LatticeMico32 Software Developer User Guide

June 2012

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

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

Contents

LatticeMico System Overview

This software developer's guide describes the flow of tools involved in creating, debugging, and deploying the software application code for the LatticeMico32 embedded microprocessor. In addition, it familiarizes you with the LatticeMico run-time environment, the managed build environment, and its associated directory structure. "Device Driver Framework" on page 121 describes the device driver framework and the advanced issues related to developing custom device drivers. Some treatment is also given to a command-line approach in "Using the LatticeMico System Software" on page 9, with "Software Development Utilities" on page 281 containing tool syntax and options for usage.

This guide is targeted to software programmers who are interested in learning the fundamentals of programming the embedded soft-core microprocessor. For a list of related documents on the LatticeMico32 microprocessor, refer to "Related Documentation" on page 7.

LatticeMico System Design Flow

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

The LatticeMico System software is composed of three bundled applications:

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

These applications work in the background through the user interface and can be accessed through different "perspectives" in the LatticeMico System

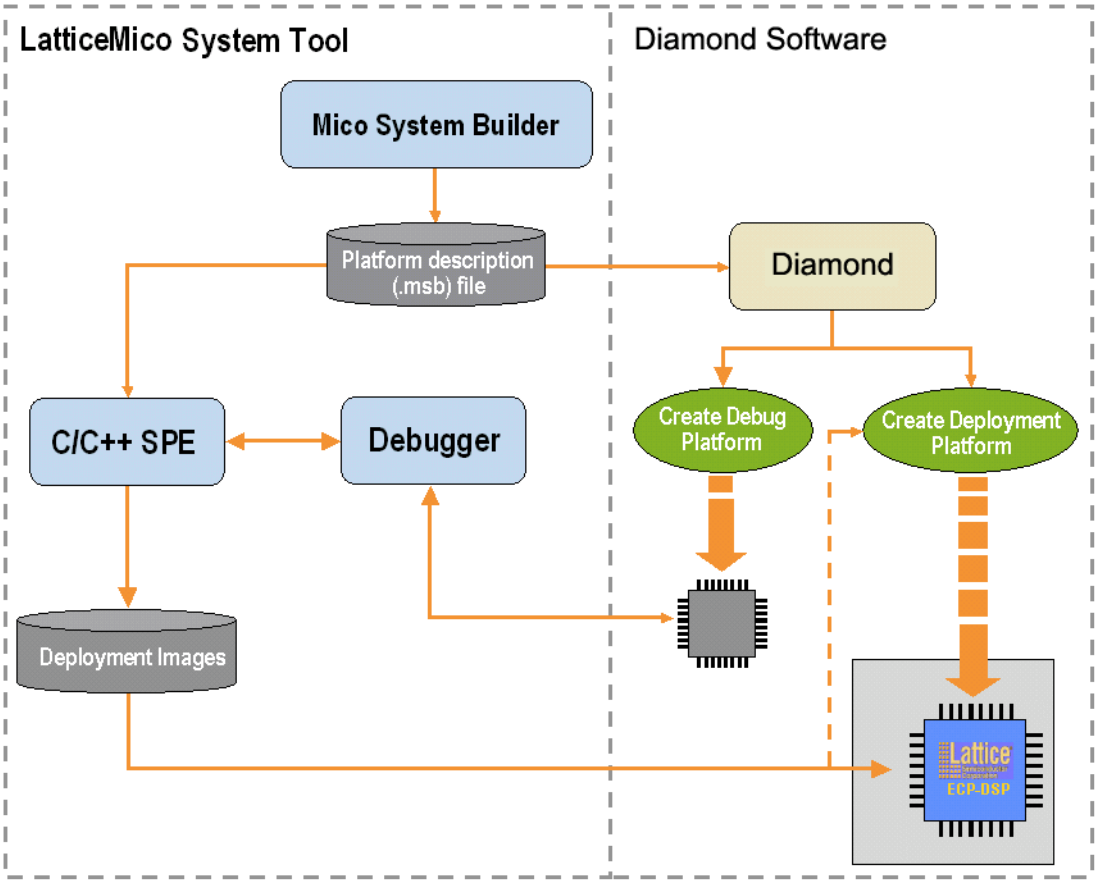
software. Perspectives are a prearranged and predefined set of user functions that can be accessed within the software user interface. You toggle different perspectives on and off by clicking on perspective tabs. Perspectives are described in more detail in “LatticeMico System Perspectives” on page 10.

MSB is used by hardware designers to create the microprocessor platform for both hardware and software development. A platform generically refers to the hardware microprocessor configuration, the CPU, its peripherals, and how these components are interconnected. This functionality in the LatticeMico System software can be accessed by using the MSB perspective in the interface. The default MSB perspective is completely separate in terms of function from the other two perspectives. For complete information about using MSB, refer to the *LatticeMico32 Hardware Developer User Guide*.

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

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

Figure 1: LatticeMico System Development Software Tool Flow



As noted earlier, you can learn more about perspectives in “LatticeMico System Perspectives” on page 10. In addition, the *LatticeMico32 Tutorial* gives step-by-step instructions on creating a sample microprocessor platform, downloading hardware images to your device, creating your application code, and deploying your application code to on-chip or flash memory. It covers all relevant topics to enable you to run through a complete LatticeMico design flow. It is highly recommended that you start out with the tutorial.

Device Support

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

- LatticeECP
- LatticeEC
- LatticeECP2

- LatticeECP2M
- LatticeECP3
- LatticeXP
- LatticeXP2
- LatticeSC
- LatticeSCM

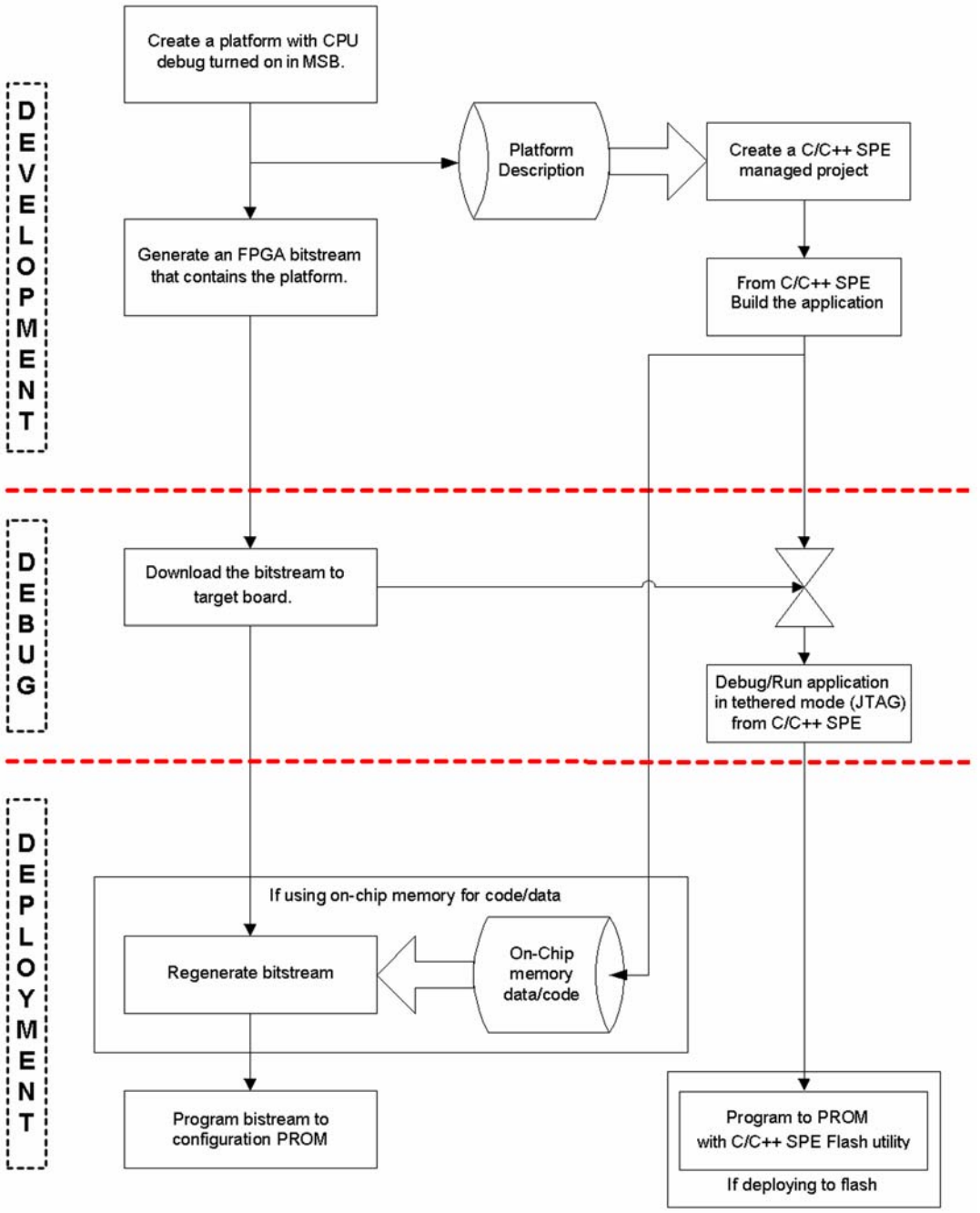
Design Flow Steps

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

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

Figure 2 shows the LatticeMico System design flow.

Figure 2: LatticeMico System Design Flow



About LatticeMico System Software Projects

The LatticeMico System project concept enables you to develop your embedded microprocessor in an integrated environment that automates some of the tasks described in “LatticeMico System Design Flow” on page 1 and makes other tasks more manageable for you.

A LatticeMico System project is a managed project, which means that the software generates a set of makefiles and build management utilities for you to ensure that your software project is built and generated properly, making it unnecessary for you to modify any files to perform this task.

The software system consists of three major functional parts:

- Project/build management
- Application debugging
- Software deployment

Project/Build Management

The project/build management function of the software does the following:

- Automatically selects Lattice Semiconductor-provided, platform-specific drivers (operating system and other selectable software components) based on the .msb file that defines your platform, user selections, or both.
- Automatically creates the appropriate makefiles for building the application, as well as included drivers and software components, without user intervention.
- Provides default project settings, as well as build configuration, to enable you to successfully generate basic platforms and requirements.
- Enables easy manipulation of linker section location for platforms containing a multitude of memory regions through an intuitive user interface.

See “Managed Build Process and Directory Structure” on page 145 for details on the managed build process.

Application Debugging

The application debugging function of the software does the following:

- Provides a default debug session configuration that builds the application in a way that allows you to run and diagnose issues within your application code.
- Provides an intuitive source-level debugging environment that gives you a comprehensive look at the application/CPU during a debug session.

See “Running the Debugger on Your Code” on page 32 and all relevant subsections for information on application debugging.

Software Deployment

The software application can be deployed in three different ways:

- To boot from a flash device
- To boot from an on-chip memory peripheral component
- To boot from multi on-chip memory

See “Software Deployment” on page 178 and relevant procedures in “Using the LatticeMico System Software” on page 9 for deploying your software.

Related Documentation

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

- *LatticeMico32 Processor Reference Manual*, which contains information on the architecture of the LatticeMico32 microprocessor chip, including configuration options, pipeline architecture, register architecture, debug architecture, and details about the instruction set.
- *LatticeMico32 Hardware Developer User Guide*, which provides instructions for using the MSB perspective to develop and configure a hardware platform using the LatticeMico32 microprocessor, peripherals, and IP.
- *LatticeMico32/DSP Development Board User Guide*, which describes the features and functionality of the LatticeMico32/DSP development board. This board is designed as a hardware platform for design and development with the LatticeMico32 microprocessor, as well as for the LatticeMico8 microcontroller, and for various DSP functions.
- *Eclipse C/C++ Development Toolkit User Guide*, which is an online manual from Eclipse that gives instructions for using the C/C++ Development Toolkit (CDT) in the Eclipse Workbench.
- *LatticeMico Asynchronous SRAM Controller*, which describes the features and functionality of the LatticeMico asynchronous SRAM controller
- *LatticeMico DMA Controller*, which describes the features and functionality of the LatticeMico DMA controller
- *LatticeMico On-Chip Memory Controller*, which which describes the features and functionality of the LatticeMico on-chip memory controller
- *LatticeMico Parallel Flash Controller*, which describes the features and functionality of the LatticeMico parallel flash controller
- *LatticeMico GPIO*, which describes the features and functionality of the LatticeMico GPIO
- *LatticeMico Master Passthrough*, which describes the features and functionality of the LatticeMico master passthrough.
- *LatticeMico Slave Passthrough*, which describes the features and functionality of the LatticeMico slave passthrough

- *LatticeMico SDR SDRAM Controlller*, which describes the features and functionality of the LatticeMico SDR SDRAM controller
- *LatticeMico SPI*, which describes the features and functionality of the LatticeMico serial peripheral interface (SPI)
- *LatticeMico SPI Flash*, which describes the features and functionality of the LatticeMico serial peripheral interface (SPI) flash memory controller
- *LatticeMico Timer*, which describes the features and functionality of the LatticeMico timer
- *LatticeMico UART*, which describes the features and functionality of the LatticeMico universal asynchronous receiver-transmitter (UART)
- *Lattice Diamond <release_number> Installation Notice*, which explains how to install the LatticeMico System software for the current release
- *LatticeECP/EC FPGA Family Handbook*, which is a collection of the data sheets and application notes on LatticeEC and LatticeECP devices
- *LatticeECP/EC Family Data Sheet*
- *LatticeECP2 FPGA Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP2 devices
- *LatticeECP2 Family Data Sheet*
- *LatticeECP2M Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP2M devices
- *LatticeECP2M Family Data Sheet*
- *LatticeECP3 Family Handbook*, which is a collection of the data sheets and application notes on LatticeECP3 devices
- *LatticeECP3 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 microprocessor platform
- C/C++ Software Project Environment (C/C++ SPE), which is used to create the software application code that drives the microprocessor platform

- Debugger, which enables you to analyze the software application code to identify and correct errors

The 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 microprocessor or your hardware platform. C/C++ SPE enables you to develop and compile your code in a managed and well-structured build environment. The Debugger includes tools that analyze your code for errors and simulates instruction calls within the software environment or to an actual programmed device on a circuit board.

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

LatticeMico System Requirements

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 Programmer’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

Before you begin learning about the basic tasks that you can perform in the LatticeMico System software, it is important to understand the concept of “perspectives” in the software and how to access the three integrated

functional tools, MSB, C/C++ SPE, and the Debugger, within the user interface. Do not confuse the underlying functional tools in the LatticeMico System software with the various perspectives in the user interface.

There are three default perspectives in the LatticeMico System software:

- MSB perspective

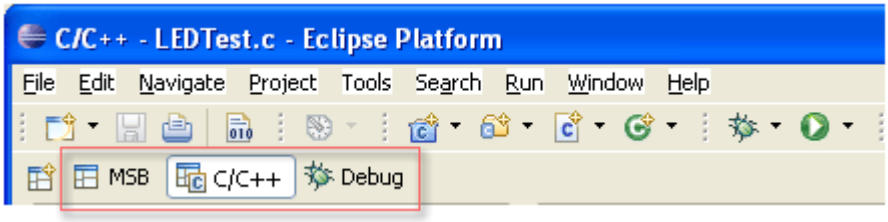
  For complete information about using the MSB perspective to configure the microprocessor hardware platform and peripherals, refer to the *LatticeMico32 Hardware Developer User Guide*.

- C/C++ SPE perspective, shown on Figure 5 on page 16
- Debug perspective, shown in Figure 15 on page 33

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.

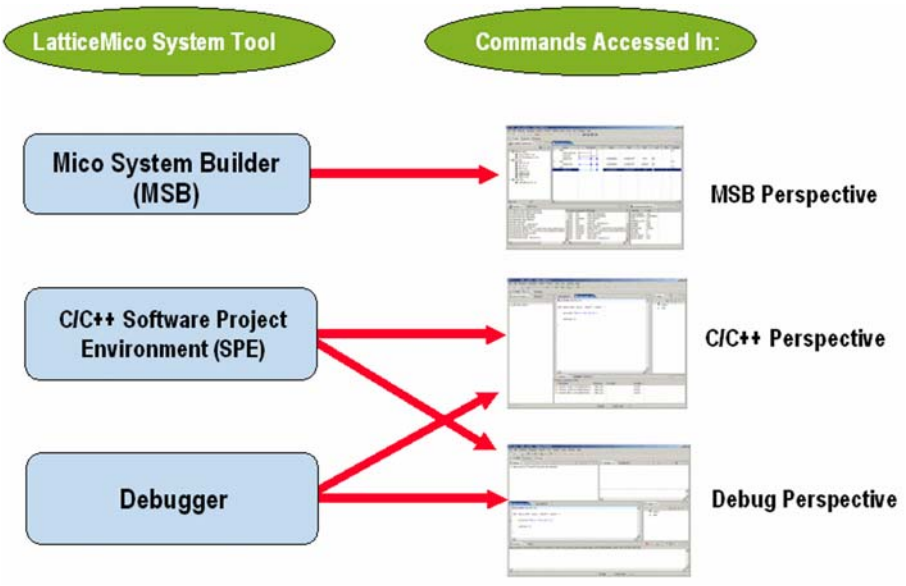
Figure 3: Perspective Activation Buttons



The three different perspectives—the MSB, the C++ SPE, and the Debug—include overlapping tool functions that you access through various commands

and interactive views, as illustrated in Figure 4. You can find more information on these commands and views later in this document and in the online Help.

Figure 4: Tool Functions Accessed in Perspectives



In Figure 4, the C/C++ perspective and the Debug perspective arrows indicate that they share many of the same or similar command functions, so you can perform the same exact operation in either perspective. By default, these two perspectives share many functions because these tasks are very closely related to each other. If you perform some changes in a view such as the Editor view in one perspective, it will affect what you see in another perspective that contains the same view. Do not assume that a given command function in the LatticeMico System is only accessible or viewable from one perspective.

Note

Particular views and options within 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.

Using C/C++ SPE to Develop Your Software

After creating your hardware microprocessor 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 microprocessor 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:

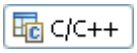
- Create and build new LatticeMico C/C++ software projects.
- Develop and compile your software application code to create executables using its workbench.
- Test or debug your software application code by directly analyzing the development board.
- Access software deployment options such as deploying to on-chip memory or a parallel flash device.
- Launch iProgrammer configuration software for downloading the FPGA bitstream to a hardware target device.

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 10 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 10.

To open the C/C++ perspective:

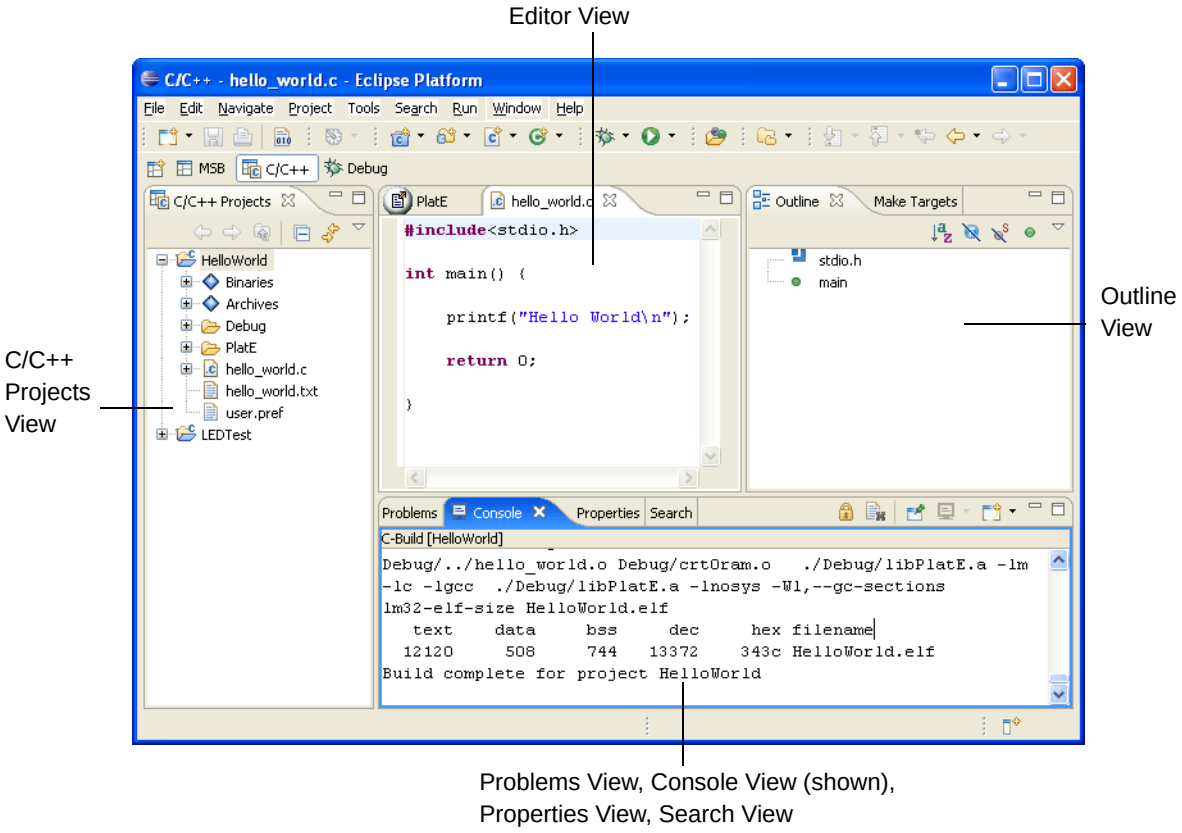
- From the default MSB perspective, click the **C/C++** activation button 

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 5.

Figure 5: C/C++ Perspective



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:

- LatticeMico managed make C/C++ project
- LatticeMico library project
- LatticeMico standard make C/C++ project

A LatticeMico managed make C/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 LatticeMico32 microprocessor platform and creates the appropriate files required for a build.

The LatticeMico library project and the LatticeMico standard-make project are described in “Advanced Programming Topics” on page 175. 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 microprocessor 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:

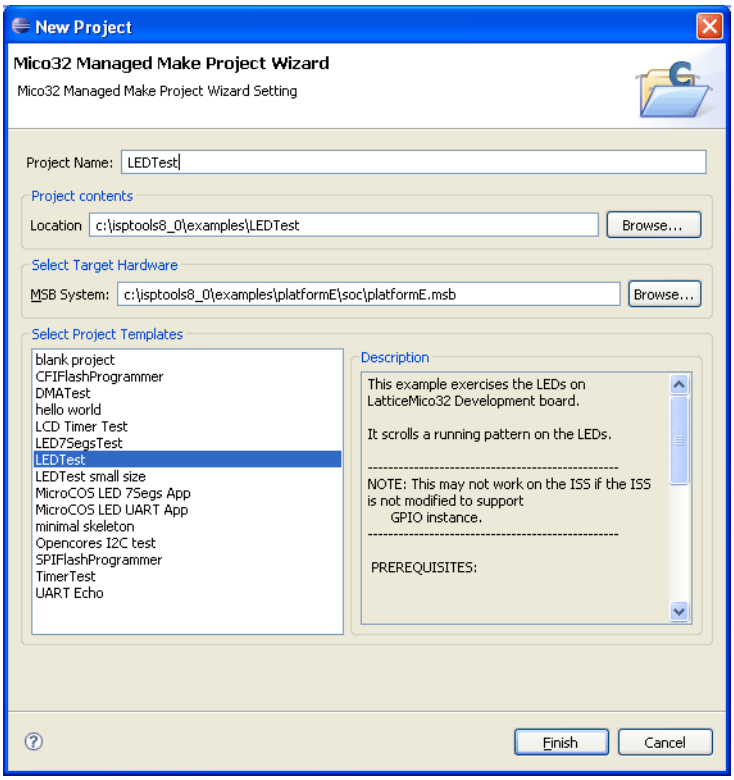
1. From the MSB perspective, click the **C/C++** button in the upper left.

   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 6.

Figure 6: New Project Dialog Box



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.

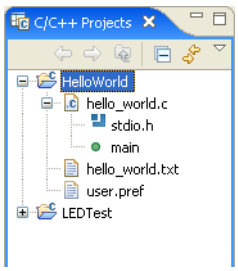
   Your new project will appear in the C/C++ Projects view.

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 7. The project folder in the view is shown expanded for illustrative purposes.

Figure 7: Hello World Projects View



As you can see in Figure 7, 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 145 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

LatticeMico C/C++ SPE is derived from Eclipse CDT, so basic project operations that apply to the Eclipse CDT perspective also apply to LatticeMico 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

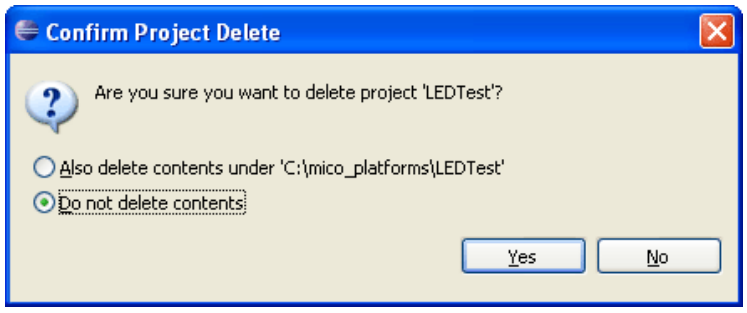
If you have created projects in your LatticeMico workspace that you want to remove, you can delete them from the Projects view.

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 8 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 8: Project Deletion Confirmation Dialog Box



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 8, 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

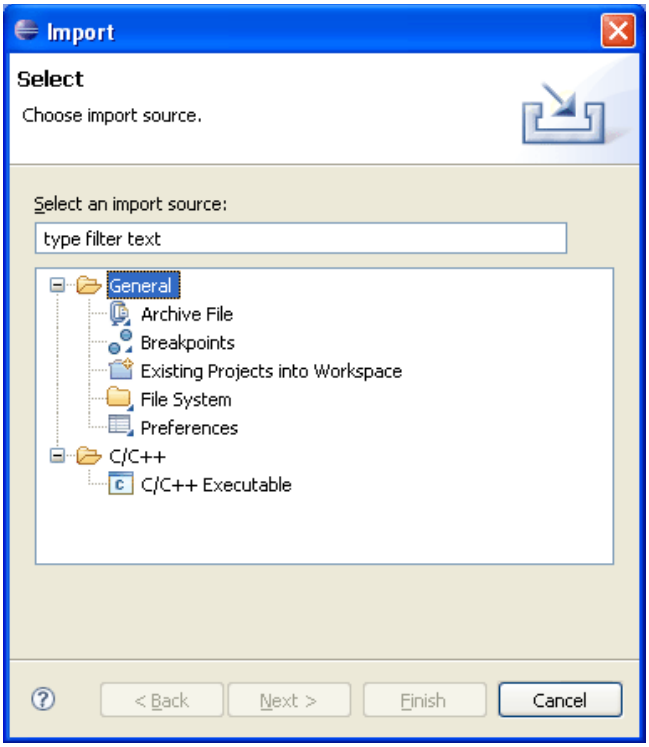
You can use the Import Wizard to copy a project from a different workspace or copy a project that previously existed in your workspace and import it into the LatticeMico software workbench. You cannot import a project that has the same project name as an existing project into the Projects view.

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 9.

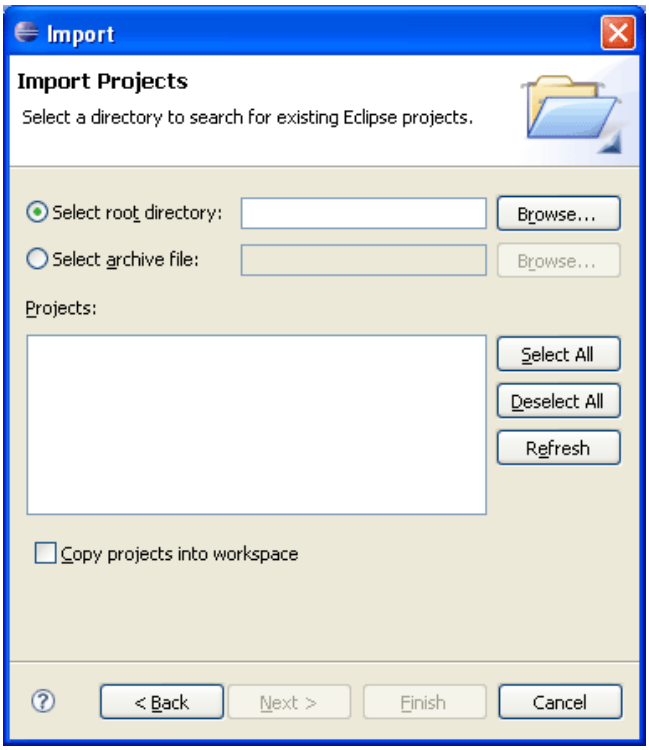
Figure 9: Import Dialog Box in Select Mode



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 10.

Figure 10: 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.

Typically you might expect a default build configuration for building software application code that can be debugged. You might also expect a default build configuration for building optimized software application code that is

unsuitable for debugging with a debugger tool because it may contain optimized code and may not include debug symbols.

A newly created C/C++ SPE project provides two default configurations:

- A debug build configuration for generating an executable that can be debugged
- A release build configuration for generating an optimized executable devoid of any debug information

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 linker settings depend on the MSB platform for locating the various compiler-dependent sections, for example, the text section that contains the executable code. These section settings become especially important for platforms that contain multiple memory components. The LatticeMico System managed project enables you to make selections for the location of the executable section, read-only data section, and read/write data section.

The section settings are updated when a change to the MSB platform file is detected and when you attempt to access the section setting. If the .msb file changes and the application is built without accessing the sections settings, the build will fail if the user-selected section memory does not exist in the updated .msb file.

The system library settings, as well as the linker section settings, are the same for all build configurations.

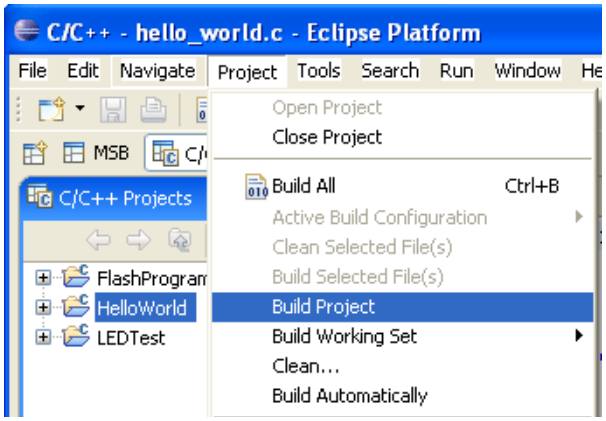
Building Your Software Project

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 11, the highlighted project folder in the Projects view is called HelloWorld.

Figure 11: 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 11.

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 LatticeMico System online Help.

Setting Project Properties

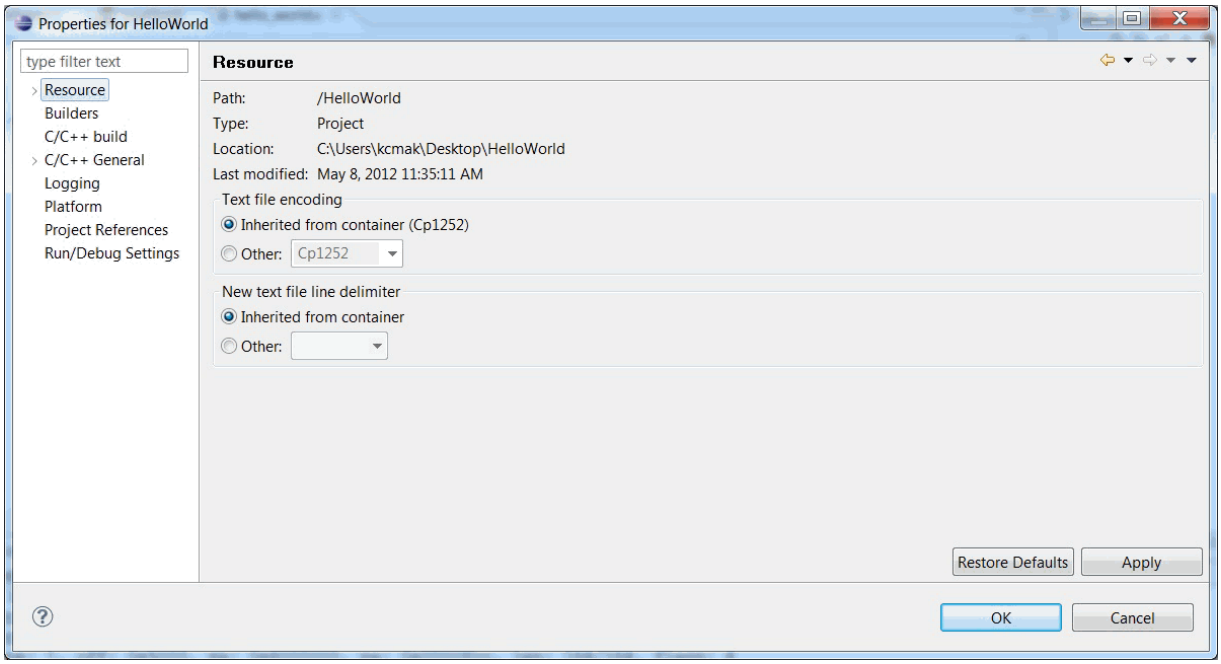
You can set up your project properties in the Properties dialog box. Project properties include various project parameters, for example, file encoding parameters, build configuration options, and platform settings. The Project Property dialog box is dynamic in that it enables you to select different “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 11 on page 26, the project is entitled HelloWorld.
2. In the pop-up menu, choose **Properties**.

   The Properties dialog box appears, as shown in Figure 12.

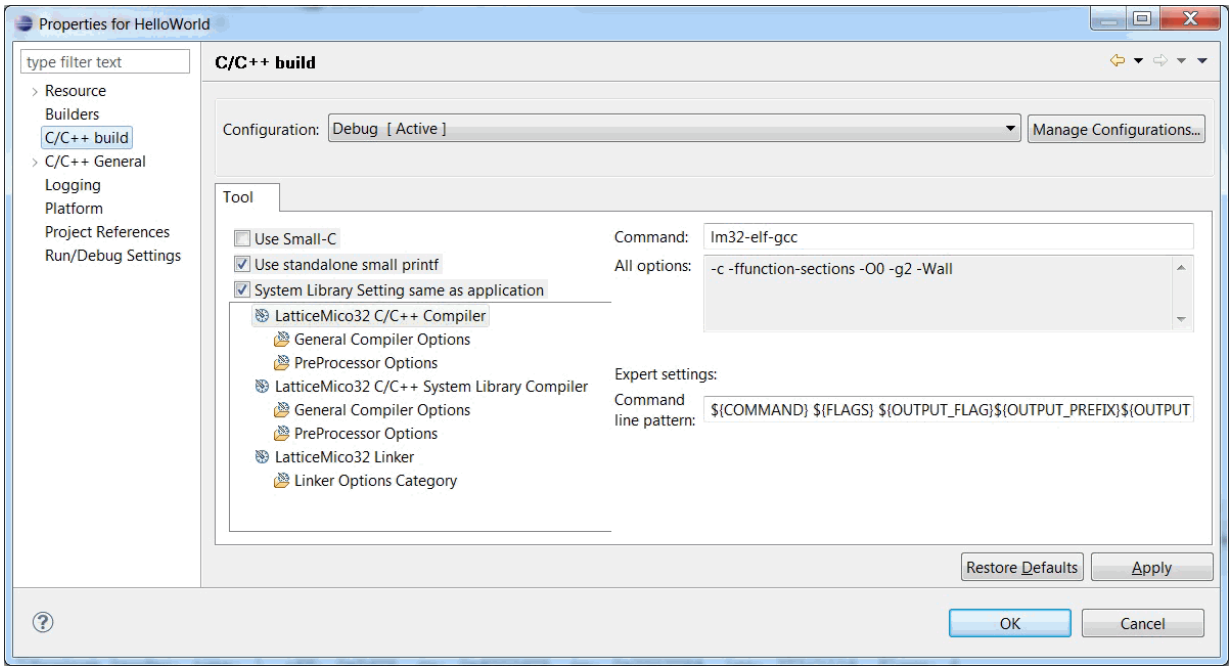
Figure 12: Properties Dialog Box



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 12.

The C/C++ build tab, as shown in Figure 13, enables you to set build properties for the project.

Figure 13: C/C++ Build Tab of Properties Dialog Box



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

- Configuration Setting – 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. LatticeMico C/C++ SPE uses the GNU C/C++ tool chain for project compilation and linking. A set of C/C++ build settings is known as a build configuration.

  The C/C++ SPE has two predefined configurations, Debug and Release. The default Debug build configuration is set to maximize debug visibility with compiler optimizations turned off. The release configuration is set to maximize program efficiency with no debug visibility. To define your own configurations, refer to the Eclipse CDT documentation.

- Tool Setting– This tab enables you to view or modify the compiler or linker settings, the run-time library, and printf selections.

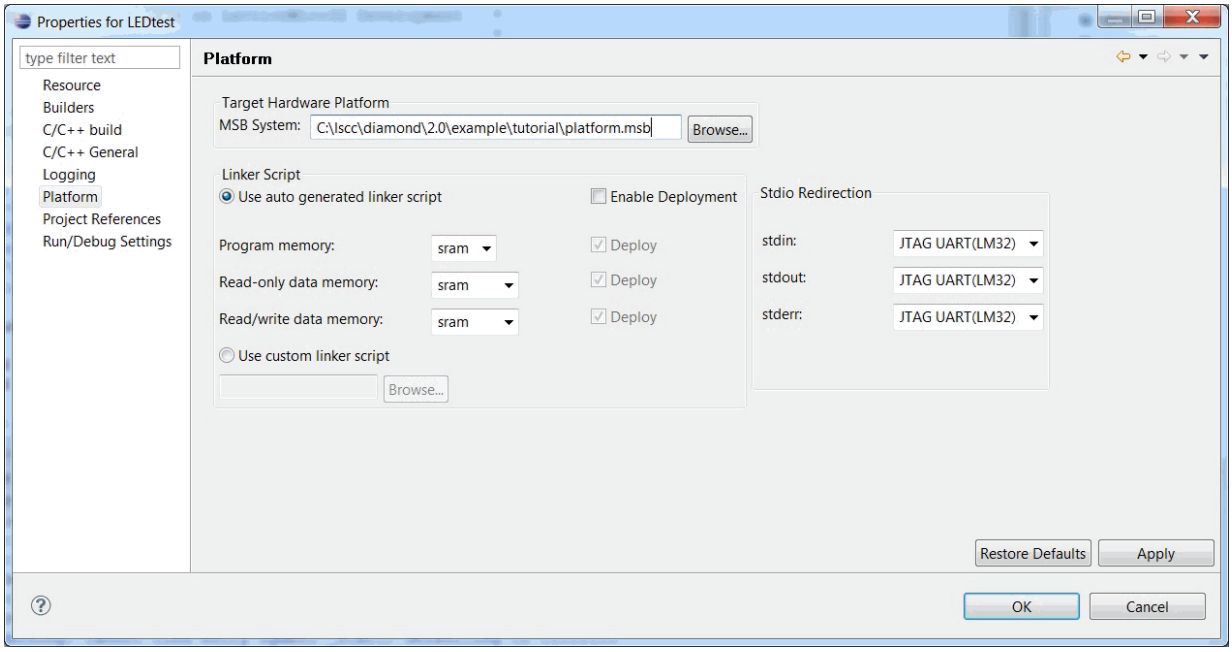
  - "Use Small-C" and "Use standalone small-printf" affect the run-time C library selection and the printf selection, respectively. These two options are described in “Run-Time Libraries” on page 45.

  - "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 145. The LatticeMico linker settings affect the generation of the application executable.

Figure 14 shows the platform settings that are accessible in the Platform tab of the Properties dialog box.

Figure 14: Platform Tab of the Properties Dialog Box



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 on the basis of the selected platform.

  You can provide your own linker script by selecting the Use Custom Linker Script button. If you use the default linker script generated by C/C++ SPE, you can choose the memories in which different linker sections will be placed. C/C++ SPE explores the selected platform and identifies memory components and their attributes, making them available for user selection in this field.

  - Program Memory – This memory, which can be a read/write memory or a read-only memory, contains the application instructions for LatticeMico.

- Read Only Data Memory – This memory, which can be a read/write memory or a read-only memory, contains the read-only application data.
- Read/Write Data Memory – This memory contains read/write data required by the program, as well as the microprocessor stack and application heap. C/C++ SPE makes available only those memories marked as read/write. Memories marked as read-only are not made available by C/C++ SPE.

Whether you choose to use the default linker script or provide your own, C/C++ SPE always generates a default linker script.

- Stdio Redirection – This group of options enables you to select an appropriate file device to handle standard input/output requests from the software application.

  The C/C++ SPE inspects the .xml file of each component included in the platform to identify which components are capable of handling the standard input/output streams. Refer to the section “Creating Custom Components” in the *LatticeMico32 Hardware Developer User Guide* for instructions on how to make a component instance available for handling standard input/output operations through this Platform tab.

  In addition to marking the component description file to make the device appear in this Platform tab selection, the component-specific software must be configured as a file device to handle file-operation requests from the LatticeMico File Service. Refer to “LatticeMico File Service” on page 98 for details on the LatticeMico File Service and how to create a file device.

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.

Using LatticeMico System as a Stand-Alone Tool

The software developer can use C/C++ SPE to develop software application code without having to install Lattice Diamond, as long as the directory structure and appropriate files have been provided by the hardware developer. The files that the hardware designer provides to the software developers are the Mico System Builder project file, the LatticeMico32 microprocessor driver files and GNU files, the component driver files, and the FPGA's configuration bitstream.

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

The following scenario shows the tasks involved:

Hardware Developer The hardware developer performs the following tasks:

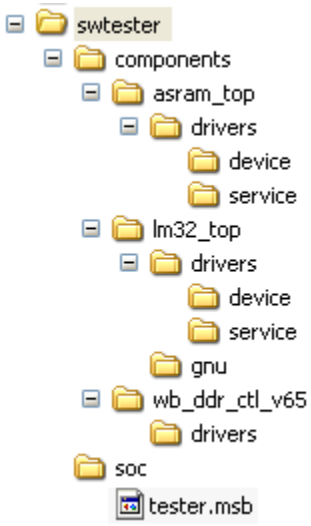
1. Uses Lattice Diamond to create an FPGA development project.

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

2. Generates the platform for the project using LatticeMico System Builder.
3. Imports the platform’s RTL source files into the project in Diamond and generates the FPGA's configuration bitstream.

4. Sends all software developers the Mico System Builder project directory.

   For example:



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

Software Developer The software developer performs the following tasks:

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

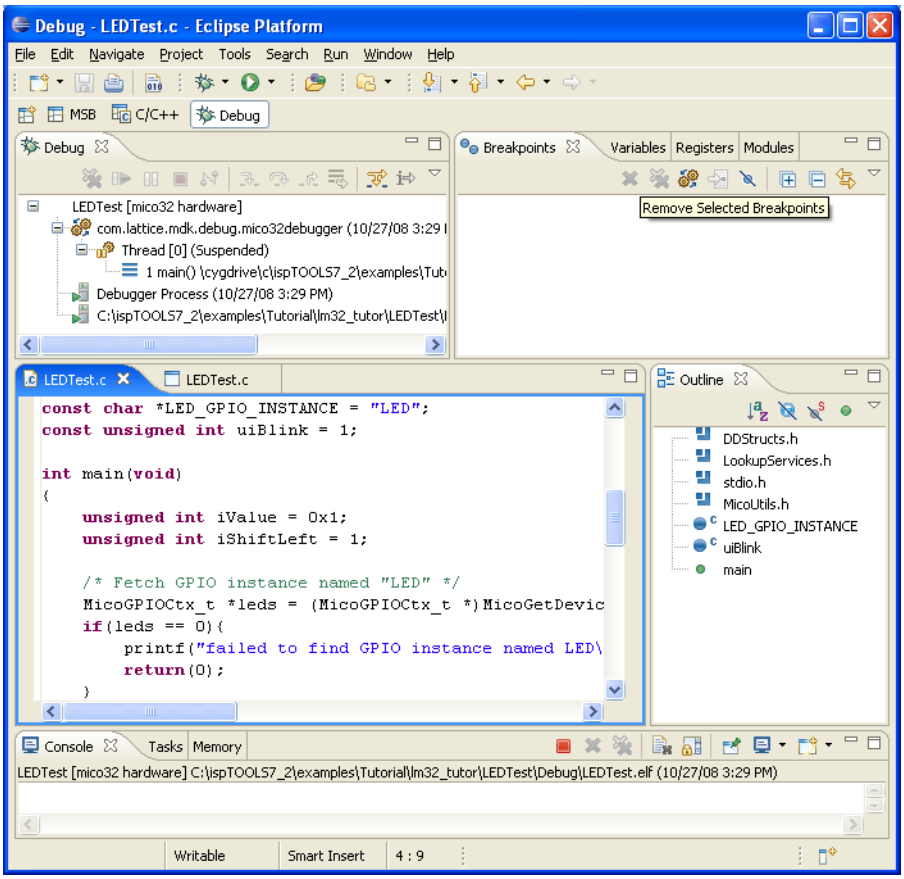
Running the Debugger on Your Code

You can run the Debugger as described in the following procedure, and as shown in Figure 15 on page 33.

To use the Debug perspective to run the Debugger on your code:

- In the upper left-hand corner of the MSB graphical user interface, select the **Debug** activation button to switch to the Debug perspective. Alternatively, you can choose **Window > Open Perspective > Debug**.

Figure 15: Debug Perspective



To run the Debugger tool from within the C/C++ perspective:

1. In the Projects view of the C/C++ perspective, click on the project folder name to select and choose **Run > Debug Configuration**.
2. To create a new configuration, click the **New launch configuration** icon on the toolbar above the left pane.

   The Debug dialog box now appears with **hello_world_0** highlighted.
3. In the Debug dialog box, click **Debug**.
4. In the Confirm Perspective Switch box, click **Yes**.

The Debug perspective consists of the following views:

- Debug view, which displays the function calls made so far
- Variables view, which displays the variables that are used in the source code functions
- Breakpoints view, which appears when you insert a breakpoint
- Source view, which displays the source code when you click on a thread in the Debug view
- Outline view, which displays the functions in the source code

- Console view, which displays the output of the debugging session
- Tasks view, which is not used
- Modules view, which displays the modules of the executable loaded. If you click on a module, C/C++SPE displays all the functions that compose that module.
- Registers view, which displays the registers in the CPU. It also shows the values on the registers at the breakpoints. Values that have changed are highlighted in the Registers view when your program stops.
- Signals view, which enables you to view the signals defined on the selected debug target and how the debugger handled each one
- Memory view, which enables you to inspect and change multiple sections of your process memory
- Expressions view, which is activated if you right-click in the Source view, choose Add Watch Expression, and enter a variable in the Add Expression dialog box
- Disassembly view, which shows the source code in assembly language with offsets. It shows the instructions that reside at each address.

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 on the available views, refer to the online Help.

Debugging and Executing Your Code

C/C++ SPE provides a GUI-based debugging environment. It uses the GNU GDB debugger for controlling program execution and debug activities. It also uses a Lattice Semiconductor-provided application to facilitate a communication channel between the LatticeMico32 microprocessor debug module and GDB. You can choose either to run the application or to debug the application.

For information about performing functional simulation of a LatticeMico32 platform, refer to the *LatticeMico32 Hardware Developer User Guide.*

Note

It is important that the platform generated using MSB include a CPU with its debug option enabled so that C/C++ SPE can download and debug an application.

C/C++ SPE provides two means of exercising the final executable:

- Debugging the software application code. To debug your application, the LatticeMico32 microprocessor instance in your platform must have the debug capability turned on. Additionally, you must enable generation of debug symbols (–g2 or –g3 compiler option) when compiling your application, as is done for the default debug build configuration. For a thorough debug session, you should disable compiler optimizations (-O0) so that the compiler does not rearrange the code.

When you debug an application, the C/C++ SPE perspective changes to the Debug perspective, which provides views for inspecting CPU registers, watching disassembled code, watching the stack trace, and so forth. Debugging typically allows you to “pause” the application being debugged and to place a breakpoint at a predefined symbol to stop execution at a predetermined location when it downloads and runs the code.

- Running the software application code. When you run the software application code, C/C++ SPE simply downloads the executable and executes it without changing the view. It does not insert any breakpoints and executes the code without allowing you to debug the application.

C/C++ SPE requires information about the target connection (for example, JTAG), the executable to be used, debug information (for example, the choice of breaking at a predetermined symbol), and the source location for a debug session. This collection of information forms a target configuration.

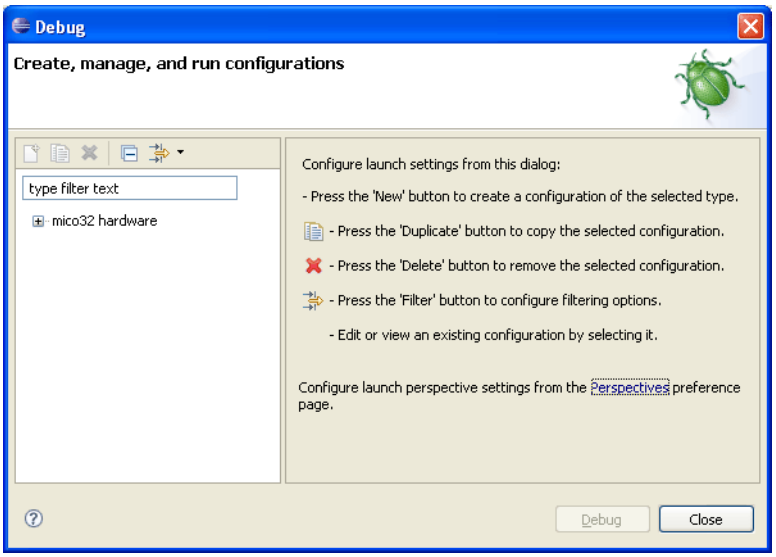
C/C++ SPE provides one target configuration for a JTAG connection to the target that you can use without having to enter or modify the default settings. If you are an advanced user, you can modify the default settings. The created configurations are saved so that you do not have to recreate the configuration for a given project every time.

To run a debug session:

1. In the C/C++ perspective’s Projects view, click the Project folder name to select it and choose **Run > Debug**.

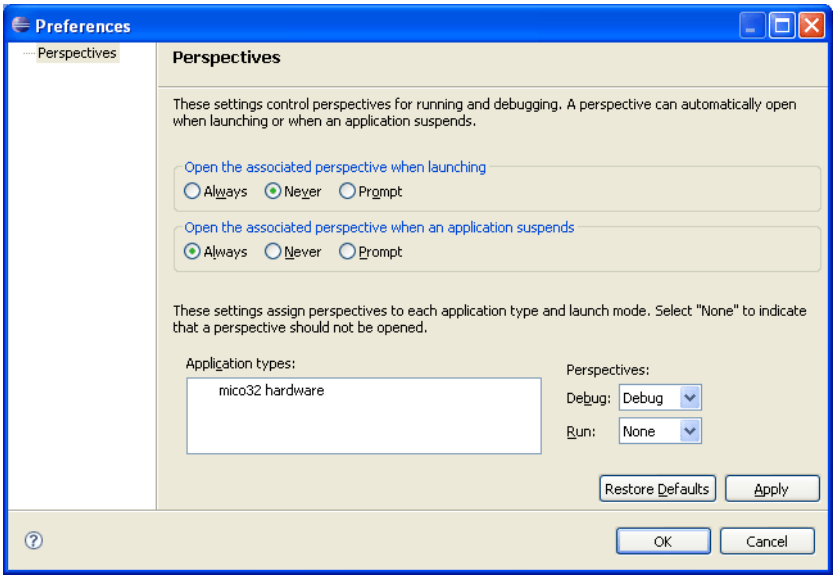
   The Debug dialog box now appears, as shown in Figure 16. It enables you to choose your target connection from the configurations list on the left

Figure 16: Debug Launch Configuration Dialog Box



2. In the sentence on the bottom right, click the **Perspectives** link to open the Perspective dialog box, shown in Figure 21

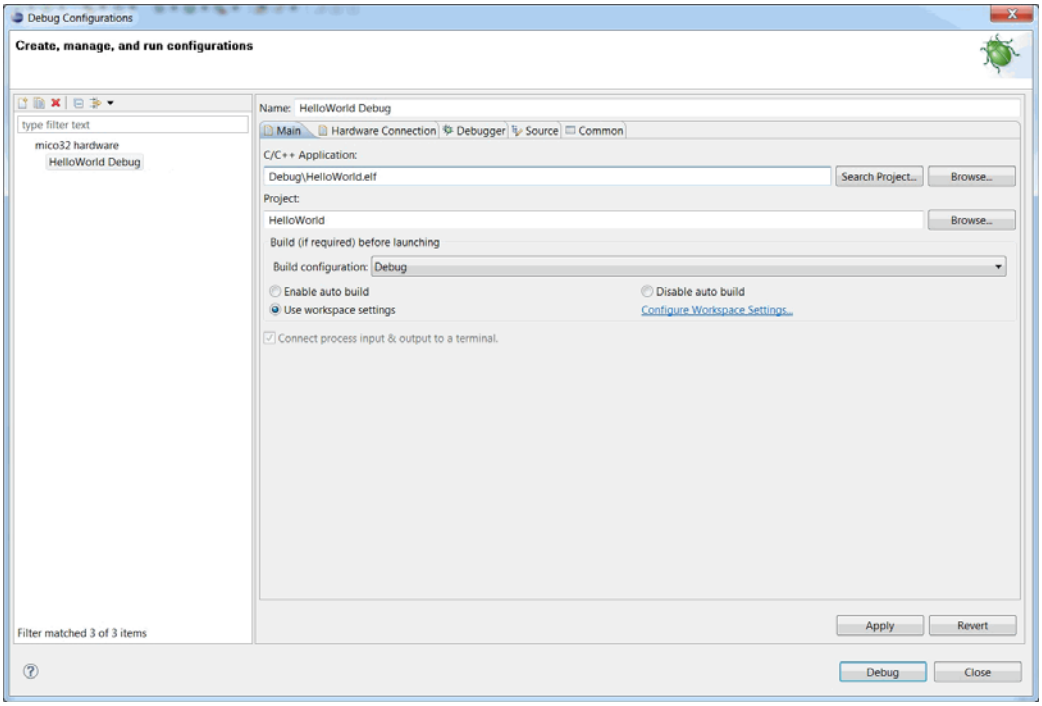
Figure 17: Perspectives



3. The Perspectives dialog box enables you to specify the associated perspective for this instance of the launch configuration. You should not change the associated perspective selection (set to Debug in Figure 17) for debug sessions, because the Debug perspective provides a complete debug environment.
3. Click **OK** to return to the Debug Configuration dialog box.
4. In the target configuration list on the left, select the desired target configuration—mico32 hardware.
5. Click the “New launch configuration” button on the toolbar to create a launch configuration based on your selections.

The dialog box changes, displaying the configuration tabs, as shown in Figure 18.

Figure 18: Main Tab of Debug Dialog Box



This dialog box includes the following tabs:

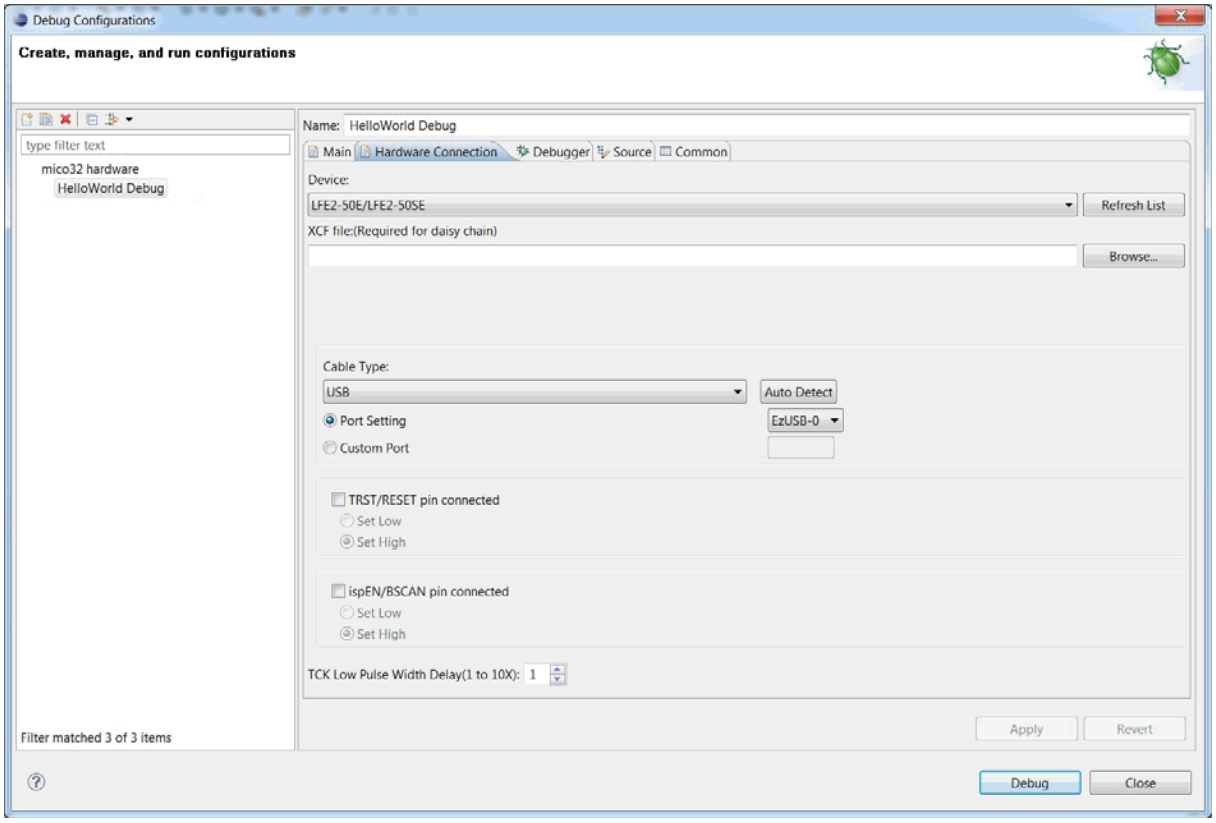
- Main tab – You select the project and its associated executable in this tab. Only those projects that are part of the C/C++ Projects view are available for selection. However, you can choose an executable (.elf) file that may or may not be associated with the selected p roject.

  If you launched the Debug configuration after selecting the project you wish to debug, the options in this tab will contain default settings based on the selected project. You must make sure that the appropriate C/C++ application is selected if you have multiple applications for the same project.

- Hardware Connection tab – This tab is available only for the mico32 hardware configuration. It enables you to specify a scan chain

configuration (.xcf) file generated by Diamond Progammer when you want to program devices in a JTAG daisy chain.

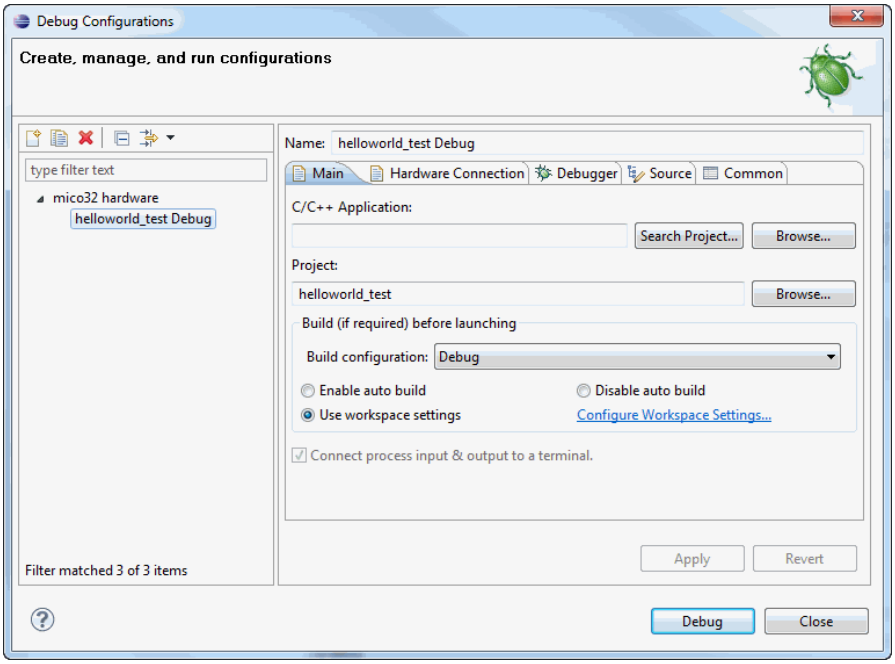
Figure 19: Hardware Connection Tab



- Debugger tab – The default debugger settings are configured appropriately and generally should not be modified. You can specify whether you want your initial breakpoint at the application main routine

(main()) or at the device driver initialization routine (LatticeDDInit) that is called before the application main routine.

Figure 20: Debugger Tab of Debug Dialog Box

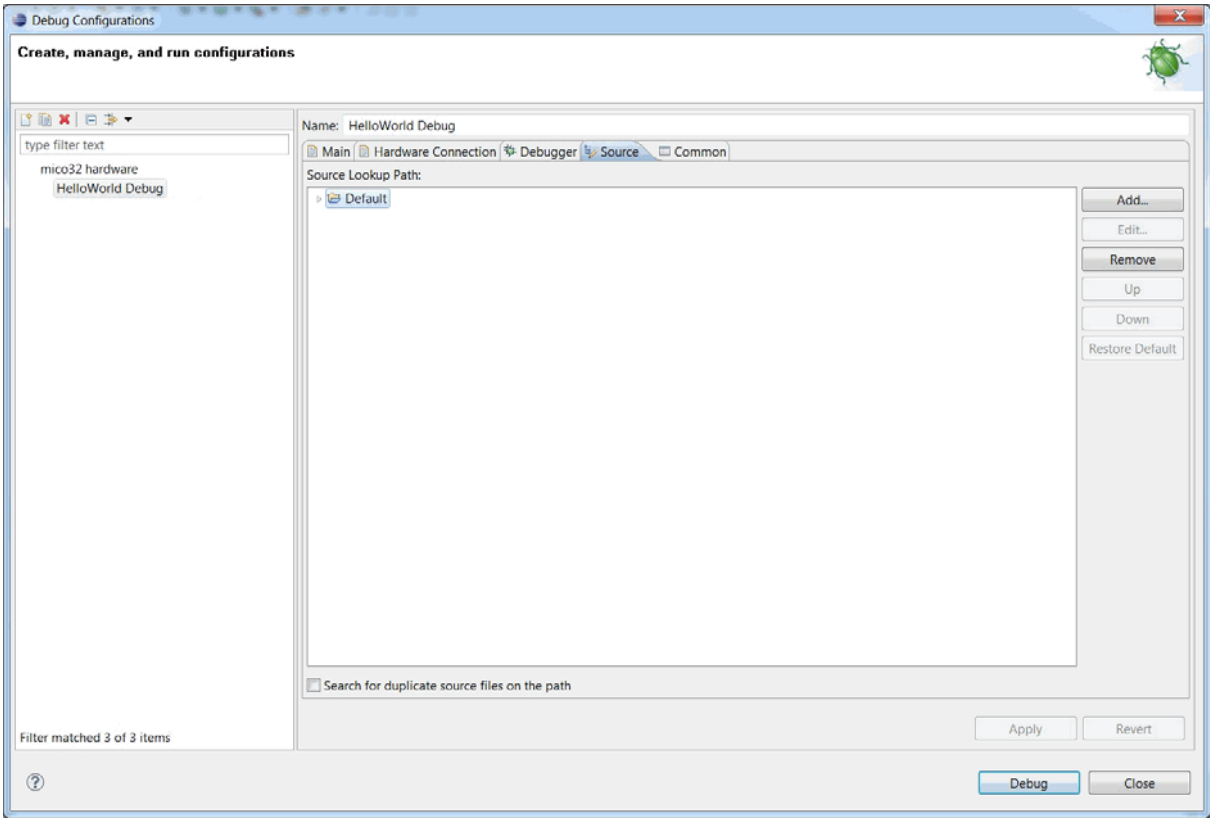


Additionally, you can always open the appropriate source file in the C/C++ perspective and put in a breakpoint before launching a debug session. The Remote Target option specifies the address that GDB will use when connecting to the Lattice Semiconductor-provided target communication executable. You should not modify the default setting for this field.

