

2013.11.4

QI151020



Subscribe



Send Feedback

Qsys is a system integration tool included as part of the Quartus® II software. Qsys captures system-level hardware designs at a high level of abstraction and automates the task of defining and integrating customized HDL components. These components include IP cores, verification IP, and other design modules. Qsys facilitates design reuse by packaging and integrating your custom components with Altera® and third-party IP components. Qsys automatically creates interconnect logic from the high-level connectivity you specify, thereby eliminating the error-prone and time-consuming task of writing HDL to specify system-level connections.

Qsys is more powerful if you design your custom components using standard interfaces. By using standard interfaces, your components inter-operate with the components in the Qsys Library. In addition, you can take advantage of bus functional models (BFMs), monitors, and other verification IP to verify your design.

Qsys supports Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), and AMBA APB™ 3 (version 1.0) interface specifications. Qsys does not support AXI4-Lite.

Qsys provides the following advantages when designing a system:

- Automates the process of customizing and integrating components
- Supports up to 64-bit addressing
- Supports modular system design
- Supports visualization of systems
- Supports optimization of interconnect and pipelining within the system
- Fully integrated with the Quartus II software

Related Information

- [Avalon Interface Specifications](#)
- [AMBA Protocol Specifications](#)
- [Creating Qsys Components](#)
- [Qsys Interconnect](#)

Component Interface Support

Components can have any number of interfaces in any combination. Each interface represents a set of signals that you can connect within a Qsys system, or export outside of a Qsys system.

© 2013 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO
9001:2008
Registered



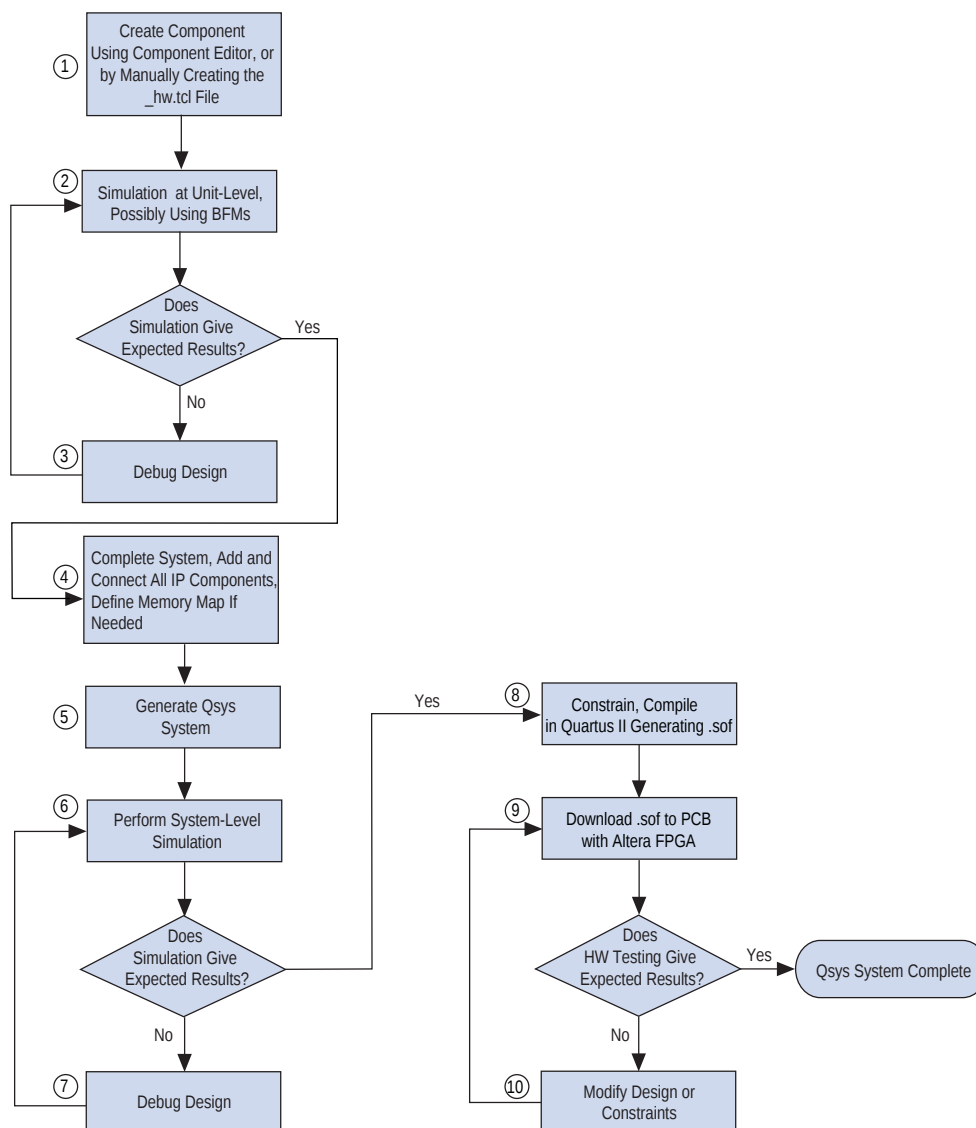
Qsys components can include the following types of interfaces:

- **Memory-Mapped**—Implements a partial crossbar interconnect structure (Avalon-MM, AXI, and APB) that provides concurrent paths between master and slaves. Interconnect consists of synchronous logic and routing resources inside the FPGA, and implementation is based on a network-on-chip architecture.
- **Streaming**—Connects Avalon Streaming (Avalon-ST) sources and sinks that stream unidirectional data, as well as high-bandwidth, low-latency components. Streaming creates datapaths for unidirectional traffic, including multichannel streams, packets, and DSP data. The Avalon-ST interconnect is flexible and can implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet, Interlaken, and video. You can define bus widths, packets, and error conditions.
- **Interrupts**—Connects interrupt senders and the interrupt receivers of the component that serves them. Qsys supports individual, single-bit interrupt requests (IRQs). In the event that multiple senders assert their IRQs simultaneously, the receiver logic (typically under software control) determines which IRQ has highest priority, then responds appropriately.
- **Clocks**—Connects clock output interfaces with clock input interfaces. Clock outputs can fan-out without the use of a bridge. A bridge is required only when a clock from an external (exported) source connects internally to more than one source.
- **Resets**—Connects reset sources with reset input interfaces. If your system requires a particular positive-edge or negative-edge synchronized reset, Qsys inserts a reset controller to create the appropriate reset signal. If you design a system with multiple reset inputs, the reset controller ORs all reset inputs and generates a single reset output.
- **Conduits**—Connects point-to-point conduit interfaces, or represent signals that are exported from the Qsys system. Qsys uses conduits for component I/O signals that are not part of any supported standard interface. You can connect two conduits directly within a Qsys system as a point-to-point connection, or conduit interfaces can be exported and brought to the top-level of the system as top-level system I/O. You can use conduits to connect to external devices, for example external DDR SDRAM memory, and to FPGA logic defined outside of the Qsys system.

Understanding the Qsys Design Flow

Figure 6-1 illustrates a Qsys design flow in which you create a custom IP component and package your custom HDL as a component using the Component Editor or manually creating a `_hw_tcl` file. In this bottom-up design flow, you simulate your custom IP before integrating it with other components as a Qsys system and complete Quartus II project.

Figure 6-1: Qsys Design Flow



In an alternative design flow, you can begin by designing the Qsys system, and then define and instantiate custom Qsys components, clarifying system requirements earlier in the design process.

Related Information

[Creating Qsys Components](#)

Creating a Qsys System

You can create a Qsys system in the Quartus II software by selecting **Qsys System File** in the **New** dialog box, or clicking **Tools > Qsys**. To open a previously created Qsys design, click **Open** on the **File** menu in the Quartus II software window, or the Qsys window.

Related Information[Creating Qsys Components](#)[Component Interface Tcl Reference](#)

Adding and Connecting System Contents

The **System Contents** tab displays the components that you add to your system, and allows you to connect the interfaces of the modules.

Adding Components

To add a component to your system, select the component in the **Library**, and then click **Add**.

When you select a component type and click **Add**, the new instance is added to your system, and a parameter editor opens that allows you to customize the new instance. The new instance appears in the **System Contents** tab, as well as the **Hierarchy** tab

You can type some or all of the component's name in the **Library** search box to help locate a particular component type. For example, you can type `memory` to locate memory-mapped components, or `axi` to locate AXI interconnect components.

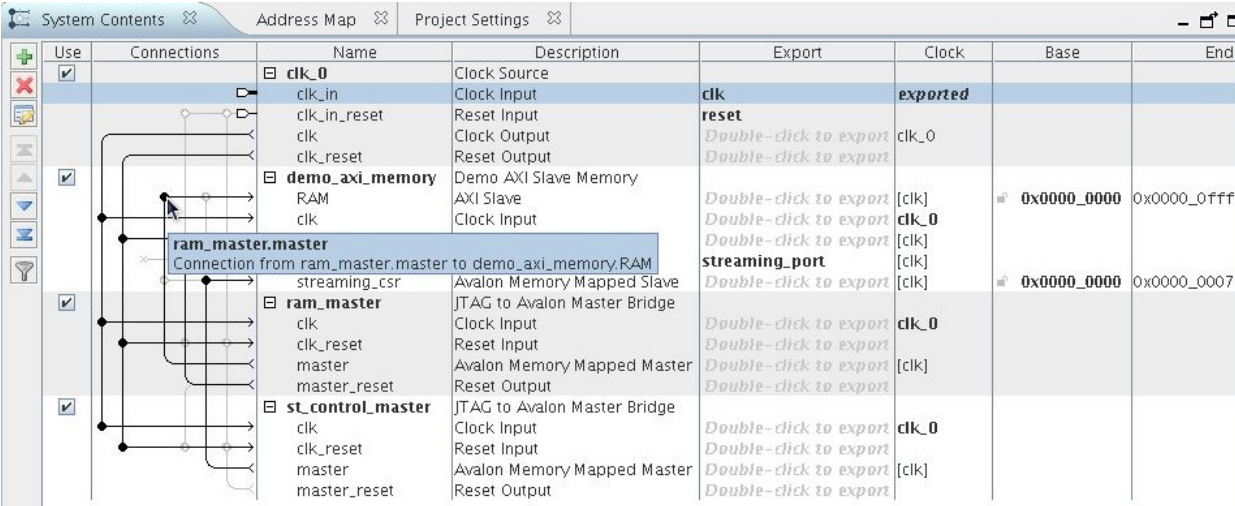
Connecting Components

When you add connections to a Qsys system, you can connect the interfaces of the modules in the **System Contents** tab. The individual signals in each interface are connected by the Qsys interconnect when the HDL for the system generates. You connect interfaces of compatible types and opposite directions. For example, you can connect a memory-mapped master interface to a slave interface, and an interrupt sender interface to an interrupt receiver interface.

Possible connections between interfaces in the system show as gray lines and open circles. When you make a connection, Qsys draws the connection line in black, and fills the connection circle. To make a connection, click the open circle at the intersection of the two interface names. Clicking a filled-in circle removes the connection.

When you are done adding connections in your system, you can deselect **Allow Connection Editing** in the right-click menu, which puts the **Connections** column into read-only mode and hides the possible connections. [Figure 6-2](#) illustrates the **Connections** column.

Figure 6-2: Connections Column in the Systems Contents Tab



Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		clk_0	Clock Source				
		clk_in	Clock Input	clk	exported		
		clk_in_reset	Reset Input	reset			
		clk	Clock Output	<i>Double-click to export</i>	clk_0		
		clk_reset	Reset Output	<i>Double-click to export</i>			
<input checked="" type="checkbox"/>		demo_axi_memory	Demo AXI Slave Memory				
		RAM	AXI Slave	<i>Double-click to export</i>	[clk]	# 0x0000_0000	0x0000_0fff
		clk	Clock Input	<i>Double-click to export</i>	clk_0		
		ram_master.master	Connection from ram_master.master to demo_axi_memory.RAM				
		streaming_csr	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]		
		ram_master	JTAG to Avalon Master Bridge	<i>Double-click to export</i>	[clk]		
		clk	Clock Input	<i>Double-click to export</i>	clk_0		
		clk_reset	Reset Input	<i>Double-click to export</i>			
		master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]		
		master_reset	Reset Output	<i>Double-click to export</i>			
<input checked="" type="checkbox"/>		st_control_master	JTAG to Avalon Master Bridge				
		clk	Clock Input	<i>Double-click to export</i>	clk_0		
		clk_reset	Reset Input	<i>Double-click to export</i>			
		master	Avalon Memory Mapped Master	<i>Double-click to export</i>	[clk]		
		master_reset	Reset Output	<i>Double-click to export</i>			

Related Information

Connecting Components

Filtering Components

You can use the **Filters** dialog box to filter the display of your system in the **System Contents** tab. You can filter the display of your system by interface type, instance name, or by using custom tags. For example, you can view only instances that include memory-mapped interfaces, instances that are connected to a particular Nios II processor, or temporarily hide clock and reset interfaces to simplify the display.

Related Information

Filters Dialog Box

Managing Views

The View menu allows you to select and open any view (tab). Qsys views allow you to review your design from different perspectives. Some views allow you to focus on a particular part of the system, while other views show the same data in another way. Making selections in the system-level views updates other views, and shows the other views in the context of the system-level selection.

For example, selecting `cpu_0` in the **Hierarchy** tab updates the **Parameters** tab to show the parameters for `cpu_0`.

Note: When you double-click a message in the **Messages** tab, Qsys selects the associated element in the relevant view to facilitate debugging.

When you create a new Qsys system, the **Library**, **Hierarchy**, and **System Contents** tabs appear by default. You can arrange your system workspace by dragging and dropping, and then grouping tabs in an order appropriate to your design process. All tabs are dockable and you can close, hide, or minimize tabs that you are not using. Minimized tabs appear minimized in the docking area below the menu bar. Tool tips on tab corners display possible workspace arrangements, for example, disconnecting or restoring a tab to the Qsys workspace.

When you save the Qsys system, the current view arrangement is saved, and when you open the Qsys system, the last saved view arrangement is restored. You can use the **Reset View Layout** command on the View menu to restore the Qsys workspace to its default configuration.

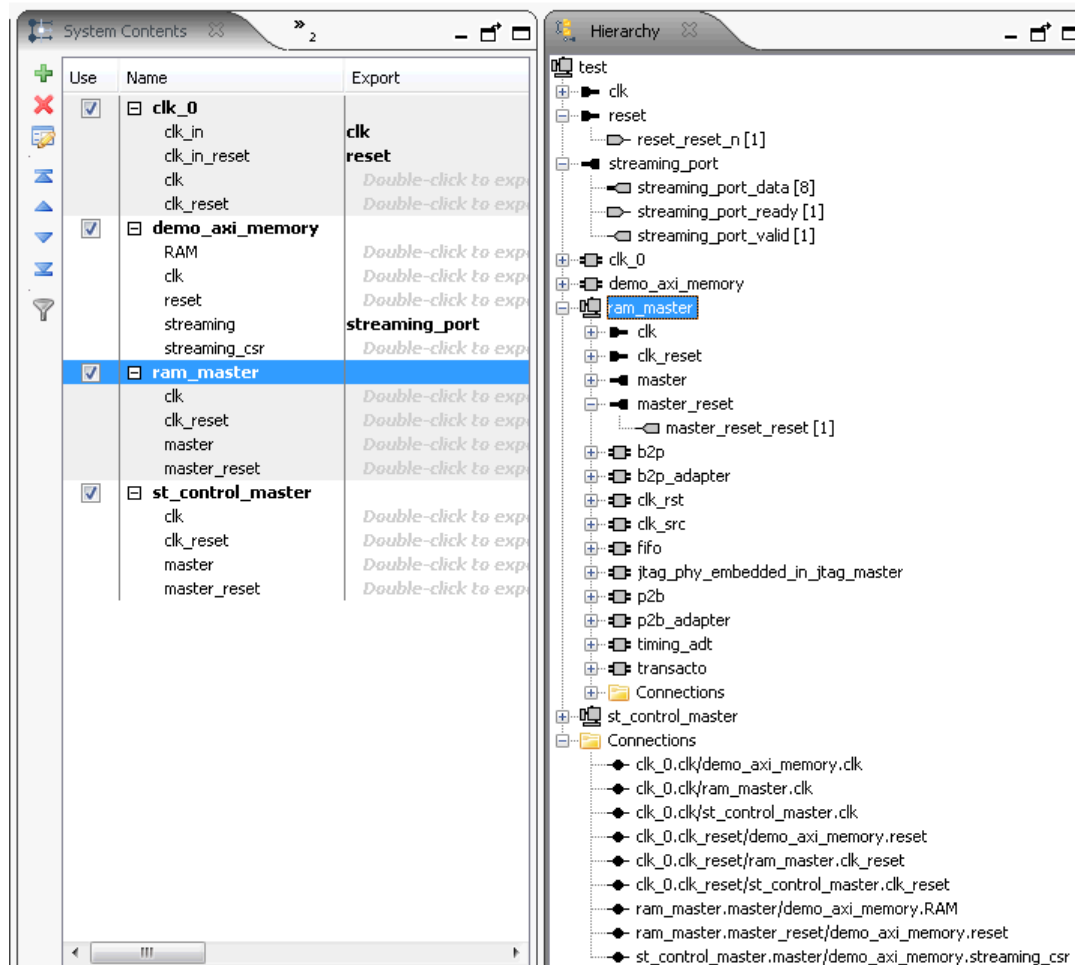
Note: Qsys contains some views which are not documented and appear on the View menu as "Beta". The purpose in presenting these views is to allow designers to explore their usefulness in Qsys system development.

Using the Hierarchy Tab

The **Hierarchy** tab is a full system hierarchical navigator, which expands the system contents to show modules, interfaces, signals, contents of subsystems, and connections.

The graphical interface of the **Hierarchy** tab displays a unique icon for each element represented in the system, including interfaces, directional pins, IP blocks, and system icons that show exported interfaces and the instances of components that make up a system, as shown in [Figure 6-3](#). In this figure, context sensitivity between the views is also shown with the `ram_master` selection highlighted in both the **System Contents** and **Hierarchy** tabs.

