Lattice RISC-V Embedded Design Guidelines

Application Note

FPGA-AN-02072-1.0

February 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Please refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
AHB-L	Advanced High-performance Bus-Lite
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
BSP	Board Support Package
CLINT	Core Local Interrupter
DMA	Direct Memory Access
EBR	Embedded Block RAM
ELF	Executable and Linkable Format
GDB	GNU Debugger
GPIO	General Purpose Input/Output
I2C	Inter-Integrated Circuit
I3C	Improved Inter-Integrated Circuit
ISA	Instruction Set Architecture
OpenOCD	Open On-Chip Debugger.
PLIC	Platform Level Interrupt Controller
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer-V
ROM	Read Only Memory
RTOS	Real-time Operating System
SDK	Software Development Kit
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory.
TCM	Tightly Coupled Memory
UART	Universal Asynchronous Receiver/Transmitter.
USB	Universal Serial Bus

1. Introduction

1.1. Overview

Lattice embedded system solutions offer RISC-V processors Intellectual Property (IP), memory IP, communication IP (UART, I2C/I3C, SPI, and others), and design software such as Lattice Propel™ Builder, Lattice Propel Software Development Kit (SDK), Lattice Radiant™ software, and Lattice Diamond™ software.

The Lattice Propel Builder is a graphical design tool used to create system that comprises of various modules. These modules are IP that the Lattice Propel Builder offers or custom IP you created. Module can be easily instantiated and connected to other modules in the Lattice Propel Builder Schematic window. The Lattice Propel Builder provides address management for memory-mapped peripherals in the system. Once system building is completed, the Lattice Propel Builder generates the hardware code in RTL files and system environment XML file for building embedded software project.

The Lattice Propel SDK is a complete set of tools to create, compile, and debug embedded software project for processor systems. The Lattice Propel SDK generates the Board Support Package (BSP) for the corresponding system created in the Lattice Propel Builder. The BSP consists of processor start-up code, device drivers, and platform header file that aids application software development. The Lattice Propel SDK provides debugging tools such as OpenOCD and GNU GDB for debugging software on device.

The Lattice Radiant software and the Lattice Diamond software offer the synthesis, mapping, place-and-route, and bitstream generation capabilities. You can create FPGA bitstream using the software for the system created in the Lattice Propel Builder. FPGA pin assignment and timing constraints are performed in this stage. The software Programmer tool offers capability to program the bitstream onto FPGA devices.

The following list shows the documentations for above-mentioned software tools. Complete reading the following documents as a prerequisite to these guidelines:

- A Step-By-Step Approach to Lattice Propel
- Lattice Propel 2023.2 Builder User Guide (FPGA-UG-02196)
- Lattice Propel 2023.2 SDK User Guide (FPGA-UG-02195)

1.2. Purpose

This document provides guidance on designing with Lattice embedded solutions and information regarding various design options for Lattice RISC-V processors and IP. The document is organized into sections covering the entire design flow: selecting the RISC-V processor IP, hardware design, software design, and debugging.

1.3. Audience

The intended audience for this document includes embedded system designers and embedded software developers using Lattice FPGA devices. The pre-requisite of this technical guidelines is the knowledge in digital design, FPGAs, and embedded systems.

2. Lattice RISC-V Processors Family

The Lattice Propel supports three versions of the RISC-V CPU: SM, MC, and RX. Table 2.1 shows an overview of the models and the basic use cases.

Table 2.1. Lattice RISC-V Use Cases

Model	Class	Supported Device Family	Targeted Software Stack	Typical Application
RX	RTOS Capable	Lattice Avant™, Lattice Nexus™, MachXO5™	• RTOS (FreeRTOS™) • Bare metal	• Higher performance • Network connected • External memory support
MC	Microcontroller	Lattice Avant, Lattice Nexus, MachXO3™ (D, L, LF), MachXO2™	• Bare metal • RTOS (third party)	• Mid-performance • Microcontroller replacement • External memory support
SM	State Machine	Lattice Avant, Lattice Nexus, MachXO3 (D, L, LF), MachXO2	Bare metal	• Small footprint • Simple monitoring and configuration applications

The RISC-V SM is the “state machine” version and trades performance for reduced area size. This model is for simple monitoring and control tasks. With small size and area efficient AHB-Lite interconnect, this model is suitable for use in smaller device families such as MachXO2 and MachXO3.

The RISC-V MC is the “microcontroller” version. This model balances performance with area. As shown in Table 2.2, the MC model supports optional features for performance improvement, including the RISC-V ‘M’ extension (hardware-based integer multiply and divide) and instruction and data caches. This model also optionally supports the RISC-V ‘C’ extension that uses 16-bit compressed instructions to save code space. The MC model uses AHB-Lite as the native interconnect and is suitable for use in smaller device families such as MachXO2 and MachXO3.

The RISC-V RX is the highest performing model in the Lattice RISC-V family. This model is “RTOS” capable as it adds Supervisor and User modes from the Privileged Architecture portion of the RISC-V ISA. The RX model uses AXI-4 as the native interconnect for higher performance and more deterministic timing. This model also adds a Platform Level Interrupt Controller (PLIC) and a Core Local Interrupter (CLINT) for managing external interrupts and timers, respectively. A watchdog timer and optional UART are integrated into the RX IP. This IP also supports the custom instructions via the Composable Custom Extension, an emerging industry standard.

Table 2.2. Features Comparison of Lattice RISC-V Variants

Model	Arch	Extensions	System Bus	Interrupt Controller	System Timer	Cache	Memory Type(s)	Watchdog	Privilege Level(s)	Custom Instructions
RX	RV32I	MC F (optional)	AXI-4	PLIC	CLINT	2-Way Set Associative RR	TCM SysMem	Integrated	Machine Supervisor, User	Yes
MC	RV32I	M (optional) C (optional)	AHBL	PIC	Memory Mapped mtime	Optional 2-Way Set Associative RR	SysMem	—	Machine	No
SM	RV32I	—	AHBL	PIC	Memory Mapped mtime	No Cache	SysMem	—	Machine	No

Refer to the respective IP User Guide for the latest resource and performance data for each RISC-V CPU.

3. RISC-V Embedded Hardware Design Guidelines

This section describes the options when designing RISC-V embedded hardware systems using the Lattice Propel Builder.

3.1. RISC-V Processor Reset and Exception Vector

RISC-V processor executes the memory address set by reset vector after released from reset. Table 3.1 shows the supported reset vectors for RISC-V SM, MC, and RX processors. Assign the memory that contains initial software to address matching the reset vector shown in the table.

Table 3.1. RISC-V Processor Reset Vector

RISC-V Processor Variant	Supported Reset Vector[1]
SM	0x0000_0000
MC	0x0000_0000
RX	0x0000_0000

Note:

1. Based on Lattice Propel version 2023.2. Configurable reset vector is not supported.

Exception vector is the memory address that contains the exception handler code. The **Machine Trap-Vector Base-Address (mtvec)** register in the RISC-V processor holds the exception vector. The vector is set by the RISC-V software driver during runtime. The driver is generated as part of the Board Support Package (BSP). Refer to the Interrupts Section for more information.

3.2. RISC-V Processor Memory Selection

Memory device is used to store RISC-V processor instructions and data of a software program. The types of memory devices are described in Table 3.2.

Table 3.2. Types of Memory Device

Types of Memory Device	Example	Description
Volatile memory	• Cache • Random Access Memory (RAM)	• Only retains the data while power is supplied to it. Data is lost when the memory power supply is turned off. • Use as temporary storage and has faster access speed.
Non-volatile memory	• Read Only Memory (ROM) • Flash	• Retains data even when power is turned off. • Use for storing contents permanently and usually has slower access speed compared to volatile memory.

3.2.1. Cache

Cache memory is a high-speed memory that integrates directly into the RISC-V processor. Cache acts as a temporary storage that processor can retrieve data faster. Table 3.3 shows the cache capabilities of the Lattice RISC-V processors.

Table 3.3. RISC-V Processor Caches

RISC-V Processor Variant	Instruction Cache Size	Data Cache Size	Option to Disable Cache	Cache Range
SM	—	—	—	—
MC	4 Kbytes	4 Kbytes	Yes	User Configurable
RX	4 Kbytes	4 Kbytes	No	Instruction: 0x00000000 to 0xFFFFFFFF Data: 0x00000000 to 0x0FFFFFFF[1]

Note:

1. Based on Lattice Propel version 2023.2.

Enabling the processor caches consumes FPGA memory resources for better CPU performance. It is recommended to enable cache for design with Lattice Avant, MachXO5-NX, CrossLink-NX, CertusPro-NX, and Certus-NX devices.

3.2.2. System Memory

System memory provides easy use of Lattice FPGA memory resources (EBR, Distributed RAM, or Large RAM) as an IP available in Lattice Propel. System memory does not require connections to external devices from FPGA and can store RISC-V software and data using FPGA resources.

Table 3.4 shows the system memory IP features and use cases.

Table 3.4. System Memory IP Features and Use Cases

Feature	Use Case
AHB-Lite interface	Use as memory for RISC-V SM and MC
AXI4 interface	Use as memory for RISC-V RX
Configurable as single or dual port memory	With dual port memory, the RISC-V Instruction port can be connected directly to one of the memory ports. The other memory port is for RISC-V data port.
Memory initialization enable	Memory is initialized with RISC-V software during FPGA configuration process. Processor executes instructions stored in memory when out of reset. Typically, use the bootloader software in this scenario.

3.2.3. Tightly Coupled Memory (TCM)

Tightly Coupled Memory provides the processor low-latency predictable access for critical instruction and data. Lattice Propel provides TCM IP that can be used with RISC-V RX processor. The TCM IP supports Local Bus Interface which is connected directly to the RISC-V RX Local Instruction and Data ports. This direct connection provides the RISC-V RX low latency and predictable access to the memory. TCM uses FPGA memory resources like the system memory.

When using TCM, the first 128 Kbytes of processor address range (0x00000000 to 0x0001FFFF) is reserved for it. If system memory (or other types of memory) is used for the processor non-TCM instruction port (for example the AXI port), this memory must start from address 0x00020000.

3.2.4. External SDRAM Memory

External SDRAM memory can be used in applications that require larger memory size. Large program and data can be stored and accessed on the external memory with increased latency. SDRAM memory requires a controller for refreshing the memory and handling memory accesses (such as switching between memory banks, rows, and columns).

Lattice Propel offers SDRAM memory controllers for CertusPro-NX and Avant devices. Both memory controllers support LPDDR4 SDRAM. The memory controller supports AXI4 interface for interfacing to the processor.

Table 3.5 shows the memory controller parameters that you can customize for RISC-V applications.

Table 3.5. LPDDR4 SDRAM Memory Controller Parameters

Parameter	Description
DDR Command Frequency (MHz)	Change the frequency to the desired value that matches your design. The maximum frequency is 800 MHz for Lattice Avant devices and 533 MHz for CertusPro-NX devices.
DDR Density	Change the density (in terms of Gb) per channel to match the SDRAM chip. This change affects the address bus width of the controller AXI4 interface.
DDR Bus Width	Change the bus width for DDR data bus to match the DRAM chip. This change affects the DQ, DQS, and DMI bus width, and requires update to FPGA pin assignments.
Data Width on Local Data Bus	Change the data bus width (range from 32-bit to 256-bit) of the AXI4 interface. RISC-V processor has 32-bit data bus. To avoid width adaptation, set this parameter to match the processor width.

3.2.5. Flash Memory

Flash memory is a non-volatile memory that can store permanent data. Most embedded applications require flash memory to store the processor program (such as Bootloader) and data (such as media file and configuration file). Flash memory accesses are different from volatile memory (such as system memory or SDRAM). Writing data to the flash memory requires erasing the corresponding page before the write occurs. Flash memory controller is required for RISC-V processors to access the memory.

Lattice Propel offers SPI Flash Controller IP to provide interface from RISC-V processor to a SPI based flash device. This IP supports single-bit serial data interface to the flash device. The IP provides AHB-Lite based interface for data access and APB interface for control (registers) access. When connecting the RISC-V RX processor to the AHB-Lite port, a converter bridge is required.

The input delay of the MISO signal (read data from SPI Flash) requires proper timing constrains for the controller to capture the correct data. Use the timing constraint set_input_delay to define the delay contributed by the flash device (refer to flash vendor datasheet) and board traces. Refer to the following examples of input delay constraints for MISO:

- set_input_delay -clock [get_clocks spi_flash0_inst_spi_clk] -max 8 [get_ports miso_i]
- set_input_delay -clock [get_clocks spi_flash0_inst_spi_clk] -min 7 [get_ports miso_i]

3.3. Interconnects and Bridges

Interconnects connect multiple managers to multiple subordinates in a system. Examples of managers include RISC-V instruction port, RISC-V data ports, and DMA. Subordinates include peripherals such as UART, SPI, and I2C controllers. Interconnect provides several functions to the system including the following:

- Decode the managers transaction address and route to the correct subordinate.
- Arbitrate concurrent transactions from multiple managers to a specific subordinate.
- Convert different data widths between managers and subordinates.

Lattice Propel provides Interconnect solutions that are in compliance with Arm Advanced Microcontroller Bus Architecture (AMBA) including AXI-4, AXI-4 Lite, AHB-Lite, and APB protocols. Select the appropriate Interconnect IP for your project. The following sub-sections describe each Interconnect solution in detail.

Bridges connect interfaces of different AMBA protocols, for example connecting an AXI-4 manager to an APB subordinate. The Bridges IP ease system building where you can use the manager or subordinate IP without changing the AMBA interface.

3.3.1. Using AXI4 Interconnect IP

The AXI4 Interconnect IP connects multiple AXI4 based managers to AXI4 subordinates. This interconnect is useful for Lattice RISC-V RX processor-based system as both the instruction and data ports are AXI4 interfaces. The AXI4 Interconnect supports both AXI4 and AXI4-Lite protocols. The protocol type can be configured for individual manager and subordinate.

The AXI4 Interconnect IP is fully parameterizable for your application. Figure 3.1, Figure 3.2, and Figure 3.3 show the IP Wizard for the configurable parameters. The subsequent subsections discuss the guideline when configuring the AXI4 Interconnect IP.

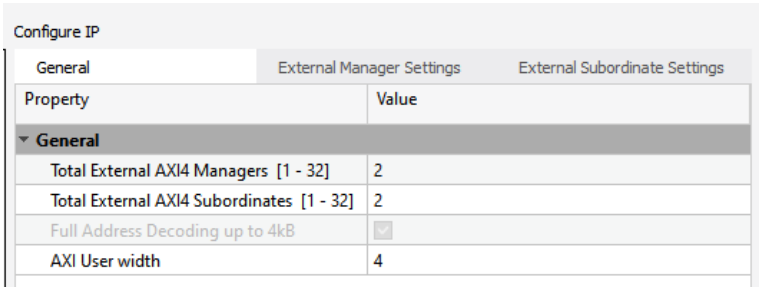

Figure 3.1. AXI4 Interconnect IP Parameters – General Tab

Configure IP

General | External Manager Settings | External Subordinate Settings

Property	Value
General	
External Manager AXI ID width	1
AXI Manager Max Address Width(bits)	32
AXI Manager Max Data Width(bits)	32
AXI Manager Max no.of ID supports	16
External Manager Access Type Settings	
INFO: Ext Manager Access type list	{2'd2,2'd2}
External Manager AXI Access Type 0	WR
External Manager AXI Access Type 1	WR
External Manager Protocol Settings	
INFO: Ext Manager Protocol type list	{1'd0,1'd0}
External Manager AXI protocol 0	AXI4
External Manager AXI protocol 1	AXI4
External Manager CDC Enable Settings	
INFO: Ext Manager CDC Enable list	{1'd0,1'd0}
External Manager CDC Enable 0	☐
External Manager CDC Enable 1	☐
External Manager Address Settings	
INFO: Ext Manager actual address list	{7'd32,7'd32}
External Manager Address width 0	32
External Manager Address width 1	32
External Manager Data Settings	
INFO: Ext Manager actual Data width list	{11'd32,11'd32}
External Manager Data width 0	32
External Manager Data width 1	32
External Manager No.of IDs supports Settings	
INFO: Ext Manager ID supports list	{7'd16,7'd16}
External Manager No.of IDs 0	16
External Manager No.of IDs 1	16

Figure 3.2. AXI4 Interconnect IP Parameters – External Manager Settings Tab

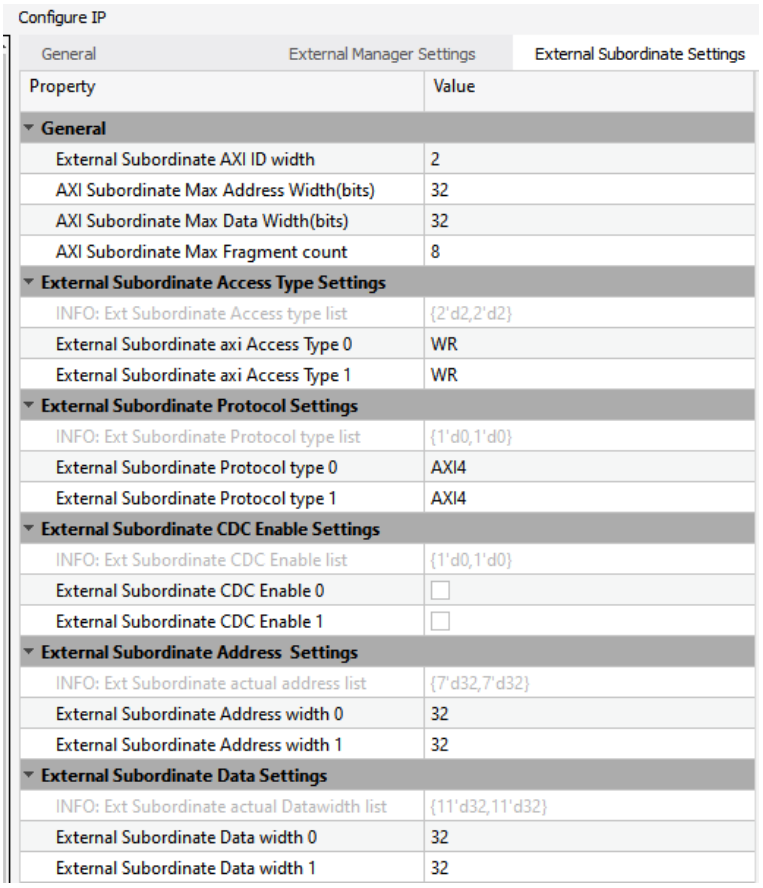

Figure 3.3. AXI4 Interconnect IP Parameters – External Subordinate Settings Tab

3.3.1.1. Total External Managers and Subordinates

The AXI4 Interconnect is configured based on the total number of external managers and subordinates in the system.

The **Total External AXI4 Managers** parameter is set to match with total Managers in the system. For example, RISC-V RX has 2 managers (instruction and data). If other Managers (such as DMA) is connected to the AXI4 Interconnect, increase this number accordingly.

The **Total External AXI4 Subordinates** parameter is set to match with total Subordinates in the system that need to connect with the Managers.

Total managers and total subordinates cannot have the values of 1 because one manager and one subordinate can be connected directly without an interconnect.

3.3.1.2. AXI Address and Data

The AXI4 Interconnect supports configurable address and data widths to match with all external managers and the connected subordinates. Set the maximum allowable address and data width on both the manager and subordinate using the following parameters:

- **AXI Manager Max Address Width**
- **AXI Manager Max Data Width**
- **AXI Subordinate Max Address Width**
- **AXI Subordinate Max Data width**

For each external manager interface, set the actual **External Manager Address Width** and **Data Width** parameters to match with the manager properties. Similar approach applies when setting the **External Subordinate Address Width** and **Data**

Width parameters. For example, the RISC-V RX processor address width is 32 bit and data width is 32 bit. The width value should not exceed the values set in the maximum allowable parameters.

When the manager and subordinate data widths are not equal, the AXI4 Interconnect inserts width converter to handle the differences. For example, when connecting the RISC-V RX processor (data width is 32 bit) to LPDDR4 memory controller subordinate (data width is 256 bit), the interconnect applies width conversion. Note that width converter increases the logic utilization and may impact the timing performance of the achievable Fmax. For optimizing the interconnect usage, configure the managers and subordinates to the same width when possible.

3.3.1.3. AXI USER

AXI User signal is for user-defined purposes. The AXI4 Interconnect IP supports configurable **AXI User Width** parameter that applies to all manager and subordinate interfaces in the IP. Set the AXI User signal (AxUSER, xUSER, and BUSER) width to match the application.

For AXI4 manager to AXI4-Lite subordinate, the AXI4 Interconnect IP ignores the AXI User signals when routing the transactions.

For AXI4 manager to AXI4 subordinate, the AXI4 Interconnect IP passes through the AXI User signals without changing the signals.

3.3.1.4. AXI ID

The AXI ID signals are used for transaction identification and ordering. The AXI4 Interconnect supports configurable AXI ID Width for external managers and subordinates. The **External Manager AXI ID Width** and **External Subordinate AXI ID Width** parameters set the ID widths to match the following equation:

$$\text{Subordinate ID width} \geq \text{Manager ID width} + \log2\,(\text{number of managers})$$

For example, on system with RISC-V RX processor where RISC- RX manager ID width = 1 and number of managers = 2, the subordinate ID width is set to 2 using the equation as follows:

$$\begin{aligned}\text{Subordinate ID width} &= \text{Manager ID width} + \log2\,(\text{number of managers})\\ &= 1 + \log2\,(2)\\ &= 2\end{aligned}$$

The AXI Interconnect external subordinate ID width must set to match with the actual subordinate. For example, if the Interconnect ID width is 2, the subordinate on system memory IP must match. If the subordinate ID width does not match, the extra bits on ID signal are left undriven and may lead to unpredictable behavior.

