LatticeMico32 Tutorial

May 2014

Copyright

Copyright © 2014 Lattice Semiconductor Corporation.

This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Lattice Semiconductor Corporation.

Trademarks

Lattice Semiconductor Corporation, L Lattice Semiconductor Corporation (logo), L (stylized), L (design), Lattice (design), LSC, CleanClock, Custom Mobile Device, DiePlus, E^2CMOS, ECP5, Extreme Performance, FlashBAK, FlexiClock, flexiFLASH, flexiMAC, flexiPCS, FreedomChip, GAL, GDX, Generic Array Logic, HDL Explorer, iCE Dice, iCE40, iCE65, iCEblink, iCEcable, iCEchip, iCEcube, iCEcube2, iCEman, iCEprog, iCEsab, iCEsocket, IPexpress, ISP, ispATE, ispClock, ispDOWNLOAD, ispGAL, ispGDS, ispGDX, ispGDX2, ispGDXV, ispGENERATOR, ispJTAG, ispLEVER, ispLeverCORE, ispLSI, ispMACH, ispPAC, ispTRACY, ispTURBO, ispVIRTUAL MACHINE, ispVM, ispXP, ispXPGA, ispXPLD, Lattice Diamond, LatticeCORE, LatticeEC, LatticeECP, LatticeECP-DSP, LatticeECP2, LatticeECP2M, LatticeECP3, LatticeECP4, LatticeMico, LatticeMico8, LatticeMico32, LatticeSC, LatticeSCM, LatticeXP, LatticeXP2, MACH, MachXO, MachXO2, MachXO3, MACO, mobileFPGA, ORCA, PAC, PAC-Designer, PAL, Performance Analyst, Platform Manager, ProcessorPM, PURESPEED, Reveal, SensorExtender, SiliconBlue, Silicon Forest, Speedlocked, Speed Locking, SuperBIG, SuperCOOL, SuperFAST, SuperWIDE, sysCLOCK, sysCONFIG, sysDSP, sysHSI, sysI/O, sysMEM, The Simple Machine for Complex Design, TraceID, TransFR, UltraMOS, and specific product designations are either registered trademarks or trademarks of Lattice Semiconductor Corporation or its subsidiaries in the United States and/or other countries. ISP, Bringing the Best Together, and More of the Best are service marks of Lattice Semiconductor Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Disclaimers

NO WARRANTIES: THE INFORMATION PROVIDED IN THIS DOCUMENT IS “AS IS” WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND INCLUDING WARRANTIES OF ACCURACY, COMPLETENESS, MERCHANTABILITY, NONINFRINGEMENT OF INTELLECTUAL PROPERTY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL LATTICE SEMICONDUCTOR CORPORATION (LSC) OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (WHETHER DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION PROVIDED IN THIS DOCUMENT, EVEN IF LSC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF CERTAIN LIABILITY, SOME OF THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

LSC may make changes to these materials, specifications, or information, or to the products described herein, at any time without notice. LSC makes no commitment to update this documentation. LSC reserves the right to discontinue any product or service without notice and assumes no obligation to correct any errors contained herein or to advise any user of this document of any correction if such be made. LSC recommends its customers obtain the latest version of the relevant information to establish, before ordering, that the information being relied upon is current.

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

LatticeMico32 Tutorial **1**

Introduction **1**
- Learning Objectives **2**
- Time to Complete This Tutorial **2**
- System Requirements **2**
- Accessing Online Help **3**
- About the Tutorial Design **3**
- Tutorial Data Flow **5**

LatticeECP3 Versa Evaluation Board **8**

Task 1: Create a New Lattice Diamond Project **8**

Task 2: Create the Development Microprocessor Platform **11**
- Create a New MSB Platform **11**
- Add the Microprocessor Core **16**
- Add On-Chip and Off-Chip Memory **20**
- Add the Peripheral Components **23**
- Specify the Connections Between Master and Slave Ports **26**
- Assign Component Addresses **29**
- Assign Interrupt Request Priorities **31**
- Perform a Design Rule Check **31**
- Generate the Microprocessor Platform **31**
- Create a User Top-Level Module **34**

Task 3: Create the Software Application Code **37**
- Create a New C/C++ SPE Project **38**
- Linker Configuration **40**
- Build the Project **43**

Task 4: Generate the Microprocessor Bitstream **45**
- Import the MSB Output File **46**

Connect the Microprocessor to the FPGA Pins **47**
- Perform Functional Simulation **49**
- Perform Timing Simulation **49**
- Generate the Bitstream **49**

Task 5: Download the Hardware Bitstream to the FPGA **50**

Task 6: Debug and Execute the Software Application Code on the Development Board **52**
- Software Application Code Execution Flow **52**
- Debug the Software Application Code on the Board **54**
- Insert Breakpoints **59**
- Execute the Software Application Code **60**
- Modify and Re-execute the Software Application Code **62**

Task 7: Setting up the Options to Deploy the Software Code to the SPI Flash Memory **64**
- Selecting the Appropriate LatticeMico EBA Value **64**
- Generating LatticeMico32 Bootable Application Binary **67**
- Generating Production Microprocessor Bitstream **72**

Task 8: Deploy the SPI Flash Image (Merged With Production Microprocessor Bitstream and Application Software) to SPI Flash Memory **76**

Summary **81**

Glossary **82**

Recommended References **84**

LatticeMico32 Tutorial

Introduction

This tutorial steps you through the basic process involved in using the LatticeMico System software to implement a LatticeMico32 32-bit soft microprocessor and attached components in a Lattice Semiconductor device for the LatticeECP3 Versa Evaluation Board. LatticeMico System encompasses three tools: the Mico System Builder (MSB), the C/C++ Software Project Environment (C/C++ SPE), and the Debugger. Together, they enable you to build an embedded microprocessor system on a single FPGA device and to write and debug the software that drives it. Such a microprocessor lowers cost by saving board space and increases performance by reducing the number of external wires.

The LatticeMico System interface is based on the Eclipse environment, which is an open-source development and application framework for building software.

Although you can install LatticeMico System as a stand-alone tool, this tutorial assumes that you have installed Lattice Diamond before installing LatticeMico System. After you have created a project in Lattice Diamond, the tutorial shows you how to use MSB to choose a Lattice Semiconductor 32-bit microprocessor, attach components to it, and generate a top-level design, including the microprocessor and the chosen components. Next you will use Lattice Diamond to synthesize, map, place, and route the design and generate a bitstream for it. You will then download this bitstream to the FPGA on the board. The tutorial then changes to the Lattice Software Project Environment (C/C++ SPE) and shows how to use C/C++ SPE to write and compile the software application code that exercises the microprocessor and components. Finally, it shows how to download and debug the code on the board and deploy it in the SPI Flash chips on to the LatticeECP3 Versa Evaluation Board.

This tutorial is intended for a new or infrequent user of the LatticeMico System software and covers only the basic aspects of it. The tutorial assumes that

you have reviewed the *LatticeECP3 Versa Evaluation Board User's Guide* to familiarize yourself with the product and to set up your board correctly.

For more detailed information on the LatticeMico System software, see the sources listed in “Recommended References” on page 84.

Learning Objectives

When you have completed this tutorial, you should be able to do the following:

- Use MSB to configure a Lattice Semiconductor 32-bit microprocessor for your design, select the desired components, and connect the selected components to the microprocessor with a shared-bus arbitration scheme, which is the default.
- Use The Lattice Software Project Environment to create the C/C++ software application code that drives the microprocessor and components.
- Import the Verilog files, Verilog/VHDL files, or an EDIF file generated by a synthesis tool.
- Import an .lpf file containing the pinout.
- Synthesize, map, place, and route the design.
- Generate a bitstream of the microprocessor and download it to an FPGA on the board.
- Compile, download, and debug the software application code on the LatticeECP3 Versa Evaluation Board.
- Debug the hardware and software on the board.

Time to Complete This Tutorial

The time to complete this tutorial is about two hours.

System Requirements

You can run this tutorial on Windows or Linux.

Windows

If you will be running this tutorial on Windows on a PC, your system must meet the following minimum system requirements:

- Pentium II PC running at 400 MHz or faster
- Microsoft Windows 2000®, Windows XP® Professional, Windows 7, or Windows Vista®
- USB port for use with the LatticeECP3 Versa Evaluation Board

The following software is required to complete the tutorial:

- Lattice Diamond 1.3 software or later with device support for the device used with your build of the LatticeECP3 Versa Evaluation Board
- LatticeMico System version 1.3 or later

See the *Lattice Diamond Installation Notice* for the current release for information on installing software on the Windows platform.

Linux

If you will be running this tutorial on Linux on a PC, your system must meet the following minimum system requirements:

- Red Hat Enterprise Linux operating system Version 4.0, 5.0, or 6.0
- Lattice Diamond version 1.0
- For mixed Verilog/VHDL support: Synopsys® Synplify Pro® 8.9 or Synplify Pro 8.9.1 for Linux
- Linux system with USB port

See the *Lattice Diamond Installation Notice* for the current release for information on installing software on the Linux platform.

Hardware

This tutorial requires the following hardware:

- A LatticeECP3 Versa Development Kit
- USB cable
- AC adapter cord

Note

If you want to perform functional simulation for the mixed Verilog/VHDL flow, you must have access to a simulator that supports mixed-mode Verilog and VHDL simulation.

Accessing Online Help

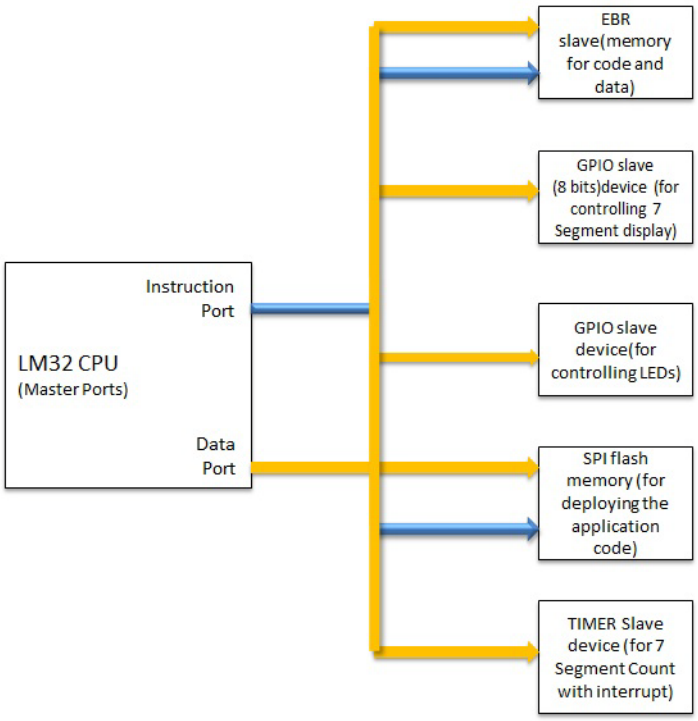
You can access the online Help for MSB, C/C++ SPE, the Debugger, or Eclipse Workbench by choosing **Help > Help Contents** in the LatticeMico System graphical user interface.

About the Tutorial Design

This tutorial uses a LatticeECP3 device, and all references are based on the LatticeECP3 device. The tutorial design consists of the LatticeMico32 embedded microprocessor, an Embedded Block RAM (EBR), GPIOs, a SPI

flash memory, and a timer. After you add these components, you will specify the connections between the master and slave ports on these components, as shown in Figure 1.

Figure 1: Desired Connections Between Master and Slave Ports



In this design, the instruction port and the data port of the CPU are the master ports. All other ports are slave ports. The instruction port will access the LatticeMico EBR and the LatticeMico SPI flash memory. The data port will access the LatticeMico EBR, the LatticeMico GPIOs, the LatticeMico SPI flash memory, the LatticeMico TImer, and the LatticeMico UART.

Tutorial Data Flow

You will perform the following major steps to create an embedded microprocessor system:

1. Create a new project in Lattice Diamond.
2. Create a microprocessor platform for the LatticeMico32 microprocessor in MSB with a shared-bus arbitration scheme, which is the default.
3. Write the software application code for the microprocessor platform in C/C++ SPE.
4. Synthesize the platform in a synthesis tool, such as Synopsys® Synplify Pro® or Mentor Graphics® Precision RTL Synthesis, to generate an EDIF file.
5. Generate a bitstream of the microprocessor platform in Diamond.
6. Download the hardware bitstream to the FPGA using Diamond Programmer.
7. Debug and execute the software application code on the board.
8. Deploy the software application code into the SPI flash memory.
9. Deploy the microprocessor bitstream.

Note

This tutorial does not show you how to debug your software application code on the instruction set simulator, but it does show you how to debug the design by downloading the bitstream and the application code to the board.

This tutorial supports both Verilog and mixed Verilog/VHDL design flows in Diamond for Windows and Linux users. The Windows Verilog design flow for using LatticeMico System to create an embedded microprocessor and the software code for it is shown in Figure 2 on page 6. The Windows mixed Verilog/VHDL design flow is shown in Figure 3 on page 7. The difference between the two methods is that mixed verilog/VHDL designs have a VHDL wrapper as an output from MSB. The VHDL wrapper is an input to Synthesis and Functional Simulation in the Diamond flow.

Figure 2: Design Flow for Windows Verilog Users

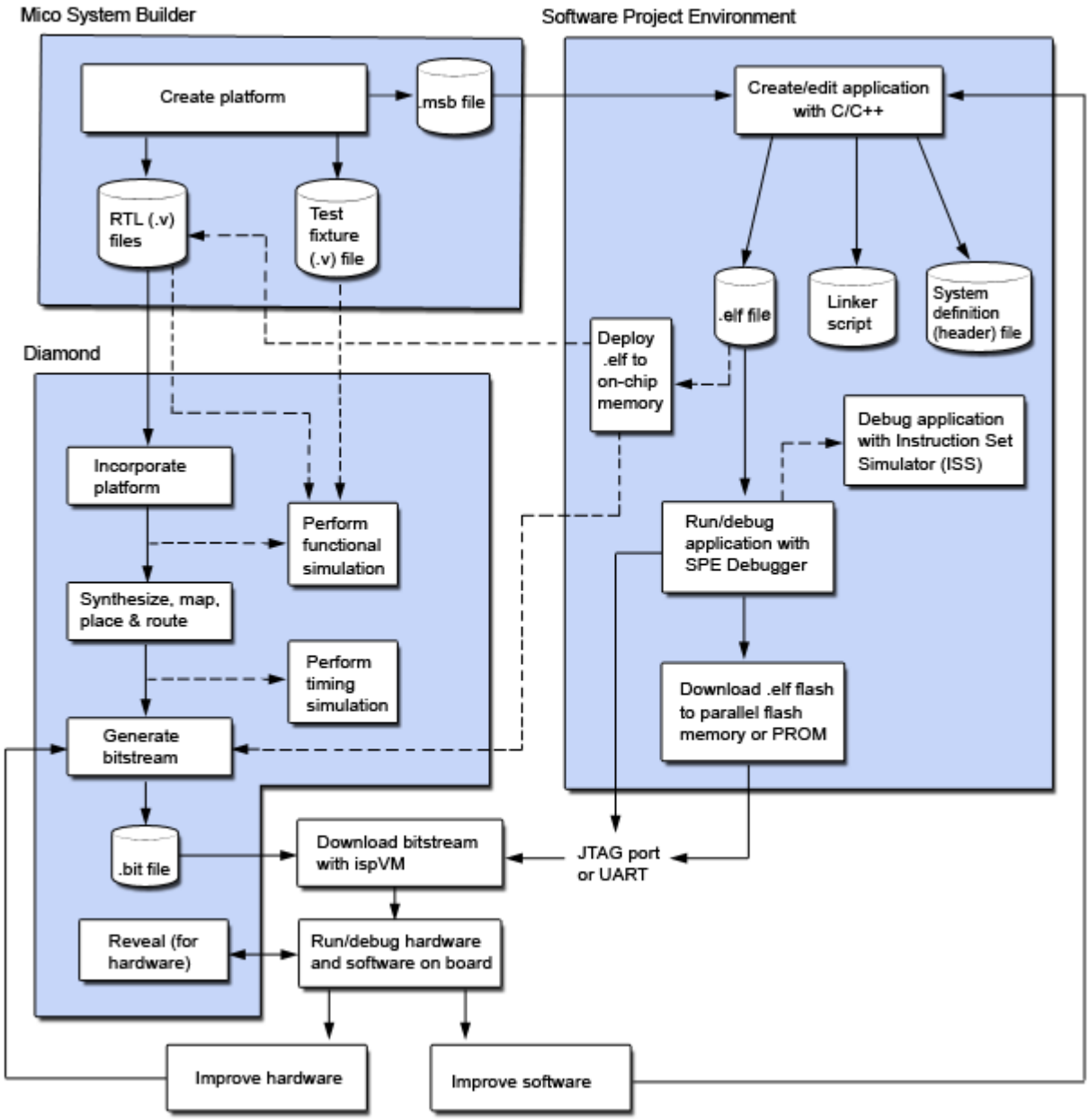
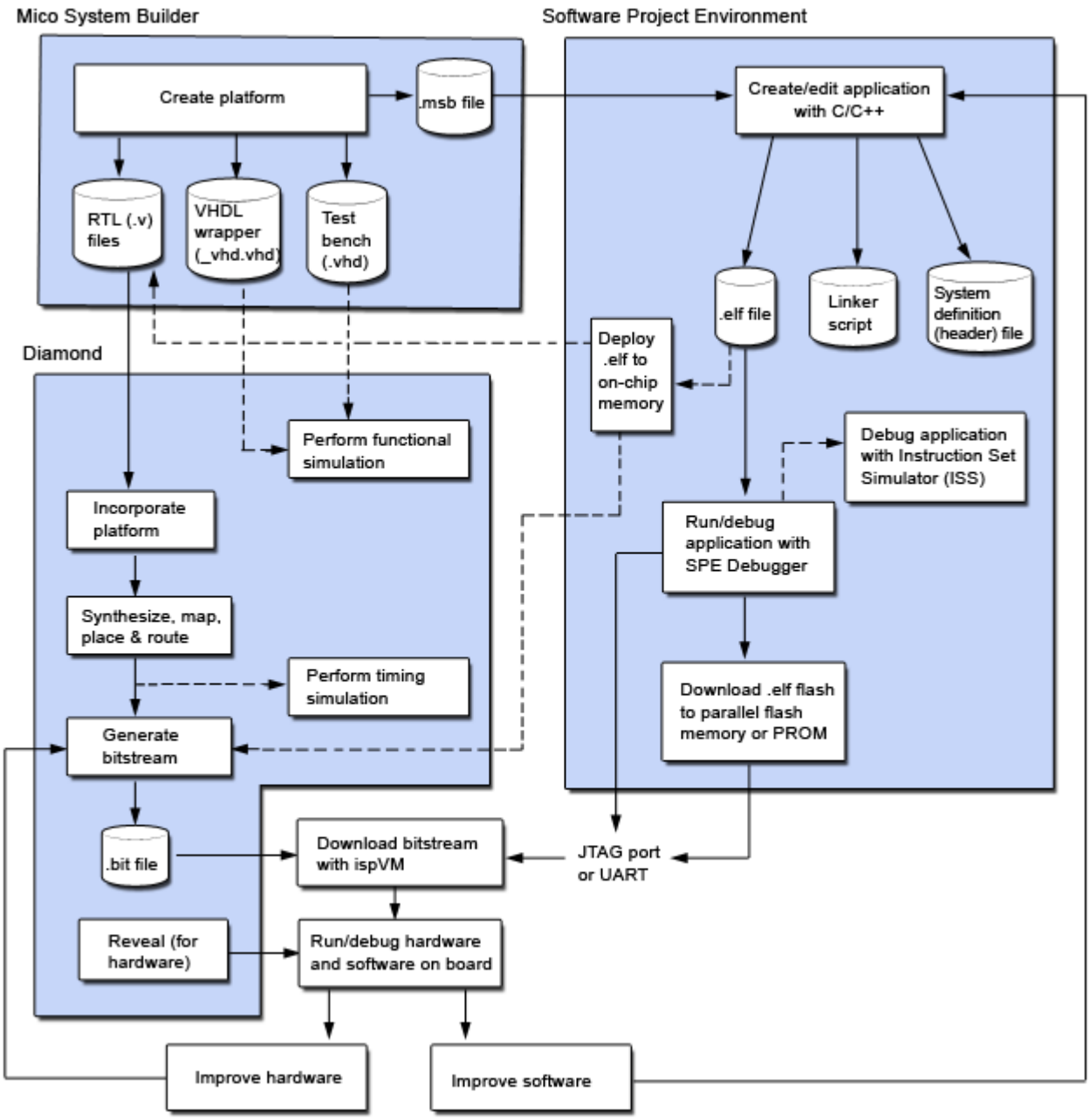


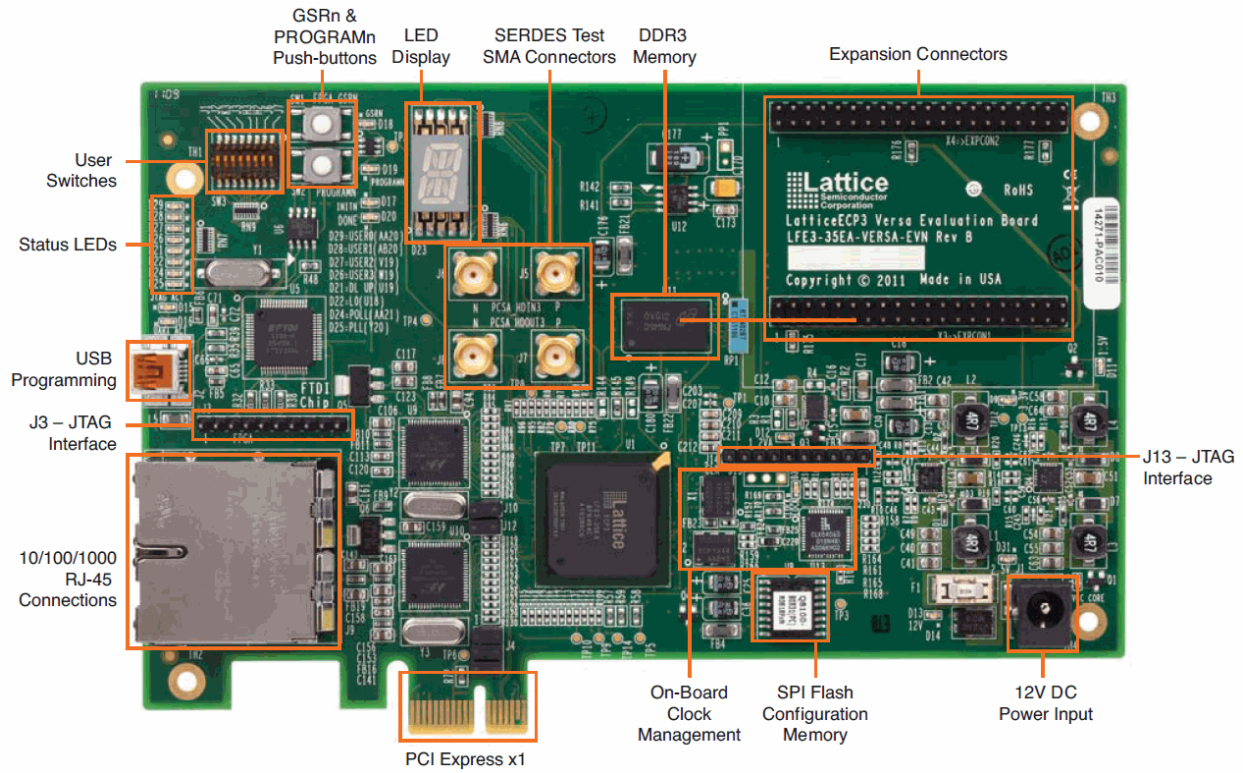
Figure 3: Design Flow for Windows VHDL Users, Using Mixed Verilog/VHDL Design Entry



LatticeECP3 Versa Evaluation Board

Figure 4 shows where some of the components mentioned in this tutorial reside on the LatticeECP3 Versa Evaluation Board.

Figure 4: LatticeECP3 Versa Evaluation Board



Task 1: Create a New Lattice Diamond Project

As a first step, you will create a new project in Diamond.

Note

In this tutorial, the directory paths follow the Windows nomenclature. For Linux, replace the "\" character with the "/" character.

To create a new Lattice Diamond project:

1. Create a folder called **lm32_tutor** in the following directory:
   - For Windows: <*Diamond_install_path*>\examples
   - For Linux: ~/LatticeMico32
2. Start Lattice Diamond:
   - On the Windows desktop, choose **Start > Programs > Lattice Diamond > Lattice Diamond**.

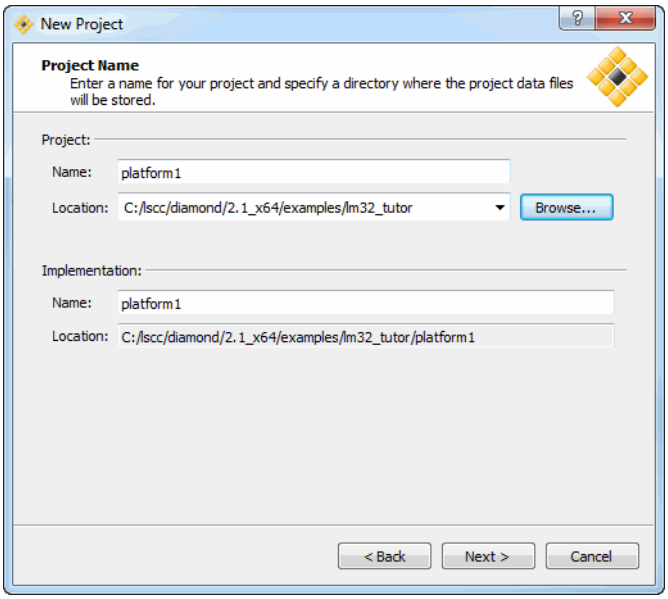
- On the Linux command line, run the following script:

  <*Diamond_install_path*>/bin/lin/**diamond**

3. Choose **File > New > Project**, and then click **Next** in the New Project wizard.
4. In the New Project wizard dialog box, shown in Figure 5 on page 9, select or specify the following:
   a. In the Project Name box, enter **platform1**.
   b. In the Location box, enter the path for the lm32_tutor directory:
      - For Windows: <*Diamond_install_path*>\examples**lm32_tutor**
      - For Linux: ~/LatticeMico32/**lm32_tutor**

      By default, Diamond uses the Project name and location for the implementation and fills in this information. Although you can change to a different name and directory for the first implementation, you will use the default settings for this tutorial.
5. Click **Next** to proceed to the Add Source dialog box, and then click **Next**. You will add the source later.

