LatticeMico8 Developer User Guide

May 2014

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

LatticeMico System Overview

This software developer's guide describes the flow that is used to create and deploy software application code for the LatticeMico8 microcontroller. In addition, it familiarizes the reader with the LatticeMico8 run time environment, the managed build environment, and it's associated directory structure. This guide is targeted to software programmers who are interested in learning the fundamentals of programming the embedded soft core microcontroller. For a list of related documents on the LatticeMico8 microcontroller, refer to '"Related Documentation" on page 4.

LatticeMico System Design Flow

This section lists the major steps involved in designing for the LatticeMico8 microcontroller. In addition to running the FPGA flow in Diamond, you will use the integrated LatticeMico System software to build both hardware and software features of the embedded soft core microcontroller.

The LatticeMico System software is composed of three bundled applications:

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

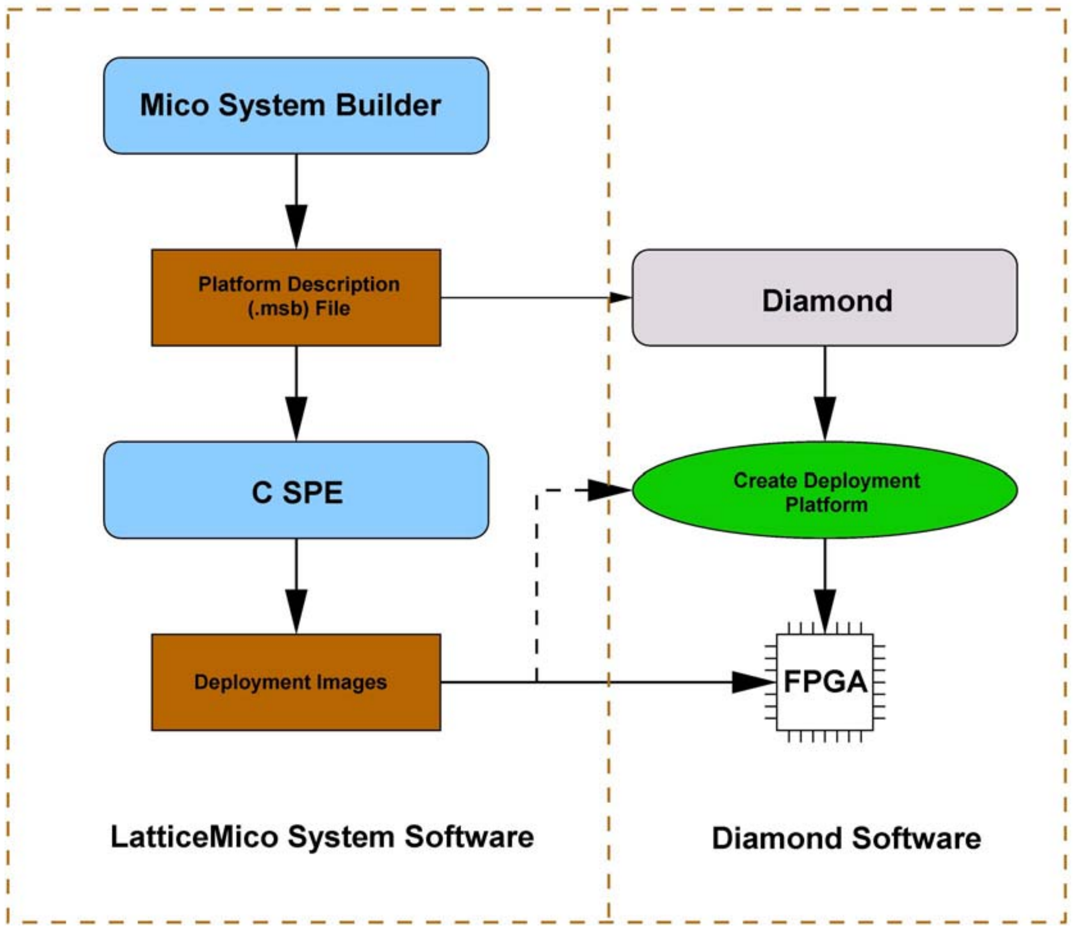
These applications work in the background through the user interface and can be accessed through different "perspectives" in the LatticeMico System software. Perspectives are a prearranged and predefined set of user functions that can be accessed within the software user interface. You toggle different perspectives on and off by clicking on perspective tabs. Perspectives are described in more detail in "LatticeMico System Perspectives" on page 8.

MSB is used by hardware designers to create the microcontroller platform for both hardware and software development. A platform generically refers to the hardware microcontroller configuration, the CPU, its peripherals, and how these components are interconnected. This functionality in the LatticeMico8 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. There is no debugger support for LatticeMico8.

Figure 1 shows the interaction of the three LatticeMico8 System applications with Lattice Diamond in the microcontroller development design flow.

Figure 1: LatticeMico8 System Development Software Tool Flow


Mico System Builder
Platform Description (.msb) File
Diamond
C SPE
Create Deployment Platform
Deployment Images
FPGA
LatticeMico System Software
Diamond Software

As noted earlier, you can learn more about perspectives in “LatticeMico System Perspectives” on page 8. In addition, the *LatticeMico8 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 LatticeMico8 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:

- MachX02
- MachX03L
- Platform Manager 2

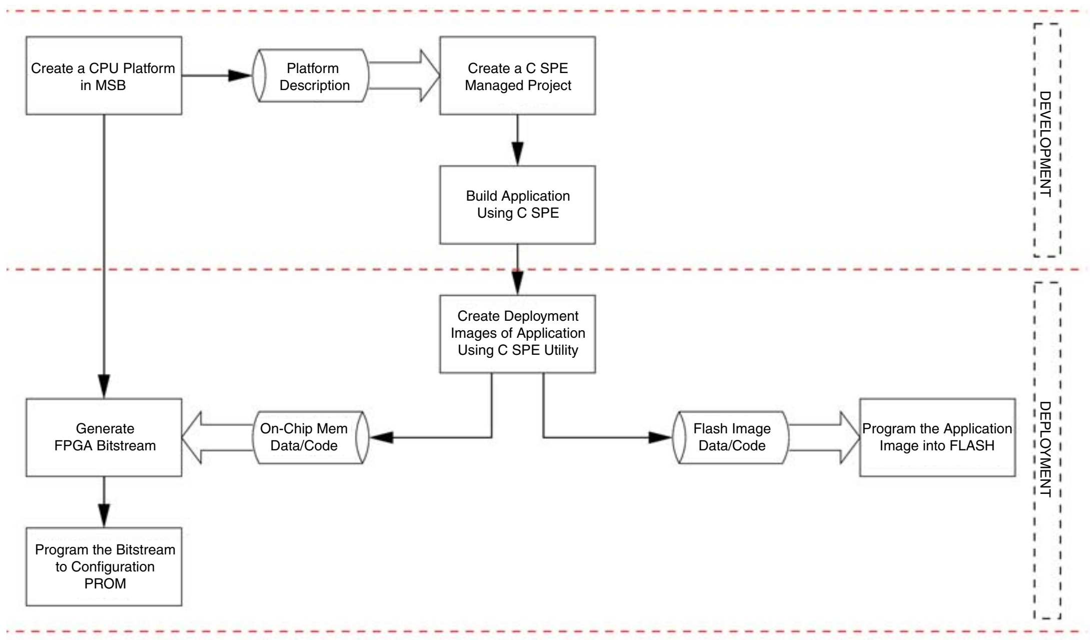
Design Flow Steps

The major steps involved in designing a LatticeMico8 soft-core microcontroller are the following:

1. Create a project in Diamond that targets the desired device family.
2. Use the Lattice Mico System Builder (MSB) in the LatticeMico8 System software to create and develop a microcontroller platform. You access this in the MSB perspective. Creating a platform involves generating an .msb file, selecting component peripherals, and connecting them to the LatticeMico8 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 perspective to write the C user application software, build the application, and created deployment images for on-chip memory or non-volatile memory.
6. In Diamond, download the executable code to either on-chip memory or non-volatile memory. In case of non-volatile memory, LatticeMico8 contains a hardware boot loader that can execute from non-volatile memory or optionally load the application from non-volatile memory to on-chip memory.
7. Repeat steps 3 through 6 for any new application development or modification to the platform in step 2.

Figure 2 shows the LatticeMico8 System design flow.

Figure 2: LatticeMico8 System Design Flow


Create a CPU Platform in MSB
Platform Description
Create a C SPE Managed Project
Build Application Using C SPE
Create Deployment Images of Application Using C SPE Utility
Generate FPGA Bitstream
On-Chip Mem Data/Code
Flash Image Data/Code
Program the Application Image into FLASH
Program the Bitstream to Configuration PROM
DEVELOPMENT
DEPLOYMENT

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:

- *LatticeMico8 Processor Reference Manual*, which contains information on the architecture of the LatticeMico8 microprocessor chip, including configuration options, pipeline architecture, register architecture, and details about the instruction set
- *MachXO2 Development Board User Guide*, which describes the features and functionality of the MachXO2 development board. This board is designed as a hardware platform for design and development with the LatticeMico8 microprocessor 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 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 SPI Flash*, which describes the features and functionality of the LatticeMico serial peripheral interface (SPI) flash memory controller
- *LatticeMico UART*, which describes the features and functionality of the LatticeMico Universal asynchronous receiver-transmitter (UART)
- *Lattice Diamond Installation Notice*, which explains how to install the LatticeMico System software for the current release
- *Lattice MachXO2 FPGA Family Handbook*, which is a collection of the data sheets and application notes on Lattice MachXO2 devices
- *Lattice MachXO2 Family Data Sheet*

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 microcontroller platform
- C/C++ Software Project Environment (C/C++ SPE), which is used to create the software application code (written in C, Assembly, or C with inlined-assembly) that drives the microcontroller platform.
- Deployment

The LatticeMico 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 microcontroller or your hardware platform. C/C++ SPE enables you to develop and compile your code in a managed and well-structured build environment. 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

System requirements for installing Lattice Diamond, LatticeMico System, and Stand-Alone Programmer, are included in the *Lattice Diamond Installation Notice*, available on the Lattice Web site for Windows and Linux.

Refer to the “Installing LatticeMico Development Tools” chapter for information about LatticeMico System’s system requirements and installation.

Refer to the “Installing Stand-Alone Programmer” section for information about Stand-Alone iProgrammer’s installation.

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 **Programs > Lattice Diamond > Accessories > LatticeMico System**.

The LatticeMico System interface initially opens with the MSB perspective active by default. After that, the software opens to the last opened perspective.

LatticeMico System Perspectives

Note

The Debugger perspective is visible with LatticeMico8, but Debugger is not supported for LatticeMico8 microcontroller.

Before you begin learning about the basic tasks that you can perform in the LatticeMico System software to design hardware platforms and software applications for the LatticeMico8 microcontroller, 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

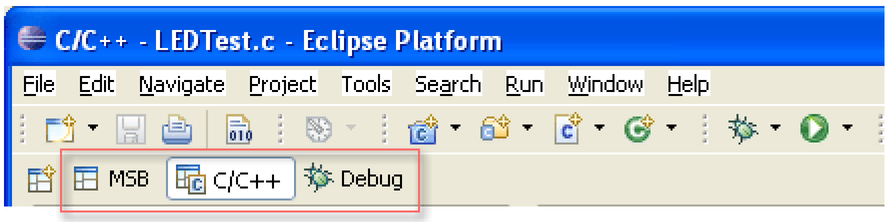
  For complete information about using the MSB perspective to configure the microcontroller hardware platform and peripherals, refer to the *LatticeMico System Hardware Developer User Guide*.

- C/C++ SPE perspective, shown on Figure 13 on page 37
- Debug perspective.

Within the Eclipse framework, the three functional tools appear as different user interfaces or "perspectives" 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. Figure 3 also shows the activated C/C++ perspective. The current active perspective is displayed in the upper left of the window's title bar.

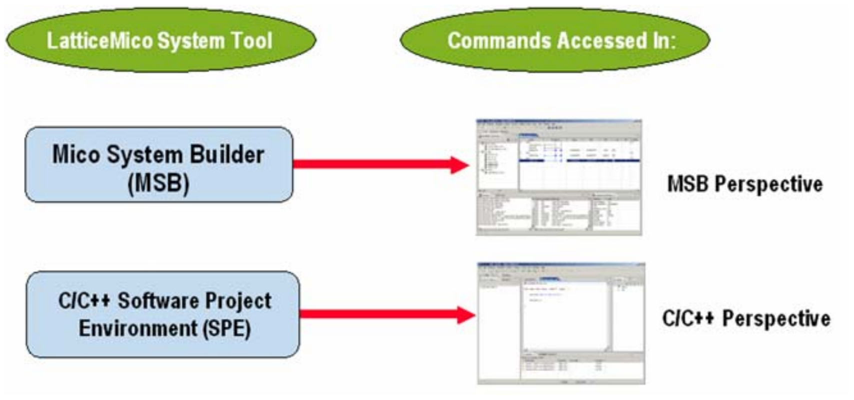
Figure 3: Perspective Activation Buttons


C/C++ - LEDTest.c - Eclipse Platform
File Edit Navigate Project Tools Search Run Window Help
MSB
C/C++
Debug

Only two perspectives - the MSB and the C/C++ SPE - are used for designing hardware platforms and software applications for the LatticeMico8 microcontroller. The Debug perspective, though visible, is not used. The MSB and C/C++ SPE perspectives include tool functions that the developer can 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.

Figure 4: Tool Functions Accessed in Perspectives



Note

Particular views and options within a given perspective are described in more detail throughout this chapter as they are encountered in the design flow. More information on the graphical user interface, views, windows, dialog boxes, and so forth are described in more detail in the LatticeMico 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 LatticeMico8 Platform

Before you create your microcontroller 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 microcontroller. 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 LatticeMico8 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 LatticeMico 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:

     <install_path>/diamond/*<version_number>*/bin/lin/diamond &
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 LatticeMico 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 18. To keep the platform within the Diamond project you just created, do not change the directory location.

4. Add the LatticeMico8 microcontroller to the platform and any desired memory and peripheral components, as described in “Adding Microcontroller and Peripherals to Your Platform” on page 20.

5. From the Available Components window, double-click the desired IP core component—for example, MachXO2 EFB—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 or .rtl file—for example, efb.v—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 or RTL File text box is now populated with the location of the .ngo or .v 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 LatticeMico8 Platform in MSB" on page 15 to connect master and slave ports, assign addresses and interrupt priorities, and generate the platform.

Creating the LatticeMico8 Platform in MSB

After you have created a new project in Diamond using your target FPGA device, you must create a new microcontroller platform in Mico System Builder (MSB). A platform generically refers to the hardware microcontroller 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**.

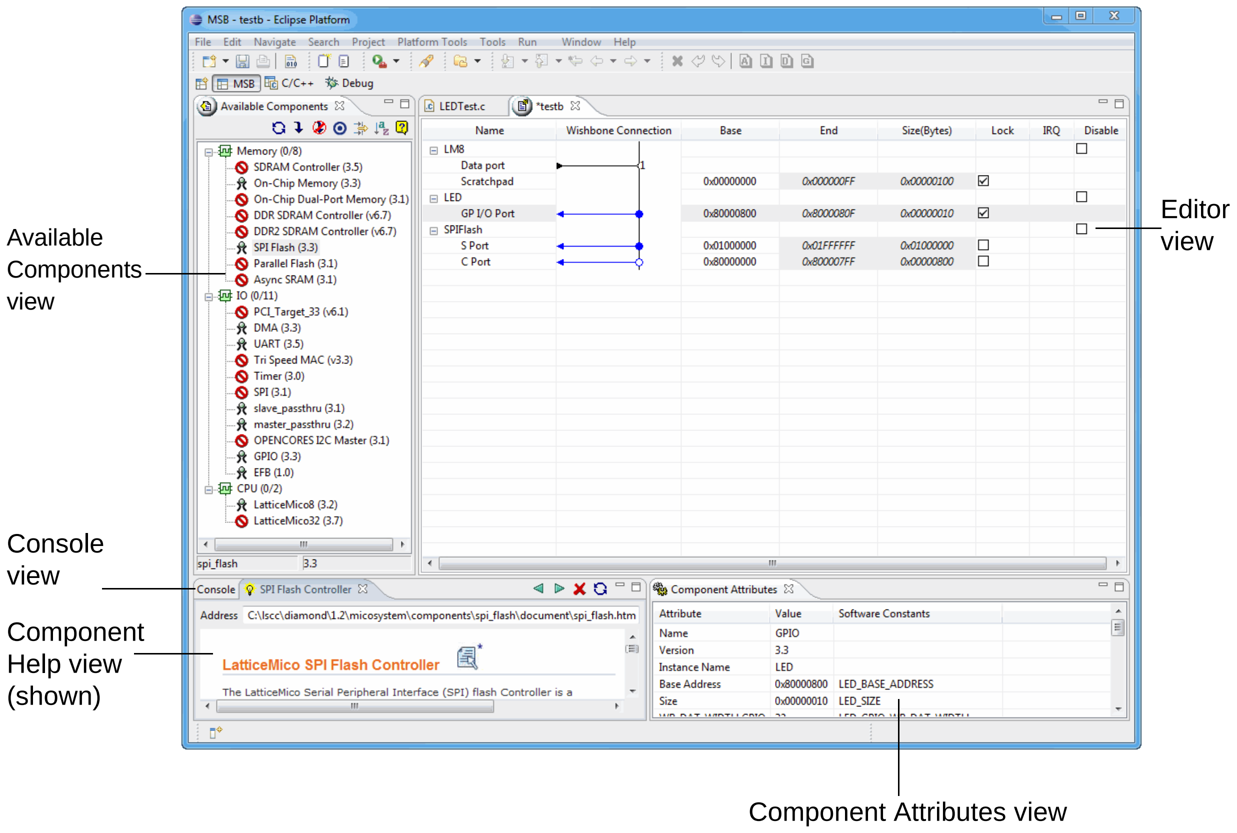
   During its launch process, the LatticeMico System software creates an Eclipse workspace file. This file is created in your home directory. On the Windows operating systems, it is in the Documents and Settings directory. On the Linux operating system, it is in ~.

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


MSB - testb - Eclipse Platform
File Edit Navigate Search Project Platform Tools Tools Run Window Help
MSB C/C++ Debug
Available Components
LEDTest.c
*testb
Console
SPI Flash Controller
Component Attributes
Available Components view
Editor view
Console view
Component Help view (shown)
Component Attributes view

Memory (0/8)
- SDRAM Controller (3.5)
- On-Chip Memory (3.3)
- On-Chip Dual-Port Memory (3.1)
- DDR SDRAM Controller (v6.7)
- DDR2 SDRAM Controller (v6.7)
- SPI Flash (3.3)
- Parallel Flash (3.1)
- Async SRAM (3.1)

IO (0/11)
- PCI_Target_33 (v6.1)
- DMA (3.3)
- UART (3.5)
- Tri Speed MAC (v3.3)
- Timer (3.0)
- SPI (3.1)
- slave_passthru (3.1)
- master_passthru (3.2)
- OPENCORES I2C Master (3.1)
- GPIO (3.3)
- EFB (1.0)

CPU (0/2)
- LatticeMico8 (3.2)
- LatticeMico32 (3.7)

spi_flash 3.3

Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disable
LM8							☐
Data port	1						
Scratchpad		0x00000000	0x000000FF	0x00000100	☑		
LED							☐
GP I/O Port		0x80000800	0x8000080F	0x00000010	☑		
SPIFlash							☐
S Port		0x01000000	0x01FFFFFF	0x01000000	☐		
C Port		0x80000000	0x800007FF	0x00000800	☐		

Address C:\lscc\diamond\1.2\micosystem\components\spi_flash\document\spi_flash.htm

LatticeMico SPI Flash Controller

The LatticeMico Serial Peripheral Interface (SPI) flash Controller is a

Attribute	Value	Software Constants
Name	GPIO	
Version	3.3	
Instance Name	LED	
Base Address	0x80000800	LED_BASE_ADDRESS
Size	0x00000010	LED_SIZE

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.

Note

The Available Components view shows all the hardware components that are part of the installer. This does not mean that all these components are available for a given combination of processor (LatticeMico8 and LatticeMico32), FPGA family and part number. The components that are not available are marked with a 🚫.

- 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 LatticeMico on-chip memory controller and LatticeMico 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 microcontroller. 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 LatticeMico online Help.

Creating a Platform Description in MSB

After you have created a new project in Lattice Diamond, you must create a new microcontroller platform description in Mico System Builder (MSB). A platform generically refers to the hardware microcontroller 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 microcontroller platform.

To create a new microcontroller .msb file:

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

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

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.

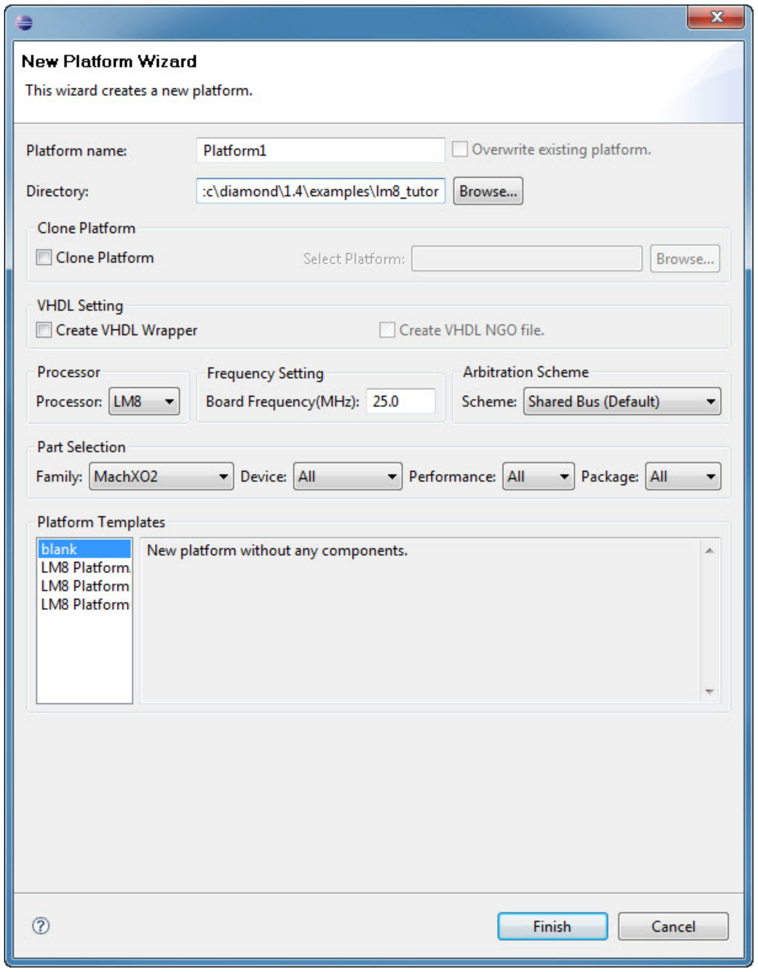
   a. Select **Create VHDL Wrapper**.

   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 Processor box, select **LM8** from the pull-down menu.
7. In the Arbitration Scheme box, select the desired arbitration scheme from the pull-down menu.
8. In the Device Family section, select a Lattice family and a device from the pull-down menus.
9. 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

Figure 6: New Platform Wizard Dialog Box


New Platform Wizard
This wizard creates a new platform.

Platform name: Platform1 ☐ Overwrite existing platform.
Directory: :c\diamond\1.4\examples\lm8_tutor Browse...

Clone Platform
☐ Clone Platform Select Platform: Browse...

VHDL Setting
☐ Create VHDL Wrapper ☐ Create VHDL NGO file.

Processor
Processor: LM8

Frequency Setting
Board Frequency(MHz): 25.0

Arbitration Scheme
Scheme: Shared Bus (Default)

Part Selection
Family: MachXO2 Device: All Performance: All Package: All

Platform Templates
blank
LM8 Platform
LM8 Platform
LM8 Platform

New platform without any components.

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.

10. If you have not selected Clone Platform, select the desired template from the Platform Templates list; or select Blank for a new template.
11. 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 Microcontroller 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 microcontroller 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 LatticeMico with the Diamond software. See the references given in “LatticeMico System Requirements” on page 8 for information on installing Diamond and LatticeMico System.

To add the LatticeMico8 microcontroller to the design:

1. Double-click on the **LatticeMico8** 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 LatticeMico8 dialog box and click **OK**.

LatticeMico System provides several peripheral components, I/Os, and memories that you can add to your microcontroller 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 LatticeMico8 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 LatticeMico8 microcontroller and the LatticeMico 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.

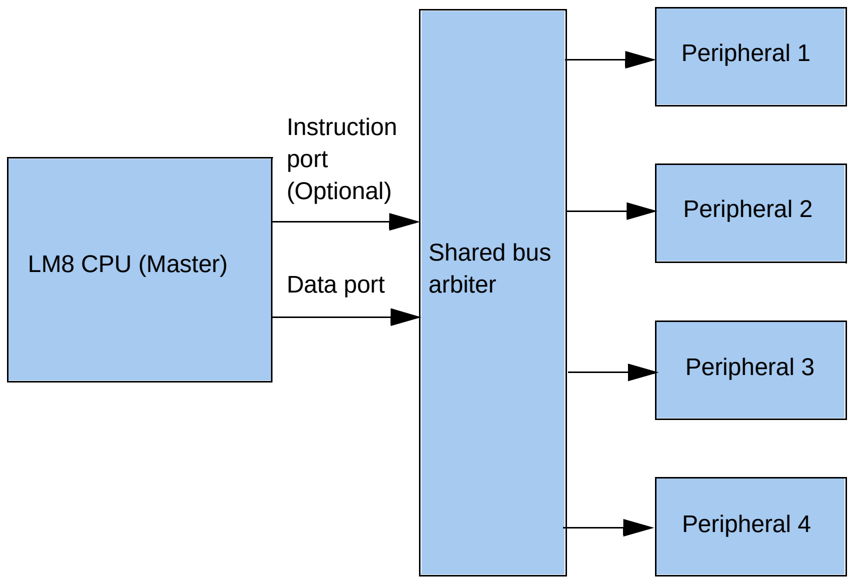
Arbitration Schemes

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 microcontroller 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.

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



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 LatticeMico8 microcontroller 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.

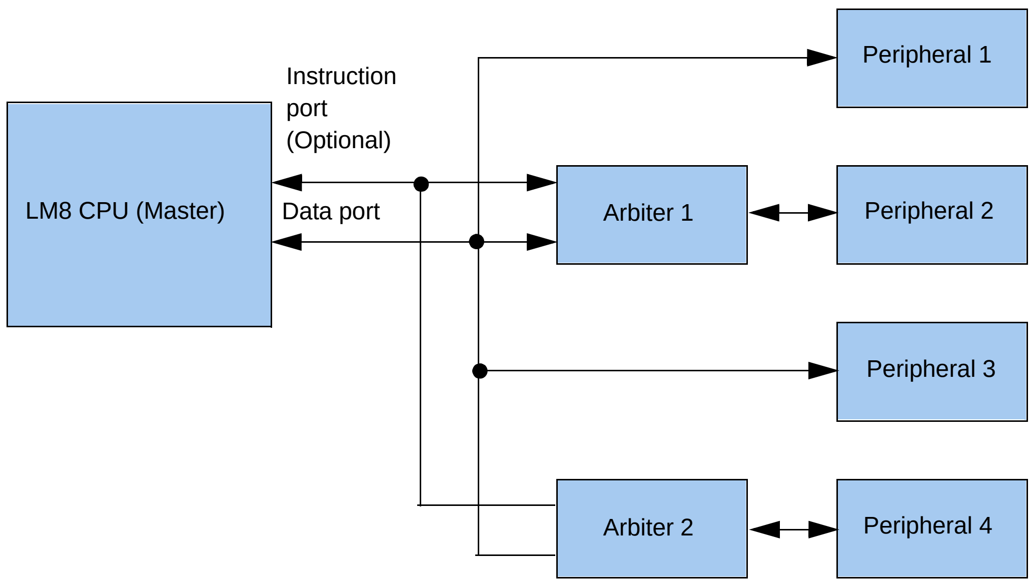
Slave-Side Arbitration Figure 8 shows the connections made by MSB when the slave-side arbitration scheme is chosen.

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

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

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



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.

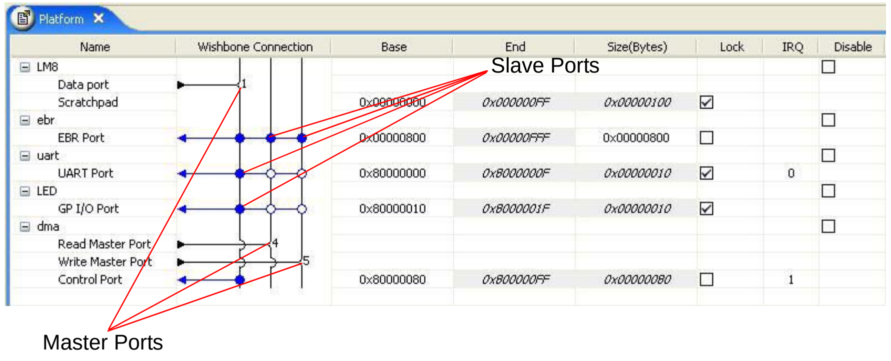
The connections that you made are saved in the .msb file.

Figure 9 shows an example of the connections that result in the Editor view when:

- The shared-bus arbitration scheme is used.
- The slave-side fixed and slave-side round-robin arbitration schemes are used.

All master signal connection lines are represented in black, and all slave connection lines are represented in blue.

Figure 9: Connecting Master/Slave Ports in Editor View



Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disable
⊟ LM8							☐
Data port	1						
Scratchpad		0x00000000	0x000000FF	0x00000100	☑		
⊟ ebr							☐
EBR Port		0x00000800	0x00000FFF	0x00000800	☐		
⊟ uart							☐
UART Port		0x80000000	0x8000000F	0x00000010	☑	0	
⊟ LED							☐
GP I/O Port		0x80000010	0x8000001F	0x00000010	☑		
⊟ dma							☐
Read Master Port	4						
Write Master Port	5						
Control Port		0x80000080	0x800000FF	0x00000080	☐	1	

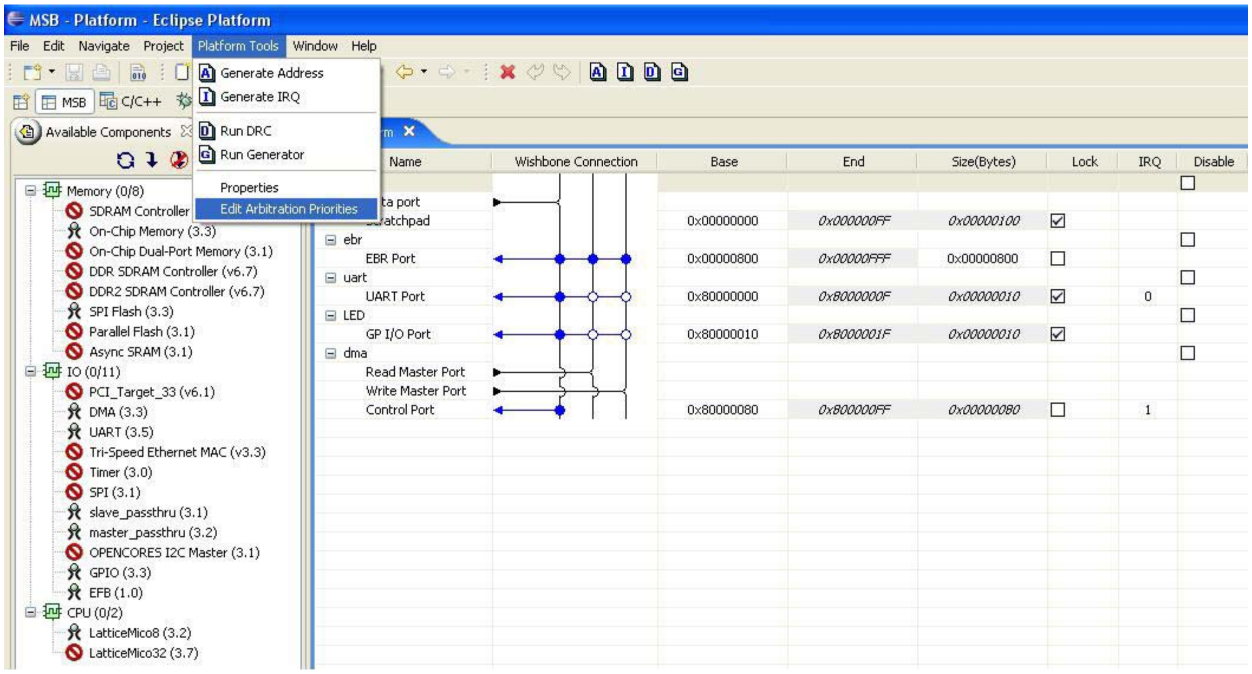
In the slave-side fixed arbitration scheme, you can change the priorities of the master ports, so the Edit Arbitration Priorities command is available on the Platform Tools menu, as shown in Figure 10. However, in the slave-side round-robin arbitration scheme, you cannot change the priorities of the master ports because the arbitration between the master ports occurs in a round-robin fashion. The Edit Arbitration Priorities command on the Platform Tools menu is therefore disabled when you use the slave-side round-robin arbitration scheme, as shown in Figure 11.

Figure 10 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.

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.

Figure 10: MSB Perspective After Adding All Components in a Slave-Side Fixed Arbitration Scheme


MSB - Platform - Eclipse Platform
File Edit Navigate Project Platform Tools Window Help
MSB C/C++
Available Components
Generate Address
Generate IRQ
Run DRC
Run Generator
Properties
Edit Arbitration Priorities

- Memory (0/8)
  - SDRAM Controller
  - On-Chip Memory (3.3)
  - On-Chip Dual-Port Memory (3.1)
  - DDR SDRAM Controller (v6.7)
  - DDR2 SDRAM Controller (v6.7)
  - SPI Flash (3.3)
  - Parallel Flash (3.1)
  - Async SRAM (3.1)
- IO (0/11)
  - PCI_Target_33 (v6.1)
  - DMA (3.3)
  - UART (3.5)
  - Tri-Speed Ethernet MAC (v3.3)
  - Timer (3.0)
  - SPI (3.1)
  - slave_passthru (3.1)
  - master_passthru (3.2)
  - OPENCORES I2C Master (3.1)
  - GPIO (3.3)
  - EFB (1.0)
- CPU (0/2)
  - LatticeMico8 (3.2)
  - LatticeMico32 (3.7)

Name	Base	End	Size(Bytes)	Lock	IRQ	Disable
						☐
ta port						
atchpad	0x00000000	0x000000FF	0x00000100	☑		
ebr						☐
EBR Port	0x00000800	0x00000FFF	0x00000800	☐		
uart						☐
UART Port	0x80000000	0x8000000F	0x00000010	☑	0	
LED						☐
GP I/O Port	0x80000010	0x8000001F	0x00000010	☑		
dma						☐
Read Master Port						
Write Master Port						
Control Port	0x80000080	0x800000FF	0x00000080	☐	1	

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.

