LatticeMico32 Hardware Developer User Guide

June 2012

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

Chapter 1

LatticeMico System Overview

This hardware developer guide describes the flow of tools involved in creating and configuring a hardware platform for the LatticeMico32 embedded microprocessor.

This guide is targeted to developers who are interested in learning the fundamentals of configuring and programming the embedded soft-core microprocessor. For a list of related documents on the LatticeMico32 microprocessor, refer to “Related Documentation” on page 5.

LatticeMico System Design Flow

This section lists the major steps involved in designing a LatticeMico32 embedded microprocessor. In addition to running the FPGA flow in Lattice Diamond, you use the integrated System software to build both hardware and software features of your embedded soft-core microprocessor.

The LatticeMico System is composed of three bundled applications:

- Mico System Builder (MSB)
- C/C++ Software Project Environment (C/C++ SPE)
- Debugger

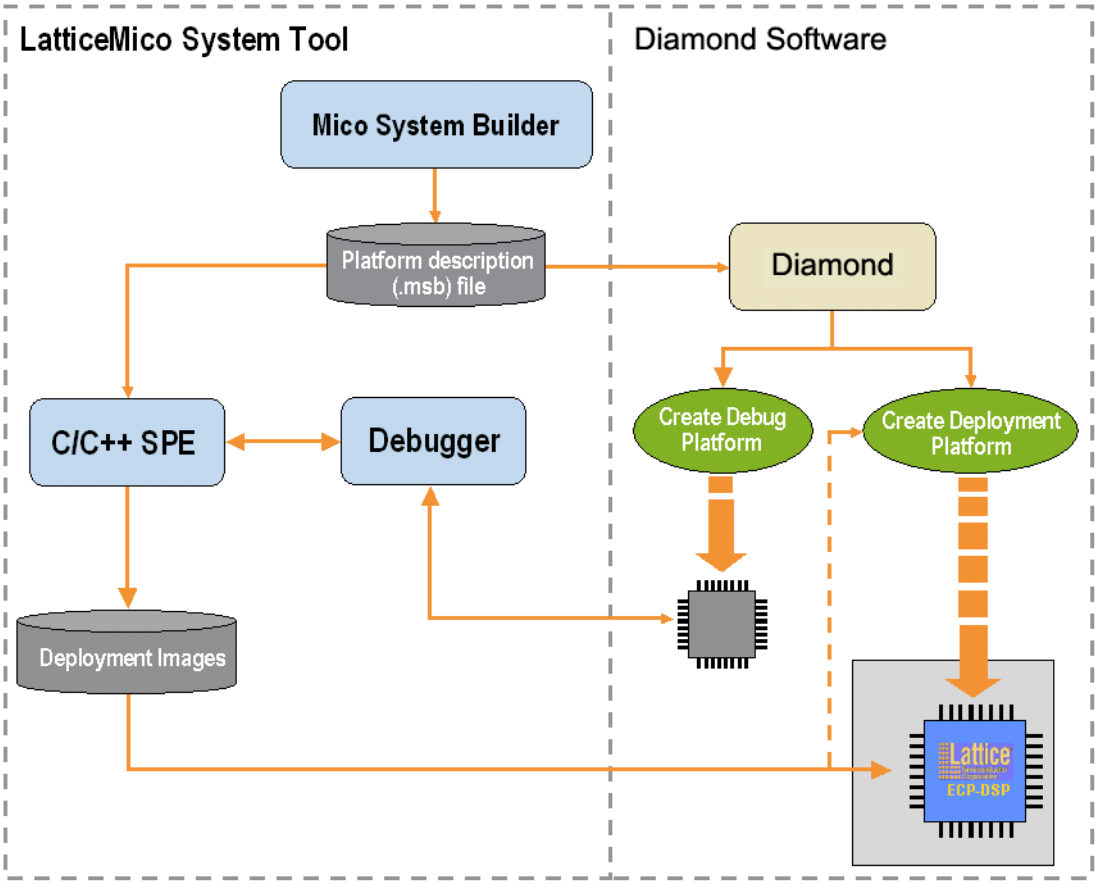
These applications work in the background through the user interface and can be accessed through different “perspectives” in the LatticeMico System software. Perspectives are a prearranged and predefined set of user functions that can be accessed within the software user interface. Perspectives are described in more detail in “LatticeMico System Perspectives” on page 9.

MSB is used by hardware designers to create the microprocessor platform for both hardware and software development. A platform generically refers to the hardware microprocessor configuration, the CPU, its peripherals, and how these components are interconnected. This functionality in the LatticeMico System software can be accessed by using the MSB perspective in the interface. The default MSB perspective is completely separate in terms of function from the other two perspectives.

You can use the C/C++ Software Project Environment (SPE) to develop the software application code that drives the platform. The Debugger is used to analyze and correct your code. You can access these programs by enabling the C/C++ and Debug perspectives, respectively. However, these two perspectives overlap in terms of functionality. Many of the same functions and views available in the C/C++ perspective are also available in the Debug perspective because the functions are so intertwined.

Figure 1 shows the interaction of the three LatticeMico System applications with Lattice Diamond in the microprocessor development design flow.

Figure 1: LatticeMico System Development Software Tool Flow



The *LatticeMico32 Tutorial* gives step-by-step instructions on creating a sample microprocessor platform, downloading hardware images to your device, creating your application code, and deploying your application code to on-chip or flash memory. It covers all relevant topics to enable you to run through a complete LatticeMico32 design flow. It is highly recommended that you start out with the tutorial.

Device Support

The Lattice FPGA devices that are currently supported in this design flow are the following:

- LatticeECP
- LatticeEC
- LatticeECP2
- LatticeECP2M
- LatticeECP3
- LatticeXP
- LatticeXP2
- LatticeSC
- LatticeSCM

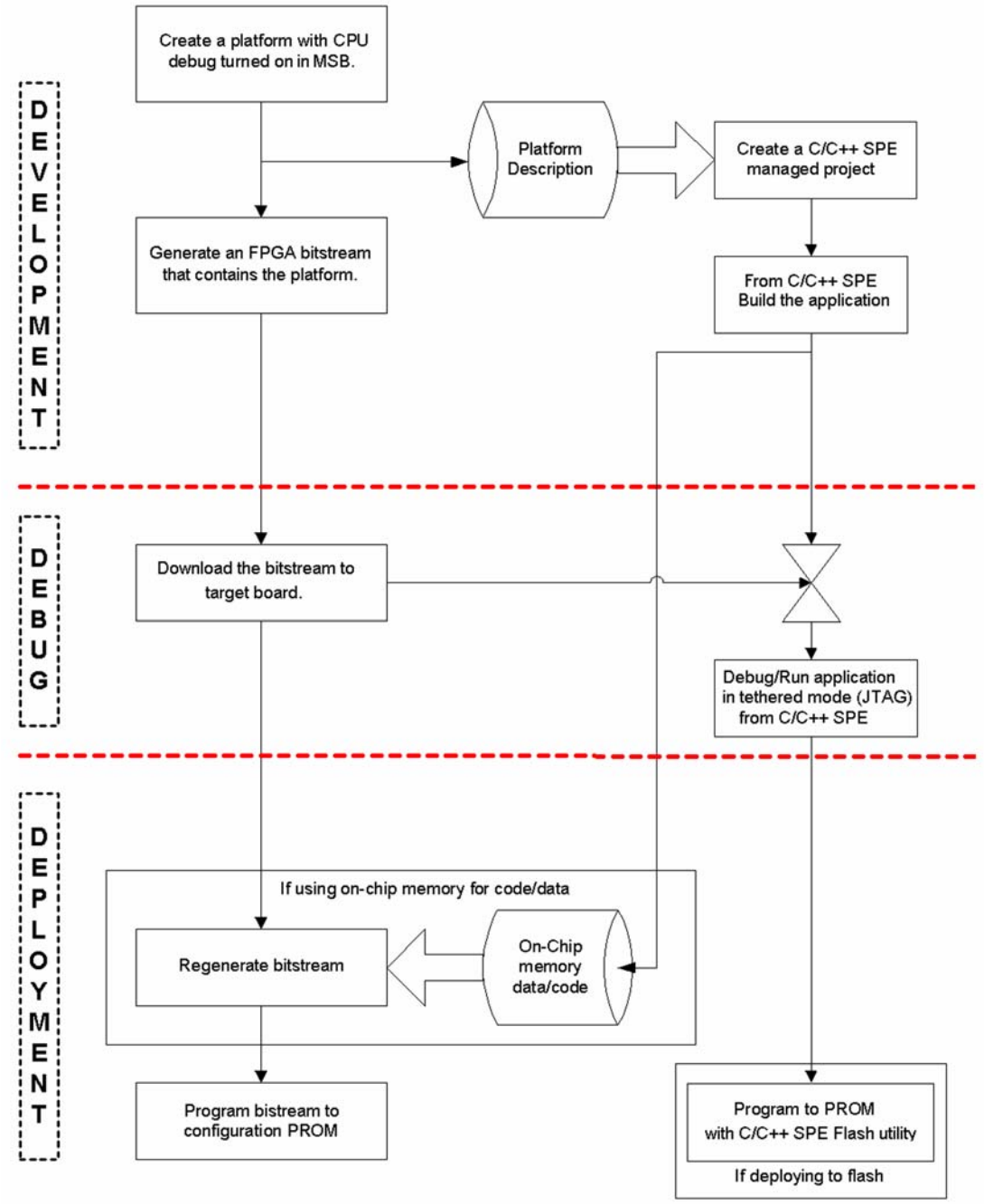
Design Flow Steps

The major steps involved in designing a LatticeMico32 soft-core microprocessor are the following:

1. Create a project in the Lattice Diamond software that targets the desired device family.
2. Use the Mico System Builder (MSB) in the LatticeMico System software to create and develop a microprocessor platform. You access this in the MSB perspective. Creating a platform involves generating an .msb file, selecting component peripherals, and connecting them to the LatticeMico32 platform.
3. In the MSB perspective, designate and develop drivers as necessary for available peripherals and add them to the platform you created.
4. In the MSB perspective, generate a platform build, which automatically creates a build structure with associated makefiles and an appropriate linker script. This process involves the device drivers and any other software components other than the user application.
5. In C/C++ SPE, use the C/C++ perspective to write the C/C++ user application software and build your application.
6. Using the Debugger in the LatticeMico System software, test your code on the target hardware, configure the target hardware, find issues with your code, and correct them. You access the Debugger in either the C/C++ perspective or the Debug perspective.
7. Using Diamond Programmer, download the executable code to on-board flash memory. You can deploy the application providing a boot loader that straps onto the application for loading the application from slow, non-volatile storage (flash memory device) to fast volatile storage (on-chip or off-chip RAM), without having to rebuild the application.
8. Repeat steps 3 through 7 for any new application development or modification to the platform in step 2.

Figure 2 shows the LatticeMico System design flow.

Figure 2: LatticeMico System Design Flow



For complete information about using the C/C++ SPE and Debugger perspectives to build and test your software application, refer to the *LatticeMico32 Software Developer User Guide*.

Related Documentation

You can access the LatticeMico System online Help and manuals by choosing **Help > Help Contents** in the LatticeMico System interface. These manuals include the following:

- *LatticeMico32 Processor Reference Manual*, which contains information on the architecture of the LatticeMico32 microprocessor chip, including configuration options, pipeline architecture, register architecture, debug architecture, and details about the instruction set.
- *LatticeMico32 Software Developer User Guide*, which introduces you to the run-time environment, the build management process, the directory structure for the managed build, information on the device driver framework, and the processes occurring in the background. It is intended for a programmer.
- *LatticeMico32/DSP Development Board User Guide*, which describes the features and functionality of the LatticeMico32/DSP development board. This board is designed as a hardware platform for design and development with the LatticeMico32 microprocessor, as well as for the LatticeMico8 microcontroller, and for various DSP functions.
- *Eclipse C/C++ Development Toolkit User Guide*, which is an online manual from Eclipse that gives instructions for using the C/C++ Development Toolkit (CDT) in the Eclipse Workbench.
- *LatticeMico Asynchronous SRAM Controller*, which describes the features and functionality of the LatticeMico asynchronous SRAM controller
- *LatticeMico DMA Controller*, which describes the features and functionality of the LatticeMico DMA controller
- *LatticeMico On-Chip Memory Controller*, which describes the features and functionality of the LatticeMico on-chip memory controller
- *LatticeMico Parallel Flash Controller*, which describes the features and functionality of the LatticeMico parallel flash controller
- *LatticeMico GPIO*, which describes the features and functionality of the LatticeMico GPIO
- *LatticeMico Master Passthrough*, which describes the features and functionality of the LatticeMico master passthrough.
- *LatticeMico Slave Passthrough*, which describes the features and functionality of the LatticeMico slave passthrough
- *LatticeMico SDR SDRAM Controller*, which describes the features and functionality of the LatticeMico SDR SDRAM controller
- *LatticeMico SPI*, which describes the features and functionality of the LatticeMico serial peripheral interface (SPI)
- *LatticeMico SPI Flash*, which describes the features and functionality of the LatticeMico serial peripheral interface (SPI) flash memory controller
- *LatticeMico Timer*, which describes the features and functionality of the LatticeMico Timer
- *LatticeMico UART*, which describes the features and functionality of the LatticeMico universal asynchronous receiver-transmitter (UART)

- *Diamond <release_number> Installation Notice*, which explains how to install the LatticeMico System software for the current release
- *LatticeECP/EC FPGA Family Handbook*, which is a collection of the data sheets and application notes on LatticeEC and LatticeECP devices
- *LatticeECP/EC Family Data Sheet*
- *LatticeECP2 FPGA Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP2 devices
- *LatticeECP2 Family Data Sheet*
- *LatticeECP2M Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP2M devices
- *LatticeECP2M Family Data Sheet*
- *LatticeECP3 FPGA Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP3 devices
- *LatticeECP3 Family Data Sheet*

Chapter 2

Using the LatticeMico System Software

This chapter introduces you to the LatticeMico System software, describes portions of its software user interface, and provides in-depth procedures for performing common and advanced

user tasks. The instructions for performing key operations are presented in the order that they occur in the design flow, and the user interface is introduced appropriately. See the LatticeMico System online Help for more details on the user interface.

This chapter assumes that you have read “LatticeMico System Overview” on page 1 and are familiar with the general high-level steps in this product flow. This chapter also assumes that you have not customized the user interface.

LatticeMico System Software Overview

This section provides a brief synopsis of the functional tools included in the software and teaches you the basic concept of user “perspectives” in the software that are designed to simplify access to command functionality.

About the LatticeMico System Tools

As noted in “LatticeMico System Overview” on page 1, the LatticeMico System software is composed of the following bundled, functional software tools:

- Mico System Builder (MSB), which is used to create the microprocessor platform
- C/C++ Software Project Environment (C/C++ SPE), which is used to create the software application code that drives the microprocessor platform
- Debugger, which enables you to analyze the software application code to identify and correct errors

The LatticeMico32 tools share the same Eclipse workbench, which provides a unified graphical user interface for the software and hardware development flows. You use MSB to define the structure of your microprocessor or your hardware platform. C/C++ SPE enables you to develop and compile your code in a managed and well-structured build environment. The Debugger includes tools that analyze your code for errors and simulates instruction calls within the software environment or to an actual programmed device on a circuit board.

You will learn more about how these functions are encountered in the software throughout this chapter. This chapter assumes that you have installed all of the necessary software and have not modified your default perspectives in any way.

LatticeMico System Requirements

For information about LatticeMico System's system requirements on the Windows operating system, see the "Installing LatticeMico32 Development Tools" chapter of the *Diamond <release_number> Installation Notice for Windows* for the current release on the Lattice Semiconductor Web site.

For information about LatticeMico System's system requirements on the Red Hat Linux operating system, see the "Installing LatticeMico32 Development Tools" chapter of the *Diamond <release_number> Installation Notice for Linux*, available on the Lattice Semiconductor Web site and the LatticeMico32 online Help.

For information on installing Diamond, see the *Diamond <release_number> Installation Notice for Windows* or the *Diamond <release_number> Installation Notice for Linux* for the current release.

Running LatticeMico System

Now you will run the software so that you can take a quick survey of the user interface to understand its basic structure.

To run the LatticeMico System from your PC desktop:

- From the Windows desktop Start menu, choose **Start > Programs > Lattice Diamond > Accessories > LatticeMico System**.

The LatticeMico System interface initially opens with the MSB perspective active by default, as shown in Figure 5 on page 16. After that, the software opens to the last opened perspective.

LatticeMico System Perspectives

Before you begin learning about the basic tasks that you can perform in the LatticeMico System software, it is important to understand the concept of “perspectives” in the software and how to access the three integrated functional tools, MSB, C/C++ SPE, and the Debugger, within the user interface. Do not confuse the underlying functional tools in the LatticeMico System software with the various perspectives in the user interface.

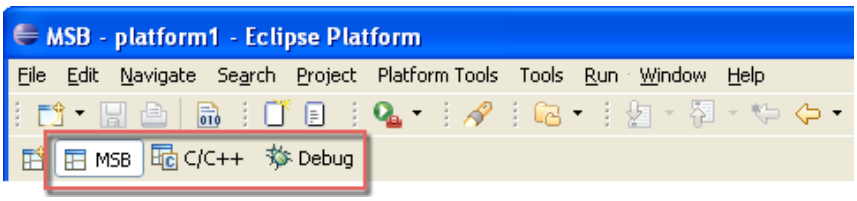
There are three default perspectives in the LatticeMico System software:

- MSB perspective
- C/C++ SPE perspective
- Debug perspective

Within the Eclipse framework, the three functional tools appear as different user interfaces integrated into the same framework. A “perspective” in the LatticeMico System software is a separate combination of views, menus, commands, and toolbars in a given graphical user interface window that enables you to perform a set of particular, predefined tasks. For example, the Debug perspective has views that enable you to debug the programs that you developed using the C++ SPE tool. For an overview on Eclipse workbench concept and terminologies, refer to the *Eclipse Reference Manual*.

When you first open LatticeMico System, the MSB perspective is the active perspective by default. After working in the interface, the software defaults to the last opened perspective. The Eclipse workbench that is integrated into the LatticeMico System software has three activation buttons for quickly toggling back and forth between the MSB, C/C++, and Debug perspectives. These buttons are shown in Figure 3. They enable you to switch between perspectives by clicking on them. The current active perspective is displayed in the upper left of the window’s title bar.

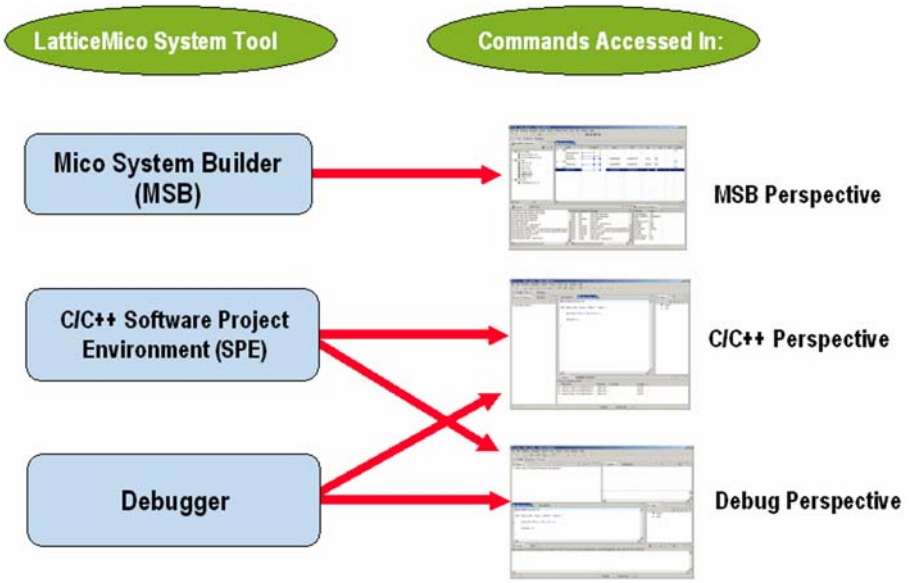
Figure 3: Perspective Activation Buttons



The three different perspectives—the MSB, the C++ SPE, and the Debug—include overlapping tool functions that you access through various commands and interactive views, as illustrated in Figure 4. You can find more information on these commands and views later in this document and in the online Help.

In Figure 4, the C/C++ perspective and the Debug perspective arrows indicate that they share many of the same or similar command functions, so you can perform the same exact operation in either perspective. By default, these two perspectives share many functions because these tasks are very closely related to each other. If you perform some changes in a view such as the Editor view in one perspective, it will affect what you see in another

Figure 4: Tool Functions Accessed in Perspectives



perspective that contains the same view. Do not assume that a given command function in the LatticeMico System is only accessible or viewable from one perspective.

Note

Particular views and options within the MSB perspective are described throughout this chapter as they are encountered in the design flow. For descriptions of the C/C++ SPE and Debugger perspectives, refer to the *LatticeMico System Software Developer User Guide*. More information about the graphical user interface for each perspective is described in more detail in the LatticeMico32 online Help.

The LatticeMico System software enables you to customize existing default perspectives, create your own perspectives, and control what views are open in a given perspective. The following procedures tell you how to customize, define, and reset perspectives. These procedures assume that you have not changed the default perspective settings.

Customizing Default Perspectives

It is possible to customize existing default perspectives in LatticeMico System by changing the existing set of commands ascribed to each perspective.

To customize an existing perspective:

1. From within a given perspective, choose **Window > Customize Perspective**.
2. In the Customize Perspective dialog box, select shortcut options in the Shortcuts tab and command options in the Commands tab.
3. Click **OK**.

You should see the results of any changes in the interface.

Creating Custom Perspectives

In addition to the three existing default perspectives, you can also add your own custom perspective with custom options to the user interface.

To create a new perspective:

1. From within a given perspective, choose **Window > Save Perspective As**.
2. In the Save Perspective As dialog box, rename an existing default perspective in the Name text box and click **OK** to save it.
3. Choose **Window > Customize Perspective** to customize the new perspective that you created.

Deleting Custom Perspectives

You can delete perspectives that you defined yourself, but you cannot delete the default perspectives that are delivered with the software workbench environment.

To delete a custom perspective:

1. From within a given perspective, choose **Window > Preferences**.

   The Preferences window opens.
2. From the Preferences window, expand the General category on the left and select **Perspectives**.

   The Perspectives preferences page opens.
3. From the Available perspectives list, select the desired perspective and click **Delete**.
4. Click **OK**.

Changing Default Perspectives

After you create a new perspective, you may want to make the new perspective a default perspective that will automatically be available when you return to the program.

To change the default perspective:

1. From within a given perspective, choose **Window > Preferences**.
2. From the Preferences window, expand the General category on the left and select **Perspectives**.

   The Perspectives preferences page opens.

3. Select the perspective that you want to define as the default and click **Make Default**.

   The default indicator moves to the perspective that you selected.

4. Click **OK**.

Resetting Default Perspectives

After customizing default perspectives, you can revert back to the original set of command options for a given perspective by resetting them in the software.

To reset your default perspectives:

1. From within a given perspective, choose **Window > Reset Perspective**.
2. In the Reset Perspective pop-up dialog box, click **OK**.

   This action returns all default perspectives back to their original option settings.

Closing and Opening Views in Perspectives

In each perspective, views are defined for each perspective that allow you to interactively perform a task. These views are described later in this chapter for each perspective.

At times, you may want to close views to make more space for working in a desired view. For example, after you add all of the components that you need in your platform, you may opt to close the Available Components view in the MSB perspective.

To close a view in a given perspective:

- In a given perspective, click on the **Close** icon that appears as an “X” at the upper right corner of the view that you wish to close.

  The view closes. In some cases where the two views did not overlap, an adjacent view moves into the vacated area in the interface, making the adjacent view larger.

To reopen a view that you previously closed:

- In a given perspective, choose **Window > Show View** and select the view that you wish to reopen from the submenu.

  The view is reopened in its original area in the interface.

Setting Up Diamond for a LatticeMico32 Platform

Before you create your microprocessor project in LatticeMico System, set up a project in the Lattice Diamond software that targets the device family that will serve as the fabric in which to embed the microprocessor. You do not add your source HDL at this point, because your Verilog or VHDL source will be generated by the LatticeMico System software later in the flow.

Note

If you are going to use LatticeMico System on the Linux platform, you must install a stand-alone synthesis tool, such as Synplicity® Synplify Pro®, before you create an Diamond project.

In addition, your Linux system must meet the minimum system requirements outlined in the *Diamond <release_number> Installation Guide for Linux*.

Creating a New Diamond Project

After you create a new Diamond project, you can import a LatticeMico32 platform into the design. If your design includes a platform with IP cores, you should also follow the guidelines in “Recommended IP Design Flow” on page 14.

To create a new Diamond project for use with a LatticeMico32 project:

1. Start the Lattice Diamond software:
   - On the Windows desktop, choose **Start > Programs > Lattice Diamond > Lattice Diamond**.
   - On the Linux command line, run the following script:

     <*Diamond_install_path*>/bin/**ispgui**.
2. Choose **File > New > Project**.
3. In the New Project wizard, click **Next**.
4. Type a name for the project in the Name box.
5. Click the **Browse** button and navigate to the directory where you would like the project to be stored.
6. Under Implementation, the project name and location are automatically filled in. If you prefer a different name for the design’s first implementation, type a new name in the Implementation name box.
7. Click **Next**.
8. Click **Next** in the Add Source dialog box. You will be adding the source files later.
9. In the Select Device dialog box, select the desired family, device, speed grade, package type, operating conditions, and part name from the drop-down menus. Leave the Show Obsolete Devices box unselected.
10. Click **Next** and review the project information. Use the Back button, if needed, to make any modifications.

11. Click **Finish**.

    The name of the new project appears in the File List pane. The initial strategy and implementation for the project are displayed in bold type. For more information about working with design implementations and strategies, see the “Managing Projects” section of the Lattice Diamond online Help.

Recommended IP Design Flow

The following design flow and guidelines will ensure that the proper data gets passed between Diamond and LatticeMico32 for platforms that contain IP cores. This procedure assumes that you are creating a new project and platform and that you will be generating an IP core from the IPexpress interface within LatticeMico System.

1. From the Windows Start Menu, choose **Programs > Lattice Diamond > Accessories > LatticeMico System**.

   LatticeMico System opens with the Mico System Builder (MSB) perspective. MSB displays the last platform that was opened. If you closed the platform before exiting MSB in the previous session, it displays no platform.

2. Choose **File > New Platform**.

   The New Platform Wizard opens. In the Directory text box, it displays the path and directory of your Diamond project.

3. Give the new platform a name and specify the settings, as described in “Creating a Platform Description in MSB” on page 17. To keep the platform within the Diamond project you just created, do not change the directory location.

4. Add the LatticeMico32 Processor to the platform and any desired memory and peripheral components, as described in “Adding Microprocessor and Peripherals to Your Platform” on page 19.

5. From the Available Components window, double-click the desired IP core component—for example, Tri-Speed Ethernet Mac—to open the Add<*IP_core*> dialog box.

   As in the New Platform Wizard, this dialog box remembers the path and directory of your Diamond design project, and it displays this path and directory in the “Diamond Project” text box in the “IPexpress Interface” section.

   When you generate the IP core, the software places a copy of the IP core’s .ngo file—for example, ts_mac_core.ngo—inside the project directory. If you click **Browse** and change the location, any future changes that you make to the IP core will not be applied to the current project.

6. Specify the desired settings in the top part of the Add<*IP_core*> dialog box. In the IPexpress Interface section, do not change to a different Diamond project directory.

This is important for your current project. Maintaining the Diamond project directory will ensure that future changes to the IP will be applied to the current design project.

7. In the IPexpress Interface section, click **Launch IPexpress**.
8. In the Lattice IP Core interface, specify the desired parameters, and then click **Generate**.

   IPexpress generates the IP core. When the process has finished, it displays a log, which shows the output directory and path and the files generated.
9. Click **Close** to return to the Add<*IP_core*> dialog box.

   The Generated NGO File text box is now populated with the location of the .ngo file inside the Diamond project directory.
10. Click **OK** to add the newly generated IP core to your project's platform.
11. Follow the remaining instructions in the section "Creating the Microprocessor Platform in MSB" on page 15 to connect master and slave ports, assign addresses and interrupt priorities, and generate the platform.

Creating the Microprocessor Platform in MSB

After you have created a new project in Diamond using your target FPGA device, you must create a new microprocessor platform in Mico System Builder (MSB). A platform generically refers to the hardware microprocessor configuration, the CPU, its peripherals, and how these components are interconnected.

Starting MSB

Note

If you are going to be using LatticeMico System on the Linux platform, set up the environment to point to the location where the stand-alone synthesis tool is installed before starting LatticeMico System, as in this example:

```
setenv IPEXPRESS_SYN_PATH /install/synplify/fpga_89/bin/
synplify_pro
```

To start MSB:

1. If you have not yet opened the software, as described in "Running LatticeMico System" on page 8, choose **Start > Programs > Lattice Diamond > Accessories > LatticeMico System**.

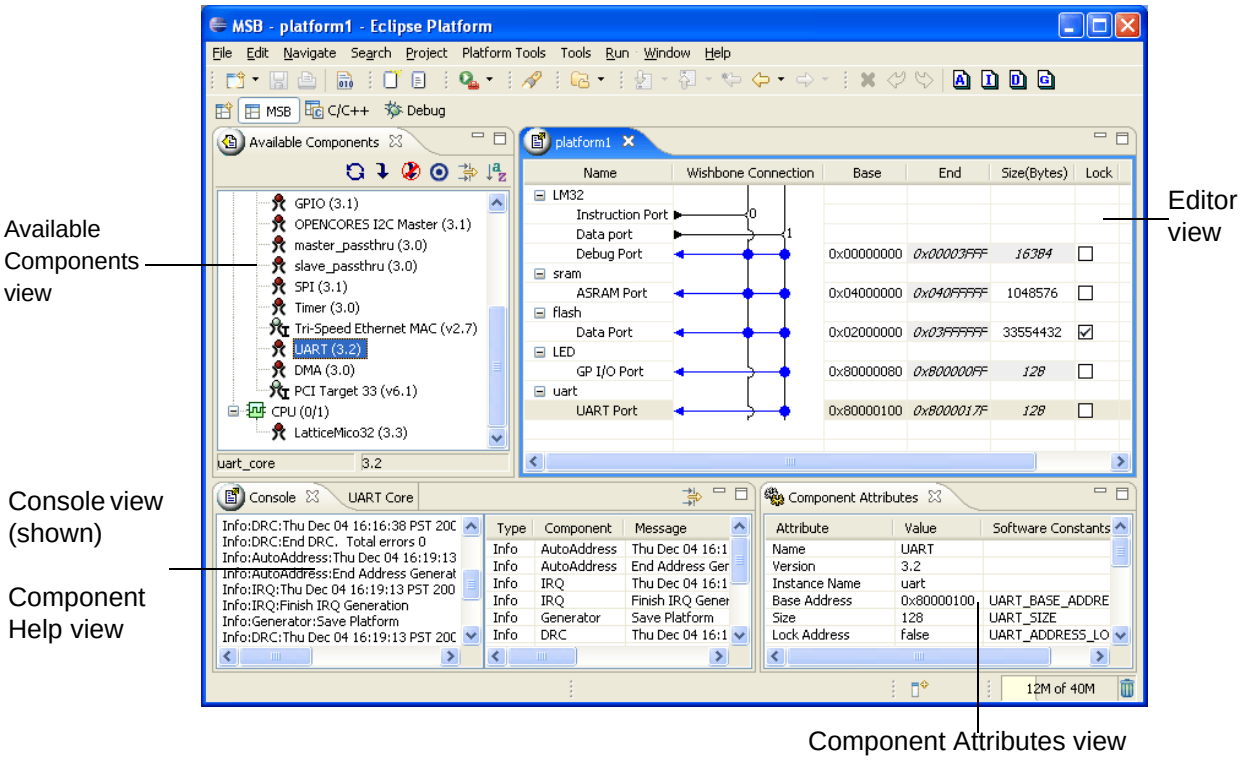
   During its launch process, the LatticeMico System software creates an Eclipse workspace file. This file is created in your home directory. On the Windows operating systems, it is in the Documents and Settings directory. On the Linux operating system, it is in ~.

Eclipse uses the workspace file to store information about your Eclipse environment and the projects that you have been working on. You can switch workspaces by selecting the File > Switch Workspace command.

2. In the upper left-hand corner of the graphical user interface, select **MSB**, if it is not already selected, to open the MSB perspective.

   The MSB perspective is active by default, as shown in Figure 5.

Figure 5: MSB Perspective



The MSB perspective consists of the following views:

- Available Components view, which displays all the available components that you can use to create the design:
  - List of hardware components: microprocessors, memories, peripherals, and bus interfaces. Bus interfaces can be masters or slaves. The component list shown in Figure 5 on page 16 is the standard list that is given for each new platform.

    You can double-click on a component to open a dialog box that allows you to customize the component before it is added to the design. The component is then shown in the Editor view.
- Editor view, which is a table that displays the current platform definition from the components that you have chosen in the Available Components view. It includes the following columns:

- Name, which displays the names of the chosen component and their ports
- Connection, which displays the connectivity between master and slave ports
- Base, which displays the start addresses for components with slave ports. This field is editable.
- End, which displays the end addresses for components with slave ports. This field is not editable. The value of the end address is equivalent to the value of the base address plus the value of the size.
- Size, which displays the number of addresses available for component access. This field is editable for the LatticeMico32 on-chip memory controller and LatticeMico32 asynchronous SRAM controller components only.
- Lock, which indicates whether addresses are locked from any assignments. If you lock a component, its address will not change when you select Platform Tools > Generate Address.
- IRQ, which displays the interrupt request priorities of all components that have an interrupt line connected to the microprocessor. It is not applicable to memories.
- Disable, which excludes a component from a platform definition. It can be toggled on and off.

- Component Help view, which displays information about the component that you selected in the Available Components view. This view is also called “About <*Component_name*>,” for example, “About Timer” or “About UART.”
- Console view, which displays informational and error messages output by MSB
- Component Attributes view, which displays the features, parameters, and values of the selected component. This view is read-only.

Clicking the “X” icon next to the View title closes the selected view. To reopen a view that you previously closed, choose **Window > Show View** and the desired view submenu option. For a detailed explanation of the available views, refer to the LatticeMico32 online Help.

Creating a Platform Description in MSB

After you have created a new project in Lattice Diamond, you must create a new microprocessor platform description in Mico System Builder (MSB). A platform generically refers to the hardware microprocessor configuration that includes the CPU component, its peripheral components, and the interconnectivity between them.