Figure 6-3: Hierarchy Tab Expanding Elements in the System Contents Tab



You can use the **Hierarchy** tab to browse, connect, and parameterize IP in your system. The **Hierarchy** tab allows you to drive changes in other views and interact with your system in more detail. As shown in [Figure 6-3](#), the **Hierarchy** tab expands each interface that appears on the **System Contents** tab and allows you to view the subcomponents, associated elements, and signals for each interface. Use the **Hierarchy** tab to focus on a particular area of your system; coordinating selections in the **Hierarchy** tab with open views in your workspace. Reviewing your system using the **Hierarchy** tab in conjunction with relevant views is also useful during the debugging phase because you can contain and focus your debugging efforts to a single element in your system.

The **Hierarchy** tab provides the following information and functionality:

- The connections between signals.
- The names of signals included in exported interfaces.
- Right-click menu to connect, edit, add, remove, or duplicate elements in the hierarchy.
- The internal connections of Qsys subsystems that are included as components. In contrast, the **System Contents** tab displays only the exported interfaces of Qsys subsystems included as components.

Using the Parameters Tab

The **Parameters** tab allows you to review and change component parameters.

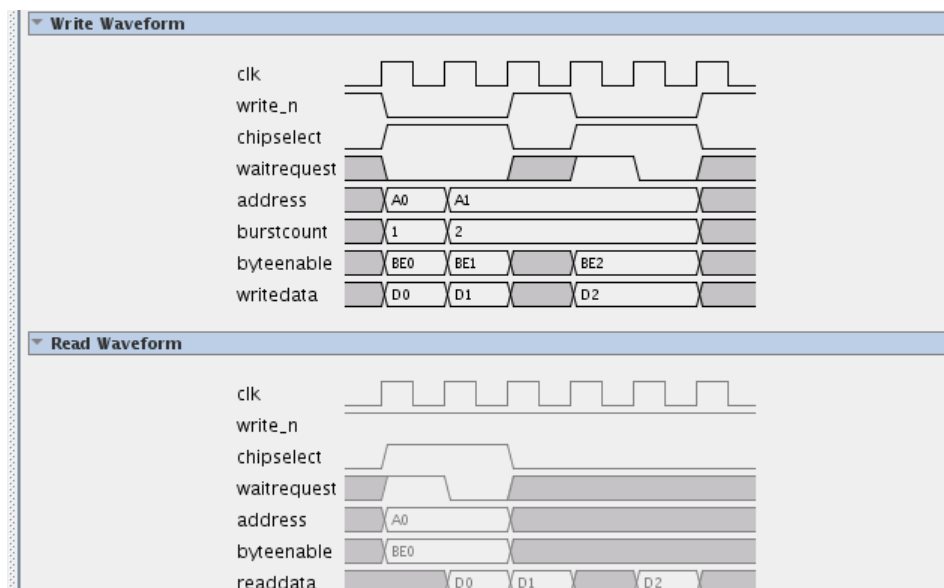
In the **Parameters** tab, Qsys displays the parameter editor for the current selection in the **Hierarchy** tab. When you double-click a component in the **System Contents** tab, Qsys opens a new window and displays the **Parameters**, **Block Symbol**, and **Presets** tabs together in a single window.

With the **Parameters** tab open, when you click an element in the **Hierarchy** tab, Qsys displays the parameter editor for the selected element.

In the parameter editor, you can change the name as it appears on the **System Contents** tab for top-level instances. Changes you make on the **Parameters** tab are immediately reflected on open views in your workspace.

If the current selection is for an interface in the system, the **Parameters** tab also allows you to review interface timing. [Figure 6-4](#) shows the timing for the Avalon-MM DMA write master for the PCI Express Subsystem Example. Qsys displays the read and write waveforms at the bottom of the **Parameters** tab.

Figure 6-4: Avalon-MM Write Master Timing Waveforms Available on the Parameters Tab

**Related Information**

- [PCI Express Subsystem Example](#) on page 6-32

Using the Presets Tab

In this view, Qsys displays the presets for the currently selected component.

The **Presets** tab allows you to create, modify, and save custom component or IP core parameter values as a preset file. You can then apply the parameter values in the preset file to the current component that you are parameterizing.

Related Information

- [Presets Editor \(Qsys\)](#)

Working With Presets for Supported IP Components

Some components provide preset configurations. If the component you are adding has presets available, then the Presets Editor appears in the editor window and lists presets that you can apply to your component, depending on the design protocol. When you apply a preset to a component, the parameters with specific required values for the protocol are automatically set for you.

Note: You can also access the Presets Editor by clicking **View > Presets**.

You can search for text to filter the **Presets** list. For example, if you select the **DDR3 SDRAM Controller with UniPHY** component, and then type **1g micron 256**, the **Presets** list shows only those presets that apply to the 1g micron 256 filter request. Presets whose parameter values match the current parameter settings are shown in bold.

Selecting a preset does not prevent you from changing any parameter to meet the requirements of your design. Clicking **Update** allows you to update parameter values for a custom preset. The **Update Preset** dialog box displays the default value, which you can edit, and the current value, which is static.

You can also create your own preset by clicking **New**. When you create a preset, you specify a name, description and the list of parameters whose values are set by the preset. You can remove a preset from the Quartus II project directory by clicking **Delete**.

Related Information

[Presets Editor](#)

Using the Block Symbol Tab

In this view, Qsys displays the block symbol for the currently selected element.

When the Block Symbol view is open, Qsys displays a graphical representation of the element selected in the **Hierarchy** or **System Contents** tabs. In the **Block Symbol** tab, the Show signals options allows you to turn on or off signal graphics, if applicable.

The **Block Symbol** tab reflects changes made in other views.

Using the Address Map Tab

The **Address Map** tab provides a table including the memory-mapped slaves in your design and the address range that each connected memory-mapped master uses to address each slave.

The table shows the slaves on the left and masters across the top, with the address span of the connection shown in each cell. If there is no connection between a master and a slave, the table cell is empty.

You can design a system where two masters access a slave at different addresses. If you use this feature, the **Base** and **End** address columns of the **System Contents** tab are labeled "mixed" rather than providing the address range.

Follow these steps to change or create a connection between master and slave components:

1. In Qsys, click the **Address Map** tab.
2. Locate the table cell that represents the connection between the master and slave component pair.
3. Either type in a base address, or update the current base address in the cell.

Note: The base address of a slave component must be a multiple of the address span of the component. This restriction is part of the Qsys interconnect to allow the address decoding logic to be efficient, and to achieve the best possible f_{MAX} .

Using the Clock Tab

The **Clocks** tab defines the **Name**, **Source**, and frequency (**MHz**) of each clock in your system.

Click **Add** to add a new clock to the system.

Using the Project Settings Tab

The **Project Settings** tab allows you to view and change the properties of your Qsys system.

Table 6-1: System-Level Parameters Available on the Project Settings Tab

Parameter Name	Description
Device Family	Specifies the Altera device family.

Parameter Name	Description
Device	Specifies the target device for the selected device family.
Clock crossing adapter type	<p>Specifies the default implementation for automatically inserted clock crossing adapters. The following choices are available:</p> <ul style="list-style-type: none"> • Handshake—This adapter uses a simple hand-shaking protocol to propagate transfer control signals and responses across the clock boundary. This methodology uses fewer hardware resources than the FIFO type because each transfer is safely propagated to the target domain before the next transfer can begin. The Handshake adapter is appropriate for systems with low throughput requirements. • FIFO—This adapter uses dual-clock FIFOs for synchronization. The latency of the FIFO-based adapter is a couple of clock cycles more than the handshaking clock crossing component. However, the FIFO-based adapter can sustain higher throughput because it supports multiple transactions at any given time. The FIFO-based clock crossers require more resources. The FIFO adapter is appropriate for memory-mapped transfers requiring high throughput across clock domains. • Auto—If you select Auto, Qsys specifies the FIFO adapter for bursting links, and the Handshake adapter for all other links.
Limit interconnect pipeline stages to	<p>Specifies the maximum number of pipeline stages that Qsys may insert in each command and response path to increase the f_{MAX} at the expense of additional latency. You can specify between 0–4 pipeline stages, where 0 means that the interconnect has a combinational data path. Choosing 3 or 4 pipeline stages may significantly increase the logic utilization of the system. This setting is specific for each Qsys system or subsystem, meaning that each subsystem can have a different setting. Note that the additional latency is for both the command and response directions.</p> <p>Note: You can manually adjust this setting in the Memory-Mapped Interconnect tab accessed by clicking Show System With Qsys Interconnect command on the System menu.</p>
Generation Id	A unique integer value that is set to a timestamp just before Qsys system generation that Qsys uses to check for software compatibility.

Note: Qsys generates a warning message if the selected device family and target device do not match the Quartus II software project settings. Also, when you open Qsys from within the Quartus II software, the device type in your Qsys project is replaced with the selected device in your open Quartus II software project.

Related Information

[Manually Controlling Pipelining in the Qsys Interconnect](#) on page 6-20

Using the Instance Parameters Tab

The **Instance Parameters** tab allows you to define parameters for a Qsys system. You can use instance parameters to modify a Qsys system when you use the system as a subcomponent in another Qsys system. The higher-level Qsys system can assign values to these instance parameters.

The **Instance Script** on the **Instance Parameters** tab defines how the specified values for the instance parameters should affect your Qsys design subcomponents. The instance script allows you to make queries about the instance parameters you define and set the values of the parameters for the subcomponents in your design.

When you click **Preview Instance**, Qsys creates a preview of the current Qsys system with the specified parameters and instance script, and shows the parameter editor for the instance. This command allows you to see how an instance of this system appears when you use it in another system. The preview instance does not affect your saved system.

To use instance parameters, the components or subsystems in your Qsys system must have parameters that can be set when they are instantiated in a higher-level system. Many components in the Library have parameters that you can set when adding the component to your system. If you create your own IP components, you use the `_hw.tcl` file to specify which parameters can be set when the component is added to a system. If you create hierarchical Qsys systems, each Qsys system in the hierarchy can include instance parameters to pass parameter values through multiple levels of hierarchy.

Related Information

[Working with Instance Parameters in Qsys](#)

Creating an Instance Script

The first command in an instance script must specify the Tcl command version for the script. This command ensures the Tcl commands behave identically in future versions of the tool. Use the following Tcl command to specify the version of the Tcl commands, where *<version>* is the Quartus II software version number, such as 13.1:

```
package require -exact qsys <version>
```

To use Tcl commands that work with instance parameters in the instance script, you must specify the commands within a Tcl procedure called a composition callback. In the instance script, you specify the name for the composition callback with the following command:

```
set_module_property COMPOSITION_CALLBACK <name of callback procedure>
```

Specify the appropriate Tcl commands inside the Tcl procedure with the following syntax:

```
proc <name of procedure defined in previous command> {}  
{#Tcl commands to query and set parameters go here}
```

Use Tcl commands in the procedure to query the parameters of a Qsys system, or to set the values of the parameters of the subcomponents instantiated in the system.

Table 6-2: Supported Tcl Commands Used in Instance Scripts

Command Name	Value	Description
get_parameters	None	Get the names of all defined parameters (as a space-separated list).
get_parameter_value	<parameter name >	Get the value of a parameter.
get_instance_parameters	<instance name>	Get the names of parameters on a child instance that can be manipulated by the parent (as a space-separated list).
get_instance_parameter_value	<instance name>	Get the value of a parameter for a child instance.
send_message	<message level> <message text>	Send a message to the user of the component, using one of the message levels Error, Warning, Info, or Debug. Enclose text with multiple words in quotation marks.
set_instance_parameter_value	<instance name> <parameter name> <parameter value>	Set a parameter value for a child instance.

You can use standard Tcl commands to manipulate parameters in the script, such as the `set` command to create variables, or the `expr` command for mathematical manipulation of the parameter values.

Example 6-1 shows an instance script of a system that uses a parameter called `pio_width` to set the width parameter of a parallel I/O (PIO) component. Note that the script combines the `get_parameter_value` and `set_instance_parameter_value` commands using brackets.

Example 6-1: Instance Script Example

```
# Request a specific version of the scripting API
package require -exact qsys 13.1

# Set the name of the procedure to manipulate parameters:
set_module_property COMPOSITION_CALLBACK compose

proc compose {} {

# Get the pio_width parameter value from this Qsys system and
# pass the value to the width parameter of the pio_0 instance

set_instance_parameter_value pio_0 width \
```

```
[get_parameter_value pio_width]
}
```

Related Information

[Component Interface Tcl Reference](#)

Using the Interconnect Requirements Tab

The **Interconnect Requirements** tab allows you to assign interconnect requirements for the system or an interface. The **Interconnect Requirements** assignments influence Qsys interconnect generation.

Interconnect Requirements settings also appear in other tabs. For instance, the **Limit interconnect pipeline stages** option appears on the **Project Settings** tab.

Selections in the **Setting** and **Value** lists vary depending on your selection in the Identifier column.

Configuring Interconnect Requirements for the System

Selecting `$system` in the **Identifier** list on the **Interconnect Requirements** tab allows you to apply system-wide interconnect assignments.

Table 6-3: Settings and Values for the \$system Identifier

Setting	Value
Limit interconnect pipeline stages to —Specifies the maximum number of pipeline stages that Qsys may insert in each command and response path to increase the f_{MAX} at the expense of additional latency.	You can specify between 0–4 pipeline stages, where 0 means that the interconnect has a combinational data path. Choosing 3 or 4 pipeline stages may significantly increase the logic utilization of the system. This setting is specific for each Qsys system or subsystem, meaning that each subsystem can have a different setting. Note that the additional latency is added once on the command path, and once on the response path.

Setting	Value
Clock crossing adapter type —Specifies the default implementation for automatically inserted clock crossing adapters.	<ul style="list-style-type: none"> • Handshake—This adapter uses a simple handshaking protocol to propagate transfer control signals and responses across the clock boundary. This methodology uses fewer hardware resources because each transfer is safely propagated to the target domain before the next transfer can begin. The Handshake adapter is appropriate for systems with low throughput requirements. • FIFO—This adapter uses dual-clock FIFOs for synchronization. The latency of the FIFO-based adapter is a couple of clock cycles more than the handshaking clock crossing component. However, the FIFO-based adapter can sustain higher throughput because it supports multiple transactions at any given time. The FIFO-based clock crossers require more resources. The FIFO adapter is appropriate for memory-mapped transfers requiring high throughput across clock domains. • Auto—If you select Auto, Qsys specifies the FIFO adapter for bursting links, and the Handshake adapter for all other links.
Automate default slave insertion —Specifies whether you want Qsys to automatically insert a default slave for undefined memory region accesses during system generation.	True or False

Configuring Interconnect Requirements for an Interface

Selecting an interface in the **Identifier** list on the **Interconnect Requirements** tab allows you to apply interface interconnect assignments.

Setting	Value
Security	<ul style="list-style-type: none"> • Non-secure • Secure • Secure ranges • TrustZone-aware <p>Note: You can also set these values in the Security column in the System Contents tab.</p>
Secure address ranges	Allows you to type in an address valid range.
Add performance monitor	True or False

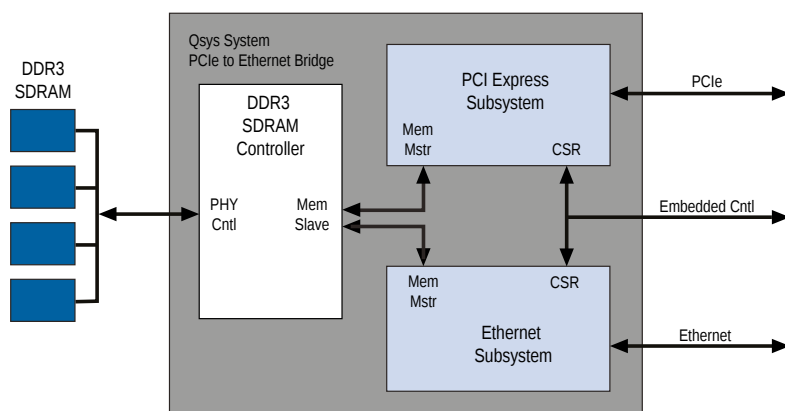
Creating Hierarchical Systems

Qsys supports team-based and hierarchical system design.

You can include any Qsys system as a component in another Qsys system. In a team-based design flow, you can have one or more systems in your design developed simultaneously by other team members, decreasing time-to-market for the complete design.

Figure 6-5 shows the top-level of a Qsys hierarchical design that implements a PCI Express™ to Ethernet bridge. This example combines separate PCI Express and Ethernet subsystems with Altera's DDR3 SDRAM Controller with UniPHY IP core.

Figure 6-5: Top-Level for a PCI Express to Ethernet Bridge



Hierarchical system design in Qsys offers the following advantages:

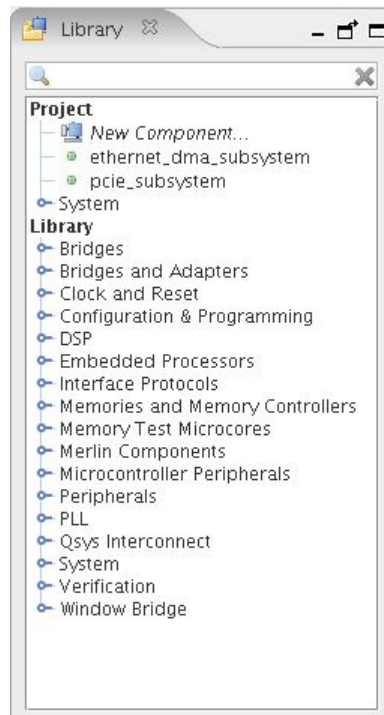
- Enables team-based, modular design by dividing large designs into subsystems.
- Enables design reuse by allowing you to use any Qsys system as a component.
- Enables scalability by allowing you to instantiate multiple instances of a Qsys system.

Adding Systems to the Library

Any Qsys system is available for use as a component in other Qsys systems.

Figure 6-6 shows the library, including the `pcie_subsystem` as a component in the library for the **Figure 6-10** example system. To include systems as components in other designs, you can add the system to the library, or include the directory for the system in the IP search path for Qsys.

Figure 6-6: Qsys Library



Creating a Component Based on a System

The **Export System as hw.tcl Component** command on the File menu allows you to save the system currently open in Qsys as an **_hw.tcl** file in the current working directory. The saved system appears as a new component in the **System** category under **Project** in the Qsys Library.

Qsys 64-Bit Addressing Support

Qsys interconnect supports up to 64-bit addressing for all Qsys interfaces and components, with a range of: 0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF, inclusive.

