses in addition to data enable (`DE`) and odd/even field (`Odd/Even`) control outputs.

All VGA block outputs have corresponding simulation inputs (`R/Cr_sim, G/Y_sim, B/Cb_sim, Hsync_sim, Vsync_sim, DE_sim, and Odd/Even_sim`) that allow you to emulate video inputs in your Simulink model.

The VGA block hardware is configured through an I^{2}C bus that communicates with the BPS embedded system. BEEcube provides API routines that simplify initializing and configuring the VGA block hardware for different video formats. The VGA API is described later in this section.

Table 122: VGA block parameters for ML50x

Parameter	**Description**
Sample period	The Simulink sample period to be declared for this block

Table 123: VGA port definitions for ML50x

Port	**Direction**	**Bit Width**	**Description**
`R/Cr`	out	8	RGB Red or YCrCb Cr video data in 4:4:4 format; encoding selected by configuration register.
`R/Cr_sim`	in	8	RGB Red or YCrCb Cr video data simulation input for software emulation of external source.
`G/Y`	out	8	RGB Green or YCrCb Y video data in 4:4:4 format; encoding selected by configuration register.
`G/Y_sim`	in	8	RGB Green or YCrCb Y video data simulation input.
`B/Cb`	out	8	RGB Blue or YCrCb Cb video data in 4:4:4 format; encoding selected by configuration register.
`B/Cb_sim`	in	8	RGB Blue or YCrCb Cb video data simulation input.
`Hsync`	out	1	Horizontal sync pulse. Active low.
`Hsync_sim`	in	1	Horizontal sync simulation input.
`Vsync`	out	1	Vertical sync pulse. Active low.
`Vsync_sim`	in	1	Vertical sync simulation input.

`DE`
 	
`DE_sim`
	out
	
	in
		1
		
		1
			Video Data Enable. Active high.
			
			Data Enable simulation input.

`Odd/Even`
 	
`Odd/Even_sim`
	out
	
	in
		1
		
		1
			For interlaced video, high indicates odd field, low indicates even field.
			Odd/Even simulation input.

VGA block NectarOS functions

The VGA block includes a set of utility functions that enable the user to control the VGA block directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever the VGA block is used in a design. The source code for these functions are contained in the *drivers/xps_vga* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

(none)

Repeated function calls

(none)

Commands

`vga_set_format <mode>`

The `vga_set_format` function takes an integer as an argument. It selects the video mode of the VGA input device.

Table 124: `vga_set_format` mode values

Value	**Description**
0	XGA
1	720p
2	1280x720

`vga_adjust <int H_BP> <int V_BP>`

The arguments to `vga_adjust` are type integer. The `vga_adjust` command adjusts the screen position horizontally by `H_BP` pixels and vertically by `V_BP` pixels.

VGA block EDK API functions

The EDK API used by the VGA block is shared with other I^{2}C devices. It encapsulates common I^{2}C device register write operations as C functions that can be used in embedded application programs. Include the header *xps_iic_wo.h* for access to the API.

Functions

The following EDK function is used by the VGA block:

```
XStatus iic_wo_write(void *baseaddr_p, u8 dev_id, u8 address, u8 val);
```

Sets the I^{2}C device `dev_id` configuration register `address` to `val`.
Returns boolean `TRUE` when operation successful.

The VGA I^{2}C `dev_id` is 0x98 for Xilinx ML50x emulation platform boards.

Refer to Analog Devices *AD9980 Data Sheet* (D04740-0-1/05(0), Rev. 0) for a description of the AD9980 device register address `address` and function of the data `val`.

XAUI

BPS provides support for accessing the native XAUI interface of the NetLogic AEL2005 PHY devices on the HiTech Global HTG-FMC-SFP-PLUS board on the ML60x hardware platform. Use of this interface also requires a BEEcube FMC102 voltage translator board, which allows the Virtex-6 FPGA to safely control the 3.3V MDIO port of the PHY devices.

XAUI block for ML60x

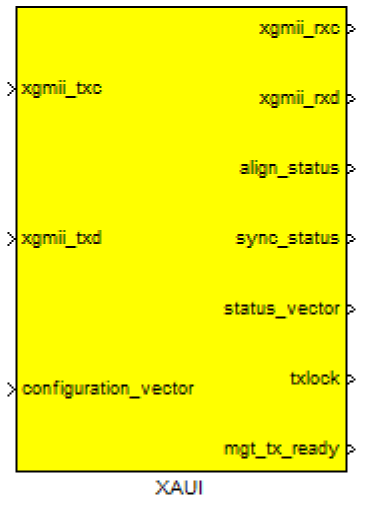
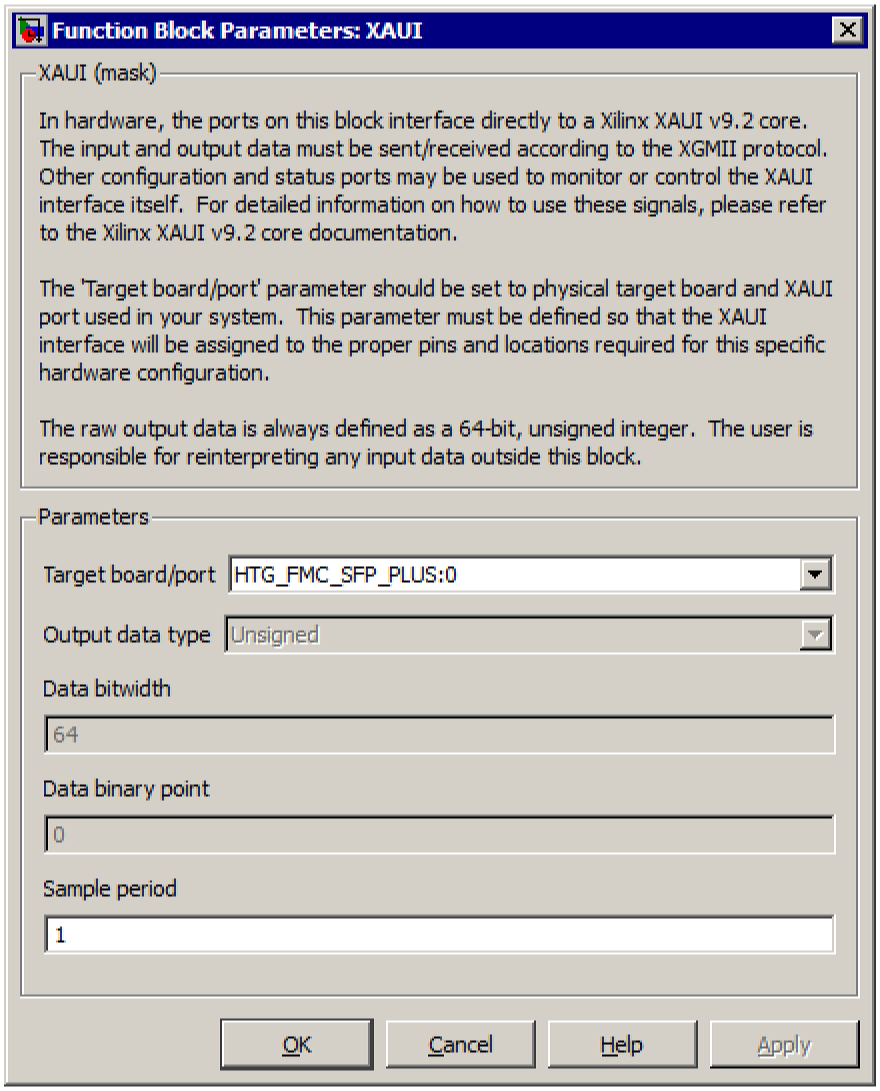


Figure 155: XAUI block for ML60x

The XAUI block provides an interface which is identical to that of the Xilinx XAUI v9.2 component in Core Generator. The data inputs and outputs adhere to the standard XGMII protocol. For more information on the usage of the auxiliary control and status ports, please refer to the XAUI core documentation in Core Generator.

Use of this block requires that the system clock be sourced by the 156.25MHz output clock from the FMC board, as the XGMII interface must run synchronously to the XAUI core.