You must perform two major steps in MSB to create a new platform: create an .msb file and add your components to the file.

Creating a Platform Description File

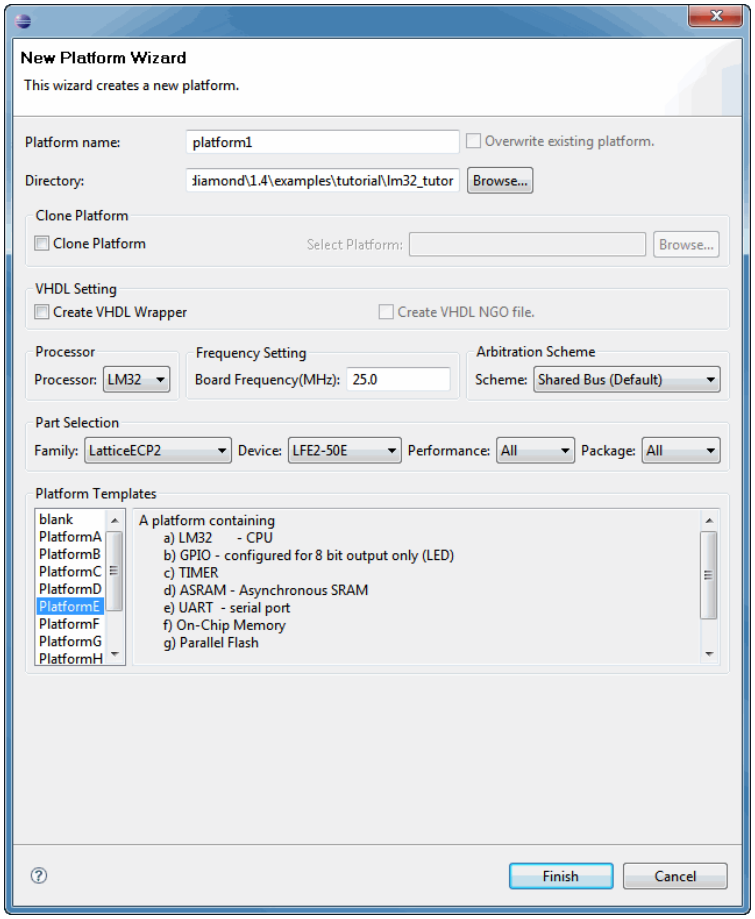
The first step in creating a new platform is to use MSB to create an .msb file. This file will eventually contain a complete definition of your microprocessor platform.

To create a new microprocessor .msb file:

1. In the MSB perspective, choose **File > New Platform**.

   The New Platform Wizard dialog box now appears, as shown in Figure 6.

Figure 6: New Platform Wizard Dialog Box



2. In the New Platform Wizard dialog box, enter the name of the platform in the Platform Name box.
3. In the Directory box, browse to the folder in which you want to store your platform files and click **OK**.
4. If the design that will incorporate this platform is in pure Verilog code, leave **Create VHDL Wrapper** unselected.

   If the design that will incorporate this platform is in mixed Verilog/VHDL, do the following.

   a. Select **Create VHDL Wrapper**.

b. If you want to continue using the NGO flow, select **Create VHDL NGO File**. Otherwise, leave this option cleared.

5. In the Board Frequency box, enter the board frequency.
6. In the Arbitration Scheme box, select the desired arbitration scheme from the pull-down menu.
7. In the Device Family section, select a Lattice family and a device from the pull-down menus.
8. If you want to duplicate the platform, select **Clone Platform**, and then browse to the platform description (.msb file) that you want to duplicate.

   The Clone Platform option is useful if your platform contains several peripherals and you want to retain them but experiment by modifying their settings. When you select this option, the Platform Templates and the Description boxes are no longer available, but the Select Platform option becomes available.

   Warning!

   If you are cloning a platform that contains IPs and you select a different device family, you will need to rerun IPexpress for the IPs in the platform. If you do not rerun IPexpress, you might encounter problems during synthesis.

9. If you have not selected Clone Platform, select the desired template from the Platform Templates list; or select Blank for a new template.
10. Click **Finish**.

    You now have created an .msb file. This file will hold the contents of your platform: a CPU, its peripherals, and the interconnections between them. Currently, the platform description contains no components. You will add components in the following procedures.

Adding Microprocessor and Peripherals to Your Platform

In the MSB perspective, you can add CPU and peripheral component definitions to your hardware platform. These definitions are added to the .msb file, which is currently empty if you did not select a template or duplicate a platform. The microprocessor and its peripherals are called components throughout this document.

Note

If you installed LatticeMico System without installing Diamond, you cannot include in the platform any PLLs or any IPs, which are components that you download from IPexpress. In addition, you cannot generate a VHDL wrapper for the platform. If you want to perform these functions, you must install LatticeMico32 with the Diamond software. See the references given in “LatticeMico System Requirements” on page 8 for information on installing Diamond and LatticeMico System.

For information on creating your own custom components to add to your platform, see See “Creating Custom Components in LatticeMico System” on page 51.

To add the LatticeMico32 microprocessor to the design:

1. Double-click on the **LatticeMico32** component listed under CPU in the Available Components view. If you want to see information about it before you place it in the Editor view, click it once.
2. Set the options in the Add LatticeMico32 dialog box and click **OK**.

LatticeMico System provides several peripheral components, I/Os, and memories that you can add to your microprocessor design structure. For example, some available peripherals include UART, a timer, an asynchronous SRAM controller, a GPIO, and a parallel flash component. In the MSB perspective, you can view all of the component types that you can add in the Available Components view. To aid in selection and option settings, you can view a complete description of each available component type. See “Accessing Component Help and Data Sheets” on page 20 for instructions.

To add a peripheral component to the design:

1. Double-click on the component in the Available Components view, set any options in the dialog box that appears, and click **OK**.
2. After you have added the last peripheral, specify the connections between the master and slave ports by clicking on the appropriate rounded endpoints in the Connection column, as described in “Connecting Master and Slave Ports” on page 21.

Accessing Component Help and Data Sheets

For each component that you can add to your platform, LatticeMico System provides a short online Help topic that describes its user-configurable parameters, as well as a complete data sheet that describes the detailed features and operations of the component. The Show View command enables you to view the appropriate Help topic in a separate view each time that you select a component in the Available Components view.

To view the online Help for a particular component:

1. In the MSB perspective, choose **Window > Show View > Component Help**.

   The Component Help view opens in a separate window.
2. In the Available Components view, select the desired component.

   The appropriate component topic appears in the Component Help view.

To view the data sheet for a component:

- In the Component Help view, click on the document icon  to view a complete description of a given component.

To quickly maximize the Component Help view, press **Ctrl+M**. Press Ctrl+M again to return to the previous size.

Connecting Master and Slave Ports

The LatticeMico32 CPU component acts as the master to the peripheral slave components that are attached to the bus structure, allowing it to have unidirectional control over those devices.

Only certain components, such as the LatticeMico32 processor and the LatticeMico32 DMA controller, have master ports. A master port can initiate read and write transactions. A slave port cannot initiate transactions but can respond to transactions initiated by a master port if it determines that it is the targeted component for the initiated transaction.

- A master port can be connected to one or more slave ports.
- A component can have one or more master ports, one or more slave ports, or both.

Attached to one or more slave ports, master port signals initiate read and write transactions that are communicated to the targeted slave device, which in turn responds appropriately. Generally, a component can have one or more master ports, one or more slave ports, or both.

Arbitration Schemes

The connections that MSB makes depend on which arbitration scheme you choose while creating the platform.

Shared-Bus Arbitration MSB automatically generates a central arbiter when it generates the microprocessor platform to allow multiple master ports access to multiple slave ports over a single shared bus.

Figure 7 shows the connections made by MSB when the shared-bus arbitration scheme is chosen.

Each master port connected to the arbiter has priority of access to the slave ports. In the case of simultaneous access requests by multiple master ports, the highest-priority master port is granted access to the bus. Master ports have default priorities assigned in their components' .xml files when you add the components to the platform. The master ports of the LatticeMico32 processor have defaults of 0 and 1. The master ports of the DMA controller have defaults of 2 and 3. However, you can change these priorities by selecting Platform Tools > Edit Arbitration Priorities and changing the priorities in the Edit Arbitration Priorities dialog box. When you perform a DRC check, MSB checks the validity of the priorities that you have changed.

Slave-Side Arbitration Figure 8 shows the connections made by MSB when the slave-side arbitration scheme is chosen.

Two types of slave-side arbitration are available: slave-side and round-robin.

Figure 7: Connections Made by MSB for Shared-Bus Arbitration

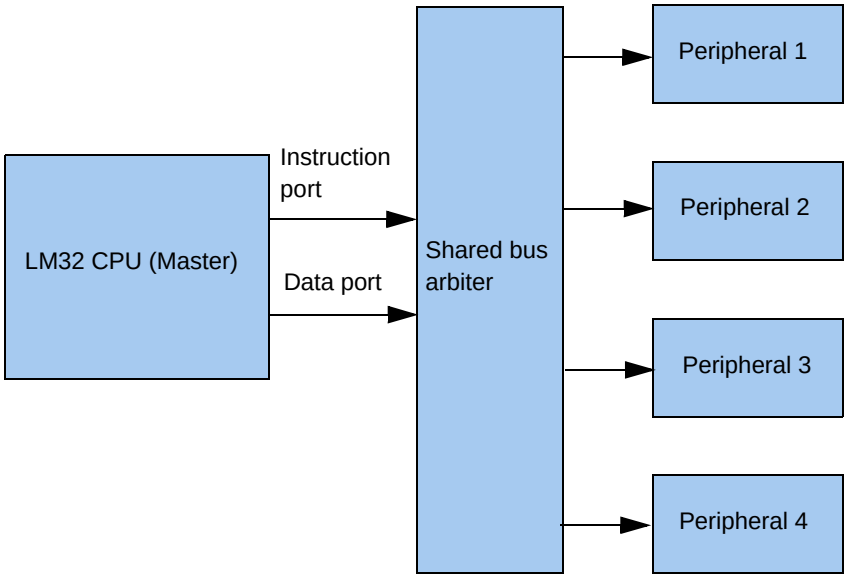
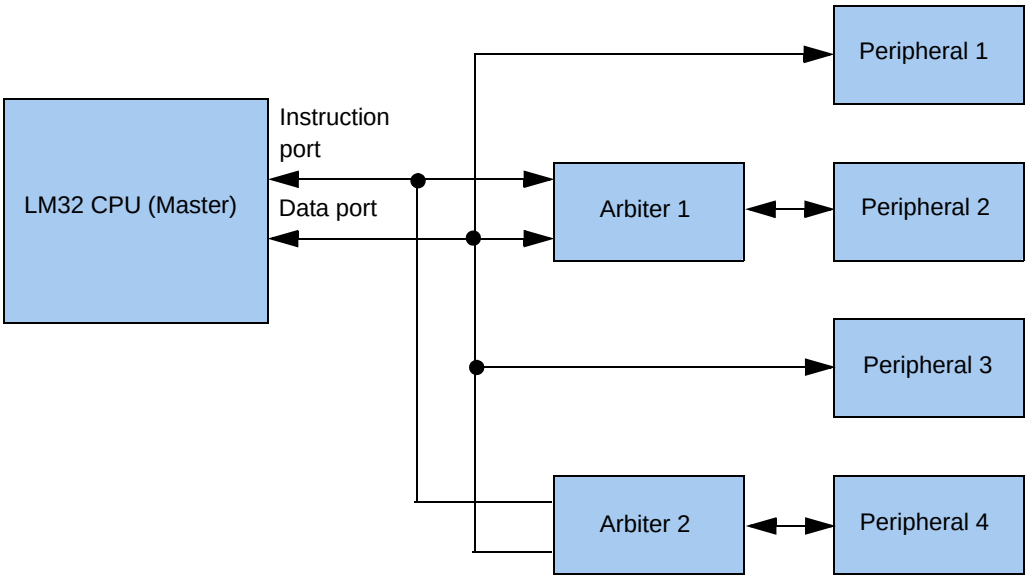


Figure 8: Connections Made by MSB for Slave-Side Arbitration



Slave-Side Fixed Arbitration The slave-side fixed arbitration scheme enables multiple masters to access multiple slaves at the same time. In this scheme, each multi-master slave has one arbiter. Arbitration between different master ports occurs at the slave side. This scheme enables multiple master ports to obtain access to multiple slave ports, as long as they do not try to access the same slave at the same time.

Each master port connected to the arbiter has priority of access to the slave ports. In the case of simultaneous access requests by multiple master ports, the highest-priority master port is granted access to the slave. Master ports

have default priorities assigned in their components' .xml files when you add the components to the platform. Since each multi-master slave has its own arbiter in this scheme, arbitration priorities are assigned per slave. However, you can change these priorities by selecting Platform Tools > Edit Arbitration Priorities and changing the priorities in the Edit Arbitration Priorities dialog box. When you perform a DRC check, MSB checks the validity of the priorities that you have changed.

Slave-Side Round-Robin Arbitration The slave-side round robin arbitration scheme is similar to the slave-side fixed arbitration scheme in that each multi-master slave has one arbiter, but all masters have the same priority. The arbiter grants access to all the masters that request a slave in a round-robin, or circular, fashion. Once the requesting master is finished with its transfer, the next master obtains access to the slave.

In the slave-side round-robin scheme, the Platform Tools > Edit Arbitration Priorities command is not available.

Comparing the Arbitration Schemes

The difference between the slave-side fixed arbitration scheme and the slave-side round-robin arbitration scheme is how the arbiter grants requesting masters access to the bus. The slave-side fixed scheme always gives the highest-priority master access to the bus if that master requests it. The slave-side round-robin scheme grants masters access to the bus in a round-robin fashion.

Both the slave-side fixed and the slave-side round-robin arbitration schemes use separate arbiters for each multi-master slave, so the area of the platforms generated with these schemes is slightly larger than that resulting from the shared-bus arbitration scheme. For example, for a typical system consisting of four multi-master slaves, the slave-side fixed and the slave-side round-robin schemes require four arbiters, but the shared-bus scheme requires only one arbiter. The area required by the system with the slave-side arbitration schemes is approximately three times more than the area required by the system with the shared-bus arbitration scheme.

The slave-side arbitration schemes offer better performance than the shared-bus arbitration scheme. For example, the SoC used in this topic (a CPU with a DMA controller) yields better performance with a slave-side arbitration scheme than with a shared-bus arbitration scheme. When a slave-side arbitration scheme is used in this SoC, the DMA controller's read and write ports can work in parallel and transfer the data from the external SRAM memory to on-chip memory. When a shared-bus arbitration scheme is used in the SoC, data cannot be transferred in parallel because there is a single arbiter for both memories.

Whether you select a slave-side fixed or slave-side round robin arbitration scheme depends on the application. If the application requires each master to have equal access to a slave, the slave-side round-robin scheme is a better option. If the application requires a certain master to have access to a slave as soon as the current master is finished with the data transfer, the slave-side fixed scheme is the best option.

Specifying Connections Between Master and Slave Ports

You interactively make your master/slave connections between these ports in the Editor view by clicking on those connection line endpoints and then by saving your project. The .msb file is updated with this information. Figure 9 on page 25 illustrates the basic structure of this connection between the master and the slave.

To specify the connections between master and slave ports:

1. Ensure that you have first added your desired components and that they appear in the Editor view in the MSB perspective.
2. If you want to select a different arbitration scheme, choose **Platform Tools > Properties**, select the desired arbitration scheme from the pull-down menu in the Arbitration Scheme box, and click **OK**.
3. In the Editor view's Connection column, for each listed slave component, click on the blue-outlined, rounded endpoint to complete the connection to the CPU's master ports. The rounded endpoint now appears filled in; that is, it turns solid blue, indicating that the slave is "connected" to the master port.

   Connection points occur at the intersection of the vertical lines down from the master at the slave horizontal lines and coincide with a desired connection to master instruction, data ports, or both. You may or may not wish to connect to both master ports, depending on the necessary input on a given slave component.

   For example, suppose that a CPU's master ports are composed of an instruction port and a data port. You want to connect the CPU's instruction port, but not its data port, to a UART's slave port. You would go to the Connection column in the UART row and click on the outline circle linked to the instruction port to fill it in, but not on the outlined circle linked to the data port.
4. Choose **File > Save** or click the **Save** toolbar button.

   The connections that you made are saved in the .msb file.

Figure 9 shows an example of the connections that result in the Editor view when the shared-bus arbitration scheme is used. All master signal connection lines are represented in black, and all slave connection lines are represented in blue.

Figure 10 shows an example of the connections that result in the Editor view when the slave-side fixed and slave-side round-robin arbitration schemes are used.

In the slave-side fixed arbitration scheme, you can change the priorities of the master ports, so the Edit Arbitration Priorities command is available on the Platform Tools menu, as shown in Figure 11. However, in the slave-side round-robin arbitration scheme, you cannot change the priorities of the master ports because the arbitration between the master ports occurs in a round-robin fashion. The Edit Arbitration Priorities command on the Platform Tools

Figure 9: Connecting Master/Slave Ports in Editor View in Shared-Bus Arbitration Scheme

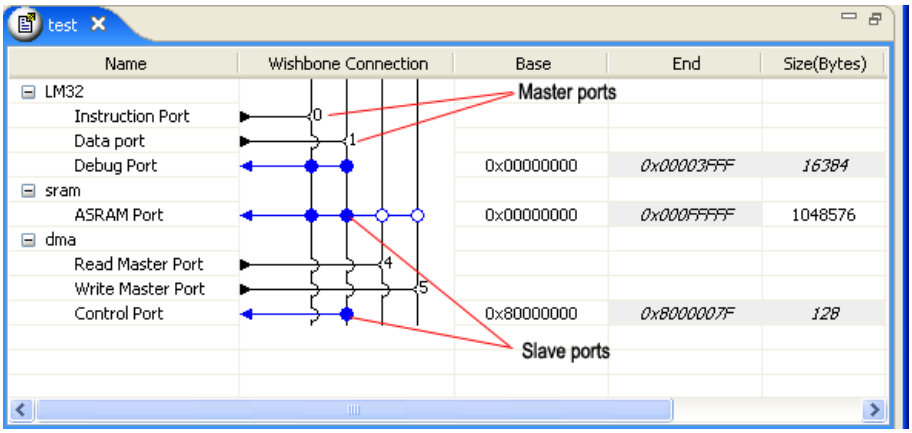
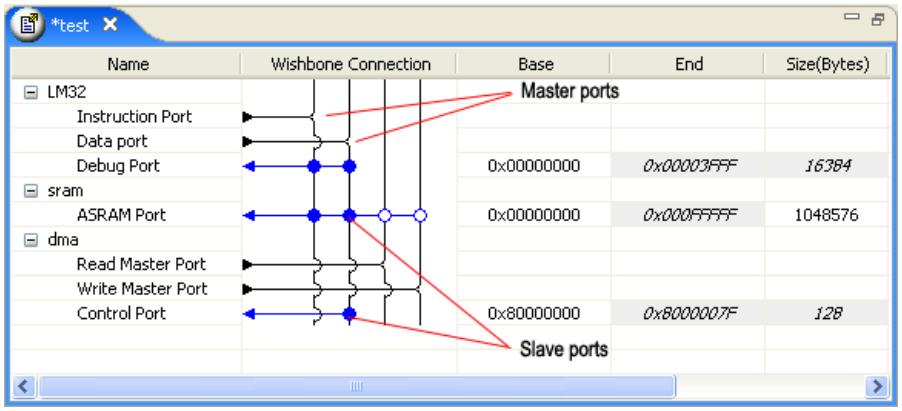


Figure 10: Connecting Master/Slave Ports in Editor View in Slave-Side Fixed and Slave-Side Round-Robin Arbitration Schemes

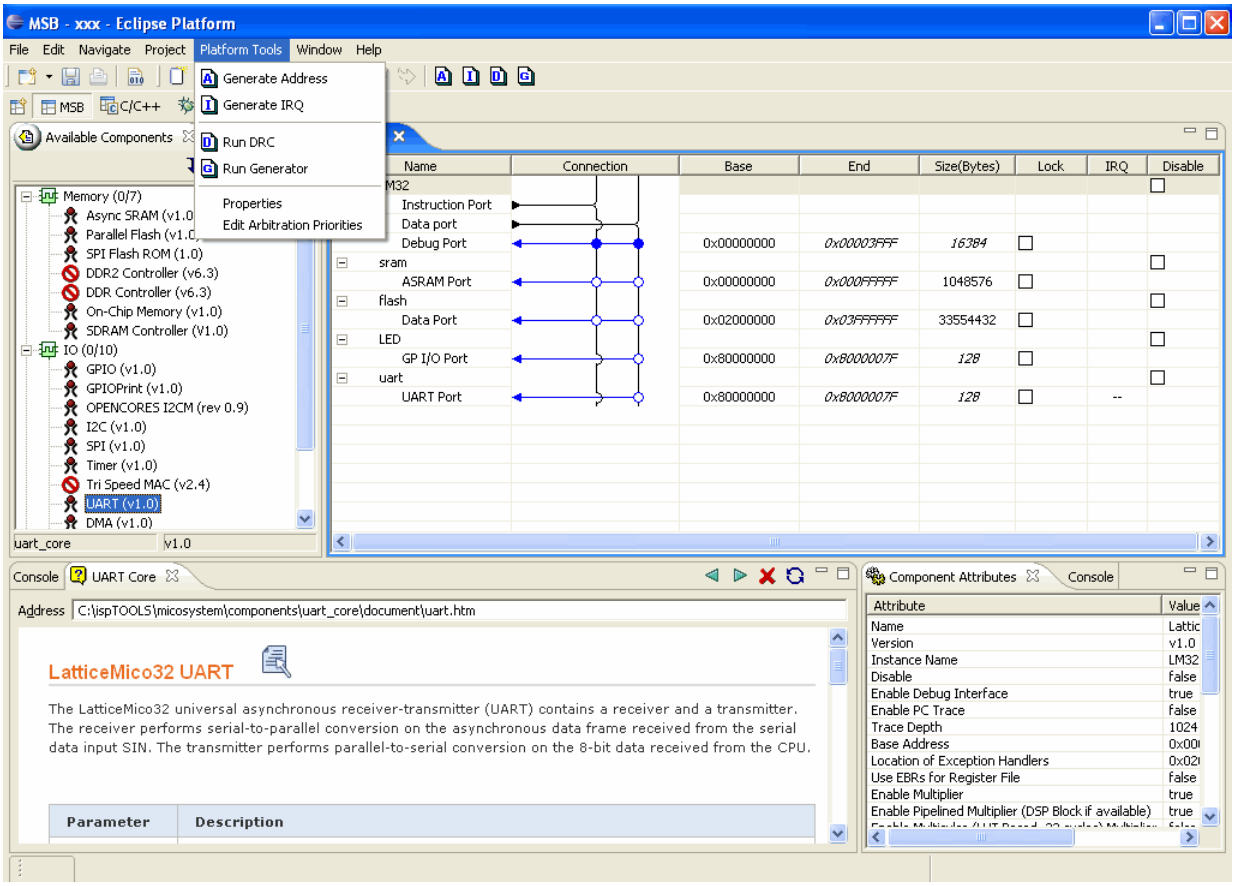


menu is therefore disabled when you use the slave-side round-robin arbitration scheme, as shown in Figure 12.

Figure 11 shows the Platform Tools menu with the Edit Arbitration Priorities command enabled in the MSB perspective after all components have been added in a slave-side fixed arbitration scheme.

Figure 12 shows the Platform Tools menu with the Edit Arbitration Priorities command disabled in the MSB perspective after all components have been added in a slave-side round-robin arbitration scheme.

Figure 11: MSB Perspective After Adding All Components in a Slave-Side Fixed Arbitration Scheme



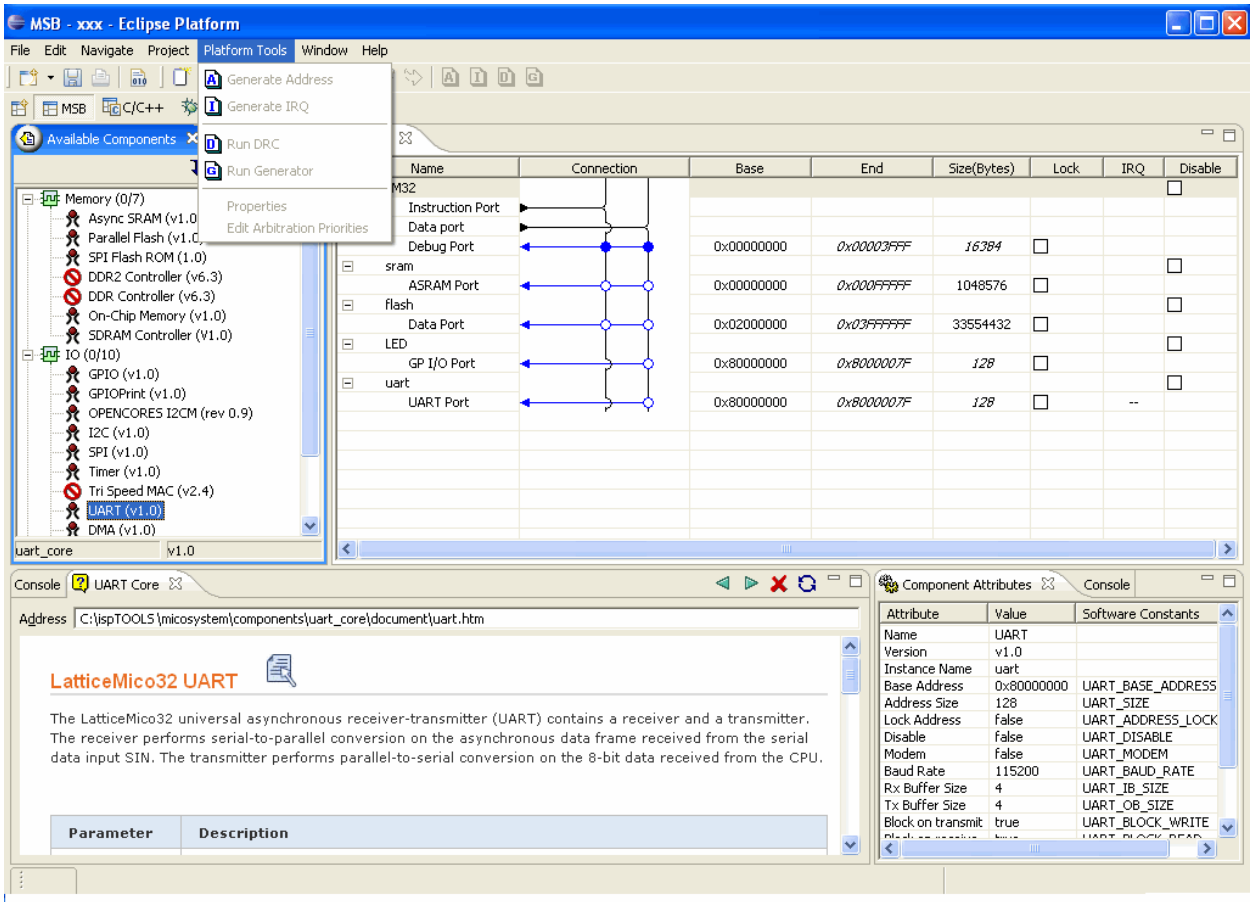
Changing Master Port Arbitration Priorities

When you first generate your platform, LatticeMico System automatically assigns priorities through the shared-bus and slave-side fixed arbitration schemes to the master ports to determine in which order they can access the slave ports through the arbiter. You can change these priorities only for the shared-bus and slave-side fixed arbitration schemes. This option is disabled for the slave-side round-robin arbitration scheme, since it is not applicable.

To change master port arbitration priorities:

1. In the MSB perspective, click in the Editor view to make it active and choose **Platform Tools > Edit Arbitration Priorities** from the menu, or right-click in the Editor view and choose **Edit Arbitration Priorities**.
2. In the Edit Arbitration Priorities dialog box, click in the **Priority** column next to the master port whose priority you wish to change.
3. Type in the new priority number.
4. Click **OK** and choose **File > Save** to save this in the .msb file.

Figure 12: MSB Perspective After Adding All Components in a Slave-Side Round-Robin Arbitration Scheme



When you perform a DRC check, MSB checks the validity of the priorities that you have changed.

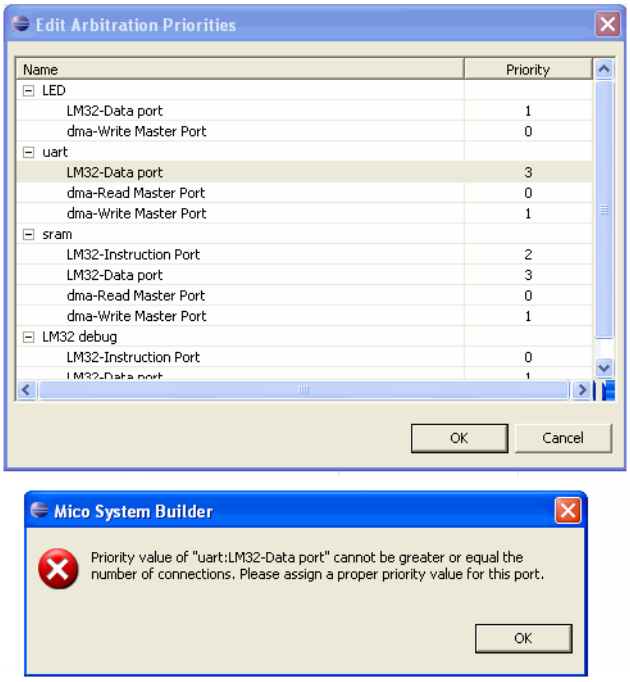
When you assign arbitration priorities to the master port of a slave in the slave-side fixed arbitration scheme, the number of priorities should not be greater than the total number of master ports for that slave. For example, if a slave has three master port values, the arbitration priorities would be 0, 1, and 2. If you defined more than three values for any master, an error message would appear, as shown for the UART slave example in Figure 13.

Assigning Component Addresses

After you add your components to your microprocessor platform, you must ensure that you assign unique address locations to each.

If you look in the Editor view in the Base column, you will notice that the components, after initial setup, all are assigned to the same default address location on creation, unless you actively assign a unique base address in a component dialog box when you first add the component to the platform. Any

Figure 13: Edit Arbitration Priorities Error Message



duplicate address locations of any component are, of course, not viable. This section provides procedures for assigning these unique address locations.

MSB can automatically generate an address in hexadecimal notation for each component with slave ports. Or, you can assign a component an individual address. Components with master ports are not assigned addresses.

Before you generate addresses, you can lock the base addresses of individual components so that MSB will not assign them new addresses. See “Locking Component Addresses” on page 29 for details.

Note

Address and size values that appear in italic font in the Editor view cannot be changed.

Automatically Assigning Component Addresses

Initially, you may want the software to automatically generate assigned address locations for the components in your platform and edit them as necessary later.

To automatically assign component addresses:

1. In the MSB perspective, choose **Platform Tools > Generate Address** or click the Generate Address toolbar button . You can also right-click in the Editor view and choose **Generate Address** from the pop-up menu.

Address locations for all of the existing components that you have created in your MSB session are now automatically generated.

2. Choose **File > Save**.

   The assigned component addresses are now saved in the .msb file.

Locking Component Addresses

Locking a component address prohibits the software from changing it after you automatically assign component addresses.

To lock any addresses from being changed after automatic address generation:

1. In the MSB perspective Editor view, select the box for the desired component in the Lock column.

   This step activates a lock during your session.

2. Choose **File > Save**.

   The locked address is now saved in the .msb file.

   Note

   To assign an address for only one component, lock all other components.

Manually Editing Component Addresses

You can manually assign an address to an individual component after automatically assigning an address to it, or you can assign locations as you wish by manually editing the locations at any time after initial component creation.

To individually change the addresses of components:

1. In the MSB perspective Editor view, click on the desired component's address in the Base column.

   The address becomes editable.

   Note

   You can only edit the Base address. You cannot edit the End address. The value of the end address is equivalent to the value of the base address plus the value of the size.

2. Manually type in the desired address hexadecimal location.
3. Choose **File > Save**.

   The edited addresses are now saved in the .msb file.

Assigning Component Interrupt Priorities

Assign an interrupt request priority (IRQ) to all components that feature a dash in the IRQ column of the Editor view. You cannot assign interrupt priorities to components lacking this dash in the IRQ column, such as memories and CPUs.

To assign interrupt priorities for all components other than memories and the CPU:

1. In the MSB perspective, choose **Platform Tools > Generate IRQ** or click the Generate IRQ toolbar button  . You can also right-click in the Editor view and choose **Generate IRQ** from the pop-up menu.
2. Choose **File > Save**.

   The interrupt priorities are now saved in the .msb file.

Performing Design Rule Checks

You can ensure that your design conforms to the design rules for a given device by performing a design rule check (DRC).

To perform a design rule check and verify the addressing:

- In the MSB perspective, choose **Platform Tools > Run DRC** or click the Run DRC toolbar button  . You can also right-click in the Editor view and choose **Run DRC** from the pop-up menu.

Saving the Microprocessor Platform

After you do a number of tasks to set up your microprocessor platform, you should save your changes.

To save your platform changes in MSB:

- In the MSB perspective, choose **File > Save**.

  This operation specifically saves any changes you made to the .msb file and any option settings you may have applied.

Generating the Microprocessor Platform

Generating the microprocessor platform saves and updates the platform definition by updating the .msb file. It also does the following:

- Assigns addresses to components without locked addresses

- Assigns interrupt priorities
- Performs design rule checking (DRC)
- Generates a platform Verilog structural implementation
- Creates hardware and software implementation support files for the components that are used in the platform
- For the Verilog user, creates an instance template for the platform that can be used to incorporate the platform within a larger design
- For VHDL user (a user who has selected “Create VHDL Wrapper” in the New Platform dialog box), creates a VHDL entity/architecture definition that instantiates the platform as a black box
- For the VHDL user who has selected the optional “Create VHDL NGO File,” synthesizes the platform during the generation step and creates a series of .ngo files that represent the post-synthesis netlist of the platform. These files are included in the rest of your VHDL design after it has been synthesized.

To generate your microprocessor platform in MSB:

- In the MSB perspective with the Editor view activated, choose **Platform Tools > Run Generator** or click the Run Generator toolbar button  . To activate the Editor view, click on the Editor view tab or anywhere inside the view. You can also right-click and choose **Run Generator** from the pop-up menu.