In Qsys, address parameters appear in the **Base** and **End** columns on the **System Contents** tab, on the **Address Map** tab, in the parameter editor, and in validation messages. The Qsys GUI displays as many digits as needed in order to display the top-most set bit, for example, 12 hex digits for a 48-bit address.

A Qsys system can have multiple 64-bit masters, with every master having its own address space. You can share slaves among masters and masters can map slaves in different ways; for example, one master can interact with slave 0 at base address 0000_0000_0000, and another master can see the same slave at base address c000_000_000.

Qsys supports 64-bit addresses for narrow-to-wide and wide-to-narrow transactions across Avalon-MM and AXI interfaces.

Quartus II debug tools that provide access to the state of an addressable system via the Avalon-MM interconnect are also 64-bit compatible and process within a 64-bit address space, including a JTAG to Avalon master bridge.

For more information about 64-bit support, refer to **Address Span Extender** in *Creating a System with Qsys*.

Related Information

- [Creating a System with Qsys](#)

Creating Secure Systems (TrustZones)

TrustZone is the security extension of the ARM architecture. It includes the concept of secure and non-secure transactions, and a protocol for processing between the designations. TrustZone security support is a part of the Qsys interconnect.

In Qsys, you can set memory-mapped interfaces to secure, non-secure, or TrustZone-aware. AXI masters are always treated as TrustZone-aware. Unless specified, all other master and slave interfaces (such as Avalon-MM) are treated as non-secure, by default.

Qsys provides compilation-time TrustZone support for non-TrustZone-aware components, for example, when an Avalon master needs to communicate with a secure AXI slave. For example, the designer can specify whether the connection point is secure or non-secure at compilation time. You can specify secure address ranges on memory slaves, if a per-interface security setting is not sufficient.

For TrustZone-aware masters, the interconnect uses the master's AXPROT signal to determine the security status of each transaction.

Table 6-4 summarizes secure and non-secure access between master, slave, and memory components in Qsys. Per-access refers to allowing a TrustZone-aware master to allow or disallow a particular access (or transactions).

Table 6-4: Secure and Non-Secure Access Between Master, Slave, and Memory Components

Transaction Type	TrustZone-aware Master	Non-TrustZone-aware Master Secure	Non-TrustZone-aware Master Non-Secure
TrustZone-aware slave/ memory	OK	OK	OK
Non-TrustZone-aware slave (secure)	Per-access	OK	Not allowed
Non-TrustZone-aware slave (non-secure)	OK	OK	OK
Non-TrustZone-aware memory (secure region)	Per-access	OK	Not allowed
Non-TrustZone-aware memory (non-secure region)	OK	OK	OK

If a master issues transactions that fall into the per-access or not allowed cells, as described in the table above, your design must contain a default slave. A transaction that violates security is rerouted to the default slave and subsequently terminated with an error. You can connect any slave as the default slave, which allows it to respond to the master with errors. You can share the default slave between multiple masters. You have one default slave for each interconnect domain, which is a group of connected memory-mapped masters

and slaves that share the same interconnect. Use the `altera_axi_default_slave` component as the default slave because this component has the required TrustZone features.

Note: For more information about interconnect domains, refer to *Qsys Interconnect*.

In Qsys, you can achieve an optimized secure system by partitioning your design. For example, for masters and slaves under the same hierarchy, it is possible for a non-secure master to initiate continuous transactions resulting in unsuccessful transfer to a secure slave. In the case of memory aliasing, you must carefully designate secure or non-secure address maps to maintain reliable data.

Related Information

- [Qsys Interconnect](#)

Managing Secure Settings in Qsys

To create a secure design, you must first add masters and slaves and the connections between them. After you establish connections between the masters and slaves, you can then set the security options, as required, with options in the **Security** column.

On the **System Contents** tab, in the **Security** column, the following selections are available for master, slave, and memory components:

- **Non-secure**—Master issues only non-secure transactions. There is no security available for the slave.
- **Secure**—Master issues only secure transactions. For the slave, Qsys prevents non-secure transactions from reaching the slave, and routes them to the default slave for the master that issued the transaction.
- **Secure Ranges**—Slave only, the specified address ranges within the slave's address span are secure; all others are not. The format is a comma-separated list of inclusiveLow:inclusiveHigh addresses, for example, `0x0 : 0xffff`, `0x2000 : 0x20ff`.
- **TrustZone-aware**—Master issues either secure or non-secure transactions at run-time. The slave accepts either secure or non-secure transactions at run-time.

After setting security options for the masters and slaves, you must identify those masters that require a default slave before generation. To designate a slave as the default slave, turn on **Default Slave** in the **Systems Contents** tab. A master can have only one default slave.

Note: The **Security** and **Default Slave** columns in the **System Contents** tab are hidden by default. You can turn them on with the right-click menu in the **System Contents** header.

Understanding Compilation-Time Security Configuration Options

The following compile-time configurations are available when creating secure designs that have mixed secure and non-secure components:

- Masters that support TrustZone and are connected to slaves that are compile-time secure. This configuration requires a default slave.
- Slaves that support TrustZone and are connected to masters that have compile-time secure settings. This configuration does not require a default slave.
- Master connected to slaves with secure address ranges. This configuration requires a default slave.

Accessing Undefined Memory Regions

When a transaction from a master targets a memory region that is not specified in the slave memory map, it is known as an "access to an undefined memory region." To ensure predictable response behavior when

this occurs, you can add a default slave to the design. All undefined memory region accesses are then routed to the default slave, which then terminates the transaction with an error response.

You can connect any memory-mapped slave as a default slave. Altera recommends that you have only one default slave for each domain in your design. Accessing undefined memory regions can occur in the following cases:

- When there are gaps within the accessible memory map region that are within the addressable range of slaves, but are not mapped.
- Accesses by a master to a region that does not belong to any slaves that is mapped to the master.
- When a non-secured transaction is accessing a secured slave. This applies to only slaves that are secured at compilation time.
- When a read-only slave is accessed with a write command, or a write-only slave is accessed with a read command.

To designate a slave as the default slave, for the selected component, turn on **Default Slave** on the **Systems Content** tab.

Note: If you do not specify the default slave, Qsys automatically assigns the slave at the lowest address within the memory map for the master that issues the request as the default slave.

Viewing the Qsys Interconnect

The System with Qsys Interconnect window allows you to see the contents of the Qsys interconnect before you generate your system. In this view of your system, you can view a graphical representation of the generated interconnect. Qsys converts connections between interfaces to interconnect logic during system generation.

You access the System with Qsys Interconnect window by clicking **Show System With Qsys Interconnect** command on the System menu.

The system with Qsys Interconnect window consists of the following tabs:

- **System Contents**—Displays the original instances in your system, as well as the inserted interconnect instances. Connections between interfaces are replaced by connections to interconnect where applicable.
- **System Inspector**—Displays a system hierarchical navigator, expanding the system contents to show modules, interfaces, signals, contents of subsystems, and connections.
- **Memory-Mapped Interconnect**—allows you to select a memory-mapped interconnect module and view its internal command and response networks. You can also insert pipeline stages to achieve timing closure.

The **System Contents** and **System Inspector** tabs are read-only. Edits that you apply on the **Memory-Mapped Interconnect** tab are automatically updated on the **Interconnect Requirements** tab.

Using the Memory-Mapped Interconnect Tab

The **Memory-Mapped Interconnect** tab in the System with Qsys Interconnect window is a graphical representation of command and response datapaths in your system. These datapaths allow you finer control over pipelining in the interconnect. Qsys displays separate graphs for the command and response datapaths. You can access the datapaths by clicking their respective tabs in the **Memory-Mapped Interconnect** tab.

Each node element in a graph can represent either a master or slave that communicates over the interconnect, or an interconnect sub-module. Each edge in a graph is an abstraction of connectivity between elements, and its direction represents the flow of the commands or responses.

Click **Highlight Path** to better identify edges and paths between modules. Turn on **Show Pipeline Locations** to add greyed-out registers on edges where pipelining is allowed in the interconnect.

Note: You must have more than one module selected in order to highlight a path.

Manually Controlling Pipelining in the Qsys Interconnect

The **Memory-Mapped Interconnect** tab allows you to manipulate pipeline connections in the Qsys interconnect. You access the **Memory-Mapped Interconnect** tab by clicking **Show System With Qsys Interconnect** command on the System menu.

Note: To increase interconnect frequency, you should first try increasing the value of the **Limit interconnect pipeline stages to** option on the **Project Settings** tab. You should only consider manually pipelining the interconnect if changes to this option do not improve frequency, and you have tried all other options to achieve timing closure, including the use of a bridge. Manually pipelining the interconnect should only be applied to complete systems.

1. In the **Project Settings** tab, first try increasing the value of the **Limit interconnect pipeline stages to** option until it no longer gives significant improvements in frequency, or until it causes unacceptable effects on other parts of the system.
2. In the Quartus II software, compile your design and run timing analysis.
3. Identify the critical path through the interconnect and determine the approximate mid-point. The following is an example of a timing report where the critical path is located in the interconnect.

```
2.800 0.000 cpu_instruction_master|out_shifter[63]|q
3.004 0.204 mm_domain_0|addr_router_001|Equal5~0|datac
3.246 0.242 mm_domain_0|addr_router_001|Equal5~0|combout
3.346 0.100 mm_domain_0|addr_router_001|Equal5~1|dataa
3.685 0.339 mm_domain_0|addr_router_001|Equal5~1|combout
4.153 0.468 mm_domain_0|addr_router_001|src_channel[5]~0|datad
4.373 0.220 mm_domain_0|addr_router_001|src_channel[5]~0|combout
```

4. **System > Show System With Qsys Interconnect.**
5. In the **Memory-Mapped Interconnect** tab, select the interconnect module that has the critical path. You can determine the name of the interconnect module from the hierarchical node names in the timing report.
6. Click **Show Pipelinable Locations**. Qsys display all pipelinable locations in the interconnect. You can right-click a pipelinable location to open a menu that allows you to insert or remove a pipeline stage.
7. Find the pipelinable location that is closest to the mid-point of the critical path. The names of blocks in the memory-mapped interconnect view correspond to the module instance names in the timing report.
8. Right-click the location where you want to insert a pipeline stage, and then click **Insert Pipeline**.
9. Regenerate the Qsys system, recompile the design, and then rerun timing analysis. If necessary, repeat the manual pipelining process again until timing requirements are met.

Manual pipelining has the following limitations:

- If you make changes to your original system's connectivity after manually pipelining an interconnect, your inserted pipelines may become invalid. Warning messages are displayed at generation time if invalid pipeline stages are detected. You can remove invalid pipeline stages with the **Remove Stale Pipelines** option button in the **Memory-Mapped Interconnect** tab. Altera recommends not making changes to the system's connectivity after manual pipeline insertion.
- Review manually-inserted pipelines when upgrading to newer versions of Qsys. Manually-inserted pipelines in one version of Qsys might not be valid in a future version.

Related Information

[Qsys System Design Components](#)

[Configuring Interconnect Requirements for the System](#) on page 6-13

Integrating Your Qsys Design with the Quartus II Software

To integrate a Qsys system into a Quartus II project, you must add one of the following files to your Quartus II project (but not both) on the **Files** tab in the **Settings** dialog box.

- **Quartus II IP File (.qip)**—Qsys generates a **.qip** file when you generate your Qsys design. Integrating your Qsys design with your Quartus II project using the **.qip** file is preferable when you want full control over generated files and Quartus II compilation phases. If you want to manage the HDL generation for your Qsys system, you generate your Qsys system first, then add the **.qip** file to your Quartus II project.
- **Qsys System File (.qsys)**—Integrating your Qsys design with your Quartus II project by adding the **.qsys** design file to your Quartus II project is more convenient for cases when there is no customization or scripts in the design flow. If you do not want to generate your Qsys system manually, add the **.qsys** file to your Quartus II project. You can add one or more top-level **.qsys** files to your Quartus II project.

Note: When integrating your Qsys designs with your Quartus II software project, you should decide on which integration flow you want to use (either adding the **.qsys** file, or the **.qip** file to your Quartus II project, but not both), and then maintain a consistent integration flow throughout development. Mixing integration flows might result in two sets of generated output files, at which point you would then have to keep track of which one is currently in use. The Quartus II software generates an error message during compilation if you add both the **.qip** and **.qsys** files to your Quartus II project.

Related Information

- [Managing Files in a Project](#)
- [Searching for Component Files to Add to the Library](#) on page 6-39
- [Generating a Qsys System](#) on page 6-23

Integrating with the .qsys File

To integrate your Qsys designs with the Quartus II software using the **.qsys** files, you create your designs in Qsys, save the design files as **<qsys design name>.qsys**, and then add the **.qsys** file(s) to your Quartus II project. When the Quartus II software starts the Analysis & Synthesis phase, it processes the **.qsys** files and generates the necessary HDL and system description files needed to compile your design.

You can add multiple **.qsys** files to a Quartus II project. Qsys stores the files generated from each **.qsys** file in the **/db/<qsys file name>** directory under the Quartus II project directory.

When a Qsys design file includes an IP component which is outside of the project directory, the directory of the **.qsys** file, or the **/ip** subdirectory, you must add these dependency paths to the Qsys **IP Search Path** before compilation.

Note: The following are design guidelines and warnings when integrating your Qsys designs with the Quartus II software:

- When you integrate your Qsys designs with the Quartus II software using the **.qsys** file, you must manually run any IP customization scripts at the appropriate stages of the Quartus II compilation process. There is no automation support for running scripts between the Quartus II software compilation stages. The *Implementing and Parameterizing Memory IP* reference describes running placement scripts for embedded memory IP interfaces.
- Do not edit the files generated under the **/ip/<qsys file name>** directory, as they are overwritten during subsequent runs of Analysis & Synthesis.

Related Information

- [Implementing and Parameterizing Memory IP](#)

Integrating with the .qip File

Qsys generates the Quartus II IP File (**.qip**) during system generation. If you choose to integrate your Qsys design with your Quartus II project using the **.qip** file, after you generate your Qsys design, you must add the **.qip** file to your Quartus II project.

The **.qip** file lists the files necessary for compilation and provides the Quartus II software with the required information about your Qsys system. The **.qip** file is saved in the **<qsys file name>/synthesis** directory, and includes references to the following information:

- HDL files in the Qsys system
- TimeQuest Timing Analyzer Synopsys Design Constraint Files (**.sdc**)
- Component definition files for archiving purposes

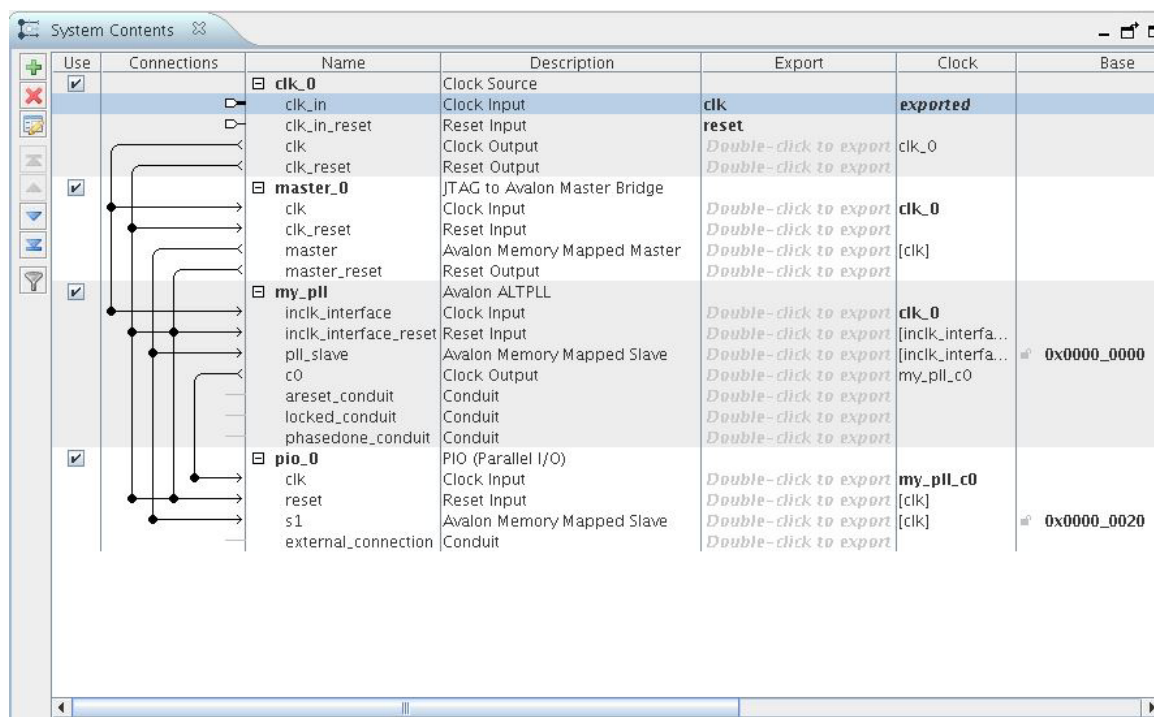
Setting Clock Constraints

Many IP cores include Synopsys Design Constraint (**.sdc**) files that provide timing constraints for the logic in the IP design. Generated **.sdc** files are included in your Quartus II project with the **.qip** file. For your top-level clocks and PLLs, you must provide clock and timing constraints in SDC format to direct synthesis and fitting to optimize the design appropriately, and to evaluate performance against timing constraints.

You can specify a base clock assignment for each clock input in the TimeQuest GUI or with the **create_clock** command, and then you can use the **derive_pll_clocks** command to define the PLL clock output frequencies and phase shifts for all PLLs in the Quartus II project.

Figure 6-7 illustrates the **.sdc** commands required for the case of a single clock input signal called **clk**, and one PLL with a single output.

Figure 6-7: Single Clock Input Signal



For this system, use the following commands in your **.sdc** file for the TimeQuest Timing Analyzer:

```
create_clock -name master_clk -period 20 [get_ports {clk}]
derive_pll_clocks
```

Related Information

- [The Quartus II TimeQuest Timing Analyzer](#)

Generating a Qsys System

The **Generation** dialog box allows you to choose options for generation of synthesis and simulation files.

