

You define SOPC Builder components by declaring their properties and behaviors in a Hardware Component Description File (`_hw.tcl`). Each `_hw.tcl` file represents one component instance which you can add to an SOPC Builder system. You can also share the components that you design with other designers. For your component to have maximum flexibility, you should consider what aspects of its behavior can be parameterized so that other users can change the default parameterization to address different design requirements.

An SOPC Builder component is usually composed of the following four types of files:

- `_hw.tcl` file—describes the SOPC Builder related characteristics, such as interface behaviors. This file is required.
- HDL files—define the component's functionality as hardware. These files are optional.
- `_sw.tcl`—used by the software build tools to compile the component driver code. This file is optional.
- Component driver files—defines the component register map and driver software to allow software to control the component. These files are optional.

This chapter discusses the following topics:

- [“Information in a Hardware Component Description File” on page 7-1](#)
- [“Component Phases” on page 7-2](#)
- [“Writing a Hardware Component Description File” on page 7-2](#)
- [“Overriding Default Behaviors” on page 7-8](#)
- [“Hardware Tcl Command Reference” on page 7-12](#)

Information in a Hardware Component Description File

A typical `_hw.tcl` file contains the following information:

- Basic component information—includes the component's name, version, and description, a link to its documentation, and pointers to HDL implementation files for synthesis and simulation.
- Parameter Declarations—Parameters are values that the user of your component can set that affect how the component is implemented, such as the size of a memory. Properties of each parameter include the parameter's name, whether or not it is visible, and, if visible, the text to display when describing it. When the SOPC Builder system is generated, the parameters can be applied to the component as Verilog HDL parameters or VHDL generics.
- Interface Properties—The interfaces of a component define how to connect it to the rest of the system and determine how other components in the system interact with it. When you add interfaces to a component, you declare which signals make up each interface. You also define interface properties, such as wait states for an Avalon® Memory-Mapped (Avalon-MM) interface.

Depending on your component design, your `_hw.tcl` file may be one of the following two types:

- **Static**—A static `_hw.tcl` file defines the top-level HDL file and associated component files. The HDL that describes a static component is created by the component author and is not changed by users of the component. HDL parameters are available when instantiating the component.
- **Generated**—A generated `_hw.tcl` file provides a user-defined program to generate the component's HDL. The HDL can be different for different parameterizations of the component.

Component Phases

The following section describes the distinct phases in the development of an SOPC Builder component.

- **Main Program**—SOPC Builder first discovers a component and adds it to the component library. The `_hw.tcl` file is executed and the Tcl statements provide non-instance-specific information to SOPC Builder. During this phase, some component interfaces may be incompletely described and ports may have a width of 0 or -1 to indicate that they are variable.
- **Validation**—Validation allows the component to generate error, warning, or informational messages. Validation occurs when an instance of a component is created, when its parameters are changed, or when some other property of the system is changed.
- **Elaboration**—Elaboration occurs as SOPC Builder queries a component for its interface information. Elaboration typically occurs immediately after validation and before generation. Interfaces defined in the main program can be enabled or disabled during elaboration. Depending on the validation callback code, elaboration and validation may alternate a few times. Elaboration and validation always occur before generation. Once elaboration is complete, the component must be completely described. For example, all port widths must have positive values.
- **Generation**—Generation creates all the information that the Quartus® II software and HDL simulator require. The required files typically include VHDL or Verilog HDL files, simulation models, timing constraints, and other information.
- **Editor**—After an instance of your component has been added to an SOPC Builder system, allows the user of your component to edit the GUI that displays the parameterization. You can change the appearance of the default editor to make it easier to use.

Writing a Hardware Component Description File

This section provides detailed information about `_hw.tcl` files and describes the default behavior of a component in all phases. The following example uses a simple UART with some simple parameterization.

Providing Basic Information

A typical `_hw.tcl` file first declares basic information such as the name, location, and the files it includes. The first command in a `_hw.tcl` file should specify the version of the `_hw.tcl` API to use, with the following Tcl command:

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

The version number is a Quartus II release version such as 9.0 or 9.1. SOPC builder guarantees that a valid `_hw.tcl` file that requests a particular `socp` package behaves identically in future versions of the tool. Because of differences between versions of the Quartus II software, you cannot assume that an HDL file that works with one `socp` package automatically works with other versions of the package.



This chapter describes the behavior of components that request the `socp 9.1` package. Refer to the 9.0 documentation for the behavior of the `socp 9.0` package.



An excellent source of information about Tcl syntax is the [Tcl Developer Xchange](#) website.

Example 7-1. Basic Information for `_hw.tcl` File

```
# The package command must be the first command in the file
package require -exact socp 9.1

# The name and VERSION of the component
set_module_property NAME example_uart
set_module_property VERSION 1.0

# The name of the component to display in the library
set_module_property DISPLAY_NAME "Example Component"

# The component's description.
set_module_property DESCRIPTION "An Example Component"

# The component library group that component belongs to
set_module_property GROUP Examples
```

Declaring Parameters

By including configuration parameters in your `_hw.tcl` file, you allow users of your component to parameterize it in different ways. Each parameter has a number of properties such as its name, type, display name, and default value that can be used to control how the parameter is displayed and used. [Example 7-2](#) illustrates the use of parameters that can be configured by users of your component.

Example 7-2. Declaring Parameters

```
# Declare Baud Rate parameter as an integer with a default value of 9600.
add_parameter BAUD_RATE int 9600

# Display this parameter as "Baud Rate" in the Parameter Editor.
set_parameter_property BAUD_RATE DISPLAY_NAME "Baud Rate (bps)"

# We only support three baud rates
set_parameter_property BAUD_RATE ALLOWED_RANGES {9600 19200 38400}
```

Parameters can be divided into three types: user parameters, system information parameters and derived parameters. The following sections describe these parameter types.

User Parameters

User parameters are parameters that users have control over and that are exposed in the component GUI.

Derived Parameters

Derived parameters are parameters that are inferred by the component itself from user parameters or other derived parameters. For example, a clock period parameter can be derived from a data rate parameter.

SYSTEM_INFO Parameters

You can use `SYSTEM_INFO` parameter to request that certain parameter values are populated with information about the system. For example, you might want to know the frequency of the clock that ends up being connected to your clock input. When you declare `SYSTEM_INFO` properties, you provide an `<info-type>` and further arguments. The `<info-type>` is the type of information you want, such as `clock_rate`, and you use the additional arguments to specify things, such as which clock input interface you require. [Example 7-3](#) illustrates the use of the `SYSTEM_INFO` parameter. For more information about the `SYSTEM_INFO` parameter properties refer to [Table 7-5 on page 7-23](#)

Example 7-3. Syntax of Tcl Command using the SYSTEM_INFO Parameter

```
set_parameter_property my_parameter SYSTEM_INFO {<info-type> [<arg>]}
```

Declaring Interfaces

To declare an interface, use the `add_interface` command. Then use the `set_interface_property` and `add_interface_port` commands to set its properties and indicate which signals belong to it. The interface declaration statement includes the name of the interface, the interface direction, and the clock interface with which it is associated. For interfaces that are not associated with clocks (such as clock interfaces themselves), omit the associated clock interface, or use the word *asynchronous*. [Example 7-4](#) illustrates interface declaration.

Example 7-4. Declare Interfaces

```
# Declare the clock sink interface, "clock_sink", type=clock, direction=sink
add_interface clock_sink clock sink

# The clock interface has two signals, named "clk" and "reset_n" of types "clk" "reset_n"
add_interface_port clock_sink clk clk input 1
add_interface_port clock_sink reset_n reset_n input 1

# Declare the Avalon slave interface, name=avalon_slave_0, type=avalon,
# direction=slave, associated with the clock_sink clock interface.
add_interface avalon_slave_0 avalon slave clock_sink

# Set a number of properties about the Avalon Slave interface
set_interface_property avalon_slave_0 writeWaitTime 0
set_interface_property avalon_slave_0 addressAlignment DYNAMIC
set_interface_property avalon_slave_0 readWaitTime 1
set_interface_property avalon_slave_0 readLatency 0

# Declare all the signals that belong to my Avalon Slave interface
add_interface_port avalon_slave_0 my_readdata readdata output 8
add_interface_port avalon_slave_0 my_read read input 1
add_interface_port avalon_slave_0 my_write write input 1
add_interface_port avalon_slave_0 my_waitrequest waitrequest output 1
add_interface_port avalon_slave_0 my_address address input 24
add_interface_port avalon_slave_0 my_writedata writedata input 8
```

Adding Files and Guiding Generation

Component description files typically provide all of the information required for generation and downstream tools, identifying the files used by the component such as HDL files and Synopsis Design Constraints files (.sdc). You also identify which of the added files is the top-level HDL file and specify which Verilog module or VHDL entity within that file is the top-level module for the component. [Example 7-5](#) illustrates the files that are typically required for generation and downstream tools.