  Note

  If you did not set the IPEXPRESS_SYN_PATH environment variable before starting Synplify Pro, as noted in “Starting MSB” on page 15, or if Synplify Pro failed to complete the synthesis, MSB may issue the following error message:

  `ERROR: edif2ngd: Cannot open input file "<platform_name>.edi".`

  If you receive this error message, verify that the IPEXPRESS_SYN_PATH is set correctly, and check the synthesis output in the log file or .srr file in the soc/ngo/rev_1 directory to see if the error is a synthesis syntax error.

  If you edit the .msb file after it has been generated, save it by choosing **File > Save As**. An asterisk (*) preceding <*platform_name*>.msb above the Editor view indicates that the <*platform_name*>.msb file has been edited.

  During the generation process, MSB creates the following files in the <*Diamond_install_path*>\<*platform_name*>\soc directory:

- A <*platform_name*>.msb file, which describes the platform. It is in XML format and contains the configurable parameters and bus interface information for the components. It is passed to C/C++ SPE, which extracts the platform information (for example, where components reside in the memory map) required by the software that will run on the platform. It is used by users of the Verilog flow and the VHDL flow.
- A <*platform_name*>.v (Verilog) file, which is used by both Verilog and VHDL users:

- Flow for Verilog users – The <*platform_name*>.v file is used in both simulation and implementation. It instantiates all the selected components and the interconnect described in the MSB graphical user interface. This file is the top-level simulation and synthesis RTL file passed to Diamond. It includes the .v files for each component in the design, which are used to synthesize and generate a bitstream to be downloaded to the FPGA. The .v files for each component reside under the top-level <*platform_name*>.v file.
- Flow for VHDL users – The <*platform_name*>.v file is used in simulation and implementation. If “Create VHDL NGO File” has been selected, the <platform_name>.v file is used for simulation only, and the <*platform_name*>_vhd.vhd file is used for implementation. In the NGO flow, the <*platform_name*> component is instantiated as a black box, and this instantiation is then automatically combined with the <*platform_name*>.ngo file after synthesis to complete the implementation netlist.

  A mixed-mode Verilog and VHDL simulator is needed for functional simulation in the flow for VHDL users.

- A <*platform_name*>_vhd.vhd (VHDL) file, if you selected the “Create VHDL Wrapper” option in the New Platform Wizard dialog box. It is intended to be used only to incorporate the Verilog-based platform into a VHDL design. The <*platform_name*>_vhd.vhd file contains the top-level design used for synthesis. This top-level design file instantiates the <*platform_name*> component as a black box. If the optional “Create VHDL NGO File” has been selected, the <platform_name>_vhd.vhd file is combined with the <*platform_name*>.ngo file after synthesis to complete the post-synthesis netlist. The common name <*platform_name*> is used to make this association.
- A <*platform_name*>.ngo file, which is a Diamond database file that is a synthesized version of <*platform_name*>.v. This file is created if the optional “Create VHDL NGO File” has been selected, along with “Create VHDL Wrapper.” It contains the same design information as <*platform_name*>.v. For more information on the .ngo file, see the “Building Modular Projects Using NGO Flow” topic in the Diamond online Help.

MSB generates a <*platform_name*>_inst.v file, which contains the Verilog instantiation template to use in a design where the platform is not the top-level module. For the VHDL user, no equivalent file is generated that contains the component declaration and component instance/portmap template for the platform wrapper <*platform_name*>_vhd.vhd. The generated <*platform_name*>_vhd.vhd file can be used to create one, if required. Figure 14 shows the instantiation template for the platform1 platform.

Figure 14: Verilog Instantiation Template

```
platform1 platform1_u (
.clk_i(clk_i),
.reset_n(reset_n)
, .sramsram_csn(sramsram_csn) //
, .sramsram_be(sramsram_be) // [3:0]
, .flashsram_csn(flashsram_csn) //
, .flashsram_be(flashsram_be) // [3:0]
, .flashsram_byten(flashsram_byten) //
, .flashsram_wpn(flashsram_wpn) //
, .flashsram_rstn(flashsram_rstn) //
, .LEDPIO_OUT(LEDPIO_OUT) // [10-1:0]
, .uartSIN(uartSIN) //
, .uartSOUT(uartSOUT) //
, .sramflashOEN(sramflashOEN)
, .sramflashWEN(sramflashWEN)
, .sramflashADDR(sramflashADDR)// [24:0]
, .sramflashDATA(sramflashDATA)// [31:0]
);
```

Implementing Shared Bidirectional Bus to Board

Some components in the microprocessor platform—such as the ASRAM, the flash controller, and the SDRAM controller—have bidirectional data buses. Tristates in the platform are used extensively to implement them. This section gives some guidelines on handling tristates.

Implementing a Shared Board Bus in MSB

When you use an evaluation board issued by a vendor other than Lattice Semiconductor, you might need to use tristates to implement a bidirectional data bus between the board and LatticeMico32 microprocessor platform. A shared bidirectional data bus reduces pin count and board traces.

If you select “Share External Ports” for the ASRAM and flash components in the generated <*platform_name*>.v file in the Verilog flow, the shared bus connection will have a fixed pattern similar to the following:

```
assign sramflashDATA = !sramsram_wen ? sramsram_data_out :
!flashsram_wen ? flashsram_data_out :
32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
```

If you want to change that connection, you must manually modify the code in the <*platform_name*>.v file in the *./*<*platform_name*>/soc directory by adding logic to the tristates so that multiple LatticeMico32 components can share this bidirectional bus.

IF you want to modify the shared bidirectional data bus in certain platform components in the VHDL, you must modify the verilog code to change the tristate control. You can do this by either pulling the tristates outside of the platform or making the changes within the platform. If you pull the tristates

from the platform and you are using the VHDL wrapper, then you must also modify the <*platform_name*>_vhd.vhd file in the ./<*platform_name*>/soc directory.

However, if you regenerate the platform in MSB after you add logic to the tristates, these additions will be lost in the automatically generated <*platform_name*>.v or <*platform_name*>_vhd.vhd file in the ./<*platform_name*>/soc directory. To avoid losing your work, copy the modified <*platform_name*>.v or <*platform_name*>_vhd.vhd file to another location, regenerate the platform, and then copy the modified <*platform_name*>.v or <*platform_name*>_vhd.vhd file back to the ./<*platform_name*>/soc directory.

Connecting Bidirectional Ports of a Platform

Tristates can only be connected to external ports.

Avoiding Double-Buffered Bidirectional Ports in VHDL NGO Flow

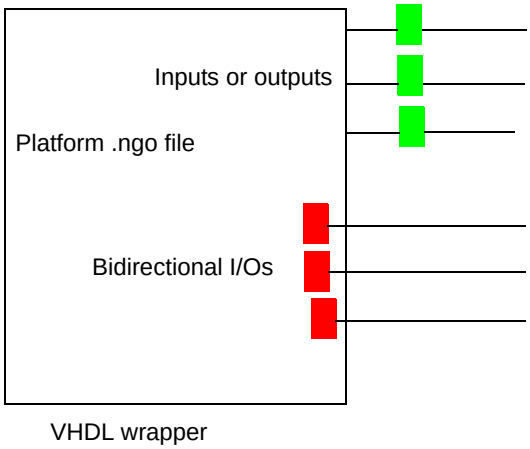
In the flow for VHDL users who have selected the optional “Create VHDL NGO File,” the platform resides in an .ngo file, and the VHDL wrapper file is used to connect to the other VHDL user logic. No I/O pad is inserted in the .ngo file except these bidirectional signals. To avoid double-buffering these bidirectional ports, you must declare them as black-box pads. This declaration tells the synthesis tool that a black-box port has an I/O buffer already implemented inside the black box and therefore the synthesis tool should not put another I/O buffer for this port in the netlist that it is creating. Here is an example:

```
component platform_xxx
port(
D: in std_logic;
E: in std_logic;
GINOUT : inout std_logic_vector(2 downto 0);
Q : out std_logic
);
end component;
attribute syn_black_box : boolean;
attribute syn_black_box of platform_xxx: component is true;
attribute black_box_pad_pin : string;
attribute black_box_pad_pin of platform_xxx: component is
"GINOUT(2:0)";
```

When you declare the I/O ports as black-box pads during VHDL synthesis, the synthesis tools do not insert I/O pads for these signals and therefore avoid double buffering.

Figure 15 further clarifies the connection. For all non-bidirectional I/Os, the I/O buffers (in green) are provided by the VHDL wrapper during VHDL synthesis. The bidirectional I/O buffers (in red) are provided by the .ngo file itself.

Figure 15: I/O Buffers in VHDL Wrapper and in .ngo File



Synthesizing the Platform to Create an EDIF File (Linux Only)

If you use Linux, you must now synthesize your platform to create an EDIF file.

Using Synplicity Synplify Pro

To use Synplicity Synplify Pro as your synthesis tool:

- Add the <*platform_name*>.v file into your Synplify Pro project.

Using Mentor Graphics Precision RTL Synthesis

To use Mentor Graphics Precision RTL Synthesis as your synthesis tool:

1. Add the <*platform_name*>.v file into your Precision RTL project.
2. Add the following directory paths into your Precision RTL search path:
   - <*platform_name*>/soc
   - <*platform_name*>/components/lm32_top/rtl/verilog
   - <*platform_name*>/components/<*uart_core*>/rtl/verilog, where <*uart_core*> is the name of the UART
   - <*platform_name*>/components/<*wb_sdr_ctrl*>/rtl/verilog, where <*wb_sdr_ctrl*> is the name of the SDRAM controller

If your platform includes an OPENCORES I2CM component, you must add an additional directory to the search path as follows:

<*platform_name*>/components/i2cm_opencores/rtl/verilog

See the *Synthesis Data Flow Tutorial* for step-by-step information about synthesizing designs in Precision RTL Synthesis and Synplify Pro.

To create an EDIF file:

1. Start the synthesis tool.
2. Create a new project in the tool.
3. Add the Verilog HDL file output by MSB to the project.
4. Set the target device and the options.
5. Compile the project and specify the timing objectives.
6. Synthesize the design to generate an EDIF (.edn or .edf) file.

Design Guidance for Platform Performance

Setting preferences and performing static timing analysis can help achieve higher platform design performance or minimize area utilization. The following documents give instructions and examples for setting design constraints:

- Achieving Timing Closure in FPGA Designs – This tutorial provides techniques for optimizing design performance and demonstrates the influence of map and place-and-route preferences. It uses a system-on-chip design that utilizes an OpenRISC 1200 processor and Wishbone on-chip bus.
- FPGA Design Guide – The chapter "Strategies for Timing Closure" gives instructions for constraining your design, performing static timing analysis, and floorplanning.

Additionally, see the following sections of the Diamond online Help

- Constraints Reference Guide – This section provides syntax and descriptions for all preferences
- Applying Design Constraints – This section consists of guidelines for setting preferences

Generating the Microprocessor Bitstream

For Windows, you now return to Diamond to import the platform source files. You import the Verilog file output by MSB; or for mixed Verilog/VHDL, you import both the Verilog and VHDL files output by MSB. For Linux, you import the EDIF file output by the synthesis tool. You also specify the connections from the microprocessor to the chip pins by importing an .lpf file. You can optionally perform functional simulation and timing simulation. Primarily, you

will build the database; map, place, and route the design; and generate the bitstream in Diamond so that you can download that configuration bitstream to the chip on a circuit board.

Configuring the Diamond Environment

1. In Diamond, choose **Tools > Options**.
2. Under “Environment” in the pane on the left, select **General**.
3. If the "Copy file to Implementation's Source directory when adding existing file" is selected, clear the selection and click **OK**.

Importing the Verilog or VHDL File on Windows

The process of importing the generated platform file into Diamond is the same for Verilog and VHDL, except that you must take a few additional steps when you import a VHDL file.

To import the Verilog (.v) and VHDL (.vhd) files output by MSB on the Windows platform:

1. Choose **File > Add > Existing File**.
2. In the dialog box, browse to the <*platform_name*>\soc\ location and do one of the following:
   - Select the **<*platform_name*>.v** file (Verilog) and click **Add**.
   - If your design is mixed Verilog/VHDL, select both the **<*platform_name*>.v** file and the **<*platform_name*>_vhd.vhd** file and click **Add**.
3. If your design is mixed Verilog/VHDL and you selected the Create VHDL Wrapper option to generate <*platform_name*>_vhd.vhd without selecting the Create VHDL NGO File option, perform these additional steps:
   a. Choose **Project > Property Pages**.
   b. In the dialog box, select the project name that appears in bold type next to the implementation icon  .
   c. In the right pane, click inside the Value cell for “Top-Level Unit” and select **<*platform_name*>_vhd** from the drop-down menu.
   d. Click inside the Value cell for “Verilog Include Search Path,” and then click the browse button to open the “Verilog Include Search Path” dialog box.
   e. In the dialog box, click the New Search Path button  , browse to the **<*platform_name*>\soc** directory, and click **OK**.
   f. Click **OK** to add the path to the Project Properties and close the “Verilog Include Search Path” dialog box.
   g. Click **OK** to return to the Diamond main window.

Importing the EDIF File on Linux

For Linux, you import the EDIF file generated by the synthesis tool into Diamond.

To import the EDIF (.edn or .edf) file output by MSB on Linux:

1. Choose **File > Add Existing File**.
2. In the dialog box, browse to the location of your .edn or .edf file, select the file, and click **Open**.

Connecting the Microprocessor to FPGA Pins

You have two options for connecting the microprocessor to the FPGA pins:

- Manually create the pin constraints and import them into Diamond.
- Import a pre-created constraints file that is part of the platform templates in the LatticeMico System software into Diamond.

For information about pin constraint assignments, see the “Applying Design Constraints” and “Constraints Reference Guide” in the Lattice Diamond online Help.

You can import the pin constraints specified for a template platform into Diamond. When you use a platform template, MSB copies the logical preference (.lpf) file associated with it into the ..\soc directory path of your LatticeMico32 project.

To import the .lpf file:

1. In the Diamond, choose **File > Add > Existing File**.
2. Browse to the .lpf file and click **Open**.

Generating the Bitstream

Now you will generate a bitstream file to download the microprocessor to the FPGA. This process automatically synthesizes, translates, maps, places, and routes the design before it generates the bitstream file.

To generate a bitstream file:

1. In Diamond, select the Process tab.
2. In the Process pane, under Export Files, double-click **Bitstream File**.

   The Diamond software generates the programming file in your project folder. It is now ready for downloading onto the device.

Downloading Hardware Bitstream to the FPGA

After you generate the bitstream file, you can download it to program your FPGA device on a circuit board. You can use Diamond Programmer to accomplish this task.

To download the hardware bitstream using the Diamond Programmer:

1. In the Diamond, choose **Tools > Programmer**.
2. The Programmer opens, displaying the bitstream file you have generated for the current design in the Data File box.
3. In the Programmer user interface, click **Auto Detect**.

   The Programmer can recognize the USB download cables plugged into your PC. If more than one USB cable is connected to your PC, the Programmer detects all available cables, but selects the first cable that it detects.

   To select a different USB cable, select **USB** (LSC USB cable) or **USB2** (FTDI USB2 cable) in the Cable Type box, and change the connection port in the Port drop-down list.
4. Click **Scan Device**.

   The Programmer scans the printed circuit board connected to your computer with the specified cable and port, and it lists the devices in the Device list.
5. In the Device list, select the device that matches the target device of the current design.
6. Under XCF File, do either of the following:

   - If you want to use an existing chain file to program the device, select **Downloading with existing XCF file**. Then click **Browse** to locate the file.
   - If you have no existing chain file for programming, select **Save to XCF file** to create a new XCF file. Then click **Browse** to specify the name and location for the new file.
7. Click **Download**.

   The Programmer downloads the data file to the target device. A Status box indicates the progress of the operation, reports any errors, and shows whether the operation was successful.

Performing HDL Functional Simulation of LatticeMico32 Platforms

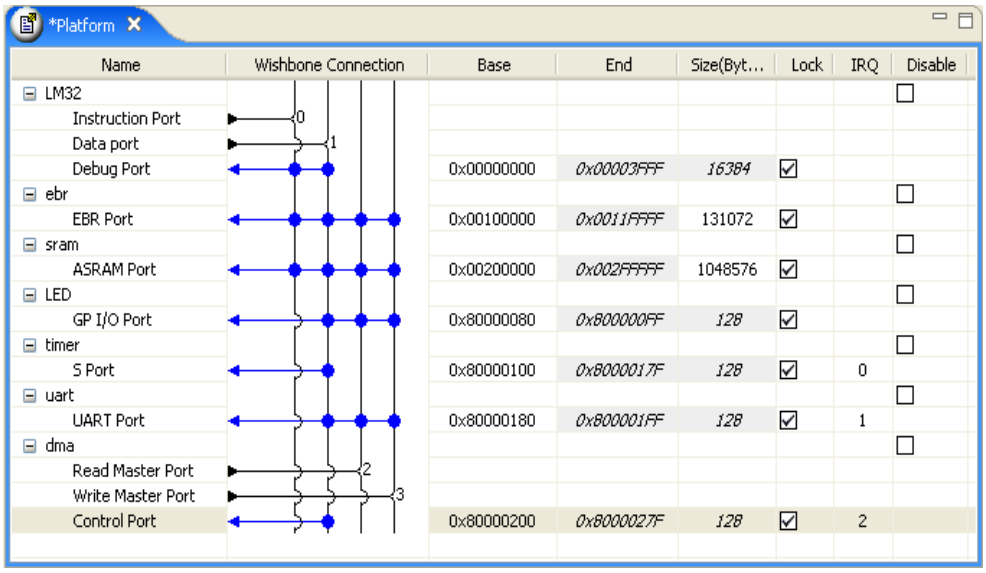
In most cases, the platforms that are created using the LatticeMico System Builder work correctly in hardware because the existing components have been tested many times. New custom components, however, start as untested elements and will probably need debugging through HDL functional simulation.

This topic describes the process for using an HDL simulation tool such as Mentor Graphics ModelSim™ or Aldec Active-HDL™. The method described is applicable to designs written in VHDL, Verilog, or a combination of both. The example LatticeMico32 platform in this topic uses the FPGA's on-chip memory, Embedded Block Ram (EBR). The firmware (C/C++ code) is compiled using the Lattice C/C++ SPE and Debug software, and a memory initialization file is created that is loaded into the on-chip memory. It is possible to locate the firmware in other off-chip memories as long as there exists a behavioral model for the memory.

The example application used in this topic is the “Hello World” application, which is available as a predefined C/C++ SPE project. See Chapter 2 and Chapter 6 of the *LatticeMico System Software Developer User Guide* for more information about creating the “Hello World” application, compiling it, and deploying it to the EBR.

The platform in Figure 16 shows a Verilog design (platform) that is instantiated from within a VHDL module.The platform is an instantiation of Platform C with the following additional components:

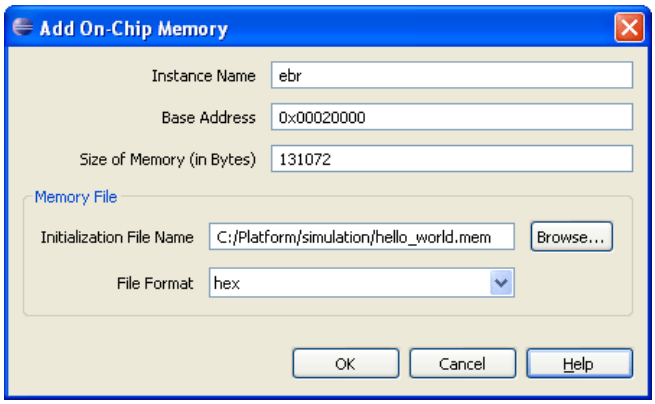
Figure 16: Platform Setup



- EBR – At least one EBR is required to hold the software/firmware. Figure 17 on page 41 shows the design’s EBR component setup, along

with the deployed software. The EBR memory size must be large enough to hold your C/C++ application. In this example, the memory is 128KB, which is more than enough for the “Hello World” application but too large for most FPGAs to support.

Figure 17: EBR Setup



- UART – The UART is an optional component that redirects `stout` and `sterr` to the HDL simulator console.

  Refer to the UART data sheet for more details on how to use the UART for redirecting `stout` and `stderr` to the HDL simulator console.

Configuring the Platform with LatticeMico System Builder

The LM32 processor instance in the platform, shown in Figure 16, must be configured to permit functional simulation of software applications through any HDL simulator. The following steps are required:

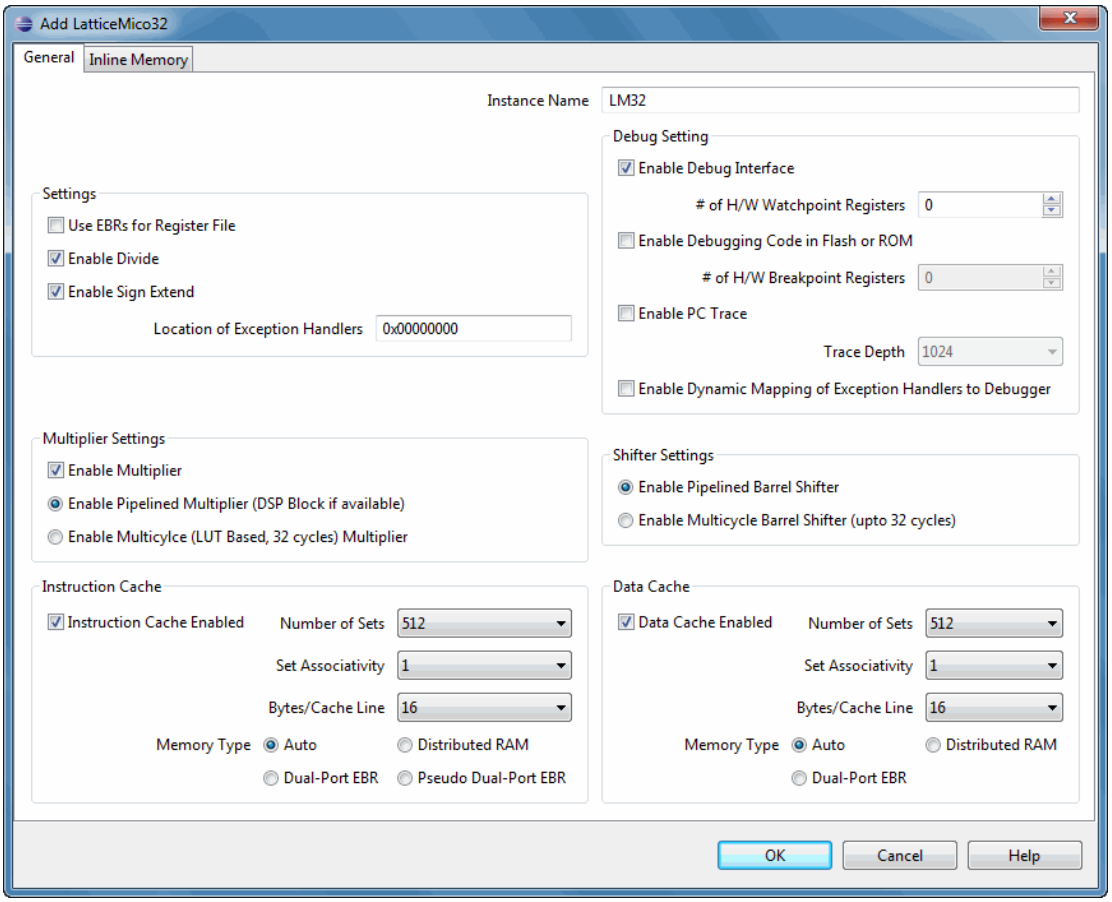
Ensure that the LM32 Exception Base Address (EBA) points to the base address of the memory component that contains the deployed software/ firmware. This address must be aligned to a 256-byte boundary, since the hardware ignores the least-significant byte. Unpredictable behavior occurs when the exception base address and the exception vectors are not aligned on a 256-byte boundary. In the platform shown in Figure 16 on page 40, the software is deployed in EBR. Therefore, modify the “Location of Exception Handlers” field in the LM32 processor dialog box to point to the base address of EBR (0x00020000), as shown in Figure 18.

Directory Structure

When MSB is used to generate a platform, a set of directories is created in a top-level platform directory. The top-level directory is automatically assigned the same name as the MSB project name, which is Platform in this example.

<*path_to_toplevel_directory*>/Platform
    components
    soc

Figure 18: LM32 Setup



The components directory contains RTL and software drivers that pertain to each of the components instantiated within the design. Important files in the soc directory include:

- system_conf.v – This file contains the auto-generated macro definitions of the various components in the design. As mentioned previously, this file must be modified if the “Enable Debug Interface” option is selected in the LM32 processor dialog box.
- platform.v – This file contains the top-level module of the design, which is Platform in this example.
- pmi_def.v – This file contains module definitions of all the PMI modules used in the design. For the purpose of functional simulation, the PMI behavioral models must be provided. See “Replace PMI Black-box Instantiations with Behavioral Models.” on page 46.

Creating an Optional VHDL Wrapper

For mixed-language designs, the VHDL Wrapper is required for simulation. To demonstrate mixed-language functional simulation, a VHDL wrapper has been created for the top-level module in the design example. The example wrapper, platform_vhdl.vhd, is located in the soc directory and is shown in Figure 19 on page 43.

Figure 19: VHDL Wrapper

```
library ieee;
use IEEE.std_logic_1164.all;
entity Platform_vhd is
  port (
    clk_i             : in    std_logic;
    reset_n           : in    std_logic;
    sramsram_wen      : out   std_logic;
    sramsram_data     : inout std_logic_vector(31 downto 0);
    sramsram_addr     : out   std_logic_vector(22 downto 0);
    sramsram_csn      : out   std_logic;
    sramsram_be       : out   std_logic_vector(3 downto 0);
    sramsram_oen      : out   std_logic;
    LEDPIO_OUT        : out   std_logic_vector(7 downto 0);
    uartSIN           : in    std_logic;
    uartSOUT          : out   std_logic;
  );
end Platform_vhd;

architecture structural of Platform_vhd is
  component Platform
    port (
        clk_i             : in    std_logic;
        reset_n           : in    std_logic;
        sramsram_wen      : out   std_logic;
        sramsram_data     : inout std_logic_vector(31 downto 0);
        sramsram_addr     : out   std_logic_vector(22 downto 0);
        sramsram_csn      : out   std_logic;
        sramsram_be       : out   std_logic_vector(3 downto 0);
        sramsram_oen      : out   std_logic;
        LEDPIO_OUT        : out   std_logic_vector(7 downto 0);
        uartSIN           : in    std_logic;
        uartSOUT          : out   std_logic;
      );
  end component;
begin
  Platform_u : Platform
    port map (
      clk_i => clk_i,
      reset_n => reset_n,
      sramsram_wen => sramsram_wen,
      sramsram_data => sramsram_data,
      sramsram_addr => sramsram_addr,
      sramsram_csn => sramsram_csn,
      sramsram_be => sramsram_be,
      sramsram_oen => sramsram_oen,
      LEDPIO_OUT => LEDPIO_OUT,
      uartSIN => uartSIN,
      uartSOUT => uartSOUT,
      );
end structural;
```

Preparing for HDL Functional Simulation

The following sections describe the steps required to perform functional simulation on a given platform.

1. **Create the Simulation Directory**.

   Functional simulation is performed in a directory that is created under the top-level directory, which is named Platform in this example.

   <path_to_toplevel_directory>/Platform
   components
   soc
   simulation

2. **Create the Testbench**.

   A testbench is required to functionally verify a design. The example testbench, shown in Figure 20 on page 44, instantiates Platform_vhdl, the top-level module of the design.

Figure 20: Testbench File

```
`timescale 1 ns / 1 ns
`include "lm32_include.v"

module testbench;
   event done;
   // Inputs
   reg clk_i;
   reg reset_n;
   // Outputs
   wire [7:0]  LEDPIO_OUT;
   wire        sramsram_wen;
   wire [31:0] sramsram_data;
   wire [22:0] sramsram_addr;
   wire        sramsram_csn;
   wire [3:0]  sramsram_be;
   wire        sramsram_oen;
   wire        uartSIN;
   wire        uartSOUT;

   Platform_vhd Platform_u
     (
       .clk_i(clk_i),
       .reset_n(reset_n),
       .sramsram_wen(sramsram_wen),
       .sramsram_data(sramsram_data),
       .sramsram_addr(sramsram_addr),
       .sramsram_csn(sramsram_csn),
       .sramsram_be(sramsram_be),
       .sramsram_oen(sramsram_oen),
       .LEDPIO_OUT(LEDPIO_OUT),
       .uartSIN(uartSIN),
       .uartSOUT(uartSOUT),
       );
```

Figure 20: Testbench File (Continued)

```
 /*-------------------------------------------------------------------------
    Clock & Reset
    -------------------------------------------------------------------------*/
   initial begin
      reset_n = 0;
      #290;  // delay 290 ns
      reset_n = 1;
   end
   initial begin
      clk_i = 0;
      #20;  // delay 20 ns
      forever #(20) clk_i = ~clk_i;  // toggle the clk_i signal every 20ns
   end

/*-------------------------------------------------------------------------
    Trap "Exit" System Call to terminate simulation
  -------------------------------------------------------------------------*/
  reg scall_m, scall_w;
  always @(negedge clk_i)
     begin
  if (Platform_u.Platform_u.LM32.cpu.stall_m == 1'b0)
    begin
       scall_m <= Platform_u.Platform_u.LM32.cpu.scall_x &
         Platform_u.Platform_u.LM32.cpu.valid_x;
       scall_w <= scall_m & Platform_u.Platform_u.LM32.cpu.valid_m;
    end
// System Call number is passed in r8, Exit System Call is call number 1
  if (scall_w && (Platform_u.Platform_u.LM32.cpu.registers[8] == 1))
    begin
       $display("Program exited with code %0d.\n",
          Platform_u.Platform_u.LM32.cpu.registers[1]);
       -> done;
    end
     end

/*-------------------------------------------------------------------------
    Tasks to perform when simulation terminates
  -------------------------------------------------------------------------*/
   always @(done)
  $finish;

endmodule
```

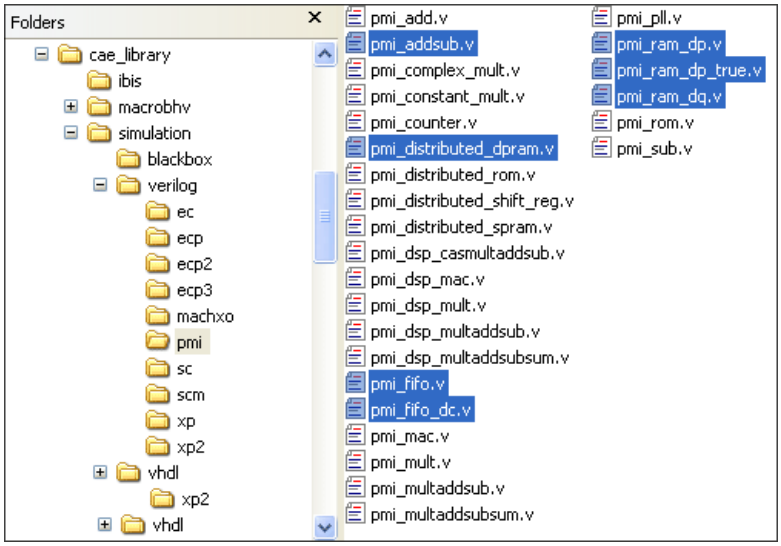
3. **Replace PMI Black-box Instantiations with Behavioral Models**.

   The black-box instantiation of each PMI module in the file pmi_def.v must be replaced with its respective behavioral model. The PMI behavior models are located in the simulation directory of the Diamond installation:

   `<diamond_install_path>/cae_library/simulation/verilog/pmi`

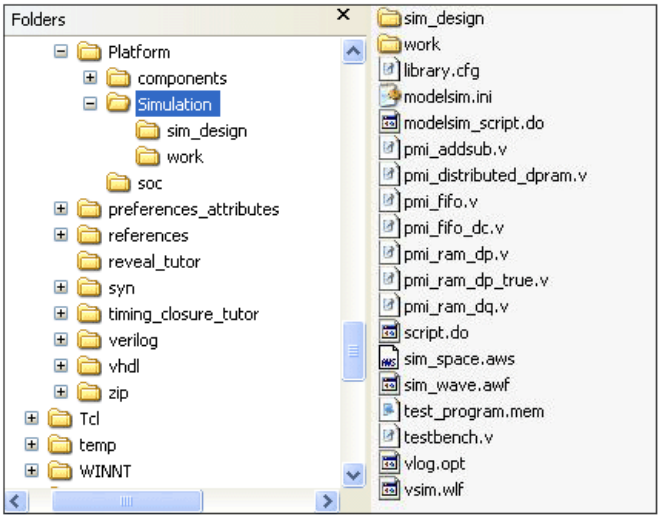
   - Select the behavioral model of each PMI module from the simulation directory in the Diamond installation. Figure 21 shows those selected for the Platform example.

Figure 21: Selected PMI Behavior Models from CAE Library



   - Copy the selected models and paste them into the platform's simulation directory. Figure 22 shows the simulation directory of the Platform example where PMI modules have been replaced by the appropriate behavior models.