Function Block Parameters: XAUI

XAUI (mask)

In hardware, the ports on this block interface directly to a Xilinx XAUI v9.2 core. The input and output data must be sent/received according to the XGMII protocol. Other configuration and status ports may be used to monitor or control the XAUI interface itself. For detailed information on how to use these signals, please refer to the Xilinx XAUI v9.2 core documentation.

The 'Target board/port' parameter should be set to physical target board and XAUI port used in your system. This parameter must be defined so that the XAUI interface will be assigned to the proper pins and locations required for this specific hardware configuration.

The raw output data is always defined as a 64-bit, unsigned integer. The user is responsible for reinterpreting any input data outside this block.

Parameters

Target board/port: HTG_FMC_SFP_PLUS:0

Output data type: Unsigned

Data bitwidth: 64

Data binary point: 0

Sample period: 1

Figure 156: XAUI dialog for ML60x

Table 125: XAUI block parameters for ML60x

Parameter	Description
Data type	This parameter determines the inferred arithmetic type for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to *Unsigned*.
Data bitwidth	This value defines the total width of the signal in bits. The standard data rate (SDR) version of XGMII requires that this value be fixed to 64 bits.
Data binary point	This value determines the inferred binary point position for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to zero.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 126: XAUI port definitions for ML60x

Port	Direction	Bit Width	Description
`xgmii_txc` `xgmii_txd`	in	8 64	XGMII transmit control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`configuration_vector`	in	7	Vector of configuration bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.
`xgmii_rxc` `xgmii_rxd`	out	8 64	XGMII receive control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`align_status` `sync_status`	out	1 4	Vectors indicating high-speed serial lane alignment and synchronization status. Alignment status is a single bit indicating that all 4 XAUI lanes are aligned, and synchronization status is a 4-bit value with the individual link status of each lane.
`status_vector`	out	8	Vector of status bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.

`txlock`	out	1	Status bit indicating that the GTX transmitter clock is locked. *This signal is intended for debug use only and may be removed in a future release.*
`mgt_tx_ready`	out	1	Status bit indicating that the GTX transmitter interface is ready to accept outgoing traffic. *This signal is intended for debug use only and may be removed in a future release.*

XAUI Direct block for BEE4

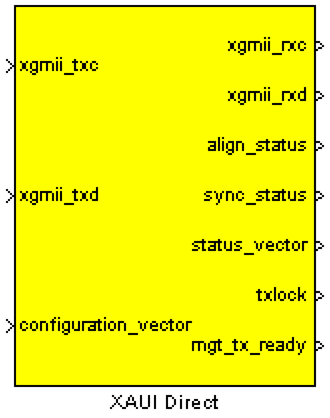
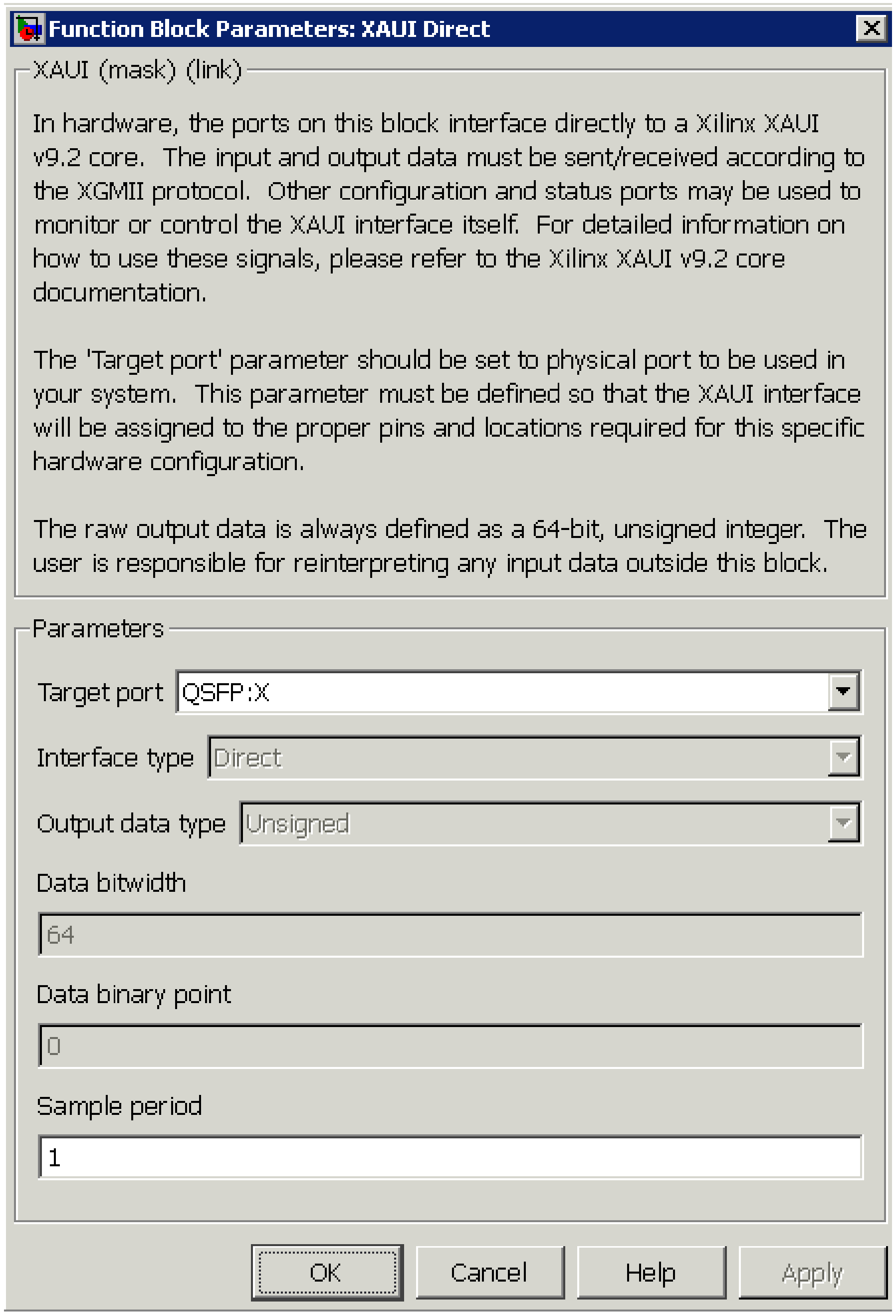


Figure 157: XAUI Direct block for BEE4

The XAUI Direct block provides an interface which is identical to that of the Xilinx XAUI v9.2 component in Core Generator. The data inputs and outputs adhere to the standard XGMII protocol. For more information on the usage of the auxiliary control and status ports, please refer to the XAUI core documentation in Core Generator.

Use of this block requires that the system clock be sourced by the 156.25MHz output clock from the FMC board, as the XGMII interface must run synchronously to the XAUI core.



Function Block Parameters: XAUI Direct

XAUI (mask) (link)

In hardware, the ports on this block interface directly to a Xilinx XAUI v9.2 core. The input and output data must be sent/received according to the XGMII protocol. Other configuration and status ports may be used to monitor or control the XAUI interface itself. For detailed information on how to use these signals, please refer to the Xilinx XAUI v9.2 core documentation.

The 'Target port' parameter should be set to physical port to be used in your system. This parameter must be defined so that the XAUI interface will be assigned to the proper pins and locations required for this specific hardware configuration.

The raw output data is always defined as a 64-bit, unsigned integer. The user is responsible for reinterpreting any input data outside this block.