The **AXI Manager Max no of ID Supports** parameter sets maximum allowable number of ID for external managers arriving at the Interconnect. The **External Manager No of ID** parameter is set for each manager interface to change the interconnect reordering depth for transactions ID tracking. The value set for this parameter should not be larger than the maximum allowable number of ID parameter.

3.3.1.5. Clock Domain Crossing

The AXI4 Interconnect operates at single clock domain which is driven by the **axi_aclk_i** clock input signal. For system with single clock domain, the axi_aclk_i is connected to the same clock as with the rest of the system.

However, some external managers or subordinates operate at different clock than the axi_aclk_i. The AXI4 Interconnect supports clock domain crossing that can be enabled on individual manager or subordinate interface. The **External Manager CDC Enable** and **External Subordinate CDC Enable** parameters is enabled according to the system clock crossing setup.

Figure 3.4 shows RISC-V RX, AXI4 Interconnect, and Memory operate in CLK A domain while DMA operates in CLK B domain. DMA is connected to the AXI Interconnect where the DMA accesses to the memory that operates in CLK A domain.

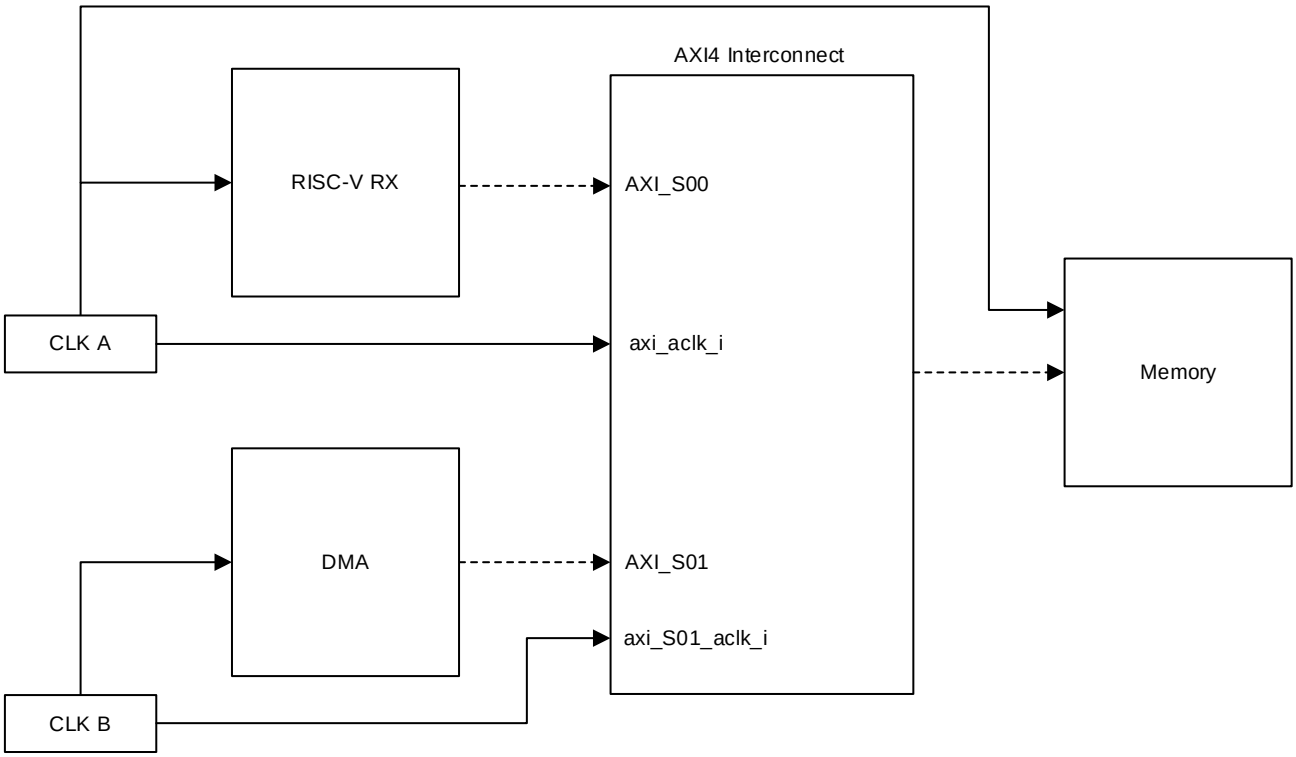

Figure 3.4. Example System with Different Clock Domains

DMA is connected to AXI_S01 port of the AXI4 Interconnect. To turn on clock domain crossing for this port, enable the **External Manager CDC Enable 1** parameter. Additional clock and reset signals listed in Table 3.6 are exported when the CDC feature is enabled.

Table 3.6. AXI4 Interconnect CDC Signals

CDC Signal	Description
axi_S01_aclk_i	Clock input for AXI_S01 port. Connect this to the clock that external manager uses (CLK B).
axi_S01_aclken_i	Clock enable input (active low) for AXI_S01 port. Connect to the clock source enable signal. For example, if the clock is from PLL, connect the PLL Lock output signal to this signal.
axi_S01_aresetn_i	Reset input (active low) for AXI_S01 port. This reset signal is de-asserted synchronously to axi_S01_aclk_i.

3.3.2. Using AHB-Lite Interconnect IP

The AHB-Lite Interconnect IP connects multiple AHB-Lite based managers to AHB-Lite subordinates. This Interconnect is useful for Lattice RISC-V MC and RISC-V SM processor-based systems as both the instruction and data ports are AHB-Lite interfaces.

The AHB-Lite Interconnect IP is fully parameterizable for your applications. Figure 3.5 shows the IP Wizard with the configurable parameters. The subsequent subsections discuss the guideline when configuring the AXI4-Lite Interconnect IP.

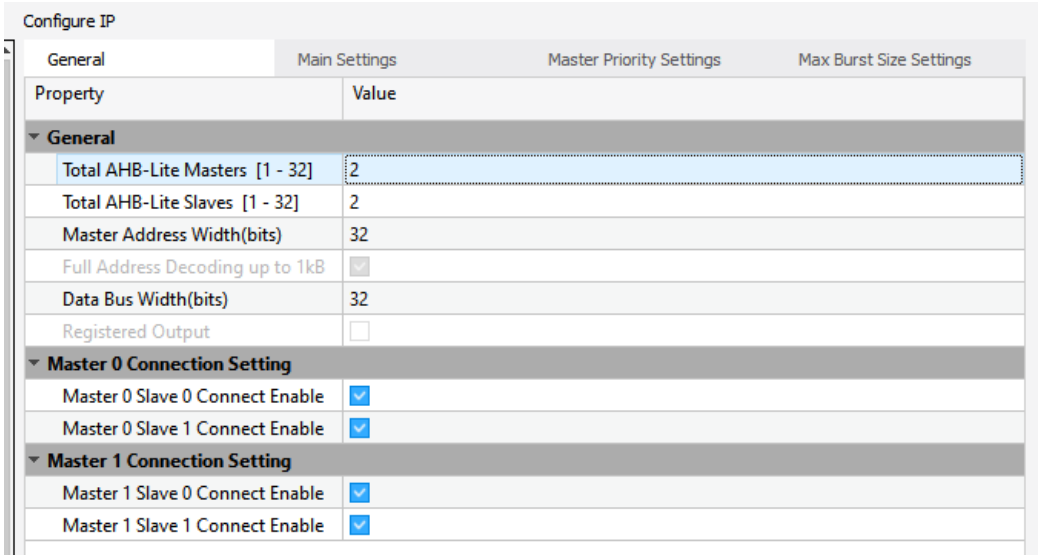

Figure 3.5. AHB-Lite Interconnect IP Parameters – General Tab

3.3.2.1. General Settings

The **Total AHB-Lite Managers** and **Total AHB-Lite Subordinates** parameters set the numbers of managers/subordinates that are connected to the interconnect. The RISC-V MC or SM processor has 2 AHB-Lite manager ports (Instruction and Data). Only the Data port requires connection to multiple subordinates (peripherals). The Instruction port is usually connected directly to memory that stores the program. When the Instruction port require access to a shared memory, increase the value for **Total AHB-Lite Managers** and connect the Instruction port to the AHB-Lite interconnect that grants access to the shared memory.

When the AHB-Lite Interconnect is configured with multiple managers and subordinates, all connections between managers and subordinates are enabled by default. In the example showed in Figure 3.5, 2 managers and 2 subordinates results in 4 interconnection logics. If one of the managers does not require access to a specific subordinate, the **Connect Enable** checkbox can be disabled. For example, if Manager 0 does not access Subordinate 1, uncheck **Manager 0 Subordinate 1 Connect Enable** to reduce the logic generated for the Interconnect and improve overall Fmax.

3.3.2.2. Main Settings

The **Main Settings** tab lists the default base address and range for each subordinate that is enabled on the AHB-Lite Interconnect. The actual addresses are not set via parameters in this setting tab.

When creating RISC-V system with the Lattice Propel Builder, each subordinate base address and range are set up in the **Address** tab of the Lattice Propel Builder GUI. This address information is then propagated to the AHB-Lite Interconnect IP during system generation in the Lattice Propel Builder. The generated system (RTL code) parameterizes the AHB-Lite Interconnect IP with the addresses from the Lattice Propel GUI during IP instantiation.

3.3.2.3. Master Priority Settings

The **Master Priority Settings** tab allows selection of the AHB-Lite subordinates arbitration scheme. Select **either Round Robin** or **Fixed Priority** scheme to match the application requirements.

3.3.2.4. Max Burst Size Settings

The **Max Burst Size Settings** tab allows selection of the maximum burst size for each AHB-Lite subordinates in the interconnect. If the manager performs burst transaction on one subordinate and the transaction exceeds the said maximum burst size setting, the interconnect sends an error response to the manager. This prevents a manager from hogging the bus by generating very long bursts.

3.3.3. Using APB Interconnect IP

The APB interface is commonly used for low bandwidth bus interface such as peripherals access. In a typical embedded system, the processor interfaces to multiple peripherals such as I2C controller, UART, and GPIO. These peripherals do not require high performance bus and typically uses the APB interface.

RISC-V RX has AXI4 interface and RISC-V SM or MC has AHB-Lite interface. The processors use conversion bridges such as AXI4-to-APB bridge to connect the processor interface to the APB-based peripherals. These bridges operate on per interface basis where a bridge is required for each peripheral. APB Interconnect allows a single manager to connect to multiple subordinates, eliminating the need for individual bridge.

Figure 3.6 shows the RISC-V RX data port accessing to GPIO, SPI flash controller, and LPDDR4 memory controller. All these peripherals have APB based subordinate interface. A single AXI-to-APB bridge converts the AXI interface from the AXI4 Interconnect to APB interface. Subsequently, the APB Interconnect connects the interface to multiple peripherals.

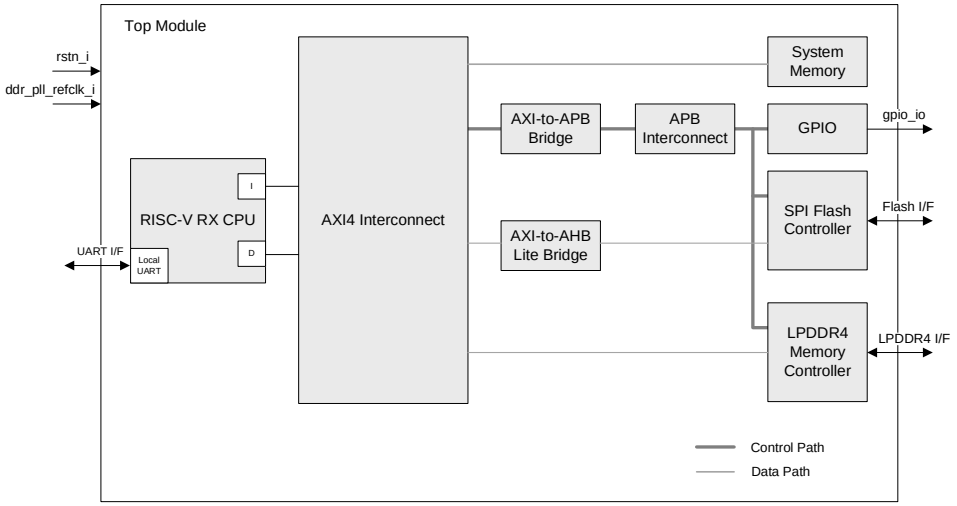

Figure 3.6. System with APB Interconnect for Peripherals Access

3.3.4. Using Bridges IP

Bridges connect different types of AMBA interfaces in the Lattice Propel system. The following bridges are available in the Lattice Propel Builder:

- AXI4-to-AHB-Lite bridge
- AXI4-to-APB bridge
- AHB-Lite-to-APB bridge

3.3.4.1. AXI4-to-AHB-Lite Bridge

This bridge connects RISC-V RX AXI4 interface to any AHB-Lite subordinate interface. As the RISC-V processor has 32-bit data width, the **AXI_AHB Data Bus Width** parameter is set to 32.

The **AXI ID Width** parameter is set to the value that matches the upstream manager. For example, if the upstream manager is an AXI4 Interconnect with ID Width of 2, the bridge value is set to 2.

The **AXI User Width** parameter defines the AXI USER signal width and is set to a value that matches the upstream manager.

The AXI4-to-AHB-Lite bridge does not support clock domain crossing where the AXI4 and AHB-Lite interfaces operate at different clock domains. However, if AXI4 Interconnect is used as the upstream, you can enable clock domain crossing in the AXI4 Interconnect. Refer to the Using AXI4 Interconnect IP section for details.

3.3.4.2. AXI4-to-APB Bridge

This bridge connects RISC-V RX AXI4 interface to any APB subordinate interface. As the RISC-V processor has 32-bit data width, the **AXI_APB Data Bus Width** parameter is set to 32.

The **AXI ID Width** parameter is set to the value that matches the upstream manager. For example, if the upstream manager is an AXI4 Interconnect with ID Width of 2, the bridge value is set to 2.

The **AXI User Width** parameter defines the AXI USER signal width and is set to a value that matches the upstream manager.

The AXI4-to-APB bridge does not support clock domain crossing where the AXI4 and APB interfaces operate at different clock domains. However, if AXI4 Interconnect is used as the upstream, you can enable clock domain crossing in the AXI4 Interconnect. Refer to the Using AXI4 Interconnect IP section for details.

3.3.4.3. AHB-Lite-to-APB Bridge

This bridge connects RISC-V SM or MC AHB-Lite interface to any APB subordinate interface. Configure the bridge **Address Width** and **Data Bus Width** parameters to 32-bit when interfacing with RISC-V processors.

The bridge supports a separate APB clock domain from the AHB-Lite clock domain. This allows the slower peripherals to be decoupled from the system clock domain and may make it easier to reach timing objectives. Check the **APB Clock Enable** parameter to enable separate clock and reset signals for APB domain, namely pclk_i and presetn_i.

3.3.5. Using Feedthrough IP

The Feedthrough IP allows you to export the specific AMBA bus out from the Lattice Propel Builder system to the higher-level module. This is useful when connecting IP that is not available in the Lattice Propel Builder. For example, you can connect the exported bus to an IP in the Lattice Radiant project. The Lattice Propel Builder offers AHB-Lite Feedthrough and APB Feedthrough IP that export the AHB-Lite and APB bus respectively.

Figure 3.7 shows the Lattice Propel Builder system with the feedthrough IP. The AHB-Lite feedthrough (ahbl_feed_inst) is connected from the AHB-Lite Interconnect for the processor to access to external IP. Similarly, the APB feedthrough (apb_feed_inst) is connected to APB Interconnect located at the top right of Figure 3.7.

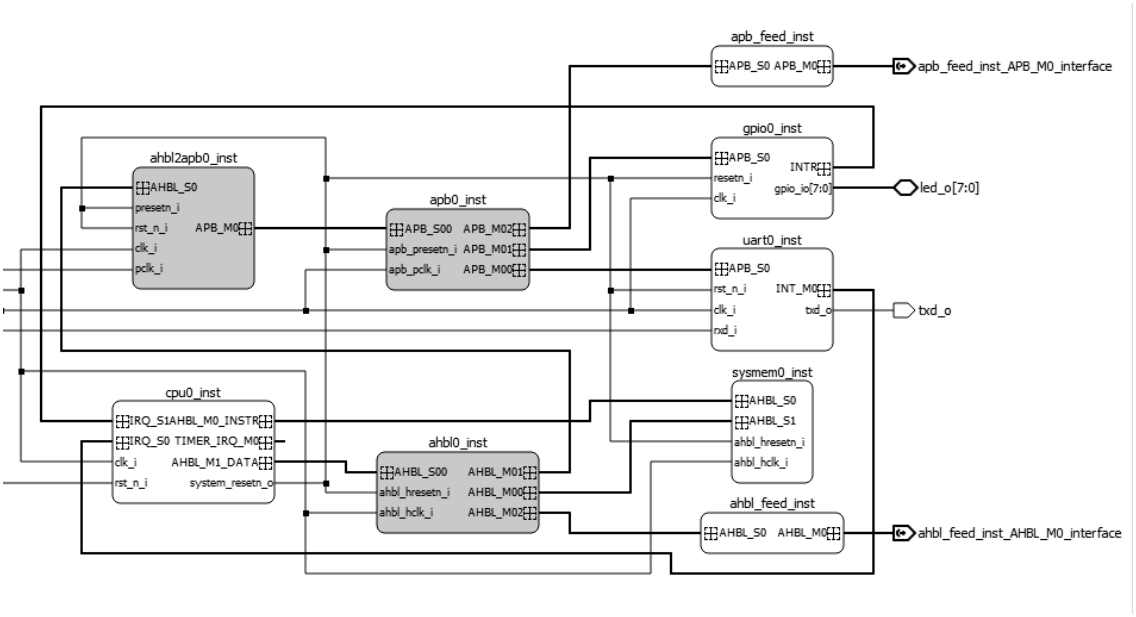

Figure 3.7. Lattice Propel Builder System with Feedthrough IP

When connecting to the external subordinate, set **Export Interface As** to **Slave** in the Bridge Module/IP Block Wizard as shown in Figure 3.8.

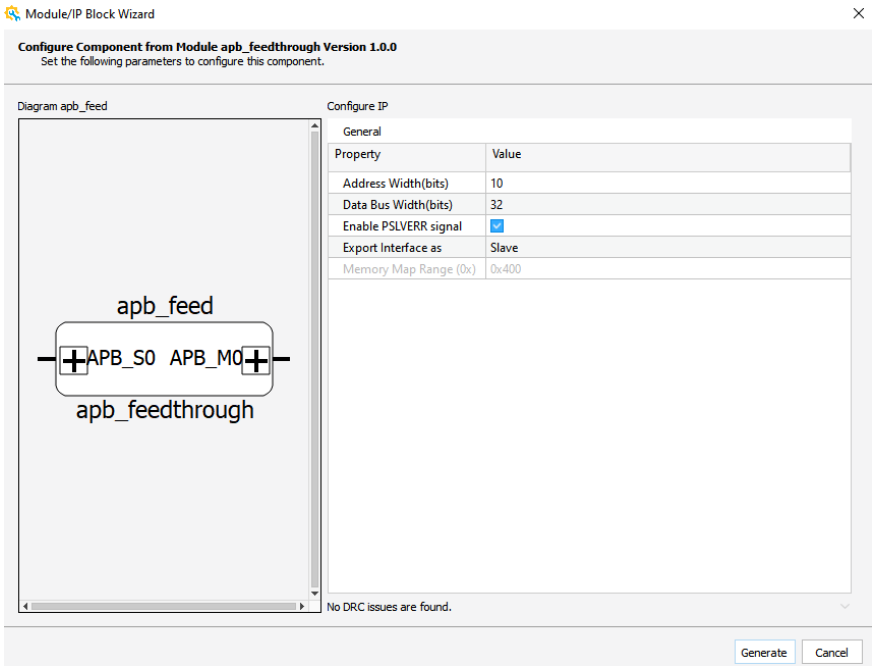

Figure 3.8. APB Feedthrough Module/IP Block Wizard

To create a memory-mapped address space for the feedthrough block, click **Generate** in the Bridge Module/IP Block Wizard in Figure 3.8. When interface to RISC-V processor, the software code accesses the block via the assigned address. Figure 3.9 shows the address space for the APB feedthrough in the **Address** tab in the Lattice Propel Builder.

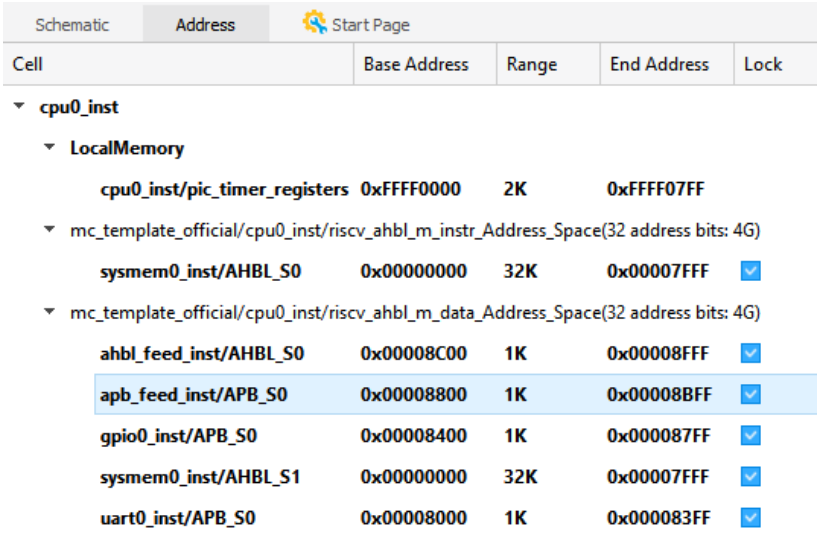

Figure 3.9. Address Map of Feedthrough IP

3.4. Assigning Address Map

The Lattice Propel Builder supports assigning addresses to all memory-mapped peripherals in the embedded system. The address map is managed via the **Address** tab in the Lattice Propel Builder. The **Auto Assign** feature in the Lattice Propel Builder allows the tool to automatically assign non-overlapping addresses to all peripherals. However, there are situations where manual address assignments are required.