Figure 22: PMI Models in Platform Simulation Directory



Performing HDL Functional Simulation with Aldec Active-HDL

To perform HDL functional simulation with Aldec Active-HDL, first create a script, “aldec_script.do,” and place it in the simulation directory. Copy the following commands into the script:

```
cd "<path_to_toplevel_directory>/Platform/simulation"
workspace create sim_space
design create sim_design .
design open sim_design
cd "<path_to_toplevel_directory>/Platform/simulation"
set sim_working_folder .
vlog pmi_addsub.v
vlog pmi_ram_dq.v
vlog pmi_ram_dp.v
vlog pmi_ram_dp_true.v
vlog pmi_distributed_dpram.v
vlog pmi_fifo.v
vlog pmi_fifo_dc.v

# add additional vlog commands for each PMI module in the
# design. The list shown is not intended to be complete for all
# possible LM32 designs.

vlog +define+SIMULATION ../soc/platform.v
acom ../soc/platform_vhd.vhd
vlog +incdir+../Components/lm32_top/rtl/verilog+../soc
testbench.v

# the VSIM command uses the Aldec for Lattice pre-compiled FPGA
# libraries. If the Aldec for Lattice simulator is not being
# used, it will be necessary to compile the behavioral code for
# the FPGA. For the ECP2, the behavioral code is located at:
# <isptools>/cae_library/simulation/verilog/ecp2

vsim testbench –L ovi_ecp2
```

Launch the Active-HDL software and execute the following command in the console window:

```
cd <path_to_toplevel_directory>/Platform/simulation

# verify that you are in the correct directory

pwd
do aldec_script.do
```

Performing HDL Functional Simulation with Mentor Graphics ModelSim

To perform HDL functional simulation with ModelSim, first create a script, “modelsim_script.do,” and place it in the simulation directory. Copy the following commands into the script:

```
vlib work
vdel -lib work -all
vlib work
vlog pmi_addsub.v
vlog pmi_ram_dq.v
vlog pmi_ram_dq.v
vlog pmi_ram_dp.v
vlog pmi_ram_dp_true.v
vlog pmi_distributed_dpram.v
vlog pmi_fifo.v
vlog pmi_fifo_dc.v
vlog +define+SIMULATION \
+incdir+../soc+../components/gpio/rtl/verilog+../components/
lm32_top/rtl/verilog+../components/timer/rtl/verilog+../
components/wb_ebr_ctrl/rtl/verilog+../components/asram_top/rtl/
verilog+../components/uart_core/rtl/verilog+../components/
wb_dma_ctrl/rtl/verilog \
 ../soc/platform.v
vcom ../soc/platform_vhdl.vhd
vlog +incdir+../components/lm32_top/rtl/verilog testbench.v

# the VSIM command shown here uses pre-compiled FPGA libraries.
# It may be necessary to compile the behavioral code for the
# FPGA.  In this example, the behavioral code was compiled to
# the ecp2_vlg working directory.
# For the ECP2, the behavioral code is located at :
# <isptools>/cae_library/simulation/verilog/ecp2

vsim work.testbench -t 1ps -novopt -L ecp2_vlg
```

Note

When doing mixed-language simulation, use the `-t 1ps` command-line option for the "vsim" command.

Using LatticeMico System as a Stand-Alone Tool

The software developer can use C/C++ SPE to develop software application code without having to install Diamond, as long as the directory structure and appropriate files have been provided by the hardware developer. The files that the hardware designer provides to the software developers are the Mico System Builder project file, the LM32 processor driver files and GNU files, the component driver files, and the FPGA's configuration bitstream.

The hardware developer needs to have both Lattice Diamond and LatticeMico System installed in order to generate the files and provide them to the software developer.

The following scenario shows the tasks involved:

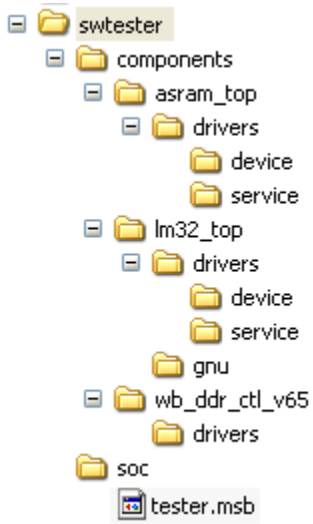
Hardware Developer The hardware developer performs the following tasks:

1. Uses Diamond to create an FPGA development project.

The Diamond software is used to generate the FPGA bitstream containing the LatticeMico32 processor and peripherals.

2. Generates the platform for the project using LatticeMico System Builder.
3. Imports the platform's RTL source files into the project in Diamond and generates the FPGA's configuration bitstream.
4. Sends all software developers the Mico System Builder project directory.

   For example:



5. Sends the software developers the FPGA bitstream file (.bit) that was generated using Diamond.

Software Developer The software developer performs the following tasks:

1. Uploads the files sent from the hardware developer:
   a. Launches the LatticeMico32 Mico System Builder.
   b. Loads the <platform_name>.msb file provided by the hardware engineer.
2. Creates a new managed make or standard make project in C/C++ SPE.
3. Implements the LatticeMico32 firmware.
4. Compiles the LatticeMico32 firmware using the **Project > Build all** command.
5. Runs and debugs the application.

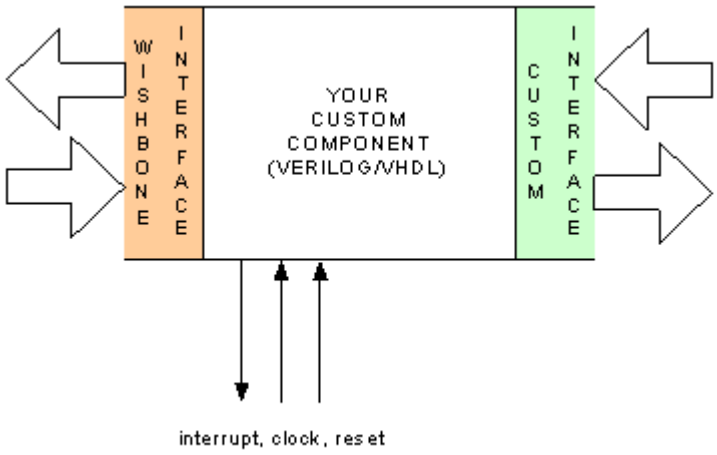
Chapter 3

Creating Custom Components in LatticeMico System

This chapter shows you how to bring your WISHBONE interface component into the Mico System Builder (MSB) so that it is listed as an available component in MSB for use in platforms. It assumes that you already have the Verilog or VHDL source code for the component that you wish to add. Refer to the section “WISHBONE Interconnect Architecture” in the *LatticeMico32 Processor Reference Manual*.

Figure 23 shows an HDL diagrammatic representation of your custom component.

Figure 23: Custom Component Representation



This chapter assumes that you have implemented your custom component and that your custom component has a WISHBONE interface that contains the signals required for connecting to the LatticeMico32’s WISHBONE fabric. Your custom component may have a custom I/O interface that may need to be used as platform input and output pins. Finally, your custom component may require a clock signal and a reset signal. It may provide an interrupt request to the LatticeMico32 processor through an interrupt pin.

The Import/Create Custom component dialog box in MSB enables you to specify the properties for your custom component and makes it available as a

component in MSB. Once imported, your custom component is available every time that you start MSB. The dialog box also enables you to create a component configuration dialog box that lists parameters that you can configure for your custom component. If your custom component has associated software drivers or routines, you can specify them in the dialog box so that they can be used in managed-make projects or a platform-library project for a platform that uses this custom component.

The following steps are required to import your custom component into MSB:

1. Open the Import/Create Custom Component dialog box.
2. Specify the component attributes.
3. Specify the WISHBONE interface connections.
4. Specify the clock/reset and optional external port connections.
5. Specify your custom component's RTL design files.
6. Specify the user-configurable parameters that your RTL design, software, or both may need, if applicable.
7. Optionally, specify software elements.
8. Specify the optional software files that your custom component may provide for use in LatticeMico32 applications.
9. Apply the changes.

Note

The entire flow is based on the assumption that your custom component is written in Verilog. If you have a custom component written in VHDL, you must perform a few more steps before performing the steps just given. Refer to "Creating the Verilog Wrapper for VHDL Designs" on page 87 for these steps.

The following sections introduce you to the Import/Create Custom Component dialog box and explain the steps just given.

Once you have imported your custom component into MSB, you can use the same Import/Create Custom Component dialog box to edit the provided information.

Opening the Import/Create Custom Component Dialog Box

The LatticeMico32 MSB perspective has an Import/Create Custom Component dialog box that allows you to create or import custom components for use in your MSB platform.

To import your WISHBONE-interface-compliant custom component, you must have the following items:

- RTL source files that implement your custom component

- Optional software files that implement software functionality for your custom component

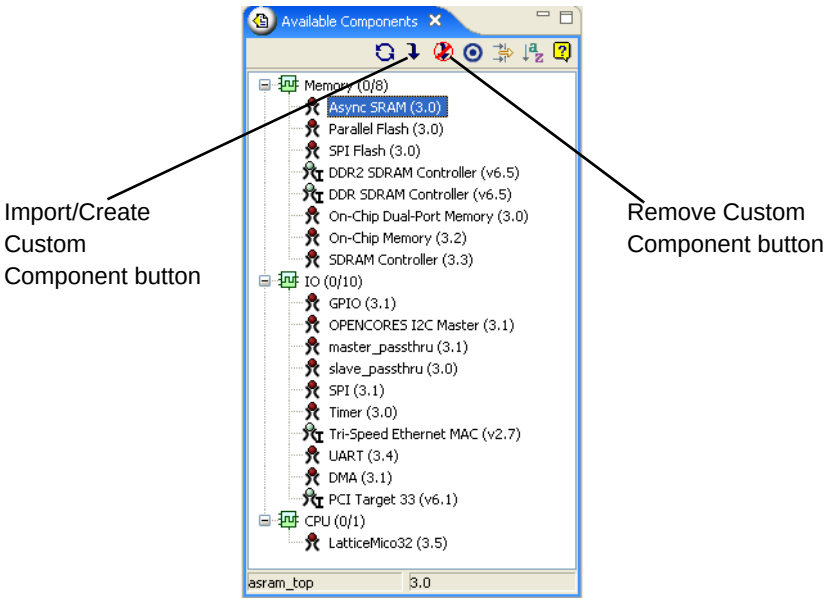
To open the Import/Create Custom Component dialog box:

- In the LatticeMico System MSB perspective, click the **Import/Create Custom Component** button in the MSB Available Components view, as shown in Figure 24.

Note

A custom component is indicated in the MSB Available Components view with the following icon: . You can remove a previously created custom component by highlighting the component in the Available Components view and clicking the Remove Custom Component button.

Figure 24: Import/Create Custom Component Button in Available Components View

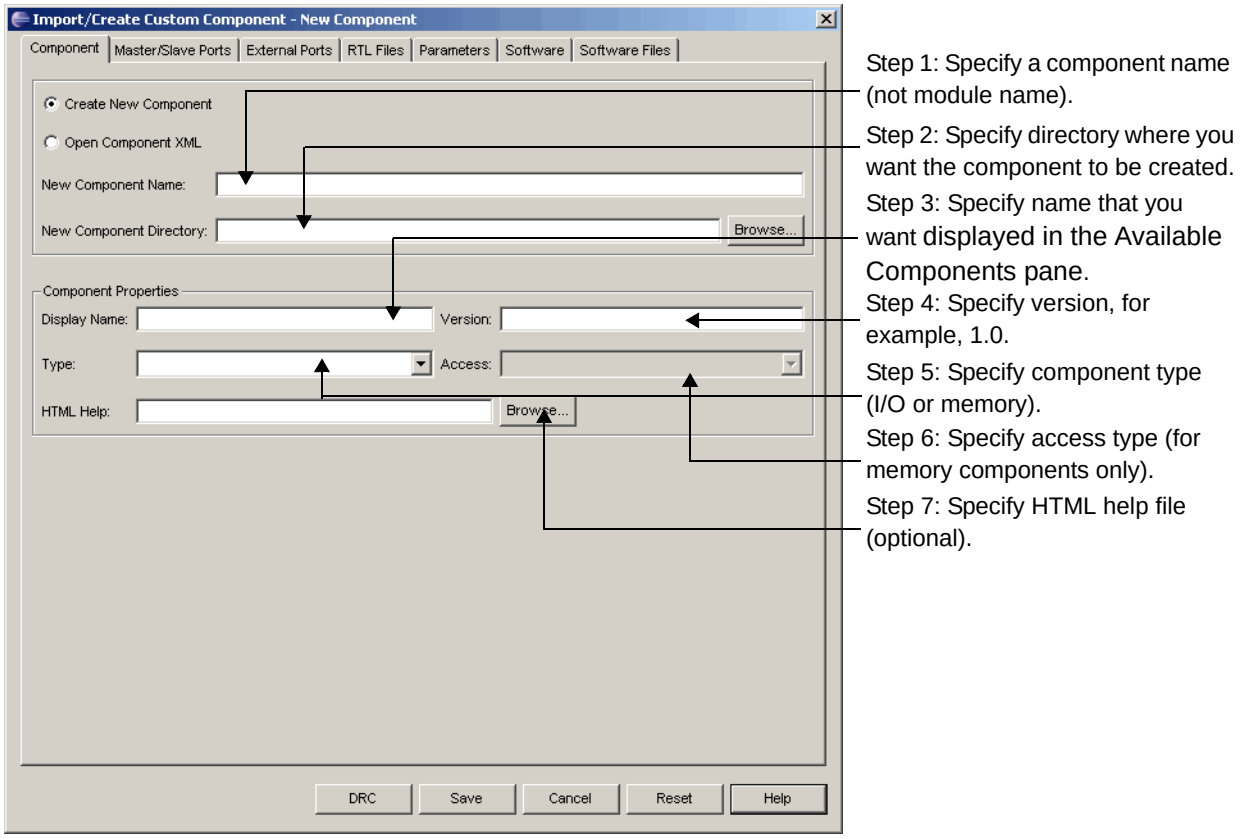


Specifying Component Attributes

The Component tab is the first tab in the Import/Create Custom Component dialog box. It enables you to specify attributes for your custom component. It also provides the location for creating the custom component and the MSB-specific component properties.

Figure 25 shows the steps involved in specifying the component attributes.

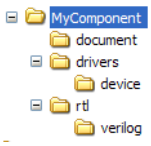
Figure 25: Specifying the Component Attributes



Component Location and Directory Structure

The Import/Create Custom Component dialog box creates the necessary directory structure according to the values that you provide in the New Component Name and New Component Directory boxes. For example, if you enter “MyComponent” in the New Component Name box, MSB creates the directory structure shown in Figure 26. This directory structure is created in the directory specified in the “New Component Directory box. This directory structure is created only after all the information is provided.

Figure 26: Directory Structure Created



The location of the new component is stored in a settings file that MSB uses to identify the available components.

Note

Do not use the top-level module name of your custom component as the component name in this tab. Also, do not specify a directory within the Diamond installation as the location for the new component.

Component Properties

You must set the parameters shown in Table 1 for your new component in the Component tab.

Table 1: Component Tab Options

Option	Description
Create New Component	Creates a new component.
Open Component XML	Opens an existing component description file (<*component_name*>.xml) to edit the existing component.
New Component Name	Specifies the name of the component. The name you enter in this box will be used to create a folder and a component description file (<*component_name*>.xml).
New Component Directory/ Select Component XML	New Component Directory – Specifies the path of the component file. This option is available if you are creating a new custom component and have selected the Create New Component option. Type in the path to your component folder, or use the Browse button to browse to your new component folder. **Note**: Your new component folder should be outside of the .\micosystem folder. Select Component XML – Specifies the path of the component description file. This option is available if you are editing an existing component and have selected the Open Component XML option. Use the Browse button to browse to the component description file (<*component_name*>.xml) of the component that you want to edit.
Display Name	Specifies the name of your component that you want MSB to display when it appears in the Available Components window. It is not used for a folder or a file name, so any combination of ASCII characters is permitted.
Version	Specifies the version number of the component that you want MSB to display next to the display name in the MSB Available Components window.
Type	Specifies the type of component: I/O or memory. MSB supports only these two types of components. The type of component determines the address assignment of the component in MSB. I/O components are located in the LatticeMico32 processor's non-cacheable region, and memory components are located in LatticeMico32 processor's cacheable region. Choose IO or Memory from the drop-down menu. Memory components reside in the lower 2G of the LatticeMico32 memory map and can be added to the instruction and data ports. I/O components reside in the upper 2G. The I/O component can only be connected to the data port.

Table 1: Component Tab Options (Continued)

Option	Description
Access	The text entered in this field is not used by MSB for any other purpose than to concatenate to the Display Name. For example, the SPI flash component has read-only access, so the SPI flash can be used to store the read-only or code sections of the .elf file, but it cannot be used for the read-write section of the .elf file. ▶ Read – Stores .elf file .rodata, .boot, and .text sections in memory. ▶ Write – Stores .elf file .bss and .data sections in memory. ▶ Read/Write – Stores any .elf section in memory.
HTML Help	Specifies the name of the help file, if your component has an HTML help file associated with it. It enables you to enter a path to an existing HTML file describing your new component. Once you close the dialog box, this file is copied into the directory structure created by the dialog box, so the original file is no longer referenced. The contents of this file are displayed in the MSB Component Help view. This file is optional.
DRC	Performs a design-rule check of the new component.
Save	Saves all the data currently entered for the component being defined. The Save button performs a DRC to determine if the component is syntactically correct and saves the data. If the DRC fails, a message is displayed indicating that the component has errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If you are going to overwrite an existing component, another message appears that asks permission to overwrite the previous design files.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

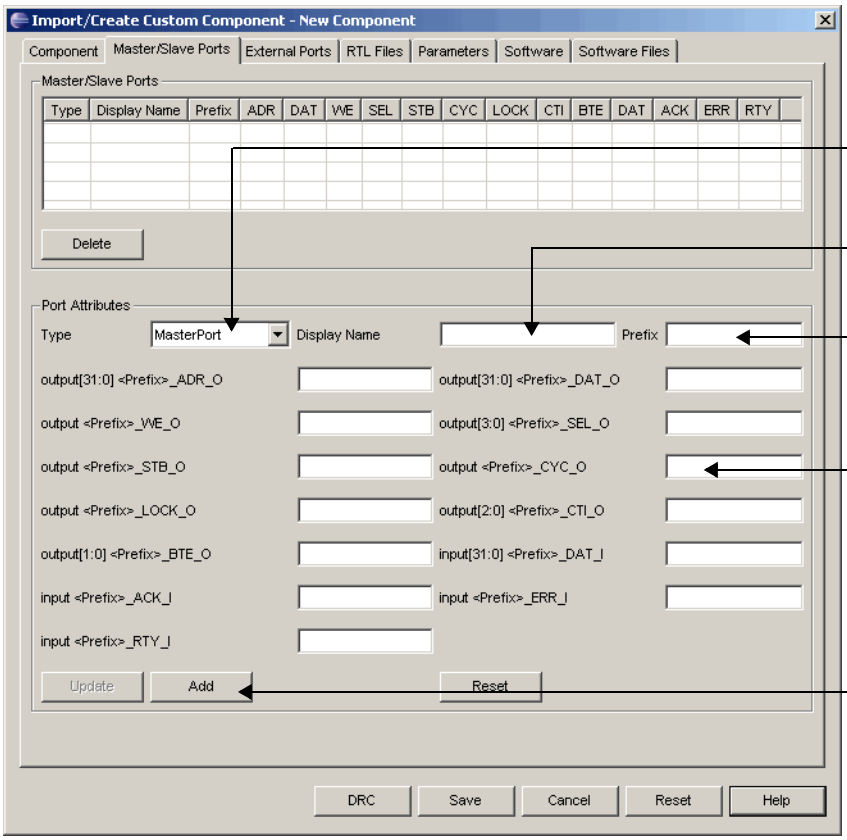
Specifying WISHBONE Interface Connections

Once you have specified the general properties for your custom component, you must specify the WISHBONE interface connections to and from your custom component. You specify the WISHBONE interface connections for master ports and slave ports in the Master/Slave Ports tab of the Import/ Create Custom Component dialog box. Refer to the *LatticeMico32 Processor Reference Manual* for information on WISHBONE port signals.

If your component has a master port—that is, if it can drive the WISHBONE bus—you must specify master port connections. If your component has a slave port—that is, it responds to bus master requests—you must specify the slave port connections. If your component has both types of ports, you must specify both port connections sequentially.

Figure 27 shows the basic steps required for specifying the connections to your component's master or slave port signals.

Figure 27: Specifying Connections to the Master or Slave Port Signals



Step 1: Specify port type (master or slave).

Step 2: Specify port display name as displayed in MSB.

Step 3: Specify signal-name prefix.

Step 4: Specify connections to component's port signals.

Step 5: Click **Add** button.

Not all port signals are mandatory. See Table 3 and Table 4 for a list of port signals required for master and slave port connections.

All of the mandatory fields in the Port Attributes group box must be supplied and the Add button clicked in order for entries to be visible in the Master/Slave Ports group box. You can update an existing master or slave port connection by clicking on the appropriate row in the spreadsheet view. Make any changes to the port highlighted in the Master/Slave Ports group box by modifying the appropriate element in the Port Attributes group box. When the changes are complete, click the Update button to make the changes permanent.

Table 2 lists the options available in the Master/Slave Ports tab of the Import/Create Custom Component dialog box.

Table 2: Master/Slave Port Tab Options

Option	Description
Master/Slave Ports	Lists the master ports and the slave ports in your component. ▶ You can create more than one master port. ▶ You can create only one slave port.
Delete	To delete a master port or a slave port, highlight the port in the Master/Slave Ports list and click **Delete**. The port is not permanently deleted until you save the component by using the Save or OK button. **Note**: You cannot "undo" a port deletion. Once you click Delete and then OK, the port is permanently deleted.
Type	Specifies the type of port. Choose either **MasterPort** or **SlavePort** in the drop-down menu. Master ports can generate WISHBONE cycles to attached WISHBONE slave components. Slave ports cannot initiate WISHBONE bus cycles; they can only respond to a WISHBONE master.
Display Name	Specifies the name of the master or slave port. The name is displayed indented and below the instance name of a component that has been added to a MSB platform. Any ASCII character is permitted in this field.
Prefix	Specifies a prefix that is used for two purposes: ▶ It creates a unique name for the component ports connected to the WISHBONE bus, for example, <*prefix*>_DAT_O. ▶ It enables you to use the same instance name for different components and avoid having name conflicts in the wires of the platform's top level. When the top-level interconnect is built, the wire connecting a slave component output back to the master or masters is named <*instance_name*><*component_port_name*>. Since <*prefix*> is used in <*component_port_name*>, <*prefix*> appears in this wire name. For example, if the prefix for a GPIO component is "GPIO," the wire name will be inst1GPIO_DAT_O.
Port Attributes	The Port Attributes group box is used to perform a name translation between the WISHBONE signal names in your custom component and the WISHBONE signal names attached to the LatticeMico32 bus arbiter. Refer to Table 3 on page 59 for a description of WISHBONE slave port signals.
Update	Updates the options in the Master/Slave Ports tab. The Update button is only available after a master or slave port entry in the Master/Slave Ports group box has been highlighted. The Port Attributes group box displays the values associated with the highlighted master or slave port element. Use the Update button after modifying any of the Port Attribute fields.
Add	Adds the master port or slave port to the Master/Slave Ports list at the top of the dialog box.
Reset	Clears the port attributes.
DRC	Performs a design-rule check of the new component.

Table 2: Master/Slave Port Tab Options (Continued)

Option	Description
Save	Saves all the data currently entered for the component being defined. The Save button performs a DRC to determine if the component is syntactically correct and saves the data. If the DRC fails, a message is displayed indicating that the component has errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If you are going to overwrite an existing component, another message appears that asks permission to overwrite the previous design files.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

Table 3 lists the signals required to connect the master port to the LatticeMico32 platform. Table 4 lists the signals required to connect the slave port to the LatticeMico32 microprocessor. The ports that make up the WISHBONE master or slave port must follow the specifications described in the *LatticeMico32 Processor Reference Manual* table entitled “List of Component Port and Signal Name Suffixes.”

Table 3: LatticeMico32 Master Component WISHBONE Ports

Component Port Names for WISHBONE Slave Port	Direction	Width	Required
<*Prefix*>_ADR_O	Output	32	Yes
<*Prefix*>_DAT_O	Output	32	No
<*Prefix*>_WE_O	Output	1	Yes
<*Prefix*>_SEL_O	Output	4	Yes
<*Prefix*>_STB_O	Output	1	Yes
<*Prefix*>_CYC_O	Output	1	Yes
<*Prefix*>_LOCK_0	Output	1	No
<*Prefix*>_CTI_O	Output	3	No
<*Prefix*>_BTE_O	Output	2	No
<*Prefix*>_DAT_I	Input	32	No
<*Prefix*>_ACK_I	Input	1	Yes
<*Prefix*>_ERR_I	Input	1	No
<*Prefix*>_RTY_I	Input	1	No

Table 4: LatticeMico32 Slave Component WISHBONE Ports

Component Port Names for WISHBONE Slave Port	Direction	Width	Required
<*Prefix*>_ADR_I	Input	32	Yes
<*Prefix*>_DAT_I	Input	32	No
<*Prefix*>_WE_I	Input	1	Yes
<*Prefix*>_SEL_I	Input	4	Yes
<*Prefix*>_STB_I	Input	1	Yes
<*Prefix*>_CYC_I	Input	1	Yes
<*Prefix*>_LOCK_I	Input	1	No
<*Prefix*>_CTI_I	Input	3	‘No
<*Prefix*>_BTE_I	Input	2	No
<*Prefix*>_DAT_O	Output	32	No
<*Prefix*>_ACK_O	Output	1	Yes
<*Prefix*>_ERR_O	Output	1	No
<*Prefix*>_RTY_O	Output	1	No

The example in Figure 28 shows the steps required for specifying a slave port connection on the custom component. Consider the following module definition for a custom component:

In this module definition, there are three sets of signals: slave WISHBONE port signals, mandatory clock/reset signals, and external interface signals specific to the component’s behavior. In the Master/Slave Ports tab, only the slave WISHBONE port signals are added. The mandatory clock/reset and the external interface signals are added in the next tab.

To specify the master and slave port connections:

1. From the drop-down menu in the Type box of the Master/Slave Ports tab, select **SlavePort**, as shown in Figure 29.
2. In the Display Name box, type **slave** so that MSB will display this port’s name as “slave” beneath the component’s instance name.
3. In the Prefix box, enter **S**. The prefix is only used internally within MSB.
4. Enter the component’s corresponding signal names, as shown in Figure 30.
5. Click **Add** to add the port specification for the component.

Figure 28: Specifying a Slave Port Connection

```
module MyVerilogComponent (
    // wishbone interface
        input [31:0] wb_slv_addr,
        input [31:0] wb_slv_master_data,
        input wb_slv_cyc,
        input wb_slv_stb,
        input [3:0] wb_slv_sel,
        input wb_slv_we,
        output [31:0] wb_slv_slave_data,
        output wb_slv_ack,
        output wb_slv_err,
        output wb_slv_rty,

    // mandatory clock/reset signals
        input wb_clk,
        input wb_rst,

    // external interface (optional)
        output [15:0] external_out_bus,
        input  [8:0]  external_in_bus,
        input  external_in_wire,
        output external_out_wire,

    // interrupt signal to the processor(s)
        output interrupt_signal
    );
endmodule
```

Specifying Clock/Reset and External Ports

Connecting the component to the WISHBONE bus enables the LatticeMico32 microprocessor to control and access the custom component. The custom component has its own unique input and output control signals that must be connected outside of the platform to the rest of the system. The External Ports tab enables these control signals to be defined so that MSB can correctly generate a top-level Verilog module. Figure 31 shows the External Ports tab of the Import/Create Custom Components dialog box.

This tab continues the task of building a Verilog wrapper around the custom component. You use this tab to define the CLK_I, RST_I, and optional INTR_O control signals. The component port specifies the signal name presented at the <*platform*>.v top-level module created by MSB when the platform is generated.

Figure 29: Selecting the Slave Port

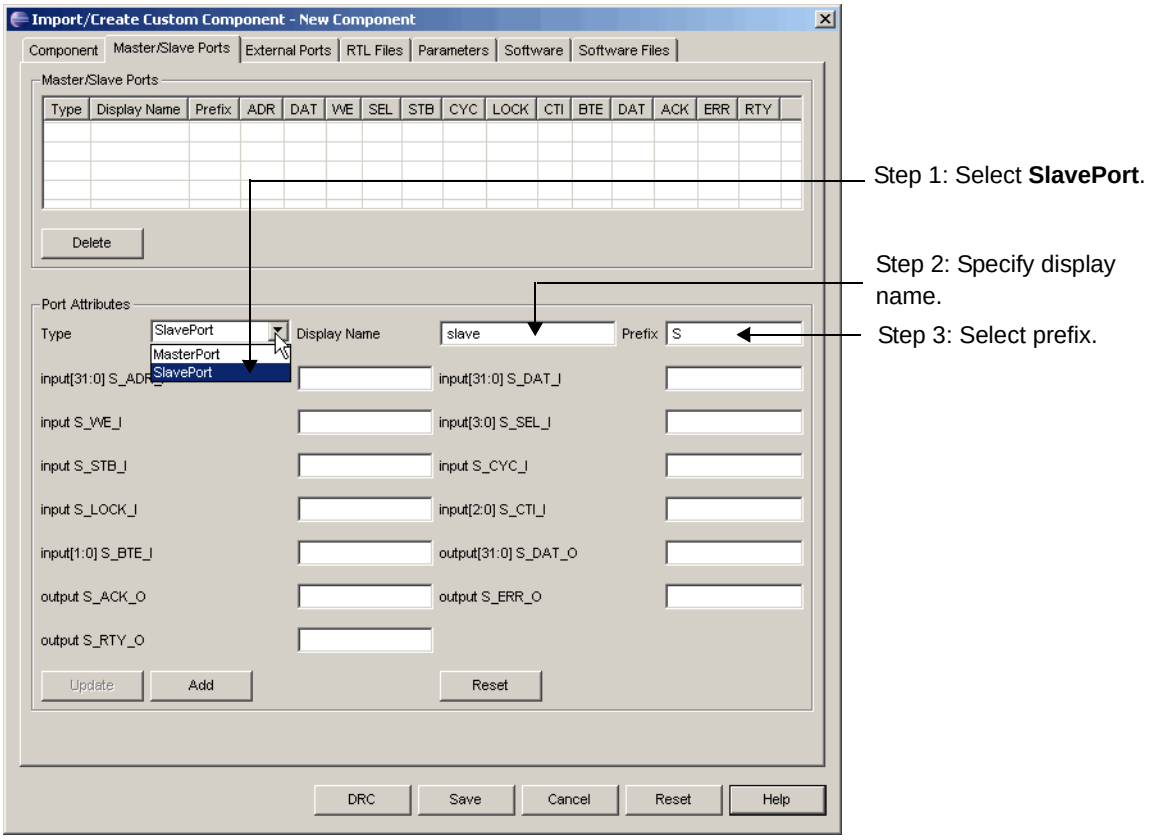


Figure 30: Entering the Signal Names

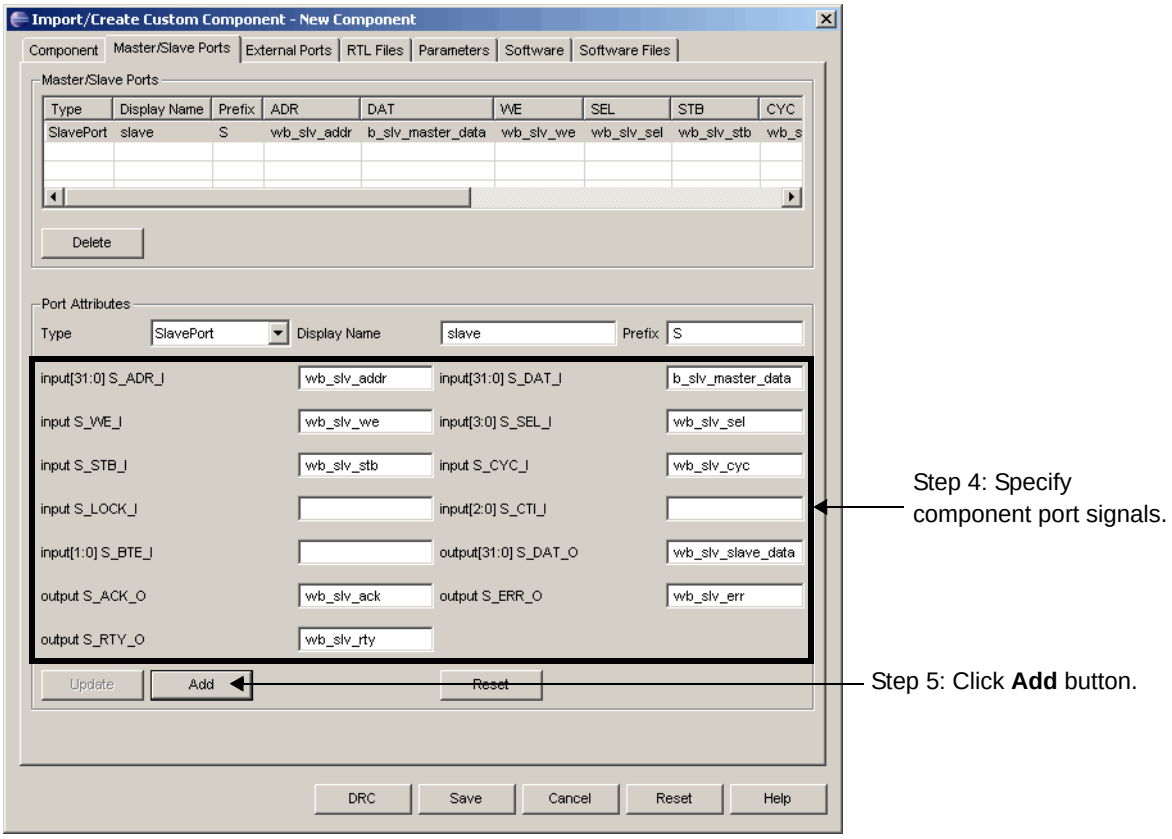
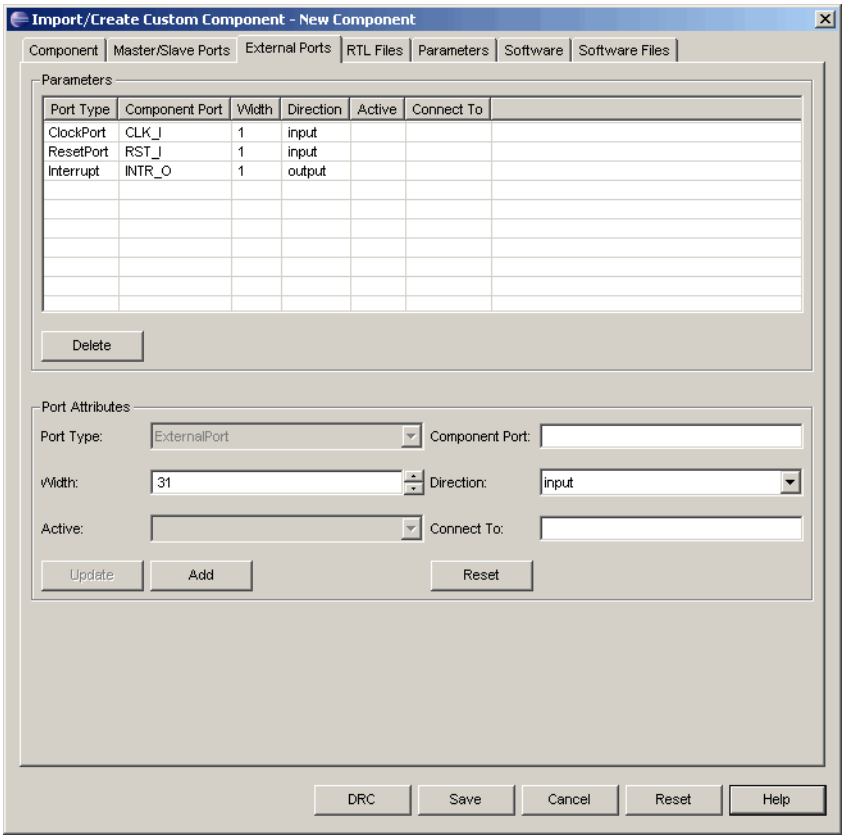


Figure 31: External Ports Tab



The Connect To entry creates a connection from the signal name entered in the Component Port box to a signal on the custom component.