Parameters

Target port: QSFP:X

Interface type: Direct

Output data type: Unsigned

Data bitwidth: 64

Data binary point: 0

Sample period: 1

Figure 158: XAUI Direct dialog for BEE4

Table 127: XAUI Direct block parameters for BEE4

Parameter	**Description**
Data type	This parameter determines the inferred arithmetic type for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to *Unsigned*.
Data bitwidth	This value defines the total width of the signal in bits. The standard data rate (SDR) version of XGMII requires that this value be fixed to 64 bits.
Data binary point	This value determines the inferred binary point position for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to zero.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 128: XAUI Direct port definitions for BEE4

Port	**Direction**	**Bit Width**	**Description**
`xgmii_txc` `xgmii_txd`	in	8 64	XGMII transmit control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`configuration_vector`	in	7	Vector of configuration bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.
`xgmii_rxc` `xgmii_rxd`	out	8 64	XGMII receive control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`align_status` `sync_status`	out	1 4	Vectors indicating high-speed serial lane alignment and synchronization status. Alignment status is a single bit indicating that all 4 XAUI lanes are aligned, and synchronization status is a 4-bit value with the individual link status of each lane.
`status_vector`	out	8	Vector of status bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.
`txlock`	out	1	Status bit indicating that the GTX transmitter clock is locked. *This signal is intended for debug use only and may be removed in a future release.*

`mgt_tx_ready`	out	1	Status bit indicating that the GTX transmitter interface is ready to accept outgoing traffic. *This signal is intended for debug use only and may be removed in a future release.*

XAUI FIFO block for BEE4

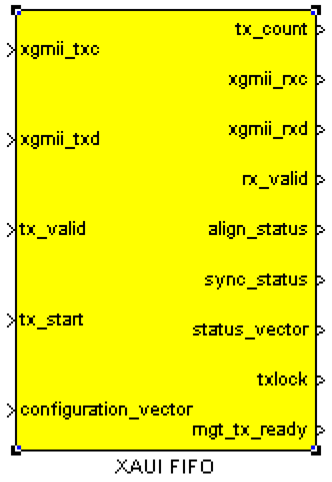
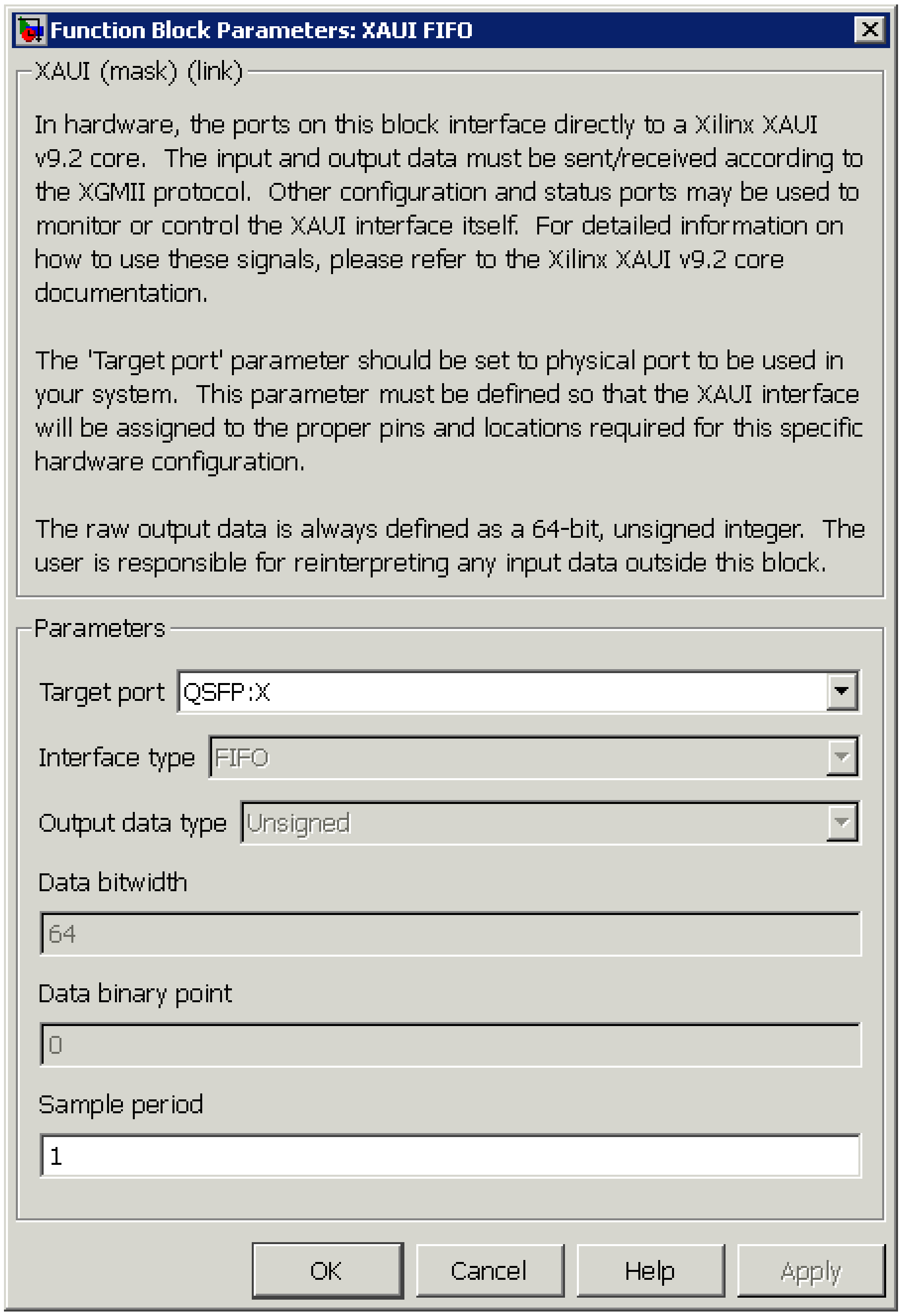


Figure 159: XAUI FIFO block for BEE4

The XAUI FIFO block provides an interface which is nearly identical to that of the Xilinx XAUI v9.2 component in Core Generator. The data inputs and outputs adhere to the standard XGMII protocol, but data is buffered in FIFOs before being sent out to the XAUI PHY. For more information on the usage of the auxiliary control and status ports, please refer to the XAUI core documentation in Core Generator.

The asynchronous FIFOs in this block enable the system clock to run at a rate independent of the 156.25MHz clock used by the XAUI PHY. On the transmit side, XGMII control and data are buffered when `tx_valid` is asserted. The XAUI PHY does not consume the buffered contents until `tx_start` is asserted. When `tx_start` is asserted, the XAUI PHY begins continuously consuming the contents of the buffer until it has emptied. On the receive side, the FIFO is feed-forward and indicates when valid XGMII data and control is applied to the `xgmii_rxd` and `xgmii_rxc` pins by asserting `rx_valid`.


Function Block Parameters: XAUI FIFO

XAUI (mask) (link)

In hardware, the ports on this block interface directly to a Xilinx XAUI v9.2 core. The input and output data must be sent/received according to the XGMII protocol. Other configuration and status ports may be used to monitor or control the XAUI interface itself. For detailed information on how to use these signals, please refer to the Xilinx XAUI v9.2 core documentation.

The 'Target port' parameter should be set to physical port to be used in your system. This parameter must be defined so that the XAUI interface will be assigned to the proper pins and locations required for this specific hardware configuration.

The raw output data is always defined as a 64-bit, unsigned integer. The user is responsible for reinterpreting any input data outside this block.

Parameters

Target port: QSFP:X

Interface type: FIFO

Output data type: Unsigned

Data bitwidth: 64

Data binary point: 0

Sample period: 1

Figure 160: XAUI FIFO dialog for BEE4

Table 129: XAUI FIFO block parameters for BEE4

Parameter	Description
Data type	This parameter determines the inferred arithmetic type for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to *Unsigned*.
Data bitwidth	This value defines the total width of the signal in bits. The standard data rate (SDR) version of XGMII requires that this value be fixed to 64 bits.
Data binary point	This value determines the inferred binary point position for the signal. The data and control ports for this block have no implicit numerical value, therefore this is fixed to zero.
Sample period	The Simulink sample period to be declared for this block. This value must match the fundamental sample period defined for the System Generator design.