Example 7-5. Add Files

```
# Add the HDL file to the component, to be used for synthesis and simulation.
add_file simple_uart.v {SYNTHESIS SIMULATION}

# Add the Timequest file with Quartus timing constraints.
add_file simple_uart.sdc SYNTHESIS

# Indicate which of the added HDL files holds the top-level module/entity
# that describes the component, name of the top-level module/entity
set_module_property TOP_LEVEL_HDL_FILE simple_uart.v
set_module_property TOP_LEVEL_HDL_MODULE simple_uart
```

Default Behaviors

The `_hw.tcl` file described in the previous section has default behaviors during the editor, validation, elaboration, and generation phases. These default behaviors apply to instances of a component. This section describes the default SOPC Builder behaviors for each of these phases. To override these default behaviors, refer to [“Overriding Default Behaviors” on page 7-8](#).

Validation Phase Behavior

SOPC Builder's default validation checks each parameter value against its `ALLOWED_RANGES` property. If the values specified are outside the allowed ranges, an error message is displayed.

The `ALLOWED_RANGES` property of each parameter is a list of ranges that the parameter can take on, where each range is a single value, or a range of values defined by a start and end value separated by a colon. Table 7-1 shows some examples of values the `ALLOWED_RANGES` property can take.

Table 7-1. `ALLOWED_RANGES` Property

<code>ALLOWED_RANGES</code>	Meaning
{a b c}	a or b or c
{1 2 4 8 16}	1, 2, 4, 8, or 16.
1:3	1 through 3, inclusive
{1 2 3 7:10}	1, 2, 3, or 7 through 10 inclusive

Elaboration Phase Behavior

If the main program does not explicitly define the widths of all ports to constant values or to an expression, then SOPC Builder's default elaboration process calls `quartus_map` to determine the correct port widths. If you define all port widths in the main program, `quartus_map` is not called.

Automatic Port Widths

When port widths are not specified, or have a value of '-1', `quartus_map` is used to determine port widths as a function of the parameter set. While this process makes authoring a component easier, SOPC Builder can end up spending a lot of time calling `quartus_map`. When using automatic port widths, you can indicate that a certain parameter does not affect any port widths or interfaces by setting that parameter's `affects_elaboration` property to `false`, meaning that `quartus_map` is not called when the parameter's value is changed by your user. Be careful with this— indicating that a parameter does not affect elaboration when it really does can lead to problems that are mysterious and difficult to debug.

As an alternative to the automatic port widths, you can set port widths to simple HDL expressions using the `width_expr` property. `width_expr` is a string that holds an expression describing the port width. By using the `width_expr` property, you can define port widths as an expression that is evaluated without needing to analyze the HDL file or set them in an elaboration callback. The syntax for width expressions is the same as the HDL language that you use; however, only the addition, subtraction, multiplication, and division operators are allowed. For more complex port widths, the width of the port can be set as an arbitrary function of the component's parameters in an elaboration callback. The width expression is the last argument to the `add_interface_port` command. Example 7-6 illustrates the use of mathematical operators and the `width_expr` property.

Example 7-6. Defining Port Widths Using Simple Mathematical Operators

```
add_interface_port din din_data data input {WIDTH * SYMBOLS}
set_port_property din_data width_expr WIDTH
```

Parameterized Parameter Widths

For VHDL users, SOPC Builder allows a `std_logic_vector` parameter to have a width that is defined by another parameter. When adding a parameter of type `std_logic_vector` you must specify its width as part of the expression. This width can be a constant or can depend on the value of another integer parameter. The syntax below adds a `std_logic_vector` parameter whose width is set by another parameter, called `width`.

```
add_parameter name_"std_logic_vector(width-1 downto 0)"
```

For `std_logic_vector` parameters the lower bound must be 0.

Generation Phase Behavior

SOPC Builder's default generation does one of the following:

- If the component defines the `TOP_LEVEL_HDL_MODULE` property, SOPC Builder creates a Verilog HDL or VHDL wrapper module to instantiate the top-level module and applies the parameters as selected by the user of your component. SOPC Builder does not apply parameters in the wrapper if they are not declared in the underlying HDL file.

or

- If the component does not define the `TOP_LEVEL_HDL_MODULE` property, but instead sets the `INSTANTIATE_IN_SYSTEM_MODULE_module` property to `false`, the module is not instantiated inside the SOPC Builder system and a wrapper file is not created. Rather, the interface to the module is exported to the top-level of the SOPC Builder system, and the module must be connected outside the system.

Edit Phase Behavior

SOPC Builder's default editor phase behavior is to use all of the parameter definitions to display the parameterization GUI. The properties of the parameters guide SOPC Builder when it builds the default GUI. [Table 7-4 on page 7-21](#) lists the properties of parameters.

You can place parameters in logical groups and provide images and text to create a custom GUI for your component. [Example 7-7](#) defines four parameters and illustrates the use of the `add_display_item` command and the `DISPLAY_HINT` and `ALLOWED_RANGES` parameters.

Example 7-7. Defining and Customizing GUI Parameters

```
# provide an icon for the sound group
add_display_item icon Speaker speaker-image speaker.png
add_parameter sound string 0 0
add_parameter volume_control boolean 0 0
add_parameter separate_control string 0 0

# Setup display_names for the parameters
set_parameter_property sound DISPLAY_NAME Audio
set_parameter_property volume_control DISPLAY_NAME "Include Volume Control Interface"
set_parameter_property separate_control DISPLAY_NAME "Treble/Bass Controls"

# Display all parameters in the Speaker group
add_display_item Speaker sound parameter
add_display_item Speaker volume_control parameter
add_display_item Speaker separate_control parameter

# There are 4 choices for the sound parameter.
# Strings with internal spaces require double quotes
set_parameter_property sound ALLOWED_RANGES {"0:No Audio" 1:Monophonic 2:Stereo
4:Quadraphonic}
set_parameter_property separate_control ALLOWED_RANGES {"No Control" "Single Control" "Dual
Controls"

#Specify how parameters should be displayed
set_parameter_property volume_control DISPLAY_HINT boolean
set_parameter_property separate_control DISPLAY_HINT radio
```

Figure 7-1 shows the GUI that the Tcl commands in Example 7-12 produces.

Figure 7-1. Parameter GUI for Audio Component



Overriding Default Behaviors

You can override each of the default behaviors by using callbacks. This section explains how to write callback procedures for each phase of component development.

Validation Callback

You can use the validation callback to provide validation that extends beyond the default range checking. A validation callback is defined by setting the VALIDATION_CALLBACK module property to be the name of the validation callback procedure, as shown in Example 7-8. This validation procedure displays an error if you select a baud rate of 38400 and odd parity.

You can also use the validation callback to set the value of derived parameters. Derived parameters are parameters that are derived from other parameters; their values are not editable and are not saved in the SOPC Builder design file (**.sopc**). You indicate that a parameter is derived by setting the parameter's `DERIVED` property to `true`. In [Example 7-8](#) `BAUDRATE_PRESCALE` is a derived parameter whose value is 1/16 of the value of the `BAUDRATE` parameter.

Example 7-8. Custom Validation Callback Function

```
# Declare the validation callback.
set_module_property VALIDATION_CALLBACK my_validation_callback

# Add the BAUDRATE_PRESCALE parameter, and indicate that it's derived
add_parameter BAUDRATE_PRESCALE int 600
set_parameter_property BAUDRATE_PRESCALE DERIVED true

# Add the PARITY parameter
add_parameter PARITY string ODD
set_parameter_property PARITY ALLOWED_RANGES {EVEN ODD}

# The validation callback
proc my_validation_callback {} {
    # Get the current value of parameters we care about
    set br [get_parameter_value BAUD_RATE]
    set p [get_parameter_value PARITY]
    # display an error for invalid combinations.
    if {($br==38400) && ($p=="ODD")} {
        send_message warning "Odd parity at 38400 bps is not supported."
    }
    # Set the value of our DERIVED parameter
    set bp [expr $br / 16]
    set_parameter_value BAUDRATE_PRESCALE $bp
}
```

Elaboration Callback

You can use an elaboration callback to change interface properties or add new interfaces as a function of parameter values. You define an elaboration callback by setting the `ELABORATION_CALLBACK` module property to the name of the elaboration callback function, as shown in [Example 7-9](#). You can enable and disable interfaces from the elaboration callback if they are only needed for some parameterizations of the component. [Example 7-9](#) shows how an Avalon-MM slave interface can be included in an instance of the component, based on the `USE_STATUS_INTERFACE` parameter. All of the functionality available in the validation callback can also be used in the elaboration callback; separate callbacks for validation and elaboration are not required.



The elaboration callback will not be called when parameters with `AFFECTS_ELABORATION=false` are changed by the user of the component.