Generating Output Files

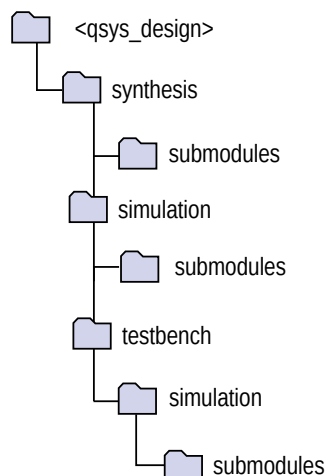
Qsys system generation creates the interconnect between components and generates synthesis and simulation files. You specify the files that you want to generate in the **Generation** dialog box. You can generate simulation models, simulation testbench files, as well as HDL files for Quartus II synthesis, or a Block Symbol File (**.bsf**) for schematic design.

For your simulation model and testbench system, you can select Verilog HDL or VHDL for the top-level module language, which applies to the system's top-level definition and child instances that support generation for the selected target language.

For synthesis, you can select the top-level module language as Verilog HDL or VHDL, which applies to the system's top-level definition.

Qsys places the generated output files in a subdirectory of your project directory, along with an HTML report file. To change the default behavior, on the **Generation** tab, specify a new directory under **Output Directory**.

Figure 6-8: Qsys Generated Files Directory Structure



Each time you generate your system, Qsys overwrites these files, therefore, you should not edit Qsys-generated output files. If you have constraints, such as board-level timing constraints, Altera recommends that you create a separate Synopsys Design Constraints File (**.sdc**) and include that file in your Quartus II project. If you need to change top-level I/O pin names or instance name, Altera recommends you create a top-level HDL file that instantiates the Qsys system, so that the Qsys-generated output is instantiated in your design without any changes to the Qsys output files.

Note: Qsys generates the files in listed in [Table 6-5](#) to the **<qsys design>/simulation** folder.

Table 6-5: Qsys Generated Files

File Name or Directory Name	Description
<Qsys system>	The top-level Qsys system directory, in the Quartus II project directory
<Qsys system>.bsf	A Block Symbol File (.bsf) representation of the top-level Qsys system for use in Quartus II Block Diagram Files (.bdf).
<Qsys system>.html	A report for the system, which provides a system overview including the following information: <ul style="list-style-type: none"> • External connections for the system • A memory map showing the address of each slave with respect to each master to which it is connected • Parameter assignments for each component

File Name or Directory Name	Description
<Qsys system>. sopinfo	<p>Describes the components and connections in your system. This file is a complete system description and is used by downstream tools such as the Nios II tool chain. It also describes the parameterization of each component in the system; consequently, you can parse its contents to get requirements when developing software drivers for Qsys components.</p> <p>This file and the system.h file generated for the Nios II tool chain include address map information for each slave relative to each master that accesses the slave. Different masters may have a different address map to access a particular slave component.</p>
<Qsys system>. spd	Required input file for <code>ip-make-simscript</code> to generate simulation script for supported simulators. The .spd file contains a list of files generated for simulation, along with information about initializable memories.
<Qsys system>/ synthesis	This directory includes the Qsys-generated output files that the Quartus II software uses to synthesize your design.
<Qsys system>/ synthesis / <Qsys system>. v or <Qsys system>/ synthesis <Qsys system>. vhd	An HDL file for the top-level Qsys system that instantiates each submodule in the system for synthesis.
<Qsys_system>/ synthesis / <Qsys system>. regmap	If IP in the system contains register information, Qsys generates a .regmap file. The .regmap file describes the register map information on master and slave interfaces. This file complements the .sopinfo file by providing more detailed register information about the system. This enables register display views and user customizable statistics providers in the SystemConsole.
<Qsys system>/ synthesis / <Qsys system>. qip	This file this file includes all the info you need to synthesize the IP components in your system.
<Qsys system>/ synthesis/submodules	Contains Verilog HDL or VHDL submodule files for synthesis.
<Qsys system>/ simulation	This directory includes the Qsys-generated output files to simulate your Qsys design or testbench system.
<Qsys system>/ simulation / <Qsys system>. sip	This file contains information required for NativeLink simulation of IP components in your system. You must add the .sip file to your Quartus II project.

File Name or Directory Name	Description
<code><Qsys system>/simulation/</code> <code><Qsys system>.v</code> or <code><Qsys system>/simulation/</code> <code><Qsys system>.vhd</code>	An HDL file for the top-level Qsys system that instantiates each submodule in the system for simulation.
<code><Qsys system>/simulation/submodules</code>	Contains Verilog HDL or VHDL submodule files for simulation.
<code><Qsys system>/simulation/mentor</code>	Contains a ModelSim® script msim_setup.tcl to set up and run a simulation.
<code><Qsys system>/simulation/aldec</code>	Contains Riviera-PRO script rivierapro_setup.tcl to setup and run a simulation.
<code><Qsys system>/simulation/synopsys/vcs</code>	Contains a shell script vcs_setup.sh to set up and run a VCS® simulation.
<code><Qsys system>/simulation</code> <code>/synopsys/vcsmx</code>	Contains a shell script vcsmx_setup.sh and synopsys_sim.setup to set up and run a VCS MX simulation.
<code><Qsys system>/simulation/cadence</code>	Contains a shell script ncsim_setup.sh and other setup files to set up and run an NCSIM simulation.
<code><Qsys system>/testbench</code>	Contains a Qsys testbench system.
<code><Qsys system> /testbench/</code> <code><Qsys system>_tb.qsys</code>	A Qsys testbench system.
<code><Qsys system>/testbench/</code> <code><Qsys system>_tb.v</code> or <code><Qsys system>/testbench/</code> <code><Qsys system>_tb.vhd</code>	The top-level testbench file, which connects BFM to the top-level interfaces of <code><qsys_design> .qsys</code> .
<code><Qsys system>/testbench/<module name></code> <code>_<master interface name>.svd</code>	<p>Allows HPS System Debug tools to view the register maps of peripherals connected to the HPS within a Qsys design.</p> <p>Similarly, during synthesis the .svd files for slave interfaces visible to System Console masters are stored in the .sof file in the debug section. System Console reads this section, which Qsys can query for register map information. When a slave is open, Qsys can access the registers by name.</p>

CMSIS Support for Qsys Systems With An HPS Component

Qsys systems that contain a Hard Processor System (HPS) component generate a System View Description (.svd) file that lists peripherals connected to the ARM processor.

The System View Description File (.svd) (or CMSIS-SVD) file format is an XML schema specified as part of the Cortex Microcontroller Software Interface Standard (CMSIS) provided by ARM. The CMSIS-SVD file allows HPS System Debug tools (such as the DS-5 Debugger) to gain visibility into the register maps of peripherals connected to the HPS within a Qsys system.

Related Information

- [Component Interface Tcl Reference](#)
- [CMSIS - Cortex Microcontroller Software](#)

Viewing the HDL Example

The **HDL Example** dialog box, accessed from the Generate menu, provides the top-level HDL definition of your system in either Verilog HDL or VHDL, and also displays VHDL component declarations.

You can copy and paste the example into a top-level HDL file that instantiates the Qsys system, if the system is not the top-level module in your Quartus II project.

Simulating a Qsys System

The Qsys **Generation** dialog box provides options for generating Qsys simulation.

The following options are available in the **Generate** dialog box.

- Generate the Verilog HDL, VHDL, or mixed-language simulation model for your system to use in your own simulation environment.
- Generate a standard or simple testbench system with BFM or Mentor Verification IP (for AXI3/AXI4) components that drive the external interfaces of your system, and generate a Verilog HDL or VHDL simulation model for the testbench system to use in your simulation tool.
- First generate a testbench system, and then modify the testbench system in Qsys before generating its simulation model.

In most cases, you should select only one of the simulation model options, that is generate a simulation model for the original system, or for the testbench system. [Table 6-6](#) summarizes the options in the **Generate** dialog box that correspond to the simulation files described above.

Table 6-6: Summary of Settings Simulation and Synthesis in the Generate Dialog Box

Simulation Setting	Value	Description
Create simulation model	None Verilog VHDL	Generates simulation model files and simulation scripts. Use this option to include the simulation model in your own custom testbench or simulation environment. You can also use this option to generate models for a testbench system that you have modified.
Allow mixed-language simulation	On Off	Generates a mixed language simulation model generation. If you have a mixed-language simulator license, generating for mixed-language simulation can shorten the generation time, and produce files that may simulate faster. When turned off, all simulation files are generated in the selected simulation model language.
Create testbench Qsys system	Standard, BFM for standard Qsys Interconnect	Creates a testbench Qsys system with BFM components attached to exported Avalon and AXI3 interfaces. Includes any simulation partner modules specified by IP cores in the system. The testbench generator supports AXI interfaces and can connect AXI3/AXI4 interfaces to Mentor Graphics AXI3/AXI4 master/slave BFM. However, BFMs support address widths only up to 32-bits.
	Simple, BFMs for clocks and resets	Creates a testbench Qsys system with BFM components driving only clock and reset interfaces. Includes any simulation partner modules specified by IP cores in the system.

Simulation Setting	Value	Description
Create testbench simulation model	None Verilog VHDL	Creates simulation model files and simulation scripts for the testbench Qsys system specified in the setting above. Use this option if you do not need to modify the Qsys-generated testbench before running the simulation.
Create HDL design files for synthesis	On Off	Creates Verilog HDL or VHDL design files.
Top-level module language for synthesis	Verilog VHDL	Creates the top-level module in the system in the selected language.
Create block symbol files (.bsf)	On Off	You can optionally create a (.bsf) file to use in schematic Block Diagram File (.bdf) designs.
Output Directory	< directory name >	Allows you to browse and locate an alternate directory than the project directory for each generation target.

Related Information

- [Avalon Verification IP Suite User Guide](#)
- [Mentor Verification IP \(VIP\) Altera Edition \(AE\)](#)
- [Generating a System for Synthesis or Simulation](#)
- [Generation Dialog Box \(Qsys\)](#)

Generate and Modify the Testbench System

You can use the following steps to create a testbench system of your design.

1. Create a Qsys system.
2. Generate a testbench system in the Qsys **Generate** dialog box.
3. Open the testbench system in Qsys. Make changes, as needed, to the BFM, such as changing the BFM instance names and BFM **VHDL ID** value. You can modify the **VHDL ID** value in the **Altera Avalon Interrupt Source** component.
4. If you modified a BFM, generate the simulation model for the testbench system on the Qsys **Generation** tab. You can generate your simulation model in either Verilog HDL or VHDL.
5. Create a custom test program for the BFM.
6. Compile and load the Qsys design and testbench in your simulator, and then run the simulation.

Generate the Testbench System and a Simulation model at the Same Time (Verilog HDL only)

You can use the following design flow to create a testbench system and a simulation model of your Verilog HDL design.

1. Create a Qsys system.
2. Generate a testbench system and the simulation model for the testbench system in the Qsys **Generate** dialog box.
3. Create a custom test program for the BFM.
4. Compile and load the Qsys design and testbench in your simulator, and then run the simulation.

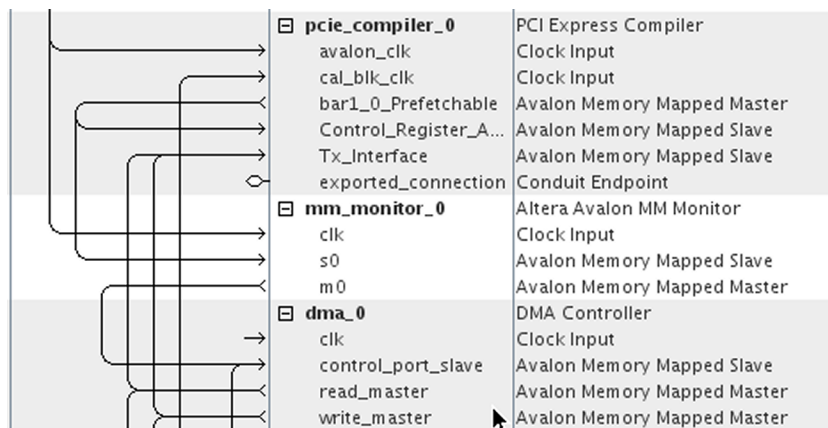
Adding Assertion Monitors

You can add monitors to Avalon-MM, AXI, and Avalon-ST interfaces in your system to verify protocol correctness and test coverage with a simulator that supports SystemVerilog assertions.

Note: Modelsim Altera Edition does not support SystemVerilog assertions. If you want to use assertion monitors, you will need to use an advanced simulator such as Mentor Questasim, Synopsys VCS, or Cadence Incisive.

Figure 6-9 demonstrates the use of monitors with an Avalon-MM monitor between the previously connected `pcie_compiler_bar1_0_Prefetchable` Avalon-MM master interface and the `dma_0_control_port_slave` Avalon-MM slave interface.

Figure 6-9: Inserting an Avalon-MM Monitor between Avalon-MM Master and Slave Interfaces



Similarly, you can insert an Avalon-ST monitor between Avalon-ST source and sink interfaces.

Simulation Scripts

Qsys generates simulation scripts to script the simulation environment set up for Mentor Graphics Modelsim[®] and Questasim[®], Synopsys VCS[®] and VCS MX[®], Cadence Incisive Enterprise Simulator[®] (NCSIM), and the Aldec Riviera-PRO[®] Simulator.

You can use the scripts to compile the required device libraries and system design files in the correct order and elaborate or load the top-level design for simulation.

The simulation scripts provide the following variables that allow flexibility in your simulation environment:

- **TOP_LEVEL_NAME**—If the Qsys testbench system is not the top-level instance in your simulation environment because you instantiate the Qsys testbench within your own top-level simulation file, set the TOP_LEVEL_NAME variable to the top-level hierarchy name.
- **QSYS_SIMDIR**—If the simulation files generated by Qsys are not in the simulation working directory, use the QSYS_SIMDIR variable to specify the directory location of the Qsys simulation files.
- **QUARTUS_INSTALL_DIR**—Points to the device family library.

Example 6-2 shows a simple top-level simulation HDL file for a testbench system `pattern_generator_tb`, which was generated for a Qsys system called `pattern_generator`. The `top.sv` file defines the top-level module that instantiates the `pattern_generator_tb` simulation model as well as a custom SystemVerilog test program with BFM transactions, called `test_program`.

Example 6-2: Top-Level Simulation HDL File Example

```
module top();  
    pattern_generator_tb tb();  
    test_program pgm();  
endmodule
```

Note: The VHDL version of the Altera Tristate Conduit BFM is not supported in Synopsys VCS, NCSim, and Riviera-PRO in the Quartus II software version 13.1. These simulators do not support the VHDL protected type, which is used to implement the BFM. For a workaround, use a simulator that supports the VHDL protected type.

Related Information

- [ModelSim-Altera software, Mentor Graphics ModelSim support](#)
- [Synopsys VCS and VCS MX support](#)
- [Cadence Incisive Enterprise Simulator \(IES\) support](#)
- [Aldec Active-HDL and Riviera-PRO support](#)

Simulating Software Running on a Nios II Processor

To simulate the software in a system driven by a Nios II embedded processor, generate the simulation model for a simple Qsys testbench system by completing the following steps:

1. On the **Generation** tab, set **Create testbench Qsys system** to **Simple, BFM for clocks and resets**.
2. Set **Create testbench simulation model** to **Verilog or VHDL**.
3. Click **Generate**.
4. Open the **Nios II Software Build Tools for Eclipse**.
5. Set up an application project and board support package (BSP) for the `<qsys_system>` **.sopcinfo** file.
6. Set up an application project and board support package (BSP) for the `<qsys_system>` **.sopcinfo** file.
7. To simulate, right-click the application project in Eclipse, point to **Run as**, and then click **4 Nios II ModelSim**. The **Run As Nios II ModelSim** command sets up the ModelSim simulation environment, compiles and loads the Nios II software simulation.

8. To run the simulation in ModelSim, type `run -all` in the ModelSim transcript window.
9. If prompted, set ModelSim configuration settings and select the correct Qsys Testbench Simulation Package Descriptor (**.spd**) file, `< qsys_system > _tb.spd`. The **.spd** file is generated with the testbench simulation model for Nios II designs and specifies all the files required for the Nios II software simulation.

Related Information

- [Getting Started with the Graphical User Interface \(Nios II\)](#)
- [Getting Started from the Command-Line \(Nios II\)](#)

System Examples

The following system examples demonstrate various design features and flows that you can replicate in your design.

[PCI Express Subsystem Example](#) on page 6-32

[Ethernet Subsystem Example](#) on page 6-34

[PCI Express to Ethernet Bridge Example](#) on page 6-36

[Hierarchical System Using Instance Parameters Example](#) on page 6-38

PCI Express Subsystem Example

Figure 6-10 and **Figure 6-11** show an example PCI Express subsystem. The application running on the root complex processor programs the DMA controller. The DMA controller's Avalon-MM read and write master interfaces initiate transfers to and from the DDR3 memory and to the PCI Express Avalon-MM TX data port. The system exports the DMA master interfaces through an Avalon-MM pipeline bridge. In the figure below, all three masters connect to a single slave interface. During system generation, Qsys automatically inserts arbitration logic to control access to this slave interface.

By default, the arbiter provides equal access to all requesting masters; however, you can weight the arbitration by changing the number of arbitration shares for the requesting masters. The second pipeline bridge allows an external master, such as a host processor, to also issue transactions to the CSR interfaces.