Table 130: XAUI FIFO port definitions for BEE4

Port	Direction	Bit Width	Description
`xgmii_txc` `xgmii_txd`	in	8 64	XGMII transmit control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`tx_start`	in	1	Control port used to indicate FIFO buffer is ready for transmission. Pulse or assert high to begin transmitting the entire contents of the FIFO.
`tx_valid`	in	1	Assert high whenever `xgmii_txc` and `xgmii_txd` are valid and ready to be added to the FIFO.
`configuration_vector`	in	7	Vector of configuration bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.
`tx_count`	out	10	FIFO status port indicating approximate count of transmit control and data words in the FIFO.
`xgmii_rxc` `xgmii_rxd`	out	8 64	XGMII receive control and data ports. Please refer to the Xilinx XAUI v9.2 and XGMII standard documentation for protocol information.
`rx_valid`	out	1	Indicates valid data is on the `xgmii_rxc` and `xgmii_rxd` ports.

`align_status` `sync_status`	out	1 4	Vectors indicating high-speed serial lane alignment and synchronization status. Alignment status is a single bit indicating that all 4 XAUI lanes are aligned, and synchronization status is a 4-bit value with the individual link status of each lane.
`status_vector`	out	8	Vector of status bits for the XAUI core. Please refer to the Xilinx XAUI v9.2 documentation for signal information.
`txlock`	out	1	Status bit indicating that the GTX transmitter clock is locked. *This signal is intended for debug use only and may be removed in a future release.*
`mgt_tx_ready`	out	1	Status bit indicating that the GTX transmitter interface is ready to accept outgoing traffic. *This signal is intended for debug use only and may be removed in a future release.*

XAUI block NectarOS functions

The XAUI block includes a set of utility functions that enable the user to control the XAUI block directly from a NectarOS shell. BPS adds the following functions to NectarOS whenever the XAUI block is used in a design. The source code for these functions are contained in the *drivers/xps_xaui* subdirectory of the BPS-generated EDK project directory.

Initialization function calls

`mdio_init`

The `mdio_init` function performs all initialization steps required to set up the physical MDIO interface to the PHY devices, and to configure the PHY devices for 10-gigabit optical Ethernet communication.

Repeated function calls

(none)

Commands

`mdio_phy_init`

The `mdio_phy_init` function will re-perform a complete initialization of both PHY devices.

`mdio_phy_reset`

The `mdio_phy_reset` function will assert a reset of the MDIO logic and the physical reset pin of the PHY devices.

`mdio_phy_read <reg_addr>`

The `mdio_phy_read` function will read the contents of the PHY register `reg_addr` on both PHY devices and print the result on the console.

`mdio_phy_write <reg_addr> <value>`

The `mdio_phy_write` function will write `value` to the PHY register `reg_addr` on both PHY devices.

Chapter 6: Host-side Programming

In addition to generating FPGA implementations, BPS offers a variety of features which are accessible from the host workstation environment. These include a library of Matlab functions provided as part of your BPS distribution, as well as a standard remote access protocol via Ethernet for transferring data to and from memory resources on the FPGA itself.

Matlab Function Reference

Several utility functions are included in the library of Matlab routines provided with BPS which can be called by the user. This section describes each of these functions and how they should be called.

Hardware access commands

The first set of utility functions are included with BPS to assist with accessing resources on a running hardware design.

`b4d_query_address`

The `b4d_query_address` function will look up the base address for a given named device on a FPGA. The *b4d* service must be running on the Embedded Control Host.

Usage: `address = b4d_query_address(host, fpga, name)`

- `address` – The base address of the requested BPS component on the main processor bus. If the component is not found in the list of known devices, the Matlab function will result in an error.
- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `name` – The name of the BPS component to look up, provided as a character string. This should be the “flat” name of the component (the name of the BPS block itself) and should not include any Simulink path information.

`b4d_reg_read`

The `b4d_reg_read` function will read the contents of a Software Register component on a BEEx FPGA. The *b4d* service must be running on the Embedded Control Host.

Usage: `val = b4d_reg_read(host, fpga, block)`

- `val` – The value read from the Software Register.
- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the “flat” name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.

`b4d_reg_write`

The `b4d_reg_write` function will write the given value into a Software Register component on a BEEx FPGA. The *b4d* service must be running on the Embedded Control Host.

Usage: `b4d_reg_write(host, fpga, block, val)`

- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `val` – The value to be written to the Software Register.

`b4d_bram_read`

The `b4d_bram_read` function will read the given number of words from a Shared BRAM component on a BEEx FPGA. The *b4d* service must be running on the Embedded Control Host.

Usage: `data = b4d_bram_read(host, fpga, block, words)`

- `data` – The data read from the Shared BRAM. The length of the array will be equal to the number of words requested.
- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `words` – The number of words to be read from the remote Shared BRAM. Boundary checking is not performed on this value, so care should be taken to not exceed the size of the BRAM.

`b4d_bram_write`

The `b4d_bram_write` function will write the given data into a Shared BRAM component on a BEEx FPGA. The *b4d* service must be running on the Embedded Control Host.

Usage: `b4d_bram_write(host, fpga, block, data)`

- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path

information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `data` – The data to be written to the Shared BRAM. Each value will be interpreted as an unsigned 32-bit integer. Boundary checking is not performed on the length of this array, so care should be taken not to exceed the size of the BRAM.

b4d_fifo_read

The `b4d_fifo_read` function will read the given number of words from a Shared FIFO component on a BEEx FPGA. This command will wait until the requested number of words are successfully read from the FIFO before completing, so care should be taken to not request more data than the hardware can provide, or else deadlock conditions can occur. The *b4d* service must be running on the Embedded Control Host.

Usage: `data = b4d_fifo_read(host, fpga, block, words)`

- `data` – The data read from the Shared FIFO. The length of the array will be equal to the number of words requested.
- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the “flat” name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `words` – The number of words to be read from the Shared FIFO. Since the FPGA will continue polling the FIFO until this number of words have been read, care should be taken to not request more data than the hardware can provide to prevent deadlock conditions.

b4d_fifo_write

The `b4d_fifo_write` function will write the given data into a Shared FIFO component on a BEEx FPGA. This command will wait until the requested number of words are successfully written to the FIFO before completing, so care should be taken to not send more data than the hardware can accept, or else deadlock conditions can occur. The *b4d* service must be running on the Embedded Control Host.

Usage: `b4d_fifo_write(host, fpga, block, data)`

- `host` – The host name or IP address of the BEEx system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor bus. If the argument is a character string, it is interpreted as the “flat” name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `data` – The data to be written to the Shared FIFO. Each value will be interpreted as an unsigned 32-bit integer. Since the FPGA will continue polling the FIFO until this number

of words have been written, care should be taken to not send more data than the hardware can accept to prevent deadlock conditions.

`b4d_dram_read`

The `b4d_dram_read` function will read the given number of 32-bit words from DRAM on a BEE7 FPGA. The *b4d* service must be running.

Usage: `data = b4d_dram_read(host, fpga, channel, address, words)`

- `data` – The data read from DRAM. The length of the array will be equal to the number of bytes requested.
- `host` – The host name or IP address of the host running b4d and connected to the BEE7 system, provided as a character string.
- `channel` – The ID for the target memory channel.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `address` – The address to begin reading from DRAM. Note that this a physical (zero-based) address in DRAM.
- `bytes` – The number of bytes to be read from the remote DRAM.

`b4d_dram_write`

The `b4d_dram_write` function will write the given 32-bit word data into DRAM on a BEE7 FPGA. The *b4d* service must be running.

Usage: `b4d_dram_write(host, fpga, channel, address, data)`

- `host` – The host name or IP address of the host running b4d and connected to the BEE7 system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `channel` – The ID for the target memory channel.
- `address` – The address to begin writing to DRAM. Note that this is a physical (zero-based) address in DRAM.
- `data` – The data to be written to DRAM. Each value will be interpreted as an unsigned 32-bit integer.

`b4d_send_command`

The `b4d_send_command` functions sends a command to nectarOS on the specified FPGA. The command does not return back the print out on the terminal.

Usage: `b4d_send_command (host, fpga, command)`

- `host` – The host name or IP address of the host running b4d and connected to the BEE7 system, provided as a character string.
- `fpga` – The ID of the desired FPGA for this operation (`A`, `B`, `C`, or `D`).
- `command` – The string command to send to the FPGA’s nectarOS

`eth_query_address`

The `eth_query_address` function will look up the base address for a given named device on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `address = eth_query_address(host, name)`

- `address` – The base address of the requested BPS component on the main processor PLB bus. If the component is not found in the list of known devices, the Matlab function will result in an error.
- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `name` – The name of the BPS component to look up, provided as a character string. This should be the “flat” name of the component (the name of the BPS block itself) and should not include any Simulink path information.