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

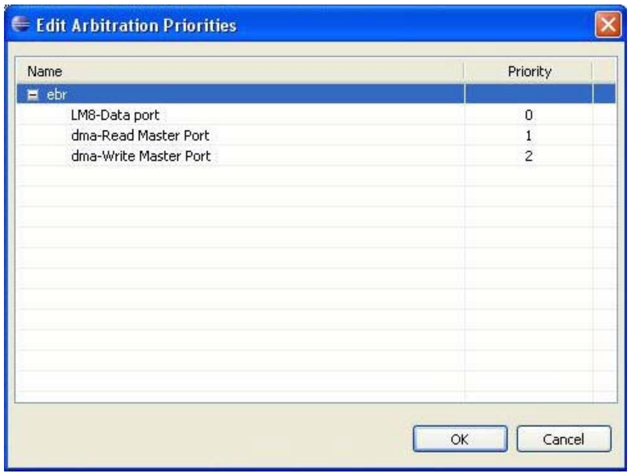
When you assign arbitration priorities to the master port of a slave in the slave-side fixed arbitration scheme, the number of priorities should not be greater than the total number of master ports for that slave. For example, if a slave has three master port values, the arbitration priorities would be 0, 1, and 2. An example is shown in Figure 11.

Assigning Component Addresses

After you add your components to your microcontroller 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

Figure 11: Edit Arbitration Priorities Error Message



Name	Priority
ebr	
LM8-Data port	0
dma-Read Master Port	1
dma-Write Master Port	2

component dialog box when you first add the component to the platform. Any 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 28 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 I. 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 D. You can also right-click in the Editor view and choose **Run DRC** from the pop-up menu.

Saving the Microcontroller Platform

After you do a number of tasks to set up your microcontroller 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 Microcontroller Platform

Generating the microcontroller 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 microcontroller 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 12 shows the instantiation template for the platform1 platform.

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.

Figure 12: 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]
);
```

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 Microcontroller 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 microcontroller 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 Microcontroller to FPGA Pins

You have two options for connecting the microcontroller 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 LatticeMico 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 microcontroller 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 **JEDEC 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 the Diamond Programmer to accomplish this task.

To download the bitstream to the FPGA on the board:

1. Remove any Lattice USB Programming cables from your system.
2. Connect the power supply to the development board.
3. Connect a USB cable from your computer to the MachXO2 Control Board. The USB cable must be connected to the USB target connector adjacent to the keypad. Give the computer a few seconds to detect the USB device on the MachXO2 Control board before moving to step 4.
4. In Diamond, choose **Tools > Programmer**.
5. In the Getting Started dialog box, choose **Create a new Blank Project**. Leave the **Import File to Current Implementation** box checked, and click **OK**. Programmer scans the device database, and then the Programmer view displays in Diamond.
6. Double-click the File Name column. Click ... to display the Open File dialog box, and browse to the .jed file you generated in the previous section.
7. Click **Open**.
8. Double-click the Operation column to display the Device Properties dialog box, and choose the following settings:
   - For Access Mode, choose **Flash Programming Mode** from the pull-down menu.
   - For Operation, choose **Flash, Erase, Program, Verify** from the pull-down menu.
9. Click the Program button on the Programmer toolbar to initiate the download. If the programming process succeeded, you will see a green-shaded PASS in the Programmer Status column.

At the end of this process, the FPGA is loaded with the microcontroller hardware configuration.

Using C/C++ SPE to Develop Your Software

After creating your hardware microcontroller platform, you must create the software application code that defines how it processes data. This section outlines how to use the LatticeMico C/C++ Software Project Environment (SPE), the primary tool that you use to develop your microcontroller application code. You do tasks that use C/C++ SPE in the C/C++ perspective in the user interface.

The C/C++ perspective enables you to do the following tasks:

- Create and build new LatticeMico8 C, Assembly, and C+Assembly software projects.
- Develop and compile your software application code to create executables using its workbench.

Starting C/C++ SPE

C/C++ SPE is another functional part of LatticeMico System, and you can access its commands in the C/C++ perspective. You can also access C/C++ commands from other perspectives. See “LatticeMico System Perspectives” on page 8 to understand how command options for various functional parts of the software are accessed in the software.

Before opening the C/C++ perspective, have the software running, as described in “Running LatticeMico System” on page 8.

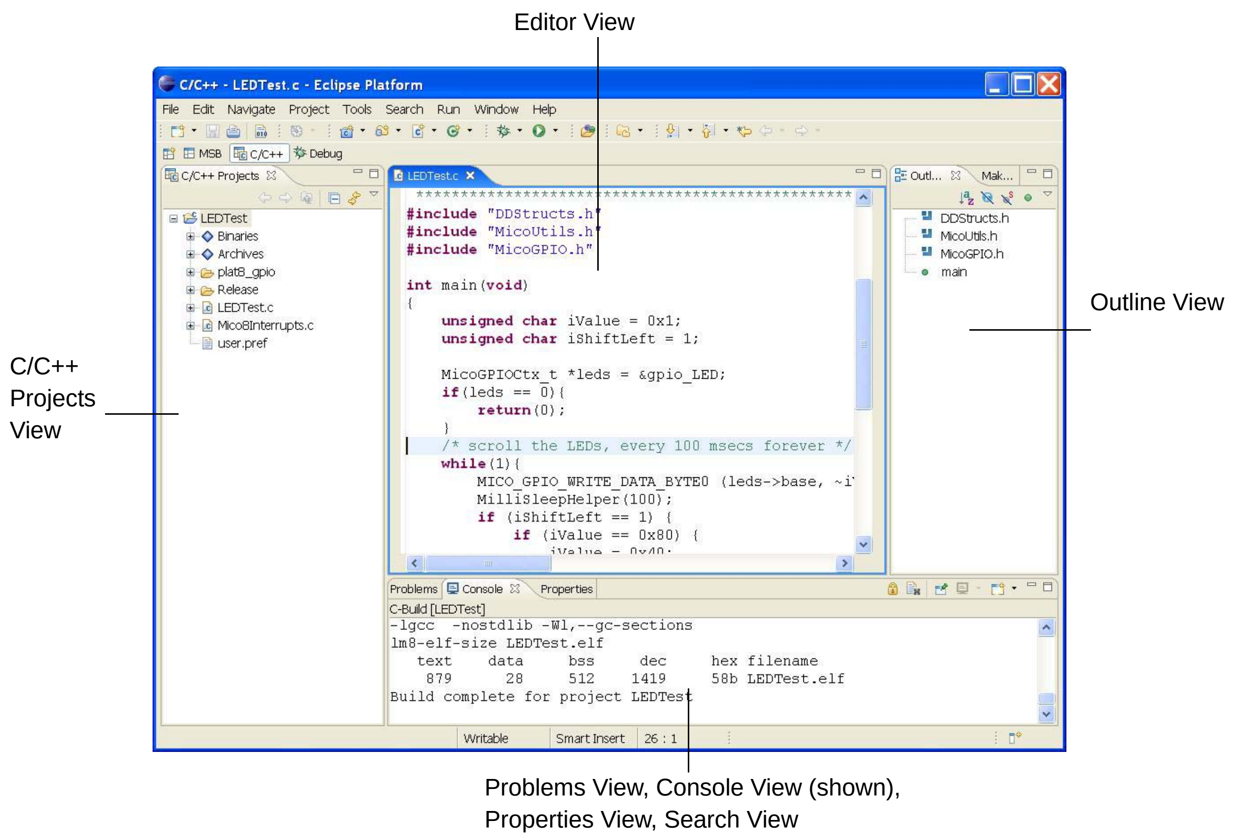
To open the C/C++ perspective:

- From the default MSB perspective, click the **C/C++** activation button C/C++ at the top left.

  Alternatively, you can choose **Window > Open Perspective > C/C++**.

The C/C++ perspective now becomes active and enables you to access C/C++ SPE commands. The current active perspective is shown in the upper left of the window's title bar, as shown in Figure 13.

Figure 13: C/C++ Perspective



```
 ****************************************************
#include "DDStructs.h"
#include "MicoUtils.h"
#include "MicoGPIO.h"

int main(void)
{
    unsigned char iValue = 0x1;
    unsigned char iShiftLeft = 1;

    MicoGPIOCtx_t *leds = &gpio_LED;
    if(leds == 0){
        return(0);
    }
    /* scroll the LEDs, every 100 msecs forever */
    while(1){
        MICO_GPIO_WRITE_DATA_BYTE0 (leds->base, ~i
        MilliSleepHelper(100);
        if (iShiftLeft == 1) {
            if (iValue == 0x80) {
```