Note

You cannot create dynamic-width input and output ports by using the Import/Create Custom Component dialog box. You must directly edit the XML to create these ports.

Table 5 lists the options available in the External Ports tab of the Import/ Create Custom Component dialog box.

Table 5: External Ports Tab Options

Option	Description
Parameters	Lists the parameters of the external ports in your component. **Note**: ClockPort and ResetPort are mandatory. Interrupt is optional.
Delete	Deletes the selected external port from the Parameters list. **Note**: You cannot “undo” a port deletion. If you click OK, the port will be permanently deleted. You cannot delete the ClockPort, ResetPort, or Interrupt entries.

Table 5: External Ports Tab Options (Continued)

Option	Description
Port Type	Displays the port type of the port selected in the Parameters box (either ClockPort, ResetPort, Interrupt, or ExternalPort). You cannot select or edit this information.
Component Port	Specifies the port name of the wrapper.
Width	Specifies the port width. Enter a number from 1 to 32.
Direction	Enables you to choose the port direction. Choose input, output, or inout from the drop-down menu.
Active	This option is only available if the Port Type is Interrupt. Choose <blank>, High, or Low from the drop-down menu.
Connect To	Specifies the name of the user-defined component port that will be connected to the wrapper port named in the Component Port field.
Update	Updates the port parameters list. Whenever a change is made to the Port Attribute entries that you wish to make permanent, you must click the Update button.
Add	Inserts a new external port into the list of ports that the custom component implements. Fill in each of the active Port Attribute fields and then click Add. If there are no syntax errors, a new entry will be appended to the list of external ports.
Reset	Clears all entries in the Port Attributes group box and permits the entry of a new external port. Use this button if the Add button is not available.
DRC	Performs a design-rule check of the new component.
Save	Adds the custom component to LatticeMico32. If the design-rule check fails, a message appears that warns you that the data to be saved contains errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If the custom component passes the design-rule check, no message box or red “x” appears, and the data is saved. If you are going to override an existing file, another message comes up to ask you for override permission.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

In addition to the mandatory WISHBONE interface, the component being imported or created must have a clock input port for the WISHBONE clock signal and a reset input port for the WISHBONE reset signal. It may optionally have its own set of input ports, output ports, or both, or an interrupt port for connecting an interrupt line to the processor.

The External Ports tab enables you to specify connectivity of the following sets of signals between your custom component’s top-level module and the GUI-generated wrapper module:

- Clock port – The clock signal is provided by the WISHBONE interconnect through the GUI-generated wrapper as an input to your component's clock port. All WISHBONE transactions are synchronized to this clock signal. This port is required.
- Reset port – The reset signal is provided by the WISHBONE interconnect through the GUI-generated wrapper as an input to your component's reset port. This port is required.
- Interrupt port – If your component needs to issue interrupts for the processor to handle, you can specify this output port from your component as an interrupt signal to the processor routed through the GUI-generated wrapper. You cannot specify multiple interrupt ports; that is, your component cannot have more than one interrupt signal to the processor. This port is optional.
- External input/output ports – If your component has input or output ports that must be made available as platform input and output signals (usually for connection to logic external to the platform or for board connection), you can specify these ports in the External Ports tab. This port is optional.

As an example, consider the port definition of a custom component that must be made available in MSB:

Figure 32: Port Definition of a Custom Component

```
module MyVerilogComponent (
// wishbone interface
    input [31:0] wb_slv_addr,
    input [31:0] wb_slv_master_data,
    input wb_slv_cyc,
    input wb_slv_stb,
    input [3:0] wb_slv_sel,
    input wb_slv_we,
    output [31:0] wb_slv_slave_data,
    output wb_slv_ack,
    output wb_slv_err,
    output wb_slv_rty,

// mandatory clock/reset signals
    input wb_clk,
    input wb_rst,

// external interface (optional)
    output [15:0] external_out_bus,
    input  [8:0]  external_in_bus,
    input  external_in_wire,
    output external_out_wire,

// interrupt signal to the processor(s)
    output interrupt_signal
)
```

In this example, the custom component has four external signals that must be made available as platform inputs and outputs. It also requires an interrupt line to be connected to the processor. The component's mandatory clock input port is named "wb_clk," and the mandatory reset port is named "wb_rst."

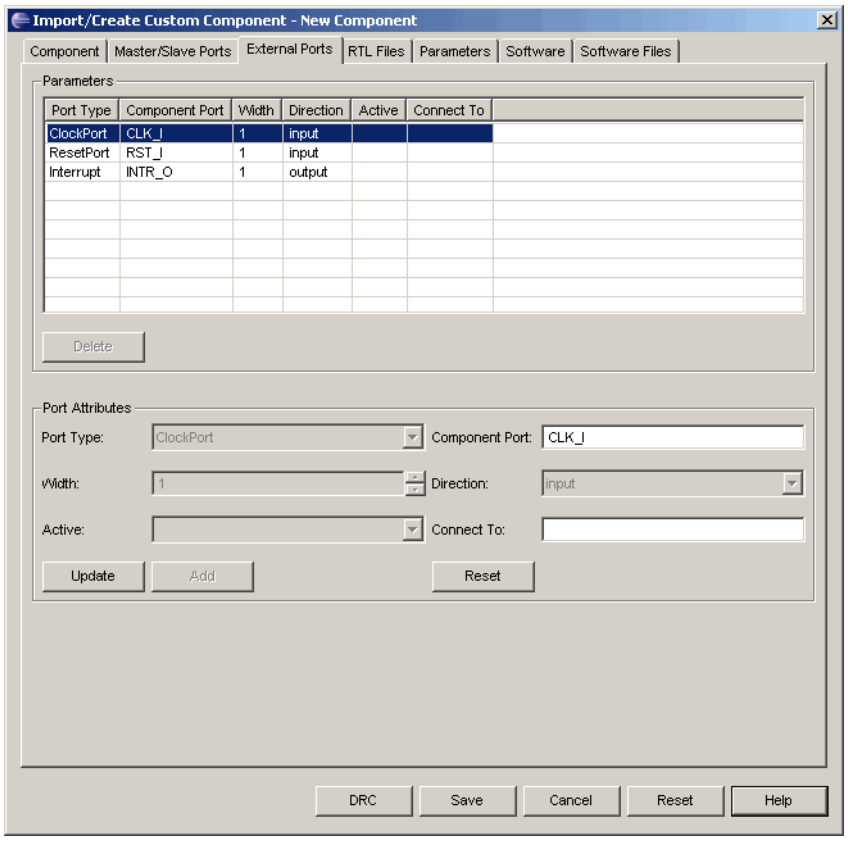
The following steps demonstrate how to connect the reset and clock ports of the GUI-generated reset and clock ports of the custom component.

Specifying Clock/Reset and External Ports

To specify the clock/reset and external port connections:

1. Select the **Clock Port** line in the External Ports tab, as shown in Figure 33.

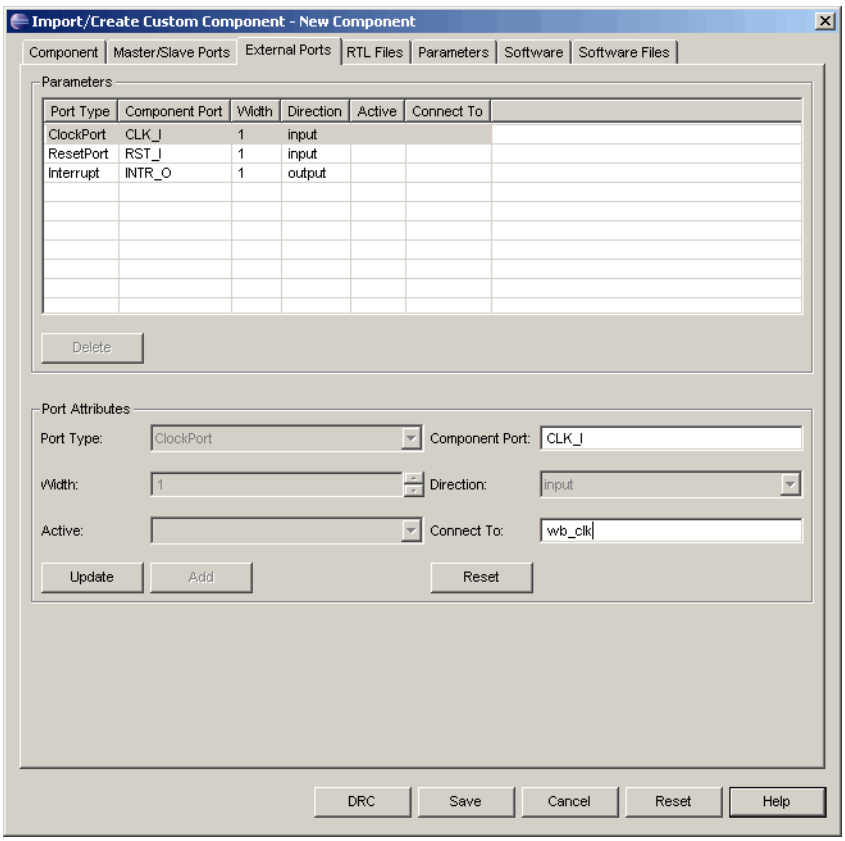
Figure 33: Selecting the Clock Port in the External Ports Tab



2. Enter **wb_clk** in the Connect To box, as shown in Figure 34.
3. Click the **Update** button to update the WISHBONE clock connection specification.
4. Repeat steps 1 through 3, but select the **Reset Port** line in step 1 and enter **wb_rst** in step 2 to specify the Reset port connection.

   The GUI should now look like the figure shown in Figure 35.

Figure 34: Entering wb_clk in Connect To Box



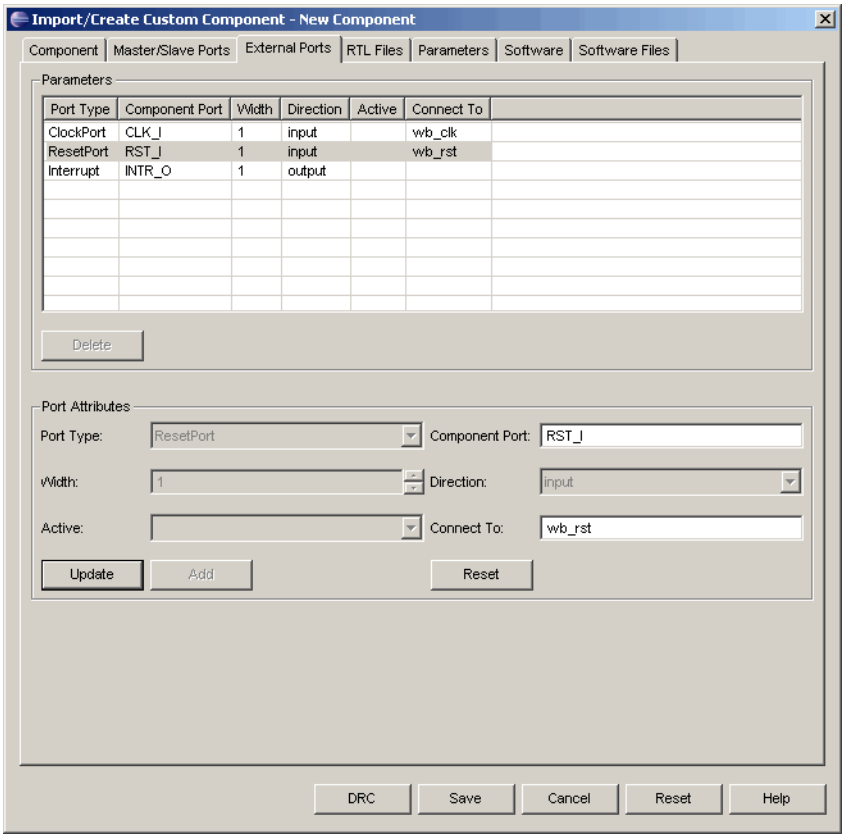
Specifying the Interrupt Port

To specify the interrupt port:

1. Select the **Interrupt** line in the Parameters window.
2. Select the Active drop-down menu and choose **High** or **Low** to specify whether your component's interrupt line is active high or active low.
   - Active high means that your component asserts an interrupt when the selected interrupt port's signal value is high.
   - Active low means that your component asserts an interrupt when the selected interrupt port's signal value is low.

   The MSB platform generator inserts the appropriate logic when connecting your component's interrupt line to the processor according to this specification and when generating a platform that contains your custom component.
3. Enter **interrupt_signal** in the Connect To box to specify the interrupt port connection between your component and the GUI-generated wrapper, as shown in Figure 36.
4. Click the **Update** button to apply this specification.

Figure 35: Selecting the Reset Port in the Parameters Window



Connecting External Output Ports

To connect the external output ports:

1. Click the **Reset** button to specify a new connection.
2. In the **Connect To** box, enter **external_out_bus** to specify a connection to the component's external_out_bus port, as shown in Figure 37.
3. Since this is an output port from the component, select the **Direction** pull-down menu and select **output**.
4. Since the external port is 16 bits wide, enter 16 in the **Width** box.

   The MSB **Run Generator** function creates the LatticeMico32 top-level Verilog file, which exposes the signals from your custom component. For this example, you will make MSB expose the "external_out_bus" from the component as "my_out_bus."

Figure 36: Specifying the Interrupt Port Connection

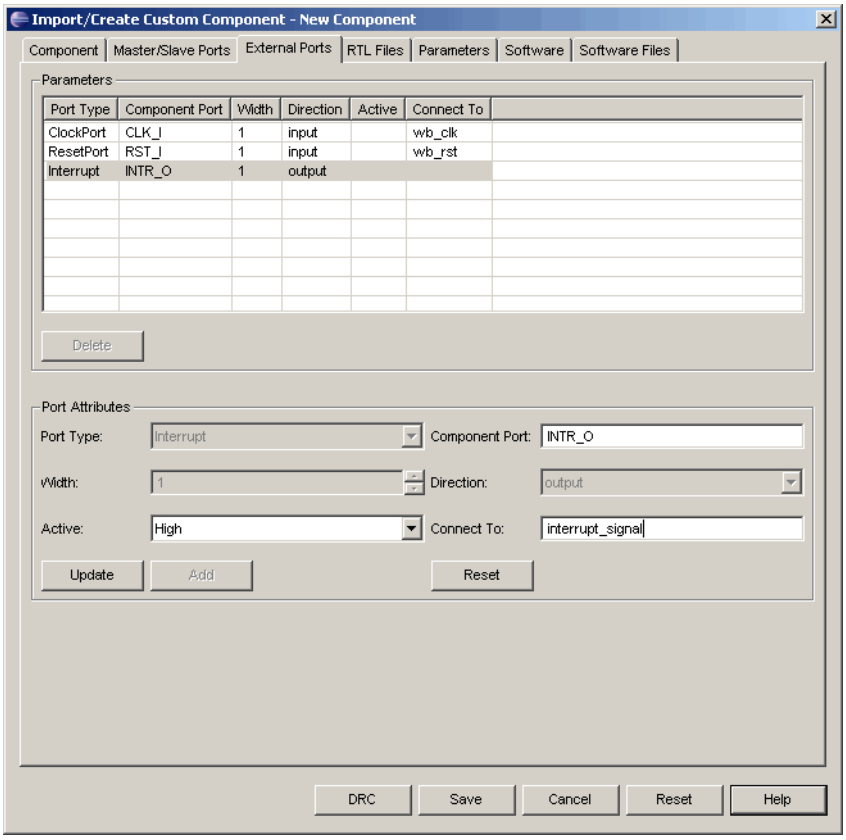
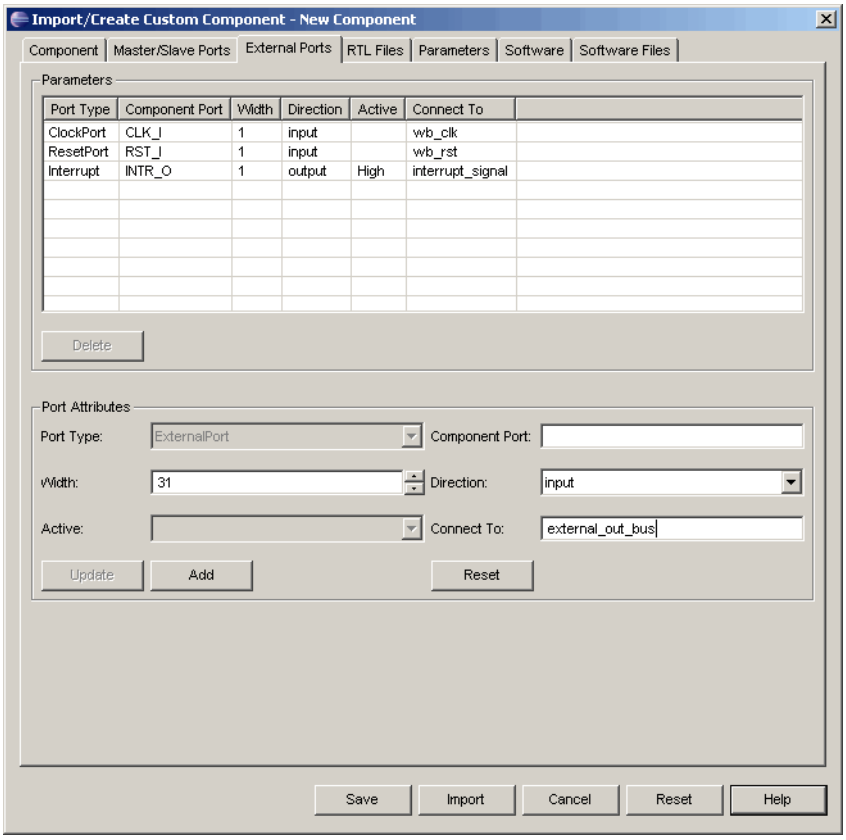


Figure 37: Specifying a Connection to the External Out Bus Port



5. Enter **my_out_bus** in the Component Port box.

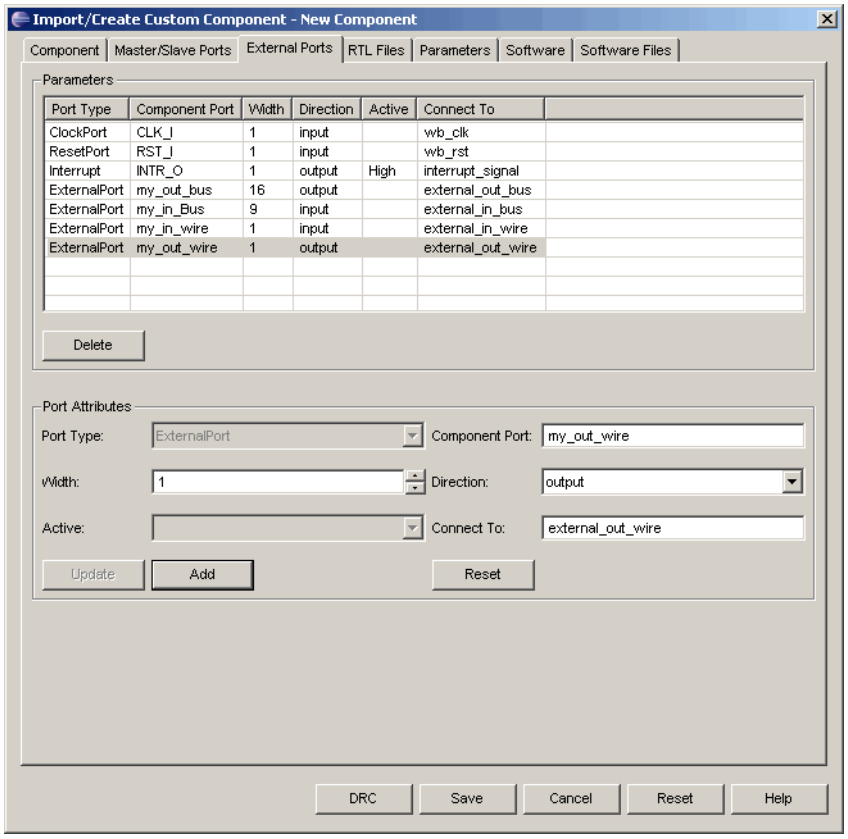
   Note

   MSB applies the platform port-naming convention when it generates the platform, so the actual port name at the top level of the platform is <*instance_name*>my_out_bus, where the <*instance_name*> is the name of the component's instance as entered in MSB.

6. To apply this port specification, select the **Add** button to update the GUI.
7. To add the other port specifications, you can do one of the following:
   - Click the **Reset** button and repeat the steps in this section for the other port specifications.
   - Modify the editable boxes and select the **Add** button to add the other port specifications.

Once you have done this, the External Ports tab should look like Figure 38, completing the step of declaring the component's external interface.

Figure 38: Specifying the External Interface Completed



Specifying RTL Files

Once you have specified the general attributes for your custom component, as well as the port connections to and from your custom component, you must specify the HDL files that implement your custom component. The only acceptable HDL type is Verilog, and the files must have a .v extension.

You specify the Verilog HDL files in the RTL Files tab of the Import/Create Custom Component dialog box.

Figure 39 provides an overview of the steps required in this tab.

Currently the only HDL available in MSB is Verilog. Components written in VHDL must have a Verilog black-box wrapper around them. The VHDL component must be compiled to NGO format independently of MSB, using Lattice Diamond, and placed in the working directory.

Figure 39: Specifying the RTL Files

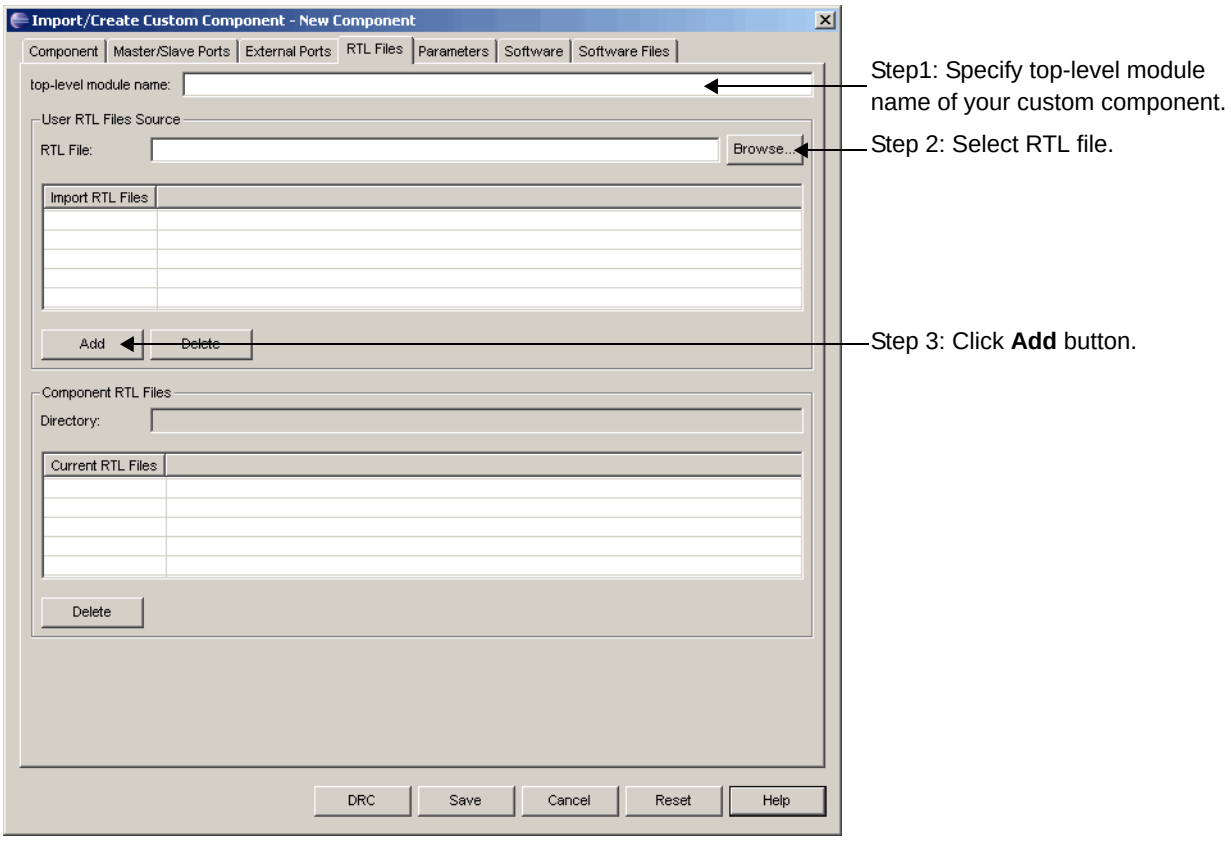


Table 6 lists the options available in the RTL Files tab of the Import/Create Custom Component dialog box.

Table 6: RTL Files Tab Options

Option	Description
Top-Level Module Name	If the custom component is a Verilog component, this option specifies the top-level module name of the custom component. If the custom component is a VHDL component, you must create a Verilog black-box definition of the VHDL custom component and specify the name of the Verilog black-box module. The GUI creates a wrapper that instantiates the top-level module of the Verilog custom component or the Verilog black-box module for a VHDL custom component according to the port specifications for the custom component provided in the Master/Slave Ports and External Ports tabs and the parameters specification provided in the Parameters tab. For VHDL custom components, passing VHDL generics is not supported, since the VHDL custom component flow relies on .ngo files. Refer to “Creating the Verilog Wrapper for VHDL Designs” on page 87 for more information on importing VHDL WISHBONE-compliant custom components.
RTL File	This entry box enables you to enter a file name containing HDL code that is a part of your component. Enter a path and module name directly or use the Browse button to add HDL files interactively.

Table 6: RTL Files Tab Options (Continued)

Option	Description
Add	Click the Add button to insert the file listed in the RTL File entry box to the list of files displayed in the Import RTL Files table. Clicking Add does not update the Component RTL Files group box. The Component RTL Files group box is only populated when you edit an existing custom component, not when you create a new custom component.
Delete	Use this button to delete files from the Import RTL Files list. Highlight the file that you wish to remove from the list and click **Delete**.
Directory	When a custom component is being edited, the Directory text box shows the path to the .rtl files currently associated with the component. You cannot edit this field.
Delete	The Delete button in the Component RTL Files group box enables you to remove files already associated with a custom component. Highlight the HDL file that you want to remove and click **Delete**.
DRC	Performs a design-rule check of the new component.
Save	Adds the custom component to LatticeMico32. If the design-rule check fails, a message appears that warns you that the data to be saved contains errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If the custom component passes the design-rule check, no message box or red “x” appears, and the data is saved. If you are going to override an existing file, another message comes up to ask you for override permission.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

Specifying User-Configurable Parameters

One of the primary advantages in using an FPGA-based microprocessor and custom components connected to that microprocessor is the ability to reconfigure them. Synthesizable HDL code is invariably written so that it can change its capabilities on the basis of a set of parameters assigned during synthesis.

The custom component editor enables you to create a simple user interface for assigning definitions and passed parameters to the component. You use the Parameters tab to create this interface. The values entered into the user interface enable you to:

- Create Verilog definitions and parameters that control how the RTL is synthesized.
- Create C/C++ #define statements to provide information to the firmware controlling the component.

RTL Parameters

You define the RTL parameters by selecting the parameter value in the Flags drop-down box. The other controls in the Parameter Attributes group box determine the properties of the parameter.

MSB uses the parameter in two ways:

- It passes the parameter in the prolog to the Verilog module.
- It stores the parameter as a `define in the soc/system_conf.v source file.

In the first method, the parameters are passed to a specific component instance, so each instance can be configured independently.

In the second case, the `define is a global value, which is useful for configuring every instance of a component, not just a single instance of a component.

Writing code that uses the system_conf.v file to find parameters is not recommended.

RTL Parameter Value Types

The parameters specified in the Parameters tab are made available to you for configuration through a component configuration dialog box in MSB. You enter or select the parameter's value through this component configuration dialog box. In this tab, you can also specify the display behavior for entering the parameter's value in the component configuration dialog box in MSB. You declare these parameters for RTL usage by selecting the flag field as "parameter."

Table 7: RTL Parameter Value Types

Value Type	Description	Allowable Values	RTL Translation Example
Define	Conditional type	def undef	.PARAMETER(1) .PARAMETER(0)
String	Character string type	Any printable characters	.PARAMETER("VALUE")
Integer	Numeric type	Any numeric value	.PARAMETER(VALUE)
List	Numeric type. The difference between Integer and List is that List lets you specify a predefined list of values.	Any numeric value	.PARAMETER(VALUE)
Frequency	Platform frequency (passed by MSB when generating a platform)	MSB provides the platform frequency value (for example, 25 MHz is passed as 25).	.PARAMETER(FREQUENCY_IN_MHz)

Predefined RTL Parameters

The following parameters are not passed to the custom component but are required by MSB when generating the platform RTL:

- Instance Name – Specifies the default instance name assigned by MSB. You can only change the default instance name. You can change this value when instantiating the component in a platform.
- Base Address – Specifies the default base address assigned by MSB. It is overridden by MSB when the custom component is used in a platform if this component is not locked by MSB. You can change this value when you instantiate the component in a platform.
- Size – Specifies the default address space that is assigned to the component, in bytes. This parameter is used by MSB for address decode generation when generating a platform. You can change this value when you instantiate the component in a platform.
- Address Lock – Specifies the default value for “lock,” as used in MSB. You can change this value when you instantiate the component in a platform
- Disable – Specifies the default value for the Disable check box in MSB.

Software Parameters

If your custom component has parameters meant for software use, you make them available to the software by not declaring these parameters as “parameter.” See Figure 41. Parameters used for RTL are also available for software use.

For platform-specific managed-make projects, C/C++ SPE generates a header file named system_conf.h, which enumerates the various parameters and their value types. See Chapter 5 of the *LatticeMico System Software Developer User Guide* for more information on the system_conf.h file.

Table 8 shows how the various value types are translated into this header file.

Table 8: Value Types for Added Parameters

Value Type	Description	Allowable Values	RTL Translation Example
Define	Conditional type	def	#define PARAMETER (1)
		undef	#define PARAMETER (0)
String	Character string type	Any printable characters	#define PARAMETER “VALUE”
Integer	Numeric type	Any numeric value	#define PARAMETER(VALUE)

Table 8: Value Types for Added Parameters

Value Type	Description	Allowable Values	RTL Translation Example
List	Numeric type. The difference between integer and List is that List lets you specify a predefined list of values.	Any numeric value	#define PARAMETER (VALUE)
Frequency	Platform frequency (passed by MSB when generating a platform)	MSB provides the platform frequency value, and SPE translates it to CPU_FREQUENCY macro (e.g. 25MHz is passed as 25000000)	#define CPU_FREQUENCY (FREQUENCY_IN_HERTZ)

Predefined Software Parameters

A single predefined software parameter, CharIODevice, enables C/C++ SPE to determine if your component supports character file input and output operations. The default value of this parameter is set to “undef.” If, however, your component (for example, UART) supports character file input and output operations, you can set the value of this parameter to “def.”
C/C++ SPE makes instances of this component available as standard input and output device selections when creating a managed-make C/C++ project.

You cannot change this parameter’s value when instantiating the component in a platform. It applies to all instances of the component in a platform. C/C++ SPE ignores this parameter and its value for components declared as memory components. Refer to Chapter 3 and Chapter 4 of the *LatticeMico System Software Developer User Guide* for more information on the file support implementation for LatticeMico32.

GUI Presentation

The MSB perspective displays a configuration dialog box for your custom component when you try to insert your component into a platform. This dialog box enables you to modify the parameter values through a GUI interface. The available GUI widgets for configuring parameters are:

- Check – Enables you to select or deselect a parameter.
- Radio – Enables you to select one of multiple parameters.
- Text – Enables you to enter a value.
- Combo – Enables you to select pre-determined values from a drop-down menu.
- Spinner – Enables you to select a value from a pre-determined range.

Table 9 shows the possible GUI widgets for the various value types.

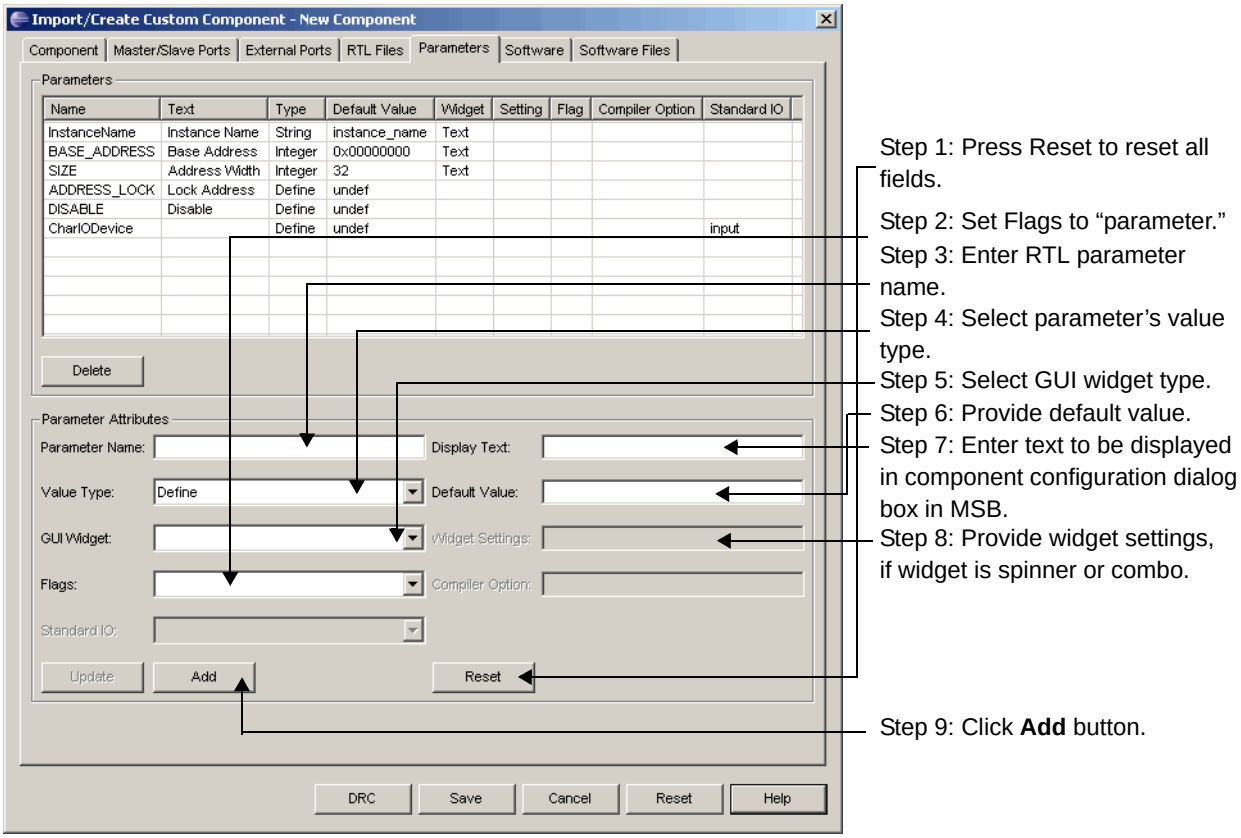
Table 9: GUI Widgets

Value Type	Allowable Widgets
Define	Check, Radio
String	Text
Integer	Text, Spinner
List	Combo
Frequency	Although the Import/Create Custom Component dialog box enables you to specify a widget, MSB overrides and automatically assigns a value to the parameter declared as a Frequency type.