Example 7-9. Elaboration Callback

```
# Declare the callback.
set_module_property ELABORATION_CALLBACK my_elaboration_callback

# add the USE_STATUS_INTERFACE parameter
add_parameter USE_STATUS_INTERFACE boolean

# declare the status slave interface
add_interface status_slave avalon slave clock_sink
set_interface_property status_slave ENABLED false

# The elaboration callback
# Declare signals
add_interface_port status_slave st_readdata readdata output 16
add_interface_port status_slave st_read read input 1
add_interface_port status_slave st_write write input 1
add_interface_port status_slave st_waitrequest waitrequest output 1
add_interface_port status_slave st_address address input 24
add_interface_port status_slave st_writedata writedata input 16

# The elaboration callback
proc my_elaboration_callback {} {

    # Get the current value of parameters we care about
    set use_status [get_parameter_value USE_STATUS_INTERFACE]

    # Optionally add the status interface
    if { $use_status } {
        set_interface_property status_slave ENABLED true
    }
}
```

Generation Callback

If you define a generation callback, SOPC Builder does not generate an HDL wrapper file to apply parameter values to your component. Instead, it calls the generation callback you defined during the generation phase, allowing the component to programmatically generate its HDL. A generation callback is defined by setting the `GENERATION_CALLBACK` module property to be the name of the generation callback function, as [Example 7-10](#) illustrates.

Generation callbacks typically retrieve the current value of the component's parameters and the generation properties that guide the generation process, and then generate the HDL files and supporting files in Tcl or by calling an external program. The callback procedure also reports the required files to SOPC Builder with the `add_file` command. Any files added in the generation callback are in addition to the files added in the main body of the `_hw.tcl` file.

The generation callback must write `<output_name.v or .sv>` for Verilog or `<output_name.vhd>` for VHDL to the specified `<output_directory>`. This file is a parameterized instance of the component. Other supporting files, such as `.hex` files to initialize memory, may be written to `<output_directory>`. These file names must begin with `<output_name>`. If the supporting files are the same for all parameterizations of the component, you add them from the main program rather than the generation callback. If your system includes multiple instantiations of a component with

different parameterizations, you must add the supporting files from the main program to prevent failures. If a static supporting file is only needed in some parameterizations of the component, you should add it from the main program and turn it on or off by setting its `SYNTHESIS` and `SIMULATION` properties appropriately from the elaboration callback.

Example 7-10. Generation Callback Example

```
set_module_property GENERATION_CALLBACK my_generate
# My generation method
proc my_generate {} {
    send_message info "Starting Generation"

    # get generation settings

    set language [get_generation_property HDL_LANGUAGE]
    set outdir [get_generation_property OUTPUT_DIRECTORY ]
    set outputname [get_generation_property OUTPUT_NAME ]

    # get parameter values

    set p1 [get_parameter_value PARAMETER_ONE]
    set csr [get_parameter_value CSR_ENABLED]

# Your callback needs to write $outdir$outputname.v here,
# perhaps by using exec to call an external program.

    # add_file creates files relative to the _hw.tcl directory; therefore specify $outdir
    # for synthesis and simulation files

    exec perl my_generate.pl lang=$language dir=$outdir name=$outputname p1=$p1 csr=$csr
    add_file ${outdir}${outputname}.v SYNTHESIS
    add_file ${outdir}${outputname}_sim.v SIMULATION
}
```

Editor Callback

You can use the editor callback procedure to replace the parameterization GUI. An editor callback is defined by setting the `EDITOR_CALLBACK` module property to the name of your editor callback procedure, as shown in the [Example 7-11](#). If the editor callback is defined, SOPC Builder calls the editor callback instead of displaying the parameterization GUI, typically when the component is added to a system or updated after it is in the system.

To display your custom GUI, the editor callback must call another program. Typically, an editor callback provides the current parameter values to your program via the command line and collects the new parameter values via `stdout`. The editor callback then uses the `set_parameter_value` command to update SOPC Builder with the new parameter values.

The editor callback returns one of the following three values:

- `OK`—indicates that the results of the edit should be applied.
- `CANCEL`—indicates that the system should revert to the state it was in before the editor callback was called.
- `ERROR`—indicates that the GUI was unable to launch. An appropriate error message should be displayed.

If no value is returned, `OK` is assumed.

Example 7-11. Editor Callback

```
set_module_property EDITOR_CALLBACK my_editor

# Define Module parameters.
add_parameter PARAMETER_ONE integer 32 "A parameter"
add_parameter CSR_ENABLED boolean true "Enable CSR interface"

# My editor method
proc my_editor {} {
    # get parameter values
    set p1 [ get_parameter_value PARAMETER_ONE ]
    set csr [ get_parameter_value CSR_ENABLED ]

    # Display UI, populated with current parameter values.
    # The stdout returned by the UI program includes the new parameter values.
    set result [exec my_component_ui.exe p1=$p1 csr=$csr]

    # Use the fictional "parse_for_new_value" procedure to parse the returned text for the
    # new parameter values.
    set p1 [parse_for_new_value $result p1]
    set csr [parse_for_new_value $result csr]

    # Return the new parameter values to SOPC Builder
    set_parameter_value PARAMETER_ONE $p1
    set_parameter_value CSR_ENABLED $csr
    return OK
}
```

Hardware Tcl Command Reference

This section provides a reference for all hardware Tcl commands, as follows:

- [“Module Definition” on page 7-14](#)
- [“Parameters” on page 7-20](#)
- [“Display Items” on page 7-27](#)
- [“Interfaces and Ports” on page 7-29](#)
- [“Generation” on page 7-35](#)

The description of each command indicates during which phases it is available: in the main body of the program (main), or during the validation, elaboration, generation, and editor callback phases, or any combination. [Table 7-2](#) summarizes the commands and provides a reference to the full description.



Starting with Quartus II software version 9.1, all Tcl commands that you can use in the validation callback are also available in the elaboration callback. With this change, you may be able to omit the custom validation callback by including some validation commands in your elaboration callback.

Table 7-2. Command Summary (Note 1) (Sheet 1 of 2)

Command	Full Description
Module Definition	
package <require> -exact sopc <version>	page 7-14
get_module_properties	page 7-15
get_module_property <propertyName>	page 7-16
set_module_property <propertyName> <propertyValue>	page 7-16
get_module_ports	page 7-17
get_module_assignments	page 7-17
get_module_assignment <moduleName>	page 7-18
set_module_assignment <moduleName> [value]	page 7-18
get_files	page 7-18
add_file filename [<fileProperties> . . .]	page 7-18
get_file_properties	page 7-19
get_file_property <filename> <propertyName>	page 7-19
set_file_property <filename> <propertyName> <propertyValue>	page 7-19
send_message <messageLevel> <messageText>	page 7-20
Parameters	
add_parameter <parameterName> <parameterType> [<defaultValue> <description>]	page 7-20
get_parameter_properties	page 7-21
get_parameters	page 7-25
get_parameter_property <parameterName> <propertyName>	page 7-25
set_parameter_property <parameterName> <propertyName> <value>	page 7-25
get_parameter_value <parameterName>	page 7-26
set_parameter_value <parameterName> <value>	page 7-26
decode_address_map <address_map_XML_string>	page 7-26
Display Items	
add_display_item <groupName> <id> <type> [<additionalInfo>]	page 7-27
get_display_items	page 7-28
Interfaces and Ports	
add_interface <interfaceName> <interfaceType> <direction> [<associatedClock>]	page 7-29
get_interfaces	page 7-30
<interfaceName>	page 7-30
get_interface_property <interfaceName> <propertyName>	page 7-31
set_interface_property <interfaceName> <propertyName> <value>	page 7-31
add_interface_port <interfaceName> <portName> <portRole> [<direction> <width_expr>]	page 7-32
get_interface_ports [<interfaceName>]	page 7-32
get_port_properties	page 7-32

Table 7-2. Command Summary (Note 1) (Sheet 2 of 2)

Command	Full Description
<code>get_port_property <portName> <propertyName></code>	page 7-34
<code>set_port_property <portName> <propertyName> [<value>]</code>	page 7-34
<code>get_interface_assignments</code>	page 7-34
<code>get_interface_assignment <interfaceName> <name></code>	page 7-35
<code>set_interface_assignmet <interfaceName> <name> [<value>]</code>	page 7-35
Generation	
<code>get_generation_property <propertyName></code>	page 7-36
<code>get_generation_properties</code>	page 7-35

Note to Table 7-2:

(1) Arguments enclosed in []'s are optional

Module Definition

This section provides information about the commands that you use to define and query a module.

package

The `package` command allows you to specify a particular version of the SOPC Builder software to avoid software compatibility issues. You should use the `package` command at the beginning of your `_hw.tcl` file. When used, the component files behave as if they are interpreted by the version of the SOPC Builder software that you specify. When the `package` command is not used, version 9.0 of the SOPC Builder software is assumed. For components designed before 9.0, you can set the required package to 9.0. This document describes the behavior of component which start with `package require -exact socp 9.1` Refer to the Quartus II version 9.0 documentation for components that use `socp 9.0`



`package` is a standard Tcl command. For more information on this command refer to the following web page: <http://www.tcl.tk/man/tcl8.0/TclCmd/package.htm>

package	
Callback availability	Main (before any other commands in the file)
Usage	<code>package require -exact socp <version></code>
Returns	None
Arguments	<code>version</code> The version of SOPC Builder that you require, specified as decimal number
Example	<code>package require -exact socp 9.1</code>

get_module_properties

This command returns the names of all the available module properties as a list of strings. You can use the `get_module_property` and `set_module_property` commands to get and set values of individual properties. The value returned by this command is always the same for a particular version of SOPC Builder.

get_module_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_module_properties</code>
Returns	List of strings
Arguments	None
Example	<code>get_module_properties</code>


Table 7-3 lists the available module properties, their use, and the phases in which they can be set.