```
-lgcc  -nostdlib -Wl,--gc-sections
lm8-elf-size LEDTest.elf
   text    data     bss     dec     hex filename
    879      28     512    1419     58b LEDTest.elf
Build complete for project LEDTest
```

The C/C++ perspective consists of the following views:

- C/C++ Projects view, which lists C/C++ SPE projects that have been created
- Navigator view, which shows all of the file system files under the workspace folder
- Editor view, which displays your editable files in the window. Each file is displayed within a separate tab within the view.
- Outline view, which displays the structure of the file currently open in the Editor view. See the online Help for more details.
- Problems view, which displays error, warning, or informational messages output related to your build
- Console view, which displays informational messages output by the C/C++ SPE build process
- Properties view, which displays the attributes of the item currently selected in the Projects view. This view is read-only.

- Search view, which displays the results of a search when you choose the Search > Search menu command
- Tasks view, which shows the tasks running concurrently in the background
- Make Targets view, which allows you to create your own custom makefiles. This ability is not necessary for managed make projects.

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 online Help.

Creating Software Projects

There are three main types of software projects:

- LatticeMico8 managed make C project
- LatticeMico8 library project
- LatticeMico8 standard make C project

A LatticeMico8 managed make C project is the easiest to use for getting started, because it manages the build environment, including linker scripts, boot code, sources, header files, and even makefiles. It also extracts platform-dependent information from the LatticeMico8 microcontroller platform and creates the appropriate files required for a build.

The LatticeMico8 library project and the LatticeMico8 standard-make project are described in “Advanced Programming Topics” on page 119. These two project types enable you to create your own build environment in which you can provide the desired make structure, as well as make files. This document refers to the managed-build process for all topics unless explicitly stated otherwise.

Creating a project is the first step in using C/C++ SPE. You select a target platform generated by MSB in the .msb file that you already created and create the software application code that controls the microcontroller and attached components. At the same time, C/C++ SPE generates system libraries based on the MSB platform, your selections, or both. Use the **File > New > Mico Managed Make C/C++ Project** menu command to create a software project.

Note

The folder in which the C/C++ SPE project is saved cannot reside at the same directory level as the folder in which the MSB project is saved. The C/C++ SPE folder can reside at a higher or lower directory level than the MSB project folder.

Before using C/C++ SPE, you must define an MSB platform to select the drivers and the available memory for the linker. C/C++ SPE references one and only one MSB platform definition. You can retarget the same software application code to another MSB platform without having to recreate the project or without having to rewrite the software application code. The components used by the software application code must reside in both platforms to ensure a successful build.

To facilitate development, you can select a project template to use in creating the software application code in C/C++ SPE and then modify this code. But once you create a project, you cannot change the template, because some templates have platform dependencies.

To create a new software project:

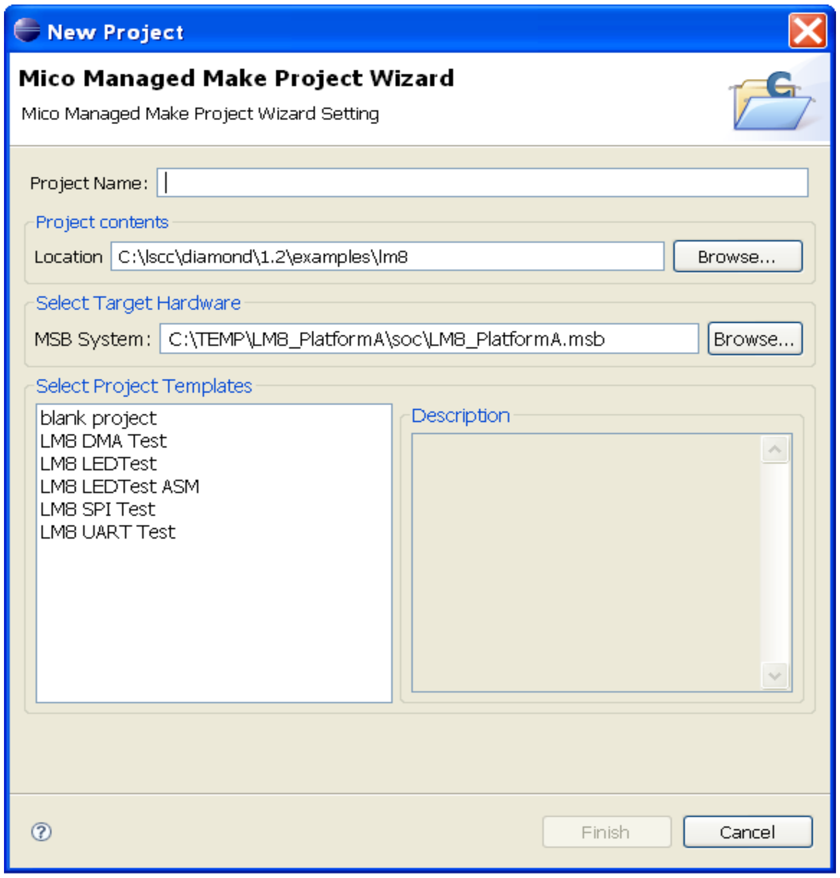
1. From the MSB perspective, click the **C/C++** button in the upper left.

   The C/C++ perspective opens.

2. In the C/C++ perspective, choose **File > New > Mico Managed Make C/C++ Project**.

   The New Project dialog box opens, as shown in Figure 14.

Figure 14: New Project Dialog Box



New Project

Mico Managed Make Project Wizard

Mico Managed Make Project Wizard Setting

Project Name:

Project contents

Location C:\lscc\diamond\1.2\examples\lm8

Select Target Hardware

MSB System: C:\TEMP\LM8_PlatformA\soc\LM8_PlatformA.msb

Select Project Templates

blank project
LM8 DMA Test
LM8 LEDTest
LM8 LEDTest ASM
LM8 SPI Test
LM8 UART Test

Description

3. In the Project Name box, enter the name of your new project.

   The Location text box points the top-level project folder where your software project's contents will be stored, including your sources as well as the managed build files. The name of your project is automatically appended to the default folder location. To override the default assignment, first enter the project name and then enter the desired location.

4. In the MSB System text box, browse to the location of the .msb file, select the .msb file, and click **Open**.

   This file is located within this MSB platform folder, where there will be an \soc folder that contains an .msb file.

   If you switched to C/C++ after opening an MSB platform or creating a new MSB platform, the MSB platform selection will, by default, contain the file name and path of that MSB platform description.

5. In the Select Project Templates list box, select the template for the application code.

   This list box allows for selection of available software templates for a quick start on software development. Software templates provide a collection of software project files that are copied into your project's folder. These provide you a starting point for creating your application. If you intend to create a blank project that contains no pre-existing files, select the blank project template. The Templates Description box provides information on selected platform component requirements and other relevant information.

6. Click **Finish**.

   Your software project has been created.

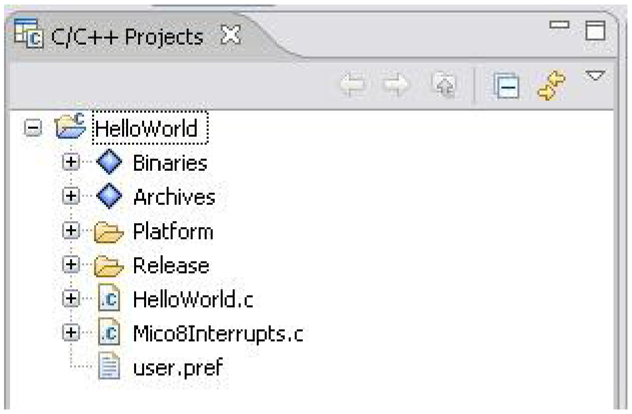
   Your new project will appear in the C/C++ Projects view.

7. Click on the project name to select it in the C/C++ Projects view on the left.

8. Choose **Project > Build Project**.

If you had selected a project template of the "hello world" variety during project setup, you would get the HelloWorld Projects view, as shown in Figure 15. The project folder in the view is shown expanded for illustrative purposes.

Figure 15: Hello World Projects View



As you can see in Figure 15, this project contains source files copied over as part of the template specification. Subsequent parts of this document describe the relevant project files, such as the ones shown here. See “Managed Build Process and Directory Structure” on page 91 for a discussion of the directory structure with a special focus on its relevance to the managed build process.

Basic Project Operations

This section describes some of the most commonly used operations for project development. The C/C++ SPE software enables you to perform a given operation in various ways, such as selecting from a pop-up menu or selecting from the application menu. This section describes the most common ways of performing these operations.

Note

LatticeMico8 C/C++ SPE is derived from Eclipse CDT, so basic project operations that apply to the Eclipse CDT perspective also apply to LatticeMico8 C/C++. Refer to the LatticeMico online Help for details on all available project manipulation operations.

Adding New Source Files or Folders

This section describes how to add new source files and folders to your C/C++ SPE project. Source files refer to .c files that contain your C programming code and are input into the C compiler to generate your object files. Source folders refer to directories that contain a host of .c files. Adding or creating a resource file in your project can refer to any file.

To add new source files to your C/C++ project:

1. In the C/C++ perspective, click on your project in the Projects view to select it.
2. Right-click on the project icon and choose **New > Source File** from the pop-up menu.
3. In the New Source File dialog box, browse to your source file and click **Finish**.

To add new source folders to your C/C++ project:

1. In the C/C++ perspective, click on your project in the Projects view to select it.
2. Right-click on the project icon and choose **New > Source Folder** from the pop-up menu.
3. In the New Source Folder dialog box, browse to your source folder and click **Finish**.

To add new file resources to your C/C++ project:

1. In the C/C++ perspective, click on your project in the Projects view to select it.
2. Right-click on the project icon and choose **New > Source File** from the pop-up menu.
3. In the New File dialog box, browse to you source folder and click **Finish**.

You can create subfolders within your project folder for organizing your source files. The managed build environment copies in the source files from these subfolders during the build process.

Deleting Software Project Contents

You can delete selected project contents in the Projects view. Deleting a project item does not erase the file from your hard disk. It simply deletes the visible project item in the C/C++ SPE interface.

To delete a C/C++ software project item:

1. In the C/C++ perspective, click on the project item in the Projects view to select it.
2. Right-click on the project it and choose **Delete** from the pop-up menu.

   This deletes the item from project definition, but not from your hard disk.

Renaming Software Project Contents

You can rename selected project contents in the Projects view. This section describes how to rename project items. Renaming a project item does not change its name on your hard disk. It simply changes the visible name of the project item in the C/C++ SPE interface.

To rename a C/C++ project item:

1. In the C/C++ perspective, click on the project item in the Projects view to select it.
2. Right-click on the project it and choose **Rename** from the pop-up menu.

   The project icon’s title box appears highlighted. It is editable.
3. Type the desired new name of the project item and click anywhere outside of the highlighted field or click **Enter**.

   The new name is established.

Adding Existing Files/Folders to a Project

You can add existing files or folders to your C project using Windows Explorer by directly copying and pasting or dragging and dropping them into your project.

To copy and paste existing files or folders into your software project:

1. In Windows Explorer, right-click on the files, folders, or both that you wish to copy into your project and choose **Copy** in the pop-up menu or use the **Ctrl+C** keyboard combination.

   This step copies the files, folders, or both to your Windows clipboard.

   If you wish to copy multiple files or folders, you can select them by using the Shift-click or Ctrl-click functionality.

2. In the C/C++ perspective's Projects view, right-click on the project folder and choose **Paste** from the pop-up menu or use the **Ctrl+V** keyboard combination.

   The file or folder appears in the hierarchy underneath the project folder.

To drag and drop files and folders into your software project:

1. In Windows Explorer, click on the files or folders or both that you wish to copy into your project. You can select multiple files for copying at once using the Shift-click or Ctrl-click functionality.
2. Drag the files over into your C/C++ perspective's Projects view onto a project folder until you see a plus sign on a “mouse over” with your cursor.
3. Release the mouse button.

   The selected files or folders are copied into the targeted folder in the Projects view.

Deleting a Project

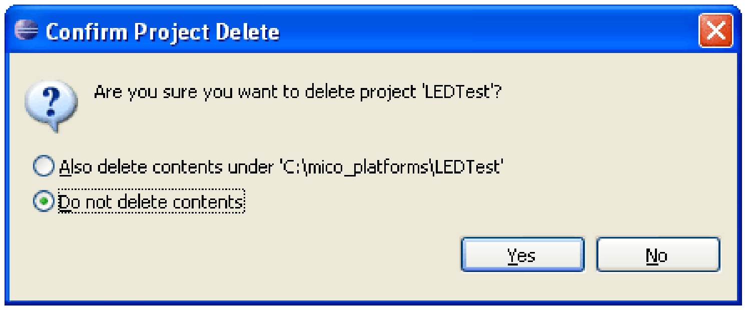
If you have created projects in your LatticeMico workspace that you want to remove, you can delete them from the Projects view.

To delete a software project:

1. In C/C++ perspective's Projects view, right-click on the folder of the project that you want to delete.
2. In the pop-up menu, choose **Delete**.

The Confirm Project Delete dialog box shown in Figure 16 now asks you if you are certain that you want to delete the project in the event that you selected this option by accident.

Figure 16: Project Deletion Confirmation Dialog Box



Are you sure you want to delete project 'LEDTest'?

Also delete contents under 'C:\mico_platforms\LEDTest' [✓ Do not delete contents]

3. Click **Yes**.

   If you select the option button to delete the contents of the folder as well, the project is deleted from your workspace on your hard disk, as well as from your Projects view.

   By default, as shown in Figure 16, the “Do not delete contents” option is selected. It only removes the folder in the Projects view. If you just remove the project from the Projects view, you have the option of importing the project back into your workspace later.

Importing an Existing Project

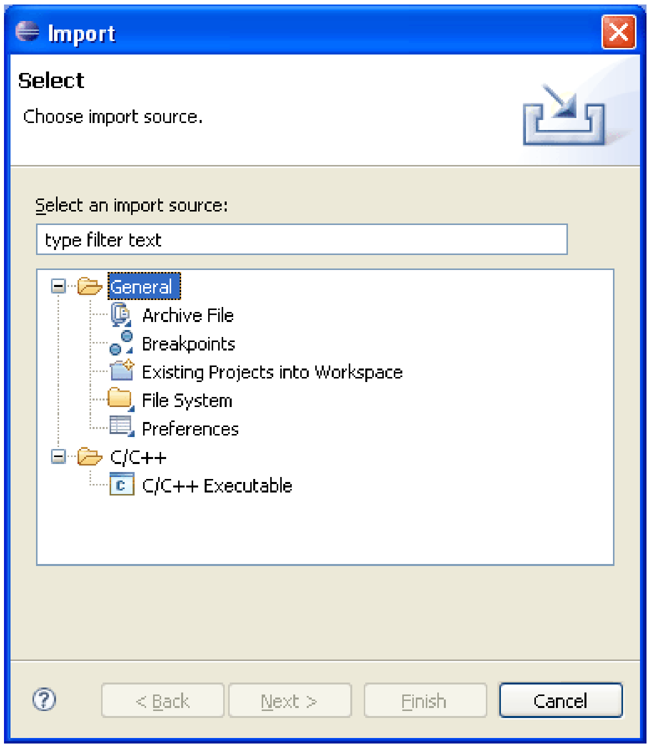
You can use the Import Wizard to copy a project from a different workspace or copy a project that previously existed in your workspace and import it into the LatticeMico8 software workbench. You cannot import a project that has the same project name as an existing project into the Projects view.

To import an existing project:

1. From within a given perspective, choose **File > Import**. You can also right-click on your project icon in the Projects view and select **Import** from the pop-up menu.

The Import dialog box opens in Select mode, as shown in Figure 17.

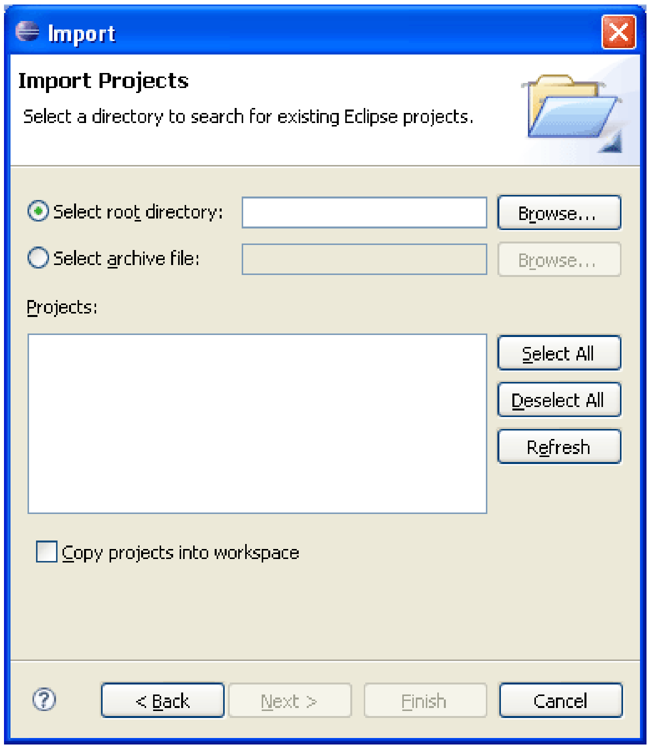
Figure 17: Import Dialog Box in Select Mode



2. Expand the General folder and select **Existing Project into Workspace**, and click **Next**.

The Import dialog box changes to Import Projects mode, as shown in Figure 18.

Figure 18: Import Dialog Box in Import Projects Mode



3. Choose either **Select root directory** or **Select archive file** and click the associated Browse button to locate the folder or file containing the project that you wish to import.
4. Under Projects, select the project or projects that you would like to import.
5. Click **Finish** to start the import.

   If the project is successfully imported, it will appear in the Projects view.

Understanding the Build Process

Once you develop the software application code, you must compile and link it to generate an executable.

Building a project involves compiling, assembling, and linking the software application code, as well as the system library code generated by the C/C++ SPE. Each step in this process has associated settings that affect the build. A group of such settings is called a build configuration.

A newly created C/C++ SPE project provides only a release build configuration for generating an optimized executable (devoid of any debug information) for the LatticeMico8 microcontroller.

The build process involves creating makefiles and then performing a make operation on the top-level makefile that, in turn, pulls in the required makefiles. This process creates makefiles for the software application code structure (typically subfolders for code organization) and creates makefiles for the platform library.

The build process also involves creating the linker settings for the software application that is being compiled in to an executable. These linker settings describe which memories within the MSB platform contain the compiler-dependent sections of the application software. For example, where the text, read-only data, and read/write data sections are located. These settings are especially important when the platform contains multiple memories. The linker settings are automatically generated from the LatticeMico8 microcontroller's configuration settings specified in the MSB platform file. The section settings are updated when a change to the MSB platform file is detected.

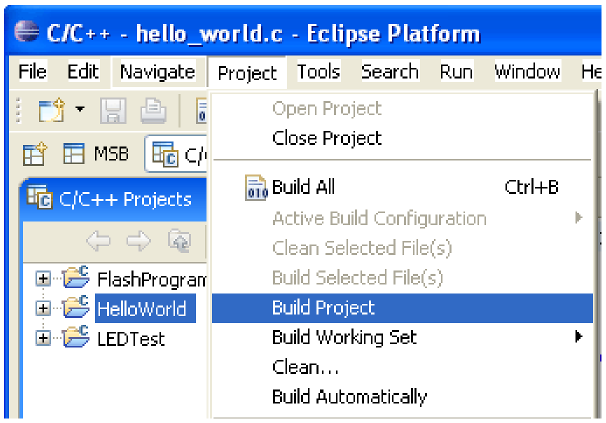
Building Your Software Project

This section describes how to build your software project, that is, to create all of the necessary files that you must have in place to properly deploy your software application code.

To build your project:

1. In the C/C++ perspective, right-click the desired project folder in the Projects view on the left. In the example in Figure 19, the highlighted project folder in the Projects view is called HelloWorld.

Figure 19: Build Project Selection



2. In the pop-up menu, choose **Build Project**. Alternatively, you can build a project by choosing the **Project > Build Project** menu command, as illustrated in Figure 19.

If the build has potential warnings or errors, Eclipse CDT might place an information icon next to the project folder in the Projects view.

The Console view in the C/C++ perspective displays the project build messages. The Problems view in the C/C++ perspective displays problems encountered during the build. Along with other icons, the Problems view may display a warning or error icon:

- The warning icon ⚠ indicates that there was an associated warning message that was generated by the build process.
- The error icon ⊗ indicates that there was an associated error message generated by the build process.

For a complete list of icons in the user interface that may be displayed and their meanings, refer to Eclipse/CDT and the LatticeMico8 System online Help.

Setting Project Properties

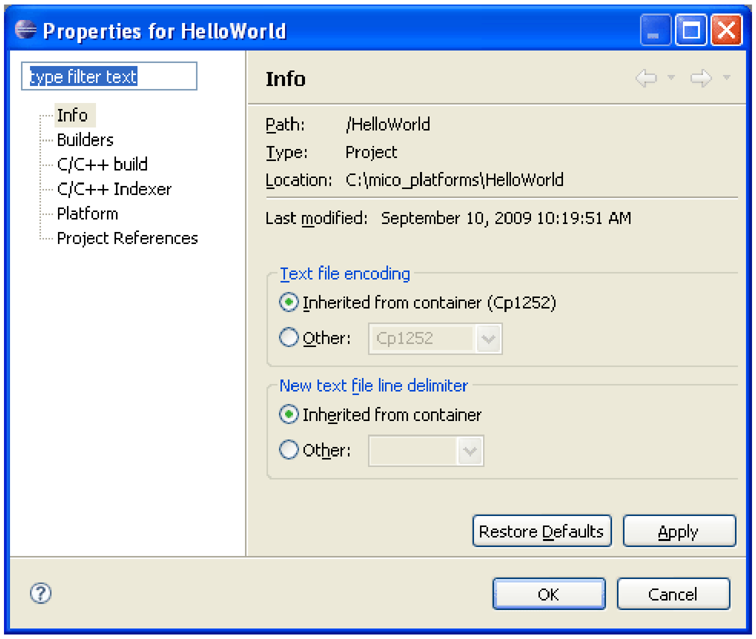
You can set up your project properties in the Properties dialog box. Project properties include various project parameters, for example, file encoding parameters, build configuration options, and platform settings. The Project Property dialog box is dynamic in that it enables you to select different “tabs” from the list box at left, which changes the display parameter set in the main option area of the dialog box.

To set project properties:

1. In the C/C++ perspective, right-click the desired project folder in the Projects view on the left. In the example in Figure 19 on page 47, the project is entitled HelloWorld.
2. In the pop-up menu, choose **Properties**.

The Properties dialog box appears, as shown in Figure 20.

Figure 20: Properties Dialog Box


Properties for HelloWorld

type filter text

- Info
- Builders
- C/C++ build
- C/C++ Indexer
- Platform
- Project References

Info

Path: /HelloWorld
Type: Project
Location: C:\mico_platforms\HelloWorld

Last modified: September 10, 2009 10:19:51 AM

Text file encoding
● Inherited from container (Cp1252)
○ Other: Cp1252

New text file line delimiter
● Inherited from container
○ Other:

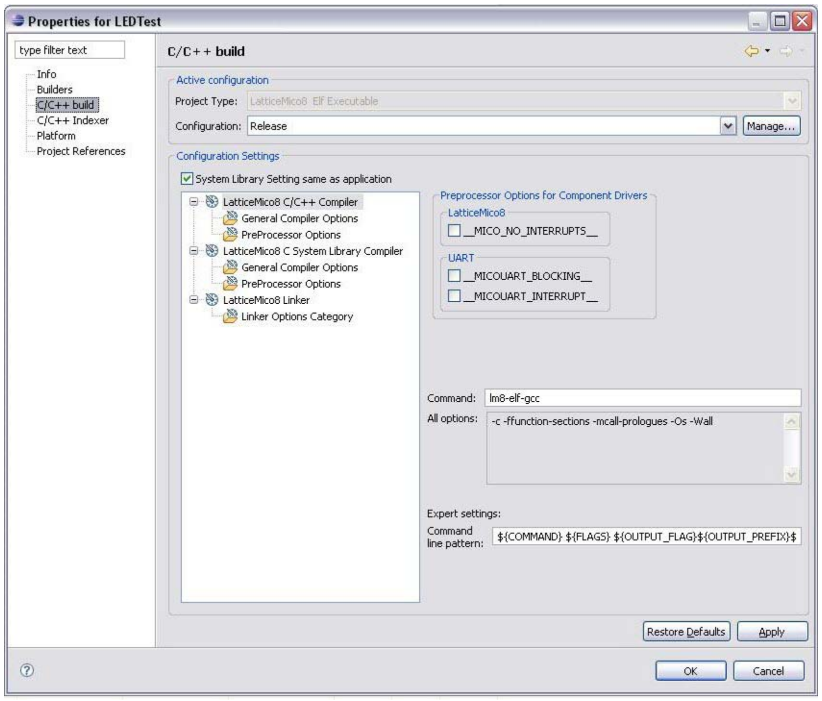
Restore Defaults | Apply

OK | Cancel

The Properties dialog box enables you to set the C/C++ build settings through the C/C++ build tab and the platform preferences through the Platform tab. See the list box on the left side of the Properties dialog box, as shown in Figure 20.

The C/C++ build tab, as shown in Figure 21, enables you to set build properties for the project.

Figure 21: C/C++ Build Tab of Properties Dialog Box



Project Type: LatticeMico8 Elf Executable

Configuration: Release

☑ System Library Setting same as application

☐ __MICO_NO_INTERRUPTS__

☐ __MICOUART_BLOCKING__

☐ __MICOUART_INTERRUPT__

Command: lm8-elf-gcc

All options: -c -ffunction-sections -mcall-prologues -Os -Wall

Expert settings:

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG}${OUTPUT_PREFIX}$

The C/C++ build tab enables you to set compiler and linker options for a given build. This tab contains several options:

- Active Configuration – This option allows you to select the active build configuration, as well as to modify the default settings. It also enables you to define your own configurations. LatticeMico8 C SPE uses the LatticeMico8 GNU C tool chain for project compilation and linking. A set of C build settings is known as a build configuration.

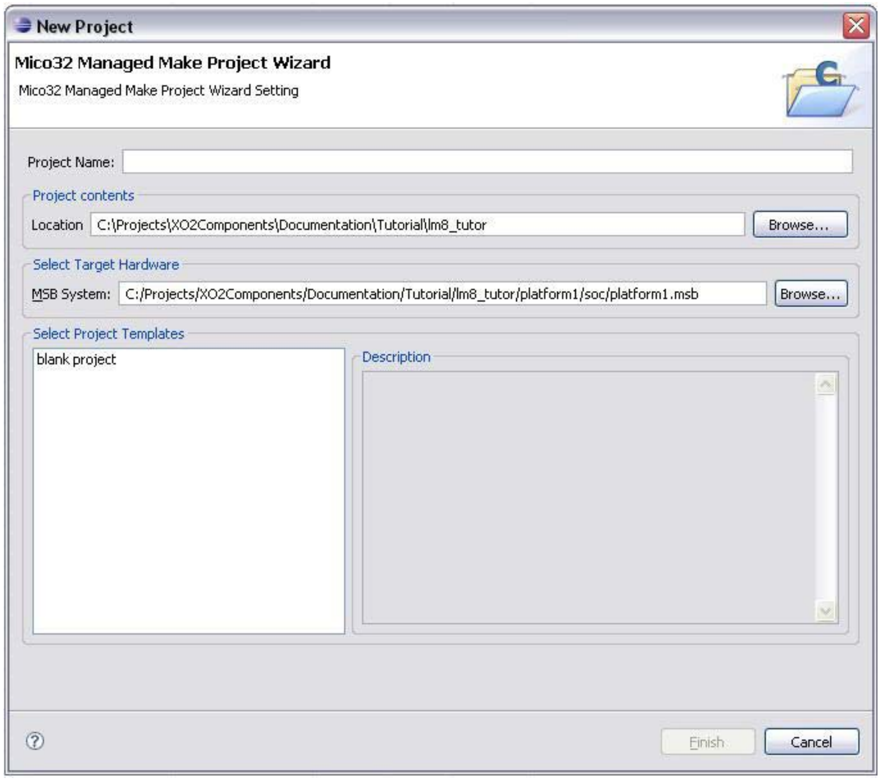
  The C/C++ SPE has only one predefined configuration, Release, for LatticeMico8 applications. To define you own configurations, refer to the Eclipse CDT documentation.

- Configuration Settings – This tab enables you to view or modify the compiler or linker settings.

  - "System Library Settings same as application" enables you to select application compiler settings and use these settings as system library compilation settings. You can apply separate compiler settings for the application and the system library build by clearing this option and selecting appropriate settings. The system library build is part of the managed build process described in "Managed Build Process and Directory Structure" on page 91. The LatticeMico8 linker settings affect the generation of the application executable.

Figure 22 shows the platform settings that are accessible in the Platform tab of the Properties dialog box.

Figure 22: Platform Tab of the Properties Dialog Box


New Project

Mico32 Managed Make Project Wizard

Mico32 Managed Make Project Wizard Setting

Project Name:

Project contents

Location C:\Projects\XO2Components\Documentation\Tutorial\lm8_tutor

Select Target Hardware

MSB System: C:/Projects/XO2Components/Documentation/Tutorial/lm8_tutor/platform1/soc/platform1.msb

Select Project Templates

blank project

Description

The Platform tab is further subdivided into the following fields:

- Target Hardware Platform – This option shows the currently selected platform for the selected project in the MSB System text box. You can retarget this software application to another platform by using the Browse button to select the appropriate platform. You must make sure that the platform that you select and your software applications are compatible with each other.
- Linker Script – By default, C/C++ SPE always generates a linker script usable for linking the selected project. This default linker script is generated from the target hardware platform's MSB file by parsing the configuration settings for the LatticeMico8 microcontroller. You can provide your own linker script by selecting the 'Use Custome Linker Script' button.

Rebuilding Your Software Project

After you create your project, you can perform subsequent builds by right-clicking the project name in the C/C++ perspective’s Projects view and choosing **Build Project** from the pop-up menu.

A release build configuration is for generating an optimized executable devoid of any debug information.

In the Eclipse/CDT, you can change the default settings that the C/C++ SPE remembers for the project, and you can create new build configurations with customized settings.

Performing Builds Automatically

You can set up the software workbench to automatically perform incremental builds whenever sources are saved.

To indicate that you want the software to perform incremental builds whenever resources are saved:

- Within a given perspective, choose **Project > Build automatically**.

  The workbench automatically performs incremental builds of resources modified since the last build. Whenever a resource is modified, another incremental build is run.

Deploying Your Software to LatticeMico8 Platform

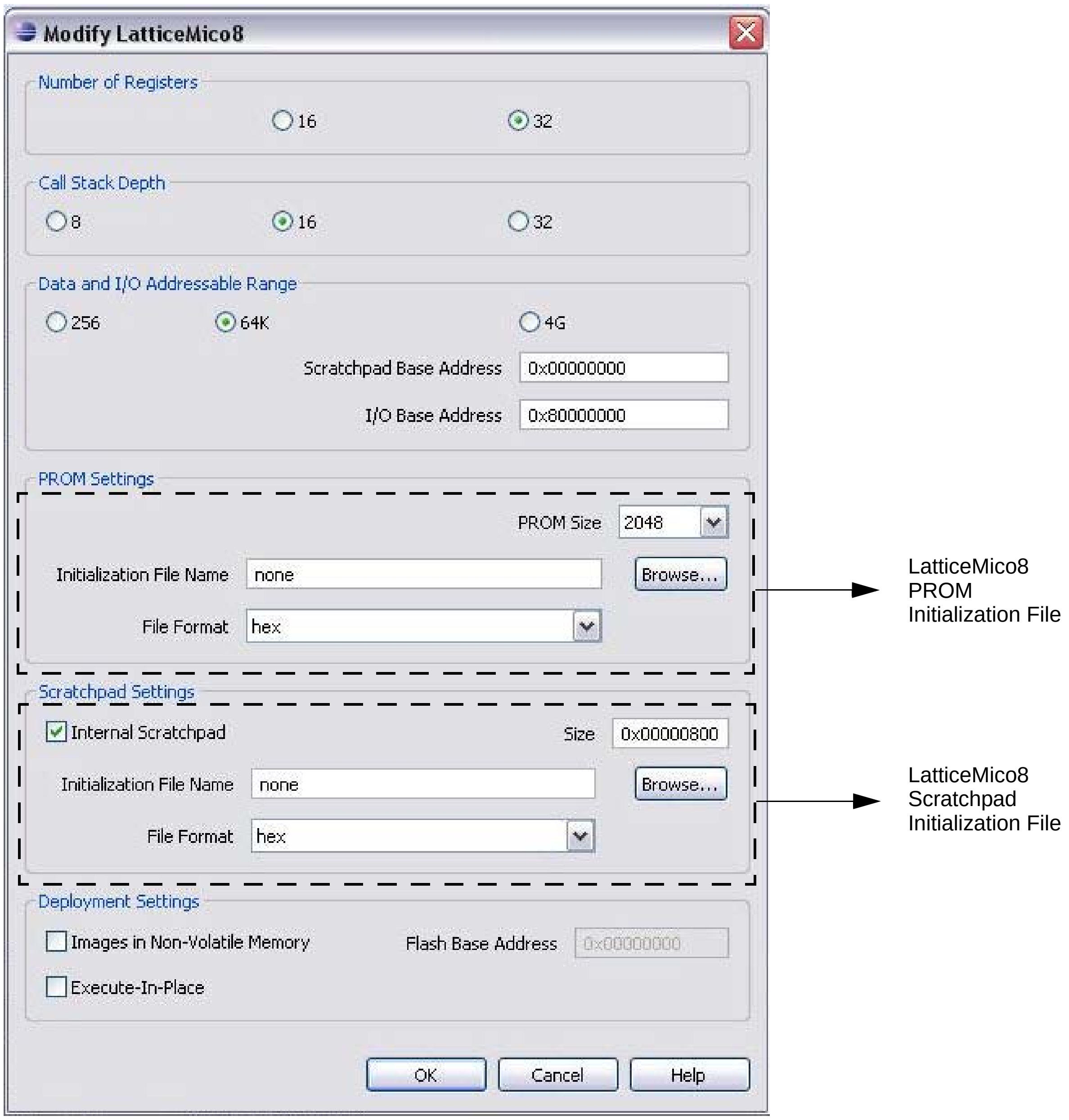
Once the software application code is created and built using the C/C++ SPE, it can be deployed to the hardware platform. The software can be deployed to on-chip memory or non-volatile memories such as SPI flash. The C/C++ SPE builds the application code in to an ELF executable in which cross-references between multiple object files are resolved, and similar sections are grouped together in to contiguous locations and loaded at the correct addresses in memory. There are two types of sections that are important from a software developer's perspective: code and data. The *code* sections should reside in the LatticeMico8 PROM and the *data* sections should reside in the LatticeMico8 Scratchpad.

The PROM and Scratchpad are implemented using On-Chip memories which can be initialized during platform bitstream generation. Section "On-Chip Memory Deployment" discusses this flow. The other option is to locate the PROM and Scratchpad images in non-volatile memories such as SPI flash and let the LatticeMico8 microcontroller load these images in to the PROM and Scratchpad at run time. Section "Non-Volatile Memory Deployment" discusses this flow.

On-Chip Memory Deployment

The Lattice on-chip memory can be initialized with valid content prior to generation of platform bitstream. The PROM and Scratchpad of the LatticeMico8 microcontroller are implemented using these on-chip memories, and can therefore be initialized with the *code* and *data* sections of the ELF executable. The deployment flow discussed in this Section generates the initialization files for the PROM (file “prom_init.mem”) and the Scratchpad (file “scratchpad_init.mem”). The developer initializes the LatticeMico8 microcontroller with these files within MSB and then regenerates the hardware platform. The sections of the LatticeMico8 GUI that deal with memory initialization are highlighted in Figure 23.

Figure 23: LatticeMico8 PROM Initialization File


Modify LatticeMico8
LatticeMico8 PROM Initialization File
LatticeMico8 Scratchpad Initialization File
Browse...
OK
Cancel
Help

Number of Registers: 16 [✓ 32]

Call Stack Depth: 8 [✓ 16] 32

Data and I/O Addressable Range: 256 [✓ 64K] 4G

Scratchpad Base Address: 0x00000000

I/O Base Address: 0x80000000

PROM Settings

PROM Size: 2048

Initialization File Name: none

File Format: hex

Scratchpad Settings

[✓ Internal Scratchpad]

Size: 0x00000800

Initialization File Name: none

File Format: hex

Deployment Settings

☐ Images in Non-Volatile Memory

Flash Base Address: 0x00000000

☐ Execute-In-Place

To generate the initialization files:

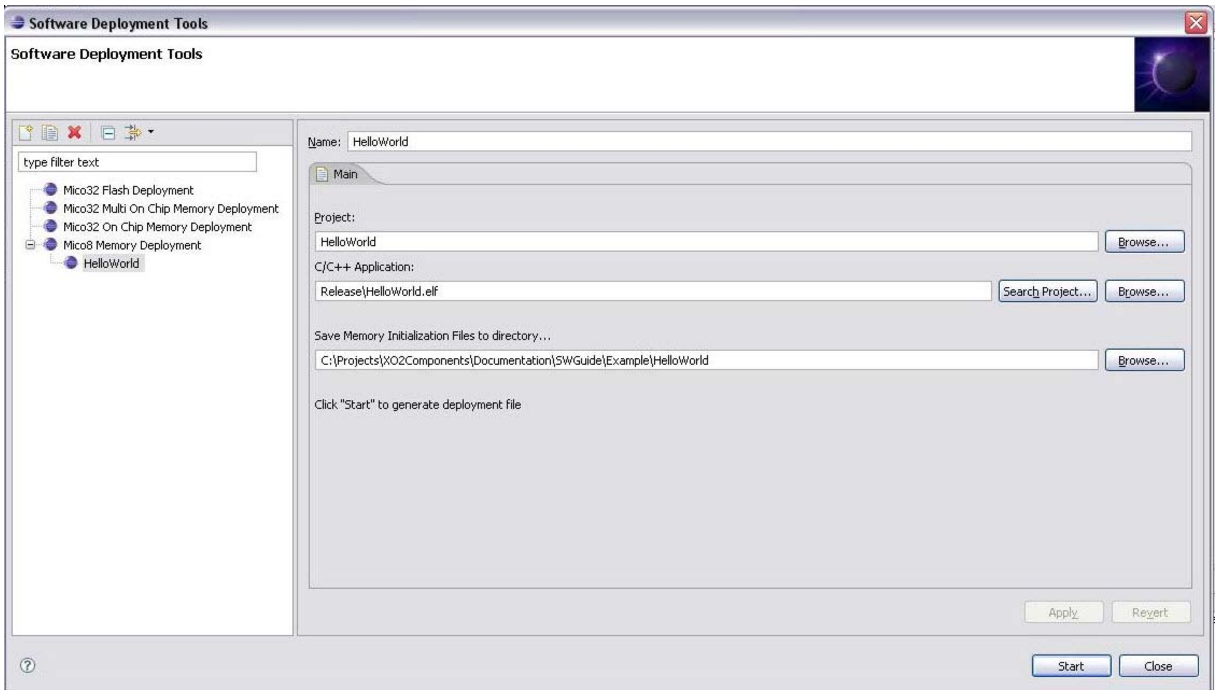
1. In the C/C++ perspective, launch the **Software Deployment** GUI from the **Tools** pull-down menu. This brings up the GUI shown in Figure 24.
2. Select the ‘Mico8 Memory Deployment’ tab and then press the ‘New’ button to create a new configuration. Figure 24 shows this new configuration. The developer must specify the following items within the configuration

- Name – This item refers to the name by which the configuration will be saved so that you can reuse it the next time you launch Mico8 Memory Deployment.
- Project – Specifies the C/C++ SPE project to use for selecting an application to deploy. Click the **Browse** button for a list of available selections.
- C/C++ Application – Specifies the application (.elf file) to be deployed in the selected project. Click the **Browse** button for a list of available applications in the selected project, or click the **Search Project** button to select an application (.elf file).
- Save Memory Initialization Files – The software developer must specify the folder within which the initialization files will be saved.

  Figure Figure 24 shows the values that are entered within the four aforementioned items for the HelloWorld software project.

3. Save the configuration by clicking on Apply.
4. Generate the initialization files “prom_init.mem” and “scratchpad_init.mem” by clicking on Start.

Figure 24: Software Deployment Dialog Box



Name: HelloWorld

Project: HelloWorld

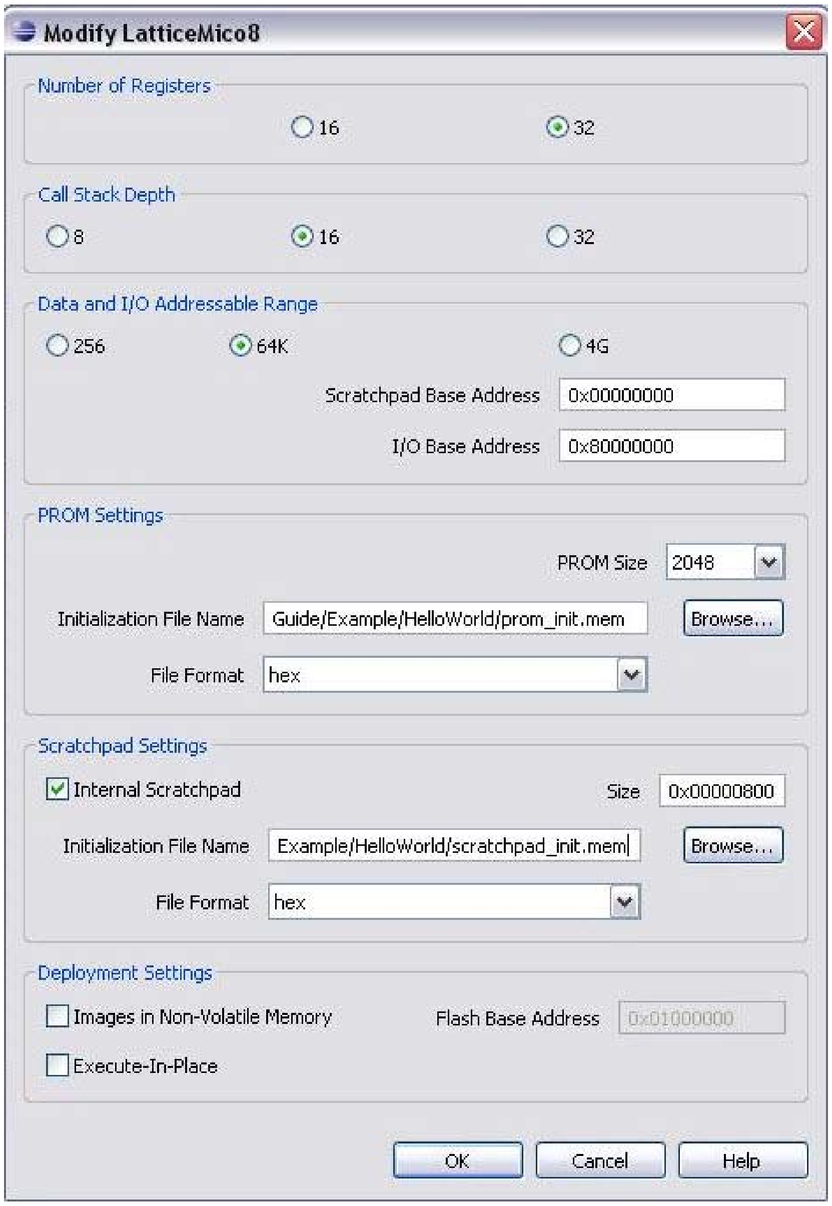
C/C++ Application: Release\HelloWorld.elf

Save Memory Initialization Files to directory... C:\Projects\XO2Components\Documentation\SWGuide\Example\HelloWorld

Click "Start" to generate deployment file

The LatticeMico8 PROM and Scratchpad can now be instantiated with these initialization files. This step is performed within MSB by launching the LatticeMico8 GUI of the microcontroller instance within the hardware platform. Once the PROM and Scratchpad have been initialized, as shown in Figure 25, the hardware platform must be regenerated prior to generating the platform bitstream in Diamond.

Figure 25: PROM and Scratchpad Initialized


Modify LatticeMico8

Number of Registers: 16 [✓ 32]

Call Stack Depth: 8 [✓ 16] 32

Data and I/O Addressable Range: 256 [✓ 64K] 4G

Scratchpad Base Address: 0x00000000

I/O Base Address: 0x80000000

PROM Settings

PROM Size: 2048

Initialization File Name: Guide/Example/HelloWorld/prom_init.mem

File Format: hex

Scratchpad Settings

[✓ Internal Scratchpad]

Size: 0x00000800

Initialization File Name: Example/HelloWorld/scratchpad_init.mem

File Format: hex

Deployment Settings

Images in Non-Volatile Memory: [ ]

Flash Base Address: 0x01000000

Execute-In-Place: [ ]

Non-Volatile Memory Deployment

The initialization files for the LatticeMico8 PROM and Scratchpad can also be located within non-volatile memories such as SPI flash. When LatticeMico8 is configured with this option, the LatticeMico8 microcontroller will automatically fetch these images and initialize the PROM and Scratchpad at power-up. Deploying the application to flash memory involves the following steps:

- Configure the microcontroller in MSB to load the PROM and Scratchpad images from non-volatile memory.

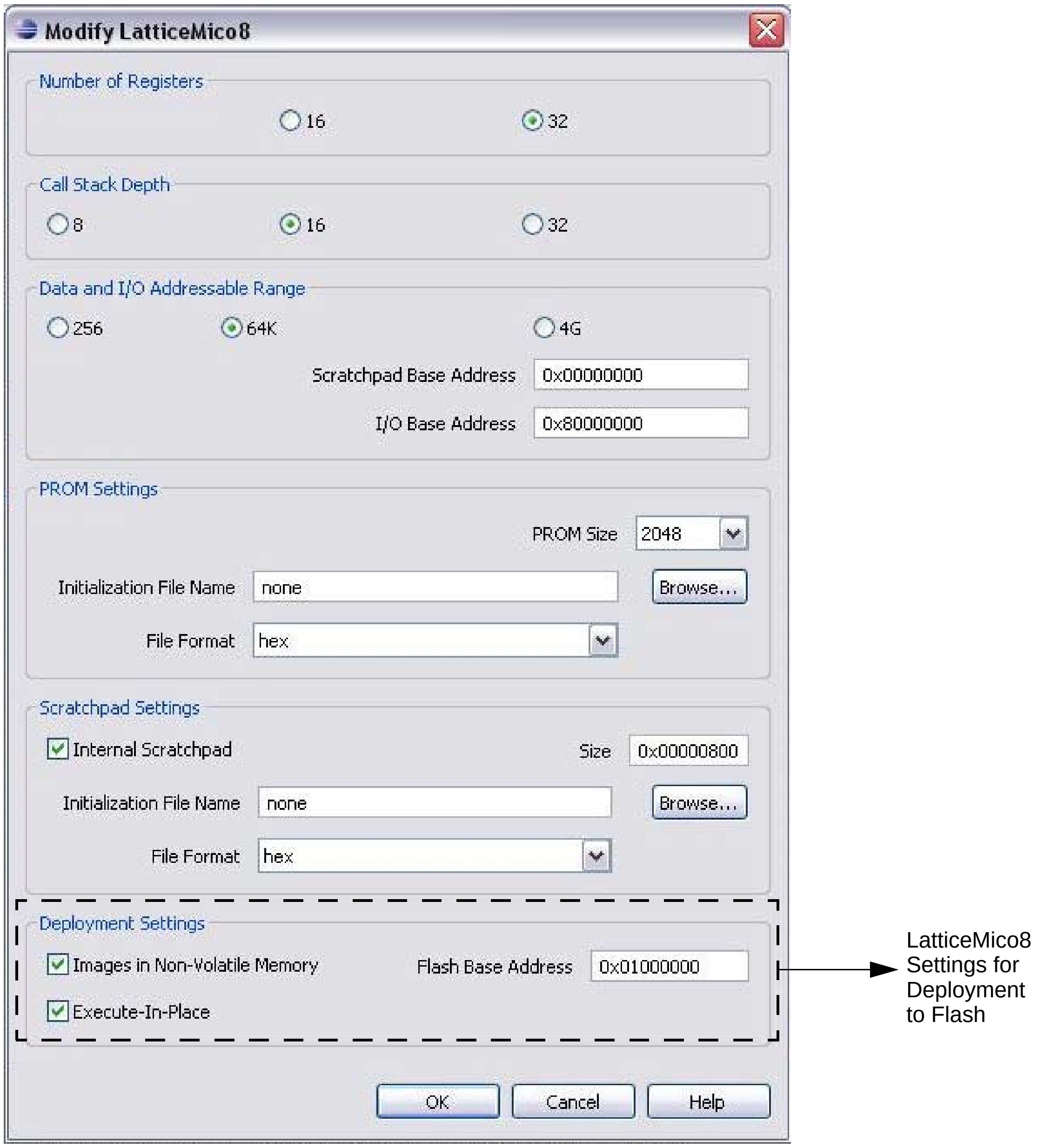
- Create the software image to be programmed to the non-volatile memory.
- Program the software image from Step 2 to non-volatile memory.

Configuring LatticeMico8 to Load from Flash

The LatticeMico8 microcontroller must be configured to load the images for the PROM and Scratchpad from flash. Figure 26 shows the items in the LatticeMico8 GUI that are configured:

- Images in Non-Volatile Memory – Specifies that the PROM and Scratchpad images are located in flash.
- Flash Base Address – Specifies the address in flash at which the images are located. The developer must ensure that the SPI flash is instantiated within the hardware platform at this address.
- Execute-In-Place – Specifies that the PROM itself is located in the flash and the code will execute out of flash.

Figure 26: LatticeMico8 PROM Initialization File



Number of Registers: 16 [✓ 32]

Call Stack Depth: 8 [✓ 16] 32

Data and I/O Addressable Range: 256 [✓ 64K] 4G

Scratchpad Base Address: 0x00000000

I/O Base Address: 0x80000000

PROM Settings

PROM Size: 2048

Initialization File Name: none

File Format: hex

Scratchpad Settings

[✓ Internal Scratchpad]

Size: 0x00000800

Initialization File Name: none

File Format: hex

Deployment Settings

[✓ Images in Non-Volatile Memory]

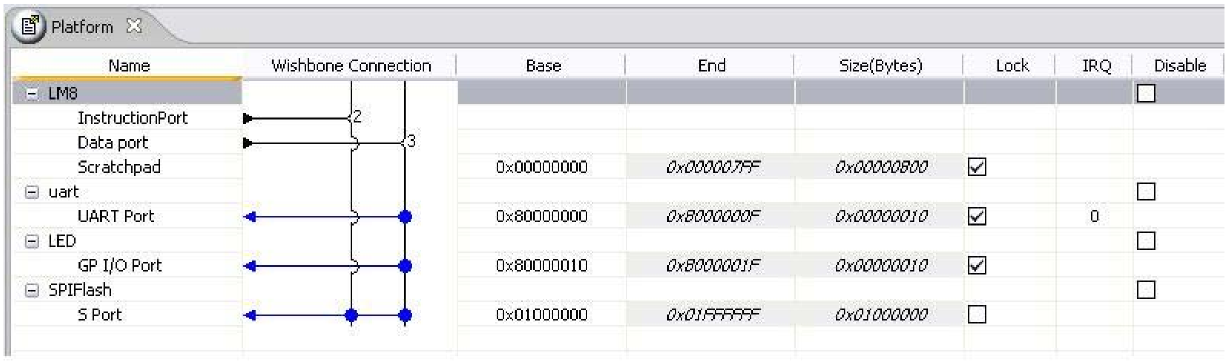
Flash Base Address: 0x01000000

[✓ Execute-In-Place]

Adding SPI Flash Component to Hardware Platform

The hardware platform must contain the LatticeMico SPI flash component which will contain the images of the LatticeMico8 PROM and Scratchpad. Figure 27 shows the platform with a SPI flash component. It is, as shown in the Figure 27, located at the “Flash Base Address” entered in the LatticeMico8 component GUI.

Figure 27: Platform with SPI Flash Component



Platform

Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disable
LM8							☐
InstructionPort	2						
Data port	3						
Scratchpad		0x00000000	0x000007FF	0x00000800	☑		
uart							☐
UART Port		0x80000000	0x8000000F	0x00000010	☑	0	
LED							☐
GP I/O Port		0x80000010	0x8000001F	0x00000010	☑		
SPIFlash							☐
S Port		0x01000000	0x01FFFFFF	0x01000000	☐		

Creating the Application Binary Image

Once the application executable is created, it is converted to a binary format. To generate the initialization file:

1. In the C/C++ perspective, launch the **Software Deployment** GUI from the **Tools** pull-down menu. This brings up the GUI shown in Figure 28.

2. Select the ‘Mico8 Memory Deployment’ tab and then press the ‘New’ button to create a new configuration. Figure 28 shows this new configuration. The developer must specify the following items within the configuration

   - Name – This item refers to the name by which the configuration will be saved so that you can reuse it the next time you launch Mico8 Memory Deployment.

   - Project – Specifies the C/C++ SPE project to use for selecting an application to deploy. Click the **Browse** button for a list of available selections.

   - C/C++ Application – Specifies the application (.elf file) to be deployed in the selected project. Click the **Browse** button for a list of available applications in the selected project, or click the **Search Project** button to select an application (.elf file).

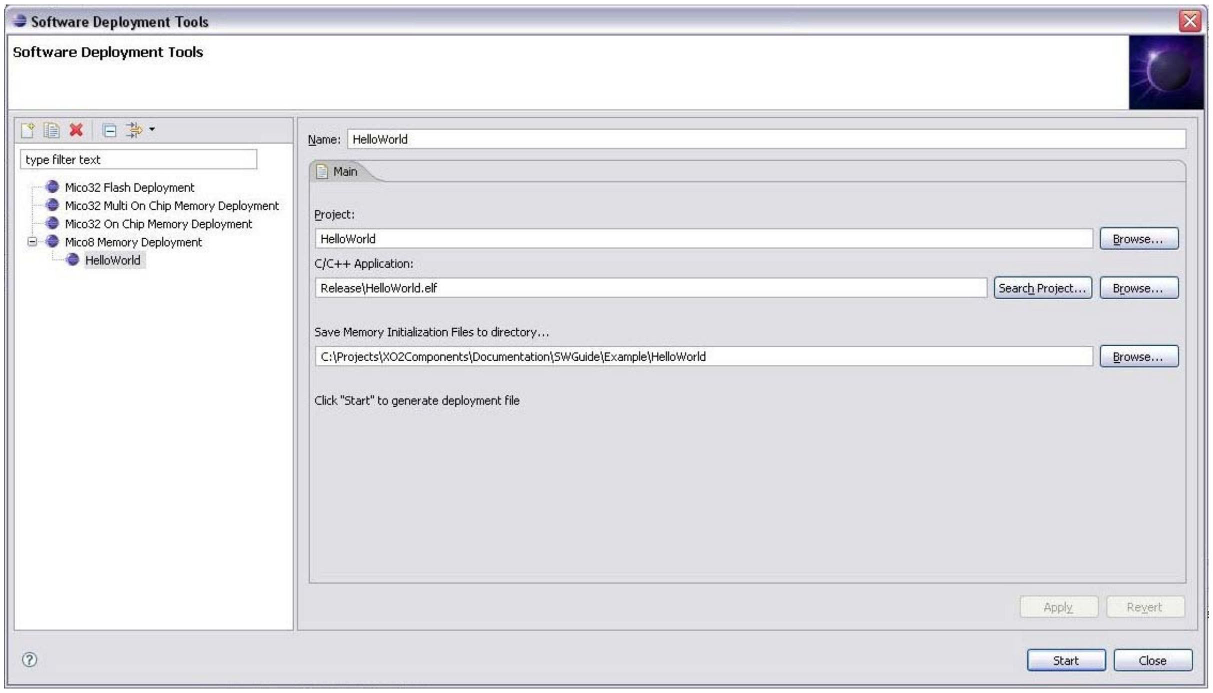
   - Save Memory Initialization Files – The software developer must specify the folder within which the initialization file will be saved.

   Figure 28 shows the values that are entered within the four aforementioned items for the HelloWorld software project.

3. Save the configuration by clicking on Apply.

4. Generate the initialization file “flash_init.bin” by clicking on Start.

Figure 28: Values Entered for HelloWorld Software Project



Name: HelloWorld

Project: HelloWorld

C/C++ Application: Release\HelloWorld.elf

Save Memory Initialization Files to directory...

C:\Projects\XO2Components\Documentation\SWGuide\Example\HelloWorld

Click "Start" to generate deployment file

Programming the Application Binary Image to SPI Flash Using Deployment Tool

Once the .bit file containing the application image is ready, it is programmed to the SPI flash by creating a .mcs file using a tool named Deployment Tool. For detailed information on this tool, refer to the Deployment Tool online Help. The procedure is below:

1. Launch Deployment Tool as follows:

   - In Windows choose **Programs > Lattice Diamond <*version number*> > Accessories > Deployment Tool**.
   - In Linux, enter the following on a command line:

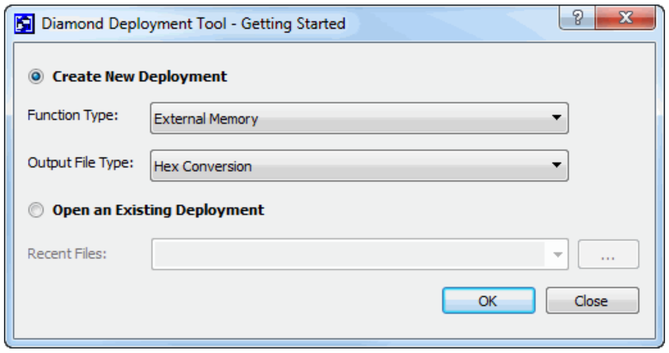
     ```
     <Programmer install path>/bin/lin/./deployment
     ```

   The Deployment Tool Getting Started dialog box appears.

2. In the Function Type dropdown menu, choose **External Memory**.

In the Ouput File Type dropdown menu, choose **Hex Conversion**, as shown in Figure 29.

Figure 29: Deployment Tool Getting Started Dialog Box


Diamond Deployment Tool - Getting Started

(●) Create New Deployment

Function Type: External Memory

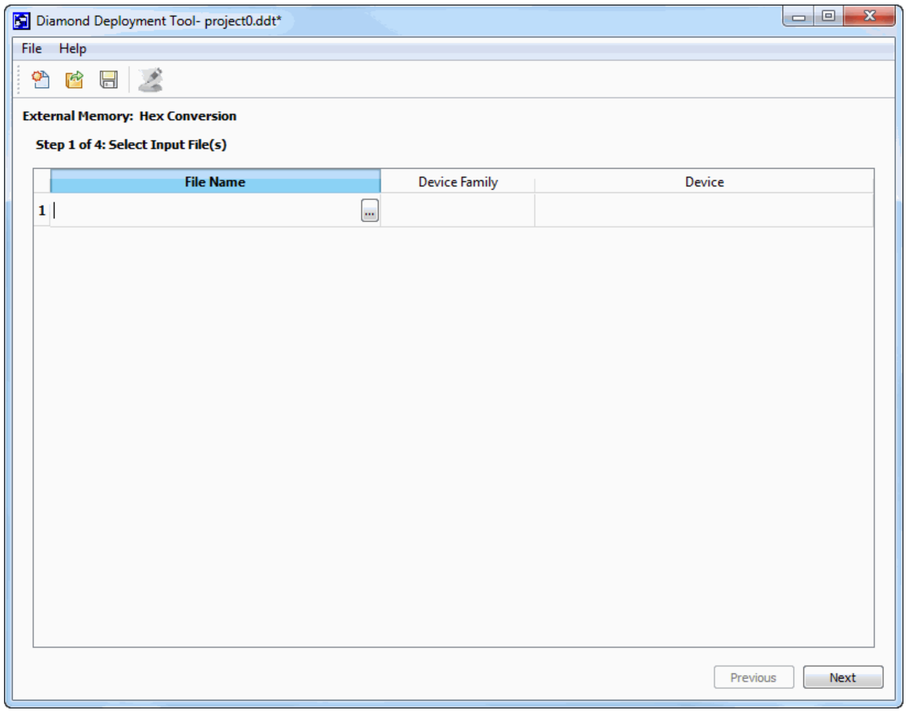
Output File Type: Hex Conversion

( ) Open an Existing Deployment

Recent Files:

3. Click **OK** to display the Step 1 of 4: Select Input File(s) dialog box, as shown in Figure 30.

Figure 30: Step 1 of 4: Select Input File(s) Dialog Box


Diamond Deployment Tool- project0.ddt*

External Memory: Hex Conversion

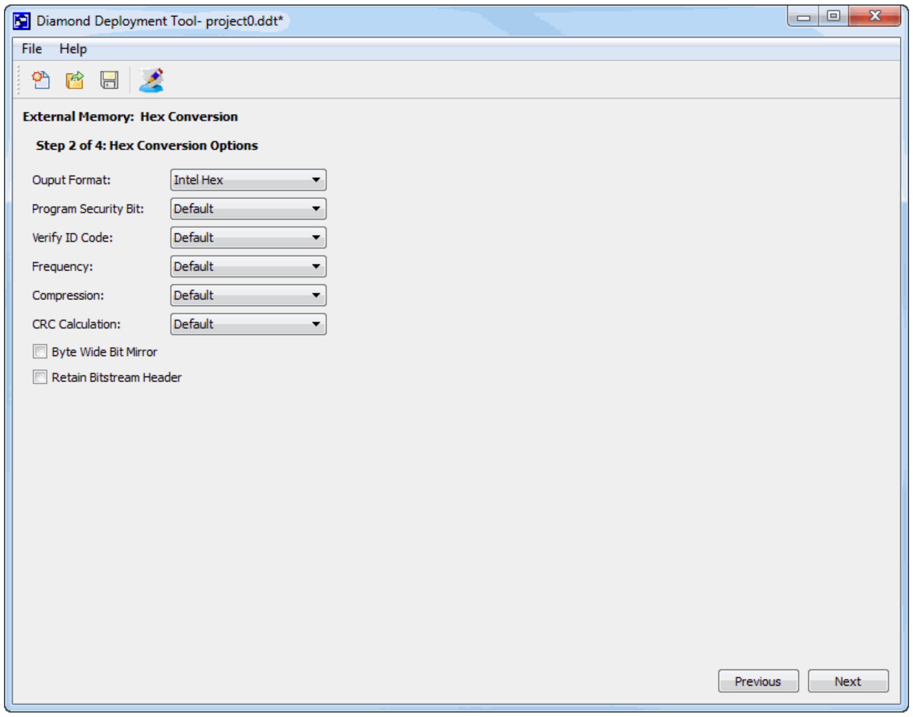
Step 1 of 4: Select Input File(s)

	File Name	Device Family	Device
1			

4. Double-click the File Name box and browse to the “flash_init.bit” file.

5. Click **Next** to display the Step 2 of 4: Hex Conversion Options dialog box, as shown in Figure 31.

Figure 31: Step 2 of 4: Hex Conversion Options Dialog Box



External Memory: Hex Conversion

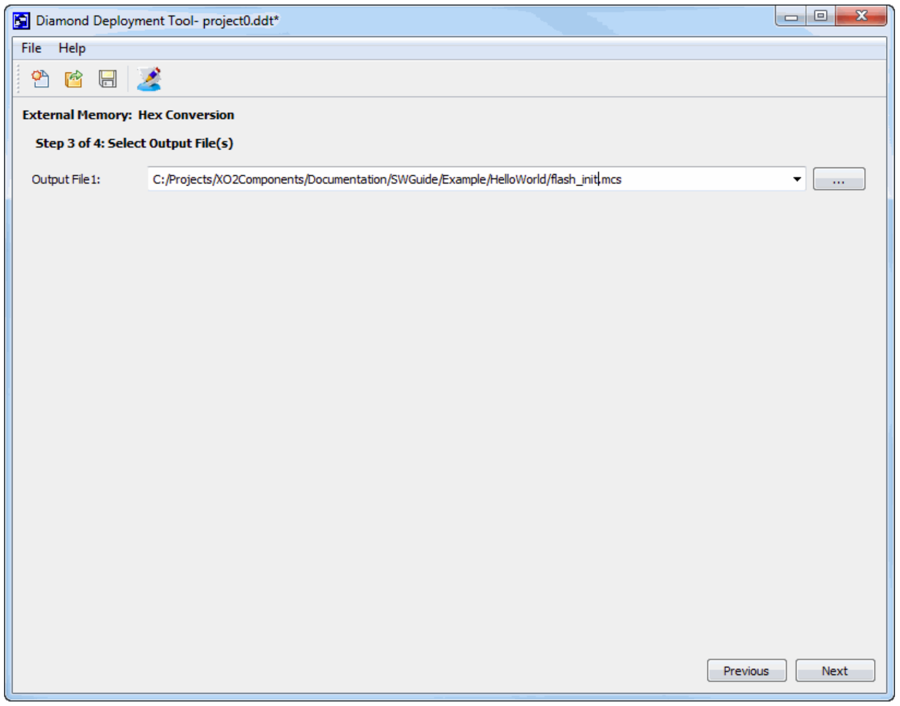
Step 2 of 4: Hex Conversion Options

Ouput Format: Intel Hex
Program Security Bit: Default
Verify ID Code: Default
Frequency: Default
Compression: Default
CRC Calculation: Default
☐ Byte Wide Bit Mirror
☐ Retain Bitstream Header

6. In Output Format dropdown menu, select **Intel Hex**. Leave all other options as default.

7. Click **Next** to display the Step 3 of 4: Hex Conversion dialog box, as shown in Figure 32.

Figure 32: Step 3 of 4: Hex Conversion Dialog Box



External Memory: Hex Conversion

Step 3 of 4: Select Output File(s)

Output File 1: C:/Projects/XO2Components/Documentation/SWGuide/Example/HelloWorld/flash_init.mcs

8. Provide the name of the output data file as “flash_init.mcs”. Then click on **Generate** to generate the .mcs file.

The“flash_init.mcs” file can be programmed to the SPI flash via Diamond Programmer.

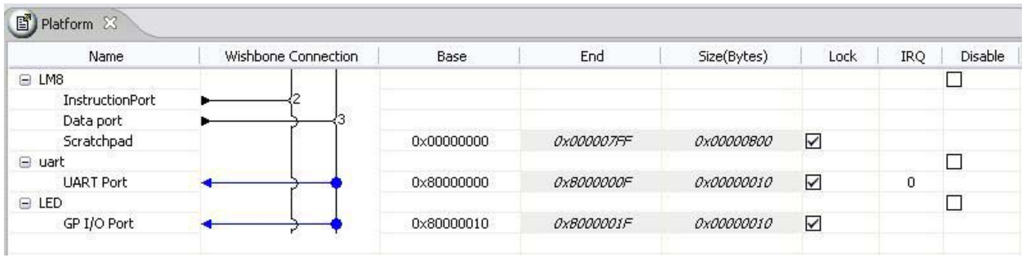
Performing HDL Functional Simulation of LatticeMico8

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 firmware (C code) is compiled using the Lattice C/C++ SPE software, and memory initialization files are created for the LatticeMico8 PROM and Scratchpad. It is possible to locate the firmware in other off-chip memories as long as there exists a behavioral model for the memory. The platform in Figure 33 shows a Verilog design that will be simulated.

:

Figure 33: Platform Setup


Platform

Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disable
LM8							☐
InstructionPort	2						
Data port	3						
Scratchpad		0x00000000	0x000007FF	0x00000800	✓		
uart							☐
UART Port		0x80000000	0x8000000F	0x00000010	✓	0	
LED							☐
GP I/O Port		0x80000010	0x8000001F	0x00000010	✓		

Configuring the Platform with LatticeMico System Builder

The LM8 microcontroller instance in the platform, shown in Figure 33, must be configured to permit functional simulation of software applications through any HDL simulator. The following steps are required:

1. Ensure that the PROM initialization file is provided.
2. Ensure that the Scratchpad is internal and the initialization file is provided.
3. Ensure that deployment to flash is turned off..

Directory Structure

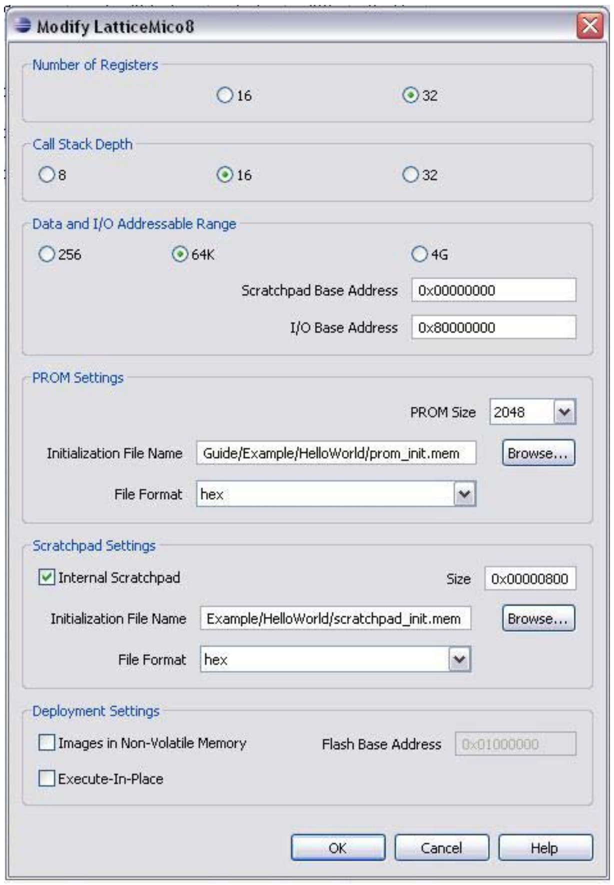
When MSB is used to generate a platform, a set of directories is created in a top-level platform directory. The top-level directory is automatically assigned the same name as the MSB project name, which is Platform in this example.