Peripherals addresses must not be assigned to the processor cacheable range, so that the processor reads the peripheral registers value from the actual register value and not from the cache.

For RISC-V RX processor, the cache range is from address 0x0000_0000 to 0x0FFF_FFFF[1]. For peripherals that do not need caching, do not assign the addresses into this address range. Figure 3.10 shows the address map for GPIO and UART peripherals that are assigned to non-cacheable range of the RISC-V RX processor that starts from address 0x1000_0000.

When the addresses are assigned correctly, check the **Lock** option to avoid unnecessary changes by you or the Auto Assign feature.

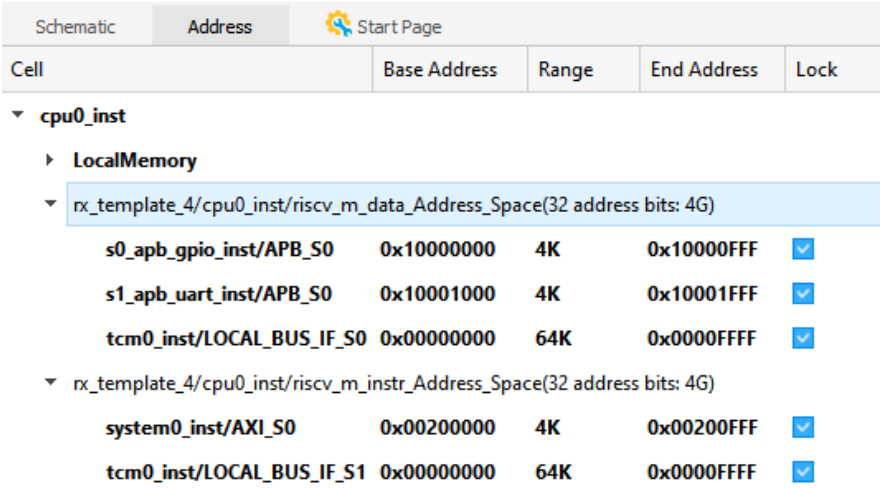

Figure 3.10. Address Map for Peripherals

When ICACHE and DCACHE features are turned on for RISC-V MC processor, the address assignment method must be applied for peripherals that do not need to be cached. The cacheable range is configured in the RISC-V MC IP Wizard via **DCACHE_RANGE_LOW** and **DCACHE_RANGE_HIGH** parameters.

Note:

1. *Based on Lattice Propel 2023.2 and RISC-V RX 2.3.0. Refer to the* RISC-V RX CPU IP Core User Guide (FPGA-IPUG-02241) *for more information.*

4. RISC-V Embedded Software Design Guidelines

This section describes the options when designing RISC-V embedded software systems using the Lattice Propel Software Development Kit (SDK).

4.1. Software Project Support

RISC-V SM, MC, and RX processors support bare metal software applications. The Board Support Package (BSP) provides drivers for processor startup initialization and accessing the IP in the system. When creating a new Lattice C/C++ Project in the Lattice Propel SDK, select **Hello World Project** for bare metal based applications.

The Lattice Propel SDK supports creating FreeRTOS based projects. FreeRTOS is available only when using the RISC-V RX processor. When creating a new Lattice C/C++ Project in the Lattice Propel SDK, select the **FreeRTOS Project**. For more information, refer to the FreeRTOS web page.

4.2. Board Support Package (BSP)

The Board Support Package (BSP) is generated when creating a Lattice C/C++ Project in the Lattice Propel SDK. The BSP provides the following:

- RISC-V processor start-up code and drivers for interrupt, exception, timer, cache, and watchdog
- Device drivers for IP in the system
- Platform header file that defines memory-mapped addresses, IP parameters, and C Macro

Table 4.1 lists the BSP components and the corresponding file hierarchy in the Lattice Propel SDK project.

Table 4.1. BSP Components and Hierarchy

Component	Hierarchy
RISC-V processor drivers	For RISC-V RX: *<Project Name>/src/bsp/driver/riscv_rtos/* For RISC-V MC: *<Project Name>/src/bsp/driver/riscv_mc/* For RISC-V SM: *<Project Name>/src/bsp/driver/riscv_sm/*
Device drivers	*<Project Name>/src/bsp/<device name>* Example: *<Project Name>/src/bsp/driver/gpio/*
Platform header file	*<Project Name>/src/bsp/sys_platform.h*

4.2.1. Updating BSP

Update BSP when changes are made in the Lattice Propel Builder. This includes adding, removing, or upgrading IP, changing IP parameters, and changing the memory-mapped addresses. To update BSP, follow these steps:

1. Click **Design > Generate** in the Lattice Propel Builder to generate the latest system.
2. In the Lattice Propel SDK, under **Project Explorer** view, select the C/C++ project to update.
3. Click **Project > Update Lattice C/C++ Project…**.

   The **Update System and BSP** wizard opens as shown in Figure 4.1.

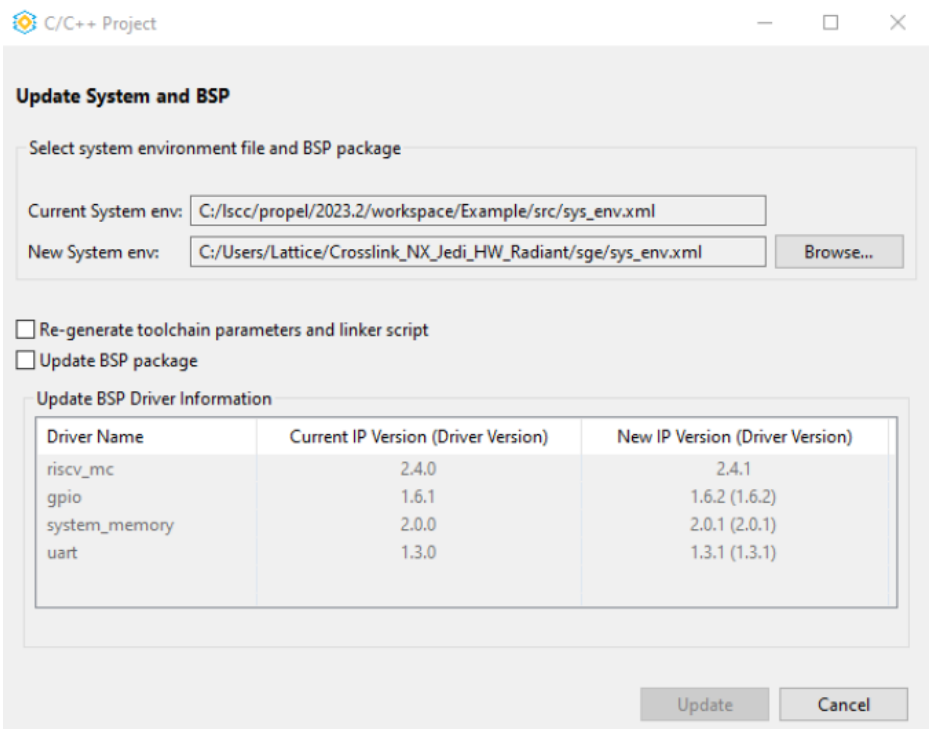

Figure 4.1. Update System and BSP Wizard

4. If your Lattice Propel Builder project has a different path as **Current System env**, click **Browse** to the latest Lattice Propel Builder project **sge/** directory and select the **sys_env.xml** file.
5. If you change the processor system memory, check **Re-generate toolchain parameters and linker script**. For example, if you change the processor connection to the program memory, update the connection in the linker script. Do not check this option if there is no change to maintain the current linker script.
6. Check **Update BSP package** to update the device drivers. The wizard shows the new driver version (when available) that for the updated BSP.
7. Click **Update** to make changes.

4.3. Software Compiler Optimization Options

Compiler optimization is a tool to improve performance and reduce code size with any modern CPU. Optimization is particularly important when dealing with Reduced Instruction Set Computer (RISC) architectures. RISC architectures such as RISC-V implicitly assume that the compiler solves structural issues that other architectures handle at the hardware level. Furthermore, well optimized code can make efficient use of the deep register files associated with RISC architectures, minimizing potentially costly accesses to main memory.

Optimization can significantly reduce instruction count, decrease memory footprint, and increase performance.

During the development phase, optimization may be counterproductive. Aggressive optimization increases compile times and slows down development. Optimization may also remove or reorder instructions, complicating debugging. For example, when single stepping through optimized code, execution may not be consistent with the high-level source (even though the code logically produces the correct result).

4.3.1. C/C++ Compiler Flow

The conventional process of compiling C/C++ source code into an executable binary image consists of four phases, as shown in Figure 4.2:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

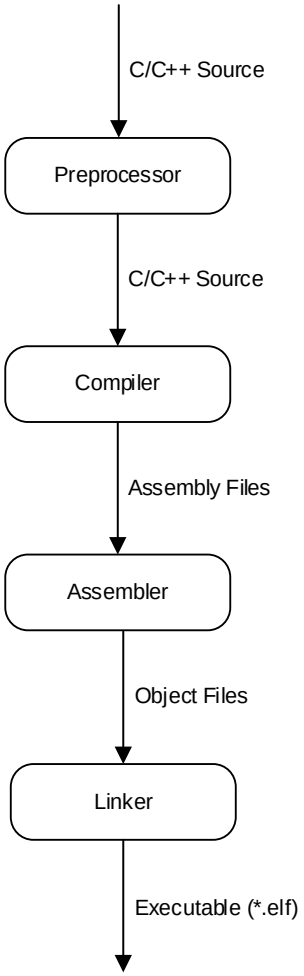

Figure 4.2. Phases of Compilation

Linking is the only phase that has a complete view of the entire program. Global optimizations across the whole program can only be performed by the linker.

Most optimizations are local optimizations performed by the compiler. Optimizations are performed on each source code module, one at a time.

4.3.2. Compiler Architecture

Compiling refers to converting high-level source into binary machine code. However, as shown in the C/C++ Compiler Flow section, the compiler performs only one part of the overall process.

The Lattice Propel SDK uses the GNU toolchain to build application images. The GNU compiler has three stages as shown in Figure 4.3.

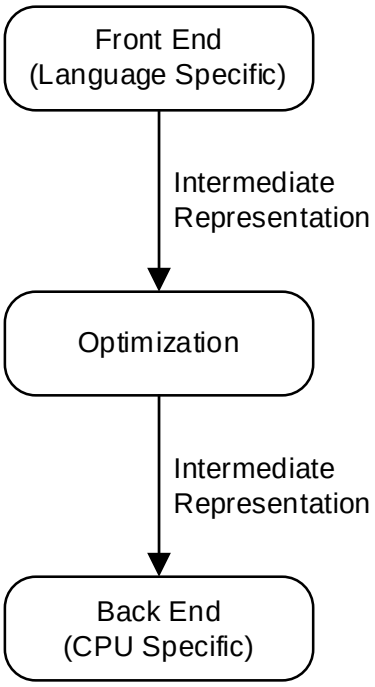

Figure 4.3. Generalized Compiler Architecture

The first stage which is the front end, is specific to a high-level language. The front end parses and analyzes high-level source code and converts the source code into a language independent intermediate representation that is passed to the next stage.

The second stage optimizes the intermediate representation that was produced by the front end. The nature of the optimization (for example performance versus code size) and the degree of effort are controlled by command line arguments to the compiler. When the second stage has completed optimization, a new, language independent, intermediate representation is passed to the final stage which is the back end.

A back end is specific to a CPU architecture. The primary function is to convert the intermediate representation produced by the optimization phase into instructions that can execute on the target hardware. The output of the back end is assembly code that can be passed to the assembler, the next stage after the compiler in the build flow.

With this modular compiler architecture, the same optimization engine and target specific back end can support multiple high-level languages by using a different, language specific front end. Similarly, the same front-end and optimization engine can generate code for different variants of the same CPU architecture (for example 32-bit versus 64-bit) and different architectures (for example ARM versus RISC-V).

The front and back ends do little to optimize the code. If the compiler is invoked with optimization disabled, the output is a literal translation of the input source into assembly code. The output code has no logical reductions in instruction count and is not optimized for architectural features of the target CPU.

4.3.3. Optimization Levels

The GNU compiler supports many optimization algorithms. For a complete list of these algorithms, refer to the GCC, the GNU Compiler Collection web page. Most optimizations reduce the number of instructions in the compiled code which increases performance and reduces the program size in memory. However, some optimization algorithms increase code space to enhance performance or vice-versa.

Some optimization routines are iterative and can significantly increase compilation time.

The GNU compiler allows individual optimization algorithms to be invoked at the command line. The compiler also groups commonly used algorithms into different levels based on the desired tradeoffs. The levels and the corresponding command line arguments are shown in Table 4.2.

Table 4.2. GNU Compiler Optimization Levels and Command Line Arguments

gcc Command Line Argument	Description
-O0	Disables optimization
-O1	Runs optimizations that increase performance without increasing memory size and compilation time
-O2	Runs optimizations in -O1 and optimizations that increase compilation time
-O3	Runs optimizations in -O2 and adds optimizations that sacrifice code space for improved performance
-Ofast	Runs optimizations in -O3 and adds several optimizations that are not valid for C/C++ standards compliant code
-Os	Runs optimizations that reduce program size without impacting the performance
-Og	Runs subset of optimizations that do not alter execution order that is useful for debugging

4.3.4. C/C++ Compiler Optimization in the Lattice Propel Software

The Lattice Propel SDK invokes the GNU compiler to build applications. You can control the arguments that the Lattice Propel software sends to the compiler, including optimization level, via the GUI settings.

You can change the optimization level using the GUI in several ways. During project creation, you can change the optimization level in the C/C++ Compiler tab in the Lattice Toolchain Setting window, as shown in Figure 4.4.

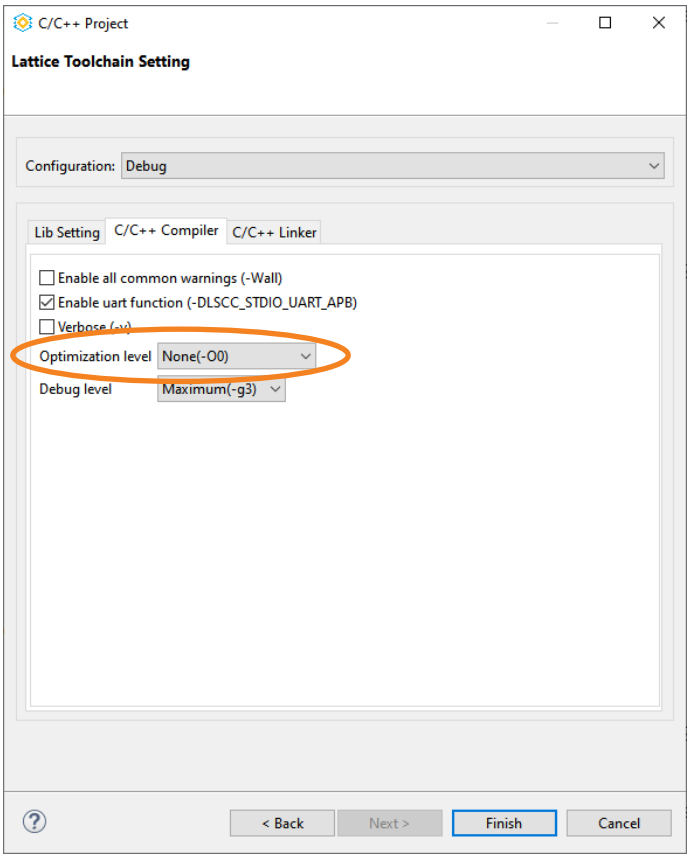

Figure 4.4. Optimization Setting during Project Creation

You can also change the optimization level after project creation in the Project Properties menu as shown in Figure 4.5.

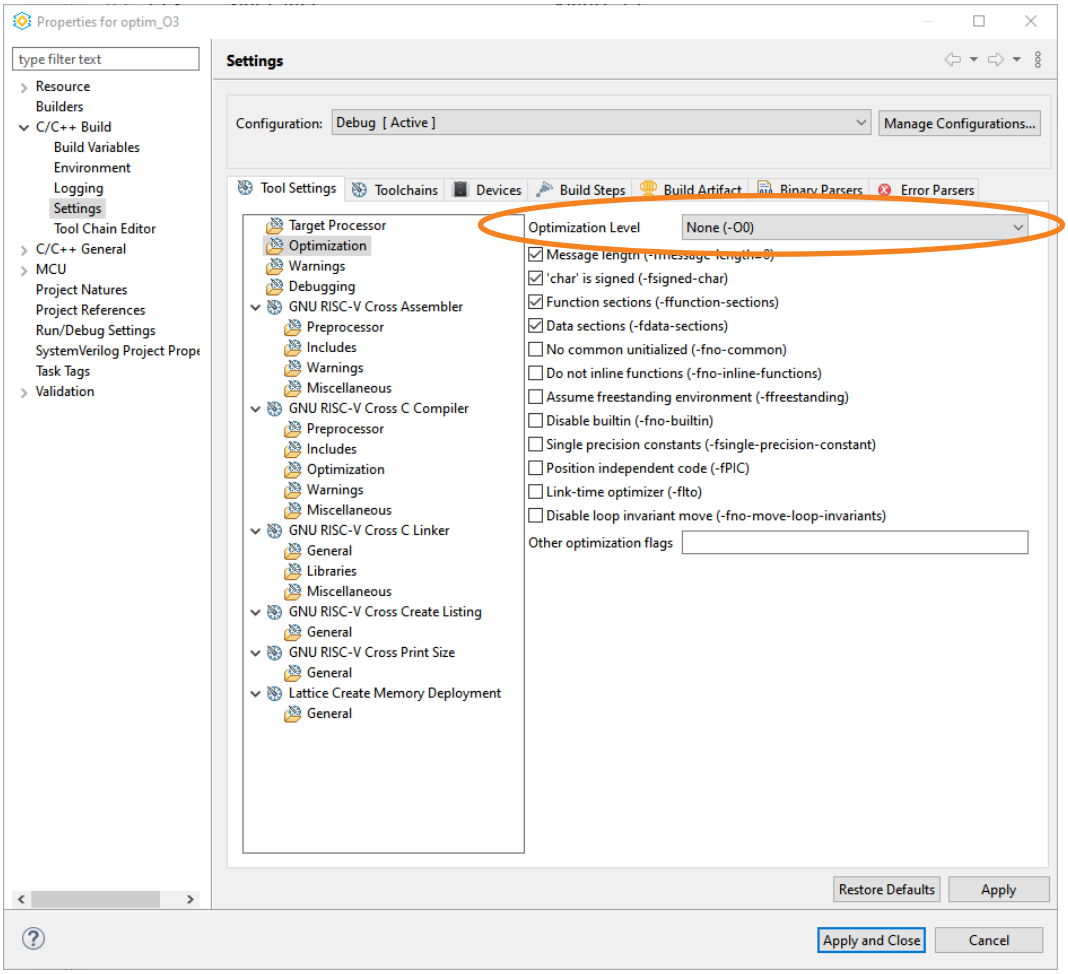

Figure 4.5. Changing Optimization Level from the Lattice Propel SDK Project Settings

The default Optimization Level is -O0 which turns off optimization. During the initial stages of a project, it is recommended to turn off optimization so that compile times remain short and instruction ordering matches that of the source code. For most applications, enable optimizations in the design cycle prior to the creation of a release candidate.

4.3.5. Fine Grain Control of Optimization

You can apply different levels of optimization to different parts of an embedded program using the GNU C/C++ compiler.

As shown in Figure 4.6, the `#pragma GCC optimize` command optimizes all subsequent code at the optimization level specified in the argument in double quotes.

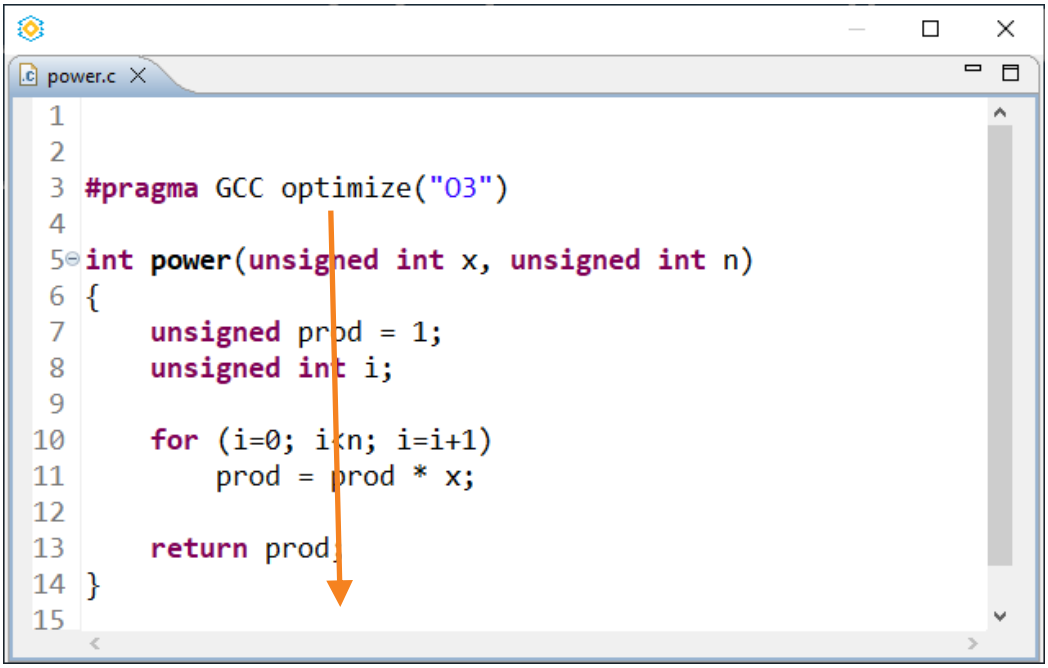

Figure 4.6. Using the #pragma Directive to Control Optimization Level

You can also apply optimization at the granularity of functions using the `__attribute__` in the GNU compiler. Different attributes can apply useful properties to functions, variables, type definitions, and optimization level. Figure 4.7 demonstrates the syntax for applying level 3 optimization to a function. The `__attribute__` keyword applies only to the specified function.