- Source tab – By default, this tab contains all the necessary information for a debug session. It enables you to specify any additional folder or source that the Debugger should look up to find source information. By design, all source files pertaining to the C/C++ SPE device driver framework are contained within the project, as discussed in subsequent chapters of this document.

If you have source files that do not reside within the C/C++ project, you can provide lookup paths for the Debugger through this tab.

Figure 21: Source Tab in Debug Launch Configurations



After you have configured the appropriate tabs, click the **Apply** button to save the debug launch configuration that you have defined for future use.

For additional information, refer to the Eclipse/CDT documentation.

6. Select the appropriate launch configuration in the **Configurations** list box on the left.

   If you have multiple launch configurations, you must select an appropriate launch configuration before starting a debug session.

7. In the Debug dialog box, click **Debug**.
8. In the Confirm Perspective Switch box, click **Yes**.

This procedure activates the Debug perspective and downloads the .elf file into the FPGA. In addition, the Debug perspective allows a more robust

environment for all aspects of debugging the application code and testing it before it is ready for download.

Note

A debug session is useful only if the executable being debugged contains debug information at a minimum. Debugging an executable built with non-debug compiler settings in the build configuration will not fail, but it will not be very valuable.

After starting the debug session for hardware targets, C/C++ SPE launches a Lattice Semiconductor-provided communications executable for communicating with the microprocessor and the GDB, which is customized in LatticeMico for controlling the debug process.

Common Debugging Tasks

This section summarizes common debug tasks. Additional information can be found in the *Workbench User's Guide* Help system that is available in the software interface.

- Step over – Click the  icon in the Debug view to step over a source line.
- Step in – Click the  icon in the Debug view to step into a function.
- Step out – Click the  icon in the Debug view to step out.
- Instruction stepping – Open the Disassembly view. Click the  icon in the Debug view to activate instruction-level stepping, then use the stepping functions. To return to source-level stepping, deactivate instruction stepping by clicking the same icon.
- Inserting breakpoints – In the Source view or the Disassembly view, click on the line where you wish to insert/remove the breakpoint, then select **Run > Toggle Line Breakpoint**.
- Terminate – Click the  icon in the Debug view to terminate the execution of your application code.
- Pause – To pause a running application, click in the Disassembly or Source view or select the thread of execution in the Debug view to activate the Pause  icon and click on it. This icon will be unavailable if the execution cannot be paused (that is, the application is already paused at a breakpoint) or if the appropriate thread is not selected.
- Resume – To resume a paused execution, click the  icon in the Debug view.

IRunning the Software from the Command Line

In addition to using the LatticeMico System's graphical user interface, you can also run through the flow using the SDK shell that is available in the LatticeMico System software. This section describes the command-line interface, how to access it, and the basic flow involved in using the command line as a primary interface.

Opening the SDK Shell

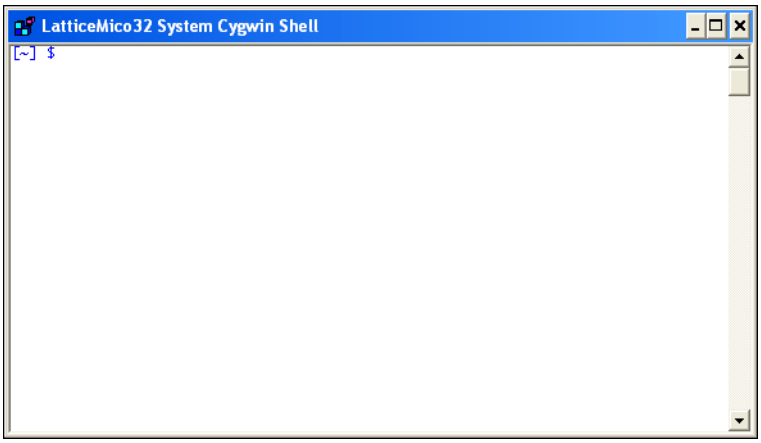
You run command line directives from the LatticeMico System SDK shell. This shell is a Cygwin Bash shell and is already configured with the proper environment variables and path definitions.

To open the LatticeMico System SDK shell:

- From the Windows desktop Start Menu, choose **Start > Programs > Lattice Semiconductor > Accessories > LatticeMico System SDK Shell**.

  A command-line interface window appears, as shown in Figure 22.

Figure 22: LatticeMico System SDK Shell



Use this LatticeMico System SDK shell to invoke the managed build and debug GNU tools that are described in more detail in "Software Development Utilities" on page 281. This appendix provides complete command-line syntax and tool descriptions.

Command-Line Managed Project Builds

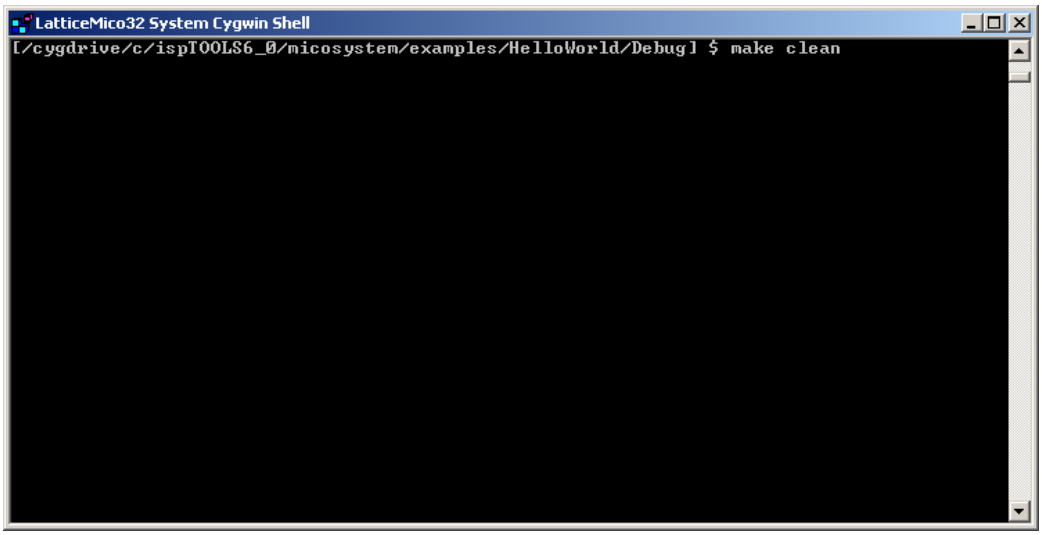
The LatticeMico managed build application invoked from the C/C++ SPE interface generates the application makefile. This application makefile in turn relies on other makefiles and Perl scripts for generating and copying appropriate platform-specific content, such as the source files, header files, and the default linker script.

The application makefile is created in the project's build configuration folder, as explained in detail in "Managed Build Process and Directory Structure" on page 145. The entire build process for this configuration is driven through this application makefile.

Cleaning Your Project

Once the managed build has created the application makefile, you can build your application from the LatticeMico System Cygwin shell. To clean the project, use the "make clean" command, as shown in Figure 23. In the figure, the LatticeMico managed build project is named "HelloWorld," and the build configuration is named "Debug."

Figure 23: Cleaning the Project

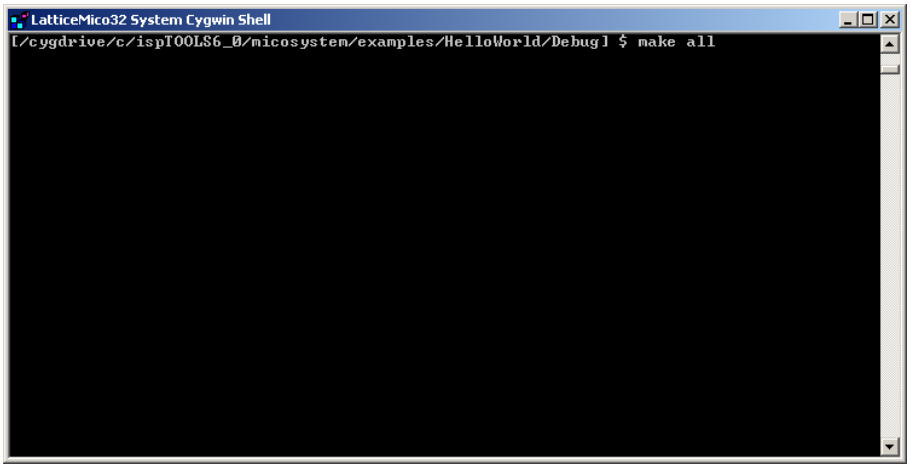


The "make clean" command cleans the intermediate project, as well as the associated platform library's files. It also removes the system library and the application executable, if they exist.

Building Your Project

The next step after ensuring that you have a clean project environment is to build your project. To build the project, use the “make all” command, as shown in Figure 24.

Figure 24: Building the Project



When you run the “make all” command, the software goes through the process of building the platform library, which includes copying appropriate device driver sources and generating the default linker script, as well as the executable.

Command-Line Unmanaged Project Builds

You can also use a makefile generated by a C/C++ SPE managed build process as a starting point to create an unmanaged make project. However, you must still adhere to the rules for peripheral.mk component makefiles. The build process relies on extracting peripheral information from the .msb file.

The entire build process, the library as well as the application, relies on the application makefile and the user.pref file, as discussed in “Managed Build Process and Directory Structure” on page 145. As part of the C/C++ SPE build process, these two files are automatically generated and can be used as starting points for creating an unmanaged make project.

LatticeMico Run-Time Environment

This chapter describes LatticeMico System's run-time environment, takes you through an example of a simple program, and introduces you to various utility functions in the supported Newlib C library.

Build/Compilation Utilities

C/C++ SPE is built on the GNU GCC compiler tool chain customized for LatticeMico32 microprocessor. It contains the standard GNU GCC executable utilities, such as objdump, gcc, g++, and ld. The names of these utilities all contain the "lm32-elf" prefix. For example, the GNU GCC compiler executable customized for the LatticeMico32 microprocessor is called lm32-elf-gcc, and the objdump utility customized for the LatticeMico32 microprocessor is called lm32-elf-objdump.

Used by the Eclipse-based C/C++ SPE managed build environment, these utilities can be accessed from the LatticeMico System SDK shell, as described in "IRunning the Software from the Command Line" on page 42. Refer to "Software Development Utilities" on page 281 for more information on compilation and build utilities and valid options for them.

Run-Time Libraries

This section describes the Newlib C library (libc.a) and the Small Newlib C library (libsmallc.a).

Newlib C and Math Libraries

The LatticeMico C/C++ SPE managed build uses the Newlib C library, as well as the Newlib math library as part of application executable generation.

Both libraries are precompiled for the different possible CPU configurations that affect the generated code, such as the barrel shifter, which affects shift instructions, and the hardware multiplier, which affects multiplication operations. You can find these precompiled libraries in the <*install_dir*>\micosystem\gtools\lm32-elf\<*lib_file_path*>.

This library folder contains various subfolders, each depicting a CPU configuration. Each of these subfolders contains a libc.a archive file, which is the precompiled Newlib C library archive file, and a libm.a archive file, which is the precompiled Newlib math library archive file. As part of the application generation by the C/C++ SPE, the archive file appropriate for the CPU configuration is picked by the lm32-elf-gcc utility.

The Newlib C library depends on certain function calls for completing the functionality of the invoked ANSI standard C function. Although the LatticeMico development framework does not currently provide an “out-of-the-box” operating system, it does include implementations for such expected function calls. Refer to Newlib C documentation for a description of the expected functionality of these subroutine calls.

Table 1 lists the functions expected by the Newlib C library.

Table 1: Functions Expected by the Newlib C Library

Function	Notes
_exit	Functionality implemented as part of the platform library
close	Functionality implemented as part of the platform library file service
environ	Dummy functionality provided
execve	Dummy functionality provided
fork	Dummy functionality provided
fstat	Functionality implemented as part of the platform library file service
getpid	Dummy functionality provided
isatty	Functionality implemented as part of the platform library file service
kill	Dummy functionality provided
link	Dummy functionality provided
lseek	Functionality implemented as part of the platform library file service
open	Functionality implemented as part of the platform library file service

Table 1: Functions Expected by the Newlib C Library (Continued)

read	Functionality implemented as part of the platform library file service
sbrk	Functionality implemented as part of the platform library file service
stat	Functionality implemented as part of the platform library file service
times	Dummy functionality provided
unlink	Dummy functionality provided
wait	Dummy functionality provided
write	Functionality implemented as part of the platform library file service

Implementation details about the “dummy” functions listed in Table 1 are contained in the libnosys.a archive file, which can be found in the *<install_dir>*\micosystem\gtools\lm32-elf\lib file path. Similar to libc.a and libm.a, this library is precompiled for the different possible CPU configurations that affect the generated code.

The managed build process picks the appropriate libnosys.a archive file based on the platform’s CPU configuration. The dummy functions are essentially “stub” functions that act as placeholders for the compilation process to succeed by providing definitions of the functions expected by the Newlib C library. These dummy functions do not contain any functionality.

The managed build process does not link in the dummy functions unless your application calls them directly or calls them indirectly through the Newlib C library. The compilation process generates a warning message in the C/C++ perspective’s Problems view if any of these dummy functions are linked into the final application through a direct call or indirectly through Newlib C library function calls.

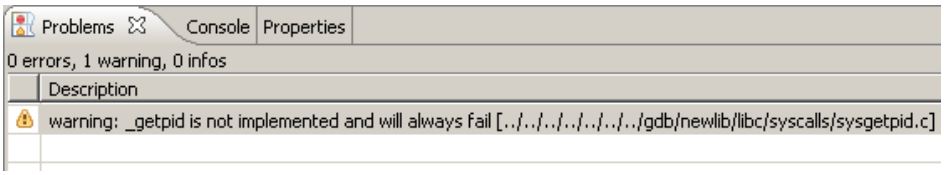
For example, generating an application based on the code shown in Figure 25 generates the compiler warning shown in Figure 26.

Figure 25: Code That Generates Compiler Warning

```
int main(void)
{
    int i;
    i= getpid();
return
}
```

Although the application was built successfully, the value of “i” subsequent to the getpid function call is meaningless.

Figure 26: Compiler Warning in Problems View



Those functions that are implemented as part of the platform library can be overridden, as described in “Overriding Default Driver Implementation” on page 124. The interaction of Newlib C file functions is described in “LatticeMico File Service” on page 98. Table 2 lists functions known to generate compile-time warnings because of dummy functionality implementation.

Table 2: Functions that Generate Compile-Time Warnings

Function	Header file
abort	stdlib.h
mktemp, mkstemp	stdio.h
remove	stdio.h
rename	stdio.h
tmpfile	stdio.h
tmpnam, tempnam	stdio.h
clock	time.h
raise	signal.h
signal	signal.h

Stand-Alone Printf Functionality

If your application relies on standard I/O functionality only for the printf function instead of standard I/O for full file functionality, you can significantly reduce the amount of code in your project by using the stand-alone printf function that Lattice Semiconductor provides. The stand-alone printf function provides stand-alone printf, puts, and putc implementations that are not

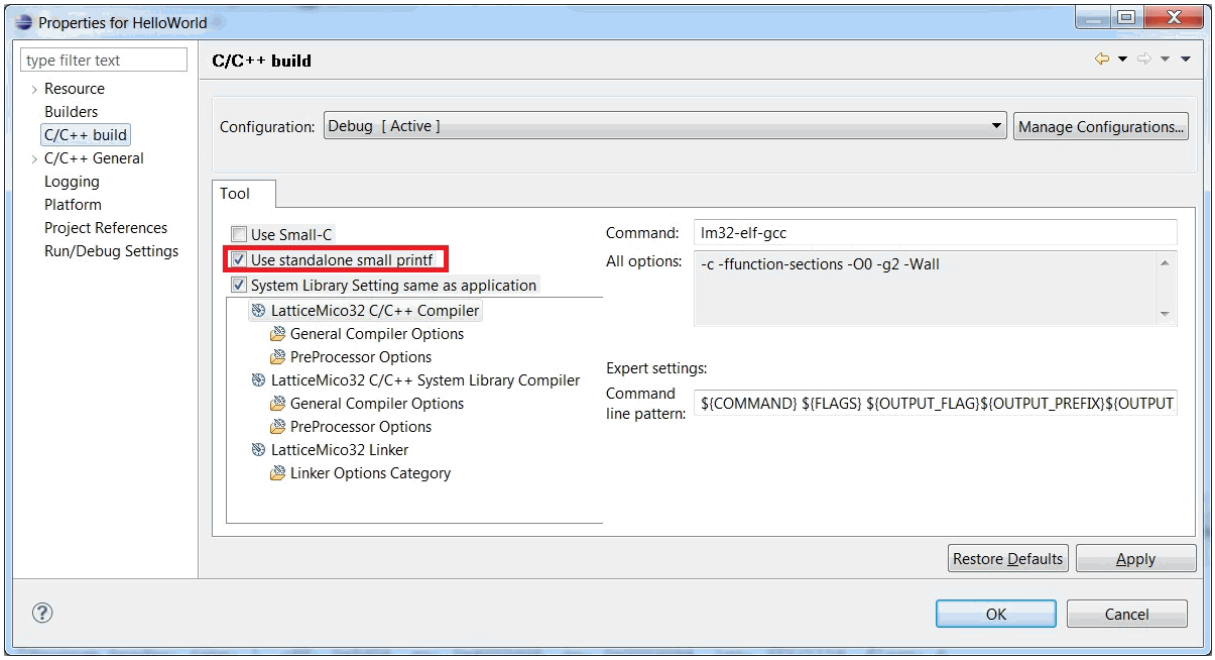
dependent on the functions contained in the C library. These functions are compiled as part of the platform library, described in “Platform Library-Generated Source Files” on page 155.

Note

Although the stand-alone printf functionality is turned on by default for new projects, it significantly reduces size only if you do not intend to use other file operation functions, such as fopen, fread, and fwrite, defined in stdio.h. If you plan to use file operation functions, disable the stand-alone small printf function and consider using the small C library, described in “Reduced-Functionality Small Newlib C Library” on page 50.

You can enable or disable this stand-alone printf feature by selecting the “Use standalone small printf” option for new projects in the project’s C/C++ Properties for Trace dialog box, as shown in Figure 27. Deselect this option to disable the stand-alone printf feature.

Figure 27: Enabling the Stand-Alone Printf Feature



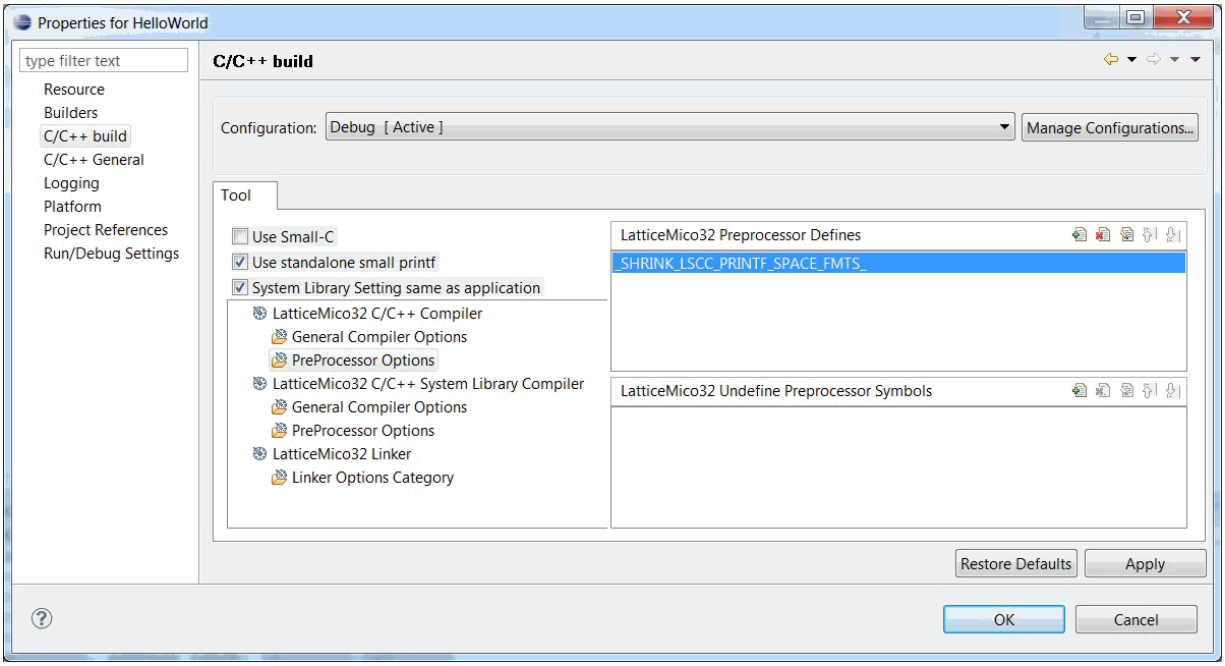
No code changes are required when you enable and disable this functionality. To use the stand-alone printf function, simply call “printf” in your application code, as you normally do.

The stand-alone printf function has the following limitations:

- Only the s, c, u, d, i, x, X, o, and p type specifications are supported. All other type specifications, such as f and e, are ignored and generate spurious characters.
- The format specifications (-, +, #, , 0) provided for the type specifications just given are ignored.
- Width and precision specifications for the type specifications just given are ignored.
- fflush has no impact on standard output. You must print a newline character to explicitly flush standard output.

You can support width and precision specifications, in addition to the format specifications of the supported printf type specifications, by using the _SHRINK_LSCC_PRINTF_SPACE_FMTS_ preprocessor definition in the project's C/C++ Properties for Trace dialog box, as shown in Figure 28.

Figure 28: Supporting Width and Precision Specifications



This function is not re-entrant and is not suitable in a multi-tasking environment. However, the source for this reduced printf function is provided in printf_shrink.c, located in the platform library source directory of the project. You can modify it to suit your particular needs, even to make it re-entrant.

Reduced-Functionality Small Newlib C Library

The Small Newlib C library is based on the Newlib C library source, but its functionality is reduced for the standard I/O functions defined in stdio.h, specifically the file functions (including standard input, output, and error), the

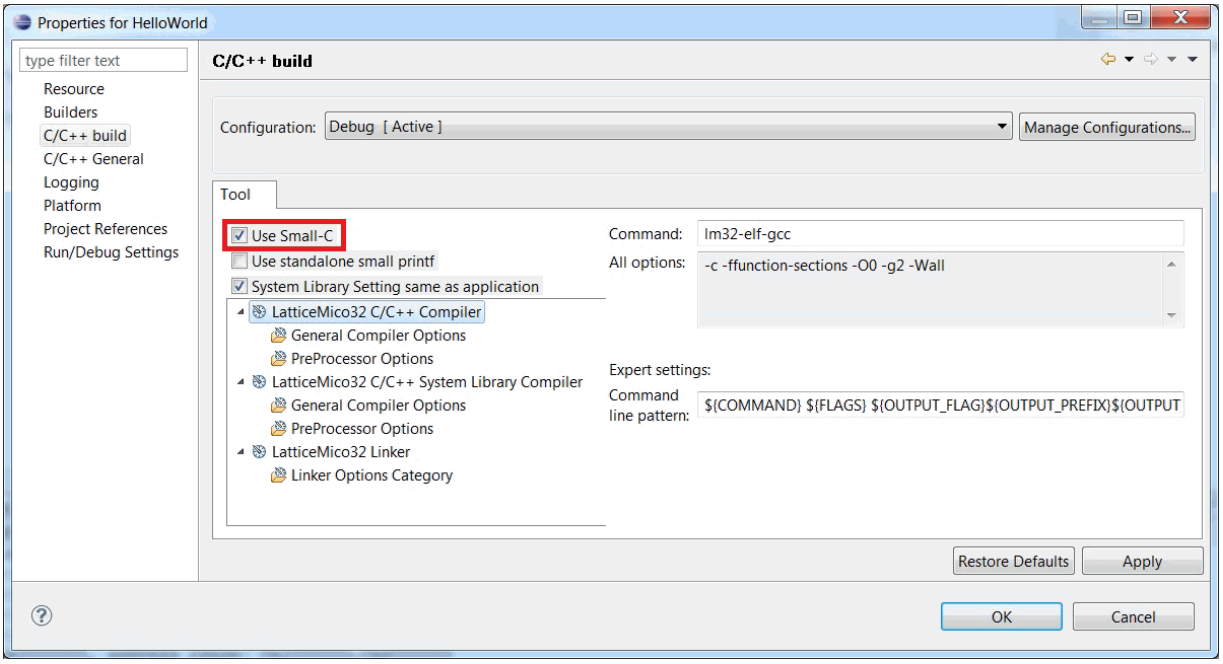
formatted input functions (printf, fprintf, and so forth), and the formatted output functions (scanf, fscanf, and so forth), as described in this section. The C library functions defined in header files other than stdio.h are identical to those in other libraries.

The Small C library option is turned off by default for new projects. To enable this functionality, select the Use Small-C option in the project's C/C++ Properties dialog box, as shown in Figure 29. When you select this option, the managed-build process modifies the project makefile to use libsmallc.a instead of libc.a.

Note

If you select the Small C library and the stand-alone printf option, the stand-alone printf function is used in the generated code, and the printf function from the Small C library is ignored. However, if your application also uses fprintf, the code size resulting from selecting the Small C library and the stand-alone printf function is larger than the code size resulting from selecting just the Small C library.

Figure 29: Selecting the Small C Library



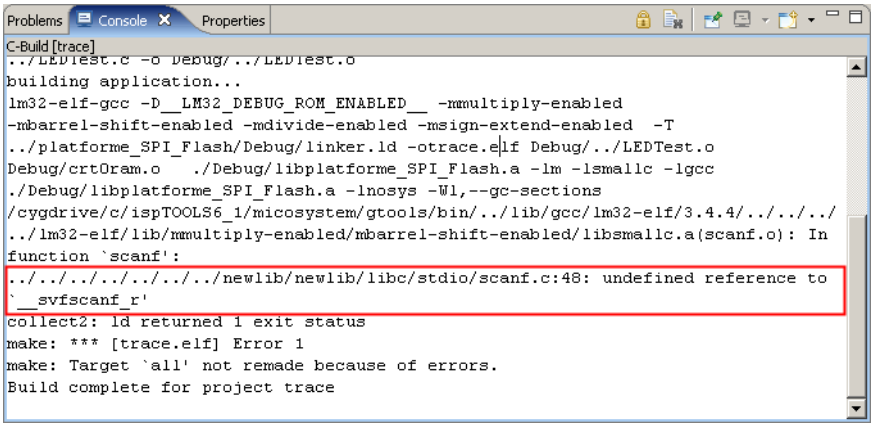
The small C library has the following limitations:

- "ungetc" is not supported, and any function calling it will cause a linker error to be issued.
- The "scanf" family of functions is not supported, including the following functions. These functions internally call functions that, in turn, call "ungetc":
  - "fscanf"
  - "scanf"

- "sscanf"

The unsupported functions cause a linker error to appear in the build console, as shown in Figure 30. The linker errors for all the unsupported functions indicate an inability to find reference to a "scanf" function type (__svfscanf_r in Figure 30) or a missing reference to "ungetc."

Figure 30: Linker Error



- "asiprintf" and "asprintf" are not supported, so invoking them causes a linker error to be issued.
- Buffered files have a buffer of 16 bytes for the small C library. The normal C library has a buffer of 1024 bytes.
- The "printf" function has the following limitations:
  - Only the s, c, u, d, l, x, X, o, and p type specifications are supported. All other type specifications, such as f and e, are ignored and generate spurious characters.
  - The format specifications (-, +, #', ", 0) provided for the type specifications just given are ignored.
  - Width and precision specifications for the type specifications just given are ignored.

Device Drivers and Services

The Mico System Builder (MSB) generates platforms that allow the LatticeMico32 microprocessor to interact with a wide range of possible devices. Also, there can be multiple instances of the same device. The piece of code that directly interacts with these devices to convert the more general I/O instructions of the operating system to messages that the device type can understand is known as the device driver. The device drivers bundled with LatticeMico are not meant for use in a multi-threading environment.

Some of these devices, either the same device type present as multiple instances or different devices providing similar functionality, can be grouped so that you do not need to know the specifics of a device. For example, you

can perform standard I/O operations without having to know what the device driver does or what the specific device characteristics are, even though the device handling the standard I/O may be either an RS-232 UART driven by an RS-232 UART device driver or the microprocessor's JTAG UART module driven by the LatticeMico JTAG UART device driver. These software abstractions that hide detailed device functionality are known as services.

The device-specific software device driver information 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.

Services Available at Run Time

This section lists the functions available to use from your application. It does not list the functions that would be needed to develop your own device drivers. For developing device drivers, see "Modifying Existing Device Drivers" on page 123.

Newlib C Library and Newlib Math Library Functions

You can use the Newlib C library and Newlib math library routines, including the file input/output routines, as mentioned in the "Newlib C and Math Libraries" on page 46. For file operations, including standard I/O operations, you can use these routines irrespective of the platform; that is, your program will still successfully compile and link.

However, if your platform does not contain a microprocessor with debug enabled or a UART, your file operations and standard I/O operations will fail unless you have implemented a file device. Refer to "LatticeMico File Service" on page 98, which describes LatticeMico File Services in detail.

Microprocessor-Related Services

Table 3 lists the available microprocessor-related functions that you can use at run time. For detailed description on these functions, refer to the sections just mentioned.

Table 3: Microprocessor-Related Functions Available at Run Time

Functional Category	Functions	Header File

Interrupt Management	mico_status MicoRegisterISR(unsigned int IntLevel, void *Context, ISRCallback Callback);
	
	mico_status MicoDisableInterrupt(unsigned int IntLevel);
	
	mico_status MicoEnableInterrupt(unsigned int IntLevel);
	
	unsigned int MicoDisableInterrupts(void);
	
	void MicoEnableInterrupts(unsigned int intrMask);
		MicoInterrupts.h

Table 3: Microprocessor-Related Functions Available at Run Time

Cache Management
 	
Note: You can use these functions in your program, even though your microprocessor may be configured not to use caches.
	void MicoFlushInstrCache(void);
	
	void MicoFlushDataCache(void);
		LatticeMico32.h

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

Macros	MILLISECONDS_TO_TICKS(X_MS) MICROSECONDS_TO_TICKS(X_MICROSECS)	MicoMacros.h
Platform Clock Speed Macro **Note**: The managed build process based on the platform configuration dynamically identifies this value.	MICO32_CPU_CLOCK_MHZ Identifies CPU Clock speed (in MHz, e.g. 66000000)	DDStructs.h

Device Lookup Services

The device lookup services shown in Table 4 are available as part of your platform, as long as your .msb file contains a LatticeMico32 microprocessor in its definition. The return values depend on whether any device within your platform actually registers itself as available for device lookup by name. Refer to “Device Lookup Service” on page 91 for more details on the device lookup service.

Table 4: Device Lookup Summary

Functional Category	Functions	Header File
Lookup Device By Name	void *MicoGetDevice(const char *name) void *MicoGetFirstDev(const char *deviceType, DevFindCtx_t *FindCtx) void *MicoGetNextDev(DevFindCtx_t *FindCtx)	LookupServices.h
Device Registration	unsigned int MicoRegisterDevice(DeviceReg_t *pDevReg)	LookupServices.h

System Timer Services

System timer services are available as part of your platform, as long as they contain at least one timer instance. If your platform does not contain a timer instance and your code invokes the functions listed in this section, you will receive a compilation error. You must explicitly register a timer instance as the system timer, using the API listed in Table 5.

Table 5: System Timer Summary

Functional Category	Functions	Header File
System Timer Registration **Note**: You must explicitly register a timer device using this API as a system timer.	MicoTimerCtx_t* RegisterSystemTimer(MicoTimerCtx_t *ctx, unsigned int TickInMS);	MicoTimerService.h
System Timer Callback Registration **Note**: This function has meaning only after registering a system timer.	void MicoRegisterActivity(MicoSysTimerActivity_t activity, void *ctx);	MicoTimerService.h
CPU Tick Retrieval **Note**: This function has meaning only after registering a system timer.	void MicoGetCPUTicks(unsigned long long int *ticks);	MicoTimerService.h

CFI Flash Device Service

While a flash device is read just as a normal read/write memory device, writing to a flash device involves configuring the flash device through a sequence of flash accesses, as specified by the CFI command set for the flash component. This section lists the available CFI flash device service APIs for writing to, erasing, and obtaining sector information for a flash component.

Although the functions listed in Table 6 are available to use, their actual functionality depends on the configuration driver available and the end points that it supports. For example, the only flash configuration driver available is the two 16-bit flash components supporting the AMD standard command set. Although you can use these APIs for a different flash configuration, these APIs will return run-time error codes, if you have not provided a configuration device driver.

When the flash or parts of flash are erased, the erased flash memory contents are set to logical 1. After this process, the 1 can be converted to a 0 by performing a write operation. However, once a 1 is written to a 0, it cannot be rewritten to a 1. The only way to write new data that requires a 1 to be converted to a 0 is by erasing the affected location and then writing the new data. So, once a flash location is “programmed” or “written to,” it must be erased before writing new data at the same location.

The difference between the FlashWrite functions and the FlashProgram function listed in the read/write functional category is this: FlashWrite functions assume that you have erased the affected flash regions before invoking them, but the FlashProgram function erases the affected sectors before writing the requested data.

Table 6: CFI Flash Device Services

Functional Category	Functions	Header File

Read/Write Operations	unsigned int LatticeMico32CFIFlashWrite(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned char *Data, unsigned int Bytes);
	
	unsigned int LatticeMico32CFIFlashWrite32 (CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned int Data);
	
	unsigned int LatticeMico32CFIFlashWrite16(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned short int Data);
	
	unsigned int LatticeMico32CFIFlashWrite8(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned char Data);
	
	unsigned int LatticeMico32CFIFlashProgramData(CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned char *pData, unsigned int bytes);
		LatticeMico32CFI.h

Erase Operation	unsigned int LatticeMico32CFIFlashEraseDevice(CFIFlashDevCtx_t *ctx);
	
	unsigned int LatticeMico32CFIFlashEraseBlock(CFIFlashDevCtx_t *, unsigned int ByteOffset);
		LatticeMico32CFI.h

Flash Query	unsigned int LatticeMico32CFIFlashSectorInfo(CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned int *SAddr, unsigned int *ByteLen);	LatticeMico32CFI.h
Flash Reset	unsigned int LatticeMico32CFIFlashReset(CFIFlashDevCtx_t *ctx);	LatticeMico32CFI.h

Device Driver APIs

The device driver APIs are device-specific functions. The Mico System Builder (MSB) includes device drivers for the following components:

- RS-232 UART
- Timer
- GPIO
- DMA
- SPI

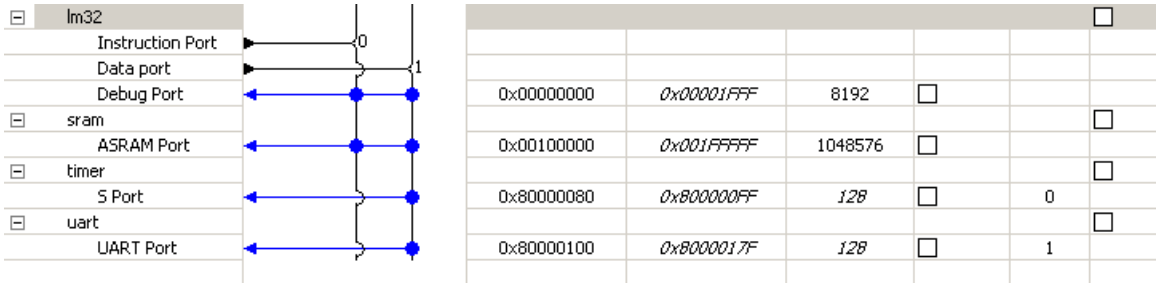
The APIs directly manipulate these devices, along with their register layout structures, as described in the respective component data sheets. These data sheets also contain component usage examples.

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.

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 31 illustrates the example platform structure.

Figure 31: HelloWorld Platform



Note

The procedures presented in this section are not a substitute for the LatticeMico32 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 LatticeMico Managed C/C++ build process is used for building the “hello world” application.

  Note

  The “hello world” application explained here is not related to the “hello world” software template that is available in the software.

- The “HelloWorldPlatform” platform consists of the following components:
  - Timer instance named “timer”
  - UART instance named “uart”
  - LatticeMico32 microprocessor instance named “lm32”
  - Asynchronous memory component named “sram”
- The UART is selected as the standard input, output, or error device in the Platform tab in the C/C++ perspective’s Properties dialog box. It is configured to use interrupts.

  See information in “Setting Project Properties” on page 26 for details on platform settings.

- All linker sections are mapped to SRAM.

Creating a Blank Project

As the first step, you must create a project based on the platform criteria outlined previously in “Basic Program Structure” on page 57.

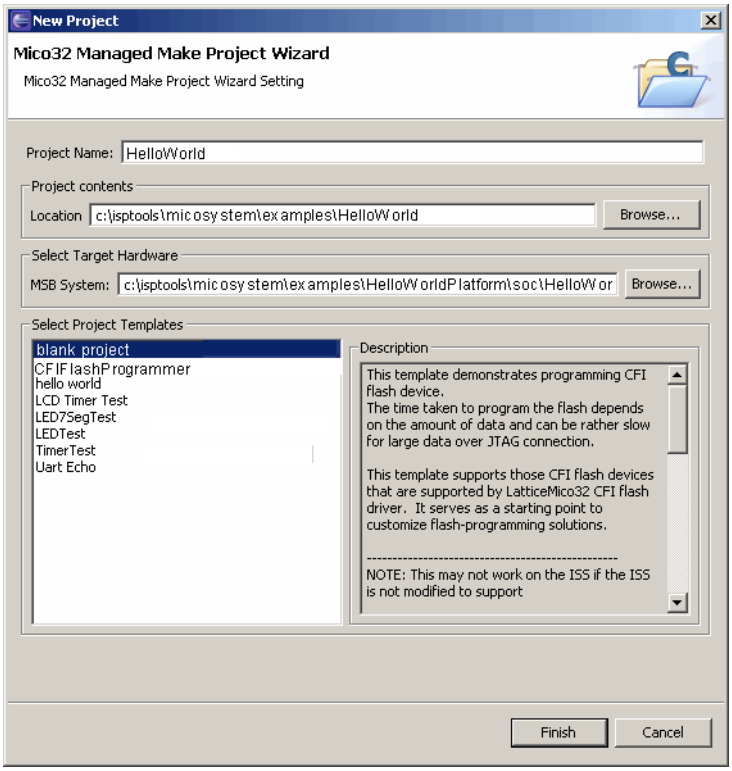
To create a blank project in the Project Wizard:

1. In the C/C++ SPE perspective, choose **File > New > Mico32 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 **HelloWorldPlatform** target hardware platform, using the Browse button in the MSB System text box.

   This is an example .msb file that is packaged with the software.
5. Select **Blank Project** in the Select Project Templates list box in the lower left portion of the dialog box.

   The New Project dialog box should now resemble the illustration in Figure 32.

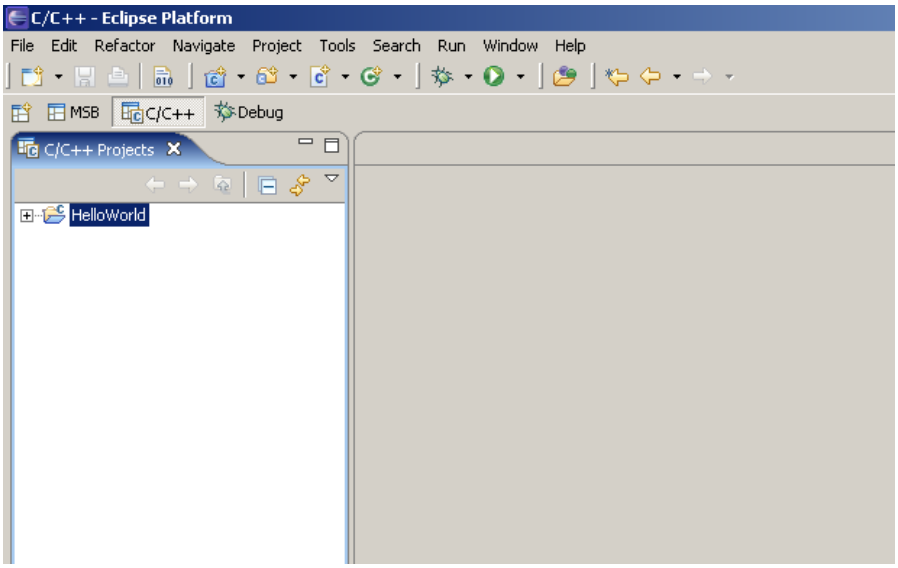
Figure 32: New Project Dialog Box



6. Click **Finish**.

This newly created project should now be visible in the C/C++ perspective's Projects view, as shown in the Figure 33.

Figure 33: New Project in Projects View



Adding a Source File to the Project

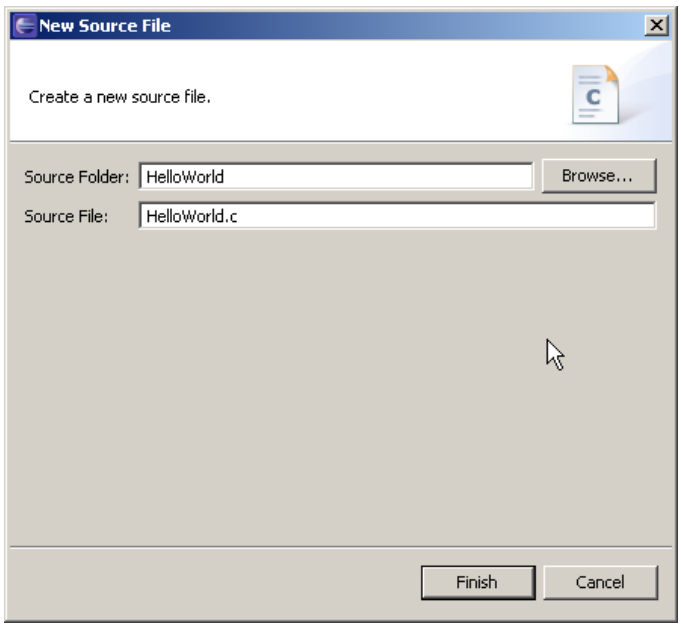
You will now add a new source file to your newly created project. Source files refer to your source C language files.

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**.

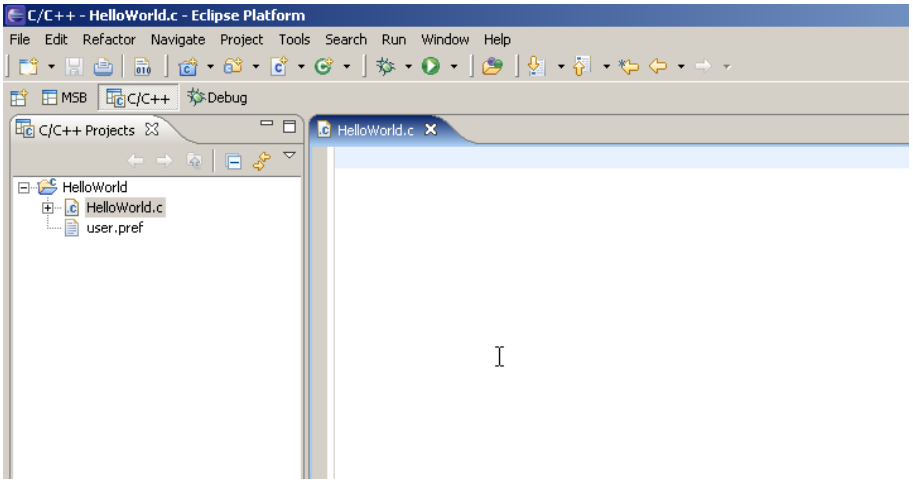
3. In the New Source File dialog box, shown in Figure 34, enter **HelloWorld.c** in the Source File text box.

Figure 34: New Source File Dialog Box



This new file is now visible beneath the project in the C/C++ perspective's Projects view, as shown in Figure 35.

Figure 35: Source File In Projects View

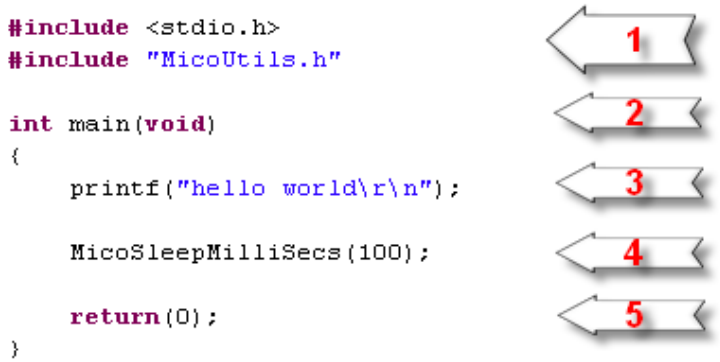


In addition, you may see the user.pref file, which is automatically generated by the C/C++ SPE managed build process and should not be modified or deleted. The user.pref file is described in in "C/C++ Perspective Project Folder File Contents" on page 149.

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 uses ANSI C standard I/O function (for example, printf) to simply print “hello world.” To do this, add the following code in the Helloworld.c source file that you created in the prior step.

Figure 36: Helloworld.c Source Code



The lines shown in this code example are described following:

- Item 1 – These two #include statements declare the header files needed to verify the function prototypes of the functions used in the code. The stdio.h value refers to the ANSI C-defined header file that contains prototype declarations for the standard I/O functions used in the code. The MicoUtils.h value refers to the standard LatticeMico32 header file that contains the prototype declaration of the function listed in item 4.
- Item 2 – 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 “The int main(void) Function” on page 73.
- Item 3 – Use the printf parameter to print a sequence of characters to the standard I/O device. Subsequent sections show how the UART component is designated as the standard I/O device.
- Item 4 – 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 that you have queued for sending through the printf statement. Typically, the UART baud rate is much slower than the CPU speed, so this delay is required. This function is part of the LatticeMico32 platform library, specifically the CPU service, and its prototype is declared in the MicoUtils.h header file.
- Item 5 – 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 145.

   The Projects view in the C/C++ perspective is updated to show the generated artifacts, as shown in Figure 37.

Figure 37: Projects View After Build Process Run

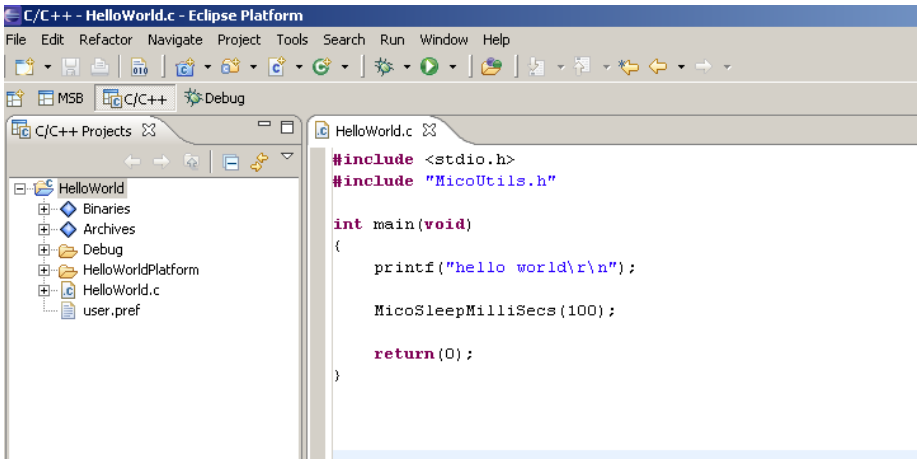
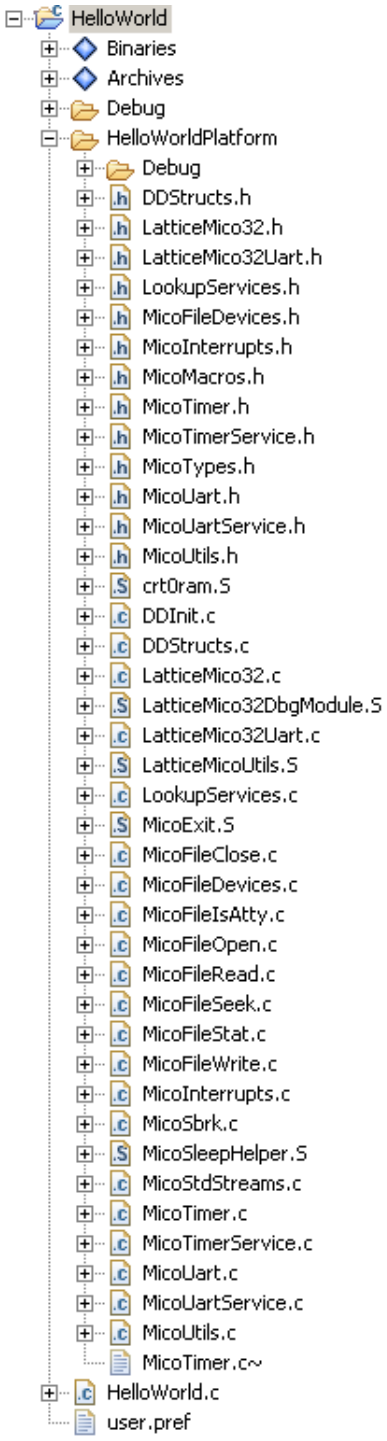


Figure 38 shows the contents of the platform library for the example.

Figure 38: Contents of Platform Library



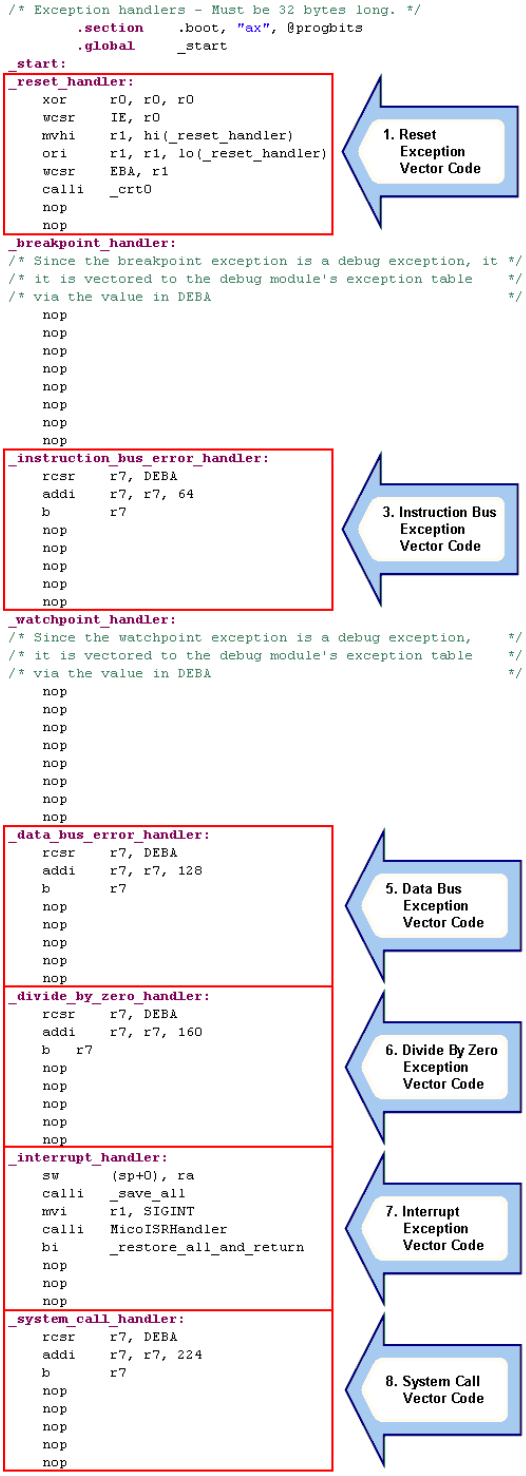
“Platform Library Folder” on page 151 describes the various items within the Platform Library folder.

Boot Sequence and crt0ram.S

An assembly language file named crt0ram.S in the platform library folder contents contains the boot-up sequence. The code in crt0ram.S is shown in the example sections in Figure 39.

.

Figure 39: Boot-Up Sequence in the Assembly .S File




```
/* Exception handlers - Must be 32 bytes long. */
        .section    .boot, "ax", @progbits
        .global     _start
_start:
_reset_handler:
    xor     r0, r0, r0
    wcsr    IE, r0
    mvhi    r1, hi(_reset_handler)
    ori     r1, r1, lo(_reset_handler)
    wcsr    EBA, r1
    calli   _crt0
    nop
    nop
_breakpoint_handler:
/* Since the breakpoint exception is a debug exception, it */
/* it is vectored to the debug module's exception table    */
/* via the value in DEBA                                   */
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
_instruction_bus_error_handler:
    rcsr    r7, DEBA
    addi    r7, r7, 64
    b       r7
    nop
    nop
    nop
    nop
    nop
_watchpoint_handler:
/* Since the watchpoint exception is a debug exception,    */
/* it is vectored to the debug module's exception table    */
/* via the value in DEBA                                   */
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
_data_bus_error_handler:
    rcsr    r7, DEBA
    addi    r7, r7, 128
    b       r7
    nop
    nop
    nop
    nop
    nop
_divide_by_zero_handler:
    rcsr    r7, DEBA
    addi    r7, r7, 160
    b   r7
    nop
    nop
    nop
    nop
    nop
_interrupt_handler:
    sw      (sp+0), ra
    calli   _save_all
    mvi     r1, SIGINT
    calli   MicoISRHandler
    bi      _restore_all_and_return
    nop
    nop
    nop
_system_call_handler:
    rcsr    r7, DEBA
    addi    r7, r7, 224
    b       r7
    nop
    nop
    nop
    nop
    nop
```

For a more general description of the boot-up sequence, refer to “Boot Sequence” on page 76.

The first piece of the boot code in crt0ram.S shown in Figure 39 corresponds to the exception vector table. The boot code is executable code; that is, it contains LatticeMico32 instructions. This code is located at the start of the program memory, which is selected through the Platform tab by the default linker script generated by the managed build process. The first piece of code in the “.text” linker section is the exception vector table.

As described in the *LatticeMico32 Processor Reference Manual*, two important registers dictate the exception vector locations: EBA (exception base address) and DEBA (debug exception base address). The debug port address assigned by MSB is the DEBA value and corresponds to the exception vector table of the debug module. The EBA is configured through the Processor Configuration dialog box.

The EBA and DEBA point to identical exception tables for handling the eight different exception types. The difference between EBA and DEBA is that the DEBA table is used for debug exceptions, such as breakpoints and watchpoints, and the EBA is used for non-debug exceptions, such as the reset vector and external interrupts. The LatticeMico32 debug module implements default handlers for all exception vectors.

The debug vectors are handled by code in the debug module that responds to breakpoints and watchpoints. Other exceptions halt operation and display an error message such as a “Divide-by-Zero Error.” The Reset vector is directed to the boot code of the LatticeMico System. The External interrupt vector invokes a handler that dispatches to your registered ISR callback. In an MSB managed build project, the application does not have to directly handle any of these interrupt vectors, and you should only use the interrupt management APIs to register or deregister an ISR. The total space available for each exception vector code in the vector table is limited to eight 32-bit locations.

The following sections describe items illustrated in Figure 39.

Reset Exception Handling

For the first item shown in Figure 39, the reset exception handling is invoked on microprocessor power-up. On power-up, the microprocessor jumps to the address contained in the EBA register. This EBA register is configured at the platform design phase in MSB. Since there are only eight 32-bit locations, the reset exception handling must be deferred to routines located elsewhere in memory that do not overlap with the exception vector table.

As part of a debug session, the Debugger downloads the application code in the appropriate memory, as specified by the linker script, then makes the microprocessor resume execution from that application’s “_start” location. This “_start” is a label in the crt0ram.S assembly source that marks the reset exception vector as the start location.

The default code within the reset vector table does the following:

- Resets R0 register to 0 – The GNU GCC compiler expects R0 to always contain 0. Since LatticeMico32 microprocessor implementation does not hardwire R0 to 0, the first operation performed is to set the value of register R0 to 0.
- Disables interrupts – Until the software has a chance to reinitialize the system (for example, initialize components and register ISRs), interrupts are disabled at this stage. If the software on the microprocessor is reset without reprogramming the FPGA with the configuration bitstream, the associated platform peripherals will not be in a known reset state. It is therefore possible for a component to have a pending interrupt as the microprocessor comes alive and executes code from the reset location. The default handler, MicoISRHandler, takes care of disabling interrupts that do not have a corresponding interrupt handler, as is the case until the user application or device drivers actually register an interrupt service routine.
- Reloads the EBA value – As mentioned earlier, the EBA value is configured through the Processor Configuration dialog box. This EBA typically contains the memory address for the final deployed software. However, for debug situations, you can download the application in a volatile memory location, but the EBA may point to a non-volatile memory location (such as flash memory) that will contain the final deployed software application.

  If, as part of the debug session, this EBA is not modified to point to the downloaded application's vector table, the EBA will point to a memory location that does not contain the application that is being debugged. In this case, any interrupt raised by a component will not be handled by the downloaded application that is being debugged. The EBA must be set to the running application's reset vector location, that is, the start of the exception vector table.
- Invokes crt0 – Once the processes just listed are completed, execution jumps to the crt0 label, which takes care of hosting the user application. "The crt0 Function" on page 68 describes this piece of code.

Interrupt Exception

Since the amount of space is limited to eight 32-bit locations, the interrupt processing must reside in a memory not overlapping with the vector table locations.

The default code for the interrupt exception performs the following functions:

- Saves the return addresses as the default interrupt process making function calls.
- Preserves the state of the CPU registers before performing interrupt processing so that these can be restored when returning from interrupt processing. This is described in detail in a subsequent section.
- Calls the MicoISRHandler function. This function is implemented in MicoInterrupts.c and can be overwritten by following the instructions mentioned in "Overriding Default Driver Implementation" on page 124. This MicoISRHandler is the default interrupt handler and calls back the

appropriate interrupt routines that are registered through MicoRegisterISR, described in “Interrupt Management” on page 81. MicoISRHandler acknowledges the microprocessor interrupt by setting the appropriate bit in the Interrupt Pending register once the user-registered callback returns.

The default interrupt handling does not allow interrupt nesting; that is, it does not interrupt a user-registered ISR callback routine. It services the interrupts on a highest-interrupt-first basis, so bit 0 of the Interrupt Pending register is treated as the highest-priority pending interrupt, and bit 31 of the Interrupt Pending register is treated as the lowest-priority pending interrupt.

The default interrupt handler performs the logical AND function of the 32-bit interrupt pending register, and the 32-bit interrupt mask register tries to determine if there are valid interrupts that need servicing. The default interrupt handler, MicoISRHandler, performs this check each time after calling the user-registered ISR for the highest priority pending interrupt, thereby implementing a highest-priority-first interrupt servicing policy. The default interrupt handler does not perform a “return from interrupt” until there are no more interrupts left to service.

- Restores the state of the CPU registers before returning from the interrupt ISR. This restores the state of the registers for the thread of execution that was interrupted. The return-from-interrupt call from “restore_all,” described in “Context Save/Restore in Interrupt Exception” on page 73, enables the interrupt-enable bit in the interrupt enable register of the microprocessor.

Other Exception Handlers

This section describes items illustrated in Figure 39 on page 65. As shown in the figure, item 2 and item 4 are missing. They are not included because item 2 is a breakpoint exception and item 4 is a watchpoint exception. These two exceptions are debug exceptions and are handled by the debug module, vectored through DEBA.

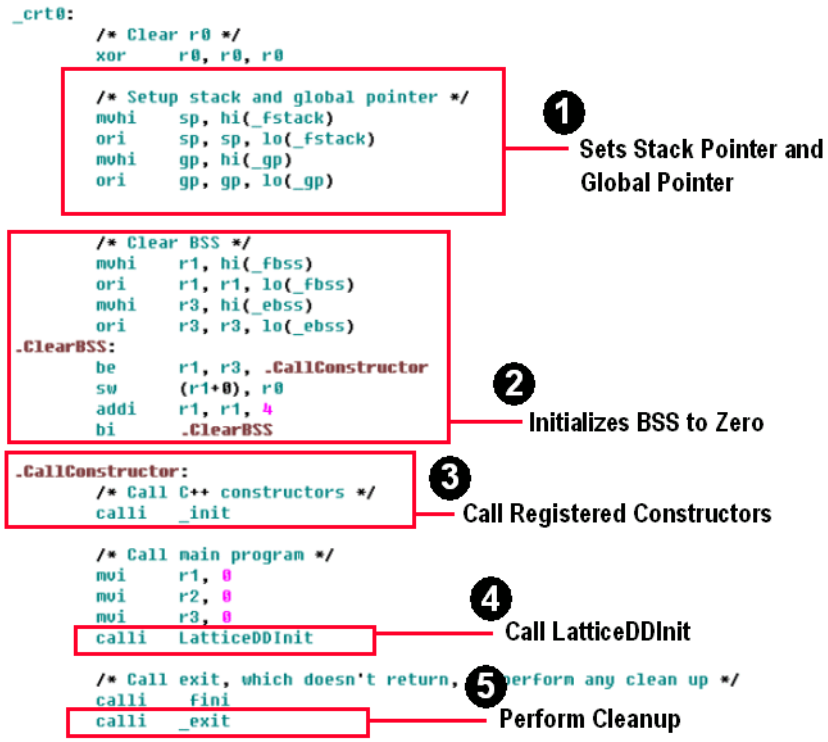
Items 3, 5, 6, and 8 correspond to an instruction bus exception, data bus exception, divide-by-zero exception, and the system call exception. The default implementation is to branch to the debug module’s exception handling implementation. The way it performs this, as illustrated in the code example in Figure 39, is that it reads the DEBA register value, adds the appropriate offset to the DEBA value, and then branches to that address. This address corresponds to the exception’s vector location in the debug module. The debug module’s implementation for these exceptions is to indicate the occurrence of the exception in the Debugger.

The crt0 Function

The crft0 function, or label, in crt0ram.S is the actual boot-up code that is executed as a result of a call from the microprocessor reset vector code. The code implemented as part of crt0 is responsible for calling the LatticeDDInit

managed-build, driver-initialization routine that, in turn, invokes your “main” program. The example code in this section shows the contents of this function as implemented in crt0ram.S.

Figure 40: Steps Performed as Part of crt0



The steps performed as part of crt0, shown in Figure 40, are as follows:

- Establishes the stack pointer – The microprocessor’s stack pointer must point to a valid memory location. This location, _fstack, is defined in the default linker script generated by the managed build process and is the topmost address (largest value) of the read/write memory selection in the C/C++ SPE platform settings.
- Clear BSS section – This section contains variables that are used in the application and must be zeroed out before the "main" application is executed.
- Calls constructors – GCC allows for the declaration of functions as constructors. Constructors must be called before the application body is executed.
- Calls LatticeDDInit – The crt0 call invokes LatticeDDInit, which is dynamically generated by the managed-build process. This function, as further described in “LatticeDDInit” on page 70, invokes the device-driver routines and then invokes your “main” program implementation. On returning from the “main” implementation, LatticeDDInit returns to crt0.
- Calls exit – Since the application execution is complete, crt0 proceeds to call the registered destructor functions. Once crt0 has completed calling

the registered destructor functions, it invokes _exit, which passes control over to the Debugger.

LatticeDDInit

The LatticeDDInit function is dynamically generated by the managed build process on the basis of the .xml file for each component that is defined in the platform. This function resides in the DDInit.c source file.

Note

LatticeMico32 interrupt management functions can be invoked at any step in LatticeDDInit, that is, in the component initialization routines or your “main” implementation invoked by LatticeDDInit.