```
<path_to_toplevel_directory>/Platform
    components
    soc
```

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

Figure 34: Setup



Modify LatticeMico8

Number of Registers: 16 [✓ 32]

Call Stack Depth: 8 [✓ 16] 32

Data and I/O Addressable Range: 256 [✓ 64K] 4G

Scratchpad Base Address: 0x00000000

I/O Base Address: 0x80000000

PROM Settings

PROM Size: 2048

Initialization File Name: Guide/Example/HelloWorld/prom_init.mem

File Format: hex

Scratchpad Settings

[✓ Internal Scratchpad]

Size: 0x00000800

Initialization File Name: Example/HelloWorld/scratchpad_init.mem

File Format: hex

Deployment Settings

☐ Images in Non-Volatile Memory

Flash Base Address: 0x01000000

☐ Execute-In-Place

behavioral models must be provided. See “Replace PMI Black-box Instantiations with Behavioral Models.” on page 66.

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.

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 35, instantiates Platform, the top-level module of the design.

Figure 35: Testbench File

```
`timescale 1 ns / 1 ns

module testbench;

   event done;

   // Inputs
   reg clk_i;
   reg reset_n;

   // Outputs
   wire uartSIN, uartSOUT;
   wire [3:0] LEDPIO_OUT;

   Platform Platform_u
     (
       .clk_i      (clk_i),
       .reset_n    (reset_n),
       .LEDPIO_OUT (LEDPIO_OUT),
       .uartSIN    (uartSIN),
       .uartSOUT   (uartSOUT)
       );

   /*-------------------------------------------------------------------
---
    Clock & Reset
    -------------------------------------------------------------------
--*/
   initial begin
      reset_n = 0;
      #290;
      reset_n = 1;
   end

   initial begin
      clk_i = 0;
      #20;
      forever #(20) clk_i = ~clk_i;
   end

endmodule
```

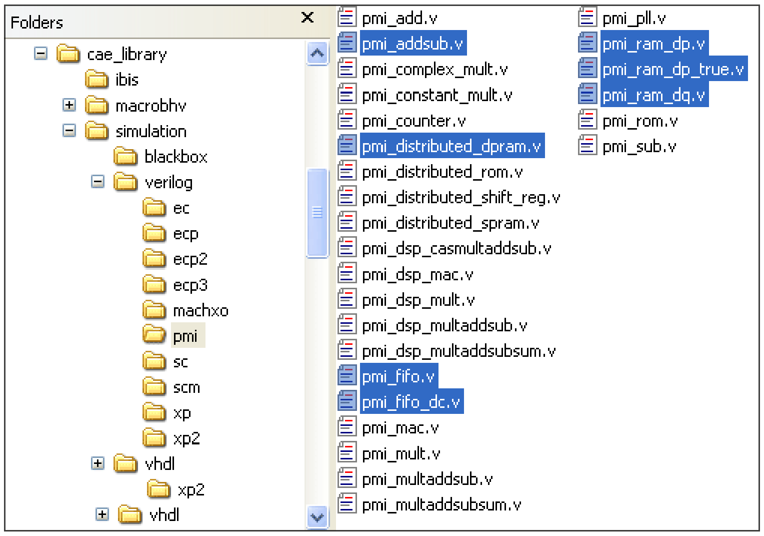
3. **Replace PMI Black-box Instantiations with Behavioral Models**.

   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 36 shows those selected for the Platform example.

Figure 36: Selected PMI Behavior Models from CAE Library



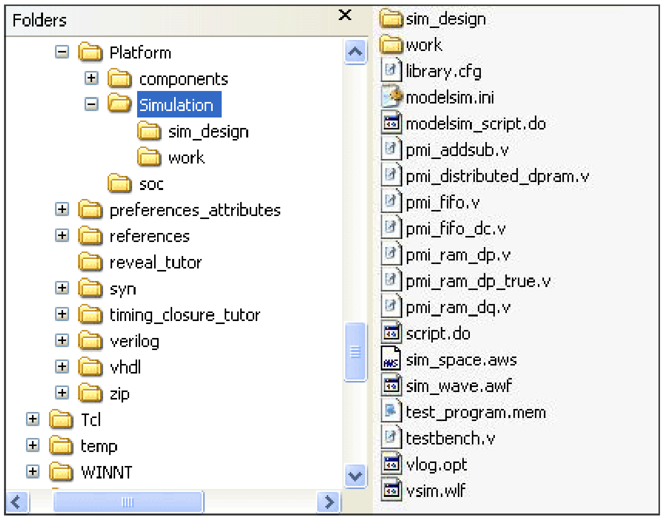
Folders

```
cae_library
  ibis
  macrobhv
  simulation
    blackbox
    verilog
      ec
      ecp
      ecp2
      ecp3
      machxo
      pmi
      sc
      scm
      xp
      xp2
    vhdl
      xp2
  vhdl
```

```
pmi_add.v                      pmi_pll.v
pmi_addsub.v                   pmi_ram_dp.v
pmi_complex_mult.v             pmi_ram_dp_true.v
pmi_constant_mult.v            pmi_ram_dq.v
pmi_counter.v                  pmi_rom.v
pmi_distributed_dpram.v        pmi_sub.v
pmi_distributed_rom.v
pmi_distributed_shift_reg.v
pmi_distributed_spram.v
pmi_dsp_casmultaddsub.v
pmi_dsp_mac.v
pmi_dsp_mult.v
pmi_dsp_multaddsub.v
pmi_dsp_multaddsubsum.v
pmi_fifo.v
pmi_fifo_dc.v
pmi_mac.v
pmi_mult.v
pmi_multaddsub.v
pmi_multaddsubsum.v
```

- Copy the selected models and paste them into the platform's simulation directory. Figure 37 shows the simulation directory of the Platform example where PMI modules have been replaced by the appropriate behavior models.

Figure 37: PMI Models in Platform Simulation Directory



Folders

```
Platform
  components
  Simulation
    sim_design
    work
  soc
preferences_attributes
references
reveal_tutor
syn
timing_closure_tutor
verilog
vhdl
zip
Tcl
temp
WINNT
```

```
sim_design
work
library.cfg
modelsim.ini
modelsim_script.do
pmi_addsub.v
pmi_distributed_dpram.v
pmi_fifo.v
pmi_fifo_dc.v
pmi_ram_dp.v
pmi_ram_dp_true.v
pmi_ram_dq.v
script.do
sim_space.aws
sim_wave.awf
test_program.mem
testbench.v
vlog.opt
vsim.wlf
```

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 LM8 designs.

vlog +define+SIMULATION ../soc/platform.v 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 MachXO2, the behavioral code is located at:
# <isptools>/cae_library/simulation/verilog/machxo2

vsim testbench –L ovi_machxo2
```

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
vmap machxo2_black_boxes C:/Diamond/diamond/1.2/cae_library/
simulation/blackbox/machxo2_black_boxes
vlog -work C:/Diamond/diamond/1.2/cae_library/simulation/
blackbox/machxo2_black_boxes -refresh
vlog -work C:/Diamond/diamond/1.2/cae_library/simulation/
blackbox/machxo2_black_boxes -refresh
vlog +define+SIMULATION \
 +incdir+../soc \
 +incdir+../components/lm8/rtl/verilog \
 +incdir+../components/uart_core/rtl/verilog \
 +incdir+../components/gpio/rtl/verilog \
 +incdir+./models \
 models/pmi_addsub.v \
 models/pmi_ram_dq.v \
 models/pmi_ram_dp.v \
 models/pmi_ram_dp_true.v \
 models/pmi_distributed_dpram.v \
 models/pmi_distributed_spram.v \
 models/pmi_fifo.v \
 models/pmi_fifo_dc.v \
 models/pmi_pll.v \
 models/mt48lc2m32b2.v \
 C:/Diamond/diamond/1.2/cae_library/simulation/verilog/machxo2/
BB.v \
 ../soc/Platform.v \
 testbench.v
vsim work.testbench -novopt
```

Note

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

LatticeMico8 Run-Time Environment

This chapter describes the run-time environment for LatticeMico8 microcontroller-based designs. It takes you through an example of a simple program.

Build/Compilation Utilities

The C/C++ SPE is built on the GNU GCC compiler tool chain customized for the LatticeMico8 micrcontroller. It contains the standard GNU GCC executable utilities, such as objdump, gcc, and ld. The names of these utilities all contain the “lm8-elf” prefix. For example, the GNU GCC compiler executable customized for LatticeMico8 is called lm8-elf-gcc, and the objdump utility customized for LatticeMico8 is called lm8-elf-objdump. Refer to “Software Development Utilities” on page 137 for more information on compilation and build utilities and valid options for them.

Device Drivers and Services

The LatticeMico System Builder (MSB) generates platforms that allow the LatticeMico8 microcontroller to interact with a wide range of possible devices. A platform can also contain multiple instances of the same device, each being configured with different capabilities and features. These devices have software drivers that provide a mechanism for user software code to interact with the device. The device drivers bundled with LatticeMico8 are not meant for use in a multi-threaded environment. The device-specific software driver information that is used for direct manipulation of the device can be found in the device’s component data sheet available as a part of the LatticeMico documentation set.

Microcontroller-Related Services Available at Run Time

Table 1 lists the available microcontroller-related functions and ‘function-like’ macros that can be used by the user application.

Table 1: Microcontroller-Related Services Available at Run Time

Functional Category	Functions/Macros	Header File

Interrupt Management	mico_status MicoDisableInterrupt (char intNum);
	
	mico_status MicoEnableInterrupt (char intNum);
	
	MICO8_DISABLE_GLOBAL_IRQ ();
	
	MICO8_ENABLE_GLOBAL_IRQ ();
		MicoInterrupts.h

Sleep
 	
Note: These are software loops approximating a delay and do not depend on a hardware timer peripheral.
	void MicoSleepMilliSecs (unsigned int timeInMilliSecs);
	
	void MicoSleepMicroSecs (unsigned int timeInMicroSecs);
		MicoUtils.h

Platform Clock Speed Note; The managed build process based on the platform configuration dynamically identifies this value.	MICO8_CPU_CLOCK_MHZ	DDStructs.h

Device Driver Framework

The LatticeMico8 platform functionality is based on the structure that is defined in the .msb file. In addition to the CPU and primary peripherals, there may also be memory components for code and data storage and some components for input and output control, such as the DMA component or the SPI flash component that must be considered. The flexibility of the LatticeMico System Builder (MSB) tool in LatticeMico System enables you to easily change parameters of these components at the system builder level. As documented in more detail in “Managed Build Process and Directory Structure” on page 91, the .xml file provides a mechanism to automatically extract the relevant information from the platform into the C/C++ SPE for software development.

The Latticemico8 device driver framework provides the following facilities:

- Ability to specify component device driver information as part of the platform build
- Ability to extract instance-specific component information from MSB into a managed build software application

- LatticeMico8 microcontroller interrupt framework
- Prepackaged sample device drivers with easy-to-use APIs for most components

To ensure that software application functionality remains unaffected by any changes to the platform, the MSB software provides ready-made device drivers that interact with these components, using the information that is automatically extracted from the .msb file. These device drivers enable you to control instantiated components without having to know component-specific details, such as register layout. It also basically protects the application from the negative effects of changes like altering a component's base address.

Device Driver APIs

The device driver APIs are device-specific functions. The LatticeMico System Builder (MSB) includes device drivers, customized for LatticeMico8 microcontroller, for the following components:

- RS-232 UART
- GPIO
- DMA
- SPI Flash
- MachXO2 EFB

The APIs directly manipulate these devices, along with their register layout structures, as described in the respective component data sheets. These data sheets also contain component usage examples. The availability of device-driver APIs is platform-dependent. These APIs can be used directly from your application, provided the platform description contains the corresponding components.

Modifying Existing Device Drivers

This section shows you how to override the default behavior of the device drivers.

Overriding Default Driver Initialization Sequence

As will be shown in "Boot Sequence" on page 82, the boot-up sequence invokes function LatticeDDInit, which initializes the components before invoking your main() implementation. If you want to override the default LatticeDDInit implementation, perform the following steps:

1. As part of your application source, create a file named DDInit.c.
2. Within DDInit.c, implement the void LatticeDDInit(void) function.

A sample skeleton of what your DDInit.c file should look like is shown in the code example in Figure 38

These steps override the default implementation of LatticeDDInit, bypassing the LatticeMico8 C/C++ SPE build-process-generated driver initialization routine. You can then dictate your own initialization sequence by placing code in your DDInit.c file. For more information on the DDInit.c file, see "DDInit.c File" on page 107.

Figure 38: Code Example

```
#include "DDStructs.h"

void LatticeDDInit(void)
{

    /* initialize uart instance of uart_core */
    MicoUartInit(&uart_core_uart);

    /* invoke application's main routine*/
    main();
}
```

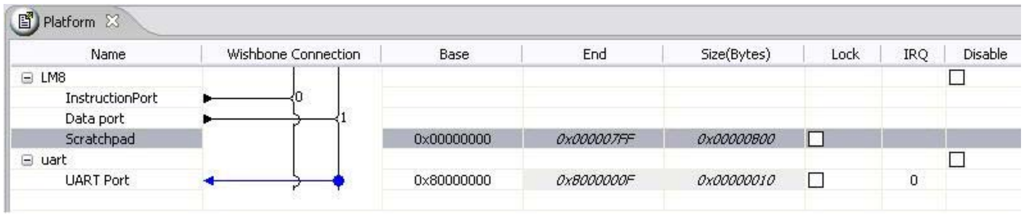
Overriding Default Driver Implementation

You can override the default driver implementation by providing your own source files that match the name of the driver source files that you want to override. You must implement all of the functions in the source file that you want to override. If you do not and if any of the functions you have not rewritten are called by another code module, the compiler will attempt to pull in the source file objects that you attempted to override and generate compiler errors. You can also override the default interrupt management implementation and implement your own scheme that handles nested interrupts. Some library files become part of the application build process instead of the library build process, such as crt0.S. The implementation in these files cannot be overridden as part of the LatticeMico C/C++ SPE managed build process.

Basic Program Structure

This section uses a simple "hello world" program to illustrate the program structure and the behind-the-scene activities of a program. The platform diagram from the MSB Editor view shown in Figure 39 illustrates the example platform structure.

Figure 39: Example Platform Structure



Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disable
⊟ LM8							☐
InstructionPort	0						
Data port	1						
Scratchpad		0x00000000	0x000007FF	0x00000800	☐		
⊟ uart							☐
UART Port		0x80000000	0x8000000F	0x00000010	☐	0	

Note

The procedures presented in this section are not a substitute for the LatticeMico8 tutorial but work together in a task-oriented way to provide a quick way to learn some key points about programming in this environment.

- The example used in this section depends on the following criteria:
- The LatticeMico8 Managed C/C++ build process is used for building the "hello world" application.
- The "Platform" platform consists of the following components:
- UART instance named "uart"
- LatticeMico8 microcontroller instance named "lm8"
- The UART is selected as the standard input, output, or error device. It is configured to use interrupts. See information in Section "Setting Project Properties" for details on how to modify the platform settings.
- The linker settings map the program code and data sections to LatticeMico8's internal PROM and Scratchpad respectively. See LatticeMico8 data sheet for details on how to enable the internal PROM and Scratchpad.

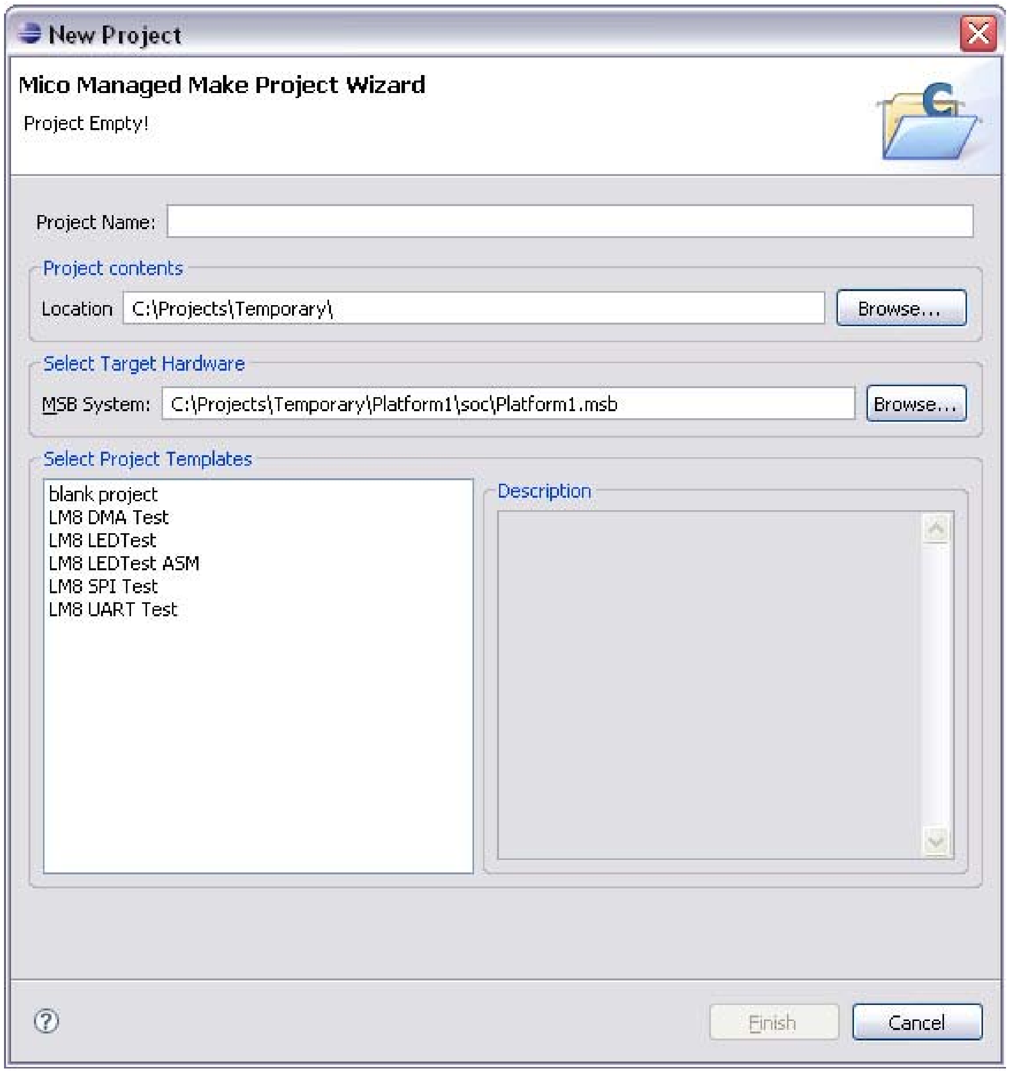
Creating a Blank Project

As the first step, the software developer must create a project based on the platform criteria outlined previously in "Basic Program Structure" on page 74.

To create a blank project in the Project Wizard:

1. In the C/C++ SPE perspective, choose **File > New > mico Managed C Project** to bring up the New Project dialog box.
2. In the Project name text box, enter **HelloWorld**.
3. Select the project contents folder using the Browse button in the Location text box.
4. Select the **Platform** target hardware platform, using the Browse button in the MSB System text box.
5. Select **Blank Project** in the Select Project Templates list box in the lower left portion of the dialog box.The New Project dialog should now resemble the illustration in Figure 40.
6. Click **Finish**.

Figure 40: New Project Dialog Box



New Project

Mico Managed Make Project Wizard

Project Empty!

Project Name:

Project contents

Location C:\Projects\Temporary\

Select Target Hardware

MSB System: C:\Projects\Temporary\Platform1\soc\Platform1.msb

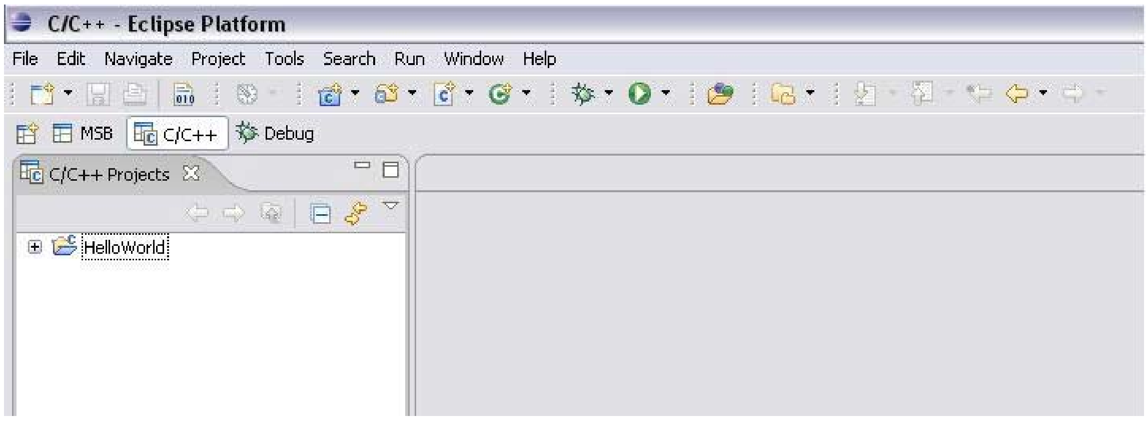
Select Project Templates

blank project
LM8 DMA Test
LM8 LEDTest
LM8 LEDTest ASM
LM8 SPI Test
LM8 UART Test

Description

This newly created project should now be visible in the C/C++ perspective's Projects view, as shown in Figure 4.

Figure 41: C/C++ Perspective Projects View



Adding a Source File to the Project

You will now add a new source file to your newly created project. Source files refer to your source C language files.

To add a source file to your project:

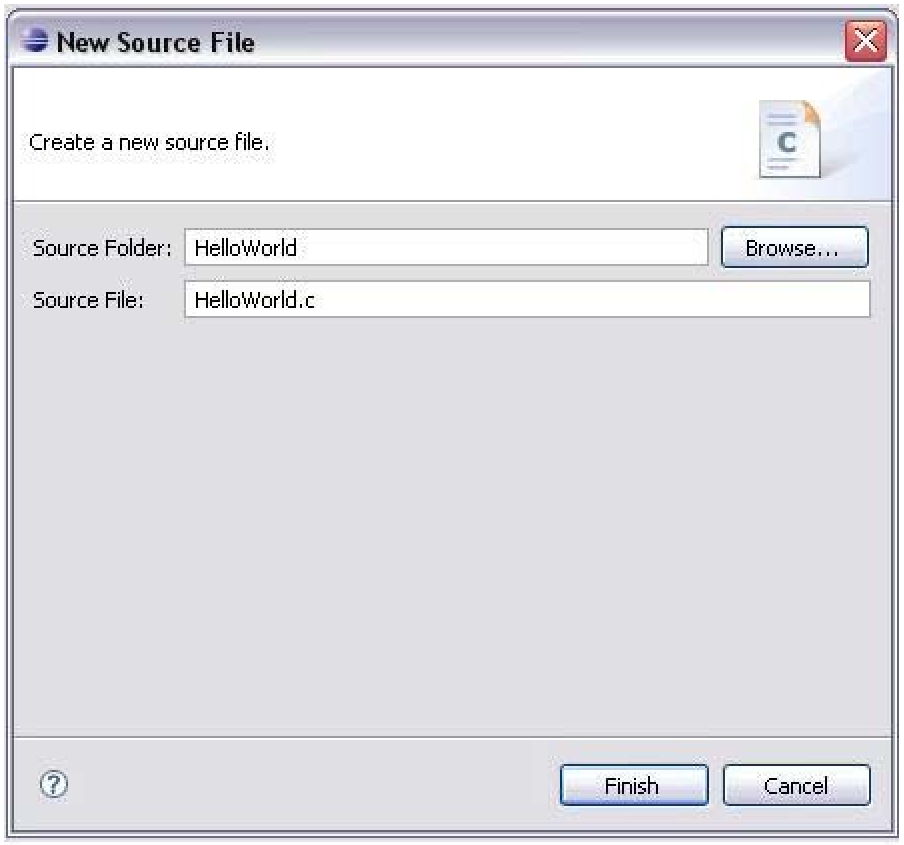
1. In the C/C++ perspective, click on the **HelloWorld** project in the Projects view.
2. In the pop-up menu, choose **File > New > Source File**.

In the New Source File dialog box, shown in Figure 42, enter **HelloWorld.c** in the Source File text box.

Note

The source file can be a C source file, an Assembly-only source file, or a C with inlined-assembly source file. The C and C with inlined-assembly source files must have a .c file extension. The Assembly-only source files must have a .S or .s file extension.

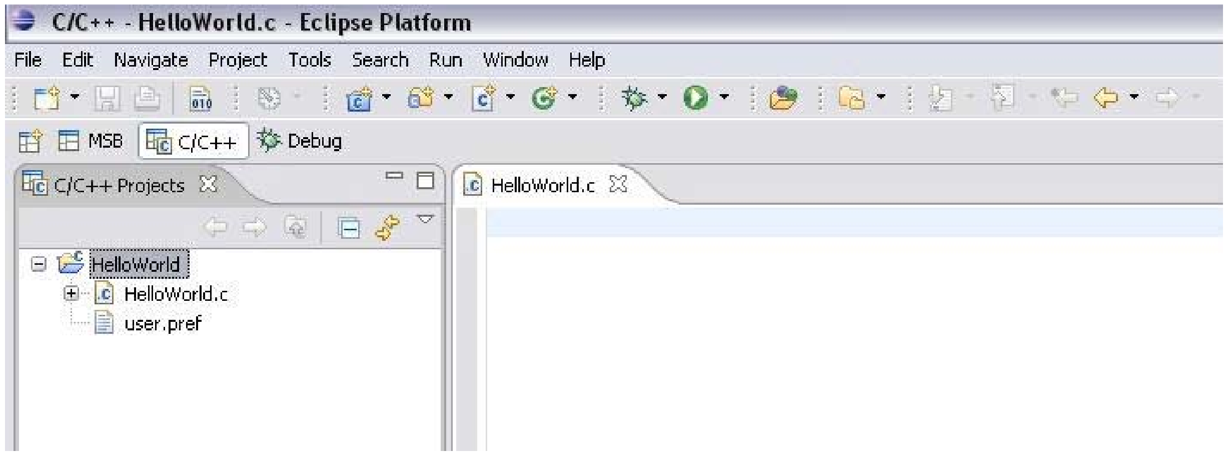
Figure 42: New Source FIle Dialog Box



Source Folder: HelloWorld
Source File: HelloWorld.c

This new file is now visible beneath the project in the C/C++ perspective's Projects view, as shown in Figure 6.

Figure 43: New FIle Visible in C/C++ Perspective Projects View


C/C++ - HelloWorld.c - Eclipse Platform
File Edit Navigate Project Tools Search Run Window Help
MSB
C/C++
Debug
C/C++ Projects
HelloWorld.c
HelloWorld
HelloWorld.c
user.pref

In addition, you may see the user.pref file, which is automatically generated by the C/C++ SPE managed build process and should not be modified or deleted. The user.pref file is described in the "C/C++ Perspective Project Folder File Contents" on page 95.

Adding Source to the Source File

Now you will want to add source to your source .c file. At this point, you are interested in using a generic Hello World application that simply prints “hello world” to the terminal via the UART. To do this, add the code shown in Figure 44 in to the HelloWorld.c source file that you created in the prior step.

Figure 44: HelloWorld .c Source File