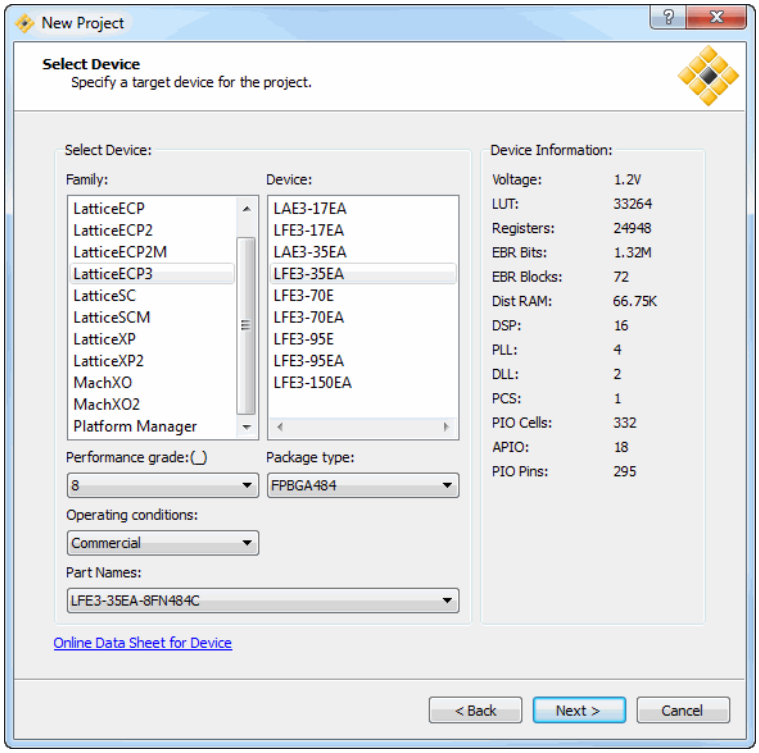
Figure 5: New Project Wizard



6. In Select Device dialog box, shown in Figure 6 on page 10, make the following selections:
   a. In the Family box, select **LatticeECP3**.
   b. In the Device box, select **LFE3-35EA**.
   c. In the Speed grade box, select **8**.
   d. In the Package Type box, select **FPBGA484.**
   e. In the Operating Conditions box, select **Commercial**.

The dialog box should now resemble the illustration in Figure 6

Figure 6: New Project Wizard – Select a Device Dialog Box

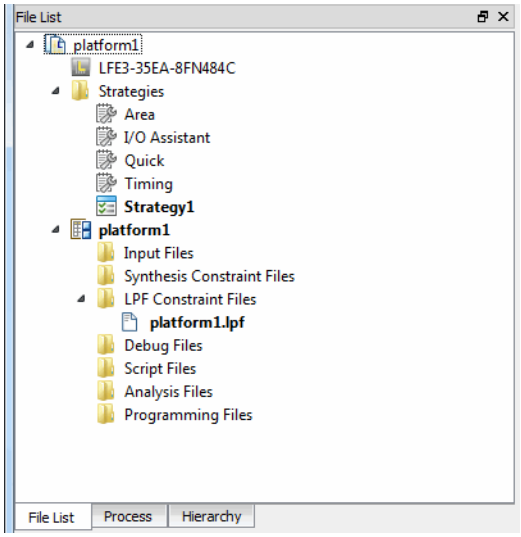


7. Click **Next**, and then click **Finish**.

   In the File List, shown in Figure 7, the project name is shown at the top. The implementation name, which has the same name as the project name, is displayed in bold type, with the implementation icon . The project is assigned a default strategy, Strategy1, which is also displayed in bold type with the strategy icon . A strategy is a collection of settings for logic synthesis, place, and route. You can view these settings by

double-clicking the strategy name. The platform1 project is also assigned a logical preference file, platform1.lpf.

Figure 7: Diamond File List



Task 2: Create the Development Microprocessor Platform

In Task 1, you created a blank Diamond project. The Diamond project is a placeholder for the LatticeMico32 microprocessor platform. You use LatticeMico System Builder (MSB) to create the microprocessor platform. MSB allows you to select components to attach to the microprocessor. Additionally, MSB allows you to customize each of the attached components. After all components are attached to the microprocessor, you use MSB to generate Verilog or VHDL source code that describes a microprocessor-based System-on-a-Chip (SOC). You then enter the HDL source code into the Diamond project in order to create the bitstream used to configure the FPGA.

The steps in this section describe how to build a LatticeMico32 microprocessor SOC that is intended for developing and debugging LatticeMico based systems. During system development, the FPGA resources and the firmware are in a state of flux, undergoing many changes. When you deploy a LatticeMico32 microprocessor, as described here, you reduce the impact the of on-going changes in the development environment.

Create a New MSB Platform

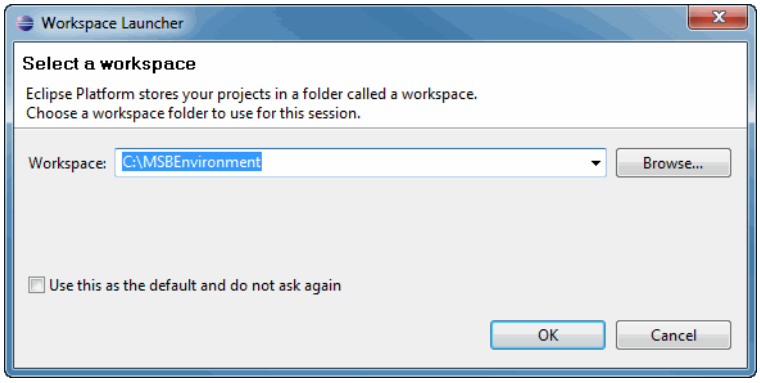
Now you will create a new platform in MSB.

To create a new platform:

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

   The Workspace Launcher dialog box, shown in Figure 8, displays a default workspace location for the platform.

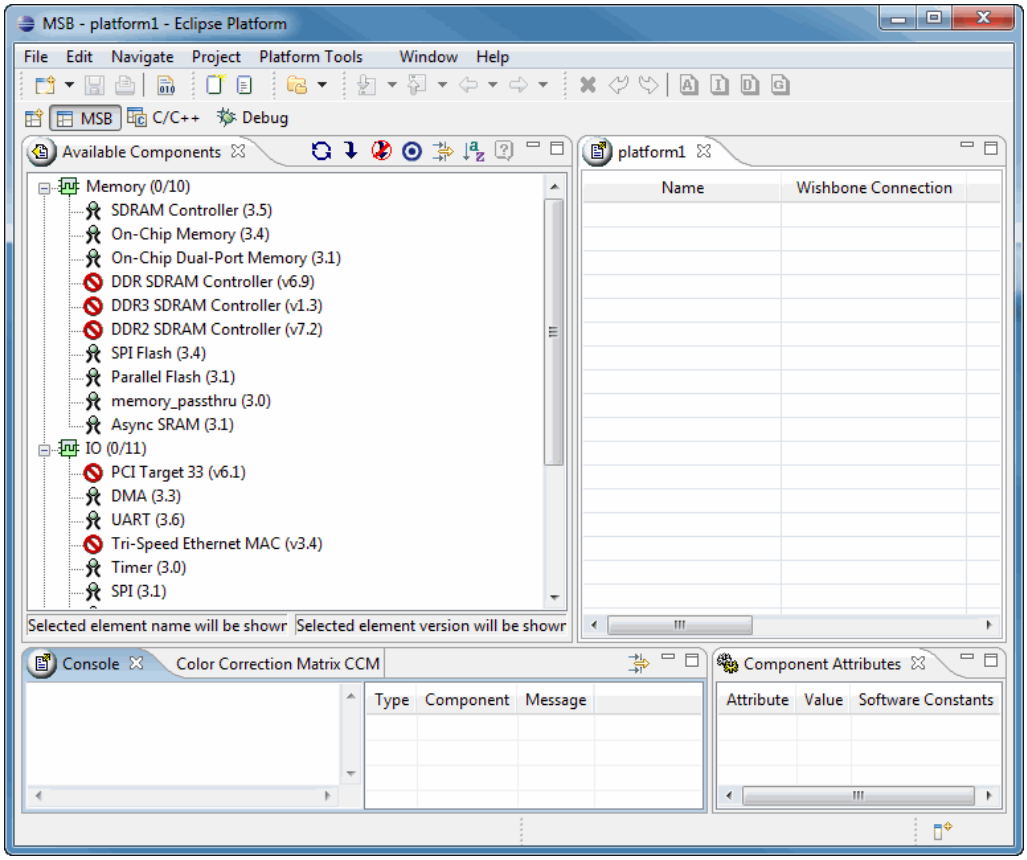
Figure 8: Workspace Launcher Dialog Box



2. Accept the default location, or click the Browse button to select a different location. To keep the same workspace for future sessions, select the “Use this as the default and do not ask again” option.
3. Click **OK**.

The LatticeMico System interface now appears, as shown in Figure 9.

Figure 9: LatticeMico System Interface



4. In the upper left-hand corner of the graphical user interface, select **MSB**, if it is not already selected, to open the MSB perspective.
5. Choose **File > New Platform**.
6. In the New Platform Wizard dialog box, make the following selections:
   a. In the Platform Name box, enter **platform1**.
   b. In the Directory box, browse to the lm32_tutor directory and click **OK**:
      - For Windows: <*Diamond_install_path*>\examples\lm32_tutor
      - For Linux: ~/LatticeMico32/lm32_tutor
   c. Do one of the following:
      - If you are generating a platform in Verilog, leave the Create VHDL Wrapper unselected.
      - If you are generating a platform in mixed Verilog/VHDL, select only Create VHDL Wrapper.
   d. In the Arbitration scheme box, select **Shared Bus (Default)** from the drop-down menu, if it is not already selected.

e. In the Device Family section, select **LatticeECP3** from the Family menu and **LFE3-35EA** from the Device menu.

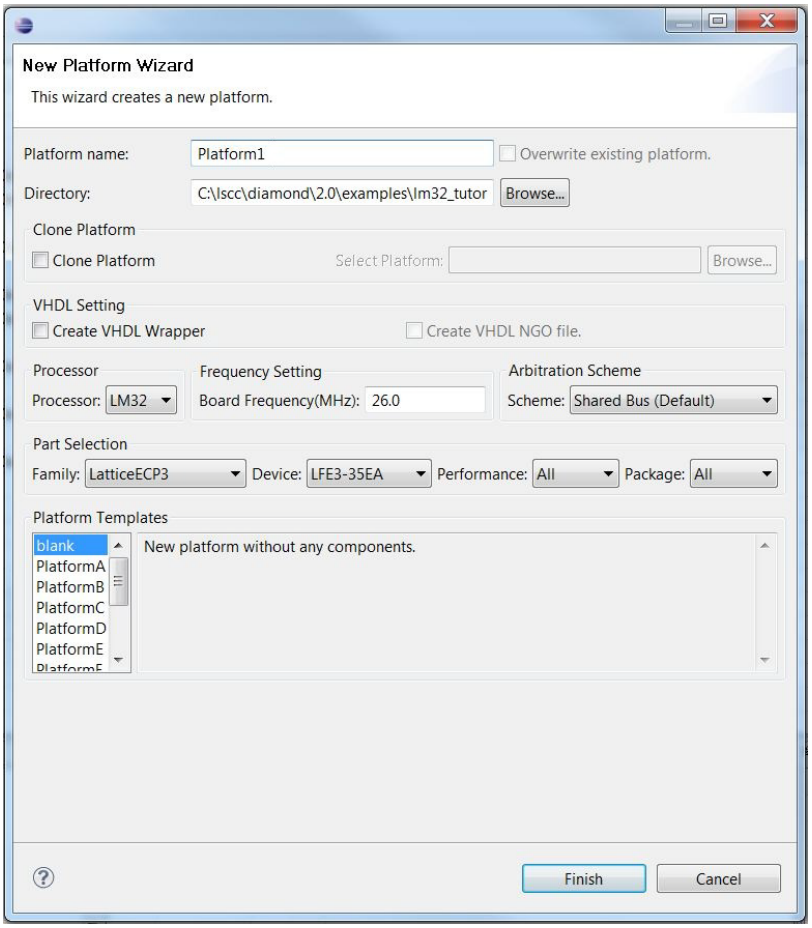
f. In the Board Frequency (MHz) box, change frequency to 26.0

g. In the Platform Templates box, select **blank**.

Templates are pre-created platforms that facilitate rapid development. They target the LatticeECP3 Versa Evaluation Board for LatticeECP3. Each platform also has an associated constraint file that you can import into Diamond to avoid the effort of creating a constraints file. MSB gives you the flexibility of creating and adding your own custom templates and associated constraints files for the LatticeECP3 Versa Evaluation Board or a custom board, in addition to using the templates provided as part of the installation package.

The New Platform Wizard dialog box should look like the illustration in .Figure 10

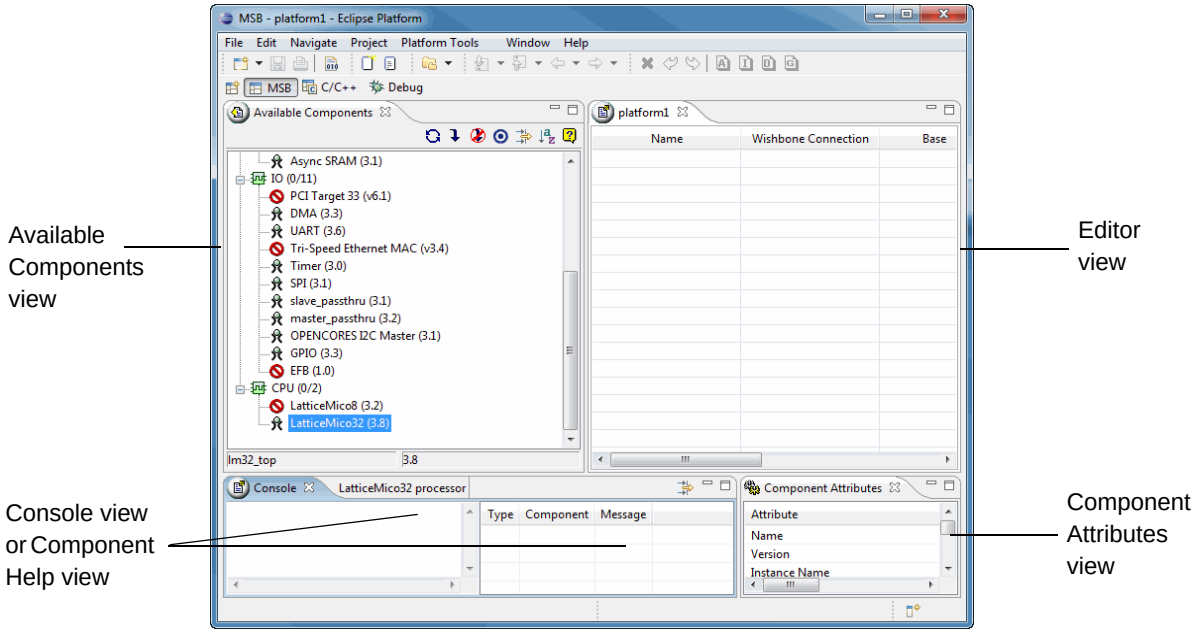
Figure 10: New Platform Wizard Dialog Box



7. Click **Finish**.

The MSB perspective now appears, as shown in Figure 11.

Figure 11: MSB Perspective



The MSB perspective consists of the following views:

- Available Components view, which displays all the available components that you can use to create the design:
  - A list of hardware components: microprocessor, memories, components, and bus interfaces. Bus interfaces can be masters or slaves (see “Specify the Connections Between Master and Slave Ports” on page 26 for more information on masters and slaves). The component list shown in Figure 11 is the standard list that is given for each new platform.
  - A list of preconfigured systems: demonstrations and pre-verified configurations for a given development board or a configuration that you previously built

  You can double-click on a component to open a dialog box that enables you to customize the component before it is added to the design. The component is then shown in the Editor view.
- Editor view, which is a table that displays the components that you have chosen in the Available Components view. It includes the following columns:
  - Name, which displays the names of the chosen components and their ports
  - Wishbone Connection, which displays the connectivity between master and slave ports
  - Base, which displays the start addresses for components with slave ports. This field is editable.

- End, which displays the end addresses for components with slave ports. This field is not editable. The value of the end address is equivalent to the value of the base address plus the value of the size.
- Size (Bytes), which displays the number of addresses available for component access. This field is editable for the LatticeMico EBR and the LatticeMico asynchronous SRAM controller components only.
- Lock, which indicates whether addresses are locked from any assignments. If you lock a component, its address will not change when you select Platform Tools > Generate Address.
- IRQ, which displays the interrupt priorities of all components that have interrupt lines connected to the LatticeMico32 microprocessor. The LatticeMico32 microprocessor can accept up to 32 external interrupt lines.
- Disable, which indicates whether components are temporarily excluded from the design

- Component Help view, which displays information about the component that you selected in the Available Components view. The Help page displays the name of the component—for example, “LatticeMico Timer” or “LatticeMico UART—and gives a brief description of the function of the component. It also provides a list and explanation of the parameters that appear in the dialog box when you double-click the component. If you click the icon next to the component name, you can view a complete description of the component in a PDF file.
- Console view, which displays informational and error messages output by MSB
- Component Attributes view, which displays the name, parameters, and values of the component selected in the Available Components view or the Editor view. This view is read-only.

Add the Microprocessor Core

The first step in building the platform is to add the microprocessor core. In this release, only the LatticeMico32 microprocessor is available.

You will be using the default cache setting for this task. Refer to the *LatticeMico32 Processor Reference Manual* for more information on caches.

To add the microprocessor core:

1. Under CPU in the Available Components view, click **LatticeMico32** to view the information available about the LatticeMico32 microprocessor.

   Information about the LatticeMico32 microprocessor, including the parameters that you can set for it, now appears in the Component Help view and in the Component Attributes view in the lower third of the screen.

   If you click the icon in the Component Help view, you can view the *LatticeMico32 Processor Reference Manual*, which provides a complete description of the microprocessor.

Similarly, if you click this icon for a memory or a peripheral component, you can view the data sheet about that component.

2. Double-click **LatticeMico32** to open the Add LatticeMico32 dialog box. Alternatively, you can select **LatticeMico32**, and then click the Add Component button ( ).

   The parameters in the dialog box, shown in Figure 12 on page 18, correspond to those in the table in the Component Help view.

   You are defining a development LatticeMico32 microprocessor. The LatticeMico32 microprocessor component, when the Enable Debug Interface option is selected, has an internal Embedded Block RAM memory attached to the Wishbone bus. This memory is automatically initialized with LatticeMico opcodes. This means that when the LatticeMico32 microprocessor comes out of reset, it has a valid set of opcodes to execute. The LatticeMico32 microprocessor needs only a few key elements to operate correctly: a good input clock, a reset strobe assertion and de-assertion, and a set of known good opcodes. During the development process, the Debug Monitor memory attached to the LatticeMico32 Wishbone bus is the only guaranteed source of known good opcodes. It is of vital importance for the Reset Exception Address to point to this memory. By default, the Debug Port Base Address is assigned to 0x00000000. This address can be changed, but it is important that the Reset Exception Address be updated to match the Debug Port Base Address.

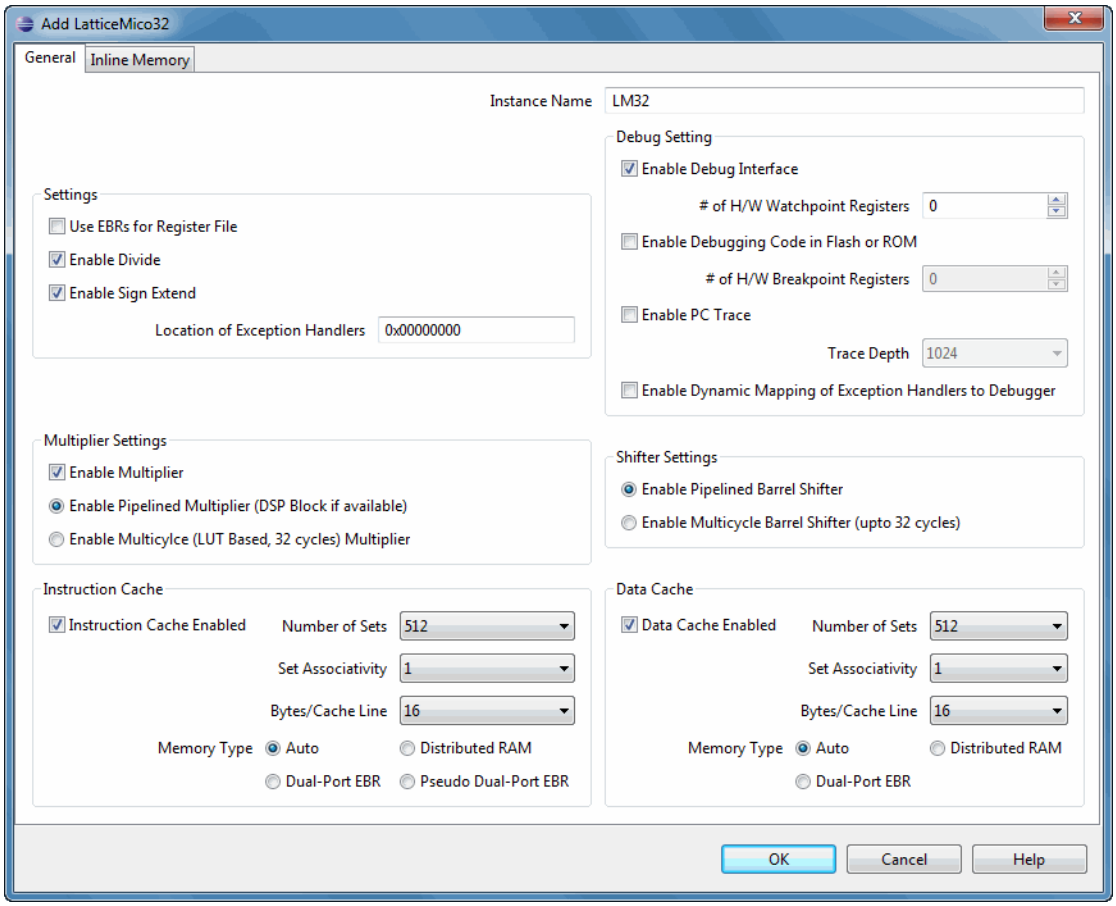
   This tutorial will leave the Debug Port Base Address set to 0x00000000.

3. In the Add LatticeMico32 dialog box, do the following:

   a. Select the **General** tab. If it is not already set as default, type **0x00000000** in the “Location of Exception Handlers” box to set the LatticeMico32 reset vector, as shown in Figure 12 on page 18. This step sets the reset exception address, which is the address from where the microprocessor will begin fetching instructions at power-up. This address must be aligned to a 256-byte boundary.

b. Under the section Instruction Cache, select **Instruction Cache Enabled**.

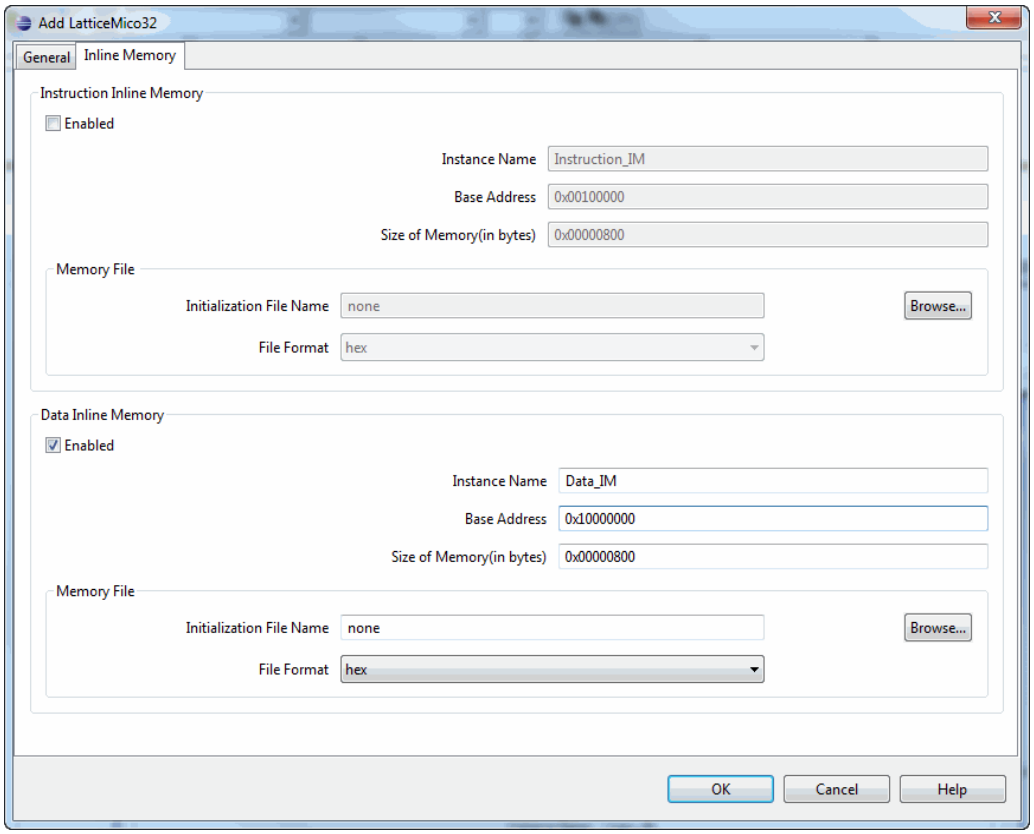
Figure 12: Add LatticeMico32 Dialog Box – General Tab



c. Select the **Inline Memory** tab. Under the section Data Inline Memory, select **Enabled**. If it is not already set as default, type **0x10000000** in the Base Address text box, as shown in Figure 13 on page 19.