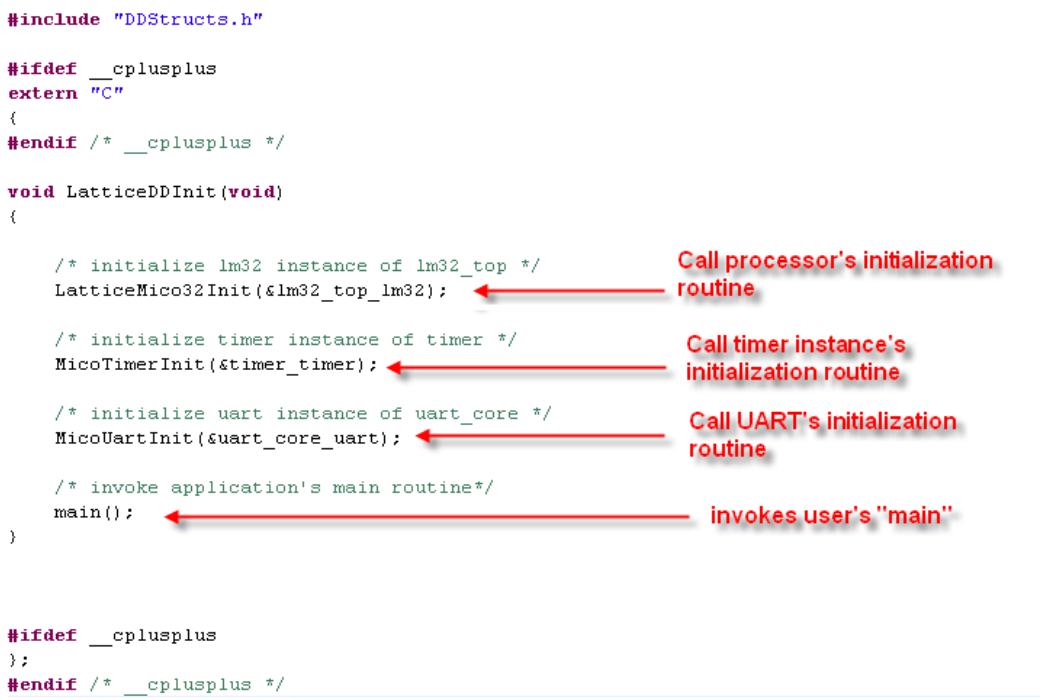
You can override the default implementation of LatticeDDInit that is dynamically generated. The primary function of LatticeDDInit is to call the initialization function that specifies the device driver initialization routines for each component in the platform. The name of the function called for a given component is specified in that component’s particular .xml file. If no function was specified in the .xml, none is called. A component’s initialization function is called for each instance of that component in the platform.

In the example code shown later in this section, LatticeDDInit calls the initialization routines for the microprocessor, timer, and the UART, in no specific sequence. Since there is a single instance of each component, LatticeDDInit calls the initialization routine only once. The initialization routine is specified in the .xml file, along with the argument type. The argument type is the component information structure declaration, declared in DDStructs.h, and the argument is a pointer to the unique instance of the component information structure, defined in DDStructs.c. See “DDStructs.h File” on page 157 and “DDStructs.c File” on page 159 for more details on this component information.

For multiple instances of the same component, the initialization routine for that component is invoked once for each instance, with each invocation having a unique component-specific information structure as the argument. What the initialization routine for each component does is specific to that component’s device driver implementation. The device driver calls are responsible for triggering the initialization of services that they are associated with, if necessary, because the services do not have explicit initialization routines that are invoked. For example, the microprocessor initialization routine or the UART initialization routine can trigger initialization of the LatticeMico File Services.

The LatticeDDInit code is shown in Figure 41:

Figure 41: LatticeDDInit Code



Once the device driver initialization routines have been completed, LatticeDDInit proceeds to call the user-implemented int main(void) function.

Microprocessor Initialization Routine

LatticeDDInit invokes the microprocessor initialization routine when the microprocessor's .xml file contains a request that it be invoked. The details of the microprocessor initialization routine can be found in the LatticeMico32.c source file.

Since all platforms are expected to have a microprocessor, the microprocessor initialization routine's key activity is the setup of the microprocessor's JTAG UART as a file device if the debug module is included with the CPU. This initialization results in a call to the LatticeMico File Service to register the microprocessor's JTAG UART as an available file device, which in turn causes the LatticeMico File Service to initialize itself.

UART Initialization Routine

The UART initialization routine implementation performs the following activities:

- Registers itself as an available file device with the LatticeMico File Service.

- Registers the UART instance as an available UART device with the lookup services.
- Initializes the UART instance for transmission and reception of data.

Timer Initialization Routine

The timer initialization routine implementation performs the following activities:

- Registers the timer instance as an available timer device with the lookup services
- Initializes the timer instance for use by your application

Setting Standard I/O

The managed build process generates a file, MicoStdStreams.c, which contains the name of the device that is assigned during platform generation. It must handle a given standard stream, that is, input, output, or error. Figure 42 shows the contents of MicoStdStreams.c file.

Figure 42: MicoStdStreams.c Code

```
/* This file defines string-constants representing stdstream devices */
/* THIS FILE IS DYNAMICALLY GENERATED AND SHOULD NOT BE MODIFIED     */
#ifdef __cplusplus
extern "C" {
#endif

const char* MICO_STDIN_DEV_NAME = "uart";
const char* MICO_STDOUT_DEV_NAME = "uart";
const char* MICO_STDERR_DEV_NAME = "uart";


#ifdef __cplusplus
}
#endif
```

This file is generated according to the C/C++ SPE settings. These constants are used by the LatticeMico File Service to detect the correct device when a new device registers itself as an available file device. If a device with a matching name registers itself as a file device, the LatticeMico File Service requests that the registering device prepare itself to handle the appropriate standard stream. See “LatticeMico File Service” on page 98 for additional details on LatticeMico File Service.

The microprocessor with its debug module and associated JTAG UART both register themselves as available file devices as part of their initialization routine. As part of this registration with LatticeMico File Service, the LatticeMico File Service identifies the UART instance on the basis of information in MicoStdStreams.c as the device to handle all of the streams

and then accordingly invokes the file device interface functions to prepare the UART instance for handling the standard streams.

If no device with a matching name is found, standard I/O function calls such as printf, gets, and scanf will return with the appropriate error codes.

The int main(void) Function

LatticeDDInit invokes your “main” function. You are responsible for implementing the “main” function, which serves as an entry point to your application.

The contents of “main” are application-specific. Within “main” or any user function, you can use the Newlib C library and Newlib math library function calls, as well as the services described in “Run-Time Libraries” on page 45. Additionally, you can directly interact with the platform’s components, using the provided device drivers and user-supplied drivers, or by performing read and write operations using C language pointers. The content of your “main” application is illustrated in “Adding Source to the Source file” on page 61.

Context Save/Restore in Interrupt Exception

This subsection introduces you to the context save/restore calls in the crt0ram.S file.

When an exception occurs, the normal execution of the microprocessor—that is, the main thread of execution—is interrupted to execute the exception-handling code. The first operation that must take place is to save the “context” of the interrupted thread of execution. The term “context” used here refers to the microprocessor state at the point of the exception. By saving this microprocessor state before exception handling, this state can be restored once the exception is handled. This behavior allows the interrupted thread to resume processing without any problems.

There are three main types of registers: caller-saved, callee-saved, and machine-status registers, such as the exception address register and the return address register. The compiler tool chain takes care of generating code so that the necessary caller-saved registers are saved before a function call and that in a function call, the appropriate callee-saved registers are saved onto the stack. When the function call returns just before returning, the callee-saved registers that were modified are restored from the stack. After performing the function call return, the caller-saved registers that were used are restored from the stack.

The compiler tool chain cannot take into account an exception because an exception can occur at any time. The compiler must depend on the exception-handling code to save the appropriate state. At a minimum, the exception-handling code must save the caller-saved registers. The callee-saved registers are saved as part of function calls. For a non-multitasking

environment, it is essential to save only the caller-saved and machine-status registers.

In the code example shown in Figure 43, the code listed for _save_all shows the registers that are saved onto the stack. Normally, as part of the managed build process, only the caller-saved and status registers are saved onto the stack. If you wish, you can save the entire stack by defining that in the MICO32_FULL_CONTEXT_SAVE_RESTORE preprocessor definition.

Figure 43 shows the context save code that is called as part of the interrupt exception handling.

Figure 43: Context Save Code in Interrupt Exception Handling

```
_save_all:
    addi    sp, sp, -128
    sw      (sp+4), r1
    sw      (sp+8), r2
    sw      (sp+12), r3
    sw      (sp+16), r4
    sw      (sp+20), r5
    sw      (sp+24), r6
    sw      (sp+28), r7
    sw      (sp+32), r8
    sw      (sp+36), r9
    sw      (sp+40), r10
#ifdef MICO32_FULL_CONTEXT_SAVE_RESTORE
    sw      (sp+44), r11
    sw      (sp+48), r12
    sw      (sp+52), r13
    sw      (sp+56), r14
    sw      (sp+60), r15
    sw      (sp+64), r16
    sw      (sp+68), r17
    sw      (sp+72), r18
    sw      (sp+76), r19
    sw      (sp+80), r20
    sw      (sp+84), r21
    sw      (sp+88), r22
    sw      (sp+92), r23
    sw      (sp+96), r24
    sw      (sp+100), r25
    sw      (sp+104), r26
    sw      (sp+108), r27
#endif
    sw      (sp+120), ea
    sw      (sp+124), ba
    /* ra and sp need special handling, as they have been modified */
    lw      r1, (sp+128)
    sw      (sp+116), r1
    mv      r1, sp
    addi    r1, r1, 128
    sw      (sp+112), r1
    xor     r1, r1, r1
    wcsr    ie, r1
    ret
```

Once the exception is processed, the saved registers must be restored to reset the microprocessor state for resuming execution of the thread of execution that was interrupted.

Figure 44 shows the context restore code. This function basically reverses the steps performed by the _save_all code that saved the context, restoring it to its previous state.

Figure 44: Context Restore Code

```
_restore_all_and_return:
    addi    r1, r0, 2
    wcsr    ie, r1
    lw      r1, (sp+4)
    lw      r2, (sp+8)
    lw      r3, (sp+12)
    lw      r4, (sp+16)
    lw      r5, (sp+20)
    lw      r6, (sp+24)
    lw      r7, (sp+28)
    lw      r8, (sp+32)
    lw      r9, (sp+36)
    lw      r10, (sp+40)
#ifdef MICO32_FULL_CONTEXT_SAVE_RESTORE
    lw      r11, (sp+44)
    lw      r12, (sp+48)
    lw      r13, (sp+52)
    lw      r14, (sp+56)
    lw      r15, (sp+60)
    lw      r16, (sp+64)
    lw      r17, (sp+68)
    lw      r18, (sp+72)
    lw      r19, (sp+76)
    lw      r20, (sp+80)
    lw      r21, (sp+84)
    lw      r22, (sp+88)
    lw      r23, (sp+92)
    lw      r24, (sp+96)
    lw      r25, (sp+100)
    lw      r26, (sp+104)
    lw      r27, (sp+108)
#endif
    lw      ra, (sp+116)
    lw      ea, (sp+120)
    lw      ba, (sp+124)
    /* Stack pointer must be restored last, in case it has been updated */
    lw      sp, (sp+112)
    nop
    eret
```

Boot Sequence

You have already seen a detailed illustration of sample boot code in “Boot Sequence and crt0ram.S” on page 64. 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 LatticeMico32 microprocessor architecture. The *LatticeMico32 Processor Reference Manual* contains a full description of the microprocessor architecture.

EBA and DEBA

From a software boot perspective, the most important parameter in the LatticeMico32 microprocessor configuration is the EBA, also known as the exception base address. As you would expect, this parameter is used to deal with run-time errors caused by unexpected events and even predictable errors or unusual results. The address location value of the EBA is set by the Location of Exception Handlers option in the Add LatticeMico32 dialog box. See Figure 135 on page 185 and related instructions on resetting this value. On platform generation, the address location of the EBA is assigned.

The 32-bit address value entered for the EBA dictates the address at which the exception vector table resides. All LatticeMico32 exception vectors are located at a relative offset from the EBA. Table 7 summarizes the exceptions and their offsets relative to the EBA address. There are 32 bytes between the offsets, providing the ability to fit eight instructions per exception.

Table 7: Exception Offsets from EBA Address Bytes

Exception Type	Offset from EBA Address (Bytes)
Reset (power-up)	0
Breakpoint	32
Instruction bus error	64
Watchpoint	96
Data bus error	128
Divide by zero	160
Interrupt	192
System call	224

The value entered in the Processor Configuration dialog box, shown in Figure 135 on page 185, is the microprocessor power-up value for EBA. Once the microprocessor is up and running, you can later modify the EBA to allow the “relocation” of the exception vector table. This relocation enables you to deploy your code in non-volatile storage by setting the EBA to the non-volatile memory address. Once the microprocessor boots up, the code in the non-volatile memory can copy itself to volatile memory and change the EBA value to the volatile memory location to allow exceptions to be handled from volatile memory, improving responsiveness.

The assembly code excerpt shown in Figure 45 shows how to set the EBA at run time. It sets the EBA location to the _reset_handler address.

Figure 45: Setting the EBA at Run Time

```
mvhi        r1, hi(_reset_handler)
ori         r1, r1, lo(_reset_handler)
wcsr        EBA,r1
```

At power-up, the microprocessor fetches the first instruction from the location set in EBA. This location in the EBA is dictated at platform generation time, as noted earlier. Although the microprocessor supports 32 interrupts, the interrupt exception is generated.

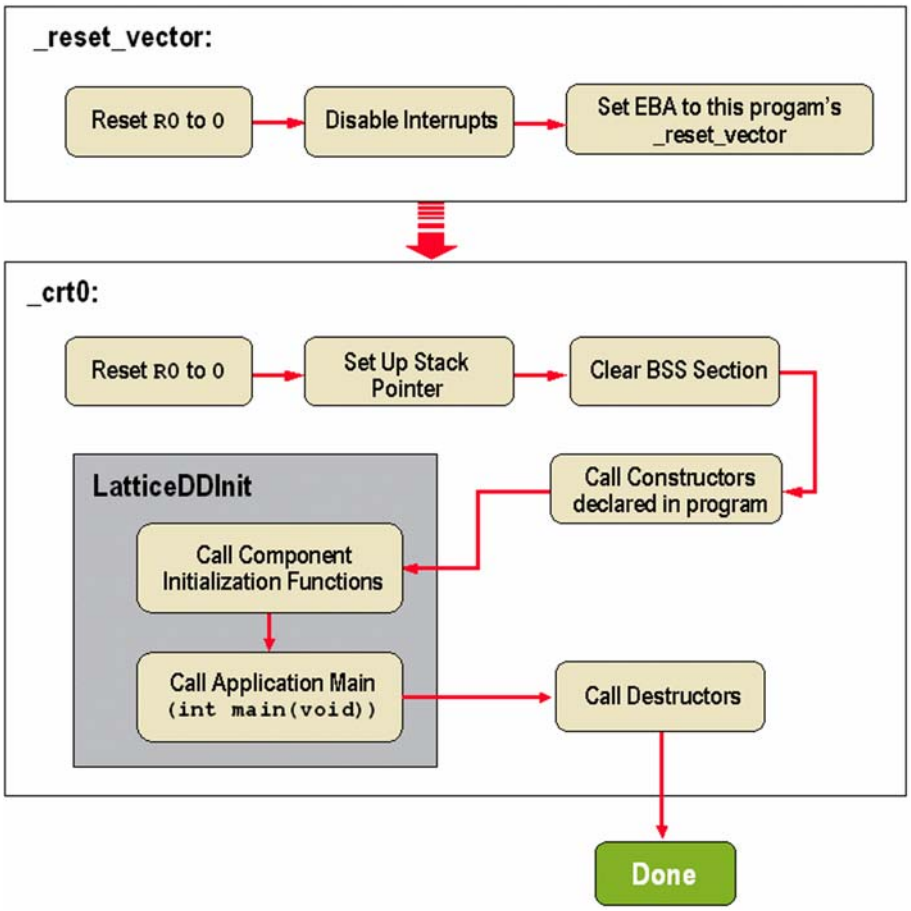
You can enable the microprocessor debug module as part of microprocessor configuration when generating the platform in the Mico System Builder (MSB). If you do so, the debug address that is assigned as part of platform address generation plays an important role in handling some of the exceptions listed in Table 7 on page 77. This debug address, called the debug exception base address, is known as the DEBA.

The DEBA represents the start of the debug exception table in the debug module that corresponds to the layout of the EBA exception table shown in Table 7 on page 77. The DEBA is used for vectoring the debug exceptions, in particular breakpoint and watchpoint exceptions, and the EBA is used for vectoring non-debug exceptions: the reset exception, divide-by-zero exception, rinstruction bus error exception, data bus error exception, interrupt exception, and system call exception.

Boot Code Sequence Flow

This section provides an overview of the boot sequence and its steps. Refer to the detailed illustration of sample boot code in “Boot Sequence and crt0ram.S” on page 64. Figure 46 illustrates the boot sequence.

Figure 46: Boot Sequence



The primary actions that the boot-up initialization code must perform are as follows:

- Flush the instruction and data caches as necessary.
- Set the R0 register to zero because the compiler expects it to always contain the value 0.
- Establish a valid EBA in the event that the code was copied from another location.

The boot code must also lay out the exception vector table. The default LatticeMico32 boot code is provided in the crt0ram.S file. This implementation vectors debug exceptions to the address contained in the DEBA register to allow the debug module to handle them. The boot code calls on the MicoISRHandler routine to handle interrupt exceptions. The MicoISRHandler routine is part of the LatticeMico32 interrupt management driver.

You can use the code contained in the crt0ram.S file as a starting point to write your own boot code. 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 during boot-up. The flow diagram in Figure 46 outlines the steps performed by the boot code. Refer to “Boot Sequence and crt0ram.S” on page 64 to view the sample boot code.

LatticeMico32 Microprocessor Usage

The LatticeMico platform is based on the LatticeMico32 microprocessor. To ease software development, the C/C++ SPE managed build includes microprocessor-specific drivers that provide access to microprocessor registers and manage the interrupt handling flow. This section lists the available microprocessor-specific functionality included as part of a managed C/C++ SPE build. For details on the LatticeMico32 architecture, refer to the *LatticeMico32 Processor Reference Manual*.

Data Types

The LatticeMico compiler tool chain is a GNU C/C++ compiler tool chain customized for the LatticeMico32 microprocessor. The data types that can be used as basic building blocks for programs are similar to those used in this tool chain. Table 8 lists well-known intrinsic data types that are supported in this development flow.

Table 8: Supported Data Types

Data Type	Bit Width
char	8
unsigned char	8
short int	16
unsigned short int	16
int	32
unsigned int	32
unsigned long long int	64

Byte Order

The byte order used for data operations by the LatticeMico32 microprocessor is big endian. For multi-byte objects, data is stored in memory with the most significant byte (MSB) first, that is, at the memory location with the lowest

address. The code excerpt shown in Figure 47 demonstrates the difference in behavior between a big-endian and a little-endian microprocessor from a software programmer's perspective in its comment lines.

Figure 47: Difference Between Little Endian and Bit Endian Microprocessor

```
unsigned char cValue;
unsigned int uiValue = 0x12345678;
unsigned int *puiValue = &uiValue;

/*
 * FOR LATTICEMICO32,
 * THE VALUE OF cValue AFTER EXECUTING THE
 * FOLLOWING INSRUCTION WILL BE
 * 0x12
 *
 * FOR A LITTLE ENDIAN PROCESSOR, SUCH AS AN
 * INTEL PROCESSOR, THE VALUE OF cValue AFTER
 * EXECUTING THE FOLLOWING INSTRUCTION WILL BE
 * 0x78
 *
 */
cValue = *((unsigned char *)puiValue);


/*
 * FOR LATTICEMICO32,THE VALUE OF cValue
 * AFTER EXECUTING EITHER OF THE TWO STATEMENTS
 * BELOW WILL BE
 * 0x78
 */
cValue =  *(((unsigned char *)puiValue)+3);
cValue = (unsigned char)(*(unsigned int *)puiValue);
```

Interrupt Management

The LatticeMico32 microprocessor accepts 32 external interrupt lines from external components. To facilitate handling interrupts and acknowledging them, the LatticeMico32 microprocessor device driver provides a framework for registering interrupt handlers and controlling interrupt generation.

As part of system boot-up, the driver disables all interrupts in the interrupt mask and waits for the application to register for an interrupt handler or enable interrupts. The interrupt management driver turns off any interrupt source (0 through 31) that does not have a corresponding registered interrupt handler (registered by either the user application or a driver).

The default interrupt handler provided as part of the LatticeMico32 interrupt management driver implements a high-priority-first scheme, where the component connected to interrupt line 0 of the LatticeMico32 microprocessor has the highest priority and the component connected to interrupt line 31 has the lowest priority. This priority is implemented in the default interrupt handler. Additional details on the exception/interrupt vector table are provided in "Other Exception Handlers" on page 68 and "Interrupt Exception" on page 67.

Nested interrupts are disabled as part of the default interrupt handler. You can provide your own interrupt handling scheme, which overrides the default implementation that could service a high-priority interrupt arriving while a lower-priority interrupt is being serviced.

Registering/Deregistering an Interrupt Handler

You can register for an interrupt handler from either your application or from a device driver. You must know the interrupt line of the microprocessor that your component of interest is connected to, from 0 through 31. The API shown in Figure 48 is used for registering an interrupt handler.

Figure 48: API Used to Register an Interrupt Handler

```
/*
 * Registers and de-registers interrupt-handler routine.
 * To register, pass a valid function pointer to the Callback parameter.
 * To deregister, pass 0 as the callback parameter.
 *
 * Arguments:
 *  unsigned int IntLevel: interrupt line number that your component is
 *       connected to (0 to 31).
 *  void *Context: pointer provided by user that will be passed to the
 *       interrupt-handler callback.
 *  ISRCallback Callback:   User-provided interrupt-handler routine.
 *
 * Return values:
 *       MICO_STATUS_OK if successful.
 */
mico_status MicoRegisterISR(unsigned int IntLevel, void *Context, ISRCallback Callback);
```

The specific interrupt that is being registered is enabled once the user-provided handler is registered. If the interrupt handler is unregistered, the interrupt will be disabled after completion of the function call.

The prototype for the callback is shown in Figure 49.

Figure 49: Callback Prototype

```
/* isr-callback typedef */
typedef void(*ISRCallback) (unsigned int intLevel, void *pUserPointer);
```

The callback's first argument is the interrupt line, and the second parameter is the pointer context provided at registration. The callback is called at the interrupt level, so the processing must be kept to a minimum to avoid interrupt responsiveness penalties.

Enabling a Specific Interrupt

You can enable a specific interrupt from 0 through 31, using the API provided in Figure 50. If you enable an interrupt that does not have a registered interrupt handler, the interrupt management software will disable that enabled interrupt if it receives an interrupt from that line.

Figure 50: API Used to Enable a Specific Interrupt

```
/*
 * Enables a specific interrupt
 *
 * Arguments:
 *  unsigned int Intlevel: interrupt 0 through 31 that needs to
 *       be enabled.
 *
 * Return values:
 *       MICO_STATUS_OK if successful.
 */
mico_status MicoEnableInterrupt(unsigned int IntLevel);
```

Disabling a Specific Interrupt

You can disable a specific interrupt from 0 through 31, using the API provided in Figure 51. Only the interrupt being disabled is disabled.

Figure 51: API Used to Disable a Specific Interrupt

```
/*
 * Disables a specific interrupt
 *
 * Arguments:
 *  unsigned int Intlevel: interrupt 0 through 31 that needs to
 *       be disabled.
 *
 * Return values:
 *       MICO_STATUS_OK if successful.
 */
mico_status MicoDisableInterrupt(unsigned int IntLevel);
```

Disabling All Interrupts

You can disable all interrupt sources (0 through 31) using the API listed in Figure 52. The function essentially masks out all the interrupts. Though the components may generate interrupts, the LatticeMico32 microprocessor effectively ignores them because its interrupt mask is set to all zeros. As a return parameter, the function returns a 32-bit value that must be passed to the MicoEnableInterrupts function to restore the interrupts to the state that they were in before this function was called.

Figure 52: API Used to Disable All Interrupts

```
/*
 * Disables all external component interrupts
 *
 * Arguments:
 *      None.
 *
 * Return values:
 *      unsigned int: 32-bit value that must be
 *          provided when re-enabling all interrupts
 *          for restoration to state that existed prior
 *          to calling this function.
 */
/* Disables all interrupts, returns mask */
unsigned int MicoDisableInterrupts(void);
```

Enabling All Interrupts

You can enable multiple interrupt sources simultaneously or restore the interrupt mask to its state before all interrupts were disabled, using the API shown in Figure 53.

Figure 53: API Used to Enable All Interrupts

```
/*
 * Enables selected interrupts from 0 through 31
 * as indicated by the unsigned int intrMask where
 * bit-0 corresponds to interrupt line 0 and
 * bit-31 corresponds to interrupt line 31.
 *
 * Arguments:
 *      None.
 *
 * Return values:
 *      unsigned int: 32-bit value that must be
 *          provided when re-enabling all interrupts
 *          for restoration to state that existed prior
 *          to calling this function.
 */
void MicoEnableInterrupts(unsigned int intrMask);
```

If you called the MicoDisableInterrupts function to disable all interrupts, the 32-bit returned value can be used as the argument to the function shown in the example code in Figure 54 to restore the interrupt-enable state.

Enabling/Disabling Interrupts

The code example shown in Figure 54, part of timer services, shows how to enable and disable interrupts.

Figure 54: Enabling and Disabling Interrupts

```
/*****************************************************************************
 * get cpu-ticks                                                             *
 *****************************************************************************/
void MicoGetCPUTicks(unsigned long long int *ticks)
{
    /*
     * We need to get a finer resolution than the system-tick and also
     * account for a possible roll-over just after we read the snaphot.
     * We're definitely not going to be exact on the dot but we'll be
     * pretty darn close; higher the clock-speed, less the error (in
     * seconds).
     */
    unsigned long long int cpuTicks = 0;
    unsigned int intrMask;
    unsigned int snapshot;

    if(s_MicoSystemTimer != 0) {

        /* disable interrupts and read the gross resolution tick-count */
        intrMask = MicoDisableInterrupts();
        cpuTicks = s_MicoCPUTicks;


        /* read the snapshot to get a finer tuning */
        snapshot = MicoTimerSnapshot(s_MicoSystemTimer);


        /*
         * Since the timer is an external peripheral over a bus, there may be
         * contention and we may not get the right reading.  So allow another
         * interrupt if it happened, to make our tick-reading as accurate
         * as possible.
         */
        MicoEnableInterrupt(s_MicoSystemTimer->intrLevel);


        /* we're done; decide what our reading should be */
        MicoDisableInterrupts();
        if(cpuTicks == s_MicoCPUTicks)
            cpuTicks += (s_MicoSysTicks - snapshot -1);
        else
            cpuTicks = s_MicoCPUTicks;


        /* restore interrupts */
        MicoEnableInterrupts(intrMask);
    }

    if(ticks != 0)
        *ticks = cpuTicks;

    return;
}
```

The code example in Figure 55 shows how to register an interrupt handler for a device using LatticeMico32 interrupt management.

Figure 55: Using Interrupt Management to Register Interrupt Handler

```
/* forward declaration of our device-specific interrupt
 * service-routine: This complies with the expected prototype
 *
 *  void(*ISRCallback)(unsigned int, void *);
 *
 */
static void myDeviceISR(unsigned int isrLevel, void *pData);


/*
 * This variable will be incremented each time our device
 * generates an interrupt
 */
volatile unsigned int iTotalInterrupts;


/*
 * The following piece of code demonstrates how to register
 * an interrupt handler
 */
void RegisterMyDeviceISR(void)
{
    /*
     * For the sake of this example, the device's interrupt
     * line is connected to the CPU's line 28.
     *
     * Also, we want the ISR to increment the iTotalInterrupts
     * variable each time the interrupt happens.
     */

    /* We will now register our device's interrupt-handler using
     * the call to MicoRegisterISR.  We will provide the following
     * arguments to the function call:
     * - interrupt-line on the CPU that our device is connected to (28)
     * - pointer to data that is specific to our code, i.e. pointer
     *   to iTotalInterrupts as our ISR must be able to access it.
     * - Pointer to our ISR callback routine (it MUST comply with
     *   the prototype mentioned).
     */
    MicoRegisterISR(28, (void *)&iTotalInterrupts, myDeviceISR);
}


/************************************************************
 * Interrupt-handler:                                       *
 * This interrupt-handler will be invoked when the device   *
 * that is connected to the registered interrupt line (0-31) *
 * is asserted by the device triggering an interrupt        *
 * exception on LatticeMico32.  The interrupt exception     *
 * invokes MicoISRHandler which in turn will call this      *
 * "callback" routine we've registered.                     *
 ************************************************************/
static void myDeviceISR(unsigned int isrLevel, void *pData)
{
    /* The arguments to this callback are:
     * - isrLevel: interrupt-level for which this ISR was
     *   registered as the handler
     * - void *pData: the pointer that we provided when
     *   registering this interrupt handler.
     */

    /* TODO: ADD CODE TO ACKNOWLEDGE THE DEVICE'S INTERRUPT
     *   SUCH AS READING A DEVICE REGISTER OR WRITING TO A
     *   DEVICE REGISTER.
     * Note: THe MicoISRHandler takes care of acknowledging the
     *   CPU's Interrupt Pending register so we don't have to
     *   worry about it in our ISR
     */
    /*
     * ... ISR HANDLER CODE ...
     */


    /*
     * Now increment the counter: we provided pointer to
     * iTotalInterrupts as argument when registering the interrupt
     * so we simply typecast the void* in this callback to our
     * datatype.
     */
    *((volatile unsigned int *)pData)++;
    return;
}
```

Cache Management

You can configure the LatticeMico32 microprocessor with or without the data cache or the instruction cache. The caches are write-through caches; that is, writing to a cached location is also translated as writing to the supporting memory. The cache implementation in LatticeMico is a simple implementation with the software supporting the ability to flush the caches, which invalidates the cache contents. Refer to the *LatticeMico32 Processor Reference Manual* for more details on cache-sizing parameters.

During the boot-up sequence, these caches are flushed to make sure there is no other instruction or data present in the cache. For example, in a typical situation, you would have multiple applications executing sequentially, such as a boot copier followed by the "main" application. It is far more common to flush the data cache, for example, reading status data from a flash device, which is a memory device mapped to a cached region.

Normally you do not want the peripherals that perform input and output operations to be in a cached region because it increases the execution speed for the rest of your instructions. Also, driver development becomes tedious if the input and output peripheral, such as a timer, is placed in a cached region. The Mico System Builder (MSB) places all non-memory peripherals in a non-cached region and all memory peripherals in a cached region. The drivers provided by MSB assume the peripherals are in a non-cached region.

The CPU does not monitor access to cached region locations performed by other masters such as DMA, so you are responsible for managing the cache, that is, invalidating the cache by flushing it in multi-master situations that may share address space.

Data Cache Flush Routine

You can flush the data cache using the API shown in Figure 56. There is no control to lock cache lines, and you cannot flush a cache selectively.

Figure 56: API Used to Flush Data Cache

```
/*
 * Flushes data cache
 */
void MicoFlushDataCache(void);
```

Instruction Cache Flush Routine

You can flush the instruction cache using the API shown in Figure 57. There is no control to lock cache lines, and you cannot flush a cache selectively.

Figure 57: API Used to Flush Instruction Cache

```
/*
 * Flushes instruction cache
 */
void MicoFlushInstrCache(void);
```

Sleep (Busy) Functions

To aid development, two functions, MicoSleepMicroSecs and MicoSleepMilliSecs, enable you to perform a “sleep” function. These sleep routines do not really put the microprocessor to sleep. The implementation involves a tight loop of instructions that aim to spend as much time as possible to that desired effect. These functions should be used only for approximate needs and not in situations where precision is required.

Note

These sleep functions are highly dependent on the memory controller latencies, as well as presence of instruction cache. These functions may be off significantly in the absence of the instruction cache or if the memory controller exhibits several cycles’ worth of latency.

Figure 58 shows the MicoSleepMilliSecs function.

Figure 58: MicoSleepMicroSecs and MicoSleepMilliSecs Functions

```
/*
 * Implements a tight-loop to achieve a "sleep" of the
 * desired time specified in microseconds.
 *
 * Note: the time spent in the tight loop is only an
 * approximation and is not very precise.
 */
void MicoSleepMicroSecs(unsigned int timeInMicroSecs);


/*
 * Implements a tight-loop to achieve a "sleep" of the
 * desired time specified in milliseconds.
 *
 * Note: the time spent in the tight loop is only an
 * approximation and is not very precise.
 */
void MicoSleepMilliSecs(unsigned int timeInMilliSecs);
```

Figure 59 demonstrates the usage of the MicoSleepMilliSecs function.

Figure 59: Usage of the MicoSleepMilliSecs Function

```
/* "sleep" i.e. wait for 2 seconds */
MicoSleepMilliSecs(2000);
```

Microprocessor Control Register Access

Some functions are provided for accessing some of the LatticeMico control or status registers. The C functions listed in Figure 60 are wrappers for assembly-level routines that can be used to write functions that operate on other control or status registers. You should not have to directly access the microprocessor's control or status registers.

Figure 60: C Functions That Control Register Access

```
/*********************************************
 * processor control/status registers access  *
 *********************************************/
/*
 * Writes a 32-bit value to the Interrupt-Enable register
 */
void MicoWriteIERegister(unsigned int ie);

/*
 * Writes a 32-bit value to the Interrupt-Mask register
 */
void MicoWriteIMRegister(unsigned int im);


/*
 * Writes a 32-bit value to the Interrupt-Pending register
 */
void MicoWriteIPRegister(unsigned int ip);

/*
 * Reads the Interrupt-Mask register
 */
unsigned int MicoReadIMRegister(void);

/*
 * Reads the interrupt-Enable register
 */
unsigned int MicoReadIERegister(void);

/*
 * Reads the Interrupt-Pending register
 */
unsigned int MicoReadIPRegister(void);
```

Macros

Some macros aid in the conversion of time units to microprocessor ticks. These macros are defined in the MicoMacros.h header file. They are shown in Figure 61.

Figure 61: Macros Used in Converting Time Units to Ticks

```
/*
 * MACROS FOR TIME CONVERSION
 */
#define MILLSECONDS_TO_TICKS(X_MS)
     (X_MS*(MICO32_CPU_CLOCK_MHZ/1000))

#define MICROSECONDS_TO_TICKS(X_MICROSECS)
     (MILLISECONDS_TO_TICKS(X_MICROSECS)/1000
```

The value of MICO32_CPU_CLOCK_MHZ is defined in the DDStructs.h file as part of the C/C++ SPE managed build.

Run-Time Services

This section refers to “services” in the run-time environment that are available to you as you program your microprocessor application code. Services refer to software abstractions that facilitate device functionality through their usage, making it unnecessary for you to know specific device information to carry out certain functions.

Device Lookup Service

As part of the Mico System Builder (MSB), components that are added to a platform definition must have a unique name. Each component in the LatticeMico managed build framework must have a <*component_name*>.xml file. As described earlier, the build process extracts this information, creates component-specific information structures, and fills in the values.

LatticeMico32 device drivers rely on this instance-specific component information for manipulating the component. This instance-specific component information allows a single device driver function to handle multiple instances of the same component.

From an application perspective, you must provide this instance-specific information to the device drivers. The LatticeMico lookup service allows easy access to this instance-specific information by looking up registered devices by name.

The component device driver registers the component instance with the LatticeMico lookup service as part of the component’s initialization routine, making it available for lookup before the start of your “main” routine.

The LatticeMico lookup service exposes intuitive API to find named devices or named services and provides component instance-specific information that you can use when manipulating a device or a service through the API made available by the relevant services or devices.

Using the Lookup Service

The lookup service makes API available to the user application to invoke. This section describes the usage of this API.

Finding a Device by Name The MicoGetDevice function shown in Figure 62 enables you to look up a device by name. The device name is case-sensitive. It returns a pointer to the component's instance-specific information structure.

Figure 62: MicoGetDevice Function Example

```
/*
 * Finds a device (that is registered with a registered
 * service)
 * Arguments:
 *  const char *Name: pointer to a character string
 *  representing device name (case-sensitive)
 * Returns:
 * void *: pointer to the looked-up device's instance-specific
       information. Will be 0 if no device with matching name
       is found.
*/
void *MicoGetDevice(const char *Name);
```

Since the components are selected and named at the time of platform generation, you should be aware of the component information structure type for the named device, and you are expected to typecast the returned pointer to an appropriate information structure type.

The Mico System Builder (MSB) software only allows you to generate a platform that contains unique names for all of its defined components. In turn, the managed build process also does not permit duplicate component names in a platform. The lookup service works on the assumption that component names in the platform are unique. Only those devices and services that follow the guidelines for developing device drivers are available for device lookup, as discussed in "Modifying Existing Device Drivers" on page 123.

The code example in Figure 63 illustrates how to use the MicoGetDevice function to find a GPIO named "LED" (case-sensitive) in the platform.

Iterating Through a List of Devices For a platform with multiple instances of components, each component instance is registered with the lookup service. As part of registration information, the device driver must provide a device type to which the component instance belongs. The device type enables your application to iterate through all available instances of a given service type or through all component instances that are available for lookup.

Figure 63: Using MicoGetDevice Function to Find a GPIO

```
#include "MicoGPIO.h"
#include "LookupSrevices.h"

int main(void)
{

     /* fetch LED GPIO by name: name is case-sensitive */
     MicoGPIOCtx_t *pLED;
     pLED = (MicoGPIOCtx_t *)MicoGetDevice("LED");
     if(pLED == 0) {
         /* platform does not contain a registered GPIO named
         "LED" */
         printf("failed to fetch GPIO (LED) instance\r\n"|;
         return(-1);
     }

     return(0);
}
```

The application must call the MicoGetFirstDev function, as shown in Figure 64.

Figure 64: Using MicoGetFirstDev Function to Iterate Through a List of Devices

```
/*
* Finds the first device (that is registered) of the specified
* type
* Arguments:
* const char *deviceType : points to named device type. If this
*        pointer is a null pointer, the first context of the
*        first device in the list of registered devices is
*        returned, irrespective of the type under which the
*        device is registered. If a non-null pointer, it must
*        point to a valid string (case-sensitive).
*
* DevFind_st *FindCtx: pointer to a valid allocation of
*        DevFind_st that will be referenced by MicGetFirstDev
*        for future invocations to MicGetNextDev
*
* Returns:
*
* void *: pointer to device context (is null if no matching
*         device is found).
*/
void *MicoGetFitrsdtDev(const char *deviceType, DevfindCtx_t
* FindCtx);
```

The first argument is a pointer to the device type name. This pointer can either be a specific named service type or a device type, or it can be a null pointer. If this pointer is null, the LatticeMico32 lookup service assumes the intent is to iterate through all the registered component instances, irrespective of the device types.

If this pointer is not null, LatticeMico32 attempts to find the first registered component instance of that device type. The second argument to this function is a pointer to a valid structure of type DevFindCtx_t. This structure is filled in by MicoGetFirstDev and should not be modified by the application. This function parameter can be used in subsequent calls to MicoGetNextDev to retrieve the next component instance of the desired type. The return value of this function is a void pointer to the device's instance-specific component information structure. This pointer is null (zero) if no matching registered device is found.

On a successful completion call to MicoGetFirstDev—that is, the returned pointer is not null—the application can then call the MicoGetNextDev function, as shown in Figure 65, to retrieve a pointer to the next matching device's instance-specific component information structure. This function takes a single parameter, a pointer to the DevFindCtx_t structure type that was provided to the MicoGetFirstDev function call. The values of the structure referenced by this pointer must not be modified by the application. If the LatticeMico lookup service is successful in finding the next matching registered device, it returns a pointer to the matching device's instance-specific component information structure.

Figure 65: Using MicoGetFirstDev Function to Find Pointer to Instance-Specific Component Information Structure

```
/*
* Finds the next registered device that matches the find
criteria provided in the prior MicGetFirstDev invocation
*
* Arguments:
*   DevFind_st *FindCtx: pointer to a valid allocation of
*           DevFind_st that was provided to MicoGetFirstDev
*           invocation. Caller must not modify the structure
*           referenced by this pointer.
*
* Returns:
*
*   void *: pointer to device context (is null if no matching
*           device is found).
*/
void *MicoGetNextDev(DevFindCts_t *FindCtx);
```

The CFIFlashPrgrmr.c flash programming software template demonstrates usage of the functions previously referenced for iterating through a list of registered devices of a specific type (for example, the CFI flash device type in the example). This flash programming software template is located in the following path:

<*install path*>\micosystem\utilities\templates\CFIFlashProgrammer

List of Device Types

LatticeMico Mico System Builder (MSB) uses the following device types:

- CFIFlashDevice: LatticeMico CFI flash component type

- GPIODevice: LatticeMico GPIO component type
- UARTDevice: LatticeMico UART component type
- TimerDevice: LatticeMico timer component type
- SPIDevice: LatticeMico SPI component type
- DMADevice: LatticeMico DMA component type

The component device drivers and their example usage are provided in the respective component data sheets available through the MSB user interface. The section “Making Devices Available to Lookup Service” on page 130 explains the steps required of the driver to make a component instance available to the LatticeMico lookup service.

See also “Accessing Component Help and Data Sheets” in the *LatticeMico System Hardware User Guide*.

LatticeMico System Timer Services

In addition to making the timer component available to the lookup service, the LatticeMico timer software also enables you to register a LatticeMico timer instance as the system timer and register for a callback on a system tick.

Registering System Timer

Before you can use the system timer facility, you must register a LatticeMico timer instance as the system timer. Use the API shown in Figure 66 to register a LatticeMico timer as the system timer.

Figure 66: Registering a LatticeMico Timer Instance as the System Timer

```
/*
 * Registers system-timer if one isn't already registered.  Once registered,
 * you cannot register some other timer as a the system timer.  You can
 * always stop the timer using LatticeMico32 device-driver routine
 * and bring the system-timer to a halt.
 *  Arguments:
 *      MicoTimerCtx_t *ctx: LatticeMico32 Timer instance information
 *      structure (context) that should be used as the system timer.
 *      unsigned int TickInMS: system-tick value in milli-seconds.
 *  Return Value:
 *      MicoTimerCtx_t *: LatticeMico32 timer instance information structure
 *      of the timer being used as the system-timer.  If there is no system
 *      timer, it returns 0.
 */
MicoTimerCtx_t* RegisterSystemTimer(MicoTimerCtx_t *ctx, unsigned int TickInMS);
```

Once you register a system timer, you cannot deregister this timer to register another system timer. Also, you must not use the system timer for any other purpose. The service programs the system timer in a continuous mode with interrupts enabled and handles the timer expiration interrupt. The system timer helps maintain a crude 64-bit system time, which measures the elapsed ticks since registering the system timer. You must ensure that the timer width (bits) is appropriate to hold the suitable TickInMS value.

Registering a System Tick Callback

Once you have registered a system timer, you can register for a callback that is called as part of the system timer's interrupt service routine (ISR). Use the API shown in Figure 67 to register a callback on a system tick expiration event.

Figure 67: Registering a Callback on a System Tick

```
/*
 * Registers system tick periodic activity
 *  Arguments:
 *      MicoSysTimerActivity_t: activity function pointer as described by the
 *      prototype:
 *
 *        void (* MicoSysTimerActivity_t) (void *);
 *
 *      void *ctx: pointer to user data that will be passed back on system tick.
 */
void MicoRegisterActivity(MicoSysTimerActivity_t activity, void *ctx);
```

The callback is part of the system timer ISR, so the activities performed as part of the callback must be kept to a minimum. For reduced indirection, you can directly manipulate the timer instance by using the LatticeMico timer device driver routines instead of using the system timer.

Retrieving CPU Ticks

The system timer facility keeps track of the elapsed CPU ticks from the time of registration of the system timer. The maintained CPU tick value is a 64-bit value but is somewhat skewed because the service should also account for a timer count rollover. The timer width is limited to 32 bits. The tick count is therefore imprecise, but it is accurate to some degree and useful for most purposes. For a highly accurate 64-bit count, you can implement an integral 64-bit high-resolution counter WISHBONE peripheral that is not part of the Mico System Builder (MSB) distribution.

Use the API shown in Figure 68 to fetch CPU ticks.

Figure 68: Retrieving CPU Ticks

```
*/
 * Retrieves 64-bit tick count, if a system timer is
 * registered. Else, it returns 0.
 */
void MicoGetCPUTicks(unsigned long long int *ticks);
```

System Timer Usage

The code example shown in Figure 69 shows how to use the system timer facilities as part of LatticeMico System timer services.

Figure 69: System Timer Usage Sample Code

```
#include<stdio.h>
#include "MicoUtils.h"
#include "DDStructs.h"
#include "MicoMacros.h"
#include "LookupServices.h"
#include "MicoTimerService.h"

/************************************
 * System tick handler              *
 ************************************/
void system_tick_handler(void *data)
{
    unsigned int *pInt = (unsigned int *)data;
    (*pInt)++;
    return;
}

/************************************
 * main program                     *
 ************************************/
int main()
{
    MicoTimerCtx_t *pTestTimer;
    unsigned long long int begin, end;
    unsigned int totalTicks = 0;

    /* fetch timer-context by name. */
    pTestTimer = (MicoTimerCtx_t *)MicoGetDevice("timer");

    /* if the device could not be found, don't perform the tests */
    if(pTestTimer == (MicoTimerCtx_t *)0){
        printf("failed to find timer: %s\n", "timer");
        return(1);
    }

    /* register system-timer for 100 millisecond*/
    RegisterSystemTimer(pTestTimer, 100);
    MicoRegisterActivity(system_tick_handler, &totalTicks);
    MicoGetCPUTicks(&begin);
    MicoSleepMilliSecs(2000);
    MicoGetCPUTicks(&end);
    end = end - begin;


    /* stop the system timer */
    MicoTimerStop(pTestTimer);

    /* dump information */
    printf("Software sleep of %d milliseconds equals %d ticks\n",
            2000,
            MILLISECONDS_TO_TICKS(2000));

    printf("Ticks reported by system-timer: %d\n", (unsigned int)end);
    printf("system interrupts: %d\n", totalTicks);
}
```

LatticeMico File Service

The managed build software development environment provides Newlib C standard C library support (libc.a) that is made available at the managed build application link step. The implementation of standard C file operations, such as printf, scanf, fopen, fprintf, fgets, fwrite, and fread, is provided through the Newlib C library.

The LatticeMico File Service provides endpoint connectivity to these Newlib C file operation routines, thereby allowing the flexibility to add devices that can be used for such file operations. This section describes the devices that support such file operations and also describes the internal operations of the file services framework. The next section describes how you can add your own file operations-capable device to the LatticeMico File Services.

LatticeMico Devices Supporting File Operations

LatticeMico software supports two devices capable of limited file operations: the LatticeMico32 microprocessor's JTAG UART and the LatticeMico UART. Other devices and other capabilities can also be included, as discussed in subsequent sections.

LatticeMico32 Microprocessor JTAG UART File Device

The LatticeMico32 microprocessor software support includes support for file operations by way of the JTAG UART microprocessor and the microprocessor debug module.

This software connects with the microprocessor's debug module to communicate with the GNU GDB debugger (GDB) running on a host or development computer through the microprocessor's JTAG UART. This connection allows the programs running on the LatticeMico32 microprocessor to access the file system on the computer hosting the LatticeMico Eclipse-based GDB debugger. The microprocessor software support uses the microprocessor instance name, as declared in the platform when registering itself as a file device.

Note

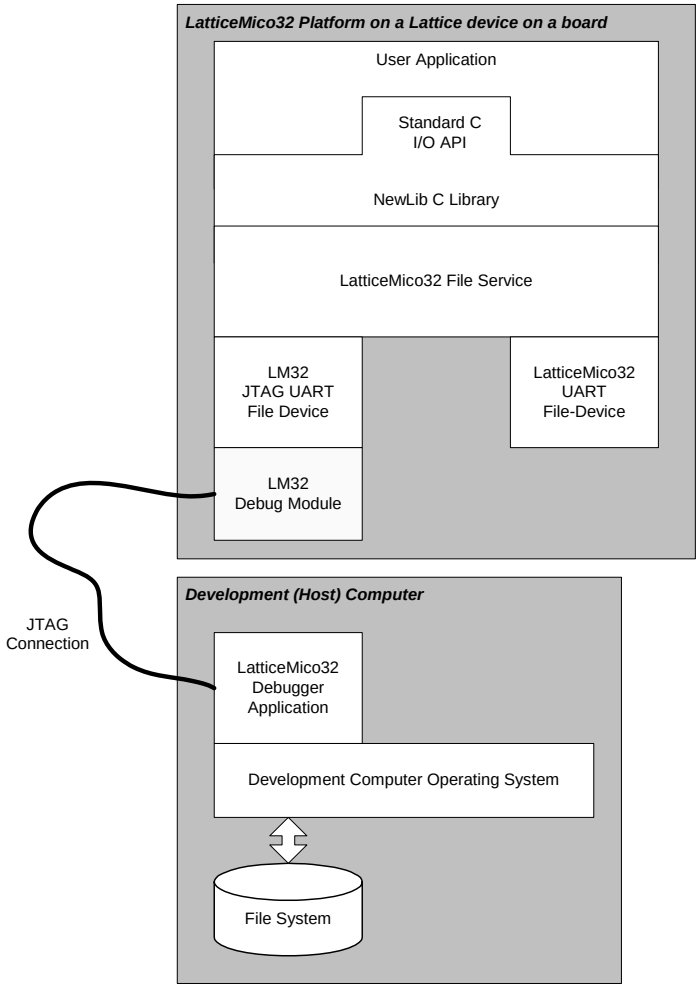
This support is available only if the debug module is included with the microprocessor and only if the LatticeMico Debugger is running on a remote computer and is connected to LatticeMico32 microprocessor debug module.

If the application code relies on the LatticeMico32 microprocessor file device for file operations, the Debugger must be running on the host and must be actively connected to the microprocessor.

- If the Debugger is disconnected, the application will appear to have hung when it performs file operations to the LatticeMico32 microprocessor file device as the software expects a Debugger to communicate with it.
- If the platform is modified so that the microprocessor does not contain the debug module, file operations relying on LatticeMico32 microprocessor file device will fail.

Figure 70 shows how the connection and software components are laid out.

Figure 70: Layout of Connection/Software Components



Maximum Simultaneously Opened Files The LatticeMico32 JTAG UART file support must maintain a mapping between the development computer's file ID and the local LatticeMico32 file descriptor. To avoid dynamic memory allocation, the space required for this map is allocated at compile time. The example code shown in Figure 71, located in LatticeMicoUart.c, shows the default value for the macro.

Figure 71: Default Macro Value

```
/* declare MICO_GDB_MAX_FILES if it is not already done */
#ifndef MICO_GDB_MAX_FILES
#define MICO_GDB_MAX_FILES (5)
#else
/* make sure there is space for at least 3 files */
#if MICO_GDB_MAX_FILES < 3
#define MICO_GDB_MAX_FILES (3)
#endif
#endif
```

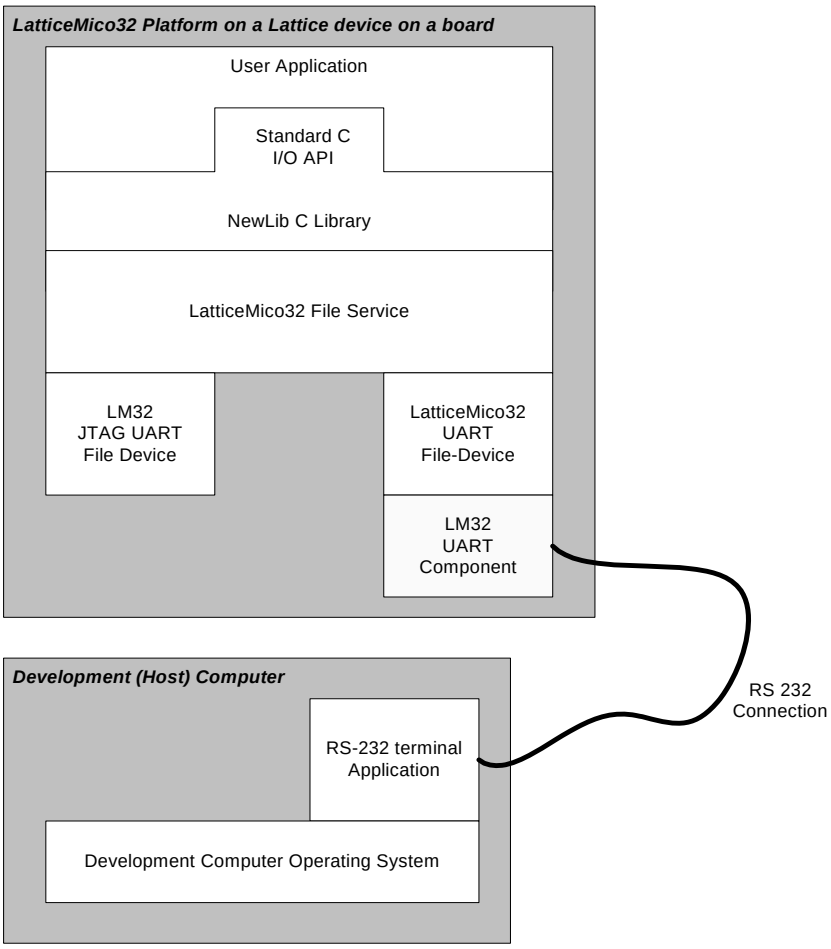
You can override this default macro value by defining it through the project's C/C++ SPE build properties (LatticeMico32 compiler preprocessor definitions). This limit is independent of the maximum file limit imposed by the LatticeMico File Service, as described later in this document.

Disabling a LatticeMico32 JTAG UART File Device To reduce the code size, you may want to exclude the JTAG UART file device code in the final executable. To do so, you must define the _MICOCPU_FILESUPPORT_DISABLED_ macro, which disables the code at compilation time. Also, this support is automatically disabled if the debug module does not reside in the microprocessor configuration.

LatticeMico UART Component The LatticeMico UART component is available to the LatticeMico File Service through the UART software services implementation. This UART file operations support is limited to console input/output, even though it can be treated as a file device through the standard C library functions such as fopen, fprintf, and so forth. The UART file service ignores the file name in an fopen function call.

Figure 72 shows how the software components and physical components are laid out for the UART file operations usage.

Figure 72: Layout of Software Components for UART File Operations



Disabling a LatticeMico UART File Device To reduce the code size, you may want to exclude the LatticeMico UART file device code in the final executable. To do so, you must define the _MICOUART_FILESUPPORT_DISABLED_ macro, which disables the associated LatticeMico UART file device code at compilation time.

Usage and File Name/Device Name Conventions

The conventions described in this section apply to Newlib C file operation APIs, such as fopen and fprintf. These file operation functions, as you would expect, perform operations on a file.

From a user perspective, the file is a named device; that is, it has a name. The syntax for the fopen standard C function is as follows:

```
FILE *fopen(const char *filename, const char *mode);
```

This API expects a file name (or *filename* parameter). Typically the file name is the name of a file associated on a disk drive. For the LatticeMico File Services, *filename* has two parts:

- Name of the file to open
- Name of the device on which to open the file, such as the CPU instance name for opening a file using the LatticeMico32 JTAG UART file support or the UART instance name for opening a file using a LatticeMico UART instance

The name of the device is optional. If it is not provided, the LatticeMico File Service passes on the fopen request to the default file device. If the name of the device is specified, the LatticeMico File Service attempts to identify a device that has a matching device name and then passes on the file name to that device's file support routine for opening that file.

The LatticeMico File Service adheres to the following conventions:

- Devices: When the file service accesses devices, the device name must be preceded by two backslashes (\\). For example, when the file service opens a uart_0 UART file device, the device must be addressed as \\uart_0 or \\uart_0\. In C/C++ code, since a single backslash (\) represents an escape character, it translates to \\\\uart_0 or \\\\uart_0\\.
- Files: When the file service opens files on a specific device, the device name must include the file name. For example, when the file service accesses file_0 on device uart_0, the file must be addressed as \\uart_0\file_0. When the file service opens files on the default file device, it can simply address the file by its name rather than including the default file device's name in the string.

The example code shown in Figure 73 illustrates the file name and device name usage. The example assumes that the LatticeMico32 JTAG UART file support is the default file device and that there is an instance of LatticeMico UART named "uart" in the platform, along with a LatticeMico32 microprocessor instance named LM32.

Figure 73: File Name/Device Name Usage Example

```
/*
 * Open the UART file-device for file-operations
 * The uart-instance in our platform is named "uart"
 * (without the quotes)
 */
fptr = fopen("\\\\uart\\", "wt");
if(fptr == 0){
    printf("\r\n failed to open \"uart\" device\r\n");
    return(-1);
}

fprintf(fptr, "This string goes to the uart");
fflush(fptr);
fclose(fptr);


/* open the file cpu.txt on the host computer for
 * file operations.  Explicitly specify that this file
 * belongs to the LatticeMico32 processor instance (named LM32)
 */
fptr = fopen("\\\\LM32\\cpu.txt", "wt");
i = (unsigned int)fprintf(fptr, "this is a valid test\r\n");
fclose(fptr);


/*
 * we want to open cpu.txt on the host development computer.
 * We have LatticeMico32 processor as the default file-device and
 * so we don't need to specify the device-name.
 */
fptr = fopen("cpu.txt", "wb");
i = fwrite(cBuf, 1, 5, fptr);
fclose(fptr);


/*
 * Re-open the just created file (on the host development computer)
 * for file-operations.  Again, we don't specify the file-device
 * as LatticeMico32 processor is set as the default file-device
 */
fptr = fopen("cpu.txt", "rb");
i = fread(cBuf, 1, 5, fptr);
fclose(fptr);
```

Setting the Default File Device

If the debug module is included and the LatticeMico32 CPU file service is enabled, the JTAG UART registers itself as the default file device with the LatticeMico File Service. It also registers itself under the microprocessor's instance name (for example, "LM32") with the file services. Subsequent file open requests that do not specify a device name are passed on to the LatticeMico32 JTAG UART file device software.

You can modify the default file device at run time, as demonstrated by the example code shown in Figure 74.

Figure 74: Modifying the Default File Device

```
/* Set the UART instance "uart" as the default file device */
if(MicoFileSetDefFileDevice("uart") != 0){
    printf("failed to set uart as the default file device\n");
}
```

If this function call returns without any error—that is, if it returns a value of zero—a subsequent file-open operation where the device name is absent is passed to this newly registered device instead of the previous default device.

The API listed shown in Figure 75 is declared in MicoFileDevices.h.

Figure 75: Setting the Default File Device

```
/*
 ---------------------------------------------------------------------
 - INCLUDE: MicoFileDevices.h                                        -
 -                                                                   -
 -   This function sets the default file-device.  It returns 0      -
 -   if successful.                                                  -
 -                                                                   -
 ---------------------------------------------------------------------
 -  PARAMETERS:                                                      -
 -   const char *devicename                                          -
 -     Name of the device to set as default file-device.             -
 -     NOTE: DO NOT USE \\ prefix to the file-device or the trailing \ -
 -           in the name. e.g. DO NOT USE "\\\\uart" or "\\\\uart\\"   -
 -           but use "uart" if your device is named "uart"           -
 -                                                                   -
 ---------------------------------------------------------------------
 -  RETURN VALUES:                                                   -
 -  0 => successful                                                  -
 -  MICO_FILE_ERR_DEVICE_NOT_FOUND                                   -
 -      Could not find a device with matching name                   -
 ---------------------------------------------------------------------
*/
int MicoFileSetDefFileDevice(const char *deviceName);
```

As illustrated in Figure 75, the device name convention that is used with fopen does not apply to this API.

File Name/Device Name Length

The LatticeMico File Service's file-open implementation must extract device and file name information. For this, it allocates stack space instead of dynamically allocated memory. To contain the stack space allocated by this function, which is freed once the function returns, the LatticeMico File Service imposes restrictions on file name and device name length.

If no prior definition exists, the following two macros, defined in MicoFileDevices.h, govern the maximum file name length and device name length.

Figure 76: Macros Governing Maximum File and Device Name Length

```
#define MICO_FILE_DEVICES_MAX_DEV_NAME_LEN (12)
#define MICO_FILE_DEVICES_MAX_FILE_NAME_LEN (32)
```

It is possible for a file device to have a longer name than the imposed limit; however, the LatticeMico File Service uses the maximum-length parameter for comparison when identifying matching named devices. The file name

restriction applies to the file device's open parameter; the file name provided in the fopen API is truncated to the maximum file name length when passing to the file device by the LatticeMico File Service.

By default, if these macros are not defined, the file name length is set to 32 bytes, including the null terminator, and the maximum device name length is set to 12, including the null terminator. The minimum length for device names is restricted to 12 bytes, including the null terminator, and the minimum length for file names is restricted to 13 characters, including the null terminator. You can override the values for these macros by defining these macros through the project's C/C++ SPE build properties, which defines the LatticeMico32 compiler's preprocessor options. See "Setting Project Properties" on page 26 for details on changing these build properties.

Standard I/O Device

Standard input, output, and errors must be directed to a registered file device. You can establish a standard I/O device in two ways:

- Platform tab in the Properties dialog box in C/C++ SPE – See Figure 14 on page 29. This tab lists the available devices that you can configure to serve as standard I/O device for any of the standard streams (in, out, and error).
- Run-time selection – You can set a registered file device to handle any of the standard streams by using the API in Figure 77. It is declared in MicoFileDevices.h.

Figure 77: API Used to Set a Registered File Device

```
/*
 ---------------------------------------------------------------------
 -                                                                   -
 -   This function redirects desired standard stream to the appro.   -
 -   priate selected device                                          -
 -                                                                   -
 ---------------------------------------------------------------------
 - ARGUMENTS:                                                        -
 -   int StreamId                                                    -
 -     0 => standard input                                           -
 -     1 => standard output                                          -
 -     2 => standard error                                           -
 -                                                                   -
 -  const char *deviceName                                           -
 -     name of the registered file-device to serve the standard      -
 -     stream.                                                       -
 ---------------------------------------------------------------------
 -  RETURN VALUES:                                                   -
 -    0 => successfully set device to handle the std stream          -
 -                                                                   -
 -    MICO_FILE_ERR_DEVICE_NOT_FOUND                                 -
 -         could not find a matching named device                    -
 -                                                                   -
 ---------------------------------------------------------------------
*/
int MicoFileRedirIO(int StreamId, const char *deviceName);
```

This API allows setting any registered name device to handle any or all of the standard streams.

You can specify that your named device handle any of the standard streams by including the piece of code shown in Figure 78 in your user-application source file, which defines the constants as listed. These constants are used by the LatticeMico File Services to identify devices at device registration time that should be set to handle standard I/O streams.

By specifying these constants in your user-application source file, you override the constants defined in the dynamically generated MicoStdStreams.c file. The constants in MicoStdStreams.c are generated by the managed build on the basis of settings chosen in the Project Properties dialog box accessible in the C/C++ perspective.

In the example code shown in Figure 78, a UART device named “uart” is chosen to handle the standard I/O streams. You can replace “uart” with the name of your registered file device.

Figure 78: UART Device Handling Standard I/O Streams

```
#ifdef __cplusplus
extern "C" {
#endif

const char* MICO_STDIN_DEV_NAME = "uart";
const char* MICO_STDOUT_DEV_NAME = "uart";
const char* MICO_STDERR_DEV_NAME = "uart";


#ifdef __cplusplus
}
#endif
```

As illustrated in Figure 78, the device name convention that is used with fopen does not apply to this API.

CFI Flash Device Service

The common flash interface (CFI) is an industry standard that allows software the flexibility of “adapting” to different CFI-compliant flash devices by storing the device characteristics on the flash device itself. The software uses parameters such as erase regions and timeout values stored on the flash device to perform modifications like erasing and writing.

Additionally, CFI flash devices manufactured by different manufacturers may support different erasing and programming algorithms, such as the Intel basic command set and the AMD command set. This information is also retrieved from the CFI parameters stored in the flash device. The CFI information is stored as tables in the flash device and can be accessed by performing reads from fixed device offsets, as dictated by the CFI specification.

While the CFI specification lists the offsets in terms of device offsets, the translation of these device offsets to memory addresses accessible by the microprocessor depends on two combined factors:

- Operating mode of the flash (8-bit, 16-bit, or 32-bit modes)
- Layout of the flash components (for example, two 16-bit flash parts operating in 16-bit modes)

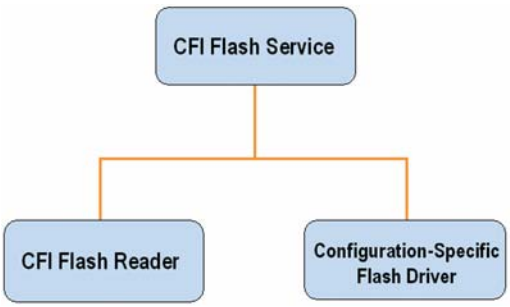
The combination of the operating mode or layout and the command set used by the flash device is called a flash configuration in discussions of the LatticeMico32 CFI flash device service.

The LatticeMico32 CFI flash device service hides such device-specific details from your application, which makes your application more portable across different CFI-compliant flash devices, as well as configuration. The service provides API for writing and erasing flash data, as well as for obtaining flash geometry information, for example, sector sizes and sector addresses.

Structure of the CFI Flash Device Service

The LatticeMico32 CFI flash device service comprises three main functional pieces, as shown in Figure 79.