```
#include "MicoUtils.h"
#include "DDStructs.h"      ←—— 1
#include "MicoUart.h"


const char *HELLO_STRING = "Hello World\r\n";  ←—— 2

static void SendCharacter(MicoUartCtx_t *pUart, unsigned char c)  ←—— 3
{
  MicoUart_putC(pUart, c);
  return;
}

int main(void)  ←—— 4
{
  MicoUartCtx_t *uart = &uart_core_uart;
  unsigned int idx = 0;

  do {          ←—— 5
    SendCharacter (uart, *HELLO_STRING);
    MicoSleepMilliSecs (1000);   ←—— 6
    HELLO_STRING++;
  } while (idx++ < 14);

  return(0);   ←—— 7
}
```

The lines shown in this code example are described following:

- Item 1 – These three #include statements declare the header files needed to verify the function prototypes of the functions used in the code. The MicoUtils.h value refers to the standard LatticeMico8 header file that contains the prototype declaration of the function listed in item 6. The MicoUart.h value refers to the standard LatticeMico UART header file that contains the prototype declaration of the function listed in item 3. The DDStructs.h value refers to the header file that is automatically generated by the C/C++ SPE for the given hardware platform.
- Item 2 – This is the “hello world” character string that will be transmitted via the LatticeMico UART.
- Item 3 – This is the function that calls the LatticeMico UART function declared in MicoUart.h header file from Item 1 that transmits one character over the UART transmit line.
- Item 4 – The int main(void) parameter is the “main” function that is executed when you execute your program. This is the main entry point of the application code. This “main” does not receive any argument, and it

passes back an integer value that has no significance for the current release. The sequence of code leading to invocation of “main” is described in Section “The int main(void) Function”.

- Item 5 – This is the device context structure associated with the UART instance named “uart” in the hardware platform. The declaration of the context structure associated with every device in the hardware platform is found in DDStructs.h header file from Step 1.
- Item 6 – Since you are using the UART with interrupts enabled (as selected during platform configuration in MSB), you must wait a reasonable amount of time for the interrupt service routine to send all the characters in the “hello world” string. Typically, the UART baud rate is much slower than the CPU speed, so this delay is required. This function is part of the LatticeMico8 platform library, specifically the CPU service, and its prototype is declared in the MicoUtils.h header file.
- Item 7 – Since you are finished with your application, you must pass back control to the calling process. Once you do this, the calling process as described in a subsequent section will terminate. For typical embedded systems, your application would never return control back from your “main.”

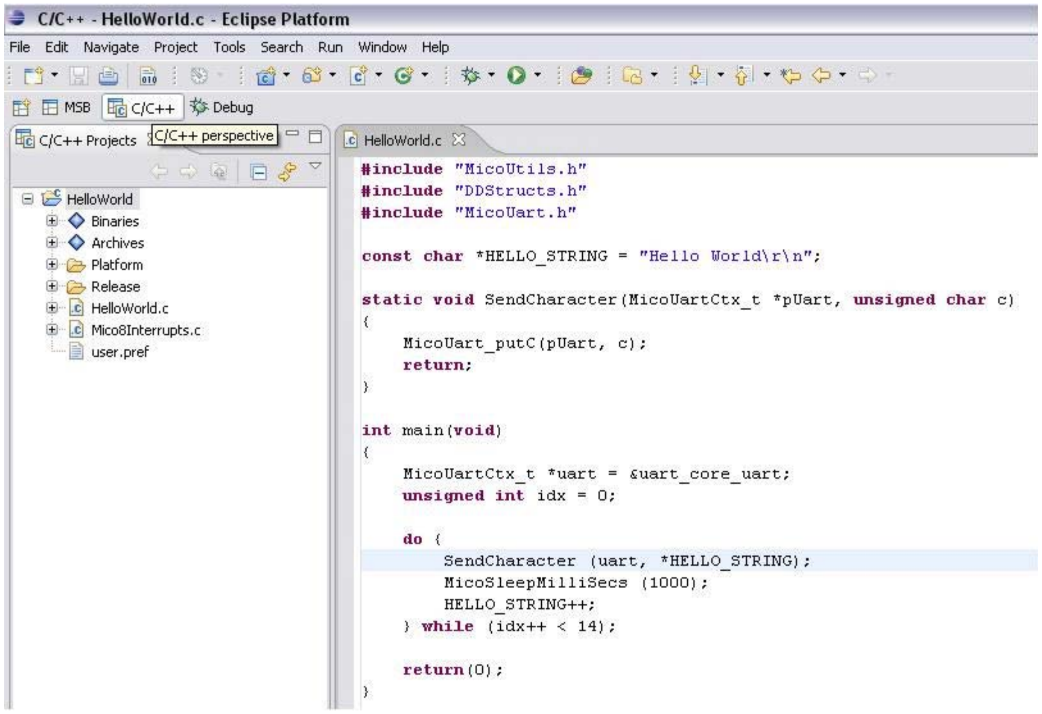
Building the Application

At this point in the example, you are ready to build your application using the C/C++ SPE managed build process.

To build the application:

1. In the C/C++ perspective, right-click on the project folder in the Projects view.
2. In the pop-up menu, choose **Build Project** to initiate the managed build process, as described in “Managed Build Process and Directory Structure” on page 91. The Projects view in the C/C++ perspective is updated to show the generated artifacts, as shown in Figure 8.

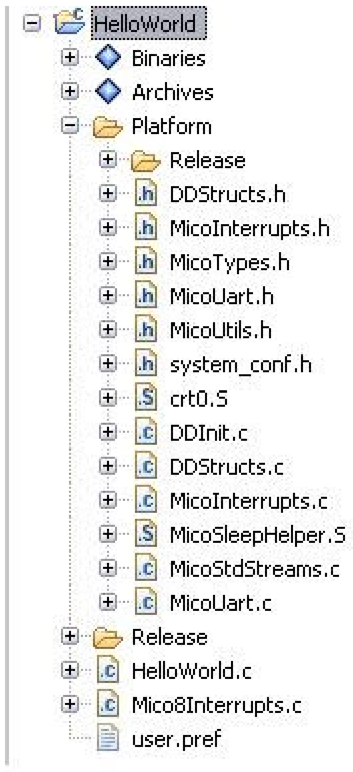
Figure 45: HelloWorld Shown in C/C++ Perspective Projects View



```
#include "MicoUtils.h"
#include "DDStructs.h"
#include "MicoUart.h"

const char *HELLO_STRING = "Hello World\r\n";

static void SendCharacter(MicoUartCtx_t *pUart, unsigned char c)
{
    MicoUart_putC(pUart, c);
    return;
}

int main(void)
{
    MicoUartCtx_t *uart = &uart_core_uart;
    unsigned int idx = 0;

    do {
        SendCharacter (uart, *HELLO_STRING);
        MicoSleepMilliSecs (1000);
        HELLO_STRING++;
    } while (idx++ < 14);

    return(0);
}
```

Figure 9 shows the contents of the platform library for the example. “Platform Library-Generated Source Files” on page 101 describes the various items within the Platform Library folder.

Figure 46: Platform Library Folder



Boot Sequence

An assembly language file named crt0.S in the platform library folder contents contains the boot-up sequence. This section generically describes the boot-up sequence, as well as the layout of the boot section. This section assumes that you are familiar with the LatticeMico8 microcontroller architecture described in the LatticeMico8 Processor Reference Manual. The code in crt0.S is shown in the example sections in Figure 10.

Figure 47: Contents of ctr0.s

```
  .section .vectors,"ax"
  .weak__irq_save_restore2
  b __irq_save_restore2

.globl _start
_start:
  /* Clear bss */
  movir0,_lo(__bss_start)
#if defined(__CMODEL_LARGE__) || defined(__CMODEL_MEDIUM__)
  movir13,_hi(__bss_start)
#endif
#ifdef __CMODEL_LARGE__
  movir14,_higher(__bss_start)
  movir15,_highest(__bss_start)
#endif
  movir1,0

3:
  cmpir0,_lo(__bss_end)
#if defined(__CMODEL_LARGE__) || defined(__CMODEL_MEDIUM__)
  bnz1f
  cmpir13,_hi(__bss_end)
#endif
#ifdef __CMODEL_LARGE__
  bnz1f
  cmpir14,_higher(__bss_end)
  cmpir15,_highest(__bss_end)
#endif
  bz2f
1:sspir1,r0
  addir0,_lo(1)
#if defined(__CMODEL_LARGE__) || defined(__CMODEL_MEDIUM__)
  addicr13,_hi(1)
#endif
#ifdef __CMODEL_LARGE__
  addicr14,_higher(1)
  addicr15,_highest(1)
#endif
  b 3b

2:/* Setup the stack */
#if defined (__CMODEL_SMALL__)
  movir14,__stack

  /* Mark the end-of-stack */
  movir15,0
#elif defined(__CMODEL_MEDIUM__)
  movir8,_lo(__stack)
  movir9,_hi(__stack)
  /* Mark the end-of-stack */
  movir10,_lo(0)
  movir11,_hi(0)
#elif defined(__CMODEL_LARGE__)
  /* -4 because main(int, char **, char **) and that third 
argument
   * is passed on the stack, otherwise the compile may access 
things
   * past the end of the stack space. */
  movir24,_lo(__stack-4)
  movir25,_hi(__stack-4)
  movir26,_higher(__stack-4)
  movir27,_highest(__stack-4)

  /* Mark the end-of-stack */
  movir28,_lo(0)
  movir29,_hi(0)
  movir30,_higher(0)
  movir31,_highest(0)
#endif

  seti
  callLatticeDDInit
  clri

  /* Kill the simulation */
  movir31,0xde
  movir30,0xad
  movir29,0xbe
  movir28,0xef
  movr27,r0

1:b 1b

__irq_save_restore2:
  iret
```

Reset Address

From a software boot perspective, the most important parameter in LatticeMico8 configuration is the location of the program memory (PROM) since it determines the Reset Address. This 32-bit address value dictates the address at which execution begins after power-up. When the PROM is internal to the LatticeMico8 instance, the address value defaults to 0x00000000 and the Reset Address is 0x00000003. When the PROM is external to the LatticeMico8 instance, the Reset Address is 3 byte locations from the starting address of the external PROM. The Reset Address is automatically calculated upon platform generation.

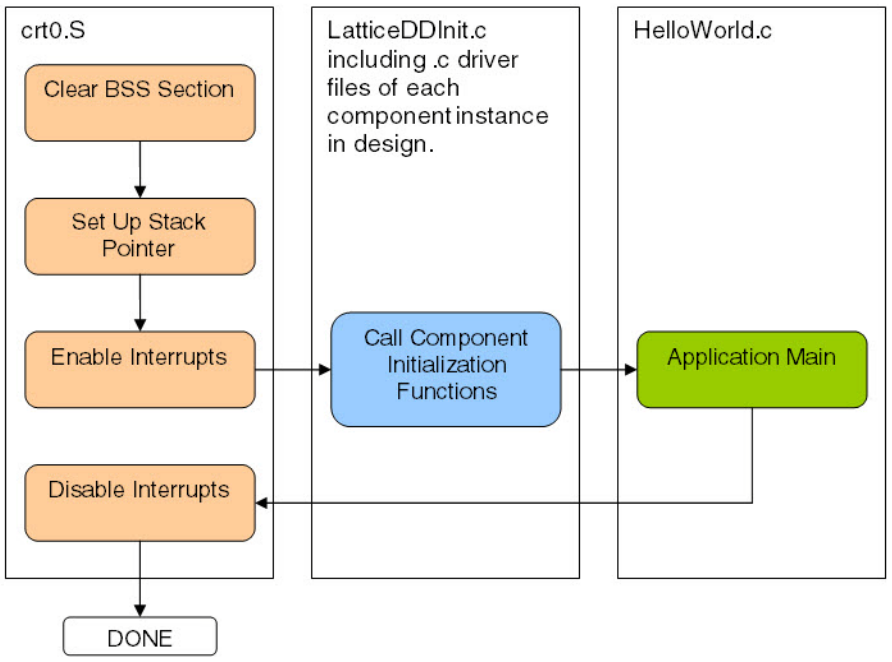
Note

The C/C++ SPE locates the boot-up sequence within the assembly language file crt0.S at the location of the PROM.

Boot Code Sequence Flow

This section provides an overview of the boot sequence in file crt0.S and its steps. Figure 48 illustrates this boot sequence.

Figure 48: Boot Code Sequence Flow


crt0.S
Clear BSS Section
Set Up Stack Pointer
Enable Interrupts
Disable Interrupts
DONE
LatticeDDInit.c including .c driver files of each component instance in design.
Call Component Initialization Functions
HelloWorld.c
Application Main

The primary actions performed during boot-up are:

- crt0.S
  - Clear .bss section of the application’s data space.
  - Initialize stack pointer to top of Scratchpad memory that is specified in the linker settings.

- Enable all interrupts
- Call LatticeDDInit()
- LatticeDDInit.c
    - Call all component initialization functions. The initialization function for each component is located within its device driver source file. The initialization function of a component is invoked for each instantiation of the component within the platform.
    - Call main()
- Application Main
    - Execute the user’s application code

Interrupt Handling Sequence

A C source file named Mico8Interrupts.c under the software project folder contains the LatticeMico8 interrupt handler. In addition, the LatticeMico8 GNU GCC compiler automatically generates assembly language code that is responsible for handing over control of the LatticeMico8 execution to the aforementioned interrupt handler after a hardware interrupt is detected. This section provides an overview of this interrupt handling sequence.

Interrupt Handlers

Hardware interrupts cause an interruption of normal application execution. LatticeMico8 supports a maximum of eight dedicated hardware interrupts. Each component instance within the platform can be tied to only one of these eight dedicated hardware interrupt lines. Some of the components implement a Lattice-provided generic interrupt handler function. The customer can either choose the generic implementation or customize it by implementing the function within his software code. When an interrupt is raised, control is transferred to a function __irq_save_restore that is automatically generated by the LatticeMico8 GNU compiler as part of the application executable. This function saves the current microcontroller state, sets up the microcontroller for interrupts, and then transfers control to the global interrupt handler. This global interrupt handler is implemented in the C source file Mico8Interrupts.c shown in Figure 50. This file is automatically generated and populated by the C/C++ SPE as part of the managed build process.

Figure 49: Contents of Mico8Interrupts.c

```
#include "DDStructs.h"
#ifndef __MICO_NO_INTERRUPTS__
#include "MicoInterrupts.h"              ←  1
#include "MicoUart.h"

void MicoISRHandler();


void __IRQ(void) __attribute__ ((interrupt));

void __IRQ(void)                                    ←  2
{
  MicoISRHandler();
}


void MicoISRHandler()          ←  3
{

  unsigned char ip, im, Mask, IntLevel;
  do {
    MICO8_READ_IM(im);
    MICO8_READ_IP(ip);

    ip &= im;
    Mask = 0x1;
    IntLevel = 0x0;

    if ( ip!=0 ) {
        do {
        if (Mask & ip) {
            switch(IntLevel) {
              case 0:
                MicoUartISR(&uart_core_uart);
                break;
              default:
                break;
          }
          MICO8_PROGRAM_IP(Mask);
            break;
          }
          Mask <<= 0x1;
          ++IntLevel;
        } while (1);
    } else {
            break;
    }
  } while (1);

  return;
 }
#endif
```

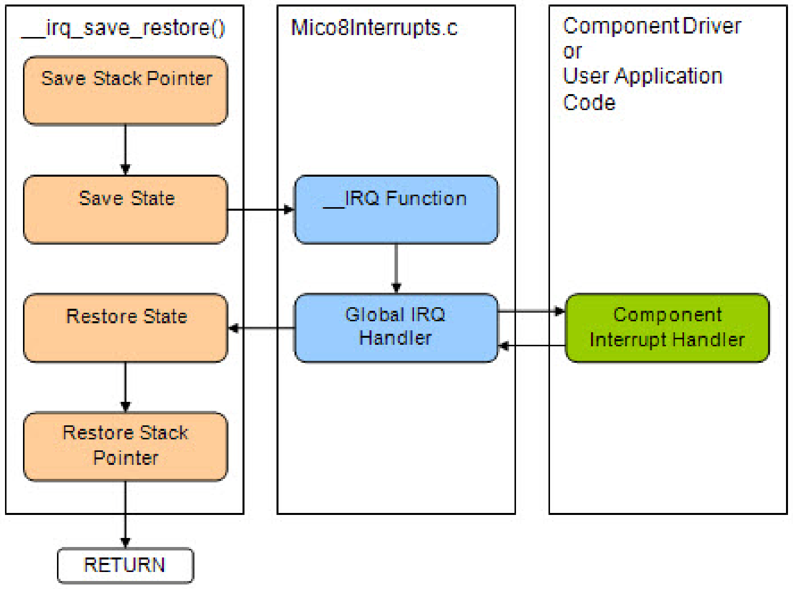
This C source file is automatically generated by the C/C++ SPE for every software project. The lines in the C source file are described below:

- Item 1 – These three #include statements declare the header files needed to verify the function prototypes of the functions used in the code. The DDStructs.h value refers to the header file that is automatically generated by the C/C++ SPE for the given hardware platform and contains each component instance's context structure declaration. The value MicoInterrupts.h refers to the LatticeMico8 header file that contains the prototype and macro declarations for enabling/disabling interrupts within LatticeMico8. The value MicoUart.h refers to the standard LatticeMico UART header file that contains the prototype declaration of the function listed in item 3.
- Item 2 – The LatticeMico8 GNU compiler only recognizes this function as the interrupt handler. When this function is implemented within the software project, the compiler will automatically do the following tasks:
  - Generate code to save the program Stack Pointer
  - Generate code to save the current microcontroller state (e.g., registers)
  - Generate code to set up the stack pointer at the base of the Interrupt Stack and then jump to function __IRQ(void)
  - Generate code to restore the current microcontroller state upon return from the function __IRQ(void)
  - Generate code to restore the program Stack Pointer upon return from the function __IRQ(void)
- Item 3 – This is LatticeMico8 microcontroller's global interrupt handler. This function services each pending hardware interrupt by calling its interrupt handler. The interrupt handler function names are obtained from the hardware platform's MSB file.

Interrupt Handling Sequence Flow

This section describes the interrupt handling sequence. Figure 50 illustrates the sequence of events that occur once an interrupt has been received by the LatticeMico8 microcontroller.

Figure 50: Interrupt Handling Sequence Flow



Managed Build Process and Directory Structure

The managed build process uses input user application code files and files associated with the targeted platform to build an output executable for that platform. It follows a specific directory structure for managing the different types of files. It automatically generates certain application source files that are specific to the platform and to its target application.

This chapter focuses on the process steps, file inputs, file outputs, and directory structure associated with the managed build flow and the installation of the LatticeMico System software. Besides giving you insights into how the managed build process works, this information is required if you wish to add any user-defined components to your platform using Mico System Builder (MSB).

Creating Managed Build Applications

The LatticeMico8 C/C++ managed build process provides a framework for developing software applications targeting a LatticeMico8 microcontroller. The build process examines the platform definition that is specified in the .msb file generated by Mico System Builder (MSB) and extracts component-specific device-driver information, if present, in addition to memory information specified for generating appropriate linker scripts. It uses this information to automatically generate platform-specific source code for platform initialization.

This framework also generates the necessary makefiles for building the system's platform library, which consists of startup and helper routines for the microprocessor, as well as specified components, and the makefiles needed for building the application.

The LatticeMico C/C++ managed build environment does the following:

- Extracts device driver information from instantiated components

- Creates a device-driver structures header (DDStructs.h) file and component instance-specific device-driver structure instances based on that header file in the device-driver structures source (DDStructs.c) file. The DDStructs.c file creates information about the components that are in the .msb file available to the C application and driver code. It also generates a device-driver initialization source (DDInit.c) file that contains the initialization sequence.
- Creates the LatticeMico8 interrupt handler source file (Mico8Interrupts.c) that contains the global interrupt handler which invokes the interrupt handlers for individual components.
- Creates and manages required makefiles
- Creates a default linker script by identifying memory components in the platform

LatticeMico8 C/C++ Project Build Flow

This section outlines the steps in the managed build process and describes the directory structure and the relevant contents of the generated folders created by the build.

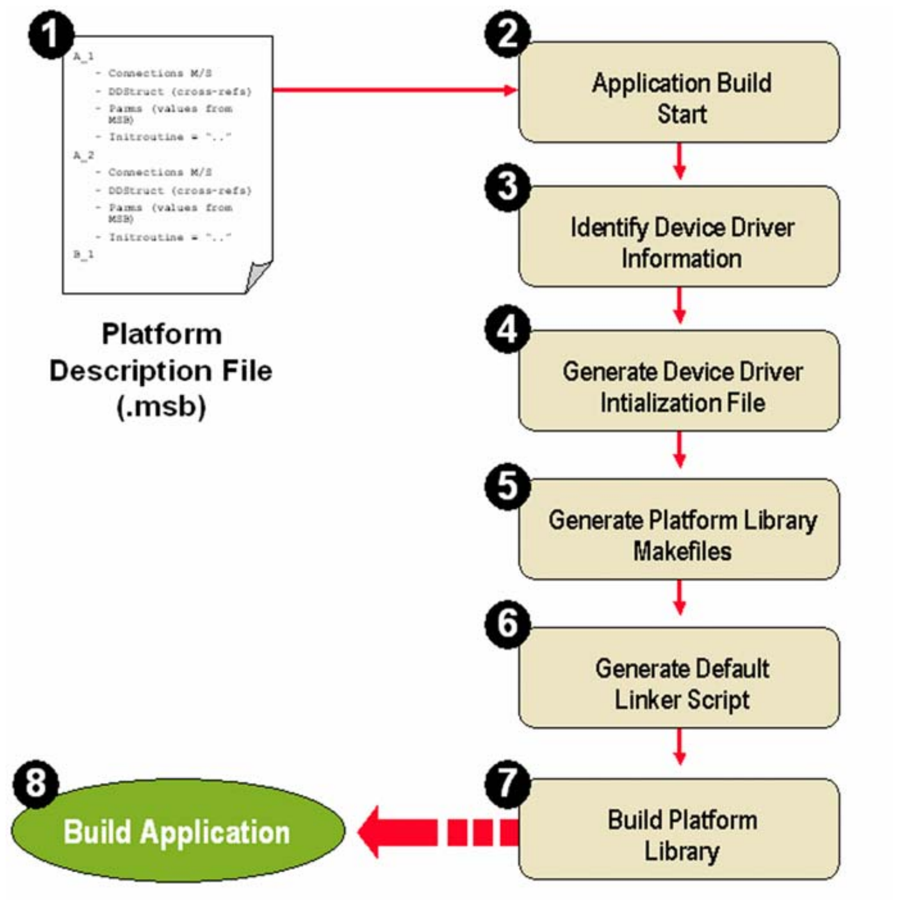
To clarify the build flow, a build example is provided in this chapter. It is based on the following information:

- LatticeMico8 C/C++ managed build project name: *MyProjectName*
- LatticeMico8 project folder: *<user_dir>\MyProjectName*
- LatticeMico8 platform name: Platform
- LatticeMico8 build configuration name: Release
- Application source code file name: HelloWorld.c

The Build Process

The LatticeMico8 managed build process is centered on information contained in the .msb file generated by MSB as part of platform generation. Figure 51 illustrates the steps in the process of building an application from the .msb file that you initially create in MSB.

Figure 51: Managed Build Process Diagram


1
A_1
- Connections M/S
- DDStruct (cross-refs)
- Parms (values from MSB)
- Initroutine = ".."
A_2
- Connections M/S
- DDStruct (cross-refs)
- Parms (values from MSB)
- Initroutine = ".."
B_1
Platform Description File (.msb)
2
Application Build Start
3
Identify Device Driver Information
4
Generate Device Driver Intialization File
5
Generate Platform Library Makefiles
6
Generate Default Linker Script
7
Build Platform Library
8
Build Application

All of the steps presented in Figure 51, particularly 2 through 7, do not necessarily occur in the order in which they are shown. They are presented in the manner shown for illustrative purposes.

These steps occur when building or rebuilding your project. In the C/C++ perspective, you build a project by choosing **Project > Build Project**. See “Understanding the Build Process” on page 46 and “Building Your Software Project” on page 47 and for more details.

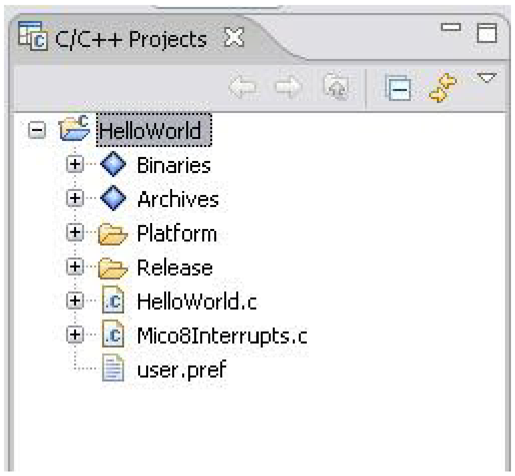
Build Directory Structure

The folder in which the C/C++ SPE project is saved cannot reside at the same directory level as the folder in which the MSB project is saved. The C/C++ SPE folder can reside at a higher or lower directory level than the MSB project folder.

LatticeMico8 C/C++ SPE splits a project build into two parts: the application build and platform library build. The platform library build outputs a platform library archive (<*platform*>.a) file that is referenced by the application build. It enables you to override any default software implementation by providing your own source file as part of the application build. Additionally, it helps maintain the demarcation between your source files and the device library source files that are used automatically by the software.

Figure 52 shows the top-level outline for the project, as viewed within the C/C++ perspective's Projects view, after performing a build.

Figure 52: Top-Level Application Structure Outline



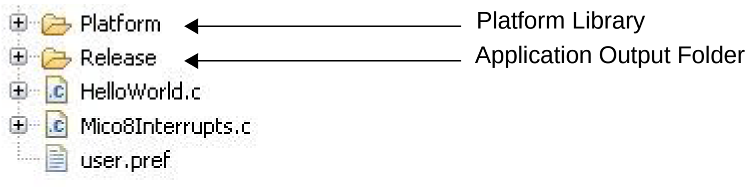
The Binaries and Archives folders in the Projects view do not actually exist in the project folder on the hard disk but contain certain files that are used by the project and are accessible here. Specifically, the Archives folder contains a platform library archive, and the Binaries folder contains an executable .elf file.

Figure 53 on page 95 shows how this top-level application structure in the Project view as shown in Figure 52 maps to the to the actual file system on your hard disk at the project folder level, that is, *MyProjectName* in the example.

C/C++ Perspective Project Folder File Contents

This section introduces you to the actual contents on your hard disk of what is represented in the project folder that is viewable in the C/C++ perspective's Projects view. Figure 53 shows the directory structure that you would view in Windows Explorer, in contrast to similar information on project content that you will view in the C/C++ perpective's Projects view, as shown in Figure 52.

Figure 53: Project Folder Contents

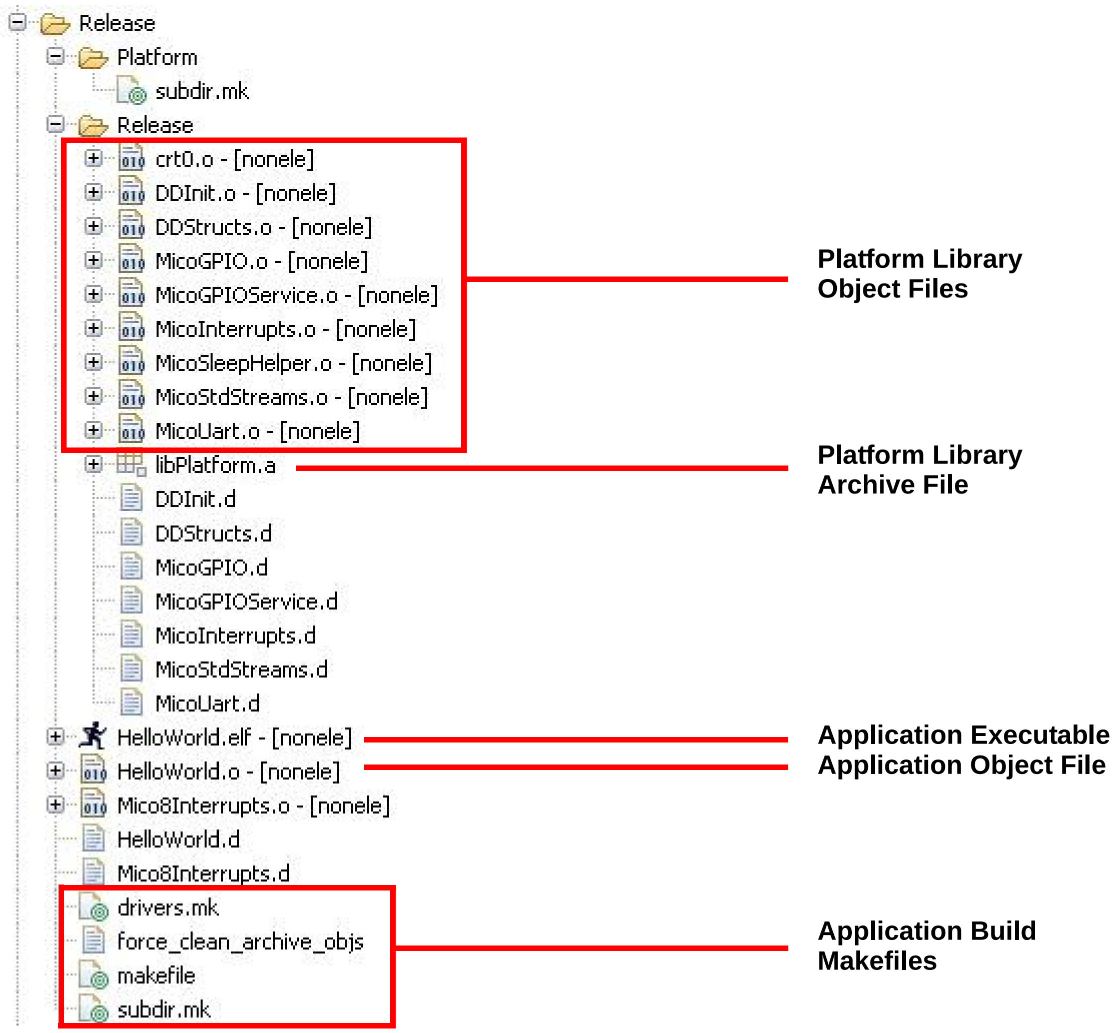


The project folder contains an application output (release) folder, a platform library folder, and various project information and user files.

Application Output Folder

The application output folder, named release in Figure 53 on page 95, contains the files that LatticeMico8 C/C++ SPE generates as it builds a particular software configuration, such as the final executable and compiled and assembled object files. The name of this folder corresponds to the name of a build configuration currently being used. If you switch between multiple build configurations, multiple directories are generated, with each named for the chosen build configuration. Figure 54 shows the contents of the application output folder.

Figure 54: Application Output Folder Contents