Adding RTL Parameters

Figure 40 shows the steps required for adding RTL parameters.

Figure 40: Steps Involved in Adding RTL Parameters



Adding Non-RTL Parameters

Figure 41 shows the steps required for adding non-RTL parameters.

Figure 41: Steps Involved in Adding Non-RTL Parameters

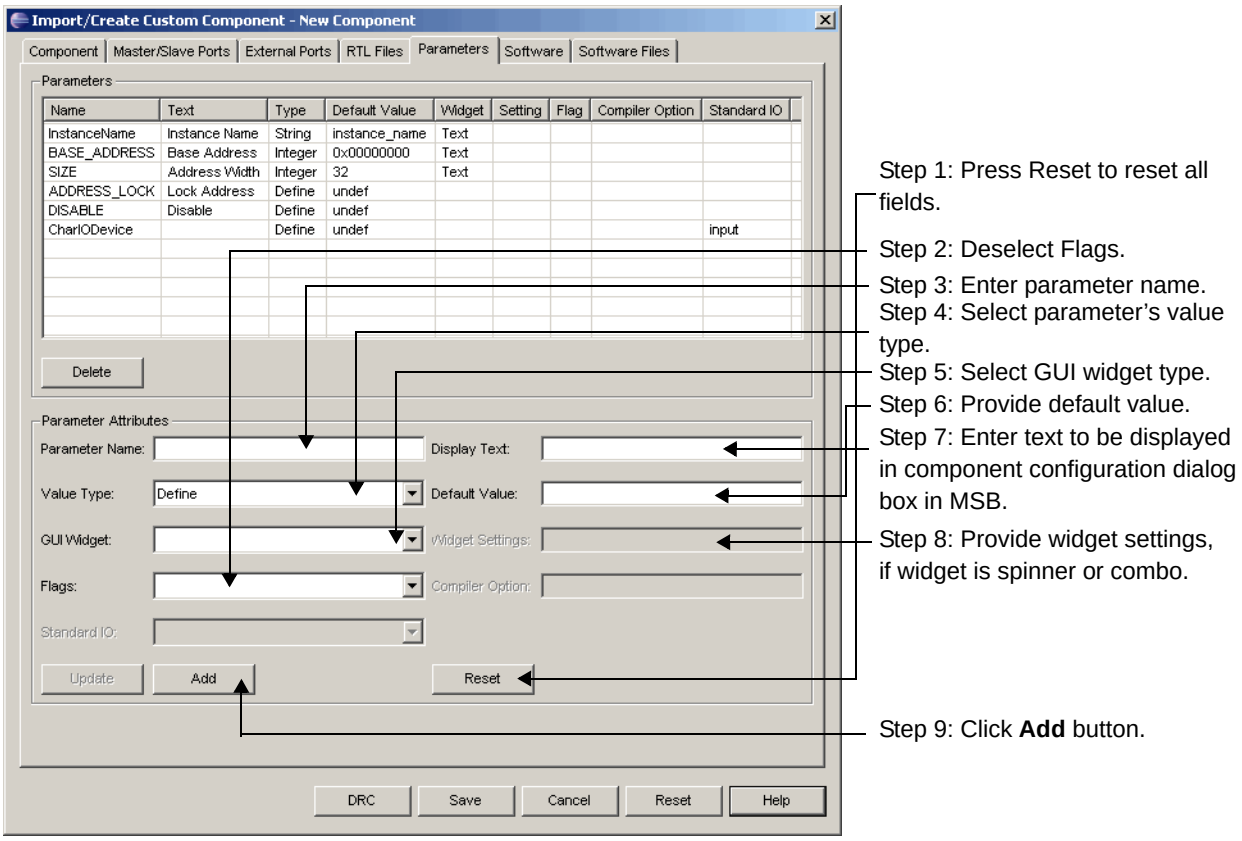


Table 10 lists the options available in the Parameters tab of the Import/Create Custom Component dialog box

Table 10: Parameters Tab Options

Option	Description
Parameter Name	Specifies the name of the parameter to be passed to the Verilog source code. When using Define types, be sure to make the name globally unique.
Display Text	Specifies the display text that will be placed adjacent to the specific control. Each component, when added to the platform, brings up an individualized dialog box. Each element in the dialog box has descriptive text placed adjacent to a control.
Value Type	Specifies the value type. Choose Define, String, Integer, List, or Frequency from the drop-down menu.
Default Value	Specifies how each parameter or `define is initialized when a component is added to the platform. This field is free-form, so you must be careful when entering default values. Any type mismatch or incorrect data entered here will impact the synthesis process later.

Table 10: Parameters Tab Options (Continued)

Option	Description
GUI Widget	Specifies the GUI widget. ▶ If the value type is Define, choose <*blank*>, Radio, or Check from the drop-down menu. ▶ If the value type is String, choose <*blank*> or Text from the drop-down menu. ▶ If the value type is Integer, choose Text or Spinner from the drop-down menu. ▶ If the value type is List, choose <*blank*> or Combo from the drop-down menu. ▶ If the value type is Frequency, choose <*blank*> or Text from the drop-down menu.
WIdget Setting	Specifies the GUI widget setting. If the GUI widget is Combo, enter comma-separated list values. If GUI widget is Spinner, enter minimum and maximum values as a hyphen-separated pair.
Flags	Choose <*blank*>, parameter, or compiler from the drop-down menu. ▶ The parameter flag specifies that it is a Verilog parameter. ▶ The compiler flag specifies that this is a compiler option to be used in C/C++ SPE.
Complier Options	If compiler flag is selected, specify flag or option.
Standard I/O	This option is only available for CharIODevice. Choose <blank>, input, output, or inout from the drop-down menu.
Update	Updates changes.
Add	Adds new parameter.
Reset	Clears the Parameters tab options.
DRC	Performs a design-rule check of the new component.
Save	Adds the custom component to LatticeMico32. If the design-rule check fails, a message appears that warns you that the data to be saved contains errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If the custom component passes the design-rule check, no message box or red “x” appears, and the data is saved. If you are going to override an existing file, another message comes up to ask you for override permission.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

Specifying Software Elements

There are two main software elements that you can optionally specify for a custom component:

- Initialization function name
- A component instance-specific information structure

You can specify these elements through the Software tab of the Import/Create Custom Component dialog box.

Chapter 5 of the *LatticeMico System Software Developer User Guide* contains information that you may find helpful before proceeding with this section.

C/C++ SPE uses the information provided through this tab for managed-make projects or for platform library projects when generating code based on a platform containing this component.

Your software may need to access instance-specific parameters that were configured when you created the platform or instance-specific private data. The C/C++ SPE managed-make process facilitates this process by creating a DDStructs.c source file that contains instance-specific populated data structures and generates the structure definition in the DDStructs.h file by using the C structure definition presented in the Software tab.

The format of the C structure generated in DDStructs.h is shown in Figure 42:

Figure 42: Format of C Structure

```
typedef struct st_STRUCTURE_NAME {
  DATA_TYPE   ELEMENT_NAME;
}STRUCTURE_NAME;
```

You can additionally specify elements of the structure to be initialized with the values configured for parameters when you generate the platform.

The initialization function is invoked by LatticeDDInit as part of the boot process for managed-make projects before calling the “int main (void)” main entry function. You must implement the initialization function. The automatically generated code invokes the initialization function for each instance of the component. The Software tab provides information on the function name that is used when you generate the automatically generated code.

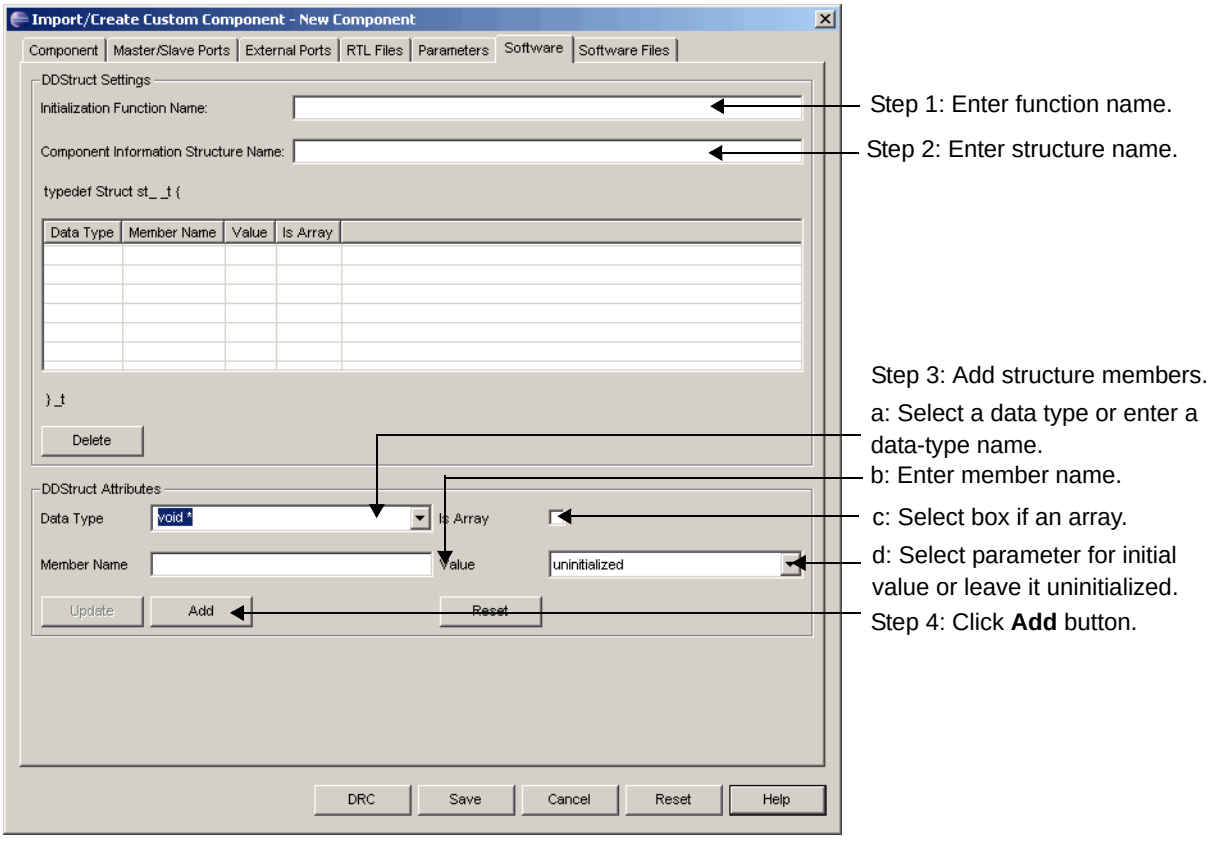
The prototype of the function is as follows:

```
void FUNCTION_NAME( st_STRUCTURE_NAME * );
```

Figure 43 shows the steps required for specifying software elements.

The data type for structure members determines if they can be marked for initialization by the C/C++ SPE managed-make build process. In Figure 43, you select the initial value by selecting an appropriate parameter available in the Value drop-down menu (step 3d). The parameter value types define which

Figure 43: Specifying Software Elements



parameters are listed in the Value drop-down menu on the basis of the structure member's data type.

Table 11 shows the data types that are available for elements of the structure and the parameters available for initializing members of an element if you choose to initialize the element during platform creation.

Table 11: Structure Element Data Types

Data Type	Can Element Be Initialized During Platform Creation?	Parameter Value Type	Notes
void *	No	N/A	
int *	No	N/A	
const char *	Yes	String, List	Values are enclosed in quotation marks.
char *	Yes	String, List	Values are enclosed in quotation marks.
unsigned char *	Yes	String, List	Values are enclosed in quotation marks.
int	Yes	Integer, List, Define	List must be a numeric list.

Table 11: Structure Element Data Types

Data Type	Can Element Be Initialized During Platform Creation?	Parameter Value Type	Notes
unsigned int	Yes	Integer, List, Define	List must be a numeric list.
char	Yes	Integer, List, Define	Values are not enclosed in quotation marks and must be valid numeric values.
unsigned char	Yes	Integer, List, Define	Values are not enclosed in quotation marks and must be valid numeric values.
User-defined	Yes	Any	You are responsible for choosing the right parameter based on the parameter's value type.

Note

If you select the Array box for any member, you must select a "value" parameter. This selected parameter's value is used to determine the array size. C/C++ SPE cannot initialize the array contents, but you can do so in your component's initialization function.

If your component needs to know the interrupt line it is connected to in a platform, you can add an "int" or an "unsigned int" data member and declare its "value" as Interrupt. C/C++ SPE automatically initializes this data member's value to the interrupt line assigned by MSB in the platform when performing a managed build.

Table 12 lists the options available in the Software tab of the Import/Create Custom Component dialog box.

Table 12: Software Tab Options

Option	Description
Initialization Function Name	Specifies the user-defined initialization function name.
Component Information Structure Name	Specifies the name of the DDStruct structure.
Data Type	Specifies the C data type of the DDStruct element being added. The drop-down menu enables you to specify the following C data types: void *, unsigned int, int, int *, const char, unsigned char, unsigned char *, char, or char *.
Member Name	Specifies the name of the DDStruct element being added.
Value	Available values in the drop-down menu depend on the chosen data type.
Is Array	Checks to see if the member name is an array.
Delete	Deletes the highlighted DDStruct setting from the list.
Update	Allows an element already added to the DDStruct to be modified. Highlight the element, make any desired changes to the element, and the click Update to activate the changes.

Table 12: Software Tab Options (Continued)

Option	Description
Add	Adds a new element to the DDStruct structure with the values active in the DDStruct Attributes group box.
Reset	Clears the DDStruct Attributes group box controls.
DRC	Performs a design-rule check of the new component.
Save	Adds the custom component to LatticeMico32. If the design-rule check fails, a message appears that warns you that the data to be saved contains errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If the custom component passes the design-rule check, no message box or red “x” appears, and the data is saved. If you are going to override an existing file, another message comes up to ask you for override permission.
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

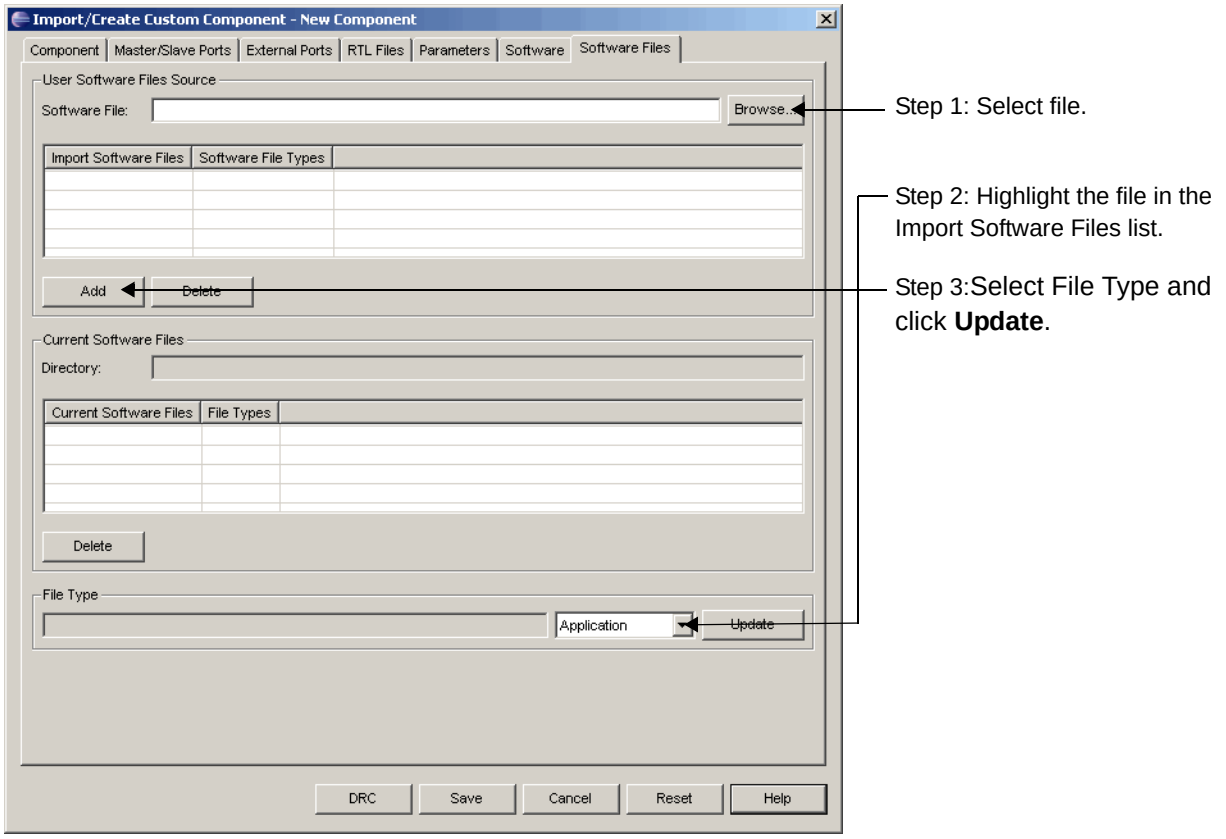
Adding Software Files to Custom Components

If your custom component provides software support, such as the component initialization function noted in the previous section, you can optionally identify these files in the Software Files tab of the Import/Create Custom Component dialog box.

The Software Files tab enables you to import C software files that pertain to your custom component. You can specify the file to be part of a managed build.

Figure 44 shows the steps required for adding software support files.

Figure 44: Adding Software Support Files



The file types that you can select by using the Browse button and their extensions are shown in Table 13.

Table 13: File Extensions and File Types of Imported C Software Files

File Extension	File Type
.c, .C	C language source file
.cpp, .CPP	C++ language source file
.s, .S	Assembly language source file
.h, .H	C/C++ language header file

For the managed build framework, the software files are classified as follows:

- Application file type – These source files are compiled and linked as part of the application build process instead of being compiled during the platform library build process and becoming part of the platform library archive.
- Platform library file type – These source files are compiled during the platform library build process and become part of the platform library archive. The functions in these source-code files can be overridden by

implementing them in source files that are compiled as part of the application build step.

- Structure header file type – These header files contain declarations that your device driver structure may reference. These files are included as preprocessor “include” statements in the automatically generated DDStructs.h header file. See Chapter 5 of the *LatticeMico System Software Developer User Guide* for information on the DDStructs.h header file.
- Header file type – These files are header files that are needed by the component source files but are not required by your device driver structure.

The application file type and platform library file type govern the composition of the component makefile.

See Chapter 5 of the *LatticeMico System Software Developer User’s Guide* for more information on the managed-build process.

Imported files do not become part of the component until you click the OK button and save the component without error. The Current Software Files portion of the Software Files tab only displays software files if an existing custom component is being edited.

Table 14 lists the options available in the Software Files tab of the Import/ Create Custom Component dialog box.

Table 14: Software Files Tab Options

Option	Description
Software File	Enables you to browse to the software driver files. Copies the selected file into the component folder.
Add	Adds the file currently listed in the Software File entry box to the table of Imported Files.
Delete	Deletes the highlighted entry in the Import Software Files table.
Directory	Displays the folder where the source code files associated with the component being edited reside.
Delete	Deletes a source file already associated with the component being edited.
File Type	Specifies the file type. Choose the following from the drop-down menu: Application, Platform Library, Structure Header, Header.
DRC	Performs a design-rule check of the new component.
Save	Adds the custom component to LatticeMico32. If the design-rule check fails, a message appears that warns you that the data to be saved contains errors and cannot be used in a platform. The component icon displays a small red “x” in the bottom left-hand corner. If the custom component passes the design-rule check, no message box or red “x” appears, and the data is saved. If you are going to override an existing file, another message comes up to ask you for override permission.

Table 14: Software Files Tab Options (Continued)

Option	Description
Cancel	Cancels the actions and closes the dialog box. If you did not save your changes, a message box comes up to warn you that the changed data will be lost.
Reset	Resets all values in all tabs in the dialog box.
Help	Displays the help for the dialog box.

Note

Older versions of LatticeMico System Builder do not provide a member element of the type DeviceReg_t in the DDStruct C structure by default. You can add it as an element of user-defined data type. The DDStruct Attributes to be specified, as shown in Figure 43, are as follows:

Data Type: DeviceReg_t

Member Name: lookupReg

Value: Uninitialized

Applying Changes

To apply the changes that you have made in the tabs of the Import/Create Custom Component dialog box, select the **OK** button at the bottom of the dialog box. LatticeMico System now performs design-rule checks. If it finds no errors, the dialog box will close and the custom component will appear in the MSB perspective.

If you need to re-edit the added custom component, select that component and open the Import/Create Custom Component dialog box.

Creating the Verilog Wrapper for VHDL Designs

If you are creating custom components for VHDL designs, you must create a Verilog wrapper before you proceed with creating a new custom component.

This section explains how to create and use new custom components in the flow for VHDL users.

To create a Verilog wrapper:

1. Create a component definition in VHDL that is LatticeMico32-compliant, for example, using WISHBONE. Refer to the section “WISHBONE Interconnect Architecture” in the *LatticeMico32 Processor Reference Manual* for information.
2. Create a completely new Diamond project that will be used just for processing this component. This project is completely distinct from the project that will eventually use this component.

3. Import the VHDL source code into the project. During synthesis, turn off I/O insertion by following these steps:

   a. Select the File List tab in Diamond and double-click the name of the currently active strategy, which is displayed in bold type.

   b. In the Strategies dialog box, expand the Synthesis folder and select the synthesis tool you will be using.

   c. In the synthesis pane on the right, set Disable IO Insertion to **True** and click **OK**.

   d. In Diamond, select the Process tab , and double-click **Translate Design**.

      Diamond now generates the <*platform*>.ngo file.

4. Create a black-box declaration of the component in Verilog.

   This declaration represents this component in any platform generated by MSB that uses this component. It is combined with the .ngo file previously created (that holds the actual functionality of the component) after synthesis in the Translate Design process. If there are any bidirectional I/Os in the custom VHDL component, you must declare them as black-box pads. Lattice Semiconductor FPGAs only have tristate buffers in their I/O cells. In a single-language implementation, the synthesis tool can reconcile multiple tristate I/O requests to a single tristate buffer. In the dual-language implementation, the Verilog wrapper has no visibility into the VHDL .ngo black-box element, preventing any reconciliation of multiple tristate buffers. The black-box pad declaration directs the synthesis process not to create a second set of tristate buffers because tristate buffers have already been created for these black-box ports.

   Figure 45 is an example Verilog black-box definition of a VHDL custom component illustrating the black_box_pad declaration in the Verilog black-box definition for the VHDL custom component's inout port. This Verilog black-box definition is the RTL input file for the custom component GUI.

   Note

   The Verilog module name must match the .ngo file name in order for Diamond to correctly link the .ngo contents to the Verilog wrapper.

5. Perform the user-defined component flow explained at the beginning of this chapter to bring a user-defined Verilog component into MSB. The Verilog component RTL file entry is the Verilog black-box file that you created in step 4.

If there are tristate (bidirectional) I/Os in the custom VHDL component, you must also add the black_box_pad_pin attribute of these ports to the VHDL wrapper files' component declaration section. The black_box_pad_pin attribute is a synthesis directive that specifies pins on a user-defined black-box component as I/O pads that are visible to the environment outside of the black box. Because the I/O primitives are added to the tristate (bidirectional) I/Os in the .ngo file, adding the black_box_pad_pin attribute to these I/Os enables the top-level VHDL RTL code to recognize them.

Figure 45: Verilog Black-Box Definition of a VHDL Custom Component

```
module vhdl_custom (
          // wishbone slave signals
          input[31:0] ADR_I,
          input[31:0] DAT_I,
          input WE_I,
          input[3:0] SEL_I,
          input STB_I,
          input CYC_I,
          input LOCK_I,
          input[2:0] CTI_I,
          input[1:0] BTE_I,
          output[31:0] DAT_O,
          output ACK_O,

          // external signals
          output [30:0] custom_ext,
          inout [30:0] custom_ext_io,
          input CLK_I,
          input RST_I,
          output INTR_O  )/*synthesis syn_black_box
black_box_pad_pin = "custom_ext_io[30:0]" */;
endmodule
```

Pointing to the Correct .ngo File

You must enable the Translate Design step to use the correct .ngo file of the VHDL-based component that was created in an earlier step.

To point to the correct .ngo file:

1. Copy the .ngo file to the \soc directory, located in the LatticeMico32 platform project directory.
2. In Diamond, choose **Project > Property Pages**.
3. In the Macro Path box of the Project Properties dialog box, provide the path to the LatticeMico32 Platform project's \soc directory that contains the .ngo file copied in step 1. You can provide an absolute or relative path.

   For example, an absolute path might be

   c:\ispTOOLS<*version*>\examples\VHDL_Test\LM32_Platform\soc.

   A relative path would be a path relative to the Diamond project directory. For example, if the LatticeMico32 platform project directory is contained in the Diamond project, the relative path might be

   .\LM32_platform\soc

   If the Diamond project is contained in the LatticeMico32 platform project directory, the relative path would be simply

   .\soc

If your directory structure is not one of these, use your best guess to provide either the relative path or absolute path and see if Diamond issues an error message saying that it cannot expand the .ngo definition.

Making Custom Components Available in MSB

LatticeMico System performs the following steps automatically through the custom component dialog box to make your custom component available in MSB. You do not have to take any action. This section is for information only.

Integrating Custom Component's RTL Design Files

The key step in connecting your custom component in a platform is to tie the WISHBONE interface signals (master or slave) to the automatically generated arbitration logic. The platform generator imposes internal naming conventions, so a Verilog wrapper is created to implicitly enforce the internal naming conventions without informing you. The automatically generated Verilog wrapper instantiates your custom component, allowing MSB to connect the custom component to the rest of the platform.

Saving the Settings

All the settings specified in the Import/Create Custom Component dialog box are saved in a component description file in XML format. Also, the RTL and the software files provided to the Import/Create Custom Component dialog box are copied to the component creation directory.

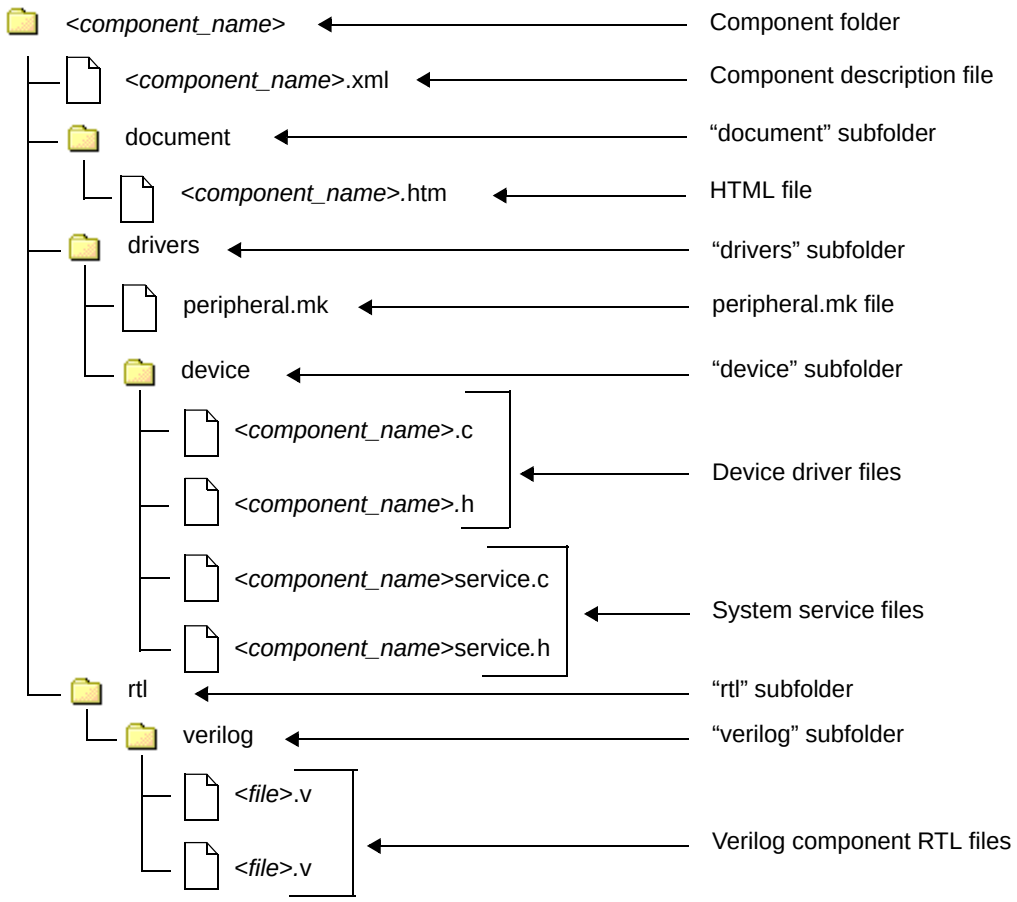
Directory Structure

Figure 46 shows a typical directory and file structure that LatticeMico System generates for a LatticeMico32 component.

The following is a brief description of the folders and files contained in a typical custom component folder:

- <*component_name*> folder – Contains the following files and directories:
  - <*component_name*>.xml – Contains the XML code required to attach your component to the LatticeMico32 processor.
  - document folder – Contains documentation file or files. At a minimum, this folder contains the <*component_name*>.htm file, which is an HTML file that is displayed in the Component Help view in the MSB main window.
  - drivers folder – Contains the peripheral.mk file, which is used to direct C/C++ System Programming Environment (SPE) to the C/assembly

Figure 46: Typical Component Folder and File Structure



driver files containing user-defined application programming interfaces (APIs).

Also inside the drivers folder are two subdirectories:

- Device driver files (<*component_name*>.c and <*component_name*>.h). The device driver files define the API function calls available to the C/C++ SPE developer. The functions are user-defined according to the specific needs of the custom component.
- System service driver files (<*component_name*>Service.c and <*component_name*>Service.h). The service files must be implemented to support the LatticeMico32 initialization process. Each component must define a basic set of service functions that have been defined by the LatticeMico32 boot process.

- rtl folder – Contains the verilog subfolder.
  - verilog folder – Contains the component Verilog RTL files.

Custom Component Example

The example in this section shows you how to add a custom component to the MSB graphical user interface so that it is available for use in other platforms.

This example demonstrates how to:

- Make the created component available in MSB.
- Provide a component customization dialog box for configuring RTL instantiation parameters.
- Add software support files and generate instance-specific data structures.

Sample Custom Component

This example includes a custom component that uses a Verilog RTL implementation file and software driver files as sources, typical sources for importing a custom component.

Verilog RTL Implementation

The Verilog (.v) source file for this example is shown in Figure 47.