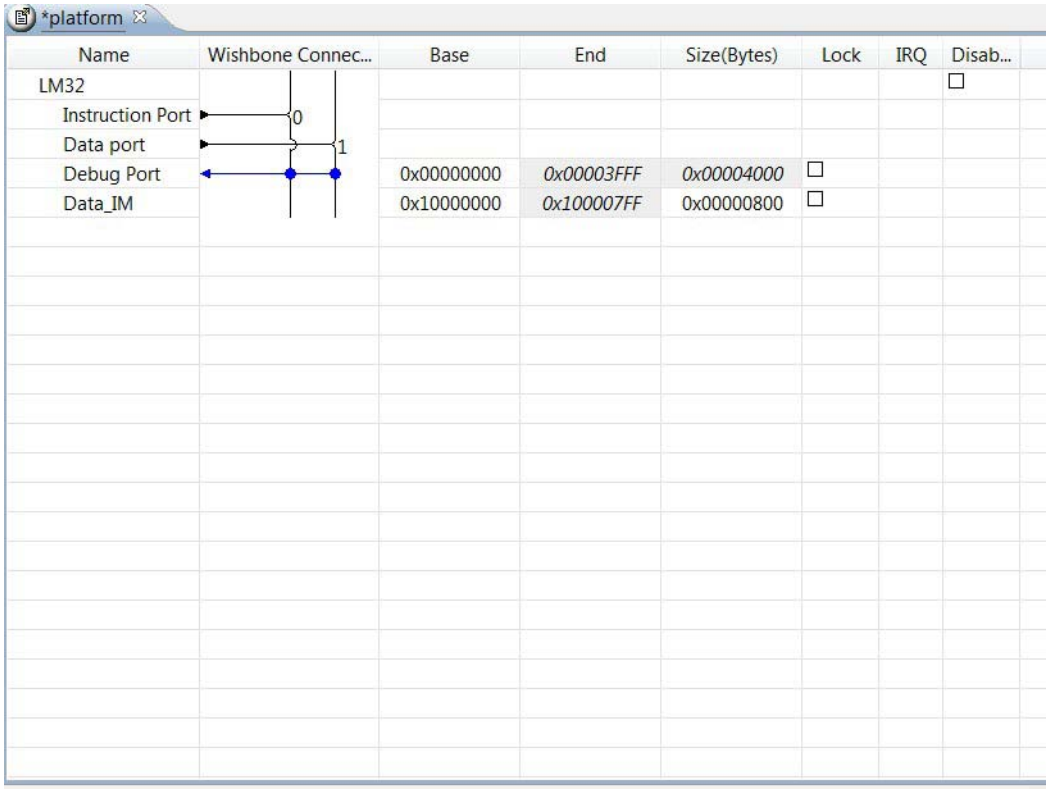
d. Click **OK** to accept the default settings for the rest of the options. Information about the microprocessor now appears in the Name, Wishbone Connection, Base, End, and Size columns of the table in the Editor view.

Figure 13: Add LatticeMico32 Dialog Box – Inline Memory Tab



The MSB perspective now shows the LatticeMico32 microprocessor in the Editor View, as shown in Figure 14.

Figure 14: MSB Perspective with Microprocessor



The Wishbone Connection column graphically displays the types of ports and connections. Black horizontal lines with outbound arrows indicate master ports, whereas blue horizontal lines with inbound arrows indicate slave ports. The vertical lines are associated with master ports, and the filled circles indicate connections between master and slave ports. The illustration shows that the microprocessor's slave Debug Port is connected to the master Instruction Port and Data Port.

Add On-Chip and Off-Chip Memory

Next you will add the LatticeMico FPGA Embedded Block RAM (EBR) and SPI Flash memory.

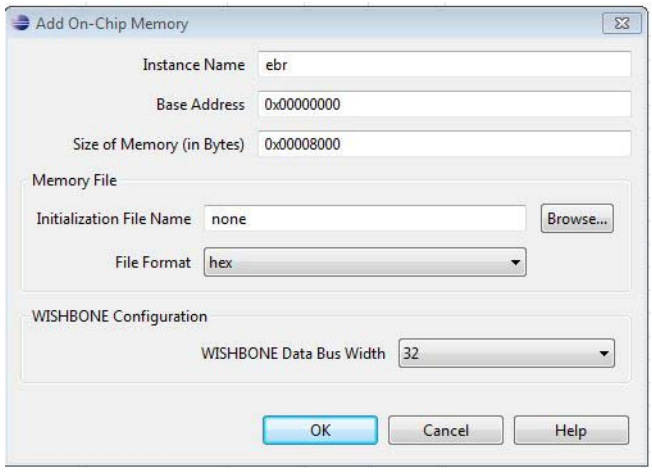
Add EBR

The LatticeMico on-chip memory controller is required to download and execute the software application code.

To add the EBR to the platform:

1. Under Memory in the Available Components view, double-click **On-Chip-Memory** to open the dialog box. Alternatively, you can select On-Chip-Memory and then click the Add Component button ( ).
2. In the Add On-Chip-Memory dialog box, change the size to 0x00008000, as shown in Figure 15, and click **OK** to accept the default settings for the rest of the options.

Figure 15: Add On-Chip-Memory Dialog Box



Note

You can delete a component from the Editor view by right-clicking the component in the Editor view and selecting Remove Component from the pop-up menu. If you cannot remove a component, this command will be unavailable on the menu.

3. Accept the default settings in the dialog box, and click **OK**.

Add SPI Flash Memory

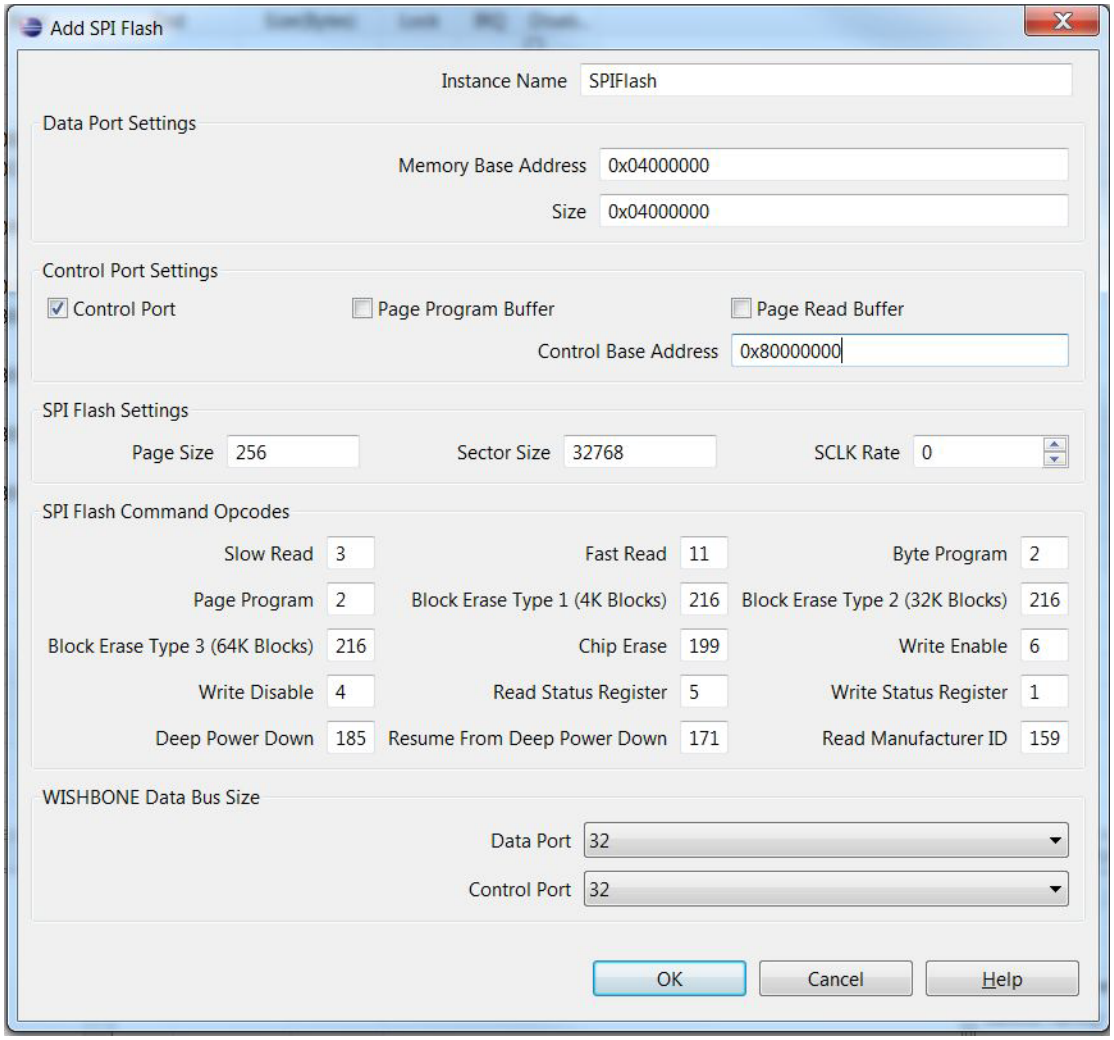
Next you will add the SPI Flash memory. The LatticeECP3 Versa Evaluation Board has a serial SPI (16-pin TSSOP, 64M) Flash memory device for non-volatile configuration memory storage. This flash will be used for software deployment.

To add the SPI flash component to the platform:

1. Under Memory in the Available Components view, double-click **SPI Flash** to open the dialog box. Alternatively, you can select SPI Flash then click the Add Component button ( ).

2. In the Add SPI Flash dialog box, shown in Figure 15 on page 21, do the following:

Figure 16: Add SPI Flash Dialog Box



a. In the Memory Base Address box, change the address to 0x04000000.

b. In the Size box, change the size to 0x04000000. The parallel Flash memory is placed at address 0x04000000 because of the address decode scheme used by the LatticeMico system. All components in a LatticeMico32 platform must be aligned to an address that corresponds to the largest size (bytes) entry. The LatticeMico32 address space is divided into two 2 GByte ranges. Addresses below the 2 GB boundary are memory components. Addresses above the 2 GB boundary are I/O components. The alignment of components is based on the memory range in which they reside. In this tutorial, the largest memory block is the 32 MByte parallel Flash component. This means that all memory components must be 32 MB-aligned.

Therefore, valid base addresses for memory components are 0x00000000, 0x4000000, 0x8000000, and so forth.

c. Select the control port option and in the Control Base Address box, change the address to 0x80000000.

   Control port, enables WISHBONE slave port that can perform all SPI flash commands such as chip/sector erase, write enable/disable, read/write status register, power up/down and the minimum boundary alignment is 0x800.

d. Click **OK** to accept the default settings for the rest of the options.

Add the Peripheral Components

Now you will add the peripheral components to the platform.

Add GPIO

The first peripheral component that you will add is the LatticeMico GPIO component, which provides a memory-mapped interface between a WISHBONE port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic or to I/O pins that connect to devices external to the FPGA

For this tutorial, two GPIO block are used: one for controlling the LEDs and the other for controlling the 14-segment display on the LatticeECP3 Versa Evaluation Board.

To add the GPIO to the platform:

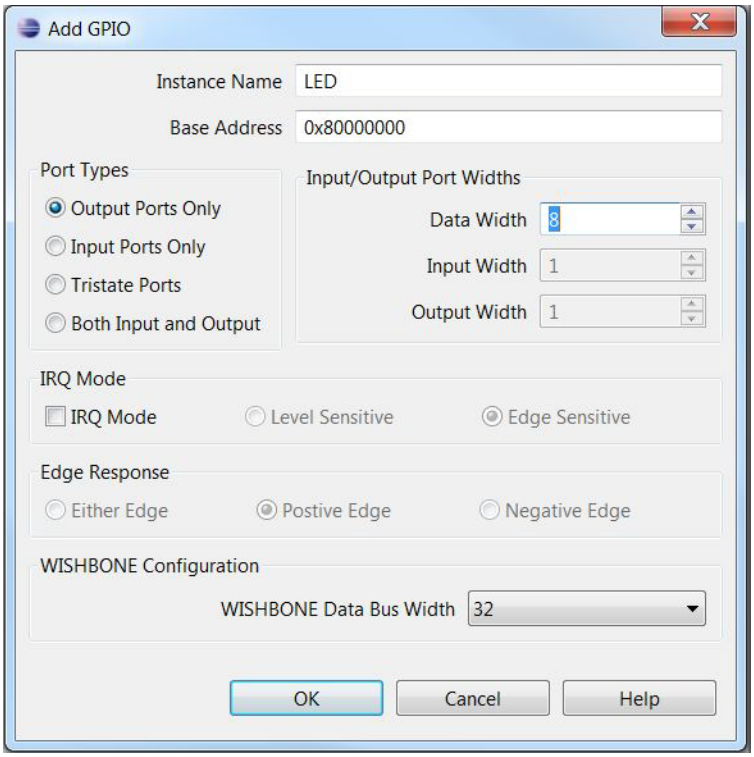
1. Under IO in the Available Components view, double-click **GPIO**. Alternatively, you can select GPIO, then click the **Add Component** button ( ).
2. In the Add GPIO dialog box, shown in Figure 17 on page 24, do the following:

   a. In the Instance Name box, change the name of the GPIO to **LED**.

      For this tutorial, the GPIO block must be named LED. Failure to name the GPIO block LED will cause mismatches in the FPGA I/O pin names. The example C source code uses this instance name to access the GPIO registers.

   b. Change the setting of the Data Width option to **8**.
3. Click OK to accept the default settings for the rest of the options.

Figure 17: Add GPIO Dialog Box - First Instance



To add the second instance of GPIO component to control the 14-segment display:

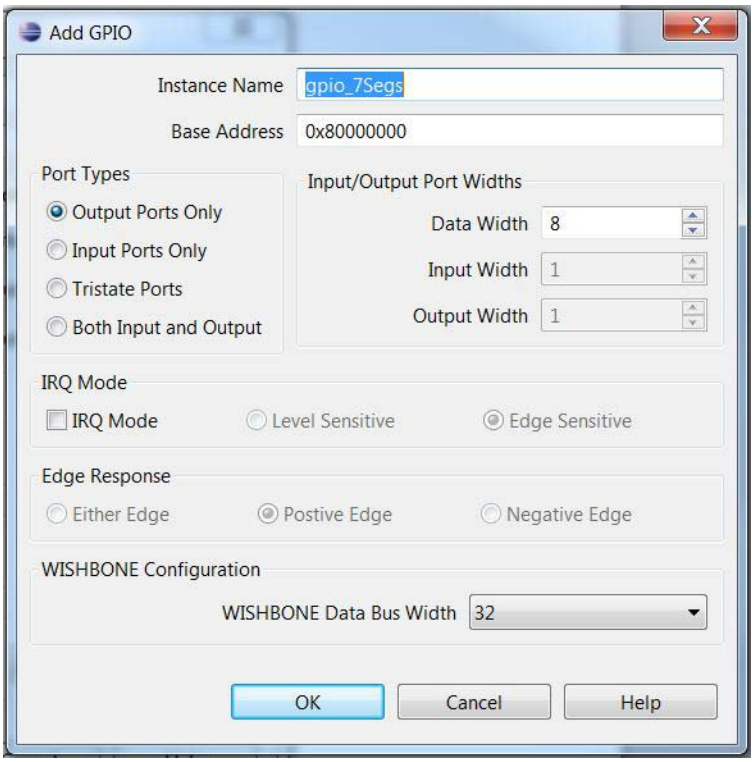
1. Under IO in the Available Components view, double-click **GPIO**. Alternatively, you can select GPIO, then click the **Add Component** button ( ).
2. In the Add GPIO dialog box, shown in Figure 18 on page 25, do the following:
   a. In the Instance Name box, change the name of the GPIO to **gpio_7Segs**.
   b. Change the setting of the Data Width option to **8**.
3. Click OK to accept the default settings for the rest of the options.

Add the Timer

The final component that you will add is a LatticeMico timer. This component is a highly configurable countdown timer with a WISHBONE-compliant slave interface compatible with the LatticeMico32 microprocessor.

In this tutorial, the timer is used to generate an interrupt for every 100 milliseconds. Depending on this interrupt, the 14-segment count value is controlled.

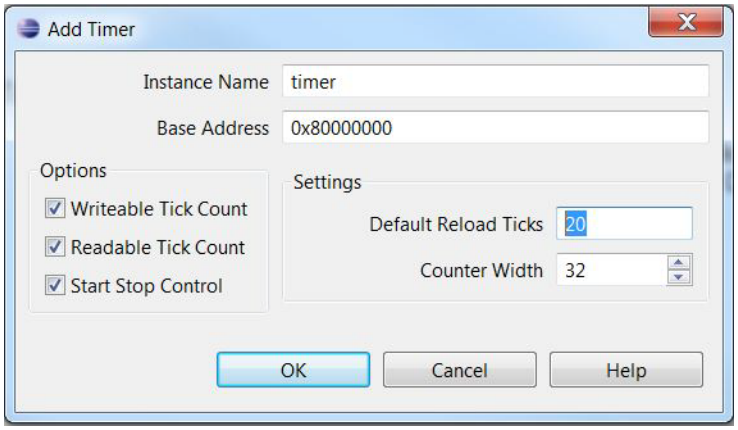
Figure 18: Add GPIO Dialog Box - Second Instance



To add the Timer component to the platform:

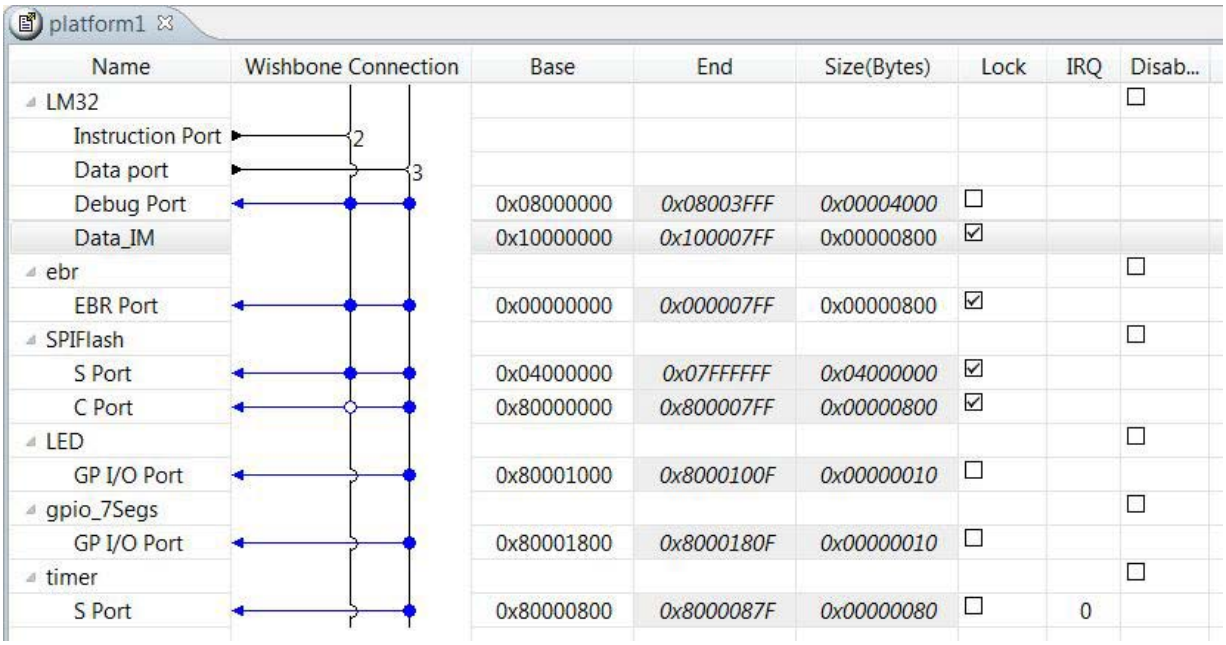
1. Under IO in the Available Components view, double-click **Timer**. Alternatively, you can select Timer, then click the Add Component button ( ).
2. Click **OK** to accept the default settings.

Figure 19: Add Timer Dialog Box



The MSB perspective now resembles the illustration in Figure 20.

Figure 20: MSB Perspective After Addition of All Components in a Shared-Bus Arbitration Scheme



Specify the Connections Between Master and Slave Ports

The connections that you will make between the master and slave ports in the Editor view will reflect the access scheme shown in Figure 1 on page 4.

The following information applies to master and slave ports in the Editor view:

- There are two types of ports: master ports and slave ports.
  - A master port can initiate read and write transactions.
  - A slave port cannot initiate transactions but can respond to transactions initiated by a master port if it determines that it is the targeted component for the initiated transaction.
- A master port can be connected to one or more slave ports.
- A component can have one or more master ports, one or more slave ports, or both.
- Horizontal lines with outbound arrows sourced from a component port indicate a master port.
- Horizontal lines with inbound arrows targeting a component port indicate a slave port.
- The vertical lines are associated with horizontal lines with outbound arrows (that is, master ports) to facilitate "connectivity" from a master port

to a slave port. A circle represents the intersection of the vertical line and a horizontal line associated with a slave port.

- A filled circle indicates a connection between the master port represented by the vertical line and the slave port represented by the horizontal line associated with the filled circle.
- A hollow circle indicates an absence of connection between the master port represented by the vertical line and the slave port represented by the horizontal line associated with the hollow circle. This can be seen in Figure 20 on page 26, where only the LatticeMico32 microprocessor Wishbone ports are connected.
- The numbers next to the lines representing the master ports are the priorities in which the master ports can access the slave ports. You can change the priority of these connections by following the instructions in the online Help for LatticeMico System.

To specify the connections between master and slave ports:

1. Connect the instruction and data ports to the LatticeMico EBR slave port by clicking both circles in the Wishbone Connection column of the ASRAM Port row.
2. Connect the instruction and data ports to the LatticeMico SPI Flash slave port by clicking both circles in the Wishbone Connection column of the Data Port row.
3. Connect the data port to the LatticeMico GPIO slave port by clicking the circle in the Wishbone Connection column of the GP I/O Port row.
4. Connect the data port to the LatticeMico Timer slave port by clicking the circle in the Wishbone Connection column of the timer port row.

   Figure 21 on page 28 shows the resulting connections in the Editor view.

This tutorial example uses the shared-bus arbitration scheme. For information about bus arbitration schemes, refer to the *LatticeMico32 Software Developer's User Guide*. The master ports are represented by black lines, and the slave ports are represented by blue lines. Both the instruction and data ports connect to the LatticeMico EBR and the SPI Flash Controller Data port,

but only the data port connects to the LatticeMico SPI Flash Controller, the LatticeMico GPIO, and the LatticeMico UART.

Figure 21: Connections for Shared Bus Arbitration

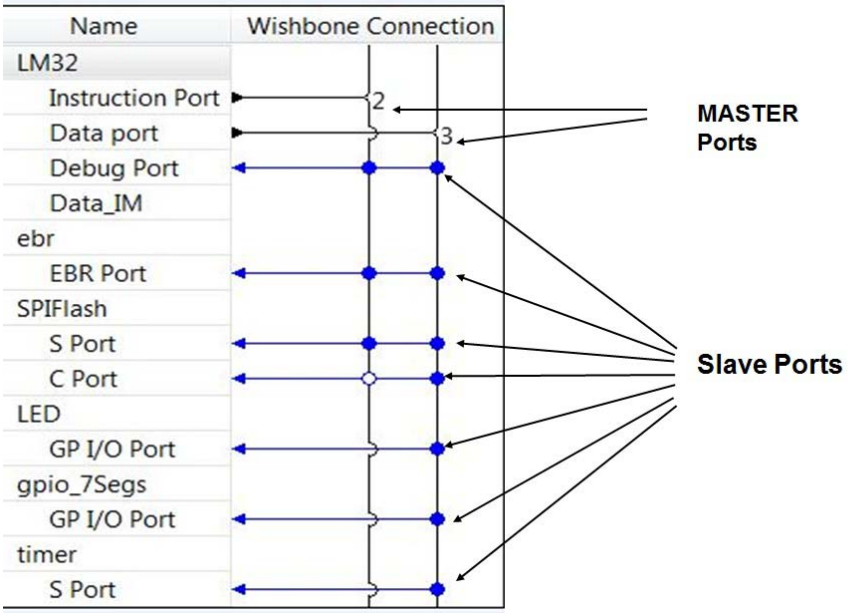
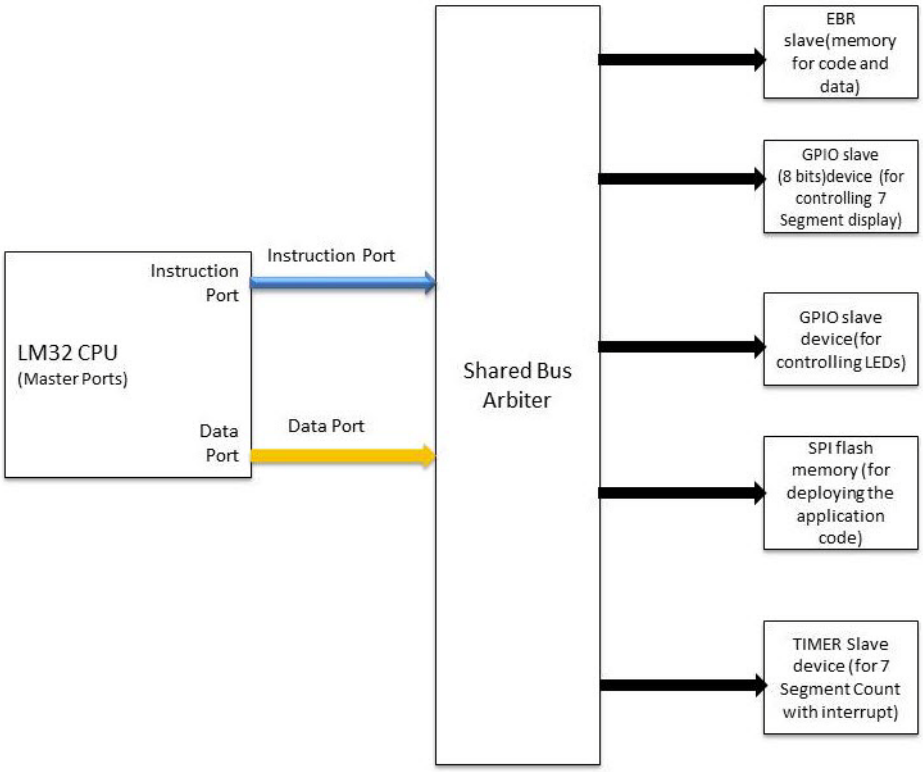


Figure 22 shows the connections generated by MSB. MSB automatically generates the arbiter, depending on which arbitration scheme is selected. In the case of the shared-bus arbitration scheme, it generates the microprocessor platform to allow multiple master ports access to multiple slave ports over a single shared bus. In the diagram, the instruction port accesses the LatticeMico On-Chip Memory Controller, the LatticeMico GPIO, and the SPI flash controller. The data port accesses the LatticeMico On-Chip

Memory controller, the LatticeMico GPIO, the LatticeMico TImer, and the LatticeMico SPI Flash Controller Data and control port.

Figure 22: Connections Generated by MSB



Assign Component Addresses

The next step is for MSB to generate an address for each component with slave ports. Addresses are specified in hexadecimal notation. Components with master ports are not assigned addresses.

Note

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

You will not assign individual addresses. There are only two addresses that need to be manually assigned: the Debug Memory and the SPI Flash Memory.

During the creation of the SPI Flash component, you explicitly assigned an address (0x04000000) to the SPI flash component data and control port, and the Inline Data memory. You must lock the parallel flash address so that MSB will not automatically assign it a new address. You do not want the flash

address to change for this example, because that is where the final software application code will reside.

To lock the address:

1. In the Lock column, select the box for the SPI flash data and control port (SPIFlash).
2. In the Lock column, select the box for the **Debug Port**.

To automatically assign component addresses:

- Choose **Platform Tools > Generate Address**, or click the **Generate Base Address** button (), or right-click in the Editor view and choose **Generate Address** from the pop-up menu.

  The addresses now appear in the Base and End columns in the Editor view, in hexadecimal notation. Slave components that are not memories are assigned addresses within the 0x80000000-0xFFFFFFFF memory range. The Generate Address command sets A31 of each of the I/O components to '1'.