Figure 79: CFI Flash Device Service Structure



These three pieces of software are the following:

- CFI flash service, which is the main software module that you interact with to perform flash manipulation operations. The LatticeMico32 flash device driver is part of this group of software.
- Configuration-specific flash driver, which implements the main functionality of the common flash operations, that is, erase and write operations. For each configuration, that is, the orientation of the flash devices and the command set used, there must be a configuration-specific flash driver. Currently the Mico System Builder (MSB) supports one configuration, a 2 x 16-bit CFI-compliant flash device operating in 16-bit mode using the AMD command set. The supported driver is documented in the LatticeMico flash component documentation.
- CFI flash reader, which is called by the CFI flash service as part of the parallel flash driver initialization routine. The CFI flash reader is

responsible for identifying the flash configuration, that is, the number of flash devices operating in parallel and the mode of operation using heuristics. From this information, it computes the addresses that must be generated for reading the flash device CFI information.

This CFI information is included in the flash device as part of the CFI specifications. The CFI information includes the CFI command set used by the flash part, in addition to other programming information, such as device timeouts for write and erase operations. The CFI flash service uses the flash configuration, that is, parallel devices and mode of operation such as 8-, 16-, 32-bit, along with the programming command set (for example, Intel and AMD) to select the appropriate flash configuration driver.

The LatticeMico CFI flash reader currently supports identifying two 16-bit flash devices operating in 16-bit mode. The current LatticeMico software support exists only for this configuration and AMD standard command set flash devices.

The CFI flash device service attempts to be “hardware-neutral,” so it does not use interrupt-driven programming, as some Intel flash devices do. The flash service does not support sector protection features.

Using the CFI Flash Service

The CFI flash service implements basic flash operation primitives, such as erasing a device and erasing a sector or programming data to the flash device. For programming data, it does not perform a sector read, superimpose, or program because there may be memory constraints. If you want to write at non-sector boundary offsets, you must read the affected sector, superimpose the overlapping data, and use the primitive to erase or program the data. The primitives also enable you to access information on sector regions for sector read facilitation.

Program Data The LatticeMico32CFIFlashProgramData function, shown in Figure 80, enables you to program a chunk of data to the flash component, if supported by the flash configuration driver. If the flash configuration driver does not support the functionality, an appropriate error is returned, as shown in this section.

Figure 80: Programming Data to the LatticeMico Flash Component

```
/*
 *  Programs flash data (erases and then writes affected sectors)
 *
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *      unsigned int ByteOffset: offset within the flash region where
 *          to program the data.
 *      unsigned char *pData: pointer to data to program.
 *      unsigned int bytes: amount of data (in bytes) to program.
 *
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 */
unsigned int LatticeMico32CFIFlashProgramData(CFIFlashDevCtx_t *ctx,
        unsigned int ByteOffset, unsigned char *pData, unsigned int bytes);
```

This function erases the affected sectors and programs the new data. It does not save data in those sectors that are only partially affected. These sectors are usually the first sector or the last sector for program operations that are not on sector boundaries. The function does not save data in these sectors because the driver would otherwise need a minimum of one sector's worth of memory storage, which can range from a few bytes to many kilobytes. For partial sector programming operation, you must first read the affected sector, impose the new data on the read data, and use this data in the function call to program data.

The configuration driver may impose restrictions on the offset, the amount of bytes to program, or both. For example, a dual 16-bit flash configuration may require the byte offset to be aligned to a 32-bit word boundary and that the amount of data be a multiple of four bytes.

Erase LatticeMico Flash Component The LatticeMico32CFIFlashEraseDevice function, shown in Figure 81, enables you to erase the entire flash region, if the flash configuration is identified and it supports erasing the entire flash region.

Figure 81: Erasing the LatticeMico Flash Component

```
/*
 * Erases entire device.
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *  Return Value:
 *      0 if successful
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 */
unsigned int LatticeMico32CFIFlashEraseDevice(CFIFlashDevCtx_t *ctx);
```

Erase LatticeMico32 Flash Sector The LatticeMico32CFIFlashEraseBlock function, shown in Figure 82, enables you to erase the sectors containing the provided offset.

Figure 82: Erasing the Sectors Containing an Offset

```
/*
 * Erases a block containing the supplied offset (byte-offset)
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *      unsigned int ByteOffset: offset in terms of byte
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 */
unsigned int LatticeMico32CFIFlashEraseBlock(CFIFlashDevCtx_t *, unsigned int ByteOffset);
```

Write Data The functions shown in Figure 83 enable you to write data in well-known sizes to the supplied byte offset in the flash region. Depending on the flash configuration, not all functions may be supported and will return the appropriate error value. These write functions assume that the affected

sectors have been erased before calling them, unlike the LatticeMico32CFIFlashProgramData function call described earlier, which erases the affected sectors and writes the data.

Figure 83: Writing Data to LatticeMico Flash Component

```
/*
 * Programs data to the flash device; assumes that the blocks to
 * which data's being written is already erased.
 *
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 *
 */
unsigned int LatticeMico32CFIFlashWrite32(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned int Data);
unsigned int LatticeMico32CFIFlashWrite16(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned short int Data);
unsigned int LatticeMico32CFIFlashWrite8(CFIFlashDevCtx_t *ctx, unsigned int ByteOffset, unsigned char Data);
```

For example, two 16-bit flash devices operating in 16-bit mode cannot support an 8-bit write primitive. To achieve an 8-bit write for such a configuration, the driver would have to read the affected 32-bit data that would contain just the 8-bit data and then would have to make a copy of the entire sector, superimpose the 8-bit data, and then write the entire sector back.

So not all configurations support the aforementioned data sizes. The configuration driver may impose restrictions on the byte offset alignment; for example, a 32-bit write may expect the byte offset to be aligned on a 32-bit boundary.

Write Block of Data The LatticeMico32CFIFlashWrite function, shown in Figure 84, enables you to write a block of data to a flash component. The configuration driver may or may not support this functionality, which results in an appropriate error code being returned. Also, the configuration driver may impose restrictions on the amount of bytes to program and the offset (that is, the byte offset) of where to program.

For example, a two 16-bit flash device region may require the amount of data to be a multiple of four bytes. Also, it may impose a restriction that the byte offset be aligned on a 32-bit boundary.

Figure 84: Writing a Block of Data to the LatticeMico Flash Component

```
/*
 * Programs a block of data with Bytes worth of bytes at the
 * provided byte offset.
 *
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *      unsigned int ByteOffset: byte-offset within the flash region
 *          where to program data.
 *      unsigned char *Data: pointer to the blcok of bytes to program.
 *      unsigned int Bytes: Number of bytes to program.
 *
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 *
 */
unsigned int LatticeMico32CFIFlashWrite(CFIFlashDevCtx_t *ctx,
                                        unsigned int ByteOffset,
                                        unsigned char *Data,
                                        unsigned int Bytes);
```

Obtaining Sector Information The LatticeMico32CFIFlashSectorInfo function, shown in Figure 85, enables you to obtain sector information. You can use this function to identify a sector and read its contents.

Figure 85: Obtaining Sector Information

```
/*
 *  Fetches the following sector information:
 *    - starting offset of the sector
 *    - length of the sector in bytes.
 *
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *      unsigned int ByteOffset: used to obtain info on sector
 *          containing this byteoffset.
 *      unsigned int *SAddr: pointer to an unsigned int that will contain
 *          the starting address of the sector containing the ByteOffset, on
 *          function-return.
 *      unsigned int *ByteLen: pointer to an unsigned int that will contain
 *          the length of the sector (in bytes) containing the ByteOffset, on
 *          function-return.
 *
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 */
unsigned int LatticeMico32CFIFlashSectorInfo(CFIFlashDevCtx_t *, unsigned int ByteOffset,
              unsigned int *SAddr, unsigned int *ByteLen);
```

Flash Component Reset Some flash devices have a soft reset command that is used especially if there is an error during a programming operation or erase operation. You can use the LatticeMico32CFIFlashReset function, shown in Figure 86, to reset a flash component, if it is supported by the flash configuration driver.

Figure 86: Resetting the LatticeMico Flash Component

```
/*
 *  Issues a flash-reset command.
 *
 *  Arguments:
 *      CFIFlashDevCtx_t *ctx: LatticeMico32 pointer to flash-device
 *          instance-specific information.
 *
 *  Return Value:
 *      0 if successful else encountered an error.
 *      Wellknown errors:
 *      MICOFLASH_ERR_UNIDENTIFIED_CFG: if configuration is not
 *          supported.
 *      MICOFLASH_ERR_UNDEFINED_FUNCTION: operation not supported
 *          for the identified configuration.
 *      MICOCFIFLASH_ERR_INVALID_ARGUMENT: context pointer is null.
 */
unsigned int LatticeMico32CFIFlashReset(CFIFlashDevCtx_t *ctx);
```

CFI Flash Service Usage

The CFI flash programmer provides an example of using the CFI flash service. This application template is located under the following folder in the file path:

`<install_dir>\micosystem\utilities\templates\CFIFlashProgrammer`

Enhancing CFI Flash Service

For the current release of LatticeMico Mico System Builder (MSB), the only supported flash configuration consists of two 16-bit flash devices operating in 16-bit mode, providing an effective 32-bit data bus width. Also, the only supported flash command set is the basic AMD command set.

The steps required to enhance the LatticeMico32 CFI flash service for supporting a custom configuration are as follows:

- Enhance the CFI flash configuration identification algorithm.
- Write your configuration-specific routines, as required by LatticeMico32 CFI flash service.
- Register your configuration information.

Enhance the CFI Flash Configuration Algorithm You must first enhance the provided flash configuration identification algorithm by updating the following three specific functions:

- CFIIdentifyConfiguration function

  The CFIIdentifyConfiguration function resides in CFICfgIdentifier.c. It is responsible for identifying the flashboard configuration, specifically the number of flash modules operating in parallel and the mode they are operating in, and must complete this information in the FlashBoardCfgInfo_t structure referenced by the pointer argument to this function. The CFI functions assume that you are using the same flash component in parallel, that is, that the programming characteristics of the flash devices operating in parallel are identical.

The CFIIdentifyConfiguration function is shown in Figure 87.

Figure 87: CFIIdentifyConfiguration Function

```
/*
 * this function uses basic heuristics to determine if there are flash-devices in
 * parallel and what mode they're operating in.
 *
 * Arguments:
 *   unsigned int base: base address of the flash component within processor's
 *       address space.
 *
 *    FlashBoardCfgInfo_t *BoardInfo: Pointer to a valid
 *      allocation of FlashBoardCfgInfo_t structure that
 *      this function will fill-in after trying to identify the
 *      flash component's physical layout.
 *
 * Return Value:
 *     0 if able to successfully identify CFI Flash layout.
 *     Non-zero if unable to successfully identify CFI Flash layout.
 *
 */
unsigned int CFIIdentifyConfiguration(unsigned int Base, FlashBoardCfgInfo_t *BoardInfo);
```

The FlashBoardCfgInfo_t structure is shown in Figure 88.

Figure 88: FlashBoardCfgInfo_t Structure

```
typedef struct st_FlashBoardCfgInfo {
     unsigned int parallels;             /* total flash devices sharing address bus */
     unsigned int mode;                  /* 8-bit mode, 16-bit mode, etc.           */
} FlashBoardCfgInfo_t;
```

In the FlashBoardCfgInfo_t structure, the first parallels element identifies the total flash devices operating in parallel. The second element, mode, identifies the mode (8, 16, 32) in which the devices are operating. This board configuration information is used to select the appropriate configuration functions, as described in subsequent sections.

- GetCFICfgAddressMultiplier

  The GetCFICfgAddressMultiplier function resides in CFICfgIdentifier.c. It returns the address multiplier that obtains the physical address offsets for the CFI tables when the CFI device offsets specified by the CFI specification are multiplied.

  For example, for two 16-bit flash devices operating in 16-bit mode, the CFI address multiplier is four; that is, the standard CFI address offsets must be multiplied by four to compensate for the board configuration and flash operational mode. You must use the BoardInfo parameter to identify the board configuration and provide the appropriate address multiplier as the return value.

  The GetCFICfgAddressMultiplier function is shown in Figure 89:

- ValidateCFIBoardCfg

  The ValidateCFIBoardCfg function resides in CFICfgIdentifier.c. It is used by the CFI routines to avoid using invalid board information.

  The ValidateCFIBoardCfg function is shown in Figure 90.

Figure 89: GetCFICfgAddressMultiplier Function

```
/*
 * Generates mulitplier for supported board-configurations.
 * This address multiplier is applied to the CFI device-offsets
 * to obtain the address offsets.
 *
 * Arguments:
 *    FlashBoardCfgInfo_t *BoardInfo: Pointer to a valid
 *      allocation of FlashBoardCfgInfo_t structure that
 *      contains flash board layout information that impacts
 *      the multiplier value.
 *
 * Return Value:
 *     The address-multiplier (1, 2, 4 etc) if boardInfo
 *     contains appropriate information, or, 0 implying
 *     the boardInfo does not contain identifiable data.
 */
unsigned int GetCFICfgAddressMultiplier(FlashBoardCfgInfo_t *BoardInfo);
```

Figure 90: ValidateCFIBoardCfg Function

```
/*
 * Validates Board Information provided as argument to this
 * function.
 *
 * Arguments:
 *    FlashBoardCfgInfo_t *BoardInfo: Pointer to a valid
 *      allocation of FlashBoardCfgInfo_t structure that
 *      contains flash board layout information.
 *
 * Return value:
 *    0 if the boardInfo contains valid data.
 *    Non-zero if the boardInfo contains invallid data.
 */
unsigned int ValidateCFIBoardCfg(FlashBoardCfgInfo_t *BoardInfo);
```

Implement Configuration-Specific Routines The LatticeMico32 CFI flash service relies on the configuration-specific function implementations shown in Table 9:

Table 9: Functions for the CFI Flash Service

Function	Description
ProgramData	Erases appropriate sectors and writes bulk data
SectorInfo	Retrieves sector-offset and sector-size that contains a specified offset
WriteData	Writes bulk-data (assumes affected sectors are erased)
WriteData8	Writes a byte at a given offset
WriteData16	Writes two bytes (short int) at a given offset
WriteData32	Writes four bytes (unsigned int) at a given offset

Table 9: Functions for the CFI Flash Service (Continued)

EraseChip	Erases entire flash region
EraseSector	Erases sector containing a given offset
FlashReset	Resets the flash parts in a flash region
FlashInit	Initializes flash parts in a flash region

Note

You can use AmdSCS_2_16_16.c and AmdSCS_2_16_16.h as a reference for implementing your configuration-specific routines. These files implement a 2 x 16 x 16 flash configuration, that is, two 16-bit flash devices operating in 16-bit mode, using the AMD command set. The routines in this file have been tested on Macronix 29LV128MBT flash device configuration. These files are located in asram_top\drivers\device folder in your LatticeMico components repository.

The prototypes for the expected functionalities are defined in the st_FlashCfgFnTbl structure in the LatticeMico32CFI.h header file. This structure is the flash configuration function table.

Figure 91: st_FlashCfgFnTbl Structure

```
typedef struct st_FlashCfgFnTbl {
    void (*Init) (CFIFlashDevCtx_t *);
    unsigned int (*ProgramData) (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned char *, unsigned int);
    unsigned int (*SectorInfo)  (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned int *SAddr, unsigned int *ByteLen);
    unsigned int (*WriteData)   (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned char *, unsigned int);
    unsigned int (*WriteData8)  (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned char);
    unsigned int (*WriteData16) (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned short int);
    unsigned int (*WriteData32) (CFIFlashDevCtx_t *, unsigned int ByteOffset, unsigned int);
    unsigned int (*EraseChip)   (CFIFlashDevCtx_t *);
    unsigned int (*EraseSector) (CFIFlashDevCtx_t *, unsigned int ByteOffset);
    unsigned int (*Reset)       (CFIFlashDevCtx_t *);
}FlashCfgFnTbl_t;
```

These function implementations must return 0 if the operation is successfully completed. If the operation is not successfully completed, they must return a non-zero value. You must allocate a static instance of this flash configuration function table structure and fill in the function pointers to the corresponding function implementation. You can set the function pointers of the structure element to zero (null) if your configuration does not need a corresponding functional implementation. This static instance is used in the next step as part of registering your configuration functions.

These function prototypes take the CFI flash device context structure as an argument. This structure is shown in Figure 92.

Figure 92: CFI Flash Device Context Structure

```
typedef struct st_CFIFlashDevCtx_t{
    const char * name;  /* name of the flash-component instance */
    unsigned int base;  /* base-address for the flash-component instance */
    unsigned int byteSize;  /* size (in bytes) of the flash component instance */
    unsigned int end;   /* end-address for the flash component instance */
    void *cfgFnTbl; /* pointer to the configuration-specific function table */
    CFIInfo_t CFIInfo; /* CFI information structure */
    void *prev; /* used internally by Flash service */
    void *next; /* used internally by Flash service */
}CFIFlashDevCtx_t;
```

Your function implementation does not need to populate information within this structure, because it is filled in by the LatticeMico32 CFI flash service. The two important elements of this structure are as follows:

- CFIInfo_t CFIInfo

  This element is the CFI information structure as defined in the CFIRoutines.h header file and contains CFI information for the identified flash part, such as timeout-values for writing and erasing and sector layout information that you can use as part of your function implementation.

- void *cfgFnTbl

  This element is a pointer to your FlashCfgFnTbl_t structure, which you provide as part of registering your configuration's functional routines.

Register the Configuration Function Table Once you have implemented the configuration routines, you must register this configuration information with the LatticeMico32 CFI flash service. This registration must occur before you use any of the LatticeMico32 CFI flash service APIs for writing or erasing the flash device. Registration allows the LatticeMico32 CFI flash service to refer to the functions provided in the function table for flash operations once the flash configuration has been identified.

There are two ways to register:

- Modify the InitializeCFIConfigurations function located in the CFIFlashConfigurations.c source file to register your function table using the LatticeMico32RegisterFlashCfg function.
- Call the LatticeMico32RegisterFlashCfg function from your application before calling any of the LatticeMico32 CFI flash service APIs for writing or erasing the flash device.

The LatticeMico32RegisterFlashCfg function is shown in Figure 93.

Figure 93: LatticeMico32RegisterFlashCfg Function

```
/*
* This function registers a flash configuration with
* LatticeMico32 CFI Flash Service. This function must be called
* prior to performing any of the LatticeMico32 CFI Flash
* Service API (write/erase/program).
*
* Arguments:
*   FlashConfiguration_st * pCfg: pointer to a valid flash
*     configuration information structure. This structure must
*     not be modified once registered.
*
* Return Values:
*     unsigned int 0: If configuration was successfully
*     registered.
*/
unsigned int LatticeMico32RegisterFlashCfg
(FlashConfiguration_st *pCfg);
```

This function takes a single parameter that is a pointer to a FlashConfiguration_st structure, shown in Figure 94. You must allocate a static instance of this structure and provide its pointer as part of registration.

Figure 94: FlashConfiguration_st Structure

```
typedef struct st_FlashConfiguration{
    unsigned int                VendorCSId;     /* Vendor Command-Set Id
    const FlashBoardCfgInfo_t   *boardInfo;     /* Board configuration information
    const FlashCfgFnTbl_t       *cfgFnTbl;      /* Configuration specific function table
}FlashConfiguration_st;
```

This structure has three elements:

- VendorCSId – You must fill in this structure element with the value corresponding to the supported CFI flash command-set identification.
- BoardInfo – This is a pointer to a static allocation of the FlashBoardCfgInfo_t structure described earlier. You must fill in the elements of this structure with the expected flash layout information (number of flash devices in parallel and operating mode).
- cfgFnTbl – This is a pointer to a static allocation of the FlashCfgFnTbl_t structure that you created in the previous step. This structure contains pointers to your flash configuration's function implementation.

The LatticeMico32 CFI flash service uses the command-set identification and the board information that you provided to register your flash configuration's functional implementation to determine if the CFI flash device that it finds has a corresponding functional implementation. Then it fills this information in the CFI flash device context structure. This information is filled in the first time that an application calls a LatticeMico32 CFI flash-service API function. If a

configuration function table that matches the command-set identification and the flash layout information is not found, the LatticeMico32 CFI flash APIs return failure codes, as described in “CFI Flash Device Service” on page 105.

The LatticeMico32 CFI flash service APIs invoke the functions pointed to by the flash configuration function table contained in the CFI flash device context to perform write, erase, and program operations on the flash device.

Flash Memory Configurations

The CFI programming code supports several flash memory configurations, which are shown in Table 10.

Table 10: Flash Memory Configurations

Number of PROMs	Number of Data Bits	Data Bus Width
1	8	8
2	8	16
4	8	32
1	16	16
2	16	32

The ability to switch between big-endian mode and little-endian mode is only available in the 1x16x16, 2x16x32, and 1x32x32 PROM configurations. If you have a platform with any one of these PROM configurations, you can connect them in either big-endian mode or little-endian mode.

Two .lpf files are provided that permit the asynchronous data bus to be configured in big-endian or little-endian mode. For each platform*X*, where *X* is A, B, C, D, E, F, G, or H, there is an ecp or ecp2 subdirectory that contains the HPE_MINI.lpf file. The HPE_MINI.lpf file is configured for little-endian mode, and the HPE_MINI.be.lpf file is configured for big-endian mode. The default is to use the little-endian version, but you can choose either for the LatticeMico32 HPE boards.

Once you build your platform and have an FPGA bitstream, you must match your CFI Flash Programmer C source code to the .lpf file that you are using. The C source code uses a #define variable to determine how to interpret the data being returned by the flash memory. By default, this variable is not defined, so the CFI Flash Programmer code uses little-endian encoding for the data.

If you right-click on the project name and select Properties > C/C++ Build > LatticeMico C++ Compiler > Preprocessor Options, and then use the LM32_BIG_ENDIAN_CONNECTION value to add a LatticeMico32 preprocessor definition and then (re)build the source code, the CFI Flash Programmer will expect the connections to the flash memory to be in big-endian form.

Device Driver Framework

This chapter describes the device driver framework in the LatticeMico System software, which is used by the run-time environment described in “LatticeMico Run-Time Environment” on page 45. This chapter offers alternatives for creating your own custom device drivers.

Overview

The LatticeMico32 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 component, that must be considered.

The flexibility of the Mico 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 145, the .xml file provides a mechanism to automatically extract the relevant information from the platform into the C/C++ SPE for software development.

The LatticeMico32 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
- A “Lookup” framework for easy access to instantiated components by name
- LatticeMico32 microprocessor interrupt framework

- Service to redirect standard input, output, and error streams to available character mode devices
- Prepackaged sample device drivers with easy-to-use APIs for components such as the LatticeMico timer and the LatticeMico parallel flash controller

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.

These device drivers can handle multiple instances of a component using the component-specific information structure. However, as part of your application development, you must still tell the device driver which instantiated device's instance-specific information to use. The device driver framework provides you with a device lookup service to access this instance-specific component information by simply providing the name of the device.

In addition to the device lookup service, the device driver framework provides some well-known services, such as the following:

- It can redirect standard I/O to character-mode devices.
- It can implement a device interrupt management structure as part of the LatticeMico32 device driver. This service enables you to develop your own device drivers for your custom components.
- It can handle the initialization of the microprocessor and call appropriate component initialization routines.

Supported Components

The LatticeMico software framework provides lookup service for all registered devices. This lookup service is described in "Device Lookup Service" on page 91. This framework provides extensive microprocessor support, such as interrupt management and APIs for cache management, as well as access to some of the microprocessor registers.

In addition, the LatticeMico software framework attempts to provide generalized "services" for some generic features, such as being able to redirect standard I/O streams to available character-mode devices and providing the ability to select a system timer. These services are as follows:

- Redirection of standard I/O to character mode devices:
  - LatticeMico UART
  - LatticeMico JTAG Debugger UART
- System timer service
  - LatticeMico timer component

- Flash devices
  - Common flash interface (CFI) compliant flash devices

Without knowing their end application, you cannot easily generalize all components into such abstract, application-level functionality, such as grouping SPI and I2C components. In such cases, the LatticeMico System software provides device drivers for these components. Each component provides a lookup service capability to allow ready access to the instance-specific component information required to access the device driver functions. These components are as follows:

- LatticeMico DMA component
- LatticeMico SPI master/slave component
- LatticeMico GPIO component

Beyond device drivers, the LatticeMico software framework incorporates Newlib C library support. The information on supported components in this section is subject to change in future releases of the LatticeMico System software. Check the Lattice Semiconductor Web site at www.latticesemi.com for updates or to obtain technical support.

Modifying Existing Device Drivers

This section shows you how to override the default behavior of the device drivers, as well as enhance the CFI flash services to support custom flash configurations. For information on creating device drivers for custom components using the C/C++ SPE for managed builds, see the section “Creating Custom Components” in the *LatticeMico32 Hardware Developer User Guide.*

Overriding Default Driver Initialization Sequence

As noted in earlier chapters, the boot-up sequence invokes 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 95.

Figure 95: DDInit.c File

```
#include "DDStructs.h"

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

void LatticeDDInit(void)
{
       /* PUT YOUR OWN IMPLEMENTATION HERE */

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



#ifdef __cplusplus
};
#endif /* __cplusplus */
```

These steps override the default implementation of LatticeDDInit, bypassing the LatticeMico 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 160.

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.

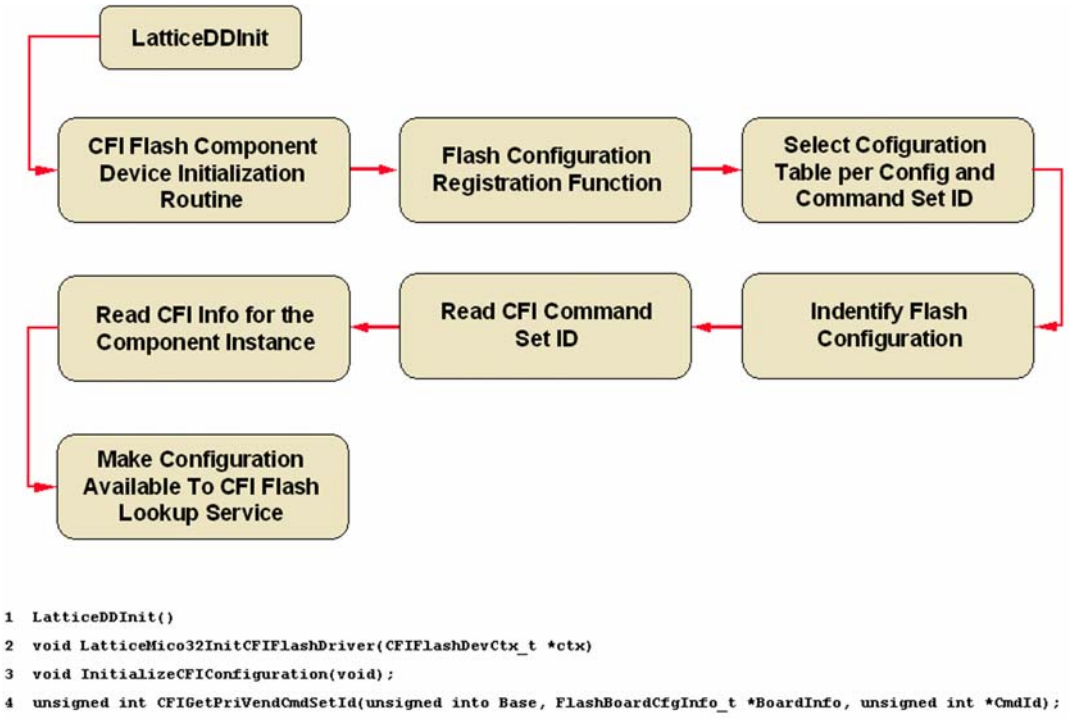
Some library files become part of the application build process instead of the library build process, such as crt0ram.S. The implementation in these files cannot be overridden as part of the LatticeMico C/C++ SPE managed build process.

Enhancing CFI Flash Service

Currently, the only supported flash configuration consists of two 16-bit flash devices operating in 16-bit mode, providing an effective 32-bit data bus width. Also, the only supported flash command set is the basic AMD command set.

Figure 96 shows the sequence that provides functional implementation to the CFI flash service user API calls. The first four steps shown in Figure 96 include the code associations with the corresponding steps shown in the diagram.

Figure 96: CFI Flash Service Flow



If your requirements involve a different flash configuration, a different registered CFI command set, or both, follow these three steps to enable your application to use the CFI flash service:

1. Enhance the flash configuration identification algorithm.
2. Write your configuration-specific routines
3. Registering your configuration's function table.

These steps are described in detail in the following sections.

Enhancing the Flash Configuration Identification Algorithm

You must first enhance the provided flash configuration identification algorithm. There are three specific functions that you must update to achieve this. The first is shown in the following example code:

```
unsigned int CFIIdentifyConfiguration(unsigned int Base,
FlashBoardCfgInfo_t *BoardInfo
```

This function is responsible for identifying the flashboard configuration, specifically the number of flash modules operating in parallel and the mode in which they are operating. The CFI functions assume that you are using the same flash component in parallel, that is, that the programming characteristics of the flash devices operating in parallel are identical.

This following function returns the address multiplier. As part of CFI specifications, the CFI tables are located at fixed device offsets.

```
unsigned int GetCFICfgAddressMultiplier(FlashBoardCfgInfo_t
*BoardInfo)
```

These offsets must be adjusted by a multiplier to account for the flash configuration. Multiplying the device offset with this address multiplier gives the effective address for the specific configuration. For example, two 16-bit flash devices operating in 16-bit mode cause the CFI table to be located at an effective address that is four times the individual device offset.

This multiplier is used by the CFI routines to fetch CFI information. As mentioned earlier, the CFI functions assume that you are using the same flash component in parallel, that is, that the programming characteristics of the flash devices operating in parallel are identical.

The following function inspects the board configuration parameters and returns 0 if the board configuration is valid:

```
unsigned int ValidateCFIBoardCfg(FlashBoardCfgInfo_t
*BoardInfo)
```

If the board configuration is invalid, it must return a non-zero value. This is used by the CFI functions to ensure that the board configuration is valid before accessing the flash devices.

To enhance these functions, copy the CFICfgIdentifier.c source file to your project folder. This file is located in the asram_top\drivers\device directory path located in your LatticeMico System components repository folder. This file contains the implementation of the functions. By making this file part of your project, you effectively override the default platform library functional implementations.

The following function takes the base address of the flash component as an argument:

```
unsigned int CFIIdentifyConfiguration(unsigned int Base,
FlashBoardCfgInfo_t *BoardInfo)
```

This function is responsible for identifying the flash configuration and returns 0 if it has successfully identified the CFI flash configuration and fills in the FlashBoardCfgInfo_t structure pointed to by this function's argument. The description of this structure is shown in Figure 97:

Figure 97: FlashBoardCfgInfo_t Structure

```
typedef struct st_FlashBoardCfgInfo  {
    unsigned int parallels;
    unsigned int mode;
}FlashBoardCfgInfo_t;
```

The first element, parallels, identifies the total flash devices operating in parallel. The second element, mode, identifies the mode (8, 16, or 32) in which the devices are operating. This board configuration information is used to select the appropriate configuration functions, as described in subsequent sections. You must use the provided base address to identify the flash configuration.

The following function validates the information provided in FlashBoardCfgInfo_t. You must enhance this function to return 0 if the board information contained in it is invalid.

```
unsigned int ValidateCFIBoardCfg(FlashBoardCfgInfo_t
*BoardInfo)
```

The next function returns the address multiplier that must be used for accessing CFI-specific structures:

```
unsigned int GetCFICfgAddressMultiplier(FlashBoardCfgInfo_t
*BoardInfo)
```

For example, for the two 16-bit flash devices operating in 16-bit mode, the CFI address multiplier is four. That is, the standard CFI address offsets must be multiplied by four to compensate for the board configuration and flash operational mode. You must use the BoardInfo parameter to identify the board configuration and provide the appropriate address multiplier as the return value. This function is not invoked if the board configuration is invalid as part of the ValidateCFIBoardCfg function call.

Writing Your Configuration-Specific Routines

The next step is to write your configuration-specific programming routines. Essentially, you will be providing implementation for the configuration function table.

Note

You can use AmdSCS_2_16_16.c and AmdSCS_2_16_16.h as a reference for implementing your configuration-specific routines. These files implement a 2 x 16 x 16 flash configuration, that is, two 16-bit flash devices operating in 16-bit mode using the AMD command set. The routines in this file have been tested on a Macronix 29LV128MBT flash device configuration. These files are located in the asram_top\drivers\device folder in your LatticeMico components repository.

The code shown in Figure 98 is the flash configuration function table structure that is used by the CFI flash service API.

Figure 98: Flash Configuration Function Table Structure

```
typedef struct st_FlashCfgFnTbl {
    void (*Init) (CFIFlashDevCtx_t *);
    unsigned int (*ProgramData) (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned char *, unsigned int);
    unsigned int (*SectorInfo)  (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned int *SAddr, unsigned int *ByteLen);
    unsigned int (*WriteData)   (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned char *, unsigned int);
    unsigned int (*WriteData8)  (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned char);
    unsigned int (*WriteData16) (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned short int);
    unsigned int (*WriteData32) (CFIFlashDevCtx_t *, unsigned int ByteOffset,
                unsigned int);
    unsigned int (*EraseChip)   (CFIFlashDevCtx_t *);
    unsigned int (*EraseSector) (CFIFlashDevCtx_t *, unsigned int ByteOffset);
    unsigned int (*Reset)       (CFIFlashDevCtx_t *);
}FlashCfgFnTbl_t;
```

These functions correspond to the CFI flash service API function calls. You must provide appropriate implementation for the applicable functionality. You do not need to implement a function if it is not applicable to your flash configuration. Each function has two points in common:

- Each function returns a value of 0 if it is successful and non-zero if unsuccessful.
- Each function takes a pointer to the CFIFlashDevCtx_t structure. This structure corresponds to the LatticeMico flash component instance and contains the instance-specific information described in this section.

The CFIFlashDevCtx_t device structure is illustrated in the code example in Figure 99.

Figure 99: CFIFlashDevCtx_t Device Structure

```
typedef struct st_CFIFlashDevCtx_t{
    const char * name;  /* name of the flash-component instance */
    unsigned int base;  /* base-address for the flash-component instance */
    unsigned int byteSize;  /* size (in bytes) of the flash component instance */
    unsigned int end;   /* end-address for the flash component instance */
    void *cfgFnTbl; /* pointer to the configuration-specific function table */
    CFIInfo_t CFIInfo; /* CFI information structure */
    void *prev; /* used internally by Flash service */
    void *next; /* used internally by Flash service */
}CFIFlashDevCtx_t;
```

The cfgFnTbl pointer is set to the appropriate configuration function table, described in a later step. Your functional implementation can use the CFIInfo structure element to access the CFI-specific features identified for the CFI flash component. The CFI specific structures are defined in the CFIInfo_t.h

and CFIRoutines.h header files that can be found in the .\asram_top\drivers\device file path.

Once you have implemented the functions required of the configuration function table, you must implement a registration function that populates a static instance of the FlashCfgFnTbl_t structure with the appropriate function pointers. This registration function is used in “Registering Your Configuration’s Function Table” on page 129. The function to register your configuration-specific function table is as follows:

```
unsigned int
LatticeMico32RegisterFlashCfg(FlashConfiguration_st *pCfg)
```

You must call this function with a static pointer to FlashConfiguration_st so that the pointer is not invalidated after exiting the context within which this function is called. The description of the FlashConfiguration_st structure is given in Figure 100:

Figure 100: FlashConfiguration_st Structure

```
typedef struct st_FlashConfiguration{
    unsigned int                VendorCSId;     /* Vendor Command-Set Id                    */
    const FlashBoardCfgInfo_t   *boardInfo;     /* Board configuration information          */
    const FlashCfgFnTbl_t       *cfgFnTbl;      /* Configuration specific function table    */
}FlashConfiguration_st;
```

The FlashConfiguration_st structure has three parameters:

- VendorCSId, which is the registered CFI command set identification that your configuration-specific function implementation supports. For example, a value of 2 indicates that the command set is for the AMD/ Fujitsu standard command set. Refer to the CFI publication for a comprehensive list of registered CFI command set identification numbers.
- BoardInfo, which is a pointer to your board configuration information. This pointer must point to a static structure instance, that is, non-local, so that it remains valid for the duration of the program.
- cfgFnTbl, which is a pointer to your configuration-specific function table structure. This structure must be a static declaration so that it is valid for the duration of the program and not just the function within which it is referred.

Registering Your Configuration’s Function Table

You must register your configuration’s function table as an available configuration to the LatticeMico flash component device driver. To do this, you must modify the following function implementation:

```
void InitializeCFIConfigurations(void)
```

This function is called by the LatticeMico32 device driver before device identification. This function calls the registration functions of the available configurations, making them available for configuration identification by the LatticeMico32 device driver.

This function is implemented in the CFIFlashConfigurations.c source file located in the .\asram_top\drivers\device file path in your LatticeMico components repository. Copy this file to your project to override the default implementation. Modify this copy to invoke the registration function defined in “Writing Your Configuration-Specific Routines” on page 127.

Making Devices Available to Lookup Service

Any LatticeMico32 platform can have multiple instances of the same component. For example, a platform can have multiple instances of a timer component. The managed build process creates instance-specific component information structures. LatticeMico32 device drivers receive a pointer to the appropriate component information structure as a function argument, allowing them to operate on multiple instances of the same component.

The LatticeMico32 device lookup service enables you to fetch pointers to these component instance-specific information structures by simply providing the instance name specified when you create the platform in MSB. The component device driver must register the component instance with the LatticeMico lookup service to enable this feature and to allow your application access to the instance-specific component information. This section lists the steps required of the component device driver.

Note

You can refer to any of the LatticeMico32 device software implementations for an example of writing device drivers that use the LatticeMico lookup service. The GPIO lookup service is an ideal example of the device lookup service implementation.

The LatticeMico lookup service provides a simple function, MicoRegisterDevice, to the device drivers for registering component instances. Example function code that illustrates this function for registering component instances is shown in Figure 101:

Figure 101: Function for Registering Component Instances

```
/* Function for registering a device for lookup by name.
 * Arguments:
 *   DeviceReg_t *pDevReg: Pointer to a valid allocation of
 *      DeviceReg_t structure. This must remain valid and
 *      should not be modified for the duration of the
 *      application.
*/
unsigned int MicoRegisterDevice( DeviceReg_t *pDevReg );
```

This function must be called by the device driver with a pointer to a valid DeviceReg_t structure allocation. This structure is used by the LatticeMico lookup service and must not be modified after calling MicoRegisterDevice. The code example in Figure 102 shows the contents of the DeviceReg_t structure type.

Figure 102: Contents of the DeviceReg_t Structure

```
/* device registration structure */
typedef struct DeviceReg_st{
    const char *name;        /* name of the device           */
    const char *deviceType;  /* device type                  */
    void *priv;              /* device-specific context      */
                             /* information                  */
    void *prev;              /* USED BY LATTICEMICO32 LOOKUP */
                             /* IMPL.                        */
    void *next;              /* USED BY LATTICEMICO32 LOOKUP */
                             /* IMPL.                        */
}DeviceReg_t;
```

The device driver must fill in the following structure members:

- const char *name

  Specifies a unique name of the component instance (for example, timer0). The MicoGetDevice function uses this field when retrieving a named device.

- const char *deviceType

  Specifies the name of the device type to which the component belongs (for example, TimerDevice). The LatticeMico lookup service's MicoGetFirstDev and MicoGetNextDev functions use this device type name when searching for a list of devices belonging to a device type. Refer to "List of Device Types" on page 94 for a list of names used by LatticeMico32 device drivers.

- void *priv

  The device driver must provide a non-null (non-zero) pointer to this field, typically a pointer to the instance-specific component information structure. This pointer is returned by the LatticeMico32 device lookup functions.

File Operations

This section describes the file operations support in LatticeMico System.

File Operations Functions

The Newlib C library expects implementation of certain functions at the system-call level. Although the Newlib C standard C library is well documented, the list in Table 11 summarizes some of the basic system support that the Newlib C library requires to enable end-to-end functionality of the standard C library routines, specifically the I/O routines. In addition, the table lists the source files in which the system call functionality exists.

Table 11 lists only those functions required for file operations. Refer to Table 1 on page 46 for a comprehensive list of function implementations required by the Newlib C library.

Table 11: System Call Source Files and Functionality

System Call	Source File	Expected Functionality
_exit	MicoExit.S	Exits a program without cleaning up files.
_close	MicoFileClose.c	Closes a file.
_fstat	MicoFileStat.c	Gives status of an open file.
i_satty	MicoFileIsAtty.c	Queries output stream to see if it is a terminal.
_link	Not supported	Establishes a new name for an existing file.
_lseek	MicoFileSeek.c	Sets position in a file.
_open	MicoFileOpen.c	Opens a file
_read	MicoFileRead.c	Reads from a file.
_sbrk	MicoSbrk.c	Increases program data space.
_unlink	Not supported	Remove a file's directory entry.
_write	MicoFileWrite.c	Writes a character to a file.

The flow for standard C I/O operations originates at the Newlib C library level, which then calls the appropriate system call implementation to achieve the expected functionality. The LatticeMico File Service is a minimal, lightweight implementation of a system-level file device management layer. This service manages the list of available devices that support file operations. It also implements a basic "file" concept, as described in this section.

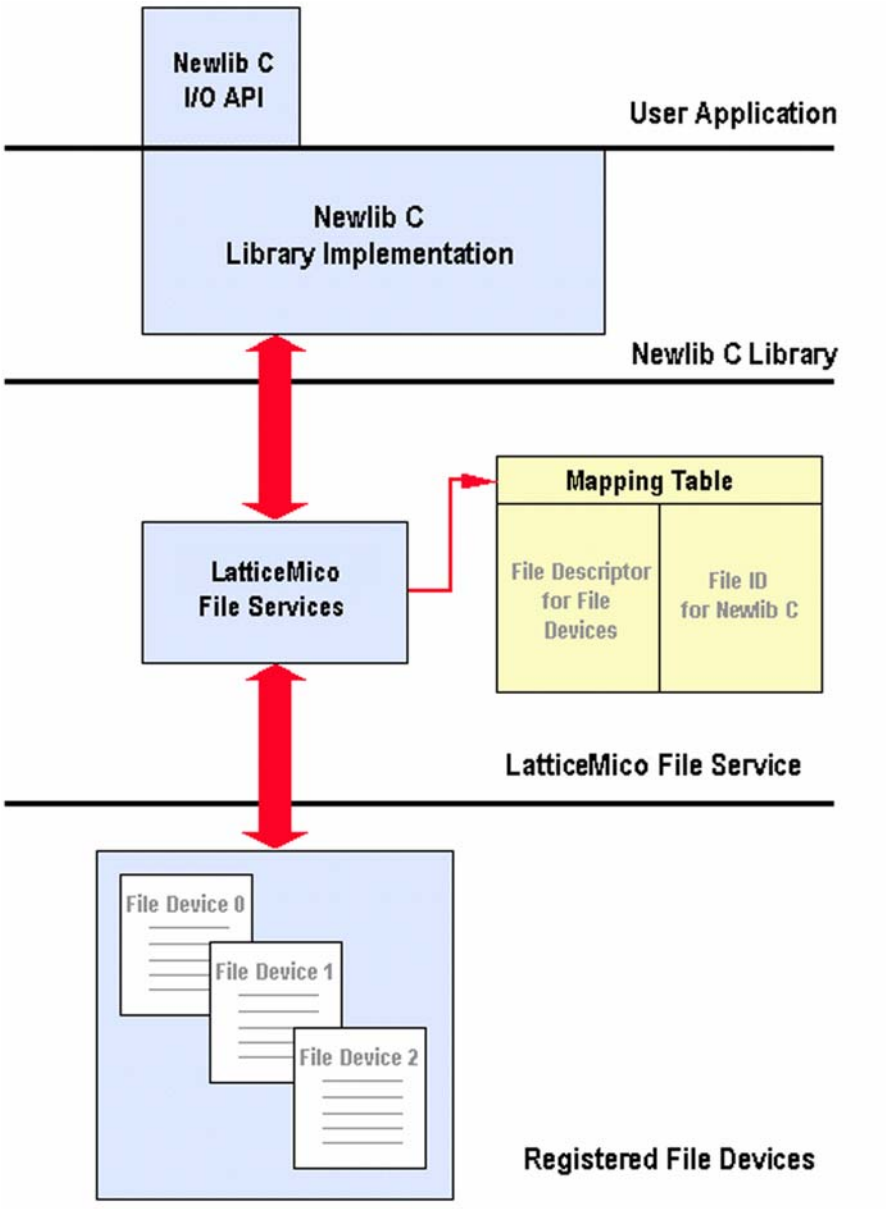
The system call functions are implemented in a separate file per function, allowing you to modify their implementation. The source files listed in Table 11 are part of LatticeMico32 microprocessor service source files and can be found in the LatticeMico32 microprocessor component software source folder. The core implementation of file devices management and file management can be found in the MicoFileDevices.c source file.

File Device and LatticeMico File Service

A device that supports file operations, as summarized in Table 11, is called a file device. A file device must implement the functionality summarized in Table 11 and explained in "Developing File Device Drivers" on page 137 for

the file device to be used by LatticeMico File Service for file operations. Figure 103 shows how the various software components layer themselves to provide end-to-end file operations support.

Figure 103: Component Layers for File Operations Support



A file device can register itself with LatticeMico File Service using the function shown in Figure 104.

Figure 104: Registering a File Service

```
/*
 ---------------------------------------------------------------------
 -                                                                   -
 -   This function registers a file-device.                          -
 -   NOTE: the pointer passed must remain valid for the duration     -
 -   of the program.                                                 -
 -   It returns 0 if successful, else, returns a non-zero value     -
 ---------------------------------------------------------------------
 -   Return Values:                                                  -
 -   0                                                               -
 -      successfully registered device as a file-ops device          -
 -                                                                   -
 -   MICO_FILE_ERR_PARAMETER_INVALID                                 -
 -      pDevCtx is null, or,                                         -
 -      pDevCtx->name is null, or,                                   -
 -      pDevCtx-> name is zero-length string                         -
 -      pDevCtx-> name is zero-length string                         -
 -      pDevCtx->p_FileFnTable is null                               -
 -                                                                   -
 -   MICO_FILE_ERR_DEVICE_EXISTS                                     -
 -      A device with the same name is already registered            -
 ---------------------------------------------------------------------
*/
unsigned int MicoRegisterFileDevice(MicoFileDevice_t *pDevCtx);
```

As part of file device registration, a pointer to a valid MicoFileDevice_t structure is passed as an argument. This structure must remain valid for the duration of the program since the LatticeMico File Service repeatedly accesses it.

This structure description is provided in the example code shown in Figure 105:

Figure 105: st_MicoFileDevice_t Structure Description

```
/*****************************************************************************
 *                                                                           *
 *                                                                           *
 * This structure defines the elements needed when registering a file-device. *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
struct st_MicoFileDevice_t{
    const char *name;                 /* device's name                                  */
    MicoFileFnTable_t *p_FileFnTable; /* pointer to device's file ops table; MUST REMAIN VALID */
    void *priv;                       /* device-specific private data                   */
    void *next;                       /* USED INTERNALLY: DO NOT MODIFY                 */
    void *prev;                       /* USED INTERNALLY: DO NOT MODIFY                 */
};
```

This registration information contains a pointer to the MicoFileFnTable_t structure, which provides LatticeMico File Services access to the registering device's file operations functions. The device must implement only the desired file support functionality that it intends to serve. It can leave the rest of the

functions unimplemented and mark their entries in this structure, described following, with a 0 value.

The LatticeMico File Service function calls, invoked by the Newlib C library, look up appropriate devices based on the file names and call the registered device's function as provided in the registration data, providing end-to-end file operations support.

Newlib C library's _open function invocation expects an integer return value that represents an "opened file." If file operations are to support the concept of a "file" on the interface between LatticeMico File Service and the registered file components, there must be information that is unique per "open" invocation.

The MicoFileFnTable_t structure is illustrated in Figure 106.

Figure 106: MicoFileFnTable_t Structure

```
/*************************************************************
 *                                                           *
 * This structure defines the expected function-table from   *
 * (and hence the expected functionality) of file-type devices *
 *                                                           *
 *************************************************************/
typedef struct st_MicoFileFnTable_t MicoFileFnTable_t;
struct st_MicoFileFnTable_t {
    int (*open)     (MicoFileDesc_t *fd, const char *name);                      /* open function-pointer      */
    int (*close)    (MicoFileDesc_t *fd);                                        /* close function-pointer     */
    int (*read)     (MicoFileDesc_t *fd, char *buffer, unsigned int bytes);      /* read function-pointer      */
    int (*write)    (MicoFileDesc_t *fd, const char *buffer, unsigned int bytes); /* write function-pointer     */
    int (*lseek)    (MicoFileDesc_t *fd, int ptr, int dir);                      /* lseek implementation       */
    int (*unlink)   (MicoFileDesc_t *fd, char *name);                            /* unlink a named file entry  */
    int (*link)     (MicoFileDesc_t *fd, char *name);                            /* link a named file entry    */
    int (*stat)     (MicoFileDesc_t *fd, struct stat* buf);                      /* file-status implementation */
    int (*ioctl)    (MicoFileDesc_t *fd, int req, void* arg);                    /* ioctl implementation       */
    int (*isatty)   (MicoFileDesc_t *fd);                                        /* isatty implementation      */
};
```

This unique information can then be used for the other file operations, such as the read, write, and close operations. For example, it can help the file components recognize which "file" is being operated on, especially if a component can open multiple "files" simultaneously. For example, a UART operating as a simple I/O device does not support the concept of multiple files and it does not need to "open" a file or "close" a file. On the other hand, a full-featured file system typically supports concepts of multiple files, so it must know on which file the operations are being performed.

The LatticeMico File Service provides this unique, per-file information through the MicoFileDesc_t pointer parameter in the file operations routines implemented by the file device. The st_MicoFileDesc_t structure is shown in Figure 107.

Figure 107: st_MicoFileDesc_t Structure

```
/*************************************************************
 *                                                           *
 * This structure defines a file-descriptor that is passed to *
 * registered devices for file-operations                    *
 * This file-descriptor is unique for each opened file       *
 * till the file is closed.                                  *
 *                                                           *
 *************************************************************/
typedef struct st_MicoFileDesc_t{
    int mode;                          /* mode associated with file when performing open operation     */
    int flags;                         /* flags associated with file when calling the open operation   */
    int special;                       /* special purpose                                              */
    MicoFileFnTable_t *pFileOpsTable;  /* pointer to registered file-operations table                  */
    MicoFileDevice_t *pDevice;         /* device's registration information                            */
    void *priv;                        /* device-specific private data provided at registration time   */
    void *pData;                       /* additional file-specific storage pointer for user's use      */
}MicoFileDesc_t;
```

The st_MicoFileDesc_t structure is allocated by the LatticeMico File Service on a file-open function call. A pointer to it is provided as an argument to the file device function calls, allowing the devices to uniquely identify the file on which the function must operate. The LatticeMico File Service maintains an internal one-to-one mapping between the integer file identification that it passes to Newlib C library and the file descriptor parameter that it allocates.

This file descriptor has two parameters, void *priv and void *pData, which the file device can use to maintain the file-specific information.

Maximum File Descriptors

The LatticeMico File Service allocates static array for available file descriptors, allowing accurate estimate of the resources being used. The maximum available file descriptors are dictated by the value of the MICO_FILE_DEVICES_MAX_DESCRIPTORS macro defined in the MicoFileDevices.h file. You can override this value by specifying an alternate value in the C/C++ SPE Project Properties dialog box, as described in “Setting Project Properties” on page 26.

The code in MicoFileDevices.c enforces a minimum value of 3, because it needs a minimum of three file descriptors for the standard streams. If the value of MICO_FILE_DEVICES_MAX_DESCRIPTORS is set to 5, three of these descriptors are used for the standard streams, and the rest are available for general file operations.

The code example shown in Figure 108 shows the definition of this macro in the MicoFileDevices.h header file.

Figure 108: MICO_FILE_DEVICES_MAX_DESCRIPTORS Macro in the MicoFileDevices.h Header File

```
   /************************************************************
    *                                                          *
    * Declares max file descriptors, that is, max opened files*
    * MUST BE A MINIMUM of 3 (stdio and stderr)                *
    *                                                          *
    ************************************************************
#ifndef MICO_FILE_DEVICES_MAX_DESCRIPTORS
#define MICO_FILE_DEVICES_MAX_DESCRIPTORS (5)
#endif
```

Developing File Device Drivers

This section describes the steps required to develop a file device driver. A file device driver resides between the LatticeMico File Service and a physical device driver, such as a UART or a disk drive. The file device driver is responsible for handling file operation requests from the LatticeMico File Service as appropriate for the device it supports.

The steps involved in developing a file device driver are as follows:

1. Implement file operation functions required by the LatticeMico File Service.
2. Register the driver as an available file device with the LatticeMico File Service.

Note

LatticeMico32Uart.c and MicoUartService.c are sample file device implementations available for reference as part of LatticeMico software support. LatticeMico32Uart.c implements the LM32 JTAG UART file device, and MicoUartService.c implements the UART lookup service as well as UART file device.

Implementing the Operation Functions

The example code in Figure 109 shows the file operation functions required of any file device by LatticeMico File Service.

Figure 109: File Operation Function Pointers

```
/* file-open request handler */
int (*open)     (MicoFileDesc_t *fd, const char *name);

/* file-close request handler */
int (*close)    (MicoFileDesc_t *fd);

/* file-read request handler */
int (*read)     (MicoFileDesc_t *fd, char *buffer, unsigned int bytes);

/* file-write request handler */
int (*write)    (MicoFileDesc_t *fd, const char *buffer, unsigned int bytes);

/* file-seek request handler */
int (*lseek)    (MicoFileDesc_t *fd, int ptr, int dir);

/* unlink request handler: CURRENTLY NOT SUPPORTED */
int (*unlink)   (MicoFileDesc_t *fd, char *name);

/* link request handler: CURRENTLY NOT SUPPORTED */
int (*link)     (MicoFileDesc_t *fd, char *name);

/* file-stat request handler */
int (*stat)     (MicoFileDesc_t *fd, struct stat* buf);

/* ioctl request handler: CURRENTLY NOT SUPPORTED */
int (*ioctl)    (MicoFileDesc_t *fd, int req, void* arg);

/* isatty request handler */
int (*isatty)   (MicoFileDesc_t *fd);
```

These function pointers are part of a structure definition shown in Figure 110:

Figure 110: Function Operation Function Pointers Structure Definition

```
/*************************************************************
 *                                                           *
 * This structure defines the expected function-table from    *
 * (and hence the expected functionality) of file-type devices *
 *                                                           *
 *************************************************************/
typedef struct st_MicoFileFnTable_t MicoFileFnTable_t;
struct st_MicoFileFnTable_t {
    int (*open)     (MicoFileDesc_t *fd, const char *name);                     /* open function-pointer      */
    int (*close)    (MicoFileDesc_t *fd);                                       /* close function-pointer     */
    int (*read)     (MicoFileDesc_t *fd, char *buffer, unsigned int bytes);     /* read function-pointer      */
    int (*write)    (MicoFileDesc_t *fd, const char *buffer, unsigned int bytes); /* write function-pointer    */
    int (*lseek)    (MicoFileDesc_t *fd, int ptr, int dir);                     /* lseek implementation       */
    int (*unlink)   (MicoFileDesc_t *fd, char *name);                           /* unlink a named file entry  */
    int (*link)     (MicoFileDesc_t *fd, char *name);                           /* link a named file entry    */
    int (*stat)     (MicoFileDesc_t *fd, struct stat* buf);                     /* file-status implementation */
    int (*ioctl)    (MicoFileDesc_t *fd, int req, void* arg);                   /* ioctl implementation       */
    int (*isatty)   (MicoFileDesc_t *fd);                                       /* isatty implementation      */
};
```

Detailed information about these file operation functions is given in the “File Device Function Handlers” on page 142. The file device does not have to implement all the function handlers. For those function handlers not implemented, it must set the corresponding function pointers in the structure to zero. The rest of the function pointer elements of the structure must point to valid function implementations.

The file device must create the file handler structure shown in Figure 110 on page 138 and populate the elements of the structure. The file device must keep this structure for the duration of the program because the LatticeMico File Service can access it anytime for the duration of the program. Once this is done, it must provide the LatticeMico File Service with this information using the MicoRegisterFileDevice function.

Registering the Driver as a File Device

The API for registering the driver as a file device with the LatticeMico File Service is shown in Figure 111.

Figure 111: Registering a File Device

```
/*
 ---------------------------------------------------------------------
 -                                                                   -
 -   This function registers a file-device.                          -
 -   NOTE: the pointer passed must remain valid for the duration     -
 -   of the program.                                                 -
 -   It returns 0 if successful, else, returns a non-zero value     -
 ---------------------------------------------------------------------
 -   Return Values:                                                  -
 -   0                                                               -
 -      successfully registered device as a file-ops device          -
 -                                                                   -
 -   MICO_FILE_ERR_PARAMETER_INVALID                                 -
 -      pDevCtx is null, or,                                         -
 -      pDevCtx->name is null, or,                                   -
 -      pDevCtx-> name is zero-length string                         -
 -      pDevCtx-> name is zero-length string                         -
 -      pDevCtx->p_FileFnTable is null                               -
 -                                                                   -
 -   MICO_FILE_ERR_DEVICE_EXISTS                                     -
 -      A device with the same name is already registered            -
 ---------------------------------------------------------------------
*/
unsigned int MicoRegisterFileDevice(MicoFileDevice_t *pDevCtx);
```

The MicoRegisterFileDevice function has a single parameter, as shown in the code example in Figure 112.

Figure 112: Structure for Registering a File Device

```
/*****************************************************************************
 *                                                                           *
 *                                                                           *
 * This structure defines the elements needed when registering a file-device. *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
typedef struct st_MicoFileDevice_t MicoFileDevice_t;
struct st_MicoFileDevice_t{
    const char *name;                    /* device's name                                        */
    MicoFileFnTable_t *p_FileFnTable;    /* pointer to device's file ops table; MUST REMAIN VALID */
    void *priv;                          /* device-specific private data                         */
    void *next;                          /* USED INTERNALLY: DO NOT MODIFY                       */
    void *prev;                          /* USED INTERNALLY: DO NOT MODIFY                       */
};
```

The file device must create this registration structure, populate it, and provide a pointer to it when registering itself with LatticeMico File Service. This structure must remain valid for the duration of the program. As part of this registration information structure, the file device can store any device-specific information in the “priv” member. This “priv” member is passed to the file device’s function handlers.

All of the file device function handlers are provided with a file descriptor pointer as an argument. This pointer is provided by the LatticeMico File Service. It is allocated when opening a file (fopen) and freed when the file is closed (fclose). This descriptor remains valid between fopen and fclose and is not reused during this period, guaranteeing the file device that this descriptor will remain unique for an open file. Once the file is closed, the LatticeMico File Service is free to recycle this descriptor and may be used for another open file.

Figure 113 shows the contents of this file descriptor parameter passed to the file device function handlers.

Figure 113: File Descriptor Parameter Passed to File Device Function Handlers

```
/*************************************************************
 *                                                           *
 * This structure defines a file-descriptor that is passed to *
 * registered devices for file-operations                    *
 * This file-descriptor is unique for each opened file       *
 * till the file is closed.                                  *
 *                                                           *
 *************************************************************/
typedef struct st_MicoFileDesc_t{

    /* mode associated with file when performing fopen operation */
    int mode;

    /* flags associated with file when calling fopen operation   */
    int flags;

    /*
     * Special purpose: Used by LatticeMico32 File Service when
     * opening a standard stream
     */
    int special;

    /* pointer to registered file-operations table */
    MicoFileFnTable_t *pFileOpsTable;

    /* device's registration information */
    MicoFileDevice_t *pDevice;

    /* device-specific private data provided at registration time */
    void *priv;

    /* additional file-specific storage pointer for user's use */
    void *pData;

}MicoFileDesc_t;
```

The parameters in this file are the following:

- mode

  Corresponds to the mode parameter of the fopen function call.

- flags

  Corresponds to the the flag parameter of the fopen function call.

- special

  Is a special parameter. Normally, when the file device's open function handler is invoked as part of an fopen function call, this parameter is set to a value other than 0, 1, or 2 and should be ignored. However, if the file device's open function handler is invoked as part of setting up the file device to handle a standard stream (0 for standard input, 1 for standard output, and 2 for standard error), this parameter is set to the appropriate standard stream identifier (0, 1, or 2). For this open invocation, the file

name is null. This special parameter helps to determine whether the open invocation is part of redirecting a standard stream or opening a file.

- pFileOpsTable

  Is a pointer to the file device's registered function handler table that was provided as part of the registration structure.

- pDevice

  Is the pointer to the file device's registration structure that was provided as part of registering the device with LatticeMico32 file system.

- priv

  Is a pointer to the file device's device-specific parameter that was provided in the registration structure as part of the file device registration process. For example, a component such as the LatticeMico UART uses this structure element to store the pointer to the UART instance-specific component information structure. It is used by the LatticeMico UART file device software routines to extract the instance-specific information when performing file operations.

- pData

  Is a structure element that can be used by the file device to store file-specific information. For example, the LatticeMico32 microprocessor JTAG UART uses this file-specific structure element to store the mapping between the host development computer's file ID and the file ID on the LatticeMico32 microprocessor. This mapping, which is based on a per-opened-file basis, allows the LatticeMico32 microprocessor JTAG UART to perform file operations on multiple opened files simultaneously.

File Device Function Handlers

This section lists the file device's function handlers that the LatticeMico File Service uses.

open

```
int (*open) (MicoFileDesc_t *fd, const char *<name>);
```

The LatticeMico File Service invokes the open function handler when it receives a request to open a file or a device through fopen (or open), or when it needs the device to prepare itself for handling a standard stream.

The LatticeMico File Service strips the device name, if present, and provides a pointer to the file name (path and name of the file) as the "name" argument to the "open" function handler. This pointer is not valid beyond the scope of the "open" function handler. Refer to "Usage and File Name/Device Name Conventions" on page 101 for information on device- and file-naming conventions.

The name of the file is provided in the <*name*> argument to this function. The LatticeMico File Service handles the <*filename*> parameter of the C API to

extract the device name and the <*filename*> parameters. It passes only the file name to this handler.

To keep the implementation of LatticeMico File Service small, no checks are done on the mode and the flags. The LatticeMico File Service also does not check to see if the requested file was already opened in an exclusive mode. These checks are left to the file device's open handler since not all devices may need to perform these checks. The flag and mode parameters are stored in the file descriptor passed as an argument to this function.

This handler must return a value of 0 if it was successfully able to open the file (or the device itself). If it returns a value other than 0, the LatticeMico File Service assumes that the device was not able to fulfill the open request. On a failure, LatticeMico makes the file descriptor available for other open requests and returns a failure to the Newlib C invocation.

If the LatticeMico File Service invokes this handler for preparing the device to handle a standard stream, the stream ID is stored in the "special" element of the file descriptor, and the file device must ignore the <*filename*>, <*mode*>, and <*flags*> parameters of the file descriptor.

close

```
init (*close) (MicoFileDesc_t *fd);
```

The LatticeMico File Service calls the close function handler when closing a file device as part of the fclose function call. The file device must perform any cleanup required before file closure. Once this function call returns, the file descriptor is made available for other open requests.

The LatticeMico File Service expects a return value of 0 for a success and a non-zero value for a failure.

read

```
int (*read (MicoFileDesc_t *fd, char *buffer, unsigned int
bytes);
```

The LatticeMico File Service calls the read function handler when it is asked to perform a file read operation. The file descriptor, provided as an argument to this function, is the file descriptor provided on a successful open.

The LatticeMico File Service expects a positive value to represent the bytes actually read by the file device. The file device can return 0 if it fails to read any bytes and can return a negative value to signal an error.

write

```
int (*write) (MicoFileDesc_t *fd, const char *buffer, unsigned
in bytes);
```

The LatticeMico File Service calls the write function handler when it is asked to perform a file write operation. The file descriptor, provided as an argument to this function, is the file descriptor provided on a successful open.

The LatticeMico File Service expects a positive value to represent the bytes actually written by the file device. The file device can return 0 if it fails to write any bytes and can return a negative value to signal an error.

lseek

```
int (*lseek) (MicoFileDesc_t *fd, int ptr, int dir);
```

The LatticeMico File Service calls the lseek function handler as part of the fseek invocation. Refer to the stdio.h header file or the Newlib C library documentation for information on the ptr and dir parameters.

The LatticeMico File Service passes the return value to the Newlib C caller.

stat

```
int (*stat) (MicoFileDesc_t *fd, struct stat* buf);
```

The LatticeMico File Service calls the stat function handler when requested by the Newlib C library's stat invocation. Refer to the Newlib C library documentation for information on the stat structure parameter.

isatty

```
int (*isatty) (MicoFileDesc_t *fd);
```

The LatticeMico File Service calls the isatty function handler when requested by the Newlib C library implementation. This function should return a value of 0 or 1 to indicate whether the device is a terminal device (1) or not (0).