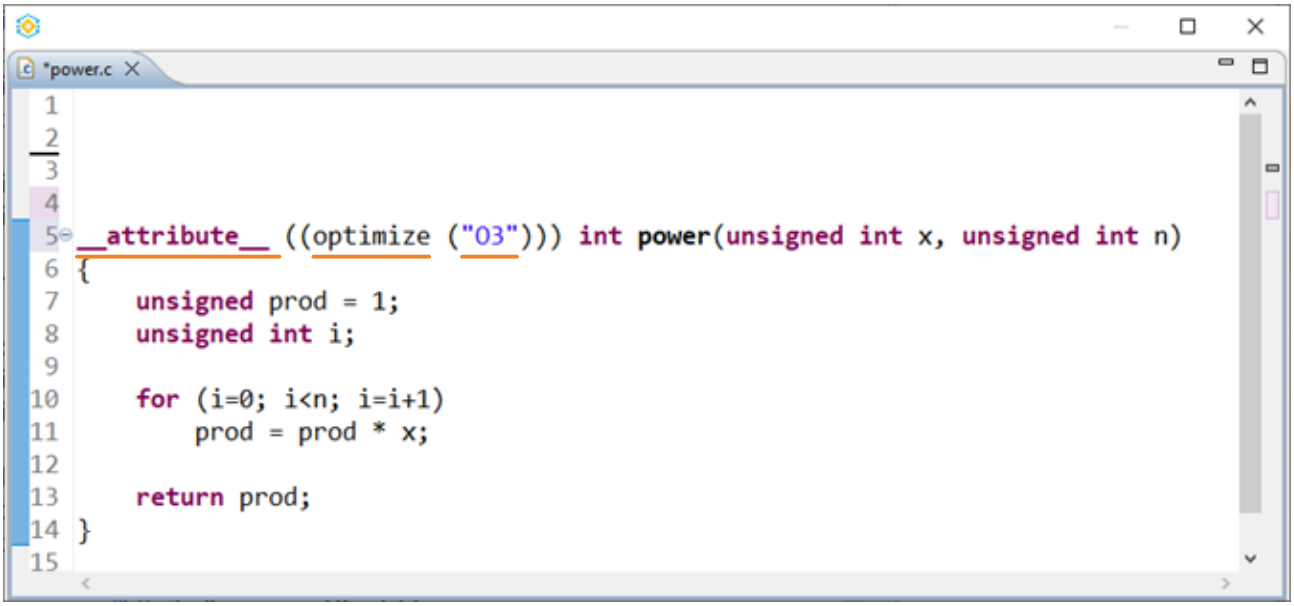

Figure 4.7. Applying an Optimize Attribute to a Function

4.3.6. Linker Relaxation

Some optimizations can only be applied globally. For RISC-V architecture, because of the design of the jump, load, and store instructions, linker relaxation optimization can only be performed once the whole program has been assembled and the locations of code and data are known.

The width of RISC-V instructions is equal to or less than the width of the address bus (32 bits). A RISC-V instruction cannot contain an immediate value that spans across the entire addressable memory space. Instead, RISC-V instructions compute a target address by adding a smaller immediate offset to a value contained in a register.

Loads and stores add a twelve-bit immediate field in the instruction to the contents of a register in the CPU register file. Jump instructions compute the target address by adding an immediate offset to either a register in the register file or to the Program Counter.

If the target address of a jump or a memory access is unknown at the time the instructions are emitted by the compiler, the compiler assumes the target address is outside the range of the offset provided by the immediate field. To cover the full range of potential target addresses, the compiler includes an instruction to load the higher order address bits of the target address into a register.

Because of code locality, the extra instruction is typically not necessary for most load, store, and jump instructions. However, it can only be determined at the linking stage whether a given operation requires a register to be loaded with the high order bits of its target address.

During linker relaxation, the linker scans the program as a whole and determines whether the target address of an operation is within the range of the immediate field from a register that holds a known value. The register for jump instructions can be the Program Counter. For load and store instructions, the register is in the register file and is designated as the Global Pointer. The Global Pointer is loaded during startup with an address that is near the location of most variables in memory.

If the linker detects that an operation can reach the target address without loading a new register value, the linker removes the extra register load instruction added by the compiler. This cuts in half, the number of instructions required for loads, stores and jumps. Linker relaxation results in significant gains in terms of both code size and performance.

Linker relaxation is enabled by default.

4.3.7. Link Time Optimization

Link time optimization is an option provided by the GNU C/C++ compiler and is a global optimization. This optimization forwards extra information to the linker and causes the linker to run optimization on the program.

Link time optimization may marginally improve performance. Link time optimization may increase code size.

Refer to the GNU GCC documentation for additional information.

Link time optimization is disabled by default.

4.4. Linker Script

Linking is the final step in translating a set of source files and libraries into an executable binary. Prior stages in the build process, which are the pre-processor, compiler, and assembler, produce a set of object files that the linker combines to form the complete program. Figure 4.8 shows the phases of compilation.

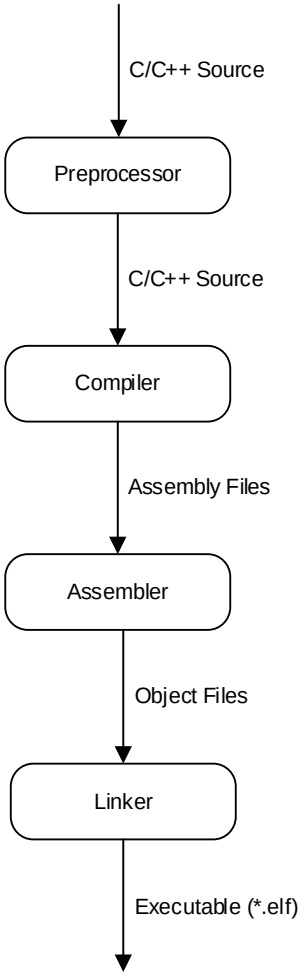

Figure 4.8. Phases of Compilation

You can control the GNU linker utility, ld, via command line arguments. For simple, natively compiled applications, the default settings are sufficient. More complex embedded applications require more control over the linking process. Control via command line becomes cumbersome and it is common practice to place linker arguments into a linker script file.

4.4.1. Linker Scripts in the Lattice Propel SDK

The Lattice Propel SDK automatically generates a linker script during the Lattice C/C++ Project creation. The autogenerated script is based on the memory configuration of the Lattice Propel Builder SoC design. This default script is suitable for simple designs. For more complex designs, add customizations to the default script.

The Lattice Propel SDK supports the following modifications to a linker script:

- GUI based linker script configuration
- Text based editing

4.4.1.1. GUI Based Linker Script Configuration

To open the GUI based linker script editor in the Lattice Propel SDK, double click the linker.ld file in the src directory of your Lattice C/C++ project. Figure 4.9 shows the Linker Configuration GUI.

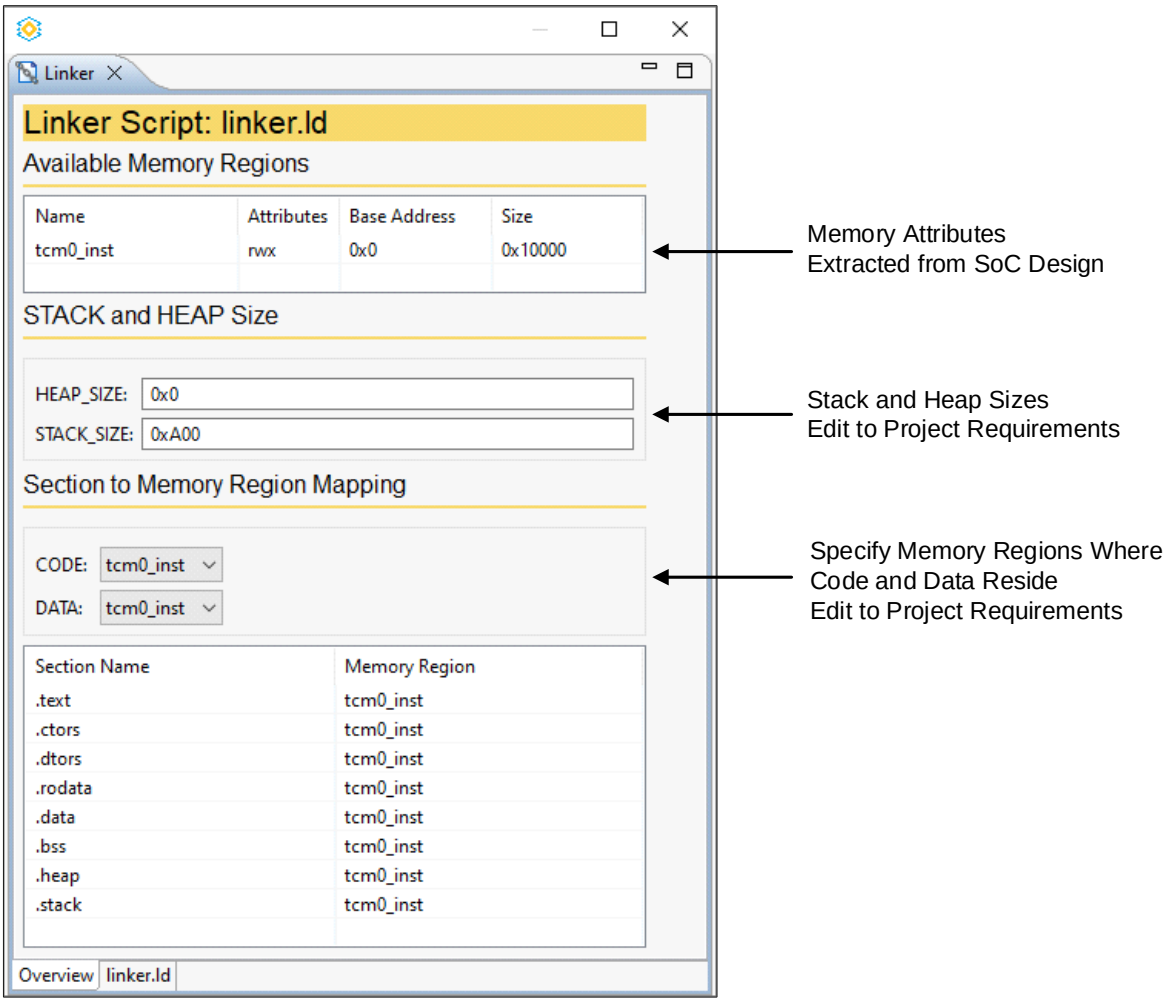

Figure 4.9. Lattice Propel SDK Linker Configuration GUI

The GUI displays the memory regions that are available in the SoC design which the project is based. Modify stack and heap sizes by changing the values in the text boxes. For SoC designs that have multiple memory regions, you can partition the code and data into separate memory regions using the GUI.

You can only make limited changes to the linking process using the Linker Configuration GUI. For more complex designs requiring more customizations, edit the text of the default linker script directly.

4.4.1.2. Text Based Linker Script Customization

To open the text based linker script editor, double click the linker.ld file in the src directory of your Lattice C/C++ project and click the linker.ld tab at the bottom of the pane as shown in Figure 4.10.

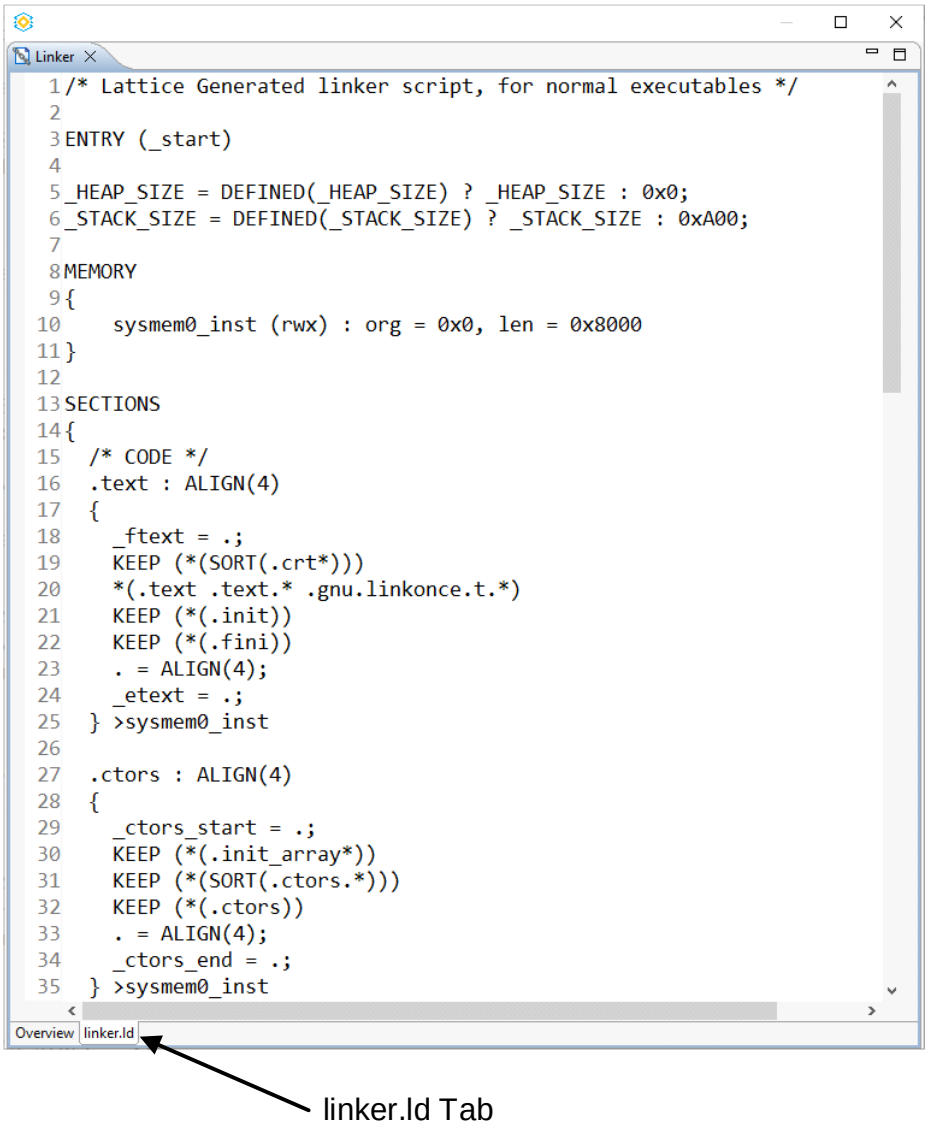

Figure 4.10. Lattice Propel SDK Linker Script Text Editor

4.4.2. Lattice Propel SDK Default Linker Script

GNU compatible linker scripts are written in the Linker Command Language. The default linker script provided by the Lattice Propel software consists of three commands in the following sequence:

1. ENTRY: Identifies the entry point, the first instruction to be executed in the program.
2. MEMORY: Specifies the different memory regions visible to the processor and the characteristics such as base address, size, and accessibility (read, write, execute).
3. SECTIONS: Specifies how the object file input sections are mapped to the ELF output sections.

Figure 4.11 shows the beginning of an autogenerated linker script for a system with two separate memories: one for storing instructions and one for storing data.

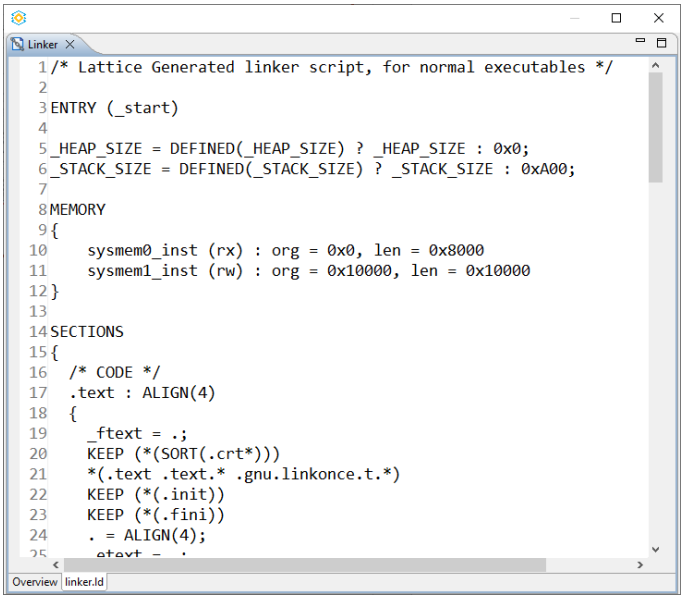

Figure 4.11. Autogenerated Linker Script for SoC System with Two Memory Regions

4.4.2.1. Linker Script ENTRY Command

The ENTRY command in the GNU Linker Command Language defines the first executable instruction in the output binary.

The ENTRY command takes a symbol name as an argument. In the Lattice BSPs for the bare metal and FreeRTOS execution environments, the startup code uses the symbol, "_start", to denote the first instruction. For example, Figure 4.12 shows the beginning of "crt0.S" for RISC-V MC where the first instruction, a jump, is assigned the "_start" symbol.

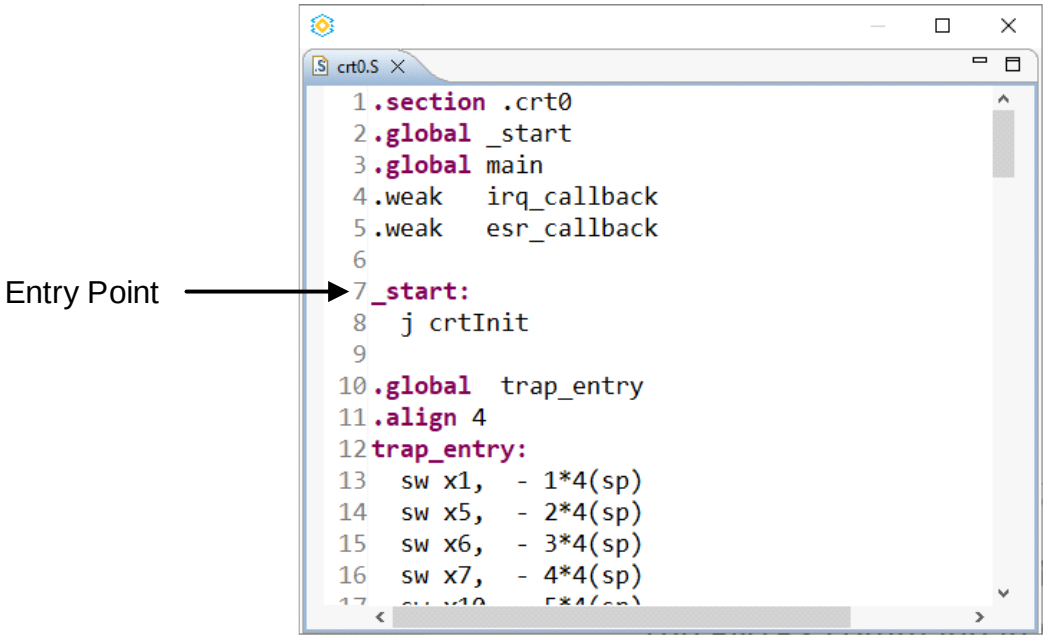

Figure 4.12. _start Entry Point in RISC-V MC BSP

4.4.2.2. Linker Script MEMORY Command

The MEMORY command in the GNU Linker Command Language informs the linker the memories available, memory locations in address space, memory sizes, and the operations the memories can support (for example read, write, or execute). The command also associates each memory block with a name that is used internally by the linker.

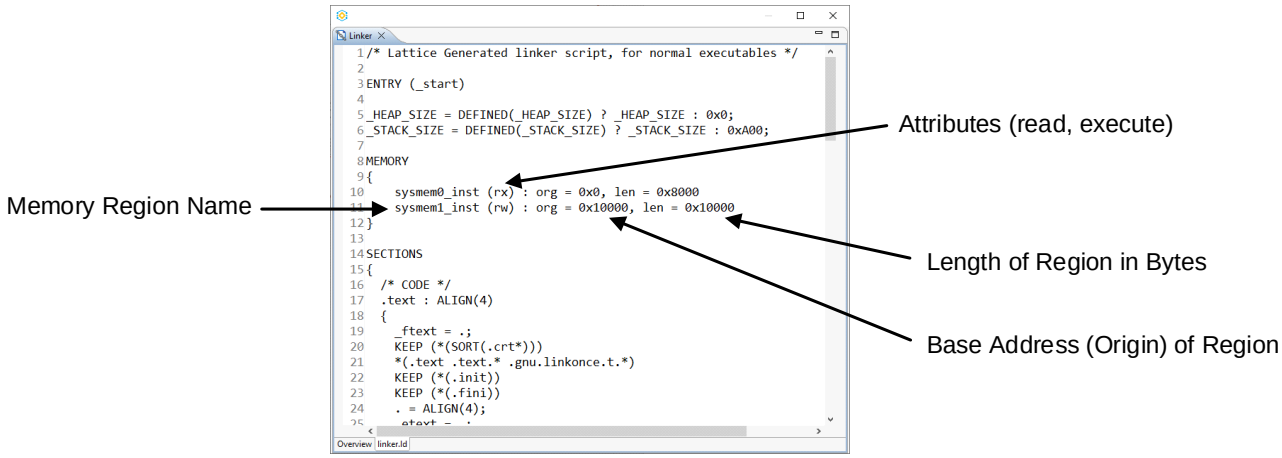

Figure 4.13. MEMORY Command Defining Two Memory Regions

In Figure 4.13, two memory regions are defined. The first, sysmem0_inst, is the instruction memory and is configured as read only in the Lattice Propel Builder SoC project. The autogenerated attribute field consists of only 'r' and 'x' for read and execute, and omits 'w' for write. Similarly, the sysmem1_inst memory is only connected to the data bus in the SoC design and the attributes are 'r' for read and 'w' for write while 'x' for execute is omitted.