Figure 6-10: PCI Express Subsystem

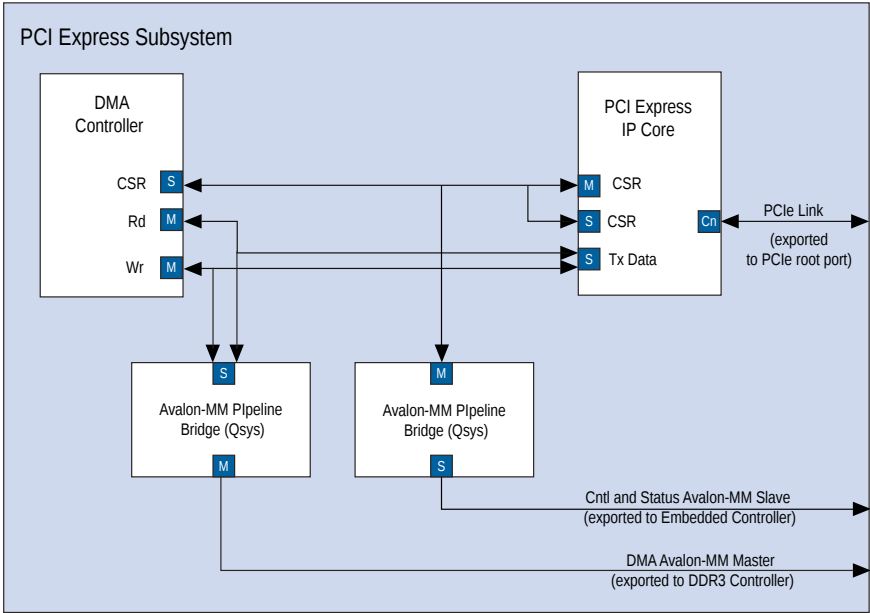
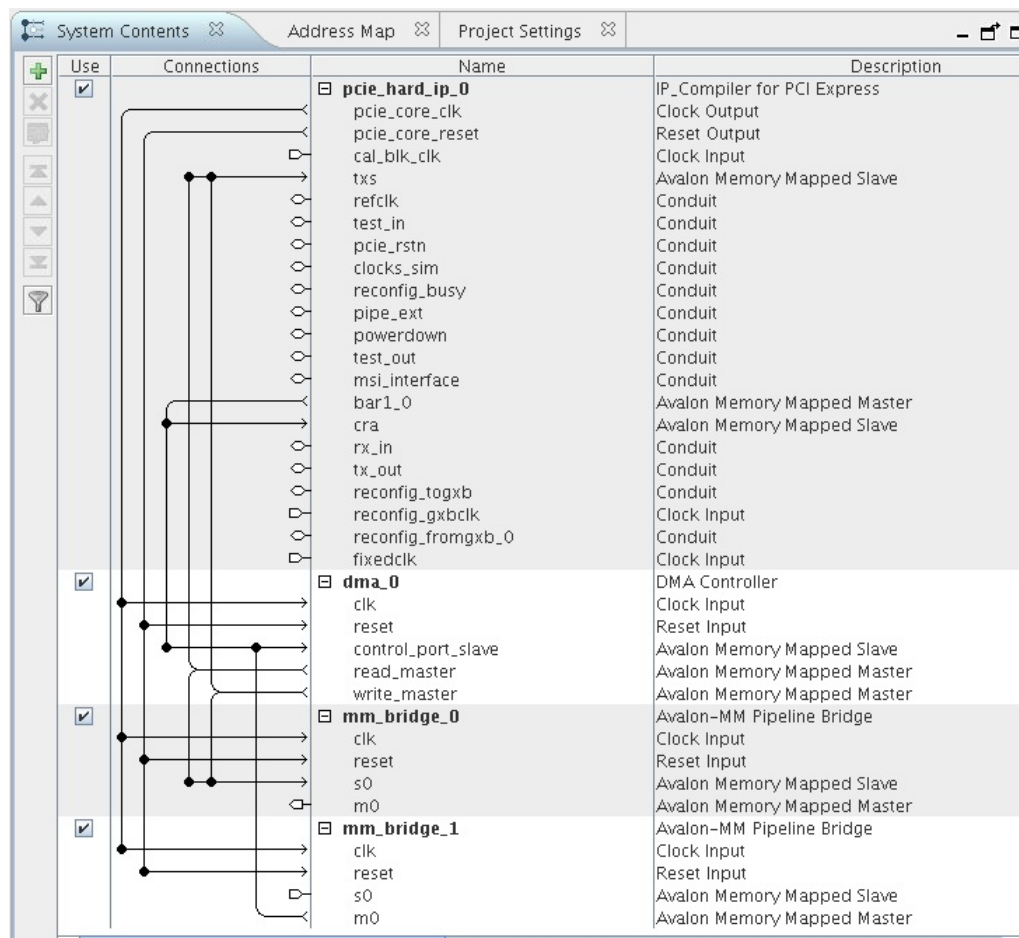


Figure 6-11: Qsys Representation of the PCI Express Subsystem

**Related Information**[Qsys Interconnect](#)**Ethernet Subsystem Example**

In this example subsystem, the transmit (TX) DMA receives data from the DDR3 memory and writes it to the Altera Triple-Speed Ethernet IP core using an Avalon-ST source interface. The receive (RX) DMA accepts data from the Triple-Speed Ethernet IP core on its Avalon-ST sink interface and writes it to DDR3 memory.

The read and write masters of both Scatter-Gather DMA controllers and the Triple-Speed Ethernet IP core connect to the DDR3 memory through an Avalon-MM pipeline bridge. This Ethernet example subsystem exports all three control and status interfaces through an Avalon-MM pipeline bridge, which connects to a controller outside of the Qsys system, as shown in [Figure 6-12](#) and [Figure 6-13](#).

Figure 6-12: Scatter-Gather DMA-to-Ethernet Subsystem

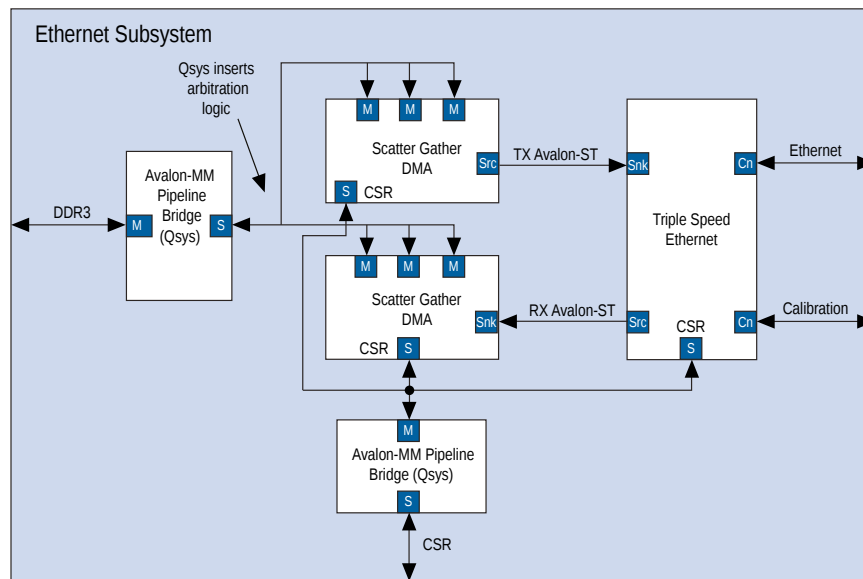
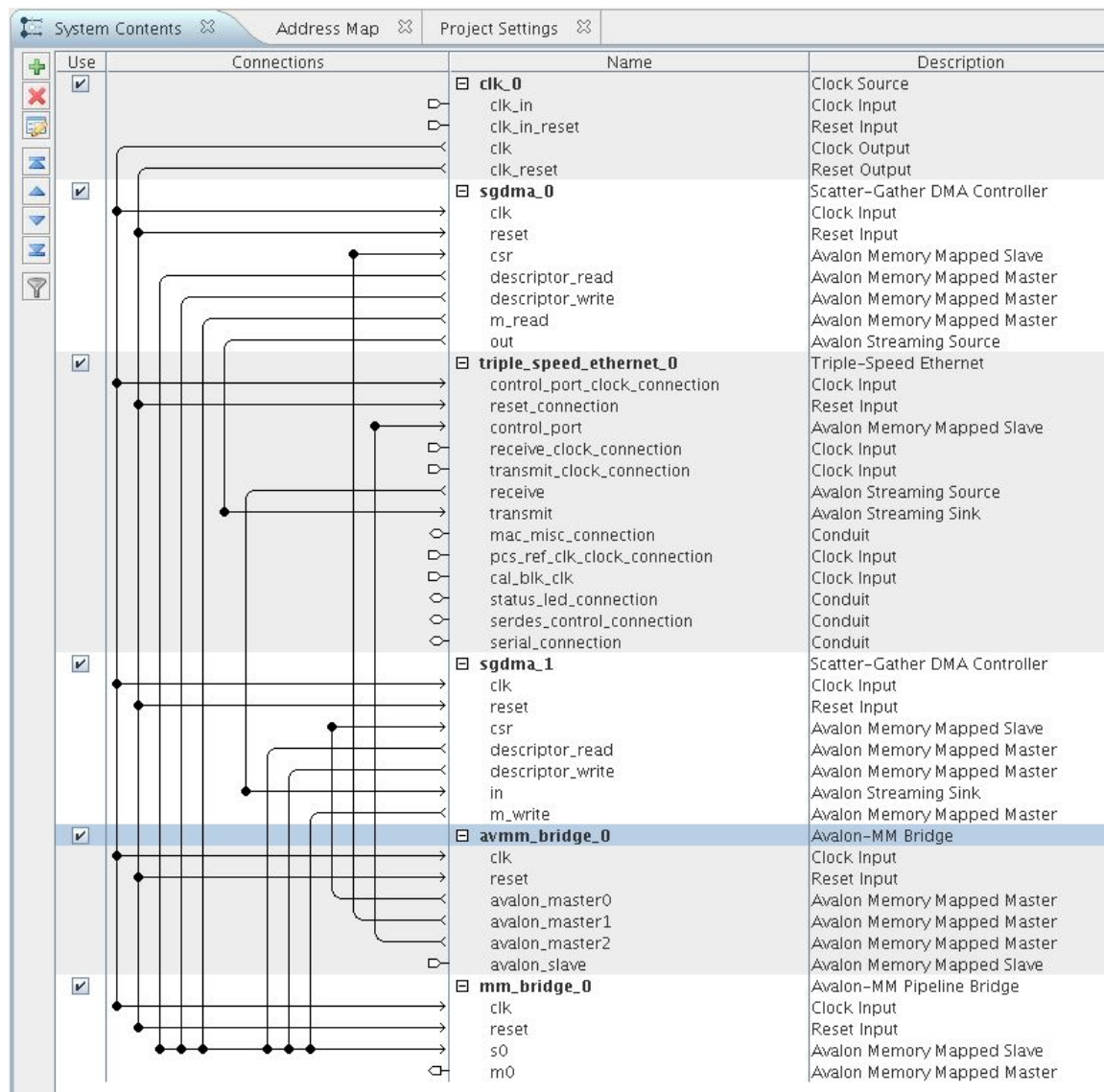


Figure 6-13: Qsys Representation of the Ethernet Subsystem



PCI Express to Ethernet Bridge Example

The PCI Express and Ethernet example subsystems run at 125 MHz and includes two clock domains and an ethernet subsystem. The DDR3 SDRAM controller runs at 200 MHz. Qsys automatically inserts clock crossing logic to synchronize the DDR3 SDRAM Controller with the PCI Express and Ethernet subsystems, as shown in [Figure 6-14](#) and [Figure 6-15](#).

Figure 6-14: PCI Express-to-Ethernet Bridge Example System

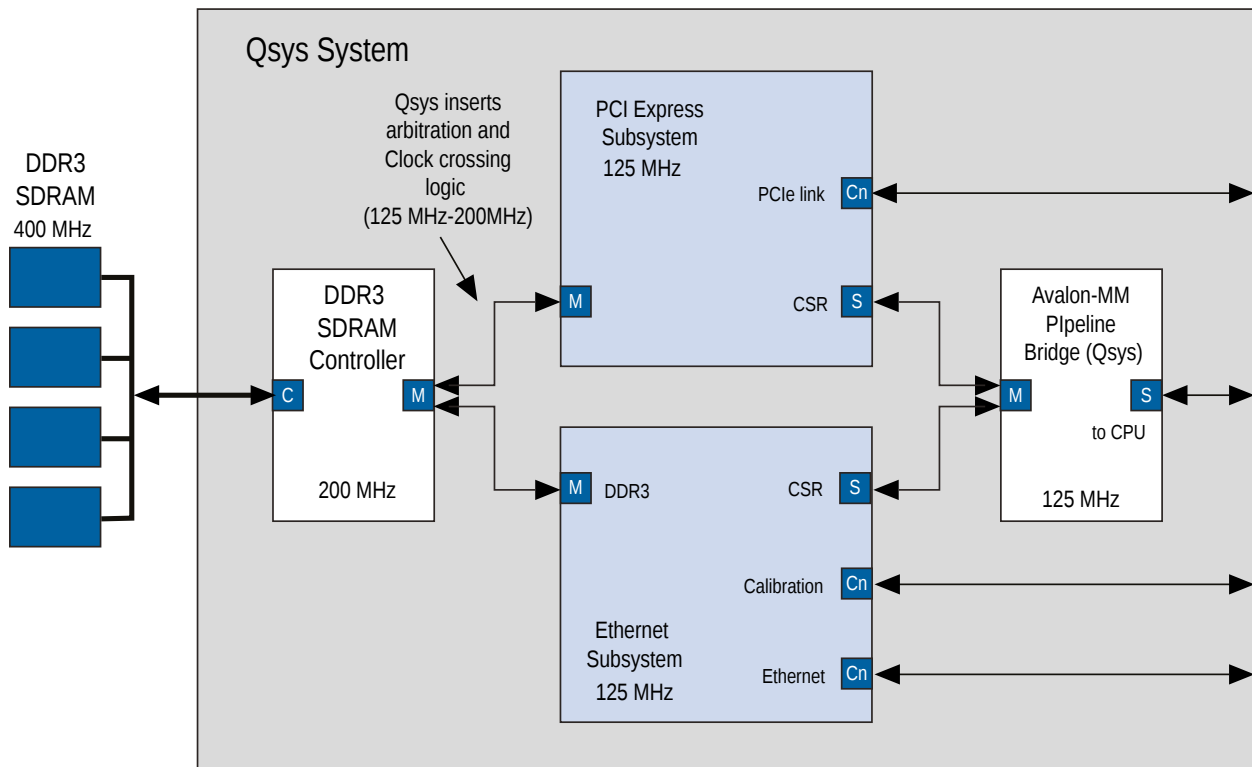


Figure 6-15: Qsys Representation of the Complete PCI Express to Ethernet Bridge

System Contents		Address Map	Project Settings		
Use	Connections	Name	Description		
<input checked="" type="checkbox"/>		uniphy_dds3_0	uniphy_dds3		
		avalon_slave_0	Avalon Memory Mapped Slave		
		memory_phy	Conduit		
		Other	Conduit		
		PLL_Sharing	Conduit		
<input checked="" type="checkbox"/>		ethernet_dma_subsystem_0	ethernet_dma_subsystem		
		cal_blk_if	Conduit		
		dma_if	Avalon Memory Mapped Master		
		csr_if	Avalon Memory Mapped Slave		
		ethernet_if	Conduit		
<input checked="" type="checkbox"/>		pcie_subsystem_0	pcie_subsystem		
		pcie_link	Conduit		
		ddr3_sdram_master	Avalon Memory Mapped Master		
		dma_control_slave	Avalon Memory Mapped Slave		
<input checked="" type="checkbox"/>		mm_bridge_0	Avalon-MM Pipeline Bridge		
		clk	Clock Input		
		reset	Reset Input		
		s0	Avalon Memory Mapped Slave		
		m0	Avalon Memory Mapped Master		

Pipeline Bridges

The PCI Express to Ethernet bridge example system uses several pipeline bridges. You must configure bridges to accommodate the address range of all of connected components, including the components in the originating subsystem and the components in the next higher level of the system hierarchy.

The pipeline bridge inserts a pipeline stage between the connected components. You should register signals at the subsystem interface level for the following reasons:

- Registering interface signals decreases the amount of combinational logic that must be completed in one cycle, making it easier to meet timing constraints.
- Registering interface signals raises the potential frequency, or f_{MAX} , of your design at the expense of an additional cycle of latency, which might adversely affect system throughput.
- The Quartus II incremental compilation feature can achieve better f_{MAX} results if the subsystem boundary is registered.

Note: Connections between AXI and Avalon interfaces are made without requiring the use of explicitly instantiated bridges; the interconnect provides the necessary bridging logic.

Related Information

- [Optimizing System Performance for Qsys](#)
- [Qsys System Design Components](#)

Hierarchical System Using Instance Parameters Example

You can use an instance parameter to control the implementation of system components from a higher-level Qsys system. You define instance parameters on the **Instance Parameters** tab in Qsys.

In [Example 6-3](#), the **my_system.qsys** system has two instances of the same IP component, **My_IP**. **My_IP** is a Qsyscomponent with a system identification parameter called **MY_SYSTEM_ID**. When **my_system.qsys** is instantiated within another higher-level Qsys system, the two **My_IP** subcomponents require different values for their **MY_SYSTEM_ID** parameters based on a value determined by the higher-level system. In this example, the value specified by the top-level system is designated **top_id** and in **my_system.qsys**, the component instance **comp0** requires **MY_SYSTEM_ID** set to **top_id + 1**, and instance **comp1** requires **MY_SYSTEM_ID** set to **top_id + 2**. [Example 6-3](#) defines the **MY_SYSTEM_ID** system ID parameter in the IP component **My_IP**:

Example 6-3: System ID Parameter Example

```
add_parameter MY_SYSTEM_ID int 8

set_parameter_property MY_SYSTEM_ID DISPLAY_NAME \
MY_SYSTEM_ID_PARAM

set_parameter_property MY_SYSTEM_ID UNITS None
```

To satisfy the design requirements for this example, you define an instance parameter in **my_system.qsys** that is set by the higher-level system, and then define an instance script to specify how the values of the parameters of the **My_IP** components instantiated in **my_system.qsys** are affected by the value set on the instance parameter.

To do this, in Qsys, open the **my_system.qsys** Qsys system that instantiates the two instances of the My_IP components. On the **Instance Parameters** tab, create a parameter called `system_id`. For this example, you can set this parameter to be of type Integer and choose **0** as the default value.

Next, you provide a **Tcl Instance Script** that defines how the value of the `system_id` parameter should affect the parameters of `comp0` and `comp1` subcomponents in **my_system.qsys**.

In **Example 6-4** Qsys gets the value of the parameter `system_id` from the top-level system and saves it as `top_id`, and then increments the value by 1 and 2. The script then uses the new calculated values to set the `MY_SYSTEM_ID` parameter in the My_IP component for the instances `comp0` and `comp1`. The script uses informational messages to print the status of the parameter settings when the **my_system.qsys** system is added to the higher-level system.