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 LatticeMico C/C++ managed build process provides a framework for developing software applications targeting a LatticeMico32 microprocessor. 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 and manages required makefiles
- Creates a default linker script by identifying memory components in the platform

LatticeMico 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.

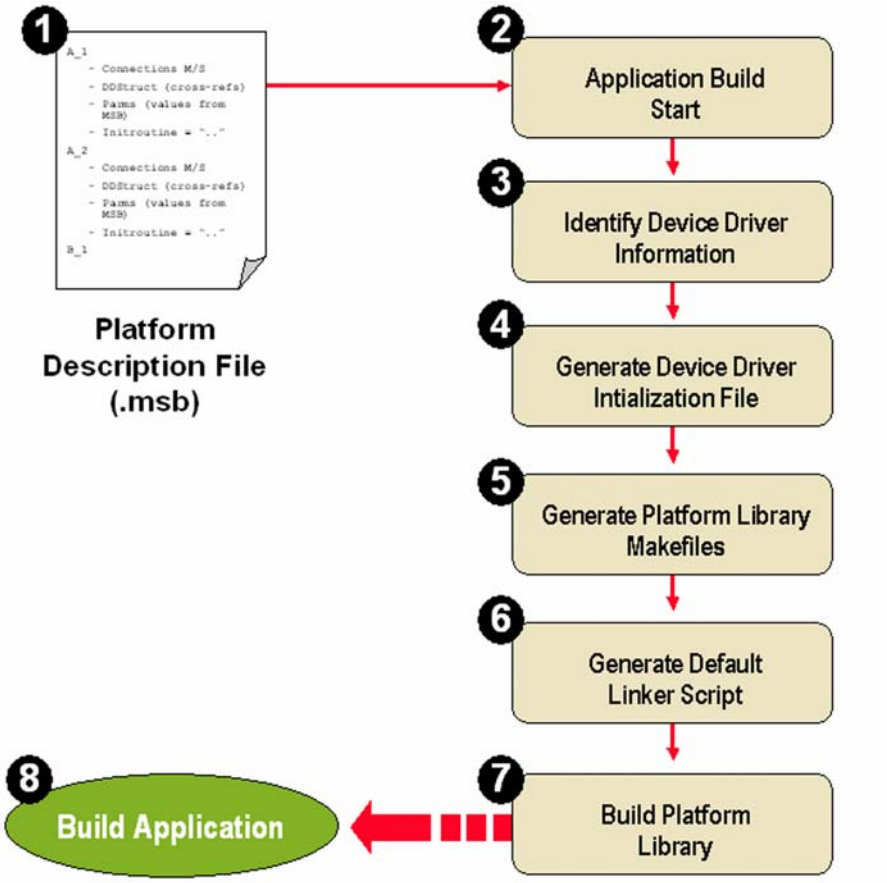
To clarify the build flow, a build example is provided in this chapter. It is based on the following information:

- LatticeMico C/C++ managed build project name: *MyProjectName*
- LatticeMico32 project folder: <*user_dir*>*MyProjectName*
- LatticeMico32 platform name: PlatformE
- LatticeMico32 build configuration name: Debug
- Application source code file name: LEDtest.c

The Build Process

The LatticeMico managed build process is centered on information contained in the .msb file generated by MSB as part of platform generation. Figure 114 illustrates the steps in the process of building an application from the .msb file that you initially create in MSB.

Figure 114: Managed Build Process Diagram



All of the steps presented in Figure 114, 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 24 and “Building Your Software Project” on page 25 and for more details.

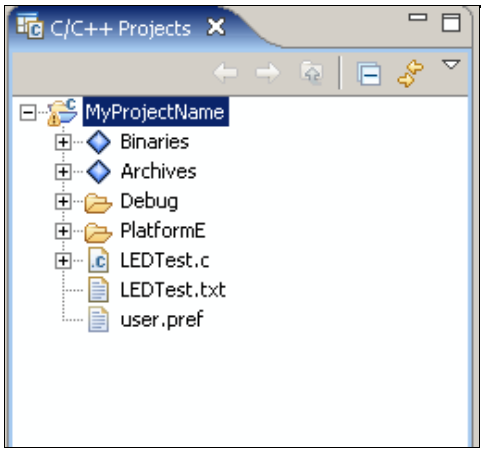
Build Directory Structure

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.

LatticeMico 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 115 shows the top-level outline for the project, as viewed within the C/C++ perspective's Projects view, after performing a build.

Figure 115: Top-Level Application Structure Outline



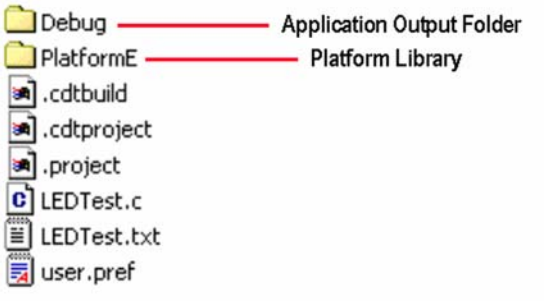
The Binaries and Archives folders in the Projects view do not actually exist in the project folder on the hard disk but contain certain files that are used by the project and are accessible here. Specifically, the Archives folder contains a platform library archive, and the Binaries folder contains an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

Figure 116 on page 149 shows how this top-level application structure in the Project view as shown in Figure 115 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 116 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 115.

Figure 116: Project Folder Contents



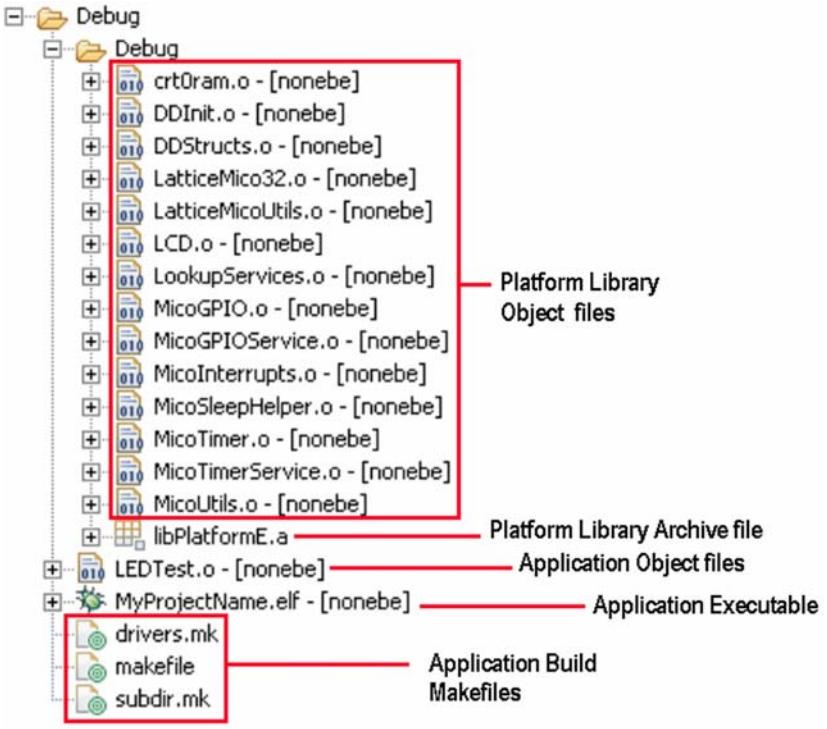
The project folder contains an application output (debug) folder, a platform library folder, and various project information and user files.

Application Output Folder

The application output folder, named debug in Figure 116 on page 149, contains the files that LatticeMico 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 117 shows the contents of the application output folder.

Figure 117: Application Output Folder Contents



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 117 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.

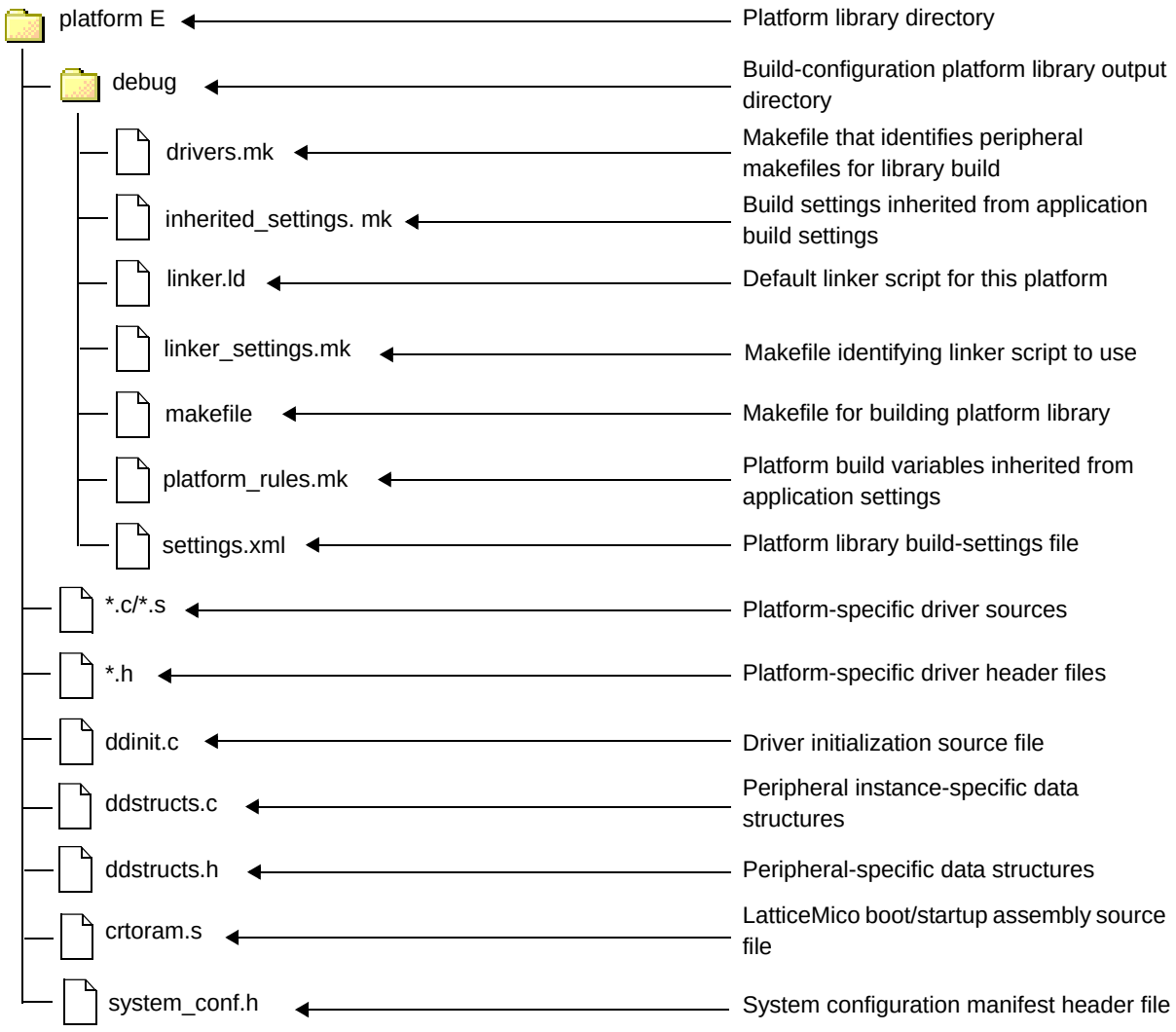
Platform Library Folder

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 118, 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 118: Platform Library Directory Structure



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 155. 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 LatticeMico 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 12 on page 27. 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 microprocessor 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 118 on page 152, a newly generated build configuration folder would be placed under the PlatformE 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 117 on page 150 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 14 on page 29. 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 151. 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, PlatformE, shown in Figure 117 on page 150, 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 118 on page 152, 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 118 on page 152. The contents of these source files depend on what components are in the platform being targeted. See the section “Creating Custom Components” in the *LatticeMico32 Hardware Developer User Guide* for more information on how these files are created when you add components.

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 157.

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:

- LatticeMico C/C++ Managed Build Project Name: *MyProjectName*
- LatticeMico32 Platform: PlatformE
- LatticeMico32 Build Configuration: Debug

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

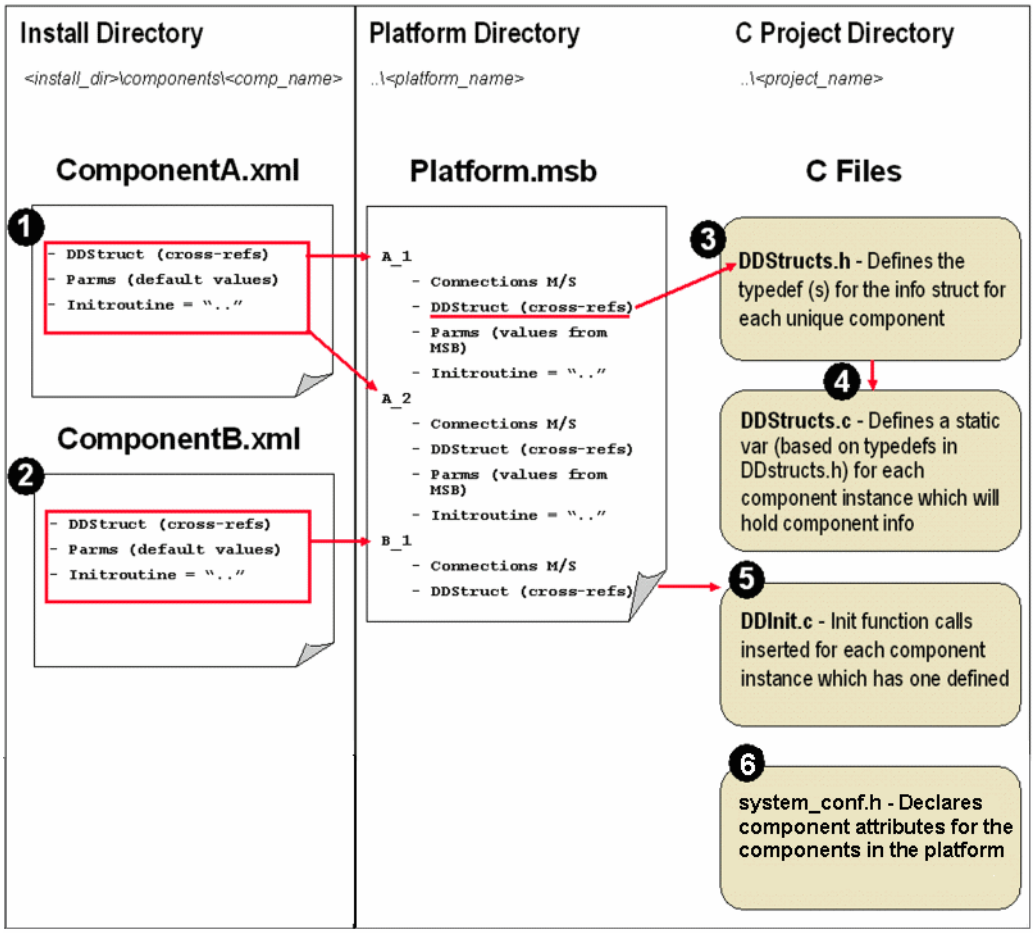
Figure 119 illustrates the following:

- 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 119: Component Information Flow to Platform Library Files



- All the C files on the right in Figure 119 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:

- MICO32_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. As shown in the sample DDStructs.h file in Figure 120, the LatticeMico timer and the LatticeMico GPIO components have their own component information structure definition. 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 timer named “timer.” Through extern, the file declares that a definition exists for the timer_timer instance of the st_MicoTimerCtx_t structure. See Figure 120. 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.

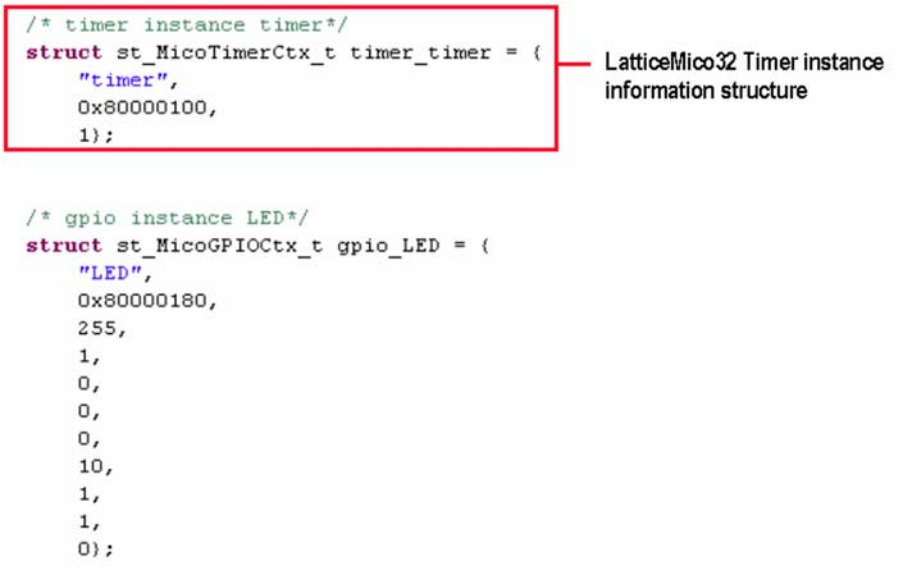
Figure 120: Sample DDStructs.h File


```
#ifndef LATTICE_DDINIT_HEADER_FILE
#define LATTICE_DDINIT_HEADER_FILE
#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

//* platform frequency in MHz */
#define MICO32_CPU_CLOCK_MHZ (25000000)

/*Device-driver structure for lm32_top*/
#define LatticeMico32Ctx_t_DEFINED (1)
typedef struct st_LatticeMico32Ctx_t {
    const char*   name;
} LatticeMico32Ctx_t;


/* declare lm32_top instance lm32_top_LM32*/
extern struct st_LatticeMico32Ctx_t lm32_top_LM32;

/* declare initialization routine for lm32_top */
extern void LatticeMico32Init(struct st_LatticeMico32Ctx_t*);

/*Device-driver structure for timer*/
#define MicoTimerCtx_t_DEFINED (1)
typedef struct st_MicoTimerCtx_t {
    const char*   name;
    unsigned int   base;
    unsigned int   intrLevel;
    void *   userCtx;
    void *   callback;
    void *   prev;
    void *   next;
} MicoTimerCtx_t;

/* declare timer instance timer_timer*/
extern struct st_MicoTimerCtx_t timer_timer;

/* declare initialization routine for timer */
extern void MicoTimerInit(struct st_MicoTimerCtx_t*);

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


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

/* declare initialization routine for gpio */
extern void MicoGPIOInit(struct st_MicoGPIOCtx_t*);

extern int main();


#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif
```

DDStructs.c File

Figure 121 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. For more details on this mechanism, see the section “Creating Custom Components” in the *LatticeMico32 Hardware Developer User Guide*..

Figure 121: Sample DDStructs.c File



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 121, you can see how this file uses the the timer instance information that it derived from the DDStructs.h file by comparing it to the sample DDStructs.h file shown in Figure 120.

The automatically generated DDInit.c file implements the LatticeDDInit function, which resets the CPU. It is described in “DDInit.c File” on page 160. 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. See “LatticeDDInit” on page 70 for more details on the LatticeDDInit function.

Figure 122: DDInit.c Source Code Sample

```
#include "DDStructs.h"

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

void LatticeDDInit(void)
{
    /* initialize LM32 instance of lm32_top */
    LatticeMico32Init(&lm32_top_LM32);

    /* initialize timer instance of timer */
    MicoTimerInit(&timer_timer);

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

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


#ifdef __cplusplus
);
#endif /* __cplusplus */
```

Components initialization routine called by `crt0` after CPU reset

Timer instance initialization function call

Call user-implemented `int main(void)` “main” 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/cygwin/lib/Perl5/5.8/ 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 123 shows the platform attributes for a sample platform in the system_conf.h header file.

Figure 123: Sample Platform Attributes in system_conf.h File

```
#define FPGA_DEVICE_FAMILY     "ECP"
#define PLATFORM_NAME          "PlatformH"
#define USE_PLL                (0)
#define CPU_FREQUENCY          (25000000)
```

Table 12 lists the platform attributes and their properties.

Table 12: Platform Attributes

Atttribute	C Type	Information
FPGA_DEVICE_FAMILY	String	FPGA device family selection in MSB
PLATFORM_NAME	String	MSB platform name
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 124 shows the processor attributes in the system_conf.h header file for a sample platform.

Figure 124: Sample Processor Attributes in system_conf.h File

```
/*
 * CPU Instance LM32 component configuration
 */
#define CPU_NAME "LM32"
#define CPU_EBA (0x00300000)
#define CPU_DIVIDE_ENABLED (1)
#define CPU_SIGN_EXTEND_ENABLED (1)
#define CPU_MULTIPLIER_ENABLED (1)
#define CPU_SHIFT_ENABLED (1)
#define CPU_DEBUG_ENABLED (1)
#define CPU_HW_BREAKPOINTS_ENABLED (0)
#define CPU_NUM_HW_BREAKPOINTS (0)
#define CPU_NUM_WATCHPOINTS (0)
#define CPU_ICACHE_ENABLED (1)
#define CPU_ICACHE_SETS (512)
#define CPU_ICACHE_ASSOC (1)
#define CPU_ICACHE_BYTES_PER_LINE (16)
#define CPU_DCACHE_ENABLED (1)
#define CPU_DCACHE_SETS (512)
#define CPU_DCACHE_ASSOC (1)
#define CPU_DCACHE_BYTES_PER_LINE (16)
#define CPU_DEBA (0x00000000)
#define CPU_CHARIO_IN         (1)
#define CPU_CHARIO_OUT        (1)
#define CPU_CHARIO_TYPE       "JTAG UART"
```

Table 13 lists the processor attributes and their properties. Refer to the *LatticeMico32 Processor Reference Manual* for the meaning of the features listed.

Table 13: Processor Attributes

Atttribute	C Type	Information
CPU_NAME	String	Processor instance name
CPU_EBA	Numeric	Processor exception base address (reset address).
CPU_DIVIDE_ENABLED	Numeric	Indicates divider selection: 0 indicates the absence of a hardware divide. 1 indicates the presence of a hardware divide.
CPU_SIGN_EXTEND_ENABLED	Numeric	Indicates sign extend selection: ▶ 0 indicates the absence of a sign-extend. ▶ 1 indicates the presence of a sign-extend.

Table 13: Processor Attributes (Continued)

Atttribute	C Type	Information
CPU_MULTIPLIER_ENABLED	Numeric	Indicates hardware multiplier selection: ▶ 0 indicates the absence of a hardware selection. ▶ 1 indicates the presence of a hardware selection.
CPU_SHIFT_ENABLED	Numeric	Indicates hardware shift selection: ▶ 0 indicates the absence of a hardware shift. ▶ 1 indicates the presence of a hardware shift.
CPU_DEBUG_ENABLED	Numeric	Indicates selection of debug module: ▶ 0 indicates the absence of the debug module. ▶ 1 indicates the presence of the debug module.
CPU_HW_BREAKPOINTS_ENABLED	Numeric	Indicates selection of hardware breakpoints: ▶ 0 indicates the absence of hardware breakpoints. ▶ 1 indicates the presence of hardware breakpoints.
CPU_NUM_HW_BREAKPOINTS	Numeric	Indicates the number of hardware breakpoints. This value is valid only if the hardware breakpoint feature is enabled.
CPU_NUM_WATCHPOINTS	Numeric	Indicates the number of watch points. This value is valid only if the debug module is present.
CPU_ICACHE_ENABLED	Numeric	Indicates the instruction cache selection: ▶ 0 indicates the absence of the instruction cache ▶ 1 indicates the presence of the instruction cache
CPU_ICACHE_SETS	Numeric	Indicates instruction cache set selection. This value is valid only if the instruction cache is present.
CPU_ICACHE_ASSOC	Numeric	Indicates the instruction cache associativity selection. This value is valid only if the instruction cache is enabled.
CPU_ICACHE_BYTES_PER_LINE	Numeric	Indicates the instruction cache line length. This value is valid only if the instruction cache is enabled.
CPU_DCACHE_ENABLED	Numeric	Indicates the data cache selection: ▶ 0 indicates the absence of the data cache. ▶ 1 indicates the presence of the data cache.
CPU_DCACHE_SETS	Numeric	Indicates the data cache set selection. This value is valid only if the data cache is present.
CPU_DCACHE_ASSOC	Numeric	Indicates data cache associativity selection. This value is valid only if the data cache is enabled.
CPU_DCACHE_BYTES_PER_LINE	Numeric	Indicates data cache line length. This value is valid only if the data cache is enabled.
CPU_DEBA	Numeric	Processor debug port address. This value is valid only if the debug module is included.

Table 13: Processor Attributes (Continued)

Atttribute	C Type	Information
CPU_CHARIO_IN	Numeric	Indicates the processor availability for the software file input routines, such as fread: ▶ 1 if the debug module is present. ▶ 0 if the debug module is absent.
CPU_CHARIO_OUT	Numeric	Indicates the processor availability for the file output routines, such as write or fprintf: ▶ 1 if the debug module is present. ▶ 0 if the debug module is absent.
CPU_CHARIO_TYPE	String	Constant value set to “JTAG UART”

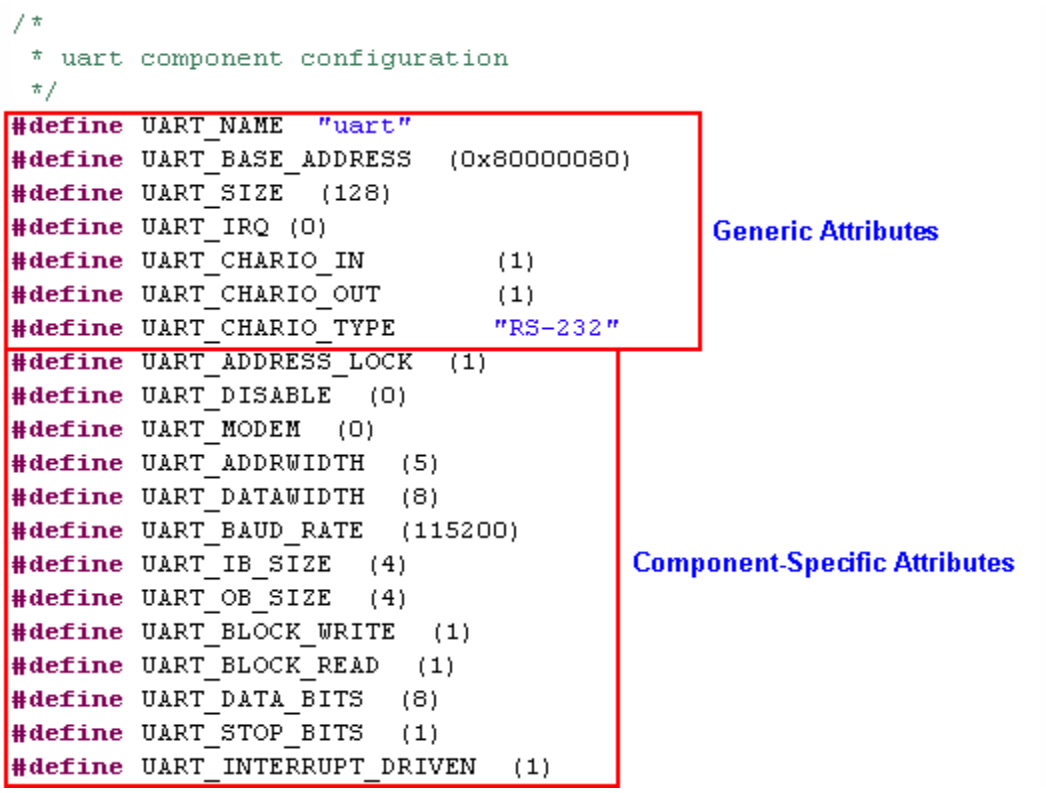
Attributes for I/O-Type Components

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

Figure 125: Sample UART Component Attributes in system_conf.h File



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 14 lists the generic attributes for all I/O-type components.

Table 14: 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 14: 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.

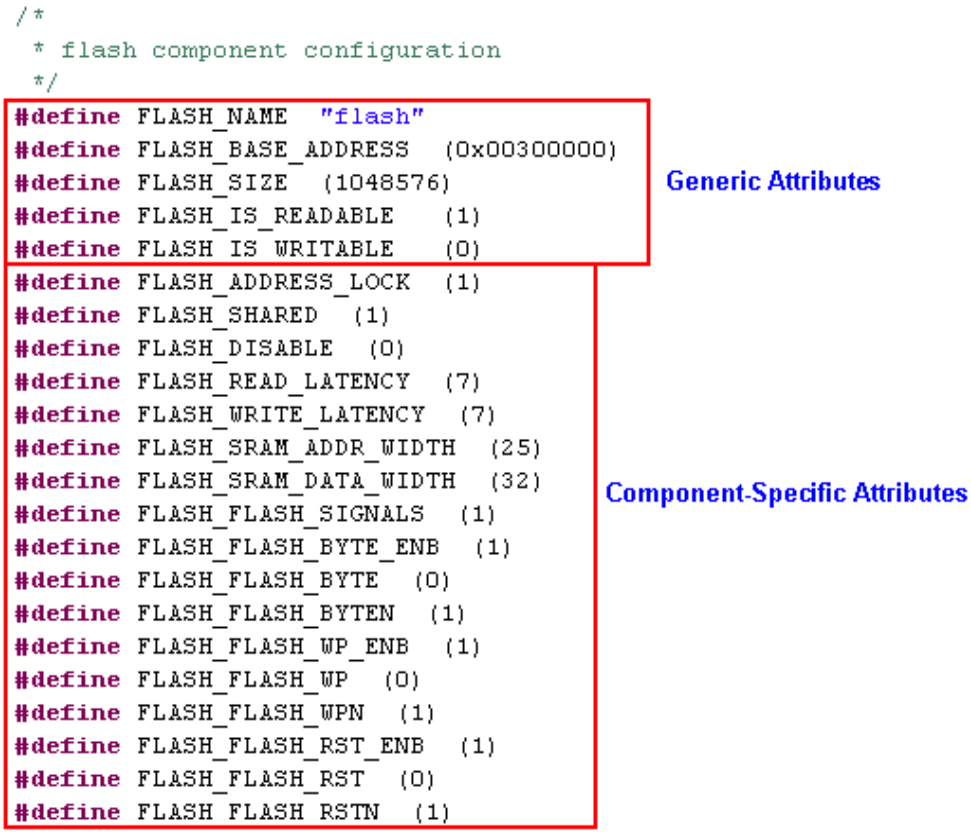
Attributes for Memory-Type Components

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

Figure 126: Sample Flash Component Attributes in system_conf.h File



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 15 lists the generic attributes present for all memory-type components.

Table 15: 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 15: Generic Attributes for Memory-Type Components (Continued)

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:

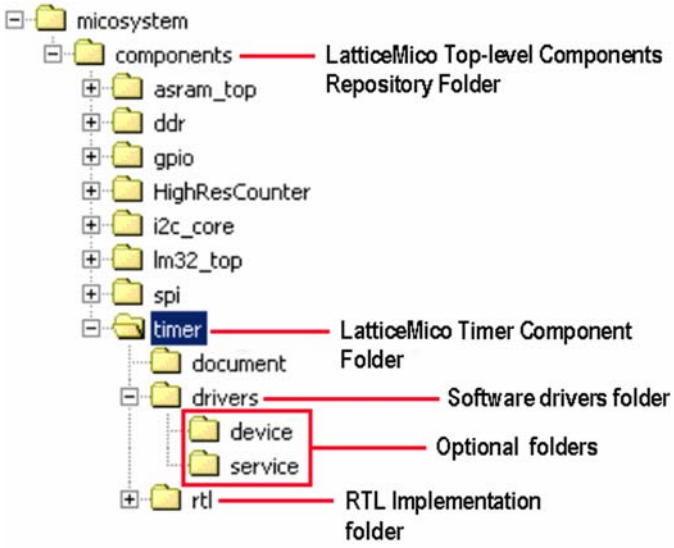
- <*comp_name*>.xml files, which exist for each element and reside in the ..\components\<*comp_name*> folder in your project. Each .xml file contains reference information on component initialization routines and a component information structure declaration that provides details about the component's 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 157.
- 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 127 depicts a typical directory structure for a Mico System Builder (MSB) component residing in the top-level components repository folder.

Figure 127: LatticeMico Timer Component Folder Directory Structure



The example used for this section is the LatticeMico timer device. In Figure 127, the timer component has two main subdirectories, drivers and RTL. The drivers folder contains software support for LatticeMico timer component, and the rtl folder contains RTL support.

The timer folder contains a single file, timer.xml. This .xml file is the timer 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 timer 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 timer 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.

For more informaton on designing components for the LatticeMico32 microprocessor and structural details of the files associated with this process, see the section “Creating Custom Components” in the *LatticeMico32 Hardware Developer User Guide*.

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 timer component provides easy-to-use API routines for manipulating the timer. 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 addition, the UART component also provides support for standard input/output redirection.

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 timer component subfolder and all of its contents are copied into your platform folder’s directory structure. You will not see the timer.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 128 shows a sample drivers folder’s directory structure for LatticeMico timer component. This figure is an extension of the LatticeMico timer component directory structure.

Figure 128: LatticeMico Timer Component Software Files

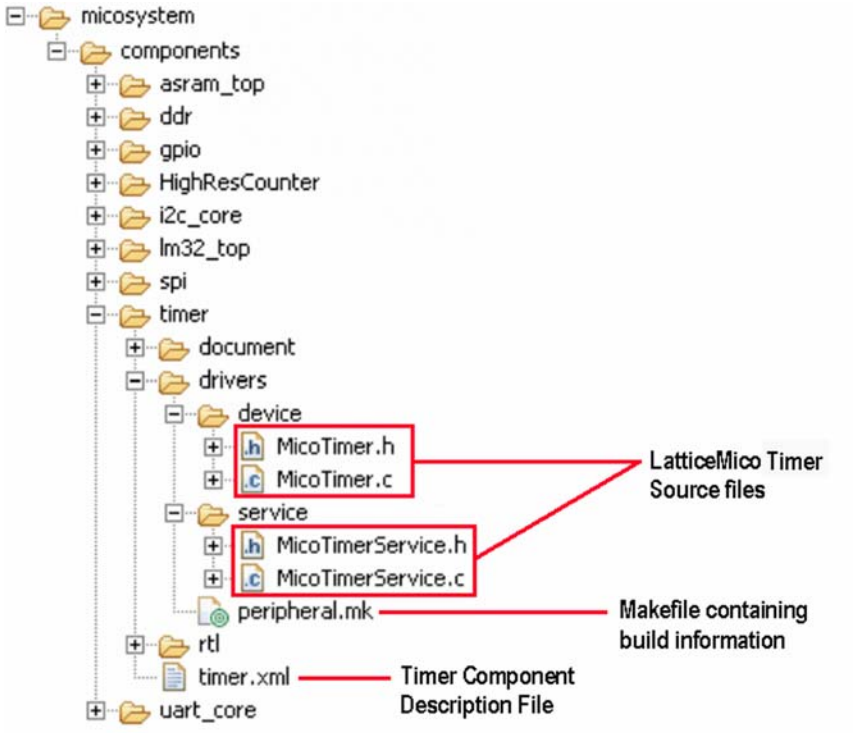


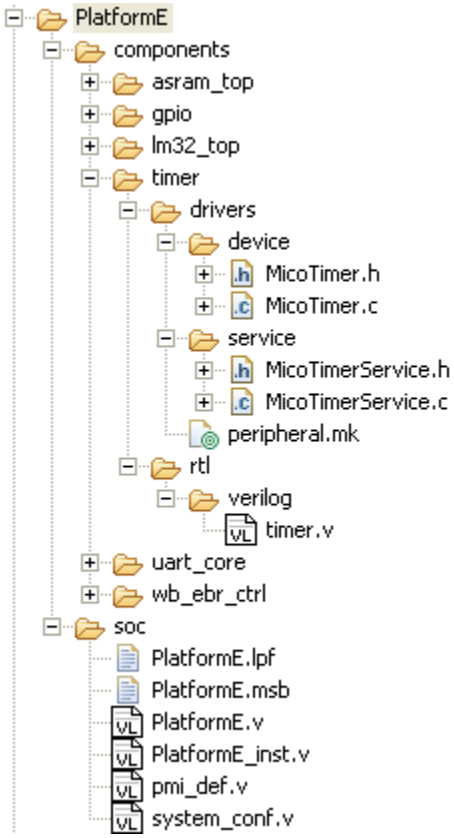
Figure 128 shows the directory structure for LatticeMico timer component as part of the components folder under the LatticeMico installation folder.

As part of platform generation (PlatformE in the current context), MSB generates the directory structure by copying the relevant RTL and device-driver directories under the components folder. In Figure 128, this example component folder is components\timer. If you compare the directory structures shown in Figure 128 and Figure 129, the .xml files are not copied across directories. Instead, the component description file contents are contained in the .msb file (PlatformE.msb).

It is this .msb file, PlatformE.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 timer component, it inspects the timer component's drivers folder in the directory structure created by MSB (shown in Figure 129). 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 117.

Figure 129: Directory Structure Created by MSB



The C/C++ SPE managed build also inspects the timer 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 130 shows the LatticeMico timer's peripheral.mk makefile.

Figure 130: Timer Makefile

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

- LIBRARY_CXX_SRCS

  Use this variable to identify C++ sources 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_CXX_SRCS

  Use this variable to identify C++ sources 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 timer makefile example in Figure 130. The C/C++ SPE build process generates only those components' peripheral.mk files that have a corresponding component instance information structure.

Advanced Programming Topics

This chapter introduces you to advanced programming topics, for example, linker script customization, software deployment, conversion of application .elf files to binary, and boot copier generation.

Linker Script and Memory Sections

The linker is responsible for combining multiple object files into a single ELF executable. In order to create a single ELF executable, it resolves cross-references between the different object files, groups together similar sections into one contiguous location, arranges for these sections to be loaded at the correct addresses in memory, and generates the necessary header information at the start of a file that allows it to be run.

Some sections in a LatticeMico ELF executable are predefined and hold specific program information. The most critical of these sections, at least from a software developer's perspective, are discussed below.

.text This section contains program instructions that will be executed by the microprocessor. This section can be configured for one of three scenarios:

- Load and execute in volatile memory
- Load and execute in non-volatile memory
- Load in non-volatile memory and execute in volatile memory.

.rodata This section contains read-only data, which means that the data is not modified by the executable at runtime. This section can be configured for one of three scenarios:

- Load and execute in volatile memory
- Load and execute in non-volatile memory

- Load in non-volatile memory and execute in volatile memory.

.bss This section contains read/write data that is initialized to zero at runtime. This section can be configured for one of two scenarios:

- Load and execute in volatile memory
- Load in non-volatile memory and execute in volatile memory.

.data This section contains initialized (non-zero) data that is modifiable at run time. This section can be configured for one of two scenarios:

- Load and execute in volatile memory
- Load in non-volatile memory and execute in volatile memory.

In addition, the program might contain a stack that is used to store register values across function calls. Of all the aforementioned sections, the LatticeMico C/C++ SPE managed build process allows a software developer to select the memory locations for the placement of ELF sections .text, .rodata, and .data. There are two memory addresses associated with these three sections. The first is the virtual memory address (VMA), which is the address the section will have when the executable is running. The second is the load memory address (LMA), which is the address in memory where the section will be loaded. In most cases, the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM and then copied in to RAM when the program starts executing. In this case, the ROM address would be the LMA and the RAM address would be the VMA.

You can use the lm32-elf-objdump utility to obtain information on various sections and their placements. See “IRunning the Software from the Command Line” on page 42 and Table 26 on page 294 for usage and valid options for the lm32-elf-objdump utility.

The LatticeMico C/C++ managed build process generates a default linker script, linker.ld, that encapsulates the user-provided placement information for the .text, .rodata, and .data sections of the executable. The placement of the .boot and .bss sections is still controlled by the managed build process to ensure that a valid executable is created by the linker script.

Table 16 on page 177 shows an example that articulates the legal placement combinations for .text, .rodata, and .data sections, the subsequent placement of .boot and .bss sections, and the "Linker Script" GUI settings that the user must manipulate to achieve the required placement for these sections.

Table 16: Example of Legal Placement Combinations

Location of Program Sections (under user control)					Location of Program Sections (NOT under user control)				User Interface				
.text		.rodata or .data			.boot		.bss					Deploy Flag	
VMA	LMA	VMA	LMA	Note	VMA	LMA	VMA	LMA	Program	RO or RW Data	Enable Deploy-ment	Program	RO or RW Data
RAM	RAM	RAM	RAM		RAM	RAM	RAM	RAM	RAM	RAM	N	N/A	N/A
RAM	ROM	RAM	RAM	1	ROM	ROM	RAM	RAM	RAM	RAM	Y	Y	N
ROM	ROM	RAM	RAM	1	ROM	ROM	RAM	RAM	ROM	RAM	Y	Y	Y
RAM	ROM	RAM	ROM		ROM	ROM	RAM	RAM	RAM	RAM	Y	Y	Y
ROM	ROM	RAM	ROM		ROM	ROM	RAM	RAM	ROM	RAM	Y	Y	Y
ROM	ROM	ROM	ROM	2	‘ROM	ROM	RAM	RAM	ROM	ROM	Y	Y	Y

1 Make sure that you use On-chip Memory Deployment or Multi On-chip Memory Deployment to ensure .rodata and .data are deployed in to RAM.

2 It is not advisable to use this combination, since ROM is a non-volatile memory and a location holding .data value must be erased prior to any write.

Note

To make sure the microprocessor starts executing your code on power-up, be sure to set the exception base address (EBA) to the LMA of the .text section.

The C/C++ SPE allows you to select a custom linker script for the software project through the Platform tab available in the Properties dialog box when you right-click on the software project in the Projects view. See Figure 141 on page 191 for details. You can use the default linker script generated by LatticeMico C/C++ SPE managed build process as a starting point for your custom linker script.

Note

The software for copying the data section from ROM to RAM is located in the .boot section (crt0ram.S). If the software developer is creating an application executable and corresponding custom linker script that contain any sections other than the default sections (.boot, .text, .data, .rodata, and .bss), the developer must modify crt0ram.S in addition to providing a custom linker script.

Software Deployment

Software deployment involves placing initial code in non-volatile memory so that a LatticeMico32 microprocessor, on FPGA configuration, can access the code and execute it without user intervention. This initial code may be a full-blown application or a minimal amount of base code that activates another set of code resident in a non-volatile medium.

As noted in “Boot Sequence” on page 76, in the boot-up sequence after a reset, the LatticeMico32 microprocessor begins to fetch instructions from the address contained in its EBA register. The EBA register holds the exception base address where the exception handling code starts and where the microprocessor starts to fetch instructions from the reset exception.

To achieve deployment, the following two conditions must be met when the FPGA powers up:

- The EBA (exception base address) register must point to a valid address location.
- The address location pointed to by the EBA must contain the instructions that the microprocessor will execute after reset.

For deployment, you must not use a JTAG UART. If your code uses standard C file operations, such as printf, scanf, or fopen, your deployed code will not work if it uses a JTAG UART as a standard I/O device or for file operations. You can use the RS-232 UART for standard I/O operations.

The LatticeMico C/C++ SPE provides sample deployment support for booting from the CFI parallel flash device on the LatticeMico32 development board, as well as booting from an on-chip memory component. The GNU objdump utility customized for LatticeMico32, lm32-elf-objdump, can provide significant insight into your code usage.

Deployment Strategies

Following are some simple deployment strategies that are used to manage the location of the various sections of a LatticeMico executable.

- On-chip memory deployment

  The LatticeMico On-chip Memory Controller is designed as a read/write memory component. However, it is unique in the sense that it can be initialized as part of the FPGA configuration.

  The application can be built as part of a managed build to contain all its sections within the on-chip memory (through the managed-build process) and can be deployed by converting the application contents to binary format that can be used for configuring the on-chip memory components as part of FPGA configuration.

- Multiple on-chip memory deployment

  Multiple on-chip memory deployment can be used for applications that contain multiple memory instances, including inline memories. A separate

memory initialization file is generated for each memory instance used by the software application.

- Parallel flash deployment

  The parallel flash deployment outlined in this document and demonstrated in the *LatticeMico32 Tutorial* adopts the strategy in which the “main” application is built for running from volatile memory (external volatile memory or on-chip memory) but is stored in external non-volatile memory. This “main” application’s run-time sections are then copied at run time by a boot loader from non-volatile memory to volatile memory. This strategy allows the "main" application to have all types of sections enumerated in “Types of Sections,” including a non-zero initialized read/write data section.

- SPI flash deployment

  With the inclusion of the SPI flash component in the LatticeMico MSB, the LatticeMico32 microprocessor can boot from a SPI flash device that is connected to a SPI flash component. From a software standpoint, this deployment strategy is identical to the parallel flash deployment in regard to the linker sections and the need for a boot-loader. Depending on the FPGA and the board layout, this deployment strategy allows the FPGA bitstream to coexist with the microprocessor application binary.

Deploying to On-Chip Memory

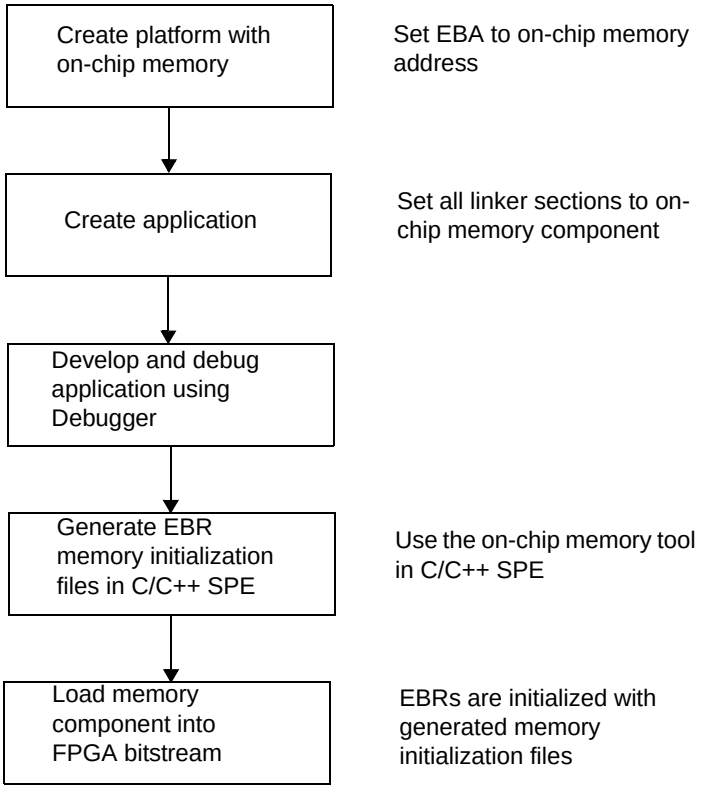
You can deploy your application to a LatticeMico on-chip memory controller if your platform contains sufficient on-chip memory in this component. After the microprocessor is configured, it can fetch instructions from the controller’s address.

When deploying to on-chip memory, consider the following:

- If you intend to run your application from on-chip memory, avoid using microprocessor caches. Microprocessor cache implementation uses EBR memory, so it reduces the amount of EBR memory available for the LatticeMico on-chip memory controller. Refer to the LatticeMico32 microprocessor data sheet for an estimate on microprocessor EBR usage if you are using debug modules.
- Do not call standard C library functions that can significantly expand your application, such as printf or malloc function calls.
- Avoid unrolling function loops and inline function calls, because these can increase the code size.
- Make sure that you have adequate on-chip memory for your application’s stack space needs.

Figure 131 depicts the steps involved for achieving on-chip memory deployment.

Figure 131: On-Chip Memory Deployment Flow



To generate a platform that can be used for software application deployment to on-chip memory:

1. In MSB, create a platform with the minimum required connectivity and lock down the addresses.
2. In the MSB perspective, select a microprocessor reset address.
3. In Lattice Diamond, generate an FPGA bitstream.
4. Using C/C++ SPE and the Debugger, develop your application and debug it.
5. Using C/C++ SPE and the Debugger, generate the memory file for on-chip memory.
6. In Diamond, initialize the memory component.

Establishing Minimal Platform Connectivity

For a platform to support on-chip memory deployment, it must contain an on-chip memory component that is accessible by the microprocessor's instruction and data ports. These port connections allow the microprocessor to fetch instructions from the on-chip memory component and to store and retrieve program data.

To add the on-chip memory component and connect it to your platform, see the following sections in the *LatticeMico System Hardware User Guide*: "Adding Microprocessor and Peripherals to Your Platform" and "Connecting Master and Slave Ports."

Figure 132 shows the minimal platform configuration required for on-chip memory deployment in the Editor view in the MSB perspective. The on-chip memory component is named ProgramMemory in this example. The on-chip memory component uses EBR blocks, so the number of EBR blocks consumed by other components must be taken into consideration when you determine the memory size. Synthesis will fail if the total EBR usage for the platform exceeds the available FPGA EBR blocks. The microprocessor needs EBR blocks for implementation, and the amount that it requires depends on the microprocessor configuration.

Figure 132: Minimal Platform for On-Chip Memory Deployment

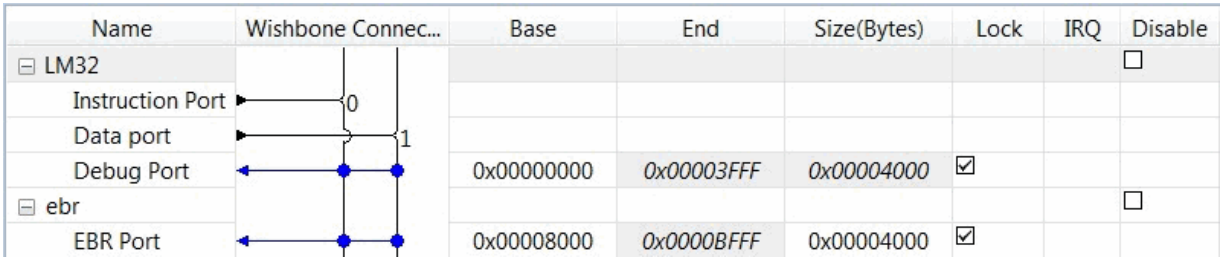


Figure 133 gives example on-chip memory configuration settings in the Add On-Chip Memory dialog box. You will not enter the name of a memory file, since you will use the ECO Editor in Diamond to initialize this on-chip memory

component. You can access this dialog box by double-clicking on On-Chip Memory in the Editor view.

Figure 133: Add On-Chip Memory Dialog Box

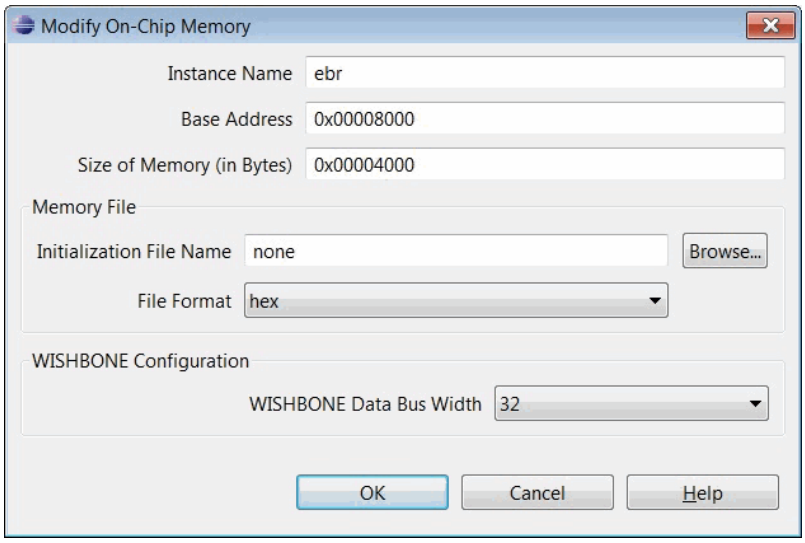
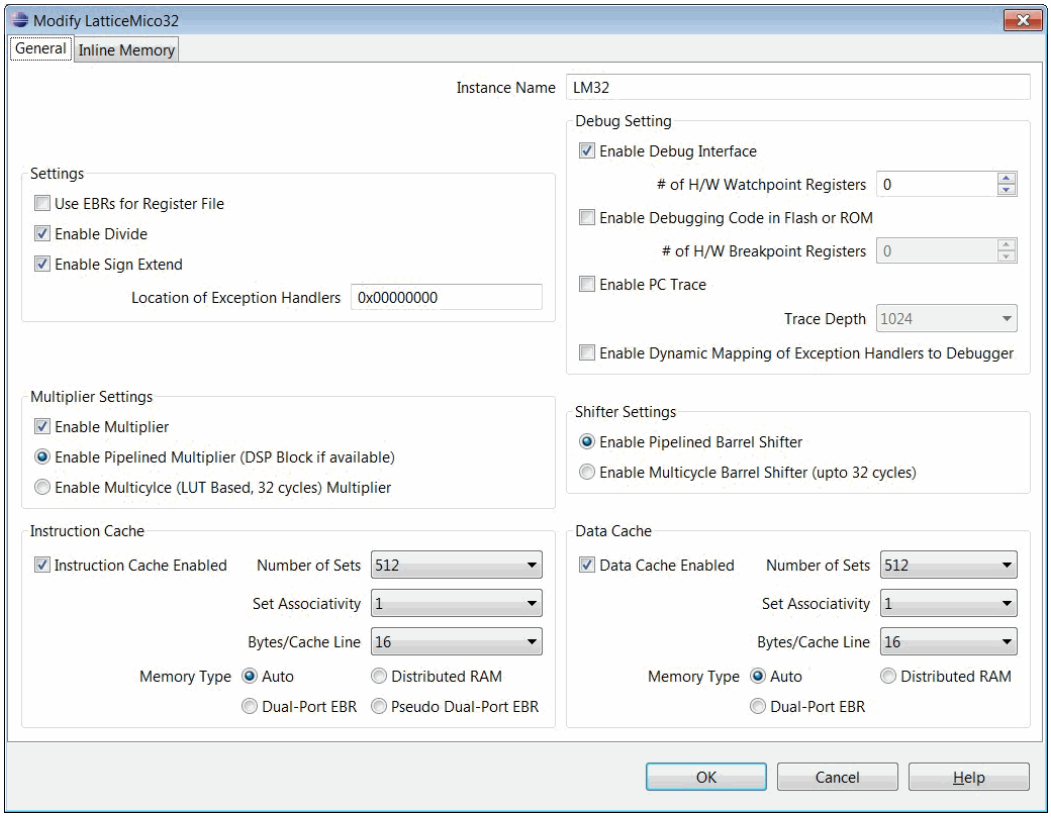


Figure 134 shows a microprocessor configuration that attempts to minimize EBR usage while still using caches. Disabling caches makes additional EBR blocks available. Refer to the LatticeMico32 microprocessor data sheet on the

Lattice Semiconductor Web site so that you can roughly estimate the EBR usage in your design with regard to configuration settings.

Figure 134: Microprocessor Configuration for Minimal EBR Usage with Caches



Generating the Platform and Lock Addresses

After you select the necessary components, you must assign the platform component addresses and interrupt priorities before you generate the platform. Once you assign these addresses, you must lock them by following the procedure in “Locking Component Addresses” in the *LatticeMico System Hardware User Guide*.

Locking the address of a component in MSB prevents that component's address from being changed when the platform is regenerated. Selecting the microprocessor reset address and initializing the memory component cause the platform to be regenerated. As an example, the final code is built to specific component addresses. Initializing the memory component is required to put the final code into the FPGA's configuration bitstream. The components' addresses cannot change during that process or the final code will not match up properly with what is defined in the platform.

See the sections “Selecting the Microprocessor Reset Address” and “Initializing the Memory Component” in the *LatticeMico System Hardware User Guide*.

The software application based on this platform, once written and tested, has address references to the components that it uses. You must lock these addresses to prevent them from being used in the software that will be deployed in the on-chip memory.

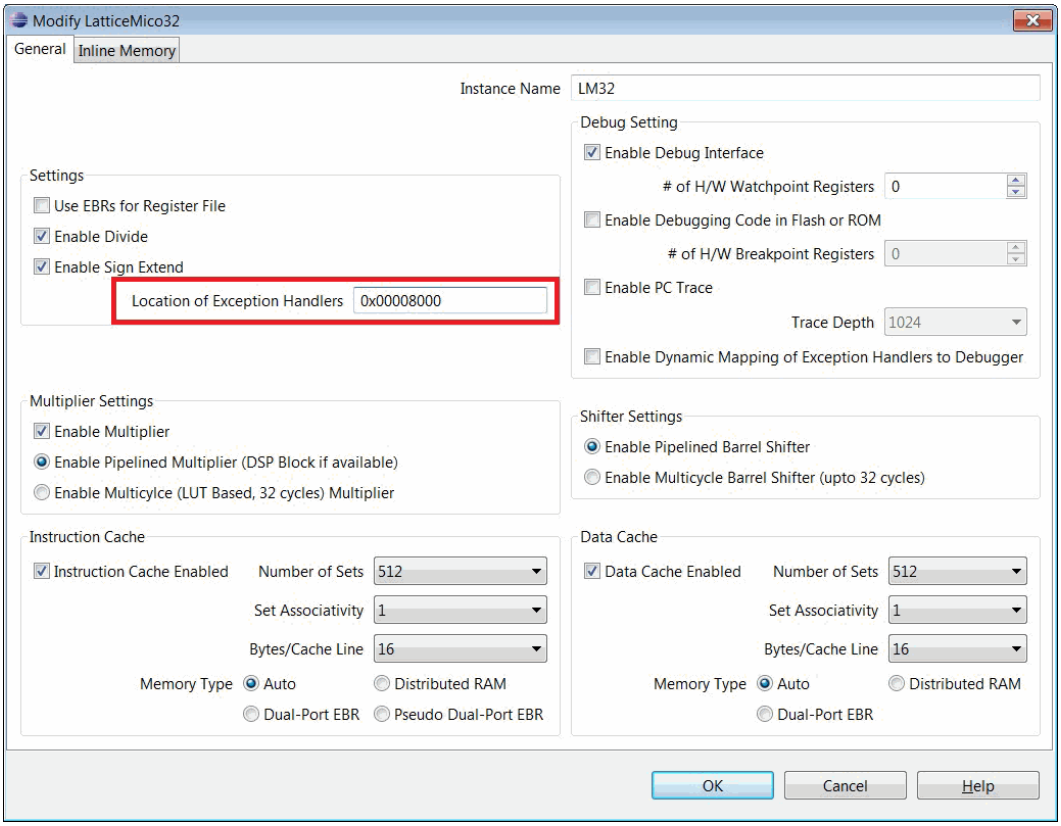
Selecting the Microprocessor Reset Address

After the component addresses are assigned and locked down, the microprocessor reset address must be modified so that it points to on-chip memory. On power-up, the microprocessor starts executing instructions from its reset address. You define the reset address during microprocessor configuration in MSB by using the Location of the Exception Handlers option in the Add LatticeMico32 dialog box. This address must be aligned to a 256-byte boundary, since the hardware ignores the least-significant byte. Unpredictable behavior occurs when the exception base address and the exception vectors are not aligned on a 256-byte boundary. The Location of the Exception Handlers value is also the reset address, since the first exception handler is the reset exception. When you set this reset address to the on-chip memory, the microprocessor starts executing instructions from the on-chip memory.

To set the reset address:

1. In the MSB perspective's Editor view, double-click the microprocessor instance in the platform to open the Add LatticeMico32 dialog box.
2. In the Settings tab, change the value in the Location of Exception Handlers text box to the address of the on-chip memory. Figure 135 on page 185 shows a sample Add LatticeMico32 dialog box that shows this text box in the dialog box with a sample reset address value.

Figure 135: Modifying the Reset Address Location



3. In the MSB perspective, make sure that addresses are locked, and then choose **Platform Tools > Run Generator** to regenerate your platform. Since the addresses are locked, they will not change after platform regeneration.

Generating the FPGA Bitstream

After generating the platform, go back into Diamond to generate the FPGA configuration bitstream. This bitstream contains the on-chip memory component, which will be initialized with the application binary generated in step 6 when the FPGA powers up and is configured.

As part of step 2, the microprocessor reset address was set to the location of the on-chip memory component. When the FPGA is configured after power-up, the LatticeMico32 microprocessor executes instructions from the on-chip memory component, which now contains the application code. The product is deployed in the FPGA bitstream.

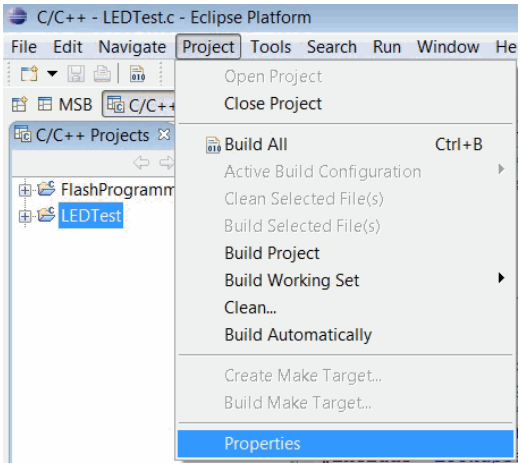
Developing and Debugging the Application

Once the platform and the bitstream are generated, you can develop and debug the application that will be deployed in the on-chip memory. To do this, it is essential that the application be built for running from the on-chip memory component.

To set up the application to run from the on-chip memory:

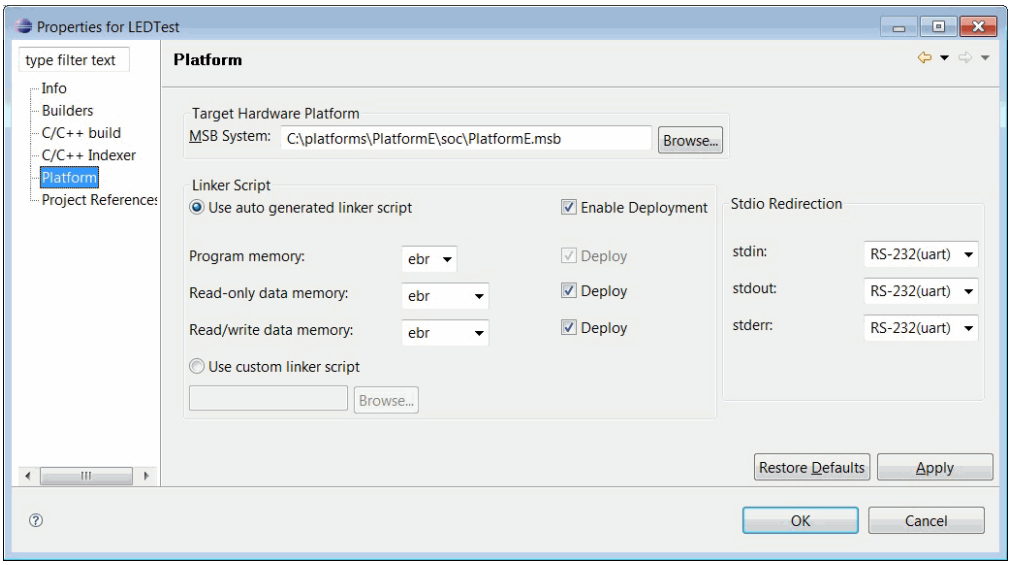
1. In the C/C++ perspective's Projects view, right-click on your project icon and choose **Properties** from the pop-up menu. You can also select the **Project > Properties** menu command, as shown in Figure 136.

Figure 136: Project Property Menu Option



2. In the Properties dialog box, select the **Platform** tab, as shown in Figure 137.

Figure 137: Platform Tab of the Properties Dialog Box



3. In the **Linker Script** field, select the **ebr** on-chip memory type in the three text box dropdown menus, since this is the type of memory being used. In Figure 137, “ProgramMemory” appears in place of “ebr,” since the on-chip memory component was named ProgramMemory.

   Selecting this linker option locates the program it generates in the selected on-chip memory component.

4. In the Stdio Redirection field, select **RS232 (UART)**, not JTAG UART (LM32), if your application uses standard C file operations, such as printf, scanf, and fopen.

5. Click **Apply** and **OK** to implement your changes.

You can now build and debug the application, using the on-chip memory component. If the amount of memory available is less than that required for the application, the linker will generate error messages indicating there is insufficient memory, in addition to other error messages.