Note: The Lattice Propel SDK debugger requires instruction memory to be read-writable via the RISC-V data bus. The debugger is not able to run on a Lattice SoC configured in Figure 4.13 because the program memory is not writable.

4.4.2.3. Linker Script SECTIONS Command

The SECTIONS command specifies how the parts (for example instructions, constants, variables, stack, and so on) of a program are arranged in memory.

The inputs to the linking process are a collection of object files. These files are a combination of pre-compiled libraries and high-level source modules of the program after being processed by the compiler and assembler.

The input object files are partitioned into sections. Some of the common sections are as follows:

- .text: program instructions
- .rodata: read only data (for example constant strings)
- .data: initialized global variables
- .bss: uninitialized global variables

The SECTIONS command defines the output sections that appears in the output *.elf file. The command also specifies the mapping of the input sections in the object files to the output sections.

The linker script that is automatically generated by the Lattice Propel SDK defines the following output sections:

- .text: program instructions
- .ctors/.dtors: C++ global constructor/destructor tables
- .rodata: read only data
- .data: initialized global variables
- .bss: uninitialized global variables
- .heap: pool of memory for dynamic allocation (for example malloc)
- .stack: program stack

The sections of the input object files and the file names (.text, .rodata, etc.) are defined by the GNU compiler and assembly tools. The output sections typically have the same names although this is not a requirement for the linker utility.

You can place the same type of program information in different sections. For critical code and non-critical code that are in the .text input sections, you can also place critical code in a cacheable memory region and place non-critical code using a different name in a non-cacheable memory region.

4.4.2.4. .text Output Section

Figure 4.14 shows the definition for the .text output section.

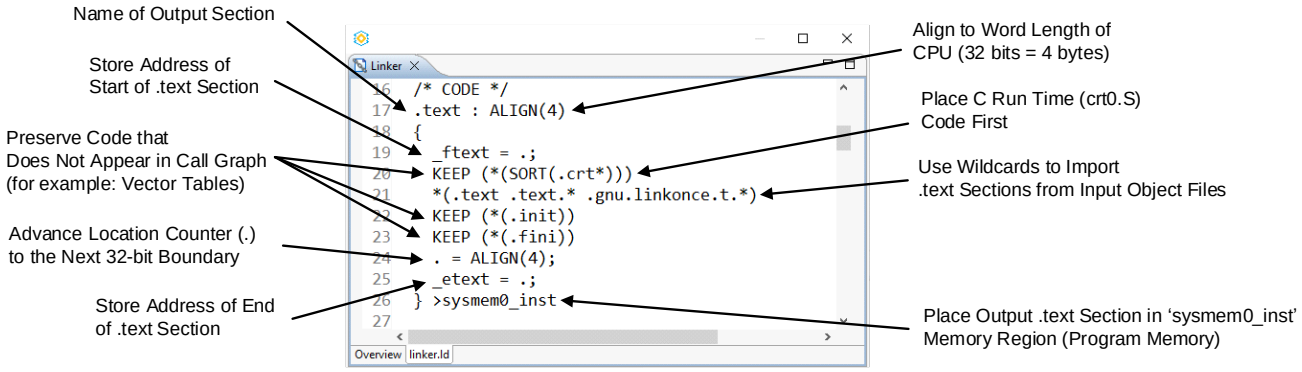

Figure 4.14. Autogenerated Linker Script .text Section

The output section declaration starts with the name of the section, followed by a colon and a call to the ALIGN function. ALIGN(4) forces the location counter to the next four byte or 32-bit boundary. This ensures that the instructions in the .text section are properly aligned for execution by the RISC-V processor.

The linker's location counter tracks the next location for code or data in memory. The period character, '.', is a built-in variable in the GNU Linker Command Language that provides access to the location counter.

The value of the location counter can be read into symbols. In the .text declaration, the '_ftext' and '_etext' symbols are assigned at the beginning and ending locations by reading the location counter before and after the input text sections are imported.

The key lines of this output section definition are in lines 20-23. These lines use wildcards ('*') to select input object files and sections. The input section names that match the specifiers are placed in the output .text section in the order in which they match.

The first match pulls the ".crt0" section into the .text output section. This section is defined in the crt0.S BSP file. As this section holds the C runtime startup code, this section is placed at the beginning of the program.

The KEEP keyword prevents the linker from pruning code and data which the call graph algorithms may mistakenly classify as unused. For example, the ".crt0" section includes interrupt vector tables that are not called explicitly, where KEEP ensures the linker does not remove the tables from the final executable.

The next matches are the .text sections from all input object files, which are the bulk of the program.

C++ constructs such as inline functions and virtual tables may cause code and data to be duplicated across multiple object files. The compiler places these functions and data into sections with names containing the "linkonce" pattern. When the linker encounters an input section name containing "linkonce", the linker adds that section to the output section for only one time. This prevents unnecessary code and data replication.

The ".init" and ".fini" contain library functions that execute global constructors at startup and global destructors when the program exits. These functions are called implicitly. The KEEP keyword ensures the functions are not removed from the .text output section.

Writing to the location counter built-in variable, '.', advances the location of the next placement. Figure 4.14 shows an example where ALIGN(4) is assigned to '.'. This assigned variable advances the location counter to the next 32-bit boundary.

The final line of the .text output definition directs the linker to place the section in the 'sysmem0_inst' memory region. This region is accessible via the SoC instruction bus and the base address is 0x00000000, which is the reset vector of the CPU.

4.4.2.5. .ctor and .dtor Sections

The .ctor and .dtor sections are mainly associated with C++ global constructors and destructors. Figure 4.15 shows the .ctors and .dtors output section definitions from the autogenerated linker script.

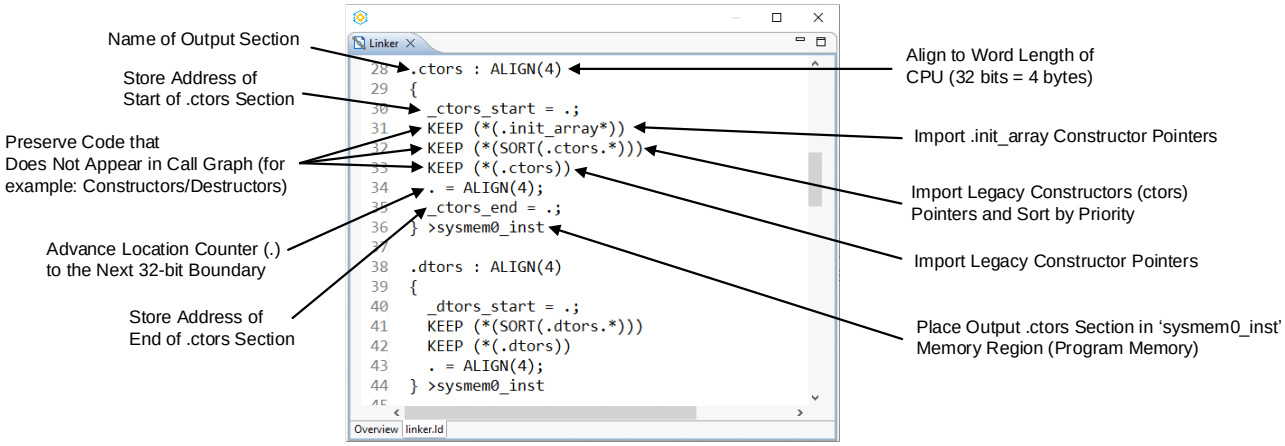

Figure 4.15. Definition for .ctors and .dtors Output Sections

The .ctors and .dtors sections are tables that hold pointers to the global constructors and destructors. As with other output sections, the .ctors and .dtors definitions start with the output section name followed by a call to the ALIGN function to ensure proper alignment to a 32-bit word boundary.

The linker script supports the following input section types that hold constructor and destructor pointer tables:

- The ".init_array" input sections are the modern format the GNU GCC uses to package global constructor and destructor pointer tables.
- The ".ctors" and ".dtors" input sections are included for backwards compatibility with the legacy code.

Although the constructor and destructor pointer tables are not executable code, the tables are not data and are closely related to the program. The .ctors and .dtors output sections are placed in the sysmem0_inst memory region with the .text section.

4.4.2.6. .rodata Section

The .rodata output section holds read only constant data. Figure 4.16 shows the output section definition for .rodata in the autogenerated linker script.

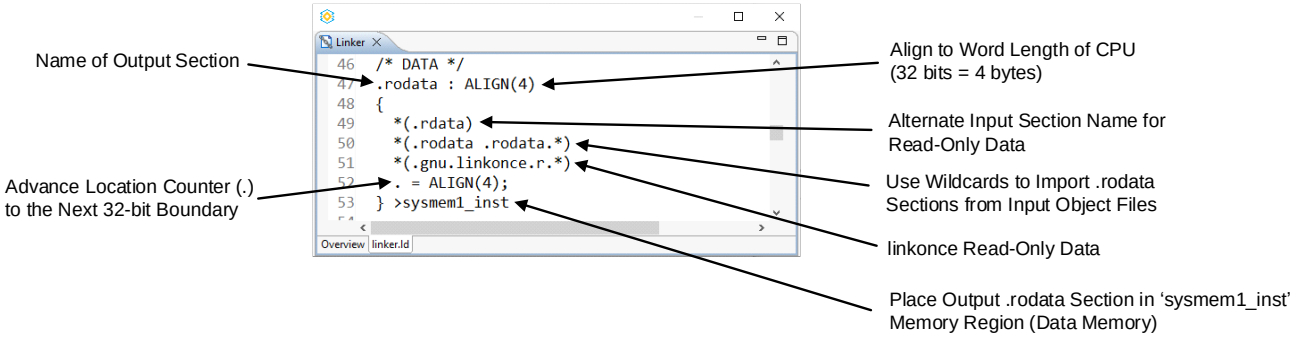

Figure 4.16. Definition for .rodata Output Section

The output section definition specifiers import ".rodata" input sections and input sections with an alternate name, ".rdata". The specifiers also match the name, ".gnu.linkonce.r", which can result from the inclusion of C++ constructs such as inline functions.

The final line of the .rodata output section definition places the section in the second memory region, sysmem1_inst, which is allocated for data.

4.4.2.7. .data Section

Figure 4.17 shows the definition for the ".data" output section.

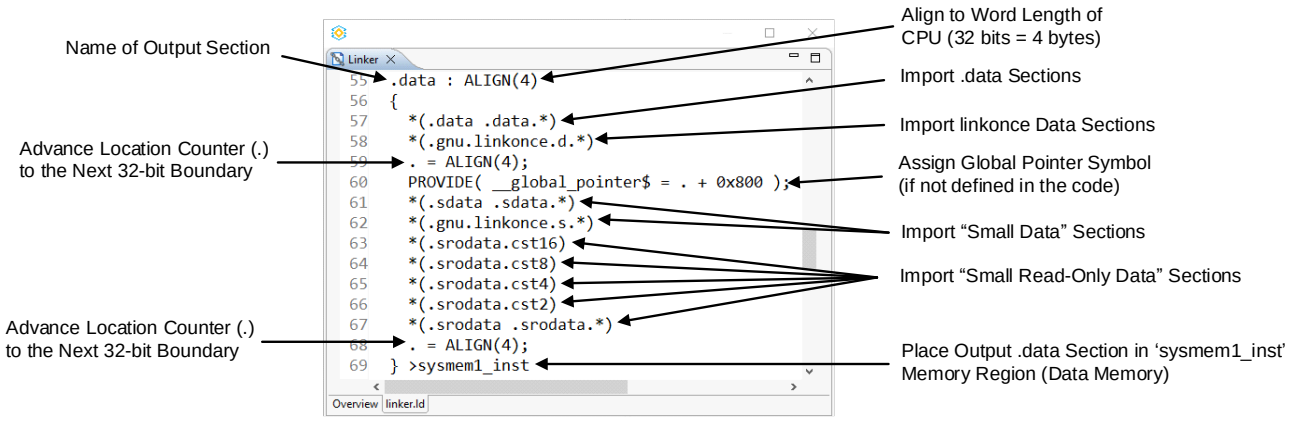

Figure 4.17. Definition for .data Output Section

The .data output section keeps global, initialized variables, particularly smaller variables, close to the address pointed at by the "global pointer" which is the "gp" register in the register file. This improves performance and code size via linker relaxation (see the Linker Relaxation section).

Figure 4.18 shows the encodings for the RISC-V load and store instructions, where a RISC-V core reads data from and writes data to the memory.

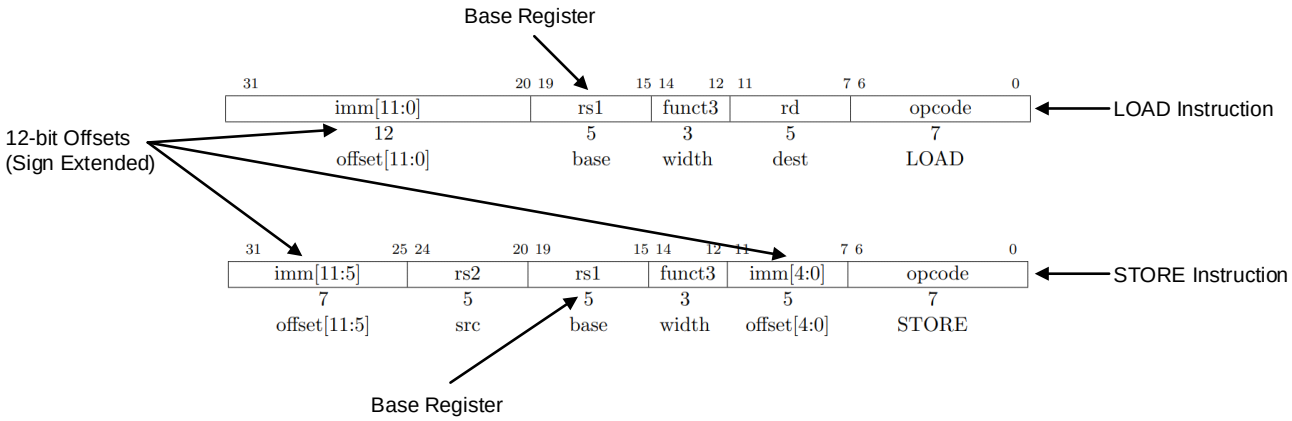

Figure 4.18. RISC-V Load and Store Instruction Encodings

The load and store instructions add a base address to a signed, 12-bit offset to specify the target address. The "rs1" field in the instruction indexes a register that holds the base address, while the immediate field encodes the offset. A naïve memory access requires two instructions: one to pre-load the base address into a register and a second to execute the load or store operation.

However, if a register already contains a value within the range of a signed, 12-bit immediate from the target address, the register pre-load is not required. The linker script places the smaller variables close to the global pointer register to ensure the memory access is within the 12-bit window of a known register value.

The output section definition imports the .data sections and linkonce data sections from the input object files. Depending on compiler settings, these sections are the larger data structures and arrays that are accessed via pointer. Linker relaxation is not able to enhance the performance for these sections.

After forcing alignment to the next 32-bit boundary, the script creates a __global_pointer$ symbol that points to the value of the location counter plus an offset of 0x800. The __global_pointer$ symbol is used in the startup code to initialize the "gp" register, a register in the register file.

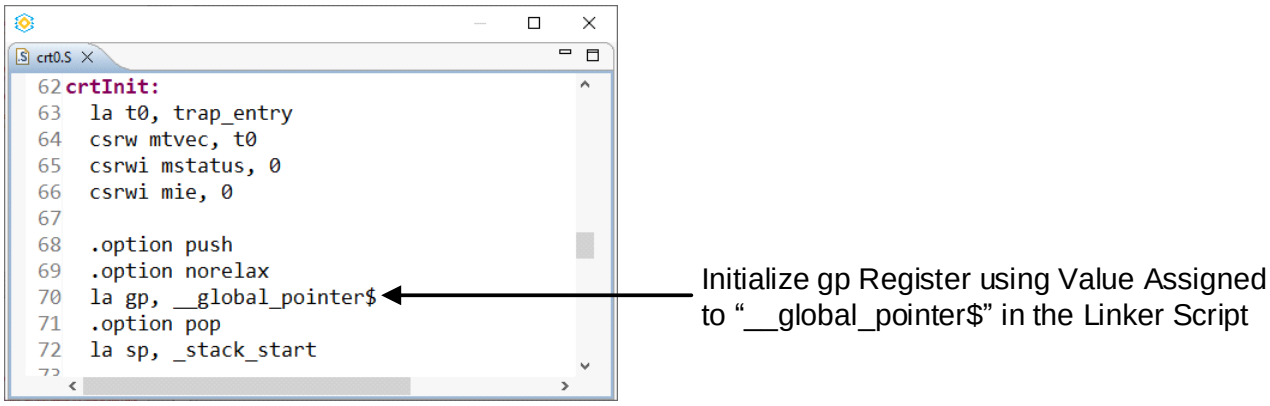

Figure 4.19. Startup Code Initializing gp (Global Pointer Register)

The "PROVIDE" keyword assigns the __global_pointer$ symbol if the symbol is not already given a value in the code.

After the assignment of the global pointer, the script imports the ".sdata" and ".srodata" sections which are the "small data" sections that the compiler creates based on a configurable threshold, as shown in Figure 4.20.

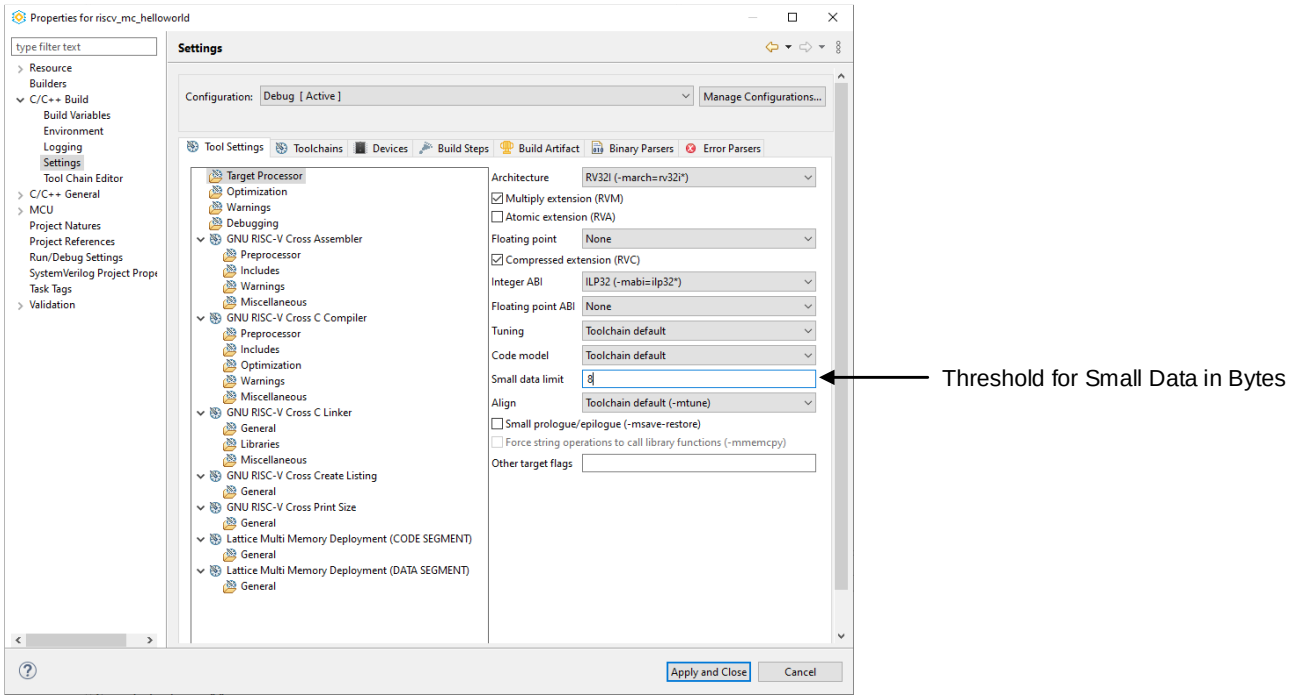

Figure 4.20. Configuration of the Small Data Limit in Lattice Propel SDK

The final line of the .data output section places the section in the sysmem1_inst memory region which is allocated for data storage.

4.4.2.8. .bss Section

The definition of the ".bss" output section is shown in Figure 4.21.