```
Release
  Platform
    subdir.mk
  Release
    crt0.o - [nonele]
    DDInit.o - [nonele]
    DDStructs.o - [nonele]
    MicoGPIO.o - [nonele]
    MicoGPIOService.o - [nonele]
    MicoInterrupts.o - [nonele]
    MicoSleepHelper.o - [nonele]
    MicoStdStreams.o - [nonele]
    MicoUart.o - [nonele]
    libPlatform.a
    DDInit.d
    DDStructs.d
    MicoGPIO.d
    MicoGPIOService.d
    MicoInterrupts.d
    MicoStdStreams.d
    MicoUart.d
  HelloWorld.elf - [nonele]
  HelloWorld.o - [nonele]
  Mico8Interrupts.o - [nonele]
  HelloWorld.d
  Mico8Interrupts.d
  drivers.mk
  force_clean_archive_objs
  makefile
  subdir.mk
```

Platform Library Object Files
Platform Library Archive File
Application Executable
Application Object File
Application Build Makefiles

The application output folder contains the following files:

- Application build makefiles: These makefiles enable the building of the application.
  - drivers.mk is similar to the drivers.mk makefile used by the library build. It includes component makefiles that provide header file relative path information for your source files. It also contains information that identifies driver sources that must be built as part of the application.
  - makefile is the application build makefile. It pulls in other makefiles that allow the generation and build of the platform library. It is responsible for generating the final executable image. This file is automatically generated and should not be modified.

  - subdir.mk identifies user sources contained within the project folder, as well as subdirectories in the project folder. It is automatically generated and maintained by Eclipse/CDT.

- Application executable is a result of linking the application and the platform library object file. It is an executable in ELF format that can be downloaded and executed by using the GNU debugger. For each build configuration, there is a unique application executable in the corresponding application output folder. If this application is targeted to another platform, the application executable and all associated files will be overwritten.
- Application object files are your source object files that have been compiled and assembled from their source C files. Figure 54 shows a single object file, LEDTest.o, in the directory structure that corresponds to its single source file as part of the application. If source subfolders are part of the project folder, the build process will contain similarly named folders containing object files generated from the source files that are present.
- Platform library object files are grouped into a subfolder that has the same name as the application output folder, for example, debug. They are put into this subdirectory to separate them from the application files in this directory structure. This folder contains the following files:
  - Platform library object (.o) files are the compiled outputs of the library source files. As explained earlier, these library source files are contained in the platform library folder.
  - Platform library archive (.a) file is derived from the platform library object files. The name of this archive file is automatically generated, prefixed with the “lib” string, with the root of the name corresponding to the name of the selected platform. This archive file is used when linking the application executable to resolve platform functions used by the application.

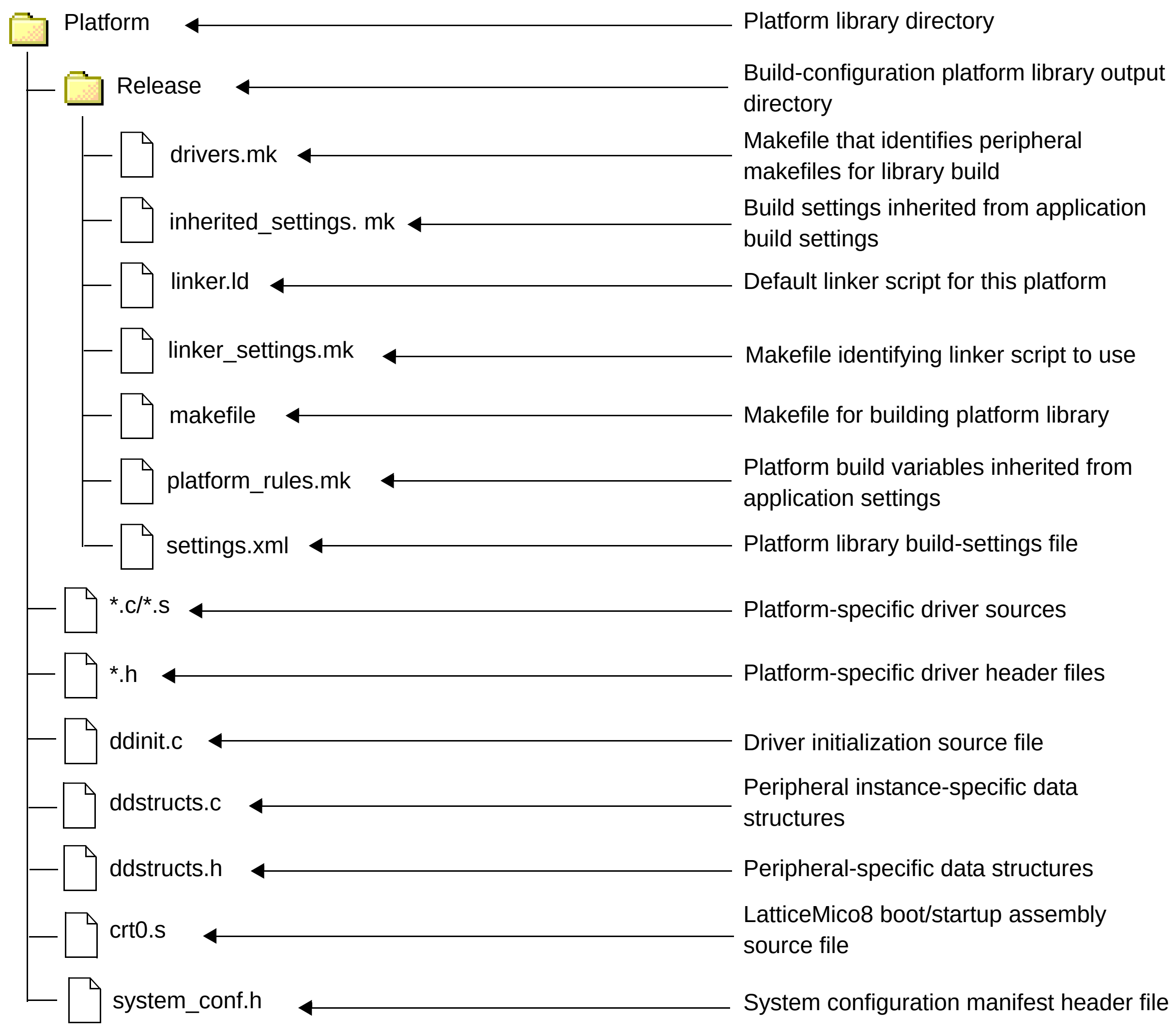
Platform Library Folder

The platform library folder contains the following:

- Source code files relevant for software support for the components specified in the platform
- Makefiles for building the project library archive (<*platform_name*>.a) file
- Makefiles that are referenced by the application makefile. These makefiles provide the following information to the application makefile:
  - Compiler flags that are activated when the following components are selected: hard multiplier, barrel shifter, sign-extend unit, and divider. These automatically activated compiler flags prevent you from having to manually set the appropriate compiler flags based on the CPU configuration.
  - Linker script selection (that is, the default or user-defined)

As shown in Figure 55, the platform library folder (PlatformE in the example) contains a subfolder named debug, the name of the build configuration used for this example. This folder contains the platform library contents, source, makefiles, and linker scripts.

Figure 55: Platform Library Directory Structure


Platform
Platform library directory
Release
Build-configuration platform library output directory
drivers.mk
Makefile that identifies peripheral makefiles for library build
inherited_settings. mk
Build settings inherited from application build settings
linker.ld
Default linker script for this platform
linker_settings.mk
Makefile identifying linker script to use
makefile
Makefile for building platform library
platform_rules.mk
Platform build variables inherited from application settings
settings.xml
Platform library build-settings file
.c/.s
Platform-specific driver sources
*.h
Platform-specific driver header files
ddinit.c
Driver initialization source file
ddstructs.c
Peripheral instance-specific data structures
ddstructs.h
Peripheral-specific data structures
crt0.s
LatticeMico8 boot/startup assembly source file
system_conf.h
System configuration manifest header file

The contents of the platform library folder are dynamically created and populated and should not be modified. The platform library folder and its associated contents are generated when you build the project for the first time.

The files and subfolders in the platform library folder are as follows.

- The build configuration folder (or debug folder in the example) contains all the files specific to that particular build configuration. As you would expect, these files can differ between build configurations that you might create in your project.

- The platform-specific component device driver source and header files are either copied from the components directories in the installation path or are automatically generated. The copied files, based on the .msb file, are described in the "Platform Library-Generated Source Files" on page 101. The DDinit.c file is an example of a file that is automatically generated. All platform library sources become part of the application project, aiding debug and source visibility.
- The project settings .xml file contains information about the parent project and its settings, as well as information on the platform referenced by the parent project. It is used to derive the makefiles for the platform library.
- The default linker script, linker.ld, is the default linker script for the particular platform or project combination and can be used as a starting point for creating a custom linker script file. The linker sections identified in this script are derived from the platform settings (user.pref) file.
- The makefiles are necessary for building the platform library, as well as for providing information to the application build. These makefiles facilitate building the application through the LatticeMico8 C/C++ SPE and the LatticeMico Cygwin shell. The platform library can be built independently of the application, using the LatticeMico Cygwin shell once the contents are populated. The following points provide a summary of the relevant makefiles:
  - makefile is the platform library makefile. It contains the commands that define the targets, rules, and dependencies that tell the make utility how to construct the software build from its sources.
  - drivers.mk includes relevant component makefiles. These component makefiles identify the sources and paths for the corresponding component device drivers. This makefile is referenced only by the platform library makefile. It is derived from information present in the .msb file.
  - inherited_settings.mk contains compiler settings derived from the build configuration. These settings can be changed in the user interface, as shown in Figure 20 on page 49. In addition to compiler settings, this file also contains the location for depositing the built platform library archive (.a) file and its associated compiled or assembled object files. This file is referenced only by the platform library makefile.
  - linker_settings.mk identifies which linker script to use, that is, either the default or a user-defined makefile. This file is referenced by the application makefile and is not used by the platform library makefile. It is derived from information present in the platform settings (user.pref) file.
  - platform_rules.mk contains compiler switches. It is referenced by the application makefile, as well as the platform library makefile. These compiler switches are extracted according to the microcontroller configuration information contained in the .msb file.

If another build configuration is created and used in addition to the default debug configuration, the managed build process generates a new platform library for each configuration. The files for this new library all reside in the platform library folder.

If you create a new build configuration, a new build configuration subfolder is created in the platform library folder.

In Figure 55 on page 98, a newly generated build configuration folder would be placed under the Platform folder at the same level as the debug build configuration folder. This new build configuration folder would hold the files specific to that particular build, its makefiles, and linker scripts. All the platform library source files are held in the platform library folder. This single copy of the source is used across all defined build configurations.

Perl scripts invoked from makefiles, included by the application build makefile, generate the contents of the platform library folder. Figure 54 on page 96 shows an outline of the application output folder contents.

Project Information and User Files Folder

The project information and user files are the following:

- Eclipse/CDT project information files should not be modified:
  - .cdtbuild
  - .cdtproject
  - .project
- User files that you create or provide as part of the project. The source files contained in the project folder or any subfolder become part of the build process without you having to explicitly specify them.
  - Template source file, LEDTest.c, is a C programming source file.
  - Template description file, LEDTest.txt, is an ASCII-formatted text file.
- The platform settings file, user.pref, contains platform information for the platform used by this project. It is generated by the managed build process. It dictates how the executable is targeted to your platform, because it stores information that you set in the Platform tab in the Properties dialog box. See Figure 22 on page 51. For example, it contains information on your settings that tell the platform build to use the default or a user-defined linker script in the linker section.

  Note

  The user.pref file is automatically generated during the build process, so it is not recommended that you modify a user-defined version of this file in its present location or it will be overwritten. You should copy any custom versions of this file to another area to maintain your user-defined preferences.

- The platform library folder contains platform-specific device-driver sources for the chosen platform. It is explained in “Platform Library Folder” on page 97. It also contains the default linker script and makefiles that are needed for building the platform library, as well as those used by the application build. The name of this folder, Platform, shown in Figure 54 on page 96, is derived from the referenced platform. If you target your project to different platforms, there will be multiple platform library directories that are given a name that corresponds to the referenced platform. Refer to Figure 55 on page 98, which outlines the platform library folder contents.

Platform Library-Generated Source Files

This section explains how platform-library-generated source files associated with components used in the platform (for example, driver code) are brought into the build process so that the application code that directly (or indirectly) uses this component-specific code can be linked properly. In the managed build process, some C source files are automatically generated and put into the platform library folder, as shown in Figure 55 on page 98. The contents of these source files depend on what components are in the platform being targeted.

A key mechanism to enable linking of the application code properly during the build process is the .msb file created in Mico System Builder (MSB). Each component in the platform is represented in the .msb file. The information about each component in the .msb file includes details about the component's C source files that must be included in the build process. This component information is called the component information structure declaration and originates from the <*component_name*>.xml file in the installation directory. For more information on this component information structure declaration, see "DDStructs.h File" on page 103.

If a component is instantiated in a platform, the contents of that component's .xml file are included in the .msb file.

To enhance the description of the concepts presented here, this section uses a build example based on the following information:

- LatticeMico8 C/C++ Managed Build Project Name: *MyProjectName*
- LatticeMico8 Platform: Platform
- LatticeMico8 Build Configuration: Release

There are four main source/header files whose contents are platform-specific and are automatically generated as part of the platform library:

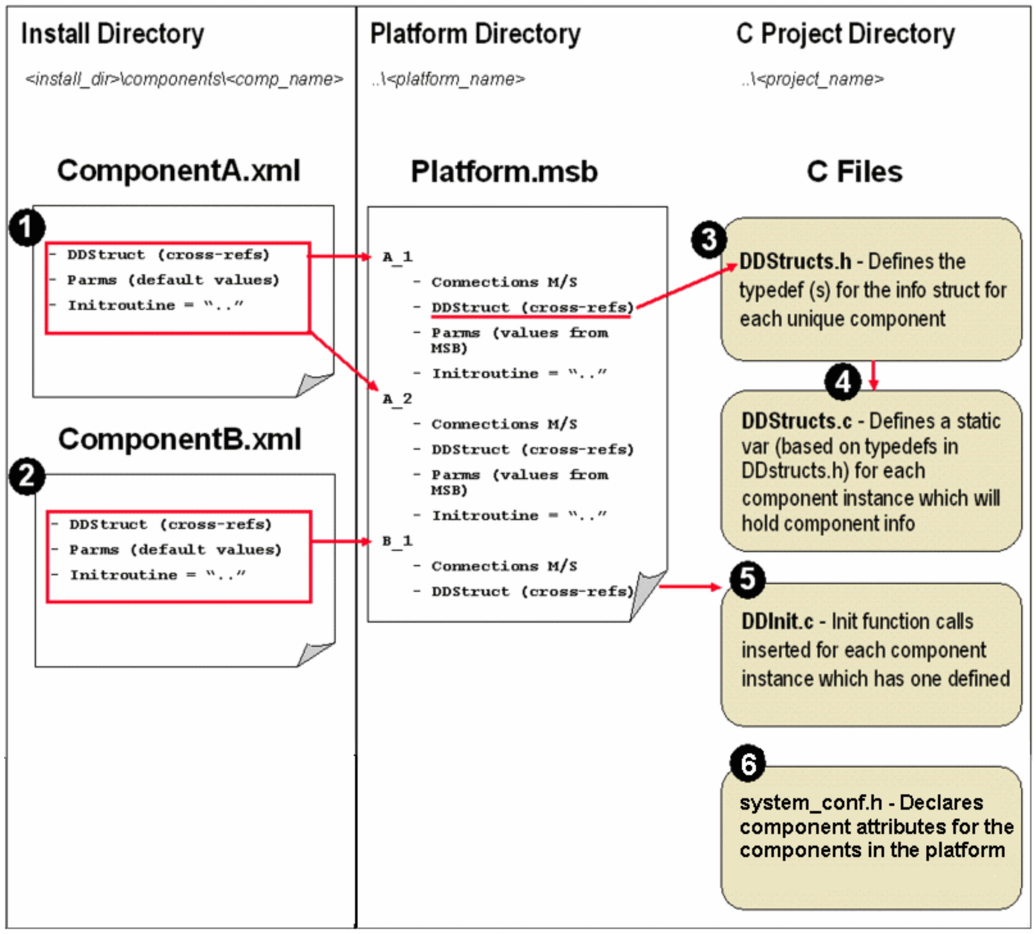
- DDStructs.h
- DDStructs.c
- DDInit.c
- system_conf.h

Figure 56 illustrates the following:

- In MSB, a platform is created using two instances of Component A, called "A_1" and "A_2," and one instance of Component B, called "B_1."
- Steps 1 and 2 are performed by the MSB tool when the .msb file is saved.
  - ComponentA.xml information is copied twice into the .msb file.
  - ComponentB.xml information is copied into the .msb file.

- Additional information is also added into the .msb file, for example, how the components are connected in the platform.
- During the managed build process, steps 3, 4, and 5 are completed.

Figure 56: Component Information Flow to Platform Library Files


Install Directory
<install_dir>\components\<comp_name>
Platform Directory
..\<platform_name>
C Project Directory
..\<project_name>
ComponentA.xml
1
- DDStruct (cross-refs)
- Parms (default values)
- Initroutine = "..″
ComponentB.xml
2
- DDStruct (cross-refs)
- Parms (default values)
- Initroutine = "..″
Platform.msb
A_1
- Connections M/S
- DDStruct (cross-refs)
- Parms (values from MSB)
- Initroutine = "..″
A_2
- Connections M/S
- DDStruct (cross-refs)
- Parms (values from MSB)
- Initroutine = "..″
B_1
- Connections M/S
- DDStruct (cross-refs)
C Files
3
DDStruct.h - Defines the typedef (s) for the info struct for each unique component
4
DDStructs.c - Defines a static var (based on typedefs in DDstructs.h) for each component instance which will hold component info
5
DDInit.c - Init function calls inserted for each component instance which has one defined
6
system_conf.h - Declares component attributes for the components in the platform

- All the C files on the right in Figure 56 are automatically generated and are deposited into your platform folder, along with your .msb file. The device driver structures header and source files, DDStructs.h and DDStructs.c, respectively, derive information from attributes assigned to them during component definition.
- The DDStructs.c file is generated according to the contents of the DDStructs.h file but information is created on the basis of unique component names relative to this construct.
- The device driver initialization source file, DDInit.c, is generated according to initialization routines for each component that are designed during component definition.
- The system configuration C header file, system_conf.h, declares component attributes as manifest constants according to the component specification for the components in the platform.

DDStructs.h File

The device driver structures header (DDStructs.h) file is the main header file for any managed C/C++ application. It is also referenced by device-driver implementations. It defines platform-specific information, such as the CPU clock frequency and component-specific information structures. This information is extracted from the .msb file.

Note

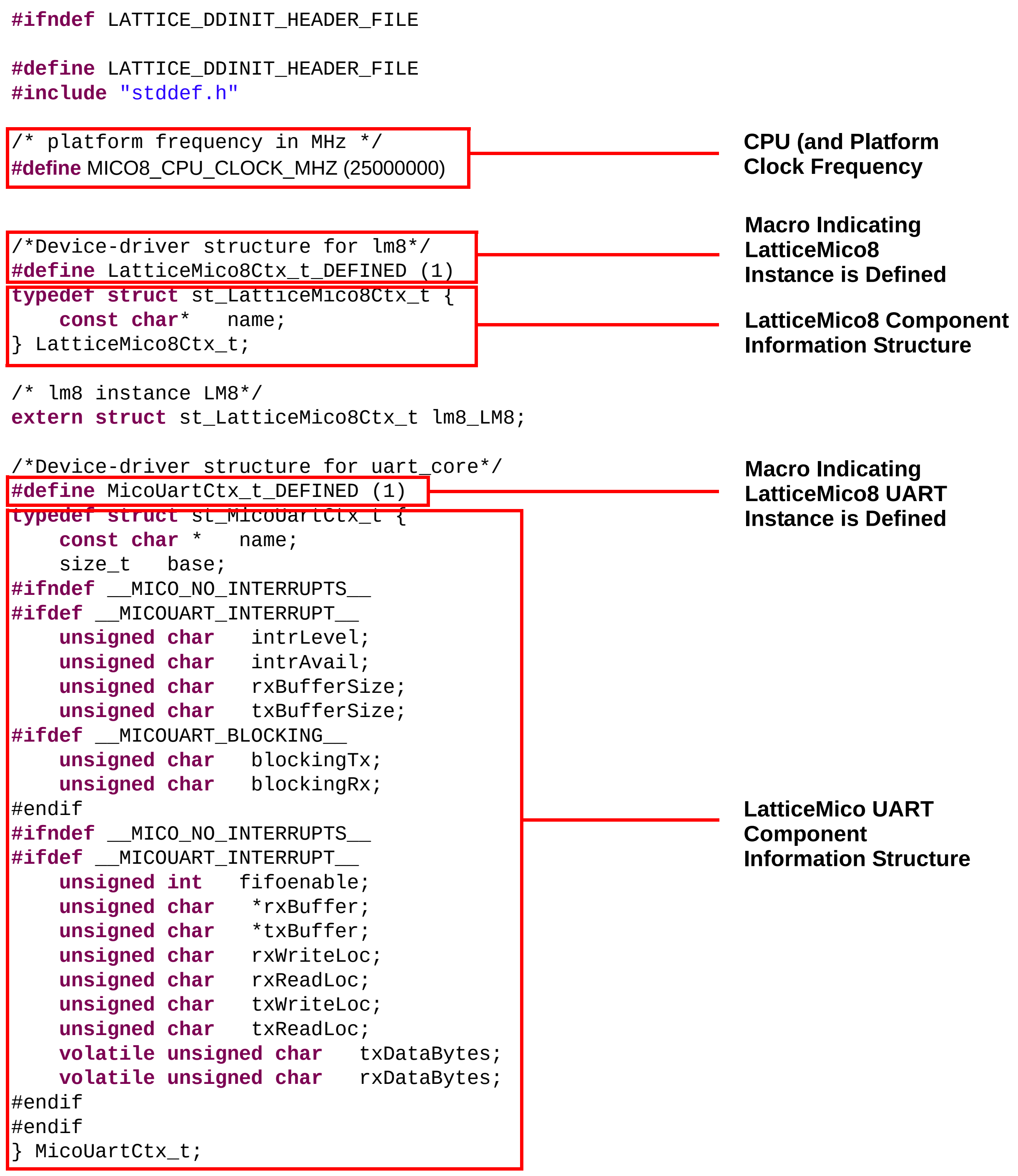
Information about the component-specific information structure and initialization function information originates in the <*comp_name*>.xml file. This information has been copied into the .msb file when the platform was created in MSB. The information presented is subject to change in future releases of MSB.

The DDStructs.h file contains the following information:

- MICO8_CPU_CLOCK_MHZ macro defines the CPU clock frequency. This information is extracted from the .msb file.
- Component information structure declarations are specified as part of the .xml file. MSB copies this information into the .msb file. The information is then extracted from the .msb file by the managed build process and translated as C structure definitions that appear in the DDStructs.h file. Each unique component has its own unique component information structure defined. For multiple instances of the same component, the build ensures that there are no duplicate structure definitions.
- Component instance declaration: Through the extern statement, the header file declares the presence of an information structure for each component instantiated in the platform. For example, in the DDStructs.h file shown, the platform has a GPIO named "LED." Through extern, the file declares that a definition exists for the gpio_LED instance of the st_MicoGPIOCtx_t structure. See Figure 57. The actual definition of the instance of this structure is in the DDStructs.c file.

  If a given component has multiple instances, the build process defines and declares uniquely named instances of the same structure type. This process relies on unique names for each instance of a given component in a platform. This rule is enforced by MSB when creating or editing a platform.

Figure 57: Sample DDStructs.h File



```
#ifndef LATTICE_DDINIT_HEADER_FILE

#define LATTICE_DDINIT_HEADER_FILE
#include "stddef.h"

/* platform frequency in MHz */
#define MICO8_CPU_CLOCK_MHZ (25000000)


/*Device-driver structure for lm8*/
#define LatticeMico8Ctx_t_DEFINED (1)
typedef struct st_LatticeMico8Ctx_t {
    const char*   name;
} LatticeMico8Ctx_t;

/* lm8 instance LM8*/
extern struct st_LatticeMico8Ctx_t lm8_LM8;

/*Device-driver structure for uart_core*/
#define MicoUartCtx_t_DEFINED (1)
typedef struct st_MicoUartCtx_t {
    const char *   name;
    size_t   base;
#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
    unsigned char   intrLevel;
    unsigned char   intrAvail;
    unsigned char   rxBufferSize;
    unsigned char   txBufferSize;
#ifdef __MICOUART_BLOCKING__
    unsigned char   blockingTx;
    unsigned char   blockingRx;
#endif
#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
    unsigned int   fifoenable;
    unsigned char   *rxBuffer;
    unsigned char   *txBuffer;
    unsigned char   rxWriteLoc;
    unsigned char   rxReadLoc;
    unsigned char   txWriteLoc;
    unsigned char   txReadLoc;
    volatile unsigned char   txDataBytes;
    volatile unsigned char   rxDataBytes;
#endif
#endif
} MicoUartCtx_t;
```

CPU (and Platform Clock Frequency
Macro Indicating LatticeMico8 Instance is Defined
LatticeMico8 Component Information Structure
Macro Indicating LatticeMico8 UART Instance is Defined
LatticeMico UART Component Information Structure

Figure 58: Sample DDStructs.h File (Continued)

```
/* uart_core instance uart*/
extern struct st_MicoUartCtx_t uart_core_uart;
```

LatticeMico UART Instance Declaration

```
/* declare uart instance of uart_core */
extern void MicoUartInit(struct st_MicoUartCtx_t*);
```

LatticeMico UART Initialization Function Declaration

```
/*Device-driver structure for gpio*/
#define MicoGPIOCtx_t_DEFINED (1)
typedef struct st_MicoGPIOCtx_t {
    const char*   name;
    size_t   base;
    unsigned char   intrLevel;
    unsigned int   output_only;
    unsigned char   input_only;
    unsigned char   in_and_out;
    unsigned char   tristate;
    unsigned char   data_width;
    unsigned char   input_width;
    unsigned char   output_width;
    unsigned char   intr_enable;
} MicoGPIOCtx_t;


/* gpio instance LED*/
extern struct st_MicoGPIOCtx_t gpio_LED;

/* declare LED instance of gpio */
extern void MicoGPIOInit(struct st_MicoGPIOCtx_t*);

extern int main();

#endif
```

DDStructs.c File

Figure 58 shows a sample device driver structures source (DDStructs.c) file corresponding to the DDStructs.h file shown earlier. The DDStructs.c file contains instance-specific component information structures. The build process populates the structure data on the basis of how that component instance was configured in MSB.

Figure 58: Sample DDStructs.c File

```
#include "DDStructs.h"

/* lm8 instance LM8*/
struct st_LatticeMico8Ctx_t lm8_LM8 = {
    "LM8"};

/* uart_core instance uart*/
#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
  /* array declaration for rxBuffer */
   unsigned char _uart_core_uart_rxBuffer[4];
  /* array declaration for txBuffer */
   unsigned char _uart_core_uart_txBuffer[4];
#endif
#endif

struct st_MicoUartCtx_t uart_core_uart = {
    "uart",
    0x0000,

#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
    0,
    1,
    4,
    4,
#endif
#endif

#ifdef __MICOUART_BLOCKING__
    1,
    1,
#endif

#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
    0,
    _uart_core_uart_rxBuffer,
    _uart_core_uart_txBuffer,
#endif
#endif
};

/* gpio instance LED*/
struct st_MicoGPIOCtx_t gpio_LED = {
    "LED",
    0x0010,
    255,
    1,
    0,
    0,
    0,
    4,
    1,
    1,
    0};
```

LatticeMico UART Instance Information Structure

The structure instances have the same name as those declared in DDStructs.h and are generated by the same Perl script, which precludes compilation issues. Since each structure has a unique name, the platform can include multiple instances of the same component, and the build process extracts and populates the information for these structures according to the configuration of the platform. In the sample DDStructs.c source file in Figure 57, you can see how this file uses the the GPIO instance information that it derived from the DDStructs.h file by comparing it to the sample DDStructs.h file shown in Figure 57.

The automatically generated DDInit.c file implements the LatticeDDInit function, which resets the CPU. It is described “Overriding Default Driver Initialization Sequence” on page 73.

If a component has an initialization function to be called at reset, it is called from the LatticeDDInit function. This LatticeDDInit function is called by the boot-up process as part of CPU reset. It allows the platform library to call the component instance initialization routines as part of boot-up.

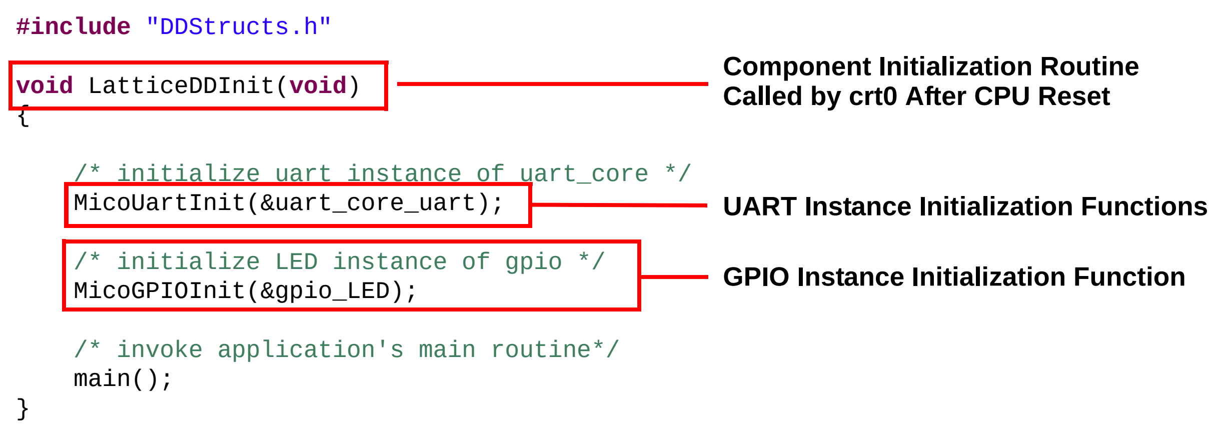
DDInit.c File

As noted in the last section, the device driver initialization source (DDInit.c) file contains the LatticeDDInit function. The LatticeDDInit function calls the initialization function for each component instance in the target platform.

The managed build process automatically creates an information structure for each component instance. An initialization routine name is defined in the .msb for each component type from the information that is specified in the <*comp_name*>.xml file. The LatticeDDInit function is automatically generated so that it calls these initialization routines for each component instance, using the component instance's information structure defined in DDStructs.h.

During boot-up, the DDInit.c file is called by crt0 as part of CPU reset in the DDStructs.c file, which tells the platform library to call the component instance initialization routines. Refer to “Overriding Default Driver Initialization Sequence” on page 73 for more details on the LatticeDDInit function.

Figure 59: DDInit.c Source Code Sample



```
#include "DDStructs.h"

void LatticeDDInit(void)
{

    /* initialize uart instance of uart_core */
    MicoUartInit(&uart_core_uart);

    /* initialize LED instance of gpio */
    MicoGPIOInit(&gpio_LED);

    /* invoke application's main routine*/
    main();
}
```

Component Initialization Routine Called by crt0 After CPU Reset
UART Instance Initialization Functions
GPIO Instance Initialization Function

The build process uses the .msb file to create the DDInit.c file. This routine takes a pointer to the instance-specific component instance information structure as its parameter, allowing the same initialization routine to be invoked multiple times for multiple instances.

After invoking the component initialization routines, LatticeDDInit calls the int main(void) function that you must implement. This int main(void) function is the starting point of your code.

System_Conf.h File

The system configuration C header file, system_conf.h, contains C syntax manifest constants for each component's attributes as configured in MSB. This information is extracted from the platform specification file for the platform chosen for the C/C++ SPE project. The system_conf.h file is overwritten during software builds, so it should not be modified.

The system_conf.h file is generated by a Perl script function in the msb_mdk_subs.pm Perl module file located in the //micosystem/utilities/perlscript/ folder. The Perl function extracts the following information from the platform specification file:

- Platform attributes
- Processor attributes
- Component attributes for I/O-type components
- Component attributes for memory-type components

Platform Attributes

Figure 60 shows the platform attributes for a sample platform in the system_conf.h header file.

Figure 60: Sample Platform Attributes in system_conf.h File

```
#define FPGA_DEVICE_FAMILY    "MachXO2"
#define PLATFORM_NAME         "Platform"
#define USE_PLL               (0)
#define CPU_FREQUENCY         (25000000)
```

Table 2 lists the platform attributes and their properties.

Table 2: Platform Attributes

Atttribute	C Type	Information
FPGA_DEVICE_FAMILY	String	FPGA device family selection in MSB
PLATFORM_NAME	String	MSB platform name

Table 2: Platform Attributes (Continued)

Atttribute	C Type	Information
USE_PLL	Numeric	Indicates PLL selection: ▶ 0 means that the PLL is absent. ▶ 1 means that the PLL is present.
CPU_FREQUENCY	Numeric	Indicates platform frequency, taking into account PLL selection.

Processor Attributes

Figure 61 shows the processor attributes in the system_conf.h header file for a sample platform.

Figure 61: Sample Processor Attributes in system_conf.h File

```
/*
 * CPU Instance LM8 component configuration
 */
#define CPU_NAME "LM8"
```

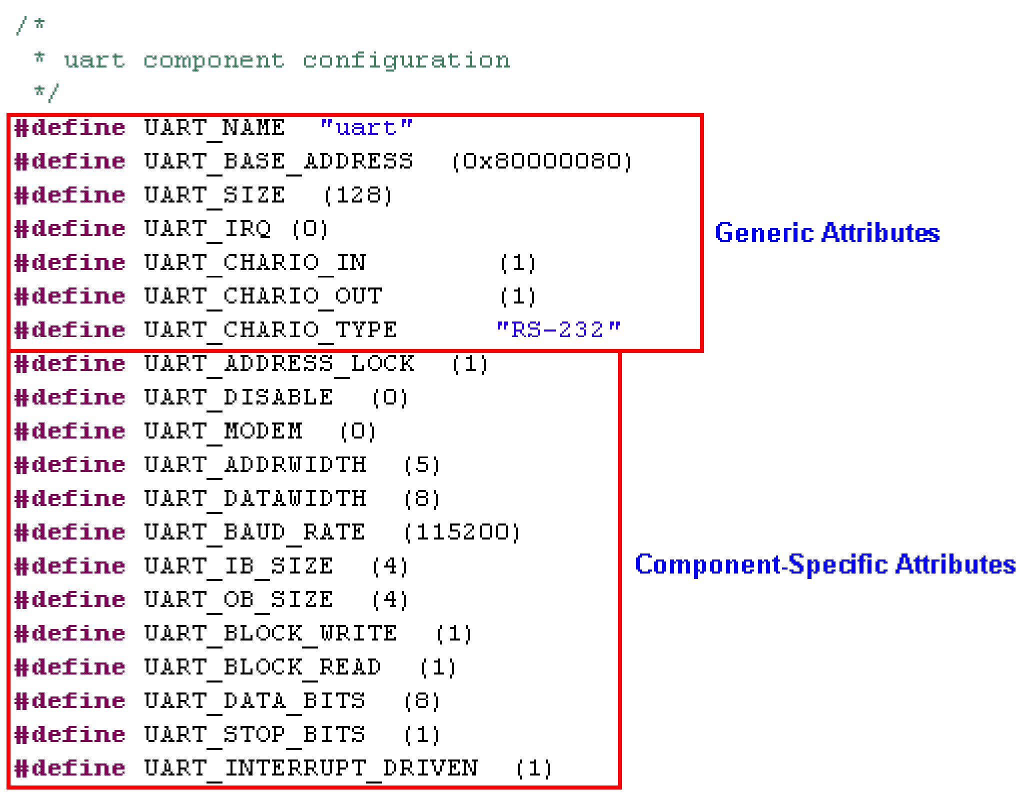
Attributes for I/O-Type Components

I/O-type components have two types of attributes:

- Generic attributes, such as base address and size, exist for all I/O-type components.
- Component-specific attributes, such as the UART baud rate selection in MSB, are specific to a component.

Figure 62 shows sample UART component attributes in the system_conf.h header file.

Figure 62: Sample UART Component Attributes in system_conf.h File



```
/*
 * uart component configuration
 */
#define UART_NAME  "uart"
#define UART_BASE_ADDRESS  (0x80000080)
#define UART_SIZE  (128)
#define UART_IRQ (0)
#define UART_CHARIO_IN          (1)
#define UART_CHARIO_OUT         (1)
#define UART_CHARIO_TYPE        "RS-232"
#define UART_ADDRESS_LOCK  (1)
#define UART_DISABLE  (0)
#define UART_MODEM  (0)
#define UART_ADDRWIDTH  (5)
#define UART_DATAWIDTH  (8)
#define UART_BAUD_RATE  (115200)
#define UART_IB_SIZE  (4)
#define UART_OB_SIZE  (4)
#define UART_BLOCK_WRITE  (1)
#define UART_BLOCK_READ  (1)
#define UART_DATA_BITS  (8)
#define UART_STOP_BITS  (1)
#define UART_INTERRUPT_DRIVEN  (1)
```

Naming Conventions The attributes are in the following format:

`#define <INSTANCE_NAME>_<ATTRIBUTE_NAME>`

- *<INSTANCE_NAME>* is the name of the component instance, in capital letters.
- *<ATTRIBUTE_NAME>* is the attribute name specified in the component description file for the component, in capital letters.

Generic Attributes for I/O-Type Components Table 3 lists the generic attributes for all I/O-type components.

Table 3: Generic Attributes for I/O-Type Components

Atttribute	C Type	Information
NAME	String	Component instance name as specified in MSB
BASE_ADDRESS	Numeric	Base address assigned in MSB
SIZE	Numeric	Address size specified in MSB, in bytes

Table 3: Generic Attributes for I/O-Type Components (Continued)

Atttribute	C Type	Information
IRQ	Numeric	IRQ assigned in MSB This attribute is absent for components that do not have an interrupt line connected to the processor. For components with an interrupt line to the processor, the value is 0 through 31. A value of 255 indicates the absence of an interrupt line (reserved for future interpretation of this field).
CHARIO_IN	Numeric	Indicates if the component's description file has marked this component available for character (file) input operations. ▶ 0 means this component is not marked as available for character input operations. ▶ 1 means this component is marked as available for character input operations.
CHARIO_OUT	Numeric	Indicates whether the component's description file has marked this component available for character (file) output operations. ▶ 0 means this component is not marked as available for character output operations. ▶ 1 means this component is marked as available for character output operations.
CHARIO_TYPE	String	Represents the character I/O type as specified in the component specification (for example, JTAG UART or RS-232 UART). This attribute is present only if the component is marked available for either input or output.

Component-Specific Attributes The component-specific attributes specified for the component in the platform specification file for the platform are listed as encountered. The MSB Component Attributes pane lists the user-configurable component attributes, along with the software constant names, that will be generated in the system_conf.h header file.

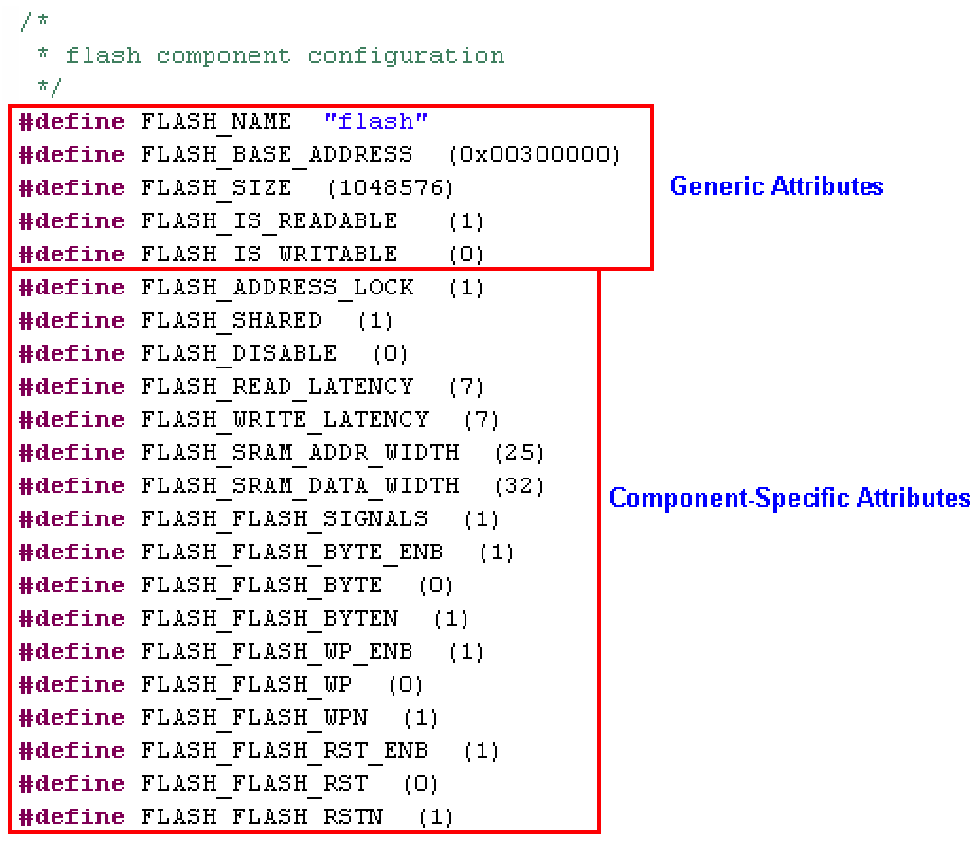
Attributes for Memory-Type Components

Memory-type components have two types of attributes:

- Generic attributes, such as base address and size, exist for all memory-type components.
- Component-specific attributes, such as data width, are specific to a component.

Figure 63 shows a sample of the flash component attributes in the system_conf.h file.

Figure 63: Sample Flash Component Attributes in system_conf.h File