Generating the Memory Initialization File

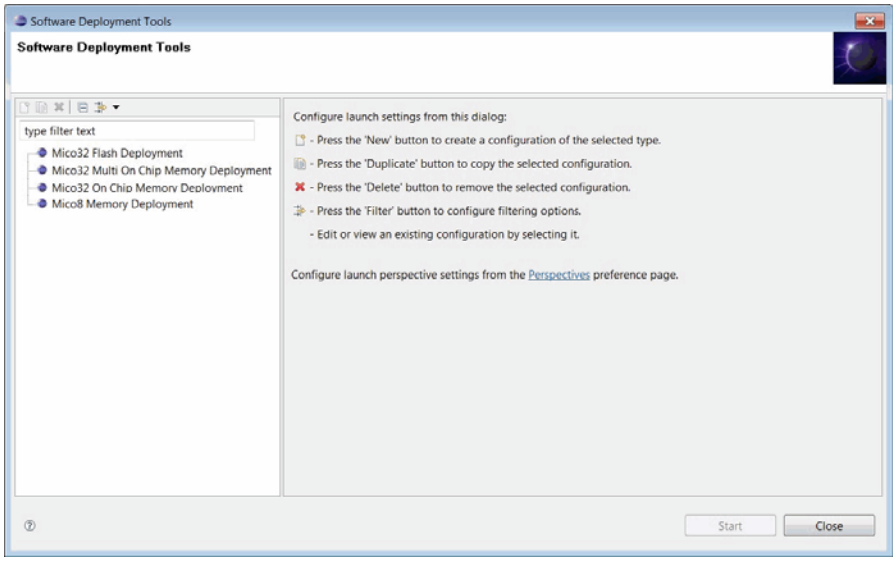
Once the application is debugged and ready for deployment, it must be converted into an EBR memory initialization file.

To generate the EBR memory initialization file:

1. In the C/C++ perspective, click **Tools > Software Deployment**.

   The Software Deployment Tools dialog box appears, as shown in Figure 138.

Figure 138: Software Deployment Tools Dialog Box



2. Select **On Chip Memory Deployment** located at left in the list box, and click the “New launch configuration” button on the toolbar.

The dialog box changes to show the Main tab and a new configuration perspective on the right side of the dialog box.

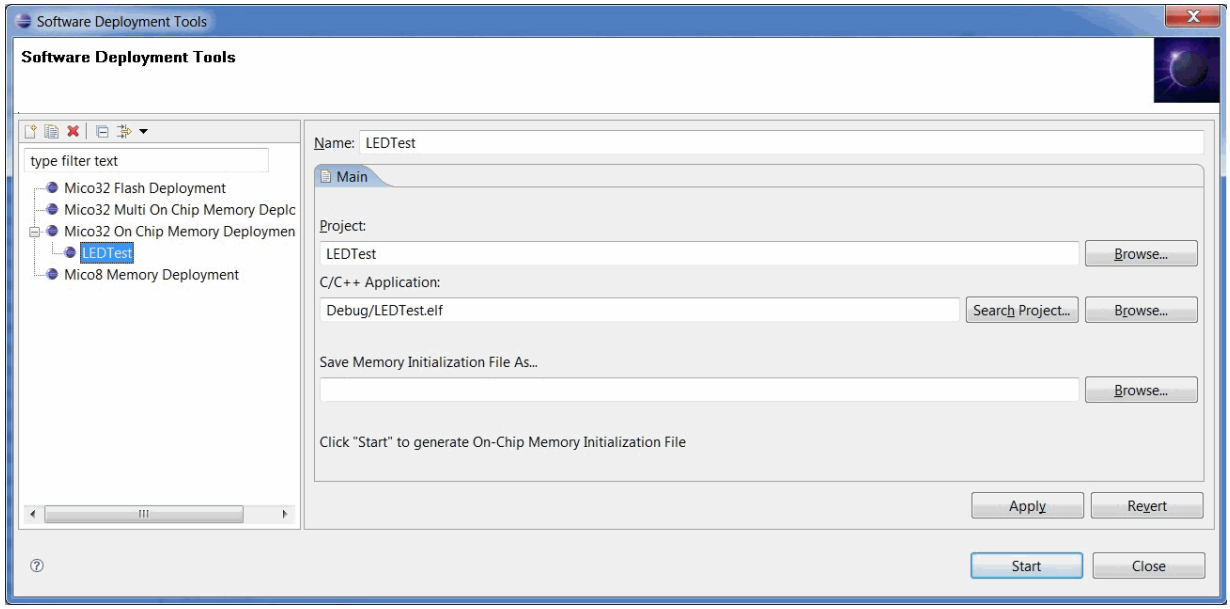
3. Enter the name of the configuration, as shown in Figure 139.
4. Select the application by clicking the **Browse** button.

   The Browse dialog box lists all available applications that were created. Be sure to select the appropriate application for on-chip memory deployment.
5. Select the appropriate C/C++ application by clicking **Search Project**.

   The pop-up dialog box shows all available executables created for the selected project. Be sure to select the appropriate executable.
6. Enter a name for the generated memory initialization file in the “Save Memory Initialization As**”** box.

   The dialog box should resemble the illustration in Figure 139.

Figure 139: Completed Main Tab of the Software Deployment Tools Dialog Box



7. Click **Apply** and **Start**.

   A file is now generated in the C/C++ application folder. The name of this file is what you entered in the Mem Ini File Name option text box. You can now use this memory file for initializing the on-chip memory platform component.

When you run the software, the underlying programmer utility that generates the on-chip memory initialization file executes the following two commands in order:

```
lm32-elf-objcopy.exe -O binary <ApplicationElfFile>
<Temporary.bin file>
```

```
bin_to_verilog --EB --width 4 <Temporary.binfile> >
<MemoryInitializationFile.mem>
```

For example, in the following code:

```
lm32-elf-objcopy.exe -O binary mytest.elf mytest.bin
bin_to_verilog --EB --width 4 mytest.bin > mytest.mem
```

- Mytest.elf is the generated application that must be deployed in on-chip memory.
- Mytest.bin is a temporary binary file that is generated by lm32-elf-objcopy.exe.
- Mytest.mem is the memory initialization file generated by bin_to_verilog.

Initializing the Memory Component

Now you load the memory initialization file into a placed and routed FPGA bitstream.

To implement the .mem file in a Lattice Diamond design:

1. In Diamond, choose **Tools > ECO Editor** or click  .
2. Click the **Memory Initialization** tab at the bottom of the ECO Editor window. The Memory Initialization window opens.
3. Right-click on **{*OCM*}/ram**, where {*OCM*} is the name of the on-chip memory component from the LatticeMico platform, and choose **Update Initial Memory** from the pop-up menu. You may also choose **Edit > Update Initial Memory**. The Update Initial Memory dialog box opens.
4. In the **File Format** drop-down menu, ensure that Memory Format is set to **Hexadecimal**.
5. In the Memory File box, browse to and select the on-chip memory initialization (.mem) file created by LatticeMico.
6. Click **Update**. If sucessful, an Update Memory Initialization Succeeded dialog box appears. Click **OK**.
7. Choose **File > Save (*file_name*).ncd**, or click  to save the .ncd file.
8. In the Process view, double-click **Bitstream File**.

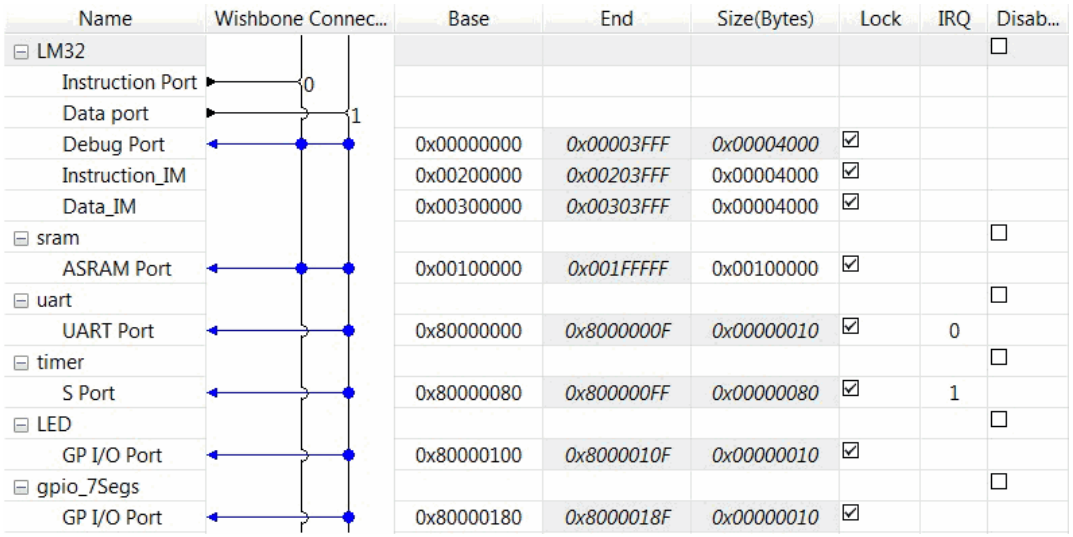
As an alternative to this procedure, you can use Add On-Chip Memory dialog box shown in “Establishing Minimal Platform Connectivity” on page 181 to initialize the memory component. However, when you use that method, you must regenerate the platform and perform all the steps again in the Diamond flow.

Deploying to Multiple On-Chip Memory

Multiple on-chip memory deployment generates a memory initialization file for each instance of the memory controllers that use the FPGAs on-chip memory resources. Examples include On-Chip Memory Controller, On-Chip Dual-Port Memory Controller, Instruction Inline Memory, and Data Inline Memory..

The example platform in Figure 140 has three memory controllers: Async SRAM, Instruction Inline Memory, and Data Inline Memory. The software application that runs on this platform uses both Inline memories. By using Multi On-Chip Memory Deployment, the software developer can create initialization files for both the inline memories that contain all the relevant sections of the software application mapped to them.

Figure 140: Example Platform for Multiple On-Chip Memory Deployment



For more information about using inline memories, refer to the “Memory Architecture” section of the *LatticeMico32 Processor Reference Manual*.

Setting up the Application

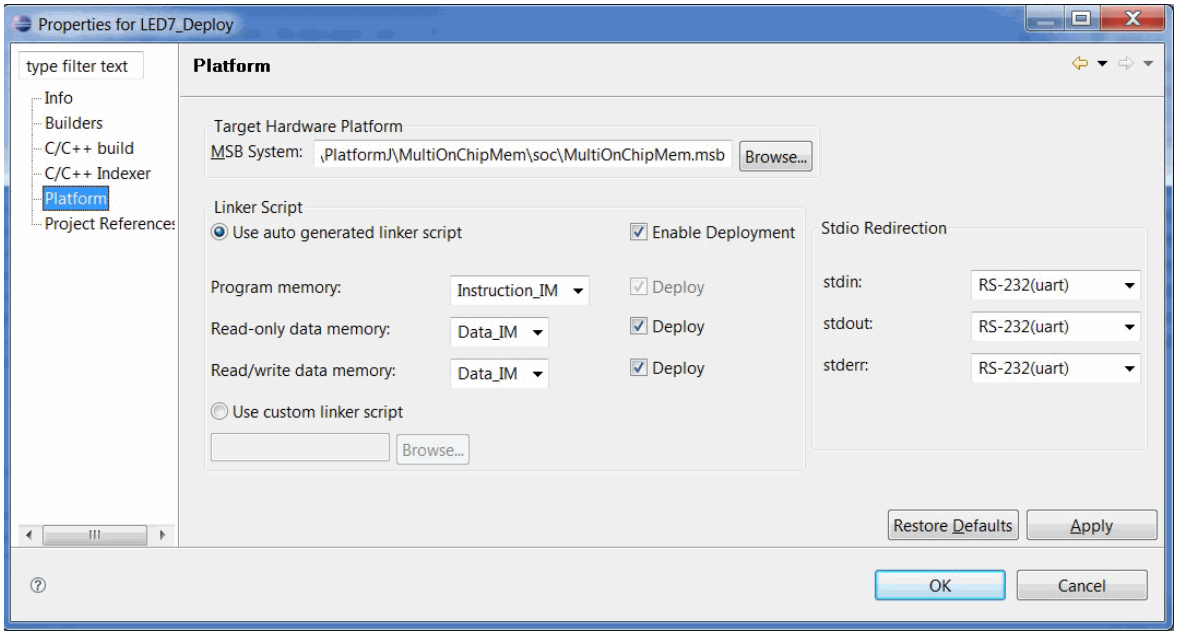
After generating the platform and bitstream, you can set the properties for the application that will be deployed to multiple on-chip memory.

To set up the application:

1. In the C/C++ perspective's Project view, right-click the name of your project and choose **Properties** from the pop-up menu.
2. In the Properties dialog box, select **Platform**.
3. In the Linker Script section, select the appropriate on-chip memory type from the three drop-down menus.

   The example in Figure 141 shows the settings for using the inline memories with the auto generated linker script. Instruction_IM and Data_IM are the default instance names that were used in the Inline Memory tab when the LM32 processor was added to the platform.

Figure 141: Platform Section with Linker Script for Inline Memory



4. Clik **OK** to implement your changes.

You can now build and debug the application, using on-chip memory. If the amount of memory available is less than that required for the application, the linker will generate error messages indicating that there is insufficient memory.

Generating the Memory Initialization Files

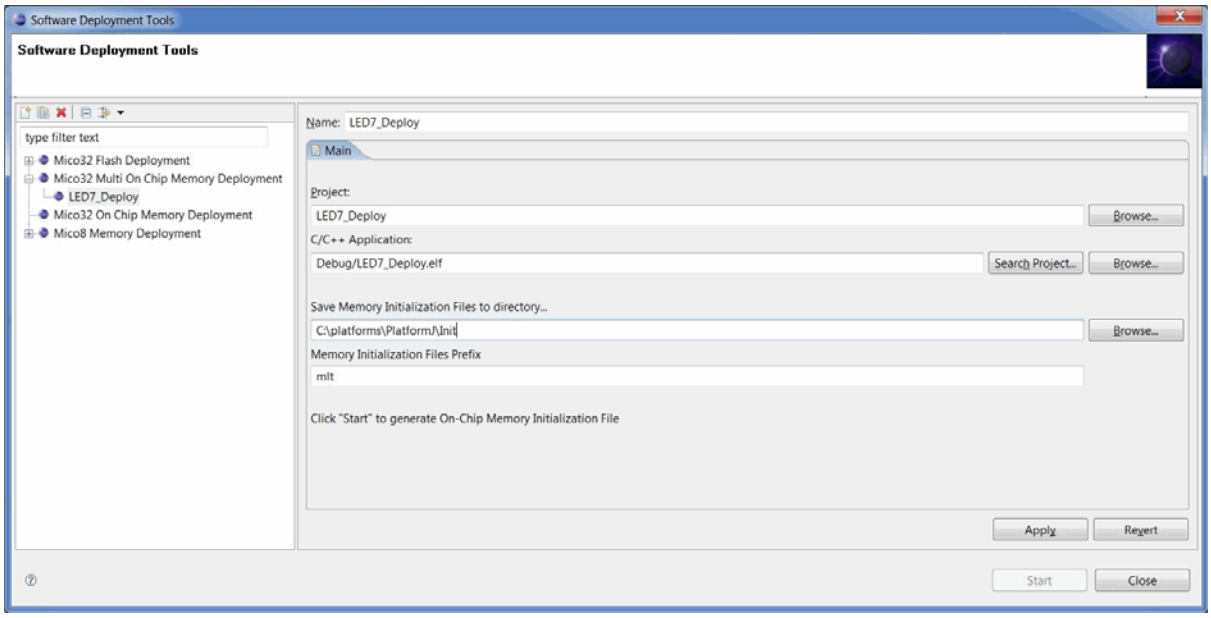
Once the application is debugged and ready for deployment, the on-chip memory initialization files can be created.

To generate the memory initialization files:

1. In the C/C++ perspective, click **Tools > Software Deployment**.
2. In the Software Deployment Tools dialog box, select **Multi On Chip Memory Deployment** from the list on the left, and click the “New launch configuration” button on the toolbar.

   The dialog box changes to show the Main tab and a new configuration perspective on the right, as shown in Figure 142.

Figure 142: Multi On Chip Memory Deployment Dialog Box



3. Enter a name for the configuration.
4. Enter the project name by clicking the **Browse** button. Select the project from the list in the Project Selection dialog box.
5. Click **Search Project** and select the appropriate executable from the list in the Program Selection dialog box.
6. Specify a directory for saving the generated memory initialization files.
7. Specify a prefix that will identify each of the generated files.
8. Click **Start** to generate the memory initialization files.

   A file is generated for every memory instance that is used by the software application.

   For the multiple on-chip memory example shown in Figure 142, the “init” directory now contains separate memory files for instruction inline memory and data inline memory. The name of each of these memory files includes the prefix “mlt.”

Deploying to a Flash Device

This section describes the steps required for deploying to a flash device, followed by an introduction to the LatticeMico flash programming utility included in LatticeMico C/C++ SPE, which automates the required steps. It is recommended that you also refer to the *LatticeMico32 Tutorial*, which provides step-by-step instructions for flash deployment.

Note

For flash deployment, the software developer must ensure that no file operations, such as C/C++, printf, file open/close, etc., are done via the JTAG UART. The JTAG UART requires a live connection to the Debugger on the PC, which is not available when an application is deployed. All I/O operations must be redirected to the RS-232 UART instead..

Flash Deployment Steps

Flash memory is a non-volatile storage memory. While read accesses to a flash device do not require any special setup from a software perspective, erase and write operations require special software sequencing. So, for all practical purposes in regard to microprocessor boot-up, this storage can be treated as read-only, non-volatile memory.

For our discussion, a memory requiring special software code to perform write operations, such as a flash memory, is called a read-only memory. A memory component that does not require special software code to perform read/write operations, such as an on-chip memory component or an ASRAM component, is called a read/write memory.

Deploying the application to flash memory involves the following steps:

1. Configuring the microprocessor to boot from flash. This means configuring the Exception Base Address (EBA) of the microprocessor to the location in flash where the deployed application will reside.
2. Creating the “main” application binary image that will be written to flash via software or Diamond Programmer.
3. Strapping a boot copier to the “main” application binary image. The boot copier copies the application binary image to the target volatile memory for execution.

   Note

   The boot copier is needed only if the “main” application’s binary image from Step 2 is created with a version of LatticeMico System Builder prior to Version 8.0. Starting with Version 8.0, the boot copier is integrated into the applicatin’s binary image created in Step 2.

   From this point forward, the boot copier of Step 2 will be referred to as “integrated” boot copier and the boot copier of Step 3 will be referred to as “stand-alone” boot copier.

4. Programming the application image from Step 2 and optional “stand-alone” boot copier from Step 3 to flash.

Note

Refer to the instructions in the *LatticeMico32 Tutorial* for step-by-step procedures.

As noted at the beginning of this chapter, the program sections that are only read at run time can reside in non-volatile memory permanently. But sections that can be written to at run time must be moved to volatile memory before any write operations.

The typical strategy for flash deployment involves storing the “main” application in a non-volatile, read-only memory. The “integrated” or “stand-alone” boot loader is also stored in the non-volatile memory. On microprocessor reset, this boot loader is the first piece of code that the microprocessor executes. The boot loader copies the stored “main” application from flash memory to the appropriate read/write memory location that is specified when the “main” application executable is generated. Afterwards, it branches to the copied code so that the processor can start executing the “main” application from volatile memory.

The following points in the subsequent sections explain the four-step process demonstrated in the *LatticeMico System Tutorial*.

Configuring the Microprocessor to Boot from Flash

The microprocessor’s EBA must be configured to point to a valid flash address that will contain the deployed boot copier and “main” application. The “Location of Exception Handler” address in the MSB tab should be the same as the “Reset Vector Address (EBA Value)” located in the C/C++ tab of the Software Deployment Tools dialog box. This configuration ensures that the boot copier is the first piece of code that is executed after microprocessor reset.

Creating a “Main” Application Binary Image

The application executable that will be deployed to flash memory and executed from volatile memory is created by setting the program’s various sections to use volatile read/write memory for execution; and by enabling the deployment option for the program, read-only data, and read/write data memory sections. These settings are enabled in the Platform tab of the Properties dialog box. See Figure 14 on page 29 for details. Once the application executable is created, it is converted to binary format. An ELF-format application includes sections of data that form part of the executable image and other sections that contain information needed only by the Debugger and not required as part of the executable image. Section data that contributes to the executable image of the application must be extracted from the application .elf file and put into the application binary image information.

The LatticeMico System software includes a utility, elf2data, which extracts the necessary section data and stores it as a LatticeMico assembly language file. This utility is located in the file path <*install_dir*>\micosystem\utilities. The elf2data utility is essentially an .elf file parser that generates an assembly file.

The assembly file that elf2data generates contains the following information as constants:

- A first 32-bit entry indicating the _reset_vector location of the application being copied. Labeled entry_address, it is referenced by the copier code to identify the start of the binary image of the application that is being copied.
- A data section that contains a piece of the application binary image. The contents of this data section go into sequential memory locations.
- Other data sections, if the application that is being copied has sections dispersed across different memories. The contents of each data section go into sequential memory locations.

Each data section generated by the elf2data utility has a 32-bit value indicating the length, in bytes, of the data contained in that particular section. This is followed by the data bytes. For multiple data sections, the sections appear sequentially. The end of these data sections is indicated by a terminating data section containing a length value of 0XFFFFFFFF.

To invoke elf2data, use the following syntax:

```
elf2data <elf_file> BinCopier.S <flash start address> <flash
end address>
```

Then create an ELF executable for the assembly file “Bin Copier.S”, generated by the elf2data utility, using the following command:

```
lm32-elf-ld BinCopier.o --section-start .text= ".$BootAddress"
-o BinCopier.elf
```

where ".$BootAddress" is the location in non-volatile memory where the application resides. This address must match the microprocessor’s EBA

value, since this is the address from which the microprocessor will start fetching instructions on power-up/reset.

Note

If the “main” application was compiled using a LatticeMico System Builder version prior to 8.0, use the following sequence of commands.

```
elf2data <elf_file> BinData.S
```

The above command will create an assembly file of the “main” application’s ELF executable, which will contain all the sections that are deployed to non-volatile memory. This assembly file is merged with the “stand-alone” boot copier using the following command:

```
lm32-elf-gcc -c BinCopier.S
```

Finally, an ELF executable is created that contains the merged “stand-alone” boot copier and “main” application.

```
lm32-elf-ld BinCopier.o --section-start .text=".$BootAddress"
-o BinCopier.elf
```

This executable is then converted into raw binary format for programming to the flash device. The command used for doing this is as follows:

```
lm32-elf-objcopy -O binary BinCopier.elf flashprog.bin
```

The output is a raw binary format file, flashprog.bin, which can now be programmed to flash memory, using a flash programmer. This file contains the boot copier executable (stand-alone or integrated) as well as the application’s binary image stored as data, which can be copied by the boot copier to target memories from the non-volatile flash memory storage.

Note

The aforementioned commands are automatically generated and executed by the GUI-based “Flash Deployment” utility. The steps were primarily shown to explain the process of creating a flashable raw binary image of an application that needs to be deployed to flash.

Programming Image to Flash

Once the binary image containing the boot copier (stand-alone or integrated) and the “main” application is ready, this binary image must be programmed to the flash device, starting at the flash address pointed to by the microprocessor EBA register. Once this binary image is programmed, on reset, the LatticeMico32 microprocessor executes instructions, starting at the address pointed to by the EBA.

Since the programmed binary image is a binary image of the boot copier using the "main" application as its data, the LatticeMico32 microprocessor in effect executes the boot copier that, in turn, copies the application’s binary data to the appropriate target memory locations. Once the boot copier finishes copying the application’s binary data, it then performs a microprocessor branch to the starting address of the application executable. This branch starts the execution of the "main" application.

The C/C++ SPE provides, as an application template, a Flash Programmer template application.The CFI Flash Programmer template is used for programming to parallel flash. The SPI Flash Programmer template is used for programming to SPI flash. Like any other application, this application is downloaded onto the platform. This template application uses the CFI flash driver service. Since the platform used for the creation of the main deployable application contains a flash component, this very same platform is used for running the flash programming application. When it executes, this flash programmer application reads the binary data contained in a file on the computer hosting the LatticeMico C/C++ SPE and programs it to flash memory.

Caution!

The SPI Flash programmer template is a software-based method for deploying an application to SPI flash. Since this template performs erase and write operations in the SPI flash, the following conditions must be true:

1. The SPI flash component has a version 3.0 or greater.

2. The “Control Port” of the SPI flash is enabled. See the *SPI Flash User Guide* for information on how to enable the control port. Note that this port is disabled by default when the SPI flash component is instantiated within the design.

The “SPI Flash ROM” component that is available in Lattice Mico System Builder versions prior to 8.0 does not support erase and write operations. (This component is deprecated starting with Version 8.0 and is replaced with a “SPI Flash” component with a “Control Port” that allows erase and write operations to SPI Flash.) If the SPI flash cannot be erased or written to from software, use Diamond Programmer to deploy the application to SPI Flash instead. This is explained in the next section.

LatticeMico Flash Programming Utility

LatticeMico C/C++ SPE includes a flash programming utility in the user interface that automates the steps described in the prior section.

To use the Flash Programmer to program a flash memory:

1. In the C/C++ perspective, choose **Tools > Software Deployment**.

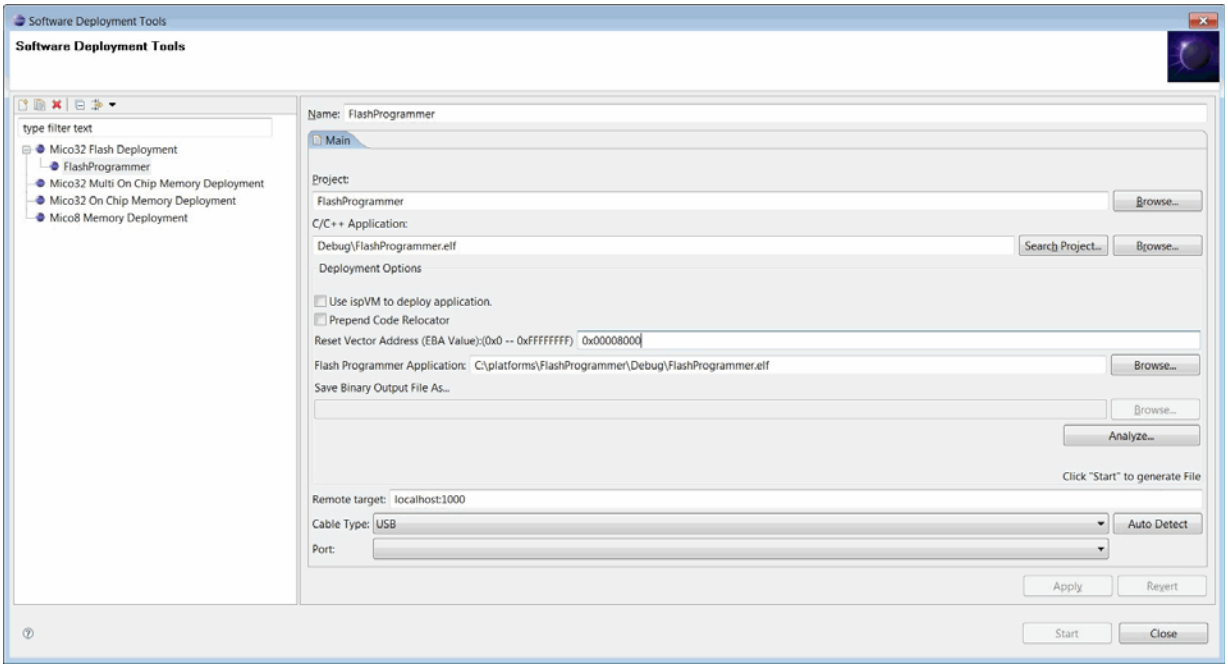
   The Software Deployment dialog box now appears, with the Software Deployment Tools screen selected.

2. Click on the **Flash Deployment** option in the Configurations list box at left.

3. Click the **New** button.

The Main tab of the Software Deployment Tools dialog box now appears, as shown in Figure 143.

Figure 143: Software Deployment Tools Flash Programmer Dialog Box



The Main tab contains the following options:

- Name

  Specifies the name of the current configuration.

- Project

  Specifies the C/C++ SPE project application to deploy to the parallel flash device.

- C/C++ Application

  Specifies the project application (.elf file). You can click the Search Project button to access a pop-up dialog box for selection of one of the available applications for the specified project.

- Prepend Code Relocator (for backward compatibility only)

  For projects compiled using a LatticeMico version prior to 8.0, enables the flash programmer utility to use the provided stand-alone boot copier and merge the application binary image with the boot copier code. In these earlier versions, the code relocator was not built into the application; therefore, it was necessary to prepend a separate relocator code to the actual application.

- Platform Reset Vector Address

  Specifies the target flash address to which to deploy the selected application. The microprocessor's exception base address (EBA) value must correspond to this address.

- Flash Programmer Application

  Specifies the flash programmer application that will be used for programming the application to flash. LatticeMico C/C++ SPE includes a flash programmer application template as a reference design for use.

4. Select the desired options and click **Apply**.
5. Click the **Start** button.

Deploying to SPI Flash Using Deployment Tool

The SPI flash ROM component included in LatticeMico MSB interfaces with an external SPI flash module. It translates WISHBONE read requests to the appropriate SPI commands to read data from the external SPI flash module and presents the read data to the WISHBONE data bus. This process allows the LatticeMico32 microprocessor and other masters to treat the external SPI flash module as a plain read-only memory.

The main advantage of SPI flash deployment is that it allows the FPGA bitstream (or portions of it) and the microprocessor bitstream to co-exist in a single SPI flash device. However, this is possible only if the FPGA user logic can access the very same SPI flash device that was used for the FPGA configuration. SPI flash deployment may impose FPGA requirements, board layout requirements, or both, which must be considered before you design the hardware.

Note

For SPI flash deployment, you must not use a JTAG UART. If your code uses standard C file operations, such as printf, scanf, or fopen, your deployed code will not work if it uses a JTAG UART as a standard I/O device or for file operations. You can use the RS-232 UART for standard I/O operations.

This section uses the LatticeMico32 LatticeECP development board as reference hardware since its user logic can also access the configuration SPI flash in addition to the configuration logic. Although this section uses an example in which the entire FPGA bitstream is contained in a single SPI flash, the concepts and steps presented remain valid for other FPGA bitstream deployment scenarios, such as a dual boot. The amount of available space depends on the total SPI flash capacity and the FPGA bitstream size. The term “LatticeMico application” refers to the initial code executed by the LatticeMico32 microprocessor on removal of the reset signal.

Once you have a platform that includes a LatticeMico SPI flash ROM, follow these steps to deploy the SPI flash:

1. Select the appropriate LatticeMico32 microprocessor EBA value (Reset Exception Vector Address).
2. Generate a bootable LatticeMico application binary.

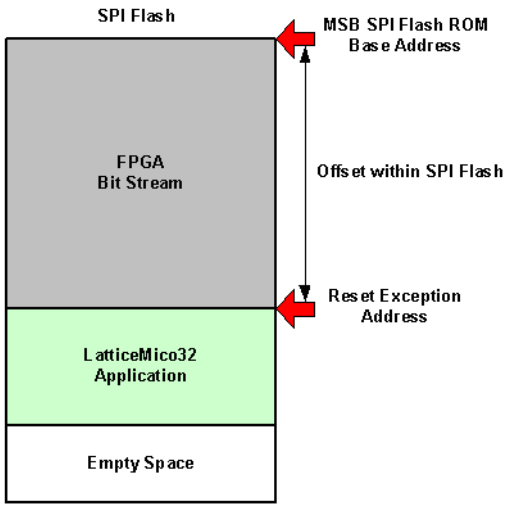
3. Merge the FPGA bitstream with the LatticeMico bootable application binary into a single SPI flash image.
4. Program the SPI flash with the SPI flash image. Make sure that your preference file has the correct SPI pin connections to the board.

The subsequent sections highlight these steps.

Selecting the Appropriate LatticeMico EBA Value

Figure 144 shows a sample layout in the SPI flash memory.

Figure 144: Sample Layout in SPI Flash Memory



In Figure 144, the first data portion is the FPGA bitstream that is used for configuring the FPGA. The second data portion is the LatticeMico application that is accessed by the LatticeMico32 microprocessor (part of the user logic) on removal of reset, once the FPGA is configured.

Reset Vector Address (EBA Value) To avoid repeatedly generating the FPGA bitstream each time that the EBA is modified, you must know the Reset Vector Address (EBA value) at the beginning of the process as part of configuring the microprocessor in the MSB. This value is the address from where the microprocessor starts fetching instructions on removal of reset. It is the sum of the LatticeMico SPI flash ROM base address assigned in the MSB perspective and the offset in the SPI flash where the LatticeMico boot application will reside. The offset depends on the FPGA bitstream size.

Offset Alignment in the SPI Flash The offset in the SPI flash must be aligned on a word boundary. It should be a multiple of 4 so that the lower two bits of the resulting EBA value are zero. The LatticeMico SPI flash and the LatticeMico32 microprocessor do not support aligned accesses, and all LatticeMico instructions are 32 bits, or 4 bytes.

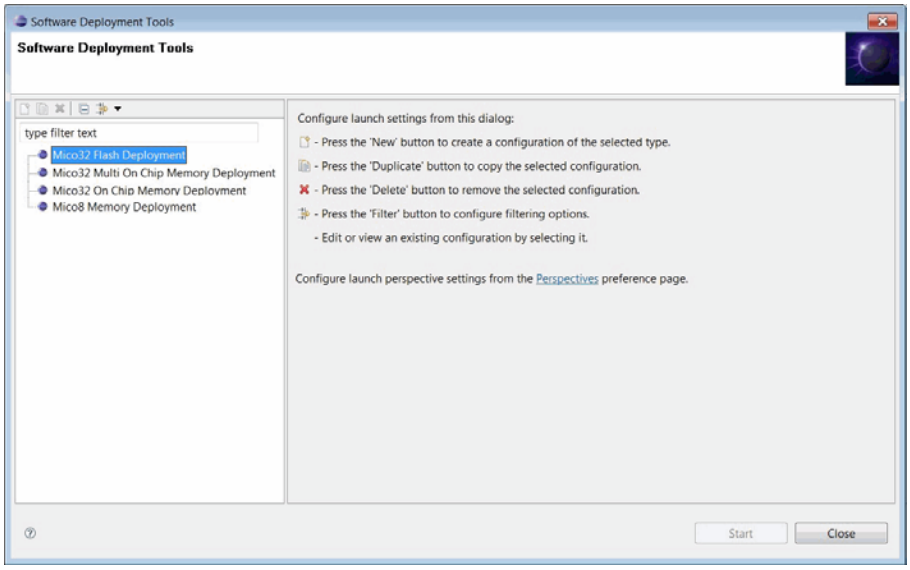
Generating LatticeMico Bootable Application Binary

Once the LatticeMico application is ready to be deployed, you must add a loader that can copy the application data to the appropriate target memories. The application data must be converted into binary format that can then be merged with the FPGA bitstream to form a SPI flash image. The LatticeMico C/C++ SPE perspective provides a graphical user interface for this purpose.

To generate a bootable application binary:

1. From the C/C++ SPE perspective, choose **Tools > Software Deployment** to activate the Software Deployment Tools dialog box, shown in Figure 145.

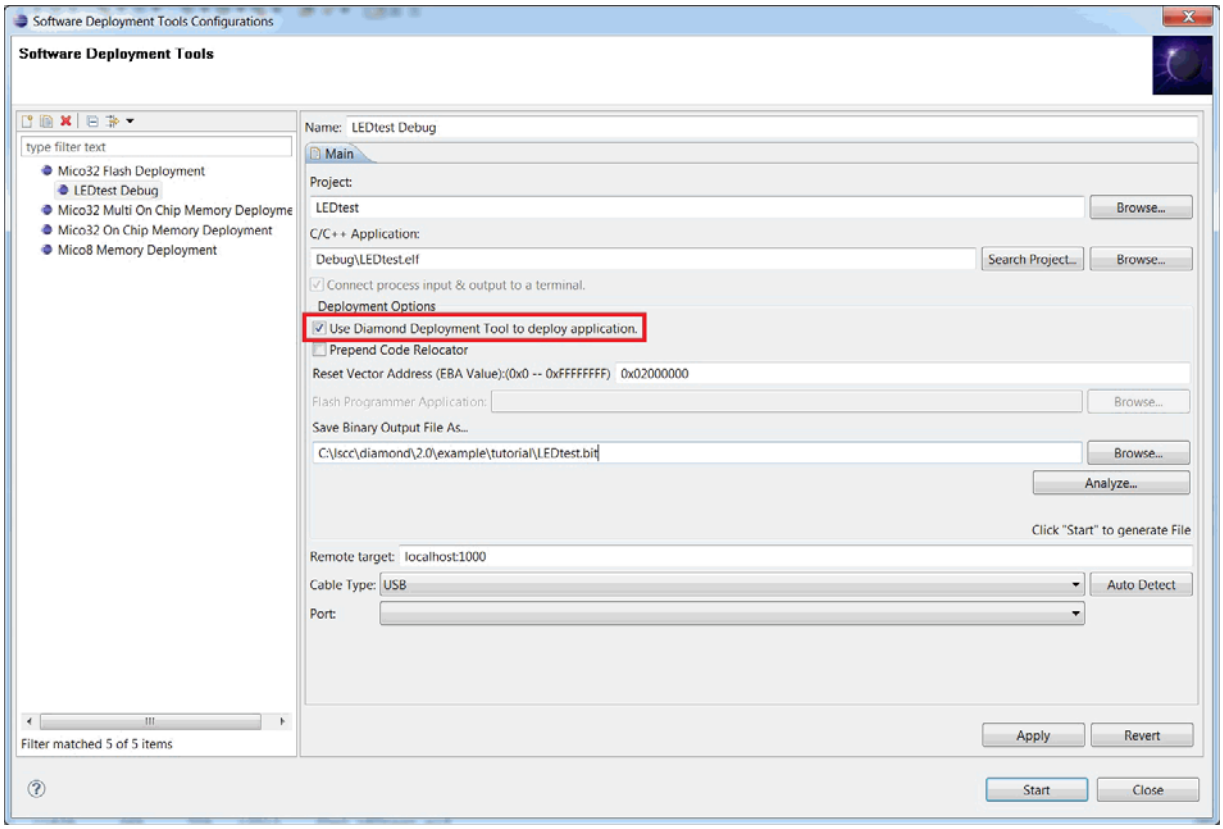
Figure 145: Software Deployment Tools Dialog Box



2. Select **Flash Deployment** from the list of configurations, and click **New**.

The main tab of the Software Deployment Tools Dialog now appears, as shown in Figure 146.

Figure 146: Main Tab of the Software Deployment Tools Dialog Box



The main tab consists of the following fields:

- Name – Specifies name of the current configuration.
- 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).
- Reset Vector Address (EBA Value) – Contains the EBA value chosen for the LatticeMico32 microprocessor, as described in “Reset Vector Address (EBA Value)” on page 200.
- Use Diamond Deployment Tool to deploy Application
- Prepend Code Relocator (for backward compatibility only) – For projects compiled using a LatticeMico version prior to 8.0, enables the flash programmer utility to use the provided boot copier and merge the application binary image with the boot copier code. In these earlier versions, the code relocator was not built into the application;

therefore, it was necessary to prepend a separate relocator code to the actual application.

- Save Binary Output File As – Selects the output file that will be generated by this tool. The output file must have a .bit extension. Click the **Browse** button to select the directory in which to generate the output file.

3. Apply the appropriate settings and click the **Start** button to generate the .bit file.

Merging the Bitstream and the Application Binary

Now you will merge the FPGA bitstream and the LatticeMico bootable application binary into a single SPI flash image.

Once the .bit file containing the bootable application binary is ready, you must program it into the SPI flash. If this application binary must co-exist with the FPGA bitstream (or a portion of it), it must be merged with the FPGA bitstream binary.

Deployment Tool is a convenient interface for performing this task. For detailed information on this tool, refer to Deployment Tool online Help.

The following steps use a sample FPGA bitstream, fpga.bit, generated by Lattice Diamond and a sample bootable application binary, mico32_sw.bit, to illustrate the procedure for merging these two FPGA bitstreams.

To merge the bitstream and the bootable application binary:

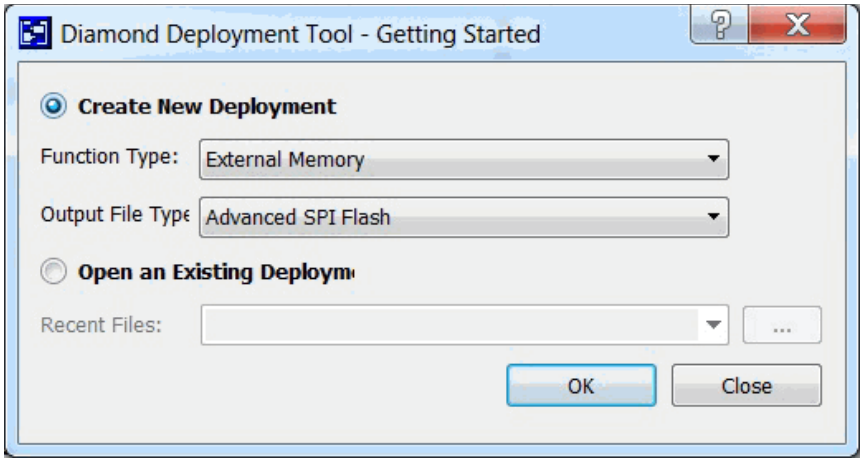
1. Launch Deployment Tool as follows:

   - 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`

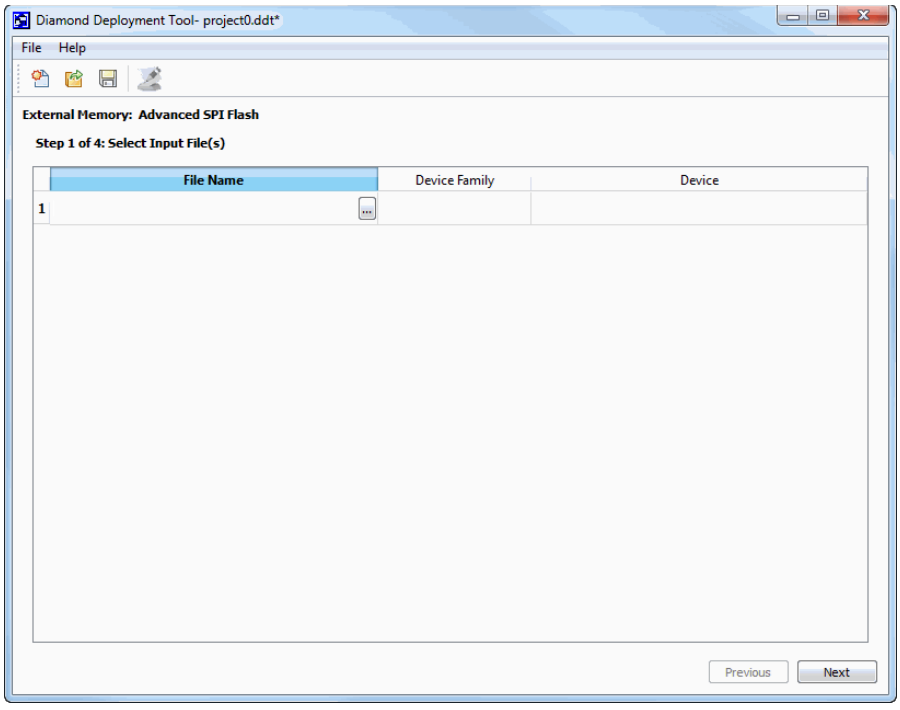
The Deployment Tool Getting Started dialog box appears, as shown in Figure 147.

Figure 147: Deployment Tool Getting Started Dialog Box



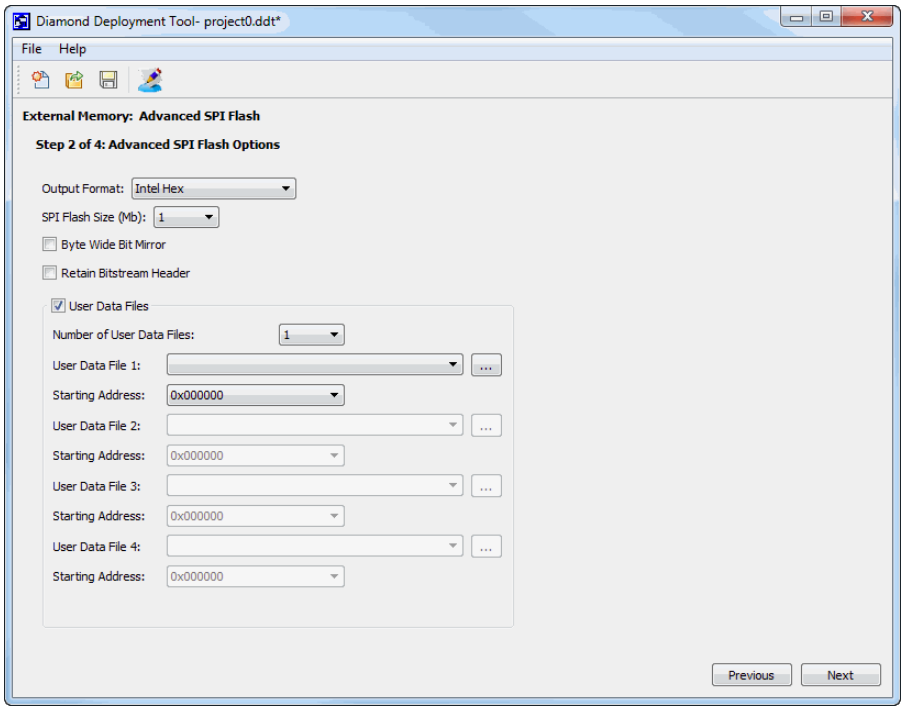
2. In the Function Type dropdown menu, choose **External Memory**.
3. In the Ouput File Type dropdown menu, choose **Advanced SPI Flash.**
4. Click **OK** to display the Step 1 of 4: Select Input File(s) dialog box, as shown in Figure 148.

Figure 148: Step 1 of 4: Select Input File(s) Dialog Box



5. Double-click the File Name box and browse to the FPGA bitstream named “fpga.bit” which targets a LatticeECP33E device.
6. Click **Next** to display the Step 2 of 4: Advanced SPI Flash Options dialog box, as shown in Figure 149.

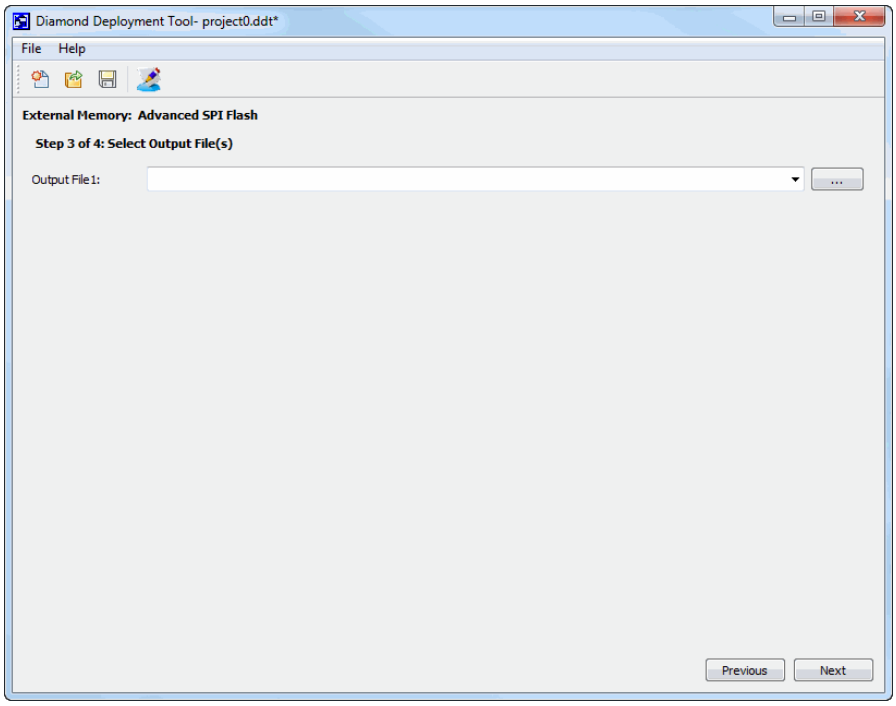
Figure 149: Step 2 of 4: Advanced SPI Flash Options Dialog Box



7. In Output Format dropdown menu, select **Intel Hex**.
8. In the SPI Flash Size (Mb) dropdown menu, choose **8**.
9. In the Number of User Data Files, dropdown menu, ensure that the number is **1**.
10. In the User Data File 1 box, click ... to browse to the application binary (.bit) file named mico32_sw.bit.
11. In the Starting Address dropdown menu, choose the starting address **0x0F0000**.

12. Click **Next** to display the Step 3 of 4: Select Output File(s) dialog box, as shown in Figure 150.

Figure 150: Step 3 of 4: Select Output File(s) Dialog Box



13. In the Output File 1 box, click ... to display the Select Output File dialog box. Name the output file data.mcs, and click **Save.**
14. Click **Next** to display display the Step 4 of 4: Advanced SPI Flash Options dialog box, and Click **Generate**.

    This generated file contains the merged FPGA bitstream and the LatticeMico bootable software application in a single SPI flash image file that Diamond Programmer can now use for programming the SPI flash.

Programming the SPI Flash with the SPI Flash Image

Diamond Programmer can use the merged SPI flash image file generated in the previous step for programming the SPI flash.

The following procedure uses the LatticeMico32 LatticeECP development board as an example and outlines the basic steps in programming the SPI flash.

Refer to the Diamond Programmer online Help for information on programming the SPI flash devices and additional details.

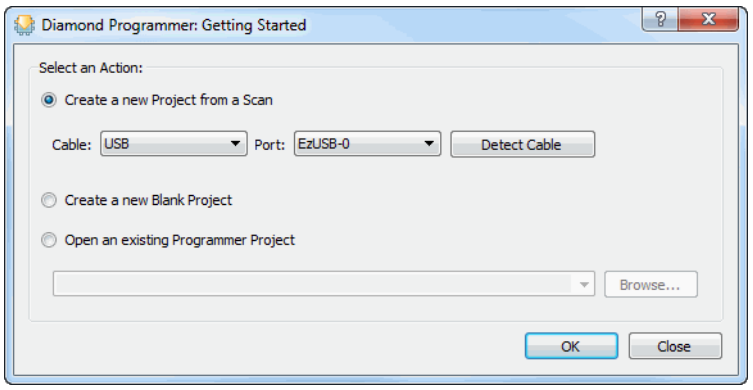
1. Launch Programmer as follows:

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

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

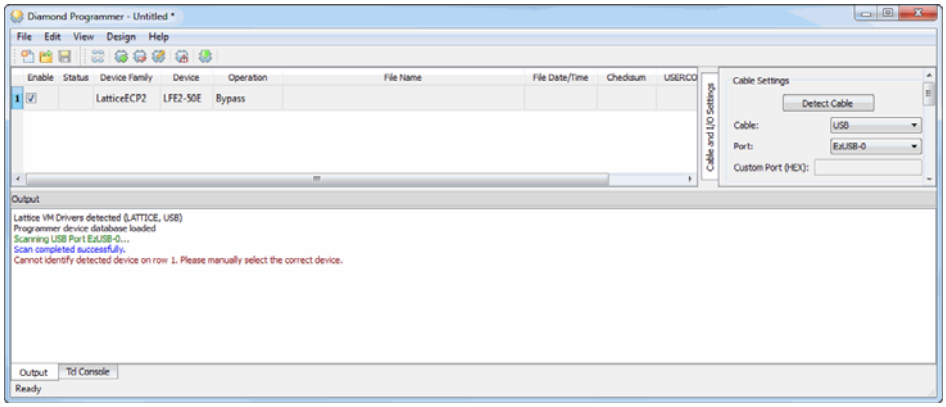
  The Diamond Programmer Getting Started dialog box appears, as shown in Figure 151.

Figure 151: Diamond Programmer Dialog Box



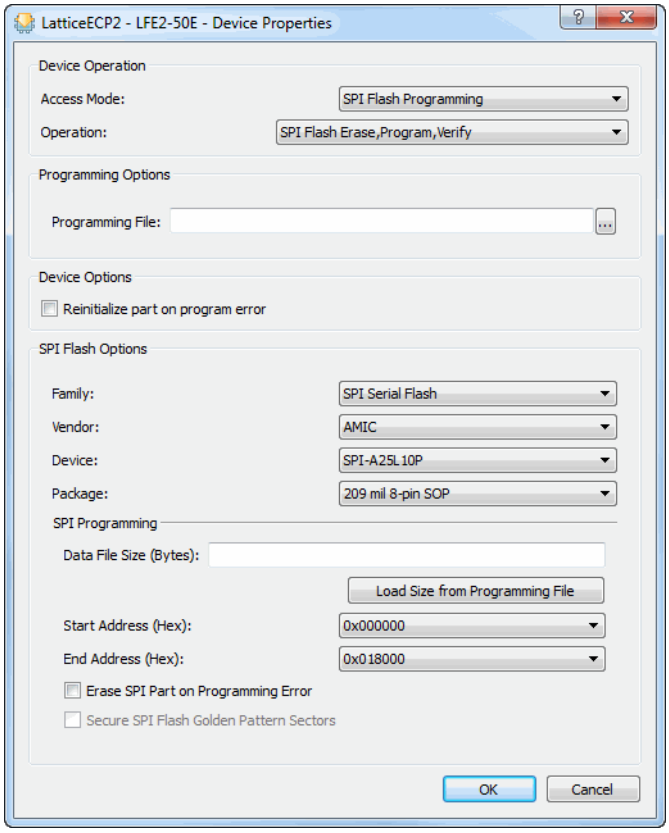
2. In the Getting Started dialog box, choose **Create a new Blank Project**. and click **OK**. Leave the **Import File to Current Implementation** box checked. Programmer scans the device database, and then the Programmer view displays, as shown in Figure 152.

Figure 152: Diamond Programmer



3. Double-click the Operation column to display the Device Properties dialog boxas shown in Figure 153.

Figure 153: Device Properties Dialog Box



4. In the Access Mode dialog box, choose **SPI Flash Programming**.
5. In the Operation Box, choose **SPI Flash Erase, Program, Verify**.
6. In the Programming Options box, in the Programming File box, click ... to browse to the data.mcs file.
7. In the SPI Flash Options box, select the following options to specify the SPI flash device on your board:
   - Family
   - Vendor
   - Device
   - Package
8. Click **Load Size**.
9. Click **OK**.
10. Click the Program button on the Programmer toolbar to initiate the download.

Once Diamond Programmer successfully programs the SPI flash, the SPI flash contains the FPGA bitstream, as well as the LatticeMico bootable application, completing deployment to the SPI flash.

Summary

The actual deployment scenario depends on the hardware setup, as well as the application requirements. The steps outlined in this section are guidelines that can be adapted to almost all SPI flash deployment scenarios, provided that the hardware, FPGA layout, or both allow the MSB SPI flash ROM device to read from the appropriate SPI flash.

Remember that:

- The LatticeMico EBA value—that is, the reset vector—must be set to the sum of the SPI flash ROM base address and the offset in the SPI flash device that contains the first instruction of the bootable LatticeMico application.
- The LatticeMico application must be programmed with the SPI flash device so that each instruction is aligned on a word boundary, because LatticeMico performs word fetches for instructions.

Debugging Tips

The diagnostic flow charts shown in Figure 154 and Figure 155 provide tips on what to look for when a SPI flash deployment fails.

Figure 154: SPI Flash Deployment Diagnostics Flow

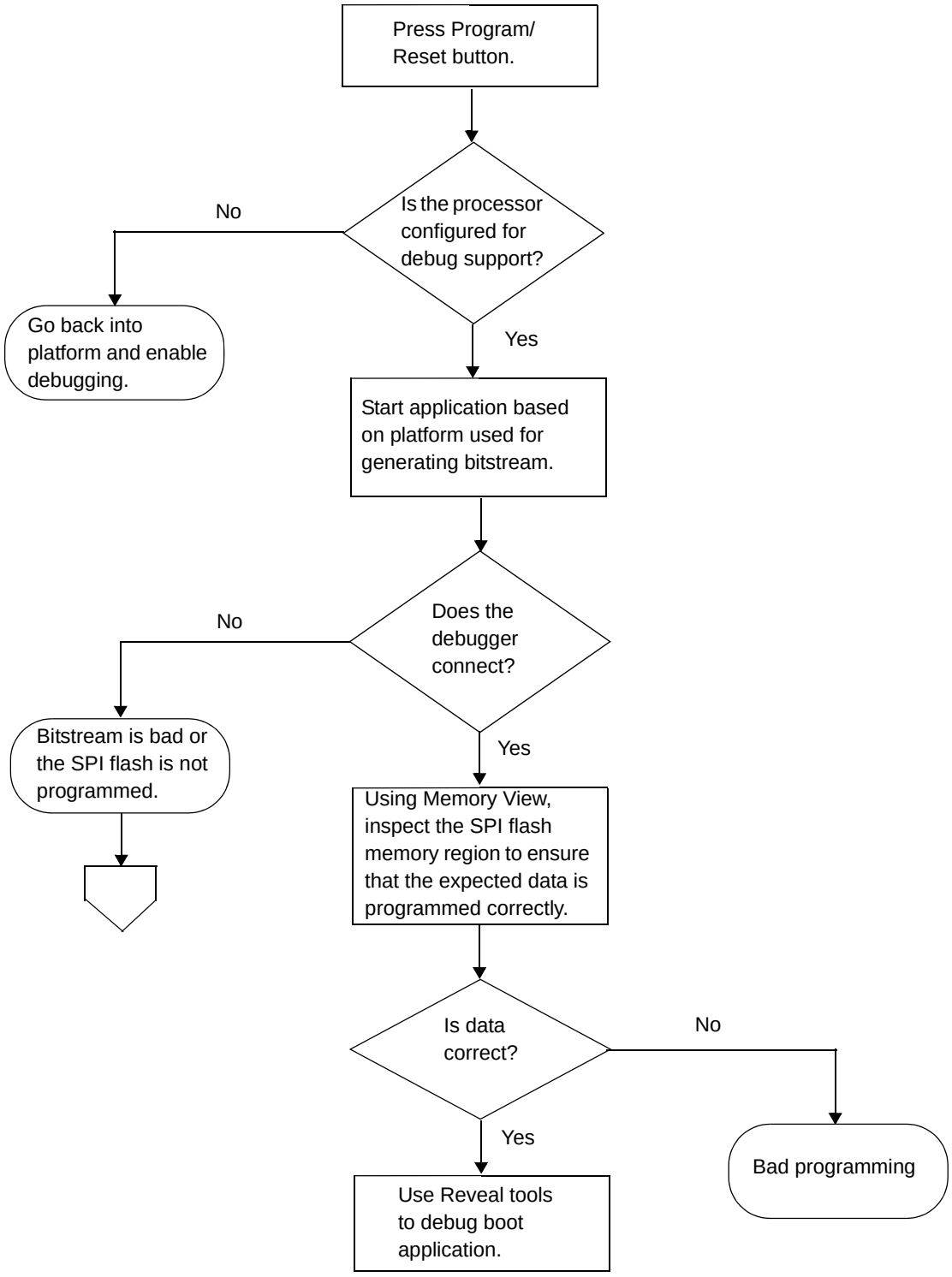
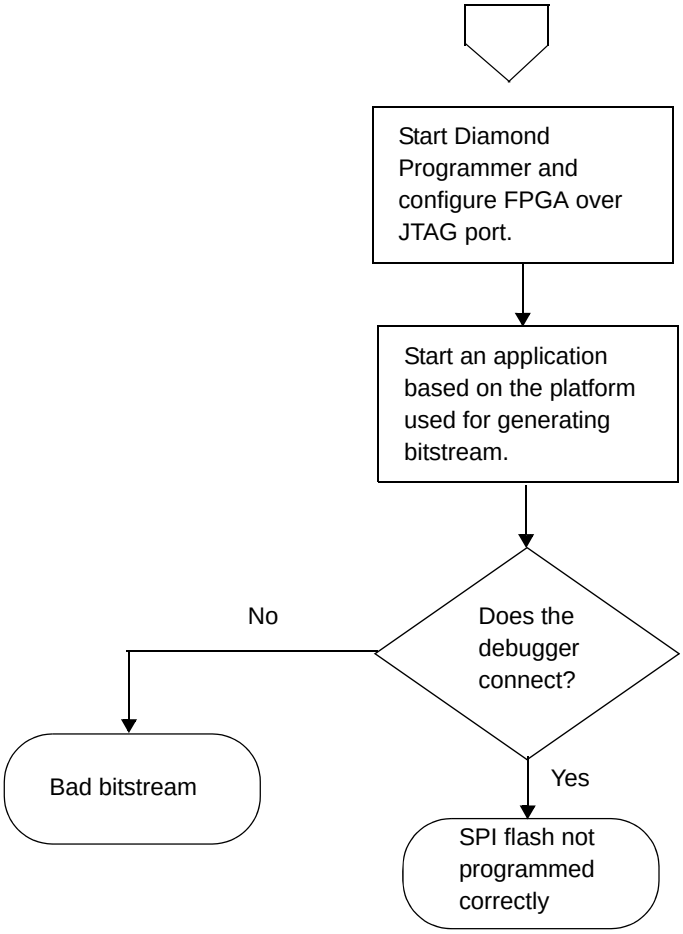


Figure 155: SPI Flash Deployment Diagnostics Flow, cont'd



"Bad programming" can result from two sources:

- An error in generating the merged binary data in Deployment Tool
- A Diamond Programmer programming error

Note

The current LatticeMico32 LatticeECP board requires you to press the Reset button after you press the Program button or after a power cycle.

Deploying Applications Across Different Memory Components

LatticeMico MSB provides packaged tools for deployment through the C/C++Software Project Environment (SPE), such as the flash deployment tool, the on-chip memory deployment tool, and the SPI flash deployment tool.

These tools assume that you are deploying the entire application to a single memory component, such as a flash, on-chip memory, or SPI, as the majority of deployment scenarios do. They do not yet support deployment of an application across different memory components.

This section introduces you to the tools that you must use to deploy your application to several different memory components. It includes usage examples. It assumes that you are attempting to deploy the raw binary contents of a LatticeMico stand-alone executable rather than an ELF (executable and linking format) loader that can interpret an .elf format file and load the contents to the destination memories itself.

Steps for Deploying an Application Across Different Memories

If you deploy an application across different memories, you will need to use the tools listed in “Useful Tools for Deployment” on page 212 to extract the appropriate information from the program executable, generate binary files or a memory initialization file from a binary format, and program the binary content for a flash memory or load a memory initialization file for an on-chip memory.

The steps are as follows:

1. Identify sections that need deployment by running objdump on the LatticeMico executable.
2. Use objcopy to extract the appropriate sections.
3. Optionally run bin_to_verilog to convert the binary output from objcopy into a hexadecimal memory initialization file for on-chip memories.

The next section discusses these tools in detail.

Useful Tools for Deployment

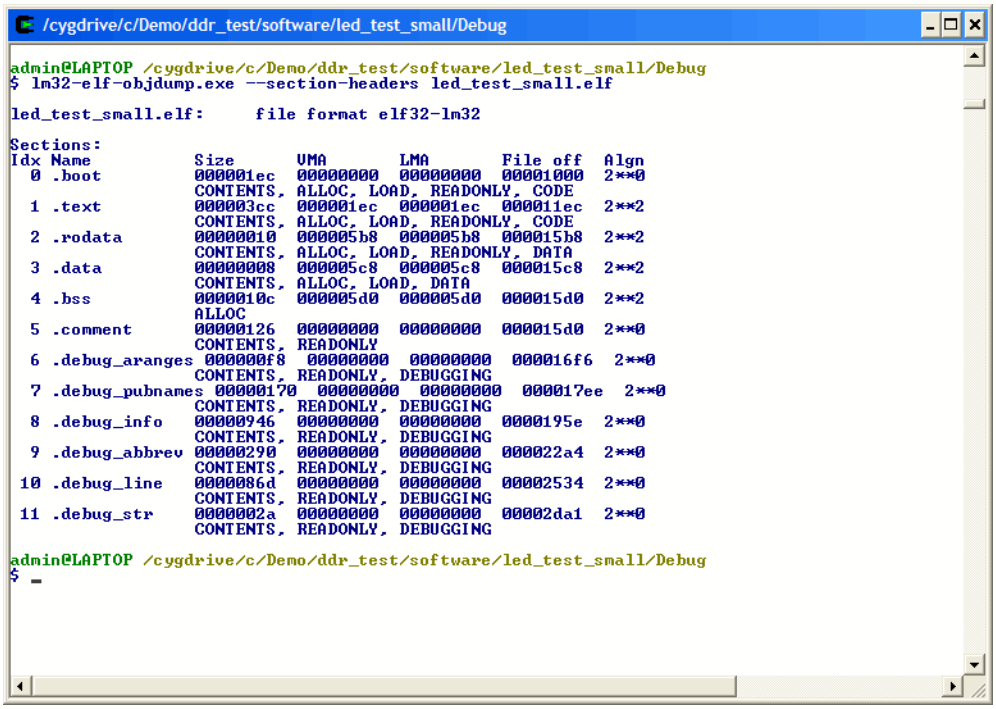
The tools used for deploying an application across multiple memory components are command-line tools, which are also used by the SPE deployment tools through Perl scripts.

lm32-elf-objdump This tool is provided as part of the LatticeMico GNU GCC/binutils port. It is located in the <*micosystem_install_dir*>/gtools/bin directory.