Example 6-4: Tcl Instance Script Example

```
package require qsys 13.1
set_module_property Composition_callback My_callback
proc My_callback { } {
    # Get The Value Of system_id parameter from the
    # higher-level system
    set top_id [get_parameter_value system_id]

    # Print Info Message
    send_message Info "system_id Value Specified: $top_id"

    # Use Above Value To Set Parameter Values For The Subcomponents

    set child_id_0 [expr {$top_id + 1} ]
    set child_id_1 [expr {$top_id + 2} ]

    # Set The Parameter Values On The Subcomponent Instances
    set_instance_parameter_value comp0 My_system_id $child_id_0
    set_instance_parameter_value comp1 My_system_id $child_id_1

    # Print Info Messages
    send_message Info "system_id Value Used In comp0: $child_id_0"
    send_message Info "system_id Value Used In comp1: $child_id_1"
}
```

You can click **Preview Instance** to modify the parameter value interactively and see the effect of the scripts in the message panel which can be useful for debugging the script. In this example, if you change the parameter value in the **Preview** screen, the component generates messages to report the top-level ID parameter value and the parameter values used for the two instances of the component.

Related Information

[Working with Instance Parameters in Qsys](#)

Searching for Component Files to Add to the Library

The Qsys Library lists design components available for use in Qsys systems. Components can include Altera-provided IP cores, third-party IP cores, and custom IP cores that you provide. Previously created

Qsys systems can also appear in the library, and you can use these systems in other designs if they have exported interfaces.

Altera and third-party developers provide ready-to-use components, which are installed automatically with the Quartus II software and are available in the Qsys Library. The Qsys Library includes the following components:

- Microprocessors, such as the Nios[®] II processor
- DSP IP cores, such as the Reed Solomon II core
- Interface protocols, such as the IP Compiler for PCI Express
- Memory controllers, such as the RLD RAM II Controller with UniPHY
- Avalon[®] Streaming (Avalon-ST) components, such as the Avalon-ST Multiplexer IP core
- Qsys Interconnect components
- Verification IP (VIP) Bus Functional Models (BFMs)

You can set the **IP Search Path** option to specify the installed locations for custom and third-party components that you want to appear in the component library. Qsys searches for component files each time you open the tool, and locates and displays the list of available components in the component library.

Qsys searches the directories listed in the **IP Search Path** for the following component file types:

- Hardware Component Description File (**_hw.tcl**)—Each **_hw.tcl** file defines a single component.
- IP Index File (**.ipx**)—Each **.ipx** file indexes a collection of available components, or a reference to other directories to search. In general, **.ipx** files facilitate faster startup for Qsys and other tools because fewer directories are searched and analyzed.

Qsys searches some directories recursively and other directories only to a specific depth. When a directory is recursively searched, the search stops at any directory containing an **_hw.tcl** or **.ipx** file; subdirectories are not searched. In the following list of search locations, a recursive descent is annotated by **. A single * signifies any file.

Note: If you add a component to your search path, you must refresh your system by clicking **File > Refresh** to update the Qsys library.

- **PROJECT_DIR/***—Finds components and index files in the Quartus II project directory.
- **PROJECT_DIR/ip/**/***—Finds components and index files in any subdirectory of the **/ip** subdirectory of the Quartus project directory.
- **QUARTUS_INSTALLDIR/./ip/**/***—In this IP directory, you can create your own subdirectories that are available for any project using this Quartus II installation directory.

Adding Components to the Library

You can use one of the following methods to add components to the library.

- Save components in your project directory.
- Save components in the **/ip** subdirectory of your project directory.
- Copy components to the install directory.
- Reference components in an IP Index File (**.ipx**).
- Integrate third-party components.

[Copy Components to a Directory Searched by Default](#) on page 6-41

[Reference Components in an IP Index File \(.ipx\)](#) on page 6-42

[Extending the Default Search Path](#) on page 6-44

Copy Components to a Directory Searched by Default

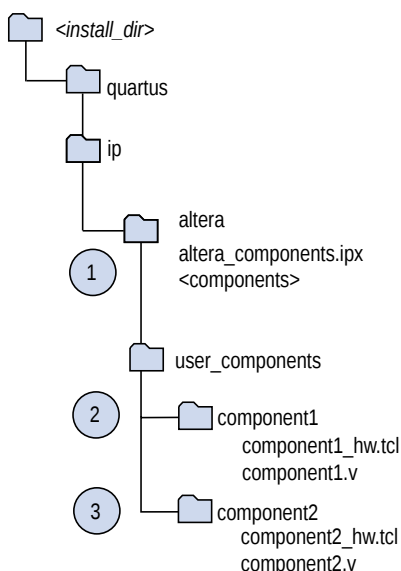
The simplest method to add a new component to the Qsys Library is to copy your components into one of the directories Qsys searches by default. You can save component files in your project directory, or in the **/ip** subdirectory of your project directory. These approaches are useful if you want to associate your components with a specific Quartus II project.

If you save the component in the project directory, the component appears in the Library in the group you specified under **Project**. Alternatively, if you save the component in the Quartus II installation directory, the component appears in the specified group under **Library**.

You can also save the component files into the default Quartus II `<install_dir>/ip/` directory. This approach is useful in the following situations and is shown in [Figure 6-16](#).

- You want to associate your components with a specific release of the Quartus II software.
- You want to have the same components available across multiple projects.

Figure 6-16: User Library Included In Subdirectory `<install_dir>/ip/`



In [Figure 6-16](#), the circled numbers identify a typical directory structure for the Quartus II software. For the directory structure above, Qsys performs the component discovery algorithm described below to locate **.ipx** and **_hw.tcl** files.

1. Qsys recursively searches the `<install_dir>/ip/` directory by default. The recursive search stops when Qsys finds an **.ipx** file.
2. As part of the recursive search, Qsys also looks in the **user_components** directory. Qsys finds the **component1** directory, which contains **component1_hw.tcl**. When Qsys finds the **component1_hw.tcl** component, the recursive search ends, and no components in subdirectories of **component1** are found.
3. Qsys then searches the **component2** directory, because this directory path also appears as an **IP Search Path**, and discovers **component2_hw.tcl**. When Qsys finds **component2_hw.tcl**, the recursive search ends.

Note: If you save your `_hw.tcl` file in the `<install_dir>/ip/` directory, Qsys finds your `_hw.tcl` file and does not search subdirectories adjacent to the `_hw.tcl` file.

Reference Components in an IP Index File (.ipx)

You can specify the search path in a `user_components.ipx` file under the `<install_dir>/ip` directory. This method allows you to add a location that is independent of the default search path. You can also save the `.ipx` file in any of the default search locations, for example, the Quartus II project directory, or the `/ip` directory in the project directory. The `user_components.ipx` file includes a single line of code redirecting Qsys to the location of each user library. The path below shows a redirection example:

```
<library> <path path="<user_lib_dir>/user_ip/**/*"/> </library>
```

You can verify that components are available with the `ip-catalog` command. You can use the `ip-make-ipx` command to create an `.ipx` file for a directory tree, which can reduce the startup time for Qsys.

Understanding the IP Index File (.ipx) Syntax

An IP Index File (`.ipx`) is an XML file that describes the search path used to discover components that are available for a Qsys system. A `<path>` entry specifies a directory in which components may be found. A `<component>` entry specifies the path to a single component.

Example 6-5: .ipx File Structure

```
<library>
  <path path="...<user directory>" />
  <path path="...<user directory>" />
  ...
  <component ... file="...<user directory>" />
  ...
</library>
```

A `<path>` element contains a path attribute, which specifies the path to a directory, or the path to another `.ipx` file, and can use wildcards in its definition. An asterisk matches any file name. If you use an asterisk as a directory name, it matches any number of subdirectories.

When searching the specified path, the following three types of files are identified:

- `.ipx`—Additional index files.
- `_hw.tcl`—Qsys component definitions.
- `_sw.tcl`—Nios II board support package (BSP) software component definitions.

A `<component>` element contains several attributes to define a component. If you provide the required details for each component in an `.ipx` file, the startup time for Qsys is less than if Qsys must discover the files in a directory. [Example 6-6](#) shows two `<component>` elements. Note that the paths for file names are specified relative to the `.ipx` file.

Example 6-6: Component Element in an .ipx File

```
<library>
  <component
```

```
    name="A Qsys Component"
    displayName="Qsys FIR Filter Component"
    version="2.1"
    file="./components/qsys_filters/fir_hw.tcl"
  />
<component
  name="rgb2cmyk_component"
  displayName="RGB2CMYK Converter(Color Conversion Category!)"

  version="0.9"
  file="./components/qsys_converters/color/rgb2cmyk_hw.tcl"
/>
</library>
```

ip-catalog

The `ip-catalog` command displays the catalog of available components relative to the current project directory in either plain text or XML format.

Usage

```
ip-catalog [--project-dir=<directory>] [--name=<value>] [--verbose]
[--xml] [--help]
```

Options

- **--project-dir= <directory>**—Optional. Components are found in locations relative to the project, if any. By default, the current directory, `'.'` is used. To exclude a project directory, leave the value empty.
- **--name= <value>**—Optional. This argument provides a pattern to filter the names of the components found. To show all components, use a `*` or `'.'`. By default, all components are shown. The argument is not case sensitive.
- **--verbose**—Optional. If set, reports the progress of the command.
- **--xml**—Optional. If set, generates the output in XML format, instead of a line and colon-delimited format.
- **--help**—Shows help for the `ip-catalog` command.

ip-make-ipx

The `ip-make-ipx` command creates an `.ipx` file and is a convenient way to include a collection of components from an arbitrary directory in the Qsys search path. You can also edit the `.ipx` file to disable visibility of one or more components in the Qsys Library.

Usage

```
ip-make-ipx [--source-directory=<directory>] [--output=<file>]
[--relative-vars=<value>] [--thorough-descent] [--message-before=<value>]
[--message-after=<value>] [--help]
```

Options

- **--source-directory= <directory>**—Optional. Specifies the root director(ies) that Qsys uses to find the component files. The default directory is “.”. You can also provide a comma-separated list of directories.
- **--output= <file>**—Optional. Specifies the name of the file to generate. The default name is `/components.ipx`.
- **--relative-vars= <value>**—Optional. Causes the output file to include references relative to the specified variable(s) where possible. You can specify multiple variables as a comma-separated list.
- **--thorough-descent**—Optional. If set, a component or `.ipx` file in a directory does not prevent subdirectories from being searched.
- **--message-before= <value>**—Optional. A message to print to `stdout` when indexing begins.
- **--message-after= <value>**—Optional. A message to send to `stdout` when indexing completes.
- **--help**—Shows help for this command.

Extending the Default Search Path

The following steps allow you to extend the default search path by specifying additional directories.

1. In Qsys, in the Tools menu, click **Options**.
2. In the **Category** list, click **IP Search Path**.
3. Click **Add**.
4. Browse to locate additional directories and click **Open** to add them to your search path.

You do not need to include the components specified in the IP Search Path as part of your Quartus II project.

Integrating Third-Party Components

You can use Qsys components created by third-party IP developers. Altera awards the Qsys Compliant label to IP cores that are fully supported in Qsys. These cores have interfaces that are supported by Qsys, such as Avalon-MM or AXI, and may include timing and placement constraints, software drivers, simulation models, and reference designs.

To find supported third-party Qsys components on Altera's web page, navigate to the **Intellectual Property & Reference Designs** page, and then type `Qsys Certified` in the **Search** box, select **IP Core & Reference Designs**, and then press **Enter**.

Refer to Altera's Intellectual Property & Reference Designs page for more information.

Related Information

[Intellectual Property & Reference Designs](#)

Using Qsys Command-Line with Utilities and Scripts

You can perform many of the functions available in the Qsys GUI from the command-line with the `qsys-generate` and `qsys-script` utilities.

You run these command-line executables from the following Quartus II installation directory:

`<Quartus II installation directory>\quartus\sopc_builder\bin`

You can use `qsys-generate` to generate Qsys output files outside of the Qsys GUI. You can use `qsys-script` to create, manipulate or manage a Qsys system with command-line scripting.

For command-line help listing all options for these executables, type the following command:

<Quartus II installation directory>\quartus\sopc_builder\bin\<executable name> --help

Example 6-7: Qsys Command-Line Scripting Example

```
qsys-script --script=my_script.tcl \  
--system-file=fancy.qsys my_script.tcl contains:  
package require -exact qsys 13.1  
# get all instance names in the system and print one by one  
set instances [ get_instances ]  
foreach instance $instances {  
    send_message Info "$instance"  
}
```

Related Information

- [Working with Instance Parameters in Qsys](#)
- [Altera Wiki Qsys Scripts](#)

Running the Qsys Editor from the Command-Line

You can use the `qsys-edit` utility to run the Qsys Editor from the command-line.

The following is a list of options that you can use with the `qsys-edit` utility:

- **<1st arg file>**—Optional. The name of the **.qsys** system or **.qvar** variation file to edit.
- **--search-path[=<value>]**—Optional. If omitted, Qsys uses a standard default path. If provided, Qsys searches a comma-separated list of paths. To include the standard path in your replacement, use "\$", for example: `/extra/dir.$`.
- **--project-directory=<directory>**—Optional. Allows you to find components in certain locations relative to the project, if any. By default, the current directory is: ' . ' . To exclude any project directory, use ' ' .
- **--new-component-type=<value>**—Optional. Allows you to specify the kind of instance that is parameterized in a variation.
- **--debug**—Optional. Enables debugging features and output.
- **--host-controller**—Optional. Launches the application with an XML host controller interface on standard input/output.
- **--jvm-max-heap-size=<value>**—Optional. The maximum memory size Qsys uses for allocations when running `qsys-edit`. You specify this value as `<size><unit>`, where unit is m (or M) for multiples of megabytes, or g (or G) for multiples of gigabytes. The default value is 512m.
- **--help**—Optional. Display help for `qsys-edit`.

Launching Qsys with Additional Computer Memory

If the Qsys system you are creating requires more than the 512 megabytes of default memory, you may need to launch the Qsys GUI from the command-line with additional memory. For example, the following `qsys-edit` command allows you to launch Qsys from the command-line with 2 gigabytes of memory.

```
qsys-edit --jvm-max-heap-size=2g
```

Generating Qsys Systems with the qsys-generate Utility

You can use the `qsys-generate` utility to generate RTL for your Qsys system, simulation models and scripts, and to create testbench systems for testing your Qsys system in a simulator using BFM. Output from the `qsys-generate` command is the same as when generating using the Qsys GUI.

The following is a list of options that you can use with the `qsys-generate` utility:

- **<1st arg file>**—Required. The name of the `.qsys` system file to generate.
- **--synthesis=<VERILOG|VHDL>**—Optional. Creates synthesis HDL files that Qsys uses to compile the system in a Quartus II project. You must specify the preferred generation language for the top-level RTL file for the generated Qsys system.
- **--block-symbol-file**—Optional. Creates a block symbol file (`.bsf`) for the system.
- **--simulation=<VERILOG|VHDL>**—Optional. Creates a simulation model for the system. The simulation model contains generated HDL files for the simulator, and may include simulation-only features. You must specify the preferred simulation language.
- **--testbench=<SIMPLE|STANDARD>**—Optional. Creates a testbench system. The testbench system instantiates the original system, adding bus functional models to drive the top-level interfaces. Once generated, the bus functional models interact with the system in the simulator.
- **--testbench-simulation=<VERILOG|VHDL>**—Optional. After creating the testbench system, also create a simulation model for the testbench system.
- **--output-directory=<value>**—Optional. Sets the output directory. Each generation target is created in a subdirectory of the output directory. If you do not specify the output directory, a subdirectory of the current working directory matching the name of the system is used.
- **--search-path=<value>**—Optional. If omitted, a standard default path is used. If provided, a comma-separated list of paths is searched. To include the standard path in your replacement, use "\$", for example, `"/extra/dir,$"`.
- **--jvm-max-heap-size=<value>**—Optional. The maximum memory size that Qsys uses for allocations when running this tool. The value is specified as `<size><unit>` where unit can be m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512m.
- **--family=<value>**—Optional. Sets the device family.
- **--part=<value>**—Optional. Sets the device part number. If set, this option overrides the `--family` option.
- **--allow-mixed-language-simulation**—Optional. Enables a mixed language simulation model generation. If true, if a preferred simulation language is set, Qsys uses a fileset of the component for the simulation model generation. When false, which is the default, Qsys uses the language specified with `--file-set=<value>` for all components for simulation model generation.
- **--file-set=<value>**—Optional. Allows you to choose the type output to generate, for example, `QUARTUS_SYNTH`, `SIM_VERLOG`, or `VHDL`.

Creating and Managing a System with qsys-script

You can use the `qsys-script` tool to create and manipulate a Qsys system with Tcl scripting commands.

Note: You must provide a package version for the `qsys-script`. If you do not specify the `--package-version=<value>` `qsys-script` command, you must then provide a Tcl script and request the system scripting API directly with the package `require -exact qsys <version>` command.

The following is a list of options that you can use with the `qsys-script` utility:

- **--system-file=<file>**—Optional. Specifies the path to a `.qsys` system file. This system is loaded before running scripting commands.
- **--script=<file>**—Optional. A file containing Tcl scripting commands for creating or manipulating Qsys systems. If you specify both `--cmd` and `--script`, the `--cmd` commands are run before the script specified by `--script`.
- **--cmd=<value>**—Optional. A string that contains Tcl scripting commands to create or manipulate a Qsys system. If you specify both `--cmd` and `--script`, the `--cmd` commands are run before the script specified by `--script`.
- **--package-version=<value>**—Optional. Specifies which system scripting Tcl API version to use and determines the functionality and behavior of the Tcl commands. The Quartus II software supports the Tcl API scripting commands. If you do not specify the version on the command-line, your Tcl script must request the system scripting API directly with the `package require -exact qsys <version>` command.
- **--help**—Optional. Displays help for the `qsys-script` tool.
- **--search-path=<value>**—Optional. If omitted, a standard default path is used. If provided, a comma-separated list of paths is searched. To include the standard path in your replacement, use "\$", for example, `/<directory path>/dir, $`. Multiple directory references are separated with a comma.
- **--jvm-max-heap-size=<value>**—Optional. The maximum memory size that is used by the `qsys-script` tool. You specify this value as `<size><unit>` where unit can be `m` or `M` for multiples of megabytes or `g` or `G` for multiples of gigabytes.

Qsys Scripting Command Reference

Interface properties work differently for qsys scripting than with `_hw.tcl` scripting. In `_hw.tcl`, interfaces do not distinguish between properties and parameters; in qsys scripting, properties and parameters are unique.

`add_connection <start> [<end>]` on page 6-52

This command connects interfaces using an appropriate connection type. Interface names consist of a child instance name, followed by the name of an interface provided by that module, for example, `mux0.out` is the interface `out` on the instance named `mux0`.

`add_instance <name> <type> [<version>]` on page 6-52

This command adds an instance of a component, referred to as a child or child instance, to the system.