`eth_reg_read`

The `eth_reg_read` function will read the contents of a Software Register component on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `val = eth_reg_read(host, block)`

- `val` – The value read from the remote Software Register.
- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the “flat” name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.

`eth_reg_write`

The `eth_reg_write` function will write the given value into a Software Register component on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `eth_reg_write(host, block, val)`

- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the “flat” name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `val` – The value to be written to the remote Software Register.

`eth_bram_read`

The `eth_bram_read` function will read the given number of words from a Shared BRAM component on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `data = eth_bram_read(host, block, words)`

- `data` – The data read from the remote Shared BRAM. The length of the array will be equal to the number of words requested.
- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `words` – The number of words to be read from the remote Shared BRAM. Boundary checking is not performed on this value, so care should be taken to not exceed the size of the BRAM.

`eth_bram_write`

The `eth_bram_write` function will write the given data into a Shared BRAM component on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `eth_bram_write(host, block, data)`

- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `data` – The data to be written to the remote Shared BRAM. Each value will be interpreted as an unsigned 32-bit integer. Boundary checking is not performed on the length of this array, so care should be taken not to exceed the size of the BRAM.

`eth_fifo_read`

The `eth_fifo_read` function will read the given number of words from a Shared FIFO component on a remote FPGA via the embedded Ethernet data transfer protocol. This command will wait until the requested number of words are successfully read from the FIFO before completing, so care should be taken to not request more data than the hardware can provide, or else deadlock conditions can occur. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `data = eth_fifo_read(host, block, words)`

- `data` – The data read from the remote Shared FIFO. The length of the array will be equal to the number of words requested.
- `host` – The host name or IP address of the remote FPGA, provided as a character string.

- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `words` – The number of words to be read from the remote Shared FIFO. Since the FPGA will continue polling the FIFO until this number of words have been read, care should be taken to not request more data than the hardware can provide to prevent deadlock conditions.

eth_fifo_write

The `eth_fifo_write` function will write the given data into a Shared FIFO component on a remote FPGA via the embedded Ethernet data transfer protocol. This command will wait until the requested number of words are successfully written to the FIFO before completing, so care should be taken to not send more data than the hardware can accept, or else deadlock conditions can occur. Note that the target FPGA must have been built with Ethernet support included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `eth_fifo_write(host, block, data)`

- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `block` – The block name or base address of the component to be accessed. If the argument is numeric, it is interpreted as the base address of the core on the main processor PLB bus. If the argument is a character string, it is interpreted as the "flat" name of the component (the name of the BPS block itself in Simulink, without any path information), and the base address will first be looked up via `eth_query_address` before performing the data operation.
- `data` – The data to be written to the remote Shared FIFO. Each value will be interpreted as an unsigned 32-bit integer. Since the FPGA will continue polling the FIFO until this number of words have been written, care should be taken to not send more data than the hardware can accept to prevent deadlock conditions.

eth_dram_read

The `eth_dram_read` function will read the given number of bytes from DRAM on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support and a processor-accessible Memory Controller included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `data = eth_dram_read(host, address, bytes)`

- `data` – The data read from the remote DRAM. The length of the array will be equal to the number of bytes requested.
- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `address` – The base address on the main processor PLB bus to begin reading from DRAM. Note that this is a processor/PLB address, not a physical (zero-based) address in DRAM.

- `bytes` – The number of bytes to be read from the remote DRAM.

`eth_dram_write`

The `eth_dram_write` function will write the given byte data into DRAM on a remote FPGA via the embedded Ethernet data transfer protocol. Note that the target FPGA must have been built with Ethernet support and a processor-accessible Memory Controller included, and must have already obtained a valid IP address from a DHCP server before it will be accessible by this function.

Usage: `eth_dram_write(host, address, data)`

- `host` – The host name or IP address of the remote FPGA, provided as a character string.
- `address` – The base address on the main processor PLB bus to begin writing to DRAM. Note that this is a processor/PLB address, not a physical (zero-based) address in DRAM.
- `data` – The data to be written to the remote DRAM. Each value will be interpreted as an unsigned 8-bit integer.

`serial_read`

The `serial_read` function uses the binary transaction special mode built into NectarOS to read raw data directly from a BPS core on the running FPGA. This function requires that the PC running Matlab be connected to the hardware platform via a serial port.

Usage: `data = serial_read(core_name, serial_port, dram_base, dram_size)`

- `data` – The data values read from the FPGA, returned as a `uint32` array.
- `core_name` – The name of the BPS core to be read. This can be the name of any Software Register block, Shared BRAM block, or the string `‘MPMC’` to access DRAM.
- `serial_port` – The name of the serial port to be used for communication, such as `‘COM3’`.
- `dram_base` – The base address from which the transaction should begin. This argument should only be specified for the `‘MPMC’` device. All other devices automatically read the entire contents of the core.
- `dram_size` – The size of the read transaction. This argument should only be specified for the `‘MPMC’` device. All other devices automatically read the entire contents of the core.

`serial_write`

The `serial_write` function uses the binary transaction special mode built into NectarOS to write raw data directly to a BPS core on the running FPGA. This function requires that the PC running Matlab be connected to the hardware platform via a serial port.

Usage: `serial_write(data, core_name, serial_port, dram_base)`

- `data` – The data values to be written to the FPGA, provided as a `uint32` array.
- `core_name` – The name of the BPS core to be written. This can be the name of any Software Register block, Shared BRAM block, or the string `‘MPMC’` to access DRAM.
- `serial_port` – The name of the serial port to be used for communication, such as `‘COM3’`.

- `dram_base` – The base address from which the transaction should begin. This argument should only be specified for the `'MPMC'` device. All other devices automatically read the entire contents of the core.

Miscellaneous commands

The second set of utility functions are included to inspect the state and configuration of the system running BPS.

`bps_check_license`

This function will report the BPS license status on the current machine. A report will be printed to the Matlab console which indicates whether a valid BPS license file was found, and if so, which features are available and when the licenses will expire.

`uint8castfix`

The `uint8castfix` function interprets an 8-bit two's complement value as a fixed-point number and returns the result as a standard Matlab double precision floating point number. This function is useful for reinterpreting data stored in hardware memory into its intended precision.

Usage: `result = uint8castfix(data, signed, bin_pt)`

- `result` – A double precision floating point number with the reinterpreted value of the input data.
- `data` – The input data to be reinterpreted, which should be an 8-bit two's complement value.
- `signed` – Determines whether the input data will be interpreted as a signed number. Any logical true value will cause signed conversion, while logical false will cause unsigned conversion.
- `bin_pt` – The implied binary point position to use for the fixed-point scaling operation. This value must be an integer between 0 and 8.

`uint16castfix`

The `uint16castfix` function interprets a 16-bit two's complement value as a fixed-point number and returns the result as a standard Matlab double precision floating point number. This function is useful for reinterpreting data stored in hardware memory into its intended precision.

Usage: `result = uint16castfix(data, signed, bin_pt)`

- `result` – A double precision floating point number with the reinterpreted value of the input data.
- `data` – The input data to be reinterpreted, which should be a 16-bit two's complement value.
- `signed` – Determines whether the input data will be interpreted as a signed number. Any logical true value will cause signed conversion, while logical false will cause unsigned conversion.
- `bin_pt` – The implied binary point position to use for the fixed-point scaling operation. This value must be an integer between 0 and 16.

`uint32castfix`

The `uint32castfix` function interprets a 32-bit two's complement value as a fixed-point number and returns the result as a standard Matlab double precision floating point number. This function is useful for reinterpreting data stored in hardware memory into its intended precision.

Usage: `result = uint32castfix(data, signed, bin_pt)`

- `result` – A double precision floating point number with the reinterpreted value of the input data.
- `data` – The input data to be reinterpreted, which should be a 32-bit two's complement value.
- `signed` – Determines whether the input data will be interpreted as a signed number. Any logical true value will cause signed conversion, while logical false will cause unsigned conversion.
- `bin_pt` – The implied binary point position to use for the fixed-point scaling operation. This value must be an integer between 0 and 32.