Objdump displays information for object files. Running this tool with the --help option displays the available options.

The --section-headers option is useful for deployment purposes. Figure 156 shows the output of running objdump on a LatticeMico executable .elf format file.

Figure 156: Objdump Example



The output lists the various sections in the executable file. Those sections with the ALLOC and LOAD tags are relevant for deployment. Sections with the ALLOC tags occupy memory during program execution, and sections with the LOAD tags contain data that memories must have for program execution. The sections with the ALLOC and LOAD tags must be deployed to non-volatile memories for power-up execution.

The SIZE column title indicates the size, in bytes, of the section contents. The LMA (Load Memory Address) column displays the address where the code is to be loaded, and the VMA (Virtual Memory Address) column displays for LatticeMico the address of the contents used when the code is compiled. For deployment purposes, the LMA is the destination address where the section contents must be copied. Refer to Technical Note 1173, “Deploying LatticeMico Software to Non-Volatile Memory,” for an example in which the VMA and LMA are different for some sections.

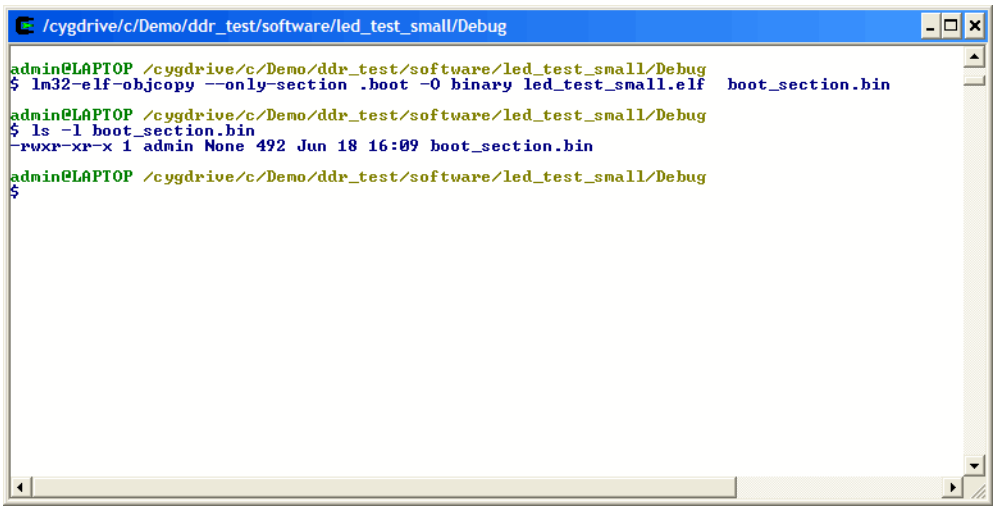
lm32-elf-objcopy This tool is provided as part of the LatticeMico GCC/ binutils port and is located in the (MICOSYSTEM_INSTALL_DIR)/gtools/bin directory.

Objcopy enables you to copy contents, either whole or selectively, to other formats. It is an essential tool for extracting sections or converting the entire program executable into binary format, which is a prerequisite for any

deployment scenario, unless LatticeMico System has an ELF loader. The SPE deployment tools use objcopy for format conversion.

The “--help" option lists the available options for this tool. The “--only-section” and “-O binary” options shown in Figure 157 are relevant for deployment.

Figure 157: Objcopy Example



In Figure 157, the --only-section .boot option tells objcopy to copy only the .boot section, and the “--O binary” option tells objcopy to create the output target in binary format. The input file is the LatticeMico executable .elf format file, led_test_small.elf, and the output file is boot-section.bin.

Using objcopy and the example shown in Figure 157, you can selectively extract the section contents as binary values for programming into destination memories. If you have more than two sections in a memory and would like to extract it to a single binary file, you can provide multiple --only-section options, such as --only-section .boot --only-section .text.

bin_to_verilog This Lattice-provided utility resides in the (MICOSYSTEM_INSTALL_DIR)/gtools/bin directory.

This utility is useful for converting binary files into a text format file suitable for initializing on-chip memories.

Running this tool with the --help option lists the available options, which are summarized in Table 17.

Table 17: bin_to_verilog Options

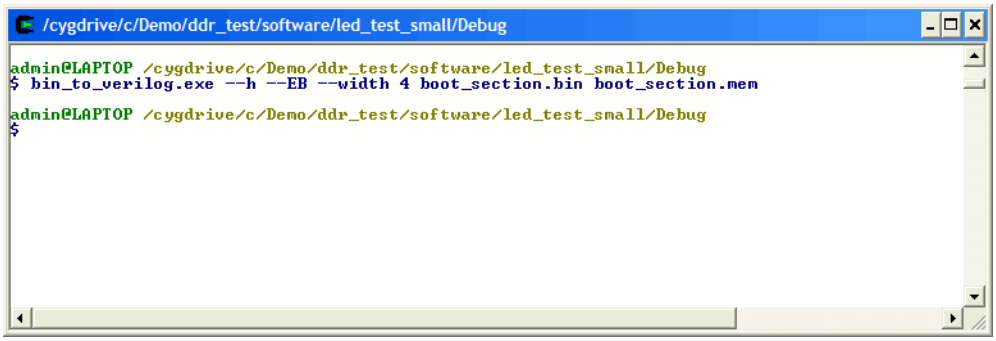
Option	Meaning
--b/--h	--h generates output words in hexadecimal notation, and --b generates words in binary representation. The default is --h.

Table 17: bin_to_verilog Options (Continued)

--EB/--EL	--EB generates output words suitable for big-endian interpretation, and --EL generates output words suitable for little-endian interpretation The default is --EL.
--width <*n*>	<*n*> denotes bytes per word (one word per line). The default value is 2.
infile	Input binary file
outfile	Output text file

Figure 158 shows how the bin_to_verilog tool converts a binary file generated by the objcopy utility into a memory initialization file for on-chip memory deployment. The on-chip memory provided as part of LatticeMico System has a word width of 4 bytes per word because LatticeMico System is a 32-bit system.

Figure 158: Using the bin_to_verilog Tool



The SPE on-chip memory deployment tool uses bin_to_verilog to generate the memory initialization file.

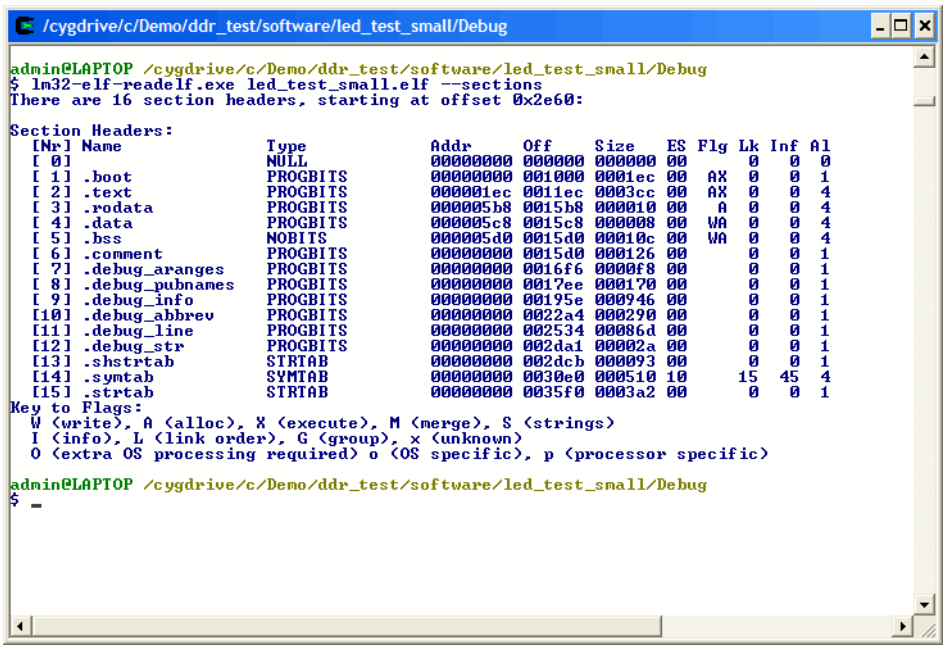
lm32-elf-readelf This tool is provided as part of the LatticeMico GNU GCC/ binutils port and is located in the (MICOSYSTEM_INSTALL_DIR)/gtools/bin directory.

The GCC tool chain generates LatticeMico executables in ELF format. The lm32_elf_readelf tool enables you to inspect the .elf file and program information and content.

Running this tool with the --help option displays the available options.

To determine the sections included in an .elf file, run the tool with the --sections option, as shown in Figure 159.

Figure 159: Using the lm32-elf-readelf Tool



In Figure 159, the sections with the “A” attribute set for the flags are those of interest for deployment (the Flg column in the figure). This flag denotes sections that occupy memory during process execution. For deployment, those sections that contain this attribute and have a size that is non-zero (Size column in the figure) must be deployed to the target memories whose addresses are shown in the Addr column.

You can explore other features of the lm32-elf-readelf tool by looking them up on the Internet and using the keywords “readelf” and “binutils.”

The objdump utility can also provide information on section headers, as shown in “lm32-elf-objdump” on page 212, so you can use either the lm32-elf-readelf tool or the objdump tool to inspect the available sections.

Additional Information

Refer to Technical Note 1173, “Deploying LatticeMic32 Software to Non-Volatile Memory,” as well as to the sections in “Software Deployment” on page 178 for additional information.

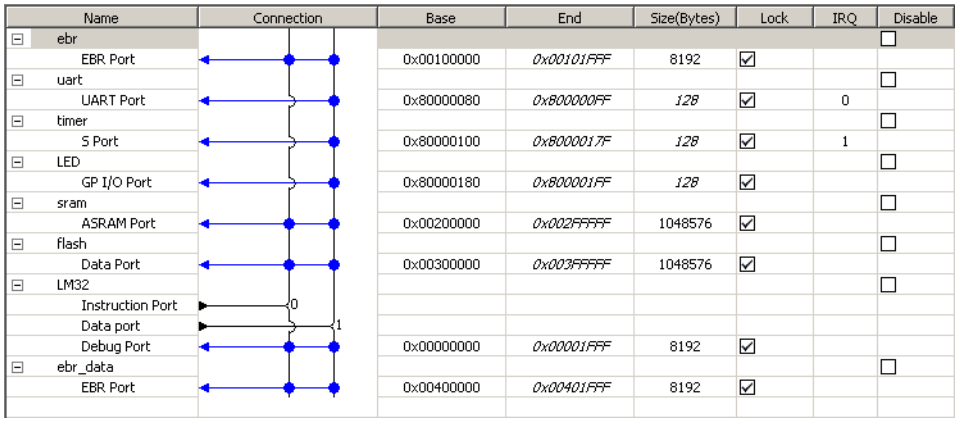
Example 1

The example in this section illustrates the usage of the tools described earlier. The goal of the example is to deploy the instruction sections of an application built as part of the managed build process to an on-chip memory component

and the initialized data sections to a separate memory component. The tools and procedures used in this example are generic, so you can use them in different deployment scenarios.

Example Platform Platform E is used for this example, although it has been modified to include an additional on-chip memory component, as shown in Figure 160.

Figure 160: Modified Platform E

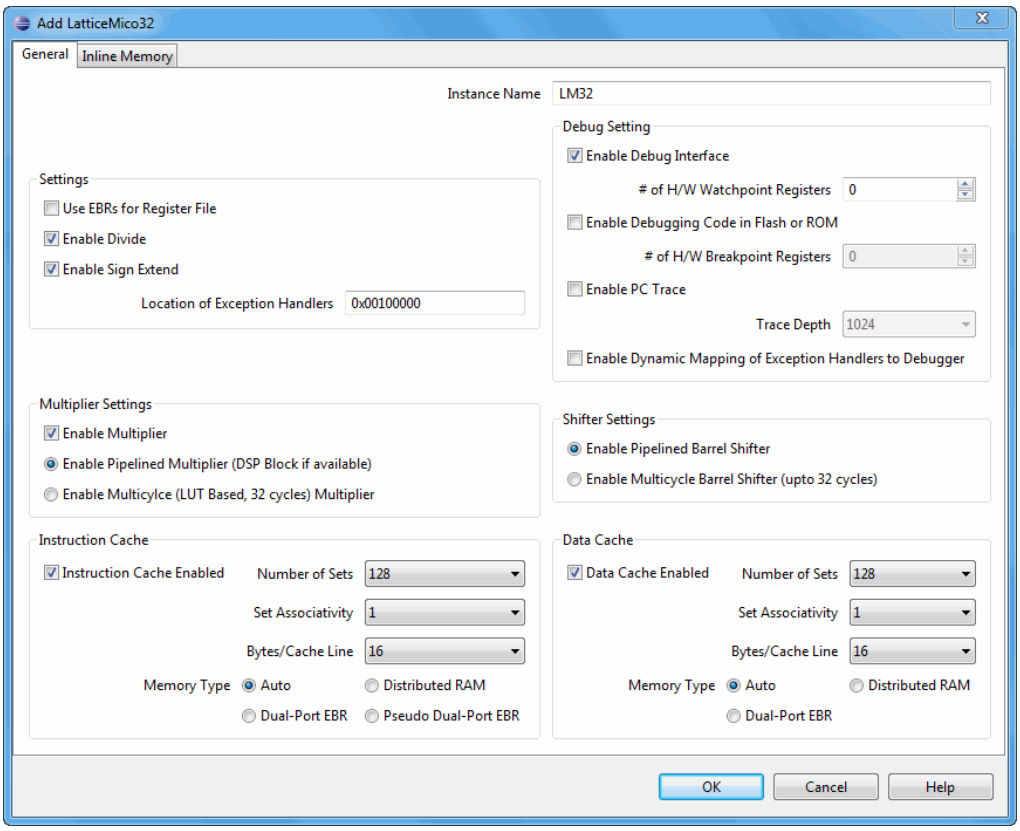


This platform contains four memory components:

- An on-chip memory component named ebr
- An on-chip memory component named ebr_data
- An external SRAM memory controller named sram
- An external flash memory controller named flash

The size of both on-chip memory components is set to 8192 bytes. The LatticeMico32 microprocessor has been modified to make this platform fit on a LatticeECP2-50E device residing on a LatticeMico32 LatticeECP2 development board. Because the deployment objective is for the on-chip memory component and not for the external flash, the address in the Location of Exception Handlers box in the Modify LatticeMico32 dialog box has been changed to the address of the ebr on-chip memory component, as shown in Figure 161.

Figure 161: Modified Processor Configuration



Note

For this example, it is essential that the address in the Location of Exception Handlers box (also known as EBA) be set to the base address of the ebr on-chip memory component, as shown in Figure 161.

Example Software Project The software application is based on LEDTest. The source in LEDTest.c has been modified, as shown in Figure 162.

Figure 162: LEDTest.c File

```
/*************************************************************
 * This example exercises LEDs on LatticeMico32 development   *
 * board.                                                     *
 *                                                            *
 *------------------------------------------------------------*
 * PREREQUISITES:                                             *
 *                                                            *
 * - GPIO with 8-bit output named LED connected to the        *
 *   board's LED pins.                                        *
 *************************************************************/
#include "system_conf.h"

/* PLATFORM DECOUPLING CONSTANTS/MACROS */
#define LED_OUTPUT(value)    \
    *((volatile unsigned int *)(LED_BASE_ADDRESS)) = (value)

#define NUM_PATTERNS    (16)

/*
 * The following array will go in .rodata section
 * i.e. read-only data section as it is initialized
 * to non-zero but is not modifiable.
 */
const unsigned int PatternData[NUM_PATTERNS]={
    0x01, 0xf0, 0x02, 0xf0, 0x04, 0xf0, 0x08, 0xf0,
    0x10, 0x0f, 0x20, 0x0f, 0x40, 0x0f, 0x80, 0xff,
};

/*
 * This "initValue" will be put in .data linker section
 * as it is initialized to non-zero AND is modifiable.
 */
volatile int initValue = 0xaa;

int main(void)
{
    int i;

    /*
     * copy data into read/write memory.  malloc will
     * allocate memory from .bss section
     */
    unsigned int *idata = malloc(NUM_PATTERNS*sizeof(unsigned int));
    for(i = 0; i < NUM_PATTERNS; i++){
        idata[i] = PatternData[i];
    }

    LED_OUTPUT(initValue);
    MicoSleepMilliSecs(10000);
```

Figure 162: LEDTest.c File (Continued)

```
 /* scroll the LEDs, every 1second forever */
    while(1){
        for(i = 0; i < NUM_PATTERNS; i++){
            initValue = ~idata[i];
            LED_OUTPUT(initValue);
            MicoSleepMilliSecs(1000);
        }
    }


    /* all done */
    return(0);
}

/*
 * override default LatticeDDInit to reduce
 * code-size (when using LEDs!)
 */
void LatticeDDInit(void)
{
    main();
}
```

The value in initValue is set to 0xaa. At start up, it is output to the LEDs on the LatticeMico32 development board. This value causes alternate LEDs to light up. This pattern is held for approximately 10 seconds before the LEDs cycle through the predefined pattern. If you successfully perform the deployment procedure outlined in this example, initValue, which is a non-zero initial value that can be modified at run time, should be located by the linker in the .data section.

Figure 163 shows the project outline created in C/C++ SPE.

Figure 163: Project Outline in C/C++ SPE

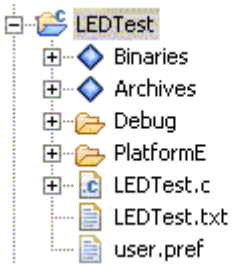
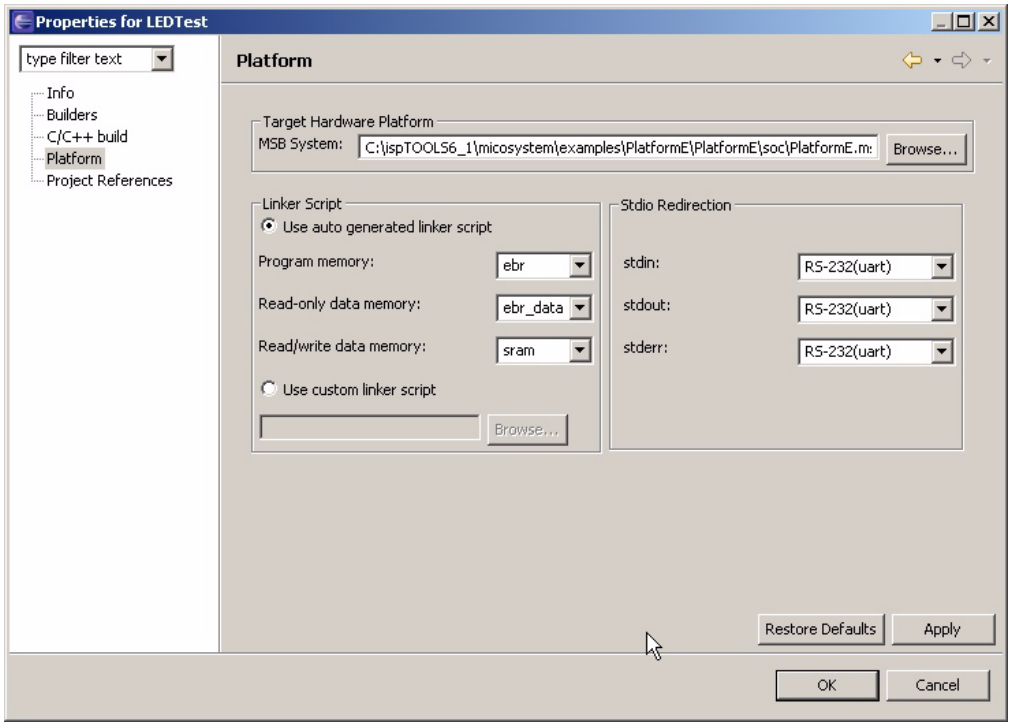


Figure 164 shows the platform settings (based on Platform E) for this software application.

Figure 164: Platform Settings



Deployment Objectives The deployment objectives for this software application are:

- Locate instructions (.text and .boot linker sections) in the ebr on-chip memory
- Locate read-only data in the ebr_data on-chip memory.
- Locate uninitialized memory in sram.

The platform settings have been changed to reflect the following:

- Program memory – ebr
- Read-only data memory – ebr_data
- Read/write data memory – sram
- Stdio Redirection: RS-232 (uart) – This change is not related to the deployment objectives. However, if the deployed application uses standard input and output, the standard I/O must not use the processor's JTAG UART.

Figure 164 shows the platform settings required to achieve these goals. As part of the managed build process, the following linker sections are generated:

- .boot – Contains instructions executed as part of the boot code (crt0ram.S).

- .text – Contains executable instructions.
- .rodata – Contains read-only initialized data that is initialized to a non-zero value and cannot be modified at run time.
- .data – Contains initialized data that is not initialized to zero but can be modified at run time, for example, the initValue variable in the source code in the LEDTest.c file.
- .bss – Contains uninitialized data and any initialized data that is initialized to a value of zero. The boot code is responsible for zeroing the memory region containing the .bss section.

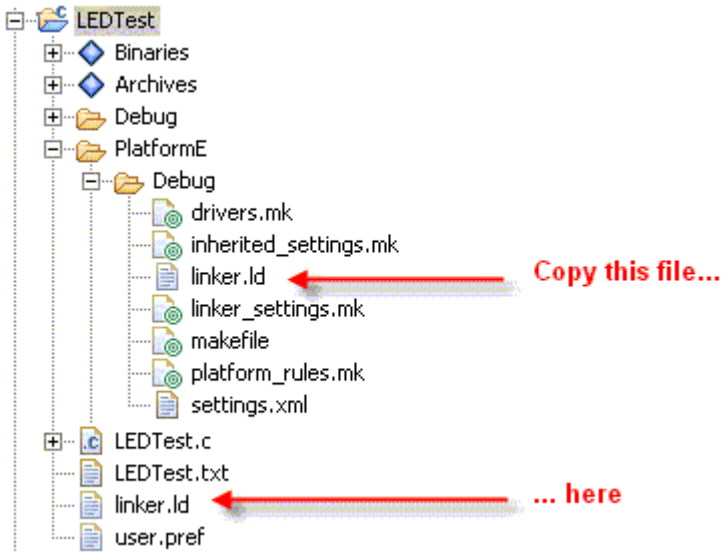
As part of the managed build process, the Properties for LEDTest dialog box (Figure 164) generates a default linker script, as described in “Managed Build Process and Directory Structure” on page 145. This default linker script automatically assigns the .data linker section to the memory component selected to hold read and write data memory. Although this assignment works well for debugging an application, it does not work when you deploy the application if the read and write data memory is a volatile memory, such as an external SRAM, because the .data linker section contains non-zero initialized values that can be modified at run time. If the deployment involved a loader that copied the application executable sections to appropriate target memory, this issue would not arise. The parallel flash deployment strategy outlined in “Deploying to a Flash Device” on page 193 provides a default boot copier that copies the various sections from the parallel flash to the target memories.

Modifying the Linker Script The first step in achieving the deployment objectives is to modify the linker script.

To modify the linker script:

1. Copy the default linker script, located in the build configuration directory of the platform library folder (created by the managed build when you first built the application) to the application folder, as shown in Figure 165.

Figure 165: Copying Linker.ld Default Linker Script



Copying the linker script to the application folder ensures that the changes made to the file will not be erased when you rebuild the project.

Since the linker script was generated with the modified platform settings, the .text and .boot sections are already targeted for the ebr on-chip memory. The .bss section is correctly targeted to the sram external memory. The .rodata section is also correctly targeted to the ebr_data on-chip memory. The only change required is to modify the linker script so that the .data section points to ebr_data instead of sram.

2. To make this change, identify the .data section specification in the copied linker.ld file, and change sram to ebr_data, as shown in Figure 166.

Figure 166: Modifying the .data Section

```
/* read/write data */
.data :
{
   . = ALIGN (4);
   _fdata = .;
   *(.data .data.* .gnu.linkonce.d.*)
   *(.data1)
   SORT(CONSTRUCTORS)
   _gp = ALIGN(16) + 0x7ff0;
   *(.sdata .sdata.* .gnu.linkonce.s.*)
   _edata = .;
} > ebr_data
```

The complete modified linker script is shown in Figure 167.

Figure 167: Modified Linker Script Code

```
OUTPUT_FORMAT("elf32-lm32")
ENTRY(_start)
INPUT(crti.o crtbegin.o crtend.o crtn.o)
/*
 * This section defines memory attributes (name, origin, length) for the platform
 */
MEMORY
{
    ebr :  ORIGIN = 0x00100000,  LENGTH = 8192
    sram :  ORIGIN = 0x00200000,  LENGTH = 1048576
    flash :  ORIGIN = 0x00300000,  LENGTH = 1048576
    ebr_data :  ORIGIN = 0x00400000,  LENGTH = 8192
}


SECTIONS
{

  /* code */
  .boot : { *(.boot) } > ebr
  .text :
  {
    . = ALIGN(4);
     _ftext = .;
     *(.text .stub .text.* .gnu.linkonce.t.*)
     *(.gnu.warning)
     KEEP (*(.init))
     KEEP (*(.fini))


     /* Exception handlers */

     *(.eh_frame_hdr)
     KEEP (*(.eh_frame))
     *(.gcc_except_table)
```

Figure 167: Modified Linker Script Code (Continued)

```
    /* Constructors and destructors */
    KEEP (*crtbegin*.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend*.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
    KEEP (*crtbegin*.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend*.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
    KEEP (*(.jcr))
   _etext = .;
} > ebr =0



/* read-only data */
.rodata :
{
   . = ALIGN(4);
   _frodata = .;
   _frodata_rom = LOADADDR(.rodata);
   *(.rodata .rodata.* .gnu.linkonce.r.*)
   *(.rodata1)
   _erodata = .;
} > ebr_data



/* read/write data */
.data :
{
   . = ALIGN(4);
   _fdata = .;
   *(.data .data.* .gnu.linkonce.d.*)
   *(.data1)
   SORT(CONSTRUCTORS)
   _gp = ALIGN(16) + 0x7ff0;
   *(.sdata .sdata.* .gnu.linkonce.s.*)
   _edata = .;
} > ebr_data

* bss */
.bss :
{
   . = ALIGN(4);
   _fbss = .;
   *(.dynsbss)
   *(.sbss .sbss.* .gnu.linkonce.sb.*)
   *(.scommon)
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   . = ALIGN(4);
   _ebss = .;
   _end = .;
   PROVIDE (end = .);
} > sram
```

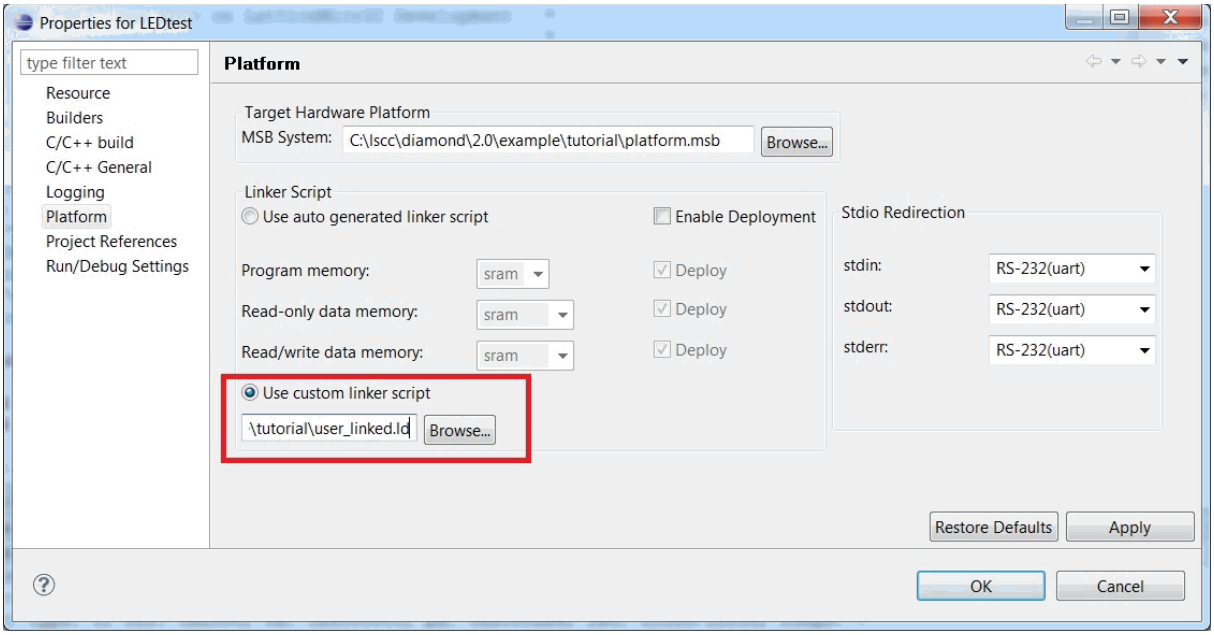
Figure 167: Modified Linker Script Code (Continued)

```
  /* first location in stack is highest address in ram */
  PROVIDE(_fstack = ORIGIN(sram) + LENGTH(sram) - 4);



  /* stabs debugging sections.  */
  .stab          0 : { *(.stab) }
  .stabstr       0 : { *(.stabstr) }
  .stab.excl     0 : { *(.stab.excl) }
  .stab.exclstr  0 : { *(.stab.exclstr) }
  .stab.index    0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment       0 : { *(.comment) }



  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 : { *(.debug) }
  .line           0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }
}
```

3. Now you must build the project using this linker script instead of the default linker script. To make this change, select **Use Custom Linker Script** from the Platform Properties tab of the Properties for LEDTest dialog box, and select the copied linker script modified in an earlier step.

Click the **OK** button. Figure 168 shows the modified Platform Properties tab of the dialog box.

Figure 168: Modified Properties in LEDTest Dialog Box

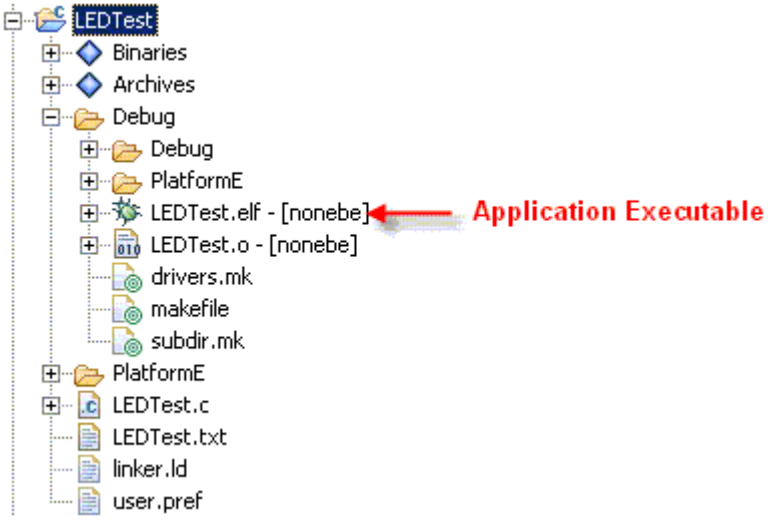


4. Rebuild the project with *all optimizations turned off*.

   These steps do not depend on the build configuration (that is, on a release or debug), but for verifying the steps outlined here, it is essential that you turn off all optimization.

Verifying the Linker Script Change The build process generates the application executable (LEDTest.elf for this example) in the build configuration folder located in the project folder, as shown in Figure 169.

Figure 169: Application Executable Location

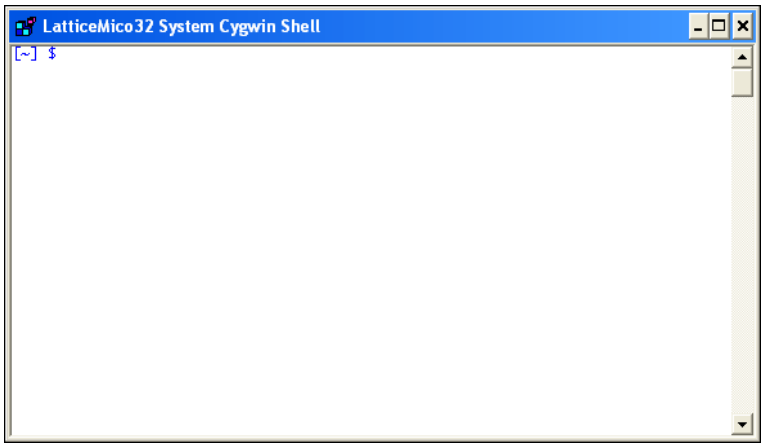


To verify the linker script change:

1. Start the LatticeMico SDK shell console window by selecting **Start > Programs > Lattice Semiconductor > Accessories > LatticeMico System SDK Shell**.

   The LatticeMico System Cygwin shell opens, as shown in Figure 170.

Figure 170: LatticeMico System Cygwin Shell



2. Change to the directory containing the application executable.

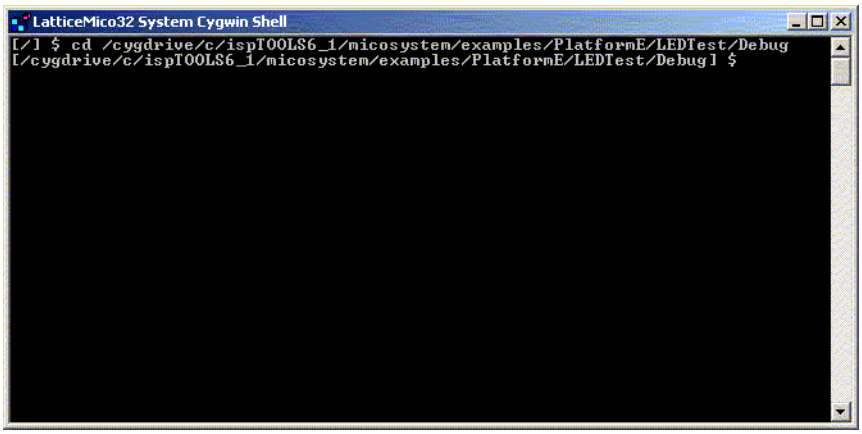
   For example, if the application executable is located in the following directory:

   `c:\ispTOOLS7_1\micosystem\examples\PlatformE\LEDTest\Debug`

you would type the following command at the prompt in the LatticeMico System Cygwin shell, as shown in Figure 171, and press the Enter key on the keyboard:

```
cd /cygdrive/c/ispTOOLS7_1/micosystem/examples/PlatformE/
LEDTest/Debug/
```

Figure 171: Moving to the Application Executable Directory

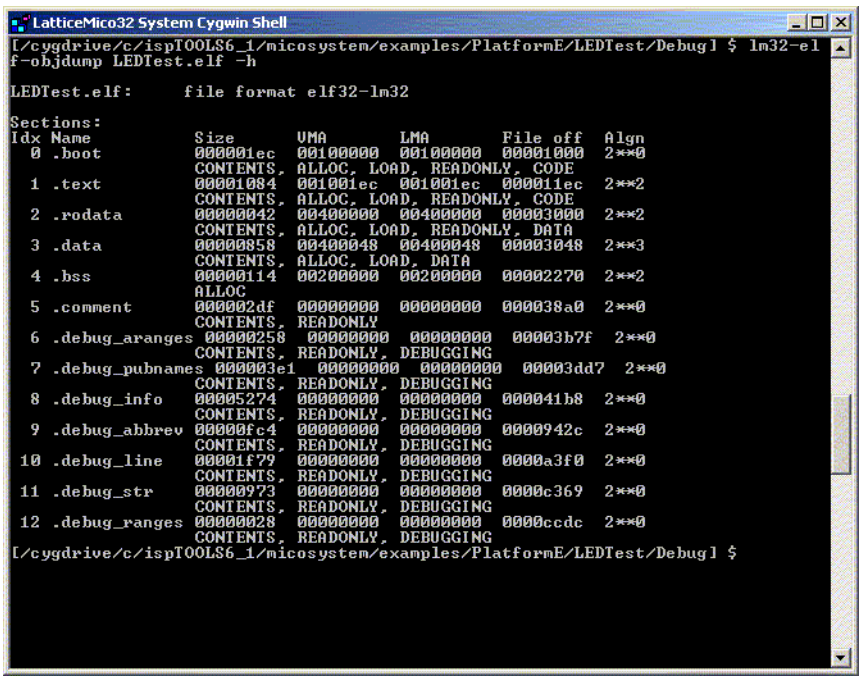


3. Type the following command at the prompt and press the **Enter** key:

   `lm32-elf-objdump LEDTest.elf -h`

   This command instructs the “objdump” utility to dump information on the section headers contained in the executable, as shown in Figure 172. The lm32-elf-objdump utility is the GNU objdump utility built for LatticeMico System. “Software Development Utilities” on page 281 contains a list of the included GNU GCC tool-chain utilities.

Figure 172: lm32-elf-objdump Output



As shown in Figure 172, the .boot and .text sections are targeted to the ebr on-chip memory, the .data and .rodata sections are targeted to the ebr_data on-chip memory, and the .bss section is targeted to the SRAM external memory.

The output in Figure 172 verifies that the build picked up the modified linker script instead of using the default linker script. It also verifies that the linker-script changes were appropriate.

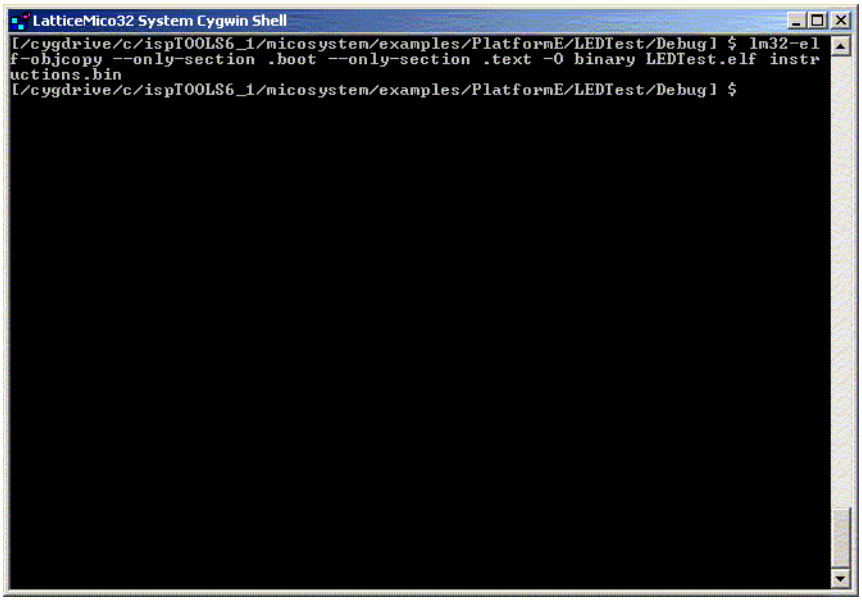
Extracting Binary Data from the Appropriate Sections The .text and .boot binary contents must be extracted as a single binary file because they will reside in the same memory region.

To extract the binary data from the appropriate sections:

1. At the command prompt in the LatticeMico System Cygwin shell, enter the following command, as shown in Figure 173, and press the **Enter** key:

```
lm32-elf-objcopy --only-section .boot --only-section .text -
O binary LEDTest.elf instructions.bin
```

Figure 173: Extracting Binary Data



This command generates a file, instructions.bin, that contain the binary contents of the .boot and .text sections.

The .rodata and .data sections must be extracted as a single binary file, because they, too, reside in the same memory region, though in a different region than .text and .boot sections.

2. At the command prompt in the LatticeMico System Cygwin shell, enter the following command, and press the **Enter** key:

   ```
   lm32-elf-objcopy --only-section .rodata --only-section .data
   -O binary LEDTest.elf data.bin
   ```

   This command generates a file, data.bin, that contains the binary contents of the .data and .rodata sections.

   The .bss section does not contain any pre-initialized data. The memory contents for this section are set to zero at run time by the boot code.

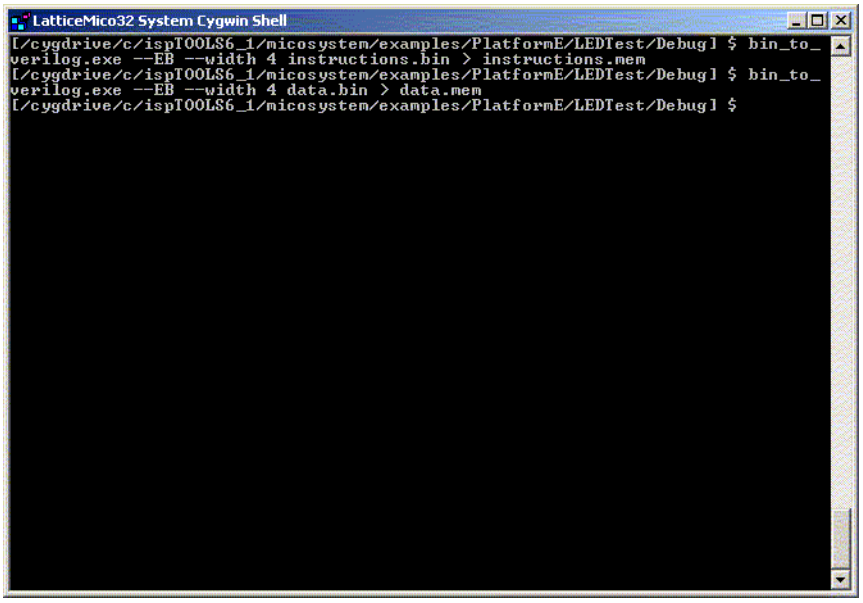
Converting Binary Data to On-Chip Memory Initialization Files Now you will convert the binary contents extracted from the sections into on-chip memory initialization files.

To convert the binary contents to on-chip memory initialization files:

- Enter the following two commands at the command prompt, as shown in Figure 174:

  ```
  bin_to_verilog --EB --width 4 instructions.bin >
  instructions.mem
  bin_to_verilog --EB --width 4 data.bin > data.mem
  ```

Figure 174: Generating Memory Initialization Files



The outputs of these commands are two files, instructions.mem and data.mem. They are text files that can be used to initialize the ebr and the ebr_data on-chip memory components.

Verifying Deployment Now you will verify that you deployed the instruction (.boot and .text) sections of the application to the ebr on-chip memory component and the initialized data (.data and .rodata) sections to the ebr_data on-chip memory component.

- Regenerate the bitstream with the ebr component's initialization file set to instructions.mem and the ebr_data component's initialization file set to data.mem.

When you download the bitstream, the LEDs should display an alternate lighting pattern (0xaa) for approximately 10 seconds before cycling through the predefined pattern noted in "Example Software Project" on page 218.

Example 2

While the earlier example illustrated a generic procedure for deploying the default instruction (.boot and .text) and data (.rodata, .data, and .bss) sections of an application into multiple memory components, the goal of this next example is to demonstrate a generic methodology to (a) locate portions of the application's instruction and data in to new user-defined sections, and (b) deploy these newly created user-defined sections into different memory components using a combination of C/C++ syntax and custom linker scripts.

Recommended C/C++ Coding Style to partition source code in to User-Defined Sections

Any given piece of code within the program source code can be mapped to a new user-defined memory section through the following sequence of steps.

Step 1 Reorganize the source code into new C/C++ functions. Any piece of source code that will be mapped to a new user-defined section must first be reorganized in to a new C/C++ function. This new function can be mapped in to a memory component that is located at an address far from the code that calls this function. In order to allow the GNU toolchain to automatically compile and link the caller and callee code, it is advisable to use function pointers. The use of function pointers is mandatory when the function being called is +/- 225 (i.e. 32MB) bytes distant from the current program counter location. Consider the code snippet shown in Figure 175.

Figure 175: Code with Function Pointers

```
int main(void)
{
  unsigned int iValue = 0x1;
  …
  …

  /* if we're not to blink, return immediately */
    if(uiBlink == 0)
        return(0);

    /* scroll the LEDs, every 100 msecs forever */
  while(1){
    *((volatile unsigned int *)(leds->base)) = ~iValue;
    MicoSleepMilliSecs(100);
    if(iShiftLeft == 1){ // code to be mapped to new section .INLINE_I
      iValue = iValue << 1;
      if(iValue == 0x100){
        iValue = 0x40;
        iShiftLeft = 0;
      }
    }else{
      iValue = iValue >> 1;
      if(iValue == 0){
        iValue = 0x02;
        iShiftLeft = 1;
      }
    }
  }

    /* all done */
  return(0);
}
```

In order to map the “if-else” code (highlighted in red) to a new section, the code must be rewritten as shown in Figure 176. A new function shift is created and the “while loop” calls this function.

Figure 176: Code with New Function

```
/* Function that contains the "if-else" code */
unsigned int shift (unsigned int iValue, unsigned int *iShiftLeft)
{
  if (*iShiftLeft == 1){
    iValue = iValue << 1;
    if (iValue == 0x100){
      iValue = 0x40;
      *iShiftLeft = 0;
    }
  }else{
    iValue = iValue >> 1;
    if(iValue == 0){
      iValue = 0x02;
      *iShiftLeft = 1;
    }
  }
  return iValue;
}

int main(void)
{
  unsigned int (*shift_ptr) (unsigned int, unsigned int *) = shift; // function
pointer
  unsigned int iValue = 0x1;
  …
  …

    /* if we're not to blink, return immediately */
    if(uiBlink == 0)
        return(0);

   /* Function the scroll the LEDs, every 100msecs, forever */
  while(1){
    *((volatile unsigned int *)(leds->base)) = ~iValue;
    MicoSleepMilliSecs(100);
    iValue = (*shift_ptr) (iValue, &iShiftLeft); // new indirect-function call
    // iValue = shift_ptr (iValue, &iShiftLeft); // valid alternate syntax for a call
  }

    /* all done */
  return(0);
}
```

Step 2 Use the C/C++ section attribute to specify a user-defined memory section for a given piece of code. This attribute can only be placed on a function prototype. In the example of Figure 176, the function shift can be placed in a user-defined section ".INLINE_I" by adding the following declaration:

```
unsigned int shift (unsigned int, unsigned int *) __attribute__
((section (".INLINE_I")));
```

Recommended C/C++ Coding Style to Partition Data into User-Defined Sections

Use the C/C++ section attribute to specify a user-defined memory section for data (read-only or read/write). For example, the iShiftLeft variable in the code in Figure 175 can be placed in a user-defined section ".INLINE_D" by declaring it as follows:

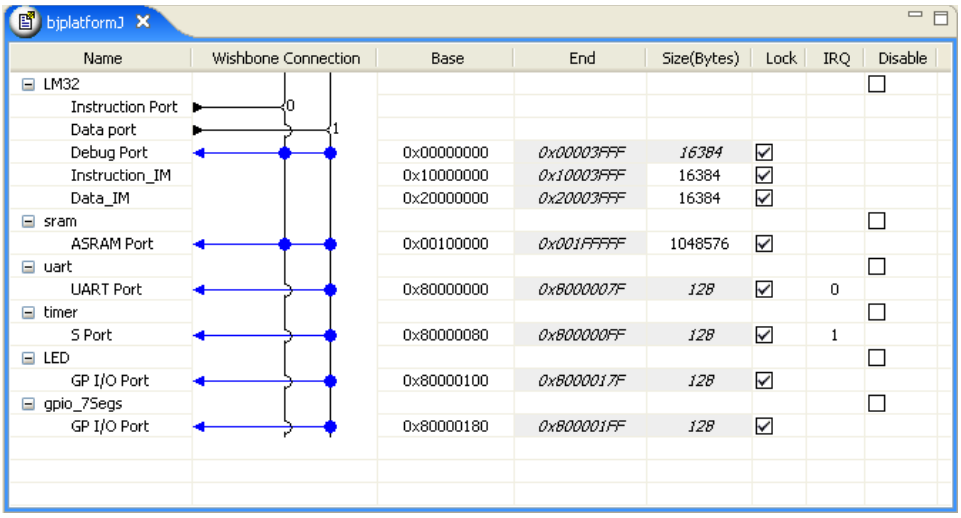
```
unsigned int iShiftLeft __attribute__ ((section
(".INLINE_D")));
```

Notice that the "section" attribute can only be used with GLOBAL or STATIC variables. The compiler toolchain will ignore the "section" attribute on any other variable.

Example Platform and Software Project

The remainder of this section describes an example of how an application was modified in order to split pieces of instruction and data across multiple memory sections. Platform J is used for this example. As shown in Figure 177, this platform contains three memory components:

Figure 177: PlatformJ



The software application is LED7SegsTest. The application code was modified for the following deployment objectives on Platform J:

- Part of source code to display characters on 7-Segement Display are split in to a new section ".INLINE_I" that is mapped to Instruction Inline Memory "Instruction_IM". The remaining instructions default to the .boot and .text sections and are mapped to "sram".
- Part of read/write data is split in to a new section ".INLINE_D" that is mapped to Data Inline Memory "Data_IM". The remaining data defaults to the .rodata, .data, and .bss sections and are mapped to "sram".
- Inspecting the memory map above it can be seen that the Instruction_IM memory block is >> 32MB distant from the sram memory block. Function calls between the two memory spaces must be done using function pointers.

The source in LED7SegsTest.c has been modified as shown in Figure 2. Function "TimerISR()" is mapped in to the new .INLINE_I instruction memory section. Data "iCount" is mapped to the new .INLINE_D data memory section. The source in 7Segs.h and 7Segs.c has been modified as shown in Figures 3 and 4 respectively. Functions "UpdateDisplay()" and "UpdateChar()" are also mapped in to the new .INLINE_I instruction memory section. The "UpdateDisplay()" function calls the "GetCode()" function, which is still mapped to the default .text section. The UpdateDisplay() fuction calls GetCode() through an indirect call (i.e., function pointer), as explained in "Step 1" on page 233.

Figure 178: LED7SegsTest.c file

```
/*************************************************************
 * This example exercies LEDs and the 7 Segment displays on  *
 * the LatticeMico32 Development board.                      *
 *                                                           *
 *                                                           *
 *-----------------------------------------------------------*
 * PREREQUISITES:                                            *
 *                                                           *
 * - GPIO with 10-bit output named gpio_LED connected to the *
 *   board's LED pins.                                       *
 * - GPIO with 10-bit output named gpio_7Segs connected to   *
 *   the board's 7-Segment Displays' pins.                   *
 *************************************************************/
#include "DDStructs.h"
#include "MicoMacros.h"
#include "7Segs.h"
#include "LookupServices.h"
#include "MicoTimer.h"
#include "MicoUtils.h"

const char *SYSTEM_TIMER = "timer";
const char *LED_GPIO_INSTANCE = "LED";
const char *SEGMENT_LED_INSTANCE = "gpio_7Segs";

typedef struct st_TimerISRCtx_t{
  int iSelection;/* segment 0 or segment 1*/
  char c[2];/* characters for the two segments*/
  SegmentDisp_t *display;/* display information*/
}TimerISRCtx_t;

/* Declarations for contents of .INLINE_D section */
static int iCount __attribute__ ((section (".INLINE_D"))) = 0;

/* Declarations for contents of .INLINE_I section */
static void TimerISR(void *data) __attribute__ ((section (".INLINE_I")));

/* Timer ISR */
static void TimerISR(void *data)
{
  TimerISRCtx_t *ctx = (TimerISRCtx_t *)data;
  SegmentDisp_t *display = ctx->display;
  /* refresh display */
  (*DisplayChar) (    display,
         ctx->iSelection,
           ctx->c[ctx->iSelection]);

  if(ctx->iSelection == 0)
    ctx->iSelection = 1;
  else
    ctx->iSelection = 0;
```

Figure 178: LED7SegsTest.c file (Continued)

```
/* every ~ 1 second, change data */
  if(iCount >= 99){
    iCount = 0;
    if(ctx->c[0] >= '9'){
      ctx->c[0] = '0';
      if(ctx->c[1] >= '9')
        ctx->c[1] = '0';
else
        ctx->c[1]++;
    }else{
      ctx->c[0]++;
    }

  }else{
    iCount++;
  }
  return;
}


int main(void)
{
unsigned int iValue = 0x1;
  unsigned int iShiftLeft = 1;
  static TimerISRCtx_t TimerCtx;
  static SegmentDisp_t display;
  char c;

  /* Fetch timer instance named "timer" */
  MicoTimerCtx_t *timer = (MicoTimerCtx_t *)MicoGetDevice(SYSTEM_TIMER);
    /* Fetch GPIO instance named "LED" */
  MicoGPIOCtx_t *leds = (MicoGPIOCtx_t *)MicoGetDevice(LED_GPIO_INSTANCE);
  /* Fetch 7segs GPIO Instance */
  MicoGPIOCtx_t *segs = (MicoGPIOCtx_t *)MicoGetDevice(SEGMENT_LED_INSTANCE);

  /* see if we found LED */
    if(leds == 0){
        printf("failed to find GPIO instance named %s\n",LED_GPIO_INSTANCE);
        return(-1);
    }
    printf("found GPIO instance named %s\n",LED_GPIO_INSTANCE);
/* see if we found 7-segment LED GPIO */
  if(segs == 0){
    printf("failed to find GPIO instance named %s\n", SEGMENT_LED_INSTANCE);
    return(-1);
  }
  printf("found GPIO instance named %s\n", SEGMENT_LED_INSTANCE);
```

Figure 178: LED7SegsTest.c file (Continued)

```
/* see if we found our timer */
  if(timer == 0){
    printf("failed to find timer instance named %s\n", SYSTEM_TIMER);
    return(-1);
  }
/*
   * Before we start blinking the LEDs, setup the timer to generate
   * 1-second periodic interrupts
   */
  display.gpio = segs;

  TimerCtx.c[0] = '0';
  TimerCtx.c[1] = '0';
  TimerCtx.display = &display;
  TimerCtx.iSelection = 0;

  MicoTimerStart(timer,/* timer instance*/
        TimerISR,/* isr routine */
        (void *)&TimerCtx,/* data for isr*/
        MILLISECONDS_TO_TICKS(10),/* timer-period*/
        1 );    /* periodic mode*/

    /* scroll the LEDs, every 100 msecs forever */
  while(1){
    *((volatile unsigned int *)(leds->base)) = ~iValue;
    MicoSleepMilliSecs(100);
    if(iShiftLeft == 1){
      iValue = iValue << 1;
      if(iValue == 0x100){
        iValue = 0x40;
        iShiftLeft = 0;
      }
    }else{
      iValue = iValue >> 1;
      if(iValue == 0){
        iValue = 0x02;
        iShiftLeft = 1;
      }
    }
  }

    /* all done */
  return(0);
}
  printf("found timer instance %s\n", SYSTEM_TIMER);
```

Figure 179: 7Segs.h

```
#ifndef _7SEGMENTS_H_
#define _7SEGMENTS_H_

#include "MicoGPIO.h"

/* context structure */
typedef struct st_SegmentDisp_t {

  MicoGPIOCtx_t *gpio;/*
         * gpio that connects to 7-segment display
         */
  int iSelectedSegment;/*
         * Identifies which segment is currently selected
         * for display
         */
  char c;/*
         * code (0 - 9 and . ' ' ) that's being displayed
         */
}SegmentDisp_t;

/* should be called from a timer ISR */
void UpdateDisplay (SegmentDisp_t *display) __attribute__ ((section (".INLINE_I")));

/* displays a character on a segment */
void DisplayChar (SegmentDisp_t *display, int iSegment, char c) __attribute__
((section (".INLINE_I")));

/* converts a character 0-9, a-f  and . to a valid code */
unsigned int GetCode (char c);


#endif /*7SEGMENTS_H_*/
```

Figure 180: 7Segs.c file

```
#include "7Segs.h"
#include "MicoGPIO.h"

#define SEG_A(0x01)
#define SEG_B(0x02)
#define SEG_C(0x04)
#define SEG_D(0x08)
#define SEG_E(0x10)
#define SEG_F(0x20)
#define SEG_G(0x40)
#define SEG_DOT(0x80)

#define SEGMENT_A(0x100)
#define SEGMENT_B(0x200)

/* converts a character 0-9, a-f  and . to a valid code */
unsigned int GetCode (char c)
```

Figure 180: 7Segs.c file (Continued)

```
{
  /*return(SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G); */
  switch(c){
    case '.':
      return(SEG_DOT);
    case '0':
      return(SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F);
    case '1':
      return(SEG_B | SEG_C);
    case '2':
      return(SEG_A | SEG_B | SEG_D | SEG_E | SEG_G);
    case '3':
      return(SEG_A | SEG_B | SEG_C | SEG_D | SEG_G);
    case '4':
      return(SEG_B | SEG_C | SEG_F | SEG_G);
    case '5':
      return(SEG_A | SEG_C | SEG_D | SEG_F | SEG_G);
    case '6':
      return(SEG_A | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G);
    case '7':
      return(SEG_A | SEG_B | SEG_C);
    case '8':
      return(SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G);
    case '9':
      return(SEG_A | SEG_B | SEG_C | SEG_D | SEG_F | SEG_G);
    default:
      return(0);
  }
}
/* updates display */
void UpdateDisplay (SegmentDisp_t *display)
{
  /* The caller and callee functions are located in different sections.
   * Since these sections can be located beyond the range of a direct
   * call (i.e., calli), we need to use function pointers to instruct
   * gcc to generate indirect calls (i.e., register-based call) */
  unsigned int (*GetCode_ptr) (char) = GetCode;
  int code;

  /* select the 7-seg module */
  code = display->iSelectedSegment;

  /* fetch display code */
  code |= (*GetCode_ptr) (display->c);

/* complement the bits of the segments are active-low */
  code = ~code;

  /* output the code on the gpio */
  *((volatile int *)(display->gpio->base)) = code;
  }
```

Figure 180: 7Segs.c file (Continued)

```
/* displays a number (0 - 9) on a segment */
void DisplayChar (SegmentDisp_t *display, int iSegment, char c)
{

  /* store the character being displayed */
  display->c = c;


  /* select segment */
  if(iSegment == 0){
    display->iSelectedSegment = SEGMENT_A;
  } else {
    display->iSelectedSegment = SEGMENT_B;
  }
  /* refresh display */
  UpdateDisplay(display);
  /* all done */
  return;
}
```

Using the LED7SegsTest Project

The aforementioned modifications are available in the example LED7SegsTest Project through Lattice MicoSystem Builder. Perform the following steps while compiling the application:

1. Set the __PLATFORMJ__ preprocessor option:
   - Right-click the LED7SegsTest project and choose **Properties**.
   - In the Properties dialog box, shown in Figure 181, select the **C/C++ Build** option.
   - To set the __PLATFORMJ__ preprocessor flag, select **Preprocessor Options** under "LatticeMico C/C++ Compiler."
   - Click the **add** button next to "LatticeMico32 Preprocessor Defines" and enter __PLATFORMJ__ in the textbox that opens.
2. Use the platformj_linker.ld custom linker script that is made available in the application folder, as shown in Figure 182.

   Refer to the previous example for more information on how to use a custom linker script to compile an application.

Figure 181: Setting the __PLATFORMJ__ Preprocessor Option

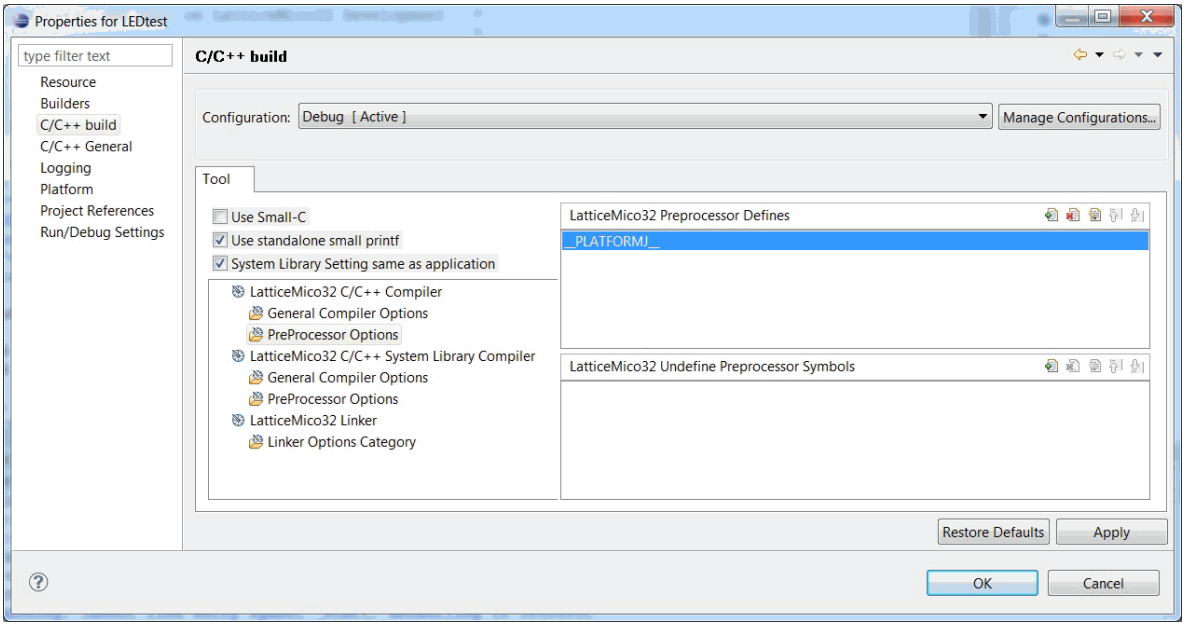
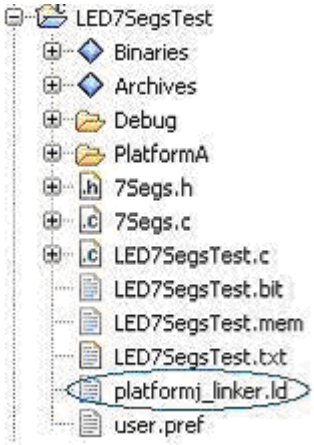


Figure 182: Location of platformj_linker.ld



Conclusion With the tools included in LatticeMico C/C++ SPE, you can selectively extract linker sections from the built executable. Although these examples use the managed build process as its basis, you can employ the tools used to extract the sections and generate memory files for a custom build that may involve user-defined sections and custom-deployment scenarios.

Device Drivers and Multitasking

The drivers provided as part of LatticeMico C/C++ SPE do not provide any inherent thread-protection mechanism to prevent multiple threads from simultaneously accessing the same component instance. However, the provided device drivers are fully re-entrant and can operate on different instances of a component simultaneously from different threads. Therefore, if different tasks do not share component instances, the provided drivers can be safely used in an application. However, if the tasks intend to share a component instance, such as a single UART, you must provide explicit protection from simultaneous access of a component instance across tasks.

Standard-Make Projects

Unlike a managed-build project that automatically generates makefiles and linker scripts, a standard-make project requires you to provide the necessary makefile to build your software application. Since you are required to provide the makefile for building the application, you are responsible for identifying the build rules, as well as selecting source files, file paths, and any other information required for building your application. Standard-make projects allow you flexibility in providing your own make structure, as well as in controlling the build process.

Once you create a standard-make project, the debugger and the deployment tools work the same on the application output as they do with the output of any managed-build project.

To make the transition to a standard-build project, LatticeMico C/C++ SPE provides a LatticeMico library project, which is a managed project. Its sole purpose is to generate platform-dependent source files, makefiles, linker script, and platform library archive. Other than its ability to generate an executable and to provide custom source files for the build, it is identical to the LatticeMico managed-build project.

Table 18 highlights some of the differences between the managed-build, standard-make, and library projects.