Table 7-3. Module Properties(Sheet 1 of 2)

Property Name	Property Type	Can Be Set	Description
NAME	String	Main program	The name of the module, such as <code>my_sopc_component</code> .
DISPLAY_NAME	String	Main program	The name to display when referencing the module, such as "My SOPC Component."
VERSION	String	Main program	The module's version, such as 8.1.
AUTHOR	String	Main program	The module's author.
DESCRIPTION	String	Main program	The description of the module, such as "Example SOPC Builder Module."
GROUP	String	Main program	The component group that the module belongs to, such as "Example Components."
ICON_PATH	String	Main program	A path to an icon to display in the module's parameter editor.
DATASHEET_URL	String	Main program	A path to the module's data sheet, using a syntax that provides the entire URL, not a relative path. For example: <code>http://www.mydomain.com/my_memory_controller.html</code> or <code>file:///datasheet.txt</code> .
EDITABLE	Boolean	Main program	Indicates if the component is editable in the component editor.
MODULE_TCL_FILE	String	Can only be read, not set	The path to the <code>_hw.tcl</code> file.
MODULE_DIRECTORY	String	Can only be read, not set	The directory containing the <code>_hw.tcl</code> file. All relative file names within the Tcl file are resolved relative to this directory. This directory is set as the current directory when running the main program or a callback.
TOP_LEVEL_HDL_FILE	String	Main program	Indicates which of the files added by the <code>add_file</code> command contains the module's top-level HDL.
TOP_LEVEL_HDL_MODULE	String	Main program	Indicates the name of the top-level module which must be defined in the module's top-level HDL file.

Table 7-3. Module Properties(Sheet 2 of 2)

Property Name	Property Type	Can Be Set	Description
INSTANTIATE_IN_SYSTEM_MODULE	Boolean	Main program	When <code>false</code> the instances of the module are not included in the generated system interconnect fabric. Instead, interfaces to the module are exported out of the top-level of the SOPC Builder system.
VALIDATION_CALLBACK	String	Main program	The name of the validation callback. This callback is run in addition to the default validation.
EDITOR_CALLBACK	String	Main program	The name of the editor callback. The default parameterization UI is displayed if this property is not set.
ELABORATION_CALLBACK	String	Main program	The name of the elaboration callback. For static and generated components, the default elaborations used if this property is not set.
GENERATION_CALLBACK	String	Main program	The name of the generation callback.

 The `INSTANTIATE_IN_SYSTEM_MODULE`, `TOP_LEVEL_HDL_MODULE` and `GENERATION_CALLBACK` commands are used to select the type of generation used by the component. You must set only one of these in the main program of your file.

get_module_property

This command returns the value of a single module property.

get_module_property	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_module_property <propertyName></code>
Returns	String, boolean, or file
Arguments	<code>propertyName</code> One of the properties listed in Table 7-3 on page 7-15
Example	<code>set my_name [get_module_property NAME]</code>

set_module_property

This command allows you to set the values for module properties.

set_module_property	
Callback availability	Main program
Usage	<code>set_module_property <propertyName> <propertyValue></code>
Returns	None
Arguments	<code>propertyName</code> One of the properties listed in Table 7-3 on page 7-15 <code>propertyValue</code> The new value of the property
Example	<code>set_module_property VERSION 9.1</code>

get_module_ports

This command returns a list of the names of all the ports which are currently defined.

get_module_ports	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_module_ports</code>
Returns	String
Arguments	None
Example	<code>get_module_ports</code>

get_module_assignments


This command returns names of the module assignment variables.

get_module_assignments	
Callback availability	Main, validation, elaboration, and compose
Usage	<code>get_module_assignments</code>
Returns	String
Arguments	None
Example	<code>get_module_assignments</code>

get_module_assignment

This command returns the value of the specified argument. You can use the `get_module_assignment` and `set_module_assignment` and the `get_interface_assignment` and `set_interface_assignment` commands to transfer information about hardware components to embedded software tools and applications.

get_module_assignment	
Callback availability	Main, validation, elaboration, and compose
Usage	<code>get_module_assignment <name></code>
Returns	String
Arguments	<code>name</code> The name whose value is being retrieved
Example	<code>get_module_assignment embedded.sw.CMacro.colorSpace</code>

 For more information about specifying information for software tools, refer to *Publishing Component Information to Embedded Software* in the *Nios II Software Developer's Handbook - Studio Edition*.

set_module_assignment

This command sets the value of the specified argument.

set_module_assignment		
Callback availability	Main, validation, elaboration, and compose	
Usage	<code>set_module_assignment <name> [<value>]</code>	
Returns	None	
Arguments	name	The name whose value is being set
	value	The value of the <name> argument
Example	<code>set_module_assignment embedded.sw.CMacro.colorSpace CMYK</code>	

get_files

This command returns a list of all the files that have been added to the module.

get_files	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_files</code>
Returns	List of strings
Arguments	None
Example	<code>set list_of_files [get_files]</code>

add_file

This command adds a synthesis, simulation, or TimeQuest constraints file to the module. Files added in the main program cannot be removed. Adding files in the generation callback allows the included files to be a function of the parameter set or to be a result of generation. Files added in callbacks are in addition to any files added in the main program.

add_file		
Callback availability	Main and generation	
Usage	<code>add_file filename [<fileProperties> . . .]</code>	
Returns	String	
Arguments	filename	The file name to be added, relative to the directory containing the <code>_hw.tcl</code> file
	fileProperties	Files support the following 3 properties: <ul style="list-style-type: none"> ■ SIMULATION—File for simulation ■ SYNTHESIS—File for synthesis ■ SDC—TimeQuest constraints (SDC behaves like a synthesis file)
Example	<code>add_file my_component.v {SIMULATION SYNTHESIS}</code>	

get_file_properties

This command returns the list of all properties that have been defined for a file.

get_file_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_file_properties</code>
Returns	List of strings
Arguments	None
Example	<code>get_file_properties</code>

get_file_property

This command returns the value of a single file property. The file name passed as an argument may be a partial as long as it is unique. For example, if the full file name is `/components/my_file.v`, `my_file.v` is sufficient.

get_file_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	<code>get_file_property <filename> <propertyName></code>	
Returns	Boolean	
Arguments	<code>filename</code>	The file name whose properties are being retrieved
	<code>propertyName</code>	The file name property whose value is being retrieved
Example	<code>set forSynthesis [get_file_property my_file.v SYNTHESIS]</code>	

set_file_property

This command sets the value of a single file property. The file name passed to the function can be a partial file name as long as it is unique. For example, if the full file name is `/components/my_file.v`, `my_file.v` is sufficient. The available properties are described in the `add_files` command.

set_file_property		
Callback availability	Main, elaboration, and generation	
Usage	<code>set_file_property <filename> <propertyName> <propertyValue></code>	
Returns	Boolean	
Arguments	<code>filename</code>	The file name whose properties are being retrieved
	<code>propertyName</code>	Name of the file property whose value is being retrieved
	<code>propertyValue</code>	Value to set for the file property
Example	<code>set_file_property my_file.v SYNTHESIS true</code>	

send_message

This command sends a message to the user of the component. The message text is normally interpreted as HTML. The `` element can be used to provide emphasis. If you do not want the message text to be interpreted as HTML then pass a list like `{ info text }` as the message level.

send_message		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	<code>send_message <messageLevel> <messageText></code>	
Returns	None	
Arguments	messageLevel	The following 4 message levels are supported: <ul style="list-style-type: none"> ■ Error—provides an error message. The SOPC Builder system cannot be generated while there are error messages. ■ Warning—provides a warning message. ■ Info—provides an informational message. ■ Debug—provides messages when debug mode is enabled.
	messageText	The text of the message
Example	<code>send_message Error "param1 must be greater than param2."</code>	

Parameters

Parameters allow users of your component to affect its operation in the same manner as Verilog HDL parameters or VHDL generics.

add_parameter

This command adds a parameter to your component.

add_parameter		
Callback availability	Main program	
Usage	<code>add_parameter <parameterName> <parameterType> [<defaultValue> <description>]</code>	
Returns	String	
Arguments	parameterName	A name that you, the component author, choose for your parameter
	parameterType	The following 7 types are supported: <code>Integer</code> , <code>Natural</code> , <code>Positive</code> , <code>Boolean</code> , <code>Std_logic</code> , <code>Std_logic_vector</code> , <code>String</code>
	defaultValue	The default length of the parameter is derived from its range.
	description	Explains the use of the parameter
Example	<code>add_parameter seed integer 17 "The seed to use for data generation."</code>	

get_parameter_properties

This command returns a list of all the available parameter properties as a list of strings. The `get_parameter_property` and `set_parameter_property` commands are used to get and set the values of these properties, respectively.

get_parameter_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_parameter_properties</code>
Returns	List of strings
Arguments	None
Example	<code>set property_summary [get_parameter_properties]</code>

Table 7-4 describes the properties available to describe the behaviors of each of the parameters you can specify, their use, and when they can be set.