Figure 47: Verilog (.v) File

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//
// A simple register device with three registers:
// ----------------------------------------------
//
////////////////////////////////////////////////////////////////////////////////
module wb_reg_dev
    #(
        parameter CLK_MHZ = 25,
        parameter reg_08_int_val = 32'h1234abcd
    )
    (
        //------------------------------------------------------------------------
        //
        // WISHBONE clock/reset signals
        //
        //------------------------------------------------------------------------
        wb_reset,//-----------------WISHBONE reset
        wb_clk,//-------------------WISHBONE clock
        //------------------------------------------------------------------------
        //
// WISHBONE interface signals below.
        // - This component does not support burst transfers.
        //
        //------------------------------------------------------------------------
        wb_adr,//-------------------Address from master
        wb_master_data,//-----------Data from master
        wb_cyc,//-------------------WISHBONE cycle-valid qualifier
        wb_stb,//-------------------WISHBONE transfer qualifier
        wb_sel,//-------------------Data byte-lane selection
        wb_we,//--------------------Write-enable
        wb_slave_data,//------------Data from slave
        wb_ack,//-------------------Data-valid qualifier from slave
        wb_err,//-------------------Error qualifier from slave (never asserted)
        wb_rty,//-------------------Retry qualifier from slave (never asserted)
        //------------------------------------------------------------------------
        //
        // Interrupt line (active-high) that will be connected to the
        // processor. Not used but for demonstrating custom component
        // connectivity.
        //
```

Figure 47: Verilog (.v) File (Continued)

```
        //
        //------------------------------------------------------------------------
        wb_intr,//------------------Interrupt request from slave
        //------------------------------------------------------------------------
        //
        // External pins exposed by this component
        //
        //------------------------------------------------------------------------
        out_pins//------------------Output pins
    );
    input wb_reset;
    input wb_clk;
    input [31:0] wb_adr;
    input [31:0] wb_master_data;
    input wb_cyc;
    input wb_stb;
    input [3:0] wb_sel;
    input wb_we;
    output [31:0] wb_slave_data;
    output wb_ack;
    output wb_err;
    output wb_rty;
    output wb_intr;
    output [7:0] out_pins;
    //------------------------------------------------------------------------
    //
    // Registers
    //
    // reg_00 : read/write 32-bit register, general purpose
    //
    // reg_04 : read-only 32-bit register that contains the WISHBONE
    // platform clock frequency (MHz)
    //
    // reg_08 : read-only register that contains a constant specified when
    // instantiating this component in a platform
    //
    //------------------------------------------------------------------------
    reg [31:0] reg_00;//------------------------32-bits, RW, offset 0
    reg [31:0] reg_04;//------------------------32-bits, RW, offset 4
    //reg_08 constant ------------------------32-bits, RO, offset 8
    reg write_ack;//-----------------------------write-ack
    //------------------------------------------------------------------------
    //
// Wires
    //
    //------------------------------------------------------------------------
    wire reg_00_sel;//--------------------------reg_00 selected
    wire reg_04_sel;//--------------------------reg_04 selected
    wire reg_08_sel;//--------------------------reg_08 selected
    wire read_ack;//----------------------------read-ack
    wire [31:0] read_data;//---------------------reg data mux (reads)
    //
// assign register-select signals:
    // since there are only two registers, use bit-2 of the
    // address bus since addressing is word addressing for LatticeMico32
    //
```

Figure 47: Verilog (.v) File (Continued)

```
 assign reg_00_sel = ((wb_stb == 1'b1) && (wb_cyc == 1'b1)
                      && (wb_adr[3:2] == 2'b00) ) ? 1'b1 : 1'b0;
    assign reg_04_sel = ((wb_stb == 1'b1) && (wb_cyc == 1'b1)
                      && (wb_adr[3:2] == 2'b01) ) ? 1'b1 : 1'b0;
    assign reg_08_sel = ((wb_stb == 1'b1) && (wb_cyc == 1'b1)
                      && (wb_adr[3:2] == 2'b10) ) ? 1'b1 : 1'b0;
    //
    // assign read ack: unregistered as data is presented
    // immediately. Can make it registered to improve timing
    //
    assign read_ack = ((wb_stb == 1'b1) && (wb_cyc == 1'b1)
                      && (wb_we == 1'b0)) ? 1'b1:1'b0;
    //
    // assign asynchronous data-output mux
    //
    assign read_data =  (reg_00_sel == 1'b1)? reg_00 :
                      (reg_04_sel == 1'b1)? reg_04 :
                      (reg_08_sel == 1'b1)? reg_08_int_val :
                      32'hdeadbeef;
    //
    // assign write-ack: registered
    //
    always @( posedge wb_clk or posedge wb_reset )
        if ( wb_reset ) begin
            write_ack <= 0;
        end
        else begin
            if( (wb_stb == 1'b1) && (wb_cyc == 1'b1) &&
                (wb_we == 1'b1) && (write_ack == 1'b0) ) begin
                write_ack <= 1'b1;
            end
            else begin
                write_ack <= 1'b0;
            end
        end
    //
// register_00 write process: supports byte-writes
    //
    always @( posedge wb_clk or posedge wb_reset )
        if ( wb_reset ) begin
            reg_00 <= 32'b0;
        end
        else begin
            if ( (reg_00_sel == 1'b1) && (wb_we == 1'b1) &&
                     (write_ack == 1'b0) ) begin
                if( wb_sel[0] == 1'b1 ) begin
                    reg_00[7:0] <= wb_master_data[7:0];
                end
                if( wb_sel[1] == 1'b1 ) begin
                    reg_00[15:8] <= wb_master_data[15:8];
                end
                if( wb_sel[2] == 1'b1 ) begin
                    reg_00[23:16] <= wb_master_data[23:16];
                end
```

Figure 47: Verilog (.v) File (Continued)

```
                if( wb_sel[3] == 1'b1 ) begin
                   reg_00[31:24] <= wb_master_data[31:24];
               end
           end
       end
  //
    // register_04 write process: supports byte-writes
    //
    function integer i_clk_mhz;
        input integer clk_mhz;
        begin
            i_clk_mhz = clk_mhz;
        end
    endfunction // i_clk_mhz

    parameter CLK_MHZ_INT_VALUE = i_clk_mhz(CLK_MHZ);

    always @( posedge wb_clk or posedge wb_reset )
        if ( wb_reset ) begin
            reg_04 <= CLK_MHZ_INT_VALUE;
        end
        else begin
            if ( (reg_04_sel == 1'b1) && (wb_we == 1'b1) &&
                (write_ack == 1'b0) ) begin
                if( wb_sel[0] == 1'b1 ) begin
                    reg_04[7:0] <= wb_master_data[7:0];
                end
                if( wb_sel[1] == 1'b1 ) begin
                    reg_04[15:8] <= wb_master_data[15:8];
                end
                if( wb_sel[2] == 1'b1 ) begin
                    reg_04[23:16] <= wb_master_data[23:16];
                end
                if( wb_sel[3] == 1'b1 ) begin
                    reg_04[31:24] <= wb_master_data[31:24];
                end
            end
        end
    //--------------------------------------------------------------------------
    //
    // MODULE OUTPUTS
    //
    //--------------------------------------------------------------------------
    // assign component ack
    assign wb_ack = read_ack | write_ack;
    // assign component data
    assign wb_slave_data = read_data;
    // unused rty/err
    assign wb_rty = 1'b0;
    assign wb_err = 1'b0;
    // unused interrupt (active-high)
    assign wb_intr = 1'b0;
    assign out_pins = reg_00[7:0];

endmodule
```

Software Source Files

Reg_Comp.c File The Reg_Comp.c file, shown in Figure 48, implements the software driver for the custom component.

Figure 48: Reg_Comp.c File

```
#include "Reg_Comp.h"

/* device initialization function */
void init_reg_device( struct st_reg_device * ctx )
{
    /* simply copy initialization data for reg_08
     * provided in the context structure to register-00 */
    REG_DEV_REGISTER(ctx->b_addr,0) =
        ctx->reg_08_value;

    return;
}
```

Reg_Comp.h File The Reg_Comp.h file, shown in Figure 49, is the header file for the software driver.

Figure 49: Reg_Comp.h File

```
#ifndef _REG_COMP_HEADER_FILE_
#define _REG_COMP_HEADER_FILE_

#include "DDStructs.h"

#ifdef __cplusplus
extern "C"{
#endif /* __cplusplus */

/* device initialization function */
void init_reg_device( struct st_reg_device * ctx );


/* macro for reading/writing registers */
#define REG_DEV_REGISTER(BASE,OFFSET)   \
    *((volatile unsigned int *)(BASE + OFFSET))


#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif//_REG_COMP_HEADER_FILE_
```

Functional Description

The sample custom component is a WISHBONE slave component containing the three registers shown in Table 16 on page 99. These three registers are general-purpose read/write registers. The lowest byte of register reg_00 is made available as external pins of the component.

The port interface of the custom component in this example is shown diagrammatically in Figure 50. The Verilog source code for this component is shown in “Verilog RTL Implementation” on page 92.

Figure 50: Component's Port Diagram

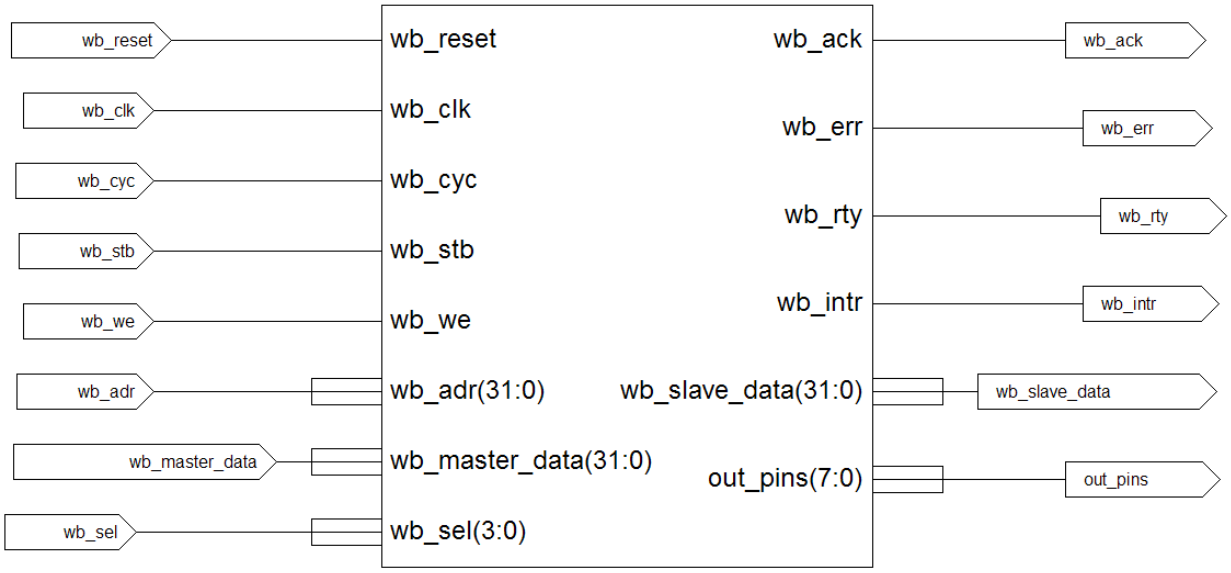


Table 15 lists the input and output signals for the example component.

Table 15: Input/Output Signals in the Example Custom Component

Port Name	Direction	Width (in Bits)	Description
wb_reset	Input	1	WISHBONE reset signal
wb_clk	Input	1	WISHBONE clock signal
wb_cyc	Input	1	WISHBONE cycle qualifier signal
wb_stb	Input	1	WISHBONE strobe signal
wb_we	Input	1	WISHBONE write-enable signal
wb_adr	Input	32	WISHBONE address
wb_master_data	Input	32	WISHBONE data from master
wb_sel	Input	4	WISHBONE byte-select signal
wb_ack	Output	1	WISHBONE ack signal
wb_err	Output	1	WISHBONE error signal

Table 15: Input/Output Signals in the Example Custom Component

wb_rty	Output	1	WISHBONE retry signal
wb_intr	Output	1	Interrupt line to the processor
wb_slave_data	Output	32	WISHBONE data from slave
out_pins	Output	8	External pins; contains value of the lowest byte of reg_00

The example custom component includes the three 32-bit registers shown in Table 16.

Table 16: Registers in the Example Custom Component

Byte Offset	Register Name	Reset Value	Description
0x00	reg_00	0x00000000	General read/write register
0x04	reg_04	CLK_MHZ	General read/write register; power-up value is set to the clock frequency specified as an RTL parameter on instantiation.
0x08	reg_08		General read/write register; power-up value is set to the constant specified as an RTL parameter on instantiation.

Software Support

For illustrative purposes, the example component's initialization function must set the reg_00 register to the value that was used to initialize the reg_08 register in the RTL. At run time, you can read this value from the reg_00 register and compare it to the RTL-initialized value in the reg_08 register. Additionally, you can also read the platform frequency for which the platform was configured in the reg_04 register.

The initialization function must be able to initialize all the instances in a platform. This initialization routine, init_reg_device, is listed in the Reg_Dev.c source file. It relies on the presence of the data structure shown in Figure 51:

Figure 51: Data Structure for Initialization

```
typedef struct st_reg_device {
    unsigned int   reg_08_value;
    unsigned int   b_addr;
} reg_device;
```

This data structure must be initialized according to the instance's configuration in MSB. The data structure contains a member, b_addr, that corresponds to the component's base address, which is assigned by MSB. It also contains a member, reg_08_*value*, which must contain the 32-bit value used for initializing the reg_08 register in the RTL in MSB. This example illustrates how to specify this data structure so that the SPE managed-build process initializes and instantiates the data structure according to the instances in the platform.

While simple, the sample custom component contains enough useful features to illustrate the key steps needed to import it into MSB:

- WISHBONE slave interface
- External pins
- INterrupt signal
- RTL parameter initialization
- Software support

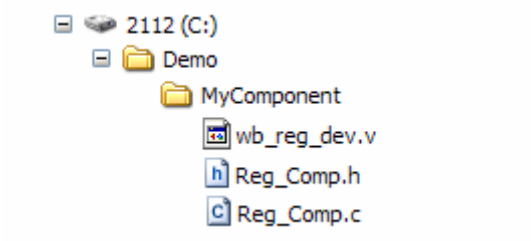
Adding the Custom Component

In this section, you will add the example custom component to the MSB graphical user interface.

It is assumed that the sources are located in the C:\Demo\MyComponent\ folder, as shown in Figure 52.

The intended destination repository for the custom component is C:\Demo\MSBComponents.

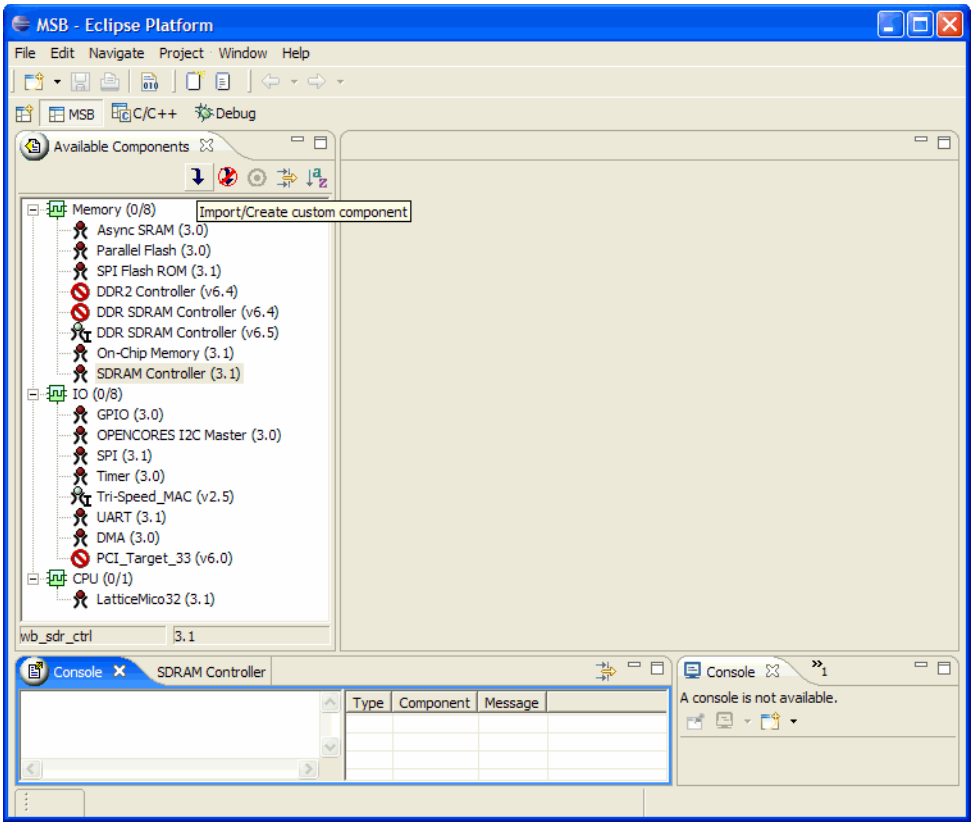
Figure 52: Source Directory



To add a custom component to MSB:

1. Select the Import/Create Custom Component button, as shown in Figure 53, to open the Import/Create Custom Component graphical user interface.

Figure 53: Opening the Import/Create Custom Component Graphical User Interface

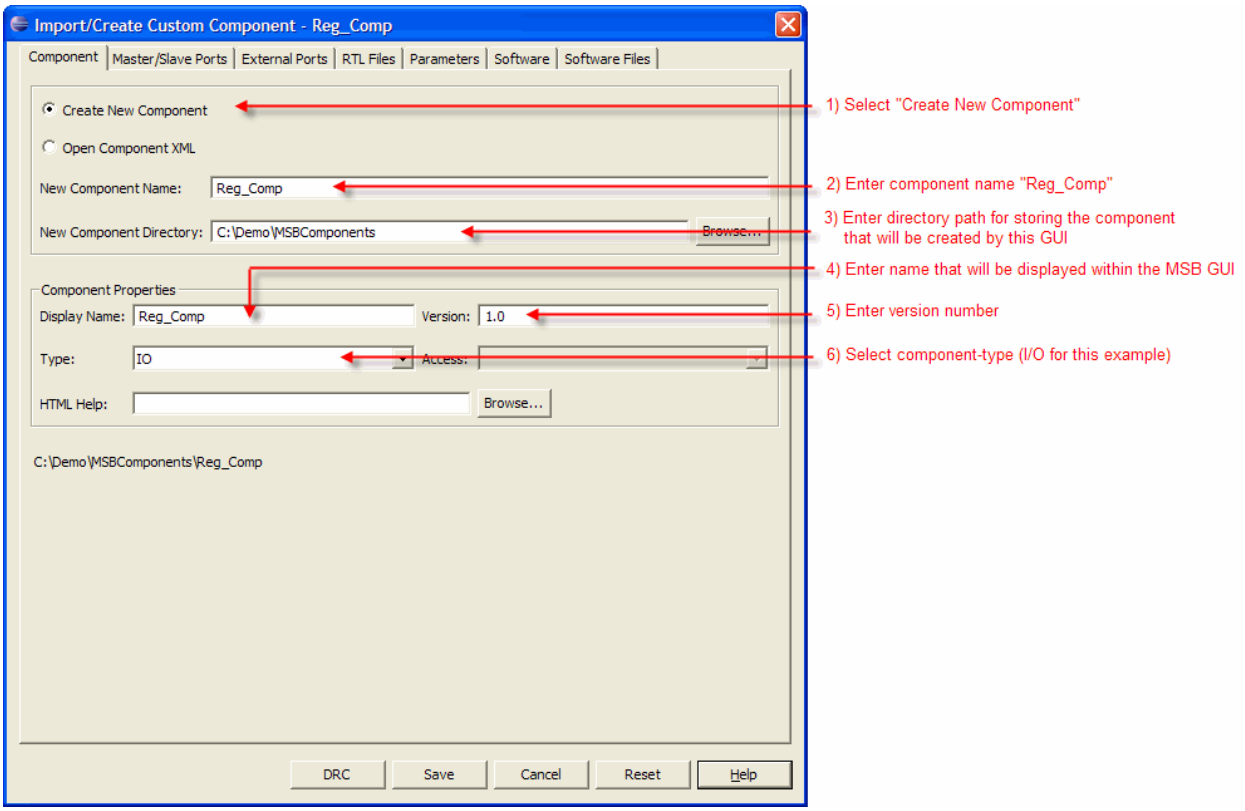


2. Enter the component information, as shown in Figure 54.

Warning

The display name should not be the same as that of any of the design RTL files. It also cannot be the same as the name of the top module file.

Figure 54: Specifying the Component's General Information



3. Specify the WISHBONE slave port signals for the component, as shown in Figure 55.
4. Specify the component's WISHBONE clock signal, as shown in Figure 56.
5. Specify the component's WISHBONE reset signal, as shown in Figure 57.
6. Optionally, specify the component's interrupt signal information, as shown in Figure 58. If your component does not have an interrupt line, you do not need to perform this step. Since the example component has an interrupt line, you must specify its properties.
7. Specify the component's external ports, as shown in Figure 59.
8. Specify the component's RTL files, as shown in Figure 60.
9. Specify the component's RTL parameters, as shown in Figure 61.

Figure 55: Specifying the WISHBONE Slave Port Signals for the Component

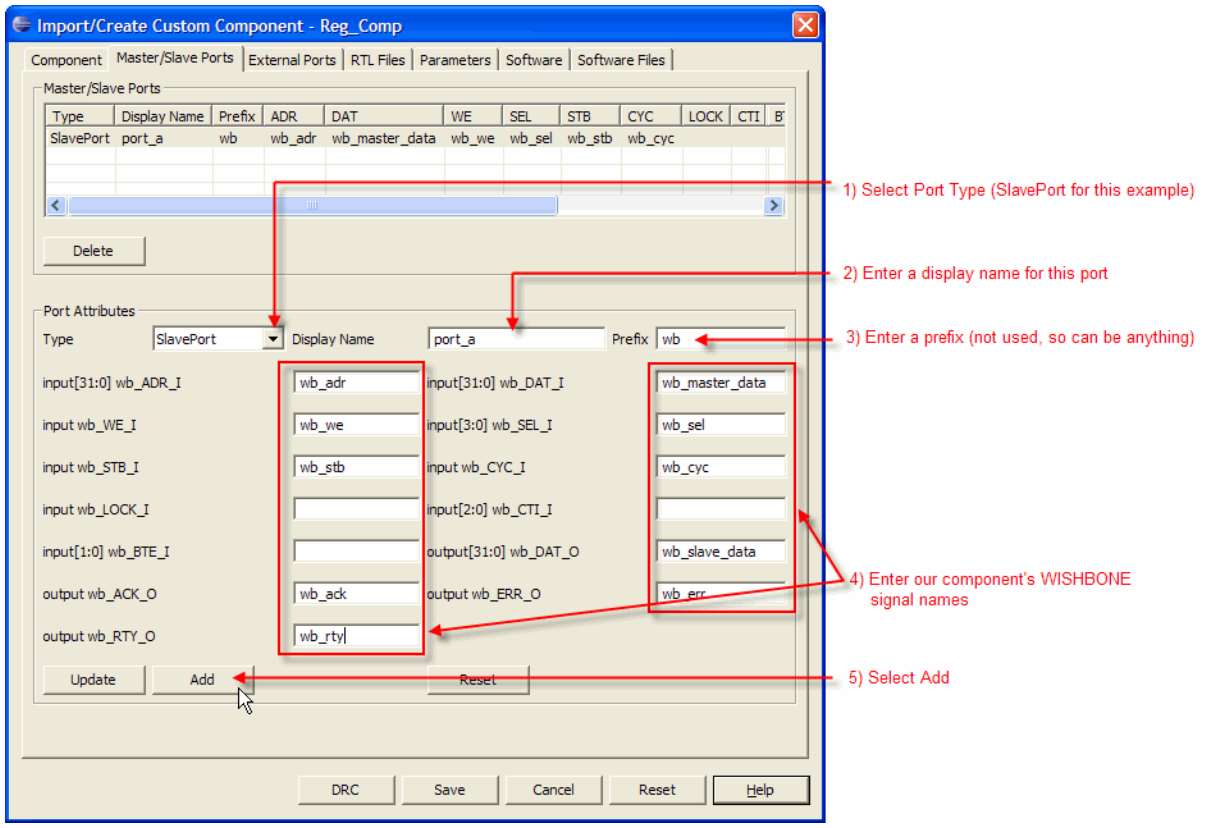


Figure 56: Specifying the WISHBONE Clock Signal for the Component

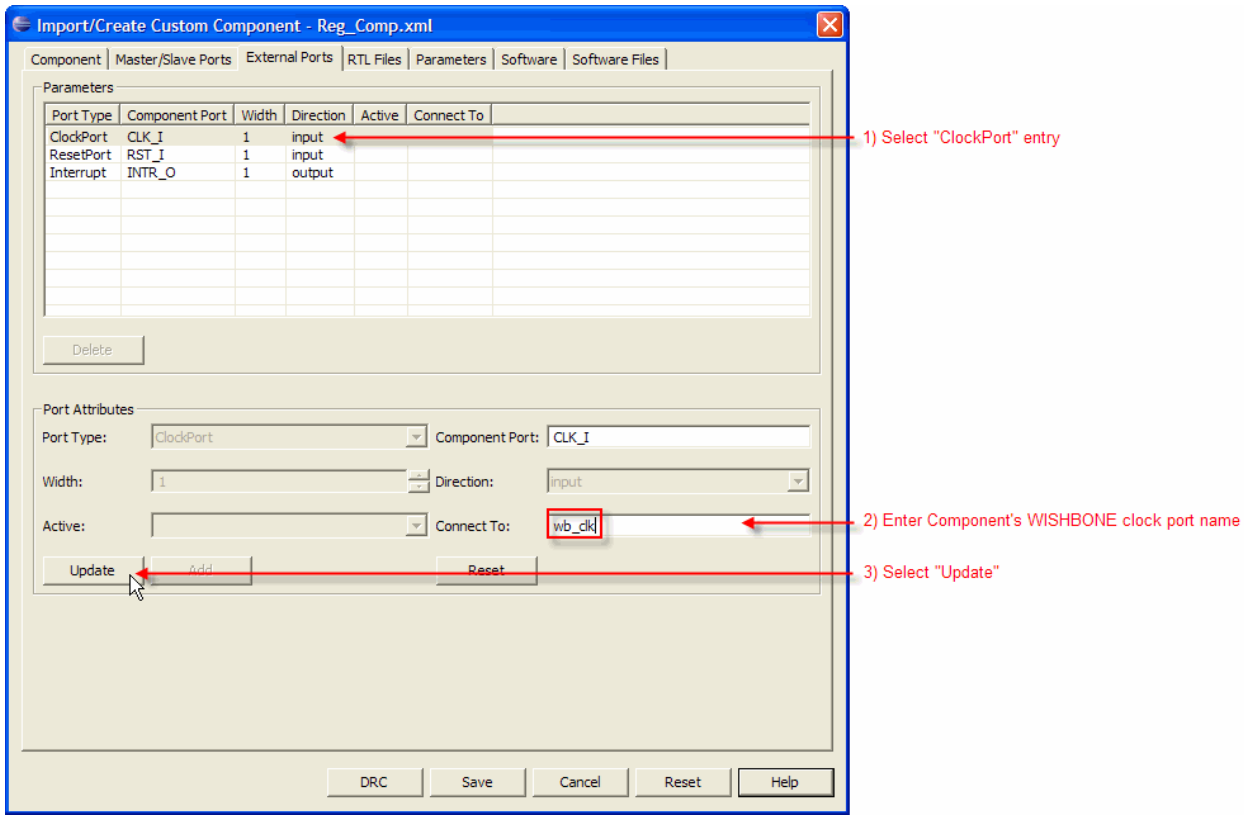


Figure 57: Specifying the WISHBONE Reset Signal for the Component

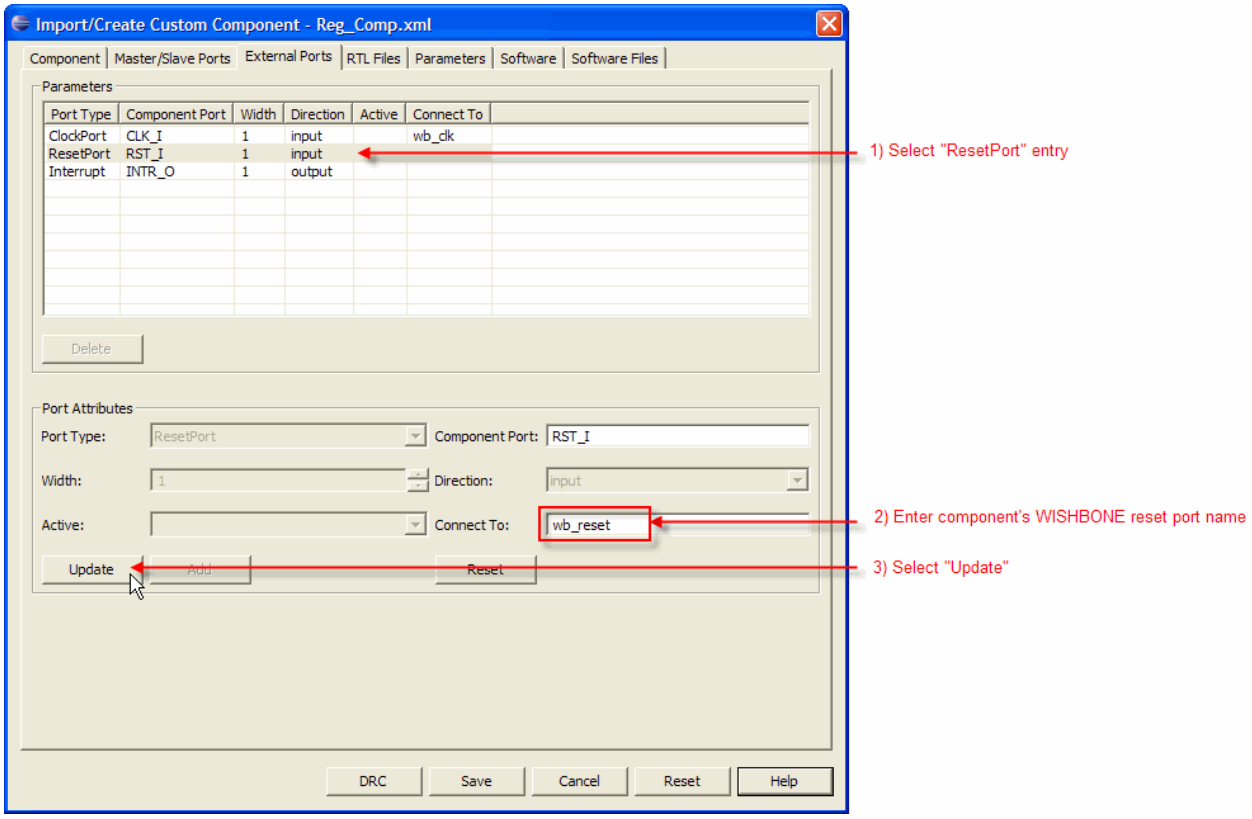


Figure 58: Specifying the Interrupt Signal for the Component

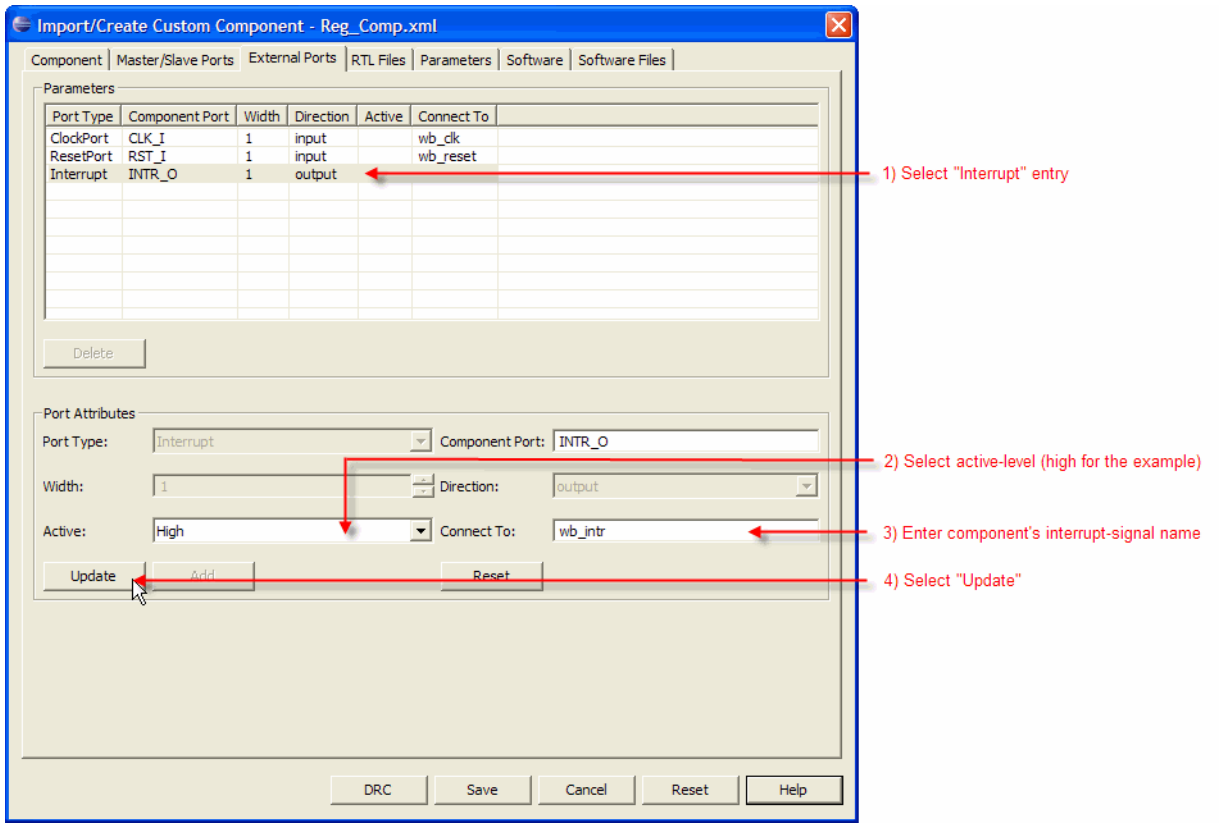


Figure 59: Specifying the External Port for the Component

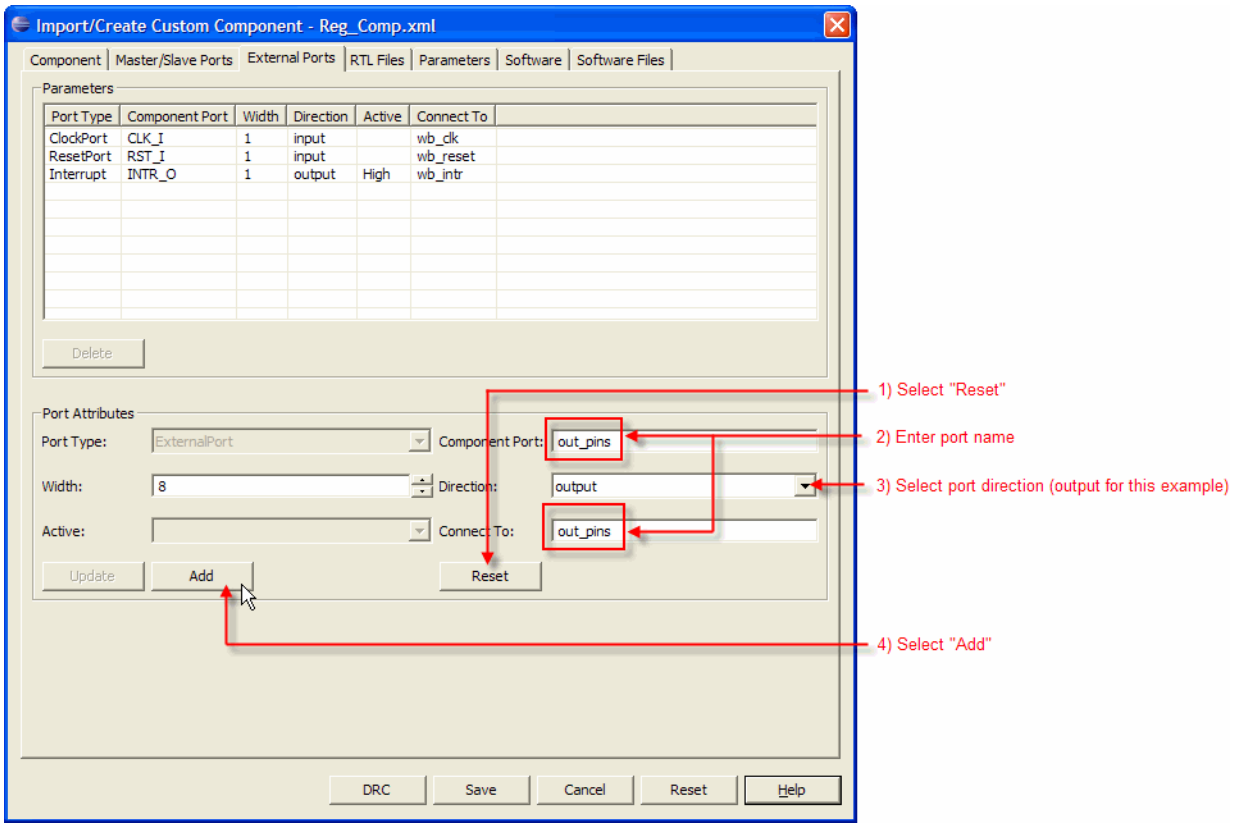


Figure 60: Specifying the RTL Files for the Component

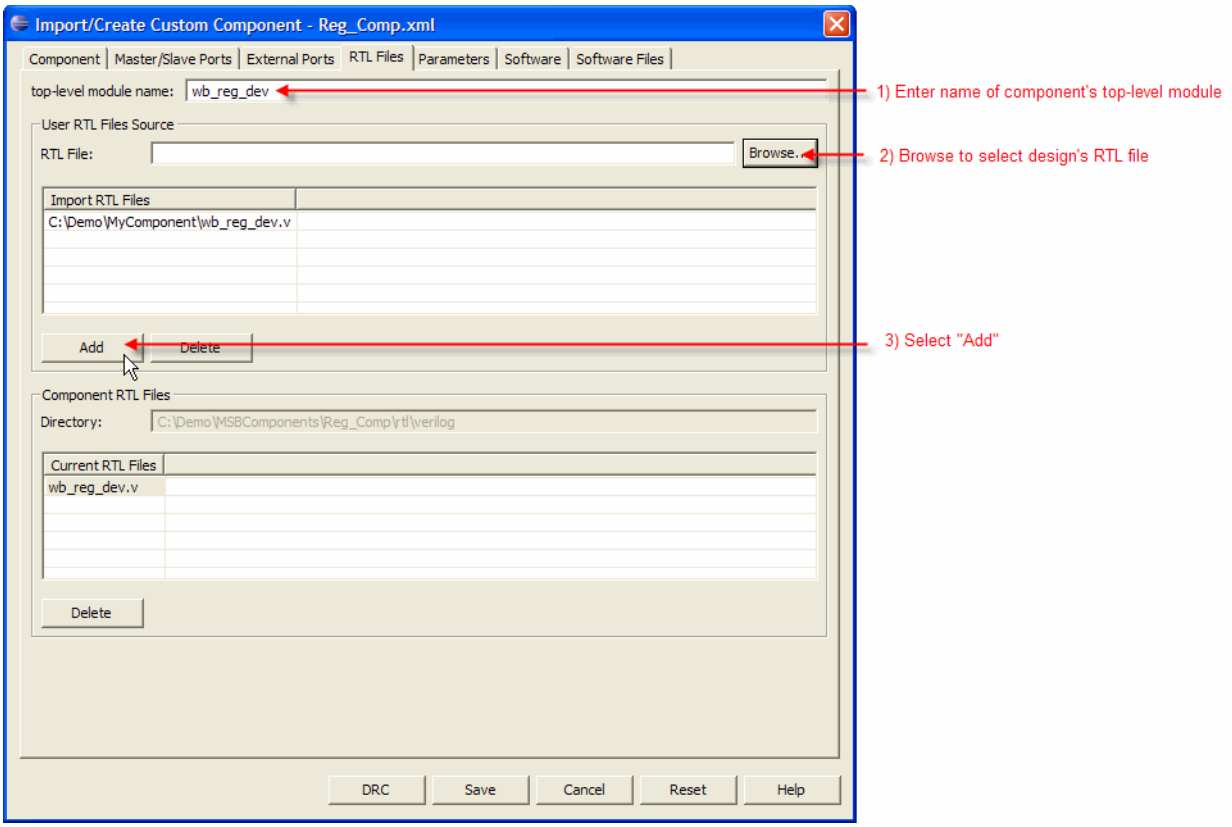


Figure 61 shows the steps required for adding a GUI widget for configuring the reg_08 register's value when you instantiate the custom component in a platform.

Note

You might need to adjust the default size for your component.

Figure 61: Adding a Configuration Widget for the reg_08 Register

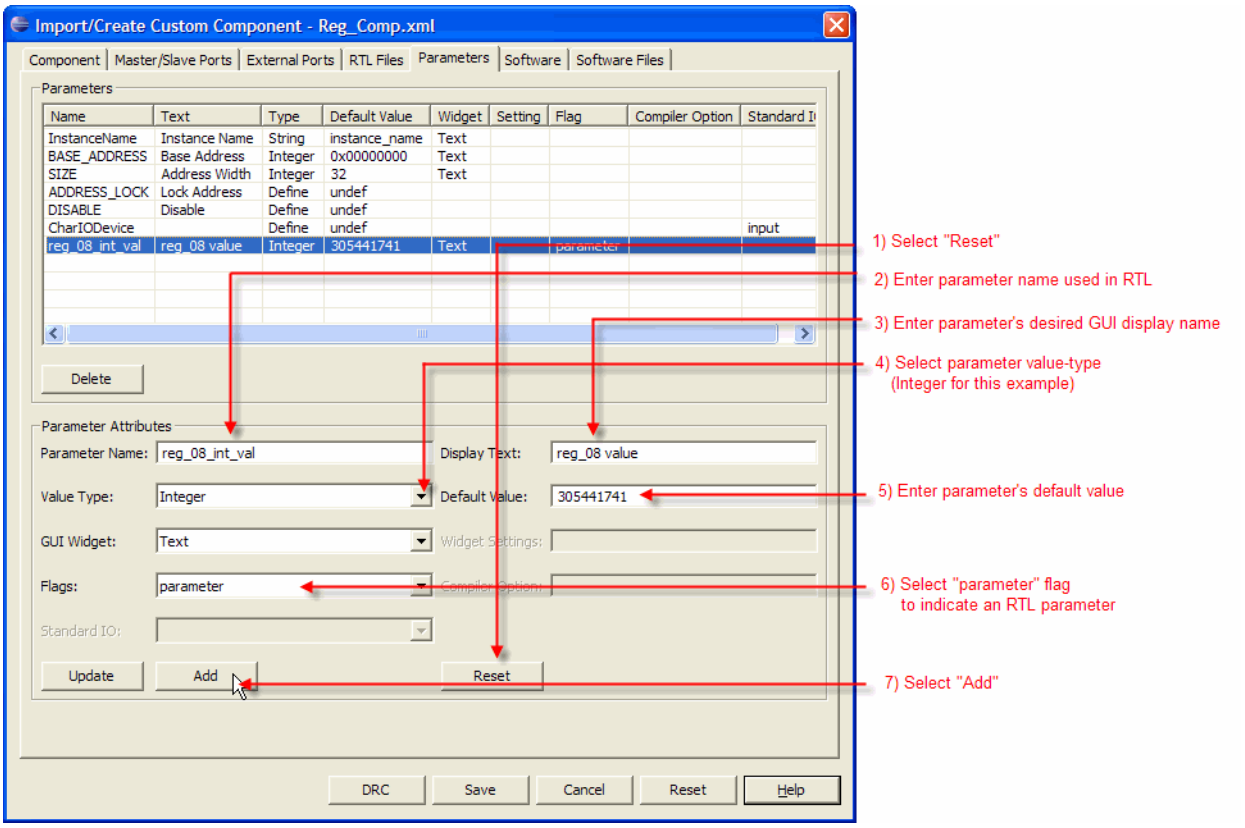
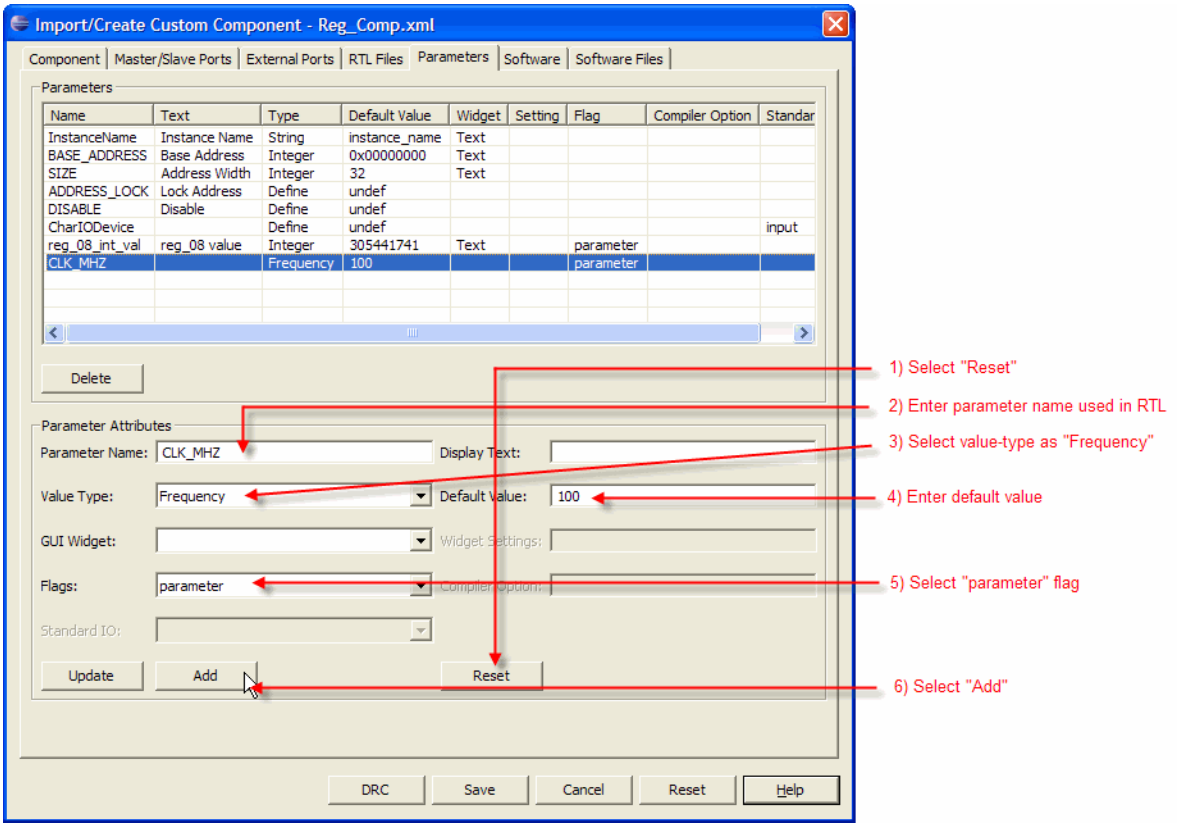


Figure 62 shows the steps required for adding the CLK_MHZ parameter for the component that will receive the platform's WISHBONE clock-frequency from MSB when instantiated a platform. This parameter will not be visible for configuration.

Figure 62: Specifying the Platform's WISHBONE Clock Frequency RTL Parameter



When importing a new component, you should always check the SIZE parameter. The default value for the SIZE parameter determines the default address-decode space for the component. Although you can change it when you instantiate it in the platform, it is always a good idea to make sure that the default value is sufficient to cover the entire addressable space (for example, the space for registers, memory, or I/Os) that is provided for the component being imported. The custom component example has three registers—that is, a total decode space of 12 bytes—so the default value of 32 for the SIZE parameter is adequate.

The example custom component requires a data structure like that shown in Figure 63.

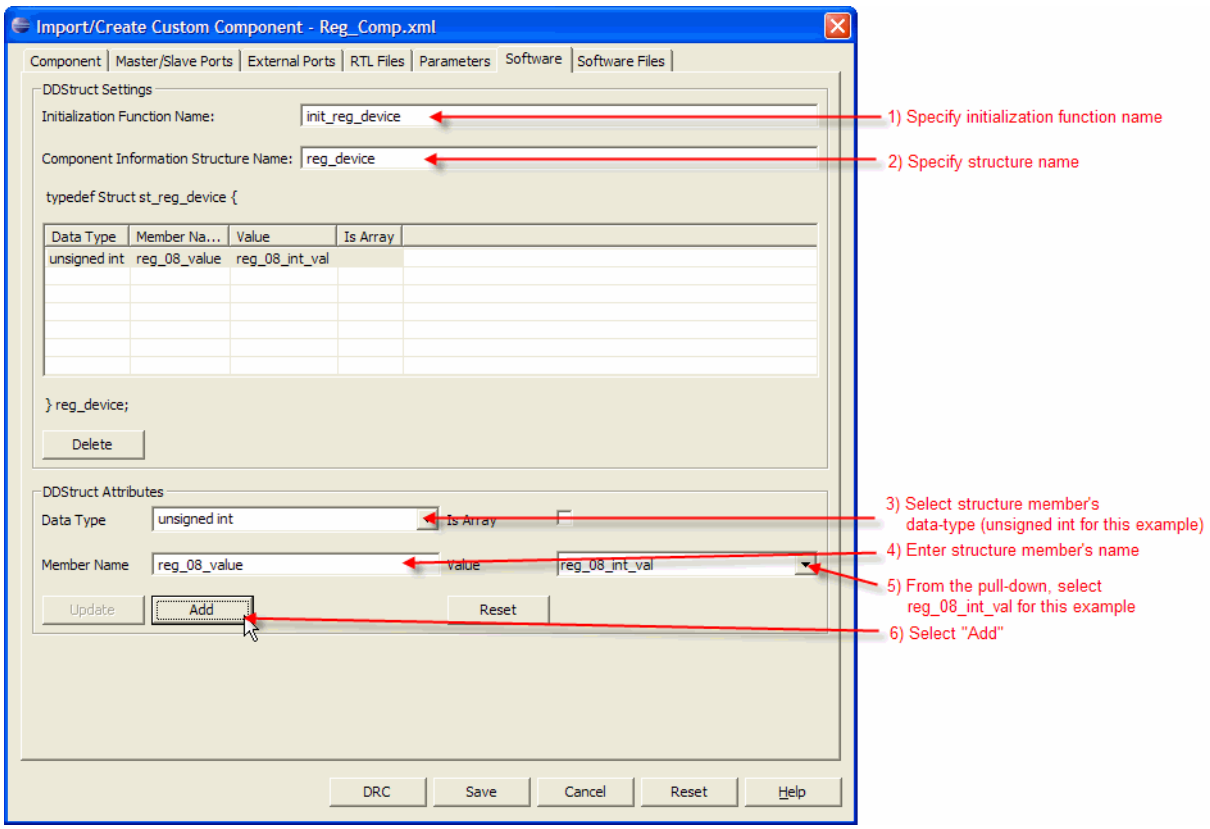
Figure 63: Data Structure Required for Creating Custom Component