Ethernet Data Transfer Protocol

When the Ethernet Interface block is included in an FPGA design, BPS will generate a TCP data transfer service into the software application which is automatically launched upon system initialization. This service provides a set of transactions which can read or write data arbitrarily from any BPS memory components in the system. This service provides an alternative for accessing on-FPGA data other than typing commands on the embedded shell or using the serial port's binary transaction mode. The Ethernet interface also provides much higher data throughput than the serial interface, and provides greater scalability by allowing one remote host to simultaneously communicate with multiple target hardware devices over the network.

Specification

Every data transfer session must begin with a single transaction header, which is composed of three 32-bit words in the following format. Note that the header itself cannot be fragmented across more than one packet.

Table 131: Ethernet data transaction header format

Operation code	*Base address*	*Length*
32 bits	32 bits	32 bits

The following table lists each of the different transaction types, and how each field is interpreted for the corresponding operation.

Table 132: Supported Ethernet data transactions

Transaction	*Opcode*	*Address*	*Length (N)*	*Bytes expected*	*Bytes returned*
DRAM write	0x00	Physical address from which writes should start	Number of bytes to write	*N*	4
DRAM read	0x01	Physical address from which reads should start	Number of bytes to read	-	*N*+4
BRAM write	0x02	Physical address of BRAM core	Number of 32-bit words to write	*N**4	4
BRAM read	0x03	Physical address of BRAM core	Number of 32-bit words to read	-	*N**4+4
FIFO write	0x04	Physical address of FIFO core	Number of 32-bit words to write	*N**4	4
FIFO read	0x05	Physical address of FIFO core	Number of 32-bit words to read	-	*N**4+4
Register write	0x06	Physical address of register core	1	4	4
Register read	0x07	Physical address of register core	1	-	8

Address query	0x100	Reserved (set to 0)	Length of core name string, including null	Null-terminated core name	8

After the completion of every transaction, the FPGA will return one 32-bit word containing the result of the operation. Therefore, the host must always receive 4 additional bytes after each transaction to remain in sync with the service. These 4 status bytes are reflected in the table above. The flags which can be set in the status word are described below.

Table 133: Ethernet data transaction result status flags

Status flag	*Value*	*Description*
STAT_SUCCESS	0x00	Operation completed successfully
STAT_INVAL_CMD	0x01	Operation code did not match any known commands
STAT_INVAL_ADDR	0x02	Address was out of range or included protected segments
STAT_INVAL_LEN	0x04	Length parameter was zero or invalid for specified operation
STAT_INVAL_HDR	0x08	Incomplete header received
STAT_INT_ERROR	0x80	An internal error occurred within the FPGA

At the conclusion of a transaction, whether successful or not, the host must close the TCP session. The embedded service will only service one transaction per session, and will then close the current connection and wait for a new incoming connection before receiving another transaction header.

There are several caveats that should be observed when using the data transaction service:

- Only one connection can be opened to the FPGA at any given time. If additional new connections are attempted, these will be refused by the service. The current implementation of the service will instantly abort the connection (listen backlogs are not implemented).
- 32-bit words sent by the FPGA (such as core addresses and the return status value) will have *big-endian* byte ordering, which is a result of both the embedded processor and the underlying hardware model. This is equivalent to “network byte order”, and therefore standard byte conversion routines can be used in the host application for cross-platform compatibility.
- Protected memory segments (i.e. processor reset vectors and application instruction/data memory) are automatically avoided by the service, but no other address validation is performed. The user is responsible for providing the correct core base addresses for each transaction. The query command can be used for this purpose, and only must be called once by the host application, as addresses cannot change at runtime for a single FPGA design.
- All FIFO transactions are *blocking* on the FPGA, meaning the embedded processor will wait as long as necessary until data is available in the read FIFO for read transactions, and until free space is available in the write FIFO for write transactions. It is expected that the hardware portion of the FPGA design will be periodically filling and emptying data from the FIFO, as there is no enforced time limit for a single transaction. Note that polling of the FIFO interface is not interrupt-driven, and will consume 100% of the

embedded processor (and consequently the shell) while blocking. The user is responsible for throttling the size of their data transactions appropriately.

- In the event that the string passed as part of a query transaction does not match any known cores, an address value of 0xFFFFFFFF will be returned. This is independent of the status result itself, which will still reflect the overall status of the transaction and will normally indicate success.

Examples

The table below shows the sequence of operations necessary to read and write from BRAM on the FPGA. This example demonstrates the lookup of the named BRAM block's base address, reading the contents of the BRAM, and writing back new values.

Table 134: Sequence of network operations for BRAM read-modify-write

FPGA operation	*Host operation*
	Connect to TCP service
TCP connection accepted	
	Send transaction header [0x00000100, 0x00000000, 6] (12 bytes)
Receive transaction header (12 bytes)	
	Send query core name string [`'bram0\0'`] (6 bytes)
Receive query core name string (6 bytes)	
Send core address value [0x10000000] (4 bytes)	
	Receive and byte-swap core address value (4 bytes)
Send return status value [0x00000000] (4 bytes)	
	Receive and byte-swap return status value (4 bytes)
Close connection	Close connection
	Connect to TCP service
TCP connection accepted	
	Send transaction header [0x00000003, 0x10000000, 2048] (12 bytes)
Receive transaction header (12 bytes)	
Send 2048 32-bit words from 0x10000000 (8192 bytes)	

Send return status value [0x00000000] (4 bytes)	
	Receive 2048 32-bit words (8192 bytes)
	Receive and byte-swap return status value (4 bytes)
Close connection	Close connection
	Connect to TCP service
TCP connection accepted	
	Send transaction header [0x00000002, 0x10000000, 2048] (12 bytes)
Receive transaction header (12 bytes)	
	Send 2048 32-bit words (8192 bytes)
Receive 2048 32-bit words into 0x10000000 (8192 bytes)	
Send return status value [0x00000000] (4 bytes)	
	Receive and byte-swap return status value (4 bytes)
Close connection	Close connection

A complete FPGA implementation model, as well as host-side source code which demonstrates each data transaction type, is included with your BPS distribution. FPGA implementation models are included in the *examples* directory, and host source intended for compilation under Linux is included in the *examples/src/linux/ethernet_lwip* directory.

Appendix

Adding Custom Software

BPS also provides a mechanism for you to add your own commands, initialization code, or polling routines into NectarOS. By placing your source code into the `src/` subdirectory within a BPS project directory and running the Software Generation build phase, BPS will integrate your custom code into the shell, complete with help text and parameter information for command functions.

BPS looks for specially formatted function prototypes and comments to identify which functions require integration into NectarOS. This section describes the code formatting and runtime behavior of each custom routine.

For examples of how to write the prototypes, refer to the C files in `examples/src/fpga/nectaros` (from the root of your BPS installation).

NectarOS commands

BPS can automatically add custom commands to NectarOS by using the following code format for each function that is a command entry point:

```
void custom_cmd(int argc, char **argv)
/* command = “custom” */
/* help = “Custom command” */
/* params = “<arg1> <arg2>” */
{ ...
```

The code segment above will create a new command named `custom` in NectarOS. The string `“Custom command”` will also be printed as the command description and `“<arg1> <arg2>”` as the argument list in all NectarOS help output. The first four lines of the code segment must be written exactly as they appear above for BPS to properly identify the command definition, although the function name and each of the descriptive strings in quotes can naturally be changed to match your command.

Initialization routines

BPS can also add custom function calls which perform initialization operations for the design by using the following code format:

```
void custom_init()
/* init */
{ ...
```

The code segment above will cause BPS to insert a call to the function `custom_init` after shell initialization, but before the first prompt string is displayed. The name of the function can be changed, but the function prototype and comment string must be written exactly as they appear above.

Polling routines

Finally, BPS can add custom function calls which are called repeatedly by using the following code format:

```
void custom_repeat()
/* repeat */
{ ...
```

The code segment above will cause BPS to insert a call to the function `custom_repeat` inside the NectarOS character polling loop. This provides a mechanism for repeatedly monitoring or updating resources which are not interrupt-driven without blocking the shell itself. The name of the function can be changed, but the function prototype and comment string must be written exactly as they appear above.