Table 7-4. Parameter Properties (Sheet 1 of 3)

Property Name	Type/Default	Can Be Set	Description
DISPLAY_NAME	String, " "	Main program	The text string to use when displaying the parameter.
ALLOWED_RANGES	String, " "	Main program	Indicates the range or ranges that the parameter value can have. For integers, The ALLOWED_RANGES property is a list of ranges that the parameter can take on, where each range is a single value, or a range of values defined by a start and end value separated by a colon, such as 11:15. This property can also specify legal values and display strings for integers, such as {0:None 1:Monophonic 2:Stereo 4:Quadrophonic} meaning 0,1,2,4 are the legal values. Refer to Example 7-7 on page 7-8 and Figure 7-1 on page 7-8 for examples illustrating the use of this property.
UNITS	String, " "	Main program	Sets the units of the parameter. The following values are possible: picoseconds, nanoseconds, microseconds, milliseconds, seconds, hertz, kilohertz, megahertz, gigahertz, address, bits, bytes, kilobytes, megabytes, gigabytes, bitspersecond, kilobitspersecond, megabitspersecond, gigabitspersecond. For example, <code>set_parameter_property frequency UNITS gigahertz</code>
HDL_PARAMETER	Boolean, false	Main program	When true, the parameter must be passed to the HDL component description. The default value is false.
DESCRIPTION	String, " "	Main program	A user-visible description of the parameter.

Table 7-4. Parameter Properties (Sheet 2 of 3)

Property Name	Type/ Default	Can Be Set	Description
AFFECTS_ELABORATION (1)	Boolean, refer to DESCRIPTION	Main program	Set AFFECTS_ELABORATION to <code>false</code> for parameters that do not affect the external interface of the module. An example of a parameter that does not affect the external interface is <code>isNonVolatileStorage</code> . An example of a parameter that does affect the external interface is <code>width</code> . When the value of a parameter changes, if that parameter has set <code>AFFECTS_ELABORATION=false</code> , the elaboration phase (calling the callback or hardware analysis) is not repeated, improving performance. Because the default value of AFFECTS_ELABORATION is <code>true</code> , the provided HDL file is normally re-analyzed to determine the new port widths and configuration every time a parameter changes.
AFFECTS_GENERATION	Boolean, refer to DESCRIPTION	Main program	The default value of AFFECTS_GENERATION is <code>false</code> if you provide a top-level HDL module, it is <code>true</code> if you provide a custom generation callback. Set AFFECTS_GENERATION to <code>false</code> if the value of a parameter does not change the results of system generation.
VISIBLE	Boolean, <code>true</code>	Main program, validation, and elaboration, callbacks	Indicates whether or not to display the parameter in the parameterization GUI.
ENABLED	Boolean, <code>true</code>	Main program, validation, and elaboration, callbacks	When <code>false</code> , the parameter is disabled, meaning that it is displayed, but greyed out, indicating that it is not editable on the parameterization GUI.
DERIVED	Boolean/ <code>false</code>	Main program	When <code>true</code> , indicates that the parameter value does not need to be stored, typically because it is set from the validation callback. The default value is <code>false</code> .
DISPLAY_HINT	String, " "	Main program	Provides a hint about how to display a property. The following values are possible: <ul style="list-style-type: none"> ■ <code>boolean</code>—for integer parameters whose value can be 0 or 1. The parameter displays as a checkbox. ■ <code>radio</code>—displays a parameter with a list of values as radio buttons instead of a drop-down list. ■ <code>hexadecimal</code>—for integer parameters, display and interpret the value as a hexadecimal number, for example: <code>0x00000010</code> instead of 16. Refer to Example 7-7 on page 7-8 and Figure 7-1 on page 7-8 for examples illustrating the use of this property.

Table 7-4. Parameter Properties (Sheet 3 of 3)

Property Name	Type/Default	Can Be Set	Description
SYSTEM_INFO	String, " "	Main program	<p>Allows you to assign information about the instantiating system to a parameter that you define. <code>SYSTEM_INFO</code> requires a keyword argument specifying the type of information requested, <code><info-type></code>. <code><info-type></code> may also take an argument. The syntax of the Tcl command is:</p> <pre>set_parameter_property my_parameter SYSTEM_INFO <info-type> [<arg>]</pre> <p>The following values for <code><info-type></code> are predefined:</p> <ul style="list-style-type: none"> ■ CLOCK_RATE ■ CLOCK_DOMAIN ■ RESET_DOMAIN ■ ADDRESS_WIDTH ■ ADDRESS_MAP ■ MAX_SLAVE_DATA_WIDTH ■ INTERRUPTS_USED ■ DEVICE_FAMILY ■ DEVICE_FEATURES

Note to Table 7-4:

(1) The `AFFECTS_ELABORATION` property was called `AFFECTS_PORT_WIDTHS` before version 9.0 of the Quartus II software.

Table 7-5 lists the properties that you can use with the `system_info` parameter property.

Table 7-5. SYSTEM_INFO Properties (Sheet 1 of 2)

Property	Type	Description
CLOCK_RATE	Integer or String	<p>Assigns a positive number which is the clock frequency in Hz to the clock input interface you specify. Assigns 0 if the clock rate is not known.</p> <pre>set_parameter_property <my_parameter> SYSTEM_INFO {CLOCK_RATE <my_clk>}</pre>
CLOCK_DOMAIN	Integer	<p>Assigns an integer representing the clock domain to the parameter you specify. You can use this command to determine whether multiple interfaces in your module are on the same clock domain. The absolute value of the integer value is arbitrary, but if two interfaces are on the same clock domain, the <code>CLOCK_DOMAIN</code> value is guaranteed to be the same and greater than zero.</p> <pre>set_parameter_property <my_parameter> SYSTEM_INFO {CLOCK_DOMAIN <my_clk>}</pre>

Table 7-5. SYSTEM_INFO Properties (Sheet 2 of 2)

Property	Type	Description
RESET_DOMAIN	Integer	Assigns an integer representing the reset domain to the parameter you specify. You can use this command to determine whether multiple interfaces in your module are on the same reset domain. The absolute value of the integer value is arbitrary, but if two interfaces are on the same reset domain, the RESET_DOMAIN value is guaranteed to be the same and greater than zero. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {RESET_DOMAIN <my_reset>}</pre>
ADDRESS_WIDTH	Integer	Assigns an integer to the parameter that you specify that is the number of bits an Avalon-MM master must drive to address all of its slaves, using byte addresses. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {ADDRESS_WIDTH <my_avalon-mm_master>}</pre>
ADDRESS_MAP	String	Assigns an XML formatted string describing the address map to the parameter you specify. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {ADDRESS_MAP <my_avalon-mm_master>}</pre>
MAX_SLAVE_DATA_WIDTH	Integer	Assigns an integer to the parameter you specify that is the data width of the widest slave connected to the specified Avalon-MM master. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {MAX_SLAVE_DATA_WIDTH <my_avalon-mm_master>}</pre>
INTERRUPTS_USED	Integer or string	Creates a mask indicating which bits of the interrupt receiver vector are connected to an interrupt sender. This mask is assigned to the parameter you specify. You can use this interrupt mask to optimize logic that handles interrupts. <pre>set_parameter_property <my_parameter> SYSTEM_INFO (INTERRUPTS_USED <my_interrupt_receiver>}</pre>
DEVICE_FAMILY	String	Assigns the family name (not the specific device part number) of the currently selected device to the parameter you specify. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {DEVICE_FAMILY}</pre>
DEVICE_FEATURES	String	Creates a list of key/value pairs delineated by spaces indicating whether a particular device feature is available in the currently selected device family. The format of the list is suitable for passing to the Tcl array set command. This list is assigned to the parameter you specify. The following features are supported: M512_MEMORY, M4K_MEMORY, M9K_MEMORY, M144K_MEMORY, MRAM_MEMORY, MLAB_MEMORY, ESB, DSP, and EMUL. <pre>set_parameter_property <my_parameter> SYSTEM_INFO {DEVICE_FEATURES}</pre>

get_parameters

This command returns the names of all parameters that have been previously defined by `add_parameter` as a space separated list.

get_parameters	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_parameters</code>
Returns	List of strings
Arguments	None
Example	<code>set parameter_summary [get_parameters]</code>

get_parameter_property

This command returns a single parameter property.