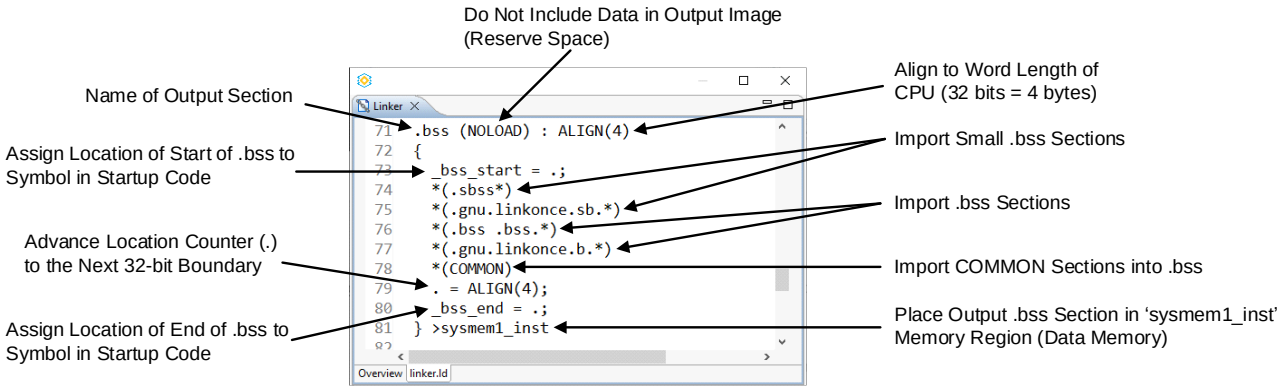

Figure 4.21. Definition for .bss Output Section

The .bss ("Block Start by Symbol") section reserves space for global and static variables that are not initialized by the source code.

In the C/C++ standards, such variables are required to be zeroed out. Instead of storing a bunch of zeros in the output ELF, the C startup code performs the zeroing of the .bss section at program launch.

The NOLOAD keyword instructs the linker to exclude the actual BSS data in the output image to avoid increasing the output binary image unnecessarily. For the C runtime code to initialize the memory associated with .bss, the linker points the C runtime code to the .bss location in the memory. The _bss_start symbol captures the value of the location counter at the beginning of the .bss section and the _bss_end symbol captures the value at the end of .bss section. C startup code starts and stops zero initializing the memory based on the _bss_start and _bss_end symbols.

Figure 4.22 shows the .bss initialization code in the crt0.S source file. The C run time implements a loop that uses the _bss_start and _bss_end symbols as initial and terminal values, respectively.

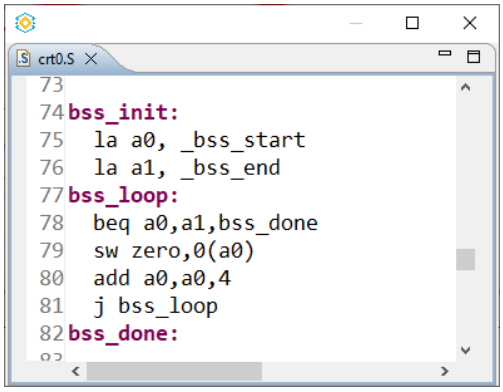

Figure 4.22. .bss Initialization Loop in crt0.S Source File

As .bss section contains variables that are placed in the memory region reserved for data, sysmem1_inst.

4.4.2.9. .heap Section

Figure 4.23 shows the output section definition for the ".heap" section.

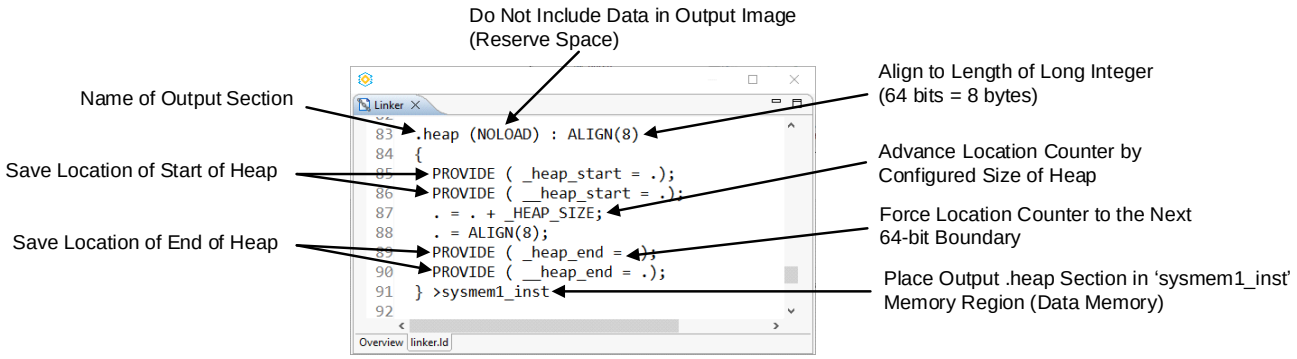

Figure 4.23. Definition for .heap Output Section

The heap is a pool of memory reserved for dynamic allocation. No variables are assigned to the heap at link time and there is nothing to initialize. The NOLOAD keyword instructs the linker to exclude initial values for heap in the output executable image.

To reserve the required space, the output section advances the location counter, '.', by the heap size as defined in the "_HEAP_SIZE" symbol near the top of the linker script.

As shown in Figure 4.24, _HEAP_SIZE and _STACK_SIZE symbols are defined using tertiary operators. If the symbols are already defined in the program, the existing value is used. If the symbols are not defined, the constant value from the Linker "Overview" GUI tab is used.

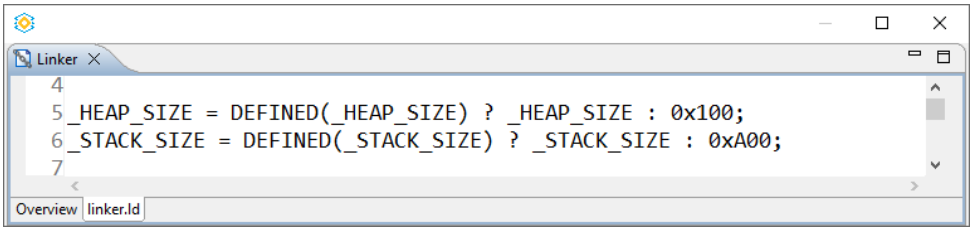

Figure 4.24. Defining Heap and Stack Size Symbol

The .heap output section definition also assigns multiple symbols to record the start and stop addresses of the heap, which is required for dynamic memory libraries such as malloc.

The last line of the .heap output section definition places the section in sysmem1_inst, the data memory region.

4.4.2.10. .stack Section

The definition of the ".stack" output section is shown in Figure 4.25.

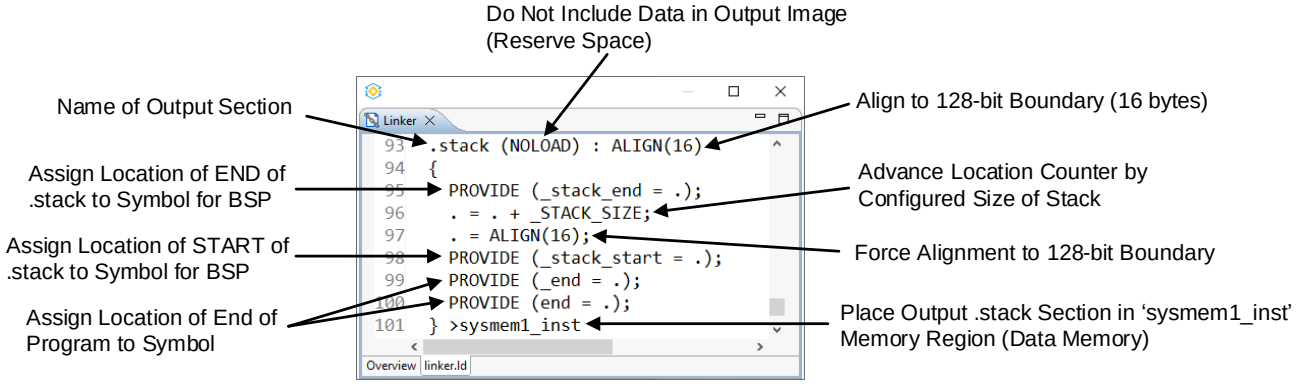

Figure 4.25. Definition for .stack Output Section

The .stack output section is aligned with a 128-bit boundary which is required by the RISC-V calling convention and is for the 'Q' extension which adds quad precision floating point. Note that 'Q' extension is not supported for Lattice RISC-V.

The .stack output section definition reserves space for stack. This section uses the NOLOAD keyword to inform the linker not to add the contents of the stack to the output ELF.

The linker script reserves memory for the stack by advancing the location counter by the amount of the _STACK_SIZE symbol. Like the _HEAP_SIZE symbol, _STACK_SIZE is defined near the beginning of the script using a tertiary expression that selects the value from the GUI if the symbol is not defined elsewhere.

The .stack output section definition also defines _stack_start and _stack_end symbols for use in the BSP. By convention, the RISC-V stack grows downward so the _stack_end symbol is assigned at the beginning of the section definition and _stack_start is defined at the end.

The stack is placed in the data memory region, sysmem1_inst.

4.5. Interrupts

Interrupts are fundamental to embedded systems. An interrupt is essentially an unscheduled function call and is not synchronized to normal program flow. Using interrupts, an embedded processor responds to time critical events quicker and more deterministically than a loop that periodically polls for the condition. Most real time operating systems (RTOS) use preemptive multitasking and interrupts are integral to preemption.

Interrupt hardware and software vary across different CPU architectures and execution environments (for example bare metal, schedulers, and real time operating systems). Understanding the CPU, interrupt controller, and BSP interrupt handler is key to effective and efficient development.

4.5.1. RISC-V Interrupt Architecture

The following is a brief overview of the RISC-V interrupt architecture. For detailed information on RISC-V interrupts, refer to the *RISC-V ISA specification, Volume II: RISC-V Privileged Architectures* in the RISC-V Specifications web page.

The RISC-V ISA groups interrupts and exceptions into a general category named "traps". Certain RISC-V architectural features, including some Control and Status Registers (CSRs), are shared by interrupts and exceptions.

While interrupts are caused by events that are asynchronous to program execution, exceptions are synchronous. The CPU generates an exception as a direct result of executing a given instruction. This exception can be a result of an error such as a misaligned address or an illegal instruction. An exception may also be forced by design, such as when the ECALL instruction is executed.

When a trap, either an interrupt or an exception occurs, RISC-V transfers control to a trap handler firmware routine. The trap handler is part of the execution environment. For embedded systems, the trap handler usually comes with the board support package (BSP). The RISC-V interrupt logic obtains the memory address of the trap handler via a register that the execution environment loads during program startup.

Trap handler determines the cause of the trap and calls the appropriate service routine to respond to the interrupt or exception event. The RISC-V ISA defines the following methods to respond to the events:

- Directed mode causes the program counter to be loaded with the same address regardless of the cause of the trap.
- Vectored mode is for exceptions but uses a jump table for the different interrupt sources which can improve interrupt latency.

The RISC-V ISA comprehends three general interrupt sources: timer, software, and external.

- Timer interrupts are generated by a CPU's integrated timer and are used for time-slicing in multi-tasking operating systems.
- Software interrupts are intended for inter-processor messaging, which allow one core or hart to interrupt another (or itself).
- External interrupts come from external sources, for example, the CPU's peripherals.

For RISC-V implementations that support multiple privilege levels, interrupt sources are split into privilege mode-specific versions. For example, a RISC-V core that supports Machine and Supervisor privilege modes can have as many as 6 interrupt inputs: machine timer, machine software, machine external, supervisor timer, supervisor software, and supervisor external.

4.5.1.1. RISC-V Trap Handling CSRs

Table 4.3 list the key RISC-V control and status registers (CSRs) related to traps. These CSRs are for machine mode or supervisor level equivalents.

Table 4.3. Key RISC-V Trap Handling CSRs

CSR Name	Description
mtvec	Contains the vector base address for the trap handler. Controls the mode: directed or vectored.
mcause	Indicates the cause of the trap. Bit fields that indicate whether the trap is an interrupt or an exception and the type of interrupt or exception.
mstatus	Contains status and control bits including global interrupt enables and the status of the CPU core prior to the trap.
mepc	The value of the program counter is copied to the mepc register when a trap occurs. The contents of the mepc register are written back to the program counter when returning from the trap.
mscratch	Scratchpad register. Useful for operating systems in storing an index into a task context table.

For complete details about the RISC-V CSRs, refer to the RISC-V ISA specification, *Volume II: RISC-V Privileged Architectures* in the RISC-V Specifications web page.

4.5.2. Lattice RISC-V Interrupt Controller Hardware

An embedded system may need to support multiple external interrupts. As a RISC-V core supports only one external interrupt input per privilege level, interrupt controller hardware must be added to RISC-V based designs to aggregate external interrupts into a single signal.

The external interrupt controller for the Lattice RISC-V CPUs varies depending on the model, as shown in Table 4.4.

Table 4.4. Lattice RISC-V External Interrupt Controllers by Model

RISC-V Model	External Interrupt Controller
RX	Platform Level Interrupt Controller (PLIC)
MC	Programmable Interrupt Controller (PIC)
SM	Programmable Interrupt Controller (PIC)

4.5.2.1. Programmable Interrupt Controller (PIC)

The Programmable Interrupt Controller (PIC) is a proprietary Lattice IP. The PIC is used on the RISC-V MC and SM processors. Figure 4.26 shows a block diagram of the PIC.

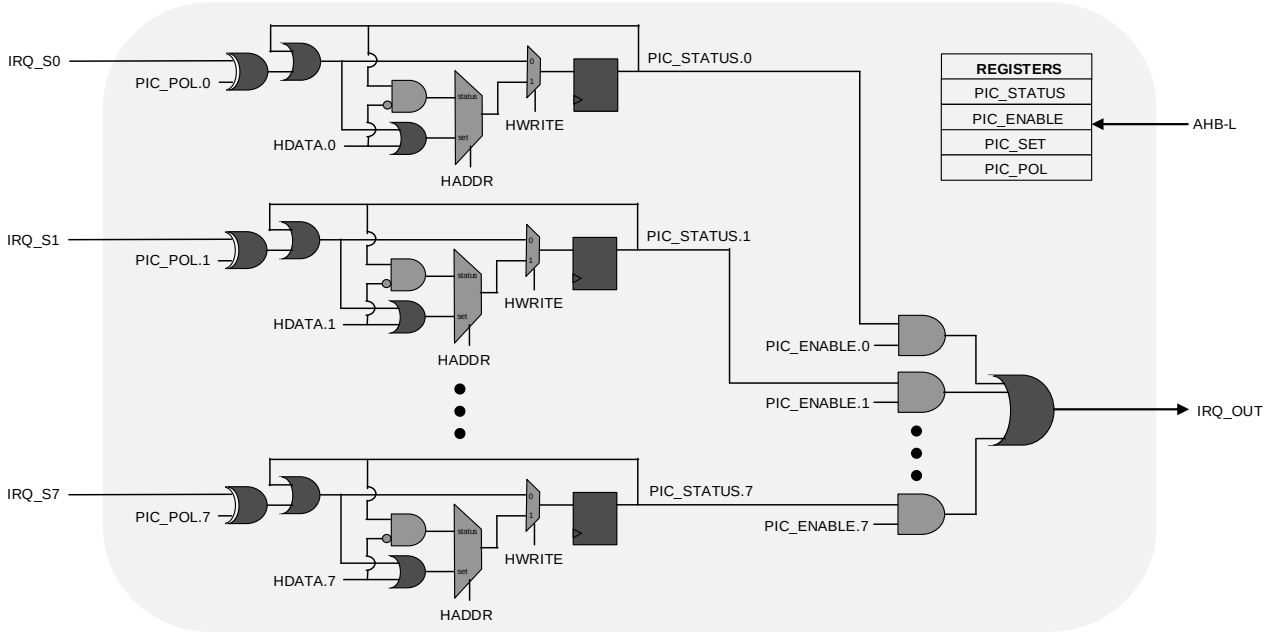

Figure 4.26. Programmable Interrupt Controller (PIC)

When enabled, the PIC supports between two and eight external interrupt inputs. The PIC has an AHB-Lite control interface consisting of four registers: PIC_STATUS, PIC_ENABLE, PIC_SET, and PIC_POL (polarity).

The interrupt channels for PIC are not complicated. The flip-flop holds the current pending interrupt state of the channel and the state can be read via the PIC_STATUS register. The input is inverted or not inverted based on the value of the respective bit in the PIC_POL register. The pending state can be cleared by an AHBL write to the PIC_STATUS register address with a value of one in the respective data bit. The firmware can force a channel into the pending state by writing a one to the corresponding bit of the PIC_SET register. The bits in the PIC_ENABLE register control whether a pending interrupt on the respective channel causes the IRQ output to be asserted. All channels are identical and the channels are independent from one another.

In the RISC-V MC and SM processors, IRQ_OUT output from the PIC drives the external machine interrupt input to the CPU core.

4.5.2.2. Platform Level Interrupt Controller (PLIC)

The Platform Level Interrupt Controller (PLIC) is defined by the RISC-V Task Group. The Lattice RISC-V RX processor uses the PLIC as the interrupt controller. Figure 4.27 shows the block diagram of the PLIC from the specification.

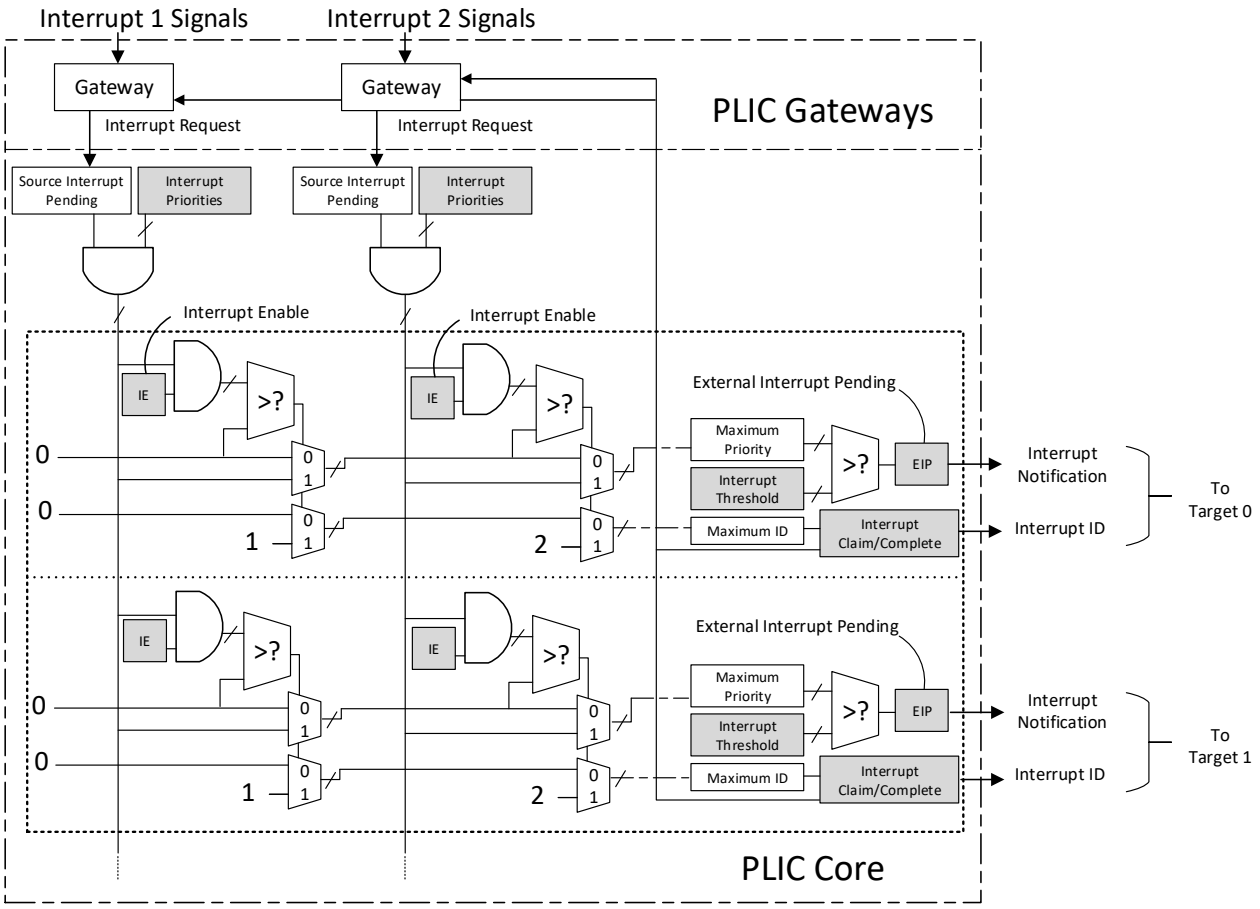

Figure 4.27. Platform Level Interrupt Controller (PLIC) Block Diagram

PLIC has up to 1024 interrupt sources. The Lattice RISC-V RX implementation supports only up to 32 interrupt sources. The first interrupt, interrupt zero (at the left of Figure 4.27), is reserved and not functional.

The PLIC has two outputs, one for machine privilege level external interrupts and one for supervisor level external interrupts.

The gateways at the top of the block diagram recognize and queue up interrupts. The PLIC gateways in the RISC-V RX support only level sensitive, high true, interrupts. When a gateway detects an interrupt assertion, the gateway sets the corresponding interrupt pending bit, IP.

The interrupt priority of an interrupt is a programmable register. When the interrupt pending bit, IP, is set, the interrupt priority bit field is forwarded to the PLIC core. If the interrupt enable bit, IE, is set for machine or supervisor outputs, the interrupt priority continues to be forwarded towards the respective output.

If a second, lower numbered interrupt is also pending at the same time, the two priorities are compared. The larger of the two continues to be forwarded, along with an interrupt ID field. If both interrupts have the same priorities, the interrupt with the lower interrupt ID value continues to be forwarded.

The winning priority is then compared with an interrupt threshold that is programmable on a per output basis. If the priority of the interrupt is greater than the threshold, the external interrupt pending bit, EIP, is set and the interrupt is asserted to the appropriate privilege level of the CPU.

The PLIC employs a Claim/Complete protocol to service and retire interrupts. To initiate a claim, the firmware performs a read of the claim/complete register of the PLIC. The PLIC returns the ID of the highest priority interrupt that is pending. If

no interrupt is pending, the PLIC returns a zero, the ID of the reserved, non-functional interrupt input. During the claim, the PLIC also clears the claimed interrupt's interrupt pending bit, IP.