```
/*
 * flash component configuration
 */
#define FLASH_NAME  "flash"
#define FLASH_BASE_ADDRESS  (0x00300000)
#define FLASH_SIZE  (1048576)
#define FLASH_IS_READABLE   (1)
#define FLASH_IS_WRITABLE   (0)
#define FLASH_ADDRESS_LOCK  (1)
#define FLASH_SHARED  (1)
#define FLASH_DISABLE  (0)
#define FLASH_READ_LATENCY  (7)
#define FLASH_WRITE_LATENCY  (7)
#define FLASH_SRAM_ADDR_WIDTH  (25)
#define FLASH_SRAM_DATA_WIDTH  (32)
#define FLASH_FLASH_SIGNALS  (1)
#define FLASH_FLASH_BYTE_ENB  (1)
#define FLASH_FLASH_BYTE  (0)
#define FLASH_FLASH_BYTEN  (1)
#define FLASH_FLASH_WP_ENB  (1)
#define FLASH_FLASH_WP  (0)
#define FLASH_FLASH_WPN  (1)
#define FLASH_FLASH_RST_ENB  (1)
#define FLASH_FLASH_RST  (0)
#define FLASH_FLASH_RSTN  (1)
```

Naming Conventions The attributes are in the following format:

`#define <INSTANCE_NAME>_<ATTRIBUTE_NAME>`

- *<INSTANCE_NAME>* is the name of the component instance, in capital letters.
- *<ATTRIBUTE_NAME>* is the attribute name specified in the component description file for the component, in capital letters.

Generic Attributes for Memory-Type Components Table 4 lists the generic attributes present for all memory-type components.

Table 4: Generic Attributes for Memory-Type Components

Atttribute	C Type	Information
NAME	String	Component instance name as specified in MSB
BASE_ADDRESS	Numeric	Base address assigned in MSB
SIZE	Numeric	Address size specified in MSB, in bytes

Table 4: Generic Attributes for Memory-Type Components

Atttribute	C Type	Information
IS_READABLE	Numeric	Indicates whether the memory component is readable by the processor without software support: ▶ 0 indicates that the memory is not readable. ▶ 1 indicates that the memory is readable.
IS_WRITABLE	Numeric	Indicates if the memory component is writable by the processor without software support: ▶ 0 indicates that the memory is not writable. ▶ 1 indicates that the memory is writable.

Component-Specific Attributes The component-specific attributes specified for the component in the platform specification file for the platform are listed as encountered. The MSB Component Attributes pane lists the user-configurable component attributes, along with the software constant names, that will be generated in the system_conf.h header file.

Component Software Elements

This section describes all of the information that a component must have to be used in the MSB tool and by the managed make project.

As previously stated, the build process automatically generates several C/C++ files whose content is determined by which components are defined in the platform and how they are configured. To do this, the process uses the .msb file that you created in MSB. However, the component-specific information in the .msb file originates in the .xml files that exist for each type of available component.

The following information is necessary for MSB and the managed make utility:

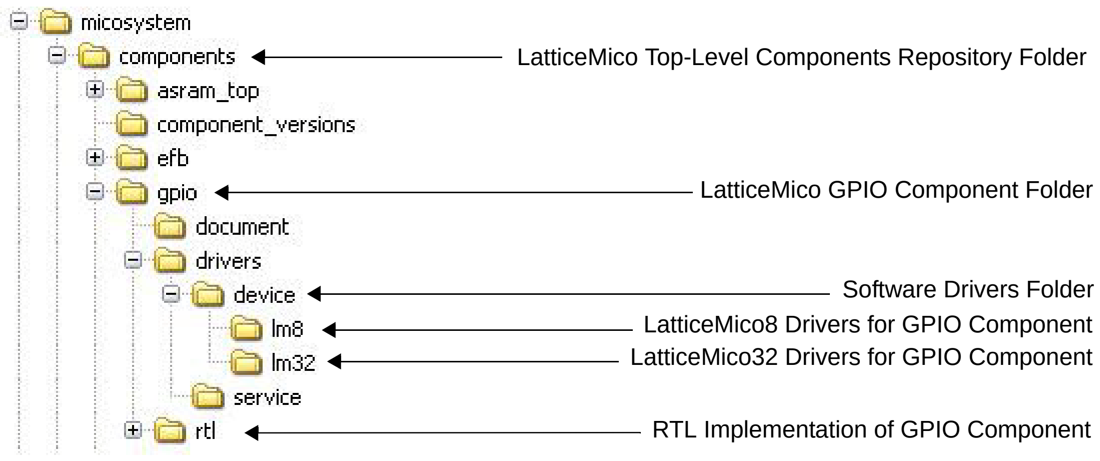
- <*comp_name*>.xml files, which exist for each element and reside in the ..\components\<*comp_name*> folder in your project. Each .xml file contains reference information on component initialization routines and a component information structure declaration that provides details about the component's source files, which are later picked up in the .msb file when a platform definition is created. For more information on this component information structure declaration, see "DDStructs.h File" on page 103.
- Device-driver files, (.c, .h) are the source files that contain driver code that is compiled into object files during the software build. Component-specific APIs are contained in these component source .c and .h files. You can also consider the .s and .S source assembly files as driver code files.

Information Structure Specification

As mentioned previously, the managed build process extracts component instance-specific information from the .msb file, creates specified structures, and calls specified initialization routines that originate from the .xml file.

Figure 64 depicts a typical directory structure for a Mico System Builder (MSB) component residing in the top-level components epository folder.

Figure 64: LatticeMico GPIO Component Folder Directory Structure


micosystem
components
asram_top
component_versions
efb
gpio
document
drivers
device
lm8
lm32
service
rtl
LatticeMico Top-Level Components Repository Folder
LatticeMico GPIO Component Folder
Software Drivers Folder
LatticeMico8 Drivers for GPIO Component
LatticeMico32 Drivers for GPIO Component
RTL Implementation of GPIO Component

The example used for this section is the LatticeMico GPIO device. In Figure 64, the GPIO component has two main subdirectories, drivers and RTL. The drivers folder contains software support for LatticeMico DMA component, and the rtl folder contains RTL support.

The gpio folder contains a single file, gpio.xml. This .xml file is the GPIO component description file that contains RTL instantiation and GUI information for MSB, as well as component information structure information. In the .msb file, this component instance has a Parms section. The values for the attributes in the Parms section were defined when the platform was created in MSB.

The build looks in the component's Parms section of the .msb file for a parameter with a name that matches the value of the attribute value. The value of this parameter is used as the initial value for this element of the structure variable. For example, for the GPIO instance in the .msb, there is a parameter named "BASE_ADDRESS" in its Parms section. If this parameter had a value of 0x80000080, this value would be the initial value for the element "base" in the GPIO information structure variable in the DDStructs.c file. Information structure element names are associated with parameter names, so that when a parameter is set in MSB, the correct information structure element is assigned that value.

Source Code Organization

The previous section described how a component's instance information in a platform definition is transferred to the generated platform library DDStructs.c, DDStructs.h, and DDInit.c source files. Typically a component that defines a component information structure has some software support in the form of source files and header files that provide device driver implementation, in addition to any services.

For example, the LatticeMico GPIO component provides easy-to-use API routines for manipulating the GPIO. The UART component provides a device-driver implementation that uses UART-specific component instance information for transfer of data over an RS-232 link.

In a mature project, the individual component directories—for example, the .\components\timer subfolder—appear in both the micosystem installation folder and also in your project C folder. After the platform generation process in MSB, the GPIO component subfolder and all of its contents are copied into your platform folder's directory structure. You will not see the gpio.xml file in the platform folder's directory structure.

The component-specific source files must be located in the drivers directory or in subdirectories in the drivers folder of the component folder. In addition, the drivers folder must contain a makefile named peripheral.mk. Makefiles with other names are not processed. Figure 65 shows a sample drivers folder's directory structure for LatticeMico GPIO component. This figure is an extension of the LatticeMico GPIO component directory structure.

Figure 65: LatticeMico GPIO Component Software Files

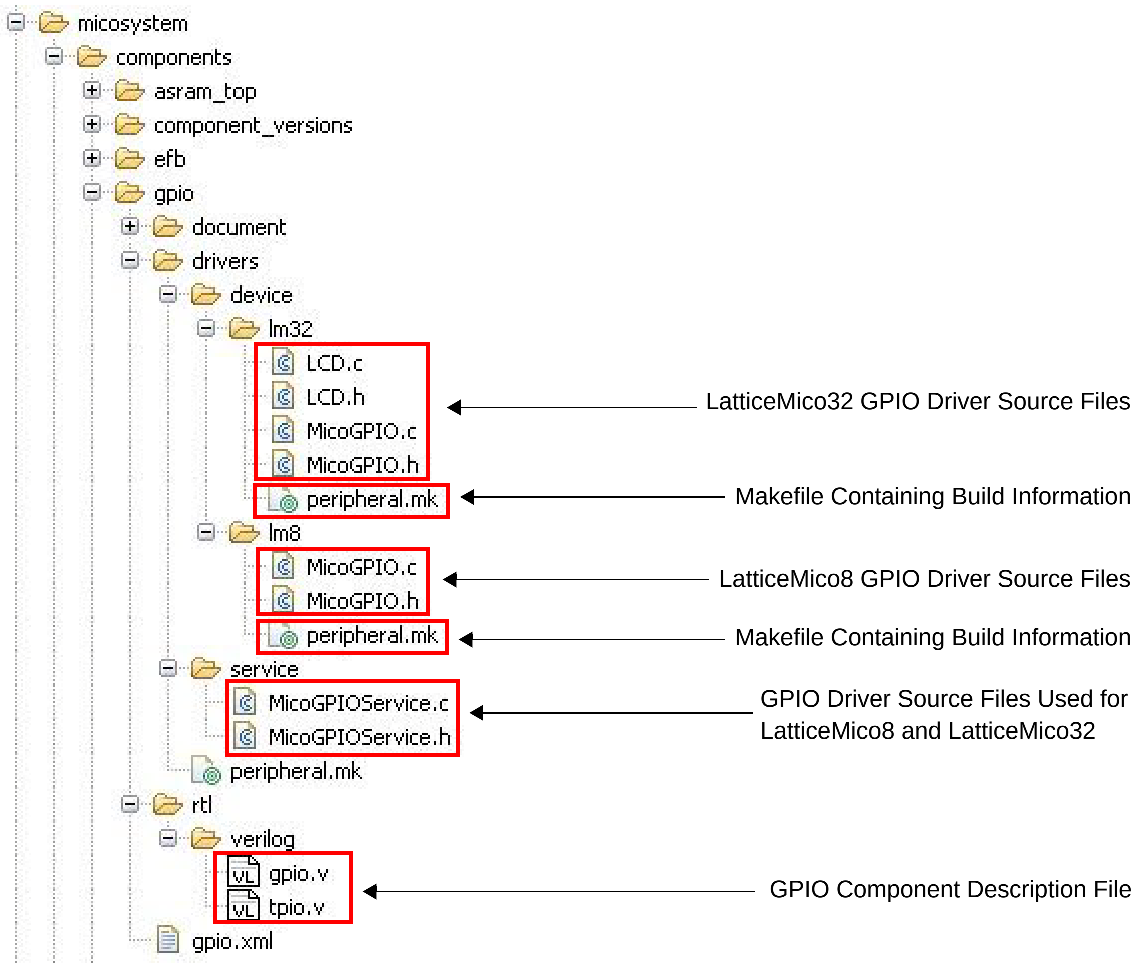

micosystem
components
asram_top
component_versions
efb
gpio
document
drivers
device
lm32
LCD.c
LCD.h
MicoGPIO.c
MicoGPIO.h
peripheral.mk
lm8
MicoGPIO.c
MicoGPIO.h
peripheral.mk
service
MicoGPIOService.c
MicoGPIOService.h
peripheral.mk
rtl
verilog
gpio.v
tpio.v
gpio.xml
LatticeMico32 GPIO Driver Source Files
Makefile Containing Build Information
LatticeMico8 GPIO Driver Source Files
Makefile Containing Build Information
GPIO Driver Source Files Used for LatticeMico8 and LatticeMico32
GPIO Component Description File

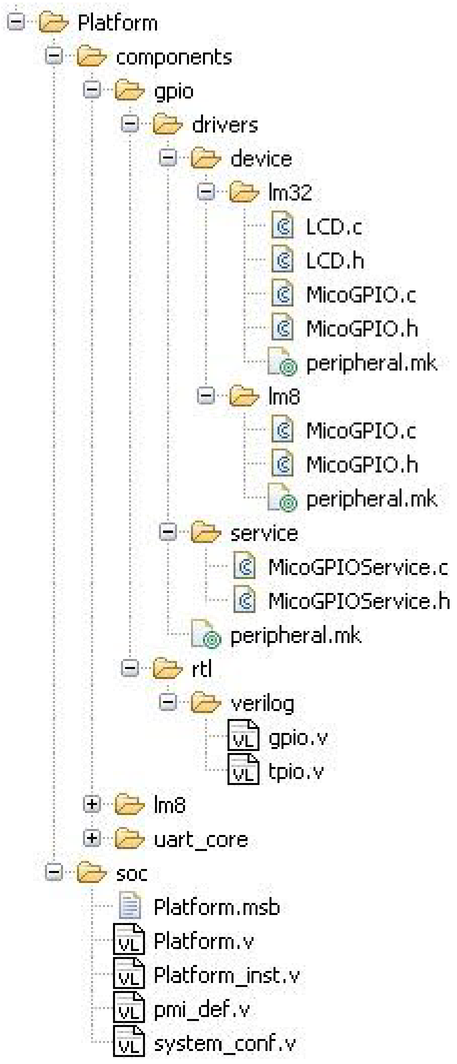
Figure 65 shows the directory structure for LatticeMico GPIO component as part of the components folder under the LatticeMico8 installation folder.

As part of platform generation (Platform in the current context), MSB generates the directory structure by copying the relevant RTL and device-driver directories under the components folder. In Figure 65, this example component folder is components\gpio. If you compare the directory structures shown in Figure 65 and Figure 66, the .xml files are not copied across directories. Instead, the component description file contents are contained in the .msb file (Platform.msb).

It is this .msb file, Platform.msb, that the C/C++ SPE managed build inspects to identify the relevant software components, that is, the structures for DDStructs.c and declarations in DDStructs.h, on the basis of the component configuration defined in the .msb file.

In addition, since the .msb file contains references to the GPIO component, it inspects the GPIO component's drivers folder in the directory structure created by MSB (shown in Figure 66). It copies all the sources and header files that it encounters in the drivers folder to the software project's platform library folder, as shown earlier in Figure 54.

Figure 66: Directory Structure Created by MSB



The C/C++ SPE managed build also inspects the GPIO component's drivers folder for peripheral.mk. If it finds a peripheral.mk file, it includes this makefile in the application build's drivers.mk and the platform library's drivers.mk file. These drivers.mk makefiles identify the sources that must be built as part of the platform library build and those that must be built as part of the application build.

The C/C++ SPE managed build process copies the files found in the drivers folder, other than the makefile, into the platform library folder of the project being built.

Though peripheral.mk is a standard makefile, it must contain only that information that is absolutely necessary. It cannot not use any other symbols or define other rules. Figure 67 shows the LatticeMico GPIO 's peripheral.mk makefile.

Figure 67: GPIO Makefile

```
---------------------------------------------------------
# Identify source-paths for this device's driver-sources,
# compiled when building the library
#---------------------------------------------------------
LIBRARY_C_SRCS += MicoGPIO.c

LIBRARY_ASM_SRCS +=
```

Here is a comprehensive list of variables that can be used in the peripheral.mk makefile:

- LIBRARY_C_SRCS

  Use this variable to identify C sources that must be built as part of the library build process.

- LIBRARY_ASM_SRCS

  Use this variable to identify assembly source files (with an .s or .S file extension) that must be built as part of the library build process.

- APP_ASM_SRCS

  Use this variable to identify assembly source files (with an .s or .S file extension) that must be built as part of the application build process.

- APP_C_SRCS

  Use this variable to identify C sources that must be built as part of the application build process.

Note

Ensure that you add the "+=" symbols to your code for the keywords just shown , as demonstrated in the LatticeMico GPIO makefile example in Figure 67. The C/C++ SPE build process generates only those components' peripheral.mk files that have a corresponding component instance information structure.

Tips on Developing Software for LatticeMico8

This chapter provides a primer on developing software for LatticeMico8. The following sections describe the GNU GCC compiler toolchain that has been customized for LatticeMico8, the Lattice-developed built-in functions/macros available to users developing in ASM or C, and the general programming model that must be followed while developing software for LatticeMico8.

GNU Toolchain

The GNU GCC compiler toolchain has been customized for the LatticeMico8 microcontroller. It contains the standard GNU GCC executable utilities, such as objdump, gcc, and ld. This compiler toolchain can be used by a software developer to compile applications created in C, ASM, or C/ASM. The LatticeMico8 C SPE is tightly integrated with this GNU GCC compiler toolchain. This allows the customer to focus primarily on software development and leaving the process of compiling the software and linking to system libraries to the compiler toolchain and the C SPE. The following sections describe some of the coding limitations imposed by the compiler toolchain and LatticeMico8 on the software developer, and the various built-in functions/macros that can be used by the developer.

Limitations

The LatticeMico8 GNU GCC compiler toolchain is not a full-featured port, primarily due to architectural limitations of LatticeMico8. From a software developer's perspective this translates in to the following C coding limitations.

1. No language frontends other than C are supported.
2. No floating point library.
3. No support for shared libraries.

4. No support for position-independent executables.
5. No support for *function pointers*. This is because LatticeMico8 does not support indirect (or register-based) jumps.
6. Limitations on the size of C/ASM applications due to short branch targets. This is because all branches and procedure calls in LatticeMico8 are relative to the current position and the maximum offset is +/-2K (i.e., a branch can go to 2K instructions backwards or forwards). As a result, if the compiler (during the link phase) determines that a branch or procedure call want to jump to a location that is larger than +/-2K, it will not create an application executable and exit with link errors.
7. Maximum depth of nested functions in a C/ASM application is limited to 8, 16, or 32. This is because the procedure call stack of LatticeMico8 is implemented in hardware and is therefore finite.
8. No support for C Standard Library. This is because LatticeMico8 GCC port does not contain a C Standard Library implementation such as Newlib, libc, or glibc. This means that the developer cannot expect built-in implementations for functions such as I/O (printf, putc, getc, etc.), string manipulation (strcpy, etc.), utility functions (malloc, free, etc.), etc.

Built-in Functions

LatticeMico8's port of the GNU GCC compiler toolchain also implements optimized (ASM) versions of common arithmetic and logical operations. Unless indicated, these optimized implementations are automatically used by the compiler toolchain when required during C code compiling. In addition, the developer can manually invoke these functions from ASM code. These ASM functions are shown in Table 5.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__ashlqi3	Input Arguments 1. R0 – Value to be shifted 2. R1 – Shift amount Output Arguments 1. R0 – Shifted Value	Shift R0 left by R1 bits
__ashlhi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – MSB of value to be shifted 3. R2 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – MSB of shifted value	Shift {R1, R0} left by R2 bits

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__ashlsi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – 2nd byte of value to be shifted 3. R3 – 3rd byte of value to be shifted 4. R4 – MSB of value to be shifted 3. R5 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – 2nd byte of shifted value 3. R3 – 3rd byte of shifted value 4. R4 – MSB of shifted value	Shift {R3, R2, R1, R0} left by R5 bits
__ashrqi3	Input Arguments 1. R0 – Value to be shifted 2. R1 – Shift amount Output Arguments 1. R0 – Shifted Value	Shift R0 right (arithmetically) by R1 bits
__ashrhi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – MSB of value to be shifted 3. R2 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – MSB of shifted value	Shift {R1, R0} right (arithmetically) by R2 bits
__ashrsi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – 2nd byte of value to be shifted 3. R3 – 3rd byte of value to be shifted 4. R4 – MSB of value to be shifted 3. R5 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – 2nd byte of shifted value 3. R3 – 3rd byte of shifted value 4. R4 – MSB of shifted value	Shift {R3, R2, R1, R0} right (arithmetically) by R5 bits
__lshrqi3	Input Arguments 1. R0 – Value to be shifted 2. R1 – Shift amount Output Arguments 1. R0 – Shifted Value	Shift R0 right (logical) by R1 bits

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__lshrhi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – MSB of value to be shifted 3. R2 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – MSB of shifted value	Shift {R1, R0} right (logical) by R2 bits
__lshrsi3	Input Arguments 1. R0 – LSB of value to be shifted 2. R1 – 2nd byte of value to be shifted 3. R3 – 3rd byte of value to be shifted 4. R4 – MSB of value to be shifted 3. R5 – Shift amount Output Arguments 1. R0 – LSB of shifted value 2. R1 – 2nd byte of shifted value 3. R3 – 3rd byte of shifted value 4. R4 – MSB of shifted value	Shift {R3, R2, R1, R0} right (logical) by R5 bits
__clzqi2	Input Arguments 1. R0 – Value Output Arguments 1. R0 – Number of leading zeros	Count leading zeroes in R0
__clzhi2	Input Arguments 1. R0 – LSB of value 2. R1 – MSB of value Output Arguments 1. R0 – Number of leading zeros	Count leading zeroes in {R1, R0}
__clzsi2	Input Arguments 1. R0 – LSB of value 2. R1 – 2nd byte of value 3. R2 – 3rd byte of value 4. R3 – MSB of value Output Arguments 1. R0 – Number of leading zeros	Count leading zeroes in {R3, R2, R1, R0}

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__ctzqi2	Input Arguments 1. R0 – Value Output Arguments 1. R0 – Number of leading zeros	Count trailing zeroes in R0
__ctzhi2	Input Arguments 1. R0 – LSB of value 2. R1 – MSB of value Output Arguments 1. R0 – Number of leading zeros	Count trailing zeroes in {R1, R0}
__ctzsi2	Input Arguments 1. R0 – LSB of value 2. R1 – 2nd byte of value 3. R2 – 3rd byte of value 4. R3 – MSB of value Output Arguments 1. R0 – Number of leading zeros	Count trailing zeroes in {R3, R2, R1, R0}
__mulqi3	Input Arguments 1. R0 – Multiplier 2. R1 – Multiplicand Output Arguments 1. R0 – Result	Multiply R0 by R1 and return result in R0. NOTE: R2 is used as a temporary.
__mulhi3	Input Arguments 1. R0 – LSB of Multiplier 2. R1 – MSB of Multiplier 3. R2 – LSB of Multiplicand 4. R3 – MSB of Multiplicand Output Arguments 1. R0 – LSB of result 2. R1 – MSB of result	Multiply {R1, R0} by {R3, R2} and return result in {R1, R0}. NOTE: R4 and R5 are used as temporaries.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__mulsi3	Input Arguments 1. R0 – LSB of Multiplier 2. R1 – 2nd byte of Multiplier 3. R2 – 3rd byte of Multiplier 4. R3 – MSB of Multiplier 5. R4 – LSB of Multiplicand 6. R5 – 2nd byte of Multiplicand 7. R6 – 3rd byte of Multiplicand 8. R7 – MSB of Multiplicand Output Arguments 1. R0 – LSB of result 2. R1 – 2nd byte of result 3. R2 – 3rd byte of result 4. R3 – MSB of result	Multiply {R3, R2, R1, R0} by {R7, R6, R5, R4} and return result in {R3, R2, R1, R0}. NOTE: R12, R15 are used as temporaries in small and medium memory models, while R12, R31 are used as temporaries in large memory model.
__udivqi3	Input Arguments 1. R0 – Dividend 2. R1 – Divisor Output Arguments 1. R0 – Quotient 2. R2 – Remainder	Divide (unsigned) R0 by R1. Place the quotient in R0 and remainder in R2. NOTE: R3 and R4 are used as temporaries.
__umodqi3	Input Arguments 1. R0 – Dividend 2. R1 – Divisor Output Arguments 1. R2 – Modulus	Modulus of R0 by R1. Place the modulus in R2. NOTE: R3 and R4 are used as temporaries.
__divqi3	Input Arguments 1. R0 – Dividend 2. R1 – Divisor Output Arguments 1. R0 – Quotient 2. R2 – Remainder	Divide (signed) R0 by R1. Place the quotient in R0 and remainder in R2. NOTE: R3, R4 and R5 are used as temporaries.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__modqi3	Input Arguments 1. R0 – Dividend 2. R1 – Divisor Output Arguments 1. R2 – Modulus	Modulus of R0 by R1. Place the modulus in R2. NOTE: R3, R4 and R5 are used as temporaries.
__udivhi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – MSB of Dividend 3. R2 – LSB of Divisor 4. R3 – MSB of Divisor Output Arguments 1. R0 – LSB of Quotient 2. R1 – MSB of Quotient 3. R4 – LSB of Remainder 4. R5 – MSB of Remainder	Divide (unsigned) {R1, R0} by {R3, R2}. Place the quotient in {R1, R0} and remainder in {R5, R4}. NOTE: R6, R7 and R12 are used as temporaries.
__umodhi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – MSB of Dividend 3. R2 – LSB of Divisor 4. R3 – MSB of Divisor Output Arguments 1. R4 – LSB of Modulus 2. R5 – MSB of Modulus	Modulus of {R1, R0} by {R3, R2}. Place the modulus in {R5, R4}. NOTE: R6, R7 and R12 are used as temporaries.
__divhi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – MSB of Dividend 3. R2 – LSB of Divisor 4. R3 – MSB of Divisor Output Arguments 1. R0 – LSB of Quotient 2. R1 – MSB of Quotient 3. R4 – LSB of Remainder 4. R5 – MSB of Remainder	Divide (signed) {R1, R0} by {R3, R2}. Place the quotient in {R1, R0} and remainder in {R5, R4}. NOTE: R6, R7, R12 and R15 are used as temporaries in small and medium memory models. R6, R7, R12 and R31 are used as temporaries in large memory model.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__modhi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – MSB of Dividend 3. R2 – LSB of Divisor 4. R3 – MSB of Divisor Output Arguments 1. R4 – LSB of Modulus 2. R5 – MSB of Modulus	Modulus of {R1, R0} by {R3, R2}. Place the modulus in {R5, R4}. NOTE: R6, R7, R12 and R15 are used as temporaries in small and medium memory models. R6, R7, R12 and R31 are used as temporaries in large memory model.
__udivsi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – 2nd byte of Dividend 3. R2 – 3rd byte of Dividend 4. R3 – MSB of Dividend 5. R4 – LSB of Divisor 6. R5 – 2nd byte of Divisor 7. R6 – 3rd byte of Divisor 4. R7 – MSB of Divisor Output Arguments 1. R0 – LSB of Quotient 2. R1 – 2nd byte of Quotient 3. R2 – 3rd byte of Quotient 4. R3 – MSB of Quotient 5. R4 – LSB of Remainder 6. R5 – 2nd byte of Remainder 7. R6 – 3rd byte of Remainder 8. R7 – MSB of Remainder	Divide (unsigned) {R3, R2, R1, R0} by {R7, R6, R4, R4}. Place the quotient in {R3, R2, R1, R0} and remainder in {R7, R6, R5, R4}. NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__umodsi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – 2nd byte of Dividend 3. R2 – 3rd byte of Dividend 4. R3 – MSB of Dividend 5. R4 – LSB of Divisor 6. R5 – 2nd byte of Divisor 7. R6 – 3rd byte of Divisor 4. R7 – MSB of Divisor Output Arguments 1. R0 – LSB of Modulus 2. R1 – 2nd byte of Modulus 3. R2 – 3rd byte of Modulus 4. R3 – MSB of Modulus	Modulus of {R3, R2, R1, R0} by {R7, R6, R4, R4}. Place the modulus in {R3, R2, R1, R0}. NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.
__divsi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – 2nd byte of Dividend 3. R2 – 3rd byte of Dividend 4. R3 – MSB of Dividend 5. R4 – LSB of Divisor 6. R5 – 2nd byte of Divisor 7. R6 – 3rd byte of Divisor 4. R7 – MSB of Divisor Output Arguments 1. R0 – LSB of Quotient 2. R1 – 2nd byte of Quotient 3. R2 – 3rd byte of Quotient 4. R3 – MSB of Quotient 5. R4 – LSB of Remainder 6. R5 – 2nd byte of Remainder 7. R6 – 3rd byte of Remainder 8. R7 – MSB of Remainder	Divide (signed) {R3, R2, R1, R0} by {R7, R6, R4, R4}. Place the quotient in {R3, R2, R1, R0} and remainder in {R7, R6, R5, R4}. NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

Table 5: ASM Functions

Name	Input and Output Arguments	Description
__modsi3	Input Arguments 1. R0 – LSB of Dividend 2. R1 – 2nd byte of Dividend 3. R2 – 3rd byte of Dividend 4. R3 – MSB of Dividend 5. R4 – LSB of Divisor 6. R5 – 2nd byte of Divisor 7. R6 – 3rd byte of Divisor 4. R7 – MSB of Divisor Output Arguments 1. R0 – LSB of Modulus 2. R1 – 2nd byte of Modulus 3. R2 – 3rd byte of Modulus 4. R3 – MSB of Modulus	Modulus of {R3, R2, R1, R0} by {R7, R6, R4, R4}. Place the modulus in {R3, R2, R1, R0}. NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

__cmpqi2	Input Arguments
	1. R0 – Comparison value 1
	2. R1 – Comparison value 2
	
	Output Arguments
		Signed compare or R0 with R1.
		CF = 1 if R0 >= R1.
		ZF = 1 if R0 == R1.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

__cmphi2	Input Arguments
	1. R0 – LSB of comparison value 1
	2. R1 – MSB of comparison value 1
	3. R2 – LSB of comparison value 2
	4. R3 – MSB of comparison value 2
	
	Output Arguments
		Signed compare or {R1, R0} with {R3, R2}.
		CF = 1 if {R1, R0} >= {R3, R2}.
		ZF = 1 if {R1, R0} == {R3, R2}.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

Table 5: ASM Functions

Name	Input and Output Arguments	Description

__cmpsi2	Input Arguments
	1. R0 – LSB of comparison value 1
	2. R1 – 2nd byte of comparison value 1
	3. R2 – 3rd byte of comparison value 1
	4. R3 – MSB of comparison value 1
	5. R4 – LSB of comparison value 2
	6. R5 – 2nd byte of comparison value 2
	7. R6 – 3rd byte of comparison value 2
	8. R7 – MSB of comparison value 2
	
	Output Arguments
		Signed compare or {R3, R3, R1, R0} with {R7, R6, R5, R4}.
		CF = 1 if {R3, R3, R1, R0} >= {R7, R6, R5, R4}.
		ZF = 1 if {R3, R3, R1, R0} == {R7, R6, R5, R4}.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

__ucmpqi2	Input Arguments
	1. R0 – Comparison value 1
	2. R1 – Comparison value 2
	
	Output Arguments
		Unsigned compare or R0 with R1.
		CF = 1 if R0 >= R1.
		ZF = 1 if R0 == R1.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

__ucmphi2	Input Arguments
	1. R0 – LSB of comparison value 1
	2. R1 – MSB of comparison value 1
	3. R2 – LSB of comparison value 2
	4. R3 – MSB of comparison value 2
	
	Output Arguments
		Unsigned compare or {R1, R0} with {R3, R2}.
		CF = 1 if {R1, R0} >= {R3, R2}.
		ZF = 1 if {R1, R0} == {R3, R2}.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model.

__ucmpsi2	Input Arguments
	1. R0 – LSB of comparison value 1
	2. R1 – 2nd byte of comparison value 1
	3. R2 – 3rd byte of comparison value 1
	4. R3 – MSB of comparison value 1
	5. R4 – LSB of comparison value 2
	6. R5 – 2nd byte of comparison value 2
	7. R6 – 3rd byte of comparison value 2
	8. R7 – MSB of comparison value 2
	
	Output Arguments
		Unsigned compare or {R3, R3, R1, R0} with {R7, R6, R5, R4}.
		CF = 1 if {R3, R3, R1, R0} >= {R7, R6, R5, R4}.
		ZF = 1 if {R3, R3, R1, R0} == {R7, R6, R5, R4}.
		
		NOTE: R12 and R15 are used as temporaries in small and medium memory models. R12 and R31 are used as temporaries in large memory model

Built-in Macros

The LatticeMico8 GNU GCC compiler toolchain also implements the following macros that can be used directly in ASM to extract bytes from an integer (2 bytes) or long (4 bytes).

1. _lo(X) – This macro extracts the least-significant byte of an integer or long.
2. _hi(X) – This macro extracts the most-significant byte of an integer, or 2nd byte of a long.
3. _higher(X) – This macro extracts the 3rd byte of a long.
4. _highest(X) – This macro extracts the most-significant byte of a long.

Usage example: The following ASM puts value 0xad in to register R0.