  Note

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

  Your MSB perspective should now resemble the example shown in Figure 23. The base addresses that you see in your Editor view might be different from those shown.

Figure 23: MSB Perspective After Assignment of Addresses in a Shared-Bus Arbitration Scheme

platform1

Name	Wishbone Connection	Base	End	Size(Bytes)	Lock	IRQ	Disab...
LM32							☐
Instruction Port	2						
Data port	3						
Debug Port		0x00000000	*0x00003FFF*	*0x00004000*	☑		
Data_IM		0x10000000	*0x100007FF*	0x00000800	☐		
ebr							☐
EBR Port		0x00000000	*0x000007FF*	0x00000800	☐		
SPIFlash							☐
S Port		0x04000000	*0x07FFFFFF*	*0x04000000*	☑		
C Port		0x80000000	*0x800007FF*	*0x00000800*	☑		
LED							☐
GP I/O Port		0x80000000	*0x8000000F*	*0x00000010*	☐		
gpio_7Segs							☐
GP I/O Port		0x80000000	*0x8000000F*	*0x00000010*	☐		
timer							☐
S Port		0x80000000	*0x8000007F*	*0x00000080*	☐	--	

Assign Interrupt Request Priorities

The interrupt request priority is the order in which hardware components request computing time from the CPU. Now you will assign an interrupt request priority (IRQ) to all components that feature a dash in the IRQ column of the Editor view. You cannot assign interrupt priorities to components lacking this dash in the IRQ column, such as memories and CPUs.

To assign interrupt priorities for all components providing interrupt functionality:

- Choose **Platform Tools > Generate IRQ**, or click the **Generate IRQ** button (), or right-click in the Editor view and choose **Generate IRQ** from the pop-up menu.

Note

To reassign an interrupt priority for a specific component, go to the IRQ column in the row for the component, click on the current interrupt priority number, and choose the new priority number from the drop-down menu. Explicitly assigned interrupt priorities will not be overridden by the interrupt generator tool. The Lock control does not affect IRQ assignment; it only prevents auto-assignment of the Base Address.

If you accidentally assign duplicate priorities, MSB will issue an error message in the Console view when you select **Platform Tools > Generate IRQ**.

Perform a Design Rule Check

You will want to perform a design rule check to verify that components in the platform have valid base addresses, interrupt request values, and other fundamental properties.

To perform a design rule check:

- Choose **Platform Tools > Run DRC**, or click the **Run DRC** button (), or right-click in the Editor view and choose **Run DRC** from the pop-up menu.

In the Console view, MSB shows that there are no errors in the platform.

Generate the Microprocessor Platform

You are now ready to generate the microprocessor platform. During the generation process, MSB creates the following files in the ..\Tutorial\lm32_tutor\platform1\soc directory:

- A platform1.msb file, which describes the platform. It is in XML format and contains the configurable parameters and bus interface information for the components.

- A platform1.v (Verilog) file, which is used by both Verilog and mixed Verilog/VHDL users:
  - For Verilog users, the platform1.v file is used in both simulation and implementation. It instantiates all the selected components and the interconnect described in the MSB graphical user interface. This file is the top-level simulation and synthesis RTL file passed to Diamond. It includes the .v files for each component in the design. These .v files are used to synthesize and generate a bitstream to be downloaded to the FPGA. The first time Generate is run, the Verilog source for each component in the platform, which is located in <Diamond_install_path>/micosystem/components/<component>, is copied into a subdirectory called “components.” The components subdirectory is a sibling to the *soc* subdirectory.
  - For mixed Verilog/VHDL users, the platform1.v file is used in simulation only.

    A mixed-mode Verilog and VHDL simulator, such as Aldec® Active-HDL™, is needed for functional simulation.
- A platform1_vhd.vhd (VHDL) file is created if you selected the “Create VHDL Wrapper” option in the New Platform Wizard dialog box. It is intended to be used only to incorporate the Verilog-based platform into a mixed Verilog/VHDL design. The platform1_vhd.vhd contains the top-level design used for synthesis. This top-level design file instantiates the platform1 component.

The contents of the platform1.msb file are used by the C/C++ development tools. The C/C++ source code build process extracts the base address information and the size of each component and uses the information to build GNU LD linker files. Each time the Generate function is run, it causes the C/C++ compiler to consider the C/C++ source code to be out of date. This means that the source code will be rebuilt from scratch after each Generate process.

To generate the microprocessor platform:

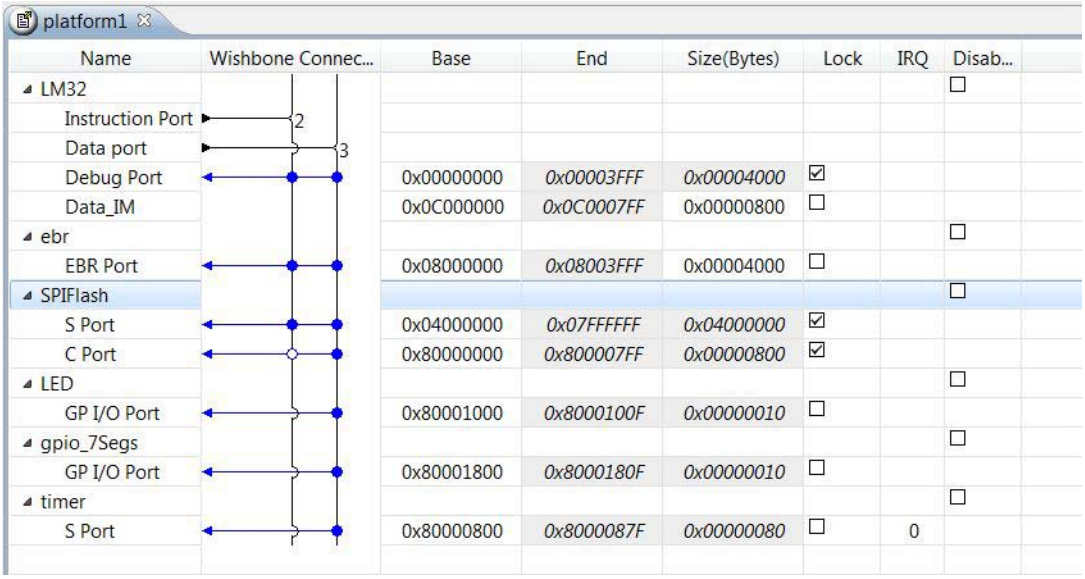
- Click anywhere in the Editor view and choose **Platform Tools > Run Generator**, or click the **Run Generator** button (), or right-click and choose **Run Generator** from the pop-up menu.

  The Console view displays the output as MSB processes the design.

  If you are using Verilog, you will see `Finish Generator` in the Console view when the generator is finished. If the project was created with the “Create VHDL Wrapper” option selected, the project is a mixed Verilog/VHDL flow and the generator silently launches Synplify synthesis and Diamond to create the wrapper. If you are using mixed Verilog/VHDL, you must wait for the `Finish VHDL Wrapper` message to appear in the Console view.

The MSB perspective now looks like the illustration in Figure 24. The assigned addresses for the components other than the parallel flash might differ.

Figure 24: MSB Perspective After Building the Microprocessor Platform in a Shared-Bus Arbitration Scheme



As shown in Figure 25, MSB generates a platform1_inst.v file, which contains the Verilog instantiation template for use in a design where the platform is not the top-level module. For a mixed Verilog/VHDL project, no equivalent file is generated.

Figure 25: Verilog Instantiation Template

```
platform1 platform1_u (
.clk_i(clk_i),
.reset_n(reset_n)
, .SPIFlashCEJ(SPIFlashCEJ) //
, .SPIFlashSCK(SPIFlashSCK) //
, .SPIFlashSI(SPIFlashSI) //
, .SPIFlashSO(SPIFlashSO) //
, .SPIFlashWPJ(SPIFlashWPJ) //
, .LEDPIO_OUT(LEDPIO_OUT) // [8-1:0]
, .gpio_7SegsPIO_OUT(gpio_7SegsPIO_OUT) // [8-1:0]
);
```

Figure 26 on page 35 shows the structure of the directory that MSB generates. The directory structure is created the first time the Generate process is run. The contents of the components subdirectory is only written the very first time the Generate function is run. After the first run it remains static. There is an exception: when a MSB project is opened after installing a

new version of the LatticeMico System Builder, a new component version might exist. You are given an opportunity to update to the new component. Accepting the update will modify the components subdirectory.

Note

Figure 26 shows an example platform. The figure does not show the entire directory and file structure.

Create a User Top-Level Module

As mentioned in the previous section, the platform that is generated by MSB may not be the top-level module. In this scenario, you can use the contents of platform1_inst.v, as shown in Figure 25 on page 33, within your top-level module. This instantiates the platform generated by MSB within your top-level module.

You must now create a top level file name platform1_top.v to instantiate the MSB-generated platform. Create the file as follows:

1. Copy and paste the contents of Figure 27 on page 36 into a text editor.
2. Name the file platform1_top.v, and save the file in the following directory:

- For Windows:
  < *Diamond_install_path*>\examples\lm32_tutor\platform1\soc
- For Linux: ~/LatticeMico32/platform1/soc

You will import this file into your Diamond project, in the section “Importing the Source Files” on page 46.

Figure 26: MSB Directory Structure

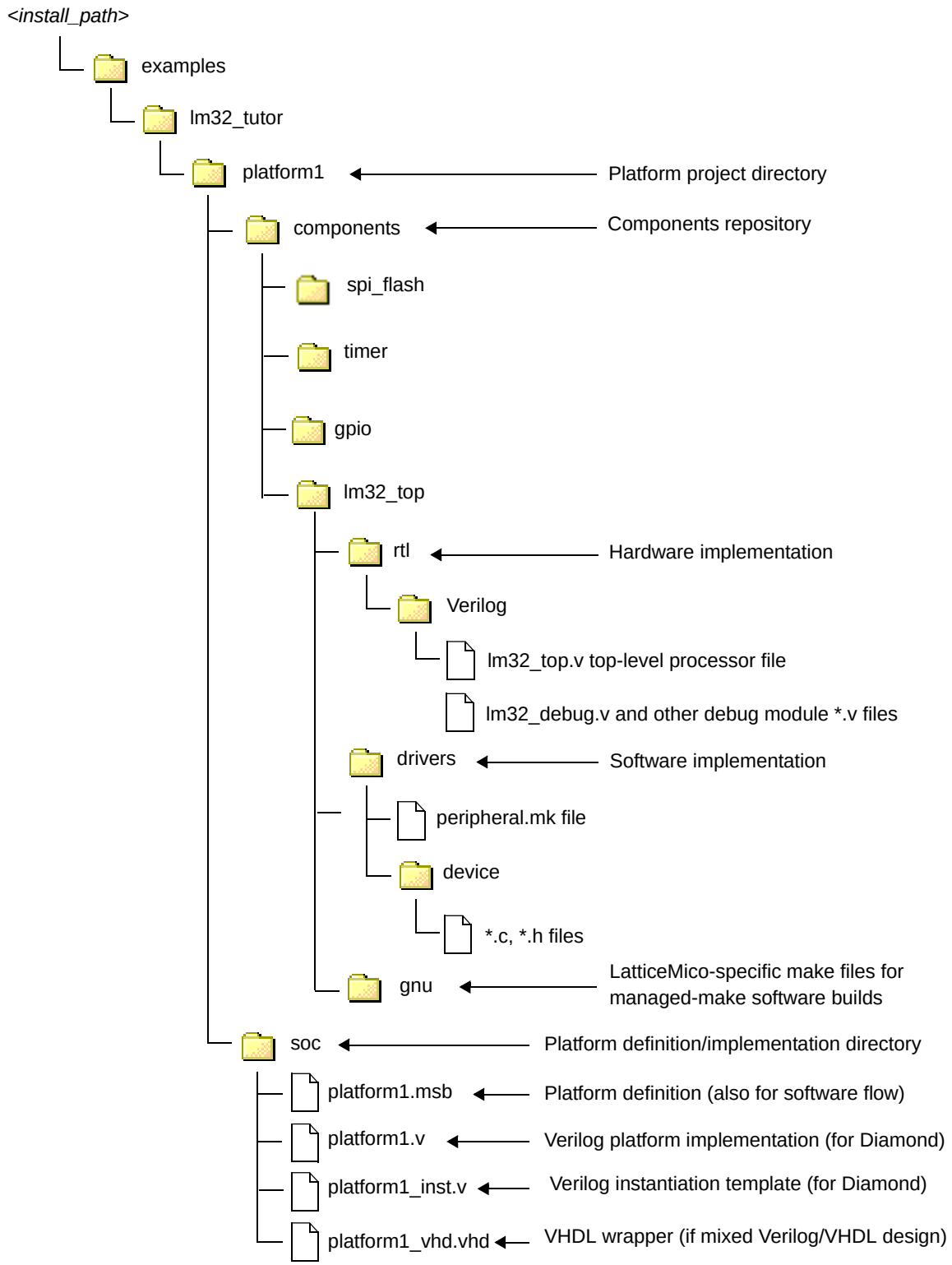


Figure 27: User Top Level File

```
`include "platform1.v"

module platform1_top
  (input reset_n,
   output SPIFlashCEJ,
   output SPIFlashSCK,
   output SPIFlashSI,
   input SPIFlashSO,
   output SPIFlashWPJ,
   output [7:0] LEDPIO_OUT,
   output [7:0] gpio_7SegsPIO_OUT
  );

// LatticeECP3 internal oscillator generates platform clock
wire clk_i;
OSCF OSCinst0 (.OSC(clk_i));
defparam OSCinst0.NOM_FREQ = "26" ;

platform1 platform1_u
  (.clk_i (clk_i),
   .reset_n (reset_n),
   .SPIFlashCEJ (SPIFlashCEJ) ,
   .SPIFlashSCK (SPIFlashSCK) ,
   .SPIFlashSI (SPIFlashSI) ,
   .SPIFlashSO (SPIFlashSO) ,
   .SPIFlashWPJ (SPIFlashWPJ) ,
   .LEDPIO_OUT (LEDPIO_OUT) ,
   .gpio_7SegsPIO_OUT (gpio_7SegsPIO_OUT)
);
endmodule
```

Task 3: Create the Software Application Code

In this task, you create the software application by using C/C++ in the LatticeMico System Software Project Environment (C/C++ SPE). The software application is the code that runs on the LatticeMico32 microprocessor to control the components, the bus, and the memories. The application is written in C/C++.

C/C++ SPE is based on the Eclipse environment and provides an integrated development environment for developing, debugging, and deploying C/C++ applications. C/C++ SPE uses the GNU C/C++ tool chain (compiler, assembler, linker, debugger, and other necessary utilities) that has been customized for the LatticeMico32 microprocessor.

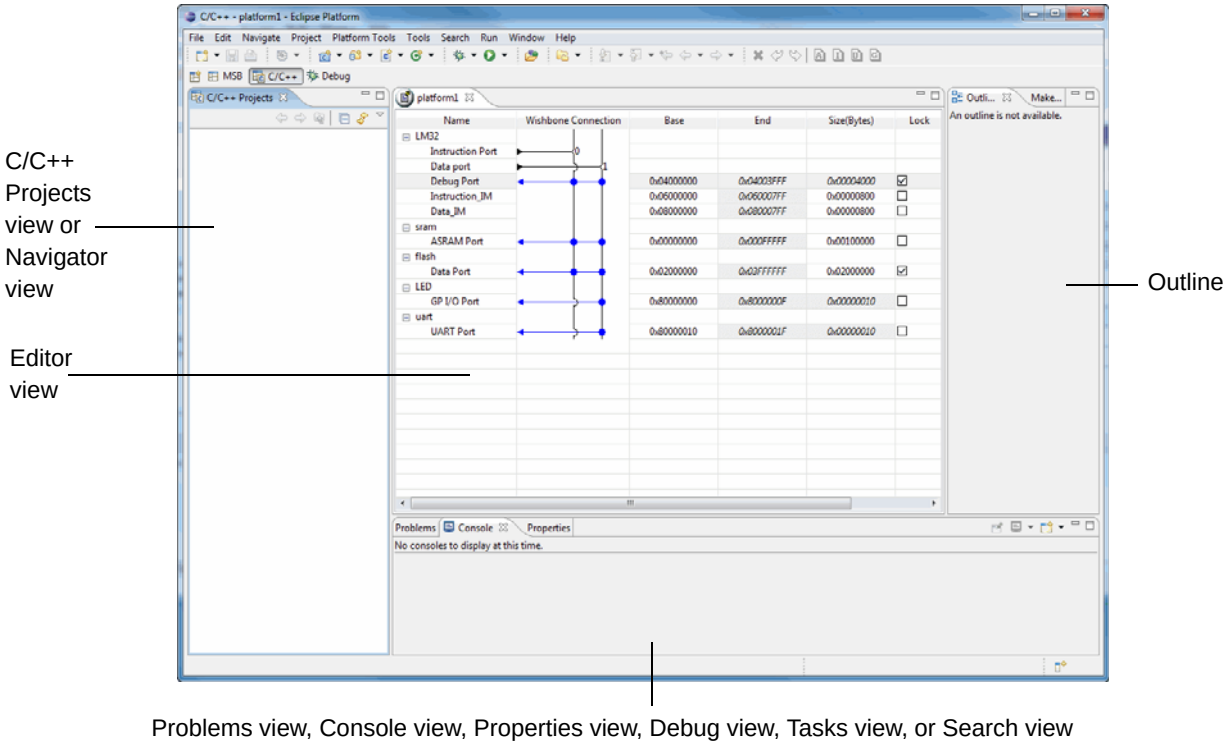
C/C++ SPE uses the same LatticeMico System interface as MSB, but it uses a different perspective called the C/C++ perspective.

To activate the C/C++ perspective:

- In the upper left-hand corner of MSB graphical user interface, select **C/C++**.

The C/C++ perspective is shown in Figure 28.

Figure 28: 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's files under the workspace directory
- Editor view, which is similar to the Editor view in the MSB perspective
- Outline view, which displays the structure of the file currently open in the Editor view
- Problems view, which displays any error, warning, or informational messages output by C/C++ SPE
- 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 C/C++ Projects view. This view is read-only.
- Search view, which displays the results of a search when you choose Search > File.
- Tasks view, which shows the tasks running concurrently in the background
- Make Targets view, which is not used in LatticeMico C/C++ projects

Create a New C/C++ SPE Project

You will create a new project in C/C++ SPE, import the platform1.msb file into the project, select the application code template to use so that you do not have to write the code yourself, and compile the code.

To create a new C/C++ SPE project:

1. In the C/C++ perspective, choose **File > New > Mico Managed Make C Project**.
2. In the New Project dialog box, make the following selections:
   a. In the Project Name box, enter **LED_Versa**.
   b. In the Location box, browse to the following directory:
      - For Windows: <*Diamond_install_path*>\examples\lm32_tutor
      - For Linux: ~/LatticeMico32/lm32_tutor/platform1
   c. In the MSB System box, browse to the following location, select the **platform1.msb** file in the dialog box, and click **Open**.
      - For Windows: <*Diamond_install_path*>\examples\lm32_tutor\platform1\soc\platform1.msb
      - For Linux: ~/LatticeMico32/lm32_tutor/platform1/soc/platform1.msb

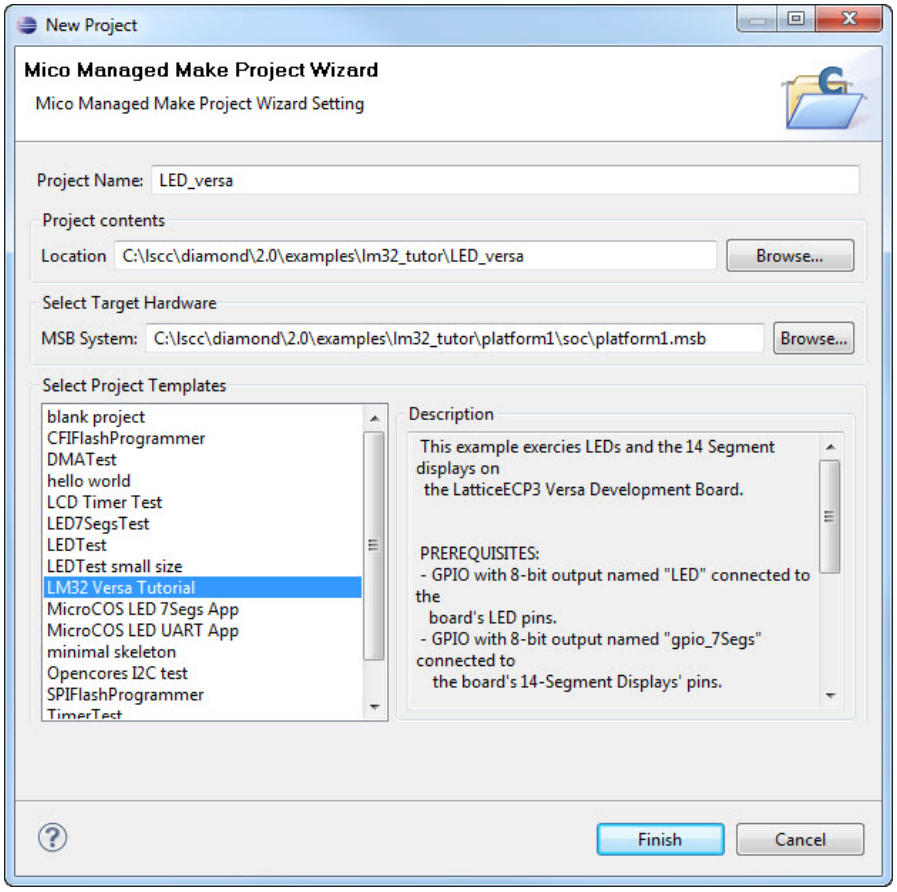
d. In the Select Project Templates box, select **LM32Tutorial** as the template for the application code.)

Note

Project templates are packaged software application files that are copied to the new project and provide a starting point for building an application. Some templates have specific requirements, as described in the description pane. If these hardware and software requirements are not met, the application built may not function correctly and may require you to debug the application by using the C/C++ SPE debug interface. C/C++ SPE enables you to create templates in addition to those included with the installation.

The New Project dialog box should resemble the figure shown in Figure 29.

Figure 29: New Project Dialog Box



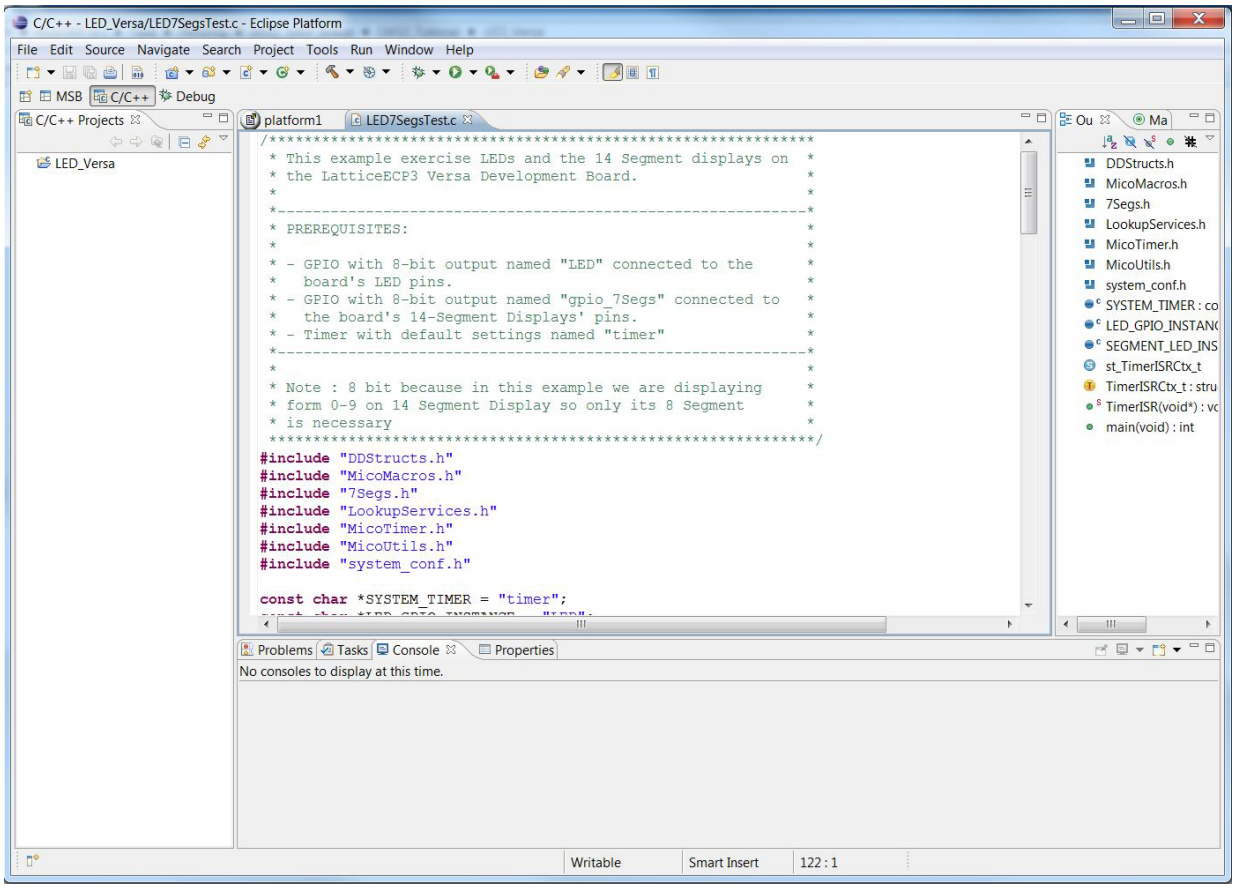
Note

The directory shown in the Location box in the Project Contents field is where the software project directory will be created. Your user files will be placed in this directory.

3. Click **Finish**.

Now you see the source code in the middle pane of the C/C++ perspective, as shown in Figure 30.

Figure 30: Source Code in C/C++ Perspective



Linker Configuration

A new C project is almost ready to be compiled and linked. Before you compile the source code, it is necessary to configure the linker. Every C/C++/ assembly file has, at a minimum, three fundamental sections that need to be placed.

The compiler splits the source code into an instruction section, a read-only data section, and a read-write section by default. The first two sections can be placed in either read-only or read-write memories, while the final section must be placed in a read-write memory. The C/C++ SPE provides you with an easy-to-use feature for selecting memories for each region.

Your platform contains three memory components: a data inline memory, a parallel flash memory, and an asynchronous SRAM memory. You will build the LED_Versa application to run from the asynchronous SRAM memory and data inline memory.