Although the claimed interrupt's IP bit is cleared, the interrupt cannot occur again until the PLIC receives a completion from the firmware. The firmware performs a completion by writing the interrupt ID back to the claim/complete register. The completion is the final step in handling the interrupt.

For complete technical details on the PLIC, see the specification in the RISC-V Platform-Level Interrupt Controller Specification web page and the RISC-V RX CPU IP Core User Guide (FPGA-IPUG-02241).

4.5.3. Lattice RISC-V Trap Handlers

The trap handling firmware that is part of the Lattice RISC-V BSPs is written in assembly. The trap handlers for RISC-V MC/SM and RISC-V RX are different, primarily because of the different execution environments for the processors: bare metal for RISC-V MC/SM and FreeRTOS for RISC-V RX.

Both trap handler routines perform the same general operations as follows:

1. Save the CPUs state, such as register file and key CSRs.
2. Determine the source of the trap: interrupt or exception.
3. Call the appropriate handler routine.
4. Restore the CPUs state. The state may be different following a task switch.
5. Execute the appropriate return instruction to resume the program.

4.5.3.1. Bare Metal Trap Handler

Figure 4.28 shows the bare metal trap handler that is part of the RISC-V MC/SM BSP.

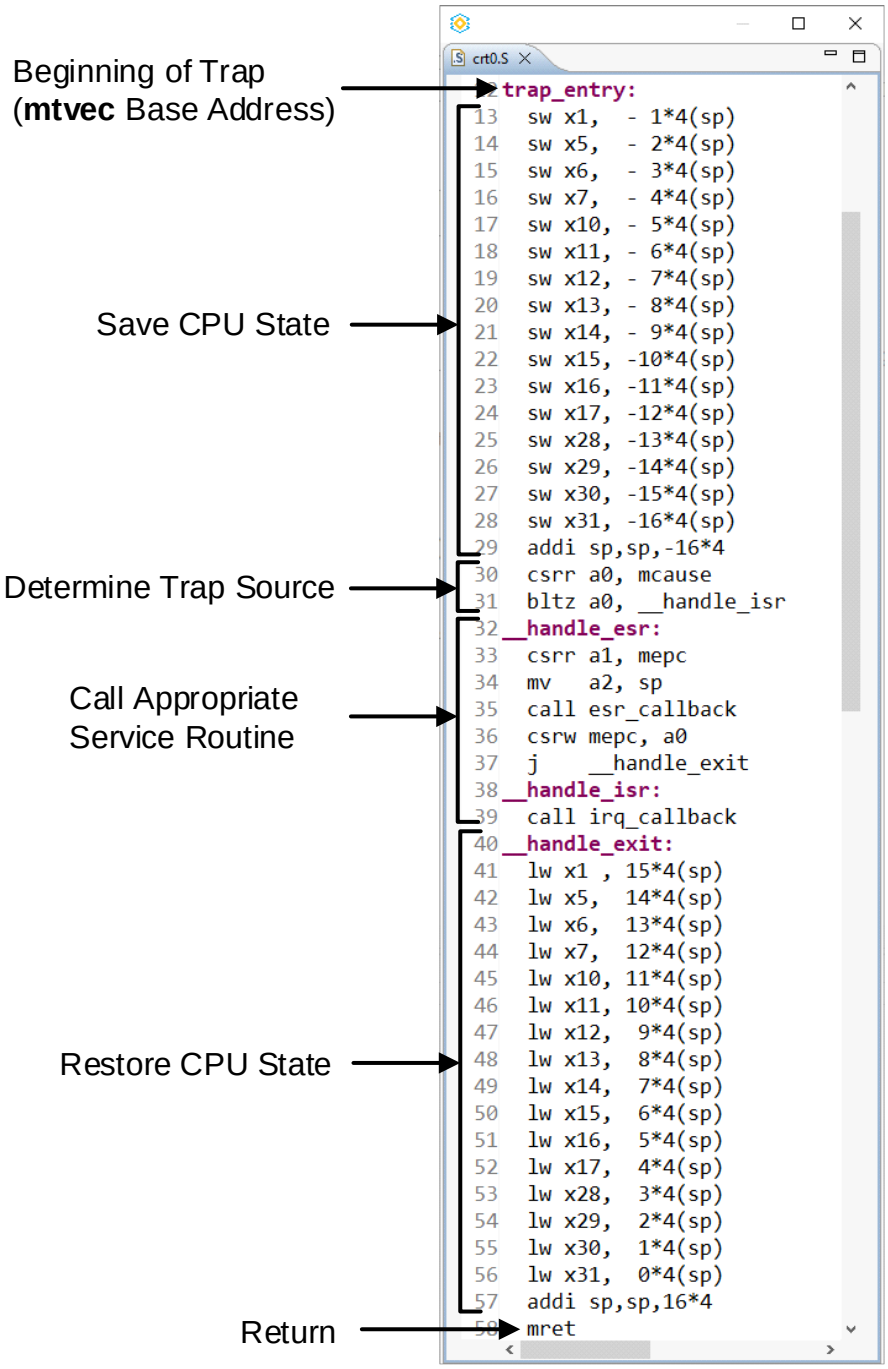

Figure 4.28. Bare Metal Trap Handler for RISC-V MC and SM

The trap_entry symbol marks the beginning of the trap handler routine. The startup code uses this symbol to initialize the **mtvec** CSR. This assignment causes the CPU hardware to jump to the trap handler when an interrupt or exception occurs.

RISC-V hardware does not automatically push the CPU state to the stack. The firmware stores the state of the machine during an interrupt by using a series of `sw` (store word) instructions and the `addi` operation. The sixteen registers stored are used by the bare metal execution environment. The integer addition operation updates the stack pointer by subtracting sixty-four bytes (16 words), the amount that the stack has grown downward in memory.

After the CPU state is saved, the trap handler retrieves the value in the **mcause** CSR and loads the value into a register, a0, for comparison. The most significant bit of **mcause** indicates whether the trap is caused by an interrupt or an exception. As the most significant bit is also the sign bit, a branch if less than zero (`bltz`) instruction can be used to test the value of the bit and branch to the appropriate routine to handle the trap.

For an interrupt, the C function, **irq_callback()** is called.

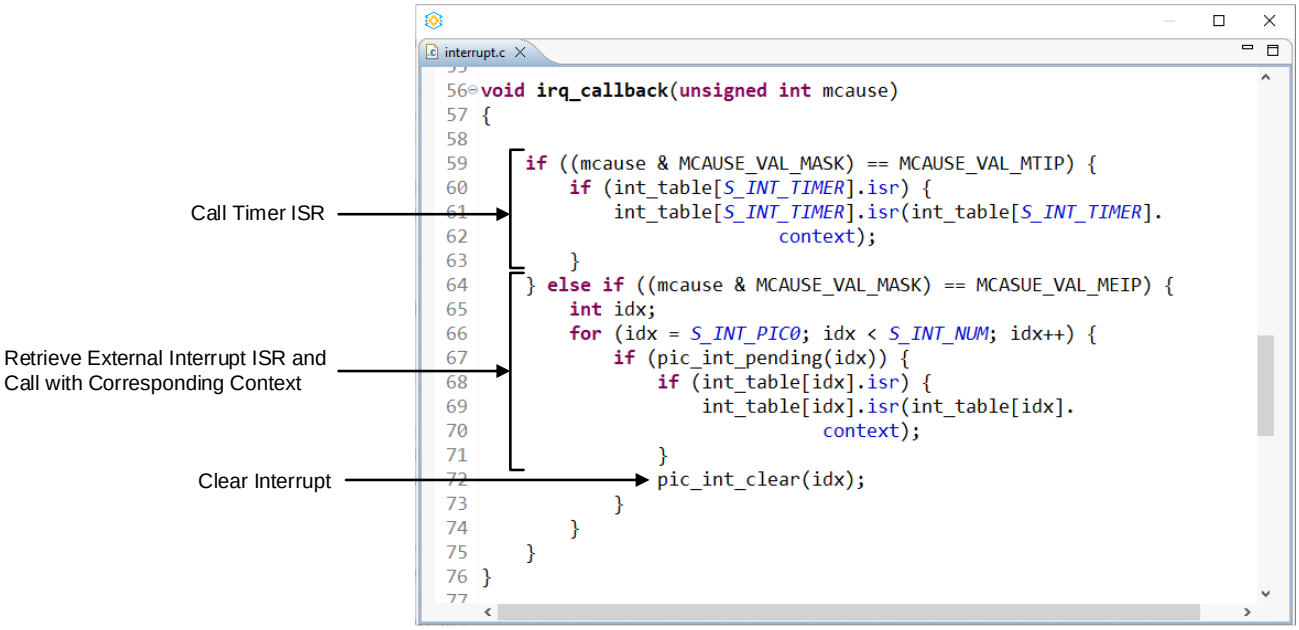

Figure 4.29. Bare Metal BSP irq_callback( )

The **irq_callback()** function relies on a global, array-based table named **int_table**. The table stores the following pointers:

- Function pointers to the interrupt service routines for the timer and the external interrupts.
- A pointer to a context data structure for each interrupt input. The pointer is passed to the interrupt service routine when the routine is called.

The context data allows the service routine function to avoid hard coding key parameters such as the base address of an associated peripheral. It also permits a single function be used as the interrupt service routine for multiple instances of the same type of peripheral.

The initialization routine of the timer configures the table entry for the timer ISR. Table entries for the external interrupts are initialized by API calls to the **pic_isr_register()** function.

When called, the **irq_callback()** function determines the source of the interrupt and looks up the corresponding table entry in **int_table**. The function calls the ISR function using the context data. Once the ISR finishes processing the interrupt and returns, the interrupt is cleared in the PIC (if the interrupt was an external interrupt) and program flow returns to the trap handler where the CPU state is restored and normal program execution resumes.

4.5.3.2. Lattice FreeRTOS Trap Handler

Lattice's trap handler for the FreeRTOS execution environment is split between an assembly routine and a C function. Figure 4.30 shows the trap handler assembly code for the Lattice implementation of FreeRTOS. The code is executed when an exception occurs. The code is also executed when an interrupt occurs, and the core is set for Direct mode interrupts.

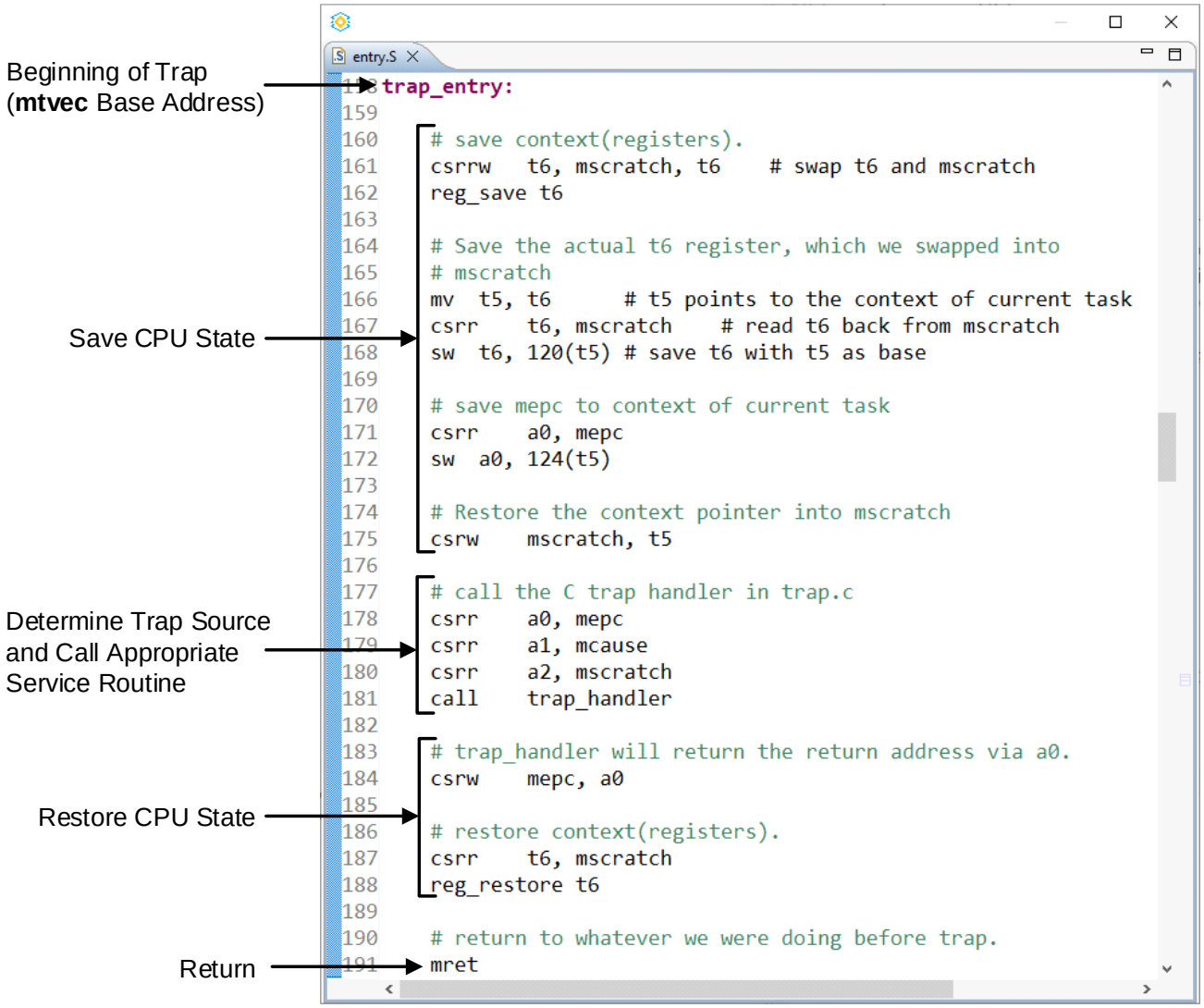

Figure 4.30. Trap Handler (Direct Mode) for Lattice Implementation of FreeRTOS

FreeRTOS declares and maintains a table of state contexts. FreeRTOS uses the **mscratch** CSR to hold a pointer to the table entry for the currently running task.

The trap handler routine first retrieves the pointer to the current state context by swapping the contents of **mscratch** with the register, `t6`. The routine then saves the contents of the register file to that table entry using the reg_save assembly macro. After the other register values have been saved, the original value of `t6` is retrieved and stored. The address of the instruction that was interrupted is retrieved from the mepc CSR and is the final state information that is stored to the state context table.

Once the CPU state has been stored, the assembly routine calls the C language portion of the trap handler. The function is called with the values of the CSRs, **mepc**, **mcause**, and **mscratch** in the argument registers, `a0-a2`.

Figure 4.31 shows the C language portion of the trap handler.

```c
reg_t trap_handler(reg_t epc, reg_t cause, context_t *cxt)
{
    reg_t return_pc = epc;
    reg_t cause_code = cause & 0xfff;

    if (cause & 0x80000000) {
        /* Asynchronous trap - interrupt */
        // DEBUG("interrupt!, cause=%d\r\n", cause_code);
        isr_callback(cause_code);
    } else {
        /* Synchronous trap - exception */
        // DEBUG("exception!, cause=%d, epc=0x%08x\r\n", cause_code, epc);
        return_pc = esr_callback(cause_code, epc, cxt);
    }

    return return_pc;
}
```

Figure 4.31. C Language Portion of Trap Handler for FreeRTOS

The most significant bit of the **`mcause`** CSR indicates whether the trap is an interrupt or an exception. The **`trap_handler()`** function calls the BSP callback function according to the CSR bit. Figure 4.32 illustrates the program flows when handling software, timer, and external interrupts.

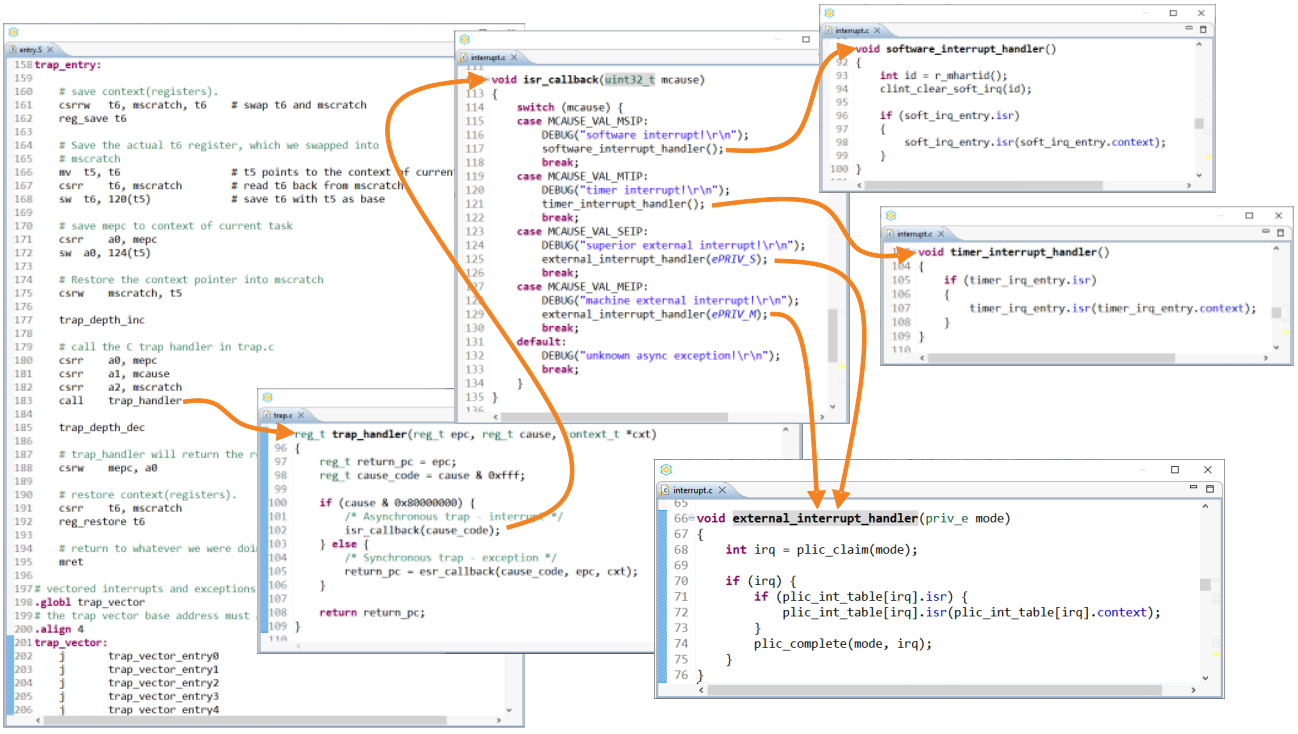

Figure 4.32. Program Flow for Handling Interrupts

Each type of interrupt has its own handler function. All handler functions use a data structure that holds a function pointer to the respective interrupt service routine and a pointer to a context data for the interrupt service routine to process the

interrupt. When an interrupt occurs, the interrupt service routine is called with the pointer to the context data as an argument.

As multiple external interrupts may occur, the data structures are organized as an array, **plic_int_table**. The entries in **plic_int_table** are ordered to match the corresponding interrupt IDs in the PLIC. When the interrupt handler performs a PLIC claim operation, the table index is the ID that the PLIC returns.

The external interrupt handler uses this index to retrieve the function pointer to the interrupt service routine and context data. Once it has called the interrupt service routine and the routine returns, the external interrupt handler completes the interrupt with the PLIC hardware and returns to the trap handler.

The trap handler then restores the CPU state. The restored state can be the original task or the next task in FreeRTOS's ready queue, if a task switch occurred during the interrupt.

4.5.4. Using the Lattice RISC-V BSP Interrupt Firmware

The differences between the programmable interrupt controller (PIC) of RISC-V MC and SM and the platform level interrupt controller (PLIC) of RISC-V RX include the driver firmware that is provided to initialize and control the hardware.

However, the BSP drivers for the PIC and PLIC utilize a table to support external interrupts. Each entry in these tables stores a pointer to the interrupt service routine function and a second pointer to context data for the interrupt service routine to process the interrupt.

Initialize these tables for the trap handling firmware in the BSP to call the appropriate interrupt service routine when an interrupt occurs. The PIC and PLIC drivers provide API calls to register external interrupts to the corresponding entries in the interrupt table. These calls also enable the interrupt in the interrupt controller hardware.

4.5.4.1. pic_isr_register()

The call signature for pic_isr_register() is:

```
unsigned char pic_isr_register(unsigned char src,
                               void (*isr) (void *),
                               void *context)
```

where,

src:	PIC input port number (interrupt number)
isr:	pointer to the ISR function
context:	void pointer to the associated context data structure

The `pic_isr_register()` function initializes the interrupt's entry in the `int_table` global array. The function also enables the interrupt in the PIC hardware.

The pointer to the context is passed as a void pointer to be generic. If you develop a driver for custom hardware, define a data structure to hold the information required by the ISR. Register pointers to instances of that data with BSP by casting the data as type void *.

4.5.4.2. plic_int_register()

The call signature for plic_int_register() is:

```
int plic_int_register(uint8_t src,
                      uint8_t priority,
                      uint8_t mode,
                      irq_handler isr,
                      void *context)
```

where,

src:	PLIC input port number (interrupt number)
priority:	PLIC priority of the interrupt
mode:	privilege level of the interrupt, supervisor or machine level
isr:	pointer to the ISR function

`context`: void pointer to the associated context data structure

The `plic_int_register()` function initializes the interrupt's entry in the plic_int_table. The function also initializes the PLIC hardware for the interrupt including the priority and enabling the interrupt at the correct privilege level.

The pointer to the context is passed as a void pointer to be generic. If you develop a driver for custom hardware, define a data structure to hold the information required by the ISR. Register pointers to instances of that data with BSP by casting the data as type void *.