```
typedef struct st_reg_device {
    unsigned int   reg_08_value;
    unsigned int   b_addr;
} reg_device;
```

In addition, the members of this structure must be initialized to the appropriate values, which are provided when you generate the platform.

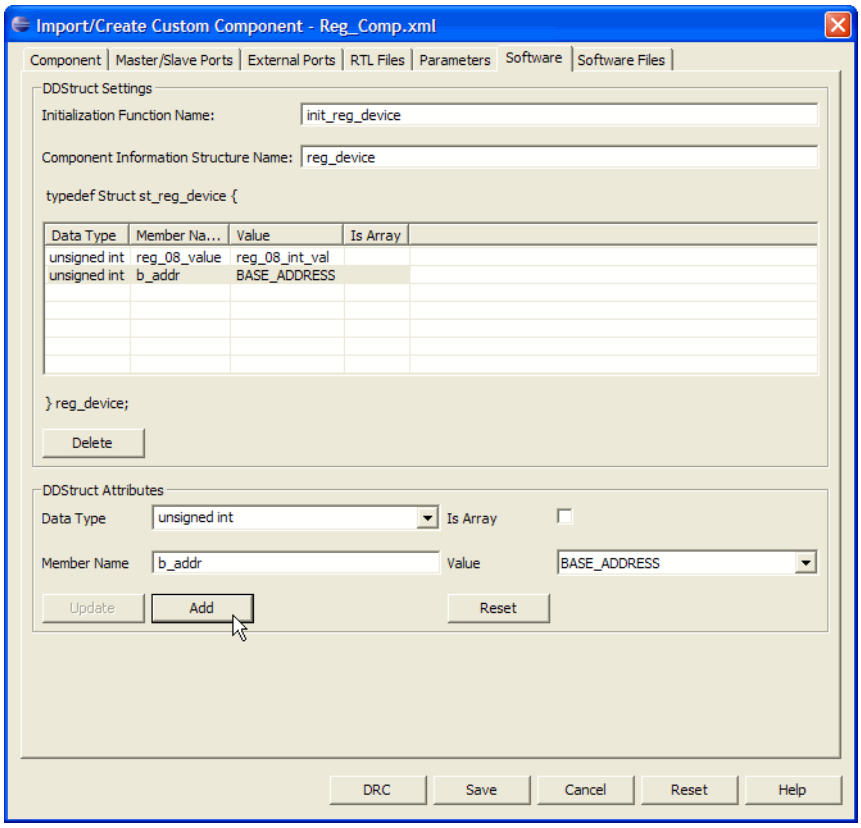
10. Specify the component's data structure and initialization function for software support, as shown in Figure 64.

Figure 64: Specifying Data Structure and Initialization Function



11. Repeat the steps shown in Figure 64 to do the following:
    - Add b_addr as an "unsigned int" member that should contain the component's base address parameter, BASE_ADDRESS, as shown in Figure 65.
    - Add name as a "const char *" member that should contain the component's name parameter, InstanceName. This member helps while registering the custom component with system software.
12. Add the C source file that should be compiled as part of the platform library, as shown in Figure 66.
13. Add the device driver's header file (.h), which is a standard header file that can be included in a user application, as shown in Figure 67.
14. Click **DRC** to check for any errors.
15. Click **Save** to save the custom component.

Figure 65: Specifying Second Data Structure



Output

After you perform the steps in the “Adding the Custom Component” on page 100, the component now appears in the MSB graphical user interface, as shown in Figure 68.

When you double-click on this component, a configuration dialog box opens, as shown in Figure 69, so that you can configure it when instantiating it in a platform.

Figure 70 shows the directory structure and the contents of the directories created by the MSB graphical user interface.

The directory structure shown in Figure 70 is created automatically by the Import/Create Custom Component dialog box. The source files are copied from the source folder into the directory structure. If you want to modify the RTL once this component is created—for example, to fix a bug—you must modify the copied files, not the original source files.

Figure 66: Adding the C Source File

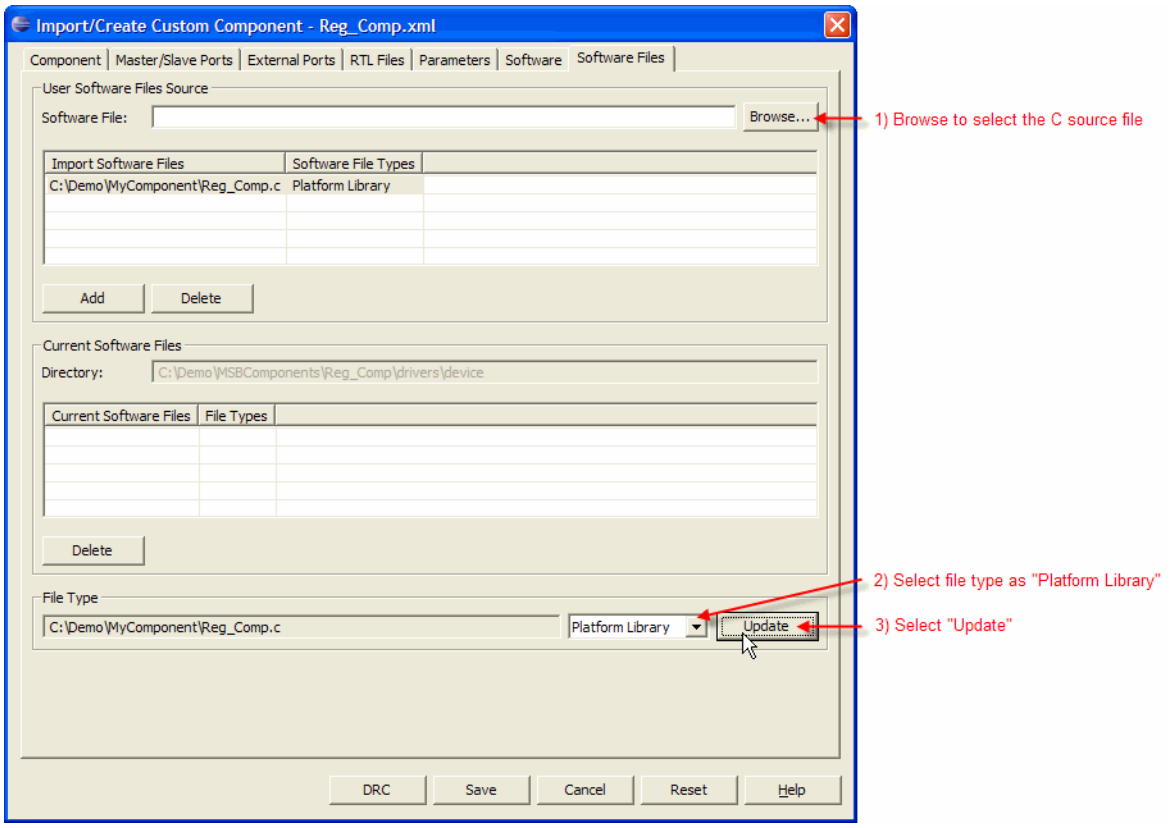


Figure 67: Adding the Device Driver Header (.h) File

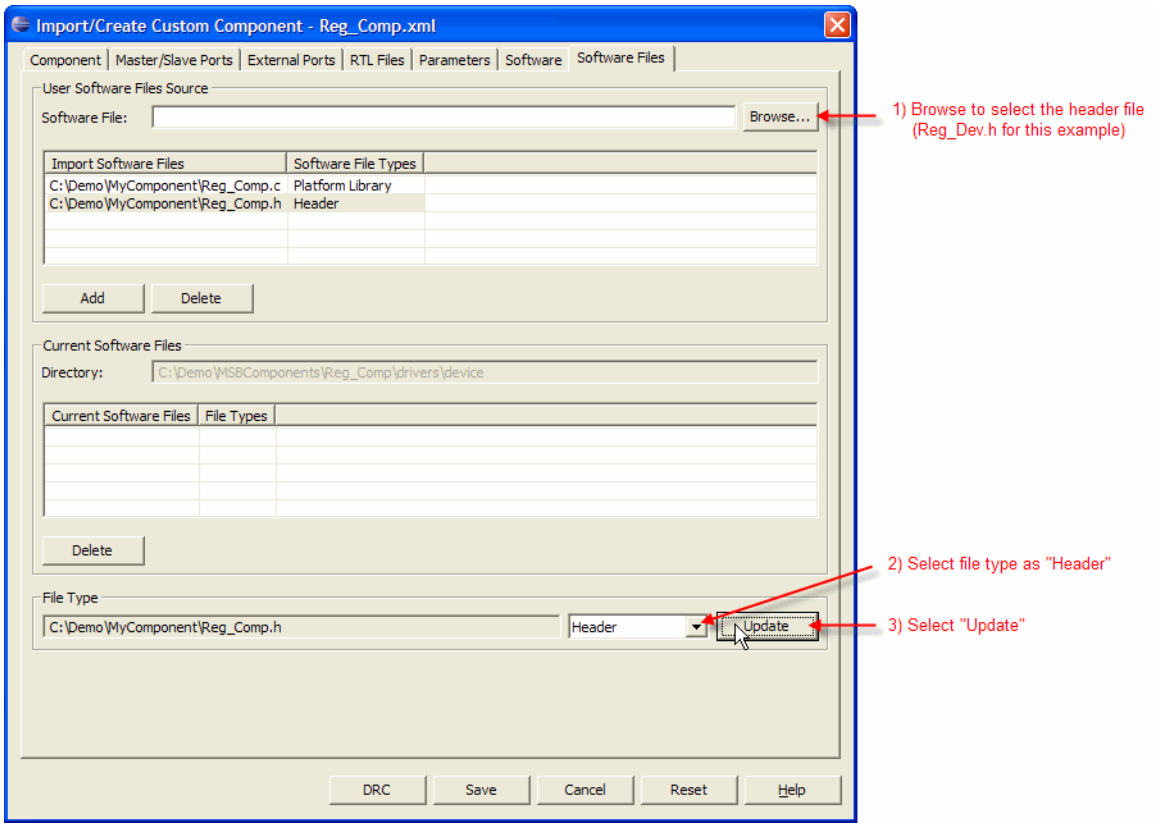


Figure 68: Custom Component in MSB Available Components View

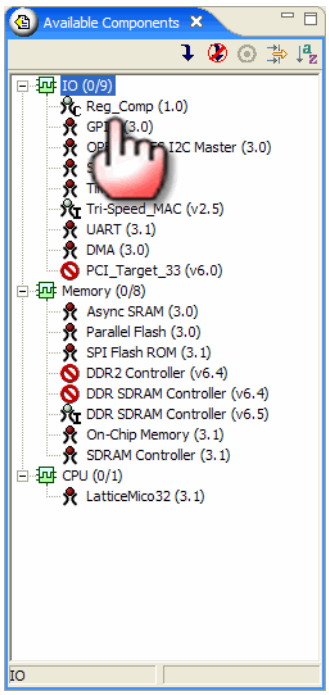


Figure 69: Add Reg_Comp Dialog Box

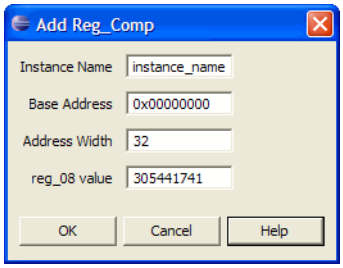
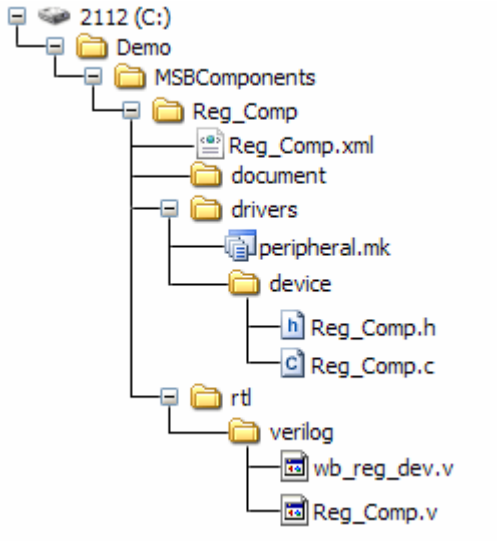


Figure 70: Directories Created by the MSB Graphical User Interface



Glossary

Following are the terms and concepts that you should understand to use this guide effectively.

application build An application build is the files that the managed build process outputs and places in the application build output folder, for example, the application executable, application build makefiles, application object files, and necessary platform library files.

application build makefiles Application build makefiles enable the building of the application.

application executable The application executable is a result of linking the application and the platform library object file. The file is an executable in ELF format that can be downloaded or executed using the GNU GDB debugger.

application object files Application object files are user source object files that have been compiled and assembled from their source C files.

breakpoints Breakpoints are a combination of signal states that are used to indicate when simulation should stop. Breakpoints enable you to stop the program at certain points to examine the current state and the test environment to determine whether the program functions as expected.

C/C++ SPE C/C++SPE is an abbreviation for the C/C++ Software Project Environment, which is an integrated development environment based on Eclipse for developing, debugging, and deploying C/C++ applications. The C/C++ SPE uses the bundled GNU C/C++ tool chain (compiler, assembler, linker, debugger, and other utilities such as objdump) customized for the LatticeMico32 process. It uses the same graphical user interface as MSB.

component information structure declaration A component information structure declaration is specified as part of the .xml file and is copied into .msb file by MSB. Each component in the platform is represented in the .msb file.

The component's information in the .msb file includes the details about the component's source files that will need to be included in the build process. The information is then extracted from the .msb file by the build process and put into the DDStructs.h file. Each unique component must have its own unique component information structure defined within its component description file.

component instance declaration For those component instances that have a corresponding information structure, this header file declares presence of an instantiated structure. Originates in the Component Description (.xml) file.

components Components are parts of the microprocessor system architecture, for example, a CPU and peripherals are referred to generically as components. Also see platform.

CSR CSR is an abbreviation for a control and status register, which is a register in most CPUs that stores additional information about the results of machine instructions, for example, comparisons. It usually consists of several independent flags, such as carry, overflow, and zero. The CSR is mainly used to determine the outcome of conditional branch instructions or other forms of conditional execution.

CDT CDT is an abbreviation for C/C++ development tools, which are components, or plug-ins, of the Eclipse development environment on which the LatticeMico System is based.

default linker script The default linker script, named linker.ld, is the default linker script for the particular platform/project combination and can be used as a starting point for creating a custom linker script file.

device driver files Device driver files are the source .c and .h C/C++ files that contain driver code that will be compiled into object files during software build.

debugging Debugging is the process of reading back or probing the states of a configured device to ensure that the device is behaving as expected while in circuit. Specifically, debugging in software is the process of locating and reducing the errors in the source code (the program logic). Debugging in hardware is the process of finding and reducing errors in the circuit design (logical circuits) or in the physical interconnections of the circuits. The difference between running and debugging software is the placement of breakpoints in debugging.

Eclipse Eclipse is an open-source community whose projects are focused on providing an extensible development platform and application frameworks for building software. The LatticeMico System interface is based on the Eclipse environment.

.elf file An .elf file is a file in executable linked format that contains the software application code written in C/C++SPE.

GDB GDB is an abbreviation for GNU GDB debugger, which is a source-level debugger based on the GNU compiler. It is part of the C/C++SPE debugger.

GNU Compiler Collection (GCC) The GNU Compiler Collection (GCC) is a set of programming language compilers produced by the GNU Project. It is free software distributed by the Free Software Foundation (FSF).

HAL HAL is an acronym for hardware abstraction layer, which is the programmer's model of the hardware platform. It enables you to change the platform with minimal impact to your C code.

hardware debugger module The hardware debugger module is a component of C/C++SPE that is used to find problems in the software application. Most times it is simply referred to as the debugger module.

hardware platform See "platform."

IRQ IRQ is an abbreviation for interrupt request, which is the means by which a hardware component requests computing time from the CPU. There are 16 IRQ assignments (0-15), each representing a different physical (or virtual) piece of hardware. For example, IRQ0 is reserved for the system timer, while `IRQ1` is reserved for the keyboard. The lower the number, the more critical the function.

JTAG ports JTAG ports are pins on an FPGA or ispXPGA device that can capture data and programming instructions.

makefiles Makefiles contain scripts that define what files the make utility must use to compile and link during the build process. There are many makefiles employed in the LatticeMico System build process. The makefile file is the application build makefile, calling all of the other makefiles that allow the generation and build of the platform library and for eventually generating the final executable image.

MSB MSB is an abbreviation for Mico System Builder, which is an integrated development environment based on Eclipse for choosing peripherals, such as a memory controller and serial interface, to attach to the Lattice Semiconductor 32-bit embedded microprocessor. It also enables you to specify the connectivity between these elements. MSB then enables you to generate a top-level design that includes the processor and the chosen peripherals. It uses the same graphical user interface as C/C++SPE.

.msb file The .msb file is the output XML file output by the MSB tool when working in the MSB perspective. This .msb file is generated or updated when you save your changes in the MSB perspective. This file defines your platform, that is, the CPU and the peripherals in your design and also their interconnectivity.

perspective A perspective is a separate combination of views, menus, commands, and toolbars in a given graphical user interface window that enable you to perform a set of particular, predefined tasks. The LatticeMico System contains three default perspectives: the MSB perspective, the C/C++ perspective, and the Debug perspective.

platform A platform (also called a hardware platform) is the embedded microprocessor in an SoC (system on a chip) design. A platform comprises the CPU and peripheral components and the interconnectivity that allows these components to work together to successfully execute processor instructions.

platform library The platform library is a set of files that contain subroutine code that references the application files that are necessary for linking during the build process.

platform library build The platform library build is an integral part of the managed build process. Another is the application build. The platform library files contain code that is necessary to the linking during the build process. The platform library build also outputs a platform library archive (<*platform*>.a) file that is referenced by the application build. It allows you to override any default software implementation.

platform library archive (.a) file The platform library archive (<*platform*>.a) file is automatically generated during a platform library build. It is used when linking the application executable to resolve platform functions used by the application and is derived from the platform library object files.

platform library object (.o) file The platform library object (.o) file is a compiled output of the library source files and is input for creating platform library archive files.

platform settings file The platform settings file is the user.pref file that is generated during the build process contains platform information for the platform used by the current project.

project A project is the software application code written in C++ SPE. Projects are contained within your workspace.

project workspace See “workspace.”

resources or resource files Resources are the projects, folders, and files that exist in the Workbench. The navigation views provide a hierarchical view of resources and allows you to open them for editing. Other tools may display and handle these resources differently.

running Running is the process of executing a software program.

software application The software application is the code that runs on the 32-bit Mico processor to control the peripherals, the bus, and the memories. The application is written in a high-level language such as C++.

source files In this document, source files generically refer to source .c and header .h files written in C/C++ programming language.

source folders Source folders are the folders you may have on your system or in the project folder that contain input for a project. Input might include source files and resource files to help enhance or to initially establish a LatticeMico32 project.

UART UART is an acronym for universal asynchronous receiver/transmitter, which is a computer component that handles asynchronous serial communication. Every computer contains a UART to manage the serial ports, and some internal modems have their own UART.

watchpoint A watchpoint is a special breakpoint that stops the execution of an application whenever the value of a given expression changes, without specifying where this may happen. A watchpoint halts program execution, even if the new value being written is the same as the old value of the field.

workspace A workspace contains all of your LatticeMico System projects, files, and folders and stores everything in a “workspace” folder. Basically a workspace represents everything you do in the LatticeMico System software, what is available, how you view it, and what options are available to you through the different perspectives based on your settings. This is a basic Eclipse-based software feature.

XML XML is an abbreviation for Extensible Markup Language, which is a general-purpose markup language used to create special-purpose markup languages for use on the Worldwide Web.

.xml file (1) The .xml file contains information about the parent project and its settings, as well as information on the platform referenced by the parent project. (2) The <*comp_name*>.xml files contain code declarations referred to as component instance definitions that define the structure of each component. These files reside in the <*install_dir*>/components folder. On build generation, this information is copied into the .msb file by MSB.

Index