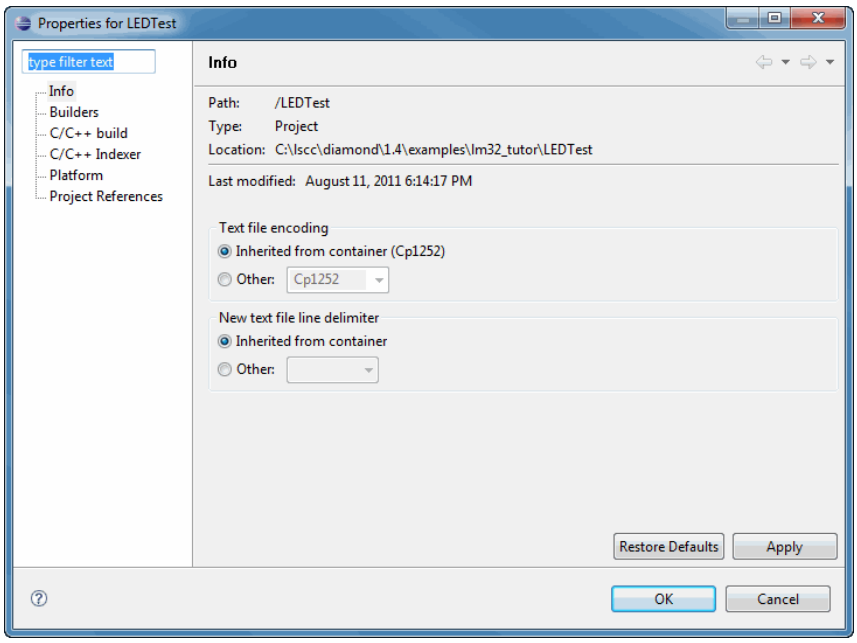
The Properties dialog box enables you to select and change where the linker places each of the sections.

To modify how the linker assigns each section:

1. Make sure that the LED_Versa is selected in the C/C++ Projects view.
2. Choose **Project > Properties.**

   The Properties for LED_Versa dialog box now appears, as shown in Figure 31.

Figure 31: Properties for LED_Versa Dialog Box



You can select from the list on the left side of the Properties window to open one of the following panes:

- Info – provides basic project location information.
- Builders – provides information on the builder system used for this managed build project. It is preconfigured to use the LatticeMico builder system.
- C/C++ Build – enables you to select and manage the compiler, assembler, and linker settings.
- C/C++ Indexer – enables you to specify the indexing method for searches: fast, full, or no indexer.
- Platform – provides information on the platform used by this project, in addition to other information such as the linker section setting.
- Project References – enables you to manage other projects referenced by the current project. Project References cannot be used for the LatticeMico C/C++ SPE managed build environment.

3. Select the **Platform** pane.

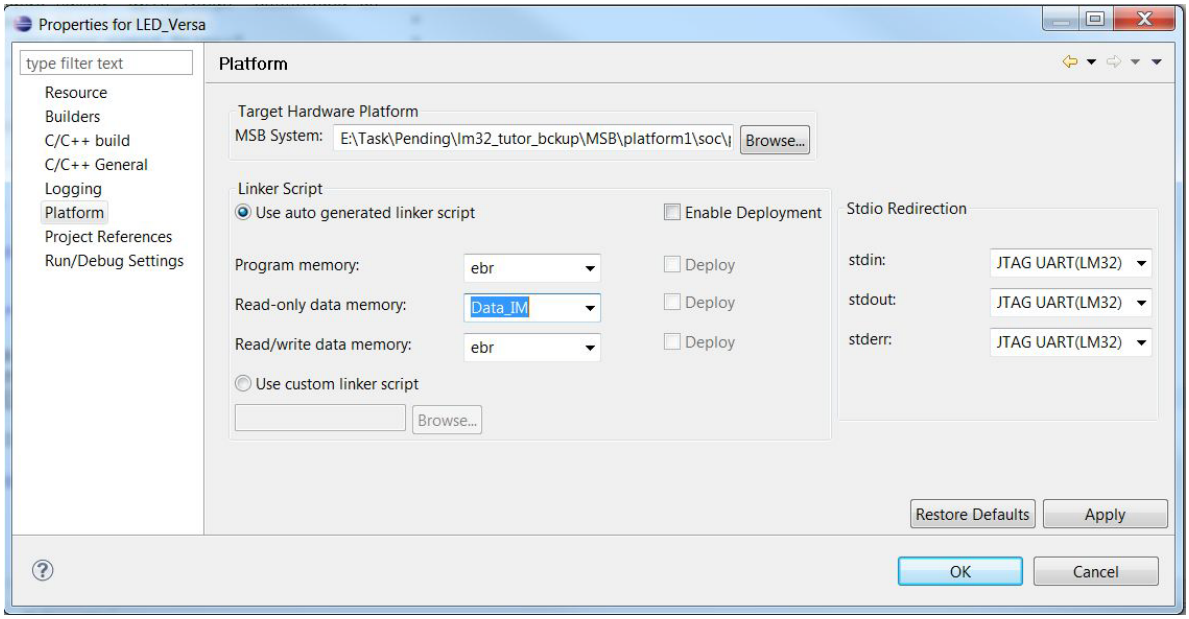
   The Target Hardware Platform text box shows the current MSB platform. You can change the hardware platform used by the software application, but you must rebuild the software application.

   The options in the Linker Script section enable you to select your own linker script. However, for this tutorial you will use the auto-generated (default) linker script. For the auto-generated linker script, you can specify the memories that will be used for the linker sections. The C/C++ SPE managed build process inspects the specified MSB platform to determine the available memory regions. As a default, the C/C++ SPE managed build process selects the largest read/write memory available to contain all the sections. For this tutorial, you will select the SRAM for program and read/write data memory sections, and it will select Data Inline Memory for read-only data memory sections.

4. In the Linker Script section, make the following selections from the drop-down menus, as shown in Figure 32:

   - For Program memory, select **ebr**.
   - For Read-only data memory, select **Data_IM**.
   - For Read/write data memory, select **ebr**.

Figure 32: Platform Pane of the Properties for LED_Versa Dialog Box



5. Click **OK** to return to the C/C++ perspective.

Build the Project

The next step is to build the project, in which C/C++ SPE compiles, assembles, and links your application code, as well as the system library code provided by C/C++ SPE.

To compile the project:

- In the C/C++ Projects view (left-hand pane), select **LED_Versa** and choose **Project > Build Project**. Do not click on any of the buttons in the Build Project dialog box.

The compilation process generates the following files, among others, in the LED_Versa\platform1 directory:

- A C header file, DDStructs.h, that describes the device-driver structures for the applicable devices, in addition to the relevant platform settings, such as the microprocessor clock frequency
- A C source file, DDStructs.c, that describes the component instance parameters required by the device drivers in appropriate structures
- A C source file, DDInit.c, that invokes specified device initialization routines for putting the relevant instantiated components in a known state
- A linker script, linker.ld (in LED_Versa\platform1\Debug), that contains the location and size of the memory components and the rules for generating an executable file image, as required by the GNU linker. C/C++ SPE uses this information to ensure that the program code and data are located at the correct addresses. Although it is not covered in this tutorial, the LatticeMico C/C++ SPE enables you to easily specify a custom linker script to be used in lieu of the generated script for the managed build.
- A LatticeMico software executable linked formal file (.elf). The .elf file contains the Mico instructions, debug information, and information about the pre-initialized data. This tutorial generates a file called platform1.elf.

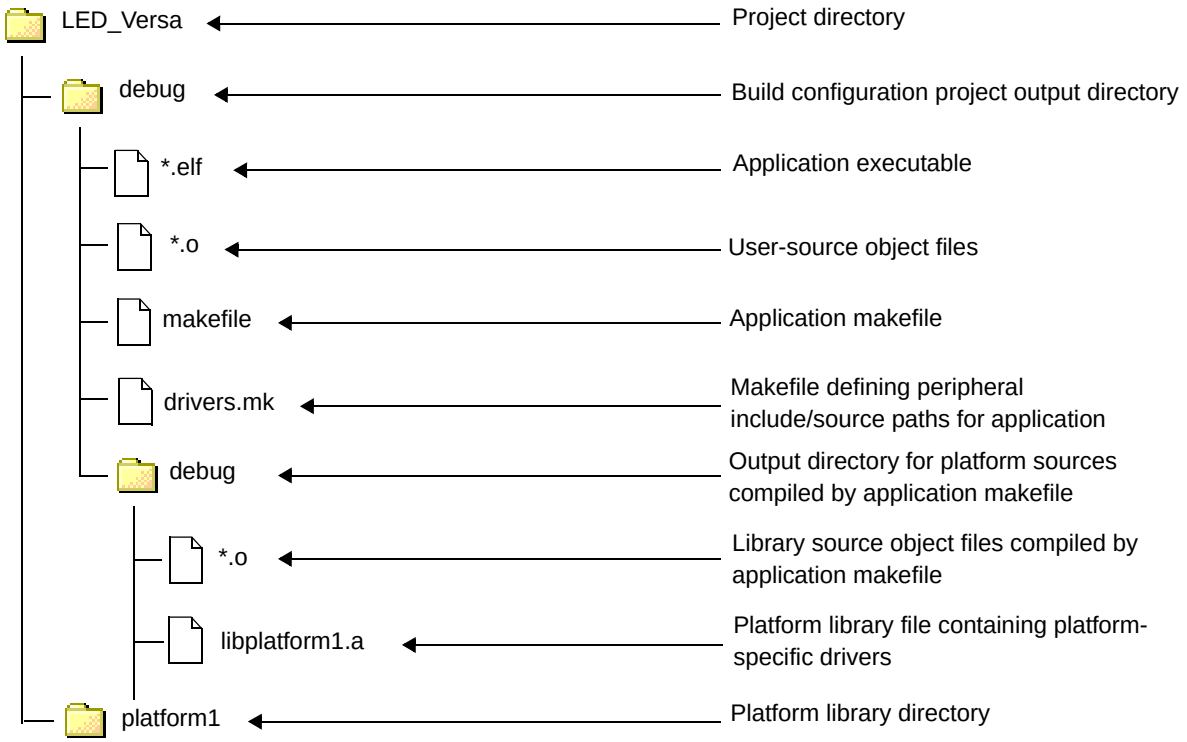
These files are included in the directory that C/C++ SPE generates in the background. The structure of this directory is shown in Figure 33 on page 44.

The contents of this directory are dynamically generated, and any changes to them are overwritten from build to build.

Note

Only the most important files are shown in Figure 33.

Figure 33: C/C++ SPE Directory Structure



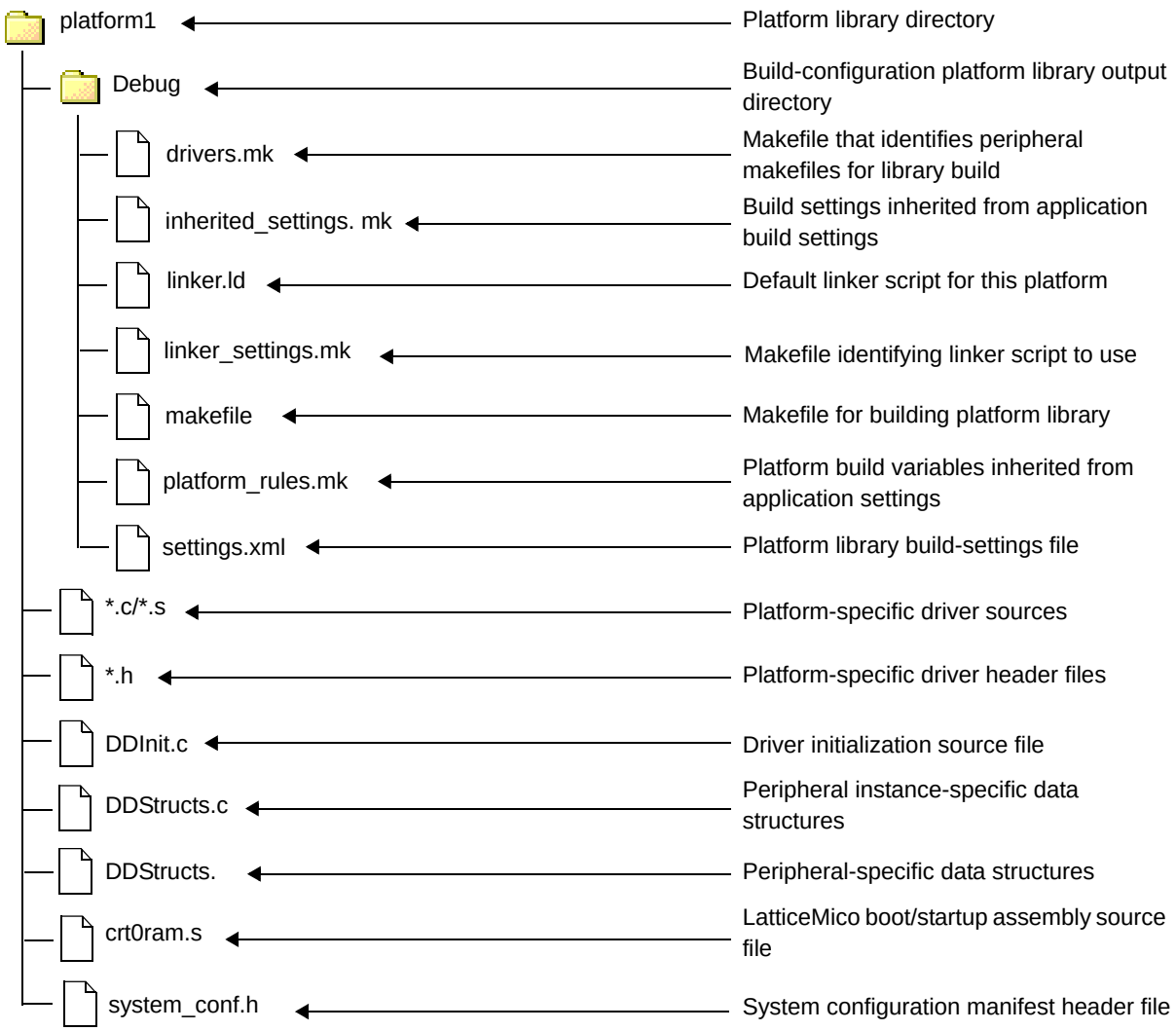
The platform1 library directory shown in Figure 33 contains platform-specific information for the building of an application.

Figure 34 shows the automatically generated files in this directory that are required to build an application. The contents of this directory are generated dynamically, and any changes to them are not preserved from build to build.

Note

Only the most important files are shown in Figure 34.

Figure 34: LED_Versa\Platform1 Library Directory Structure



Task 4: Generate the Microprocessor Bitstream

The next step in the flow is to generate the microprocessor bitstream file. This bitstream file is then downloaded to the FPGA on the circuit board. To generate the bitstream file, return to Diamond.

Import the MSB Output File

First, you must import the Verilog file output by MSB, the Verilog and VHDL files for mixed Verilog/VHDL, or the EDIF file created by the synthesis tool into Diamond.

The process of importing the generated platform file into Diamond is the same for Verilog and mixed Verilog/VHDL, except that you must import the VHDL wrapper file in addition to the Verilog file for mixed Verilog/VHDL.

Configure the Lattice Diamond Environment

The Diamond build process has the ability to operate in two different modes. One is to copy all HDL source files into the Diamond project directory, and the other is to reference them in their current directory structure. The LatticeMico build requires that the source files remain in the directory structure created by MSB. The default Diamond behavior is to leave the files where they are, but it is advisable to verify that Diamond is configured correctly.

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

Importing the Source Files

You can import the HDL source files generated by MSB into Diamond. If your design is in Verilog only, you will import the platform1.v file. If your design is a mixed Verilog/VHDL design, you will import both the platform1_vhd.vhd file and the platform.v file.

To import the Verilog or Verilog/VHDL files for the tutorial example:

1. In Diamond, choose **File > Add > Existing File**.
2. In the dialog box, browse to the ..\platform1\soc directory:
3. Select the **platform1_top.v file** (Verilog), and click **Add**.
4. If your design is mixed Verilog/VHDL, select both the **platform1_top.v** file and the **platform1_vhd.vhd** file and click **Add**.
5. If your design is mixed Verilog/VHDL, perform the following additional steps:
   a. Choose **Project > Property Pages**.
   b. In the dialog box, select the project name that appears in bold type next to the implementation icon .
   c. In the right pane, click inside the Value cell for “Top-Level Unit” and select ***<platform1>*_vhd** from the drop-down menu.

d. Click inside the Value cell for “Verilog Include Search Path,” and then click the browse button to open the “Verilog Include Search Path” dialog box.

e. In the dialog box, click the New Search Path button , browse to the ***<platform1>*\soc** directory, and click **OK**.

f. Click **OK** to add the path to the Project Properties and close the “Verilog Include Search Path” dialog box.

g. Click **OK** to return to the Diamond main window.

Connect the Microprocessor to the FPGA Pins

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

- Manually create the pin constraints and import them into Diamond.
- Import a preconfigured preference file into Diamond.

For this tutorial, you will import a preconfigured pin preference file into Diamond. Create a preconfigured pin preference file named “untitled.lpf” as follows:

1. Copy and paste the contents of Figure 35 on page 48 into a text editor.
2. Name the file **untitled.lpf,** and save the file in the following directory:
   - For Windows: <*Diamond_install_path*>\examples\lm32_tutor
   - For Linux: ~/LatticeMico/lm32_tutor

To import this preconfigured pin preference file:

1. In Diamond, select the File List tab and double-click **Strategy1**.
2. In the Strategies dialog box, select **Translate Design** in the left pane.
3. In the right pane, double-click the cell in the Value column for **Consistent Bus Name Conversion**.
4. Choose **Lattice** from the drop-down menu, and click **OK.**
5. In Diamond, choose **File > Add > Existing File**.
6. In the Add Existing File dialog box, select **Constraint Files (*.lpf)** from the Files of type menu.
7. Right click on untitled.lpf and select “Set as Active Preference File”.

Figure 35: Preconfigured Pin Preference File

```
BLOCK RESETPATHS ;
BLOCK ASYNCPATHS ;
SYSCONFIG MCCLK_FREQ=26 WAKE_UP=21 CONFIG_SECURE=OFF STRTUP=CCLK
WAKE_ON_LOCK=OFF ;

LOCATE COMP "reset_n" SITE "J1" ;
#-----------------------------------------------------------------------
---------------------
# SPI Flash assumes the memory component instance name is SPIFlash
#-----------------------------------------------------------------------
---------------------
LOCATE COMP "SPIFlashSCK" SITE "B22" ;
LOCATE COMP "SPIFlashCEJ" SITE "D21" ;
LOCATE COMP "SPIFlashSI" SITE "H20" ;
LOCATE COMP "SPIFlashSO" SITE "H19" ;
LOCATE COMP "SPIFlashWPJ" SITE "G17" ;

#-----------------------------------------------------------------------
---------------------
# LEDs assumes the component instance name is LED & has
# a width of 8
#-----------------------------------------------------------------------
---------------------
LOCATE COMP "LEDPIO_OUT[0]" SITE "U19" ;
LOCATE COMP "LEDPIO_OUT[1]" SITE "U18" ;
LOCATE COMP "LEDPIO_OUT[2]" SITE "AA21" ;
LOCATE COMP "LEDPIO_OUT[3]" SITE "Y20" ;
LOCATE COMP "LEDPIO_OUT[4]" SITE "W19" ;
LOCATE COMP "LEDPIO_OUT[5]" SITE "V19" ;
LOCATE COMP "LEDPIO_OUT[6]" SITE "AA20" ;
LOCATE COMP "LEDPIO_OUT[7]" SITE "AB20" ;
#-----------------------------------------------------------------------
---------------------
# 7 segments assumes the component instance name is gpio_7Segs & has
# a width of 8
#-----------------------------------------------------------------------
---------------------
LOCATE COMP "gpio_7SegsPIO_OUT[0]" SITE "V6" ;
LOCATE COMP "gpio_7SegsPIO_OUT[1]" SITE "U7" ;
LOCATE COMP "gpio_7SegsPIO_OUT[2]" SITE "Y6" ;
LOCATE COMP "gpio_7SegsPIO_OUT[3]" SITE "AA6" ;
LOCATE COMP "gpio_7SegsPIO_OUT[4]" SITE "U8" ;
LOCATE COMP "gpio_7SegsPIO_OUT[5]" SITE "T8" ;
LOCATE COMP "gpio_7SegsPIO_OUT[6]" SITE "AA5" ;
LOCATE COMP "gpio_7SegsPIO_OUT[7]" SITE "AB4" ;
```

Perform Functional Simulation

You can optionally simulate the functionality of the output top-level platform1.v or platform1_vhd.vhd module by using a simulator such as Active-HDL in Diamond. See the Active-HDL online Help in Diamond for more information on this procedure.

- For Verilog simulation, you use platform1.v and all the Verilog files for each attached component.
- For mixed Verilog/VHDL simulation, you use platform1_vhd.vhd, platform1.v, and all the Verilog files for each attached component. You must use a mixed-language simulator such as ModelSim® SE or Aldec Active-HDL.

See Also "Performing HDL Functional Simulation of LatticeMico Platforms" in the *LatticeMico32 Software Developer User Guide*.

Perform Timing Simulation

You can optionally validate the timing of your design by performing timing simulation. Because timing simulation is a complex topic, it is not addressed in this tutorial. For information on timing simulation, see the *Achieving Timing Closure in FPGA Designs Tutorial*, the "Design Verification" topic in the Diamond online Help, or the "Strategies for Timing Closure" chapter of the *FPGA Design Guide*.

The timing simulation process automatically builds a database and maps, places, and routes the design.

Generate the Bitstream

Now you will generate a bitstream to download the microprocessor platform to the FPGA. If you did not perform timing simulation, the bitstream generation process will automatically synthesize, map, place, and route the design before it generates the bitstream.

To generate a bitstream (.bit) file:

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

Diamond now generates a bitstream data file, platform1.bit, that is ready to be downloaded into the device. This process takes several minutes.

Task 5: Download the Hardware Bitstream to the FPGA

The bitstream file generated in the previous section contains all the information required to program the LatticeECP3 FPGA. Lattice Semiconductor provides the Diamond Programmer tool that sends the programming bitstream to the FPGA over a parallel port or USB port communications link. Now you will use Programmer to download the hardware bitstream that you generated in the previous section to the FPGA on the board. For instructions on connecting the USB cable to the board, refer to the *LatticeECP3 Versa Evaluation Board User's Guide*.

To download the bitstream to the FPGA on the board:

1. Remove any Lattice USB Programming cables from your system.
2. Connect the power supply to the development board.
3. Connect a USB cable from your computer to the LatticeECP3 Versa Evaluation Board. The USB cable must be connected to the USB target connector adjacent to the keypad. Give the computer a few seconds to detect the USB device on the LatticeECP3 Versa Evaluation Board before moving to step 4.

   Note

   A USB cable is included with the board.

4. In Diamond, choose **Tools > Programmer**.
5. 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 in Diamond.

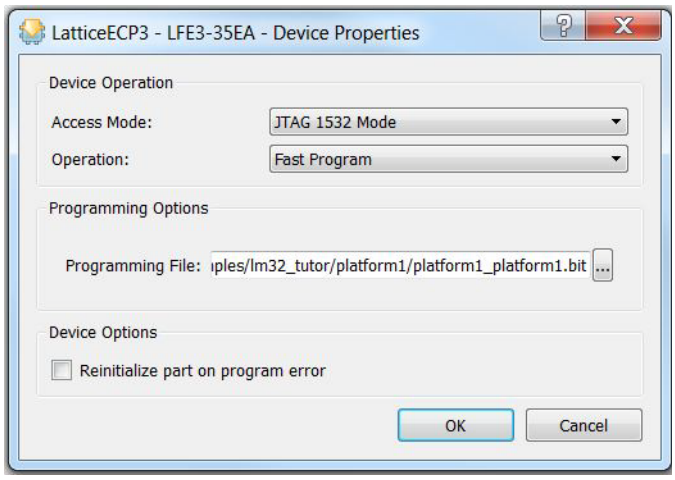
Note

If the Programmer output window displays "Cannot identify detected device on row 1. Please manually select correct device," choose **LFE3-35EA** from the Device column drop down menu.

6. In the Cable Settings dialog box on the right side of the Programmer window, do the following:
   a. In the Cable box, select **HW-USBN-2B (FTDI)**.
   b. In the Port box, choose the only setting available in the drop-down menu, **FTUSB-0**.
7. Double-click the Operation column to display the Device Properties dialog box, as shown in Figure 36, and choose the following settings:
   - For Access Mode, choose **JTAG 1532 Mode** from the pull-down menu.
   - For Operation, choose **Fast Program** from the pull-down menu,

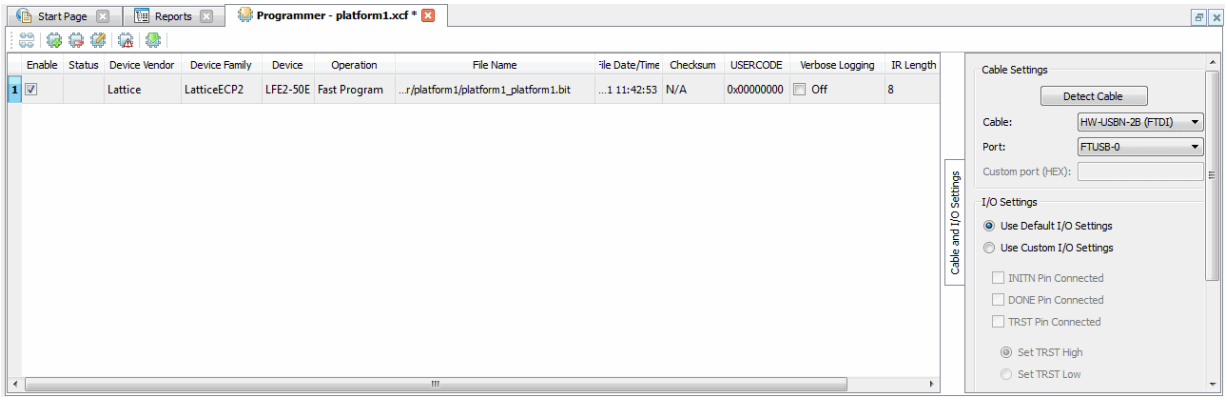
8. Double-click the File Name column. Click ... to display the Open File dialog box, and browse to the platform1_platform1.bit file in the following directory:
   - For Windows, *<Diamond_install_path>*\examples\lm32_tutor\platform1_platform1.bit
   - For Linux, ~/LatticeMico/lm32_tutor/platform1_platform1.bit
9. Click **Open**.

Figure 36: Device Properties Dialog Box.