5. RISC-V System Debugging

5.1. Using OpenOCD Debugger and Reveal Analyzer

The Lattice Propel SDK supports software debugging using OpenOCD and GNU GDB to debug software related issues. The Lattice Propel SDK User Guide (FPGA-UG-02195) provides information about using the OpenOCD debugger.

The Lattice Radiant software and the Lattice Diamond software provide the Reveal Analyzer for debugging Lattice FPGA designs. This debugging method monitors the FPGA logic and signals for finding hardware related issues. The Debugging with Reveal Usage Guidelines and Tips Application Note (FPGA-AN-02060) provides information about using the Reveal Analyzer.

Because problems encountered during FPGA embedded system development is difficult to identify whether it is software or hardware related, use the OpenOCD Debugger and Reveal tools to perform root cause investigation. An example is when the software code performs the correct sequences to setup the hardware but the expected behavior is not observed. Capture the hardware signals using the Reveal Analyzer to provide insight for the issues. Coupled with software debugger, software execution (using breakpoint) can be paused at the precise moment for the Reveal Analyzer to trigger on the desired conditions and capture the signals.

Figure 5.1 shows the general flow when debugging using the OpenOCD Debugger and the Reveal Analyzer.

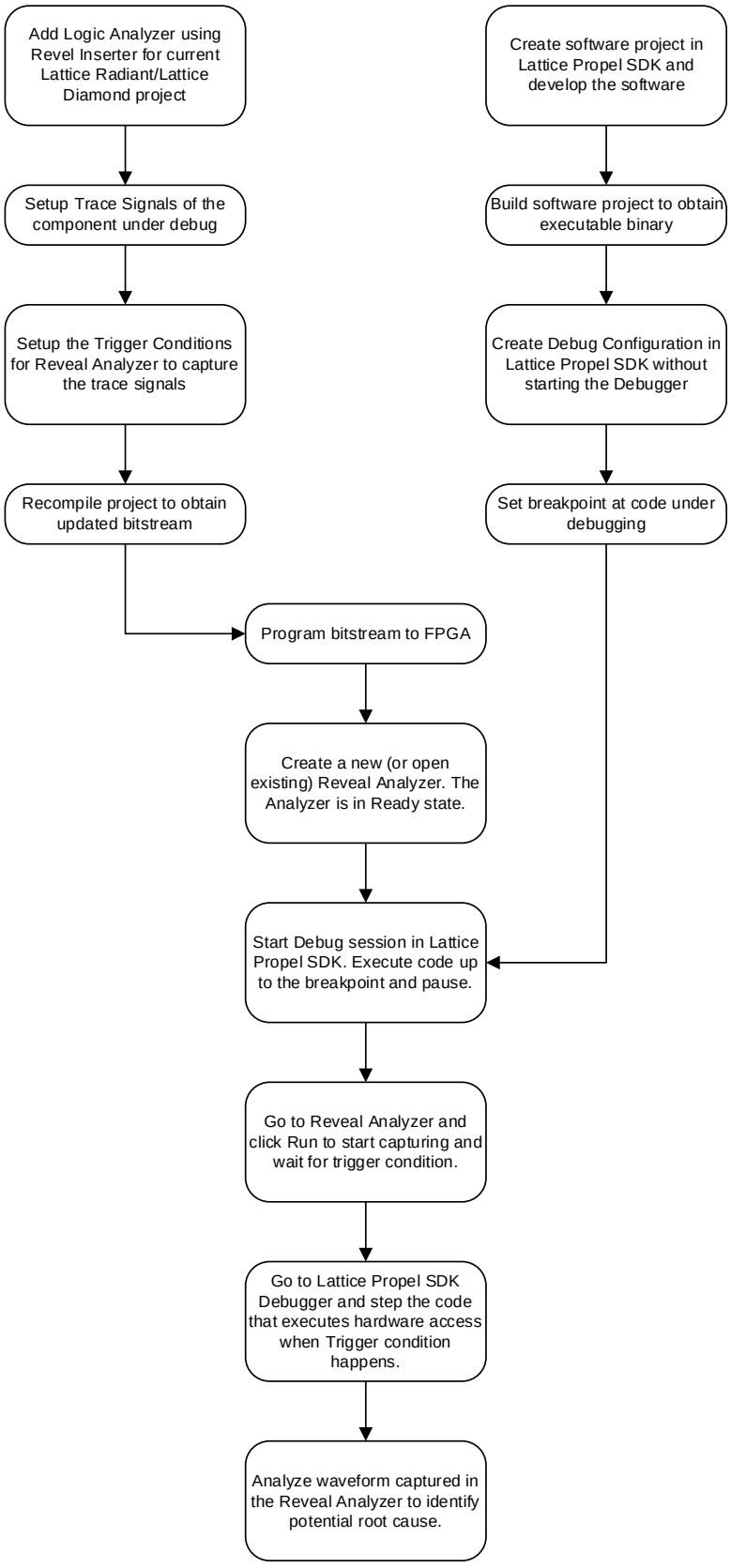

Figure 5.1. Debugging Flow Using the OpenOCD Debugger and the Reveal Analyzer

5.2. Using Breakpoints

Breakpoints are a fundamental tool in the debugging process. They permit the designer to halt the program execution when reaching a prespecified condition. The designer can then perform typical debug functions such as checking the state of the CPU's registers and CSRs, verifying that variables in memory contain expected values, and observing program flow by single stepping the code following the breakpoint.

Lattice RISC-V implementations currently support two types of breakpoints: software breakpoints and hardware breakpoints. Both types of breakpoints halt program execution when reaching a particular location in the instruction memory. However, the approaches are different for the two types of breakpoints.

5.2.1. Hardware Breakpoints

A hardware breakpoint uses a comparator to test the value of the program counter against a value that debugger utility programs into the debug logic. When the PC matches the hardware breakpoint address, control of CPU execution is transferred to the debug core.

Execution resumes only when the debugger utility directs the CPU debug core to exit the debug state. The instruction that resides at the address of the hardware breakpoint is executed upon return from the debug state, not prior to entering it.

Lattice RISC-V CPUs support two hardware breakpoints.

5.2.2. Software Breakpoints

The RISC-V ISA defines a special instruction, EBREAK, that passes control to the debugging environment. The debugger uses the EBREAK instruction to implement software breakpoints.

When a developer sets a software breakpoint, the debugger utility identifies the address in program memory that corresponds to the targeted line of source code. The debugger utility replaces the instruction at that target address with the EBREAK instruction or the 16-bit equivalent, C.EBREAK, depending on the size of the original instruction.

When the program execution encounters the EBREAK instruction, control is transferred to the CPU debug module which halts the program execution. The developer can examine or modify the system state according to their debug strategy.

When the developer commands the debugger to resume program execution, the debugger performs the following operations:

1. Replace the EBREAK instruction with the original instruction.
2. Transfer control to the CPU to execute the original instruction.
3. Rewrite the EBREAK instruction back to the location of the software breakpoint.
4. Resume normal program execution.

The RISC-V CPUs support unlimited software breakpoints as software breakpoints do not rely on finite hardware resources.

However, when using software breakpoints, the debugger must be able to modify instruction memory at the address of the targeted instruction. If the debugger is unable to modify the instruction memory, the design can only use hardware breakpoints. For example, a design that executes instructions from SPI flash via Execute in Place (XiP) feature must use hardware breakpoints as the debugger is unable to dynamically alter the SPI flash memory. Similarly, the Physical Memory Protection unit (PMP) for RISC-V RX could be configured to block write access to instruction memory following startup.

5.3. Using Semihosting

Semihosting is a mechanism that enables code running on the target to communicate with and use the I/O of the host computer. This method is useful during the development stage to output messages to the debug console without using the UART interface. Semihosting operates by stopping the processor execution and transferring the data from target to the host. Hence, semihosting generally does not provide high performance.

Lattice Propel SDK provides semihosting support. You can enable semihosting during project creation or changing the project property after project creation.

5.3.1. Enabling Semihosting During Project Creation in Lattice Propel SDK

Select Semihosting (--oslib=semihost) under System Library when creating a new C/C++ project in the Lattice Propel SDK as shown in Figure 5.2.

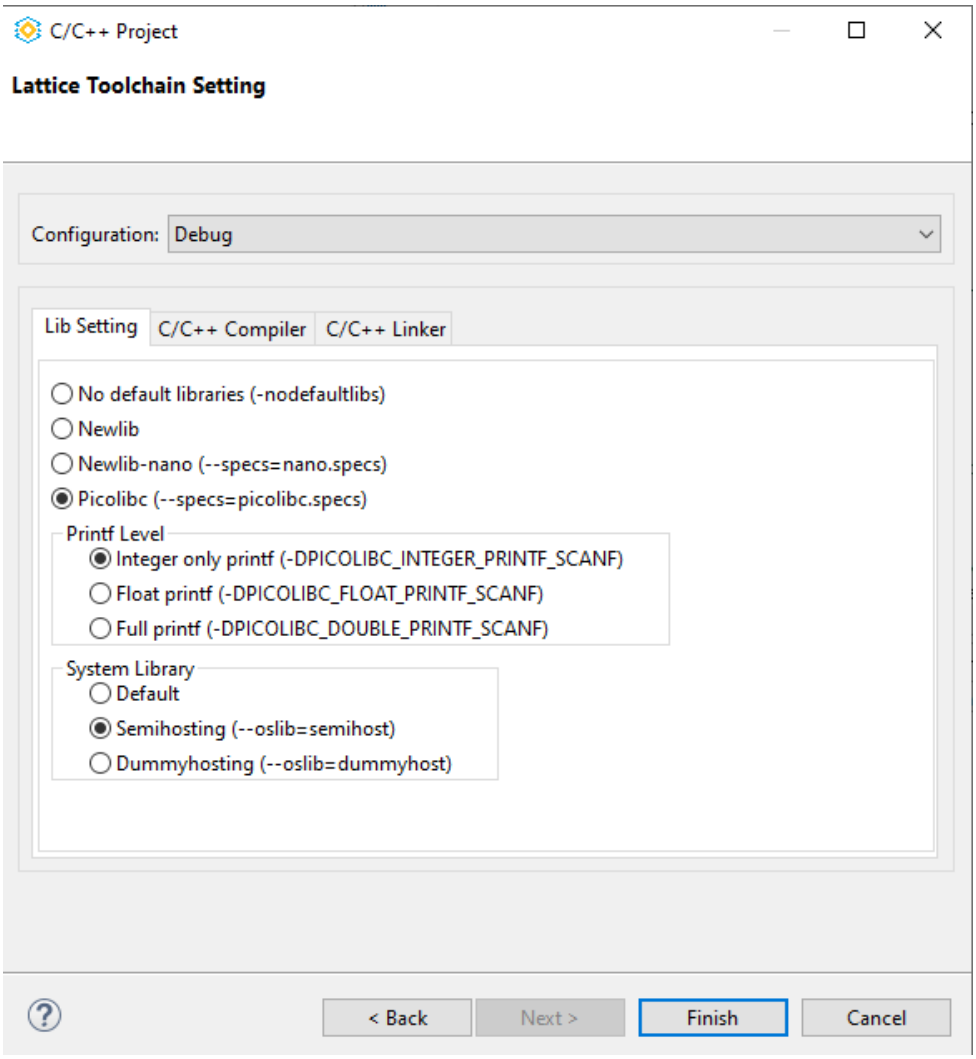

Figure 5.2. System Library Settings During C/C++ Project Creation

5.3.2. Enable Semihosting After Project Creation

If semihosting is not selected during project creation, you can change the project property to enable semihosting as follows:

1. Select the project in the Lattice Propel Project Explorer window.
2. Click **File > Properties**.
3. Select **C/C++ Build > Settings > GNU RISC-V Cross C Linker > Miscellaneous**.
4. In **Other linker flags**, type **--oslib=semihost**.
5. Click **Apply and Close**.
6. Click **Project > Build Project**.

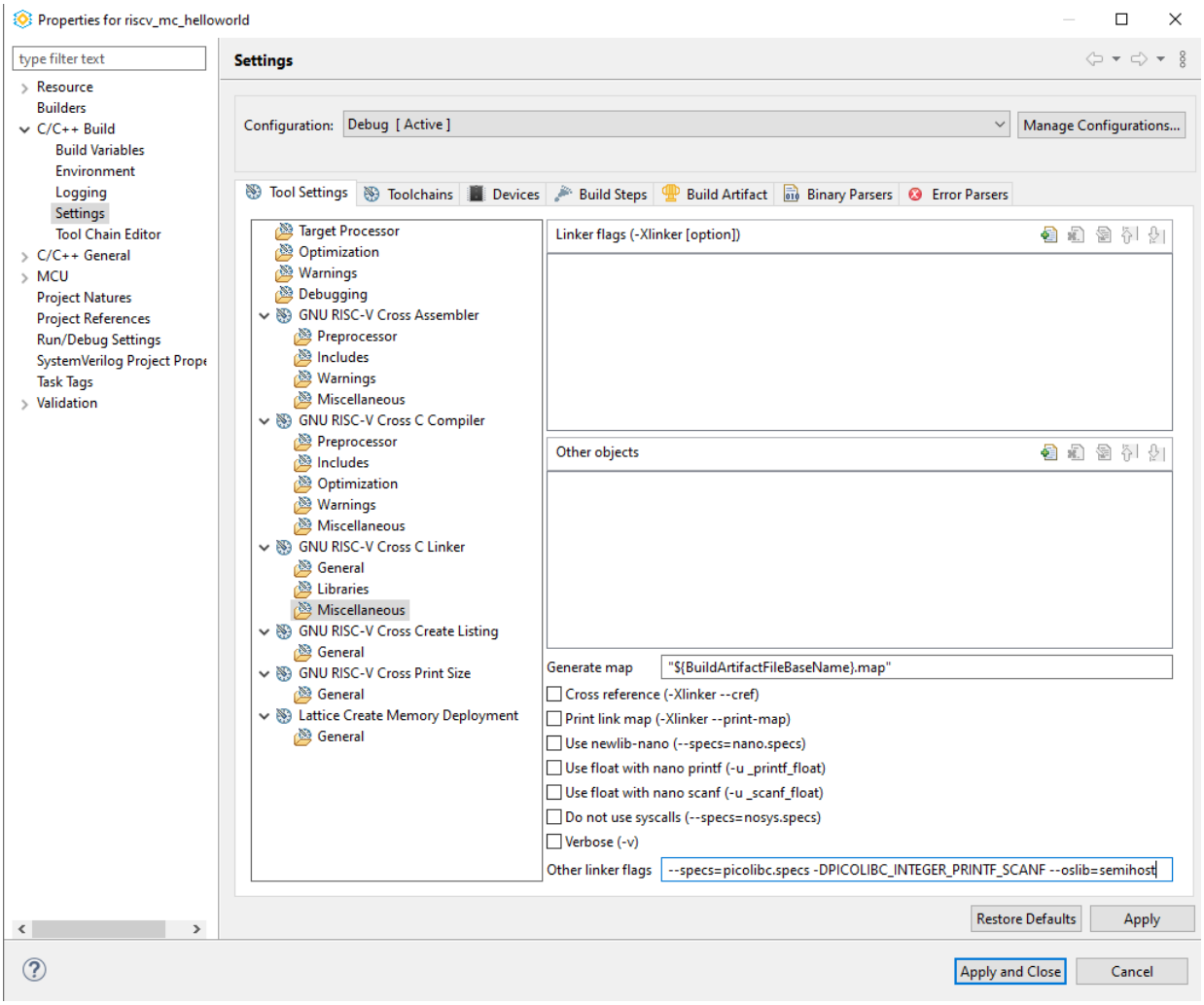

Figure 5.3. Enable Semihosting After Project Creation

Semihosting supports file I/O access to the host using fopen, fwrite, and fread functions. To enable this feature, set the heap size to non-zero in your project linker script as follows:

1. Expand the project in Project Explorer and locate the linker script in **src > linker.ld**.
2. Double click linker.ld.
3. Change the **HEAP_SIZE** value. You can set the value to 0x800 as shown in the example in Figure 5.4.

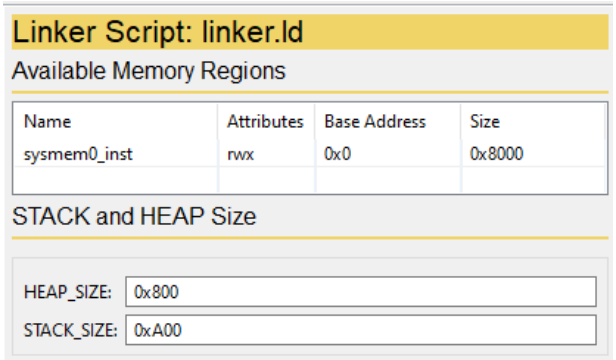

Figure 5.4. Changing Linker Script Heap Size

4. Write code in application that exercises fopen, fwrite, or fread.
5. Clean and rebuild the project.

5.4. Setting UART Serial Interface

Serial interface such as UART is a debug technique for printing messages to the console.

Most Lattice development boards have a component that converts the USB interface to UART. This conversion allows the board to send or receive UART signals to computer over the USB.

Figure 5.5 shows the Certus-Pro NX Evaluation Board which has the FTDI2232H chip for USB to UART interface. The UART signals are connected to the BDBUS channel of the chip, and shares the channel with the I2C interface. Install jumpers on the board to connect the UART signals to the FTDI2232H chip.

When assigning FPGA pins to the UART controller in your Lattice Propel Builder system, assign the UART IP TXD signal to the RXD pin and IP RXD signal to the TXD pin of the FTDI2232H chip.

When opening a Terminal program on computer (for example Putty or TeraTerm), the program may show 2 COM ports instead of 1. As there are 2 channels on the FTDI2232H chip, select the COM port that corresponds to the number of channels on the chip. The larger number COM port represents the UART interface.

Refer to the development board user guide and schematic when performing pin assignments and using the UART interface.

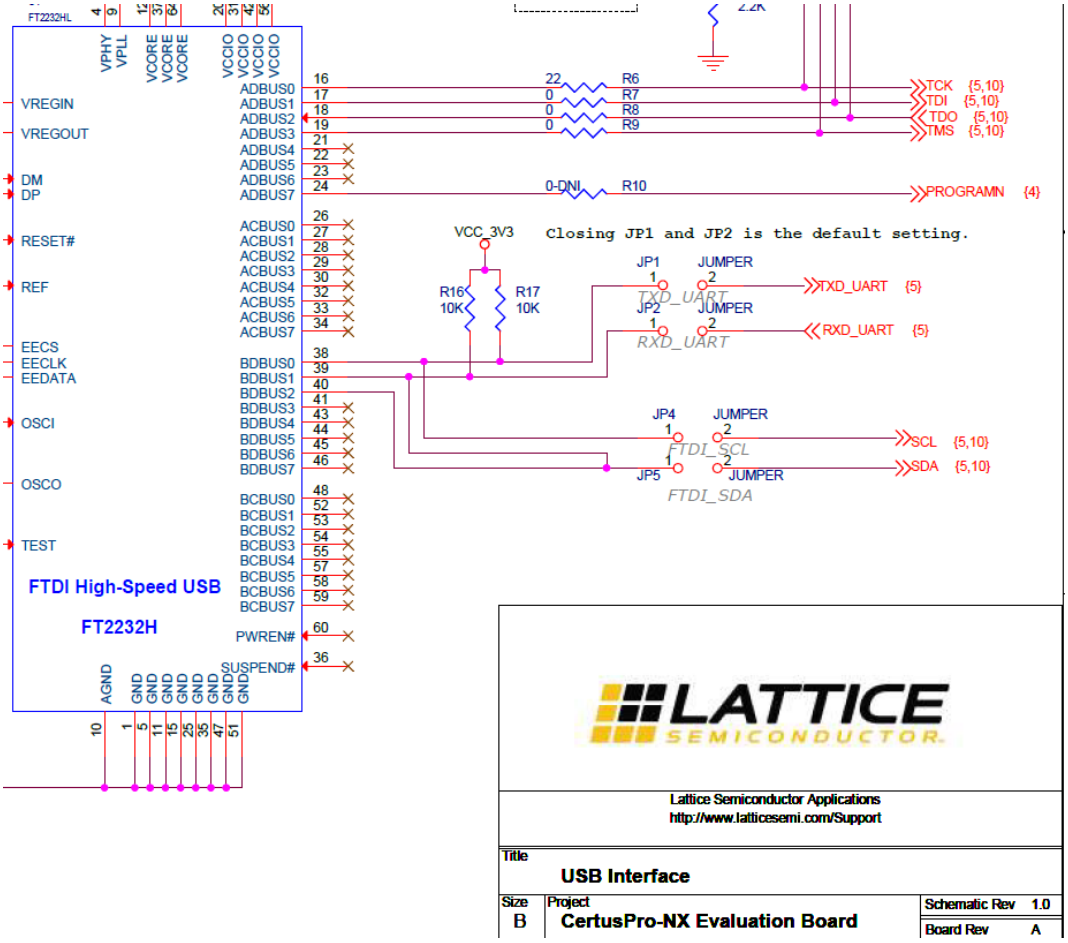

Figure 5.5. CertusPro-NX Evaluation Board UART Interface Schematic

References

- CrossLink-NX web page
- Certus-NX web page
- CertusPro-NX web page
- MachXO2 web page
- MachXO3 web page
- MachXO3D web page
- MachXO5-NX web page
- Avant-E web page
- Avant-G web page
- Avant-X web page
- Lattice Nexus Platform web page
- RISC-V RX and LPDDR4 Memory Controller web page
- FreeRTOS web page
- GCC, the GNU Compiler Collection web page
- RISC-V Platform-Level Interrupt Controller Specification web page
- Lattice Propel Design Environment web page
- Lattice Radiant Software web page
- Lattice Diamond Software web page
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, February 2024

Section	Change Summary
All	Production release.

www.latticesemi.com