`add_interface <name> <type> <direction>` on page 6-53

This command adds an interface to your system, which you can use to export an interface from within the system. You specify the exported interface with the command `set_interface_property EXPORT_OF <instance.interface>`.

`auto_assign_base_addresses <instance>` on page 6-53

This command assigns base addresses to memory-mapped interfaces on an instance in the system. Instance interfaces that are locked with `lock_avalon_base_address` command keep their addresses during address auto-assignment.

`auto_assign_irqs <instance>` on page 6-53

This command assigns interrupt numbers to all connected interrupt senders on an instance in the system.

auto_connect *<element>* on page 6-54

This command creates connections from an instance or instance interface to matching interfaces in other instances in the system. For example, Avalon-MM slaves are connected to Avalon-MM masters.

create_system [*<name>*] on page 6-54

This command replaces the current system in the system script with a new system with the specified name.

get_instance_interface_parameter_value *<instance>* *<interface>* *<parameter>* on page 6-54

This command returns the value of a parameter of an interface in a child instance.

get_composed_connection_parameters *<instance>* *<childConnection>* on page 6-55

This command returns a list of parameters on a connection in the subsystem, for an instance that contains a subsystem.

get_composed_connections *<instance>* on page 6-55

This command returns a list of all connections in a subsystem, for an instance that contains a subsystem.

get_composed_instance_assignment *<instance>* *<childInstance>* *<key>* on page 6-55

This command returns the value of an assignment on an instance of a subsystem, for an instance that models a subsystem.

get_composed_instance_assignments *<instance>* *<childInstance>* on page 6-56

This command returns a list of assignments on an instance of a subsystem, for an instance that contains a subsystem.

get_composed_instance_parameter_value *<instance>* *<childInstance>* *<parameter>* on page 6-56

This command returns the value of a parameters on an instance in a subsystem, for an instance that contains a subsystem.

get_composed_instance_parameters *<instance>* *<childInstance>* on page 6-57

This command returns a list of parameters on an instance of a subsystem, for an instance that contains a subsystem.

get_composed_instances *<instance>* on page 6-57

This command returns a list of child instances in the subsystem, for an instance that contains a subsystem.

get_connection_parameter_property *<connection>* *<parameter>* *<property>* on page 6-57

This command returns the value of a parameter property in a connection.

get_connection_parameter_value *<connection>* *<parameter>* on page 6-58

This command gets the value of a parameter on the connection. Parameters represent aspects of the connection that can be modified once the connection is created, such as the base address for an Avalon-MM connection.

get_connection_parameters *<connection>* on page 6-58

This command returns a list of parameters found on a connection. The list of connection parameters is the same for all connections of the same type.

get_connection_properties on page 6-58

This command returns a list of properties found on a connection. The list of connection properties is the same for all connections, regardless of type.

get_connection_property *<connection>* *<property>* on page 6-59

This command returns the value of a connection property.

get_connections [*<element>*] on page 6-59

This command returns a list of connections in the system if no element is specified. If a child instance is specified, for example `cpu`, all connections to any interface on the instance are returned. If an interface on a child instance is specified, for example `cpu.instruction_master`, only connections to that interface are returned.

get_instance_assignment *<instance>* *<key>* on page 6-59

This command returns the value of an assignment on a child instance.

get_instance_assignments *<instance>* on page 6-60

This command returns a list of assignment keys for any assignments defined for the instance.

get_instance_interface_assignment *<instance>* *<interface>* *<key>* on page 6-60

This command returns the value of an assignment on an interface of a child instance.

get_instance_interface_assignments *<instance>* *<interface>* on page 6-60

This command returns the value of an assignment on an interface of a child instance.

get_instance_interface_parameter_property *<instance>* *<interface>* *<parameter>* *<property>* on page 6-61

This command returns the property value on a parameter in an interface of a child instance.

get_instance_interface_parameter_value *<instance>* *<interface>* *<parameter>* on page 6-61

This command returns the value of a parameter of an interface in a child instance.

get_instance_interface_parameters *<instance>* *<interface>* on page 6-62

This command returns a list of parameters for an interface in a child instance.

get_instance_interface_port_property *<instance>* *<interface>* *<port>* *<property>* on page 6-62

This command returns the property value of a port in the interface of a child instance.

get_instance_interface_ports *<instance>* *<interface>* on page 6-62

This command returns a list of ports in an interface of a child instance.

get_instance_interface_properties on page 6-63

This command returns a list of properties that you can be query for an interface in a child instance.

get_instance_interface_property *<instance>* *<interface>* *<property>* on page 6-63

This command returns the property value for an interface in a child instance.

get_instance_interfaces *<instance>* on page 6-63

This command returns a list of interfaces in a child instance.

get_instance_parameter_property *<instance>* *<parameter>* *<property>* on page 6-64

This command returns the value of a parameter in a connection in the subsystem, for an instance that contains a subsystem.

get_instance_parameter_value *<instance>* *<parameter>* on page 6-64

This command returns the value of a property in a child instance.

get_instance_parameters *<instance>* on page 6-64

This command returns a list of parameters in a child instance.

get_instance_port_property *<instance>* *<port>* *<property>* on page 6-65

This command returns the value of a property of a port contained by an interface in a child instance.

get_instance_properties on page 6-65

This command returns a list of properties for a child instance.

get_instance_property *<instance>* *<property>* on page 6-65

This command returns the value of a property for a child instance.

get_instances on page 6-66

This command returns a list of the instance names for all child instances in the system.

get_interface_port_property *<interface>* *<port>* *<property>* on page 6-66

This command returns the value of a property of a port contained by an interface in a child instance.

get_interface_ports *<interface>* on page 6-66

This command returns the names of all of the ports that have been added to an interface.

get_interface_properties on page 6-67

This command returns the names of all the available interface properties. The list of interface properties is the same for all interface types.

get_interface_property *<interface>* *<property>* on page 6-67

This command returns the value of a property from the specified interface.

get_interfaces on page 6-67

This command returns a list of top-level interfaces in the system.

get_module_properties on page 6-68

This command returns the properties that you can manage for the top-level module.

get_module_property *<property>* on page 6-68

This command returns the value of a top-level system property.

get_parameter_properties on page 6-68

This command returns a list of properties that you can query on parameters. These properties can be queried on any parameter, such as parameters on instances, interfaces, instance interfaces, and connections.

get_port_properties on page 6-68

This command returns a list of properties that you can query on ports.

get_project_properties on page 6-69

This command returns a list of properties that you can query for the Quartus II project.

get_project_property *<property>* on page 6-69

This command returns the value of a Quartus II project property.

load_system *<file>* on page 6-69

This command loads a Qsys system from a file, and uses the system as the current system for scripting commands.

lock_avalon_base_address *<instance.interface>* on page 6-69

This command prevents the memory-mapped base address from being changed for connections to an interface on an instance when the `auto_assign_base_addresses` or `auto_assign_system_base_addresses` commands are run.

preview_insert_avalon_streaming_adapters on page 6-70

This command runs the adapter insertion for Avalon-ST connections, which adapt connections with mismatched configuration, such as mismatched data widths.

remove_connection *<connection>* on page 6-70

This command removes a connection from the system.

remove_instance <instance> on page 6-70

This command removes a child instance from the system.

remove_interface <interface> on page 6-70

This command removes an exported top-level interface from the system.

save_system [<file>] on page 6-71

This command saves the current in-memory system to the named file. If the file is not specified, the system saves to the same file that was opened with the `load_system` command.

send_message <level> <message> on page 6-71

This command sends a message to the user of the script. The message text is normally interpreted as HTML. You can use the `` element to provide emphasis.

set_connection_parameter_value <connection> <parameter> <value> on page 6-72

This command sets the parameter value for a connection.

set_instance_parameter_value <instance> <parameter> <value> on page 6-72

This command set the parameter value for a child instance. Derived parameters and `SYSTEM_INFO` parameters for the child instance can not be set with this command.

set_instance_property <instance> <property> <value> on page 6-73

This command sets the property value of a child instance. Most instance properties are read-only and can only be set by the instance itself. The primary use for this command is to update the `ENABLED` parameter, which includes or excludes a child instance when generating the system.

set_interface_property <interface> <property> <value> on page 6-73

This command sets the property value on an exported top-level interface. This command is used to set the `EXPORT_OF` property to specify which interface of a child instance is exported by the top-level interface.

set_module_property <property> <value> on page 6-73

This command sets the system property value, such as the name of the system using the `NAME` property.

set_project_property <property> <value> on page 6-74

This command sets the project property value, such as the device family.

set_validation_property <property> <value> on page 6-74

This command sets a property that affects how and when validation is run during system scripting. To disable system validation after each scripting command, set `AUTOMATIC_VALIDATION` to false.

unlock_avalon_base_address <instance.interface> on page 6-74

This command allows the memory-mapped base address to be changed for connections to an interface on an instance when the `auto_assign_base_addresses` or `auto_assign_system_base_addresses` commands are run.

upgrade_sopc_system <filename> on page 6-75

This command loads the specified `.sopc` file, which then upgrades the file as a Qsys-compatible system. Some child instances and interconnect are replaced so that the system functions in Qsys. You must save the new Qsys-compatible system with the `save_system` command.

validate_connection <connection> on page 6-75

This command validates the specified connection, and returns the during validation messages.

validate_instance <instance> on page 6-75

This command validates the specified child instance, and returns the validation messages.

validate_instance_interface <instance> <interface> on page 6-76

This command validates an interface on a child instance, and returns the validation messages.

validate_system on page 6-76

This command validates the system, and returns the validation messages.

add_connection <start> [<end>]

This command connects interfaces using an appropriate connection type. Interface names consist of a child instance name, followed by the name of an interface provided by that module, for example, mux0.out is the interface out on the instance named mux0.

add_connection		
Usage	add_connection <start> [<end>]	
Returns	None	
Arguments	start	The start interface to be connected, in <instance_name>.<interface_name> format.
	end	The end interface to be connected <instance_name>.<interface_name>. format.
Example	add_connection dma.read_master sdram.s1	

add_instance <name> <type> [<version>]

This command adds an instance of a component, referred to as a child or child instance, to the system.

add_instance		
Usage	add_instance<name> <type> [<version>]	
Returns	None	
Arguments	name	Specifies a unique local name that you can use to manipulate the instance. This name is used in the generated HDL to identify the instance.
	type	The type refers to a kind of instance available in a library, for example altera_avalon_uart.
	version (optional)	The required version of the specified instance type. If no version is specified, the latest version is used.
Example	add_instance uart_0 altera_avalon_uart	

add_interface <name> <type> <direction>

This command adds an interface to your system, which you can use to export an interface from within the system. You specify the exported interface with the command `set_interface_property EXPORT_OF <instance.interface>`.

add_interface		
Usage	add_interface <name> <type> <direction>	
Returns	None	
Arguments	name	The name of the interface that will be exported from the system.
	type	The type of interface.
	direction	The interface direction.
Example	<pre>add_interface my_export conduit end add_interface my_export conduit end set_interface_property my_export EXPORT_OF uart_0.external_connection</pre>	

auto_assign_base_addresses <instance>

This command assigns base addresses to memory-mapped interfaces on an instance in the system. Instance interfaces that are locked with `lock_avalon_base_address` command keep their addresses during address auto-assignment.

auto_assign_base_addresses		
Usage	auto_assign_base_addresses <instance>	
Returns	None	
Arguments	instance	The name of the instance with memory mapped interfaces.
Example	auto_assign_base_addresses sdram	

auto_assign_irqs <instance>

This command assigns interrupt numbers to all connected interrupt senders on an instance in the system.

auto_assign_irqs		
Usage	auto_assign_irqs <instance>	
Returns	None	
Arguments	instance	The name of the instance with an interrupt sender.
Example	auto_assign_irqs sdram	

auto_connect <element>

This command creates connections from an instance or instance interface to matching interfaces in other instances in the system. For example, Avalon-MM slaves are connected to Avalon-MM masters.

auto_connect		
Usage	auto_connect <element>	
Returns	None	
Arguments	element	The name of the instance interface, or the name of an instance.
Example	auto_connect sdram auto_connect uart_0.s1	

create_system [<name>]

This command replaces the current system in the system script with a new system with the specified name.

create_system		
Usage	create_system [<name>]	
Returns	None	
Arguments	name (optional)	The name of the new system.
Example	create_system my_new_system_name	

get_instance_interface_parameter_value <instance> <interface> <parameter>

This command returns the value of a parameter of an interface in a child instance.

get_composed_connection_parameter_value		
Usage	get_instance_interface_parameter_value <instance> <interface> <parameter>	
Return	various	The value of the parameter.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	parameter	The name of the parameter on the interface.
Example	get_instance_interface_parameter_value uart_0 s0 setupTime	

get_composed_connection_parameters <instance> <childConnection>

This command returns a list of parameters on a connection in the subsystem, for an instance that contains a subsystem.

get_composed_connection_parameters		
Usage	get_composed_connection_parameters <instance> <childConnection>	
Returns	string[]	A list of parameter names.
Arguments	instance	The child instance containing a subsystem.
	childConnection	The name of the connection in the subsystem.
Example	get_composed_connection_parameters subsystem_0 cpu.data_master/memory.s0	

get_composed_connections <instance>

This command returns a list of all connections in a subsystem, for an instance that contains a subsystem.

get_composed_connections		
Usage	get_composed_connections <instance>	
Returns	string[]	A list of connection names in the subsystem. These connection names are not qualified with the instance name.
Arguments	instance	The child instance containing a subsystem.
Example	get_composed_connections subsystem_0	

get_composed_instance_assignment <instance> <childInstance> <key>

This command returns the value of an assignment on an instance of a subsystem, for an instance that models a subsystem.

get_composed_instance_assignment		
Usage	get_composed_instance_assignment <instance> <childInstance> <key>	
Returns	string[]	The value of the assignment.

get_composed_instance_assignment		
Arguments	instance	The child instance containing a subsystem.
	childInstance	The name of a child instance found in the subsystem.
	key	The assignment key.
Example	get_composed_instance_assignment subsystem_0 video_0 "embeddedsw.CMacro.colorSpace"	

get_composed_instance_assignments <instance> <childInstance>

This command returns a list of assignments on an instance of a subsystem, for an instance that contains a subsystem.

get_composed_instance_assignments		
Usage	get_composed_instance_assignments <instance> <childInstance>	
Returns	string[]	A list of assignment names.
Arguments	instance	The child instance containing a subsystem.
	childInstance	The name of a child instance found in the subsystem.
Example	get_composed_instance_assignments subsystem_0 cpu	

get_composed_instance_parameter_value <instance> <childInstance> <parameter>

This command returns the value of a parameters on an instance in a subsystem, for an instance that contains a subsystem.

get_composed_instance_parameter_value		
Usage	get_composed_instance_parameter_value <instance> <childInstance> <parameter>	
Returns	string []	The value of a parameter on an instance of a subsystem.
Arguments	instance	The child instance containing a subsystem.
	childInstance	The name of a child instance found in the subsystem.
	parameter	The name of the parameter to query on an instance of a subsystem.

get_composed_instance_parameter_value

Example	get_composed_instance_parameter_value subsystem_0 cpu DATA_WIDTH
---------	--

get_composed_instance_parameters <instance> <childInstance>

This command returns a list of parameters on an instance of a subsystem, for an instance that contains a subsystem.

get_composed_instance_parameters

Usage	get_composed_instance_parameters <instance> <childInstance>	
Returns	string []	A list of parameter names.
Arguments	instance	The child instance containing a subsystem.
	childInstance	The name of a child instance found in the subsystem.
Example	get_composed_instance_parameters subsystem_0 cpu	

get_composed_instances <instance>

This command returns a list of child instances in the subsystem, for an instance that contains a subsystem.

get_composed_instances

Usage	get_composed_instances <instance>	
Returns	string []	A list of instance names found in the subsystem.
Arguments	instance	The child instance containing a subsystem.
Example	get_composed_instances subsystem_0	

get_connection_parameter_property <connection> <parameter> <property>

This command returns the value of a parameter property in a connection.

get_connection_parameter_property

Usage	get_connection_parameter_property <connection> <parameter> <property>	
Returns	various	The value of the parameter property.

get_connection_parameter_property		
Arguments	connection	The connection to query.
	parameter	The name of the parameter.
	property	The property of the connection.
Example	get_connection_parameter_property cpu.data_master/ dma0.csr baseAddress UNITS	

get_connection_parameter_value <connection> <parameter>

This command gets the value of a parameter on the connection. Parameters represent aspects of the connection that can be modified once the connection is created, such as the base address for an Avalon-MM connection.

get_connection_parameter_value		
Usage	get_connection_parameter_value <connection> <parameter>	
Returns	various	The value of the parameter.
Arguments	connection	The connection to query.
	parameter	The name of the parameter.
Example	get_connection_parameter_value cpu.data_master/ dma0.csr baseAddress	

get_connection_parameters <connection>

This command returns a list of parameters found on a connection. The list of connection parameters is the same for all connections of the same type.

get_connection_parameters		
Usage	get_connection_parameters <connection>	
Returns	string []	A list of parameter names.
Arguments	connection	The connection to query.
Example	get_connection_parameters cpu.data_master/dma0.csr	

get_connection_properties

This command returns a list of properties found on a connection. The list of connection properties is the same for all connections, regardless of type.

get_connection_properties		
Usage	get_connection_properties	
Returns	string []	A list of connection properties.

get_connection_properties		
Arguments	None	
Example	get_connection_properties	

get_connection_property <connection> <property>

This command returns the value of a connection property.

get_connection_property		
Usage	get_connection_property <connection> <property>	
Returns	string[]	The value of a connection property.
Arguments	connection	The connection to query.
	property	The name of the connection property.
Example	get_connection_property cpu.data_master/dma0.csr TYPE	

get_connections [<element>]

This command returns a list of connections in the system if no element is specified. If a child instance is specified, for example `cpu`, all connections to any interface on the instance are returned. If an interface on a child instance is specified, for example `cpu.instruction_master`, only connections to that interface are returned.

get_connections		
Usage	get_connections [<element>]	
Returns	string[]	A list of connections.
Arguments	element (optional)	The name of a child instance, or the qualified name of an interface on a child instance.
Example	get_connections get_connections cpu get_connections cpu.instruction_master	

get_instance_assignment <instance> <key>

This command returns the value of an assignment on a child instance.