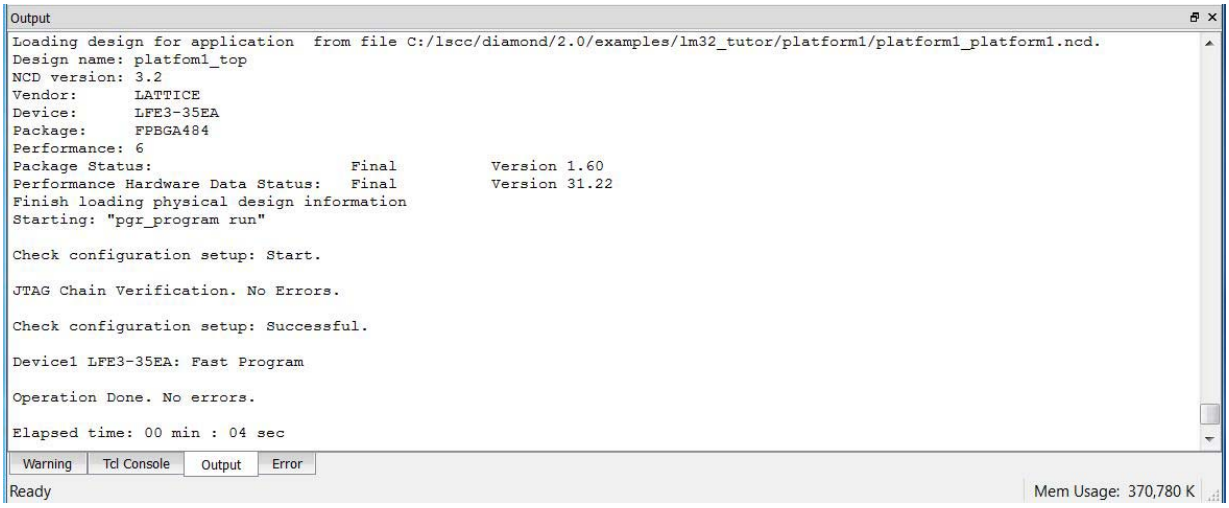
10. Click **OK**.
11. The Programmer view should look as shown in Figure 37.

Figure 37: Diamond Programmer



12. Click the Program button on the Programmer toolbar to initiate the download.
13. Check the Programmer output console to see if the download passed, as shown in Figure 38. If the programming process succeeded, you will see a green-shaded PASS in the Programmer Status column.

Figure 38: Programmer Output Console



14. At the end of this process, the FPGA is loaded with the Microprocessor hardware configuration.
15. In Diamond, choose **File > Save platform1.xcf**.
16. Exit Diamond by choosing **File > Exit.**

Task 6: Debug and Execute the Software Application Code on the Development Board

In this task, you will use the debugger to download the executable file containing the software application code to the LatticeECP3 Versa Evaluation Board. This enables the LatticeMico32 microprocessor, which is part of the FPGA bitstream you downloaded in Task 5, to execute the application code.

This task assumes that you have successfully downloaded the platform FPGA bitstream to the development board in “Task 5: Download the Hardware Bitstream to the FPGA” on page 50.

If you encounter any problems with the debug session, refer to "Debug Session Troubleshooting" in the *Lattice Software Project Environment* online Help. This troubleshooting topic describes the most common problems encountered in launching a debug session and the reasons the debugger sometimes fails to operate.

Software Application Code Execution Flow

The FPGA is now configured with the LatticeMico32 Development microprocessor platform. The order in which the LatticeMico32 microprocessor executes the software application code images is as follows:

1. The LatticeMico32 microprocessor starts execution at the address contained in its exceptions base address (EBA).

   This is the address you specified when you added the LatticeMico32 microprocessor core in Task 2.

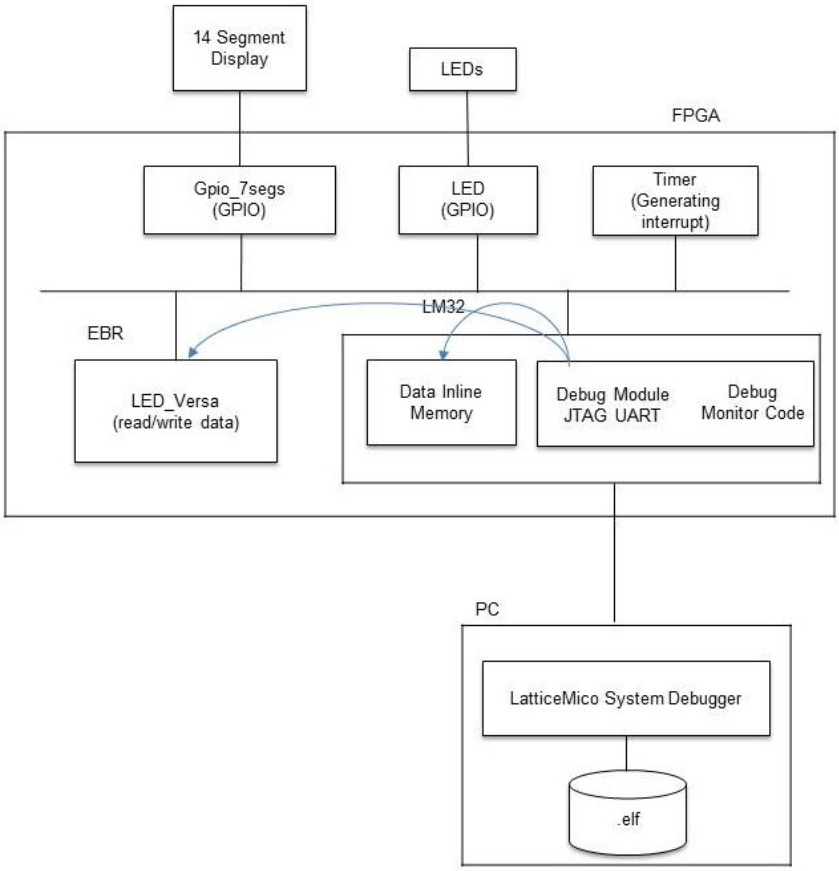
2. When you start the LatticeMico System debugger, it communicates with the microprocessor over the microprocessor's debug module.

   The debug module is a collection of files inside the lm32_top\rtl\verilog directory, as shown in Figure 26 on page 35. The debug module, in turn, generates a debug exception that causes the microprocessor to execute the debug monitor code. The LatticeMico32 microprocessor, in order to respond to the debug exception, must be running valid opcodes and must not be stuck waiting for a bus cycle to complete. Upon successful execution of the debug exception, the debug monitor code then communicates with the LatticeMico System debugger running on the host computer.

3. At this point, the debugger has control over the microprocessor and can access the platform's memory through the debug module or microprocessor to download the application to the selected memories.

4. After it has downloaded the application to be debugged to the target memory or memories, the debugger sets the microprocessor's program counter to start executing the downloaded code.

Figure 39: Software Application Code Execution Flow



Debug the Software Application Code on the Board

Now that you have a LatticeMico32 platform loaded into the LatticeECP3 Versa Evaluation Board and a compiled and linked C program, you can begin working with the LatticeMico source code debugger.

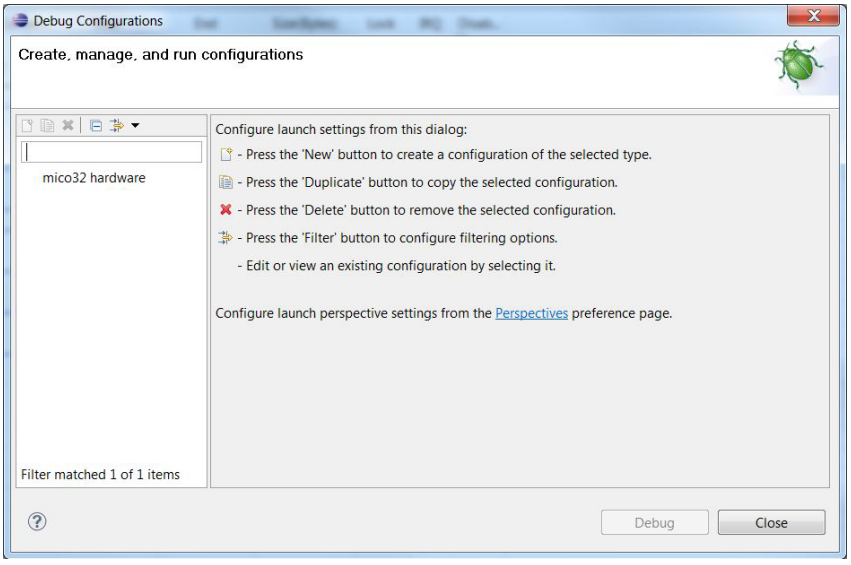
The source code debugger allows you to download the fully resolved ELF file created by the linker into the memories specified by the auto-generated linker script. The debugger enables you to set breakpoints, control the program flow, and inspect variables, registers, and memory. It enables you to validate that your program is functioning correctly, and it enables you to find any problems that exist in the applications source code.

To debug the software application code on the board:

1. In the C/C++ SPE perspective, click **LED_Versa** in the C/C++ Projects view (left-hand pane).
2. Right click the project and choose **Debug As-> Debug Configurations.**

   The Debug dialog box opens, as shown in Figure 40.

Figure 40: Debug Dialog Box

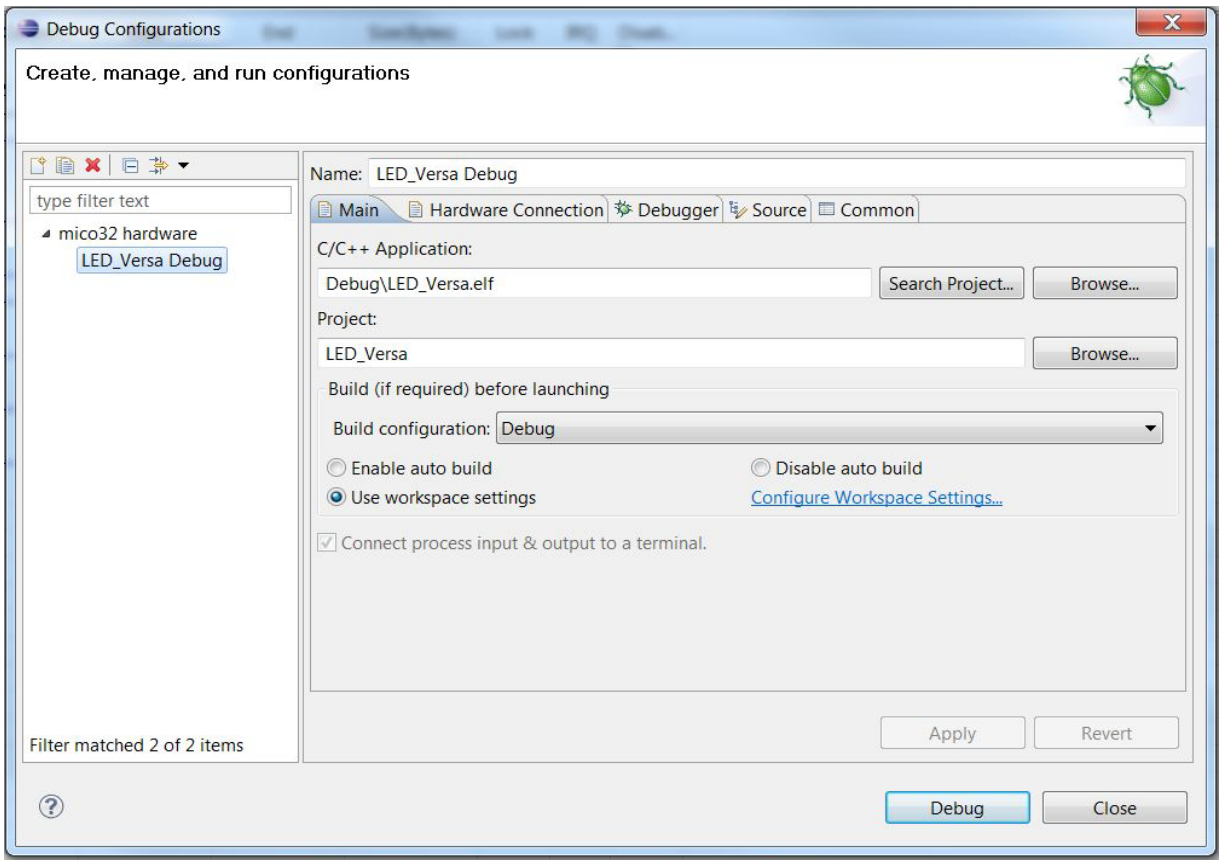


3. Select **mico32hardware**, and then click the **New launch configuration button** on the toolbar.

   If you are connecting to the evaluation board for the first time, the Progress Information message box appears.

   The appearance of the Debug dialog box changes again, as shown in Figure 41.

Figure 41: Debug Dialog Box with Tabs



In this dialog box, you specify the project or executable to debug. Since you selected the project before selecting **Run > Debug Configurations**, the boxes are filled in by Eclipse. If these boxes are not populated, follow these instructions to configure the items in this dialog box:

a. Use the **Browse** button to select the Eclipse project.

   Clicking Browse activates a dialog box that lists the available projects created or imported in Eclipse.

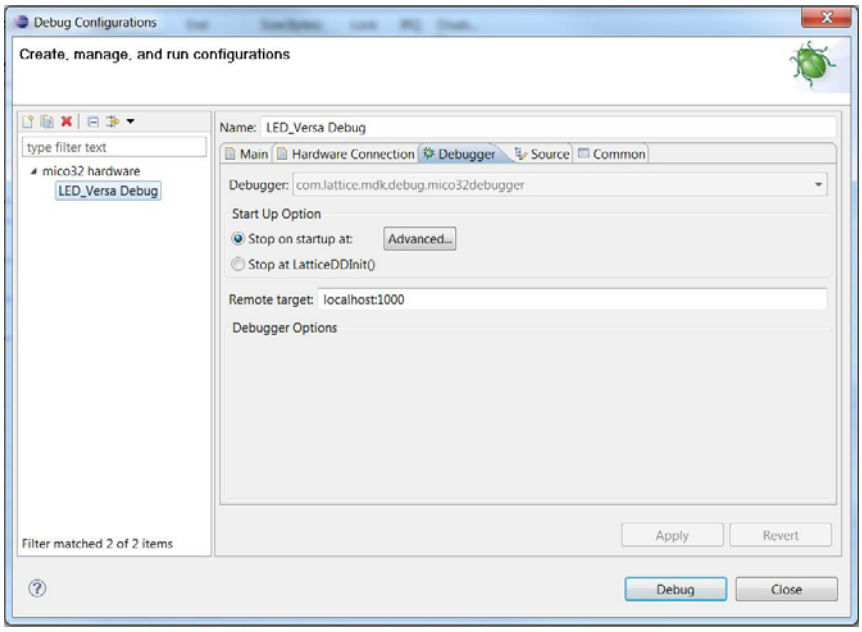
b. Select **LED_Versa**.

c. Click the **Search Project** button to select the executable (.elf) file that you want to debug.

A project may have multiple executables. Clicking the Search Project button activates a dialog box that lists the executables built for the project. If you want to use an executable not built within C/C++ SPE, click the **Browse** button to activate a file selection dialog box in which to select the appropriate .elf-format executable file.

4. Click the **Debugger** tab of the Debug dialog box, as shown in Figure 42.

Figure 42: Debugger Tab of the Debug Dialog Box



The Debugger tab features the following Debugger settings:

- The Start Up Option section enables you to choose where you want your initial breakpoint. For a debug launch, the Debugger downloads the code and sets an initial breakpoint to enable debugging. You can place your breakpoint either at the start of your main program or at the start of the Device Driver initialization routine generated by the C/C++ SPE managed build process. The default behavior is to set the initial breakpoint at the first executable source line inside the main() function.
- Remote target option, which provides the address for the LatticeMico debug proxy program that will be launched on your computer. This proxy program allows C/C++ SPE to debug the program by using the GNU GDB program and provides a communication channel to the microprocessor over a JTAG connection. Refer to the *LatticeMico32 Software Developer User Guide* for more details on the debugging setup.

If you attempt to change settings, the Apply button might become available. In this case, click the **Apply** button to save your settings.

5. Click the **Debug** button located on the lower right side of the dialog box.

Note

If you encounter any problems with the debug session, refer to "Debug Session Troubleshooting" in the *Lattice Software Project Environment* online Help. This troubleshooting topic describes the most common problems encountered in launching a debug session and the reasons the debugger sometimes fails to operate.

When you click the Debug button, the dialog box closes, and C/C++ SPE attempts to interface to the debug monitor in the LatticeMico32 platform. Once it has established a connection to the debug monitor it downloads the LatticeMico executable code to the memories specified by the linker script. After it has successfully done this, the Confirm Perspective Switch prompt box containing the following message appears:

```
This kind of launch is configured to open the Debug
perspective when it suspends. Do you want to open this
perspective now?
```

6. Select the **Remember my decision** box, and click the **Yes** button. Click **Yes** in the prompt box.

Note

If you did not previously download the bitstream, a message box with the following error message may appear:

```
Check that the target FPGA contains an LM32 CPU with
DEBUG_ENABLED equal to TRUE and that the FPGA has configured
successfully.
```

Return to “Task 5: Download the Hardware Bitstream to the FPGA” on page 50, and download the bitstream before proceeding.

Note

Selecting **Run > Debug Configurations** on a computer running the Windows operating system might activate the Windows firewall. The Windows Security Alert dialog box shown in Figure 43 might appear.

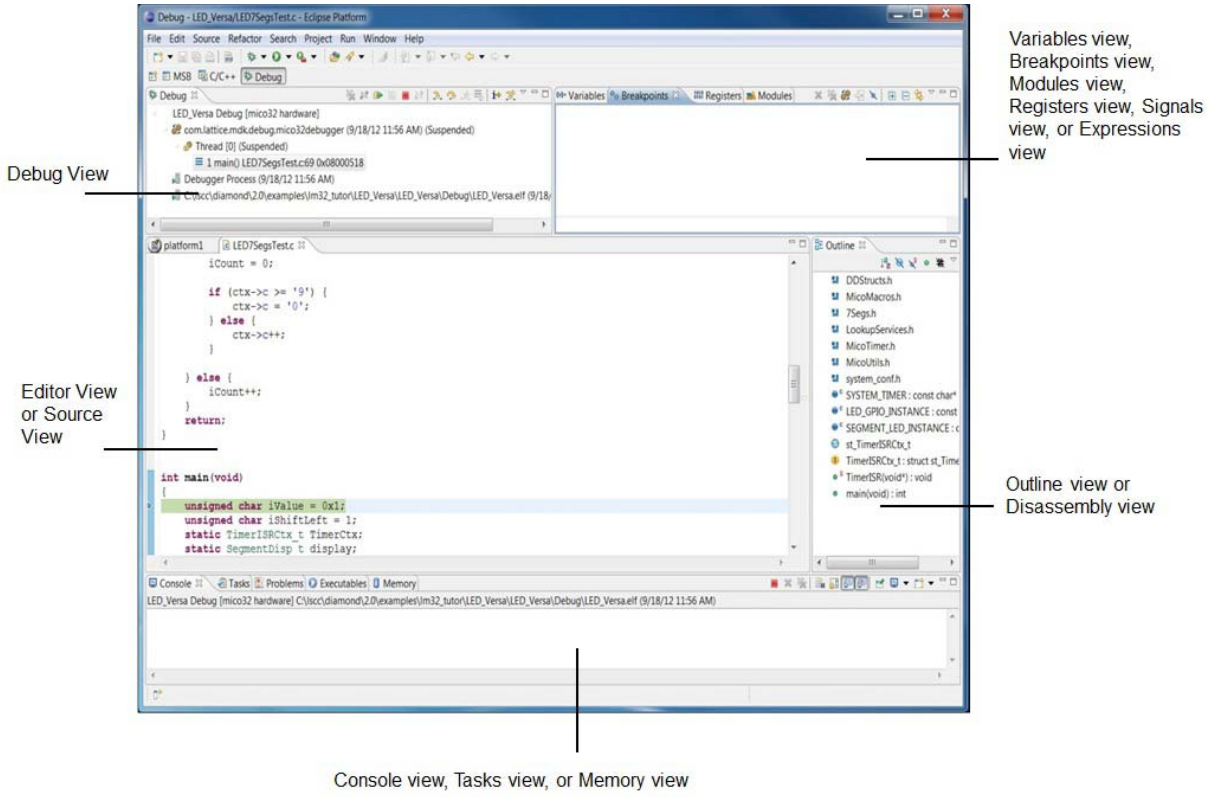
Figure 43: Windows Security Alert Dialog Box

Click **unblock** to continue debugging.

TCP2JTAGVC is the application that provides the communication channel between the LatticeMico32 microprocessor debug module and lm32-elf-gdb (GDB modified for the LatticeMico32 microprocessor).

C/C++ SPE now switches to the Debug perspective, shown in Figure 44.

Figure 44: Debug Perspective



The Debug perspective consists of many views, some of which may not be visible:

- Debug view, which displays the function calls made so far. It also contains application and process information.
- 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, which shows the source code in assembly language with offsets. It shows the instructions that reside at each address.

To select views that are not visible for this perspective, click **Window > Show View** and choose the appropriate view.

7. If it is not already displayed, expand the LED_Versa in the top left of Debug view. It should resemble the illustration in Figure 45.

Figure 45: Expanded Debug View

This shows the processes that are running on the host PC.

Insert Breakpoints

The information in the expanded Debug view under com.lattice.mdk.debug.mico32debugger contains information about the executable downloaded to the FPGA and executed by LatticeMico. It shows that the execution is suspended because of a breakpoint at a line within the LED7SegsTest.c source file.

1. In the Debug view, click on the statement containing the line **main()**.

   This step activates the file in the Source view, located below the Debug view. A line with green highlighting shows the line at which the LatticeMico32 microprocessor has been suspended because of a breakpoint. The breakpoint is at the beginning of your main program, as configured for this debug launch.

   You will now insert a breakpoint to check the software and platform functionality.

2. In the LED7SegsTest.c file displayed in the Source view, click on the line beginning with “MicoGPIOCtx_t,” as indicated in Figure 46.

Figure 46: Breakpoint Line

3. Insert a breakpoint by double-clicking in the left margin, aligned to the line shown in Figure 46. Alternatively, you can select **Run > Toggle Line Breakpoint**.

   As shown in Figure 47, LED7SegsTest.c should now appear in the Source view with a blue bubble and a check mark in the margin aligned to the line of interest. If the Breakpoint view is open, it be should updated to show this breakpoint.

Figure 47: Inserted Breakpoint

Execute the Software Application Code

Now you can resume executing the software application code on the board.

1. In the Debug view, click the green arrow to the right of the “Debug” tab title. Alternatively, you can choose **Run > Resume**.

   The Debugger now issues a “continue” command to the LatticeMico32 microprocessor, which executes the code until it reaches the breakpoint that you inserted previously.

2. Step over the C source line by clicking the icon in the same line as the Debug tab title. Alternatively, you can choose **Run > Step Over** or press the **F6** key.

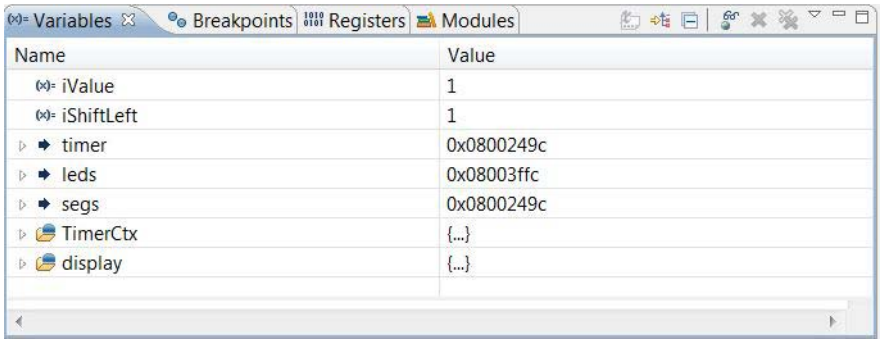
The Debugger causes the microprocessor to execute the source line at which the breakpoint was inserted.

At this point, the Variables view is updated, as shown in Figure 48.

Note

If the Variables view is not visible, choose **Window > Show View > Variables** to make it visible. If the Variables view is inactive—that is, the tab is shown in gray tones—click on the Variables tab to make it active.

Figure 48: Updated Variables View



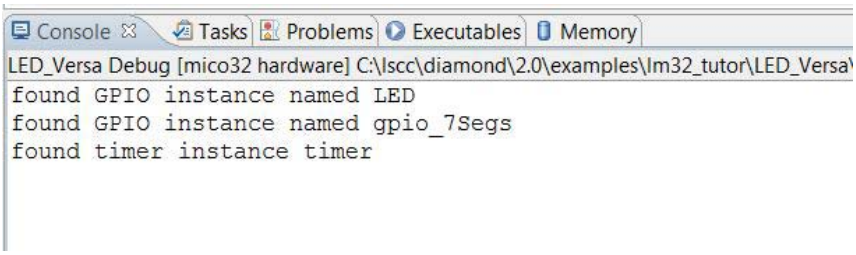
The value of the “leds” variable might be different from that shown in Figure 48. However, if the value of the “leds” variable shown in Figure 48 is 0x00000000 (or 0) for your view, the platform most likely does not have a GPIO named LED in the platform. Repeat the tutorial, following the procedures exactly.

3. In the Debug view, click the green arrow next to the tab title, or choose **Run > Resume**.

   The Debugger issues a “continue” command to the LatticeMico32 microprocessor, which causes the microprocessor to continue execution of the downloaded code.

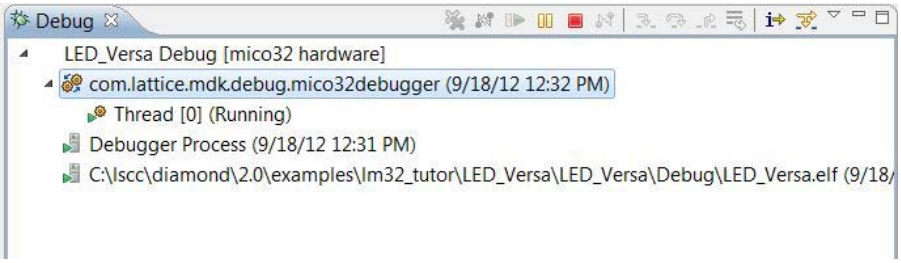
   The Console view in the bottom of the C/C++ SPE window should display the text line shown in Figure 49 on page 61. This text is output by the LED_Versa application running on LatticeMico, which uses the JTAG connection to the Debugger for standard input/output communication to the C/C++ SPE console.

Figure 49: Console Output



4. Observe the LEDs on the LatticeMico32 development board to confirm a back-and-forth scrolling LED pattern, which is controlled by the code executed by LatticeMico.
5. Expand the Debug view to show the active processes, shown in Figure 50.

Figure 50: Running Processes



6. Click the line containing the text **Thread[0] (Running)** to activate the following two buttons:
   - A button with two orange bars, , located towards the center of the debug view title bar, which pauses execution. It inserts an asynchronous breakpoint similar to a pre-set line breakpoint.
   - A button with a red square, , which terminates the running application on LatticeMico. The Debugger no longer provides access to the code being debugged. Use **Run > Debug Configurations** to restart the debugging session.
7. Click the red-square button to terminate execution of the LED_Versa application on LatticeMico.
8. Click the C/C++ button on the top left of the Debug perspective window to return to the C/C++ perspective. Alternatively, you can select **Windows > Open Perspective > C/C++** to return to the C/C++ perspective.