Using the BPS IP Import Utility

BPS comes with an additional tool, the IP Import Utility (IPIU) for importing your own custom hardware components into the BPS environment as a new block. The IPIU can take any NGC or EDIF netlist as an input, and will generate an equivalent Simulink block and EDK pcore, which BPS will then be able to instantiate directly in the generated EDK project. Currently, netlists that instantiate I/O buffers are unsupported.

You can launch the IPIU by typing `ip_import` at the Matlab console, which will bring up the following dialog.

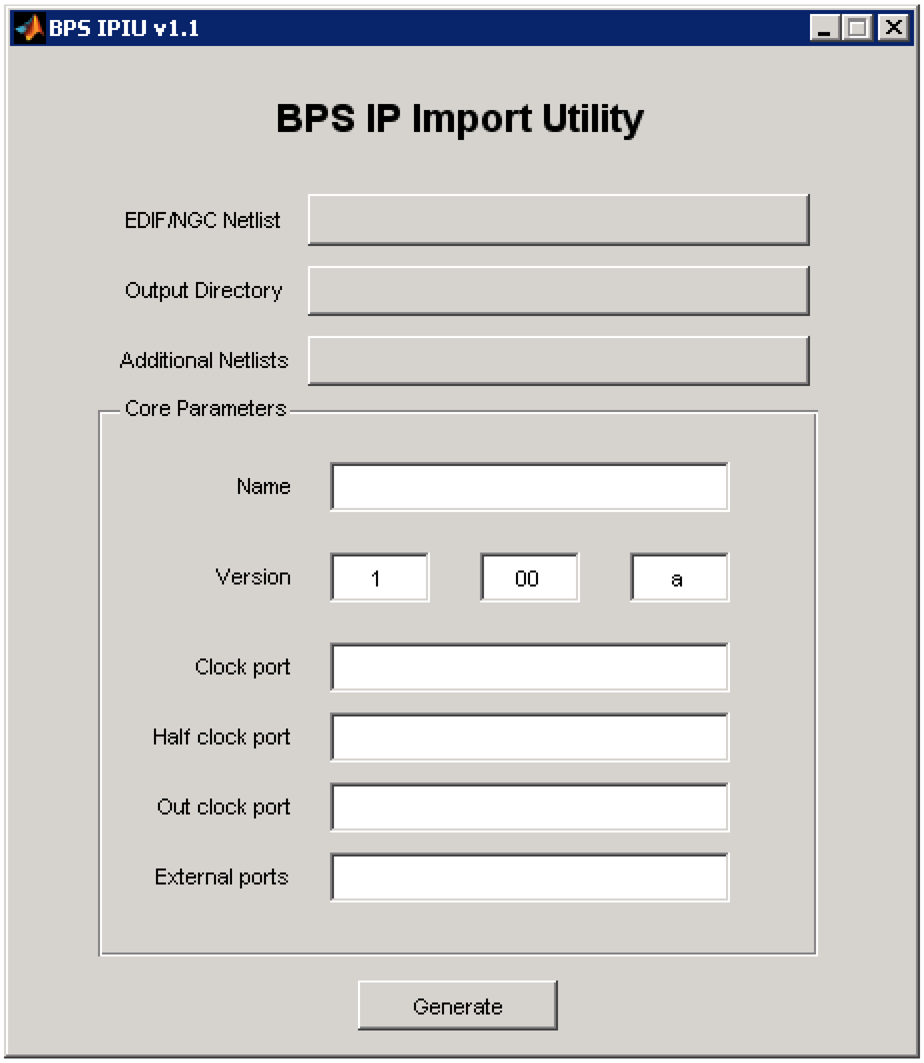


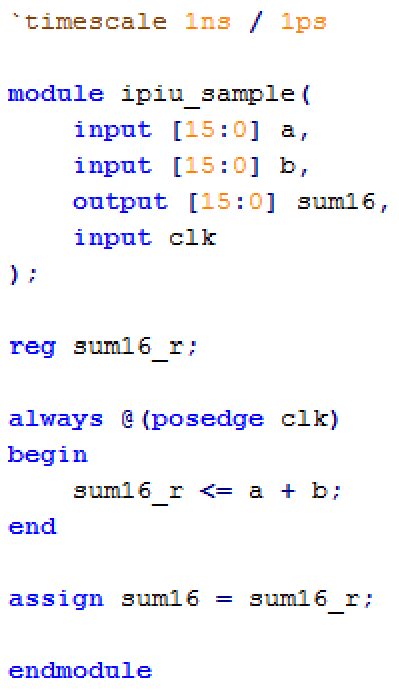
Figure 161: BPS IP Import Utility main dialog

The IPIU main dialog has several fields which must be filled in by the user. Each of these fields is described in the table below.

Table 135: BPS IP Import Utility required fields

Name	*Description*
EDIF/NGC Netlist	The EDIF or NGC netlist which will be imported by the IPIU. This netlist must be the top module of the pcore. The netlist file will be copied into the generated pcore directory and will not be moved or modified by the tool.
Output Directory	The directory into which the IPIU should write all output files, which includes a Simulink library and a complete EDK pcore. If the BPS_USER_IP_IMPORT_PATH environment variable is set, this field will be initialized with that path.
Additional Netlists	Other netlists the module needs to function. The netlists must all be in one directory and selected together in the file dialog. These netlists will be added to the pcore and included during synthesis.
Name	The name of the entity/module defined by the netlist. This must be provided by the user, since the name of the netlist itself may not be identical to the hardware component's logical name. This field will initially be filled in by the IPIU to match the root name of the netlist. Note that the generated EDK pcore will also inherit this name.
Version	The version string to use for the generated EDK pcore.
Clock port	The name of the port which should be inferred as the system clock input in the hardware component. BPS will automatically connect this port to the system clock defined in the Platform Configuration block of the FPGA implementation or the Clock Override block (if used). Note that this field is mandatory, therefore the hardware component must provide one port to be labeled as a clock input.
Half clock port	The name of the port which should be inferred as an input for the half-rate system clock. This port can only be used with clocks from the Platform Configuration block. It cannot be used with Clock Override blocks.
Out clock port	The name of the port which should drive the `imported_clk` net, which may be used as the system clock in the Platform Configuration block.
External ports	A cell array of port names that should be exposed as top-level ports in the BPS-generated EDK project.