get_parameter_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	<code>get_parameter_property <parameterName> <propertyName></code>	
Returns	string, boolean, or units depending on property refer to Table 7-4 on page 7-21	
Arguments	<code>parameterName</code>	The name of the parameter whose property value is being retrieved
	<code>propertyName</code>	One of the properties listed in Table 7-4 on page 7-21
Example	<code>get_parameter_property parameter1 GROUP</code>	

set_parameter_property

This command sets a single parameter property.

set_parameter_property		
Callback availability	Main, validation, and elaboration	
Usage	<code>set_parameter_property <parameterName> <propertyName> <value></code>	
Returns	string, boolean, or units depending on property	
Arguments	<code>parameterName</code>	Specifies the parameter that is being set
	<code>propertyName</code>	Specifies the property of <code>parameterName</code> that is being set, refer to Table 7-4 on page 7-21 for a list of properties
	<code>value</code>	Provides the values
Example	<code>set_parameter_property BAUD_RATE ALLOWED_RANGES {9600 19200 38400}</code>	

get_parameter_value

This command returns the current value of a parameter defined previously with the `add_parameter` command.

get_parameter_value	
Callback availability	Validation, elaboration (1), compose. generation, and editor
Usage	<code>get_parameter_value <parameterName></code>
Returns	String
Arguments	<code>parameterName</code> Specifies the parameter that is being retrieved
Example	<code>set fifo_width [get_parameter_value fifo_width]</code>

Note:

- (1) If `AFFECTS_ELABORATION=false` for a given parameter, `get_parameter_value` is not available for that parameter from the elaboration callback. If `affects_generation=false` then it is not available from the generation callback.

set_parameter_value

This command sets a parameter value. The values of derived parameters can be set from the validation and elaboration callbacks. The values of parameters which are not marked as `derived` or `system_info` can be set from the editor callback.

set_parameter_value	
Callback availability	Validation, elaboration, and editor
Usage	<code>set_parameter_value <parameterName> <value></code>
Returns	None
Arguments	<code>parameterName</code> Specifies the parameter that is being set <code>value</code> Specifies the value of <code>parameterName</code>
Example	<code>set_parameter_value BAUD_RATE 19200</code>

decode_address_map

This is a utility function to convert an XML-formatted address map into a list of Tcl lists. Each inner list is in the correct format for conversion to an array. The XML code describing each slave includes: its name, start address, and end address + 1. [Figure 7-2](#) shows a portion of an SOPC Builder system with three Avalon-MM slave devices.

Figure 7-2. SOPC Builder System with Three Avalon-MM Slaves

<input checked="" type="checkbox"/>	<code>ext_ssram</code>	Cypress CY7C1380C SSRAM	<code>pll_c0</code>	<code>0x01000000</code>	<code>0x011fffff</code>
<input checked="" type="checkbox"/>	<code>sys_clk_timer</code>	Interval Timer	<code>pll_c0</code>	<code>0x02120800</code>	<code>0x0212081f</code>
<input checked="" type="checkbox"/>	<code>sysid</code>	System ID Peripheral	<code>pll_c0</code>	<code>0x021208b8</code>	<code>0x021208bf</code>

Example 7-12 shows the XML that describes the address map for the Avalon-MM master that accesses these slaves. The format of the XML string provided may differ from that described here, it may have different white space between the elements and could include additional attributes or elements. Using `decode_address_map` command to decode the XML representing an Avalon-MM master's address map is easier and ensures that your code will work with future versions of the XML address map.



Altera recommends that you use the code provided in the description of **Example 7-12** to enumerate over the components within an address map, rather than writing your own parser.

Example 7-12. Address Map for an Avalon-MM Master

```
<address-map>
  <slave name='ext_ssram' start='0x01000000' end='0x01200000' />
  <slave name='sys_clk_timer' start='0x02120800' end='0x02120820' />
  <slave name='sysid' start='0x021208B8' end='0x021208C0' />
</address-map>
```

decode_address_map			
Callback availability	Validation, compose, elaboration, and generation		
Usage	<code>decode_address_map <address_map_XML_string></code>		
Returns	List of Tcl lists, each one suitable for passing to array set		
Arguments	<table border="1" style="width: 100%;"> <tr> <td style="width: 20%;"><code>address_map_XML_string</code></td> <td>An XML string describing the address map of an Avalon-MM master.</td> </tr> </table>	<code>address_map_XML_string</code>	An XML string describing the address map of an Avalon-MM master.
<code>address_map_XML_string</code>	An XML string describing the address map of an Avalon-MM master.		
Example	<pre>set address_map_xml [get_parameter_value my_map_param] set address_map_dec [decode_address_map \$address_map_xml] foreach i \$address_map_dec { array set info \$i send_message info "Connected to slave \$info(name)" }</pre>		

Display Items

You specify your component GUI using the display commands.

add_display_item

You can use this command to specify the following two aspects of component display:

- You can create logical groups for a component's parameters. For example, you might want to create separate groups for the component's timing, size, and simulation parameters. A component displays the groups and parameters in the order that you specify the display items for them in the `_hw.tcl` file.
- You can specify an image to provide a pictorial representation of a parameter or parameter group.

You create a display group by adding display items to it.

add_display_item		
Callback availability	Main program	
Usage	<code>add_display_item <groupName> <id> <type> [<additionalInfo>]</code>	
Returns	String	
Arguments	<code>groupName</code>	Specifies the group to which a display item belongs.
	<code>id</code>	Specifies the parameter or icon to be displayed in a group. Each display item associated with a component must have a different ID.
	<code>type</code>	Specifies the category of the display item. The following types are defined: <ul style="list-style-type: none"> ■ <code>icon</code>—a <code>.gif</code>, <code>.jpg</code>, or <code>.png</code> file ■ <code>parameter</code>—a parameter in the instance ■ <code>text</code>—a block of text ■ <code>group</code>—a group. If the <code>groupName</code> is also defined, the new group is a child of the <code>groupName</code> group. If <code>groupName</code> is an empty string, the group is top-level.
	<code>additionalInfo</code>	Provides extra information required for display items. The following examples illustrate how you use the <code>additionalInfo</code> argument for the various types: <ul style="list-style-type: none"> ■ <code>add_display_item groupName id icon path-to-image-file</code> ■ <code>add_display_item groupName parameterName parameter (additionalInfo not required)</code> ■ <code>add_display_item groupName id text "your-text"</code> The <code>your-text</code> argument is a block of text that is displayed in the GUI. Some simple HTML formatting is allowed, such as <code></code> and <code><i></code>, if the text starts with <code>"html"></code>. ■ <code>add_display_item parentGroupName childGroupName group [tab]</code> The <code>tab</code> is an optional parameter. If present, the group appears in separate tab in the GUI for the instance.
Examples	<pre>add_display_item timing read_latency parameter add_display_item sound speaker icon speaker.jpg</pre>	

get_display_items

This command returns a list of all items to be displayed as part of the parameterization GUI.


get_display_items	
Callback availability	Main, elaboration, generation, and editor
Usage	<code>get_display_items</code>
Returns	List of strings
Arguments	None
Example	<code>get_display_items</code>

Interfaces and Ports

You can use the interface and port commands to define interfaces and ports and retrieve their properties.

add_interface

This command adds an interface to your module. As the component author, you choose the name of the interface. By default, interfaces are enabled. You can set the interface property `ENABLED` to `false`, to disable a component interface. If an interface is disabled, it is hidden and its ports are automatically terminated to their default values. Signals that you designate as active low by appending a `_n` are terminated to 1. All other signals are terminated to 0.

 The properties available for each interface type are different. The common properties, `ENABLED` and `ASSOCIATED_CLOCK` apply to all interface types. Refer to the [Avalon Interface Specifications](#) for a description of other properties.