Modify and Re-execute the Software Application Code

The LED_Versa.c application contained some printf statements for test purposes. The platform is configured so that these printf statements communicate through the microprocessor's debug module to the debugger running on the host machine for outputting information to the C/C++ SPE console. If the debugger is absent, the printf statements cause the debug module to wait indefinitely for a client to communicate with. Therefore, now that the code is validated and needs to be deployed, it must be devoid of printf statements.

1. Delete the four printf statements from the code to make it similar to the example shown in Figure 51.

Figure 51: Modified LED_Versa Code

```
int main(void)
{
    unsigned char iValue = 0x1;
    unsigned char iShiftLeft = 1;
    static TimerISRCtx_t TimerCtx;
    static SegmentDisp_t display;

    /* 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) {
        return(-1);
    }

    /* see if we found 16-segment LED GPIO */
    if (segs == 0) {
        return(-1);
    }

    if (segs->wb_data_size != 32) {
        return(-1);
    }

    /* see if we found our timer */
    if (timer == 0) {
        return(-1);
    }
```

2. Choose **File > Save** to save the modified file.

   Before you rebuild the project, it is important that you terminate any prior debug session. If the Debugger is still paused or running, the Build Project command will fail when the linker tries to overwrite the platform1.elf file.

3. To rebuild the modified code, select LED_Versa and choose **Project > Build Project**.

4. Return to the Debug perspective.

5. To download, debug, and execute the modified code, do the following:

   a. Click **Run > Debug Configurations**, and then click **Debug** in the Debug window.

   b. Click the green arrow next to the tab title.

   c. Step over the C source line by clicking the icon in the Debug view or choose **Run > Step Over** or press the **F6** key.

   d. Click the green arrow again.

   This code is now ready for stand-alone deployment in the parallel flash memory.

   You have now completed the task of downloading and executing the software application code on the LatticeECP3 Versa Evaluation Board.

6. Verify that the LED_Versa program is functioning by noting the sweeping LED pattern and counting from 0-9 on the 14-segment display.
7. Click the **Terminate** ( ) button to stop the demonstration program and unload the debug session. Failing to unload the debug session interferes with programming the parallel flash memory, a process that is described in the next session.

Task 7: Setting up the Options to Deploy the Software Code to the SPI Flash Memory

As part of Task 6, you debugged and executed the LEDTest software application code from Lattice Software Project Environment. That is, you used the Lattice Software Project Environment to load the LEDTest software code onto volatile memory on the development board and then debug/execute it.

In this task, you will prepare the LED_Versa software for deployment to SPI flash memory and merge the production microprocessor bitstream and LED_Versa software into one SPI Flash Image, which will be loaded into non-volatile memory (i.e. SPI Flash memory).

Refer to the *LatticeMico32 Software Developer User Guide* for details on deployment strategies and user flow.

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.

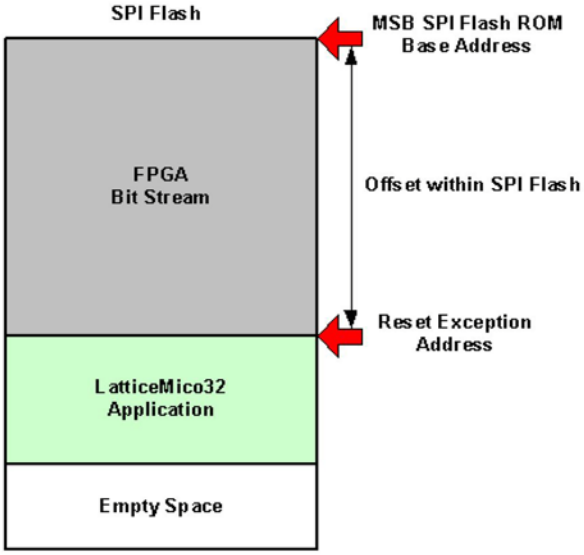
The main advantage of SPI flash deployment is that it allows the FPGA bitstream (or portions of it) and the microprocessor bitstream to co-exist in a single SPI flash device. However, this is possible only if the FPGA user logic can access the very same SPI flash device that was used for the FPGA configuration. SPI flash deployment may impose FPGA requirements, board layout requirements, or both, which must be considered before you design the hardware. The LatticeECP3 Versa board can be used for SPI Flash deployment, since its user logic can also access the configuration SPI flash in addition to the configuration logic.

Selecting the Appropriate LatticeMico EBA Value

Figure 52 shows a sample layout in the SPI flash memory.

In Figure 52, the first data portion is the FPGA bitstream that is used for configuring the FPGA. The second data portion is the LatticeMico application

Figure 52: Sample Layout in SPI Flash Memory



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) 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 LatticeMico32 boot application will reside. The offset depends on the FPGA bitstream size.

Offset Alignment in the SPI Flasht 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 LatticeMico32 instructions are 32 bits, or 4 bytes.

Prepare the LatticeMico32 to Execute from the SPI Flash on Removal of Reset

1. Return to the MSB Perspective.
2. Modify the LatticeMico32 Exception Vector base address to point to the SPI flash memories base address plus the offset where the software code resides in the SPI Flash (0x04120000).

   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 LatticeMico32 boot application will reside. The offset depends on the FPGA bitstream size.
3. Generate the platform.

Prepare LED_Versa for Flash Deployment

The first step is to compile the LED_Versa ELF. You cannot use the LED_Versa ELF created in Task 3, since it was built for both deployment and execution from SRAM and data inline memory. What you need is an LED_Versa ELF that is built for deployment into parallel flash memory and is built for execution from SRAM and data inline memory. Therefore, before deploying LED_Versa application to parallel flash memory, you must recompile LED_Versa ELF to change the deployment location to parallel flash memory. As part of this recompilation, the Lattice Software Project Environment (C/C++ SPE) will instruct GCC to build a code relocator into the LED_Versa ELF. This code relocator is essential, because it will be responsible for copying the LED_Versa program and read/write data memory sections to EBR (On-chip memory) and copying the read-only sections to data inline memory from SPI flash memory for execution of the LED_Versa software upon board reset.

Note

You can no longer use the new LEDTest ELF for debugging and execution purposes from Lattice C/C++ SPE, since it has been prepared for parallel flash deployment. The LEDTest ELF must be recreated, as shown in Task 3, for this purpose.

Note

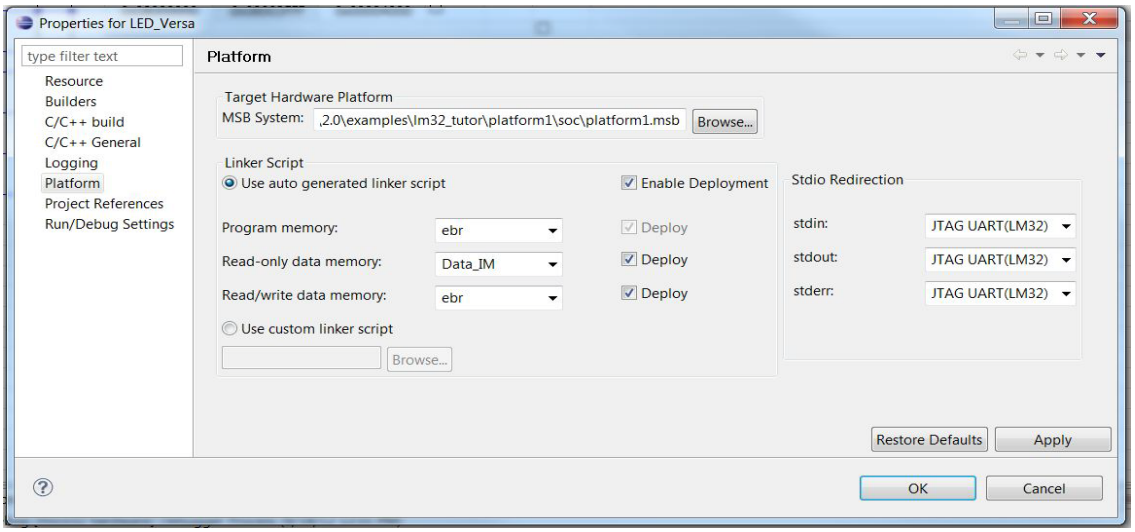
For deployment, you must not use a JTAG UART. If the code uses standard C file operations, such as printf, scanf, or fopen, your deployed code will not work if it uses a JTAG UART as a standard I/O device or for file operations. You can use the RS-232 UART for standard I/O operations, for that you need to include the UART platform into the MSB platform. As this tutorial doesn't uses the standard C file operations, so no need to worry about setting Stdio Redirection.

To change the properties and rebuild the LEDTest project:

1. In the C/C++ pers pective, select LEDTest and choose Project > Properties .
2. In the Properties dialog box, select Platform.
3. In the Linker Script section, do the following:
   a. Select Enable Deployment.
   b. For Program memory, choose ebr
   c. For Read-only data memory, choose Data_IM.
   d. For Read/write data memory, choose ebr.

   The Platform Properties dialog box should look like the example shown in Figure 53.
4. Click **OK.**

Figure 53: Platform Properties



5. In the MSB perspective, verify that the LatticeMico32 Exception Handler address is set to 0x04120000. If it is not, update the Exception Handler address and regenerate the platform.
6. In the C/C++ perspective, select LEDTest and choose **Project > Build Project**.

Generating LatticeMico32 Bootable Application Binary

Once the LatticeMico32 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.

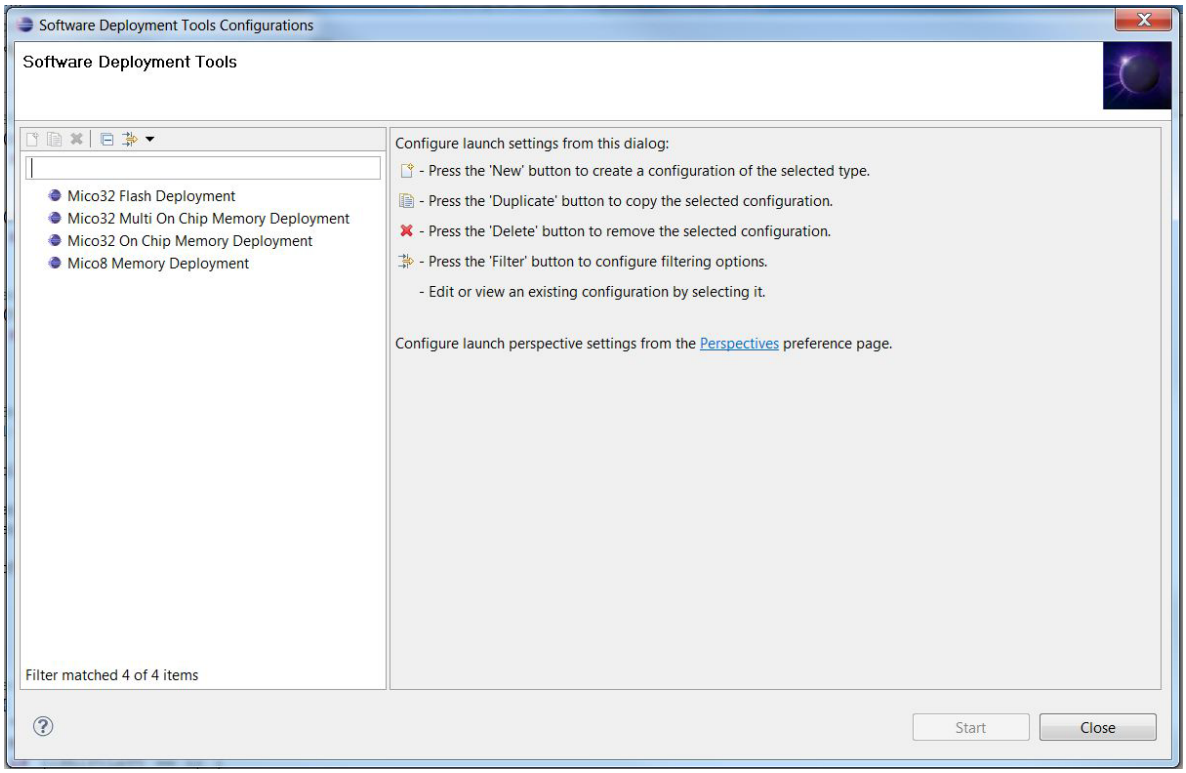
The LatticeMico C/C++ SPE perspective provides a graphical user interface for this purpose.

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 54 on page 68.
2. Select F**lash Deployment** from the list of configurations, and click **New**.

   The main tab of the Software Deployment Tools Dialog now appears, as Shown in Figure 55 on page 69.

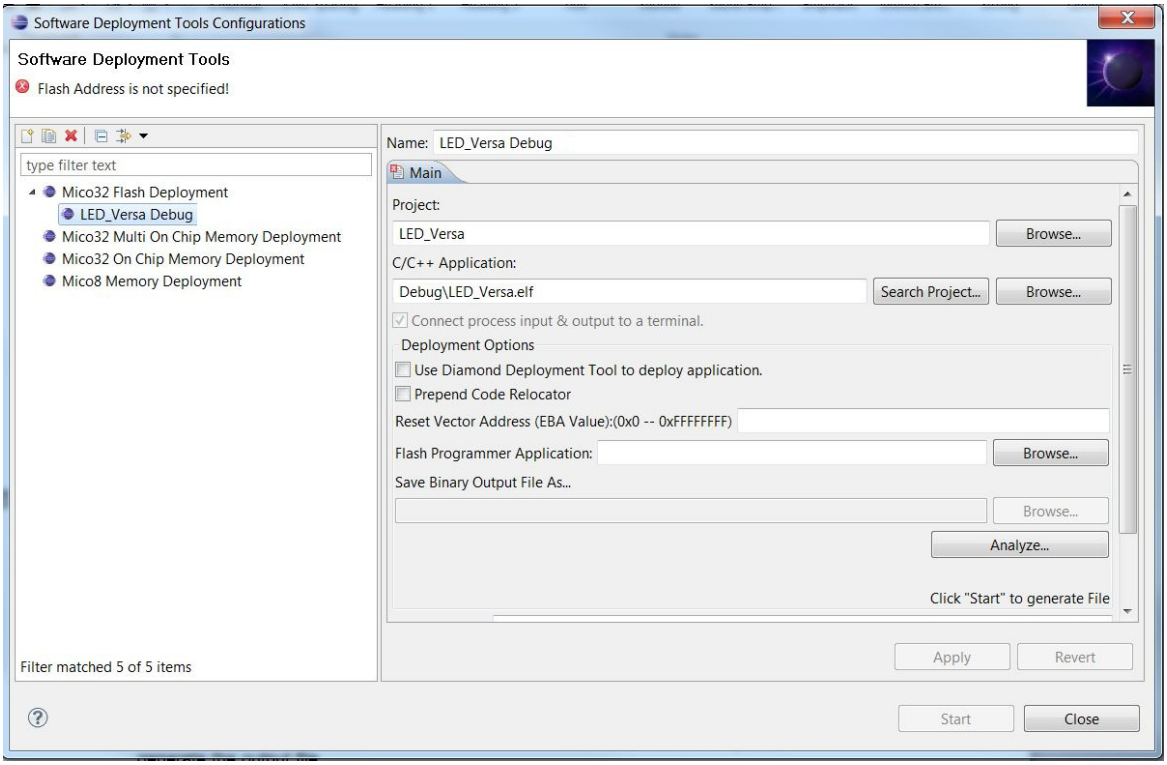
Figure 54: Software Deployment Tools Dialog Box



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 65.
- **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.

Figure 55: Main Tab of the Software Deployment Tools Dialog Box



- **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.

To prepare LED_Versa for flash deployment:

1. Click the **Browse** button next to the Project box.

   The Project Selection dialog box appears, as shown in Figure 56 on page 70.

2. Select **LED_Versa** and click **OK** to select the project containing the executable that needs to be programmed to flash.

3. Click **Search Project** next to the C/C++ Application text box.

   The Program Selection dialog box appears, as shown in Figure 57 on page 70. It contains the list of executables for the selected project, LED_Versa.

4. From the Binaries list, select **LED_Versa.elf** and click **OK**.

5. Select **Use Diamond Deployment Tool to deploy Application** option.

6. In the Reset Vector Address box, under Deployment Options, enter **0x04120000**, 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 LatticeMico32 boot application (LED_Versa) will reside, this value should be equal to the LatticeMico32 EBA value.

Figure 56: Project Selection Dialog Box

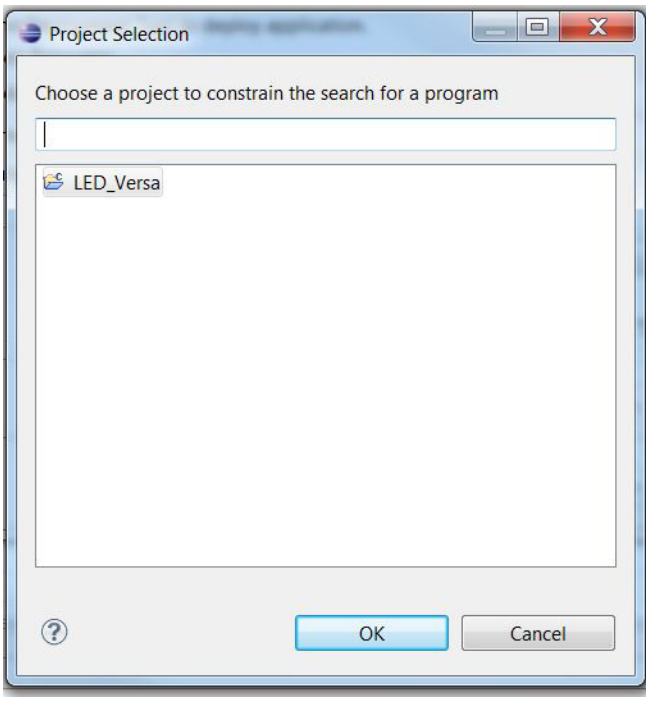
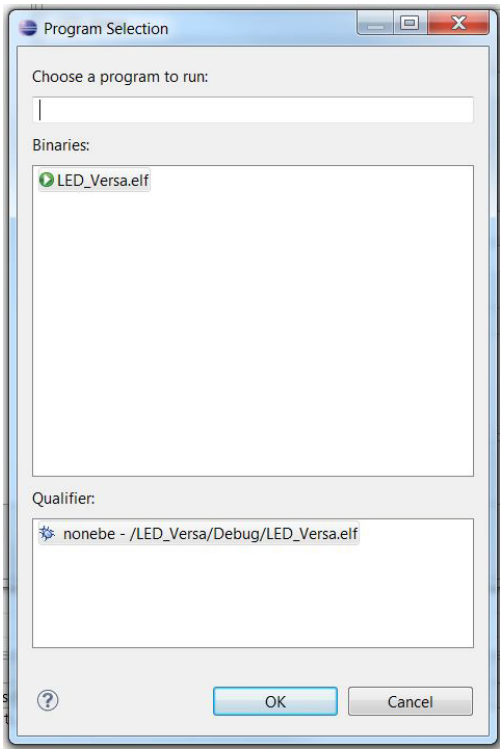


Figure 57: Program Selection Dialog Box



7. Save Binary Output File As - Click the **Browse** button and browse to the following location and save it as **LEDtest.bit**.

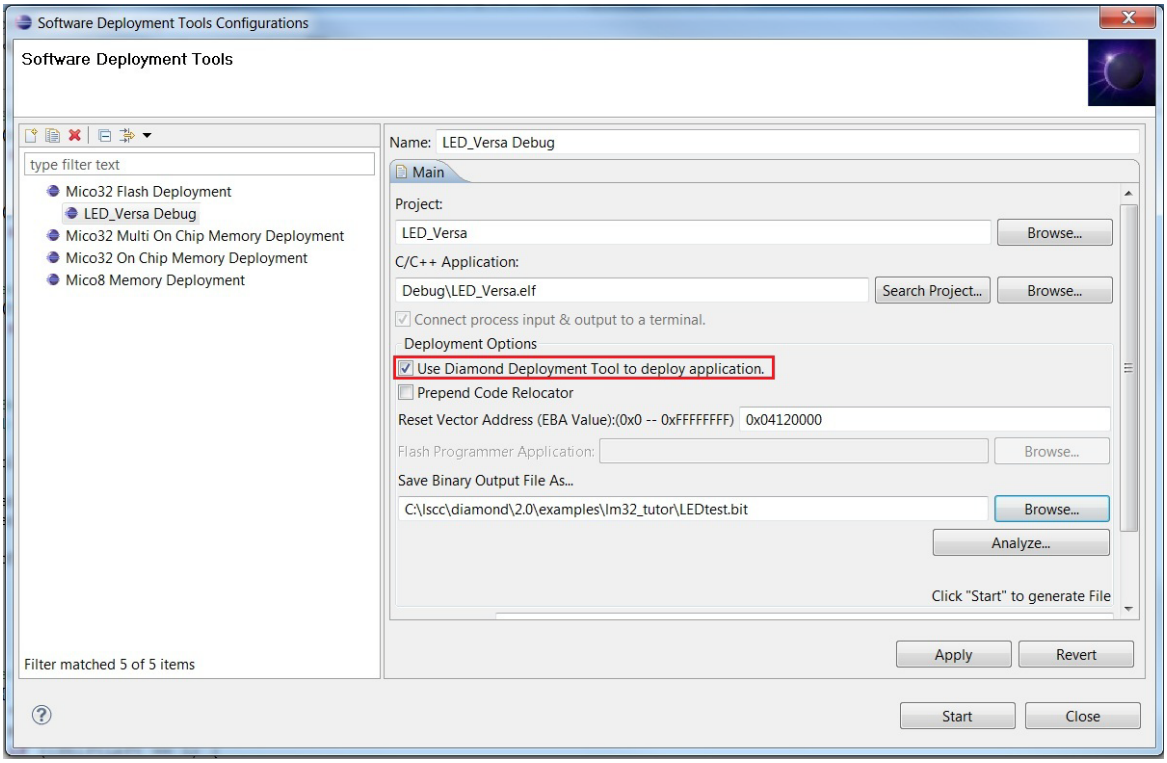
- For Windows: <*Diamond_install_path*>\examples\lm32_tutor\LEDtest.bit
- For Linux: ~/LatticeMico32/lm32_tutor/LEDtest.bit

8. Click **Analyze** to confirm that the selected LED_Versa sections are being deployed to SPI flash memory.

   In this tutorial, you deploy all LED_Versa sections to SPI flash memory. Clicking on **Analyze** should show that the following sections are deployed: .boot, .text, .rodata, .data, .bss.

   The Software Deployment Tools Configuration dialog box should look like the example shown in Figure 58.

Figure 58: Main Tab of the Software Deployment Tools Configuration Dialog Box with Use Diamond Deployment Tool to Deploy Application Enabled



9. Click **Apply** to save the configuration if you want to reprogram the application with these settings.

10. Click **Start.**

    As the programmer application executes successfully, you see a console display similar to the one shown in Figure 59.

Figure 59: Console Display

```
Problems  Tasks  Console  Properties
<terminated> LED_Versa Debug (1) [Mico32 Flash Deployment] Spi Launch Process
Type: 2, Machine: 1638, Version: 1, Entry: 0x8000000, PHoff: 0x34/32/2, SHoff: 0x17b90/40/19
Program header: type: 1, off: 0x1000, va: 0x8000000, pa: 0x8000000, len: 12984/13636, flags: 7
Program header: type: 1, off: 0x5000, va: 0xc000000, pa: 0xc000000, len: 704/704, flags: 4
Entry point: 0x8000000, address range: 0x8000000-0xc000000
lm32-elf-gcc -c BinCopier.S
lm32-elf-ld BinCopier.o --section-start .boot=0x04120000 -o BinCopier.elf
lm32-elf-ld: warning: cannot find entry symbol _start; defaulting to 00000000
 flash binary (12984 bytes) prepared for loading at 0x04120000
```

Generating Production Microprocessor Bitstream

You might have observed that the Exception Handler address was changed to 0x04120000 and the platform Generate function was performed, but the Diamond Bitstream Generation process was never run. Now you will build a new FPGA bitstream following these steps:

1. Save the bitstream containing the Development LatticeMico32 microprocessor. Go to the lm32_tutorial directory and rename “platform1.bit” to “platform1_development.bit.” This bitstream is your fail-safe recovery point to allow debugging to continue in the event that the Production LatticeMico32 microprocessor fails to operate.
2. Return to Diamond and run the Bitstream File process. When Diamond finishes running this process, you have a new platform1.bit file. This file contains the Production LatticeMico32 microprocessor.

Merging the Bitstream and the Application Binary to Deploy into the SPI Flash Memory Using Deployment Tool

Now you will merge the FPGA bitstream and the LatticeMico32 bootable Application (LED_Versa) 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:

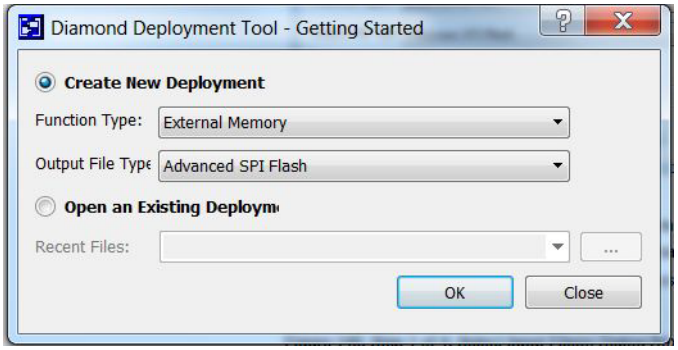
1. Launch Deployment Tool as follows:

- 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 60.