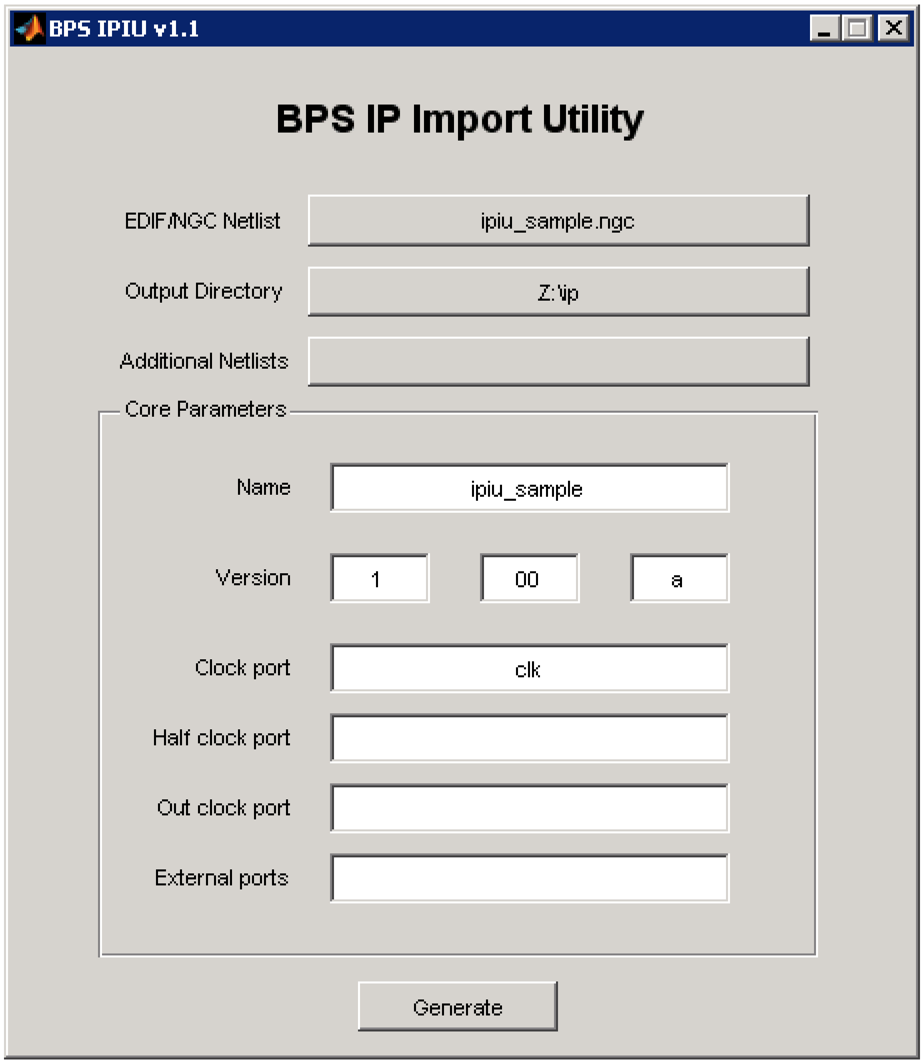
For this example, the following Verilog source was used to generate the ipiu_sample.ngc netlist passed to the IPIU, which models a simple 16-bit adder with one cycle of latency.



```
`timescale 1ns / 1ps

module ipiu_sample(
    input [15:0] a,
    input [15:0] b,
    output [15:0] sum16,
    input clk
);

reg sum16_r;

always @(posedge clk)
begin
    sum16_r <= a + b;
end

assign sum16 = sum16_r;

endmodule
```

Figure 162: BPS IP Import Utility sample Verilog source

Once all the required fields are completed to match this component, the IPIU main dialog box should appear like this:


BPS IPIU v1.1

BPS IP Import Utility

EDIF/NGC Netlist: ipiu_sample.ngc

Output Directory: Z:\ip

Additional Netlists:

Core Parameters

Name: ipiu_sample

Version: 1 00 a

Clock port: clk

Half clock port:

Out clock port:

External ports:

Figure 163: BPS IP Import Utility main dialog (completed)

Click on “Generate” to launch the IP conversion routine and generate the BPS-compatible Simulink and EDK components. This can take up to minutes for very large netlists. Once the conversion is complete, you will see the following output on the Matlab console to describe the steps that have been performed.

```
Generating EDK core 'ipiu_sample_v1_00_a' into directory '.'
 Copied source netlist to './ipiu_sample_v1_00_a/netlist/ipiu_sample.ngc'
 Inferring VHDL simulation netlist from 'netlist/ipiu_sample.ngc'
  Found entity declaration for 'ipiu_sample'
  Found implied clock port 'clk'
  Found 4 total ports in entity 'ipiu_sample'
 VHDL port interface parsed successfully
 Wrote pcore MPD file to './ipiu_sample_v1_00_a/data/ipiu_sample_v2_1_0.mpd'
 Wrote pcore BBD file to './ipiu_sample_v1_00_a/data/ipiu_sample_v2_1_0.bbd'
 Generated Simulink block in library './ipiu_sample.mdl'
IP import complete
```

Figure 164: BPS IP Import Utility console output

In addition to the console output, a new Simulink library window will also appear, which contains the new BPS-compatible Simulink block which may now be used in any BPS design. Note that the ports on the block match the ports of the original HDL source, minus the clock port, which is inferred by BPS and connected "underneath" the Simulink domain.

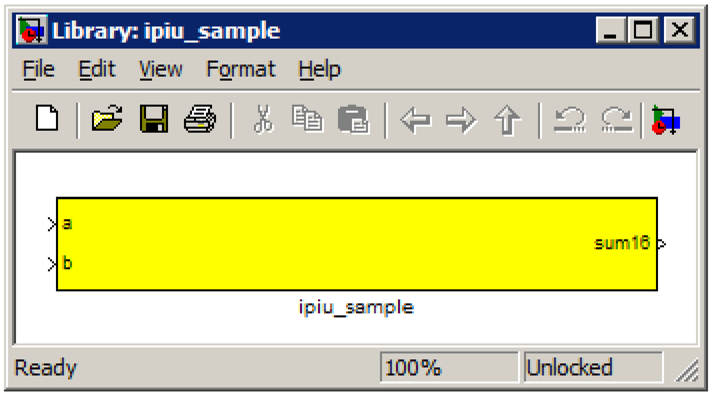
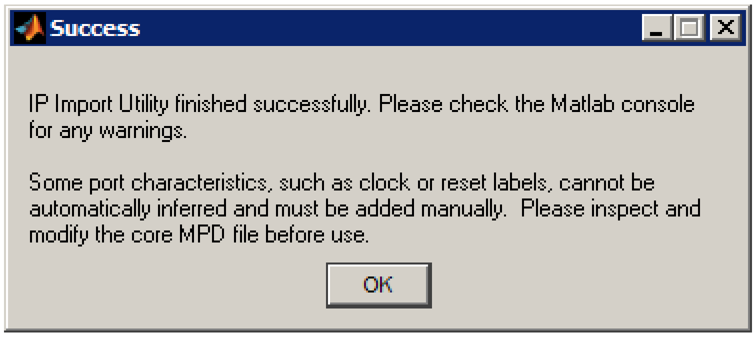


Figure 165: BPS IP Import Utility library block

If no errors were encountered during conversion, one final window will appear on the screen to indicate that the IPIU completed successfully. Otherwise, please follow any instructions given in the error messages and rerun the utility.


Success

IP Import Utility finished successfully. Please check the Matlab console for any warnings.

Some port characteristics, such as clock or reset labels, cannot be automatically inferred and must be added manually. Please inspect and modify the core MPD file before use.

Figure 166: BPS IP Import Utility success dialog

The custom IP import process is now complete. Two items were written into the output directory specified in the main dialog box: a Simulink library (in this example, `ipiu_sample.mdl`) and an EDK pcore (in this example, `ipiu_sample_v1_00_a`). In order to use the component with

BPS, the Simulink library only needs to be in the current working directory or anywhere in the Matlab path. The EDK pcore can be placed in the custom user IP repository location (under the directory `repository` which exists inside the BPS project directory) which is created the first time BPS is run. If the BPS_USER_IP_IMPORT_PATH environment variable is set, the pcore will copied into user IP repository location when Hardware generation is run.

BPS Address Space Allocation

The following address space ranges are pre-defined within BPS. Any custom-made base package or other manual customizations must be careful to respect these address ranges, or conflicts can arise with BPS-generated user IP components.

Table 136: BPS address space allocation

Start address	End address	Description
`0x00000000`	`0x0EFFFFFF`	Available for use by base package components
`0x0F000000`	`0x0F07FFFF`	Reserved for PLB bridge control registers on main PLB bus
`0x0F080000`	`0x0F08FFFF`	Reserved for memory controller ECC control registers
`0x0F090000`	`0x0F09FFFF`	Reserved for memory controller SDMA control registers
`0x0F0A0000`	`0x0F0AFFFF`	Reserved for Ethernet timer
`0x0F0B0000`	`0x0F0BFFFF`	Reserved for Ethernet interrupt controller
`0x0F0C0000`	`0x0F0CFFFF`	Reserved for PCI Express Endpoint bridge control registers
`0x0F0D0000`	`0x0F0DFFFF`	Reserved for central DMA control registers
`0x0F0E0000`	`0x0F0EFFFF`	Reserved for PCI Express Endpoint IPIFBAR
`0x0F100000`	`0x0F17FFFF`	Reserved for Ethernet MAC control registers
`0x10000000`	`0x7FFFFFFF`	Reserved for PLB bridge mapped address space to host all BPS-generated cores
`0x80000000`	`0xFFFFFFFF`	Reserved for DRAM