Table 18: Differences Between Managed-Build Projects, Standard-Make Projects, and Library Projects

Feature	Managed-Build Project	Library Project	Standard-Build Project
Managed	Yes	Yes	No
Custom makefiles	No	No	Yes
Command-line build	Yes	Yes	Yes
Produce executable	Yes	No	Yes - it is up to you
Produce platform library	Yes	No	Yes - it is up to you
Reference external platform library	No	No	Yes
GUI debugging	Yes	Not applicable	Yes
Deployment tools support	Yes	Not applicable	Yes

Creating a LatticeMico Library Project

Any software project, whether a standard-make project or a managed-build project, depends on the microprocessor platform for basic information, such as the base addresses of the components or the desired standard output device or the linker script parameters like memory sizes and their base addresses. The managed-build process automatically extracts this information in the platform library folder, as described in “Managed Build Process and Directory Structure” on page 145.

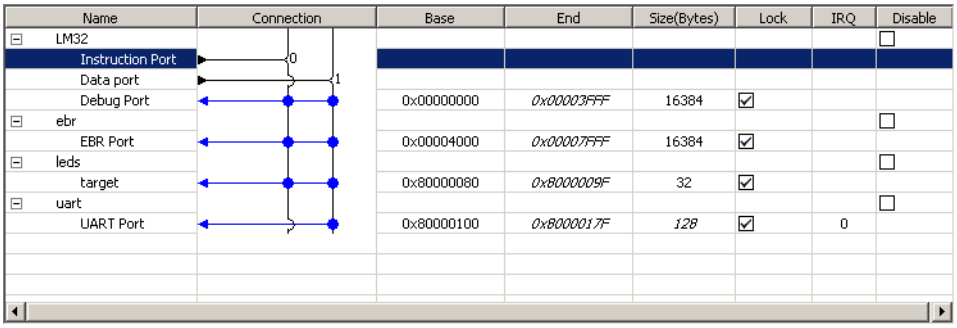
LatticeMico provides another type of project called the LatticeMico library project. It is also a managed project. The sole purpose of this project is to create the platform library folder and its contents. These contents, which are described in “Managed Build Process and Directory Structure” on page 145, include platform-dependent C/C++ source and header files, as well as the platform-dependent linker script and the microprocessor-dependent compiler settings makefile. “Creating a LatticeMico Standard-Make Project” on page 250 describes the commonly required platform-dependent outputs. Once you create and build the LatticeMico library project, its output can be referenced by a standard-make project.

The platform settings used as input for the sample steps in creating a library project are the following:

- Platform name: basic_platform
- Platform location: c:\ispTOOLS\micosystem\examples\basic_platform

Figure 183 shows the platform components and their connectivity. This platform uses a LatticeMico32 microprocessor, on-chip memory named EBR, an 8-bit output GPIO named leds, and a UART named uart.

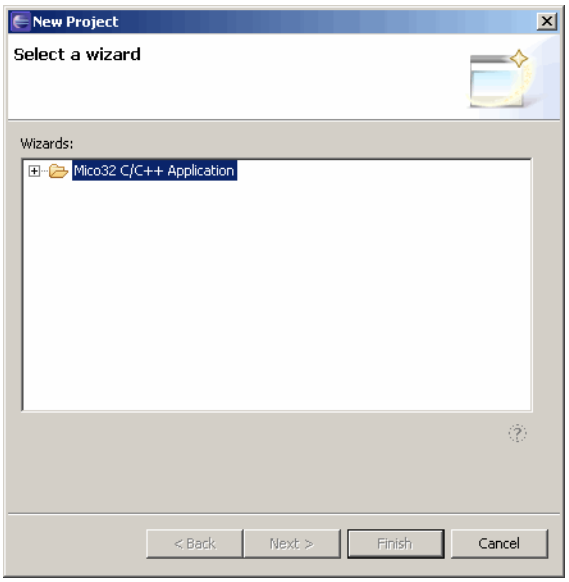
Figure 183: Basic Platform



To create a LatticeMico library project:

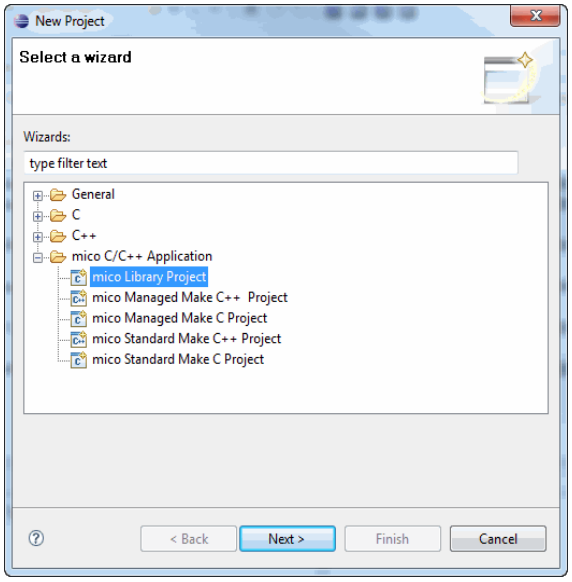
1. Select **File > New > Project** to open the **New Project** dialog box, shown in Figure 184.

Figure 184: New Project Dialog Box



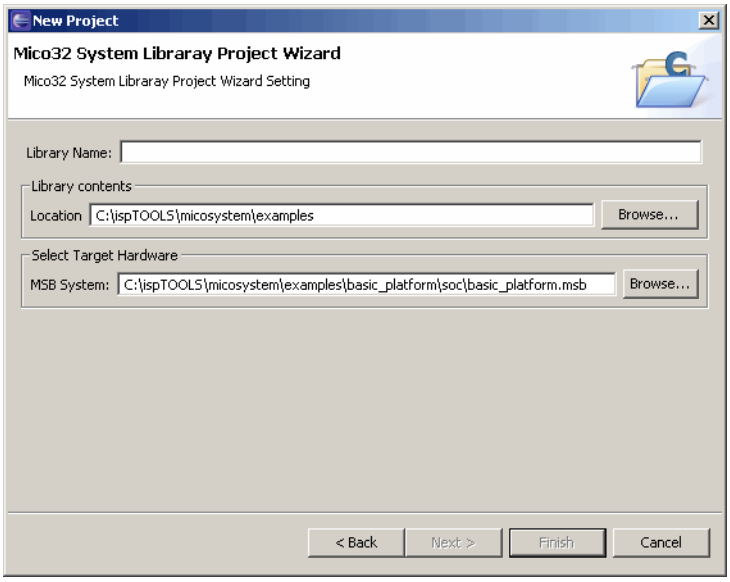
2. Expand the **Mico C/C++ Application** tree, and select **Mico Library Project**, as shown in Figure 185.

Figure 185: Selecting a LatticeMico Library Project



3. Click **Next**.
4. In the New Project dialog box, shown in Figure 186, enter the following information:
   - Library Name: basic_platform_lib
   - Library Contents Location: c:\standard_make\basic_platform_lib.

Figure 186: Library Name and Location



This step creates the standard_make directory, if it does not already exist. It then creates the basic_platform_lib directory, if it does not exist. The files for this project will be located in this basic_platform_lib directory.

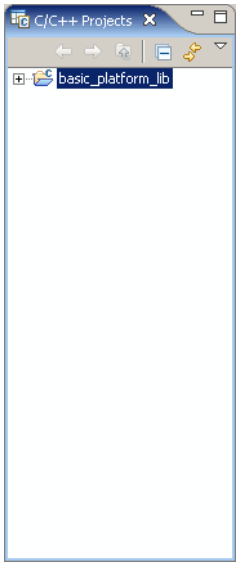
5. Use the **Browse** button to select the appropriate platform in the MSB System box.

   For this example, the platform is basic_platform.msb and is located in c:\ispTOOLS\micosystem\examples\basic_platform\soc\ directory.

6. Click **Finish** after entering the information just given.

   Figure 187 shows the C/C++ Projects view in the C/C++ perspective after you successfully complete step 4.

Figure 187: C/C++ Projects View After Creation of Platform Library Project



7. Select the basic_platform_lib project and build it by selecting **Project > Build Project**.

Note

The LatticeMico Library Project properties, accessible by selecting Project > Properties, have property tabs similar to those of a managed build project. You can change platform properties by selecting the Platform tab in the Properties dialog box.

The contents of the platform library project are identical to those of a managed-build project. Refer to “Managed Build Process and Directory Structure” on page 145 for information on the contents of a managed-build project. The managed-build project generates the platform library archive file, compiles the user-provided source files, and generates an .elf file.

The LatticeMico library project generates only the platform library archive file. It does not compile the user-provided source files, nor does it generate an

executable. This archive file and platform-dependent generated source and makefiles can then be referenced by a standard-make project, allowing the standard-make application flexibility in being platform-independent to a great extent. They also give several standard-make applications the ability to reference a single platform library archive file.

Figure 188 displays the contents of the application output folder.

Figure 188: Contents of the Platform Library Application Output Folder

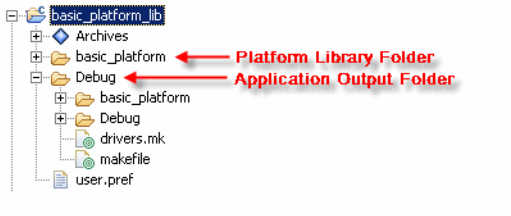
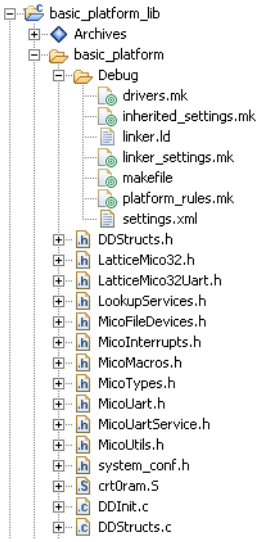


Figure 189 shows the contents of the platform library folder. It does not show a complete list of C source files.

Figure 189: Contents of the Platform Library Folder



As shown in Figure 189, the contents of this platform library folder are identical to those of the managed-build project's platform library folder. As the next step demonstrates, these contents can be referenced from a standard-build project.

As shown in Figure 190, the platform library archive file is located in the application output folder's build-configuration directory. Its contents are also identical to those of the managed-build project's output folder.

Figure 190: Platform Library Archive File

You have now successfully built a platform library project and have all the platform-dependent outputs, including makefiles, linker scripts, source files, and microprocessor-specific compiler settings file, that are used in a managed-build project. This project's output is linked to the platform, so whenever a change is made to the platform in the LatticeMico MSB, the project build will reflect changes made to the platform. The platform library project is therefore dependent on the microprocessor platform.

Creating a LatticeMico Standard-Make Project

In the previous section, you created a platform library project. In this section, you will create a standard-make project for the basic_platform platform. A standard make does not provide any facility to extract platform-dependent information, unless you provide the necessary scripts and information as part of the makefiles, which you must also provide. However, it is possible to

reference other projects from a standard-make project. This section describes the steps required for creating such a project and for referencing the generated LatticeMico library project for platform-dependent information.

Note

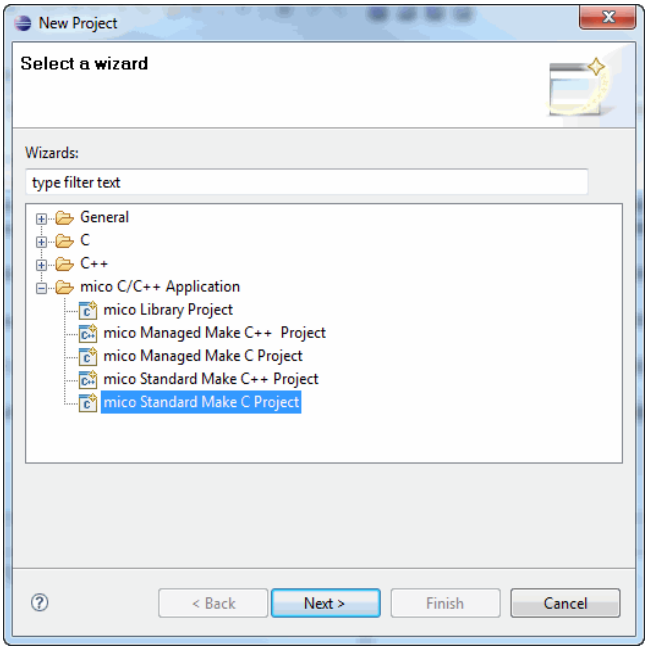
A standard-make project is an Eclipse CDT project type. This project type has comprehensive configuration and customization options through various project-property settings. For the simple goal of creating a LatticeMico project that has a user-provided makefile, you do not need to set the various project-property options.

Avoid setting properties from within the GUI. Instead, place them in the custom makefile so that you can build this project from a LatticeMico SDK shell (that is, as command-line build). Follow the steps listed in this chapter to create a standard-make project that avoids these project-property settings.

To create a standard-make project:

1. Select **File > New > Project** in the C/C++ SPE perspective to display the New Projects dialog box, shown in Figure 191.

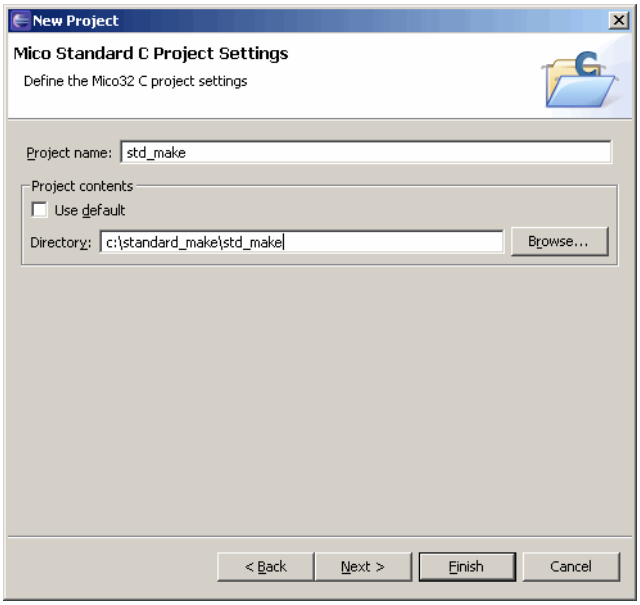
Figure 191: New Projects Dialog Box



2. Select **Mico Standard Make Project** and click **Next**.
3. Deselect **Use Default** and enter the following information, as shown in Figure 192:
   - Project Name: std_make

- Directory: c:\standard_make\std_make

Figure 192: Standard Make Creation Settings



The LatticeMico library project was created in the c:\standard_make\basic_platform_lib directory, so creating this project in the c:\standard_make\std_make directory makes it convenient to reference the library project contents from this directory.

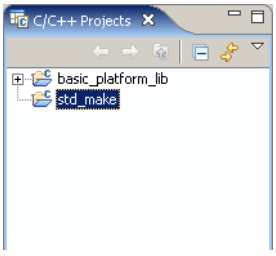
4. Click **Finish**.

Note

Refer to the Eclipse/CDT documentation if you choose to explore other selections by clicking the Next button.

Figure 193 shows the C/C++ Projects View after you click the Finish button.

Figure 193: C/C++ Projects View



Creating an Application Source File

At this stage, your project is empty. There are no source files and no make file. You must provide all this information for your application.

Start by creating a simple application source file.

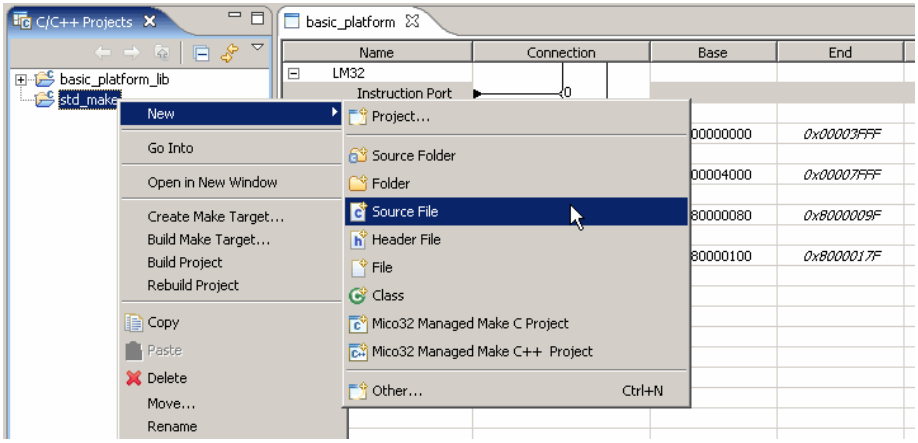
Note

You can copy the code shown in this section from the sample makefiles in the <*install_path*>/micosystem/utilities/templates/std_mk_makefile_sample/ directory.

To create an application source file:

1. Click on **std_make** in the C/C++ Projects View.
2. Right-click and select **New > Source File**, as shown in Figure 194.

Figure 194: Adding a New Source File



3. In the Source File box of the New Source File dialog box, enter **hello_world.c**, as shown in Figure 195, and click **Finish**.

Figure 195: Entering hello_world.c in New Source File Dialog Box

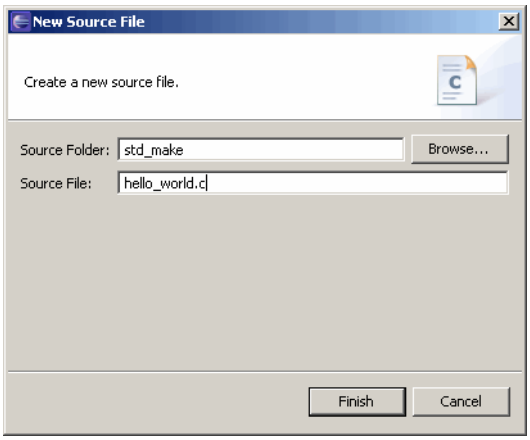
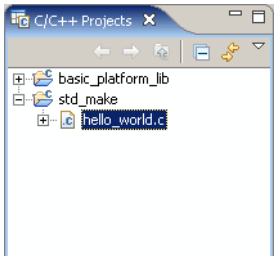


Figure 196 shows the C/C++ Projects View with the new source file.

Figure 196: C/C++ Projects View



At this stage, your source file is an empty source file.

4. Enter the code shown in Figure 197 in the hello_world.c source file.

Figure 197: Code in hello_world.c Source File

```
#include "system_conf.h"

int main(void)
{
  volatile unsigned char *LEDS;
  volatile unsigned char iValue = 0x01;
  unsigned char iShiftLeft = 1;

  LEDS = (volatile unsigned char *)LED_BASE_ADDRESS;

  while(1){
    printf("hello world\n");
    *LEDS = ~iValue;
    if(iShiftLeft == 1){
      iValue = iValue << 1;
      if(iValue == 0x80){
        iValue = 0x40;
        iShiftLeft = 0;
      }
    }else{
      iValue = iValue >> 1;
      if(iValue == 0){
        iValue = 0x02;
        iShiftLeft = 1;
      }
    }
    MicoSleepMilliSecs(1000);
  }
  return(0);
}
```

You now have a source file. The included file, system_conf.h, provides the base address value LEDS_BASE_ADDRESS.

Creating a Source-Identification Makefile

You still cannot build this application, because you do not have a makefile that tells Eclipse/CDT the rules to build this application. To keep the build process clean, you will add several makefiles. First, you will add a makefile that identifies the sources required to build your application.

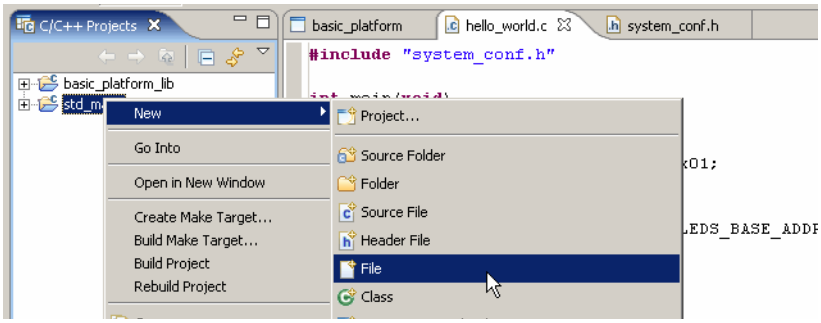
Note

You can copy the code shown in this section from the sample makefiles in the *<install_path>*/micosystem/utilities/templates/std_mk_makefile_sample/ directory.

To create a source-identification makefile:

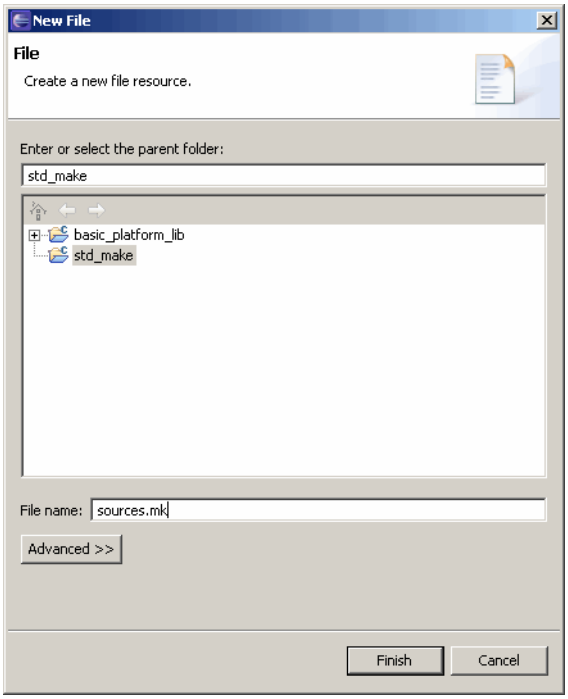
1. Select the **std_make** project in the C/C++ view.
2. Right-click and select **New > File** from the pop-up menu, as shown in Figure 198.

Figure 198: Adding a Makefile



3. In the New File dialog box, enter the file name, sources.mk, as shown in Figure 199, and click **Finish**.

Figure 199: Naming the sources.mk Makefile



This step creates an empty file named sources.mk in the C/C++ Projects View for the std_make project.

You will now add information to this sources.mk file to identify the sources needed to build this simple application.

The following description explains the contents in the sources.mk file. The contents of this sources.mk file are provided at the end of this description. One C source file, hello_world.c, forms the bulk of the application. Additionally, the application requires the boot code that will invoke the application's "main" function. This boot code is contained in the crt0ram.S file in the LatticeMico library project.

4. Add the lines shown in Figure 200 to the sources.mk file.

Figure 200: Identifying the Sources

```
#-------------------------------------------------------------
#- The source files that you want to compile
#- - main.c
#- - crt0ram.S (boot code) provided by platform library
#- CXX_SRCS are the .cpp source files (C++)
#- C_SRCS are the .c source files (C)
#- .S and .s are assembly source files for LatticeMico32
#-------------------------------------------------------------
#- C++ sources (.cpp)
CXX_SRCS=

#- C sources (.c)
C_SRCS=main.c

#- Assembly source files (.s and .S)
ASM_SRCS=crt0ram.S
```

Next, you must help the build process identify where it can find the crt0ram.S source file. This file is located in the platform library project's platform library folder in the c:\standard_make\basic_platform_lib\basic_platform directory.

5. Add the line shown in Figure 201 to modify the VPATH variable. VPATH is a make variable that specifies a list of directories that should be searched for finding sources. Refer to the GNU make documentation for information on VPATH and other application variables.

Figure 201: Modifying the VPATH Variable

```
#-------------------------------------------------------------
# Specify where this project can find platform-
# specific source files that are required for your build (in
# this case, the only such file is crt0ram.S.)
#-------------------------------------------------------------
VPATH+=$(PLATFORM_LIB_PATH)/$(PLATFORM_NAME)/
```

You have not yet defined the PLATFORM_LIB_PATH and PLATFORM_NAME variables, but you will use them to indicate the following:

- PLATFORM_LIB_PATH – Root directory of the LatticeMico library project (in this case, c:\standard_make\basic_platform_lib)
- PLATFORM_NAME – Name of the platform (in this case, basic_platform)

You will define these two variables in the next step in a separate makefile.

The system_conf.h header file is included in the hello_world.c source file. You must indicate where the related header files are located.

6. Add the lines shown in Figure 202 to the sources.mk file.

Figure 202: Specifing Location of Header Files

```
#-------------------------------------------------------------
# In case the source files include header files provided by
# the platform library project, specify where these can be
# found.
#-------------------------------------------------------------
INCLUDE_PATH+=$(PLATFORM_LIB_PATH)/$(PLATFORM_NAME)/
```

The sources.mk file is now complete. It provides information on the sources required to build the application and identifies where the external sources (crt0ram.S) and the included files (system_conf.h) can be found.

Figure 203 lists the complete contents of sources.mk file.

Figure 203: Contents of the sources.mk File

```
#-------------------------------------------------------------
#- The source files that you want to compile:
#- - main.c
#- - crt0ram.S (boot code) provided by platform library
#- CXX_SRCS are the .cpp source files (C++)
#- C_SRCS are the .c source files (C)
#- .S and .s are assembly source files for LatticeMico32
#-------------------------------------------------------------
#- C++ sources (.cpp)
CXX_SRCS=

#- C sources (.c)
C_SRCS=hello_world.c

#- Assembly source files (.s and .S)
ASM_SRCS=crt0ram.S

#-------------------------------------------------------------
# Specify where this project can find platform-
# specific source files that are required for your build (in
# this case, the only such file is crt0ram.S.)
#-------------------------------------------------------------
VPATH+=$(PLATFORM_LIB_PATH)/$(PLATFORM_NAME)/


#-------------------------------------------------------------
# In case your source files include header files provided by
# the platform library project, specify where these can be
# found.
#-------------------------------------------------------------
INCLUDE_PATH+=$(PLATFORM_LIB_PATH)/$(PLATFORM_NAME)/
```

Creating the Platform-Settings Makefile

In the sources.mk makefile, you identified some sources and header files that are platform-dependent and are generated by the LatticeMico library project. These, however, are not the only file types that you need. You also need the following information, which is contained in the LatticeMico library project:

- CPU configuration-dependent compiler settings, which are contained in the platform_rules.mk file of the LatticeMico library project. It is defined by the CPU_CONFIG variable.
- Linker file – At this stage, you will reference the automatically generated linker script contained in the LatticeMico library project.

Also, since you are using the default boot code contained in the automatically generated crt0ram.S file, you must include the platform library archive file for routines that it may need to invoke.

Note

You can copy the code shown in this section from the sample makefiles in the <*install_path*>/micosystem/utilities/templates/std_mk_makefile_sample/ directory.

To add a makefile containing the configuration-dependent compiler settings:

1. Create a new file, platform.mk, in the std_make project.
2. In this platform.mk file, enter the information shown in Figure 204. The following description explains the contents in the platform.mk file. The contents of this platform.mk file are provided at the end of this description.

Figure 204: Defining Platform-Specific Settings

```
#
#Define platform-specific settings that can be quickly changed.
#
PLATFORM_NAME=basic_platform
PLATFORM_LIB_PATH=../basic_platform_lib
PLATFORM_BLD_CFG=debug
```

The lines shown in Figure 204 identify the platform name, the directory containing the LatticeMico library project, and the build configuration of the LatticeMico library project that you want to reference. You can retarget your application to a different library by modifying this basic information.

3. Next, you must identify the items from the LatticeMico library project that you want to reference, so enter the information shown in Figure 205 in this file.

Figure 205: Identifying Items from the LatticeMico Library Project

```
#
# Derive other information from the basic platform
# information as required by main makefile.
#

# 1. Specify where these platform-dependent makefiles are
#    located.
PLATFORM_MAKEFILES_DIR = $(addprefix $(PLATFORM_LIB_PATH),\
  $(addprefix /$(PLATFORM_NAME), /$(PLATFORM_BLD_CFG)))


# 2. Platform library (relative path and name)
#    required by main makefile.
PLATFORM_LIBRARY=$(addprefix $(PLATFORM_LIB_PATH)/,\
  $(addprefix $(PLATFORM_BLD_CFG)/,\
  $(addprefix $(PLATFORM_BLD_CFG)/, lib$(PLATFORM_NAME).a)))


# 3. Linker file required by main makefile
LD_FILE=$(PLATFORM_MAKEFILES_DIR)/linker.ld


# 4. $(CPU_CONFIG) defines CPU-specific configuration.
# CPU-specific configuration is platform-dependent, so you
# put it in this file.
# Platform_rules.mk contains CPU configuration.
include $(PLATFORM_MAKEFILES_DIR)/platform_rules.mk
```

You have now specified the platform-dependent information for your application. Figure 206 lists the complete contents of the platform.mk makefile.

Figure 206: Contents of platform.mk File

```
#
# Define platform-specific settings that can be
# quickly changed.

#
PLATFORM_NAME=basic_platform
PLATFORM_LIB_PATH=../basic_platform_lib
PLATFORM_BLD_CFG=debug


#
# Derive other information from the basic platform
# information as required by main makefile
#

# 1. Specify where these platform-dependent makefiles are
#    located.
PLATFORM_MAKEFILES_DIR = $(addprefix $(PLATFORM_LIB_PATH),\
  $(addprefix /$(PLATFORM_NAME), /$(PLATFORM_BLD_CFG)))


# 2. Platform library (relative path and name),
#    required by main makefile.
PLATFORM_LIBRARY=$(addprefix $(PLATFORM_LIB_PATH)/,\
  $(addprefix $(PLATFORM_BLD_CFG)/,\
  $(addprefix $(PLATFORM_BLD_CFG)/, lib$(PLATFORM_NAME).a)))


# 3. Linker file required by main makefile
LD_FILE=$(PLATFORM_MAKEFILES_DIR)/linker.ld


# 4. $(CPU_CONFIG) defines CPU-specific configuration.
# CPU-specific configuration is platform-dependent, so you
# put it in this file.
# Platform_rules.mk contains CPU configuration.
include $(PLATFORM_MAKEFILES_DIR)/platform_rules.mk
```

Creating C Compiler and Linker Settings Makefile

You must now provide information for the compiler tool chain. You will create a separate file, settings.mk, which contain this information instead of putting it in the main makefile, which is yet to be created. This way, you can change your build settings without having to search the main makefile.

Note

You can copy the code shown in this section from the sample makefiles in the <*install_path*>/micosystem/utilities/templates/std_mk_makefile_sample/ directory.

To create a makefile that defines the settings for the C compiler and the linker:

1. Create a new file named settings.mk.

   You will first identify your application.

   The following description explains the contents in the settings.mk file. The contents of this settings.mk file are provided at the end of this description.

2. Enter the lines shown in Figure 207 in this new settings.mk file. Your application will be called hello_world.elf.

Figure 207: Identifying Your Application

```
#-------------------------------------------------
#
# Name your output executable here:
# APP_OUTPUT_ELF will be used by the main makefile,
# so it should not be renamed to something else.
#
#-------------------------------------------------
APP_OUTPUT_ELF=hello_world.elf
```

3. To ensure that the build process places its output and intermediate outputs in a separate temporary directory so that it does not clutter your main project directory, enter the lines shown in Figure 208 in the settings.mk file.

Figure 208: Specifying an Output Directory

```
#-------------------------------------------------
# You do not want the intermediate and final outputs
# that will deleted by a clean to clutter
# this project folder, so you specify an output
# directory.
#-------------------------------------------------
OUTPUT_DIR=./output
```

Now you define the compiler and linker flags that you want to pass to the compiler and linker when building your application. The LatticeMico library project compiler settings are located in the inherited_settings.mk makefile in the LatticeMico library project. You will not reference these settings, so the compilation rules for the application will be separated from the library. If you wanted to use the same settings as the library, you could include the

inherited_settings.mk makefile, just as you included platform library outputs in the platform.mk file.

Figure 209: Setting Compiler, Assembler, and Linker Flags

```
#-----------------------------------------------
# Provide compiler, assembler, and linker flags that will
# be used when building the application.
# Note: These flags do not affect the platform
# library if this project references the
# platform library.
#-----------------------------------------------
# CFLAGS affect C compiler: (standard gcc flags)
# 1. You want functions to be in their own sections for
#    size reduction (-ffunction-sections)
# 2. You want no optimization (-O0)
# 3. You want debug symbols (-g2)
# 4. You want warnings (-w)
# 5. You want to use the standalone printf
# 6. You want to define __lm32__ preprocessor definition
CFLAGS= -ffunction-sections -O0 -g2 -w
#Add preprocessor definitions to CFLAGS to include
# standalone printf implementation (-D_USE_LSCC_PRINTF_)
CFLAGS+=-D_USE_LSCC_PRINTF_ -D__lm32__


# LDFLAGS affect linker:
# Since you are using lm32-elf-gcc (and not lm32-elf-ld),
# you must specify -Wl before the linker flag.
# (refer to GCC documentation)
# Delete sections that are not used
# (thereby making your executable more compact).
LDFLAGS +=-Wl,--gc-sections
```

4. Specify the C library that you plan to use by adding the self-explanatory lines shown in Figure 210 to this file.

Figure 210: Specifying the C Library to Use

```
# Define which C library you want to use.
# You have two choices according to what Lattice provides:
# 1. -lsmallc (small Newlib C library)
# 2. -lc (complete C library)
C_LIB=-lsmallc
```

You now have completed the settings file that defines the settings for the C compiler and linker. Figure 211 lists the complete contents of this file.

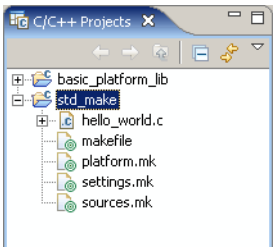
Figure 211: Contents of the settings.mk File

```
#----------------------------------------------
#
# Name your output executable here:
# APP_OUTPUT_ELF will be used by the main makefile,
# so it should not be renamed to something else.
#
#----------------------------------------------
APP_OUTPUT_ELF=hello_world.elf


#----------------------------------------------
# You do not want the intermediate and final outputs
# that will be deleted by a clean to clutter
# this project folder, so you specify your output
# directory.
#----------------------------------------------
OUTPUT_DIR=./output

#----------------------------------------------
# Provide compiler, assembler, and linker flags that will
# be used when building the application.
# Note: These flags do not affect the platform
# library if this project references the
# platform library.
#----------------------------------------------
# CFLAGS affect C compiler: (standard gcc flags)
# 1. You want functions to be in their own sections for
#    size reduction (-ffunction-sections)
# 2. You want no optimization (-O0)
# 3. You want debug symbols (-g2)
# 4. You want warnings (-w)
# 5. You want to use the standalone printf
# 6. You want to define __lm32__ preprocessor definition
CFLAGS= -ffunction-sections -O0 -g2 -w
#Add preprocessor defines to CFLAGS
# standalone printf implementation (-D_USE_LSCC_PRINTF_)
CFLAGS+=-D_USE_LSCC_PRINTF_ -D__lm32__

# LDFLAGS affect linker:
# Since you are using lm32-elf-gcc (and not lm32-elf-ld),
# you must specify -Wl, before the linker flag
# (refer to gcc documentation)
# Delete sections that are not used
# (thereby making your executable more compact)
LDFLAGS +=-Wl,--gc-sections

# Define which C library you want to use.
# You have two choices according to what Lattice provides:
# 1. -lsmallc (small Newlib C library)
# 2. -lc (complete C library)
C_LIB=-lsmallc
```

Creating the Main Makefile

At this point, you have most of the necessary pieces to build your application. The last step is defining the rules for building your application. For this, you will create the main makefile, which is named “makefile.” Figure 212 shows the contents of the project folder with all these makefiles.

Figure 212: Project Folder Containing Makefiles Created



In this makefile, you will define the rules for building your application. First, you will include the other makefiles that you have defined so far.

Note

You can copy the code shown in this section from the sample makefiles in the <*install_path*>/micosystem/utilities/templates/std_mk_makefile_sample/ directory.

To create the main makefile:

1. Enter the lines shown in Figure 213.

Figure 213: Including Other Makefiles

```
#
# Include settings.mk that contain settings
# required by this makefile.
#
include settings.mk
include platform.mk

#
# Include makefile that contains the sources
# information, that is, sources to build and the
# locations where these sources can be found.
#
include sources.mk
```

2. You must prefix –I to the include paths defined in the sources.mk file, as required by the compiler. Add the lines of code shown in Figure 214 to the “makefile” makefile.

Figure 214: Prefixing Include Paths

```
#
# You must prefix -I to each of the include paths,
# then make this available to the compiler.
# Modify CFLAGS to contain the include path.
#
CFLAGS += $(foreach inc_path, $(INCLUDE_PATH), -I$(inc_path))
```

You defined your C/C++ and assembly source inputs in sources.mk.

3. Enter the lines of code shown in Figure 215 to identify the required objects by replacing the file extensions with .o extensions.

Figure 215: Identifying .o Object Files

```
#-------------------------------------------------------------
# Now that you have defined your .c, .s, .S and .cpp sources,
# extract the .o files that are needed for a successful build.
#-------------------------------------------------------------
OBJS=$(sort $(C_SRCS:.c=.o)\
  $(patsubst %.cxx, %.o, \
  $(patsubst %.cc, %.o, $(patsubst %.cpp, %.o, \
  $(patsubst %.C, %.o,$(CXX_SRCS)))))\
  $(patsubst %.S, %.o, $(patsubst %.s, %.o, $(ASM_SRCS))))
```

Now that you have identified the .o object files, you must decide where these files will be created and found. They must be created in the output directory that you defined in the settings.mk file.

4. Place the lines shown in Figure 216 in the “makefile” file to indicate the location of the .o object files.

Figure 216: Specifying the Location of the .o Object Files

```
#-------------------------------------------------------
# Prefix the output directory to the .o files,
# because that is where you want these .o files to be created
# and where the build process can find these created objects
# for the .elf build.
#-------------------------------------------------------
APP_OBJS=$(addprefix $(OUTPUT_DIR)/, $(OBJS))
```

5. Add the lines shown in Figure 217 to define where the application executable (.elf) will be created.

Figure 217: Specifying the Location of the .elf Executable

```
# we also modify where we want our .elf to be created
APP_ELF=$(addprefix $(OUTPUT_DIR)/, $(APP_OUTPUT_ELF))
```

Now you will define the build rules. The build process invokes make with “clean” when performing a clean and “all” when performing a build. When performing a rebuild, the build process invokes make with “clean” followed by “all.”

6. Enter the following lines to define your rule for “clean.”

   During a “clean,” you want all files removed from the output directory, then you delete the output directory.

   You have not yet defined the “all” and “dummy” rules, but you will define them next. Refer to GNU make documentation for the .PHONY keyword.

Figure 218: Specifying Cleaning Rule

```
.PHONY: all dummy clean
#-------------------------------------------------------
#
# Rule to clean this project.
# You just want to delete items that you created when building,
# including the output directory.
#
#-------------------------------------------------------
clean:
    @echo cleaning...
    rm -r -f $(OUTPUT_DIR)
```

Be very careful to insert tabs for each line instead of white spaces for the lines given in Figure 218. Refer to the GNU make documentation for an explanation of the interpretation of white spaces (tabs versus spaces).

Now you will define your rule for building your application. You must first create an output directory, if one does not exist, then invoke the compiler on each object file. First you will create the output directory, then compile. Enter the lines shown in Figure 219 to define this first step of creating the temporary directory. These lines comprise the “dummy” rule.

Figure 219: Specifying the Rule for Building the Output Directory

```
# Dummy rule prepares for a build, such as creating the output
# directory.
dummy:
    @mkdir -p $(OUTPUT_DIR)
    @echo $(APP_OBJS)
    @echo $(VPATH)
```

7. Enter the lines shown in Figure 220 to define the “all” rule.

Figure 220: Specifying the “all” Rule

```
#
# “All” rule defines how to build the output desired.
#
all: dummy $(APP_ELF)
```

The statement in Figure 220 says that “all” depends on “dummy” and on APP_ELF. “Dummy” is the rule for creating the output directory. You will now define the APP_ELF rule that actually builds the application.

8. Enter the lines shown in Figure 221 for the APP_ELF rule.

Figure 221: Specifying the Rule that Builds the Application

```
# Define a function to build the application.
# # documented source
define build_app
    lm32-elf-gcc\
    $(CPU_CONFIG)\
    -T $(LD_FILE)\
    -o$1  \
    $(APP_OBJS)\
    $(PLATFORM_LIBRARY)\
    -lm       \
    $(C_LIB)\
    -lgcc   \
    $(PLATFORM_LIBRARY)\
    -lnosys \
    $(LDFLAGS)
endef

#
# Define how to build an .elf file.
# This depends on the objects required, as well as
# the CPU configuration makefile.
# That is, if any change is detected, rebuild the
# elf file.
#
$(APP_ELF): $(APP_OBJS) $(PLATFORM_RULES_MAKEFILE) $(LD_FILE)\
$(PLATFORM_LIBRARY)
  @echo
  @echo
  @echo building $(APP_ELF)
  $(call build_app,$@)
  lm32-elf-size $(APP_ELF)
  lm32-elf-objdump -d $(APP_ELF) > $(OUTPUT_DIR)/dump.txt
```

The call to build_app is the command that builds the executable. This command expects all the .o files to be present. Declaring the APP_ELF rule dependent on the application object (.o) files generates the .o files. Additionally, making this APP_ELF rule dependent on the platform rules makefile, the linker script, and the platform library archive file ensures that this application is built each time that any of these three files changes.

The APP_ELF rule causes the objects to be generated the first time that this application is built (or each time it is rebuilt). You do not want to apply a default rule for building these objects, so add the following lines to build the .o files, using the following explicit rules.

Figure 222: Specifying the Rule for Building the .o Object Files

```
#-------------------------------------------------------------
# Create a generic rule to build .o files from .c files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.c
   @echo
   @echo compiling $< to $@
   @echo
   lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o$@



#-------------------------------------------------------------
# Create a generic rule to build .o files from .S files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.S
   @echo
   @echo compiling $< to $@
   @echo
   lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o$@
```

You now have defined your main makefile. Figure 223 shows the entire makefile source.

Figure 223: Contents of the Main Makefile

```
#
# Include settings.mk that contain settings
# required by this makefile.
#
include settings.mk
include platform.mk


#
# Include makefile that contains the sources
# information, that is, sources to build and the
# locations where these sources can be found.
#
include sources.mk


#
# Prefix -I to each of the include paths,
# then make this available
# to the compiler. Modify CFLAGS to
# also contain the include path.
#
CFLAGS += $(foreach inc_path, $(INCLUDE_PATH), -I$(inc_path))
```

Figure 223: Contents of the Main Makefile (Continued)

```
#-------------------------------------------------------------
# Now that you have defined your .c, .s, .S and .cpp sources,
# extract the .o files that are needed for a successful build.
#-------------------------------------------------------------
OBJS=$(sort $(C_SRCS:.c=.o)\
  $(patsubst %.cxx, %.o, \
  $(patsubst %.cc, %.o, $(patsubst %.cpp, %.o, \
  $(patsubst %.C, %.o,$(CXX_SRCS)))))\
  $(patsubst %.S, %.o, $(patsubst %.s, %.o, $(ASM_SRCS))))
#-----------------------------------------------------
# Prefix the output directory to the .o files
# because that is where you want these .o files to be created
# and where the build process can find these
# created objects for the .elf build.
#-----------------------------------------------------
APP_OBJS=$(addprefix $(OUTPUT_DIR)/, $(OBJS))


# Modify where you want the .elf to be created.
APP_ELF=$(addprefix $(OUTPUT_DIR)/, $(APP_OUTPUT_ELF))

.PHONY: all dummy clean
#-----------------------------------------------------------
#
# Rule to clean this project.
# You want to delete items created when building,
# including the output directory.
#
#-----------------------------------------------------------
clean:
  @echo cleaning...
  rm -r -f $(OUTPUT_DIR)
```

Figure 223: Contents of the Main Makefile (Continued)

```
# Dummy rule prepares for a build such as creating the output
# directory.
dummy:
  @mkdir -p $(OUTPUT_DIR)
  @echo $(APP_OBJS)
  @echo $(VPATH)


#
# "All" rule defines how to build the output desired.
#
all: dummy $(APP_ELF)


# Define a function to build application.
define build_app
  lm32-elf-gcc\
    $(CPU_CONFIG)\
    -T $(LD_FILE)\
    -o$1  \
    $(APP_OBJS)\
    $(PLATFORM_LIBRARY)\
    -lm       \
    $(C_LIB)\
    -lgcc   \
    $(PLATFORM_LIBRARY)\
    -lnosys \
    $(LDFLAGS)
endef

#
# Define how to build the .elf file.
# This depends on the objects required, as well as
# the CPU configuration makefile.
# That is, if any change is detected, rebuild the
# elf file.
#
$(APP_ELF): $(APP_OBJS) $(PLATFORM_RULES_MAKEFILE) $(LD_FILE)\
$(PLATFORM_LIBRARY)
  @echo
  @echo
  @echo building $(APP_ELF)
  $(call build_app,$@)
  lm32-elf-size $(APP_ELF)
  lm32-elf-objdump -d $(APP_ELF) > $(OUTPUT_DIR)/dump.txt


#-------------------------------------------------------------
# Create a generic rule to build .o files from .c files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.c
  @echo
  @echo compiling $< to $@
  @echo
  lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Figure 223: Contents of the Main Makefile (Continued)

```
#-------------------------------------------------------------
# Create a generic rule to build .o files from .S files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.S
  @echo
  @echo compiling $< to $@
  @echo
  lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Building the Project

Now that you have provided a makefile, the application is all ready to be built, rebuilt, or cleaned.

To build the project:

- Select **Project > Build Project** to build the project.

The preceding sections introduced the standard-make project. They referenced a LatticeMico library project to help you get started in creating a standard make. However, you are not required to reference a LatticeMico library project if you choose to create a completely stand-alone standard-make project. In this case, you must supply all the necessary pieces of information, such as the linker file, the boot code, and any other information your application may require.

Linking the LatticeMico Library Project as a Dependency on the Standard-Make Project

To conclude the example, this section reviews the steps required to make the standard-make project dependent on the library project. Performing a clean on the standard make project also cleans up the library project.

Making the LatticeMico library project a dependency on the standard-make project has the following advantages:

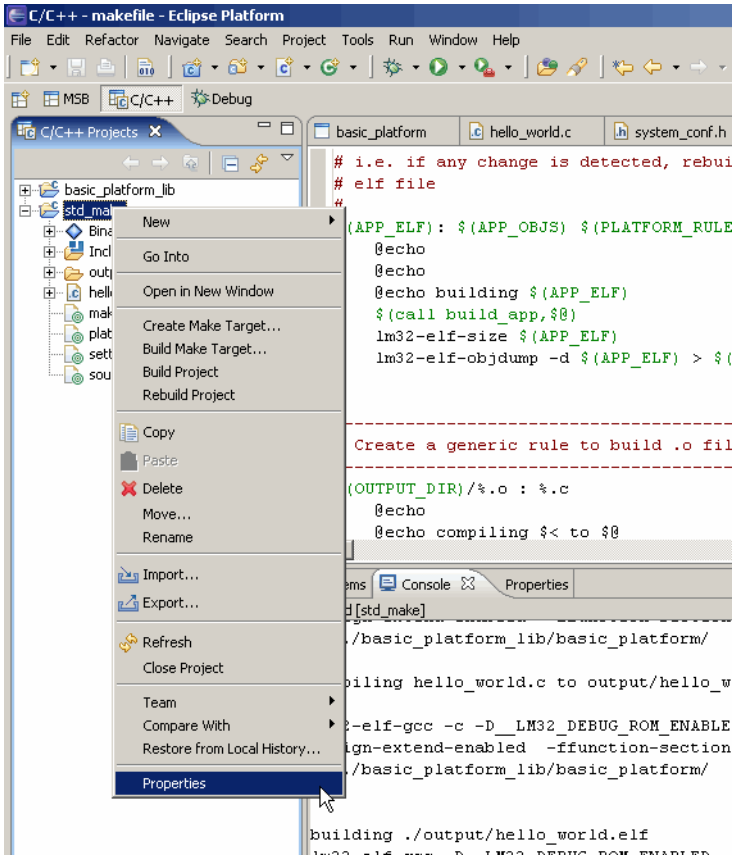
- If any of the files that the LatticeMico library project depends on is changed, building the standard-make project causes the LatticeMico library project to be built, followed by updating the standard-make project.
- Rebuilding the standard-make project causes the LatticeMico library project to be rebuilt, followed by rebuilding the standard-make project.

To establish the LatticeMico library project as a dependency on the standard-make project:

1. Click on the **std-make** project in the C/C++ SPE view.

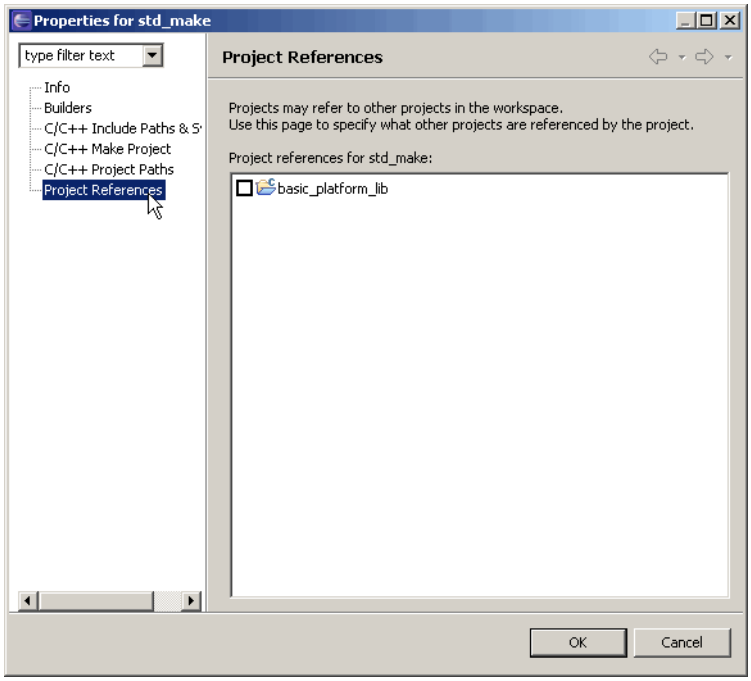
2. Right-click and select **Properties**, as shown in Figure 224.

Figure 224: Selecting the Properties Command



3. In the Properties for Std_make dialog box, select the **Project References** tab, as shown in Figure 225.

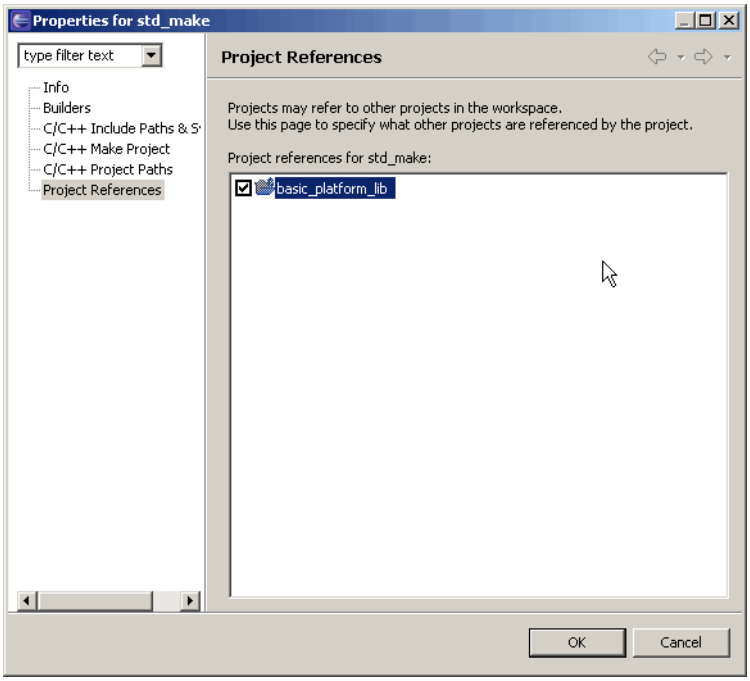
Figure 225: Selecting the Project References Tab



The pane to the right lists all available projects in the C/C++ SPE projects view for the given workspace.

4. Select the platform library that this standard make project references, and select the option next to the project list, as shown in Figure 226.

Figure 226: Selecting the Platform Library



You have now linked the LatticeMico library project to the standard-make project. Each time that a build is performed on the standard-make project, the build system first builds the project library, if there are any changes specified in the project library makefiles, before building the standard-make project.

The example makefile is written so that any manual change to the crt0ram.S file contained in the LatticeMico library project causes the standard-make project to rebuild itself to account for the change. However, it does not detect a manual change to the system_conf.h header file. If you want a manual change to this header file to be detected, you must modify the standard-make project's makefile to generate dependency information and perform a build accordingly. The modified makefile shown in Figure 227 accomplishes this. The method used in it is from Robert Mecklenburg's book, *Managing Projects*

with GNU Make (Sebastopol, CA: O'Reilly Media, Inc., 2004) and is a very good introduction, as well as reference, to GNU make.

Figure 227: Modified Makefile

```
#
# Include settings.mk that contain settings
# required by this makefile.
#
include settings.mk
include platform.mk


#
# Include makefile that contains the sources
# information, that is, sources to build and the
# locations where these sources can be found.
#
include sources.mk


#
# Prefix -I to each of the include paths,
# then make this available
# to the compiler. Modify CFLAGS to
# also contain the include path.
#
CFLAGS += $(foreach inc_path, $(INCLUDE_PATH), -I$(inc_path))

#---------------------------------------------------------------
# Now that you have defined your .c, .s, .S and .cpp sources,
# extract the .o files that are needed for a successful build.
#---------------------------------------------------------------
OBJS=$(sort $(C_SRCS:.c=.o)\
  $(patsubst %.cxx, %.o, \
  $(patsubst %.cc, %.o, $(patsubst %.cpp, %.o, \
  $(patsubst %.C, %.o,$(CXX_SRCS)))))\
  $(patsubst %.S, %.o, $(patsubst %.s, %.o, $(ASM_SRCS))))

#-----------------------------------------------------
# Prefix the output directory to the .o files to indicate
# where to create the .o files
# and where the build process can find these
# created objects for the .elf build.
#-----------------------------------------------------
APP_OBJS=$(addprefix $(OUTPUT_DIR)/, $(OBJS))
```

Figure 227: Modified Makefile (Continued)

```
# Modify where you want the .elf to be created.
APP_ELF=$(addprefix $(OUTPUT_DIR)/, $(APP_OUTPUT_ELF))

#
# Dependency generation (From
# "Managing Projects with GNU Make" by
# Robert Mecklenburg, published by
# O'Reilly
#
dependencies=$(subst .o,.d,$(APP_OBJS))
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

define make-depend
  lm32-elf-gcc-MM\
          -MF$3\
          -MP \
          -MT $2\
          $(CPU_CONFIG)\
          $(CFLAGS)\
          $(CPPFLAGS)\
          $1
endef

.PHONY: all dummy clean
#-------------------------------------------------------------
#
# Rule to clean this project.
# You want to delete items that you created when building,
# including the output directory.
#
#-------------------------------------------------------------
clean:
  @echo cleaning...
  rm -r -f $(OUTPUT_DIR)

# Dummy rule prepares for a build such as creating the output
# directory.
dummy:
  @mkdir -p $(OUTPUT_DIR)
  @echo $(APP_OBJS)
  @echo $(VPATH)
```

Figure 227: Modified Makefile (Continued)

```
#
# "All" rule defines how to build the output desired.
#
all: dummy $(APP_ELF)


# Define a function to build application.
define build_app
  lm32-elf-gcc\
    $(CPU_CONFIG)\
    -T $(LD_FILE)\
    -o$1  \
    $(APP_OBJS)\
    $(PLATFORM_LIBRARY)\
    -lm       \
    $(C_LIB)\
    -lgcc   \
    $(PLATFORM_LIBRARY)\
    -lnosys \
    $(LDFLAGS)
endef

#
# Define how to build the .elf file.
# This depends on the objects required, as well as
# the CPU configuration makefile.
# That is, if any change is detected, rebuild the
# elf file.
#
$(APP_ELF): $(APP_OBJS) $(PLATFORM_RULES_MAKEFILE) $(LD_FILE)\
$(PLATFORM_LIBRARY)
  @echo
  @echo
  @echo building $(APP_ELF)
  $(call build_app,$@)
  lm32-elf-size $(APP_ELF)
  lm32-elf-objdump -d $(APP_ELF) > $(OUTPUT_DIR)/dump.txt
```

Figure 227: Modified Makefile (Continued)

```
#-------------------------------------------------------------
# Create a generic rule to build .o files from .c files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.c
  @echo
  @echo compiling $< to $@
  @echo
  $(call make-depend,$<,$@,$(subst .o,.d,$@))
  @echo
  lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o $@


#-------------------------------------------------------------
# Create a generic rule to build .o files from .S files.
#-------------------------------------------------------------
$(OUTPUT_DIR)/%.o : %.S
  @echo
  @echo compiling $< to $@
  @echo
  lm32-elf-gcc -c $(CPU_CONFIG) $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Software Development Utilities

This chapter describes the software development utilities in the LatticeMico 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 LatticeMico and the build flow in the C/C++ Software Project Environment (SPE). This section also includes commonly used parameters for the tools, along with LM32-specific build options. References to the Newlib and 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.

lm32-elf-ar

The lm32-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

```
lm32-elf-ar [emulation_options] [-]{options}[modifiers]
[member_name] [count] archive_file_name file_name ...

lm32-elf-ar -M [<mri_script]
```

Options can be any of the options listed in Table 19.

Table 19: lm32-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 20 or Table 21.

Table 20: lm32-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 21: lm32-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 21: lm32-elf-ar Generic Modifiers

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

The lm32-elf-ar utility has no emulation-specific options.

The lm32-elf-ar utility supports the following targets: elf32-lm32, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary ihex.

lm32-elf-as

The lm32-elf-as utility is the assembler utility. It takes as input an assembler source (.s) file and generates a relocatable object (.o) file.

Usage

`lm32-elf-as [`*`options`*`] [`*`asmfile...`*`]`

where *options* can be one or more of the options shown in Table 22.

Table 22: lm32-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 22: lm32-elf-as Options (Continued)

-f	Skips white space and comment preprocessing.
-g --gen-debug	Generates debugging information.
--gstabs	Generates STABS debugging information.
--gstabs+	Generates STABS debug info with GNU extensions.
--gdwarf-2	Generates DWARF2 debugging information.
--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 22: lm32-elf-as Options (Continued)

--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.
LM32-specific Options	**Description**
-mmultiply-enabled	Enables multiply instructions.
-mdivide-enabled	Enables divide and modulus instructions.
-mbarrel-shift-enabled	Enables multi-bit shift instructions.
-msign-extend-enabled	Enables sign-extension instructions.
-muser-enabled	Enables user-defined instructions.
-micache-enabled	Enables instruction cache instructions.
-mdcache-enabled	Enables data cache instructions.
-mbreak-enabled	Enables the break instruction.
-mall-enabled	Enables all optional instructions.

lm32-elf-gcc

The lm32-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

`lm32-elf-gcc [`*`options`*`] `*`file...`*

where *options* can be one or more of the options shown in Table 23.

Table 23: lm32-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.

Table 23: lm32-elf-gcc Options (Continued)

Option	Description
'-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.
-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.

Table 23: lm32-elf-gcc Options (Continued)

Option	Description
-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.
-o <*file*>	Places the output into <*file*>.
-x <*language*>	Specifies the language of the following input files. Permissible languages includec c++ assembler or none. “None” means reverting to the default behavior of guessing the language based on the file's extension.

Options starting with -g, -f, -m, -O, -W, or --param are automatically passed on to the various subprocesses invoked by lm32-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.

lm32-elf-ld

The lm32-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

```
lm32-elf-ld [options] file...
```

where *options* can be one or more of the options shown in Table 24.

Table 24: lm32-elf-ld Options

Options	Description
a KEYWORD	Shares library control for HP/UX compatibility.

Table 24: lm32-elf-ld Options (Continued)

-A ARCH, --architecture ARCH	Sets architecture.
-b TARGET, --format TARGET	Specifies target for following input files.
-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.

Table 24: lm32-elf-ld Options (Continued)

-S, --strip-debug	Strips debugging symbols.
--strip-discarded	Strips symbols in discarded sections.
--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.

Table 24: lm32-elf-ld Options (Continued)

--check-sections	Checks section addresses for overlaps (default).
--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.

Table 24: lm32-elf-ld Options (Continued)

-nostdlib	Only uses library directories specified on the command line.
--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.

Table 24: lm32-elf-ld Options (Continued)

--warn-constructors	Warns if global constructors and destructors are seen.
--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.
lm32-elf-ld: supported targets:	elf32-lm32, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex.
lm32-elf-ld: supported emulations:	elf32lm32
lm32-elf-ld: emulation specific options:	No emulation-specific options.

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

lm32-elf-nm

The lm32-elf-nm utility lists symbols in [*files*] (a.out by default).

Usage

`lm32-elf-nm [options] [files]`

where *options* can be one or more of the options shown in Table 25.

Table 25: lm32-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.
-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.

Table 25: lm32-elf-nm Options (Continued)

-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.
lm32-elf-nm: supported targets:	elf32-lm32 elf32-little elf32-big srec symbolsrec tekhex binary ihex.

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

lm32-elf-objcopy

The lm32-elf-objcopy utility copies a binary file, possibly transforming it in the process.

Usage

```
lm32-elf-objcopy [options] in_file [out_file]
```

where *options* can be one or more of the options shown in Table 26.

Table 26: lm32-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.
--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.

Table 26: lm32-elf-objcopy Options (Continued)

-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.
--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*>.

Table 26: lm32-elf-objcopy Options (Continued)

--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.
lm32-elf-objcopy: supported targets:	elf32-lm32, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex.

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

lm32-elf-objdump

The lm32-elf-objdump (lm32-elf-objcopy) utility displays information from object (.o) files.

Usage

`lm32-elf-objdump <options> <files>`

where *options* can be one or more of the options shown in Table 27. At least one of the options must be given.

Table 27: lm32-elf-objdump Options

Option	Description
-a, --archive-headers	Displays archive header information.

Table 27: lm32-elf-objdump Options (Continued)

-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.
-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.

Table 27: lm32-elf-objdump Options (Continued)

-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.
lm32-elf-objdump: supported targets:	elf32-lm32, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex
lm32-elf-objdump: supported architectures:	lm32

lm32-elf-size

The lm32-elf-size program displays the sizes of sections inside binary files. If no input files are specified, a.out is assumed.

Usage

`lm32-elf-size [`*`options`*`] [`*`files`*`]`

where *options* can be one or more of the options shown in Table 28.

Table 28: lm32-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.
lm32-elf-size: supported targets:	elf32-lm32, elf32-little, elf32-big, srec, symbolsrec, tekhex, binary, ihex

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

Debug Tools

This section provides information on the GDB target stub as it pertains to its incorporation in an application and its activation from the host. This section does not cover all the details of GDB usage for debugging applications with the C/C++ Software Programming Environment (SPE) tools.

lm32-elf-gdb

The lm32-elf-gdb utility is the GNU GDB debugger.

Usage

```
lm32-elf-gdb [options] [executable_file [core_file or
process_id]] gdb [options] --args executable_file
[inferior_arguments ...]
```

where *options* can be one or more of the options shown in Table 29.

Table 29: lm32-elf-gdb Options

Options	Description
--args	Arguments after executable file are passed to inferior.
--[no]async	Enables (disable) asynchronous version of CLI.

Table 29: lm32-elf-gdb Options (Continued)

-b BAUDRATE	Sets serial port baud rate used for remote debugging.
--batch	Exits after processing options.
--cd=DIR	Changes current directory to DIR.
--command=FILE	Executes GDB commands from FILE.
--core=COREFILE	Analyzes the core dump COREFILE.
--pid=PID	Attaches to running process PID.
--dbx	DBX compatibility mode.
--directory=DIR	Searches for source files in DIR.
--epoch	Outputs information used by epoch emacs-GDB interface.
--exec=EXECFILE	Uses EXECFILE as the executable.
--fullname	Outputs information used by emacs-GDB interface.
--help	Prints this message.
--interpreter=INTERP	Selects a specific interpreter and user interface.
--mapped	Uses mapped symbol files if supported on this system.
--nw	Does not use a window interface.
--nx	Does not read .gdbinit file.
--quiet	Does not print version number on startup.
--readnow	Fully reads symbol files on first access.
--se=FILE	Uses FILE as symbol file and executable file.
--symbols=SYMFILE	Reads symbols from SYMFILE.
--tty=TTY	Uses TTY for input/output by the program being debugged.
--tui	Uses a terminal user interface.
--version	Prints version information and then exit.
-w	Uses a window interface.
--write	Sets writing into executable and core files.
--xdb	XDB compatibility mode.

For more information, type **help** from within GDB, or consult the GDB manual available as online information or as a printed manual. Report bugs by email to bug-gdb@gnu.org.

Glossary

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

hardware platform See "platform."

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

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

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

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

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

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

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

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

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

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

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

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

project workspace See “workspace.”

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

running Running is the process of executing a software progam.

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

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

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

E