Figure 60: Deployment Tool Getting Started Dialog Box

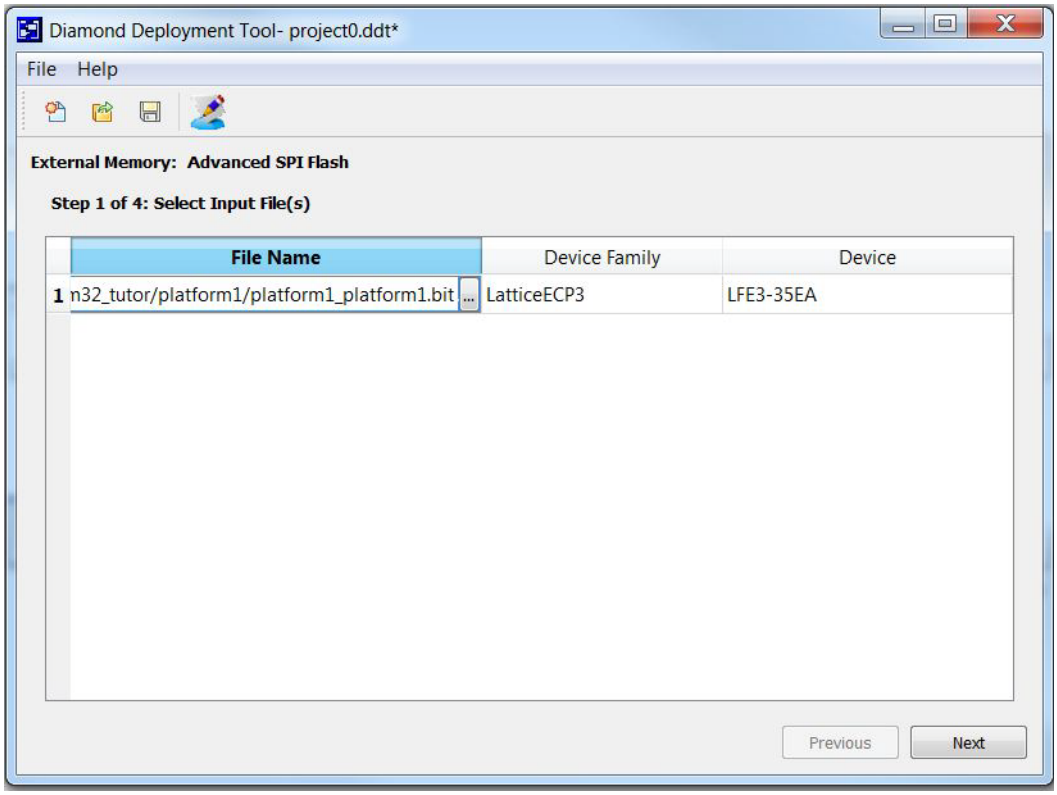


2. In the Function Type dropdown menu, choose **External Memory**.
3. In the Output File Type dropdown menu, choose **Advanced SPI Flash**.
4. Click **OK** to display the Step 1 of 4: Select Input File(s) dialog box, s shown in Figure 61 on page 74.
5. Double-click the File Name box and browse to the FPGA bitstream,

- For Windows: <*Diamond_install_path*>\examples\lm32_tutor\platform1\platform1_platform1.bit
- For Linux: ~/LatticeMico32/examples/lm32_tutor/platform1/platform1_platform1.bit

6. Click **Next** to display the Step 2 of 4: Advanced SPI Flash Options dialog box, as shown in as shown in Figure 62 on page 75.
7. In the Output Format dropdown menu, select **Intel Hex**.
8. In the SPI Flash Size (Mb) dropdown menu, choose **64**.
9. Select the option **Retain Bitstream Header**.
10. Select **User Data Files** option.
11. In the Number of User Data Files dropdown menu, ensure that the number is **1.**
12. In the User Data File 1 box, click ... to browse to the application binary(.bit) file.

Figure 61: Step 1 of 4: Select Input File(s) Dialog Box



- For Windows: <*Diamond_install_path*>\examples\lm32_tutor\LEDtest.bit
- For Linux: ~/LatticeMico32/lm32_tutor/LEDtest.bit

13. In the Starting Address dropdown menu, choose the starting address 0x00120000.
14. Click Next to display the Step 3 of 4: Select Output File(s) dialog box, as as shown in Figure 63 on page 75.
15. In the Output File 1 box, click ... to change output file (.mcs) save location.

    By default it will save in the following location for this tutorial:

    < *Diamond_install_path*>/examples/lm32_tutor/platform1/platform1_platform1.mcs
16. Click **Next** to display the Step 4 of 4: Advanced SPI Flash Options dialog box, as shown in Figure 64 on page 76.
17. Click **Generate**.

This generated file contains the merged FPGA bitstream and the LatticeMico bootable software application (LED_Versa) in a single SPI flash image file that Diamond Programmer can now use for programming the SPI flash.

1. You can see the Deployment Generation Status as shown Figure 65 on page 76.

Figure 62: Step 2 of 4: Advanced SPI Flash Options Dialog Box

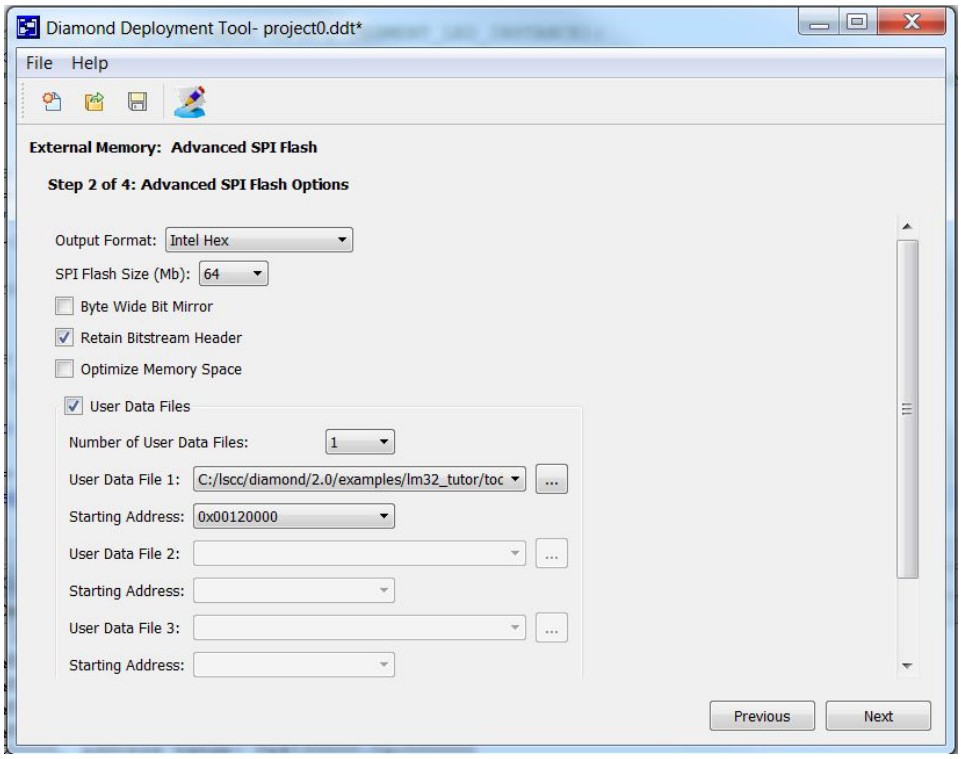


Figure 63: Step 3 of 4: Select Output File(s) Dialog Box

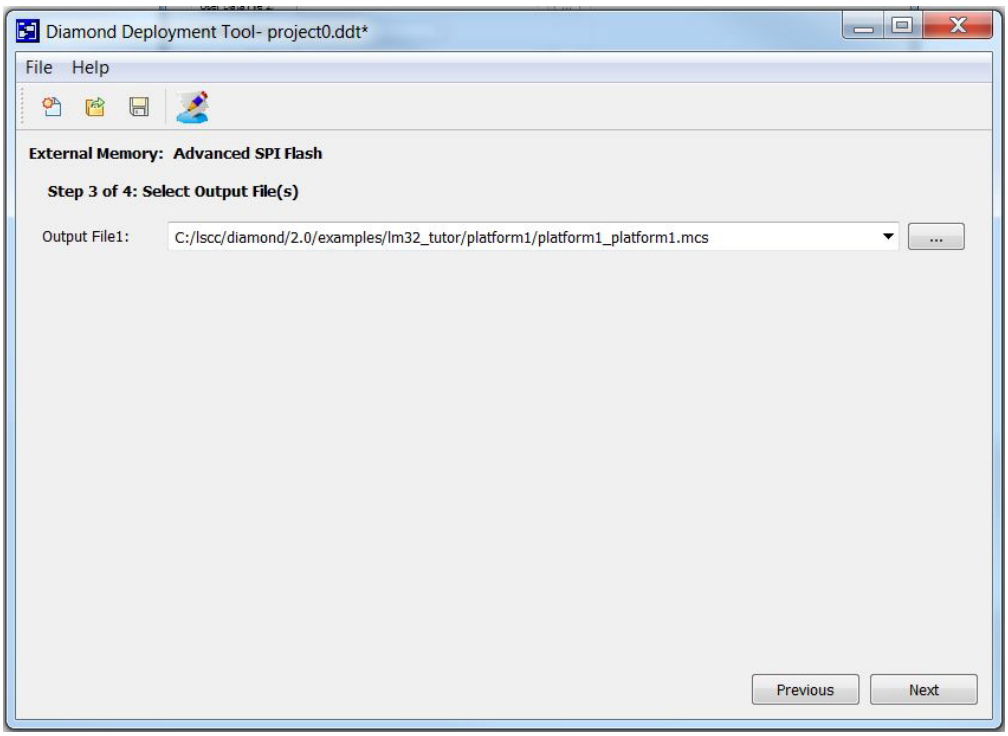


Figure 64: Step 4 of 4: Generate Deployment Dialog Box

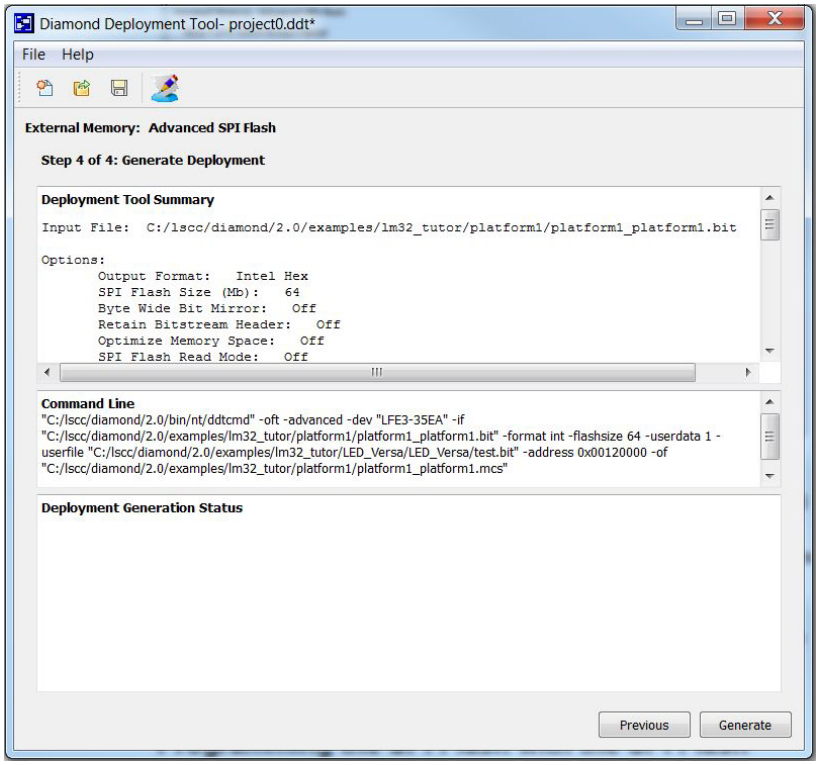
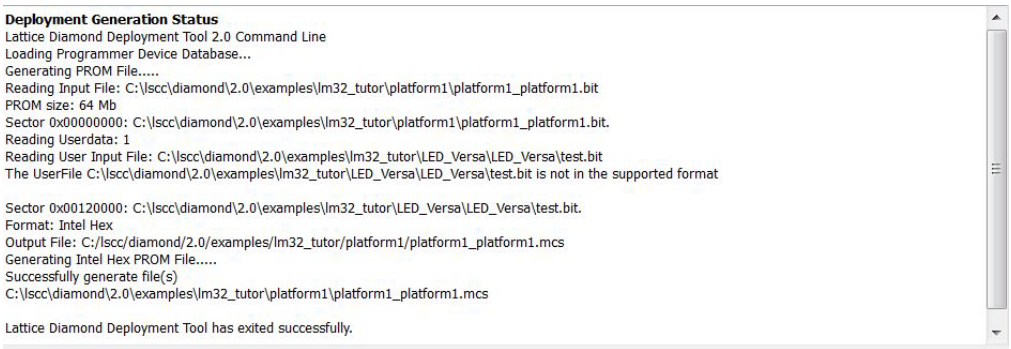


Figure 65: Deployment Generation Status



Task 8: Deploy the SPI Flash Image (Merged With Production Microprocessor Bitstream and Application Software) to SPI Flash Memory

To deploy the microprocessor bitstream:

1. In Diamond, choose **Tools > Programmer.**

In the Programmer window, the **LFE3-35EA** device and platform1.bit should still be displayed. If they are not displayed, follow the instructions in “Task 5: Download the Hardware Bitstream to the FPGA” on page 50.

2. Highlight the row, and then click the **Device Properties** button on the Programmer toolbar to display the Device Properties dialog box.
3. Under Access Mode, select **SPI Flash Background Programming**.
4. Under Operation, select **SPI Flash Erase, Program, Verify**.
5. Under Programming options, click to provide the merged SPI Flash Image (.mcs file) created by the Deployment tool.

   For this tutorial it will be in the following location:

   < *Diamond_install_path*>/examples/lm32_tutor/platform1/platform1_platform1.mcs
6. In the SPI Flash Options box:
   a. Under Family, select **SPI Serial Flash**.
   b. For Vendor, select **STMicro**.
   c. For Device, select **SPI-M2564**.
   d. For Package, select **16-lead SOIC**.
7. Click **Load Size from Programming File** to load the data file size.

   The Device Properties dialog box should resemble the illustration shown in as shown Figure 66 on page 78.
8. Click **OK** in the Device Proerties dialog box.
9. Click the **Program** button on the Programmer toolbar to initiate the deployment.

   Programmer deploys the SPI flash by means of the FPGA. The results are shown in the Programmer output console in shown Figure 67 on page 79.
10. Disconnect and then reconnect the power supply.

    The FPGA takes about three seconds to be programmed by the SPI flash.

    After the FPGA is programmed, the first part of the SPI Flash Image is used to configure the FPGA, after configuration LatticeMico32 microprocessor starts executing from the SPI flash memory base address 0x04120000.

    The code locator built into the LED_Versa application (as part of crt0ram.S), performs the following tasks,

    - Copies the LED_Versa instructions and read/write data from the SPI flash memory and writes them into the EBR.
    - Copies the LED_Versa read-only data from the spi flash memory and writes them into the data inline memory.

      After the software application is copied into the EBR and data inline memory, the code locator performs a control transfer (unconditional branch) and begins running the LED_Versa program. Figure 68 on page 80 illustrates these steps.

Figure 66: Device Properties Dialog Box

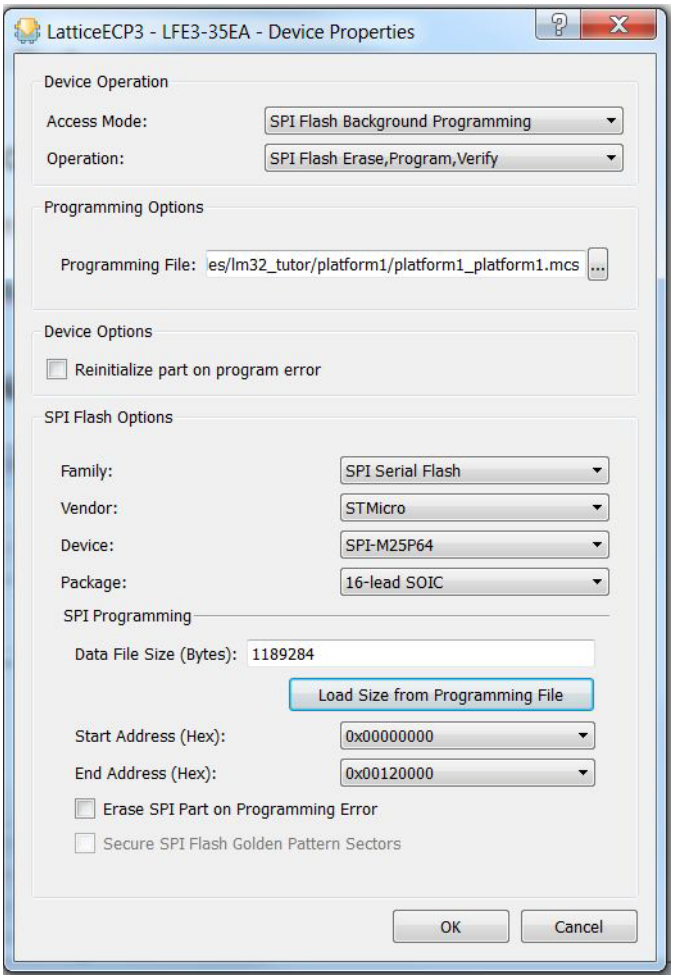


Figure 67: Programmer Output Console

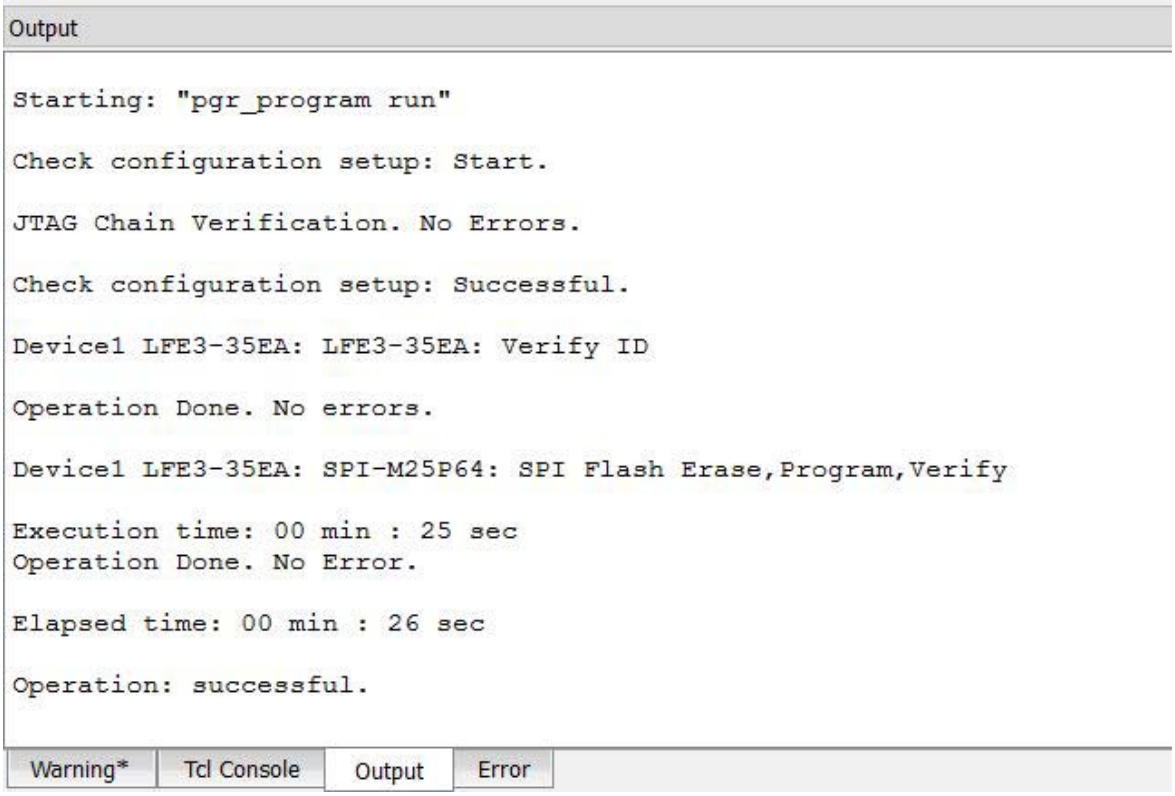
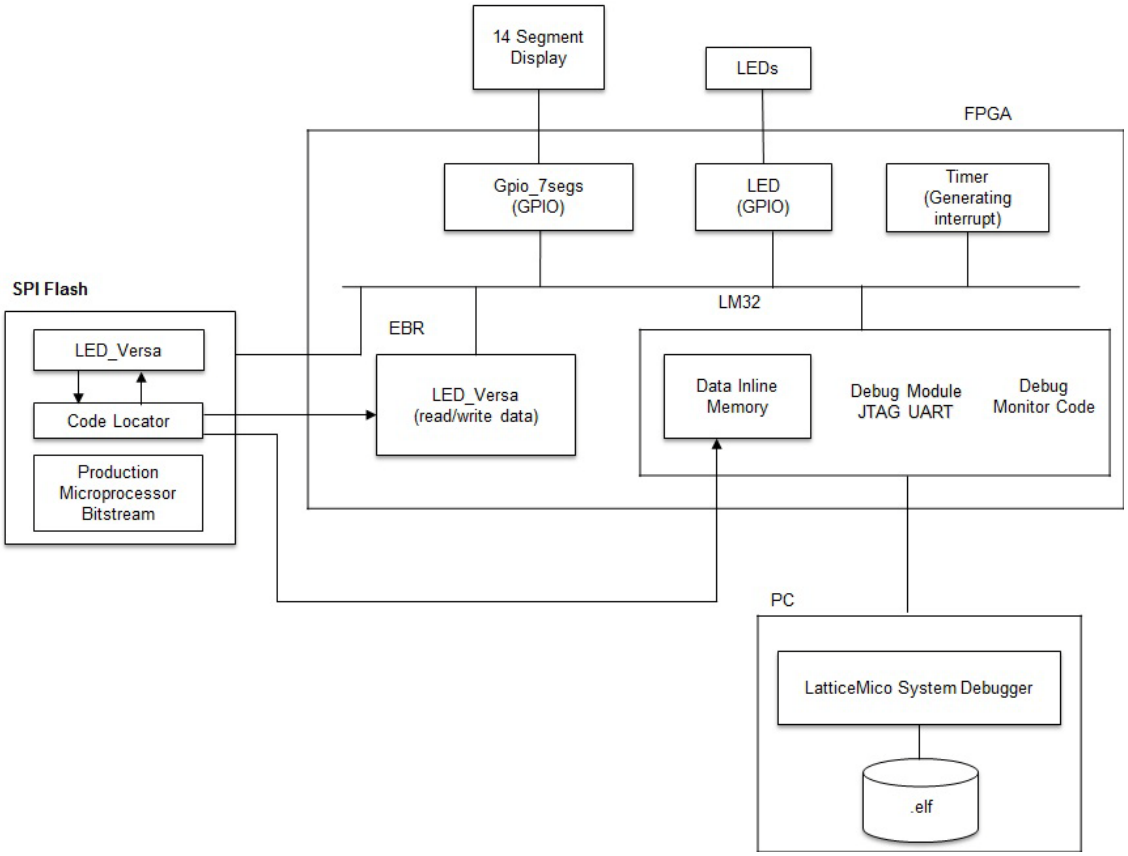


Figure 68: Running the LED_Versa Program



Summary

You have finished the *LatticeMico32 Tutorial*. In this tutorial, you have learned how to do the following:

- Set up a Lattice Diamond FPGA project.
- Create microprocessor platform for the LatticeMico32 embedded microprocessor in MSB.
- Create the software application code for the microprocessor platform with C/C++ SPE.
- Generate a bitstream of the microprocessor platform in Diamond and download it to the board with Programmer.
- Download the hardware bitstream to the FPGA on the board.
- Debug and execute the software application code on the board.
- Download the .elf file containing the software application code to the parallel flash memory.
- Deploy the microprocessor bitstream to the SPI flash memory.

Glossary

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

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 tool chain uses a GNU C/C++ tool chain (compiler, assembler, linker, debugger, and other utilities such as objdump) optimized for the LatticeMico process. It uses the same graphical user interface as MSB.

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.

CFI. CFI is an abbreviation for Common Flash Interface (CFI) parallel flash memory, which is an open standard jointly developed by a number of chip vendors for a type of EEPROM that stores information without requiring a power source.

code-relocator code. Code-relocator code is code that copies the software application code to a destination memory and jumps to the application start address to run the application.

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.

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 platform that provides application frameworks for software application development. 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 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. It is free software produced by the GNU Project.

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.

hardware platform. A hardware platform is the embedded microprocessor in an SoC (system on a chip) design and the attached components, buses, component properties, and their connectivity.

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.

.lpf file. The logical preference file (.lpf) is a post-synthesis FPGA constraint file that stores logical preferences that have been defined in the pre-map stage and post-map stage. This file is automatically generated when you create a new project in Lattice Diamond, and it stores logical preferences only.

master port. A master port is a port that can initiate read and write transactions.

MSB. MSB is an abbreviation for Mico System Builder, which is an integrated development environment based on Eclipse for choosing components, 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 microprocessor and the chosen components. It uses the same graphical user interface as C/C++ SPE.

.msb file. An .msb file is an XML-format file output by MSB.

perspective. A perspective is a combination of windows, menus, and toolbars in the LatticeMico System graphical user interface that enables you to perform a particular task. For example, the Debug perspective has views that enable you to debug the programs that you created in C++ SPE.

project. A project is the software application code written in C/C++ SPE.

PROM. Programmable read-only memory (PROM) is a permanent memory device that is programmed by the customer rather than by the device manufacturer. It differs from a ROM, which is programmed at the time of manufacture. PROMs have been mostly superseded by EEPROMs, which can be reprogrammed.

running. Running is the process of executing a software program.

slave port. A slave port is a port that cannot initiate transactions but can respond to transactions initiated by a master port if it determines that it is the targeted component for the initiated transaction.

software application. The software application is the code that runs on the LatticeMico32 microprocessor to control the components, the bus, and the memories. The application is written in a high-level language such as C++.

SPI. SPI is an acronym for serial peripheral interface, a core that allows high-speed synchronous serial data transfers between microprocessors, microcontrollers, and peripheral devices. It can operate either as a master or as a slave.

watchpoint. A watchpoint is a type of breakpoint that stops the execution of a software program whenever the value of a specific expression changes, without indicating where this may occur. A watchpoint halts program execution, even if the new value being written is the same as the old value of the field.

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.

Recommended References

The following reference materials are recommended to supplement this tutorial:

- *LatticeMico System online Help*. From the LatticeMico Help menu, choose **Help > Help Contents.**
- *LatticeMico32 Hardware Developer User Guide*, which explains how to use the Lattice Mico System Builder to create and configure a hardware platform for the LatticeMico32 embedded microprocessor
- *LatticeMico32 Software Developer User Guide*, which explains how to use C/C++ SPE to program the microprocessor, gives examples of the code used for different parts of the architecture, and describes the processes occurring in the background
- *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.

- *LatticeECP3 Versa Evaluation Board User's Guide*, which describes the features and functionality of the the LatticeECP3 Versa Evaluation Board.
- *Lattice Diamond Installation Guide*, which explains how to install LatticeMico System on the Linux Red Hat operating system.
- *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 Parallel Flash Controller*, which describes the features and functionality of the LatticeMico parallel flash controller
- *LatticeMico DMA Controller*, which describes the features and functionality of the LatticeMico DMA controller
- *LatticeMico On-Chip Memory Controller*, which describes the features and functionality of the LatticeMico on-chip memory controller
- *LatticeMico GPIO*, which describes the features and functionality of the LatticeMico GPIO
- *LatticeMico 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 SPI flash component
- *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 Installation Notice* for the current release, which explains how to install the LatticeMico System software
- *LLatticeECP3 Family Handbook,* which is a collection of the data sheets and application notes on LatticeECP3 devices
- *LatticeECP3 Family Data Sheet*