```
movi     r0, _higher(0xdeadbeef)
```

Using I/O (Peripheral) Instructions

LatticeMico8 implements special instructions to access the I/O or Peripheral region. These instructions are *import*, *importi*, *export*, *exporti*. There are two ways in which the developer can force the compiler generate these instructions within the compiled application:

1. Inlined Assembly – The developer can use inlined assembly to explicitly code these instructions.
2. Builtin Functions – The functions shown in Table 6 can be invoked by the developer in C/ASM code when he needs to access an address in the Peripheral region.

Table 6: Builtin Functions

Function	Effect
void __builtin_export (char value, size_t address)	Generates an export or exporti instruction
char __builtin_import (size_t address)	Generated an import or importi instruction. The result of the import instruction is the returned value.

Note: The size of size_t type reflects the size of pointers and is dictated by the memory mode used. Refer to Table 7 for the number of bytes needed for a pointer

Programming Model

This section describes the LatticeMico8 programming model, including data types, calling sequence, and interrupt convention.

Data Representation

The LatticeMico8 microcontroller supports the data types listed in Table 7.

Table 7: LatticeMico8 Data Types

		Size in Memory Model			Alignment in Memory Model		
Type	**C Type**	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
Integer	Signed char	1	1	1	1	1	1
Integer	Unsigned char	1	1	1	1	1	1
Integer	Signed short	2	2	2	2	2	2
Integer	Unsigned short	2	2	2	2	2	2
Integer	Signed int	2	2	2	2	2	2
Integer	Unsigned int	2	2	2	2	2	2
Integer	Signed long	4	4	4	4	4	4
Integer	Unsigned long	4	4	4	4	4	4
Integer	Unsigned long long	4	4	4	4	4	4
Pointer	Any-type*	1	2	4	1	2	4
Floating-Point	Float	4	4	4	4	4	4
Floating-Point	Double	4	4	4	4	4	4
Floating-Point	Long double	4	4	4	4	4	4

Procedure Caller-Callee Convention

This section describes the standard function calling sequence, including stack frame layout, register usage, and parameter passing. The standard calling sequence requirements apply only to global functions; however, it is recommended that all functions use the standard calling sequence.

Register Usage

The register usage model shown in Table 8 is used by the LatticeMico8 Compiler. It must be used by developers who are writing ASM code that will be compiled into an executable using the LatticeMico8 compiler.

Table 8: Register Usage (SP – Stack Pointer, FP – Frame Pointer, PP – Page Pointer)

	Preserved Across Functions			Usage		
Register	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
R0	N	N	N	Arg 0/Return 0	Arg 0/Return 0	Arg 0/Return 0
R1	N	N	N	Arg 1/Return 1	Arg 1/Return 1	Arg 1/Return 1

Table 8: Register Usage (SP – Stack Pointer, FP – Frame Pointer, PP – Page Pointer) (Continued)

	Preserved Across Functions			Usage		
Register	**Small**	**Medium**	**Large**	**Small**	**Medium**	**Large**
R2	N	N	N	Arg 2/Return 2	Arg 2/Return 2	Arg 2/Return 2
R3	N	N	N	Arg 3/Return 3	Arg 3/Return 3	Arg 3/Return 3
R4	N	N	N	Arg 4	Arg 4	Arg 4
R5	N	N	N	Arg 5	Arg 5	Arg 5
R6	N	N	N	Arg 6	Arg 6	Arg 6
R7	N	N	N	Arg 7	Arg 7	Arg 7
R8	Y	Y	Y		Fixed – SP	
R9	Y	Y	Y		Fixed – SP	
R10	N	Y	N		Fixed – FP	
R11	N	Y	N		Fixed – FP	
R12	N	N	N			
R13	N	N	N		Fixed – PP	Fixed – PP
R14	Y	Y	N	Fixed – SP		Fixed – PP
R15	Y	N	N	Fixed – FP		Fixed – PP
R16	Y	Y	Y			
R17	Y	Y	Y			
R18	Y	Y	Y			
R19	Y	Y	Y			
R20	Y	Y	N			
R21	Y	Y	N			
R22	Y	Y	N			
R23	Y	Y	N			
R24	Y	Y	Y			Fixed – SP
R25	Y	Y	Y			Fixed – SP
R26	Y	Y	Y			Fixed – SP
R27	Y	Y	Y			Fixed – SP
R28	N	N	Y			Fixed – FP
R29	N	N	Y			Fixed – FP
R30	N	N	Y			Fixed – FP
R31	N	N	Y			Fixed – FP

Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 9 shows the stack frame organization.

Table 9: Stack Frame Layout

FP-relative Position	SP-relative Position	Contents	Frame
FP + (M – 6) . . . FP + 0	SP + (N + M + 4) . . . SP + (N + 4)	Function Argument Byte M . . . Function Argument Byte 6	Previous
FP – 1 FP – 2 FP – 3 FP – 4	SP + (N + 3) SP + (N + 2) SP + (N + 1) SP + N	Previous FP (byte 3) Previous FP (byte 2) Previous FP (byte 1) Previous FP (byte 0)	Current
FP – 5 . . . FP – (N + 5)	SP + (N – 1) . . . SP + 0	Local Variable N . . . Local Variable 0	
FP – (N + 6) . . . FP – (N + 37)	SP – 1 . . . SP – 32	Red Zone Area – Start Red Zone Area – End	Future

Parameter Passing

There are two mechanisms to pass data between functions when the LatticeMico8 GNU compiler toolchain is used. The first mechanism is via registers R0 – R7. Since each register is a byte wide, the maximum number of bytes that can be passed via registers is 8 bytes. If the number of bytes to be passed exceeds 8 bytes, then the remaining bytes are passed on the stack. The return value is passed in registers R0 – R3. The following rules are followed:

1. The argument variables are processed in order when assigning them argument register(s). For example, variable ‘x’ is assigned R0 and ‘y’ is assigned R1 when calling the following function:
   void foo (char x, char y)
2. If a variable cannot be completely allocated to argument registers, it is passed on the stack. For example, variable ‘z’ is passed on the stack since variables ‘x’ and ‘y’ use up registers R0 – R5 and variable ‘z’ requires 4 registers (only R6 and R7 are available).
   void foo (long x, int y, long z)
3. Each register starting with R0 must be assigned to a valid argument variable prior to moving on to the next assignment. This allows maximum number of argument variables to be fit in to the 8 argument registers R0 – R7 regardless of the individual sizes. For example, variable ‘x’ is assigned

R0, ‘y’ is assigned R1 and R2, and ‘z’ is assigned R3, R4, R5, and R6.
void foo (char x, int y, long z)

4. Multi-byte variables are assigned registers in little-endian format. For example, least significant byte of variable ‘x’ is assigned R0 and most-significant byte is assigned R3.
void foo (long x)

5. Small structures and unions are passed in argument registers if all the elements of the structure/union fits within the 8 argument registers. Otherwise, the entire structure/union is passed on the stack.

6. If the returned variable is a structure/union and the size exceeds 4 bytes, then the value is returned in memory, pointed by the “invisible” first function argument.

Interrupt Convention

Interrupts are managed on an interrupt stack that is separate from the normal program stack. In the event of an interrupt, the stack pointer is switched to the top of the interrupt stack minus 32 where all the registers are saved according to the convention shown in Table 10.

Table 10: Interrupt Frame Layout

	Register		
Position	**Small**	**Medium**	**Large**
Top of Interrupt Stack – 1	R11	R9	R11
Top of Interrupt Stack – 2	R10	R8	R10
Top of Interrupt Stack – 3	R31	R31	R31
Top of Interrupt Stack – 4	R30	R30	R30
Top of Interrupt Stack – 5	R29	R29	R29
Top of Interrupt Stack – 6	R28	R28	R28
Top of Interrupt Stack – 7	R7	R7	R7
Top of Interrupt Stack – 8	R6	R6	R6
Top of Interrupt Stack – 9	R5	R5	R5
Top of Interrupt Stack – 10	R4	R4	R4
Top of Interrupt Stack – 11	R3	R3	R3
Top of Interrupt Stack – 12	R2	R2	R2
Top of Interrupt Stack – 13	R1	R1	R1
Top of Interrupt Stack – 14	R0	R0	R0

The first four bytes of the scratchpad memory area are reserved to set up the interrupt stack in the event of an interrupt. The compiler will generate code to setup the interrupt stack frame suitable for an interrupt handler in the prologue of the function that has the "interrupt" attribute. For this interrupt handler to link correctly, it must be named "__IRQ". An example is shown in Figure 68.

Figure 68: LatticeMico8 Interrupt Handler

```
__attribute ((interrupt)) __IRQ (void)
{
  // user's interrupt handling code
}
```

Chapter 6

Software Development Utilities

This chapter describes the software development utilities in the LatticeMico8 GNU C/C++ tool chain that are used to accomplish tasks, even though they are not visible in the graphical user interface. This tool chain includes general-purpose software development utilities, such as a command-line interface, that incorporate UNIX shell capabilities on a PC platform. In addition, the tool chain consists of LatticeMico System-specific utilities for generating and debugging software code.

Build Tools

This section explains the GCC tools used for building software programs for LatticeMico8 and the build flow in the C/C++ Software Project Environment (SPE). This section also includes commonly used parameters for the tools, along with LM8-specific build options. References to the GNU tool chain Web site are provided here to supplement your information on this open-source development tool.

If there are any issues or problems with any of these tools, report them at the http://www.sourceware.org/bugzilla/ Web site.

lm8-elf-ar

The lm8-elf-ar utility generates an archive from the given input object files.

Refer to the GCC and GNU Binary Utilities documentation for more information.

Usage

```
lm8-elf-ar [emulation_options] [-]{options}[modifiers]
[member_name] [count] archive_file_name file_name ...

lm8-elf-ar -M [<mri_script]
```

Options can be any of the options listed in Table 11.

Table 11: lm8-elf-ar Options

Options	Description
d	Deletes files from the archive.
m[ab]	Moves files in the archive.
p	Prints files found in the archive.
q[f]	Appends files to the archive.
r[ab][f][u]	Replaces existing files or inserts new files into the archive.
t	Displays contents of the archive.
x[o]	Extracts files from the archive.

Modifiers can be any of the command-specific or generic modifiers listed in Table 12 or Table 13.

Table 12: lm8-elf-ar Command-Specific Modifiers

Options	Description
[a]	Puts files after [*member_name*].
[b]	Puts files before [*member_name*] (same as [i]).
[N]	Uses instance [*count*] of name.
[f]	Truncates inserted file names.
[P]	Uses full path names when matching.
[o]	Preserves original dates.
[u]	Only replaces files that are newer than current archive contents.

Table 13: lm8-elf-ar Generic Modifiers

Options	Description
[c]	Does not warn if the library had to be created.
[s]	Creates an archive index (cf. ranlib)
[S]	Does not build a symbol table.

Table 13: lm8-elf-ar Generic Modifiers

[v]	Is verbose.
[V]	Displays the version number.

The lm8-elf-ar utility has no emulation-specific options.

The lm8-elf-ar utility supports the following targets: elf32-lm8, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary ihex.

lm8-elf-as

The lm8-elf-as utility is the assembler utility. It takes as input an assembler source (.s) file and generates a relocatable object (.o) file.

Usage

```
lm8-elf-as [options] [asmfile...]
```

where *options* can be one or more of the options shown in Table 14.

Table 14: lm8-elf-as Options

Options	Description
-a[sub-option...]	Turns on listings.
Sub-options [default hls]:	
c	Omits false conditionals.
d	Omits debugging directives.
h	Includes high-level source.
l	Includes assembly.
m	Includes macro expansions.
n	Omits forms processing.
s	Includes symbols.
=FILE	Lists to FILE (must be last sub-option).
--alternate	Initially turns on alternate macro syntax.
-D	Produces assembler debugging messages.
--defsym SYM=VAL	Defines symbol SYM to given value.
--execstack	Requires executable stack for this object.
--noexecstack	Does not require executable stack for this object.

Table 14: lm8-elf-as Options

-f	Skips white space and comment preprocessing.
-g --gen-debug	Ignored.
--gstabs	Ignored.
--gstabs+	Ignored.
--gdwarf-2	Ignored.
--help	Shows these option descriptions and exits.
--target-help	Shows target-specific options.
-I DIR	Adds DIR to search list for .include directives.
-J	Does not warn about signed overflow.
-K	Warns when differences altered for long displacements.
-L,--keep-locals	Keeps local symbols (for example, starting with “L”).
-M,--mri	Assembles in MRI compatibility mode.
--MD FILE	Writes dependency information in FILE (default is none).
-nocpp	Ignored.
-o OBJFILE	Names the object-file output OBJFILE (default a.out).
-R	Folds data section into text section.
--statistics	Prints various measured statistics from execution.
--strip-local-absolute	Strips local absolute symbols.
--traditional-format	Uses same format as native assembler when possible.
--version	Prints assembler version number and exit.
-W --no-warn	Suppresses warnings.
--warn	Does not suppress warnings.
--fatal-warnings	Treats warnings as errors.
--itbl INSTTBL	Extends instruction set to include instructions matching the specifications defined in file INSTTBL.
-w	Ignored.
-X	Ignored.
-Z	Generates object file even after errors.

Table 14: lm8-elf-as Options

--listing-lhs-width	Sets the width in words of the output data column of the listing.
--listing-lhs-width2	Sets the width in words of the continuation lines of the output data column; ignored if smaller than the width of the first line.
--listing-rhs-width	Sets the maximum width in characters of the lines from the source file.
--listing-cont-lines	Sets the maximum number of continuation lines used for the output data column of the listing.

lm8-elf-gcc

The lm8-elf-gcc utility is the compiler utility. It compiles a C code (.c) file into a relocatable object (.o) file. It can call the linker as well, depending on the file extension.

Usage

```
lm8-elf-gcc [options] file...
```

where *options* can be one or more of the options shown in Table 15.

Table 15: lm8-elf-gcc Options

Option	Description
-pass-exit-codes	Exits with highest error code from a phase.
--help	Displays these option descriptions.
--target-help	Displays target-specific command-line options.
'-v --help'	Displays command-line options of sub-processes.
-dumpspecs	Displays all of the built-in specification strings.
-dumpversion	Displays the version of the compiler.
-dumpmachine	Displays the compiler's target microprocessor.
-print-search-dirs	Displays the directories in the compiler's search path.
-print-libgcc-file-name	Displays the name of the compiler's companion library.
-print-file-name=<*lib*>	Displays the full path to the <*lib*> library.

Table 15: lm8-elf-gcc Options

Option	Description
-print-prog-name=<*prog*>	Displays the full path to the <*prog*> compiler component .
-print-multi-directory	Displays the root directory for versions of libgcc.
-print-multi-lib	Displays the mapping between command-line options and multiple library search directories.
-print-multi-os-directory	Displays the relative path to OS libraries.
-Wa,<*options*>	Passes comma-separated <*options*> to the assembler.
-Wp,<*options*>	Passes comma-separated <*options*> to the preprocessor.
-Wl,<*options*>	Passes comma-separated <*options*> to the linker.
-Xassembler <*arg*>	Passes <arg> to the assembler.
-Xpreprocessor <*arg*>	Passes <*arg*> to the preprocessor.
-Xlinker <arg>	Passes <*arg*> to the linker.
-save-temps	Does not delete intermediate files.
-pipe	Uses pipes rather than intermediate files.
-time	Times the execution of each sub-process.
-specs=<*file*>	Overrides built-in specifications with the contents of <*file*>.
-std=<*standard*>	Assumes that the input sources are for <*standard*>.
-B <*directory*>	Adds <*directory*> to the compiler's search paths.
-b <*machine*>	Runs GCC for target <*machine*>, if installed.
-V <*version*>	Runs GCC version number <*version*>, if installed.
-v	Displays the programs invoked by the compiler.
-###	Like -v but options quoted and commands not executed.
-E	Preprocesses only; does not compile, assemble, or link.
-S	Compiles only; does not assemble or link.
-c	Compiles and assembles but does not link.

Table 15: lm8-elf-gcc Options

Option	Description
-o *<file>*	Places the output into *<file>*.
lm8-specific Options	**Description**
-mcmodel=small	Generates only 8-bit addresses for any I/O or memory access.
-mcmodel=medium	Generates only 16-bit addresses for an I/O or memory access (default when no switch is used).
-mcmodel=large	Generates 32-bit addresses for an I/O or memory access.
-mint8	The common 'int' type is 8 bits instead of the standard 16 bits.
-m16regs	Use registers 0 through 15 only.
-mcall-stack-size=<value>	Set the size of call stack to a user-defined value.
-mcall-prologues	Don't inline the function epilogue/prologue.

Options starting with -g, -f, -m, -O, -W, or --param are automatically passed on to the various subprocesses invoked by lm8-elf-gcc. In order to pass other options on to these processes, the -W<*letter*> options must be used. Report bugs for this tool to the http://www.sourceware.org/bugzilla/ Web site.

lm8-elf-ld

The lm8-elf-ld utility is the link-editor utility. It takes a single or multiple object (.o) files as input, as well as library archives (.a), and produces the final executable (.elf) file.

Usage

```
lm8-elf-ld [options] file...
```

where *options* can be one or more of the options shown in Table 16.

Table 16: lm8-elf-ld Options

Options	Description
a KEYWORD	Shares library control for HP/UX compatibility.
-A ARCH, --architecture ARCH	Sets architecture.
-b TARGET, --format TARGET	Specifies target for following input files.

Table 16: lm8-elf-ld Options

-c FILE, --mri-script FILE	Reads MRI format linker script.
-d, -dc, -dp	Forces common symbols to be defined.
-e ADDRESS, --entry ADDRESS	Sets start address.
-E, --export-dynamic	Exports all dynamic symbols.
-EB	Links big-endian objects.
-EL	Links little-endian objects.
-f SHLIB, --auxiliary SHLIB	Specifies an auxiliary filter for shared object symbol table.
-F SHLIB, --filter SHLIB	Specifies filter for shared object symbol table.
-g	Ignored.
-G SIZE, --gpsize SIZE	Specifies small data size (if no size, same as --shared).
-h FILENAME, -soname FILENAME	Sets internal name of shared library.
-I PROGRAM, --dynamic-linker PROGRAM	Sets PROGRAM as the dynamic linker to use.
-l LIBNAME, --library LIBNAME	Searches for *LIBNAME* library.
-L DIRECTORY, --library-path DIRECTORY	Adds DIRECTORY to library search path.
--sysroot=<*DIRECTORY*>	Overrides the default sysroot location.
-m EMULATION	Sets emulation.
-M, --print-map	Prints map file on standard output.
-n, --nmagic	Does not page-align data.
-N, --omagic	Does not page-align data and does not make text read only.
--no-omagic	Page-aligns data and makes text read only.
-o FILE, --output FILE	Sets output file name.
-O	Optimizes output file.
-Qy	Ignored for SVR4 compatibility.
-q, --emit-relocs	Generates relocations in final output.
-r, -i, --relocatable	Generates relocatable output.
-R FILE, --just-symbols FILE	Just links symbols (if directory, same as --rpath).
-s, --strip-all	Strips all symbols.
-S, --strip-debug	Strips debugging symbols.
--strip-discarded	Strips symbols in discarded sections.

Table 16: lm8-elf-ld Options

--no-strip-discarded	Does not strip symbols in discarded sections.
-t, --trace	Traces file opens.
-T FILE, --script FILE	Reads linker script.
-u SYMBOL, --undefined SYMBOL	Starts with undefined reference to SYMBOL.
-unique [=SECTION]	Does not merge input [SECTION | orphan] sections.
-Ur	Builds global constructor/destructor tables.
-v, --version	Prints version information.
-V	Prints version and emulation information.
-x, --discard-all	Discards all local symbols.
-X, --discard-locals	Discards temporary local symbols (default).
--discard-none	Does not discard any local symbols.
-y SYMBOL, --trace-symbol SYMBOL	Traces mentions of SYMBOL.
-Y PATH	Sets default search path for Solaris compatibility.
-(, --start-group	Starts a group.
-), --end-group	Ends a group.
--accept-unknown-input-arch	Accepts input files whose architecture cannot be determined.
--no-accept-unknown-input-arch	Rejects input files whose architecture is unknown following dynamic libraries.
-add-needed	Sets DT_NEEDED tags for DT_NEEDED entries in following dynamic libraries.
--no-add-needed	Does not set DT_NEEDED tags for DT_NEEDED entries in following dynamic libraries.
--as-needed	Only sets DT_NEEDED for following dynamic libraries, if used.
--no-as-needed	Always sets DT_NEEDED for following dynamic libraries.
-assert KEYWORD	Ignored for SunOS compatibility.
-Bdynamic, -dy, -call_shared	Links against shared libraries.
-Bstatic, -dn, -non_shared, -static	Does not link against shared libraries.
-Bsymbolic	Binds global references locally.
--check-sections	Checks section addresses for overlaps (default).

Table 16: lm8-elf-ld Options

--no-check-sections	Does not check section addresses for overlaps.
--cref	Outputs cross reference table.
--defsym SYMBOL=EXPRESSION	Defines a symbol.
--demangle [=STYLE]	Demangles symbol names [using STYLE].
--embedded-relocs	Generates embedded relocations.
--fatal-warnings	Treats warnings as errors.
-fini SYMBOL	Calls SYMBOL at unload time.
--force-exe-suffix	Forces generation of file with .exe suffix.
--gc-sections	Removes unused sections (on some targets).
--no-gc-sections	Does not remove unused sections (default).
--hash-size=<*NUMBER*>	Sets default hash table size close to <*NUMBER*>.
--help	Prints option help.
-init SYMBOL	Calls SYMBOL at load time.
-Map FILE	Writes a map file.
--no-define-common	Does not define common storage.
--no-demangle	Does not demangle symbol names.
--no-keep-memory	Uses less memory and more disk I/O.
--no-undefined	Does not allow unresolved references in object files.
--allow-shlib-undefined	Allows unresolved references in shared libaries.
--no-allow-shlib-undefined	Does not allow unresolved references in shared libraries.
--allow-multiple-definition	Allows multiple definitions.
--no-undefined-version	Does not allow undefined version.
--default-symver	Creates default symbol version.
--default-imported-symver	Creates default symbol version for imported symbols.
--no-warn-mismatch	Does not warn about mismatched input files.
--no-whole-archive	Turns off --whole-archive.
--noinhibit-exec	Creates an output file even if errors occur.
-nostdlib	Only uses library directories specified on the command line.

Table 16: lm8-elf-ld Options

--oformat TARGET	Specifies target of output file.
-qmagic	Ignored for Linux compatibility.
--reduce-memory-overheads	Reduces memory overheads, possibly taking much longer.
--relax	Relaxes branches on certain targets.
--retain-symbols-file FILE	Keeps only symbols listed in FILE.
-rpath PATH	Sets run-time shared library search path.
-rpath-link PATH	Sets link-time shared library search path.
-shared, -Bshareable	Creates a shared library.
-pie, --pic-executable	Creates a position-independent executable.
--sort-common	Sorts common symbols by size.
--sort-section name|alignment	Sorts sections by name or maximum alignment.
--spare-dynamic-tags COUNT	Specifies how many tags to reserve in .dynamic section.
--split-by-file [=SIZE]	Splits output sections every SIZE octets.
--split-by-reloc [=COUNT]	Splits output sections every COUNT relocations.
--stats	Prints memory usage statistics.
--target-help	Displays target specific options.
--task-link SYMBOL	Does task-level linking.
--traditional-format	Uses same format as native linker.
--section-start SECTION=ADDRESS	Sets address of named section.
-Tbss ADDRESS	Sets address of .bss section.
-Tdata ADDRESS	Sets address of .data section.
-Ttext ADDRESS	Sets address of .text section.
--unresolved-symbols=<*method*>	Specifies how to handle unresolved symbols. <*method*> can be ignore-all, report-all, ignore-in-object-files, ignore-in-shared-libs.
--verbose	Outputs lots of information during link.
--version-script FILE	Reads version information script.
--version-exports-section SYMBOL	Takes export symbols list from .exports, using SYMBOL as the version.
--warn-common	Warns about duplicate common symbols.
--warn-constructors	Warns if global constructors and destructors are seen.

Table 16: lm8-elf-ld Options

--warn-multiple-gp	Warns if the multiple GP values are used.
--warn-once	Warns only once per undefined symbol.
--warn-section-align	Warns if start of section changes because of alignment.
--warn-shared-textrel	Warns if shared object has DT_TEXTREL.
--warn-unresolved-symbols	Reports unresolved symbols as warnings.
--error-unresolved-symbols	Reports unresolved symbols as errors.
--whole-archive	Includes all objects from following archives.
--wrap SYMBOL	Uses wrapper functions for SYMBOL.
lm8-elf-ld: supported targets:	elf32-lm8, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex.
lm8-elf-ld: supported emulations:	elf32lm8
lm8-elf-ld: emulation specific options:	No emulation-specific options.
lm8-specific Options	**Description**
-mlm8-elf-small	Generates only 8-bit addresses for any I/O or memory access.
-mlm8-elf-medium	Generates only 16-bit addresses for an I/O or memory access (default when no switch is used).
-mlm8-elf-large	Generates 32-bit addresses for an I/O or memory access.

Report bugs for this tool to the http://www.sourceware.org/bugzilla/ Web site.

lm8-elf-nm

The lm8-elf-nm utility lists symbols in [*files*] (a.out by default).

Usage

```
lm8-elf-nm [options] [files]
```

where *options* can be one or more of the options shown in Table 17.

Table 17: lm8-elf-nm Options

Options	Description
-a, --debug-syms	Displays debugger-only symbols.
-A, --print-file-name	Prints name of the input file before every symbol.

Table 17: lm8-elf-nm Options

-B	Performs same function as --format=bsd.
-C, --demangle[=STYLE]	Decodes low-level symbol names into user-level names. The STYLE, if specified, can be `auto' (the default), `gnu,' `lucid,' `arm,' `hp,' `edg,' `gnu-v3,' `java,' or `gnat.'
--no-demangle	Does not demangle low-level symbol names.
-D, --dynamic	Displays dynamic symbols instead of normal symbols.
--defined-only	Displays only defined symbols.
-e	Ignored.
-f, --format=FORMAT	Uses the output format FORMAT. FORMAT can be `bsd,' `sysv,' or `posix.' The default is `bsd'.
-g, --extern-only	Displays only external symbols.
-l, --line-numbers	Uses debugging information to find a file name and line number for each symbol.
-n, --numeric-sort	Sorts symbols numerically by address.
-o	Performs same function as -A.
-p, --no-sort	Does not sort the symbols.
-P, --portability	Same as --format=posix.
-r, --reverse-sort	Reverse the sense of the sort.
-S, --print-size	Prints size of defined symbols.
-s, --print-armap	Includes index for symbols from archive members.
--size-sort	Sorts symbols by size.
--special-syms	Includes special symbols in the output.
--synthetic	Displays synthetic symbols as well.
-t, --radix=RADIX	Uses RADIX for printing symbol values.
--target=BFDNAME	Specifies the target object format as BFDNAME.
-u, --undefined-only	Displays only undefined symbols.
-X 32_64	Ignored.
-h, --help	Displays this information.
-V, --version	Displays this program's version number.
lm8-elf-nm: supported targets:	elf32-lm8 elf32-little elf32-big srec symbolsrec tekhex binary ihex.

Report bugs to the http://www.sourceware.org/bugzilla/ Web site.

lm8-elf-objcopy

The lm8-elf-objcopy utility copies a binary file, possibly transforming it in the process.

Usage

```
lm8-elf-objcopy [options] in_file [out_file]
```

where *options* can be one or more of the options shown in Table 18.

Table 18: lm8-elf-objcopy Options

Options	Description
-I --input-target <*bfdname*>	Assumes input file is in format <*bfd_name*>.
-O --output-target <*bfdname*>	Creates an output file in format <*bfd_name*>.
-B --binary-architecture <*arch*>	Set sarch of output file, when input is binary.
-F --target <*bfdname*>	Sets both input and output format to <*bfd_name*>.
--debugging	Converts debugging information, if possible.
-p --preserve-dates	Copies modified/access timestamps into the output.
-j --only-section <*name*>	Only copies section <*name*> into the output.
--add-gnu-debuglink=<*file*>	Adds .gnu_debuglink section linking to <*file*>.
-R --remove-section <*name*>	Removes the <*name*> section from the output.
-S --strip-all	Removes all symbol and relocation information.
-g --strip-debug	Removes all debugging symbols and sections.
--strip-unneeded	Removes all symbols not needed by relocations.
-N --strip-symbol <*name*>	Does not copy the <*name*> symbol.
--strip-unneeded-symbol <*name*>	Does not copy the <*name*> symbol unless needed by relocations.
--only-keep-debug	Strips everything but the debug information.
-K --keep-symbol <*name*>	Only copies the <*name*> symbol.
-L --localize-symbol <*name*>	Forces the <*name*> symbol to be marked as a local.
-G --keep-global-symbol <*name*>	Localizes all symbols except <*name*>.
-W --weaken-symbol <*name*>	Forces the <*name*> symbol to be marked as a weak.

Table 18: lm8-elf-objcopy Options

--weaken	Forces all global symbols to be marked as weak.
-w --wildcard	Permits wildcard in symbol comparison.
-x --discard-all	Removes all non-global symbols.
-X --discard-locals	Removes any compiler-generated symbols.
-i --interleave <*number*>	Only copies one out of every <*number*> bytes.
-b --byte <*num*>	Selects byte <*num*> in every interleaved block.
--gap-fill <*val*>	Fills gaps between sections with <*val*>.
--pad-to <*addr*>	Pads the last section up to address <*addr*>.
--set-start <*addr*>	Sets the start address to <*addr*>.
{--change-start|--adjust-start} <*incr*>	Adds <*incr*> to the start address.
{--change-addresses|--adjust-vma} <*incr*>	Adds <*incr*> to LMA, VMA and start addresses.
{--change-section-address|--adjust-section-vma} <*name*>{=|+|-}<*val*>me>	Changes LMA and VMA of the <*name*> section by <*val*>.
--change-section-lma <*name*>{=|+|-}<*val*>	Changes the LMA of the <*name*> section by <*val*>.
--change-section-vma <*name*>{=|+|-}<*val*>	Changes the VMA of the <*name*> section by <*val*>.
{--[no-]change-warnings|--[no-]adjust-warnings}	Warns if a named section does not exist.
--set-section-flags <*name*>=<*flags*>	Sets the <*name*> section's properties to <*flags*>.
--add-section <*name*>=<*file*>	Adds the <*name*> section found in the <*file*> to output.
--rename-section <*old*>=<*new*>[,<*flags*>]	Renames the <*old*> section to <*new*>.
--change-leading-char	Forces output format's leading character style.
--remove-leading-char	Removes leading character from global symbols.
--redefine-sym <*old*>=<*new*>	Redefines the <*old*> symbol name to <*new*>.
--redefine-syms <*file*>	Redefines the symbol name for all symbol pairs listed in the <*file*>.
--srec-len <*number*>	Restricts the length of generated Srecords.
--srec-forceS3	Restricts the type of generated Srecords to S3.

Table 18: lm8-elf-objcopy Options

--strip-symbols <*file*>	-N for all symbols listed in <*file*>.
--strip-unneeded-symbols <*file*>	Strips unneeded symbols for all symbols listed in <*file*>.
--keep-symbols <*file*>	-K for all symbols listed in <*file*>.
--localize-symbols <*file*>	-L for all symbols listed in <*file*>.
--keep-global-symbols <*file*>	-G for all symbols listed in <*file*>.
--weaken-symbols <*file*>	-W for all symbols listed in <*file*>.
--alt-machine-code <*index*>	Uses alternate machine code for output.
--writable-text	Marks the output text as writable.
--readonly-text	Makes the output text write protected.
--pure	Marks the output file as demand paged.
--impure	Marks the output file as impure.
--prefix-symbols <*prefix*>	Adds <*prefix*> to start of every symbol name.
--prefix-sections <*prefix*>	Adds <*prefix*> to start of every section name.
--prefix-alloc-sections <*prefix*>	Adds <*prefix*> to start of every allocatable section name.
-v --verbose	Lists all modified object files.
-V --version	Displays this program's version number.
-h --help	Displays this output.
--info	Lists object formats & architectures supported.
lm8-elf-objcopy: supported targets:	elf32-lm8, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex.

Report bugs to the http://www.sourceware.org/bugzilla/ Web site.

lm8-elf-objdump

The lm8-elf-objdump (lm8-elf-objcopy) utility displays information from object (.o) files.

Usage

```
lm8-elf-objdump <options> <files>
```

where *options* can be one or more of the options shown in Table 19. At least one of the options must be given.

Table 19: lm8-elf-objdump Options

Option	Description
-a, --archive-headers	Displays archive header information.
-f, --file-headers	Displays the contents of the overall file header.
-p, --private-headers	Displays the contents of the object format-specific file header.
-h, --[section-]headers	Displays the contents of the section headers.
-x, --all-headers	Displays the contents of all headers.
-d, --disassemble	Displays the assembler contents of executable sections.
-D, --disassemble-all	Displays the assembler contents of all sections.
-S, --source	Intermixes source code with disassembly.
-s, --full-contents	Displays the full contents of all sections requested.
-g, --debugging	Displays debug information in object file.
-e, --debugging-tags	Displays debug information using ctags style.
-G, --stabs	Displays (in raw form) any STABS info in the file.
-t, --syms	Displays the contents of the symbol tables.
-T, --dynamic-syms	Displays the contents of the dynamic symbol table.
-r, --reloc	Displays the relocation entries in the file.
-R, --dynamic-reloc	Displays the dynamic relocation entries in the file.
-v, --version	Displays this program's version number.
-i, --info	Lists object formats and architectures supported.
-H, --help	Displays these option descriptions.
The following switches are optional:	
-b, --target=BFDNAME	Specifies the target object format as BFDNAME.

Table 19: lm8-elf-objdump Options

-m, --architecture=MACHINE	Specifies the target architecture as MACHINE.
-j, --section=NAME	Only displays information for section NAME.
-M, --disassembler-options=OPT	Passes text OPT on to the disassembler.
-EB --endian=big	Assumes big endian format when disassembling.
-EL --endian=little	Assumes little endian format when disassembling.
--file-start-context	Includes context from start of file (with -S).
-I, --include=DIR	Adds DIR to search list for source files.
-l, --line-numbers	Includes line numbers and filenames in output.
-C, --demangle[=STYLE]	Decodes mangled and processed symbol names. STYLE, if specified, can be auto, gnu, lucid, arm, hp, edg, gnu-v3, java, or gnat.
-w, --wide	Formats output for more than 80 columns.
-z, --disassemble-zeroes	Does not skip blocks of zeroes when disassembling.
--start-address=ADDR	Only processes data whose address is >= ADDR.
--stop-address=ADDR	Only processes data whose address is <= ADDR.
--prefix-addresses	Prints complete address alongside disassembly.
--[no-]show-raw-insn	Displays hexadecimal alongside symbolic disassembly.
--adjust-vma=OFFSET	Adds OFFSET to all displayed section addresses.
--special-syms	Includes special symbols in symbol dumps.
lm8-elf-objdump: supported targets:	elf32-lm8, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex
lm8-elf-objdump: supported architectures:	lm8

lm8-elf-size

The lm8-elf-size program displays the sizes of sections inside binary files. If no input files are specified, a.out is assumed.

Usage

```
lm8-elf-size [options] [files]
```

where *options* can be one or more of the options shown in Table 20.

Table 20: lm8-elf-size Options

Option	Description
-A|-B --format={sysv|berkeley}	Selects output style (default is Berkeley).
-o|-d|-x --radix={8|10|16}	Displays numbers in octal, decimal, or hexadecimal.
-t --totals	Displays the total sizes (Berkeley only).
--target=<bfdname>	Sets the binary file format.
-h --help	Displays this information.
-v --version	Displays the program's version.
lm8-elf-size: supported targets:	elf32-lm8, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex

Report bugs for this tool to the http://www.sourceware.org/bugzilla/ Web site.

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.

application object files Application object files are user source object files that have been compiled and assembled from their source C and Assembly 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/Assembly/C+Assembly 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 LatticeMico 8 microcontroller. 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 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.

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 8 IRQ assignments (0-7), each representing a different physical (or virtual) piece of hardware. 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 LatticeMico8 microcontroller. 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 perspectivre 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 LatticeMico8 microcontroller 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/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 progam.

software application The software application is the code that runs on the LatticeMico8 microcontroller to control the peripherals, the bus, and the memories. The application is written in a high-level language such as C (with or without inlined Assembly) or low-level language such as Assembly.

source files In this document, source files generically refer to source .c and header .h files written in C programming language. Source files can also refer to source .S or .s files written in Assembly.

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 LatticeMico 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, Thes files reside in the *<install_dir>/*components folder. On build generation, this information is copied into the .msb file by MSB.

Index