get_instance_assignment	
Usage	get_instance_assignment <instance> <key>

get_instance_assignment		
Returns	string[]	The value of the specified assignment.
Arguments	instance	The name of the child instance.
	key	The assignment key to query.
Example	get_instance_assignment video_processor embeddedsw.CMacro.colorSpace	

get_instance_assignments <instance>

This command returns a list of assignment keys for any assignments defined for the instance.

get_instance_assignments		
Usage	get_instance_assignments <instance>	
Returns	string[]	A list of assignment keys.
Arguments	instance	The name of the child instance.
Example	get_instance_assignments sdram	

get_instance_interface_assignment <instance> <interface> <key>

This command returns the value of an assignment on an interface of a child instance.

get_instance_interface_assignment		
Usage	get_instance_interface_assignment <instance> <interface> <key>	
Returns	string []	The value of the specified assignment.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	key	The assignment key to query.
Example	get_instance_interface_assignment sdram s1 embeddedsw.configuration.isFlash	

get_instance_interface_assignments <instance> <interface>

This command returns the value of an assignment on an interface of a child instance.

get_instance_interface_assignments	
Usage	get_instance_interface_assignments <instance> <interface>

get_instance_interface_assignments		
Returns	string[]	A list of assignment keys.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
Example	get_instance_interface_assignments sdram s1	

get_instance_interface_parameter_property <instance> <interface> <parameter> <property>

This command returns the property value on a parameter in an interface of a child instance.

get_instance_interface_parameter_property		
Usage	get_instance_interface_parameter_property <instance> <interface> <parameter> <property>	
Returns	various	The value of the parameter property.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	parameter	The name of the parameter on the interface.
	property	The name of the property on the parameter.
Example	get_instance_interface_parameter_property uart_0 s0 setupTime ENABLED	

get_instance_interface_parameter_value <instance> <interface> <parameter>

This command returns the value of a parameter of an interface in a child instance.

get_composed_connection_parameter_value		
Usage	get_instance_interface_parameter_value <instance> <interface> <parameter>	
Return	various	The value of the parameter.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	parameter	The name of the parameter on the instance.

get_composed_connection_parameter_value

Example	get_instance_interface_parameter_value uart_0 s0 setupTime
---------	---

get_instance_interface_parameters <instance> <interface>

This command returns a list of parameters for an interface in a child instance.

get_instance_interface_parameters

Usage	get_instance_interface_parameters <instance> <interface>	
Returns	string[]	A list of parameter names for parameters in the interface.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
Example	get_instance_interface_parameters uart_0 s0	

get_instance_interface_port_property <instance> <interface> <port> <property>

This command returns the property value of a port in the interface of a child instance.

get_instance_interface_port_property

Usage	get_instance_interface_port_property <instance> <interface> <port> <property>	
Returns	various	The value of the port property.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	port	The name of the port in the interface.
	property	The name of the property of the port.
Example	get_instance_interface_port_property uart_0 exports tx WIDTH	

get_instance_interface_ports <instance> <interface>

This command returns a list of ports in an interface of a child instance.

get_instance_interface_ports		
Usage	get_instance_interface_ports <instance> <interface>	
Returns	string[]	A list of port names found in the interface.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
Example	get_instance_interface_ports uart_0 s0	

get_instance_interface_properties

This command returns a list of properties that you can be query for an interface in a child instance.

get_instance_interface_properties		
Usage	get_instance_interface_properties	
Returns	string[]	A list of property names.
Arguments	None	
Example	get_instance_interface_properties	

get_instance_interface_property <instance> <interface> <property>

This command returns the property value for an interface in a child instance.

get_instance_interface_property		
Usage	get_instance_interface_property <instance> <interface> <property>	
Return	string []	The value of the property.
Arguments	instance	The name of the child instance.
	interface	The name of an interface on the child instance.
	property	he name of the property of the interface.
Example	get_instance_interface_property uart_0 s0 DESCRIPTION	

get_instance_interfaces <instance>

This command returns a list of interfaces in a child instance.

get_instance_interfaces		
Usage	get_instance_interfaces <instance>	
Returns	string[]	A list of interface names.
Arguments	instance	The name of the child instance.
Example	get_instance_interfaces uart_0	

get_instance_parameter_property <instance> <parameter> <property>

This command returns the value of a parameter in a connection in the subsystem, for an instance that contains a subsystem.

get_instance_parameter_property		
Usage	get_instance_parameter_property <instance> <parameter> <property>	
Return	various	The name of the child instance.
Arguments	instance	The child instance containing a subsystem.
	parameter	The name of the parameter in the instance.
	property	The name of the property of the parameter.
Example	get_instance_parameter_property uart_0 baudRate ENABLED	

get_instance_parameter_value <instance> <parameter>

This command returns the value of a property in a child instance.

get_instance_parameter_value		
Usage	get_instance_parameter_value <instance> <parameter>	
Returns	various	The value of the parameter.
Arguments	instance	The name of the child instance.
	parameter	The name of the parameter in the instance.
Example	get_instance_parameter_value uart_0 baudRate	

get_instance_parameters <instance>

This command returns a list of parameters in a child instance.

get_instance_parameters		
Usage	get_instance_parameters <instance>	
Returns	string[]	A list of parameters in the instance.
Arguments	instance	The name of the child instance.
Example	get_instance_parameters uart_0	

get_instance_port_property <instance> <port> <property>

This command returns the value of a property of a port contained by an interface in a child instance.

get_instance_port_property		
Usage	get_instance_port_property <instance> <port> <property>	
Return	various	The value of the property for the port.
Arguments	instance	The name of the child instance.
	port	The name of a port in one of the interfaces on the child instance.
	property	The name of a property found on the port; DIRECTION, ROLE, WIDTH.
Example	get_instance_port_property uart_0 tx WIDTH	

get_instance_properties

This command returns a list of properties for a child instance.

get_instance_properties		
Usage	get_instance_properties	
Returns	string[]	A list of property names for the child instance.
Arguments	None	
Example	get_instance_properties	

get_instance_property <instance> <property>

This command returns the value of a property for a child instance.

get_instance_property	
Usage	get_instance_property <instance> <property>

get_instance_property		
Returns	string[]	The value of the property.
Arguments	instance	The name of the child instance.
	property	The name of a property found on the instance.
Example	get_instance_property cpu ENABLED	

get_instances

This command returns a list of the instance names for all child instances in the system.

get_instances		
Usage	get_instances	
Returns	string[]	A list of child instance names.
Arguments	None	
Example	get_instances	

get_interface_port_property <interface><port><property>

This command returns the value of a property of a port contained by an interface in a child instance.

get_interface_port_property		
Usage	get_interface_port_property <interface><port><property>	
Return	various	The value of the property.
Arguments	instance	The name of a top-level interface on the system.
	port	The name of a port found in the interface.
	property	The name of a property found on the port.
Example	get_interface_port_property uart_exports tx DIRECTION	

get_interface_ports <interface>

This command returns the names of all of the ports that have been added to an interface.

get_interface_ports	
Usage	get_interface_ports <interface>

get_interface_ports		
Returns	string[]	A list of port names.
Arguments	interface	The name of a top-level interface on the system.
Example	get_interface_ports export_clk_out	

get_interface_properties

This command returns the names of all the available interface properties. The list of interface properties is the same for all interface types.

get_interface_properties		
Usage	get_interface_properties	
Returns	string[]	A list of interface properties.
Arguments	None	
Example	get_interface_properties	

get_interface_property <interface> <property>

This command returns the value of a property from the specified interface.

get_interface_property		
Usage	get_interface_property <interface> <property>	
Return	various	The property value.
Arguments	interface	The name of a top-level interface on the system.
	property	The name of the property, EXPORT_OF.
Example	get_interface_property export_clk_out EXPORT_OF	

get_interfaces

This command returns a list of top-level interfaces in the system.

get_interfaces		
Usage	get_interfaces	
Returns	string[]	A list of the top-level interfaces exported from the system.
Arguments	None	
Example	get_interfaces	

get_module_properties

This command returns the properties that you can manage for the top-level module.

get_module_properties		
Usage	get_module_properties	
Returns	string[]	A list of property names.
Arguments	None	
Example	get_module_properties	

get_module_property <property>

This command returns the value of a top-level system property.

get_module_property		
Usage	get_module_property <property>	
Returns	string[]	The value of the property.
Arguments	property	The name of the property to query; NAME.
Example	get_module_property NAME	

get_parameter_properties

This command returns a list of properties that you can query on parameters. These properties can be queried on any parameter, such as parameters on instances, interfaces, instance interfaces, and connections.

get_parameter_properties		
Usage	get_parameter_properties	
Returns	string[]	A list of parameter properties.
Arguments	None	
Example	get_parameter_properties	

get_port_properties

This command returns a list of properties that you can query on ports.

get_port_properties		
Usage	get_port_properties	
Returns	string[]	A list of port properties.
Arguments	None	
Example	get_port_properties	

get_project_properties

This command returns a list of properties that you can query for the Quartus II project.

get_project_properties		
Usage	get_project_properties	
Returns	string[]	A list of project properties.
Arguments	None	
Example	get_project_properties	

get_project_property <property>

This command returns the value of a Quartus II project property.

get_project_property		
Usage	get_project_property <property>	
Returns	string[]	The value of the property.
Arguments	property	The name of the project property; DEVICE_FAMILY.
Example	get_project_property DEVICE_FAMILY	

load_system <file>

This command loads a Qsys system from a file, and uses the system as the current system for scripting commands.

load_system		
Usage	load_system <file>	
Returns	None	
Arguments	file	The path to a .qsys file.
Example	load_system example.qsys	

lock_avalon_base_address <instance.interface>

This command prevents the memory-mapped base address from being changed for connections to an interface on an instance when the auto_assign_base_addresses or auto_assign_system_base_addresses commands are run.

lock_avalon_base_address	
Usage	lock_avalon_base_address <instance.interface>
Returns	None

lock_avalon_base_address		
Arguments	<code>instance.interface</code>	The qualified name of the interface of an instance, in <code><instance>.<interface></code> format.
Example	<code>lock_avalon_base_address sdram.s1</code>	

preview_insert_avalon_streaming_adapters

This command runs the adapter insertion for Avalon-ST connections, which adapt connections with mismatched configuration, such as mismatched data widths.

preview_insert_avalon_streaming_adapters	
Usage	<code>preview_insert_avalon_streaming_adapters</code>
Returns	None
Arguments	None
Example	<code>preview_insert_avalon_streaming_adapters</code>

remove_connection <connection>

This command removes a connection from the system.

remove_connection		
Usage	<code>remove_connection <connection></code>	
Returns	None	
Arguments	<code>connection</code>	The name of the connection to remove.
Example	<code>remove_connection cpu.data_master/sdram.s0</code>	

remove_instance <instance>

This command removes a child instance from the system.

remove_instance		
Usage	<code>remove_instance <instance></code>	
Returns	None	
Arguments	<code>instance</code>	The name of the child instance to remove.
Example	<code>remove_instance cpu</code>	

remove_interface <interface>

This command removes an exported top-level interface from the system.

remove_interface		
Usage	remove_interface <interface>	
Returns	None	
Arguments	interface	The name of the exported top-level interface.
Example	remove_interface clk_out	

save_system [<file>]

This command saves the current in-memory system to the named file. If the file is not specified, the system saves to the same file that was opened with the `load_system` command.

save_system		
Usage	save_system [<file>]	
Returns	None	
Arguments	file optional	If present, the path of the .qsys file to save.
Example	save_system save_system example.qsys	

send_message <level> <message>

This command sends a message to the user of the script. The message text is normally interpreted as HTML. You can use the `` element to provide emphasis.

send_message	
Usage	send_message <level> <message>
Return	None

send_message		
Arguments	level	<p>The following message levels are supported:</p> <ul style="list-style-type: none"> • ERROR—Provides an error message. • WARNING—Provides a warning message. • INFO—Provides an informational message. • PROGRESS—Provides a progress message. • DEBUG—Provides a debug message when debug mode is enabled.
	message	The text of the message.
Example	send_message ERROR "The system is down!"	

set_connection_parameter_value <connection> <parameter> <value>

This command sets the parameter value for a connection.

set_connection_parameter_value		
Usage	set_connection_parameter_value <connection> <parameter> <value>	
Return	None	
Arguments	connection	The connection.
	parameter	The name of the parameter.
	value	The new parameter value.
Example	set_connection_parameter_value cpu.data_master/dma0.csr baseAddress "0x000a0000"	

set_instance_parameter_value <instance> <parameter> <value>

This command set the parameter value for a child instance. Derived parameters and SYSTEM_INFO parameters for the child instance can not be set with this command.

set_instance_parameter_value	
Usage	set_instance_parameter_value <instance> <parameter> <value>
Return	None

set_instance_parameter_value		
Arguments	instance	The name of the child instance.
	parameter	The name of the parameter.
	value	The new parameter value.
Example	set_instance_parameter_value uart_0 baudRate 9600	

set_instance_property <instance> <property> <value>

This command sets the property value of a child instance. Most instance properties are read-only and can only be set by the instance itself. The primary use for this command is to update the ENABLED parameter, which includes or excludes a child instance when generating the system.

set_instance_property		
Usage	set_instance_property <instance> <property> <value>	
Return	None	
Arguments	instance	The name of the child instance.
	property	The name of the property.
	value	The new parameter value.
Example	set_instance_property cpu ENABLED false	

set_interface_property <interface> <property> <value>

This command sets the property value on an exported top-level interface. This command is used to set the EXPORT_OF property to specify which interface of a child instance is exported by the top-level interface.

set_interface_property		
Usage	set_interface_property <interface> <property> <value>	
Return	None	
Arguments	interface	The name of an exported top-level interface.
	property	The name of the property.
	value	The new parameter value.
Example	set_interface_property clk_out EXPORT_OF clk.clk_out	

set_module_property <property> <value>

This command sets the system property value, such as the name of the system using the NAME property.

set_module_property		
Usage	set_module_property <property> <value>	
Return	None	
Arguments	property	The name of the property.
	value	The new property value.
Example	set_module_property NAME "new_system_name"	

set_project_property <property> <value>

This command sets the project property value, such as the device family.

set_project_property		
Usage	set_project_property <property> <value>	
Return	None	
Arguments	property	The name of the property.
	value	The new property value.
Example	set_project_property DEVICE_FAMILY "Cyclone IV GX"	

set_validation_property <property> <value>

This command sets a property that affects how and when validation is run during system scripting. To disable system validation after each scripting command, set AUTOMATIC_VALIDATION to false.

set_validation_property		
Usage	set_validation_property <property> <value>	
Return	None	
Arguments	property	The name of the property.
	value	The new property value.
Example	set_validation_property AUTOMATIC_VALIDATION false	

unlock_avalon_base_address <instance.interface>

This command allows the memory-mapped base address to be changed for connections to an interface on an instance when the auto_assign_base_addresses or auto_assign_system_base_addresses commands are run.

unlock_avalon_base_address	
Usage	unlock_avalon_base_address <instance.interface>

unlock_avalon_base_address		
Return	None	
Arguments	instance.interface	The qualified name of the interface of an instance, in <instance>.<interface> format
Example	unlock_avalon_base_address sdram.s1	

upgrade_sopc_system <filename>

This command loads the specified **.sopc** file, which then upgrades the file as a Qsys-compatible system. Some child instances and interconnect are replaced so that the system functions in Qsys. You must save the new Qsys-compatible system with the `save_system` command.

upgrade_sopc_system		
Usage	upgrade_sopc_system <filename>	
Return	None	
Arguments	filename	The path to the .sopc file being upgraded. The upgrade moves the .sopc file and related generation files to a backup directory.
Example	upgrade_sopc_system old_system.sopc	

validate_connection <connection>

This command validates the specified connection, and returns the during validation messages.

validate_connection		
Usage	validate_connection <connection>	
Return	string []	A list of messages produced validation.
Arguments	connection	The path to the .sopc file being upgraded. The upgrade moves the .sopc file and related generation files to a backup directory.
Example	validate_connection cpu.data_master/sdram.s1	

validate_instance <instance>

This command validates the specified child instance, and returns the validation messages.

validate_instance	
Usage	validate_instance <instance>

validate_instance		
Return	string []	A list of messages produced validation.
Arguments	instance	The name of the child instance to validate.
Example	validate_instance cpu	

validate_instance_interface <instance> <interface>

This command validates an interface on a child instance, and returns the validation messages.

validate_instance_interface		
Usage	validate_instance_interface <instance> <interface>	
Return	string []	A list of messages produced validation.
Arguments	instance	The name of the child instance.
	interface	The name of the instance on the child instance to validate.
Example	validate_instance_interface cpu data_master	

validate_system

This command validates the system, and returns the validation messages.

validate_system		
Usage	validate_system	
Return	string []	A list of messages produced validation.
Arguments	None	
Example	validate_system	

Document Revision History

Table 6-7 indicates edits made to the *Creating a System With Qsys* content since its creation.

Table 6-7: Document Revision History

Date	Version	Changes
November 2013	13.1.0	<ul style="list-style-type: none"> Added: <i>Integrating with the .qsys File.</i> Added: <i>Using the Hierarchy Tab.</i> Added: <i>Managing Interconnect Requirements.</i> Added: <i>Viewing Qsys Interconnect.</i>
May 2013	13.0.0	<ul style="list-style-type: none"> Added AMBA APB support. Added qsys-generate utility. Added VHDL BF <p>May 2013</p> <p>M ID support.</p> <ul style="list-style-type: none"> Added <i>Creating Secure Systems (TrustZones)</i> . Added <i>CMSIS Support for Qsys Systems With An HPS Component.</i> Added VHDL language support options.
November 2012	12.1.0	<ul style="list-style-type: none"> Added AMBA AXI4 support.
June 2012	12.0.0	<ul style="list-style-type: none"> Added AMBA AX3I support. Added Preset Editor updates. Added command-line utilities, and scripts.
November 2011	11.1.0	<ul style="list-style-type: none"> Added Synopsys VCS and VCS MX Simulation Shell Script. Added Cadence Incisive Enterprise (NCSIM) Simulation Shell Script. Added Using Instance Parameters and Example Hierarchical System Using Parameters.
May 2011	11.0.0	<ul style="list-style-type: none"> Added simulation support in Verilog HDL and VHDL. Added testbench generation support. Updated simulation and file generation sections.
December 2010	10.1.0	Initial release.

Related Information

[Quartus II Handbook Archive](#)