add_interface																							
Callback availability	Main program, and elaboration																						
Usage	<code>add_interface <interfaceName> <interfaceType> <direction> [<associatedClock>](1)</code>																						
Returns	String																						
Arguments	<table border="1"> <tr> <td>interfaceName</td> <td>A name that you choose to identify an interface.</td> </tr> <tr> <td>interfaceType and direction</td> <td> <p>There are 7 interfaceTypes. The following directions are possible for these interfaceTypes:</p> <table border="1"> <thead> <tr> <th>Interface Type</th> <th>Direction</th> </tr> </thead> <tbody> <tr> <td>avalon</td> <td>master, slave (2)</td> </tr> <tr> <td>avalon_tristate</td> <td>slave</td> </tr> <tr> <td>avalon_streaming</td> <td>source, sink</td> </tr> <tr> <td>interrupt</td> <td>sender, receiver</td> </tr> <tr> <td>conduit</td> <td>end</td> </tr> <tr> <td>clock</td> <td>source, sink</td> </tr> <tr> <td>nios_custom_instruction</td> <td>slave</td> </tr> </tbody> </table> </td> </tr> <tr> <td>associatedClock</td> <td>This defines the clock associated with the interface. It is required for all interfaces except clock interfaces.</td> </tr> </table>	interfaceName	A name that you choose to identify an interface.	interfaceType and direction	<p>There are 7 interfaceTypes. The following directions are possible for these interfaceTypes:</p> <table border="1"> <thead> <tr> <th>Interface Type</th> <th>Direction</th> </tr> </thead> <tbody> <tr> <td>avalon</td> <td>master, slave (2)</td> </tr> <tr> <td>avalon_tristate</td> <td>slave</td> </tr> <tr> <td>avalon_streaming</td> <td>source, sink</td> </tr> <tr> <td>interrupt</td> <td>sender, receiver</td> </tr> <tr> <td>conduit</td> <td>end</td> </tr> <tr> <td>clock</td> <td>source, sink</td> </tr> <tr> <td>nios_custom_instruction</td> <td>slave</td> </tr> </tbody> </table>	Interface Type	Direction	avalon	master, slave (2)	avalon_tristate	slave	avalon_streaming	source, sink	interrupt	sender, receiver	conduit	end	clock	source, sink	nios_custom_instruction	slave	associatedClock	This defines the clock associated with the interface. It is required for all interfaces except clock interfaces.
	interfaceName	A name that you choose to identify an interface.																					
	interfaceType and direction	<p>There are 7 interfaceTypes. The following directions are possible for these interfaceTypes:</p> <table border="1"> <thead> <tr> <th>Interface Type</th> <th>Direction</th> </tr> </thead> <tbody> <tr> <td>avalon</td> <td>master, slave (2)</td> </tr> <tr> <td>avalon_tristate</td> <td>slave</td> </tr> <tr> <td>avalon_streaming</td> <td>source, sink</td> </tr> <tr> <td>interrupt</td> <td>sender, receiver</td> </tr> <tr> <td>conduit</td> <td>end</td> </tr> <tr> <td>clock</td> <td>source, sink</td> </tr> <tr> <td>nios_custom_instruction</td> <td>slave</td> </tr> </tbody> </table>	Interface Type	Direction	avalon	master, slave (2)	avalon_tristate	slave	avalon_streaming	source, sink	interrupt	sender, receiver	conduit	end	clock	source, sink	nios_custom_instruction	slave					
Interface Type	Direction																						
avalon	master, slave (2)																						
avalon_tristate	slave																						
avalon_streaming	source, sink																						
interrupt	sender, receiver																						
conduit	end																						
clock	source, sink																						
nios_custom_instruction	slave																						
associatedClock	This defines the clock associated with the interface. It is required for all interfaces except clock interfaces.																						
Example	<code>add_interface mm_slave avalon slave clock0</code>																						

Notes:

- (1) For interfaces that are not associated with clocks, such as clock interfaces themselves, the `associatedClock` is omitted. Another option is to specify the `associatedClock` argument as `asynchronous`.
- (2) The terms *master*, *source*, and *start* are interchangeable. The terms *slave*, *sink*, and *end* are interchangeable.

get_interfaces


This command returns the names of all interfaces that have been previously defined by `add_interface` as a space separated list.

get_interfaces	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_interfaces</code>
Returns	List of strings
Arguments	None
Example	<code>set all_interfaces [get_interfaces]</code>

get_interface_properties

This command returns the names of all the available interface properties for the specified interface as a space separated list.

get_interface_properties	
Callback availability	Main program, validation, elaborations, and editor
Usage	<code>get_interface_properties <interfaceName></code>
Returns	List of strings
Arguments	<code>interfaceName</code> The name of an interface that you defined
Example	<code>get_interface_properties mm_slave</code>

 The properties available for each interface type are different. Refer to the *Avalon Interface Specifications* for more information about interface properties.

The interface properties that are common to all interface types are listed below in [Table 7-6](#).

Table 7-6. Interface Properties Common to All Interface Types

Property	Type	Description
ASSOCIATED_CLOCK	String	The name of the clock interface that this interface is synchronous to.
ENABLED	Boolean	Specifies whether or not interface is enabled.

get_interface_property

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

get_interface_property	
Callback availability	Main program, and elaboration
Usage	<code>get_interface_property <interfaceName> <propertyName></code>
Returns	string, boolean, or units depending on property Refer to the <i>Avalon Interface Specifications</i> for more information about interface properties
Arguments	<code>interfaceName</code> The name of an interface from which you want to retrieve information
	<code>propertyName</code> The name of the property whose value you want to retrieve. This property is either ENABLED or ASSOCIATED_CLOCK or a property name defined by the interface.
Example	<code>get_interface_property mm_slave readWaitTime</code>

set_interface_property

This command sets a single interface property for an interface.

set_interface_property	
Callback availability	Main and elaboration
Usage	<code>set_interface_property <interfaceName> <propertyName> <value></code>
Returns	String
Arguments	<code>interfaceName</code> The name of an interface that includes this property
	<code>propertyName</code> The name of the property whose value you want to set, which is ENABLED or ASSOCIATED_CLK or a name from the <i>Avalon Interface Specifications</i> .
	<code>value</code> The value to set for the specified property
Example	<code>set_interface_property mm_slave linewrapBursts false</code>

add_interface_port

This command adds a port to an interface on your module. As the component author, you determine the name of the port. The port width and direction must be set by the end of the elaboration phase. The port width can be set with one of the following mechanisms:

- A constant width or a width expression can be set in the main program
- A constant width can be set in the elaboration callback



Without an elaboration callback, for static components `quartus_map` determines the port width from the HDL

add_interface_port	
Callback availability	Main program and elaboration
Usage	<code>add_interface_port <interfaceName> <portName> <portRole> [<direction> <width_expr>]</code>
Returns	String
Arguments	<code>interfaceName</code> The name of the interface to which the port belongs.
	<code>portName</code> The name of the port that you, the component author, have chosen.
	<code>portRole</code> The role of this port within the interfaces. Port roles are referred to as <code>signal types</code> in the <i>Avalon Interface Specification</i> . Refer to the <i>Avalon Interface Specifications</i> for the <code>signal types</code> available for each interface type.
	<code>direction</code> The direction can be input, output, or bidir
	<code>width_expr</code> The port's width expression. In simple cases, this is just the width of the port in bits.
Example	<code>add_interface_port mm_slave s0_rdata readdata output 32</code>

get_interface_ports

This command returns the names of all of the ports that have been added to a given interface. If the interface name is omitted, all ports for all interfaces are returned.

get_interface_ports	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_interface_ports [<interfaceName>]</code>
Returns	String
Arguments	<code>interfaceName</code> The name of the interface whose ports you want to list. (Optional)
Example	<code>get_interface_ports mm_slave</code>

get_port_properties

This command returns a list of all available port properties.

get_port_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_port_properties <portName></code>
Returns	String, boolean, or units depending on property refer to Table 7-4 on page 7-21

Arguments	<p><code>portName</code> The name of the port whose properties are required. The following 7 port properties are supported:</p> <ul style="list-style-type: none"> ■ <code>direction</code> ■ <code>width</code> ■ <code>termination</code> ■ <code>termination_value</code> ■ <code>width_expr</code> ■ <code>vhdl_type</code> <p>Refer to Table 7-7 for a description of these properties.</p>
Example	<code>get_port_properties mm_slave</code>

[Table 7-7](#) describes the available port properties

Table 7-7. Port Properties

Name	Type	Description
<code>direction</code>	<code>input</code> , <code>output</code> , <code>bidir</code>	The direction of the port from the component's perspective.
<code>width</code>	<code>integer</code>	The width of the port in bits.
<code>width_expr</code>	<code>string</code>	The width expression of a port. Setting the <code>width</code> and <code>width_expr</code> properties have the same effect; they both update the effective width expression. The <code>width/width_expr</code> properties can be set to an <code>integer</code> at any time. They can only be set to arithmetic expressions in the main program. The values of the <code>width</code> and <code>width_expr</code> properties behave differently when <code>get_port_property</code> is used. <code>width</code> always returns the current integer width of the port. <code>width_expr</code> always returns the unevaluated width expression.
<code>termination</code>	<code>boolean</code>	When <code>true</code> , instead of connecting the port to the SOPC Builder system, it is left unconnected for <code>OUTPUT</code> and <code>BIDIR</code> or set to a fixed value for <code>INPUT</code> . Has no effect for components that implement a generation callback instead of using the default wrapper generation.
<code>termination_value</code>	<code>integer</code>	The constant value to drive an input port.
<code>vhdl_type</code>	<code>std_logic</code> <code>std_logic_vector</code> <code>auto</code>	indicates the type of a VHDL port. The default value, <code>auto</code> , selects <code>std_logic</code> if the width is fixed at 1, and <code>std_logic_vector</code> otherwise.

get_port_property

This command returns the value of single port property for the specified port.

get_port_property	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_port_property <portName> <propertyName></code>
Returns	Depends on the type of the property
Arguments	<code>portName</code> The name of the port
	<code>propertyName</code> One of the supported properties described in Table 7-7 .
Example	<code>get_port_property rdata WIDTH</code>

set_port_property

This command sets a single port property.

set_port_property	
Callback availability	Main program, elaboration, and generation
Usage	<code>set_port_property <portName> <propertyName> [<value>]</code>
Returns	String, boolean, or units depending on property refer to Table 7-4 on page 7-21
Arguments	<code>portName</code> The name of the port
	<code>propertyName</code> One of the supported properties described in Table 7-7 .
	<code>value</code> The value to set
Example	<code>set_port_property rdata WIDTH 32</code>

get_interface_assignments

This command returns the value of all interface assignments for the specified interface.

get_interface_assignments	
Callback availability	Main, validation, and elaboration
Usage	<code>get_interface_assignments <interfaceName></code>
Returns	String
Arguments	<code>interfaceName</code> The name of the Avalon interface whose assignment is being retrieved
Example	<code>get_interface_assignments s1</code>

get_interface_assignment


This command returns the value of the specified name for the specified interface.

get_interface_assignment	
Callback availability	Main, validation, and elaboration
Usage	<code>get_interface_assignments <interfaceName> <name></code>
Returns	String
Arguments	<code>interfaceName</code> The name of the Avalon interface whose assignment is being retrieved
	<code>name</code> The assignment whose value is being retrieved
Example	<code>get_interface_assignment s1 embeddedsw.configuration.isFlash</code>

set_interface_assignment

This command sets the value of the specified assignment for the specified interface.

set_interface_assignment	
Callback availability	Main, validation, and elaboration
Usage	<code>set_interface_assignment <interfaceName> <name> [<value>]</code>
Returns	None
Arguments	<code>interfaceName</code> The name of the Avalon interface whose assignment is being set
	<code>name</code> The assignment whose value is being set
	<code>value</code> The value to assign
Example	<code>set_interface_assignment s1 embeddedsw.configuration.isFlash 1</code>

 For more information about the use of the `set_interface_assignment` command, refer to the “Publishing Component Information to Embedded Software” chapter in the *Nios II Software Developer’s Handbook: Studio Edition*.

Generation

This section covers the commands that get generation properties.

get_generation_properties

This command returns the names of all the available generation properties as a space separated list. These properties cannot be changed by the module. Generation properties are provided to the generation callback to support per-instance HDL generation.

get_generation_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	<code>get_generation_properties</code>
Returns	String. The following generation properties are supported: <ul style="list-style-type: none"> ■ <code>hdl_language</code> ■ <code>output_directory</code> ■ <code>output_name</code> Refer to Table 7-8 for a description of the generation properties.
Arguments	None
Example	<code>get_generation_properties</code>

[Table 7-8](#) describes the generation properties.

Table 7-8. Generation Properties

Name	Type	Description
<code>hdl_language</code>	enum	The HDL language to generate. Is either <code>verilog</code> or <code>vhdl</code> (lowercase). If the module cannot generate the specified language, generating in the other language is acceptable.
<code>output_directory</code>	file	The location in which files must be generated. The filename components in the directory name are separated with forward slashes.
<code>output_name</code>	string	<code>OUTPUT_NAME</code> is <code>module_0</code> and the <code>HDL_LANGUAGE</code> is <code>verilog</code> , the file module_0.v or module_0.sv must be generated and must contain the module, <code>module_0</code> .

get_generation_property

This command returns the value of a single generation property.

get_generation_property	
Callback availability	Generation
Usage	<code>get_generation_property <propertyName></code>
Returns	String, boolean, or units depending on property refer to Table 7-4 on page 7-21
Arguments	<code>propertyName</code> <ul style="list-style-type: none"> One of the 3 generation properties: <ul style="list-style-type: none"> ■ <code>HDL_LANGUAGE</code> ■ <code>OUTPUT_DIRECTORY</code> ■ <code>OUTPUT_NAME</code>
Example	<code>get_generation_property OUTPUT_DIRECTORY</code>

Deprecated Commands and Properties

Table 7-9 lists commands and properties that were available in previous versions of the Quartus II software and the command that have replaced them.

Table 7-9. Deprecated Commands and Properties

Deprecated Command	Replacement
<code>add_clock_interface <name></code>	<code>add_interface <name> clock input</code>
<code>add_port_to_clock_interface <name> <role> <interface></code>	<code>add_interface_port <interface> <name> <role> 1</code>
<code>add_port_to_interface <interface> <name> <role></code>	<code>add_interface_port <interface> <name> <role> <input> 1</code>
<code>get_generation_setting <property></code>	<code>get_generation_property <property></code>
<code>get_list_of_ports <direction></code>	<code>get_interface_ports</code>
<code>set_module <name></code>	<code>set_module_property name <name></code>
<code>set_module_description <description></code>	<code>set_module_property description <description></code>
<code>set_port_direction_and_width <name> <direction> <width></code>	<code>set_port_property <name> direction <direction>;set_port_property <name> width <width></code>
<code>set_source_file <file></code>	<code>set_module_property top_level_hdl_file <file></code>
<code>get_project_property</code>	To determine device family, use the following commands: In the main program: <pre>add_parameter DEVICE_FAMILY string "unknown" set_parameter_property DEVICE_FAMILY SYSTEM_INFO device_family</pre> In the validation, generation callback read it using the following command: <pre>get_parameter_value DEVICE_FAMILY</pre>
Deprecated Module Properties	
<code>libraries</code>	Unnecessary
<code>class_name</code>	<code>name</code>
<code>module_file_name</code>	<code>top_level_hdl_file</code>
<code>preview_<n>_callback</code>	<code><n>_callback</code>
Deprecated Parameter Properties	
<code>affects_port_widths</code>	<code>affects_elaboration</code>
<code>GROUP</code>	Use <code>add_display_item</code> instead
Deprecated Generation Properties	
<code>language</code>	<code>hdl_language</code>

Document Revision History

Table 7-10 shows the revision history for this chapter.

Table 7-10. Document Revision History (Sheet 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2009, v9.1.0	<ul style="list-style-type: none"> ■ Tcl interpreter now stops upon error rather than continuing to interpret commands. ■ Changes <code>validation</code> and <code>elaboration</code> callbacks to be more similar. All commands permitted in the <code>validate</code> callback are also permitted in the <code>validation</code> callback. ■ Added the <code>package</code> command. ■ Added ability to optimize a component to wires or constant values using the <code>driven_by</code> argument. ■ Added parameter to set port width using an expression. ■ Changed syntax for setting the width of a <code>std_logic_vector</code>. ■ Renamed <code>TERMINATION_WIDE</code> <code>TERMINATION_VALUE</code> in the “get_port_properties” on page 7-32. ■ Corrected Example 7-11 on page 7-12. <code>set result = [...]</code> should be <code>set result [...]</code> ■ Added Table 7-9 listing deprecated commands and properties and their replacements. 	Added new functionality, a deprecated commands table, and corrected a few typographical errors.
March 2009, v9.0.0	<ul style="list-style-type: none"> ■ Added <code>add_display_item</code> commands. ■ Added <code>DISPLAY_HINT</code>, <code>IS_HDL_PARAMETER</code>, <code>DERIVED</code>, and <code>SYSTEM_INFO</code> parameters to Table 7-4 on page 7-21. Described <code>SYSTEM_INFO</code> parameter in detail. ■ Added <code>ENABLED</code> interface property to enable or disable an interface. ■ The <code>AFFECTS_PORT_WIDTHS</code> parameter has been renamed <code>AFFECTS_ELABORATION</code> to better reflect its function. ■ Added note saying that the <code>add_file</code> command will be restricted to the main and generation callbacks starting in version 9.1 of the Quartus II software. ■ Explained that before the elaboration phase, parameters may have values of 0 or -1 that are determined during HDL analysis. 	Added several new commands to increase functionality, clarified a few others, and corrected typographic errors.

Table 7-10. Document Revision History (Sheet 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008, v8.1	<ul style="list-style-type: none"> ■ Added <code>get_module_ports</code>, <code>get_interface_assignment</code>, <code>set_interface_assignment</code>, <code>get_module_assignment</code>, and <code>set_module_assignment</code> commands ■ Corrected availability to include more callbacks for several commands ■ Added two additional types for <code>add_parameter</code> command: <code>natural</code> and <code>positive</code> ■ Added brackets for some optional parameters ■ Changed <code>add_file</code> command for <code>SIMULATION</code> and <code>SYNTHESIS</code> in Example 7-10 to write to <code>\$outdir</code> ■ <code>get_project_property</code> is available in validation callback ■ Changed page size to 8.5 x 11 inches 	Added 5 new commands and corrected commands that did not define optional arguments or omitted some callback availability.
June 2008, v8.0.1	<ul style="list-style-type: none"> ■ Reformatted command information in tables. 	—
May 2008, v 8.0.0	<ul style="list-style-type: none"> ■ Added new Editing <code>_hw.tcl</code> commands and debug commands sections. ■ Changed chapter title from <i>Building a Component Interface with Tcl Scripting Commands</i> to <i>Component Interface Tcl Reference</i>. 	—
October 2007, v7.2.0	Major reorganization of chapter to better reflect work flow when using tcl scripting. Includes new commands, properties, and parameters.	—
May 2007, v7.1.0	Initial release.	—

